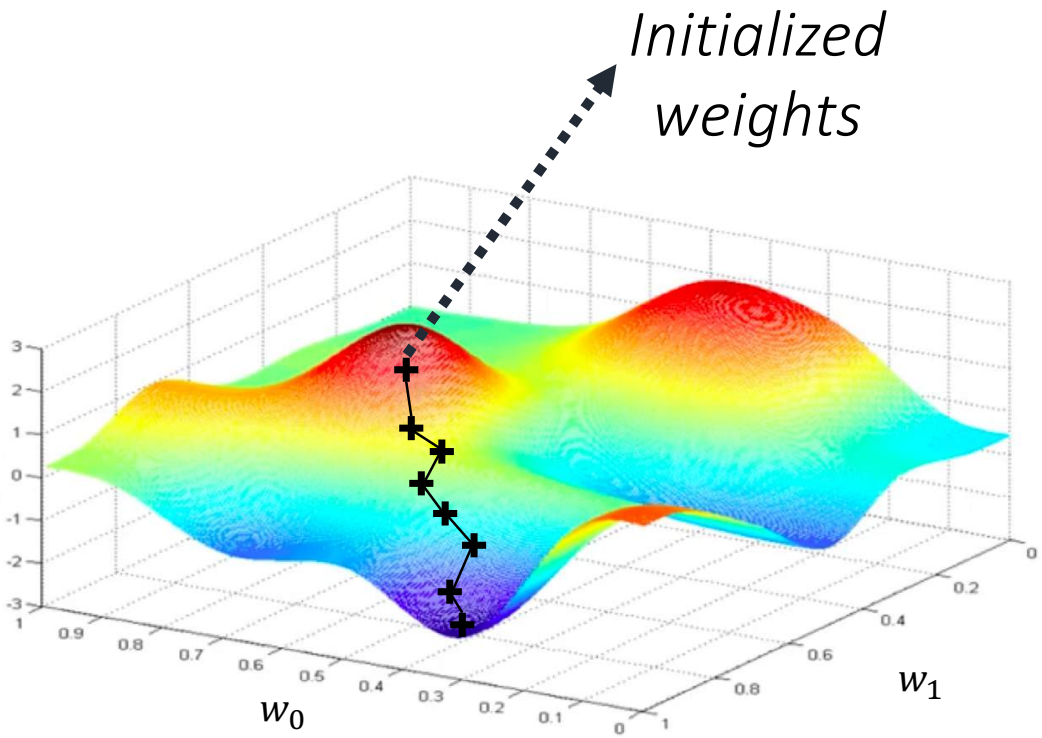
Deep Learning - 2019

Training - Optimization

Prof. Avishek Anand

What we learnt before ?



$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Gradient Descent Algorithm

Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$

while *training error has not converged* **do**

  Compute gradient $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

  Update weights $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

return weights

Minimizing Loss = finding the minima in this loss surface

Mini-batches

Gradient Descent Algorithm

Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

while *not converged* **do**

  Compute gradient $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

  Update weights $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

return weights

Stochastic Gradient Descent

Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

while *not converged* **do**

  Pick single data point i

  Compute gradient $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$

  Update weights $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

return weights

Computationally expensive

Batched Gradient Descent

Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

while *not converged* **do**

  Pick batch B of data points

  Compute gradient $\frac{\partial J(W)}{\partial \boldsymbol{W}} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$

  Update weights $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

return weights

Fast to compute and much better estimate of the true gradient

- **Stochastic Gradient Descent (SGD)** : Compute gradients on the samples
- Mini-batches : Computed the gradients for a batch of points rather than each
- Smoother convergence, allows for larger learning rates
- Can parallelize computation, fully use GPU potential

Your Optimization Toolkit

- Training is based on gradient computation/updation
- Problems in training
  - Gradients do not change
  - Training is slow
  - Training does not converge
- Improve training performance using
  - Stochastic Gradient Descent
  - Setting the learning rate
  - Weight initialization
  - Normalization

Gradient Descent Algorithm

Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$

while *training error has not converged* **do**

  Compute gradient $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

  Update weights $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

return weights

Stochastic Gradient Descent

Algorithm 2 Stochastic Gradient Descent at Iteration k

Require: Learning rate ϵ_k → Initial learning rate

Require: Initial Parameter θ → Initialization

1: **while** stopping criteria not met **do**

2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

3: Compute gradient estimate:

4: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$ → Computing gradients

5: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$ → Updating Parameters/weights

6: **end while**

We change notations to align with the book and most papers

Gradient vector

Partial deriv. w.r.t a weights

Parameters or weights

$$\hat{\mathbf{g}} \leftarrow +\nabla_\theta L\left(f\left(\mathbf{x}^{(i)}; \theta\right), \mathbf{y}^{(i)}\right)$$

Mini-batching

- 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

$$\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})$$

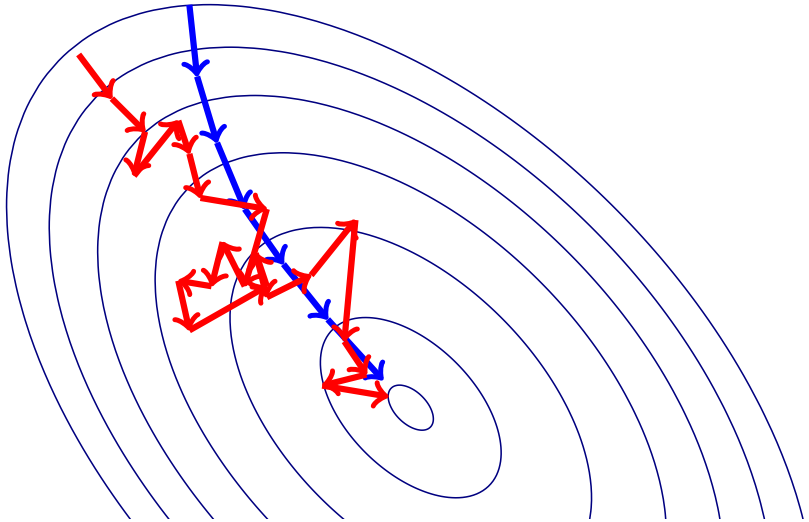
$$\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$$

Stochastic Gradient Descent

$$\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})$$

$$\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$$

Batched Gradient Descent



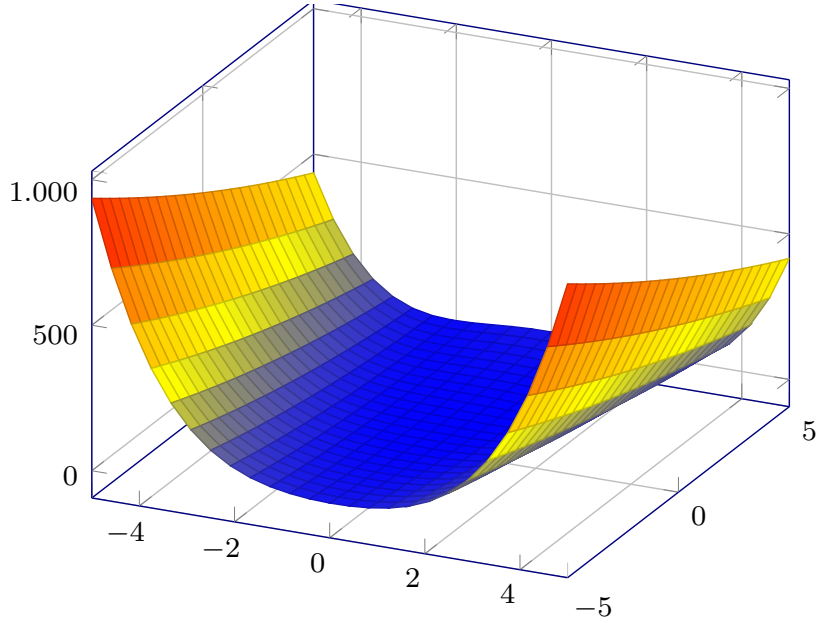
Momentum

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

- 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

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\nabla_\theta\left(L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})\right) \qquad \alpha \in [0,1)$$

$$\theta \leftarrow \theta + \mathbf{v}$$

- The velocity accumulates the previous gradients
- What is the role of α?
  - If α is larger than ε the current update is more affected by the previous gradients
  - Usually values for α are set high ≈ 0.8, 0.9

SGD with Momentum

Algorithm 2 Stochastic Gradient Descent with Momentum

Require: Learning rate ϵ_k
Require: Momentum Parameter α
Require: Initial Parameter θ
Require: Initial Velocity $\mathbf{v}$

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

Earlier

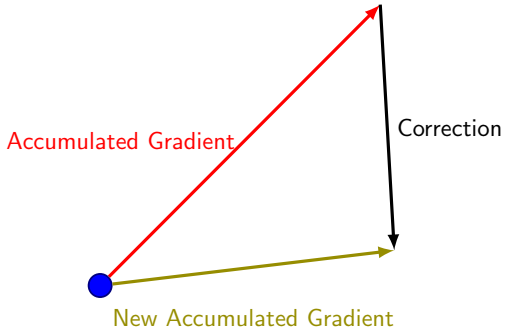
$$\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$$

Momentum

$$\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}} + \alpha\mathbf{v}$$

Nesterov Momentum

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

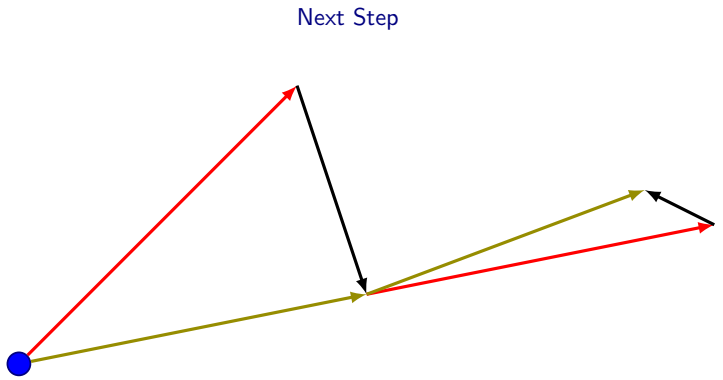


- Standard Momentum

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

- Nesterov Momentum

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

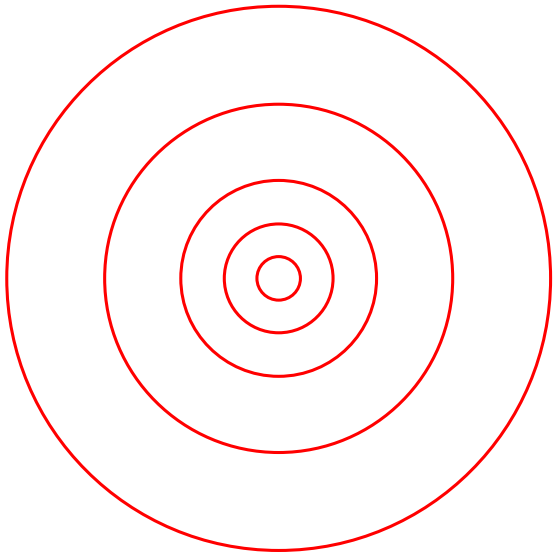


- Parameter Update

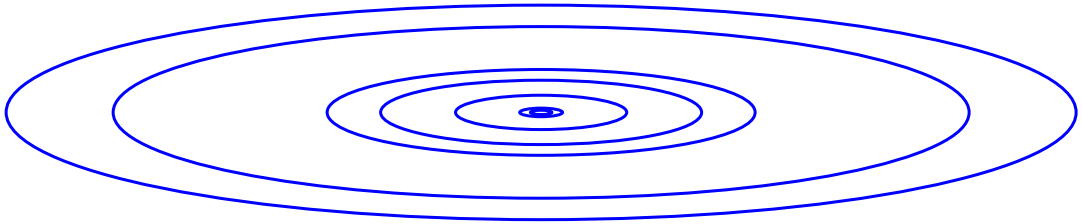
$$\theta \leftarrow \theta + \mathbf{v}$$

Adaptive Learning Rates

Why adaptive learning rates ?



Nice (all features are equally important)



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

Algorithm 4 AdaGrad

Require: Global Learning rate ϵ, Initial Parameter θ, δ

Initialize $\mathbf{r} = 0$

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

Algorithm 5 RMSProp

Require: Global Learning rate ϵ, decay parameter ρ, δ

Initialize $\mathbf{r} = 0$

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

Require: ϵ (set to 0.0001), decay rates ρ_1 (set to 0.9), ρ_2 (set to 0.9), θ, δ

Initialize moments variables $\mathbf{s} = 0$ and $\mathbf{r} = 0$, time step t $= 0$

while stopping criteria not met **do**
- Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
- Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
- $t \leftarrow t + 1$
- Update: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\hat{\mathbf{g}}$
- Update: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
- Correct Biases: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t}$
- Compute Update: $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\delta}$
- Apply Update: $\theta \leftarrow \theta + \Delta\theta$

end while

Summary

SGD: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

Momentum: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \mathbf{v}$

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

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

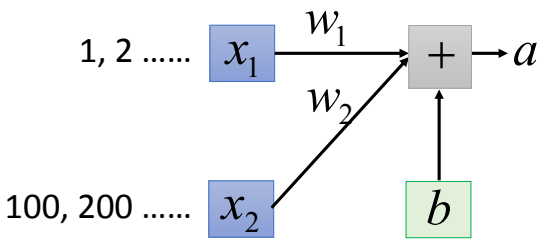
RMSProp: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$ then $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \Delta\theta$

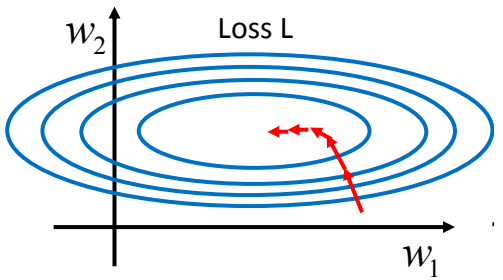
Adam: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$ then $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$ then $\theta \leftarrow \theta + \Delta\theta$

Batch Normalization

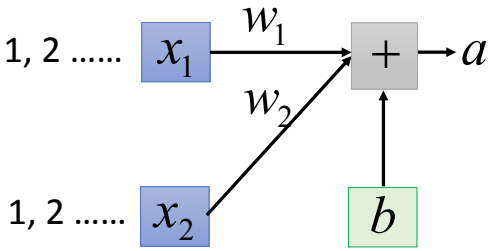
Scaling Features

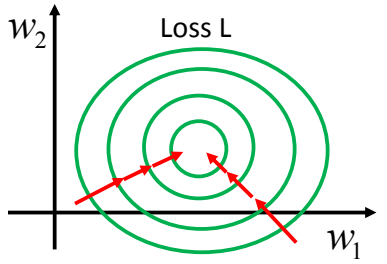
- Problems in optimization if features are of different scales
  - Slow progress of gradient descent





- Normalization vs Standardization
  - **Normalization:** Normalise the range
  - Divide each feature value by (max -min)
  - Drawback ?



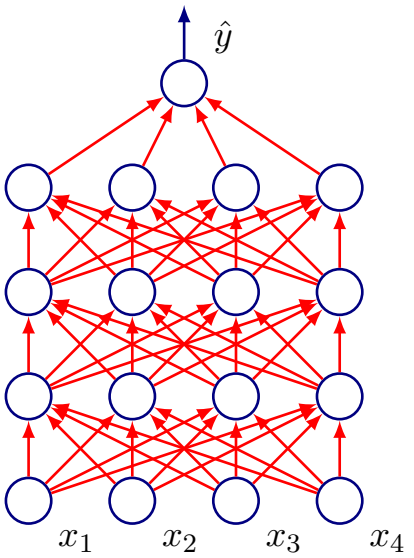


- **Standardization** (or **Z-score normalization**) is the process of rescaling the features so that they'll have the properties of a Gaussian distribution

$$z = \frac{x - \mu}{\sigma}$$

A problem in training deep NN

- Hidden layer of a neural networks are latent representations or features
  - Can show the same variability in activation values
  - Lead to slow training
- Why not use the same idea here ?
- Idea 1:
  - Make a forward pass with all training instances
  - Features at layer k are the vector of activation values at layer k
  - Now use Standardization



$$\mu_k \leftarrow \frac{1}{N}\sum_{i=1}^{N} h_i \qquad \sigma_k^2 \leftarrow \frac{1}{N}\sum_{i=1}^{k} (h_i - \mu_k)^2 \qquad \hat{h}_i = \frac{h_i - \mu_k}{\sqrt{\sigma_k^2}}$$

Batch Normalization

- For a mini-batch of M instances B = $h_1, h_2, \ldots, h_M$
- Mini-batch mean and variance are computed

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{M}\sum_{i=1}^{M} h_i \qquad \sigma^2_{\mathcal{B}} \leftarrow \frac{1}{M}\sum_{i=1}^{M} (h_i - \mu_{\mathcal{B}})^2$$

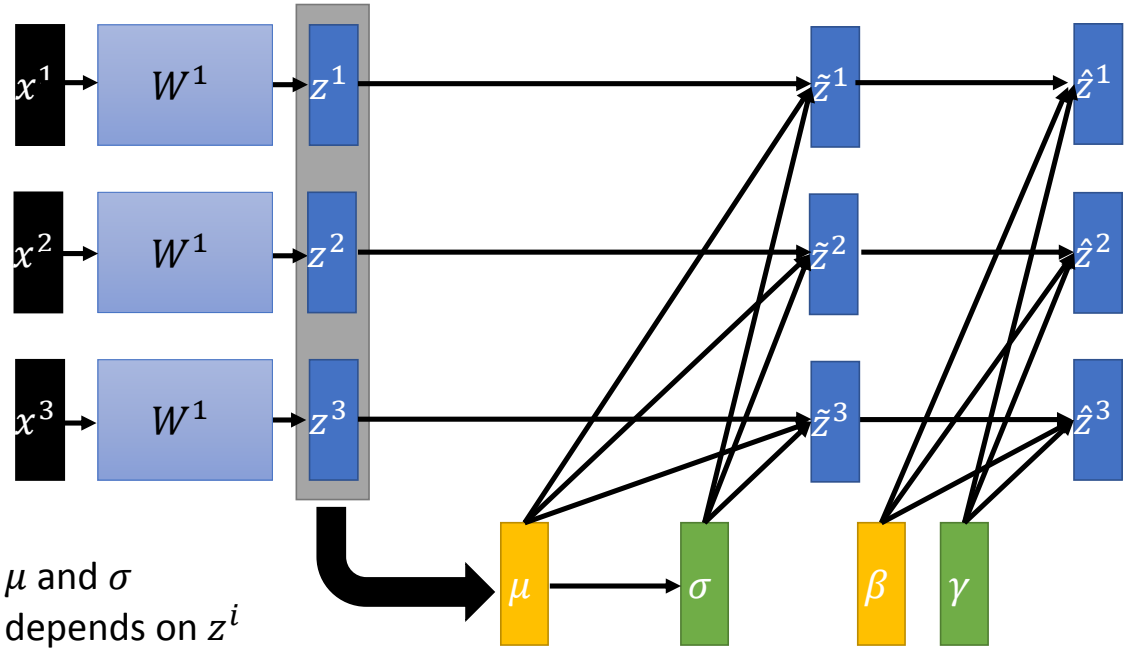
- New Activations are normalized

$$\hat{h}_i = \gamma\left(\frac{h_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}\right) + \beta$$

- μ: mean of x in mini-batch
- σ: std of x in mini-batch
- γ: scale
- β: shift

- The scale and shift parameters are both trainable parameters

Batch Normalization



μ and σ depends on z^i

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

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

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

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

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

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

$$\hat{h}_i = \gamma\left(\frac{h_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}\right) + \beta$$

Batch Normalization

- 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

- During Test time
  - How do we use BN when we have a single instance ?
  - Keep a mean and variance over the entire training set

Initialization

Motivation

- In convex problems with good learning rate
  - 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 happens when we initialize all weights to 0 ?
- What is known: Scale of weights is important
  - Most initialization strategies are based on intuitions and heuristics

Xavier Initialization

- 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 nonlinearites

- Works well in practice!

Saxe Initialization

- 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 divided, 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 1/sqrt(m)

Practical Tips

General Recipe for Successful NN

1. Select network structure appropriate for problem
   - CNN, RNN, Generative Models, FCN
2. Check for implementation bugs with gradient checks
3. Parameter initialization (this lecture)
4. Optimization tricks (this lecture)
5. Check if the model is powerful enough to overfit
   - If not, change model structure or make model "larger"
   - If you can overfit: **Regularize** (next lecture)

Gradient Checks

- Allow you to know that there are no bugs in your neural network implementation!
- **Steps:**
  - Implement your gradient
  - Implement a finite difference computation by looping through the parameters of your network, adding and subtracting a small epsilon (~10-4) and estimate derivative

$$f'(\theta) \approx \frac{J\left(\theta^{(i+)}\right) - J\left(\theta^{(i-)}\right)}{2\epsilon} \qquad \theta^{(i+)} = \theta + \epsilon \times e_i$$

  - Compare the two and make sure they are almost the same

Gradient Checks

- If you gradient fails and you don't know why?
- What now? Create a very tiny synthetic model and dataset
- Simplify your model until you have no bug!

- Example: Start from simplest model then go to what you want:
  - Only softmax on fixed input
  - Backprop into input and softmax
  - Add single unit single hidden layer
  - Add multi unit single layer
  - Add bias
  - Add second layer single unit, add multiple units, bias
  - Add one softmax on top, then two softmax layers

References

- https://medium.com/@SeoJaeDuk/deeper-understanding-of-batch-normalization-with-interactive-code-in-tensorflow-manual-back-1d50d6903d35
- http://ruder.io/optimizing-gradient-descent/
- Deep Learning by goodfellow et al. Chapter on Optimization