Jacob Wahbeh

Professor Eggert

CS 35L Lab 2

4 December 2019

Neural Networks: Learning and Evaluation

Neural networks can be extremely useful and versatile for many applications in and outside of computer science. They are a set of algorithms modeled to recognize patterns and improve with more training. They interpret data and group the unlabeled data as correct or incorrect according to the similarities decided during the testing phase. On top of that, with each iteration they improve their algorithms until they are able to make accurate predictions for most data input. Some examples that use neural networks are biological evolution simulations, maze finders, self-driving cars, medicine, and finance.

In Stanford's Computer Science 231n class, they went into detail discussing how exactly the learning and evaluation part of the neural network operates. In order to train a neural network, one must implement gradient checks with a small batch of data, and then make sure the initial loss is reasonable. Then during training, make sure to be monitoring the loss and validity accuracy and decay the learning rate as time goes on. Following, search for decent hyperparameters with random search, start with a wide range then narrow down. Finally form model ensembles for optimization.

For gradient checks, the basics regard comparing the analytical and numerical gradient, but of course, the actual process is much more tedious and trickier. By taking the absolute value of the difference of the two gradients and then dividing them by the max of the two results in the relative error. Once you have a number 10^-7 or less you should be happy with the result. There

are some nuances with this method however. Always use double precision and stay close to the active range of the floating point to minimize errors. Some functions may also have nondifferentiable aspects to them which need outside intervention to fix. This introduces methods such as ReLU or SVM loss. Additionally, only check a few dimensions a few data points. There may be many parameters and a large amount of data but using less makes kinks less likely.

After the gradient checks, it is time to start the learning process and there are many variables to monitor. One of the most important is the loss function. Having one that is too high can be chaotic and risk parameters bouncing around too much and never settling. On the other hand, having one too low would be too slow and cause linear progression. Next is the training/validation accuracy, which may cause your network to overfit or underfit. When it is too accurate, the data would be overfitted to the training data and will not generalize to the real world. When it is underfitted, the plot would produce too many false positives. Other checks include Ratio of the weights and gradient distribution per layer. Also, if you are working with pixels it might be useful to visualize the first layer to confirm smooth diverse features.

During the learning process, the parameters need to be updated as the program runs, and there are many methods to do so. The vanilla way is the simplest where it changes the parameters along the negative gradient direction. This minimizes the loss function and guarantees non-negative progress, but it can be often slow and not fully optimized. Another method is the momentum update where they treat the graph as a terrain with higher points having greater "potential energy." Here, the force vector is the negative gradient and influences the velocity, changing the position. The parameter vector will build up velocity and change more rapidly in the direction with consistent gradient. The Nesterov momentum is slightly different where it

calculates the step ahead instead of just the basic add the velocity to the position. Other more complicated approaches involve annealing the learning rate and are especially useful in deep networks.

To fully optimize the network, we have to configure the hyperparameters such as the initial learning rate, learning rate decay schedule, and regularization strength. Implementation can take many forms but a popular one is to have a ‘master’ and ‘worker.’ The worker object continuously samples random hyperparameters and performs the optimization best to its ability. The master sorts, launches, and kills workers depending on how it wants the outcome of the code. Good practice for this is to use random search and to search on a logarithmic scale. also one thing to look out for is if your parameter falls on the border of the range, you might be choosing the wrong range. While doing this process it is important to start from a wide search narrow it down as time progresses.

To evaluate, one technique involves using many independent networks and averaging their predictions, called a model ensemble, which themselves have different methods. Same model, different initializations starts with the best hyperparameters and has many different random initial values. Top models also start with the best hyperparameters, but they pick the top few to continue and form the ensemble. Different checkpoints are for when the training process is expensive, where they average at each epoch. Running average saves a second copy of the weights and maintains a decaying sum of the previous weights. The Evaluation helps increase the overall accuracy by usually a few percentage points.

Neural networks have always fascinated me, and I have always been blown away at their practical uses. From forecasting to movie recommendations, the field greatly intrigues me, and I would like to learn more about it. I find it remarkably interesting how we are using calculus and

physics in a computer science class, but the more I read about it, the more it seemed like the perfect fit. To finally piece my various classes together in computer science is quite fulfilling. I might even consider doing my masters in machine learning.

Works Cited

Stanford. “Neural Networks Learning and Evaluation.” *CS231n Convolutional Neural Networks for Visual Recognition*, cs231n.github.io/neural-networks-3/#summary.