Training DNNs II

Ming-Chang Chiu
mingchac@usc.edu

1 Optimization Strategies and Meta-algorithms

1.1 Batch Normalization

Batch normalization is a method of adaptive reparametrization for almost every deep network. Batch normalization can be applied to any input or hidden layer in a network. Let $\boldsymbol{H}$ be be a minibatch of activations of the layer to normalize, arranged as a design matrix, with the activations for each example appearing in a row of the matrix. To normalize $\boldsymbol{H}$, we replaice it with

$$\boldsymbol{H}' = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{1}$$

where $\boldsymbol{\mu}$ is a vector containing the mean of each unit and $\boldsymbol{\sigma}$ is a vector containing the standard deviation of each unit. The rest of the network then operates on $\mathbf{H}$.

At training time,

$$\boldsymbol{\mu} = \frac{1}{m}\sum_i \boldsymbol{H}_{i,:} \tag{2}$$

and

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m}\sum_i (\boldsymbol{H} - \boldsymbol{\mu})_i^2} \tag{3}$$

where δ is a small positive value to avoid encountering the undefined gradient of $\sqrt{z}$ at $z = 0$

At test time, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ may be replaced by running averages that were collected during training time. So the model can be evaluated on a simgle example.

Normalizing the mean and standard deviation of a unit can reduce the expressive power of the neural network containing that unit. In order to maintain the expressive power of the network, it is common to replace the batch of hidden unit activations $\boldsymbol{H}$ with $\boldsymbol{\gamma H} + \boldsymbol{\beta}$ rather than simply the normalized $\boldsymbol{H}$.

In sum, learnable parameters are $\boldsymbol{\mu}, \boldsymbol{\sigma}, \boldsymbol{\gamma}, \boldsymbol{\beta}$, and in practice, it is recommended to normalize $\boldsymbol{XH}$ rather than $\boldsymbol{X}$.

1.2 Coordinate Descent

Coordinate Descent proceeds by optimizing one coordinate at a time. For example, we minimize $f(\boldsymbol{x})$ with respect to a single variable x_i , then minimize it with respect to another variable x_j and so on, repeatedly cycling through all variables, we are guaranteed to arrive at a (local) minimum. This practice can generalize to *block coordinate descent* where we minimize with respect to a subset of the variables simultaneously. The term “coordinate descent" is often used to refer to block coordinate descent as well as the strictly individual coordinate descent.

Coordinate descent makes the most sense when the different variables in the optimization problem can be clearly separated into groups that play relatively isolated roles, or when optimization with respect to one group of variables is significantly more efficient than optimization with respect to all of the variables.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

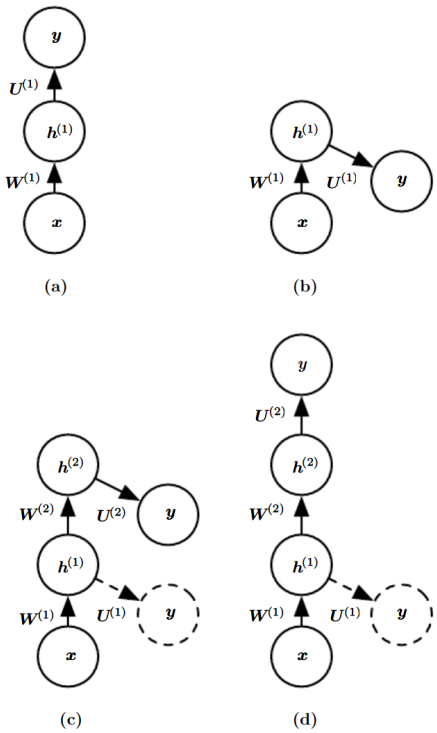

Figure 1: Illustration of one form of greedy supervised pretraining

1.3 Polyak Averaging

If t iterations of gradient descent visit points $\boldsymbol{\theta}^{(1)}, ..., \boldsymbol{\theta}^{(t)}$, then the output of the Polyak averaging algorithm is $\hat{\boldsymbol{\theta}}^{(t)} = \frac{1}{t}\sum_i \boldsymbol{\theta}^{(i)}$. The basic idea is that the optimization algorithm may leap back and forth across a valley several times without ever visiting a point near the bottom of the valley. The average of all of the locations on either side should be close to the bottom of the valley though.

In non-convex problems, it is typical to use an exponentially decaying running average:

$$\hat{\boldsymbol{\theta}}^{(t)} = \alpha\hat{\boldsymbol{\theta}}^{(t-1)} + (1-\alpha)\boldsymbol{\theta}^{(t)} \tag{4}$$

1.4 Supervised Pretraining

Pretraining generally means strategies that first train simple models on simple tasks before confronting the challenge of training the desired model to perform the desired task.

Pretraining algorithms that break supervised learning problems into other simpler supervised learning problems. This approach is known as *greedy supervised pretraining*.

Greedy supervised pretraining helps to provide better guidance to the intermediate levels of a deep hierarchy. In general, pretraining may help both in terms of optimization and in terms of generalization.

Some examples are like the original greedy supervised pretraining as in Fig. 1, *transfer learning*, *teacher-student network*

1.5 Designing Models to Aid Optimization

It is more important to choose a model family that is easy to optimize than to use a powerful optimization algorithm.

For example, using linear transformations betwen layers, using ReLU as activation so neural network is differentiable almost everywhere. These models have nice properties that make optimization easier.

Also, skip connections between layers reduce the length of the shortest path from the lower layer's parameters to the output, and thus mitigate the vanishing gradient problem.

1.6 Continuation Methdos and Curriculum Learning

Continuation methods are a family of strategies that can make optimization easier by choosing initial points to ensure that local optimization spends most of its time in well-behaved regions of space.

First construct new increasingly difficult objectives $\{J^{(0)}, ..., J^{(n)}\}$ for $J(\boldsymbol{\theta})$, with $J^{(0)}$ being the simpliest and $J^{(n)}$ the most difficult. By that we mean $J^{(i)}$ is well-behaved over more of $\boldsymbol{\theta}$ space than $J^{(i+1)}$ Continuation methods traditionally were mostly designed with the goal of overcoming the challenge of local minima by "blurring" the original cost function. This can be done by approximating

$$J^{(i)}(\boldsymbol{\theta}) = \boldsymbol{E}_{\theta' \sim N(\theta'; \theta, \sigma^{(i)2})} J(\boldsymbol{\theta}') \tag{5}$$

via sampling. The intuition for this approach is that some non-convex functions become approximately convex when blurred.

Curriculum learning is based on the idea of planning a learning process to begin by learning simple concepts and progress to learning more complex concepts that depend on these simpler concepts. Earlier $J^{(i)}$ are made easier by increasing the influence of simpler examples (either by assigning their contributions to the cost function larger coefficients, or by sampling them more frequently).

2 Practical Methodology

For day to day development of machine learning systems, the authors recommend the following practical design process

- Determine your goals—what error metric to use, and your target value for this error metric. These goals and error metrics should be driven by the problem that the application is intended to solve.
- Establish a working end-to-end pipeline as soon as possible, including the estimation of the appropriate performance metrics.
- Instrument the system well to determine bottlenecks in performance. Diagnose which components are performing worse than expected and whether it is due to overfitting, underfitting, or a defect in the data or software.
- Repeatedly make incremental changes such as gathering new data, adjusting hyperparameters, or changing algorithms, based on specific findings from your instrumentation.

2.1 Performance Metrics

What is important is to determine which performance metric to improve ahead of time, then concentrate on improving this metric. This will guide all of your future actions and you should also have an idea of what level of performance you desire.

Many metrics are possible, a common metric for classifciation is the *F-score*, where we compute

$$F = \frac{2pr}{p + r} \tag{6}$$

where p being precision, the fraction of detections reported by the model that were correct, and r being recall, the fraction of true events that were detected.

Another option is to report the total area lying beneath the PR curve, *area under curve* or *AUC*.

2.2 Default Baseline Models

In sum, get something done ASAP. After choosing performance metric, the next step is to establish a reasonable end-to-end system.

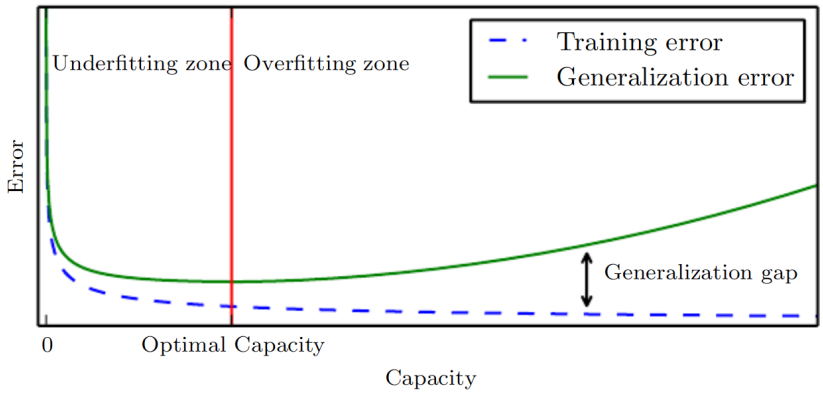

Figure 2: Typical relationship between capacity and error

Depending on the complexity of your problem, somtimes starting with statistical model like logistic regression can be a good choice. If it is an “AI-complete" task such like, object recognition, machine translation, and so on, it may be better to begin with a Deep Learning model. Some common initial setups to consider:

Activation: ReLU, Leaky ReLU
Regularization: Early stopping, dropout
Optimization: SGD with decaying learning rate, Adam, Batch normalization

If your task is similar to another task that is already studied extensively, then copy their models in your first model. For instances, in NLP document classification domain, you may want to include Hierarchical Attention Network, etc. And then modify them.

2.3 Determining Whether to Gather More Data

In Machine Learning, it is often much better to gather more data than to improve the learning algorithm.

If current performance on training set is bad: Try adding more layers, or hidden units, and tune hyperparameters. If they still do not work well, then the problem might be the **quality** of the training data. The data may be too noisy or may not include the right inputs needed to predict the desired outputs.

$\rightarrow$ collect cleaner data or a richer set of features.

If performance on training set is acceptable: If performance on test set is worse, $\rightarrow$ collect more data.

It is often recommended to gather data of size on a logarithmic scale, for example doubling the number of examples between consecutive experiments. Refer to Ch 5.2 for more detailed analysis.

2.4 Selecting Hyperparameters

2.4.1 Manual Hyperparameter Tuning

To set hyperparameters manually, one must understand the relationship between hyperparameters, training error, generalization error and computational resources (memory and runtime).

Not every hyperparameter will be able to explore the entire U-shaped curve. For instance, binary, discrete (number of units). Which can only explore some points of the curve

Learning rate perhaps is the most important hyper parameter. **If you have time to tune only one hyperparameter, tune the learning rate.**

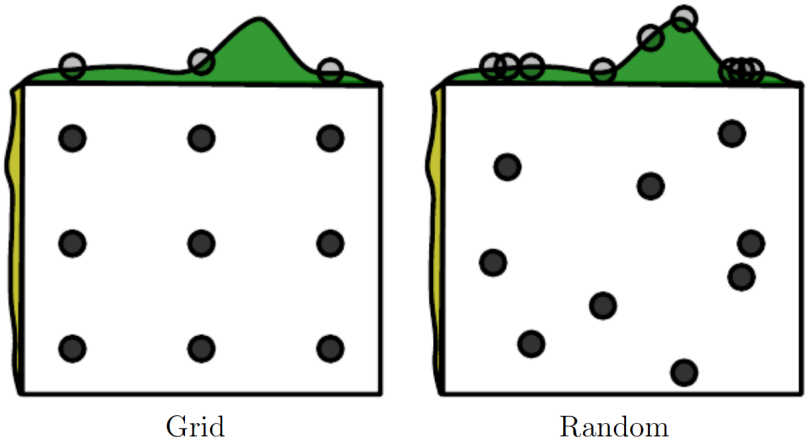

Figure 3: Comparison of grid search and random search

If your error on the training set is higher than your target error rate, you have no choice but to increase capacity.

If your error on the test set is higher than than your target error rate, you can change regularization hyperparameters to reduce effective model capacity, such as by adding dropout or weight decay.

2.4.2 Grid Search

When there are three or fewer hyperparameters, the common practice is to perform grid search. Typically, a grid search involves picking values approximately on a logarithmic scale, e.g., a learning rate taken within the set $\{.1, .01, 10^3, 10^4, 10^5\}$,

Computational cost of grid search grow exponentially. If there are m hyperparameters, each taking at most n values, then the number of training and evaluation trials required grows as $O(n^m)$.

2.4.3 Random Search

A random search starts with defining a marginal distribution for each hyperparameter, e.g., a Bernoulli or multinoulli for binary or discrete hyperparameters, or a uniform distribution on a log-scale for positive real-valued hyperparameters.

The main reason why random search finds good solutions faster than grid search is that the there are no wasted experimental runs, unlike in the case of grid search, when two values of a hyperparameter.

2.4.4 Model-Based Hyperparameter Optimization

The search for good hyperparameters can be cast as an optimization problem.

The decision variables are the hyperparameters. The cost to be optimized is the validation set error that results from training using these hyperparameters. In simplified settings where it is feasible to compute the gradient of some differentiable error measure on the validation set with respect to the hyperparameters, we can simply follow this gradient. But it is often not the case, e.g., when hyperparameter is discrete.

For deep learning, Bayesian hyperparameter optimization is normally recommended. In some cases it is comparable to human expert, but fails on others. It is worth trying.

One drawback common to most hyperparameter optimization algorithms with more sophistication than random search is that they require for a training experiment to run to completion before they are able to extract any information from the experiment. This is much less efficient, in the sense of how much information can be gleaned early in an experiment, than manual search by a human practitioner, since one can usually tell early on if some set of hyperparameters is completely pathological.

2.5 Debugging Strategies

It is sometimes hard to tell if it is intrinsic to the algorithm or there is a bug in implementation when performance goes wrong. Either we design a case that is so simple that the correct behavior actually can be predicted, or we design a test that exercises one part of the neural net implementation in isolation. Some important debugging tests include:

Visualize the model in action: It may be harder for NLP, but for Computer Vision, when training an object detection model, it is helpful to view deirectly the bounding boxes.

Reasoning about software using train error: Usually if one cannot train a classifier to correctly label a single example, an autoencoder to successfully reproduce a single example with high fidelity, or a generative model to consistently emit samples resembling a single example, there is a software defect.

Monitor histograms of activations and gradient: It is often useful to visualize statistics of neural network activations and gradients. The pre-activation value of hidden units can tell us if the units saturate (take tanh for example). Also, optimization is impeded when gradients are growing too fast or vanishing too quickly. Finally, it is useful to compare the magnitude of parameter gradients to the magnitude of the parameters themselves. We would like the magnitude of parameter updates over a minibatch to represent something like 1% of the magnitude of the parameter (so it does not move too fast or slowly).

References

[1] Zaremba, Wojciech and Ilya Sutskever. “Learning to Execute.” ArXiv abs/1410.4615 (2015): n. pag.