Neural Network Optimization and Tuning

11-785 / Spring 2018 / Recitation 3

Logistics

- You will work through a Jupyter notebook that contains sample and starter code with explanations and comments throughout. Hopefully this will help you learn more quickly than us discussing code on a lecture slide!
- Please follow https://github.com/cmudeeplearning11785/deep-learning-tutorials to download the notebook and relevant dependencies for today's recitation.
- Once you have the notebook ready to go, look back up to the front so that we can start the presentation and help guide you through the material in the notebook.
- If you have any questions about getting Jupyter up and running (or any other course dependencies) please ask one of us now or post on Piazza.

From GD to SGD

- The back-propagation algorithm computes the gradient of the error w.r.t to our parameters.
- The average of these gradients across our training set with *n* examples is known as the "true gradient".
- It is often impractical to compute the true gradient for each step of gradient descent.
- Instead we will partition the training set into equivalently sized "mini-batches" of size *m*.
- Less noisy than a single example and also more efficient with modern parallel computation (GPU).

$$\hat{y}_i = F(x_i; \theta)$$

$$g = \frac{1}{n} \sum_{i=0}^{n} \nabla_\theta L(\hat{y}_i, y_i)$$

$$\hat{g} = \frac{1}{m} \sum_{i=0}^{m} \nabla_\theta L(\hat{y}_i, y_i)$$

SGD with Mini-Batches in PyTorch

- It is standard convention in PyTorch and other tensor frameworks to treat the first dimension of tensor as the "batch dimension."
- Remember that a neural network is a series of linear transformations with each one followed by a non-linearity applied element wise.
- It is trivial to type check these transformations (matrix multiplications) to see that the batch dimension is preserved through the computation.
- As such, the final output of a network performing classification should be *m x c*, where m is the batch size and c is the # of categories.

```
>>> torch.randn(100, 784)

-1.1014e+00 -1.6781e+00 -1.0974e+00
-1.2826e-01  7.2327e-01  1.0409e+00
 8.0002e-01 -1.4476e-01  9.7467e-01
                ...
-8.4452e-01 -1.9254e+00 -1.5217e+00
 3.0576e-01  8.3397e-01 -8.5753e-01
-8.7783e-01 -4.1706e-01  1.1486e-01
[torch.FloatTensor of size 100x784]
```

A batch of gaussian noise that you can feed into your MNIST network to type check.

Vanilla SGD in PyTorch

$$\theta' \leftarrow \theta - \eta \nabla_\theta \sum_{i=0}^{m} L(\hat{y}_i, y_i)$$

- Once gradients are calculated across a batch and averaged, we can take a step in the opposite direction.
- Because (stochastic) gradient descent is a first order method, we only have a linear approximation of the true error surface nearest the current position in parameter space.
- We can take small steps (scale the gradient) so we don't overshoot. If our steps are too small, we learn very slowly.
- The step size, aka learning rate, is a hyper-parameter that remains fixed throughout training in the simplest (vanilla) form of SGD. If this sounds naïve, why?
- Food for thought: Does it matter whether we average the gradients of the batch in each step? What would change if instead we summed them?

NOTEBOOK SECTION TO FOLLOW: **A Simple Optimizer in PyTorch**

Parameter Initialization

- Random
- Xavier/Glorot
- Pretraining
- Custom

Random Initialization

- We cannot start with equal initializations (why?)
- We don't want to start at saturated values, so we should sample between -1 and 1
- Okay for smaller networks, but not really a serious candidate.

NOTEBOOK SECTION TO FOLLOW: **Parameter initialization**

Xavier Initialization

- We want the variance of the inputs to equal the variance of outputs
- If we have lesser number of inputs, we must multiply them with larger weights to get the same variance as more number of inputs with smaller weights
- Sample from Normal(0,*std*)

$$std = gain \times \sqrt{2/(fan\_in + fan\_out)}$$

- Sample from Uniform(-*a*,*a*)

$$a = gain \times \sqrt{2/(fan\_in + fan\_out)} \times \sqrt{3}$$

- Assumption - linear activations. Breaks with ReLU

NOTEBOOK SECTION TO FOLLOW: **Your turn!**

Understanding the difficulty of training deep feedforward neural networks" - Glorot, X. & Bengio, Y.

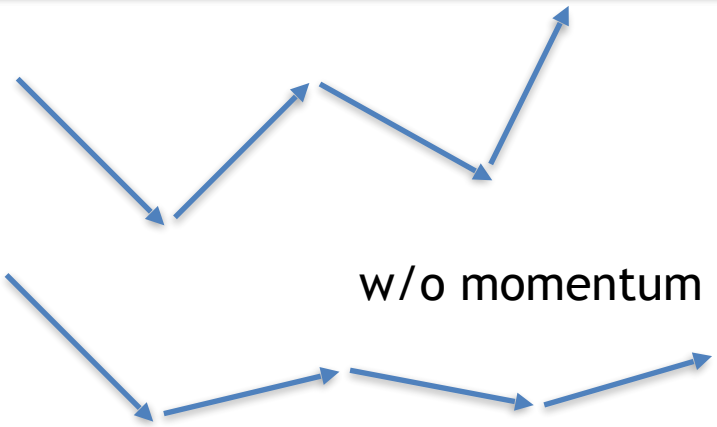
Pretraining

- Train on a large dataset and use the weights learnt as the initialization.
- Weights learnt for a task can give the network a head start for related tasks.
- Makes most sense when you have two related tasks/datasets

Why Does Unsupervised Pre-training Help Deep Learning? Erhan, Bengio, Bengio

Momentum in PyTorch

- Poorly conditioned vs well conditioned problem. This can be quantified.
- Maybe we can resist sudden changes in the gradient and smooth them along our prior trajectory through the parameter space.
- Let's use an approximation of the running average of prior gradients to update the parameters.
- We can keep track of this quantity and update in constant time at each update step.



w/o momentum

w/ momentum

$$\phi' \leftarrow \mu\phi - \eta\nabla_\theta \sum_{i=0}^{m} L(\hat{y}_i, y_i)$$

$$\theta' \leftarrow \theta + \phi'$$

NOTEBOOK SECTION TO FOLLOW: **Adding Momentum**

rprop: Works with Full-Batch

- So far we have used a global learning rate, but the magnitude of the gradient can vary greatly for different rates.
- Consider vanishing gradient and the issue with top-down learning.
- It makes sense to have an adaptive learning rate for each parameter. How can we do this effectively?
- For full batch learning (calculating the gradient over the whole training set), we can try to adaptively change the learning rate for each parameter.
- This can be done by scaling the gradient terms of a batch to be larger when the signs of the terms of the last 2 gradients agree, and smaller if not.

rprop: FAILS with Mini-Batches

Let's pick a simple example of a single weight *w* being updated by a series of mini-batch gradients. Say 1000 training examples with a batch size of 100, so there are 10 updates in total.

Parameter updates over epoch:

+0.1 +0.1 +0.1 +0.1 +0.1 +0.1 +0.1 +0.1 +0.1 -0.9

In the case of full-batch training, we would expect there to be a negligible change to the parameter.

In the case of the mini-batch training, we can see that the learning rate would grow after the 3rd step until the end of the epoch. Unexpectedly, the weight would grow massively instead of remaining steady.

What if we could combine the effectiveness of the rprop with the benefits of mini-batch training?

rmsprop

To address the issue of using rprop with mini-batch training, Geoff Hinton's research group informally came up with the idea to ensure that the gradients from adjacent mini-batches are scaled similarly.

Instead of scaling based on changes in direction, scale based on **a running average of the magnitude of the (second moment of the) gradient.**

If the running average grows, we scale the gradient down.

If the running average shrinks, we scale the gradient up.

Effectively, rmsprop adaptively scales the gradient to reduce the chance of over-shooting a minimum.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

The ADAM Optimizer

- Adaptive moment estimation for each parameter
- RMSprop stores exponentially decaying average of past *squared* gradients

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

The ADAM Optimizer

- Adaptive moment estimation for each parameter
- RMSprop stores exponentially decaying average of past *squared* gradients
- Adam adds exponentially decaying average of past gradients

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

The ADAM Optimizer

- Adaptive moment estimation for each parameter
- RMSprop stores exponentially decaying average of past *squared* gradients
- Adam adds exponentially decaying average of past gradients

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

The ADAM Optimizer

- Adaptive moment estimation for each parameter
- RMSprop stores exponentially decaying average of past *squared* gradients
- Adam adds exponentially decaying average of past gradients

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t .$$

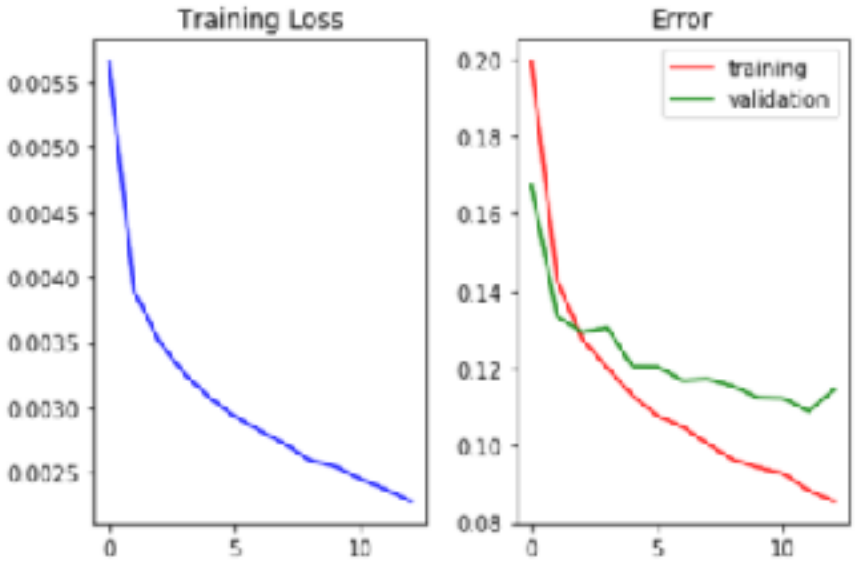
Tracking Metrics

- In order to compare the utility of different architectures or hyper parameter settings, it is important to track certain track certain training metrics.
- Start with these:
  - Loss per epoch -> cheap
  - Training error -> expensive but can sidestep
  - Validation error -> moderate depending on training split
- It is helpful to define a class or other data structure to keep track of this data.
- Visualizations are critical. Learning Matplotlib and other visualization tools like Visdom is critical.

insanity | in'sanədē |

noun

"Doing the same thing repeatedly and expecting a different result."



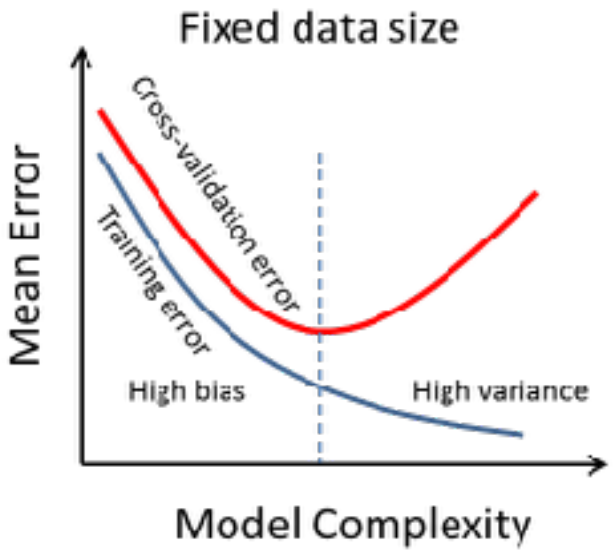
Early Stopping

- Sometimes we can use these metrics online to adjust training.
- How can we keep the model with best validation performance? Stop training as soon as validation error increases (maybe for a little while after?).
- Theoretical limits exist by estimating generalization error vs epoch #, but an easy practical solution is to simply track performance on validation set.
- Practical considerations (all of which add a parameter(s)):
  - Initial few epochs should not be part of early stopping.
  - Patience: number of epochs to wait before stopping if no progress
  - Validation frequency
  - Cumulative one-step difference over a specified number of steps is positive.

Cross-Validation

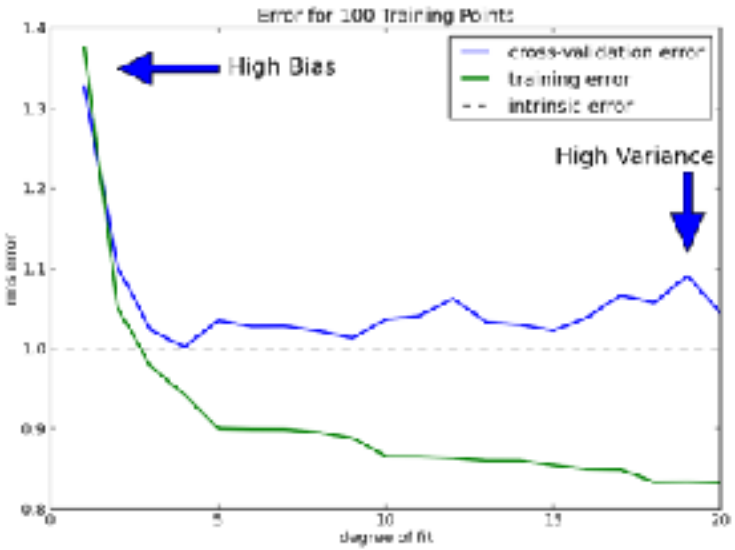
- How do we evaluate the performance of our model on unseen data?
- Given some training data, we can *emulate* unseen data by partitioning it into a train set and a validation set.
- Typically use *k*-fold cross-validation by randomly dividing the data into k partitions and using one as a validation set. Metrics are averaged over the folds.
- Often in practice it is easier and performs decently to hold out a single validation set. One heuristic is to split the training set 80/20 into a separate training and validation set.

Cross-Validation



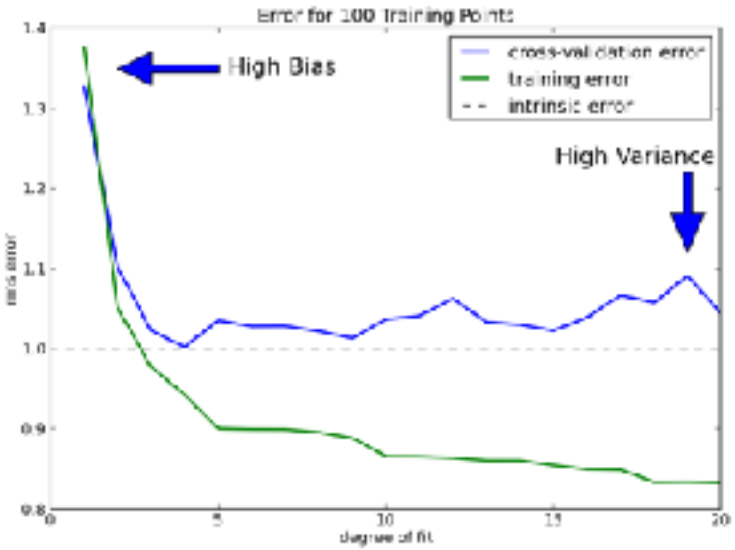
- Overfitting is easy to recognize with idealized curves.

Cross-Validation



- Overfitting is easy to recognize with idealized curves.
- In most cases, such conclusions are not so straightforward.

Cross-Validation



- Overfitting is easy to recognize with idealized curves.
- In most cases, such conclusions are not so straightforward.
- Identifying the point after which network starts overfitting is of great interest – we can prevent overfitting, and potentially speed up training by stopping before convergence.

Hyperparameter Tuning

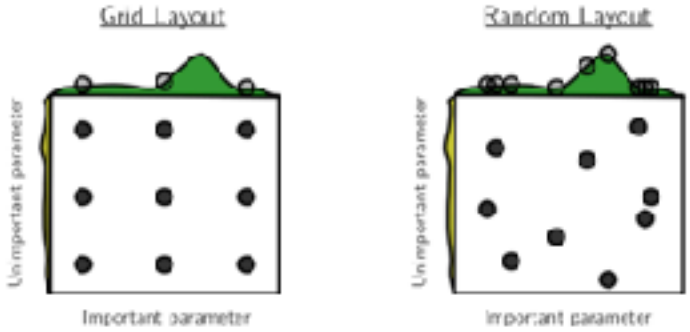
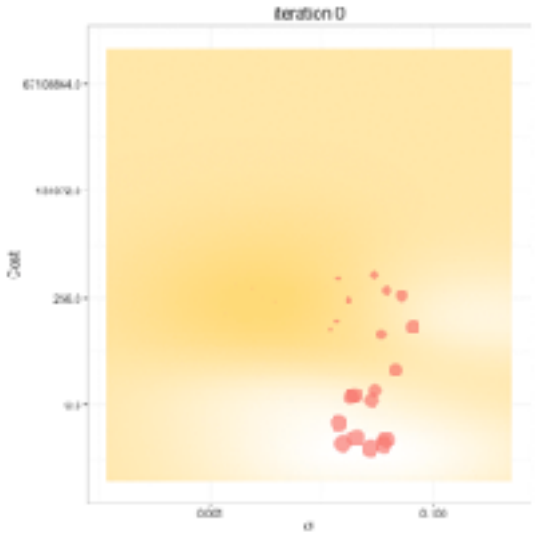


Figure 1: Grid and random search of nine trials for optimizing a function $f(x, y) = g(x) + h(y) \approx g(x)$ with low effective dimensionality. Above each square $g(x)$ is shown in green, and left of each square $h(y)$ is shown in yellow. With grid search, nine trials only test $g(x)$ in three distinct places. With random search, all nine trials explore distinct values of g. This failure of grid search is the rule rather than the exception in high dimensional hyper-parameter optimization.

- Grid search
  - Can test many parameters independently – parallelizable
  - Only goes through a manually defined subset of hyperparameter space
- Random search
  - Has been shown to be more effective than grid search.
- Bayesian
  - Assume a smooth function that maps hyperparameters to objective function.

James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* 13 (February 2012), 281-305.

Hyperparameter Tuning



- Grid search
- Random search
- Bayesian

Assume a smooth function maps hyperparameters to the objective function. In this case, sigma is the radial basis kernel parameter in an SVM model mapping hyperparameters to cost.

Explore hyperparameter space by predicting good tuning values

Regularization

- Neural networks have amazing representational power.
- More parameters -> more representational ability -> higher model variance.
- The network should learn "regular" patterns in the data to perform better on the objective, not memorize peculiarities of training examples.
- Can be of use when model performs well on training data, poorly on validation data.
- Many approaches, each attempts to motivate generalization and/or penalize overfitting by the model.
- It is typical to add a hyper parameter to adjust the strength of regularization relative to the overall objective.

"Keep it simple, no more assumptions should be made in explaining something than are necessary" — Occam's Razor

$$g = \frac{1}{n} \sum_{i=0}^{n} \nabla_\theta L(\hat{y}_i, y_i) + R(\theta)$$

L1 & L2 Regularization

- L1 Regularization
  - Penalizes the number of non-zero parameters.
  - Motivates sparse representations.

$$R(\theta) = \frac{1}{m} \sum_{i}^{m} |\theta_i|$$

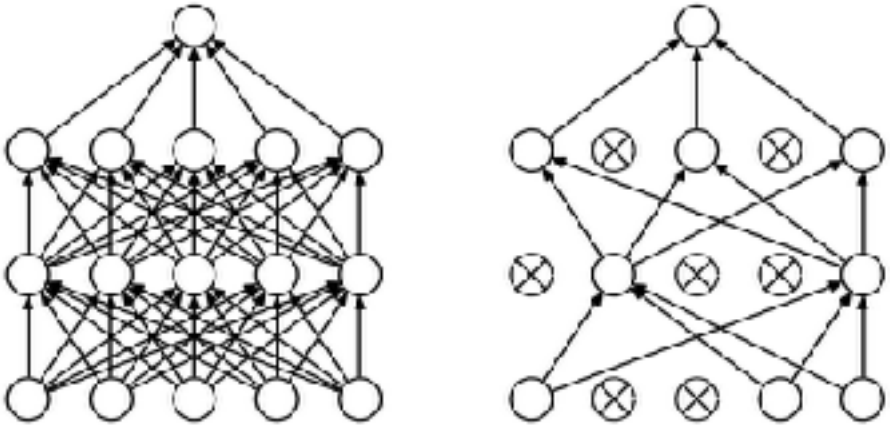
- L2 Regularization
  - Penalizes large weights.
  - Motivates small weights.

$$R(\theta) = \frac{1}{m} \sum_{i}^{m} \theta_i^2$$

Random Dropout

Implementation

- Dropout each parameter with probability *p* (considerations for input?)
- All parameters not dropped at test time. How to compensate?



(a) Standard Neural Net (b) After applying dropout.

Results

- Network is forced to learn a distributed representation.
- Somewhat equivalent to paying a sparsity penalty.
- How does dropout improve generalization?
- Averaging exponentially many different topologies. Akin to model combination (ensemble) methods.

Batch Normalization

- Covariate shift is a phenomenon in which the covariate distribution is non-stationary over the course of training. This is a common phenomenon in online learning.
- When training a neural network on a fixed, standardized dataset, there is no covariate shift, but the distribution of individual node and layer activity shifts as the network parameters are updated.
- Consider each node's activity to be a covariate of the downstream nodes in the network. Think of the non-stationarity of node (and layer) activations as a sort of **internal covariate shift**.
- Why is internal covariate shift a problem? Each subsequent layer has to account for a shifting distribution of its inputs. For saturating non-linearities the problem becomes even more dire, as the shift in activity will more likely place the unit output in the saturated region.
- Prior to batch-norm, clever initialization was needed to mitigate some of these problems.
- Solution: Adaptively normalize each unit’s activity across a batch. Why adaptive? Because the network should have access to the raw unnormalized output if that is advantageous.

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Algorithm is amazingly straight-forward.
- Calculate mean and variance of the mini-batch and standardize.
- Use learned BN parameters to adjust the estimator.
- Which setting of the parameters recovers the original activity?