Lab 8 - Neural Networks II

Kyle Swanson

January 23, 2018

0 Introduction

This lab is a continuation of Lab 7, where we were building our own neural network from scratch. In Lab 7, we wrote the methods which performed the feed-forward computation, which took in a data point and used the weights of the network to compute the probability of each class. We then made a prediction for this data point based on the class with the highest probability. However, these predictions are going to be terrible unless we train the weights of the network to make good predictions. This lab is going to be focused on the learning phase, where we apply backpropagation and gradient descent to learn the weights of the network.

After you have implemented the learning phase (Parts 1, 2, and 3), you'll be ready to finally train and test your neural network in Part 4. In Part 5, we'll experiment with different numbers of hidden units to see how and when overfitting can occur. In Part 6, we're going to build a neural network using the Python package Keras, which is fortunately much easier than building a neural network from scratch. In Parts 7 and 8, you'll be using your Keras model on different datasets to see the sorts of decision boundaries that neural networks are able to draw. Finally, in Part 9 you're going to have freedom to explore neural networks on your own. I've provided some suggestions for things you can try, but this is really a time to have fun and explore the power of neural networks.

Note: You'll notice that the methods which you implemented in Lab 7 already have solutions in Lab 8. You can copy over your solutions if you want, but I've provided optimized solutions which will run faster and avoid some errors with integer overflow when working with large numbers, so I recommend using the method implementations I have provided.

1 Computing Loss

While accuracy (which you wrote a method to compute in Lab 7) is a decent measure of how well our network is performing, it only provides us with a binary correct/incorrect for each data point. It's valuable to have a more sensitive measure of how far away we are from predicting the correct answer, so we define

a loss function to measure this. The loss function also gives us an objective to optimize when updating the weights (see Part 2).

The loss function that we're going to use for our network is called the *categorical cross-entropy* loss, which is the most common loss function used for networks which are making classification predictions like ours. The categorical cross-entropy loss is defined as follows:

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k}[[y^{(i)} == j]]\log(p_{ij})$$

where n is the number of data points, k is the number of classes, $y^{(i)}$ is the correct label of the i^{th} data point, $[[y^{(i)} == j]]$ is 1 if $y^{(i)} = j$ and 0 otherwise, and p_{ij} is the probability predicted by the network of the j^{th} class for the i^{th} data point.

Your job is to implement the method `compute_loss` in the `NN` class in `lab8.py` according to the equation for $J(\theta)$ above. Note that unlike in Lab 7 where most of the functions took as input a vector x representing a single data point, the `compute_loss` method takes in a matrix X where each row represents a data point. n is the number of data points so it is equal to the length of the matrix X, while k is the number of classes so it is equal to `self.n_output`. A vector of correct labels y is passed into the function. The probabilities p_{ij} can be obtained by calling `self.feed_forward` with the matrix of data points X, which will return a matrix of probabilities where the i, j entry contains the probability that the i^{th} data point is the j^{th} class (i.e. p_{ij}).

Once you've completed you're implementation, proceed to Part 2 (you'll be able to check in Part 4 whether your implementation is correct).

2 Updating Weights

In this part, we're going to finally learn from the data and update the weights of the network to decrease the loss and make more accurate predictions. This involves three steps:

1. Performing a forward pass through the network to determine the activations of the neurons.

2. Computing the derivatives of the weights using backpropagation (this requires knowing the activations of the neurons from the forward pass).

3. Using the derivatives to update the weights with gradient descent.

Your task is to implement these three steps in the `train_step` method.

First, copy and past the code from the `feed_forward` method into the `train_step` method (but leave out the `return probs` line). It might seem silly to copy and paste code rather than just calling the function and getting

the result, but we're going to need some of the intermediate values which are computed as part of the forward pass.

Next, compute the derivative of the loss function with respect to the two weight matrices and the two bias vectors. Rather than making you compute the derivatives by hand (which is a huge pain), here are the derivatives (and two helper variables δ_1 and δ_2):

$$\delta_2 = P - Y$$
$$\frac{\partial L}{\partial W_2} = A^T \delta_2$$
$$\frac{\partial L}{\partial b_2} = \delta_2$$
$$\delta_1 = (1 - A^2) \circ \delta_2 W_2^T$$
$$\frac{\partial L}{\partial W_1} = X^T \delta_1$$
$$\frac{\partial L}{\partial b_1} = \delta_1$$

Write the code which will compute each of these derivatives. You can essentially write these equations almost exactly as stated above but using numpy syntax. Note that all multiplications are dot products (`np.dot`) except for $\circ$, which is elementwise multiplication (just use `*`). The one difference is that for $\frac{\partial L}{\partial b_2} = \delta_2$ and $\frac{\partial L}{\partial b_1}$, you need to sum down the columns of the δ_2 and δ_1 matrices, so you need to use `dL_db2 = np.sum(delta2, axis=0)` and `dL_db1 = np.sum(delta1, axis=0)`.

Finally, perform the gradient descent step for the two weight matrices and the two bias vectors:

$$W_2 = W_2 - \eta \frac{\partial L}{\partial W_2}$$
$$b_2 = b_2 - \eta \frac{\partial L}{\partial b_2}$$
$$W_1 = W_1 - \eta \frac{\partial L}{\partial W_1}$$
$$b_1 = b_1 - \eta \frac{\partial L}{\partial b_1}$$

where η is the learning rate (which you can access with `self.learning_rate`). Remember that W_2, b_2, W_1, b_1 are available as `self.W2, self.b2, self.W1, self.b1`.

When your implementation is complete, proceed to Part 3.

3 Training

In this section, you need to implement the `train` method. This method should loop `num_epoch` times, and on each loop, it should call `self.train_step` with X and y to update the weights based on all of the training examples. Additionally, if `verbose` is set to `True`, it should compute the loss and accuracy (using `self.compute_loss` and `self.compute_accuracy`) and print both to the terminal.

Once your implementation is complete, your neural network is ready to go. Proceed to Part 4, where we will test our network.

4 Testing our Neural Network

Now we're finally ready to test our neural network.

Go into `main.py`, uncomment Part 4, and run `python main.py`. You should first see a plot of the data. After closing this plot, you should see 100 losses and accuracies printed to the terminal, since we're training our network for 100 epochs. After training is complete, you should see another plot which shows the same data along with the decision boundary of the neural network. You can see that the network easily learns a non-linear decision boundary which does a very good job of separating the data, even though it is a relatively difficult dataset to separate.

Once you think your network is working, show me so I can check over it.

5 Experimenting with the Number of Neurons

The size of the input layer of our network is determined by the dimensionality of the data and the size of the output layer is determined by the number of classes, but the size of the hidden layer is entirely our choice. In this part, we'll be experimenting with several different size of hidden layers.

Go into `main.py`, uncomment Part 5, and run the file. After a few seconds, you should see a plot with 9 different decision boundaries on the same data. Each one is the result of training your neural network on the dataset with a different number of neurons in the hidden layer.

Notice how the decision boundaries with a small number of neurons aren't able to separate the data. This is *underfitting* - our model is not complex enough to capture the true distribution of the data. Notice also how the decision boundaries with a large number of neurons take on strange shapes in order to include more training points rather than capturing the general trend of the data. This is *overfitting* - our model is too complex and memorizes the training set rather than learning general patterns. The best models are the ones in the middle, which have enough complexity to capture the underlying distribution of the data but don't have some much complexity that they overfit to the training data.

6 Neural Networks with Keras

After spending a lab and half building our own neural network from scratch, we're going to see in this part how incredibly easy it is to build neural networks with Keras, a deep learning package for Python.

Keras is a deep learning API which runs on top of other neural network packages. We're going to be using Tensorflow as the backend for Keras. To install Keras and Tensorflow, run the following commands:

```
pip install keras
pip install tensorflow
```

Go back into `lab8.py` and implement the function `keras_nn`. This function takes in the sizes of the input, hidden, and output layers and it should return a Keras model representing a neural network with a single hidden layer. To build a Keras model, first create a `Sequential` object. Call it `model`. Then add to the model a `Dense` layer with `n_hidden` neurons, an `input_dim` of size `n_input`, and an `activation` of "relu". Next, add another `Dense` layer, this time with `n_output` neurons and with an `activation` of "softmax". Finally, return the `model`. Feel free to use Google and the Keras documentation (`https://keras.io/`) if you're unsure of what to do (or you can just ask me). It should only take about 4 lines of code, which is infinitely simpler than the 200 or so lines of code needed to write a neural network from scratch.

When your implementation is complete, uncomment Part 6 in `main.py` and run `python main.py`. You should first see a plot of the loss and accuracy as the model trains for 500 epochs, then you should see a plot of the decision boundary of the model.

7 Clustering Dataset with Keras

Now that we have a Keras model built, let's train it on various datasets to see what sort of decision boundaries it can learn.

If you remember from Lab 4, we showed the results of the k-nearest neighbors algorithm, a very simple non-linear algorithm which works well for data which appears in small clusters. (Take a look here if you don't remember: `https://github.com/swansonk14/IntroML/blob/master/Labs/Lab4Solutions/knn.png`).

While perceptron or SVM would struggle to find decision boundaries that can separate clusters like these, it turns out that single layer neural networks are able to classify data in clusters with ease.

Uncomment Part 7 in `main.py` and run the file. You'll see that a neural network with just 50 neurons in the hidden layer does a great job of separating the data into the 5 different clusters.

8 Additional Datasets with Keras

Uncomment Part 8 in `main.py` and run the file. First you'll see a plot with 6 different datasets, then (after a few seconds) you'll see the same datasets but with the decision boundaries learned by a neural network with 50 neurons in the hidden layer. You can see how the neural network is amazingly flexible and classifies all 6 datasets with ease. Back in Lab 4 we would have had to work hard to determine the right non-linear function to use for each of these datasets, but with neural networks, the non-linearity built into the network can freely capture almost any distribution of data with no extra work required by us. It's all learned directly from the data.

9 Free Experimentation

The last couple of parts gave a few demonstrations of the power of neural networks and hopefully sparked your curiosity about neural networks. In this part, I want you to take some time to play around with neural networks and learn more about them. You're free to do whatever you want (the more creative, the better!), but here are a couple suggestions:

- Try building even more complicated datasets and see what sort of decision boundaries the neural network can draw. sklearn has many functions for creating datasets (`http://scikit-learn.org/stable/datasets/index.html`) or you can create your own.
- Try experimenting with different numbers of neurons in the hidden layer. When does the network overfit? When does it underfit?
- So far we've only been working with single layer neural networks. Try adding more layers to the Keras model and see how it changes.
- Experiment with different types of networks. Try different numbers of neurons, different numbers of layers, different activation functions, different loss functions, etc.
- I said in lecture that neural networks can approximate any (reasonable) function. Try to see how well a neural network can approximate a function. For example, create a set of 1-dimensional training points x and labels y where $y = x^2$. Train the network on these data points and labels, then try plotting the predictions of the network as a function of x. Does the resulting function look quadratic?

Have fun!