When working in neural network, there are lots of decision needs to be made such as number of hidden layers, number of hidden units, learning rates, what activation function to use, and etc. On first attempt, it's hard to get all parameters right, by experiment we find the best parameters based on outcome. It's iterative process to find the parameters.

There are so many applications for deep learning such as Ads, search, security, logistic regression. Computer vision, speech recognition and etc.

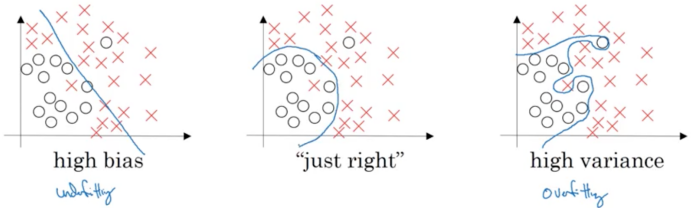
Train/Dev/Test Set

Usually we divide the data into three sets of training set, development set or cross validation set and test set. We pick the model on development set based on error. When we have around 100 to 10,000 datasets, it's common to divide into 60-20-20 percent, but in big data that we have around 1,000,000 training set, we divide them into 98-1-1 percent, since more of training causes a better understanding and modeling (like 10,000 for testing and 10,000 for dev).

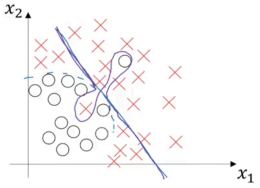
We may use training set from cat pictures from the web, and dev/test set taken from cat pictures that user using in the app. *It's important that the development set and test set come from the same distribution*. It's OK not to have test set, since it's only for unbiased estimate of network. In this case we have training and dev set but sometimes people call it training and test set and they use the test set for cross validation.

Bias and Variance

In deep learning error, there is less of bias variance tradeoff. In image below, we may use linear regression and make a model that doesn't fit well and cause a high bias problem, or we may use a neural network with so many hidden layers and make a great fit or overfit the training set and make a high variance problem, we may use some classifier that fits just right to the examples.



In 2D examples with 2 features we can plot and visualize the data. In features more than more, there are some ways to identify the bias and variance. Using the training and dev set error we can identify. For example, training set of 1% and dev error of 11% shows the high variance problem, or if it shows 15% error in training and 16% error in dev set is high bias problem. On other hand, if we get 15% training error and 30% dev error means that we have high variance and high bias problem. If we get 0.5% error on training and 1% in dev set means we have low bias and low variance model. The base error which is human error is 0%. Imagine we have blurry images that no classifier can predict them with low error, then base error is higher since probably human will have a hard time distinguishing the images as well. The plot on the right shows high bias and high variance model. It's high bias since most of the examples are separated by linear line and it's high bias since it tries to



overfit the examples. In summary, it's high bias in some regions and high variance in some other regions.

Basic Recipe for Machine Learning

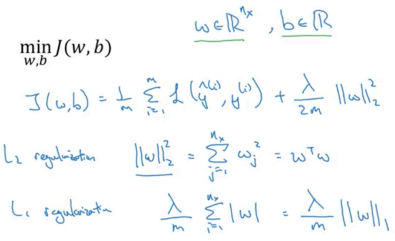
When training neural network, we evaluate it.

- Is it high bias? Must look at training performance. In this case we may use more hidden layers or use bigger networks, train longer.
- Is it high variance? Must look at dev set performance. In this case, we may get more data, use regularization

So depends on the problem, we use different approach. In modern deep learning, we have enough tools to reduce the variance and bias. Almost always, getting a bigger network reduces the bias without hurting the variance, as far as we regularize appropriately. That's the main reason, we don't need to worry about the variance bias tradeoff in neural network. Training a bigger network never hurts, except the computation time. Using regularization might increase the bias little bit, but using a bigger network it won't affect much.

Regularization

If we have high variance problem, one of the first solution is to use regularization. Other solution getting more training data. Let's start with logistic regression, we try to minimize the cost function (J). By adding λ we have regularization in our equation. L2 regularization is the most popular regularization form. By using L1 regularization, w will end up being sparse meaning that it has lots of 0 in it, which helps taking less space in memory and compressing your model.



In neural network, we use the regularization in similar matter as well.

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|w^{[l]}\|^2$$

$$\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}(w_{ij}^{[l]})^2 \qquad w: (n^{[l]}, n^{[l-1]})$$

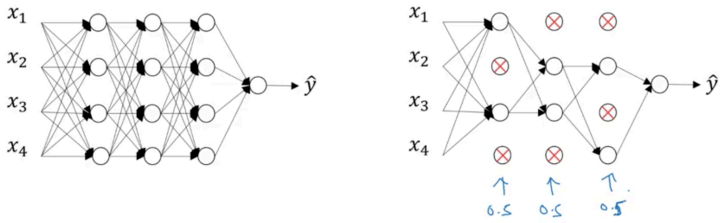
Where n is number of hidden layers in layer l. The matrix norm is often called the Frobenius norm (w).

How Regularization Reduces Overfitting?

Imagine we have a neural network with some hidden layers that already give us overfitting. If we set λ to a really big number, the weight matrix must be near 0, meaning that many nodes in hidden layer will be zeroing out. In this case, the complex model will be simplified very much (maybe to linear regression). This will take you from high variance to high bias. By finding a right value of λ, we will be getting a good model.

Dropout Regularization

Another form or regularization is dropout. Let's say we have a neural network as shown in the image below on the left and is overfit. With dropout, we go through each of the layers and set some probability of eliminating a node in neural network. For example, we toss a coin and have 0.5 chance of keeping and 0.5 of eliminating a node. After that we eliminate all of ingoing outgoing bridge from it and make the neural network simpler, then doing backpropagation process. Then again do the eliminating some other nodes and do the training.



Remember to do the testing and validation on no dropout network. In summary dropout randomly knocking out the units in neural network and amazingly works great. Even it works better than regularization.

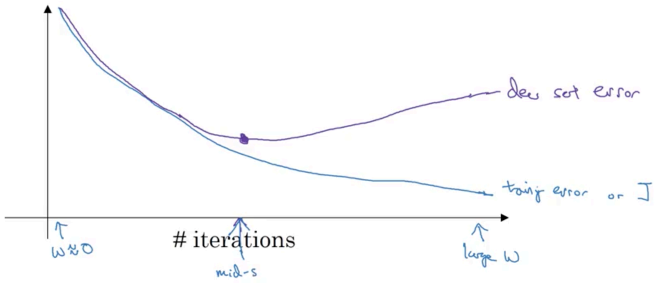
In dropout, the model can't rely on any features since it can get knocked out in a random selection of units. Therefore, it tries to spread out the weight, rather than focusing on few features. Therefore, it helps taking care of overfitting. In different layers, we may assign different keep prob. Usually all of input are kept in neural network. Dropout is mainly used in computer vision since we have a lot of features. In computer vision, there are mainly overfitting problem and that's why dropout is used.

The downside is that cost function is not clearly defined due to the dropout. In this case, calculate the cost function as a number of iteration first, then turn on the dropout.

Other Regularization Techniques

Adding more dataset helps. If you can't collect a new one try to synthesize data. For example, if working in images, make images by rotating, flipping the image. It's not as good as collecting new images, but it's a way of making your dataset bigger.

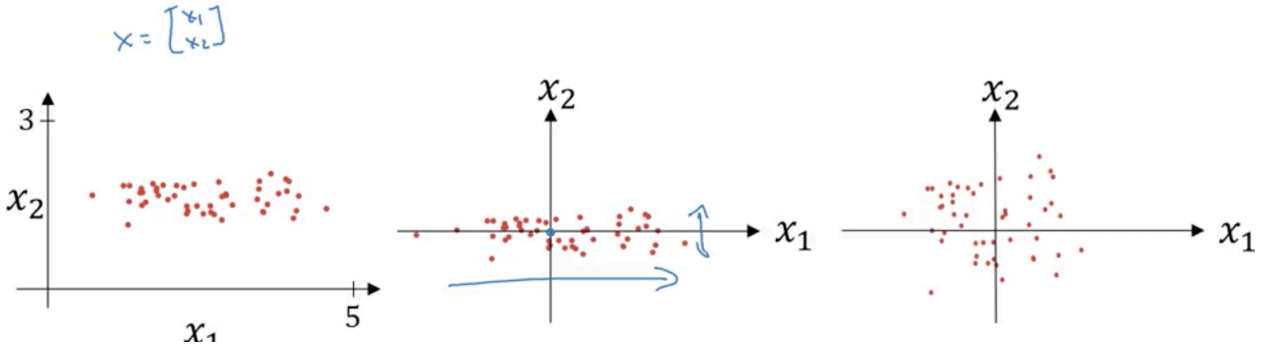
Early stopping: plot the dev and training error (J) vs number of iterations. Training error usually goes down and dev error goes down and then increases. In early stopping, it chooses the number of layers when dev error is minimum. At first, we randomly assign values to w close to 0, but as we train w gets bigger and bigger in neural network. Early stopping, stops the algorithm in mid-size of w. Normally we like to have one set of parameter to worry about like cost function J and after that we need to be careful not to overfit and use regularization techniques. Sometimes it's called orthogonalization, meaning think about one task



at a time. The downside is that we use one tool to take care of optimizing cost function and not overfitting, since we are stopping early before dev error increases.

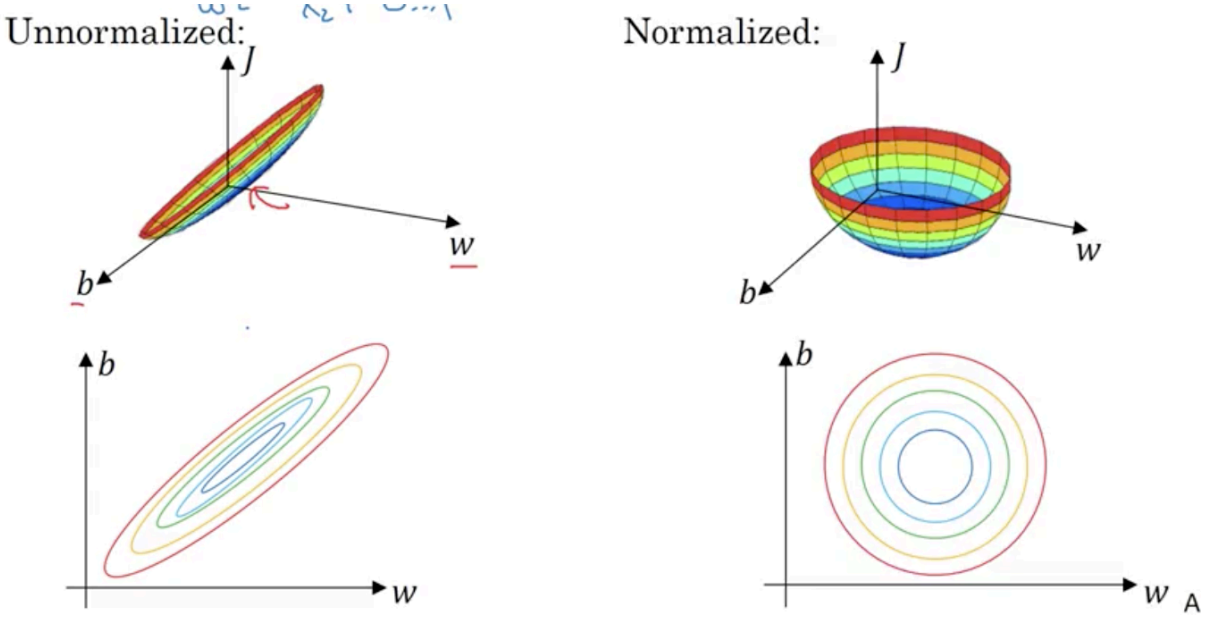
Setting Up Optimization Problem- Normalizing Inputs

When training the neural network, normalizing the inputs will speed up training. Normalizing the input corresponds to two steps: the first step is subtracting out or to zero out the mean, the second step is normalizing the variances. Imagine we have two features shown in image below, first we made the mean 0. As we can see the x2 has much more variance than the other features. By normalizing the variance, we get similar variance for both features; equal to 1.(right image below).



Make sure you normalize the test and training set in a same way, because we want both set goes through same transition.

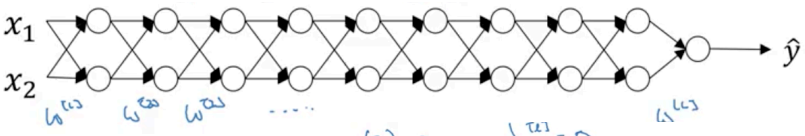
If we use non-normalize inputs, the cost function will be squishing in some directions.



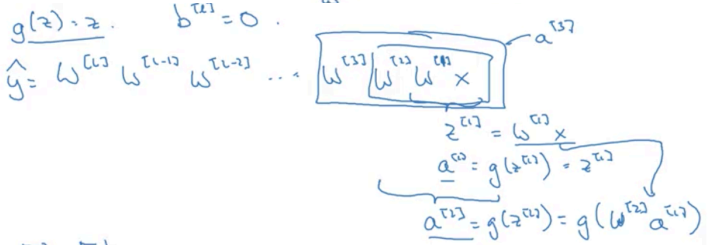
If we don't normalize the input, we probably need to use a small learning rate in gradient descent.

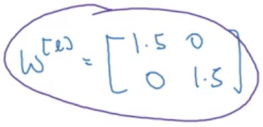
Vanishing/Exploding the Gradients

One of the problems of training neural network is vanishing/exploding the gradients. It means that when we are training a very deep network, the derivatives can get very very big or very very small. By using a random weight initialization, we can reduce the problem significantly. Imagine we are training a neural network as shown below with 2 nodes on each layer.



Imagine we are using linear activation function of g(z)= z. Let's say the weight function is abit more than the identity like 1.5 times of matrix identity. Value of y will explode exponentially (y=$1.5^l x$) in deep neural network.

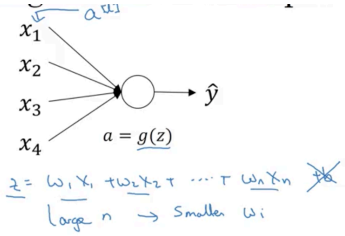


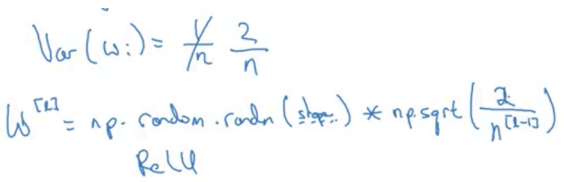


On the other hand, imagine the weight matrix is value of less than 1 like 0.5, then the y will vanish exponentially (y=$0.5^l x$) in deep neural network.

Weight Initialization for Deep Neural Network

We know that gradient in deep learning can explode or vanish. The partial solution is to use a better initialization (this doesn't solve the problem completely though). Let's start with a simple example and later we generalize the finding. The larger n (number of input features) is, we want to make the w_i smaller, since the z is the summation of all w's. we define variance of w_i's to be 1/n, var(w_i)=1/n. It's been shown that if we set the variance 2/n works better. So w can be defined as (ReLU):

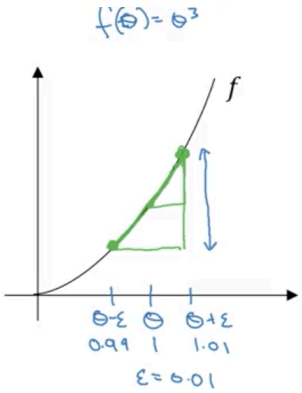




Initializing it this way probably helps since the value of w will be a little bigger or smaller than 1.

Numerical Approximation of Gradients

When doing backpropagation, there's a testing called gradient checking to help you make sure you got the details right in backpropagation. It turns out we get a better approximation of gradient if we use the point after and before for calculation.



$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta)$$

When using this methods in backpropagation, it turns out it runs twice as slow as we were to use a one-sided defense in calculation, but it worth using this method since it's more accurate.

$$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \qquad O(\varepsilon^2) \quad 0.01$$

Using this method, shows the error is in order of squared ε. So if epsilon is 0.01 then the error is in order of 0.0001.

Gradient Checking

This helps saving tons of time to find faults in algorithm. Gradient checking is applied during backpropagation. Here is the process to do so:

Take $W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ.

concatenate

$J(W^{[1]}, b^{[1]}, \ldots, W^{[L]}, b^{[L]}) = J(\theta)$

Take $dW^{[1]}, db^{[1]}, \ldots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$.

concatenate

Is $d\theta$ the gradient of J

Finally write a for loop for cost function to do gradient checking.

for each i:

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \ldots, \theta_i + \varepsilon, \ldots) - J(\theta_1, \theta_2, \ldots, \theta_i - \varepsilon, \ldots)}{2\varepsilon}$$

$$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i} \quad | \quad d\theta_{approx} \overset{?}{\approx} d\theta$$

Check

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$$

$\varepsilon = 10^{-7}$

By calculating the approximate value of cost function and actual value of cost function and setting a threshold we can check whether our neural network works properly or not.

Gradient Checking Implementation Tip

Here is some useful tip:

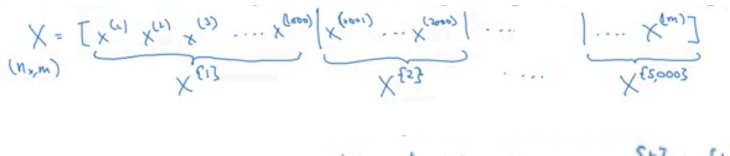
- Don't use in training – only to debug
- If algorithm fails grad check, look at components to try to identify bug.
- Remember regularization.
- Doesn't work with dropout.
- Run at random initialization; perhaps again after some training.

Optimization algorithm

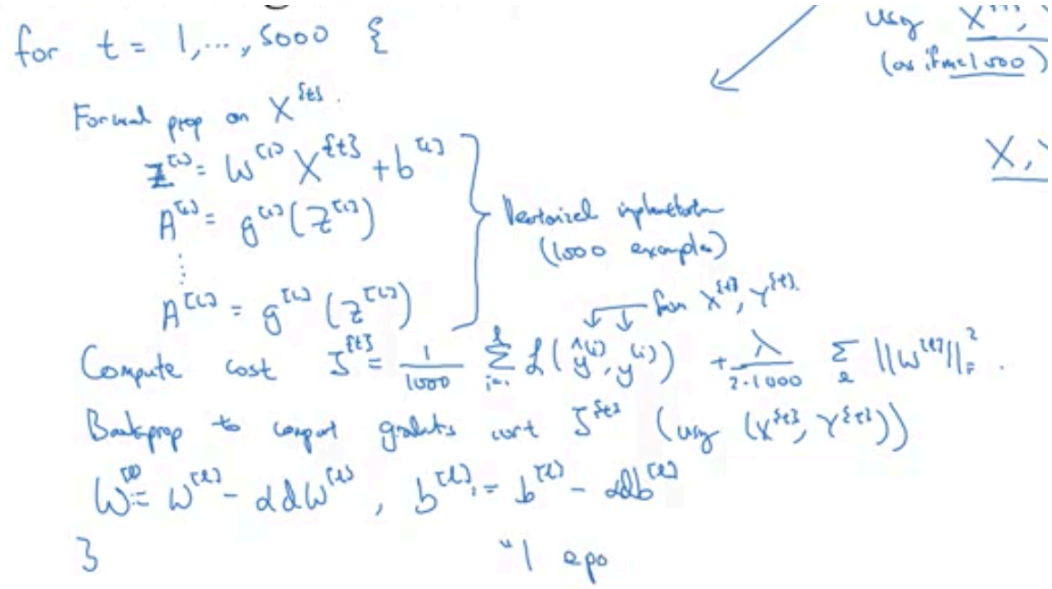
Optimization techniques enable us to train the neural network faster. Applying machine learning is highly iterative process. We need to train a lot of models to find the one that really works. It helps if we can build a training model quickly. Deep learning does not work great in huge dataset. The training of data in DNN is very slow. Therefore, having some algorithms can really speed up the efficiency. In following note, we will talk about several ways of optimizing the DNN.

Mini batch gradient descent

We know vectorization allow us to compute examples more efficient without an explicit formula. We take training examples and stack them into the huge matrix of X and Y. The dimension of X is n by m and Y is 1 by m. If m is relatively large (like 5 million), even using vectorization, the process still can be slow. Using gradient decent, we need to process the whole training set before we can take step into the gradient descent. Then we have to process the whole training set before taking the next step into the gradient descent. If we let the gradient descent to take some progress before finishing the entire process on the training examples. If we split up the training set into smaller sets (small training sets), we can make the process of training faster. Let's say each small batch have about 1,000 examples each (we show it with X superscripted with curly bracket showing the annotation) in a 5 million example set. This means we have 5,000 of the mini batches.



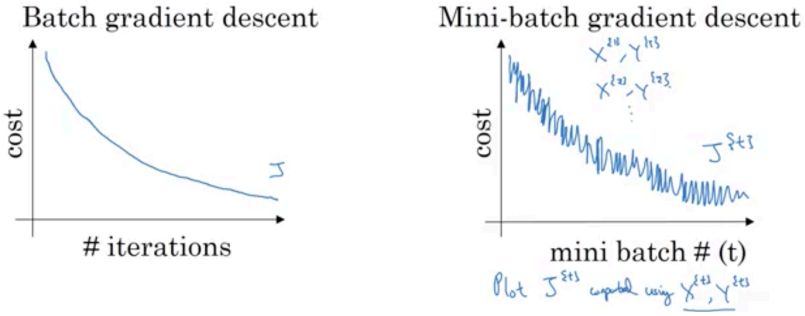
We create mini batch for both X and Y in similar manner. It's called mini batch gradient descent, since we process each batch individually. Epoch is a single pass through the training set. Algorithm below shows an epoch of the process. Whereas in mini batch gradient, a single pass through training allows you to take bunch of (in our example 5,000) gradient descent step until in converges.



Almost everyone uses mini batch gradient descent in deep learning when training lots of example, since its much faster than batch gradient decent.

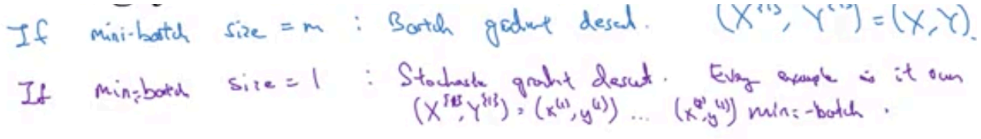
Implementation of mini batch gradient decent

In this session, we dive deeper into better understanding of this technique. In batch gradient, decent on every iteration we go through the entire training set and we expect the cost goes down on every single iteration. If it goes up, even a bit, it means something is going wrong. On the other hand, the mini batch gradient decent cost function may not decrease. It is because we are training on a different batch each time. The trend is downwards with some fluctuations.

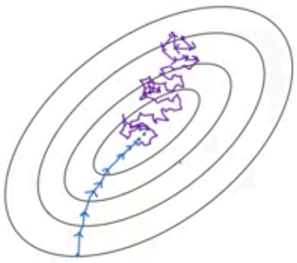


The reason is that some training examples are harder than rest and causes the cost function J increases during training.

How can we choose the mini batch size? Let's take a look at extreme cases first.:



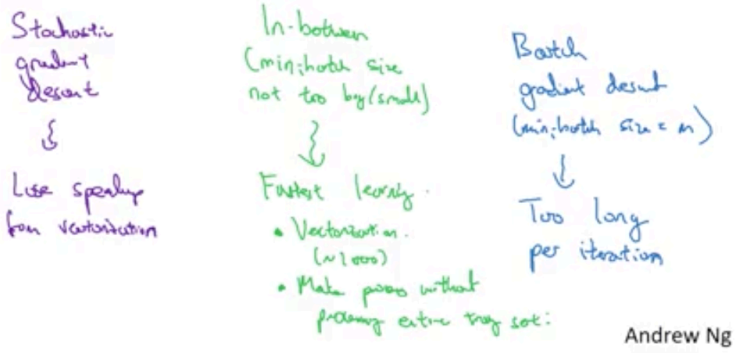
When size of mini batch is equal to m, it starts with relatively larges steps and low noise and keep marching toward the minimum. In contrast in stochastic GD, we take step only with one training example. Sometimes it causes it get result in wrong direction and sometimes hit toward the global minimum (shown in purple in the figure). Noise can be reduced by using smaller learning rate.

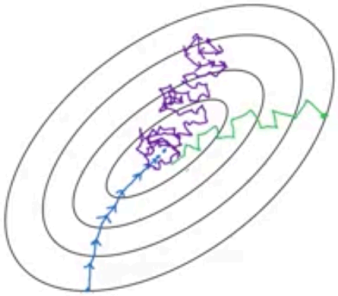


Batch GD takes too much time in practice to get result from a large training set. But if we have small training set, we may use batch GD.

Disadvantage of stochastic gradient descent is that we lose all of our speed up from vectorization. Processing each training example is very inefficient.

In practice, the mini batch size should not be too big or too small (between 1 and m). Mini-batch GD gives the fastest learning in practice. In goes toward the global minimum faster than stochastic gradient decent.



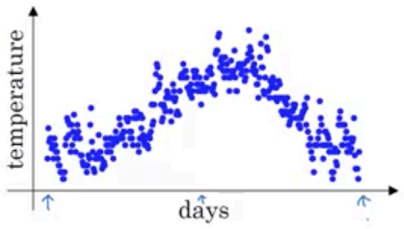


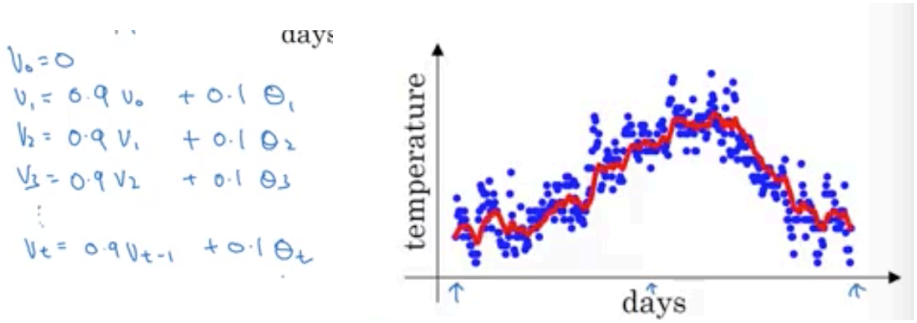
Because of how computer memory is laid out and accessed. Sometimes the code run faster if mini batch size is a power of 2. Usually mini batch size of less than 1,000 are more in common. Make sure mini batches fits in CPU/GPU memory. Otherwise the performance suddenly falls off the cliff. Mini batch size is another hyperparamters that you might do analysis to find the best size. Try several values and pick the one that makes your algorithm more efficient.

There is even more efficient algorithm than mini BGD and we discuss them in next session. In order to use them, first we need to learn about exponentially weighted average.

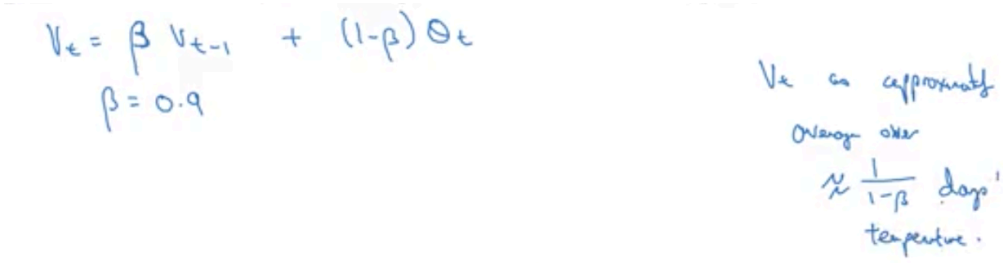
Exponentially weighted averages

Imagine we have daily temperature of London. The plot looks like the right figure during the year. It looks quite noisy. Finding the trends of moving local average, here is what we can do. Initialize the v to be zero and use equation below to calculate the temperature of each day. We get the moving average that's shown in red in the image.

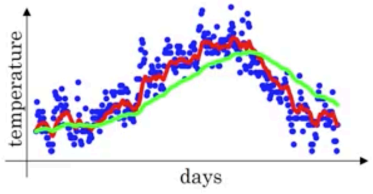




Here is how we get moving average known as exponentially weighted average of daily temperature.

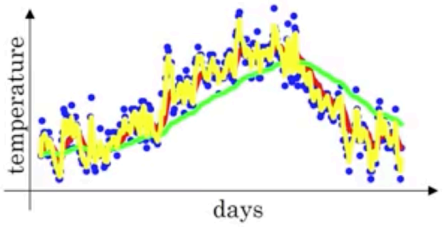


So, when beta is 0.9 it means we are averaging over 10 days of temperature. If beta is 0.98 then it means averaging over 50 days of temperature, which gives us the green line in the plot. The plot is smoother since we are averaging over more days. On the flip side, the curve shifted to the right. larger exponential values adapt more slowly when temperature changes. When beta is 0.98, it gives a lot of weight to the previous value of temperature.



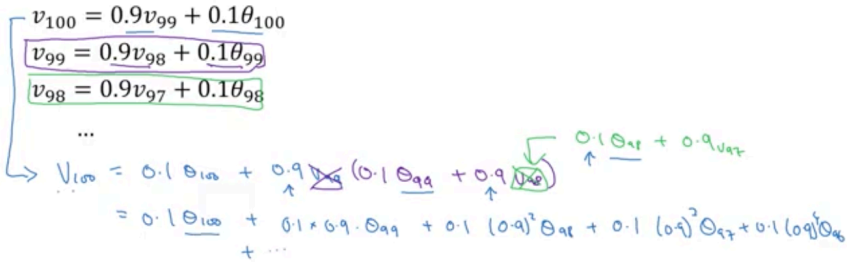
When beta is 0.5, we are averaging over 2 days temperature, shown in yellow line. It is more noisy and susceptible to outliers and it adapts much more quickly to temperature changes.

By changing the hyper parameter, beta, we can get the result we are searching for better. The equation below is known as exponentially weighted averages:



$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

Here is how it works. Let's write one equation and expand it more. It's basically sum of weighted average. It's basically slowly decaying function.



All of the coefficient adds up to a number close to 1, up to a detail called bias correction which we discuss it later. When beta is 0.9, it's averaging over the last 10 days, since after 10 days, the weight drops below the third of the weight of current days temperature.

$0.9^{10} \approx 0.35 \approx \frac{1}{e}$

Here is how we implement it:

$$v_0 = 0$$
$$v_1 = \beta v_0 + (1-\beta)\theta_1$$
$$v_2 = \beta v_1 + (1-\beta)\theta_2$$
$$v_3 = \beta v_2 + (1-\beta)\theta_3$$
$$\dots$$

$V_\theta := 0$
$V_\theta := \beta V + (1-\beta)\theta_1$
$V_\theta := \beta V + (1-\beta)\theta_2$
$\vdots$

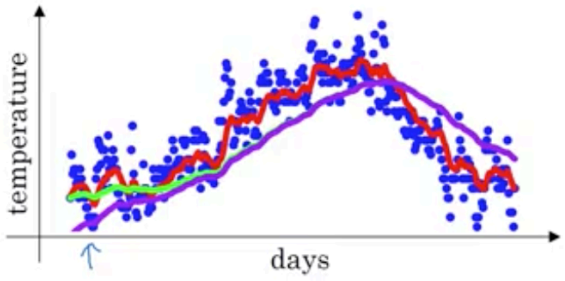
$V_\theta = 0$
Repeat {
  Get next θ_t
  $V_\theta := \beta V_\theta + (1-\beta)\theta_t$
}

The advantage of exponentially weighted average formula is takes very little memory. It only needs to keep one number of beta in the memory and keep on overwriting the formula based on the latest values it got.

Another way is calculating the average value of temperature over 10 days explicitly. The disadvantage of this method is that it requires more memory and more computationally expensive.

Bias correction exponentially weighted average

Bias correction makes computation of these averages more accurately. Using exponentially average values gives the purple line which starts below the green line, which bias correction is used. And the reason is a bad estimate od first temperature since V is initialized by 0. We can modify estimate of first few days estimate using bias correction. We can remove the bias by:



$v_t = \beta v_{t-1} + (1-\beta)\theta$
$V_0 = 0$
$V_1 = 0.98 V_0 + 0.02\,\theta$

$$\frac{V_t}{1-\beta^t}$$

$t=2: \quad 1-\beta^t = 1-(0.98)^2 = 0.0396$

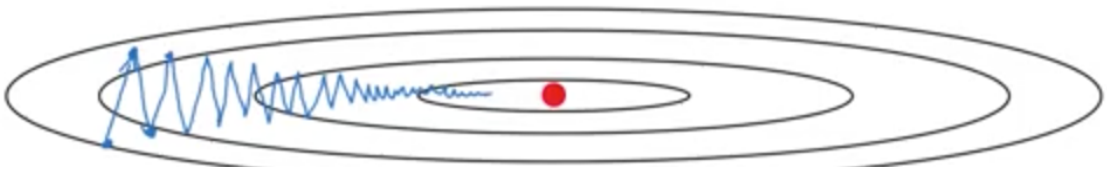
$$\frac{V_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

It helps estimating the first few temperatures more accurately, and after a while green and purple line overlap.

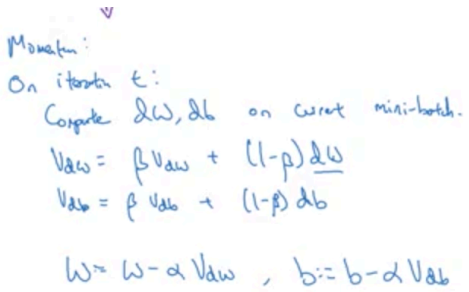
Gradient descent with momentum

This algorithm almost always works better than GD. The idea is to compute an exponentially weighted average of the gradients, and then use that gradient to update the weight instead.

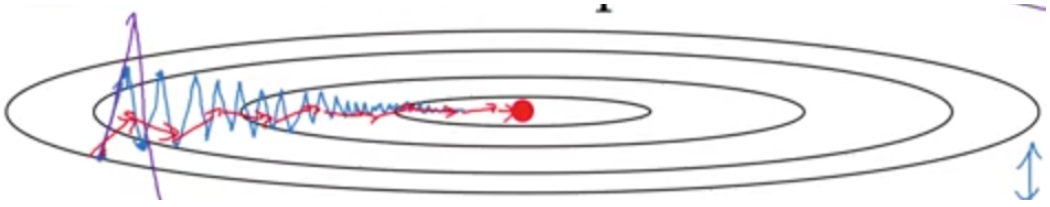
Imagine we have a cost function which its contour looks like figure below. The red dot shows the position of the minimum. Using larger learning rates causes over shooting or divergence of algorithm. Therefore, we use smaller learning rate to avoid those issues.



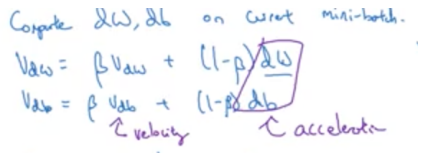
We would like to have slower learning rate in the y direction in our example, but faster learning on X direction. Here is how we do it with momentum.



This smooth out the GD. if averaging over the vertical direction, it tends to get some values close to zero. All the derivatives in horizontal direction are pointing to the right horizontal direction. So, the average in horizontal direction will be still pretty big. Momentum GD with few iterations, eventually ends up taking steps with much smaller oscillation in vertical direction and more directed in horizontal direction toward the minimum. This algorithm takes more direct path, showing in red line in the plot below.



If we have a bowl shape contour, the derivative terms provide acceleration to a ball that we are rolling down the hill. Momentum terms are like velocity. Using momentum causes the algorithm accelerates toward the downhill.



Here is how we implement it. We have 2 hyper parameters.

On iteration t:

Compute dW, db on the current mini-batch

$v_{dW} = \beta v_{dW} + (1-\beta)dW$

$v_{db} = \beta v_{db} + (1-\beta)db$

$W = W - \alpha v_{dW}, \; b = b - \alpha v_{db}$

Hyperparameters: α, β $\beta = 0.9$

The most common value for beta is 0.9, averaging over ten last points. Usually people don't use bias correction, because after few iterations, the moving average is not bias anymore. Sometimes in some literature, we will see they omit 1-beta in the equation, shown below.

$V_{dW} = 0, \; V_{db} = 0$

On iteration t:

Compute dW, db on the curre

$v_{dW} = \beta v_{dW} + (1-\beta)dW$

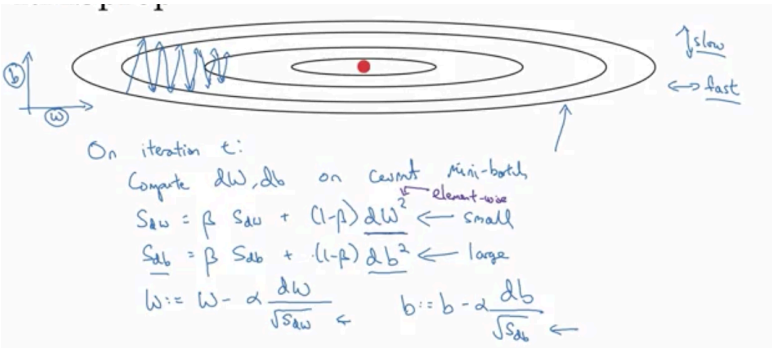
$v_{db} = \beta v_{db} + (1-\beta)db$

$W = W - \alpha v_{dW}, \; b = b - \alpha v_{db}$

The net effect of using this version, is that V of db scaled by a factor of 1 minus beta, or really q over 1 minus beta. Both of the equation works, it only effects the value of learning rate alpha. However, the initial equation on top is preferred.

RMSprop

RMSprop stands for root mean square prop. This method also speeds up the gradient descent. We learned that GD can have a big oscillation during the process, shown in blue color in plot below. Imagine we have parameter b and w in y and x directions. we would like to slow down the learning in b direction and speed up in horizontal direction. RMSprop accomplish this goal.



In equation, dw is relatively small and db is relatively big, which causes to update the values in horizontal direction by larger value and in vertical by smaller value (because we are dividing by the large value). In this case, we can use a larger learning rate without worrying about divergence. In our example, we studied a 2d example. In higher dimensions, we have set of parameters in each direction.

In order to avoid diving by very small value, we may add epsilon like ten to the power minus 8, to avoid making the very huge number.

Adam optimization (adaptive moment estimation)

We can combine RMSprop and momentum GD and get a much better algorithm. The algorithm is known as Adam optimization. It works well in wide range of deep learning architecture.

$V_{dw} = 0, S_{dw} = 0, V_{db} = 0, S_{db} = 0$

On iteration t:

Compute dw, db using current mini-batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$, $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$ ← "momentum" β_1

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$, $S_{db} = \beta_2 S_{db} + (1-\beta_2) db$ ← "RMSprop" β_2

$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t)$, $V_{db}^{corrected} = V_{db} / (1-\beta_1^t)$

$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t)$, $S_{db}^{corrected} = S_{db} / (1-\beta_2^t)$

$$w := w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon} \qquad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

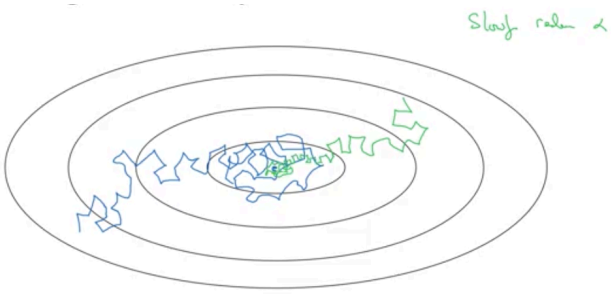
This algorithm has some hyperparamters. Learning rate needs to be tuned. The rest of variables are usually assigned as follow:

Hyperparameters choice:

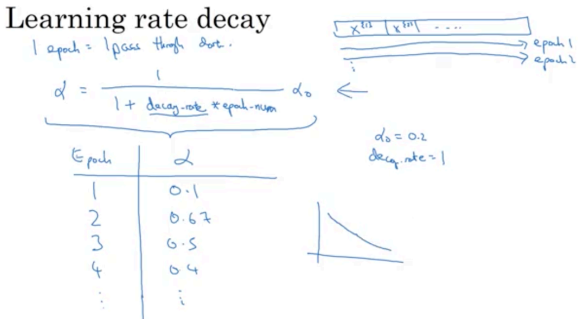
- α : needs to be tune
- → β_1 : 0.9 (dw)
- → β_2 : 0.999 (dw^2)
- ε : 10^{-8}

Learning rate Decay

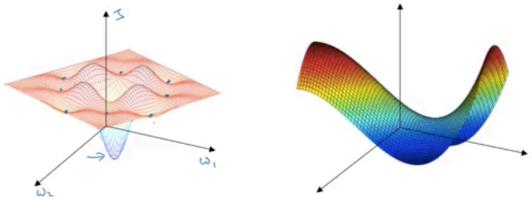
One thing that might help with speed of algorithm is to slowly to reduce the learning rate over time, known as learning rate decay. Using some fixed value of alpha may cause we never reach the minimum and only wonder around it. On the other hand, if we use large leanring rate we still can have a reletively fast learning. Then alpha get slower and smaller by time. Therefore we end up oscilating closer and tighter to the miniumn.



Here is how we implement it. The decay rate is a hyperparamter. Also, there are some other formula you might use for learning rate decay. You may use manually decreasing the learning rate, this mainly works when we have small number of training examples.



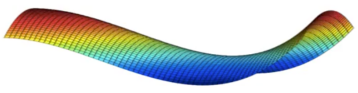
The local optima problem



Most of neural network problem are similar to right graph on plots shown here. Most points have zero gradient in saddle points, rather than local optima.

In high dimensional problem chance of having local minimum or maxima is very low. For example, if we have 20,000-dimensional space, all need to be concave or convex which is pretty rare to happen.

Problem of plateaus



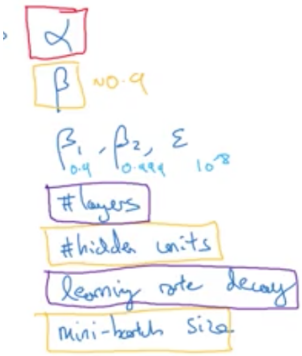
In neural network, the plateaus are problematic since they can slow down the learning. Plateau are regions where derivatives is close to zero for a long time. Therefore it takes a while to get off the plateau.

In summary:

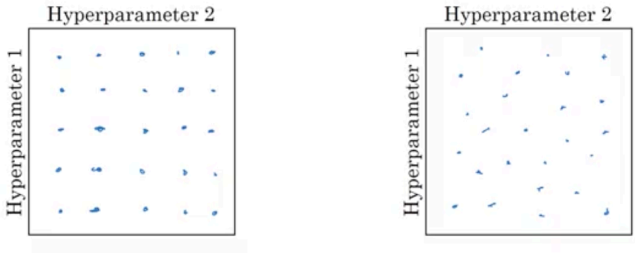
- Unlikely to get stuck in bad local optima when we have lots of training example
- Plateaus are main problem in neural network, using Adam really help in this kind of situation.

Hyperparamter tuning (Don't use grid)

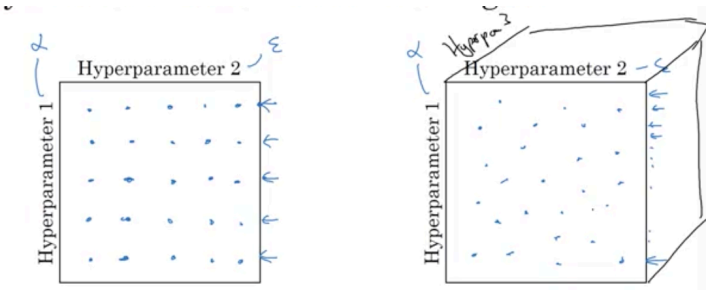
Training neural network may requires setting bunch of hyperparamters ranging from learning rate alpha to the momentum terms beta, or beta 1, beta 2, number of layers, epsilon, learning rate decay and etc. Some of these hyperparamter are more important than the rest, such as: learning rate (alpha), momentum term (beta) with 0.9 default value, mini batch size, and number of hidden layers. The least important parameter to tune are number of layers, learning rate decay. Almost always set values for remaining hyperparamters such as: beta1=0.9, beta2=0.999, epsilon= 10^-8.



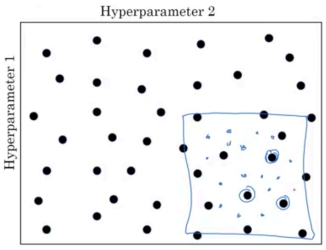
When we have a small number of hyperparamters, we may layout a grid for 2 hyper parameters and try different values in a grid to get the best value for hyperparamters, shown on the left image. However, in deep learning we choose the points at random, then try out the values on this randomly set., shown on the right image below.



In a grid way of tuning, imagine we have learning rate and epsilon. The learning rate is much more effective than the epsilon. The result of grid will be similar for each value of alpha and different value of epsilon. In this case we only tried out 5 different values for alpha. However, in contrast, when we have randomly, we have tried out 25 different values for alpha. When we have 3 or more hyperparamter, we get to use more values of alpha.



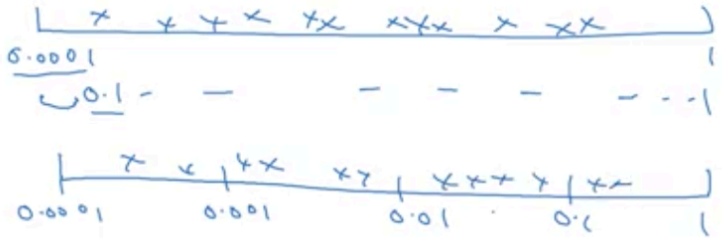
Coarse to fine



It's common practice to use coarse to fine sampling scheme. Imagine we have tries several points and found out some of them tends to work better. In coarse of final scheme, we zoon into smaller region of hyperparamters and then sample more points within that space. Finally pick the value that does the best in our algorithm.

Appropriate scale for hyperparamters

It's important to explore the right scale of values in our randomly searching of right values for hyperparamters. Let's start with number of hidden layers. We might think a good range is values between 50 to 100, which this assumption is pretty reasonable to try values in these range. On the other hand, imagine we think the lowest learning rate of 0.0001 and highest of 1. If we put a randomly variables in between. We put only 10% of effort between 0.1 and 1. It's not really good. Instead we use the log. Here we have more uniform randomly points.

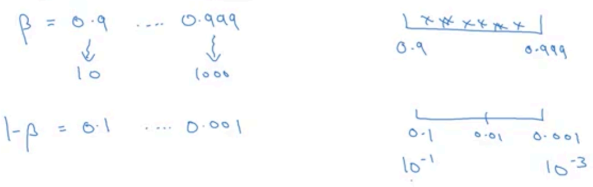


And here is how we implement it (you may generalize the formula below):

$r = -4 * np.random.rand()$ $\leftarrow$ $r \in [-4, 0]$

$\alpha = 10^r$ $\leftarrow$ $10^{-4} \dots 10^0$

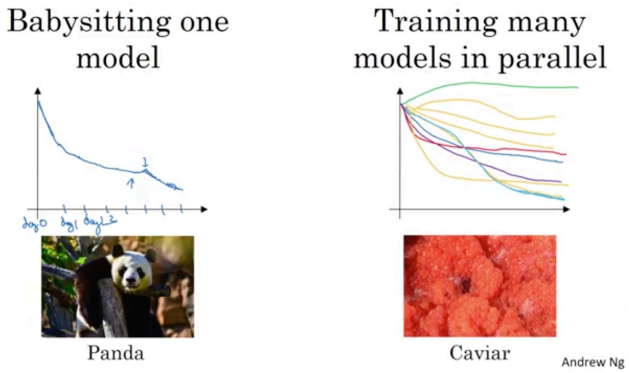
Finally, one other tricky case is sampling the hyperparamter beta (hyperparamters for exponentially weighted average). Imagine we have a small range of value for beta in mind from 0.9 to 0.999. We remember using 0.9 is like averaging over the last 10 points, whereas 0.999 is like averaging over the last 1,000 values. It doesn't make sense to sample on the linear scale. Instead we use 1-beta. Then take a log and uniformly randomly pick values between -3 and -1.



For small range of values, where hyperparamter is very sensitive to value it has a huge impact over the algorithm and we need to make sure to use log instead of linearly randomly points.

Hyperparamter tuning (Caviar vs Pandas)

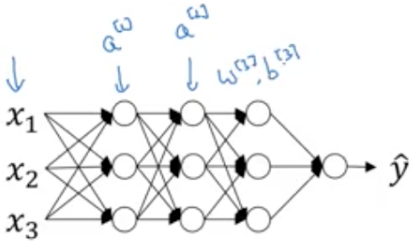
People go out searching for hyper parameters in main two major classes: one is babysit one model. We may use it when we have lots of data and not many computational resources, such as CPU or GPU. Where you can only train small number of models at a time. For example, on day 0, you picked an initial value and look at the cost function on a next day. We see that it's decreasing. Then you might increase the value of momentum abit more or decrease the learning variables. The second approach is to train many models in parallel. Each model gives different cost function. Finally pick the one works best.



Depends on your computational resources, you may pick caviar or panda approaches. Panda has few babies at a time and babysit each of them, but caviar lays bunch of eggs and hope some of them survive.

Batch normalization

Batch normalization makes our hyperparamters search much easier, makes NN much more robust, and enable us to make training very deep network much easier. We know during logistic regression, normalization helps to optimize the model. In neural network, we have bunch of hidden layers, input and output layers. Imagine we would like to train parameters w3 and b3 in network below:



It would be nice if we can normalize the mean and the variance of a2 to make the training of w3 and b3 easier. That's what batch normalization does. There are some debates whether to do normalization after or before the activation functions. In practice, normalizing before activation function (z) is what we talk about here.

Here is the implementation of algorithm: (gamma and beta are learnable parameters. These get updated as we would update the weight in neural network)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

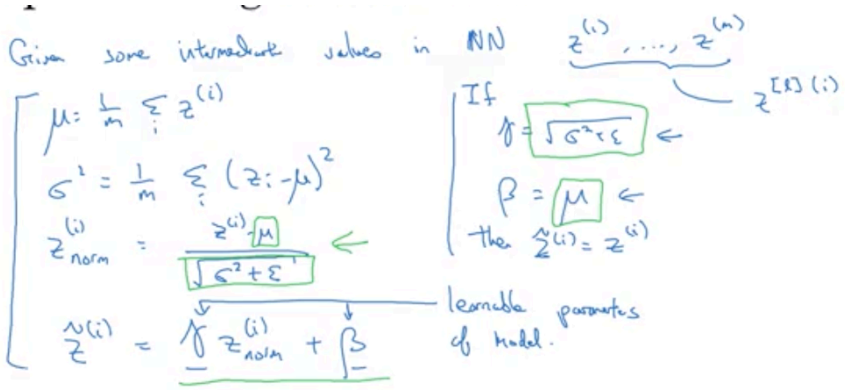
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

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

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

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

Notice that the effect of gamma and beta is that it allows you to set the mean of z tilde to be whatever you want it to be. Basically, by setting the gamma and beta as follow, we get z as identity function.

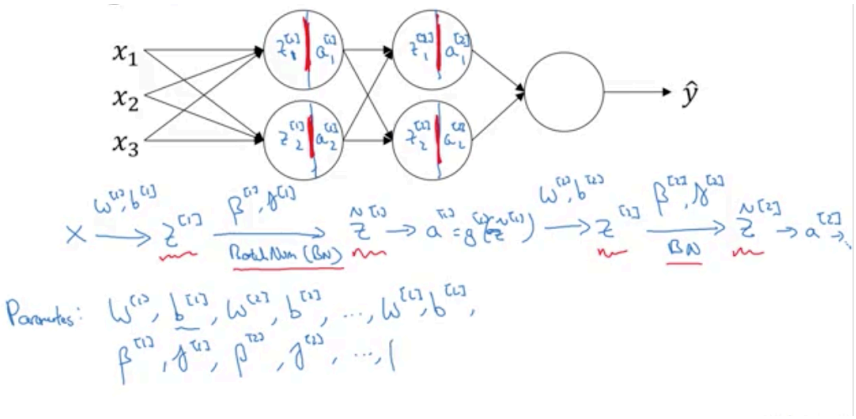


In summary, the scale and shift allow each NN layers have different variance and mean. The batch normalization allows normalization in input layer as well as hidden layers in NN. The different between training input layer and hidden units is you may not want the hidden layers forced to have mean 0 and variance 1. For example, if you have a sigmoid activation function, you don't want your values to always be clustered here. You might want them to have a larger variance or have a mean that's different than 0, in order to better take advantage of the nonlinearity of the sigmoid function rather than have all your values be in just this linear regime.

Batch normalization makes sure the hidden layers have standardized mean and variance, where they are controlled by parameters gamma and beta.

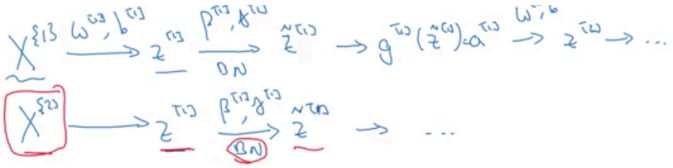
Fit batch norm into NN

Here we discuss how to apply normalization of batch in deep neural network. Imagine you have NN shown below. We may view each unit as computing two things, one is computing z and then applies the activation function to compute a. each circle is computing two things.

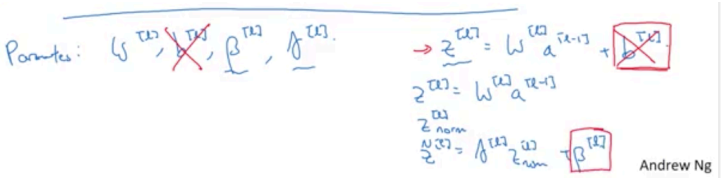


The normalization happens between computing z and a. We normalize the z using batch normalization, then use that z to calculate a. Note that this beta is not related to Adam optimization.

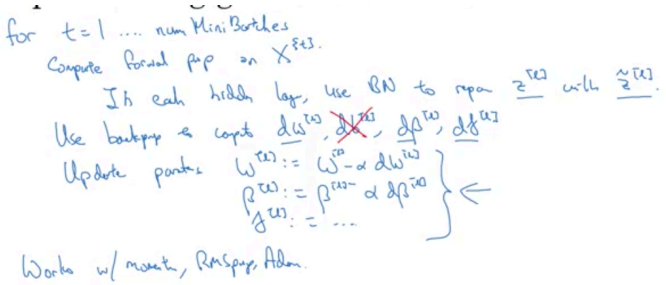
Batch norm is usually applied with mini batches of the training set. You do normalization on each mini batch independent from the other one. So the beta and gamma value on one mini batch might be different than the other ones.



Value of b is not really important, since during normalization it get substract out. If using batch norm, you can omit value of b.



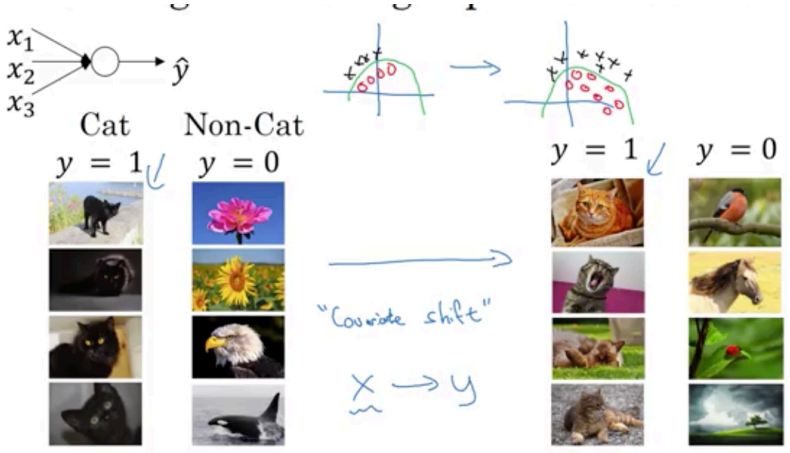
Here is how you implement gradient descent on mini batch using normalization.



Why batch norm?

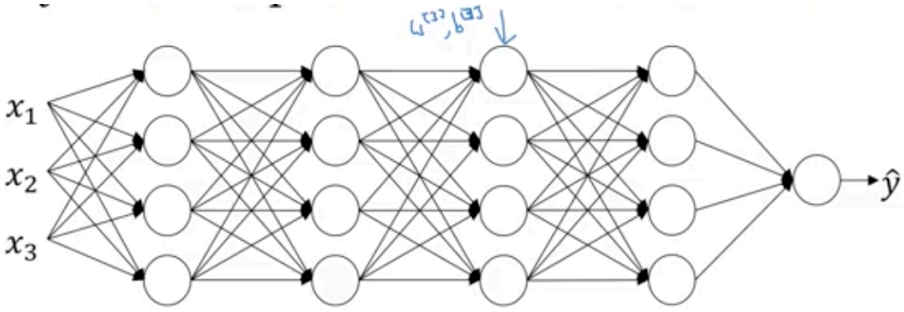
You may be wondering why batch norm increases the speed of training a lot. Batch normalization does the similar thing to the regression when normalizing the features, by applying it to the hidden layers. The second reason is that it makes weight deeper into the network, more robust to changes in weights in earlier layers of NN.

Imagine you use logistic regression (very shallow NN) to train in order to detect cats that are black, if you use the that model to detect the images of the cats that are colorful, the model won't detect them very well. If the cost function for negative and positive cost function looks like the one on the left, for the new images.

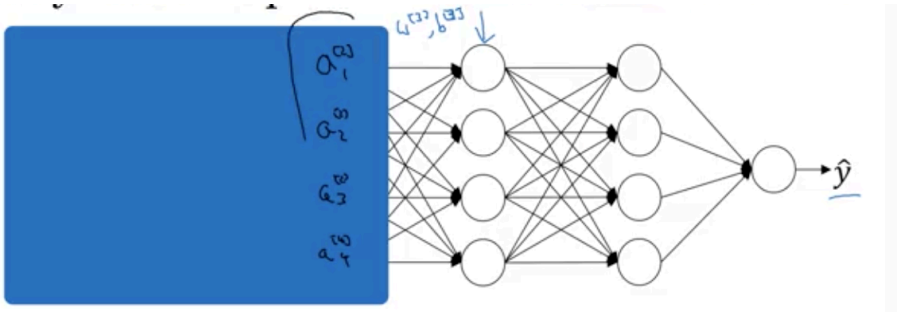


The boundary might change in our example. The data distribution changes in new dataset which is called covariate shift. When we have a map from x to y, and distribution of X changes, then we might need to retrain our learning algorithm, even if the mapping ground function remains unchanged. For example, in our example we have cat or not cat pictures. We need to retrain the algorithm to make it more acute while the ground true function shifts.

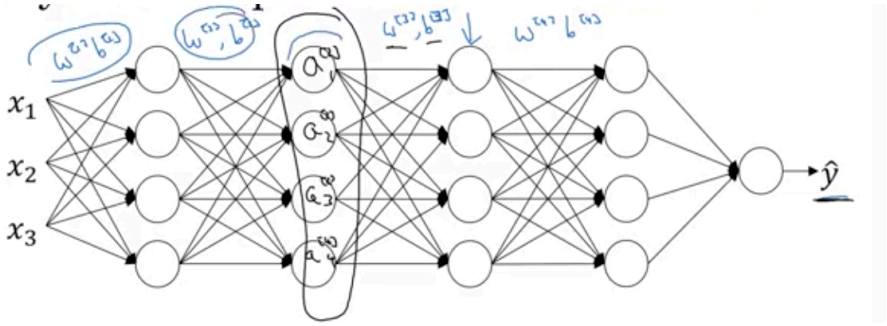
How the covariate shift applies to NN? Consider a deep NN shown as below. We would like to look at the learning process from the perspective of the certain layer, for example layer 3. This network has learned the parameters w3 and b3. This layer get some set of values from previous layer and make some changes to them in order to make the output value y close to the ground true value of y.



Layer 3 gets some value from previous layer. Let's call them a21, a22, a23, and a24. These values might be features as well (like x1, x2, x3, x4). The 3rd layers job is to take those values and find a way to map them to the y.



If we uncover the values on the left of network, it shows that layer 3 is also adapting parameters of w1 and b1, and w2, b2. As these parameters change, values of a2 will also change.



From perspective of layer 3, the hidden values are changing all the time and it is suffering from covariate shift. Batch norm reduces the amount that distribution of these hidden layer unit shifts around. The distribution of hidden layer values (normalize z). During the batch normalization, the values of z21 and z22 change when NN updates the parameters in earlier layers. Batch norm ensures that no matter how the earlier layers changes, the mean and variance of z21 and z22 remains same. In other word, batch normalization limits the amount to which updating the parameters in earlier layers can affect the distribution of values that the 3rd layer sees and therefor has to learn on. Batch norm reduces the problem of the input values changing by making them more stable. Having more stable inputs results in firmer ground for next layers to stand on (input distribution changes abit). In this case, there will be little changes in the amount can vary in each layer. It weakens the coupling between layers and each layer of network learns by itself with little dependency on other layers. As a result, it speeds up the learning in the whole process. Since the mean and variance of each layer remains same, the new layer will learn faster.

Batch normalization also has slight regularization effect on NN.

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch.
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect.

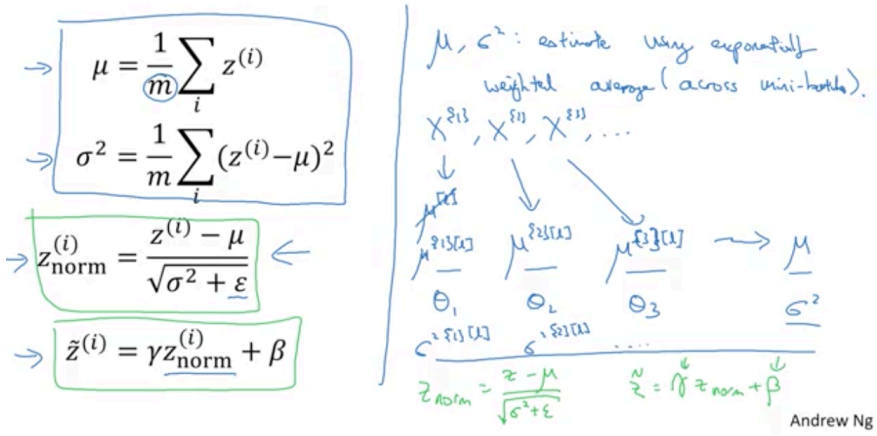
Since batch normalization adds noise to hidden layers, it has a slight regularization effect. batch norm has multiples of noise because of scaling by the standard deviation as well as additive noise because its subtracting by mean. By adding noise to the hidden layers, it is forcing the downstream units not to rely too much on any of the hidden layers. We may use batch norm with dropout in order to make more powerful regularization effect.

By using larger mini batch size (like 512 instead of 64), the noise is reduced further causing less regularization effect. In general, it's not a good practice to use batch norm as regularizer (since it's not the intent). Use batch norm to speed up the learning but be aware of regularization as an unattended side effect.

Batch norm process each mini batch separately. During testing when we want to make prediction and evaluate the NN, we might not have mini batch examples. We might process one single example at the time. Therefore, during the test, we need to make some changes to make sure the prediction makes sense.

Batch norm at test time

In this session, we talk about how we can modify the model we built using batch norm for the test dataset (in case, we do not have mini batch testing set). Taking mean and variance of one example doesn't really makes sense. The right side of image below shows how to modify the batch norm during training (on the left).



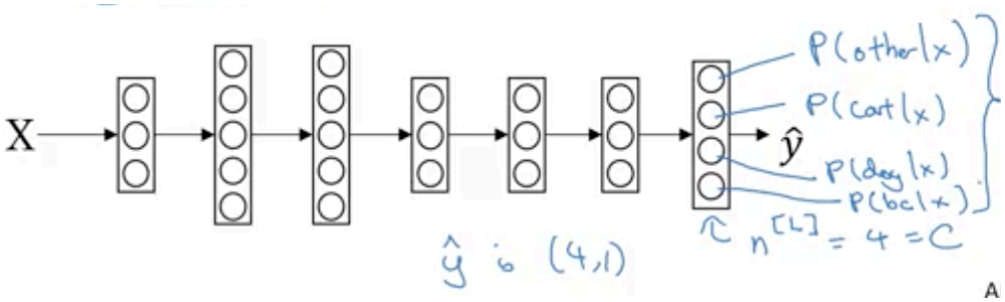
In test set, we may find the mu and sigma through all training set and use that value for test set. Better way is finding mu and sigma using exponentially weighted average. We keep track of mu and sigma during training and use running average (exponentially weighted average), to get rough estimate of mu and sigma. Then use these values during test time to scale value of z. This process is pretty robust.

Multiclass classification (Softmax regression)

So far, we talked about mainly binary classification in our examples. There is a generalization of logistic regression called Softmax regression. Imagine we would like to recognize the pictures of cats, dogs, and chicks. If image is neither of those 3 classes, we show them as 0.

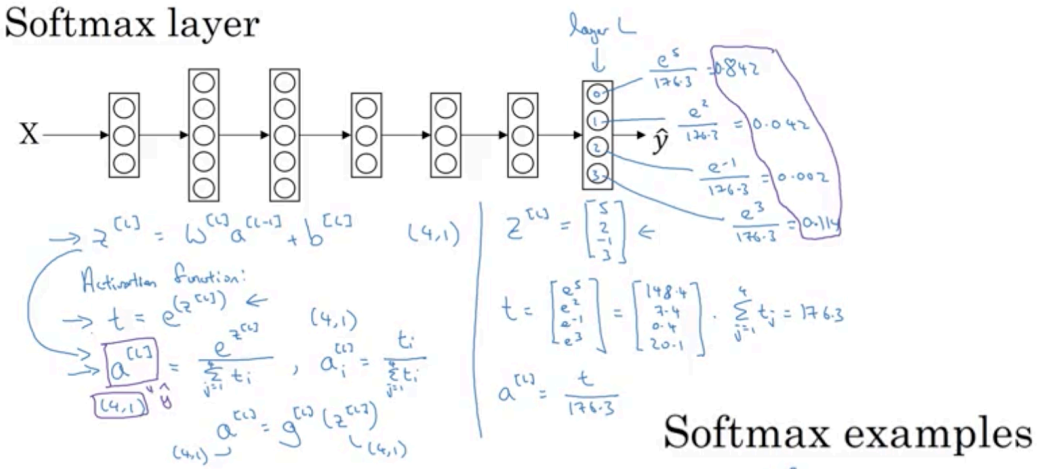
3 1 2 0 3 2 0 1

C is denoted at number of classes. In our example it's 4 (0 to3). Therefore, output layer should have 4 layers. It will tell us what is the probability of each of these four classes. Sum of these probability should be 1. We use Softmax layer in order to generate these outputs.

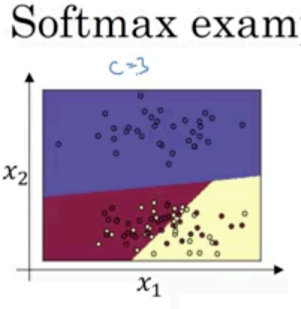


A

For Softmax, we add temporary variable t and calculate elementwise of exponential function.



Previously, we had a real number as input and output, but in Softmax, it gets vector in input and output. Decision boundary between classes will be linear, when there are no hidden layers.



Softmax training

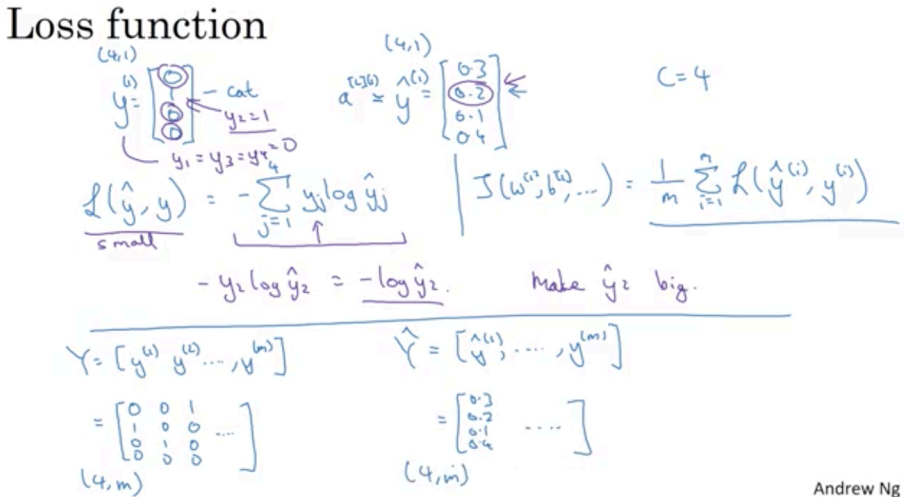
We may use Softmax for training NN. In example below, you can see the biggest element of z is 5 and it has the highest probability. Softmax is as opposed to the hardmax, where element of z would be [1 0 0 0]. The biggest element gets 1 and rest gets value of zero in hardmax. In Softmax, it does more gentle mapping.

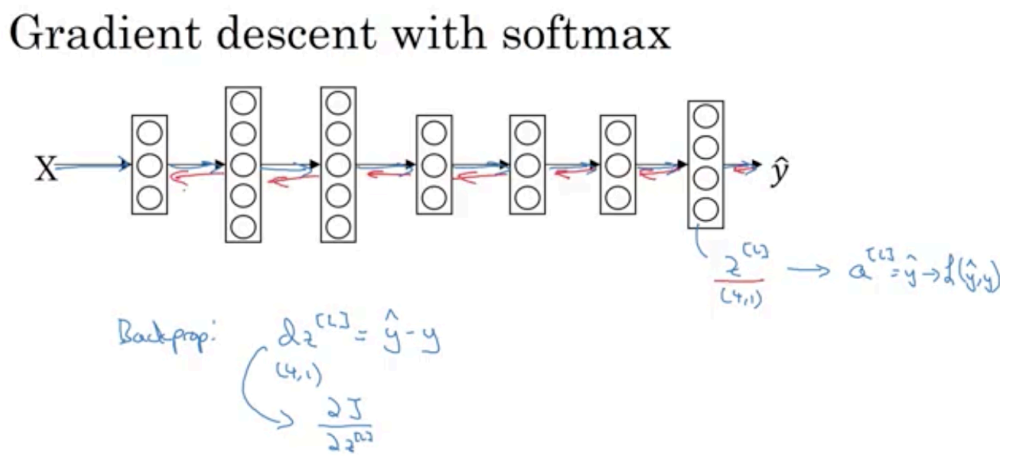
$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \qquad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix}$$

$$a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5+e^2+e^{-1}+e^3) \\ e^2/(e^5+e^2+e^{-1}+e^3) \\ e^{-1}/(e^5+e^2+e^{-1}+e^3) \\ e^3/(e^5+e^2+e^{-1}+e^3) \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

Softmax function generalizes logistic regression to C classes. if C=2, NN is basically the logistic regression. In this case, you only need to calculate it for one class and the other class will be one minus that value.

In Softmax, we use loss function. In some example, maybe the result is not very satisfying. In this case, we use loss function. what this loss function does is it looks at whatever is the ground true class in your training set, and it tries to make the corresponding probability of that class as high as possible. For calculating cost function, we use gradient descent to minimize the value of J.





Using TensorFlow, will take care of backpropagation for you and you would not need to take care of that from scratch.

Tensorflow

When working on large sets, it's good idea to use the libraries and function available to use since that's less time consuming and well implemented. There are many deep learning frameworks to use such as: Caffe, CNTK, DL4J, Keras, TensorFlow, Torch, Theano and etc. It's important to choose the frameworks that is easy to program, high running speed, and truly open source with good governance (they don't close it later or make you to pay).

Tensor flow is one of the great open source framework.