Training Deep Neural Networks: Optimization and Regularization

Vineeth N Balasubramanian
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

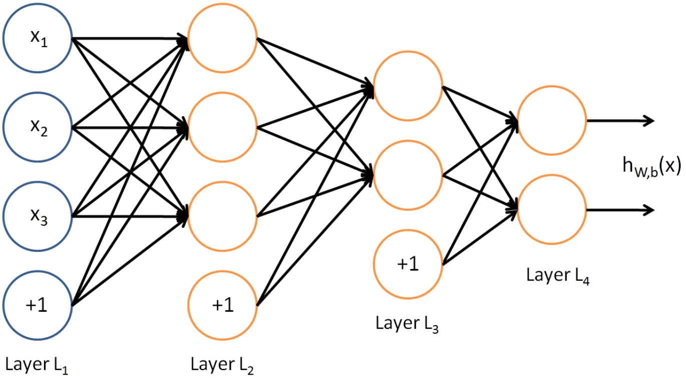
CVIT ML Summer School
IIIT, Hyderabad

8^{th} Jul 2019

Outline

Backprop

Given a simple multilayer neural network (or multilayer perceptron):



Backprop

- A fixed training set $\{(x(1), y(1)), \cdots, (x(m), y(m))\}$ of m training examples
- Parameters $\theta = \{W, b\}$, weights and biases
- Mean square cost function[1] for a single example:

$$J(\theta; x, y) = \frac{1}{2} \|h_\theta(x) - y\|^2$$

- Overall cost function is given by:

$$\begin{aligned} J(\theta) &= \left[\frac{1}{m}\sum_{i=1}^{m} J(\theta; x^{(i)}, y^{(i)})\right] \\ &= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_\theta(x^{(i)}) - y^{(i)}\right\|^2\right)\right] \end{aligned}$$

[1] Cost function, Error function and Loss function are synonymous in this context

Backprop

- A weight decay term is added to decrease the magnitude of the weights, and help prevent overfitting
- Cost function now given by:

Weight Decay Term

$$J(\theta) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\| h_\theta(x^{(i)}) - y^{(i)}\right\|^2\right)\right] + \boxed{\frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2}$$

- Weight decay parameter λ controls the relative importance of the two terms

Backprop

- Derivative of overall cost function given by:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)}) \right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_{i}^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_{i}^{(l)}} J(W,b;x^{(i)},y^{(i)})$$

Backprop Algorithm

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit i in layer n_l (the output layer), set:

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \cdots, 2$
   For each node i in layer l, set:

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

4. Compute the desired partial derivatives, which are given as:

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T,$$
$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}.$$

Gradient Descent using Backpropagation

1. Set $\Delta W^{(l)} := 0, \Delta b^{(l)} := 0$ (matrix/vector of zeros) for all l.
2. For $i = 1$ to m
    1. Use backpropagation to compute $\nabla_{\theta^{(l)}} J(\theta; x, y)$
    2. Set: $\Delta\theta^{(l)} := \Delta\theta^{(l)} + \nabla_{\theta^{(l)}} J(\theta; x, y)$
3. Update the parameters:

$$W^{(l)} = W^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right]$$

$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right]$$

4. Repeat for all data points until convergence

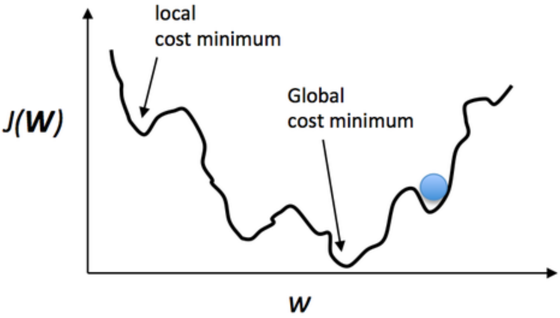
Some keywords to keep track of

1. Loss function: Error used for backpropagation
2. Activation function: A (typically non-linear) function applied on output of neurons
3. Iteration: equivalent to when a weight update is done (could be after every training example, or after a batch of training examples, or after the entire training set
4. Epoch: When the entire training set has been used once to update the weights (Note: we have to run training for many such epochs to train deep networks!)
5. Learning rate (α): Size of the step in the direction of the negative gradient

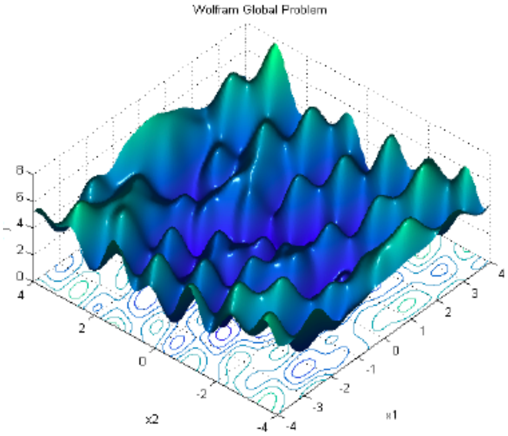
Outline

Local Minima

- Unlike convex objective functions that have a global minimum, non-convex functions as in DL have multiple local minima

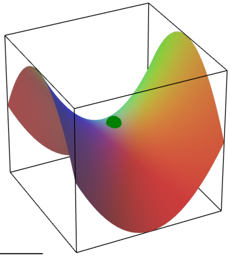


Loss Surface



Saddle Points, Plateaus and Other Flat Regions

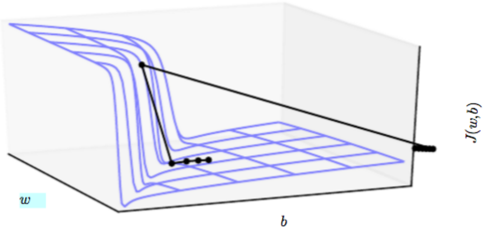
- More points on the cost surface with low gradients: *saddle points*, *plateaus*, *flat regions*
- Saddle Points: Local minimum along one cross-section of cost function, and local maximum along another
- In higher-dimensional spaces, local minima are rare and saddle points are more common[2]



[2] Dauphin, Pascanu, Gulcehre, Cho, Ganguli, Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

Cliffs

- Highly non-linear deep networks have cliff areas
- Most common in cost functions for RNNs, since such models involve multiplication of many terms (over time)

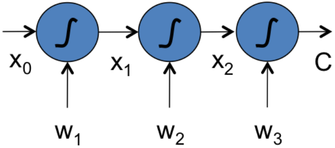


[3]

[3] Pascanu et al. "On the difficulty of training recurrent neural networks." ICML 2013.

Vanishing/Exploding Gradient

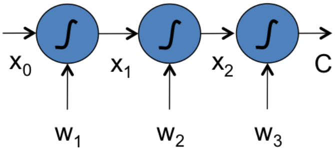
Consider a simple network:



$$\frac{\partial J}{\partial x_0} = \underbrace{\sigma'(w_3^T x_2)}_{< 1/4} \times w_3 \times \underbrace{\sigma'(w_2^T x_1)}_{< 1/4} \times w_2 \times \underbrace{\sigma'(w_1^T x_0)}_{< 1/4} \times w_1$$

Vanishing/Exploding Gradient

Consider a simple network:



$$\frac{\partial J}{\partial x_0} = \boxed{\underbrace{\sigma'(w_3^T x_2)}_{< 1/4}} \times w_3 \times \boxed{\underbrace{\sigma'(w_2^T x_1)}_{< 1/4}} \times w_2 \times \boxed{\underbrace{\sigma'(w_1^T x_0)}_{< 1/4}} \times w_1$$

- Deeper the network, gradients vanish quickly, thereby slowing the rate of change in initial layers
- Problem accentuated in long-term RNNs
- Exploding gradients happen when the individual layer gradients are much higher than 1, for instance

Slow Convergence

Given the issues:

- Cost surface is often non-quadratic, non-convex, high-dimensional
- Potentially, many minima and flat regions
- No guarantee that
  - Network will converge to a good solution
  - Convergence is swift
  - Convergence occurs at all

Other Challenges[5]

- Ill-conditioning[4]
- Inexact gradients
- Poor correspondence between local and global structure
- Choosing learning rate, other parameters

[4] ftp://ftp.sas.com/pub/neural/illcond/illcond.html

[5] http://www.deeplearningbook.org/contents/optimization.html

How to address?

Algorithmic Approaches

- Batch GD, SGD, Mini-batch SGD
- Momentum, Nesterov Momentum
- Adagrad, Adadelta, RMSProp, Adam
- Advanced Optimization Methods

Part 1

Practical Tricks

- Regularization Methods (including DropOut)
- Data Manipulation Methods
- Parameter Choices/Initialization Methods (Activation Functions, Loss Functions, Weights)

Part 2

Recent Research

- No bad local minima[6]
- What makes deep neural networks generalize? An optimization perspective[7]
- Implicit bias of SGD[8]
- Other directions: SGD happens in subspaces[9]

[6] Choromanska, Mathieu & LeCun, "The Loss Surface of Multilayer Nets", AISTATS'2015; Kawaguchi, "Deep Learning without Poor Local Minima", NIPS 2016; Kawaguchi & Kaelbling, "Every Local Minimum is a Global Minimum of an Induced Model", 2019

[7] Kawaguchi et al, "Generalization in Deep Learning", 2017; Zhang et al, "Understanding deep learning requires rethinking generalization", ICLR 2018; Neyshabur et al, "Exploring Generalization in Deep Learning", 2017

[8] Gunasekar et al, "Characterizing Implicit Bias in Terms of Optimization Geometry", ICML 2018; "Implicit Bias of Gradient Descent on Linear Convolutional Networks", NeurIPS 2018; "The Implicit Bias of Gradient Descent on Separable Data", JMLR 2018

[9] Gur-Ari et al, "Gradient Descent Happens in a Tiny Subspace", arXiv 2018

Outline

Batch GD, Stochastic GD and Mini-Batch SGD

- Batch GD: Update the parameters after the gradients are computed for the entire training set
- Stochastic GD: Randomly shuffle the training set, and update the parameters after gradients are computed for each training example
- Mini-Batch Stochastic GD: Update the parameters after gradients are computed for a randomly drawn mini-batch of training examples *(this is the default option today)*

Batch GD, Stochastic GD and Mini-Batch SGD

Advantages of SGD

- Usually much faster than batch learning. Why? Redundancy in batch learning
- Often results in better solutions. Why? Noise can help!
- Can be used for tracking changes. Why and how? Some systems can change over time.

Issues with SGD

- Noise in SGD weight updates – can lead to no convergence!
- Can be controlled using learning rate
- Equivalent to use of "mini-batches" in SGD (Start with a small batch size and increase size as training proceeds)

Batch GD, Stochastic GD and Mini-Batch SGD

Advantages of Batch GD

- Conditions of convergence are well understood.
- Many acceleration techniques (e.g. conjugate gradient) only operate in batch learning.
- Theoretical analysis of the weight dynamics and convergence rates are simpler.

Mini-batch SGD is the most commonly used method, with a mini-batch size of 20 or so (can be higher depending on dataset size).

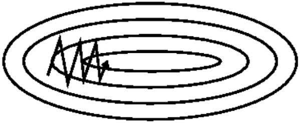
Review: Some keywords to keep track of

1. Loss function: Error used for backpropagation
2. Activation function: A (typically non-linear) function applied on output of neurons
3. Iteration: equivalent to when a weight update is done (could be after every training example, or after a batch of training examples, or after the entire training set
4. Epoch: When the entire training set has been used once to update the weights (Note: we have to run training for many such epochs to train deep networks!)
5. Learning rate (α): Size of the step in the direction of the negative gradient
6. Batch size: Size of a mini-batch when using SGD

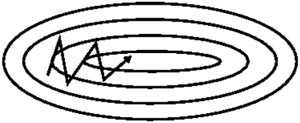
Momentum

Weight update given by:

$$\Delta\theta_{t+1} = \alpha\nabla_\theta J(\theta_t; x^{(i)}, y^{(i)}) + \overset{\text{Momentum Term}}{\boxed{\gamma\Delta\theta_t}}$$



Without momentum



With momentum

Momentum

Weight update given by:

Momentum Term

$$\Delta\theta_{t+1} = \alpha\nabla_\theta J(\theta_t; x^{(i)}, y^{(i)}) + \boxed{\gamma\Delta\theta_t}$$

- Can increase speed when the cost surface is highly non-spherical
- Damps step sizes along directions of high curvature, yielding a larger effective learning rate along the directions of low curvature
- Larger the γ, more the previous gradients affect the current step
- Generally γ is set to 0.5 until initial learning stabilizes and then increased to 0.9 or higher

Momentum

Weight update given by:

$$\Delta\theta_{t+1} = \alpha \nabla_\theta J(\theta_t; x^{(i)}, y^{(i)}) + \underbrace{\boxed{\gamma \Delta\theta_t}}_{\text{Momentum Term}}$$

- Can increase speed when the cost surface is highly non-spherical
- Damps step sizes along directions of high curvature, yielding a larger effective learning rate along the directions of low curvature
- Larger the γ, more the previous gradients affect the current step
- Generally γ is set to 0.5 until initial learning stabilizes and then increased to 0.9 or higher

SGD with Momentum

Require: Learning rate α, momentum parameter γ, minibatch size m, Initial weights θ_t

while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set
  Compute gradient estimate $\nabla_\theta \Sigma_{i=1}^{m} J(\theta_t; x^{(i)}, y^{(i)})$
  Compute update $\Delta\theta_{t+1} = \alpha \nabla_\theta J + \gamma \Delta\theta_t$
  Apply update $\theta_{t+1} = \theta_t - \Delta\theta_{t+1}$
end while

Momentum Update: Alternate View

Weight update given by:

$$\underbrace{\mathbf{v}_{t+1}}_{\text{Velocity vector}} = \gamma \underbrace{\mathbf{v}_t}_{\text{Past velocity vector}} + \alpha \nabla_\theta J(\theta_t; x^{(i)}, y^{(i)})$$

$$\theta_{t+1} = \theta_t - \mathbf{v}_{t+1}$$

Nesterov Accelerated Momentum

- Introduced by Sutskever in ICML 2013
- Based on Nesterov's Accelerated Gradient Descent published in 1983
- Weight update given by:

$$\mathbf{v}_{t+1} = \gamma \mathbf{v}_t + \alpha \nabla_\theta J(\theta_t - \gamma \mathbf{v}_t; x^{(i)}, y^{(i)})$$

$$\theta_{t+1} = \theta_t - \mathbf{v}_{t+1}$$

Can you spot the difference?

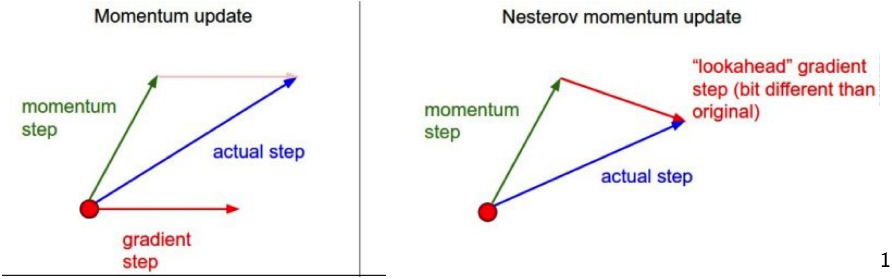
Nesterov Accelerated Momentum

- Weight update given by:

$$\mathbf{v}_{t+1} = \gamma \mathbf{v}_t + \alpha \nabla_\theta J(\theta_t - \gamma \mathbf{v}_t; x^{(i)}, y^{(i)})$$

$$\theta_{t+1} = \theta_t - \mathbf{v}_{t+1}$$

- Empirically found to give good performance



[10]

[10] Image courtesy: Fei-Fei Li course on CNNs, Stanford

SGD with Nesterov Momentum

Require: Learning rate α, momentum parameter γ, minibatch size m, Initial weights θ_t, Initial velocity $\mathbf{v}_t$

while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set
    Apply interim update $\tilde{\theta}_t = \theta_t - \gamma \mathbf{v}_t$
    Compute gradient estimate at interim weights $\nabla_\theta \Sigma_{i=1}^m J(\tilde{\theta}_t; x^{(i)}, y^{(i)})$
    Compute update $\mathbf{v}_{t+1} = \gamma \mathbf{v}_t + \alpha \nabla_\theta \Sigma_{i=1}^m J(\tilde{\theta}_t; x^{(i)}, y^{(i)})$
    Apply update $\theta_{t+1} = \theta_t - \mathbf{v}_{t+1}$
end while

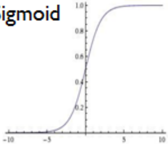
Review: Some keywords to keep track of

1. Loss function: Error used for backpropagation
2. Activation function: A (typically non-linear) function applied on output of neurons
3. Iteration: equivalent to when a weight update is done (could be after every training example, or after a batch of training examples, or after the entire training set
4. Epoch: When the entire training set has been used once to update the weights (Note: we have to run training for many such epochs to train deep networks!)
5. Learning rate (α): Size of the step in the direction of the negative gradient
6. Batch size: Size of a mini-batch when using SGD
7. Momentum parameter (γ): Weightage given to earlier steps taken in the process of gradient descent

Outline

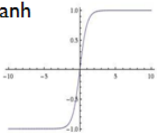
Activation Functions

Sigmoid



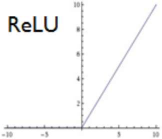
$$y = \frac{1}{1+e^{-x}}$$

tanh



$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

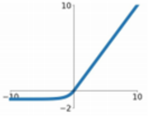
ReLU



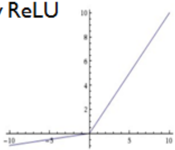
$$y = max(0, x)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Leaky ReLU



$$y = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{if } otherwise \end{cases}$$

maxout

$$max(w_1^T x + b_1, w_2^T x + b_2)$$

Softmax activation function: $a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$ (typically used in output layer)

Activation Functions

- ReLUs the default option today
- The dying ReLU problem → the leaky ReLU
- Found to accelerate convergence of SGD compared to sigmoid/tanh functions (a factor of 6) in AlexNet
- Compared to tanh/sigmoid neurons that involve expensive operations (e.g. exponentials), can be implemented by simply thresholding a matrix of activations at zero.
- MaxOut → a generalization of ReLUs

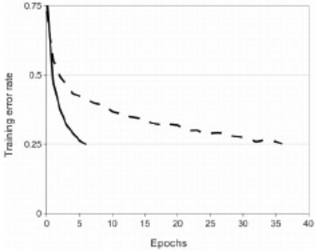


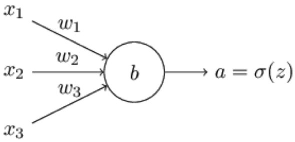
Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons

Activation Functions: Which one to choose?[11]

- Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of "dead" units in a network.
- If this concerns you, give Leaky ReLU or Maxout a try.
- Sigmoid doesn't work well for vision applications.
- Try tanh, but expect it to work worse than ReLU/Maxout.

[11] As advised by Fei-Fei Li (Stanford) in her course "CNNs for Visual Recognition"

Loss Functions: Beyond Mean Square Error



- Cross-Entropy Loss Function: Most popular for classification
- Given by:

$$J = -\frac{1}{m}\sum_{i=1}^{m} y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)$$

- When the activation function is sigmoid $(\sigma(x) = \frac{1}{1+e^{-x}})$, the derivative of cross-entropy loss function, $\frac{\partial J}{\partial w_j}$ becomes:

$$= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial w_j}$$

$$= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\sigma'(z)x_j$$

Loss Functions: Cross-Entropy

$$\frac{\partial J}{\partial w_j} =$$

$$= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial w_j}$$

$$= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\sigma'(z)x_j$$

$$= \frac{1}{n}\sum_x \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z)-y)$$

$$= \frac{1}{n}\sum_x x_j(\sigma(z)-y)$$

Loss Functions: Negative Log-Likelihood

- $J = -\sum_{i=1}^{m} \log P(y_i = \hat{y}_i | x_i, \theta)$
- Assuming a softmax activation function: $a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$
- Gradient of negative log likelihood w.r.t softmax activation function[12]:

Similar to cross-entropy

$$\frac{\partial J}{\partial w_{jk}^L} = \boxed{a_k^{L-1}(a_j^L - y_j)}$$

[12] http://neuralnetworksanddeeplearning.com/chap3.html

Other Loss Functions: Examples from Torch

Classification

- BCECriterion: binary cross-entropy for Sigmoid (two-class version)
- ClassNLLCriterion: negative log-likelihood (multi-class)
- MarginCriterion: two class margin-based loss

Regression

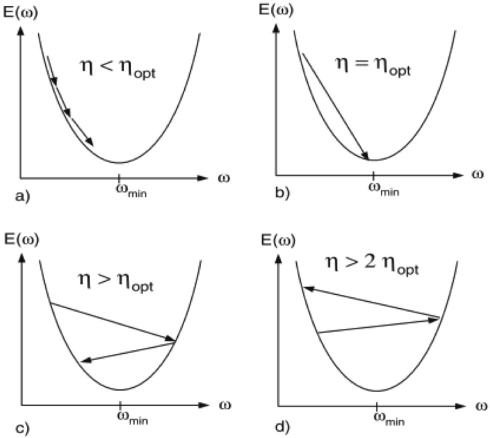
- AbsCriterion: measures the mean absolute value of the element-wise difference between input
- MSECriterion: mean square error (a classic)
- DistKLDivCriterion: Kullback–Leibler divergence (for fitting continuous probability distributions)

Embedding (measuring whether two inputs are similar or dissimilar)

- L1HingeEmbeddingCriterion: L1 distance between two inputs
- CosineEmbeddingCriterion: cosine distance between two inputs

Choosing a learning rate α

What's the optimal learning rate?



Choosing a learning rate α

- If error surface is quadratic and convex, optimal learning rate is given by $\alpha_{opt} = (\nabla_\theta^2 J)^{-1}$, the inverse of the second-derivative of the error function w.r.t. the weights (assuming 1-D weights). Why?

Choosing a learning rate α

- If error surface is quadratic and convex, optimal learning rate is given by $\alpha_{opt} = (\nabla_\theta^2 J)^{-1}$, the inverse of the second-derivative of the error function w.r.t. the weights (assuming 1-D weights). Why?
- In higher-dimensions, the optimal learning rate along each dimension will be given w.r.t. the eigenvalues of the (diagonalized) Hessian
- Largest learning rate that can be used without causing divergence: $\alpha_{max} = 2\alpha_{opt}$

Adaptive Learning Rate Methods: Adagrad

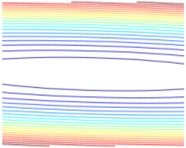
Require: Global learning rate α, minibatch size m, Initial weights θ_t, Small constant, δ, perhaps 10^{-7}, for numerical stability

1: Initialize gradient accumulation variable $\mathbf{r}{=}\mathbf{0}$
2: **while** stopping criterion not met **do**
3: Sample a minibatch of m examples from the training set
4: Compute gradient estimate $\nabla_\theta \Sigma_{i=1}^m J(\theta_t; x^{(i)}, y^{(i)})$
5: Accumulate squared gradient $\mathbf{r} = \mathbf{r} + (\nabla_\theta J \odot \nabla_\theta J)$
6: Compute update $\Delta\theta_{t+1} = \frac{\alpha}{\delta + \sqrt{\mathbf{r}}} \odot \nabla_\theta J$ (division and square root computed elementwise)
7: Apply update $\theta_{t+1} = \theta_t - \Delta\theta_{t+1}$
8: **end while**

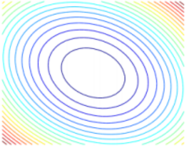
Adagrad: What's the intuition?

- Calculates a different learning rate for each feature
- Sparse features have higher learning rate

Why adapt to geometry?



Hard



Nice

y_t	$\phi_{t,1}$	$\phi_{t,2}$	$\phi_{t,3}$
1	1	0	0
-1	.5	0	1
1	-.5	1	0
-1	0	0	0
1	.5	0	0
-1	1	0	0
1	-1	1	0
-1	-.5	0	1

1. Frequent, irrelevant
2. Infrequent, predictive
3. Infrequent, predictive

[a]

[a] http://seed.ucsd.edu/mediawiki/images/6/6a/Adagrad.pdf

RMSProp

Require: Global learning rate α, Decay rate ρ, Minibatch size m, Initial weights θ_t, Small constant, δ, usually 10^{-6}, for numerical stability

```
Initialize accumulation variable r=0
while stopping criterion not met do
    Sample a minibatch of m examples from the training set
    Compute gradient estimate ∇_θ Σ_{i=1}^m J(θ_t; x^(i), y^(i))
    Accumulate squared gradient r = ρr + (1 − ρ)(∇_θ J ⊙ ∇_θ J)
    Compute update Δθ_{t+1} = α/(δ+√r) ⊙ ∇_θ J (division and square root computed elementwise)
    Apply update θ_{t+1} = θ_t − Δθ_{t+1}
end while
```

while stopping criterion not met **do**: Accumulate squared gradient $\mathbf{r} = \rho\mathbf{r} + (1-\rho)(\nabla_\theta J \odot \nabla_\theta J)$; Compute update $\Delta\theta_{t+1} = \frac{\alpha}{\delta+\sqrt{\mathbf{r}}} \odot \nabla_\theta J$ (division and square root computed elementwise); Apply update $\theta_{t+1} = \theta_t - \Delta\theta_{t+1}$ — gradient estimate $\nabla_\theta \Sigma_{i=1}^{m} J(\theta_t; x^{(i)}, y^{(i)})$. **end while**

What's the intuition? Pretty straightforward from the algorithm and knowledge of Adagrad

RMSProp with Nesterov Momentum

Require: Global learning rate α, Decay rate ρ, Momentum co-efficient γ, Initial velocity $\mathbf{v}$, Minibatch size m, Initial weights θ_t, Small constant, δ, usually 10^{-6}, for numerical stability

1: Initialize accumulation variable $\mathbf{r}$=$\mathbf{0}$
2: **while** stopping criterion not met **do**
3: Sample a minibatch of m examples from the training set
4: Apply interim update $\tilde{\theta}_t = \theta_t - \gamma \mathbf{v}_t$
5: Compute gradient estimate $\nabla_\theta \Sigma_{i=1}^m J(\tilde{\theta}_t; x^{(i)}, y^{(i)})$
6: Accumulate squared gradient $\mathbf{r} = \rho \mathbf{r} + (1-\rho)(\nabla_\theta J \odot \nabla_\theta J)$
7: Compute update $\Delta\theta_{t+1} = \frac{\alpha}{\delta + \sqrt{\mathbf{r}}} \odot \nabla_\theta J$ (division and square root computed elementwise)
8: Apply update $\theta_{t+1} = \theta_t - \Delta\theta_{t+1}$
9: **end while**

Adam

Require: Global learning rate α, Decay rates for moment estimates ρ_1 and ρ_2, Minibatch size m, Initial weights θ_t, Small constant, δ, usually 10^{-8}, for numerical stability

```
Initialize 1st and 2nd moment variables r=0 and s=0
while stopping criterion not met do
    Sample a minibatch of m examples from the training set
    Compute gradient estimate ∇_θ Σ_{i=1}^m J(θ_t; x^(i), y^(i))
    Update biased first moment estimate s = ρ_1 s + (1 − ρ_1) ∇_θ J
    Update biased second moment estimate r = ρ_2 r + (1 − ρ_2)(∇_θ J ⊙ ∇_θ J)
    Correct bias in first moment: s̃ = s / (1 − ρ_1^t)
    Correct bias in second moment: r̃ = r / (1 − ρ_2^t)
    Compute update Δθ_{t+1} = α s̃ / (δ + √r̃) (division and square root computed
    elementwise)
    Apply update θ_{t+1} = θ_t − Δθ_{t+1}
end while
```

Adam

What's the intuition?

- Similar to RMSProp with momentum
- Uses the idea of momentum, as well as having a different learning rate for each dimension (which is automatically adjusted, as in Adagrad, Adadelta or RMS)

Another method in this list, Adadelta, is homework! The idea is similar to methods you have seen so far. Works reasonably well, but not that popular.

Ignorance is bliss: Now, which one to choose?

Visualization: http://sebastianruder.com/optimizing-gradient-descent/

- If input data is sparse, adaptive learning-rate methods may be best.
  - Additional benefit: No need to tune learning rate
- Learning rates diminish fast in Adagrad $\rightarrow$ RMSProp addresses this issue
- Adam adds bias-correction and momentum to RMSprop
- RMSprop, Adadelta, and Adam are similar algorithms $\rightarrow$ bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser
- Adam might be the best overall choice (May not be always true!)

Ignorance is bliss: Now, which one to choose?

Visualization: http://sebastianruder.com/optimizing-gradient-descent/

- If input data is sparse, adaptive learning-rate methods may be best.
  - Additional benefit: No need to tune learning rate
- Learning rates diminish fast in Adagrad $\rightarrow$ RMSProp addresses this issue
- Adam adds bias-correction and momentum to RMSprop
- RMSprop, Adadelta, and Adam are similar algorithms $\rightarrow$ bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser
- Adam might be the best overall choice (May not be always true!)

Ignorance is bliss: Now, which one to choose?

Visualization: http://sebastianruder.com/optimizing-gradient-descent/

- If input data is sparse, adaptive learning-rate methods may be best.
  - Additional benefit: No need to tune learning rate
- Learning rates diminish fast in Adagrad $\rightarrow$ RMSProp addresses this issue
- Adam adds bias-correction and momentum to RMSprop
- RMSprop, Adadelta, and Adam are similar algorithms $\rightarrow$ bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser
- Adam might be the best overall choice (May not be always true!)

Ignorance is bliss: Now, which one to choose?

Visualization: http://sebastianruder.com/optimizing-gradient-descent/

- If input data is sparse, adaptive learning-rate methods may be best.
  - Additional benefit: No need to tune learning rate
- Learning rates diminish fast in Adagrad $\rightarrow$ RMSProp addresses this issue
- Adam adds bias-correction and momentum to RMSprop
- RMSprop, Adadelta, and Adam are similar algorithms $\rightarrow$ bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser
- Adam might be the best overall choice (May not be always true!)

Ignorance is bliss: Now, which one to choose?

Visualization: http://sebastianruder.com/optimizing-gradient-descent/

- If input data is sparse, adaptive learning-rate methods may be best.
  - Additional benefit: No need to tune learning rate
- Learning rates diminish fast in Adagrad $\rightarrow$ RMSProp addresses this issue
- Adam adds bias-correction and momentum to RMSprop
- RMSprop, Adadelta, and Adam are similar algorithms $\rightarrow$ bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser
- Adam might be the best overall choice (May not be always true!)

Some notes to keep in mind

- Vanilla SGD depends on a robust initialization, and may get stuck in saddle points rather than local minima
- In general, adaptive learning rate methods may be the way to go
- Many recent papers use vanilla SGD without momentum, but with a simple learning rate annealing schedule

Some notes to keep in mind

- Vanilla SGD depends on a robust initialization, and may get stuck in saddle points rather than local minima
- In general, adaptive learning rate methods may be the way to go
- Many recent papers use vanilla SGD without momentum, but with a simple learning rate annealing schedule

Some notes to keep in mind

- Vanilla SGD depends on a robust initialization, and may get stuck in saddle points rather than local minima
- In general, adaptive learning rate methods may be the way to go
- Many recent papers use vanilla SGD without momentum, but with a simple learning rate annealing schedule

Outline

Takeaways

- Some standard choices for training deep networks: SGD + Nesterov momentum, SGD with Adagrad/RMSProp/Adam
- ReLUs, Leaky ReLUs and MaxOut are the best bets for activation functions

Challenges in GD: How to address?

Algorithmic Approaches

- Batch GD, SGD, Mini-batch SGD
- Momentum, Nesterov Momentum
- Adagrad, Adadelta, RMSProp, Adam
- Advanced Optimization Methods

So far

Practical Tricks

- Regularization Methods (including DropOut)
- Data Manipulation Methods
- Parameter Choices/Initialization Methods (Activation Functions, Loss Functions, Weights)

Now

Outline

Difference between Machine Learning and Optimization

- Thoughts?

Difference between Machine Learning and Optimization

- Thoughts? Generalization!
- In mainstream optimization, minimizing J is itself the goal; whereas in deep learning, minimizing J so as to minimize a generalizable out-of-sample performance measure is the goal
- Empirical Risk Minimization (ERM):

$$\mathbb{E}_{\mathbf{x},y\approx\hat{p}_{data}(\mathbf{x},y)}(J(\theta;\mathbf{x},y)) = \frac{1}{m}\sum_{i=1}^{m} J(\theta;\mathbf{x}_i,y_i)$$

- However, ERM can lead to overfitting. Avoiding overfitting is regularization.

Learning and Generalization

What's my rule?

- 1 2 3 $\Longrightarrow$ satisfies rule
- 4 5 6 $\Longrightarrow$ satisfies rule
- 7 8 9 $\Longrightarrow$ satisfies rule
- 9 2 31 $\Longrightarrow$ does not satisfy rule

Learning and Generalization

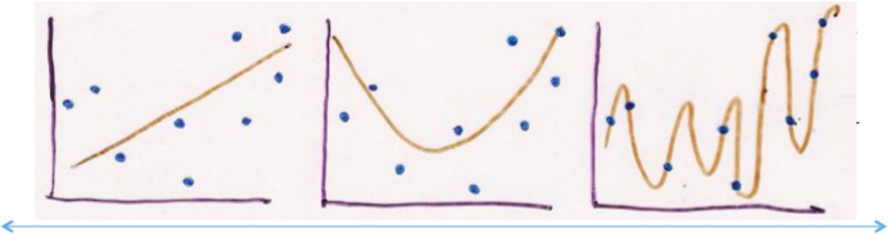
What's my rule?

- 1 2 3 $\Longrightarrow$ satisfies rule
- 4 5 6 $\Longrightarrow$ satisfies rule
- 7 8 9 $\Longrightarrow$ satisfies rule
- 9 2 31 $\Longrightarrow$ does not satisfy rule

Plausible rules

- 3 consecutive single digits
- 3 consecutive integers
- 3 numbers in ascending order
- 3 numbers whose sum is less than 25
- 3 numbers < 10
- 1, 4, or 7 in first column
- "yes" to first 3 sequences, "no" to all others

Overfitting

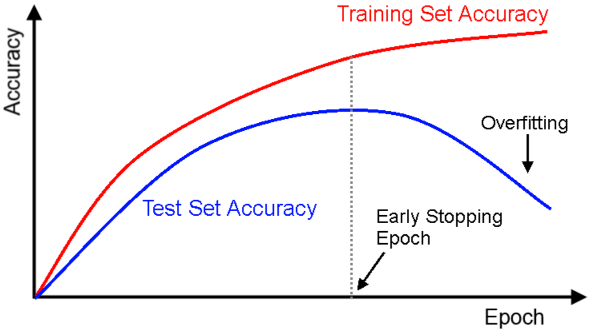


- simple model
- constrained
- small capacity may prevent it from representing all structure in data

- complex model
- unconstrained
- large capacity may allow it to memorize data and fail to capture regularities

Early Stopping

- Simple idea to keep monitoring the cost function, and not let it become too consistently low; stop at an earlier iteration



Early Stopping

When to stop?

- Train n epochs; lower learning rate; train m epochs $\rightarrow$ Bad idea: can't assume one-size-fits-all approach
- *Error-change criterion:*
  - Stop when error isn't dropping over a window of, say, 10 epochs
  - Train for a fixed number of epochs after criterion is reached (possibly with lower learning rate)
- *Weight-change criterion:*
  - Compare weights at epochs t-10 and t and test: $\max_i \|w_i^t - w_i^{t-10}\| < \rho$
  - Don't base on length of overall weight change vector
  - Possibly express as a percentage of the weight

Weight Decay

- We have already seen a regularization method: weight decay in gradient descent

L2-Weight Decay Term

$$J(\theta) = \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_\theta(x^{(i)}) - y^{(i)} \right\|^2 \right) \right] + \boxed{\frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2}$$

L1-Weight Decay Term

$$J(\theta) = \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_\theta(x^{(i)}) - y^{(i)} \right\|^2 \right) \right] + \boxed{\lambda \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left| W_{ji}^{(l)} \right|}$$

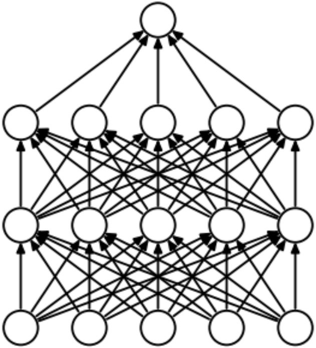
DropOut

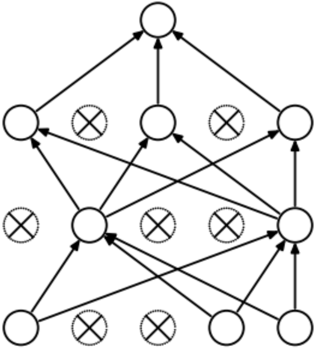
- Another standard approach to regularization in ML: *Model Averaging*
- DropOut $\rightarrow$ a very interesting way to perform model averaging in deep learning

DropOut

- Another standard approach to regularization in ML: *Model Averaging*
- DropOut $\rightarrow$ a very interesting way to perform model averaging in deep learning
- Training Phase: For each hidden layer, for each training sample, for each iteration, ignore (zero out) a random fraction, p, of nodes (and corresponding activations)
- Test Phase: Use all activations, but reduce them by a factor p (to account for the missing activations during training)

DropOut


(a) Standard Neural Net


(b) After applying dropout. [13]

[13] Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

DropOut

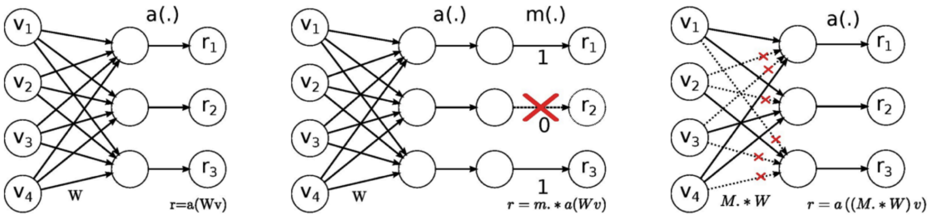
- With H hidden units, each of which can be dropped, we have 2^H possible models
- Each of the 2^{H-1} models that include hidden unit h must share the same weights for the unit
  - serves as a form of regularization
  - makes the models cooperate
- Including all hidden units at test with a scaling of 0.5 is equivalent to computing the geometric mean of all 2^H models [14] [15]

---

[14] Hinton et al, Improving neural networks by preventing co-adaptation of feature detectors, 2012

[15] Warde-Farley et al, An empirical analysis of dropout in piecewise linear networks, 2014

DropConnect: An Extension



No-Drop Network DropOut Network DropConnect Network

a = activation function; m = dropping rate; M = binary mask matrix

[16] Wan, Li, et al. "Regularization of neural networks using dropconnect." ICML 2013

Noise in Data, Label and Gradient

Using noise is another form of regularization; has shown some impressive results recently. Could be:

- Data Noise
  - Has been there for a while: add noise to data while training
  - Minimization of sum-of-squares error with zero-mean gaussian noise(added to training data) is equivalent to minimization of sum-of-squares error without noise with an added regularized term [17]
  - Very similar to data augmentation that we will see later
- Label Noise
- Gradient Noise

[17] Bishop. Training with noise is equivalent to Tikhonov regularization. Neural Computation, 1995.

Regularization through Label Noise[18]

- Disturb each training sample with the probability α.
- For each disturbed sample, label is randomly drawn from a uniform distribution over $\{1, 2, \cdots, C\}$, regardless of the true label.

Algorithm 1 DisturbLabel

Input: $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N}$, noise rate α.
Initialization: a network model $\mathbb{M}$: $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}_0) \in \mathbb{R}^C$;
for each mini-batch $\mathcal{D}_t = \{(\mathbf{x}_m, \mathbf{y}_m)\}_{m=1}^{M}$ **do**
    for each sample $(\mathbf{x}_m, \mathbf{y}_m)$ **do**
        Generate a disturbed label $\widetilde{\mathbf{y}}_m$ with Eqn (2);
    end for
    Update the parameters $\boldsymbol{\theta}_t$ with Eqn (1);
end for
Output: the trained model $\mathbb{M}'$: $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}_T) \in \mathbb{R}^C$.

$$\begin{cases} \widetilde{c} & \sim & \mathcal{P}(\alpha), \\ \widetilde{y}_{\widetilde{c}} & = & 1, \\ \widetilde{y}_{i} & = & 0, \qquad \forall i \neq \widetilde{c}. \end{cases} \tag{2}$$

[18] Xie, Lingxi et al. "DisturbLabel: Regularizing CNN on the Loss Layer". In: Xi… [illegible]

Regularization through Gradient Noise[19]

- Simple idea: add noise to gradient

$$g_t \leftarrow g_t + N(0, \sigma_t^2)$$

- Annealed Gaussian noise by decaying the variance

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma}$$

- Showed significant improvement in performance

[19] Neelakantan, Arvind, et al. "Adding gradient noise improves learning for very deep networks." arXiv preprint arXiv:1511.06807 (2015).

Outline

Data Transformation

- Normalize/standardize the inputs
  - Convergence is faster if average input over the training set is close to zero[20]
- Scaled to have the same covariance - speeds learning
  - Ideally, value of covariance should be matched with output of activation function (e.g. sigmoid)

[20] Le Cun et al, Efficient Backprop, 1998

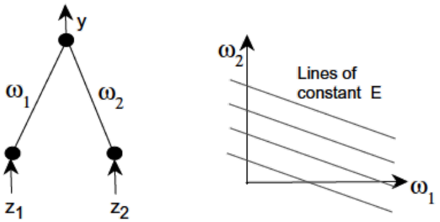
Data Transformation

- Normalize/standardize the inputs
  - Convergence is faster if average input over the training set is close to zero[20]
- Scaled to have the same covariance - speeds learning
  - Ideally, value of covariance should be matched with output of activation function (e.g. sigmoid)

[20] Le Cun et al, Efficient Backprop, 1998

Data Transformation

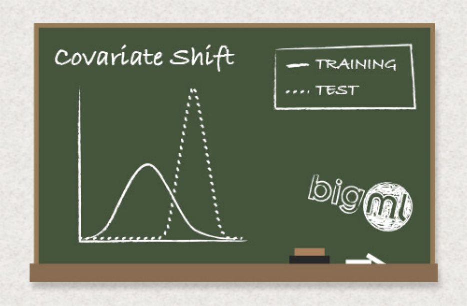
- Decorrelate the inputs
  - Why? Imagine one input is always twice the other, i.e. $z_2 = 2z_1$. Output y will be constant on lines $w_2 + \frac{1}{2}w_1 = \text{const.}$ No use making weight changes on these lines.
  - How? PCA!



Batch Normalization

Covariate Shift

- Change in distributions of data inputs is a problem because the model needs to continuously adapt to the new distribution $\rightarrow$ called covariate shift
- This is typically handled using domain adaptation



Batch normalization[21]

- What if this happens in a subnetwork in DL? $\rightarrow$ called internal covariate shift. How to handle?

[21] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint 2015

Batch normalization[21]

- What if this happens in a subnetwork in DL? $\rightarrow$ called internal covariate shift. How to handle?
- Whiten every layer's inputs $\rightarrow$ helps obtain a fixed distribution of inputs into each layer

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

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

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

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

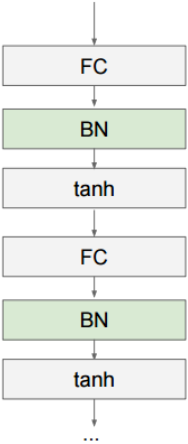
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

[21] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint 2015

Batch normalization



- BN layer usually inserted before non-linearity layer (after FC or convolutional layer)
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization too

How do we handle test time? Evaluate a mini-batch at a time?

Shuffling Inputs[22]

- Choose examples with maximum information content
  - Shuffle the training set so that successive training examples never (rarely) belong to the same class.
- Present input examples that produce a large error more frequently than examples that produce a small error. Why?

[22] LeCun, Yann A., et al. "Efficient backprop." Neural networks: Tricks of the Trade. Springer Berlin Heidelberg, 2012. 9-48.

Shuffling Inputs[22]

- Choose examples with maximum information content
  - Shuffle the training set so that successive training examples never (rarely) belong to the same class.
- Present input examples that produce a large error more frequently than examples that produce a small error. Why? Helps take large steps in the gradient descent
- Do you see any problems?

[22] LeCun, Yann A., et al. "Efficient backprop." Neural networks: Tricks of the Trade. Springer Berlin Heidelberg, 2012. 9-48.

Shuffling Inputs[22]

- Choose examples with maximum information content
  - Shuffle the training set so that successive training examples never (rarely) belong to the same class.
- Present input examples that produce a large error more frequently than examples that produce a small error. Why? Helps take large steps in the gradient descent
- Do you see any problems? What if the data sample is an outlier?

[22] LeCun, Yann A., et al. "Efficient backprop." Neural networks: Tricks of the Trade. Springer Berlin Heidelberg, 2012. 9-48.

Shuffling Inputs[22]

- Choose examples with maximum information content
  - Shuffle the training set so that successive training examples never (rarely) belong to the same class.
- Present input examples that produce a large error more frequently than examples that produce a small error. Why? Helps take large steps in the gradient descent
- Do you see any problems? What if the data sample is an outlier?
- Is this relevant for Batch GD?

[22]LeCun, Yann A., et al. "Efficient backprop." Neural networks: Tricks of the Trade. Springer Berlin Heidelberg, 2012. 9-48.

Curriculum Learning[23]

- Old idea, proposed by Elman in 1993
- Humans and animals learn much better when examples are not randomly presented but organized in a meaningful order which illustrates gradually more concepts, and gradually more complex ones.
- Start small, learn easier aspects of the task or easier sub-tasks, and then gradually increase the difficulty level
- By choosing examples and their order, one can guide training and remarkably increase learning speed
- Introduces the concept of a teacher who:
  - has prior knowledge about the training data to decide on a sequence of concepts that can more easily be learned when presented in that order
  - monitoring ‘learner’s progress to decide when to move on to new material from the curriculum

[23] Bengio, Yoshua, et al. "Curriculum learning." ICML 2009.

Curriculum Learning[23]

- Old idea, proposed by Elman in 1993
- Humans and animals learn much better when examples are not randomly presented but organized in a meaningful order which illustrates gradually more concepts, and gradually more complex ones.
- Start small, learn easier aspects of the task or easier sub-tasks, and then gradually increase the difficulty level
- By choosing examples and their order, one can guide training and remarkably increase learning speed
- Introduces the concept of a teacher who:
  - has prior knowledge about the training data to decide on a sequence of concepts that can more easily be learned when presented in that order
  - monitoring 'learner's progress to decide when to move on to new material from the curriculum

[23] Bengio, Yoshua, et al. "Curriculum learning." ICML 2009.

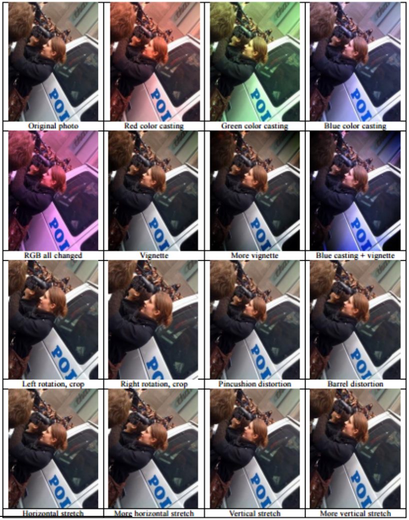
Data Augmentation

Methods

- Data jittering (E.g. Distortion and blurring of images)
- Rotations
- Color changes
- Noise injection
- Mirroring

- Helps increase data; is useful when training data provided is less (CNNs lead large amounts of training data to work!)
- Also acts as a regularizer (by avoiding overfitting to provided data)

Data Augmentation: Example [24]



[24] Wu, Ren, et al. "Deep image: Scaling up image recognition." arXiv 2015

Outline

Parameter Choices

- Activation Functions: We discussed this earlier
- Loss Functions: We discussed this earlier
- Learning Rates: We discussed this earlier
  - All of them decrease it when weight vector "oscillates", and increase it when the weight vector follows a relatively steady direction
  - Worthwhile picking a different learning rate for each weight (e.g. based on curvature)

Choosing Target Values

- Assuming a binary classification problem, what do you choose the target labels to be? $+1$ and -1?

Choosing Target Values

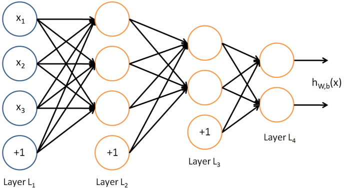
- Assuming a binary classification problem, what do you choose the target labels to be? $+1$ and -1?
- What if these are the sigmoid's asymptotes?
  - Weights will be increased continuously to very high values to match the target
  - Weights multiplied by small sigmoid derivative $\rightarrow$ small weight updates $\rightarrow$ Stuck!

Choosing Target Values

- Assuming a binary classification problem, what do you choose the target labels to be? $+1$ and -1?
- What if these are the sigmoid's asymptotes?
  - Weights will be increased continuously to very high values to match the target
  - Weights multiplied by small sigmoid derivative $\rightarrow$ small weight updates $\rightarrow$ Stuck!
- Choose target values at the point of the maximum second derivative on the sigmoid so as to avoid saturating the output units.

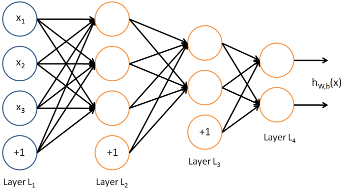
Weight Initialization

- What do you think? What if we started weights with zeroes?



Weight Initialization

- What do you think? What if we started weights with zeroes?



- To be chosen randomly, but in such a way that the activation function is in its linear region
  - Both large and small weights can cause very low gradients (in case of sigmoid activation)

Weight Initialization

Most recommended today (removed the need for unsupervised pre-training):

- Xavier's initialization[25]: $\texttt{uniform}\left(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}\right)$
- Caffe implements a simpler version of Xavier's initialization as: $\texttt{uniform}\left(-\frac{2}{fan_{in}+fan_{out}}, \frac{2}{fan_{in}+fan_{out}}\right)$
- He's initialization[26]: $\texttt{uniform}\left(-\frac{4}{fan_{in}+fan_{out}}, \frac{4}{fan_{in}+fan_{out}}\right)$

---

[25] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." AISTATS 2010

[26] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." CVPR 2015

Weight Initialization

Still an active area of research...

- Understanding the difficulty o training deep feedforward neural networks by Glorot and Bengio, 2010
- Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013
- Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014
- Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015
- Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015
- All you need is a good init, Mishkin and Matas, 2015

Outline

Takeaways

- Some standard choices for training deep networks: SGD + Nesterov momentum, SGD with Adagrad/RMSProp/Adam
- ReLUs, Leaky ReLUs and MaxOut are the best bets for activation functions
- Batch Normalization layers are here to stay (at least, for now)
- DropOut is an excellent regularizer
- Data Augmentation is a must in vision applications
- Weight Initialization is very important while training a new network

Readings

- Deep Learning book, Sections 7.1-7.5, 7.8, 7.12: http://www.deeplearningbook.org/contents/regularization.html
- Deep Learning book, Sections 8.1-8.5: http://www.deeplearningbook.org/contents/optimization.html
- Efficient Backprop by Yann Le Cun, 1998: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf