SECOND COURSE: IMPROVING DEEP NEURAL NETWORKS: HYPERPARAMETER TUNING, REGULARIZATION AND OPTIMIZATION **2**
Setting up your Machine Learning Application 2
Train / Dev / Test sets 2
Bias and Variance 3
Basic Recipe for Machine Learning 4
Regularizing your neural network 5
Regularization 5
Dropout Regularization 6
Other regularization methods 7
Setting up your optimization problem 8
Normalizing inputs 8
Vanishing / Exploding gradients 12
Weight Initialization for Deep Networks 13
Numerical approximation of gradients 14
Gradient Checking 15
Optimization Algorithms 16
Mini-batch gradient descent 16
Exponentially weighted averages 18
Bias correction in exponentially weighted averages 19
Gradient descent with momentum 20
RMSprop (Root Mean Square Prop) 21
Adam optimization algorithm 22
Learning Rate Decay 23
The problem of local optima 24
Hyperparameter tuning 25
Tuning Process 25
Hyperparameters tuning in practice: Pandas vs. Caviar 27
Batch Normalization 28
Normalizing activations in a network 28
Fitting Batch Norm into a neural network 29
Softmax Regression 30
Deep Learning Frameworks 34
TensorFlow 34

SECOND COURSE: IMPROVING DEEP NEURAL NETWORKS: HYPERPARAMETER TUNING, REGULARIZATION AND OPTIMIZATION

Setting up your Machine Learning Application

Train / Dev / Test sets

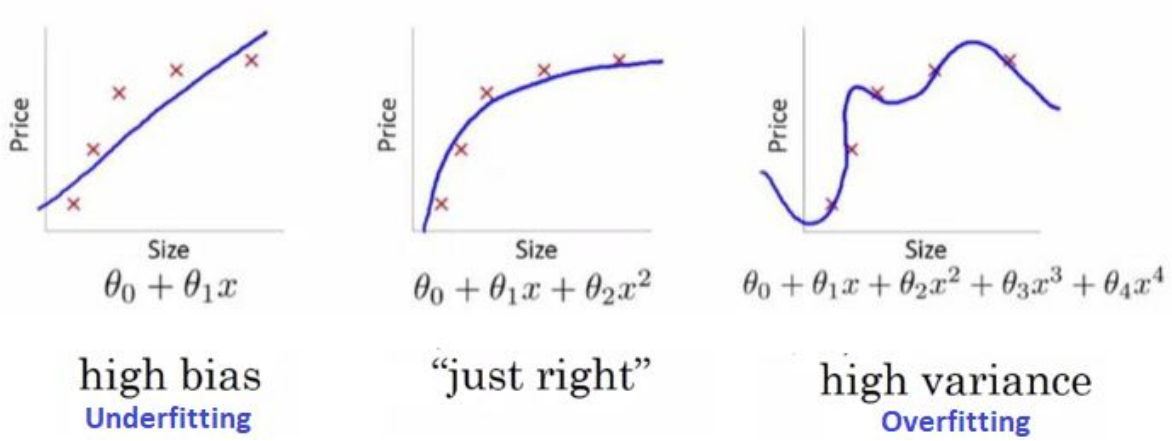
The available data is usually divided into 3 different parts: Training set, Development set and Testing set. The training set, as expected, is used to train the neural network. The development set (hold-out and cross-validation set | dev set) is used to optimize the model (trying different models and changing hyperparameters), the dev set can be a bit biased, that's why you need to separate the data from training and testing sets. Finally, the testing set is used to determine the performance of the model.
If the number of examples in the dataset is small (less than 10K examples), you might divide your examples in the following form:
Train set = 60%
Dev set = 20%
Test set = 20%
However, we are in the big data era, so it's common to have datasets with millions of examples. In this case, you might change the percentage to:
Train set = 98%
Dev set = 1%
Test set = 1%
It happens because the dev set and the test set will probably not improve the performances with a high value of examples. So you might pass these examples to the training set.

One of the most common problems that might happen when modeling a neural network, is to take the dev set and test set from different distributions. For example: you are modeling a neural network to recognize if certain image is either a cat or not - for the dev set you take cat pictures from google images, and for the test set you take cat pictures from users on a social media (in this case, the dev set pictures will probably have higher resolution/quality compared to the social media ones). The different distribution might lead to a problem in the performance, because you are changing your model and hyperparameters to work with a distribution, and your tests are running on a different distribution. So make sure the dev and test sets come from the same distribution.

One more detail to have in mind is: not having a test set might be okay (just have a dev set), when you don't need an unbiased estimate of the performance of the algorithm. When people have only train and dev sets, they end up calling the dev set as test set, but it actually is a hold-out cross validation dev set.

Bias and Variance

The concept about bias and variance is very important in machine and deep learning. When we are trying to adjust a certain model, these two values can help us to identify if our model is good or not. When the model works very well with the training data, but this performance does not persist with generical new data, in this case we have overfitting (high variance and low bias). When the model works poorly with the training data and with new generic data, we have underfitting (high bias and low variance). A good model is a model with balance in the variance and bias values.



Let's see an example of these concepts applied to the cat classification model. For this example, we will assume humans have approximately 0% error in classifying a cat picture.

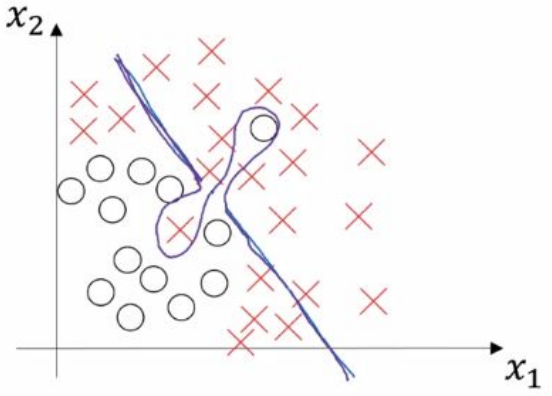
Train set error	1%	15%	15%	0.5%
Dev set error	11%	16%	30%	1%
	high variance (overfitting)	high bias (underfitting)	high bias and variance (worst scenario)	low bias and low variance (good model)

Remember that these analyses were made considering a specific model with optimal error close to 0% (optimal error (or base error) in this case is the error humans get when classifying cats).

In conclusion, to analyse the bias you have to compare the optimal error to the train set error. If these values differ much one from another, you have high bias (and

probably underfitting). And finally, if your train set and dev set errors are very different, you have high variance (and probably overfitting).

One plot example of high bias and high variance can be seen below:



As you can see, the classifier has high bias because it is not fitting correctly the values under the plot (lots of X are selected -> high error) and high variance because there is too much flexibility in the middle of the curve to fit those two mislabel examples in the middle.

Basic Recipe for Machine Learning

After training an initial model, you have to ask the following questions:

1 - “Does my algorithm have a high bias?”
If it has a high bias and it’s not fitting the training set that well, you might try picking a bigger network (with more hidden layers or more hidden units), run the training for a longer period of time, try some advanced optimization algorithms or even try different NN architectures.

2 - “Does my algorithm have high variance?”
If it has a high variance, one of the best solutions is to get more data when possible. If it’s not possible to get more data, you can try regularization or even in the worst scenarios try different NN architectures.

If both bias and variance have acceptable values, your model is just right!

In the beginning era of machine learning, there was a concept called Bias Variance Tradeoff. This concept existed because it was rarely possible to reduce bias and variance at the same time. If you reduced the bias, the variance would get bigger, and if you reduce the variance the bias would get bigger. However, in the big data era, this problem doesn’t affect the applications with the same impact. Today we are able to reduce both variance and bias without compromising each other that much.

Regularizing your neural network

Regularization

If you have a high variance (overfitting) problem, one of the first things you should try is probably regularization. The other way to deal with a high variance is to get more data, but this solution might not always be available. So let's see how regularization works.

$$J(w,\ b)\ =\ \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)},\ y^{(i)})\ \ +\frac{\lambda}{2m}||w||_2^2$$

$$||w||_2^2\ =\sum_{j=1}^{n_x} w_j^2\ =\ w^T w\ \ (1)$$

The equation (1) is called L2 regularization. The parameter b is usually not included in the regularization, because b is just one parameter over a very number of parameters and regularizing the parameters w already make a big difference. There is also the L1 regularization:

$$\frac{\lambda}{2m}||w||_1\ =\frac{\lambda}{2m}\sum_{i=1}^{n_x}|w|$$

In both cases, λ is called the regularization parameter. This parameter is usually set using the dev set or hold-out cross validation (λ is also a hyperparameter).

If you use L1 regularization, then w will end up being sparse, that means the w vector will have a lot of zeros in it. Although it seems to be useful to compress the model, it actually doesn't help that much, the L2 regularization is much more used instead.

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

where $||w||_F^2$ is the Frobenius norm of a matrix. It can be seen in the equation below:

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

The backprop formula of dw will also have a new addition:

$$dw^{[l]}\ =\ (equals\ to\ backprop\ \ldots)\ +\ \frac{\lambda}{m}w^{[l]}$$

$$w^{[l]}\ :=\ w^{[l]}\ -\ \alpha dw^{[l]}$$

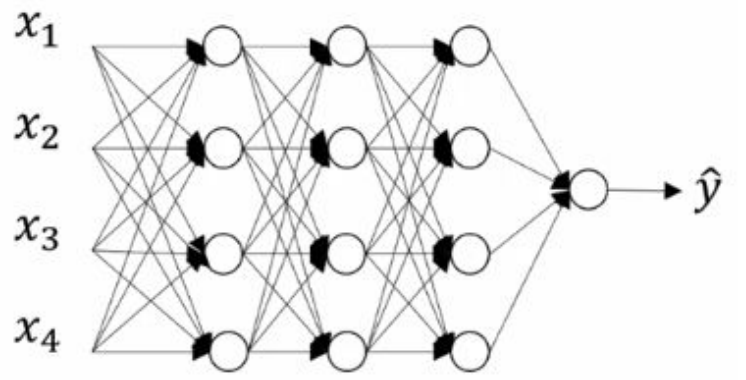
The L2 regularization is commonly called weight decay. It receives this name because it adds a decaying factor when the parameter w is updated.

$$w^{[l]}\ :=\ w^{[l]}\ -\ \alpha\frac{\lambda}{m}w^{[l]}\ \ -\alpha(equals\ to\ backprop\ \ldots)$$

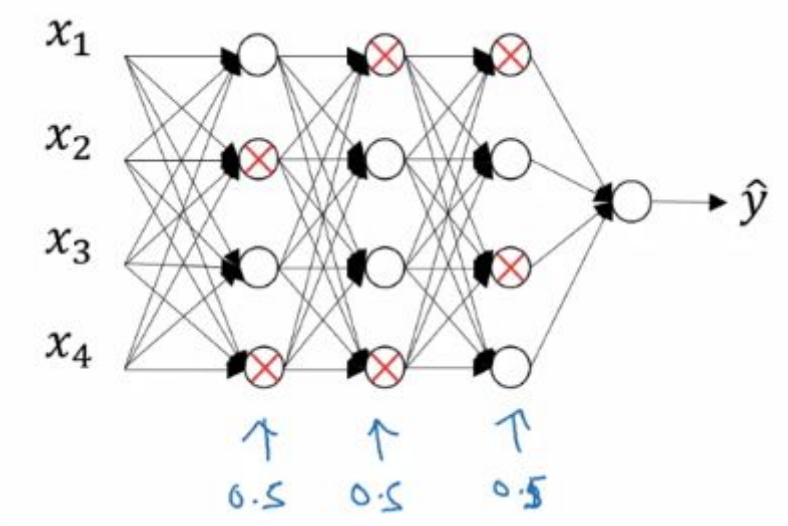
$$w^{[l]} := w^{[l]}(1 - \alpha\frac{\lambda}{m}) - \alpha(equals\ to\ backprop\ ...)$$

Dropout Regularization

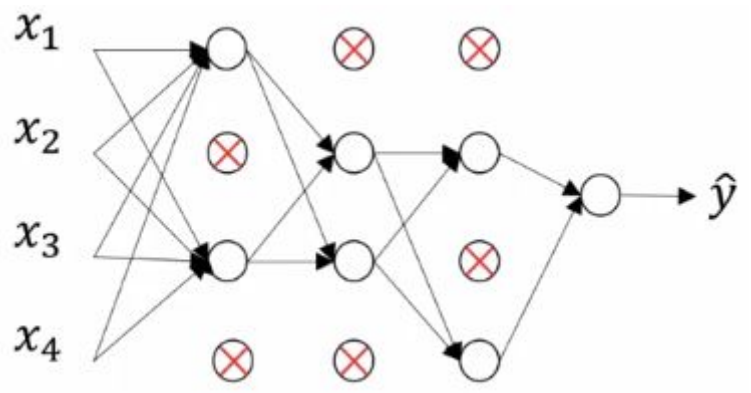
In addition to the L2 regularization, another powerful regularization technique is called dropout. With dropout regularization, we go through each of the layers of the neural network and set some probability of eliminating a node. For each node of the neural network, toss a coin. There will be 50% of keeping a node and 50% of removing a node. Then remove all the selected nodes with all the incoming and outgoing connections from these nodes. Then do the backward propagation. Repeat this process with every training example, and this process is going to reduce the variance. You can see this process in the images below:



(Initial neural network representation)



(Select the nodes to remove)



(Remove the nodes and their connections, then perform all the needed computations)

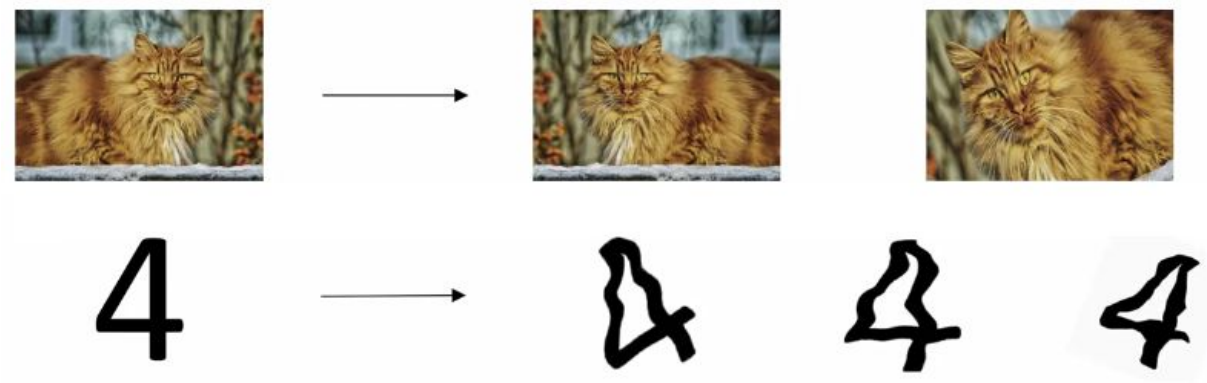
Implementing Dropout Regularization (using “Inverted dropout” technique)
Let’s illustrate this using the layer $l = 3$.

```
keep_prob = 0.8  #there is 0.2 chance of removing a hidden unit
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)        #this is a element wise multiplication
a3 /= keep_prob
```

Different layers can have different probabilities of keeping the nodes. A hidden layer with a large number of hidden units can have a keep_prop equals to 0.5, while other hidden layers with less hidden units can have a keep_prop equals to 0.8 and even 1 (keeping all the units)… . This means the value of keep_prop can change through the neural network. The negative side of this alternative is to add more hyperparameters to track and store.
The dropout technique is usually used in computer vision, because you almost always don’t have enough data and is probably always overfitting. One last point in this discussion, when you are using dropout, your cost function J is not well defined anymore, because the cost in each layer and iteration changes every time, so you will not be able to debug your model analysing the plot of the cost function anymore.

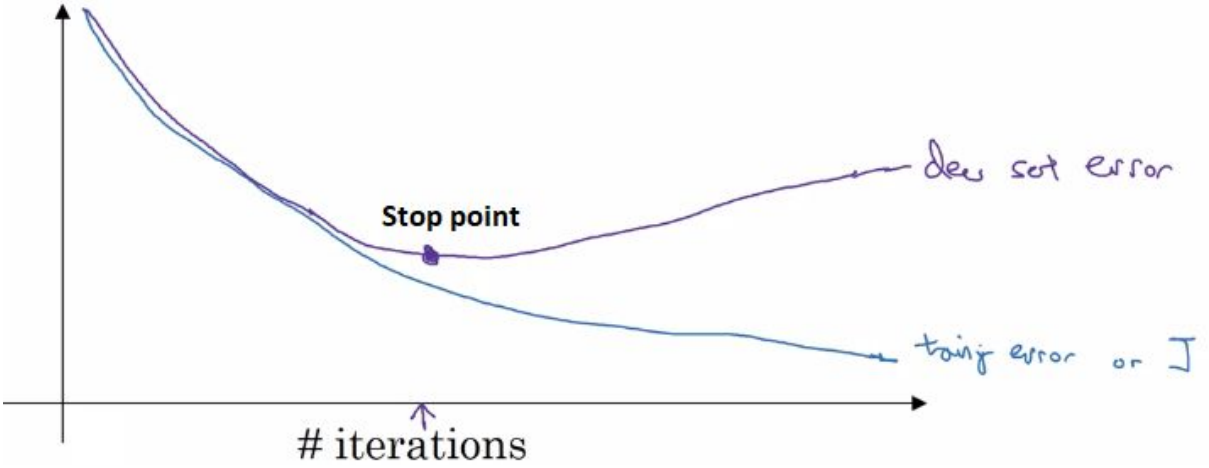
Other regularization methods

One of the regularization methods you can try when dealing with images, is the data augmentation. It consists in making transformations in the images to make the dataset bigger. For example:



The first example above is the cat recognizer model. To get more data, you can flip the images horizontally, zoom in the images, and make different transformations to get more data and reduce overfitting.
The second example is to recognize the number four. In this case, the number four has been rotated and distorted in different ways to make the dataset larger, probably reducing overfitting.

There is one more technique that is often used called Early Stopping. As you run gradient descent, you're going to plot your training error | cost function J. Additionally, you are going to plot the dev set error. What you will realize is that the dev set error will usually go down for a while, and then it will increase again. What the Early stopping technique does, is to stop training the neural network halfway.

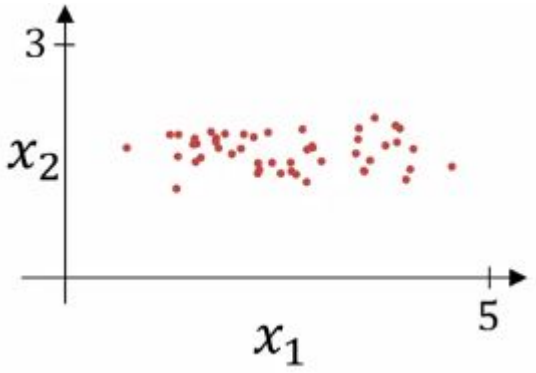


One of the problems of this method is when you stop the learning in the halfway, you end up not optimizing your cost function that much, and it makes the performance of the model less efficient.

Setting up your optimization problem

Normalizing inputs

Let's normalize an input set with two features: x1 and x2.

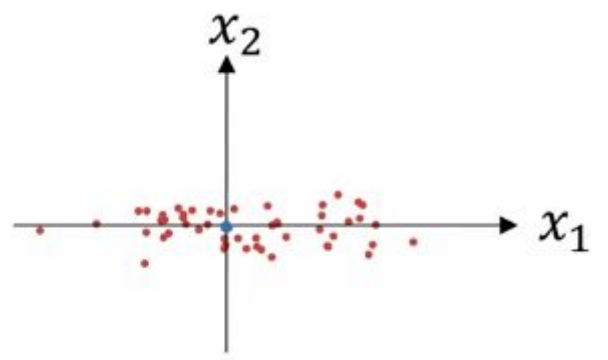


Normalizing your inputs corresponds to two steps:
1) The first step is to subtract out the mean from the training examples:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$x := x - \mu$$

This means you just move the training set until it has zero mean. The plot will be like this:



2) The second step is to normalize the variance:

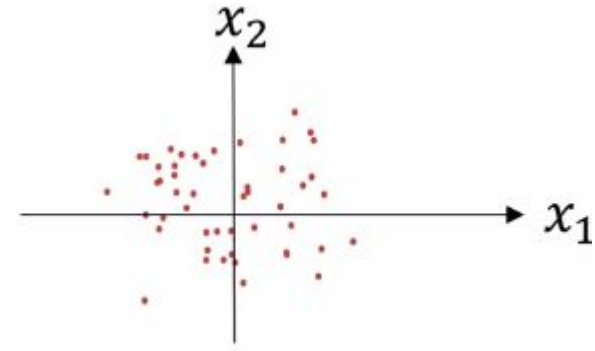
$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} ** 2 \ (this\ is\ a\ element-wise\ square\ multiplication)$$

$$x /= \sigma$$

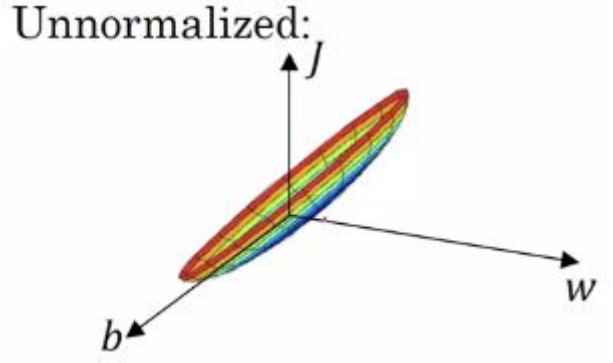
So the formula for normalization is:

$$\frac{x-\mu}{\sigma}$$

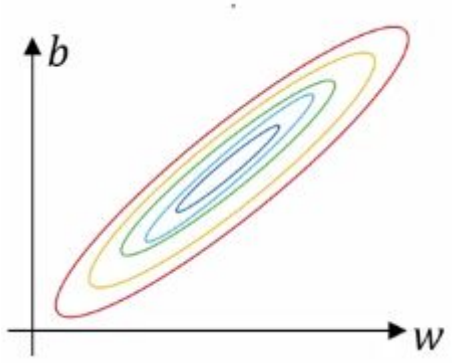
The plot will be like this:



Normalizing the data is a very important step, primarily in the cases where the features are in very different scales (x1 ranges from 1 to 10000 and x2 ranges from 0 to 1, for example). This difference is going to make the parameters w1 and w2 end up taking very different values.

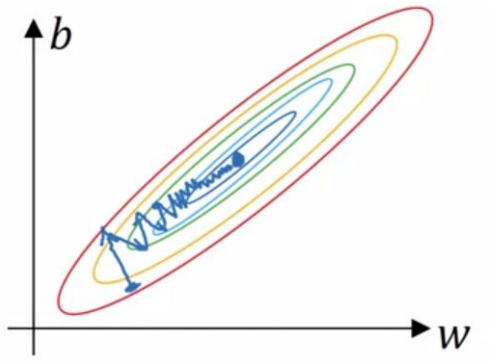


(Example of cost function plot with unnormalized data)



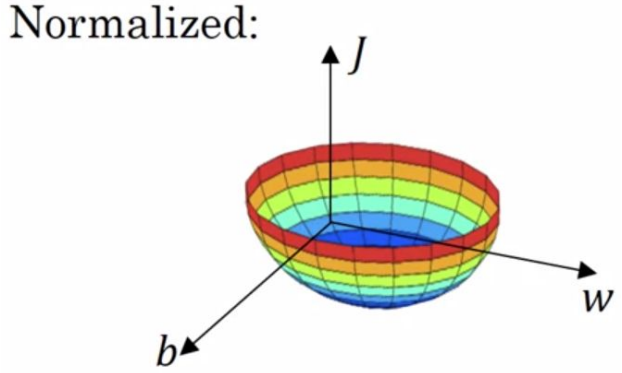
(level set of the cost function)

If you are running gradient descent with unnormalized data, you might have to use a small learning rate, due to the shape of the curve - it might take more time to reach the optimal value of the cost function. The gradient descent steps can be seen below:



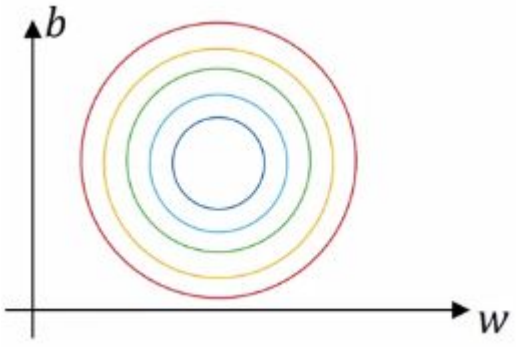
(Steps of the gradient descent through the function)

However, if you normalize your data, the shape of the curve will be like this:



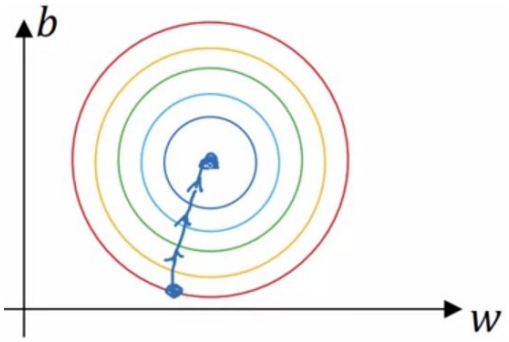
(Example of cost function plot with unnormalized data)

And its level set:



(level set of the normalized cost function)

Using normalization, the gradient descent can take larger steps into the optimal value, because the function's behavior is more predictable. These steps can be seen in the image below:



So if your features come in very different scales, a very important step to take is to normalize the data. If your features come in a close scale, you can jump this step, but doing it will make your algorithm work better, so don't be afraid to use this technique!

Vanishing / Exploding gradients

One of the problems of training neural networks, especially very deep neural networks, is data vanishing or exploding gradients. What that means, is that when you're training a very deep network your derivatives or your slopes can sometimes get either very big, or very small, and this makes training difficult.
Depending on the value you choose to start the weight variables, your gradient can explode/vanish - each layer of the neural network, the gradient becomes much bigger/smaller, and it grows/decays exponentially. To explain this problem, let's consider a neural network with two features (x1 and x2), all the parameters b equal to zero, lots of hidden layers and a linear activation function $g(z) = z$. Let's compute the prediction value $\hat{y}$.

$$g(z) = z \quad ; \quad b^{[l]} = 0$$
$$z^{[1]} = w^{[1]}x$$
$$a^{[1]} = g(z^{[1]}) = z^{[1]}$$
$$z^{[2]} = w^{[2]}a^{[1]}$$
$$a^{[2]} = g(z^{[2]}) = z^{[2]} = w^{[2]}a^{[1]} = w^{[2]}w^{[1]}x$$
$$\dots$$
$$\hat{y} = w^{[L]}w^{[L-1]}w^{[L-2]}w^{[L-3]}\dots w^{[3]}w^{[2]}w^{[1]}x$$

If the weights are bigger than 1, the gradient can explode:

$$W^{[2]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

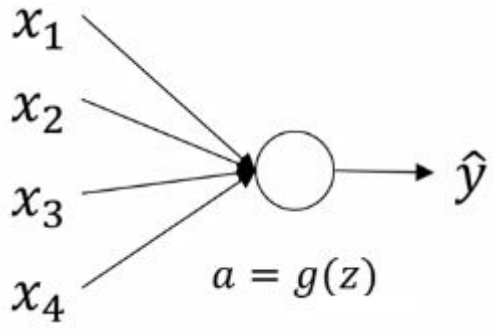
Imagine this matrix multiplied $L-1$ times. This number will be very large, exploding.

Now if you imagine a small weight, the matrix will be multiplied and the gradient will eventually “vanish”, becoming very small.

This problem for a long period of time was a huge barrier in training deep neural networks. It turns out that there is a partial solution that doesn’t completely solve this problem but helps a lot, that is choosing the initial values of the weights carefully.

Weight Initialization for Deep Networks

A partial solution to the vanishing/exploding gradient is to carefully choose the random weights initialization values. Let’s see how to do it with a neural network with 1 neuron. For this example, let’s consider the parameter b equals to zero.



In this case:

$$z = w_1x_1 + w_2x_2 + \dots w_nx_n$$

The larger the number n is, the smaller we want w_i to be. One reasonable thing to do, is setting the variance equals to:

$$Var(w_i) = \frac{1}{n}$$

Where n is the number of features that go into the neuron. In practice what we can do is:

$$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$$

If you are using a ReLU activation function, it’s better to use a 2/n value for the variance, instead of 1/n. The weights will be in this case:

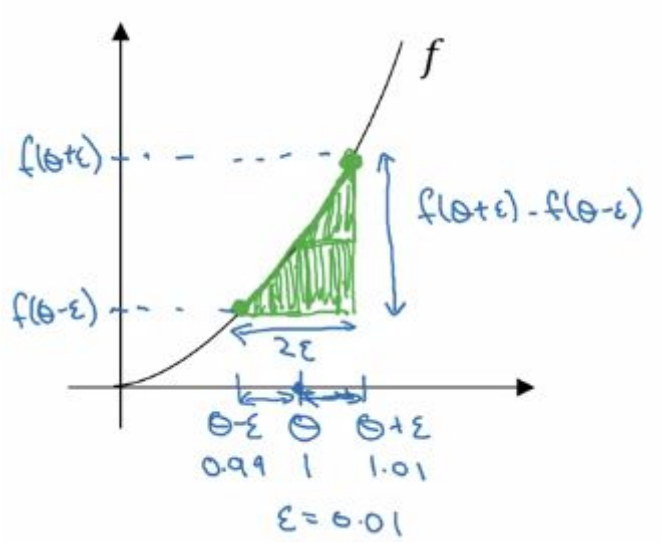
$$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$$

If you are using other activation functions, you might change the variance as well:

For the tanh, you change the term $np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$ to $\sqrt{\frac{1}{n^{[l-1]}}}$. This is called the Xavier initialization. You can also change the term to or $\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$, that is the He initialization.

Numerical approximation of gradients

When you're implementing backpropagation, you'll find out that there's a test called Gradient Checking that can help you make sure your implementation of back prop is correct. In order to build up the gradient checking, let's first talk about how to numerically approximate computations of gradients. Let's assume we are dealing with a cubic function -> $f(\theta) = \theta^3$. Let's also assume $\varepsilon = 0.01$ to make the calculations of the derivative. You can see the plot in the image below:



Let's compute an approximation to the derivative of the function using the height divided by the width:

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

$$\frac{(1.01)^3-(0.99)^3}{2(0.01)} = 3.0001$$

If we use the real value for the derivative:

$$g(\theta) = 3\theta^2 = 3$$

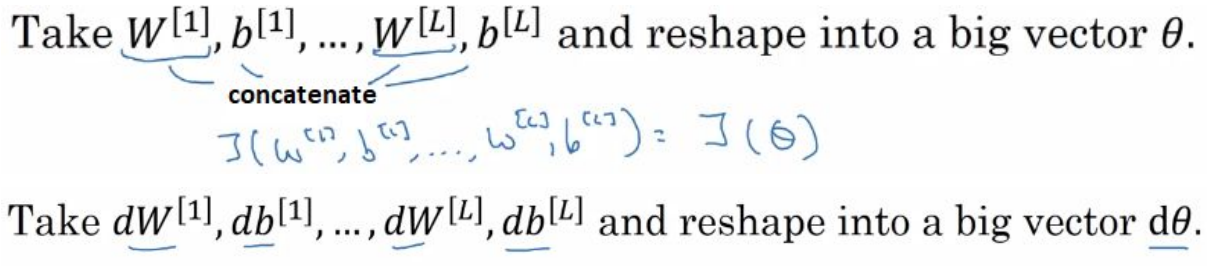
The approximate error is 0.0001. So this method is a very good method for approximating the value of the derivative.

Obs: we use 2ε instead of just ε (the original definition of the derivative) because it is a better approximation for the derivative value. If we used just one ε, the error would be 0.01, much bigger than 0.0001.

Let's see how to implement gradient checking!

Gradient Checking

To check if your gradient is computed correctly, follow these steps below:



$J(\theta)$ is a giant vector with all the parameters θ_i inside of it ->
$J(\theta) = J(\theta_1, \theta_2, \theta_3, \ldots)$

To check the gradient, use the following code:
for each i:

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

Now we need to check if the approximation is good compared to the original value:
$d\theta_{approx} \approx d\theta$?

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

When implementing these equations, use a really small value to ε, like $\varepsilon = 10^{-7}$. If the computed value above is approximately 10^{-7}, you have a really good approximation. If it is close to 10^{-5} you might take a careful look in the calculations, but it can be ok, and if it is 10^{-3} you have to check for errors and miscalculations, because the approximation is not good!.

Some tips about gradient checking:
-Don't use in training - only to debug, because the calculations are very costful.
-If algorithm fails grad check, look at components to try to identify bug.
-Remember regularization.
-Doesn't work with dropout, don't use both at the same time.
-Run at random initialization; perhaps again after some training.

Optimization Algorithms

Mini-batch gradient descent

Deep learning tends to work best in the regime of big data. But even when using techniques such as vectorization, with huge amounts of training examples the training can become very slow. When we are performing gradient descent, we have to manipulate all the training examples in every step of the gradient descent. This process can take a lot of time to run. In this case, we have a technique called mini-batch gradient descent, that consists in dividing the training data in lots of mini-batches (instead of manipulating a huge batch with all the training examples), in order to compute the gradient descent faster. Let's see how it works!

Let's consider a vector X with 5.000.000 training examples, and a vector Y with all the true labels:

$$X = [x^{(1)}, x^{(2)}, x^{(3)}, ..., x^{(1000)}, x^{(1001)}, ..., x^{(2000)}, ..., x^{(m)}]$$
$$Y = [y^{(1)}, y^{(2)}, y^{(3)}, ..., y^{(1000)}, y^{(1001)}, ..., y^{(2000)}, ..., y^{(m)}]$$

The idea behind mini-batch gradient descent is to divide this whole data in small parts that can be easily processed. Let's divide the vectors X and Y into mini batches of a thousand training examples each:

$$X = [x^{(1)}, x^{(2)}, x^{(3)}, ..., x^{(1000)} | x^{(1001)}, ..., x^{(2000)} | ..., x^{(m)}]$$
$$Y = [y^{(1)}, y^{(2)}, y^{(3)}, ..., y^{(1000)} | y^{(1001)}, ..., y^{(2000)} | ..., y^{(m)}]$$

The X vector is a (n_x, m) matrix. Each one of the mini-batches will be a $(n_x, 1000)$ matrix then. The same happens with the vector Y -> it is a $(1, m)$ vector. Each one of the mini-batches will be a $(1, 1000)$ vector.

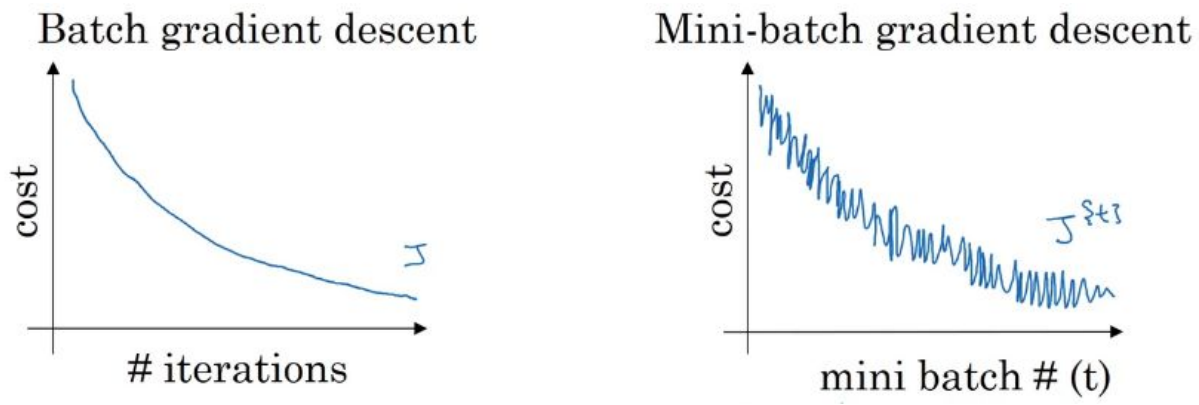
To differentiate each one of the mini-batches, let's introduce a new notation: $X^{\{t\}}, Y^{\{t\}}$, where t corresponds the number of the mini-batch (remember: $x^{(i)}$ represents the i-th training example, $z^{[l]}$ represents the z value for the l-th layer of the neural network and $X^{\{t\}}$ represents the mini-batch t from the training set matrix). Let's see how the mini-batch gradient descent is implemented.

$$for\ t = 1, ..., 5000\{$$
$$Forward\ prop\ on\ X^{\{t\}}$$
$$Z^{[1]} = W^{[1]}X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$...$$
$$A^{[L]} = g^{[L]}(Z^{[L]})$$
$$Compute\ cost\ J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{l} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2*1000}\sum_{i=1}^{l} ||w^{[l]}||_F^2$$
$$Backprop\ to\ compute\ gradients\ with\ respect\ to\ J^{\{t\}}\ (using\ (X^{\{t\}}, Y^{\{t\}}))$$
$$w^{[l]} := w^{[l]} - \alpha dw^{[l]}$$

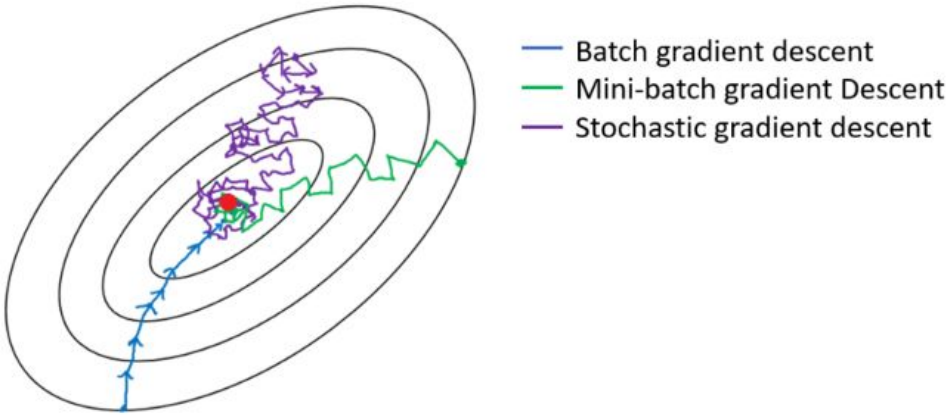
$$b^{[l]} := b^{[l]} - \alpha db^{[l]}$$

}

The code written above is called “1 epoch”. It means one forward pass and one backward pass through all the training examples.
When we use a single batch with all the training examples, the gradient descent takes bigger steps into the optimal value. When we use mini-batches, the gradient descent takes lots of mini-steps (in this example, it takes 5000 mini-steps, while using a single batch would take just 1 step). The cost function varies for each mini-bach, you can see the gradient descent using a single batch and mini-batches in the image below:



One of the parameters we have to choose is the size of the mini-batch. If we choose a mini-batch size = m, we would have the tradicional batch gradient descent. In the other extreme, if we choose a mini-batch size = 1, we’ll have the Stochastic gradient descent (where every example is its own mini-batch). We’ve already analyzed the batch gradient descent, so let’s analyse the stochastic gradient descent:
-It adds even more noise to the learning process, which helps improve generalization error. However, this would increase the run time;
-We can’t utilize vectorization over 1 example, which increases the running time. Also, the variance becomes large since we only use 1 example for each learning step.
In practice, what we usually use is a size between 1 and m, not too big nor too small, and this size will provide the fastest learning. Choosing an in-between size provides the following advantages:
-Use vectorization is now efficient;
-The gradient descent makes progress without the need to train the entire training set.
The path that the gradient descent takes in each of the sizes can be seen in the image below:

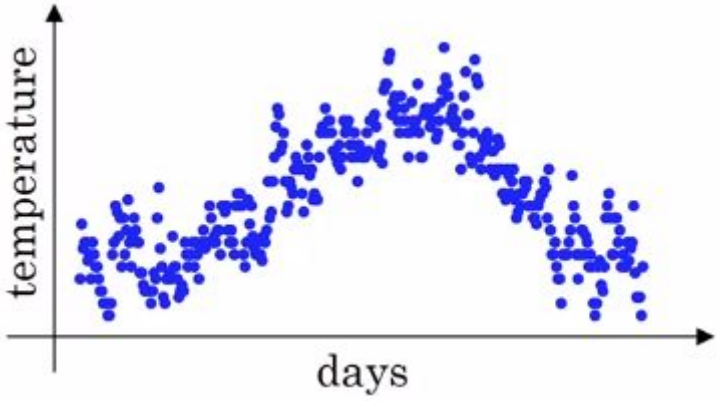


In conclusion:
-With small training set ($m \leq 2000$): Use batch gradient descent;
-With bigger training set: Use typical mini-batch sizes, such as 64, 128, 256, 512 (the code might end up with a better performance when using base-two batch sizes);
-Make sure your mini-batch fits in CPU/GPU memory.

Exponentially weighted averages

In order to introduce a few optimization algorithms, you need to be able to understand the exponentially weighted averages (also called exponentially weighted moving averages, in statistics). Let's use an example of the temperature in London throughout the year.

$$\theta_1 = 40°\text{F}$$
$$\theta_2 = 49°\text{F}$$
$$\theta_3 = 45°\text{F}$$
$$\vdots$$
$$\theta_{180} = 60°\text{F}$$
$$\theta_{181} = 56°\text{F}$$
$$\vdots$$



Let's make some equations:

$V_0 = 0$ *#initializing the parameter with zero*

$V_1 = 0.9V_0 + 0.1\theta_1$ *#for everyday we are going to average with the weight of* 0.9 *times the previous va*

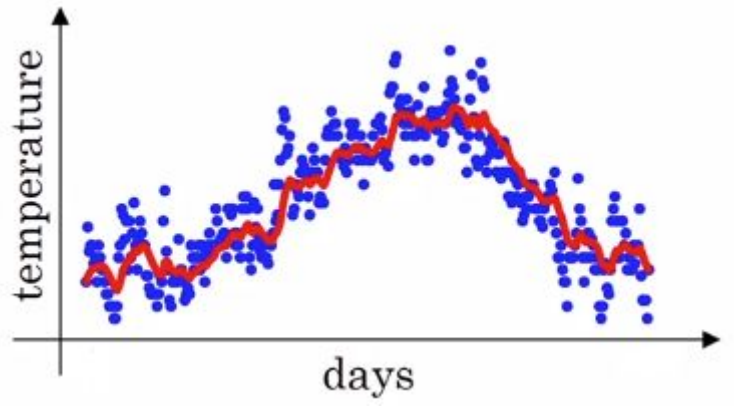
$V_2 = 0.9V_1 + 0.1\theta_2$ *#... +* 0.1 *times that days temperature*

$V_3 = 0.9V_2 + 0.1\theta_3$

…

$V_t = 0.9V_{t-1} + 0.1\theta_t$

This plot below is what you get:



You get an exponentially weighted average of the daily temperature. Let's generalize the formula:

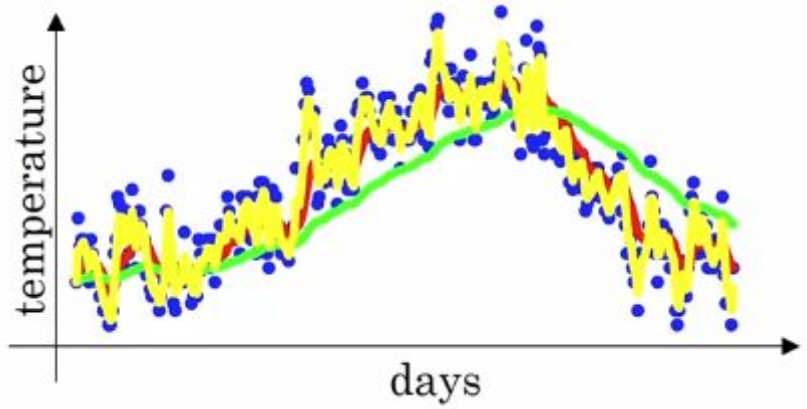
$V_t = \beta V_{t-1} + (1-\beta)\theta_t$

In the example above, we've had $\beta = 0.9$. We can see the value V_t as approximately an average over $\approx \frac{1}{1-\beta}$ days' temperature. You can see how the plot changes when we change the value of the β (actually, β is a hyperparameter).

$\beta = 0.9 \approx 10$ *days' temperature* (*represented as the red line*)

$\beta = 0.98 \approx 50$ *days' temperature* (*represented as the green line*)

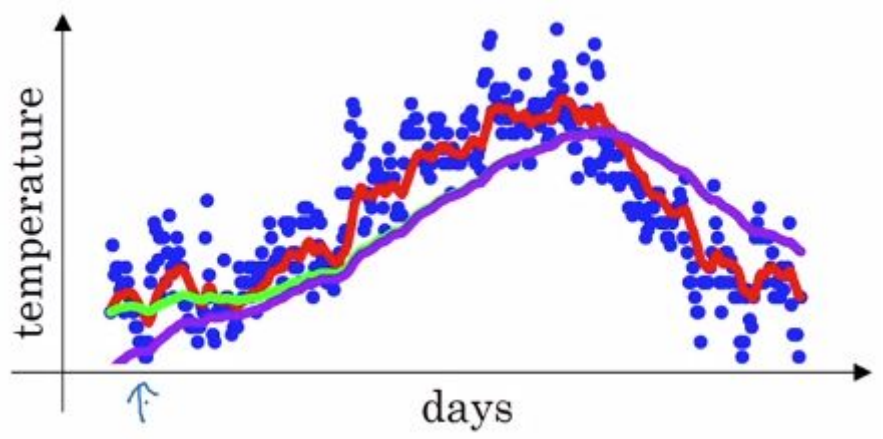
$\beta = 0.5 \approx 2$ *days' temperature* (*represented as the yellow line*)



You can see, that for $\beta = 0.98$, the green line is much smoother (because it represents the average of a larger period of time) and it gets shifted a little to the right because in the equation we give much weight to the previous days ($0.98V_{t-1}$) and a small weight to the actual day ($0.02\theta_t$). While the yellow line is much noisier, because it represents the weighted average of just two days.

Bias correction in exponentially weighted averages

We saw in the previous section, that for a value of $\beta = 0.98$, the plot we would get is the green one. However, in reality the plot we get is the purple one, that can be seen below:



It happens because when we initialize the parameter V_0 with zero, a term in V_1 will be equals to zero, not corresponding the real value V_1 should be.

$V_0 = 0$

$V_1 = 0.98V_0 + 0.02\theta_1 = 0.02\theta_1$

$V_2 = 0.98V_1 + 0.02\theta_2 = 0.98 * 0.02 * \theta_1 + 0.02 * \theta_2$

…

This initialization will make the first values too small, this is why the purple line starts lower compared to the green line. The bias correction fixes the problem. Let's see how it's calculated:

Instead of using just V_t in the general equation, we'll divide the whole equation by the following term:

$1 - \beta^t$

Let's divide the general equation by this term:

$$\frac{V_t}{1-\beta^t} = (\beta V_{t-1} + (1-\beta)\theta_t) / (1-\beta^t)$$

This term will adjust the bias in the start of the plot. If t becomes large, the number $(1-\beta^t)$ becomes close to 1 (remember β is always lower than 1), and the bias correction doesn't affect the function anymore, solving our problems!

Gradient descent with momentum

The gradient descent with momentum almost always works faster than the standard gradient descent algorithm. The basic idea is to compute an exponentially weighted average of your gradients, and then use that gradient to update your weights. The algorithm to this method is described below:

On iteration t :

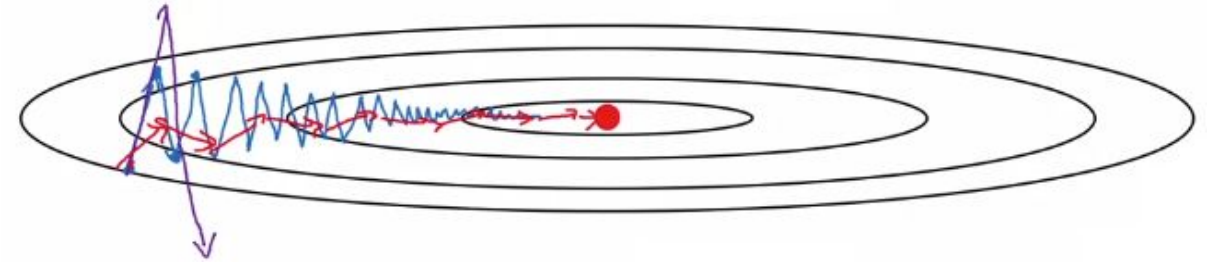
Compute dw, db on current mini − batch

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

$$V_{db} = \beta V_{db} + (1-\beta)db$$
$$w := w - \alpha V_{dw}$$
$$b := b - \alpha V_{db}$$

Hyperparameters: α, β $\beta = 0.9$

What this technique does, is to smooth out the steps of gradient descent. This prevents the gradient from overshooting and diverging. You can see the steps taken by the gradient descent in the image below. The blue path represents the tradicional gradient descent, using batch or mini-batches. The purple overshoot represents a possible error in the tradicional gradient descent calculations. The red path represents the steps taken by the gradient descent with momentum.



RMSprop (Root Mean Square Prop)

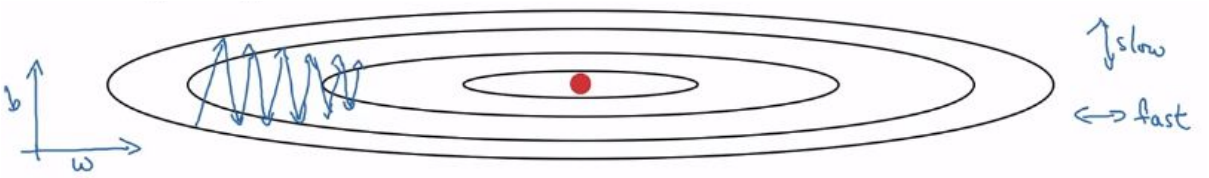
If you implement gradient descent, you can end up with huge oscillations in some direction. The RMSprop algorithm gives a solution to this problem. You can see the implementation of the algorithm below:

On iteration t :

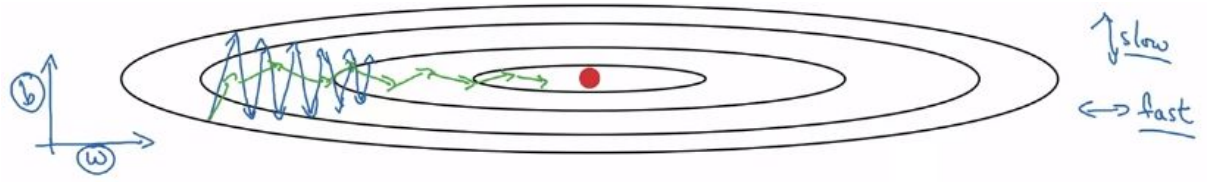
$$Compute\ dw,\ db\ on\ current\ mini-batch$$
$$S_{dw} = \beta S_{dw} + (1-\beta)dW^2\ (element-wise\ square\ multiplication)$$
$$S_{db} = \beta S_{db} + (1-\beta)db^2$$
$$w := w - \alpha \frac{d_w}{\sqrt{S_{dw}+\varepsilon}}$$
$$b := b - \alpha \frac{d_b}{\sqrt{S_{db}+\varepsilon}}$$

The factor ε is used to add numerical stability, making sure your algorithm doesn't divide by zero in cases the value of S_{di} is too small ($\varepsilon \approx 10^{-8}$).

Let's analyse the same example from the Gradient Descent with Momentum:



In this case, if we consider the vertical values equal to b, and the horizontal values equal to w, there's a possibility that b diverges in some point of the gradient descent. To avoid that, we want b to update slowly, compared to w. In the equations, what solve this problem is the term $\sqrt{S_{di}}$ -> if w is a small number, it will get updated quickly (because the root term is dividing the updating equation), and if b is a large number, it will get updated slowly. These division terms avoid the larger parameters to diverge/overshoot. The path taken by the RMSprop algorithm is represented as the green line in the image below:



Adam optimization algorithm

The Adam optimization algorithm is basically taking the gradient descent with momentum and RMSprop and putting them together. So let's see how it is implemented:

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

$$On\ iteration\ t:$$

$$Compute\ dw,\ db\ on\ current\ mini-batch$$

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)dW$$

$$V_{db} = \beta_1 V_{db} + (1-\beta_1)db$$

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)dW^2$$

$$S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2$$

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

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

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

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

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

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

This is a commonly used algorithm that is proven to be very effective for many different neural networks of a wide variety of architectures. The common choices for the hyperparameters value can be seen below:

α : *needs to be tuned*

β_1 : 0.9

β_2 : 0.999

ε : 10^{-8}

Curiosity: Adam comes from “Adaptive moment estimation”.

Learning Rate Decay

One of the things that might help speed up your learning algorithm, is to slowly reduce your learning rate over time. We call this learning rate decay. The intuition behind slowly reducing α, is that maybe during the initial steps of learning, you could afford to take much bigger steps. But then, as learning approaches convergence, then having a slower learning rate allows you to take smaller steps, improving the performance of the learning. So here’s how you can implement learning rate decay: (Remember: 1 epoch = 1 pass through all the training data)

$$\alpha = \frac{1}{1 + decay\_rate * epoch\_num} * \alpha_0$$

Where decay_rate is another hyperparameter, epoch_num is the number of the current epoch and α_0 is the initial learning rate. The values of decay_rate and α_0 can be determined by empirical processes (test different values multiple times and check which ones are good values).

There are other learning rate decay methods, you can see them below:

$\alpha = 0.95^{epoch\_num}.\alpha_0$ (exponentially decay)

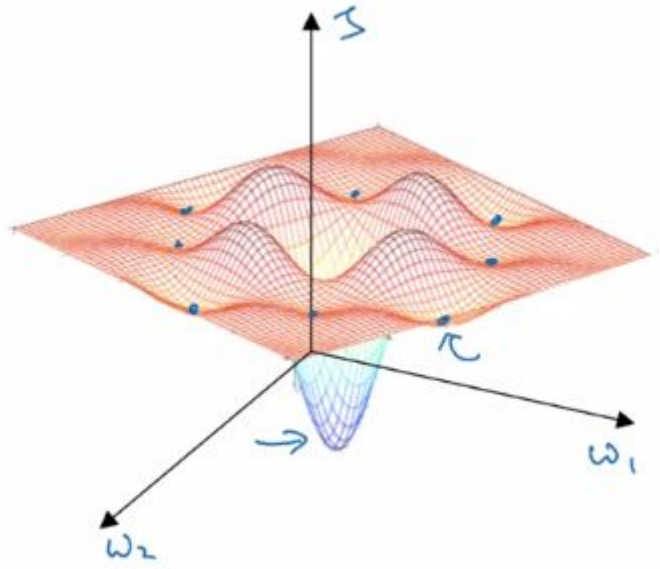
$\alpha = \frac{k}{\sqrt{epoch\_num}}.\alpha_0$ (where k is a constant)

$\alpha = \frac{k}{\sqrt{t}}.\alpha_0$ (where t is the number of the current mini-batch)

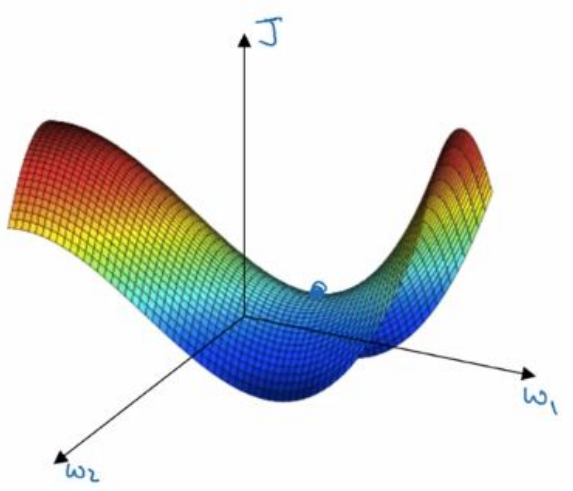
If you are trying to train a small number of modules, you can also change the learning manually (checking the training in periods of time and if necessary, change the learning rate while the learning is still going).

The problem of local optima

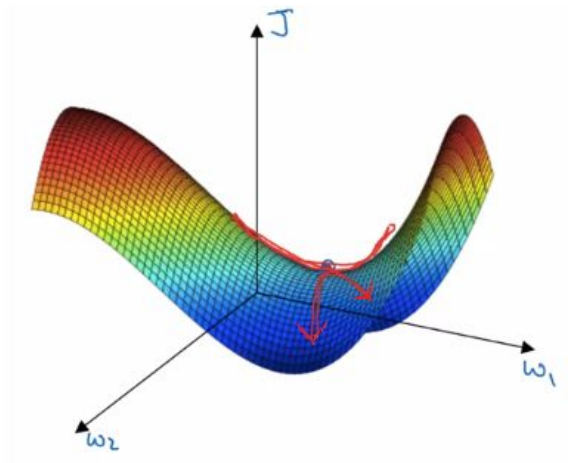
In the early days of deep learning, people used to worry a lot about the optimization algorithm getting stuck in a bad local optima. This problem can be seen in the picture below:



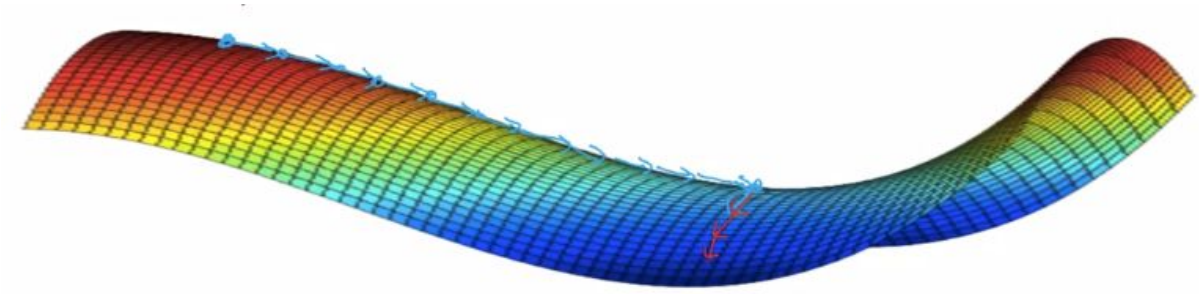
But as this theory of deep learning has advanced, our understanding of local optima is also changing. It turns out that when you are training a neural network, most points of zero gradient in the cost function are actually saddle points, like in the image below:



However, if you are training a high dimension neural network, to spot a local optima you need all the directions in that point to be optimal, and the chance of this happening is quite small. When you are in a saddle point, for example, the gradient can be improved because you can go down in some direction, like in the image below:



So for today's problems (that have multiple dimensions), it'll be difficult to get stuck in some local optima.
There is also the problem with the plateaus. The plateaus are regions in which the gradient is very small, slowing the learning process. As you can see in the next image, the blue path represents a path throughout a plateau region. The gradient descent takes a lot of time to exit these regions, and finally accelerate the learning process again (represented by the red path).



In conclusion:
-It is unlikely to get stuck in a bad local optima, so long as you are training a reasonably large neural network.
-Plateaus can make learning slow.

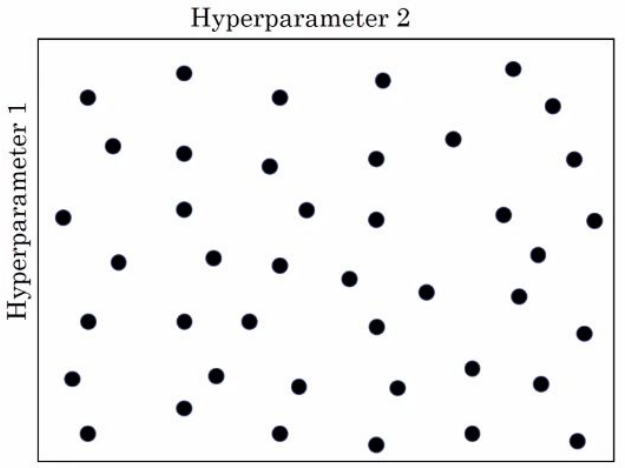
Hyperparameter tuning

Tuning Process

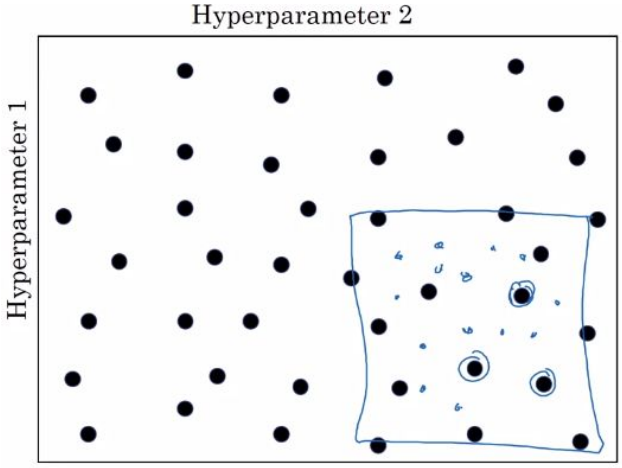
We've seen that training neural networks can involve setting a lot of different hyperparameters, such as α, β, β_1, β_2, ε, # hidden layers, # hidden units, learning rate decay, mini-batches size, and so on… . Some hyperparameters work fine almost always with the same value, such as the hyperparameters from the Adam optimization ($\beta_1 = 0.9,\ \beta_2 = 0.999,\ \varepsilon = 10^{-8}$), and don't need to be tuned that often. Other hyperparameters, like the learning rate α are very important hyperparameters, and usually need to be tuned in order to have a better performance. The priority of the hyperparameters tuning can be seen below:

High priority: α
Medium priority: β , # hidden units
Low priority: # hidden layers, learning rate decay
Almost never tuned: β_1, β_2, ε

To tune your hyperparameters, you can try random values and then choose the values that give a better performance. One example can be seen in the picture below. Each dot inside the square is a random value for the hyperparameters. If you want to tune lots of hyperparameters at the same time, the dimension will get bigger.



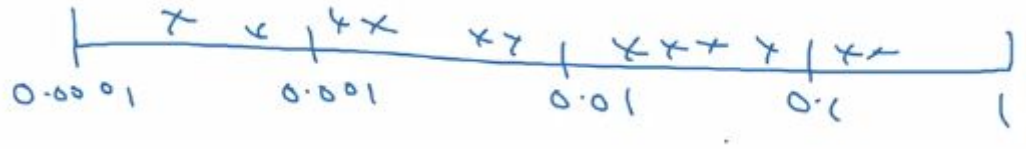
To optimize the searching of the best hyperparameters, you might check some regions of the square, and check which regions perform the best. Finding a good region, you can plot more points in that region and focus your analysis on it. In the image below, the bottom-right corner performed better compared to the other regions, so you might focus in that region.



Using an appropriate scale to pick hyperparameters

Let's start with an example. Imagine you have to tune the hyperparameter α. You think the optimal value for α is between 0.0001 and 1. In the first moment, you might think to uniformly sample random values around this interval.

However, these are not good values. Around 90% of the random values will be between 0.1 and 1. Only 10% of the random values will be between 0.0001 and 0.1, this is a very small percentage to search in this promising interval. Instead of using a linear scale, you might want to consider using a logarithmic scale, as you can see in the image below:



This scale will allow your algorithm to check more random values between 0.0001 and 0.1, improving the search for the optimal value.
You can see below the Python implementation:

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

$\alpha = 10^r$ $\quad \# \alpha \in [10^{-4}, \ldots, 10^0]$

Another example: You are tuning the parameter β, and you think that there is a good value of β between 0.9 and 0.999. This is what you should do:

$Compute\ (1-\beta) = 0.1, \ldots, 0.001$

$Compute\ the\ log_{10}\ of\ the\ numbers\ in\ the\ extremes\ of\ the\ interval$

$In\ this\ case,\ [10^{-1}, \ldots, 10^{-1}] \rightarrow r \in [-3, -1]$

$Set\ (1-\beta) = 10^r$

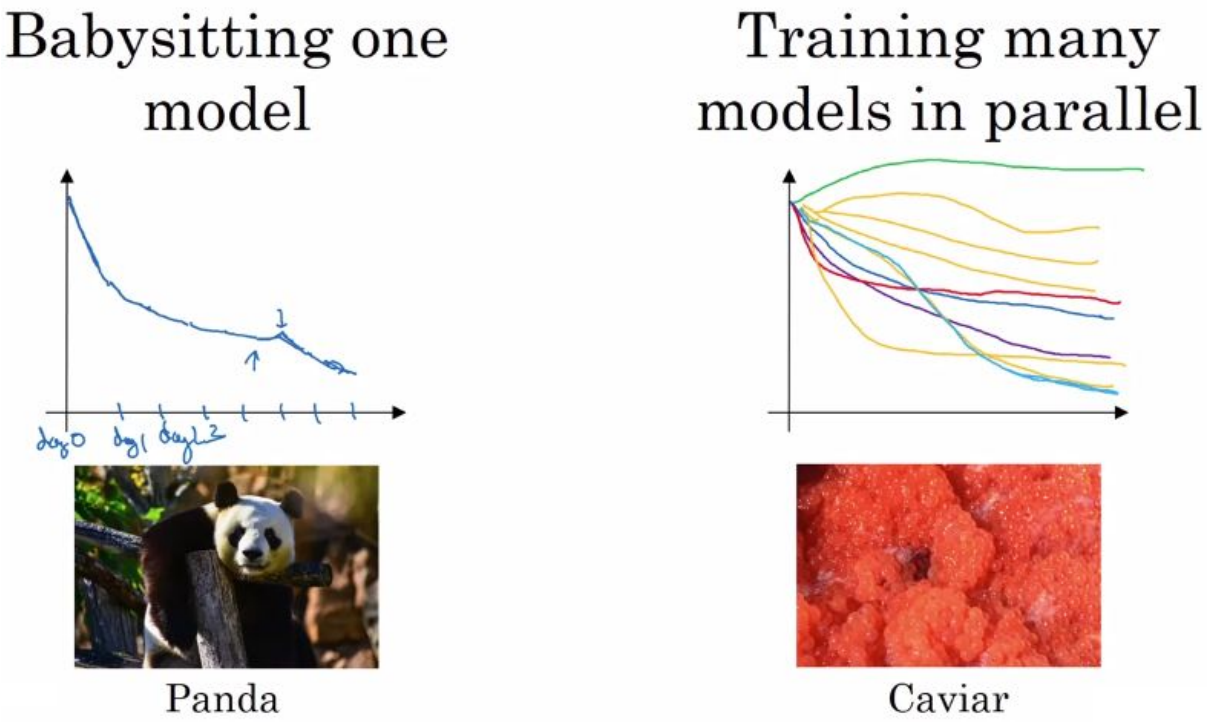
$\beta = 1 - 10^r$

This will make your scale much better, and your resources will be focused in good intervals.

Hyperparameters tuning in practice: Pandas vs. Caviar

When you have lower computational resources, you might want to “babysit” one single model. Checking if the model is doing great day after day, changing the hyperparameters… . However, if you have a high computational power, you might want to run many models in parallel, and at the end of the training, you choose the model that fits the best your data.
The babysitting way of setting good hyperparameters is similar to what pandas do to their children. Pandas usually have a small number of children and pay much attention to make sure the baby pandas survive. In the other way, fishes usually lay thousands of eggs in each season, and don’t pay too much attention to any of them,

but just hope some of them will survive. You can see this analogy in the image below:



Batch Normalization

Normalizing activations in a network

In the previous studies, we've seen how to normalize a single neuron of a neural network. We'll study now how to normalize activations in large networks. The objective of this section, is to learn how to normalize $z^{[l]}$, in order to train $w^{[l+1]}$, $b^{[l+1]}$ faster, using batch normalization (batch norm). Let's see how to implement Batch Norm:

$$\textit{Given some intermediate values in a layer } l \textit{ in } NN: \; z^{(1)}, \ldots, z^{(m)}$$

$$\mu = \frac{1}{m}\sum_{i=1}^{m} z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(z^{(i)} - \mu)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

With these formulas, we now have zero mean and standard unit variance. However, it might be useful sometimes to have hidden units with different means and variances. In this case, we use the following formula:

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

In this formula, γ and β are learnable parameters of the model. The effect of γ and β is that it allows you to set the mean of $\tilde{z}$ to be whatever you want it to be. In fact, if the values of γ and β are the following:

$$\gamma = \sqrt{\sigma^2 + \varepsilon}$$

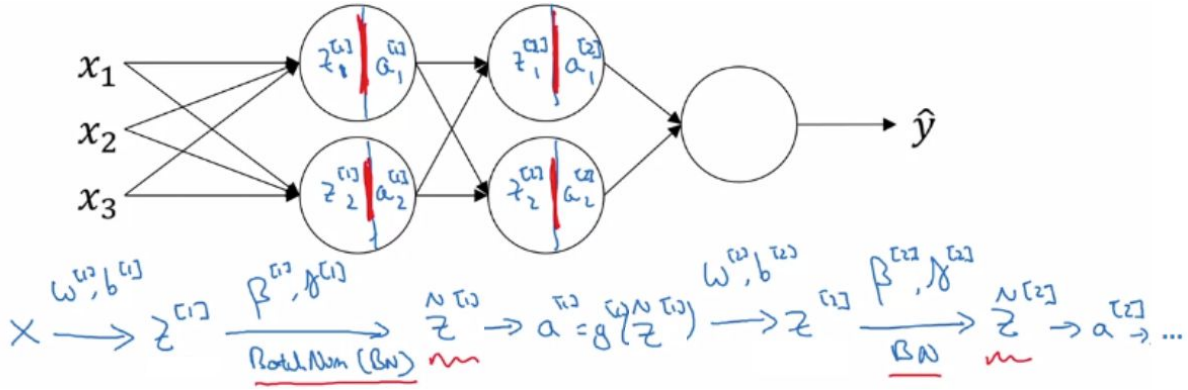
and

$$\beta = \mu$$

Then you'll end up with $\tilde{z}^{(i)} = z^{(i)}$ (to test it, just put these values in the $\tilde{z}^{(i)}$ equation and you'll realise that). So by choosing other values of γ and β, this allows you to make the hidden unit values have other variances and means.

Fitting Batch Norm into a neural network

To add the Batch Norm into a neural network, instead of computing the activation function with the regular value of z, you use the value of $\tilde{z}$. This change will normalize your hidden layers.



As you can see in the image above, the input values are uses with the parameters w and b to compute the z value. Then, using the Batch Norm the value of $\tilde{z}$ is computed. And finally the value of the activation function of $\tilde{z}$ is computed, passing this value to the next hidden layer and the process starts again, until the output is computed. With this method, we have new parameters to deal with:

$$Parameters: W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, ..., W^{[L]}, b^{[L]}, \beta^{[1]}, \gamma^{[1]}, \beta^{[1]}, \gamma^{[2]}, ..., \beta^{[L]}, \gamma^{[L]}$$

Note that $\beta^{[L]}$ is not the β hyperparameter used to compute the exponentially weighted averages algorithms (such as Adam and momentum). We can update these new parameters using gradient descent, in the same way we've already done a lot of times updating the W and b parameters (or if you prefer, you can also use Adam, RMSprop or momentum in order to update these parameters).

Batch Norm is usually applied with mini-batches of your training set. The only difference is that instead of using the whole batch to compute the functions, you're going to compute the functions with the mini-batches. One other important detail you might want to look at, is that the Batch Norm computes the means of the z, and if you add some constant to these z values, the constants will end up being cancelled. What this means is that we can get rid off the parameters $b^{[l]}$ in the z functions, because in the end it will be cancelled.

Implementing gradient descent::

$for\ t = 1, \ldots, numMiniBatches:$

$compute\ forward\ prop\ on\ X^{\{t\}}$

$In\ each\ hidden\ layer,\ use\ BN\ to\ replace\ z^{[l]}\ with\ \tilde{z}^{[l]}$

$use\ back\ prop\ to\ compute\ dW^{[l]},\ d\beta^{[l]},\ d\gamma^{[l]}$

$update\ parameters:\quad W^{[l]} := W^{[l]} - \alpha dW^{[l]}$

$\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$

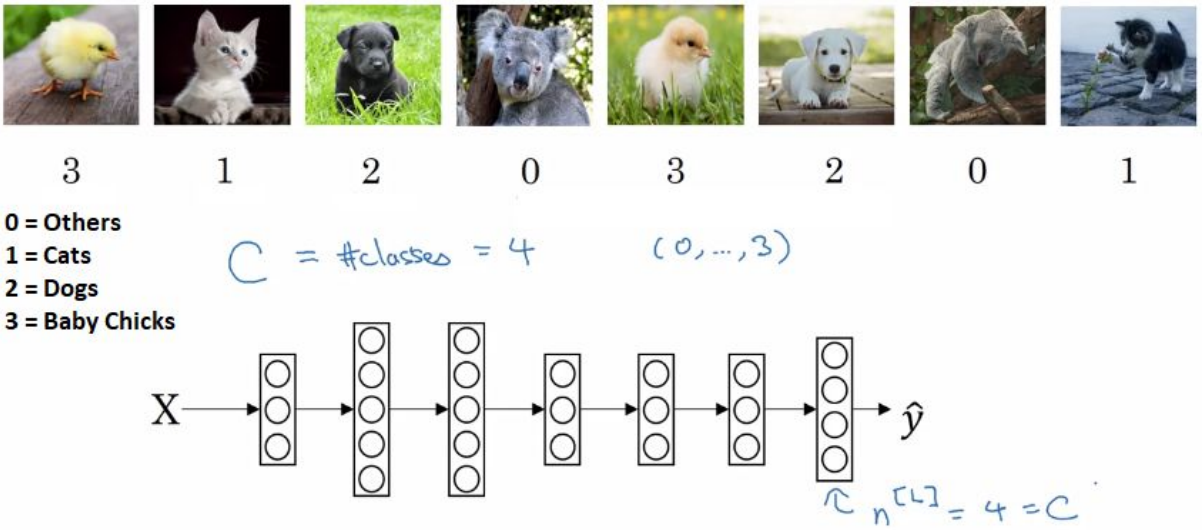
$\gamma^{[l]} := \gamma^{[l]} - \alpha d\gamma^{[l]}$

Remember: this technique also works with momentum, RMSprop and Adam optimization algorithms.

The Batch Norm also solves the problem of Covariate Shift. Covariate shift refers to the change in the distribution of the input values to a learning algorithm. For instance, if the train and test sets come from entirely different sources, the distributions would differ. The reason covariance shift can be a problem is that the behaviour of machine learning algorithms can change when the input distribution changes.
The basic idea behind batch normalization is to limit covariate shift by normalizing the activations of each layer (transforming the inputs to be a common mean and common variance). This, supposedly, allows each layer to learn on a more stable distribution of inputs, and would thus accelerate the training of the network.

Softmax Regression

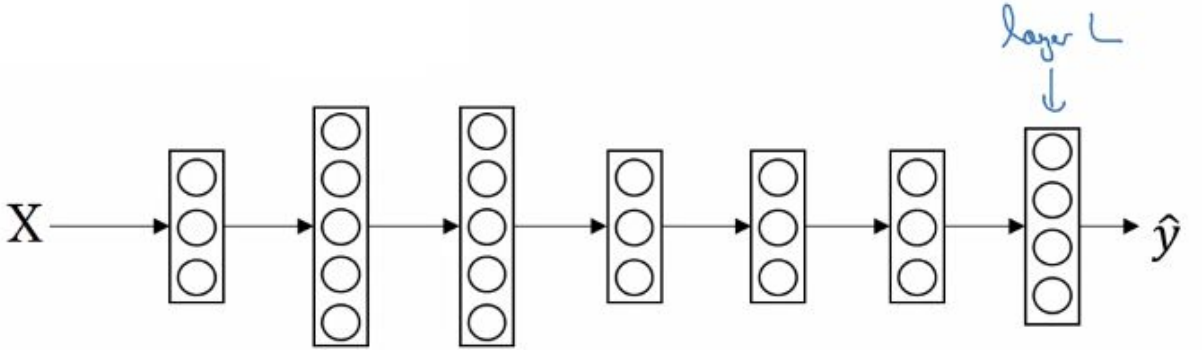
Softmax regression is a generalization of logistic regression. Instead of recognizing just two classes (like in the cat recognizer model - is a cat (1) or not (0)), Softmax regression allows your model to recognize one of multiple classes. Imagine you are training a neural network to recognize cats, dogs and baby chicks. In this case, your neural network is going to have 4 classes (cats, dogs, baby chicks and others). We use the notation C to denote the number of classes. In this case, C = 4 (0, …, 3).



As you can see in the picture above, the output layer of the neural network is going to have 4 neurons, one for each class. The dimension of the output vector $\hat{y}$ is (4,1). Each one of the neurons in the output layer is going to have a probability output, and the sum of all the probabilities is going to be 1. For example:
For an input x image:

P(other | x) = 0.5%
P(cat | x) = 97%
P(dog | x) = 2.5%
P(baby chicks | x) = 0%

In order to generate these outputs, we use the Softmax layer in the layer L. You can see a representation of the neural network below:



The equations of this layer are:

$$Z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$$

Activation function:

$$t = e^{(Z^{[L]})}$$

$$a^{[L]} = \frac{e^{(Z^{[L]})}}{\sum_{i=1}^{4} t_i}$$

$$a_i^{[L]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

Let's do an example:

Assume for a given input X, you have the following z in the Softmax layer:

$$Z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$$

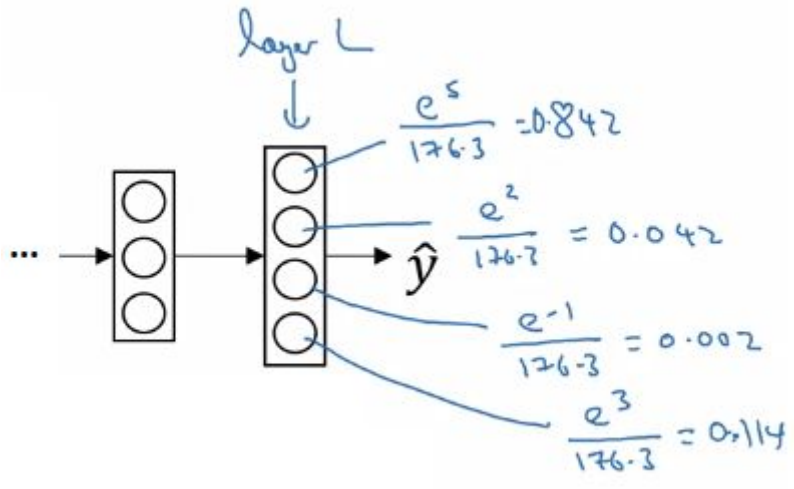
With these values, we can now compute the values of t, and the sum of all t terms:

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix} \qquad \sum_{i=1}^{4} t_i = 176.3$$

Once these values are computed, we can now compute the activations of each neuron, following this equation:

$$a^{[L]} = \frac{t}{176.3}$$

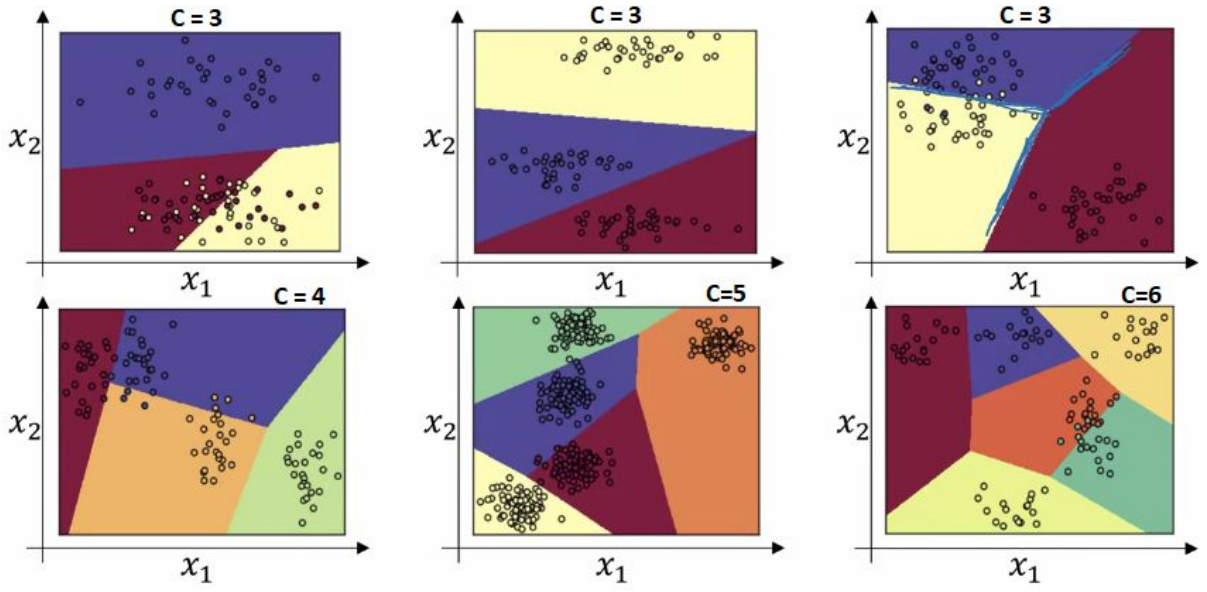
Finally, we'll have these probabilities:



As we can see, it's more likely that the input is some random image (84.2%), and not a cat (4.2%), a dog (0.2%) nor a baby chick (11.4%). Then:

$$\hat{y} = a^{[L]} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

You can see in the image below, some softmax examples:



In conclusion: softmax regression generalizes logistic regression to C classes (If C=2, softmax reduces to logistic regression).

This is how you implement the loss function with a softmax classifier, in a NN with C classes:

$$L(\hat{y}, y) = -\sum_{j=1}^{C} y_j log\hat{y}_j$$

$$J(W^{[1]}, b^{[1]}, ...) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

And the gradient descent:

$$Backprop: \; dz^{[L]} = \hat{y} - y$$

Let's analyse the dimensions of these variables:

$$dim\, y^{(i)} = dim\, \hat{y}^{(i)} = dim\, dz^{[L](i)} = (C, 1)$$

In case you vectorize $y^{(i)}$:

$$dim\, Y = dim\, \widehat{Y} = dim\, dZ^{[L]} = (C, m)$$

Deep Learning Frameworks

Deep learning frameworks are tools that help you to train and model your neural networks without the necessity to implement the NN from scratch. You can see below a list of the most popular frameworks:
-Caffe/Caffe2
-CNTK
-DL4J
-Keras
-Lasagne
-mxnet
-PaddlePaddle
-TensorFlow
-Theano
-Torch

You have to do some considerations before choosing a framework:
-Ease of programming (development and deployment)
-Running speed
-Truly open (open source with good governance)

TensorFlow

In this section, we'll study a deep learning framework called TensorFlow. Let's do an example:

Imagine you want to optimize the cost function $J(w) = w^2 - 10w + 25$. The optimal value of this function is $w = 5$. Let's see how it is implemented using TensorFlow:

[7]:
```
import numpy as np
import tensorflow as tf

w = tf.Variable(0, dtype=tf.float32)  #initialize the variable to zero
cost = tf.add(tf.add(w**2, tf.multiply(-10. , w)), 25)  #define the cost function
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
#set the train, the optimization method (gradient descent),
#the learning rate (0.01) and minimize the cost function (cost)
```

[8]:
```
init = tf.global_variables_initializer()
session = tf.Session()   #starts a tensorflow session
session.run(init)   #initialize the global variables
```

[9]:
```
for i in range(1000):     #run 1000 iterations of gradient descent
    session.run(train)
print(session.run(w))
```

```
4.9999886
```

You can also use the traditional notation for the numerical expression of the cost function, TensorFlow will work just fine:

[3]:
```
import numpy as np
import tensorflow as tf

w = tf.Variable(0, dtype=tf.float32)  #initialize the variable to zero
#cost = tf.add(tf.add(w**2, tf.multiply(-10. , w)), 25)  #define the cost function
cost = w**2 - 10*w + 25
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
#set the train, the optimization method (gradient descent),
#the learning rate (0.01) and minimize the cost function (cost)

init = tf.global_variables_initializer()
session = tf.Session()   #starts a tensorflow session
session.run(init)   #initialize the global variables

for i in range(1000):     #run 1000 iterations of gradient descent
    session.run(train)
print(session.run(w))
```

```
4.9999886
```

Now imagine the function you want to minimize is a function of your training set. You'll have some training data x, and when training the neural network the value of x can change. In this case, this is what you should do:

In [8]:
```
import numpy as np
import tensorflow.compat.v1 as tf #have to use this in order to use tensorflow
tf.disable_v2_behavior() #in my computer, the version of the class is obsolete


coefficients = np.array([[1.], [-10.], [25.]]) #the data we are going to use in x

w = tf.Variable(0, dtype=tf.float32)  #initialize the variable to zero
x = tf.placeholder(tf.float32, [3,1]) #initialize variables, dimension =  [3,1]
#cost = tf.add(tf.add(w**2, tf.multiply(-10. , w)), 25)  #define the cost function
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
#set the train, the optimization method (gradient descent),
#the learning rate (0.01) and minimize the cost function (cost)

init = tf.global_variables_initializer()
session = tf.Session()   #starts a tensorflow session
session.run(init)   #initialize the global variables

for i in range(1000):     #run 1000 iterations of gradient descent
    session.run(train, feed_dict= {x:coefficients})
print(session.run(w))
```

```
4.9999886
```