Lab 7 - Neural Networks I

Kyle Swanson

January 22, 2018

0 Introduction

0.1 Lab Overview

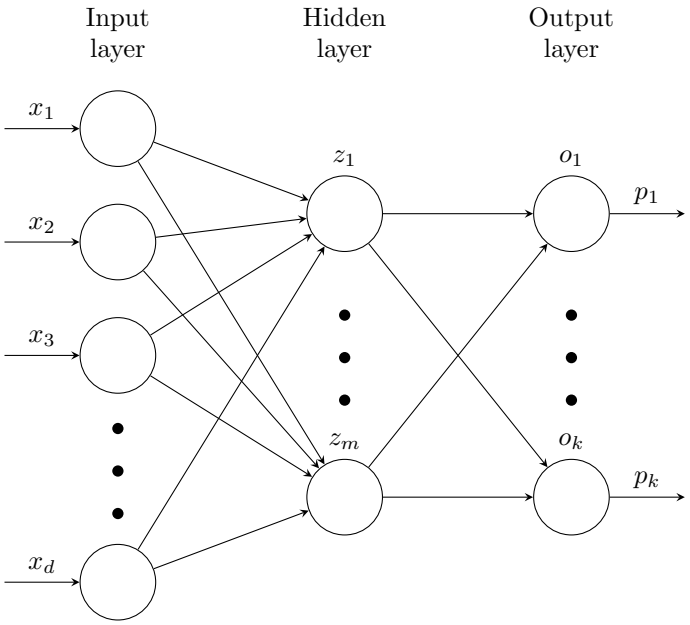
This week we're learning about neural networks, which are the state-of-the-art in machine learning. Deep neural networks in particular have extraordinary computation power and can learn to solve problems and perform tasks that even humans struggle with.

Thanks to the extreme popularity of neural networks recently, there are several excellent libraries (mostly written in Python) which make building deep neural networks easy. We'll be using one of them (keras) in Lab 9. However, I think it's a valuable (and fun!) experience to build a neural network from scratch using only the mathematical operations provided by Python and numpy. My hope is that building a neural network from scratch will give you a better understanding of how neural networks really work on the inside so that you understand what keras and the other neural network libraries are doing behind the scenes.

Building a neural network from scratch takes some work, so I've split it into two labs, Lab 7 (today) and Lab 8 (tomorrow). In these labs, we're going to be building a network with only a single hidden layer (a "shallow" neural network). In this lab (Lab 7), we're going to be building the prediction part of the network, which will take the input data, pass it through the network, compute the output probabilities, and make a prediction based on the output probabilities. In tomorrow's lab (Lab 8), we'll be implementing the learning portion of the network, which involves using the backpropagation algorithm and gradient descent to update the weights of the network. Once the network is complete at the end of Lab 8, we'll test it on a simple data set and we'll examine the decision boundaries it draws when the network uses different numbers of neurons in its single hidden layer.

0.2 Single Layer Neural Network

0.2.1 Neural Networks Overview



As a reminder, above is a diagram of a single layer neural network. The first layer is the input layer. If our input data points are d-dimensional, then there will be d input units. The middle layer is the hidden layer, which consists of m neurons (m is a number that we get to choose). The final layer is the output layer, which consists of k neurons where k is the number of classes that we're trying to choose between. For instance, if we're trying to predict whether an image shows a car, a train, or an airplane, then we have three classes ($k = 3$) and our output layer will have three neurons, one each for car, train, and airplane.

Between layers, we have weights which connect each layer to the next layer. There is a weight between every unit in one layer and every unit in the next layer, as illustrated in the diagram above. Note that there is also a bias weight added to each neuron, but it is not illustrated in this diagram. The goal of neural networks is to learn the weights which will allow the network to make the most accurate predictions. Today we are just concerned with using the weights to make predictions; tomorrow we will discuss how to learn the weights from the data.

0.2.2 Predicting with Neural Networks

In order to make a prediction with a neural network, we need to pass a data point through the network to determine its final output, which requires a series of computations. Our input is a d-dimensional vector x, as represented by the input layer. We multiply the input by the weights connecting the input layer

to the hidden layer and we sum the result for each neuron to get the input to the neuron z. Then we apply a non-linear activation function f to get $f(z)$, the output of the neuron. Next, we multiply the outputs $f(z)$ of the neurons in the hidden layer by weights connecting the hidden layer to the output layer to get the inputs o to the output layer. Finally, we apply the softmax function to convert these inputs into probabilities p, and our prediction is the class with the highest probability.

Before diving into the equations, we need some notation. Let x_i be the i^{th} input unit, let z_j be the input to the j^{th} hidden unit, let $a_j = f(z_j)$ be the output of the j^{th} hidden unit (where f is the activation function), let o_l be the input to the l^{th} output unit, and let p_l be the output of the l^{th} output unit (this will be the probability of class l). Then w^1_{ij} will be the weight connect x_i (input unit) to z_j (hidden unit input) and w^2_{jl} will be the weight connecting a_j (hidden unit output) to o_l (output unit). w^1_{0j} will be the bias added to z_j (hidden unit) while w^2_{0j} will be the bias added to the o_l (output unit).

We can represent the operations of a feed-forward neural network in two ways, either using summation notation or using matrix notation. Summation notation is easier to understand, but matrix notation is more concise and results in faster code. Below I'll write the same operations in both forms.

Summation Notation
Input to j^{th} hidden unit:

$$z_j = \sum_{i=1}^{d} w^1_{ij} x_i + w^1_{0j} = w^1_{1j} x_1 + w^1_{2j} x_2 + \cdots + w^1_{dj} x_d + w^1_{0j}$$

Output from j^{th} hidden unit $a_j = f(z_j)$.
Input to l^{th} output unit:

$$o_l = \sum_{j=1}^{m} w^2_{jl} a_j + w^2_{0l} = w^2_{1l} a_1 + w^2_{2l} a_2 + \cdots + w^2_{ml} a_m + w^2_{0l}$$

Output from l^{th} output unit $p_l = softmax(o_l)$ where

$$softmax(o_l) = \frac{e^{o_l}}{\sum_{l'=1}^{k} e^{l'}}$$

Vector/Matrix Notation
Define the following vectors/matrices:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}, \; z = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix}, \; a = f(z) = \begin{bmatrix} f(z_1) \\ f(z_2) \\ \vdots \\ f(z_m) \end{bmatrix}, \; o = \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_k \end{bmatrix}, \; p = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_k \end{bmatrix},$$

$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & \dots & w^1_{1m} \\ w^1_{21} & w^1_{22} & \dots & w^1_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w^1_{d1} & w^1_{d2} & \dots & w^1_{dm} \end{bmatrix}, \; b^1 = \begin{bmatrix} w^1_{01} \\ w^1_{02} \\ \vdots \\ w^1_{0m} \end{bmatrix}, \; W^2 = \begin{bmatrix} w^2_{11} & w^2_{12} & \dots & w^2_{1k} \\ w^2_{21} & w^2_{22} & \dots & w^2_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w^2_{d1} & w^2_{d2} & \dots & w^2_{mk} \end{bmatrix}, \; b^2 = \begin{bmatrix} w^2_{01} \\ w^2_{02} \\ \vdots \\ w^2_{0m} \end{bmatrix}$$

Using these vectors/matrices, we can easily represent the operations of the feed-forward network:

$$z = x^T W^1 + b^1$$
$$a = f(z)$$
$$o = a^T W^2 + b^2$$
$$p = softmax(o)$$

Note that T is the matrix transpose operation (`https://en.wikipedia.org/wiki/Transpose`).

Finally, our prediction is the class with the greatest probability, which can be represented as follows:

$$\text{prediction} = \underset{l}{\text{argmax}} \; p_l$$

1 The `NN` Class

In this lab, you will be building a Python class called `NN` in `lab7.py` which will represent a single layer neural network. Your job in the following sections is to implement some of the methods of this class in order to make a prediction using the neural network.

Since the neural network won't be complete until the end of next lab, you won't have a chance to directly test your implementation on data. Instead, I've written some tests in a file called `checker.py` which will test your implementations. After completing each of the following sections, run `python checker.py` and you should see that you are passing the relevant tests.

2 Initializing Weights

The first step in building a neural network is to initialize the weights between the input and hidden layers and between the hidden and output layers. This is accomplished by the `initialize_weights` method. This method should create four numpy arrays: `self.W1`, `self.b1`, `self.W2`, and `self.b2`. These correspond to the weight and bias matrices/vectors W^1, b^1, W^2, b^2 defined above in section 0.2.2.

These arrays should be initialized randomly (use the numpy function `np.random.randn`) and should have the correct shapes. Use the sizes `self.n_input`, `self.n_hidden`,

and `self.n_output` (which correspond to d, m, k respectively) to construct these arrays.

After completing your implementation, run `python checker.py`. If your implementation is correct, you should see that 5 tests were run but there were only 4 errors (i.e. you passed 1 test).

3 Softmax

Next you need to implement the `softmax` method, which takes in a numpy array containing the input to the output layer $o = \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_k \end{bmatrix}$ and computes the softmax function on the vector. This function should return the vector

$$p = softmax(o) = \begin{bmatrix} softmax(o_1) \\ softmax(o_2) \\ \vdots \\ softmax(o_k) \end{bmatrix} = \begin{bmatrix} \frac{e^{o_1}}{\sum_{l'=1}^{k} e^{o_{l'}}} \\ \frac{e^{o_2}}{\sum_{l'=1}^{k} e^{o_{l'}}} \\ \vdots \\ \frac{e^{o_k}}{\sum_{l'=1}^{k} e^{o_{l'}}} \end{bmatrix}$$

After completing your implementation and running `python checker.py`, you should only be failing 3 tests.

4 Feed-forward

Now you need to implement the feed-forward step of the neural network, which happens in the `feed_forward` method. This method takes as input a data point x (represented as a length d numpy array) and returns a length k numpy array containing the output probabilities as computed by the network. You should try to work with matrices and vectors where possible, using the matrix equations from section 0.2.2. The activation function we will be using is the hyperbolic tangent function, $f(z) = \tanh(z)$ (in numpy this is `np.tanh`). You should also make use of your `softmax` method to convert the vector o to a vector of probabilities p, which is the vector that you will return.

Once your implementation is complete, run `python checker.py`. You should only be failing two tests.

5 Predicting

Next is an method called `predict`. It takes in a data point x and makes a prediction. You should first use your `feed_forward` method to compute the probabilities of each class. Then return the index of the class with the highest

probability. This is represented as: prediction = $\underset{l}{\text{argmax}}\ p_l$ where p_l is the probability of the l^{th} class. Numpy has a function `np.argmax` which makes this easy.

Once your implementation is complete and you have run `python checker.py`, you should only be failing one test.

6 Computing Accuracy

Finally, implement the `compute_accuracy` method. This method takes in a numpy matrix of data points X where each row represents a data point (vector) x along with a numpy array y with the correct class for each data point. Use the method `predict` to make a prediction for each row of X and then use `sklearn`'s `accuracy_score` method to compute the accuracy of the predictions by comparing the predictions to the true classes contained in y.

After your implementation is complete, run `python checker.py` one more time. You should be passing all the tests. Now you have a fully functioning neural network! The only problem is that it's predictions will be terrible because we haven't trained the weights. Tomorrow in lab we'll write the code which will train the weights of the network to make good predictions.