Neural Network Optimization and Tuning

11-785 / Fall 2018 / Recitation 3

Logistics

Please follow
"https://github.com/cmudeeplearning11785/Fall2018-tutorials"
to download the notebook for today's recitation notebook.

SGD

- The average of the gradients across our training set with "***n***" examples is known as the "**true gradient**".

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

- This is impractical.
  - Time and Space constraints

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

- Instead, partition the training set into equivalently sized "**mini-batches**" of size "***m***".
  - Not "true gradient" but a stochastic estimate

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

- Less noisy than a single example and also more efficient with modern parallel computation techniques (GPU being one).

SGD

$$\theta' = \theta - \eta.\hat{g}$$

- Once we get the (stochastic) gradient estimates, we start taking small steps in the opposite direction (descent not ascent).
    - Small Steps ensure that we don't overshoot. "**eta**" = "**Learning Rate**".
- Fixed "eta" in vanilla SGD.
    - Is it good?
    - Will discuss other options later in the recitation

- Random thought:
    - Does it matter if we sum the gradients of a batch instead of averaging them in each step?

RProp

- Resilient Backpropagation
- Problem
    - Gradients we calculate are multi-dimensional.
        - Depends on the number of neurons in the layer
        - But we apply same learning rate to all the parameters
        - Gradients can be sparse ("**Vanishing Gradients**")
            - Requires us to have an adaptive learning rate for each parameter.
- If we are doing full-batch training,
    - We know the "**true gradient**".
    - We can try to adaptively change the learning rate for each parameter.
- **IF** sign of the last two gradients agree, LR increases (scales up) **ELSE** LR decreases.
- What happens when we don't have true gradients?

RProp

- An example of RProp failing 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.

RMSProp

- To address the issue of using Rprop with mini-batch training, we can 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.

- RMSProp adaptively scales the gradient to reduce the chance of over-shooting a minimum.

$$\theta' = \theta - \frac{\eta}{\sqrt{E[\hat{g}^2] + \epsilon}} \hat{g}$$

Adam

- **Ada**ptive **m**oment estimation. Combines advantages of RMSProp + AdaGrad (left for self-exploration).
- One of the most widely used Optimizers.
- In addition to the first moments used in rmsprop, Adam also uses second moments.
- Default : Initial Learning Rate of *1e-3*, *beta1 = 9e-1* and *beta2 = 999e-3*

$$t := t + 1$$

$$lr_t := extlearning_rate * \sqrt{(1 - beta_2^t)/(1 - beta_1^t)}$$

$$m_t := beta_1 * m_{t-1} + (1 - beta_1) * g$$

$$v_t := beta_2 * v_{t-1} + (1 - beta_2) * g * g$$

$$variable := variable - lr_t * m_t/(\sqrt{v_t} + \epsilon)$$

Hyperparameter Tuning

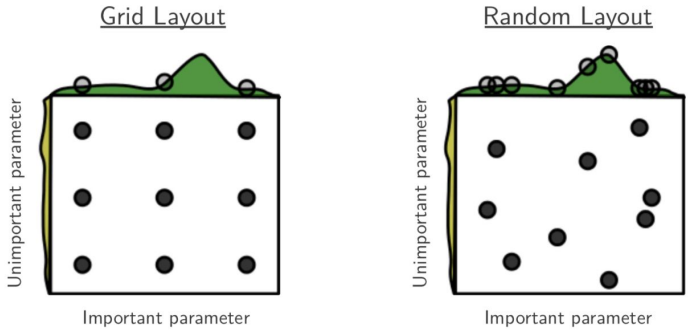


Figure 1: Grid and random search of nine trials for optimizing a function $f(x,y) = g(x) + h(y) \approx g(x)$ with low effective dimensionality. Above each square $g(x)$ is shown in green, and left of each square $h(y)$ is shown in yellow. With grid search, nine trials only test $g(x)$ in three distinct 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

  The Good - Can test many parameters independently – parallelizable

  The Bad - Only goes through a manually defined subset of hyperparameter space
- Random search

  Similar to Random Search - has been shown to be more effective than grid search.
- Bayesian

  Assume a smooth function that maps hyperparameters to objective function and minimize that using Bayesian Regression/other techniques.

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

Parameter Initialization

- Can we start with zero initial weights?
- Can we have equal initial weights?
- Methods to initialize
  - Random (typically gaussian)
  - Xavier
  - Pretraining
  - Implementing custom initialization

Xavier Initialization

- You want the variance of Input and Output to be same.

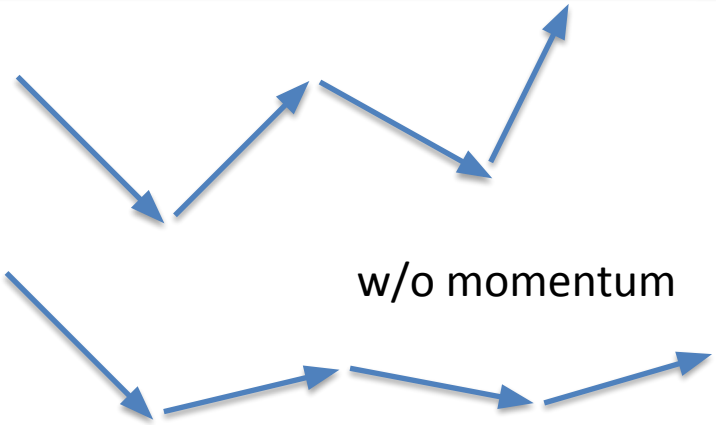
$$Y = \sum_{i=0}^{n} W_i X_i$$

- If you work out the math, Var(W) = 1/n
- But NNs have multiple inputs and multiple outputs

$$\sigma = G * \sqrt{\frac{2}{fan\_in + fan\_out}}$$

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

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

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

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.

L1 & L2 Regularization

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

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

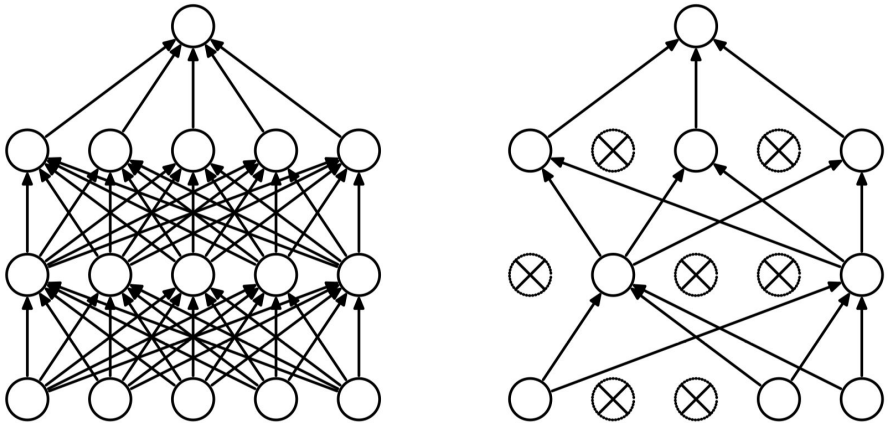
- L2 Regularization
  - Penalizes large weights.
  - Motivates small weights.

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

Random Dropout

Implementation

- Dropout each parameter with probability *p*
- 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: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

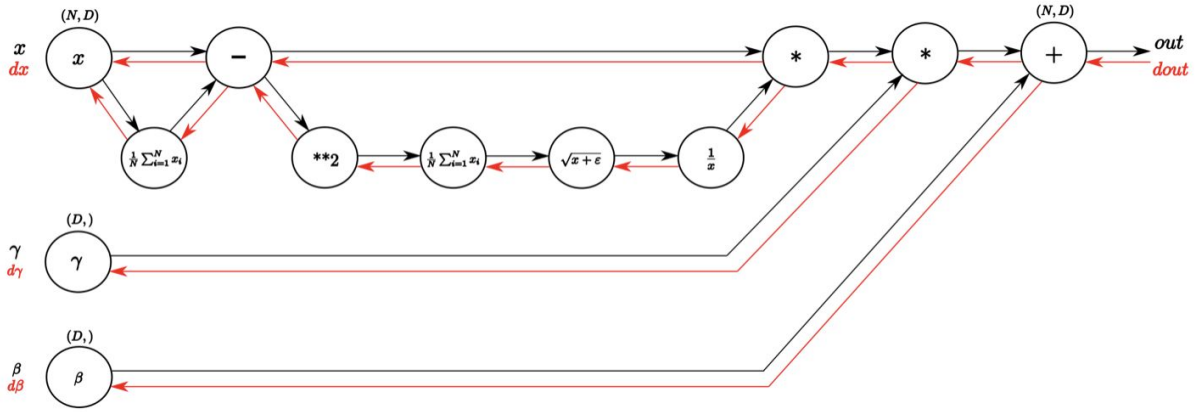
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

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

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

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

- 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?

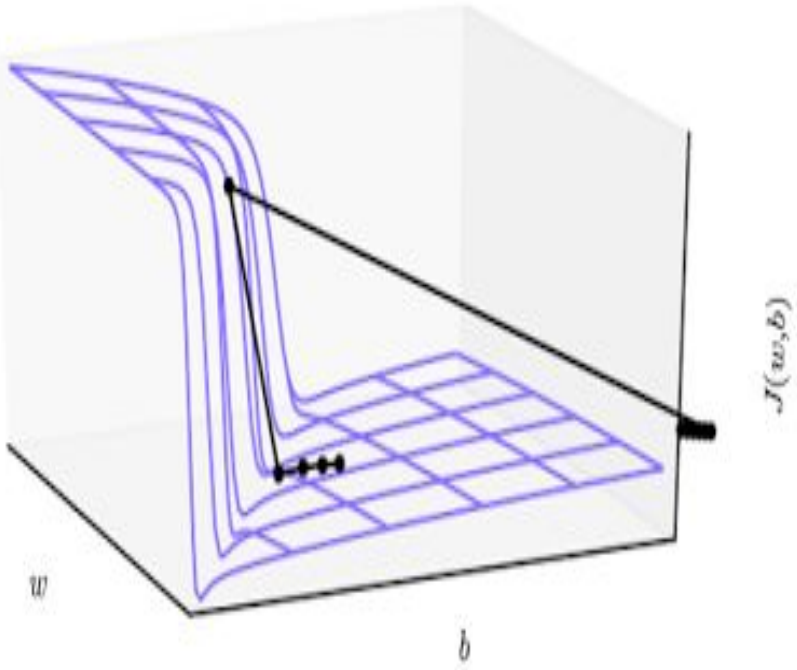


Gradient Clipping

During training, sometimes the gradient value grows extremely large, it causes an overflow which is easily detectable at runtime or in a less extreme situation, the Model starts overshooting past our Minima; this issue is called the **Gradient Explosion Problem**.

Gradient clipping will 'clip' the gradients or cap them to a Threshold value to prevent the gradients from getting too large.

In the above image, Gradient is clipped from Overshooting and our cost function follows the Dotted values rather than its original trajectory.



Annealing the learning rate

- Usually helpful to anneal the learning rate over time
- High learning rates can cause the parameter vector to bounce around chaotically, unable to settle down into deeper, but narrower parts of the loss function.
  - **Step decay**: Reduce the learning rate by some factor every few epochs. Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs. One heuristic you may see in practice is to watch the validation error while training with a fixed learning rate, and reduce the learning rate by a constant whenever the validation error stops improving.
  - **Exponential decay** : It has the mathematical form `lr = lr0 * e^(-kt)`, where `lr`, `k` are hyperparameters and `t` is the iteration number.