

Lecture 7: Training Neural Networks

Administrative: A2

A2 is out, due Monday May 2nd, 11:59pm

Where we are now...

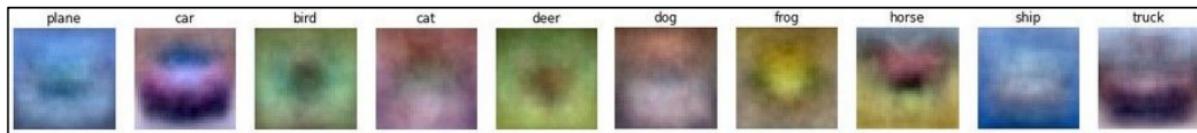
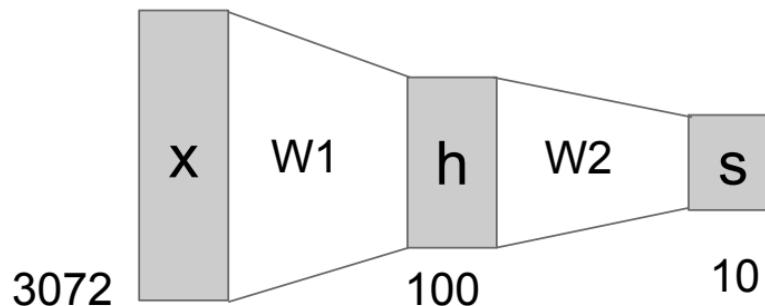
Neural Networks

Linear score function:

$$f = Wx$$

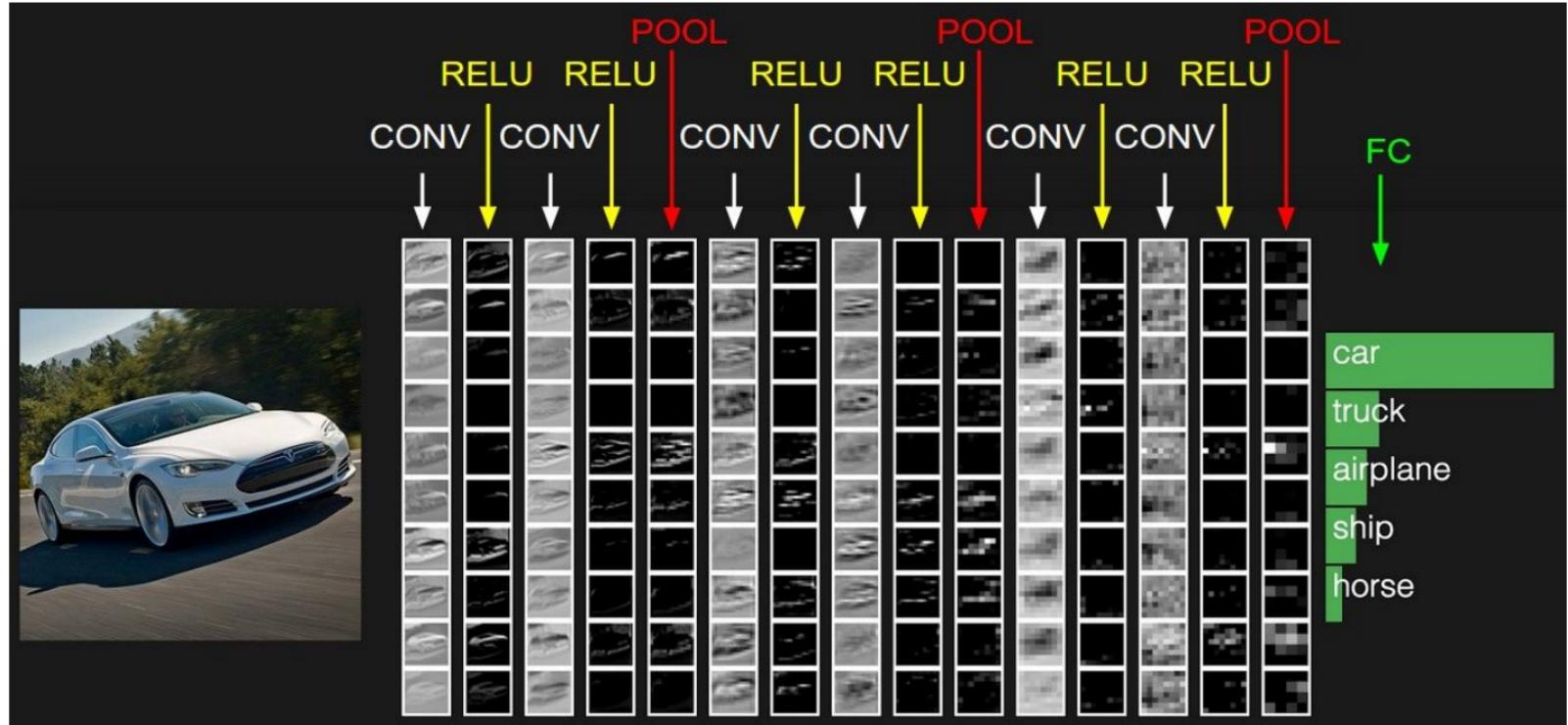
2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$



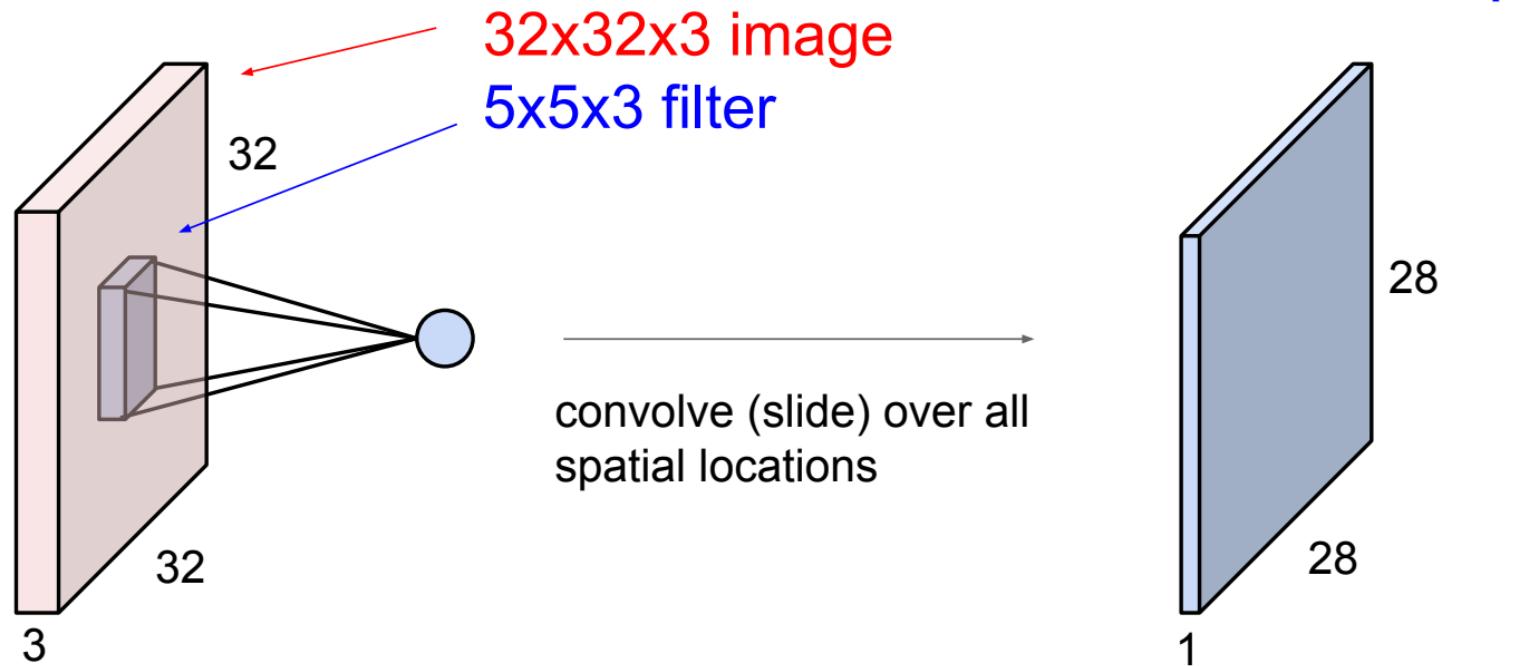
Where we are now...

Convolutional Neural Networks



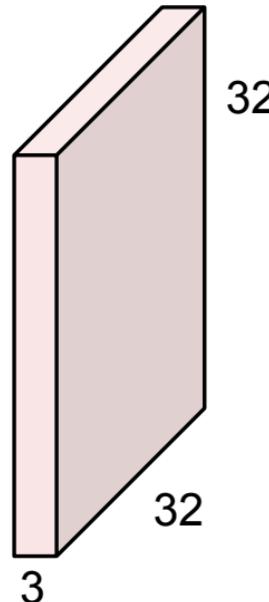
Where we are now...

Convolutional Layer



Where we are now...

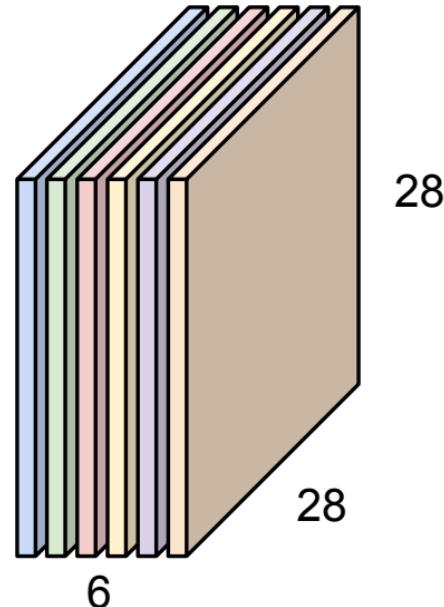
Convolutional Layer



Convolution Layer

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

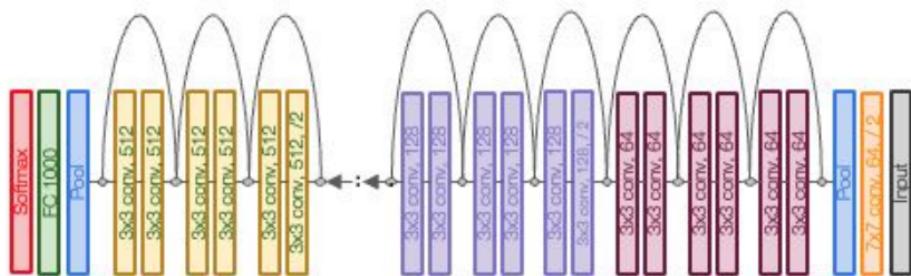
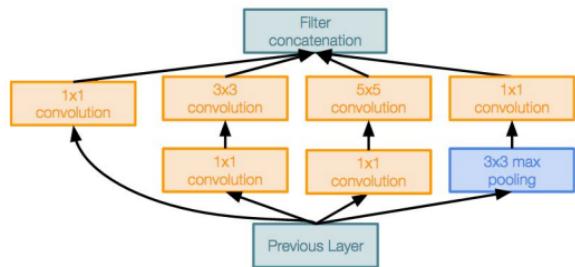
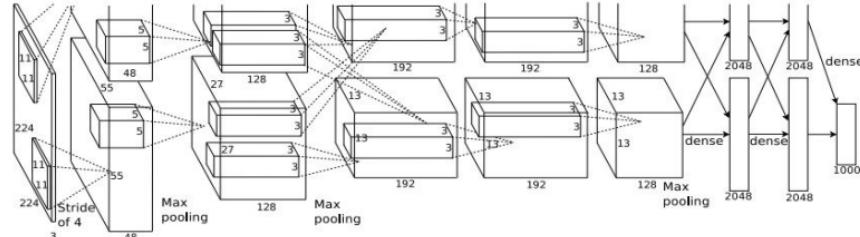
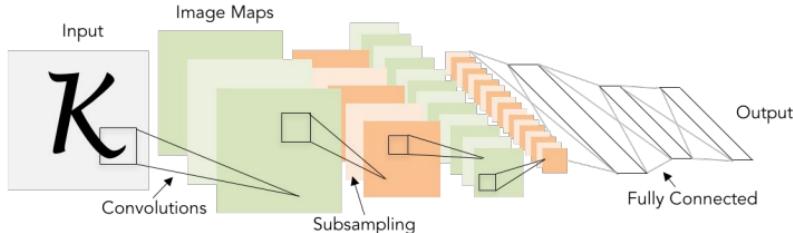
activation maps



We stack these up to get a “new image” of size 28x28x6!

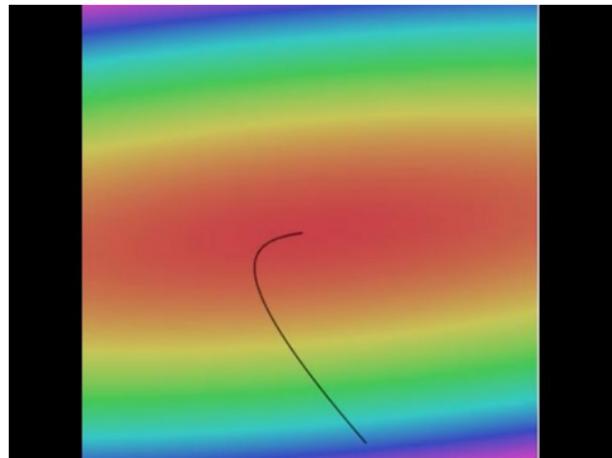
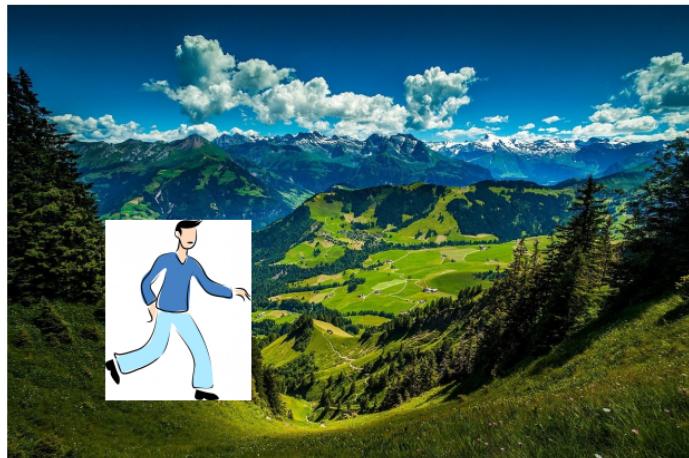
Where we are now...

CNN Architectures



Where we are now...

Learning network parameters through optimization



```
# Vanilla Gradient Descent  
  
while True:  
    weights_grad = evaluate_gradient(loss_fun, data, weights)  
    weights += - step_size * weights_grad # perform parameter update
```

[Landscape image](#) is CC0 1.0 public domain
[Walking man image](#) is CC0 1.0 public domain

Where we are now...

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

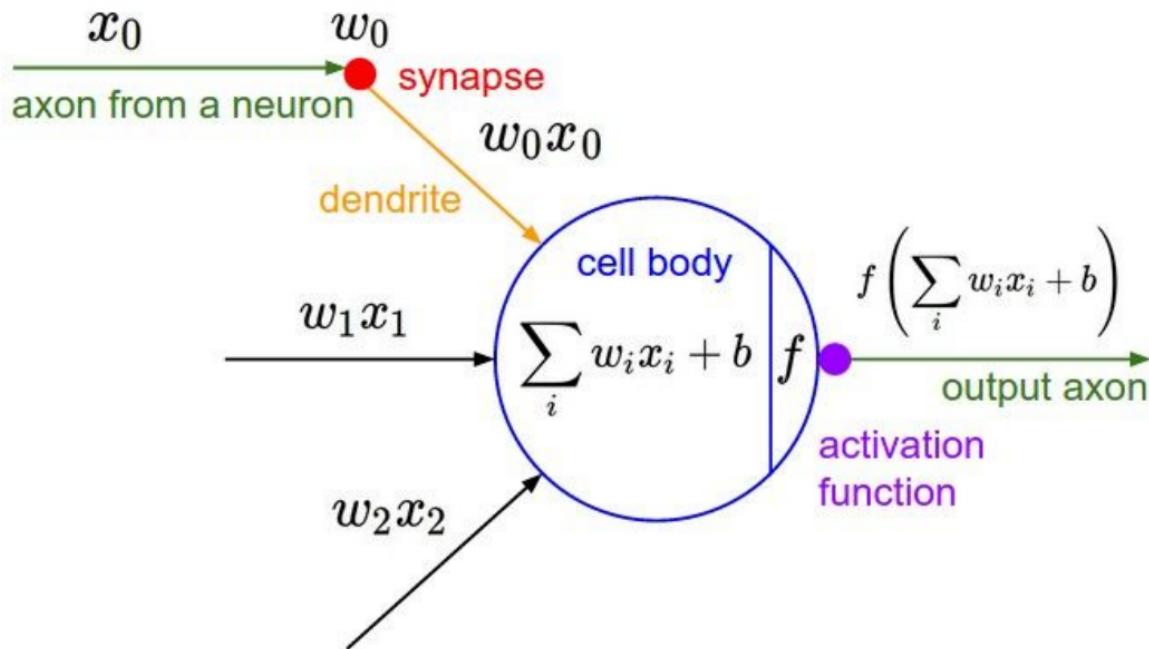
Today: Training Neural Networks

Overview

1. **One time set up:** activation functions, preprocessing, weight initialization, regularization, gradient checking
2. **Training dynamics:** babysitting the learning process, parameter updates, hyperparameter optimization
3. **Evaluation:** model ensembles, test-time augmentation, transfer learning

Activation Functions

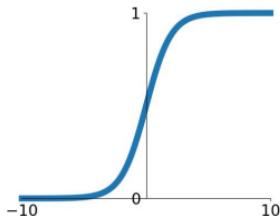
Activation Functions



Activation Functions

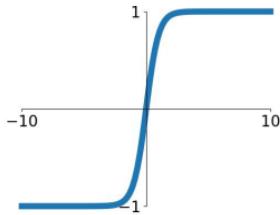
Sigmoid

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



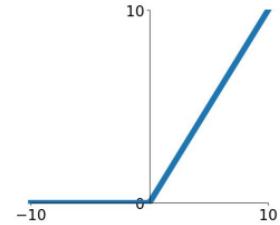
tanh

$$\tanh(x)$$



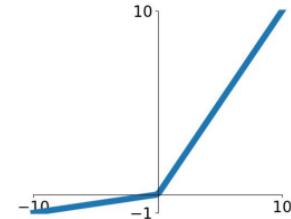
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

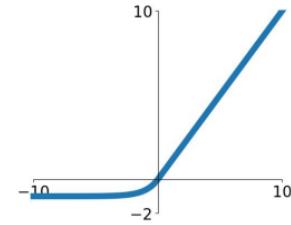


Maxout

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

ELU

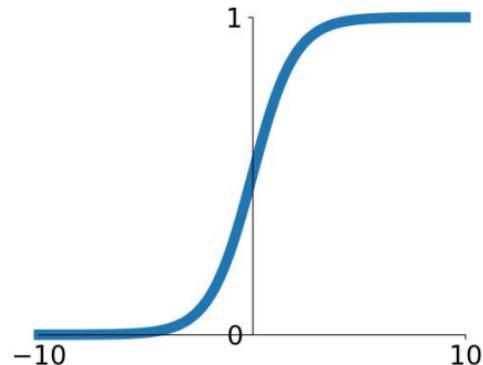
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation Functions

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

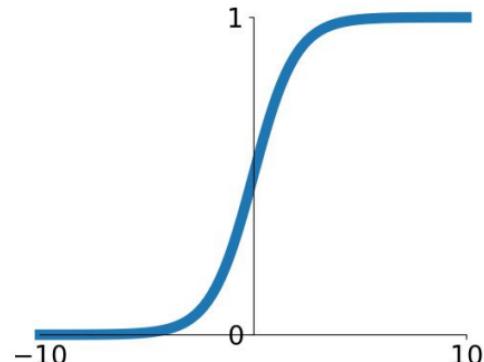
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron



Sigmoid

Activation Functions

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



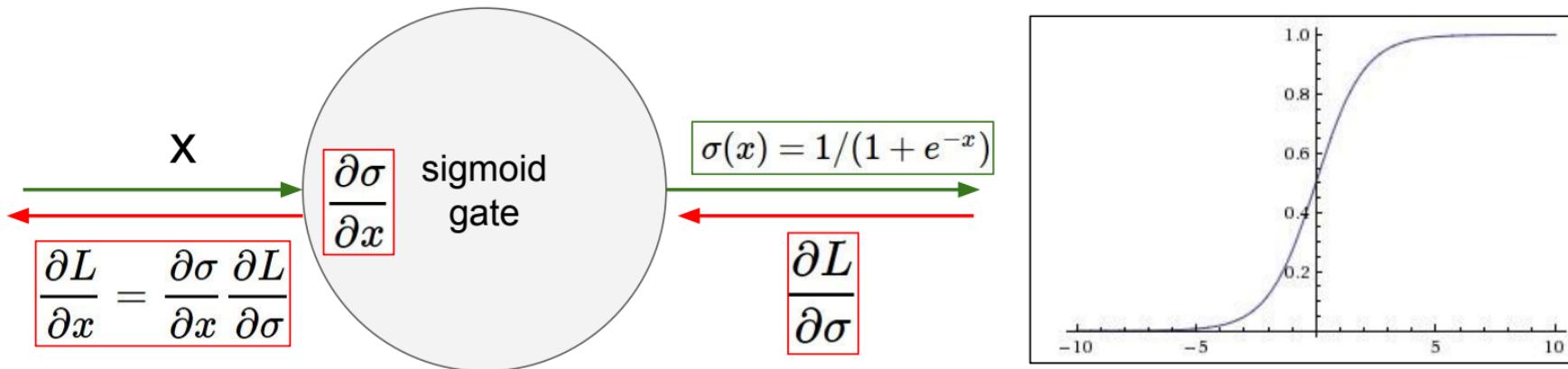
Sigmoid

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

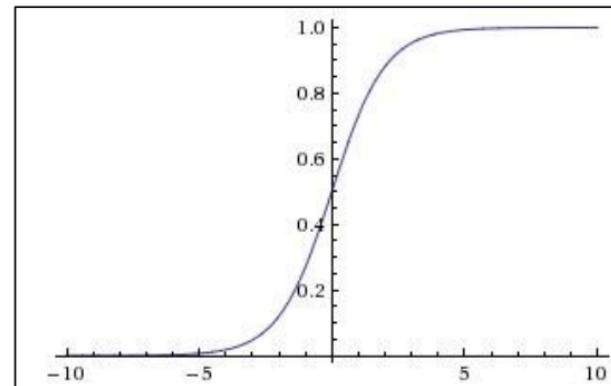
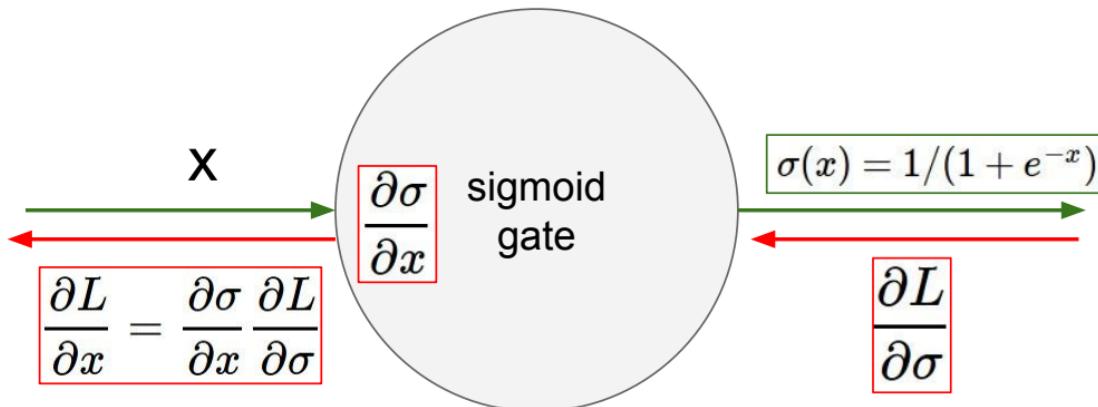
3 problems:

1. **Saturated neurons “kill” the gradients**

梯度消失

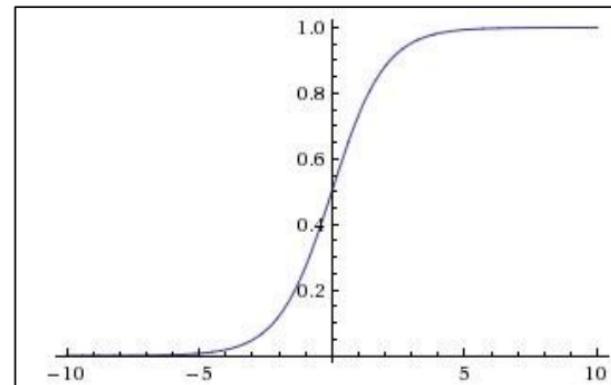
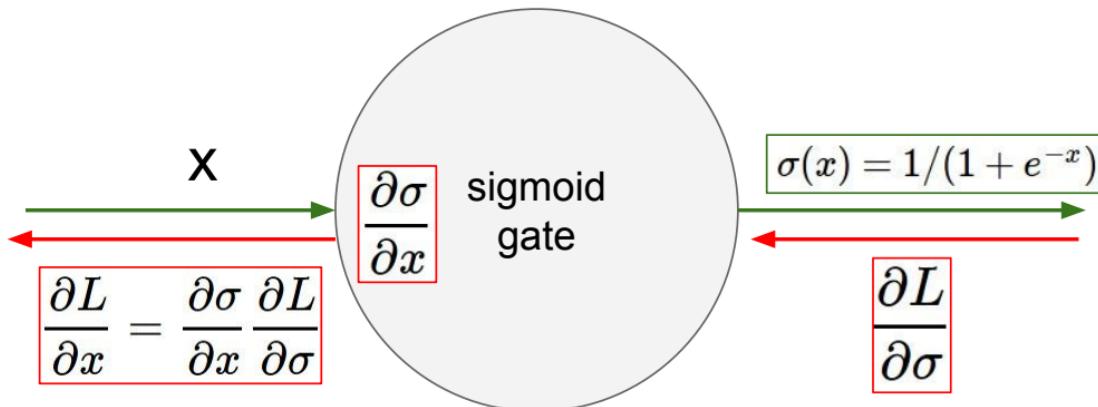


$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$



What happens when $x = -10$?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

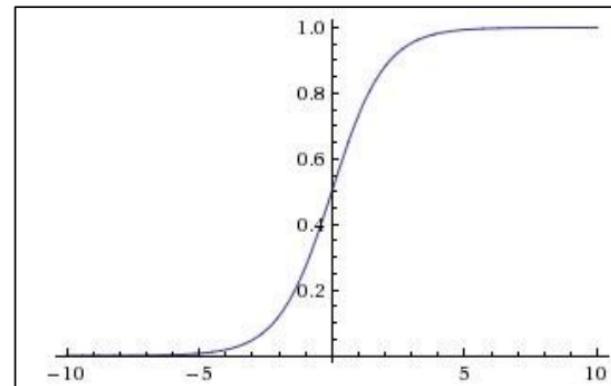
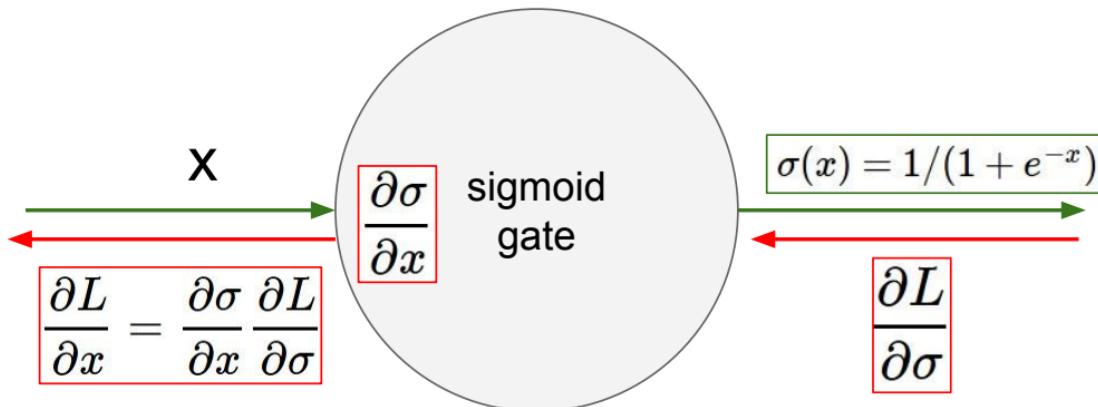


What happens when $x = -10$?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

$$\sigma(x) = \sim 0$$

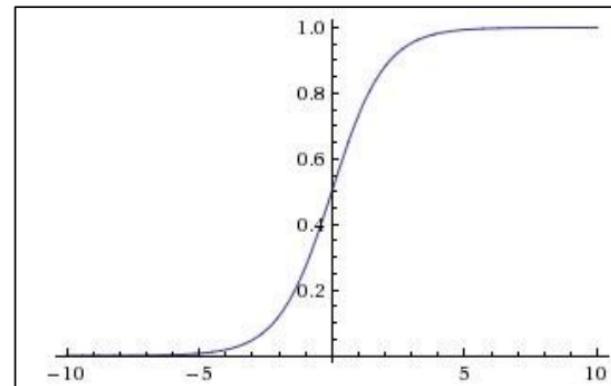
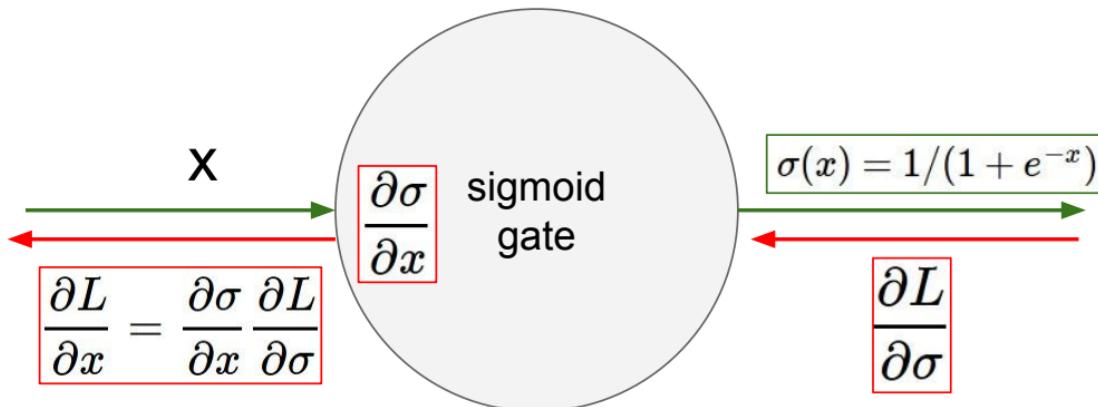
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)) = 0(1 - 0) = 0$$



What happens when $x = -10$?

What happens when $x = 0$?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$



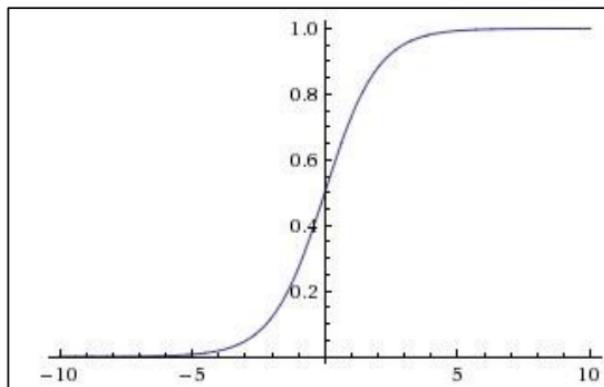
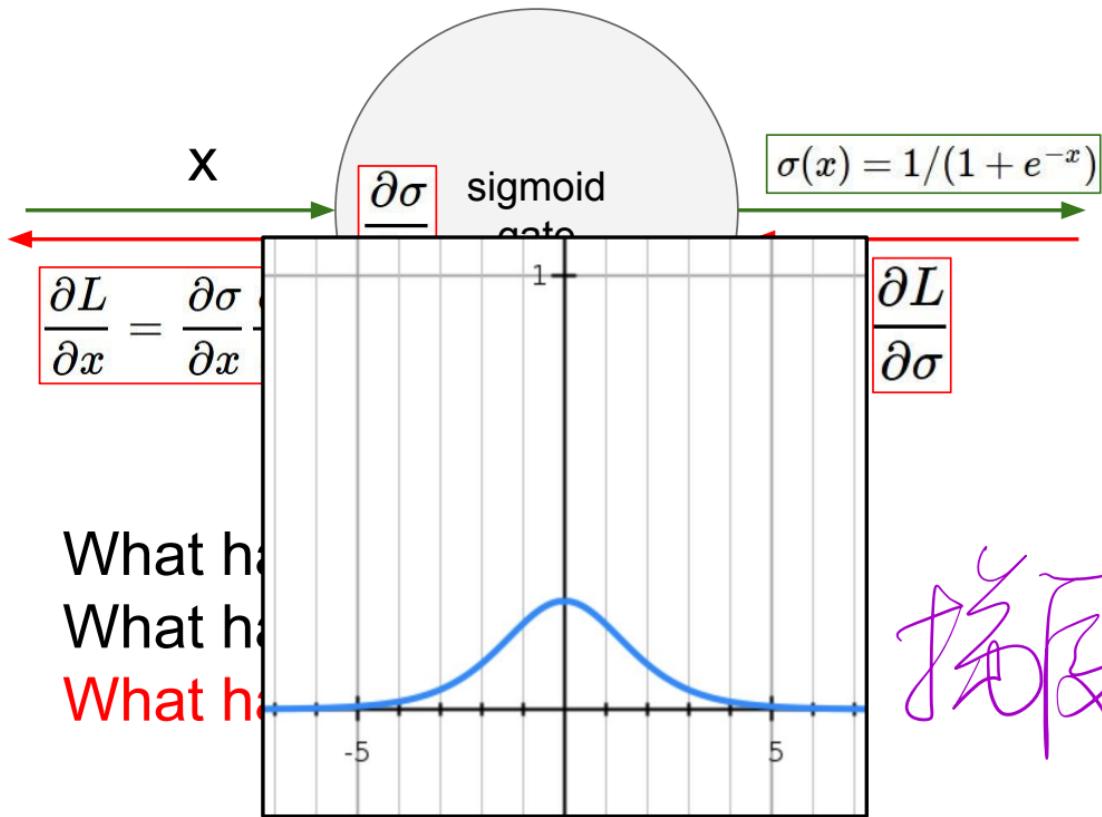
What happens when $x = -10$?

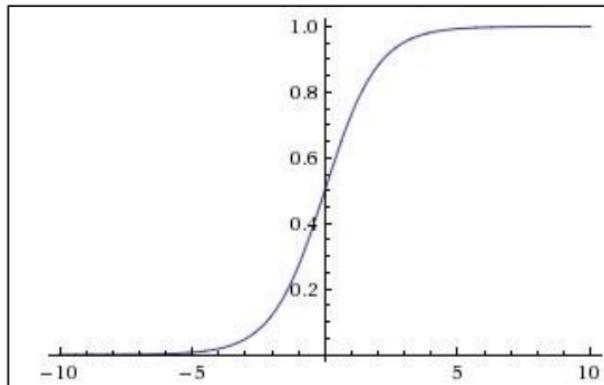
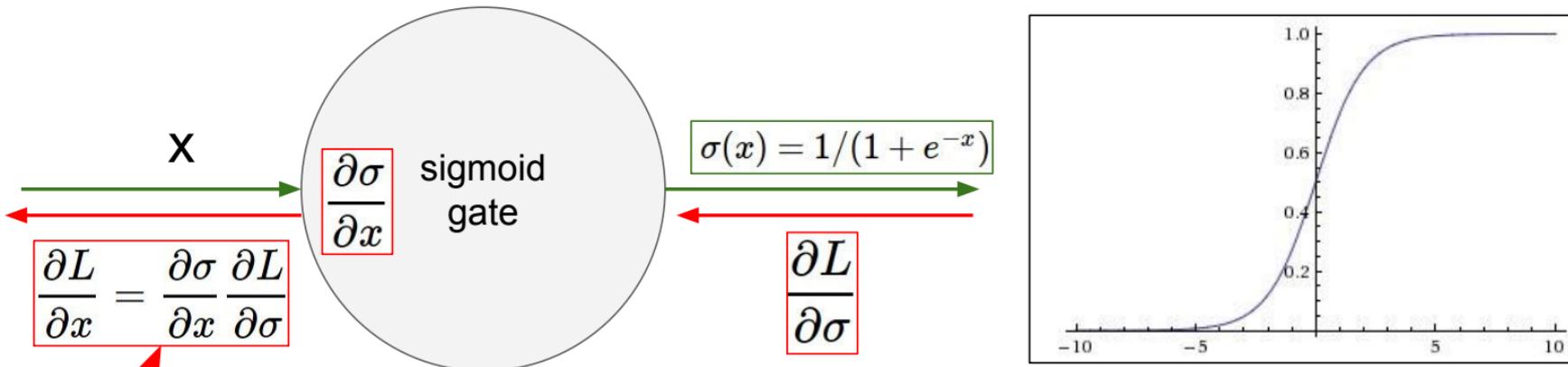
What happens when $x = 0$?

What happens when $x = 10$?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

$$\sigma(x) = \sim 1 \quad \frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)) = 1(1 - 1) = 0$$





Why is this a problem?

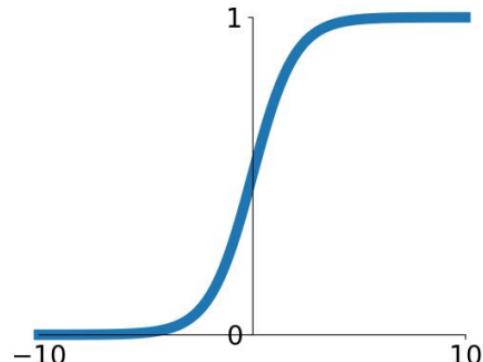
If all the gradients flowing back will be zero and weights will never change

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

Activation Functions

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

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



Sigmoid

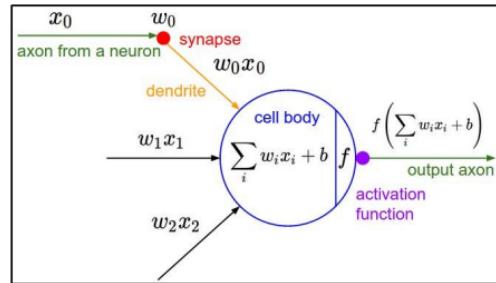
3 problems:

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

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

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

L2

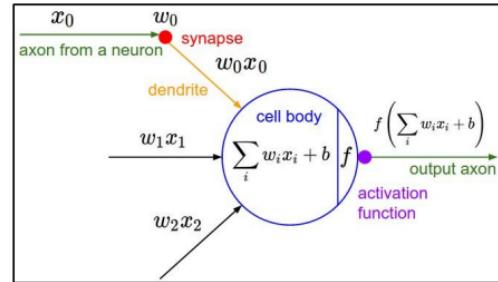


What can we say about the gradients on w ?

逐层正向传播，~~反向传播~~ upstream

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

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

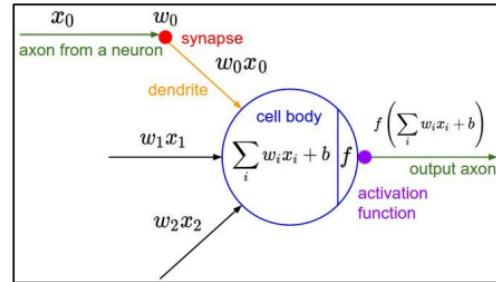


What can we say about the gradients on w ?

$$\frac{\partial L}{\partial w} = \sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))x \times \text{upstream_gradient}$$

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

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



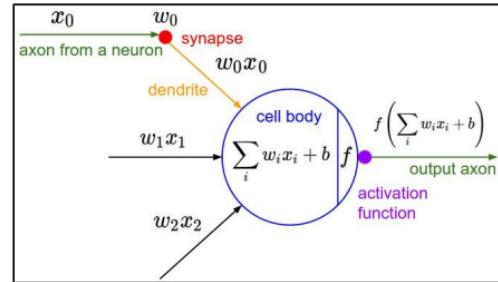
What can we say about the gradients on w ?

We know that local gradient of sigmoid is always positive

$$\frac{\partial L}{\partial w} = \boxed{\sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))x \times \text{upstream_gradient}}$$

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 ?

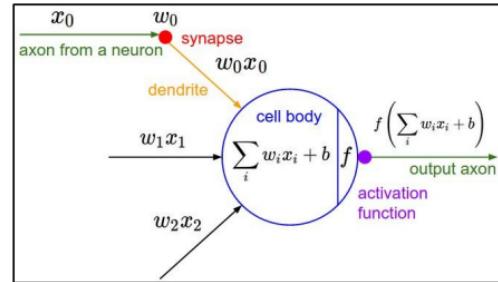
We know that local gradient of sigmoid is always positive

We are assuming x is always positive

$$\frac{\partial L}{\partial w} = \boxed{\sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))x} \times \text{upstream_gradient}$$

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 ?

We know that local gradient of sigmoid is always positive

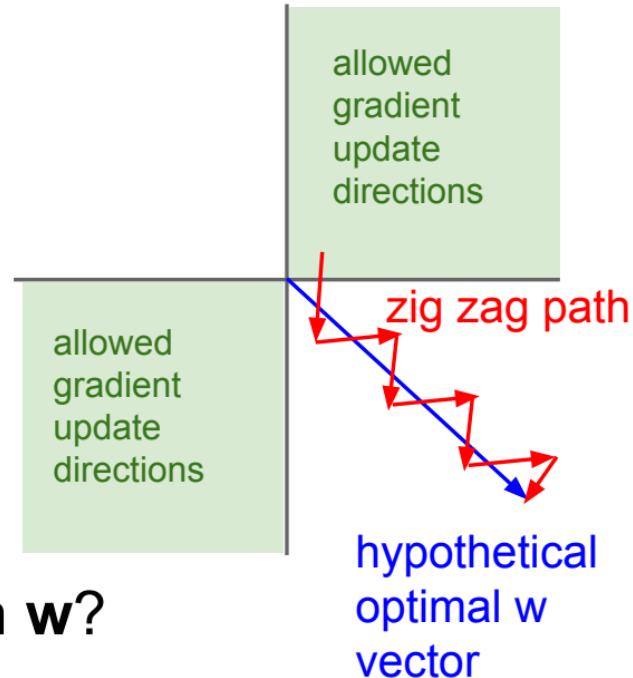
We are assuming x is always positive

So!! Sign of gradient for all w_i is the same as the sign of upstream scalar gradient!

$$\frac{\partial L}{\partial w} = \sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))x \times \text{upstream_gradient}$$

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 :(

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

如果所有的输入都是正的，那么
会有什么样的梯度呢

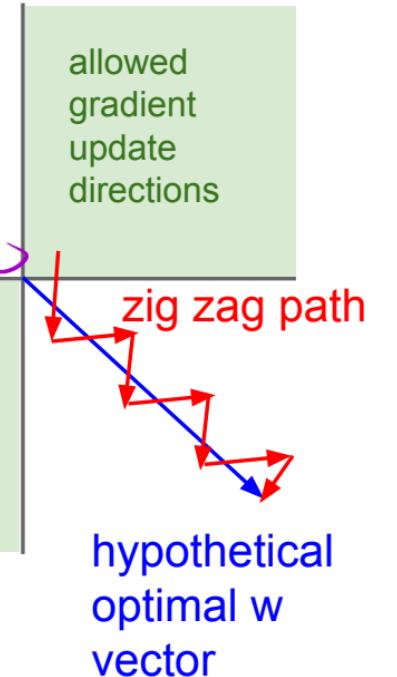
$$f\left(\sum_i w_i x_i + b\right)$$

但是若 sigmoid
函数上一级神经元, 那么下一级
神经元将根据输入值进行加工了

What can we say about the gradients on w ?

Always all positive or all negative :(

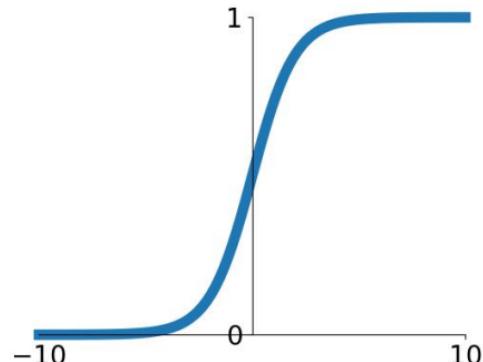
(For a single element! Minibatches help)



Activation Functions

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

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

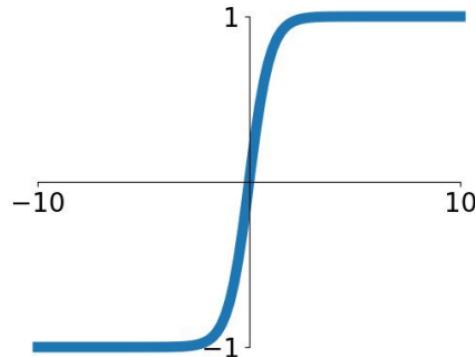


Sigmoid

3 problems:

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

Activation Functions



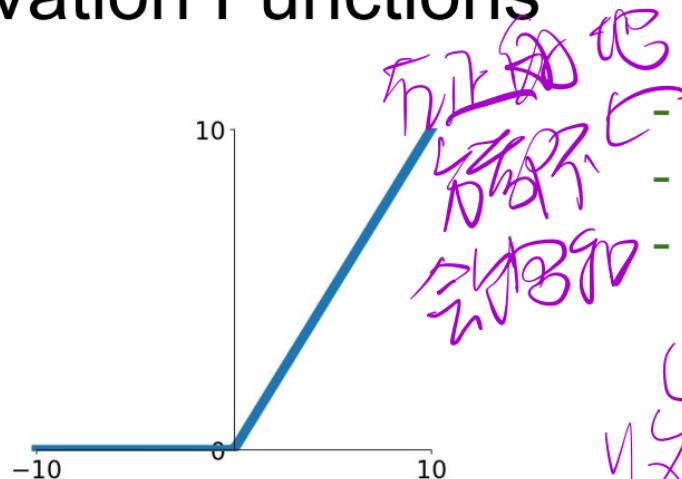
tanh(x)

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

[LeCun et al., 1991]

Activation Functions

- Computes $f(x) = \max(0, x)$



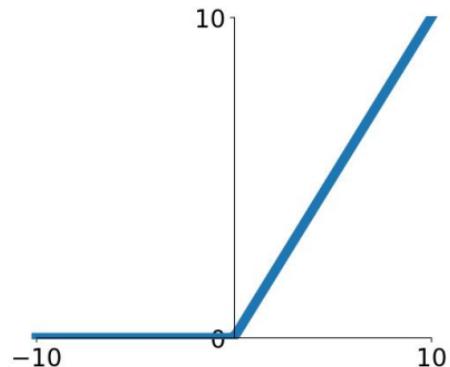
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

ReLU
(Rectified Linear Unit)

无生生物活性

[Krizhevsky et al., 2012]

Activation Functions

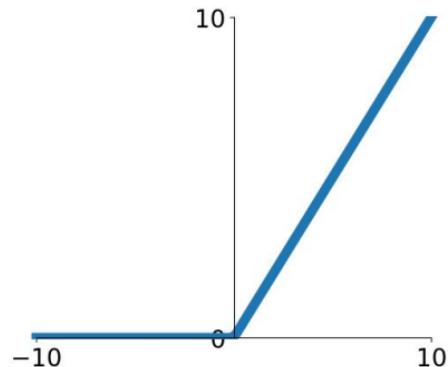


ReLU
(Rectified Linear Unit)

- 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

Activation Functions

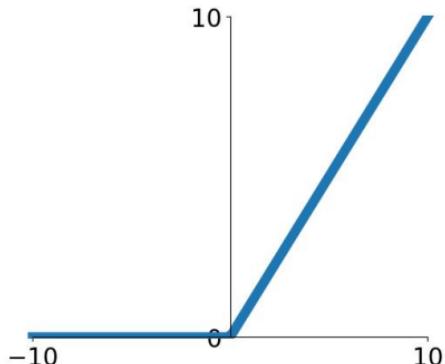
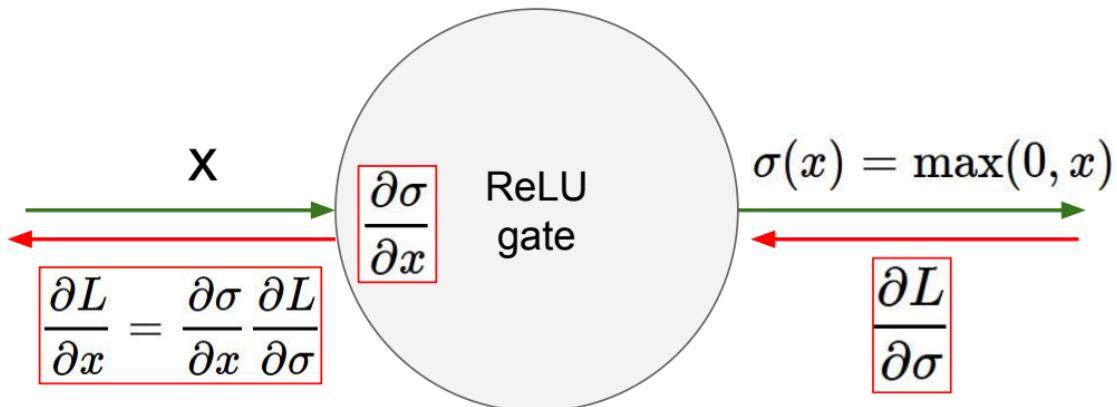


ReLU
(Rectified Linear Unit)

- 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 $x < 0$?



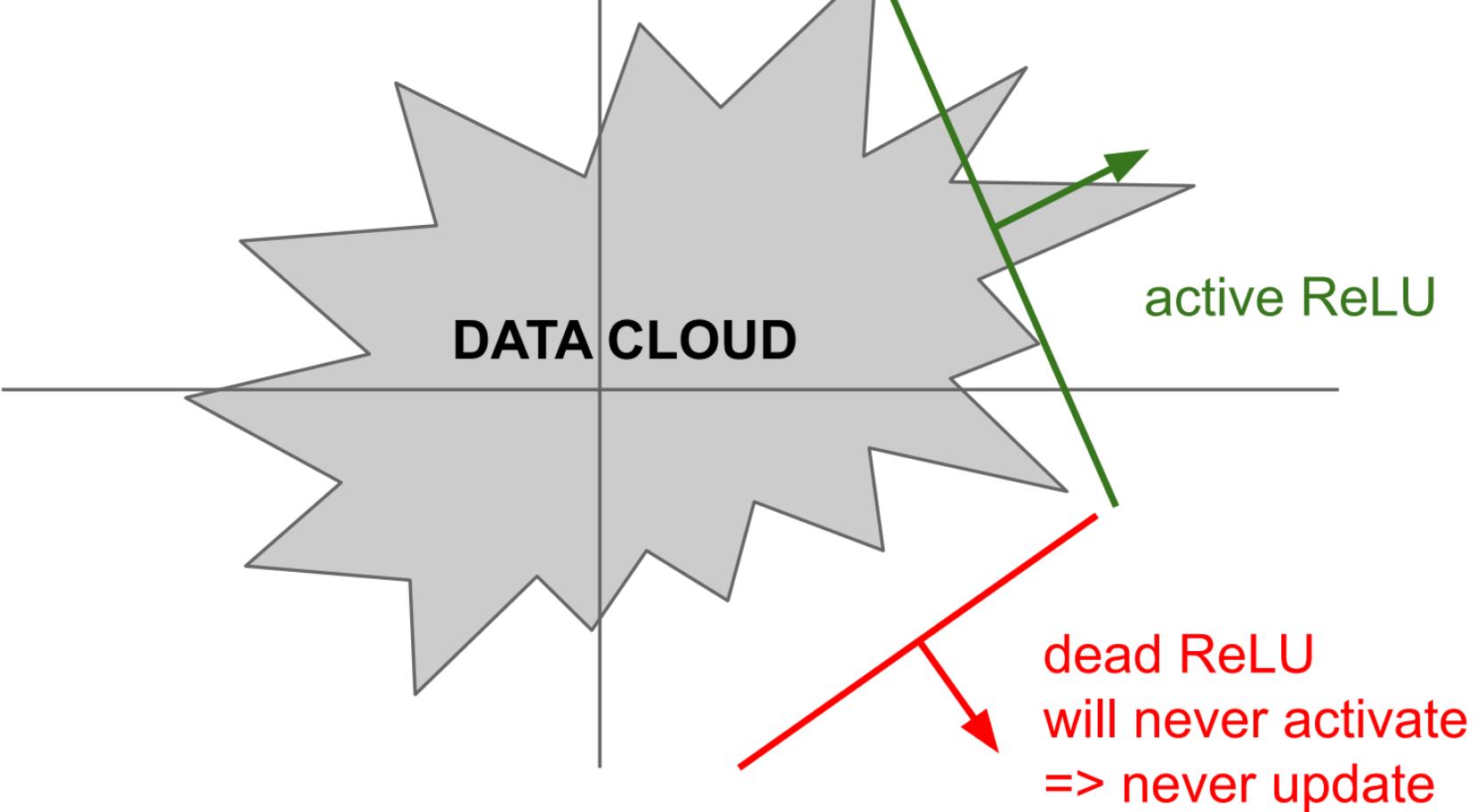
What happens when $x = -10$?

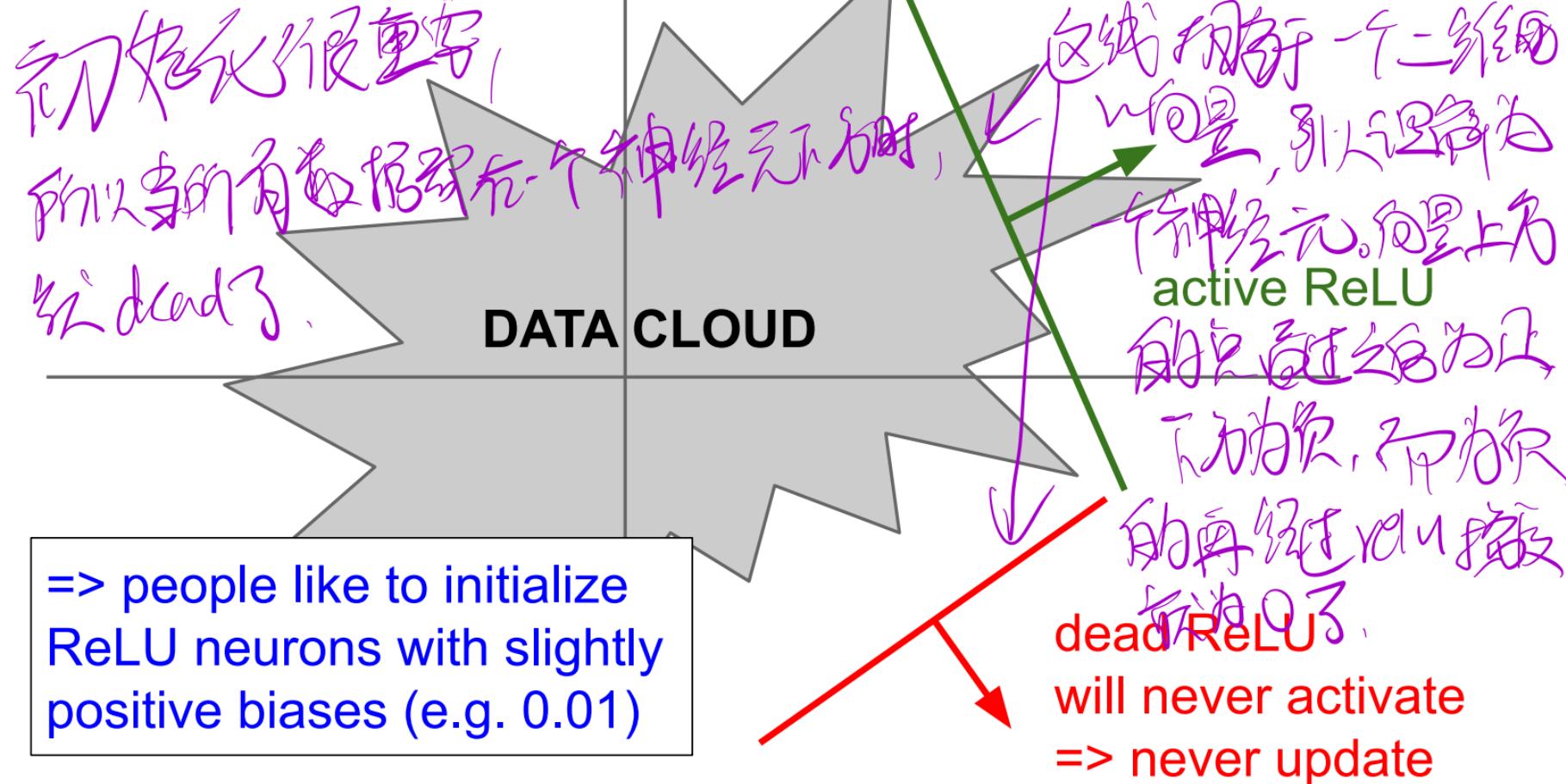
What happens when $x = 0$?

What happens when $x = 10$?

Handwritten notes in purple ink:

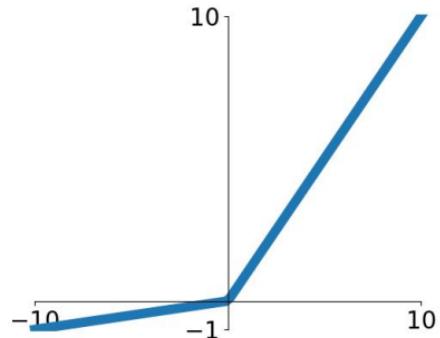
- At $x = -10$, the value is 0.
- At $x = 0$, the value is 0.
- At $x = 10$, the value is 10.





Activation Functions

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



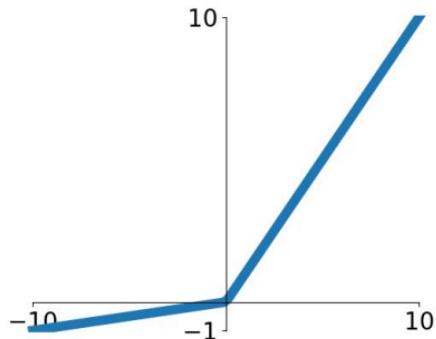
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

Leaky ReLU

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

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”.**

Parametric Rectifier (PReLU)

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

backprop into α (parameter)

3326
3317

Activation Functions

[Clevert et al., 2015]

More Robust Exponential Linear Units (ELU)



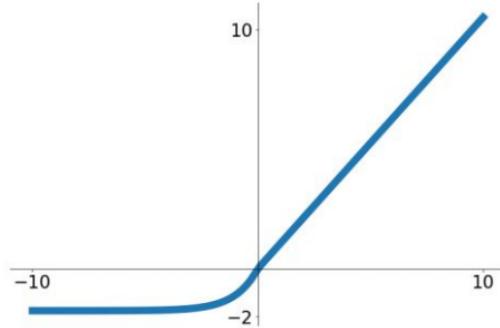
- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

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

(Alpha default = 1)

- Computation requires `exp()`

Scaled Exponential Linear Units (SELU)



- Scaled version of ELU that works better for deep networks
- “Self-normalizing” property;
- Can train deep SELU networks without BatchNorm

$$f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha(e^x - 1) & \text{otherwise} \end{cases}$$

$$\alpha = 1.6732632423543772848170429916717$$

$$\lambda = 1.0507009873554804934193349852946$$

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!

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

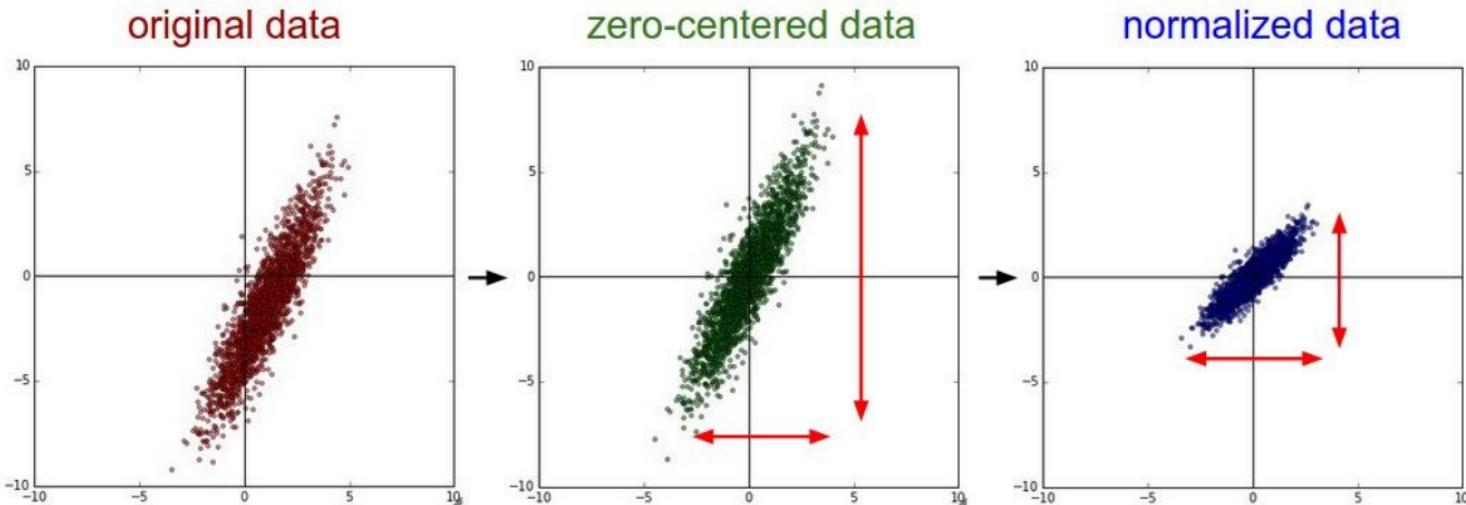
Problem: doubles the number of parameters/neuron :(

TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU / SELU
 - To squeeze out some marginal gains
- Don't use sigmoid or tanh

Data Preprocessing

Data Preprocessing



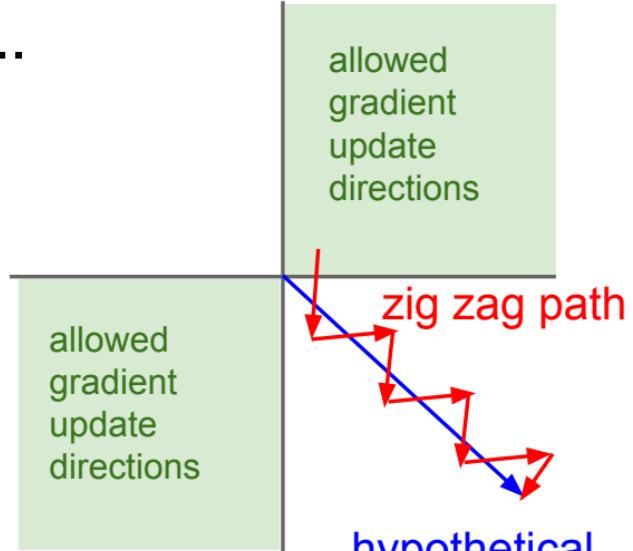
```
X -= np.mean(X, axis = 0)
```

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

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

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

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



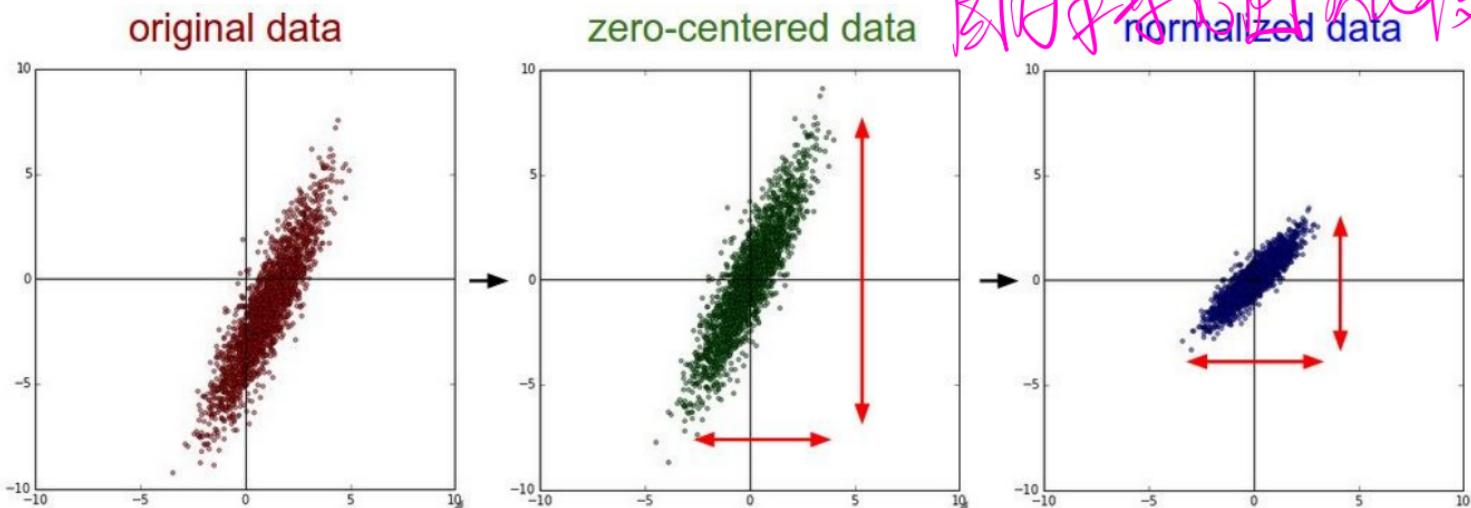
What can we say about the gradients on w ?

Always all positive or all negative :(

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

Data Preprocessing

数据预处理不带高斯化，
由特征归一化到数据



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

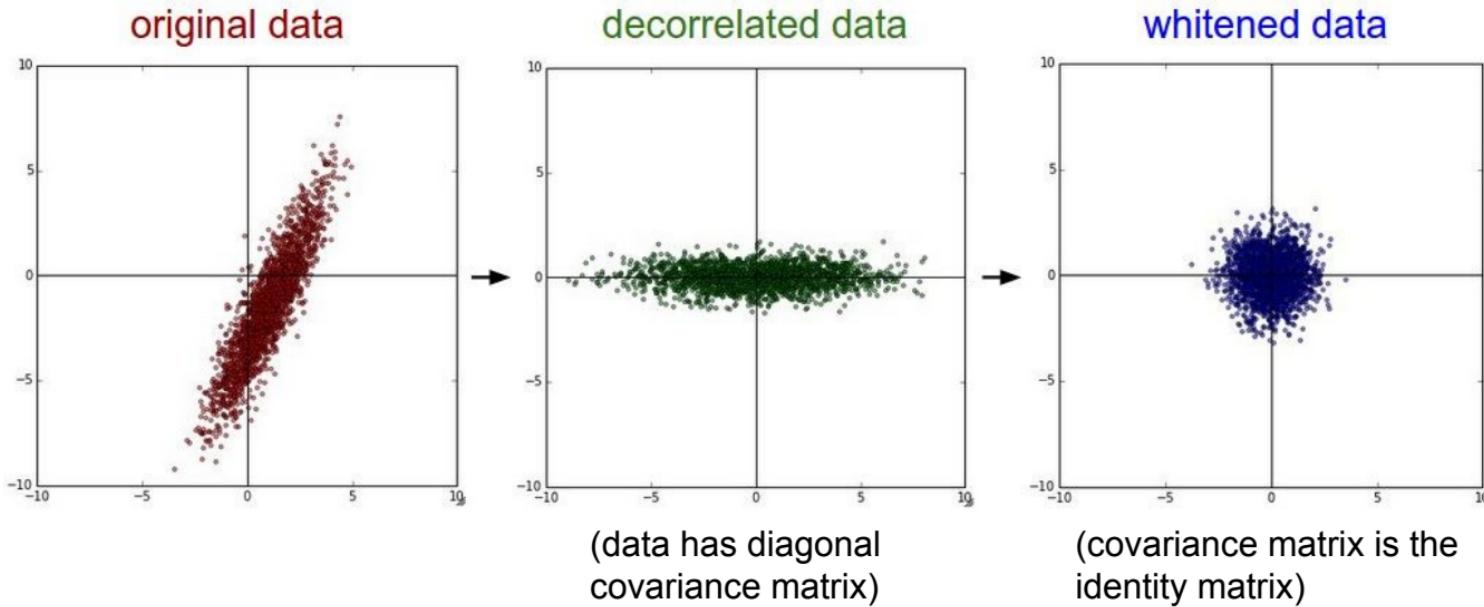
```
X /= np.std(X, axis = 0)
```

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

数据预处理：空间直角坐标系，不同归一化

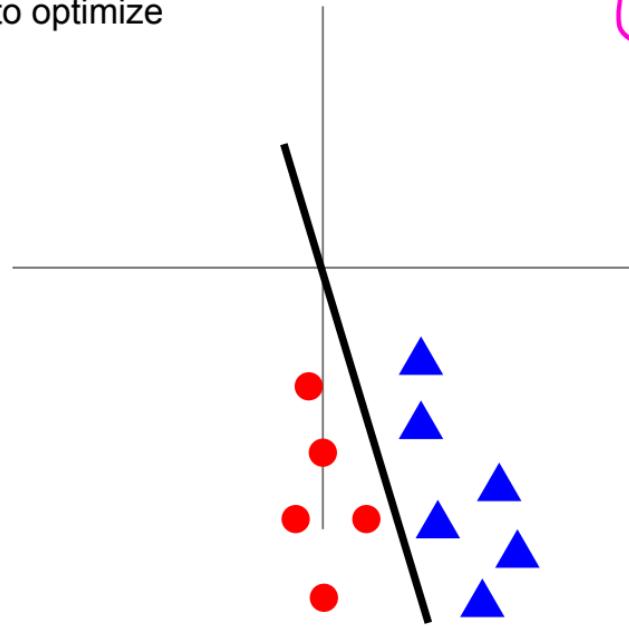
Data Preprocessing

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

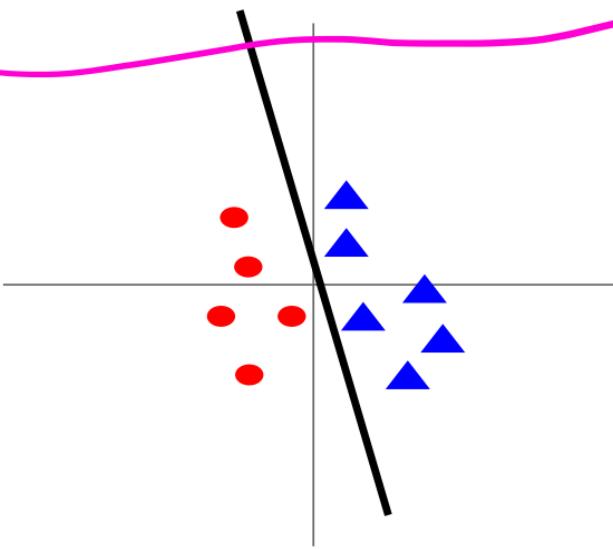


Data Preprocessing

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



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



TLDR: In practice for Images: center only

~~3/4 of the time~~

在大部分时间

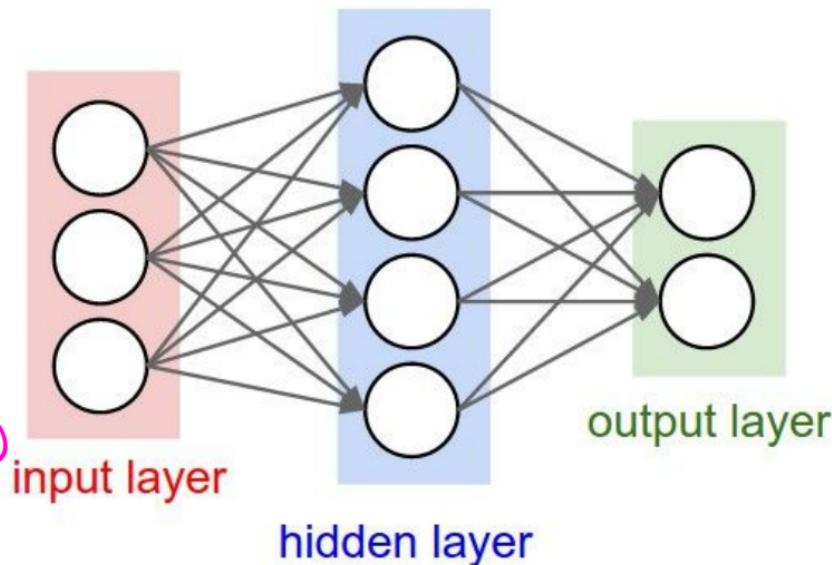
e.g. consider CIFAR-10 example with [32,32,3] images

- 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)
- Subtract per-channel mean and
Divide by per-channel std (e.g. ResNet)
(mean along each channel = 3 numbers)

Not common
to do PCA or
whitening

Weight Initialization

- Q: what happens when $W=\text{constant init}$ is used?



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

```
W = 0.01 * np.random.randn(Din, Dout)
```

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

```
W = 0.01 * np.random.randn(Din, Dout)
```

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

Weight Initialization: Activation statistics

```
dims = [4096] * 7      Forward pass for a 6-layer  
hs = []                  net with hidden size 4096  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.01 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

What will happen to the activations for the last layer?

X 线的 [6x 996]

Weight Initialization: Activation statistics

```
dims = [4096] * 7      Forward pass for a 6-layer  
hs = []                net with hidden size 4096  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.01 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

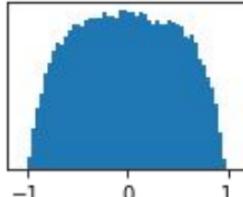
All activations tend to zero
for deeper network layers

Q: What do the gradients
dL/dW look like?

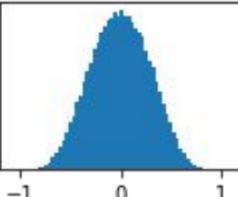
$$W = \text{6x 996}$$

前面几层的 W 值会变小
直到接近 0

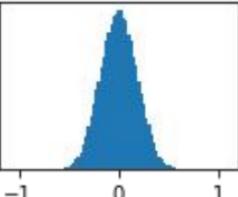
Layer 1
mean=-0.00
std=0.49



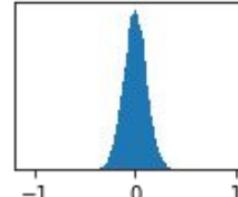
Layer 2
mean=0.00
std=0.29



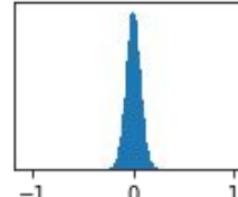
Layer 3
mean=0.00
std=0.18



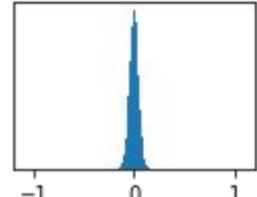
Layer 4
mean=-0.00
std=0.11



Layer 5
mean=-0.00
std=0.07



Layer 6
mean=0.00
std=0.05



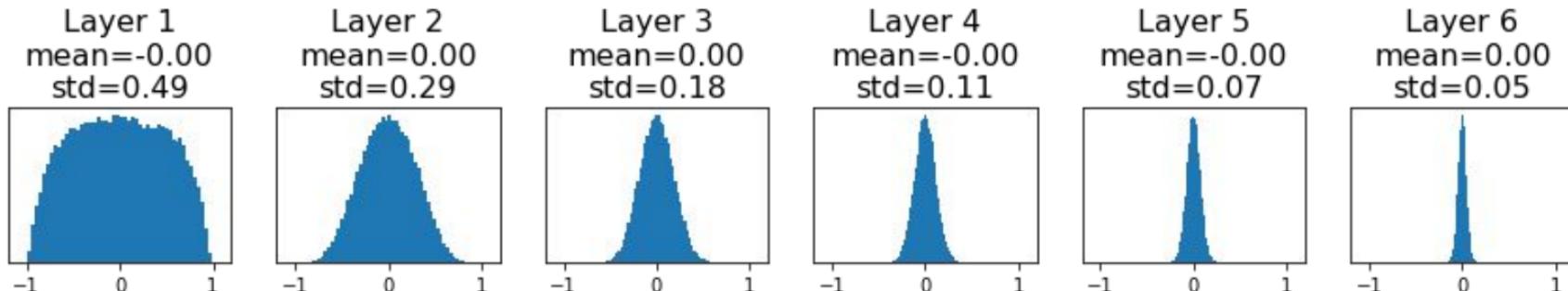
Weight Initialization: Activation statistics

```
dims = [4096] * 7      Forward pass for a 6-layer  
hs = []                  net with hidden size 4096  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.01 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?

A: All zero, no learning =(



Weight Initialization: Activation statistics

```
dims = [4096] * 7    Increase std of initial  
hs = []              weights from 0.01 to 0.05  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.05 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

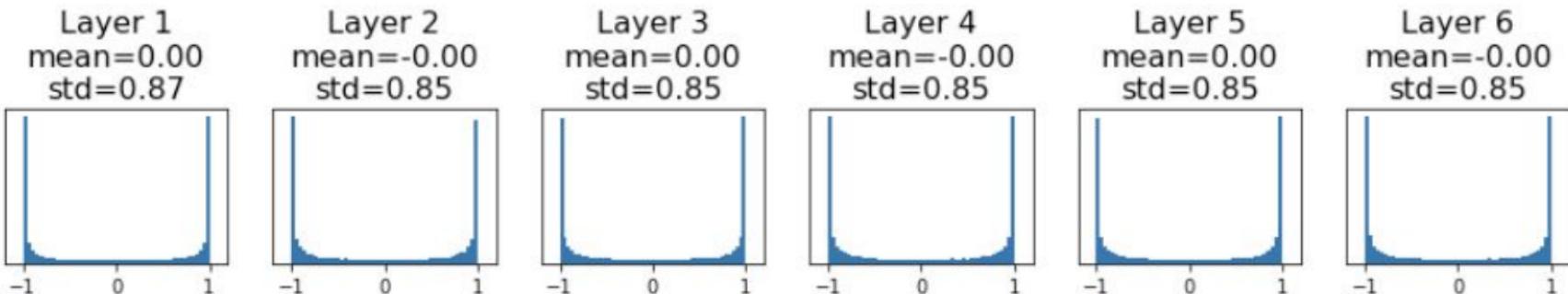
What will happen to the activations for the last layer?

Weight Initialization: Activation statistics

```
dims = [4096] * 7    Increase std of initial  
hs = []                weights from 0.01 to 0.05  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.05 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

All activations saturate

Q: What do the gradients look like?



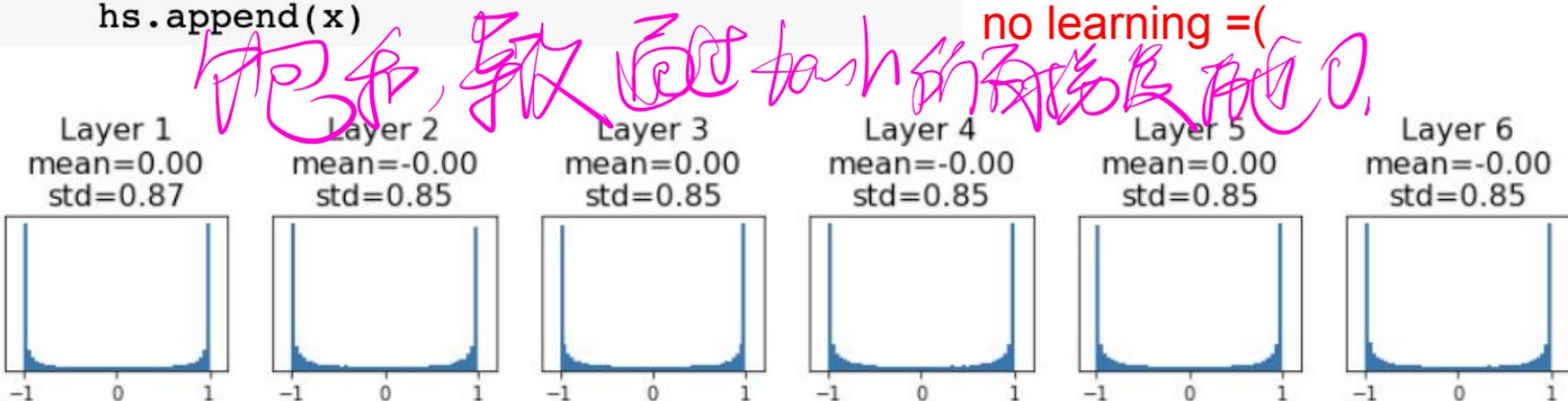
Weight Initialization: Activation statistics

```
dims = [4096] * 7    Increase std of initial  
hs = []                weights from 0.01 to 0.05  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:1], dims[1:]):  
    W = 0.05 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero,
no learning =(



Weight Initialization: “Xavier” Initialization

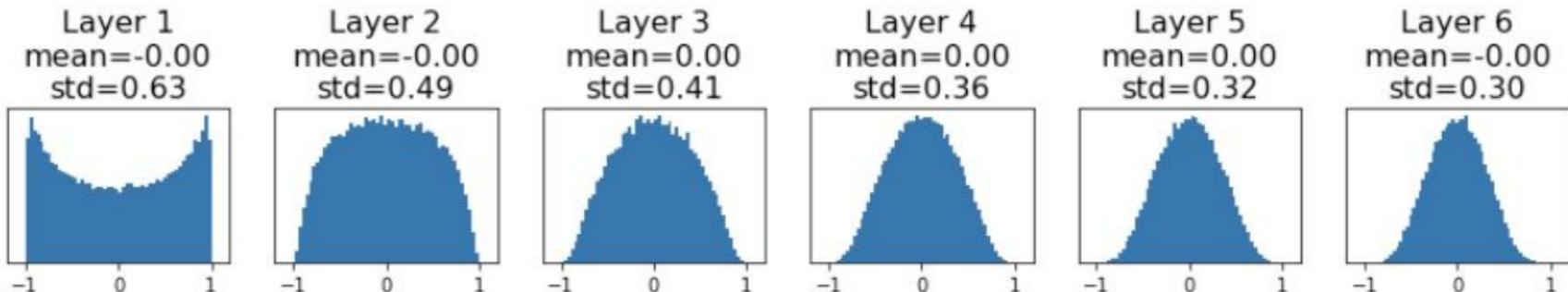
```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!



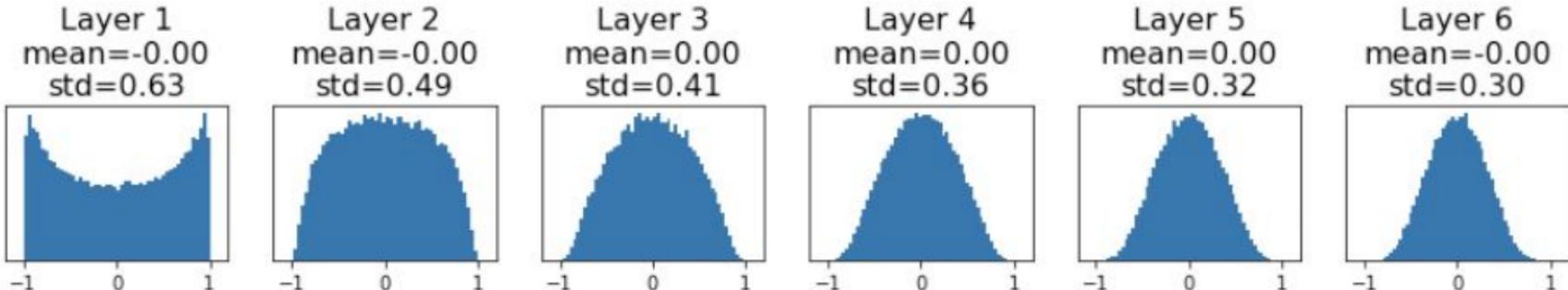
Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is $\text{filter_size}^2 * \text{input_channels}$



Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is filter_size² * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din}$

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is filter_size² * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{Din})$

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{Din})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is filter_size² * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{Din})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

$$\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din})$$

[substituting value of y]

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is filter_size² * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{Din})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

$$\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din}) \\ = \text{Din } \text{Var}(x_i w_i)$$

[Assume all x_i, w_i are iid]

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is filter_size² * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{Din})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

$$\begin{aligned}\text{Var}(y) &= \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din}) \\ &= \text{Din} \text{Var}(x_i w_i) \\ &= \text{Din} \text{Var}(x_i) \text{Var}(w_i)\end{aligned}$$

[Assume all x_i, w_i are zero mean]

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is filter_size² * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{Din})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

$$\begin{aligned}\text{Var}(y) &= \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{Din} w_{Din}) \\ &= \text{Din} \text{Var}(x_i w_i) \\ &= \text{Din} \text{Var}(x_i) \text{Var}(w_i)\end{aligned}$$

[Assume all x_i, w_i are iid]

So, $\text{Var}(y) = \text{Var}(x_i)$ only when $\text{Var}(w_i) = 1/\text{Din}$

Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: What about ReLU?

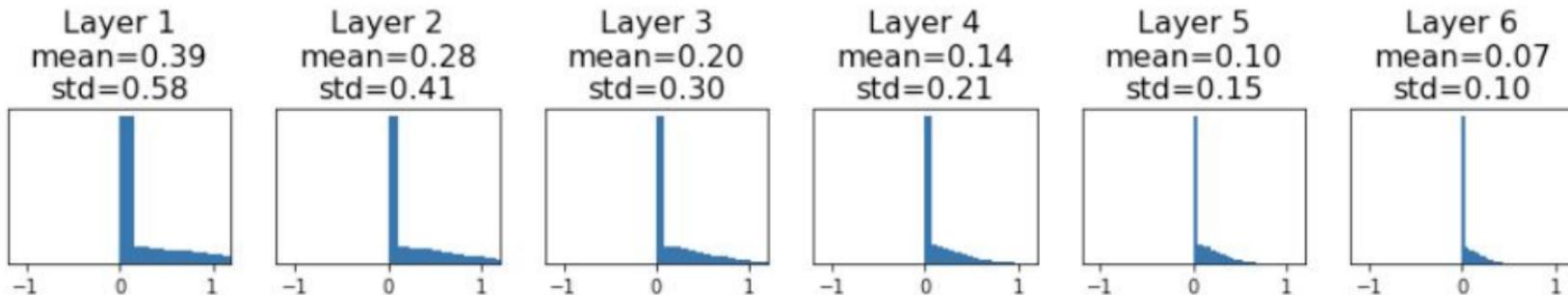
```
dims = [4096] * 7      Change from tanh to ReLU
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Weight Initialization: What about ReLU?

Yelin 余林会做 ReLU - 举
dims = [4096] * 7 Change from tanh to ReLU
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
 W = np.random.randn(Din, Dout) / np.sqrt(Din)
 x = np.maximum(0, x.dot(W))
 hs.append(x)

Xavier assumes zero centered activation function

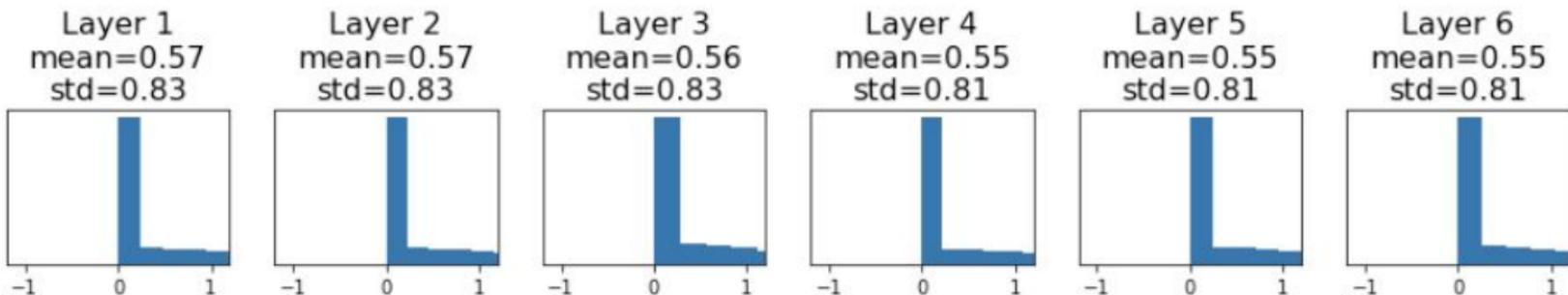
Activations collapse to zero again, no learning =(



Weight Initialization: Kaiming / MSRA Initialization

```
dims = [4096] * 7  ReLU correction: std = sqrt(2 / Din)
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

“Just right”: Activations are nicely scaled for all layers!



He et al, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, ICCV 2015

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

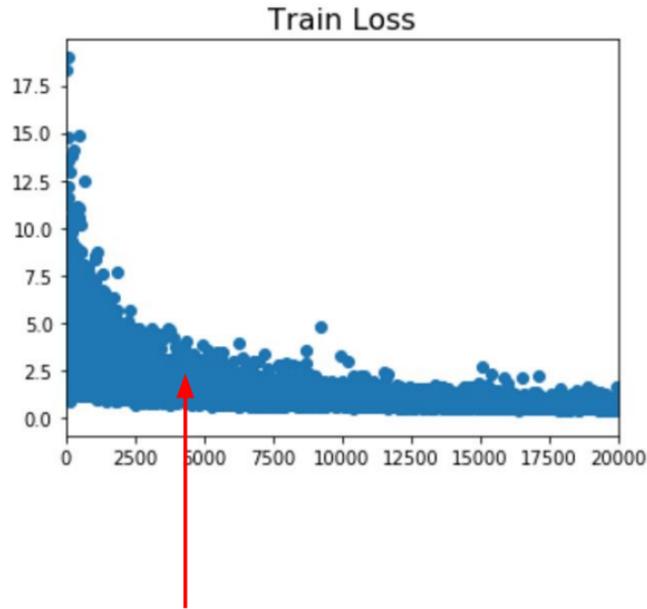
All you need is a good init, Mishkin and Matas, 2015

Fixup Initialization: Residual Learning Without Normalization, Zhang et al, 2019

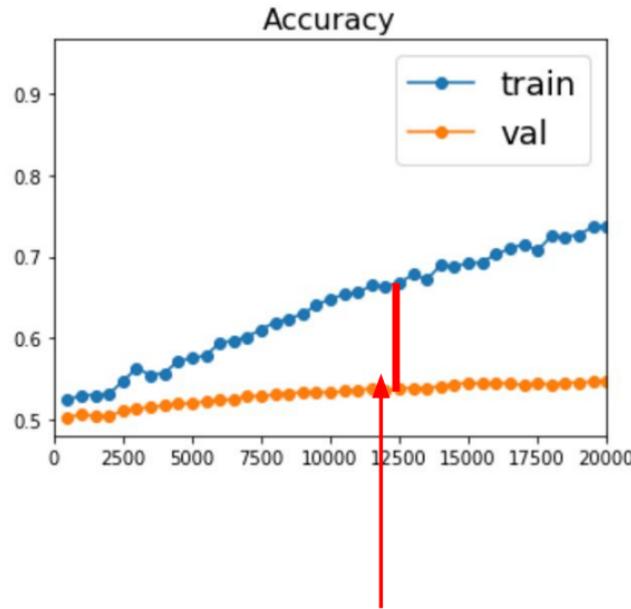
The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin, 2019

Training vs. Testing Error

Beyond Training Error

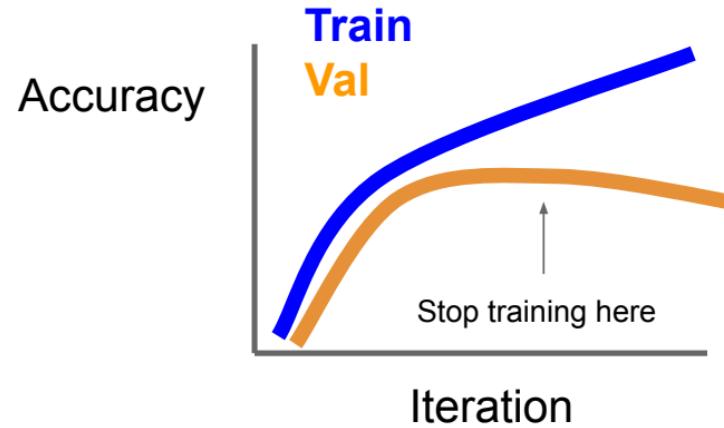
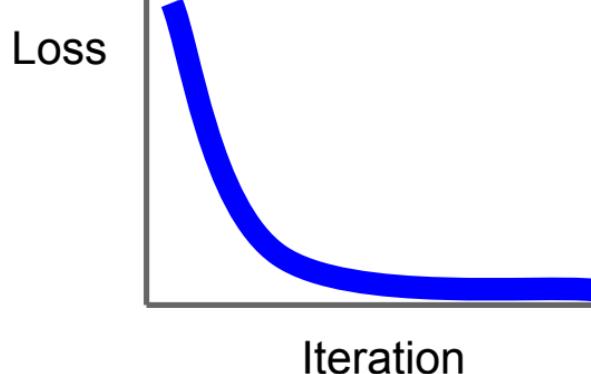


Better optimization algorithms help reduce training loss



But we really care about error on new data - how to reduce the gap?

Early Stopping: Always do this



Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot
that worked best on val

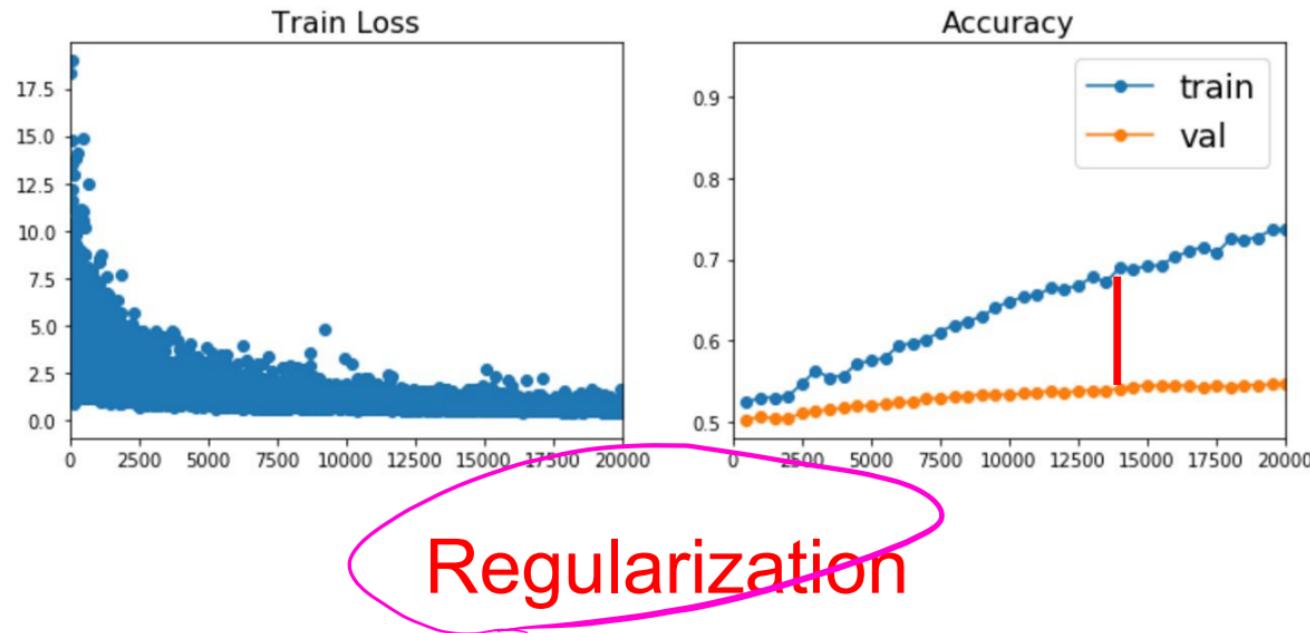
Model Ensembles

1. Train multiple independent models
2. At test time average their results

(Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

How to improve single-model performance?



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

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

L1 regularization

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

Elastic net (L1 + L2)

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

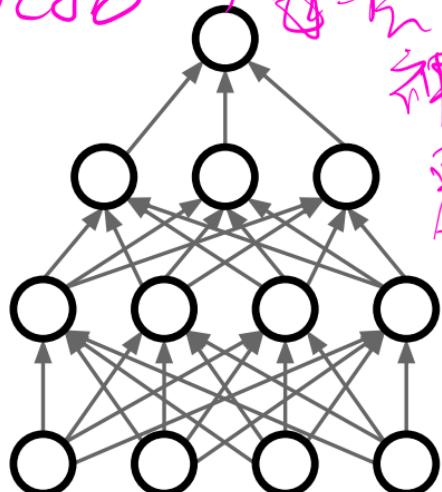
Regularization: Dropout

dropout 和 P 是停用和概率

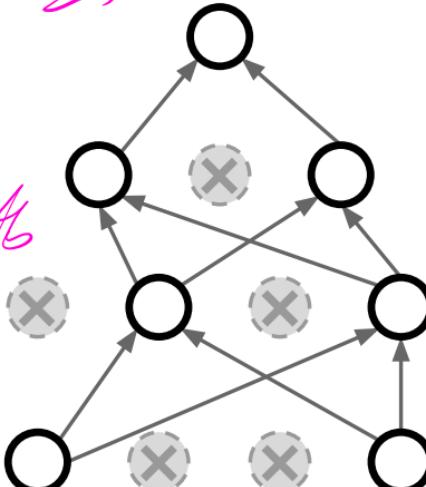
In each forward pass, randomly set some neurons to zero

Probability of dropping is a hyperparameter; 0.5 is common

前向传播时，随机选取一些神经元置0。



只将激活的神经元值置0。
前向传播时，随机选取一些神经元置0。



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

Regularization: Dropout

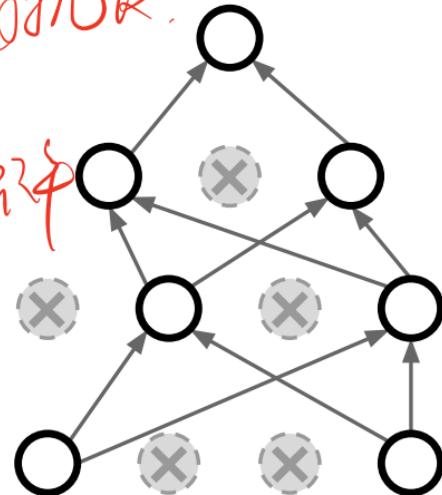
```
p = 0.5 # probability of keeping a unit active. higher = less dropout  
  
def train_step(X):  
    """ X contains the data """  
  
    # forward pass for example 3-layer neural network  
    H1 = np.maximum(0, np.dot(W1, X) + b1)  
    U1 = np.random.rand(*H1.shape) < p # first dropout mask  
    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)
```

np.random.rand, 0-1均匀分布。随机数，0-1之间。

当 $U < p$, 为真，
否则为假。

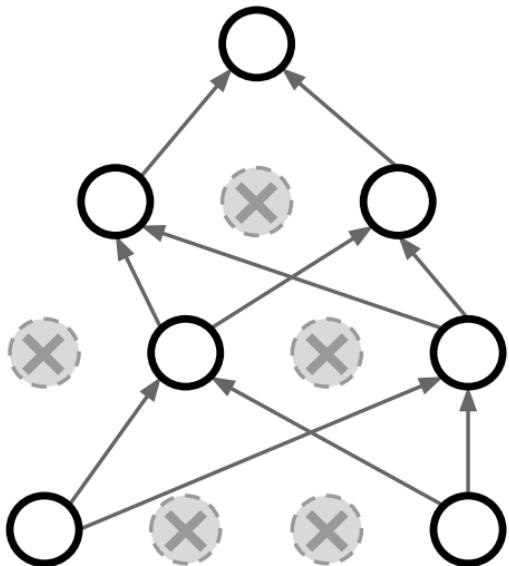
随机，生成
从 P 为真
随机，
false.

Example forward pass with a 3-layer network using dropout



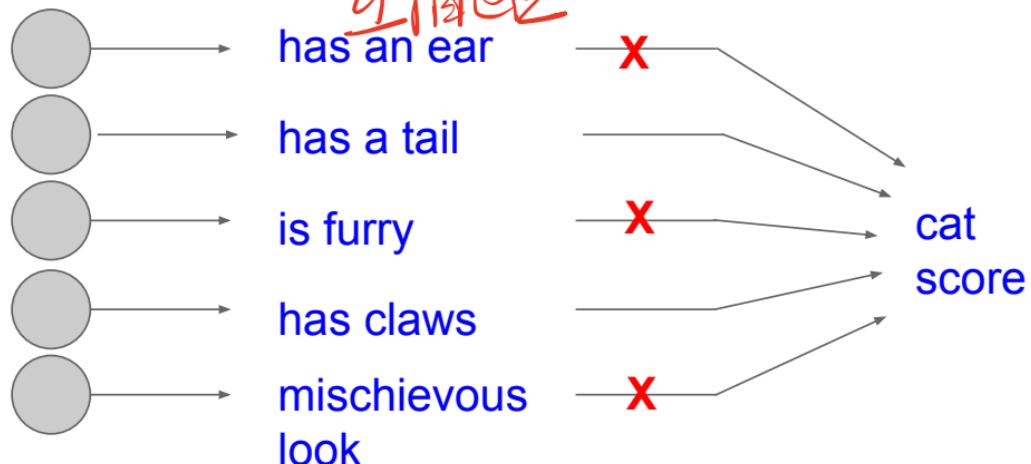
Regularization: Dropout

How can this possibly be a good idea?



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

冗余
互相适应



Regularization: Dropout

How can this possibly be a good idea?

神经元随机失活
相当于做了很多模型

Another interpretation:

FPB

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!

$$\boxed{y} = f_W(\boxed{x}, \boxed{z})$$

Output (label) Input (image) Random mask

Want to “average out” the randomness at test-time

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

But this integral seems hard ...

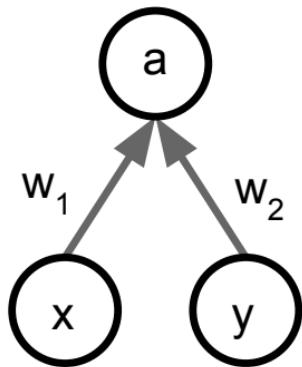
完
成

Dropout: Test time

Want to approximate
the integral

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

Consider a single neuron.

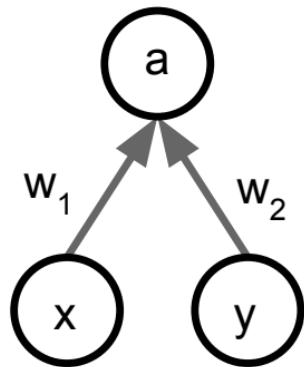


Dropout: Test time

Want to approximate
the integral

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

Consider a single neuron.

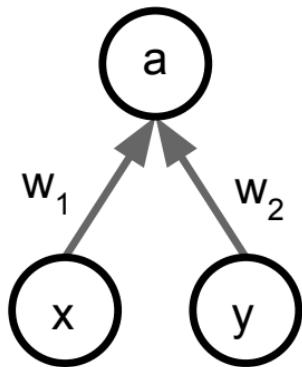


At test time we have: $E[a] = w_1x + w_2y$

Dropout: Test time

Want to approximate
the integral

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



Consider a single neuron.

At test time we have: $E[a] = w_1x + w_2y$

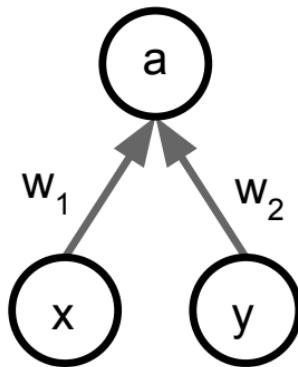
During training we have:

$$\begin{aligned}E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\&\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\&= \frac{1}{2}(w_1x + w_2y)\end{aligned}$$

Dropout: Test time

Want to approximate
the integral

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



Consider a single neuron.

At test time we have: $E[a] = w_1x + w_2y$

During training we have:

$$\begin{aligned}E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\&\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\&= \frac{1}{2}(w_1x + w_2y)\end{aligned}$$

At test time, multiply
by dropout probability

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

Dropout Summary

```
""" 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
```

drop in train time

scale at test time

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
```

右側標註：
左側標註：
紅色手寫：
這就是說，
在訓練時，
我們會隨機地
將某些神經元
的輸出設為零。
而在測試時，
我們不會改變
這些神經元的
輸出。

test time is unchanged!

Regularization: A common pattern

Training: Add some kind of randomness

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

Testing: Average out randomness (sometimes approximate)

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

Stochastic

Regularization: A common pattern

Training: Add some kind of randomness

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

Testing: Average out randomness (sometimes approximate)

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

但 dropout 从属于随机正则化的方法

随机正则化
随机权重

在训练时引入随机性
性，测试时消除随机性

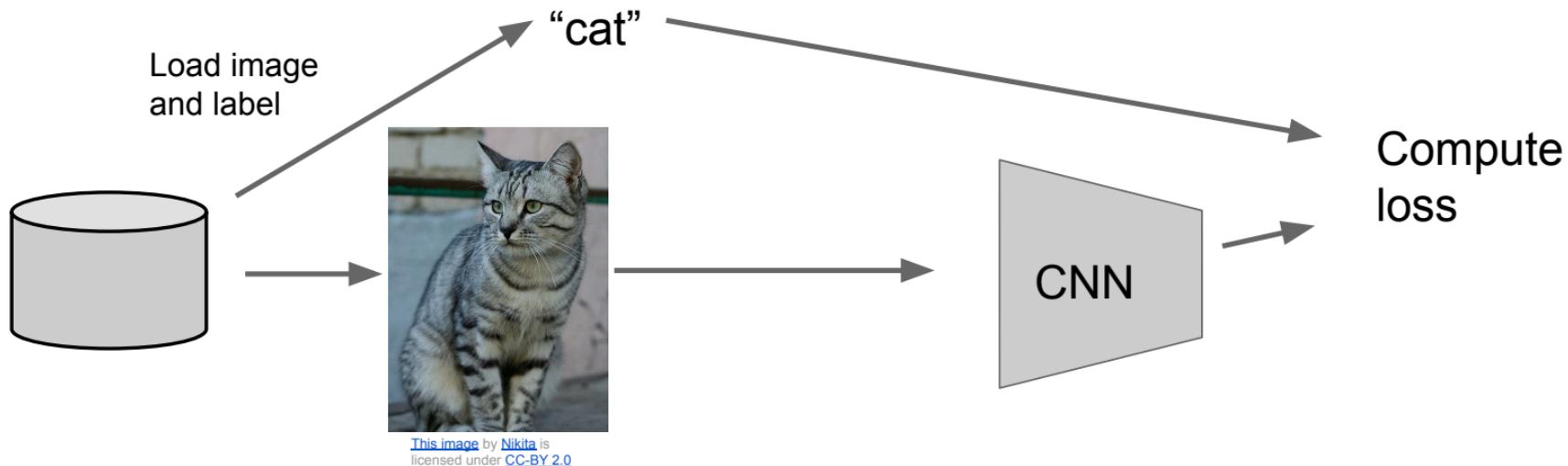
Example: Batch Normalization

Training: Normalize using stats from random minibatches

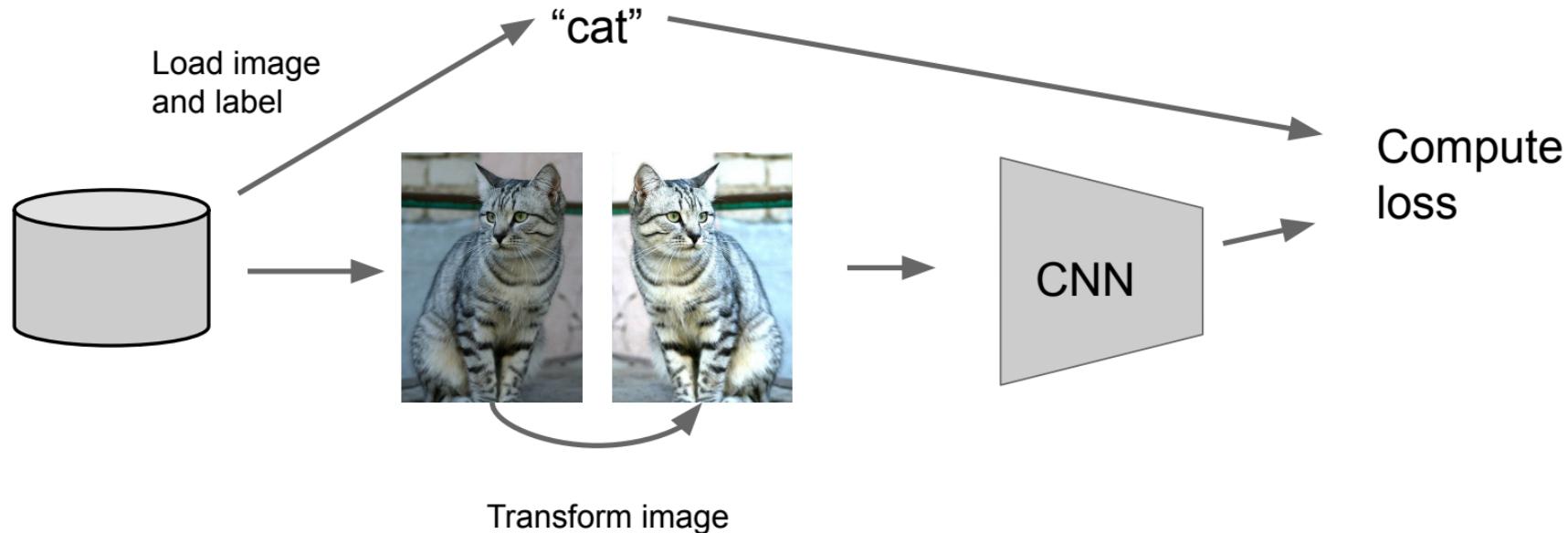
Testing: Use fixed stats to normalize

Regularization: Data Augmentation

数据增强

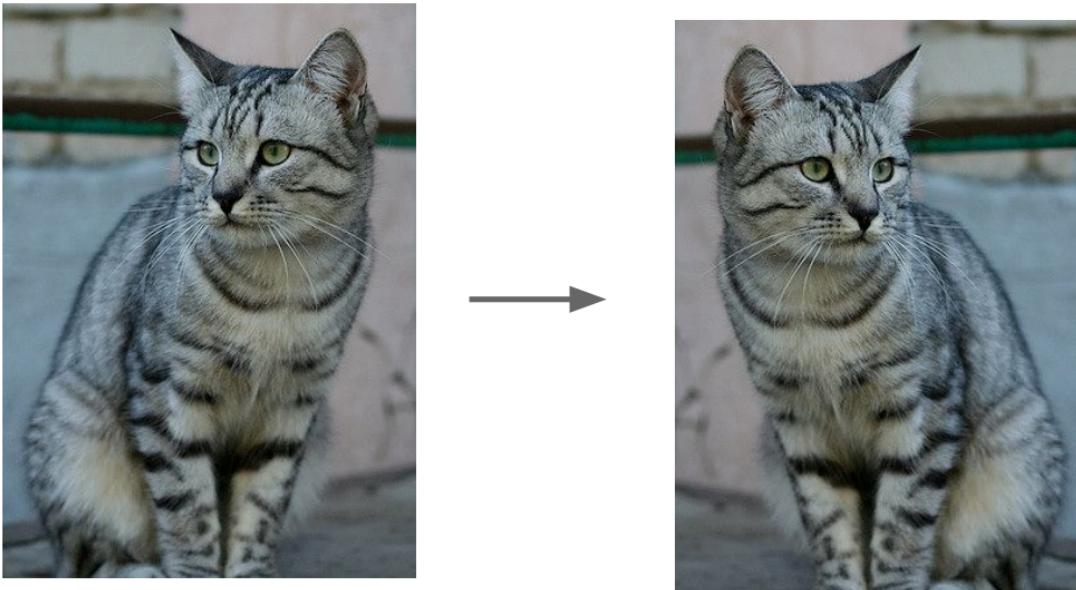


Regularization: Data Augmentation



Data Augmentation

Horizontal Flips



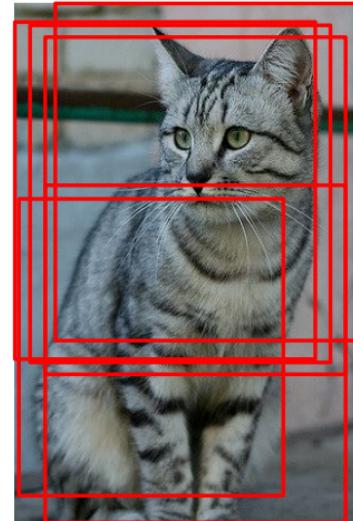
Data Augmentation

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 224×224 patch



Data Augmentation

Random crops and scales

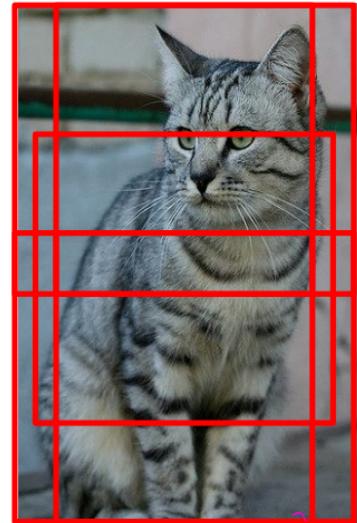
Training: sample random crops / scales

ResNet:

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

怎样，
这样

训练时随机采样
的随机裁剪



Testing: average a fixed set of crops

测试时固定剪裁

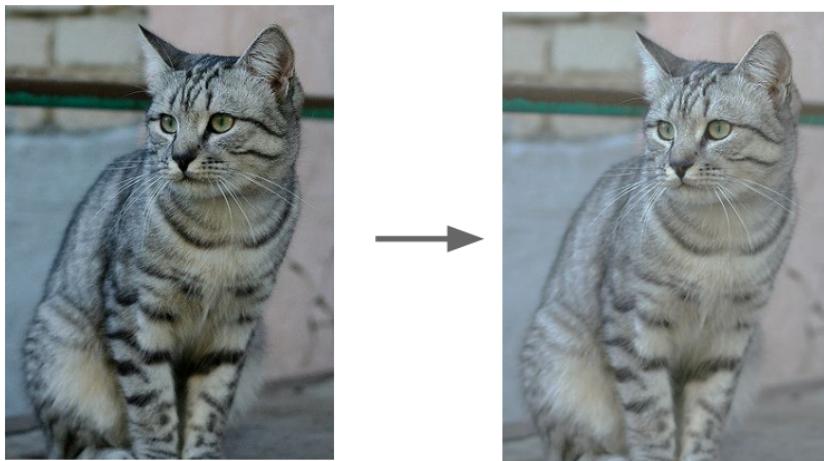
ResNet:

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

Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness



Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness

色
彩
抖
动



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)

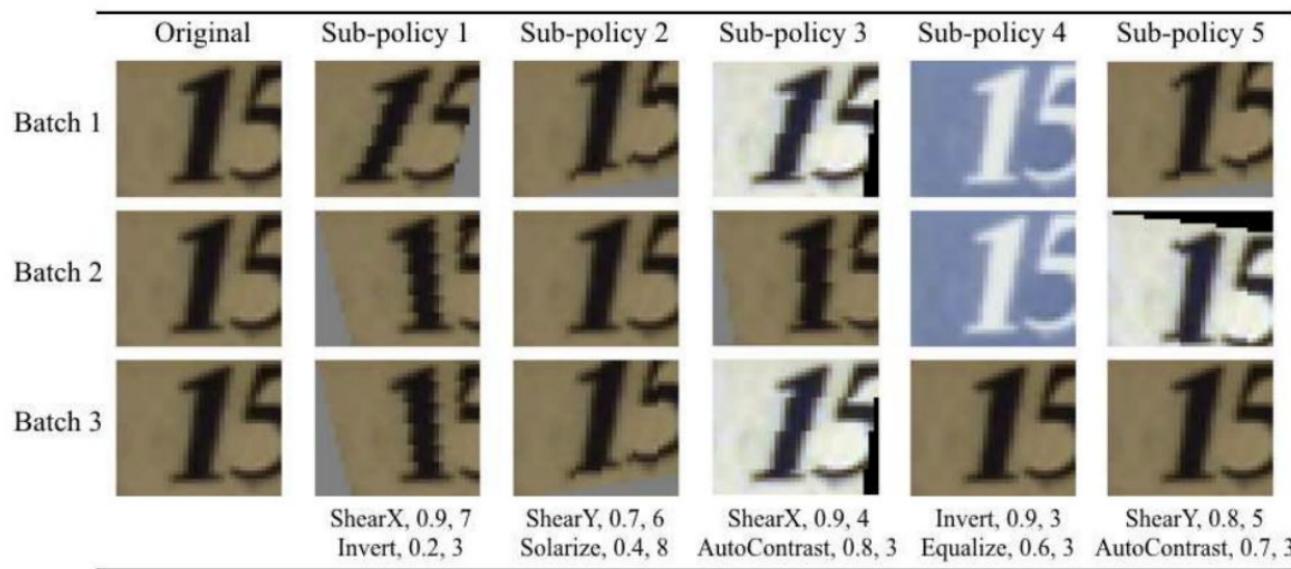
Data Augmentation

Get creative for your problem!

Examples of data augmentations:

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

Automatic Data Augmentation



Cubuk et al., "AutoAugment: Learning Augmentation Strategies from Data", CVPR 2019

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout

Batch Normalization

Data Augmentation

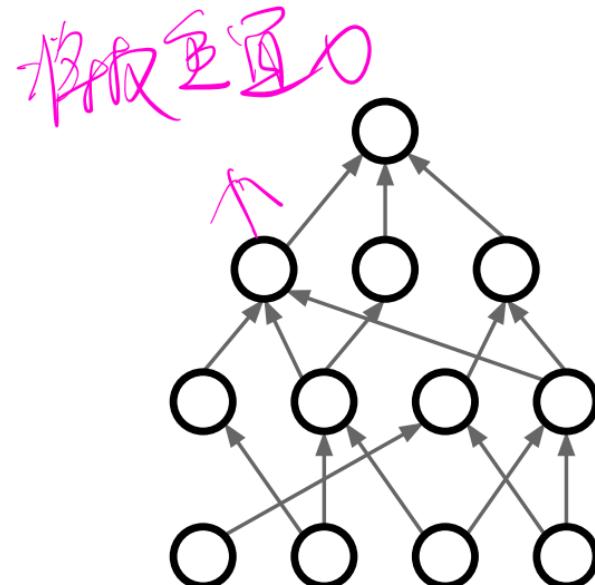
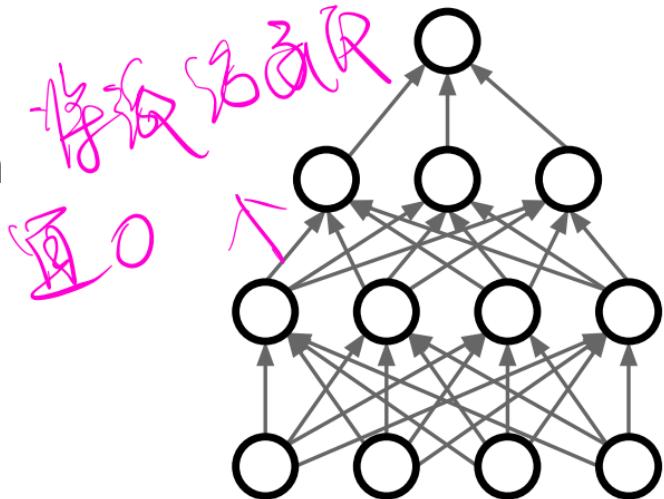
Regularization: DropConnect

Training: Drop connections between neurons (set weights to 0)

Testing: Use all the connections

Examples:

- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect



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

Regularization: Fractional Pooling

Training: Use randomized pooling regions

Testing: Average predictions from several regions

Examples:

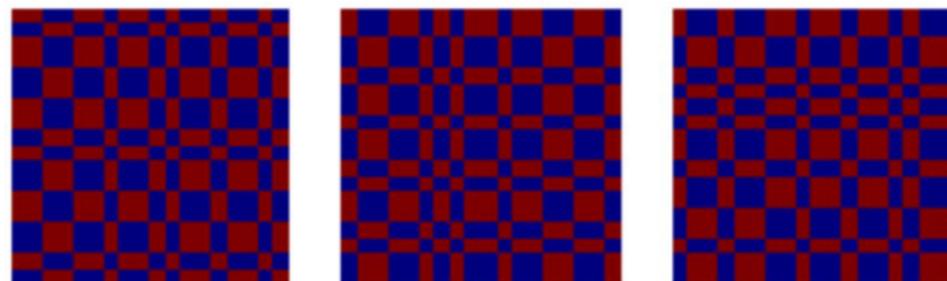
Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling



Graham, "Fractional Max Pooling", arXiv 2014

Regularization: Stochastic Depth

Training: Skip some layers in the network

Testing: Use all the layer

Examples:

Dropout

Batch Normalization

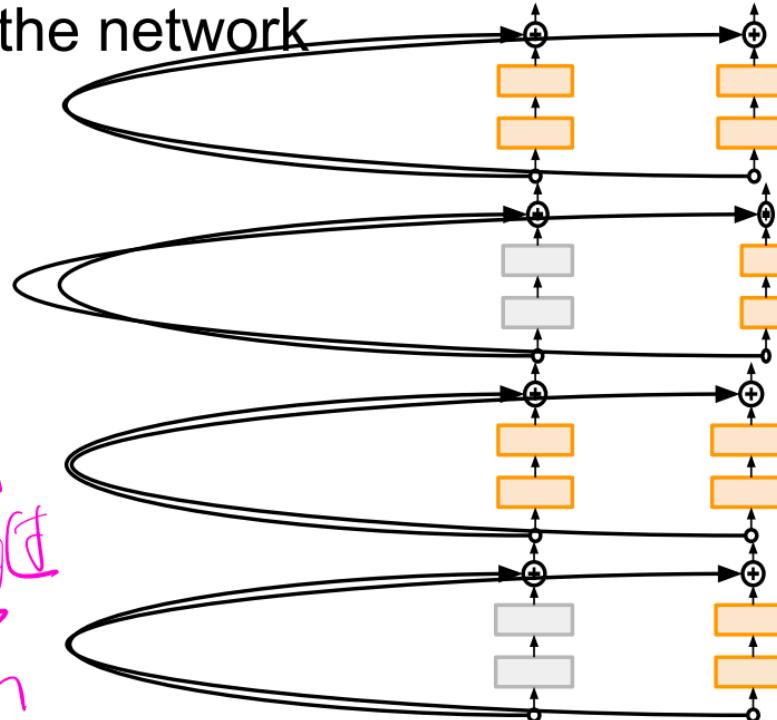
Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth

Handwritten notes in pink ink:
- 11/18/17
- BN
- FMP
- SD



Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016

Regularization: Cutout

Training: Set random image regions to zero

Testing: Use full image

Examples:

Dropout

Batch Normalization

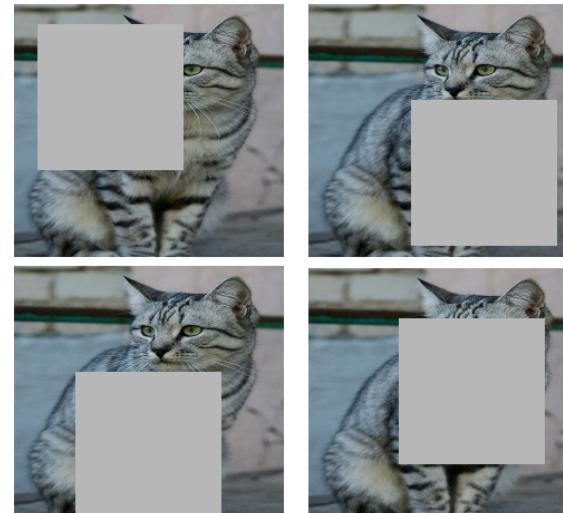
Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout / Random Crop



Works very well for small datasets like CIFAR,
less common for large datasets like ImageNet

DeVries and Taylor, "Improved Regularization of
Convolutional Neural Networks with Cutout", arXiv 2017

Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

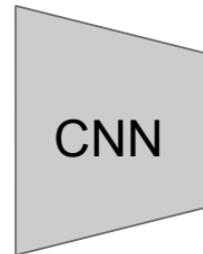
Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth

Cutout / Random Crop

Mixup

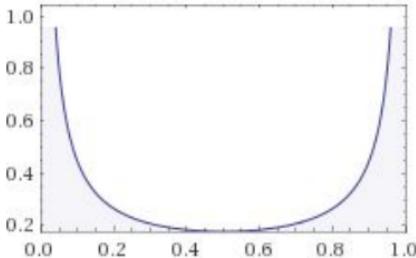
JPB



Target label:
cat: 0.4
dog: 0.6

Randomly blend the pixels
of pairs of training images,
e.g. 40% cat, 60% dog

Zhang et al, "mixup: Beyond Empirical Risk Minimization", ICLR 2018



Regularization - In practice

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout / Random Crop

Mixup

- Consider dropout for large fully-connected layers
- Batch normalization and data augmentation almost always a good idea
- Try cutout and mixup especially for small classification datasets

Choosing Hyperparameters

(without tons of GPUs)

Choosing Hyperparameters

Step 1: Check initial loss

Turn off weight decay, sanity check loss at initialization
e.g. $\log(C)$ for softmax with C classes

IN

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample



Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization



Loss not going down? LR too low, bad initialization

Loss explodes to Inf or NaN? LR too high, bad initialization

Choosing Hyperparameters

$W \in [1e^{-3} \dots 10^{-5}]$

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: $1e-1, 1e-2, 1e-3, 1e-4$

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

~~LR~~

~~WD~~

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

Good weight decay to try: 1e-4, 1e-5, 0

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer



Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

Choosing Hyperparameters

Step 1: Check initial loss

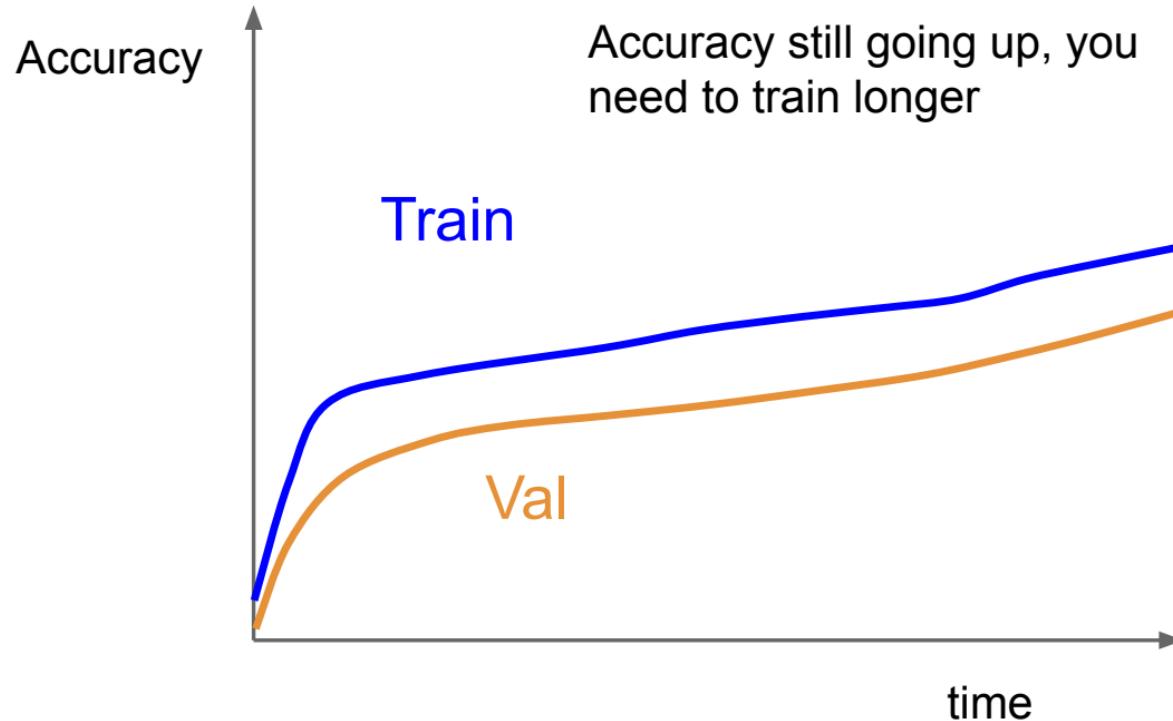
Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

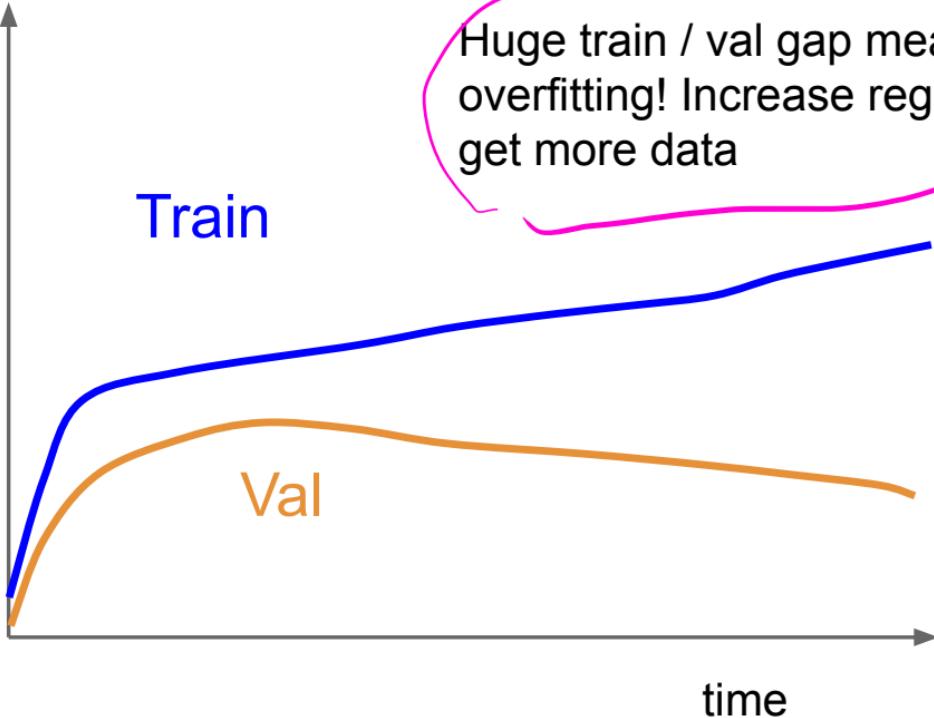
Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

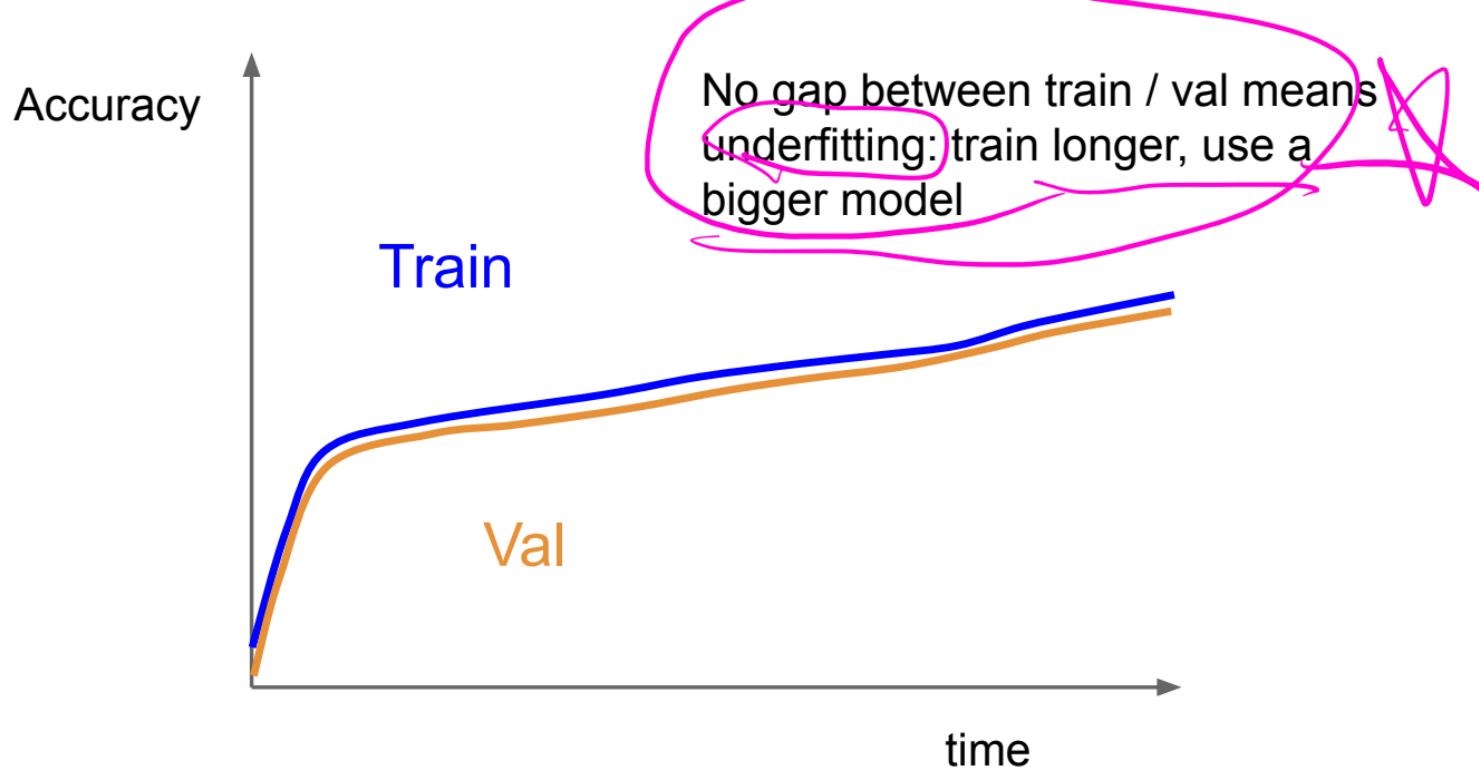
Step 6: Look at loss and accuracy curves



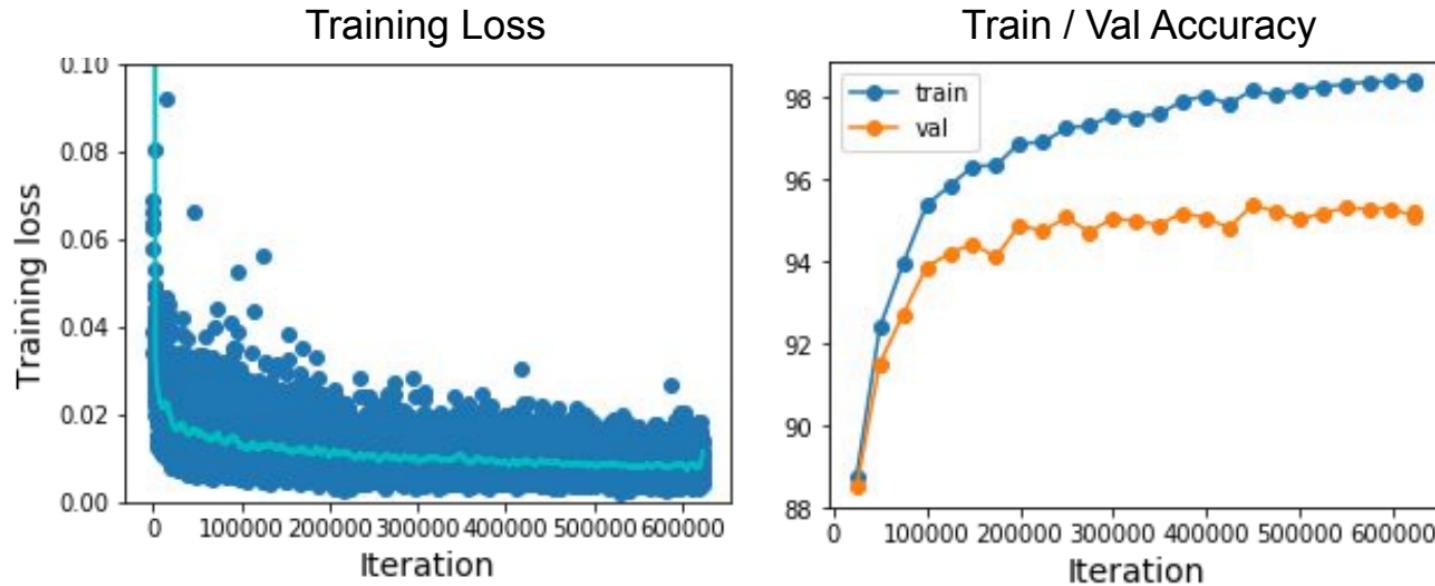
Accuracy



Huge train / val gap means
overfitting! Increase regularization,
get more data



Look at learning curves!

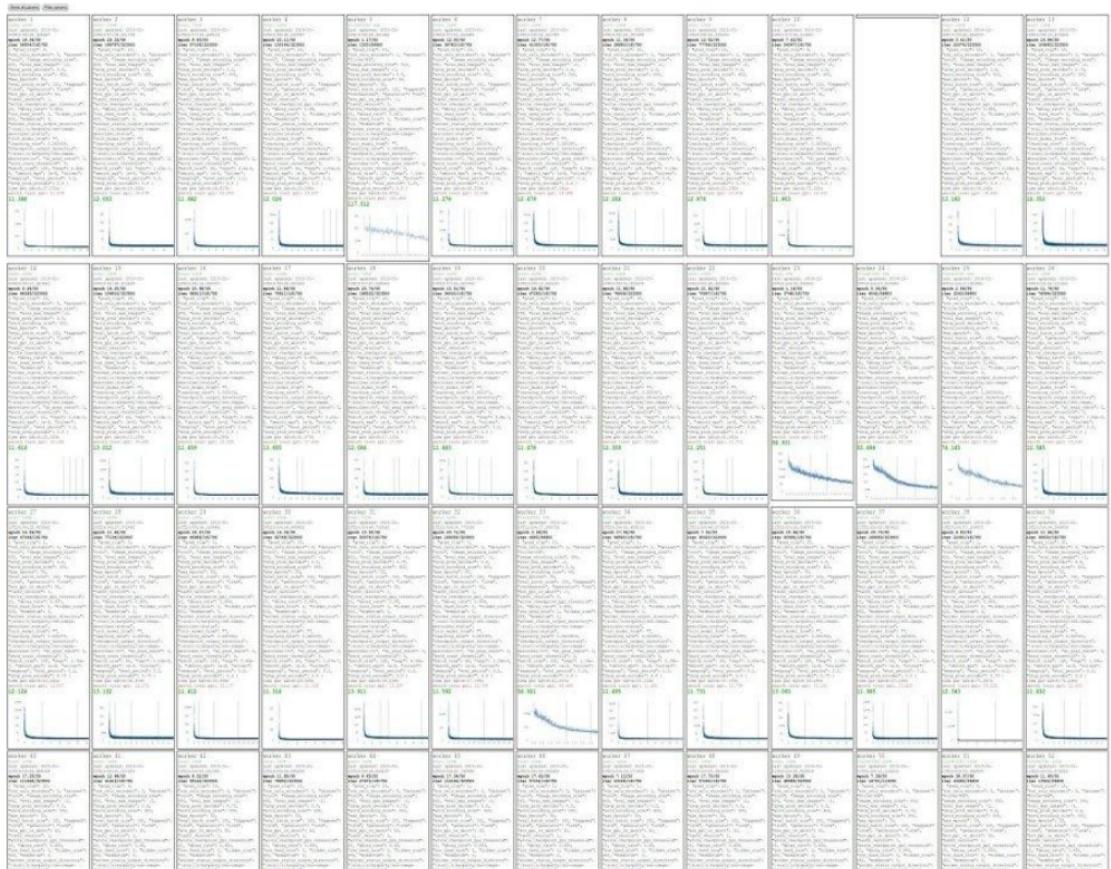


Losses may be noisy, use a scatter plot and also plot moving average to see trends better

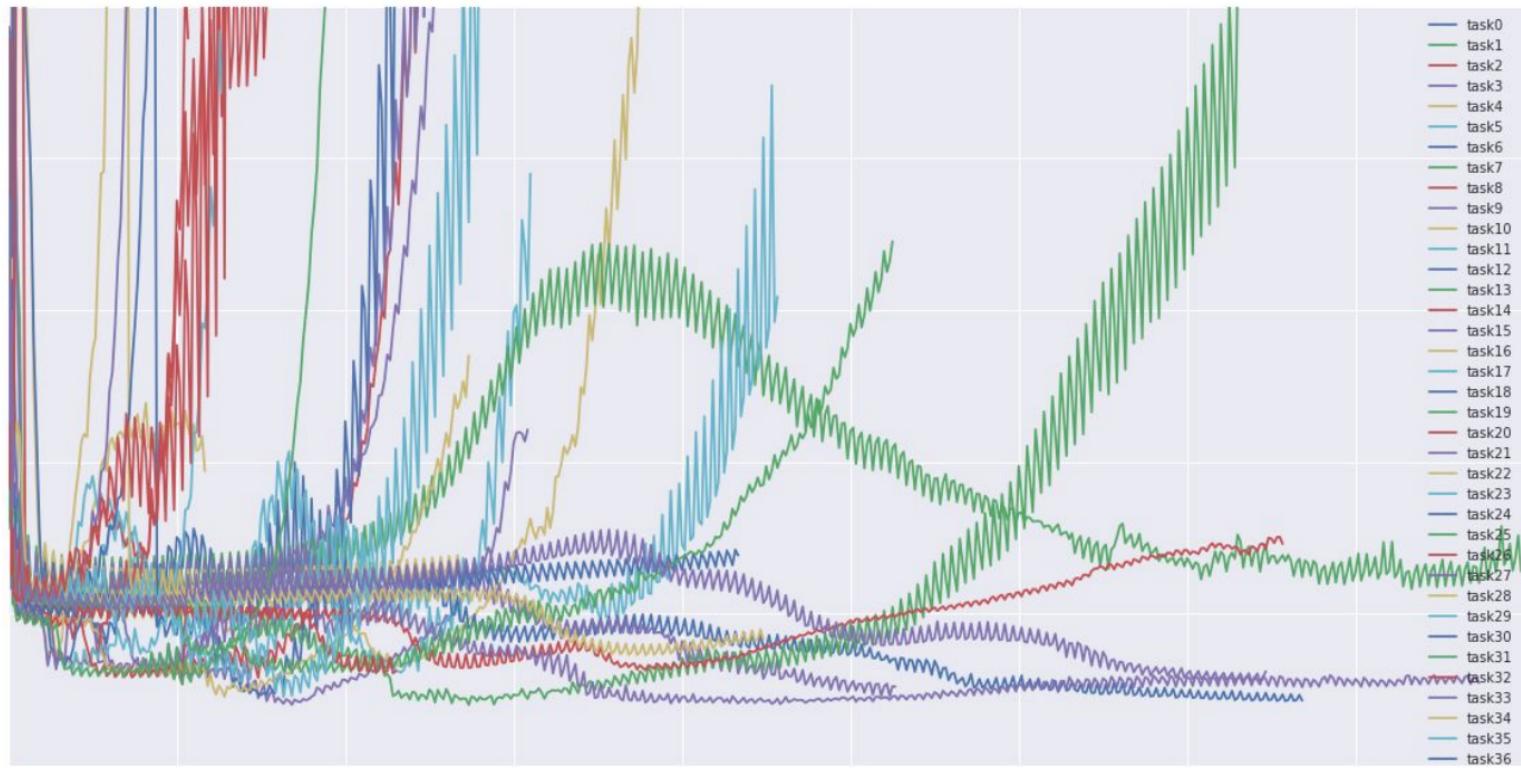
Cross-validation

We develop "command centers" to visualize all our models training with different hyperparameters

check out [weights and biases](#)



You can plot all your loss curves for different hyperparameters on a single plot



Don't look at accuracy or loss curves for too long!



Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

Step 6: Look at loss and accuracy curves

Step 7: GOTO step 5

Random Search vs. Grid Search

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

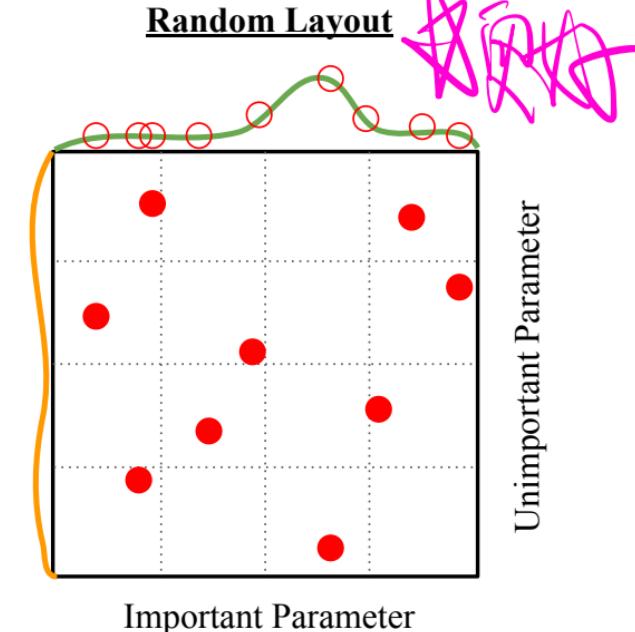
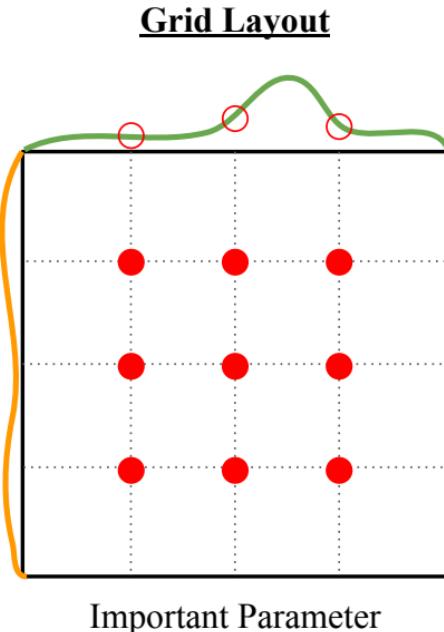


Illustration of Bergstra et al., 2012 by Shayne
Longpre, copyright CS231n 2017

Summary

- Improve your training error:
 - Optimizers
 - Learning rate schedules
- Improve your test error:
 - Regularization
 - Choosing Hyperparameters

Summary

TLDRs

We looked in detail at:

- Activation Functions ([use ReLU](#))
- Data Preprocessing ([images: subtract mean](#))
- Weight Initialization ([use Xavier/He init](#))
- Batch Normalization ([use this!](#))
- Transfer learning ([use this if you can!](#))

Next time: Visualizing and Understanding