Gradient Descent and Optimization

Monday, December 13, 2021 9:16 PM

18. Gradient Descent and Optimization

Optimization

- define cost func (objective func) to measure quality of soln
  - minimize diff bw model prediction and labels
- use optimizer to search/solve for a soln that minimizes/maximizes cost func
  - iteratively search for good solution

Training NN Classifier

- objective: minimize diff bw label y and predicted label y_hat
- prediction = func of input x, params w,b
- training objective = func of params w,b, sample x, label y
- for training set, x,y are const. (can exclude b for now)

$$J = f(y, \hat{y})$$

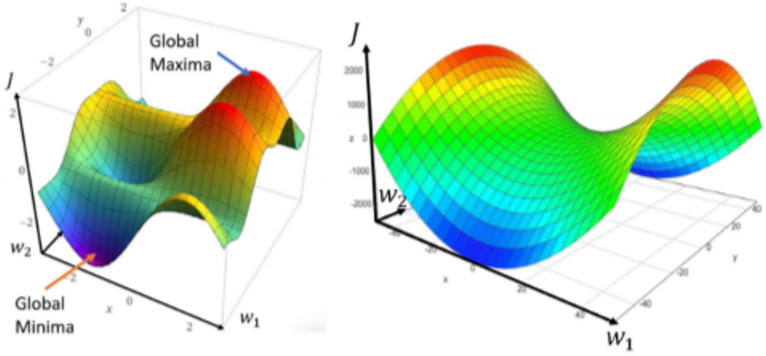
$$\hat{y} = h(x, w, b)$$

$$J = f(y, x, w, b)$$

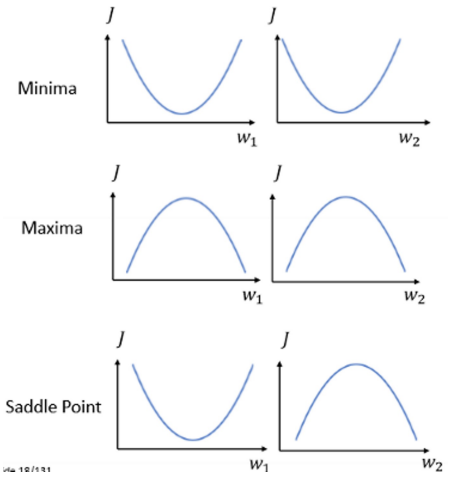
$$J = f(w, b)$$

$$J = f(w)$$

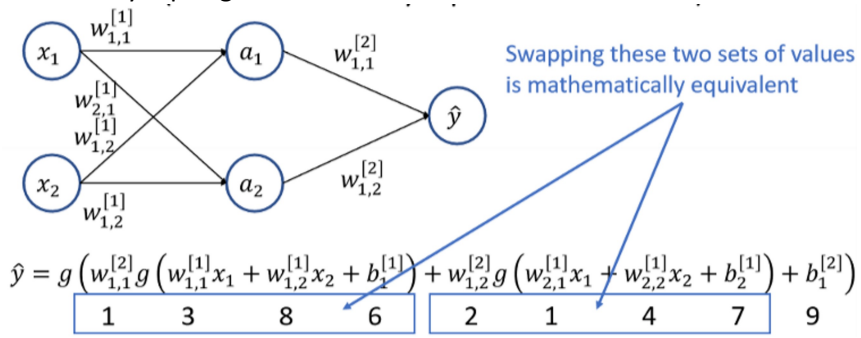
- for NN and training set, cost func J(w) is a func of only w
- **Global Optima**
  - Optima: maxima and minima
  - Biggest/smallest among maxima/minima
  - local maxima/minima refers to the rest



- gradients at optima and saddle points = 0, **dJ/dw_i = 0** for all params w_i

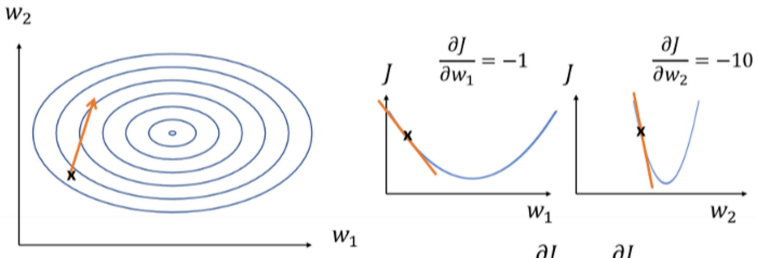


- DL Cost function is not convex
    - many equal global minimal



Gradient Descent

- start somewhere in space
- more in direction with steepest decrease in cost
- repeat
- Steps:
    - Train model
    - evaluate
    - change hyperparam
    - repeat
- **Problem:**
    - too long to compute gradient for one training iteration
    - requires too much memory to store activations
- **Mini-Batch Gradient Descent**
    - use small subset of training set as approx of overall training set
    - Sampling:
        - random shuffle full set
        - partition into mini-batch
        - iterate across each mini-batch
    - one full pass through the set = epoch
    - mini-batch size of 1 = stocahstic gradient descent (SGD)
- **Problem:**
    - **Different dimension change at diff rates**
        - direciton of steepest descent isn't directly to min unless is a circle



Larger steps at steeper areas, and smaller steps at shallower areas

$$\frac{\partial J}{\partial w_1} \ll \frac{\partial J}{\partial w_2}$$

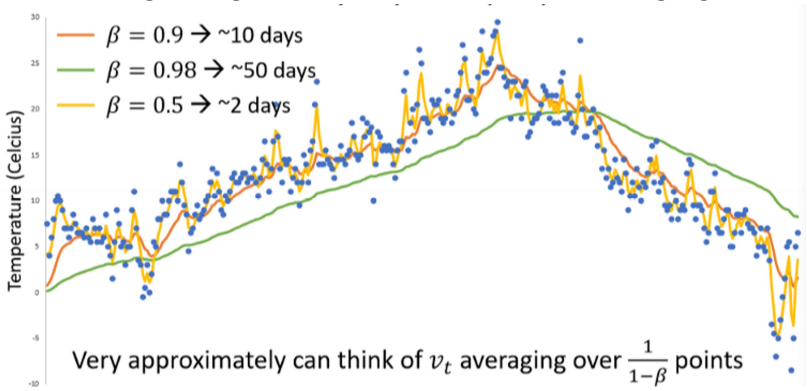
$$w_1 = w_1 - \alpha(-1)$$

$$w_2 = w_2 - \alpha(-10)$$

- **Local optima and Saddle Points**
  - gradient = 0 at local optima and saddle points
  - saddle points are unstable but no gradient info
  - gradient descent will stop updating params
- **Meadering mini-batch gradients**

Gradient Descent with Momentum

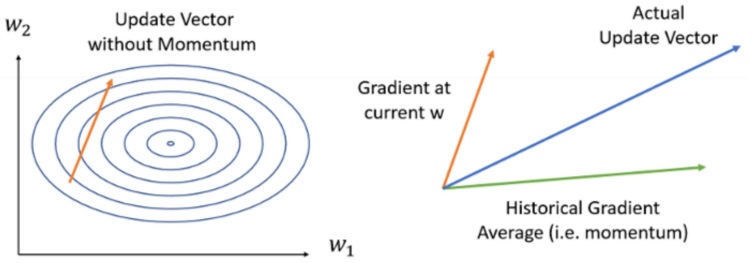
- Use historical data with exp. weighted avg method to get out of 0 gradients saddle point
- **Exponentially weighted averages**
  - moving average, smooth out flunctuations and highlight trends



- **Solution:**
  - instead of using gradient directly to update params, use exp weighted avg of past gradients
- **Problem:**
  - Gradient in diff dimensions
    - consistent gradients build up velocity from accumulated acceleration
    - inconsistent gradients will cancel out
  - Local minima and saddle point
    - at saddle point, gradient = 0, but momentum won't be (historical component)
    - at local minima, velocity can help get back out of local minima
  - Meandering mini-batch gradients
    - moving avg will create smoothing effect

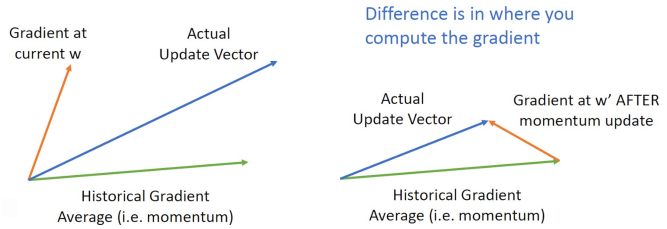
Momentum so far

$$v_t = \beta \cdot v_{t-1} + \frac{\partial J}{\partial w}$$



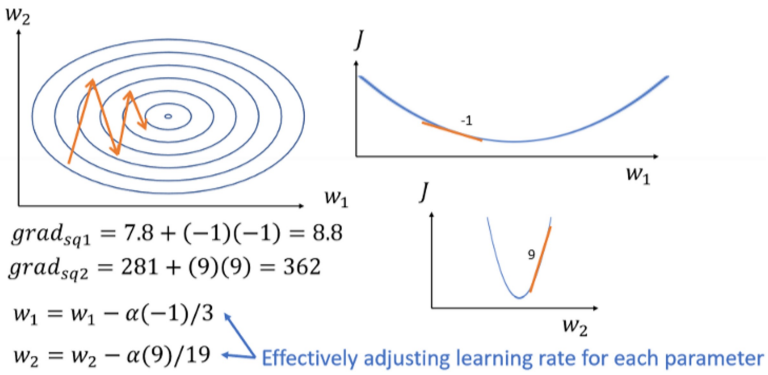
- **Nesterov Momentum**

Classic Momentum Nesterov Momentum



Per-Parameter Adaptive Learning Rates

- to fix adaptive per parameter learning rates, use per-parameter learning rates
- **Adagrad**
  - square of gradients focuses on magnitude, not direction
  - dampens steep gradients and accelerates shallow gradients by dividing by grad_sq



$grad_{sq1} = 7.8 + (-1)(-1) = 8.8$

$grad_{sq2} = 281 + (9)(9) = 362$

$w_1 = w_1 - \alpha(-1)/3$

$w_2 = w_2 - \alpha(9)/19$ ← Effectively adjusting learning rate for each parameter

- **Adam**
  - combines RMSProp and Momentum
  - good default choice for optimizer

Bias Correction for Exponentially weighted avg

- use biased vals in moving avg
- use corrected versions when doing gradient descent parameter update

$$v_{(t)\_biased} = \beta \cdot v_{(t-1)\_biased} + (1-\beta) \cdot T_t$$

$$v_t = \frac{v_{(t)\_biased}}{1-\beta^t}$$ ← Denominator approaches 1 as t increases

- bias correction
  - sorts itself out after several iterations but
  - will make large updates at start which destroy weight initialization
  - may result in no gradients

Keras Optimizers:

- SGD
- RMSProp
- Adagrad
- Adadelta
- Adam
- AdaMax
- Nadam

19. Optimization

Learning Rate Schedules

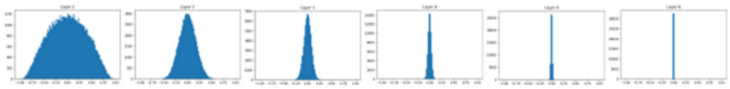
- how to choose leanrning rate: vary over time
  - start high, reduce over time
  - annealing/decaying learning rate
- **Decay learning rate**
  - reduce learning rate once progress plateas
  - Decay too slow = wasting time bounding
  - Decay too fast = slow down training unnecesarily
- decay based on func

- Exponential Decay $\alpha_t = \alpha_0 0.95^t$ $\alpha_t = \alpha_0 e^{-kt}$
- Linear Decay $\alpha_t = \alpha_0 \left(1 - \frac{t}{T}\right)$
- Cosine Decay $\alpha_t = \alpha_0 0.5 \left(1 + \cos\left(\frac{\pi t}{T}\right)\right)$
- Inverse sqrt Decay $\alpha_t = \alpha_0 \frac{1}{\sqrt{t}}$
- 1/t Decay $\alpha_t = \alpha_0 \frac{1}{1 + k * t}$

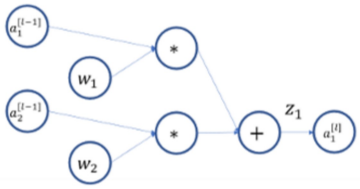
- t – current training iteration
- T – Total training iterations
- α_0– Initial learning rate
- α_t - learning rate at iteration t
- k – a hyperaparameter

Parameter Initialization

- want gradients to be well behaved
  - don't want to start off somewhere where gradients are all 0, not clear where to move
  - don't start close to global minima, initializing with 0s
  - initialize with a constant instead, **gaussian random dist** (breaks symmetry, not all initialized to same value)
  - Gaussian with mean 0, can also expect that final weights might be zero-centered
  - Gaussian random var * x will give rand var a std dev = x
- **Gaussian random dist**



- As you go deeper, activations of the units become 0 or close to 0
- This means the gradients also approach 0



  - for DNN, activation and gradients become 0
  - can change std.dev

Was 0.01 before

```
W = 0.1*np.random.randn(fan_in, fan_out)
```

- need to find ideal variance
  - Xavier initialization (for TanH)
  - set variance = # inputs to layer (fan_in)

```
W = (1/np.sqrt(fan_in))*np.random.randn(fan_in, fan_out)`
```

  - Kaiming/He_normal for ReLU

Initializing Bias

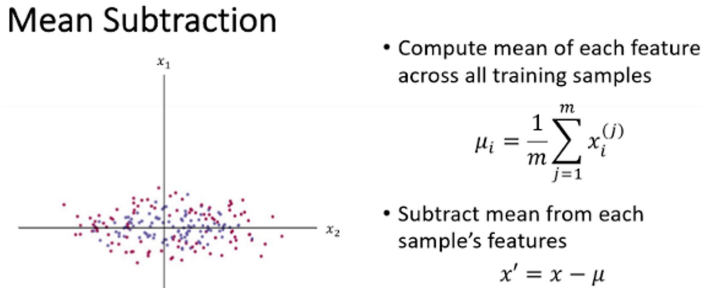
- can initialize with 0s

Gaussian vs Uniform Distribution

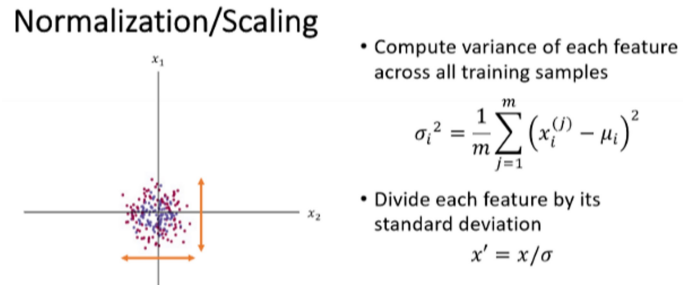
- some init schemes may draw from uniform dist instead of Gaussian

Data Preprocessing

- sigmoid always positive
  - all param updates will be positive
  - inefficient training
  - pick zero-centered activation func
  - 1st layer, inputs are training data, no activation function -> non-zero centered?
- **Mean subtraction**



- **Normalization/Scaling**



  - relative feature scales, corresponding weights become same scale
  - absolute feature scales, don't want scales to be large, leads to large gradients, small change = big change in final cost, harder to optimize
- **Standardization (Z-Score Normalization)**

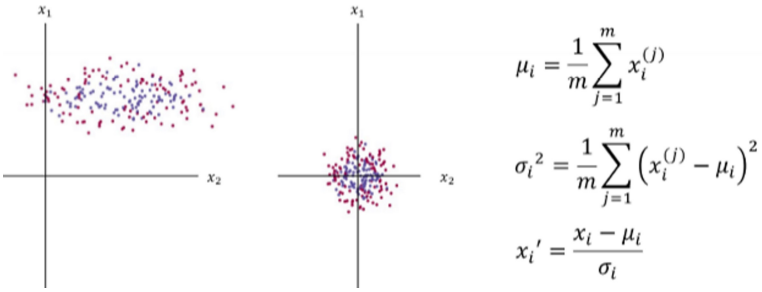


Image data

- each pixel = feature, same scale relative, but may still want to normalize
  - AlexNet
    - Subtracted mean image
    - i.e. computed mean on each feature (pixel) across dataset
  - VGGNet
    - Calculated a mean for each color channel (e.g. μ_r, μ_g, μ_b)
    - Each channel's feature (pixel) subtracted its corresponding channel mean
  - ResNet
    - Subtracted channel mean like VGGNet
    - Also divided by channel standard deviation
- **At inference/prediction time**
  - any transformation performed on input for training performed for inputs supplied at prediction time
  - if used Z-Score Normalization, need to apply same func w/ same mean and std during

prediction time

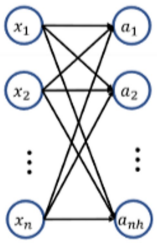
$$x_i' = \frac{x_i - \mu_i}{\sigma_i}$$

Batch Normalization

- Normalizing Network inputs
  - helps training params in 1st layer, optimize
- normalizing inputs of hidden layers
  - inputs to hidden layers are outputs of previous layers, batch normalization
  - func of each layer changing as training progresses
  - hard for intermediate layer optimization, inputs change
  - small change = big change on deeper layer
  - batch normalization helps stabilize optimization problem by giving layers target mean and variance

- For a given mini-batch with m samples, x is a matrix of shape (n, m)
- For each input x_i, compute its
  - mean μ_i across the mini-batch → n means
  - variance ${\sigma_i}^2$ across the mini-batch → n variances

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \qquad {\sigma_i}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$



- For each sample and each feature, normalize → (n, m)

$$x_i' = \frac{x_i - \mu_i}{\sigma_i}$$

- forcing each layer's outputs to have zero-mean and unit variance too strict
- let model learn target mean and variance for each layer

Learned mean and variance

- Two new trainable parameters, γ_i, β_i for each layer output that act to shift and scale the normalized layer outputs
- Normalize like before

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \qquad {\sigma_i}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2 \qquad x_i' = \frac{x_i - \mu_i}{\sigma_i}$$

- Now shift and scale normalized values → (n, m)

$$\tilde{x_i} = \gamma_i x_i' + \beta_i$$

- if $\gamma_i = \sigma_i$ and $\beta_i = \mu_i$, then $\tilde{x_i} = x_i$ and hence $\tilde{x_i}$ has the same mean and variance as the original input distribution
- if $\gamma_i = 1$ and $\beta_i = 0$, then $\tilde{x_i} = x_i'$ and hence $\tilde{x_i}$ has 0 mean and unit variance
- For other values of γ_i and β_i, x_i will have some other mean and variance

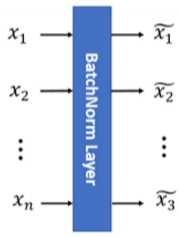
Batch Norm layer

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$

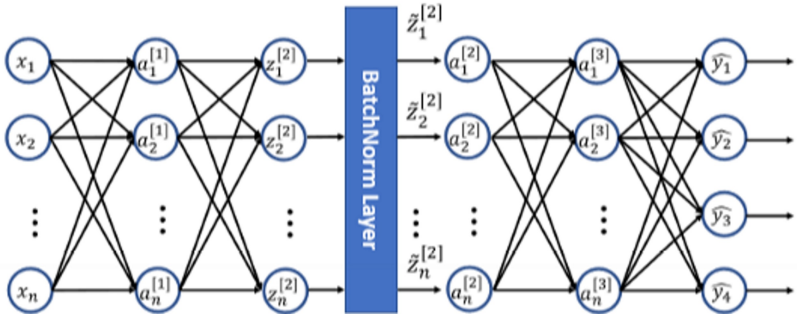
$$\sigma_i{}^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

$$x_i{}' = \frac{x_i - \mu_i}{\sigma_i}$$

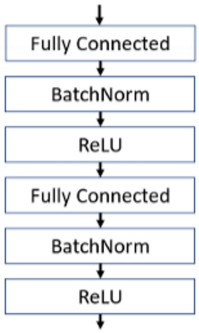
$$\widetilde{x_i} = \gamma_i x_i' + \beta_i$$



- put layer before nonlinear activation

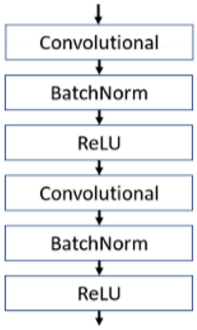


- **BatchNorm and FCL**



Term	Symbol	Shape
BatchNorm Inputs	x	(n, m)
Mean	μ	$(n,)$
Stddev/Variance	σ	$(n,)$
Scale param	γ	$(n,)$
Shift param	β	$(n,)$
Normalized Inputs	x'	(n, m)
Batch Normalized Inputs	$\tilde{x}$	(n, m)

- **BatchNorm and Conv Layer**

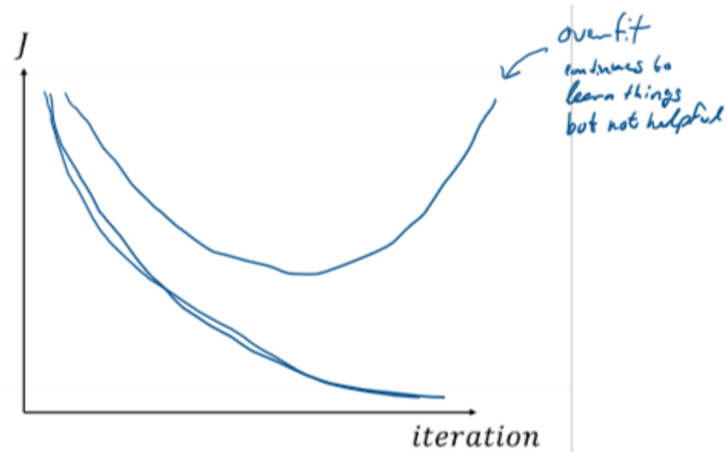


Term	Symbol	Shape
BatchNorm Inputs	x	(h, w, c, m)
Mean	μ	$(c,)$
Stddev/Variance	σ	$(c,)$
Scale param	γ	$(c,)$
Shift param	β	$(c,)$
Normalized Inputs	x'	(h, w, c, m)
Batch Normalized Inputs	$\tilde{x}$	(h, w, c, m)

- At prediction time, keep moving average (exp weighted average), use these avg vals instead of u and sigma during prediction time

20. Regularization and Training

Overfitting



- model continues to learn things that does not help
- Need:
    - more training data
    - regularization

Regularization via Cost func

- cost/obj func describes quality we want in our soln (param values)
- so far cost func only accounts for prediction loss
- can add terms to encourage regularization in our soln

$$J = \frac{1}{m}\sum_{j=1}^{m} L(\hat{y}, y)$$

Cost function so far

$$J = \left(\frac{1}{m}\sum_{j=1}^{m} L(\hat{y}, y)\right) + R$$

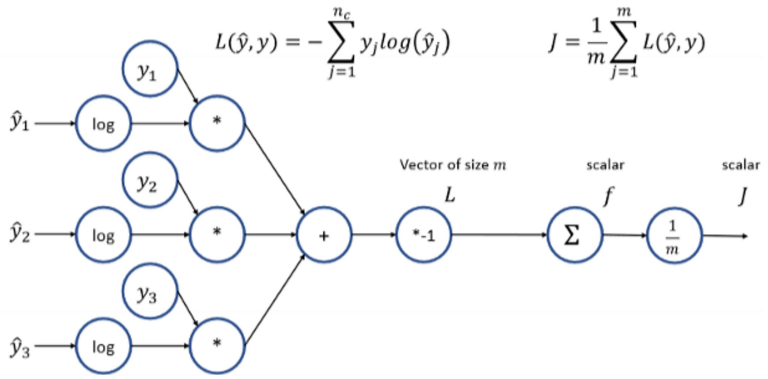
Cost function with Regularization Term

- **L2 Regularization**
    - sum the square of each param value
    - cost is minimized when each param value is small
    - convex func independent of training set
        - regularization term is global min when all weights = 0

$$J = \frac{1}{m}\sum_{j=1}^{m} L(\hat{y}, y) + \sum w^2$$

    - try to minimize the loss AND regularization terms
        - loss term = large if all weights = 0
    - Since loss and regularization are competing terms, specify importance of each term based on lambda hyper param
        - lambda = 0: we don't optimize for regularization
        - lambda = inf: we don't optimize for loss
        - default in Keras = 0.01

$$J = \frac{1}{m}\sum_{j=1}^{m} L(\hat{y}, y) + \lambda \sum w^2$$

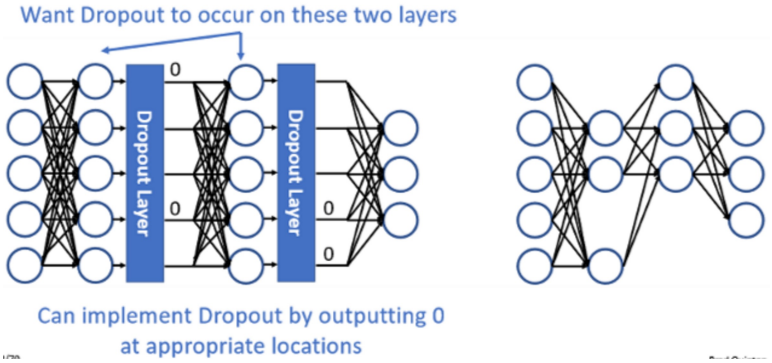


- Reduces overfitting by discouragng subset of weights dominating

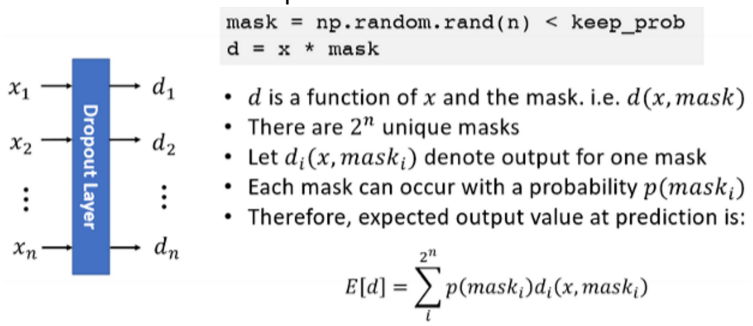
- **Regularizing Bias Params**
  - doesn't have big impact since orders of magnitude more in weight params

Dropout

- on each param update iteration, **randomly** remove some hidden units from network
- training bunch of smaller simpler models and ensembling them together
- each model overfits in different ways, averages out
- don't put too much weight
- force each unit to learn to work well with rand subset of input units



- **At prediction time**
  - nondeterministic predictions!



- where to put dropout layers?
  - FCL since prone to overfitting compared to conv layers
  - conv layer less prone to overfitting since eatch swatch is a separate piece of training data

DropConnect

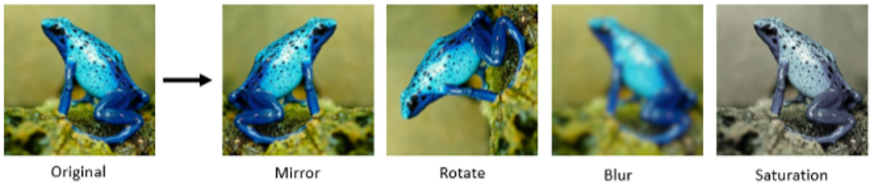
- Similar to dropout except 0 out random weights (connections) at training instead of nodes

BatchNorm Regularization Effect

- mean and variance on minibatch is an approx to actual mean and variance
- introduces randomness
- unintended regularization effect

Data Augmentation

- generating new training data grom existing training data



- take different crops to generate new images
- **Types of regularization to use**
  - L2 most common
  - Dropout for large FCL
  - don't rely on BatchNorm for regularization
  - Data Augmentation for imaged data

Hyperparameter Tuning

- Architecture
  - # layers
  - # units/filters per layer
- Optimization
  - learning rate
  - weight init
  - otpimizer hyperparameters (momentum beta)
  - regularization techniques
- for some hyperparams, want to search over log scale, eg. learning rate range 0.0001 to 1
- coarse to fine
  - search in initial range of hyperparam values
  - find values that minimize cost
- **Training Advice**
  - start by using small subset of training set, get model to 100% accuracy (get training accuracy high first)
    - turn off regularization
    - flush out bugs in optimization flow
  - using full training set, find learning rate that shows good decrease in cost (close the gap and improve validation accuracy)
    - turn on regularization
    - see effect of learning rate in small number of training iters
  - do the hyperparam search as shown before
  - look at failing cases and cost curves (visualize)

Looking at Cost Curves

***Need to see lectures for curves graphs

Transfer Learning

- take model trained for a task, repurpose it for second similar task
- keep some of the learnings (param values)
- used for img data and text/speech
- train new data set, but only let new output layer's params be updated
  - early layers of CNNs learn vocab of visual constructs
  - don't need to relearn these
- can let last couple conv layers also be retrained
- when to use
  - both tasks have same inputs
  - significantly less training data available for new task

    - low level features are similar for tasks
- Pros
    - leverage previous training efforts, don’t start from scratch
    - lets yous tart with good param values
    - don’t need to relearn common low level featuers
    - helps pre train model if not much data