

## Introduction to Machine Learning and Deep Learning

### Introduction to Machine Learning and Deep Learning: A Hands-On Starter's Guide

Samuel Dodge and Lina Karam

Copyright 2017-2020 Samuel Dodge and Lina Karam

[Table of Content](#)

## Conditions for Use and Distribution

**Conditions of Use and Distribution** for "Introduction to Machine Learning and Deep Learning: A Hands-On Starter's Guide," Samuel Dodge and Lina Karam follow below:

1. Authorship and copyright should be as follows:

Samuel Dodge and Lina Karam

Copyright 2017-2023 Samuel Dodge and Lina Karam

2. Any redistribution and use of the content with or without modification including but not limited to derivatives of the work and code, must retain the above authorship and copyright note and should cite the following:

Samuel Dodge and Lina Karam, Introduction to Machine Learning and Deep Learning: A Hands-On Starter's Guide, Copyright 2017-2019 Samuel Dodge and Lina Karam. Available online at [deeplearningtextbook.org](https://deeplearningtextbook.org)

**Citing this book in Latex:**

```
@book{DodgeAndKaram-2019,  
  title={Introduction to Machine Learning and Deep Learning: A Hands-On  
  Starter's Guide},  
  author = {Samuel Dodge and Lina Karam},  
  publisher = {},  
  note = {first version published 2017 and revised 2019. Available online at  
  \url{https://www.deeplearningtextbook.org}},  
  year = {2017-2019}  
}
```

3. For educational use only and derivative work cannot be republished and/or resold

---

A Hands-On Starter's Guide: Table of Content

---

## Introduction to Machine Learning and Deep Learning: A Hands-On Starter's Guide

To obtain the DATA and CODE FILES, PLEASE CONTACT US.

[Contact Us](#)

## Title Page and Table of Content

[View](#)

## 1. Python Basics, Unit Testing, and Introduction to Image Processing with Python

[View](#)

## 2. Machine Learning Basics

[View](#)

## 3. A Neural Network from Scratch and Application to Image Classification

[View](#)

## 4. A Convolutional Neural Network from Scratch and Application to Image Classification

[View](#)

## 5. Transfer Learning and Computer Vision in PyTorch

[View](#)

## 6. Natural Language Processing with Recurrent Neural Networks in PyTorch and Tensorflow

[View](#)

## 7. Bibliography

[View](#)

## Contact Us

This site is protected by reCAPTCHA and the Google [Privacy Policy](#) and [Terms of Service](#) apply.

## Introduction to Deep Learning and Machine Learning

Copyright ©2017- 2023 S. Dodge and L. Karam - All Rights Reserved.

Powered by [GoDaddy](#) Website Builder

# **Introduction to Machine Learning and Deep Learning: A Hands-On Starter's Guide**

Samuel Dodge and Lina Karam

# Contents

|                                                            |            |
|------------------------------------------------------------|------------|
| <b>Website</b>                                             | <b>iii</b> |
| <b>Preface</b>                                             | <b>iv</b>  |
| <b>1 Python Basics, Unit Testing, and Image Processing</b> | <b>1</b>   |
| 1.1 Objectives                                             | 1          |
| 1.2 Setup Ubuntu Linux environment using VirtualBox        | 1          |
| 1.3 Installing Python Packages                             | 4          |
| 1.4 Python and NumPy fundamentals                          | 5          |
| 1.4.1 Task 1: Read Python Tutorial                         | 5          |
| 1.4.2 Task 2: Running code and Debugging in PyCharm        | 6          |
| 1.4.3 Task 3: Introduction to Unit Testing in Python       | 10         |
| 1.5 Application: An Image Container Class                  | 13         |
| 1.6 Submission Instructions                                | 20         |
| <b>2 Machine Learning Basics</b>                           | <b>21</b>  |
| 2.1 Objectives                                             | 21         |
| 2.2 Brief Machine Learning Introduction                    | 21         |
| 2.3 Least Squares and Model Capacity                       | 22         |
| 2.4 Implement K-Nearest Neighbors Model (KNN)              | 24         |
| 2.5 Implement Logistic Regression                          | 27         |
| 2.6 Implementing the Support Vector Machine                | 31         |
| 2.7 Using Scikit-Learn Models                              | 32         |
| 2.8 Submission Instructions                                | 35         |
| <b>3 A Neural Network from Scratch</b>                     | <b>36</b>  |
| 3.1 Objectives                                             | 36         |
| 3.2 Understanding Backpropagation                          | 36         |
| 3.3 Implementing Layers of a Neural Network                | 38         |
| 3.3.1 Fully Connected Layer                                | 40         |
| 3.3.2 ReLu Layer                                           | 42         |
| 3.3.3 SoftMax Layer                                        | 42         |
| 3.3.4 Cross Entropy                                        | 43         |
| 3.4 Putting it all together                                | 44         |
| 3.4.1 Implement the Sequential model                       | 44         |
| 3.4.2 Training the model with CIFAR100 data                | 45         |
| 3.5 Submission Instructions                                | 48         |

## CONTENTS

---

|          |                                                                                 |           |
|----------|---------------------------------------------------------------------------------|-----------|
| <b>4</b> | <b>A Convolutional Neural Network from Scratch</b>                              | <b>49</b> |
| 4.1      | Objectives                                                                      | 49        |
| 4.2      | Convolutional Layer                                                             | 49        |
| 4.2.1    | Data and Dimensions                                                             | 50        |
| 4.2.2    | Forward pass                                                                    | 51        |
| 4.2.3    | Backwards Pass                                                                  | 53        |
| 4.2.4    | Initialization                                                                  | 54        |
| 4.2.5    | Update parameters                                                               | 54        |
| 4.3      | Max Pooling Layer                                                               | 55        |
| 4.4      | Flatten Layer                                                                   | 56        |
| 4.5      | Putting it all together again                                                   | 57        |
| 4.6      | Submission Instructions                                                         | 58        |
| <b>5</b> | <b>Transfer Learning and Computer Vision in PyTorch</b>                         | <b>59</b> |
| 5.1      | PyTorch                                                                         | 59        |
| 5.1.1    | Installation                                                                    | 59        |
| 5.1.2    | Tutorial                                                                        | 59        |
| 5.2      | CIFAR100 Example in PyTorch                                                     | 60        |
| 5.3      | Transfer Learning                                                               | 62        |
| 5.3.1    | Datasets                                                                        | 63        |
| 5.3.2    | Base Networks                                                                   | 63        |
| 5.3.3    | Pre-processing                                                                  | 65        |
| 5.3.4    | Feature Extractor + SVM                                                         | 66        |
| 5.3.5    | Fine-tuning                                                                     | 67        |
| <b>6</b> | <b>Natural Language Processing with Recurrent Neural Networks in Tensorflow</b> | <b>70</b> |
| 6.1      | Tensorflow                                                                      | 70        |
| 6.2      | CIFAR100 in Tensorflow                                                          | 71        |
| 6.3      | RNN Background                                                                  | 75        |
| 6.3.1    | RNN                                                                             | 76        |
| 6.3.2    | LSTM                                                                            | 76        |
| 6.3.3    | GRU                                                                             | 77        |
| 6.4      | RNNs for sentence classification                                                | 77        |
| 6.4.1    | Datasets                                                                        | 77        |
| 6.4.2    | Network Design                                                                  | 78        |

# Website

[www.deeplearningtextbook.org](http://www.deeplearningtextbook.org)

# Chapter 1

## Python Basics, Unit Testing, and Image Processing

### 1.1 Objectives

- Setup Ubuntu Linux using the VirtualBox Virtual Environment
- Setup Python and basic Python packages
- Learn Python and Numpy fundamentals
- Implement the `ImageClass` class to practice using Python and Numpy

### 1.2 Setup Ubuntu Linux environment using VirtualBox

We will be using Ubuntu Linux. Although it is possible to use Windows or macOS to train deep models, library support can be poor, especially for deep learning. Additionally, most deep learning development and research is done on Linux machines, even if the product is eventually to be deployed on other operating systems. Ubuntu Linux is a popular distribution of Linux which can run the latest versions of popular machine learning libraries.

We will be using a Virtual Machine (VM) to install and work in a Linux environment. The VM will run as an application in the “host” OS on your computer, so we do not need to dual-boot or partition the hard drive to run Ubuntu. Additionally by using a VM, we ensure that everyone has an identical development environment to work with Python. Another advantage of the VM is that it is easy to restart with a “fresh” Ubuntu installation if needed.

#### Extra Information: Other ways to run Python

It is possible to run Python and most Python packages in Windows or macOS. However, we will not support any problems that may occur when installing Python on these operating systems. We recommend installing Ubuntu Linux. Ubuntu Linux instances can also later be used on the Amazon Web Services (AWS) Cloud to run deep learning models using GPUs. It is necessary that you become familiar with the Linux operating system.

It is also possible to dual boot Ubuntu Linux with another operating system on your personal PC. We will not support any problems with dual-boot installation for the hands-on assignments in this book. If you do try to dual boot, it is recommended that all of the data is backed up before installation.

We will be using Oracle VirtualBox to run the virtual machine. This is free software that is available for Windows, macOS, and Linux. First we need to download the VirtualBox package and the Ubuntu installation disk ISO file. For Ubuntu we will be using version 18.04 LTS, which is the latest stable release. For VirtualBox we will be using the latest version. Please download the two files at the following links:

1. <https://www.virtualbox.org/wiki/Downloads>
2. <https://www.ubuntu.com/download/desktop>

Install VirtualBox and remember the location of the Ubuntu installation ISO file that you just downloaded.

**Important: Disk Space**

Your computer will need approximately 20GB of space. We are installing Ubuntu, several large Python packages, and in the future we will need space for datasets. Do not try to make the VirtualBox hard drive size smaller than 20GB, as in the future you may run out of space.

**VirtualBox Setup** Open VirtualBox and create a new virtual machine by pressing the **New** button. Name your virtual machine “Ubuntu”, and VirtualBox will automatically infer that you want to install 64-bit Ubuntu Linux (Figure 1.1).

VirtualBox will ask how much memory to allocate to the VM. If possible, we recommend to give the Virtual Machine at least 4096MB. However, if your total computer memory is small you should leave enough memory for the host OS to run. Ubuntu will run with less than 4096MB, but performance may be sluggish. The memory allocation can be changed later in the VirtualBox settings.

Next VirtualBox will ask about the virtual disk. Select **Create a virtual hard disk now** and click **Continue**. Create a VDI (Virtual Box Disk Image) that is **Dynamically allocated**. Dynamic allocation means that the virtual disk will automatically grow and shrink as the data in the virtual machine changes. However, the disk cannot easily extend beyond the maximum allocation. So set the maximum allocation to 20GB. This space will be enough for the operating system, the Python libraries, and some data for experiments.

Now that the VM is setup, we need to configure it to launch from the Ubuntu ISO file that we just downloaded so we can install Ubuntu. To do this click on **Settings** and go to **Storage**. Under **Controller: IDE**, select **Empty**. On the right side, click on the disk icon  and select **Choose Virtual Optical Disk File...** select the Ubuntu ISO file that we downloaded (Figure 1.2). Click **OK** to finish this setup.

**Installing Ubuntu** Now we can start the VM and install Ubuntu. To do this, click the green **Start** button. Once the installation disk boots click **Install Ubuntu**. Follow the instructions for a basic install. There is no need to install third-party software or download updates while installing.

During installation choose **Erase Disk and install Ubuntu**. This will not erase the local disk of your computer. This will only erase the virtual disk image used by the virtual machine (which is currently empty). Later you will be asked to setup a username and password. Remember these because we will need it later to install software.

When the installation is complete you will be asked to restart the machine. After restarting, Ubuntu will ask you to remove the installation media. VirtualBox should automatically unmount the Ubuntu install ISO file after installation, so just press enter. Ubuntu should boot to the login screen.

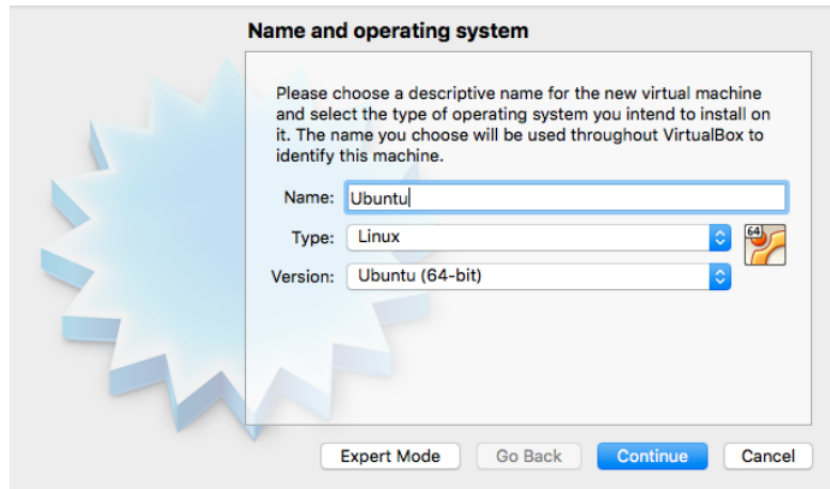


Figure 1.1: Starting a new virtual machine.

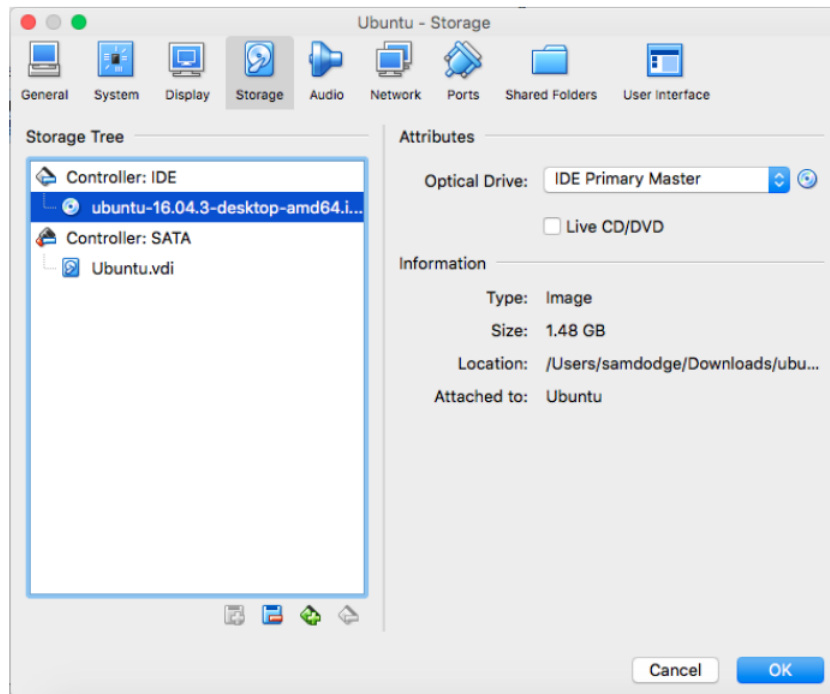


Figure 1.2: Selecting startup disk.

**Extra Information: Using a shared folder**

Right now the file systems of our host and our VM are completely separate. We can keep the file systems this way and work on all of our projects locally with the VM. Ubuntu comes with Firefox pre-installed, so we can download and upload files from this VM. Additionally, Dropbox is available for Linux, which offers an easy way to share files between the host OS and Ubuntu.

Another way is to setup a “shared” folder between the host OS and the VM. The shared folder can be seen and edited by both operating systems at the same time. To enable this functionality, we need to install **VirtualBox Guest Additions**.

To install the guest additions, while **VM running machine** is active (i.e., click on running VM window and notice that VirtualBox menu bar on top includes now **Devices**) from the VirtualBox menu bar select **Devices->Insert Guest Additions CD...** In the Ubuntu VM, you will be prompted to run a program from the inserted CD image. Click run and wait for the software to finish installing. After installation reboot the VM.

After the additions are installed we can setup the shared folder. From the VirtualBox menu bar, go to **Machine->Settings->Shared Folders** and click the  icon to add a new shared folder. Select the folder path you want to share, and select a name that you will remember. Be sure that **Read Only** is not selected, and select **Make Permanent**. Now we can return to the VM and mount the shared folder somewhere in the VM file system. Open a Terminal window (**ctrl+alt+t**) and type the commands shown below to mount the folder while replacing **FOLDER\_NAME** with the name you defined in VirtualBox.

```
mkdir ~/share
sudo mount -t vboxsf -o uid=1000,gid=1000 FOLDER_NAME ~/share/
```

At this point, you can copy the files provided to you for this chapter in your shared folder for these to appear in **~/share** on the Linux virtual machine.

*Note: At the shell prompt, one can type **id -u username** to get the value of **uid** and one can type **id -g username** to get the value of **gid**.*

### 1.3 Installing Python Packages

In Ubuntu, open a new terminal shell by typing **ctrl+alt+t** (or **ctrl+option+t** on a MAC keyboard). The terminal shell allows us to access the underlying Linux programs. We can launch programs, navigate the file system, edit files, etc. Table 1.1 shows a list of useful commands while working in the terminal. To run any of these commands, type the command at the prompt in terminal.

To manage Python packages we use the **pip** (Python Packaging Index) program. The Python Packaging Index contains many useful Python packages and allows us to install, remove, and upgrade those Python packages. To install **pip** we use the Ubuntu package manager program **apt-get**. Install **pip** using the following command:

```
sudo apt-get install python-pip
```

**sudo** runs the **apt-get** command in super user mode. This is analogous to **Run as administrator...** in Windows operating systems. **sudo** is often needed when installing programs because the programs will be installed system-wide.

**Table 1.1:** Useful Linux commands

| Command | Description                            | Example                    |
|---------|----------------------------------------|----------------------------|
| ls      | List files in current directory        |                            |
| cd      | Change directory                       | cd new_directory           |
| cp      | Copy files                             | cp old_file.py new_file.py |
| rm      | Remove files (warning no undo!)        | rm file.py                 |
| mv      | Move files                             | mv old_name.py new_name.py |
| mkdir   | make new directory                     | mkdir new_dir              |
| ssh     | Open SSH connection to server          | ssh username@server        |
| sudo    | Execute command as super user          | sudo command               |
| pip     | Install python packages                | pip install package        |
| python  | Run python script                      | python script.py           |
| apt-get | Package management for Ubuntu packages | apt-get install package    |
| man     | List command description and arguments | man command                |

`apt-get` is a Linux command-line tool for handling packages. At the terminal prompt, type `apt-get -h` to get more information about this command.

With `pip` installed, we can use the command `pip install` to install needed Python packages. Type `pip -h` for more information about `pip`. Run the following commands to install all of the packages we need for this lab. This may take a moment to complete. We also install `python-tk` using `apt-get`, which is necessary for plotting results to the screen.

```
sudo pip install numpy scipy matplotlib imageio
sudo apt-get install python-tk
```

The Numpy package includes all of the basic matrix operations we will need. Think of Numpy as MATLAB-like operations for Python. Scipy extends on Numpy with added scientific computation functionality including but not limited to integration, differentiation, and gradient-based optimization. Matplotlib is a MATLAB-style plotting library. Finally, Imageio is used to load and save images. In later chapters we will again use `pip` to install deep learning libraries.

## 1.4 Python and NumPy fundamentals

We will be using the Python programming language. Python has become a popular language because it is open source, easy to learn, and has support for many popular libraries. There are two popular branches of Python: Python 2.X and Python 3.X. We will be using Python 2.X because of better compatibility with some older packages.

The main strength of Python for machine learning and data science is the abundance of dependable libraries. There are packages for scientific computing (`Numpy` and `Scipy`), machine learning (`Scikit-learn`), image processing (`Scikit-image`), computer vision (`OpenCV`), data visualization (`Matplotlib`) and deep learning (`Pytorch`, `Tensorflow`, `MxNet`, etc.). Most of these libraries can be installed with a single `pip` command, which makes setup very easy.

### 1.4.1 Task 1: Read Python Tutorial

Python documentation and tutorials are available for various Python versions at: <https://docs.python.org/>.

Type `python --version` on the terminal command prompt to see the current installed python version. Please read the tutorial for the installed python version. Type `help()` for interactive help. Then enter a keyword to get help on that keyword. When done, type `quit` to exist the interactive help.

Please go through the following Python and NumPy tutorials:

A Byte of Python available at <https://python.swaroopch.com/>

NumPy Quick Start Guide at <https://numpy.org/devdocs/user/quickstart.html>

NumPy Reference at <https://numpy.org/devdocs/reference/index.html#reference>

These tutorials are useful as references for the rest of this chapter. It is also useful to consult the Numpy documentation [1], Scipy documentation [2], and PyPlot documentation [3].

#### Extra Information: Coming from MATLAB

If you are familiar with MATLAB, the following link is a cheat sheet that shows equivalent functions in Python/NumPy:

<http://mathesaurus.sourceforge.net/matlab-numpy.html>

### 1.4.2 Task 2: Running code and Debugging in PyCharm

In this class we use the PyCharm IDE for editing and debugging code. PyCharm provides useful features such as auto-completion, versioning integration, visual debugging, etc.

**Installing PyCharm** We will use the `snap` command to install PyCharm. The `snap` command works similarly to `apt-get`:

```
sudo snap install pycharm-community --classic
```

We are installing the `pycharm-community` package instead of the `pycharm-professional` package. The `pycharm-professional` package is non-free software that has a limited evaluation period. However the free `pycharm-community` package has everything needed for the hands-on assignments in this book.

After installation, PyCharm can be started by typing `pycharm-community` in a terminal window, or by using Ubuntu's search button and searching for "PyCharm".

**Using PyCharm** First lets open PyCharm. When you start PyCharm select **Do not import settings**. Next click the button **Skip Remaining and Set Defaults**. We have provided the file `pycharm_settings.jar` with settings for the built in Python interpreter. As mentioned previously, one way to move this file to the running virtual machine is to copy it to the "shared" folder on your local machine; this will make it available in the `~/share` folder of the virtual machine. On the splash screen select **Configure->Import Settings**. Navigate to the included `pycharm_settings.jar` file (available in `~/share` after being copied to the "shared" folder). Hit ok to import settings, and ok to restart PyCharm.

Now in the PyCharm launch screen, select **Open** and navigate to the `hello.project` folder. We see that the default PyCharm layout has our files in the left pane; the code will appear in the center pane by double clicking on `hello.py`, and at the bottom there is a pane which we will use later for debugging our code (Figure 1.3). In the left pane our project contains a single file called `hello.py`.

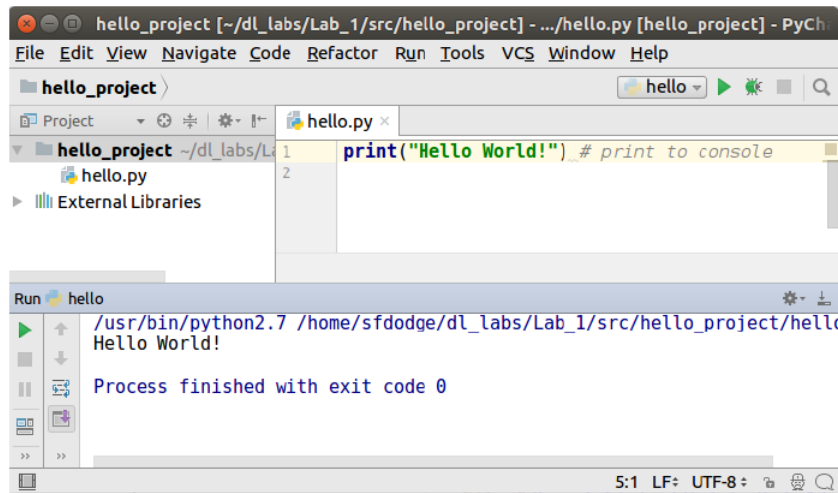


Figure 1.3: PyCharm layout.

**Running scripts in PyCharm** In PyCharm open the script `hello.py` by double clicking on its name in the left panel. Right click on the `hello.py` file in the left pane, and select run (Figure 1.4). After running, in the bottom panel the program will output: “Hello World!”.

We can also run python scripts in the terminal. Open a new terminal (`ctrl + alt + t`) and navigate to the folder that contains `hello.py`. We can navigate the file system using the change directory command (`cd directory-name`). In the `hello-project` folder, type `python hello.py` to see the output of the script (Figure 1.5). Alternatively, you can type `python -m hello` to run the script.

**Debugging in PyCharm** Now add the following code under the print statement:

```
foo = bar / 2
```

Obviously the code shouldn’t run since `bar` has not been defined anywhere. But lets see what happens when we run the code in PyCharm. We see an error message (Figure 1.6) which tells us on which line the error happens and a description of the error. In this case we get: `NameError: name 'bar' is not defined`.

We can fix this error by modifying the code to be:

```
print("Hello World!")
bar = 4
foo = bar / 2
print("Done")
```

Now the code should run with no errors. Since we are not printing `foo` or `bar` to the screen, we don’t know if the variables are actually set properly. We could print the variables to the screen using the `print` function, but this becomes tedious with larger scripts with many variables. Instead, we can set a breakpoint in the code and view the values of the variables. Click to the left of the `print("Done")` statement to add a breakpoint (Figure 1.7).

Now we will run the code in debug mode so that PyCharm will stop at the breakpoint. To run

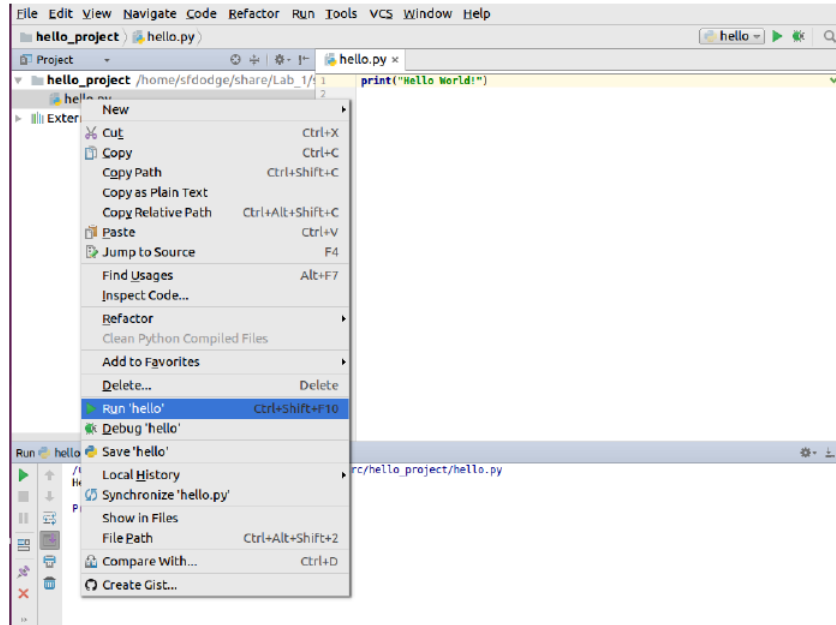


Figure 1.4: Running Python scripts in PyCharm

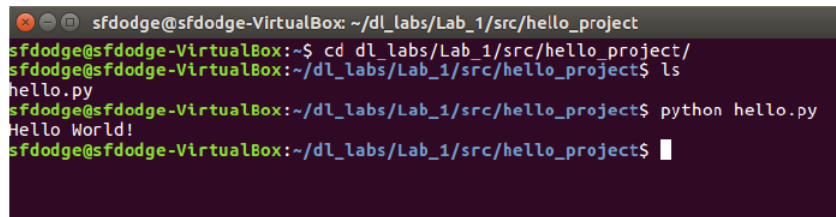


Figure 1.5: Running Python scripts in the terminal

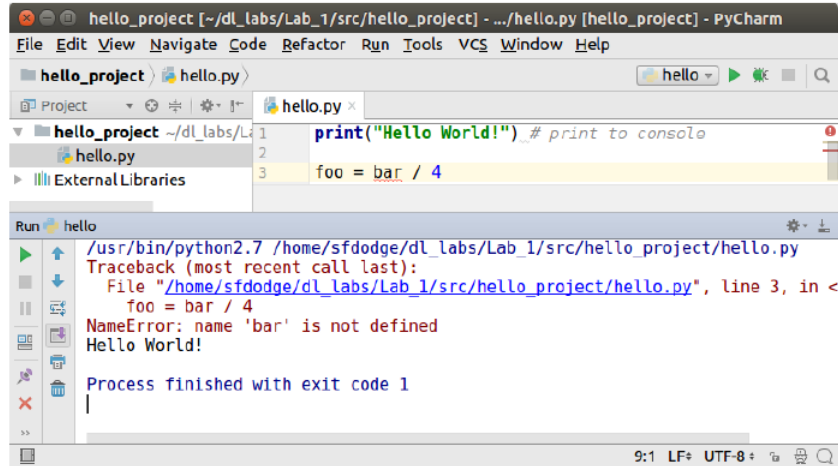


Figure 1.6: Error message

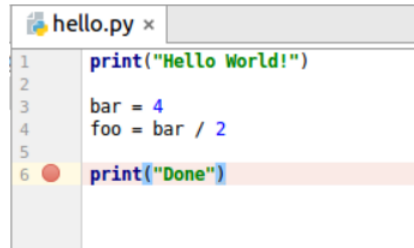


Figure 1.7: Setting a breakpoint

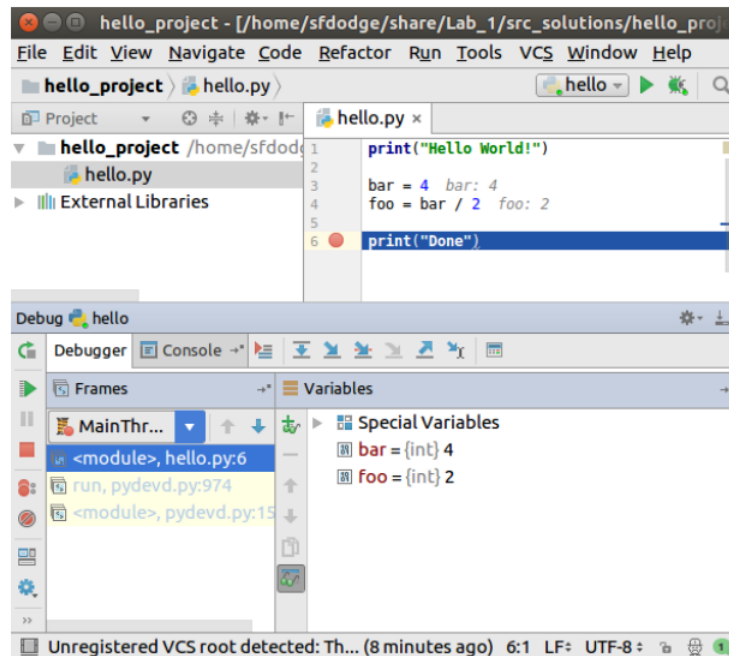


Figure 1.8: Running in Debug Mode

the script in debug mode right click the “hello.py” name in the left pane and select **Debug**. Now the program should halt at the breakpoint, and we can inspect the local variables in the bottom pane (Figure 1.8). You can remove the breakpoint by clicking on the added breakpoint.

#### Extra Information: Other Python tools

PyCharm is not the only tool for debugging/editing Python scripts. Other popular Python editors include Atom, Sublime Text, Vim, Emacs, and Spyder. Debugging can be done directly in the terminal with the `pdb` command. `pdb` behaves very similarly to the popular `gdb` tool for debugging C/C++ code.

### 1.4.3 Task 3: Introduction to Unit Testing in Python

For the hands-on assignments, we will make use of automated unit testing. In unit testing we write functions that test parts of our code. If all of these test functions pass, we have some “proof” that our code works as intended. Automated unit testing is not completely infallible, as we could have a bug in our testing code. However, in general automated unit testing will lead to more correct code.

Without automated testing, we would typically test the function or code we are currently implementing by manually trying some inputs and observing the outputs. After we are convinced our function is working correctly we would move on to some other function, and assume that the first function we implemented still works. However, it is possible that we unintentionally break the first function while implementing some new functionality. In this case we might not realize this until much later, when it will be difficult to determine what is the cause of the failing first function. However if we have automated testing, we can run all the tests after implementing new functionality. If any tests fail, we will know which functions are failing, and for which situations they are failing.

Unit testing means that we test small components (units) of our code independently. When testing the entire functionality of a system it may be difficult to pinpoint exactly where the problem is occurring. But if we test small units, we can determine where the problem lies. As a side benefit, unit testing encourages us to construct our code in well contained functions and classes instead of large monolithic blocks of code. This makes our code easier to read, maintain and reuse.

For the hands-on exercises in this chapter, it may seem tedious to use unit testing, because all the functions implemented are very simple. However in later chapters we will be implementing more complicated functions. For example, we will be implementing the gradient calculation of parameters in a deep neural network. It is possible to derive or implement an incorrect gradient calculation, but we can write a test that calculates the gradient numerically and compare it with our function. If the numeric gradient and our calculated gradient match, then we can be more confident that our function has no mistakes.

Note that it is also sometimes useful to test manually. A simple Python script can be written for this purpose. Manual testing can be used in conjunction with automated testing. However if you find yourself repeatedly testing the same thing manually, it might be useful to write an automated test for it.

We will use a simple project to introduce automated unit testing in Python. In PyCharm open the folder `test_project`. In this folder we have two files `src.py` and `test.py`. The `src.py` file includes the source code and `test.py` includes testing functions that are used to test the correctness of our code.

`src.py` contains the following function that adds two numbers and returns the result:

```
def add_two_numbers(a, b):  
    return a + b
```

`test.py` includes a test to insure that if the function is passed 2 and 3 as inputs, the return value will be 5. For testing we will use the built-in Python package `unittest`. We use the `import` statement to use this package in our Python code. PyCharm is integrated with the `unittest` package, so we can visually inspect the test results.

To create a test case we create a class that is a subclass of `unittest.TestCase`. We can create different test case classes for different parts of our code. This is useful if we don't want to run all of the tests every time. Within the test case class we define several test functions to test different parts of our code. Note that there can be an optional `SetUp` function in the test case class that will be called before each test is run. This can be used to set parameters or initial conditions for the test. A short introduction about classes and inheritance in Python is available at <http://www.jesshamrick.com/2011/05/18/an-introduction-to-classes-and-inheritance-in-python/>.

Our first test case is defined as follows:

```
def test_integer_add(self):  
    self.assertEqual(add_two_numbers(2, 3), 5)
```

This test will “pass” if the output is correct. The `assertEqual` will pass if the arguments are equal, and fail if the arguments are not equal. Note that there are many types of assert statements found in the `unittest` package, such as `assertTrue` and `assertFalse`.

In PyCharm we can run the tests by right-clicking the `test.py` name in the left pane and clicking `Run Unittests...` The tests will run and we can see that the test passed in the bottom pane (Figure 1.9). We can also run the tests using the terminal window by navigating to the folder where the `test.py` file is located and typing `python -m unittest test` at the command prompt. You can get more information while running the tests by using the verbose `-v` flag and typing `python -m unittest -v test`.

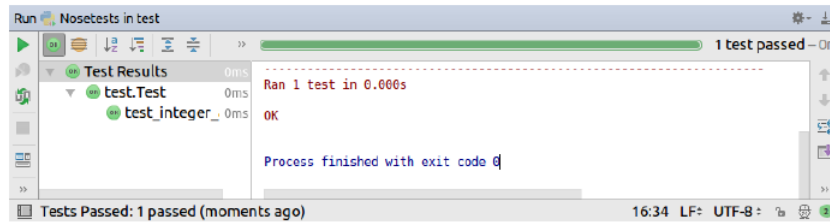


Figure 1.9: Running a passing test in PyCharm.

Modify the `add_two_numbers` in `src.py` to return an incorrect value (e.g. return  $a - b$ ), and run the test again. You should see that the test indeed fails. In the console we can see why the test failed: `AssertionError: -1 != 5`. This says that the output (-1) is different than the expected output (5). After this, revert the `add_two_numbers` function to the original definition.

#### Deliverable: Screenshot of failing test

Take a screenshot of the PyCharm window showing the failing test. The screenshot should include the error message that shows the expected output compared with the output. In Ubuntu screenshots can be taken by using `shift + print screen`, or the screenshot can be taken from the host operating system. Name this screenshot `failing_test.png` and save it in the `test_project` folder.

Next we will test if our function can add a floating point number to an integer number. What does Python do here? Does it know how to cast the integer to a float? Lets add the following test to `test.py` and run the tests again.

```
def test_integer_plus_float(self):
    self.assertEqual(add_two_numbers(1.1, 2), 3.1)
```

After running the updated `test.py`, we see that the new test passes (in addition to the first test). This shows that Python is automatically converting the integer to a floating point. We could further test things like overflow, negative numbers, imaginary numbers, etc. We leave this to the interested reader.

#### Hint: Python indentation

Python uses indentation instead of brackets to separate code blocks. When we add the `test_integer_plus_float` function it needs to be part of the `Test` class. In order for the function to be a member of the class, the indentation must be the same as with the other functions in the class. Use tabs to make sure of this. Additionally, the body of the function must also be indented to be executed properly when the function is called.

#### Deliverable: `src.py` and `test.py`

You must submit your `src.py` (original, unchanged) and `test.py` file with the modifications that we made in this section. Specifically, the `test.py` should contain two test functions.

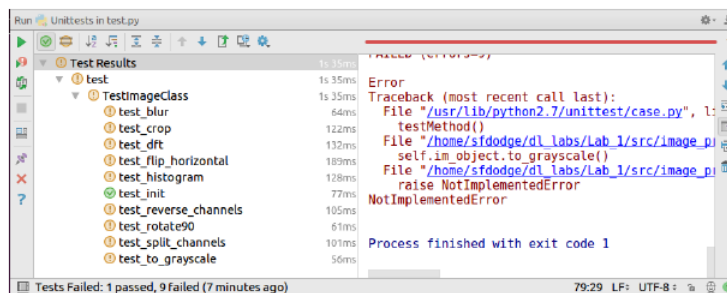


Figure 1.10: Failing tests for ImageClass.

## 1.5 Application: An Image Container Class

In this section we will write a Python class with functions for image manipulations called `ImageClass`. This class will be able to do things like load an image, show the image, plot the histogram, crop the image, etc.

The folder `image.project` has the starting code for this task. Inside this folder there are two files `test.py` and `image-class.py`. In `test.py` we have written unit tests to test the implementation of our new `ImageClass`. If we run the unit tests in `test.py` we see that most of our tests have failed (Figure 1.10).

The functions you need to implement currently have one line: `raise NotImplementedError`. This line forces the test to fail and serves as a reminder that the test is failing because you have not implemented the function. You will need to replace this line with your own code that implements the desired functions. After implementation your code should pass all of the tests.

We have decided to make this a class rather than a collection of functions for two reasons. First it is useful to learn how to program a class, because object oriented programming is very common in Python. Secondly, by using a class we can encapsulate our data without having to pass it back and forth between functions. This lets us use this class to compute several functions on an image very easily. For example the following code uses our class (after implementation) to load an image `sparky.png`, flip it horizontally, crop it, blur the image, and save it to a file. Other images can be loaded by replacing `sparky.png` by another image file name.

```

from image import ImageClass
im = ImageClass('sparky.png')
im.flip_horizontal()
im.crop(100, 100, 50, 50)
im.blur()
im.save('output.png')

```

In `image-class.py`, we have provided the skeleton of the class. We have provided implementations of the `show`, `save` and `__init__` methods. The rest of the functions need to be implemented. Figure 1.11 shows the expected outputs for a sample image for the functions you will implement.

At the beginning of the file we will load the required libraries. Note the `as` keyword which can replace a package name with a shorthand name. In this case we use this to replace `numpy` with `np` as a shorthand, and `matplotlib.pyplot` with `plt`. This means that we can use `np.x()` to call a function from Numpy, and `plt.x()` to call a function from PyPlot.

**Table 1.2:** Useful Numpy functions

| Command                        | Description                                                |
|--------------------------------|------------------------------------------------------------|
| a[0]                           | Take the 0th index of the array a                          |
| a[-2]                          | Take the second from the last index of the array a         |
| a[::-1]                        | Reverse the order of array a                               |
| A[3,5]                         | Take the element at the 3rd row and 5th column of matrix A |
| A[3,:]                         | Take the 3rd row of matrix A                               |
| np.dot(A, B)                   | Dot product of A and B                                     |
| np.concatenate((A, B), axis=0) | Concatenate matrices on an existing axis                   |
| np.stack((A, B), axis=0)       | Concatenate matrices on a new axis                         |
| A = A[:, np.newaxis]           | Add a new axis to array A                                  |
| A * B                          | Elementwise multiplication of matrices A and B             |
| np.zeros((w, h, c))            | Create a matrix of zeros with size $w \times h \times c$   |
| np.ones((w, h, c))             | Create a matrix of ones with size $w \times h \times c$    |
| A.dtype                        | Variable storing the current type of matrix A              |
| A.astype('uint8')              | Convert the matrix A to have type <code>uint8</code>       |

```
import numpy as np # for matrix computations
import imageio # for loading the image
import scipy.signal # for convolution
import matplotlib.pyplot as plt # for plotting functions
```

For this class we will be using many functions from the Numpy package. This package can handle many matrix operations. Images can be thought of as matrices, so the Numpy package can be used for many image processing tasks. Many of the Numpy functions are analogous to similar MATLAB operations, but note that there are some differences (such as matrix multiplication). Table 1.2 lists some common Numpy commands. You can also refer to the Python tutorial from earlier for help [4], or the official Numpy documentation [1].

#### Extra Information: Python Doc-strings

In Python programs it is common practice to comment the beginning of a function with a “doc-string” that describes the function. The doc-string is enclosed by `"""` which denotes multi-line comments in Python. We can easily view the doc-string of any function in PyCharm by pressing `ctrl + q` when the text cursor is over that function.

**ImageClass.\_\_init\_\_** We have provided the initialization function for the class that takes a single argument `path` that specifies the path of the image file to be load. This function loads the image into a variable called `self.im` using the `imageio.imread` function. The `self.im` variable is part of the `self` object which is like a pointer to the current object. This makes this variable a member of the class. In general it is best to initialize member variables to default values in the initialization function.

```
def __init__(self, path):
    """ Loads the image specified by path """
    self.im = imageio.imread(path)
```

**ImageClass.show()** We have provided a function that uses pyplot to show the image to the screen using the `plt.imshow` and `plt.show` functions. These functions will be very useful in future chapters. For the hands-on assignments in this chapter, the `show` function can be used to visually check the output of your code while debugging.

```
def show(self):  
    """ Shows the image using Matplotlib's pyplot """  
    plt.imshow(self.im)  
    plt.show()
```

**ImageClass.save(path)** We have also provided a function that will save the current image to the destination specified by the variable `path`. This function uses the `imageio.imwrite` function.

```
def save(self, path):  
    """ Saves the image to path """  
    imageio.imwrite(path, self.im)
```

**ImageClass.crop(self, r, c, h, w)** You need to implement the remaining functions in the `ImageClass` class. The `crop` function crops the image from row `r` and column `c` with height `h` and width `w`. This function should overwrite the value in `self.im` with the new cropped image. We can crop the image by using Numpy slicing:

<https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html>

**ImageClass.flip\_horizontal()** This function will flip the image horizontally and store the flipped image back in `self.im`. There is a built-in Numpy function that does this (`np.fliplr`), but we can also do this with indexing.

**ImageClass.transpose()** This function should transpose the image and store the image back in `self.im`. We want to only transpose the rows and columns. The `np.transpose` function could be used to do this.

**ImageClass.reverse\_channels()** This function should reverse the order of the color channels. Usually the channels in an RGB image are ordered (R, G, B). This function should change this so that the image channels are ordered (B, G, R). Again the resulting image should overwrite the original image in `self.im`. If you plot an image with these reversed channels the colors should be different (Figure 1.11).

**ImageClass.split\_channels()** This function will split the image by channels and return a list of three matrices: the red channel, the green channel, and the blue channel. This can be accomplished with Numpy indexing. To return multiple variables in Python we can return a list (e.g. `return red, green, blue`).

**ImageClass.to\_grayscale()** To create the grayscale image we combine the channels to create a single channel image and overwrite the `self.im` variable. We can use the previously defined function `split_channels()` to get the individual channels. If we are inside a class function we can call this function using `self.split_channels()`. We use the following formula to combine the channels:

$$I_{gray} = 0.30I_{red} + 0.59I_{green} + 0.11I_{blue} \quad (1.1)$$

where  $I_{red}$ ,  $I_{green}$ ,  $I_{blue}$  represent the original color channels, and  $I_{gray}$  is the desired grayscale output. The weights are determined from the Rec. 601 standard used in PAL and NTSC systems for television [5].

For this function we have to be careful of the type of the data. If the original image is of size  $w \times h \times c$ , the grayscale output should have dimensions  $w \times h$ . The output data type should be the same as the input data type: `uint8` (unsigned 8-bit integer). However, we want to multiply the channels of the image with floating point values. We could first cast the data to floating point using `channel.astype('float64')`. Then we can weight and add the channels. In fact Numpy will automatically cast the variable to `float64` if we multiply a matrix with a floating point constant. However, Numpy does not automatically cast back to `uint8`. We can manually cast back to unsigned 8-bit integer using the command `im.astype('uint8')`. Usually before casting to `uint8` we need to make sure that the data lies between 0 and 255, but in this case equation 1.1 does not change the range of the data so rescaling is not needed.

#### Extra Information:

Note that after implementing `to_grayscale` there may be bugs in our code. For example if we call `to_grayscale` and then call `split_channels`, our code will likely fail because there are no longer three channels in the image. A better implementation could check the number of channels and raise an exception or return only the single channel. For now we won't worry about this in our implementation.

**ImageClass.blur()** This function will blur the image using 2D convolution and store the resulting image in `self.im`. We will use a  $7 \times 7$  averaging filter:

$$\mathbf{Y} = \frac{1}{49} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} ** \mathbf{X} \quad (1.2)$$

where `**` is the 2D convolution operation. Recall that for convolution we flip the filter, slide the filter over the image, and compute the sum of the multiplication of the corresponding filter and image pixels. For this filter, the convolution will replace each pixel with the average of the  $7 \times 7$  pixel region around the pixel.

You will need to define the filter in your code as a Numpy matrix using the `numpy.ones` method. For the actual convolution we will be using the function: `scipy.signal.convolve2d`. We want to perform 'valid' convolution, that is convolution with no padding or symmetric extensions at the edges. The documentation for this function can be found here:

<https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve2d.html>

The `scipy.signal.convolve2d` function works on a single channel image. For this function we want to blur the color image. To do this we can blur each channel separately. The channels can be added back together using the `np.stack` command. Note that the convolution function may change the data type of our result, so again we need to convert the image back to `uint8`.

**ImageClass.plot\_histogram()** We will use pyplot again to plot the histogram. In this case we have a 1-D array that we want to plot. For this function we will need to first convert the image to grayscale. The histogram can then be computed using the `np.histogram` function. We want to create a histogram with 256 bins, covering the full range from 0 to 255. In addition to plotting the

histogram, this function should also **return the computed histogram**. The histogram should be an array of 256 values with each element of the array corresponding to the count of pixels with grayscale value equal to the index of the element. Plotting can be performed with Pyplot's `plt.plot` function.

**ImageClass.compute\_dft()** Next we will compute the DFT of the image. It is possible to directly compute the DFT using the Numpy function `numpy.fft.fft2`. However, we want to practice using the lower level Numpy functions to calculate the DFT ourselves.

The DFT can be computed using a DFT matrix [6]. The DFT matrix is defined as:

$$\mathbf{W} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix} \quad (1.3)$$

where  $N$  is the size of the matrix we want to compute the DFT for, and  $\omega = e^{-2\pi i/N}$ . Using this matrix we can compute the DFT of our image as follows:

$$\mathbf{Y} = \mathbf{W}\mathbf{X}\mathbf{W} \quad (1.4)$$

where  $\mathbf{X}$  is the original image and  $\mathbf{Y}$  is the output of the 2D DFT.

We will implement this using Numpy's `np.dot` function. We have written a function `_DFT_matrix()` that computes the matrix  $\mathbf{W}$  given the size of the image. For this function we will also need to convert the image to grayscale before computing the DFT.

**ImageClass.plot\_save\_dft(self, dft, fname)** This function plots the magnitude of the 2D DFT matrix (such as the one computed using the `compute_dft()` function) and saves the plot as an image in a file whose name and type are specified by `fname`. This function is provided, and you need only to run it to plot a 2D DFT and save the plot. You need to set `fname` to the desired filename.

```
def plot_save_dft(self, dft, fname="dft_plot.png"):
    """
    Plot and save the 2D dft of the image

    Parameters
    -----
    dft      : complex
                2D array contain the dft of an image.
    fname    : string
                The file name to save the 2D dft.
    """
    dft = np.log10(np.abs(dft) + 1)
    dft = np.roll(dft, dft.shape[0]/2, axis=0)
    dft = np.roll(dft, dft.shape[1]/2, axis=1)
    dft = dft - dft.min()
    dft = dft / dft.max()
    plt.imshow(dft, cmap='gray')
    scipy.misc.imsave(fname, dft)
    plt.show()
```

**Hint:**

All of these functions can be implemented in less than 10 lines each. Some functions can be implemented with a single Numpy function call. If one of your functions is becoming very long, there is likely a simple Numpy function that you are missing.

**Important:**

Do not import any other packages. You should be able to implement the entire `ImageClass` using only `numpy`, `matplotlib`, `imageio` and `scipy` packages.

**Deliverable: image\_class.py**

You must implement the following functions in the `image_class.py` file:



| Function                        | Description                                          |
|---------------------------------|------------------------------------------------------|
| <code>crop(r,c,w,h)</code>      | Crop the image                                       |
| <code>flip_horizontal()</code>  | Horizontally flip the image                          |
| <code>transpose()</code>        | Transpose the image                                  |
| <code>reverse_channels()</code> | Reverse the input channels (RGB becomes BGR)         |
| <code>split_channels()</code>   | Split the image by channel and return channels       |
| <code>to_grayscale()</code>     | Convert the color image to grayscale                 |
| <code>blur()</code>             | Blur the image with a $7 \times 7$ averaging filter  |
| <code>plot_histogram()</code>   | Plot the histogram of the image                      |
| <code>compute_dft()</code>      | Compute the DFT using the DFT matrix and dot product |

**Deliverable: image\_blur\_dft.py**

Use the functions in `image_class.py` and write a script to perform the following tasks in the order specified below:

- 
- 1) Load the image `sparky.png` or other image provided to you.
  - 2) Compute the 2D DFT of the input image; generate a plot of the DFT magnitude and save the plot in a file called `orig_img_dft.png`.
  - 3) Blur the image using a  $7 \times 7$  averaging filter.
  - 4) Compute the 2D DFT of the blurred image; generate a plot of the DFT magnitude and save the plot in a file called `blur_img_dft.png`.

We have provided an empty script called `image_blur_dft.py`; please write your code in this script. Run the script and compare the generated DFT plots. What do you observe? Provide your observations as instructed in the commented section at the beginning of the script. Submit the completed `image_blur_dft.py` file along with the two saved images corresponding to the two generated DFT plots.

**Important:**

Do not use functions from other packages. You should be able to implement these tasks using functions from `ImageClass`. There will not be a test unit for this script.

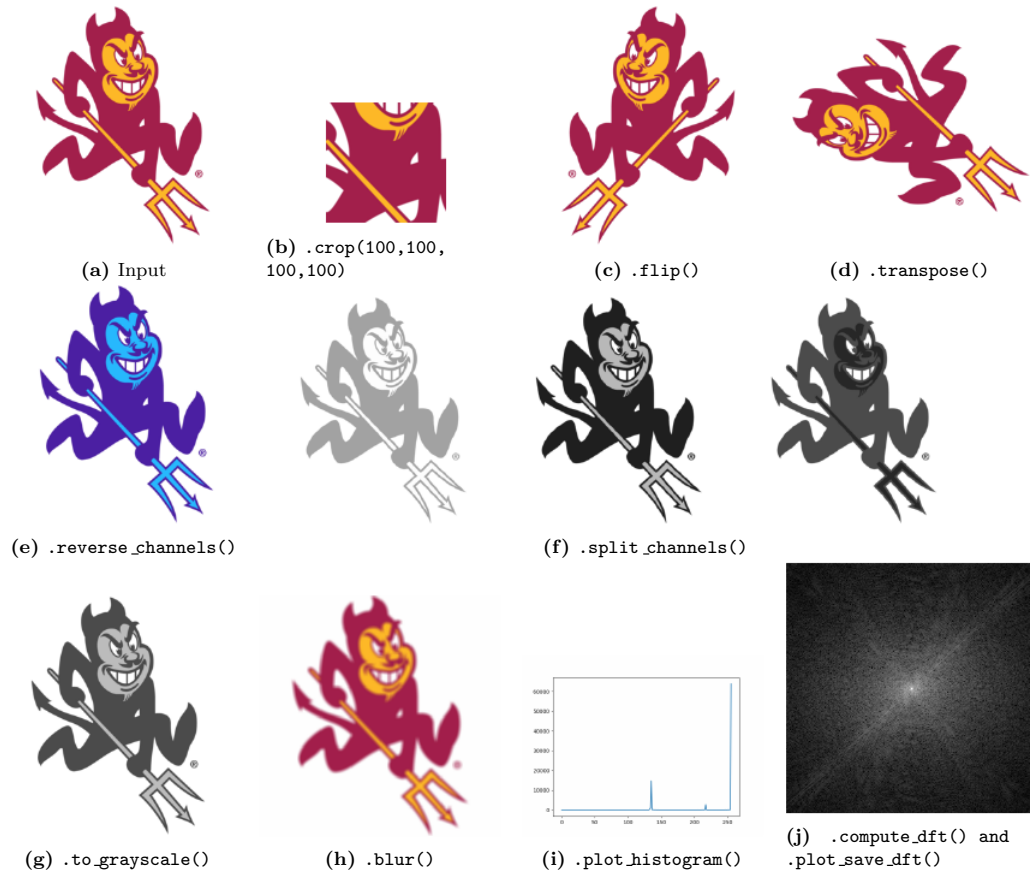


Figure 1.11: Expected outputs of functions implemented in ImageClass

## 1.6 Submission Instructions

We will test your code using unit tests similar to (but not identical to) the unit tests that we used in this chapter. Thus you must be sure that your code works for all cases, not just the particular cases in the given unit tests. Do not change any of the names of the classes or functions. Also do not change the folder structure of the code as this is assumed by the testing code used for grading. You will be graded based on how many tests your code passes.

You must use the same folder structure as in the provided `src` folder. Replace the name of the `src` folder with `FIRSTNAME_LASTNAME_LAB1` and zip the folder for submission.

The following are the grading guidelines we will use for the hands-on exercises in this chapter:

**Table 1.3:** Grading rubric

| Points    | Description                                                           |
|-----------|-----------------------------------------------------------------------|
| 10        | Correct file names and folder structure                               |
| 10        | <code>hello_project</code> folder with modified <code>hello.py</code> |
| 10        | <code>test_project</code> folder with added unit test                 |
| 10        | Screenshot of the failing test                                        |
| 45        | <code>image_project</code> folder with code that passes all tests     |
| 15        | Two DFT figures and script <code>image_blur_dft.py</code>             |
| Total 100 |                                                                       |

# Bibliography

- [1] “Numpy reference.” <https://docs.scipy.org/doc/numpy-1.13.0/reference/>, 2017.
- [2] “Scipy reference.” <https://docs.scipy.org/doc/scipy-1.0.0/reference/>, 2017.
- [3] “Matplotlib pyplot reference.” [https://matplotlib.org/api/pyplot\\_api.html](https://matplotlib.org/api/pyplot_api.html), 2017.
- [4] J. Johnson, “Python numpy tutorial.” <http://cs231n.github.io/python-numpy-tutorial/>, 2017.
- [5] “Luma coding in video systems.” [https://en.wikipedia.org/wiki/Grayscale#Luma\\_coding\\_in\\_video\\_systems](https://en.wikipedia.org/wiki/Grayscale#Luma_coding_in_video_systems), 2017.
- [6] Wikipedia, “DFT matrix — Wikipedia, the free encyclopedia.” <http://en.wikipedia.org/w/index.php?title=DFT%20matrix&oldid=811427639>, 2017.

## Chapter 2

# Machine Learning Basics

### 2.1 Objectives

In this Chapter, several basic machine learning models will be implemented. Specifically we will:

- Implement Least Squares Fitting in Python
- Implement K-Nearest neighbors (KNN) in Python
- Implement Logistic Regression in Python
- Implement Support Vector Machines (SVM) in Python
- Learn how to use Scikit-Learn models

### 2.2 Brief Machine Learning Introduction

A machine learning model takes the general form of  $f(\mathbf{x}; \theta) = \mathbf{y}$ , where  $\mathbf{x}$  contains the input features,  $\mathbf{y}$  contains the corresponding class labels, and  $\theta$  represents the parameters of the model.  $f$  is the machine learning model which could take many forms, from the simple linear classifiers we will implement in this lab, to layered deep networks we will implement in later chapters. The choice of the model is largely problem dependent.  $\mathbf{x}$  contains several data points  $\mathbf{x}_i$  where each data point is a vector of “features”. The features could be some measurements of the data, such as image pixels, physical measurements, word embeddings etc. The choice of features is problem dependent, and for some problems the choice of features is more important than the choice of machine learning model. The label  $\mathbf{y}$  could be a category label, a continuous valued number, or even a vector of numbers. In this chapter,  $\mathbf{y}$  is constrained to be either -1 or 1 (a binary classification problem). The model parameters  $\theta$  must be optimized based on the given training data  $(\mathbf{x}_{train}, \mathbf{y}_{train})$ . After this optimization procedure, we say that the model has been “trained”. We can then test the model on new data  $\mathbf{x}_{test}$ .

For machine learning we must be careful to separate training data and testing data. We give training data to the model to learn parameters, but we never give testing data to the model for this purpose. If testing data is accidentally used for training, model performance may be misleadingly high on the testing data. It is possible for some machine learning models to obtain perfect performance on the training set (K-Nearest neighbors can do this). So if testing data is used for training, the performance on the testing dataset may be perfect, but if new data is collected the performance will not be perfect. The goal of the separate testing set is to get an idea of how the model would work “in the real world” with new, unseen data.

We use training data to optimize the parameters of the network, but there is also the choice of the model  $f$  that we must “optimize”. Even for a single model type there is often one or more parameters of the model. These parameters often cannot be easily optimized with numerical optimization methods. We optimize them “by hand” by selecting several values and seeing what works best, or by using a grid search. We term these parameters “hyper-parameters” to distinguish them from the learnable parameters in the model. However if we use the testing set to optimize the hyper-parameters, the model may have unfairly achieved higher performance on the testing set. It is for this reason that we usually use a completely separate testing set to optimize hyper-parameters. We call this the *validation set*. After choosing good parameters on the validation set, we finally test on the testing set to get a measure of real world performance. No parameters or hyper-parameters have been tuned to the testing set, so it is an unbiased estimate of performance. In this chapter, we will not be doing much hyper-parameter optimization; so, for simplicity, we will only consider a training and testing set. For real world problems, it is recommended to use a validation set as well.

## 2.3 Least Squares and Model Capacity

To start, we will implement least squares fitting of a polynomial model. In this example we consider a regression problem where we want to predict real values  $y$  for a given  $x$ . To do this we will use a polynomial model of the form  $y = c_0 + c_1x + c_2x^2 + c_3x^3 + \dots c_kx^k$ . We want to learn the values of  $c_i$  to achieve the best fit. The key hyper-parameter of this model is the degree of the polynomial  $k$ . With a larger degree, we can model more complicated functions, but we might overfit the data. Conversely, a polynomial degree that is too small will not be able to fit the data well.

Figure 2.1 shows the given data. We have separate training data and testing data that are generated using the same process, but because of noise the training and testing data are not the same. We would like our polynomial function to fit the data well, but without fitting the noise. We will use the training data to fit the coefficients of the polynomial model, and then test the performance of the model on the test data. The data is provided in a file called `ls_data.pkl`. This file stores a dictionary of Numpy vectors using the Python `pickle` package. This data can be loaded with the following code:

```
data = pickle.load(open("ls_data.pkl", "rb"))
x_train = data['x_train']
x_test = data['x_test']
y_train = data['y_train']
y_test = data['y_test']
```

We will implement the least squares fitting model as a Python class called `LeastSquares` in the file `ls.py`. We also provided tests in the `test_ls.py` file. Use these tests to make sure your implementation is correct. The class has three functions `__init__`, `fit`, and `predict`. The `__init__` function is provided. This function simply initializes the class with some provided polynomial degree  $k$ :

```
def __init__(self, k):
    """
    Initialize the LeastSquares class

    The input parameter k specifies the degree of the polynomial
    """
    self.k = k
    self.coeff = None
```

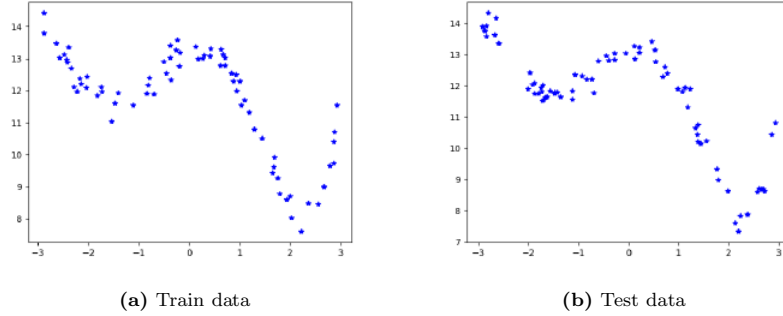


Figure 2.1: Data for polynomial fitting

The `fit` function takes some data and fits the parameters of our model using least squares. In least squares we consider the problem  $\mathbf{Ax} = \mathbf{b}$ , and solve for  $\mathbf{x}$ . To avoid confusion with the previously defined  $x$ , we will rewrite this as  $\mathbf{Ac} = \mathbf{y}$ , where  $\mathbf{c}$  is the vector of polynomial coefficients for which we want to find, and  $\mathbf{y}$  are the target outputs we will use for fitting.

The solution to this least squares problem is  $\mathbf{c} = \mathbf{A}^\dagger \mathbf{y}$ , where  $^\dagger$  is the pseudo-inverse. To adapt this to our polynomial fitting problem we need to define the matrix  $\mathbf{A}$  appropriately. Since  $\mathbf{c}$  is the polynomial coefficients, we can see that  $\mathbf{A}$  should contain the polynomial forms of our data.  $\mathbf{A}$  is defined as:

$$\mathbf{A} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^k \\ 1 & x_1 & x_1^2 & \dots & x_1^k \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{N-1} & x_{N-1}^2 & \dots & x_{N-1}^k \end{bmatrix} \quad (2.1)$$

where  $x_i$  is the  $i$ th sample of the input in  $\mathbf{x}$ .

We can use the Numpy function `np.linalg.pinv` to compute the pseudo-inverse. It is important that we use the pseudo-inverse instead of the inverse because we have an over-determined system, and  $A$  is not invertible.

In the `predict` function we can take the computed  $\mathbf{A}$  and stored  $\mathbf{c}$  (in the `self.coeff` variable) and compute the outputs using the function  $\mathbf{y} = \mathbf{Ac}$ . The predict function should return this output.

After implementing the class in the `ls.py` file, we can run the model by creating a new Python script `run_ls.py` to load the data, train, and test the model. Keeping the model implementation separate from the `run_ls.py` code allows us to easily reuse our model for other applications. In `run_ls.py` we call the `fit` function using the training data to fit the parameters of the model, and then call the `predict` function using the testing data. To assess the performance of the model we can measure the mean square error (MSE) of the testing data. Recall that mean square error is defined as:

$$\text{MSE}(\mathbf{y}, \mathbf{z}) = \frac{1}{N} \sum_{i=0}^{N-1} (y_i - z_i)^2 \quad (2.2)$$

We have provided a starting point for `run_ls.py`. This code loads the data, trains the model, and gets the test predictions. You need to modify this code to compute the MSE of the test and train data. Also we want to test the performance of the model as we vary the degree of the polynomial from 1 to 20. Plot the train and test MSE as a function of the polynomial degree using Matplotlib. By plotting the train and test error, we can see for which polynomial degrees the model is underfitting (train and test error are both high), or overfitting (train error is low, but test error is high).



#### Deliverable: Least Square Polynomial Fitting

- A working version of `ls.py` that passes all of the automated unit tests in `test_ls.py`
- A plot of the train error and test error vs the degree of the polynomial saved as `ls_error.png`
- A modified working version of `run_ls.py` that plots the train error and test error vs the degree of the polynomial.

## 2.4 Implement K-Nearest Neighbors Model (KNN)

Next we will implement is the K-Nearest Neighbors model (KNN). For this model, training involves memorizing the entire training set. During testing, the KNN model simply compares the test sample to its  $k$  nearest neighbors in the training set using some distance measure in the feature space, where  $k$  is a hyper-parameter of the model. The predicted output is the mode of the nearest neighbors' classes. For example, if  $k = 1$  the model finds the nearest training sample in the feature space, and predicts the same class as that training sample. Figure 2.2 shows an example of KNN for  $k = 3$ . For our implementation of KNN, we will use the Euclidean distance measure in the feature space, however it is possible to use any distance measure in KNN.

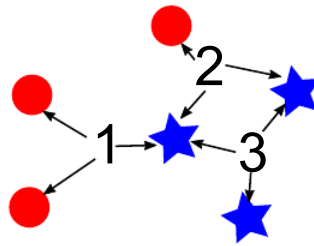


**Figure 2.2: KNN toy example.** The blue stars correspond to the training data from one class, and the red circles correspond to the training data from another class. The numbers are new test data points. For each test data point, we predict the label using the  $k$  nearest neighbors from the training data. For  $k = 3$ , 1 is mapped to the red circle class, and 2 and 3 are mapped to the blue star class. But note that if  $k = 1$ , 1 and 3 are mapped to the blue star class, and 2 is mapped to the red circle class.

**Implementation** We will implement a simple version of the KNN model. Our implementation will take the form of a Python class. By using a class, we can train the model once, and then use the trained object to test many times. Of course “training” in KNN is simply storing the training data in some class variables. We will implement the class using an interface similar to the **Scikit-Learn** models we will use later. By using a common class interface (i.e., functions have the same names and same arguments), we can swap model objects in our code to compare the performance of different models with minimal effort.

We provided the initialization routine of the class in the file `knn.py`. Note the presence of the optional keyword argument in the function definition. This means that if the class is initialized using the code `KNN()`, then the `k` variable will be set to the default value 3. But we can also call the code and explicitly set `k` using `KNN(k=5)`. We set the `x_train` and `y_train` variables to `None`. It is good practice to initialize these variables in the initialization function, even if the variables are initialized to `None`. This allows us to later check if the model has been trained by checking if `self.x_train`

is not None. Also defining all the class variables in the initialization function is good for code readability.

```
def __init__(self, k=3):
    self.x_train = None
    self.y_train = None
    self.k = k
```

You will need to implement the remaining `predict(x)` and `fit(x,y)` functions in the `knn.py` file. The `fit(x,y)` function takes two arguments: the training data `x` and the training labels `y`. We will be using the Scikit-learn convention for the input data format. The input variable `x` is a  $N \times D$  matrix where  $N$  is the number of data samples, and  $D$  is the number of feature dimensions. In other words, the rows of `x` correspond to data samples and the columns correspond to features. The variable `y` is a vector (or equivalently a Numpy matrix with one dimension) of size  $N$ . A typical fit function would run some optimization procedure to find parameters of the model. For the KNN case, the parameters of the model are the training data itself, so we only need to store the training data in the `fit(x,y)` function.

Most of the implementation occurs in the `predict(x)` function. For a given testing data point, we need to compute the Euclidean distance between the testing point and all of the training data points. The  $k$  closest points using these distances are the  $k$ -nearest neighbors. We can use the `np.argsort` function to get the indices of the neighbors sorted by distance. Using the nearest  $k$  indices, we can find the class labels of the  $k$ -nearest neighbors by using the stored labels in the `self.y_train` variable. Finally, of the  $k$ -nearest class labels we take the most common label as the class prediction. This can be computed using `scipy.stats.mode`. The input `x` in the `predict(x)` function is of the same format as in the training function ( $N \times D$ ). This means that the `predict(x)` function should handle inputs with multiple data samples, and should return a vector of size  $N$  of the predicted class labels.

#### Extra Information: KNN Speed

The naïve implementation of the KNN model must loop through every data point in the training set to compute the distances for a single test point. The computational complexity of testing a single sample grows with the size of the dataset. This is clearly not desirable as modern datasets may have thousands or millions of samples. However, we do not need to check every sample in the training dataset every time we test a sample in KNN. There are algorithms which attempt to partition the search space in order to make prediction more efficient. The KD-tree is one such popular algorithm that can be used to speed up KNN [7].

We have provided unit tests in `test_knn.py`. This will use some synthetic data to make sure the nearest neighbors computation is correct. For running on a dataset we do not use unit testing, because it is difficult to know what the output should be for a large dataset before training the model. For this we have provided a small script `run_knn.py` that shows how to train and test on a dataset. We will use a synthetic dataset (Figure 2.3) to test our model because it is easy to visualize the input features. The synthetic dataset is implemented in the `datasets.py` which we can import to test our code. The dataset consists of two classes that are randomly generated from two 2-d Gaussian distributions. The classes slightly overlap which makes the learning problem difficult because some data points are ambiguous. In a real world problem, the two dimensions could correspond to two different features of the data. For example, the  $x$  axis could be height of an individual, the  $y$  axis could represent the peak running speed of that individual. The label could be whether or not that person played basketball in high school. Although the input features should be good indicators of the output label, there may not be a clear separation with these features. For example, there could be some people that are tall and fast, yet didn't play basketball.

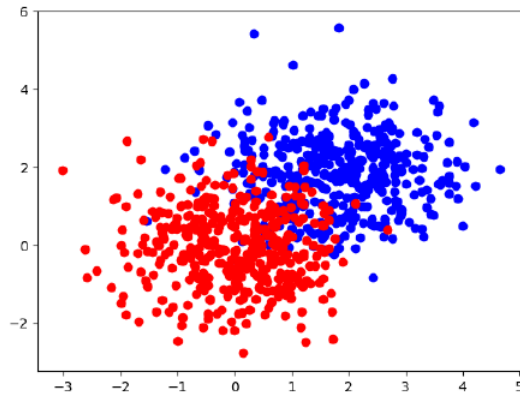


**Figure 2.3: Toy classification dataset.** Blue dots represent class  $y = -1$  and red dots represent class  $y = +1$ . The classes are not perfectly separable which makes classification difficult. Real world problems often have similar noise which makes perfect classification difficult.

**run\_knn.py**

```
from knn import KNN
import numpy as np
import datasets

# load data
x_train, y_train, x_test, y_test = datasets.gaussian_dataset(n_train
    =800, n_test=800)

model = KNN(k=3)
model.fit(x_train, y_train)

y_pred = model.predict(x_test)
print("knn accuracy: " + str(np.mean(y_pred == y_test)))
```

#### Deliverable: KNN Implementation

- A working version of `knn.py` that passes all of the automated unit tests in `test_knn.py`
- A plot of the classification test accuracy on the Gaussian data vs the  $k$  parameter (vary from 1 to 51 in steps of 5 saved as `knn_k.png`)
- A modified working version of `run_knn.py` that plots the test accuracy on the Gaussian data vs the  $k$  parameter.



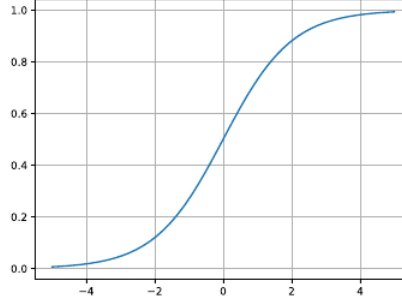


**Figure 2.4: Logistic function**  $f(x) = \frac{1}{1+\exp(-x)}$ . The logistic function forces the output to be between 0 and 1. This allows us to use the output as a probability.

## 2.5 Implement Logistic Regression

**Background** Next we will implement *logistic regression* [8]. Despite the name, logistic regression is actually used as a classification model. Logistic regression is a binary classification model, meaning that it can handle only two classes ( $y = -1$  and  $y = +1$ ). Although this may seem restrictive, it is easy to extend a binary classification model to a multi-class model [9].

Different from KNN, logistic regression outputs probabilities instead of hard class labels. This is why the name contains the word “regression” instead of “classification”. Lets define the output of the logistic regression model as  $f(x)$ , where  $0 \leq f(x) \leq 1$ . If  $f(x)$  is close to 1, then the model predicts a high probability of class 1. Likewise, if  $1 - f(x)$  is close to 1 then the model predicts a high probability of class 0. The probabilities give us more intuitive explanation of the prediction for a test example. For a KNN prediction, we don’t know whether the data sample is very representative of a particular class or is somewhere between the two classes. With logistic regression, if  $f(x)$  is near 0.5 then the data sample has characteristics of both classes which makes it difficult to classify. In certain applications this probability is very useful, because we might want to include human judgment for these difficult to classify samples. A probability output also allows us to use logistic regression in conjunction with Bayes’ rule, which allows us to design priors for our problem. Similar to logistic regression, deep neural networks for classification problems also output probabilities instead of hard labels.

Logistic regression is a simple linear classifier, that learns a function with a vector weight parameter  $\mathbf{w}$  and a scalar bias term  $b$ . In terms of the input data this linear function is  $\mathbf{w}^T \mathbf{x} + b$ . To achieve the probabilistic output, logistic regression uses a logistic function to “squash” the linear output (Figure 2.4). This function ensures that the output is always between 0 and 1. The complete model for the logistic regression takes the form:

$$f(\mathbf{x}_i) = \frac{1}{1 + \exp(-(\mathbf{w}^T \mathbf{x}_i + b))} \quad (2.3)$$

Now lets assume that we have some data  $\mathbf{x}$  and some corresponding labels  $\mathbf{y}$ . We assume that we have a binary classification problem where  $y_i$  can be either  $-1$  or  $+1$ . To optimize the parameters we need to define a *loss function* to minimize. The loss function is sometimes referred to as a *cost function*. We use the cross entropy loss function which, for two classes, is defined as:

$$\ell(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_i -\mathbb{I}(y_i = 1) \ln(f(\mathbf{x}_i)) - \mathbb{I}(y_i = -1) \ln(1 - f(\mathbf{x}_i)) \quad (2.4)$$

where  $N$  is the number of data samples,  $i$  is an index to a particular sample, and  $\mathbb{I}$  is an indicator function. The indicator function is 1 if the argument is true, and 0 otherwise. Basically this loss

function says that when the label  $y_i = 1$ , we want the corresponding value of  $f(x_i)$  to be high. Similarly, when  $y_i = -1$  we want  $f(x_i)$  to be low. After some algebra, this loss can be simplified for our problem to:

$$\ell(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_i \ln(1 + \exp(-y_i(\mathbf{w}^T \mathbf{x}_i + b))) \quad (2.5)$$

**Gradients** There are many ways to minimize the loss function we just defined (e.g., see [10]). We will use the simplest approach and use gradient descent to minimize the loss. In gradient descent we compute the gradient of the loss with respect to each learnable parameter of our model. We then perturb each parameter by small step in the opposite direction of the gradient. For this problem we have two parameters that we adjust according to:

$$b' = b - \eta \left( \frac{d\ell}{db} \right) \quad (2.6)$$

$$\mathbf{w}' = \mathbf{w} - \eta \left( \frac{d\ell}{d\mathbf{w}} \right) \quad (2.7)$$

where  $\eta$  is the *learning rate*. Note that we are subtracting the derivative because we want to minimize the loss function (i.e., we want to “go downhill”). The learning rate governs how big of a step is taken in the descent direction. We might be tempted to set a large learning rate to try to make the optimization converge faster. However if the rate is too large, the gradient descent update might overshoot the function that is being minimized which might cause slow convergence or no convergence at all. A small learning rate won’t have this overshooting problem. However, if the learning rate is too small the optimization may get trapped in a small local minima, because the step is not large enough to “climb out” of the local minima. In practice the learning rate is a hyper-parameter that must be appropriately set for the problem.

To do the gradient descent update we need to compute the gradients of the loss function with respect to our learnable parameters. To do this we can use the chain rule on equation 2.5:

$$\begin{aligned} f &= -y_i(\mathbf{w}^T \mathbf{x}_i + b) \\ g &= 1 + \exp(f) \\ h &= \frac{1}{N} \sum_i \ln(g) \\ \frac{d\ell(\mathbf{x})}{db} &= \left( \frac{dh}{dg} \right) \left( \frac{dg}{df} \right) \left( \frac{df}{db} \right) \\ \frac{d\ell(\mathbf{x})}{d\mathbf{w}} &= \left( \frac{dh}{dg} \right) \left( \frac{dg}{df} \right) \left( \frac{df}{d\mathbf{w}} \right) \end{aligned} \quad (2.8)$$

Computing these derivatives is relatively simple:

$$\begin{aligned} \frac{df}{d\mathbf{w}} &= -y_i \mathbf{x}_i \\ \frac{df}{db} &= -y_i \\ \frac{dg}{df} &= \exp(-y_i(\mathbf{w}^T \mathbf{x}_i + b)) \\ \frac{dh}{dg} &= \frac{1}{N} \sum_i \frac{1}{1 + \exp(-y_i(\mathbf{w}^T \mathbf{x}_i + b))} \end{aligned} \quad (2.9)$$

Finally we get the two derivatives we are interested in:

$$\frac{d\ell(\mathbf{x}, \mathbf{y})}{db} = \frac{1}{N} \sum_i \frac{-y_i}{1 + \exp(+y_i(\mathbf{w}^T \mathbf{x}_i + b))} \quad (2.10)$$

$$\frac{d\ell(\mathbf{x}, \mathbf{y})}{d\mathbf{w}} = \frac{1}{N} \sum_i \frac{-y_i \mathbf{x}_i}{1 + \exp(+y_i(\mathbf{w}^T \mathbf{x}_i + b))} \quad (2.11)$$

The gradient descent method is a repeated application of Eq. 2.7 (Algorithm 1). For the logistic regression implementation, we will compute the average gradient using the entire training dataset. One pass through the entire dataset is called an *epoch*. Updating using the entire dataset is feasible because our training dataset is relatively small. However if our training set is larger, we can split the training set into smaller batches and perform a gradient descent update after each batch. We can check the loss after each epoch to ensure that it is decreasing. Figure 2.5 shows the logistic regression model decision boundary on the toy Gaussian dataset. As we train for more epochs the decision boundary makes a better separation between the two classes.

---

**Algorithm 1:** Gradient Descent Algorithm

---

**Input** : Training data  $(\mathbf{x}, \mathbf{y})$   
1  $\mathbf{w} \leftarrow$  randomly initialized  
2  $b \leftarrow 0$   
3 **for**  $i$  in  $\text{range}(n\_epochs)$  **do**  
4      $b \leftarrow b - \eta \frac{d\ell(\mathbf{x}, \mathbf{y})}{db}$   
5      $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{d\ell(\mathbf{x}, \mathbf{y})}{d\mathbf{w}}$   
6 **end**

---

After training we can predict the output for new test data. Equation 2.3 gives us a continuous number representing the probability of class  $y = 1$ . In this case, we want to predict discrete class labels. This can be done by thresholding the output of Eq. 2.3:

$$y_{test} = \begin{cases} -1 & f(\mathbf{x}_{test}) \leq 0.5 \\ +1 & f(\mathbf{x}_{test}) > 0.5 \end{cases} \quad (2.12)$$

**Implementation** We will be implementing the logistic regression model in the script we offered `logistic_regression.py` using the gradient expressions we just derived. The structure of the class is similar to our KNN implementation, however we have additional methods for computing the gradients and the loss. As usual, the unit tests for the logistic regression model are in the `test_logistic_regression.py` file. Now the unit testing is starting to become useful, because we need to implement several functions for the model to train properly, and if any of these functions are incorrect, our model may not train. The unit tests allow us to pinpoint possible reasons for why the model is not training.

Table 2.1 provides descriptions of the methods of the `LogisticRegression` class. Table 2.2 shows the expected dimensions of the variables in this class. The variables in your implementation should have the same dimensions. Recall that in math, vectors are by convention column vectors, however in machine learning implementations it is convention to express data in terms of row vectors. Like in the KNN implementation, the input variable  $\mathbf{x}$  has data samples as rows and features as columns. A single sample  $\mathbf{x}[i, :]$  is a row vector. To deal with this difference between the math and the implementation you can make use of the `np.transpose` function. To check the dimensions of a Numpy variable you can use `ndarray.shape`.

We will be using the same toy dataset that we used for the KNN implementation. We can copy the `run_knn.py` to create a `run_logreg.py` file. Since we implement the `LogisticRegression` class

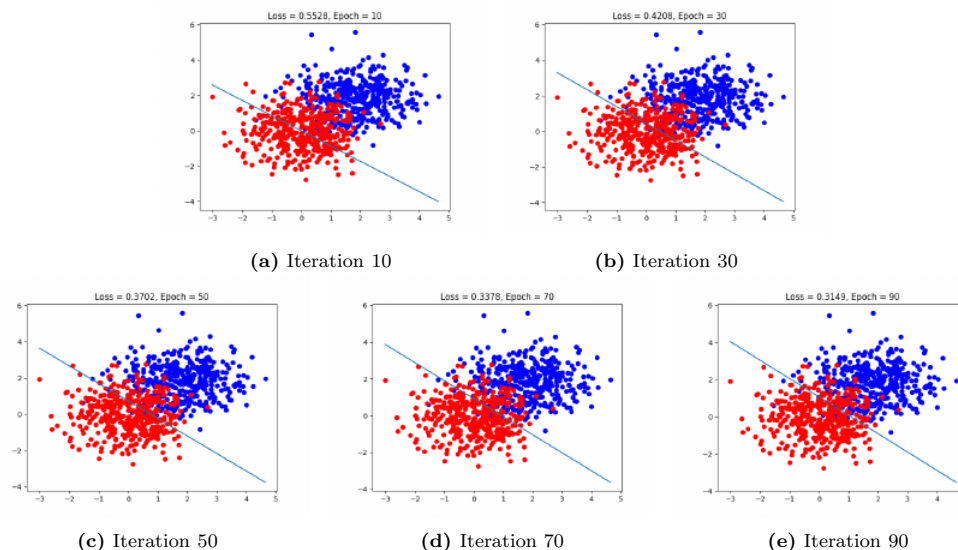


**Figure 2.5: Convergence of logistic regression.** Here we show the learned decision boundary at different stages of training with a learning rate of 0.1. Each epoch is one gradient descent update computed over the entire training data. We plot the decision boundary that separates the predicted positive class from the predicted negative class. As gradient descent progresses, the decision boundary comes closer to the optimal boundary.

**Table 2.1:** Functions in `logistic_regression.py`

| Function                     | Description                                                             |
|------------------------------|-------------------------------------------------------------------------|
| <code>__init__</code>        | Initialization of class (provided)                                      |
| <code>forward</code>         | Implement logistic function (Eq. 2.3)                                   |
| <code>loss</code>            | Implement logistic loss function (Eq. 2.5)                              |
| <code>grad_loss_wrt_b</code> | Compute gradient of loss with respect to $b$ (Eq. 2.10)                 |
| <code>grad_loss_wrt_w</code> | Compute gradient of loss with respect to $\mathbf{w}$ (Eq. 2.11)        |
| <code>fit</code>             | Run gradient descent on given data to optimize parameters (Algorithm 1) |
| <code>predict</code>         | Predict output given data (Eq. 2.12)                                    |

with an identical interface as the KNN model, all we need to do is import the `LogisticRegression` class and change the line that defines the model variable. The `fit` and `predict` functions are called in exactly the same way as in the KNN example.

**Adding Regularization** One potential problem with the logistic regression model we implemented is that there is no way to limit the magnitude of  $\mathbf{w}$ . During optimization the magnitude of  $\mathbf{w}$  can become arbitrarily large. This may cause “over-fitting” to the training data by over-emphasizing a dimension of the input by learning a very large weight. It is common practice to add a regularization term to the cost function that limits the magnitude of  $\mathbf{w}$ :

$$\ell(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_i \ln(1 + \exp(-y_i(\mathbf{w}^T \mathbf{x}_i + b))) + \frac{1}{2} \lambda \mathbf{w}^T \mathbf{w} \quad (2.13)$$

This forces the magnitude of  $\mathbf{w}$  to be small. The trade-off between this and the original cost

**Table 2.2:** Variables in `logistic_regression.py`

| Variable                 | Description              | Dimensions (rows $\times$ cols) |
|--------------------------|--------------------------|---------------------------------|
| <code>x</code>           | Input data               | $(N \times D)$                  |
| <code>y</code>           | Input labels             | $(N)$                           |
| <code>self.b</code>      | Bias parameter           | scalar                          |
| <code>self.w</code>      | Weight parameter         | $(1 \times D)$                  |
| <code>self.lr</code>     | Learning rate            | scalar                          |
| <code>self.l2_reg</code> | L2 regularization weight | scalar                          |

function is governed by the  $\lambda$  parameter. It is easy to derive a new gradient for  $d\ell(\mathbf{x}, \mathbf{y})/d\mathbf{w}$  (left for the reader) using a similar method that we used for the logistic regression model. The gradient for  $d\ell(\mathbf{x}, \mathbf{y})/db$  does not change with the added regularization.

Modify the `grad_wrt_w` and `loss` functions to include the term from the new cost function. The  $\lambda$  parameter is represented by the `l2_reg` variable, which to this point has been set to 0. The rest of the functions should not change.

#### Deliverable: Logistic Regression class

- A working version of `logistic_regression.py` that passes all of the automated unit tests in `test_logistic_regression.py`.
- Using the same Gaussian dataset as `run_knn.py`, generate a plot of the loss function value vs iteration for learning rates of 0.1, 0.01, and 50.0 for 100 epochs. Use the `plt.legend` function to add a legend to the plot. Save this plot as `log_res_lr.png`.
- A modified working version of `run_logreg.py` that plots the loss function value vs gradient descent iterations on the same Gaussian dataset as `run_knn.py`.

## 2.6 Implementing the Support Vector Machine

Now we will implement the support vector machine (SVM) model [11]. Like the logistic regression model, the basic SVM model learns a separating hyper-plane of the form  $\mathbf{w}^T \mathbf{x} + b$ . The primary difference between the SVM and the logistic regression model is the SVM's hinge loss function. Although there exist more efficient ways to train SVMs [12], we will use the same gradient descent algorithm that we used for logistic regression. The SVM hinge loss is defined as:

$$\ell(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_i \max(0, 1 - y_i(\mathbf{w}^T \mathbf{x}_i + b)) + \frac{1}{2} \lambda \mathbf{w}^T \mathbf{w} \quad (2.14)$$

Like logistic regression we will need to compute the gradients  $\frac{d\ell(\mathbf{x}, \mathbf{y})}{d\mathbf{w}}$  and  $\frac{d\ell(\mathbf{x}, \mathbf{y})}{db}$ . This time we leave the gradient derivation to the reader. Mathematically, the max function does not have a smooth gradient because of the discontinuity at  $1 - y_i(\mathbf{w}^T \mathbf{x}_i + b) = 0$ . In practice we don't need to worry too much about this discontinuity, and we can just set the derivative at  $1 - y_i(\mathbf{w}^T \mathbf{x}_i + b) = 0$  equal to 0.

The project folder includes unit tests in the `test_svm.py`. You will implement an SVM class in the `svm.py`. Similar to KNN and logistic regression, you can create a `run_svm.py` file to test your code with the toy Gaussian dataset. The dimensions of the variables are the same as in the logistic regression model (Table 2.2). The SVM will use the same gradient descent algorithm defined in 1, however the computation of the gradients will be different. Finally for testing, the SVM does not

have a probabilistic output like logistic regression. Instead a value that is less than 0 corresponds to a predicted class of -1 and a value greater than 0 corresponds to a predicted class of +1.

**Hint:** `ndarray.max` vs `np.maximum`

`ndarray.max` and `np.maximum` are different. Read the documentation of these functions to make sure you are using the appropriate function.

#### Deliverable: SVM Implementation

- A working version of `svm.py` that passes all of the automated unit tests in `test_svm.py`
- Using the same Gaussian dataset as `run_knn.py`, generate a plot of the loss function value vs gradient descent iteration for learning rates of 0.01, 0.1, and 1 for 100 epochs. Use the `plt.legend` function to add a legend to the plot. Save this plot as `svm_lr.png`.
- A modified working version of `run_svm.py` that plots the loss function value vs gradient descent iteration on the same Gaussian dataset as `run_knn.py`.



## 2.7 Using Scikit-Learn Models

There are many proposed models for classification problems. Thankfully instead of implementing them ourselves, there exists high quality Python implementations. The **Scikit-Learn** library provides implementations of many machine learning algorithms with a consistent interface. This means that we can use the same evaluation code and replace one or two lines to use an entirely different machine learning model. In fact, the KNN, SVM and logistic regression classes we implemented in this chapter use the same `fit` and `predict` function definitions as in the Scikit-Learn models. So we already know how to use the Scikit-Learn models.

To install Scikit-Learn run the following `pip` command in the terminal:

```
sudo pip install scikit-learn
```

As a first example let's take our previous experiments and replace them with an SVM model from Scikit-Learn. Duplicate the `run_svm.py` file we created earlier and name the new file `run_sklearn_svm.py`. To use Scikit-Learn, first we need to import the SVM models from Scikit-Learn:

```
import sklearn.svm
```

Note that the package for Scikit-Learn is called `sklearn`. The Scikit-Learn SVM model can be instantiated by:

```
model = sklearn.svm.SVC(C=1.0, kernel='linear')
```

Here `C` is a parameter that controls the trade-off between the regularizer and the rest of the cost function. Instead of using a  $\lambda$  to weight the regularization term, the Scikit-Learn implementation weights the  $\max(0, \dots)$  in the cost function by the scalar `C`. We defined the cost function in terms of  $\lambda$  because this is closer to the optimization literature and similar to the logistic regression example. Note that `C` and  $\lambda$  are used in similar ways to control the relative weight of the regularizer term. The `kernel='linear'` term will make the SVM implementation use the same  $\mathbf{W}^T \mathbf{x} + b$  form that we implemented in our SVM. We will discuss more about the kernel parameter later. Scikit-Learn

calls the class `SVC` which stands for Support Vector Classifier. This is to distinguish it from other support vector models that can be used for regression problems.

The data format for the Scikit-Learn models is the same that we have been using for our models. We use the `fit(x,y)` function to train the model and the `predict(y)` function to test the model. The rows of the inputs  $\mathbf{x}$  are samples and the columns are features.  $\mathbf{y}$  is a vector of the corresponding class labels. Run the Scikit-learn SVM on the Gaussian dataset and make sure that the classification accuracy is similar to that of the SVM model you implemented earlier.

#### Extra Information: Scikit-Learn SVM Implementation

The Scikit-learn implementation (based on Lib-SVM [12]) reformulates the optimization problem as a quadratic programming problem to perform more efficient optimization, so the final parameters will likely be different than the parameters we learned using simple gradient descent. In fact, the Scikit-Learn implementation does not require a learning rate for its optimization procedure. If you are curious, you can benchmark your SVM implementation and compare the training time with the Scikit-Learn implementation.

**Non-linear SVM** The toy problem we have been using is good to test linear classifiers, however often real world problems cannot be solved with a linear classifier. One example for such a problem is the half-moon problem (Figure 2.6). Although this problem should be easier than the Gaussian dataset because there are no overlapping data points, the linear classification models we used thus far will not give a good classification accuracy. For this type of problem we need a non-linear classifier.

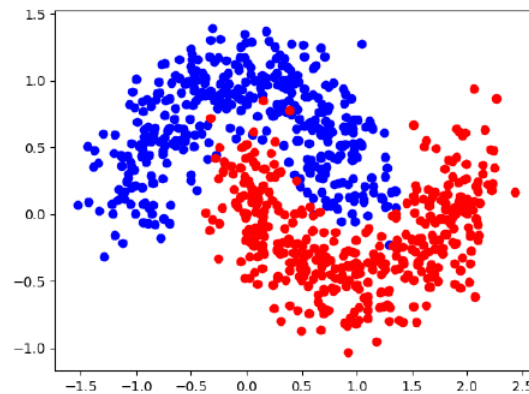


**Figure 2.6: Half moon dataset.** The half moon dataset consists of two classes that cannot be linearly separated. This will cause a low classification accuracy for the linear classifiers we implemented in this chapter. However if we use a non-linear SVM, we can achieve much better accuracy.

The linear classifier is easier to formulate and optimize, so if possible we would like to keep these advantages. The SVM model uses the so called “kernel trick” to essentially use a linear classifier to classify non-linear data. The kernel trick does this by applying a kernel to transform the input data into more dimensions. The motivation is that in higher dimensions it becomes easier to find a separating hyper-plane because there are more degrees of freedom. Figure 2.7 shows an intuitive example of this. In one dimension the data cannot be perfectly separated by a linear classifier. However if we apply the kernel  $k(x) = x^2$  we can fit a line that perfectly separates the data. A full

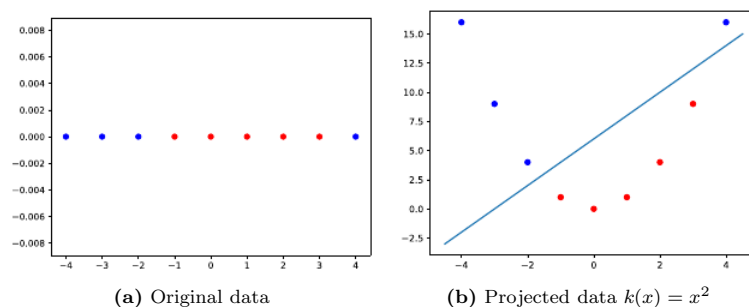


**Figure 2.7: Conceptual example of the SVM kernel trick.** (a) shows data with a single feature of two classes: red dots and blue dots. The data is clearly not linearly separable, so a linear SVM will fail with this data. However if we use the function  $k(x) = x^2$  to project the data to a second dimension, it is possible to fit a linear separating line to the data. In practice our data often has much more than two dimensions, so it is difficult to visualize the effect of the kernel trick. Nevertheless, this example shows how projecting the data to additional dimensions can help classification.

description of the mathematics of the SVM kernel trick is beyond the scope of this lab, but it can be found in [13].

Scikit-Learn allows 5 different options for the SVM kernel. The `linear` kernel implements the same SVM that we implemented ourselves earlier. `poly` uses a polynomial function for the kernel. `rbf` uses a radial basis (Gaussian) function. `sigmoid` uses a hyperbolic tangent function. Finally `pre-computed` allows for a custom pre-computed kernel.

To demonstrate the non-linear SVM we will use the toy half moon dataset (Figure 2.6). We have provided a function to generate this dataset in the `datasets.py` file. In a testing script this data can be loaded with the code:

```
x_train, y_train, x_test, y_test = datasets.moon_dataset(n_train=800,
n_test=800)
```

#### Extra Information: Scikit-Learn

Scikit-Learn contains implementations of many popular machine learning models, including classification, regression, and unsupervised models. While deep learning can handle many problems, the “shallow” learning models in Scikit-Learn can be used in conjunction with deep learning, or can solve problems on their own. The shallow models can be much more efficient, and could be tried before deep learning when working on a new problem.

You will need to modify `run.sklearn.svm.py` to use this dataset and compare the accuracy using 4 different kernel functions.



### Deliverable: Scikit-Learn SVM

- Test the Scikit-Learn linear SVM on the half moon dataset and report the accuracy
- Change the kernel of the Scikit-Learn SVM to polynomial, rbf, sigmoid and report the accuracy. You may need to change other parameters of the kernels to obtain good results.
- A modified working version of `run_sklearn_svm.py` that calculates the accuracy of the Scikit-Learn SVM with different kernels.
- Store these results in a comma separated value file called `svm_results.csv` where each line is of the form `algorithm,0.XXXX`

## 2.8 Submission Instructions

We will test your code using unit tests similar to (but not identical to) the unit tests that we used in this chapter. Thus you must be sure that your code works for all cases, not just the particular cases in the given unit tests. Do not change any of the names of the classes or functions. Also do not change the folder structure of the code as this is assumed by the testing code used for grading. You will be graded based on how many tests your code passes.

Replace the name of the given source folder with `FIRSTNAME_LASTNAME_LAB2` and zip the folder for submission. The plots that you generated as well as the `svm_results.csv` file should be placed in a folder called `results` inside the `FIRSTNAME_LASTNAME_LAB2` folder.

The following are the grading guidelines we will use for the hands-on exercises in this chapter:

**Table 2.3:** Grading rubric

| Points | Description                                                                                                        |
|--------|--------------------------------------------------------------------------------------------------------------------|
| 10     | Correct file names and folder structure                                                                            |
| 10     | <code>ls.py</code> implementation that passes all tests                                                            |
| 5      | <code>ls_error.png</code> plot of the error of least squares regression vs $k$                                     |
| 5      | <code>run_ls.py</code> that plots the error of least squares regression vs $k$                                     |
| 10     | <code>knn.py</code> implementation that passes all tests                                                           |
| 5      | Plot of KNN accuracy vs $k$                                                                                        |
| 5      | <code>run_knn.py</code> that plots KNN accuracy vs $k$                                                             |
| 20     | <code>logistic_regression.py</code> implementation that passes all tests                                           |
| 5      | Plot of logistic regression training loss vs iteration for learning rates 0.1, 0.01 and 50.0                       |
| 5      | <code>run_logreg.py</code> that plots logistic regression training loss vs iteration for different $lr$            |
| 20     | <code>svm.py</code> implementation that passes all tests                                                           |
| 5      | Plot of SVM training loss vs iteration for learning rates 0.1, 0.01 and 1                                          |
| 5      | <code>run_svm.py</code> that plots SVM training loss vs iteration for different $lr$                               |
| 5      | <code>run_sklearn_svm.py</code> that runs Scikit-Learn SVM and that calculates the accuracy with different kernels |
| 5      | <code>svm_results.csv</code> file with results from running Scikit-Learn SVM                                       |
| 120    | Total                                                                                                              |

# Bibliography

- [1] “Numpy reference.” <https://docs.scipy.org/doc/numpy-1.13.0/reference/>, 2017.
- [2] “Scipy reference.” <https://docs.scipy.org/doc/scipy-1.0.0/reference/>, 2017.
- [3] “Matplotlib pyplot reference.” [https://matplotlib.org/api/pyplot\\_api.html](https://matplotlib.org/api/pyplot_api.html), 2017.
- [4] J. Johnson, “Python numpy tutorial.” <http://cs231n.github.io/python-numpy-tutorial/>, 2017.
- [5] “Luma coding in video systems.” [https://en.wikipedia.org/wiki/Grayscale#Luma\\_coding\\_in\\_video\\_systems](https://en.wikipedia.org/wiki/Grayscale#Luma_coding_in_video_systems), 2017.
- [6] Wikipedia, “DFT matrix — Wikipedia, the free encyclopedia.” <http://en.wikipedia.org/w/index.php?title=DFT%20matrix&oldid=811427639>, 2017.
- [7] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- [8] D. R. Cox, “The regression analysis of binary sequences,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 215–242, 1958.
- [9] M. Aly, “Survey on multiclass classification methods,” *Neural Networks*, 2005.
- [10] T. P. Minka, “A comparison of numerical optimizers for logistic regression,” 2003.
- [11] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [12] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [13] H. Yu and S. Kim, “SVM tutorial: classification, regression and ranking,” in *Handbook of Natural computing*, pp. 479–506, Springer, 2012.

## Chapter 3

# A Neural Network from Scratch

### 3.1 Objectives

- Understand backpropagation
- Implement the fully connected Layer, ReLU layer, Softmax layer, and cross entropy loss function
- Create a network with the implemented layers, and test the implemented network on a subset of the CIFAR100 dataset.

### 3.2 Understanding Backpropagation

Backpropagation is the method used to compute gradients in deep neural networks. The gradients allow us to use gradient descent based optimization algorithms to train deep networks. To understand backpropagation, we will first look at a simple example. Consider the functions:

$$f_1(x_1; w_1, b_1) = w_1x_1 + b_1 \quad (3.1)$$

$$f_2(x_2; w_2, b_2) = w_2x_2 + b_2 \quad (3.2)$$

$$z(x) = f_2(f_1(x)) \quad (3.3)$$

where  $x_i$ ,  $w_i$ , and  $b_i$  are all scalars. Let's say we want to "train" this function to achieve a lower value of  $\ell(x) = z(x)$ . Obviously this function is trivial to minimize, but we can still see how to compute the gradients for all of the parameters using the backpropagation technique. Specifically, we need to compute  $\frac{d\ell}{dw_1}$ ,  $\frac{d\ell}{dw_2}$ ,  $\frac{d\ell}{db_1}$ , and  $\frac{d\ell}{db_2}$ . With these gradients we can use gradient descent to update the parameters.

We could ignore the decomposition of  $z$  into  $f_1$  and  $f_2$ , write the full equation for  $z$  and derive the gradients manually. For this example, this method will work. However if we have a function composed of many other functions, as in a deep neural network, the final function derivatives will be very tedious to derive. It is usually much easier to compute the gradients of the smaller functions and build the gradient for the final function using the chain rule. This decomposition process will be the same that we use in backpropagation for training deep neural networks later in this chapter.

Let's first consider the gradients in the  $f_1$  function:  $\frac{df_1}{dw_1}$ ,  $\frac{df_1}{db_1}$ , and  $\frac{df_1}{dx_1}$ . These are fairly easy to

derive:

$$\begin{aligned}\frac{df_1}{dw_1} &= x_1 \\ \frac{df_1}{db_1} &= 1 \\ \frac{df_1}{dx_1} &= w_1\end{aligned}\tag{3.4}$$

The gradients of the parameters in  $f_2$  are identical to those in  $f_1$ , except for the different parameters  $w_2$  and  $b_2$ . If we had many of these same functions composed together, we would only need to do these three gradient derivations, and we can reuse the derivations many times.

Now to find the gradients with respect to the final function  $z$  we can use the chain rule. Lets start with the parameters of  $f_2$ . These are the simplest gradients to compute because the function is closest to the output  $z$ .

$$\begin{aligned}\frac{d\ell}{dw_2} &= \frac{df_2}{dw_2} = x_2 = f_1(x_1) = w_1x_1 + b \\ \frac{d\ell}{db_2} &= \frac{df_2}{db_2} = 1\end{aligned}\tag{3.5}$$

Next we will compute the gradients of the parameters of  $f_1$ . Using the chain rule, we need to multiply the gradients of the  $f_1$  parameters by  $\frac{df_2}{dx_2}$ . In other words, the gradients of the  $f_1$  parameters depend on the computation of the gradient of the input to the  $f_2$  function. The gradient flows backwards from the top of the function composition to the bottom. This gives the term *backpropagation*. Figure 3.1 shows this process. Using backpropagation we compute the gradients as follows:

$$\begin{aligned}\frac{d\ell}{dw_1} &= \left(\frac{df_2}{dx_2}\right) \left(\frac{df_1}{dw_1}\right) = w_2x_1 \\ \frac{d\ell}{db_1} &= \left(\frac{df_2}{dx_2}\right) \left(\frac{df_1}{db_1}\right) = w_2\end{aligned}\tag{3.6}$$

Now with all the gradients computed we can update the parameters using gradient descent to give a lower value of the loss  $\ell$ . If this was a neural network, we could consider the functions  $f_1$  and  $f_2$  as layers of the network. Thus in backpropagation we compute the gradient of the last layer with respect to the input of the layer. The second to last layer uses this gradient to compute the gradient with respect to its own parameters. The gradient continues to flow down the network until we compute the gradient of the parameters of the first layer. For more details about the backpropagation algorithm, refer to Chapter 6 of [14].

#### Extra Information: Testing Gradients Numerically

When implementing the gradients of a function, or a complex nested function, it is useful to have a way to check whether our gradient derivation is correct. We can check our derivation numerically using the definition of the derivative:

$$\frac{df}{da} = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}$$

This equation says that we change the parameter by a small value  $h$ . We then use the change in the output value to compute the derivative. If  $h$  is chosen to be small, the calculated gradient and the derived gradient should be very close. We use this type of check in the testing code provided in this chapter to verify the implementation of deep network layers.

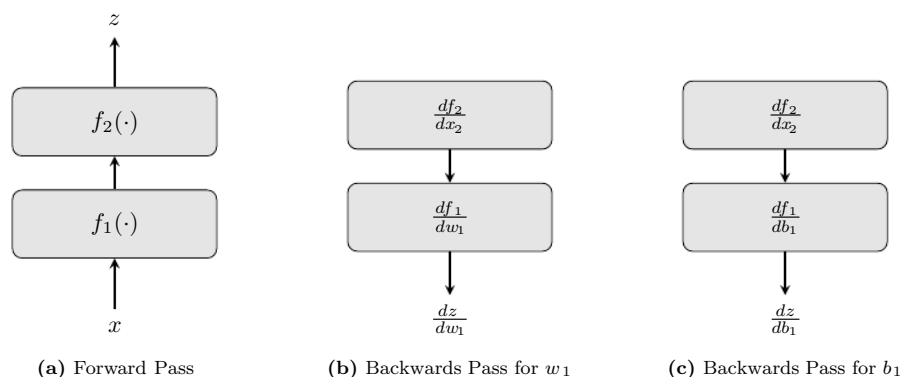


**Figure 3.1: Backpropagation example.** In the forward pass (a) we pass the input  $x$  forward through the functions  $f_1$  and  $f_2$ . In the backwards pass (b) we first need to compute  $\frac{df_2}{dx_2}$ , and then using chain rule we can obtain the desired  $\frac{dz}{dw_1}$ . If we included more functions (e.g.  $f_3$ ), the process would be similar with more steps in the forward and backwards passes. Notice that in the backwards computation for the two parameters, we only need to compute  $\frac{df_2}{dx_2}$  once. By utilizing this observation we can implement a fast version of backpropagation at the expense of memory utilization.

### 3.3 Implementing Layers of a Neural Network

With a basic understanding of backpropagation, we can build a simple neural network. We will be implementing the neural network as a collection of layer classes. We will implement classes for a fully connected layer, a ReLU layer, a softmax layer, and a cross entropy layer.

Each layer class implements an `__init__` function, a `forward` function, and a `backward` function. Layers with learnable parameters will additionally need a `update_params` function. The `__init__` function initializes the layer parameters. The `forward` function implements the forward pass of the layer. The `backward` function of a layer  $l$  takes as input the gradient of the loss function with respect to the input of layer  $l + 1$  and computes the loss function's gradient with respect to the input and with respect of the learnable parameters of the considered layer  $l$ . Typically, the computed loss function's gradient with respect to the input of layer  $l$  is output to be used as input of the `backward` function of layer  $l - 1$ , while the gradients with respect to the parameters are stored for later use in the gradient descent step. Finally, the `update_params` method will take some learning rate and use the computed gradients to update the values of the parameters. In the project folder we have provided templates for all of the layers that we need to implement (`full.py`, `relu.py`, `softmax.py`, `cross_entropy.py`).

Our implementation will be a Python package which we will name `layers`. As a Python package we can use it with `import` statements, just as we use the `numpy` library. A Python package can be easily reused in other projects. Creating a Python package called `layers` only requires the presence of a folder named `layers` with an `__init__.py` file, which we have provided. We put all of the implementation of our network in the `layers` folder. Then a file in the parent folder of the `layers` folder can import our network implementation with commands like:

```
import layers.FullLayer # can be used as: full = layers.FullLayer()
import layers.ReluLayer # can be used as: relu = layers.ReluLayer()
from layers import ReluLayer # can be used as: relu = ReluLayer()
from layers import FullLayer # can be used as: relu = FullLayer()
```

For now we will keep our package local, meaning that it can only be imported by files from the parent directory. It is possible to make the package global using `pip`. It is even possible to publish the package so that other users of `pip` can install the package from a network (but we won't be doing that here).

We have provided a set of tests for the neural network layers in the `test` folder. Different from previous chapters, we have divided up the tests into individual files, where each file tests a single layer class. In this way you can run the single test file for the particular layer that you are working on. You can also run all of the tests in the `test` folder by right clicking on the folder name and selecting `Run 'Unittests in test'` in PyCharm. You can add additional tests to ensure your code is correct, but do not modify the existing tests to make your code pass.

#### Extra Information: Speed

Loops in Python are slow, so if you want training to be fast, it is best to avoid them if possible. Numpy operations like `np.dot` call efficient LAPACK C libraries, so they are much faster than looping. The forward/backwards functions will be called thousands of times during the training, so if the forward/backward functions are very slow, the training will also be very slow.

**Data Format and Batches** We will be using stochastic gradient descent to train our network. In stochastic gradient descent, the dataset is split into batches of data points. The size of the batch is commonly 128 or 256, but there are works that use large batch sizes (e.g.,  $> 1000$ ) or very small batch sizes (e.g., 1). We run the forward and backwards passes of the network on the batch, and then update the parameters. This is different from Chapter 2 where we computed the gradient on the entire training set.

The data passed between layers will be a matrix of size  $n_b \times n_{f_i}$ , where  $n_b$  is the batch size, and  $n_{f_i}$  is the number of features (outputs) in a particular layer  $i$ .  $n_{f_i}$  can change from layer to layer, but  $n_b$  will be the same throughout the entire network. This data organization means that a single data sample will be a row vector.

Our network will be able to do multi-class classification. To represent the class labels we will be using *one-hot encoding*. One-hot encoding represents the label as a vector of zeros with a one at the index of the label. For example if we have 5 classes, class 0 would be represented by the row vector `[1, 0, 0, 0, 0]` and class 3 would be represented by `[0, 0, 0, 1, 0]`. Compared with representing the label as an integer, one-hot encoding simplifies the math and the implementation.

**Layer Interface** All of the layers will use the same class interface. In other words, they will all have the same function definitions with the same input parameter names. Only the implementation of each function varies for the different layers. With a common class interface, we can “plug-in” different layers to change a neural network architecture without needing any additional coding.

The initialization function (`__init__`) of the classes will need to set default values for the class variables. The initial values for network parameters can affect training convergence, so it is important to set these properly. Some network layers have hyper-parameters (such as the number of units in a fully connected layer). The settings of the hyper-parameters will be passed as arguments to the initialization function.

The `forward` function takes an input matrix `x` and returns some output. The input will always be of size  $n_b \times n_{f_{i-1}}$  where  $n_b$  is the batch size, and  $n_{f_{i-1}}$  is the dimensionality of the output from the previous layer.

The `backward` function takes in the input gradient from the higher layer (`y_grad`), returns the gradient with respect to the input, and stores the gradients of any trainable layer parameters. `y_grad` has shape  $n_b \times n_{f_i}$  where  $n_{f_i}$  is the number of outputs of the layer. For some backwards functions we

| Function                  | Description                                  |
|---------------------------|----------------------------------------------|
| <code>--init--</code>     | Initialization of class parameters           |
| <code>forward</code>      | Implement forwards pass of layer             |
| <code>loss</code>         | Implement backwards pass of layer            |
| <code>update_param</code> | Update the learnable parameters of the layer |

**Table 3.1:** Functions for each layer

may need some computed values from the forward function. These computed values can be stored in class variables during the execution of the `forward` function.

The `update_param` method will take a learning rate `lr` and update the parameters of the layer. For layers without learnable parameters, `update_param` needs no implementation. We can use the Python command `pass` to implement a function that performs no operations. For other layers with learnable parameters, the stored gradients from the backwards function should be used to update the parameters.

### 3.3.1 Fully Connected Layer

**Forward pass** The fully connected layer is named as such, because each output of the fully connected layer is a weighted sum of each input element. The fully connected layer takes an input matrix of size  $n_b \times n_i$  and outputs a matrix of size  $n_b \times n_o$ , where  $n_b$  is the batch size,  $n_i$  is the input size, and  $n_o$  is the output size. The weights for each output are learnable and stored in a matrix  $\mathbf{W}$  of size  $n_o \times n_i$ . The implementation of the fully connected layer will be in `full.py`. The forward pass is defined by:

$$f_{full}(\mathbf{x}_i) = \mathbf{x}_i \mathbf{W}^T + \mathbf{b} \quad (3.7)$$

where  $\mathbf{x}_i$  is a *row* vector for one sample of the input of the layer. In this chapter, we will be doing all the math with row vectors, so that the math can more easily transfer into code. In some textbooks or other resources you may see slightly different equations such as  $\mathbf{W}\mathbf{x}_i$ . The weight matrix is sometimes of size  $n_i \times n_o$ , and the vectors are sometimes expressed as column vectors. After the dimensions are taken into account, the different versions of the fully connected layer equation should express the same output. Eq. 3.7 describes the computation for one data sample. Our layer will need to handle a batch of data samples as the input, instead of only a single data sample.

**Backwards pass** Next, for the backwards pass we need to derive the Jacobians:  $\frac{df_{full}}{d\mathbf{x}_i}$ ,  $\frac{df_{full}}{d\mathbf{W}}$ , and  $\frac{df_{full}}{d\mathbf{b}}$ . In this case the input and the output are both multi-dimensional vectors, so we actually need to compute Jacobian matrices. Recall that a Jacobian matrix is simply a matrix of the derivatives of every output with respect to every input.

$$\begin{aligned} \frac{df_{full}}{d\mathbf{x}_i} &= \mathbf{W} \\ \frac{df_{full}}{d\mathbf{b}} &= \mathbf{I}_{n_o} \end{aligned} \quad (3.8)$$

where  $\mathbf{I}_{n_o}$  is the  $n_o \times n_o$  identity matrix.

The Jacobian for  $\frac{df_{full}}{d\mathbf{W}}$  is a tensor (multi-dimensional matrix) because we have an output of size  $n_o$  with respect to an input of size  $n_o \times n_i$ . The mathematics becomes a little difficult for this tensor Jacobian, however when we actually need to compute the gradient during backpropagation, the math simplifies greatly as we will see shortly. See [15] for a detailed derivation of this Jacobian.

These Jacobians give the gradient of the outputs of the layer relative to the inputs or parameters. However, what we actually need to compute is the gradient of the loss function ( $\ell$ ) with respect to

these values. In the `backward` implementation, the input to the function `y_grad` gives the gradient of the output calculated by the later layer in the network. Mathematically `y_grad` represents  $\left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right)$ , where  $\mathbf{x}_i^{l+1}$  is the input to the next layer, or equivalently the output of the current layer. `y_grad` is of size  $n_b \times n_o$ . Similarly, we use  $\mathbf{x}_i^l$  to denote the input to the current layer. We can take this and use matrix multiplication with the Jacobians we just derived to get the gradient of the network output (or loss) relative to the parameter:

$$\begin{aligned}\frac{d\ell}{d\mathbf{x}_i^l} &= \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right) \left(\frac{d\mathbf{f}_{full}}{d\mathbf{x}_i^l}\right) = \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right) \mathbf{W}^l \\ \frac{d\ell}{d\mathbf{W}^l} &= \sum_i \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right) \left(\frac{d\mathbf{f}_{full}}{d\mathbf{W}^l}\right) = \sum_i \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right)^T \mathbf{x}_i^l \\ \frac{d\ell}{d\mathbf{b}^l} &= \sum_i \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right) \left(\frac{d\mathbf{f}_{full}}{d\mathbf{b}^l}\right) = \sum_i \left(\frac{d\ell}{d\mathbf{x}_i^{l+1}}\right)\end{aligned}\tag{3.9}$$

Note that here the gradients of the parameters sum over the data samples  $\mathbf{x}_i$ .

The `backward` function will return  $\frac{d\ell}{d\mathbf{x}_i^l}$ . This return value will be used by the `backward` function of the layer below the current layer.  $\frac{d\ell}{d\mathbf{b}^l}$  and  $\frac{d\ell}{d\mathbf{W}^l}$  should be stored in the `b_grad` and `w_grad` class variables. These stored values will later be used by the `update_param` function.

**Updating the parameters** The `update_param` function will update the parameters of the layer using the stored gradients computed in the `backward` function. We could have easily made this parameter update step a part of the `backward` function, but we leave it separate for flexibility. The PyTorch library, which we will see in a later chapter, also separates the backwards pass from the parameter update step.

Recall from Chapter 2 that when we optimize parameters, we want to move in the opposite direction of the gradient such that the loss decreases. We use some learning rate ( $\eta$ ) to control the speed of the descent. In general we could use different learning rates for different layers or parameters, but in our case we use the same learning rate for all of the learnable parameters.

$$\mathbf{b}' = \mathbf{b} - \eta \left(\frac{d\ell}{d\mathbf{b}}\right)\tag{3.10}$$

$$\mathbf{W}' = \mathbf{W} - \eta \left(\frac{d\ell}{d\mathbf{W}}\right)\tag{3.11}$$

**Initialization** The initial values for the layer parameters must be set properly to achieve good network convergence. If all the values of  $\mathbf{W}$  are set equally, then the network won't learn interesting features because the values of  $\mathbf{W}$  will all be updated with exactly the same values. To break this symmetry we can initialize  $\mathbf{W}$  to have random values. The initialization of  $\mathbf{b}$  is not as important. Setting  $\mathbf{b}$  to a vector of zeros or small random numbers will work in most cases.

The scale of the random initialization of  $\mathbf{W}$  can affect the training. For example, large initial values can lead to overflow. This is because each output of the fully connected layer is a weighted sum of all of the (potentially many) elements of the input sample. Glorot [16] found that a good initialization should depend on the number of inputs  $n_i$  and number of outputs  $n_o$ .  $n_i$  and  $n_o$  are sometimes called “fan-in” and “fan-out”, respectively. Our network will use this type of initialization which takes the form:

$$\mathbf{W} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_i + n_o}}\right)\tag{3.12}$$

$$\mathbf{b} \sim \mathbf{0}\tag{3.13}$$

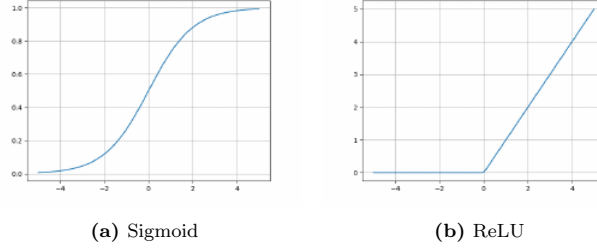


**Figure 3.2: Non-linearity functions used for training neural networks.** Historically, the sigmoid (a) was used, but most modern neural networks use the ReLU (b) function.

### 3.3.2 ReLu Layer

To stack fully connected layers to create a deep network, we need to insert a non-linearity between the layers. If we do not insert the non-linearity, the two stacked fully connected layers are equivalent to a single fully connected layer. Thus without the non-linearity, the network does not have true depth, which may limit its ability to learn more complicated functions.

Historically, the sigmoid non-linearity was used. However, it was shown in [17] that the rectified linear unit (ReLU) nonlinearity works better for training neural networks. The ReLU is defined as:

$$f_{relu}(x_{i,j}) = \begin{cases} x_{i,j} & x_{i,j} > 0 \\ 0 & x_{i,j} \leq 0 \end{cases} \quad (3.14)$$

where  $x_{i,j}$  is the element at index  $j$  of a single data sample  $\mathbf{x}_i$ . Figure 3.2 shows a plot of the ReLU nonlinearity.

For this layer we ask you to derive the equations needed for backpropagation yourself. The ReLU function mathematically does not have a derivative when  $x_{i,j} = 0$ . Similar to what we did in the SVM implementation from Chapter 2, we can set the derivative at this point to 0.

### 3.3.3 SoftMax Layer

For classification problems, the last fully connected layer contains an output for each of the classes we are trying to learn. A larger output value means that the network has a stronger prediction for that particular class. However, the unbounded output values may not be very informative because there is no normalization of the output values. The softmax function will take these output values and normalize them such that the output distribution becomes a probability. The output probabilities will be more meaningful than the raw outputs of the last layer. The softmax function is defined as:

$$z_{i,j} = f_{softmax}(y_{i,j}) = \frac{e^{y_{i,j}}}{\sum_k e^{y_{i,k}}} \quad (3.15)$$

where  $y_{i,j}$  is a single element of the input vector  $\mathbf{y}_i$ , which is the output of the last layer of the network.  $\mathbf{y}_i$  is a row vector whose size is equal to the number of classes. In our implementation we need to compute the softmax for every element  $y_{i,j}$  of every batch sample  $\mathbf{y}_i$ . We note that the softmax function looks very similar to the logistic function from Chapter 2. In fact, the softmax function can be seen as a generalization of the logistic function to multiple classes.

Equation 3.15 is not numerically stable. One problem is that if  $y$  is large then  $\exp(y)$  will be very large. Dividing two large numbers can be numerically unstable because of limited precision in floating point arithmetic. In practice, we instead implement the softmax function by first subtracting

the max of the data. Mathematically, the subtraction has no effect on the result, however it avoids the stability problems with large positive numbers.

$$\begin{aligned} y'_{i,j} &= y_{i,j} - \max_{i,j}(y_{i,j}) \\ z_{i,j} &= f_{softmax}(y_{i,j}) = \frac{e^{y'_{i,j}}}{\sum_k e^{y'_{i,k}}} \end{aligned} \quad (3.16)$$

For the backwards pass we need to compute the Jacobian matrix. For the softmax this is:

$$\frac{dz_i}{dy_i} = \text{diag}(\mathbf{z}_i) - \mathbf{z}_i^T \mathbf{z}_i \quad (3.17)$$

Again, like the other layers the softmax layer's **backward** function should take a gradient from the above loss layer and multiply with the Jacobian.

### 3.3.4 Cross Entropy

Since the softmax layer will transform the output values of the last fully connected layer into probabilities, we can use loss functions that assume probabilities as input. We will use *cross entropy* as our loss function. This cross-entropy function is similar to the cross entropy we used to train the logistic regression classifier in Chapter 2. Cross entropy is usually preferred for classification tasks because of its relationship to KL-divergence and maximum likelihood. For our network, we use the multi-class version of cross entropy, because we want our network to be able to predict multiple classes. The multi-class cross entropy is defined as:

$$\ell(\mathbf{z}, \mathbf{t}) = -\frac{1}{n_b} \sum_i \sum_j \mathbf{t}_{i,j} \log(\mathbf{z}_{i,j}) \quad (3.18)$$

where  $\mathbf{z}$  and  $\mathbf{t}$  contain the softmax output and label for an entire mini-batch.  $\mathbf{t}$  is assumed to be one-hot encoded.  $\mathbf{y}_{i,j}$  is the  $j$ th output of the  $i$ th sample of the mini-batch, and  $\mathbf{t}_{i,j}$  is the corresponding index in the one-hot encoding of the sample class. Log here refers to the base  $e$  logarithm.

The cross entropy layer does not have any learnable parameters, so we only need to compute the gradient with respect to the input of the layer. Additionally, we will assume that the cross entropy layer is the last layer in the network. There is no gradient coming from a higher layer, and so in the **backwards** function the input variable **y\_grad** will be set to **None**. The **backwards** function for this layer simply needs to return the gradient given by:

$$\frac{d\ell(\mathbf{z}, \mathbf{t})}{d\mathbf{z}_{i,j}} = -\frac{1}{n_b} \frac{\mathbf{t}_{i,j}}{\mathbf{z}_{i,j}} \quad (3.19)$$

#### Deliverable: Network Layers

The following layers must be implemented with all unit tests passing:

- `relu.py`
- `full.py`
- `softmax.py`
- `cross_entropy.py`



## 3.4 Putting it all together

### 3.4.1 Implement the Sequential model

To put all of the layers together we will implement another class that collects all of the layers to form a deep neural network. Similar to the Scikit-Learn classes, and the classes we implemented in Chapter 2, the deep network class will implement a `fit(x,y)` function and a `predict(x)` function. We call our model `Sequential` as our model implements a sequential list of layers. Our simple model won't support any connections that are not sequential (i.e., we do not support residual connections as in [18]). We could implement a different model that connects the layers in a more flexible directed graph, but we leave this exercise for the interested reader.

The `Sequential` model will be initialized with a list of layers and a loss layer. We will design the model such that it is generic and will accept an arbitrarily long list of layers. For example we can define our sequential model as follows:

```
from layers import (FullLayer, ReLU, SoftMaxLayer,
                    CrossEntropyLayer, Sequential)

model = Sequential(layers=(FullLayer(32*32*3, 500),
                           ReLU(),
                           FullLayer(500, 5),
                           SoftMaxLayer()),
                  loss=CrossEntropyLayer())
```

This code defines a network with 2 learnable fully connected layers. In between the fully connected layers is a Relu layer, and at the end of the second fully connected layer is a soft-max layer. The first fully connected layer has an input size of  $32 \times 32 \times 3$  which could correspond to a flattened  $32 \times 32$  pixel color image and an output size of 500. The output size here is a designed hyper-parameter of the network. The next fully connected layer has an input size of 500. For our network implementation, the input size of a fully connected layer must be the same as the output size of the previous layer. The output size of the second fully connected layer is 5, which corresponds to 5 output classes. Finally, we must also define the loss function when we initialize the model.

The sequential model is much like a layer, in that it contains `forward` and `backward` functions of its own. These functions will simply call all of the layers' `forward` or `backward` functions and make sure the data is appropriately passed between the layers. We have provided the template in `sequential.py` and tests in `test/test_sequential.py`.

**The forward function** Like a layer, the sequential model has a `forward` function. This function will take the input and pass it through the first layer. The output of the first layer should be passed to the second layer, and so on until we obtain the output of the last layer. This function should work with an arbitrary number of layers that are stored in the `self.layers` class variable. We allow the `forward` function to either return the output of the last layer, or the output of the loss function, depending on whether the optional `target` variable is passed. This allows us to use the `forward` function both for computing the loss and for obtaining the output prediction during testing.

**The backward function** The sequential model should also have a `backward` function that will go through all of the layers for the backwards computation of the gradients. For this function, we start with the loss layer and compute the gradient. The `backward` function of the loss layer can be called without any inputs. Then we feed the loss layer gradient as input to the `backward` function of the previous layer. The output of that latter backward function is again fed as input to the backwards function of the preceding layer, and so forth. The value of a layer gradient is

computed using the value of the input of that layer, which is obtained from the values stored by the `forward` function. We keep doing this until we call the `backward` function of the first layer. In our implementation of the `forward` function for the layers, we should have saved all of the values needed in the `backward` functions. After the backwards pass, the gradients of each parameter will be stored by the corresponding layers to be updated.

**The `update_params` function** We also need a simple function that will update the parameters of the network by some learning rate. For this function we don't need to worry about passing data between layers. We can simply call the `update_params` function of each layer in the network.

**The `fit` function** Finally we can implement the `fit` method to enable our model to be trained. This method will be similar to the `fit` function that we implemented for the logistic regression classifier in Chapter 2 that used gradient descent.

In Chapter 2, we computed the gradient on the entire input data, and then updated the parameters. For the `Sequential` class, we will assume that the size of our data set is very large ( $> 1000$  samples). In this case it may be impossible to calculate the gradients at once as we did in Chapter 2, because of memory limitations. Instead we will split the data into smaller batches of images. These batches are often also called mini-batches. This method of batch-wise training is called *stochastic gradient descent*. Besides saving memory, it also has some theoretical advantages over training using the entire dataset.

For a particular mini-batch we need to perform several operations for training. First, we use the `forward` function to do the forward pass of the network for the particular batch. After this pass, the layers should store all the necessary outputs used in the backwards computation. Next we call the `backward` function to compute the gradients of all of the layers. Finally, we can call the `update_params` function to update the parameters of the network, based on this particular batch.

It is important that the mini-batch is a decent sampling of the actual dataset. If the mini-batch consisted of only a single class, then the network won't learn differences between classes. The easiest way to prevent this is to shuffle the data before splitting into mini-batches. In our case, we assume that the data is already shuffled before calling the `fit` function.

**The `predict` function** Most of the work in the `predict` function is done by the previously mentioned `forward` function. The `forward` function gives the estimated probability of each class for each input data sample. We want our `Sequential` class to be a classifier, so it should return class labels instead of probabilities for each input data sample. In Chapter 2, we obtained labels from the probabilities by using a threshold of 0.5. In this case, there are several output classes, so we want to return the class that has the highest predicted probability. We assume that the `predict` function is called on a small batch of data. So we do not need to worry about splitting the data into batches for this function.

### 3.4.2 Training the model with CIFAR100 data

Now, assuming all of the tests are passing, we can attempt to train our model on a subset of the CIFAR100 dataset [19]. This dataset consists of 60,000  $32 \times 32$  pixel color images for 100 classes. Although the images are of a very small size, humans can still recognize the class of the image (Figure 3.3). The CIFAR datasets are often used for machine learning research because the small image size makes training fast, but there is still a large number of images so we can fit large models.

In the hands-on exercises of this chapter, we will train and test on a 5-class subset of the CIFAR100 dataset. The restriction to 5 classes makes the network faster to train, as well as makes the problem a little easier for our network to learn. Each individual will train/test on a different subset of the dataset. Testing accuracies of different sets can vary since different class combinations can be more difficult.

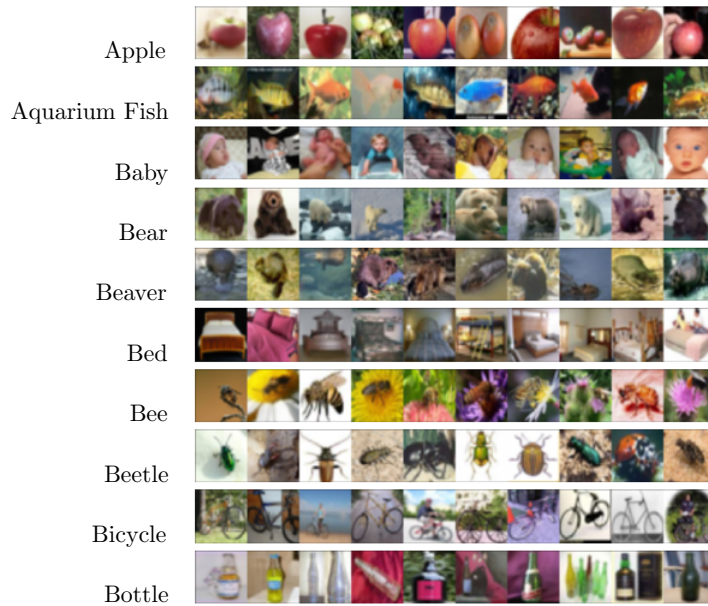


**Figure 3.3: Example classes from CIFAR100 dataset.** For viewing, images are enlarged from the original  $32 \times 32$  size.

We have provided a function that will load the CIFAR100 dataset in the `dataset.py` file. This file loads the provided CIFAR100 data from a `cifar-100-python` folder. The `cifar-100-python` folder can be downloaded at the following link:

<https://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz>

After downloading the `cifar-100-python` folder, put it in the provided `src` folder.

The subset classes is determined using the seed parameter of the `cifar100` function. **Set this seed to your student ID number to get the subset of the data.** We scale the image data to be between -0.5 and 0.5. It is generally a good idea to scale your data before training a network, because some parameter initializations such as Glorot assume that the inputs have zero mean. You can load the dataset with the following code:

```
from layers.dataset import cifar100
(x_train, y_train), (x_test, y_test) = cifar100(seed)
```

You will need to take this data and filter it such that it only has the data corresponding to the assigned subset.

Now, assuming all of the tests are passing, we can try to train our model. For the hands-on exercise in this chapter, we will be training a simple 2-layer neural network. The input to the network is the raw image pixels. We will not be considering any spatial relationships between pixels (as in convolutional neural networks), so we can simply flatten the  $32 \times 32 \times 3$  image into a 3072 length vector for the input of our network. The first layer has 500 units, so the output of the first layer will be of length 500. After the first layer, we use a ReLU layer to add nonlinearity to the network. After the ReLU layer, we use another fully connected layer that outputs 5 values, corresponding to the 5 classes of our classification problem. These outputs will be fed into a softmax layer to output final class probabilities. We will use the cross entropy function as our loss function.

This network structure can be built with the code presented in Section 3.4.1.

For each mini-batch we get the loss from the `forward` function. If we plot this loss for each mini-batch, we will see a noisy, but generally decreasing plot. The plot is noisy because the different (random) mini-batches have different samples that may be easier or more difficult than the average sample in the dataset. A less noisy estimate of model loss can be found by averaging all of the mini-batch losses over an entire epoch. This average loss is different than the loss over the entire dataset, because we update the weights after every mini-batch, so the true epoch loss changes after each mini-batch update. The average mini-batch loss is usually a good approximation of the loss of an epoch, and is generally preferred over recomputing the epoch loss because of the computational savings. When we ask you to plot the loss for an epoch, you can use this average of mini-batch losses.

#### Deliverable: `run_cifar100.py`

Use the finished Sequential model to implement a simple model (network structure shown in Section 3.4.1) on a subset of the CIFAR100 dataset. We have provided an empty script called `run_cifar100.py`, please write your code in it to perform the following tasks:

- 1) Plot the average training loss vs epoch for learning rates 0.1, 0.01 and 0.3 for 20 epochs and save the plot in a file called `acc_lr.png`.
- 2) Run the same network with learning rate 10. What do you observe? Describe your observation in the commented section at the beginning of the script.
- 3) Report the test accuracy with learning rates 0.1, 0.01, 0.3 and 10 for 20 epochs. Store these results in a comma separated value file called `acc_results.csv` where each line is of the form `learning_rate,0.XXXX`.

Submit the completed `run_cifar100.py` file along with the saved plot corresponding to average training loss vs epoch and the `acc_results.csv`.

#### Deliverable: Sequential model with CIFAR100 results

- Implementation in `sequential.py` that passes all tests
- Use the sequential model to construct a neural network that achieves higher testing accuracy than the given network architecture. You can try adding more layers and changing the width (number of outputs) of each layer. Submit a file that trains and tests this model as `better_cifar.py`.

#### Hint: Debugging

If the unit tests pass, this is some indication that your code is correct. However, the unit tests do not verify everything that could potentially go wrong during training. When training deep networks, the code can be correct, but the training may still not converge. This could be because of poor parameter initialization, exploding/vanishing gradients, lack of data normalization, insufficient data, etc.

### 3.5 Submission Instructions

We will test your code using unit tests similar to (but not identical to) the unit tests that are used in this chapter. Thus you must be sure that your code works for all cases, not just the particular cases in the given unit tests. Do not change any of the names of the classes or functions. Also do not change the folder structure of the code as this is assumed by the testing code used for grading. You will be graded based on how many tests your code passes.

Replace the name of the given source folder with `FIRSTNAME-LASTNAME-CH3` and zip the folder for submission. The plots that you generate while performing the hands-on exercises of this chapter should be placed in a folder called `results`.

The grading rubric is shown in Table 3.2 for the hands-on exercises of this chapter.

**Table 3.2:** Grading rubric

| Points    | Description                                                                                                                                             |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| 10        | Correct file names and folder structure (DO NOT submit the <code>cifar-100-python</code> folder)                                                        |
| 15        | <code>full.py</code> implementation that passes all tests                                                                                               |
| 10        | <code>relu.py</code> implementation that passes all tests                                                                                               |
| 15        | <code>softmax.py</code> implementation that passes all tests                                                                                            |
| 15        | <code>cross_entropy.py</code> implementation that passes all tests                                                                                      |
| 15        | <code>sequential.py</code> implementation that passes all tests                                                                                         |
| 5         | Plot of CIFAR100 subset training loss vs epoch for learning rates 0.1, 0.01 and 0.3 for 20 epochs                                                       |
| 5         | <code>acc_results.csv</code> file with results for test accuracy for learning rates 0.1, 0.01, 0.3 and 10 for 20 epochs                                 |
| 10        | <code>run_cifar.py</code> implementation that plots the training loss vs epoch and report the test accuracy (also your observation on learning rate 10) |
| 10        | <code>better_cifar.py</code> that trains and tests a network that achieves better accuracy than the given network                                       |
| Total 110 |                                                                                                                                                         |

# Bibliography

- [1] “Numpy reference.” <https://docs.scipy.org/doc/numpy-1.13.0/reference/>, 2017.
- [2] “Scipy reference.” <https://docs.scipy.org/doc/scipy-1.0.0/reference/>, 2017.
- [3] “Matplotlib pyplot reference.” [https://matplotlib.org/api/ pyplot\\_api.html](https://matplotlib.org/api/ pyplot_api.html), 2017.
- [4] J. Johnson, “Python numpy tutorial.” <http://cs231n.github.io/python-numpy-tutorial/>, 2017.
- [5] “Luma coding in video systems.” [https://en.wikipedia.org/wiki/Grayscale#Luma\\_coding\\_in\\_video\\_systems](https://en.wikipedia.org/wiki/Grayscale#Luma_coding_in_video_systems), 2017.
- [6] Wikipedia, “DFT matrix — Wikipedia, the free encyclopedia.” <http://en.wikipedia.org/w/index.php?title=DFT%20matrix&oldid=811427639>, 2017.
- [7] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- [8] D. R. Cox, “The regression analysis of binary sequences,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 215–242, 1958.
- [9] M. Aly, “Survey on multiclass classification methods,” *Neural Networks*, 2005.
- [10] T. P. Minka, “A comparison of numerical optimizers for logistic regression,” 2003.
- [11] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [12] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [13] H. Yu and S. Kim, “SVM tutorial: classification, regression and ranking,” in *Handbook of Natural computing*, pp. 479–506, Springer, 2012.
- [14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [15] J. Johnson, “Backpropagation for a Linear Layer.” <http://cs231n.stanford.edu/handouts/linear-backprop.pdf>. [Online; accessed 22-Dec-2017].
- [16] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

## BIBLIOGRAPHY

---

- [17] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.
- [18] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.
- [19] A. Krizhevsky, “Learning multiple layers of features from tiny images,” *Technical Report*, 2009.

## Chapter 4

# A Convolutional Neural Network from Scratch

### 4.1 Objectives

- Understand and implement backwards and forwards passes of a convolutional layer
- Understand and implement backwards and forwards passes of a max pooling layer
- Understand and implement backwards and forwards passes of a flattening layer
- Test a convolutional neural network on a subset of the CIFAR100 dataset

### 4.2 Convolutional Layer

In this chapter, we will be implementing convolutional layers and construct a basic convolutional neural network. Why do we need convolutional layers? Theoretically, the fully connected layers can model any function. In practice, learning the function with fully connected layers may be difficult because of the large number of parameters. For example if our input is a  $244 \times 224 \times 3$  image, and our fully connected layer has a modest output size of 500, then we need  $3 \times 224 \times 224 \times 500 \approx 75$  million weights for a single layer. This is clearly not desirable as our dataset probably has much fewer samples than this, so the model will likely have an overfitting problem. One solution to this problem is to utilize *weight sharing*, of which convolutional neural networks is one popular type.

Convolution operations are useful when we expect our data to exhibit translations in some dimensions. Using convolution to recognize entities in images is intuitive because we expect image data to have some degree of translational invariance. For example, if we are trying to detect a car, it would be useful to detect wheels. These wheels could be anywhere in the image, so we would like our wheel detector to be translationally invariant. Before the rise of deep learning, this was often accomplished by a “sliding-window” detector. The convolution operator is analogous to the sliding window approach. The difference is that convolution implements a simple multiply add operation instead of being a full classifier, as in the traditional sliding window approach. As in the fully connected networks, a stacked network of convolutional layers can learn complex functions of the input data.

The convolution operation uses a kernel, or filter, to compute outputs. In convolutional networks for image recognition, the filter size is usually small (e.g.,  $3 \times 3$  or  $7 \times 7$ ). The weights of the filters are learnable parameters. Additionally, the convolutional layer usually incorporates a learnable bias term, which is added to the filter output. As the filter slides across the input, outputs are generated

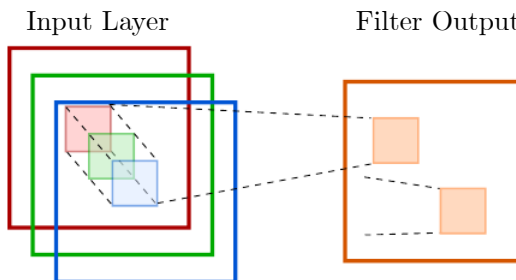


**Figure 4.1:** Example of convolutional layer operation for a single output map.

for each input location. At the borders of the image, input data for the convolution is missing. This data can be replaced with zeros (zero padding), or the convolution output at these locations can be ignored (“valid” convolution). In this chapter we will consider convolutions that slide over 2 dimensions, so the output of our filter will be a 2D map of the outputs corresponding to different locations. It is also possible to do convolutions in 1 dimension (e.g., for audio applications), or convolution in higher dimensions (e.g., applications using 3D point clouds, or video data).

We allow the convolutional layer to have multiple filters, which would give multiple outputs. Often the outputs of the layer are called channels, analogous to image channels because the output channels have the same spatial dimensions. Each filter performs a convolution over the spatial dimensions (row and column). The filter also operates over the channel dimension of the input, but the filter does not “slide” over this dimension. For a first convolutional layer in a network trained on image data, the filters will be RGB filters (assuming image data). For convolutional layers not connected to the input, the input channels represent some intermediate representation learned by the network. Each of the filters has different learnable parameters, but we will combine all of the filter parameters into a single 4 dimensional tensor for each particular layer. Each filter has a separate learnable bias term, so we store the biases in a vector. Figure 4.1 shows a visual example of the convolution operation.

The implementation of the convolutional layer will use the same class interface as we used in Chapter 3. The convolutional layer will be implemented in `conv.py` and the tests are given in `test_conv.py`. You need to implement the `forward`, `backward`, `update_param`, and `__init__` methods.

Again, the convolutional layer will “learn” good filters to use for our particular problem. It is useful to visualize properties of our network so that we gain some intuition as to how the network is working. There are many ways to visualize properties of the network, but the simplest way is to visualize the filter weights. This is easiest to do at the input layer, because the filters can be visualized as color images. Figure 4.2 shows filters from the AlexNet model [20] trained on a large dataset. We can see that these learned filters look like different edge and blob detectors. In this case the filter size is  $3 \times 11 \times 11$ . 3 corresponds to the number of input channels, and  $11 \times 11$  is often referred to as the filter size.

#### 4.2.1 Data and Dimensions

The dimensions of our data are slightly complicated to keep track of, so it is important to be aware of them. If you perform an operation on the wrong dimension, the output will be incorrect. The order of the dimensions is arbitrary; in fact, even different machine learning libraries have different dimension ordering. To pass the tests, your code will need to follow the dimensions listed in this section.

For the variables corresponding to data, we will always leave the first dimension to correspond

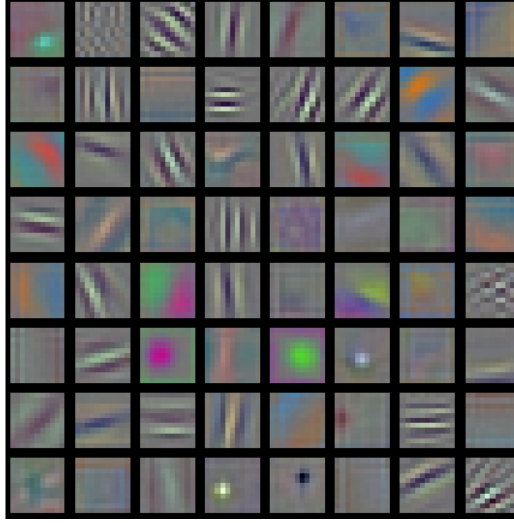


**Figure 4.2: First layer filters from AlexNet.** The first convolutional layer operates on the RGB input image, so we can visualize the filters as color images. We can see that the layer tends to learn Gabor-like filters

to individual data samples. The size of this first dimension will be the size of the batch, since we will process one batch at a time. The next dimension is the channel dimension. By “channels” here we mean the number of input/output filter responses in each layer. For example an input color image has 3 channels. A convolutional layer with 16 filters would result in an output with 16 channels. Each channel can be thought of as a two dimensional map of responses. Finally, the last two dimensions are the rows and columns of this two dimensional map.

Table 4.1 lists the expected sizes of different variables and function outputs of the convolutional layer.

#### 4.2.2 Forward pass

The data passed into a convolutional layer is multidimensional with dimensions  $(n_b, n_i, n_r, n_c)$ , where  $n_b$  is the mini-batch size,  $n_i$  is the number of input channels (also called input depth),  $n_r$  is the number of rows of the input, and  $n_c$  is the number of columns of the input. The output will be of dimension  $(n_b, n_o, n_r, n_c)$ , where  $n_o$  is the number of output filter maps.

If we have  $n_o$  filters of size  $n_i \times h \times h$ , we store the filter weights in a variable  $W$  with shape  $(n_o, n_i, h, h)$ . The channel dimension of the filter is almost always equal to the number of input channels. An example of an exception to this case is AlexNet [20], where input channels are grouped to enable computation on 2 GPUs. Thus, the filter will only move in the two spatial dimensions. In addition, to the convolution weights it is also common to add a bias, as in the fully connected layer. There are two ways to implement the bias: *tied* bias and *untied* bias. For the untied bias, there is a different bias for each corresponding output location. For the tied bias, there is one bias corresponding to each output channel. In this case the bias is a vector of size  $n_o$ . In this chapter, we will implement the tied bias configuration.

For a single output map (indexed by  $o$ ), and a single input sample (indexed by  $i$ ), the convolution layer implements the following:

$$f_{conv_o}(\mathbf{x}_i) = \mathbf{W}_o^{\text{corr}} ** \mathbf{x}_i + \mathbf{b}_o \quad (4.1)$$

Table 4.1: Data dimensions

| Variable                      | Description               | 1st dim              | 2nd dim              | 3rd dim | 4th dim |
|-------------------------------|---------------------------|----------------------|----------------------|---------|---------|
| $\mathbf{x}$                  | Input                     | batch size           | # of input channels  | rows    | columns |
| <code>forward(x)</code>       | Output of forward pass    | batch size           | # of output channels | rows    | columns |
| <code>y_grad</code>           | Gradient at output        | batch size           | # of output channels | rows    | columns |
| <code>backward(y_grad)</code> | Output of backward pass   | batch size           | # of input channels  | rows    | columns |
| $\mathbf{W}$                  | Weight tensor             | # of output channels | # of input channels  | rows    | columns |
| <code>W_grad</code>           | Gradient of weight tensor | # of output channels | # of input channels  | rows    | columns |
| $\mathbf{b}$                  | Bias vector               | 1                    | # of output channels | -       | -       |
| <code>b_grad</code>           | Gradient of bias vector   | 1                    | # of output channels | -       | -       |

where  $\mathbf{x}_i$  is a  $(1, n_i, n_r, n_c)$  shaped input from the previous layer,  $\mathbf{W}_o$  is a single filter corresponding to the  $o$ th output,  $\mathbf{b}_o$  is a single bias corresponding to the output indexed by  $o$ , and  $**$  is the convolution operator.

One thing to be careful with convolution is the behavior at the edges of the input. What is the behavior of convolution if the receptive field overlaps with the border? In *valid* convolution, the outputs of the locations that overlap with the border are ignored. In *full* convolution, we pad the inputs with some value (typically 0) and then we are able to compute outputs corresponding to every input. In our implementation we will be using *full* convolution.

In signal processing, we are taught to flip the filter before we slide the filter across the input. Many machine learning libraries do not perform such flipping, because the flipping makes no difference when training a convolutional neural network. In our implementation, we will be doing convolution without flipping the filter to match the implementation in common machine learning libraries (e.g. PyTorch). We can use the `scipy.signal.correlate` to perform the convolution with flipped filters. If the `scipy.signal.convolve` is used instead, you need to first flip the filters to obtain the same result as `scipy.signal.correlate`.

We will illustrate the desired behavior of the forward pass of the convolutional network with a small example. The input to our layer will be of size 1, 2, 4, 4. This would correspond to a batch with a single image with two channels, and spatial dimensions of  $4 \times 4$ . We will apply 2 filters of size 2, 3, 3, so our weight tensor will be of size 2, 2, 3, 3. The output will be of size 1, 2, 4, 4. Figure 4.3 shows this convolution operation for a fake input without using a bias term.

We could implement the convolution ourselves with loops. It is easier, however, to use a convolution function from one of the included Python libraries. In this case we need to do convolution with flipped filters, also called correlation. We can use the `scipy.signal.correlate` function. Our inputs and outputs have four dimensions (batch, channel, rows, columns), but we only want to do proper convolution for two of the dimensions (rows and columns). If we use the `scipy.signal.correlate` with “full” correlation naively, then we will not get the expected result, because the function will perform zero padding in all dimensions. The “valid” setting does not perform zero padding, but we actually do want zero padding for the row and column dimensions. So we can either zero pad the data in the appropriate dimensions ourselves (using the `np.pad` function), or we can use loops to perform separate convolutions.

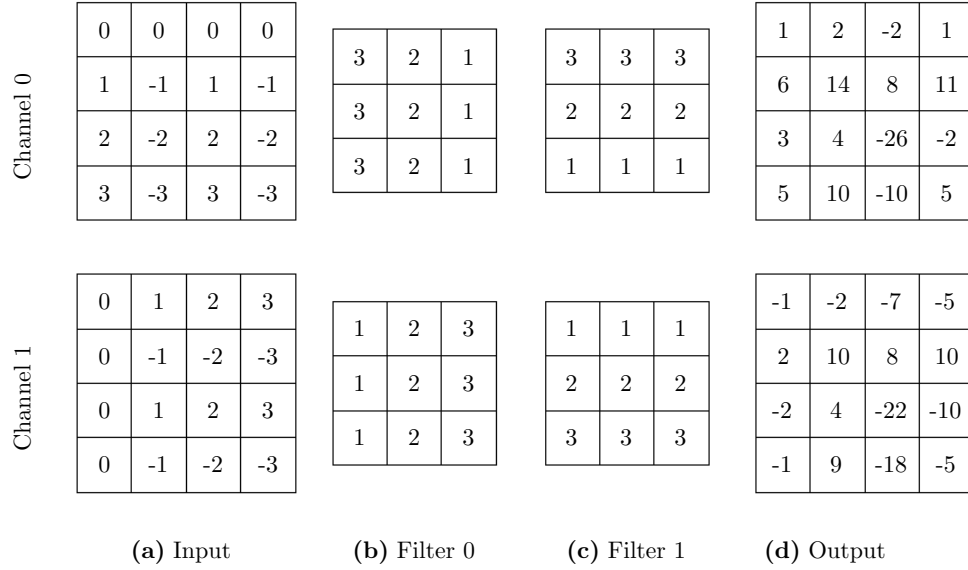


**Figure 4.3: Convolutional forward pass example with no bias term.** This example assumes the filters are already flipped.

### 4.2.3 Backwards Pass

For the backwards pass of the convolutional layer we need to compute  $\frac{d\ell}{d\mathbf{W}}$ ,  $\frac{d\ell}{d\mathbf{b}}$ , and  $\frac{d\ell}{d\mathbf{x}}$ . Recall that  $\ell$  is the loss function at the end of the network. To compute these gradients we assume that we have the gradient at the output of the considered layer  $l$  as  $\frac{d\ell}{d\mathbf{x}^{l+1}}$ . Similar to backpropagation in the previous chapter, the **backward** function has an input **y\_grad** that represents the gradient of the output of the neural network with respect to the output of the current layer.

First we will start with the easiest case: the gradient of **b**. This gradient will be stored in the class variable **b\_grad**. Similar to the fully connected case, **b\_grad** does not depend on **x** or **W**. We can compute **b\_grad** only with the gradient at the output **y\_grad**. Recall that each output map corresponds to a different element of the vector **b**. For backpropagation, we would need to sum the values of all of the gradient at the output **y\_grad** values across the row, column, and batch dimensions. We do not need to sum over the output filter dimension  $n_o$  because there is a different value of **b** for each output filter. If the sum is done correctly, we should be left with a variable that is the same shape as **b**.

Deriving the gradients for **W** and **x** is more difficult than the fully connected case, because the convolution operation produces different output values for each output location. However, we can view a convolutional layer as a set of masked fully connected layers with weight sharing enforced. For example, consider one filter location. The output at one location can be expressed as  $\mathbf{x}'\mathbf{W}'^T$ , where  $\mathbf{x}'$  is a row vector of the flattened data needed for one filter location, and  $\mathbf{W}'$  is a row vector of the flattened kernel. We know from Chapter 3 how to compute the backwards pass for this function. The only difference here is that the filter is sliding, so we need to sum over filter locations that are overlapping. This sum over locations operation can be expressed as a convolution.

Now we will consider  $\frac{d\ell}{d\mathbf{x}}$ . Every channel of the input affects every channel of the output. We can consider the contribution of each output channel to the input channels separately. We will use  $c_{out}$  to represent a particular output channel. The gradient due to a particular output channel  $\frac{d\ell}{d\mathbf{x}_{c_{out}}}$  can be computed as a convolution between the output channel gradient  $\frac{d\ell}{d\mathbf{x}_{c_{out}}^{l+1}}$ , and the weight matrix

for that channel  $\mathbf{W}_{c_{out}}$ . For this operation, we use a normal convolution instead of a correlation, because we are going in the “opposite” direction as the original operation. To obtain the actual gradient  $\frac{d\ell}{d\mathbf{x}}$ , we can sum the gradients due to each channel.

$$\frac{d\ell}{d\mathbf{x}} = \frac{d\ell}{d\mathbf{x}_{c_{out}^{l+1}}} ** \mathbf{W}_{c_{out}} \quad (4.2)$$

For computing  $\frac{d\ell}{d\mathbf{W}}$  it is easiest to consider each input/output pair at a time. For a particular input/output pair,  $\frac{d\ell}{d\mathbf{W}}$  is equal to the correlation between an input channel of  $\mathbf{x}$  and an output channel of  $\frac{d\ell}{d\mathbf{x}_{c_{out}^{l+1}}}$  (**y-grad**). For this convolution we assume that the input  $\mathbf{x}$  is zero padded in the appropriate dimensions. A correlation with the padded  $\mathbf{x}$  and  $\frac{d\ell}{d\mathbf{x}_{c_{out}^{l+1}}}$  should give an output which is the same size as  $\mathbf{W}$ .

$$\frac{d\ell}{d\mathbf{W}_{j,k}} = \mathbf{x}_{channel=j} \overset{\text{corr}}{**} \frac{d\ell}{d\mathbf{x}_{c_{out}^{l+1}}}_{channel=k} \quad (4.3)$$

For these examples, we have left out an index  $i$ , corresponding to the sample index. As in Chapter 3, all of the functions need to operate on a batch of data.

#### 4.2.4 Initialization

As discussed in Chapter 3, the initial values for the layer parameters must be set properly to achieve good network convergence. For this convolutional layer, we use same initialization strategy of Glorot [16] as we did for the fully-connected layer in Chapter 3. The initialization in our network takes the form:

$$\mathbf{W} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{f_{in} + f_{out}}}\right) \quad (4.4)$$

$$\mathbf{b} \sim \mathbf{0} \quad (4.5)$$

where

$$f_{in} = n_i \times h \times h \quad (4.6)$$

$$f_{out} = n_o \times h \times h. \quad (4.7)$$



#### Deliverable: conv.py

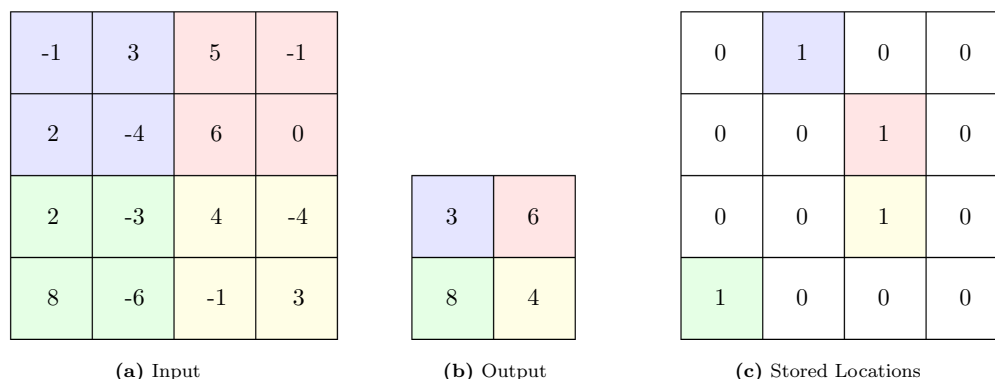
Submit a `conv.py` file that passes all of the tests in the `test_conv.py` file.

#### Extra Information: Convolution Speed

The speed of the convolution is often the bottleneck when training convolutional neural networks. Our implementation of the convolutional layer is likely to be several times slower than other library implementations, even when the other library is run on the CPU. Most libraries achieve this speedup by reformulating convolution as a matrix multiplication operation. As a matrix multiplication, efficient linear algebra libraries can be used for the convolution. Other methods for achieving faster convolutions include the Winnograd method [21], and FFT-based convolutions.

#### 4.2.5 Update parameters

The `update_param` function is identical to that of the previous chapter. The primary difference in this case is that our variables and gradients have more dimensions than the previous chapter.

Figure 4.4: Maxpool forward pass for  $2 \times 2$  max pooling.

### 4.3 Max Pooling Layer

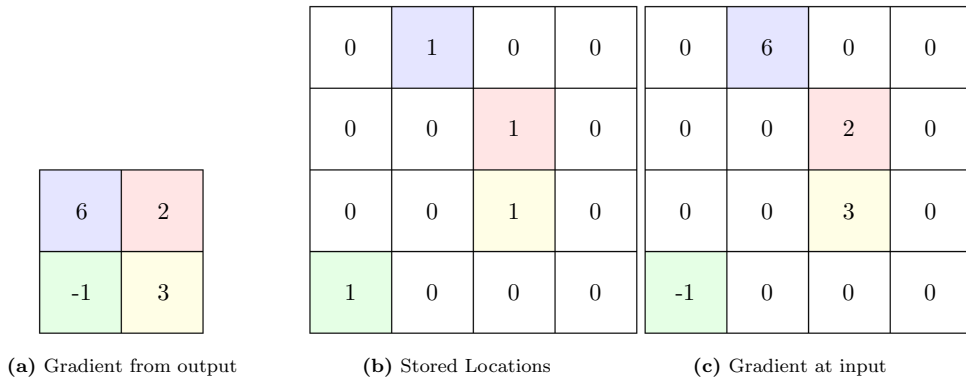
A network that incorporates convolutional layers often uses Max Pooling layers as well. The max pooling operation simply takes the maximum activation over small regions in the input. For example, we could take the maximum value of each input over  $2 \times 2$  sized spatial regions. There are two benefits to this operation. First, max pooling can add some robustness because it eliminates small activations and only takes the largest response in a spatial region. Another practical benefit of max pooling is that it can reduce the spatial dimensions of the inputs, which is similar to downsampling. The reduced spatial dimensions allow later convolutional layers to have a greater effective receptive field with respect to the input.

**Forward pass** We assume that the max pooling layer takes the maximum response using  $k \times k$  sized square regions. There is no need to restrict the region to be square, however in your implementation you can assume that the region is square. Like in convolution, this region will move across the input to generate an output map. In general this region could move with some step size, which is often called the *stride*. For example unless a stride other than 1 is specified, in a typical convolution operation, the stride is one, meaning that the filter moves along every pixel. For the max pooling operation we won't use a stride of one. Instead we will use a stride that is of size  $k$ , the same size as our region. With this stride, the max pooling regions do not overlap. Figure 4.4 shows an example of this strided max pooling operation with  $k = 2$ .

The examples in Figure 4.4 assume a one channel input (and a one-channel output). In a typical network there will likely be an input with multiple channels. The max pooling layer should find the max in the region for each channel separately. The number of channels of the output should be the same as the number of channels of the input.

It is possible that the max pooling regions overlap with the edge of the input map. For example if the size of a dimension of the input map is odd, and the size of the max pooling region is  $2 \times 2$ , there will be a "left-over" position at the end. We will follow the convention of some machine libraries which ignore these left over positions.

**Backward pass** For the backward passes of the max pooling layers, we simply need to propagate the gradients to the positions that had the maximum activation value. In order to do this, we need to store the location of the maxima during the forward pass. Since the max pooling layer does not perform any arithmetic operations, the output gradient is passed exactly to the locations of the


 Figure 4.5: Maxpool backwards pass for  $2 \times 2$  max pooling.

maxima. For inputs that did not correspond to a maximum the gradient is 0. Figure 4.5 shows an example of this operation for  $k = 2$ .

For max pooling, it is possible that two (or more) inputs are both the max. In this case, max pooling should propagate the gradient for both of the inputs that were the max. This can occur often in neural networks, because the ReLU will force many inputs to be 0.



#### Deliverable: maxpool.py

Submit a `maxpool.py` file that passes all of the tests in the `test_maxpool.py` file.

## 4.4 Flatten Layer

Many network architectures are a combination of convolutional layers and fully connected layers. However the output of our convolutional layer is not compatible with the input of the fully connected layer from Chapter 3. The convolutional layer output is of size  $(n_b, n_o, n_r, n_c)$  whereas the fully connected layer input should be of size  $(n_b, n_i)$ . To correct for this we create a layer called `FlattenLayer` that “flattens” the last three dimensions of the convolutional layer such that the output of the `FlattenLayer` is of size  $(n_b, n_o \times n_r \times n_c)$ .

For the backwards pass, we simply need to apply the “inverse” flattening operation on the gradient. There are no learnable parameters or gradient calculations, it is simply another reshape operation. To perform the reshape for the forwards/backwards passes we can use the `np.reshape` function.

#### Hint: Tips in using `np.reshape` function

`np.reshape(x)` does not generate another instance of the variable `x`, but rather it generates a pointer to the variable `x`. Thus, if `y = np.reshape(x)`, changing the value of `y` will change the value of `x`. One solution is to use the `np.copy` function with the `reshape` function.



#### Deliverable: flatten.py

Submit a `flatten.py` file that passes all of the tests in the `test_flatten.py` file.

## 4.5 Putting it all together again

Finally, we will revisit training a neural network on a subset of the CIFAR-100 dataset. This time we will be using convolutional layers and maxpooling, in addition to ReLU and fully connected layers from the previous chapter.

### Important: Location for `cifar-100-python` folder:

You can download the CIFAR100 dataset using same link from Chapter 3:

<https://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz>

But different from Chapter 3, in this chapter, after downloading the `cifar-100-python` folder, put it outside the `src` folder but in the same folder as the `src` folder.

We can reuse all of the layers from the previous chapter. The only thing that might have to be modified is the ReLU layer, as the ReLU layer should be able to handle inputs from a fully connected layer, or a convolutional layer. Since the layers from the previous chapter and this chapter follow the same class interface, we can plug them into the same `Sequential` model that we developed in the previous chapter. We could continue to develop different types of layers with the same interface and plug them into this model structure.

To classify CIFAR-100 images in the last chapter, the data was flattened so that each data sample was a vector of pixel data. Here we do not want to flatten the data, because we want to use convolutional layers, which assumes that the input data has channels, rows, and columns. We have provided an updated `dataset.py` that provides the CIFAR-100 data in this form. The subset of 3 classes is determined using the seed parameter of the `cifar100` function. **Set this seed to your student ID number to get the subset of the data.**

We will build a simple network with 2 convolutional layers and one fully connected layer. Figure 4.6 shows the architecture of this model. To keep the model from running too slow we will use small  $3 \times 3$  sized filters. The small filter size is not a huge classification performance bottleneck, as high performing networks can be built entirely using  $3 \times 3$  sized filters (e.g. [22]). Also to make training fast, we limit the number of filters in the convolutional layers to 16 and 32 units. After each convolutional layer, we apply a ReLU nonlinearity and a max pooling operation. The fully connected layer has an input size of 2048 which is equal to the size at the output of the second max pooling layer ( $32 \times 8 \times 8$ ). The final fully connected layer has 3 outputs, corresponding to the 3 classes.

Training may be slow due to the convolutional operation. So in this chapter we do not require plots for different learning rates. However, we do require a plot of the training and validation loss for a learning rate of 0.1 for 15 epochs and a batch size of 128.

### Deliverable: CIFAR100 ConvNet Results

Submit a script called `run_cifar100.py` that plots the average training loss vs epoch and that reports the final accuracy on the test set (please write the accuracy in the commented section at the beginning of the script).

Submit the plot of the average training loss vs epoch using a learning rate of 0.1 for 15 epochs and a batch size of 128.



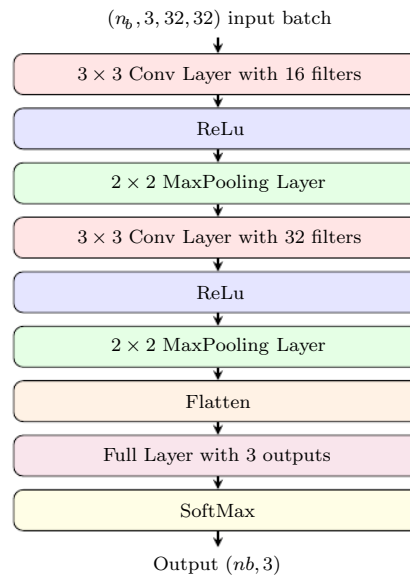


Figure 4.6: Network architecture.

## 4.6 Submission Instructions

We will test your code using unit tests similar to (but not identical to) the unit tests that we used in this chapter. Thus you must be sure that your code works for all cases, not just the particular cases in the given unit tests. Do not change any of the names of the classes or functions. Also do not change the folder structure of the code as this is assumed by the testing code used for grading. You will be graded based on how many tests your code passes.

Replace the name of the given source folder with `FIRSTNAME_LASTNAME_CH4` and zip the folder for submission. The plots that you generated while performing the hands-on assignments in this chapter should be placed in a folder called `results`.

The grading rubric for this lab is shown in Table 4.2.

Table 4.2: Grading rubric

| Points    | Description                                                                                      |
|-----------|--------------------------------------------------------------------------------------------------|
| 10        | Correct file names and folder structure (DO NOT submit the <code>cifar-100-python</code> folder) |
| 30        | <code>conv.py</code> implementation that passes all tests                                        |
| 30        | <code>pool.py</code> implementation that passes all tests                                        |
| 15        | <code>flatten.py</code> implementation that passes all tests                                     |
| 8         | Plot of CIFAR100 subset training loss saved in <code>loss_plot.png</code>                        |
| 7         | <code>run_cifar100.py</code> that plots the training loss and report the accuracy                |
| Total 100 |                                                                                                  |

# Bibliography

- [1] “Numpy reference.” <https://docs.scipy.org/doc/numpy-1.13.0/reference/>, 2017.
- [2] “Scipy reference.” <https://docs.scipy.org/doc/scipy-1.0.0/reference/>, 2017.
- [3] “Matplotlib pyplot reference.” [https://matplotlib.org/api/pyplot\\_api.html](https://matplotlib.org/api/pyplot_api.html), 2017.
- [4] J. Johnson, “Python numpy tutorial.” <http://cs231n.github.io/python-numpy-tutorial/>, 2017.
- [5] “Luma coding in video systems.” [https://en.wikipedia.org/wiki/Grayscale#Luma\\_coding\\_in\\_video\\_systems](https://en.wikipedia.org/wiki/Grayscale#Luma_coding_in_video_systems), 2017.
- [6] Wikipedia, “DFT matrix — Wikipedia, the free encyclopedia.” <http://en.wikipedia.org/w/index.php?title=DFT%20matrix&oldid=811427639>, 2017.
- [7] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- [8] D. R. Cox, “The regression analysis of binary sequences,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 215–242, 1958.
- [9] M. Aly, “Survey on multiclass classification methods,” *Neural Networks*, 2005.
- [10] T. P. Minka, “A comparison of numerical optimizers for logistic regression,” 2003.
- [11] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [12] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [13] H. Yu and S. Kim, “SVM tutorial: classification, regression and ranking,” in *Handbook of Natural computing*, pp. 479–506, Springer, 2012.
- [14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [15] J. Johnson, “Backpropagation for a Linear Layer.” <http://cs231n.stanford.edu/handouts/linear-backprop.pdf>. [Online; accessed 22-Dec-2017].
- [16] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

## BIBLIOGRAPHY

---

- [17] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.
- [18] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.
- [19] A. Krizhevsky, “Learning multiple layers of features from tiny images,” *Technical Report*, 2009.
- [20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, pp. 1097–1105, 2012.
- [21] A. Lavin and S. Gray, “Fast algorithms for convolutional neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.
- [22] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

## Chapter 5

# Transfer Learning and Computer Vision in PyTorch

### Objectives

- Learn PyTorch Basics
- Extract features from pre-trained networks and train SVM classifiers using the features
- Fine-tune the networks to classify on different datasets

### 5.1 PyTorch

PyTorch [23] is an open source machine learning library that is particularly useful for deep learning. PyTorch contains auto-differentiation, meaning that if we write code using PyTorch functions, we can obtain the derivatives without any additional derivation or code. This saves us from having to implement any backwards functions for deep networks. Secondly, PyTorch comes with many functions and classes for common deep learning layers, optimizers, and loss functions. Lastly, PyTorch is able to efficiently run computations on either the CPU or GPU.

#### 5.1.1 Installation

To install PyTorch run the following commands in the Linux terminal:

```
pip install https://download.pytorch.org/whl/cpu/torch-1.0.1.post2-cp27-cp27mu-linux_x86_64.whl
pip install torchvision
```

The first command installs the basic PyTorch package, and the second installs `torchvision` which includes functions, classes, and models useful for deep learning for computer vision.

#### 5.1.2 Tutorial

To start we ask you to complete the following tutorial: [http://pytorch.org/tutorials/beginner/pytorch\\_with\\_examples.html](http://pytorch.org/tutorials/beginner/pytorch_with_examples.html)

## 5.2 CIFAR100 Example in PyTorch

Next as an example, we will re-implement the neural network from Chapter 4 using PyTorch instead of the library we built. The code for this example is in the included `cifar_pytorch.py` file.

PyTorch has a module called `nn` that contains implementations of the most common layers used for neural networks. If you look at the code for some layer (for example `linear.py`), you will see that it is similar to our implementation from Chapter 3. There is a `forward` method and an `__init__` method. There is no need for a `backward` method because PyTorch uses automatic differentiation.

In this example, we will implement our model as a class with `forward`, `__init__`, `fit` and `predict` functions. The initialization function simply sets up our layers using the layer types in the `nn` package. In the forward pass we pass the data through our layers and return the output. Note that we can reuse the pool layer, since this layer has no learnable parameters. Also instead of a ReLU layer, we can use the `F.relu` function in our forward pass. Similarly, instead of a “flatten” layer, we can just use PyTorch’s `view` to reshape the data.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # Conv2d args: (input depth, output depth, filter size)
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.conv2 = nn.Conv2d(16, 32, 3)
        # Linear args: (# of input units, # of output units)
        self.fc1 = nn.Linear(1152, 5)
        # MaxPool2d args: (kernel size, stride)
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 1152) # -1 means shape is inferred
        x = self.fc1(x)
        return x
```

**Fit function** Next we will implement a fit function. Here we implement the fit function as a class method, similar to Chapter 3. It is also common to see the code for training be implemented outside of the model class in a separate function. The fit function is very similar to our own fit function from Chapter 3. First we set an optimization criterion, and an optimizer. Next we loop for a number of epochs. In each epoch, we use PyTorch’s `DataLoader` class to loop through the data in batches. The `DataLoader` automatically takes care of splitting the data into batches. We access the batches with a simple `for` loop through the `DataLoader`. For each batch we zero the previously calculated gradients using the optimizers `zero_grad` method. Then we call the forward function, compute the loss, call the backwards function, and perform one optimization step. The optimization step is similar to the `update_params` methods we used in previous chapters.

```
def fit(self, trainloader):
    # switch to train mode
    self.train()

    # define loss function
    criterion = nn.CrossEntropyLoss()

    # setup SGD
    optimizer = optim.SGD(self.parameters(), lr=0.1, momentum=0.0)

    for epoch in range(20): # loop over the dataset multiple
        times
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # get the inputs
            inputs, labels = data

            # zero the parameter gradients
            optimizer.zero_grad()

            # compute forward pass
            outputs = self.forward(inputs)

            # get loss function
            loss = criterion(outputs, labels)

            # do backward pass
            loss.backward()

            # do one gradient step
            optimizer.step()

            # print statistics
            running_loss += loss.item()

        print('[Epoch: %d] loss: %.3f' %
              (epoch + 1, running_loss / (i+1)))
```

Finally it is also useful to provide a predict function to run our model on some test data. This predict function will also use the PyTorch loader.

```

def predict(self, testloader):
    # switch to evaluate mode
    self.eval()

    correct = 0
    total = 0
    all_predicted = []
    with torch.no_grad():
        for images, labels in testloader:
            outputs = self.forward(Variable(images))
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            all_predicted += predicted.numpy().tolist()

    print('Accuracy on test images: %d %%' % (
        100 * correct / total))

    return all_predicted

```

Notice that we use `torch.nn.module.train()` for training and `torch.nn.module.eval()` for prediction. Function `train()` sets the module in training mode and function `eval()` sets the module in evaluation mode. These two functions have an effect only on certain modules. They control certain layers like `Dropout` and `BatchNorm` to be developed during training and disabled during evaluation. For simple neural network structures like in Chapter 3 and Chapter 4, we do not have any layers that will be affected by `self.train()` or `self.eval()`; so it is okay for you to delete them. But it is a good habit to always add them when implementing a neural network.

We also use `torch.no_grad()` when doing prediction. This function was introduced starting with PyTorch version 0.4. It disables the gradient calculation. You can use it when you are sure that you will not call the backward function. This will reduce memory consumption and computations consumption.

To evaluate the two models we look at both the final classification accuracy as well as the *confusion matrix*. The rows of the confusion matrix show the predicted class and the columns show the actual class. This lets us analyze the patterns of missclassifications. The included `util.py` includes the function `plot_confus_matrix` which will plot this matrix. Figure 5.1 shows the confusion matrix for this CIFAR100 example.

## 5.3 Transfer Learning

*Transfer learning* is when we use a model trained on one set of data, and adapt it to another set of data. For image datasets, transfer learning works because many features (e.g. edges) are useful across different image datasets. Transfer learning using neural networks trained on large image datasets is highly successful, and can easily be competitive with other approaches that do not use deep learning. Transfer learning also does not require a huge amount of data, since the pre-trained initialization is a good starting point.

In this chapter, we will consider two types of transfer learning: a feature-extraction based method and a fine-tuning based method. We will be using networks that have been pre-trained on the ImageNet dataset [24], and adapt them for different datasets.

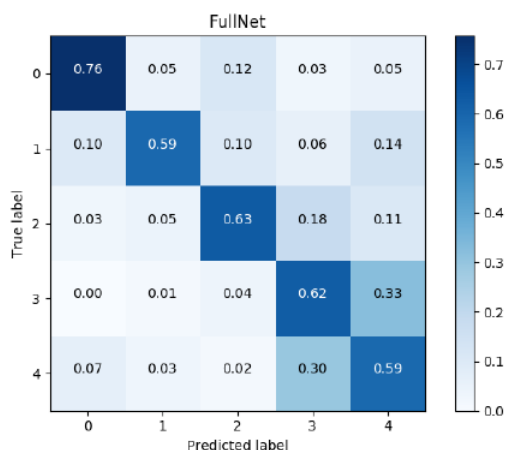


Figure 5.1: CIFAR100 confusion matrix.

### 5.3.1 Datasets

We have collected five datasets for use in this chapter (Table 5.1). You will be assigned one of these datasets to work on. The datasets are subsets of existing datasets. Figure 5.2 shows example images from these datasets. You will not get credit if you download and submit a model trained on the full version of these datasets. We want to fine-tune models that were originally trained on the ImageNet dataset.

With transfer learning we alleviate some of the problems with using small datasets. Typically if we tried to train a network from scratch on a small dataset, we might experience overfitting problems. But in transfer learning, we start with some network trained on a much larger dataset. Because of this, the features from the pre-trained network are not likely to overfit our data, yet still likely to be useful for classification.

Table 5.1: Datasets for Transfer Learning in PyTorch

| Dataset    | Description                            | # Categories | # Train / # Test |
|------------|----------------------------------------|--------------|------------------|
| Animals    | iNat2017 challenge [25]                | 9            | 1384 / 466       |
| Faces      | Labeled faces in the wild dataset [26] | 31           | 1021 / 439       |
| Places     | Places dataset [27]                    | 9            | 1437 / 367       |
| Household  | iMaterialist 2018 challenge [28]       | 9            | 1383 / 375       |
| Caltech101 | CVPR 2004 Workshop [29]                | 30           | 1500 / 450       |

### 5.3.2 Base Networks

Table 5.2 shows the base networks that we will consider in this chapter. All of these networks have PyTorch versions trained on the ImageNet dataset [24], but the architectures of the networks vary. The input size and the last layer input size also vary among the networks.

As an example we will be using DenseNet [30] to explain how to do fine-tuning in PyTorch. In PyTorch we can load a pre-trained DenseNet model with the command:

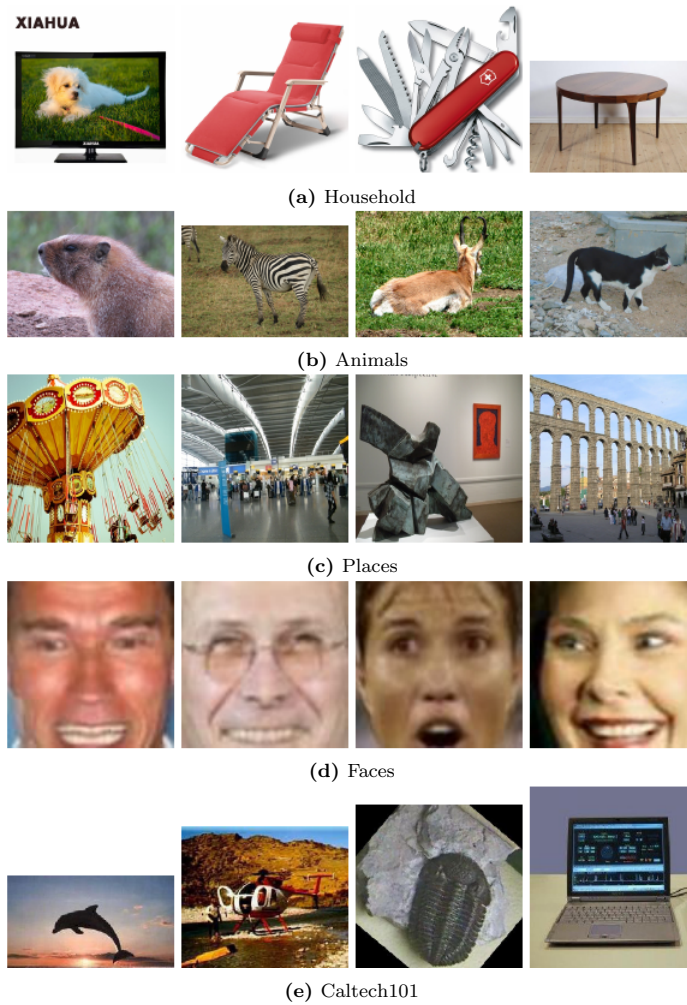


**Figure 5.2:** Example images from datasets for transfer learning in PyTorch.

```
import torchvision.models
model = torchvision.models.densenet121(pretrained=True)
```

It is important to use the `pretrained=True` argument. Otherwise, the model will be initialized with random weights.

**Table 5.2:** Base Networks for Transfer Learning in PyTorch

| Model             | Year | Input Size       | Last layer input size | PyTorch model                                |
|-------------------|------|------------------|-----------------------|----------------------------------------------|
| AlexNet [20]      | 2012 | $224 \times 224$ | 4096                  | <code>torchvision.models.alexnet</code>      |
| VGG16 [22]        | 2014 | $224 \times 224$ | 4096                  | <code>torchvision.models.vgg16</code>        |
| ResNet18 [18]     | 2016 | $224 \times 224$ | 512                   | <code>torchvision.models.resnet18</code>     |
| Inception v3 [31] | 2015 | $299 \times 299$ | 2048                  | <code>torchvision.models.inception_v3</code> |
| DenseNet121 [30]  | 2017 | $224 \times 224$ | 1024                  | <code>torchvision.models.densenet121</code>  |

### 5.3.3 Pre-processing

It is important that we pre-process the images before sending them to the network. Most DNNs preprocess the images to be zero mean and unit standard deviation before training. During testing if we pass an image that is also not normalized (with respect to the training data), then the network output won't be useful. We can say that the network “expects” the input to be normalized. PyTorch includes a `transform` module that implements the common transformations, including normalization, used in pre-processing:

```
import torchvision.transforms as transforms
```

For normalization we can utilize the built in PyTorch function `Normalize`. The values used for normalization can be computed from the images in the ImageNet dataset. For each channel in the image there is a separate mean and standard deviation used for normalization.

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                std=[0.229, 0.224, 0.225])
```

To apply our data with our pretrained neural network, we must resize the images to the input size expected by the network. For Alexnet, VGG16, ResNet18, and DenseNet this is  $224 \times 224$  pixels, but for Inception this is  $299 \times 299$  pixels. There are different ways to ensure our input is the correct size. The simplest way is to resize our input to the correct size. This has the disadvantage of potentially stretching the image if there is an aspect ratio mismatch. An alternative method is to resize the image so that the minimum side length is equal to the required size, and then crop out the extra part of the image to get the desired size. For this tutorial, we use the first method with PyTorch's `Resize` method:

```
resize = transforms.Resize((224, 224))
```

In preprocessing we would like to apply these transformations in a pipeline to every image. PyTorch includes a useful function called `Compose` to combine the transformations into a single object representing the pipeline:

```
preprocessor = transforms.Compose([
    resize ,
    transforms.ToTensor() ,
    normalize ,
])
```

Note the presence of `ToTensor()` which is needed to convert from the image representation to the PyTorch tensor representation. We can easily add other transformations to this preprocessor object to do more preprocessing, or to add data augmentation.

### 5.3.4 Feature Extractor + SVM

First we will use the base network as a feature extractor. This means that we simply run the images through the pre-trained base network, and take outputs from layer(s) as a feature representation of the image. These features are usually good for classification with a shallow machine learning algorithm. In this case we will use an SVM for classification.

To extract features we need to stop the forward pass after a certain layer. As of writing this tutorial, PyTorch doesn't offer a simple call to extract a certain layer. However it is very easy to change the model to give the output of a certain layer. How exactly to extract a certain layer depends on the implementation of the model. We recommend you look at the source code for the networks available at <https://github.com/pytorch/vision/tree/master/torchvision/models>.

The PyTorch AlexNet and VGG models are split into a feature extractor stage, and a classifier stage. The feature extractor consists of the convolutional layers, and the classifier consists of the fully connected layers. As an example, we want to extract the output of the layer before the last fully connected layer. The easiest way to do this is to modify the sequential part of the model to remove the second to last layer.

```
new_classifier = nn.Sequential(*list(model.classifier.children())
                              [:-1])
model.classifier = new_classifier
```

If we are using a model that does not use the feature extractor/classifier decomposition, we need to modify the forward pass of the model to only compute up to the requested layer. For example, for DenseNet, we can comment out the last layer to extract the features before the final fully connected layer.

```
def forward(self, x):
    features = self.features(x)
    out = F.relu(features, inplace=True)
    # average pool features and reshape to (batch size, feature size)
    out = F.avg_pool2d(out, kernel_size=7, stride=1).view(features.
        size(0), -1)
    # out = self.classifier(out) # commented out to get the features
    instead
    return out
```

Next we need some way to load the data from our dataset. Again PyTorch provides some convenient tools to do this. We will use the `datasets.ImageFolder` class to load our dataset. The `ImageFolder` loader assumes a file structure of our data where each of the classes is stored in a separate folder. Next we will use the `torch.utils.data.DataLoader` class to create a loader that

can be used to loop through our data with some batch size. We would need to have separate loaders for the training data and the testing data. A loader can be constructed with:

```
loader = torch.utils.data.DataLoader(
    datasets.ImageFolder(data_dir, preprocessor),
    batch_size=batch_size,
    shuffle=True)
```

With the loader set, we can now loop through the data and extract features. During looping we can use Python's `enumerate` function to keep track of the batch index. The loader returns a tuple with the data and the target label.

In the case of testing, we don't need to feed the label to the network, but it would be useful to save the label so that we can use it later for computing the SVM classification accuracy for the test set. To use the input data in our PyTorch model, we need to wrap it as a PyTorch `Variable`.

```
for i, (in_data, target) in enumerate(loader):
    input_var = torch.autograd.Variable(in_data, volatile=True)
    output = model(input_var)
```

With the extracted features for each sample, we use Sci-kit learn's SVM model. To review how to use the model, revisit Chapter 2. It is up to you to determine the type of SVM and best hyper parameters.

We can assess the accuracy of the SVM model using the classification accuracy on the entire testing set. In addition to this, we can plot a confusion matrix, which tells more information about the errors that the model made. `util.py` includes the function `plot_confus_matrix` which will plot this matrix.

#### Important:

The `plot_confus_matrix` function plots a confusion matrix. Note that there is an input parameter `size` to indicate how many classes you want to show in the plot. The default value for `size` is `None` in which case it will plot the whole confusion matrix. If `size` is given some integer value, for example 9, it will plot the confusion matrix from class 0 to class 8. To make the result more clear to observe for large datasets, such as Faces and Caltech101, please plot the confusion matrix for the first 9 classes. For the other three datasets, use the default setting to plot the whole confusion matrix.

### 5.3.5 Fine-tuning

The feature extractor + SVM approach already may give decent results for your problem. This is because, for certain problems, the intermediate representations learned by the pre-trained network can be very useful for the new problem. However, even if the pre-trained filters are giving good performance, we may be able to achieve even greater performance by allowing the parameters of the pre-trained model to adapt to our new dataset. This adaptation process is called fine-tuning.

Performing fine-tuning is exactly the same process as performing training. Refer to the included `cifar_pytorch.py` to see how to train in PyTorch. The main differences in this case is that we want to start from the pre-trained model. We can use the same `DataLoader` and `transform` modules that we used for feature extraction. During fine-tuning we will usually use a very small learning rate. We want to adapt the existing filters to our data, but not move the parameters so far from the pre-trained parameters.

During fine-tuning we can speed up the process by running the model on the GPU. In PyTorch this can be accomplished by using the `.cuda()` command on the model and loss function. For example:

```
model = model.cuda()
criterion = criterion.cuda()
```

Finally, after training, we would like to save the model so that we can use it in the future without training again. We can use the `model.save(filename)` function to save the model.

For the hands-on exercises in this chapter, we do not specify the hyper parameters for you to use. It is up to you to choose a good set of hyper parameters. Examples of hyper parameters that can be changed are learning rate, batch size, and number of epochs.

**Data Augmentation** To achieve higher performance, we can experiment with data augmentations. Data augmentation is the process of slightly perturbing the input images to generate more samples than were originally available. This data augmentation could be image rotation, scale, gray scale transformation, etc.

PyTorch includes transformations useful for data augmentation in the `torchvision.transforms` module. As part of the deliverables, you need to add some of these data augmentation methods to your preprocessing object to achieve greater test accuracy.

#### Deliverable: Transfer Learning with PyTorch

Depending on your task number, you are assigned a different dataset and model to perform transfer learning (Table 5.3).

- Provide the test accuracy and confusion matrices for the considered networks as feature extractors for the dataset; name the confusion matrix plot as `conf_feature.png`
- Provide the test accuracy and confusion matrices for the considered fine-tuned network for the dataset; name the confusion matrix plot as `conf_finetune.png`
- Submit the trained model saved using the `model.save` function
- Submit code for both the feature extraction based method and the fine-tuning based method named as `feature.py` and `finetune.py`
- Submit the code for the fine-tuning based method with data argumentation, as well as the saved trained model



## Additional Resources

- PyTorch Documentation - <http://pytorch.org/docs/stable/index.html>
- TorchVision Documentation - <http://pytorch.org/docs/master/torchvision/index.html>
- PyTorch tutorials - <http://pytorch.org/tutorials/>
- PyTorch Github - <https://github.com/pytorch/pytorch>
- TorchVision Github - <https://github.com/pytorch/vision>
- PyTorch Discussion Forum - <https://discuss.pytorch.org/>

## Task Assignment

**Table 5.3:** Sample tasks for transfer learning in PyTorch

| Task number | Dataset    | Model     |
|-------------|------------|-----------|
| 1           | Animals    | AlexNet   |
| 2           | Animals    | Inception |
| 3           | Animals    | ResNet18  |
| 4           | Animals    | VGG16     |
| 5           | Places     | AlexNet   |
| 6           | Places     | Inception |
| 7           | Places     | ResNet18  |
| 8           | Places     | VGG16     |
| 9           | Faces      | AlexNet   |
| 10          | Faces      | Inception |
| 11          | Faces      | ResNet18  |
| 12          | Faces      | VGG16     |
| 13          | Household  | AlexNet   |
| 14          | Household  | Inception |
| 15          | Household  | ResNet18  |
| 16          | Household  | VGG16     |
| 17          | Caltech101 | AlexNet   |
| 18          | Caltech101 | Inception |
| 19          | Caltech101 | ResNet18  |
| 20          | Caltech101 | VGG16     |

## Submission Instructions

Submit your work as a folder named `FIRSTNAME_LASTNAME_TASKNUMBER_CH5` and zip the folder for submission. The plots and saved models that you generated while performing the hands-on assignments in this chapter should be placed in a folder called `results`. The grading rubric is shown in Table 5.4. There are no unit tests for the submitted code, but your code will be graded by attempting to run it.

**Table 5.4:** Grading rubric

| Points    | Description                                                                  |
|-----------|------------------------------------------------------------------------------|
| NOTE:     | DO NOT SUBMIT THE DATASET                                                    |
| 35        | Working code for feature extraction based method ( <code>feature.py</code> ) |
| 35        | Working code for fine-tuning based method ( <code>finetune.py</code> )       |
| 10        | Fine-tuning code using data augmentation                                     |
| 10        | Test accuracies and confusion matrices for the assigned dataset and model    |
| 10        | Submit saved models (two in total)                                           |
| Total 100 |                                                                              |

# Bibliography

- [1] “Numpy reference.” <https://docs.scipy.org/doc/numpy-1.13.0/reference/>, 2017.
- [2] “Scipy reference.” <https://docs.scipy.org/doc/scipy-1.0.0/reference/>, 2017.
- [3] “Matplotlib pyplot reference.” [https://matplotlib.org/api/pyplot\\_api.html](https://matplotlib.org/api/pyplot_api.html), 2017.
- [4] J. Johnson, “Python numpy tutorial.” <http://cs231n.github.io/python-numpy-tutorial/>, 2017.
- [5] “Luma coding in video systems.” [https://en.wikipedia.org/wiki/Grayscale#Luma\\_coding\\_in\\_video\\_systems](https://en.wikipedia.org/wiki/Grayscale#Luma_coding_in_video_systems), 2017.
- [6] Wikipedia, “DFT matrix — Wikipedia, the free encyclopedia.” <http://en.wikipedia.org/w/index.php?title=DFT%20matrix&oldid=811427639>, 2017.
- [7] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- [8] D. R. Cox, “The regression analysis of binary sequences,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 215–242, 1958.
- [9] M. Aly, “Survey on multiclass classification methods,” *Neural Networks*, 2005.
- [10] T. P. Minka, “A comparison of numerical optimizers for logistic regression,” 2003.
- [11] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [12] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [13] H. Yu and S. Kim, “SVM tutorial: classification, regression and ranking,” in *Handbook of Natural computing*, pp. 479–506, Springer, 2012.
- [14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [15] J. Johnson, “Backpropagation for a Linear Layer.” <http://cs231n.stanford.edu/handouts/linear-backprop.pdf>. [Online; accessed 22-Dec-2017].
- [16] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

## BIBLIOGRAPHY

---

- [17] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.
- [18] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.
- [19] A. Krizhevsky, “Learning multiple layers of features from tiny images,” *Technical Report*, 2009.
- [20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, pp. 1097–1105, 2012.
- [21] A. Lavin and S. Gray, “Fast algorithms for convolutional neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.
- [22] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.
- [23] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.
- [24] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, “Imagenet large scale visual recognition challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.
- [25] “iNaturalist challenge at FGVC 2017.” <https://www.kaggle.com/c/inaturalist-challenge-at-fgvc-2017>. Accessed: 2018-04-11.
- [26] E. Learned-Miller, G. B. Huang, A. RoyChowdhury, H. Li, and G. Hua, “Labeled faces in the wild: A survey,” in *Advances in face detection and facial image analysis*, pp. 189–248, Springer, 2016.
- [27] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, “Learning deep features for scene recognition using places database,” in *Advances in neural information processing systems*, pp. 487–495, 2014.
- [28] “iMaterialist challenge at FGVC 2018.” <https://www.kaggle.com/c/imaterialist-challenge-furniture-2018>. Accessed: 2018-04-11.
- [29] L. Fei-Fei, R. Fergus, and P. Perona, “Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories,” *Computer vision and Image understanding*, vol. 106, no. 1, pp. 59–70, 2007.
- [30] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.
- [31] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” *IEEE International Conference on Computer Vision (CVPR)*, pp. 1–9, 2015.

## Chapter 6

# Natural Language Processing with Recurrent Neural Networks in Tensorflow

### Objectives

- Learn Tensorflow Basics
- Learn how to use RNNs, LSTMs, and GRUs in Tensorflow
- Use recurrent modules for sequence modeling tasks

### 6.1 Tensorflow

Tensorflow is a popular library for machine learning developed by Google. Tensorflow includes all of things that might be useful for deep learning including: neural network layers, optimizers, loss functions, and auto differentiation. Different from PyTorch, Tensorflow primarily uses a static computation graph. This means that the computations describing a neural network must be “compiled” into a static graph. PyTorch uses a dynamic computation graph, where operations can be added without the need for recompiling. The advantage of the static graph is better performance in some cases, however the disadvantage is that it is more difficult to debug. To alleviate the difficulty in debugging, Tensorflow added eager mode, which uses a dynamic computation graph. However, this current chapter is based on Tensorflow 1 and eager mode does not support parts of Tensorflow 1, so we will be using the default static computation graph.

#### Hint: Migrating to Tensorflow 2

To learn how to migrate code examples in this chapter to Tensorflow 2, please refer to <https://www.tensorflow.org/guide/migrate> and to the [Tensorflow API Documentation](#). To install the needed Tensorflow 1.x version that would work with code examples in this chapter, please refer to <https://www.tensorflow.org/install/pip>.

Tensorflow can be installed with a `pip` command:

```
sudo pip install tensorflow
```

For code examples in this chapter, you need to install a Tensorflow 1 version. For a CPU-only release, use the following command:

```
sudo pip install tensorflow==1.15
```

For a release with GPU support , use the following command:

```
sudo pip install tensorflow-gpu==1.15
```

Tensorflow has many different high level API extensions such as `tf.keras` and `tf.estimator`. These extensions abstract some of the underlying Tensorflow code away. While these libraries are useful for quickly prototyping code, knowledge of the underlying Tensorflow processes is also important. Please note that Keras has been tightly integrated with Tensorflow 2; Tensorflow 1 `tf.layers` has been replaced with `tf.keras.layers` in Tensorflow 2. For more information, please consult the Tensorflow API documentation at <https://www.tensorflow.org/versions>.

Please follow the following tutorials to become familiar with Tensorflow:

- Low level intro - <https://www.tensorflow.org/overview/>
- Tensors - [https://www.tensorflow.org/programmers\\_guide/tensors](https://www.tensorflow.org/programmers_guide/tensors)
- Variables - [https://www.tensorflow.org/programmers\\_guide/variables](https://www.tensorflow.org/programmers_guide/variables)

## 6.2 CIFAR100 in Tensorflow

We will re-implement the convolutional network from Chapter 4 in Tensorflow for demonstration purposes. We provided the code for this example in `cifar_tensorflow.py`. First we will write a function describing the forward pass of the network. We don't need to write any code for the backwards pass, since the gradient computations are computed automatically from the computation graph. To implement the network, we will utilize the layers provided in the `tf.layers` package. Specifically, we will use `tf.layers.conv2d` to implement a convolutional layer, `tf.layers.max_pooling2d` for max pooling, and `tf.layers.dense` for fully connected layers. The functions `tf.nn.relu` and `tf.reshape` are used for ReLU and flattening, respectively.

```
def cnn_model_fn(x):
    """
    Define 3-layer cnn from Chapter 4
    """
    # define network
    # Convolutional Layer #1
    conv1 = tf.layers.conv2d(
        inputs=x,
        filters=16,
        kernel_size=[3, 3],
        padding="same",
        activation=tf.nn.relu)

    # Pooling Layer #1
    pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2],
        strides=2)

    # Convolutional Layer #2 and Pooling Layer #2
    conv2 = tf.layers.conv2d(
        inputs=pool1,
        filters=32,
        kernel_size=[3, 3],
        padding="same",
        activation=tf.nn.relu)

    pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2],
        strides=2)

    # Logits Layer
    pool2_flat = tf.reshape(pool2, [-1, 8 * 8 * 32])
    dense = tf.layers.dense(inputs=pool2_flat, units=4, activation=tf.
        nn.relu)

    return dense
```

Next we will load the dataset. We have provided a function `cifar100` in the `dataset.py` file that loads the dataset into Numpy arrays. Similar to Chapter 3 and Chapter 4, this function takes a seed and generates a random subset of the dataset with 4 classes.

```
# Load dataset
train_data, test_data = cifar100(1234)
train_x, train_y = train_data
test_x, test_y = train_data
```

In Tensorflow instead of writing functions that directly perform some calculations, we actually are writing functions that define part of a computation graph, which can be run later. To define inputs in the computation graph, we use `tf.placeholder`. As the name suggests, this creates a “placeholder” variable which could correspond to some input values. Later when we call the graph, we can pass real data to these placeholders.

```
# placeholder for input variables
x_placeholder = tf.placeholder(tf.float32,
                               shape=(BATCH_SIZE,) + train_x.shape
                               [1:])
y_placeholder = tf.placeholder(tf.int32, shape=(BATCH_SIZE))
```

Next we define a few operations (ops) that we will use for training and testing. An operation is some output or function that we are interested in computing from the computation graph. In this case, we want an operation to get the output of the network, one to get the loss value, and an operation to perform the gradient descent update. The gradient descent operation uses the `tf.train.GradientDescentOptimizer` class.

```
# get the loss function and the prediction function for the
network
pred_op = cnn_model_fn(x_placeholder)
loss_op = tf.losses.sparse_softmax_cross_entropy(labels=
y_placeholder, logits=pred_op)

# define optimizer
optimizer = tf.train.GradientDescentOptimizer(LR)
train_op = optimizer.minimize(loss_op)
```

Next we can start a Tensorflow *session*. We need to define a session in order to run the previously defined operations. An operation can be run in a session by using `sess.run(op)`, where `sess` is the Tensorflow session, and `op` is the operation we want to run. As an example, we use the session to run an operation that initializes the variables in our network using `tf.global_variables_initializer`.

```
# start tensorflow session
sess = tf.Session()

# initialization
init = tf.global_variables_initializer()
sess.run(init)
```

Finally, we can begin training. There are several ways to perform training in Tensorflow. Here we will manually loop through our data and call the Tensorflow operations for each batch. This approach should be familiar from Chapters 3 and 4. To pass data into the computation graph, we use the `feed_dict` argument of the `sess.run` command. Here we pass Numpy matrices into the place holder values of the graph.

```
# train loop
for epoch in range(NUMEPOCHS):
    running_loss = 0.0
    n_batch = 0
    for i in range(0, train_x.shape[0]-BATCH_SIZE, BATCH_SIZE):
        # get batch data
        x_batch = train_x[i:i+BATCH_SIZE]
        y_batch = train_y[i:i+BATCH_SIZE]

        # run step of gradient descent
        feed_dict = {
            x_placeholder: x_batch,
            y_placeholder: y_batch,
        }
        _, loss_value = sess.run([train_op, loss_op],
                                feed_dict=feed_dict)

        running_loss += loss_value
        n_batch += 1

    print('[Epoch: %d] loss: %.3f' %
          (epoch + 1, running_loss / (n_batch)))
```

We can perform testing in a similar way as training. One difficulty is that the static graph requires a constant batch size. Our dataset might not be able to be evenly split into batches, depending on the batch size. There are a few possible ways to deal with this. One way (shown below) is to pad the last batch such that it matches the batch size. After padding, we should make sure that we don't take any of the padded outputs, since these won't have ground truth labels.

```
# test loop
all_predictions = np.zeros((0, 1))
for i in range(0, test_x.shape[0], BATCH_SIZE):
    x_batch = test_x[i:i+BATCH_SIZE]

    # pad small batch
    padded = BATCH_SIZE - x_batch.shape[0]
    if padded > 0:
        x_batch = np.pad(x_batch,
                        ((0, padded), (0, 0), (0, 0), (0, 0)),
                        'constant')

    # run step
    feed_dict = {x_placeholder: x_batch}
    batch_pred = sess.run(pred_op,
                          feed_dict=feed_dict)

    # recover if padding
    if padded > 0:
        batch_pred = batch_pred[0:-padded]

    # get argmax to get class prediction
    batch_pred = np.argmax(batch_pred, axis=1)

    all_predictions = np.append(all_predictions, batch_pred)
```

## 6.3 RNN Background

Convolutional neural networks work well for certain types of data that can be made into fixed size inputs. But what if the data cannot be assumed to be of fixed size? This is common in problems that consider some sort of time series input like speech or text.

If the input data cannot be assumed to be a fixed size, we need some way to have our network automatically adapt to the different sized inputs. One way to do this is to use *recurrent neurons*, where a neuron's output can be used as an input to the same neuron at a different time step (Figure 6.1). With this formulation we can “unroll” the network for as many time steps as needed to process the data (Figure 6.2). The unrolled representation can be used to perform backpropagation.

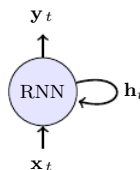


Figure 6.1: Basic RNN Unit.

There are several “flavors” of recurrent units. In this project we will consider three common variants: vanilla recurrent neurons (RNN), long-short term memory units (LSTM) [32], and gated

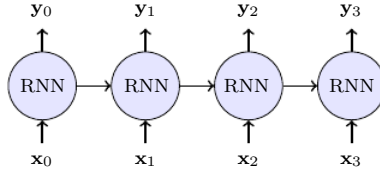


**Figure 6.2:** Unrolled Recurrent Network.

recurrent units (GRU) [33]. We will use these recurrent units to create a network that can classify text.

### 6.3.1 RNN

The equation for the hidden state of a vanilla RNN layer takes the following form:

$$\mathbf{h}_t = \tanh(\mathbf{W}_x \mathbf{x}_t + \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{b}_h) \quad (6.1)$$

where  $\mathbf{x}_t$  is an input at time  $t$ , and  $\mathbf{h}_t$  is the corresponding state vector. The  $\mathbf{W}$  matrices and  $\mathbf{b}$  vectors represent learnable parameters.

RNN layers can be used in Tensorflow with the `tf.contrib.rnn.BasicRNNCell` layer. The main argument in the initialization of this layer is the output feature dimension (`num_units`). You must set this parameter appropriately to achieve good performance.

To use the RNN layer, we wrap it in a `tf.contrib.rnn.static_rnn` function. This function takes the RNN cell and a list of tensors as input. Each tensor of the input list corresponds to a different time step, and the tensor at each time step is of dimensions (batch size, input feature size). The `static_rnn` function yields two outputs: the hidden state tensor at all time steps, and the final value of the hidden state.

The output of the RNN is based on the value of the hidden state. The form of the output is identical to a fully connected layer:

$$\mathbf{y}_t = \text{softmax}(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y) \quad (6.2)$$

where  $\mathbf{y}_t$  is an output at time  $t$ . For this project we are only interested in the output at the last time step. The `BasicRNNCell` layer itself does not compute this output. To implement the output we need to use a separate fully connected layer in Tensorflow (`tf.layers.dense`).

### 6.3.2 LSTM

The long-short term memory unit (LSTM) is a more complex recurrent node that incorporates different “gates” to allow or reject information from passing through the network. There is an input gate, an output gate, and a forget gate that controls the flow of information. Additionally, the LSTM has both a memory cell and a hidden state. The LSTM is computed as:

$$\begin{aligned} \mathbf{f}_t &= \sigma(\mathbf{W}_f x_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f) \\ \mathbf{i}_t &= \sigma(\mathbf{W}_i x_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i) \\ \mathbf{o}_t &= \sigma(\mathbf{W}_o x_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o) \\ \mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tanh(\mathbf{W}_c x_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c) \\ \mathbf{h}_t &= \mathbf{o}_t \circ \tanh(\mathbf{c}_t) \end{aligned}$$

where  $\circ$  is the Hadamard product (e.g., element-wise product),  $\sigma$  is a sigmoid function,  $\mathbf{f}$  is the forget gate,  $\mathbf{i}$  is the input gate,  $\mathbf{o}$  is the output gate,  $\mathbf{c}$  is the memory cell, and  $\mathbf{h}$  is the hidden state.  $\mathbf{W}$ ,  $\mathbf{U}$  and  $\mathbf{b}$  represent learnable parameters.

LSTM layers can be used in Tensorflow with the `rnn.BasicLSTMCell`. As with the RNN, the main arguments in the initialization of this layer is the output feature dimension. The output feature dimension is a parameter that you must set yourself to achieve good performance. To use the LSTM, we again use the `tf.contrib.rnn.static_rnn` function. This function takes the LSTM cell and a list of tensors as input. Each element of the input list corresponds to a different time step, and the tensor at each time step is of dimensions (batch size, input feature size). The `static_rnn` function yields two outputs: the hidden state tensor at all positions, and the final value of the hidden state.

The Tensorflow LSTM implementation has no output  $\mathbf{y}$ . We can create an output using the hidden state at a certain time using equation 6.2. This corresponds to a fully connected layer with the hidden state as the input.

### 6.3.3 GRU

The Gated Recurrent unit GRU can be thought of as a simplified version of LSTM. In practice the performance of the GRU and the LSTM can often be similar. The GRU incorporates only two gates: an update gate and a reset gate. It does not use a memory cell like the LSTM. The GRU is computed by:

$$\begin{aligned} \mathbf{z}_t &= \sigma(\mathbf{W}_z \mathbf{x}_t + \mathbf{U}_z \mathbf{h}_{t-1} + \mathbf{b}_z) \\ \mathbf{r}_t &= \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}_r \mathbf{h}_{t-1} + \mathbf{b}_r) \\ \mathbf{h}_t &= (1 - \mathbf{z}_t) \circ \mathbf{h}_{t-1} + \mathbf{z}_t \circ \tanh(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h (\mathbf{r}_t \circ \mathbf{h}_{t-1}) + \mathbf{b}_h) \end{aligned}$$

where  $\circ$  is the Hadamard product,  $\sigma$  is a sigmoid function,  $\mathbf{z}$  is the update gate,  $\mathbf{r}$  is the reset gate, and  $\mathbf{h}$  is the hidden state.  $\mathbf{W}$ ,  $\mathbf{U}$  and  $\mathbf{b}$  represent learnable parameters.

GRU layers can be used in Tensorflow with the `rnn.GRUCell`. As with the RNN, the main arguments in the initialization of this layer is the output feature dimension. The output feature dimension is a parameter that you must set yourself to achieve good performance. To use the GRU, we again use the `tf.contrib.rnn.static_rnn` function. This function takes the LSTM cell and a list of tensors as input. Each element of the input list corresponds to a different time step, and the tensor at each time step is of dimensions (batch size, input feature size). The `static_rnn` function yields two outputs: the hidden state tensor at all positions, and the final value of the hidden state.

The Tensorflow GRU implementation has no output  $\mathbf{y}$ . We can create an output using the hidden state at a certain time using equation 6.2. This corresponds to a fully connected layer with the hidden state as the input.

## 6.4 RNNs for sentence classification

In this project we will utilize recurrent layers to classify sentences.

### 6.4.1 Datasets

In this project, we consider 4 datasets: sentiment, spam, questions, newsgroups. Each dataset is a collection of text entries, with a corresponding text label for each entry. Each group will be assigned two datasets to work on (see Section 6.4.2). Table 6.1 describes each dataset.

The sentiment dataset consists of movie reviews that are either positive or negative [34]. The spam dataset consists of emails that are either spam or not spam [35].

The questions dataset [36] contains questions classified by what type is the answer to the question. For example the questions “Who is Snoopy’s arch-enemy?” and “What actress has received the most

Oscar nominations?” have the same class. As another example the questions “What novel inspired the movie BladeRunner?” and “What’s the only work by Michelangelo that bears his signature?” also have the same class. The classes are varied enough that the network has to understand the whole sentence to classify, rather than just looking for key words like “what” or “who”

The newsgroups dataset [37] consists of newsgroup postings from twenty different categories (Newsgroups are where people posted things on the internet before Facebook and Reddit). Examples of the newsgroup classes are “comp.sys.mac.hardware”, “talk.politics.guns”, “rec.sport.hockey”.

Each dataset is stored in two CSV files: `train.csv` and `test.csv`. Each line of the CSV file has a sentence and a label. We have provided a `load_text_dataset` function in `dataset.py` to load the datasets as numpy arrays. This function takes as arguments the name of the dataset and the maximum sequence size. Any sentences that are smaller than the maximum sequence size will be padded, and any sequences that are larger will be truncated. The dataset loader will translate each unique word to a unique number using a lookup table. The number of unique words is the vocabulary size of the dataset. This `load_text_dataset` function will also print out how many unique words (the vocabulary size) are there in the loaded `.csv` file. This will be the vocabulary size that you used for embedding.

**Table 6.1:** Datasets for NLP with RNN in Tensorflow

| Dataset    | Description                                | # Categories | # Train / # Test |
|------------|--------------------------------------------|--------------|------------------|
| Sentiment  | Movie review dataset [34]                  | 2            | 7996 / 2666      |
| Spam       | Enron spam dataset [35]                    | 2            | 3879 / 1293      |
| Questions  | Learning question classifiers dataset [36] | 50           | 4464 / 1488      |
| Newsgroups | 20 newsgroups dataset [37]                 | 20           | 14121 / 4707     |

## 6.4.2 Network Design

The input to our network is a batch of sentences and the output is a batch of predicted labels. From the input we first want to perform a word embedding. The embedding layer takes the words, which have been mapped to integers by the dataset loader, and outputs a vector embedding for each word. For example, the word “yes” might be mapped to the vector  $[0.1, 0.7, -0.5]$  and the word “no” might be mapped to the vector  $[1.1, -0.5, 0.3]$ . The dimension of the embedding is a hyper-parameter that must be set. We can use the `tf.nn.embedding_lookup` layer for this. The embedding layer uses an embedding matrix. For our application the embedding matrix can be randomly initialized.

Next we pass the embedded words into the recurrent layer. In general, we could utilize several stacks of recurrent layers, but in this project it is only required to consider a single recurrent layer stack. We want to obtain a class prediction from the output of the recurrent unit, so we can use a fully connected layer after the recurrent layer. The number of outputs of the fully connected layer should be equal to the number of classes.

Since we are doing classification, we can use the `tf.losses.sparse_softmax_cross_entropy` loss function. This loss function incorporates the softmax internally, so we do not need to explicitly add a softmax layer to our model.

Training the sequence models is almost identical to training convolutional neural networks. We still need to loop through our data, and for each batch call the train operation.

**Table 6.2:** Layers for text processing

| Layer                               | Description            | Initialization Arguments            |
|-------------------------------------|------------------------|-------------------------------------|
| <code>tf.nn.embedding_lookup</code> | Embed word vectors     | (embedding matrix, input tensor)    |
| <code>rnn.RNNCell</code>            | Vanilla RNN unit       | (output feature size)               |
| <code>rnn.BasicLSTMCell</code>      | LSTM unit              | (output feature size)               |
| <code>rnn.GRUCell</code>            | GRU unit               | (output feature size)               |
| <code>layers.dense</code>           | Fully connected output | (input tensor, output feature size) |

#### Hint: Training RNNs

Training RNNs may be a little more difficult than training convolutional neural networks. If training is not working well, you may need to fiddle with the learning rate, or try another optimizer such as ADAM [38]. ADAM can be used with the Tensorflow `tf.train.AdamOptimizer` class.

#### Deliverable: NLP with Tensorflow

Depending on your task number, you are assigned two different datasets and one recurrent unit type (Table 6.3). For the assigned datasets and recurrent unit type, you must determine the hyper parameters (e.g., learning rate, batch size, embedding size, hidden state feature dimension) of the network to achieve good performance.

- Submit code for both datasets named as `dataset.unitname.py`, where `dataset` is replaced by your assigned dataset and `unitname` is replaced by your assigned recurrent unit type (lstm, gru)
- Provide the final classification accuracy for both datasets
- Submit the two trained models

## Additional Resources

- Tensorflow API Documentation - [https://www.tensorflow.org/api\\_docs/](https://www.tensorflow.org/api_docs/)
- Tensorflow Tutorials - <https://www.tensorflow.org/tutorials/>
- Tensorflow Github - <https://github.com/tensorflow/tensorflow>
- Tensorflow Discussion Forum - <https://www.tensorflow.org/community/forums>

## Task Assignment

### Submission Instructions

Submit your project as a folder named `FIRSTNAME.LASTNAME.TASKNUMBER.CH6` and zip the folder for submission. The plots and saved models that you generated during this lab should be placed in a folder called `results`. The grading rubric is shown in Table 6.4. There are no unit tests for the submitted code, but your code will be graded by attempting to run it.

**Table 6.3:** Sample tasks for RNN in Tensorflow

| Task number | Datasets              | RNN unit |
|-------------|-----------------------|----------|
| 1           | Sentiment, Spam       | LSTM     |
| 2           | Sentiment, Questions  | LSTM     |
| 3           | Sentiment, Newsgroups | LSTM     |
| 4           | Spam, Questions       | LSTM     |
| 5           | Spam, Newsgroups      | LSTM     |
| 6           | Questions, Newsgroups | LSTM     |
| 7           | Sentiment, Spam       | GRU      |
| 8           | Sentiment, Questions  | GRU      |
| 9           | Sentiment, Newsgroups | GRU      |
| 10          | Spam, Questions       | GRU      |
| 11          | Spam, Newsgroups      | GRU      |
| 12          | Questions, Newsgroups | GRU      |

**Table 6.4:** Grading rubric

| Points    | Description                                            |
|-----------|--------------------------------------------------------|
| NOTE:     | DO NOT SUBMIT THE DATASET                              |
| 40        | Working code for first dataset                         |
| 40        | Working code for second dataset                        |
| 10        | Report final classification accuracy for both datasets |
| 10        | Submit saved models for both datasets                  |
| Total 100 |                                                        |

# Bibliography

- [1] “Numpy reference.” <https://docs.scipy.org/doc/numpy-1.13.0/reference/>, 2017.
- [2] “Scipy reference.” <https://docs.scipy.org/doc/scipy-1.0.0/reference/>, 2017.
- [3] “Matplotlib pyplot reference.” [https://matplotlib.org/api/pyplot\\_api.html](https://matplotlib.org/api/pyplot_api.html), 2017.
- [4] J. Johnson, “Python numpy tutorial.” <http://cs231n.github.io/python-numpy-tutorial/>, 2017.
- [5] “Luma coding in video systems.” [https://en.wikipedia.org/wiki/Grayscale#Luma\\_coding\\_in\\_video\\_systems](https://en.wikipedia.org/wiki/Grayscale#Luma_coding_in_video_systems), 2017.
- [6] Wikipedia, “DFT matrix — Wikipedia, the free encyclopedia.” <http://en.wikipedia.org/w/index.php?title=DFT%20matrix&oldid=811427639>, 2017.
- [7] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- [8] D. R. Cox, “The regression analysis of binary sequences,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 215–242, 1958.
- [9] M. Aly, “Survey on multiclass classification methods,” *Neural Networks*, 2005.
- [10] T. P. Minka, “A comparison of numerical optimizers for logistic regression,” 2003.
- [11] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [12] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [13] H. Yu and S. Kim, “SVM tutorial: classification, regression and ranking,” in *Handbook of Natural computing*, pp. 479–506, Springer, 2012.
- [14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [15] J. Johnson, “Backpropagation for a Linear Layer.” <http://cs231n.stanford.edu/handouts/linear-backprop.pdf>. [Online; accessed 22-Dec-2017].
- [16] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

## BIBLIOGRAPHY

---

- [17] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.
- [18] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.
- [19] A. Krizhevsky, “Learning multiple layers of features from tiny images,” *Technical Report*, 2009.
- [20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, pp. 1097–1105, 2012.
- [21] A. Lavin and S. Gray, “Fast algorithms for convolutional neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.
- [22] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.
- [23] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.
- [24] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, “Imagenet large scale visual recognition challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.
- [25] “iNaturalist challenge at FGVC 2017.” <https://www.kaggle.com/c/inaturalist-challenge-at-fgvc-2017>. Accessed: 2018-04-11.
- [26] E. Learned-Miller, G. B. Huang, A. RoyChowdhury, H. Li, and G. Hua, “Labeled faces in the wild: A survey,” in *Advances in face detection and facial image analysis*, pp. 189–248, Springer, 2016.
- [27] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, “Learning deep features for scene recognition using places database,” in *Advances in neural information processing systems*, pp. 487–495, 2014.
- [28] “iMaterialist challenge at FGVC 2018.” <https://www.kaggle.com/c/imaterialist-challenge-furniture-2018>. Accessed: 2018-04-11.
- [29] L. Fei-Fei, R. Fergus, and P. Perona, “Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories,” *Computer vision and Image understanding*, vol. 106, no. 1, pp. 59–70, 2007.
- [30] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.
- [31] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” *IEEE International Conference on Computer Vision (CVPR)*, pp. 1–9, 2015.
- [32] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

## BIBLIOGRAPHY

---

- [33] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.
- [34] B. Pang, L. Lee, and S. Vaithyanathan, “Thumbs up?: sentiment classification using machine learning techniques,” in *Proceedings of the ACL-02 conference on Empirical methods in natural language processing- Volume 10*, pp. 79–86, Association for Computational Linguistics, 2002.
- [35] V. Metsis, I. Androutsopoulos, and G. Paliouras, “Spam filtering with naive bayes-which naive bayes?,” in *Proceedings of the 3rd Conference on Email and Anti-Spam (CEAS 2006)*, 2006.
- [36] X. Li and D. Roth, “Learning question classifiers,” in *Proceedings of the 19th international conference on Computational linguistics- Volume 1*, pp. 1–7, Association for Computational Linguistics, 2002.
- [37] K. Lang, “Newsweeder: Learning to filter netnews,” in *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 331–339, 1995.
- [38] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

# Bibliography

- [1] “Numpy reference.” <https://docs.scipy.org/doc/numpy-1.13.0/reference/>, 2017.
- [2] “Scipy reference.” <https://docs.scipy.org/doc/scipy-1.0.0/reference/>, 2017.
- [3] “Matplotlib pyplot reference.” [https://matplotlib.org/api/pyplot\\_api.html](https://matplotlib.org/api/pyplot_api.html), 2017.
- [4] J. Johnson, “Python numpy tutorial.” <http://cs231n.github.io/python-numpy-tutorial/>, 2017.
- [5] “Luma coding in video systems.” [https://en.wikipedia.org/wiki/Grayscale#Luma\\_coding\\_in\\_video\\_systems](https://en.wikipedia.org/wiki/Grayscale#Luma_coding_in_video_systems), 2017.
- [6] Wikipedia, “DFT matrix — Wikipedia, the free encyclopedia.” <http://en.wikipedia.org/w/index.php?title=DFT%20matrix&oldid=811427639>, 2017.
- [7] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- [8] D. R. Cox, “The regression analysis of binary sequences,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 215–242, 1958.
- [9] M. Aly, “Survey on multiclass classification methods,” *Neural Networks*, 2005.
- [10] T. P. Minka, “A comparison of numerical optimizers for logistic regression,” 2003.
- [11] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [12] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [13] H. Yu and S. Kim, “SVM tutorial: classification, regression and ranking,” in *Handbook of Natural computing*, pp. 479–506, Springer, 2012.
- [14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [15] J. Johnson, “Backpropagation for a Linear Layer.” <http://cs231n.stanford.edu/handouts/linear-backprop.pdf>. [Online; accessed 22-Dec-2017].
- [16] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

## BIBLIOGRAPHY

---

- [17] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.
- [18] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.
- [19] A. Krizhevsky, “Learning multiple layers of features from tiny images,” *Technical Report*, 2009.
- [20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, pp. 1097–1105, 2012.
- [21] A. Lavin and S. Gray, “Fast algorithms for convolutional neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.
- [22] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.
- [23] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.
- [24] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, “Imagenet large scale visual recognition challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.
- [25] “iNaturalist challenge at FGVC 2017.” <https://www.kaggle.com/c/inaturalist-challenge-at-fgvc-2017>. Accessed: 2018-04-11.
- [26] E. Learned-Miller, G. B. Huang, A. RoyChowdhury, H. Li, and G. Hua, “Labeled faces in the wild: A survey,” in *Advances in face detection and facial image analysis*, pp. 189–248, Springer, 2016.
- [27] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, “Learning deep features for scene recognition using places database,” in *Advances in neural information processing systems*, pp. 487–495, 2014.
- [28] “iMaterialist challenge at FGVC 2018.” <https://www.kaggle.com/c/imaterialist-challenge-furniture-2018>. Accessed: 2018-04-11.
- [29] L. Fei-Fei, R. Fergus, and P. Perona, “Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories,” *Computer vision and Image understanding*, vol. 106, no. 1, pp. 59–70, 2007.
- [30] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.
- [31] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” *IEEE International Conference on Computer Vision (CVPR)*, pp. 1–9, 2015.
- [32] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

## BIBLIOGRAPHY

---

- [33] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.
- [34] B. Pang, L. Lee, and S. Vaithyanathan, “Thumbs up?: sentiment classification using machine learning techniques,” in *Proceedings of the ACL-02 conference on Empirical methods in natural language processing- Volume 10*, pp. 79–86, Association for Computational Linguistics, 2002.
- [35] V. Metsis, I. Androutsopoulos, and G. Paliouras, “Spam filtering with naive bayes-which naive bayes?,” in *Proceedings of the 3rd Conference on Email and Anti-Spam (CEAS 2006)*, 2006.
- [36] X. Li and D. Roth, “Learning question classifiers,” in *Proceedings of the 19th international conference on Computational linguistics-Volume 1*, pp. 1–7, Association for Computational Linguistics, 2002.
- [37] K. Lang, “Newsweeder: Learning to filter netnews,” in *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 331–339, 1995.
- [38] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.