

# Machine Learning

## CNN Training 2

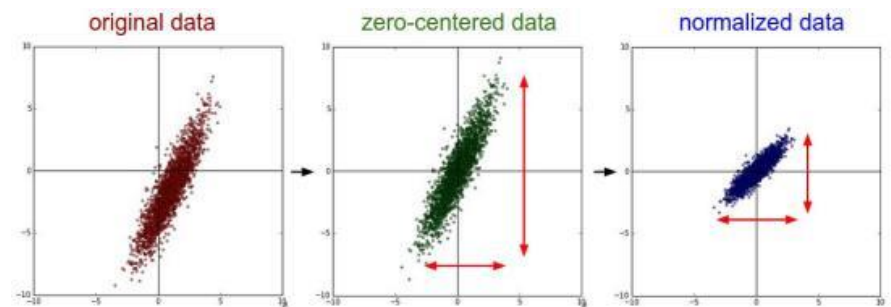
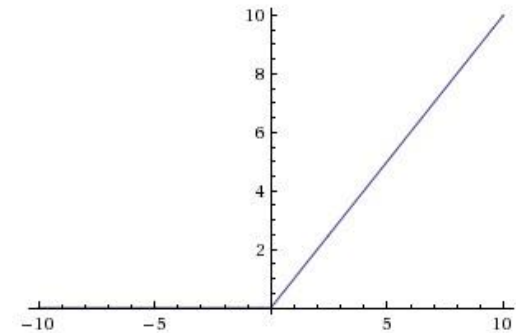
Indian Institute of Information Technology  
Sri City, Chittoor



# Previous Class

## Training Aspects of CNN

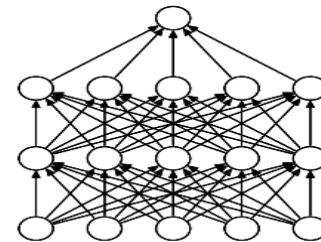
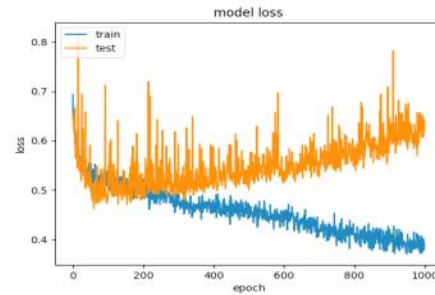
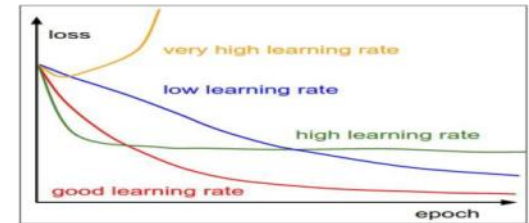
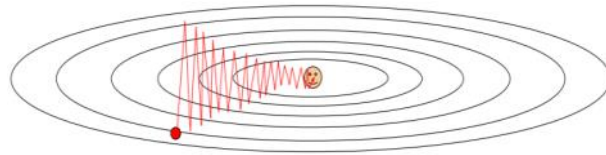
- Activation Functions
- Dataset Preparation
- Data Preprocessing
- Weight Initialization



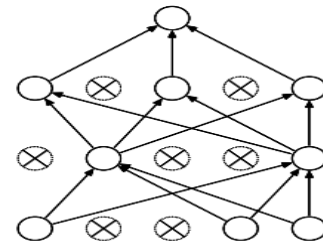
# This Class

## Training Aspects of CNN

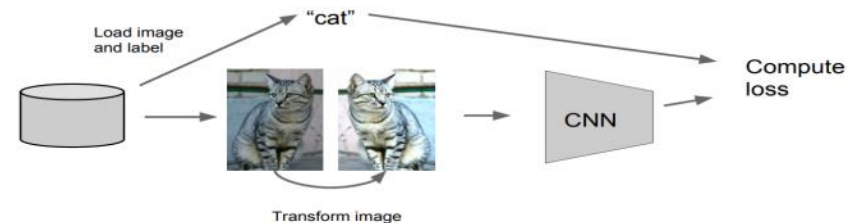
- Optimization
- Learning Rate
- Regularization
- Dropout
- Batch Normalization
- Data Augmentation
- Transfer Learning
- Interpreting Loss Curve



(a) Standard Neural Net



(b) After applying dropout.



# Optimization





# Mini-batch SGD

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

# Stochastic Gradient Descent (SGD)

The procedure of repeatedly evaluating the **gradient of loss function** and then performing a **parameter update**.

*Vanilla (Original) Gradient Descent:*

```
while True:
    dx = compute_gradient(x)
    x += learning_rate * dx
```

# SGD: Problems

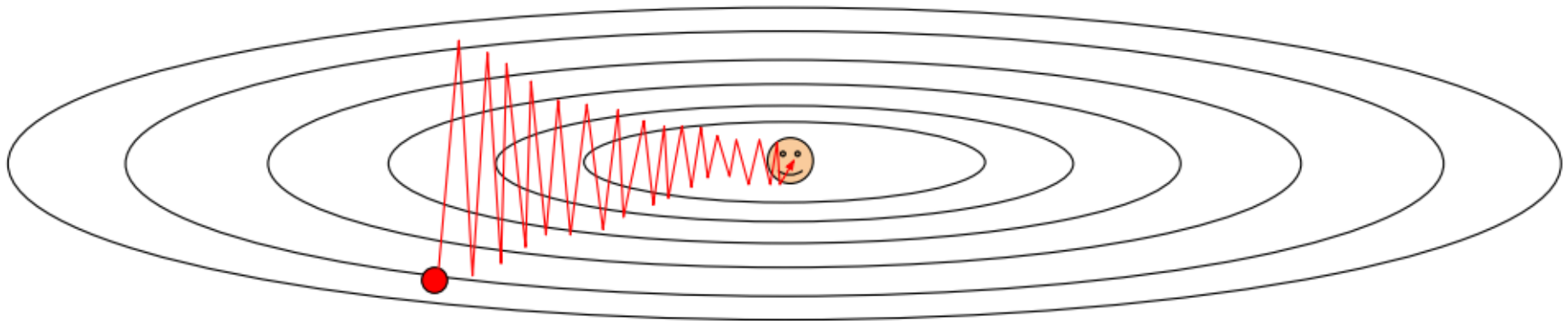
What if loss changes quickly in one direction and slowly in another?



# SGD: Problems

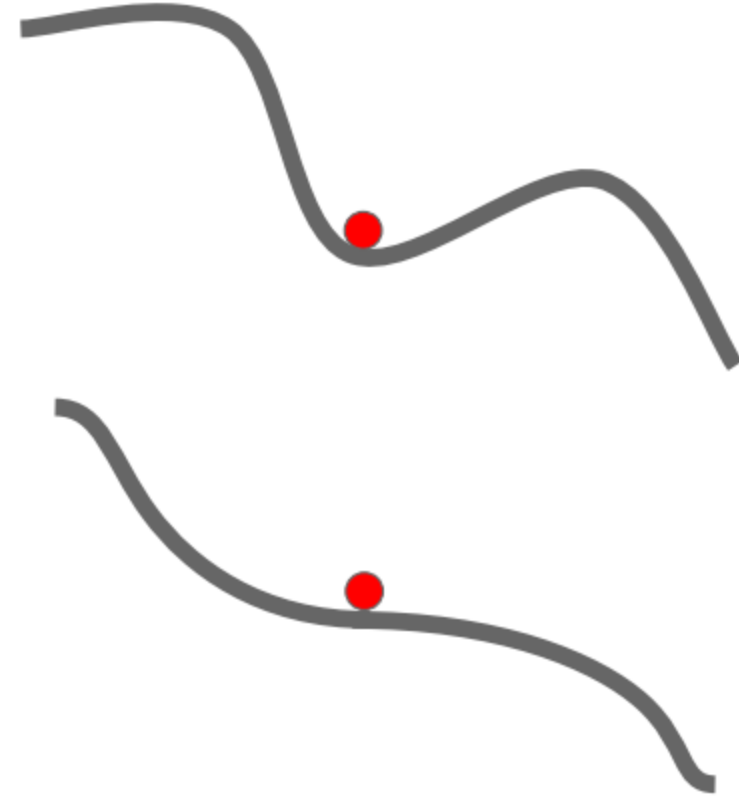
What if loss changes quickly in one direction and slowly in another?

Very slow progress along shallow dimension, jitter along steep direction



# SGD: Problems

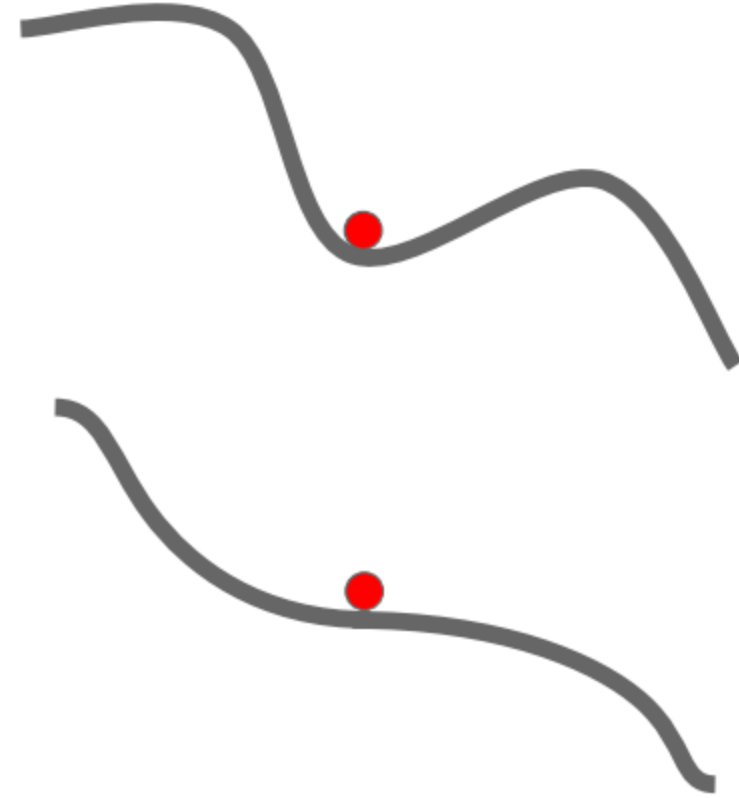
What if the loss function has a **local minima** or **saddle point**?



# SGD: Problems

What if the loss function has a **local minima** or **saddle point**?

Zero gradient,  
gradient descent  
gets stuck



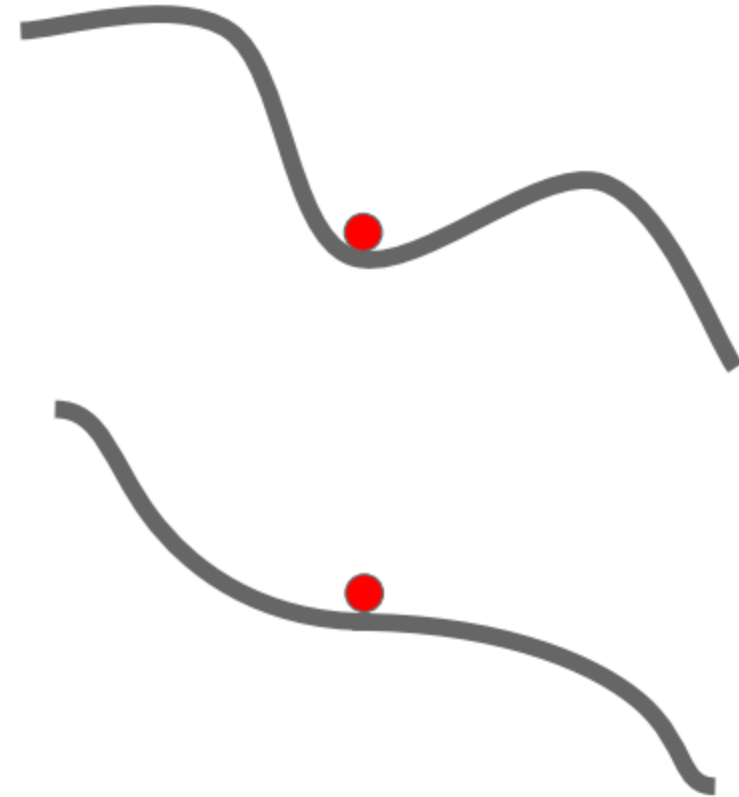
# SGD: Problems

What if the loss function has a **local minima** or **saddle point**?

Zero gradient,  
gradient descent  
gets stuck

Saddle points much more  
common in high  
dimension

Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

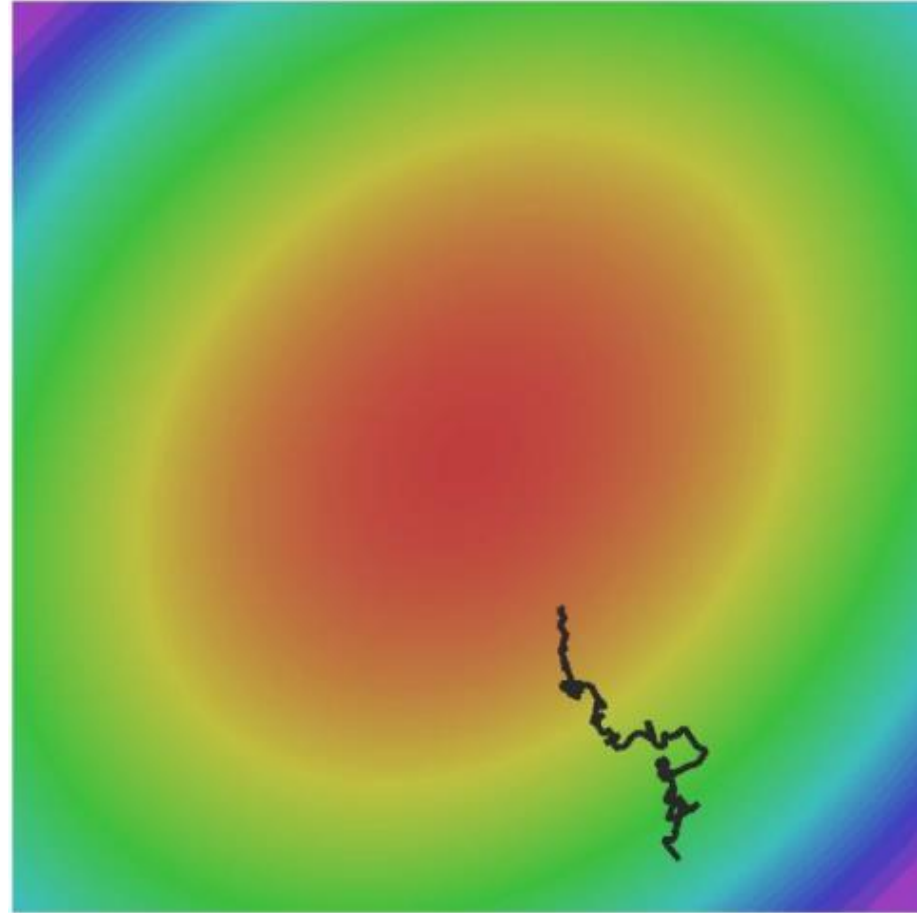


# SGD: Problems

Our gradients come from **minibatches** so they can be **noisy!**

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W)$$



# SGD + Momentum

## SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:  
    dx = compute_gradient(x)  
    x += learning_rate * dx
```



# SGD + Momentum

## SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
```

```
    dx = compute_gradient(x)
```

```
    x += learning_rate * dx
```

## SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

# SGD + Momentum

## SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
    dx = compute_gradient(x)
    x += learning_rate * dx
```

## SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x += learning_rate * vx
```

# SGD + Momentum

## SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
```

```
    dx = compute_gradient(x)
    x += learning_rate * dx
```

## SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
```

```
while True:
```

```
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x += learning_rate * vx
```

- Build up “velocity” in any direction that has consistent gradient
- Rho gives “friction”; typically rho=0.9 or 0.99

# SGD + Momentum

## SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
```

```
    dx = compute_gradient(x)
```

```
    x += learning_rate * dx
```

## SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

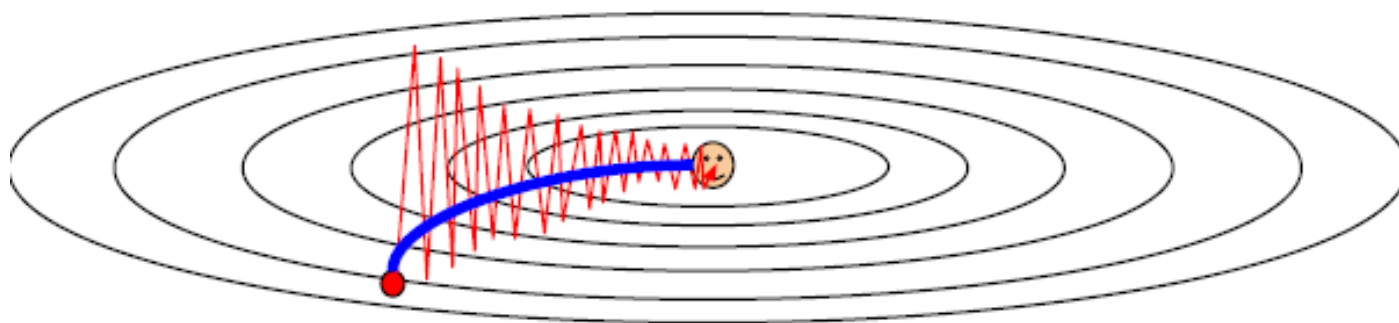
```
vx = 0
```

```
while True:
```

```
    dx = compute_gradient(x)
```

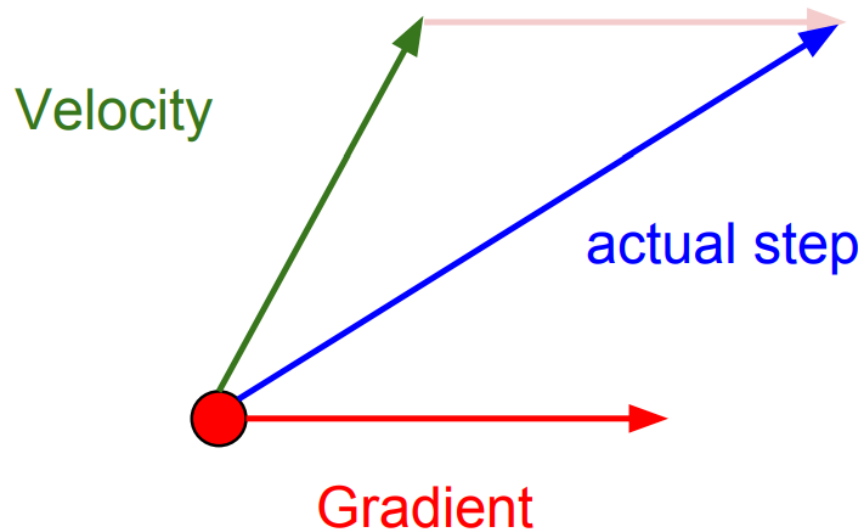
```
    vx = rho * vx + dx
```

```
    x += learning_rate * vx
```



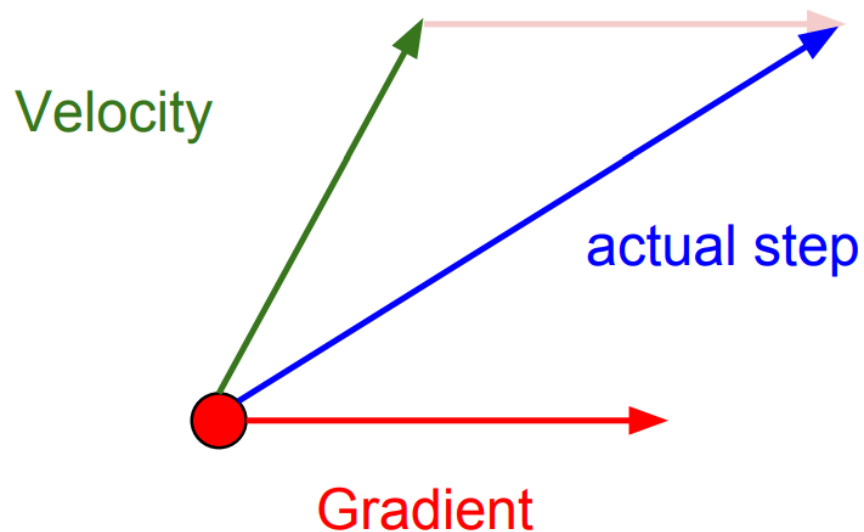
# SGD + Momentum

Momentum update:

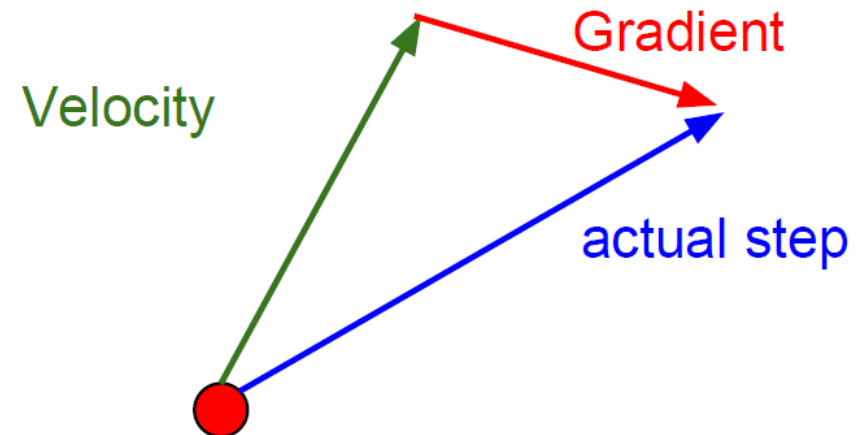


# Nesterov Momentum

Momentum update:



Nesterov Momentum





# Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

# Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

Change of variables  $\tilde{x}_t = x_t + \rho v_t$  and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$

$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1}$$

# Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

Change of variables  $\tilde{x}_t = x_t + \rho v_t$  and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$

$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1}$$

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Source: cs231n

# AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

# AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

What happens to the step size over long time?

# AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

What happens to the step size over long time?

Effective learning rate diminishing problem



# RMSProp

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

**AdaGrad**



**RMSProp**

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

# Adam

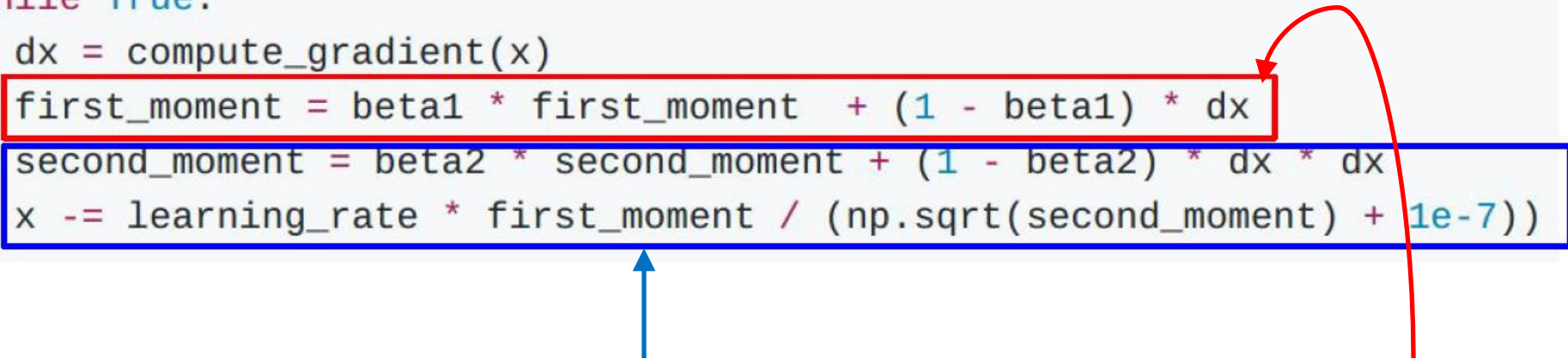
Kingma and Ba, “Adam:  
A method for stochastic  
optimization”, ICLR 2015

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

# Adam

Kingma and Ba, "Adam:  
A method for stochastic  
optimization", ICLR 2015

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

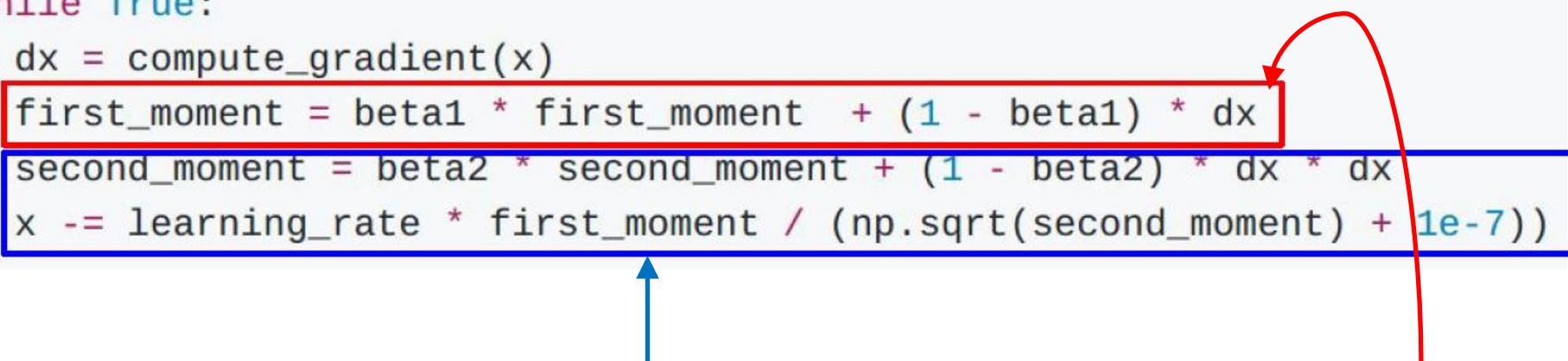


Sort of like RMSProp with Momentum

# Adam

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```



Sort of like **RMSProp** with **Momentum**

Problem:

Initially, second\_moment=0 and beta2=0.999

After 1<sup>st</sup> iteration, second\_moment -> close to zero

So, very large step for update of x

# Adam (with Bias correction)

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

AdaGrad/  
RMSProp

Bias Correction

Bias correction for the fact that first and second  
moment estimates start at zero

Momentum



# Adam (with Bias correction)

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

AdaGrad/  
RMSProp

Bias Correction

Momentum

Bias correction for the fact that first and second  
moment estimates start at zero

**Adam with  $\text{beta1} = 0.9$ ,  
 $\text{beta2} = 0.999$ , and  $\text{learning\_rate} = 1\text{e-}3$  or  $5\text{e-}4$   
is a great starting point for many models!**



# AMSGrad

Some minibatches (rarely occur) provide large and informative gradients, exponential averaging diminishes their influence, which leads to poor convergence.

# AMSGrad

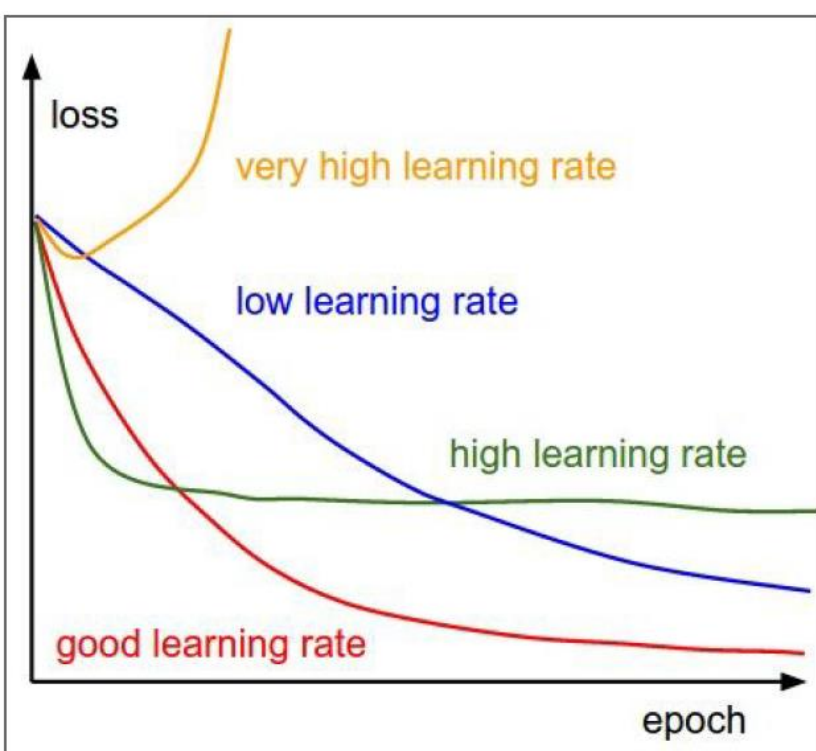
Some minibatches (rarely occur) provide large and informative gradients, exponential averaging diminishes their influence, which leads to poor convergence.

AMSGrad that uses the maximum of past squared gradients  $v_t$

$$\begin{aligned}m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\\hat{v}_t &= \max(\hat{v}_{t-1}, v_t) \\\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} m_t\end{aligned}$$

# Learning Rate

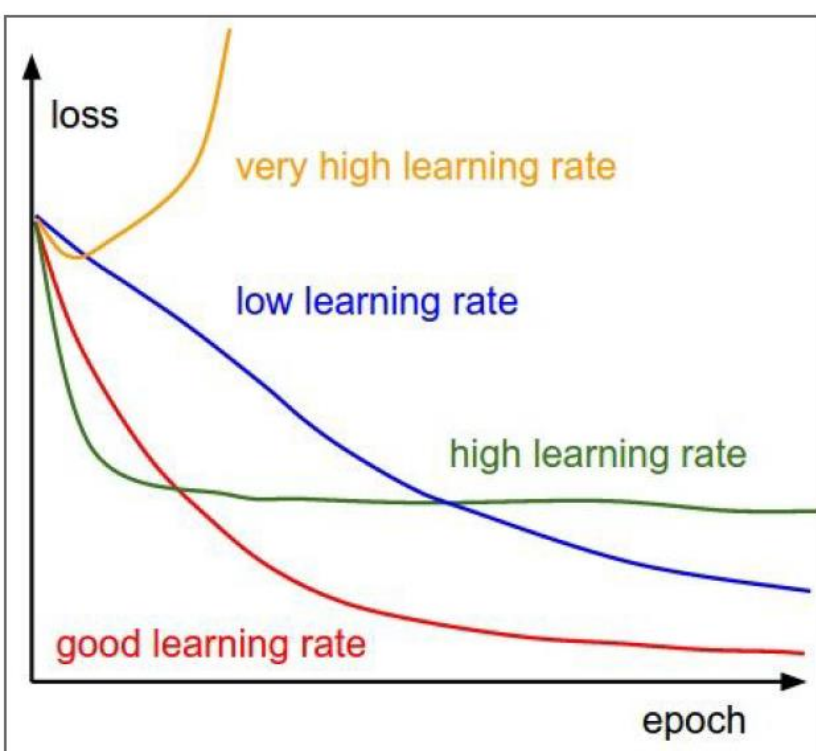
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

# Learning Rate

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> **Learning rate decay over time!**

**step decay:**

e.g. decay learning rate by half every few epochs.

**exponential decay:**

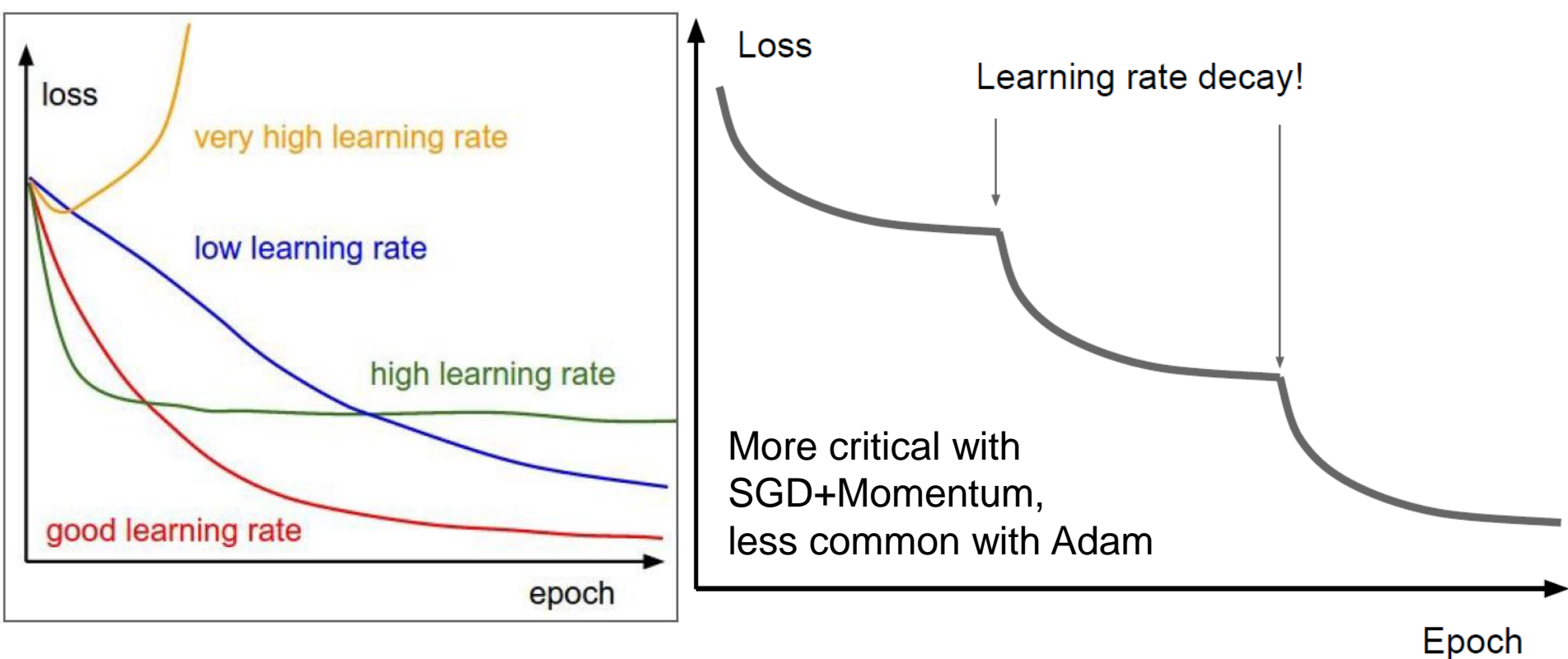
$$\alpha = \alpha_0 e^{-kt}$$

**1/t decay:**

$$\alpha = \alpha_0 / (1 + kt)$$

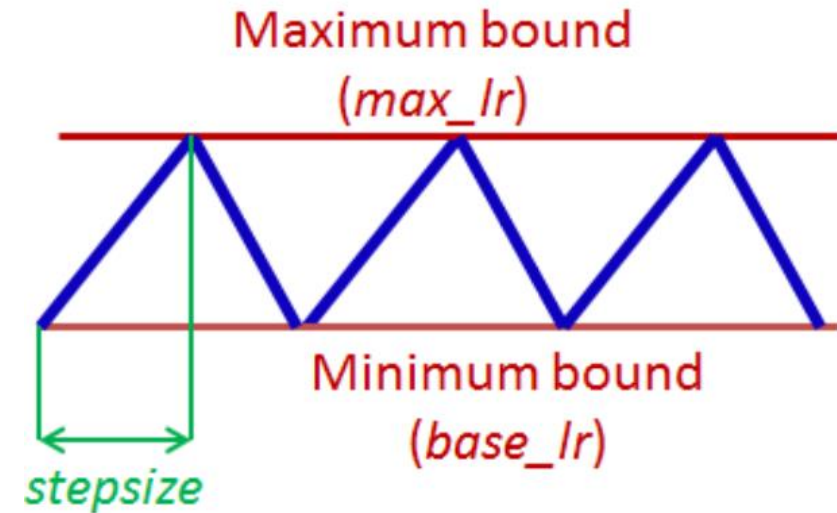
# Learning Rate

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



# Learning Rate

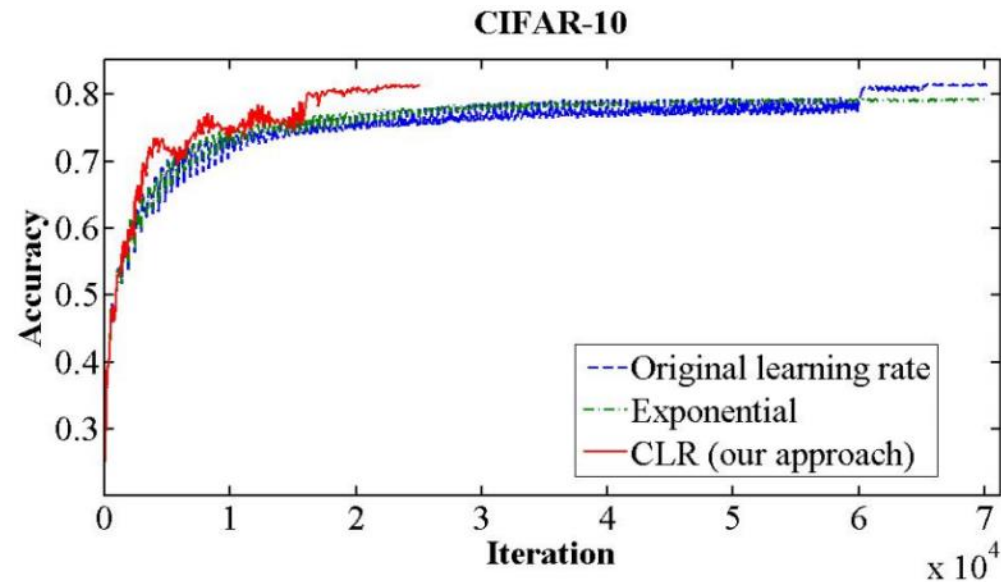
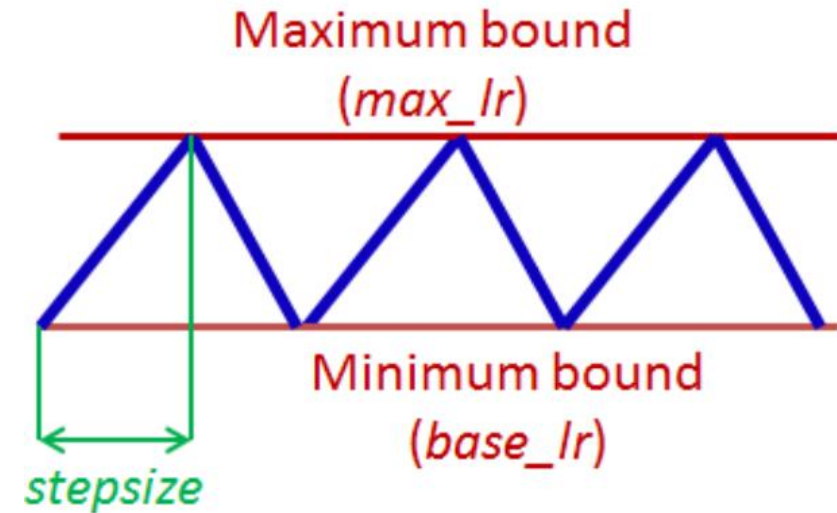
## Cyclic Learning Rate



Smith, Leslie N. "Cyclical learning rates for training neural networks." *WACV 2017*.

# Learning Rate

## Cyclic Learning Rate



Smith, Leslie N. "Cyclical learning rates for training neural networks." *WACV 2017*.

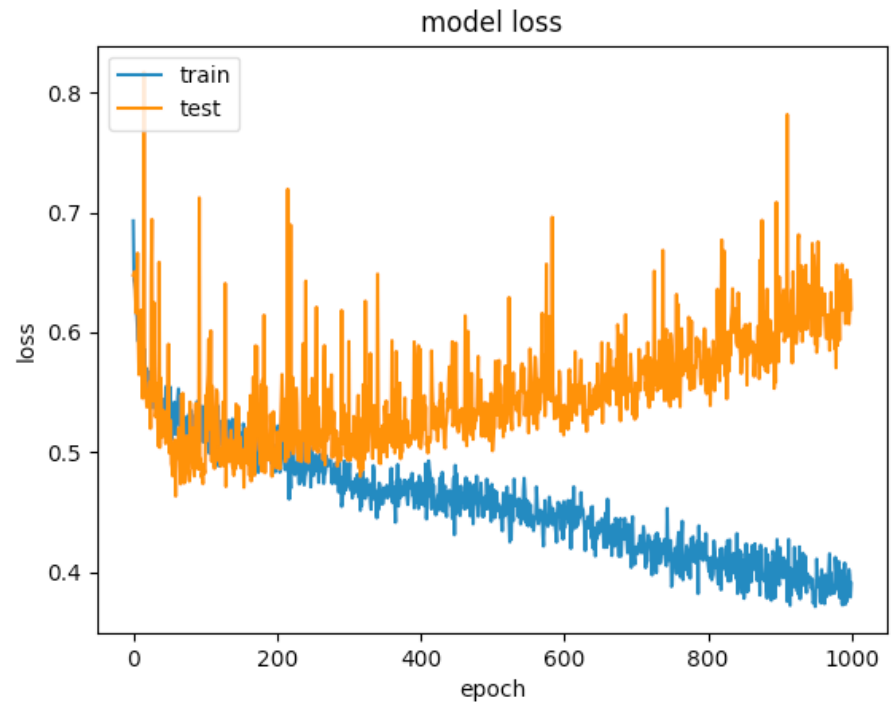
# Optimizer and Learning Rate

## In Practice:

- **Adam** is a good default choice in most cases
- **Learning rate** with step decay is commonly used

More Optimizer: <http://ruder.io/optimizing-gradient-descent/>





# Regularization

Image Source: <https://stackoverflow.com/questions/44909134/how-to-avoid-overfitting-on-a-simple-feed-forward-network/44985765>

# Regularization

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}$$

**Data loss:** Model predictions  
should match training data

# Regularization

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too* well on training data

# Regularization

$\lambda$  = regularization strength  
(hyperparameter)

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too* well on training data

# Regularization

$\lambda$  = regularization strength  
(hyperparameter)

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too* well on training data

## Simple examples

L2 regularization:  $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization:  $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2):  $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

# Regularization

$\lambda$  = regularization strength  
(hyperparameter)

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too* well on training data

Why regularize?

- Express preferences over weights
- Make the model *simple* so it works on test data
- Improve optimization by adding curvature

# Regularization

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

# Regularization

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1 \cdot x = w_2 \cdot x = 1$$



# Regularization

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1 \cdot x = w_2 \cdot x = 1$$

**Which W to consider?**

# Regularization

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1 \cdot x = w_2 \cdot x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

# Regularization

$$x = [1,1,1,1]$$

$$w_1 = [1,0,0,0]$$

$$w_2 = [0.25,0.25,0.25,0.25]$$

$$w_1 \cdot x = w_2 \cdot x = 1$$

L2 regularization likes to  
“spread out” the weights

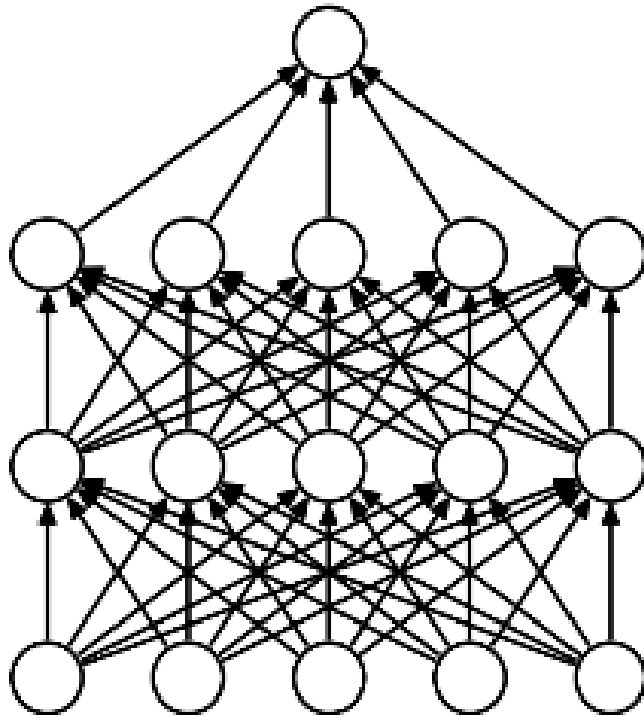
L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

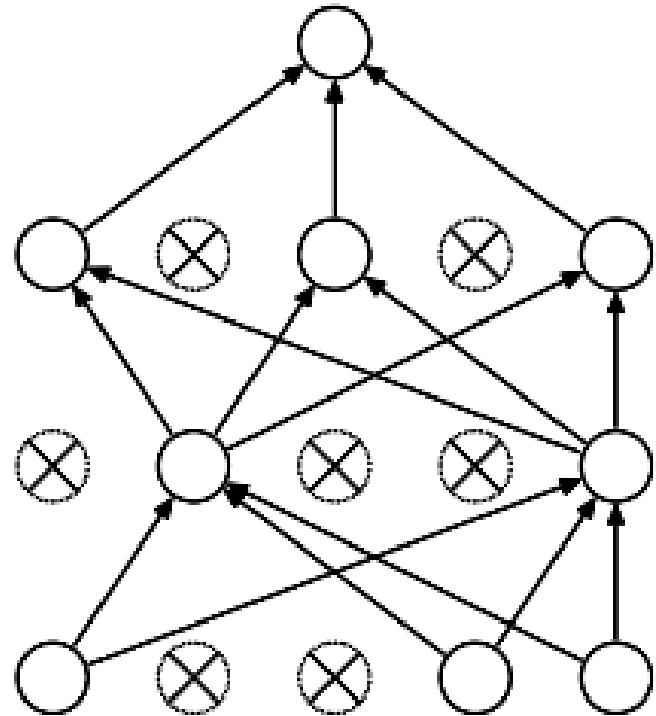
# Dropout

# Dropout

In each forward pass, randomly set some neurons to zero  
Probability of dropping is a hyperparameter; 0.5 is common



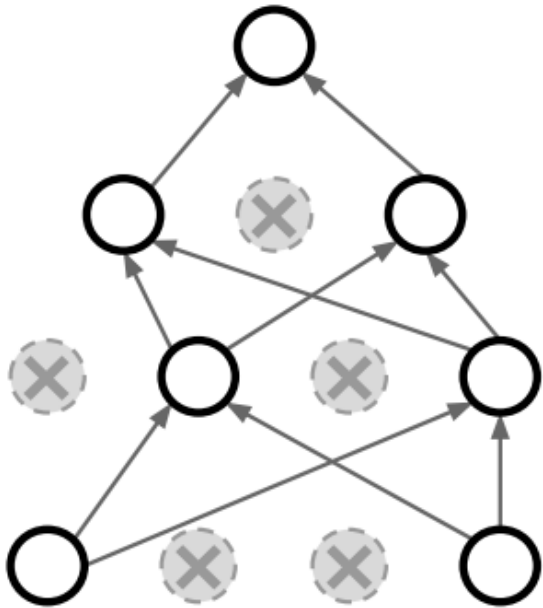
(a) Standard Neural Net



(b) After applying dropout.

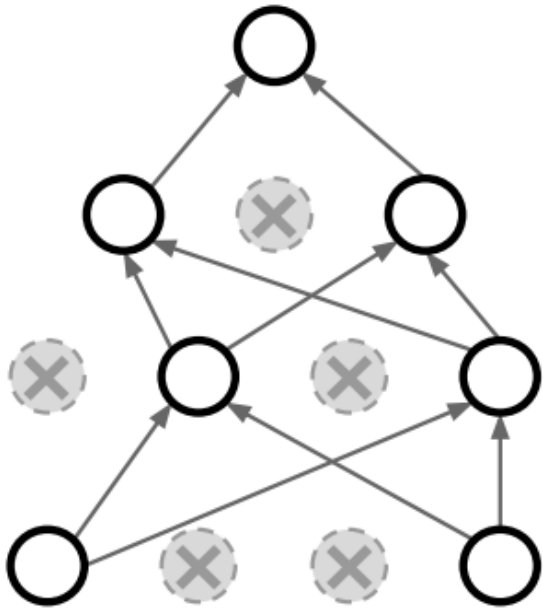
# Dropout

How can this possibly be a good idea?

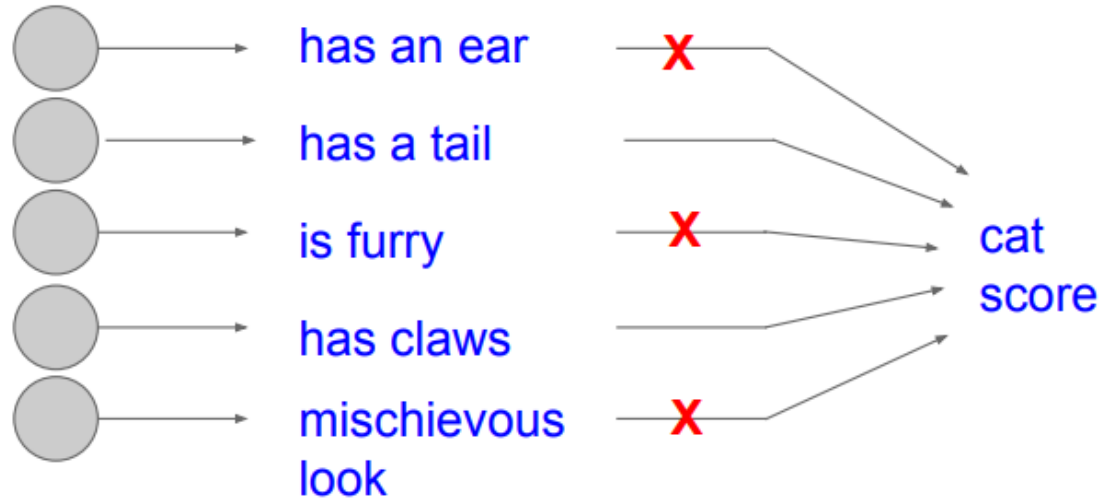


# Dropout

How can this possibly be a good idea?



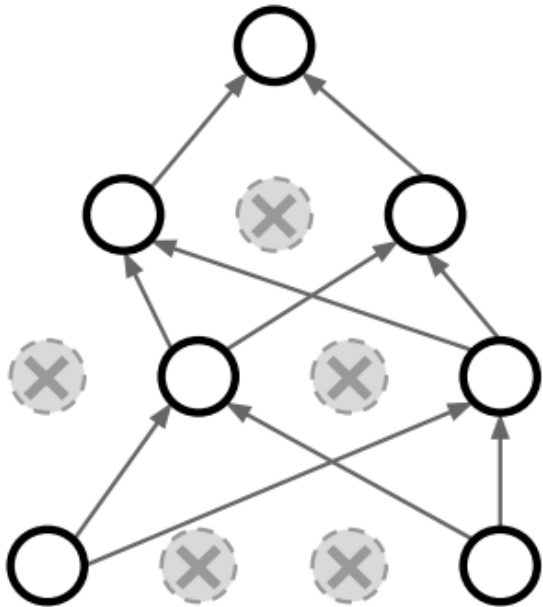
Forces the network to have a redundant representation;  
Prevents co-adaptation of features



# Dropout

How can this possibly be a good idea?

**Dropout is training a large ensemble of models (that share parameters).**



Intuition: successful conspiracies

- 50 people planning a conspiracy
- Strategy A: plan a big conspiracy involving 50 people
  - Likely to fail. 50 people need to play their parts correctly.
- Strategy B: plan 10 conspiracies each involving 5 people
  - Likely to succeed!



# Dropout: Test Time

```
def predict(X):  
    # ensembled forward pass  
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations  
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations  
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always  
=> We must scale the activations so that for each neuron:  
output at test time = expected output at training time

**More common: “Inverted dropout”**

# Dropout: More common: “Inverted dropout”

We drop and scale at train time and don't do anything at test time.

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

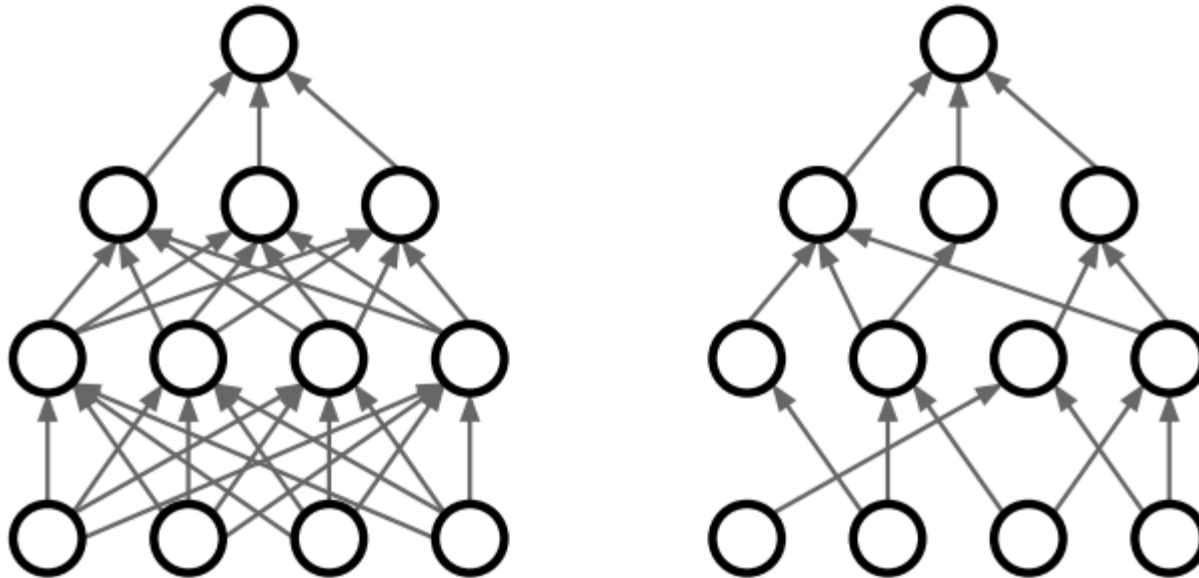
def predict(X):
    # ensemble forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

test time is unchanged!



# DropConnect

Dropping some connections



Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013

Source: cs231n

# Batch Normalization

# Batch Normalization

“We want zero-mean unit-variance activations? lets make them so.”

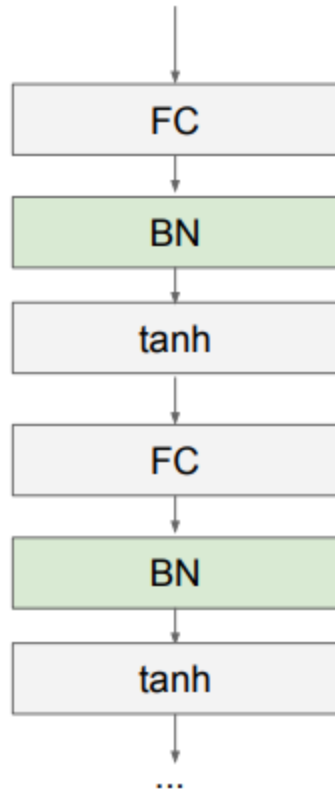
# Batch Normalization

“We want zero-mean unit-variance activations? lets make them so.”

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

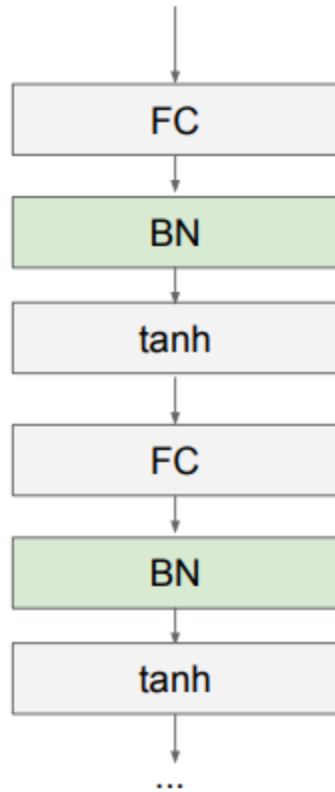
$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

# Batch Normalization



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

# Batch Normalization



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

**Problem:**  
do we necessarily want a zero-mean unit-variance input?



# Batch Normalization

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

# Batch Normalization

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathbb{E}[x^{(k)}]$$

to recover the identity mapping.

# Batch Normalization

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots x_m\}$ ;

Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

# Batch Normalization

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization

# Batch Normalization

Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch.

Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

# Batch Normalization: Recent Trends

## Layer Normalization:

Ba, Kiros, and Hinton, “Layer Normalization”, arXiv 2016

## Instance Normalization:

Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

## Group Normalization:

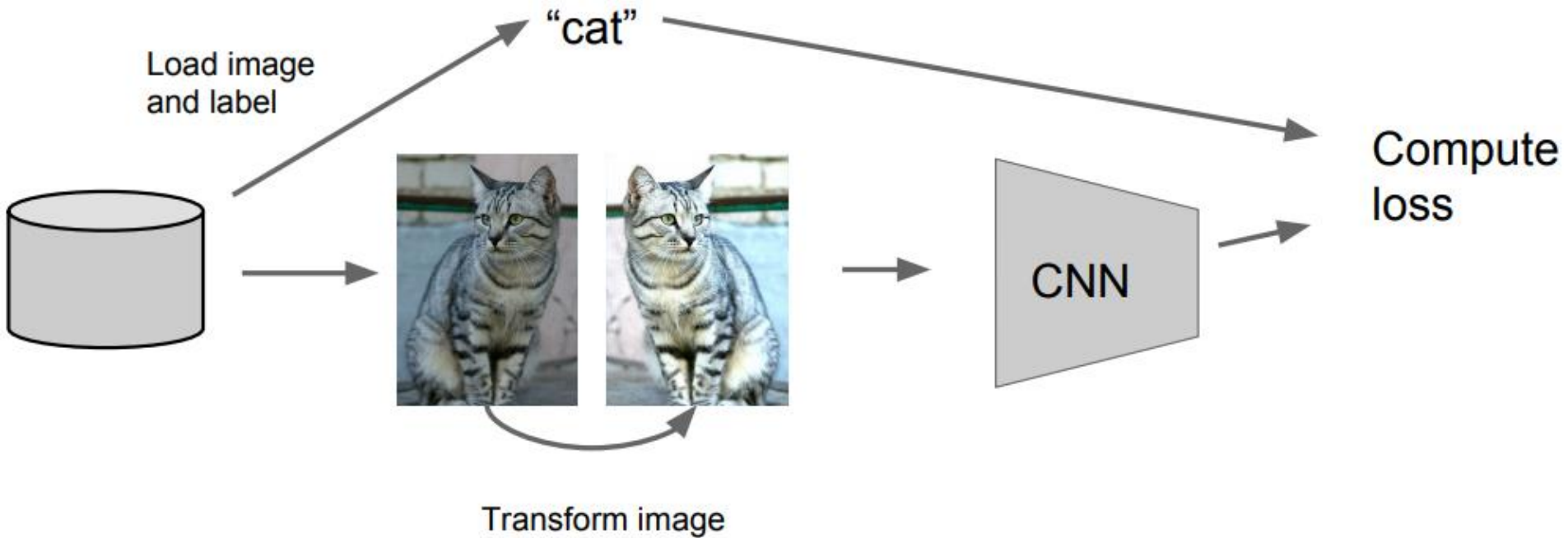
Wu and He, “Group Normalization”, arXiv 2018  
(Appeared 3/22/2018)

## Decorrelated Normalization:

Huang et al, “Decorrelated Batch Normalization”, arXiv 2018  
(Appeared 4/23/2018)

# Data Augmentation

# Data Augmentation (Jittering)





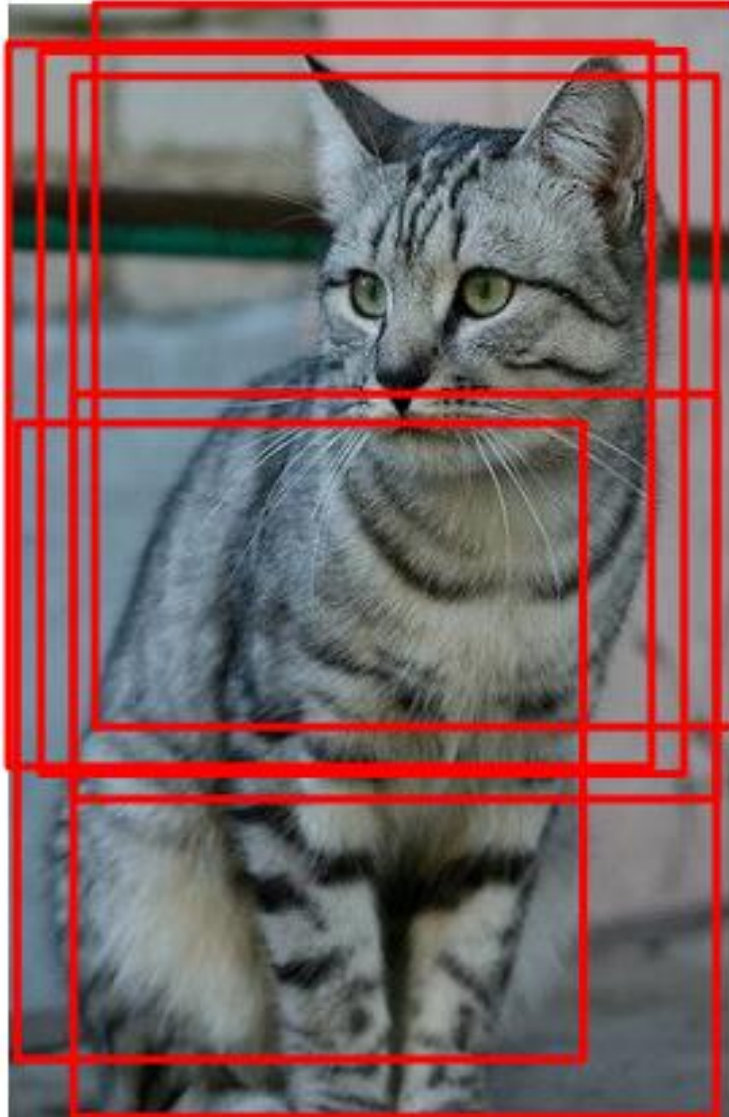
# Data Augmentation (Jittering)

## Horizontal Flips



# Data Augmentation (Jittering)

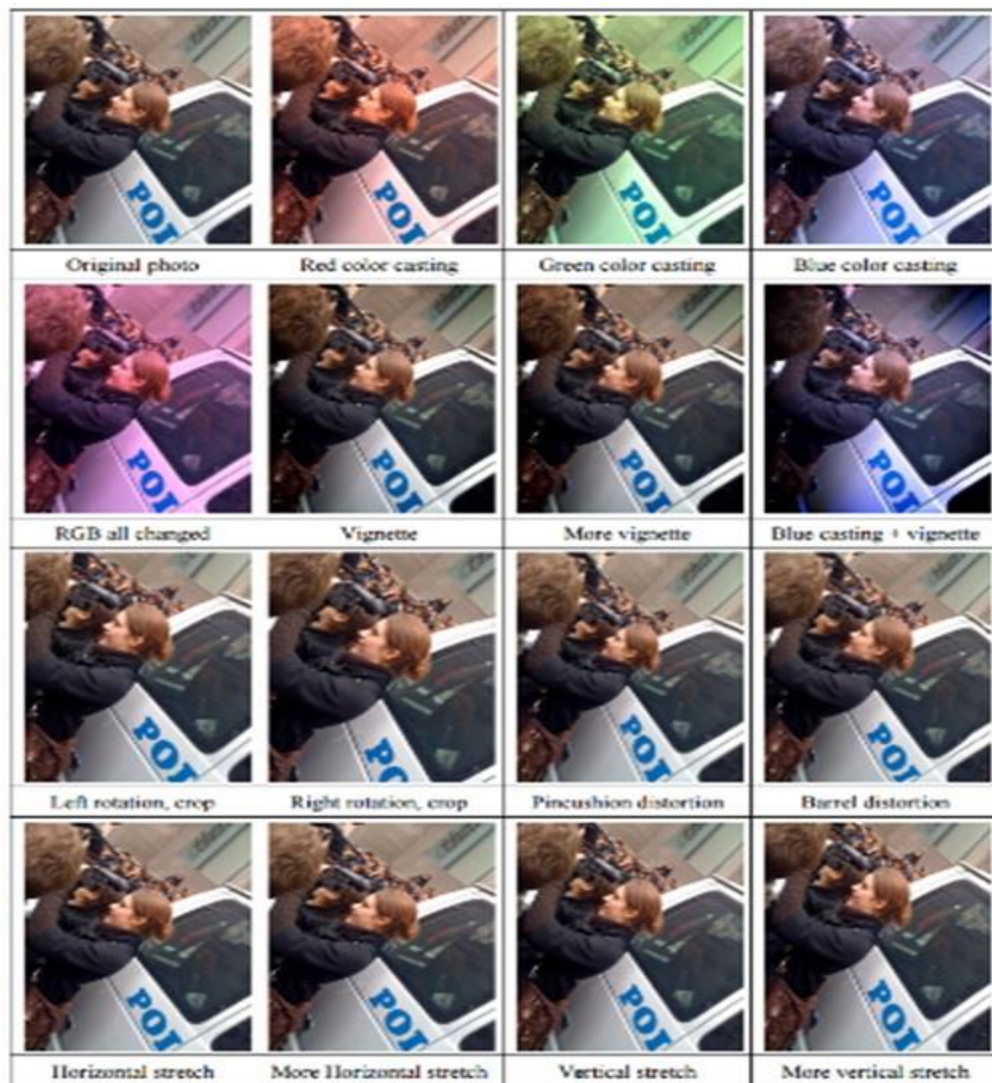
Random crops and scales



# Data Augmentation (Jittering)

- Create *virtual* training samples
- Get creative for your problem!

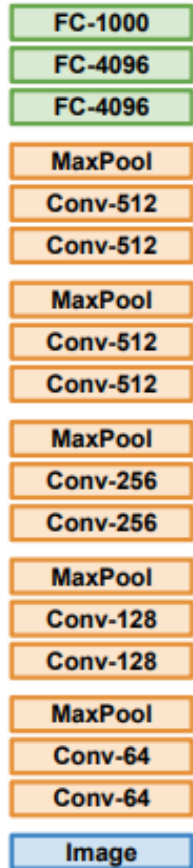
- Horizontal flip
- Random crop
- Color casting
- Randomize contrast
- Randomize brightness
- Geometric distortion
- Rotation
- Photometric changes



# Transfer Learning

# Transfer Learning with CNNs

## 1. Train on Imagenet



Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014

Razavian et al, “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014

Source: cs231n

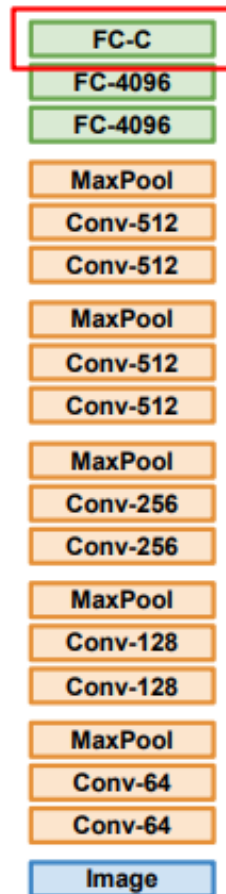


# Transfer Learning with CNNs

1. Train on Imagenet



2. Small Dataset (C classes)



Reinitialize  
this and train

Freeze these

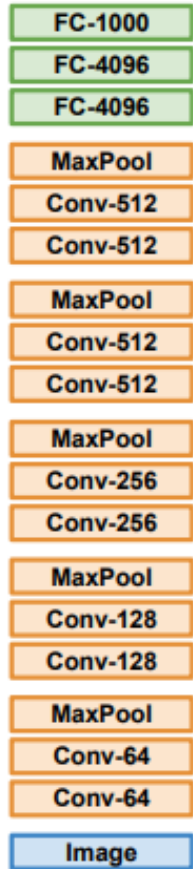
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

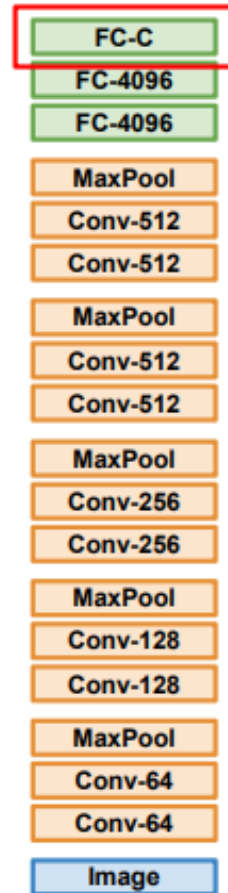
Source: cs231n

# Transfer Learning with CNNs

## 1. Train on Imagenet



## 2. Small Dataset (C classes)



Reinitialize  
this and train

Freeze these

### 3. Bigger dataset



## Train these

With bigger dataset, train more layers

Freeze these

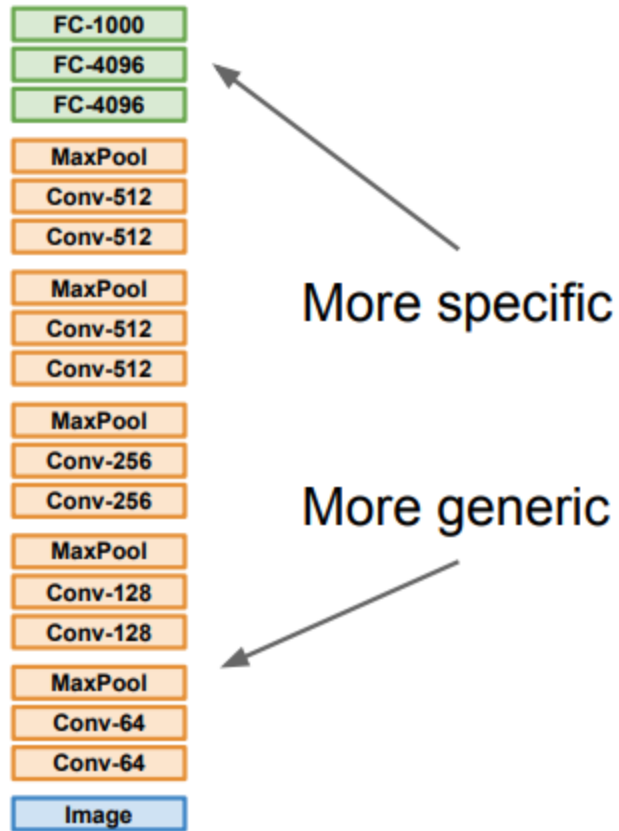
Lower learning rate  
when finetuning;  
 $1/10$  of original LR  
is good starting  
point

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

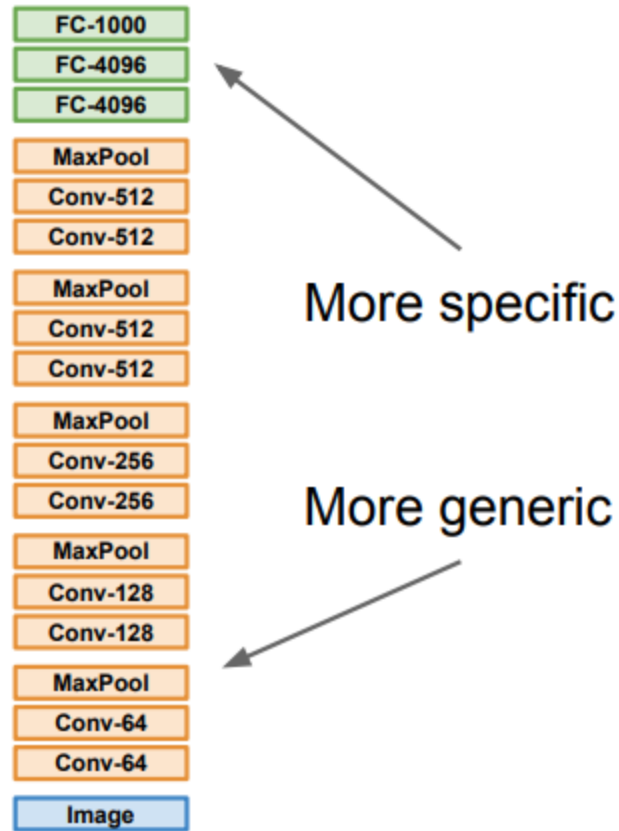
Source: cs231n

# Transfer Learning with CNNs



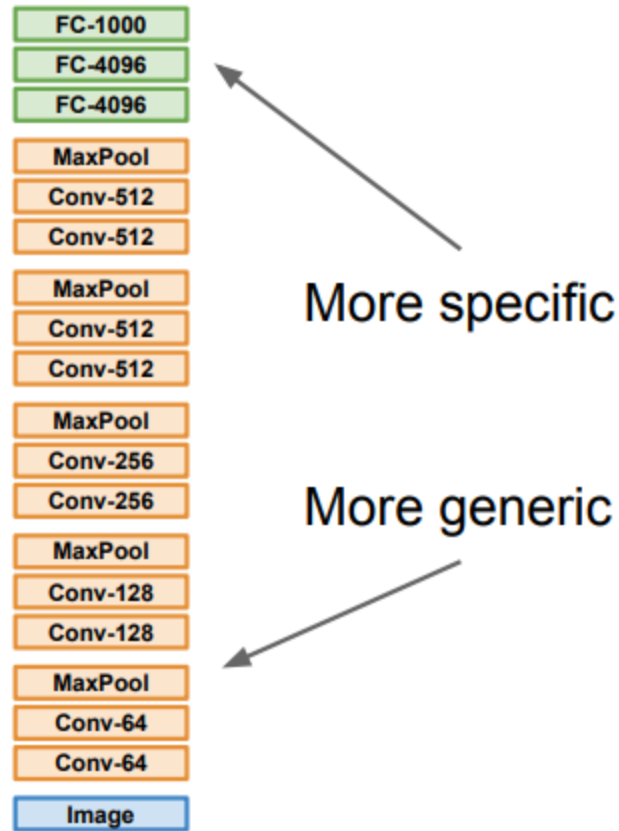


# Transfer Learning with CNNs



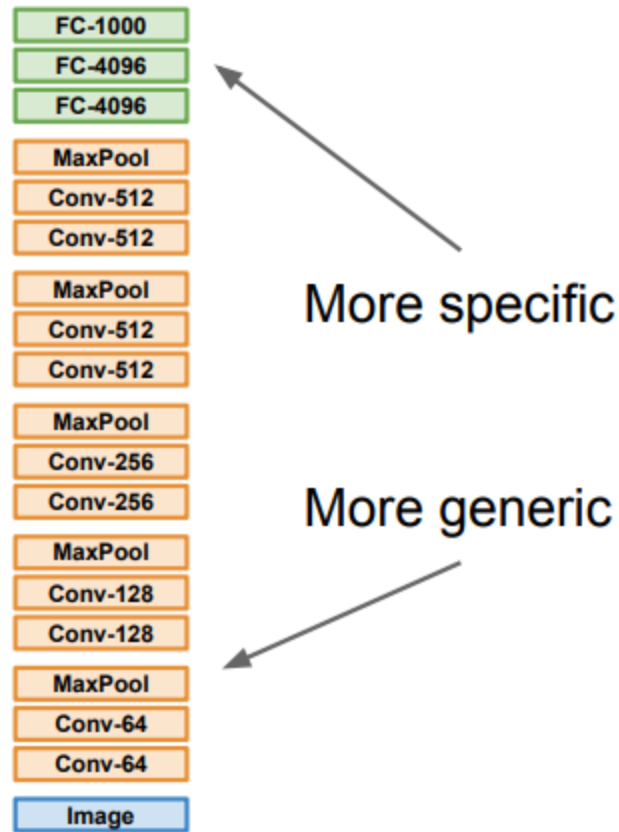
	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	
quite a lot of data		

# Transfer Learning with CNNs



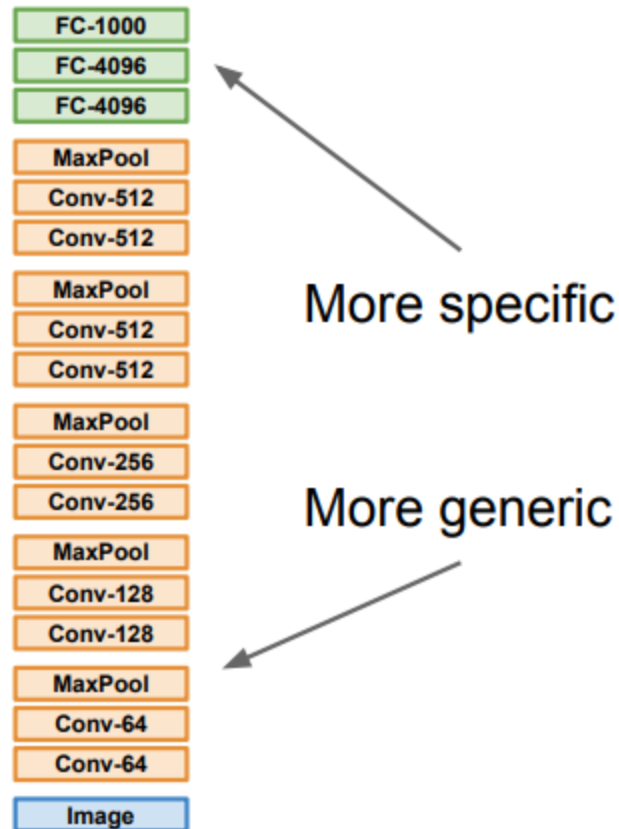
	<b>very similar dataset</b>	<b>very different dataset</b>
<b>very little data</b>	Use Linear Classifier on top layer	
<b>quite a lot of data</b>	Finetune a few layers	

# Transfer Learning with CNNs



	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

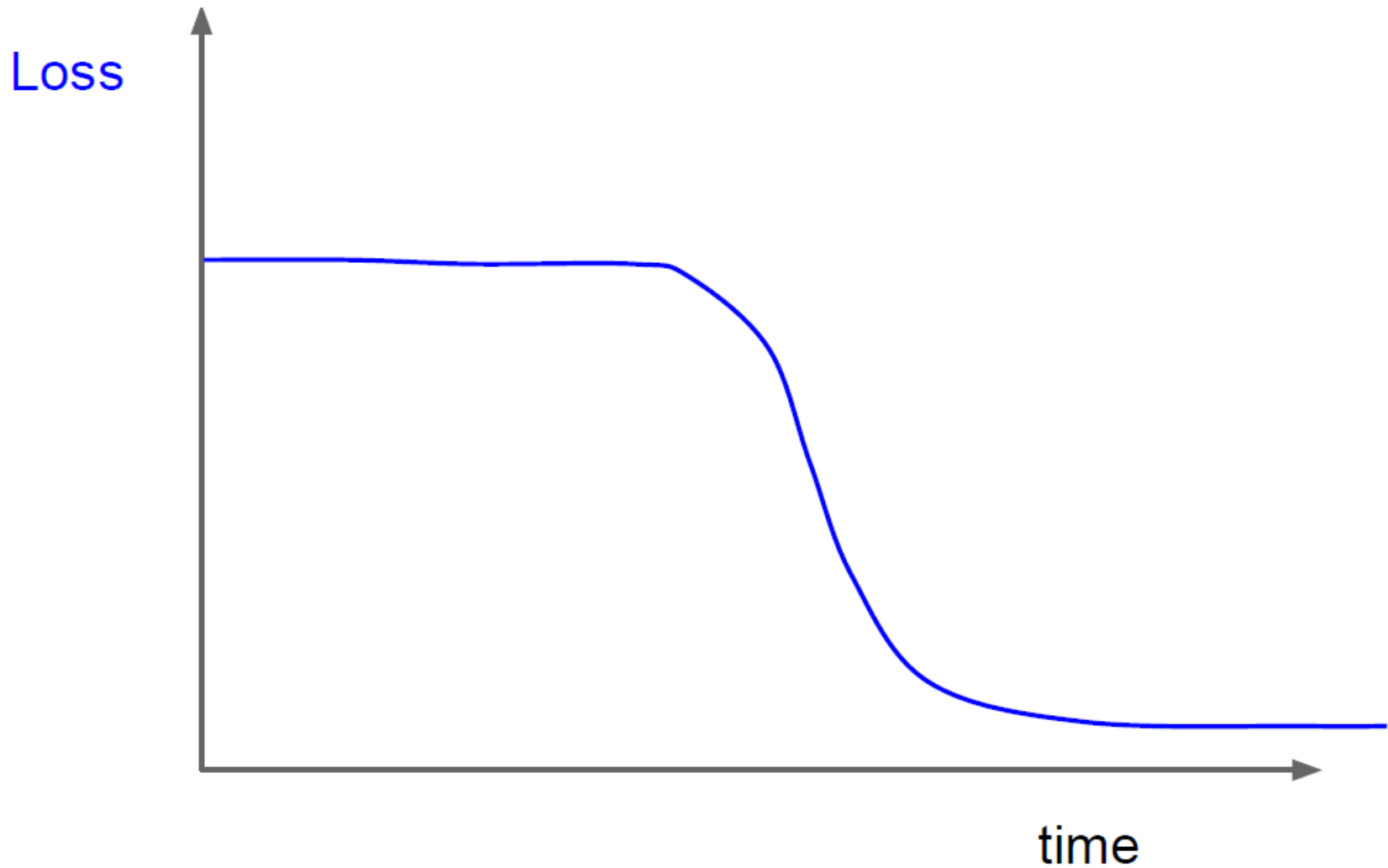
# Transfer Learning with CNNs



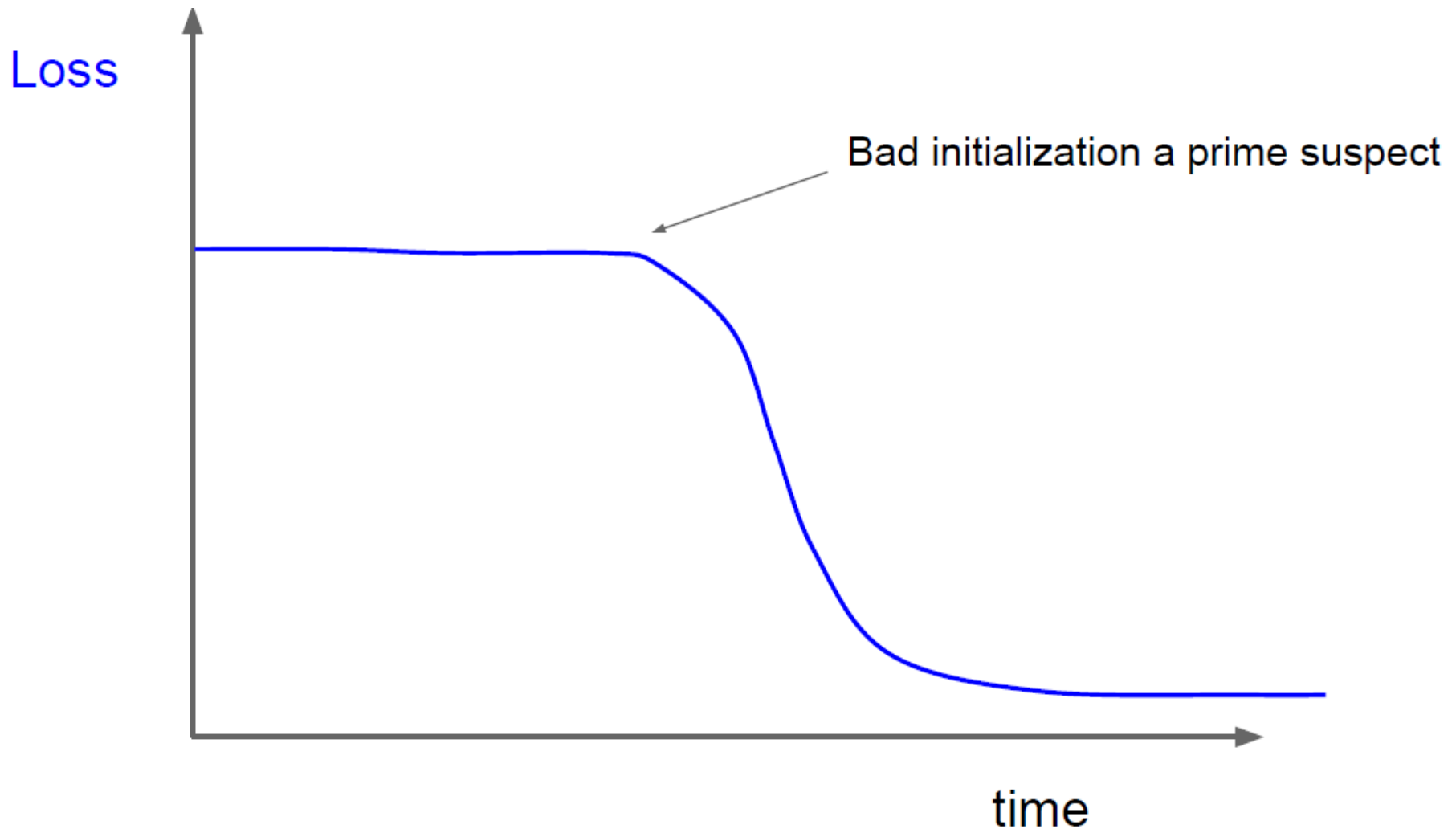
	<b>very similar dataset</b>	<b>very different dataset</b>
<b>very little data</b>	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different stages
<b>quite a lot of data</b>	Finetune a few layers	Finetune a larger number of layers

# Monitor and Visualize the Loss Curve

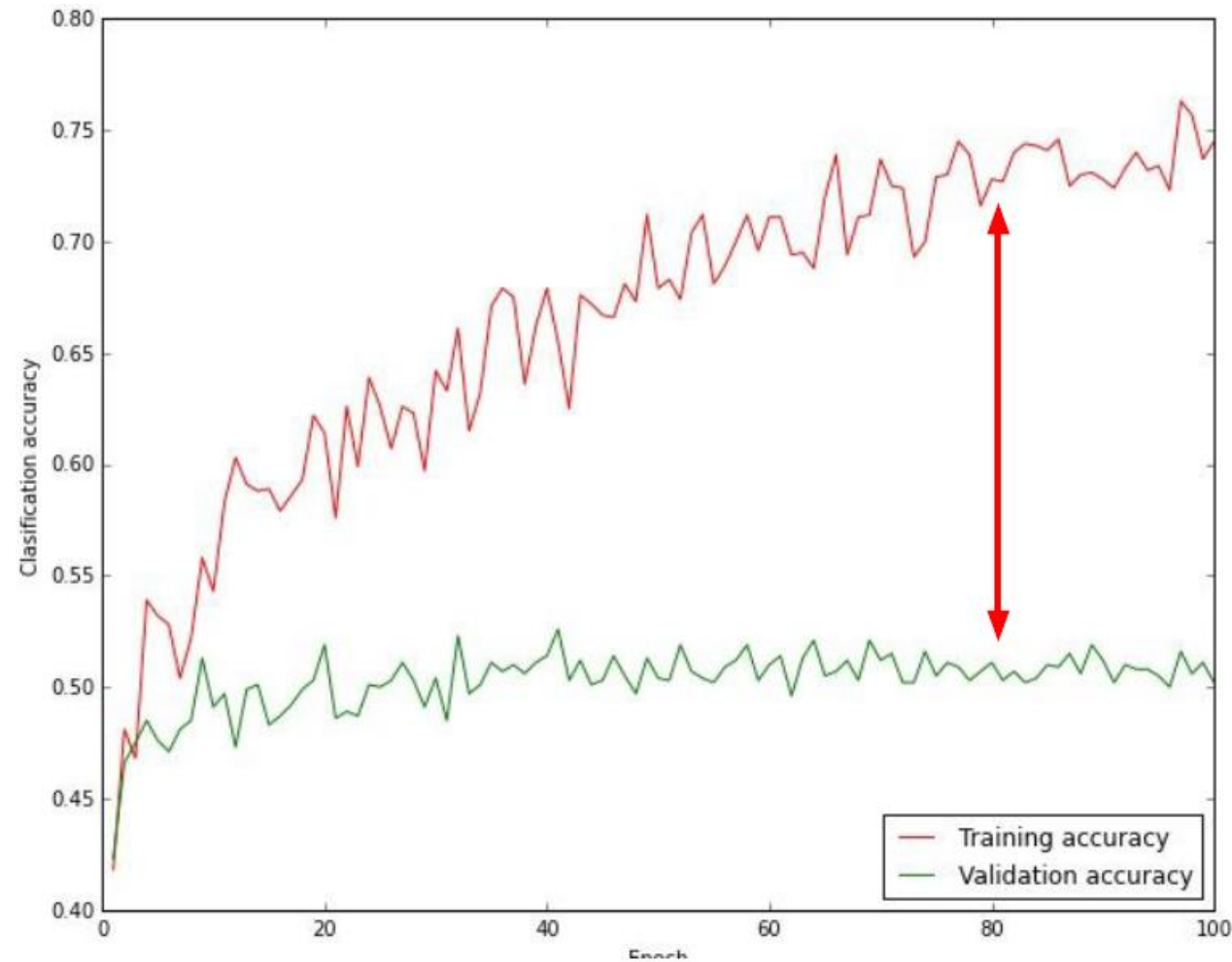
# Monitor and visualize the loss curve



# Monitor and visualize the loss curve



# Monitor and visualize the loss curve

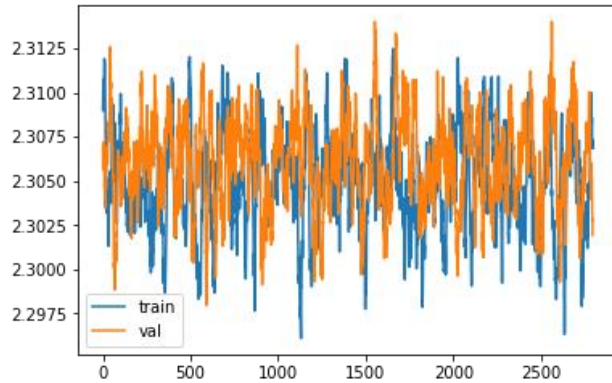


big gap = overfitting  
=> increase  
regularization  
strength?

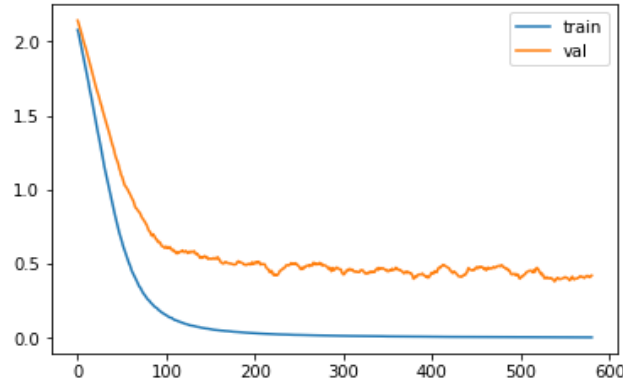
no gap  
=> increase model  
capacity?



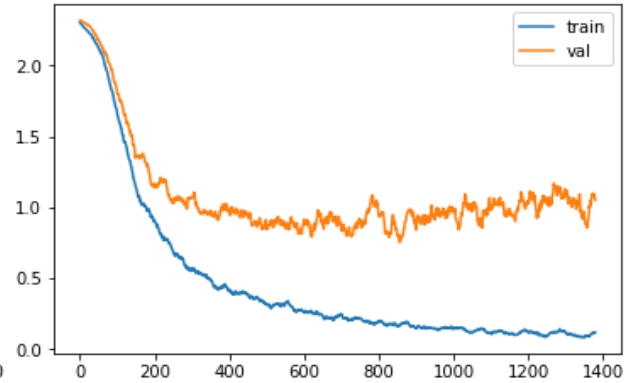
# Monitor and visualize the loss curve



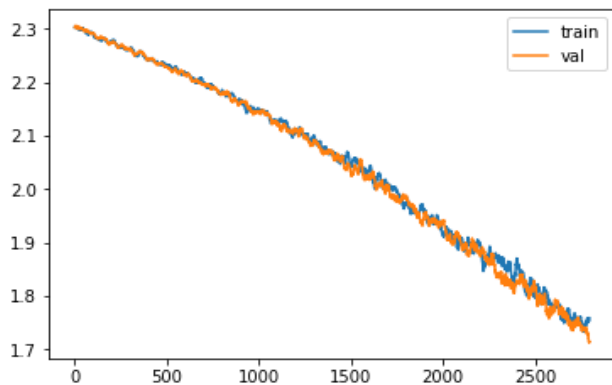
Not learning: gradients not applied to weights



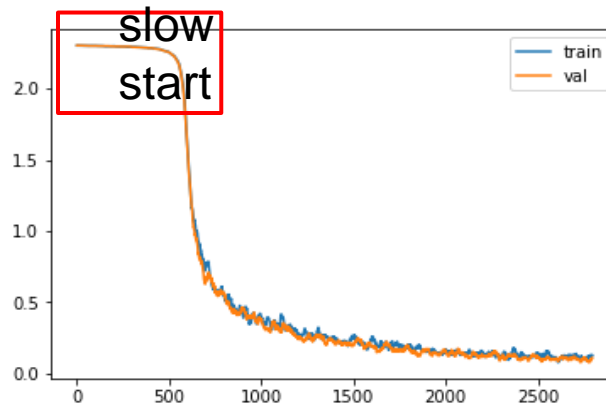
Overfit: model too large/dataset too small



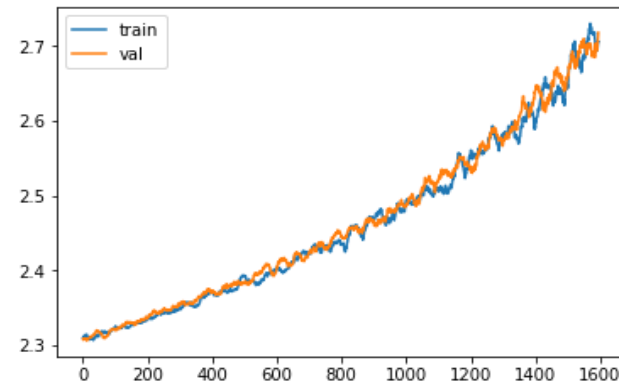
More extreme case of overfitting



Not converged yet: need longer training

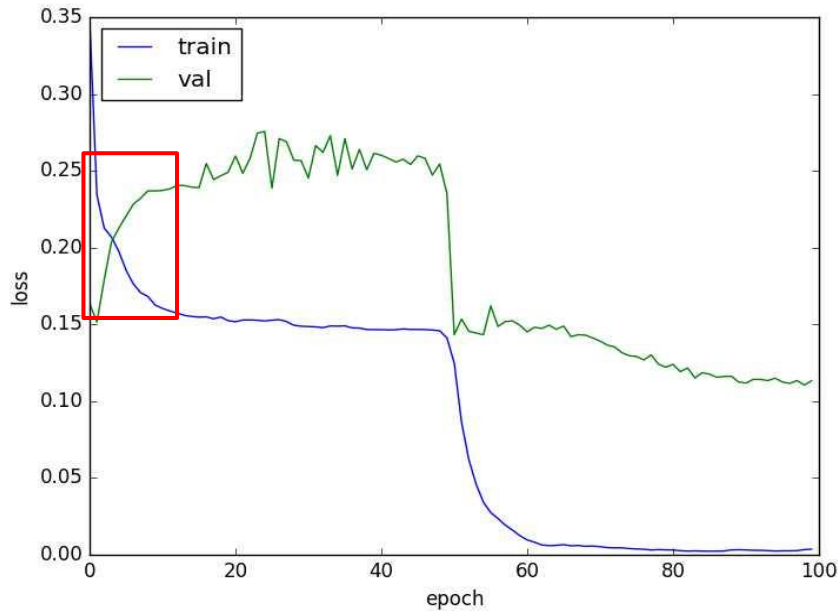


Slow start: initialization weights too small

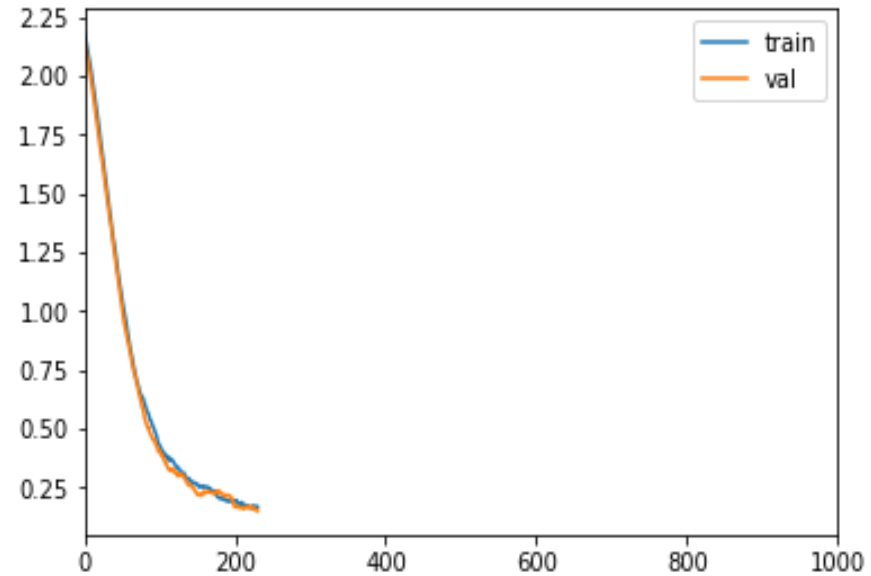


Applied the negative of gradients

# Monitor and visualize the loss curve



Problem: val set too small,  
statistics not meaningful



Get nans in the loss after a number of  
iterations: caused by high learning rate  
and numerical instability in models

# Things to remember

- **Training CNN**
  - Adam is common (AMSGrad can be tried)
  - Learning rate: Step decay, Cyclic learning rate
  - Transfer learning, Fine tuning
- **Regularization**
  - L2/L1/Elastic regularization
  - Dropout and Dropconnect
  - Batch Norm
  - Data Augmentation: Flip, Crop, Contrast, etc.
- **Interpreting Loss**
  - Bad initialization
  - Overfitting
  - Slow/High learning rates
  - Update in wrong direction
  - Etc.

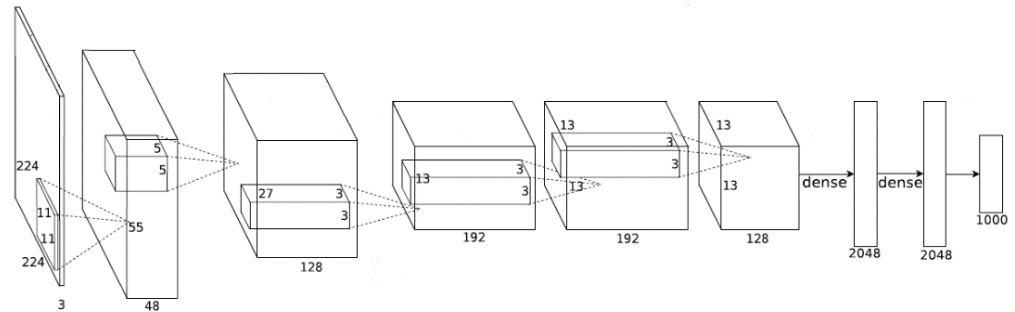
# Acknowledgements

- Thanks to the following researchers for making their teaching/research material online
  - Forsyth
  - Steve Seitz
  - Noah Snavely
  - J.B. Huang
  - Derek Hoiem
  - D. Lowe
  - A. Bobick
  - S. Lazebnik
  - K. Grauman
  - R. Zaleski
  - Antonio Torralba
  - Rob Fergus
  - Leibe
  - And many more .....

# Next Class

## CNN Architectures: Plain Models

- LeNet
- AlexNet
- ZFNet
- VggNet
- Network in Network



## CNN Architectures: DAG Models

- GoogLeNet
- ResNet
- Pre-act ResNet
- SENet
- DenseNet
- ResNetXt
- Etc.

