

## Week 1: Practical aspects of deep learning

### Setting up your machine learning application

#### Train / Dev / Test Sets

as stated before, defining ml is a **highly iterative** process. we must make decisions on things like the # layers, # of midunits, learning rates, and which activation functions to utilize, among others.

↳ experiment → refine → iterate

• specific working in applications such as vision, speech, NLP, structured data (bank), often try to switch into another domain of the aforementioned. typically, intuitions learned in the **test domain**, do not carry over into the new applications (even w/ a specific combination of hyperparameter tuning).

↳ therefore, experimentation in these is key to optimize the models. (importance of iterativity)



• the work flow is to train multiple models on the **training set**, test with the **dev set** and after finding the best model we evaluate it with the **test set**.

- ↳ the final step is done to test that the model is producing an unbiased estimate.
- previously, the best practice was
  - ↳ 70/30%, train/test split
  - ↳ 60/20/20, train/dev/test split

\* these ratios are reasonable if we have 100 to 10,000 examples in our datasets.
- In the modern big data era where we might have a million examples in a dataset, the trend is that the dev/test sets are becoming a much smaller % of the total.
  - ↳ the dev set is only used to determine which of the models is doing better, therefore, we don't need 20% of the dataset for comparison in huge datasets. 10,000 examples might be enough for test and dev sets in cash.
  - ↳ 98/1/1 for 1 million examples.
- mismatched train/test distributions
  - ↳ example:
  - ↳ train set: high quality pictures of cats from websites online
  - ↳ dev/test set: Cat pictures from users using an app (maybe they're blurrier, less resolution)

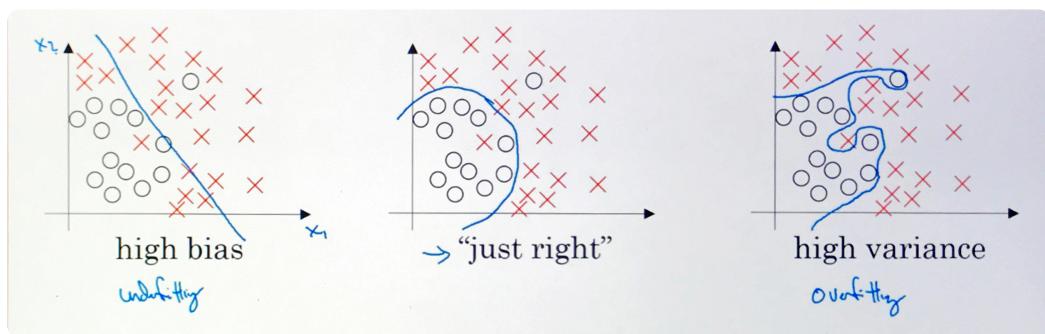
\* we have to make sure that the composition

of our sets are based on the same distribution  
to accurately validate the performance of the  
model in its real-world application.  
↳ this involves testing models.

- Sometimes, it might be okay to not have a test set.
  - ↳ test you won't have an unbiased estimate of performance.
  - ↳ sometimes you don't need one...
  - ↳ we might be overfitting to the test set if we are not careful as we are selecting our hyperparameters based on its results. (causing a biased model)

### Bias / variance

- In the deep learning era, there's less of a bias vs. variance trade off.



- In the high bias example, we are underfitting at the expense of having a lot of errors in our prediction cause we are trying to generalize too broadly.

— In the high variance situation, we are fitting too closely to the training data. This makes our model perform worse in dev/test/affiliation because our model is very constrained and does not generalize well.

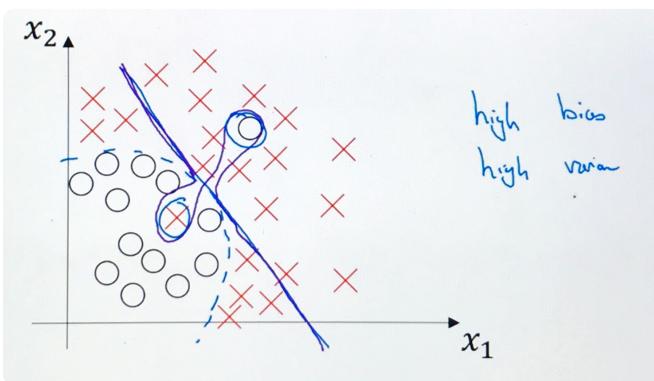
- In the previous example, we could visualize the boundary as we were observing in a 2D plane. In higher dimensions, different metrics are required to perceive the bias/variance in our model.

Human reference metric: ~ 0%

train set error	1%	15%	15%	0.5%
dev set error	11%	16%	30%	1%
high variance	high bias Underfitting	high bias Overfitting	low bias and low variance	

\* this analysis is based on the fact that the optimal (bayes) error is near 0%.

↳ this analysis is also assuming that both sets are taken from the same distribution.



- the high bias comes from the linear parts not incorporating the true quadratic behaviour of the data (underfitting the true function).

- the high variance comes from the high flexibility to fit these two outlier examples in the middle ( $\Rightarrow$  shown by the purple line)
- \*these effects are more common/possible in high dimensionalities.

### Basic Heuristics for machine learning

- systematic reduction when we have a high bias/variance.
- high bias (based on training set performance)
  - i.  $\hookrightarrow$  try a bigger network with more layers or hidden units
  - ii.  $\hookrightarrow$  train the model longer or use more advanced optimization algorithms.
  - iii.  $\hookrightarrow$  find a neural network architecture which is better suited for the problem. (this might work, but this is not guaranteed)
- \* with a big enough network, fitting the training set  $\Rightarrow$  generally possible if it's a problem that's generally solvable

- ↳ If an image is very blurry it might be impossible to fit it.
- ↳ If layer error is not too high on the problem.
- Once we have fit the data to the training set, we might ask if we have a high variance problem (by using the dev set).
  - ↳ To solve this,
    - i. We might get more data (if possible)
    - ii. Try regularization to reduce overfitting.
    - iii. Try a more appropriate neural network architecture
  - If we have a high bias problem, getting more training data will generally not help.
    - ↳ Try the appropriate techniques mentioned above based on what problem you actually have.
  - Back in the pre-deep learning era, we used to call it a bias/variance tradeoff because as we lowered one, the other one would vary inversely (increase).
    - ↳ Back then, we didn't have many tools that just affected one of the two without impacting the other.
      - ↳ Bigger networks and more data solved

these independent nodes to interact one without hurting the other.

↳ this is why neural nets really shine at supervised learning tasks.

\* if we have well regularized data, training bigger networks doesn't hurt performance.

↳ the only downside of the increase is a rise in computational time.

## regularization

• if we have a situation where overfitting is happening (high variance), regularization might be a way to solve it. Another way is to get more training data... which sometimes might not be possible.

• Dealing with logistic regression,

$$\text{loss } J(w, b) = \frac{1}{m} \cdot \sum_{i=1}^m J(\hat{y}^{(i)}, y)$$

$$+ \frac{\lambda}{2m} \|w\|_2^2 \quad \text{regularization parameter}$$

$$\text{↳ where, the norm of } w \text{ squared } (\|w\|_2^2) = \sum_{j=1}^n w_j^2 = w^T w \quad \text{(a square euclidean norm)}$$

not the parameter vector

\* this is called L2 regularization because we are using the euclidean norm

\* this method of regularization is more popular than

↳ ..

- Why do we regularize the parameter  $w$ , but not  $b$  in this expression (such as  $\frac{\lambda}{2m} \cdot b^2$ )

↳ In practice this can be done but it's omitted because  $w$  encompasses almost all the parameters (in high dimension) while  $b$  is only one parameter. Therefore this one parameter won't make much of a difference in practicality.

- L<sub>1</sub> regularization:

↳ instead we add  $+\frac{\lambda}{2m} \sum_{i=1}^m |w_i| = \frac{\lambda}{2m} \cdot \|w\|_1$ ,

↳ if we use L<sub>1</sub> regularization, w will end up being sparse. This means that the w vector will have a lot of zeros in it.

lambda

- $\lambda$  = regularization parameter (hyperparameter)

↳ this is set using the development set

- Because the cost function is,

$$\hookrightarrow \frac{1}{m} \cdot \sum_{i=1}^m J(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \cdot \|w\|_2^2, \text{ there's a}$$

constant  $\lambda$  of var between optimizing the  $w$ 's and setting them small enough to not increase the cost function (use  $\lambda$  tuned to find suitable tradeoff)

Frobenius norm

- $L_2$  in a neural network

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

$$\hookrightarrow \|w^{(l)}\|^2 = \sum_{i=1}^m \sum_{j=1}^{c_{l-1}} (w_{i,j}^{(l)})^2$$

$i=1 \ j=1$

↳ Where, rows " $i$ " should be the # of neurons  
in the current layer ( $n^{(l)}$ )

↳ and columns " $j$ " of the weights of the  
matrix should be equal to the neurons  
in the previous layer ( $n^{(l-1)}$ )

- how do we implement gradient descent with this?

$$\hookrightarrow \delta w^{(l)} = \dots + \frac{1}{m} \cdot w^{(l)}$$

$$\hookrightarrow w^{(l)} = w^{(l)} - \alpha \cdot \delta w^{(l)}$$

\* L2 normalization is also called weight decay because  
for each update of  $w$  we are subtracting by a  
 $\alpha \cdot \lambda \cdot w^{(l)}$  term additional to what we used to get from  
Backprop.

### Why does regularization reduce overfitting

- Why is it that shrinking the Frobenius norm will  
cause less overfitting? (intuition)

↳ if we set  $\lambda$  large, we will make the magnitude of  
weights close to zero as we are trying to  
minimize the cost function ( $J$ )

↳ this will zero out a lot of the impact of  
many hidden units in a neural net  
meaning that the resulting neural network  
can be simplified into a smaller equivalent  
neural network stepped multiple layers deep.

↳ this will take up from the overfitting

case towards a more general  
(Gaussian) model (such as higher bias).

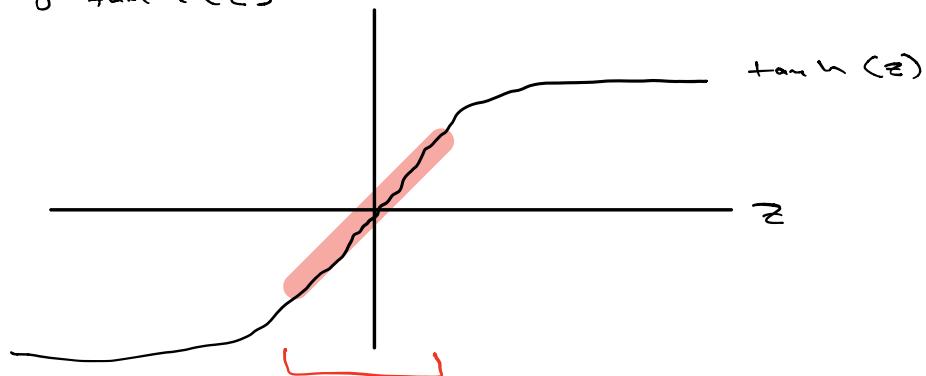
↳ the intermediate state between these two  
is what we desire.

↳ by removing most hidden unit we are minimizing  
the individual effects of each unit and  
giving more weighting to the model as a whole.

↳ making it less prone to overfitting.



Using  $\tanh(z)$ :



↳ so long as we keep the  $z$  value small, we are  
using the linear regime of the tanh function.

↳ if  $\lambda$  is large,  $w^{(l)}$  will be small, hence  
 $z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)}$  will also be small.

↳ hence every layer will be closely linear.

If every layer  $\Rightarrow$  linear (as seen previously),  
even a deep network will only be able to  
construct a linear function.

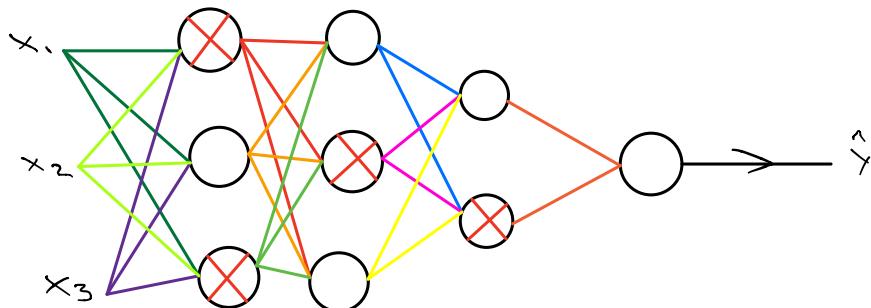
↳ we will not be able to fit those

Complicated decisions that lead to high variance / overfitting.

- \* To debug gradient descent, it is important to plot this new definition of the cost function ( $J(\cdot)$ ) with regularization vs the # of iterations. Not the previous definition.
  - ↳ to see it decrease across iterations.

### Dropout regularization

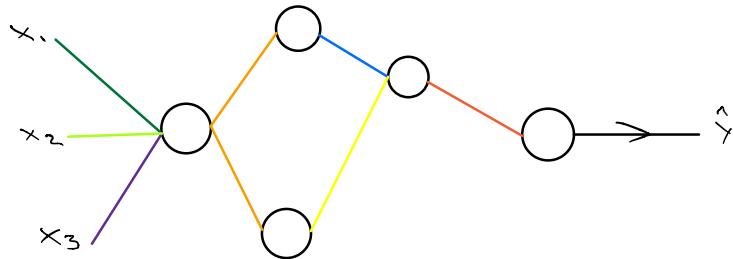
- with dropout we set a probability of eliminating a node in the neural network at each iteration of training in order to reduce overfitting.



- imagine that for each layer, there's a 0.5 chance of keeping a node. The resulting smaller/diminished network trained in the specific iteration of each iteration looks like the resulting network below.

↳ on a different iteration, the nodes that get eliminated might be different as there's a

there's a random probability of each specific one disappearing (0.5) in any given iteration.



- On each set of examples we would train the net under a different set of reduced nodes.

Implementing dropout ("Inverted dropout") ( $l=3$ , stochastic layer)

- Dropout vector for layer 3,

$$\leftarrow d_3 = \text{np.random.rand}(a_3.\text{shape}[0], a_3.\text{shape}[1]) < \text{keep\_prob}$$

\* keep\_prob is a number (such as 0.8) and it is the probability that a given hidden unit will be kept.

$\leftarrow$  0.2 chance of eliminating any one hidden unit in this example (for layer  $l=3$ )

\*  $d_3$  will be a matrix where there's a 90% chance that a node will be one and a 10% chance the node will be zero for a given iteration.

$$\leftarrow a_3 = \text{np.multiply}(a_3, d_3) * \text{element-wise multiplication}$$

\* the corresponding elements get erased out.

$$\leftarrow a_3 /= \text{keep\_prob}$$

\* we scale up our activations by the given

## Kerf\_Prob constant.

- take  $l=3$  to have 50 units. Given the probability (0.8), 10 units will shut off. When computing  $\bar{z}^{[4]}$  for the next layer, ( $\bar{z}^{[4]} = \bar{w}^{[4]}\cdot \bar{a}^{[3]} + b^{[4]}$ ), we must scale  $\bar{a}^{[3]}$  by the Kerf\_Prob in order to not reduce the original expected value of  $\bar{z}^{[4]}$ 
  - ↳ this line is known as the inverted dropout technique
- \* no matter what we set Kerf\_Prob to, inverted dropout ensures that the expected value of the next layer remains the same (because of the scaling)
  - ↳ this makes testing easier as we cancel out the scaling problem.
- this technique is the most common implementation of dropout.
- using  $\bar{a}_j$  means that for different training examples we zero out different units. Given the random nature, we also zero out different units on different passes (iterations) of the training set.
  - ↳ the vector zeroes out better in forward pass and back pass.
- at test time we do not implement dropout (we go back to business as usual). Because of the scaling this works and the expected values of the activations across layers don't change.

## Understanding Dropout

- using dropout randomly throws out units in a neural network making it as if we are working with a smaller neural network (which should have a regularizing effect in theory)
- Single unit perspective: a next stage unit doesn't rely too heavily on any one feature as these can go away at random throughout the training stage. This has the effect of spreading out the weights among the inputs.
  - ↳ thinks the square norm of the weights of the individuals are minimized (reducing overfitting)
  - ↳ drop out has a similar effect to L2 regularization

Reminder: in general, the # of neurons of the previous layer = # columns and the # of neurons of the current layer = # of rows in the layer's weight matrix (present layer, previous layer)

- dropout can be done on the input layer, but this is often not done in practice.
- if we're worried about some layers overfitting more than others, we can set them to a lower keep\_prob and let drop out do its thing
  - ↳ this tactic gives you more hyperparameters to tune when doing cross validation.

- Implementation tips:

- ↳ many of the first successful implementations of dropout happened in computer vision

- ↳ the input sizes are large, but we never work enough training data

- don't use dropout if there's not much overfitting

- \* major downside to dropout is that the cost function becomes not well defined

- ↳ on every iteration we are randomly killing off different nodes. this makes it harder to check the actual performance of  $\mathcal{D}$  in gradient descent.

### Other regularization techniques

- by flipping images horizontally (mirroring), we can double the size of training set

- ↳ not as good as collecting n new independent examples, but can be done without incurring any additional cost.

- ↳ this works also by taking random crops of an image.

- \* all of these lead to different examples / training of the same object / subject (known as **data augmentation**)

- early stopping:

- ↳ plot training error vs # of iterations and dev set error vs # of iterations

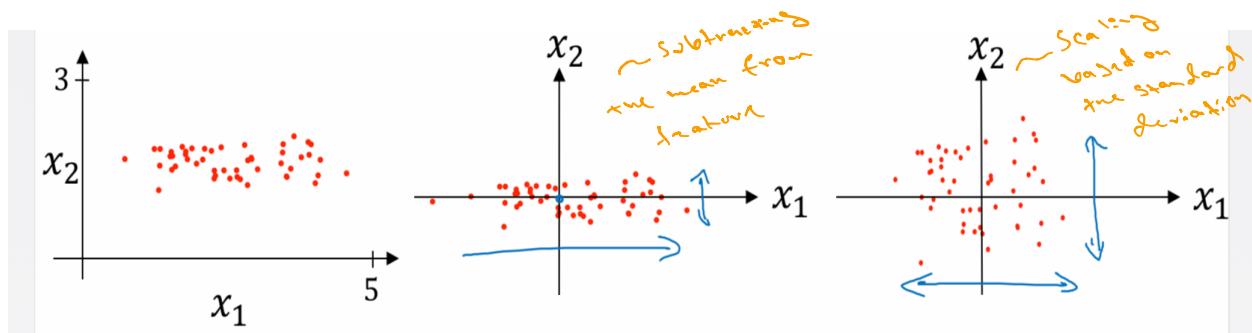
- ↳ When the two diverge (because the model is overfitting on the train set) we can stop gradient descent early to save the current weights and biases.
- ↳ we get a mid-size  $w$ , instead of a large  $w$  at later iterations.
- Orthogonalization is the idea that you want to focus on one task at a time
  - ↳ specifying an optimum cost function and descent strategy vs. not overfitting the model to the train data
    - ↳ two separate focused tasks. Through early stopping, we solve these two tasks and limit their influence.
    - \* this is what makes L2 regularization preferable over early stopping. Not solving these two sub problems together.
    - ↳ however, we must search over many values of **lambda** (computationally expensive)

### Normalizing Inputs

- Normalization: 
$$\frac{x - \mu}{\sigma} \sim \text{mean}$$
  $\sim \text{standard deviation}$
- normalization speeds up training of the neural network.

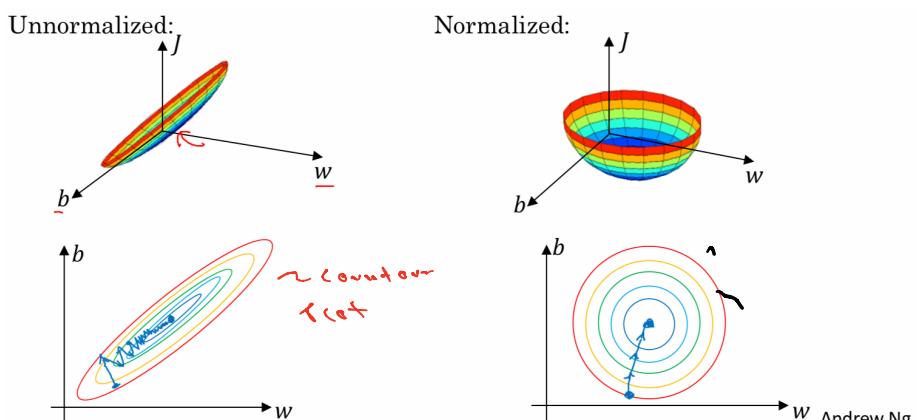
- if we normalize the data, we are basically doing a transformation where we are recentering it with the axis and resizing the spread such that the shape of the data is defined on a Standardized range.

- $\text{mean}(N) = \frac{\text{Sum}}{n}$ ,  $\text{std} = \sqrt{\frac{\sum (x^i - \mu)^2}{n-1}}$



- we want both the train and test data to go through the same transformation ( $\mu$  and  $\sigma$ ) which is calculated on the train set.

- why normalize the inputs?



- ↳ unnormalized features result in very elongated cost functions as the weight parameters ( $w_1, \dots, w_n$ ) might be in totally different ranges.
  - ↳ this might require a smaller training rate during gradient descent as we might need a lot of small steps to find the optimum high dimensional point
- ↳ By normalizing the features, the cost function will look more symmetric
  - ↳ in spherical contours we can move easily move towards the minimum regardless of where we start our parameters
- \* these intuitions scale to high dimensionality sets.
- we never want our ranges on totally different ranges such as  $x_1 = 1 \dots 1000$  and  $x_2 = 0 \dots 1$ 
  - ↳ performing normalization never does any harm

### Vanishing / Exploding Gradients

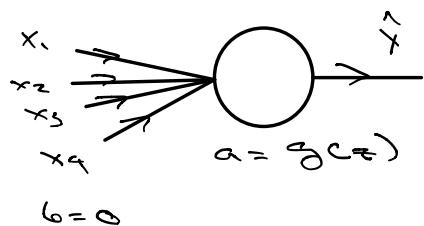
- when training a very deep network, steps can get very large or exponentially small.
  - ↳ careful choices of random weight initialization can be used to reduce this problem.
- if the weights are all a bit bigger than one at initialization, with a very deep network they can explode. If  $W^T > 1$  (such as 0.9), the value of  $W$  will decrease exponentially with depth.

↳ This argument can be extended to the gradients we compute through gradient descent for the weights in respect to the cost function.

↳ If the gradient explodes/vanishes, gradient descent will take tiny steps when learning and the time required for learning will be extended.

### Weight initialization for deep neural networks

- Single neuron case



$$z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

the larger  $n$  is, the smaller we want  $w_i$  to be  
so that if we are adding all the terms,  $z$  doesn't blow up.

↳ This can be achieved by setting the variance  $(\text{Var}(w_i)) = \frac{1}{n}$  (such that it increases as the # of inputs to the neuron increases)

↳  $w^{[l]} = \text{np.random.randn}(\text{shape of matrix})$   
 $\ast \sqrt{\frac{1}{n^{[l-1]}}}$  ( $n^{[l-1]}$ , # of units that are feeding into each of the neurons in layer  $l$ )

↳ for relu,  $\text{var}(w_i) = \frac{2}{n}$  works a bit

better

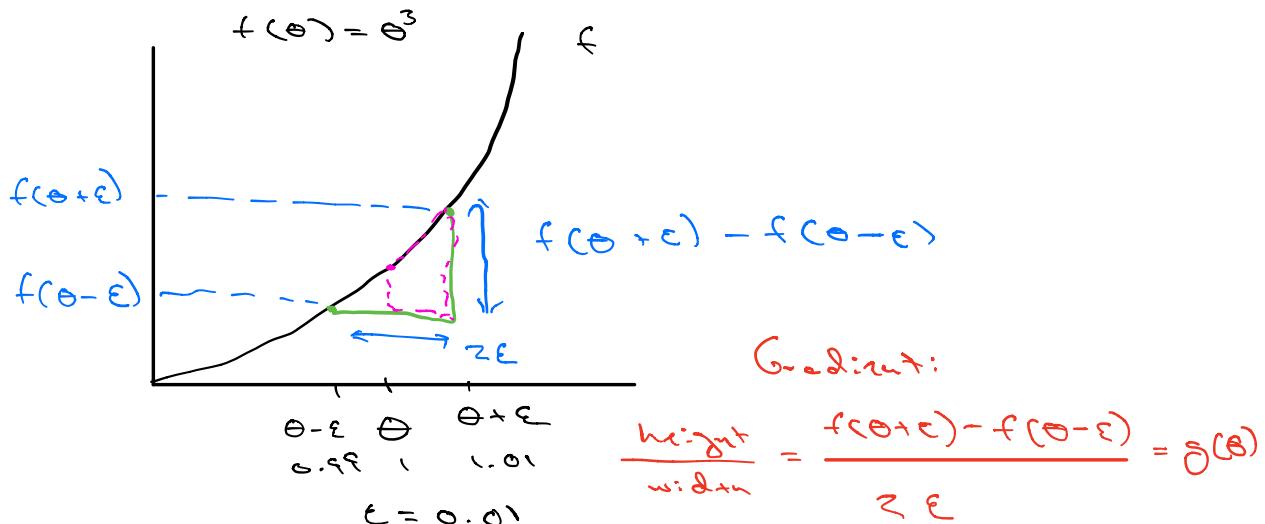
- This method doesn't solve exploding/vanishing gradients problem, but it helps reduce it.
  - ↳ it sets each row of the weight matrices  $W^{(l)}$  so that it's not too far off 1.

- Other variants:

↳ tanh :  $\sqrt{\frac{1}{n_{\text{in}} \cdot n_{\text{out}}}}$  (Xavier initialization)

or  $\sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}$

### Numerical approximation of Gradients



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

approximation error: 0.001

(By taking a one-sided difference, we get

3.0301 which is a larger approximation error)

- a two-sided difference is practice runs, two times as slow as a one-sided difference (but it's worth it because it's much more accurate)
  - ↳ for a non-zero value of  $\epsilon$ , the error is order of  $O(\epsilon^2)$

Gradient checking (used to check that backprop implementation is correct)

- take  $w^{(1)}, b^{(1)}, \dots, w^{(L)}, b^{(L)}$  and reshape them into a big vector  $\Theta$  (by concatenating them)
  - ↳  $J(w^{(1)}, b^{(1)}, \dots, w^{(L)}, b^{(L)}) = J(\Theta)$  cost function
- next, take the derivatives of the weights and biases and reshape them into a large vector  $\Delta\Theta$  through concatenation.
  - ↳ Is  $\Delta\Theta$  the gradient of  $J$ ?

Grad check algorithm:

for each  $i$ :

$$\text{derivative}[i] = \frac{J(\Theta_1, \Theta_2, \dots, \Theta_i + \epsilon, \dots) - J(\Theta_1, \Theta_2, \dots, \Theta_i - \epsilon, \dots)}{2\epsilon}$$

$$\approx \Delta\Theta[i] = \frac{\partial J}{\partial \Theta_i}$$

After doing this for all  $\Theta$ , we will have  $\Delta\Theta$  and  $\Delta\Theta_{\text{approx}}$

We can check if these two are equal to each other by taking the euclidean distance:

$$\|d\theta_{approximation} - d\theta\|_2$$

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

\* Setting  $\epsilon = 10^{-7}$ , we would expect that the difference between  $d\theta_{approx}$  and  $d\theta$  would be in the order of  $10^{-7}$ .

If some of the components of the difference are too large, we might have a bug somewhere.  
(such as  $10^{-3}$  error)

### Notes on Grad check

- don't use it in training, only to debug
- if an algorithm fails grad check, look at the components to try to identify the bug
- remember regularization term (if using it) when doing grad check.
- grad check does not work with dropout
  - the cost function is very difficult to compute in this case.
- Implement grad check without drop-out and then enable drop out when we make sure the gradients are being calculated correctly.