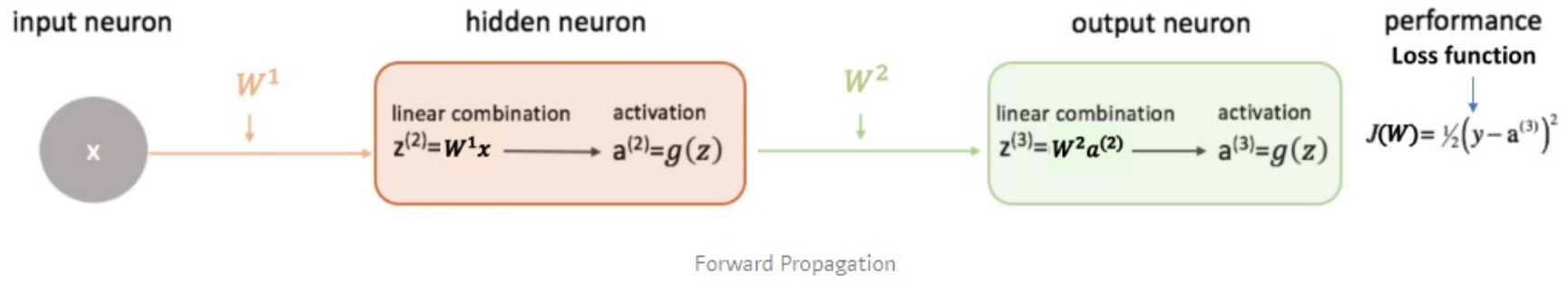
Practice in learning neural network

Many slides from Fei-fei Li's course @Standford and Xiaogang Wang's tutorial @CUHK
http://cs231n.stanford.edu/ http://www.ee.cuhk.edu.hk/~xgwang/

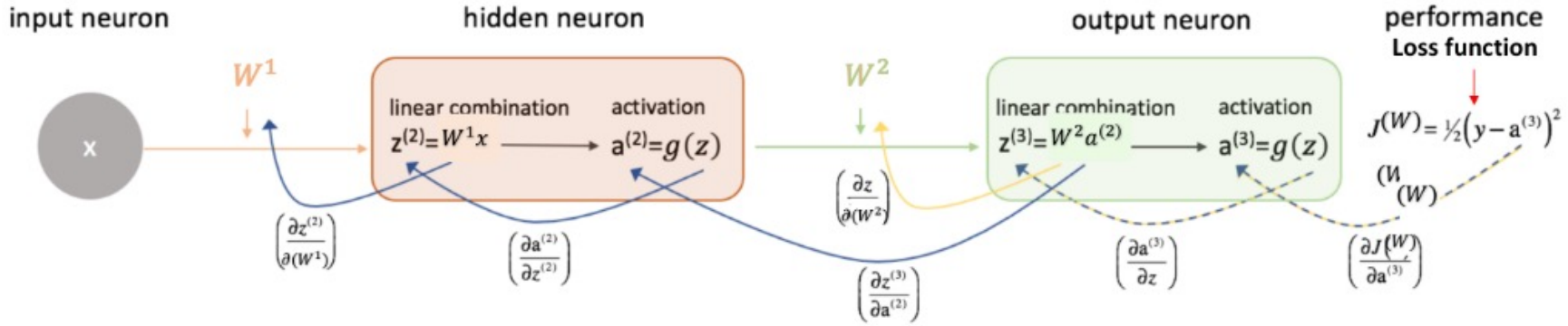
Outline

- Practice in learning neural network
  - Data Preprocessing
  - Weight Initialization
  - Choose the architecture
  - Tune a good learning rate
  - Overfitting and underfitting
  - Regularization (add term to loss and dropout)
  - Stochastic gradient descent (SGD)
  - Data augmentation

Training neural network



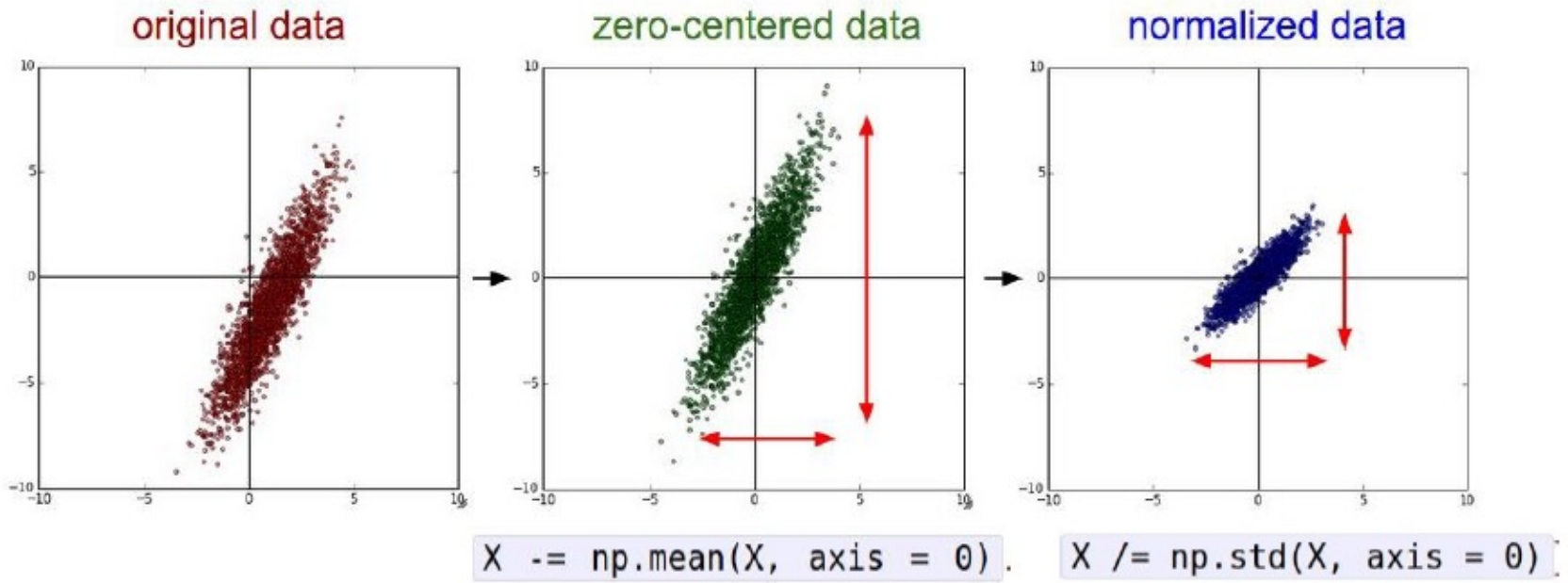
Forward Propagation



Backward Propagation

T1. Data pre-processing

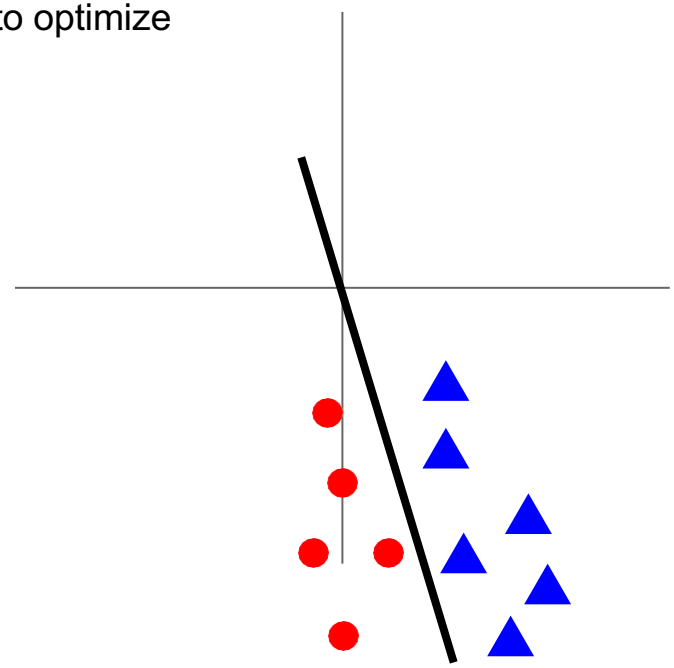
- Closely related to our initialization of weights and assumption on the activation



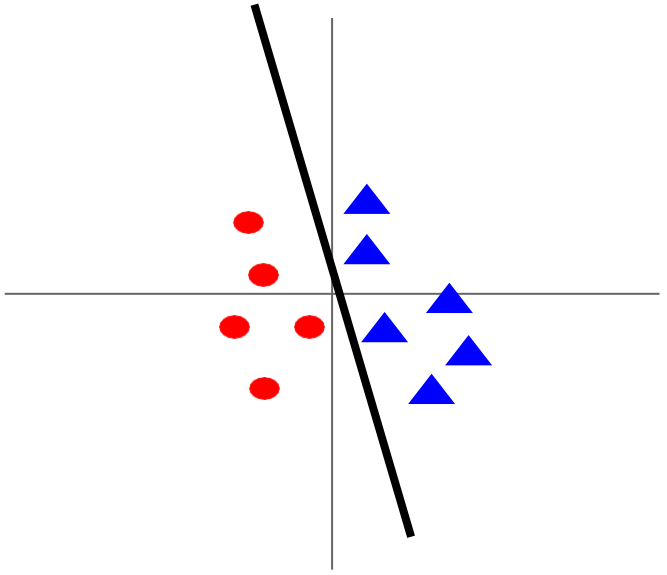
(Assume X [NxD] is data matrix,
each example in a row)

T1. Data pre-processing

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

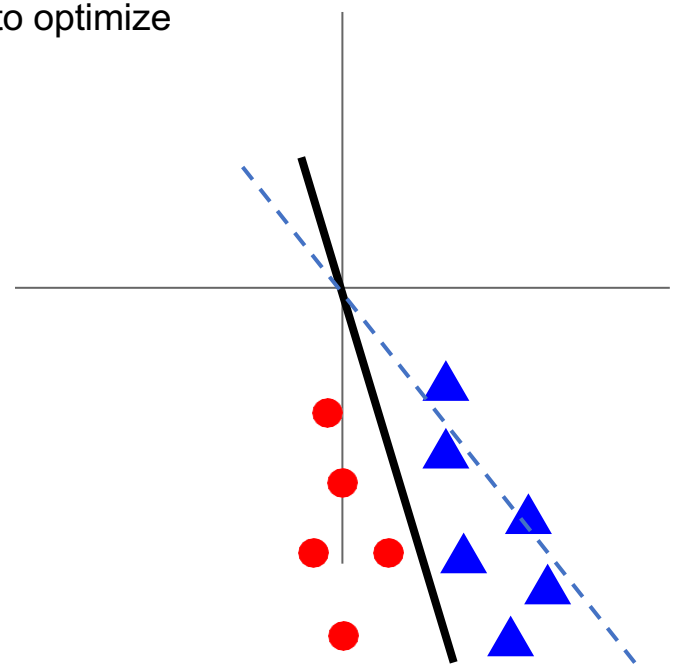


After normalization: less sensitive to small changes in weights; easier to optimize

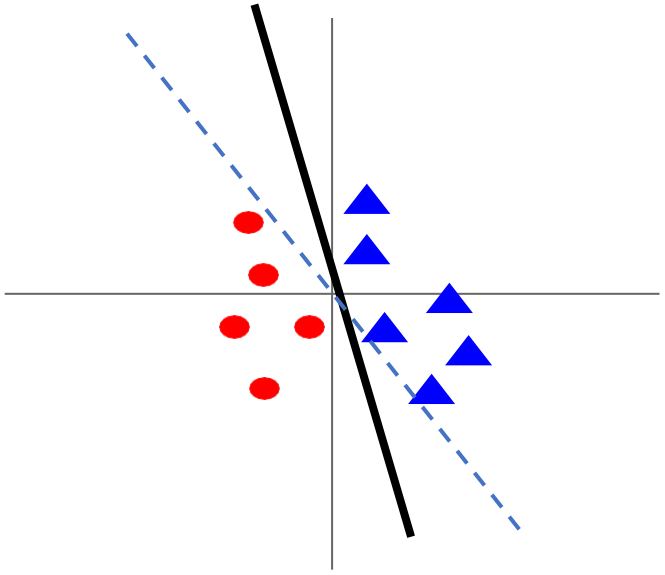


T1. Data pre-processing

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



T1. Data pre-processing

In practice for Images: center only

- e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

T2: Weight Initialization

- - First idea: **Small random numbers**
- (gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but problems with deeper networks.

T2: Weight Initialization

Lets look at some activation statistics

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

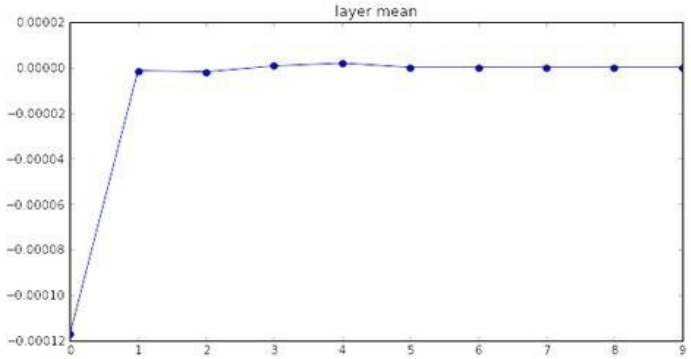
```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

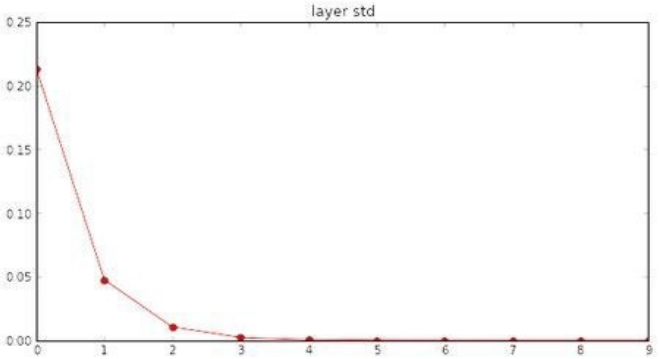
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

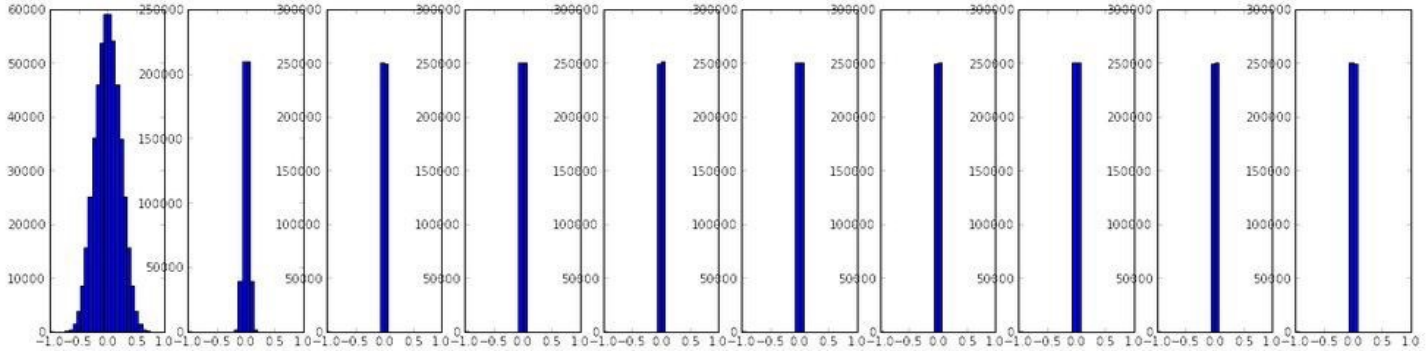
```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

T2: Weight Initialization

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```

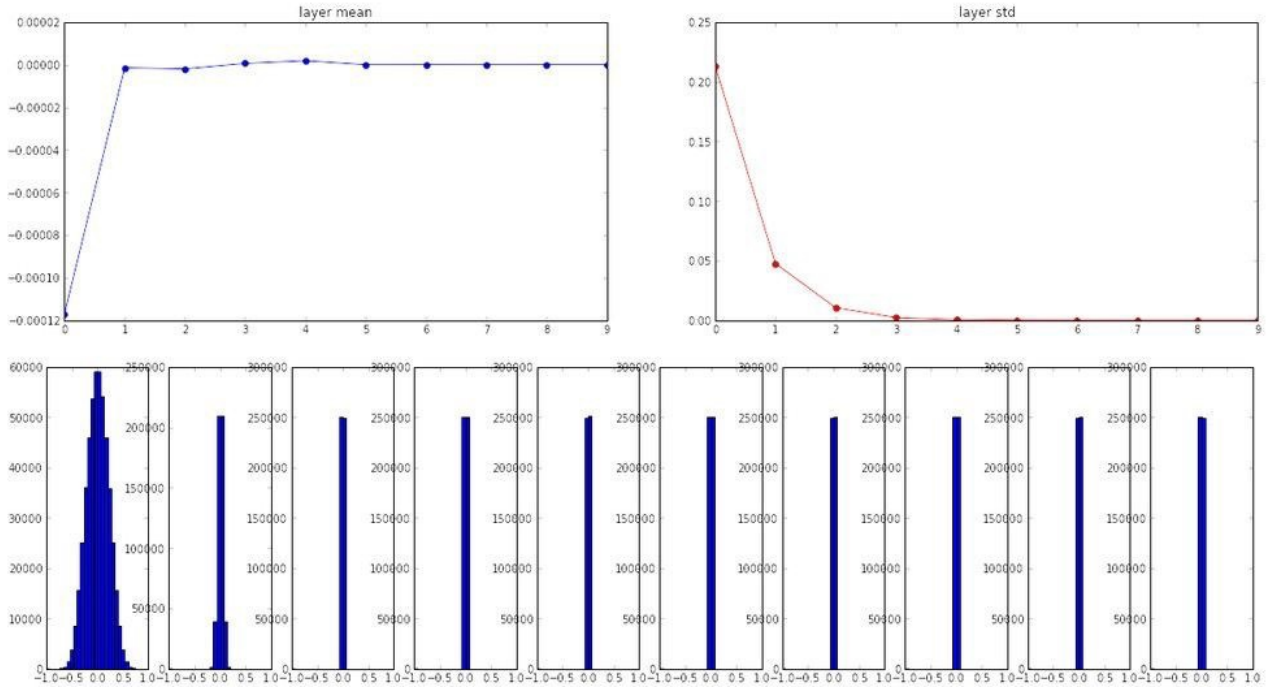






T2: Weight Initialization

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



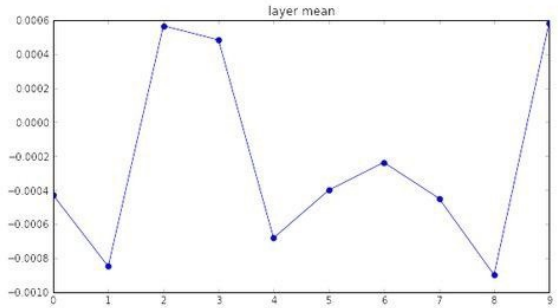
All activations become zero!

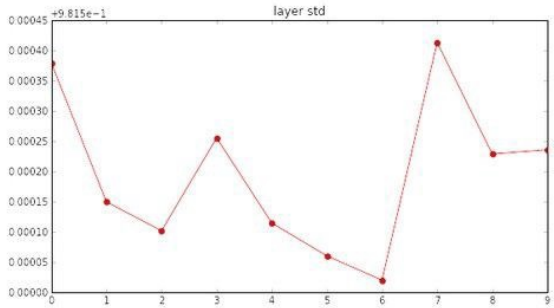
T2: Weight Initialization

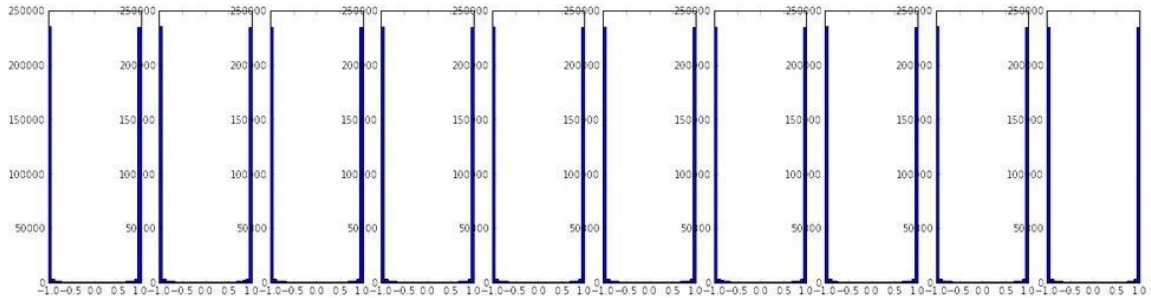
```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

*1.0 instead of *0.01

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```





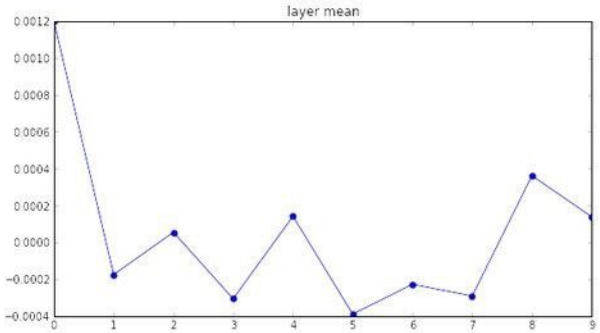


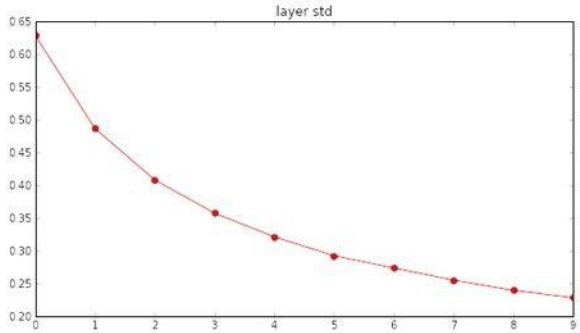
Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

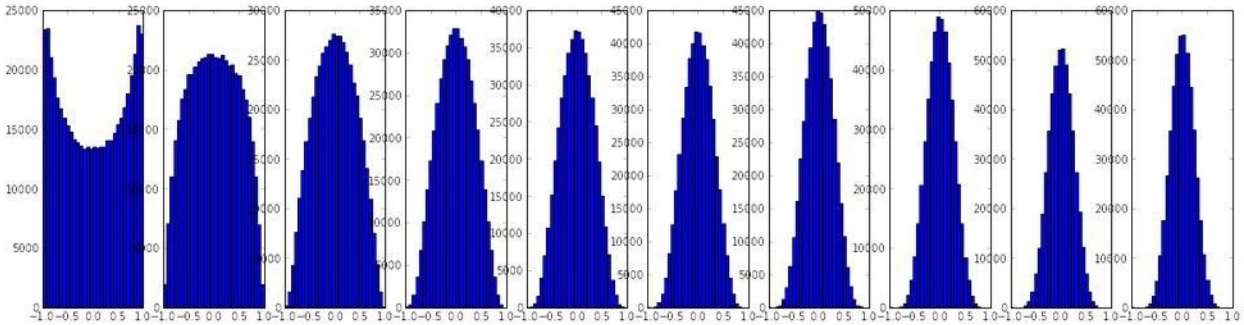
T2: Weight Initialization

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```







"Xavier initialization"
[Glorot et al., 2010]

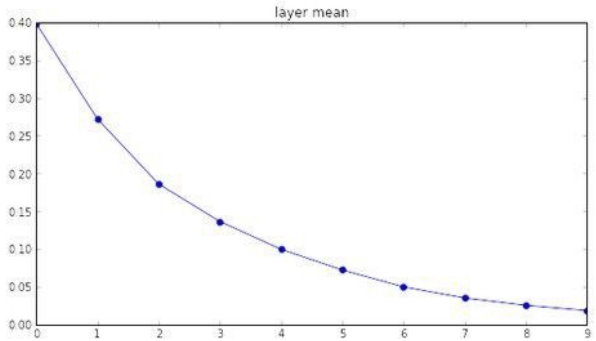
Reasonable initialization.
(Mathematical derivation assumes linear activations)

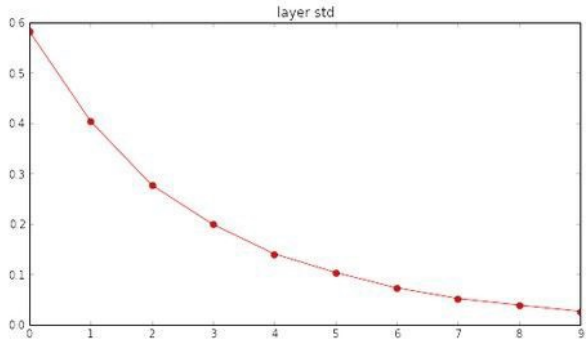
T2: Weight Initialization

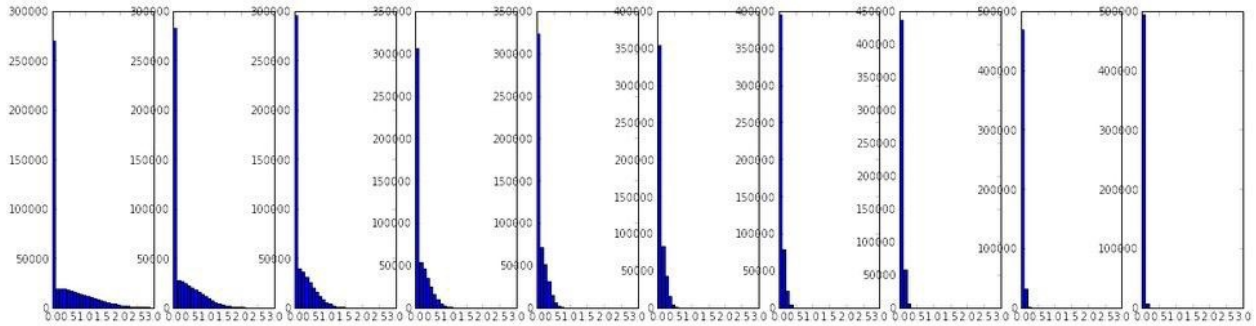
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

but when using the ReLU nonlinearity it breaks.





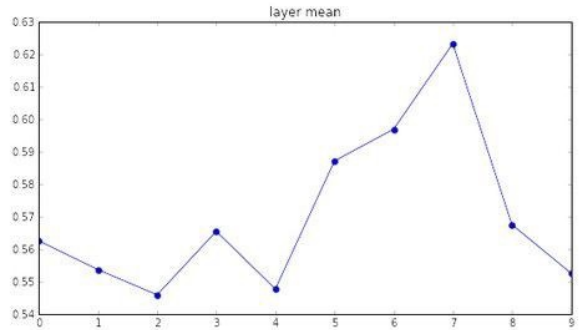


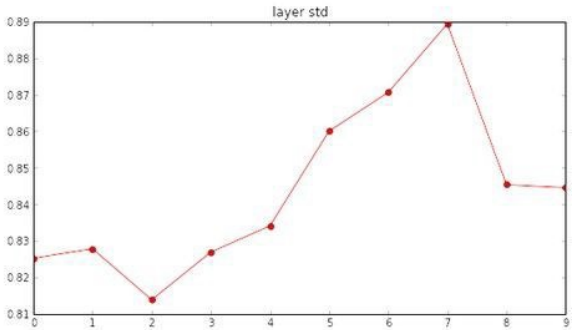
T2: Weight Initialization

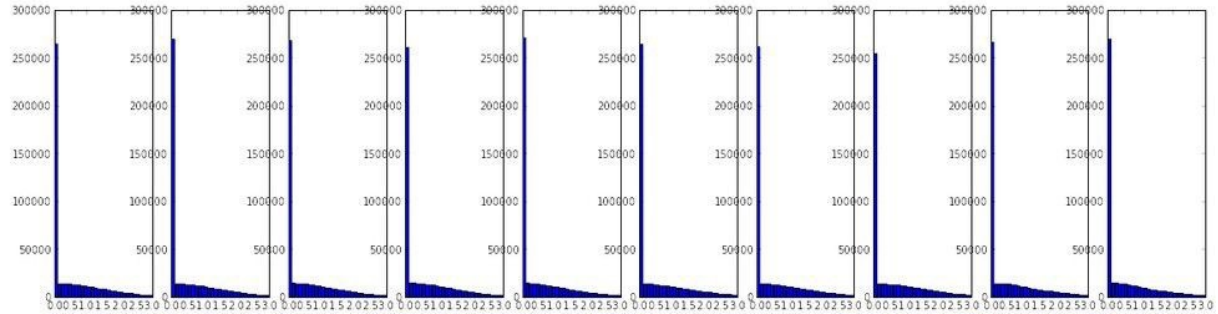
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

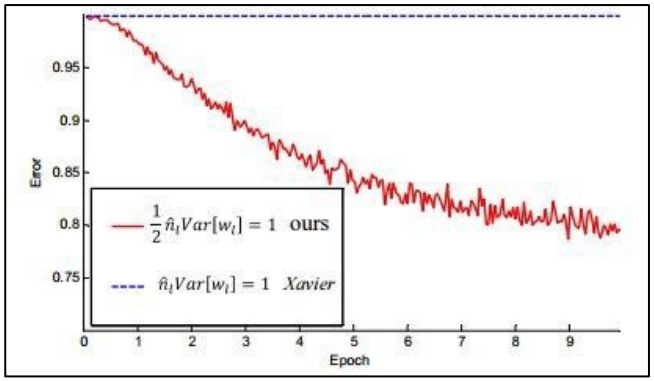
He et al., 2015 (note additional /2)

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```









T2: Weight Initialization

Proper initialization is an active area of research…

Understanding the difficulty of training deep feedforward neural networks by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

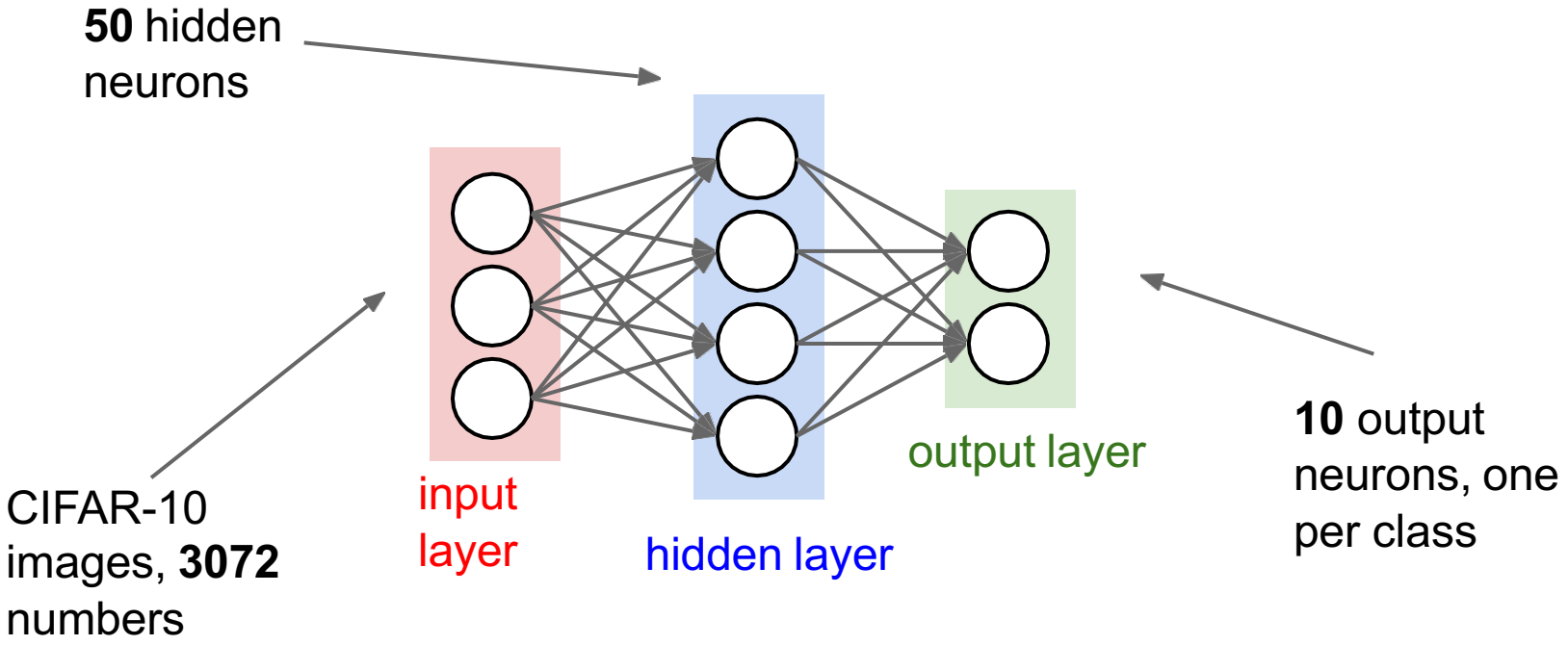
All you need is a good init, Mishkin and Matas, 2015

Learning Feature Pyramids for Human Pose Estimation, Wei Yang , et al. ICCV 2017.

…

T3: Choose the architecture:

say we start with one hidden layer of 50 neurons:



T4: Tune a good learning rate

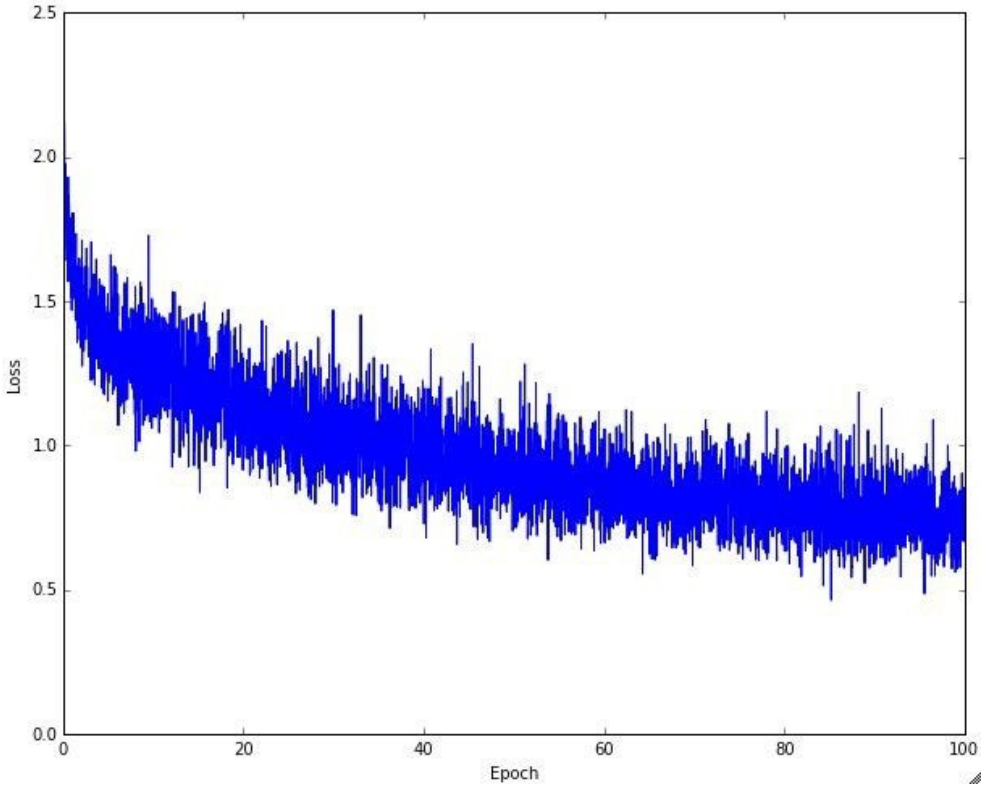
- **Tip**: Make sure that you can overfit very small portion of the training data

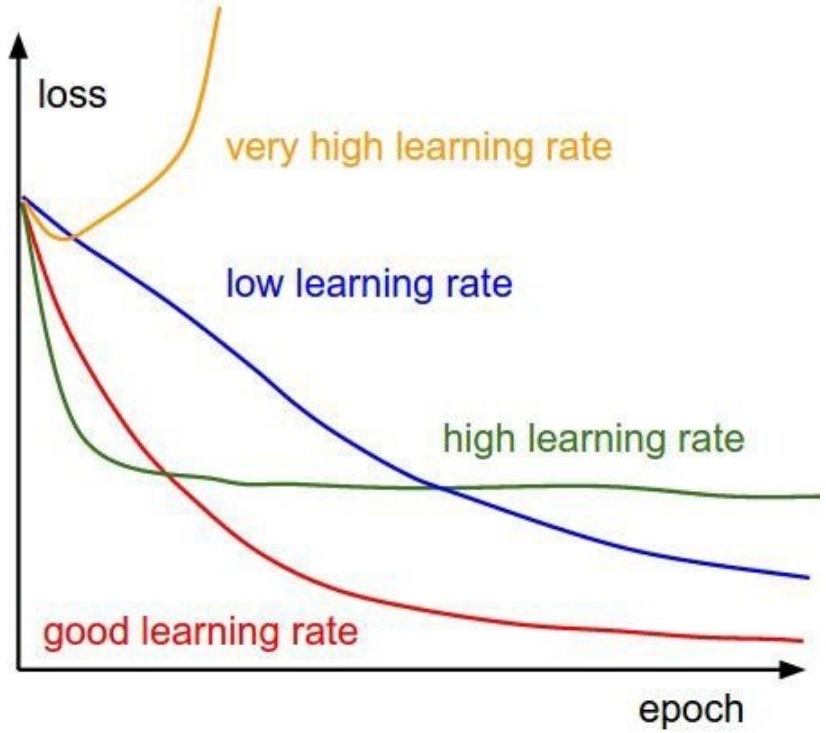
- Start with small regularization and find learning rate that makes the loss go down.

- Check the error on both train and val

T4: Tune a good learning rate

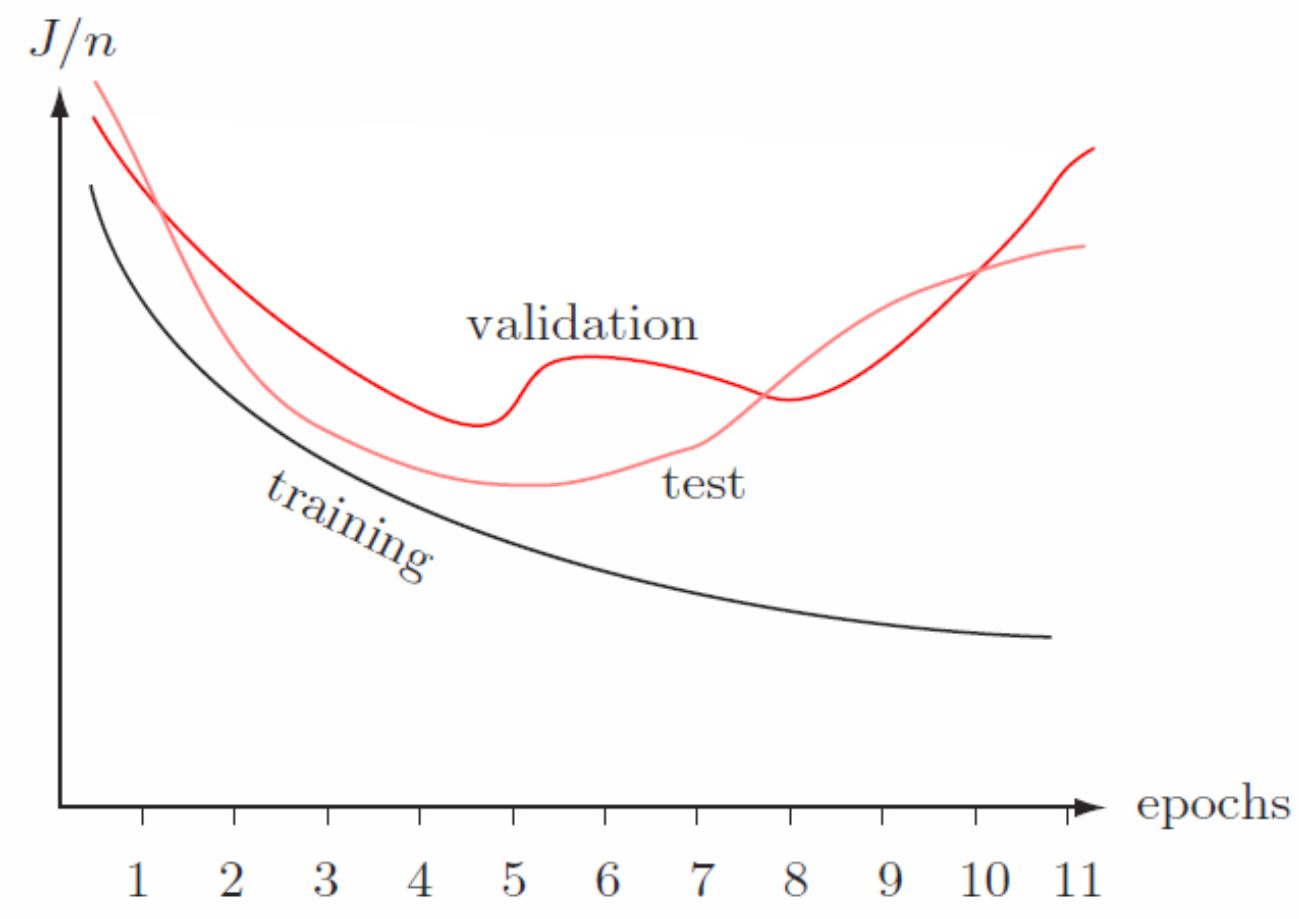
Monitor and visualize the loss curve





T4: Tune a good learning rate

Check the error on both train and val



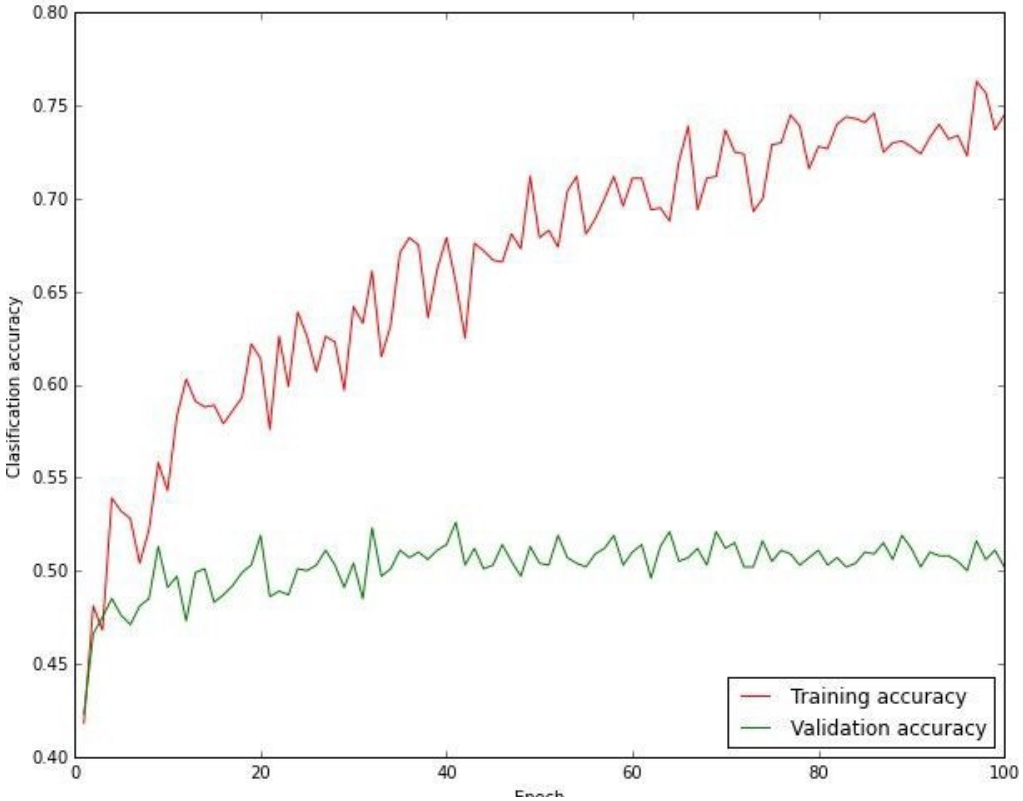
T5: Overfitting and Underfitting

Overfitting

- The learner fits the training data well, but loses the ability to generalize well, i.e. it has small training error but larger generalization error
- A learner with large capacity tends to overfit
  - The family of functions is too large (compared with the size of the training data) and it contains many functions which all fit the training data well.
  - Without sufficient data, the learner cannot distinguish which one is most appropriate and would make an arbitrary choice among these apparently good solutions
  - A separate validation set helps to choose a more appropriate one
  - In most cases, data is contaminated by noise. The learner with large capacity tends to describe random errors or noise instead of the underlying models of data (classes)

T5: Overfitting and Underfitting

Check the error on both train and val



big gap = overfitting

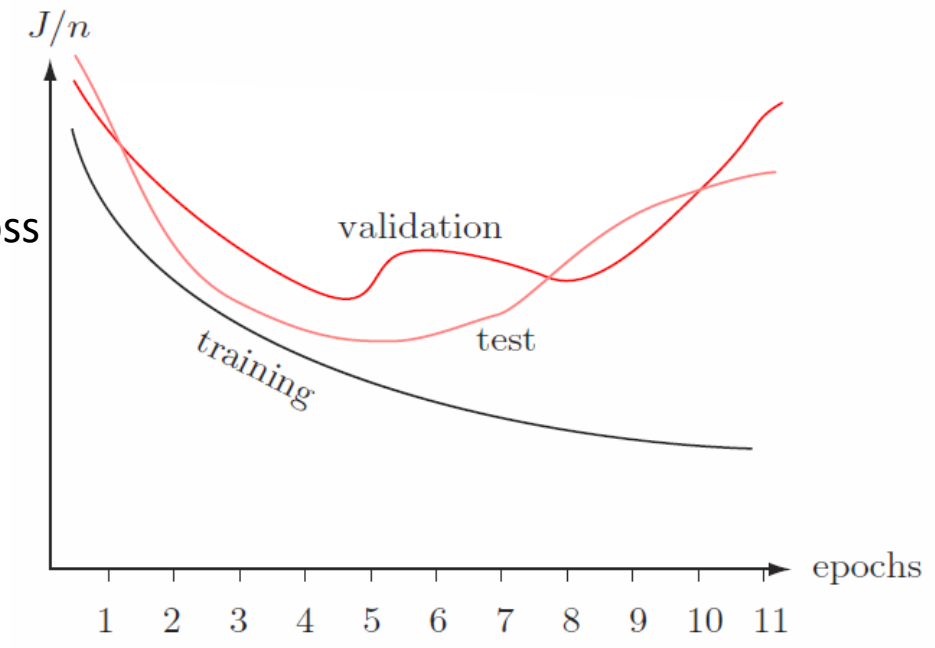
=> increase regularization strength?

no gap

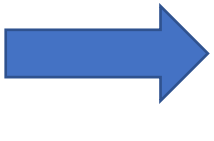
=> increase model capacity?

T5: Overfitting and Underfitting

Examples of overfitting

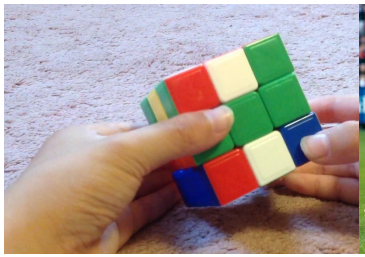


Accuracy



T5: Overfitting and Underfitting

Underfitting


- The learner cannot find a solution that fits training examples well
- Underfitting means the learner cannot capture some important aspects of the data
- Reasons for underfitting happening
  - Model is not rich enough
  - Difficult to find the global optimum of the objective function on the training set or easy to get stuck at local minimum
  - Limitation on the computation resources (not enough training iterations of an iterative optimization procedure)
- Underfitting commonly happens in deep learning with large scale training data and could be even a more serious problem than overfitting in some cases

T6: Regularization: Add term to loss

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

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

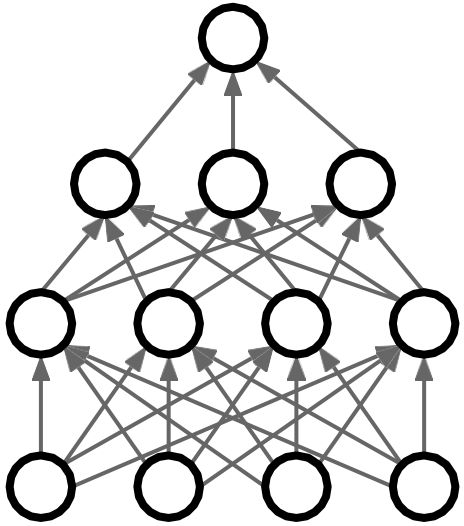
L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

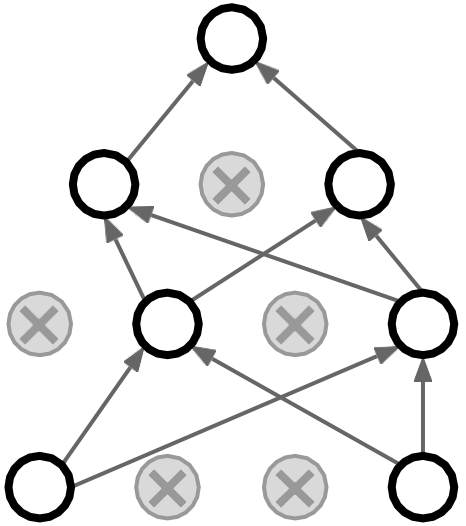
Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

T6: Regularization : Dropout

In each forward pass, randomly set some neurons to zero

Probability of dropping is a hyperparameter; 0.5 is common





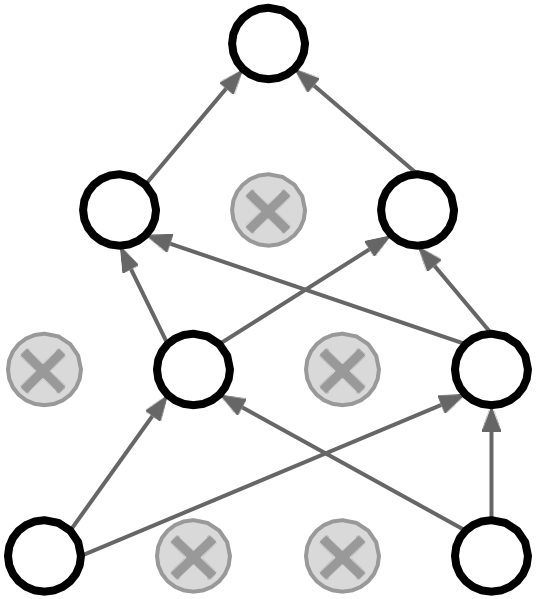
Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

T6: Regularization : Dropout

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

def train_step(X):
  """ X contains the data """

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

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

T6: Regularization : Dropout

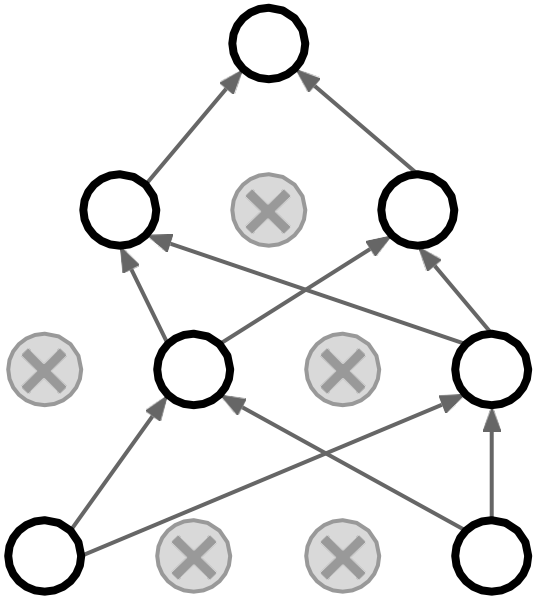
How can this possibly be a good idea?



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

T6: Regularization : Dropout

How can this possibly be a good idea?



Another interpretation:

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

Each binary mask is one model

An FC layer with 4096 units has
$2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test time

Dropout makes our output random!

Output (label) Input (image)

$$y = f_W(x, z)$$

Random mask

Want to "average out" the randomness at test-time

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$

But this integral seems hard …

Dropout: Test time

1. Scaling at the 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

Dropout: Test time

1. Scaling at the test time

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

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

def train_step(X):
  """ X contains the data """

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

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

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
```

p: the probability **not** to drop

drop in forward pass

scale at test time

p***a*** + (1-p)*0 = p*a

Dropout: Test time

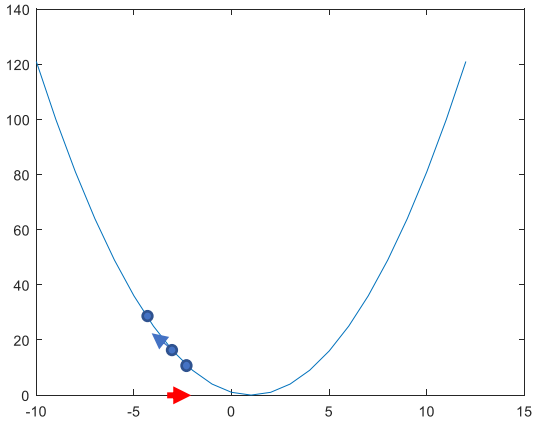
2. More common: “Inverted dropout”

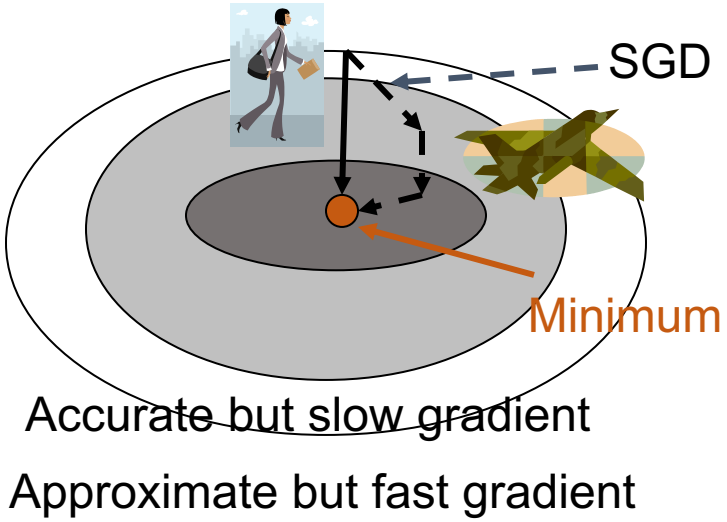
```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

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):
  # ensembled 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
```

T7: Stochastic Gradient Descent (SGD)

- $J(\mathbf{w})=\sum_i(y_i- p_i)^2$, $i = 1, \ldots, N$
- Compute gradient using all samples is too slow if N is too large.
- Each time, a small number of samples is used for computing gradient (Stochastic gradient descent)

$$w_t = w_{t-1}-\alpha J(w_{t-1})'$$

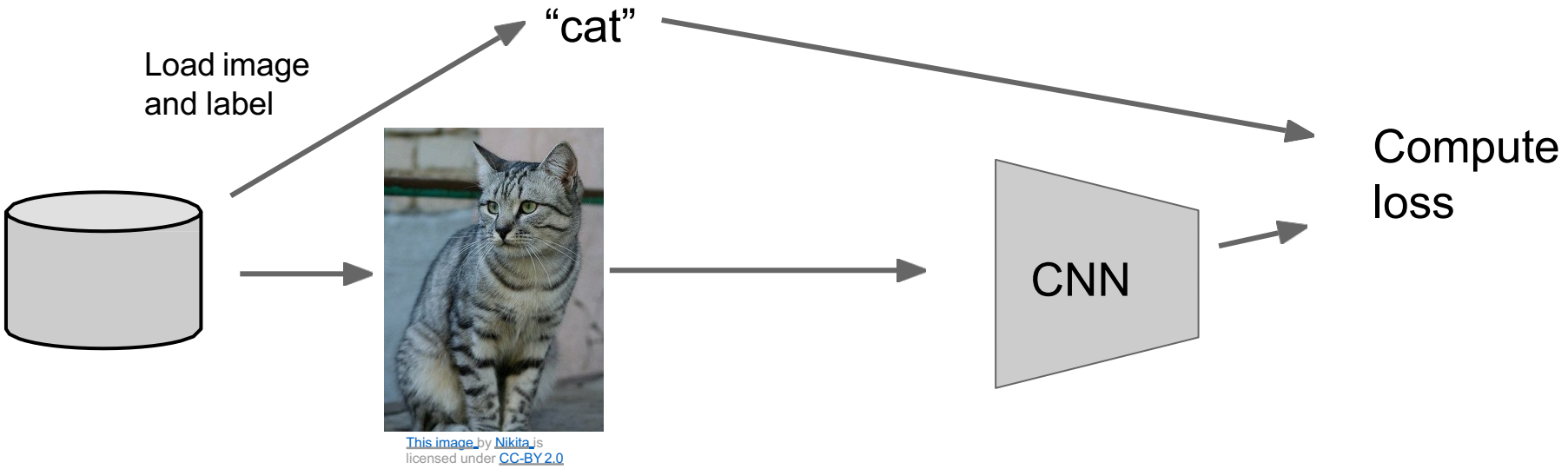




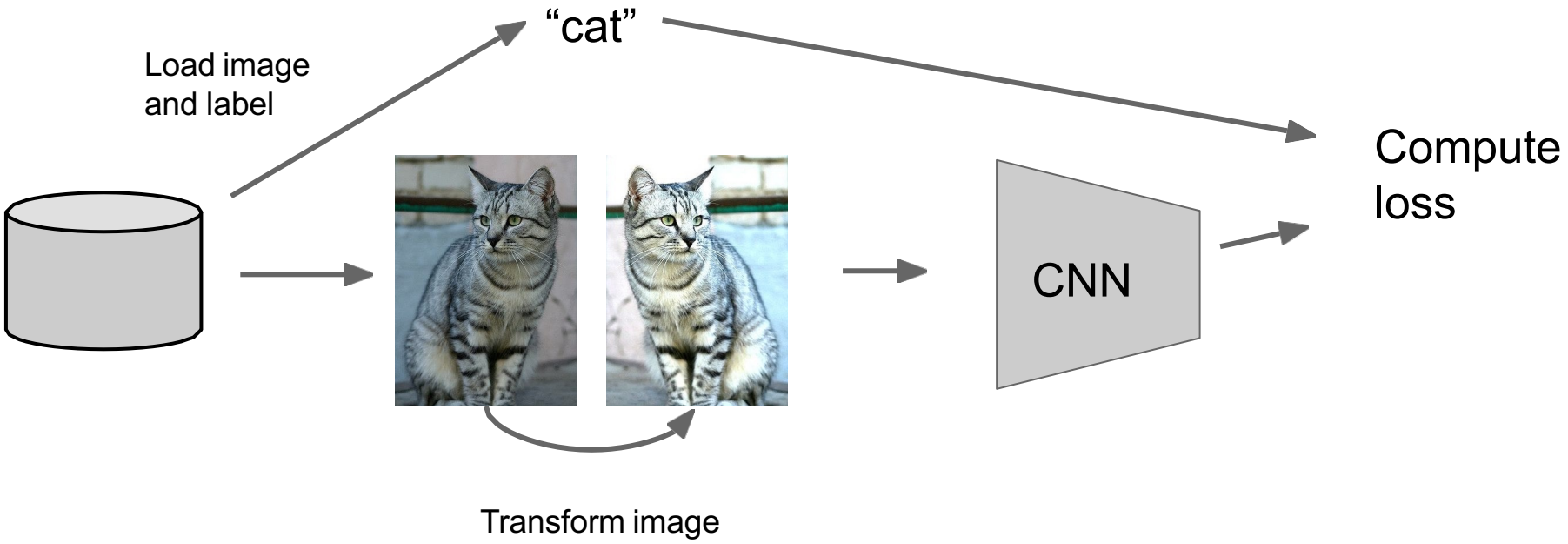
T7: Mini-batch based SGD

- Divide the training set into mini-batches.
- In each epoch, randomly permute mini-batches and take a mini-batch sequentially to approximate the gradient
  - One epoch corresponds to a single presentation of all patterns in the training set
- The estimated gradient at each iteration is more reliable

T8: Data Augmentation

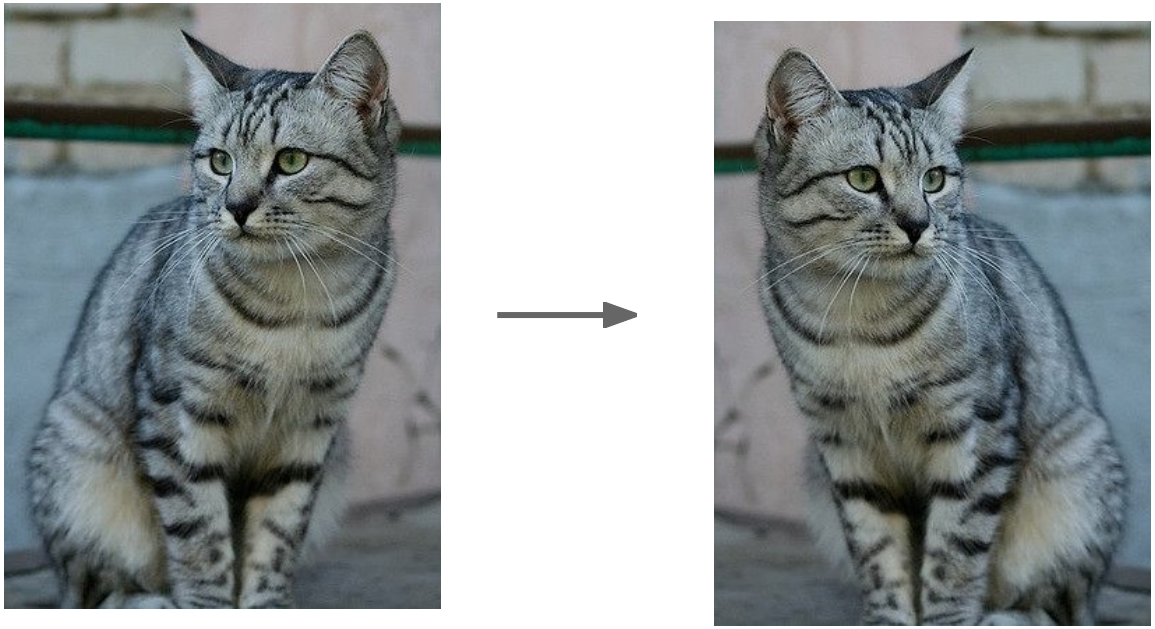


T8: Data Augmentation



T8: Data Augmentation

Horizontal Flips



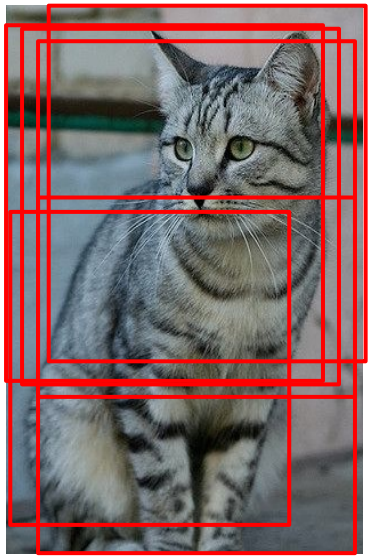
T8: Data Augmentation

Random crops and scales

Training: sample random crops / scales

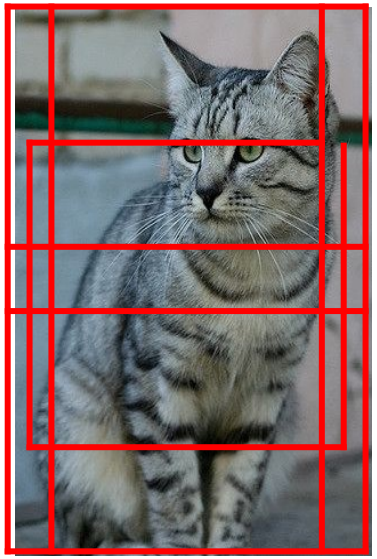
ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



T8: Data Augmentation

Random crops and scales



Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

Testing: average a fixed set of crops

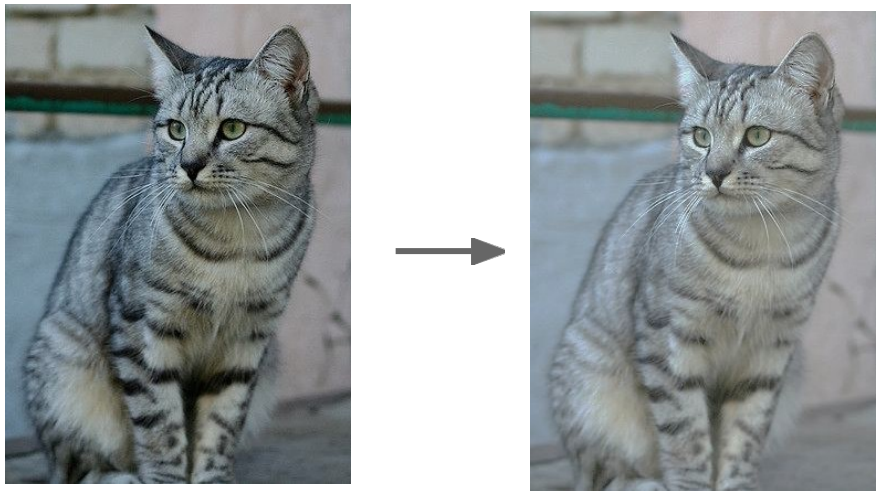
ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

T8: Data Augmentation

Color Jitter

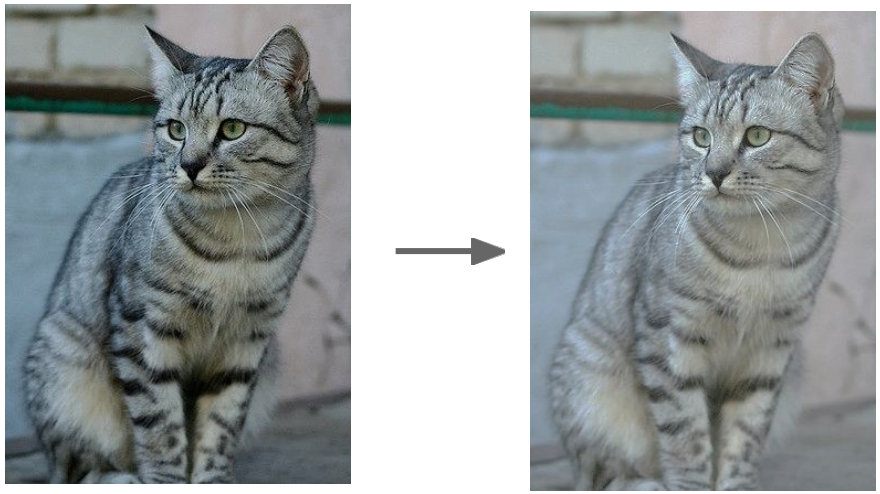
Simple: Randomize
contrast and brightness



T8: Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness



More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012],* ResNet, etc)

T8: Data Augmentation

Takeaway

- Simple to implement, use it
- Especially useful for small datasets
- Fits into framework of noise / marginalization

T8: Data Augmentation

Get creative for your problem!

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

Tune your own model

- How many channels?
- Kernel size?
- How many layers?
- Which block is better?
- Find some clue from existing architectures
- Try your own.
  - Always good to report them in a systematically and **analyse** them.
- Discuss

Deep model generalization:

Domain Adaption in Transfer Learning

Transfer Learning

Traditional Machine Learning:

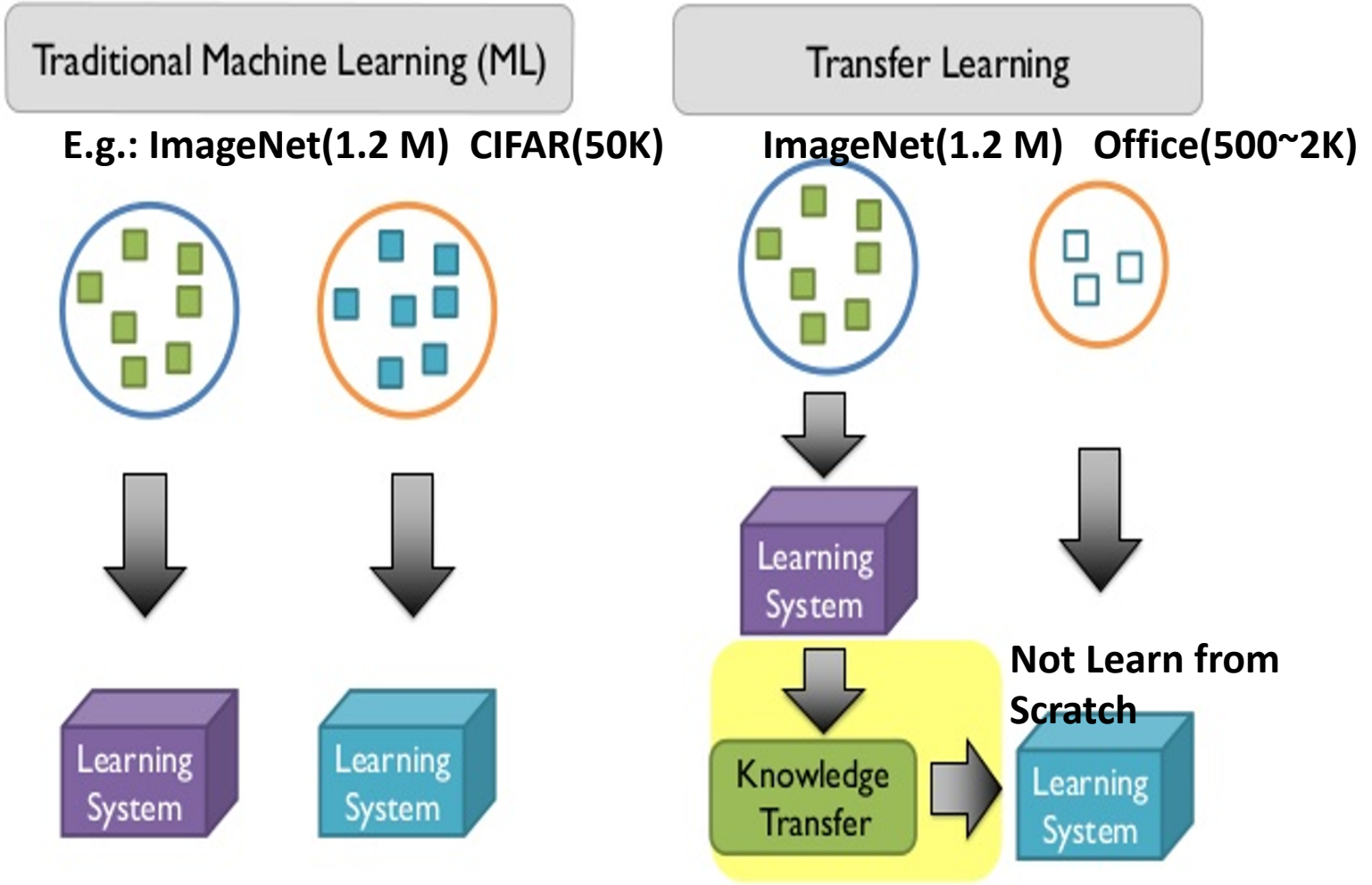
Different dataset need **different models**. Requires **huge dataset**.

Reason: Training a good model requires large number of parameters.

Transfer Learning:

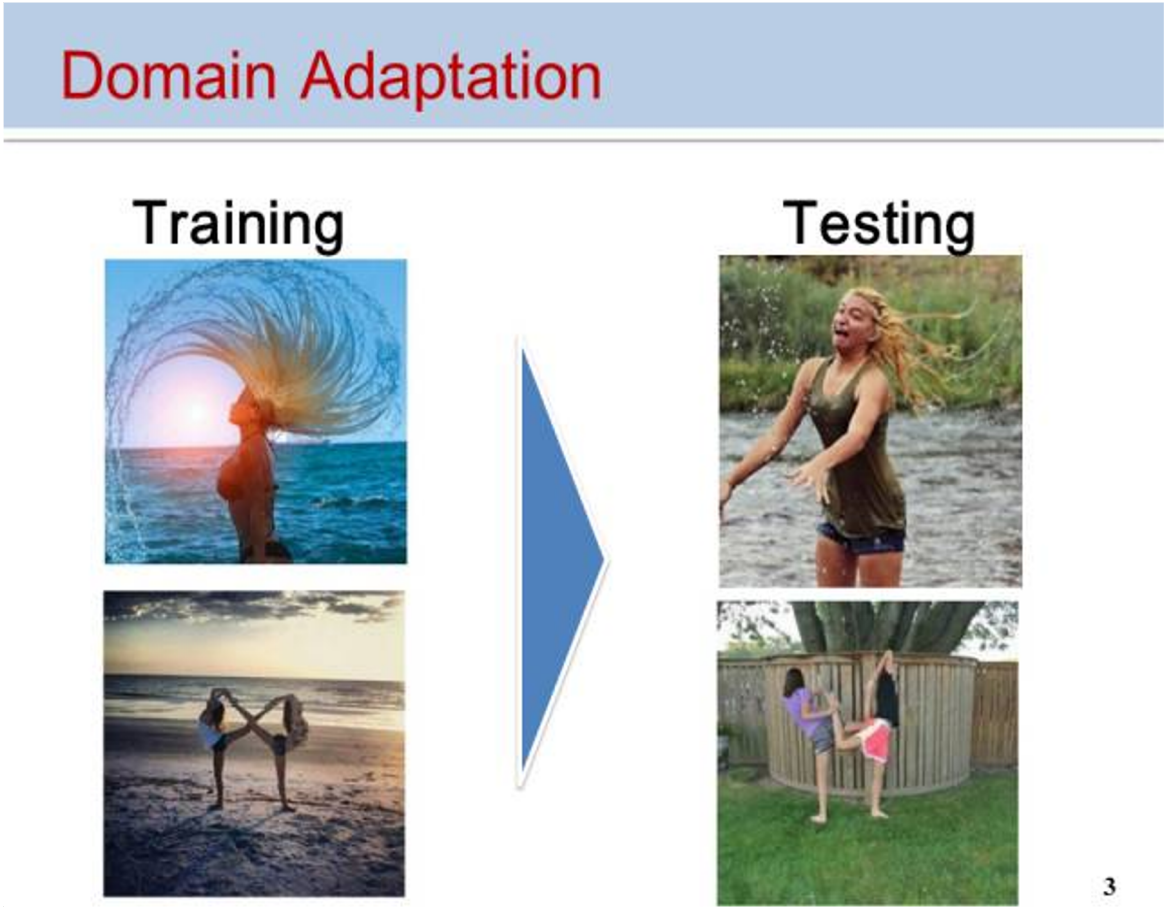
A trend in Deep Learning, which utilises the parameters in trained model, to **transfer** them to new models that **assist training** other datasets.

Transfer Learning

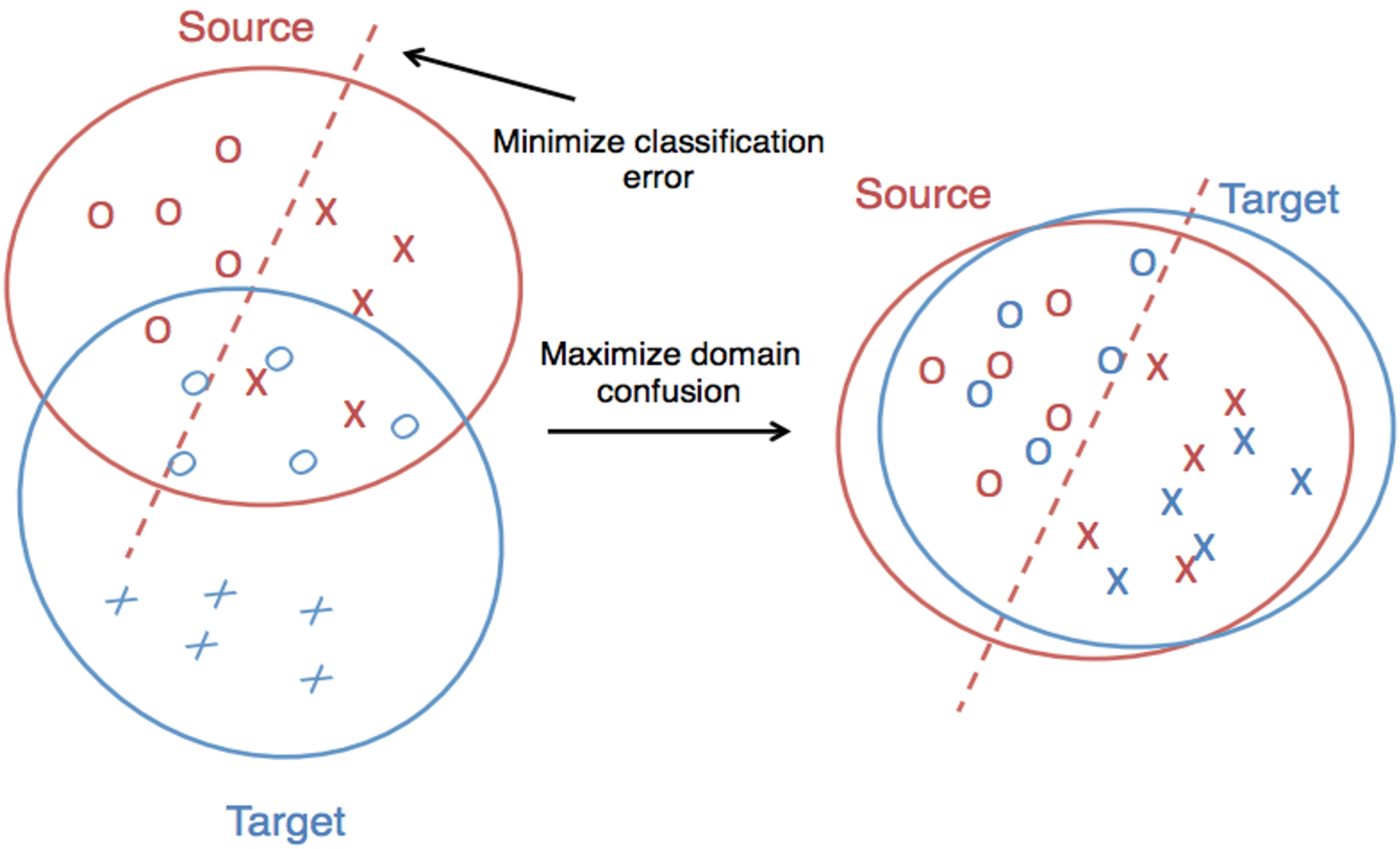


Domain Adaptation

Same feature space and different Probability Distribution

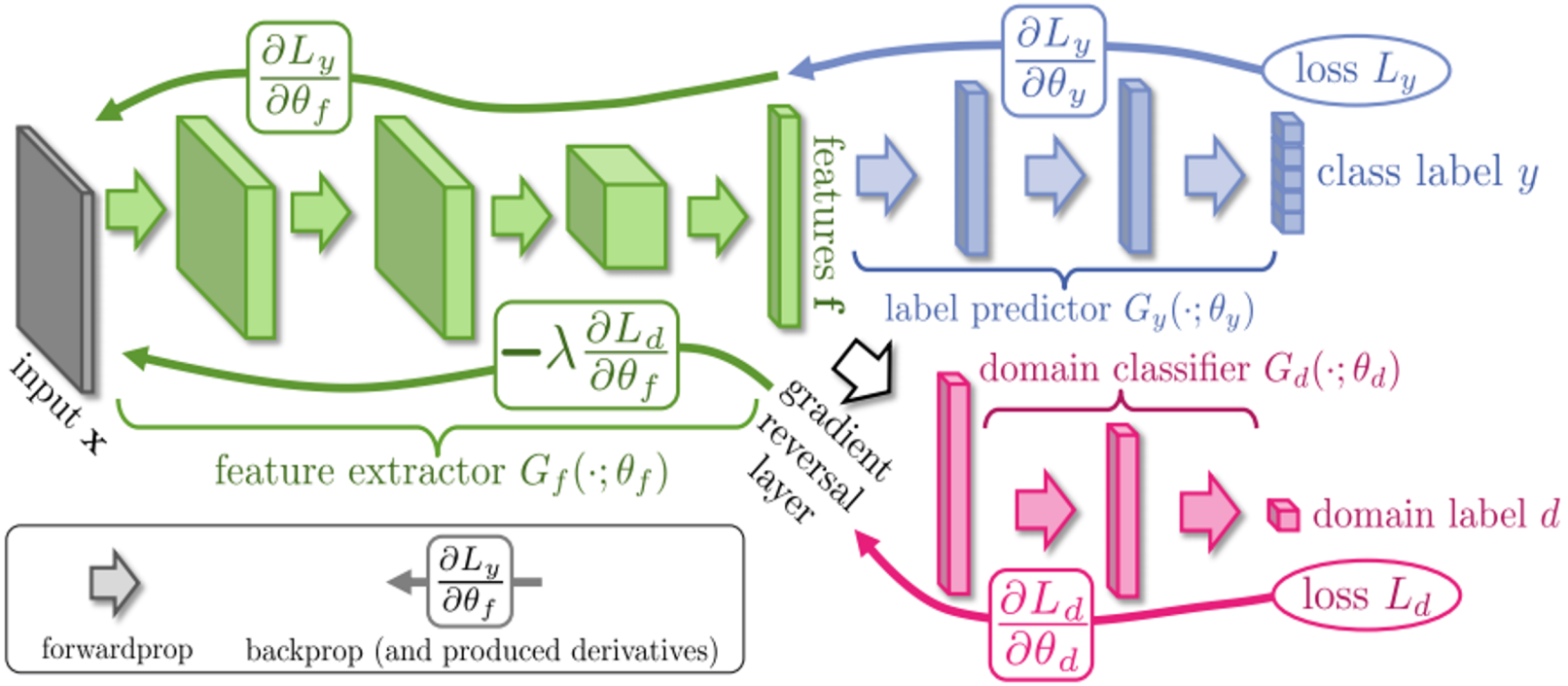


Domain Adaptation



Domain Adversarial Neural Net (DANN)

Yaroslav Ganin, Victor Lempitsky (Skoltech, RU) … (CA)



Paper: Domain Adversarial Training of Neural Networks, JMLR, 2016
Code: https://github.com/fungtion/DANN

Domain Adversarial Neural Net (DANN)

Yaroslav Ganin, Victor Lempitsky (Skoltech, RU) … (CA)

- Suggest that for effective domain transfer to be achieved, predictions must be made based on features that cannot discriminate between the training (source) and test (target) domains.

- Two joint classification tasks to learn features

i) discriminative for the main learning task on the source domain (source image classification)

(ii) indiscriminate with respect to the shift between the domains (domain classification)

- Achieved by augmenting the model by CNN layers + gradient reversal layer

Domain Adaptation

MNIST → MNIST-M: top feature extractor layer

(a) Non-adapted (b) Adapted

SYN NUMBERS → SVHN: last hidden layer of the label predictor

(a) Non-adapted (b) Adapted

Domain Adaptation

Method (Source / Target)	Amazon / Webcam	DSLR / Webcam	Webcam / DSLR
GFK(PLS, PCA) (Gong et al., 2012)	.197	.497	.6631
SA* (Fernando et al., 2013)	.450	.648	.699
DLID (Chopra et al., 2013)	.519	.782	.899
DDC (Tzeng et al., 2014)	.618	.950	.985
DAN (Long and Wang, 2015)	.685	.960	.990
Source only	.642	.961	.978
DANN	**.730**	**.964**	**.992**