# PYTHON 100%

HEAD 99.8%

HISS 96.4%

TAIL 97.5%

# MACHINE LEARNING PROJECTS

COMPILED BY BRIAN BOUCHERON & LISA TAGLIAFERRI

# Python Machine Learning Projects

Written by Lisa Tagliaferri, Michelle Morales, Ellie Birbeck, and
Alvin Wan, with editing by Brian Hogan and Mark Drake

# Python Machine Learning Projects

# Foreword

As machine learning is increasingly leveraged to find patterns, conduct analysis, and make decisions without final input from humans, it is of equal importance to not only provide resources to advance algorithms and methodologies, but to also invest in bringing more stakeholders into the fold. This book of Python projects in machine learning tries to do just that: to equip the developers of today and tomorrow with tools they can use to better understand, evaluate, and shape machine learning to help ensure that it is serving us all.

This book will set you up with a Python programming environment if you don't have one already, then provide you with a conceptual understanding of machine learning in the chapter "An Introduction to Machine Learning." What follows next are three Python machine learning projects. They will help you create a machine learning classifier, build a neural network to recognize handwritten digits, and give you a background in deep reinforcement learning through building a bot for Atari.

These chapters originally appeared as articles on DigitalOcean Community, written by members of the international software developer community. If you are interested in contributing to this knowledge base, consider proposing a tutorial to the Write for DOnations program at [do.co/w4do](do.co/w4do). DigitalOcean offers payment to authors and provides a matching donation to tech-focused nonprofits.

## Other Books in this Series

If you are learning Python or are looking for reference material, you can download our free Python eBook, How To Code in Python 3 which is available via [do.co/python-book](do.co/python-book).

For other programming languages and DevOps engineering articles, our knowledge base of over 2,100 tutorials is available as a Creative-Commons-licensed resource via [do.co/tutorials](do.co/tutorials).

# Setting Up a Python Programming Environment
Written by Lisa Tagliaferri

Python is a flexible and versatile programming language suitable for many use cases, with strengths in scripting, automation, data analysis, machine learning, and back-end development. First published in 1991 the Python development team was inspired by the British comedy group Monty Python to make a programming language that was fun to use. Python 3 is the most current version of the language and is considered to be the future of Python.

This tutorial will help get your remote server or local computer set up with a Python 3 programming environment. If you already have Python 3 installed, along with pip and venv, feel free to move onto the next chapter!

## Prerequisites

This tutorial will be based on working with a Linux or Unix-like (*nix) system and use of a command line or terminal environment. Both macOS and specifically the PowerShell program of Windows should be able to achieve similar results.

## Step 1 — Installing Python 3

Many operating systems come with Python 3 already installed. You can check to see whether you have Python 3 installed by opening up a terminal window and typing the following:

```
python3 -V
```

You'll receive output in the terminal window that will let you know the version number. While this number may vary, the output will be similar to this:

Output

```
Python 3.7.2
```

If you received alternate output, you can navigate in a web browser to [python.org](python.org) in order to download Python 3 and install it to your machine by following the instructions.

Once you are able to type the `python3 -V` command above and receive output that states your computer's Python version number, you are ready to continue.

## Step 2 — Installing pip

To manage software packages for Python, let's install pip, a tool that will install and manage programming packages we may want to use in our development projects.

If you have downloaded Python from python.org, you should have pip already installed. If you are on an Ubuntu or Debian server or computer, you can download pip by typing the following:

```
sudo apt install -y python3-pip
```

Now that you have pip installed, you can download Python packages with the following command:

```
pip3 install package_name
```

Here, `package_name` can refer to any Python package or library, such as Django for web development or NumPy for scientific computing. So if you would like to install NumPy, you can do so with the command `pip3 install numpy`.

There are a few more packages and development tools to install to ensure that we have a robust set-up for our programming environment:

```
sudo apt install build-essential libssl-dev libffi-dev python3-dev
```

Once Python is set up, and pip and other tools are installed, we can set up a virtual environment for our development projects.

## Step 3 — Setting Up a Virtual Environment

Virtual environments enable you to have an isolated space on your server for Python projects, ensuring that each of your projects can have its own set of dependencies that won't disrupt any of your other projects.

Setting up a programming environment provides us with greater control over our Python projects and over how different versions of packages are handled. This is especially important when working with third-party packages.

You can set up as many Python programming environments as you want. Each environment is basically a directory or folder on your server that has a few scripts in it to make it act as an environment.

While there are a few ways to achieve a programming environment in Python, we'll be using the venv module here, which is part of the standard Python 3 library.

If you have installed Python with through the installer available from python.org, you should have venv ready to go.

To install venv into an Ubuntu or Debian server or machine, you can install it with the following:

```
sudo apt install -y python3-venv
```

With venv installed, we can now create environments. Let's either choose which directory we would like to put our Python programming environments in, or create a new directory with `mkdir`, as in:

```
mkdir environments
cd environments
```

Once you are in the directory where you would like the environments to live, you can create an environment. You should use the version of Python that is installed on your machine as the first part of the command (the output you received when typing `python -V`). If that version was `Python 3.6.3`, you can type the following:

```
python3.6 -m venv my_env
```

If, instead, your computer has `Python 3.7.3` installed, use the following command:

```
python3.7 -m venv my_env
```

Windows machines may allow you to remove the version number entirely:

```
python -m venv my_env
```

Once you run the appropriate command, you can verify that the environment is set up be continuing.

Essentially, `pyvenv` sets up a new directory that contains a few items which we can view with the `ls` command:

```
ls my_env
```

Output
```
bin include lib lib64 pyvenv.cfg share
```

Together, these files work to make sure that your projects are isolated from the broader context of your local machine, so that system files and project files don't mix. This is good practice for version control and to ensure that each of your projects has access to the particular packages that it needs. Python Wheels, a built-package format for Python that can speed up your software production by reducing the number of times you need to compile, will be in the Ubuntu 18.04 `share` directory.

To use this environment, you need to activate it, which you can achieve by typing the following command that calls the activate script:

```
source my_env/bin/activate
```

Your command prompt will now be prefixed with the name of your environment, in this case it is called my_env. Depending on what version o f Debian Linux you are running, your prefix may appear somewhat

differently, but the name of your environment in parentheses should be the first thing you see on your line:

```
(my_env) sammy@sammy:~/environments$
```

This prefix lets us know that the environment my_env is currently active, meaning that when we create programs here they will use only this particular environment's settings and packages.

Note: Within the virtual environment, you can use the command `python` instead of `python3`, and `pip` instead of `pip3` if you would prefer. If you use Python 3 on your machine outside of an environment, you will need to use the `python3` and `pip3` commands exclusively.

After following these steps, your virtual environment is ready to use.

## Step 4 — Creating a "Hello, World" Program

Now that we have our virtual environment set up, let's create a traditional "Hello, World!" program. This will let us test our environment and provides us with the opportunity to become more familiar with Python if we aren't already.

To do this, we'll open up a command-line text editor such as nano and create a new file:

```
(my_env) sammy@sammy:~/environments$ nano hello.py
```

Once the text file opens up in the terminal window we'll type out our program:

```
print("Hello, World!")
```

Exit nano by typing the CTRL and X keys, and when prompted to save the file press y.

Once you exit out of nano and return to your shell, let's run the program:

```
(my_env) sammy@sammy:~/environments$ python hello.py
```

The hello.py program that you just created should cause your terminal to produce the following output:

```
Output
Hello, World!
```

To leave the environment, simply type the command deactivate and you will return to your original directory.

## Conclusion

At this point you have a Python 3 programming environment set up on your machine and you can now begin a coding project!

If you would like to learn more about Python, you can download our free How To Code in Python 3 eBook via [do.co/python-book](do.co/python-book).

# An Introduction to Machine Learning
Written by Lisa Tagliaferri

Machine learning is a subfield of artificial intelligence (AI). The goal of machine learning generally is to understand the structure of data and fit that data into models that can be understood and utilized by people.

Although machine learning is a field within computer science, it differs from traditional computational approaches. In traditional computing, algorithms are sets of explicitly programmed instructions used by computers to calculate or problem solve. Machine learning algorithms instead allow for computers to train on data inputs and use statistical analysis in order to output values that fall within a specific range. Because of this, machine learning facilitates computers in building models from sample data in order to automate decision-making processes based on data inputs.

Any technology user today has benefitted from machine learning. Facial recognition technology allows social media platforms to help users tag and share photos of friends. Optical character recognition (OCR) technology converts images of text into movable type. Recommendation engines, powered by machine learning, suggest what movies or television shows to watch next based on user preferences. Self-driving cars that rely on machine learning to navigate may soon be available to consumers.

Machine learning is a continuously developing field. Because of this, there are some considerations to keep in mind as you work with machine learning methodologies, or analyze the impact of machine learning processes.

In this tutorial, we'll look into the common machine learning methods of supervised and unsupervised learning, and common algorithmic approaches in machine learning, including the k-nearest neighbor algorithm, decision tree learning, and deep learning. We'll explore which programming languages are most used in machine learning, providing you with some of the positive and negative attributes of each. Additionally, we'll discuss biases that are perpetuated by machine learning algorithms, and consider what can be kept in mind to prevent these biases when building algorithms.

## Machine Learning Methods

In machine learning, tasks are generally classified into broad categories. These categories are based on how learning is received or how feedback on the learning is given to the system developed.

Two of the most widely adopted machine learning methods are supervised learning which trains algorithms based on example input and output data that is labeled by humans, and unsupervised learning which provides the algorithm with no labeled data in order to allow it to find structure within its input data. Let's explore these methods in more detail.

### Supervised Learning

In supervised learning, the computer is provided with example inputs that are labeled with their desired outputs. The purpose of this method is for the algorithm to be able to "learn" by comparing its actual output with the "taught" outputs to find errors, and modify the model accordingly. Supervised learning therefore uses patterns to predict label values on additional unlabeled data.

For example, with supervised learning, an algorithm may be fed data with images of sharks labeled as `fish` and images of oceans labeled as `water`. By being trained on this data, the supervised learning algorithm should be able to later identify unlabeled shark images as `fish` and unlabeled ocean images as `water`.

A common use case of supervised learning is to use historical data to predict statistically likely future events. It may use historical stock market information to anticipate upcoming fluctuations, or be employed to filter out spam emails. In supervised learning, tagged photos of dogs can be used as input data to classify untagged photos of dogs.

## Unsupervised Learning

In unsupervised learning, data is unlabeled, so the learning algorithm is left to find commonalities among its input data. As unlabeled data are more abundant than labeled data, machine learning methods that facilitate unsupervised learning are particularly valuable.

The goal of unsupervised learning may be as straightforward as discovering hidden patterns within a dataset, but it may also have a goal of feature learning, which allows the computational machine to automatically discover the representations that are needed to classify raw data.

Unsupervised learning is commonly used for transactional data. You may have a large dataset of customers and their purchases, but as a human you will likely not be able to make sense of what similar attributes can be drawn from customer profiles and their types of purchases. With this data fed into an unsupervised learning algorithm, it may be determined that women of a certain age range who buy unscented soaps are likely to be pregnant, and therefore a marketing

campaign related to pregnancy and baby products can be targeted to this audience in order to increase their number of purchases.

Without being told a "correct" answer, unsupervised learning methods can look at complex data that is more expansive and seemingly unrelated in order to organize it in potentially meaningful ways. Unsupervised learning is often used for anomaly detection including for fraudulent credit card purchases, and recommender systems that recommend what products to buy next. In unsupervised learning, untagged photos of dogs can be used as input data for the algorithm to find likenesses and classify dog photos together.

## Approaches

As a field, machine learning is closely related to computational statistics, so having a background knowledge in statistics is useful for understanding and leveraging machine learning algorithms.

For those who may not have studied statistics, it can be helpful to first define correlation and regression, as they are commonly used techniques for investigating the relationship among quantitative variables. Correlation is a measure of association between two variables that are not designated as either dependent or independent. Regression at a basic level is used to examine the relationship between one dependent and one independent variable. Because regression statistics can be used to anticipate  the dependent variable when the independent variable is known, regression enables prediction capabilities.

Approaches to machine learning are continuously being developed. For our purposes, we'll go through a few of the popular approaches that are being used in machine learning at the time of writing.

## k-nearest neighbor

The k-nearest neighbor algorithm is a pattern recognition model that can be used for classification as well as regression. Often abbreviated as k-NN, the k in k-nearest neighbor is a positive integer, which is typically small. In either classification or regression, the input will consist of the k closest training examples within a space.

We will focus on k-NN classification. In this method, the output is class membership. This will assign a new object to the class most common among its k nearest neighbors. In the case of k = 1, the object is assigned to the class of the single nearest neighbor.

Let's look at an example of k-nearest neighbor. In the diagram below, there are blue diamond objects and orange star objects. These belong to two separate classes: the diamond class and the star class.
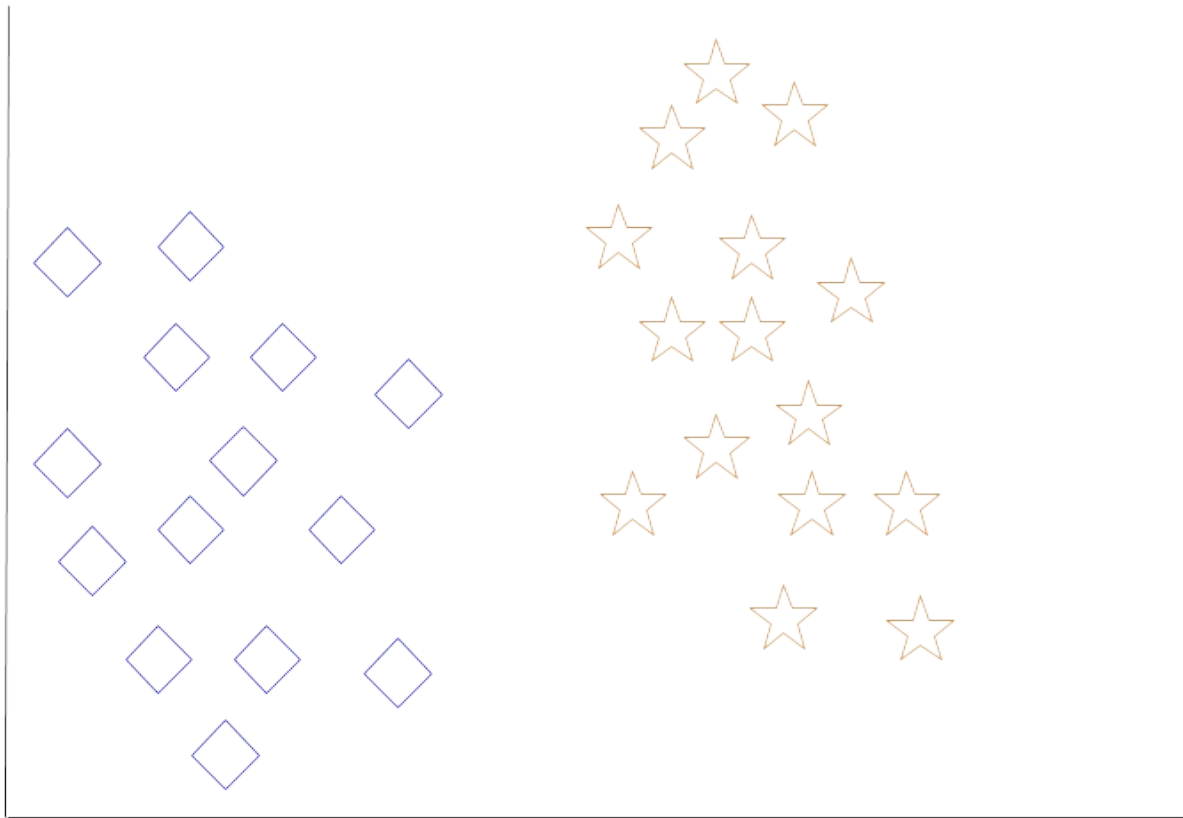
k-nearest neighbor initial data set

When a new object is added to the space — in this case a green heart — we will want the machine learning algorithm to classify the heart to a certain class.
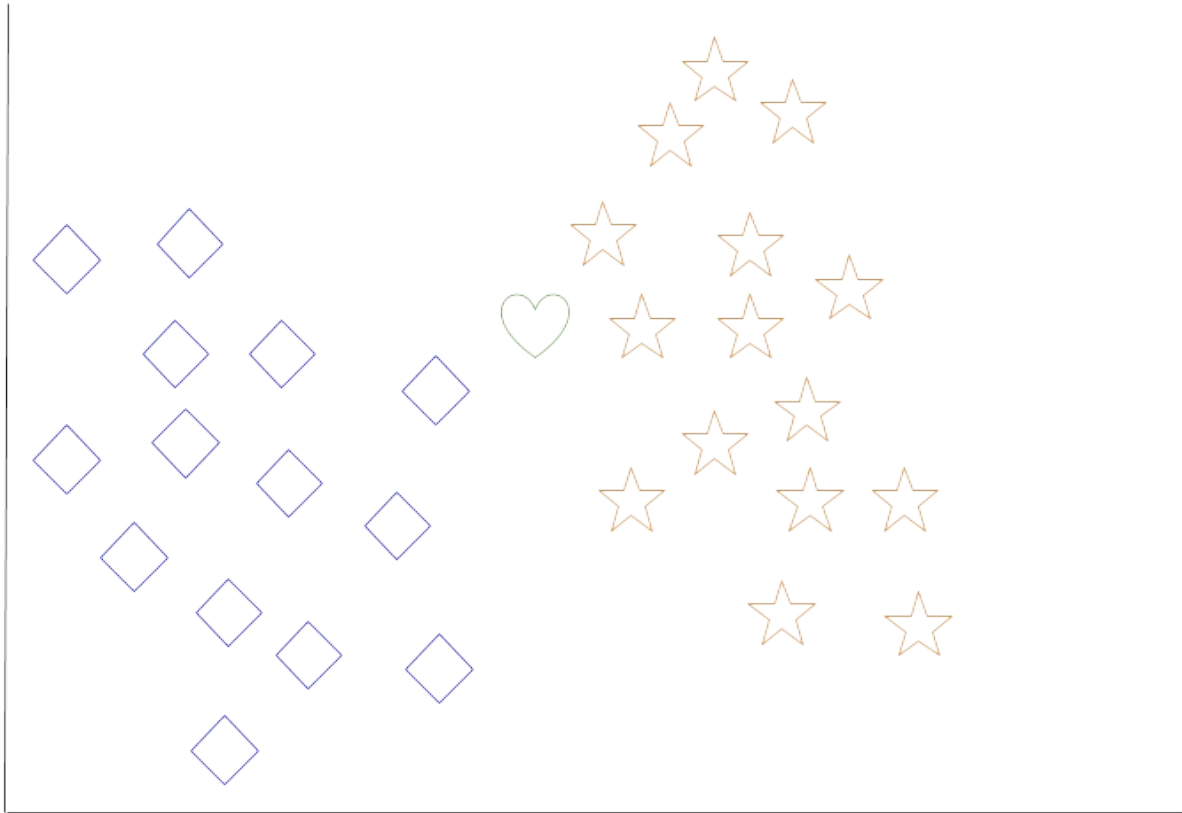
k-nearest neighbor data set with new object to classify

When we choose k = 3, the algorithm will find the three nearest neighbors of the green heart in order to classify it to either the diamond class or the star class.

In our diagram, the three nearest neighbors of the green heart are one diamond and two stars. Therefore, the algorithm will classify the heart with the star class.
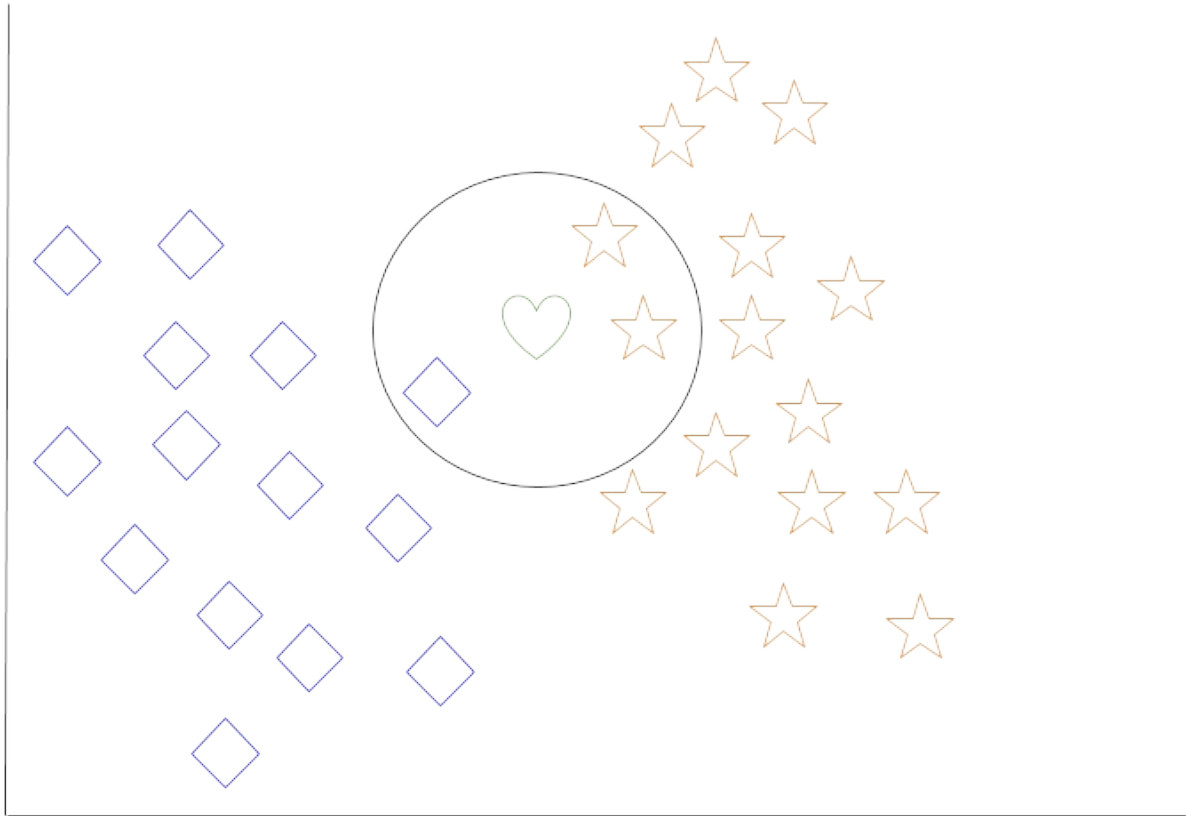
k-nearest neighbor data set with classification complete

Among the most basic of machine learning algorithms, k-nearest neighbor is considered to be a type of "lazy learning" as generalization beyond the training data does not occur until a query is made to the system.

## Decision Tree Learning

For general use, decision trees are employed to visually represent decisions and show or inform decision making. When working with machine learning and data mining, decision trees are used as a predictive model. These models map observations about data to conclusions about the data's target value.

The goal of decision tree learning is to create a model that will predict

the value of a target based on input variables.

In the predictive model, the data's attributes that are determined through observation are represented by the branches, while the conclusions about the data's target value are represented in the leaves.

When "learning" a tree, the source data is divided into subsets based on an attribute value test, which is repeated on each of the derived subsets recursively. Once the subset at a node has the equivalent value as its target value has, the recursion process will be complete.

Let's look at an example of various conditions that can determine whether or not someone should go fishing. This includes weather conditions as well as barometric pressure conditions.
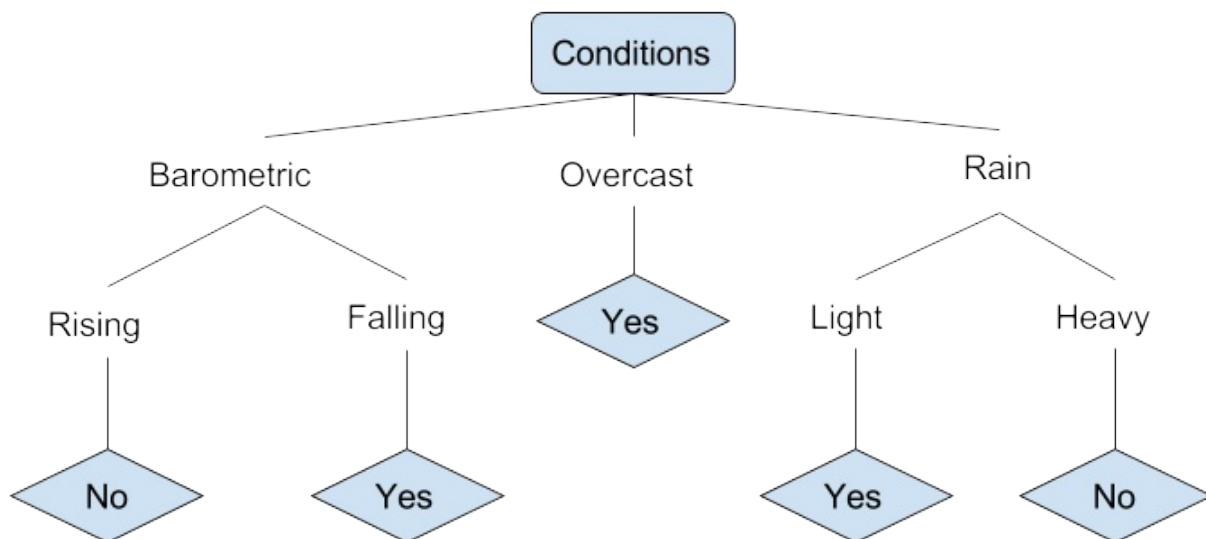
fishing decision tree example

In the simplified decision tree above, an example is classified by sorting it through the tree to the appropriate leaf node. This then returns the classification associated with the particular leaf, which in this case is

either a `Yes` or a `No`. The tree classifies a day's conditions based on whether or not it is suitable for going fishing.

A true classification tree data set would have a lot more features than what is outlined above, but relationships should be straightforward to determine. When working with decision tree learning, several determinations need to be made, including what features to choose, what conditions to use for splitting, and understanding when the decision tree has reached a clear ending.

## Deep Learning

Deep learning attempts to imitate how the human brain can process light and sound stimuli into vision and hearing. A deep learning architecture is inspired by biological neural networks and consists of multiple layers in an artificial neural network made up of hardware and GPUs.

Deep learning uses a cascade of nonlinear processing unit layers in order to extract or transform features (or representations) of the data. The output of one layer serves as the input of the successive layer. In deep learning, algorithms can be either supervised and serve to classify data, or unsupervised and perform pattern analysis.

Among the machine learning algorithms that are currently being used and developed, deep learning absorbs the most data and has been able to beat humans in some cognitive tasks. Because of these attributes, deep learning has become the approach with significant potential in the artificial intelligence space

Computer vision and speech recognition have both realized significant advances from deep learning approaches. IBM Watson is a well-known example of a system that leverages deep learning.

## Human Biases

Although data and computational analysis may make us think that we are receiving objective information, this is not the case; being based on data does not mean that machine learning outputs are neutral. Human bias plays a role in how data is collected, organized, and ultimately in the algorithms that determine how machine learning will interact with that data.

If, for example, people are providing images for "fish" as data to train an algorithm, and these people overwhelmingly select images of goldfish, a computer may not classify a shark as a fish. This would create a bias against sharks as fish, and sharks would not be counted as fish.

When using historical photographs of scientists as training data, a computer may not properly classify scientists who are also people of color or women. In fact, recent peer-reviewed research has indicated that AI and machine learning programs exhibit human-like biases that include race and gender prejudices. See, for example "[Semantics derived automatically from language corpora contain human-like biases](#)" and "[Men Also Like Shopping: Reducing Gender Bias Amplification using Corpus-level Constraints](#)" [PDF].

As machine learning is increasingly leveraged in business, uncaught biases can perpetuate systemic issues that may prevent people from qualifying for loans, from being shown ads for high-paying job opportunities, or from receiving same-day delivery options.

Because human bias can negatively impact others, it is extremely important to be aware of it, and to also work towards eliminating it as much as possible. One way to work towards achieving this is by ensuring that there are diverse people working on a project and that diverse

people are testing and reviewing it. Others have called for [regulatory third parties to monitor and audit algorithms](#), [building alternative systems that can detect biases](#), and [ethics reviews](#) as part of data science project planning. Raising awareness about biases, being mindful of our own unconscious biases, and structuring equity in our machine learning projects and pipelines can work to combat bias in this field.

## Conclusion

This tutorial reviewed some of the use cases of machine learning, common methods and popular approaches used in the field, suitable machine learning programming languages, and also covered some things to keep in mind in terms of unconscious biases being replicated in algorithms.

Because machine learning is a field that is continuously being innovated, it is important to keep in mind that algorithms, methods, and approaches will continue to change.

Currently, Python is one of the most popular programming languages to use with machine learning applications in professional fields. Other languages you may wish to investigate include Java, R, and C++.

# How To Build a Machine Learning Classifier in Python with Scikit-learn

Written by Michelle Morales

Edited by Brian Hogan

In this tutorial, you'll implement a simple machine learning algorithm in Python using [Scikit-learn](#), a machine learning tool for Python. Using a database of breast cancer tumor information, you'll use a [Naive Bayes (NB)](#) classifier that predicts whether or not a tumor is malignant or benign.

By the end of this tutorial, you'll know how to build your very own machine learning model in Python.

## Prerequisites

To complete this tutorial, we'll use Jupyter Notebooks, which are a useful and interactive way to run machine learning experiments. With Jupyter Notebooks, you can run short blocks of code and see the results quickly, making it easy to test and debug your code.

To get up and running quickly, you can open up a web browser and navigate to the Try Jupyter website: [jupyter.org/try](#). From there, click on [Try Jupyter with Python](#), and you will be taken to an interactive Jupyter Notebook where you can start to write Python code.

If you would like to learn more about Jupyter Notebooks and how to set up your own Python programming environment to use with Jupyter, you can read our tutorial on [How To Set Up Jupyter Notebook for Python 3](#).

## Step 1 — Importing Scikit-learn

Let's begin by installing the Python module Scikit-learn, one of the best and most documented machine learning libraries for Python.

To begin our coding project, let's activate our Python 3 programming environment. Make sure you're in the directory where your environment is located, and run the following command:

```
. my_env/bin/activate
```

With our programming environment activated, check to see if the Sckikit-learn module is already installed:

```
(my_env) $ python -c "import sklearn"
```

If `sklearn` is installed, this command will complete with no error. If it is not installed, you will see the following error message:

Output
```
Traceback (most recent call last): File "<string>", line 1, in <module>
ImportError: No module named 'sklearn'
```

The error message indicates that `sklearn` is not installed, so download the library using `pip`:

```
(my_env) $ pip install scikit-learn[alldeps]
```

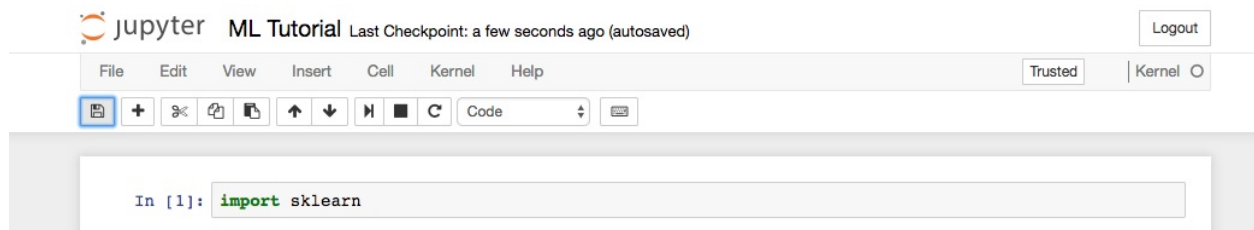Once the installation completes, launch Jupyter Notebook:

```
(my_env) $ jupyter notebook
```

In Jupyter, create a new Python Notebook called ML Tutorial. In the first cell of the Notebook, [import](#) the `sklearn` module:

ML Tutorial

```
import sklearn
```

Your notebook should look like the following figure:



Jupyter Notebook with one Python cell, which imports sklearn

Now that we have `sklearn` imported in our notebook, we can begin working with the dataset for our machine learning model.

## Step 2 — Importing Scikit-learn's Dataset

The dataset we will be working with in this tutorial is the [Breast Cancer Wisconsin Diagnostic Database](#). The dataset includes various information about breast cancer tumors, as well as classification labels of malignant or benign. The dataset has 569 instances, or data, on 569 tumors and includes information on 30 attributes, or features, such as the radius of the tumor, texture, smoothness, and area.

Using this dataset, we will build a machine learning model to use tumor information to predict whether or not a tumor is malignant or

benign.

Scikit-learn comes installed with various datasets which we can load into Python, and the dataset we want is included. Import and load the dataset:

```
...
from sklearn.datasets import load_breast_cancer

# Load dataset

data = load_breast_cancer()
```

The data [variable](#) represents a Python object that works like a [dictionary](#). The important dictionary keys to consider are the classification label names (`target_names`), the actual labels (`target`), the attribute/feature names (`feature_names`), and the attributes (`data`).

Attributes are a critical part of any classifier. Attributes capture important characteristics about the nature of the data. Given the label we are trying to predict (malignant versus benign tumor), possible useful attributes include the size, radius, and texture of the tumor.

Create new variables for each important set of information and assign the data:

```
...
# Organize our data
```

```
label_names = data['target_names']

labels = data['target']

feature_names = data['feature_names']

features = data['data']
```

We now have [lists](#) for each set of information. To get a better understanding of our dataset, let's take a look at our data by printing our class  labels, the first data instance's label, our feature names, and the feature values for the first data instance:

ML Tutorial

```
...

# Look at our data

print(label_names)

print(labels[0])

print(feature_names[0])

print(features[0])
```

You'll see the following results if you run the code:

```
In [3]:  # Look at our data
         print(label_names)
         print(labels[0])
         print(feature_names[0])
         print(features[0])

         ['malignant' 'benign']
         0
         mean radius
         [  1.79900000e+01   1.03800000e+01   1.22800000e+02   1.00100000e+03
            1.18400000e-01   2.77600000e-01   3.00100000e-01   1.47100000e-01
            2.41900000e-01   7.87100000e-02   1.09500000e+00   9.05300000e-01
            8.58900000e+00   1.53400000e+02   6.39900000e-03   4.90400000e-02
            5.37300000e-02   1.58700000e-02   3.00300000e-02   6.19300000e-03
            2.53800000e+01   1.73300000e+01   1.84600000e+02   2.01900000e+03
            1.62200000e-01   6.65600000e-01   7.11900000e-01   2.65400000e-01
            4.60100000e-01   1.18900000e-01]
```

Alt Jupyter Notebook with three Python cells, which prints the first instance in our dataset

As the image shows, our class names are malignant and benign, which are then mapped to binary values of `0` and `1`, where `0` represents malignant tumors and `1` represents benign tumors. Therefore, our first data instance is a malignant tumor whose mean radius is `1.79900000e+01`.

Now that we have our data loaded, we can work with our data to build our machine learning classifier.

## Step 3 — Organizing Data into Sets

To evaluate how well a classifier is performing, you should always test the model on unseen data. Therefore, before building a model, split your data into two parts: a training set and a test set.

You use the training set to train and evaluate the model during the development stage. You then use the trained model to make predictions on the unseen test set. This approach gives you a sense of the model's performance and robustness.

Fortunately, `sklearn` has a function called `train_test_split()`, which divides your data into these sets. Import the function and then use it to split the data:

ML Tutorial

```
...
from sklearn.model_selection import train_test_split


# Split our data


train, test, train_labels, test_labels = train_test_split(features,
labels,
```

```
test_size=0.33,

random_state=42)
```

The function randomly splits the data using the `test_size` parameter. In this example, we now have a test set (`test`) that represents 33% of the original dataset. The remaining data (`train`) then makes up the training data. We also have the respective labels for both the train/test variables, i.e. `train_labels` and `test_labels`.

We can now move on to training our first model.

## Step 4 — Building and Evaluating the Model

There are many models for machine learning, and each model has its own strengths and weaknesses. In this tutorial, we will focus on a simple algorithm that usually performs well in binary classification tasks, namely [Naive Bayes (NB)](#).

First, import the `GaussianNB` module. Then initialize the model with the `GaussianNB()` function, then train the model by fitting it to the data using `gnb.fit()`:

```
ML Tutorial

...

from sklearn.naive_bayes import GaussianNB



# Initialize our classifier

gnb = GaussianNB()



# Train our classifier
```

```
model = gnb.fit(train, train_labels)
```

After we train the model, we can then use the trained model to make predictions on our test set, which we do using the `predict()` function. The `predict()` function returns an array of predictions for each data instance in the test set. We can then print our predictions to get a sense of what the model determined.

Use the `predict()` function with the `test` set and print the results:

ML Tutorial

...

```
# Make predictions
preds = gnb.predict(test)
print(preds)
```

Run the code and you'll see the following results:



Jupyter Notebook with Python cell that prints the predicted values of the Naive Bayes classifier on our test data

As you see in the Jupyter Notebook output, the `predict()` function returned an array of 0s and 1s which represent our predicted values for

the tumor class (malignant vs. benign).

Now that we have our predictions, let's evaluate how well our classifier is performing.
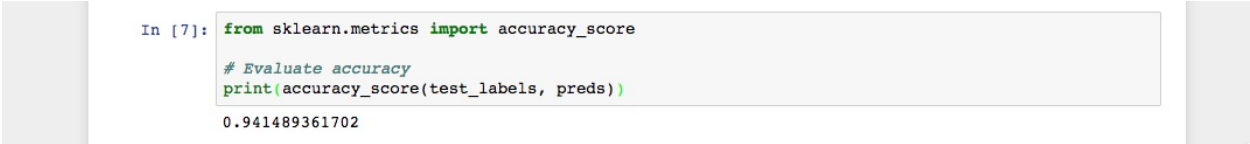
## Step 5 — Evaluating the Model's Accuracy

Using the array of true class labels, we can evaluate the accuracy of our model's predicted values by comparing the two arrays (`test_labels` vs. `preds`). We will use the `sklearn` function `accuracy_score()` to determine the accuracy of our machine learning classifier.

ML Tutorial

```
...

from sklearn.metrics import accuracy_score


# Evaluate accuracy


print(accuracy_score(test_labels, preds))
```

You'll see the following results:

```
In [7]: from sklearn.metrics import accuracy_score

        # Evaluate accuracy
        print(accuracy_score(test_labels, preds))

        0.941489361702
```

Alt Jupyter Notebook with Python cell that prints the accuracy of our NB classifier

As you see in the output, the NB classifier is 94.15% accurate. This means that 94.15 percent of the time the classifier is able to make the correct prediction as to whether or not the tumor is malignant or benign.

These results suggest that our feature set of 30 attributes are good indicators of tumor class.

You have successfully built your first machine learning classifier. Let's reorganize the code by placing all `import` statements at the top of the Notebook or script. The final version of the code should look like this:

ML Tutorial

```
from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split

from sklearn.naive_bayes import GaussianNB

from sklearn.metrics import accuracy_score



# Load dataset
data = load_breast_cancer()


# Organize our data
label_names = data['target_names']

labels = data['target']

feature_names = data['feature_names']

features = data['data']


# Look at our data
print(label_names)

print('Class label = ', labels[0])

print(feature_names)

print(features[0])
```

```
# Split our data
train, test, train_labels, test_labels = train_test_split(features,

                                                          labels,

test_size=0.33,

random_state=42)


# Initialize our classifier
gnb = GaussianNB()


# Train our classifier
model = gnb.fit(train, train_labels)


# Make predictions
preds = gnb.predict(test)
print(preds)


# Evaluate accuracy
print(accuracy_score(test_labels, preds))
```

Now you can continue to work with your code to see if you can make your classifier perform even better. You could experiment with different subsets of features or even try completely different algorithms. Check out Scikit-learn's website at scikit-learn.org/stable for more machine learning ideas.

## Conclusion

In this tutorial, you learned how to build a machine learning classifier in Python. Now you can load data, organize data, train, predict, and evaluate machine learning classifiers in Python using Scikit-learn. The steps in this tutorial should help you facilitate the process of working with your own data in Python.

# How To Build a Neural Network to Recognize Handwritten Digits with TensorFlow

Written by Ellie Birbeck

Edited by Brian Hogan

Neural networks are used as a method of deep learning, one of the many subfields of artificial intelligence. They were first proposed around 70 years ago as an attempt at simulating the way the human brain works, though in a much more simplified form. Individual 'neurons' are connected in layers, with weights assigned to determine how the neuron responds when signals are propagated through the network. Previously, neural networks were limited in the number of neurons they were able to simulate, and therefore the complexity of learning they could achieve. But in recent years, due to advancements in hardware development, we have been able to build very deep networks, and train them on enormous datasets to achieve breakthroughs in machine intelligence.

These breakthroughs have allowed machines to match and exceed the capabilities of humans at performing certain tasks. One such task is object recognition. Though machines have historically been unable to match human vision, recent advances in deep learning have made it possible to build neural networks which can recognize objects, faces, text, and even emotions.

In this tutorial, you will implement a small subsection of object recognition—digit recognition. Using TensorFlow (https://www.tensorflow.org/), an open-source Python library developed by the Google Brain labs for deep learning research, you will

take hand-drawn images of the numbers 0-9 and build and train a neural network to recognize and predict the correct label for the digit displayed.

While you won't need prior experience in practical deep learning or TensorFlow to follow along with this tutorial, we'll assume some familiarity with machine learning terms and concepts such as training and testing, features and labels, optimization, and evaluation.

## Prerequisites

To complete this tutorial, you'll need a local or remote Python 3 development environment that includes pip for installing Python packages, and venv for creating virtual environments.

## Step 1 — Configuring the Project

Before you can develop the recognition program, you'll need to install a few dependencies and create a workspace to hold your files.

We'll use a Python 3 virtual environment to manage our project's dependencies. Create a new directory for your project and navigate to the new directory:

```
mkdir tensorflow-demo
cd tensorflow-demo
```

Execute the following commands to set up the virtual environment for this tutorial:

```
python3 -m venv tensorflow-demo
source tensorflow-demo/bin/activate
```

Next, install the libraries you'll use in this tutorial. We'll use specific versions of these libraries by creating a `requirements.txt` file in the project directory which specifies the requirement and the version we need. Create the `requirements.txt` file:

```
(tensorflow-demo) $ touch requirements.txt
```

Open the file in your text editor and add the following lines to specify the Image, NumPy, and TensorFlow libraries and their versions:

requirements.txt

```
image==1.5.20

numpy==1.14.3

tensorflow==1.4.0
```

Save the file and exit the editor. Then install these libraries with the following command:

```
(tensorflow-demo) $ pip install -r requirements.txt
```

With the dependencies installed, we can start working on our project.

## Step 2 — Importing the MNIST Dataset

The dataset we will be using in this tutorial is called the [MNIST](#) dataset, and it is a classic in the machine learning community. This dataset is made up of images of handwritten digits, 28x28 pixels in size. Here are some examples of the digits included in the dataset:
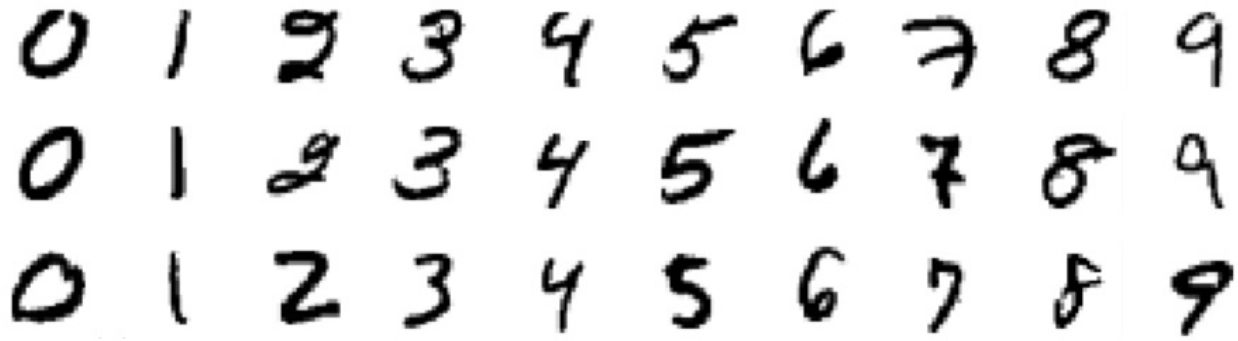
Examples of MNIST images

Let's create a Python program to work with this dataset. We will use one file for all of our work in this tutorial. Create a new file called `main.py`:

```
(tensorflow-demo) $ touch main.py
```

Now open this file in your text editor of choice and add this line of code to the file to import the TensorFlow library:

main.py

```
import tensorflow as tf
```

Add the following lines of code to your file to import the MNIST dataset and store the image data in the variable `mnist`:

main.py

```
...
from tensorflow.examples.tutorials.mnist import input_data
```

```
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True) # y
labels are oh-encoded
```

When reading in the data, we are using one-hot-encoding to represent the labels (the actual digit drawn, e.g. "3") of the images. One-hot-encoding uses a vector of binary values to represent numeric or categorical values. As our labels are for the digits 0-9, the vector contains ten values, one for each possible digit. One of these values is set to 1, to represent the digit at that index of the vector, and the rest are set to 0. For example, the digit 3 is represented using the vector `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]`. As the value at index 3 is stored as 1, the vector therefore represents the digit 3.

To represent the actual images themselves, the 28x28 pixels are flattened into a 1D vector which is 784 pixels in size. Each of the 784 pixels making up the image is stored as a value between 0 and 255. This determines the grayscale of the pixel, as our images are presented in black and white only. So a black pixel is represented by 255, and a white pixel by 0, with the various shades of gray somewhere in between.

We can use the `mnist` variable to find out the size of the dataset we have just imported. Looking at the `num_examples` for each of the three subsets, we can determine that the dataset has been split into 55,000 images for training, 5000 for validation, and 10,000 for testing. Add the following lines to your file:

main.py

```
...
n_train = mnist.train.num_examples  # 55,000
n_validation = mnist.validation.num_examples  # 5000
```

```
n_test = mnist.test.num_examples  # 10,000
```

Now that we have our data imported, it's time to think about the neural network.

## Step 3 — Defining the Neural Network Architecture

The architecture of the neural network refers to elements such as the number of layers in the network, the number of units in each layer, and how the units are connected between layers. As neural networks are loosely inspired by the workings of the human brain, here the term unit is used to represent what we would biologically think of as a neuron. Like neurons passing signals around the brain, units take some values from previous units as input, perform a computation, and then pass on the new value as output to other units. These units are layered to form the network, starting at a minimum with one layer for inputting values, and one layer to output values. The term hidden layer is used for all of the layers in between the input and output layers, i.e. those "hidden" from the real world.

Different architectures can yield dramatically different results, as the performance can be thought of as a function of the architecture among other things, such as the parameters, the data, and the duration of training.

Add the following lines of code to your file to store the number of units per layer in global variables. This allows us to alter the network architecture in one place, and at the end of the tutorial you can test for yourself how different numbers of layers and units will impact the results of our model:

main.py

```
...
n_input = 784  # input layer (28x28 pixels)

n_hidden1 = 512  # 1st hidden layer

n_hidden2 = 256  # 2nd hidden layer

n_hidden3 = 128  # 3rd hidden layer

n_output = 10  # output layer (0-9 digits)
```

The following diagram shows a visualization of the architecture we've designed, with each layer fully connected to the surrounding layers:
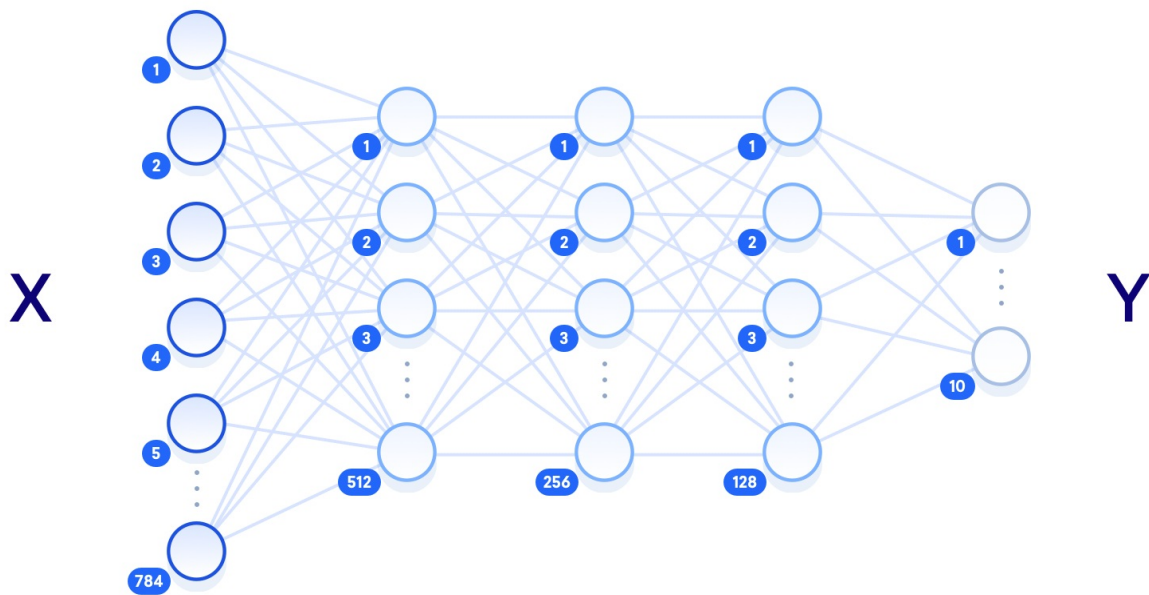


Diagram of a neural network

The term "deep neural network" relates to the number of hidden layers, with "shallow" usually meaning just one hidden layer, and "deep" referring to multiple hidden layers. Given enough training data, a shallow neural network with a sufficient number of units should

theoretically be able to represent any function that a deep neural network can. But it is often more computationally efficient to use a smaller deep neural network to achieve the same task that would require a shallow network with exponentially more hidden units. Shallow neural networks also often encounter overfitting, where the network essentially memorizes the training data that it has seen, and is not able to generalize the knowledge to new data. This is why deep neural networks are more commonly used: the multiple layers between the raw input data and the output label allow the network to learn features at various levels of abstraction, making the network itself better able to generalize.

Other elements of the neural network that need to be defined here are the hyperparameters. Unlike the parameters that will get updated during training, these values are set initially and remain constant throughout the process. In your file, set the following variables and values:

main.py

```
...
learning_rate = 1e-4
n_iterations = 1000
batch_size = 128
dropout = 0.5
```

The learning rate represents how much the parameters will adjust at each step of the learning process. These adjustments are a key component of training: after each pass through the network we tune the weights slightly to try and reduce the loss. Larger learning rates can converge faster, but also have the potential to overshoot the optimal values as they are updated. The number of iterations refers to how many times we go

through the training step, and the batch size refers to how many training examples we are using at each step. The `dropout` variable represents a threshold at which we eliminate some units at random. We will be using `dropout` in our final hidden layer to give each unit a 50% chance of being eliminated at every training step. This helps prevent overfitting.

We have now defined the architecture of our neural network, and the hyperparameters that impact the learning process. The next step is to build the network as a TensorFlow graph.

## Step 4 — Building the TensorFlow Graph

To build our network, we will set up the network as a computational graph for TensorFlow to execute. The core concept of TensorFlow is the tensor, a data structure similar to an array or list. initialized, manipulated as they are passed through the graph, and updated through the learning process.

We'll start by defining three tensors as placeholders, which are tensors that we'll feed values into later. Add the following to your file:

main.py

```
...
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_output])
keep_prob = tf.placeholder(tf.float32)
```

The only parameter that needs to be specified at its declaration is the size of the data we will be feeding in. For `X` we use a shape of `[None, 784]`, where `None` represents any amount, as we will be feeding in an undefined number of 784-pixel images. The shape of `Y` is `[None, 10]` as

we will be using it for an undefined number of label outputs, with 10 possible classes. The `keep_prob` tensor is used to control the dropout rate, and we initialize it as a placeholder rather than an immutable variable because we want to use the same tensor both for training (when `dropout` is set to `0.5`) and testing (when `dropout` is set to `1.0`).

The parameters that the network will update in the training process are the `weight` and `bias` values, so for these we need to set an initial value rather than an empty placeholder. These values are essentially where the network does its learning, as they are used in the activation functions of the neurons, representing the strength of the connections between units.

Since the values are optimized during training, we could set them to zero for now. But the initial value actually has a significant impact on the final accuracy of the model. We'll use random values from a truncated normal distribution for the weights. We want them to be close to zero, so they can adjust in either a positive or negative direction, and slightly different, so they generate different errors. This will ensure that the model learns something useful. Add these lines:

main.py

```
...
weights = {
    'w1': tf.Variable(tf.truncated_normal([n_input, n_hidden1],
stddev=0.1)),
    'w2': tf.Variable(tf.truncated_normal([n_hidden1, n_hidden2],
stddev=0.1)),
    'w3': tf.Variable(tf.truncated_normal([n_hidden2, n_hidden3],
stddev=0.1)),
    'out': tf.Variable(tf.truncated_normal([n_hidden3, n_output],
```

```
stddev=0.1)),

}
```

For the bias, we use a small constant value to ensure that the tensors activate in the intial stages and therefore contribute to the propagation. The weights and bias tensors are stored in dictionary objects for ease of access. Add this code to your file to define the biases:

main.py

```
...

biases = {

    'b1': tf.Variable(tf.constant(0.1, shape=[n_hidden1])),

    'b2': tf.Variable(tf.constant(0.1, shape=[n_hidden2])),

    'b3': tf.Variable(tf.constant(0.1, shape=[n_hidden3])),

    'out': tf.Variable(tf.constant(0.1, shape=[n_output]))

}
```

Next, set up the layers of the network by defining the operations that will manipulate the tensors. Add these lines to your file:

main.py

```
...
layer_1 = tf.add(tf.matmul(X, weights['w1']), biases['b1'])

layer_2 = tf.add(tf.matmul(layer_1, weights['w2']), biases['b2'])

layer_3 = tf.add(tf.matmul(layer_2, weights['w3']), biases['b3'])

layer_drop = tf.nn.dropout(layer_3, keep_prob)

output_layer = tf.matmul(layer_3, weights['out']) + biases['out']
```

Each hidden layer will execute matrix multiplication on the previous layer's outputs and the current layer's weights, and add the bias to these values. At the last hidden layer, we will apply a dropout operation using our `keep_prob` value of 0.5.

The final step in building the graph is to define the loss function that we want to optimize. A popular choice of loss function in TensorFlow programs is cross-entropy, also known as log-loss, which quantifies the difference between two probability distributions (the predictions and the labels). A perfect classification would result in a cross-entropy of 0, with the loss completely minimized.

We also need to choose the optimization algorithm which will be used to minimize the loss function. A process named gradient descent optimization is a common method for finding the (local) minimum of a function by taking iterative steps along the gradient in a negative (descending) direction. There are several choices of gradient descent optimization algorithms already implemented in TensorFlow, and in this tutorial we will be using the [Adam optimizer](). This extends upon gradient descent optimization by using momentum to speed up the process through computing an exponentially weighted average of the gradients and using that in the adjustments. Add the following code to your file:

main.py

```
...
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
        labels=Y, logits=output_layer
        ))
```

```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

We've now defined the network and built it out with TensorFlow. The next step is to feed data through the graph to train it, and then test that it has actually learnt something.

## Step 5 — Training and Testing

The training process involves feeding the training dataset through the graph and optimizing the loss function. Every time the network iterates through a batch of more training images, it updates the parameters to reduce the loss in order to more accurately predict the digits shown. The testing process involves running our testing dataset through the trained graph, and keeping track of the number of images that are correctly predicted, so that we can calculate the accuracy.

Before starting the training process, we will define our method of evaluating the accuracy so we can print it out on mini-batches of data while we train. These printed statements will allow us to check that from the first iteration to the last, loss decreases and accuracy increases; they will also allow us to track whether or not we have ran enough iterations to reach a consistent and optimal result:

main.py

```
...
correct_pred = tf.equal(tf.argmax(output_layer, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

In `correct_pred`, we use the `arg_max` function to compare which images are being predicted correctly by looking at the `output_layer`

(predictions) and `Y` (labels), and we use the `equal` function to return this as a list of [Booleans](#). We can then cast this list to floats and calculate the mean to get a total accuracy score.

We are now ready to initialize a session for running the graph. In this session we will feed the network with our training examples, and once trained, we feed the same graph with new test examples to determine the accuracy of the model. Add the following lines of code to your file:

main.py

```
...
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

The essence of the training process in deep learning is to optimize the loss function. Here we are aiming to minimize the difference between the predicted labels of the images, and the true labels of the images. The process involves four steps which are repeated for a set number of iterations:

- Propagate values forward through the network
- Compute the loss
- Propagate values backward through the network
- Update the parameters

At each training step, the parameters are adjusted slightly to try and reduce the loss for the next step. As the learning progresses, we should

see a reduction in loss, and eventually we can stop training and use the network as a model for testing our new data.

Add this code to the file:

main.py

```
...
# train on mini batches
for i in range(n_iterations):
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    sess.run(train_step, feed_dict={
        X: batch_x, Y: batch_y, keep_prob: dropout
        })

    # print loss and accuracy (per minibatch)
    if i % 100 == 0:
        minibatch_loss, minibatch_accuracy = sess.run(
            [cross_entropy, accuracy],
            feed_dict={X: batch_x, Y: batch_y, keep_prob: 1.0}
            )
        print(
            "Iteration",
            str(i),
            "\t| Loss =",
            str(minibatch_loss),
            "\t| Accuracy =",
            str(minibatch_accuracy)
            )
```

After 100 iterations of each training step in which we feed a mini-batch of images through the network, we print out the loss and accuracy of that batch. Note that we should not be expecting a decreasing loss and increasing accuracy here, as the values are per batch, not for the entire model. We use mini-batches of images rather than feeding them through individually to speed up the training process and allow the network to see a number of different examples before updating the parameters.

Once the training is complete, we can run the session on the test images. This time we are using a `keep_prob` dropout rate of `1.0` to ensure all units are active in the testing process.

Add this code to the file:

main.py
```
...
test_accuracy = sess.run(accuracy, feed_dict={X: mnist.test.images, Y:
mnist.test.labels, keep_prob: 1.0})
print("\nAccuracy on test set:", test_accuracy)
```

It's now time to run our program and see how accurately our neural network can recognize these handwritten digits. Save the `main.py` file and execute the following command in the terminal to run the script:

```
(tensorflow-demo) $ python main.py
```

You'll see an output similar to the following, although individual loss and accuracy results may vary slightly:

Output

```
Iteration 0      | Loss = 3.67079    | Accuracy = 0.140625

Iteration 100    | Loss = 0.492122   | Accuracy = 0.84375

Iteration 200    | Loss = 0.421595   | Accuracy = 0.882812

Iteration 300    | Loss = 0.307726   | Accuracy = 0.921875

Iteration 400    | Loss = 0.392948   | Accuracy = 0.882812

Iteration 500    | Loss = 0.371461   | Accuracy = 0.90625

Iteration 600    | Loss = 0.378425   | Accuracy = 0.882812

Iteration 700    | Loss = 0.338605   | Accuracy = 0.914062

Iteration 800    | Loss = 0.379697   | Accuracy = 0.875

Iteration 900    | Loss = 0.444303   | Accuracy = 0.90625


Accuracy on test set: 0.9206
```

To try and improve the accuracy of our model, or to learn more about the impact of tuning hyperparameters, we can test the effect of changing the learning rate, the dropout threshold, the batch size, and the number of iterations. We can also change the number of units in our hidden layers, and change the amount of hidden layers themselves, to see how different architectures increase or decrease the model accuracy.

To demonstrate that the network is actually recognizing the hand-drawn images, let's test it on a single image of our own.

If you are on a local machine and you would like to use your own hand-drawn number, you can use a graphics editor to create your own 28x28 pixel image of a digit. Otherwise, you can use `curl` to download the following sample test image to your server or computer:

```
(tensorflow-demo) $ curl -O images/test_img.png
```

Open the `main.py` file in your editor and add the following lines of code to the top of the file to import two libraries necessary for image manipulation.

main.py

```
import numpy as np
from PIL import Image
...
```

Then at the end of the file, add the following line of code to load the test image of the handwritten digit:

main.py

```
...
img = np.invert(Image.open("test_img.png").convert('L')).ravel()
```

The `open` function of the `Image` library loads the test image as a 4D array containing the three RGB color channels and the Alpha transparency. This is not the same representation we used previously when reading in the dataset with TensorFlow, so we'll need to do some extra work to match the format.

First, we use the `convert` function with the `L` parameter to reduce the 4D RGBA representation to one grayscale color channel. We store this as a `numpy` array and invert it using `np.invert`, because the current matrix represents black as 0 and white as 255, whereas we need the opposite. Finally, we call `ravel` to flatten the array.

Now that the image data is structured correctly, we can run a session in the same way as previously, but this time only feeding in the single

image for testing.

Add the following code to your file to test the image and print the outputted label.

main.py

```
...
prediction = sess.run(tf.argmax(output_layer, 1), feed_dict={X: [img]})
print ("Prediction for test image:", np.squeeze(prediction))
```

The `np.squeeze` function is called on the prediction to return the single integer from the array (i.e. to go from [2] to 2). The resulting output demonstrates that the network has recognized this image as the digit 2.

Output

```
Prediction for test image: 2
```

You can try testing the network with more complex images — digits that look like other digits, for example, or digits that have been drawn poorly or incorrectly — to see how well it fares.

## Conclusion

In this tutorial you successfully trained a neural network to classify the MNIST dataset with around 92% accuracy and tested it on an image of your own. Current state-of-the-art research achieves around 99% on this same problem, using more complex network architectures involving convolutional layers. These use the 2D structure of the image to better represent the contents, unlike our method which flattened all the pixels

into one vector of 784 units. You can read more about this topic on the [TensorFlow website](#), and see the research papers detailing the most accurate results on the [MNIST website](#).

Now that you know how to build and train a neural network, you can try and use this implementation on your own data, or test it on other popular datasets such as the [Google StreetView House Numbers](#), or the [CIFAR-10](#) dataset for more general image recognition.

# Bias-Variance for Deep Reinforcement Learning: How To Build a Bot for Atari with OpenAI Gym
Written by Alvin Wan
Edited by Mark Drake

Reinforcement learning is a subfield within control theory, which concerns controlling systems that change over time and broadly includes applications such as self-driving cars, robotics, and bots for games. Throughout this guide, you will use reinforcement learning to build a bot for Atari video games. This bot is not given access to internal information about the game. Instead, it's only given access to the game's rendered display and the reward for that display, meaning that it can only see what a human player would see.

In machine learning, a bot is formally known as an agent. In the case of this tutorial, an agent is a "player" in the system that acts according to a decision-making function, called a policy. The primary goal is to develop strong agents by arming them with strong policies. In other words, our aim is to develop intelligent bots by arming them with strong decision-making capabilities.

You will begin this tutorial by training a basic reinforcement learning agent that takes random actions when playing Space Invaders, the classic Atari arcade game, which will serve as your baseline for comparison. Following this, you will explore several other techniques — including Q-learning, deep Q-learning, and least squares — while building agents that play Space Invaders and Frozen Lake, a simple game environment included in Gym (https://gym.openai.com/), a reinforcement learning toolkit released by OpenAI (https://openai.com/). By following this

tutorial, you will gain an understanding of the fundamental concepts that govern one's choice of model complexity in machine learning.

## Prerequisites

To complete this tutorial, you will need:

- A server running Ubuntu 18.04, with at least 1GB of RAM. This server should have a non-root user with `sudo` privileges configured, as well as a firewall set up with UFW. You can set this up by following this [Initial Server Setup Guide for Ubuntu 18.04](#).
- A Python 3 virtual environment which you can achieve by reading our guide "[How To Install Python 3 and Set Up a Programming Environment on an Ubuntu 18.04 Server](#)."

Alternatively, if you are using a local machine, you can install Python 3 and set up a local programming environment by reading the appropriate tutorial for your operating system via our [Python Installation and Setup Series](#).

## Step 1 — Creating the Project and Installing Dependencies

In order to set up the development environment for your bots, you must download the game itself and the libraries needed for computation.

Begin by creating a workspace for this project named `AtariBot`:

```
mkdir ~/AtariBot
```

Navigate to the new `AtariBot` directory:

```
cd ~/AtariBot
```

Then create a new virtual environment for the project. You can name this virtual environment anything you'd like; here, we will name it ataribot:

```
python3 -m venv ataribot
```

Activate your environment:

```
source ataribot/bin/activate
```

On Ubuntu, as of version 16.04, OpenCV requires a few more packages t o be installed in order to function. These include CMake — an application that manages software build processes — as well as a session manager, miscellaneous extensions, and digital image composition. Run the following command to install these packages:

```
sudo apt-get install -y cmake libsm6 libxext6 libxrender-dev libz-dev
```

NOTE: If you're following this guide on a local machine running MacOS, the only additional software you need to install is CMake. Install it using Homebrew (which you will have installed if you followed the prerequisite MacOS tutorial) by typing:

```
brew install cmake
```

Next, use `pip` to install the `wheel` package, the reference implementation of the wheel packaging standard. A Python library, this package serves as an extension for building wheels and includes a command line tool for working with `.whl` files:

```
python -m pip install wheel
```

In addition to `wheel`, you'll need to install the following packages:

- [Gym](), a Python library that makes various games available for research, as well as all dependencies for the Atari games. Developed by [OpenAI](), Gym offers public benchmarks for each of the games so that the performance for various agents and algorithms can be uniformly /evaluated.
- [Tensorflow](), a deep learning library. This library gives us the ability to run computations more efficiently. Specifically, it does this by building mathematical functions using Tensorflow's abstractions that run exclusively on your GPU.
- [OpenCV](), the computer vision library mentioned previously.
- [SciPy](), a scientific computing library that offers efficient optimization algorithms.
- [NumPy](), a linear algebra library.

Install each of these packages with the following command. Note that this command specifies which version of each package to install:

```
python -m pip install gym==0.9.5 tensorflow==1.5.0 tensorpack==0.8.0
numpy==1.14.0 scipy==1.1.0 opencv-python==3.4.1.15
```

Following this, use `pip` once more to install Gym's Atari environments, which includes a variety of Atari video games, including Space Invaders:

```
python -m pip install gym[atari]
```

If your installation of the `gym[atari]` package was successful, your output will end with the following:

Output

```
Installing collected packages: atari-py, Pillow, PyOpenGL
Successfully installed Pillow-5.4.1 PyOpenGL-3.1.0 atari-py-0.1.7
```

With these dependencies installed, you're ready to move on and build an agent that plays randomly to serve as your baseline for comparison.

## Step 2 — Creating a Baseline Random Agent with Gym

Now that the required software is on your server, you will set up an agent that will play a simplified version of the classic Atari game, Space Invaders. For any experiment, it is necessary to obtain a baseline to help you understand how well your model performs. Because this agent takes random actions at each frame, we'll refer to it as our random, baseline agent. In this case, you will compare against this baseline agent to understand how well your agents perform in later steps.

With Gym, you maintain your own game loop. This means that you handle every step of the game's execution: at every time step, you give the `gym` a new action and ask `gym` for the game state. In this tutorial, the

game state is the game's appearance at a given time step, and is precisely what you would see if you were playing the game.

Using your preferred text editor, create a Python file named `bot_2_random.py`. Here, we'll use `nano`:

```
nano bot_2_random.py
```

Note: Throughout this guide, the bots' names are aligned with the Step number in which they appear, rather than the order in which they appear. Hence, this bot is named `bot\_2\_random.py` rather than `bot\_1\_random.py`.

Start this script by adding the following highlighted lines. These lines include a comment block that explains what this script will do and two `import` statements that will import the packages this script will ultimately need in order to function:

/AtariBot/bot_2_random.py

```
"""
Bot 2 -- Make a random, baseline agent for the SpaceInvaders game.
"""

import gym
import random
```

Add a `main` function. In this function, create the game environment — `SpaceInvaders-v0` — and then initialize the game using `env.reset`:

/AtariBot/bot_2_random.py

```
. . .

import gym

import random


def main():

env = gym.make('SpaceInvaders-v0')

env.reset()
```

Next, add an `env.step` function. This function can return the following kinds of values:

- `state`: The new state of the game, after applying the provided action.
- `reward`: The increase in score that the state incurs. By way of example, this could be when a bullet has destroyed an alien, and the score increases by 50 points. Then, `reward = 50`. In playing any score-based game, the player's goal is to maximize the score. This is synonymous with maximizing the total reward.
- `done`: Whether or not the episode has ended, which usually occurs when a player has lost all lives.
- `info`: Extraneous information that you'll put aside for now.

You will use `reward` to count your total reward. You'll also use `done` to determine when the player dies, which will be when `done` returns `True`.

Add the following game loop, which instructs the game to loop until the player dies:

/AtariBot/bot_2_random.py

```
. . .

def main():

env = gym.make('SpaceInvaders-v0')

env.reset()


    episode_reward = 0

    while True:

        action = env.action_space.sample()

        _, reward, done, _ = env.step(action)

        episode_reward += reward

        if done:

            print('Reward: %s' % episode_reward)

            break
```

Finally, run the `main` function. Include a `__name__` check to ensure that `main` only runs when you invoke it directly with `python bot_2_random.py`. If you do not add the `if` check, `main` will always be triggered when the Python file is executed, even when you import the file. Consequently, it's a good practice to place the code in a `main` function, executed only when `__name__ == '__main__'`.

/AtariBot/bot_2_random.py

```
. . .

def main():

. . .

if done:

print('Reward %s' % episode_reward)
```

```
break
```

```
if **name** == '**main**':
main()
```

Save the file and exit the editor. If you're using `nano`, do so by pressing `CTRL+X`, `Y`, then `ENTER`. Then, run your script by typing:

```
python bot_2_random.py
```

Your program will output a number, akin to the following. Note that each time you run the file you will get a different result:

Output
```
Making new env: SpaceInvaders-v0
Reward: 210.0
```

These random results present an issue. In order to produce work that other researchers and practitioners can benefit from, your results and trials must be reproducible. To correct this, reopen the script file:

```
nano bot_2_random.py
```

After `import random,` add `random.seed(0)`. After `env = gym.make('SpaceInvaders-v0'),` add `env.seed(0)`. Together, these lines "seed" the environment with a consistent starting point, ensuring that the results will always be reproducible. Your final file will match the following, exactly:

/AtariBot/bot_2_random.py

```python
"""
Bot 2 -- Make a random, baseline agent for the SpaceInvaders game.
"""

import gym
import random


random.seed(0)


def main():
    env = gym.make('SpaceInvaders-v0')
    env.seed(0)

    env.reset()
    episode_reward = 0
    while True:
        action = env.action_space.sample()
        _, reward, done, _ = env.step(action)
        episode_reward += reward
        if done:
            print('Reward: %s' % episode_reward)
            break


if **name** == '**main**':
    main()
```

Save the file and close your editor, then run the script by typing the following in your terminal:

```
python bot_2_random.py
```

This will output the following reward, exactly:

Output
```
Making new env: SpaceInvaders-v0
Reward: 555.0
```

This is your very first bot, although it's rather unintelligent since it doesn't account for the surrounding environment when it makes decisions. For a more reliable estimate of your bot's performance, you could have the agent run for multiple episodes at a time, reporting rewards averaged across multiple episodes. To configure this, first reopen the file:

```
nano bot_2_random.py
```

After `random.seed(0)`, add the following highlighted line which tells the agent to play the game for 10 episodes:

/AtariBot/bot_2_random.py
```
. . .
random.seed(0)

num_episodes = 10
```

. . .

Right after `env.seed(0)`, start a new list of rewards:

/AtariBot/bot_2_random.py

```
. . .
env.seed(0)
rewards = []
. . .
```

Nest all code from `env.reset()` to the end of `main()` in a `for` loop, iterating `num_episodes` times. Make sure to indent each line from `env.reset()` to `break` by four spaces:

/AtariBot/bot_2_random.py

```
. . .
def main():
env = gym.make('SpaceInvaders-v0')
env.seed(0)
rewards = []

    for _ in range(num_episodes):
        env.reset()
        episode_reward = 0

        while True:
            ...
```

Right before `break`, currently the last line of the main game loop, add the current episode's reward to the list of all rewards:

/AtariBot/bot_2_random.py

```
. . .
if done:
print('Reward: %s' % episode_reward)
rewards.append(episode_reward)
break
. . .
```

At the end of the `main` function, report the average reward:

/AtariBot/bot_2_random.py

```
. . .
def main():
...
print('Reward: %s' % episode_reward)
break
print('Average reward: %.2f' % (sum(rewards) / len(rewards)))
. . .
```

Your file will now align with the following. Please note that the following code block includes a few comments to clarify key parts of the script:

/AtariBot/bot_2_random.py

```
"""
```

```
Bot 2 -- Make a random, baseline agent for the SpaceInvaders game.
"""


import gym
import random


random.seed(0)  # make results reproducible


num_episodes = 10


def main():
    env = gym.make('SpaceInvaders-v0')  # create the game
    env.seed(0)  # make results reproducible
    rewards = []

    for _ in range(num_episodes):
        env.reset()
        episode_reward = 0
        while True:
            action = env.action_space.sample()
            _, reward, done, _ = env.step(action)  # random action
            episode_reward += reward
            if done:
                print('Reward: %d' % episode_reward)
                rewards.append(episode_reward)
                break
    print('Average reward: %.2f' % (sum(rewards) / len(rewards)))
```

```
if __name__ == '__main__':

    main()
```

Save the file, exit the editor, and run the script:

```
python bot_2_random.py
```

This will print the following average reward, exactly:

Output

```
Making new env: SpaceInvaders-v0

. . .

Average reward: 163.50
```

We now have a more reliable estimate of the baseline score to beat. To create a superior agent, though, you will need to understand the framework for reinforcement learning. How can one make the abstract notion of "decision-making" more concrete?

## Understanding Reinforcement Learning

In any game, the player's goal is to maximize their score. In this guide, the player's score is referred to as its reward. To maximize their reward, the player must be able to refine its decision-making abilities. Formally, a decision is the process of looking at the game, or observing the game's state, and picking an action. Our decision-making function is called a policy; a policy accepts a state as input and "decides" on an action:

```
policy: state -> action
```

To build such a function, we will start with a specific set of algorithms in reinforcement learning called Q-learning algorithms. To illustrate these, consider the initial state of a game, which we'll call `state0`: your spaceship and the aliens are all in their starting positions. Then, assume we have access to a magical "Q-table" which tells us how much reward each action will earn:

| STATE | ACTION | REWARD |
|-------|--------|--------|
| state0 | shoot | 10 |
| state0 | right | 3 |
| state0 | left | 3 |

The `shoot` action will maximize your reward, as it results in the reward with the highest value: 10. As you can see, a Q-table provides a straightforward way to make decisions, based on the observed state:

```
policy: state -> look at Q-table, pick action with greatest reward
```

However, most games have too many states to list in a table. In such cases, the Q-learning agent learns a Q-function instead of a Q-table. We use this Q-function similarly to how we used the Q-table previously. Rewriting the table entries as functions gives us the following:

```
Q(state0, shoot) = 10
Q(state0, right) = 3
```

```
Q(state0, left) = 3
```

Given a particular state, it's easy for us to make a decision: we simply look at each possible action and its reward, then take the action that corresponds with the highest expected reward. Reformulating the earlier policy more formally, we have:

```
policy: state -> argmax_{action} Q(state, action)
```

This satisfies the requirements of a decision-making function: given a state in the game, it decides on an action. However, this solution depends on knowing `Q(state, action)` for every state and action. To estimate `Q(state, action)`, consider the following:

1. Given many observations of an agent's states, actions, and rewards, one can obtain an estimate of the reward for every state and action by taking a running average.
2. Space Invaders is a game with delayed rewards: the player is rewarded when the alien is blown up and not when the player shoots. However, the player taking an action by shooting is the true impetus for the reward. Somehow, the Q-function must assign `(state0, shoot)` a positive reward.

These two insights are codified in the following equations:

```
Q(state, action) = (1 - learning_rate) * Q(state, action) +
learning_rate * Q_target
Q_target = reward + discount_factor * max_{action'} Q(state', action')
```

These equations use the following definitions:

- `state`: the state at current time step
- `action`: the action taken at current time step
- `reward`: the reward for current time step
- `state'`: the new state for next time step, given that we took action `a`
- `action'`: all possible actions
- `learning_rate`: the learning rate
- `discount_factor`: the discount factor, how much reward "degrades" as we propagate it

For a complete explanation of these two equations, see this article on [Understanding Q-Learning](#).

With this understanding of reinforcement learning in mind, all that remains is to actually run the game and obtain these Q-value estimates for a new policy.

## Step 3 — Creating a Simple Q-learning Agent for Frozen Lake

Now that you have a baseline agent, you can begin creating new agents and compare them against the original. In this step, you will create an agent that uses [Q-learning](#), a reinforcement learning technique used to teach an agent which action to take given a certain state. This agent will play a new game, [FrozenLake](#). The setup for this game is described as follows on the Gym website:

*Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing*

*water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend.*

*The surface is described using a grid like the following:*

```
SFFF        (S: starting point, safe)

FHFH        (F: frozen surface, safe)

FFFH        (H: hole, fall to your doom)

HFFG        (G: goal, where the frisbee is located)
```

The player starts at the top left, denoted by `S`, and works its way to the goal at the bottom right, denoted by `G`. The available actions are right, left, up, and down, and reaching the goal results in a score of 1. There are a number of holes, denoted `H`, and falling into one immediately results in a score of 0.

In this section, you will implement a simple Q-learning agent. Using what you've learned previously, you will create an agent that trades off between exploration and exploitation. In this context, exploration means the agent acts randomly, and exploitation means it uses its Q-values to choose what it believes to be the optimal action. You will also create a table to hold the Q-values, updating it incrementally as the agent acts and learns.

Make a copy of your script from Step 2:

```
cp bot_2_random.py bot_3_q_table.py
```

Then open up this new file for editing:

```
nano bot_3_q_table.py
```

Begin by updating the comment at the top of the file that describes the script's purpose. Because this is only a comment, this change isn't necessary for the script to function properly, but it can be helpful for keeping track of what the script does:

/AtariBot/bot_3_q_table.py

```
"""
Bot 3 -- Build simple q-learning agent for FrozenLake
"""
. . .
```

Before you make functional modifications to the script, you will need to import `numpy` for its linear algebra utilities. Right underneath `import gym`, add the highlighted line:

/AtariBot/bot_3_q_table.py

```
"""
Bot 3 -- Build simple q-learning agent for FrozenLake
"""

import gym
import numpy as np
import random

random.seed(0) # make results reproducible
. . .
```

Underneath `random.seed(0)`, add a seed for `numpy`:

/AtariBot/bot_3_q_table.py

```
. . .
import random

random.seed(0) # make results reproducible
np.random.seed(0)
. . .
```

Next, make the game states accessible. Update the `env.reset()` line to say the following, which stores the initial state of the game in the variable `state`:

/AtariBot/bot_3_q_table.py

```
. . .
for \_  in range(num_episodes):
state = env.reset()
. . .
```

Update the `env.step(...)` line to say the following, which stores the next state, `state2`. You will need both the current `state` and the next one — `state2` — to update the Q-function.

/AtariBot/bot_3_q_table.py

```
        . . .
        while True:
            action = env.action_space.sample()
```

```
            state2, reward, done, _ = env.step(action)

            . . .
```

After `episode_reward += reward`, add a line updating the variable `state`. This keeps the variable `state` updated for the next iteration, as you will expect `state` to reflect the current state:

/AtariBot/bot_3_q_table.py

```
. . .
while True:
. . .
episode_reward += reward
state = state2
if done:
. . .
```

In the `if done` block, delete the `print` statement which prints the reward for each episode. Instead, you'll output the average reward over many episodes. The `if done` block will then look like this:

/AtariBot/bot_3_q_table.py

```
            . . .
        if done:
            rewards.append(episode_reward)
            break
            . . .
```

After these modifications your game loop will match the following:

/AtariBot/bot_3_q_table.py

```
. . .

    for _ in range(num_episodes):
        state = env.reset()
        episode_reward = 0
        while True:
            action = env.action_space.sample()
            state2, reward, done, _ = env.step(action)
            episode_reward += reward
            state = state2
            if done:
                rewards.append(episode_reward))
                break
                . . .
```

Next, add the ability for the agent to trade off between exploration and exploitation. Right before your main game loop (which starts with `for...`), create the Q-value table:

/AtariBot/bot_3_q_table.py

```
. . .

    Q = np.zeros((env.observation_space.n, env.action_space.n))
    for _ in range(num_episodes):
        . . .
```

Then, rewrite the `for` loop to expose the episode number:

/AtariBot/bot_3_q_table.py

```
. . .

    Q = np.zeros((env.observation_space.n, env.action_space.n))
    for episode in range(1, num_episodes + 1):

        . . .
```

Inside the `while True:` inner game loop, create `noise`. Noise, or meaningless, random data, is sometimes introduced when training deep neural networks because it can improve both the performance and the accuracy of the model. Note that the higher the noise, the less the values in `Q[state, :]` matter. As a result, the higher the noise, the more likely that the agent acts independently of its knowledge of the game. In other words, higher noise encourages the agent to explore random actions:

/AtariBot/bot_3_q_table.py

```
        . . .

        while True:

            noise = np.random.random((1, env.action_space.n)) /
(episode**2.)

            action = env.action_space.sample()

            . . .
```

Note that as `episodes` increases, the amount of noise decreases quadratically: as time goes on, the agent explores less and less because it can trust its own assessment of the game's reward and begin to exploit its knowledge.

Update the `action` line to have your agent pick actions according to the Q-value table, with some exploration built in:

/AtariBot/bot_3_q_table.py

```
            . . .
            noise = np.random.random((1, env.action_space.n)) /
(episode**2.)
            action = np.argmax(Q[state, :] + noise)
            state2, reward, done, _ = env.step(action)
            . . .
```

Your main game loop will then match the following:

/AtariBot/bot_3_q_table.py

```
. . .
    Q = np.zeros((env.observation_space.n, env.action_space.n))
    for episode in range(1, num_episodes + 1):
        state = env.reset()
        episode_reward = 0
        while True:
            noise = np.random.random((1, env.action_space.n)) /
(episode**2.)
            action = np.argmax(Q[state, :] + noise)
            state2, reward, done, _ = env.step(action)
            episode_reward += reward
            state = state2
            if done:
                rewards.append(episode_reward)
                break
                . . .
```

Next, you will update your Q-value table using the [Bellman update equation](#), an equation widely used in machine learning to find the optimal policy within a given environment.

The Bellman equation incorporates two ideas that are highly relevant to this project. First, taking a particular action from a particular state many times will result in a good estimate for the Q-value associated with that state and action. To this end, you will increase the number of episodes this bot must play through in order to return a stronger Q-value estimate. Second, rewards must propagate through time, so that the original action is assigned a non-zero reward. This idea is clearest in games with delayed rewards; for example, in Space Invaders, the player is rewarded when the alien is blown up and not when the player shoots. However, the player shooting is the true impetus for a reward. Likewise, the Q-function must assign (`state0`, `shoot`) a positive reward.

First, update `num_episodes` to equal 4000:

/AtariBot/bot_3_q_table.py

```
. . .
np.random.seed(0)


num_episodes = 4000
. . .
```

Then, add the necessary hyperparameters to the top of the file in the form of two more variables:

/AtariBot/bot_3_q_table.py

```
. . .
```

```
num_episodes = 4000

discount_factor = 0.8

learning_rate = 0.9

. . .
```

Compute the new target Q-value, right after the line containing `env.step(...)`:

/AtariBot/bot_3_q_table.py

```
        . . .

        state2, reward, done, _ = env.step(action)

        Qtarget = reward + discount_factor * np.max(Q[state2, :])

        episode_reward += reward

        . . .
```

On the line directly after `Qtarget`, update the Q-value table using a weighted average of the old and new Q-values:

/AtariBot/bot_3_q_table.py

```
        . . .

        Qtarget = reward + discount_factor * np.max(Q[state2, :])

        Q[state, action] = (
            1-learning_rate
        ) * Q[state, action] + learning_rate * Qtarget

        episode_reward += reward

        . . .
```

Check that your main game loop now matches the following:

```
. . .

    Q = np.zeros((env.observation_space.n, env.action_space.n))

    for episode in range(1, num_episodes + 1):

        state = env.reset()

        episode_reward = 0

        while True:

            noise = np.random.random((1, env.action_space.n)) /
(episode**2.)

            action = np.argmax(Q[state, :] + noise)

            state2, reward, done, _ = env.step(action)

            Qtarget = reward + discount_factor * np.max(Q[state2, :])

            Q[state, action] = (

                1-learning_rate

                ) * Q[state, action] + learning_rate * Qtarget

            episode_reward += reward

            state = state2

            if done:

                rewards.append(episode_reward)

                break

                . . .
```

Our logic for training the agent is now complete. All that's left is to add reporting mechanisms.

Even though Python does not enforce strict type checking, add types to your function declarations for cleanliness. At the top of the file, before the first line reading `import gym`, import the `List` type:

/AtariBot/bot_3_q_table.py

. . .

```
from typing import List
import gym
```

. . .

Right after `learning_rate = 0.9`, outside of the `main` function, declare the interval and format for reports:

/AtariBot/bot_3_q_table.py

. . .

```
learning_rate = 0.9
report_interval = 500
report = '100-ep Average: %.2f . Best 100-ep Average: %.2f . Average: %.2f ' \
         '(Episode %d)'


def main():
```

. . .

Before the `main` function, add a new function that will populate this `report` string, using the list of all rewards:

/AtariBot/bot_3_q_table.py

. . .

```
report = '100-ep Average: %.2f . Best 100-ep Average: %.2f . Average: %.2f ' \
 '(Episode %d)'
```

```python
def print_report(rewards: List, episode: int):
    """Print rewards report for current episode
    - Average for last 100 episodes
    - Best 100-episode average across all time
    - Average for all episodes across time
    """
    print(report % (
        np.mean(rewards[-100:]),
        max([np.mean(rewards[i:i+100]) for i in range(len(rewards) - 100)]),
        np.mean(rewards),
        episode))


def main():
    . . .
```

Change the game to `FrozenLake` instead of `SpaceInvaders`:

/AtariBot/bot_3_q_table.py

```python
. . .
def main():
    env = gym.make('FrozenLake-v0') # create the game
    . . .
```

After `rewards.append(...)`, print the average reward over the last 100 episodes and print the average reward across all episodes:

/AtariBot/bot_3_q_table.py

```
            . . .
        if done:
            rewards.append(episode_reward)
            if episode % report_interval == 0:
                print_report(rewards, episode)
            . . .
```

At the end of the `main()` function, report both averages once more. Do this by replacing the line that reads `print('Average reward: %.2f' % (sum(rewards) / len(rewards)))` with the following highlighted line:

/AtariBot/bot_3_q_table.py

```
. . .
def main():
...
break
print_report(rewards, -1)
. . .
```

Finally, you have completed your Q-learning agent. Check that your script aligns with the following:

/AtariBot/bot_3_q_table.py

```
"""
Bot 3 -- Build simple q-learning agent for FrozenLake
"""
```

```python
from typing import List

import gym

import numpy as np

import random


random.seed(0)  # make results reproducible
np.random.seed(0)  # make results reproducible


num_episodes = 4000
discount_factor = 0.8
learning_rate = 0.9
report_interval = 500
report = '100-ep Average: %.2f . Best 100-ep Average: %.2f . Average: %.2f ' \
        '(Episode %d)'


def print_report(rewards: List, episode: int):
    """Print rewards report for current episode
    - Average for last 100 episodes
    - Best 100-episode average across all time
    - Average for all episodes across time
    """
    print(report % (
        np.mean(rewards[-100:]),
        max([np.mean(rewards[i:i+100]) for i in range(len(rewards) - 100)]),
```

```python
            np.mean(rewards),

            episode))




def main():

    env = gym.make('FrozenLake-v0')  # create the game

    env.seed(0)  # make results reproducible

    rewards = []


    Q = np.zeros((env.observation_space.n, env.action_space.n))

    for episode in range(1, num_episodes + 1):

        state = env.reset()

        episode_reward = 0

        while True:

            noise = np.random.random((1, env.action_space.n)) /
(episode**2.)

            action = np.argmax(Q[state, :] + noise)

            state2, reward, done, _ = env.step(action)

            Qtarget = reward + discount_factor * np.max(Q[state2, :])

            Q[state, action] = (

                1-learning_rate

                ) * Q[state, action] + learning_rate * Qtarget

            episode_reward += reward

            state = state2

            if done:

                rewards.append(episode_reward)

                if episode % report_interval == 0:

                    print_report(rewards, episode)
```

```
            break

    print_report(rewards, -1)


if __name__ == '__main__':
    main()
```

Save the file, exit your editor, and run the script:

```
python bot_3_q_table.py
```

Your output will match the following:

Output

```
100-ep Average: 0.11 . Best 100-ep Average: 0.12 . Average: 0.03

(Episode 500)

100-ep Average: 0.25 . Best 100-ep Average: 0.24 . Average: 0.09

(Episode 1000)

100-ep Average: 0.39 . Best 100-ep Average: 0.48 . Average: 0.19

(Episode 1500)

100-ep Average: 0.43 . Best 100-ep Average: 0.55 . Average: 0.25

(Episode 2000)

100-ep Average: 0.44 . Best 100-ep Average: 0.55 . Average: 0.29

(Episode 2500)

100-ep Average: 0.64 . Best 100-ep Average: 0.68 . Average: 0.32

(Episode 3000)

100-ep Average: 0.63 . Best 100-ep Average: 0.71 . Average: 0.36

(Episode 3500)

100-ep Average: 0.56 . Best 100-ep Average: 0.78 . Average: 0.40
```

```
(Episode 4000)

100-ep Average: 0.56 . Best 100-ep Average: 0.78 . Average: 0.40

(Episode -1)
```

You now have your first non-trivial bot for games, but let's put this average reward of `0.78` into perspective. According to the [Gym FrozenLake page](#), "solving" the game means attaining a 100-episode average of `0.78`. Informally, "solving" means "plays the game very well". While not in record time, the Q-table agent is able to solve FrozenLake in 4000 episodes.

However, the game may be more complex. Here, you used a table to store all of the 144 possible states, but consider tic tac toe in which there are 19,683 possible states. Likewise, consider Space Invaders where there are too many possible states to count. A Q-table is not sustainable as games grow increasingly complex. For this reason, you need some way to approximate the Q-table. As you continue experimenting in the next step, you will design a function that can accept states and actions as inputs and output a Q-value.

## Step 4 — Building a Deep Q-learning Agent for Frozen Lake

In reinforcement learning, the neural network effectively predicts the value of Q based on the `state` and `action` inputs, using a table to store all the possible values, but this becomes unstable in complex games. Deep reinforcement learning instead uses a neural network to approximate the Q-function. For more details, see [Understanding Deep Q-Learning](#).

To get accustomed to [Tensorflow](#), a deep learning library you installed in Step 1, you will reimplement all of the logic used so far with

Tensorflow's abstractions and you'll use a neural network to approximate your Q-function. However, your neural network will be extremely simple: your output `Q(s)` is a matrix `W` multiplied by your input `s`. This is known as a neural network with one fully-connected layer:

```
Q(s) = Ws
```

To reiterate, the goal is to reimplement all of the logic from the bots we've already built using Tensorflow's abstractions. This will make your operations more efficient, as Tensorflow can then perform all computation on the GPU.

Begin by duplicating your Q-table script from Step 3:

```
cp bot_3_q_table.py bot_4_q_network.py
```

Then open the new file with `nano` or your preferred text editor:

```
nano bot_4_q_network.py
```

First, update the comment at the top of the file:

/AtariBot/bot_4_q_network.py

```
"""
Bot 4 -- Use Q-learning network to train bot
"""



. . .
```

Next, import the Tensorflow package by adding an `import` directive right below `import random`. Additionally, add `tf.set_radon_seed(0)` right below `np.random.seed(0)`. This will ensure that the results of this script will be repeatable across all sessions:

/AtariBot/bot_4_q_network.py

```
. . .
import random
import tensorflow as tf


random.seed(0)
np.random.seed(0)
tf.set_random_seed(0)
. . .
```

Redefine your hyperparameters at the top of the file to match the following and add a function called `exploration_probability`, which will return the probability of exploration at each step. Remember that, in this context, "exploration" means taking a random action, as opposed to taking the action recommended by the Q-value estimates:

/AtariBot/bot_4_q_network.py

```
. . .
num_episodes = 4000
discount_factor = 0.99
learning_rate = 0.15
report_interval = 500
exploration_probability = lambda episode: 50. / (episode + 10)
```

```
report = '100-ep Average: %.2f . Best 100-ep Average: %.2f . Average:
%.2f ' \
 '(Episode %d)'
. . .
```

Next, you will add a one-hot encoding function. In short, one-hot encoding is a process through which variables are converted into a form that helps machine learning algorithms make better predictions. If you'd like to learn more about one-hot encoding, you can check out [Adversarial Examples in Computer Vision: How to Build then Fool an Emotion-Based Dog Filter](#).

Directly beneath `report = ...`, add a `one_hot` function:

/AtariBot/bot_4_q_network.py

```
. . .
report = '100-ep Average: %.2f . Best 100-ep Average: %.2f . Average:
%.2f ' \
 '(Episode %d)'

def one_hot(i: int, n: int) -> np.array:
"""Implements one-hot encoding by selecting the ith standard basis
vector"""
return np.identity(n)[i].reshape((1, -1))


def print_report(rewards: List, episode: int):
. . .
```

Next, you will rewrite your algorithm logic using Tensorflow's abstractions. Before doing that, though, you'll need to first create placeholders for your data.

In your `main` function, directly beneath `rewards=[]`, insert the following highlighted content. Here, you define placeholders for your observation at time t (as `obs_t_ph`) and time t+1 (as `obs_tp1_ph`), as well as placeholders for your action, reward, and Q target:

/AtariBot/bot_4_q_network.py

```
. . .
def main():
env = gym.make('FrozenLake-v0') # create the game
env.seed(0) # make results reproducible
rewards = []


    # 1. Setup placeholders
    n_obs, n_actions = env.observation_space.n, env.action_space.n
    obs_t_ph = tf.placeholder(shape=[1, n_obs], dtype=tf.float32)
    obs_tp1_ph = tf.placeholder(shape=[1, n_obs], dtype=tf.float32)
    act_ph = tf.placeholder(tf.int32, shape=())
    rew_ph = tf.placeholder(shape=(), dtype=tf.float32)
    q_target_ph = tf.placeholder(shape=[1, n_actions], dtype=tf.float32)


    Q = np.zeros((env.observation_space.n, env.action_space.n))
    for episode in range(1, num_episodes + 1):
        . . .
```

Directly beneath the line beginning with `q_target_ph =`, insert the following highlighted lines. This code starts your computation by computing Q(s, a) for all a to make `q_current` and Q(s', a') for all a' to make `q_target`:

/AtariBot/bot_4_q_network.py

```
    . . .

    rew_ph = tf.placeholder(shape=(), dtype=tf.float32)

    q_target_ph = tf.placeholder(shape=[1, n_actions], dtype=tf.float32)


    # 2. Setup computation graph

    W = tf.Variable(tf.random_uniform([n_obs, n_actions], 0, 0.01))

    q_current = tf.matmul(obs_t_ph, W)

    q_target = tf.matmul(obs_tp1_ph, W)


    Q = np.zeros((env.observation_space.n, env.action_space.n))

    for episode in range(1, num_episodes + 1):

        . . .
```

Again directly beneath the last line you added, insert the following higlighted code. The first two lines are equivalent to the line added in Step 3 that computes `Qtarget`, where `Qtarget = reward + discount_factor * np.max(Q[state2, :])`. The next two lines set up your loss, while the last line computes the action that maximizes your Q-value:

/AtariBot/bot_4_q_network.py

```
        . . .
```

```python
    q_current = tf.matmul(obs_t_ph, W)
    q_target = tf.matmul(obs_tp1_ph, W)


    q_target_max = tf.reduce_max(q_target_ph, axis=1)
    q_target_sa = rew_ph + discount_factor * q_target_max
    q_current_sa = q_current[0, act_ph]
    error = tf.reduce_sum(tf.square(q_target_sa - q_current_sa))
    pred_act_ph = tf.argmax(q_current, 1)


    Q = np.zeros((env.observation_space.n, env.action_space.n))
    for episode in range(1, num_episodes + 1):
        . . .
```

After setting up your algorithm and the loss function, define your optimizer:

/AtariBot/bot_4_q_network.py

```python
    . . .
    error = tf.reduce_sum(tf.square(q_target_sa - q_current_sa))
    pred_act_ph = tf.argmax(q_current, 1)


    # 3. Setup optimization
    trainer =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
    update_model = trainer.minimize(error)


    Q = np.zeros((env.observation_space.n, env.action_space.n))
    for episode in range(1, num_episodes + 1):
```

. . .

Next, set up the body of the game loop. To do this, pass data to the Tensorflow placeholders and Tensorflow's abstractions will handle the computation on the GPU, returning the result of the algorithm.

Start by deleting the old Q-table and logic. Specifically, delete the lines that define `Q` (right before the `for` loop), `noise` (in the `while` loop), `action`, `Qtarget`, and `Q[state, action]`. Rename `state` to `obs_t` and `state2` to `obs_tp1` to align with the Tensorflow placeholders you set previously. When finished, your `for` loop will match the following:

/AtariBot/bot_4_q_network.py

```python
    . . .

    # 3. Setup optimization
    trainer =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
    update_model = trainer.minimize(error)


    for episode in range(1, num_episodes + 1):
        obs_t = env.reset()
        episode_reward = 0
        while True:

            obs_tp1, reward, done, _ = env.step(action)


            episode_reward += reward
            obs_t = obs_tp1
            if done:
```

...

Directly above the `for` loop, add the following two highlighted lines. These lines initialize a Tensorflow session which in turn manages the resources needed to run operations on the GPU. The second line initializes all the variables in your computation graph; for example, initializing weights to 0 before updating them. Additionally, you will nest the `for` loop within the `with` statement, so indent the entire `for` loop by four spaces:

/AtariBot/bot_4_q_network.py

```
    . . .

    trainer =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
        update_model = trainer.minimize(error)

    with tf.Session() as session:
        session.run(tf.global_variables_initializer())

        for episode in range(1, num_episodes + 1):
            obs_t = env.reset()
            ...
```

Before the line reading `obs_tp1, reward, done, _ = env.step(action)`, insert the following lines to compute the `action`. This code evaluates the corresponding placeholder and replaces the action with a random action with some probability:

/AtariBot/bot_4_q_network.py

```
            . . .
        while True:
            # 4. Take step using best action or random action
            obs_t_oh = one_hot(obs_t, n_obs)
            action = session.run(pred_act_ph, feed_dict={obs_t_ph:
obs_t_oh})[0]
            if np.random.rand(1) < exploration_probability(episode):
                action = env.action_space.sample()
            . . .
```

After the line containing `env.step(action)`, insert the following to train the neural network in estimating your Q-value function:

/AtariBot/bot_4_q_network.py

```
            . . .
            obs_tp1, reward, done, _ = env.step(action)

            # 5. Train model
            obs_tp1_oh = one_hot(obs_tp1, n_obs)
            q_target_val = session.run(q_target, feed_dict={
                obs_tp1_ph: obs_tp1_oh
                })
            session.run(update_model, feed_dict={
                obs_t_ph: obs_t_oh,
                rew_ph: reward,
                q_target_ph: q_target_val,
                act_ph: action
```

```
            })
            episode_reward += reward
        . . .
```

Your final file will match this source code:

/AtariBot/bot_4_q_network.py

```python
"""
Bot 4 -- Use Q-learning network to train bot
"""

from typing import List
import gym
import numpy as np
import random
import tensorflow as tf

random.seed(0)
np.random.seed(0)
tf.set_random_seed(0)

num_episodes = 4000
discount_factor = 0.99
learning_rate = 0.15
report_interval = 500
exploration_probability = lambda episode: 50. / (episode + 10)
report = '100-ep Average: %.2f . Best 100-ep Average: %.2f . Average: \
%.2f ' \
```

```python
            '(Episode %d)'


def one_hot(i: int, n: int) -> np.array:
    """Implements one-hot encoding by selecting the ith standard basis
vector"""
    return np.identity(n)[i].reshape((1, -1))


def print_report(rewards: List, episode: int):
    """Print rewards report for current episode
    - Average for last 100 episodes
    - Best 100-episode average across all time
    - Average for all episodes across time
    """
    print(report % (
        np.mean(rewards[-100:]),
        max([np.mean(rewards[i:i+100]) for i in range(len(rewards) -
100)]),
        np.mean(rewards),
        episode))


def main():
    env = gym.make('FrozenLake-v0')  # create the game
    env.seed(0)  # make results reproducible
    rewards = []
```

```python
# 1. Setup placeholders
n_obs, n_actions = env.observation_space.n, env.action_space.n
obs_t_ph = tf.placeholder(shape=[1, n_obs], dtype=tf.float32)
obs_tp1_ph = tf.placeholder(shape=[1, n_obs], dtype=tf.float32)
act_ph = tf.placeholder(tf.int32, shape=())
rew_ph = tf.placeholder(shape=(), dtype=tf.float32)
q_target_ph = tf.placeholder(shape=[1, n_actions], dtype=tf.float32)


# 2. Setup computation graph
W = tf.Variable(tf.random_uniform([n_obs, n_actions], 0, 0.01))
q_current = tf.matmul(obs_t_ph, W)
q_target = tf.matmul(obs_tp1_ph, W)


q_target_max = tf.reduce_max(q_target_ph, axis=1)
q_target_sa = rew_ph + discount_factor * q_target_max
q_current_sa = q_current[0, act_ph]
error = tf.reduce_sum(tf.square(q_target_sa - q_current_sa))
pred_act_ph = tf.argmax(q_current, 1)


# 3. Setup optimization
trainer =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
update_model = trainer.minimize(error)


with tf.Session() as session:
    session.run(tf.global_variables_initializer())


    for episode in range(1, num_episodes + 1):
```

```python
        obs_t = env.reset()
        episode_reward = 0
        while True:

            # 4. Take step using best action or random action
            obs_t_oh = one_hot(obs_t, n_obs)
            action = session.run(pred_act_ph, feed_dict={obs_t_ph:
obs_t_oh})[0]
            if np.random.rand(1) < exploration_probability(episode):
                action = env.action_space.sample()
            obs_tp1, reward, done, _ = env.step(action)

            # 5. Train model
            obs_tp1_oh = one_hot(obs_tp1, n_obs)
            q_target_val = session.run(q_target, feed_dict={
                obs_tp1_ph: obs_tp1_oh
                })
            session.run(update_model, feed_dict={
                obs_t_ph: obs_t_oh,
                rew_ph: reward,
                q_target_ph: q_target_val,
                act_ph: action
            })
            episode_reward += reward
            obs_t = obs_tp1

            if done:
                rewards.append(episode_reward)
```

```
                    if episode % report_interval == 0:
                        print_report(rewards, episode)
                    break
        print_report(rewards, -1)


if __name__ == '__main__':
    main()
```

Save the file, exit your editor, and run the script:

```
python bot_4_q_network.py
```

Your output will end with the following, exactly:

Output

```
100-ep Average: 0.11 . Best 100-ep Average: 0.11 . Average: 0.05
(Episode 500)
100-ep Average: 0.41 . Best 100-ep Average: 0.54 . Average: 0.19
(Episode 1000)
100-ep Average: 0.56 . Best 100-ep Average: 0.73 . Average: 0.31
(Episode 1500)
100-ep Average: 0.57 . Best 100-ep Average: 0.73 . Average: 0.36
(Episode 2000)
100-ep Average: 0.65 . Best 100-ep Average: 0.73 . Average: 0.41
(Episode 2500)
100-ep Average: 0.65 . Best 100-ep Average: 0.73 . Average: 0.43
(Episode 3000)
100-ep Average: 0.69 . Best 100-ep Average: 0.73 . Average: 0.46
```

```
(Episode 3500)

100-ep Average: 0.77 . Best 100-ep Average: 0.79 . Average: 0.48

(Episode 4000)

100-ep Average: 0.77 . Best 100-ep Average: 0.79 . Average: 0.48

(Episode -1)
```

You've now trained your very first deep Q-learning agent. For a game as simple as FrozenLake, your deep Q-learning agent required 4000 episodes to train. Imagine if the game were far more complex. How many training samples would that require to train? As it turns out, the agent could require millions of samples. The number of samples required is referred to as sample complexity, a concept explored further in the next section.

## Understanding Bias-Variance Tradeoffs

Generally speaking, sample complexity is at odds with model complexity in machine learning:

1. Model complexity: One wants a sufficiently complex model to solve their problem. For example, a model as simple as a line is not sufficiently complex to predict a car's trajectory.
2. Sample complexity: One would like a model that does not require many samples. This could be because they have a limited access to labeled data, an insufficient amount of computing power, limited memory, etc.

Say we have two models, one simple and one extremely complex. For both models to attain the same performance, bias-variance tells us that

the extremely complex model will need exponentially more samples to `train`. Case in point: your neural network-based Q-learning agent required 4000 episodes to solve FrozenLake. Adding a second layer to the `neural` network agent quadruples the number of necessary training episodes. With increasingly complex neural networks, this divide only grows. To maintain the same error rate, increasing model complexity `increases` the sample complexity exponentially. Likewise, decreasing `sample` complexity decreases model complexity. Thus, we cannot maximize model complexity and minimize sample complexity to our heart's desire.

We can, however, leverage our knowledge of this tradeoff. For a visual interpretation of the mathematics behind the bias-variance decomposition, see [Understanding the Bias-Variance Tradeoff](#). At a high level, the bias-variance decomposition is a breakdown of "true error" into two components: bias and variance. We refer to "true error" as mean squared error (MSE), which is the expected difference between our predicted labels and the true labels. The following is a plot showing the change of "true error" as model complexity increases:
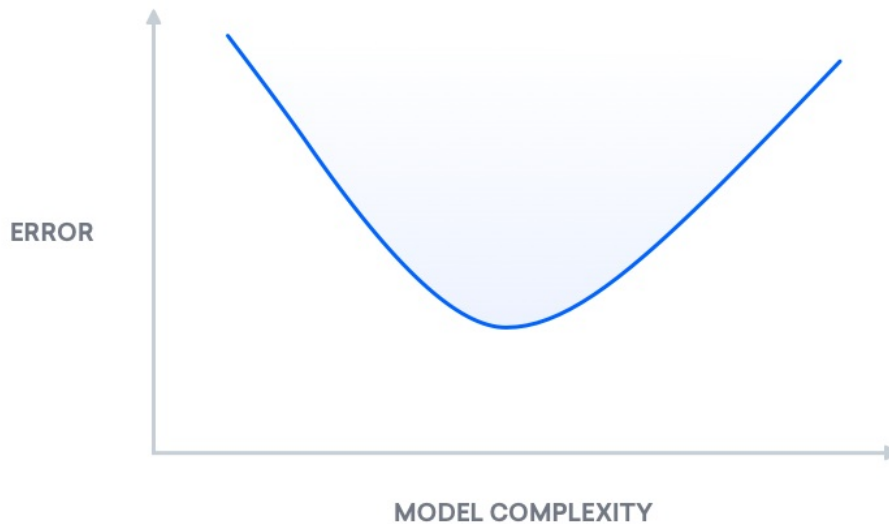
Mean Squared Error curve

## Step 5 — Building a Least Squares Agent for Frozen Lake

The least squares method, also known as linear regression, is a means of regression analysis used widely in the fields of mathematics and data science. In machine learning, it's often used to find the optimal linear model of two parameters or datasets.

In Step 4, you built a neural network to compute Q-values. Instead of a neural network, in this step you will use ridge regression, a variant of least squares, to compute this vector of Q-values. The hope is that with a model as uncomplicated as least squares, solving the game will require fewer training episodes.

Start by duplicating the script from Step 3:

```
cp bot_3_q_table.py bot_5_ls.py
```

Open the new file:

```
nano bot_5_ls.py
```

Again, update the comment at the top of the file describing what this script will do:

/AtariBot/bot_4_q_network.py

```
"""
Bot 5 -- Build least squares q-learning agent for FrozenLake
"""
. . .
```

Before the block of imports near the top of your file, add two more imports for type checking:

/AtariBot/bot_5_ls.py

```
. . .
from typing import Tuple
from typing import Callable
from typing import List
import gym
. . .
```

In your list of hyperparameters, add another hyperparameter, `w_lr`, to control the second Q-function's learning rate. Additionally, update the number of episodes to 5000 and the discount factor to `0.85`. By changing

both the `num_episodes` and `discount_factor` hyperparameters to larger values, the agent will be able to issue a stronger performance:

/AtariBot/bot_5_ls.py

```
. . .
num_episodes = 5000
discount_factor = 0.85
learning_rate = 0.9
w_lr = 0.5
report_interval = 500
. . .
```

Before your `print_report` function, add the following higher-order function. It returns a lambda — an anonymous function — that abstracts away the model:

/AtariBot/bot_5_ls.py

```
. . .
report_interval = 500
report = '100-ep Average: %.2f . Best 100-ep Average: %.2f . Average: %.2f ' \
 '(Episode %d)'

def makeQ(model: np.array) -> Callable[[np.array], np.array]:
"""Returns a Q-function, which takes state -> distribution over actions"""
return lambda X: X.dot(model)
```

```
def print_report(rewards: List, episode: int):
```

. . .

After `makeQ`, add another function, `initialize`, which initializes the model using normally-distributed values:

/AtariBot/bot_5_ls.py

. . .

```
def makeQ(model: np.array) -> Callable[[np.array], np.array]:
"""Returns a Q-function, which takes state -> distribution over
actions"""
return lambda X: X.dot(model)


def initialize(shape: Tuple):
"""Initialize model"""
W = np.random.normal(0.0, 0.1, shape)
Q = makeQ(W)
return W, Q


def print_report(rewards: List, episode: int):
```

. . .

After the `initialize` block, add a `train` method that computes the ridge regression closed-form solution, then weights the old model with the new one. It returns both the model and the abstracted Q-function:

/AtariBot/bot_5_ls.py

. . .

```
def initialize(shape: Tuple):

...

return W, Q


def train(X: np.array, y: np.array, W: np.array) -> Tuple[np.array,

Callable]:

"""Train the model, using solution to ridge regression"""

I = np.eye(X.shape[1])

newW = np.linalg.inv(X.T.dot(X) + 10e-4 _ I).dot(X.T.dot(y))

W = w_lr _ newW + (1 - w_lr) \* W

Q = makeQ(W)

return W, Q


def print_report(rewards: List, episode: int):

. . .
```

After `train`, add one last function, `one_hot`, to perform one-hot encoding for your states and actions:

/AtariBot/bot_5_ls.py

```
. . .

def train(X: np.array, y: np.array, W: np.array) -> Tuple[np.array,

Callable]:

...

return W, Q


def one_hot(i: int, n: int) -> np.array:

"""Implements one-hot encoding by selecting the ith standard basis
```

```
vector"""
return np.identity(n)[i]


def print_report(rewards: List, episode: int):

. . .
```

Following this, you will need to modify the training logic. In the previous script you wrote, the Q-table was updated every iteration. This script, however, will collect samples and labels every time step and train a new model every 10 steps. Additionally, instead of holding a Q-table or a neural network, it will use a least squares model to predict Q-values.

Go to the `main` function and replace the definition of the Q-table ($Q = $ `np.zeros(...)`) with the following:

/AtariBot/bot_5_ls.py

```
. . .
def main():
...
rewards = []

    n_obs, n_actions = env.observation_space.n, env.action_space.n
    W, Q = initialize((n_obs, n_actions))
    states, labels = [], []
    for episode in range(1, num_episodes + 1):
        . . .
```

Scroll down before the `for` loop. Directly below this, add the following lines which reset the `states` and `labels` lists if there is too much

information stored:

/AtariBot/bot_5_ls.py

```
. . .

def main():

...

for episode in range(1, num_episodes + 1):

if len(states) >= 10000:

states, labels = [], []

. . .
```

Modify the line directly after this one, which defines `state = env.reset()`, so that it becomes the following. This will one-hot encode the state immediately, as all of its usages will require a one-hot vector:

/AtariBot/bot_5_ls.py

```
. . .

for episode in range(1, num_episodes + 1):

if len(states) >= 10000:

states, labels = [], []

state = one_hot(env.reset(), n_obs)

. . .
```

Before the first line in your `while` main game loop, amend the list of `states`:

/AtariBot/bot_5_ls.py

```
. . .

for episode in range(1, num_episodes + 1):

...

episode_reward = 0

while True:

states.append(state)

noise = np.random.random((1, env.action_space.n)) / (episode\*\*2.)

. . .
```

Update the computation for `action`, decrease the probability of noise, and modify the Q-function evaluation:

/AtariBot/bot_5_ls.py

```
. . .

while True:

states.append(state)

noise = np.random.random((1, n*actions)) / episode

action = np.argmax(Q(state) + noise)

state2, reward, done, * = env.step(action)

. . .
```

Add a one-hot version of `state2` and amend the Q-function call in your definition for `Qtarget` as follows:

/AtariBot/bot_5_ls.py

```
. . .

while True:

...
```

```
state2, reward, done, \_ = env.step(action)

                state2 = one_hot(state2, n_obs)

                Qtarget = reward + discount_factor * np.max(Q(state2))

                . . .
```

Delete the line that updates `Q[state,action] = ...` and replace it with the following lines. This code takes the output of the current model and updates only the value in this output that corresponds to the current action taken. As a result, Q-values for the other actions don't incur loss:

/AtariBot/bot_5_ls.py

```
. . .

state2 = one_hot(state2, n_obs)

Qtarget = reward + discount_factor _ np.max(Q(state2))

label = Q(state)

label[action] = (1 - learning_rate) _ label[action] + learning_rate \*
Qtarget

labels.append(label)

                episode_reward += reward

                . . .
```

Right after `state = state2`, add a periodic update to the model. This trains your model every 10 time steps:

/AtariBot/bot_5_ls.py

```
. . .
```

```
            state = state2
            if len(states) % 10 == 0:
                W, Q = train(np.array(states), np.array(labels), W)
        if done:

. . .
```

Ensure that your code matches the following:

/AtariBot_5_ls.py

```python
"""
Bot 5 -- Build least squares q-learning agent for FrozenLake
"""

from typing import Tuple
from typing import Callable
from typing import List
import gym
import numpy as np
import random


random.seed(0)  # make results reproducible
np.random.seed(0)  # make results reproducible


num_episodes = 5000
discount_factor = 0.85
learning_rate = 0.9
w_lr = 0.5
report_interval = 500
```

```python
report = '100-ep Average: %.2f . Best 100-ep Average: %.2f . Average: 
%.2f ' \
        '(Episode %d)'


def makeQ(model: np.array) -> Callable[[np.array], np.array]:
    """Returns a Q-function, which takes state -> distribution over 
actions"""
    return lambda X: X.dot(model)


def initialize(shape: Tuple):
    """Initialize model"""
    W = np.random.normal(0.0, 0.1, shape)
    Q = makeQ(W)
    return W, Q


def train(X: np.array, y: np.array, W: np.array) -> Tuple[np.array, 
Callable]:
    """Train the model, using solution to ridge regression"""
    I = np.eye(X.shape[1])
    newW = np.linalg.inv(X.T.dot(X) + 10e-4 * I).dot(X.T.dot(y))
    W = w_lr * newW + (1 - w_lr) * W
    Q = makeQ(W)
    return W, Q
```

```python
def one_hot(i: int, n: int) -> np.array:
    """Implements one-hot encoding by selecting the ith standard basis
vector"""
    return np.identity(n)[i]



def print_report(rewards: List, episode: int):
    """Print rewards report for current episode
    - Average for last 100 episodes
    - Best 100-episode average across all time
    - Average for all episodes across time
    """
    print(report % (
        np.mean(rewards[-100:]),
        max([np.mean(rewards[i:i+100]) for i in range(len(rewards) -
100)]),
        np.mean(rewards),
        episode))



def main():
    env = gym.make('FrozenLake-v0')  # create the game
    env.seed(0)  # make results reproducible
    rewards = []

    n_obs, n_actions = env.observation_space.n, env.action_space.n
    W, Q = initialize((n_obs, n_actions))
    states, labels = [], []
```

```python
for episode in range(1, num_episodes + 1):

    if len(states) >= 10000:

        states, labels = [], []

    state = one_hot(env.reset(), n_obs)

    episode_reward = 0

    while True:

        states.append(state)

        noise = np.random.random((1, n_actions)) / episode

        action = np.argmax(Q(state) + noise)

        state2, reward, done, _ = env.step(action)


        state2 = one_hot(state2, n_obs)

        Qtarget = reward + discount_factor * np.max(Q(state2))

        label = Q(state)

        label[action] = (1 - learning_rate) * label[action] + \
            learning_rate * Qtarget

        labels.append(label)


        episode_reward += reward

        state = state2

        if len(states) % 10 == 0:

            W, Q = train(np.array(states), np.array(labels), W)

        if done:

            rewards.append(episode_reward)

            if episode % report_interval == 0:

                print_report(rewards, episode)

            break
print_report(rewards, -1)
```

```
if __name__ == '__main__':

    main()
```

Then, save the file, exit the editor, and run the script:

```
python bot_5_ls.py
```

This will output the following:

Output

```
100-ep Average: 0.17 . Best 100-ep Average: 0.17 . Average: 0.09

(Episode 500)

100-ep Average: 0.11 . Best 100-ep Average: 0.24 . Average: 0.10

(Episode 1000)

100-ep Average: 0.08 . Best 100-ep Average: 0.24 . Average: 0.10

(Episode 1500)

100-ep Average: 0.24 . Best 100-ep Average: 0.25 . Average: 0.11

(Episode 2000)

100-ep Average: 0.32 . Best 100-ep Average: 0.31 . Average: 0.14

(Episode 2500)

100-ep Average: 0.35 . Best 100-ep Average: 0.38 . Average: 0.16

(Episode 3000)

100-ep Average: 0.59 . Best 100-ep Average: 0.62 . Average: 0.22

(Episode 3500)

100-ep Average: 0.66 . Best 100-ep Average: 0.66 . Average: 0.26

(Episode 4000)

100-ep Average: 0.60 . Best 100-ep Average: 0.72 . Average: 0.30
```

```
(Episode 4500)

100-ep Average: 0.75 . Best 100-ep Average: 0.82 . Average: 0.34

(Episode 5000)

100-ep Average: 0.75 . Best 100-ep Average: 0.82 . Average: 0.34

(Episode -1)
```

Recall that, according to the [Gym FrozenLake page](#), "solving" the game means attaining a 100-episode average of 0.78. Here the agent acheives an average of 0.82, meaning it was able to solve the game in 5000 episodes. Although this does not solve the game in fewer episodes, this basic least squares method is still able to solve a simple game with roughly the same number of training episodes. Although your neural networks may grow in complexity, you've shown that simple models are sufficient for FrozenLake.

With that, you have explored three Q-learning agents: one using a Q-table, another using a neural network, and a third using least squares. Next, you will build a deep reinforcement learning agent for a more complex game: Space Invaders.

## Step 6 — Creating a Deep Q-learning Agent for Space Invaders

Say you tuned the previous Q-learning algorithm's model complexity and sample complexity perfectly, regardless of whether you picked a neural network or least squares method. As it turns out, this unintelligent Q-learning agent still performs poorly on more complex games, even with an especially high number of training episodes. This section will cover two techniques that can improve performance, then you will test an agent that was trained using these techniques.

The first general-purpose agent able to continually adapt its behavior without any human intervention was developed by the researchers at DeepMind, who also trained their agent to play a variety of Atari games. [DeepMind's original deep Q-learning (DQN) paper](#) recognized two important issues:

1. Correlated states: Take the state of our game at time 0, which we will call s0. Say we update Q(s0), according to the rules we derived previously. Now, take the state at time 1, which we call s1, and update Q(s1) according to the same rules. Note that the game's state at time 0 is very similar to its state at time 1. In Space Invaders, for example, the aliens may have moved by one pixel each. Said more succinctly, s0 and s1 are very similar. Likewise, we also expect Q(s0) and Q(s1) to be very similar, so updating one affects the other. This leads to fluctuating Q values, as an update to Q(s0) may in fact counter the update to Q(s1). More formally, s0 and s1 are correlated. Since the Q-function is deterministic, Q(s1) is correlated with Q(s0).

2. Q-function instability: Recall that the Q function is both the model we train and the source of our labels. Say that our labels are randomly-selected values that truly represent a distribution, L. Every time we update Q, we change L, meaning that our model is trying to learn a moving target. This is an issue, as the models we use assume a fixed distribution.

To combat correlated states and an unstable Q-function:

1. One could keep a list of states called a replay buffer. Each time step, you add the game state that you observe to this replay buffer. You

also randomly sample a subset of states from this list, and train on those states.

2. The team at DeepMind duplicated Q(s, a). One is called Q_current(s, a), which is the Q-function you update. You need another Q-function for successor states, Q_target(s', a'), which you won't update. Recall Q_target(s', a') is used to generate your labels. By separating Q_current from Q_target and fixing the latter, you fix the distribution your labels are sampled from. Then, your deep learning model can spend a short period learning this distribution. After a period of time, you then re-duplicate Q_current for a new Q_target.

You won't implement these yourself, but you will load pretrained models that trained with these solutions. To do this, create a new directory where you will store these models' parameters:

```
mkdir models
```

Then use `wget` to download a pretrained Space Invaders model's parameters:

```
wget http://models.tensorpack.com/OpenAIGym/SpaceInvaders-v0.tfmodel -P models
```

Next, download a Python script that specifies the model associated with the parameters you just downloaded. Note that this pretrained model has two constraints on the input that are necessary to keep in mind:

- The states must be downsampled, or reduced in size, to 84 x 84.
- The input consists of four states, stacked.

We will address these constraints in more detail later on. For now, download the script by typing:

```
wget https://github.com/alvinwan/bots-for-atari-
games/raw/master/src/bot_6_a3c.py
```

You will now run this pretrained Space Invaders agent to see how it performs. Unlike the past few bots we've used, you will write this script from scratch.

Create a new script file:

```
nano bot_6_dqn.py
```

Begin this script by adding a header comment, importing the necessary utilities, and beginning the main game loop:

/AtariBot/bot_6_dqn.py

```
"""
Bot 6 - Fully featured deep q-learning network.
"""

import cv2
import gym
import numpy as np
import random
```

```
import tensorflow as tf

from bot_6_a3c import a3c_model


def main():


if **name** == '**main**':

main()
```

Directly after your imports, set random seeds to make your results reproducible. Also, define a hyperparameter `num_episodes` which will tell the script how many episodes to run the agent for:

/AtariBot/bot_6_dqn.py

```
. . .

import tensorflow as tf

from bot_6_a3c import a3c_model

random.seed(0) # make results reproducible

tf.set_random_seed(0)


num_episodes = 10


def main():

. . .
```

Two lines after declaring `num_episodes`, define a `downsample` function that downsamples all images to a size of 84 x 84. You will downsample all images before passing them into the pretrained neural network, as the pretrained model was trained on 84 x 84 images:

/AtariBot/bot_6_dqn.py

```
. . .

num_episodes = 10


def downsample(state):
    return cv2.resize(state, (84, 84), interpolation=cv2.INTER_LINEAR)[None]


def main():
. . .
```

Create the game environment at the start of your `main` function and seed the environment so that the results are reproducible:

/AtariBot/bot_6_dqn.py

```
. . .
def main():
    env = gym.make('SpaceInvaders-v0') # create the game
    env.seed(0) # make results reproducible
. . .
```

Directly after the environment seed, initialize an empty list to hold the rewards:

/AtariBot/bot_6_dqn.py

```
. . .
def main():
    env = gym.make('SpaceInvaders-v0') # create the game
    env.seed(0) # make results reproducible
```

```
rewards = []
```

. . .

Initialize the pretrained model with the pretrained model parameters that you downloaded at the beginning of this step:

/AtariBot/bot_6_dqn.py

```
. . .
def main():
env = gym.make('SpaceInvaders-v0') # create the game
env.seed(0) # make results reproducible
rewards = []
model = a3c_model(load='models/SpaceInvaders-v0.tfmodel')
. . .
```

Next, add some lines telling the script to iterate for `num_episodes` times to compute average performance and initialize each episode's reward to 0. Additionally, add a line to reset the environment (`env.reset()`), collecting the new initial state in the process, downsample this initial state with `downsample()`, and start the game loop using a `while` loop:

/AtariBot/bot_6_dqn.py

```
. . .
def main():
env = gym.make('SpaceInvaders-v0') # create the game
env.seed(0) # make results reproducible
rewards = []
```

```
model = a3c*model(load='models/SpaceInvaders-v0.tfmodel')

for * in range(num_episodes):

episode_reward = 0

states = [downsample(env.reset())]

while True:

. . .
```

Instead of accepting one state at a time, the new neural network accepts four states at a time. As a result, you must wait until the list of `states` contains at least four states before applying the pretrained model. Add the following lines below the line reading `while True:`. These tell the agent to take a random action if there are fewer than four states or to concatenate the states and pass it to the pretrained model if there are at least four:

/AtariBot/bot_6_dqn.py

```
        . . .

    while True:

        if len(states) < 4:

            action = env.action_space.sample()

        else:

            frames = np.concatenate(states[-4:], axis=3)

            action = np.argmax(model([frames]))

            . . .
```

Then take an action and update the relevant data. Add a downsampled version of the observed state, and update the reward for this episode:

/AtariBot/bot_6_dqn.py

```
        . . .

        while True:

            ...

                action = np.argmax(model([frames]))
            state, reward, done, _ = env.step(action)
            states.append(downsample(state))
            episode_reward += reward

            . . .
```

Next, add the following lines which check whether the episode is `done` and, if it is, print the episode's total reward and amend the list of all results and break the `while` loop early:

/AtariBot/bot_6_dqn.py

```
        . . .

        while True:

            ...

            episode_reward += reward

            if done:

                print('Reward: %d' % episode_reward)

                rewards.append(episode_reward)

                break

                . . .
```

Outside of the `while` and `for` loops, print the average reward. Place this at the end of your `main` function:

/AtariBot/bot_6_dqn.py

```python
def main():
...
break
print('Average reward: %.2f' % (sum(rewards) / len(rewards)))
```

Check that your file matches the following:

/AtariBot/bot_6_dqn.py

```python
"""
Bot 6 - Fully featured deep q-learning network.
"""

import cv2
import gym
import numpy as np
import random
import tensorflow as tf
from bot_6_a3c import a3c_model


random.seed(0)  # make results reproducible
tf.set_random_seed(0)


num_episodes = 10


def downsample(state):
    return cv2.resize(state, (84, 84), interpolation=cv2.INTER_LINEAR)
```

[None]

```python
def main():
    env = gym.make('SpaceInvaders-v0')  # create the game
    env.seed(0)  # make results reproducible
    rewards = []


    model = a3c_model(load='models/SpaceInvaders-v0.tfmodel')
    for _ in range(num_episodes):
        episode_reward = 0
        states = [downsample(env.reset())]
        while True:
            if len(states) < 4:
                action = env.action_space.sample()
            else:
                frames = np.concatenate(states[-4:], axis=3)
                action = np.argmax(model([frames]))
            state, reward, done, _ = env.step(action)
            states.append(downsample(state))
            episode_reward += reward
            if done:
                print('Reward: %d' % episode_reward)
                rewards.append(episode_reward)
                break
    print('Average reward: %.2f' % (sum(rewards) / len(rewards)))
```

```
if __name__ == '__main__':

    main()
```

Save the file and exit your editor. Then, run the script:

```
python bot_6_dqn.py
```

Your output will end with the following:

Output
```
. . .
Reward: 1230
Reward: 4510
Reward: 1860
Reward: 2555
Reward: 515
Reward: 1830
Reward: 4100
Reward: 4350
Reward: 1705
Reward: 4905
Average reward: 2756.00
```

Compare this to the result from the first script, where you ran a random agent for Space Invaders. The average reward in that case was only about 150, meaning this result is over twenty times better. However, you only ran your code for three episodes, as it's fairly slow, and the average of three episodes is not a reliable metric. Running this over 10

episodes, the average is 2756; over 100 episodes, the average is around 2500. Only with these averages can you comfortably conclude that your agent is indeed performing an order of magnitude better, and that you now have an agent that plays Space Invaders reasonably well.

However, recall the issue that was raised in the previous section regarding sample complexity. As it turns out, this Space Invaders agent takes millions of samples to train. In fact, this agent required 24 hours on four Titan X GPUs to train up to this current level; in other words, it took a significant amount of compute to train it adequately. Can you train a similarly high-performing agent with far fewer samples? The previous steps should arm you with enough knowledge to begin exploring this question. Using far simpler models and per bias-variance tradeoffs, it may be possible.

## Conclusion

In this tutorial, you built several bots for games and explored a fundamental concept in machine learning called bias-variance. A natural next question is: Can you build bots for more complex games, such as StarCraft 2? As it turns out, this is a pending research question, supplemented with open-source tools from collaborators across Google, DeepMind, and Blizzard. If these are problems that interest you, see [open calls for research at OpenAI](#), for current problems.

The main takeaway from this tutorial is the bias-variance tradeoff. It is up to the machine learning practitioner to consider the effects of model complexity. Whereas it is possible to leverage highly complex models and layer on excessive amounts of compute, samples, and time, reduced model complexity could significantly reduce the resources required.