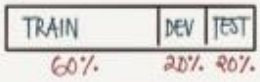
Practical aspects of Deep Learning

- Train / Dev / Test sets
- Bias / Variance
- Basic Recipe for Machine Learning
- Regularization
- Why regularization reduces overfitting?
- Dropout Regularization
- Understanding Dropout
- Other regularization methods
- Normalizing inputs
- Vanishing / Exploding gradients
- Weight Initialization for Deep Networks
- Numerical approximation of gradients
- Gradient checking implementation notes
- Initialization summary
- Regularization summary

SETTING UP YOUR ML APP

CLASSIC ML

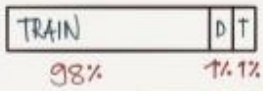
100 - 10000 SAMPLES



ALL FROM SAME PLACE → DISTRIBUTION

DEEP LEARNING

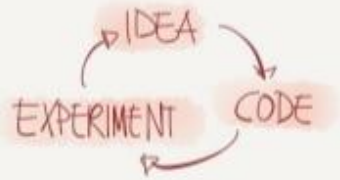
1M SAMPLES



EX: TRAIN  PRO CAT PICS FROM INTERNET

DEV/TEST  BLURRY CAT PICS FROM APP

TIP
DEV & TEST SHOULD COME FROM SAME DISTRIBUTION

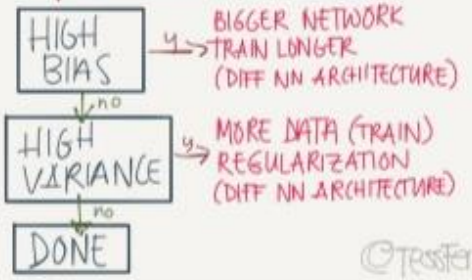


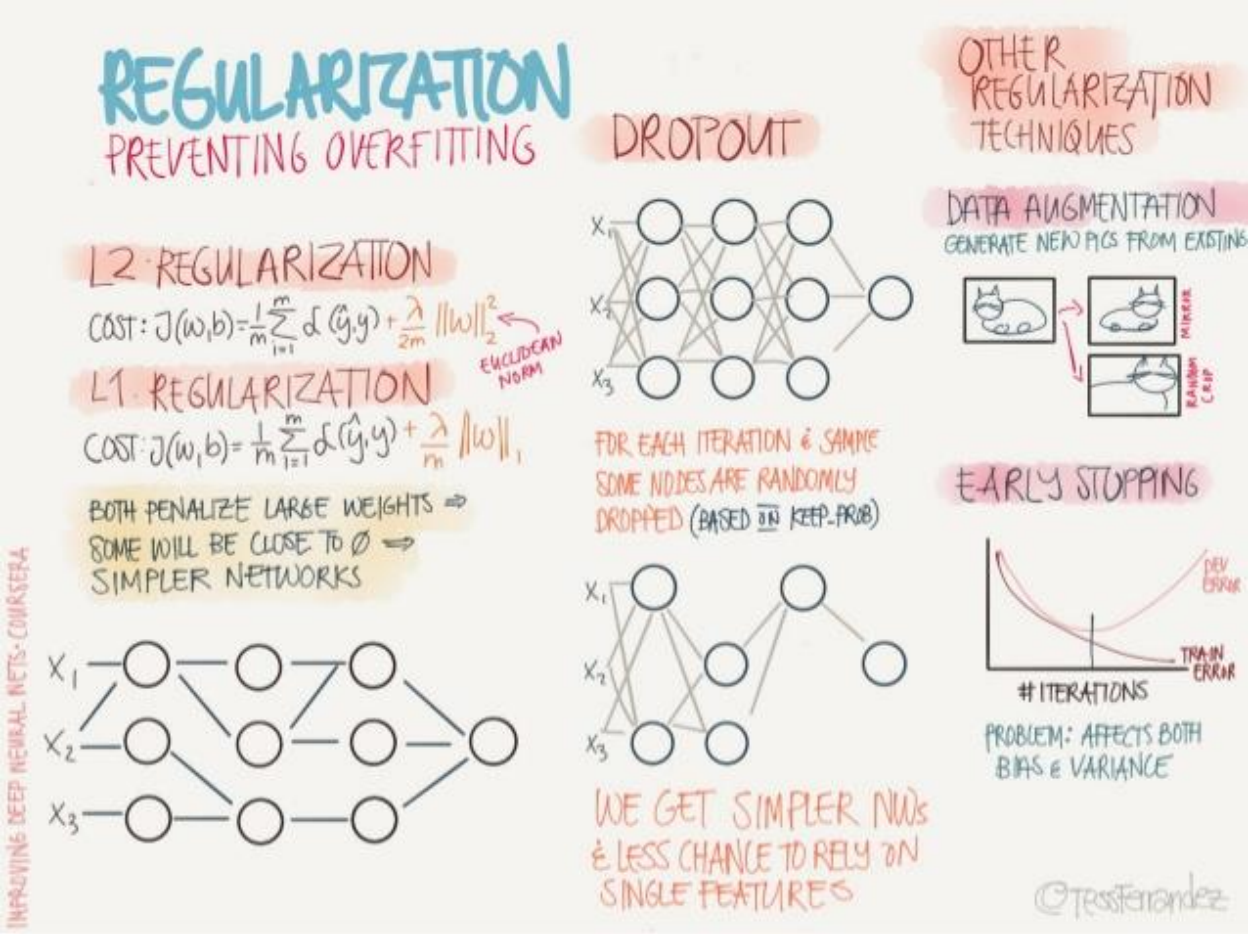
BIAS/VARIANCE

	ERROR			
TRAIN	1%	15%	15%	0.5%
TEST	11%	16%	30%	1%
	HIGH VARIANCE	HIGH BIAS	HIGH BIAS & VARIANCE	LOW BIAS & VARIANCE

ASSUMING HUMANS GET 0% ERROR

THE ML RECIPE





Practical aspects of Deep Learning

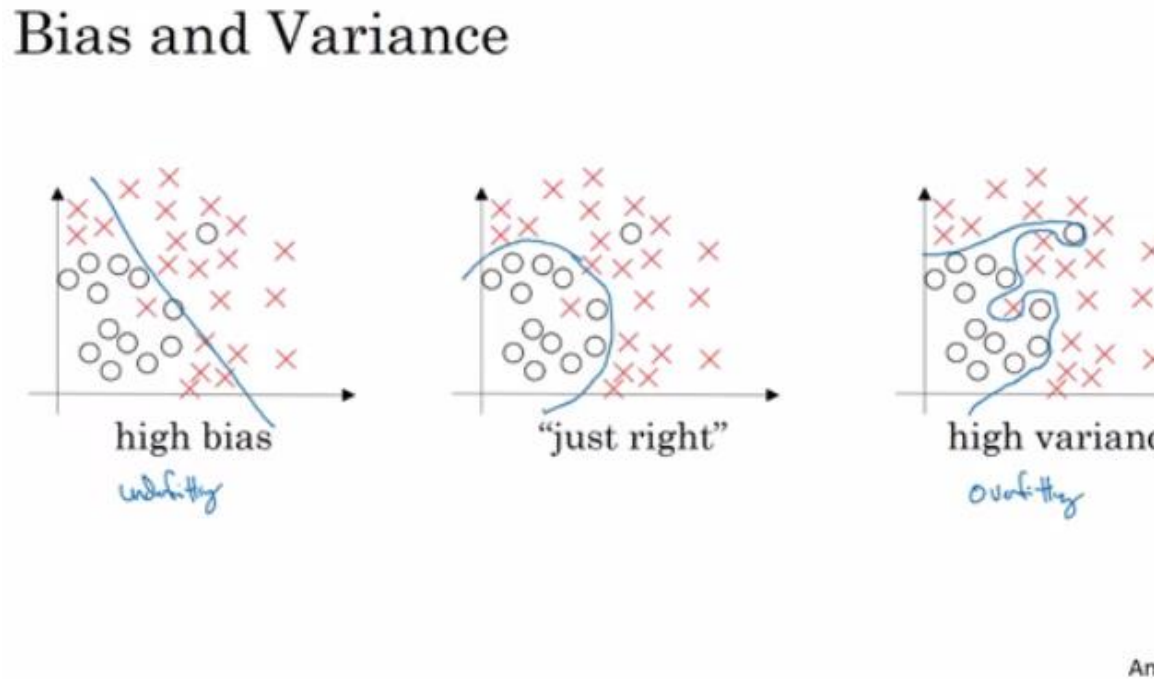
Train / Dev / Test sets

- Its impossible to get all your hyperparameters right on a new application from the first time.
- So the idea is you go through the loop: `Idea ==> Code ==> Experiment`.
- You have to go through the loop many times to figure out your hyperparameters.
- Your data will be split into three parts:
    - Training set. (Has to be the largest set)
    - Hold-out cross validation set / Development or "dev" set.
    - Testing set.
- You will try to build a model upon training set then try to optimize hyperparameters on dev set as much as possible. Then after your model is ready you try and evaluate the testing set.
- so the trend on the ratio of splitting the models:
    - If size of the dataset is 100 to 1000000 ==> 60/20/20
    - If size of the dataset is 1000000 to INF ==> 98/1/1 or 99.5/0.25/0.25

- The trend now gives the training data the biggest sets.
- Make sure the dev and test set are coming from the same distribution.
    - For example if cat training pictures is from the web and the dev/test pictures are from users cell phone they will mismatch. It is better to make sure that dev and test set are from the same distribution.
- The dev set rule is to try them on some of the good models you've created.
- Its OK to only have a dev set without a testing set. But a lot of people in this case call the dev set as the test set. A better terminology is to call it a dev set as its used in the development.

Bias / Variance

- Bias / Variance techniques are Easy to learn, but difficult to master.
- So here the explanation of Bias / Variance:
    - If your model is underfitting (logistic regression of non linear data) it has a "high bias"
    - If your model is overfitting then it has a "high variance"
    - Your model will be alright if you balance the Bias / Variance
    - For more:



- Another idea to get the bias / variance if you don't have a 2D plotting mechanism:
    - High variance (overfitting) for example:
        - Training error: 1%
        - Dev error: 11%
    - high Bias (underfitting) for example:
        - Training error: 15%
        - Dev error: 14%
    - high Bias (underfitting) && High variance (overfitting) for example:
        - Training error: 15%

        - Test error: 30%
    - Best:
        - Training error: 0.5%
        - Test error: 1%
    - These Assumptions came from that human has 0% error. If the problem isn't like that you'll need to use human error as baseline.

Basic Recipe for Machine Learning

- If your algorithm has a high bias:
    - Try to make your NN bigger (size of hidden units, number of layers)
    - Try a different model that is suitable for your data.
    - Try to run it longer.
    - Different (advanced) optimization algorithms.
- If your algorithm has a high variance:
    - More data.
    - Try regularization.
    - Try a different model that is suitable for your data.
- You should try the previous two points until you have a low bias and low variance.
- In the older days before deep learning, there was a "Bias/variance tradeoff". But because now you have more options/tools for solving the bias and variance problem its really helpful to use deep learning.
- Training a bigger neural network never hurts.

Regularization

- Adding regularization to NN will help it reduce variance (overfitting)
- L1 matrix norm:
    - `||W|| = Sum(|w[i,j]|) # sum of absolute values of all w`
- L2 matrix norm because of arcane technical math reasons is called Frobenius norm:
    - `||W||^2 = Sum(|w[i,j]|^2) # sum of all w squared`
    - Also can be calculated as `||W||^2 = W.T * W`
- Regularization for logistic regression:
    - The normal cost function that we want to minimize is: `J(w,b) = (1/m) * Sum(L(y(i),y'(i)))`
    - The L2 regularization version: `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum(|w[i]|^2)`
    - The L1 regularization version: `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum(|w[i]|)`
    - The L1 regularization version makes a lot of w values become zeros, which makes the model size smaller.
    - L2 regularization is being used much more often.
    - `lambda` here is the regularization parameter (hyperparameter)
- Regularization for NN:
    - The normal cost function that we want to minimize is: `J(W1,b1...,WL,bL) = (1/m) * Sum(L(y(i),y'(i)))`

    - The L2 regularization version:
      `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum((||W[l]||^2)`
    - We stack the matrix as one vector `(mn,1)` and then we apply `sqrt(w1^2 + w2^2.....)`
    - To do back propagation (old way):
      `dw[l] = (from back propagation)`
    - The new way:
      `dw[l] = (from back propagation) + lambda/m * w[l]`
    - So plugging it in weight update step:
        - `w[l] = w[l] - learning_rate * dw[l]`
        - `= w[l] - learning_rate * ((from back propagation) + lambda/m * w[l])`
        - `= w[l] - (learning_rate*lambda/m) * w[l] - learning_rate * (from back propagation)`
        - `= (1 - (learning_rate*lambda)/m) * w[l] - learning_rate * (from back propagation)`
        -
    - In practice this penalizes large weights and effectively limits the freedom in your model.
    - The new term `(1 - (learning_rate*lambda)/m) * w[l]` causes the **weight to decay** in proportion to its size.

Why regularization reduces overfitting?

Here are some intuitions:

- Intuition 1:
    - If `lambda` is too large - a lot of w's will be close to zeros which will make the NN simpler (you can think of it as it would behave closer to logistic regression).
    - If `lambda` is good enough it will just reduce some weights that makes the neural network overfit.
- Intuition 2 (with *tanh* activation function):
    - If `lambda` is too large, w's will be small (close to zero) - will use the linear part of the *tanh* activation function, so we will go from non linear activation to *roughly* linear which would make the NN a *roughly* linear classifier.
    - If `lambda` good enough it will just make some of *tanh* activations *roughly* linear which will prevent overfitting.

Implementation tip: if you implement gradient descent, one of the steps to debug gradient descent is to plot the cost function J as a function of the number of iterations of gradient descent and you want to see that the cost function J decreases **monotonically** after every elevation of gradient descent with regularization. If you plot the old definition of J (no regularization) then you might not see it decrease monotonically.

Dropout Regularization

- In most cases Andrew Ng tells that he uses the L2 regularization.

- The dropout regularization eliminates some neurons/weights on each iteration based on a probability.
- A most common technique to implement dropout is called "Inverted dropout".
- Code for Inverted dropout:

```
keep_prob = 0.8   # 0 <= keep_prob <= 1
l = 3  # this code is only for layer 3
# the generated number that are less than 0.8 will be dropped. 80% stay, 20% dropped
d3 = np.random.rand(a[l].shape[0], a[l].shape[1]) < keep_prob

a3 = np.multiply(a3,d3)   # keep only the values in d3

# increase a3 to not reduce the expected value of output
# (ensures that the expected value of a3 remains the same) - to solve the scaling problem
a3 = a3 / keep_prob
```

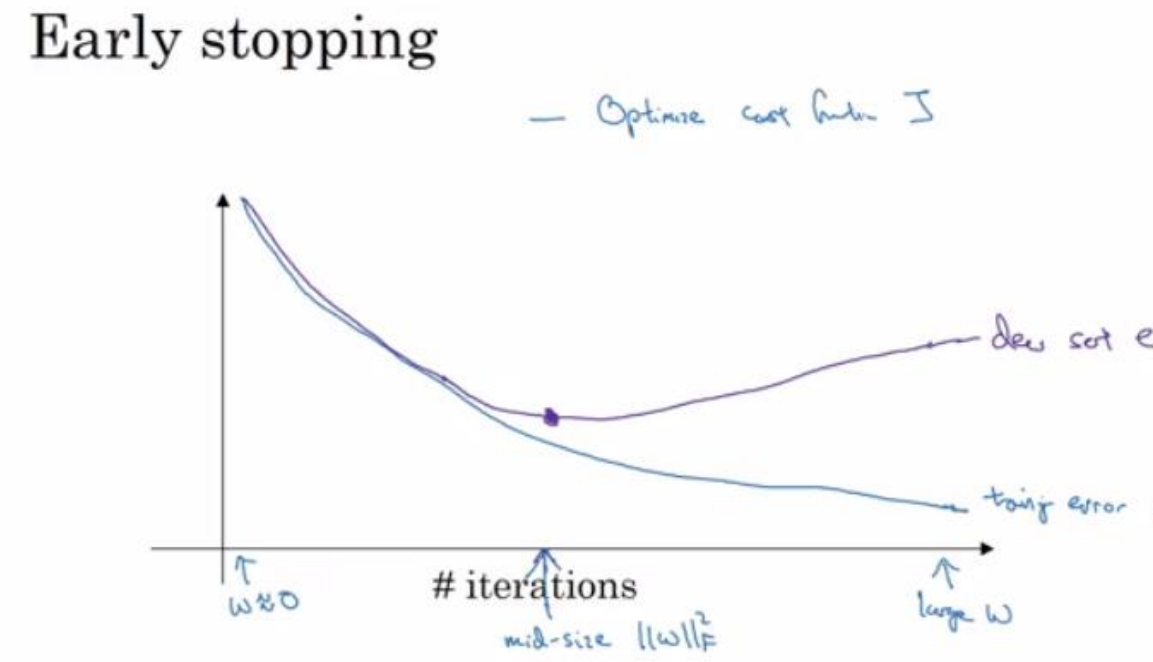
- Vector d[l] is used for forward and back propagation and is the same for them, but it is different for each iteration (pass) or training example.
- At test time we don't use dropout. If you implement dropout at test time - it would add noise to predictions.

Understanding Dropout

- In the previous video, the intuition was that dropout randomly knocks out units in your network. So it's as if on every iteration you're working with a smaller NN, and so using a smaller NN seems like it should have a regularizing effect.
- Another intuition: can't rely on any one feature, so have to spread out weights.
- It's possible to show that dropout has a similar effect to L2 regularization.
- Dropout can have different `keep_prob` per layer.
- The input layer dropout has to be near 1 (or 1 - no dropout) because you don't want to eliminate a lot of features.
- If you're more worried about some layers overfitting than others, you can set a lower `keep_prob` for some layers than others. The downside is, this gives you even more hyperparameters to search for using cross-validation. One other alternative might be to have some layers where you apply dropout and some layers where you don't apply dropout and then just have one hyperparameter, which is a `keep_prob` for the layers for which you do apply dropouts.
- A lot of researchers are using dropout with Computer Vision (CV) because they have a very big input size and almost never have enough data, so overfitting is the usual problem. And dropout is a regularization technique to prevent overfitting.
- A downside of dropout is that the cost function J is not well defined and it will be hard to debug (plot J by iteration).
    - To solve that you'll need to turn off dropout, set all the `keep_prob`s to 1, and then run the code and check that it monotonically decreases J and then turn on the dropouts again.

Other regularization methods

- **Data augmentation**:
  - For example in a computer vision data:
    - You can flip all your pictures horizontally this will give you m more data instances.
    - You could also apply a random position and rotation to an image to get more data.
  - For example in OCR, you can impose random rotations and distortions to digits/letters.
  - New data obtained using this technique isn't as good as the real independent data, but still can be used as a regularization technique.
- **Early stopping**:
  - In this technique we plot the training set and the dev set cost together for each iteration. At some iteration the dev set cost will stop decreasing and will start increasing.
  - We will pick the point at which the training set error and dev set error are best (lowest training cost with lowest dev cost).
  - We will take these parameters as the best parameters.



  - Andrew prefers to use L2 regularization instead of early stopping because this technique simultaneously tries to minimize the cost function and not to overfit which contradicts the orthogonalization approach (will be discussed further).
  - But its advantage is that you don't need to search a hyperparameter like in other regularization approaches (like `lambda` in L2 regularization).
- **Model Ensembles**:
  - Algorithm:
    - Train multiple independent models.

        - At test time average their results.
    - It can get you extra 2% performance.
    - It reduces the generalization error.
    - You can use some snapshots of your NN at the training ensembles them and take the results.

Normalizing inputs

- If you normalize your inputs this will speed up the training process a lot.
- Normalization are going on these steps:
    1. Get the mean of the training set: `mean = (1/m) * sum(x(i))`
    2. Subtract the mean from each input: `X = X - mean`
        - This makes your inputs centered around 0.
    3. Get the variance of the training set: `variance = (1/m) * sum(x(i)^2)`
    4. Normalize the variance. `X /= variance`
- These steps should be applied to training, dev, and testing sets (but using mean and variance of the train set).
- Why normalize?
    - If we don't normalize the inputs our cost function will be deep and its shape will be inconsistent (elongated) then optimizing it will take a long time.
    - But if we normalize it the opposite will occur. The shape of the cost function will be consistent (look more symmetric like circle in 2D example) and we can use a larger learning rate alpha - the optimization will be faster.

Vanishing / Exploding gradients

- The Vanishing / Exploding gradients occurs when your derivatives become very small or very big.
- To understand the problem, suppose that we have a deep neural network with number of layers L, and all the activation functions are **linear** and each `b = 0`
    - Then:
    - `Y' = W[L]W[L-1].....W[2]W[1]X`
    - Then, if we have 2 hidden units per layer and x1 = x2 = 1, we result in:
    - `if W[l] = [1.5  0]`
    - `[0  1.5] (l != L because of different dimensions in the output layer)`
    - `Y' = W[L] [1.5  0]^(L-1) X = 1.5^L  # which will be very large`
    - `[0  1.5]`
    - `if W[l] = [0.5  0]`
    - `[0  0.5]`
    - `Y' = W[L] [0.5  0]^(L-1) X = 0.5^L  # which will be very small`
    - `[0  0.5]`
- The last example explains that the activations (and similarly derivatives) will be decreased/increased exponentially as a function of number of layers.
- So If W > I (Identity matrix) the activation and gradients will explode.
- And If W < I (Identity matrix) the activation and gradients will vanish.

- Recently Microsoft trained 152 layers (ResNet)! which is a really big number. With such a deep neural network, if your activations or gradients increase or decrease exponentially as a function of L, then these values could get really big or really small. And this makes training difficult, especially if your gradients are exponentially smaller than L, then gradient descent will take tiny little steps. It will take a long time for gradient descent to learn anything.
- There is a partial solution that doesn't completely solve this problem but it helps a lot - careful choice of how you initialize the weights (next video).

Weight Initialization for Deep Networks

- A partial solution to the Vanishing / Exploding gradients in NN is better or more careful choice of the random initialization of weights
- In a single neuron (Perceptron model): `Z = w1x1 + w2x2 + ... + wnxn`
    - So if `n_x` is large we want `W`'s to be smaller to not explode the cost.
- So it turns out that we need the variance which equals `1/n_x` to be the range of `W`'s
- So lets say when we initialize `W`'s like this (better to use with `tanh` activation):
- `np.random.rand(shape) * np.sqrt(1/n[l-1])`

  or variation of this (Bengio et al.):

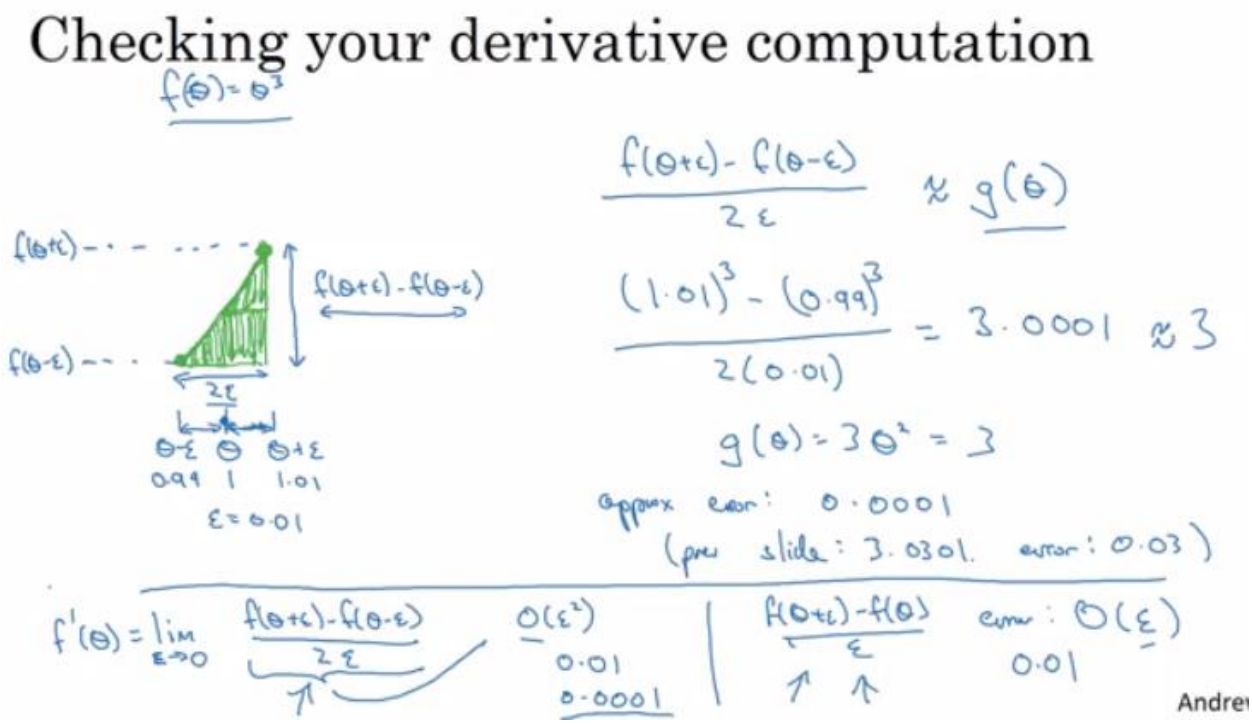
  ```
  np.random.rand(shape) * np.sqrt(2/(n[l-1] + n[l]))
  ```

- Setting initialization part inside sqrt to `2/n[l-1]` for `ReLU` is better:
- `np.random.rand(shape) * np.sqrt(2/n[l-1])`
- Number 1 or 2 in the nominator can also be a hyperparameter to tune (but not the first to start with)
- This is one of the best way of partially solution to Vanishing / Exploding gradients (ReLU + Weight Initialization with variance) which will help gradients not to vanish/explode too quickly
- The initialization in this video is called "He Initialization / Xavier Initialization" and has been published in 2015 paper.

Numerical approximation of gradients

- There is an technique called gradient checking which tells you if your implementation of backpropagation is correct.

- There's a numerical way to calculate the derivative:



- Gradient checking approximates the gradients and is very helpful for finding the errors in your backpropagation implementation but it's slower than gradient descent (so use only for debugging).
- Implementation of this is very simple.
- Gradient checking:
  - First take `W[1],b[1],...,W[L],b[L]` and reshape into one big vector (`theta`)
  - The cost function will be `J(theta)`
  - Then take `dW[1],db[1],...,dW[L],db[L]` into one big vector (`d_theta`)
  - **Algorithm**:
  - `eps = 10^-7  # small number`
  - `for i in len(theta):`
  - `  d_theta_approx[i] = (J(theta1,...,theta[i] + eps) - J(theta1,...,theta[i] - eps)) / 2*eps`
  - Finally we evaluate this formula `(||d_theta_approx - d_theta||) / (||d_theta_approx||+||d_theta||)` (`||` - Euclidean vector norm) and check (with eps = 10^-7):
    - if it is < 10^-7 - great, very likely the backpropagation implementation is correct
    - if around 10^-5 - can be OK, but need to inspect if there are no particularly big values in `d_theta_approx - d_theta` vector
    - if it is >= 10^-3 - bad, probably there is a bug in backpropagation implementation

Gradient checking implementation notes

- Don't use the gradient checking algorithm at training time because it's very slow.

- Use gradient checking only for debugging.
- If algorithm fails grad check, look at components to try to identify the bug.
- Don't forget to add `lamda/(2m) * sum(W[l])` to `J` if you are using L1 or L2 regularization.
- Gradient checking doesn't work with dropout because J is not consistent.
    - You can first turn off dropout (set `keep_prob = 1.0`), run gradient checking and then turn on dropout again.
- Run gradient checking at random initialization and train the network for a while maybe there's a bug which can be seen when w's and b's become larger (further from 0) and can't be seen on the first iteration (when w's and b's are very small).

Initialization summary

- The weights $W^{[l]}$ should be initialized randomly to break symmetry
- It is however okay to initialize the biases $b^{[l]}$ to zeros. Symmetry is still broken so long as $W^{[l]}$ is initialized randomly
- Different initializations lead to different results
- Random initialization is used to break symmetry and make sure different hidden units can learn different things
- Don't intialize to values that are too large
- He initialization works well for networks with ReLU activations.

Regularization summary

1. L2 Regularization

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.

What you should remember:
Implications of L2-regularization on:

- cost computation:
    - A regularization term is added to the cost
- backpropagation function:

    - There are extra terms in the gradients with respect to weight matrices
- weights:
    - weights end up smaller ("weight decay") - are pushed to smaller values.

2. Dropout

What you should remember about dropout:

- 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.

Week 1 Quiz - Practical aspects of deep learning

1. If you have 10,000,000 examples, how would you split the train/dev/test set?
    - 98% train . 1% dev . 1% test
2. The dev and test set should:
    - Come from the same distribution
3. If your Neural Network model seems to have high variance, what of the following would be promising things to try?
    - Add regularization
    - Get more training data

    Note: Check here.

4. You are working on an automated check-out kiosk for a supermarket, and are building a classifier for apples, bananas and oranges. Suppose your classifier obtains a training set error of 0.5%, and a dev set error of 7%. Which of the following are promising things to try to improve your classifier? (Check all that apply.)
    - Increase the regularization parameter lambda
    - Get more training data

    Note: Check here.

5. What is weight decay?
    - A regularization technique (such as L2 regularization) that results in gradient descent shrinking the weights on every iteration.
6. What happens when you increase the regularization hyperparameter lambda?
    - Weights are pushed toward becoming smaller (closer to 0)

7. With the inverted dropout technique, at test time:
    - You do not apply dropout (do not randomly eliminate units) and do not keep the 1/keep_prob factor in the calculations used in training

8. Increasing the parameter keep_prob from (say) 0.5 to 0.6 will likely cause the following: (Check the two that apply)

- [ ] Increasing the regularization effect
- [ ] Reducing the regularization effect
- [ ] Causing the neural network to end up with a higher training set error
- [ ] Causing the neural network to end up with a lower training set error

9. Which of these techniques are useful for reducing variance (reducing overfitting)? (Check all that apply.)

- [ ] Xavier initialization
- [ ] Vanishing gradient
- [ ] Data augmentation
- [ ] Exploding gradient
- [ ] Gradient Checking
- [ ] L2 regularization
- [ ] Dropout

8. Increasing the parameter keep_prob from (say) 0.5 to 0.6 will likely cause the following: (Check the two that apply)
    - Reducing the regularization effect
    - Causing the neural network to end up with a lower training set error
9. Which of these techniques are useful for reducing variance (reducing overfitting)? (Check all that apply.)
    - Dropout
    - L2 regularization
    - Data augmentation
10. Why do we normalize the inputs x?
    - It makes the cost function faster to optimize