# Techniques in Training Deep NN

Dr. Yijun Zhao

Fordham University

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Optimizers
- Callbacks
- Batch Normalization
- Dropout
- Babysitting Model Training

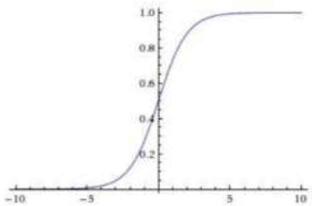
# Activation Functions

# Activation

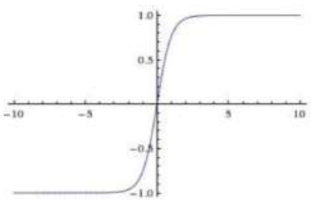
## Activation Functions

**Sigmoid**

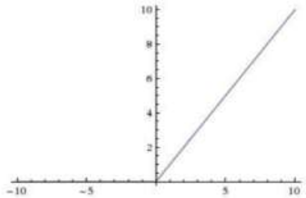
$$\sigma(x) = 1/(1+e^{-x})$$



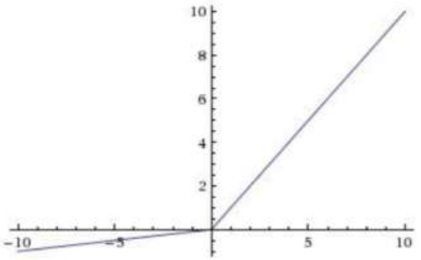
**tanh** tanh(x)



**ReLU** max(0,x)

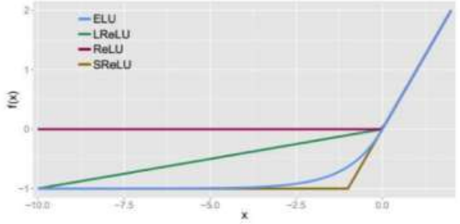


**Leaky ReLU**
max(0.1x, x)



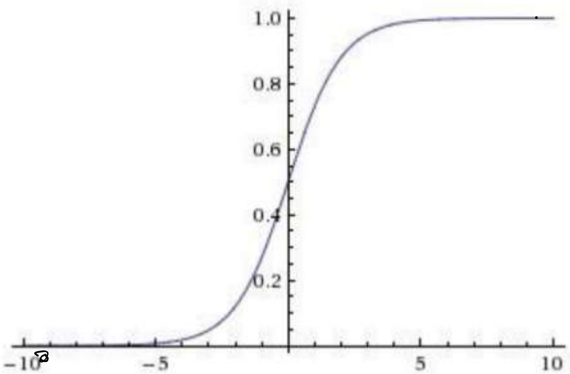
**Maxout** $\max(w_1^T x + b_1, w_2^T x + b_2)$

**ELU** $$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$



# Activation

## Activation Functions




**Sigmoid**

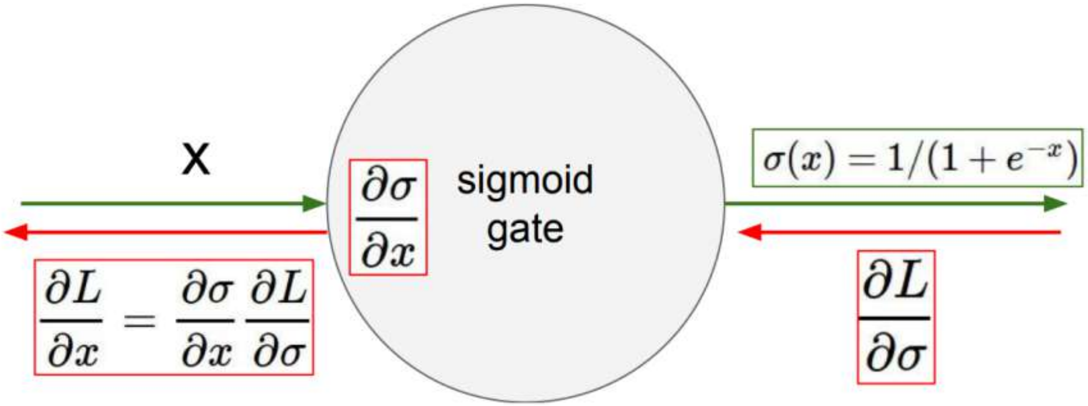
$$\sigma(x) = 1/(1+e^{-x})$$

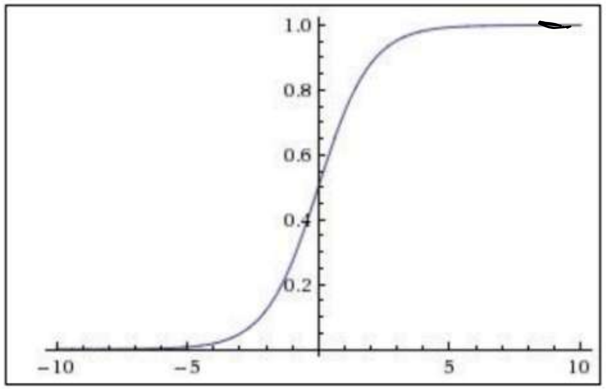
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients

# Activation



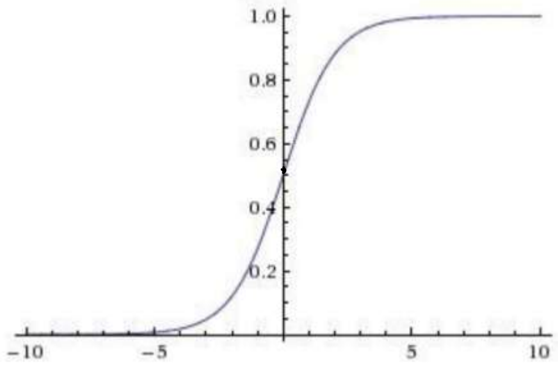




What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

# Activation

## Activation Functions

$$\sigma(x) = 1/(1 + e^{-x})$$




**Sigmoid**

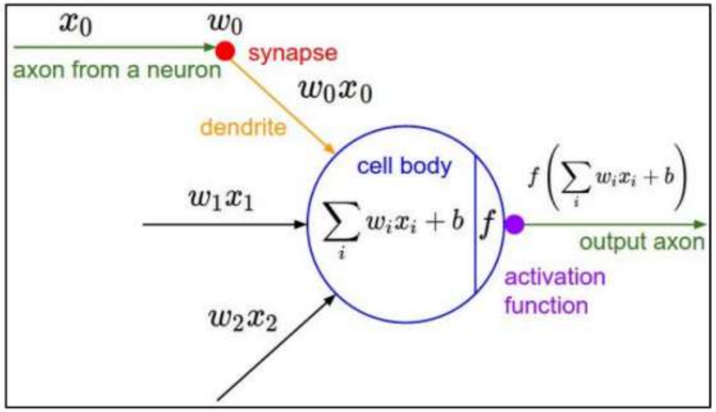
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered

# Activation

Consider what happens when the input to a neuron (x) is always positive:



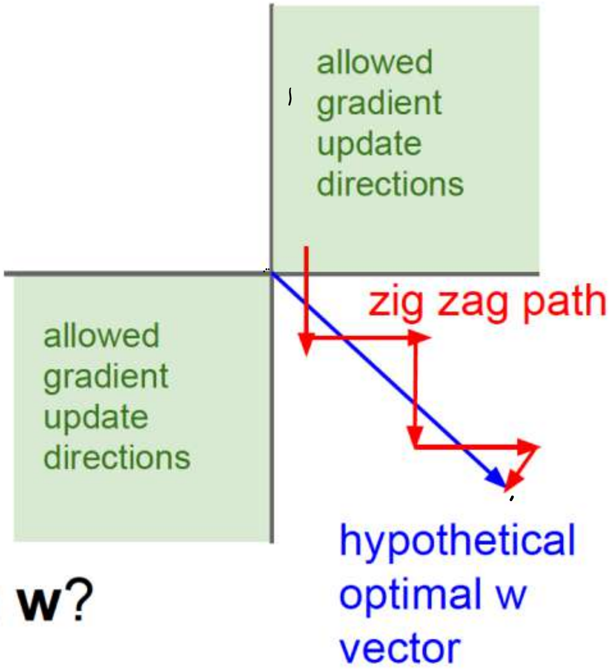

$$f\left(\sum_i w_i x_i + b\right)$$

What can we say about the gradients on **w**?

# Activation

Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$




What can we say about the gradients on **w**?

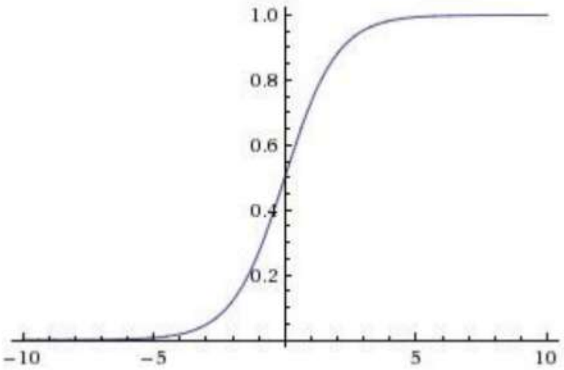
Always all positive or all negative :(

(this is also why you want zero-mean data!)

# Activation

## Activation Functions

$$\sigma(x) = 1/(1 + e^{-x})$$




**Sigmoid**

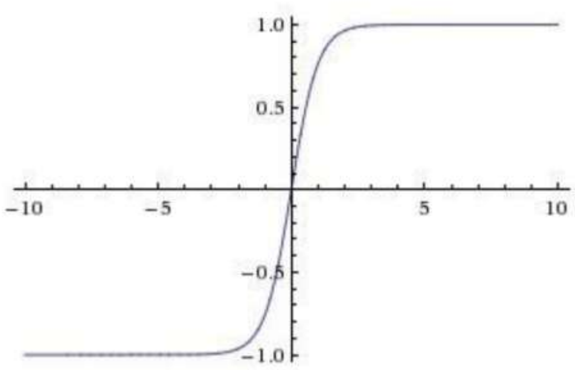
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

# Activation

## Activation Functions



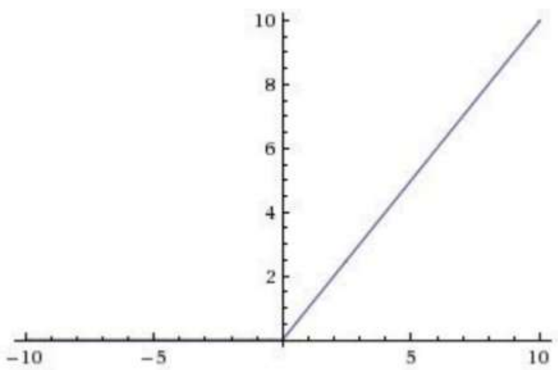

**tanh(x)**

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

# Activation

Activation Functions




**ReLU**
(Rectified Linear Unit)

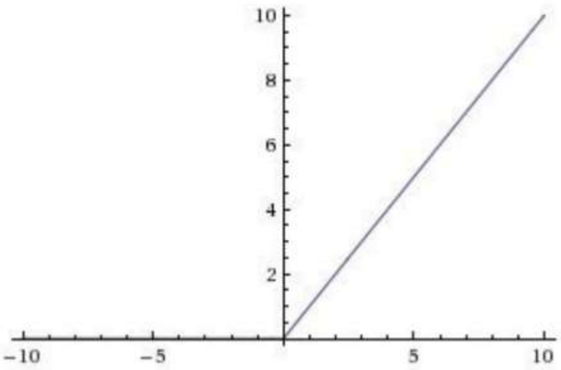
- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

[Krizhevsky et al., 2012]

# Activation

Activation Functions




**ReLU**
(Rectified Linear Unit)

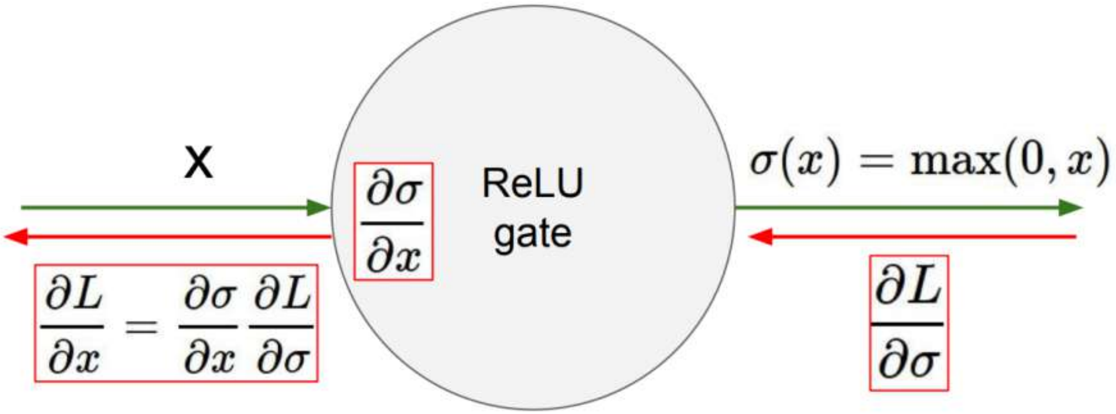
- Computes **f(x) = max(0,x)**

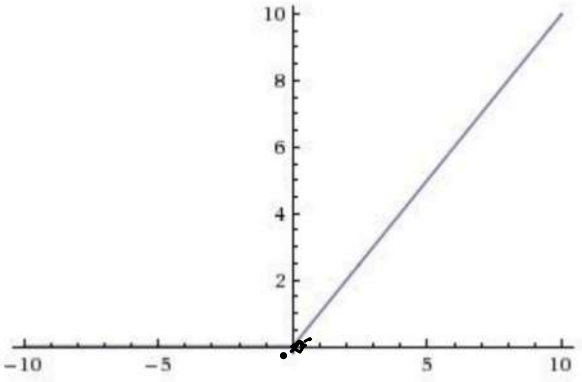
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?

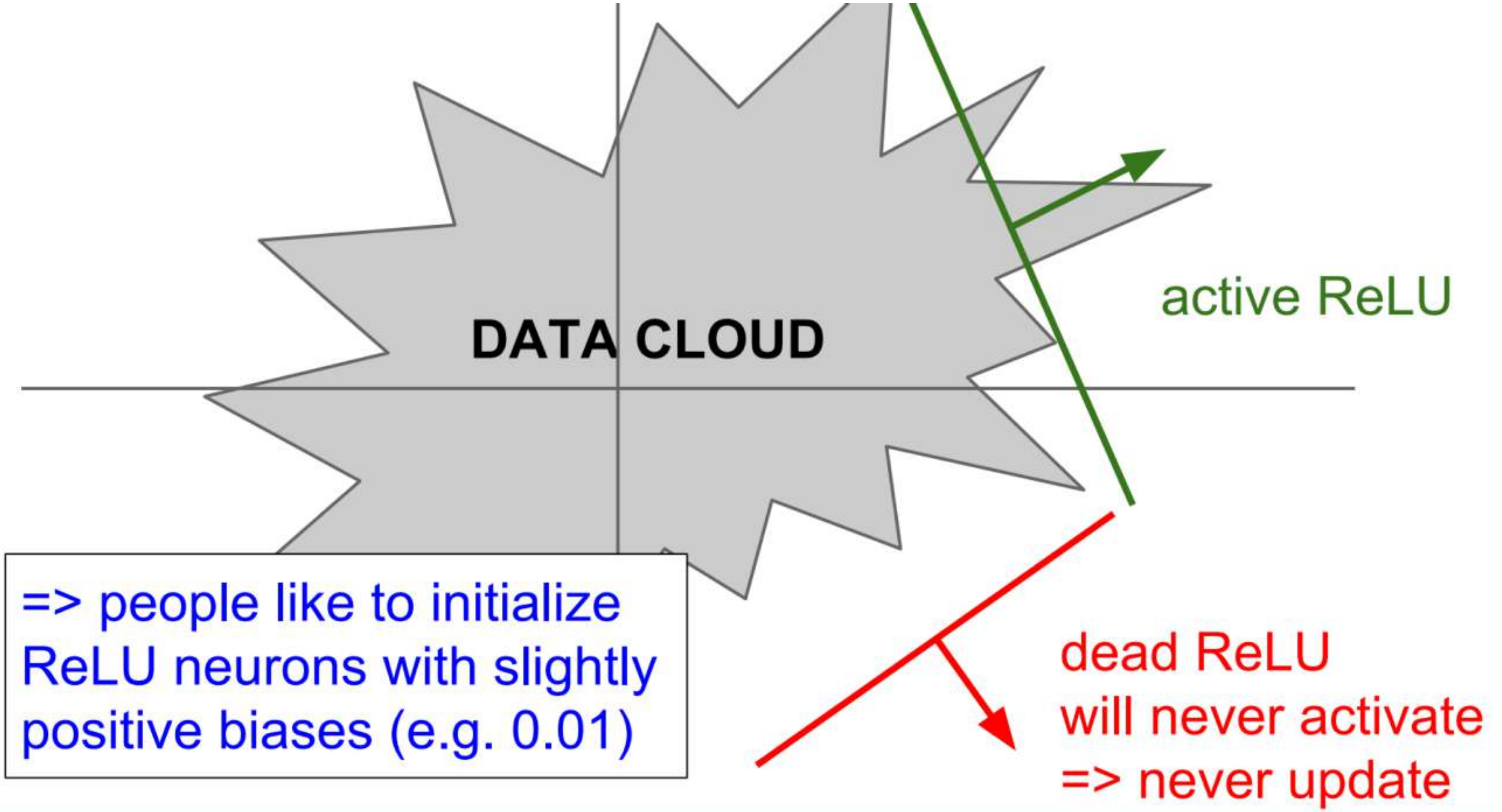
# Activation




What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

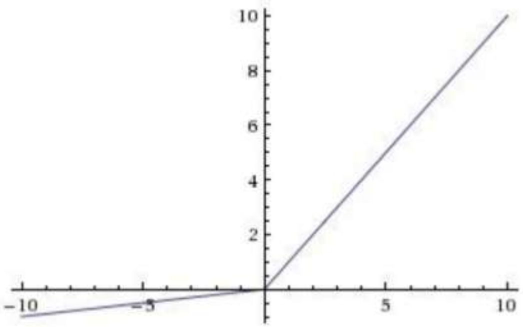
# Activation


DATA CLOUD
active ReLU
=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)
dead ReLU
will never activate
=> never update

# Activation

## Activation Functions

[Mass et al., 2013]
[He et al., 2015]




**Leaky ReLU**

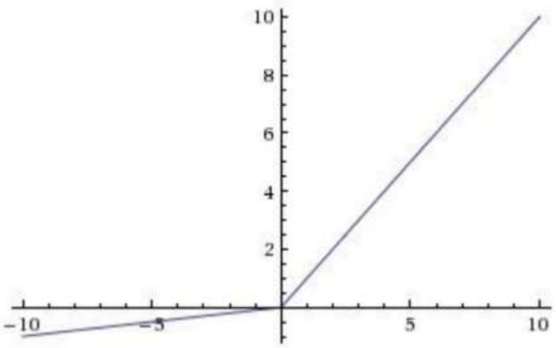
$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

# Activation

## Activation Functions

[Mass et al., 2013]
[He et al., 2015]




- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

**Leaky ReLU**

$$f(x) = \max(0.01x, x)$$

**Parametric Rectifier (PReLU)**

$$f(x) = \max(\alpha x, x)$$

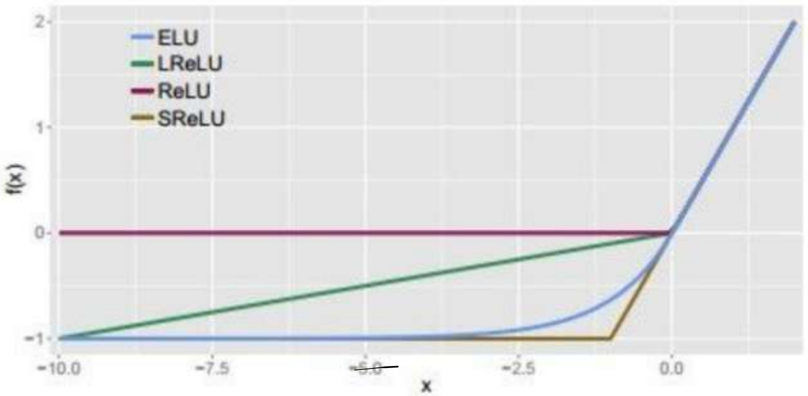
backprop into \alpha
(parameter)

# Activation

## Activation Functions

[Clevert et al., 2015]

### Exponential Linear Units (ELU)




$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\left(\exp(x) - 1\right) & \text{if } x \leq 0 \end{cases}$$

- All benefits of ReLU
- Does not die
- Closer to zero mean outputs

- Computation requires exp()

# Activation

**Maxout "Neuron"** [Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Problem: doubles the number of parameters/neuron :(

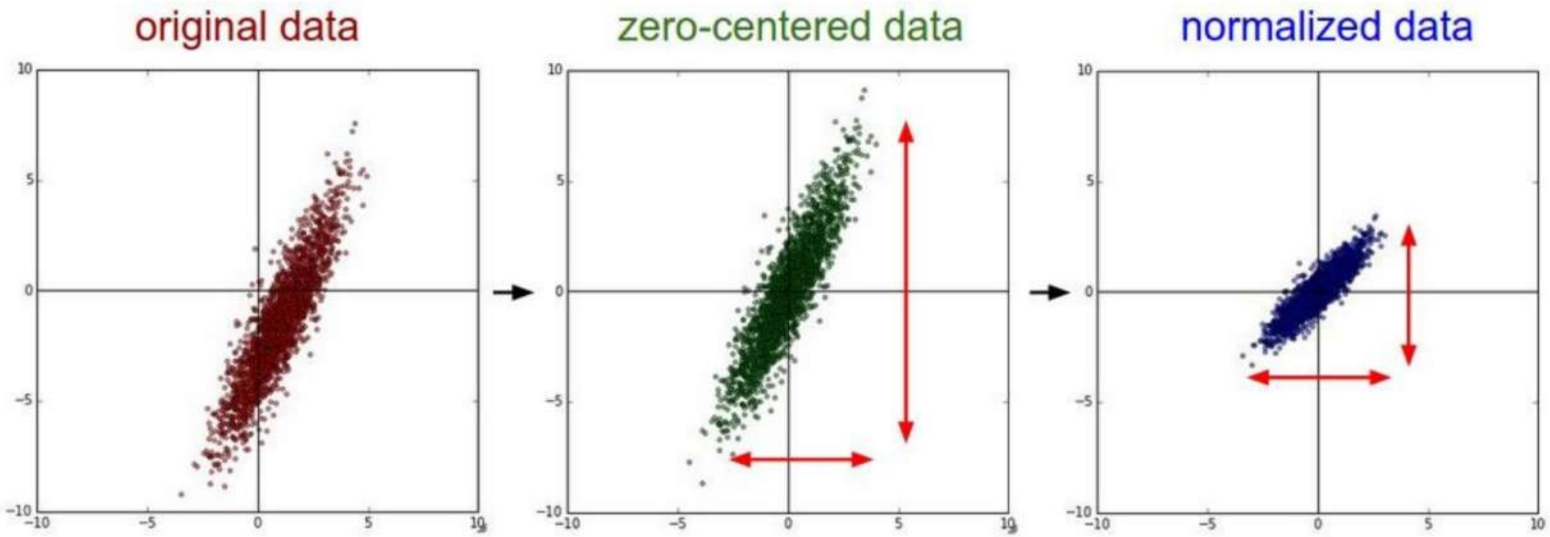
# Activation

**In practice:**

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid

# Data Preprocessing

# Data Preprocessing



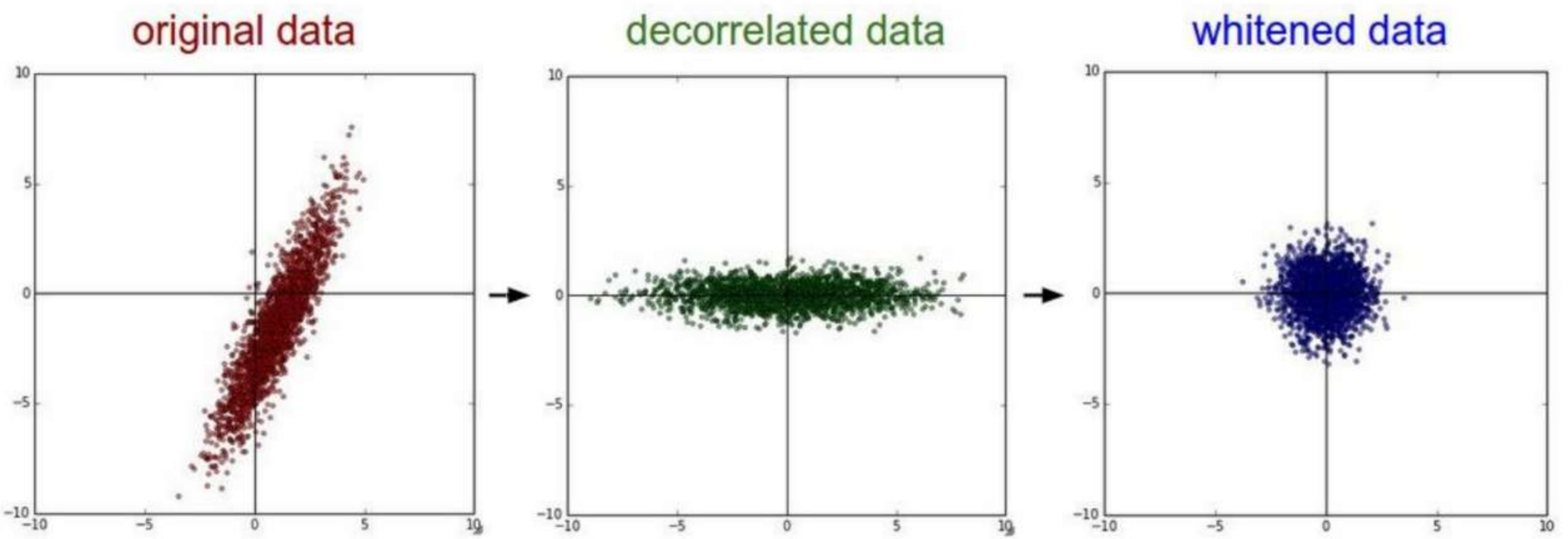

```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

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

# Data Preprocessing

In practice, you may also see **PCA** and **Whitening** of the data




(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

# Data Preprocessing

**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)

Not common to normalize variance, to do PCA or whitening

# Weight Initialization

# Initialization

Basic Methods

- Weights should NOT be initialized to same value because all the gradients will be the same
- Instead, draw from some distribution
- Uniform from [-0.1, 0.1] is a reasonable starting spot for shallow networks
- Biases may need special constant initialization

# 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 can lead to non-homogeneous distributions of activations across the layers of a network.

# 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)
```

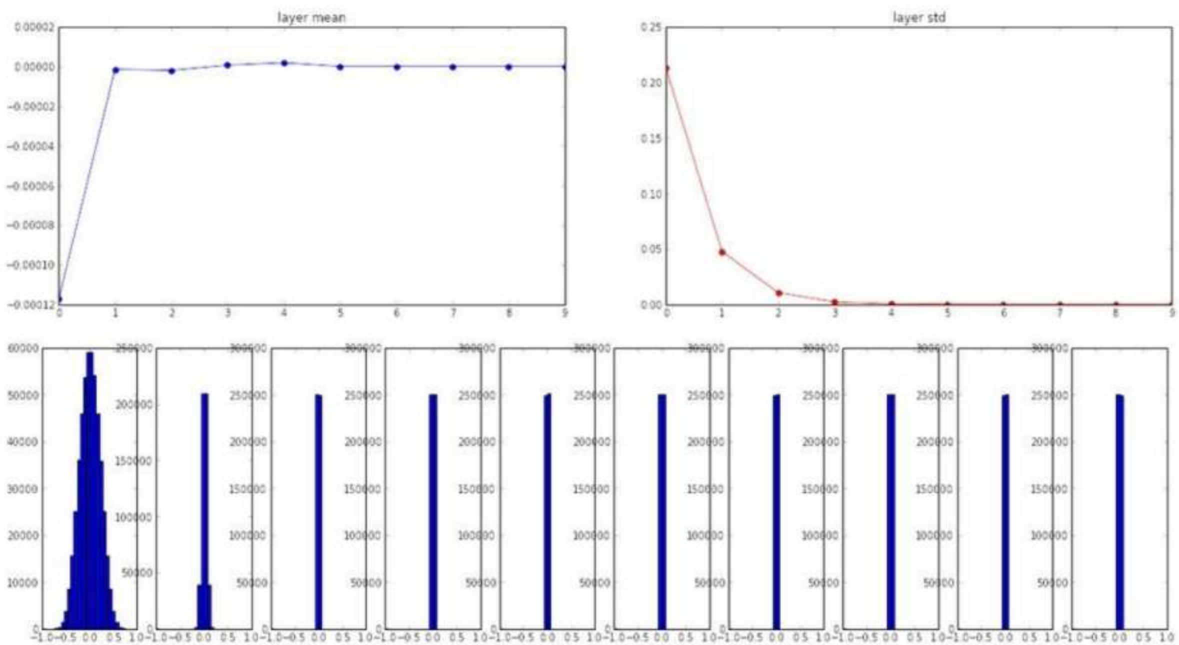
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    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))
```

# Initialization

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

```
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!

Q: think about the backward pass. What do the gradients look like?

Hint: think about backward pass for a W*X gate.

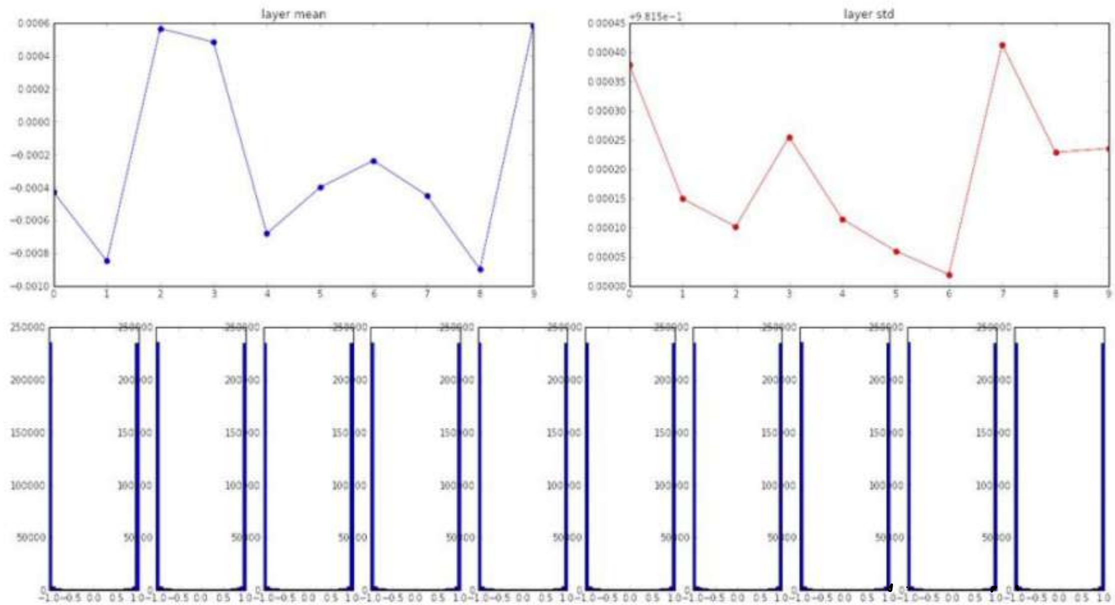
# Initialization

```
W = 1.0*np.random.randn(D, H)
```

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

```
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
```

*1.0 instead of *0.01




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

# Initialization

Smarter methods:

- The *mean* of the activations should be zero.
- The *variance* of the activations should stay the same across every layer.

$$W^{[l]} \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{n^{[l-1]}})$$
$$b^{[l]} = 0$$

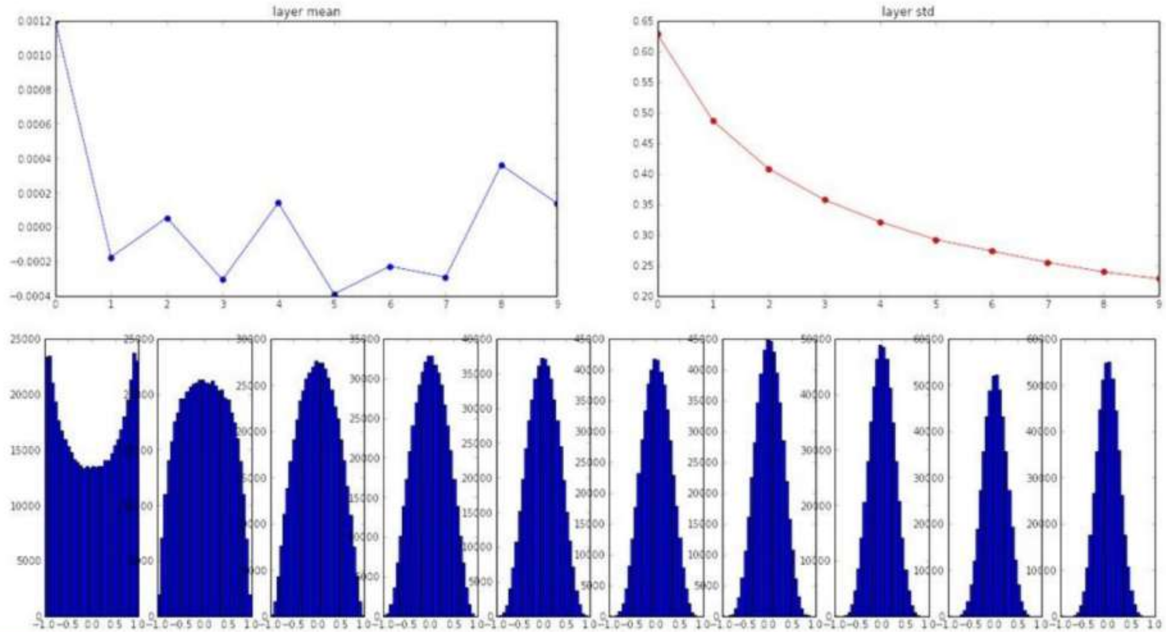
Xavier initialization

# 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
```

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

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



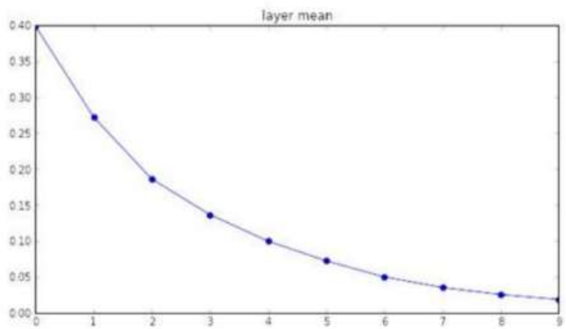

**Reasonable initialization.**

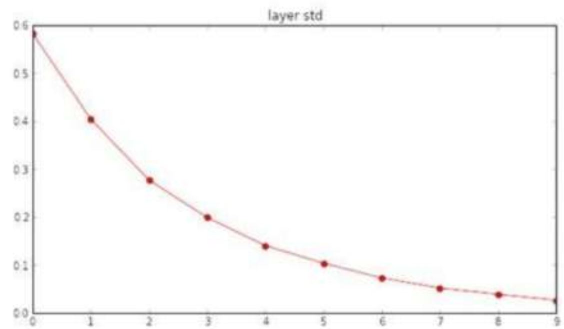
# Initialization

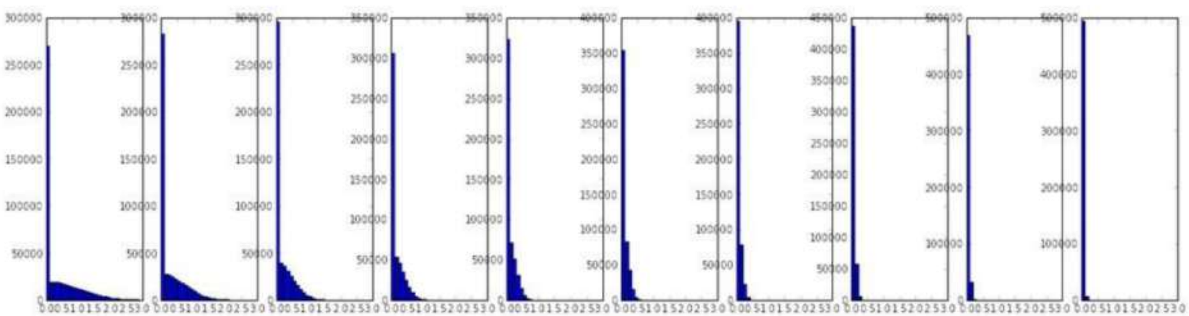
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
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
```

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

but when using the ReLU nonlinearity it breaks.



# Initialization

Smarter methods:

- He initialization

https://arxiv.org/pdf/1502.01852.pdf

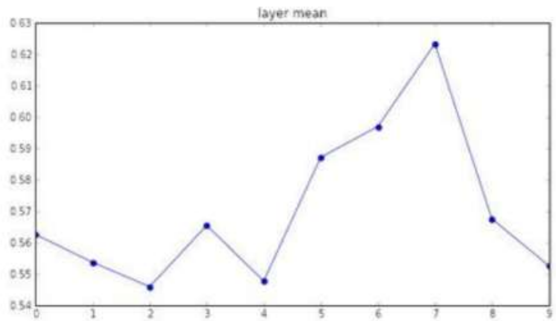
- The weights are initialized by multiplying by 2 the variance of the Xavier initialization.
- Used with ReLU activation

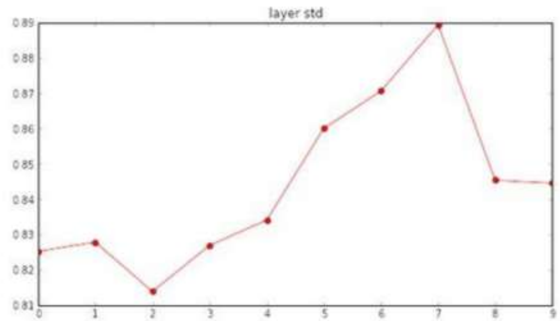
# Initialization

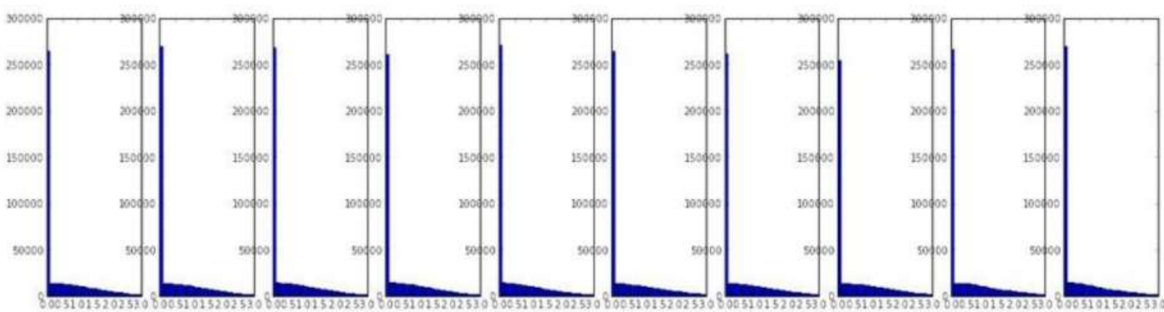
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
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
```

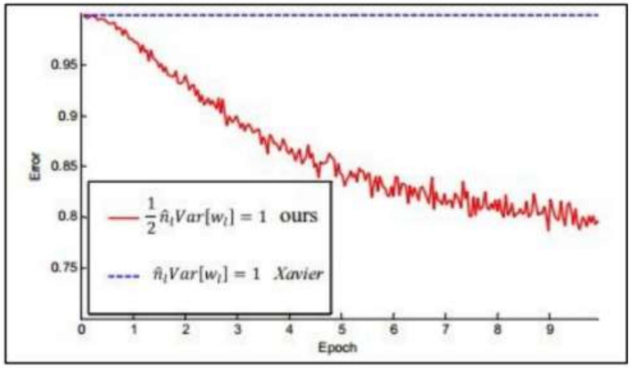
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

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





# Initialization

**Using Pretraining Models:**

- Initialize with weights from a network trained for another task / dataset
- Much faster convergence and better generalization
- Can either freeze or finetune the pretrained weights

# Initialization

Pretraining Methods:

- Initialize using unsupervised learning
  - Autoencoder
  - Restricted Boltzmann Machine (RBM)

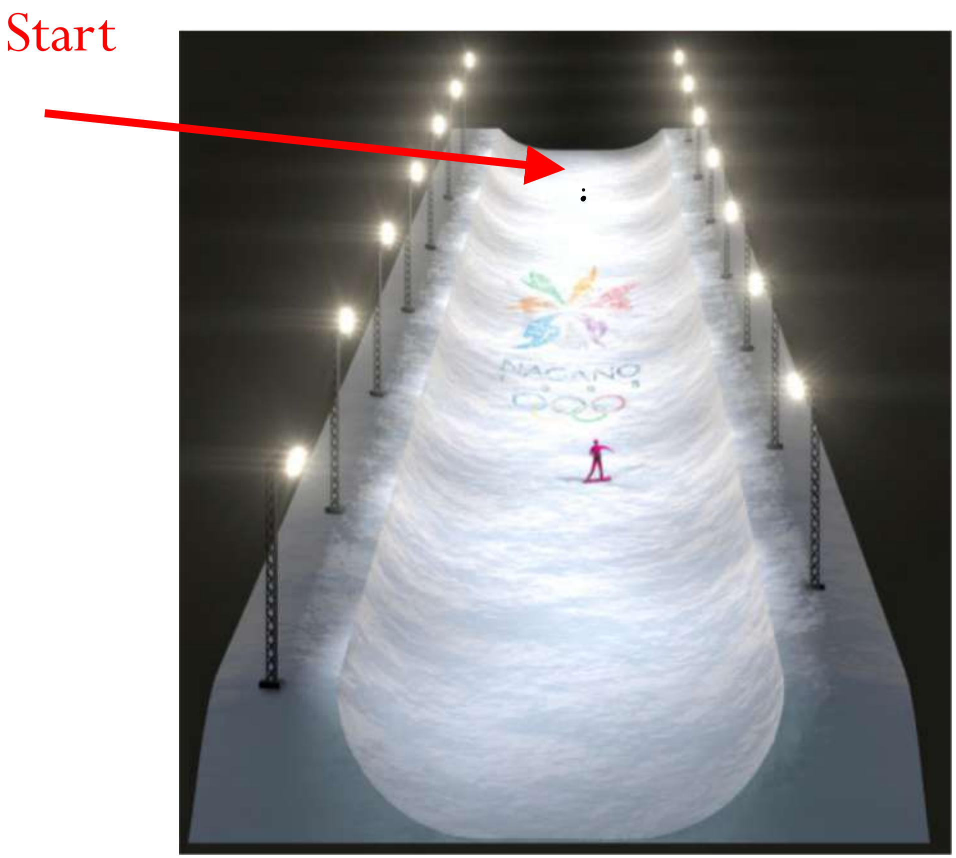
# Optimizers

# Optimizers

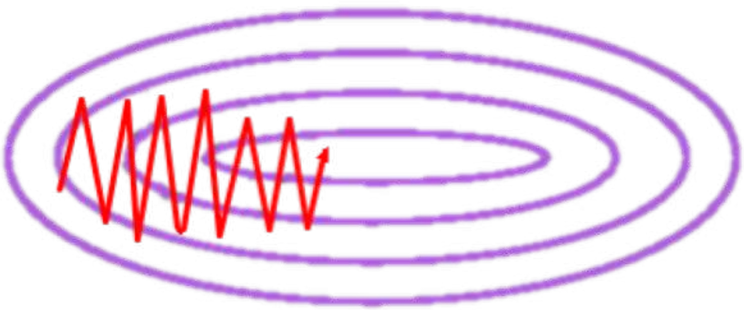
- Gradient Descent
- Stochastic Gradient Descent
- **Momentum**
- **Adagrad**
- **RMSProp**
- **Adam**

# Momentum Optimizer

What will SGD do?




Zig-zagging



# Momentum Optimizer

## SGD with Momentum

- Move faster in directions with consistent gradient
- Damps oscillating gradients in directions of high curvature
- Friction / momentum hyperparameter $\mu$ typically set to {0.50, 0.90, 0.99}

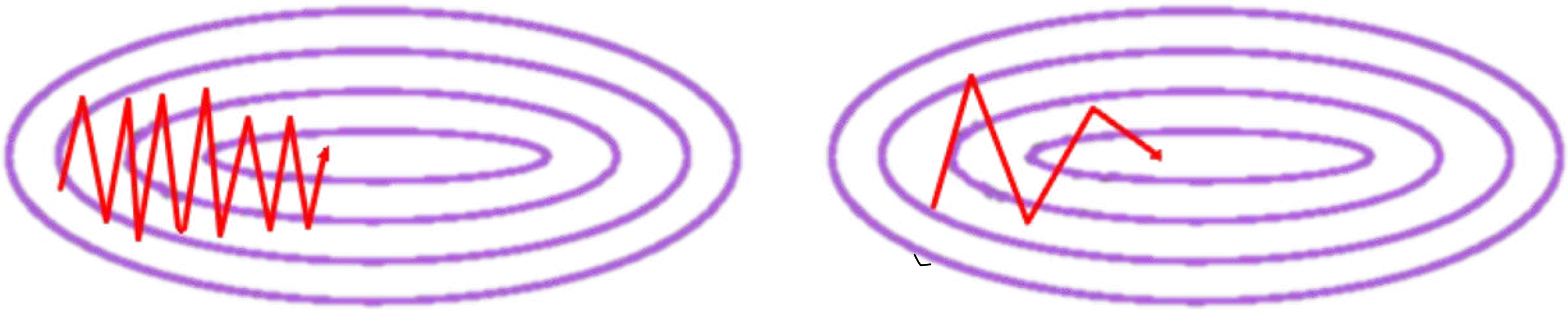
# Momentum Optimizers

$$\boxed{\theta_t} = \boxed{\theta_{t-1}} - \boxed{\alpha}\boxed{\nabla_{\theta_{t-1}} f(\theta_{t-1})}$$

$$\boxed{v_t} = \mu v_{t-1} - \boxed{\alpha}\boxed{\nabla f(\theta_{t-1})}$$

$$\boxed{\theta_t} = \boxed{\theta_{t-1}} + \boxed{v_t}$$

# Momentum Optimizer



- Cancels out oscillation
- Gathers speed in direction that matters

# Momentum Optimizer

$$v_t = \boxed{\mu} v_{t-1} - \alpha \nabla f(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

$$\theta_t = \theta_{t-1} + (\boxed{0} \cdot v_{t-1} - \alpha \nabla f(\theta_{t-1}))$$

Same as vanilla gradient descent

# Adagrad Optimizer

## Adagrad

Duchi et al 2011. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization"

- Gradient update depends on history of magnitude of gradients
- Parameters with small / sparse updates have larger learning rates
- Square root important for good performance
- More tolerance for learning rate

# Adagrad Optimizer

$$\boxed{\theta_t} = \boxed{\theta_{t-1}} - \boxed{\alpha}\boxed{\nabla_{\theta_{t-1}} f(\theta_{t-1})}$$

$$\boxed{g_t} = \sum_{\tau=1}^{t-1} (\nabla f(\theta_\tau))^2$$

$$= g_{t-1} + (\boxed{\nabla f(\theta_{t-1})})^2$$

$$\boxed{\theta_t} = \boxed{\theta_{t-1}} - \frac{\boxed{\alpha}}{\sqrt{\boxed{g_t} + \epsilon}} \odot \boxed{\nabla f(\theta_{t-1})}$$

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

# Adagrad Optimizers

$$\boxed{g_t} = \boxed{\sum_{\tau=1}^{t-1}} (\nabla f(\theta_\tau))^2$$

$$= g_{t-1} + (\nabla f(\theta_{t-1}))^2$$

What happens as t increases?

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\boxed{g_t} + \epsilon}} \odot \nabla f(\theta_{t-1})$$

# Adagrad Optimizer

Adagrad learning rate goes to 0

- Maintain entire history of gradients
- Sum of magnitude of gradients always increasing
- Forces learning rate to 0 over time
- Hard to compensate for in advance

Solutions?

# Adagrad Optimizer

Adagrad learning rate goes to 0

- Maintain entire history of gradients
- Sum of magnitude of gradients always increasing
- Forces learning rate to 0 over time
- Hard to compensate for in advance

Solutions:

- Forget gradients far in the past
- In practice, downweight previous gradients exponentially

# RMSProp Optimizer

Hinton et al. 2012,
http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

- Only cares about recent gradients
- Good property because optimization landscape changes
- Otherwise like Adagrad
- Standard gamma is 0.9

# RMSProp Optimizer

$$g_t = \sum_{\tau=1}^{t-1} (\nabla f(\theta_\tau))^2$$

$$= g_{t-1} + (\nabla f(\theta_{t-1}))^2$$

$$g_t = (1 - \boxed{\gamma}) \sum_{\tau=1}^{t-1} \boxed{\gamma^{t-1-\tau}} (\nabla f(\theta_\tau))^2$$

$$= \boxed{\gamma} g_{t-1} + \boxed{(1 - \gamma)} \nabla (f(\theta_{t-1}))^2$$

g

# RMSProp Optimizer

$$g_t = (1-\gamma)\sum_{\tau=1}^{t-1} \gamma^{t-1-\tau}\big(\nabla f(\theta_\tau)\big)^2$$

$$= \gamma g_{t-1} + (1-\gamma)\nabla\big(f(\theta_{t-1})\big)^2$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{g_t + \epsilon}} \odot \nabla f(\theta_{t-1})$$

# Adam Optimizer

Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014)

- Essentially, combines RMSProp and Momentum
- Includes bias correction term from m and v
- Default parameters are surprisingly good

# Adam Optimizer

$$\boxed{\mu} v_{t-1} - \boxed{\alpha} \nabla f(\theta_{t-1})$$ Momentum

$$\boxed{\gamma} g_{t-1} + \boxed{(1-\gamma)} \nabla (f(\theta_{t-1}))^2$$ RMSProp

$$m_t = \boxed{\beta_1} m_{t-1} + \boxed{(1-\beta_1)} \nabla f(\theta_{t-1})$$

$$v_t = \boxed{\beta_2} v_{t-1} + \boxed{(1-\beta_2)} (\nabla f(\theta_{t-1}))^2$$

# Adam Optimizer

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla f(\theta_{t-1})$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla f(\theta_{t-1}))^2$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{v_t + \epsilon}} \odot m_t$$

# What to use

- SGD + momentum and Adam are good first steps
- Just use default parameters for Adam
- Learning rate decay always good

# Callbacks

# Callbacks

- A set of functions to be applied at given stages of the training procedure

- Use callbacks to

  - get a view on internal states and statistics of the model during training

  - Customize/interfere with training

# Callbacks

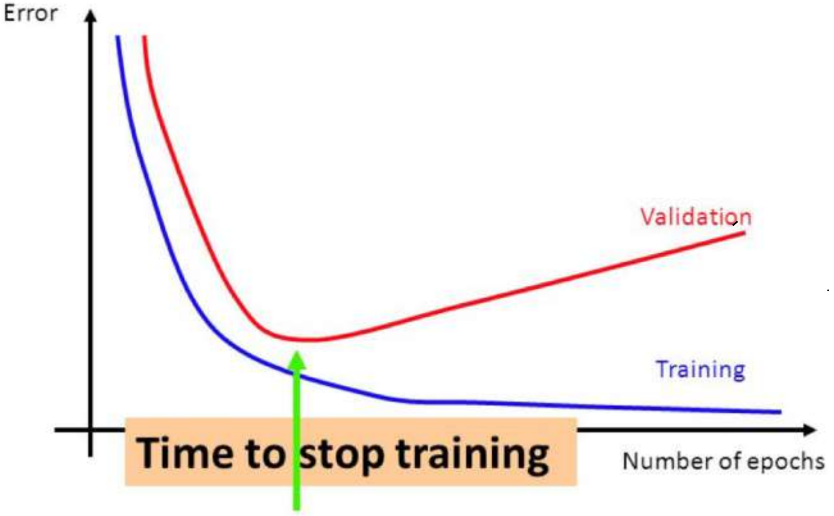
- **ModelCheckpoint:** save the model after every epoch

```
ModelCheckpoint (
    filepath,
    monitor='val_loss',
    verbose=0,
    save_best_only=False,
    save_weights_only=False,
    mode='auto',
    period=1
)
```

# Callbacks

- **EarlyStopping:** Stop training when a monitored quantity has stopped improving.

```
EarlyStopping(
    monitor='val_loss',
    min_delta=0,
    patience=0,
    verbose=0,
    mode='auto',
    baseline=None,
    restore_best_weights=False
)
```



# Callbacks

- **ReduceLROnPlateau:** Reduce learning rate when a metric has stopped improving

```
reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.2,
    patience=5,
    min_lr=0.001
)
```

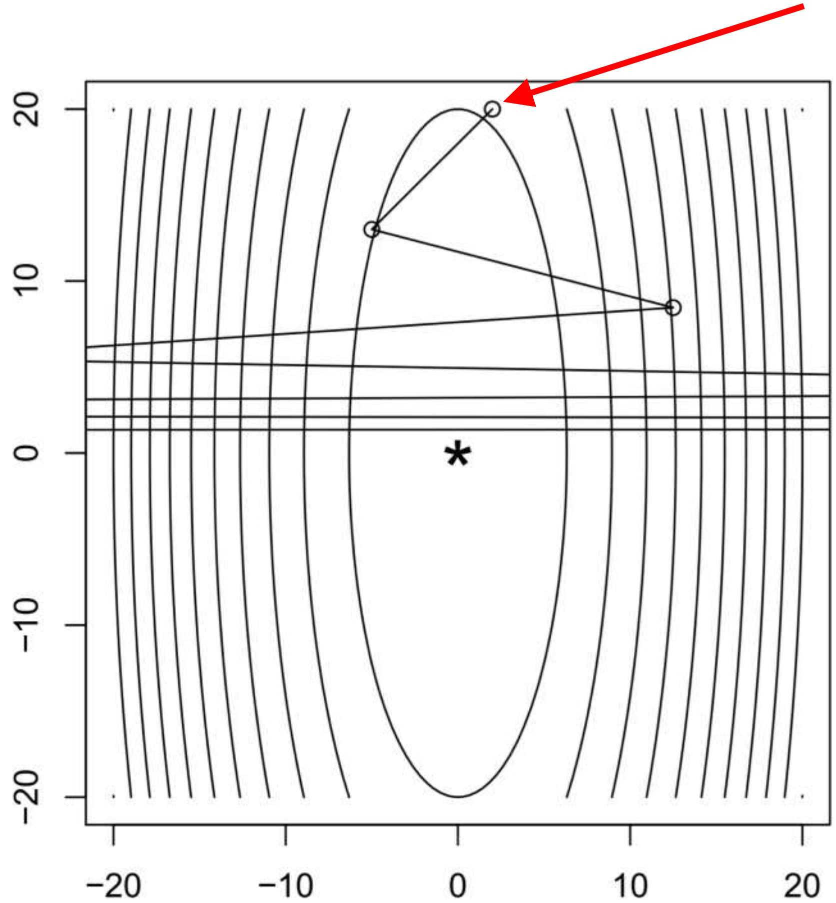
# Callbacks

- **LearningRateScheduler:**

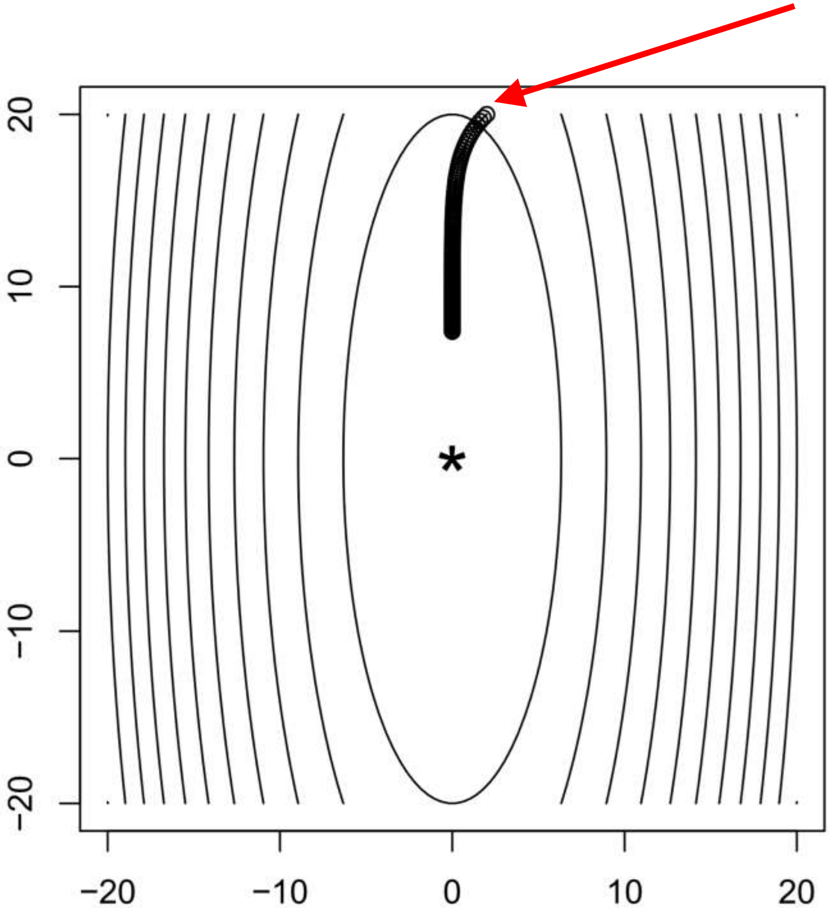
  LearningRateScheduler(schedule, verbose=0)

  **schedule**: a function that takes an epoch index as input (integer, indexed from 0) and current learning rate and returns a new learning rate as output (float).

# Too big learning rate



# Too small learning rate



# How to pick the learning rate?

- Too big = diverge, too small = slow convergence
- No “one learning rate to rule them all”
- Start from a high value and keep cutting by half if model diverges
- Learning rate schedule: decay learning rate over time

# Example: Exponential Decay Learning Rate Schedule

```
def exp_decay(t):

      initial_lrate = 0.1
      k = 0.1
      lrate = initial_lrate * exp(-k*t)
      return lrate

Lrate = LearningRateScheduler(exp_decay)

callbacks_list = [lrate]

model.fit(
      X_train, y_train,
      validation_data=(X_test, y_test),
      epochs=epochs,
      batch_size=batch_size,
      callbacks=callbacks_list,
      verbose=2
)
```

lr = lr0 * e^(−kt), where lr, k are hyperparameters and t is the iteration number.

# Callbacks

on_batch_begin
on_batch_end
on_epoch_begin
on_epoch_end
on_predict_batch_begin
on_predict_batch_end
on_predict_begin
on_predict_end
on_test_batch_begin
on_test_batch_end
on_test_begin
on_test_end
on_train_batch_begin
on_train_batch_end
on_train_begin
on_train_end
set_model
set_params

```
import datetime

class MyCustomCallback(tf.keras.callbacks.Callback):

  def on_train_batch_begin(self, batch, logs=None):
    print('Training: batch {} begins at {}'.format(batch,
          datetime.datetime.now().time()))

  def on_train_batch_end(self, batch, logs=None):
    print('Training: batch {} ends at {}'.format(batch,
          datetime.datetime.now().time()))

def on_test_batch_begin(self, batch, logs=None):
    print('Evaluating: batch {} begins at {}'.format(batch,
          datetime.datetime.now().time()))

  def on_test_batch_end(self, batch, logs=None):
    print('Evaluating: batch {} ends at {}'.format(batch,
          datetime.datetime.now().time()))
```

```python
model = get_model()
_ = model.fit(x_train, y_train,
              batch_size=64,
              epochs=1,
              steps_per_epoch=5,
              verbose=0,
              callbacks=[MyCustomCallback()])
```

```
Training: batch 0 begins at 22:53:23.497069
Training: batch 0 ends at 22:53:24.406412
Training: batch 1 begins at 22:53:24.406879
Training: batch 1 ends at 22:53:24.410124
Training: batch 2 begins at 22:53:24.410395
Training: batch 2 ends at 22:53:24.412897
Training: batch 3 begins at 22:53:24.413114
Training: batch 3 ends at 22:53:24.415623
Training: batch 4 begins at 22:53:24.415865
Training: batch 4 ends at 22:53:24.418233
```

# Batch Normalization

Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *arXiv preprint arXiv:1502.03167* (2015).

# Batch Normalization

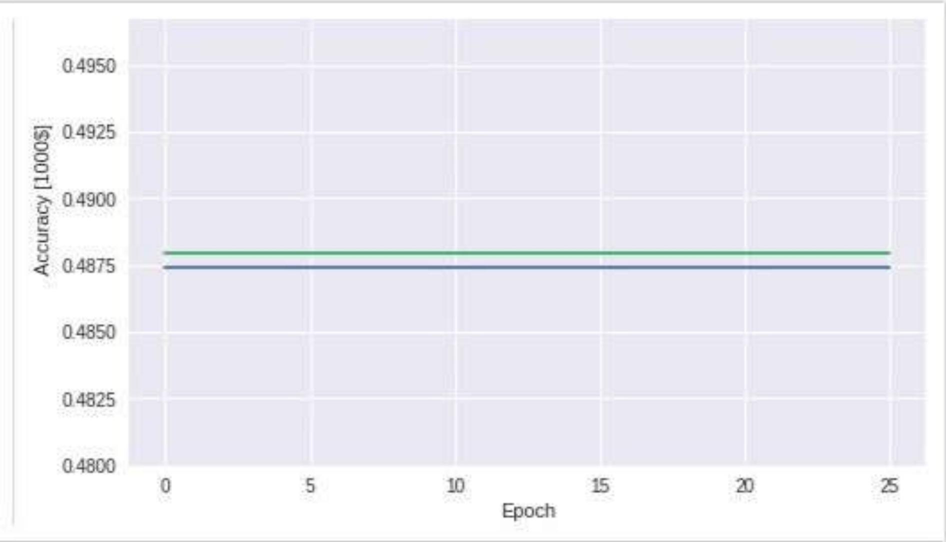
- Data normalization is necessary for majority of machine learning models

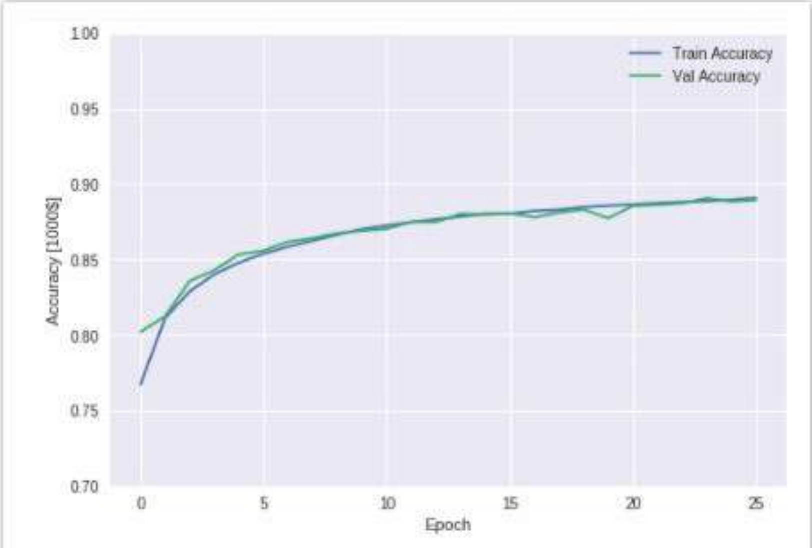
| Elevation | Aspect | Slope | Horizontal_D | Vertical_Dist | Horizontal_D | Hillshade_9a | Hillshade_No | Hillshade_3p | Horizontal_Distance_To_Fire_Points |
|---|---|---|---|---|---|---|---|---|---|
| 2596 | 51 | 3 | 258 | 0 | 510 | 221 | 232 | 148 | 6279 |
| 2590 | 56 | 2 | 212 | -6 | 390 | 220 | 235 | 151 | 6225 |
| 2804 | 139 | 9 | 268 | 65 | 3180 | 234 | 238 | 135 | 6121 |
| 2785 | 155 | 18 | 242 | 118 | 3090 | 238 | 238 | 122 | 6211 |
| 2595 | 45 | 2 | 153 | -1 | 391 | 220 | 234 | 150 | 6172 |
| 2579 | 132 | 6 | 300 | -15 | 67 | 230 | 237 | 140 | 6031 |
| 2606 | 45 | 7 | 270 | 5 | 633 | 222 | 225 | 138 | 6256 |
| 2605 | 49 | 4 | 234 | 7 | 573 | 222 | 230 | 144 | 6228 |
| 2617 | 45 | 9 | 240 | 56 | 666 | 223 | 221 | 133 | 6244 |
| 2612 | 59 | 10 | 247 | 11 | 636 | 228 | 219 | 124 | 6230 |
| 2612 | 201 | 4 | 180 | 51 | 735 | 218 | 243 | 161 | 6222 |
| 2886 | 151 | 11 | 371 | 26 | 5253 | 234 | 240 | 136 | 4051 |
| 2742 | 134 | 22 | 150 | 69 | 3215 | 248 | 224 | 92 | 6091 |
| 2609 | 214 | 7 | 150 | 46 | 771 | 213 | 247 | 170 | 6211 |
| 2503 | 157 | 4 | 67 | 4 | 674 | 224 | 240 | 151 | 5600 |
| 2495 | 51 | 7 | 42 | 2 | 752 | 224 | 225 | 137 | 5576 |
| 2610 | 259 | 1 | 120 | -1 | 607 | 216 | 239 | 161 | 6096 |
| 2517 | 72 | 7 | 85 | 6 | 595 | 228 | 227 | 133 | 5607 |
| 2504 | 0 | 4 | 95 | 5 | 691 | 214 | 232 | 156 | 5572 |

# Batch Normalization

- Data normalization is necessary for majority of machine learning models

**Left: Model Accuracy, without normalized data**
**Right: Model Accuracy with normalized data**

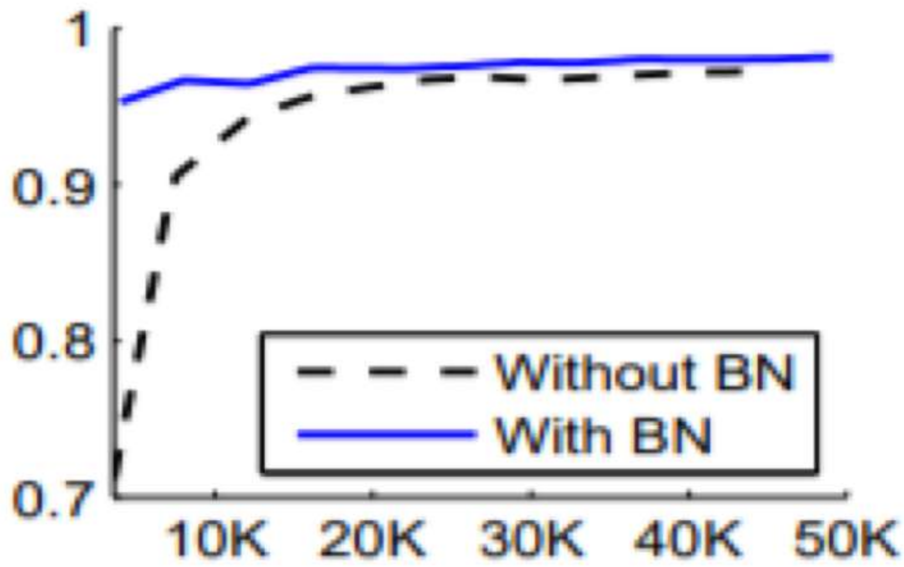


# Batch Normalization

More stable inputs = faster training

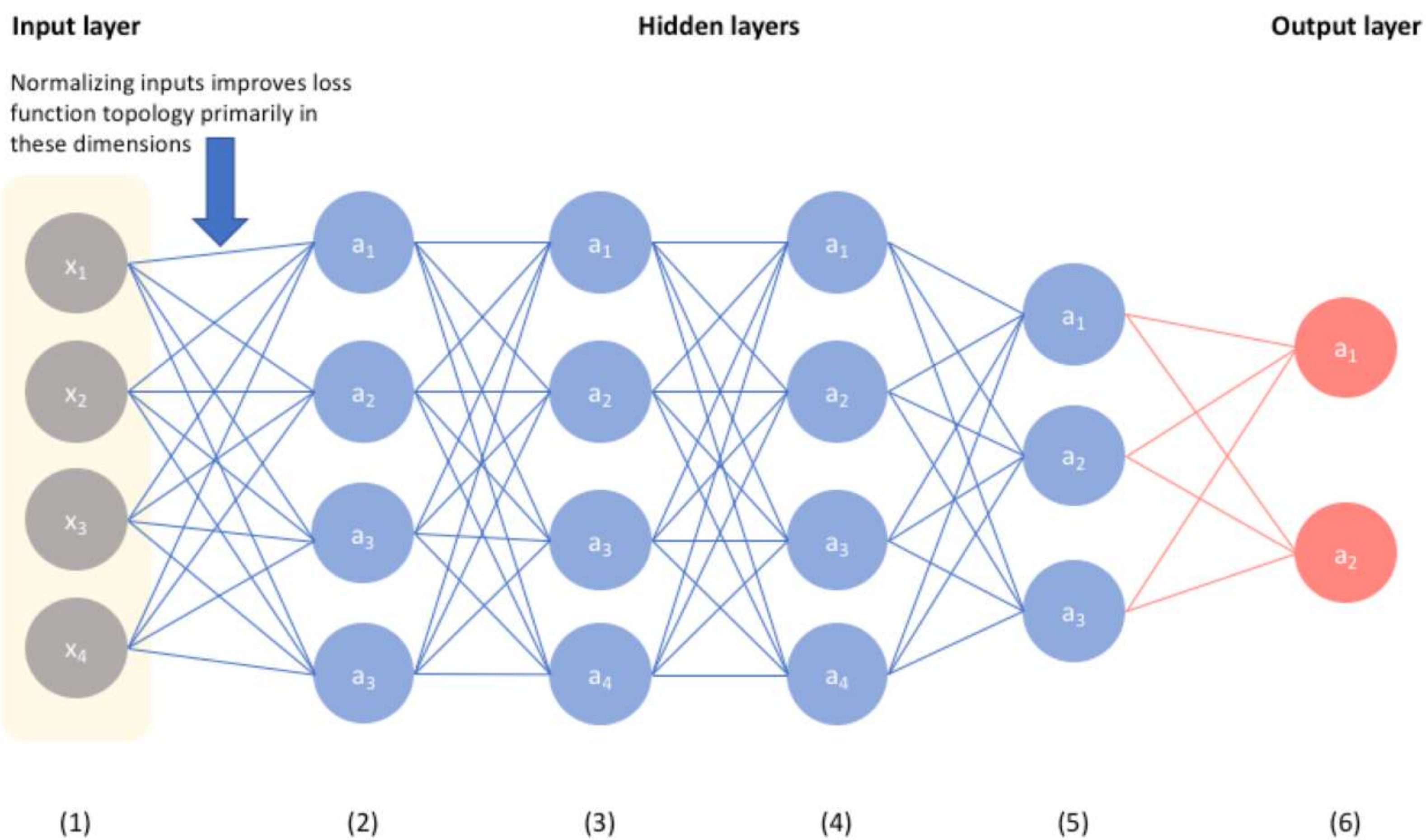
MNIST test accuracy vs number of training steps




https://arxiv.org/pdf/1502.03167.pdf

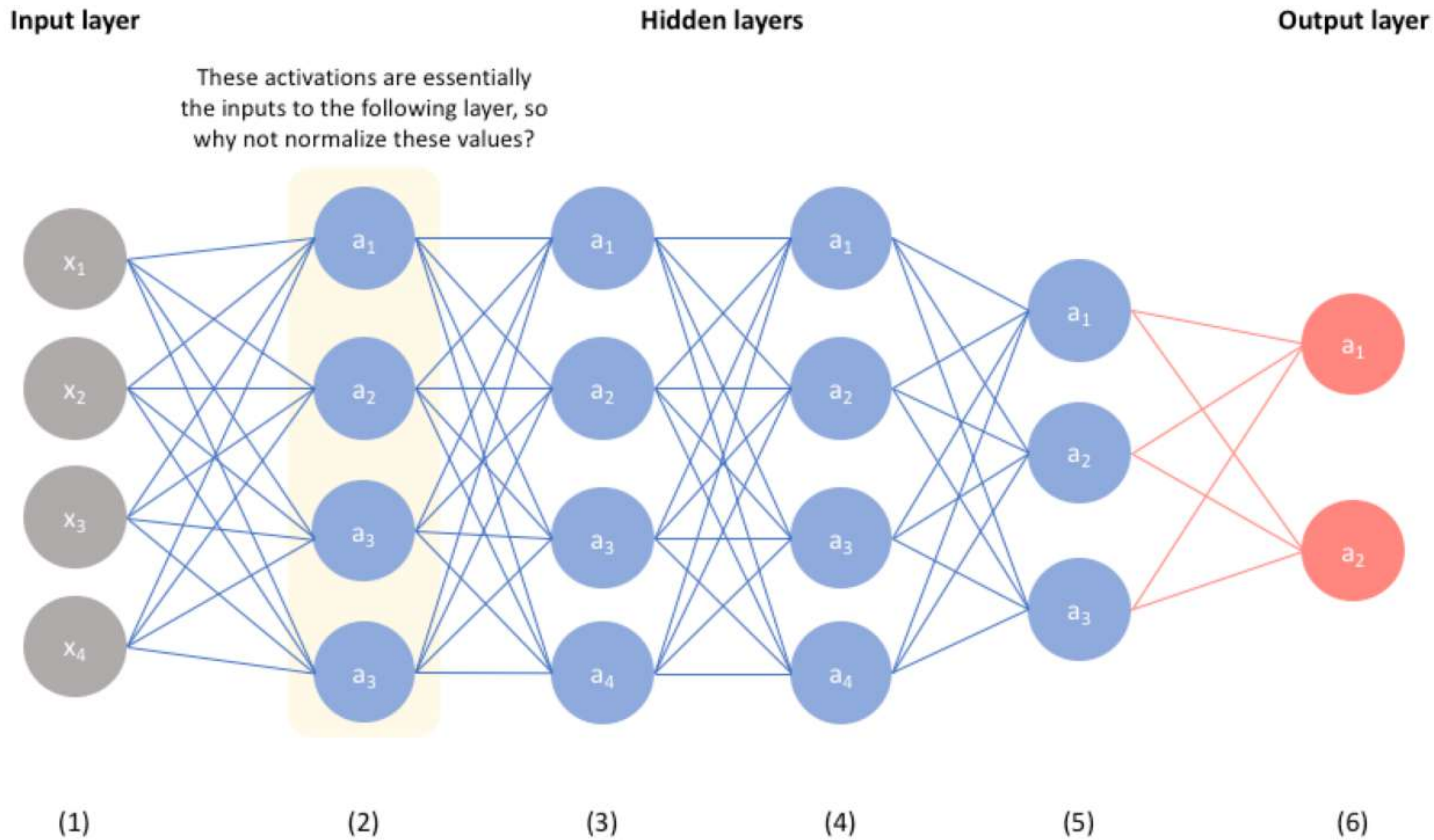
# Batch Normalization

Normalizing the input of your network is a well-established technique for improving the convergence of training

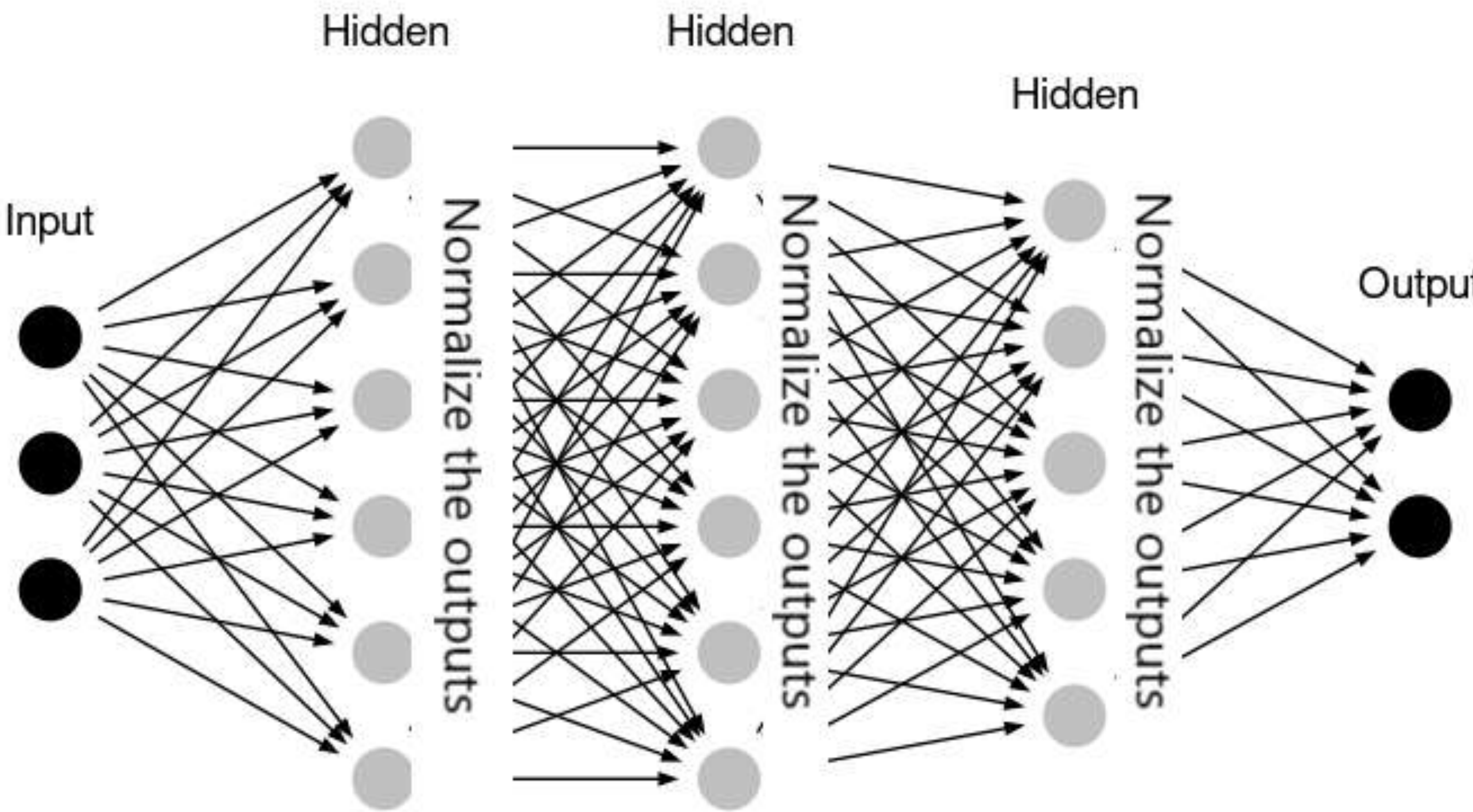


# Batch Normalization

Idea: normalize each hidden layer to help speed up convergence



# Batch Normalization



# Batch Normalization

Implementation: before activation

$z_i^{[l]}$ : linear combinations from the previous layer for observation $i$

$$\mu = \frac{1}{m}\sum_i z_i^{[l]} \qquad \sigma^2 = \frac{1}{m}\sum_i \left(z_i^{[l]} - \mu\right)^2$$

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \qquad \tilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta$$

Why?

# Batch Normalization

Benefits of Batch Normalization:

- Speed up Learning
- Makes weights in deeper layers more robust to weight changes of earlier layers
- BN has a regularization effect because
  - Each mini-batch is scaled by the mean/variance computed only on that mini-batch
  - This adds some noise to the value of $z_i^{[l]}$
  - Larger mini-batch size reduces this effect

# Batch Normalization

What about test time?

- Only seeing one instance

# Batch Normalization

What about test time?

- Only seeing one instance
- Use training sample mean and variance
- Use moving average mean and variance

# Batch Normalization in Tensorflow

**`tf.keras.layers.BatchNormalization(inpu`**

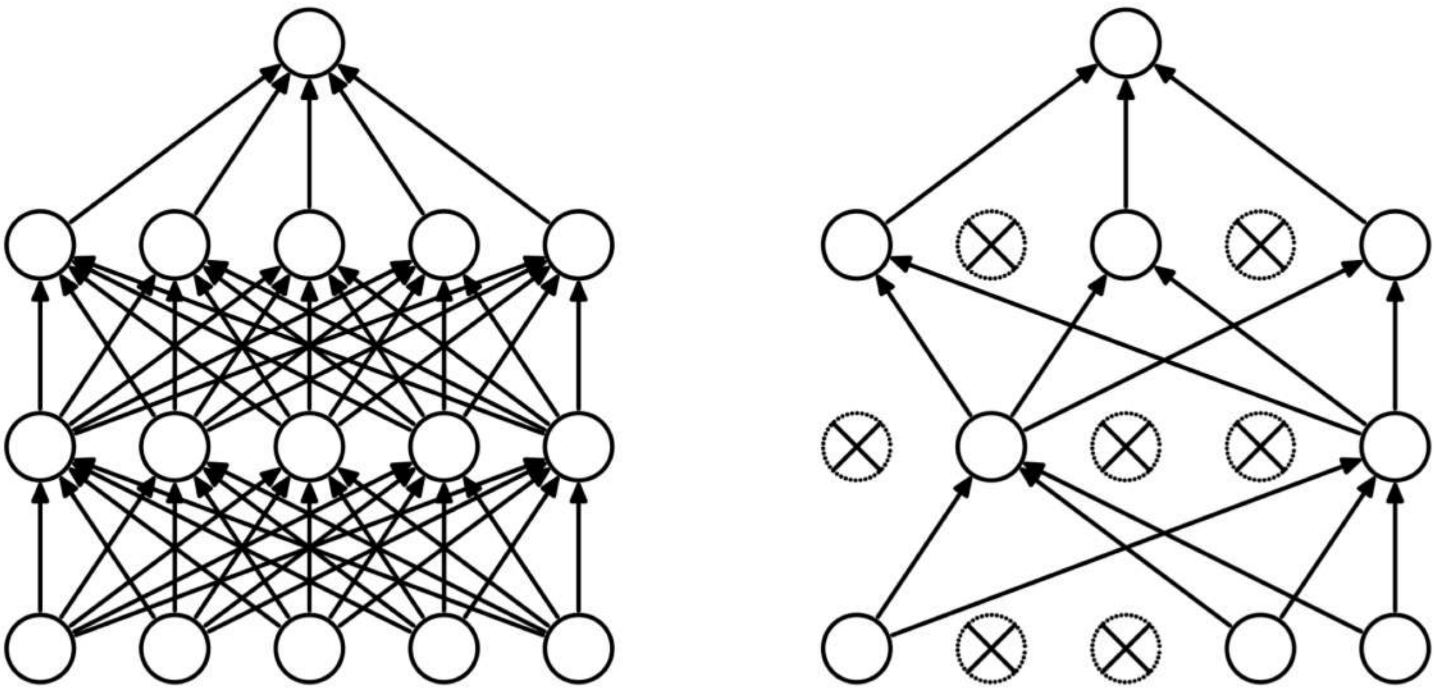
Documentation:
https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/BatchNormalization

# Dropout

# Dropout

- Regularization: **Dropout**
  - Randomly set some neurons to zero in the forward pass

Srivastava et al. 2014. "Dropout: a simple way to prevent neural networks from overfitting"

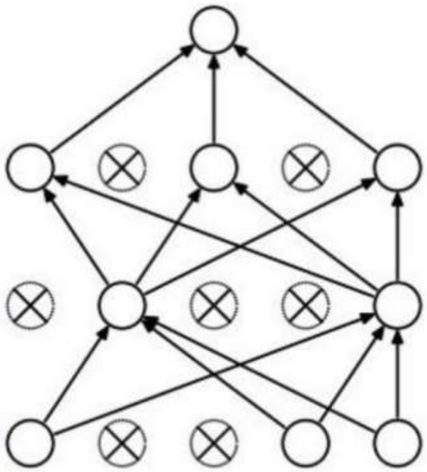


(a) Standard Neural Net (b) After applying dropout.

# Dropout

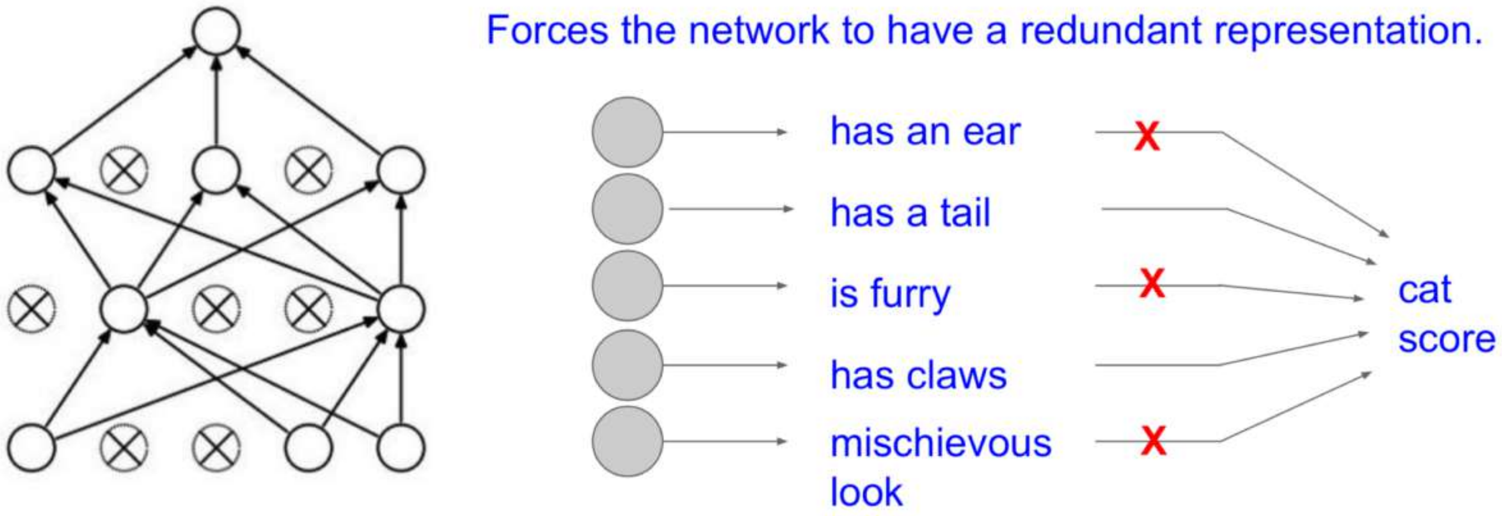
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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)
```

Example forward pass with a 3-layer network using dropout



# Dropout

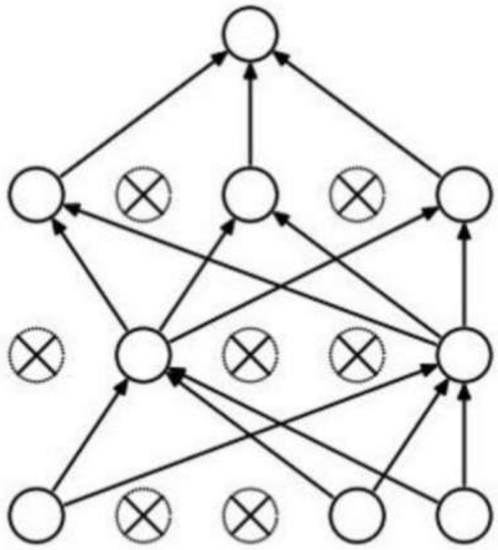
Waaaait a second…
How could this possibly be a good idea?




- Complex co-adaptations
- Forces hidden units to derive useful features on their own
- Sampling from $2^n$ possible related networks

# Dropout

Waaaait a second…
How could this possibly be a good idea?



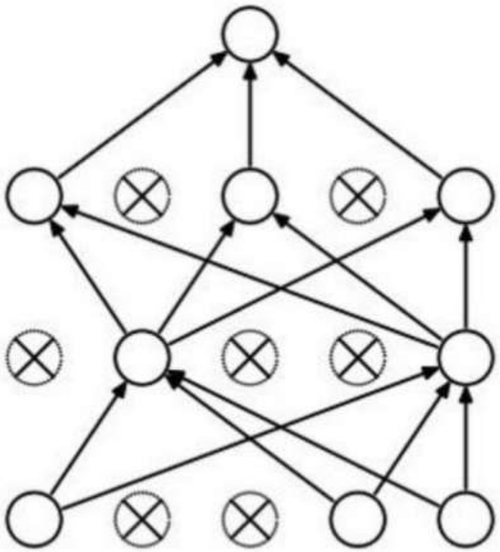
Another interpretation:

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

Each binary mask is one model, gets trained on only ~one datapoint.

# Dropout

At test time….



**Ideally**:
want to integrate out all the noise

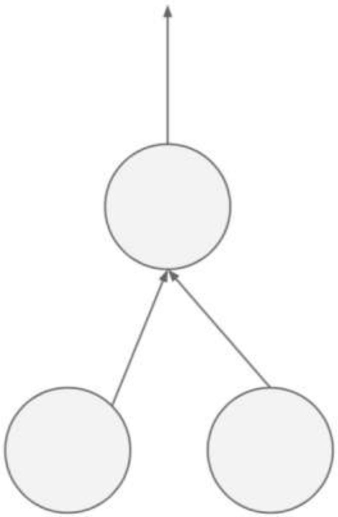
**Monte Carlo approximation:**
do many forward passes with different dropout masks, average all predictions

# Dropout

## At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



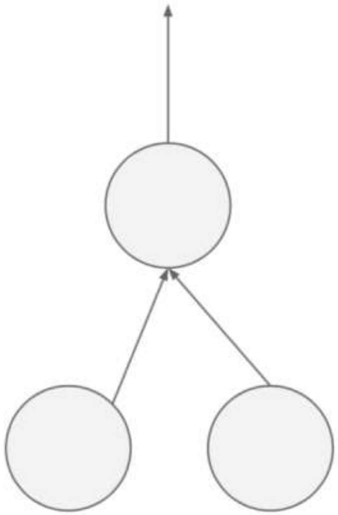
(this can be shown to be an approximation to evaluating the whole ensemble)

# Dropout

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



Q: Suppose that with all inputs present at test time the output of this neuron is x.

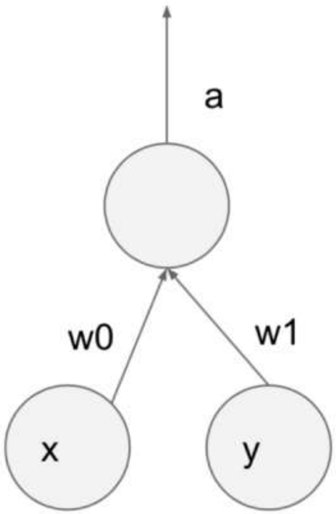
What would its output be during training time, in expectation? (e.g. if p = 0.5)

# Dropout

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).




during test: **a = w0*x + w1*y**

during train:

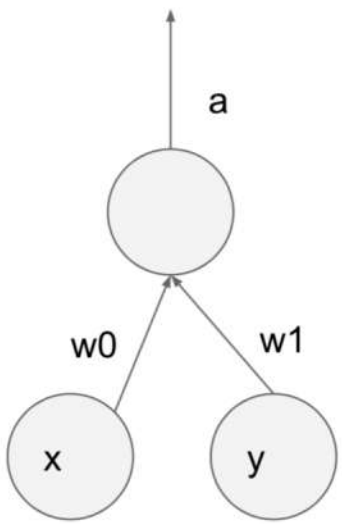
E[a] = ¼ * (w0*0 + w1*0
w0*0 + w1*y
w0*x + w1*0
w0*x + w1*y)
= ¼ * (2 w0*x + 2 w1*y)
= ½ * **(w0*x + w1*y)**

# Dropout

## At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).




during test: $\mathbf{a = w0*x + w1*y}$

during train:

$$
\begin{aligned}
E[a] = \tfrac{1}{4} * (&w0*0 + w1*0 \\
&w0*0 + w1*y \\
&w0*x + w1*0 \\
&w0*x + w1*y) \\
= \tfrac{1}{4} * (&2\, w0*x + 2\, w1*y) \\
= \tfrac{1}{2} * (&\mathbf{w0*x + w1*y})
\end{aligned}
$$

With p=0.5, using all inputs in the forward pass would inflate the activations by 2x from what the network was "used to" during training!
=> Have to compensate by scaling the activations back down by ½

# Dropout

We can do something approximate analytically

```
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:
<u>output at test time</u> = <u>expected output at training time</u>

# Dropout

```
""" 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
```

Dropout Summary

drop in forward pass

scale at test time

# Dropout

## More common: “Inverted dropout”

```
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
```

test time is unchanged!

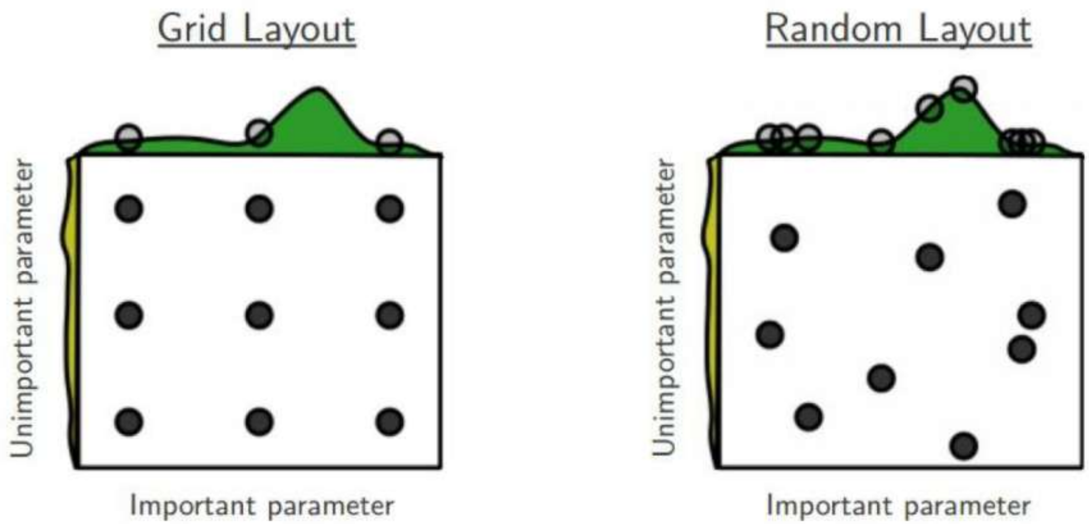
# Babysitting Model Training

# BabySitting

- Check the loss is reasonable using a simple model and a small input size
- Make sure that you can overfit a small size of the input data
- Monitor the training and validation loss
  - loss not changing: maybe learning rate is too low
  - loss = NaN, learning rate is too high
  - add regularization gradually

# BabySitting

- Hyperparameter Optimization
  - Cross Validation: start with few epochs, move on to longer runs.
  - Grid Search vs. Random Search




*Random Search for Hyper-Parameter Optimiz*
Bergstra and Bengio, 2012

# BabySitting

**Hyperparameters to play with:**

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

neural networks practitioner
music = loss function