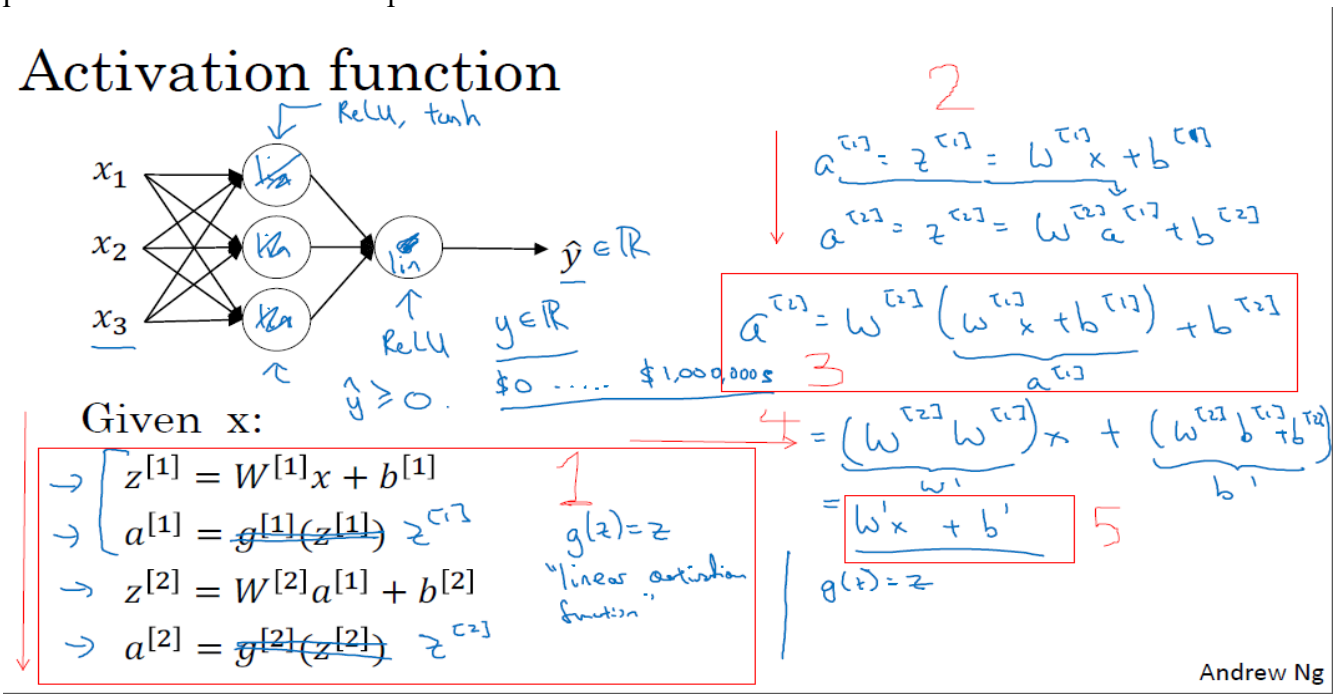
Course 1:

Activation Functions

One place that we use a linear activation function is when we want to do a kind of regression problem (continuous values, not with classes). In this case, we only use linear activation function at the output layer only. And for the rest of layers we use tanh or ReLu activation function.
If we want to use a linear activation function all over the places, we end up with a Linear Regression problem as can be seen in the picture below.



For the derivatives of **activation functions** we have: (look at the activation function figures as well)

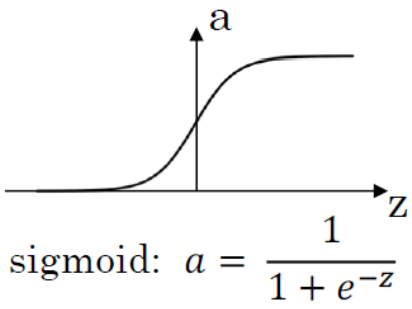
Sigmoid: It totally makes sense to use sigmoid function for the output layer if the expected output is either 0 or 1 (binary classification), otherwise, better to use tanh function for the middle layers rather than the sigmoid one. The reason is written on tanh description part. (look at the next item)
$g(z) = \frac{1}{1+e^{-z}}$ ➔ $g'(z) = g(z) \times \big(1 - g(z)\big)$

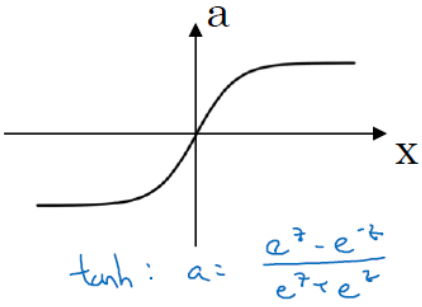
Tanh: It is better to use tanh instead of sigmoid since the values are shifted between 1 and -1, so that the mean goes to 0 rather than 0.5 which makes learning for the next layers much easier.
$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$ ➔ $g'(z) = 1 - g(z)^2$

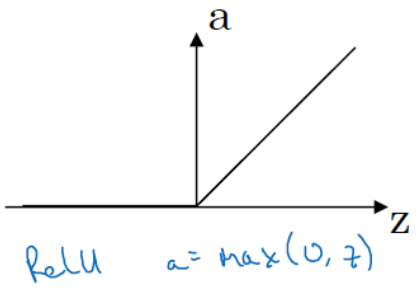
ReLu: The problem with tanh and sigmoid functions are that, when the values go toward two ends of the function, the derivative goes to 0, which slowing down the gradient descent. In this function only for the negative values we have derivative of 0 but for higher than that the derivative is 1. In most of the cases it is the default choice of activation function as it is faster on gradient descent.
$g(z) = \max(0, z)$ ➔ $g'(z) = 0 \; if \; z < 0 \; else \; 1$

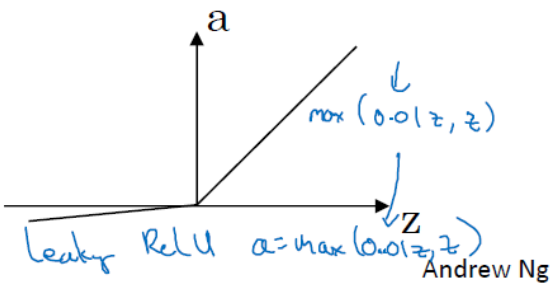
Leaky ReLu: If having a 0 as derivative of negative values is problematic for a problem, this version works.

$g(z) = \max(0.01 \times z, z)$ ➔ $g'(z) = 0.01 \; if \; z < 0 \; else \; 1$









Weight Initialization

What happens when we initialize weights with 0s? It causes to weights to have same rows and causes **symmetric weights** meaning all the weights compute same function as their effect on the output at the very first was the same. So it is useless to do so.

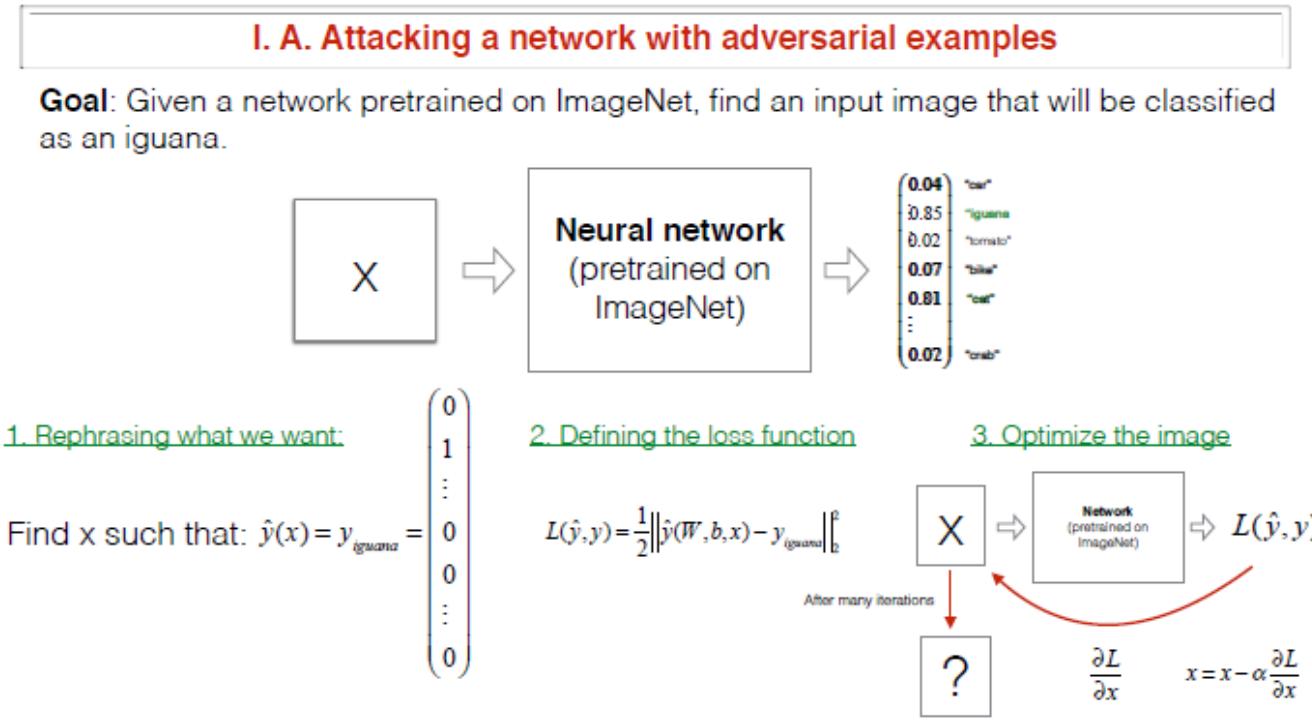
So we initialize it with very small random values. Why small? Because if we use sigmoid or tanh, the weights tend to get large and the gradient descent takes very small steps as it gets so close to 0.

CS230: Lecture 3

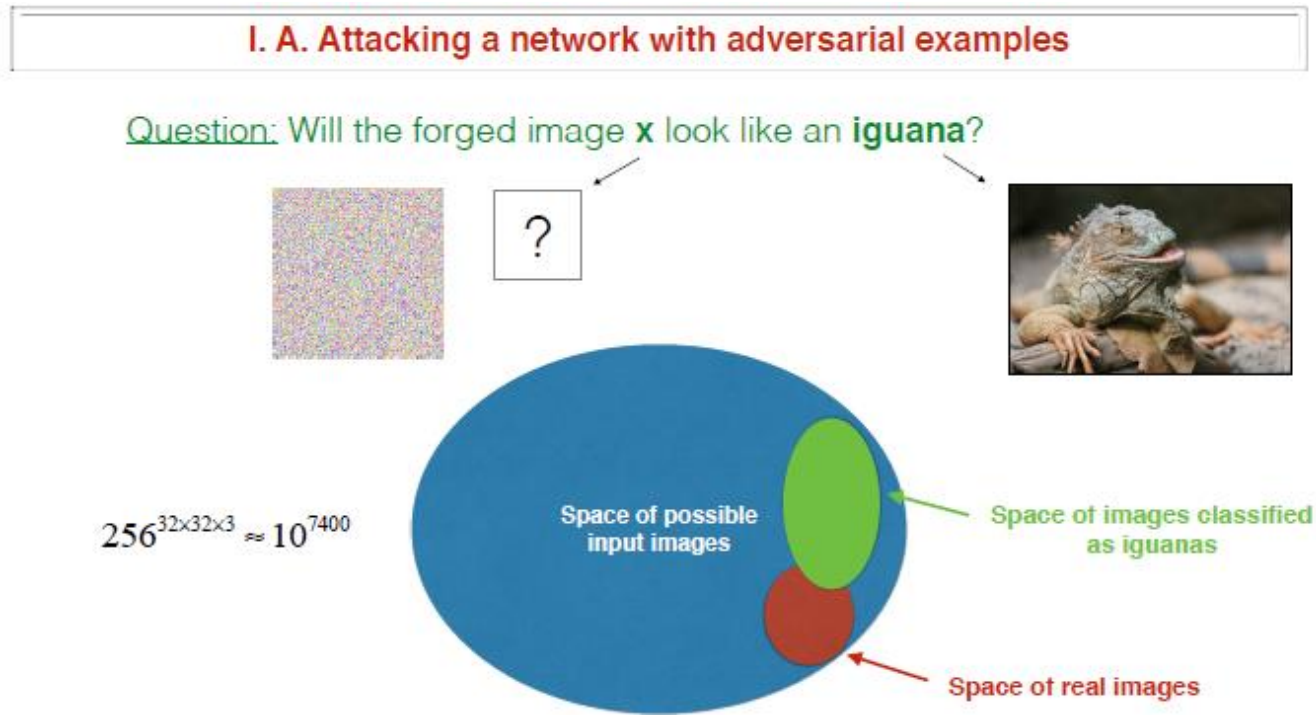
Attacking Networks with Adversarial Examples

The goal of attacking a network, with adversarial examples is:
Given a network pertained on ImageNet, we want to find an input image, that is not an iguana, but will be classified as an iguana by the network.

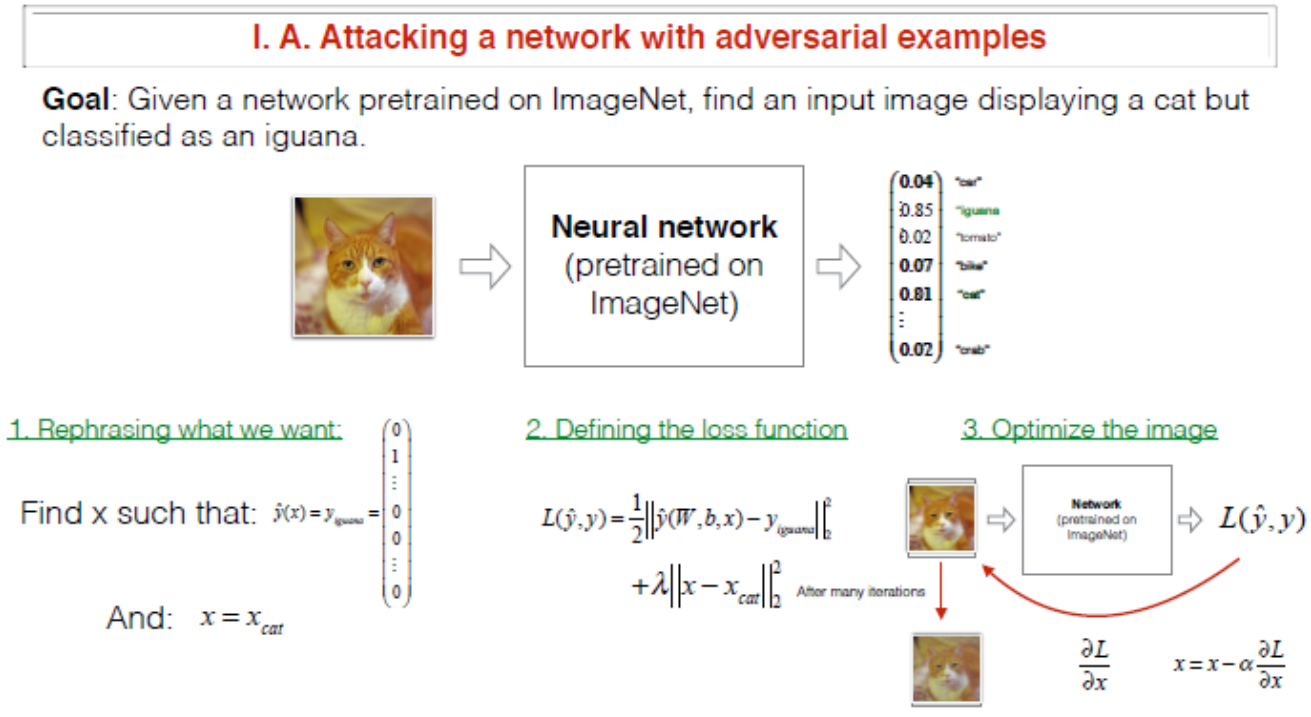
For a good network, we expect the output probabilities of an input cat image, have maximum probability for the cat (out of Softmax). But, we want to fool the network in a way to see highest probability for the iguana instead. If we want to formulize the problem, we can see it in the slide below. Note that the loss function could be any other loss like cross entropy.



The space of all possible images for a 32×32 image, with 3 RGB channels would be: $256^{32 \times 32 \times 3}$. The space of real and iguana images are shown in red and green respectively. There is a slight overlap between the red and green space. The overlap is where we input an image of a cat and the network finds it as an iguana. Some of its malicious applications could be used in passing face recognition, CAPTCHAs, or violent content detection in social media.
So, in order to fool the network, our loss function needs some constraints which pushes it to the overlapped space.



In our loss, we want to minimize the difference of the output image and the iguana one as well the difference of the output image and a cat, simultaneously. We can now give the image of the same cat to the network and get an image which is so similar to a cat (human can differentiate easily) but outputted as an iguana.



Now let's see how it is going to be:

I. A. Attacking a network with adversarial examples

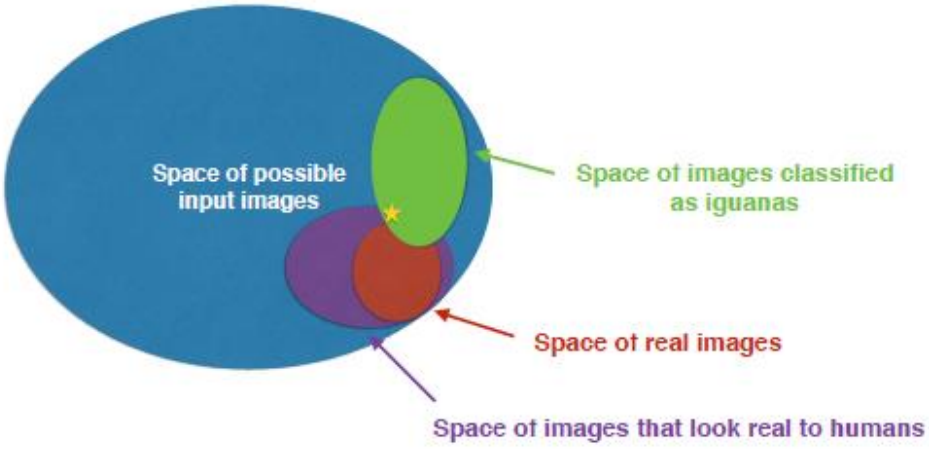
92% Cat

94% Iguana

Why is it happening? Because the space of the images that look real to human is much more than the space of real images, like the right one you see above; it is a blurry image that look real to human.

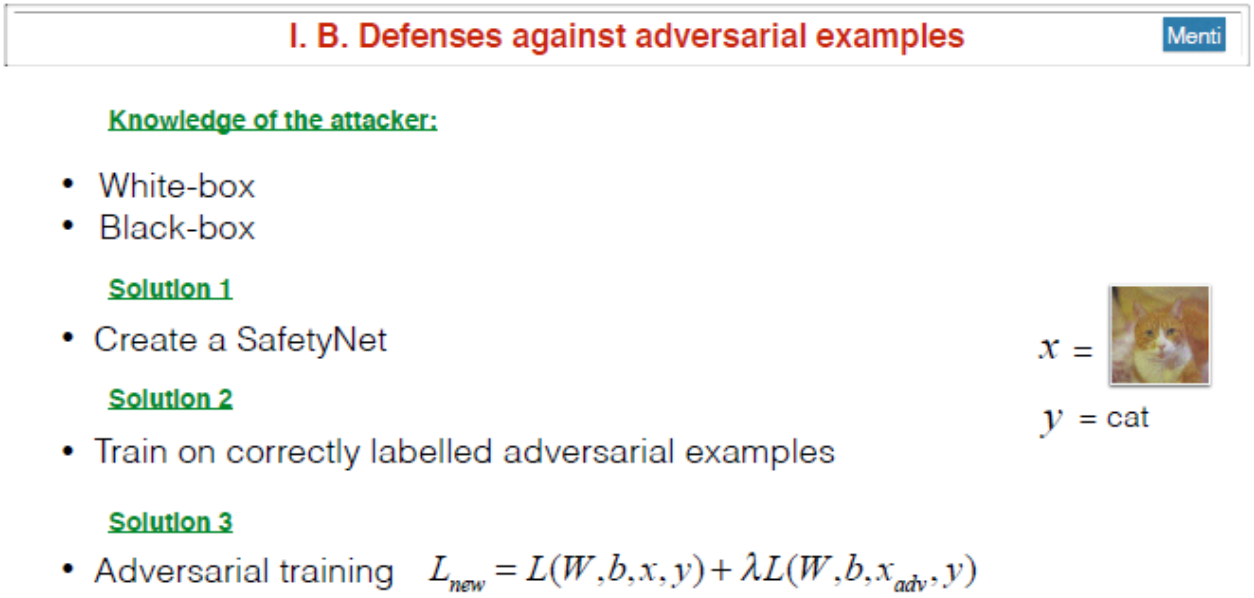
I. A. Attacking a network with adversarial examples

$256^{32\times32\times3} \approx 10^{7400}$



For defending against these adversarial attacks, we need to first warm up with some knowledge. The attacker may have access to the network and its hyper-parameters as well as parameters. So it is a white-box for the attacker. It is easier to perform white-box attack than black-box one. But how to attack in a black-box attack? One solution is that, if we can query the model, we can use numerical gradient, which means we change the picture and we look into the changes in the gradients. If we do this many times, we can understand how the loss works. But what if we cannot query the model? There is a very complex property of the adversarial attacks. They are **highly transferrable**. To this end, if I make my own animal recognition network, then fool it with an adversarial example, it is very likely that the same example work for the black-box network.

One solution to defend against the adversarial attacks is to use a **SafetyNet**. It is a kind of firewall, before any image goes into the actual network, it goes through this firewall layer. But now, the attacker can see this firewall layer as a black-box network to fool again. Yes, it is possible, but now it has two constraints. But maybe if you fool the first one, the second one get fooled as well.
We can also generate adversarial examples and train on them so that they cannot get fooled with similar adversarial examples. But it is super costly and we don't know if we can generalize for the adversarial examples we never saw.
Finally, we can train the model with adversarial examples as well as the actual loss function. Its complexity is still super costly but, at least, it is online.



Previously, researchers thought as we are training our model on true cats, due to high non-linearities of neural networks and overfitting on the training set, the adversarial examples exist. In other words, the network cannot understand what cats are, they only know what we trained on; so cannot generalize well. But it turns out to be wrong. Researchers found out, actually it is the linear part of the network leads to existence of adversarial examples. So, let's use a linear regression for this problem.
For the forward propagation of the linear regression we have: $y_{predicted} = w^T x + b$
So having 6 features, let's for this example say the network has been trained and converged to:

$$w = (1, 3, -1, 2, 2, 3), b = 0, x = [1, -1, 2, 0, 3, -2]$$
$$\Rightarrow y_{predicted} = -4$$

Now, the question is how to change x into x^* such that $y_{predicted}$ changes radically but $x \approx x^*$?

$$\frac{\delta y_{predicted}}{\delta x} = \text{impact of small changes of } x \text{ on } y_{predicted}$$
$$\frac{\delta y_{predicted}}{\delta x} = w^T$$

Now, if $x^* = x + \varepsilon w^T$ where ε is the value of the perturbation, then: $y^*_{predicted} = wx^* = wx + \varepsilon w w^T$
And we know: $ww^T = \| w \|^2$ which is always positive and shows that if I push x a little bit, it will pushes the $y^*_{predicted}$ to a larger value.

So let's say $\varepsilon = 0.2$, then the $x^* = [1.2, -0.4, 1.8, 0.4, 3.4, -1.4]$. Now, $y^*_{predicted} = 0.5$ while it was previously was -4. We just pushed x^* by 0.2 and it pushed the result 4.5.
Insights:

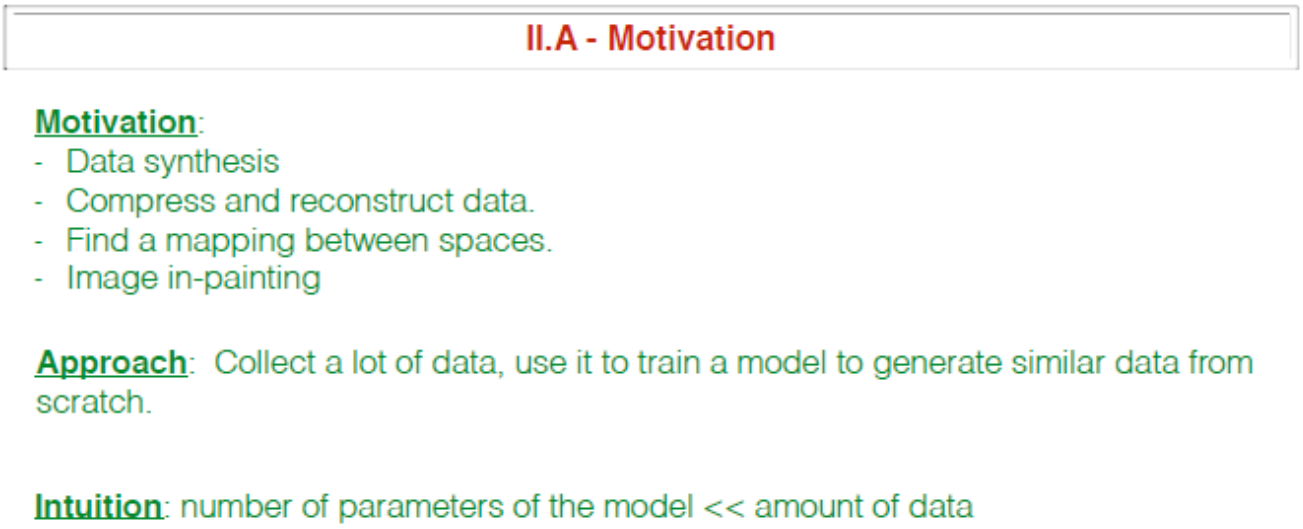
1. if w is large then $x \approx x^*$. So, for pushing it to the positive side, we take the $sign(w)$.
2. As x grows in dimension, the impact of $\varepsilon \times sign(w)$ on $y_{predicted}$ increases.

So the general form of generating adversarial, named "fast gradient sign method" is:

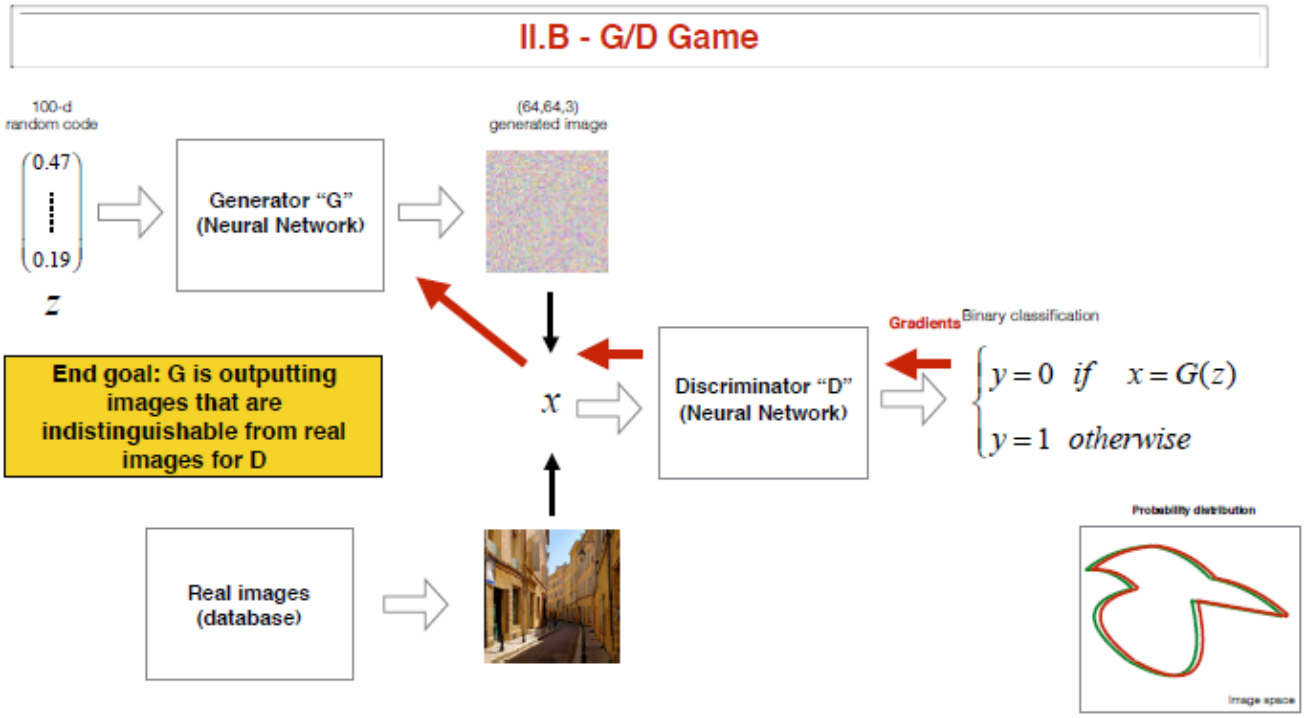
$$x^* = x + \varepsilon \times sign(\nabla_x, J(w, x, y))$$

Generative Adversarial Networks

Let's now go to the Generative Adversarial Networks (GANs).



The goal here is to not give a cat or overfit to the cat, but train the network in a way that it generates a cat that doesn't exist. For doing so, we have a database of real images. The distribution of the real world images are shown in green and the generated images are shown in red. We want the distribution of generated images be look like the real world images.
We define a discriminator "D" whose job is to recognize an image is real or not – a binary classifier. So, it is a simple task to do for the "D" network. We want in the backward propagate gradients to the discriminator in order to train generator "G". If the image was generated, then we have $G(z)$ and we can take the gradient and update weights of the generator, but if the image was real, then we do not have any $G(z)$ to take the gradient and it would be 0.



As for the loss function of the discriminator, we tweak the cross entropy a little bit so that the first term says "D" should correctly label real data as 1 and the second term says "D" should correctly label generated data as 0. So two terms are for two different mini-batches, one on the real images and the other on the "D".

Now for the loss function of the generator, we want "G" to fool "D". So, our whole idea would be whether "G" fools "D" or not. If "D" is bad, doesn't make "G" good. So both should perform well.

II.B - G/D Game

Training procedure, we want to minimize:

Labels: $\begin{cases} y_{real} \text{ is always 1} \\ y_{gen} \text{ is always 0} \end{cases}$

- The cost of the discriminator

$$J^{(D)} = \underbrace{-\frac{1}{m_{real}} \sum_{i=1}^{m_{real}} y_{real}^{(i)} . \log(D(x^{(i)}))}_{\text{cross-entropy 1: "D should correctly label real data as 1"}} \underbrace{-\frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} (1 - y_{gen}^{(i)}) . \log(1 - D(G(z^{(i)})))}_{\text{cross-entropy 2: "D should correctly label generated data as 0"}}$$

- The cost of the generator

$$J^{(G)} = -J^{(D)} = \frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \log(1 - D(G(z^{(i)}))) \quad \text{"G should try to fool D: by minimizing the opposite of what D is trying to minimize"}$$

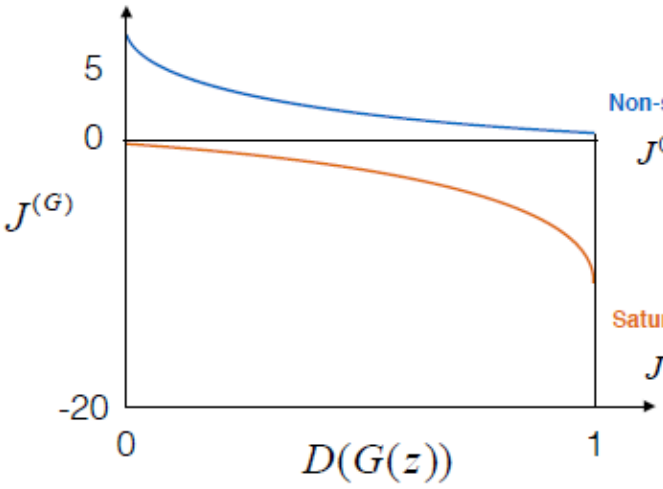
Now, when $D(G(z)) = 1$, then we already fooled the network, but while it is getting closer to fool the network, it goes toward $-\infty$. So a better loss function would be the one with only $\log(D(G(z)))$. Now, the question is, how we should manipulate the loss function above in a way that it uses $\log(D(G(z)))$ instead of $1 - \log(D(G(z)))$? For doing so, we have this formulation:

$$\min \log(1-x) = \max \log(x) = \min(-\log x)$$

II.C - Training GANs

Saturating cost for the generator:

$$\min\left[\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(1-D(G(z^{(i)})))\right] \Leftrightarrow \max\left[\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D(G(z^{(i)})))\right] \Leftrightarrow \min\left[-\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D(G(z^{(i)})))\right]$$



Non-saturating cost

$$J^{(G)} = -\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D(G(z^{(i)})))$$

Saturating cost

$$J^{(G)} = \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(1-D(G(z^{(i)})))$$

[Ian Goodfellow (2014): NIPS Tutorial: GANs]

Kian Katanforoosh

Then we rewrite it as:

II.C - Training GANs

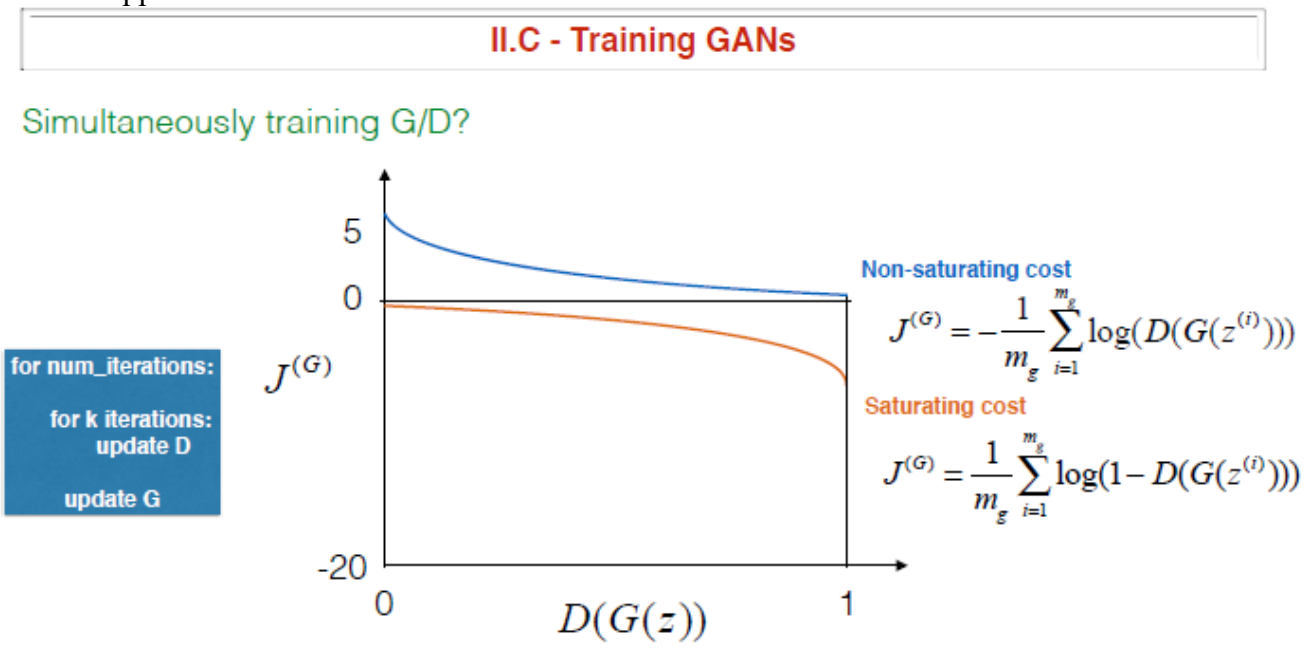
Note that: $\min\left[\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(1-D(G(z^{(i)})))\right] \Leftrightarrow \max\left[\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D(G(z^{(i)})))\right] \Leftrightarrow \min\left[-\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D(G(z^{(i)})))\right]$

New training procedure, we want to minimize:

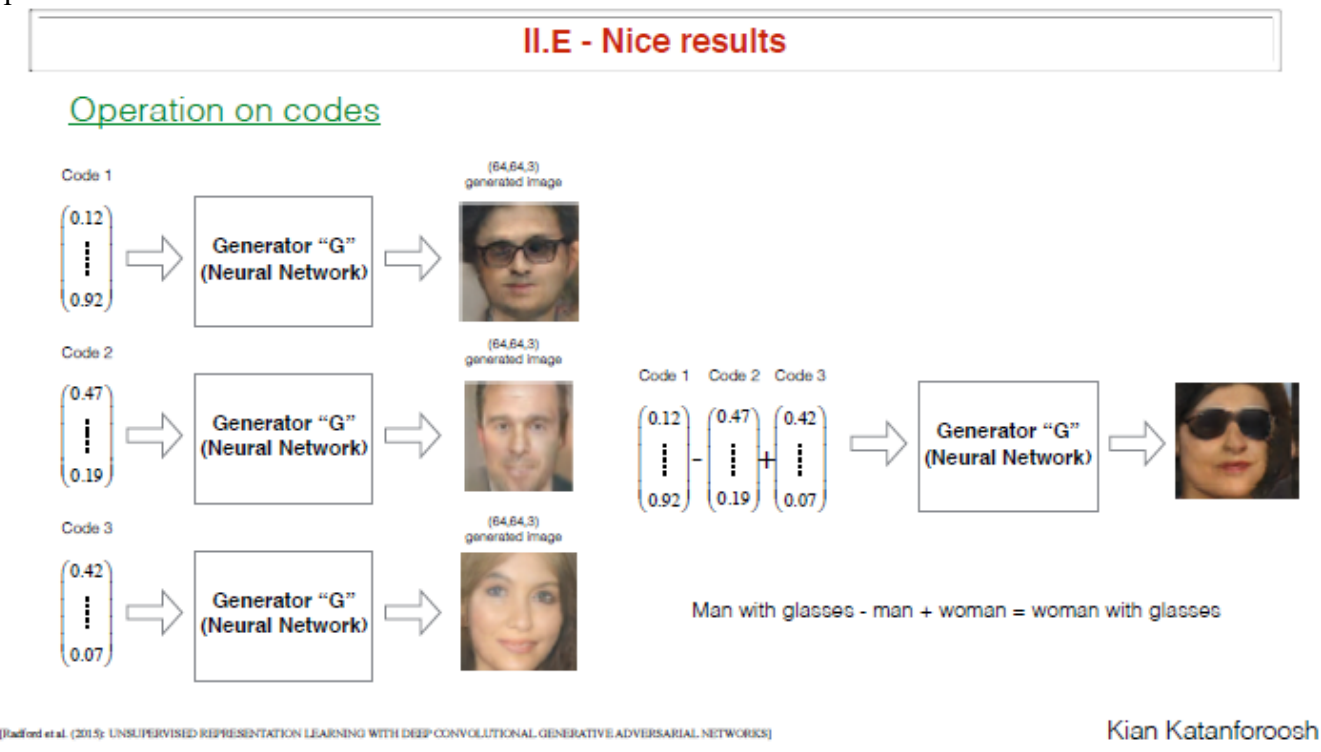
$$J^{(D)} = \underbrace{-\frac{1}{m_{real}}\sum_{i=1}^{m_{real}} y_{real}^{(i)}.\log(D(x^{(i)}))}_{\text{cross-entropy 1: "D should correctly label real data as 1"}} \underbrace{-\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}(1-y_{gen}^{(i)}).\log(1-D(G(z^{(i)})))}_{\text{cross-entropy 2: "D should correctly label generated data as 0"}}$$

$$J^{(G)} = -\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D(G(z^{(i)}))) \quad \text{"G should try to fool D: by minimizing this"}$$

Another trick to train GANs is to use the fact that "D" is usually easier to train than "G". In addition, as "D" improves, "G" can improve and if "D" cannot improve, "G" cannot as well. So the performance of "D" is an upper bound to what "G" can achieve. So we have to train "D" more than "G".



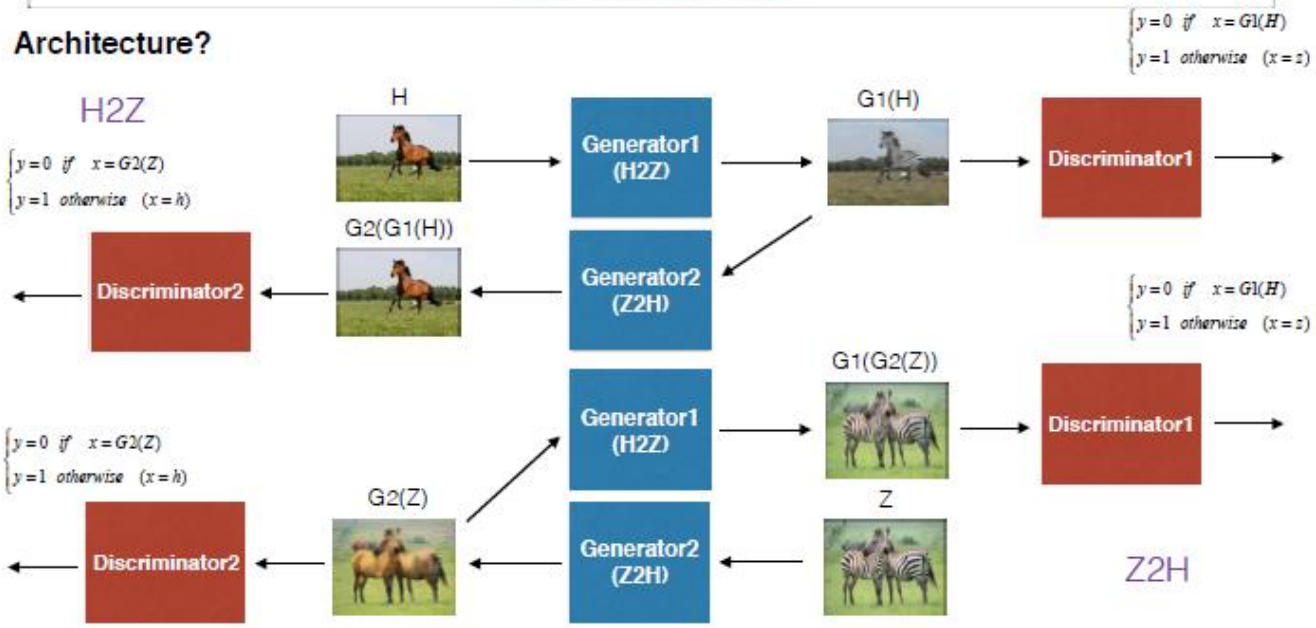
Something funny is that linear operation in latent space of codes have impact directly on the image space.



And let's see a **CycleGAN**s as well:

II.E - Nice results

Architecture?

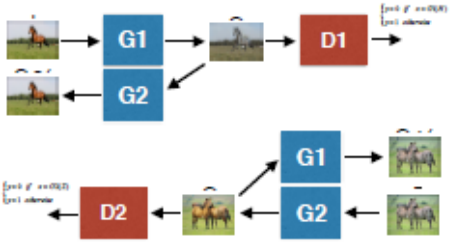


[Zhu, Park et al. (2017): Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks]

Kian Katanforoosh

II.E - Nice results

Loss to minimize?



$$J^{(D1)} = -\frac{1}{m_{real}}\sum_{i=1}^{m_{real}}\log(D1(z^{(i)})) - \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(1-D1(G1(H^{(i)})))$$

$$J^{(G1)} = -\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D1(G1(H^{(i)})))$$

$$J^{(D2)} = -\frac{1}{m_{real}}\sum_{i=1}^{m_{real}}\log(D2(h^{(i)})) - \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(1-D2(G2(Z^{(i)})))$$

$$J^{(G2)} = -\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\log(D2(G2(Z^{(i)})))$$

$$J^{cycle} = \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\| G2(G1(H^{(i)}) - H^{(i)} \|_1 + \frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}}\| G1(G2(Z^{(i)}) - Z^{(i)} \|_1$$

$$J = J^{(D1)} + J^{(G1)} + J^{(D2)} + J^{(G2)} + \lambda J^{cycle}$$

GAN Paper Review

Generative Adversarial Nets

The paper uses two models: I) a generative model G that captures the data distribution and II) a discriminative model D that given the sample, estimates the probability of its origin coming from either the real data or the generative model. The generative model has the goal of generating a data that fools the discriminator so that it finds the data real. In other words, the generative model is similar to a criminal trying to produce fake and undetectable currency while the discriminative model plays the role of the police trying to detect the factitious currency.

This paper mainly focuses on multilayer perceptron (MLP) generative models and discriminative models that the former is manipulated by a random noise. The generator distribution p_g over data x with input noise variables of $p_z(z)$ having a differentiable generative model G with parameters of Θ_g is shown by $G(z;\ \Theta_g)$. With the same settings, we define the differentiable discriminative model D over data x coming is shown by $D(x;\ \Theta_d)$. The whole goal of the discriminator is to maximize the probability of assigning the correct label to samples from G and the real data. It is the same as minimizing the $\log\Big(1 - D\big(G(Z)\big)\Big)$. We can rewrite the D and G assuming it as a two-player minimax game with value function of $V(G, D)$ as:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Minimax Algorithm Review: Minimax algorithm is about maximizing the minimum gain in a game that a player can achieve the maximum value without knowing the actions of the other ones by forcing them to receive the minimum value.

To fully optimize D, either the training computational resource is very high, in the case of the finite datasets leads to overfitting, or both. Consequently, a numerical approach in which alternates between k steps of optimizing D and one step optimizing G would be the solution. In this approach, as far as G's distribution moves in a slow pace enough toward D's distribution without meeting its upper bound but moving very close to it, the optimization computational power is optimized. (Algorithm 1)

We cannot improve G without improving D as it is easy for D to reject G at very early in learning. Having this, as $\log(1 - D\big(G(z)\big))$ saturates, our aim goes toward maximizing $\log(D\big(G(z)\big))$ or minimizing $-\log(D\big(G(z)\big))$. To get a better grasp of it, let's look at the picture below used in Stanford deep learning course.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

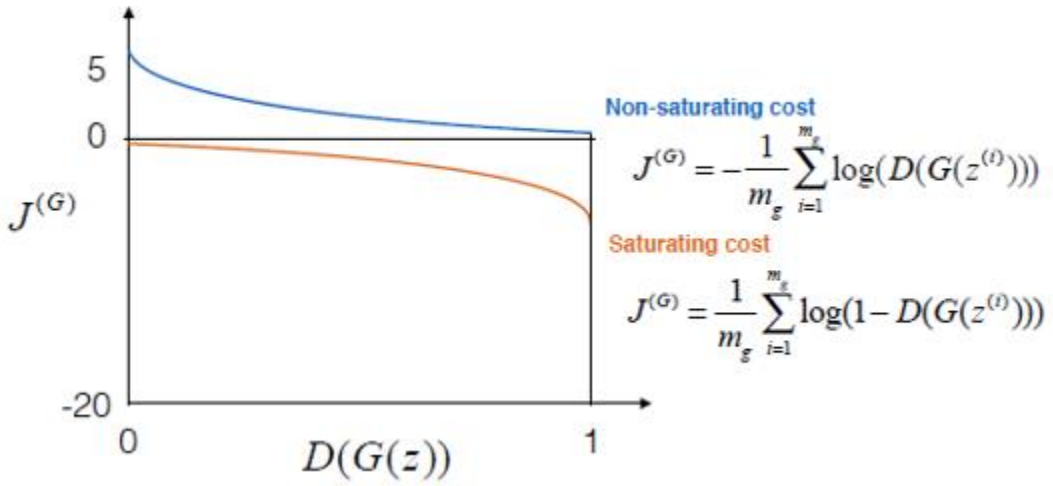
  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.



Having the loss function defined above and given the knowledge that in every iteration we first optimize D based on samples coming from a fixed G distribution, it is easy to achieve the optimal discriminator of D, D^*:

$$V(G,D) = \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x}) \log(D(\boldsymbol{x}))dx + \int_{z} p_{\boldsymbol{z}}(\boldsymbol{z}) \log(1 - D(g(\boldsymbol{z})))dz$$
$$= \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x}) \log(D(\boldsymbol{x})) + p_g(\boldsymbol{x}) \log(1 - D(\boldsymbol{x}))dx$$

Notice that taking integral on noise distribution added to the actual image $p_z(z)$ over z is the same as integral over data generator distribution $p_g(x)$ over x. To put it differently, the area under the curve of the noisy data is the same as area under the curve of the generated data. Moreover, mathematically it is

easy to calculate that for a function: $a\log(y) + b\log(1-y)$ having the result as 1 for real data and 0 for fake data for a fixed y, maximum value achieves at $\frac{a}{a+b}$.

$$\frac{a}{y \times \ln(10)} - \frac{b}{(1-y) \times \ln(10)} = 0$$

$$\frac{a}{y \times \ln(10)} = \frac{b}{(1-y) \times \ln(10)}$$

$$a - ay = by$$

$$y = \frac{a}{a+b}$$

Consequently, we have:

$$D_G^*(\boldsymbol{x}) = \frac{p_{data}(\boldsymbol{x})}{p_{data}(\boldsymbol{x}) + p_g(\boldsymbol{x})}$$

Now, if $p_g = p_{data}$, then $D_G^*(x) = \frac{1}{2}$; so the loss function will converge to $-\log(4)$. If this is true, then by subtracting this value from value function above using KL divergence, we get:

$$C(G) = -\log(4) + KL\left(p_{\text{data}} \middle\| \frac{p_{\text{data}} + p_g}{2}\right) + KL\left(p_g \middle\| \frac{p_{\text{data}} + p_g}{2}\right)$$

$$C(G) = -\log(4) + 2 \cdot JSD\left(p_{\text{data}} \| p_g\right)$$

As JS divergence always is a zero or a positive number, the minimum value achieve when JS divergence part of formula is 0 which is when $p_g = p_{data}$.

Jensen-Shannon divergence review: it is a method of measuring the similarity between two probability distributions using KL divergence with symmetric forms and finite value properties.

Finally we can prove that if G and D have enough capacity, D is allowed to reach its given G, and p_g is updated so as to improve the criterion below, then p_g converges to p_{data}.

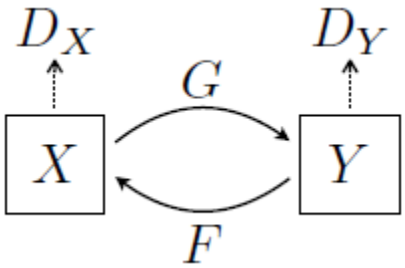
$\mathbb{E}_{\boldsymbol{x}\sim p_{data}}[\log D_G^*(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_g}[\log(1 - D_G^*(\boldsymbol{x}))]$

Cycle-GAN: Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

For many vision tasks, paired training data is very essential but not much available. Consequently, the effort in this paper is building a model that maps $G : X \rightarrow Y$ in a way that the distribution of $G(X)$ for every $x \in X$ is a $\hat{y}$ that has the same distribution of $y \in Y$. To put it differently, the output of $G(X)$ should be distinguishable for a trained adversarial network to classify $\hat{y}$ apart from y. So, the optimal G should translate the domain X to domain $\hat{Y}$ distributed identically to Y. However, based on the adversarial networks, we know there are infinite ways to generate the same $\hat{y}$. In addition, (in my opinion due to the same reason behind transferability of the GAN models) many of the input images $\hat{y}$ map to the same output image $\hat{y}$; so that the optimization fails to progress.

This problem urges the authors to come up with the idea of using a model which is mathematically invertible, however visible it is not for the deep model. For clarification, we can assume the example of translating a sentence from English to French. It is only the best translation when the inverse of it, translation from French to English gives exactly the same English sentence inputted to the model. The same way, if we input image distribution into a model, $G : X \rightarrow Y$, the output has the best distribution and has not the both of the mentioned problems above, when $F : Y \rightarrow X$ is true. It's been called *cycle consistency loss* that encourages $F(G(x)) \approx G(F(y))$ and $G(F(y)) \approx y$. Combining these two loss functions, can gives the full unpaired image-to-image model translation.

For building this model, which $x \sim p_{data}(x)$ and $y \sim p_{data}(y)$, two adversarial discriminators of D_X and D_Y exists that the former aims to distinguish between image x and translated image $F(y)$ and the latter aims to distinguish between y and $G(x)$. To put it differently, the objective is for the first adversarial loss to match the distribution of the generated image and data distribution ($G(F(y)) \approx y$), and the second adversarial one, namely *cycle consistency loss*, to ensure the functions F and G are not contradicting each other ($F(G(x)) \approx G(F(y))$).

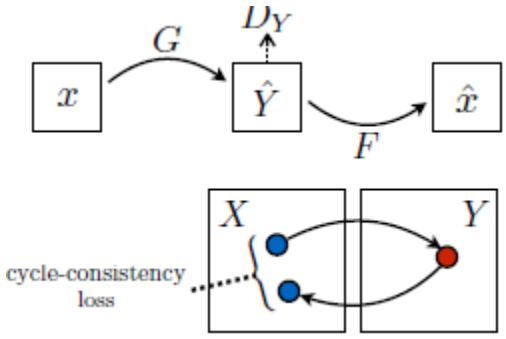


This model comes with the objective function shown below where G tries to generate images having the same distribution as Y and D_Y aims to distinguish between translated samples $G(x)$ and real samples y:

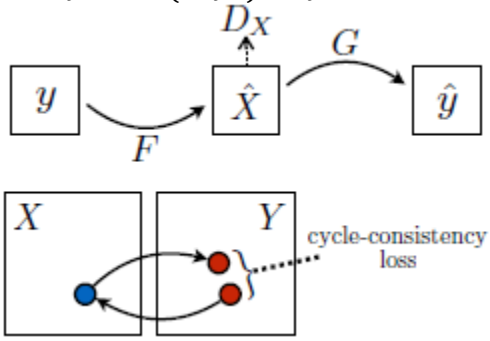
$$\mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log(1 - D_Y(G(x))]$$

So G tries to maximize the similarity of generated image distribution with Y and D tries to minimize the probability of getting fooled by G. To put it differently, for $G : X \rightarrow Y$, G aims to minimize the objective above against and adversary D that tries to maximize it. So it is the same min-max game we saw in GAN models: $\min_G max_{D_Y} \zeta_{GAN}(G, G_Y, X, Y)$. The same way for $F : Y \rightarrow X$ the D_X aims to $\min_F max_{D_X} \zeta_{GAN}(F, D_X, Y, X)$.

Now as for the cycle-consistent loss, input image x should be translated back to the original image x. So the conversion would be $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$. This is the *forward cycle consistency*:



The same way image y should be translated back to the original image y. This is *backward cycle consistency* which its job is to $y \rightarrow F(y) \rightarrow G\big(F(y)\big) \approx y$.



This whole cycle consistency loss can be formulized as:

$$\begin{aligned}\mathcal{L}_{\text{cyc}}(G,F) = &\mathbb{E}_{x\sim p_{\text{data}}(x)}[\|F(G(x)) - x\|_1] \\ &+ \mathbb{E}_{y\sim p_{\text{data}}(y)}[\|G(F(y)) - y\|_1].\end{aligned}$$

Now the whole objective is:

$$\begin{aligned}\mathcal{L}(G,F,D_X,D_Y) = &\mathcal{L}_{\text{GAN}}(G,D_Y,X,Y) \\ &+ \mathcal{L}_{\text{GAN}}(F,D_X,Y,X) \\ &+ \lambda\mathcal{L}_{\text{cyc}}(G,F),\end{aligned}$$

where λ controls the relative importance of the two objectives. So the final goal of the model gets back to the two min-max problems and for this reason the optimal G^* and F^* can be found as:

$$G^*, F^* = \arg\min_{G,F}\max_{D_x,D_Y}\mathcal{L}(G,F,D_X,D_Y)$$

This model can be seen as training of two auto-encoders, each with its own architecture.

Course 2

Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

Week 1

Practical aspects of Deep Learning

Train / Dev / Test sets

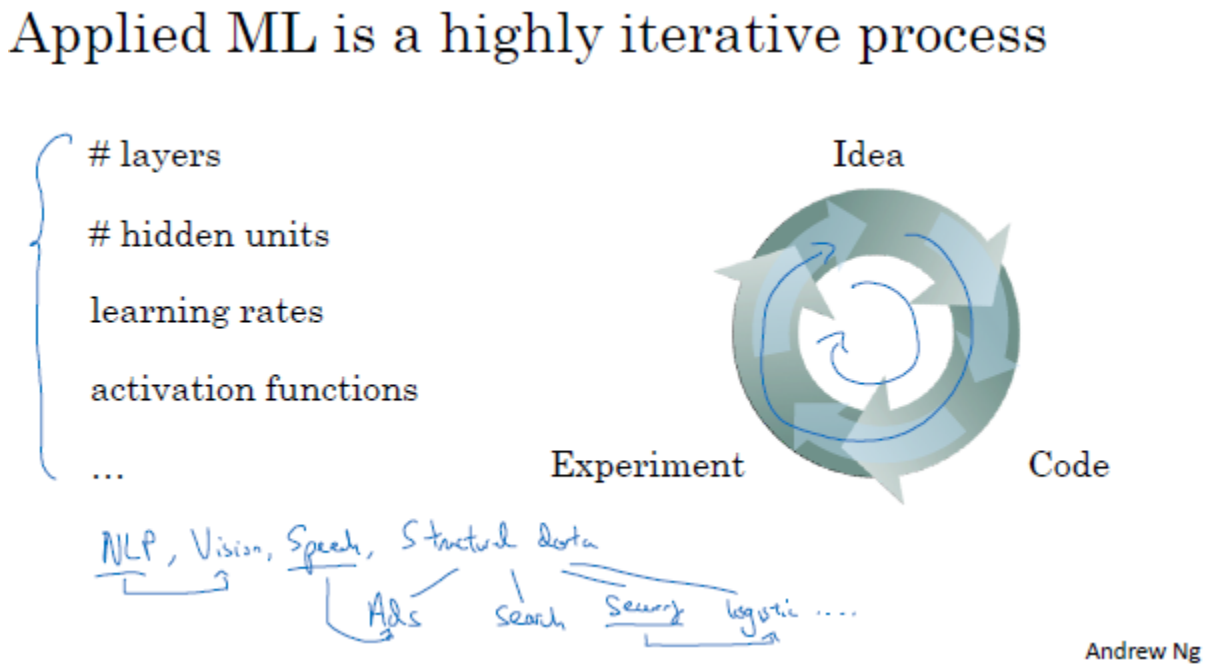
Some of the hyper-parameters For NNs are:

1. # layers
2. # hidden units
3. Value of the learning rate
4. Choice of the activation functions
5. ...

In practical, applied ML is a highly iterative process, so we start with:

1. Idea → number of hidden units, number of layers, activation functions, and so forth
2. Code → we program it and we run it
3. Experiment → we get the results of the experiments and based on the outcome we refine the idea → getting back to step 1

Finding the best choice of network and hyper-parameters is impossible unless test it over and over again



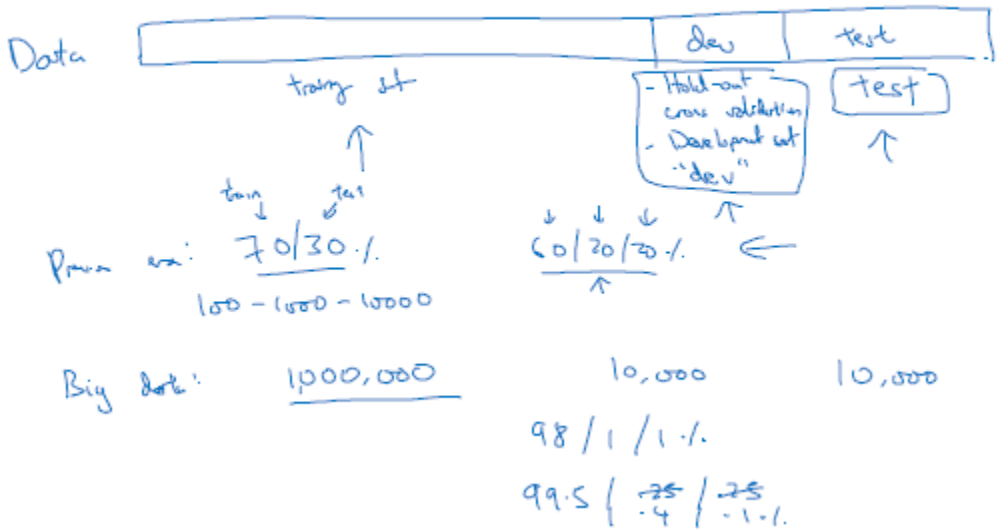
To increase the pace of iterating, we divide the dataset into train, dev, and test sets.

A data should have:
Training Set + Hold-out Cross Validation or Development Set + Test Set

So we run the learning algorithms on the training set and then test it with dev set to find out the best model for the problem. Finally, after finding the best model, we evaluate it with the test set.

Previously: 70/30 train and test sets without dev set or 60/20/20 train/dev/test sets
Modern big data era: dev and test should be so much smaller in percentage since dev is only for testing the models and finding the best and for test set it is only about validating how well the model is doing. So we allocate the most of the data to the training set so that it can generalize better.

Train/dev/test sets



There is a concept about mismatched train and test distributions. For example, we may get cat pictures from web with high resolutions for variety of cat models. Then, our dev and test sets contain user images with low resolution and limited types of cats. So, they are not from the same distributions. What may happen here is that even if our model learns the best on the training set, it cannot perform well on dev and test sets because the type of data is not uniform.
So, we have to make sure the dev/test set and training set are from the same distribution

Not having test set might be okay → dev set should be there at any event → evaluation on dev set is mandatory to find the good model
Though in this setting, the result is not unbiased estimate of performance → so it is better to have test set as well
In this setting, sometimes people call dev set as test set.

Mismatched train/test distribution

Cats

Training set:
Cat pictures from webpages

Dev/test sets:
Cat pictures from users using your app

→ Make sure dev and test come from same distribution.

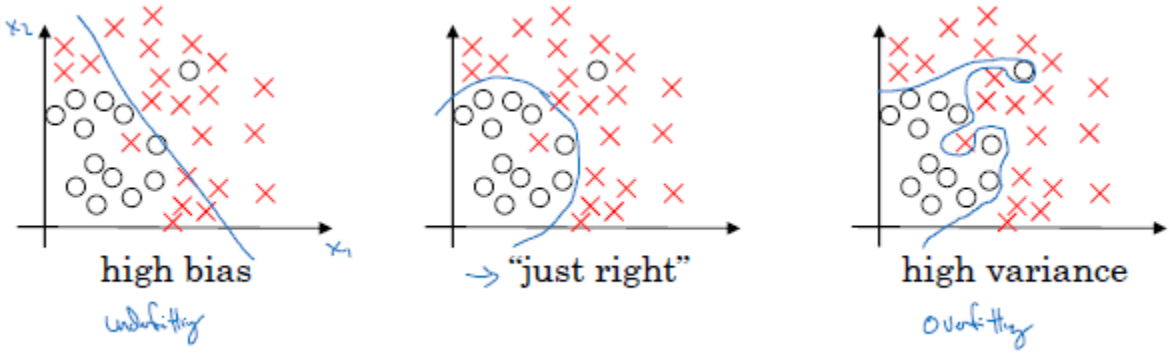
"test"
train / dev
train / test
→ train / dev

Not having a test set might be okay. (Only dev set.)

Bias / Variance

There is not much about bias/variance trade-off in deep learning.

The leftmost image draws a line as the classifier, so it has no flexibility →high bias → underfitting
The rightmost image is too flexible and is not good for generalization →high variance → overfitting
The middle one is in between of previous two cases so it is right.



Let's see some examples to grasp bias and variance concept better. We use the assumption that human can do the same task with around 0% error.

1. High variance
    a. Training error → 1% → very well
    b. Dev error → 11% → relatively poorly
        i. The model is probably overfitting
2. High bias
    a. Training error → 15% → not fitting
    b. Dev error → 16% → very well
        i. The model is probably underfitting
3. High bias and high variance
    a. Training error → 15% → not fitting
    b. Dev error → 30% → poor
        i. Worst of two worlds
4. Low bias and low variance
    a. Training error → 0.5% → very well fitting
    b. Dev error → 1% → very well model
        i. Best of two worlds

We started with the assumption that human level error or even better, the optimal Bayes error, is 0%. If it was 15% we could have said the 2nd model is the best. Then the 4th model is not acceptable because we cannot outperform the optimal Bayes error.

Rule of thumb → compare the error difference of the Train set and Dev set

Bias and Variance

Cat classification


Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high variance	low bias low variance

Human: ~0%

Optimal (Bayes) error: ~~0%~~ 15%

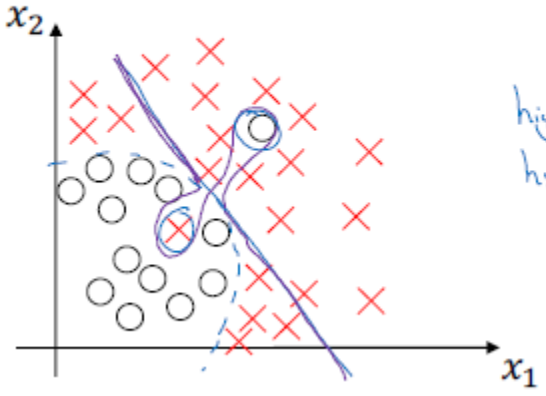
Blurry images

What is High Bias and High Variance look like?

Not fitting the quadratic shape and with too much flexibility in middle or some parts of it

⇨ High bias at some regions and high variance at some other regions.

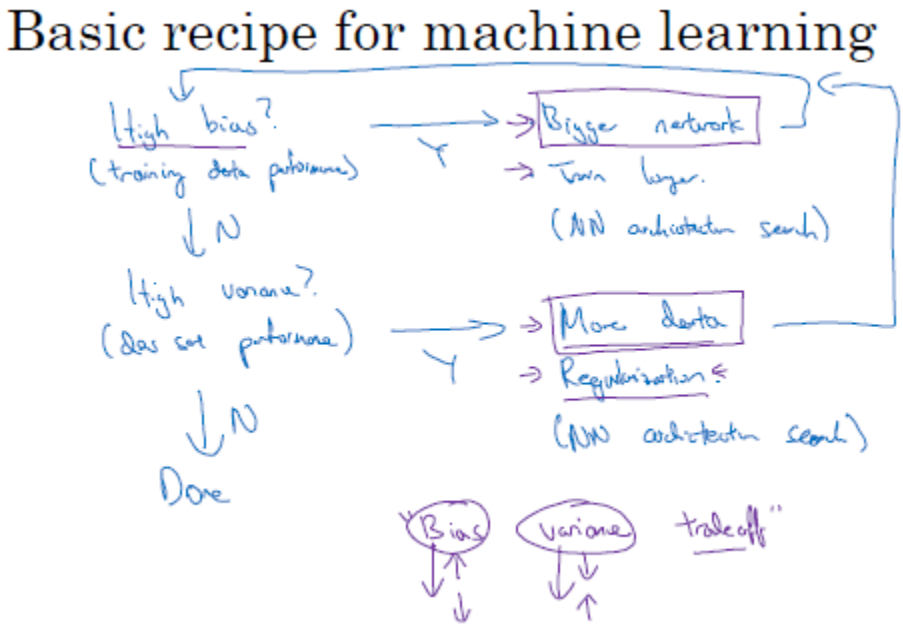
High bias and high variance



Basic Recipe for Machine Learning

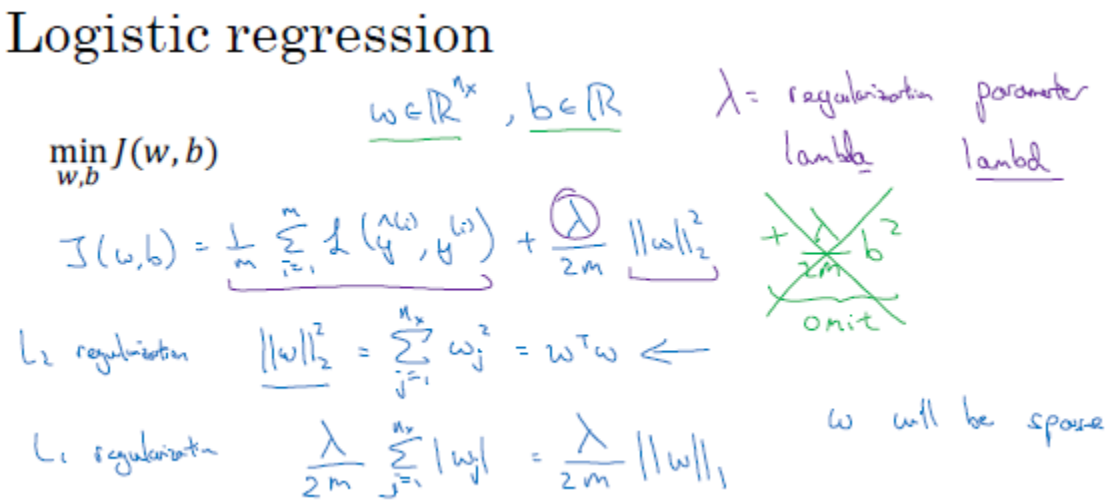
1. Is the algorithm **high bias**? (we look at training data performance)
    a. Yes?
        i. Bigger Network
        ii. Different NN architecture may help
        iii. Running the model longer
    b. No?
        i. Go to next step
2. Is the model high variance? (we look at the error difference of training and dev set)
    a. Yes?
        i. Getting more data
        ii. Trying Regularization
        iii. Another NN architecture
            1. Going back to step 1
    b. No?
        i. We are done!

With bigger network and more data we can reduce bias and variance without hurting the other one.

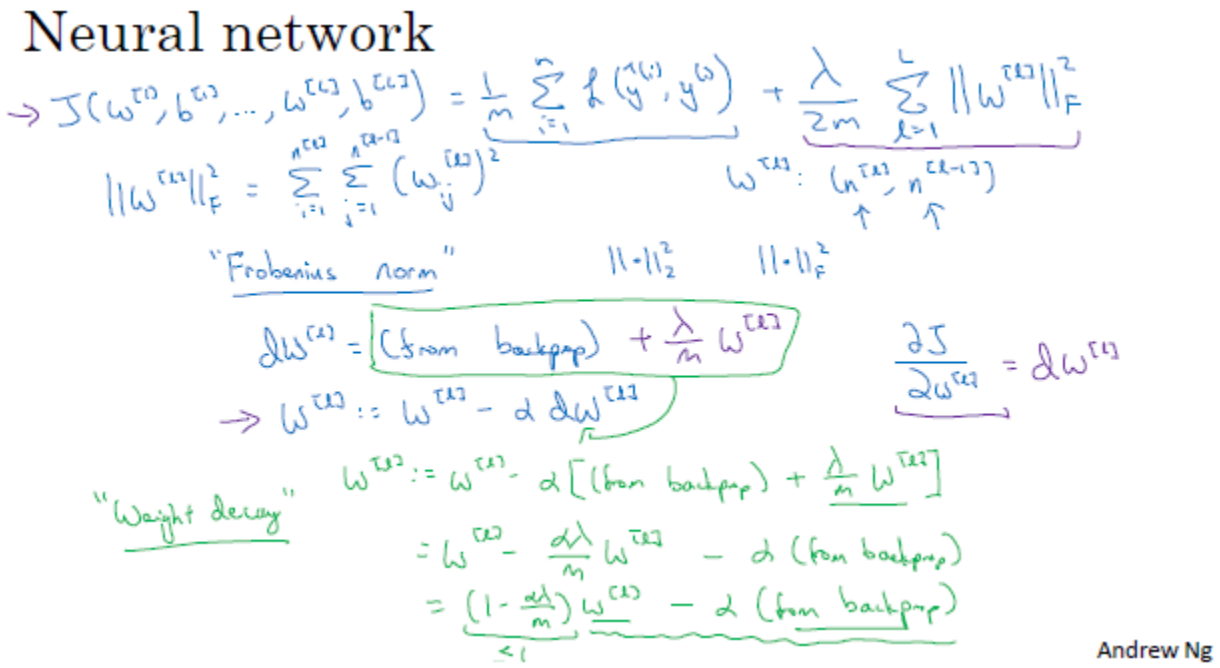


Regularization

When we have a high variance, we first going to use a regularization, if no more data is available
We can regularize "bias" value as well, but as it is only a single number, it does not change anything much so we can easily omit it. However, "weights" are pretty high dimensional data.
L2 regularizer is the most well-known regularizer, but we also can use L1 regularizer.
L1 regularizer causes the "weights" to become sparse with many zeros → Can be used in compressing the model because of too many zeros
We also use a λ hyper-parameter along with L1 and L2 regularizers.



Now for the Neural Network, the concept is still the same. Note that, the weight update rule changes as we add regularization term since its derivative also comes in the formula. L2 regularizer sometimes being called "weight decay" as it makes the "weight" update gets a little smaller by multiplying by a number less than 1.



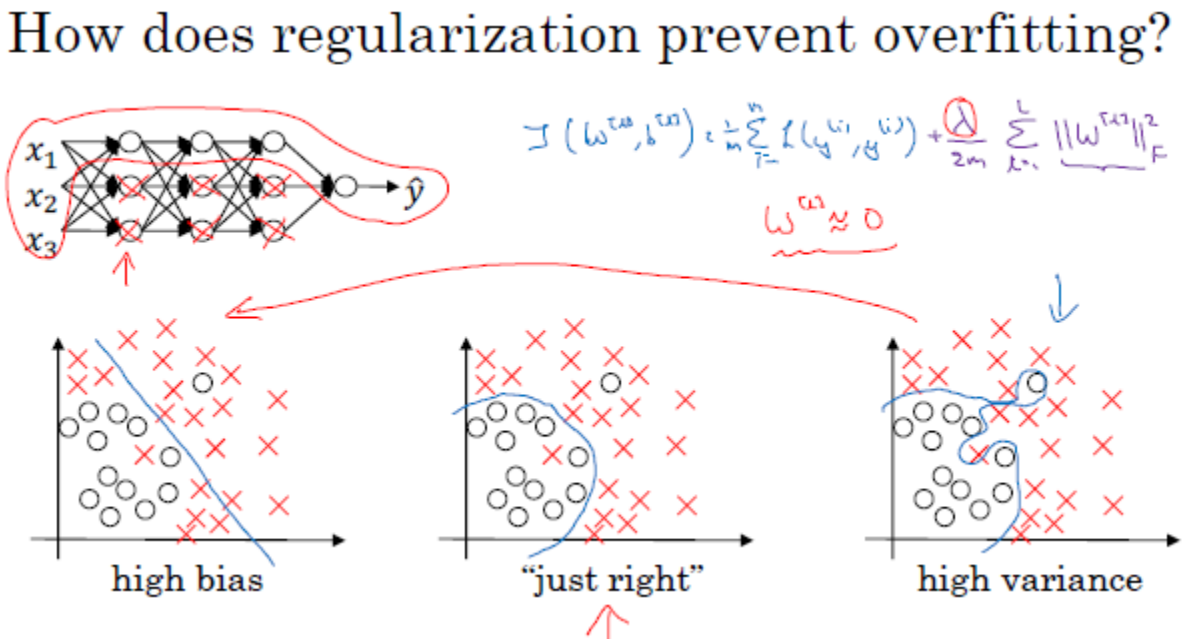
Andrew Ng

Additional Readings:
https://www.quora.com/What-is-the-difference-between-L1-and-L2-regularization-How-does-it-solve-the-problem-of-overfitting-Which-regularizer-to-use-and-when

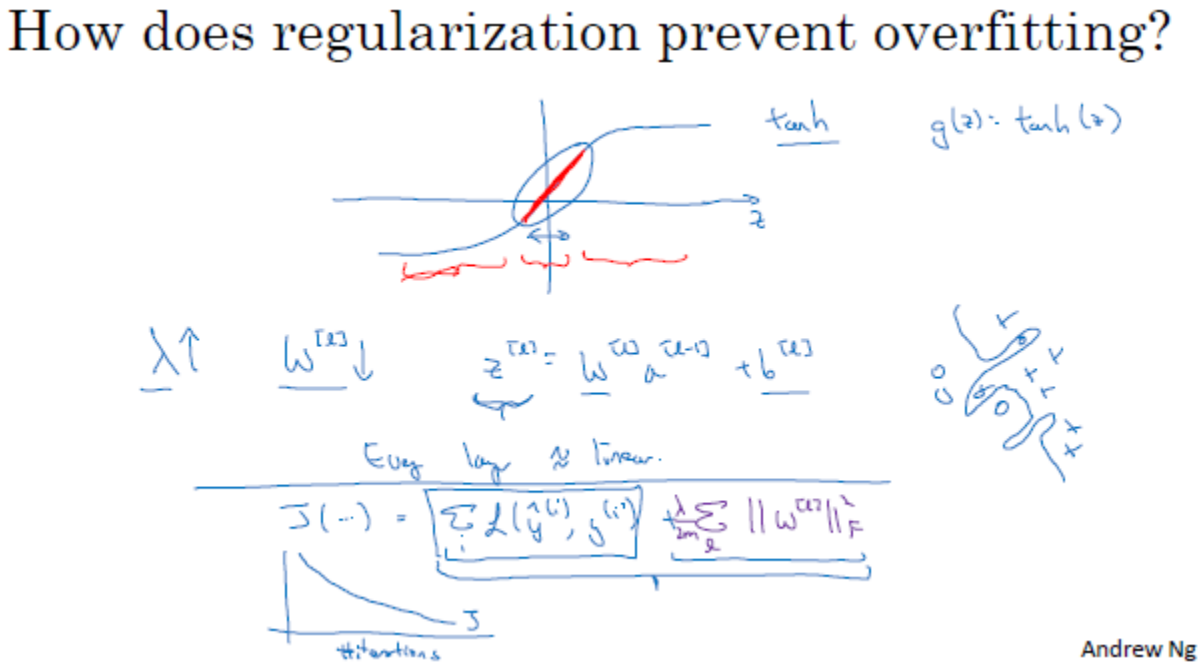
https://www.quora.com/What-are-the-Frobenius-L2-and-Euclidian-norms-How-are-they-different-from-one-another

Why regularization reduces overfitting?

One intuition for using regularization is that it zeroing out many hidden units and make it so much simpler so it goes from high variance to lower variance. However, it is not quit right. It actually reduces the impact of some of these hidden units and prevents from overfitting.

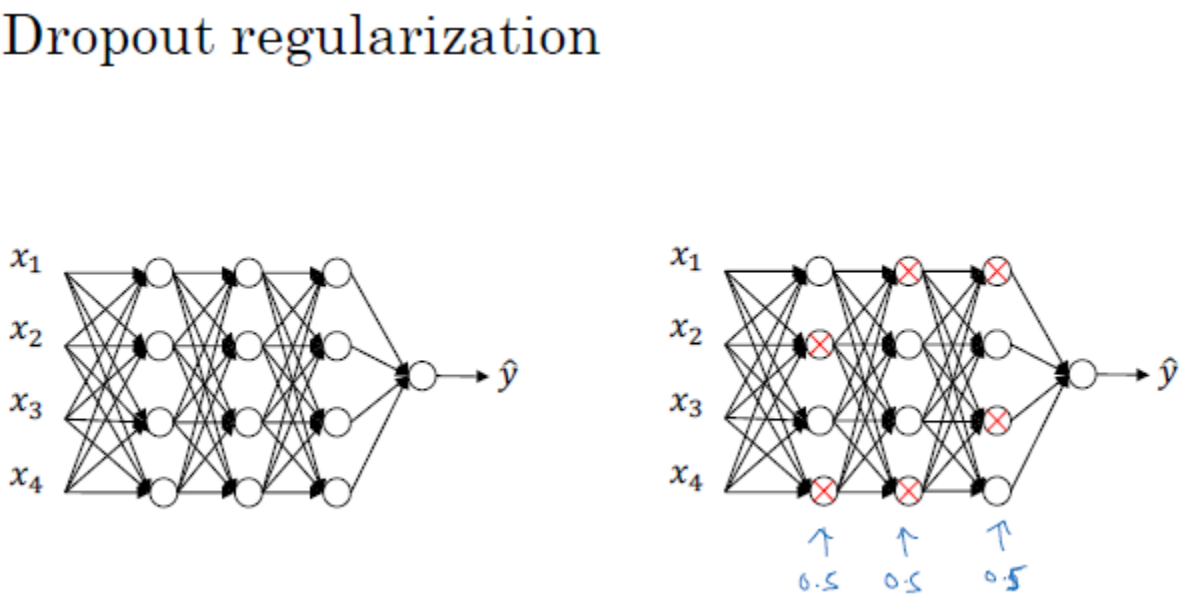


For another intuition we use tanh. In tanh activation function, with a large value of λ, we are going to have smaller weights. With smaller weights, the "Z" gets smaller as well. Now, small "Z" values have G(Z) close to 0, the part that would be roughly linear and as we go far from the zero it does not grow very fast. So it is calculating a linear like function which is so simple and less prone to overfitting.

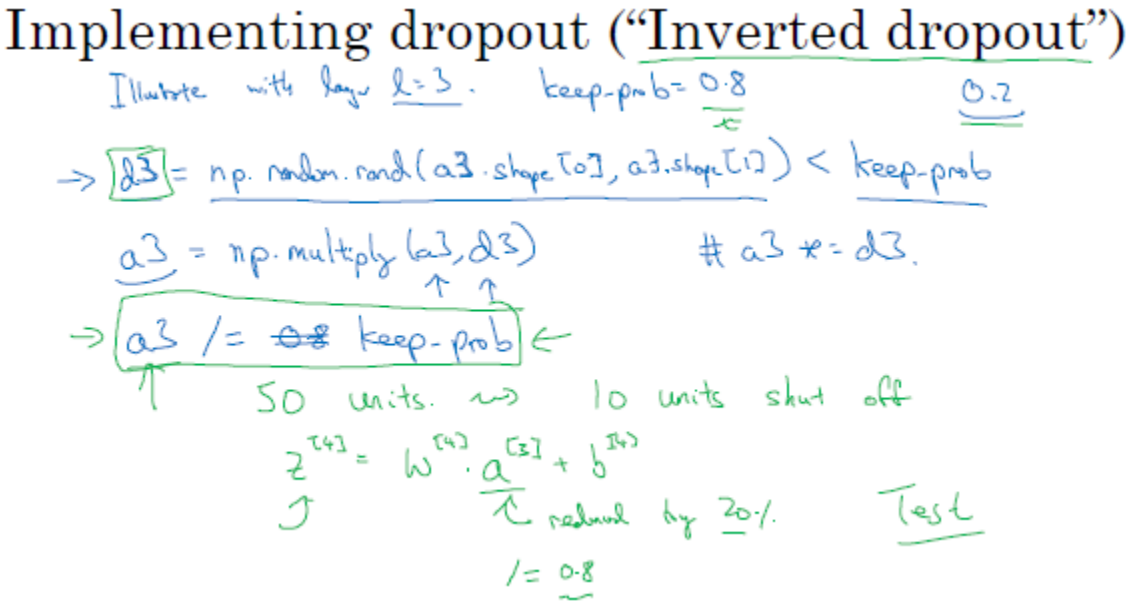


Dropout Regularization

In the dropout, we give a probability to remove or keep some nodes in NN → we end up with much smaller and diminished network. We then do it over and over again to create other diminished networks.



A concrete approach of dropout implementation is called "inverted dropout". We define a "keep-prob" as a value which is the chance of keeping a node. We randomly initialize a matrix for each layer which is either 0 for eliminating a node or 1 for keeping the node. Then we multiply this matrix with the activation matrix. Then we need to scale up the result by "keep-prob" to keep the expected value of the matrix as before. The last step is for "inverted dropout" which helps us a lot in the test time.



In the test time, we never use dropout. Otherwise we need to run dropout many times. If we don't use inverted dropout, we need to scale down the results we get in the test time for each epoch which is much harder.

Making predictions at test time

$a^{[0]} = X$

No drop out.

$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$

$a^{[1]} = g^{[1]}(z^{[1]})$

$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$

$a^{[2]} = \ldots$

$\downarrow$

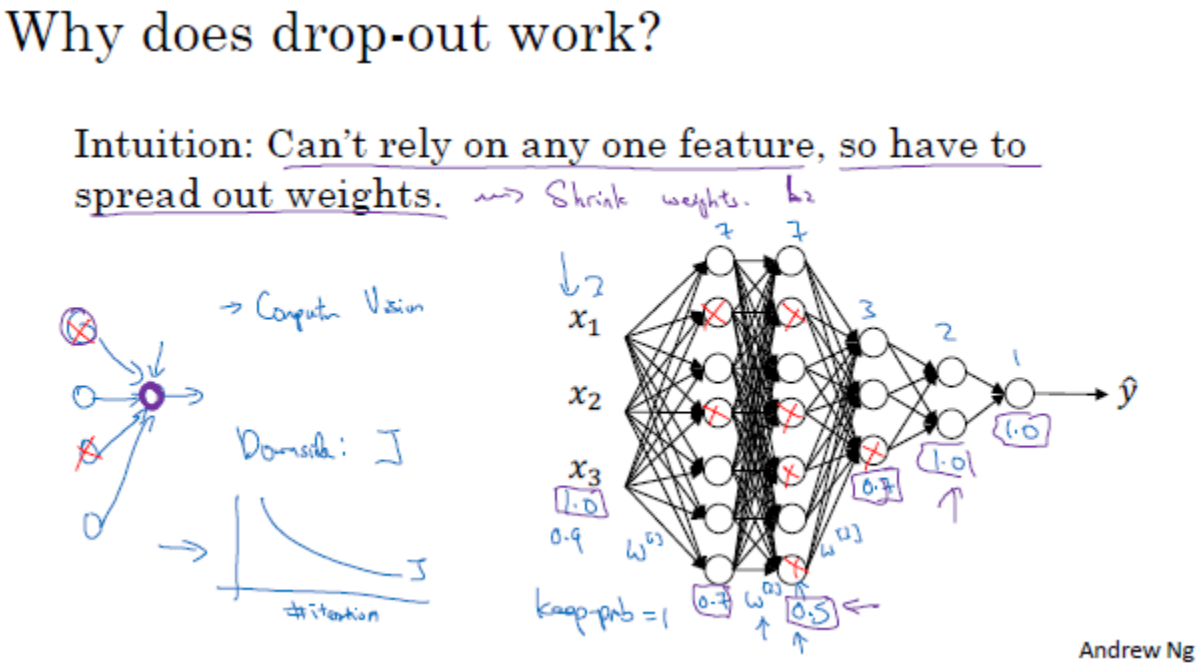
$\hat{y}$

/= keep-prob

Understanding Dropout

The intuition is that we can't rely on any one of the features as it might fade away, so have to spread out weights. It is like shrinking the weights as we previously did with L2 regularizer.
One note is that, we can choose different keep-prob for each layer. Mainly we use lower keep-prob values for those layers with many units and parameters that may cause the network to overfit.
For example, for layers having nodes more than 7×7, it is more likely that this matrix tend to overfit, so we can dropout more here but in somewhere else with less possiblity of overfitting we can elminate less units.
Note that for the input and output layers it is better to keep the keep-prob as 1.
We mainly use the dropout in computer vision as we have many layers in this application.

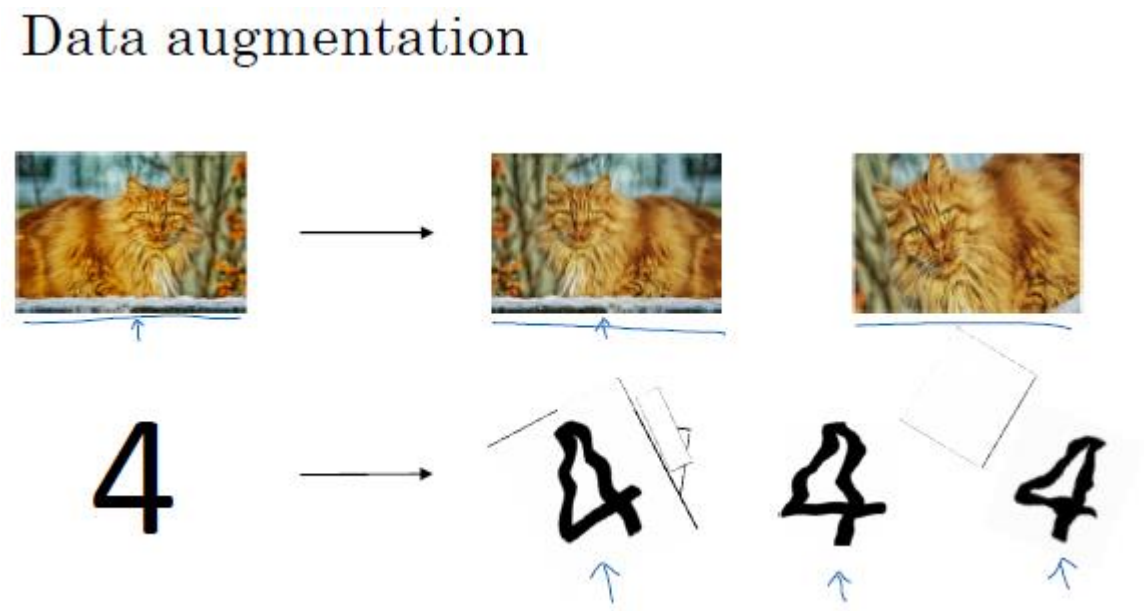
The downside of the dropout is that the cost function is no longer well-defined as we are eliminating some nodes randomly at each iteration. So for making sure that the cost function monotonically decreases we have to turn off the dropout.



Other regularization methods

Data augmentation: We may use dropout or any kind of regularizer as we do not have data to enlarge training set. In this case, we can use data augmentation to gain more data. Although the new data set is not as good as having brand new independent examples.

1. flipping a picture → gaining 2X samples
2. Zooming into part of the image
3. Adding distortion is also practical

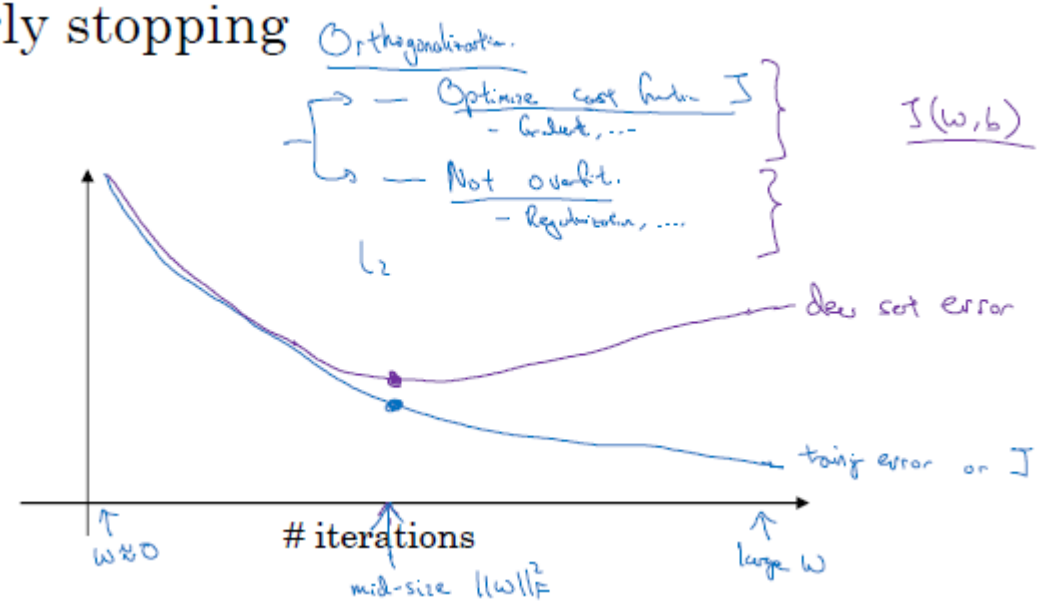


Early stopping: As iterations go on, the training error and cost function decreases, but for dev set error at some point starts to increase. At very first the weights are close to zero and very small. As we train, the weights get bigger and bigger. So, similar to regularization, the early stopping keep the weights at the mid size. like a L2 that scale the values, it keeps the early values of W.

Its downside:
We want to optimize the cost function with an optimizer like Adam or gradient descent. We also don't want to overfit by using a technique like regularizer. So they are many hyper-parameters to tune and each one of them are for independent tasks, namely orthogonalizing. Early stopping bounds these tasks into one so we cannot understand exactly which of the chosen parameters are not quit right.

Its benefit:
It reduces the search space because we don't need to check all the possible hyper-parameter values. Moreover, by running it once, we check weights with small, mid, and large size.

Early stopping

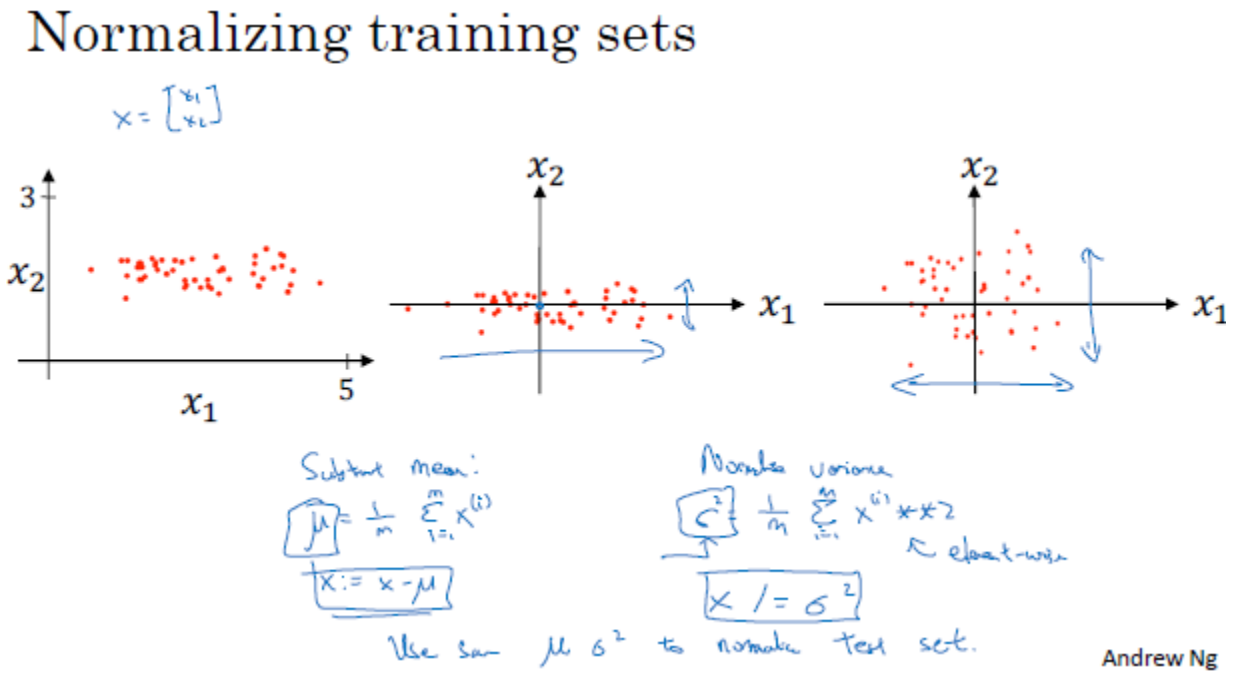


Normalizing inputs

One technique to increase the pace of the iteration while learning algorithm is working is to normalize the input data. For normalizing input data, we have to take 2 steps:

1- zeroing out the mean → subtracting data from the mean value of the data
2- normalizing the variance → so that data concentrated around the 0,0

Note that we use the same variance and mean for the test set as well.

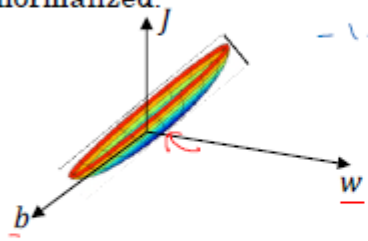


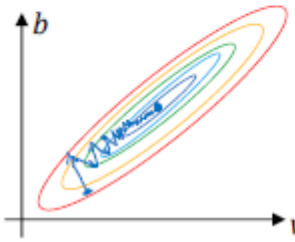
Now the question is, how does it affect the training pace. Non-normalized data is elongated and normalized is more symmetric. For the gradient descent in the former case takes small steps because need many steps to reach the minimum but in the latter case larger steps works very well and learning algorithm runs faster. So we aim to bring them to similar scale.

Why normalize inputs?

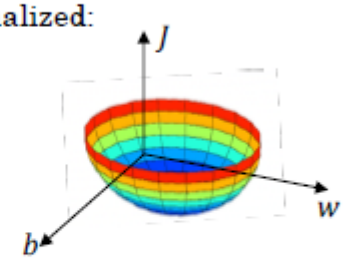
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

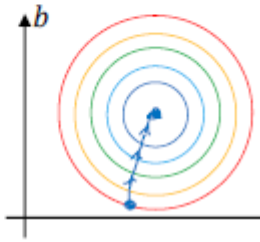
Unnormalized:





Normalized:

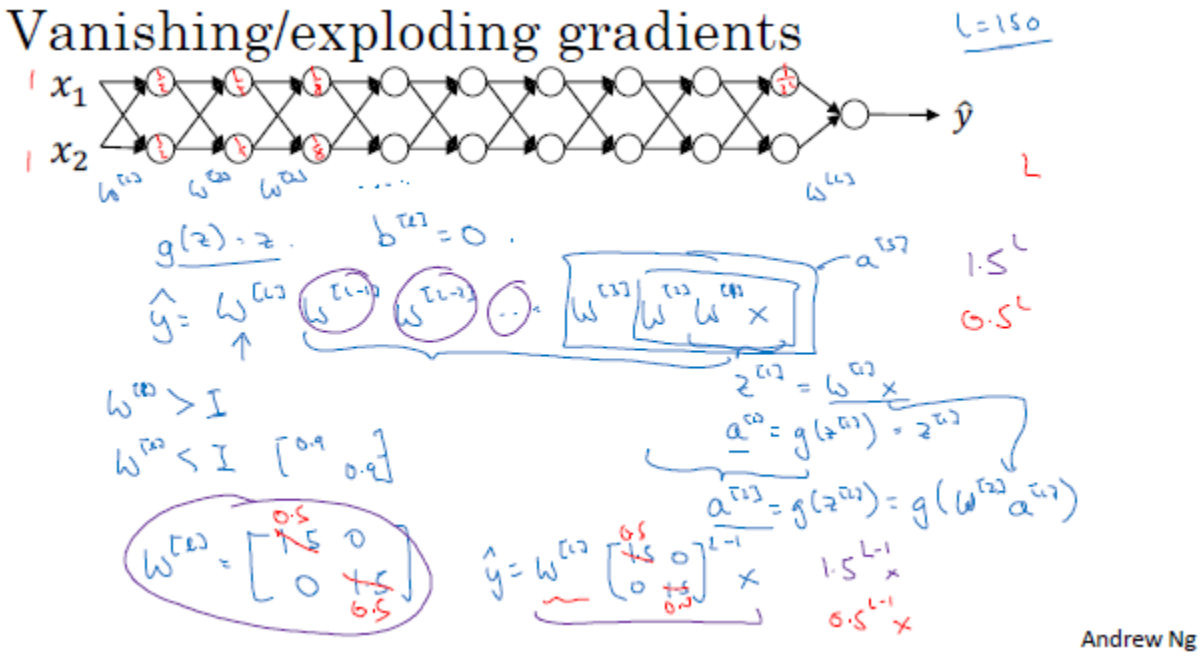




Andrew Ng

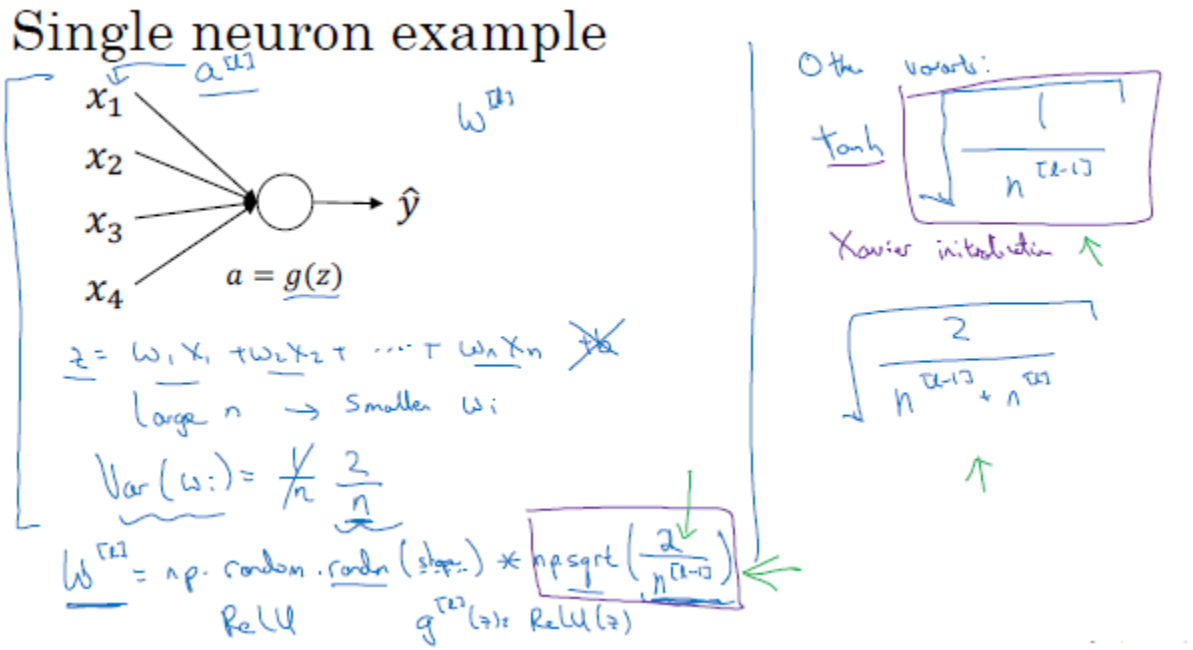
Vanishing / Exploding gradients

A very huge problem in very deep networks is vanishing and exploding gradients. Let's see its math below to understand how vanishing and exploding gradients is happening. Now, starting with a weight value larger than 1, either using an activation function or assuming it linear, in the long run, they the value larger than 1 goes to the power of a big value which ends up with a very huge value. In the contrast, using weight with value less than 1, when it goes to the power of a large value, it ends up with a very small value, very close to 0. Specially with very small values, the optimizer should take very small steps which reduces the pace of training.



Weight Initialization for Deep Networks

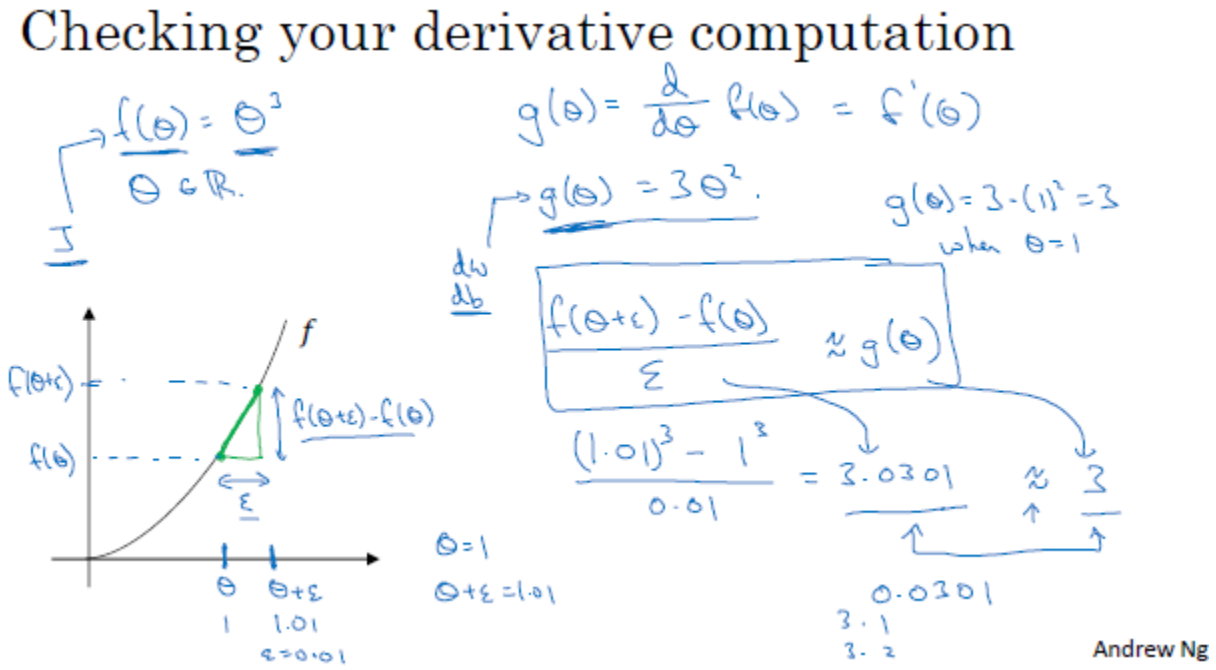
Initializing the weights showed that helps a lot in preventing vanishing and exploding gradients. For weight initialization, we see that as the number of features increase, we need smaller weights because as they are summing up, it doesn't cause explosion of gradient. So, we set the variance of the weight equal to 1 or 2 over number of features at each layer. We can see different initialization methods for different activation functions.



Numerical approximation of gradients

When we implement back propagation, we need to make sure the implementation works fine. To this end, we use a tool, named "numerical checking". Let's see how it works.

First of all, for taking derivative of a function, we define a very small value, ε, in which while we are calculating the slope of the function, we push the ε to 0 that is the mathematical definition of derivative. Let's see its result:

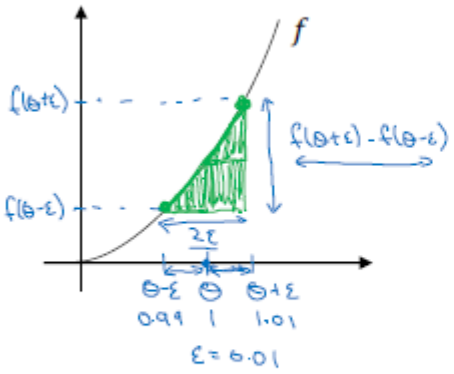


Now we use ε both on right and left side of the actual θ. By doing so, we make a bigger triangle. Now if take the derivative for $\theta \pm \varepsilon$ equation, we should get a value very closer to actual derivative of function than the one side we saw before. Although it is much more accurate, but twice slower than the previous one.

We can see accuracy of using two side difference is in the order of $O(\varepsilon^2)$ while the one side difference gives the order of $O(\varepsilon)$.

Checking your derivative computation

$f(\theta) = \theta^3$



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

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

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

Approx error: 0.0001

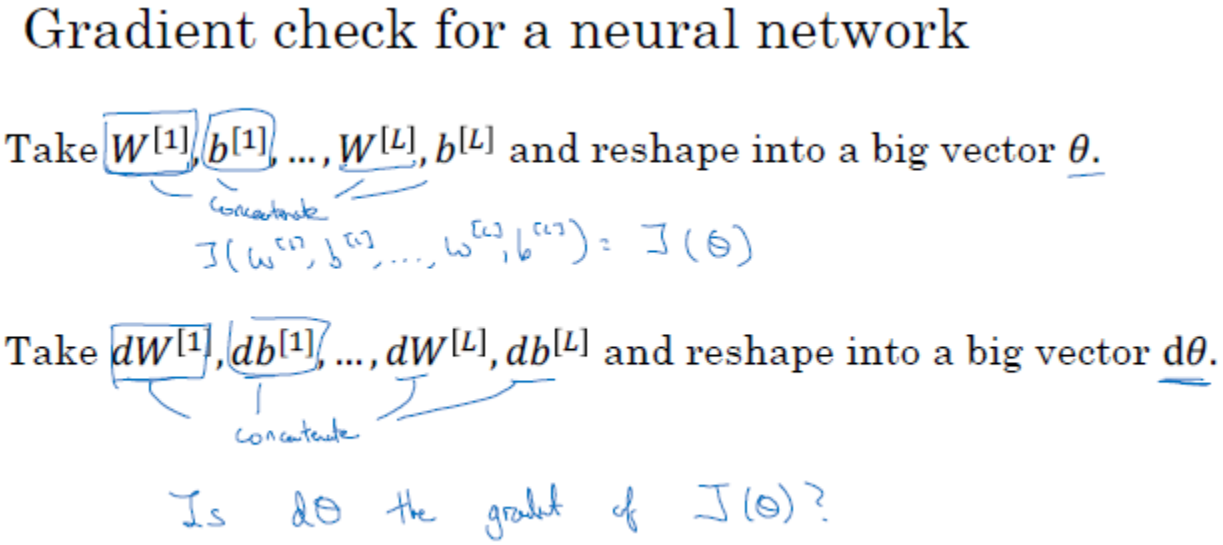
(prev slide: 3.0301. error: 0.03)

$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$ $O(\varepsilon^2)$ 0.01 0.0001 | $\frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$ error: $O(\varepsilon)$ 0.01

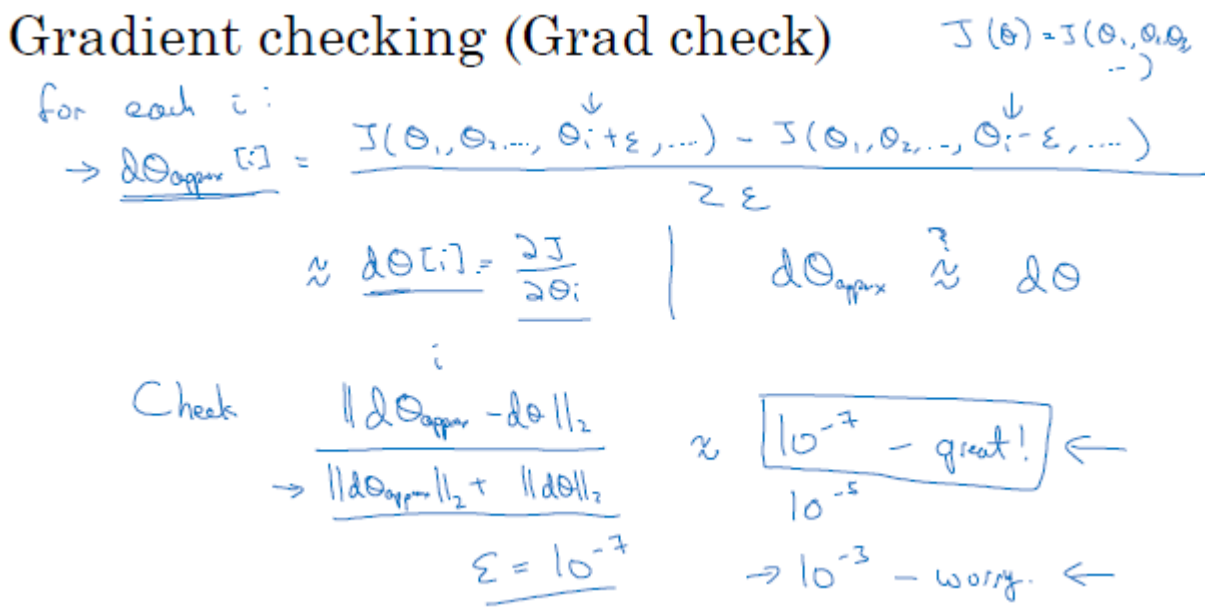
Andrew Ng

Gradient checking

We use gradient checking to make sure our back-propagation implementation is right. For using it first we concatenate all the weights and biases into a very big matrix. Then we take the derivative of all weights and biases, then again concatenate them into a very giant matrix. Now let's see whether or not the derivative we performed is the gradient of cost function $J(\theta)$.

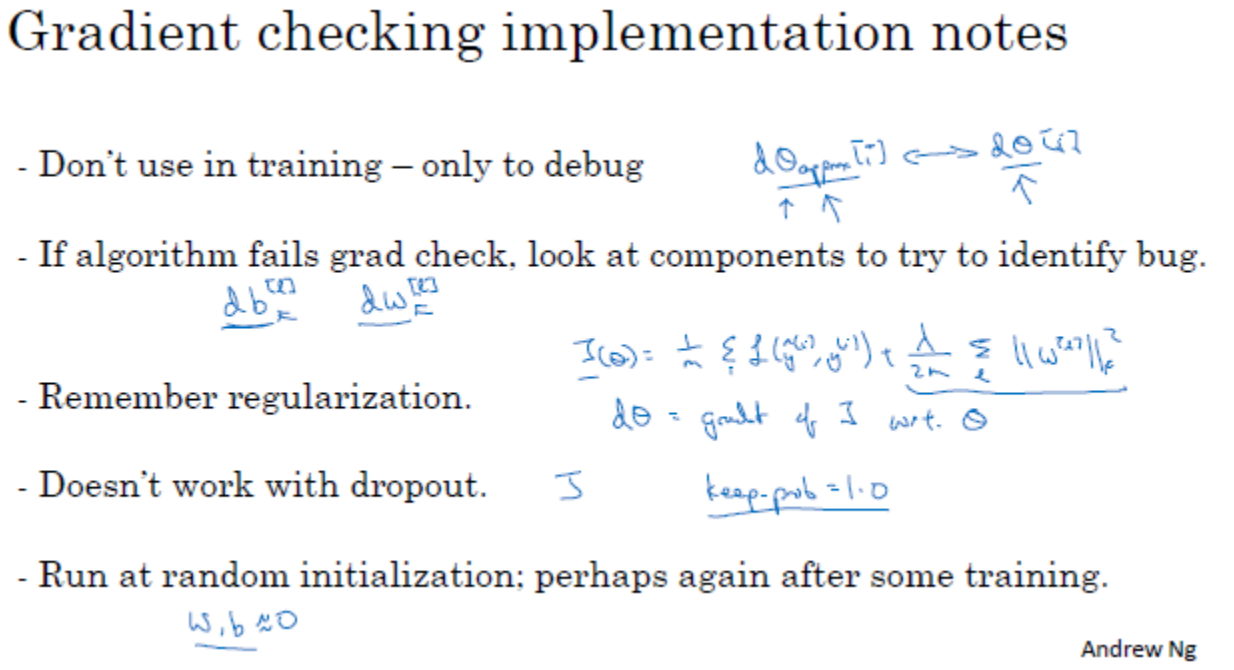


Now we apply the idea we got from numerical approximation, we take two side difference of the of the giant matrix $d(\theta)$. Then we take the distance between this approximated and actual result. It should get a value less than or around 10^{-7}, it is great. But if it is larger than or around 10^{-3}, we should get worried.



Gradient Checking Implementation Notes

The first bullet, suggest that we should not use grad check in the whole training process as it is so slow.
The second bullet, provides the fact that with grad checking, we can identify whether the problem is from the gradient of the bias terms or weights.
The third bullet, highlights that if we want to use grad check and we have a regularizer, we should take it into account in the grad check formulation.
The fourth bullet, warns that we should not use grad check while we are using dropout. We can do each one of them separately.
The last bullet is a hint about when to use grad check. It is the best to use at the random initialization step and after some epochs because if the weights and biases get large values, even if we implemented everything correctly, the difference of approximation and actual gradient will be larger than expected.



Assignment Notes

In general, initializing all the weights to zero results in the network failing to break symmetry. This means that every neuron in each layer will learn the same thing, and you might as well be training a neural network with $n^{[l]}=1$ for every layer, and the network is no more powerful than a linear classifier such as logistic regression.
Poor initialization can lead to vanishing/exploding gradients, which also slows down the optimization algorithm.

If you train this network longer you will see better results, but initializing with overly large random numbers slows down the optimization.

The cost starts very high. This is because with large random-valued weights, the last activation (sigmoid) outputs results that are very close to 0 or 1 for some examples, and when it gets that example wrong it incurs a very high loss for that example. Indeed, when $\log(a^{[3]})=\log(0)$, the loss goes to infinity.

In summary:
- Initializing weights to very large random values does not work well.
- Hopefully intializing with small random values does better. The important question is: how small should be these random values be? Lets find out in the next part!

Observations:
- The value of λ is a hyperparameter that you can tune using a dev set.
- L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias.

What is L2-regularization actually doing?:

L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

<font color='blue'>
What you should remember -- the implications of L2-regularization on:
- The cost computation:
  - A regularization term is added to the cost
- The backpropagation function:
  - There are extra terms in the gradients with respect to weight matrices
- Weights end up smaller ("weight decay"):
- Weights are pushed to smaller values.

- Dropout is a regularization technique.
- You only use dropout during training. Don't use dropout (randomly eliminate nodes) during test time.
- Apply dropout both during forward and backward propagation.
- During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations. For example, if keep_prob is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when keep_prob is other values than 0.5.

It seems that there were errors in the `backward_propagation_n` code we gave you! Good that you've implemented the gradient check. Go back to `backward_propagation` and try to find/correct the errors *(Hint: check dW2 and db1)*. Rerun the gradient check when you think you've fixed it. Remember you'll need to re-execute the cell defining `backward_propagation_n()` if you modify the code.

Can you get gradient check to declare your derivative computation correct? Even though this part of the assignment isn't graded, we strongly urge you to try to find the bug and re-run gradient check until you're convinced backprop is now correctly implemented.

Note
- Gradient Checking is slow! Approximating the gradient with $\frac{\partial J}{\partial \theta} \approx \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2 \varepsilon}$ is computationally costly. For this reason, we don't run gradient checking at every iteration during training. Just a few times to check if the gradient is correct.
- Gradient Checking, at least as we've presented it, doesn't work with dropout. You would usually run the gradient check algorithm without dropout to make sure your backprop is correct, then add dropout.

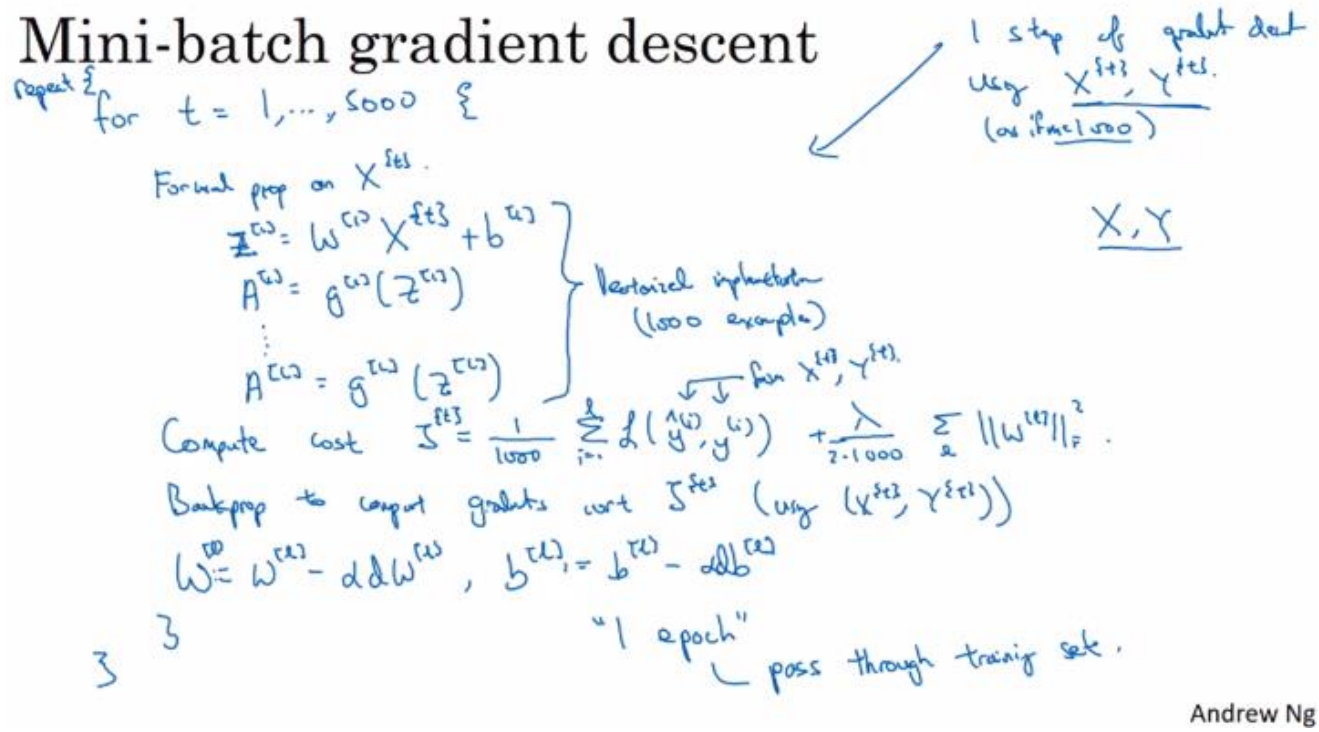
Course 2
Week 2

Mini-batch gradient descent

Vectorization allows you to efficiently compute on *m* examples.

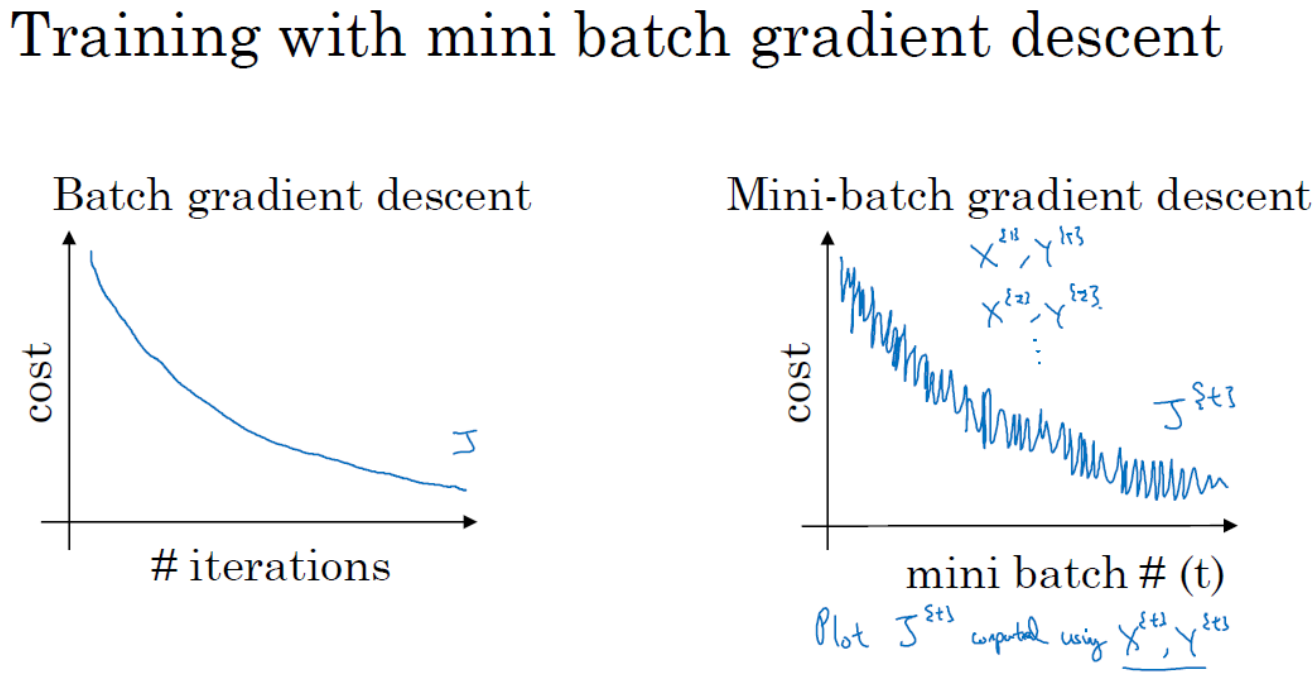
Processing the whole training set and take its gradient descent to find the optimal solution. However, processing the entire training set is time consuming. Get a faster algorithm is possible if we start to make some progress even before finishing processing entire dataset. That is why we jump from batch gradient descent into mini-batch gradient descent.

We have to take mini-batches (baby training set). We take the mini-batch gradient descent which is a gradient descent for each batch.



Understanding mini-batch gradient descent

In batch gradient descent the cost decreases continuously (if everything goes well), however, in the mini-batch gradient descent we are oscillating as the trend goes down in overall. The reason is that maybe in one batch we are doing well in training and the cost goes down; but, on the next batch we may see that the training is harder due to many reasons such as incorrect labels, so the cost may not go down and instead may go a little worse than the previous one.



Two extremes are:

1) If mini-batch size = *m* = batch size*:* Batch gradient descent → too long per iteration
2) If mini-batch size = 1: stochastic gradient descent → lose speed up from not having vectorization

In the first extreme, we know that we have a little noise and we end-up to the minimum; however takes longer. (the blue in the figure below)

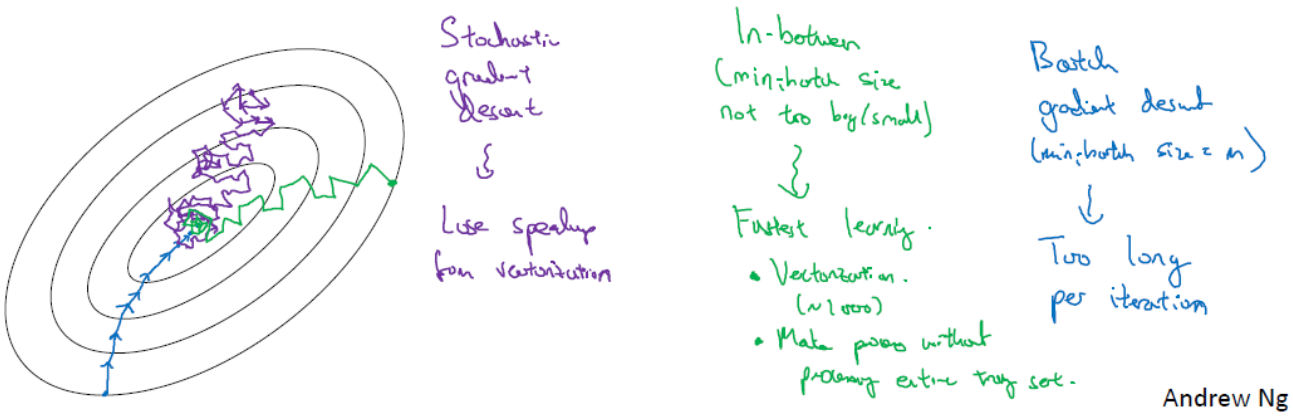
In the second extreme, we are always oscillating because updating the rule based on one sample is not ideal and causes oscillation, specifically around the global minimum. (the purple in the figure below)

Something in between is the best option for mini-batch size which is not too big or too small → fastest learning since it:

- supports vectorization
- does not need to wait for the whole training set to update weights

We can see it as the green in the figure below.

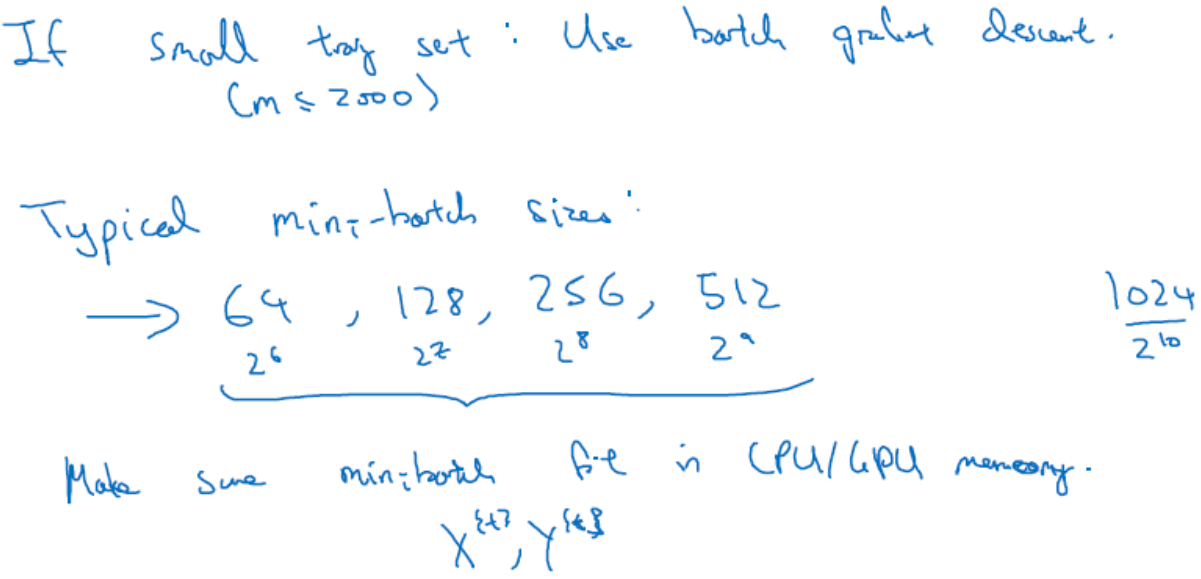
Choosing your mini-batch size



But how to choose its size? There are some rule of thumbs:

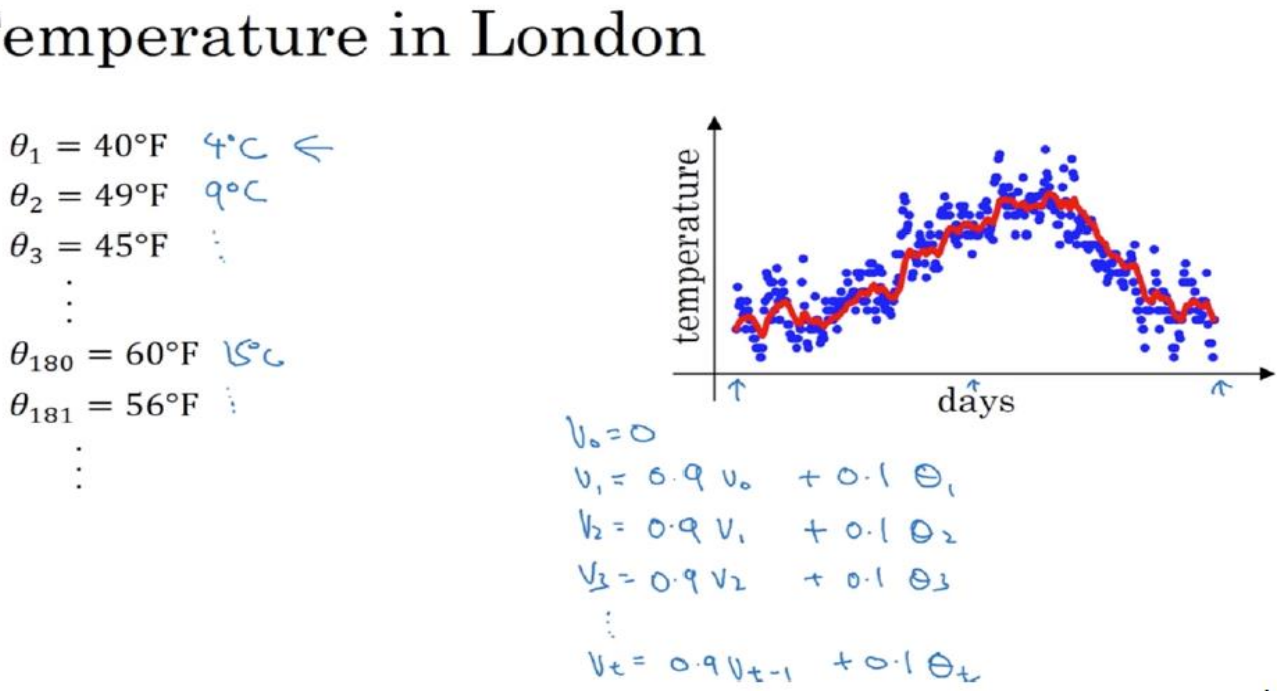
- Small training set: use BGD because we can go through the whole dataset very fast (m <= 2000)
- Typical mini-batch size: better to be power of 2 like *64, 128, 256, 512*, or even 1024, etc.
- Make sure mini-batch size fits in CPU/GPU memory → mini-batch size is a hyper-parameter, try some of the power of 2 and check which ones converge faster.

Choosing your mini-batch size

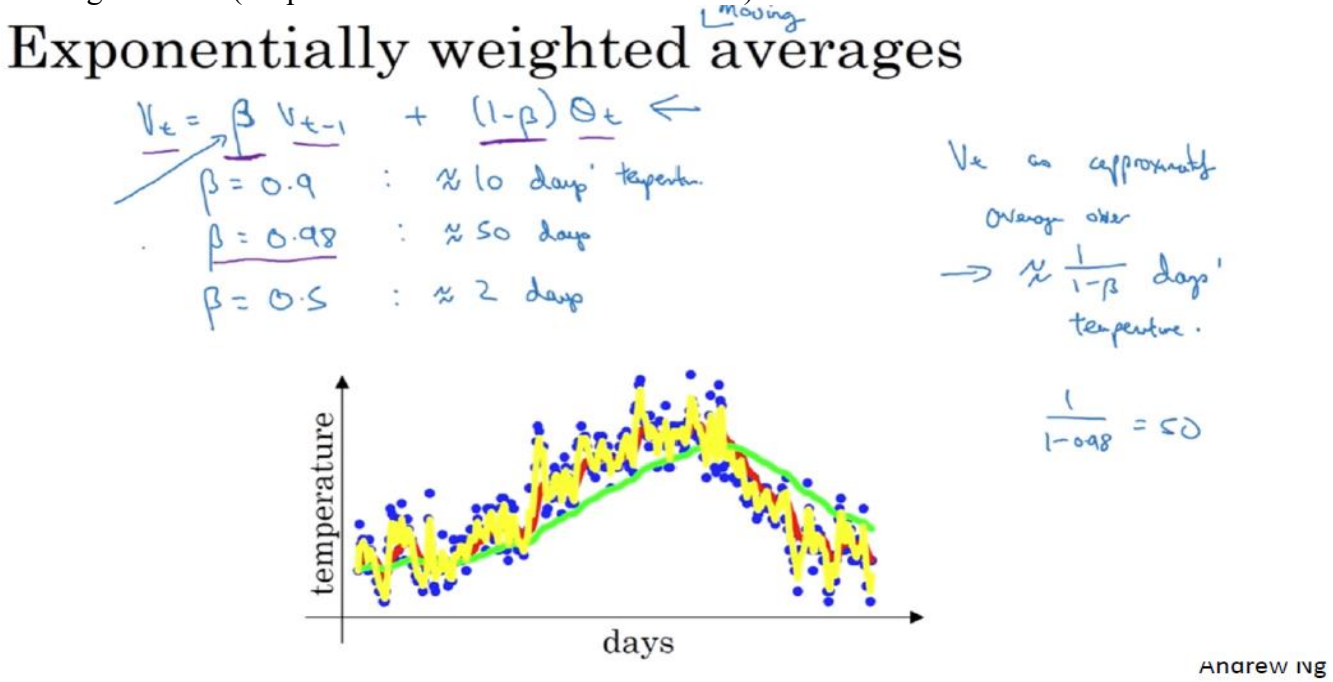


Exponentially weighted averages

For this one, let's start with an example of a moving average which gives 0.9 weight to the previous value and 0.1 to the current one. Its plot is in red:

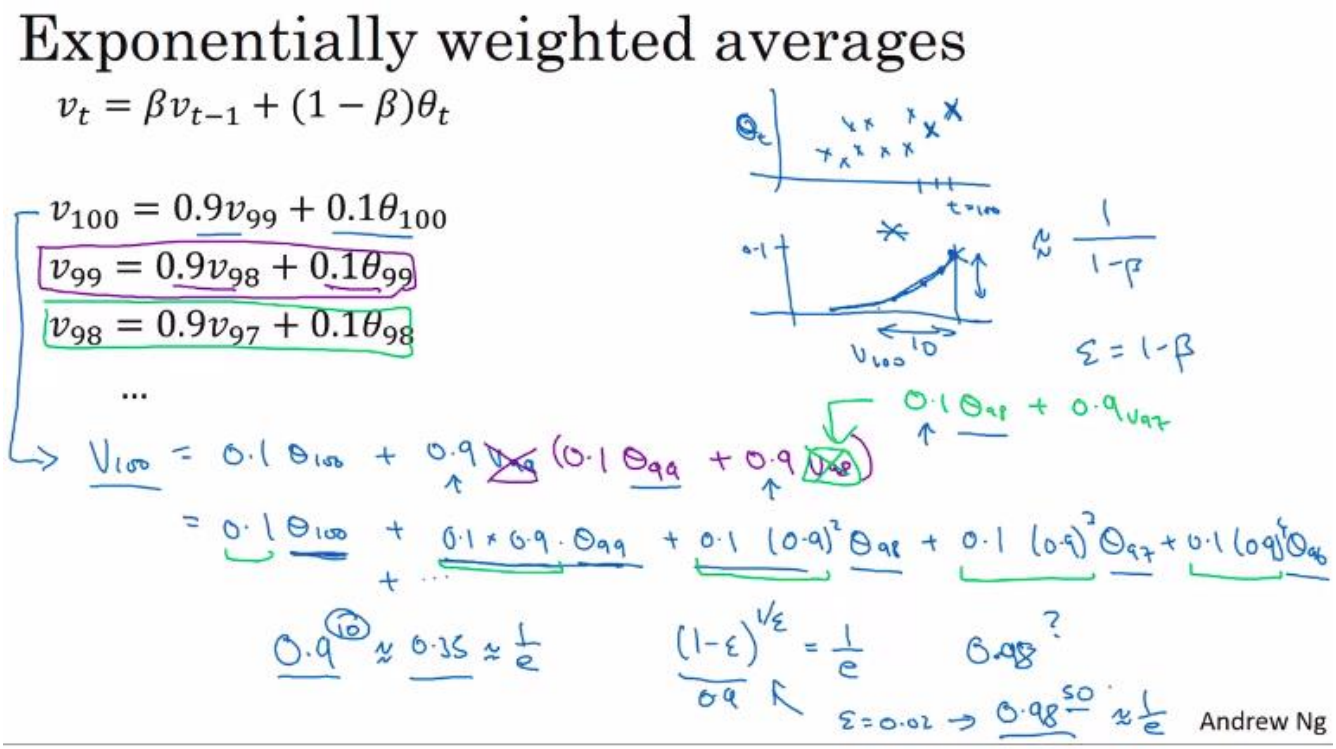


In the concept of exponentially weighted average, the information from the past $1 / (1 - \beta)$ days are preserved. For this reason, if we increase β, the number of preserved days in the moving average is higher so the green one tends to be smoother and shifted to the right as adapts slower since the current value get less importance. On the flip side, having β as 0.5 (the yellow one) oscillates more and converges sooner (adapts itself sooner to the current value).



Understanding exponentially weighted averages

if the beta is 0.9, it takes around 10 days to get around one third of the 0.1. If the beta is 0.98 it takes 50 days to go to the 1/3 of the first value which is 0.02. In other words if the very first value is epsilon we have (1-ep)^(1/ep) = 1/e



$$v_t = (1-\beta)\theta_t + \beta v_{t-1} = (1-\beta)\theta_t + \beta(1-\beta)\theta_{t-1} + \beta^2 v_{t-1}$$
$$= (1-\beta)\theta_t + \beta(1-\beta)\theta_{t-1} + \beta^2(1-\beta)\theta_{t-2} + \cdots + \beta^t(1-\beta)\theta_{t-t}$$
$$= (1-\beta)[\theta_t + \beta\theta_{t-1} + \beta^2\theta_{t-2} + \cdots + \beta^t\theta_{t-t}]$$

We know that:

$$\frac{1}{(1-\beta)} = 1 + (1-(1-\beta)) + \left(1-(1-\beta)\right)^2 + \cdots$$
$$= 1 + (\beta) + (\beta)^2 + \cdots$$

Now, back to our problem, we have:

$$v_t = \frac{[\theta_t + \beta\theta_{t-1} + \beta^2\theta_{t-2} + \cdots + \beta^t\theta_{t-t}]}{1 + (\beta) + (\beta)^2 + \cdots}$$

If we assume the $\theta_t s'$ variation is small, we can omit this variable. Moreover, we assume that after k_{th} averaging, we can neglect the previous ones. So we re-write the above equation as:

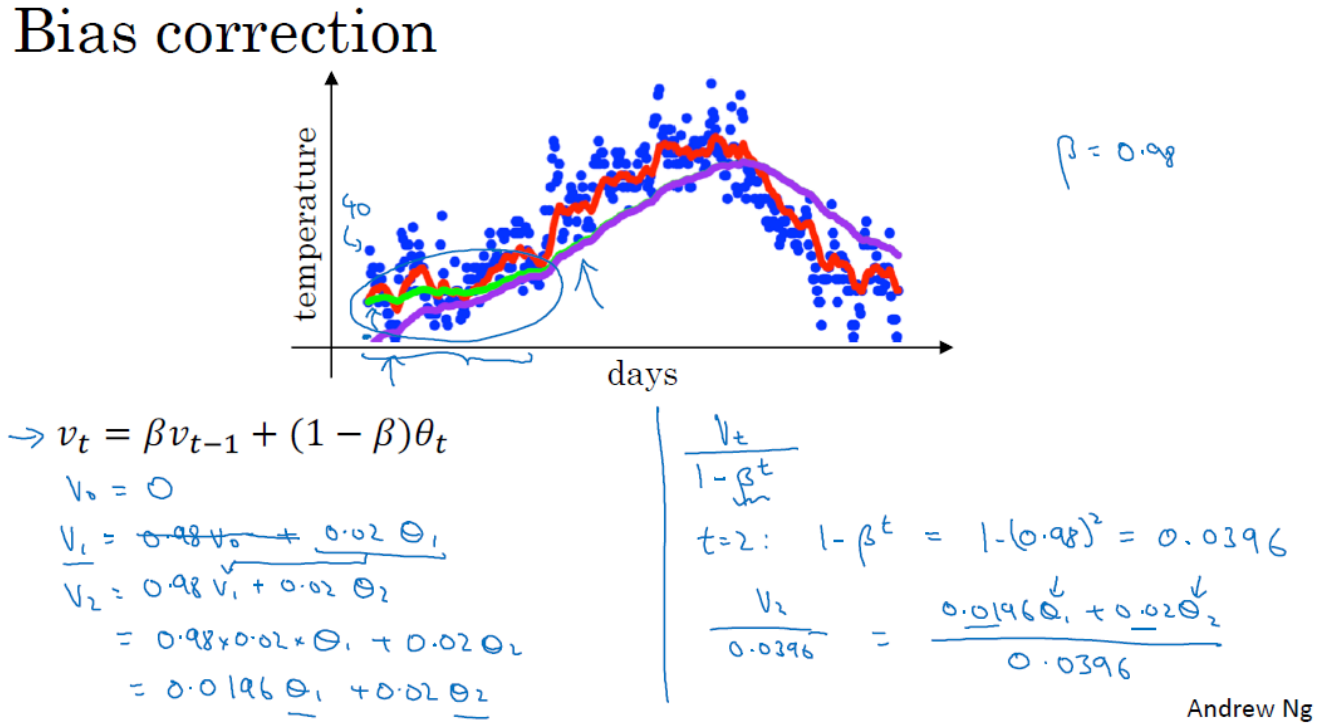
$$\frac{(\beta^k + \beta^{k+1} + \beta^{k+2} + \cdots)}{1 + (\beta) + (\beta)^2 + \cdots} = \frac{\beta^k(1 + \beta^1 + \beta^2 + \cdots)}{1 + (\beta) + (\beta)^2 + \cdots} = \beta^k$$

Now let's say that a value less than $1/e$ which is around 0.37 is negligible. Then we can write it down as:

$\beta^k = \frac{1}{e}$ ➔ $k\log(\beta) = -\log(e)$ ➔ $k = -\frac{\log(e)}{\log(\beta)}$

For example, if $\beta = 0.9$ as the above example, we get $k = 9.5$ in which means that after around 10 days the importance of the first day goes less than $1/e$ and we can ignore it.

Bias correction in exponentially weighted averages



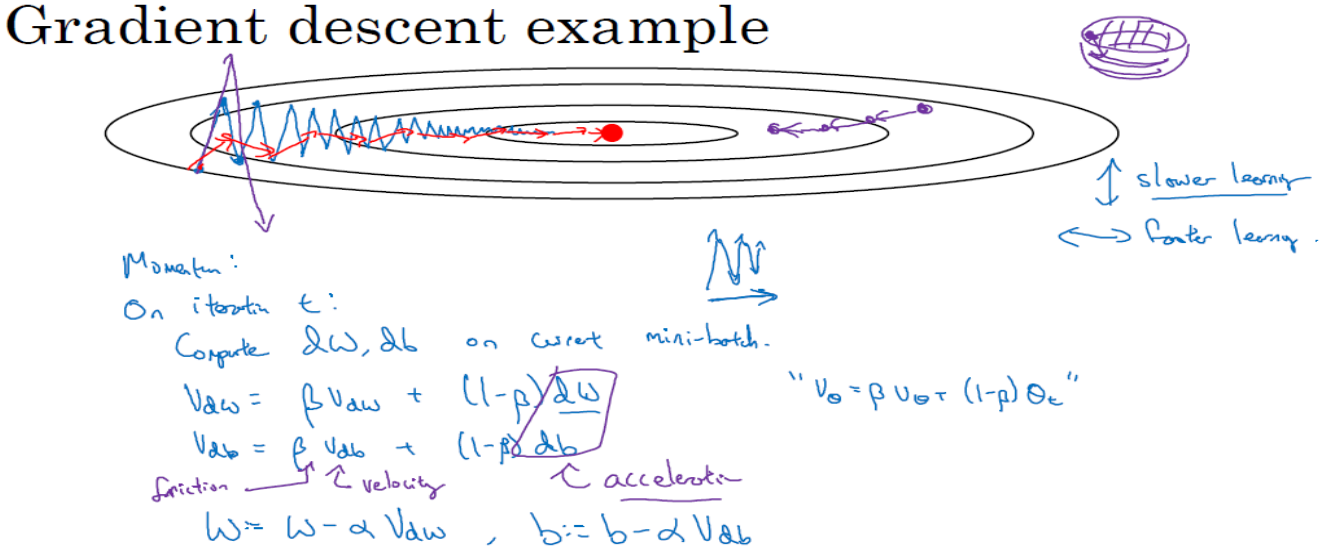
If we look at the example on the left side of the figure above, we see that when we start from $v_0 = 0$, it takes some time for the system to warm up. In this case we need to give it a bias which helps the system to warm up sooner. For doing so, we want the coefficient of the first data in the first averaging to be 1, instead of $1 - \beta$ (in the example above it is 0.02). For doing so, the whole equation should be divided by $1 - \beta$. However, we also want the effect of bias to converge to 0 in the long run. This happens when we use β^t which means when $t \to \infty$ and $0 < \beta < 1, \beta^t \to 0$; so $1 - \beta^t \to 1$.
By doing so, the purple line becomes the green one.

Gradient Descent with momentum

Oscillations toward the center, most of the times, takes many steps as sometimes steps causes divergence from the correct path. Moreover, like the example below, learning rate decreases faster vertically than horizontally.

Momentum optimizer smooth up each iteration in the gradient descent by manipulating learning rate for each dimension. If you take the average of the vertical oscillation, it will tends to zero. So it slows down the vertical learning rate. But for the horizontal one it is much faster. → more straight forward path to the minimum.

In these equations, v_{dw} and v_{db} are acting as velocity or momentum toward directions by taking the moving average of the previous steps and dw and db play the role of accelerator as always.



α and β control exponentially moving weighted average and both of them are hyper-parameters. You can use beta as hyper parameters. Bias correction is not needed in practice; however, using it does not hurt and actually is useful.

Also, sometimes $1-\beta$ terms are omitted. Using it or not using it are both acceptable but omitting them makes it less intuitive; then we need to retune α again.

Implementation details

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

On iteration t:

Compute dW, db on the current mini-batch

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

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

$V_{dw} = \beta V_{dw} + dW$

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

$\frac{V_{dw}}{1-\beta^t}$

Hyperparameters: α, β

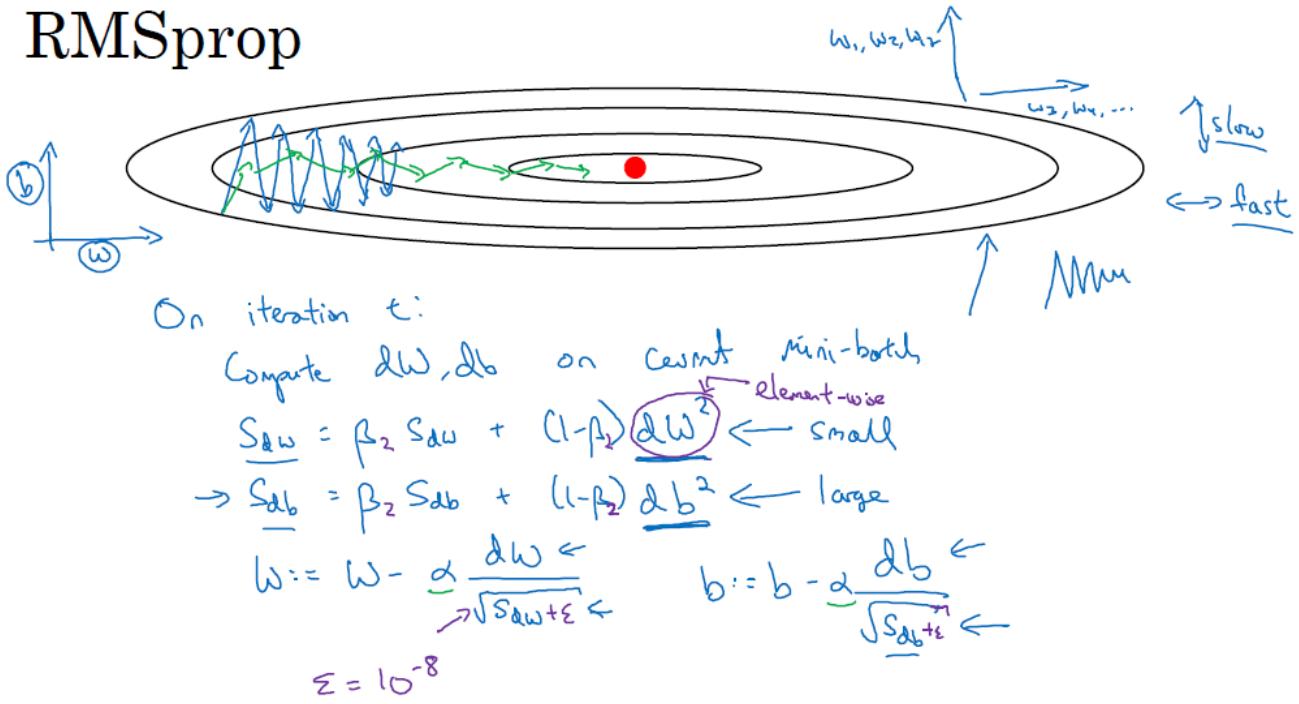
$\beta = 0.9$

average over last ≈10 gradients

Andrew Ng

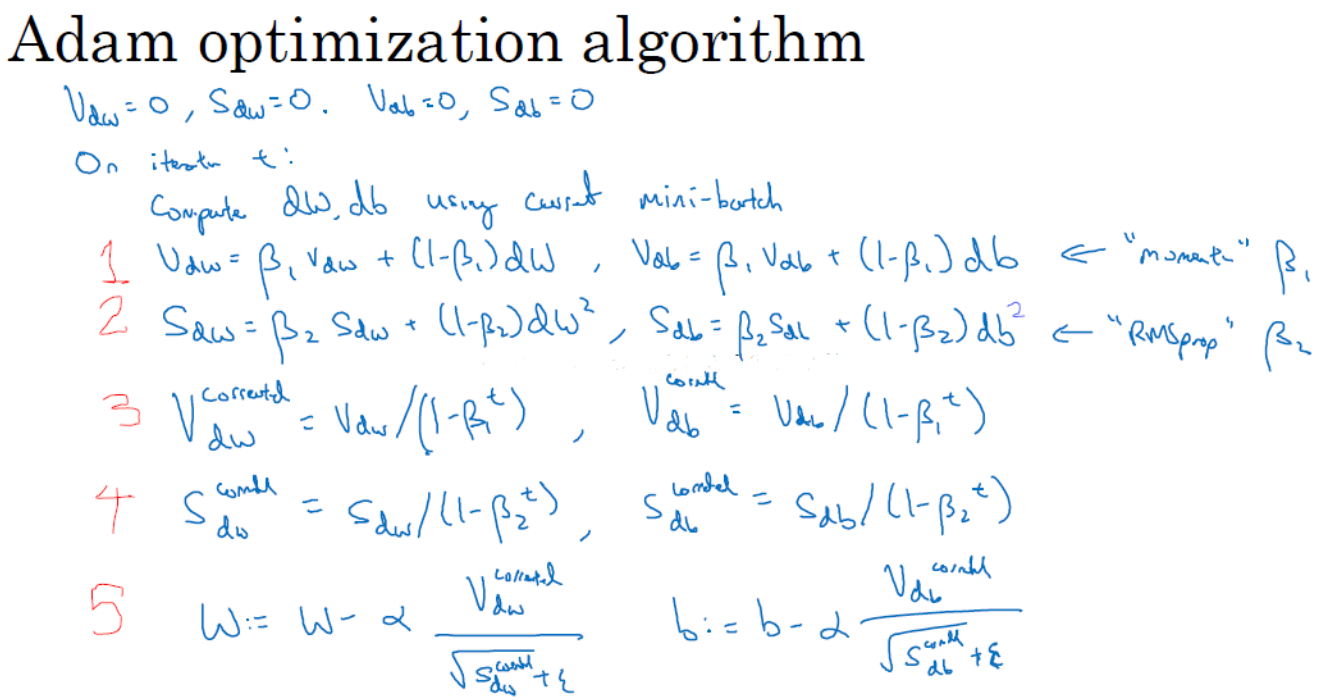
RMSprop

To increase the momentum we have RMSprop. For the sake of the example shown in the slide below, if we assume *b* is the horizontal movement and *w* is the vertical movement, our goal is to move toward the goal by decreasing *b* such way that it oscillates less vertically. The same applies on *w*. In this example, we want to slow down learning on vertical, speed up on horizontal. The power of 2 causes the *dw* to become smaller (if it is less than 1) and *db* becomes larger so that the *Sdw* and *Sdb* are getting smaller and bigger respectively. Then in the update rule, by dividing the gradient on a squared root of these values the path toward the global minima would be less oscillated. In this way we can use larger learning rate. In practice dW is a high dimensional value.
One more point is that we need to add epsilon to denominator as it might get so close to 0 and we may encounter a division by 0 problem.



Adam Optimization Algorithm

Momentum + RMSprop = Adam (Adaptive Moment Estimation)
It has bias correction in it. → highly popular and useful
In the slide below, line 1 is the same as what we did in the momentum. Line 2 is the same as what we did in the RMSprop. Lines 3 and 4 are only about bias correction. Finally, line 5 is the mixture of momentum and RMSprop algorithms for updating rules.

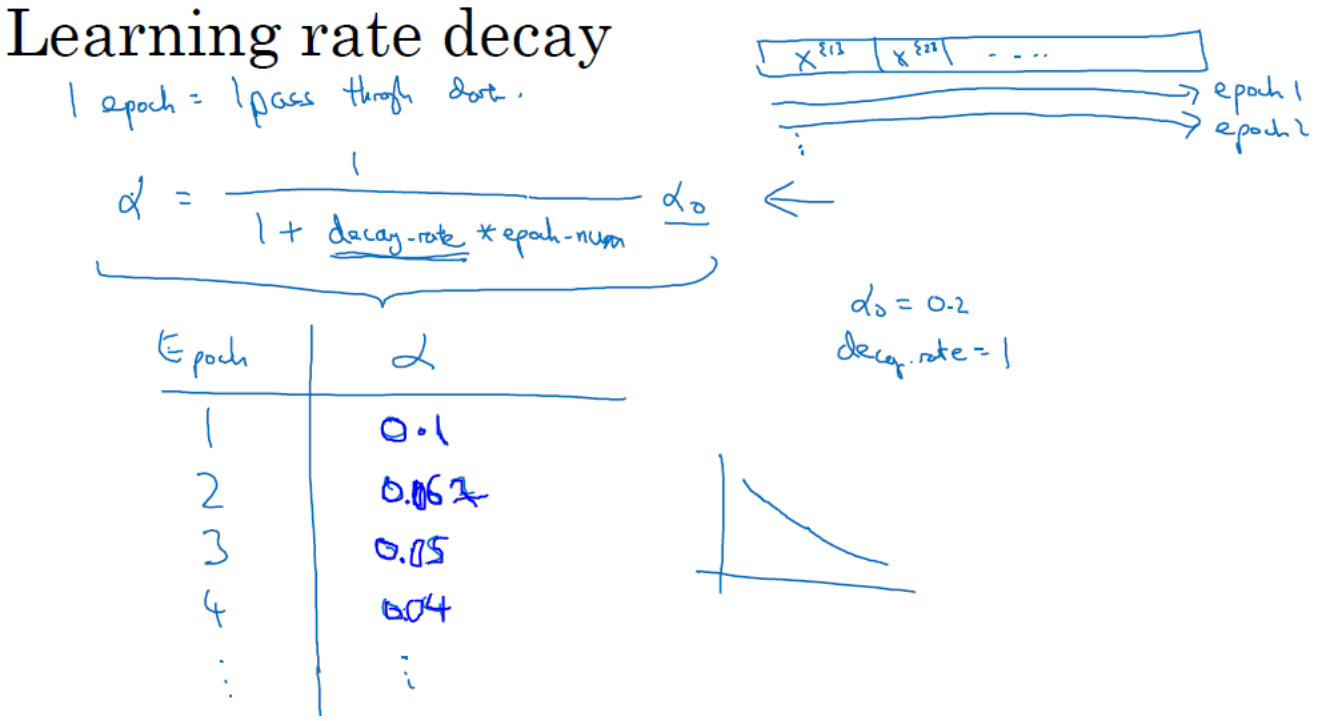


For the hyper-parameters:

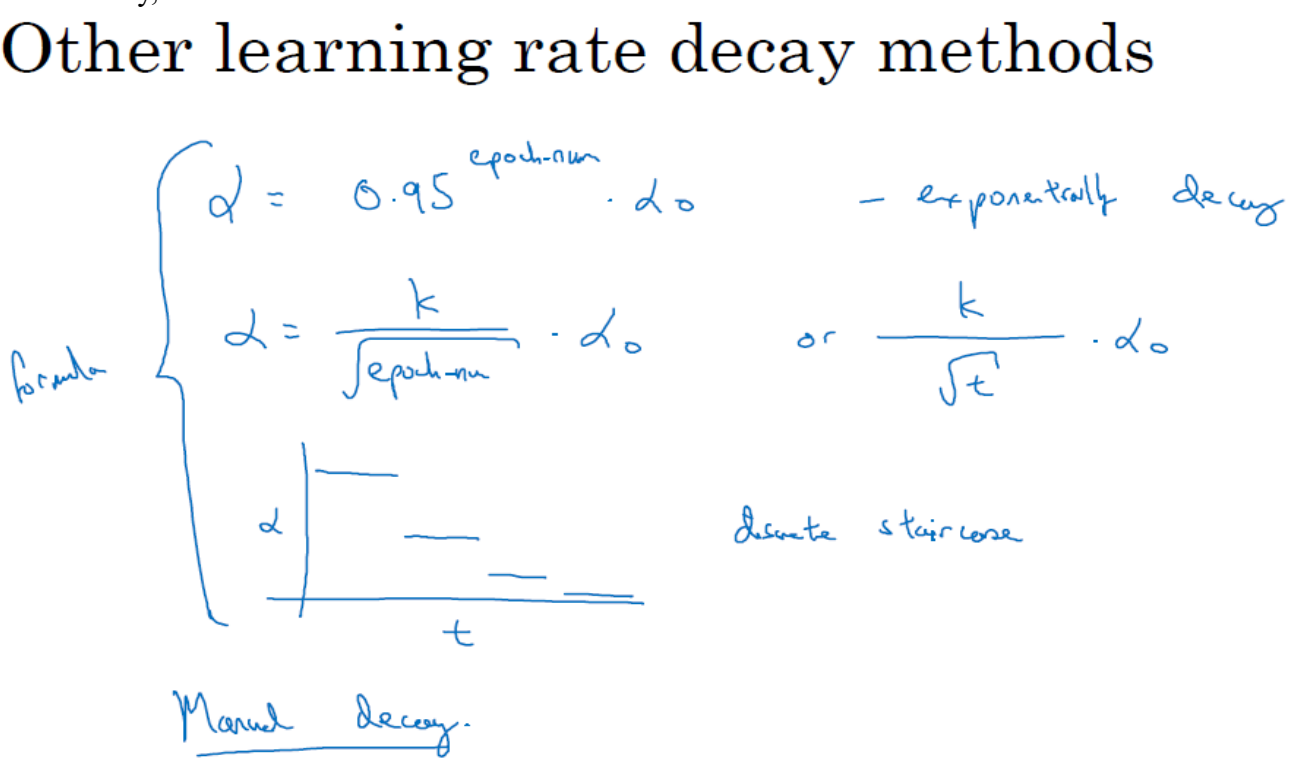
- α: needs to be tune
- β_1: 0.9 (dw) → default value and does not need to be tuned most of the times
- β_2: 0.999 (dw^2) → default value and does not need to be tuned most of the times
- ε: 10^{-8} → default value and does not need to be tuned at all

Learning rate decay

Sometimes with fix value alpha and noise we wander around min but never meet it because the learning rate is large. So we need to implement an adaptive learning rate.



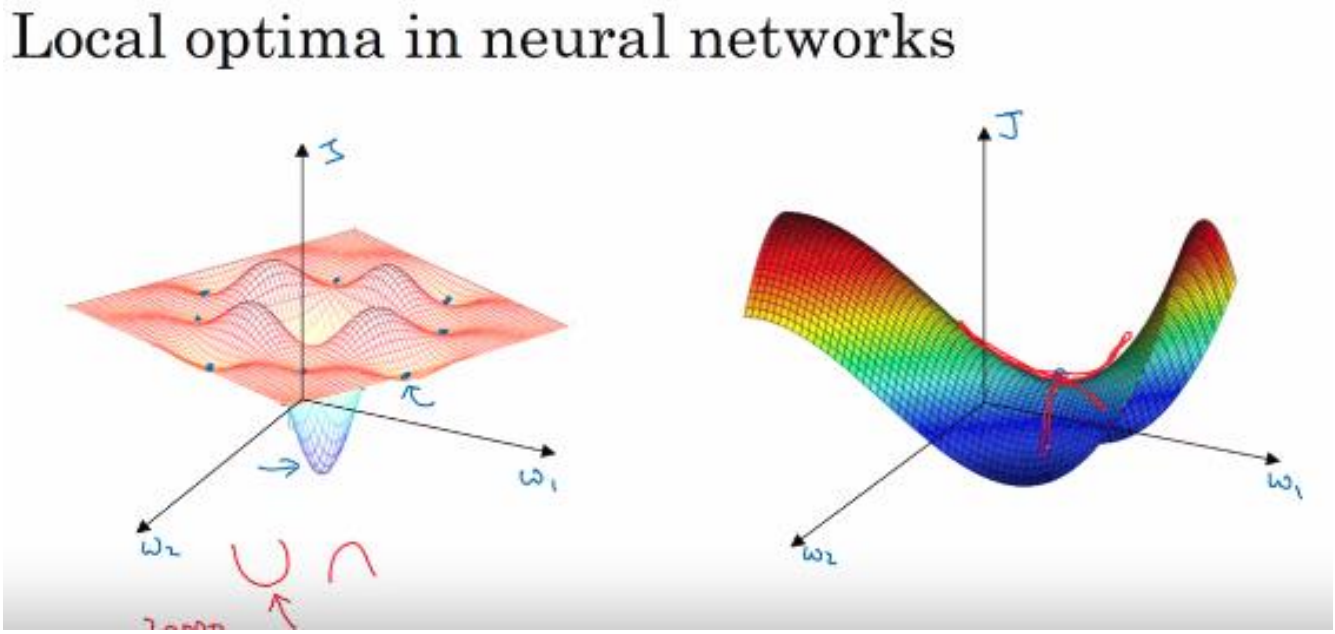
Decay rate is another hyper-parameter. We have also exponentially decay, discrete stair case decay, manual decay, etc.

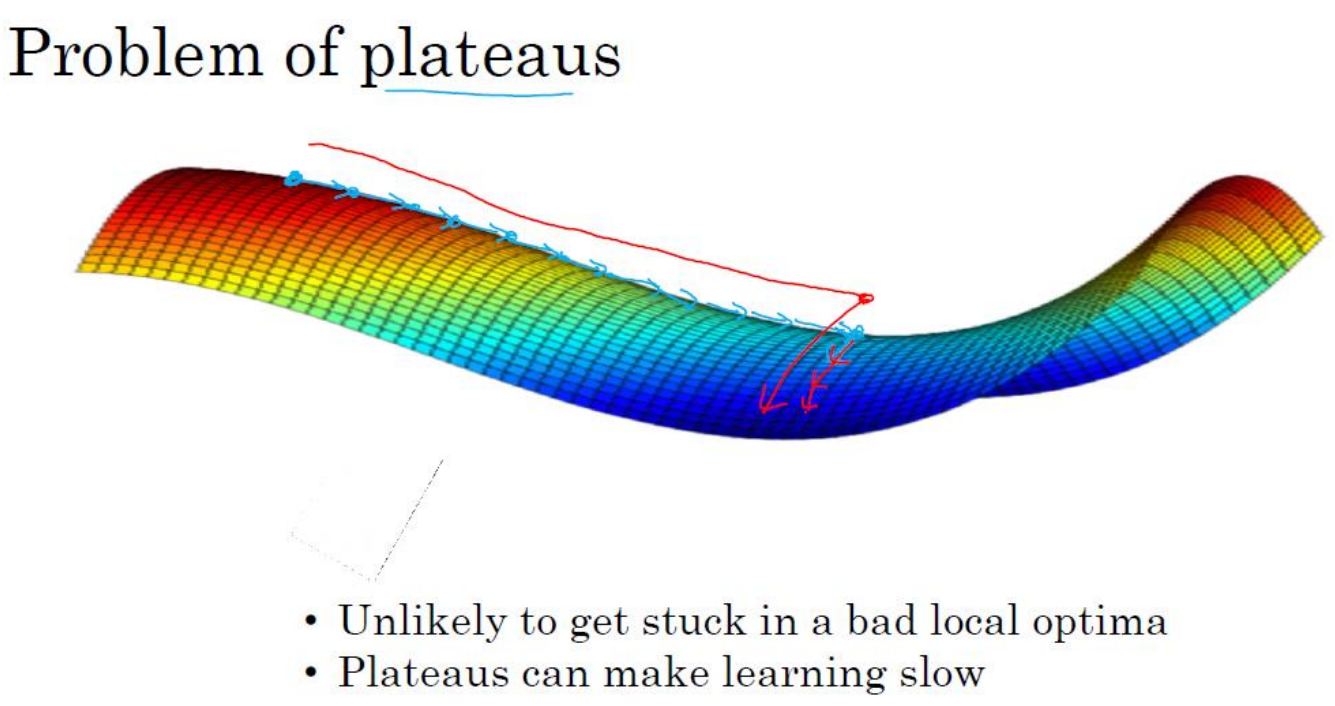


The problem of local optima

Local optima problem: gradient descent might end up in local optima. However, in neural network, most points that were assumed as local optima are only saddle points (derivative of zero).

The plateau is a region where the derivative is close to zero for a long time and this is the problem that really slow down the learning. So not only local optima is a problem, but plateau is also a problem. So it is unlikely to get stuck in a bad local optima. Plateaus are the problems which adam or RMSprop can be helpful here to move down the plateau.





Assignment

A variant of this is Stochastic Gradient Descent (SGD), which is equivalent to mini-batch gradient descent where each mini-batch has just 1 example. The update rule that you have just implemented does not change. What changes is that you would be computing gradients on just one training example at a time, rather than on the whole training set. The code examples below illustrate the difference between stochastic gradient descent and (batch) gradient descent.

- **(Batch) Gradient Descent**:

```` python
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    # Forward propagation
    a, caches = forward_propagation(X, parameters)
    # Compute cost.
    cost += compute_cost(a, Y)
    # Backward propagation.
    grads = backward_propagation(a, caches, parameters)
    # Update parameters.
    parameters = update_parameters(parameters, grads)

```

- **Stochastic Gradient Descent**:

```python
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    for j in range(0, m):
        # Forward propagation
        a, caches = forward_propagation(X[:,j], parameters)
        # Compute cost
        cost += compute_cost(a, Y[:,j])
        # Backward propagation
        grads = backward_propagation(a, caches, parameters)
        # Update parameters.
        parameters = update_parameters(parameters, grads)
```

- The difference between gradient descent, mini-batch gradient descent and stochastic gradient descent is the number of examples you use to perform one update step.
- You have to tune a learning rate hyperparameter $\alpha$.

- With a well-turned mini-batch size, usually it outperforms either gradient descent or stochastic gradient descent (particularly when the training set is large).

In Stochastic Gradient Descent, you use only 1 training example before updating the gradients. When the training set is large, SGD can be faster. But the parameters will "oscillate" toward the minimum rather than converge smoothly. Here is an illustration of this:

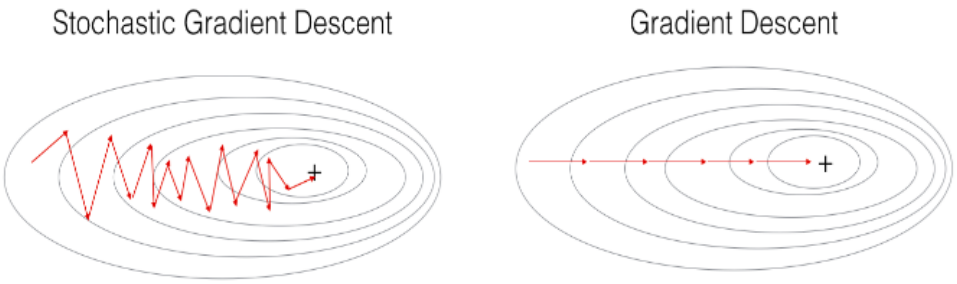



**Figure 1** : **SGD vs GD**
"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

**Note** also that implementing SGD requires 3 for-loops in total:

1. Over the number of iterations
2. Over the $m$ training examples
3. Over the layers (to update all parameters, from $(W^{[1]}, b^{[1]})$ to $(W^{[L]}, b^{[L]})$)

In practice, you'll often get faster results if you do not use neither the whole training set, nor only one training example, to perform each update. Mini-batch gradient descent uses an intermediate number of examples for each step. With mini-batch gradient descent, you loop over the mini-batches instead of looping over individual training examples.

In practice, you'll often get faster results if you do not use neither the whole training set, nor only one training example, to perform each update. Mini-batch gradient descent uses an intermediate number of examples for each step. With mini-batch gradient descent, you loop over the mini-batches instead of looping over individual training examples.

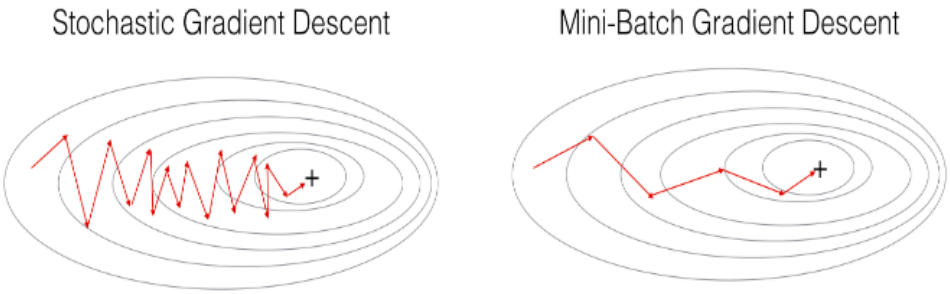



**Figure 2** : **SGD vs Mini-Batch GD**
"+" denotes a minimum of the cost. Using mini-batches in your optimization algorithm often leads to faster optimization.

**What you should remember**:

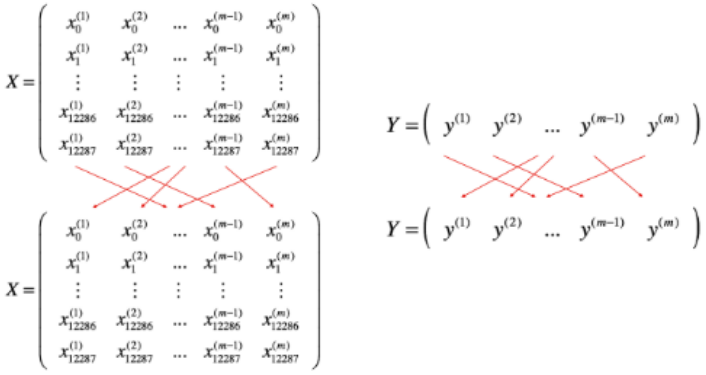
- The difference between gradient descent, mini-batch gradient descent and stochastic gradient descent is the number of examples you use to perform one update step.
- You have to tune a learning rate hyperparameter $\alpha$.
- With a well-turned mini-batch size, usually it outperforms either gradient descent or stochastic gradient descent (particularly when the training set is large).

## 2 - Mini-Batch Gradient descent

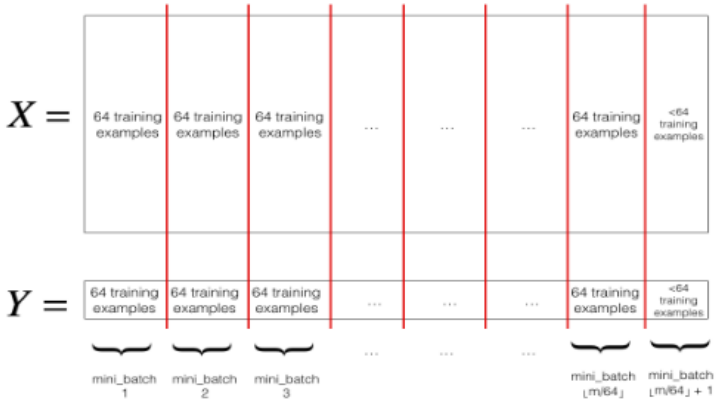
Let's learn how to build mini-batches from the training set (X, Y).

There are two steps:

- **Shuffle**: Create a shuffled version of the training set (X, Y) as shown below. Each column of X and Y represents a training example. Note that the random shuffling is done synchronously between X and Y. Such that after the shuffling the $i^{th}$ column of X is the example corresponding to the $i^{th}$ label in Y. The shuffling step ensures that examples will be split randomly into different mini-batches.



- **Partition**: Partition the shuffled (X, Y) into mini-batches of size `mini_batch_size` (here 64). Note that the number of training examples is not always divisible by `mini_batch_size`. The last mini batch might be smaller, but you don't need to worry about this. When the final mini-batch is smaller than the full `mini_batch_size`, it will look like this:



**What you should remember**:

- Shuffling and Partitioning are the two steps required to build mini-batches
- Powers of two are often chosen to be the mini-batch size, e.g., 16, 32, 64, 128.

## 3 - Momentum

Because mini-batch gradient descent makes a parameter update after seeing just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch gradient descent will "oscillate" toward convergence. Using momentum can reduce these oscillations.

Momentum takes into account the past gradients to smooth out the update. We will store the 'direction' of the previous gradients in the variable $v$. Formally, this will be the exponentially weighted average of the gradient on previous steps. You can also think of $v$ as the "velocity" of a ball rolling downhill, building up speed (and momentum) according to the direction of the gradient/slope of the hill.

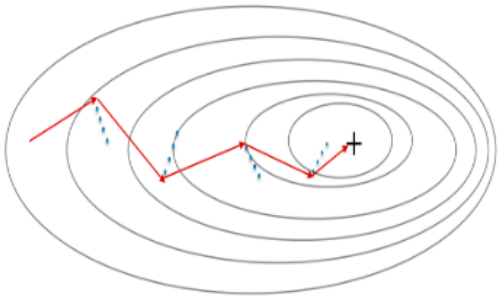


**Figure 3**: The red arrows shows the direction taken by one step of mini-batch gradient descent with momentum. The blue points show the direction of the gradient (with respect to the current mini-batch) on each step. Rather than just following the gradient, we let the gradient influence $v$ and then take a step in the direction of $v$.

**Note** that:

- The velocity is initialized with zeros. So the algorithm will take a few iterations to "build up" velocity and start to take bigger steps.
- If $\beta = 0$, then this just becomes standard gradient descent without momentum.

**How do you choose $\beta$?**

- The larger the momentum $\beta$ is, the smoother the update because the more we take the past gradients into account. But if $\beta$ is too big, it could also smooth out the updates too much.
- Common values for $\beta$ range from 0.8 to 0.999. If you don't feel inclined to tune this, $\beta = 0.9$ is often a reasonable default.
- Tuning the optimal $\beta$ for your model might need trying several values to see what works best in term of reducing the value of the cost function $J$.

**What you should remember**:

- Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.
- You have to tune a momentum hyperparameter $\beta$ and a learning rate $\alpha$.

## 4 - Adam

Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from RMSProp (described in lecture) and Momentum.

**How does Adam work?**

1. It calculates an exponentially weighted average of past gradients, and stores it in variables $v$ (before bias correction) and $v^{corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables $s$ (before bias correction) and $s^{corrected}$ (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

The update rule is, for $l = 1, ..., L$ :

$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1 - \beta_1) \frac{\partial \mathcal{J}}{\partial W^{[l]}} \\ v^{corrected}_{dW^{[l]}} = \frac{v_{dW^{[l]}}}{1 - (\beta_1)^t} \\ s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1 - \beta_2) (\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2 \\ s^{corrected}_{dW^{[l]}} = \frac{s_{dW^{[l]}}}{1 - (\beta_2)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v^{corrected}_{dW^{[l]}}}{\sqrt{s^{corrected}_{dW^{[l]}}} + \varepsilon} \end{cases}$$

where:

- t counts the number of steps taken of Adam
- L is the number of layers
- $\beta_1$ and $\beta_2$ are hyperparameters that control the two exponentially weighted averages.
- $\alpha$ is the learning rate
- $\varepsilon$ is a very small number to avoid dividing by zero

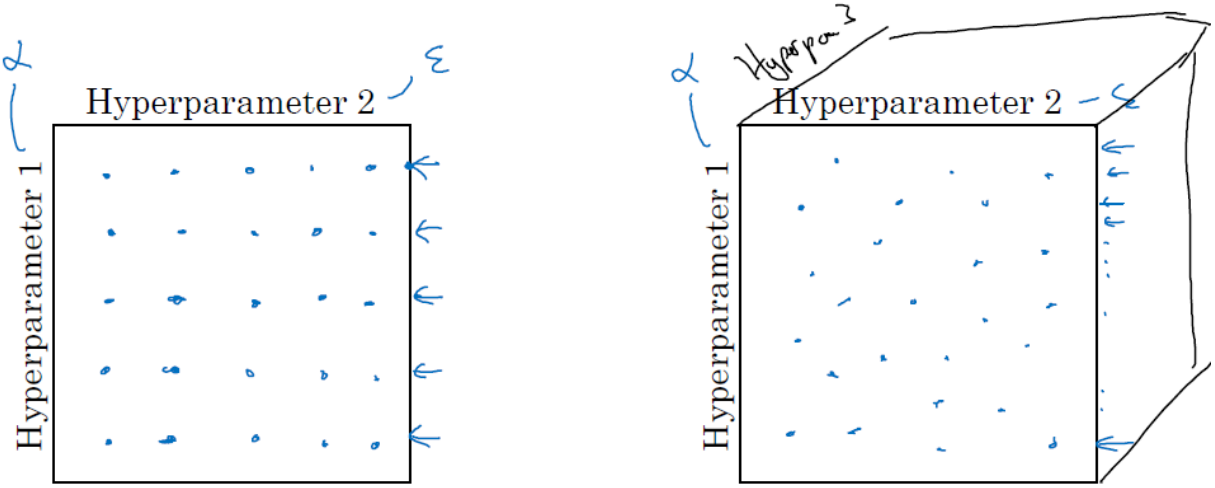
As usual, we will store all parameters in the `parameters` dictionary

# Course 2
# WEEK 3

**Tunning process**

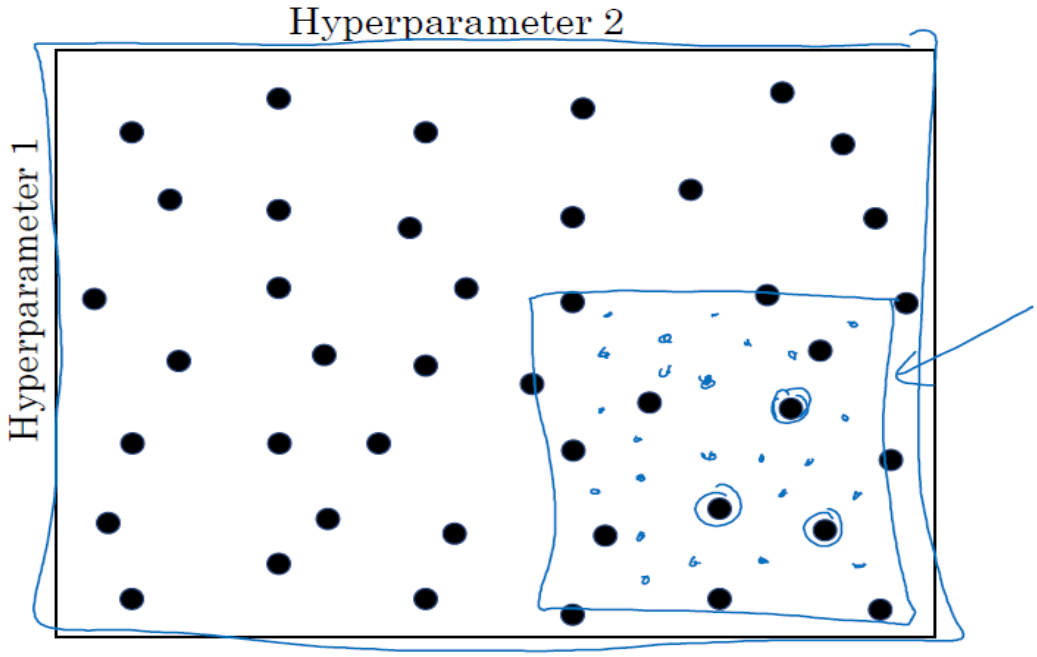
Sampling the points in the grid is practical. In an era before the deep learning, choosing the points such that is shown in left figure of the below slide was practical. In deep learning era, points must be random. Because in advance it is hard to know which hyper-parameter is the best for tunning. Because some are more important than the others. The reason is that, in the left figure, for the hyper-parameter 2, we only have 5 values to check. However, in the right figure, there are 25 of them which gives us more flexibility to choose the value wisely.

## Try random values: Don't use a grid



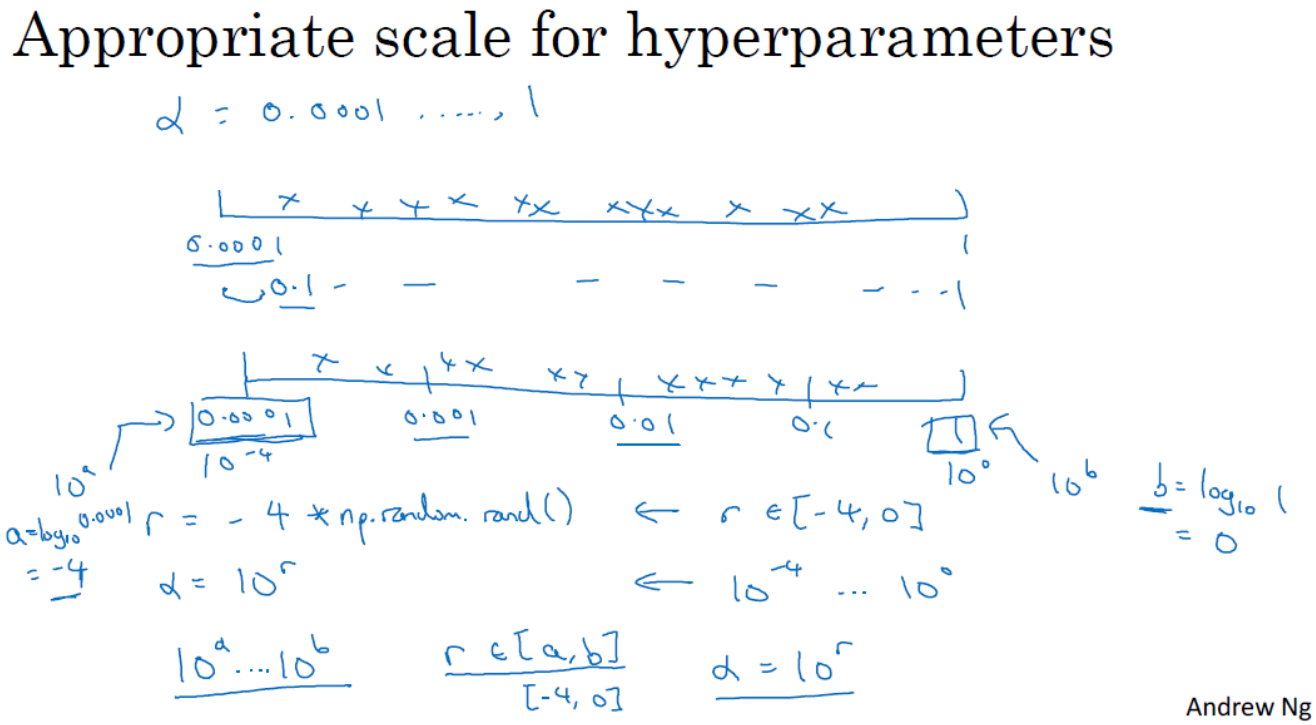

In the coarse to fine scheme, we want to find which values for hyper-parameters in our random grid works the best. So we find it out roughly by search different point of the grid. Then for the one we found the best, we zoom into that part of the grid and try different values randomly again.

## Coarse to fine



**Using an appropriate scale to pick hyper-parameters**

Sometimes, for some hyper-parameter, choosing values uniformly is the best way. Like for the time when we want to choose number of hidden layers which we assume the best would be between 2 and 5. So we try 2, 3, 4, and 5 uniformly. But it is not always the case.
Sometimes picking an appropriate value in an original scale does not work well like the example below since it is uniformly spread. In the example below if we want to find the learning rate which is between 0.0001 and 1, 90% of times we are checking values between 0.1 and 1 and only 10% of times we are checking between 0.0001 and 0.1. For this reason, we use a logarithmic scale for searching the learning rate. So we divide the hyper-parameter environment to different scaled environment so that it would be way more useful in searching for the actual hyper-parameter.




For weighted averages we know that if $\beta$ was 0.9, we are averaging for the past 10 days while for 0.999 we are averaging over the past 1000 days. So again, linear environment is not useful at all. So instead we use $1-\beta$ for 0.1 to 0.001 in the logarithmic scale which is between $10^{-1}$ and $10^{-3}$.

# Hyperparameters for exponentially weighted averages

$\beta = 0.9 \quad \ldots \quad 0.999$

$10 \qquad\qquad 1000$

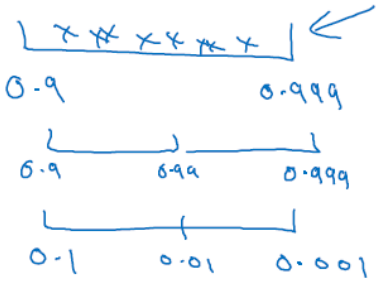
$1-\beta = 0.1 \quad \ldots \quad 0.001$

$\beta: 0.900 \rightarrow 0.9005$ } $\sim 10$

$\beta: 0.999 \rightarrow 0.9995$

$\sim 1000 \qquad \sim 2000$

$\frac{1}{1-\beta}$




$10^{-1} \qquad\qquad 10^{-3}$

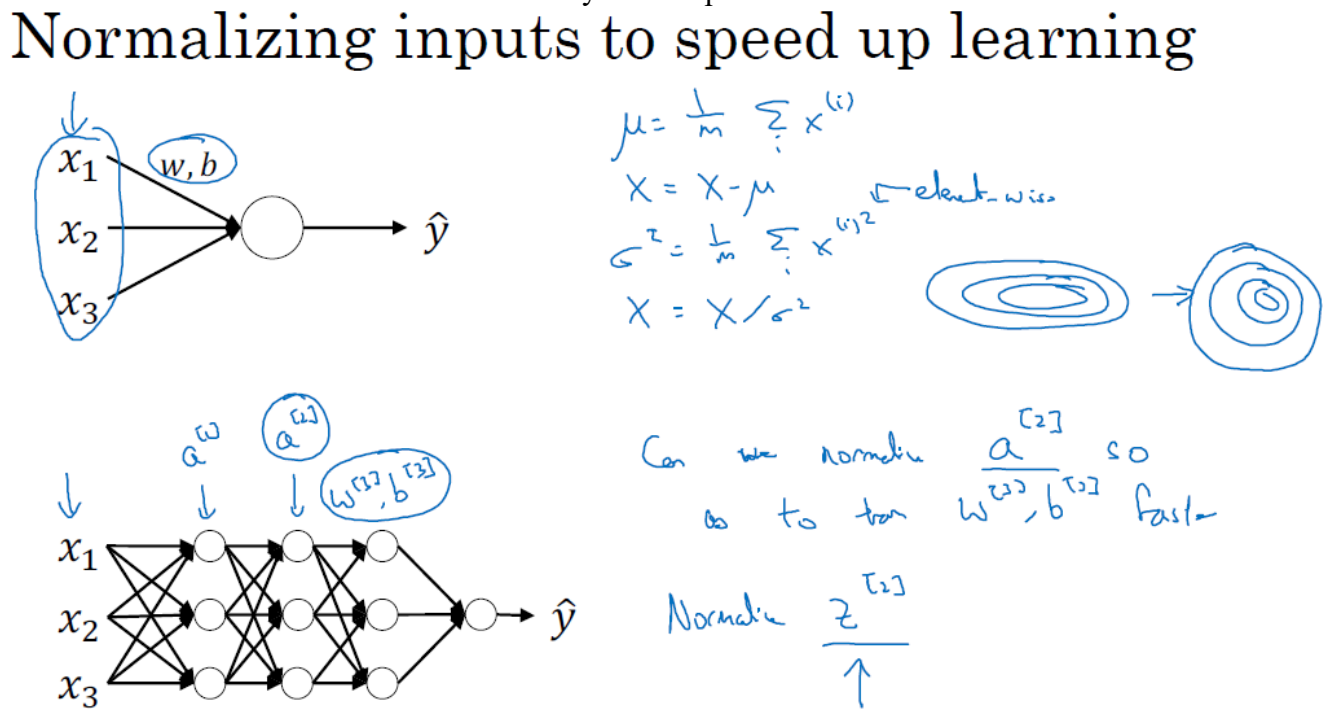
$r \in [-3, -1]$

$1-\beta = 10^{r}$

$\beta = 1 - 10^{r}$

**Normalizing activations in a network**

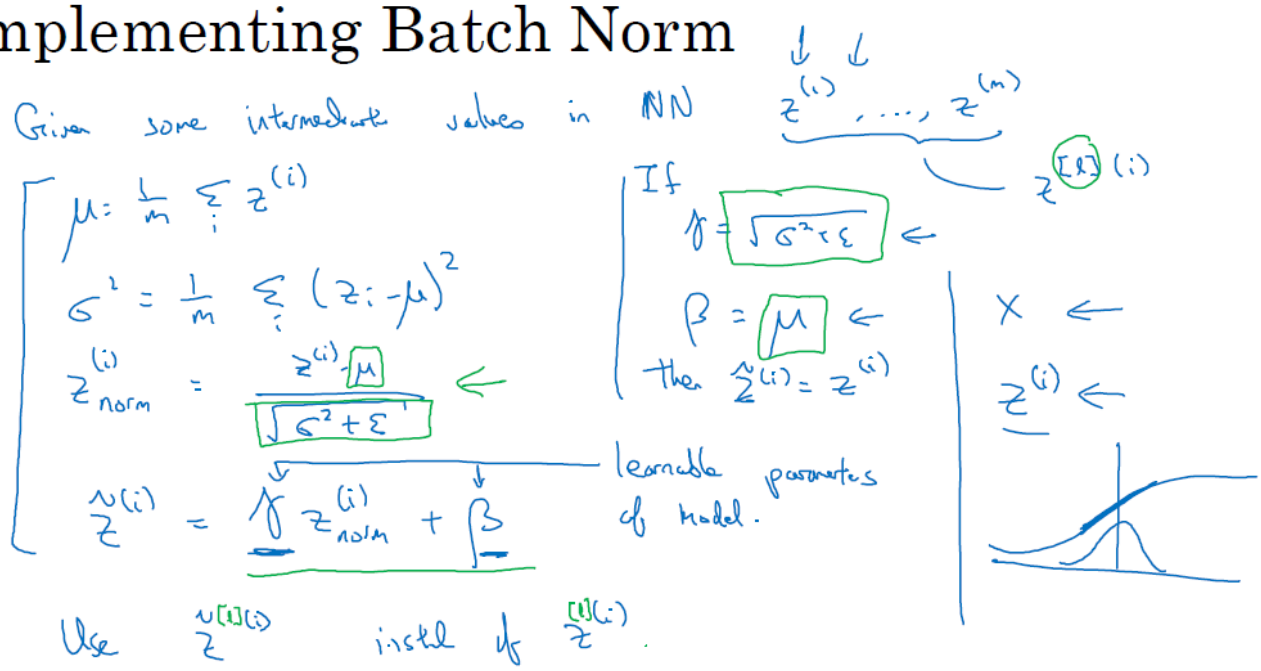
We saw it before that normalizing inputs are important to speed up the learning. So we compute the variance and mean and we normalize the input features based on them. But in deep network we have many input to layers. So two of our options is to normalize data before going into the activation function or after that. The latter is more widely used in practice.



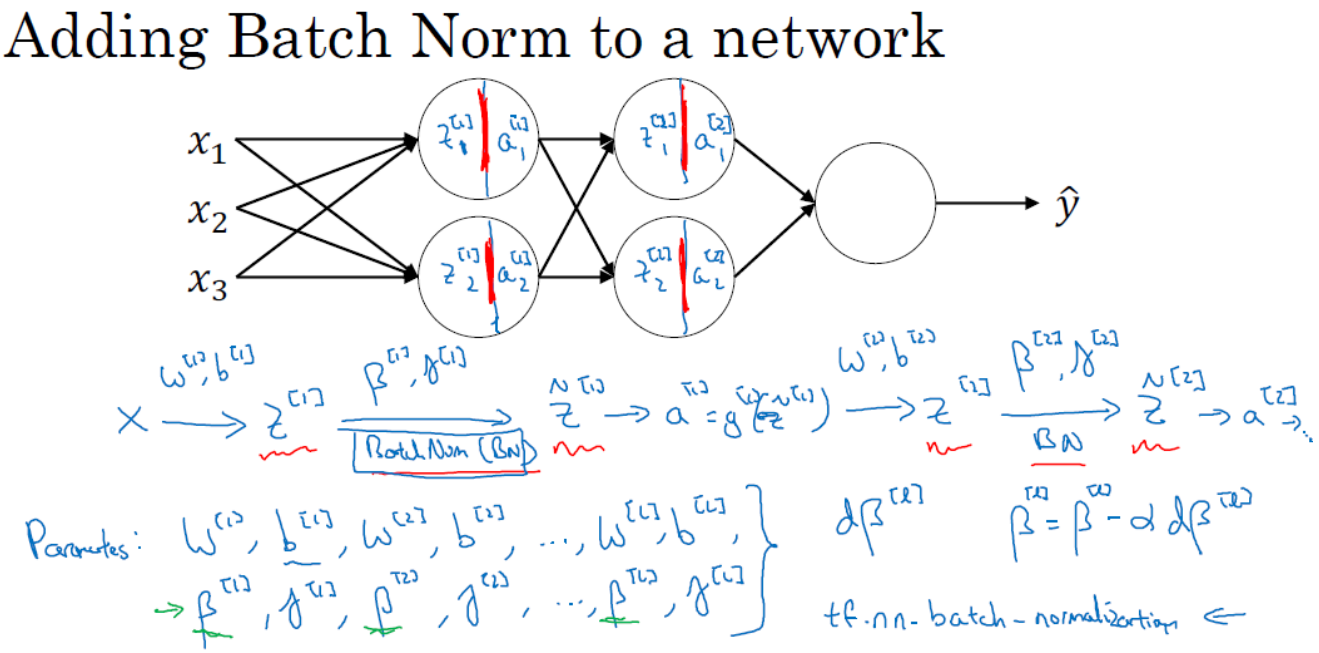

How to implement batch norm?

First we get the mean and variance for values in each layer independently and we will normalize them. It makes the model to have mean 1 and variance zero. However, we do not always want that to happen since maybe we want to use different distributions. So we will come with different $\gamma$ and $\beta$ hyper-parameters in order to control the distribution shape. It allows the data to be whatever mean and variance we want so $\gamma$ is the same as variance and $\beta$ same as mean. So for example for sigmoid, we don't want our values get too large or small because it will end up with 0 and 1. So we use a distribution which prevents that problem. They may get the same values of mean and variance of the actual values of that layer which means that the normalized values would be the same as the actual values.

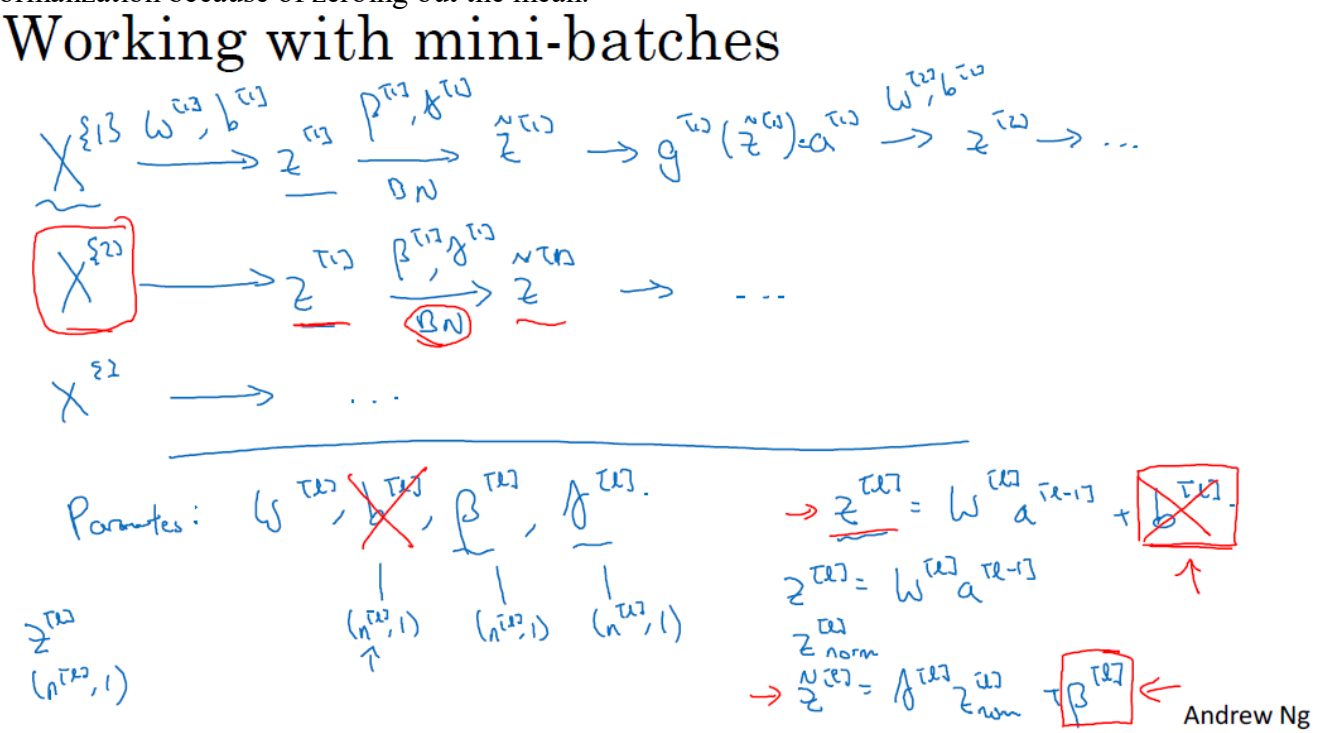
# Implementing Batch Norm



**Fitting Batch Norm into a neural network**



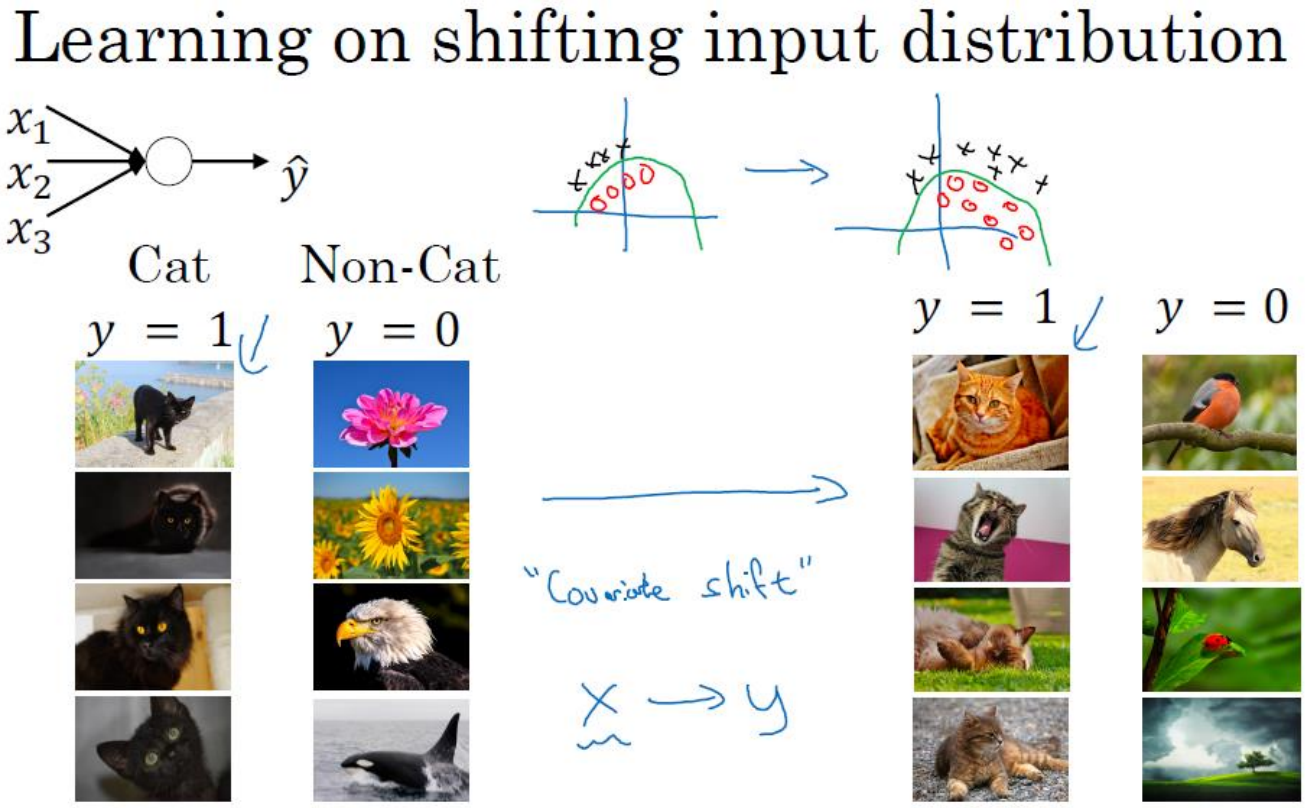

The batch norm is different for different mini-batches for each layer. The bias constant will get cancelled out when it comes to normalization. So "b" is not needed as parameters for batch normalization because of zeroing out the mean.



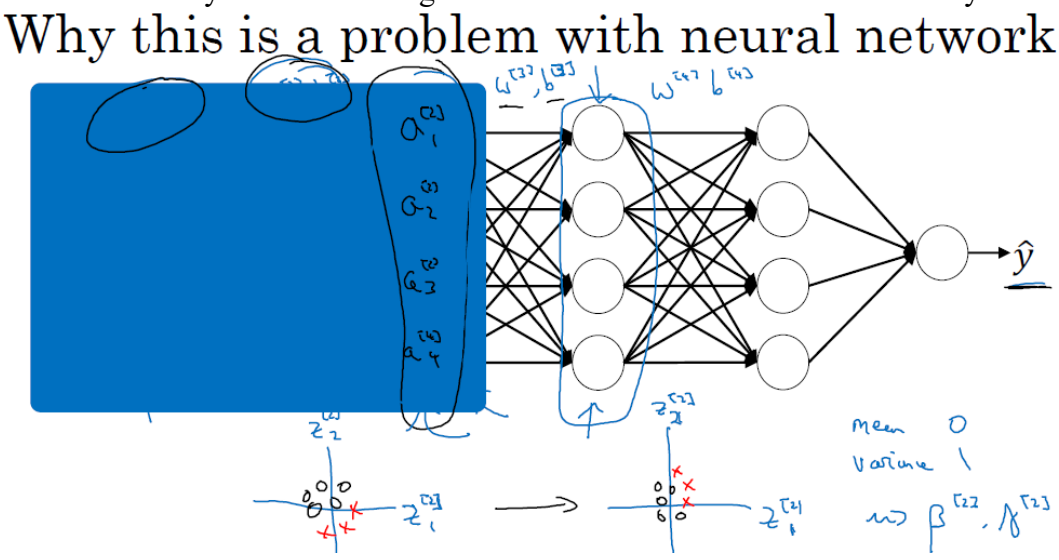

Any update algorithm works for updating parameters.

**Why does batch norm work?**

One reason is the same reason why we normalized input data.
We also need to learn the shifting input distribution without re-learning. So one thing that may happen is that the data distribution may change so that the previously learned decision boundary cannot work on the data with new distribution. This incident has been called *covariant shift*. So we do not want to retrain our network.




It is obvious that the weights and biases in the previous layers are changing so that the values of activation layers will change as well. When the values of an activation layer changes all the time, it suffers from covariant shifting. So batch norm tries to use a single distribution for that layer so that the values in that layer will not change a lot as their mean and variance will stay the same.



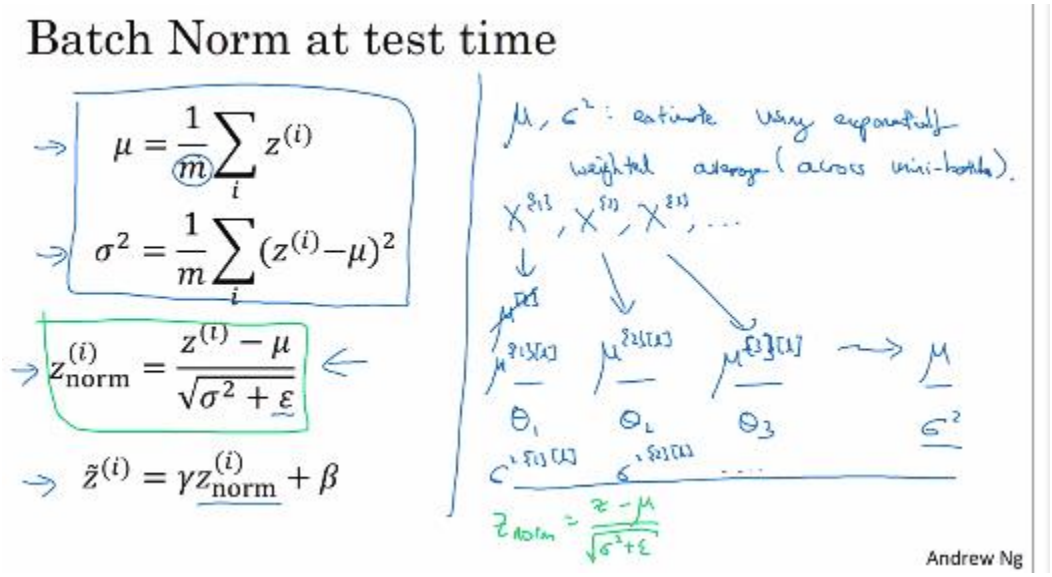
Batch norm can also be used as regularization but practically it is not known for that and try to not use that as a regularizer because its effect is so less than expected.

# Batch Norm as regularization

- 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.

**Batch norm for testing**

Mean and variance is getting calculated for every single mini-batch in training time. But for testing we do not have mini-batches so we use exponential weighted average across all mini-batches in the training time and we will use it for the test time.



# COURSE 3
# WEEK1

**Why ML strategy?**

There are many options we can try so that the model may improve. But which ones shall we try and why some of them worth to try and some don't. For example, we may put a 6 months effort on collecting more data while it may not help at all in increasing the accuracy.

## Motivating example

90%

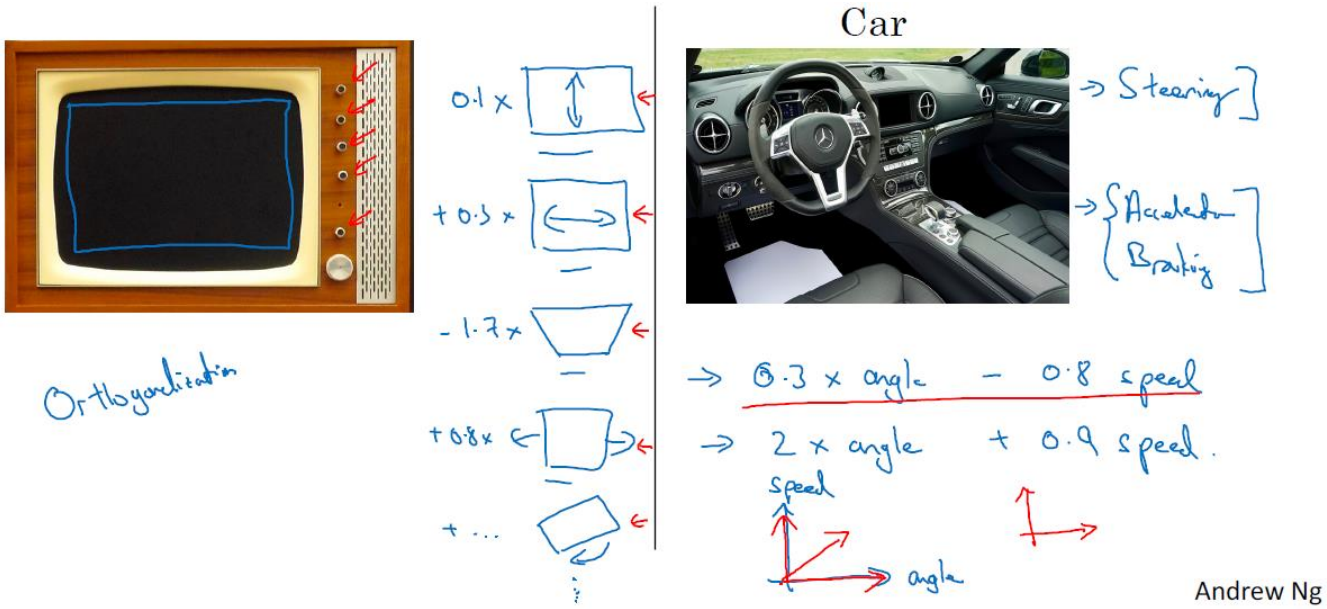
Ideas:

- Collect more data
- Collect more diverse training set
- Train algorithm longer with gradient descent
- Try Adam instead of gradient descent
- Try bigger network
- Try smaller network
- Try dropout
- Add $L_2$ regularization
- Network architecture
  - Activation functions
  - # hidden units
  - …

Andrew Ng

**Orthogonalization**

One of the things happening is about the most effective machine learning people which are very clear eyed about what to tune in order to try to achieve one effect. This is a process we call orthogonalization. Like old TV knobs for tuning the video. In this context, orthogonalization is existence of different knobs in which each one is responsible for changing the width, height, and many other things. Each knob does not interfere with work of the other one. In the figures below, we can see orthogonalization for car and TV.




We expect for ideal case:

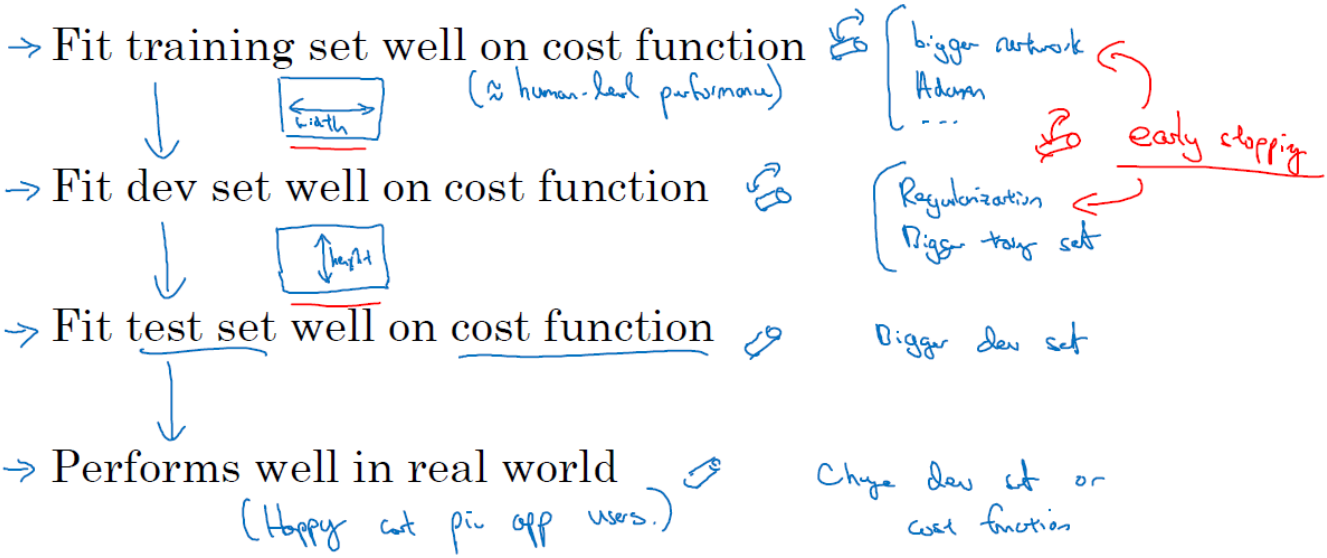
- training set fits well on cost function
- → then *we hope* dev set fits well on cost function
- → then *we hope* test set fits well on cost function
- → then *we hope* the result can perform well in real world

However, for each state, it may not work as expected, so we can play with its knob:

- training set does NOT fit well on cost function
    1- maybe bigger network
    2- better optimization algorithm
    3- …
- dev set does NOT fit well on cost function but training set fits
    1- using regularization
    2- bigger training set to make sure the algorithm generalize better
    3- …
- test set does NOT fit well on cost function but dev set fits
    1. bigger dev set might be needed due to overtuning on dev set
    2. …
- does not perform well on real world but works great on test set
    1- changing dev set
    2- changing cost function
    3- …

Note:
Try not to use early stopping. Why?
Because it both works on how well we are doing on the training set and dev set. The former happens since we stop early on training set and we are not letting the model to be trained on the training set very well. The latter happens since it improves the dev set performance. It is less orthogonolize since it is a knob to control two things simultaneously.

# Chain of assumptions in ML



**Single number evaluation metric**

For choosing the best model, we have to choose the correct metric which is the best if it can be a single valued number.

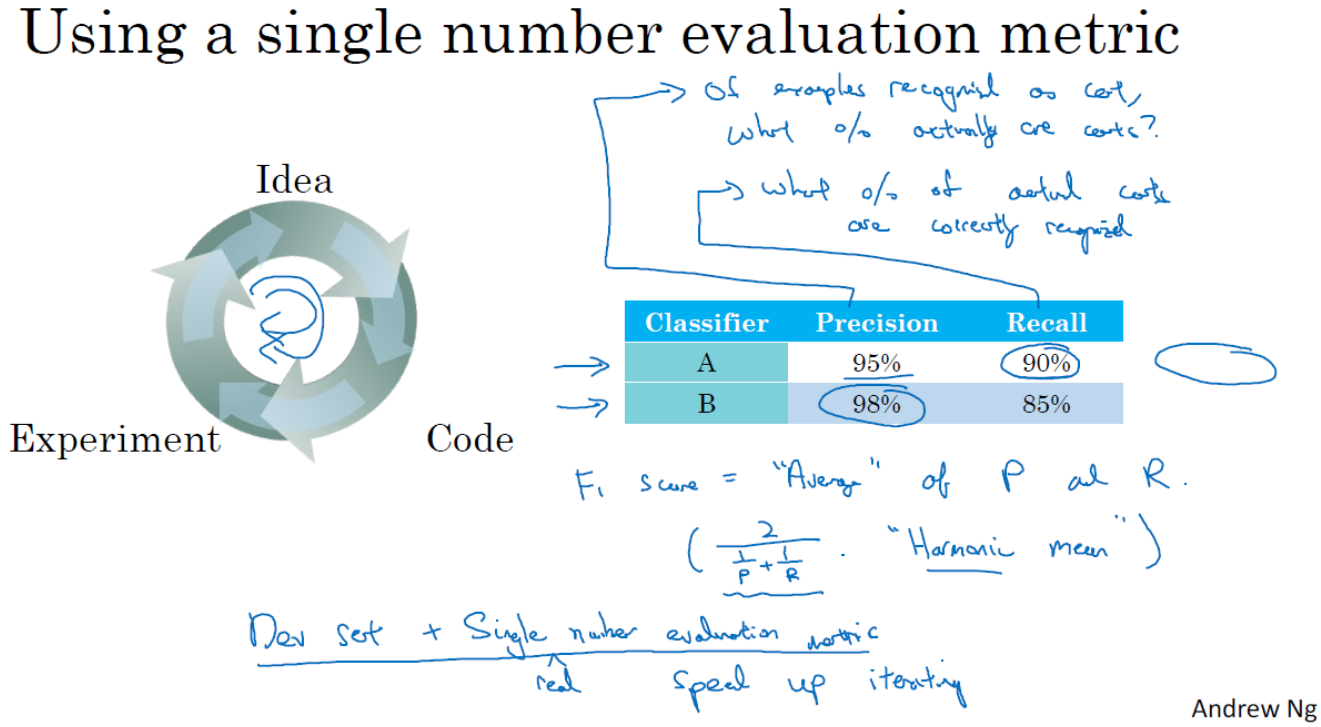
As an example, in the slide below, we are looking at *precision* and *recall:*

- precision: Of the examples that our classifier recognizes as cat, what percentage are actually cats
- recall: from all of cats images, what percentage of actual cats are correctly recognized

We can see in this example, recall of the classifier A and precision of the classifier B outperform the others. So which one now is better? Classifier A or B?
For this example, we can use *F1 score* which is the *harmonic mean* of precision and recall. Then we will end up with one single metric, with one we can iterate faster as for our dev set we know what we want to increase and what is exactly important for us. In this case it is F1 score. The metrics should always be a single real number for evaluation.

For the case of F1 score with harmonic mean we have:

$$F_1\ Score = \frac{2}{\frac{1}{P}+\frac{1}{R}} = \frac{2PR}{P+R}$$




As another example we have several classifiers, each with values of error percentage for different countries. It is really hard to say which one of the classifiers works the best. A technique would be taking the error average of each classifier.

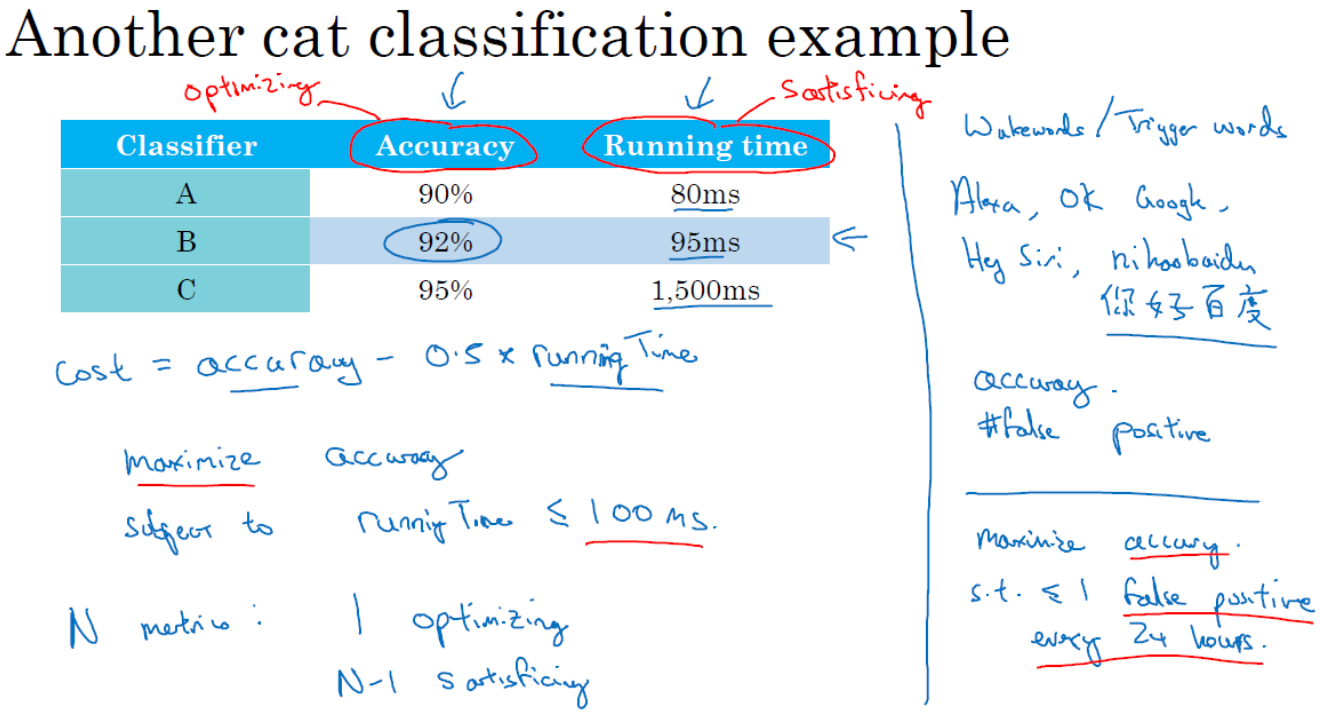
| Algorithm | US | China | India | Other |
|---|---|---|---|---|
| A | 3% | 7% | 5% | 9% |
| B | 5% | 6% | 5% | 10% |
| C | 2% | 3% | 4% | 5% |
| D | 5% | 8% | 7% | 2% |
| E | 4% | 5% | 2% | 4% |
| F | 7% | 11% | 8% | 12% |

**Satisficing and Optimizing metric**

Sometimes it is hard to reach a single real number value when there are many metrics involved in the problem.
In the example below, we have two metrics, one is the accuracy of the model and the other is the running time. It is hard to come up with a single value for accuracy and running time, writing it in a linear fashion. So instead we take another approach:

- We want to maximize accuracy (→ optimizing) while the *running time is less than X* (→ *satisficing metric)*.

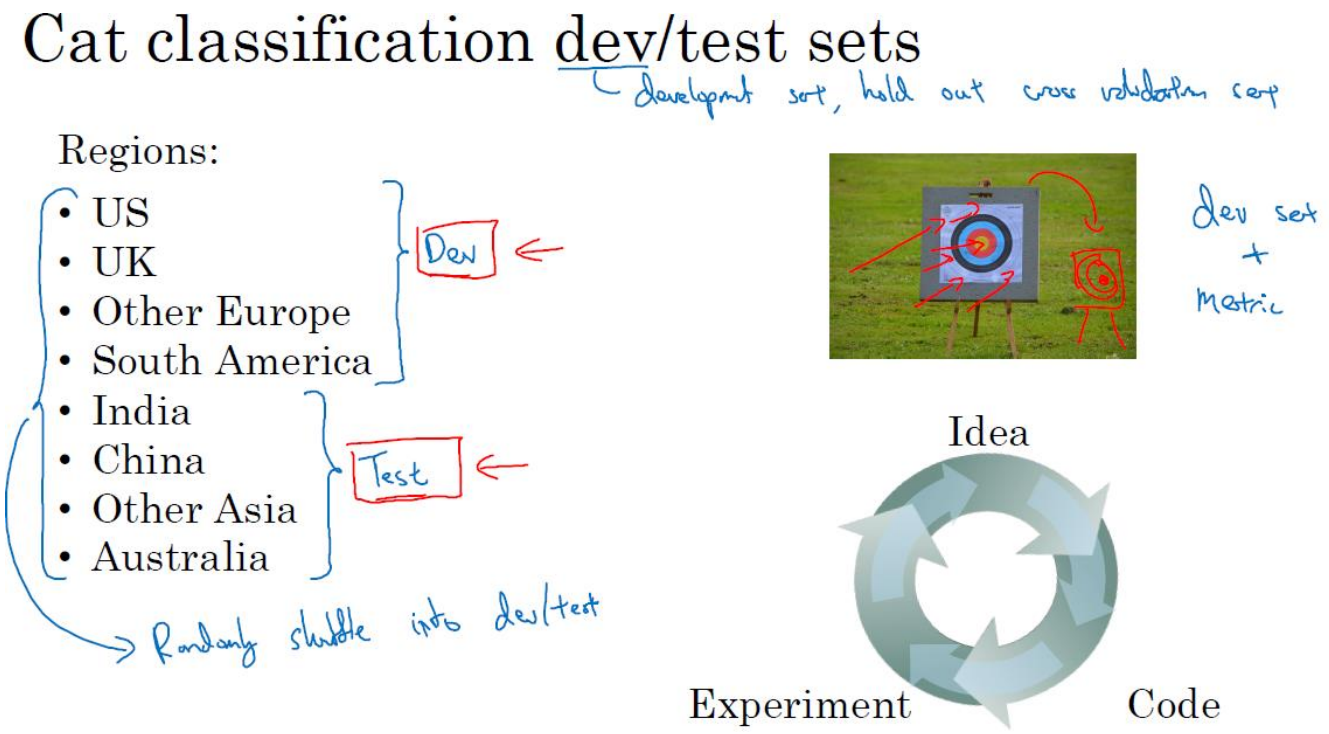
So for N metrics, one of them will be optimization variable and N-1 will be satisficing variables.
Instead of writing a formula to reach a real number, we write it like this as the optimization problem.



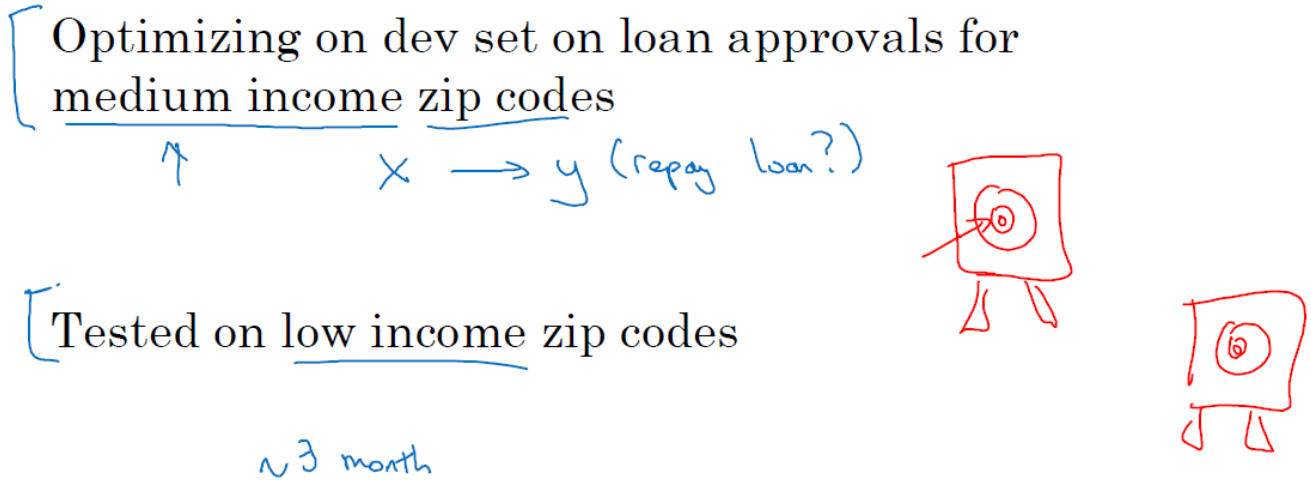

Another example would be accuracy of wake up detection: when someone says the trigger word, accuracy of false positive is also important ➔ maximizing accuracy + at most one false positive in every 24 hours of operation.
In this example, the first one is optimizing metric and second is satisfice metric.

**Train/dev/test distributions**

*How to setup dev and test set*?
The problem here is that, we have data from different countries. We may spend months on training on the dev set so that we can target the bull's eye. However, we may finally hit it but when we go to the test set, we see that the distribution is different and the result will not well generalized. In this scenario we have to change the bull's eye completely which means months of waste of time.
So, they must be from the same distribution, not each one from different parts of the world. A way is to shuffle the dev and test set which are from different distribution.



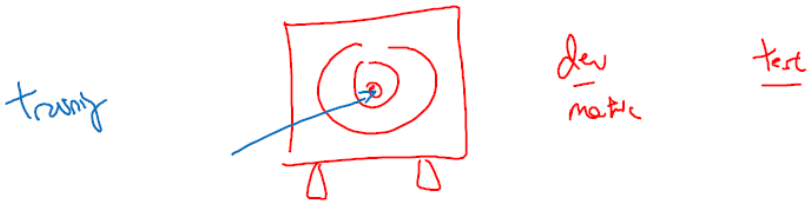

Another example is shown in the slide below.



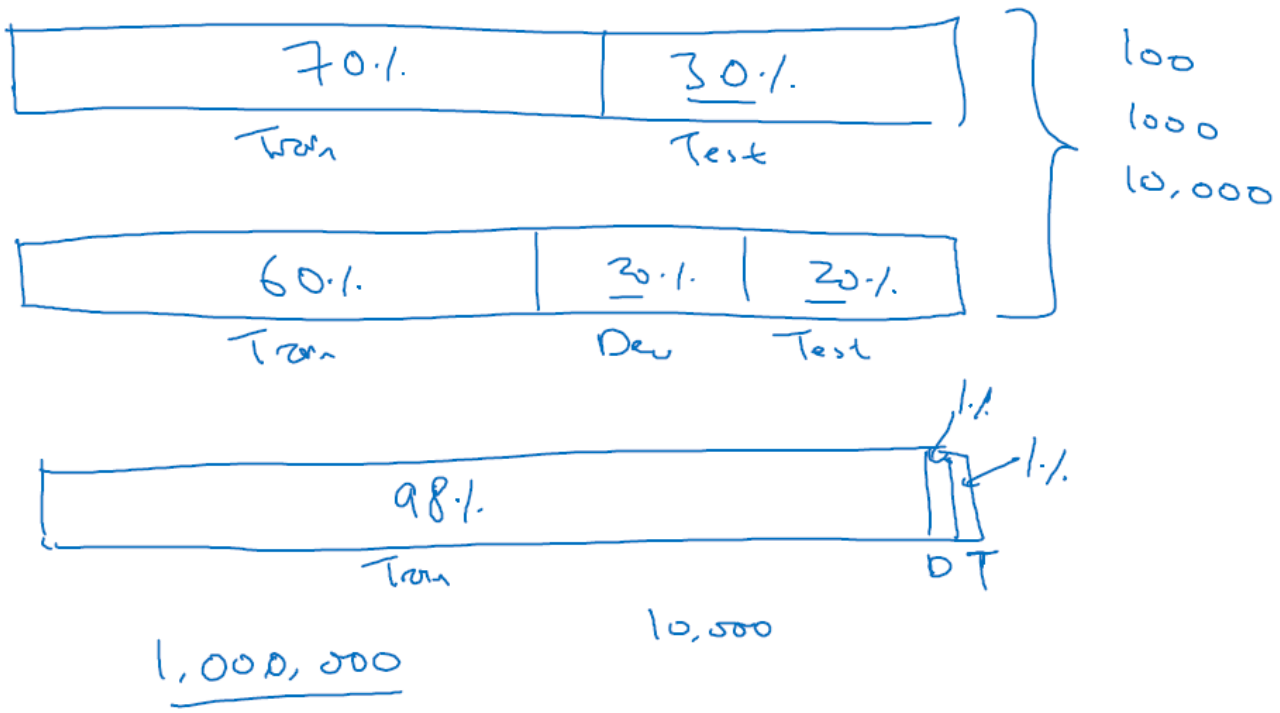
# Guideline

Same distribution

Choose a dev set and test set to reflect data you expect to get in the future and consider important to do well on.



**Size of the dev and test sets**



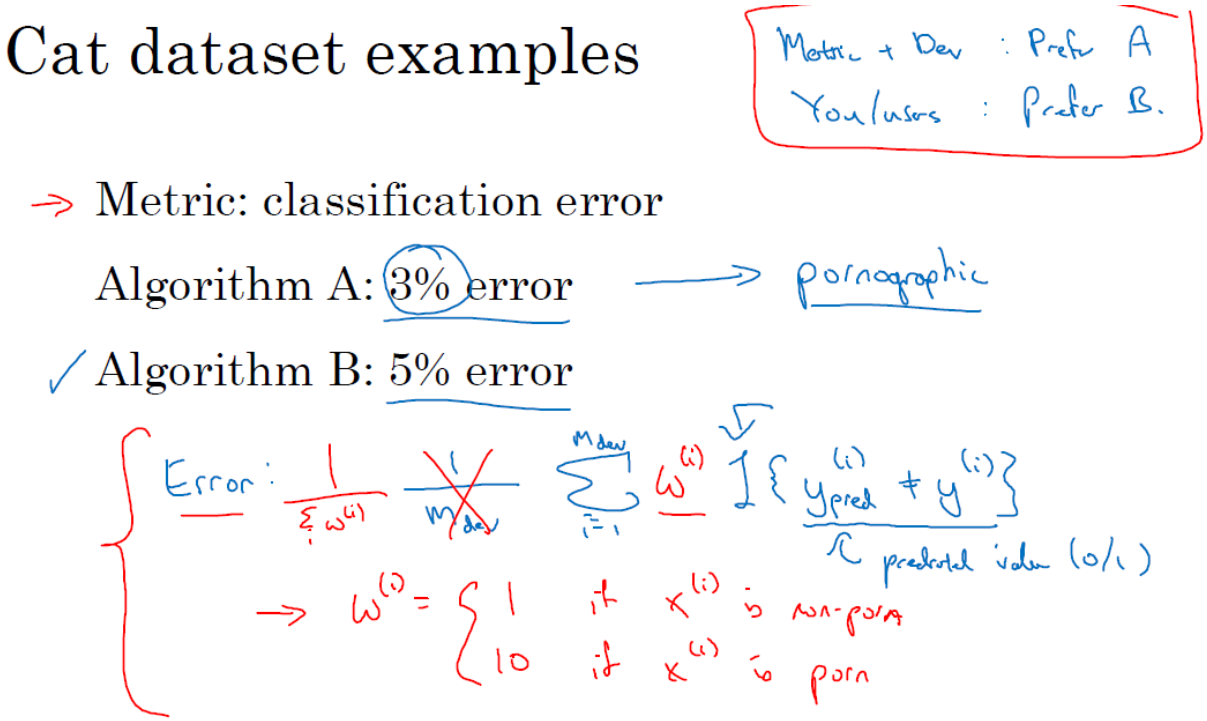

For the size of test set:

- set your test set to be big enough to give high confidence in the overall performance of your system

Not having test set is ok while having train and dev sets. But it is always recommended to have all three instead of only train and dev sets.

**When to change dev/test sets and metrics**

Algorithm A has some pornographic images which is not acceptable while Algorithm B having higher error with no pornographic images in. So in this example, the metric and dev set prefer algorithm A, while we prefer B. Consequently, we have to change the bull's eye.
Changing metric by giving weights to pornographic images and normal ones.




So the whole goal here with respect to orthogonalization would be:

1. define a metric to evaluate classifiers → placing the target
2. doing well on this metric → aim to shot at the target (maybe by changing the cost function)

As another example, shown in the slide below, sometimes, we are doing well on dev/test set, but the goal is to do well on user images which might be blurry and from different distribution of what we trained our model based on.

# Another example

Algorithm A: 3% error

Algorithm B: 5% error

→ Dev/test



→ User images



If doing well on your metric + dev/test set does not correspond to doing well on your application, change your metric and/or dev/test set.

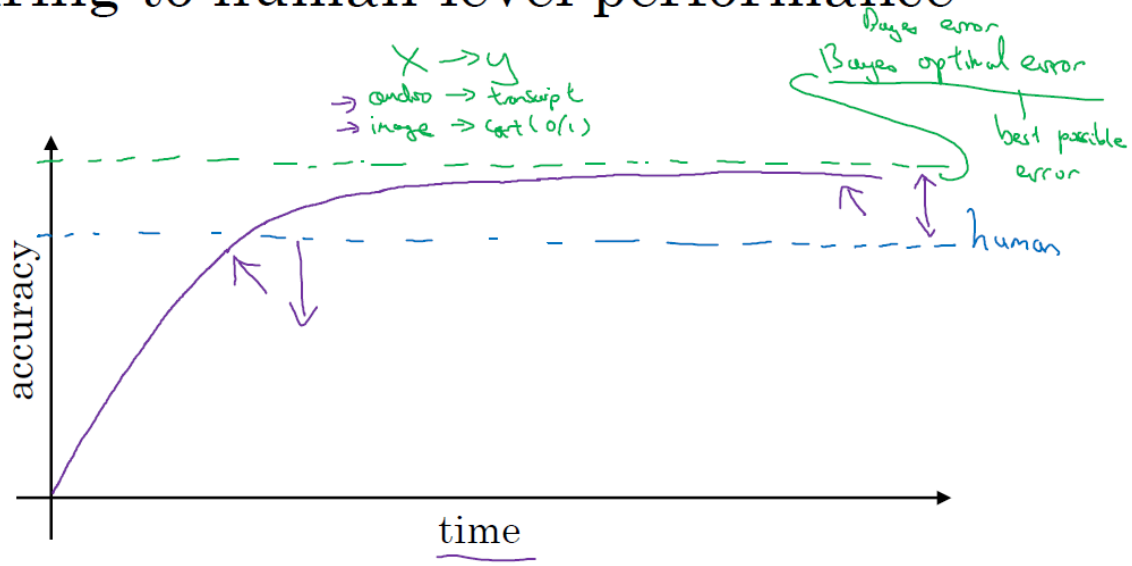
**Why human-level performance?**

In big data era, the accuracy of machine learning models can reach the human-level performance. Moreover, these models try to mimic the human way of solving the problems. This is the reason recently models are being compared with human-level performance.

The accuracy of machine learning models can beat the human accuracy but they cannot surpass a theoretical limit, "Bayes Optimal Error" which has the highest accuracy.

In the below diagram, we can see that when the model reaches the accuracy of human-level performance, its momentum decreases highly or maybe never reaches to that. Because the difference between the human-level performance is not much different than the Bayes error. Moreover, as long as the model didn't reach the human-level performance, they are some ways and tools to improve them, but it is not the case when the model passes the human-level performance.

## Comparing to human-level performance



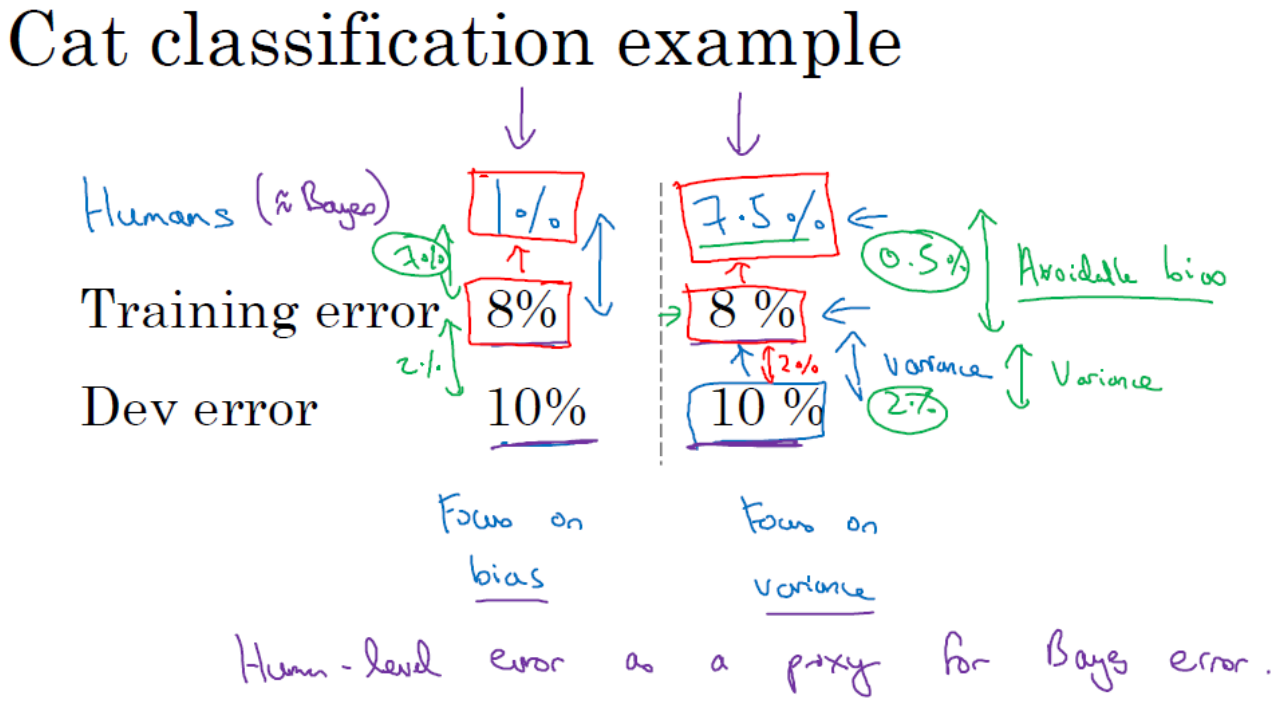

## Why compare to human-level performance

Humans are quite good at a lot of tasks. So long as ML is worse than humans, you can:

- Get labeled data from humans.
- Gain insight from manual error analysis: Why did a person get this right?
- Better analysis of bias/variance.

**Avoidable bias**

Previously, we assumed the human error is better than the Bayes error which is 0%, however it is not true. → we have an avoidable bias.
Human-level error is a proxy for Bayes error in most of the cases.
Difference between the training error and Bayes error is "avoidable bias" since we cannot do better than Bayes error. We need to get as close as possible to the Bayes Error and then try to reduce difference error between training and dev error which is variance.



**Understanding human-level performance**

# Human-level error as a proxy for Bayes error

Medical image classification example:

Suppose:

(a) Typical human ..... 3 % error

(b) Typical doctor ..... 1 % error

(c) Experienced doctor ..... 0.7 % error

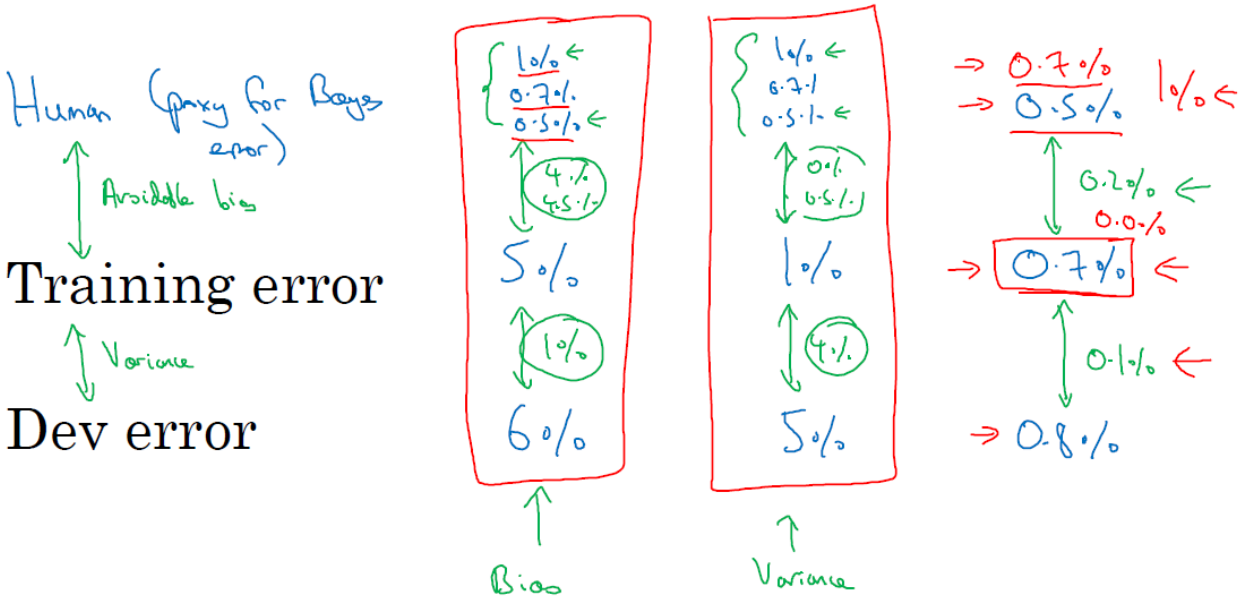
(d) Team of experienced doctors .. 0.5 % error

What is "human-level" error?

Choice D is close to Bayes error so we can assume it as human-level error. We have also this concept that if we pass the average doctor diagnosis accuracy, it is human-level error. In that case choice B is human-level error but it is not a proxy to Bayes error.

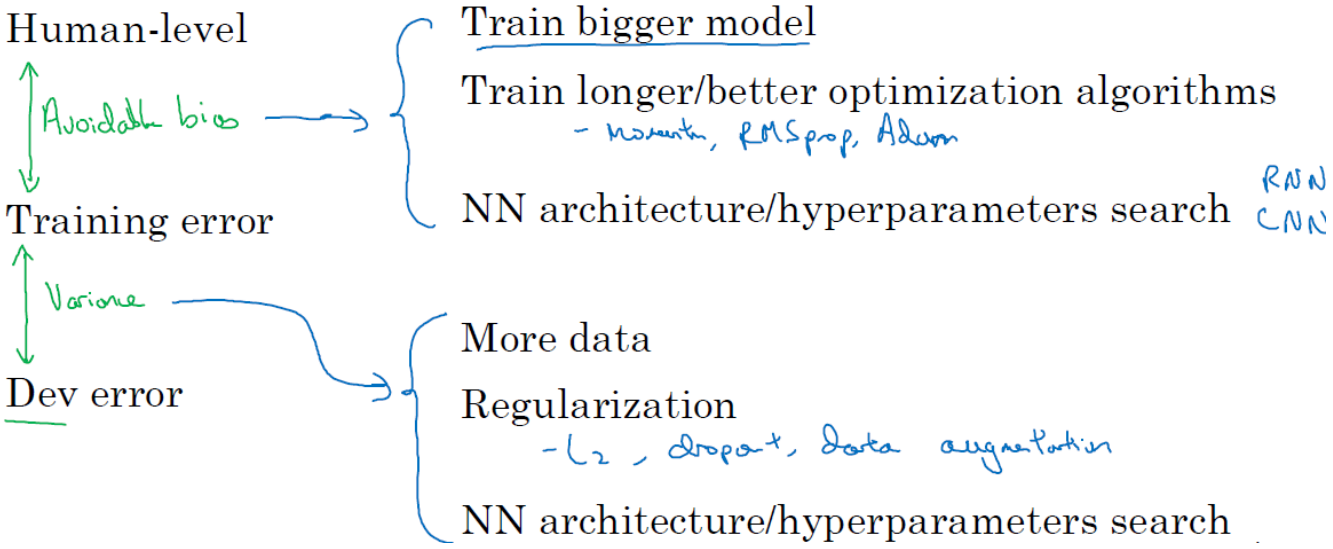
In the example below, in the left column, it doesn't matter which proxy we choose as far as its difference with training error is so much higher; so we want to reduce bias. The next one same applies for variance. Finally, the last one, we should be concerned about the proxy we use as it affects which one of bias or variance we should focus on.

# Error analysis example



There are many cases that computer can surpass human-level performance due to limitation of human in learning patterns. Like voice recognition.

# Reducing (avoidable) bias and variance



# Course 3
# Week 2

**Carrying out error analysis**

Does working on something new worth the effort and time we put on it?
So let's say that we want to come up with a dog classifier. In the *error analysis* phase, we see around 100 pictures of dogs are mislabeled in the dev set. Then we count up how many of the mislabeled pictures are dog pictures. If it is only 5% of them, it means that if the error of dog classifier was 10%, after fixing the mislabeled data in dev set, the error goes down to 9.5% (it is relative error). On the other hand, if 50% of mislabeled dev set examples are dog, from 10% error we can reduce is to 5%. In other words, we can find the ceiling in this way and understand whether it worth our time or not.

# Look at dev examples to evaluate ideas

90% accuracy
→ 10% error

Should you try to make your cat classifier do better on dogs? ←

Error analysis: → 5-10 min

- Get ~100 mislabeled dev set examples.
- Count up how many are dogs.

"ceiling"

→ 5% → 5/100 → 10% → 9.5%

→ 50% → 50/100 → 10% → 5%

# Evaluate multiple ideas in parallel

Ideas for cat detection:

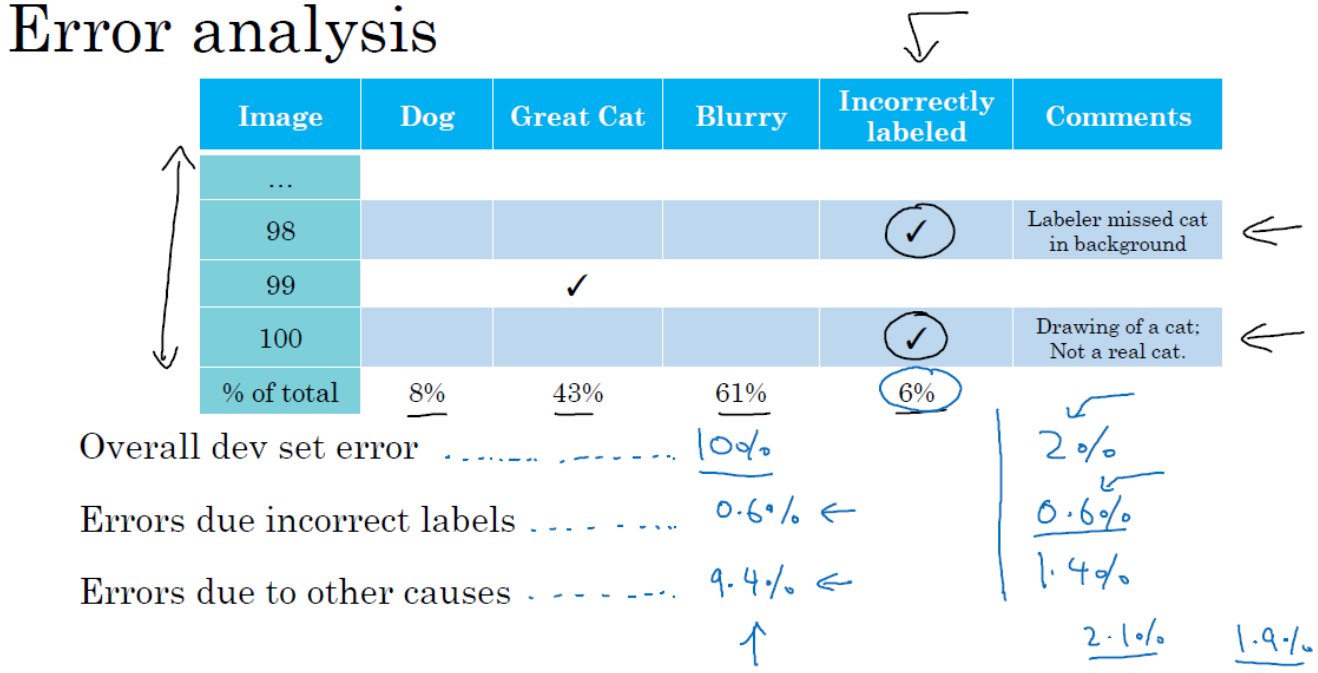
- Fix pictures of dogs being recognized as cats ←
- Fix great cats (lions, panthers, etc..) being misrecognized
- Improve performance on blurry images ←

| Image | Dog | Great Cats | Blurry | Instagram | Comments |
|---|---|---|---|---|---|
| 1 | ✓ | | | ✓ | Pitbull |
| 2 | | | ✓ | ✓ | |
| 3 | | ✓ | ✓ | | Rainy day at zoo |
| ⋮ | ⋮ | ⋮ | ⋮ | | |
| % of total | 8% | 43% | 61% | 12% | |

**Cleaning up incorrectly labeled data**

As far as the number of samples are too much and the errors are not much, for the training set we can say that as DL algorithms are quite robust to random errors in the training set, we do not need to correct the mislabelled data. However, they are not robust at all to the systematic errors.

For dev set and test set → in the error analysis table we add a column of incorrectly labeled.
If it makes a significant difference to your ability to evaluate algorithms on your dev set, so fix it.
But if its difference is not significant, just let it go.
For the fixing of mislabelled data, as it is time consuming, in the left example does not help much; but, in the right one it does help.




Now some guidelines in correcting the mislabelled data:

1. First bullet: dev and test must come from the same distribution. So everything we do with dev set, same must be done on test set.
2. Third bullet: training set may come from slightly different distribution, but as far as dev and test sets are from the same distribution, it is ok as the DL models are quite robust to slight changes in distributions.
3. Second bullet: the examples of correct prediction might get wrong by changing the correcting a label. But if the model is accurate enough, it may not happen very often or if it happens, it is not much prevalent.

# Correcting incorrect dev/test set examples

- Apply same process to your dev and test sets to make sure they continue to come from the same distribution
- Consider examining examples your algorithm got right as well as ones it got wrong.
- Train and dev/test data may now come from slightly different distributions.

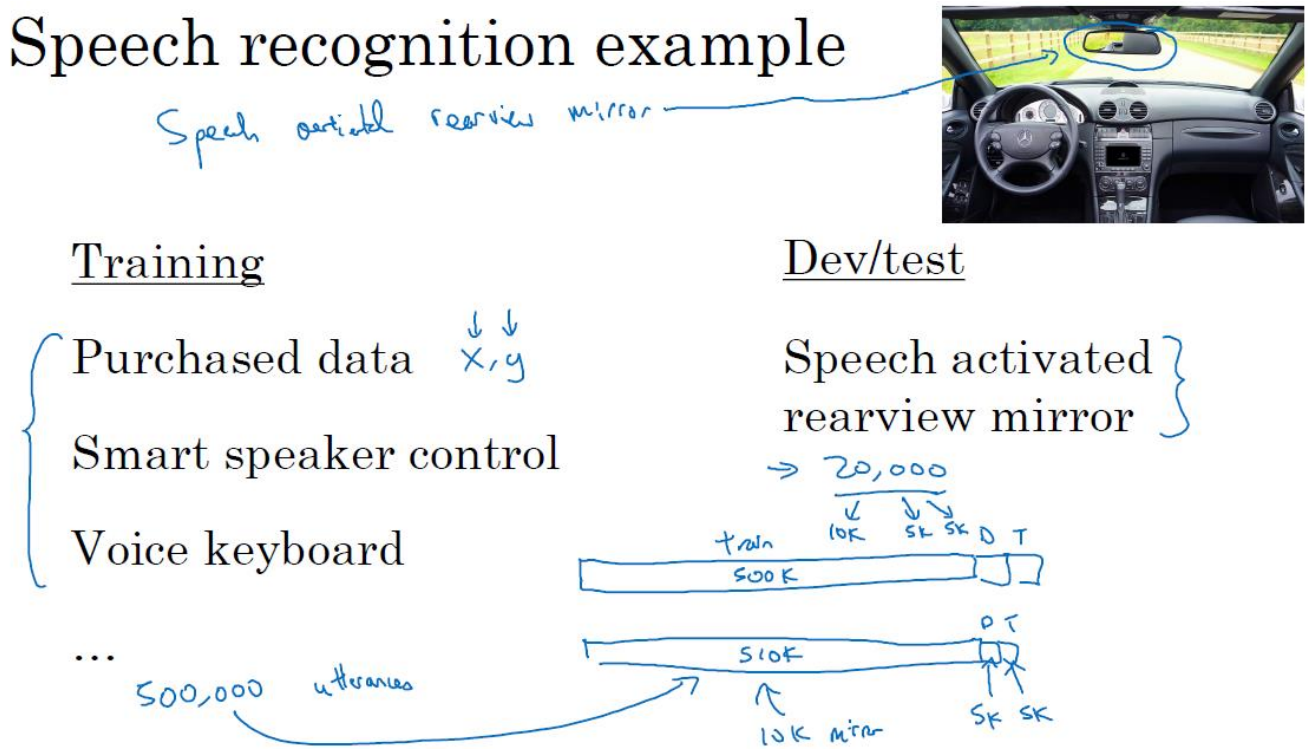
**Build your first system quickly, then iterate**

1. Quickly setup dev/set and metric
2. Build initial system quickly
3. Use bias/variance analysis and error analysis to prioritize next steps.

This is for systems that we do not have much literature for it. If we have some, then it is ok to make a more complex system.

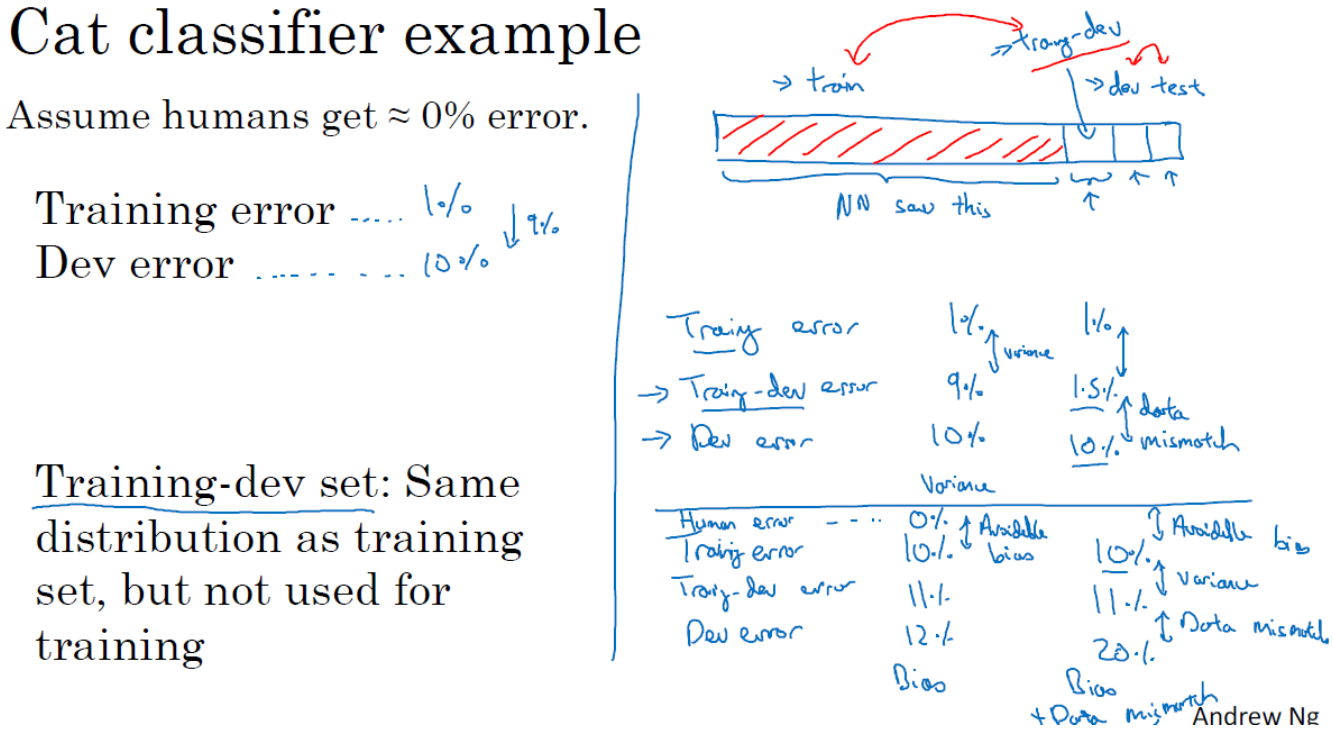
**Training and testing on different distributions**

We have some pictures from web and some from mobile app. We want our model works the best for the mobile app. If we combine them and shuffle them, then split it to train, dev, and test sets, our data in dev and test might be more about web rather than mobile, so the target is not right due to different distributions of dev, test, and train sets although shuffling them urge them to come from the same distributions. If dev and test are from different distributions that is also wrong.
ANSWER is to use mobile app pictures for dev and test and put the remaining for the training set. ➔ in the long run it works better although the distributions of training with the rest is different but we come up with techniques so that the model works the best on the dev set.

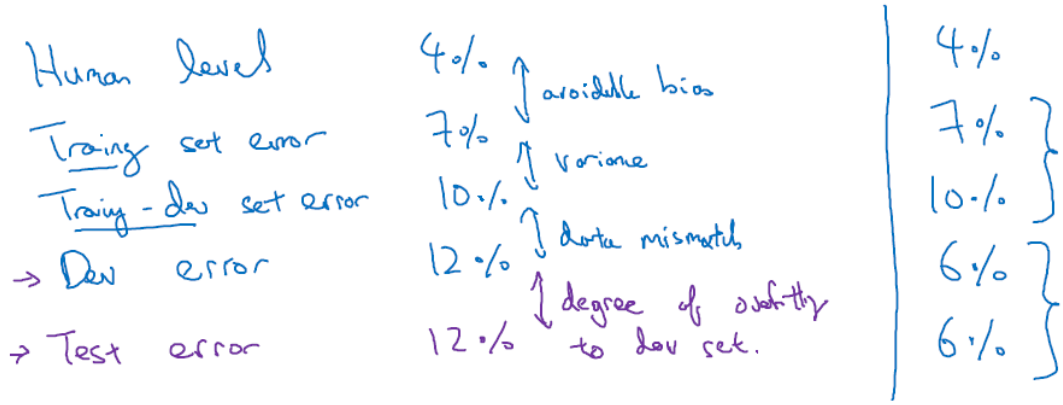


**Bias and Variance with mismatched data distributions**

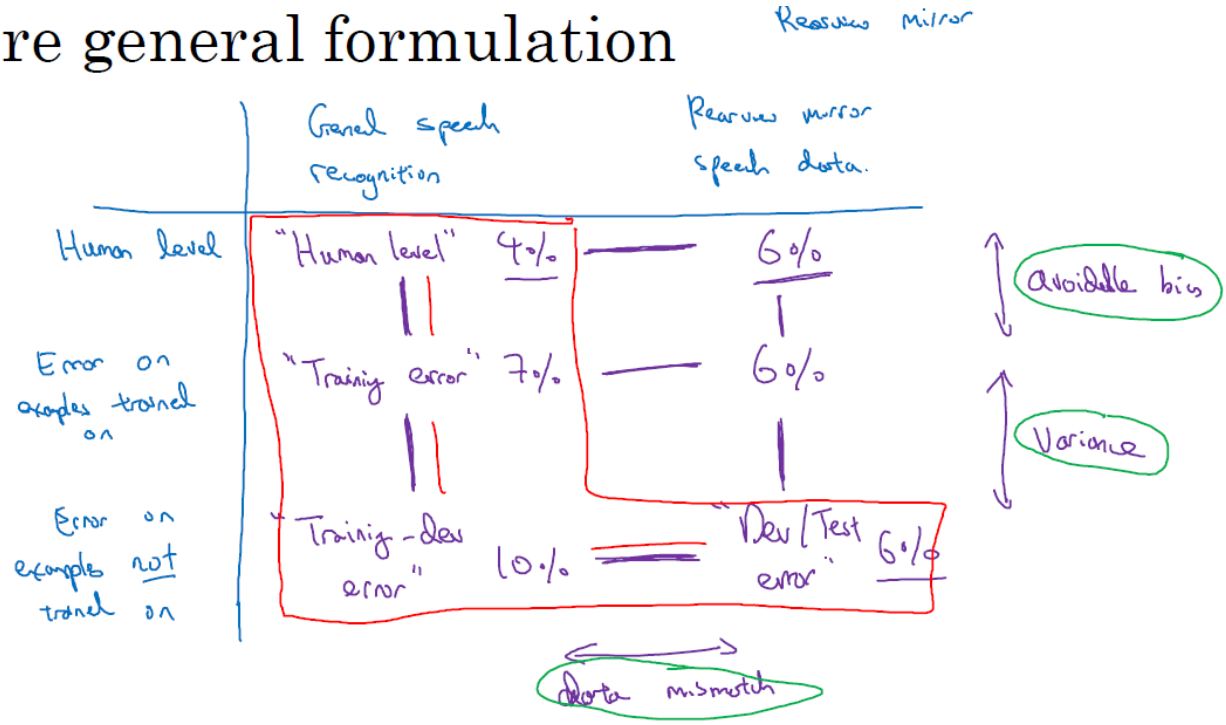
If training error is low, but dev error is high, the reason might be due to different distributions of training and dev or maybe the dev set is harder for the model to recognize. So we use a training-dev set from the training distribution as the new dev set. Then we compare the results on dev set and training-dev set. Now, if the difference of training and train-dev set is high, it means the problem is only variance because there is no difference in distributions anymore. If the difference of train-dev set and dev set is high, we call it data-mismatch problem and happening due to different distributions.




The errors on dev and test set can go down which means their examples are so much easier for the learner to recognize than the training set examples.



# More general formulation



**Addressing data mismatch**

There is no systematic way to handle it but what we can do is:

# Addressing data mismatch

- Carry out manual error analysis to try to understand difference between training and dev/test sets

  E.g. noisy – car noise  street numbers

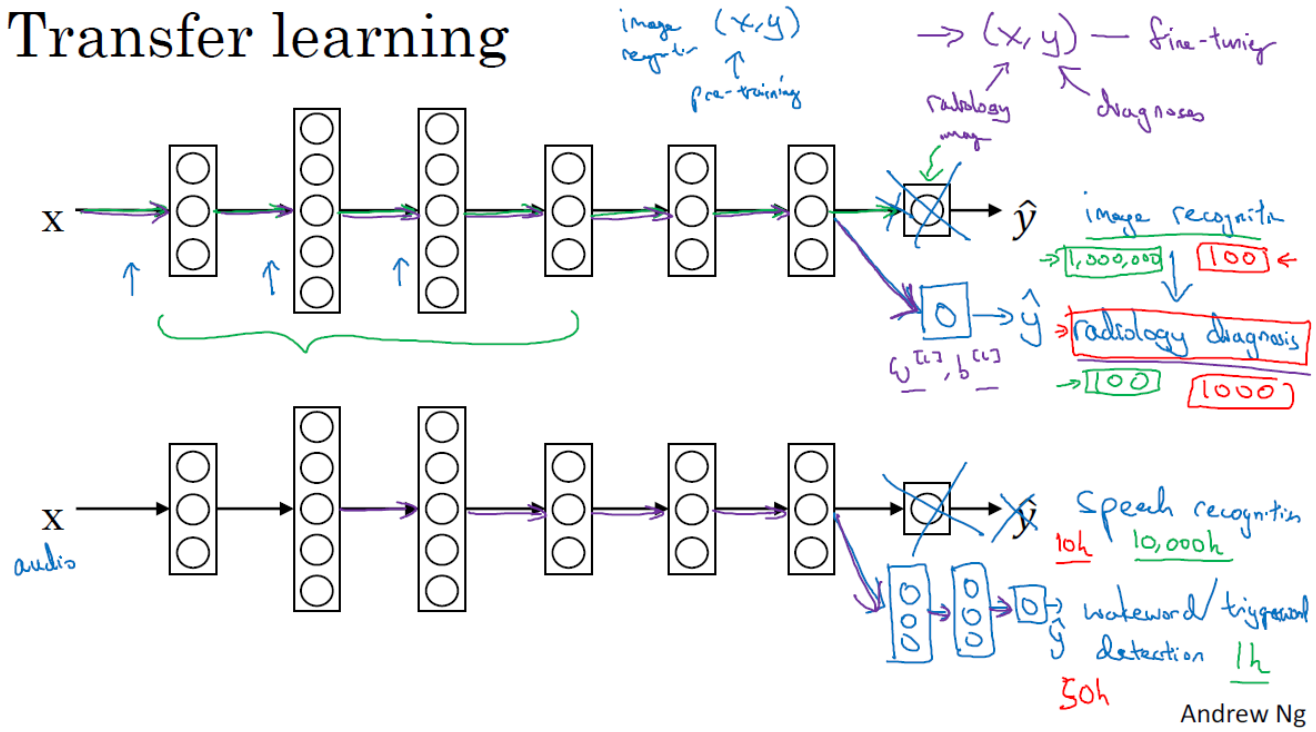
- Make training data more similar; or collect more data similar to dev/test sets

  E.g. Simulate noisy in-car data

One of the ways to make it more similar is creating artificial data synthesis. E.G. adding car noise to a sentence to make it more similar to dev data with car noise.

**Transfer learning**

We might want to take advantage of trained weights to use it on another similar task that we do not have much data on it. So we reinitialize the last layer (for example) of the NN weights and run it for the new data set. Some of its knowledge like detecting edges and some shapes are the same, so we can take advantage of them.




Transfer learning makes sense when:

## When transfer learning makes sense

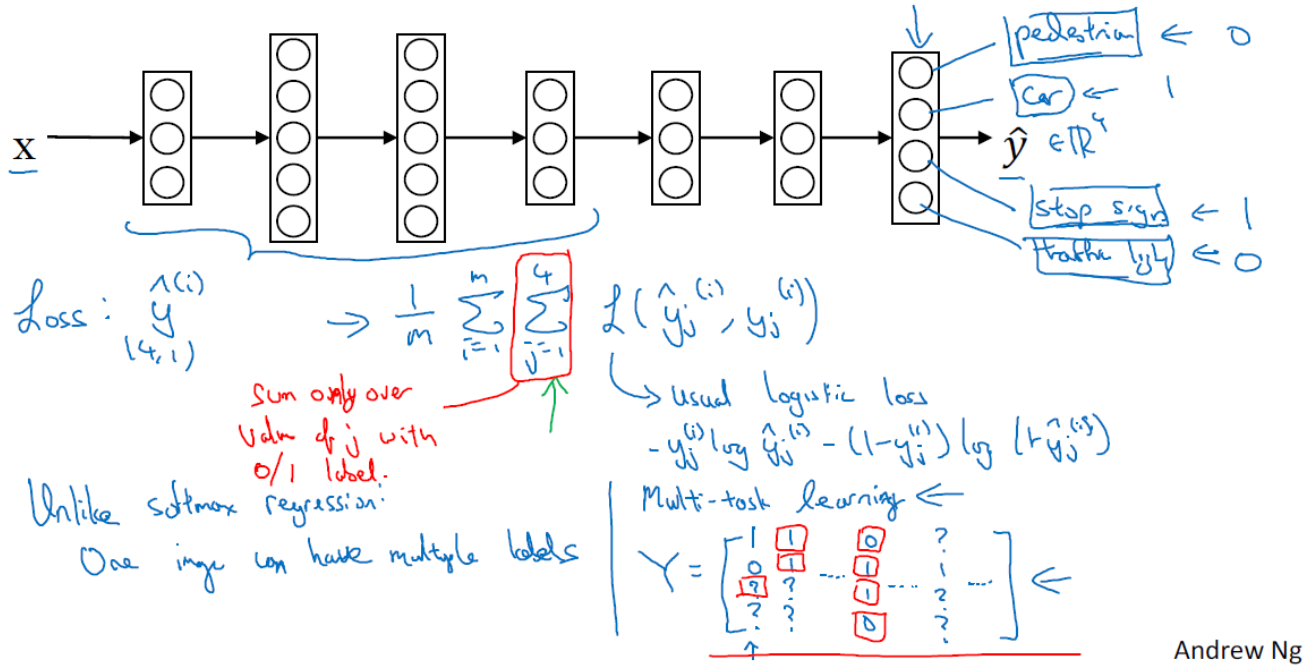
Transfer from A → B

- Task A and B have the same input x.
- You have a lot more data for Task A than Task B.
- Low level features from A could be helpful for learning B.

**Multi-task learning**

Solving several problems in a single architecture NN like a multi-label task → 4 different NNs
Note that the last layer can have multiple labels, so it cannot be Softmax.
Note that it is also acceptable to not have all the labels

# Neural network architecture

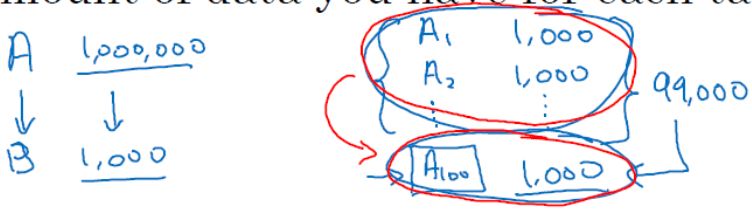



Andrew Ng

It makes sense when:

# When multi-task learning makes sense

- Training on a set of tasks that could benefit from having shared lower-level features.
- Usually: Amount of data you have for each task is quite similar.



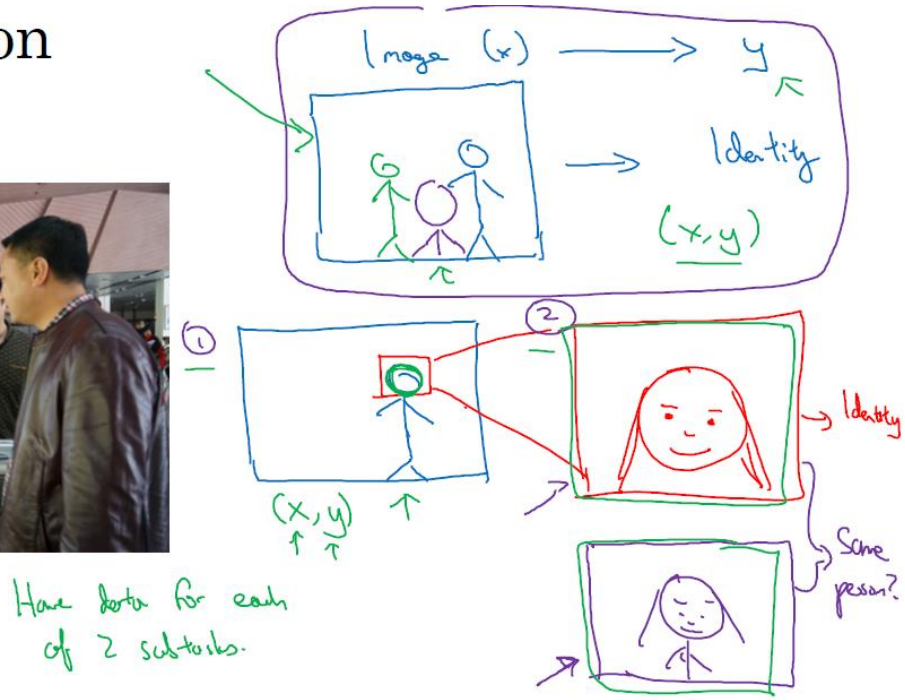

- Can train a big enough neural network to do well on all the tasks.

**What is end-to-end deep learning?**

Breaking down the problem into two simplified problems which ends into better overall performance.

# Face recognition


[Image courtesy of Baidu]

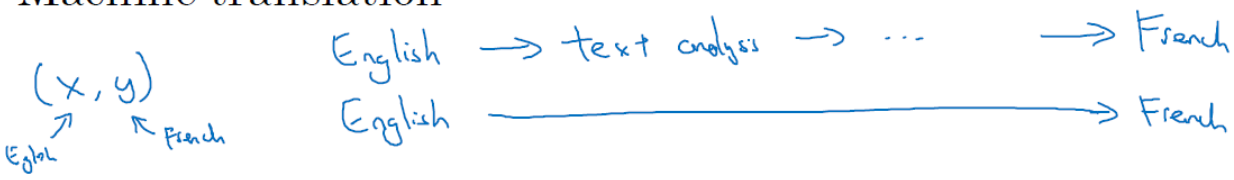



We have a lot of data for each task here (I) recognizing face from an image (II) identifying person, but we have not much data for the mixed version.

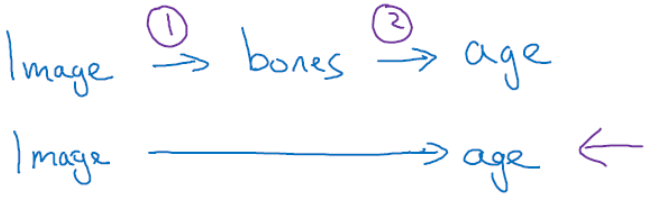
For machine translation but as we have a lot of data for translating a language to other one, so we do not need to simplify the model and we can have an end-to-end translation. It is not works well for estimating child's age by the x-ray.

# More examples

## Machine translation




## Estimating child's age:




Andrew Ng

**Whether to use end-to-end deep learning**

First bullet: we do not want to think about phonemes when instead we can pronounce the whole word in once. While if we have less data it might be more useful to let the model learn on its own so it might find better representative of data.

The last bullet is about injecting manual information into network and can be so much helpful in many places.

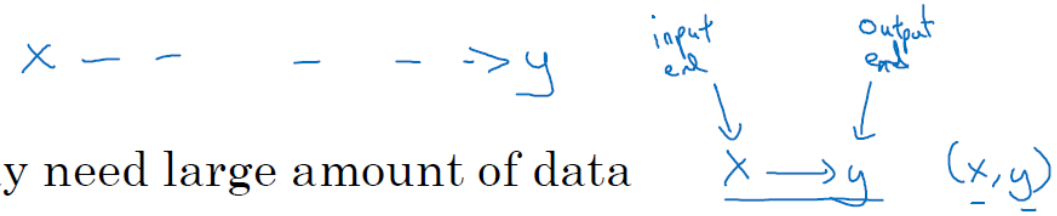
# Pros and cons of end-to-end deep learning

Pros:

- Let the data speak


- Less hand-designing of components needed

Cons:




- May need large amount of data
- Excludes potentially useful hand-designed components