Neural Networks

WITH TENSORFLOW

Yasaman Mirmohammad | Datamining | December 2018

Task 1 : Drinks

Here we will build a **classifier** that recognizes the drinks based on 13 attributes

Because our data is labeled,our task is a **Supervised learning** problem.

Essentially, what we want to do is use our input data (the 178 unclassified bottles), put it through a neural network, and then get the right label for each one as the output.

We will train our algorithm to get better and better at predicting (y-hat) which bottle belongs to which label.

- input layer (x) consists of 178 neurons.
- A1, the first layer, consists of 8 neurons.
- A2, the second layer, consists of 5 neurons.
- A3, the third and output layer, consists of 3 neurons.

Step1: preprocessing

Step2:initialization

we have to initialize the weights. Because we don't have values to use for the weights yet, we use random values between 0 and 1.

In Python, the random.seed function generates "random numbers:

Step3:forward propagation

* “making steps” forward and comparing those results with the real values to get the difference between your output and what it should be. We will basically see how the NN is doing and find the errors.

After we have initialized the weights with a pseudo-random number, we take a linear step forward. We calculate this by taking our input A0 times the **dot product** of the random initialized weights plus a **bias.**

Our first step is Z1:

$$Z1=A0.w1 + b$$

we take our z1 and pass it through the first **activation function.** (Activation functions are very important in neural networks. Essentially, they convert an input signal to an output signal—this is why they are also known as *Transfer functions.* They introduce **non-linear properties** to our functions by converting the linear input to a non-linear output, making it possible to represent more complex functions.)

I used **tanh** activation function for the two hidden layers—A1 and A2—which gives an output value between -1 and 1.

Since this is a **multi-class classification problem** (we have 3 output labels), we will use the **softmax** function for the output layer—A3—because

this will compute the probabilities for the classes by spitting out a value between 0 and 1.

By passing z1 through the activation function, we have created our first hidden layer—A1—which can be used as input for the computation of the next linear step, z2.

A1=tanh(z1)

Step4:Backward Propagation

After we forward propagate through our NN, we backward propagate our error gradient to update our weight parameters. We know our error, and want to minimize it as much as possible.

We do this by taking the **derivative of the error function,** with respect to the weights (W) of our NN, using **gradient descent.**

Next we calculate the **slope of the loss function** with respect to our weights and biases. Because this is a 3 layer NN, we will iterate this process for z3,2,1 + W3,2,1 and b3,2,1. Propagating backwards from the output to the input layer:

d3=A3-y

dw3=1/m*A2.dz3

db3=1/m*∑dz3

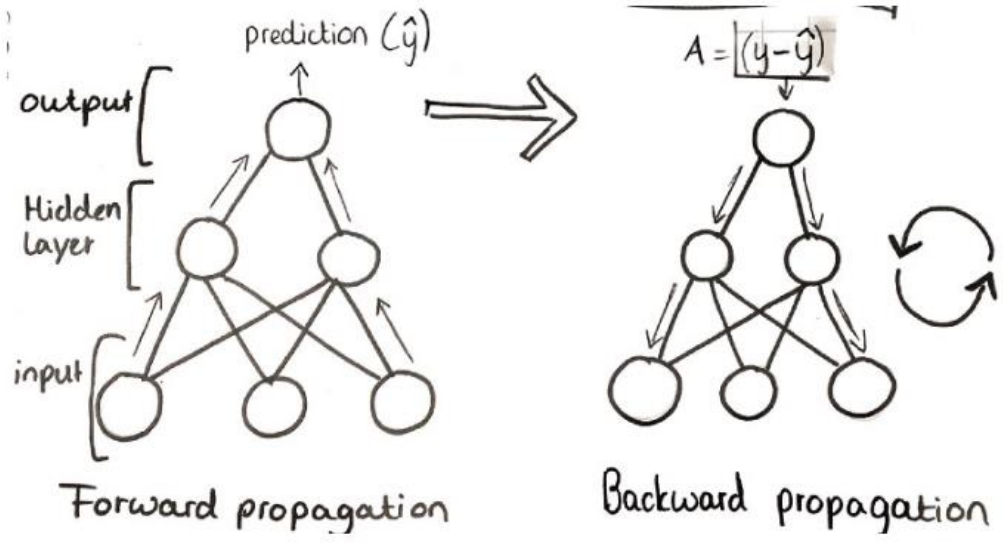
Step5:training

In order to reach the **optimal weights and biases** that will give us the desired output (the three wine cultivars), we will have to **train** our neural network.

a neural network will have to undergo many epochs or iterations to give us an accurate prediction.

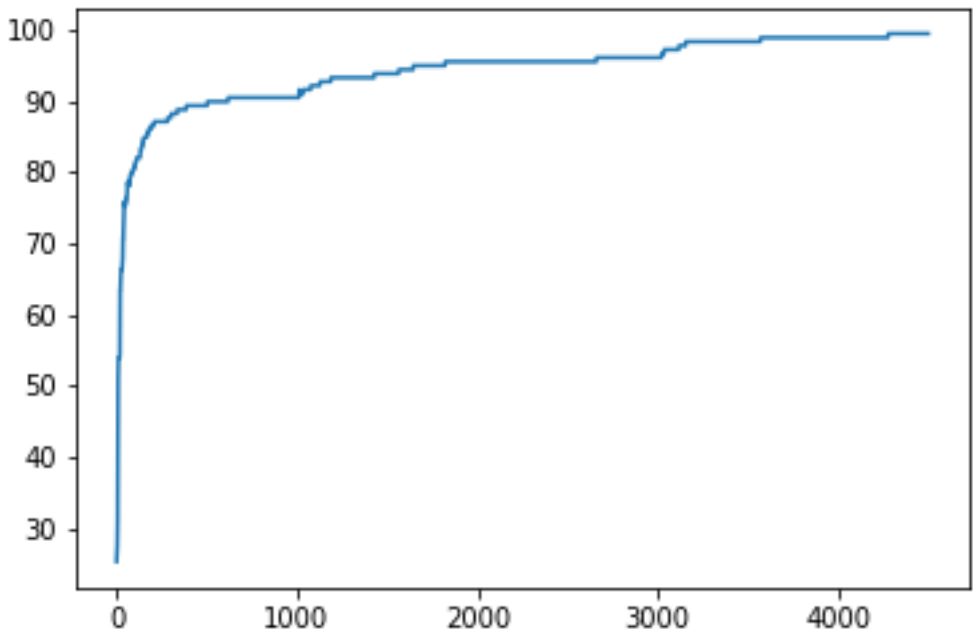
The learning rate is the multiplier to update the parameters. It determines how rapidly they can change. If the learning rate is low, training will take longer. However, if the learning rate is too high, we might miss a minimum. The learning rate is expressed as:

a:=a-a*dl(w)/da




```
model = initialize_parameters(nn_input_dim=13, nn_hdim=5, nn_output_dim=3)
model = train(model, X, y, learning_rate=0.07, epochs=4500, print_loss=True)
```

RESULT:



After 5000 epochs, we have accuracy near 100(99.4382022471910)

Part2:

We can do this with tensoeflow or keras , too:

```
import tensorflow as tf
import keras.layers

n_neurons_h = 178
n_neurons_out = 3
n_epochs = 4500
learning_rate = 0.7

model = tf.keras.Sequential()
model.add(layers.Dense(n_neurons_h, activation="tanh"))
model.add(layers.Dense(n_neurons_h, activation="tanh"))
model.add(layers.Dense(n_neurons_out, activation="softmax"))

model.fit(training_data, training_labels, epochs=n_epochs, batch_size=32)

model.compile(optimizer=tf.train.GradientDescentOptimizer(learning_rate=learning_rate),
loss="binary_crossentropy",
            metrics=["accuracy"])
model.fit(training_X, training_y, epochs=n_epochs)
```

and the result would be the same.

TASK 2: MNIST

Implementation of kmeans on MNIST dataset(a clustering Task)

About Tensorflow:

Tensorflow is an open source library created by the Google Brain Trust for heavy computational work, geared towards machine learning and deep learning tasks. It is built on C, C++ making its computations very fast while it is available for use via a Python, C++, Haskell, Java and Go API.

It created data graph flows for each model, where a graph consists of two units – a **tensor** and a **node**.

- **Tensor:** A tensor is any multidimensional array.
- **Node:** A node is a mathematical computation that is being worked at the moment.

A data graph flow essentially maps the flow of information via the interchange between these two components. Once this graph is complete, the model is executed and the output is computed.

You can learn a lot more from the tensorflow official document

Now let's begin start building handwritten digits recognition application. To start we need the dataset of handwritten digits for training and for testing the model. MNIST is the most popular dataset having handwritten digits as image files.

ABOUT THE MNIST DATASET

Mnist database handwritten digits

To begin our journey with Tensorflow, we will be using the MNIST database to create an image identifying model based on simple **feedforward neural network** with no hidden layers.

MNIST is a computer vision database consisting of handwritten digits, with labels identifying the digits. As mentioned earlier, every **MNIST** data point has two parts: an image of a handwritten digit and a corresponding label.

We'll call the images **"x"** and the labels **"y"**. Both the training set and test set contain images and their corresponding labels; for example, the training images are **mnist.train.images** and the training labels are **mnist.train.labels**.

Each image is **28** pixels by **28** pixels. We can interpret this as a big array of numbers. We can flatten this array into a vector of 28×28 = 784 numbers.

It doesn't matter how we flatten the array, as long as we're consistent between images. From this perspective, the MNIST images are just a bunch of points in a **784**-dimentional vector space.

Step1: preprocessing data

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
full_data_x = mnist.train.images
```

parameters(google implementation):

```
# Parameters
num_steps = 50 # Total steps to train
batch_size = 1024 # The number of samples per batch
k = 25 # The number of clusters
num_classes = 10 # The 10 digits
num_features = 784 # Each image is 28x28 pixels
```

```
# Input images
X = tf.placeholder(tf.float32, shape=[None, num_features])
# Labels (for assigning a label to a centroid and testing)
Y = tf.placeholder(tf.float32, shape=[None, num_classes])
```

```
# K-Means Parameters
kmeans = KMeans(inputs=X, num_clusters=k, distance_metric='cosine',
                use_mini_batch=True)

# Build KMeans graph
training_graph = kmeans.training_graph()
```

Result:

Step 1, Avg Distance: 0.341471
Step 10, Avg Distance: 0.221609
Step 20, Avg Distance: 0.220328
Step 30, Avg Distance: 0.219776
Step 40, Avg Distance: 0.219419
Step 50, Avg Distance: 0.219154

Test Accuracy: 0.7127

My implementation was like this:

(I got help from: https://www.kaggle.com/raoulma/mnist-image-class-tensorflow-cnn-99-51-test-acc)

And I used:

tf.contrib.factorization.KMeans

CLASS KMEANS

Defined in tensorflow/contrib/factorization/python/ops/clustering_ops.py.

Creates the graph for k-means clustering.

After trial and error on accuracy, I got these:

```
# Parameters
num_steps = 50 # Total steps to train
batch_size = 1024 # The number of samples per batch
k = 78 # The number of clusters
num_classes = 10 # The 10 digits
num_features = 784 # Each image is 28x28 pixels
```

Result:

Step 1, Avg Distance: 0.284920
Step 10, Avg Distance: 0.184646
Step 20, Avg Distance: 0.183262
Step 30, Avg Distance: 0.182651
Step 40, Avg Distance: 0.182276
Step 50, Avg Distance: 0.182016

Test Accuracy: 0.8375