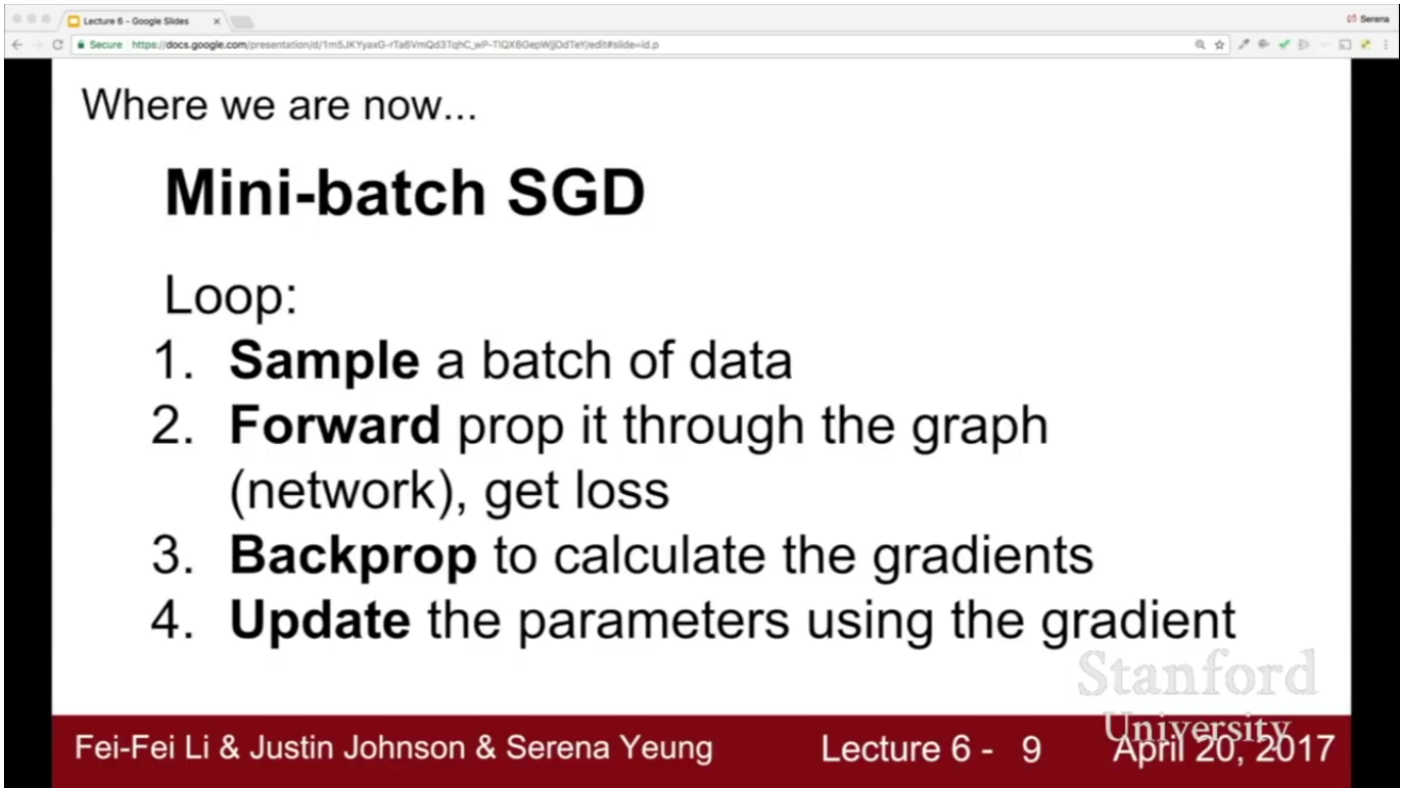
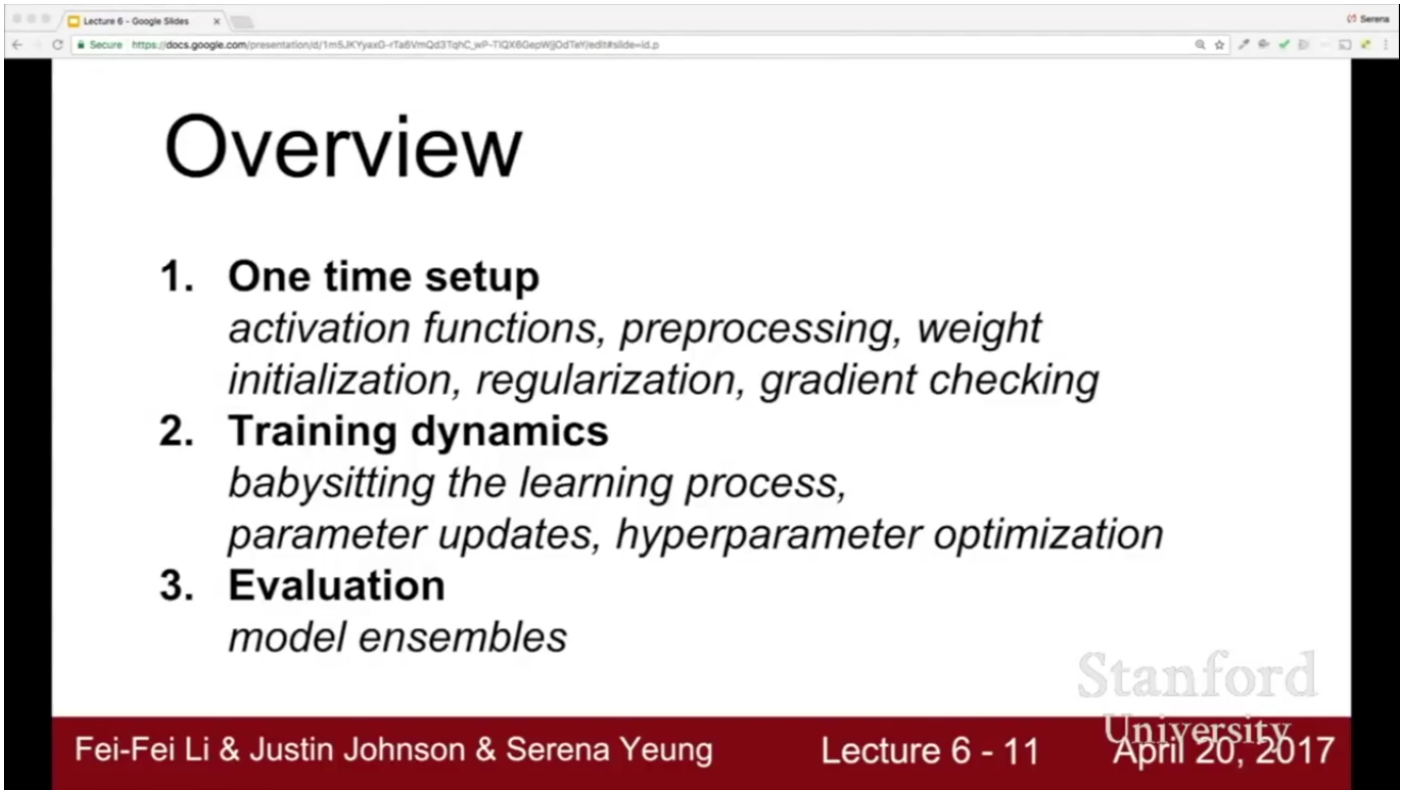
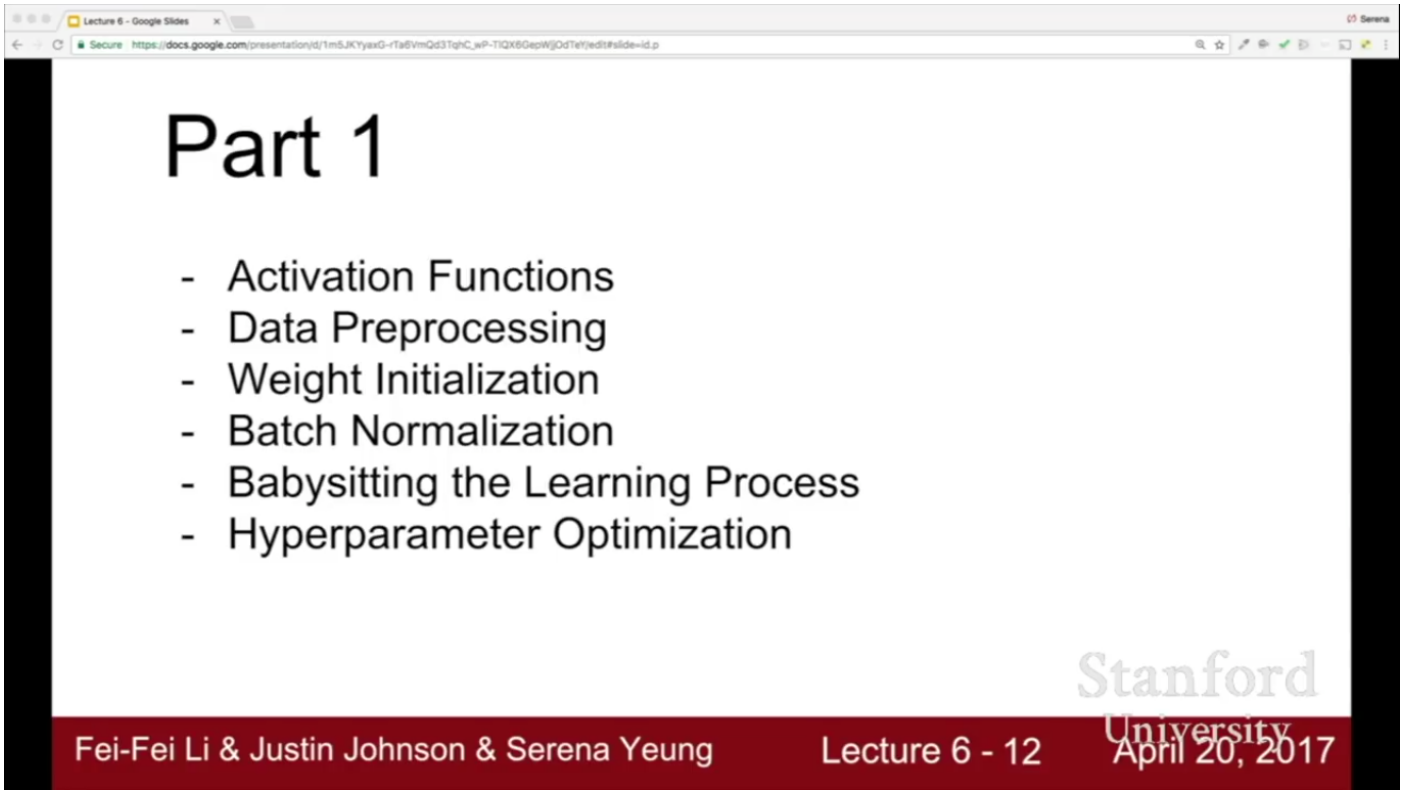
lec6

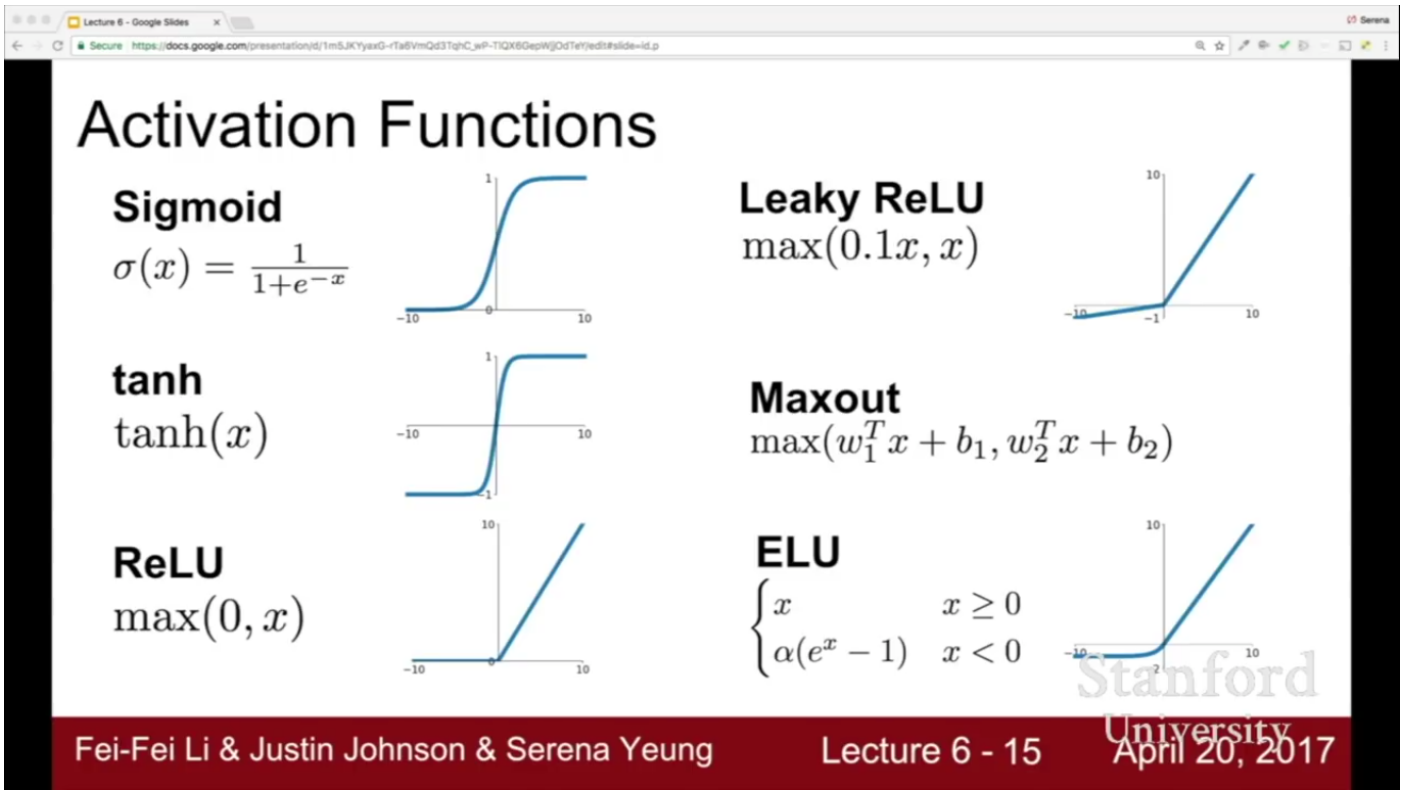
Creation date: 09/01/2020 13:31
Last Modified: 10/01/2020 13:40

Lec 6: Training Neural Networks



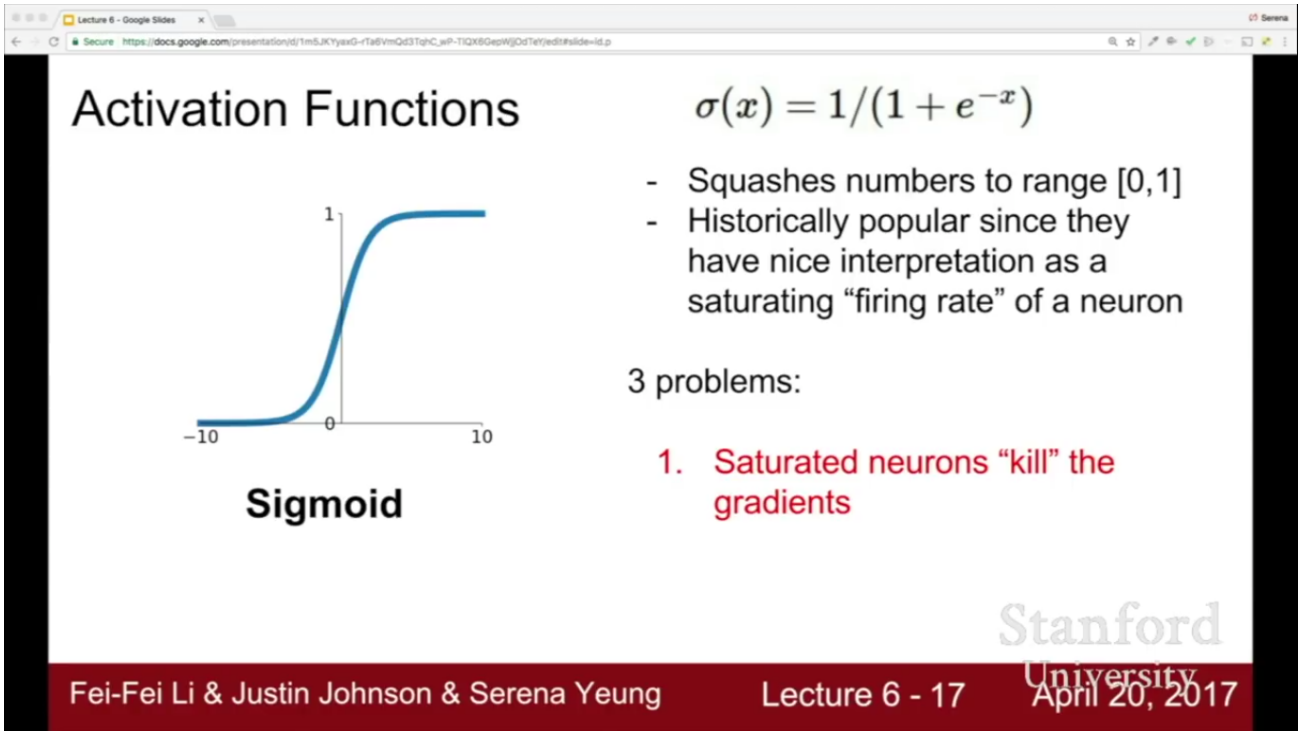


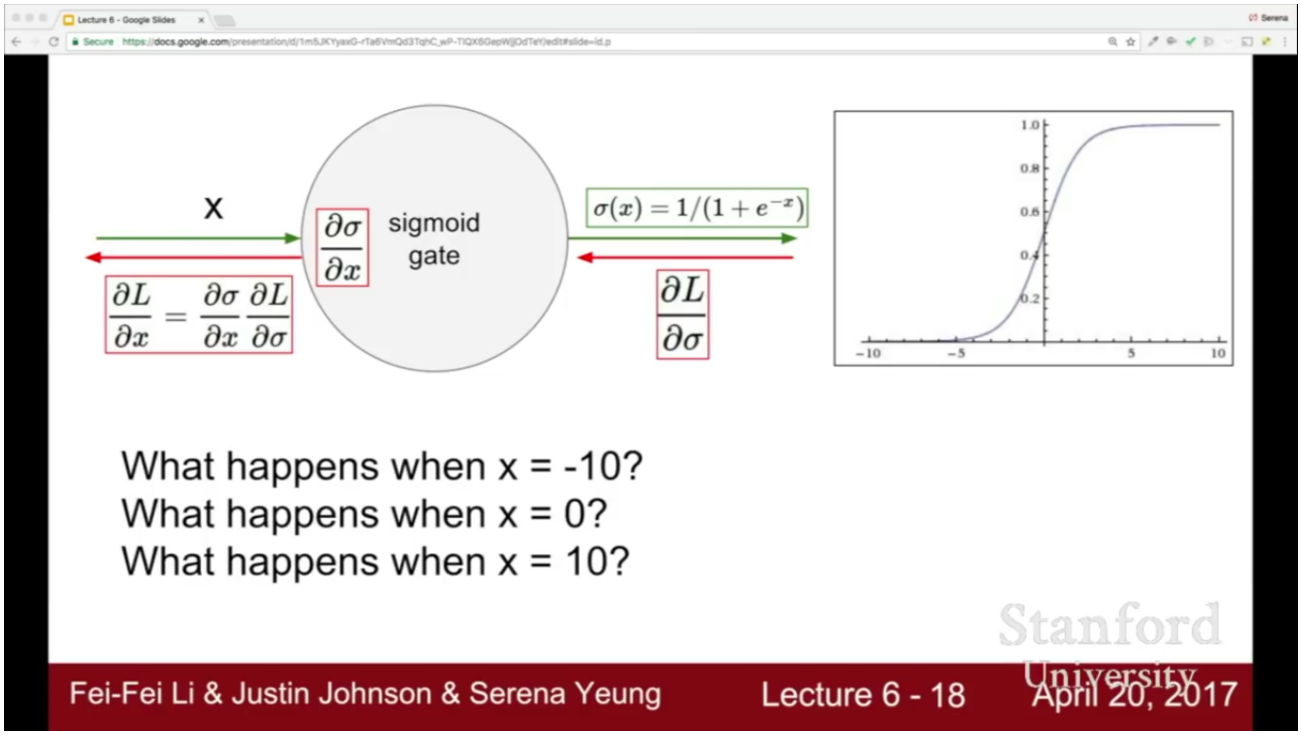




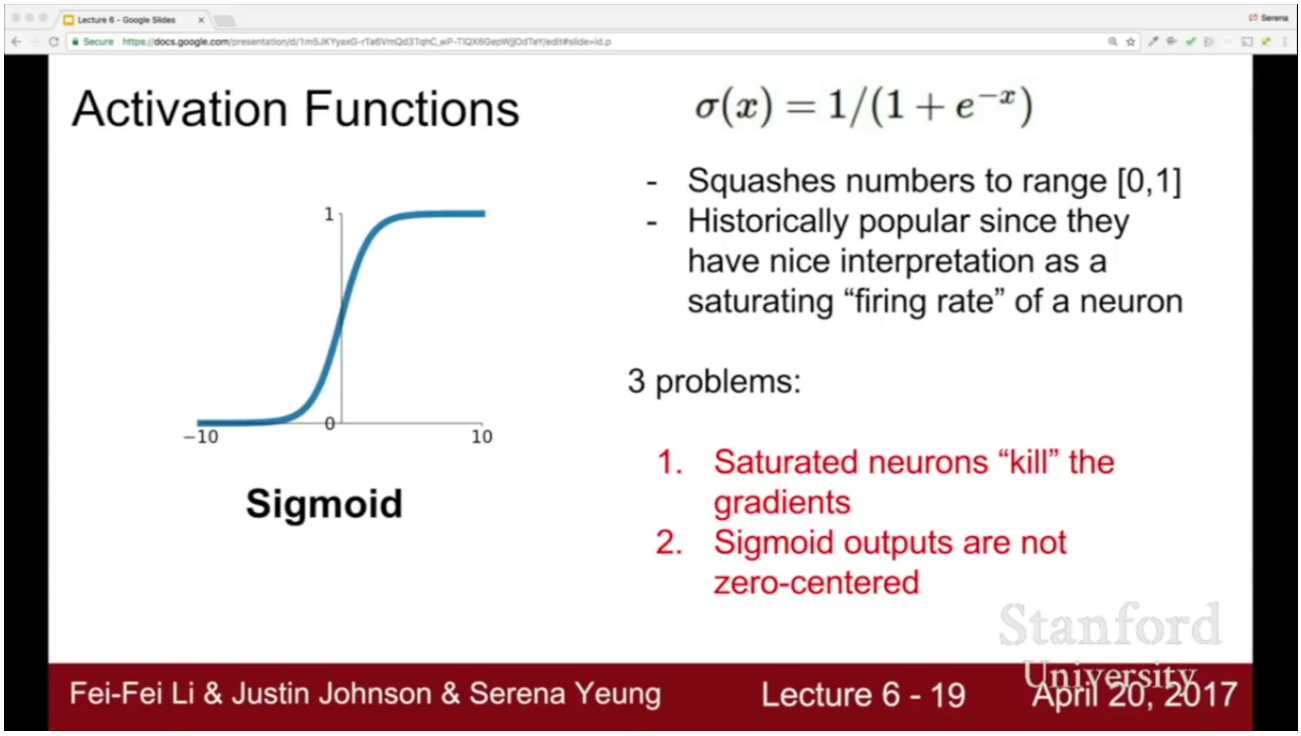
Activation Functions:

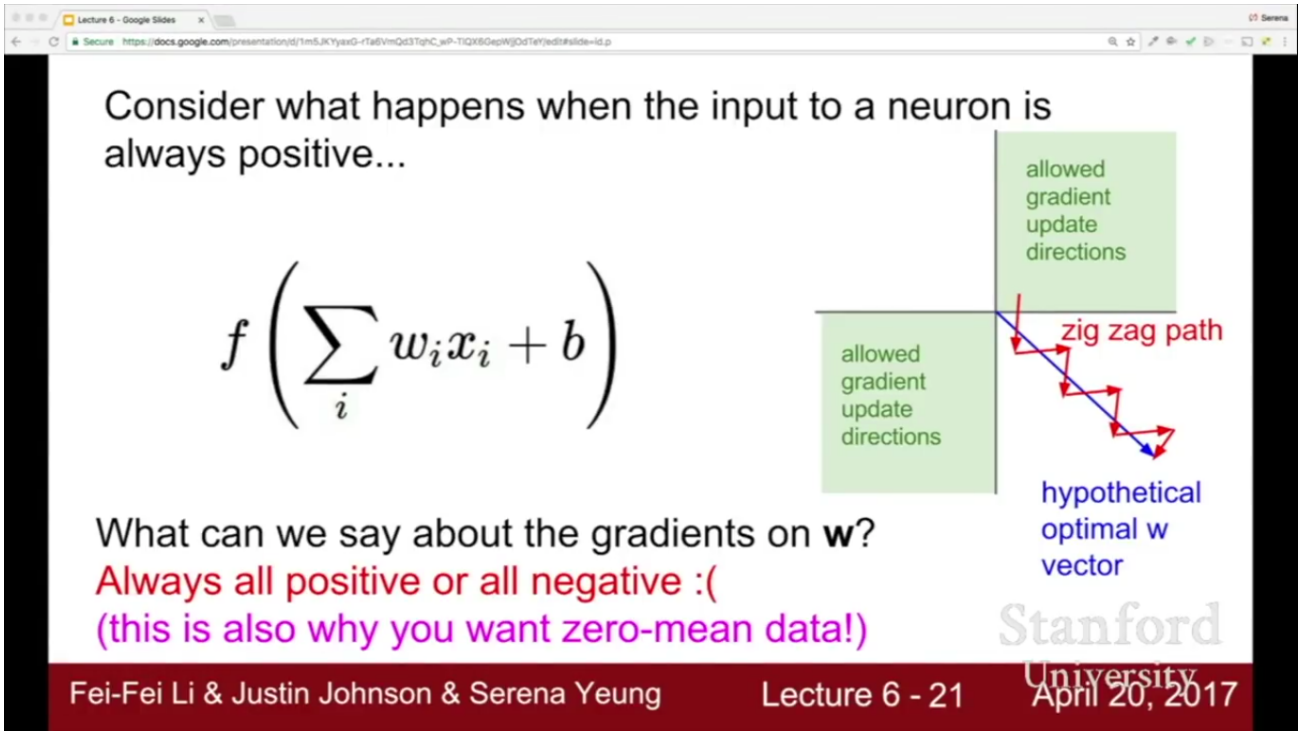
- Sigmoid

- 

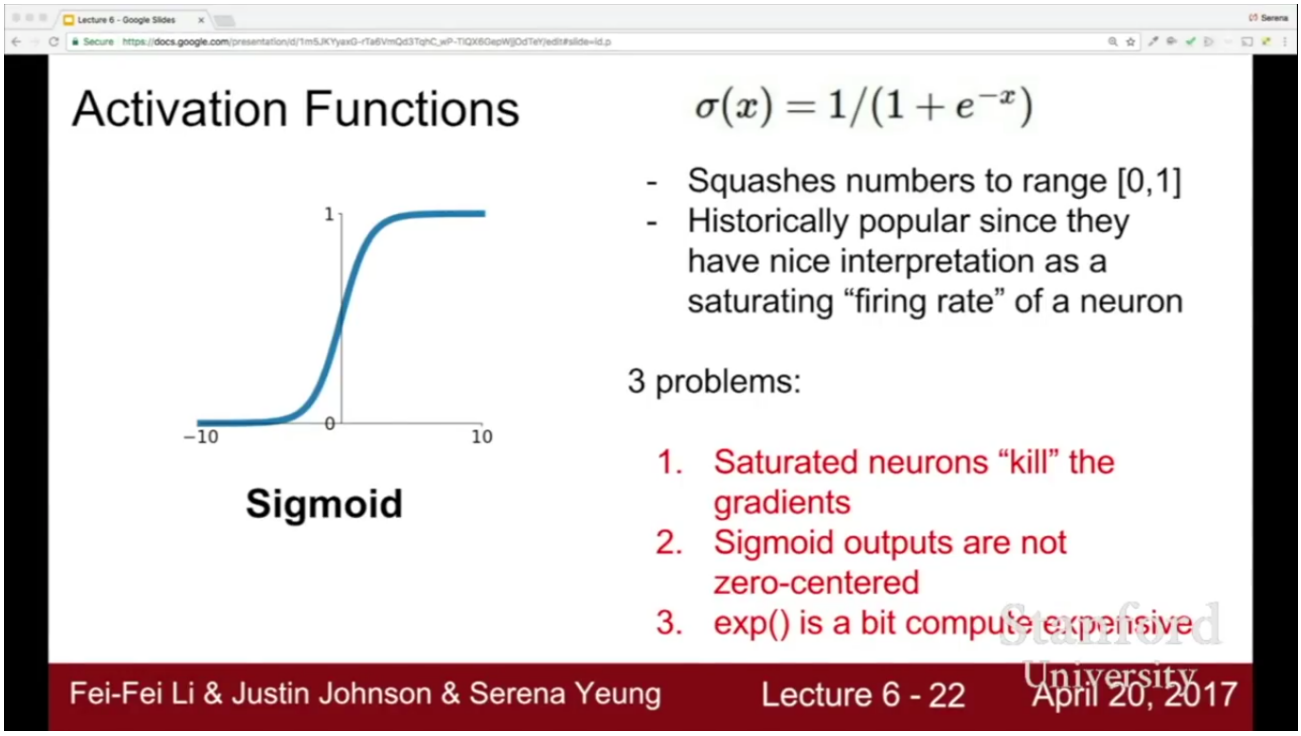
- 

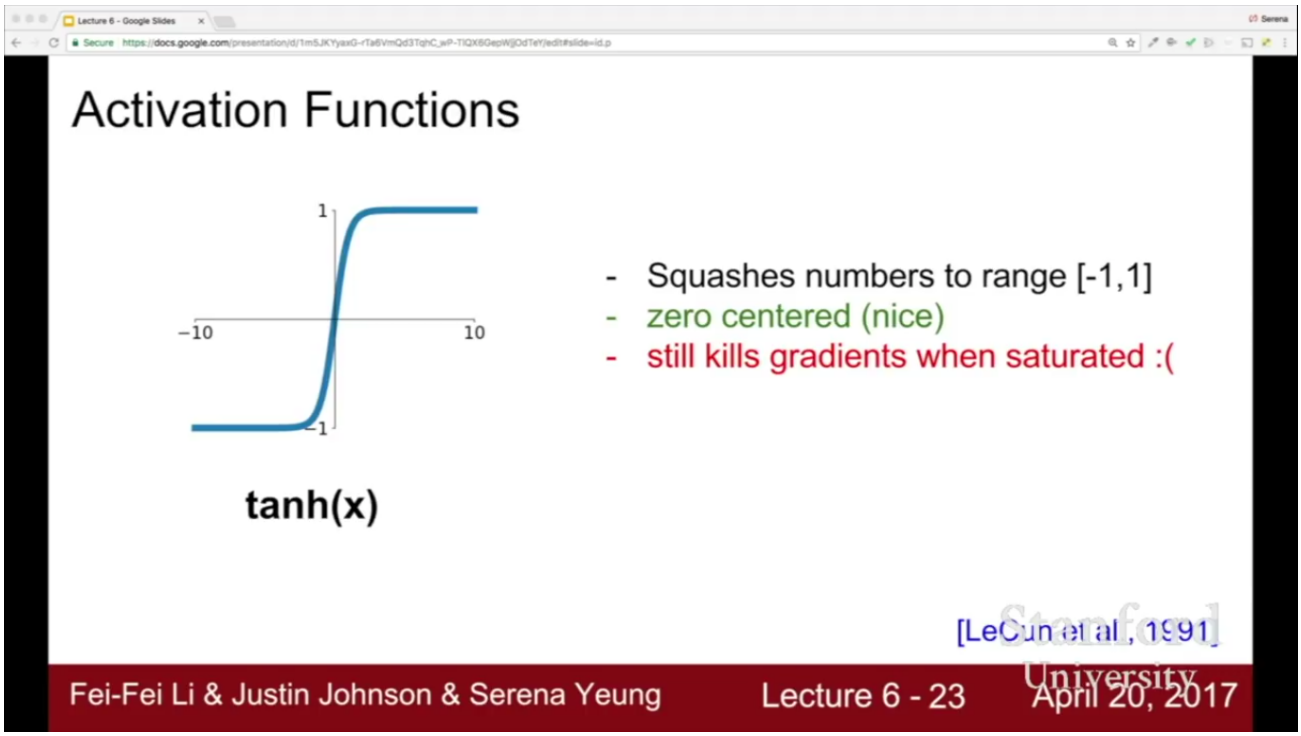
- At 10 or -10 the gradient of the sigmoid layer will be 0 causing no gradient to be learnt.
- Only at the values near to x=0 we will have some gradients.

- 

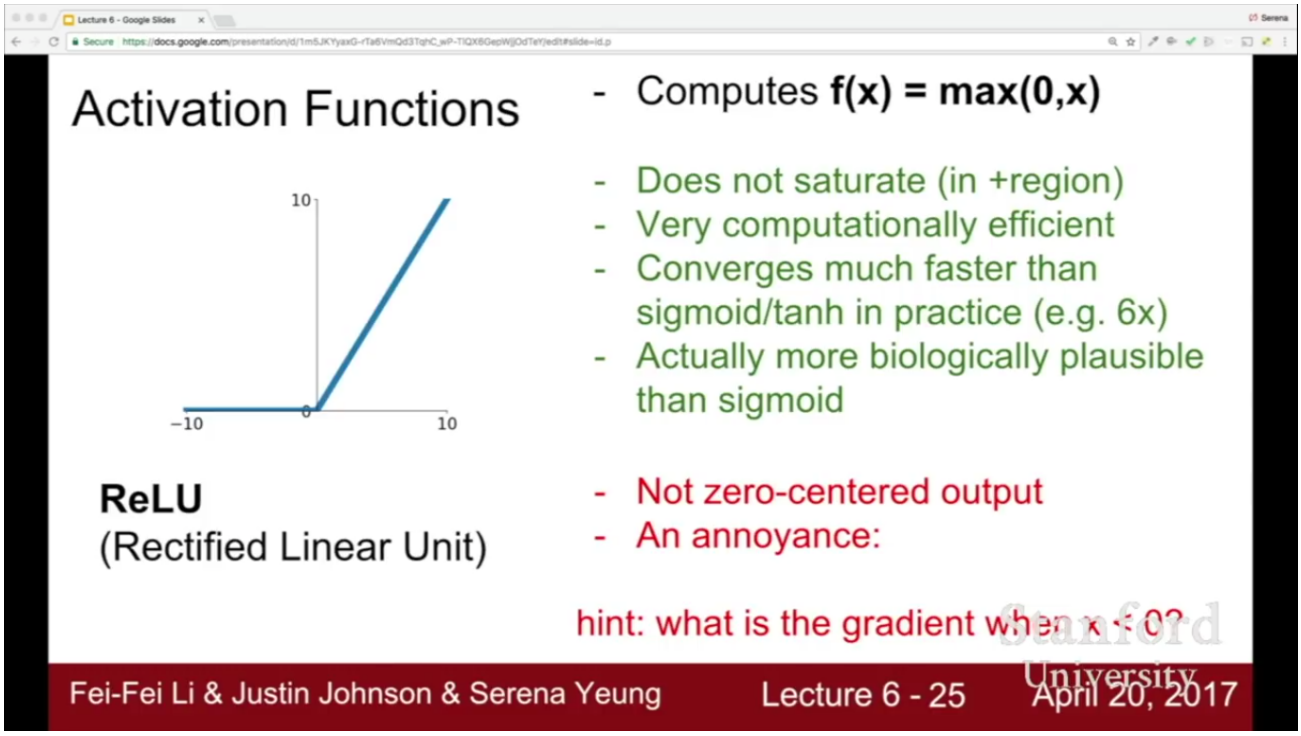
-


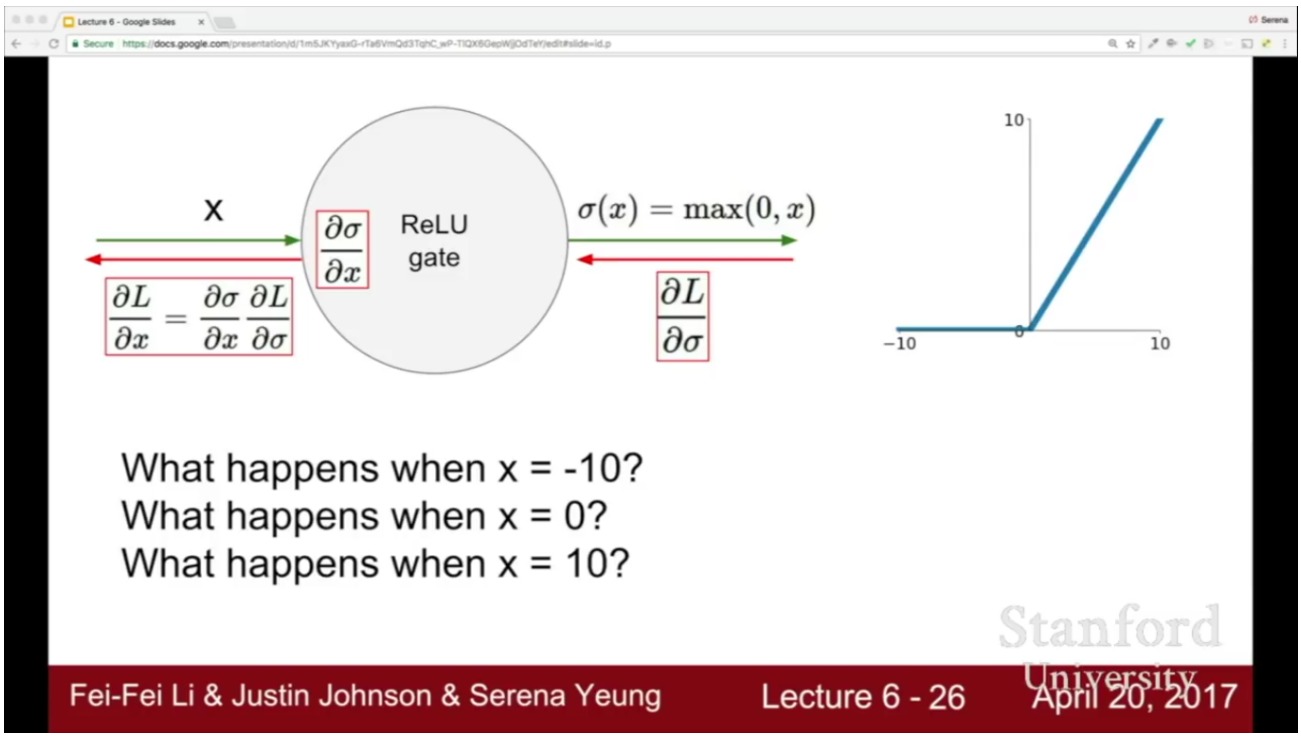
- Because the gradient for W will depend on x and if x is positive we will have only +ve gradients or when x is negative we'll have only -ve gradients.

- 
Activation Functions
$\sigma(x) = 1/(1+e^{-x})$
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron
3 problems:
1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive
1
0
−10
10
Sigmoid
Fei-Fei Li & Justin Johnson & Serena Yeung
Lecture 6 - 22
April 20, 2017
Stanford University

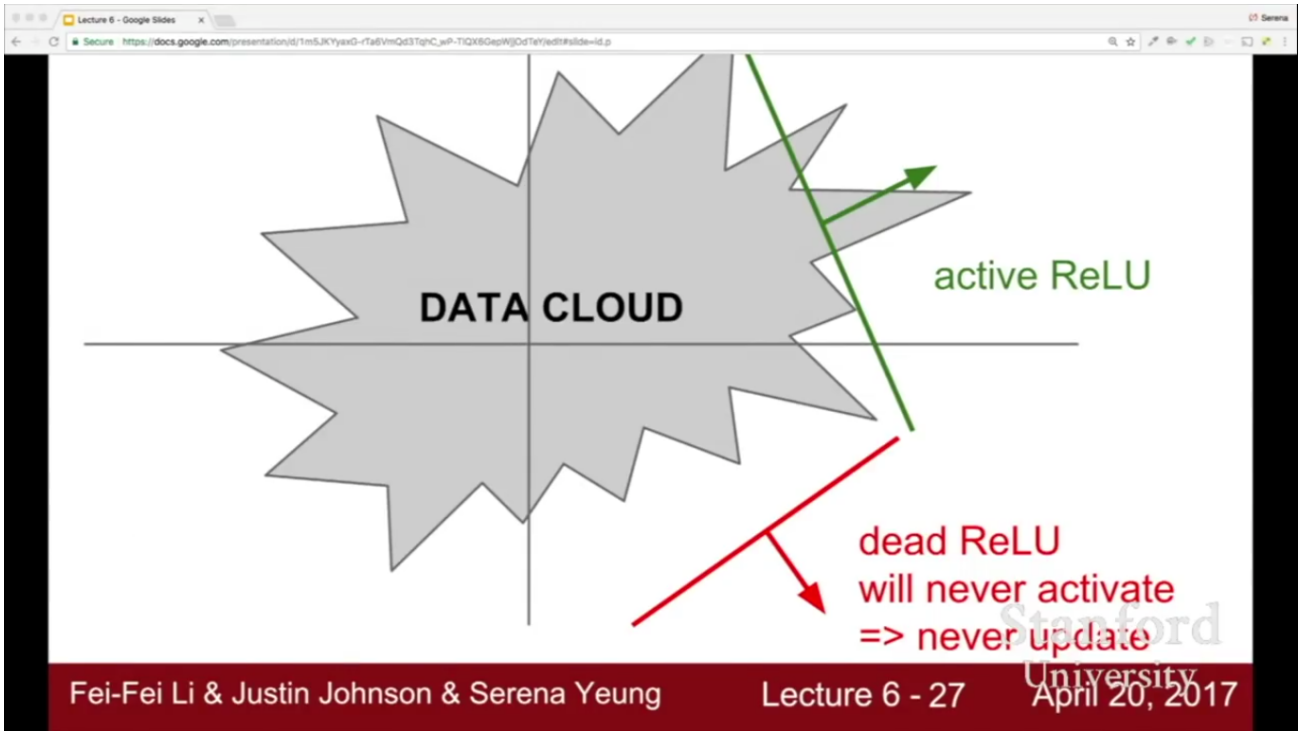
- TanH
  - 
Activation Functions
- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(
1
−1
−10
10
tanh(x)
[LeCun et al., 1991]
Fei-Fei Li & Justin Johnson & Serena Yeung
Lecture 6 - 23
April 20, 2017
Stanford University

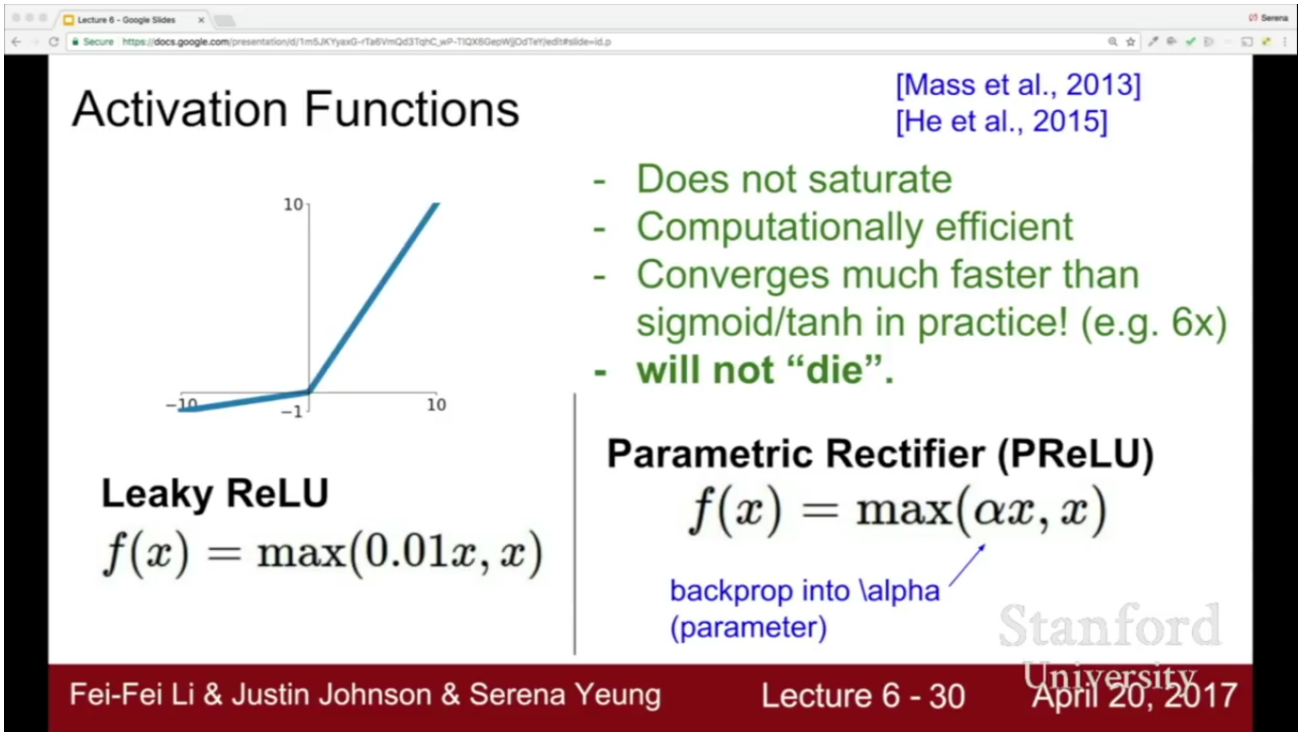
- ReLU

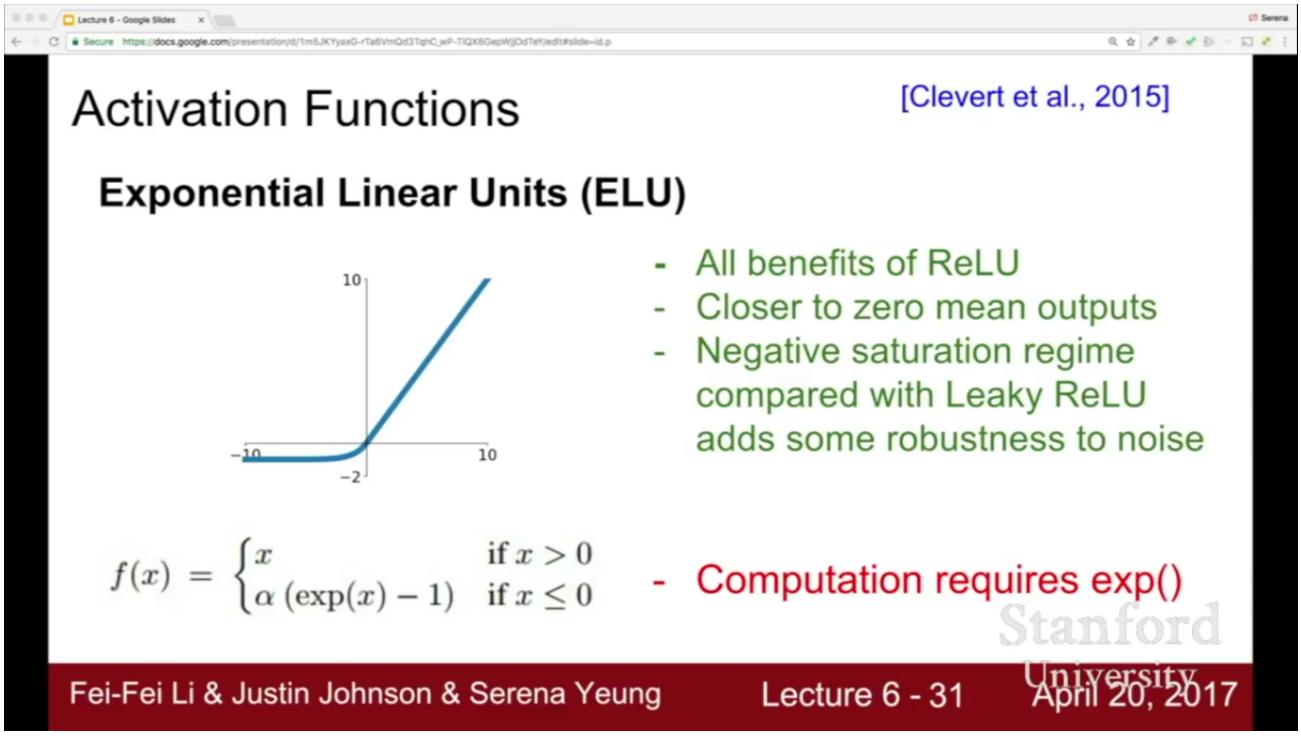
- 

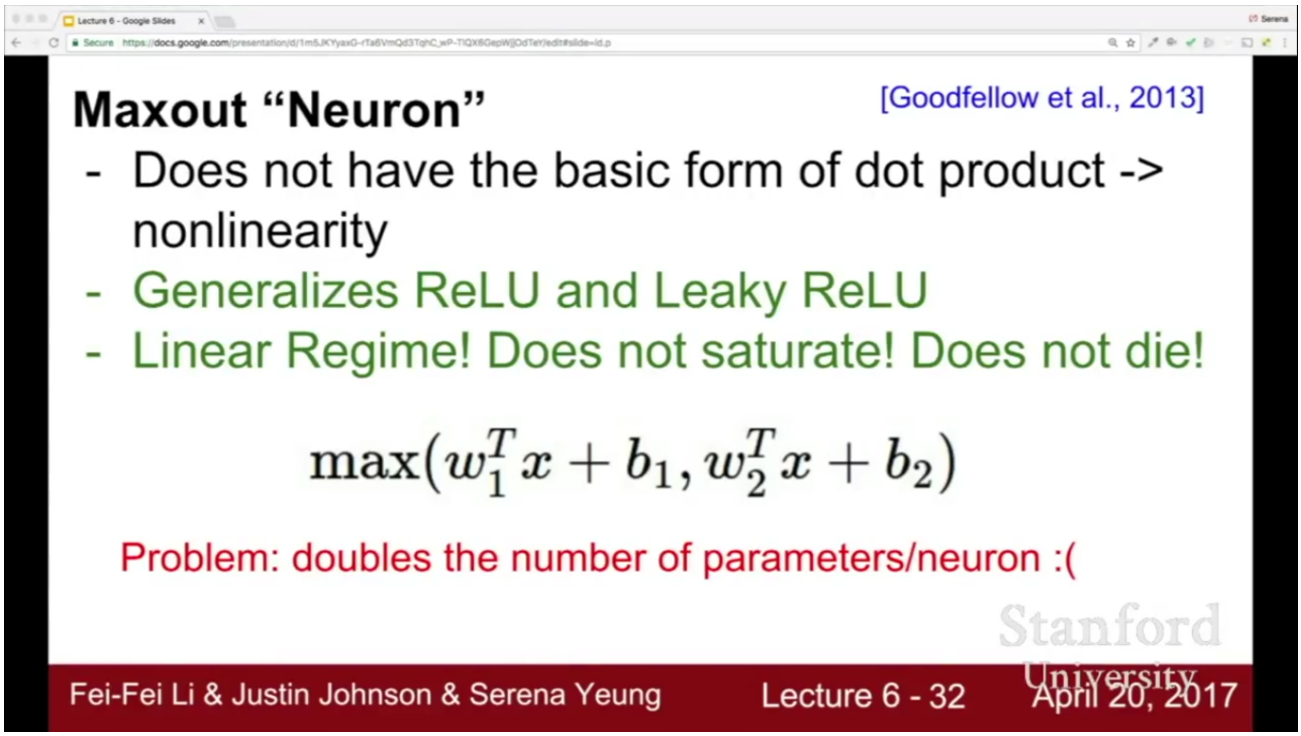
- 

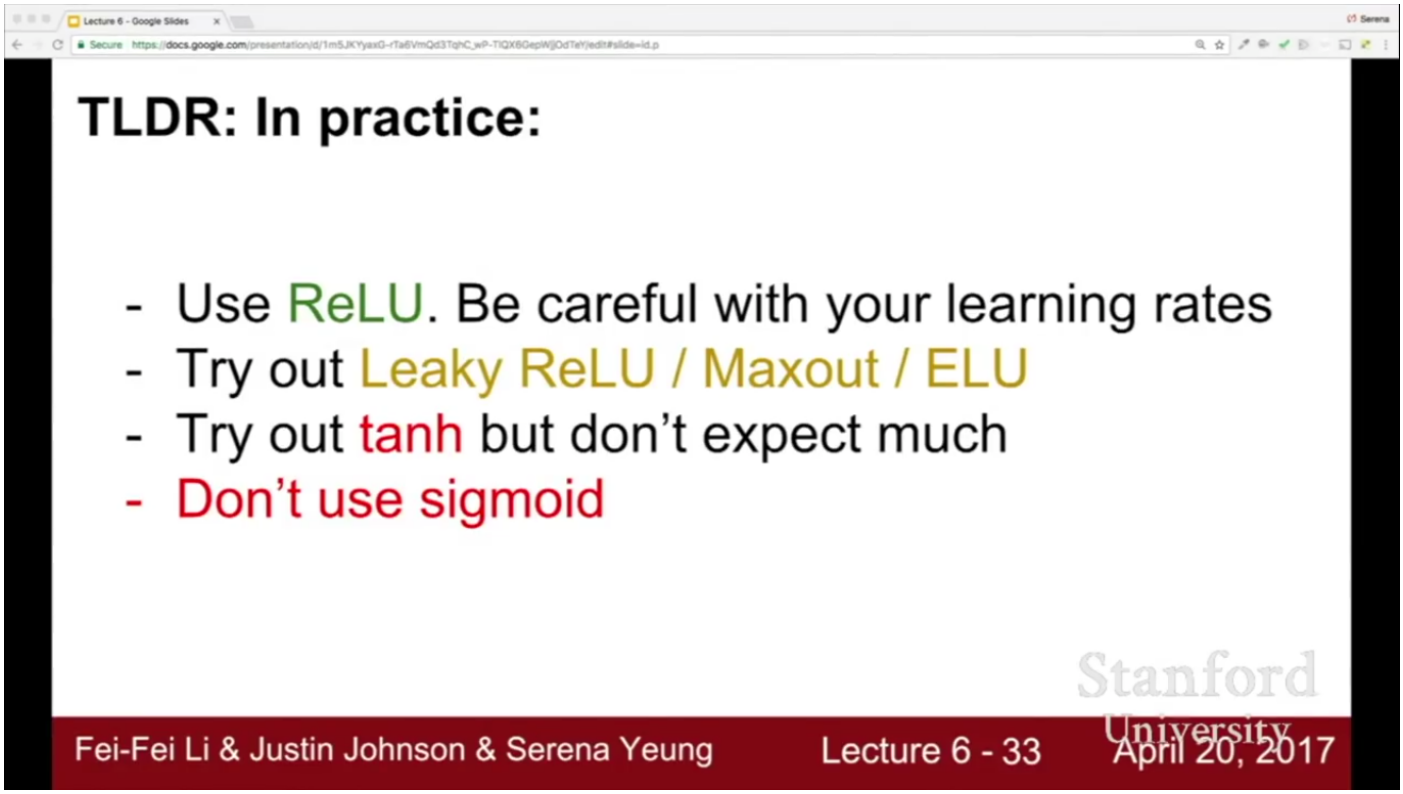
- For x = -10 , the gradient is 0.
- For x = +10, we have a linear function so we have a constant gradient value.
- For x = 0, it is undefined but for practical purposes we consider the gradient to be 0 at x=0.
- So in reality we have only half of the space available to find the gradients.

- 

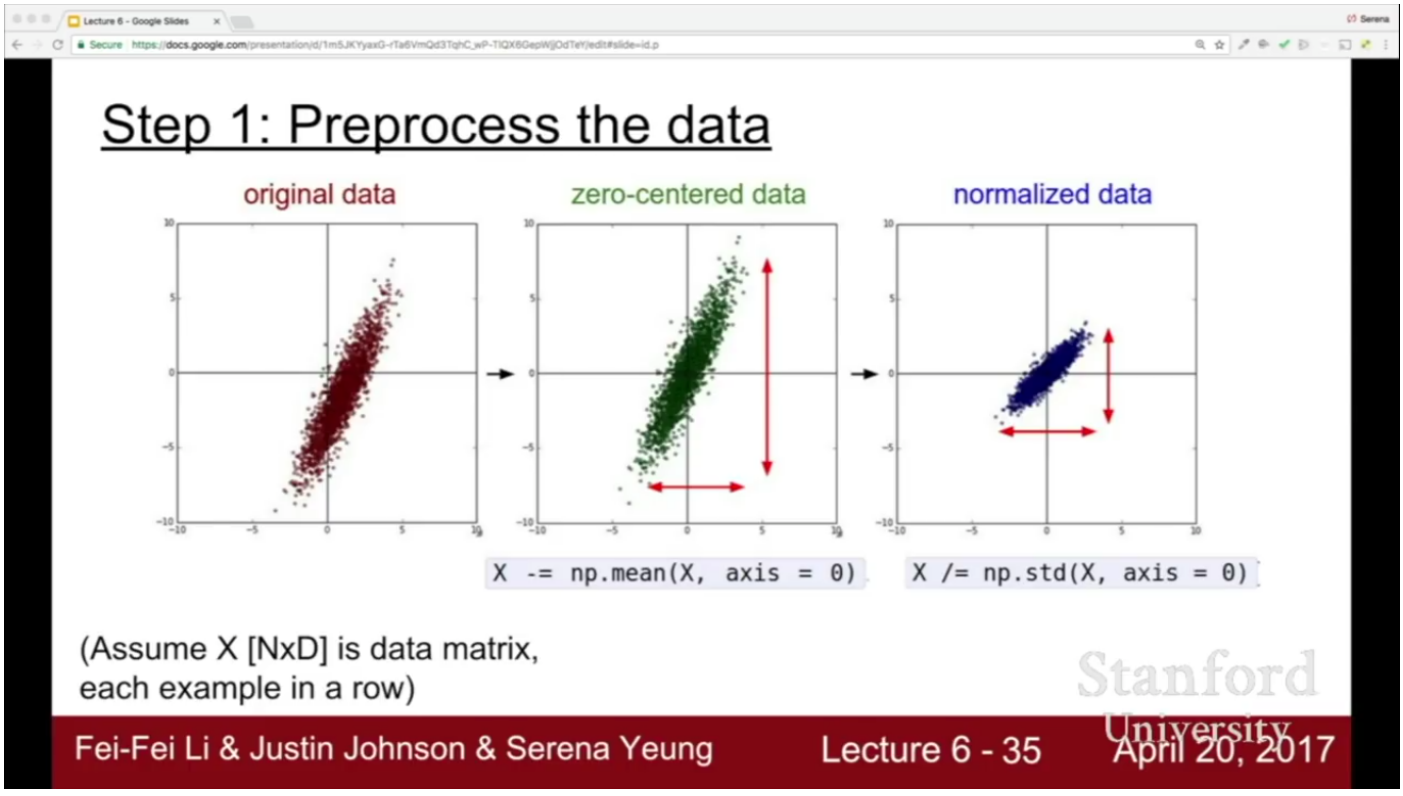
- So some region in the possible gradient region will never contribute to any gradients.
- **So it depends on the value of initialization also.**
- **Even if we have a very high learning rate, the update will oscillate and it can knock of the values of the gradient to the dead ReLU region and never update henceforth.**
- ***This problem is still existing and it is still an open research problem but never the less the training happens and it works but around 10% of the neurons will be in the dead ReLU region.***
- ***People also initialize the ReLU with slightly positive biases (like 0.01).*** So we have activate values. But initializing with 0 bias is also common.
- 

- 

- MaxOut "Neuron":
  - 

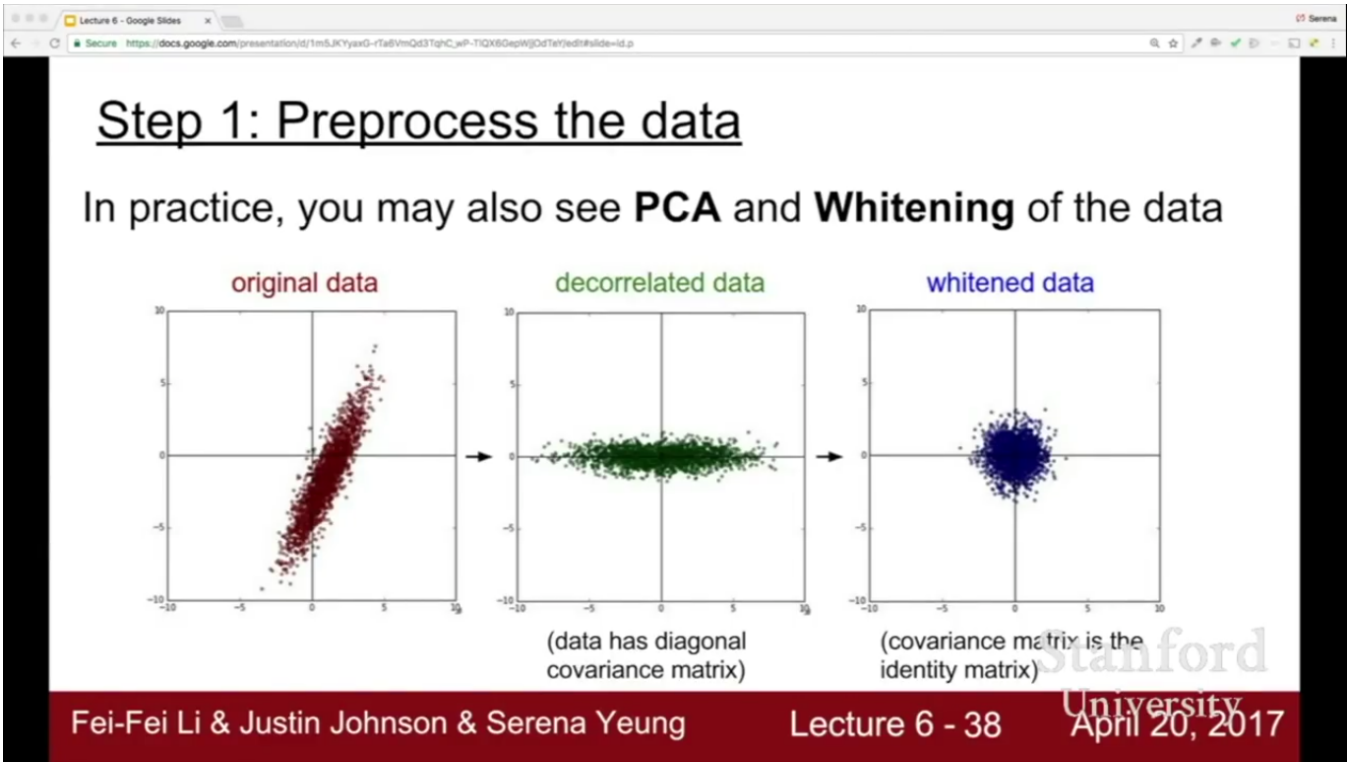


Data Preprocessing:

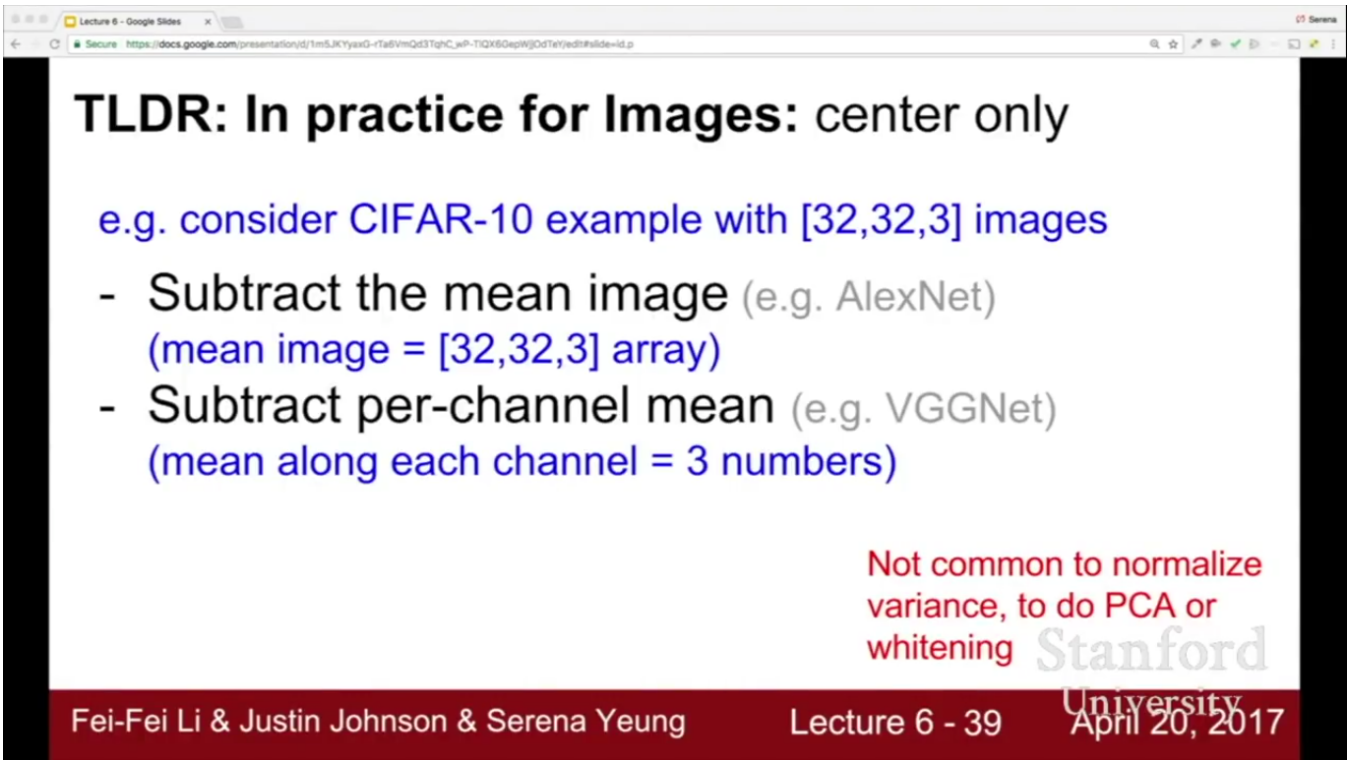


- we normalize it by standard deviation for most of the cases.
- We do preprocessing to bring the values in the same range.
- For images we don't usually normalize the data, we just do zero-centering assuming that the values are normally distributed.

- But in other machine learning problems we do zero-centering and also perform normalization.

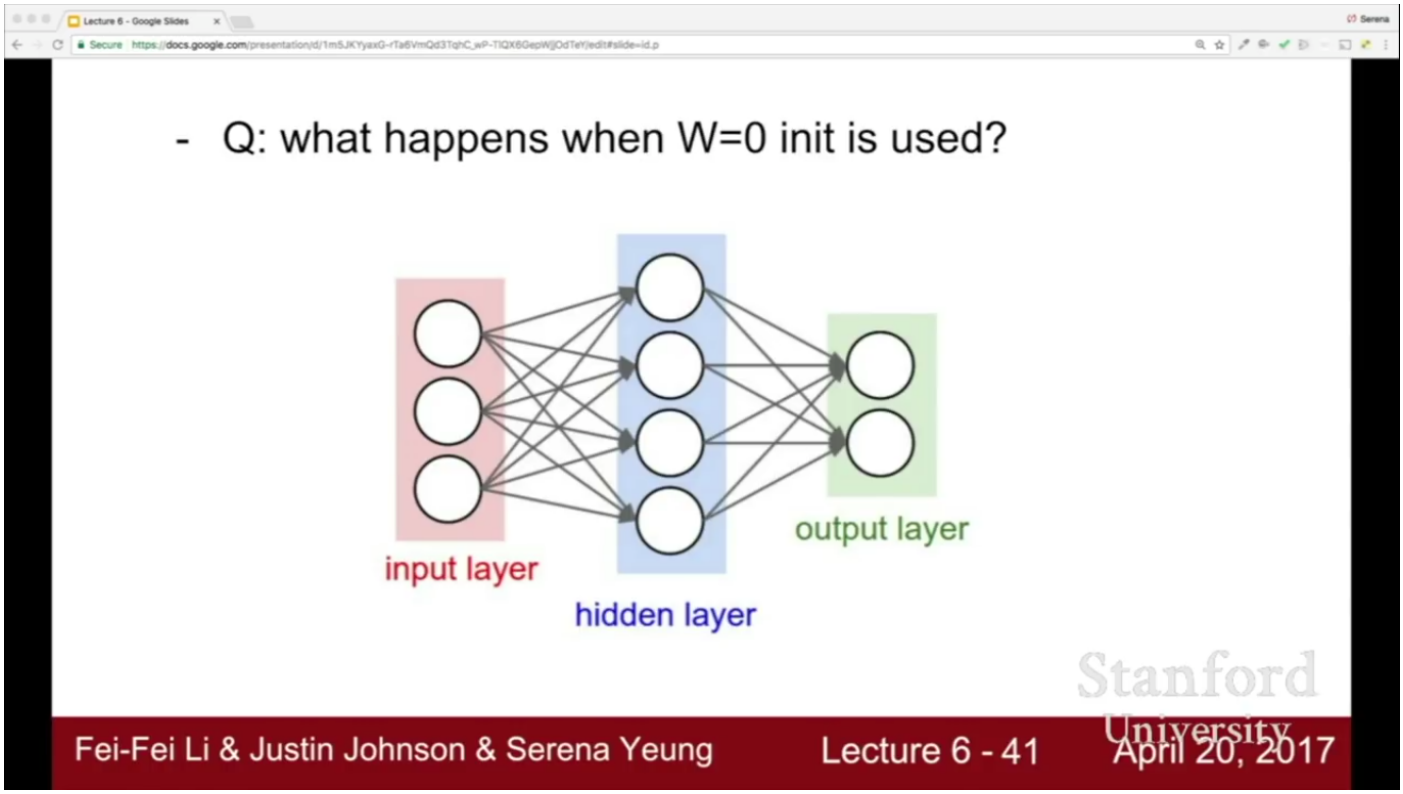


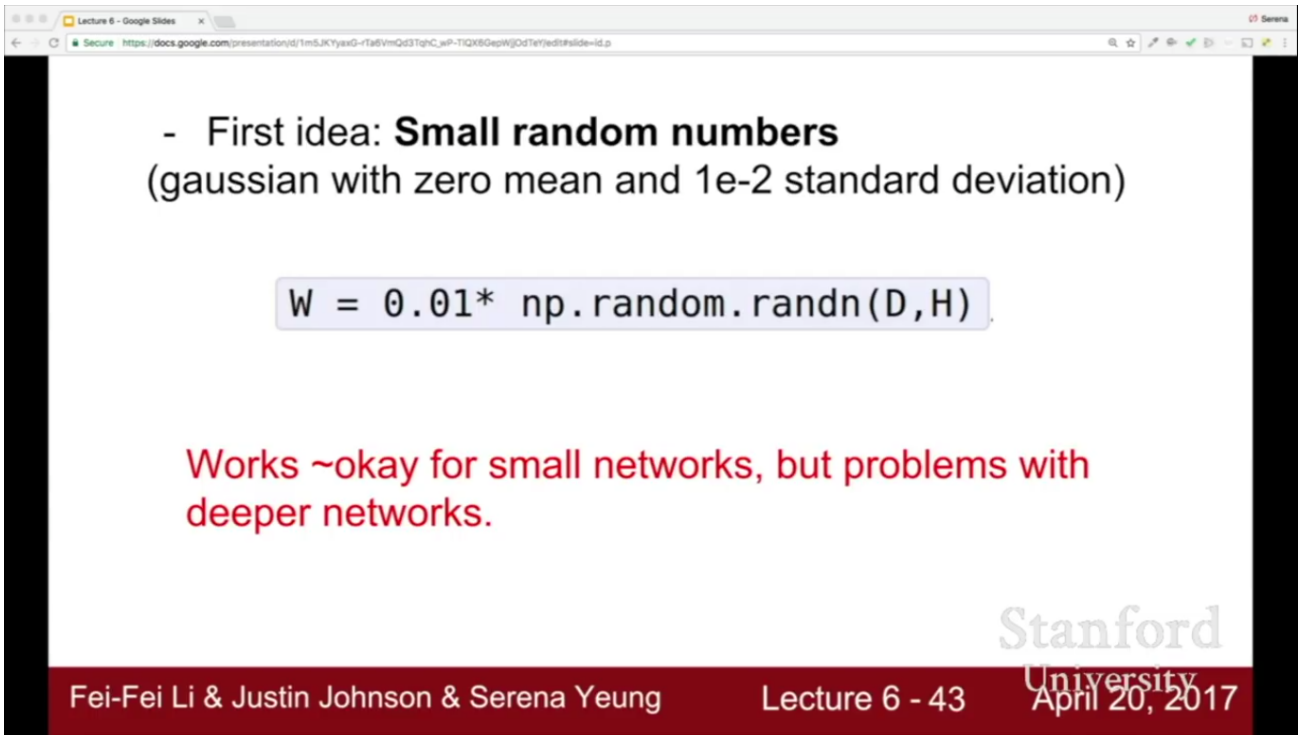
- We also perform **PCA to decorrelate** the data and **whitening to normalize the zero-centered data.**

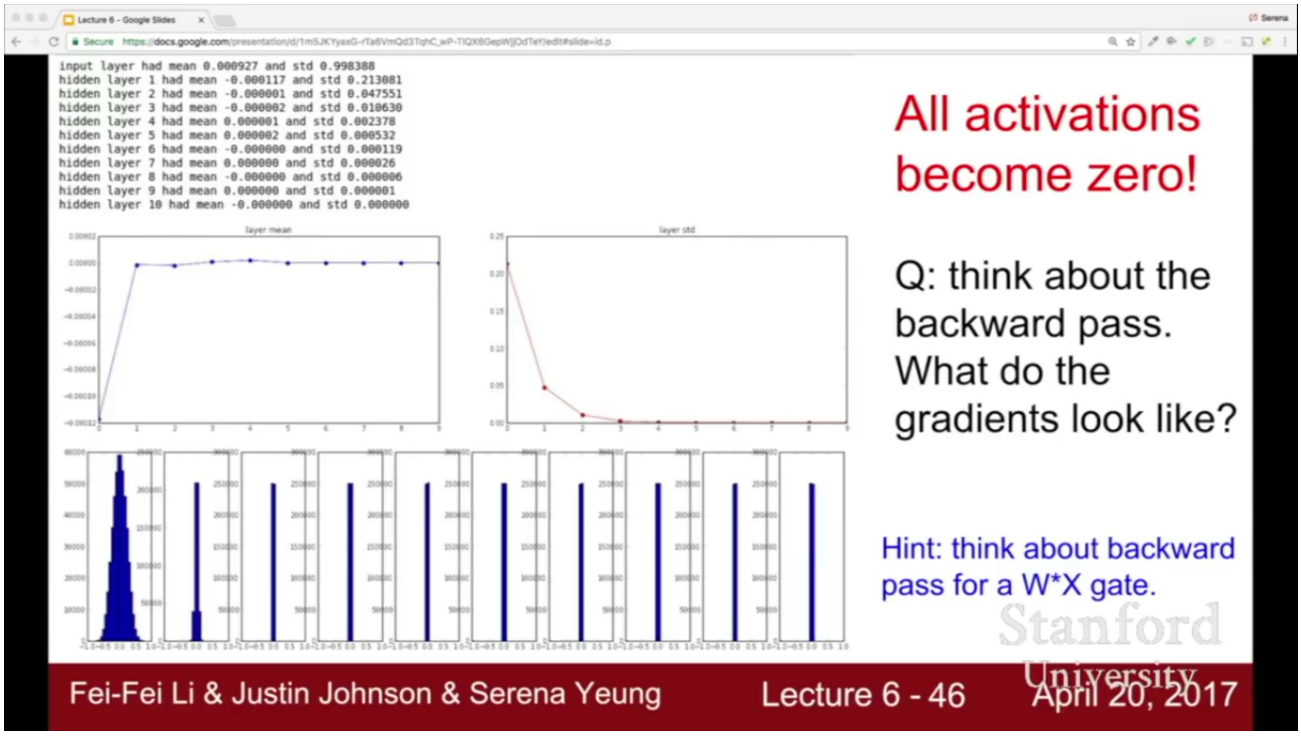


- **FOR IMAGE DATA WE DONT HAVE TO NORMALIZE THE VARIANCE, OR TO PCA OR WHITENING.**
- **IF TRAINING HAS CERTAIN PREPROCESSING DONE, WE HAVE TO PERFORM THE SAME PREPROCESSING WHILE TESTING.**
- THE MEAN IS FOUND OVER THE ENTIRE TRAINING DATA.
- We can estimate the MEAN for a batch too if the batch is of considerable size.

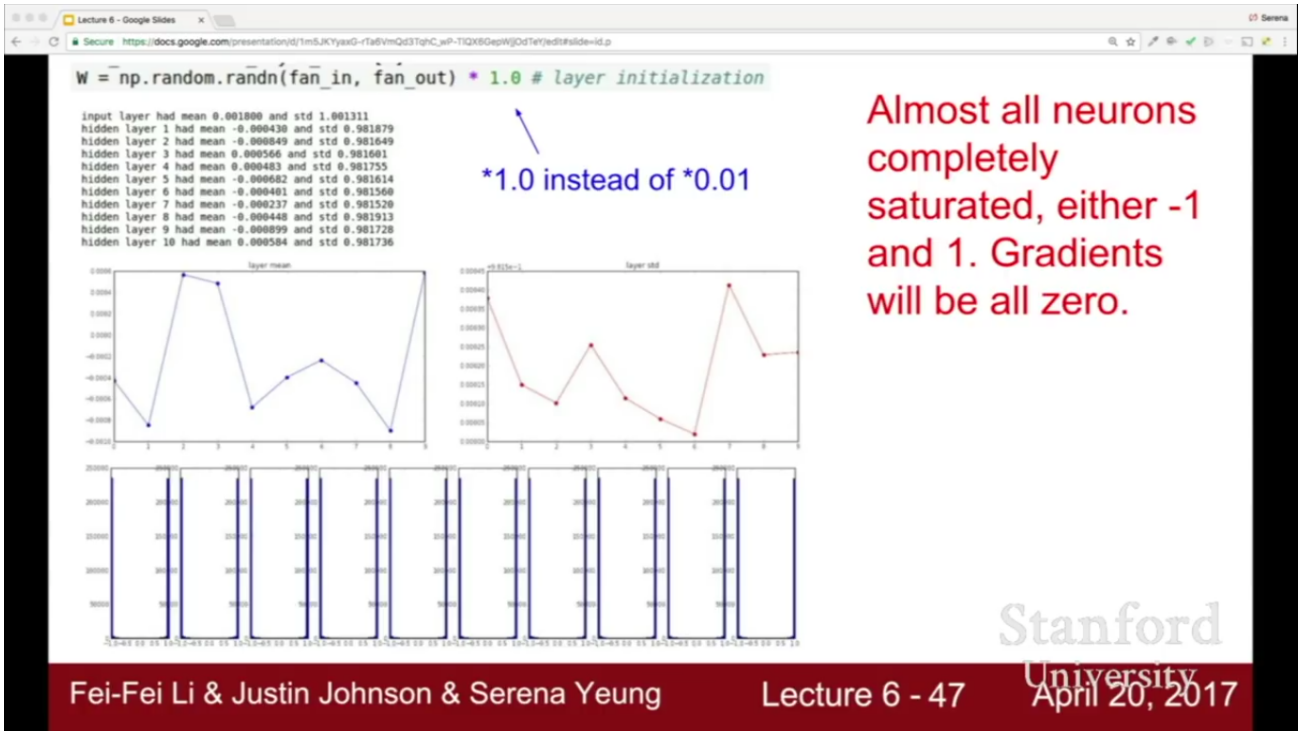
Weight Initialization:



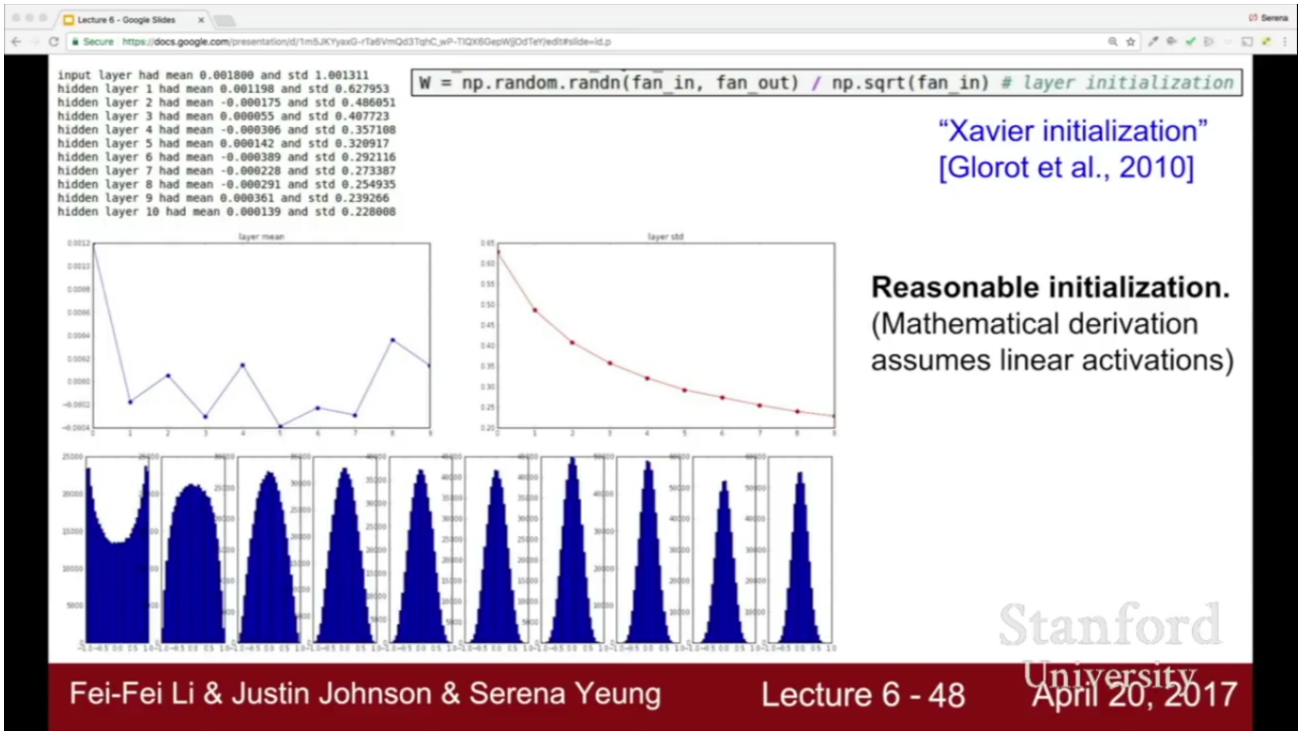
- If W=0, all the neurons will have the same value since activation(0.x + b) => activation(b)
- So since all the neurons have the same value, the gradients will also be the same for each neuron. So they will update in the same way. Hence there will be no considerable learning because we want each neuron to learn different things.
- There will be **NO SYMMETRY** if we initialize with W=0
- Solutions:
  - 

- 

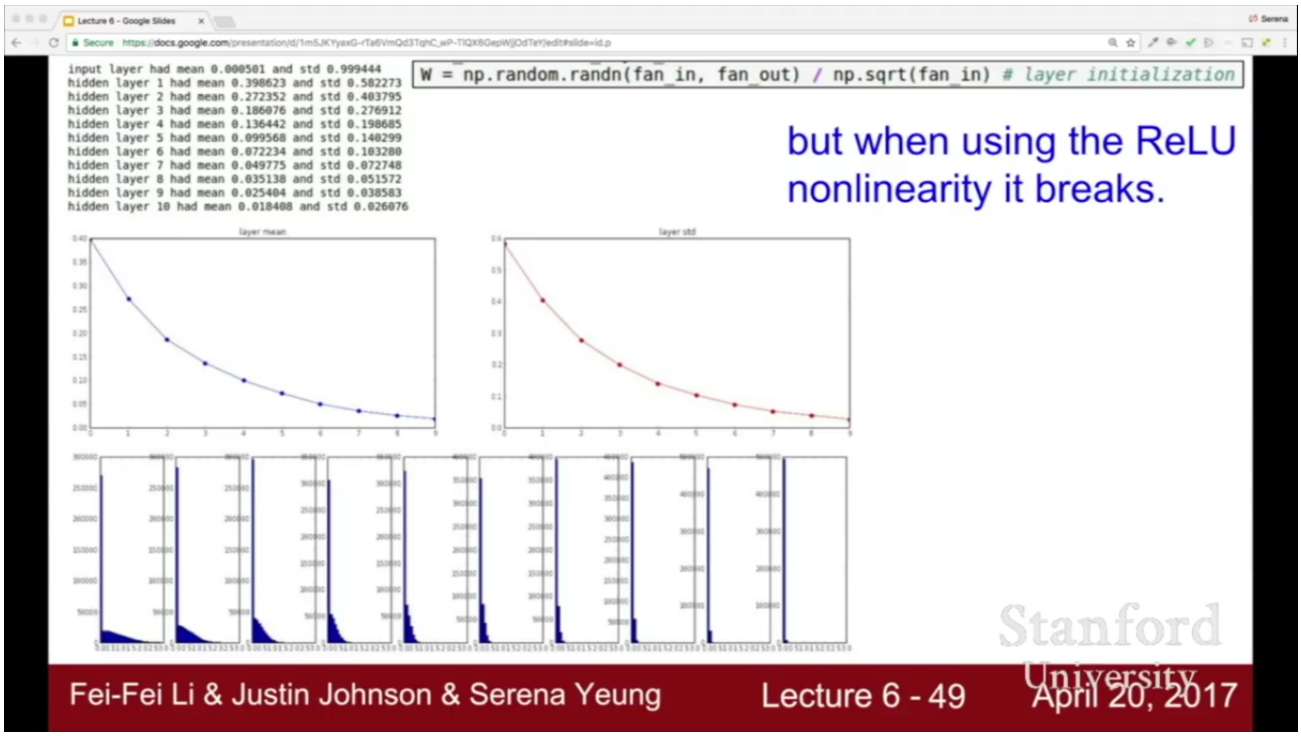
- We can notice in the line graph that all the layers have 0 mean value and the std is reaching 0. Hence all the values of Wx will result in 0. The local gradient will be multiplied with the upstream gradient multiplied with the local gradient value. For W, it depends on X, so eventually all the gradients will collapse to 0.

- 

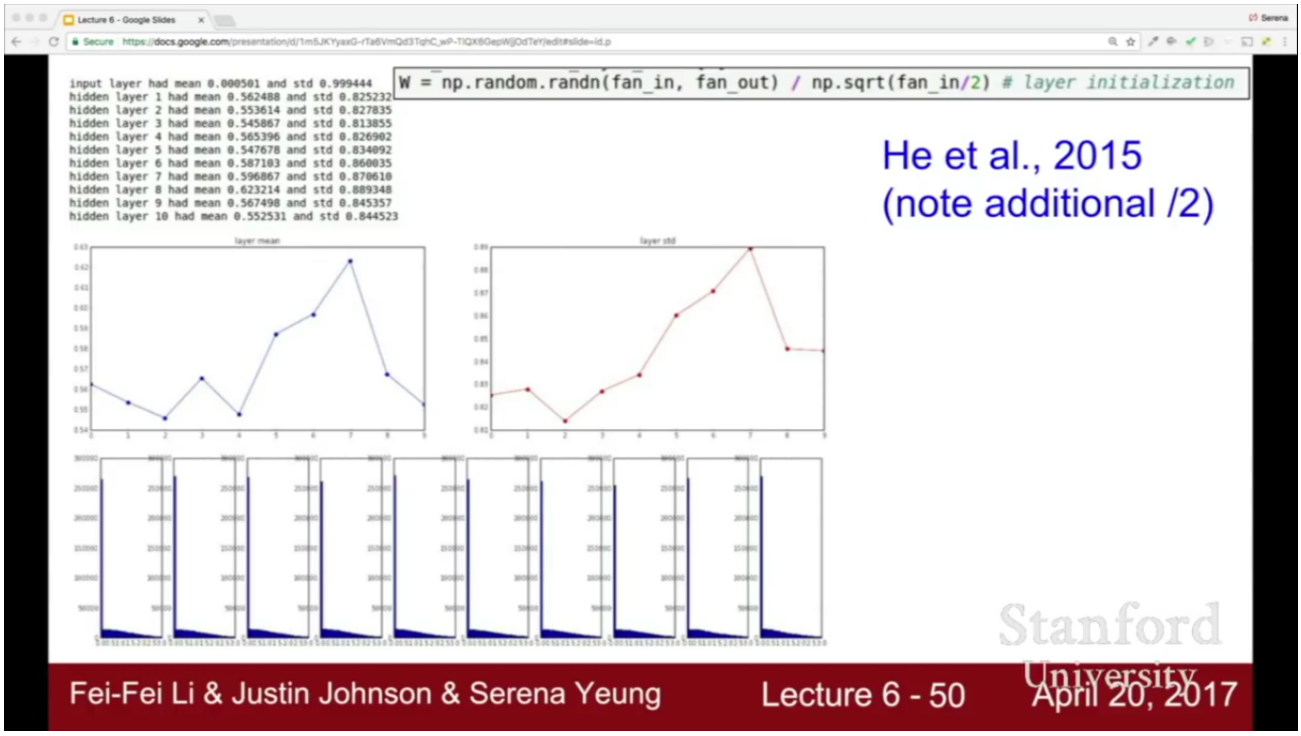
- If we use a std of 1.0 instead of 0.01 and tanH the values will get saturated between -1 and +1.

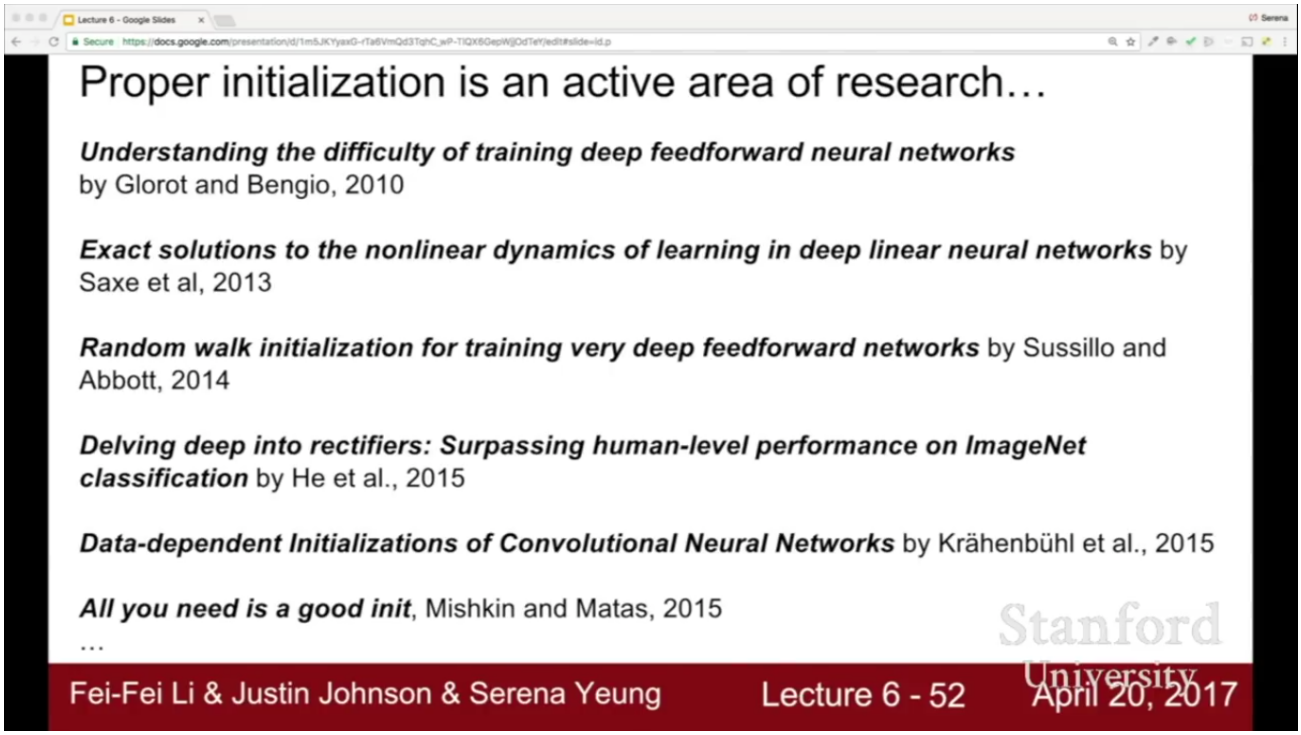
- 

- For smaller networks Xavier initialization is acceptable but not for deeper networks since eventually the std deviation is collapsing to 0. And Xavier initialization assumes the activation to be linear.

- 

- When we use ReLU activations, w.k.t half of the activation are 0 (for negative input values.).

- 

- He initialization has the additional term of dividing by 2 for compensating for half of the activation being 0 when using ReLU.
- 

Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

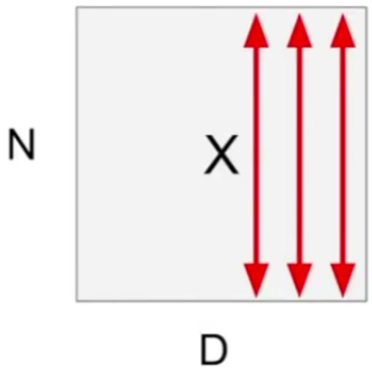
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

Batch Normalization

[Ioffe and Szegedy, 2015]

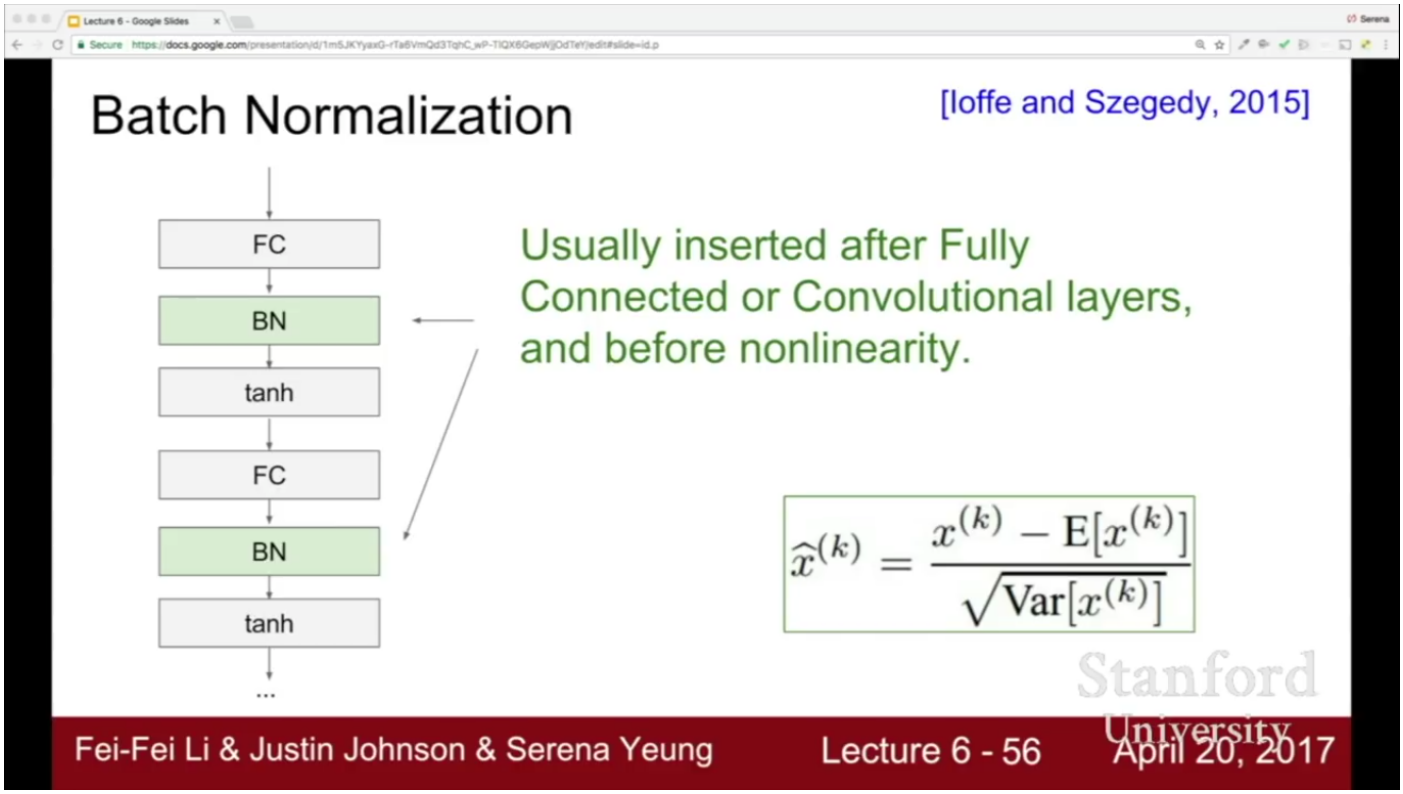
"you want unit gaussian activations?
just make them so."



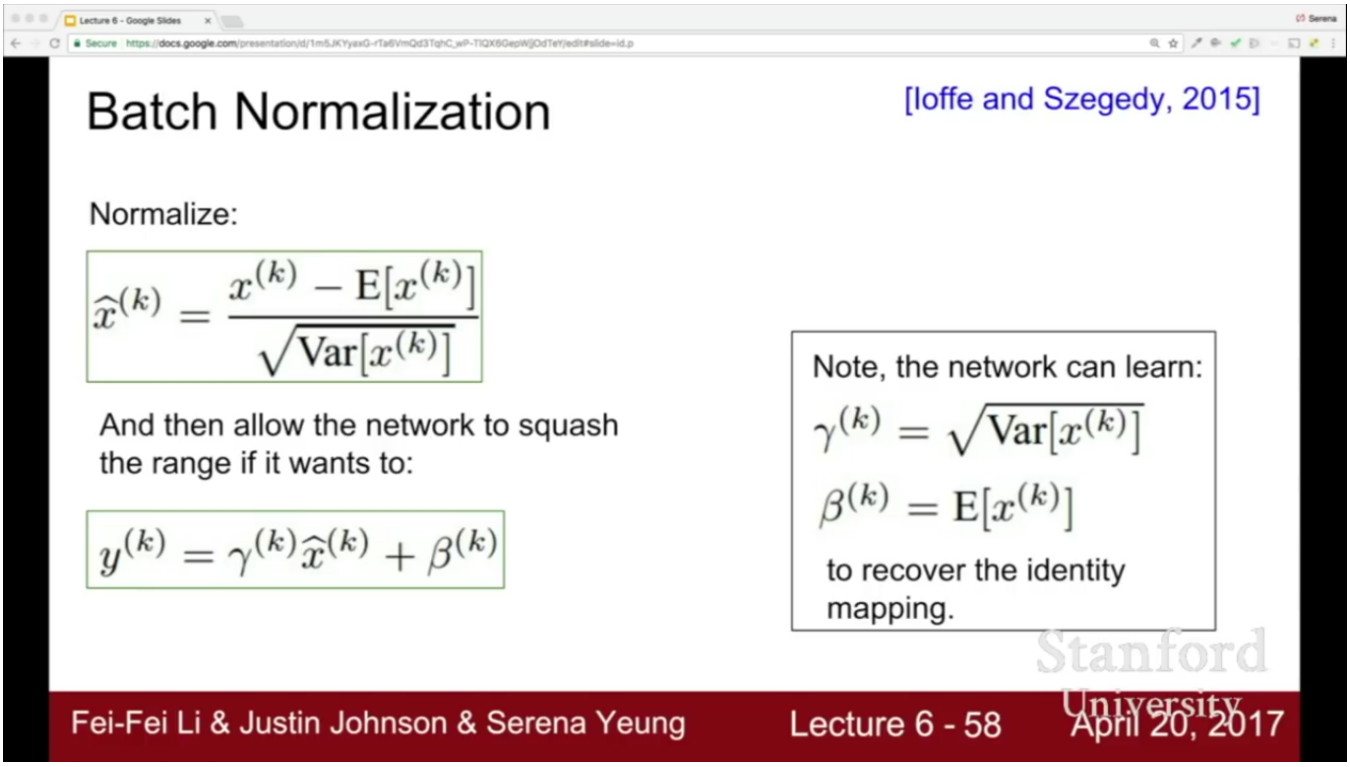
1. compute the empirical mean and variance independently for each dimension.

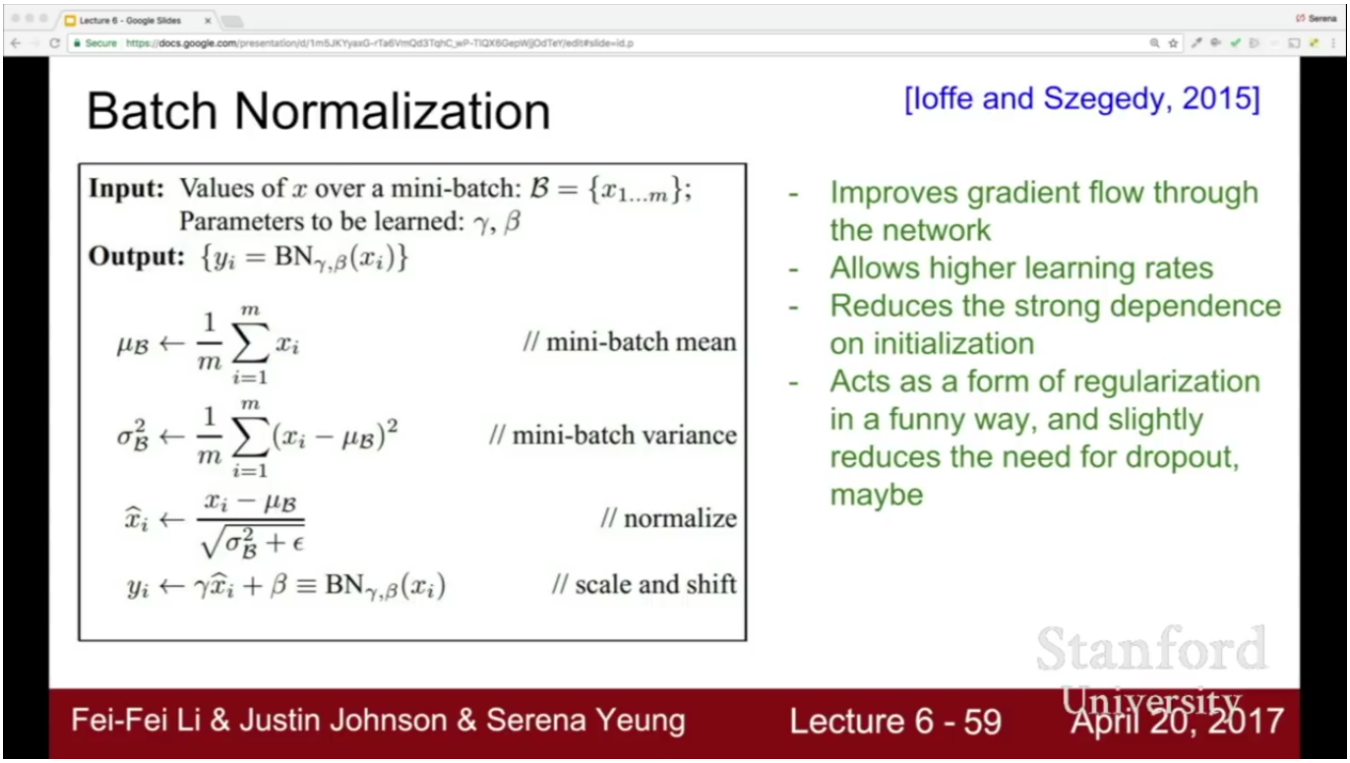
2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$



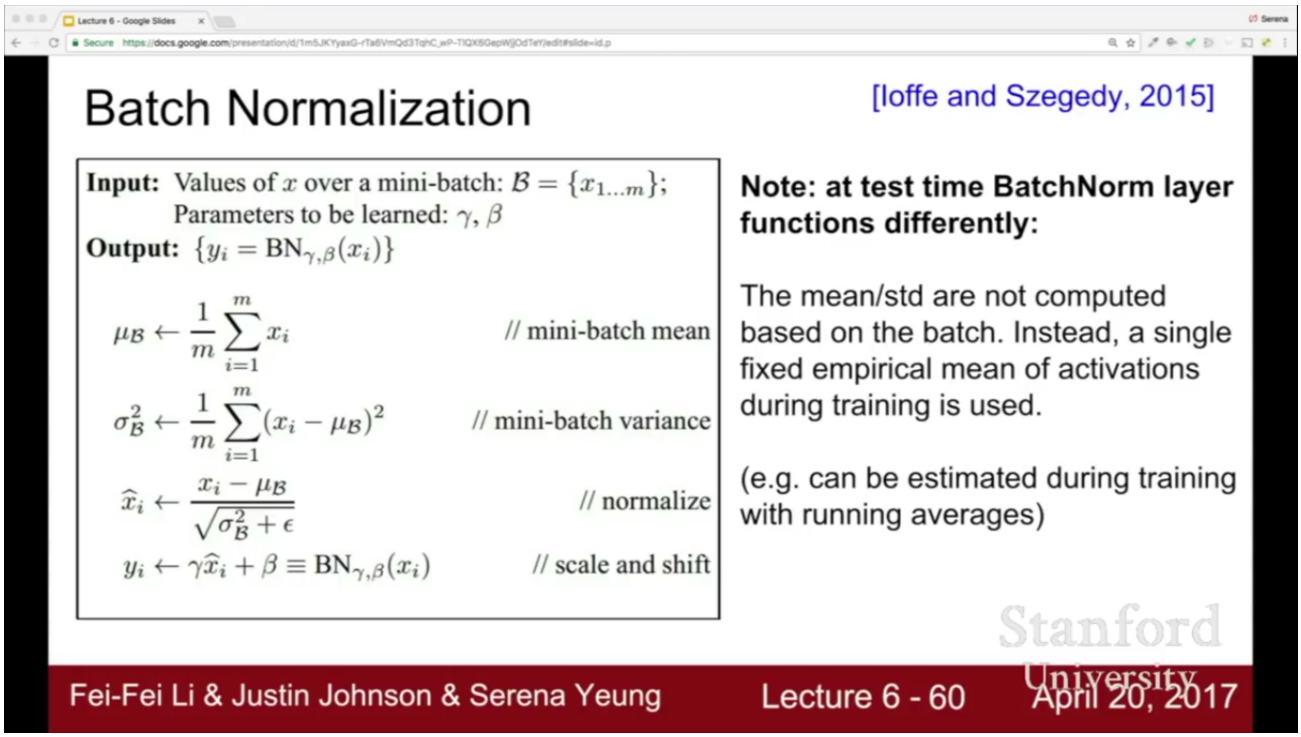
- We perform the Batch Normalization before applying any non-linearity and after Convolutional/FC layer.

- 

- 

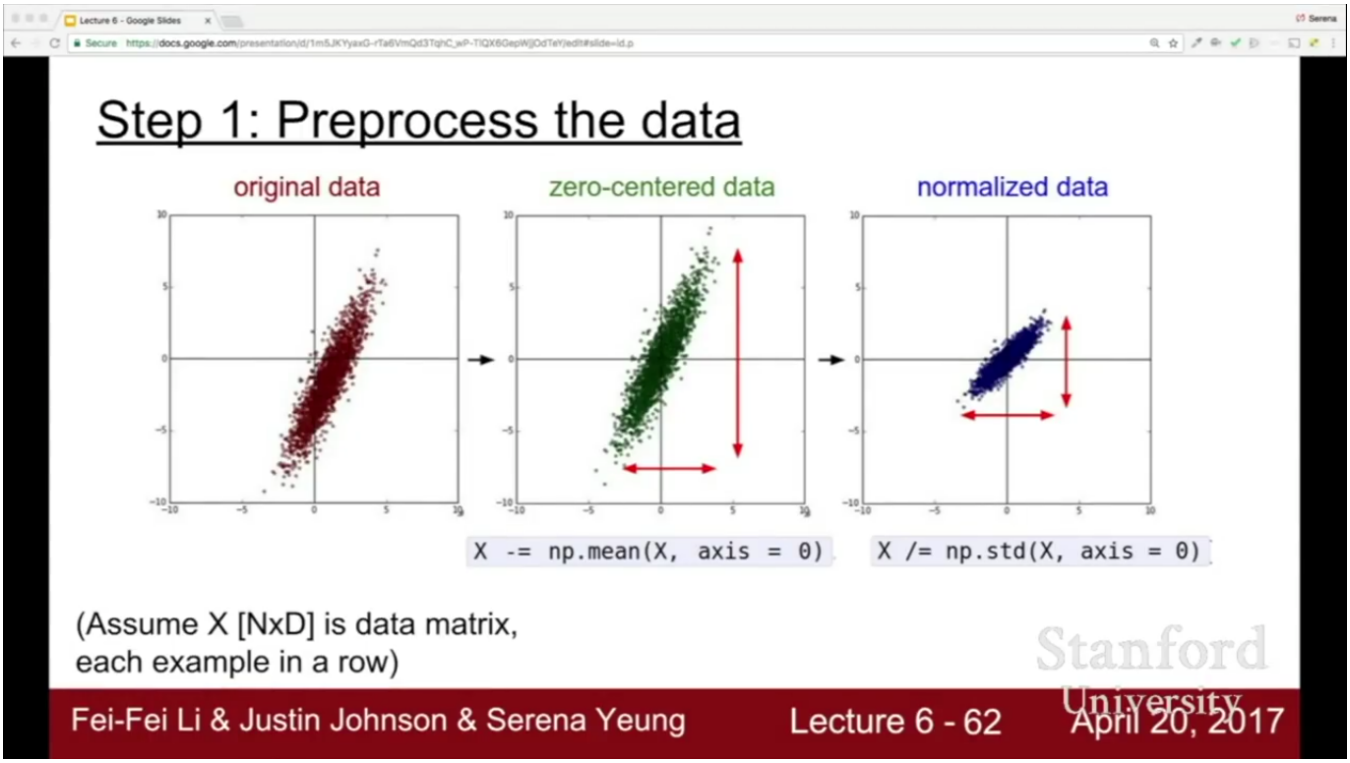
- **Here γ and β are learned parameters.**
-
- Advantages of BatchNorm:
  - Improves gradient flow through the network.
  - Allows higher learning rates.
  - Reduces the strong dependence on initialization.
  - Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe.
  - It acts like a regularizer because, each input now depends on itself and also on other samples(mean and variance calculation). So now it is not producing deterministic value for a

given training example and it is tying all of these input in a batch together.

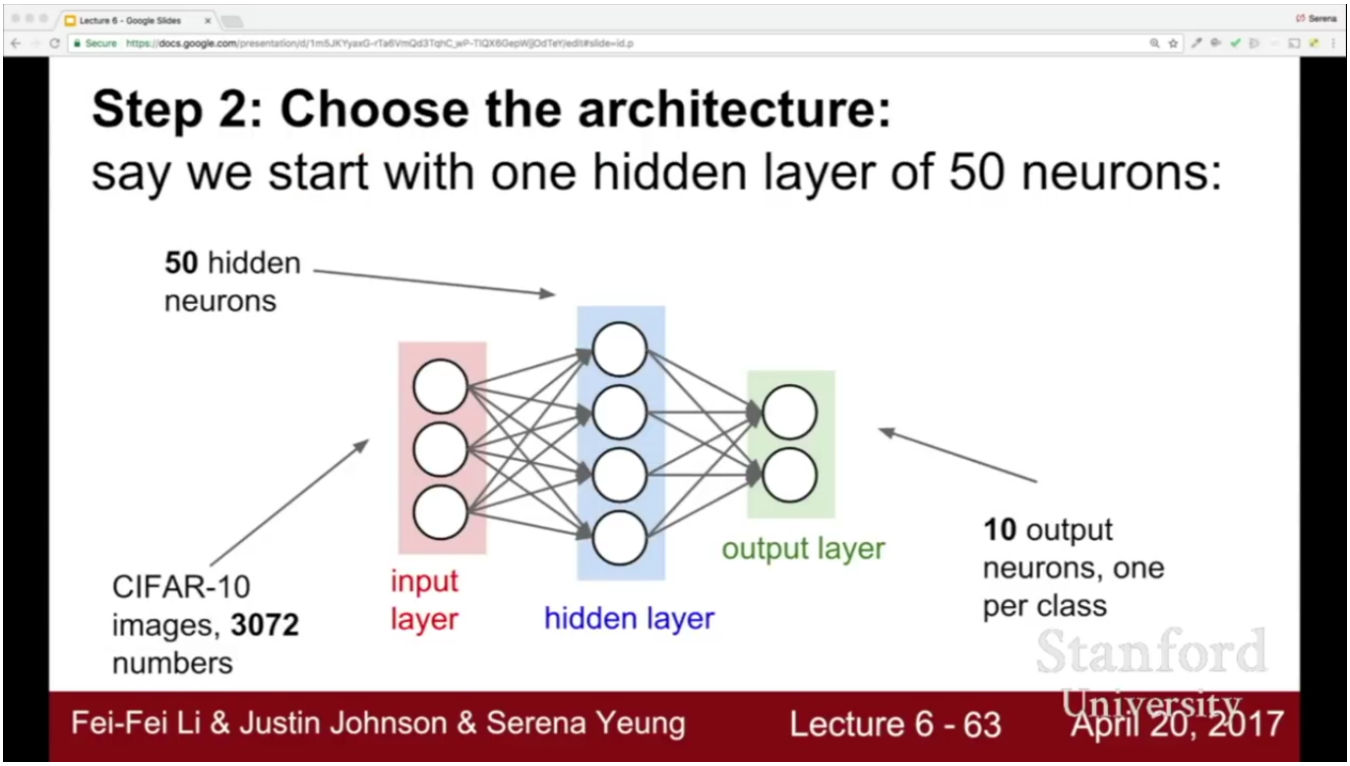
- Since it is no longer deterministic, it kind of jitters your representation of X a little bit and in some sense gives a regularization effect.
- 

Babysitting the learning process

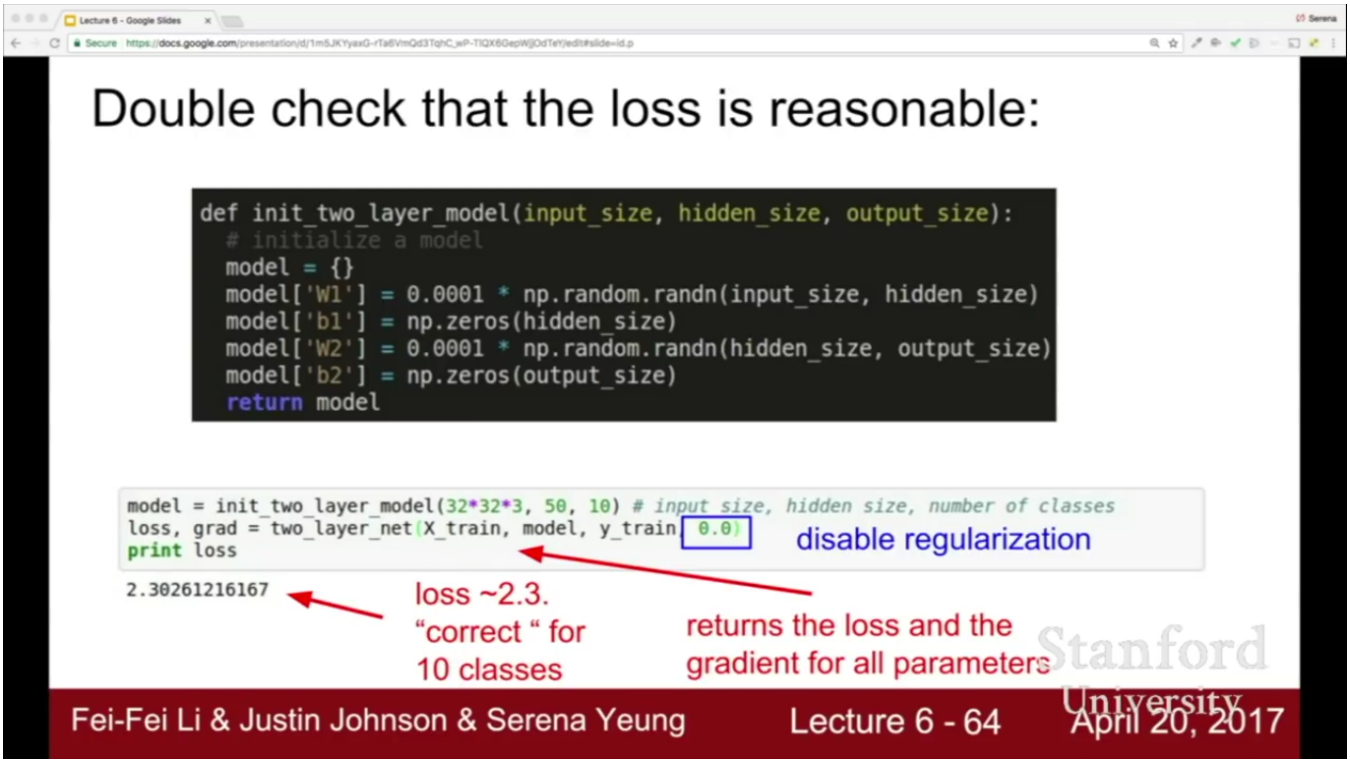
- Preprocess the data.



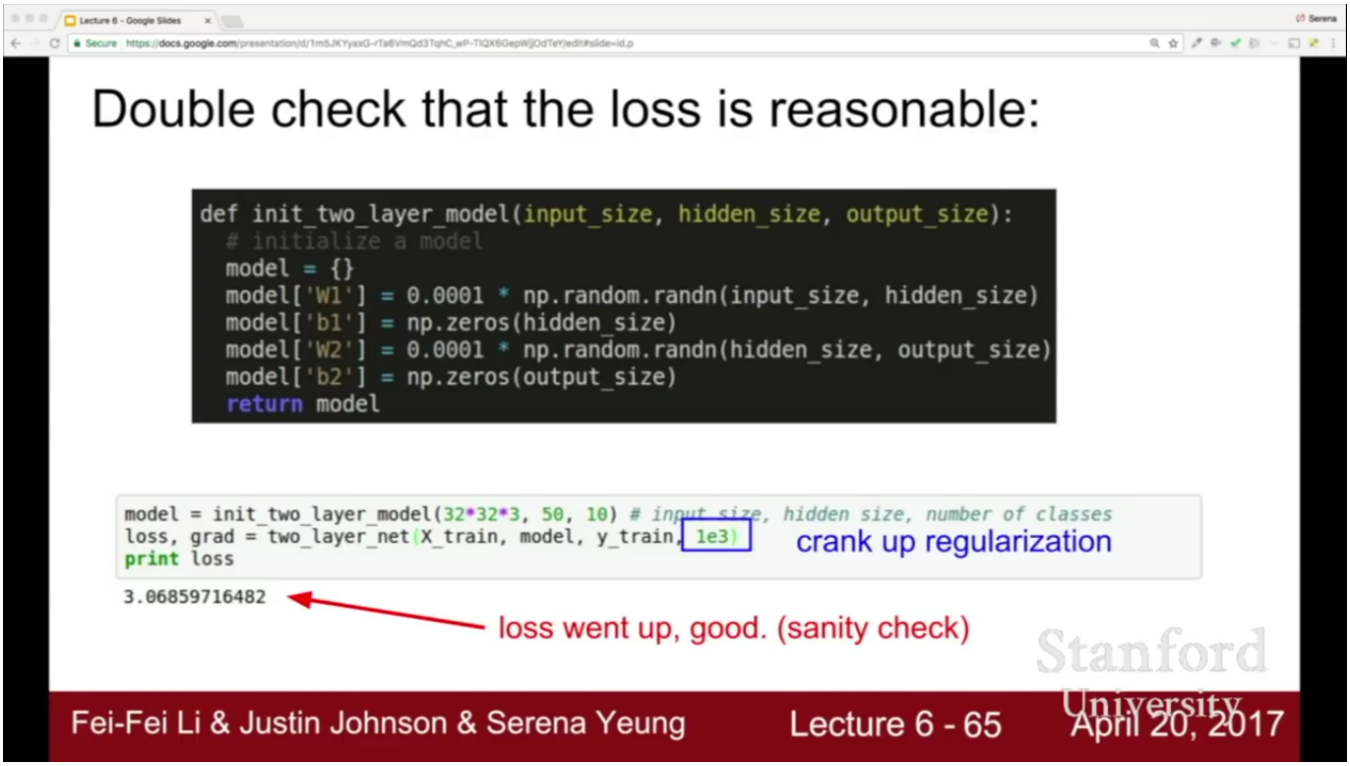
- Choose the architecture.



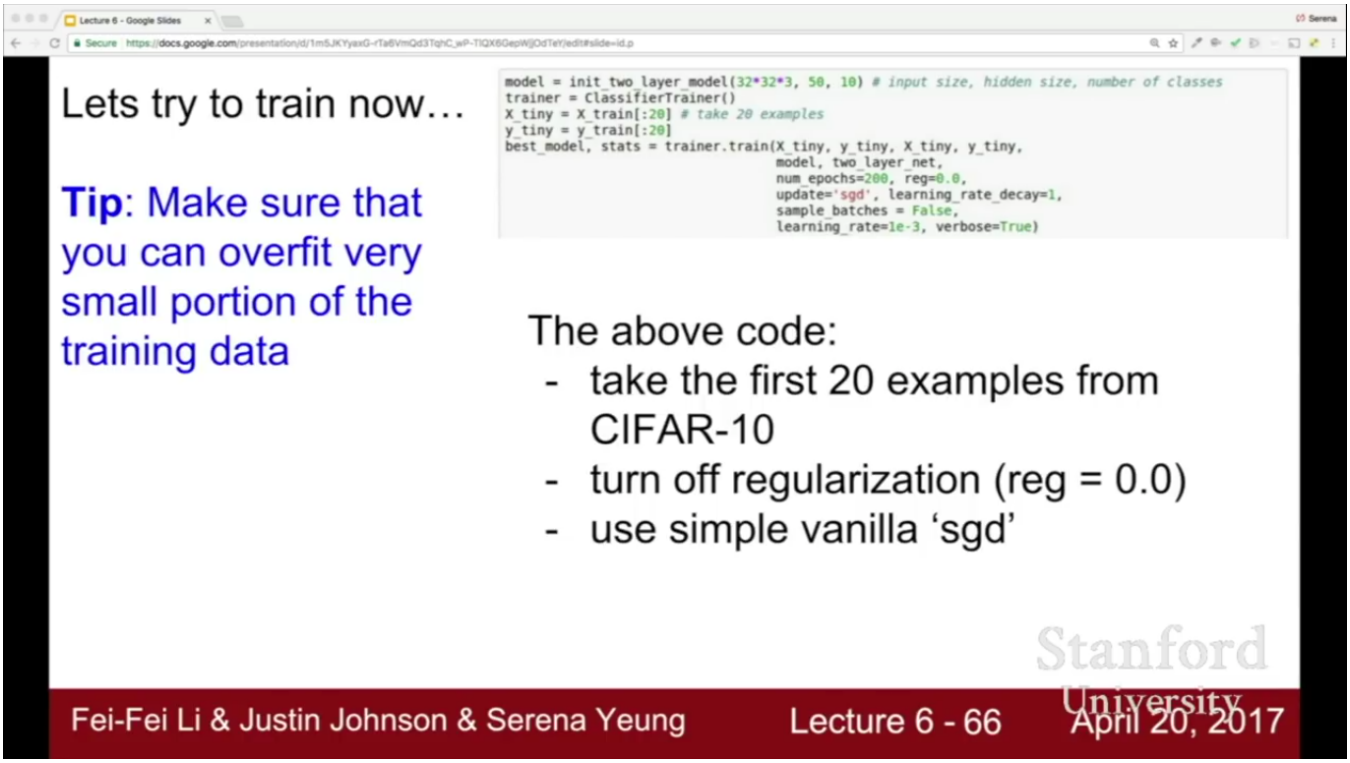
- Double check the loss is reasonable.



- Check the loss after regularization. It should increase



- Use a small dataset to make it overfit



  - If we overfit we should get a very low training loss.
- Make sure the training accuracy is very high. It should be since we need to overfit on the small amount of training data.

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

Very small loss, train accuracy 1.00, nice!

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
...
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```


Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

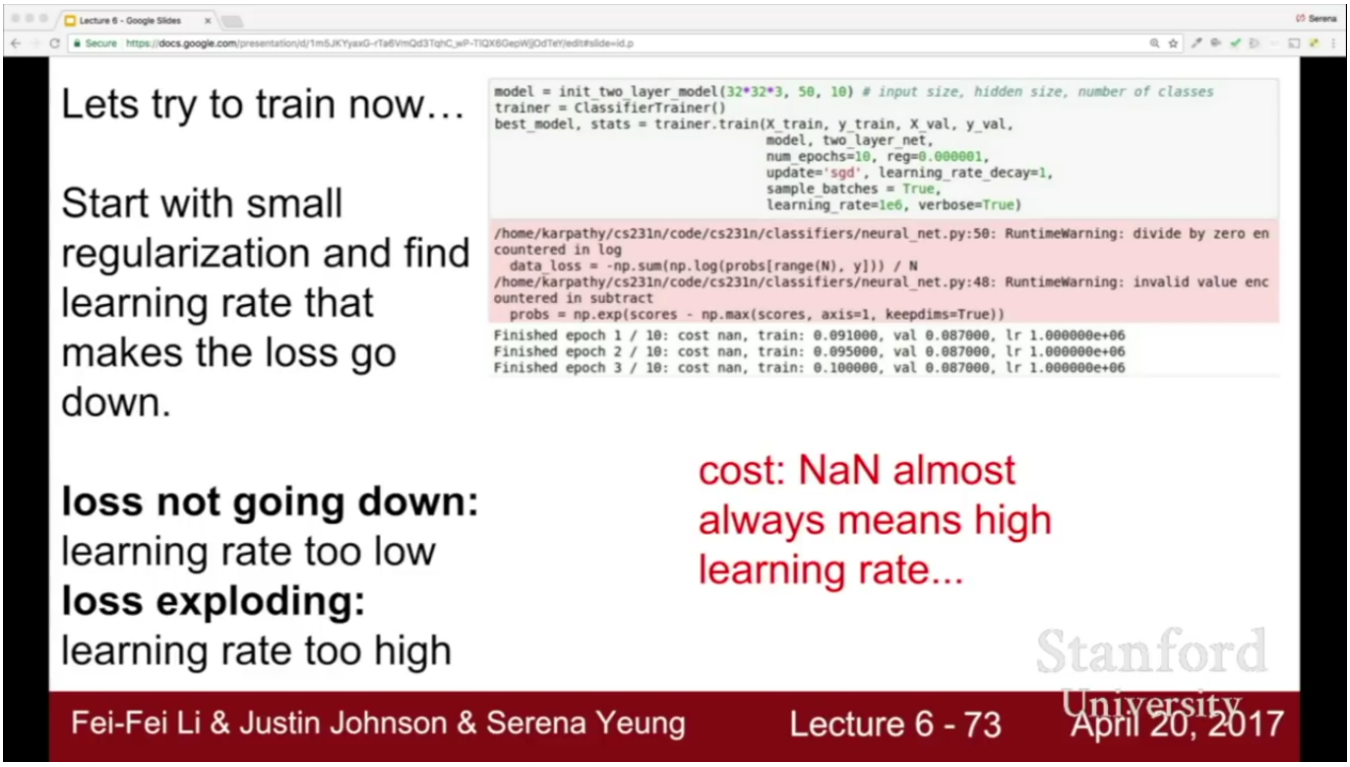
loss not going down: learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)



Hyperparameter Optimization

Cross-validation strategy

coarse -> fine cross-validation in stages

First stage: only a few epochs to get rough idea of what params work
Second stage: longer running time, finer search
… (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early

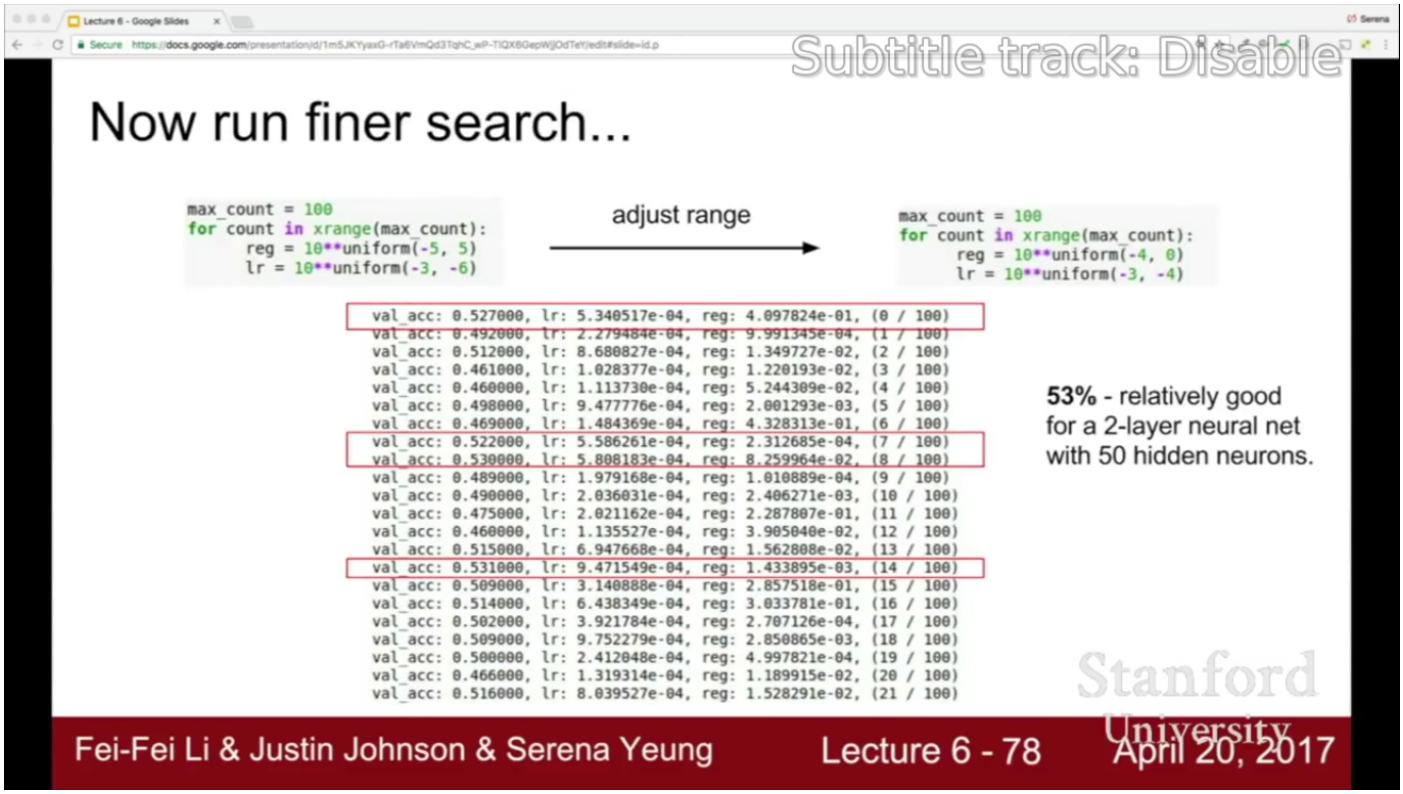
For example: run coarse search for 5 epochs

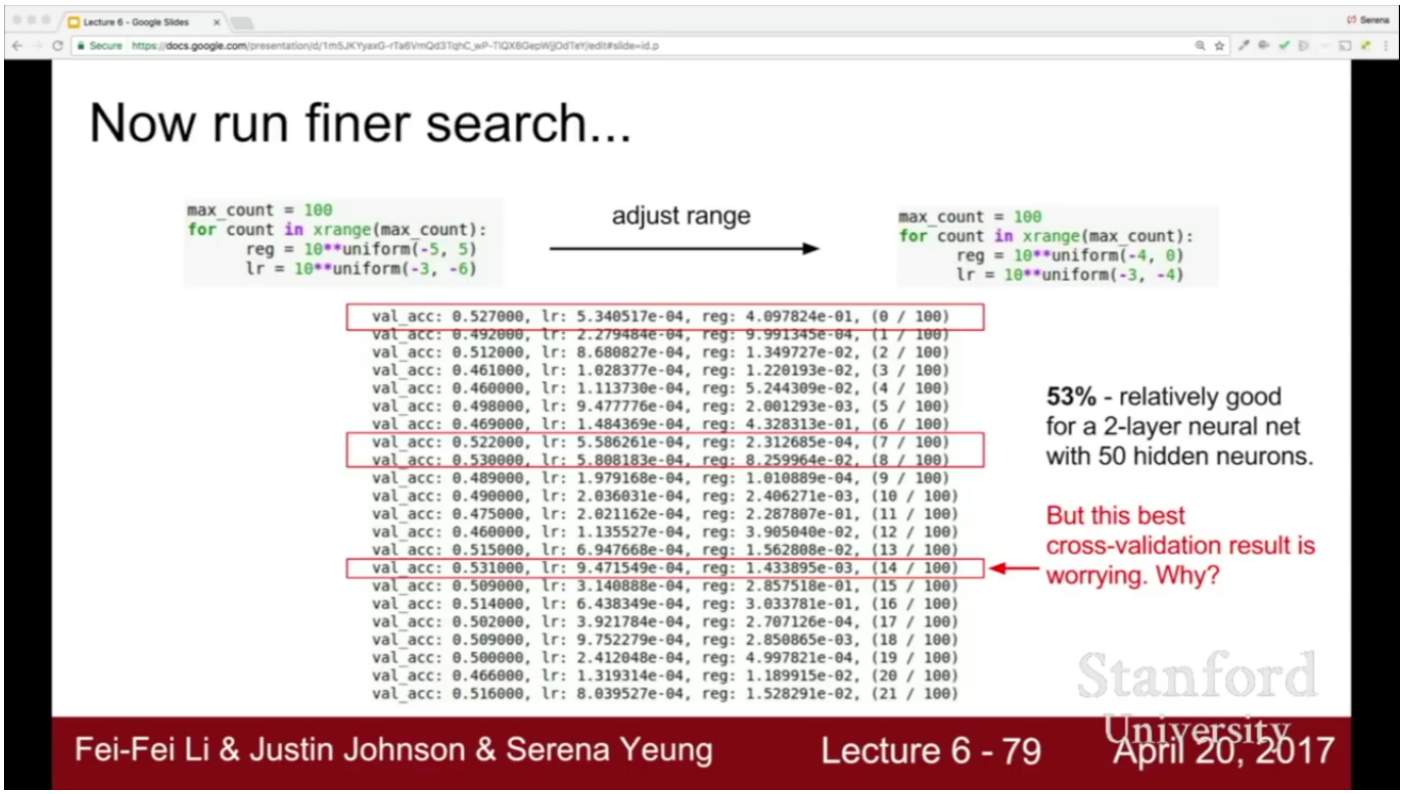
```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)

    trainer = ClassifierTrainer()
    model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
    trainer = ClassifierTrainer()
    best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                            model, two_layer_net,
                                            num_epochs=5, reg=reg,
                                            update='momentum', learning_rate_decay=0.9,
                                            sample_batches = True, batch_size = 100,
                                            learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice





- We can notice that the best result of 53.1% accuracy has a learning rate of almost 10^{-3} and we have considerably good result of 53% around 10^{-4}, this can be an issue since we might not have explored our learning rate space thoroughly, so better to keep a learning rate space from 10^{-6} to 10^{-1}.

Random Search vs. Grid Search

Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

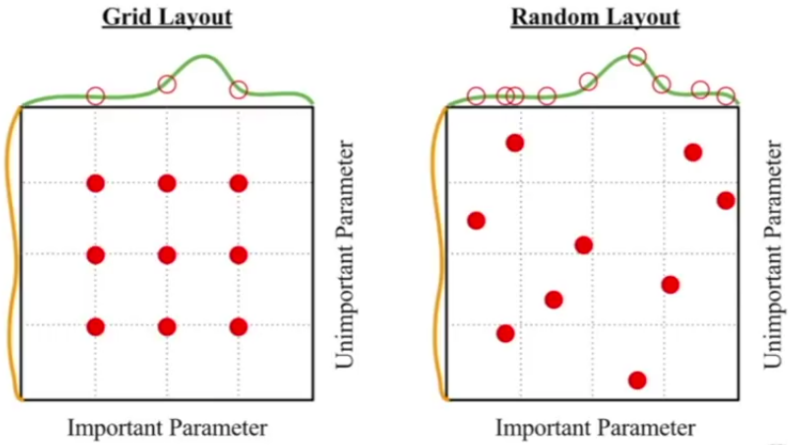


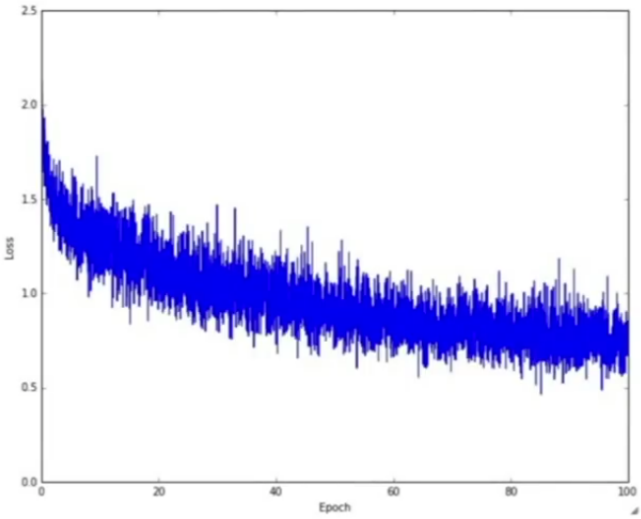
Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

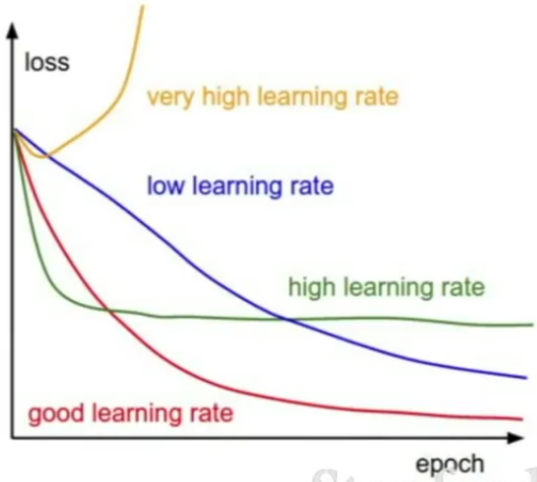
Hyperparameters to play with:

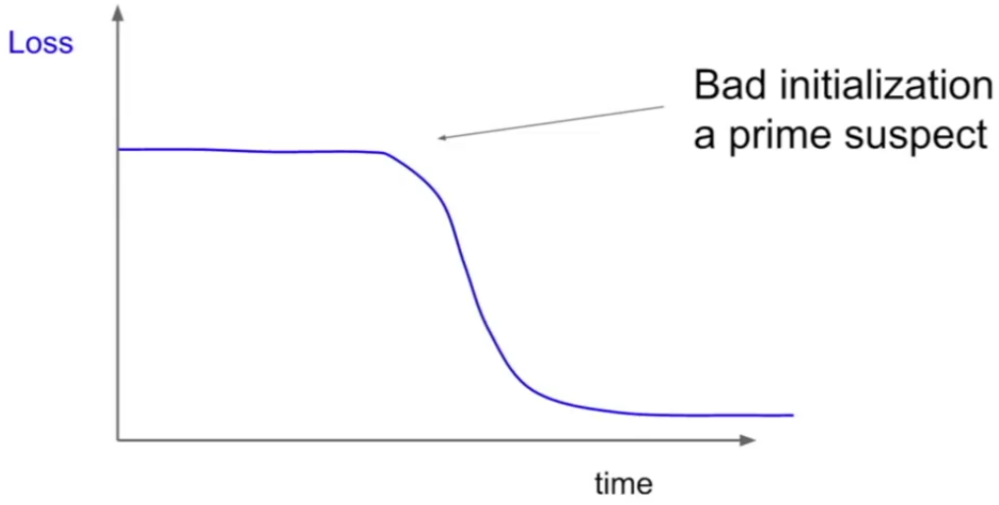
- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner
music = loss function

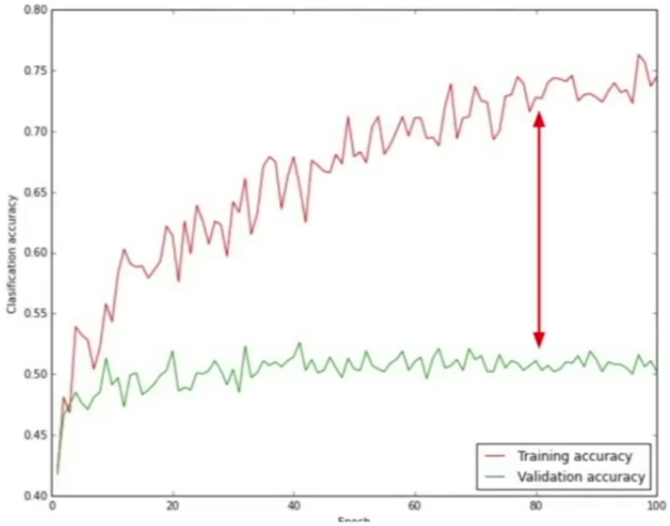
Monitor and visualize the loss curve







Monitor and visualize the accuracy:



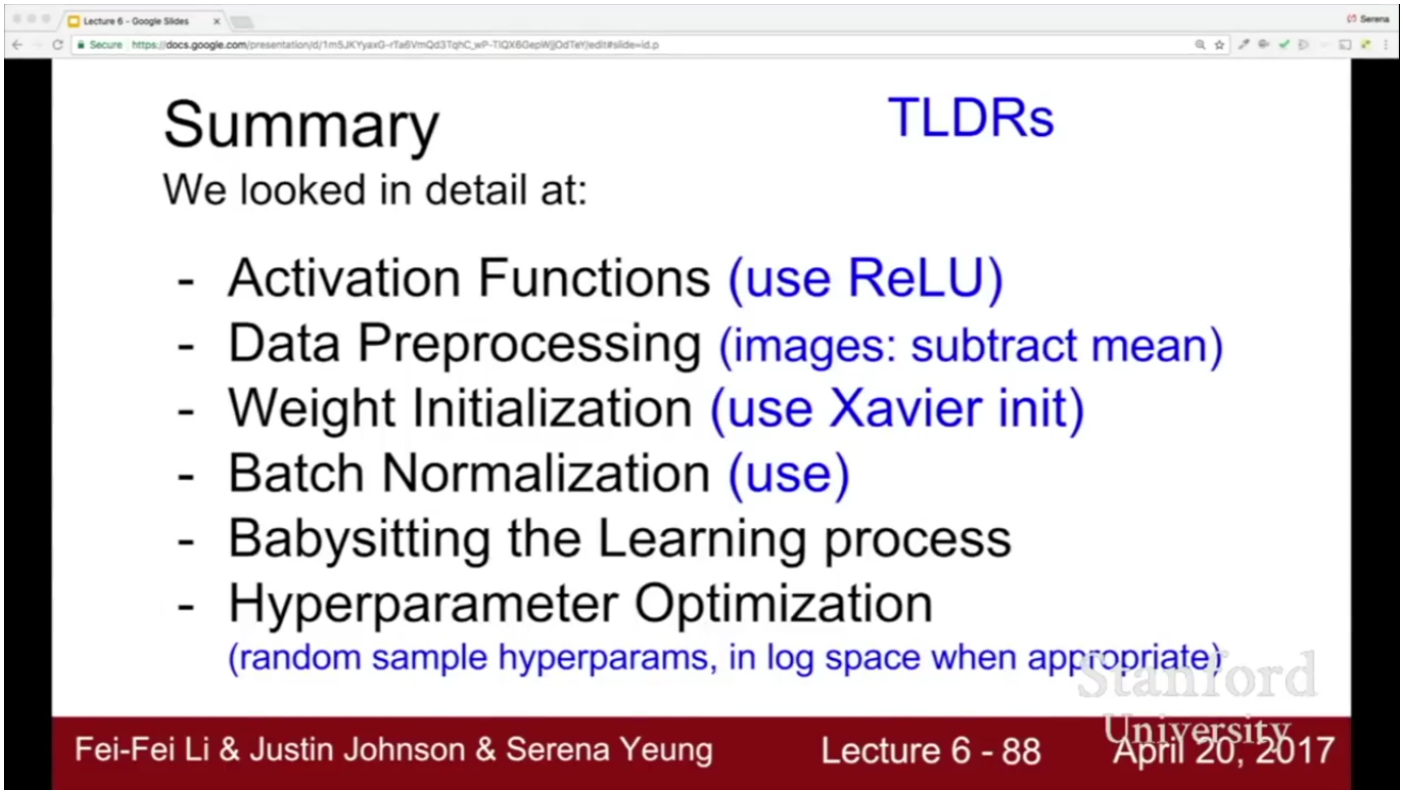
big gap = overfitting
=> increase regularization strength?

no gap
=> increase model capacity?

Track the ratio of weight updates / weight magnitudes:

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the updates and values: ~ 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.001 or so


Lecture 6 - Google Slides
Secure https://docs.google.com/presentation/d/1m5JKYyaxG-rTa6VmQd3TqhC_wP-TIQX6GepWjOdTeY/edit#slide=id.p
Summary
TLDRs
We looked in detail at:
- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization
(random sample hyperparams, in log space when appropriate)
Stanford University
Fei-Fei Li & Justin Johnson & Serena Yeung
Lecture 6 - 88
April 20, 2017