CS 194-129 Deep Neural Networks
Spring 2018 Canny Lecture 7

1 Reading

Learning

SOURCE: CS 231N

In this section, we're concerned with learning the parameters and finding good hyperparameters.

Gradient Checks

A gradient check means comparing the analytic gradient to the numerical gradient, probably using a small batch of data. This is to ensure that our computed gradients are correct.

Sanity Checks

Before learning, it's a good idea to make sure that the loss for small chance parameters looks right, and that the network is able to overfit (to $\sim 100\%$) on a small subset of the data.

Babysitting the Learning Process

During training, it's useful to monitor *loss* (where a smaller batch size means more "wiggle" i.e. variance), *training and validation accuracy* (giving us insight about overfitting), *the ratio of update magnitudes to value magnitudes* (giving us insight about the learning rate), *activation/gradient histograms for all layers of the network* (for diagnosing bad initializations), and *first-layer visualizations* if working with image pixels. Incidentally, we should plot values in terms of epochs – where one epoch means that in expectation every example has been seen once – instead of iterations, which differs based on batch size.

Parameter Updates

We can update parameters with *SGD*, *SGD + momentum* (which integrates velocity as well as position and has improved convergence rates), or *SGD + Nesterov momentum* (which computes the gradient at the *updated* position).

It's also helpful to anneal the learning rate over time; with too high a learning rate, the parameter vector bounces around chaotically and is unable to settle down into deeper but narrower parts of the loss function. Typically, learning rate decay is implemented according to a *step decay* scheme, where we reduce the learning rate by some factor every few epochs. Note: decay too slowly and we waste computation bouncing around with little improvement for a while. Decay too aggressively and the system will cool too quickly, unable to reach the best position it can. If possible, err on the side of slower decay and train for a longer time.

We can also manipulate the learning rate on a per-parameter basis, according to *adaptive* update methods. Such methods include AdaGrad, RMSProp, and Adam.

In practice, the recommended updates to use are SGD + Nesterov momentum or Adam.

Hyperparameter Optimization

The most common hyperparameters include *initial learning rate*, *learning rate decay schedule (e.g. the decay constant)*, and *regularization strength (L2 penalty, dropout strength)*. But there are many more.

There are a number of tricks that can be applied to hyperparameter optimization. For more information, see the 231n notes. As a tl;dr, we should search for good hyperparameters using random search, staging this search from a coarse to a fine level.

Evaluation: Model Ensembles

One way to improve the performance of neural networks is to train multiple independent models, and at test time average their predictions. As the number of models in the ensemble increases, the performance typically improves monotonically (but with diminishing returns, increased evaluation time on test examples, and obviously a greater memory and computation requirement). Note that improvements tend to be more dramatic when there is higher model variety in the ensemble.

Wisdom of the crowd!

2 Lecture

Recap

Residual connections allow us to make networks more complex and much deeper.

Xavier initialization simulates random input activations and weights while avoiding vanishing and exploding gradients. At initialization time, we want approximately the same energy (magnitude) to come out of every layer as goes in, so that at least some signal propagates all the way through our (potentially very deep) network and we can have gradients flow all the way back with some reasonable magnitude.

Batch Normalization

After normalizing, we un-normalize – applying the "optimal" scaling and bias to each neuron. We learn this scaling γ and bias β (which are the same dimensions as μ and σ^2) via backprop in order to minimize the overall loss. Note that the network can simply learn the identity mapping, and maybe it should.

Normally, batch normalization comes either after or before a linear layer. Note that it is performing a scaling and bias operation on a per-pixel basis.

The advantages of BatchNorm include pseudo-random regularization (which reduces the need for dropout) and activation/gradient clipping to some extent (which allows higher learning rates).

The gradient of a weight layer is basically the product of the activation coming in from the previous layer times the gradient coming from the next layer. So when we have really big activations coming in from the previous layer, the gradients of the layer will typically be a lot larger. High activations typically cause runaway gradients. And batch normalization deals with these high activations – so by extension it deals with the runaway gradients.

As an alternative, we can clip gradients by value in order to address the big gradient problem. We can also clip gradients by norm (i.e. scale them so their overall magnitudes are not so big).

Dropout

During dropout, we randomly set some neurons to zero in the forward pass (/multiply by random Bernoulli variables with parameter p). Usually p is about 0.5. This adds a whole lot of noise, so we shouldn't use it during evaluation, but it's good in that it combats overfitting during training. It also helps the network learn diverse features that are alternative contributors to its inferences (in other words a redundant representation). Effectively, dropout trains a large ensemble of models which share parameters.

Note that dropout on its own would reduce the expected value of outputs during training time. Therefore, we also have to scale the contribution of each layer by $1/p$ while training, such that the expected output at test time is equal to the expected output at training time.

Ensemble Learning

An ensemble is a model built from many simpler models. There are two main methods, bagging and boosting. **Bagging** is when we train multiple models, and every model sees a different snapshot of the data using a bootstrap sample (with replacement). The models are trained independently of each other. **Boosting** is a sequential method that tries to do error reduction on a nested set of classifiers. There's a series of learners, and each learner tries to refine the results of the previous learner. Each learner has an additional emphasis on the "hard" examples (those that the previous learner failed to classify). With boosting, models are highly dependent on each other and have a linear ordering.

Bagging works best with models that don't have bias (training a lot of models independently reduces variance, but not bias). Also, it's helpful to train the models on different data in order to reduce any kind of common bias.

Boosting typically does better than bagging for a given complexity of model. It's able to reduce bias as well as variance (each learner after the first will see a biased sample, and make corrections on those examples where the previous learner was biased).

Bagging is often used with deep learning models; boosting is not. This is mainly because differences in deep learning models are mostly due to variance and not bias, so boosting is unnecessary in this regard. (Deep networks already correct bias by making small adjustments over a series of layers.) Also, models take a long time to train and it's nice if we can parallelize things.

We should average the predicted values from all of the ensemble models... or just vote.

Note: a cheap way to ensemble is to take a single network, share the weights of the head (i.e. the first X layers) and train multiple tails (the last Y layers) on locally bootstrapped data. *We can also train a single base model (with a fairly high learning rate) and keep taking periodic snapshots of the parameters as the model evolves.*

Gradient Noise

We can add a little noise to gradients in order to reduce overfitting in complex models. The noise should decay over time (reducing the variance as we get closer to the optimum).

Hyperparameter Optimization

Hyperparameters are typically set by a human instead of the algorithm, and include *learning rate*, *momentum decay*, *dropout rate*, and many more. When tuning hyperparameters, we want to try to evaluate the model with different settings of the hyperparameters, and then pick the settings that give the best performance. [In order to accurately measure performance, we should use cross-validation on the validation set. The variance of our evaluations will be lower if we have multiple evaluations (over different data).]

We use the validation set for hyperparameter evaluation/tuning!

As we go on, we should reduce the range of parameter values (corresponding to a coarse-to-fine search). Also, we should use random search over grid search.

References

[1] CS 231n neural network notes (pt. 3),
`http://cs231n.github.io/neural-networks-3/`