ANLY 590 - Optimization, Regularization, Representational Capacity

Joshuah Touyz

August 30, 2018

Table of Contents

1 Optimization methods

- So far we've only used gradient descent
- In practice there are different ways to implement gradient descent
- Alternate forms of gradient descent exist
- Each modifies the learning parameter in such a way:
  - That information about previous moves are used in updating the learning rate
  - Additional parameters are included
- Examples of methods include:
  - Nesterov Momentum (physics inspired)
  - ADAM
  - Adagrad
  - Adadelta
  - RMSprop
- In what next follows we consider:
  - Loss functions
  - Challenges with Fitting Networks
  - Batch methods
  - Connection with Taylor Expansion
  - Different types of optimizers

1.1 Deeper dive into loss functions and parameter optimization

- In machine learning we are interested in optimizing some performance measure P defined with respect to the test set that is frequently intractable
  - Rather than optimize P we optimize the cost function $J(\theta)$

  - Pure optimization however has a goal of optimizing $J(\theta)$
- Typically the cost function is written as

$$J(\theta) = E_{(x,y)\sim \hat{p}_{data}}[L(f(x;\theta),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)})$$

- Where:
  - $E\_\{(x,y) \sim \hat{p}\_\{data\}\}$ is the expectation with respect to the empirical distribution (training set)
  - $L(\cdot,\cdot)$ is the loss function (think of this as a distance metric)
- Note:
  - Ideally we want to minimize with respect to the true data generating mechanism p_{data} however it is frequently unknown and so $\hat{p}_{data}$ is used a proxy
    * A situation in which p_{data} is known is in simpler situations like flipping a coin
  - When minimizing with respect to p_{data}, $J^*(\theta)E_{(x,y)d\tilde{ata}}[L(f(x;\theta),y)]$ is known as the **Risk**
    * Minimizing with respect to $J(\theta)E_{(x,y)\sim \hat{p}_{data}}[L(f(x;\theta),y)]$ is therefore similarly known as **Empirical Risk Minimization**

1.1.1 Challenges with Network Fitting

- Below we list a couple of challenges with Network optimization
  - Traditional methods may not work as well i.e. L-BFGS
  - Models with high capacity can memorize the training set
    * Using regularization forces the network to learn
  - Second order (i.e. matrix) optimization methods suffer from:
    * Poor condition numbers (matrix instability with inverses due to over/underflow)
    * Computationally expensive
  - NNs are **non-convex functions**
    * This means it is possible to have local minima (it's almost guaranteed to have a large number of them) and saddle points
    * Saddle points pose a particular problem for first-order optimization since only gradient information is available
    * Models are not **identifiable**
      · This means parameter can be permuted within the network and we'll still achievethe same local minima
      · Put another way there is no one set of parameters for a given local minima
      · Several reasons exist for parameter non-identifiability , two commonly cited are **weight space symmetry** and **constant scaling** (in ReLU networks)
      · _Weight space symmetry__ means for m layers with n units each, there are $n!^m$ ways of arranging hidden units
      · **Constant Scaling** in a ReLU network means that we can scale incoming weights by some α provided we scale outgoing weights by $1/\alpha$

        * Local minima are problematic if they have a high cost in comparison to global minimum
    - Efficiently distributing and parallelizing optimization is challenging
        * The methods need to be shown to work in parallel
        * Frequently empirical

- Optimization remains an open area of deep learning research
- Early challenges with neural networks suffered both from hardware limitations and optimization limitations
- Two major changes in the last 10 years included:
    1. Parallel distributed computing on commodity GPUs (more on this in Lecture 4)
    2. Using alternate activation functions (i.e. ReLU vs sigmoid) which increased network stability and speed at which networks could be trained (see HW1)
- In practice most local minima have a low enough cost function value that it isn't necessary to find the true global minimum but one that is "good enough"

1.1.2 Surrogate Loss functions

- **Surrogate loss functions** are used as approximates to our true loss functions because:
    - They have efficiency advantages
    - Their derivative does not contain useful information (0-1 loss in classification)
        * An example of a surrogate loss function in classification is cross-entropy rather than accuracy
    - Their transforms are usually bijective mappings over the space

1.1.3 Batch, Minibatch and Stochastic Gradient Descent

High-level Description

- When the amount of data points is large we won't be able to fit it in memory and train our models
    - There are also stability considerations for why we would want to use batch gradient descent
- Instead of using pure gradient descent we use batched forms
- That is we train our models on small sets of the data

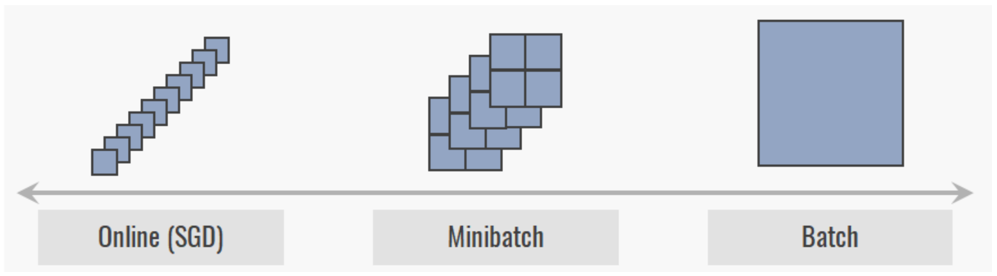
Stochastic vs Minibatch Gradient Descent

- Optimization algorithms that use the entire data set are called **batch** methods and are deterministic (batch gradient descent uses the entire training set)

1. Stochastic gradient descent:
    - Optimization algorithms that use a single example to update parameter values are called **stochastic** (sometimes known as *online* learning because data is being streamed)
    - Implementation:

    - Randomly samples one element at time
    - Runs through backwards/forwards propagation
    - It then updates the cost function

2. Mini-batch gradient descent:
    - Most parameter learning algorithms fall somewhere in between batch and stochastic and are frequently called **minibatch** methods (although it is now common to refer to them as *stochastic* which is a misnomer)
    - Implementation:
        - Samples several points at a time and then
        - Applies forwards/backwards prop to update parameters and cost function
    - Commonly called batch gradient descent
    - Synonymous with SGD, although incorrect (people refer to mini-batch as SGD)



1.1.4 General Formula for Gradient descent

- Let B the batch size
- Then the general formula for the weight update step in mini-batch SGD is given by:

$$\theta_{t+1} = \theta_t - \epsilon(t)\frac{1}{B}\sum_{b=0}^{B-1}\frac{\partial L(\theta, \mathbf{m}_b)}{\partial \theta}$$

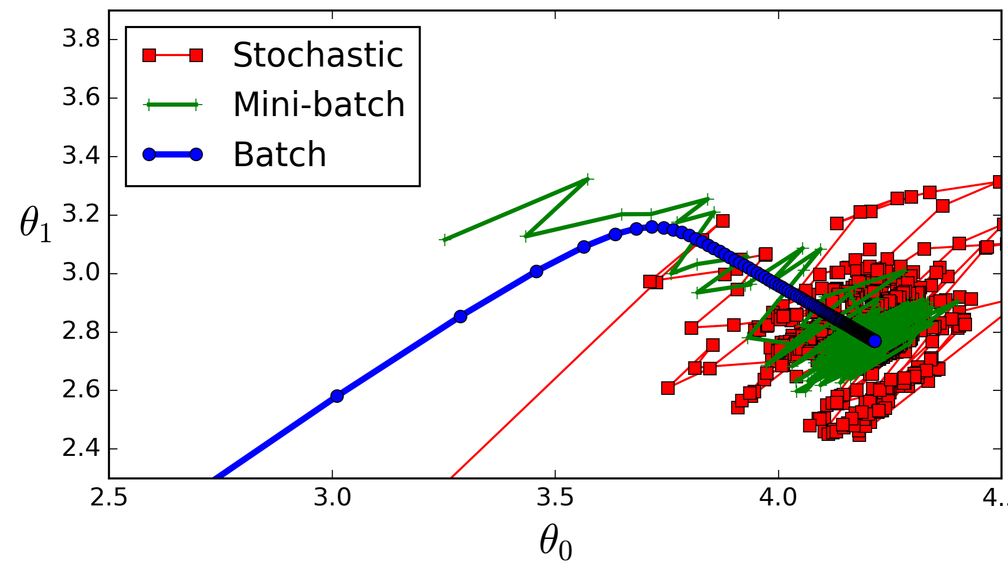
- where:
    - Batch gradient descent, $B = |x|$
    - Online stochastic gradient descent: $B = 1$
    - Mini-batch stochastic gradient descent: $B > 1$ but $B < |x|$
- **Note**:
    - When taking samples for minibatch they are done *without replacement*
    - Once all samples have been iterated through it is called an *epoch*
    - Batches are averaged after each epoch to yield an updated parameter
    - Note that when using the entire data set, the loss function is no longer a random variable and is not a stochastic approximation
    - For a greater exposition see the stats.stackexchange link

Considerations with minibatch:

1. Large batches are more accurate however performance return is is less than linear

- i.e. bigger batches don't mean you get a linear increase in accuracy

2. Multicore architectures are underutilized with small batches
3. When processing batches in parallel - memory scales with batches (this is usually a hardware limitation - we'll discuss this more in the tool kit section)
4. Some hardware is specialized to training (GPUs) and works better with batch sizes
    - Typical batch sizes follow powers of 2
    - 32-256 are common with 16 sometimes being used for large models
5. Small batches cab offer a regularizing effect (perhaps due to the noise added during the learning process)
6. Generalization error is minimized with a small batch size (i.e. 1) however should be done with a small learning rate to maintain stability (total runtime will however be much longer)



Newton's method, Hessians and their connection to Gradient Descent

- A parallel interpretation of gradient descent can be cast in terms of Newton's method
- Below we present the intuition
- Newton-Raphson is a common iterative optimization method
- Loosely for some function $f_T(x)$ it

    1. Takes the 2nd order Taylor expansion around some x:

$$f_T(x) = f_T(x_n + \Delta x) \approx f(x_n) + f'(x_n)\Delta x + \frac{1}{2}f''(x_n)\Delta x^2$$

    2. Takes it's derivative with respect to Δx such that $x_n + \Delta x$ is a stationary point (derivatice of $f(x) = 0$:

$$\frac{df(x)}{d\Delta x} = \frac{d}{d\Delta x}\left(f(x_n) + f'(x_n)\Delta x + \frac{1}{2}f''(x_n)\Delta x^2\right) = f'(x_n) + f''(x_n)\Delta x$$

3. Set the equation to 0 and find the Δx such that $x_n + \Delta x :\Rightarrow f(x_n + \Delta x) = 0$:

$$f'(x_n) + f''(x_n)\Delta x = 0 \Leftrightarrow \Delta x = -\frac{f'(x_n)}{f''(x_n)}$$

4. And then solves for recursive solution by subbing in the value Δx

$$x_{n+1} = x_n - \underbrace{\frac{f'(x_n)}{f''(x_n)}}_{\Delta x}$$

- It can be shown that the convergence rate for Newton-Raphson is quadratic
- Similarly for the multidimensional case where $\mathbf{x} \in \mathbb{R}^M$ we have:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [H(f(\mathbf{x}_n))]^{-1}\nabla f(\mathbf{x}_n)$$

  where $H = \frac{\partial^2 f(\mathbf{x}}{\partial x_i \partial x_j}\Big|_{i,j=1,\ldots M}$ is the Hessian
- Notice that if $f''(x_n)^{-1}$ and $[H(f(\mathbf{x}_n)]^{-1}$ are replaced by some constant η then

$$x_{n+1} = x_n - \eta f'(x_n)\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla f(\mathbf{x}_n)$$

  and NR is similar to gradient descent
- Similarly if $\eta(x_n)$ is a function of previous values it may be used to approximate the Hessian (we'll see this later with adaptive learning rates)
- Perhaps the central question is "if second order methods gradient methods provide greater accuracy why are they not used?"
  - They can be used for small data sets
  - Second order methods face issues with non-convex functions at saddle points
  - As the amount of data and parameters increases so does the computational burden
  - For a k parameter network matrix storage is k^2 and it's inversion at each iteration is $O(k^3)$
  - There are ways to side step inverse matrix computation such as conjugate gradients or low-rank matrix approximations such as L-BFGS (Limited memory Broyden-Fletcher-Goldfarb-Shannon algorithm)

1.1.5 Momentum optimization methods

- Having covered how to sample batches let's now consider alternate optimization methods
- Here we draw inspiration from the movement of a particle
- It accumulates an exponentially decaying moving average of past gradients to move in their direction
- The method of momentum is designed to accelerate learning, particularly:
  - When the surface has high curvature
  - Small but consistent gradients or
  - Noisy gradients
- **Vanilla Momentum**:

  - We introduce a momentum variable $\mathbf{v}$ (direction and speed a particle movies through parameter space)
  - Update algorithm:

$$\begin{aligned}\mathbf{v}_{t+1} &\leftarrow \alpha\mathbf{v}_t - \epsilon\nabla_\theta\left(\frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})\right)\\ \theta_{t+1} &\leftarrow \theta_t + \mathbf{v}_{t+1}\end{aligned}$$

- **Nesterov Momentum**:
  - Sutskever et al. (2013) introduced Nserterov Momentum that adds a correction factor to vanilla momentum
  - When applied to *convex batch gradient descent* the rate of convergence of the excess error decreases from $O(1/k)$ to $O(1/k^2)$ after k steps

$$\begin{aligned}\mathbf{v}_{t+1} &\leftarrow \alpha\mathbf{v}_t - \epsilon\nabla_{\text{`}}\left(\frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\theta+\alpha\mathbf{v}_t),\mathbf{y}^{(i)})\right)\\ \theta_{t+1} &\leftarrow \theta_t + \mathbf{v}_{t+1}\end{aligned}$$

1.1.6 Adaptive Learning Rates

- Learning rates are of central importance as they have significant impact on model performance
- Given their sensitivity (or insensitivity) across the parameter surfaces have an adaptive rate that decreases or increases based on curvature
- Several recent methods are listed below:
- In each we include a $0 < \delta << 1$ for numerical stability

1. **AdaGrad**(Duchi et al 2011):
   - Individually adapts learning rates of all model parameters
   - Accumulates the gradient over time
   - Scale the current gradient by accumulated gradients

$$\begin{aligned}\mathbf{r}_0 &\leftarrow 0\\ \mathbf{g}_t &\leftarrow \nabla_{\text{`}}\left(\frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})\right)\\ \mathbf{r}_{t+1} &\leftarrow \mathbf{r}_t + \mathbf{g}_t \odot \mathbf{g}_t\\ \Delta\theta &\leftarrow -\frac{\epsilon}{\delta+\sqrt{r}} \odot \mathbf{g}_t\\ \theta &\leftarrow \theta + \Delta\theta\end{aligned}$$

2. **RMSProp**(Hinton 2012):
   - Modifies ADAGrad to perform better in nonconvex settings
   - Gradient accumulate as an exponentially moving average
   - Discards older history from the extreme past to converge more rapidly than ADAGrad
   - Show to be an effective and practical algorithm
   - Introduces new parameter ρ that controls the length scale of the moving average

- A variant on RMSProp exists with Nesterov momentum adds a momentums factor to θ
- One of the go-tos for training deep networks

$$
\begin{aligned}
\mathbf{r}_0 &\leftarrow 0 \\
\mathbf{g}_t &\leftarrow \nabla_{`}\left(\frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})\right) \\
\mathbf{r}_{t+1} &\leftarrow \rho\mathbf{r}_t + (1-\rho)\mathbf{g}_t \odot \mathbf{g}_t \\
\Delta\theta &\leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot \mathbf{g}_t \\
\theta &\leftarrow \theta + \Delta\theta
\end{aligned}
$$

3. **Adam** (Kingma and Ba, 2014):
   - Derives from "adaptive moments"
   - Variant of RMSProp and momentum
   - A form that adapts learning rates based on 1st and 2nd moment adjusted with momentum
   - Has two scaling factors ρ_1, ρ_2 with default rate between 0.9 and 0.999, respectively

$$
\begin{aligned}
k &\leftarrow 0; \mathbf{s}_0 \leftarrow 0; \mathbf{r}_0 \leftarrow 0 \\
\mathbf{g}_t &\leftarrow \nabla_{`}\left(\frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})\right) \\
\mathbf{s}_{t+1} &\leftarrow \rho_1\mathbf{s}_t + (1-\rho_1)\mathbf{g}_t \quad ; \quad \mathbf{r}_{t+1} \leftarrow \rho_2\mathbf{r}_t + (1-\rho_2)\mathbf{g}_t \odot \mathbf{g}_t \\
k &\leftarrow k+1 \quad \hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^k} \quad \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^k} \\
\Delta\theta &\leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot \mathbf{g}_t \\
\theta &\leftarrow \theta + \Delta\theta
\end{aligned}
$$

1.1.7 Which optimization algorithm to select?

- There is no broad concensus
- Schaul et al 2014 study suggested adaptive methods (RMSProp and AdaDelta) are robust however do not conclude one method is consistently better than others
- The most popular methods which yield reasonable results are:
  1. SGD
  2. SGD with momentum
  3. RMSProp
  4. RMSPropr with Momentum
  5. AdaDelta and
  6. Adam

2 Regularization

- When fitting networks the goal is to optimize a loss function that allows generalization
  - We want our methods to best optimize on the training set without loss of generalization

  - Regularization is among the central methods by which we can enforce "learning"

- Here we will introduce several regularization methods
- Regularizations prevents models from overfitting
- Recall in logistic regression that by adding a penalty parameter to our loss function parameters were bounded or downweighted
- Here discuss several additional approaches used with neural networks
  - Our focus will be on the most common set of methods
  - We'll also touch on some less common methods
- In all cases regularization acts to restrict the hypothesis space
- It controls the generalization of the model by trading some bias for large decreases in variability
- To quote 5.2.2. of Goodfellow: **"Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error."**

2.1 Methods

- We'll discuss the following methods and their implementation:
  - Parameter Norm Penalties - L1/L2 regularization
  - Stochastic dropout
  - Noise injection
  - Early stopping
  - Data augmentation
- Other methods we mention here but will not dive into include:
  - Ensembling (TBD with advanced topics)
  - Explicit optimization using Karush-Kuhn-Tucker conditions
  - Multi-task learning
  - Sparse representations

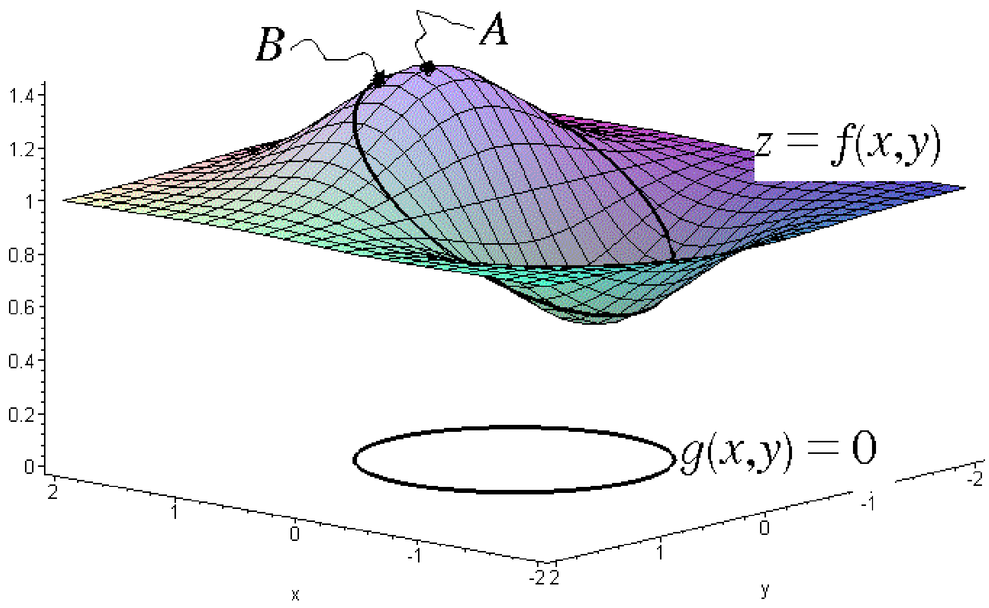
2.2 Parameter Norm Penalties

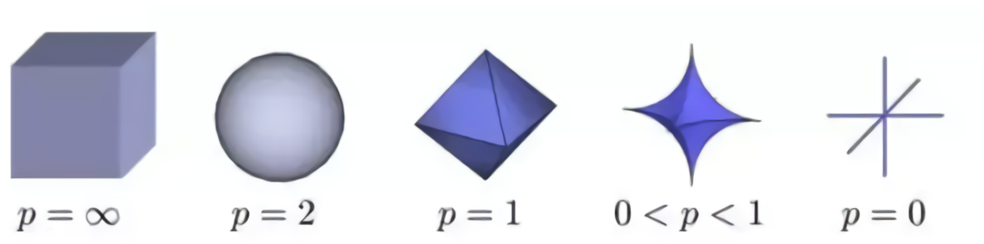
- The two most common operation penalties are L1/L2 penalties
- L1/L2 penalties are effectively LaGrange multipliers
  - They are a form of constrained optimization
- Below we present three different views of L1/L2 penalties to drive the intuition
  - 3d View of different regularizers, these are the surfaces constraints
  - Below are 3 different views for how L1/L2 penalties work for minimization
- Norm penalties are added in the cost function:

$$\tilde{J}(\theta;\mathbf{X},\mathbf{y}) = J(\theta;\mathbf{X},\mathbf{y}) + \lambda||\theta||_p$$

where $||\theta||_p$ is the p-norm and λ is the penalizing constant and gradient

$$\nabla_\theta\tilde{J}(\theta;\mathbf{X},\mathbf{y}) = \nabla_\theta J(\theta;\mathbf{X},\mathbf{y}) + \nabla_\theta\lambda||\theta||_p$$





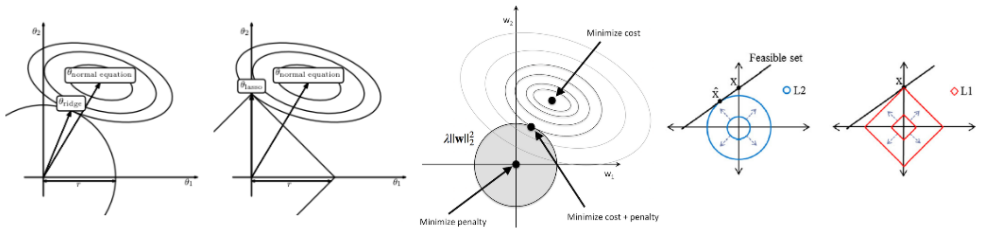
- For L_1 cost function is:
$$\tilde{J}(\mathbf{w};\mathbf{X},\mathbf{y}) = J(\mathbf{w};\mathbf{X},\mathbf{y}) + \lambda||\mathbf{w}||_1$$
- For L_2 the
$$\tilde{J}(\mathbf{w};\mathbf{X},\mathbf{y}) = J(\mathbf{w};\mathbf{X},\mathbf{y}) + \lambda\frac{1}{2}||\mathbf{w}||_2^2$$
- When applying regularizers the L_1 and L_2 gradients are updated as in backprop algorithm as follows:

$$L_1 : \mathbf{w} \leftarrow \mathbf{w} - \epsilon(\nabla_{\mathbf{w}} J(\mathbf{w};\mathbf{X},\mathbf{y} + \lambda \text{sign}(\mathbf{w}))$$
$$L_2 : \mathbf{w} \leftarrow \mathbf{w} - \epsilon(\nabla_{\mathbf{w}} J(\mathbf{w};\mathbf{X},\mathbf{y} + \lambda \mathbf{w})$$

- Some additional theoretical notes:
  - Interpreted in the Bayesian context L_2 and L_1 regularization are equivalent to Gaussian and isotropic Laplace priors placed over the network
  - L_2 norms can be interpreted as rescaled weights along of the eigenvectors of the cost function

2.2.1 Dataset Augmentation

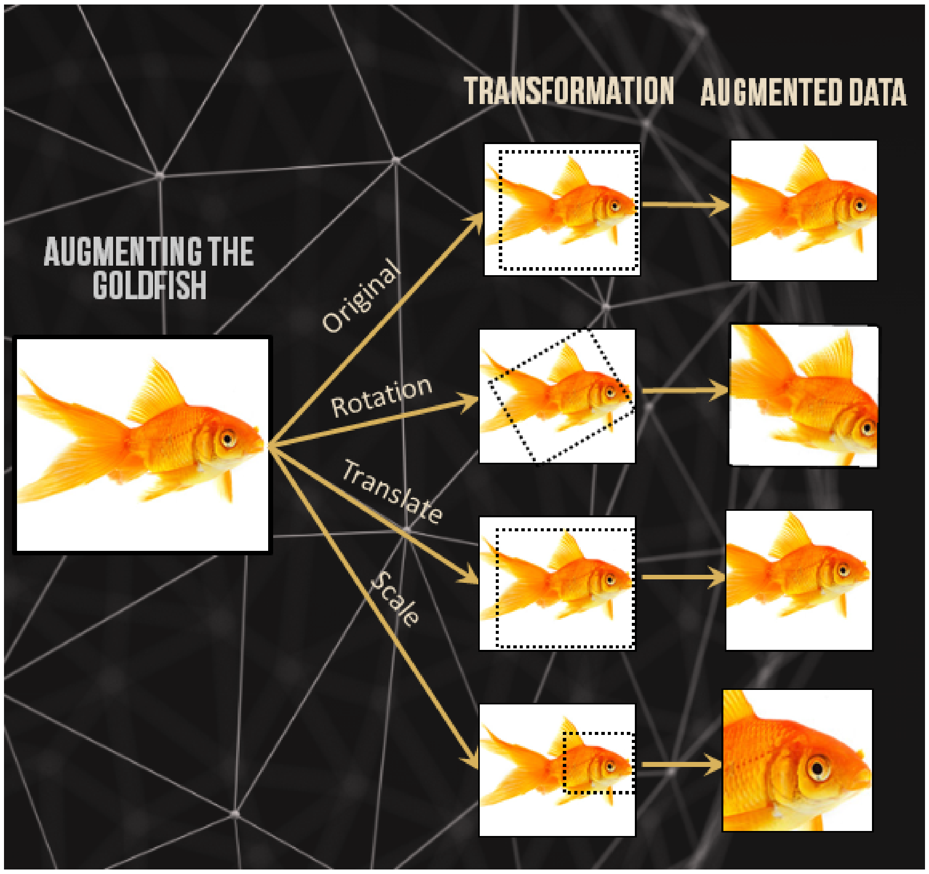
- Dataset augmentation applies a set of transforms to data in the feature space



- These hand-designed perturbations don't change the content of the features however can dramatically reduce the generalization error
- It's frequently used with image recognition where common transforms include:
  - Rotation
  - Decrease size
  - Scaling
  - Shearing
  - Translation
  - Occlusion
  - Mirroring
  - Lighting shifts
- By increasing the number of "exemplars", a model will learn their feature invariance at different resolutions, positions, lighting conditions and symmetries
- A word of caution however is that certain transformations may result in flipping classes such as 6 to a 9, these should be avoided since they add noise
- Noise injection may also be viewed as a form of dataset augmentation
  - Poole et al 2014 showed noise acts a form of regularization
  - We'll come back to this when discussing autoencoders

2.3 Noise robustness

- Noise can be added at the:
  1. Input
  2. Hidden layers
  3. Output
- Input layer noise:
  - When noise is added at the input it is similar to data augmentation
  - In some cases in may also be interpretted as a penalty norm (Bishop 15a,b)
- Hidden layer noise:
  - Adding noise to the weights



    - Is primarily used in recurrent neural networks (Jim et al. 1996; Graves, 2011)
- May be interpreted as a:
    - A stochastic implementation of Bayesian inference over the weights
        - In Bayesian approaches model weights are uncertain
        - Injecting noise is a way to stochastically emulate the uncertainty over these weights
    - Form of regularization that encourages model stability
    - Example:
        - Consider the simplified regression setting where J is the loss function

$$J = E[(\hat{Y}(X) - T)^2] = E[(\beta^T X - T)^2]$$

where X are the observed values, T is the target variable and β are the parameters to be estimated

        - Now inject some $\eta \sim MVN(0, \sigma I)$ to the weights β

$$\tilde{J} = E[((\beta + \eta)^T X - T)^2]$$

        - By expanding the terms the resultant equation is:

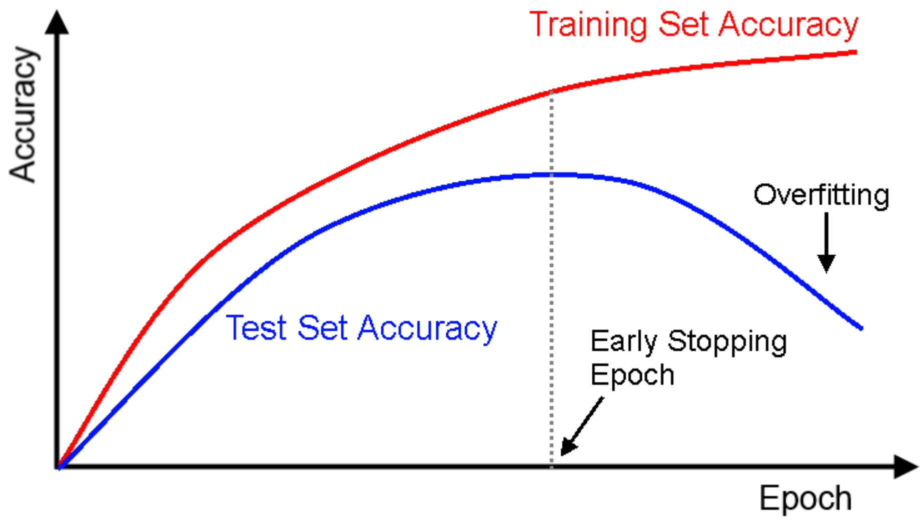
$$\begin{aligned}\tilde{J} =& E[((\beta+\eta)^T X - T)^2] = E[((\beta^T\beta + \eta^T\eta + 2\beta^T\eta)^T X)^2 - 2(\beta+\eta)^T XT + T^2] \\ =& E[(\beta^T X - T)^2] + \underbrace{2\eta^T E[(\beta - \beta^T\eta^{-1})^T XT]}_{\text{regularizer}}\end{aligned}$$

where the last part of the equation is a form of regularization
    - Intuitively when X is replaced by h the result can be ported to a deep network
  - We'll discuss regularization at hidden layers more when discussing drop-out

- Output perturbation
  - Some data sets have errors in their output values
  - Consider how difficult it may be to differentiate between a 1 and 7 in MNIST
  - Using **label smoothing** refularizes model based on a softmax with k output values by replacing the hard classification targets 0 and 1 with $\epsilon/(k-1)$ and ϵ

2.4 Early stopping

- One of the most effective and simplest forms of regularized
- Look for U shape in training set
- Bishop 1995 argued that has the effect of restricting the procedure to a small volume in the parameter space (in the neighbourhood of the original parameter space)
- Early stopping is similar to controlling the weight decay and is equivalent under certain condition to L_2 regularization

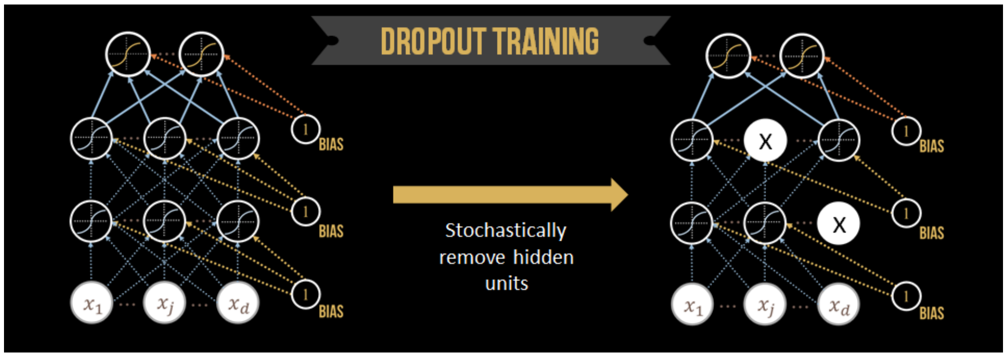


2.5 Stochastic dropout

- Here we "stochastically" set hidden units to 0 during forward and backward passes
  - We do this by applying a binary mask
  - For hidden layer $h^{(k)}$ we apply the binary mask $\mathbf{m}^{(k)}$

$$\tilde{h} = h^{(k)} \odot \mathbf{m}^{(k)}$$

- This has the effect of forcing other units to learn patterns rather than memorizing the data
  - Hidden units cannot co-adapt to other units
  - Hidden units need to learn features



3 Importance of feature engineering

- Deep networks allow us to explore complex hypothesis spaces
- However well designed features have additional benefits including:

  1. Decreased computational resources
  2. Decrease model complexity
  3. Less data is required for training

- A good example is predicting the time on a clock from an image
- In order of complexity of approach (most to least)

  - Approach 1: Scan clock and build model to process image
  - Approach 2: Get point coordinates for the hands
  - Approach 3: Get angle between large and small hands

- Accordingly a first approach is to understand and simplify the problem

  - It greatly decreases model complexity, increases generalizability and subsequently increases accuracy

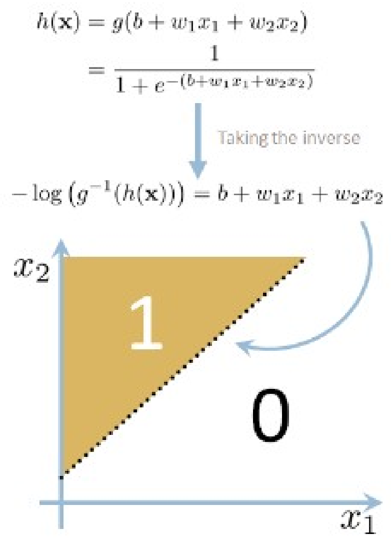
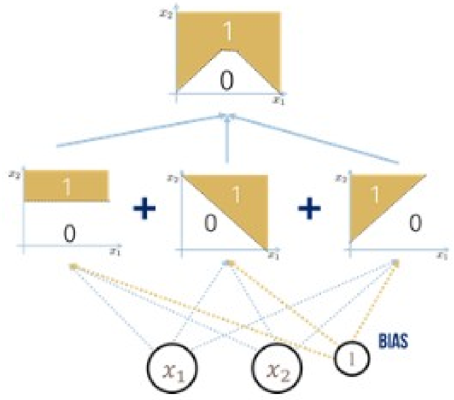
4 Representational Capacity

- The idea of **representational capacity** is the degree to which a neural network can represent a function $f(x)$
- Representational capacity is important in so far as it helps us determine the appropriate architecture for our neural networks
  - In particular the number of units within our hidden layers

    The number of layers

  - The connectivity between the layers

  - Pooling and unit types
- A well known theorem states that any 2-layer neural network with an infinite number of hidden units can approximate any function $f(\mathbf{x})$ with arbitrary precision.
  - Neural networks, even while shallow, are therefore known as **universal approximators**
- The utility of having a very large number of hidden units in a 2-layer network is questionable because

- The number of parameters increases with the domain of $f(\mathbf{x})$
  Looking at the decision boundary of logistic unit we see that we can compose more complicated boundaries:
  -
    
    $h(\mathbf{x}) = g(b + w_1x_1 + w_2x_2)$
$= \frac{1}{1+e^{-(b+w_1x_1+w_2x_2)}}$
Taking the inverse
$-\log(g^{-1}(h(\mathbf{x}))) = b + w_1x_1 + w_2x_2$
x_2
1
0
x_1
    

  - Approximating a continuous function with binary nodes requires an infinite number of hidden units
- A couple of notes
  - Capacity depends on the underlying transformation of the activations
  - Shallow networks fail to generalize and max out in terms of utility around 3 hidden layers (most practitioners don't go beyond this)
  - Deep networks that use alternate architectures are able to generalize
- Three questions arise:
  - How do we increase representational capacity with the addition of hidden layers over hidden units?
    * How much additional representation do we get by adding a hidden layer over a hidden unit?
  - How do we prevent over-fitting and 'learn' targets
  - At what point is there "sufficient" representation capacity that we can stop growing our network?