Deep Learning with Keras

Joseph Rejive

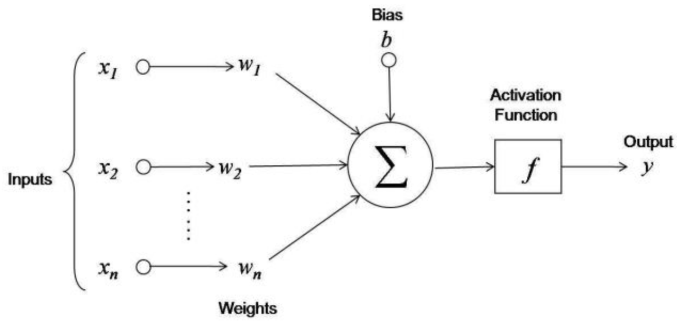
May 2019

1 Introduction

A neural network is a powerful algorithm which can detect complicated patterns in data. It's loosely modeled after the human brain, consisting of hundreds to thousands of processing cores (or neurons). Neural networks have many applications, which range from voice recognition to object detection to language translation. In this lecture, we'll be focusing on using a neural network to classify handwritten digits.

2 The Neuron

The fundamental building block of neural networks is the neuron.

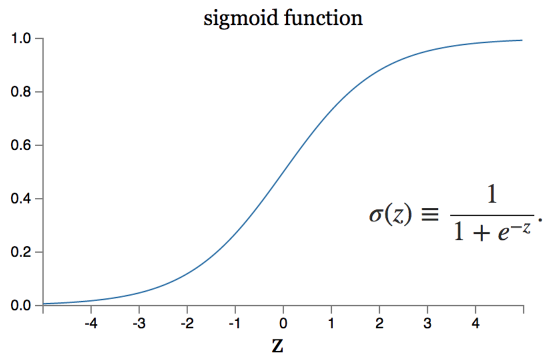


A neuron consists of two or more inputs, a weight for each input, a bias, and an activation function. In this case, the output is simply:

$$y = f(w_1x_1 + w_2x_2 + ...w_nx_n... + b)$$

Here, the bias is a term which is similar to the y-intercept in a linear model. It's a constant which helps the neural network fit the data in the most optimal way.

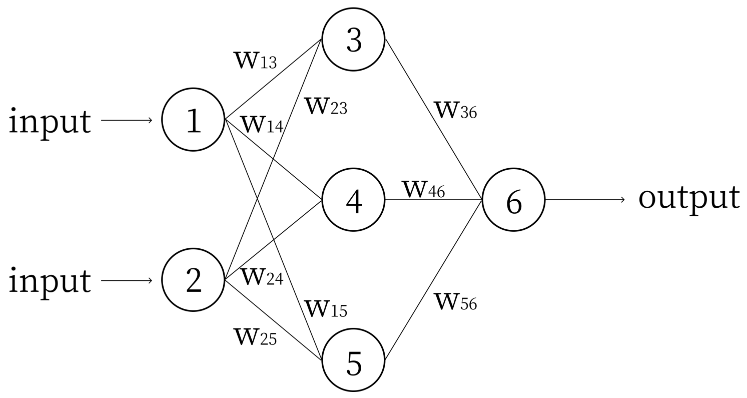
The activation function is a nonlinear function which maps an input to an output. Nonlinear activation functions are important as they help the model learn complicated patterns. Without the activation function, the output would merely be a linear function of the input data. For complicated tasks such as image classification, we'll need a neural network that is capable of fitting to nonlinear data. One popular activation function is the sigmoid function.



The sigmoid function returns a value that's always in the range (0,1). Essentially it takes any number from (-∞, ∞) and compresses it between (0,1).

3 Neural Network

A neural network is just multiple neurons connected together.



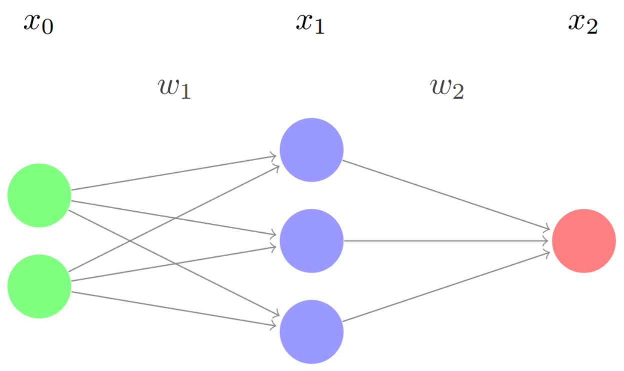
We'll denote the value of node i as n_i and the bias of node i as b_i. The sigmoid function will be represented as σ(x). Computing the network using these variables, we get:

$$n_3 = \sigma(w_{13}n_1 + w_{23}n_2 + b_3)$$
$$n_4 = \sigma(w_{14}n_1 + w_{24}n_2 + b_4)$$
$$n_5 = \sigma(w_{15}n_1 + w_{25}n_2 + b_5)$$
$$n_6 = \sigma(w_{36}n_3 + w_{46}n_4 + w_{56}n_5 + b_6)$$

We can rewrite this as

$$\begin{bmatrix} n_3 \\ n_4 \\ n_5 \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{13} & w_{23} \\ w_{14} & w_{24} \\ w_{15} & w_{25} \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} + \begin{bmatrix} b_3 \\ b_4 \\ b_5 \end{bmatrix} \right)$$

If we relabel the diagram to show layers and weights, we get:



We can represent each layer of the neural network as a column vector of dimension $n * 1$ where n is the number of neurons in the layer. Therefore, we can represent the layers as,

$$x_0 = \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} \qquad x_1 = \begin{bmatrix} n_3 \\ n_4 \\ n_5 \end{bmatrix} \qquad x_2 = \begin{bmatrix} n_6 \end{bmatrix}$$

Similarly, the weights that connect one layer to the next can also be represented by a matrix.

$$w_1 = \begin{bmatrix} w_{13} & w_{23} \\ w_{14} & w_{24} \\ w_{15} & w_{25} \end{bmatrix} \qquad w_2 = \begin{bmatrix} w_{36} \\ w_{46} \\ w_{56} \end{bmatrix}$$

Each layer also has a bias vector which is the same dimension as the layer itself. In this example we have

$$b_1 = \begin{bmatrix} b_3 \\ b_4 \\ b_5 \end{bmatrix} \qquad b_2 = \begin{bmatrix} b_6 \end{bmatrix}$$

We now understand the fundamentals of how neural networks make predictions. To calculate each layer, the model multiplies a weight matrix by the previous layer, adds a bias vector, and passes the result through an activation function. In any n layer network, for a given layer x_{i+1} (assuming $0 \le i < n-1$):

$$x_{i+1} = \sigma(w_i x_i + b_{i+1})$$

4 Backpropagation

Backpropagation is the key to how neural networks 'learn.' Neural networks are always given training data and labels. The training data is the input to the model while the labels are the ground truth (what the input actually is). In object classification, the inputs would be the pixel values for the image while the label would be what the picture actually shows. The model trains by comparing its output to the ground truth and updating the weights and biases to reduce the error.

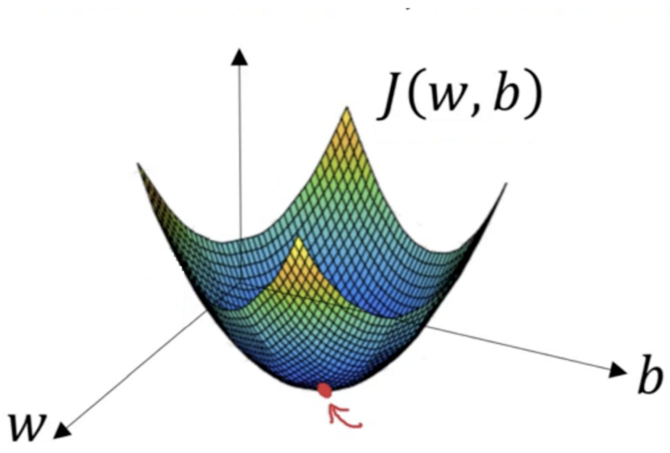
4.1 Error

Let's take a look at the previous neural network. If x_2 is the output, we can define the error between the output and the ground truth y to be defined by the function E:

$$E = \frac{1}{2}||x_2 - y||^2$$

The error of our neural network changes based off x_2, which is dependent on the weights and biases in our model. Neural networks learn by adjusting the weights and biases such that the error is minimized, and to do so, we use a technique called gradient descent.

4.2 Gradient Descent



In gradient descent, we start at an arbitrary point and move a small amount in the direction of the greatest decline. When we move in the direction of the greatest decline, we are updating our weight and bias matrices in a way which reduces our error. We iteratively repeat this process until we reach the minimum error.

Finding the values to update the weight and bias matrices by requires multi-variable calculus, as we need to take the partial derivative of the error function with respect to each of our weights and biases. This is beyond the scope of this lecture, but if you are interested, a more rigorous explanation of gradient descent can be found on tjmachinelearning.com.

5 Classifying Handwritten Digits with Keras

Keras is a high level machine learning framework. It runs off a backend (typically Tensorflow, Theano, or CNTK). Now, we'll create a model that can classify handwritten digits.

```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Activation
from keras.optimizers import SGD
from keras.layers import Dense
from keras.utils import np_utils

batch_size = 100
epochs = 20
layer1_size = 512
layer2_size = 256
num_classes = 10
```

In this section, we imported our dependencies and defined some parameters of our model. The batch_size specifies how many images at a time our model will train on. Since 60,000 images is too large to fit into memory, we break them down into smaller chunks and iteratively train over the entire dataset in 100 image chunks. The epochs parameter determines how long we will train for. One epoch consists of training over all 60,000 training images and performing gradient descent. We'll be performing this process 20 times. Finally, we assigned values for the number of neurons in each layer. The first layer will have 512 neurons, the second layer will have 256 neurons, and the output layer will have 10 neurons.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train.reshape(60000, 784), x_test.reshape(10000, 784)
x_train, x_test = x_train/255.0, x_test/255.0
y_train, y_test = np_utils.to_categorical(y_train), np_utils.to_categorical(y_test)
```

Here, we grabbed our training and testing data. We need to reshape the input images into a matrix that is compatible with our neural network. Therefore, we squashed the 28 by 28 image into one long array of 784 pixels. We then normalized the inputs by dividing each pixel value by 255. Since grey-scale pixels range from 0 to 255, dividing by 255 causes the pixel values to range from 0 to 1 instead.

The last line in this code snippet serves to one-hot encode the labels. y_train and y_test contain numbers from 0-9 which are the ground truth for the images in x_train and x_test. By one-hot encoding the labels, we convert each number to an array of size 10 where all the elements are 0s except for the index represented by the digit, which is a 1. For example, we would represent the digit 3 through the array: [0,0,0,1,0,0,0,0,0,0].

```
model = Sequential()
model.add(Dense(layer1_size, activation = 'sigmoid', input_shape = (784,)))
model.add(Dense(layer2_size, activation = 'sigmoid'))
model.add(Dense(num_classes, activation = 'softmax'))
```

In this section, we defined the actual neural network architecture. We created a Sequential model, which simply means that data will be transferred through the network in a linear fashion. The Dense layer is our weight matrix. The three Dense layers we specified create two hidden layers of size 512 and 256, respectively, and an output layer of size 10. We used the sigmoid activation function in the first two hidden layers.

$$S(y_i) = \frac{e^{y_i}}{\sum\limits_j e^{y_j}}$$

The softmax function is an activation function which assigns a probability to each item in an array. For each element in an array, the function raises e to that power and divides it by the sum of raising e to the power of each element in the array. We use this activation function at the final layer of a classification task so that the model can determine the probability of an input imagine belonging to each class.

```
model.compile(loss = 'categorical_crossentropy', optimizer = SGD(lr = 0.05), metrics =
    ['accuracy'])
model.fit(x_train, y_train, batch_size = batch_size, epochs = epochs, verbose = 1)
loss, accuracy = model.evaluate(x_test, y_test)
print(loss, accuracy)
```

In this final section, we compiled our model by specifying a loss function and an optimizer. In this example, we're using the SGD (Stochastic Gradient Descent) optimizer. We then fit our model on our training data and then evaluated our model's performance using our testing data.

By no means is this architecture the best one for classifying images. There are other algorithms, such as Convolutional Neural Networks (CNNs), which outperform classic Feedforward Neural Networks. To improve the accuracy of this feedforward network, you can try adding more layers, changing the activation functions, and increasing/decreasing the size of each hidden layer.