Hyperiondev

TASK

Neural Networks II: Training Neural Networks

Visit our website

Introduction

WELCOME TO THE SECOND NEURAL NETWORKS TASK!

In the previous task, you learned how a simple neural network arrives at predictions based on training data. In the lesson, we will learn how to improve those predictions by training the neural network.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with a code reviewer. You can also schedule a call or get support via email.

Our expert code reviewers are happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

LOSS FUNCTION

You should now have a basic understanding of neural networks and be able to implement a simple neural network using Python to get a final predicted output. However, we still need a way to evaluate the accuracy of our predictions. In other words, we need to check how far off our predictions are from the true labels; how well we modelled our training data. The loss function allows us to do exactly that.

A loss function is a method of evaluating how well your neural network models your dataset. If your predictions are totally off, your loss function will output a high number. If they're pretty good, it will output a low number.

There are many available loss functions, however, we will use one of the most basic and easy to understand loss functions: the Mean Squared Error (MSE). To calculate MSE, you take the difference between your predictions and the actual value, square it, and average it out across the whole dataset.

The mean squared error is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2$$

Let's take a closer look at this function:

- **n** is the number of samples in a dataset
- **y** represents the label being predicted
- **y_true** is the true label (the "correct answer")
- **y_pred** is the predicted label that the network outputs

TRAINING A NEURAL NETWORK

The goal of training a neural network is to find the set of weights and biases that minimises the loss function. When we start off with our neural network we initialise our weights and biases randomly. Outputs that are computed based on these random weights and biases will be far off from our gold standard: the outcome of the loss function will be a high number and the accuracy will be low. The process of training lets us update the weights and biases to output better predictions. The ultimate goal is to find weights and bias values associated with the lowest possible loss. But how do we know how to change the weights, given a high loss in the output of the network? This is where the concept of gradient descent comes in.

Gradient Descent is an optimisation algorithm that lets us tweak parameters iteratively to minimise a given function to its local minimum. Take a look at the graph below:

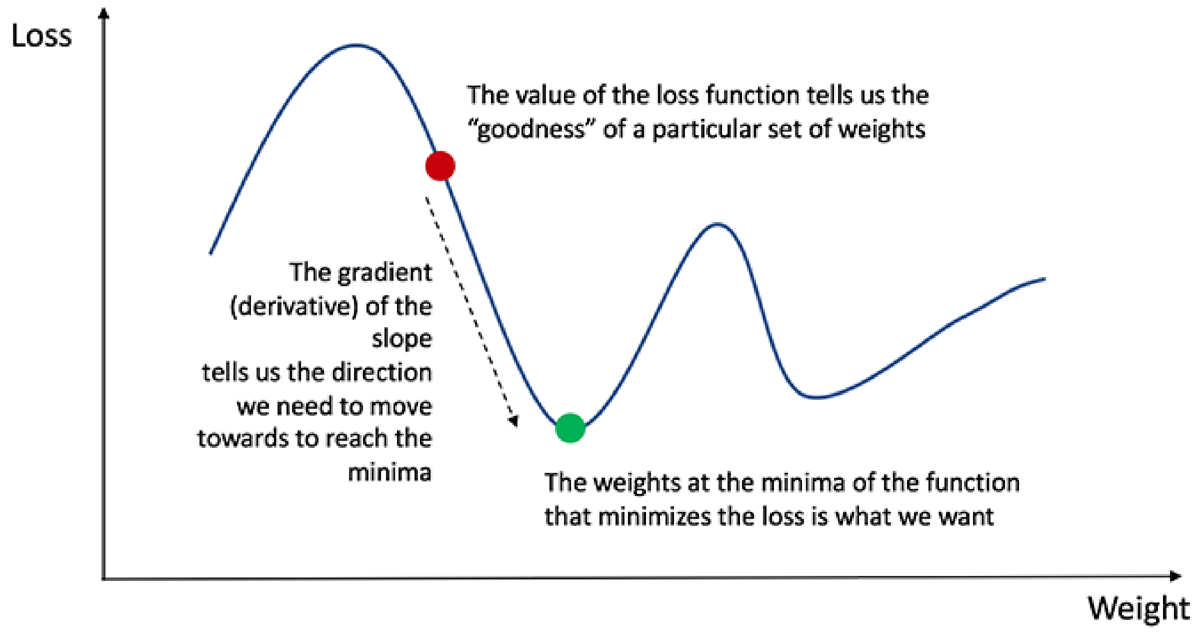

Loss
The value of the loss function tells us the "goodness" of a particular set of weights
The gradient (derivative) of the slope tells us the direction we need to move towards to reach the minima
The weights at the minima of the function that minimizes the loss is what we want
Weight

Image source: (Loy, 2018)

You may notice that at the red point in the graph the slope of the line is quite steep, while at the green point the gradient of the line is close to zero (you can see this more clearly if you draw a tangent to the line at each of these points). The slope, or gradient, of a function can be mathematically defined as the derivative of the function. Hence, we can find the local minimum by minimising the derivative (gradient) of the function. This would be done through a series of steps, as the red point moves closer and closer to the green point — this is gradient descent. On each iteration, the weights of the network are thus adjusted to minimise the derivative.

BACKPROPAGATION

A neural network propagates input data forward through its parameters towards the output, produces a prediction and the loss of the network is calculated. The weights of the network now need to be adjusted to minimise the error.

Recall that the loss at each point on the graph is calculated using only the predicted labels and the expected labels and that, in turn, the predicted labels depend on the outputs of each of the nodes in the network, which in turn depend on the weights of the inputs and the bias. This means that we can't compute the derivative of the loss function directly from the weights and biases, and need to employ a trick from calculus called the "chain rule" to allow us to calculate it. We will not be discussing the mathematics of backpropagation in this course, but you can read more about it **here**.

The backpropagation algorithm then moves through the network, adjusting the weights and biases so as to minimise the overall loss of the network, starting at the output layer and working backwards towards the inputs. This means that each time we pass training data into the network, it moves forwards through the network to make a prediction, and the error produced by this prediction then travels back through the network and is used to adjust the neural network's weights in the direction of lower error.

USING KERAS TO TRAIN YOUR FIRST NEURAL NETWORK

As you can imagine, building a neural network from scratch can be quite daunting. Luckily, however, we have machine learning frameworks such as Google's TensorFlow to ease the process of acquiring data, training models, serving predictions, and refining future results (documentation **here**).

Keras is TensorFlow's high-level Python API for building and training deep learning models. It was developed to make implementing deep learning models as fast and as easy as possible for research and development.

To download TensorFlow you need to run the following command in your terminal or command prompt:

```
pip install tensorflow
```

If you are having trouble, you can read more about installing TensorFlow **here**.

In order to get an understanding of how to use Keras, we will be using it to train a neural network model to classify images of clothing like dresses and shirts. The dataset we will be using is called the **fashion MNIST dataset**.

TensorFlow provides a tutorial on how to do this task, which is adapted in this section. The original tutorial can be found **here**. We will be going through the code step by step. Feel free to follow along by copying and pasting the code snippets into a Jupyter notebook.

Before you begin you need to import TensorFlow as well as Keras, along with the helper libraries for using arrays (numpy) and plotting graphs (matplotlib.pyplot).

```
# TensorFlow and Keras
import tensorflow as tf
from tensorflow import keras

# Helper libraries

import numpy as np
import matplotlib.pyplot as plt
```

The fashion MNIST dataset contains 70 000 grayscale images in 10 categories. The images show individual articles of clothing at low resolution (28 by 28 pixels). The dataset can be directly imported and loaded from TensorFlow as follows:

```
fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()
```

Loading the dataset returns four NumPy arrays:

- The `train_images` and `train_labels` arrays are the training set (the data the model uses to learn). There are 60 000 in this dataset.
- The `test_images` and `test_labels` arrays are the test set (the data the model is evaluated against). There are 10 000 in this dataset.

The images are 28x28 NumPy arrays, with pixel values ranging between 0 and 255. The labels are an array of integers, ranging from 0 to 9. These correspond to the class of clothing the image represents:

Label	Class
0	T-shirt/top
1	Trouser

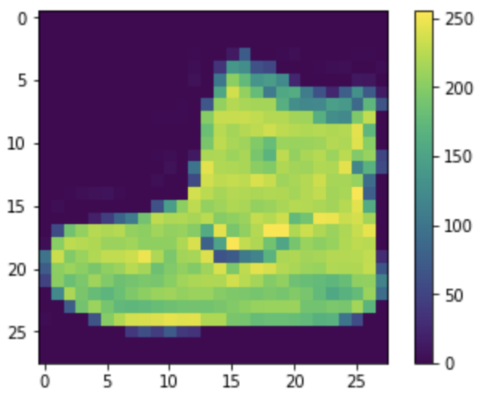
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Each image is mapped to a single label. Since the class names are not included with the dataset, we should store them to use later when plotting the images:

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

The data must be preprocessed before training the network. If you inspect the first image in the training set, you will see that the pixel values fall in the range of 0 to 255:

```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



Scale these values to a range of 0 to 1 before feeding them to the neural network model. To do so, divide the values by 255. It's important that the training set and the testing set be preprocessed in the same way:

```
train_images = train_images / 255.0

test_images = test_images / 255.0
```

We can now verify that the data is in the correct format by displaying the first 25 images from the training set, along with the class name below each image.

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

Now that the data has been loaded and verified, we can build the neural network, which requires configuring the layers of the model and then compiling it.

The basic building block of a neural network is the layer. Layers extract representations from the data fed into them. Hopefully, these representations are meaningful for the problem at hand.

Most deep learning consists of chaining together simple layers. Most layers, such as **`tf.keras.layers.Dense`**, have parameters that are learned during training.

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation = 'softmax')
])
```

The first layer in this network, **`tf.keras.layers.Flatten`**, transforms the format of the images from a two-dimensional array (of 28 by 28 pixels) to a one-dimensional array (of 28 * 28 = 784 pixels). Think of this layer as unstacking rows of pixels in the image and lining them up. This layer has no parameters to learn; it only reformats the data.

After the pixels are flattened, the network consists of a sequence of two **`tf.keras.layers.Dense`** layers. These are densely connected, or fully-connected, neural layers. The first Dense layer has 128 nodes (or neurons). The second (and last) layer returns a logits array with a length of 10. Each node contains a score that indicates the current image belongs to one of the 10 classes. The softmax activation is used, which indicates that the output will be in the range [0:1], indicating the probability of the prediction belonging to each class.

We can now compile the model:

```
model.compile(optimiser='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Before the model is ready for training, it needs a few more settings. These are added during the model's compile step:

- Loss function — This measures how accurate the model is during training. You want to minimise this function to "steer" the model in the right direction.
- Optimiser — This is how the model is updated based on the data it sees and its loss function.
- Metrics — Used to monitor the training and testing steps. The following example uses accuracy, the fraction of the images that are correctly classified.

The model is now ready to be trained. Training the neural network model requires the following steps:

- Feed the training data to the model. In this example, the training data is in the **`train_images`** and **`train_labels`** arrays.
- The model learns to associate images and labels.
- You ask the model to make predictions about a test set — in this example, the **`test_images`** array.
- Verify that the predictions match the labels from the **`test_labels`** array.

To start training, call the **`model.fit`** method — so-called because it "fits" the model to the training data. This method passes the training data through the network and updates the weights and biases through backpropagation.

```
model.fit(train_images, train_labels, epochs=10)
```

This will output something like the following:

```
Epoch 1/10
60000/60000 [==============================] - 2s 40us/sample - loss: 0.5057 -
acc: 0.8235
Epoch 2/10
60000/60000 [==============================] - 3s 48us/sample - loss: 0.3779 -
acc: 0.8650
Epoch 3/10
60000/60000 [==============================] - 2s 37us/sample - loss: 0.3368 -
acc: 0.8787
Epoch 4/10
60000/60000 [==============================] - 2s 38us/sample - loss: 0.3159 -
acc: 0.8838
Epoch 5/10
60000/60000 [==============================] - 2s 38us/sample - loss: 0.2964 -
acc: 0.8901
Epoch 6/10
60000/60000 [==============================] - 2s 38us/sample - loss: 0.2814 -
acc: 0.8961
Epoch 7/10
60000/60000 [==============================] - 2s 38us/sample - loss: 0.2667 -
acc: 0.9028
Epoch 8/10
60000/60000 [==============================] - 2s 38us/sample - loss: 0.2566 -
acc: 0.9041
Epoch 9/10
60000/60000 [==============================] - 2s 38us/sample - loss: 0.2482 -
acc: 0.9071
Epoch 10/10
60000/60000 [==============================] - 2s 37us/sample - loss: 0.2409 -
acc: 0.9104
```

As the model trains, the loss and accuracy metrics are displayed. This model reaches an accuracy of about 0.91 (or 91%) on the training data.

Now that the model has been trained, it can be used to make predictions for the test data.

```
predictions = model.predict(test_images)
```

Here, the model has predicted the label for each image in the testing set. Let's take a look at the first prediction:

```
predictions[0]
```

Which outputs:

```
array([9.6798196e-09, 3.9551045e-12, 1.1966002e-09, 6.4597640e-11,
       6.9501125e-09, 1.3103469e-04, 1.2785731e-07, 2.0190427e-01,
       1.3739854e-09, 7.9796457e-01], dtype=float32)
```

A prediction is an array of 10 numbers. They represent the model's "confidence" that the image corresponds to each of the 10 different articles of clothing. You can see which label has the highest confidence value:

```
np.argmax(predictions[0])
```

```
9
```

So, the model is most confident that this image is an ankle boot, or class_names[9]. Examining the test label shows that this classification is correct:

```
test_labels[0]
```

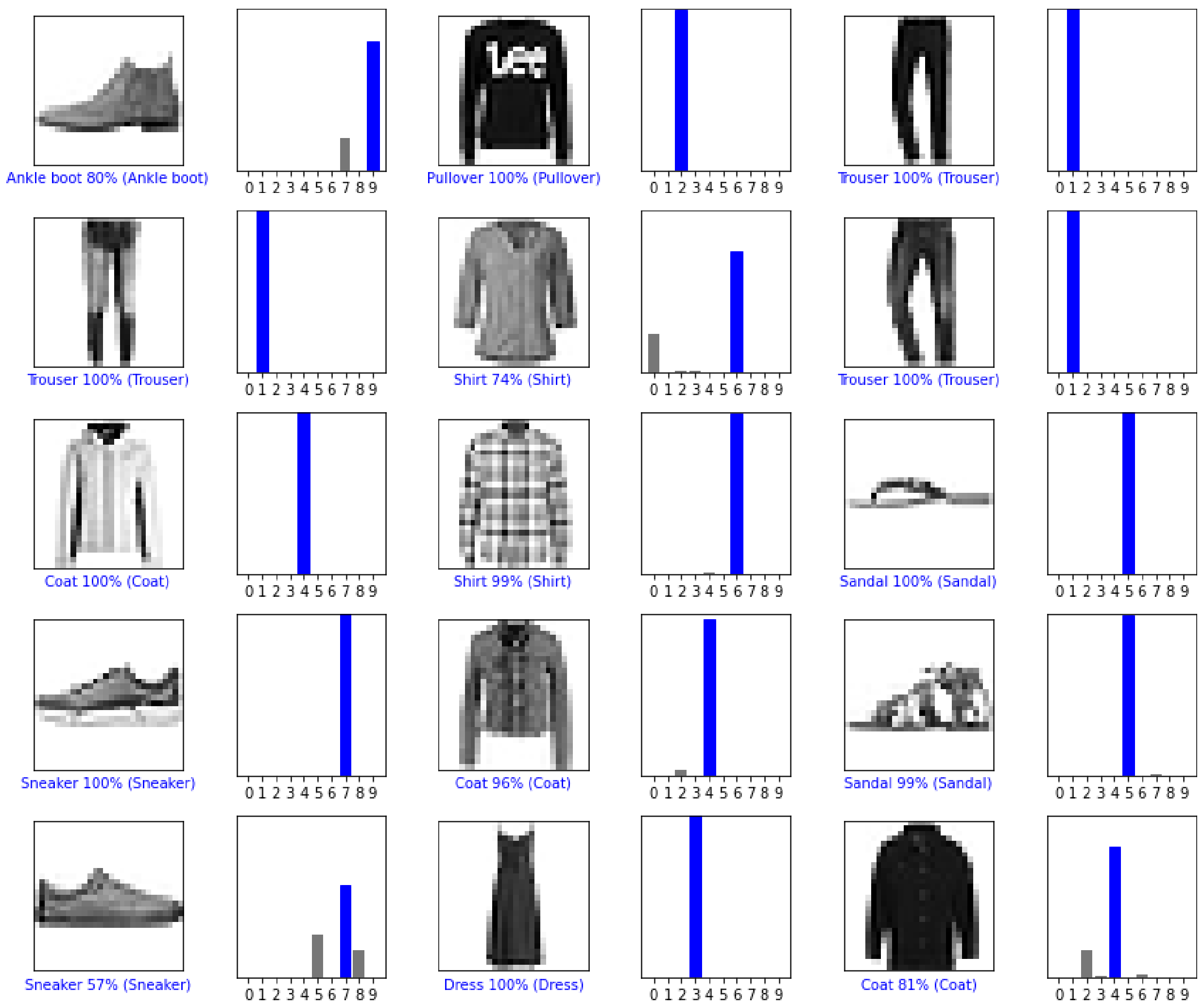
```
9
```

Graph this to look at the full set of 10 class predictions.

```
def plot_image(i, predictions_array, true_label, img):
  predictions_array, true_label, img = predictions_array, true_label[i],
img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  predictions_array, true_label = predictions_array, true_label[i]
  plt.grid(False)
  plt.xticks(range(10))
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
```

Using these methods, we can now verify predictions by displaying a few of the images with their predictions:

```
# Plot the first X test images, their predicted labels, and the true labels.
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions[i], test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
```


Ankle boot 80% (Ankle boot)
0 1 2 3 4 5 6 7 8 9
Pullover 100% (Pullover)
0 1 2 3 4 5 6 7 8 9
Trouser 100% (Trouser)
0 1 2 3 4 5 6 7 8 9
Trouser 100% (Trouser)
0 1 2 3 4 5 6 7 8 9
Shirt 74% (Shirt)
0 1 2 3 4 5 6 7 8 9
Trouser 100% (Trouser)
0 1 2 3 4 5 6 7 8 9
Coat 100% (Coat)
0 1 2 3 4 5 6 7 8 9
Shirt 99% (Shirt)
0 1 2 3 4 5 6 7 8 9
Sandal 100% (Sandal)
0 1 2 3 4 5 6 7 8 9
Sneaker 100% (Sneaker)
0 1 2 3 4 5 6 7 8 9
Coat 96% (Coat)
0 1 2 3 4 5 6 7 8 9
Sandal 99% (Sandal)
0 1 2 3 4 5 6 7 8 9
Sneaker 57% (Sneaker)
0 1 2 3 4 5 6 7 8 9
Dress 100% (Dress)
0 1 2 3 4 5 6 7 8 9
Coat 81% (Coat)
0 1 2 3 4 5 6 7 8 9

Instructions

The exercises in these tasks require users to run Jupyter Notebooks through the Anaconda environment. Please make sure that you download the Anaconda Distribution from **https://www.anaconda.com/distribution/**.

Compulsory Task 1

Launch Jupyter Notebook via your Anaconda environment and upload the file named **Backprop.ipynb** from your task folder and follow the instructions.
Submit your Task in **.ipynb** extension format.

Completed the task(s)?

Ask an expert to review your work!

Review work

Things to look out for:

1. Make sure that you have installed and set up all programs correctly. You have set up **Dropbox** correctly if you are reading this, but **Python or Notepad++** may not be installed correctly.
2. If you are not using Windows, please ask your mentor for alternative instructions.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

References:

Loy, J. (2018). How to build your own Neural Network from scratch in Python. Retrieved 28 August 2020, from **https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6**