

Optimization for Deep Learning

Chris Paciorek

2026-08-01

Table of contents

Overview	1
1. SGD and variations for deep learning	1
Momentum	2
RMSprop and Adam	5
2. The Bias-Variance Tradeoff, Overfitting, Regularization, and Double Descent	6
Minibatches	9
The learning rate and the process of updating weights	10
Other aspects of optimization/training	11
Learning rate decay	11
Input normalization	11
Local optima	11
Hyperparameter tuning	12
Batch normalization	12

Overview

The goal of this unit is two-fold: first to explore the concepts from the optimization unit in an important specific context and second to specifically understand more about the practical aspects of optimizing deep learning models (and how that relates back to statistical concepts as well).

Note that this unit will not cover the different architectures for deep learning models as I am woefully unqualified to do so and it is outside the scope of this course. But hopefully this will be useful for Stat 214, other courses you might take, or your work in the future.

1. SGD and variations for deep learning

We've seen some of the issues arising from simple gradient descent. One way to think of those is that reliance on the single current value of the gradient introduces sensitivity to the exact value. Also if one is at a saddle point, the gradient is small, leading to small updates.

Momentum

An common enhancement when using gradient descent is to use “momentum”. The basic idea of momentum is to modify the current gradient using past values of the gradient, using a exponentially-weighted (decaying) average of past gradient values. It can help by pushing the optimization in a consistent direction and reducing oscillations. Here’s the smoothed gradient, often called the “velocity”.

$$v_{t+1} = \beta v_t + (1 - \beta) f'(x_t)$$

β is a tuning parameter and the averaging is roughly similar to averaging over a window of $1/(1 - \beta)$ gradient values.

One can plug into gradient descent in place of the current gradient or use a variation that gives this update step, taking:

$$v_{t+1} = \beta v_t + f'(x_t) \quad x_{t+1} = x_t - v_{t+1}$$

A common value is $\beta = 0.9$.

What’s the intuition? Suppose the gradient values in a given direction have the same sign. Then the smoothed gradient in that direction will be large, so the algorithm will take large steps in that direction (including if one has just hit an area such as a saddle point where the surface is more flat), giving the algorithm inertia (momentum). Alternatively, suppose the gradient values in a given direction oscillate (change sign). Then the smoothed gradient will be small in magnitude and the algorithm will take smaller steps in that direction, damping oscillations.

[revisit quad example with fixed alpha; also show zoom in for final iterations]

```
par(mai = c(.5,.4,.1,.4))
f <- function(x){
  x[1]^2/1000 + 4*x[1]*x[2]/1000 + 5*x[2]^2/1000
}
f_deriv1 <- function(x){
  c(2 * x[1]/1000 + 4 * x[2]/1000,
    4 * x[1]/1000 + 10 * x[2]/1000)
}

grad_descent <- function(init, deriv_fun, alpha, nIt) {
  xvals <- matrix(NA, nr = nIt, nc = 2)
  xvals[1, ] <- init
  for(t in 2:nIt){
    xvals[t, ] <- xvals[t-1, ] - alpha * deriv_fun(xvals[t-1, ])
  }
  return(xvals)
}

plot_path <- function(x1s, x2s, f, xvals) {
  fx <- apply(expand.grid(x1s, x2s), 1, f)
```

```

    ## plot f(x) surface on log scale
    fields::image.plot(x1s, x2s, matrix(log(fx), 100, 100),
        xlim = range(x1s), ylim = range(x2s))
    lines(xvals) ## overlay optimization path
}

start <- c(7, -4)
xvals <- grad_descent(start, f_deriv1, alpha = 100, nIt = 200) ## This works pretty well.
x1s <- seq(-5, 8, len = 100); x2s = seq(-5, 2, len = 100)
plot_path(x1s, x2s, f, xvals)

## Zoom in near solution.
x1s_alt <- seq(-2, 2, len = 100); x2s_alt = seq(-1, 1, len = 100)
plot_path(x1s_alt, x2s_alt, f, xvals)

xvals <- grad_descent(start, f_deriv1, alpha = 150, nIt = 200) ## This oscillates somewhat.
x1s <- seq(-2, 2, len = 100); x2s = seq(-1, 1, len = 100)
plot_path(x1s, x2s, f, xvals)

xvals <- grad_descent(start, f_deriv1, alpha = 200, nIt = 200) ## This oscillates crazily.
x1s <- seq(-2, 2, len = 100); x2s = seq(-1, 1, len = 100)
plot_path(x1s, x2s, f, xvals)

```

Let's see what happens with momentum. It's much more robust to the step size!

```

grad_descent_with_momentum <- function(init, deriv_fun, alpha, nIt, beta = 0.9) {
  xvals <- matrix(NA, nr = nIt, nc = 2)
  xvals[1, ] <- init
  v <- 0
  for(t in 2:nIt){
    v <- beta * v + (1-beta)* deriv_fun(xvals[t-1,])
    xvals[t, ] <- xvals[t-1, ] - alpha * v
  }
  return(xvals)
}

xvals <- grad_descent_with_momentum(start, f_deriv1, alpha = 200, nIt = 200)
plot_path(x1s, x2s, f, xvals)
## Zoom in near solution.
plot_path(x1s_alt, x2s_alt, f, xvals)

## Perhaps leave this out.
## The alternate formulation; we need to modify alpha to get better behavior.
alpha <- 10
v <- 0

```

```

beta <- 0.9
for(t in 2:nIt){
  v <- beta *v - alpha *f_deriv1(xvals[t-1,])
  xvals[t, ] <- xvals[t-1, ] + v
}
x1s <- seq(-5, 8, len = 100); x2s = seq(-5, 2, len = 100)
fx <- apply(expand.grid(x1s, x2s), 1, f)
## plot f(x) surface on log scale
fields::image.plot(x1s, x2s, matrix(log(fx), 100, 100),
  xlim = c(-5, 8), ylim = c(-5,2))
lines(xvals) ## overlay optimization path

```

In the 1980s Nesterov came up with the idea of calculating the gradient at a position slightly ahead in the direction of the accumulated momentum, which is called Nesterov Accelerated Gradient (NAG). By looking ahead, one can reduce overshooting (in particular oscillations when one is near the optimum).

We first determine the look-ahead value:

$$x_{\text{look}} = x_t - \beta v_t$$

Then we update the velocity based on the gradient at that value:

$$v_{t+1} = \beta v_t + \alpha_t f'(x_{\text{look}})$$

with the update

$$x_{t+1} = x_t - v_{t+1}$$

[show this on example, and with zoom in]

```

grad_descent_with_nag <- function(init, deriv_fun, alpha, nIt, beta = 0.9) {
  xvals <- matrix(NA, nr = nIt, nc = 2)
  xvals[1, ] <- init
  v <- 0
  for(t in 2:nIt){
    xlook <- xvals[t-1,] -beta *v
    v <- beta *v + alpha * deriv_fun(xlook)
    xvals[t, ] <- xvals[t-1, ] - v
  }
  return(xvals)
}

alpha <- 50
xvals <- grad_descent_with_nag(start, f_deriv1, alpha = alpha, nIt = 200)

```

```

plot_path(x1s, x2s, f, xvals)

alpha <- 5
xvals <- grad_descent_with_nag(start, f_deriv1, alpha = alpha, nIt = 200)
plot_path(x1s, x2s, f, xvals)

## per cmu
grad_descent_with_nag <- function(init, deriv_fun, alpha, nIt, beta = 0.9) {
  xvals <- matrix(NA, nr = nIt, nc = 2)
  xvals[1, ] <- init
  v <- 0
  for(t in 2:nIt){
    xlook <- xvals[t-1,] -alpha *v
    v <- beta *v + alpha * deriv_fun(xlook)
    xvals[t, ] <- xvals[t-1, ] - alpha*v
  }
  return(xvals)
}

```

RMSprop and Adam

RMSprop tries to adapt the learning rate for different parameters at different rates, damping the gradient in directions with highly variable (potentially oscillating) gradient values. It maintains a weighted average of the (element-wise) squares of the past gradient values and scales the current gradient by the square root of that average. The result is to slow the learning in the directions with high variability.

Adam (Adaptive Moment Estimation) combines momentum and RMSprop and is widely-used.

It maintains a weighted average of the past gradient values and a weighted average of the squared gradients and, as with RMSprop, scales by the square root of the average of the squared gradients.

$$\begin{aligned}
 v_{t+1} &= \beta_1 v_t + (1 - \beta_1) f'(x_t) \\
 S_{t+1} &= \beta_2 S_t + (1 - \beta_2) f'(x_t)^2 \\
 x_{t+1} &= x_t - \alpha_t \frac{v_{t+1}}{\sqrt{S_{t+1}} + \epsilon}
 \end{aligned}$$

ϵ plays the usual role of dealing with denominators near zero and is often set to 10^{-8} . Note the presence of another tuning parameter, $\beta_2 \approx 0.999$.

One also often sees that one “bias corrects” v_{t+1} and S_{t+1} . Since they are initialized at zero, they are biased towards zero early in the iterations, which can be improved by using $v_{t+1}/(1 - \beta_1^{t+1})$ and $S_{t+1}/(1 - \beta_2^{t+1})$

```

grad_descent_with_adam <- function(init, deriv_fun, alpha, nIt, beta1 = 0.9, beta2 = 0.999, eps = 1e-
  xvals <- matrix(NA, nr = nIt, nc = 2)

```

```

xvals[1, ] <- init
v <- 0
s <- 0
for(t in 2:nIt){
  grad <- deriv_fun(xvals[t-1,])
  v <- beta1 * v + (1-beta1) * grad
  s <- beta2 * s + (1-beta2) * grad^2
  v_use <- v; s_use <- s
  if(bias_correct) {
    v_use <- v_use / (1-beta1^t)
    s_use <- s_use / (1-beta2^t)
  }
  xvals[t, ] <- xvals[t-1, ] - alpha * v_use / (sqrt(s_use) + eps)
}
return(xvals)
}

alpha <- 1
xvals <- grad_descent_with_adam(start, f_deriv1, alpha = alpha, nIt = 200, bias_correct = TRUE)
plot_path(x1s, x2s, f, xvals)

alpha <- .1
xvals <- grad_descent_with_adam(start, f_deriv1, alpha = alpha, nIt = 200, bias_correct = TRUE)
plot_path(x1s, x2s, f, xvals)

alpha <- .01
xvals <- grad_descent_with_adam(start, f_deriv1, alpha = alpha, nIt = 200, bias_correct = FALSE) # S1
plot_path(x1s, x2s, f, xvals)

```

2. The Bias-Variance Tradeoff, Overfitting, Regularization, and Double Descent

A core principle in statistics is the bias-variance tradeoff. Models with few parameters and/or limited flexibility/expressivity will be biased in fitting the data because they are not flexible enough to capture all the structure in the data but have low variance because there is a lot of data relative to the number of parameters. Models with many parameters will have little bias, but high variance, with the variance caused by the model fitting to noise in addition to structure/signal in the data. Remember that in a classical statistical context, signal is what occurs in all the different samples that one might observe, while noise differs from sample to sample. The result is a U-shaped curve for the test error, which decreases initially with more parameters (from lower bias) and then increases (from higher variance).

Some strategies for addressing the bias-variance tradeoff to try to fit signal but not noise include model selection (of which variable selection is one form) based on optimizing a loss that tries to capture test (not train) error, penalization/regularization to avoid overly variable fitting, and early optimization stopping (which can be seen as a kind of implicit regularization), such as by monitoring hold-out error.

Therefore, the classical statistical paradigm for fitting parametric models is that, holding the sample size fixed, as the number of parameters increases, the model will overfit. What does this mean?

There will be enough parameters to start fitting to noise in the data as well as to real features in the data. One will see very good predictive performance on the training data, and with enough parameters (e.g., as many parameters as observations) one can exactly fit the training data, even though some portion of the variation in the training data is noise. Then when the fitted model is applied to test data, which has different values of the noise than the training data, predictive performance will be much lower. In other words the generalization error (difference between train and test error) is large.

We can see this for a simple nonparametric regression setting in one dimension. Standard generalized additive model tools (mgcv::gam in R) fit a penalized spline with the amount of regularization determined based on trying to estimate test loss in some fashion (GCV, AIC, REML, etc.).

```
set.seed(1)

fun <- function(x) x^2 + 2*x -5
n = 200
x = seq(-2, 2, length = n)
f <- fun(x)
y = rnorm(n, f, 1)

# Unpenalized, ~150 knots
library(splines)
Xmat150 <- bs(x, df = 150)
mod150 <- lm(y ~ Xmat150)
pred150 <- predict(mod150)

# Unpenalized, 1 knot
Xmat4 <- bs(x, df = 4)
mod4 <- lm(y ~ Xmat4)
pred4 <- predict(mod4)

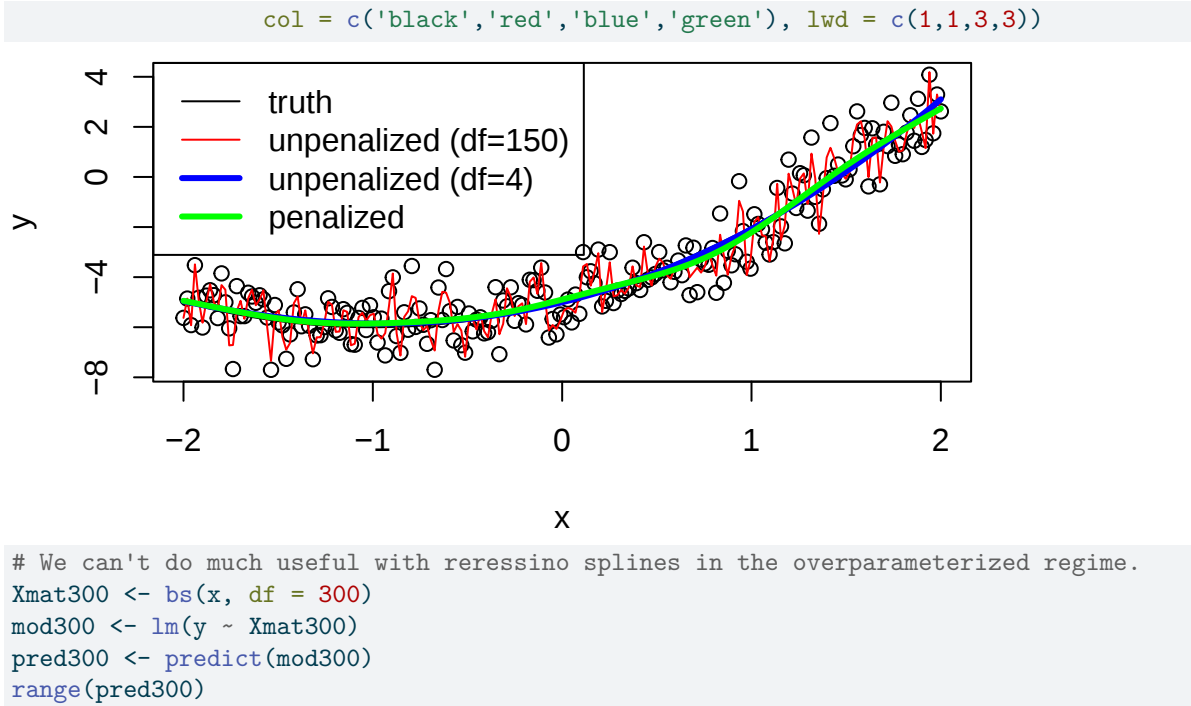
# With automatic penalization
library(mgcv)
```

Loading required package: nlme

This is mgcv 1.9-4. For overview type '?mgcv'.

```
gam_mod <- gam(y ~ s(x))
gam_pred <- predict(gam_mod)

plot(x, y)
lines(x, f)
lines(x, pred150, col = 'red')
lines(x, pred4, col = 'blue', lwd = 3)
lines(x, gam_pred, col = 'green', lwd = 3)
legend('topleft', legend = c('truth', 'unpenalized (df=150)', 'unpenalized (df=4)', 'penalized'),
```



```
[1] -7.024817e+89  7.638885e+89
```

[This well-known ArXiv paper](#) shows that deep neural nets (in particular ConvNets for image processing) can perfectly fit noise in observations, resulting in zero train error and large test error. This indicates that nothing in the model architecture is providing regularization. And while one can introduce regularization explicitly (such as through weight decay and dropout), in many uses regularization is not used and the authors show the explicit regularization helps a bit but that unregularized solutions do nearly as well. So how can deep neural networks avoid overfitting? The authors say that it is SGD itself that results in a regularized solution that generalizes well, without requiring explicit regularization.

Belkin et al. introduced the term “double descent” to describe how increasing model size past the interpolation point ($n=p$) improves test error (hence the second descent after the first descent as one initially adds parameters) (see Fig. 1). They discuss the idea that with more parameters (a larger model/function class) one can find a model that fits the training data perfectly while also being smooth (having small norm on the model parameters/coefficients) (and thereby generalizing well). They illustrate this by explicitly including the norm in their loss function. And as above, the reason that neural net performance is good (even without regularization) is that standard optimization procedures used for these problems tend to find small norm solutions.

In some cases, practitioners have also seen double descent occur in epoch-space - that as the test error decreases then increases as the number of epochs becomes larger and then decreases again, presumably as the optimization finds more smooth solutions (perhaps because it [favors flatter areas of the parameter space](#)). Note that one can potentially achieve good performance based on early stopping by monitoring test error on a held-out set (or something similar), since the model weights/parameters generally start

at small values and take some time to become larger, which is generally when the overfitting happens.

PNAS paper: <https://www.pnas.org/doi/pdf/10.1073/pnas.1903070116>

Let's see what happens if we use neural networks to fit the nonparametric regression. With a small model (64 neurons), if one stops before a lot of iterations (epochs), there is not much overfitting. If one lets the optimization continue, training error decreases slowly and we see some additional overfitting. Using a larger model illustrates (larger layers or two levels) the overfitting quite noticeably unless one stops early, although there is clearly still some regularization happening (given we know that the model is large enough to exactly fit the data).

Minibatches

Stochastic gradient descent (and related methods described above) substitute a noisy (but unbiased) estimate of the gradient in place of the full gradient, as discussed earlier in the optimization unit. That noisy estimate is generally an estimate based on the gradient contributions from a (random) subset of the observations, using a small subset such as 32-512 data points. There are different ways to choose the minibatches – one potentially important choice is whether to cycle through all the observations in minibatches in an epoch or randomly create a minibatch. The latter would be true SGD and is easier to analyze mathematically because the minibatches are independent but in practice researchers have found good results if one randomly shuffles the observations at the start of each epoch (e.g., PyTorch's DataLoader with `shuffle=True`) and within an epoch cycle through all the observations in minibatches.

One thing to think about in choosing a minibatch size is that the minibatch datasize fits in CPU/GPU memory and/or in the CPU cache.

Recall from the main optimization unit that each step only moves us somewhat in the right direction and we don't want to spend a lot of time choosing the exact right amount to go in that single step. Similarly we don't want to spend a lot of time choosing exactly the right direction. We are iteratively exploring a space and making many descent steps quickly is better than making one "best" step.

Apparently the term 'batch' refers to the entire dataset, hence 'mini-batch' is a small subset of the dataset. That terminology might be confusing (since I might think of a batch as being the same as a chunk or shard or subset) but seems to be standard.

One advantage of minibatches is purely computational efficiency. We can choose a reasonable direction to go and make progress in that direction without having to compute the gradient for all of the observations. This was the first motivation. The connection with low generalization error and implicit regularization was realized later. In [this paper](#) the authors argue that use of small batches results in finding optima in flat parts of the space that give smoother solutions that generalize better. However, [this paper](#) provides evidence against the idea that finding solutions in flat areas is what explains the good generalization performance).

One would also want to consider the learning rate as that plays an important role both in computational efficiency and in how the optimization moves through the space and settles on a solution. In [this paper](#), the authors discuss first that with a very small learning rate, the optimization with minibatches follows the gradient flow to a local minimum but with larger learning rates the optimization path is what one would get if optimizing a regularized objective function.

For many years, the standard approach was to use multiple minibatches in an epoch and train for many epochs. More recently there has been some focus on using a very small number of epochs, particularly in settings with huge data sizes [find Jingfeng citations]

What happens if we use minibatching in the regression example? By comparing with the previous results, we can see a few things:

- There's not a lot of evidence in this particular example that using minibatches results in a much lower generalization error or test error near the oracle error.
- In general (particularly larger models and many epochs), we do still see a generalization gap that grows with the number of epochs.
- Early-ish stopping combined with minibatches often seems to work well.

[at one point (n=200?) I was seeing minibatching help with generalization error; not sure why not anymore]

I say “early-ish” because 2000 epochs is quite large for such a small problem, so investigating what happens with 25000 epochs is interesting conceptually but presumably not something one would do in practice.

The learning rate and the process of updating weights

The learning rate is also an important part of the puzzle. It affects computational efficiency (overly small values will take a long time to reach an ‘optimum’), but also the performance of the trained model and interplays with the batch size since the amount of information in any given step relates to the number of observations in the batch.

Suppose we are interested in whether we can improve the value of a specific weight in a neural network using sigmoid activation. Consider that at a given layer in the network, $a^{(l)} = \sigma(\sum_k w_k^{(l)} a^{(l-1)} + b^{(l)})$. With a bit of investigation of the backpropagation algorithm for calculating the gradient of the loss with respect to the weights (e.g., considering the simplified update for a neuron at layer l as $a^{(l)} = \sigma(w^{(l)} a^{(l-1)} + b^{(l)})$, with gradient $\sigma'(w^{(l)} a^{(l-1)} + b^{(l)}) a^{(l-1)}$), one can see that the update to a weight will be small if:

- The input neuron, $a^{(l-1)}$ has a small value (low activation)
- The output neuron has saturated (the value of $w^{(l)} a^{(l-1)} + b^{(l)}$ is very large or small, such that the derivative of the sigmoid function is near zero.

This plays out differently when using ReLU activation functions since their derivative is zero for negative inputs and unbounded for positive. The unboundedness can result in less stability in the updates. And the zero derivative can mean that a neuron gets stuck in the “off” position if its inputs are always negative. In backpropagation, the weights/coefficients that feed into such a neuron don't get updated through that neuron (though they would through other “live” neurons).

One can also see a slow learning rate if you have a deep network and small magnitude weights, in which case the gradient behaves like the product of the weights (modulated by the activation function) and can be quite small. This is known as the vanishing gradient problem; with the counterpart exploding gradient problem when the weights are large, which can result in instability in fitting since individual gradient descent steps can be very large.

Careful initialization can help with vanishing/exploding gradients. In particular the magnitude (quantified by the variance) of the initial values in a layer should scale inversely with the number of weights in the layer, to give weights with values whose magnitude is around 1. For RELU, the standard is to scale as 2 divided by the number of weights.

Other aspects of optimization/training

Learning rate decay

If one chooses a constant value of the learning rate, α , when using mini-batches, one will never converge to a (local) optimum but will oscillate around the optimum because the gradient estimate at each step will not be exactly zero. So an additional option for improving optimization is to set the learning rate to decay with the number of epochs (or the number of steps).

In recognition of this, and also that as one gets closer to the minimum one generally would want to be more careful about the steps one takes, one often uses a “schedule” by which the learning rate decreases as a function of the epoch number.

A common schedule is:

$$\alpha = \frac{1}{1 + \rho m} \alpha_0$$

where α_0 is the initial learning rate, ρ is a decay parameter, and m is the number of epochs. One may need to tune ρ . Or one might use exponential decay: $\alpha = \rho^m \alpha_0$. And there are other approaches that are used in practice.

Input normalization

It’s generally a good idea to ‘normalize’ your inputs (i.e., shift and scale them to remove the mean and standardize the variance). This generally produces an objective function that is more spherical and less elliptical/elongated. We saw in the Optimization unit that part of the difficulty with pure gradient descent is that it doesn’t account for how the gradient changes as one moves through the space and that Newton’s method, by using the Hessian, accounts for this and helps avoid big oscillations. Of course in deep learning we can use 2nd derivative information for efficiency reasons, so transforming your inputs is quite useful. Another way to think about this is that if we have an elliptical surface, we would want the learning rate to vary by dimension, whereas with a spherical one, we can use a single learning rate, which of course is what one wants, to avoid a high-dimensional hyperparameter search space.

Local optima

In low dimensions, it’s easy to think about having local optima that an optimization algorithm will get stuck in. But in high dimensions, having a local optimum requires that the Hessian in all directions be zero. This is much less likely than that some of second derivatives will be positive and some negative, in which case one is at a saddle point. So long as your optimization algorithm doesn’t get stuck at exactly the saddle point (which it shouldn’t because of the stochastic gradients (and as well from using techniques like momentum), then this shouldn’t cause a big problem.

What tends to be more of a problem is having slow learning in plateaus where the gradients are shallow and steps are small. Momentum and related techniques can help with this.

Hyperparameter tuning

We've seen that there are a variety of tuning parameters, including model architecture and tuning parameters. In his deep learning course, Andrew Ng suggests first considering the learning rate, α , and then after that the momentum hyperparameter, the mini-batch size, and the number of hidden units.

An obvious strategy is to do a grid search over the hyperparameters. An alternative is to choose the points at random. This results in more unique values of each hyperparameter, which can be particularly useful if it turns out one or more of the hyperparameters don't really matter at all, which when using a grid would mean that a bunch of your grid points are redundant.

One might also try to zoom in on the best values by using a second round of optimization after determining roughly the right magnitude of values initially.

What about the scale of values to consider? Often we want to range over multiple orders of magnitude, so we want to search on the log scale. For the parameter in an exponentially-weighted average, you'd want to search over $1 - \beta$ on the log scale.

Batch normalization

Another common strategy when fitting neural networks is to normalize the values of the hidden units (usually before passing through the activation function) to have a particular layer-specific mean and variance (that might be learned during training). The intuition is similar to that for normalizing the inputs, to have all the predictors on the same scale and hopefully have more spherical objective function contours.

One way to think about the effect of batch normalization is that it keeps the magnitudes and spread of the hidden unit values stable over the iterations, reducing what one might think of as a sort of 'covariate shift' within the model (since the hidden unit values are constantly changing as the optimization updates the parameters (the weights)).