Things we will look at today

- Stochastic Gradient Descent
- Momentum Method and the Nesterov Variant
- Adaptive Learning Methods (AdaGrad, RMSProp, Adam)
- Batch Normalization
- Initialization Heuristics
- Polyak Averaging
- On Slides but for self study: Newton and Quasi Newton Methods (BFGS, L-BFGS, Conjugate Gradient)
Optimization

- We’ve seen backpropagation as a method for computing gradients
- Assignment: Was about implementation of SGD in conjunction with backprop
- Let’s see a family of first order methods
Algorithm 1 Batch Gradient Descent at Iteration $k$

Require: Learning rate $\epsilon_k$

Require: Initial Parameter $\theta$

1: while stopping criteria not met do
2: Compute gradient estimate over $N$ examples:
3: $\hat{g} \leftarrow +\frac{1}{N} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
4: Apply Update: $\theta \leftarrow \theta - \epsilon \hat{g}$
5: end while

- Positive: Gradient estimates are stable
- Negative: Need to compute gradients over the entire training for one update
Gradient Descent
Stochastic Gradient Descent

**Algorithm 2** Stochastic Gradient Descent at Iteration $k$

**Require:** Learning rate $\epsilon_k$

**Require:** Initial Parameter $\theta$

1. `while` stopping criteria not met `do`
2. Sample example $(x^{(i)}, y^{(i)})$ from training set
3. Compute gradient estimate:
4. $\hat{g} \leftarrow +\nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$
5. Apply Update: $\theta \leftarrow \theta - \epsilon \hat{g}$
6. `end while`

- $\epsilon_k$ is learning rate at step $k$
- Sufficient condition to guarantee convergence:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$
Learning Rate Schedule

- In practice the learning rate is decayed linearly till iteration $\tau$
  
  $$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \text{ with } \alpha = \frac{k}{\tau}$$

- $\tau$ is usually set to the number of iterations needed for a large number of passes through the data
- $\epsilon_\tau$ should roughly be set to 1% of $\epsilon_0$
- How to set $\epsilon_0$?
Minibatching

Potential Problem: Gradient estimates can be very noisy

Obvious Solution: Use larger mini-batches

Advantage: Computation time per update does not depend on number of training examples $N$

This allows convergence on extremely large datasets

See: Large Scale Learning with Stochastic Gradient Descent by Leon Bottou
Stochastic Gradient Descent
Batch Gradient Descent:

\[ \hat{g} \leftarrow + \frac{1}{N} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)}) \]
\[ \theta \leftarrow \theta - \epsilon \hat{g} \]

SGD:

\[ \hat{g} \leftarrow + \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)}) \]
\[ \theta \leftarrow \theta - \epsilon \hat{g} \]
Momentum

• The Momentum method is a method to accelerate learning using SGD
• In particular SGD suffers in the following scenarios:
  • Error surface has high curvature
  • We get small but consistent gradients
  • The gradients are very noisy
Gradient Descent would move quickly down the walls, but very slowly through the valley floor.
Momentum

- How do we try and solve this problem?
- Introduce a new variable \( v \), the velocity
- We think of \( v \) as the direction and speed by which the parameters move as the learning dynamics progresses
- The velocity is an exponentially decaying moving average of the negative gradients

\[
v \leftarrow \alpha v - \epsilon \nabla_{\theta} \left( L(f(x^{(i)}; \theta), y^{(i)}) \right)
\]

\[\alpha \in [0, 1)\] Update rule: \( \theta \leftarrow \theta + v \)
Momentum

Let’s look at the velocity term:

\[ \mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left( L(f(x^{(i)}; \theta), y^{(i)}) \right) \]

The velocity accumulates the previous gradients

What is the role of \( \alpha \)?

- If \( \alpha \) is larger than \( \epsilon \) the current update is more affected by the previous gradients
- Usually values for \( \alpha \) are set high \( \approx 0.8, 0.9 \)
Momentum

Momentum Step

Actual Step

Gradient Step
Momentum: Step Sizes

- In SGD, the step size was the norm of the gradient scaled by the learning rate $\epsilon \| g \|$. Why?
- While using momentum, the step size will also depend on the norm and alignment of a sequence of gradients.
- For example, if at each step we observed $g$, the step size would be (exercise!):
  $$\epsilon \frac{\| g \|}{1 - \alpha}$$

- If $\alpha = 0.9 \implies$ multiply the maximum speed by 10 relative to the current gradient direction.
Momentum

Illustration of how momentum traverses such an error surface better compared to Gradient Descent
Algorithm 2 Stochastic Gradient Descent with Momentum

Require: Learning rate $\epsilon_k$
Require: Momentum Parameter $\alpha$
Require: Initial Parameter $\theta$
Require: Initial Velocity $v$

1: while stopping criteria not met do
2: Sample example $(x^{(i)}, y^{(i)})$ from training set
3: Compute gradient estimate:
4: $\hat{g} \leftarrow +\nabla_{\theta} L(f(x^{(i)}; \theta), y^{(i)})$
5: Compute the velocity update:
6: $v \leftarrow \alpha v - \epsilon \hat{g}$
7: Apply Update: $\theta \leftarrow \theta + v$
8: end while
Nesterov Momentum

- Another approach: First take a step in the direction of the accumulated gradient
- Then calculate the gradient and make a correction

Accumulated Gradient → Correction → New Accumulated Gradient
Nesterov Momentum

Next Step
Let’s Write it out..

- Recall the velocity term in the Momentum method:

  \[ \mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\theta} \left( L(f(x^{(i)}; \theta), y^{(i)}) \right) \]

- Nesterov Momentum:

  \[ \mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\theta} \left( L(f(x^{(i)}; \theta + \alpha \mathbf{v}), y^{(i)}) \right) \]

- Update: \( \theta \leftarrow \theta + \mathbf{v} \)
Algorithm 3 SGD with Nesterov Momentum

Require: Learning rate $\epsilon$
Require: Momentum Parameter $\alpha$
Require: Initial Parameter $\theta$
Require: Initial Velocity $v$

1: while stopping criteria not met do
2: Sample example $(\mathbf{x}^{(i)}, y^{(i)})$ from training set
3: Update parameters: $\tilde{\theta} \leftarrow \theta + \alpha v$
4: Compute gradient estimate:
5: $\hat{\mathbf{g}} \leftarrow + \nabla_{\tilde{\theta}} L(f(\mathbf{x}^{(i)}; \tilde{\theta}), y^{(i)})$
6: Compute the velocity update: $v \leftarrow \alpha v - \epsilon \hat{\mathbf{g}}$
7: Apply Update: $\theta \leftarrow \theta + v$
8: end while
Adaptive Learning Rate Methods
Motivation

- Till now we assign the same learning rate to all features
- If the features vary in importance and frequency, why is this a good idea?
- It’s probably not!
Motivation

Nice (all features are equally important)
Motivation

Harder!
AdaGrad

- **Idea**: Downscale a model parameter by square-root of sum of squares of all its historical values.
- Parameters that have large partial derivative of the loss – learning rates for them are rapidly declined.
- Some interesting theoretical properties.
AdaGrad

**Algorithm 4 AdaGrad**

**Require:** Global Learning rate $\epsilon$, Initial Parameter $\theta$, $\delta$

Initialize $r = 0$

1. **while** stopping criteria not met **do**
2. Sample example $(x^{(i)}, y^{(i)})$ from training set
3. Compute gradient estimate: $\hat{g} \leftarrow +\nabla_{\theta} L(f(x^{(i)}; \theta), y^{(i)})$
4. Accumulate: $r \leftarrow r + \hat{g} \odot \hat{g}$
5. Compute update: $\Delta \theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot \hat{g}$
6. Apply Update: $\theta \leftarrow \theta + \Delta \theta$
7. **end while**
RMSProp

- AdaGrad is good when the objective is convex.
- AdaGrad might shrink the learning rate too aggressively, we want to keep the history in mind.
- We can adapt it to perform better in non-convex settings by accumulating an exponentially decaying average of the gradient.
- This is an idea that we use again and again in Neural Networks.
- Currently has about 500 citations on scholar, but was proposed in a slide in Geoffrey Hinton’s coursera course.
RMSProp

Algorithm 5 RMSProp

Require: Global Learning rate $\epsilon$, decay parameter $\rho$, $\delta$

Initialize $r = 0$

1: while stopping criteria not met do
2: Sample example $(x^{(i)}, y^{(i)})$ from training set
3: Compute gradient estimate: $\hat{g} \leftarrow +\nabla_{\theta} L(f(x^{(i)}; \theta), y^{(i)})$
4: Accumulate: $r \leftarrow \rho r + (1 - \rho) \hat{g} \odot \hat{g}$
5: Compute update: $\Delta \theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot \hat{g}$
6: Apply Update: $\theta \leftarrow \theta + \Delta \theta$

7: end while
RMSProp with Nesterov

**Algorithm 6** RMSProp with Nesterov

**Require:** Global Learning rate $\epsilon$, decay parameter $\rho$, $\delta$, $\alpha$, $v$

Initialize $r = 0$

1: while stopping criteria not met do
2: Sample example $(x^{(i)}, y^{(i)})$ from training set
3: Compute Update: $\tilde{\theta} \leftarrow \theta + \alpha v$
4: Compute gradient estimate: $\hat{g} \leftarrow +\nabla_{\tilde{\theta}} L(f(x^{(i)}; \tilde{\theta}), y^{(i)})$
5: Accumulate: $r \leftarrow \rho r + (1 - \rho) \hat{g} \odot \hat{g}$
6: Compute Velocity: $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot \hat{g}$
7: Apply Update: $\theta \leftarrow \theta + v$
8: end while
Adam

- We could have used RMSProp with momentum
- Use of Momentum with rescaling is not well motivated
- Adam is like RMSProp with Momentum but with bias correction terms for the first and second moments
Adam: ADAdptive Moments

Algorithm 7 RMSProp with Nesterov

Require: $\epsilon$ (set to 0.0001), decay rates $\rho_1$ (set to 0.9), $\rho_2$ (set to 0.9), $\theta$, $\delta$

Initialize moments variables $s = 0$ and $r = 0$, time step $t = 0$

1: while stopping criteria not met do
2: Sample example $(x^{(i)}, y^{(i)})$ from training set
3: Compute gradient estimate: $\hat{g} \leftarrow +\nabla_{\theta} L(f(x^{(i)}; \theta), y^{(i)})$
4: $t \leftarrow t + 1$
5: Update: $s \leftarrow \rho_1 s + (1 - \rho_1) \hat{g}$
6: Update: $r \leftarrow \rho_2 r + (1 - \rho_2) \hat{g} \odot \hat{g}$
7: Correct Biases: $\hat{s} \leftarrow \frac{s}{1 - \rho_1^t}$, $\hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$
8: Compute Update: $\Delta \theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r} + \delta}}$
9: Apply Update: $\theta \leftarrow \theta + \Delta \theta$
10: end while
All your GRADs are belong to us!

\textbf{SGD}: \theta \leftarrow \theta - \epsilon \hat{g}

\textbf{Momentum}: \textbf{v} \leftarrow \alpha \textbf{v} - \epsilon \hat{g} \text{ then } \theta \leftarrow \theta + \textbf{v}

\textbf{Nesterov}: \textbf{v} \leftarrow \alpha \textbf{v} - \epsilon \nabla_{\theta} \left( L(f(x^{(i)}; \theta + \alpha \textbf{v}), y^{(i)}) \right) \text{ then } \theta \leftarrow \theta + \textbf{v}

\textbf{AdaGrad}: \textbf{r} \leftarrow \textbf{r} + g \odot g \text{ then } \Delta \theta - \leftarrow \frac{\epsilon}{\delta + \sqrt{\textbf{r}}} \odot g \text{ then } \theta \leftarrow \theta + \Delta \theta

\textbf{RMSProp}: \textbf{r} \leftarrow \rho \textbf{r} + (1 - \rho) \hat{g} \odot \hat{g} \text{ then } \Delta \theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\textbf{r}}} \odot \hat{g} \text{ then } \theta \leftarrow \theta + \Delta \theta

\textbf{Adam}: \hat{s} \leftarrow \frac{s}{1 - \rho_1^t}, \hat{r} \leftarrow \frac{r}{1 - \rho_2^t} \text{ then } \Delta \theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta} \text{ then } \theta \leftarrow \theta + \Delta \theta
Batch Normalization
A Difficulty in Training Deep Neural Networks

A deep model involves composition of several functions

\[ \hat{y} = W_4^T (\tanh(W_3^T (\tanh(W_2^T (\tanh(W_1^T \mathbf{x} + b_1) + b_2) + b_3)))) \]
A Difficulty in Training Deep Neural Networks

- We have a recipe to compute gradients (Backpropagation), and update every parameter (we saw half a dozen methods).
- Implicit Assumption: Other layers don’t change i.e. other functions are fixed.
- In Practice: We update all layers simultaneously.
- This can give rise to unexpected difficulties.
- Let’s look at two illustrations.
Consider a second order approximation of our cost function (which is a function composition) around current point $\theta^{(0)}$:

$$J(\theta) \approx J(\theta^{(0)}) + (\theta - \theta^{(0)})^T g + \frac{1}{2}(\theta - \theta^{(0)})^T H(\theta - \theta^{(0)})$$

- $g$ is gradient and $H$ the Hessian at $\theta^{(0)}$
- If $\epsilon$ is the learning rate, the new point

$$\theta = \theta^{(0)} - \epsilon g$$
Intuition

- Plugging our new point, \( \theta = \theta^{(0)} - \epsilon g \) into the approximation:

\[
J(\theta^{(0)} - \epsilon g) = J(\theta^{(0)}) - \epsilon g^T g + \frac{1}{2} g^T H g
\]

- There are three terms here:
  - Value of function before update
  - Improvement using gradient (i.e. first order information)
  - Correction factor that accounts for the curvature of the function
Intuition

\[ J(\theta^{(0)} - \epsilon g) = J(\theta^{(0)}) - \epsilon g^T g + \frac{1}{2} g^T H g \]

**Observations:**
- \( g^T H g \) too large: Gradient will start moving upwards
- \( g^T H g = 0 \): \( J \) will decrease for even large \( \epsilon \)
- Optimal step size \( \epsilon^* = g^T g \) for zero curvature,
  \[ \epsilon^* = \frac{g^T g}{g^T H g} \] to take into account curvature

**Conclusion:** Just neglecting second order effects can cause problems (remedy: second order methods). What about higher order effects?
Higher Order Effects: Toy Model

Just one node per layer, no non-linearity

\( \hat{y} \) is linear in \( x \) but non-linear in \( w_i \)
Higher Order Effects: Toy Model

- Suppose $\delta = 1$, so we want to decrease our output $\hat{y}$
- Usual strategy:
  - Using backprop find $g = \nabla_w (\hat{y} - y)^2$
  - Update weights $w := w - \epsilon g$
- The first order Taylor approximation (in previous slide) says the cost will reduce by $\epsilon g^T g$
- If we need to reduce cost by 0.1, then learning rate should be $\frac{0.1}{g^T g}$
The new $\hat{y}$ will however be:

$$\hat{y} = x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \ldots (w_l - \epsilon g_l)$$

Contains terms like $\epsilon^3 g_1 g_2 g_3 w_4 w_5 \ldots w_l$

If weights $w_4, w_5, \ldots, w_l$ are small, the term is negligible. But if large, it would explode

**Conclusion:** Higher order terms make it very hard to choose the right learning rate

**Second Order Methods** are already expensive, $n$th order methods are hopeless. Solution?
Batch Normalization

- Method to reparameterize a deep network to reduce co-ordination of update across layers
- Can be applied to input layer, or any hidden layer
- Let $H$ be a design matrix having activations in any layer for $m$ examples in the mini-batch

$$H = \begin{bmatrix}
h_{11} & h_{12} & h_{13} & \ldots & h_{1k} \\
h_{21} & h_{22} & h_{23} & \ldots & h_{2k} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
h_{m1} & h_{m2} & h_{m3} & \ldots & h_{mk}
\end{bmatrix}$$
Batch Normalization

\[ H = \begin{bmatrix}
    h_{11} & h_{12} & h_{13} & \ldots & h_{1k} \\
    h_{21} & h_{22} & h_{23} & \ldots & h_{2k} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    h_{m1} & h_{m2} & h_{m3} & \ldots & h_{mk}
\end{bmatrix} \]

- Each row represents all the activations in layer for one example
- \textbf{Idea:} Replace \( H \) by \( H' \) such that:
  \[ H' = \frac{H - \mu}{\sigma} \]
  - \( \mu \) is mean of each unit and \( \sigma \) the standard deviation
Batch Normalization

- $\mu$ is a vector with $\mu_j$ the column mean
- $\sigma$ is a vector with $\sigma_j$ the column standard deviation
- $H_{i,j}$ is normalized by subtracting $\mu_j$ and dividing by $\sigma_j$
Batch Normalization

During training we have:

\[ \mu = \frac{1}{m} \sum_j H_{:,j} \]

\[ \sigma = \sqrt{\delta + \frac{1}{m} \sum_j (H - \mu)^2} \]

We then operate on \( H' \) as before \( \implies \) we backpropagate through the normalized activations
Why is this good?

- The update will never act to only increase the mean and standard deviation of any activation.
- Previous approaches added penalties to cost or per layer to encourage units to have standardized outputs.
- Batch normalization makes the reparameterization easier.
- At test time: Use running averages of $\mu$ and $\sigma$ collected during training, use these for evaluating new input $x$. 

An Innovation

- Standardizing the output of a unit can limit the expressive power of the neural network
- Solution: Instead of replacing $H$ by $H'$, replace it with $\gamma H' + \beta$
- $\gamma$ and $\beta$ are also learned by backpropagation
- Normalizing for mean and standard deviation was the goal of batch normalization, why add $\gamma$ and $\beta$ again?
Initialization Strategies
In convex problems with good \( \epsilon \) no matter what the initialization, convergence is guaranteed.

In the non-convex regime initialization is much more important.

Some parameter initialization can be unstable, not converge.

Neural Networks are not well understood to have principled, mathematically nice initialization strategies.

What is known: Initialization should break symmetry (quiz!)

What is known: Scale of weights is important.

Most initialization strategies are based on intuitions and heuristics.
Some Heuristics

- For a fully connected layer with $m$ inputs and $n$ outputs, sample:
  \[ W_{ij} \sim U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}) \]

- Xavier Initialization: Sample
  \[ W_{ij} \sim U(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}) \]

- Xavier initialization is derived considering that the network consists of matrix multiplications with no nonlinearities
- Works well in practice!
More Heuristics

- Saxe et al. 2013, recommend initializing to random orthogonal matrices, with a carefully chosen gain $g$ that accounts for non-linearities.
- If $g$ could be divined, it could solve the vanishing and exploding gradients problem (more later).
- The idea of choosing $g$ and initializing weights accordingly is that we want norm of activations to increase, and pass back strong gradients.
- Martens 2010, suggested an initialization that was sparse: Each unit could only receive $k$ non-zero weights.
- **Motivation:** It is a bad idea to have all initial weights to have the same standard deviation $\frac{1}{\sqrt{m}}$. 

Lecture 6 Optimization for Deep Neural Networks CMSC 35246
Consider gradient descent above with high step size $\epsilon$
Polyak Averaging: Motivation

Gradient points towards left
Polyak Averaging: Motivation

Gradient points towards right
Polyak Averaging: Motivation

Gradient points towards left
Polyak Averaging: Motivation

Gradient points towards right
A Solution: Polyak Averaging

- Suppose in $t$ iterations you have parameters $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(t)}$
- Polyak Averaging suggests setting $\hat{\theta}(t) = \frac{1}{t} \sum_i \theta^{(i)}$
- Has strong convergence guarantees in convex settings
- Is this a good idea in non-convex problems?
In non-convex surfaces the parameter space can differ greatly in different regions.

Averaging is not useful.

Typical to consider the exponentially decaying average instead:

\[ \hat{\theta}(t) = \alpha \hat{\theta}(t-1) + (1 - \alpha) \hat{\theta}(t) \text{ with } \alpha \in [0, 1] \]
Next time

- Convolutional Neural Networks