Lecture 4: Optimization for Training Deep Models - Algorithms

Ali Harakeh

University of Waterloo

WAVE Lab
ali.harakeh@uwaterloo.ca

June 13, 2017

Overview

1. Parameter Initialization Strategies
2. First Order Optimization Algorithms
3. Optimization Strategies And Meta-Algorithms
4. Conclusion: Designing Models to Aid Optimization

Section 1

Parameter Initialization Strategies

Introduction

- Training deep learning algorithms is a non-convex problem that is iterative in nature, and thus requires us to specify initial points.
- The effect of the choice of the initial point in deep learning is very important.
- The initial point can effect the **speed** of convergence, the **quality** of the final solution, and if the algorithm **converges** all together.
- The major, most important observation that I want you all to remember is the following: **points of comparable cost will have a different generalization error !**

Good Characteristics Of Initial Parameters

- Modern initialization strategies are simple and heuristic.
- These strategies are designed to achieve some "nice" properties when the network is initialized.

Good Characteristics Of Initial Parameters

- However, we do not have a good understanding of which of these properties remain after the first iteration of training.
- Furthermore, what is beneficial for optimization can be detrimental to learning and generalization.
- Basically, we have very primitive understanding on the effect of the initial point on generalization, which offers no guidance in the selection procedure.

Breaking The Weight Space Symmetry

- One of the properties we are certain of is that initial parameters should break the symmetry.
- If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters.
- This property prevents both the loss of input patterns in the null space of the forward pass, and the loss of gradient patterns in the null space of the backward pass.
- The above motivates **random initialization**.

Random Initialization

- We could explicitly search for a large set of basis functions that are all mutually different from each other. However, the computation cost of this method outweighs the benefits that it provides.
- Example: If we have at most as many outputs as inputs, we could use Gram-Schmidt orthogonalization on an initial weight matrix, and be guaranteed that each unit computes a very different function from each other unit.
- **Random Initialization** from a high entropy distribution over a high dimensional space is much computationally cheaper, while resulting in almost the same symmetry breaking effect.

Random Initialization

- Random Initialization is performed by setting the **biases** to a constant (0 or 0.01 in most cases).
- Weights are initialized by sampling from either a Gaussian or a uniform distribution (the choice doesn't seem to matter much).
- The scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and on the ability of the network to generalize.

Random Initialization

- Larger initial weights will yield a stronger symmetry breaking effect, helping to avoid redundant units. (Also prevents loss of signal in forward pass)
- However, initial weights that are too large may result in exploding values during forward propagation or back-propagation. (**choas** in RNNs)
- The scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and on the ability of the network to generalize.

Normalized (Xavier) Initialization

- Proposed by Xavier Glorot and Yoshua Bengio in 2010.
- Validated by He et al. for ReLU layers in 2015.
- He et al. in their paper titled: "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification" derived an initialization specific to ReLU layers, where the weights are initialized as:

$$\mathbf{w} = \mathcal{N}(0,1)\sqrt{\frac{2}{n}}$$

- A rule of the thumb is to always use xavier initialization when training ReLU based networks from scratch.

Section 2

First Order Optimization Algorithms

Stochastic Gradient Descent

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

Stochastic Gradient Descent

- Stochastic Gradient Decent (SGD) and its variants are probably the most used optimization techniques for deep model training.
- The learning rate ϵ is an essential parameter. In practice, it is necessary to gradually decrease the learning rate over time.
- This is because SGD gradient estimation introduces a source of noise, caused by the random minibatch sampling.
- However, the true gradient becomes closer to 0 as we converge to a minimum.

Stochastic Gradient Descent

- The *sufficient* conditions for guaranteed convergence of SGD is that:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty$$
$$\sum_{k=1}^{\infty} \epsilon_k^2 < 0$$

Stochastic Gradient Descent

- In practice, it is common to decay the learning rate linearly until iteration τ according to:

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau$$

- $\alpha = \frac{k}{\tau}$.
- After iteration τ we keep ϵ constant.

Stochastic Gradient Descent

- Choosing ϵ_τ, ϵ_0 and τ is more of an art than a science.
- This is done by monitoring the learning curves that plot the objective functions as a function of time.
- Large oscillations implies one is using a large ϵ_0.
- Gentle oscillations are fine, especially if we are using a stochastic cost functions.

Stochastic Gradient Descent

- Typically, the optimal initial learning rate, in terms of total training time and the final cost value, is higher than the learning rate that yields the best performance after the first 100 iterations or so.
- It is usually best to monitor the first several iterations and use a learning rate that is higher than the best-performing learning rate at this time, but not so high that it causes severe instability
- τ is chosen as the number of iterations it takes for the algorithm to go through a few hundred passes through the training set. ϵ_τ is chosen to be approximately 1% of ϵ_0.

Momentum

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Momentum

- Momentum tries to remedy the slowness of SGD especially in face of high curvature, small but consistent gradients, or noisy gradients. (Ill conditioning !)
- The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.
- Analogous to rolling a ball with mass and gravity on the topology of the objective function.
- $\alpha \in [0, 1]$ is a hyperparameter that determines how quickly the contributions of previous gradients exponentially decay. In practice, α is set to be 0.5, 0.9 and 0.99.

Nestrov Momentum

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Nestrov Momentum

- The difference between Nesterov momentum and standard momentum is where the gradient is evaluated.
- With Nesterov momentum the gradient is evaluated after the current velocity is applied.
- Thus one can interpret Nesterov momentum as attempting to add a correction factor to the standard method of momentum.

AdaGrad

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

AdaGrad

- AdaGrad individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values.
- The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivative shave a relatively small decrease in their learning rate.
- The net effect is greater progress in the more gently sloped directions of parameter space.

AdaGrad

- AdaGrad performs well for some but not all deep learning models.
- Empirically it has been found that for training deep neural network models the accumulation of squared gradients from the beginning of training can result in a premature and excessive decrease in the effective learning rate.

AdaGrad

- AdaGrad is designed to converge rapidly when applied to a convex function. When applied to a non-convex function to train a neural network, the learning trajectory may pass through many different structures and eventually arrive at a region that is a locally convex bowl.
- AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure.

RMSProp

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
    Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
  end while

RMSProp

- RMSProp modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average.
- RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.
- Empirically, RMSProp has been shown to be an eïňĂective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods being employed routinely by deep learning practitioners.

Adam

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$

Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
Initialize time step $t = 0$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  $t \leftarrow t + 1$
  Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$
  Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$
  Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
  Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
  Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

Adam

- Adam (adaptive moments) is a variant of RMS prop and momentum with a few important distinctions.
- First, in Adam, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient. (No theoretical motivation !)
- Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin.
- Conclusion: Always use Adam, it is fairly robust to the choices of hyperparameters and available in many deep learning packages.

Section 3

Optimization Strategies And Meta-Algorithms

Batch Normalization

- **Batch Normalization**(Ioffe and Szegedy, 2015) is one of the most exciting innovations in optimizing neural networks.
- It is not an optimization algorithm, but a method of **adaptive reparameterization**.
- Motivated by the difficulty of training very deep models.

Batch Normalization

- Training a deep model involves parameter updates for each layer via gradient direction under the assumptions that other layers are not changing.
- In practice, all layers are updated simultaneously.
- This can cause unexpected results in optimization. Example ?

Batch Normalization

- It is very hard to choose an appropriate learning rate, because the effects of an update to the parameters for one layer depends so strongly on all of the other layers.
- Second order optimization methods tries to remedy this phenomenon by taking into account second order effects. However, in very deep networks, the effects of higher order effects is very prominent.
- Solution: Build an n-th order optimization algorithm !

Batch Normalization

- Off course not !
- Batch normalization provides an elegant way of reparametrizing almost any deep network.
- It can be applied to any layer, and the reparametrization significantly reduces the problem of coordinating updates across many layers.

Batch Normalization

- Let $\mathbf{X}$ be a minibatch output of the layer we would like to normalize.
- Batch normalization operates according to the following formula:

$$\mathbf{X} \leftarrow \frac{\mathbf{X} - \mu}{\sigma}$$

- μ is a vector containing the mean of each unit, σ is a vector of standard deviations for each unit. These vectors are **broadcasted** i.e. normalization occurs row wise.
- The rest of the network operates on $\mathbf{X}$ as usual.

Batch Normalization

- At training time:

$$\mu = \frac{1}{m} \sum \mathbf{X}_{i,:}$$

$$\sigma = \sqrt{\delta + \frac{1}{m} \sum (\mathbf{X} - \mu)_i^2}$$

- We can **back-propagate** through these operations!

Batch Normalization

- This means that the gradient will never propose an operation that acts simply to increase the standard deviation or mean of $\mathbf{x}_i$; the normalization operations remove the effect of such an action and zero out its component in the gradient.
- At test time, μ and σ are replaced by a moving average of the mean and standard deviation that was collected during training.

Batch Normalization

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

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

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

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

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

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

Batch Normalization: Conclusion

- Improves gradient flow through the network.
- Allows higher learning rates.
- Reduces the strong dependence on initialization.
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe ?

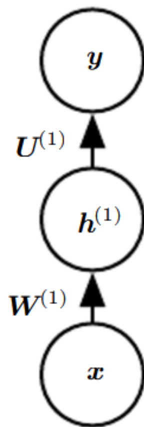
Greedy Supervised Pretraining

- Sometimes, directly training a model to solve a specific task can be too ambitious if the model is complex and hard to optimize or if the task is very difficult.
- It can also be more effective to train the model to solve a simpler task, then move on to confront the final task.
- Greedy algorithms break a problem into many components, then solve for the optimal version of each component in isolation.
- Unfortunately, combining the individually optimal components is not guaranteed to yield an optimal complete solution.

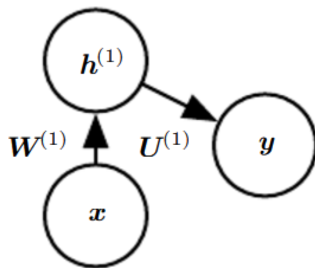
Greedy Supervised Pretraining

- Greedy algorithms can be computationally much cheaper than algorithms that solve for the best joint solution, and the quality of a greedy solution is often acceptable if not optimal.
- Greedy algorithms may also be followed by a fine tuning stage in which a joint optimization algorithm searches for an optimal solution to the full problem.
- Initializing the joint optimization algorithm with a greedy solution can greatly speed it up and improve the quality of the solution it finds.

Greedy Supervised Pretraining

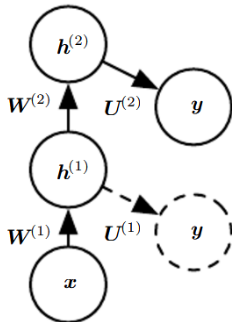


(a)

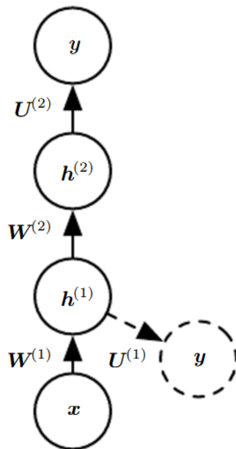


(b)

Greedy Supervised Pretraining



(c)



(d)

Section 4

Conclusion: Designing Models to Aid Optimization

Conclusion

- To improve optimization, the best strategy is not always to improve the optimization algorithm.
- In practice, it is more important to choose a model family that is easy to optimize than to use a powerful optimization algorithm.
- Modern neural nets have been designed so that their local gradient information corresponds reasonably well to moving toward a distant solution.
- Other model design strategies can help to make optimization easier.
- Example: **auxiliary losses**, **skip connections**.