Neural Networks Using TensorFlow

Prathusha Boppana

June 11, 2019

1 Introduction/Background

I started this project hoping to understand TensorFlow and Neural Networks better. I did this by reading "Hands-On Machine Learning with Scikit-Learn & TensorFlow" by Aurelien Geron and doing the exercises for each chapter. I first learned about TensorFlow and how the code for it is structured and executed. Then, I learned about basic Deep Neural Networks (DNNs) and how they work in detail with each parameter and optimization. Basically, the DNN takes the data and applies weights and functions to it, optimizes the functions for the weights, and spits out a model to classify the data with. We can then test the model on a certain portion of the validation set and run the data through again in epochs so that the DNN can continue learning. There are also several ways to normalize this algorithm, such as batch normalization and early stopping. Then, I learned about Convolutional Neural Networks (CNNs) and what distinguishes them as a specific type of Neural Net (NN). This NN takes the data, applies a filter (or several filters) in sections, takes the max values from each filter section, reduces the dimensionality, and spits out a model. This method is very effective for classification problems shown by its high yields of accuracy.

2 Experiment

I used the Labeled Faces in the Wild (LFW) recognition dataset, which consisted of images of people's faces. For some people there would be multiple pictures and for others there was a single picture of them. This included pictures of actors and actresses while they were in character which made the

Figure 1: Example pictures from the Labeled Faces in the Wild (LFW) dataset.

classification problem interesting. There was one dataset of pairs where there were two pictures side-by-side and the algorithm would have to distinguish if it was the same person, but it did not work with my CNN. Therefore, I used the full collection of individual pictures, which is commonly used to solve the supervised problem of categorizing the pictures into the various people. There were ∼13,000 pictures with ∼5,000 people and ∼1,400 of them having multiple pictures of themselves in the dataset. The images were 50x37 pixels. Some example images are below (1).

When I called the LFW dataset, the function asked for the minimum number of faces per person and a resize ratio. I kept these parameters constant as 70 and 0.4, respectively. I fetched one set for training and another set for testing. Then, I split the training set into the main training set and a validation set to test the model's accuracy at each step. It ended up being about 10% of the data for the validation set and 90% of the data for the training set.

Using Aurélien Géron's CNN structure from his GitHub repository as a template, I created a CNN for this dataset. The CNN had many more parameters to set; initially setting the height and width of the image, the number of channels, and the number of inputs. The convolutional layers required the number of f maps, the filter size, the stride size, the type of padding, and an activation function to be specified. The max pooling layer required

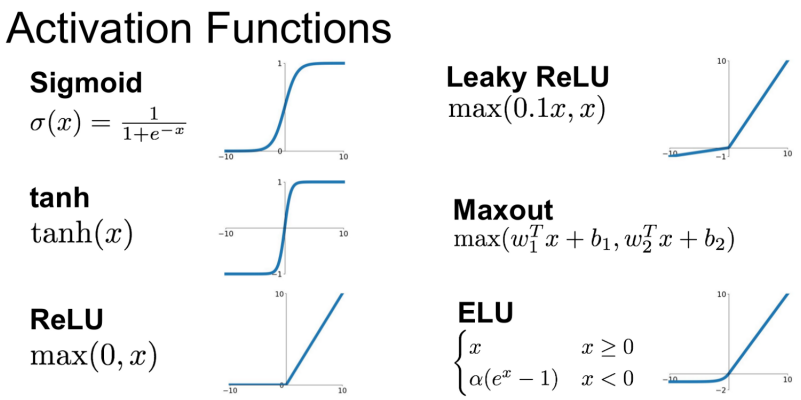

Figure 2: Some commonly used activation functions

these same parameters excluding the activation function. After that, there was a flattening layer that reshaped the data, requiring specific dimensions. The CNN is then finished off with the basic DNN structure by running the data through a dense layer, which requires an activation function (2). Then, I calculated the logits and predicted y value, ran the logits through a loss function, and optimized the loss function by using Adam Optimization. Then, I finished the CNN up by calculating the correct values and accuracy while also setting up the initialization of the TensorFlow variables and saver object.

Then, I define a function to pick our random batches from the provided dataset, a function to get the model parameters, and a function to restore the model parameters. To wrap up, I set up a session to run the CNN on the data using batch optimization, print out the accuracies, and stop training the CNN once I get a decent accuracy using early stopping. After the CNN is trained, I test the optimal parameters on a test set and print out the accuracy.
I tried to achieve a higher accuracy by modifying the stated parameters with the following tests:
Tests:
Same as Chapt 13 (fmaps: 32 64, kern: 1, stride: 1, relu, maxpool, kern: 1 2 2 1, stride: 1 1 1 1, relu, fmap: 64, poolflat, softmax, Adam):
93.24534%

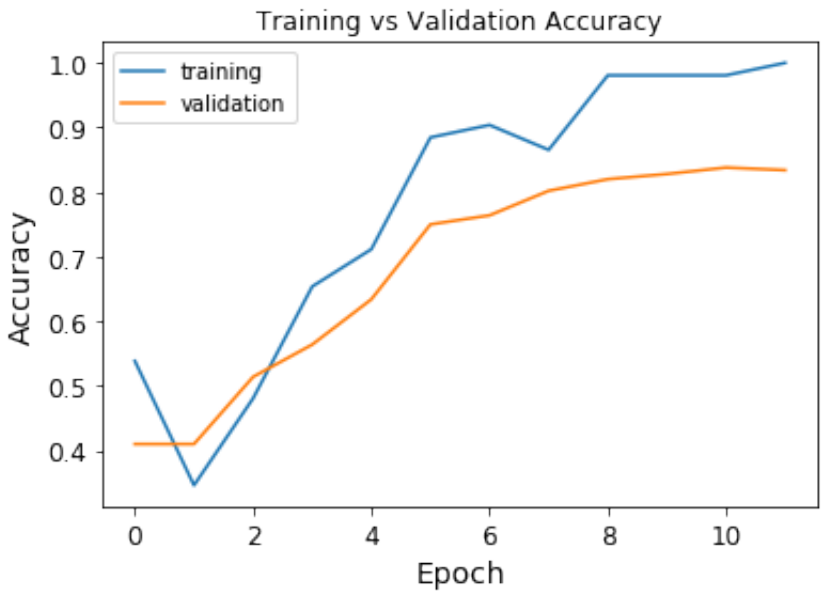

Figure 3: This graph shows the training vs validation accuracy for the test case with the highest test accuracy.

(Can't increase kern or stride o/w it throws error)

With Leaky_Relu for convolutional layers:
lower %
(I didn't record these accuracies because it took a long time to run the code and the accuracies were never very good compared to my initial accuracy)

With ELU for conv layers:
lower %

With Relu and ksize 5:
lower %

With Relu, ksize 3, and GradDesc (at various learning rates including 0.1 and 0.01):
lower %

3 Analysis

As you can see, the parameters that I tried to change in order to increase the CNN's accuracy failed to yield a better accuracy than my initial try. Most of these tests likely underfit the data; taking out information that was necessary to classify the picture properly and resulting in a low accuracy. For example, the leaky relu activation function lets the weights drop out so that the associated data is not significant to the training anymore. When the filter size increases, the data is also getting skewed because the filter is less precise with the location of the piece in the picture, i.e., the picture gets blurred. ELU usually works better than Relu, but in this case, we see that it did not. This might be because not all of the activation functions were changed and I need to do more research on that. Gradient descent was likely too slow to converge regardless of the learning rate because Adam optimization does the required calculations more efficiently (Géron).

4 Conclusion

Therefore, my attempt at producing a CNN with high accuracy when classifying the LFW dataset yielded a 93.24534% accuracy. This is a good accuracy, but the highest recorded accuracy for this dataset was 99.54% (Faceter). In order to improve my accuracy, there are several more things that I could try. I could try using batch normalization and dropout since those techniques are generally said to yield high accuracy, especially with the combination of the ELU activation function and Adam optimization. Although, dropout may cause the CNN to lose vital parts of the picture since an image of a human is more complex compared to other images, i.e., an image from the MNIST dataset. I could increase layers, try other activation functions, and try other loss functions. Even though the ones I tried did not yield better results, there is a chance that there is an odd combination that works better for this dataset. I could also try decreasing the batch size so that the runtime decreases and increasing the overall data used because if there is more data to train the CNN on, the accuracy on the test set will usually increase. In all of these cases we also keep in mind that overfitting might also become an issue. However, CNNs are very effective in classification and are a good method to solve these types of problems.

5 References

Faceter. "LFW - The Facial Recognition Worldwide Contest. Are You Better Than The Algorithm?" Medium, Faceter, 19 Nov. 2017, medium.com/faceter/lfw-the-facial-recognition-worldwide-contest-are-you-better-than-the-algorithm-316350609366.

Géron, Aurélien. "Ageron/Handson-Ml." GitHub, GitHub, github.com/ageron/handson-ml.

Géron, Aurélien. Hands-on Machine Learning with Scikit-Learn & TensorFlow. Edited by Nicole Tache, O'Reilly, 2017.

Li, Jessica. "Labelled Faces in the Wild (LFW) Dataset." Kaggle, 17 May 2018, www.kaggle.com/jessicali9530/lfw-dataset.

"The Labeled Faces in the Wild Face Recognition Dataset." Scikit Learn, scikit-learn.org/0.15/datasets/labeled_faces.html.