How Learning Differs from Pure Optimization
P is the unknown true performance measure, which we hope to improve by optimizing J
In pure optimization, optimizing J is a goal in and of itself
The cost function can be written as an average over the training set such as
$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim\hat{p}_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y),$ where L is the per-example loss function, f(x;0) is the predicted output, p-hat$_{\text{data}}$ is the empirical distribution, and *y* is the true output
This is the objective function with the training set (although the data generating distribution is preferable): $J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim p_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y).$ p_{data} is the true distribution here

The quantity above is known as the risk, with the expectation taken over the true underlying distribution p_{data}
When we do not know p_{data}(x, y), we have a machine learning problem
Minimize the expected loss on the training set, which consists of replacing the true distribution with the empirical distribution defined the training set; we minimize the empirical risk

Empirical risk minimization may cause the model to overfit and memorize the training data
Many useful loss functions such as the 0-1 loss have no useful derivatives (0 or undefined)
This means we rarely use empirical risk minimization; we use an even more different qty

Minimizing expected 0-1 loss - number of inaccurate examples - is intractable (exponential in input dimension), even for a linear classifier, so we use a surrogate loss function
Negative log likelihood of the correct class is used as a surrogate for the 0-1 loss, allowing the model to estimate the conditional probability of the classes given the input
If the model does that well, pick the classes which yield the least expected classification error
Test set 0-1 loss decreases even after training set 0-1 loss has reached 0, since when the expected 0-1 loss is 0, the classifier can be made more robust by pushing classes away, obtaining a more reliable classifier which extracts more info

Training algorithms usually don't halt at a local minimum; they minimize a surrogate loss, but halt when a convergence criterion based on early stopping is satisfied
Early stopping is based on the true underlying loss function, such as 0-1 on the validation set, and causes the algorithm to stop whenever overfitting occurs

- This causes training to stop while there are still large derivatives

Max likelihood estimation, viewed in log space, decompose into a sum over each example

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\text{model}}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}).$$

Maximizing this sum maximizes the expectation over the empirical distribution defined by the training set

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}).$$

The most commonly used property of the objective function is its gradient
$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},y\sim\hat{p}_{\text{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\text{model}}(\boldsymbol{x}, y; \boldsymbol{\theta})$. We estimate this by sampling a small subset of examples
The standard error of the mean estimated by n samples is (std)/sqrt(n), which shows very low returns when increasing the # of examples when it reaches a fairly high number
It is better to make small approximation errors when trying to make the model converge

Redundancy in the training set can be used to our advantage by not computing the same gradient multiple times for the same training example
Optimization algorithms using the entire training set are deterministic or batch gradient methods
Those which use only a single example are called stochastic or online methods
Online usually refers to receiving examples from a continuous stream rather than fixed data

Minibatch stochastic is in between and tends to be the best: memory scales with batch size, using right size batches saves computational resources, specific sizes of arrays (and therefore batch sizes such as 2, 4 … 2^n) work better when using GPUs, and small batches offer a regularizing effect by adding noise

Some algorithms are more sensitive to error sampling than others, as they compound errors
Methods computing updates based only on g can handle smaller batch sizes like 100
Second order methods which also use the hessian H and compute updates like $H^{-1}g$ require much larger batch sizes like 10,000 which minimize fluctuations in the estimate of $H^{-1}g$
If H is estimated perfectly but has a poor condition number, multiplication by H or its inverse amplifies pre-existing errors (such as estimation errors in g)
Even small changes in the estimate of g can cause large changes in $H^{-1}g$ even with a perfect H

Minibatches should be selected randomly - an unbiased estimate requires examples to be independent, and two subsequent gradient estimates should be independent
It is sometimes impractical to keep randomly sampling minibatches each time we want one
Randomly shuffling the order of the dataset once and storing it will impose a fixed set of minibatches of consecutive examples where each model reuses this ordering each epoch
This deviation from true random selection does not have a very negative impact in practice

Many optimization problems permit us to compute updates over different batches in parallel
Minibatch gradient descent follows the gradient of the true generalization error as long as no examples are repeated
After the data is shuffled, on the first pass, each minibatch is unbiased; on the second pass, the estimate becomes biased since it resamples values which have already been used rather than obtaining new fair samples from the data generating distribution

In the online learning case, examples of minibatches are drawn from a stream of data, and therefore every experience is a fair sample from the p_{data}

When x and y are discrete the generalization error can be written as follows:

$$J^*(\boldsymbol{\theta}) = \sum_{\boldsymbol{x}} \sum_{y} p_{\text{data}}(\boldsymbol{x}, y) L(f(\boldsymbol{x}; \boldsymbol{\theta}), y),$$ with gradient $$\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) = \sum_{\boldsymbol{x}} \sum_{y} p_{\text{data}}(\boldsymbol{x}, y) \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}; \boldsymbol{\theta}), y).$$

p_{data}(x, y) represents joint probability and acts as a weight for the loss
This holds for other functions L besides the likelihood
An unbiased estimator of the exact gradient of the generalization error can be obtained by sampling a minibatch of examples w/ corresponding targets, and computing the gradient of the loss w/ respect to parameters for that minibatch $$\hat{\boldsymbol{g}} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}).$$
Note that the unbiased estimator does not have a sense of weight and treats each example equally

Updating **ϴ** in the direction of g performs SGD on the generalization error
The additional epochs' bias is outweighed by decreasing the training set error such that the disparity between the training and test error caused by bias is less than the amount by which the generalization error of the test is decreased

Challenges in Neural Net Optimization
Neural nets have a nonconvex case
Ill-conditioning of the hessian H is a general problem, manifesting in nets by causing SGD to get stuck by causing even very small steps to increase the cost function
Second order Taylor expansion predicts gradient descent step of -epsilon*g adds
$$\frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g} - \epsilon \boldsymbol{g}^\top \boldsymbol{g}$$ to the cost (this gives us a good cost approximation)
Ill conditioning becomes a problem when the left term exceeds the right
Monitor the squared norm gradient g^Tg and g^THg to determine if ill conditioning is detrimental
g^THg can grow more than by an order of magnitude even though the gradient norm does not shrink significantly, causing the learning to become very slow since learning must be shrunk to compensate for strong curvature (as given by the equation)
Some techniques used to combat ill combating (such as Newton's method) are inapplicable to neural networks

In a convex function, any local minimum is guaranteed to be a global minimum
Neural nets have multiple local minima, but this is not necessarily a major problem
Neural nets have multiple equivalently parametrized latent variables due to model identifiability; a model is said to be identifiable if a large training set can rule out all but one set of params
Models w/ latent vars are not identifiable because equivalent models can be obtained by swapping - if we have m layers with n units each, there are $(n!)^m$ ways of arranging hidden units
This non identifiability is known as weight space symmetry

Furthermore, many kinds of neural nets have additional causes of non-identifiability - e.g. the incoming weights and biases can be scaled by a if the outgoing weights are scaled by 1/a

If the cost function does not depend on things like weight decay, every local minimum of a ReLU lies on an (m x n) dimensional hyperbola of equivalent solutions
All the local minima arising from non-identifiability are equivalent in the cost function, so they are not a problem
If local minima have high cost compared to the global minimum, this could be problematic
Many experts believe that it is not important to find a true global min, as long as the local minimum has low cost
If the norm of the gradient does not shrink to insignificant size, the problem is not local minima or any kind of critical point; in high dimensional spaces, however, many structures other than local minima have small gradients

High dimensional non-convex functions have a gradient of 0 at saddle points
Some points around a saddle point have a lower cost while others have a higher cost
At a saddle pt, the Hessian has positive and negative eigenvalues; points lying along positive eigenvalues have greater cost than the saddle point

The expected ratio of # of saddle points to local minima grows exponentially w/ n, the input dim.
It is exponentially unlikely all n inputs will be the same way, whereas 1 input is guaranteed to be
Eigenvalues of Hessian are more likely to become positive for regions of low cost, which means local minima are much more likely to have low cost than high cost
Critical points w/ low cost tend to be local minima; high cost tend to be saddle points

Shallow autoencoders w/o nonlinearities have global minima and saddle points, but no local minima with higher cost than the global minima
This extends to deeper networks without nonlinearities, which are still non-convex functions of their parameters; research has shown they share many properties w/ nonlinear neural nets

For first order optimization algorithms, the gradient can become very small near a saddle
However, gradient descent empirically seems to be able to escape saddle points
Saddle points are a problem for Newton's method

Newton's method is designed specifically to solve for when the gradient is 0, not minima
The proliferation of saddle points in high dim spaces explain why second order methods have not replaced gradient descent for training
Saddle-free Newton Method for second order optimization improves significantly over the traditional version, but it is hard to scale to large neural nets
A general optimization problem may contain a wide, flat region of constant value where both the Hessian and the gradient are 0, posing a major problem

Multiplication of several weights cause neural nets w/ many layers to have extremely steep regions resembling cliffs; gradient update can move the parameters very far in this case, moving them way off the cliff
The cliff can be overcome by using gradient clipping

The idea behind gradient clipping is the gradient only specifies an optimal direction within an infinitesimal region
Gradient clipping reduces the step size such that it is less likely to go outside the region where the gradient indicates the direction of steepest descent
Cliff structures are most typical in RNNs, due to temporal multiplication

Feedforward networks with many layers have deep computational graphs
Repeated application of the same params give rise to pronounced difficulties
Suppose a computational graph contains a path consisting of repeated multiplication by W; after t steps, this is equivalent to multiplying by W^T
If W has an eigendecomposition W = Vdiag(lambda)V^{-1} , $\boldsymbol{W}^t = \left(\boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}\right)^t = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}.$
Any eigenvalues that are not near an absolute value of 1 will explode if they are greater than 1 or vanish if they are less than 1; vanishing and exploding gradient is a result of the gradients through such a graph are scaled by $\text{diag}(\boldsymbol{\lambda})^t.$ since it is a constant used when calculating the gradient of W

The repeated multiplication by W at each time step is similar to the power method algorithm used to find the largest eigenvalue of a matrix W and its corresponding eigenvector
x^TW^t discards all components of x that are orthogonal to the principal eigenvector of W
Recurrent networks use the same matrix W at each time step, but feedforward nets do not, so deep feedforward networks can largely avoid vanishing and exploding gradients

In practice, we only have a noisy / biased estimate of the Hessian or exact gradient
In other cases, the objective function to minimize is intractable, causing the gradient to only be approximated
Neural net algorithms in practice are made to deal with these issues; choosing a surrogate loss function which is easier to approximate than the true loss is effective

Difficult to make a step at a point if J(**ϴ**) is poorly conditioned, **ϴ** lies on a cliff or saddle point
If these problems are overcome, the model may not perform well if the local direction of minima has high cost
Much of the runtime of training is due to the length of the trajectory needed to arrive at the sol'n

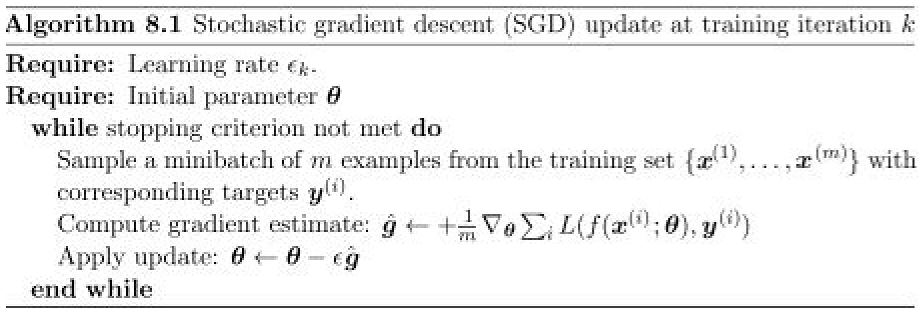
Most networks do not approach a critical point, or even a region of low gradient
For a classifier with a discrete y and p(y | x) provided by softmax, negative log likelihood can become very close to 0 if the model classifies everything correctly, but can never reach 0
A model of real values $p(y \mid \boldsymbol{x}) = \mathcal{N}(y; f(\boldsymbol{\theta}), \beta^{-1})$ - which works by having a mean of f(**ϴ**) and inverse variance - can have negative log likelihood that asymptotes to negative infinity; if f(**ϴ**) correctly predicts all training y, then B increases w/o bound since B^{-1} decreases due to an increasing variance check whether this is a result of softmax

We may be able to compute some properties of the objective function, such as its gradient only approximately w/ bias or variance in the correct direction's estimate
We are not able to follow local descent in this case

The objective function may have issues like poor conditioning or discontinuous gradients, causing the region where the gradient provides a good model of the objective function to be small

- Local descent w/ step sizes of epsilon may define a reasonably short path to the sol'n but we are only able to compute local descent dir. w/ steps of size d << epsilon
- Local descent may or may not define a path to the sol'n, but the path contains many steps, so following the path has a high computational cost

All these problems might be avoided if there exists a region of space connected reasonably directly to a solution by a path that local descent may follow (no issues in approximation, etc.)



Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

```
Require: Learning rate $\epsilon_k$.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while
```

In practice, it is necessary to change the learning rate over time, so we denote the learning rate at iteration k as ϵ_k.
This is because the SGD gradient estimator introduces noise, even when arriving at a minimum (whereas batch GD, which uses the entire dataset, reaches a gradient of 0 at the minimum)
Batch GD can use a fixed learning rate
A sufficient condition to guarantee convergence of SGD

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \quad \text{and} \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty.$$

It is common to decay the LR linearly until iteration T in practice $\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau$ with α = k/T

The learning rate is chosen by trial and error, best done by monitoring its learning curves which plot the objective function as a function of time
Parameters to choose when using the linear schedule are ϵ_0, ϵ_T, and T
ϵ_T should be set roughly to 1% of ϵ_0
If ϵ_0 is too high, then the learning curve will show violent oscillations; if it's low, then the algorithm will converge very slowly
Gentle oscillations are fine when working with stochastic cost functions such as those produced by dropout

Computation time per update for SGD / online learning does not grow w/ # of examples
For a large dataset, SGD may converge even before seeing the entire training set

It is common to measure the excess error $J(\Theta) - \min_{\theta} J(\Theta)$, the amount the current cost exceeds the minimum possible cost
When SGD is applied to a convex problem, the excess error is O(1/sqrt(k)) after k iterations, and is O(1/k) for the strongly convex case
Generalization error cannot decrease faster than O(1/k), so pursuing batch gradient descent which may converge faster than this presumably corresponds to overfitting
SGD's ability to make rapid initial progress while evaluating the gradient for only a few examples outweights its slow asymptotic convergence

Momentum is meant to accelerate learning in directions of high curvature, small and consistent gradients, or noisy gradients; it accumulates an exponentially decaying moving average of past gradients and continues to move in their direction
v plays the role of velocity, the direction and speed through which params move
Velocity is set to a negative exponentially decaying average of the gradient

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right),$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

The velocity v accumulates the gradient elements given by loss of examples
The larger α is relative to ϵ, the more previous gradients affect current direction
Momentum solves poor conditioning of the Hessian and variance in stochastic gradient

The step size now is largest when successive gradients point in the same direction; if the momentum algorithm always observes a gradient g, it accelerates in the direction of -g, reaching a terminal velocity where the size of each step is $\frac{\epsilon ||\boldsymbol{g}||}{1-\alpha}$. (formula of convergence of series)

The momentum hyperparamter can be thought of as $1/(1-\alpha)$
Common values of α is 0.5, 0.9, and 0.99; α is usually adapcted by increasing it over time

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```

The position of a particle at any point is given by Θ(t); it experiences a net force of f(t)

This force causes the particle to accelerate $\boldsymbol{f}(t) = \frac{\partial^2}{\partial t^2}\boldsymbol{\theta}(t).$ (note we deal with a unit mass)

$$\boldsymbol{v}(t) = \frac{\partial}{\partial t}\boldsymbol{\theta}(t),$$

$$\boldsymbol{f}(t) = \frac{\partial}{\partial t}\boldsymbol{v}(t)$$

Momentum algorithm consists of solving the differential equations through numerical simulation
Euler's method consists of taking small finite steps in the direction of the gradient

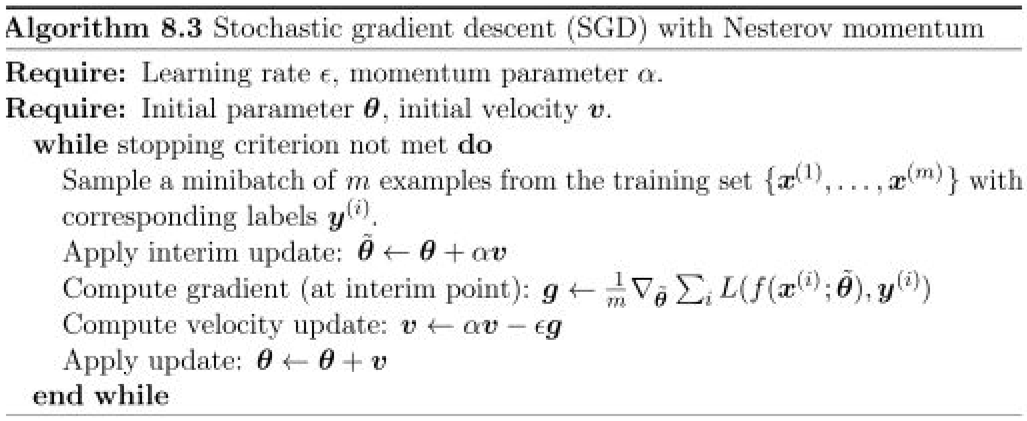
One force is proportional to the negative gradient of the cost function; the force pushes it downhill along the cost function surface, where as SGD would simply take a single step
Newtonian scenario used by momentum alters the velocity

To prevent the particle from sliding back and forth, we add another force, proportional to -v, corresponding to a viscous drag force, causing the particle to lose energy and converge
We must use a power of 1 for v, since turbulent drag (which squares v), becomes very weak when v is small, and when v is nonzero initial velocity, the distance from the start position grows with O(log t)
A power of zero causes the force to be too strong; a small nonzero constant force can cause the particle to come to rest before reaching a minima
Viscous drag lets the gradient continue motion until a minimum is reached, but is strong enough to prevent motion if the gradient does not justify movement

A variant, called nesterov momentum is shown below

$$\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\nabla_{\boldsymbol{\theta}}\left[\frac{1}{m}\sum_{i=1}^{m} L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}+\alpha\boldsymbol{v}), \boldsymbol{y}^{(i)}\right)\right]$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v},$$

The gradient is evaluated in a different location; with nesterov, the gradient is evaluated after the current velocity is applied, adding a correction factor
In convex BGD, Nesterov brings the rate of convergence error to an excess of O(1/k^2)
In SGD, Nesterov does not improve rate of convergence



Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with
    corresponding labels $\boldsymbol{y}^{(i)}$.
    Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```

Parameter Initalization Strategies
Initial point determines how quickly learning converges, and how much generalization error points of equal cost may have
There is not much concrete strategy for determining optimal modern initialization strategies

Initial parameters must break symmetry between different units; if two hidden units with the same activations are connected to the same inputs, then these inputs must have different initial parameters to prevent them from being updated in the same way
No input patterns are lost in the null space of forward prop or null space of backprop is each unit is initialized such that it computes a different function

We could explicitly search for a large basis of functions (e.g through Grahm Schmidt orthogonalization onto a weight matrix, such that all of them have different properties) which are mutually different from one another, but this has a high computational cost
Random initialization from a high entropy distribution over high dim. space is cheaper and unlikely to assign any units to compute the same functions as one another

Usually, only weights are initialized randomly; biases and parameters encoding the conditional variance of a prediction, are heuristically chosen constants
Weights are almost always initialized from a Gaussian or uniform distribution
Scale of the initial weight distribution seems to be more important

Larger initial weights yield a stronger symmetry breaking effect, and help avoid losing signal during forward or backprop through the linear component of each layer (since larger matrices yield larger matrix multiplication output and give a stronger gradient)
Initial weights that are too large may result in exploding values during forward or back prop
In recurrent networks, large weights can result in chaos - small perturbations of the input causing erratic behavior in GD
Exploding gradients may be solved to some extent by clipping
Large weights may result in extreme values causing activations to saturate, causing complete loss of gradient through saturated units

Optimization initialization perspective suggests large weights, while regulariaztion suggests smaller ones
An optimization algorithm such as SGD which tends to halt near the initial parameters expresses a prior that the final params should be close to the initial
Initializing params Θ to Θ_0 is similar to imposing a Gaussian prior p(Θ) w/ mean Θ_0
It makes sense to choose Θ_0 to be near 0
This prior says it is more likely units do not interact
However, if Θ_0 is initialized to a large value, the prior specifies which units should interact and how they should interact

Initial scale of weights: m inputs and n outputs by sampling from U(-1/sqrt(m), 1,sqrt(m)), or using a normalized distribution:

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

The latter compromises between the goals of all layers to have same activation variance, and all layers to have same gradient variance
Formula derived using assumption that network consists of only linear matrix multiplication

Alternative is to initialize random orthogonal matrices, with a gain factor g accounting for nonlinearity applied at each layer; g varies based on activation function
This guarantees # of iterations to convergence is independent of depth

Increasing g => activations increase in norm during forward prop, gradients increase in norm during backprop
Setting gain factor correctly can help train networks w/ 1000 layers
Activations and gradients can grow or shrink on each step, following a random walk; if the random walk is tuned to preserve norms, feedforward nets can avoid vanishing/exploding grads

Different factors may cause optimal weight criteria to not be sufficient
- May not be beneficial to preserve a signal's norm through the network
- Properties may not persist after learning has begun
- Improvign speed of convergence may not improve generalization error

Scale of weights should be a hyperparameter near the theoretical predicted value

Set of all weights having same std, such as 1/sqrt(m), leads to very small weights when layers become large
Sparse initialization - each unit has k nonzero weights
Total amount of input to the unit independent of the # of inputs, without making the magnitude of the individual weight elements shrink with m
Imposes a strong prior on weights chosen to have large Gaussian vaule
"Incorrect" large values can take a long time to shrink, which can cause problems for things like maxout units which have filters which need to work together

Treat scale of weights per layer and whether to use dense / sparse init. as hyperparameter
Look at the range of activations -> if the weights are too small, range of activations shrinks as forward prop occurs
By finding the first activation layer with very small activations and increasing its weights, we can obtain a network with reasonable activations throughout
If learning is still too slow, look at range / std of gradients
Less costly than hyperparameter optimization based on validation data since it is based off feedback from the behavior of the model on a single batch of data

Setting biases to 0 is compatible with most weight initialization schemes
Initialize bias to obtain correct marginal statistics; assume weights are small enough such that output of the unit is determined only by the bias -> bias is inverse of the activation function applied to the marginal statistics of the training set output (e.g solve softmax(b) = c)

Sometimes, choose the bias to avoid saturation at initialization (like in a ReLU)
This approach is comptible w/ weight initialization schemes which expect bias input

Sometimes, a unit controls whether others are able to participate
Unit w/ output u and another h is a member of [0, 1] -> they are multiplied to produce an output uh, where h is a gate determining whether uh ~ u or uh ~ 0
Set bias for h | h ~ 1 most of time at initialization, otherwise u cannot learn

Variance or precision parameter; linreg can be performed w/ a conditional variance estimate using the model $p(y \mid \boldsymbol{x}) = \mathcal{N}(y \mid \boldsymbol{w}^T \boldsymbol{x} + b, 1/\beta)$ where B is a precision parameter
Initialize variance or precision to 1
Alternatively, set biases to output (by making wts negligible), and then set variance parameters to match marginal variance of output in training set
Unsupervised or supervised learning can be used to initialize parameters of a neural net

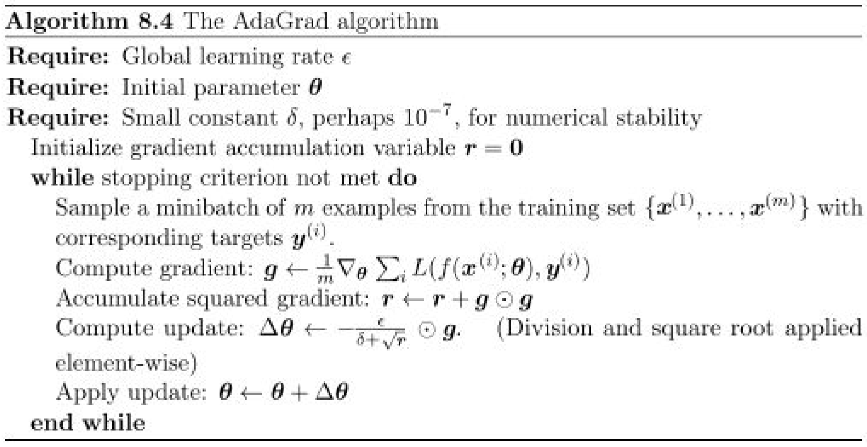
Algorithms with Adaptive Learning Rates
Separate learning rate for each of these parameters which are automatically adapted if directions of sensitivity are somewhat axis-aligned
Delta-bar-delta - if the partial derivative of the loss remains the same sign, LR should increase; if it changes sign, it should decrease <= requires full batch optimization

AdaGrad scales each parameter by a factor inversely proportional to the square root of the sum of all their historical squared values
Larger partial derivative of loss have a rapid decrease in learning rate
Leads to more progress in gently sloped directions of parameter space

Empirically, accumulation of squared gradients from the beginning of training can lead to a premature and excessive decrease in the learning rate

RMSProp modifies AdaGrad by changing the gradient accumulation into an exponentially weighted moving average
AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at a convex local min area

RMSProp discards very old iterations so it can converge rapidly after finding a convex region, as if it were an instance of AdaGrad initialized once it got into that bowl
Hyperparameter p controls length scale of the moving average
RMSProp is empirically effective

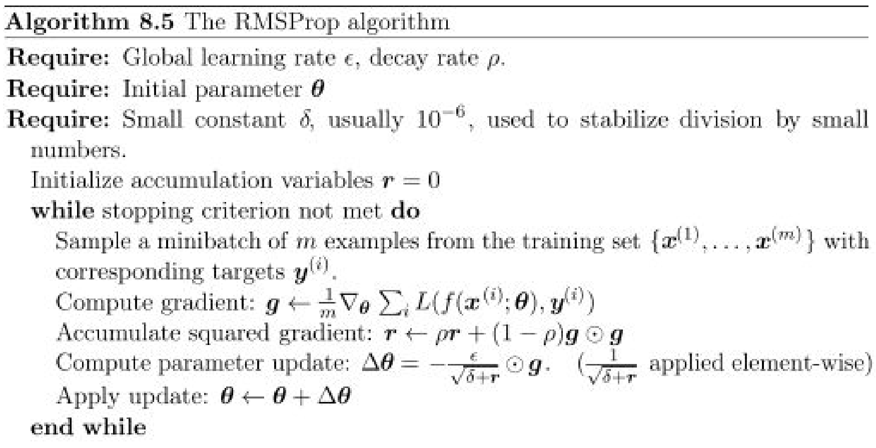


Algorithm 8.4 The AdaGrad algorithm

```
Require: Global learning rate $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, perhaps $10^{-7}$, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta+\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.   (Division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Note how denominator scales the learning rate

Adam (8.6) can be seen as a combination of RMSprop and momentum
Momentum is incorporated as an estimate of the first order moment - expectation - (w/ exponential weighting) of the gradient
Apply momentum to rescaled gradients in order add momentum to RMSprop
Adam includes bias corrections to first and second order moment estimates to account for initialization at the origin
RMSProp also incorporates second order moment, though it does not have bias correction



Algorithm 8.5 The RMSProp algorithm

```
Require: Global learning rate $\epsilon$, decay rate $\rho$.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, usually $10^{-6}$, used to stabilize division by small numbers.
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
    Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta+\boldsymbol{r}}} \odot \boldsymbol{g}$.   ($\frac{1}{\sqrt{\delta+\boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

```
Require: Global learning rate $\epsilon$, decay rate $\rho$, momentum coefficient $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
    Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.   ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```

Algorithm 8.7 The Adam algorithm

```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$
  (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization. (Suggested default
  $10^{-8}$)
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$   (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
  end while
```

Approximate Second Order Methods

Newton's method uses Taylor expansion to approximate J($\boldsymbol{\Theta}$) near $\boldsymbol{\Theta}_0$

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0),$$

H is the Hessian evaluated at $\boldsymbol{\Theta}_0$

If we solve for the critical point, we obtain the Newton parameter update rule

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

For a locally quadratic function, with a positive definite H, by rescaling the gradient by H^{-1}, Newton's method jumps directly to the minimum

For convex, non-quadratic functions, this method can be iterated, if the Hessian is positive definite

Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$.

```
Require: Initial parameter $\boldsymbol{\theta}_0$
Require: Training set of $m$ examples
  while stopping criterion not met do
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian inverse: $\boldsymbol{H}^{-1}$
    Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Update the inverse Hessian, and update the params according to the equation

If the eigenvalues of the Hessian are not all positive (e.g near saddle pt), the Hessian is not positive definite and Newton's method can cause movement in the wrong direction
This can be fixed by regularizing the Hessian
One way is to add a constant, **α** along the Hessian's diagonal

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \left[H\left(f(\boldsymbol{\theta}_0)\right) + \alpha \boldsymbol{I}\right]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0).$$

This makes Newton's method work well if the negative eigenvalues of the Hessian are close to 0
When there is a high amount of curvature, **α** has to be so high to offset negative eigenvalues; as **α** increases in size, the Hessian becomes dominated by **α**I and Newton's method converges to the standard gradient divided by **α**, which can become slower than SGD

Newton's method requires a lot of computational resources, since the Hessian has k^2 elements where k is the # of parameters
Newton's method requires the inversion of a k x k matrix, which is $O(k^3)$ complexity
The inverse Hessian has to be computed at every training iteration, so only networks with a small # of parameters can be practically trained with Newton's method

Conjugate gradients efficiently avoids the calculation of the inverse Hessian by iteratively descending conjugate directions
Steepest descent is shown to converge relatively poorly on a convex bowl since in each line search direction is guaranteed to be orthogonal to the previous line search direction

- Line search refers to methods such as Newton's method & SGD

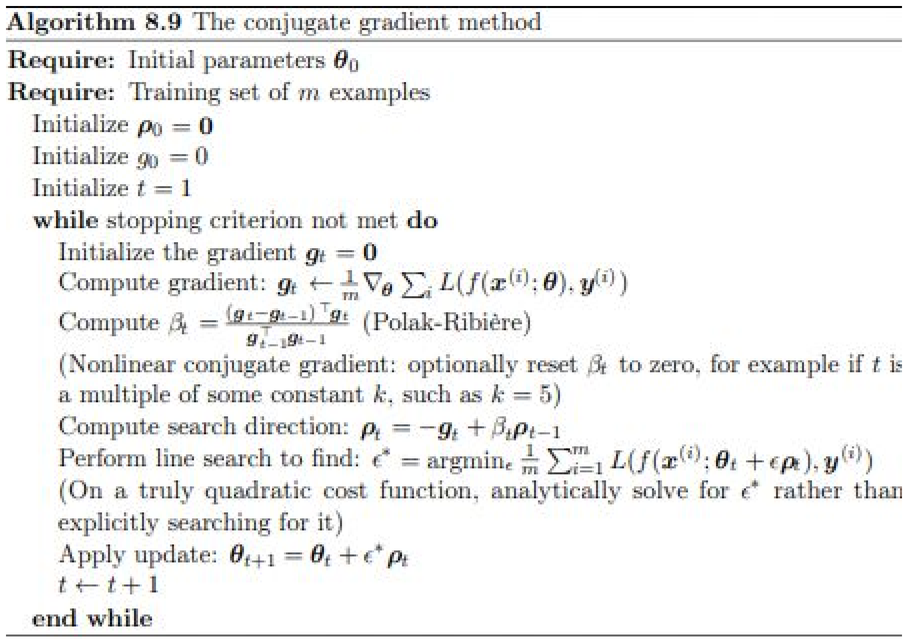
Let the previous search direction be d_{t-1}
When line search terminates at the min, the directional derivative is 0 in the direction d_{t-1}
The gradient's dot product with $d_{t-1} = 0$
Since d_t is the search direction defined by the gradient, d_t will have no contribution in the direction d_{t-1} and d_t is orthogonal to d_{t-1}

This causes a zig-zag pattern of progress, where by descending to the min in the current direction, we must re-minimize the objective in the previous gradient direction
Each time we follow the gradient at the end of line search, we are undoing progress we have already made in the direction of the previous line search

Find the directino which is conjugate to the previous line search direction (won't undo progress)
The next search direction d_t takes the form $\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1}$
B_t is a constant specifying how much of the direction of d_{t-1} should be added to the current search direction
They are defined as conjugate if $\boldsymbol{d}_t^\top \boldsymbol{H} \boldsymbol{d}_{t-1} = 0,$
Calculating the eigenvectors of H to choose B_t would not be computationally viable compared to Newton's method

One alternative is Fletcher-rieves: $\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$

The other is $\beta_t = \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}))^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})}$
For a quadratic surface, conjugate directions ensure that the gradient along the previous direction does not increase in magnitude, staying at the minimum along previous search directions
Conjugate gradient method requires at most k line searches in a k-dimensional space



```
Algorithm 8.9 The conjugate gradient method
Require: Initial parameters $\boldsymbol{\theta}_0$
Require: Training set of $m$ examples
  Initialize $\boldsymbol{\rho}_0 = \boldsymbol{0}$
  Initialize $g_0 = 0$
  Initialize $t = 1$
  while stopping criterion not met do
    Initialize the gradient $\boldsymbol{g}_t = \boldsymbol{0}$
    Compute gradient: $\boldsymbol{g}_t \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute $\beta_t = \frac{(\boldsymbol{g}_t - \boldsymbol{g}_{t-1})^\top \boldsymbol{g}_t}{\boldsymbol{g}_{t-1}^\top \boldsymbol{g}_{t-1}}$ (Polak-Ribière)
    (Nonlinear conjugate gradient: optionally reset $\beta_t$ to zero, for example if $t$ is
    a multiple of some constant $k$, such as $k = 5$)
    Compute search direction: $\boldsymbol{\rho}_t = -\boldsymbol{g}_t + \beta_t \boldsymbol{\rho}_{t-1}$
    Perform line search to find: $\epsilon^* = \text{argmin}_\epsilon \frac{1}{m} \sum_{i=1}^m L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_t + \epsilon \boldsymbol{\rho}_t), \boldsymbol{y}^{(i)})$
    (On a truly quadratic cost function, analytically solve for $\epsilon^*$ rather than
    explicitly searching for it)
    Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t$
    $t \leftarrow t + 1$
  end while
```

Nonlinear conjugate gradients - minimum along previous search direction is not guaranteed to be preserved; nonlinear conjugate gradients includes resets where method of conjugate gradients is restarted with the search along the unaltered gradient
Usually made better w/ a few steps of SGD before applying nonlinear conjugate gradients

BGFS takes a direct approach to the approximation of Newton's method

Recall Newton's update: $\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0),$

H is the Hessian of J w/ respect to $\boldsymbol{\Theta}$ evaluaated at $\boldsymbol{\Theta}_0$

The main issue is calculating H^{-1} - BFGS approximates H^{-1} w/ a matrix M_t iteratively refined by low rank updates

The direction of descent $p_t = M_t g_t$ <- line search applied to find the optimal step size $\boldsymbol{\epsilon}^*$

The update is given by $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t.$

BFGS iterates a series of line searches w/ the direction incoporating second order info

The success of the approach is not heavily dependent on line search finding a true minimum along the line

BFGS spends less time refining each search; BFGS must store the inverse Hessian matrix M, which requires $O(n^2)$ memory, becoming inpractical for many DL problems

L-BFGS avoids storing the complete inverse Hessian approximation M

M^{t-1} is assumed to be the identity matrix, rather than storing the approximation from one step to the next

If used with line searches, the directions are mutually conjugate, but it reamins well-behaved when the minimum of the line search is reached only approximately

Storing some of the vectors used to update M at each step costs only O(n) per step - but this accumulates over time - and can improve generalization

Optimization Strategies and Meta-Algorithms

Batch normalization is a method of adaptive reparametrization

When we make an update, we assume that the other functions are held constant; however, many layers are updated at once which may cause unpredicable behavior

Consider a simple NN: y-hat = $xw_1w_2...w_i$

y-hat is a linear function of x, but nonlinear function of the weights w_i

If we want to decrease y-hat, the backprop algorithm computes a gradient $\boldsymbol{g} = \nabla_{\boldsymbol{w}} \hat{y}.$

When we make an update $\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \boldsymbol{g}.$, the first order Taylor approximation predicts y decreases by $\epsilon \boldsymbol{g}^\top \boldsymbol{g}.$

If we want to decrease y by 0.1, first-order info suggests we set $\boldsymbol{\epsilon}$ to $\frac{.1}{\boldsymbol{g}^\top \boldsymbol{g}}.$

However, since the actual updaet includes second, third, up to l, order terms, the new value of y is given by $x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l).$

An example of a second order term is $\epsilon^2 g_1 g_2 \prod_{i=3}^{l} w_i.$, which an be exponentially large if the weights on layers i through l are greater than 1

This makes it hard to choose a learning rate, since the effects of parameter updates to one layer affect another
Second order optimization algorithms somewhat fix this issue, but for $n > 2$, an n-th order algorithm seems difficult

Batch normalization can reparametrize any deep network, reducing the problem of coordinating updates across many layers
Batch normalization can be applied to any input or hidden layer in a network
H is a minibatch of activations of the layers to normalize, arranged as a design matrix, with the activations for each example appearing in a row of the matrix
To minimize H, we replace it with $\boldsymbol{H}' = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}},$
where $\boldsymbol{\mu}$ contains the mean of each unit and sig contains the standard deviation of each unit
$\boldsymbol{\mu}$ and sig are applied to every row of H
Within each row, the arithmetic is element wise, so $H_{i, j}$ is normalized by subtracting the column $\boldsymbol{\mu}_j$ and dividing by sig_j

$$\boldsymbol{\mu} = \frac{1}{m} \sum_i \boldsymbol{H}_{i,:}$$

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m} \sum_i (\boldsymbol{H} - \boldsymbol{\mu})_i^2},$$

d is a small value meant to avoid an undefined gradient at sqrt(z) = 0
Backpropagate through these operations for computing mean and std, and for applying them to normalize H -> gradient never proposes an operation which acts on the mean or std of h_i
The normalization operation removes the effect of such an action and zero out its component in the gradient

Batch normalization reparametrizes the model to make some units always be standardized by definition
This means it does not have the problem of the learning algorithm changing the mean and variance, nor does it have imperfect normalization (which could have been a result of the cost function penalty)

At test time, $\boldsymbol{\mu}$ and sig can be replaced with running averages collected during training time

Learning $y = w_1 w_2 ... w_l x$ can be made easier by normalizing h_{l-1}
If x is drawn from a unit Gaussian, h_{l-1} also comes from a Gaussian since the transformation is linear
Applying batch normalization maintains a zero mean and unit variance on h_{l-1} which lets us learn the output y-hat = $w_l h_{l-1}$ as a simple linear function

This makes learning simple since the parameters in the lower layers do not have an effect in most cases - their output is renormalized in most cases
Without batch normalization, each update would affect the statistics of h_{l-1} -> the lower layers do not have a beneficial effect, but they do not have a detrimental effect either, as a linear network can only influence the first and second order statistics
For a deep network, the lower layers remain useful since they can perform nonlinear transformations on the data, allowing the relationship between units and nonlinear statistics of a single unit to change, while maintaining stabilized learning

Batch normalization is the most practical approach to reducing linearity in the lower layers (since the final layer can learn a linear transformation on its own)

Normalizing the mean and std of a unit can decrease its expressive power
It is common to replace the batch of hidden activations H with yH' + B, rather than just H'
y and B are learned parameters allowing for any mean and std
The new parametrization can represent the same family of functions as w/o batch norm. but it has different learning dynamics which do not depend on the lower layers (only on B) which makes applying grad. descent much easier

Neural nets take the form $\phi(\boldsymbol{XW} + \boldsymbol{b})$ where phi is a function such as ReLU
It is recommended to normalize XW + b (or just XW, since batch norm provides a learned parameter for b)
The input to a layer is the output of a nonlinear activation, making the stats of the input non-Gaussian and harder to standardize
In CNNs, should apply same $\boldsymbol{\mu}$ and sig at every spatial location so the statistics of the feature map are the same independent of location

Coordinate descent - optimizes x_i then x_j and so on, each variable one at a time
Block coordinate descent refers to optimizing for a subset off variables simultaneously
Useful when the optimization strategy for one group of vars is different than for another, or for when variables are clearly isolated

$$J(\boldsymbol{H}, \boldsymbol{W}) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} \left(\boldsymbol{X} - \boldsymbol{W}^\top \boldsymbol{H}\right)^2_{i,j}.$$

This function represents sparse encoding, where the weight matrix H must be able to decode H, the encoded activations, to reproduce the original input X
J is not convex, but if we divide the algorithm into two sets (the dictionary params W and code representations H), minimizing J w.r.t one of the variable sets is convex
Efficient convex optimization by optimizing for W w/ H fixed, and then H w/ W fixed
However, for a problem such as $(x_1 - x_2)^2 + \alpha\left(x_1^2 + x_2^2\right)$, with a small a, coordinate descent makes slow progress since the first term does not allow a single var to be changed such that the other variable will significantly differ from it

This generally applies when one variable strongly influences the optimization of another variable

Polyak averaging averages out different parameter values over time
$\hat{\boldsymbol{\theta}}^{(t)} = \frac{1}{t}\sum_i \boldsymbol{\theta}^{(i)}$

This has strong convergence guarantees for grad. descent applied to convex problems, and works well in practice for neural nets
If a neural net jumps back and forth across the valley, the average of the wts will be at the bottom
Old points, which may have not learned enough of the cost, could be ineffective w/ this method, so it is common to use exponential decay Polyak: $\hat{\boldsymbol{\theta}}^{(t)} = \alpha\hat{\boldsymbol{\theta}}^{(t-1)} + (1-\alpha)\boldsymbol{\theta}^{(t)}.$

Pretraining - training the model on simple tasks before making it take on more complex ones
Greedy algorithms break a problem into many components, solving for the optimal version of each component; they are followed by a fine tuning state where each of the components are jointly optimized to yield an optimal overall solution
Greedy solutions are often acceptable and cheaper than the alternative

Greedy supervised pretraining - algorithm breaks up problem into smaller and simpler problems

Supervised training of only a subset of layers in the final network
Each added hidden layer takes in as input of the previous fully trained layer
Alternatively, use the raw inputs and outputs of a NN or separate layer as inputs to a new NN or inputs to different layers

Transfer learning uses a different subset of examples to train a separate network, initialized with the weights from the first network

FitNets - first a wide network, which is easy to train, is trained on the data
This network becomes the teacher; a student network attempts to predict not only the output of the original task, but also the value of the middle layer of the teacher
Extra parameters which predict the outputs of the teacher middle layer from the middle layer of the student have objectives to predict the middle layer of the teacher network
Lower layers must be able to predict the middle layer and the overall outputs
This method reduces parameters and makes training deep networks easier

In practice, it is more important to choose a model family which is easy to optimize rather than a powerful optimization algorithm
Specifically, modern NNs use a design choice to use linear transformations between layers, and activation functions that are differentiable almost everywhere

Skip connections reduce the shortest path of the lower params to the outputs, mitigating vanishing gradients problem
Adding extra copies of the output attached to the intermediate hidden layers are auxiliary heads which perform the same task as the top of the network to ensure the lower levels receive a strong gradient
This method also provides a strong error signal to lower layers

Continuation methods make optimization easier by choosing initial points such that local optimization spends its time in well-behaved regions of space
Uses a series of cost functions over the same parameters
We have new cost functions $J^0(\mathbf{\Theta})$, $J^1(\mathbf{\Theta})$...J^n
J^n is the true cost function; the cost functions J^i become harder to optimize as i increases
The motivation is that better initialization points for J^{i+1} will be found by solving for J^i
Traditional smoothing methods are based on smoothing the objective function

Continuation methods were designed to avoid local minima, done through "blurring" the cost function to make it more convex, and easier to optimize

$$J^{(i)}(\boldsymbol{\theta}) = \mathbb{E}_{\theta' \sim \mathcal{N}(\boldsymbol{\theta}'; \boldsymbol{\theta}, \sigma^{(i)2})} J(\boldsymbol{\theta}')$$

Continuation methods may be effective, but high cost
Continuation methods may fail because the objective function is impossible to make convex, no matter how much blur is applied
It may also fail since the minimum of the blurred function tracks to a local minimum of J
Besides eliminating local minima, continuation methods are effective for eliminating flat regions, decreasing variance in the gradient estimate, improving Hessian conditioning, and finding a good local direction

Curriculum learning focuses on simpler tasks before complex ones; this can be done by sampling simple examples more, or by weighting their cost contribution
Average proportion of difficult examples being increased gradually improves training, while a deterministic approach (self-made curriculum) does not