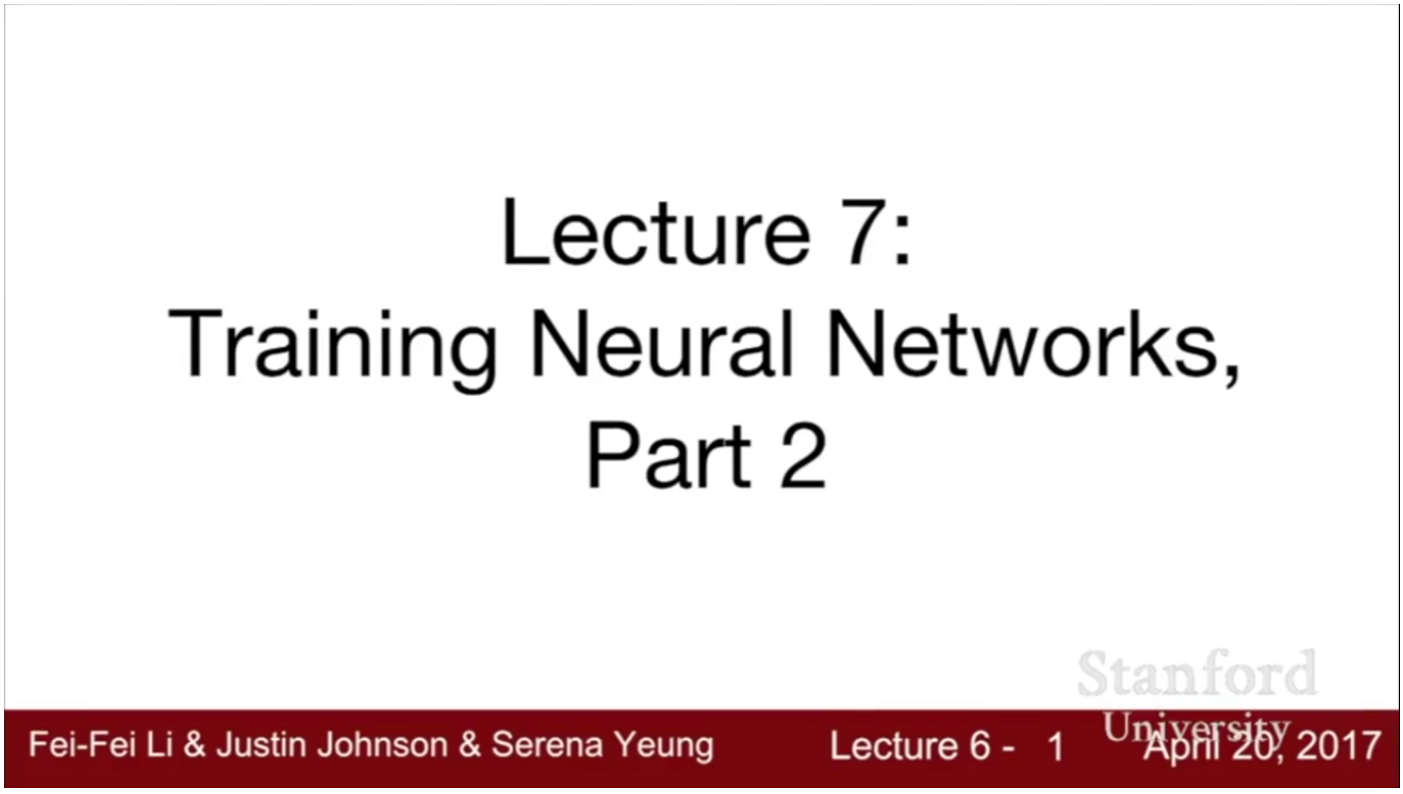
lec7

Creation Date: 10/01/2020 22:11
Last Modified Date: 11/01/2020 16:25

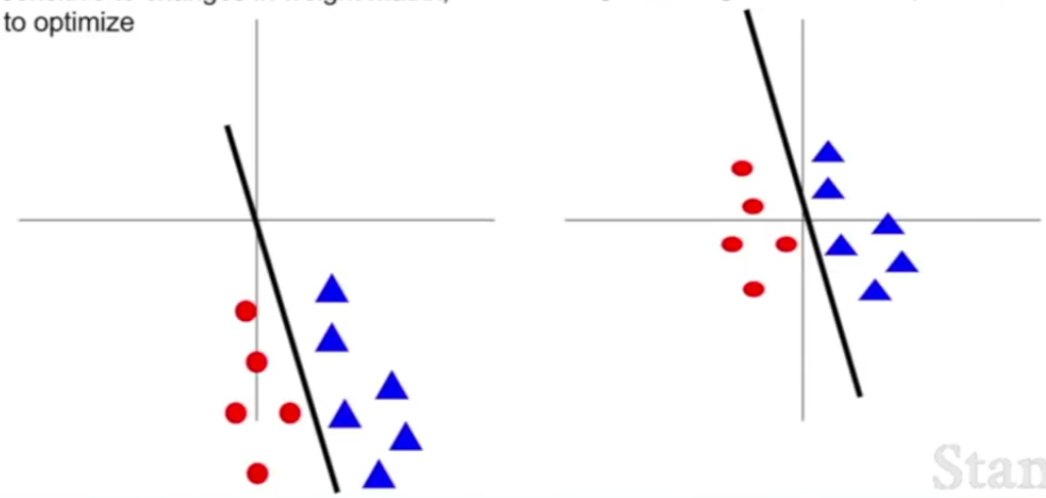
Lec 7 : Training Neural Networks - 2



Last time: Data Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize

After normalization: less sensitive to small changes in weights; easier to optimize



- Because without normalization if the discriminator varies a little bit we will have error. Because the parameter for the discriminator is provided by W, **we can say that without normalization the discriminator is highly sensitive to parameter perturbation.**
- So normalization in optimization.

Last time: Batch Normalization

Input: $x : N \times D$

Learnable params: $\gamma, \beta : D$

Intermediates: $\mu, \sigma : D$, $\hat{x} : N \times D$

Output: $y : N \times D$

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

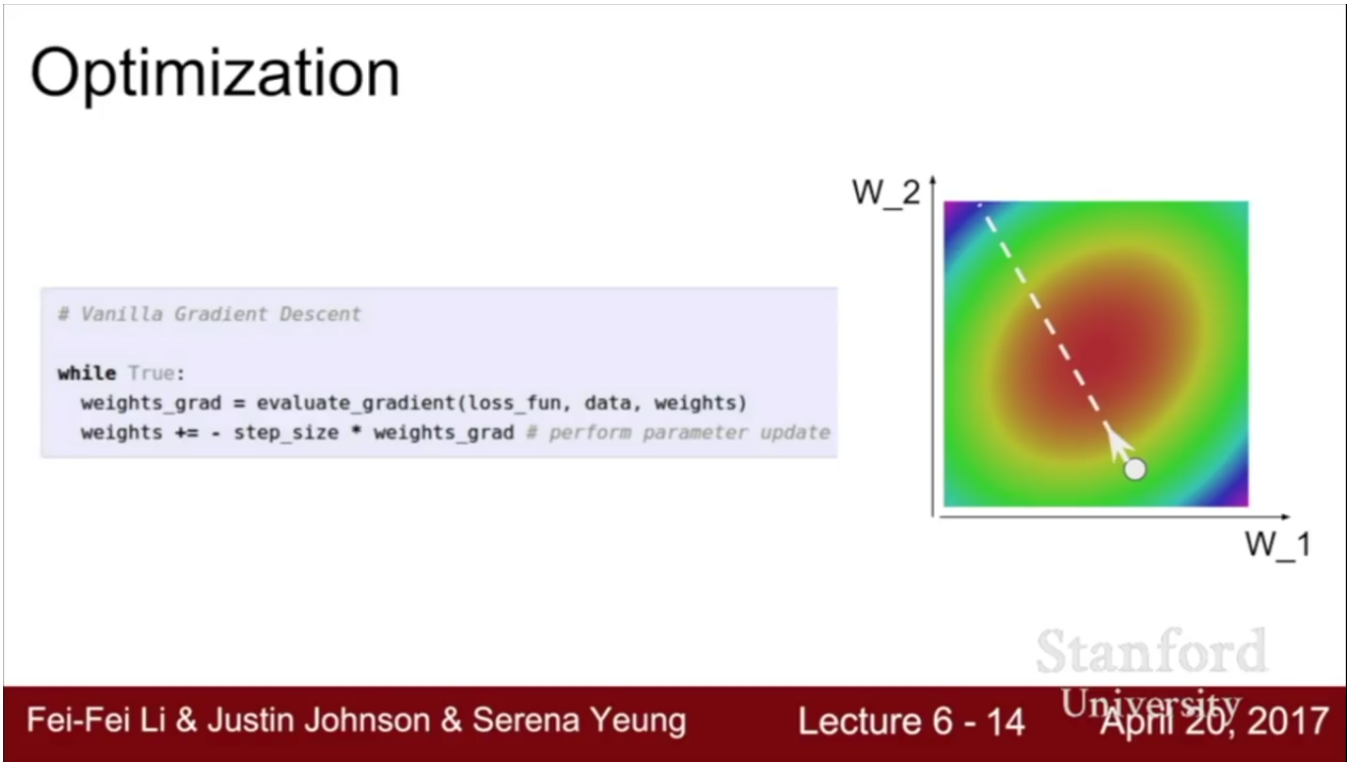
$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \sigma_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

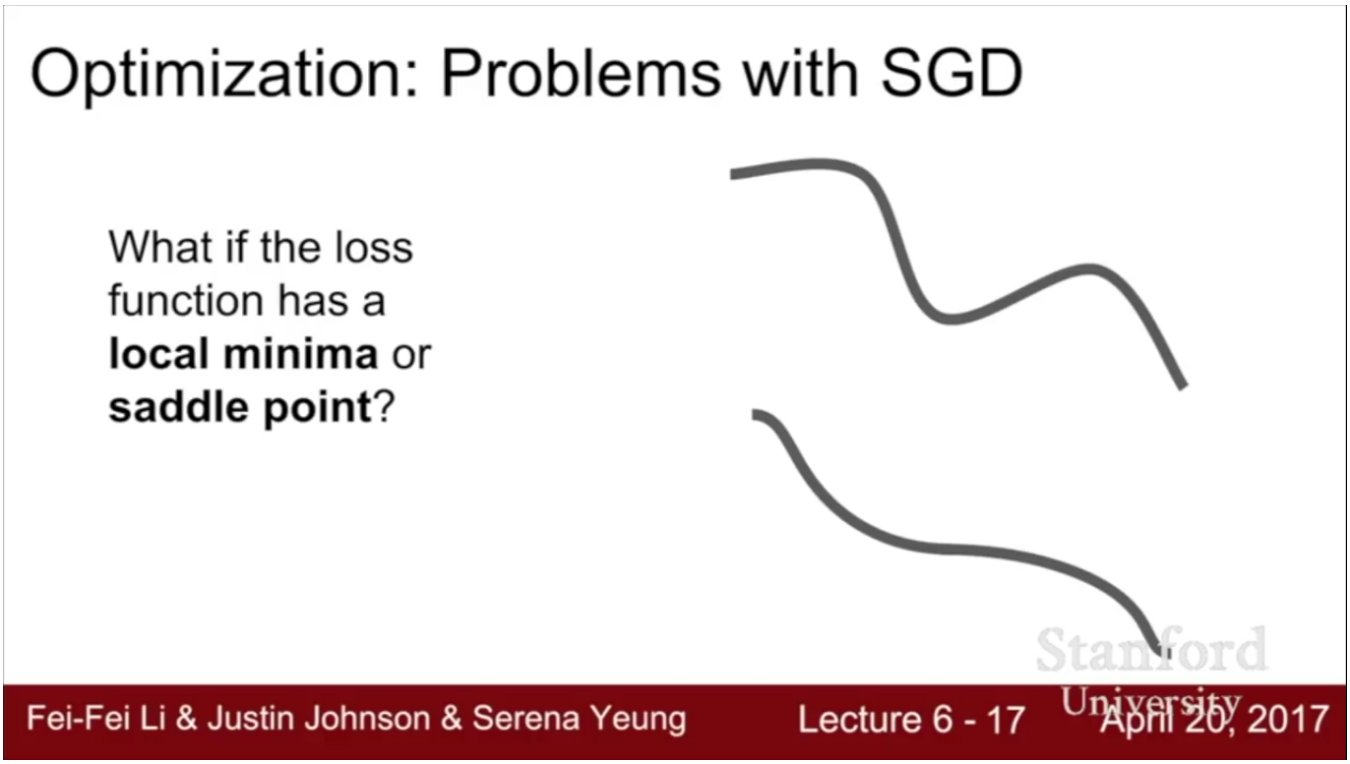
Today

- Fancier optimization
- Regularization
- Transfer Learning

- 

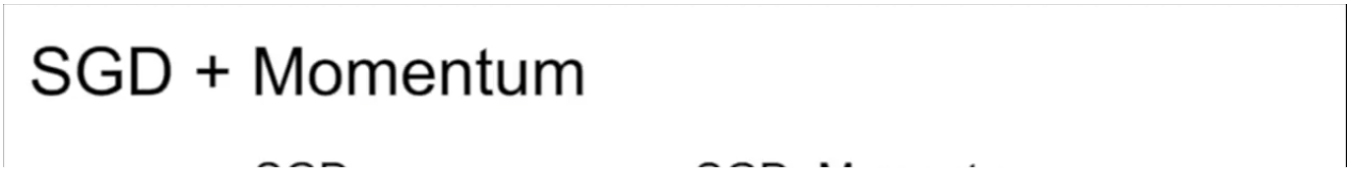
- At a given point in the above contour map, the slope is very low when we move in the horizontal direction, hence the value changes very low if we move in that direction.

- So we will have slow progress in the horizontal direction and jitter along the vertical direction.
- **So in high dimensional space the condition number can be very large**

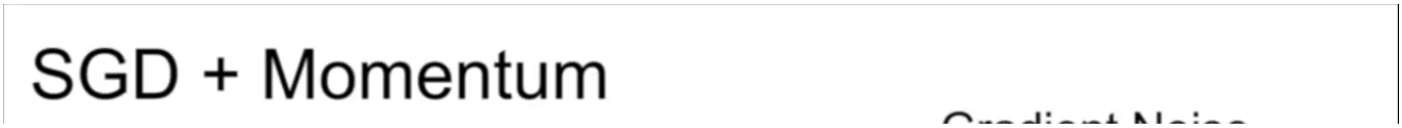


- The model will get stuck in the local minima and at saddle point. Because of zero gradient.
- **Saddle points are much more common in high dimension. Because in one direction the loss may go up and in other direction the loss may go down giving rise to saddle point.**
- **But we have less chance of having local minima because we need the loss to go up in all the neighbouring direction to get stuck in local minima.**

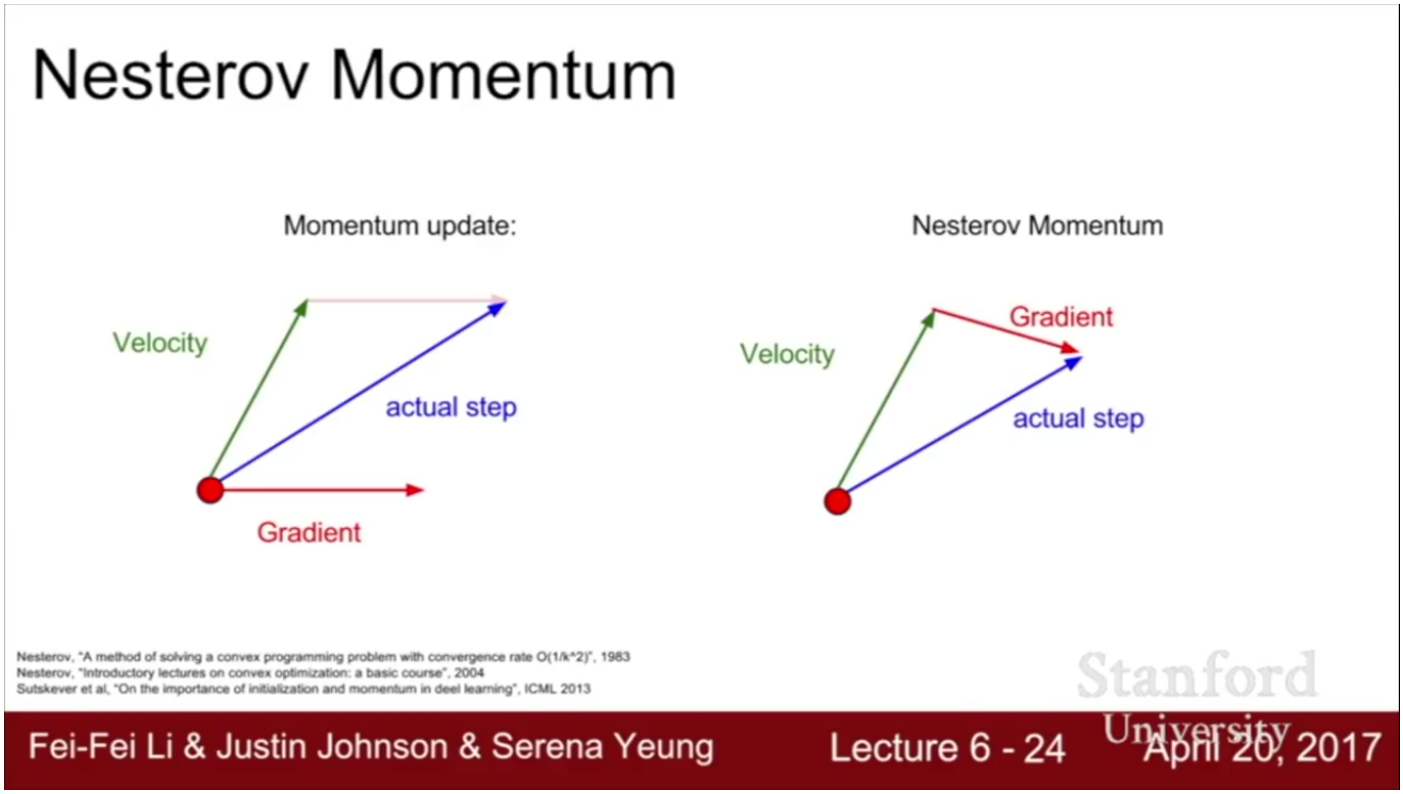
- When we use SGD, we use batch of inputs at once, hence the loss values fluctuate. Because for each batch we will have a different loss value.



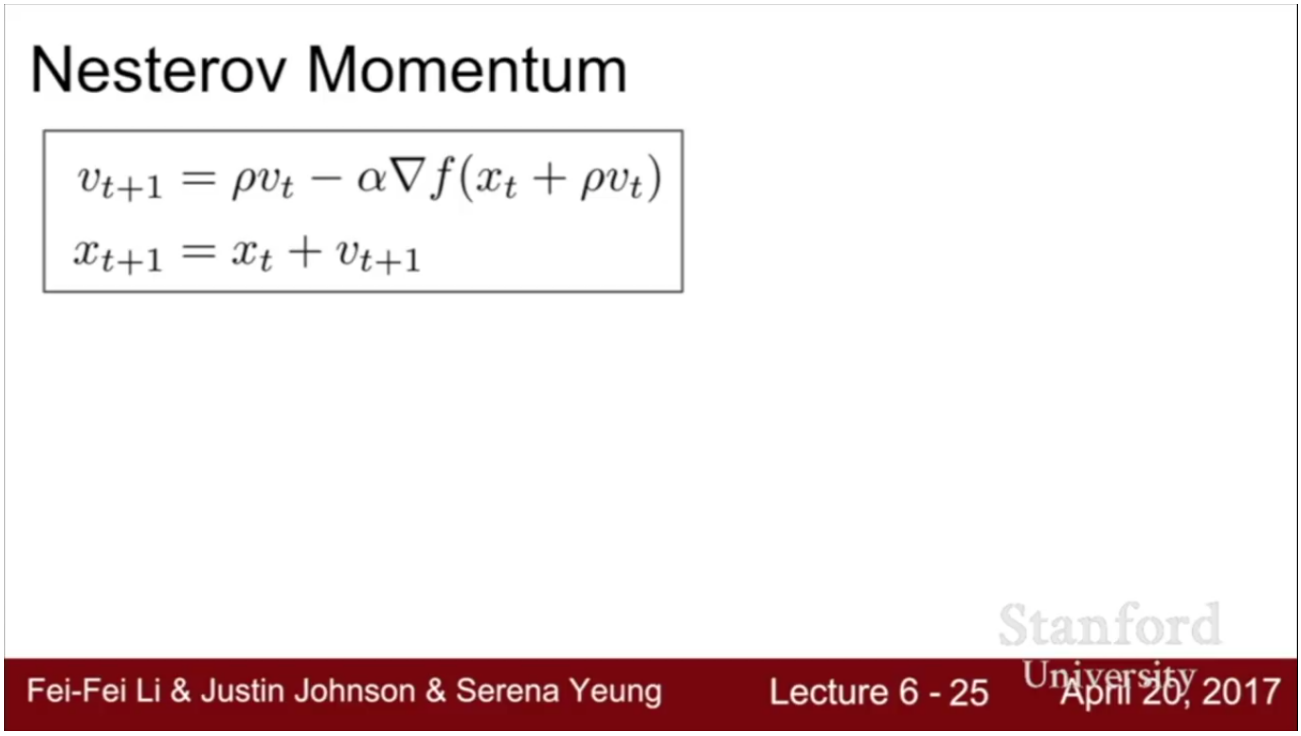
- For SGD with momentum:
  - We maintain the velocity over time and we add the gradient estimates to it and we step in the direction of the velocity **rather** than stepping in the direction of the gradient.
  - The ρ term acts like friction and we decay with that amount in each step

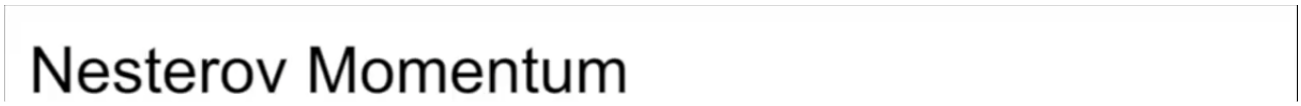


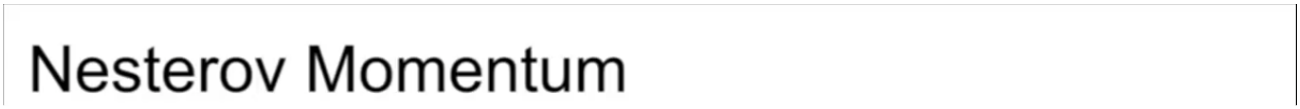
- **AT SADDLE POINT: WE REACH IT AFTER SOME ITERATION OF GRADIENT DECREASE AND THEN WHEN WE REACH THE SADDLE POINT THE GRADIENT TERM BECOMES 0, BUT SINCE WE ARE USING THE VELOCITY TERM, IT WONT BE ZERO, AND DUE TO THIS WE WONT GET STUCK AT SADDLE POINTS.**
- THIS IS WHY THE INTUITION OF A ROLLING BALL IS PROVIDED IN MOMENTUM, BECAUSE WHEN WE REACH THE BOTTOM POINT, THE GRADIENT IS 0 BUT THE BALL HAS MOMENTUM IN IT TO COME OUT OF THE SADDLE POINT.
- **DUE TO POOR CONDITIONING, OUR VANILLA SGD WILL HAVE ZIG-ZAG MOTION BUT WHEN WE USE THE ACCUMULATED VELOCITIES IN MOMENTUM THE ZIG-ZAG MOTION IS CANCELED.**



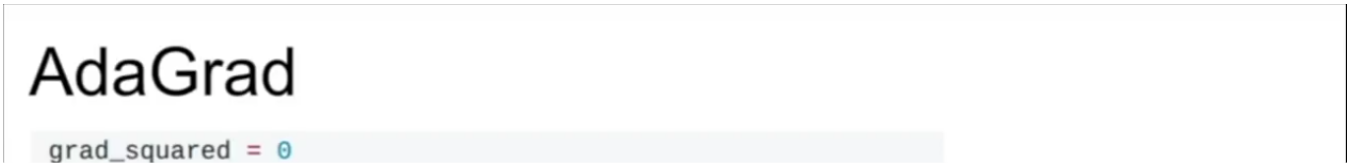
- Nesterov Accelerated Gradient is also called as Nesterov Momentum.
- In normal SGD Momentum:
  - We find gradient at the given point and then add the velocity term
- In Nesterov Momentum:
  - We start at the red point, we step in the direction of the velocity, we find the gradient at this new point. Then we go back to the original red point and then add the gradient term to go the final point.
  - We mix information, maybe if the velocity direction was actually a bit wrong, it lets you incorporate gradient information from the larger space from the objective landscape
- The velocity term is weighted velocity where the recent velocity is given high value (usually 0.9)(this 0.9 is friction term to decay the rate).
- Nesterov Momentum

- 

- We can notice that we are moving in the direction of the previous velocity v_t, and then find the gradient at that point.
- Then we move in the direction of the gradient.
- Good initialization for velocity: 0
- 

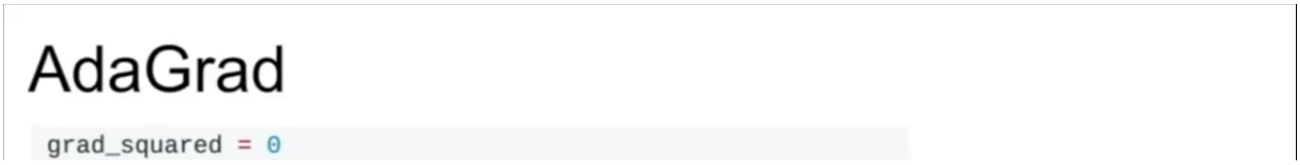


-
    - Nesterov and SGD with Momentum has this habit of overshooting but they correct it to come back to minima.
    - Due to the error correction in older velocity and current velocity Nesterov Momentum does not overshoot as much as SGD+MOMENTUM.
- AdaGrad
    -



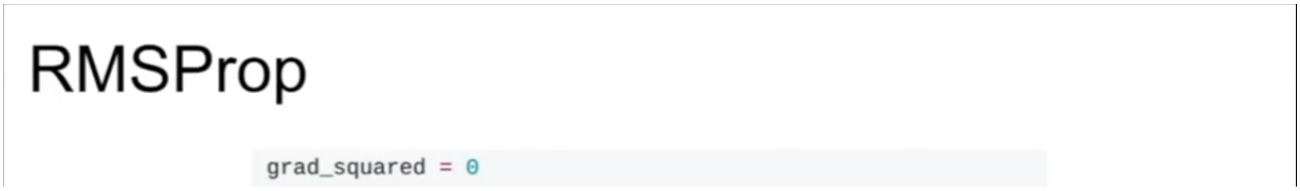
    - In AdaGrad, instead of using the velocity term, we use squared gradient term.
    - But why do we square the gradients?
    - Ans: **In the direction where the gradient is less, when we divide by square of it, we get a large value, so we move faster in the direction where the gradient is less, now in the**

direction where the gradient is large, when we divide by the squared gradient, we move slower, so we wont fluctuate more in the direction where the gradient is large.



-
- Ans: **The step size decreases as we approach the minima**
- Problem with AdaGrad: Since we do not have momentum term and the step size decreases as we reach the point of zero gradient(minima or saddle point). **We tend to get stuck there.** So AdaGrad has the problem of getting stuck at local minima and saddle point(mostly).

- RMSProp

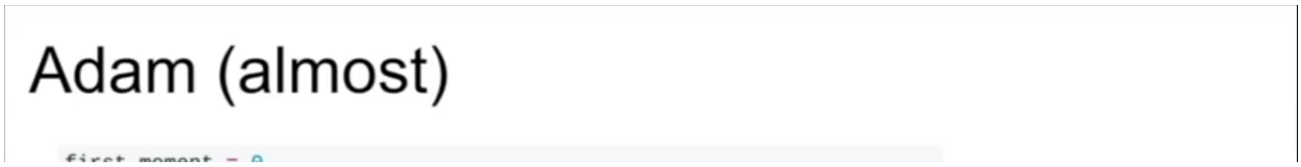


-
- Here we introduce the decaying of stored squared gradients instead of just storing the squared gradients.

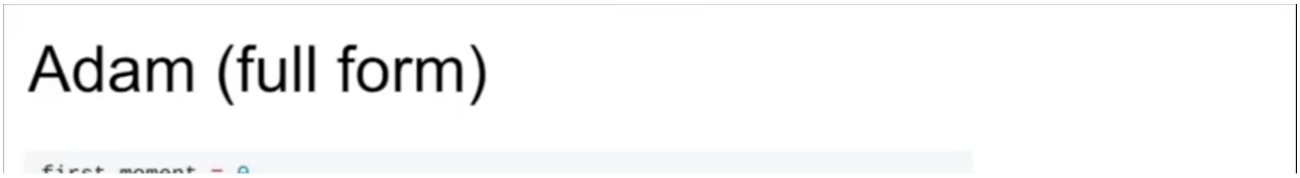
- Similar to introducing momentum with gradients but where here we are using the squared of gradients.

- SGD+MOMENTUM overshoots, but RMSProp does not.

- Adam
  - In Adam we combine the idea of both velocity term(Momentum) and squared gradients(AdaGrad, RMSProp).



  - First Moment is for finding velocity(momentum) using gradients.
  - Second Moment is using the estimate of squared gradients.
  - In the first time step, since we have initialized the first_moment and second_moment to 0, and we are even dividing by second_moment, So we will have a large step in the first timestep.



-
  - **Solution to taking large steps in the beginning: *Bias Correction***
  - We correct the bias by dividing it by using the iteration step t
  - We then use the unbiased moment terms.
  - Adam with beta1 = 0.9, beta2 = 0.999, learning_rate = 1e-3 or 5e-4 is a common starting point in various models.
- We don't have to use a constant learning rate, In most of the implementations we can find decaying learning rates

- plots like the above show that at certain time step there was learning rate decay being performed.
- Because if we do not decay the learning rate, then once we reach a region where the gradient is very small, then the learning rate being used maybe large for that region,and may keep bouncing around the region having same values. Hence we reduce the learning rate to reach lesser and lesser gradients.

-
- Learning rate decay is more common with SGD+Momentum than with Adam.
- **The idea is to first try to optimize with no learning rate decay, then once we have our primary hyperparamters like learning rate etc set, then move on to secondary hyperparameters like learning rate decay etc to get even better models.**

First-Order Optimization

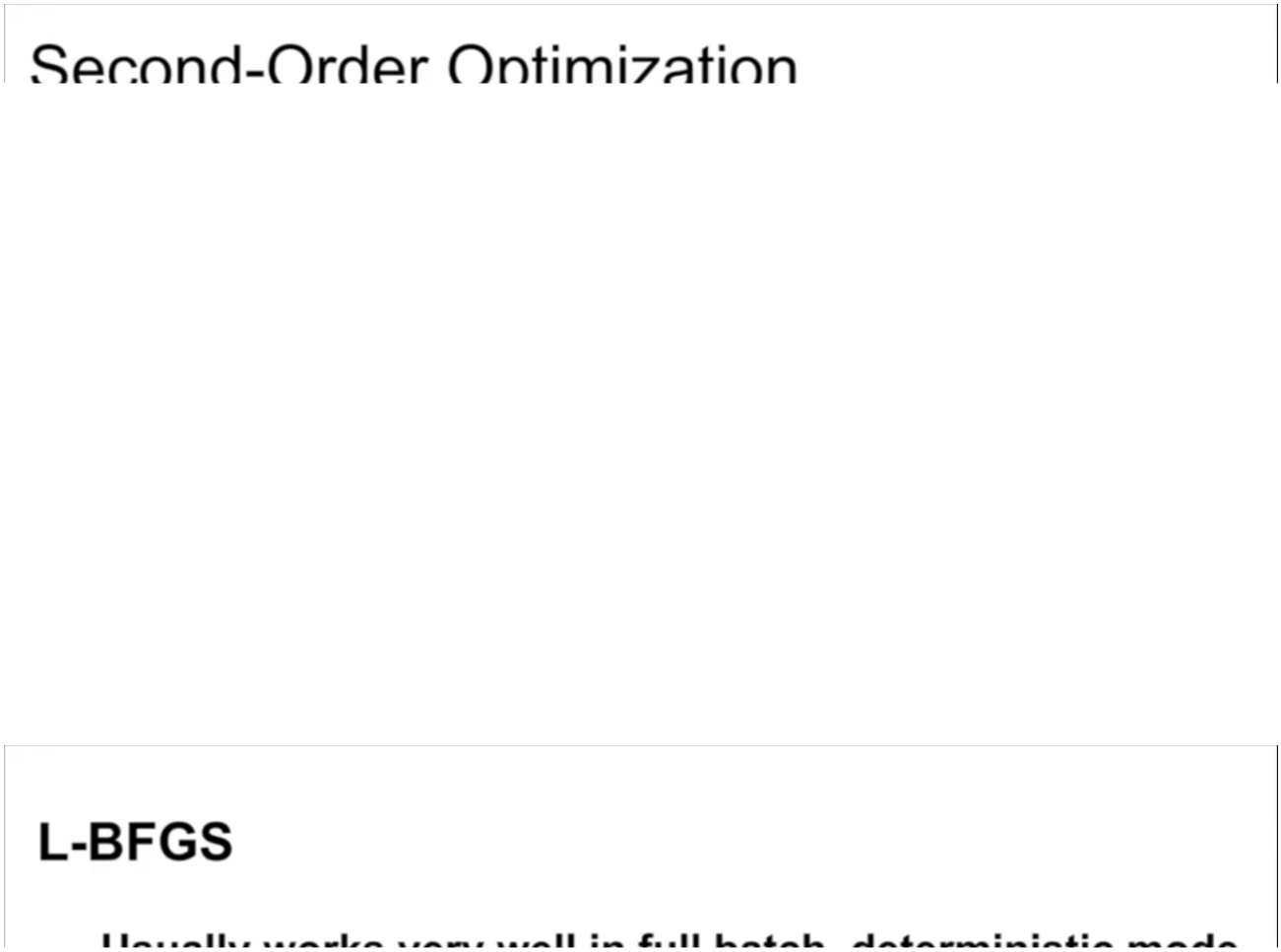
- First order approximation does not hold for very large regions and we cannot step too far in that direction.

Second-Order Optimization

- Here we incorporate both the first order and second order gradients to approximate the function.

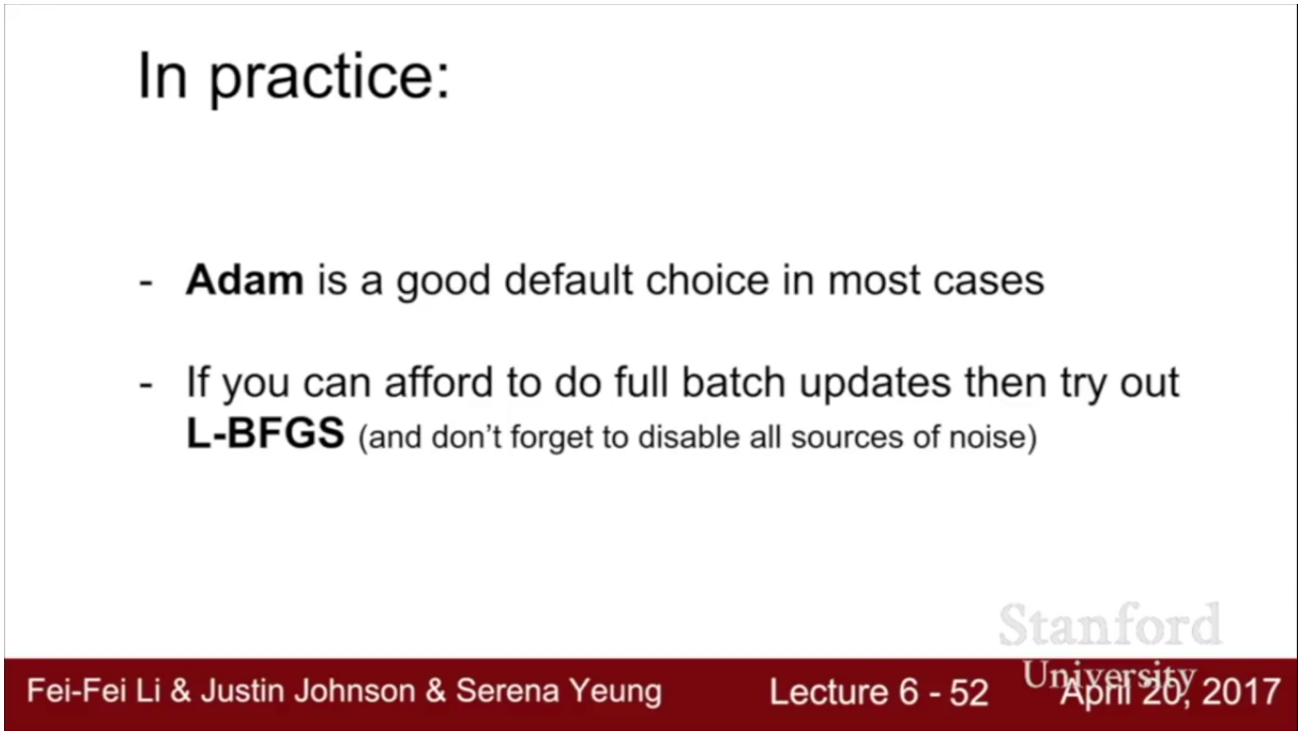
- When we generalize this approximation to multiple dimensions we call it **'Newton step'**.
- What is nice about this update?
- Ans: When we invert the Hessian matrix H, we directly step to the minimum of the quadratic approximation of the function.
- **NOTE: It does not have learning rate term.**
- But usually we tend to use the learning rate. Because the quadratic approximation may not be perfect, So we want to step in the direction fo the minimum than to the actual minimum.

-
- **We don't prefer Second Order Approximation because of the time complexity of computing the Hessian and also the inverse of the Hessian.**

- Solution:
  - Use Quasi-Newton, L-BFGS methods
    - 

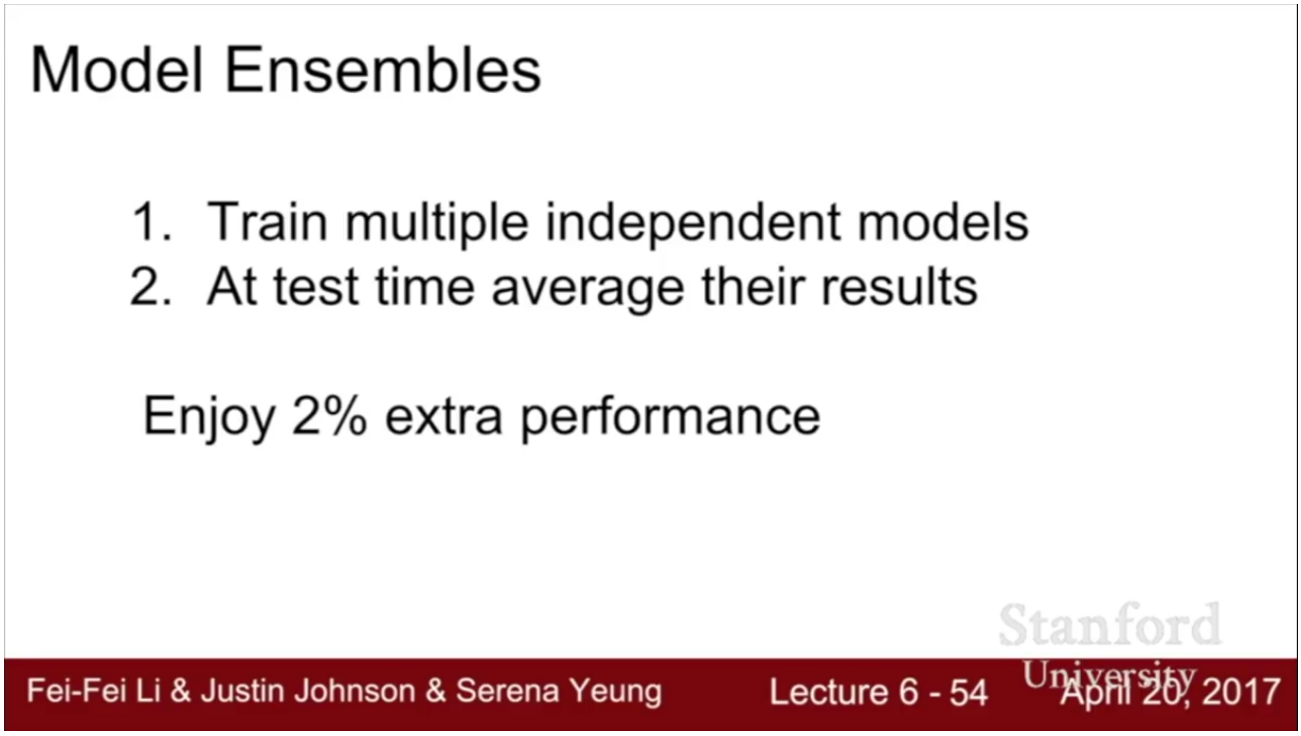
    -

    -

- 

Beyond Training Error

- We care about the performance on unseen data more than the performance on training data.
- Approach 1:



- Since we have multiple model with random initialization for each, we have more variance and less overfitting to the training data.

- Instead of training multiple models we can have multiple snapshot of a single model, where we have learning schedules such that it increases, decreases, increases and so on.
- The idea behind such approach of using a crazy learning schedule is that the model will have snapshots where it would have reached multiple local minimas and the ensemble of these saved snapshots will give better results and we wont have to use multiple models.

- Our goal is to increase the performance on validation set, rather than reduce the gap between training and validation set.
- They hyperparameters across the ensembles will be different, we can use different depth of the networks, different learning rate schedules for the models and then ensemble them.

Model Ensembles: Tips and Tricks

How to improve single-model performance?

- We can use regularization if we want to use a single model to avoid overfitting the training data too well so that the model performs better on unseen data.

Regularization: Add term to loss

- L2 Regularization is also called as weight decay.
- **L2 regularization does not make much sense in Neural Networks so only we have Dropout.**

Regularization: Dropout

- In Dropout we set the **'activation'** to **zero**
- **Dropout is commonly used in FC layers, but it can be used for Convolutional layers also**
- During Dropout in Convolutional, we drop the feature maps randomly. That is we will drop channels from channel dimensions rather than dropping random elements from the layers.

Regularization: Dropout

Example forward

Regularization: Dropout

- Uses of Dropot:
  - Forces the network to have a redundant representation
  - Prevent co-adaptation of features.
  - Prevent overfitting by avoiding a neuron to depend strictly on previous neurons.

Regularization: Dropout

- Dropout is like creating a very large ensemble.

Dropout: Test time

- Here z is the dropout mask, which is an added term during test time.

Dropout: Test time

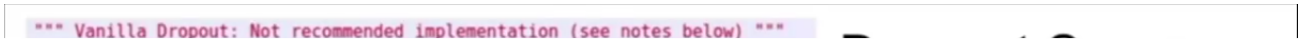
Want to approximate

Dropout: Test time

Want to approximate

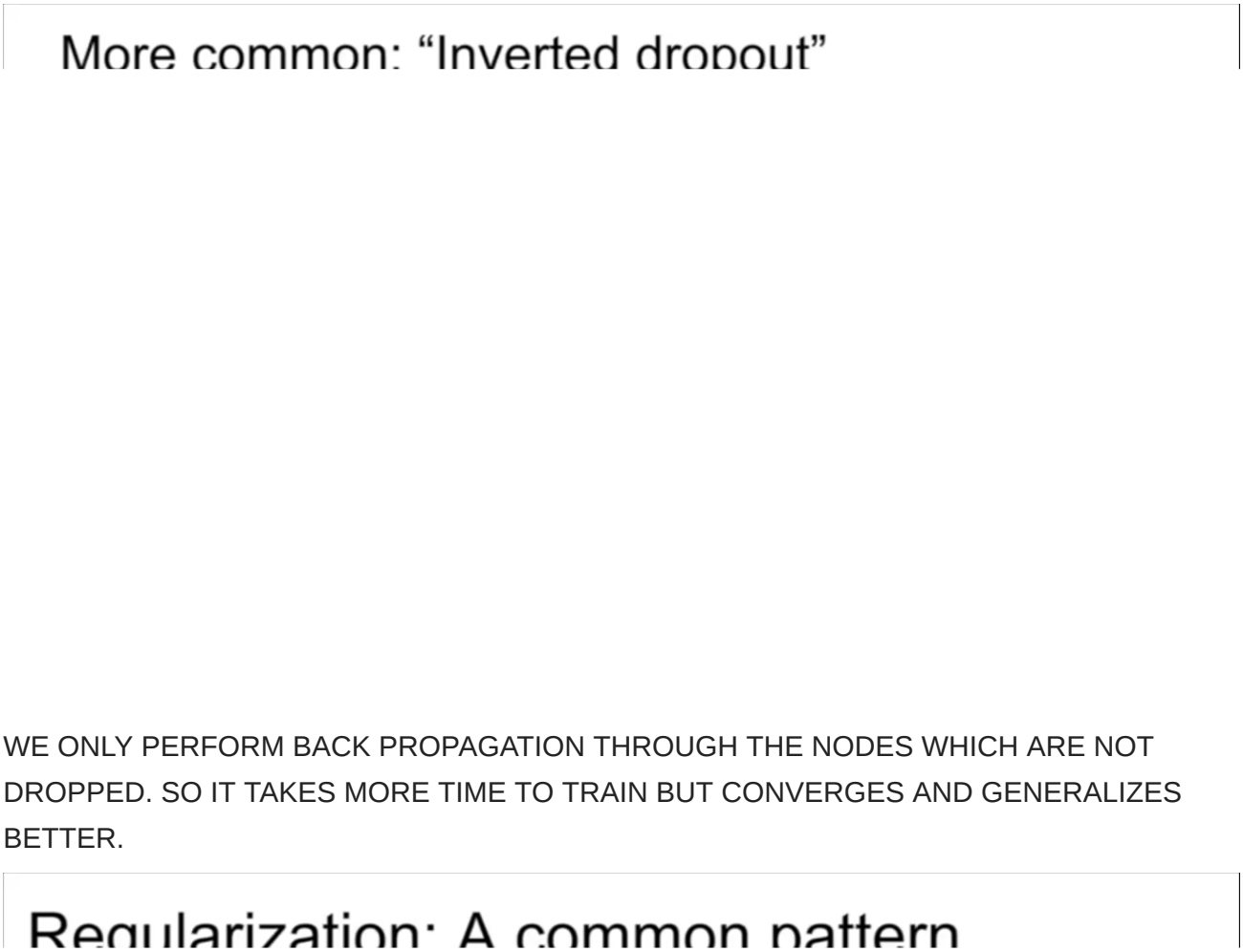
NOTE:

- Summary of DROPOUT:



  -

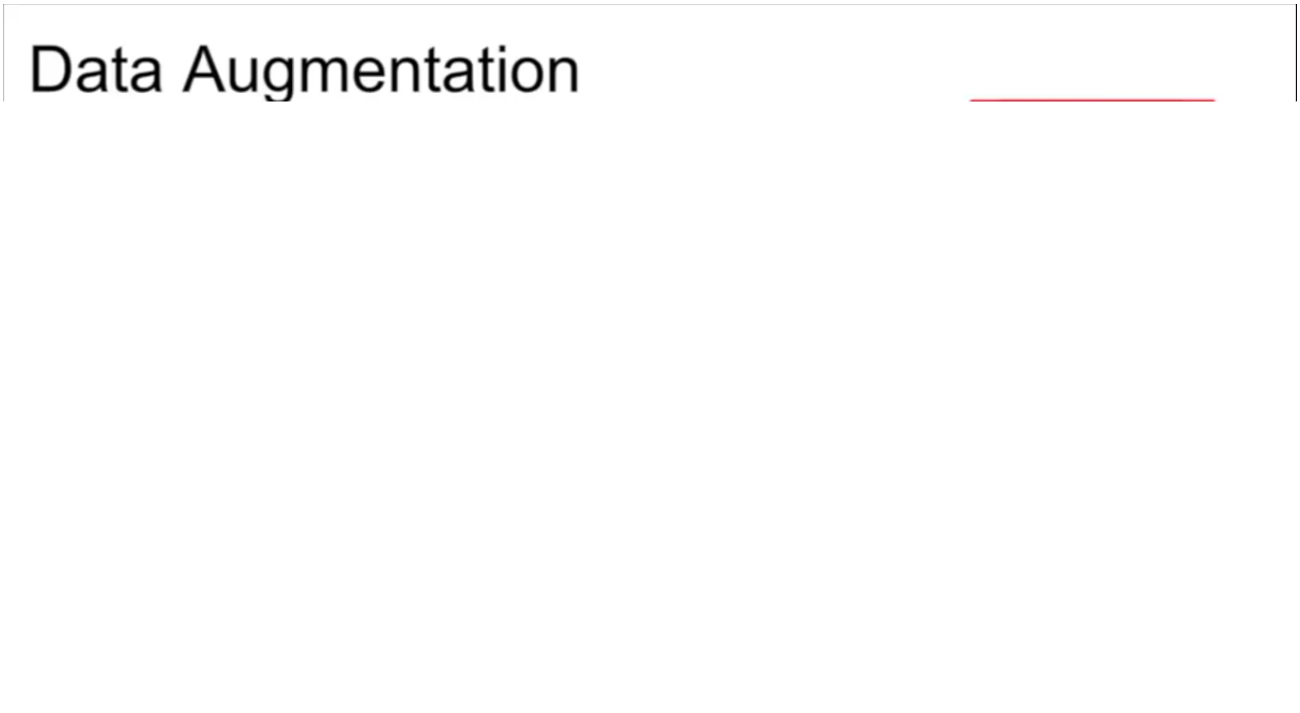
- Inverted Dropout is more common in implementation:



- WE ONLY PERFORM BACK PROPAGATION THROUGH THE NODES WHICH ARE NOT DROPPED. SO IT TAKES MORE TIME TO TRAIN BUT CONVERGES AND GENERALIZES BETTER.

-
- BATCH NORM ALSO HAS THE EFFECT OF REGULARIZATION BECAUSE WE INTRODUCE STOCHASTICITY, SINCE EACH SAMPLE IN THE BATCH WILL ALSO BE A SAMPLE IN OTHER BATCHES WHICH ACTS LIKE INTRODUCING RANDOMNESS.
- SO SOMETIMES IF WE ARE USING BATCHNORM WE DONT USE DROPOUT.
- WE PREFER DROPOUT BECAUSE WE CAN TUNE THE DROPOUT PARAMETER WHEREAS THERE IS NO SUCH CONTROL IN BATCH NORMALIZATION.

Regularization: Data Augmentation

- 

-

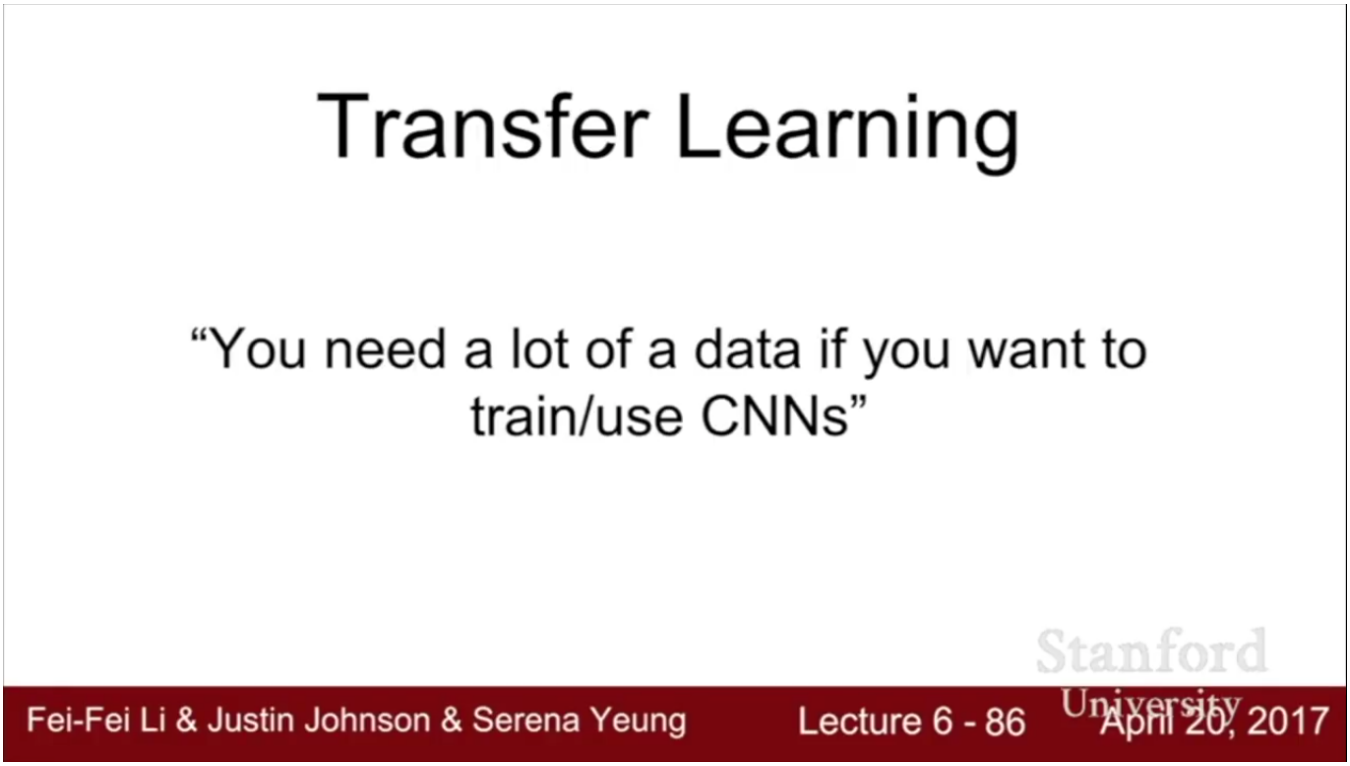
- 
Data Augmentation
More Complex:
- 
Subtitle track: Disable

- In DropConnect we zero out the weight entries and not the activations of the nodes during the forward pass.

- **Fractional Pooling**: During training we have different sizes of pooling regions and during testing we can either choose a set of pre-determined pooling sizes or drawing many samples using random pooling sizes then average over them.

- Stoachastic Depth: We drop layers during training but during testing we use the whole network without dropping any layers.
- Can we use multiple regularization ideas in a single model:
  - Using BatchNorm is prefered since it performs consistently for deeper networks.
  - We can choose to add dropout if using batchnorm is not enough and the model is still having overfitting issue.
  - We dont just add different regularization techniques blindly, we add them if the other ones are not working or improving.

- Transfer Learning:



- If we use a large model and if our dataset is small, we will be overfitting to that small dataset.
- One solution is to use Transfer Learning.

-
- We use smaller learning rate because our pretrained model has optimized weights and do not need a large learning rate to converge.

- In Deep Learning we mostly use pretrained models.

- **Takeaway for your projects and beyond:**

-