

Training Neural Networks

Practical Issues

M. Soleymani

Sharif University of Technology

Spring 2026

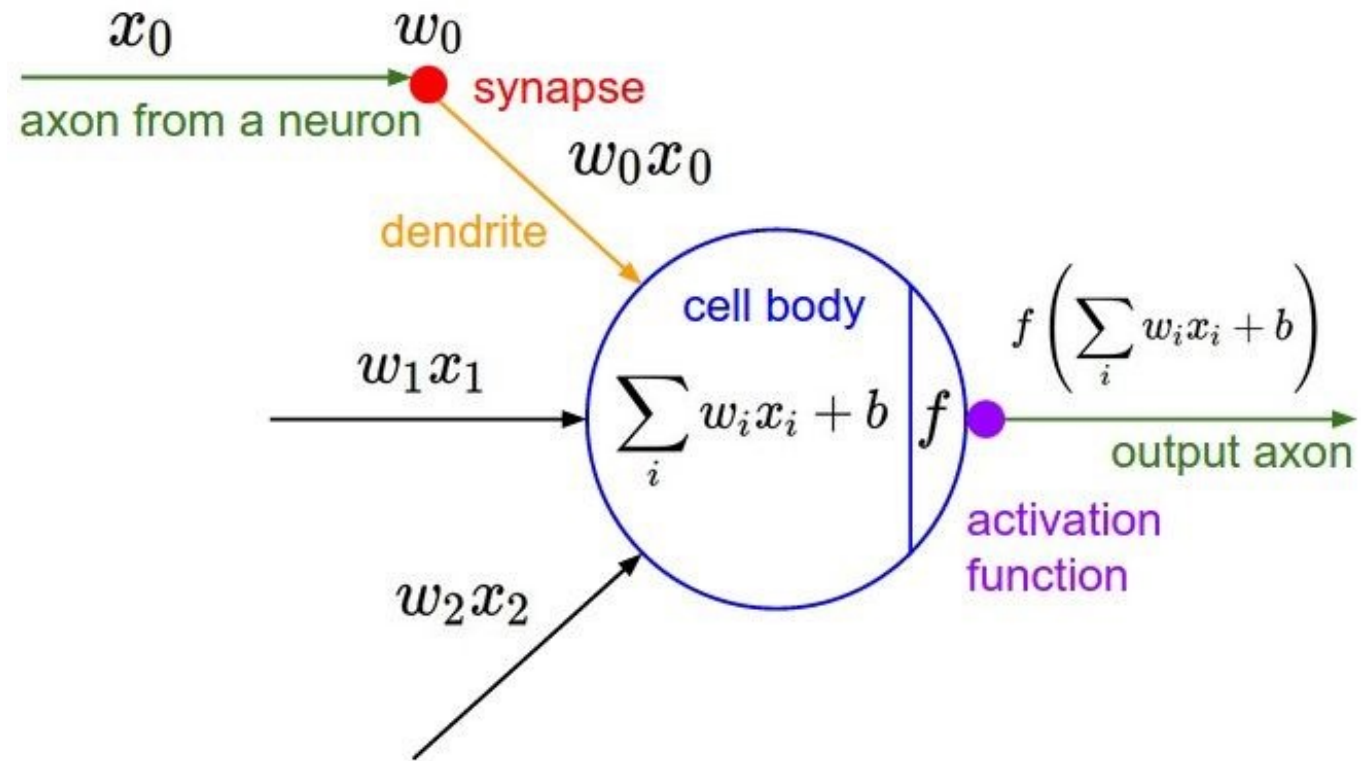
Most slides have been adapted from Fei Fei Li and colleagues lectures, cs231n

Outline

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Choosing hyperparameters

Activation Functions

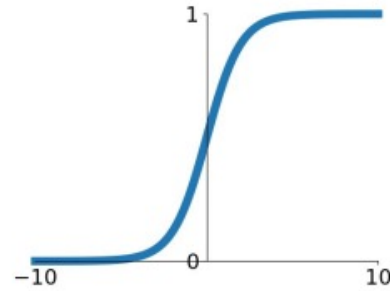
Neuron



Activation Functions

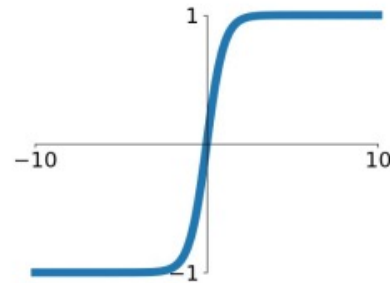
Sigmoid

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



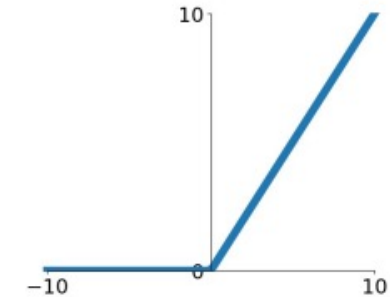
tanh

$$\tanh(x)$$



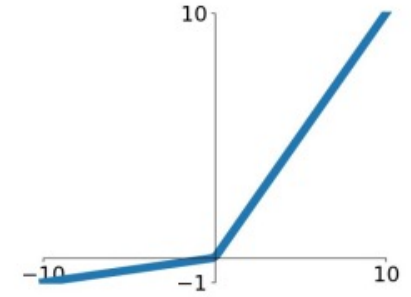
ReLU

$$\max(0, x)$$



Leaky ReLU

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

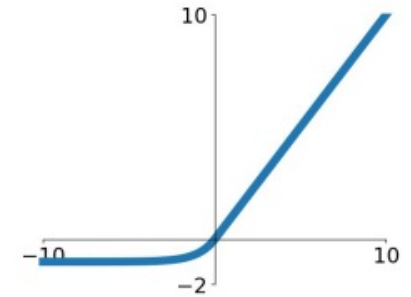


Maxout

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

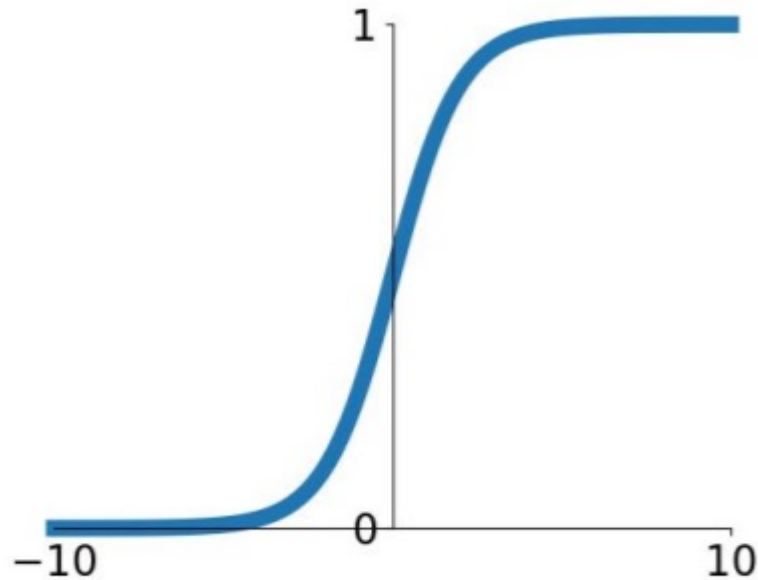
ELU

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



Activation functions: sigmoid

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



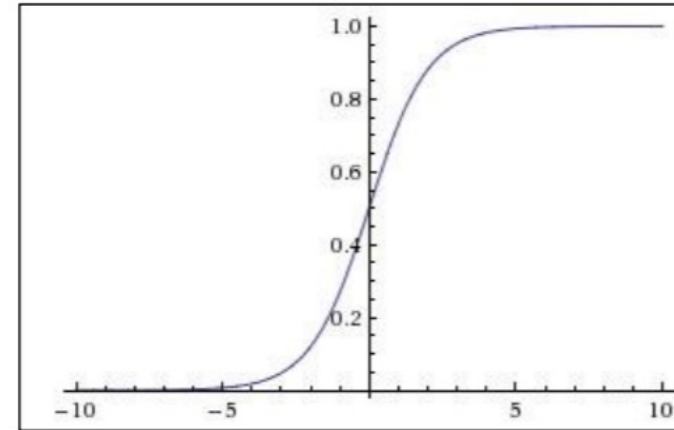
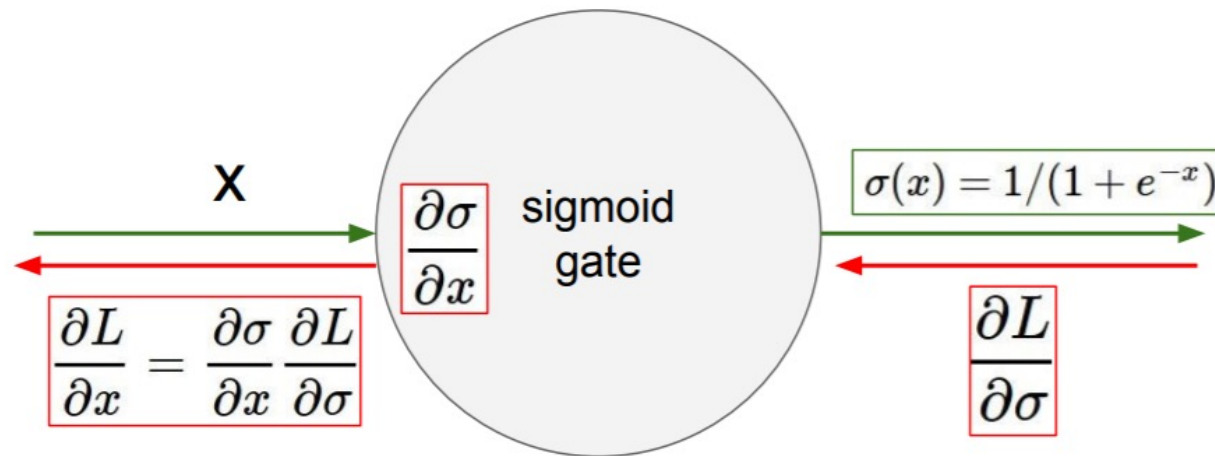
Sigmoid

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

Problems:

1. Saturated neurons “kill” the gradients

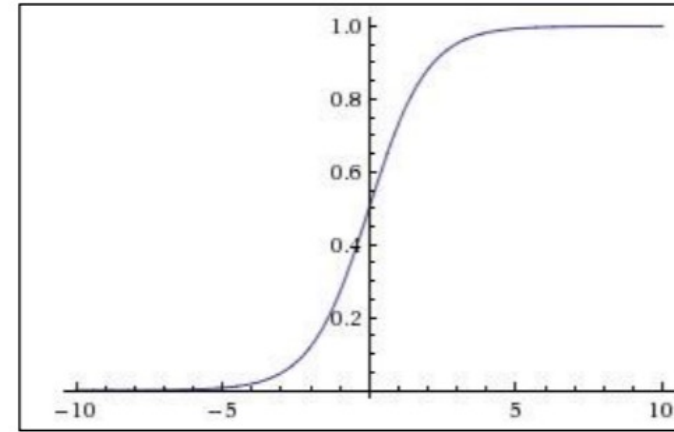
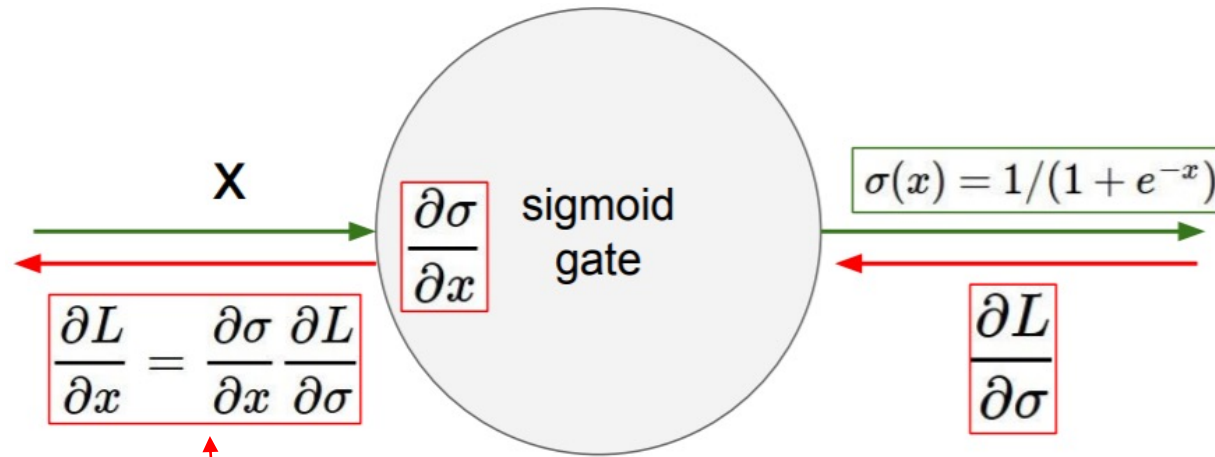
Activation functions: sigmoid



$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x)$$

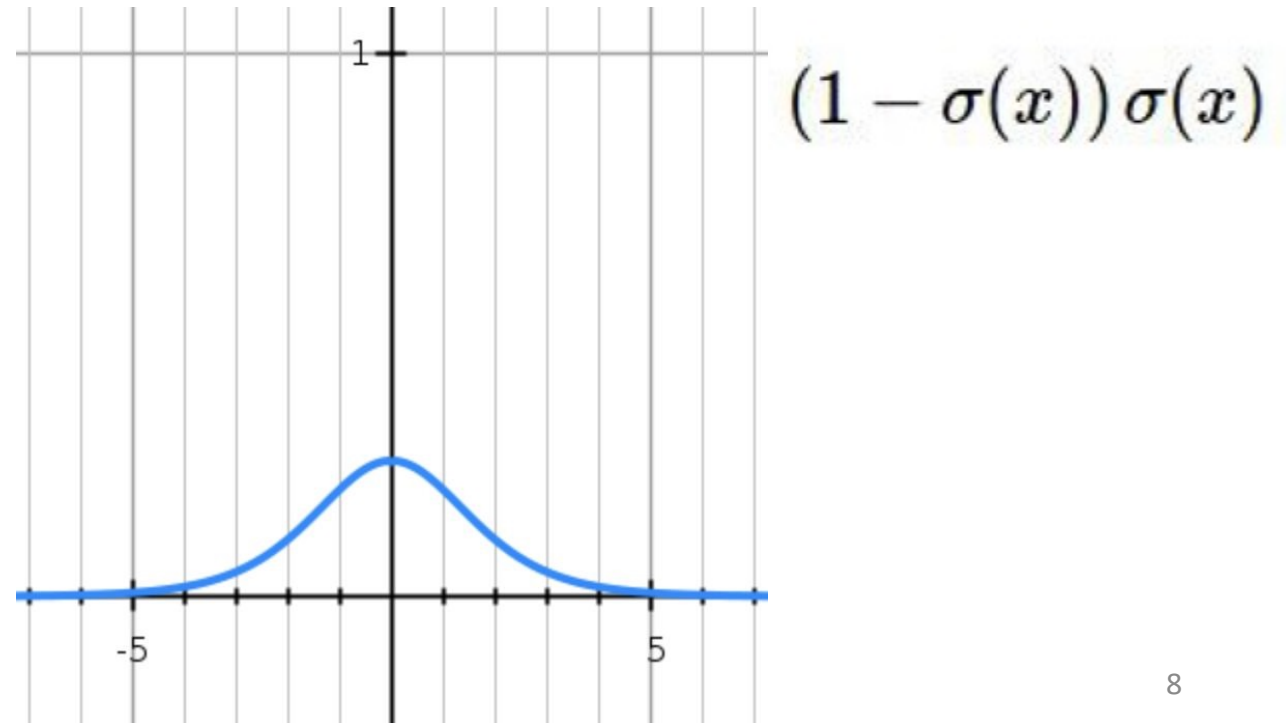
- What happens when $x = -10$?
- What happens when $x = 0$?
- What happens when $x = 10$?

Activation functions: sigmoid



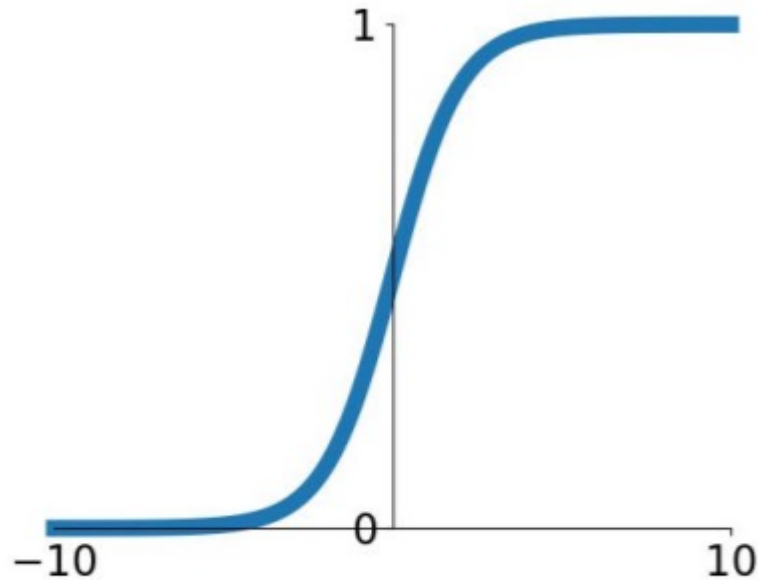
Why is this a problem?

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



Activation functions: sigmoid

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



Sigmoid

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

Problems:

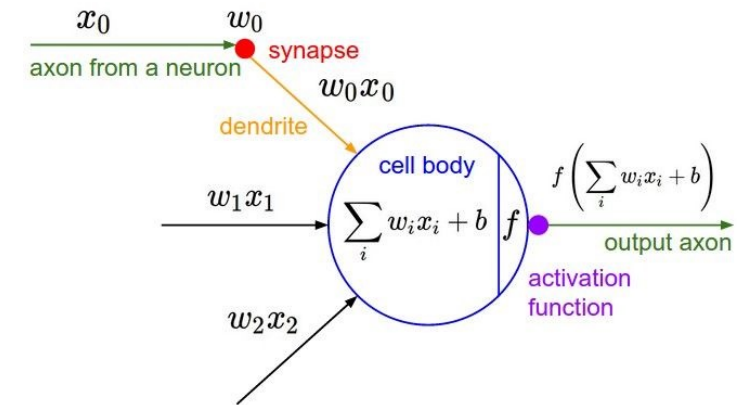
1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered

Activation functions: sigmoid

- 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 $\mathbf{w}$?



Activation functions: sigmoid

- 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 ?

We know that local gradient of sigmoid is always positive

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

We are assuming x is always positive

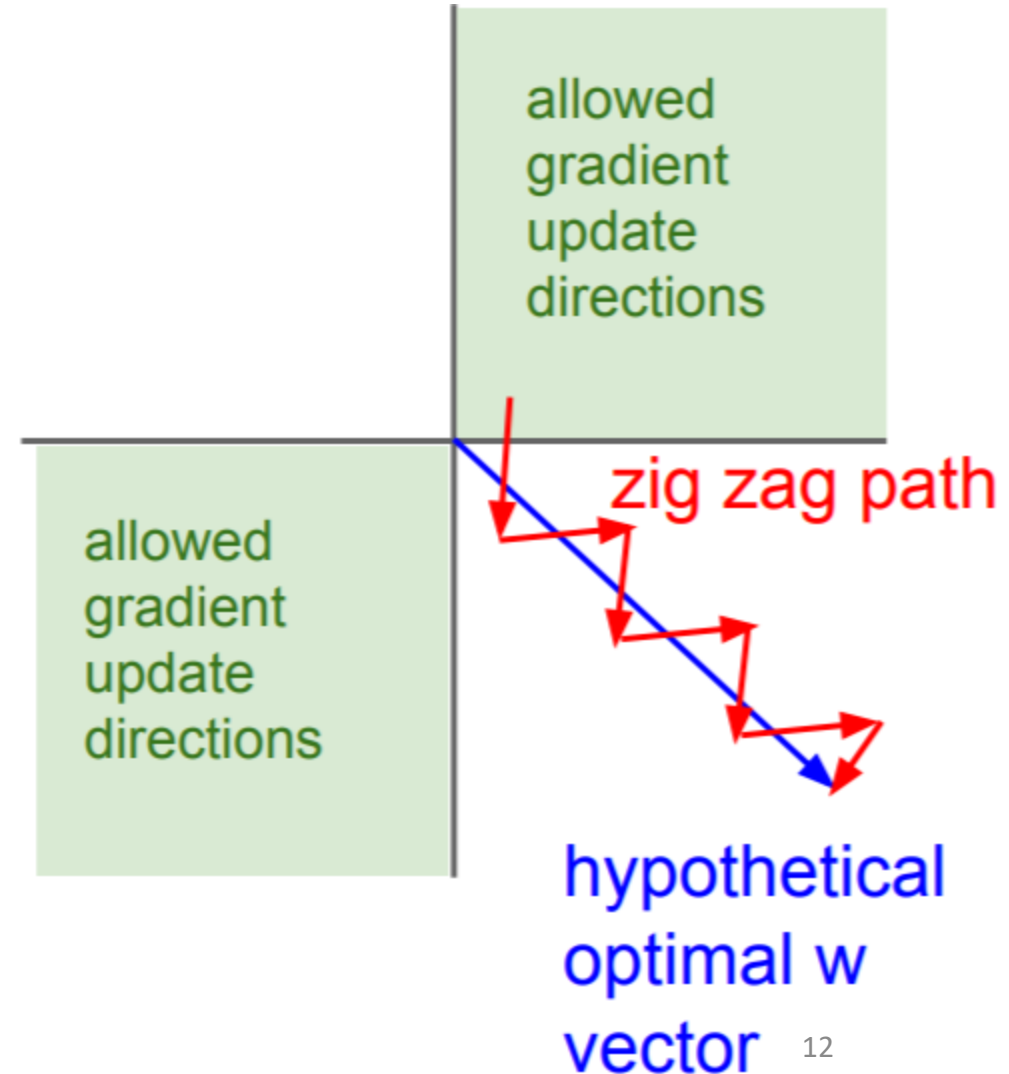
So!! Sign of gradient **for all w_i** is the same as the sign of upstream scalar gradient!

Activation functions: sigmoid

- Consider what happens when the input to a neuron is always positive...
- What can we say about the gradients on w ? **Always all positive or all negative** 😞

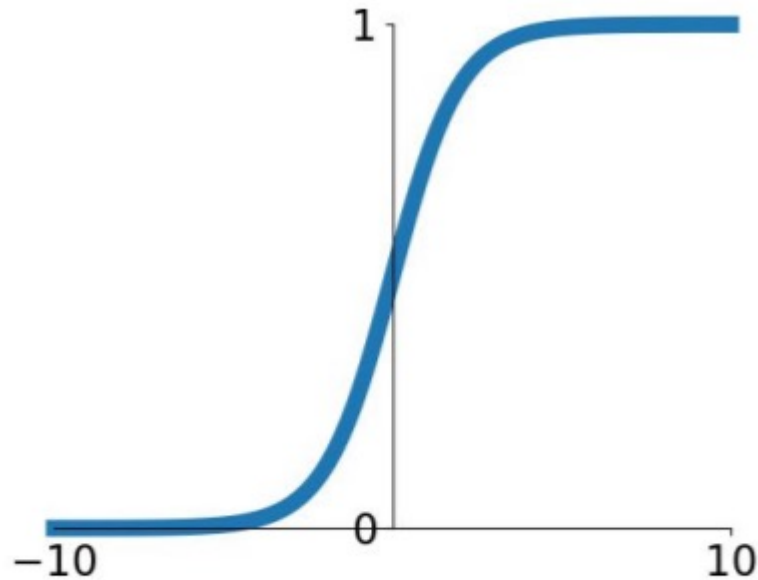
$$f\left(\sum_i w_i x_i + b\right)$$

(For a single element! Minibatches help)



Activation functions: sigmoid

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



Sigmoid

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

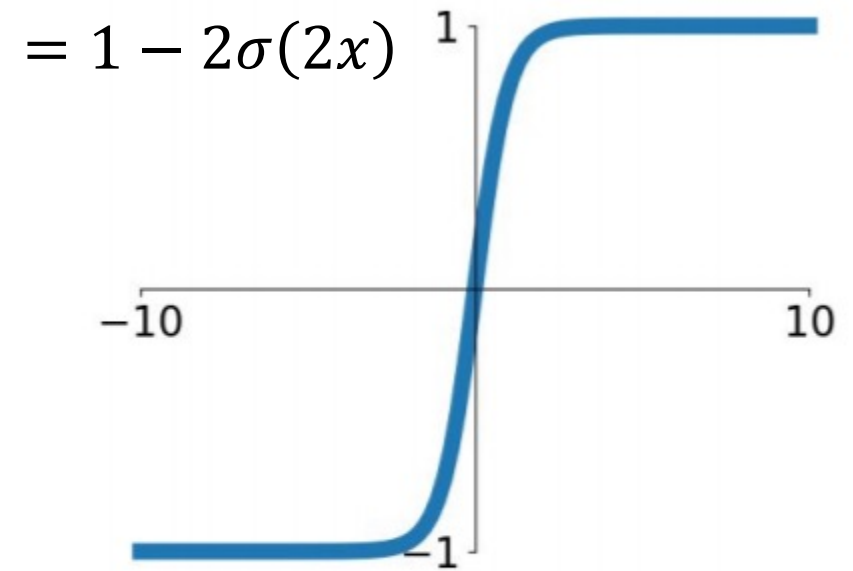
3 problems:

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

Activation functions: tanh

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

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \quad [\text{LeCun et al., 1991}]$$

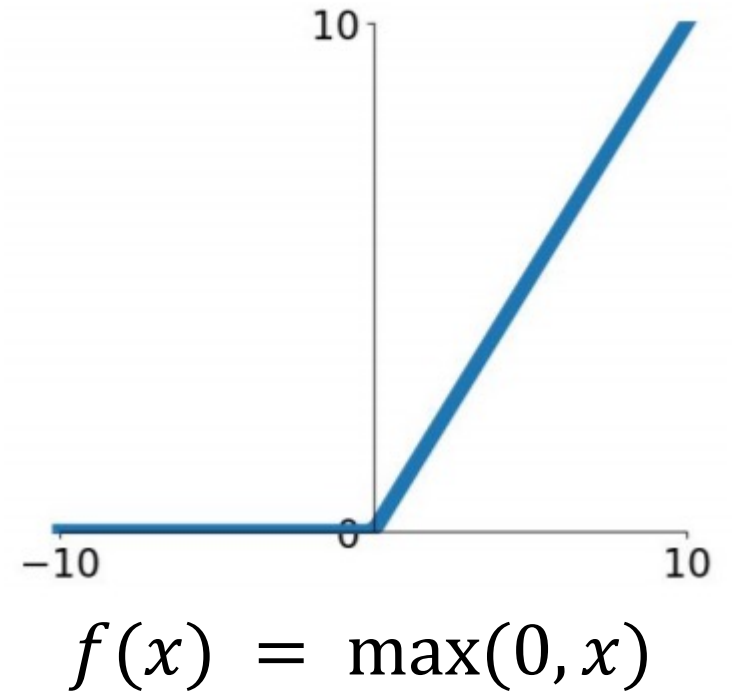


tanh(x)

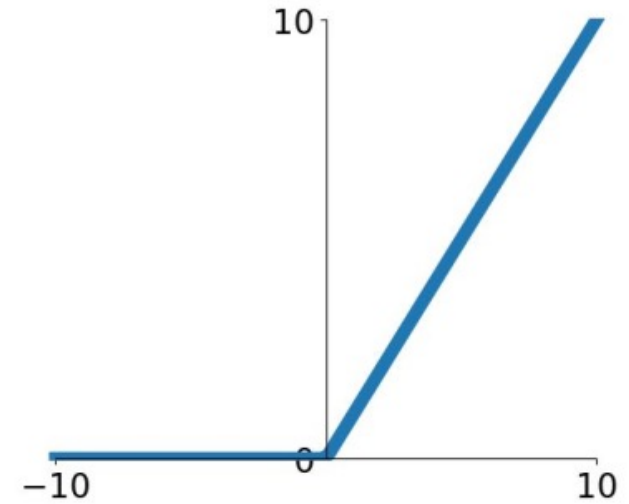
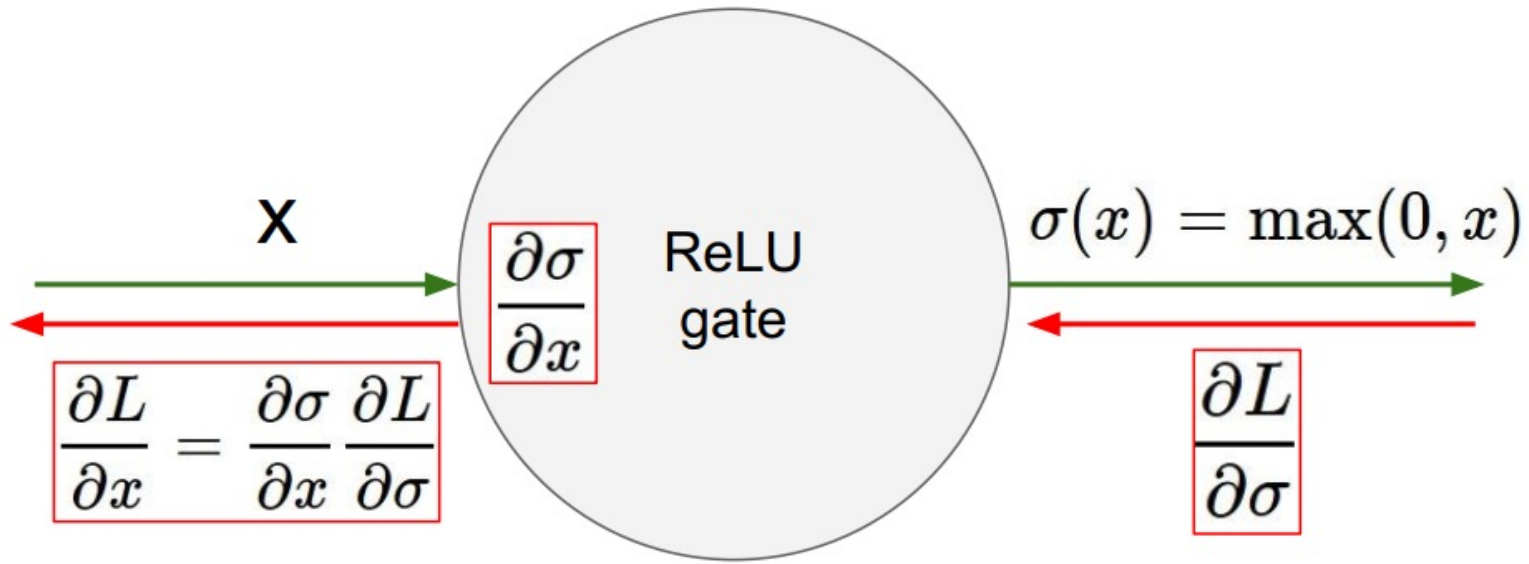
Activation functions: ReLU (Rectified Linear Unit)

[Krizhevsky et al., 2012]

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



Activation functions: ReLU

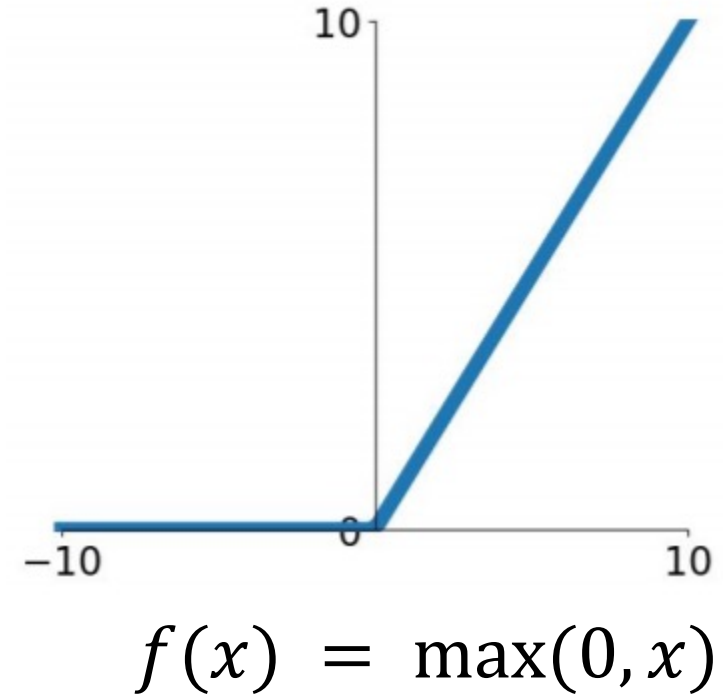


- What happens when $x = -10$?
- What happens when $x = 0$?
- What happens when $x = 10$?

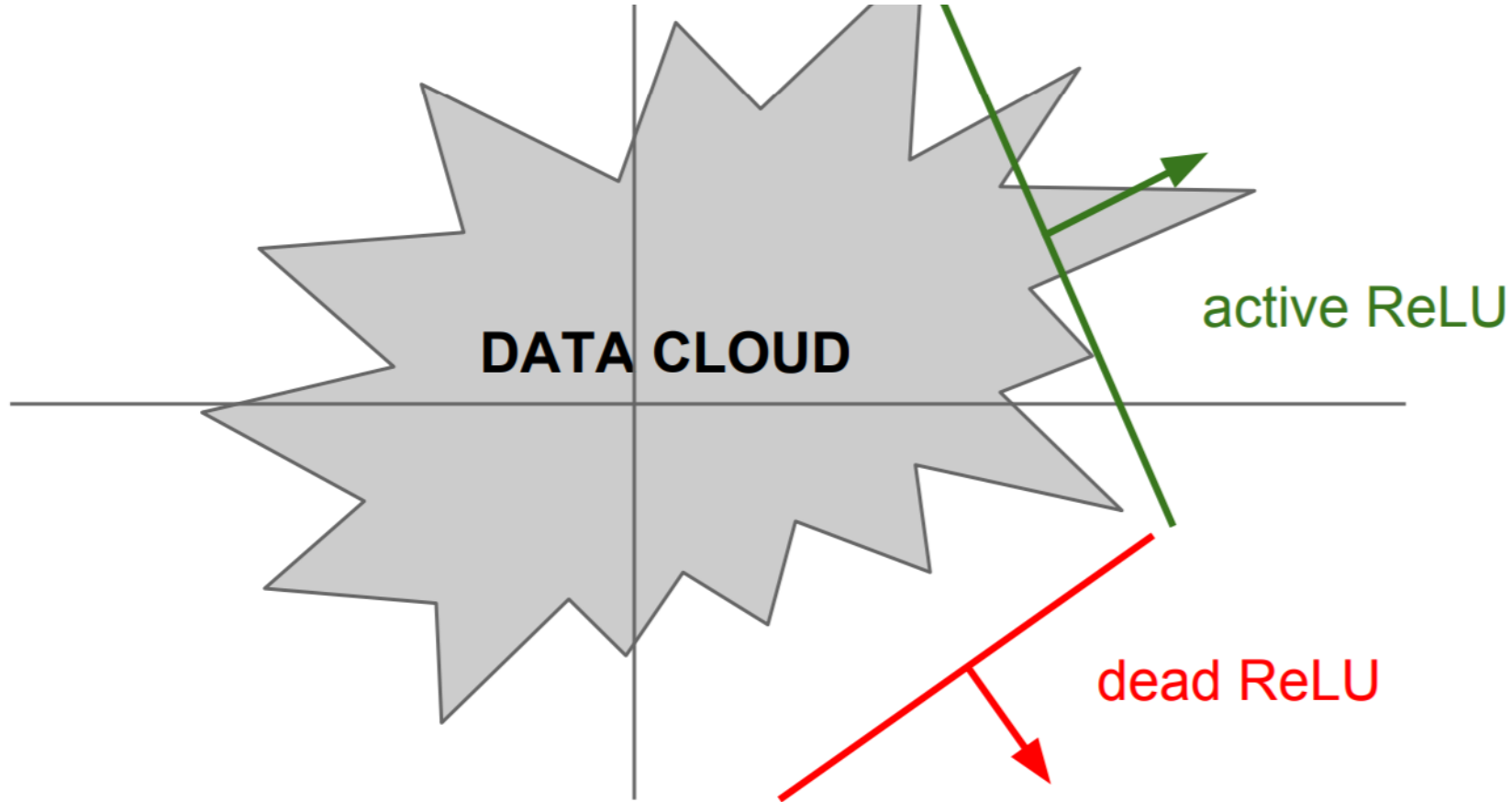
Activation functions: ReLU

[Krizhevsky et al., 2012]

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid
- Problems:
 - Not zero-centered output
 - An annoyance:
 - hint: what is the gradient when $x < 0$?



Activation functions: ReLU

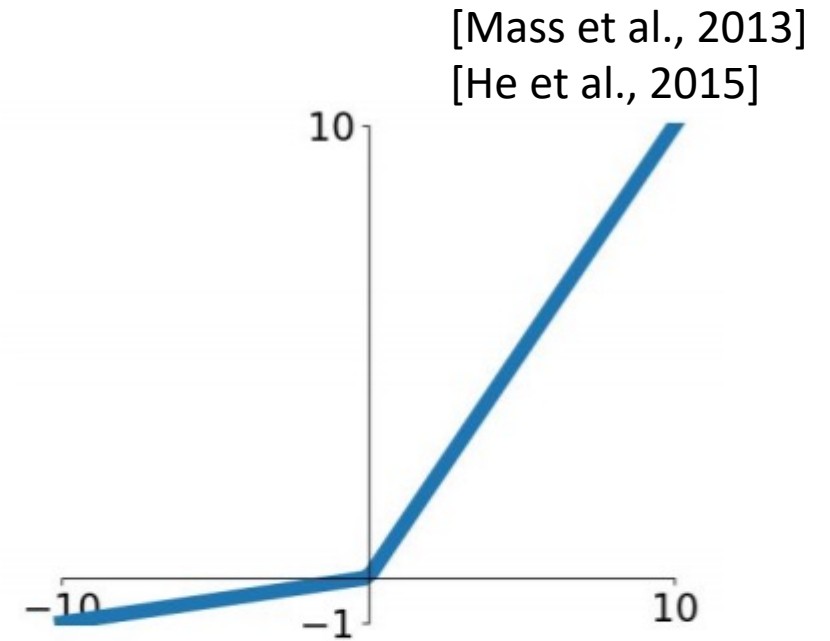


=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

It can be due initialization or high learning rate

Activation functions: Leaky ReLU

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



Parametric Rectifier (PReLU)

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

α is determined during backpropagation

Leaky ReLU

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

Maxout “Neuron”

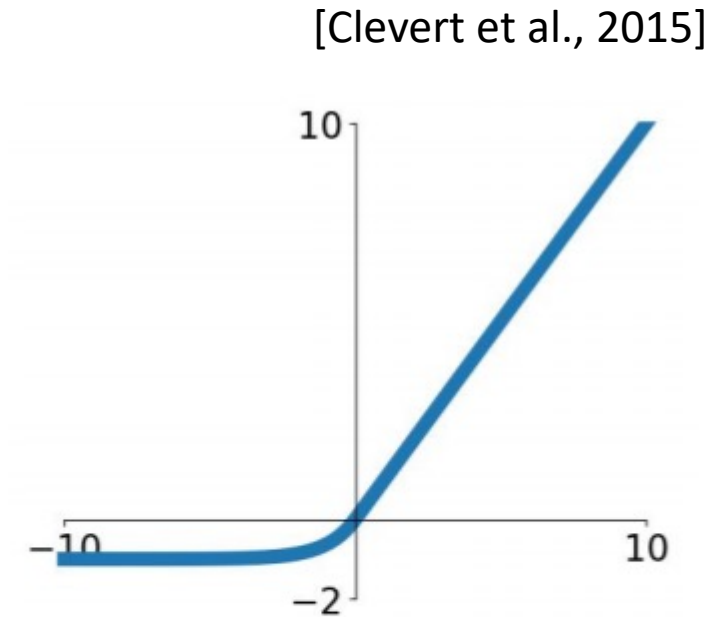
[Goodfellow et al., 2013]

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

- Does not have the basic form of “dot product -> nonlinearity”
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!
- Problem: doubles the number of parameters/neuron :(

Activation Functions: 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
- Computation requires $\exp()$



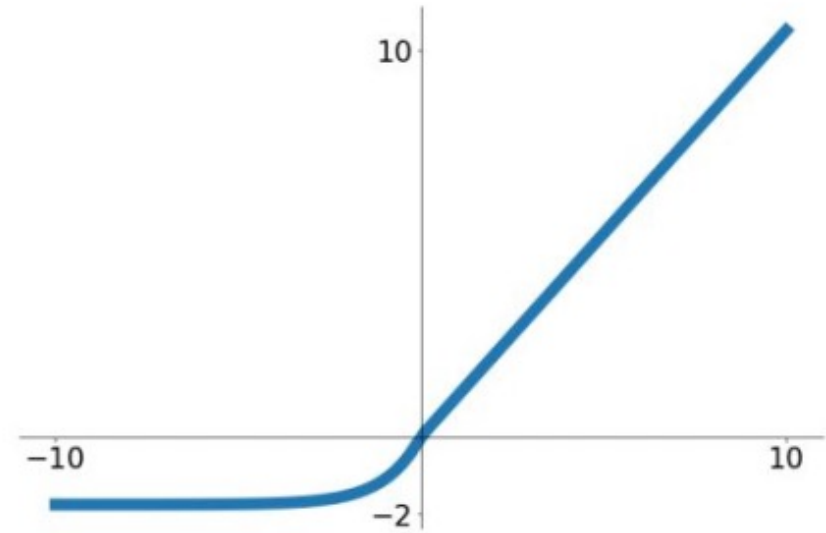
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

(Alpha default = 1)

Activation Functions: 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

[Klambauer et al. ICLR 2017]



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

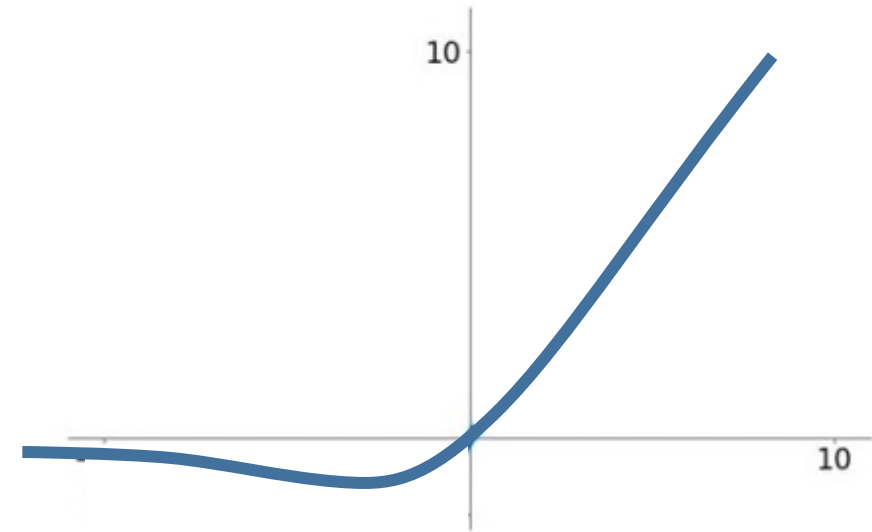
$$\alpha = 1.6732632423543772848170429916717$$

$$\lambda = 1.0507009873554804934193349852946$$

Gaussian Error Linear Unit (GELU)

$$GELU(x) = x\phi(\alpha x) = x \int_{-\infty}^{\alpha x} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt$$

- α controls the smoothness of the GELU function (generally $\alpha = 1$)
- Used in BERT, ViT, and GPT

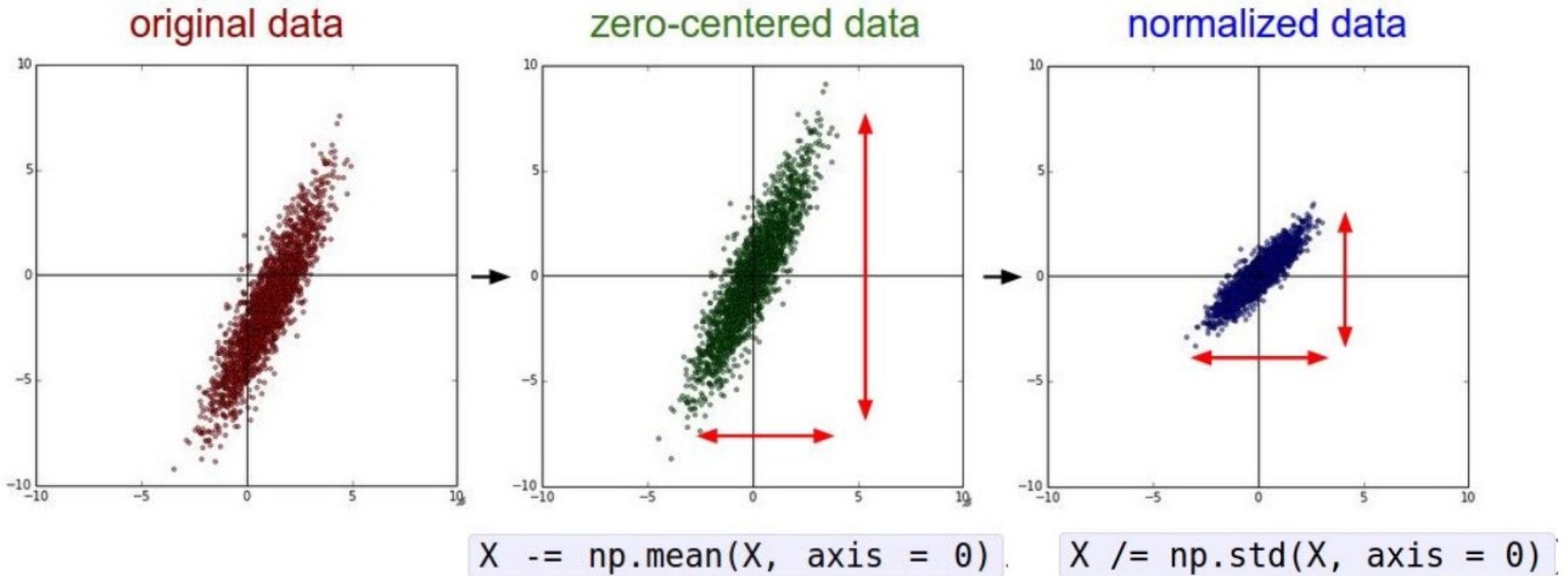


Activation functions

- In practice:
 - Use **ReLU**. Be careful with your learning rates
 - Try out **Leaky ReLU / Maxout / ELU / SELU / GELU**
 - To squeeze out some marginal gains
 - Use **sigmoid** and **tanh** only for gating purposes

Data Preprocessing

Data Preprocessing



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

Normalizing the input

- On the training set compute mean of each input (feature)

$$- \mu_i = \frac{\sum_{n=1}^N x_i^{(n)}}{N}$$

$$- \sigma_i^2 = \frac{\sum_{n=1}^N (x_i^{(n)} - \mu_i)^2}{N}$$

- **Remove mean:** from each of the input mean of the corresponding input

$$- x_i \leftarrow x_i - \mu_i$$

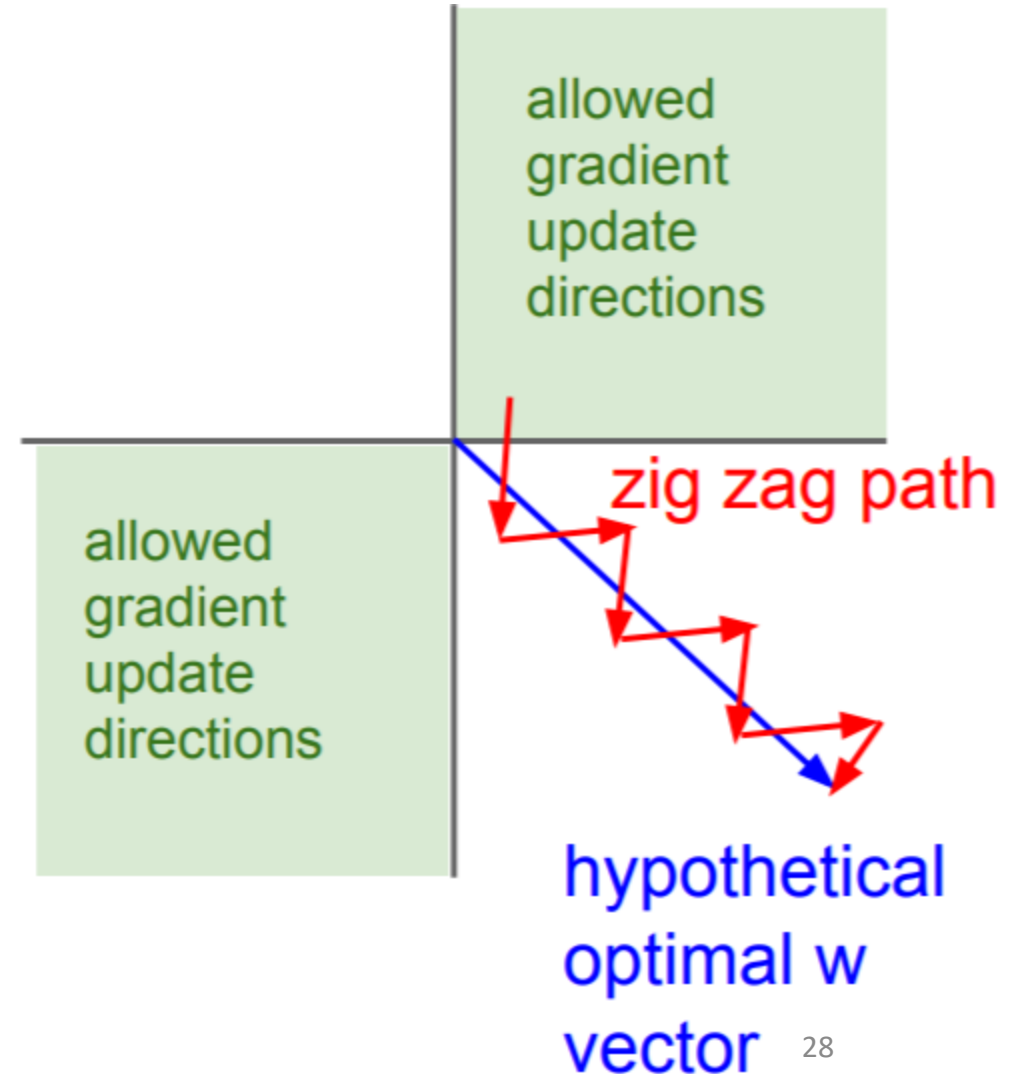
- **Normalize variance:**

$$- x_i \leftarrow \frac{x_i}{\sigma_i}$$

- If we normalize the training set we use the same μ and σ^2 to normalize test data too

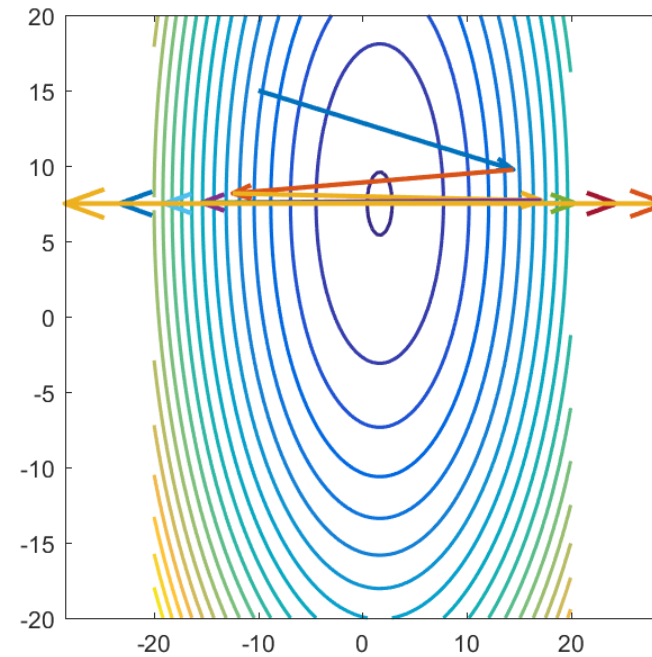
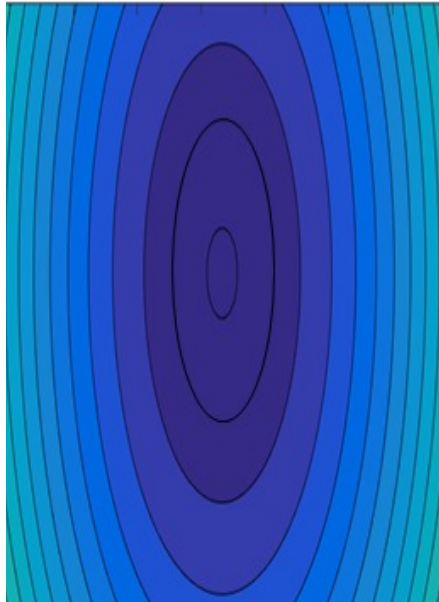
Why zero-mean input?

- Reminder: sigmoid
- Consider what happens when the input to a neuron is always positive...
- What can we say about the gradients on w ? **Always all positive or all negative ☹️**
 - this is also why you want zero-mean data!

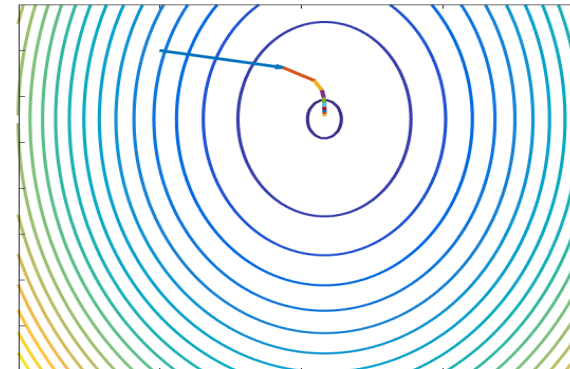
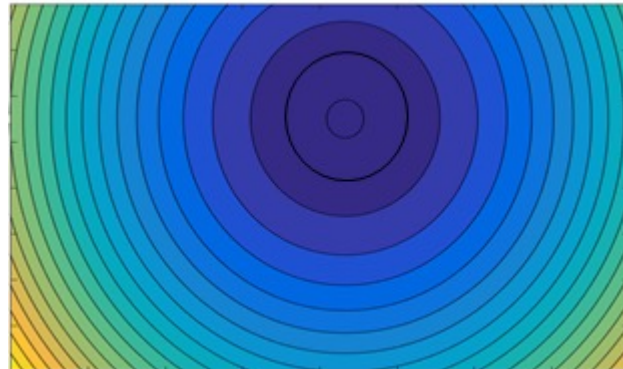


Why normalize inputs?

Very different range
of input features



Normalized input:
Speedup training



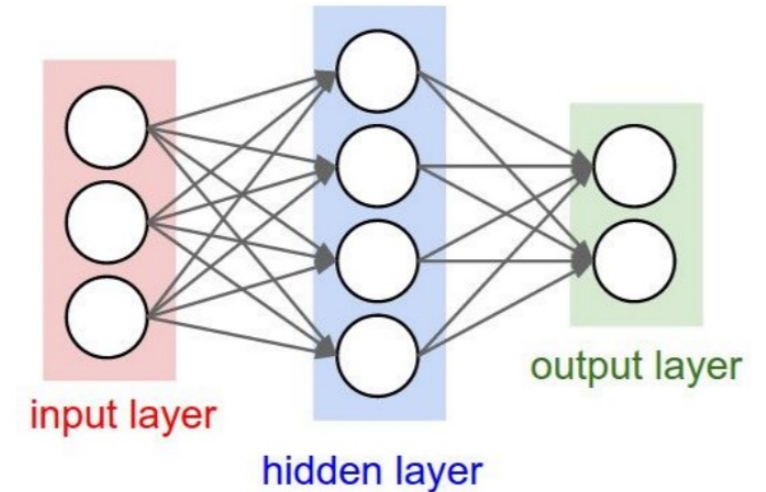
TLDR: In practice for Images: center only

- Example: 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)

Weight Initialization

Weight initialization

- How to choose the starting point for the iterative process of optimization
- Q: what happens when $W=\text{constant}$ init is used?
- Key point: neurons with identical connections that are identically initialized will never diverge
 - All neurons of a layer do the same thing



Weight initialization

- 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

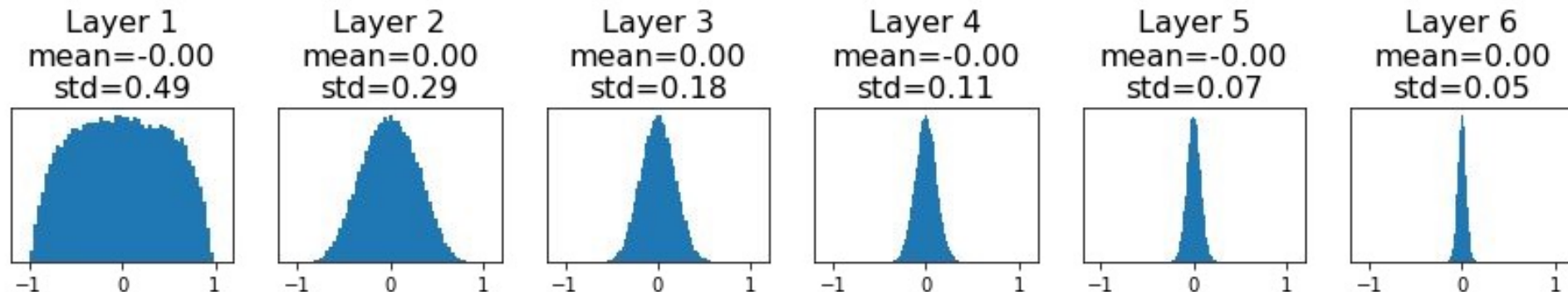
```
dims = [4096] * 7    Forward pass for a 6-layer
hs = []              net with hidden size 4096
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

What will happen to the activations for the last layer?

All activations tend to zero for deeper network layers

What do the gradients dL/dW look like?

All zero, no learning =(



Weight Initialization: Activation statistics

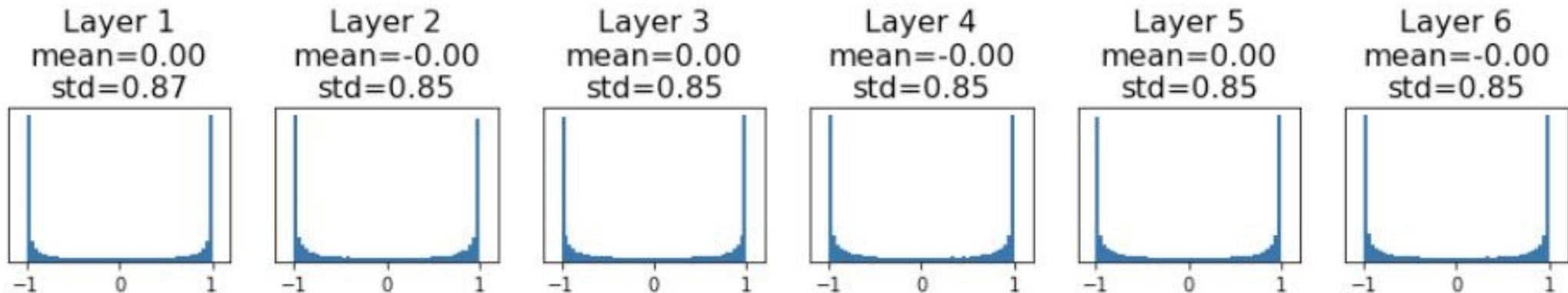
```
dims = [4096] * 7    Increase std of initial
hs = []              weights from 0.01 to 0.05
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

What will happen to the activations for the last layer?

All activations saturate

What do the gradients look like?

Local gradients all zero, no learning =(

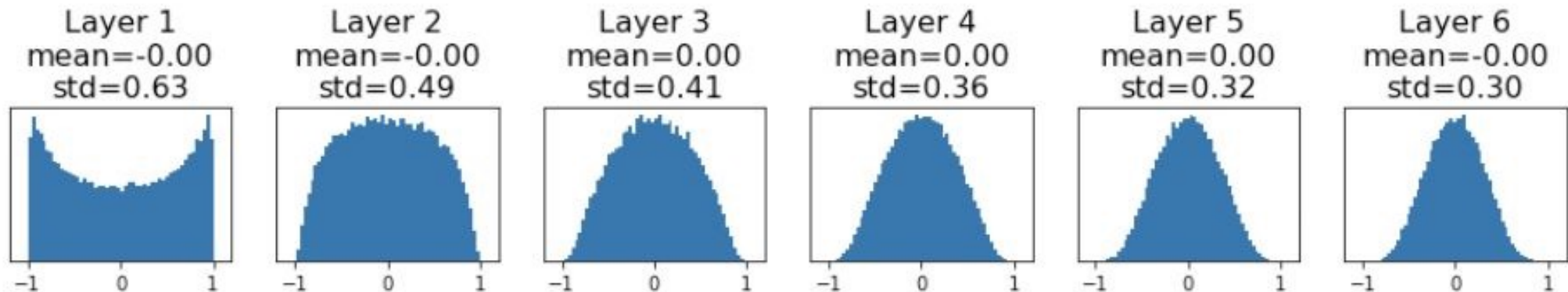


Weight Initialization: “Xavier” Initialization

Initialization: Gaussian with **zero mean** and **$1/\sqrt{\text{fan_in}}$ standard deviation** [Glorot et al., 2010]
fan_in for fully connected layers = number of neurons in the previous layer

```
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!



Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7
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.tanh(x.dot(W))
    hs.append(x)
```

“Xavier” initialization:
std = $1/\sqrt{D_{in}}$

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

For conv layers, we will see D_{in} is $\text{filter_size}^2 * \text{input_channels}$

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

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

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

$\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}})$
[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!

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

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

[Assume all x_i, w_i are iid]

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!

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

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

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

$$\begin{aligned}\text{Var}(y) &= \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}) \\ &= D_{in} \text{Var}(x_i w_i) \\ &= D_{in} \text{Var}(x_i) \text{Var}(w_i) \\ &\text{[Assume all } x_i, w_i \text{ are zero mean]}\end{aligned}$$

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7
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.tanh(x.dot(W))
    hs.append(x)
```

“Xavier” initialization:

std = $1/\sqrt{D_{in}}$

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

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

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

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

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

$$\begin{aligned}\text{Var}(y) &= \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}) \\ &= D_{in} \text{Var}(x_i w_i) \\ &= D_{in} \text{Var}(x_i) \text{Var}(w_i) \\ &\text{[Assume all } x_i, w_i \text{ are iid]}\end{aligned}$$

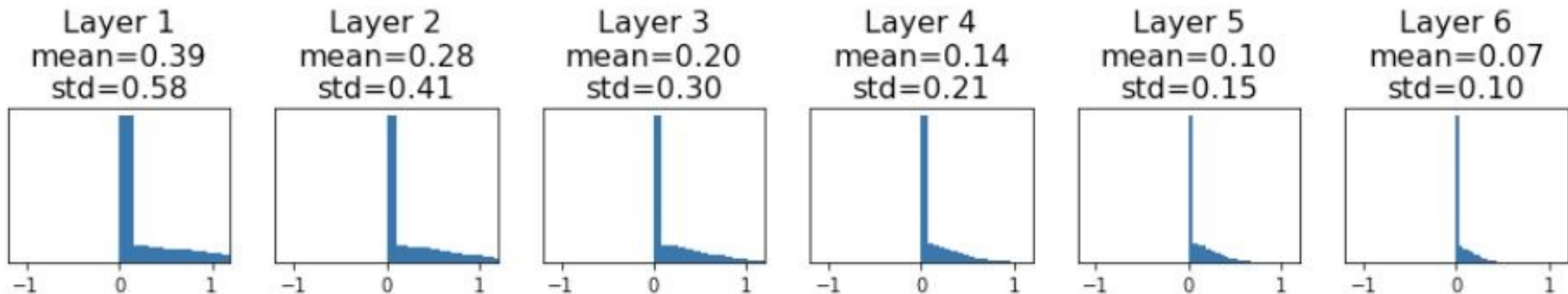
So, $\text{Var}(y) = \text{Var}(x_i)$ only when $\text{Var}(w_i) = 1/D_{in}$

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)
```

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(



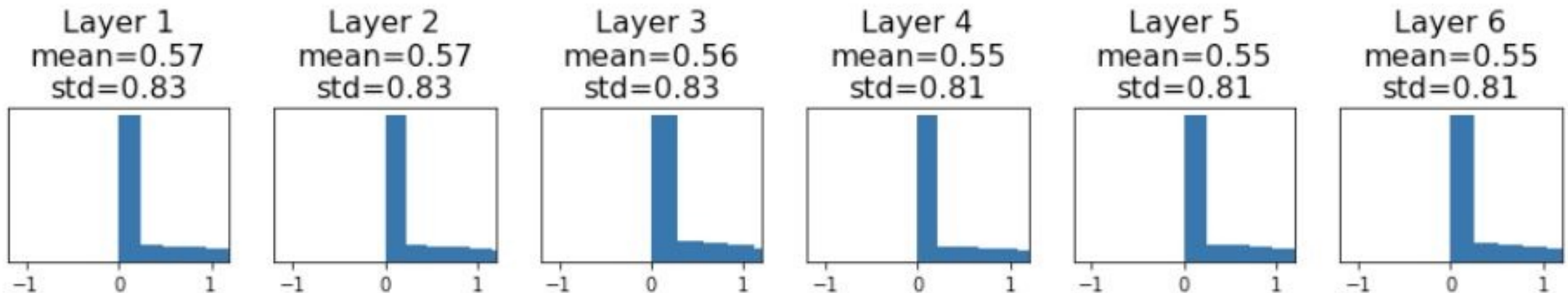
Weight Initialization: Kaiming / MSRA Initialization

Initialization: gaussian with zero mean and $1/\sqrt{\text{fan}_{\text{in}}/2}$ standard deviation

```
dims = [4096] * 7
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)
```

ReLU correction: $\text{std} = \sqrt{2 / \text{Din}}$

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



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

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

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 mini-batches); fiddle with architecture, learning rate, weight initialization

```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Take the first 20 examples from CIFAR-10
turn off regularization
use simple vanilla SGD

Loss not going down? LR too low, bad initialization

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

Choosing Hyperparameters

- 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: $1e-1$, $1e-2$, $1e-3$, $1e-4$

Hyperparameter search

- Larger Neural Networks typically require a long time to train
 - so performing hyperparameter search can take many days/weeks
- A single validation set of respectable size substantially simplifies the code base, without the need for cross-validation with multiple folds

Validation strategy

- **coarse -> fine** validation in stages
 - Zoom in to smaller region of hyperparameters and sample very densely in them
- **First stage:** only a few epochs to get rough idea of what params work
- **Second stage:** longer running time, finer search ... (repeat as necessary)

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**

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

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)

    trainer = ClassifierTrainer()
    model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
    trainer = ClassifierTrainer()
    best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                           model, two_layer_net,
                                           num_epochs=5, reg=reg,
                                           update='momentum', learning_rate_decay=0.9,
                                           sample_batches = True, batch_size = 100,
                                           learning_rate=lr, verbose=False)
```

note it's best to optimize
in log space!

val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)

nice

Now run finer search

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)
```

adjust range



```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-4, 0)
    lr = 10**uniform(-3, -4)
```

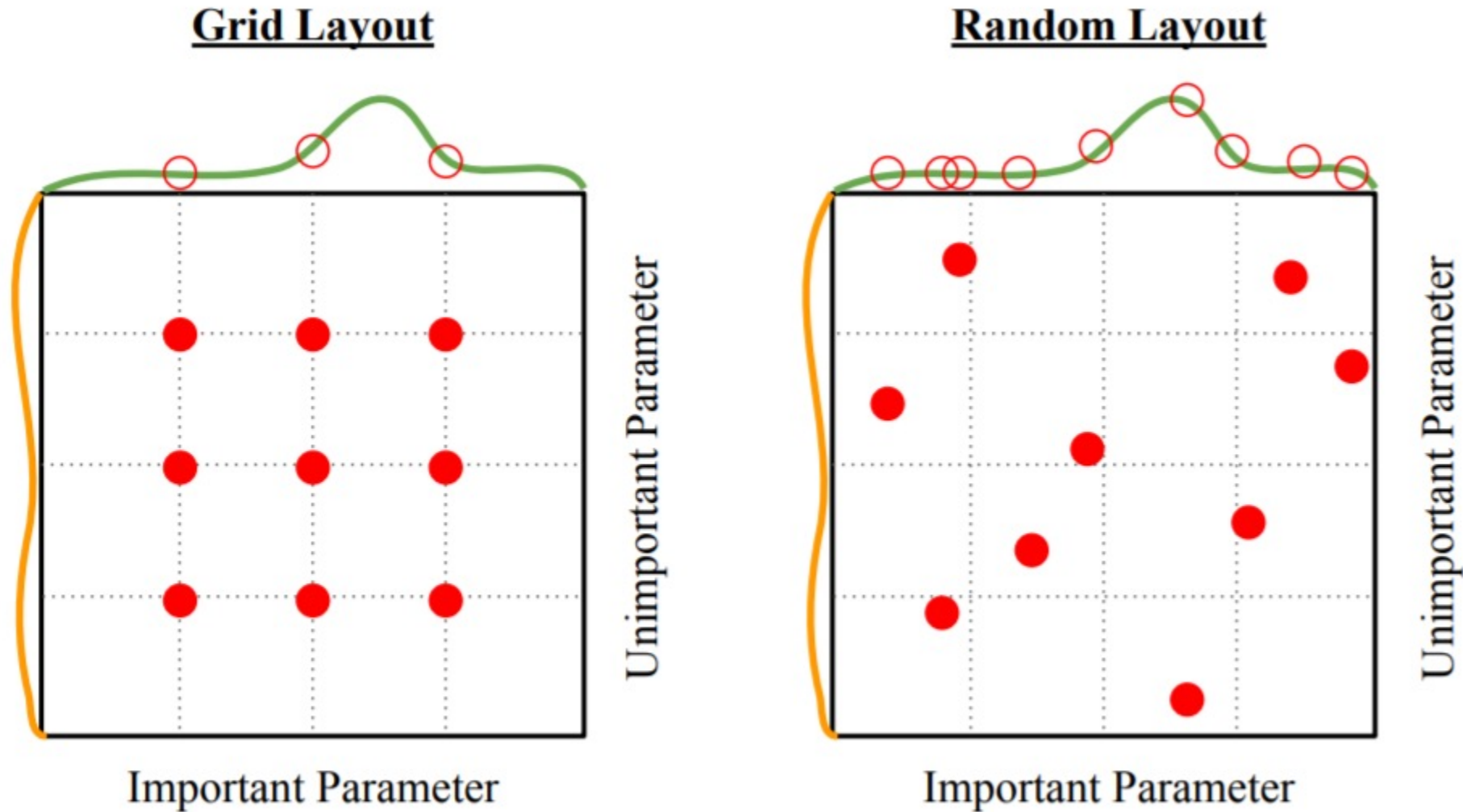
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)

But this best cross-validation result is worrying. Why?

Careful with best values on border

Random search vs. grid search

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

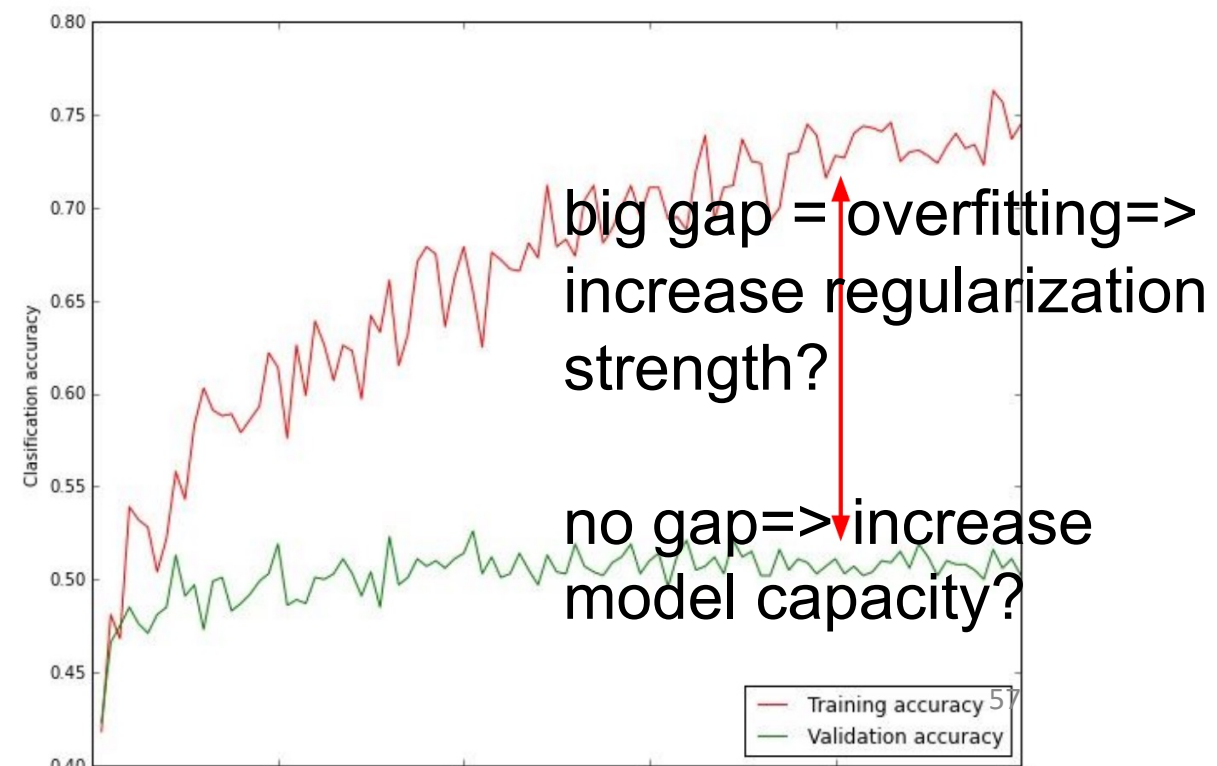
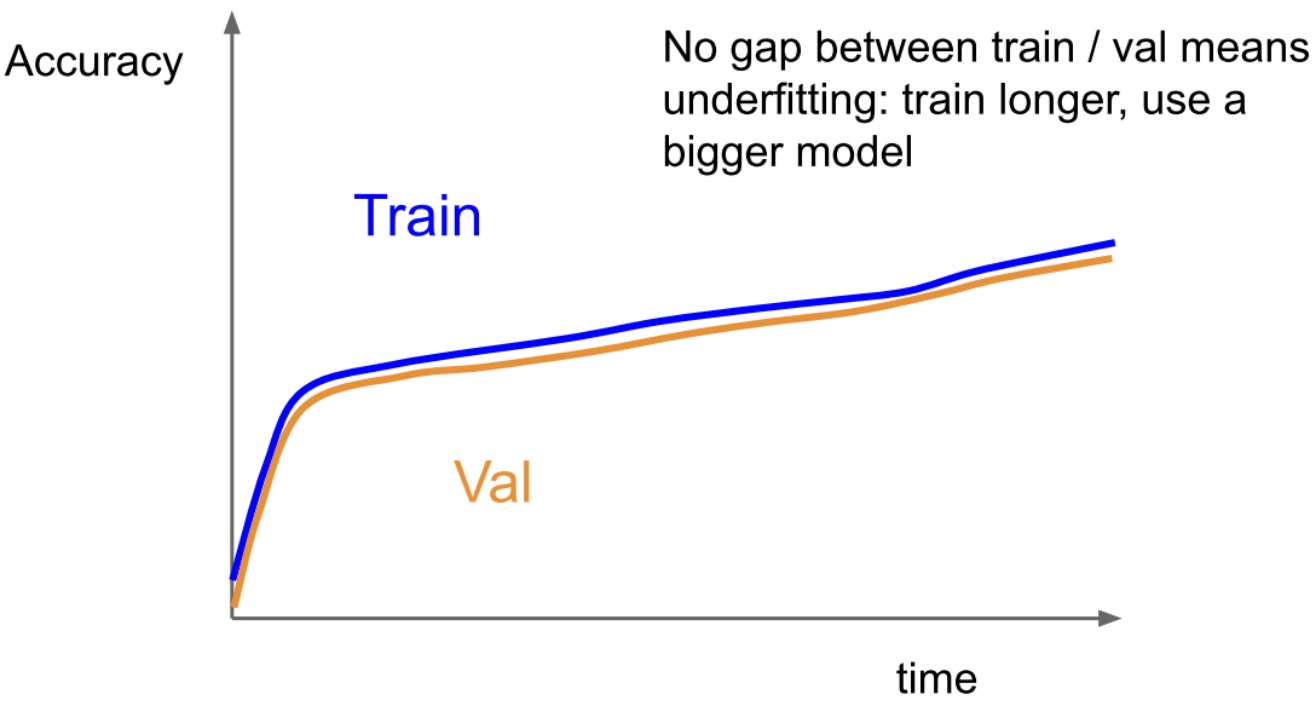
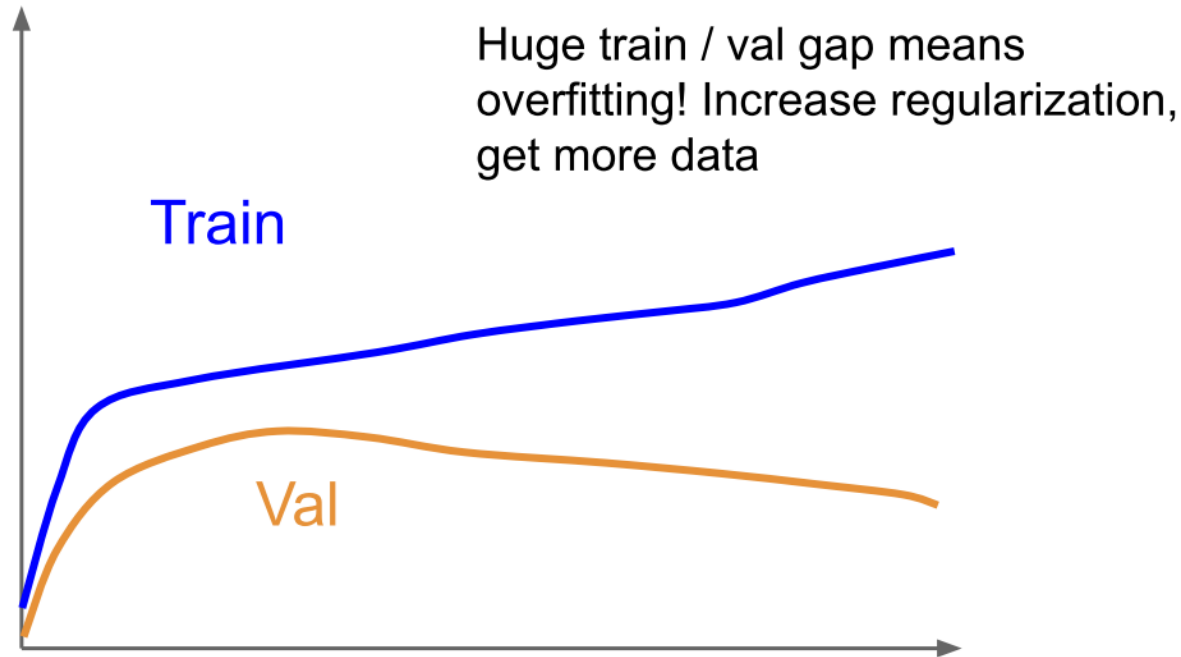
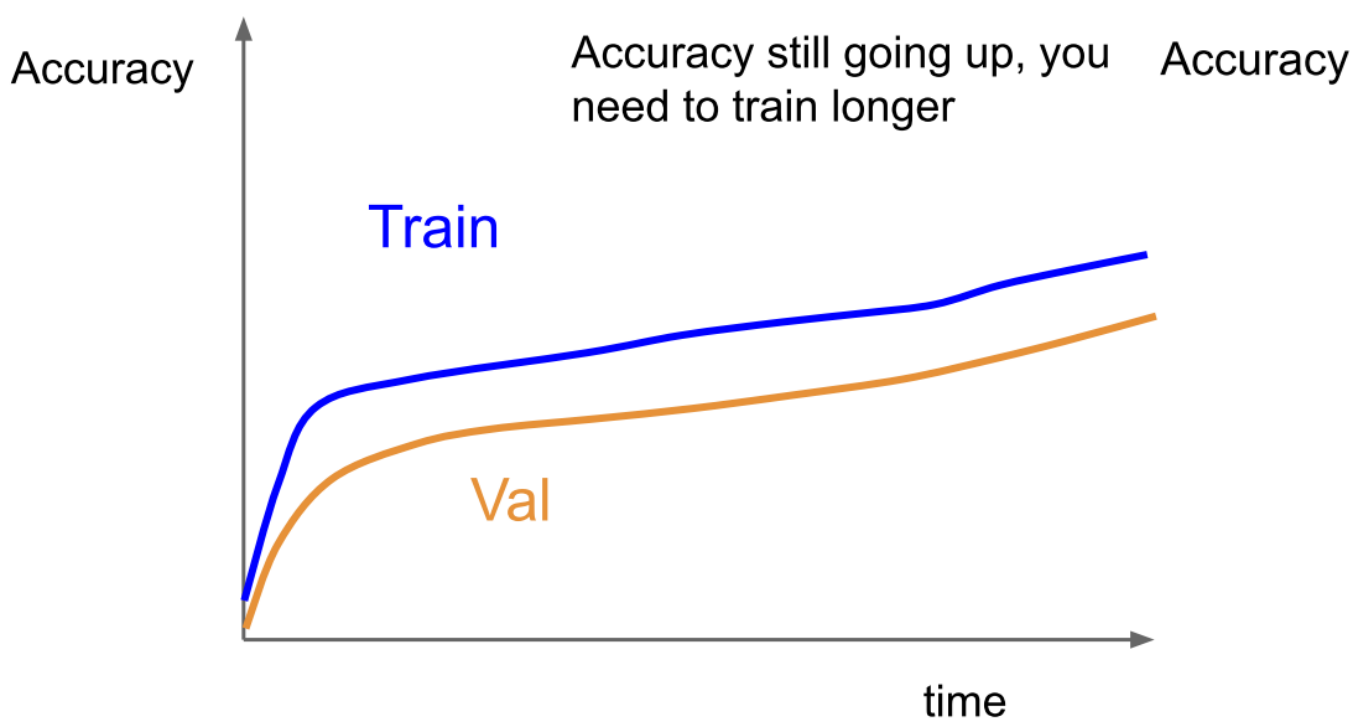


Random search

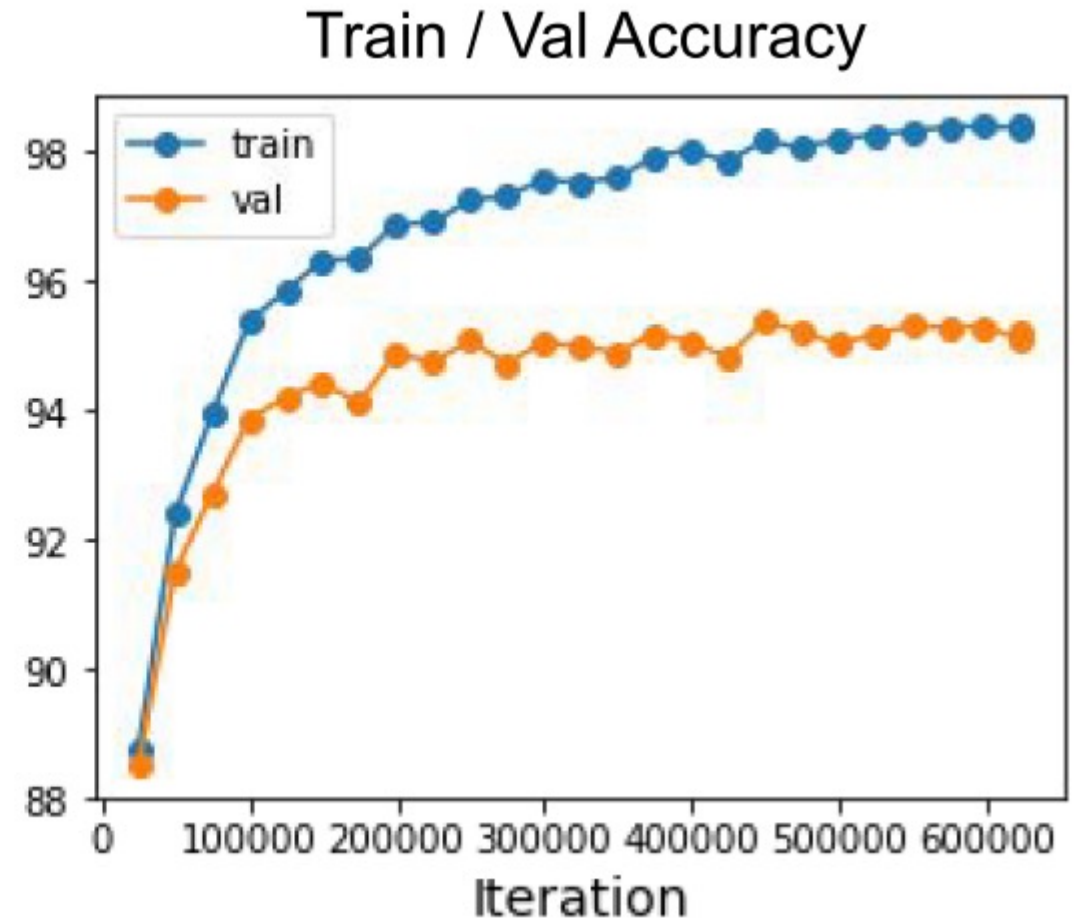
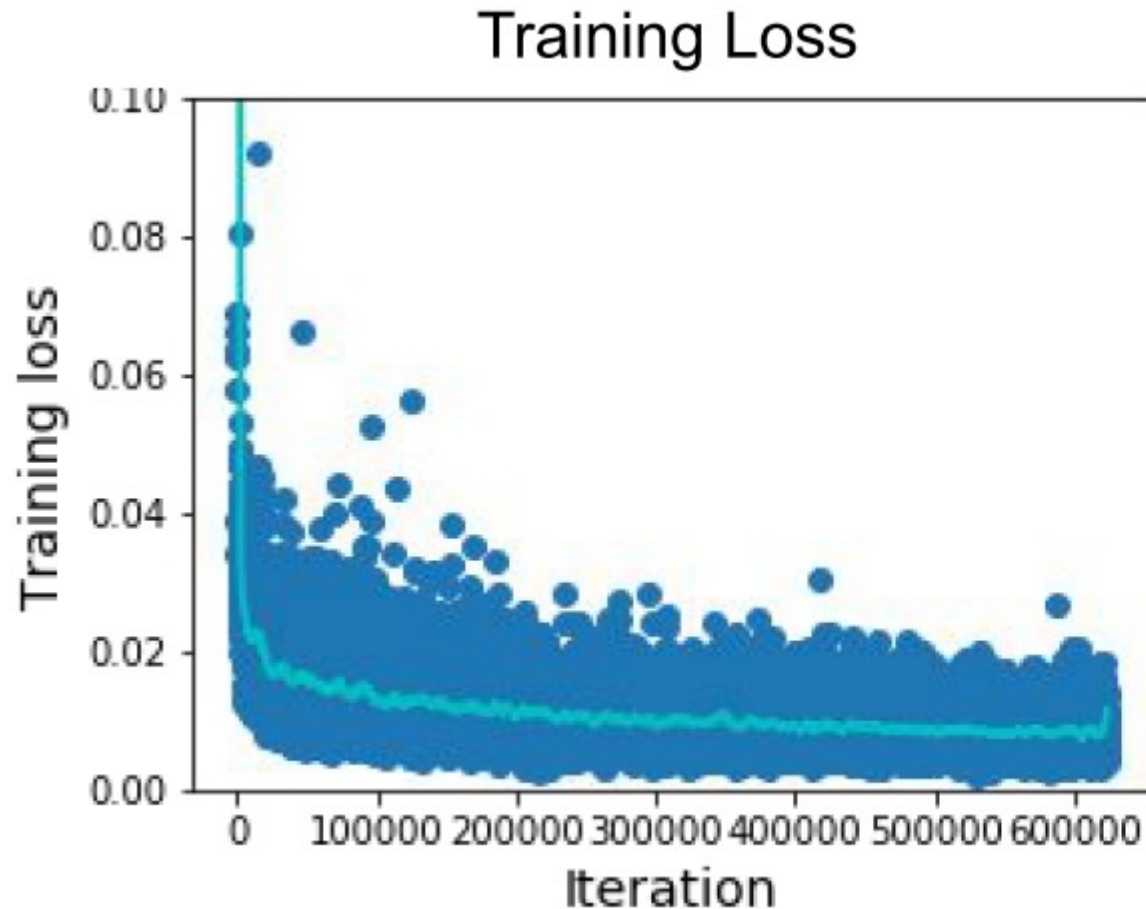
- Random search is more effective and richly exploring when:
 - The performance may not be such sensitive to the value of all hyperparameters (some parameters are actually much more important)
 - We don't know it in advance
 - More distinct values of the more important hyperparameter are tried.
 - Especially when the number of hyperparameters becomes larger

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

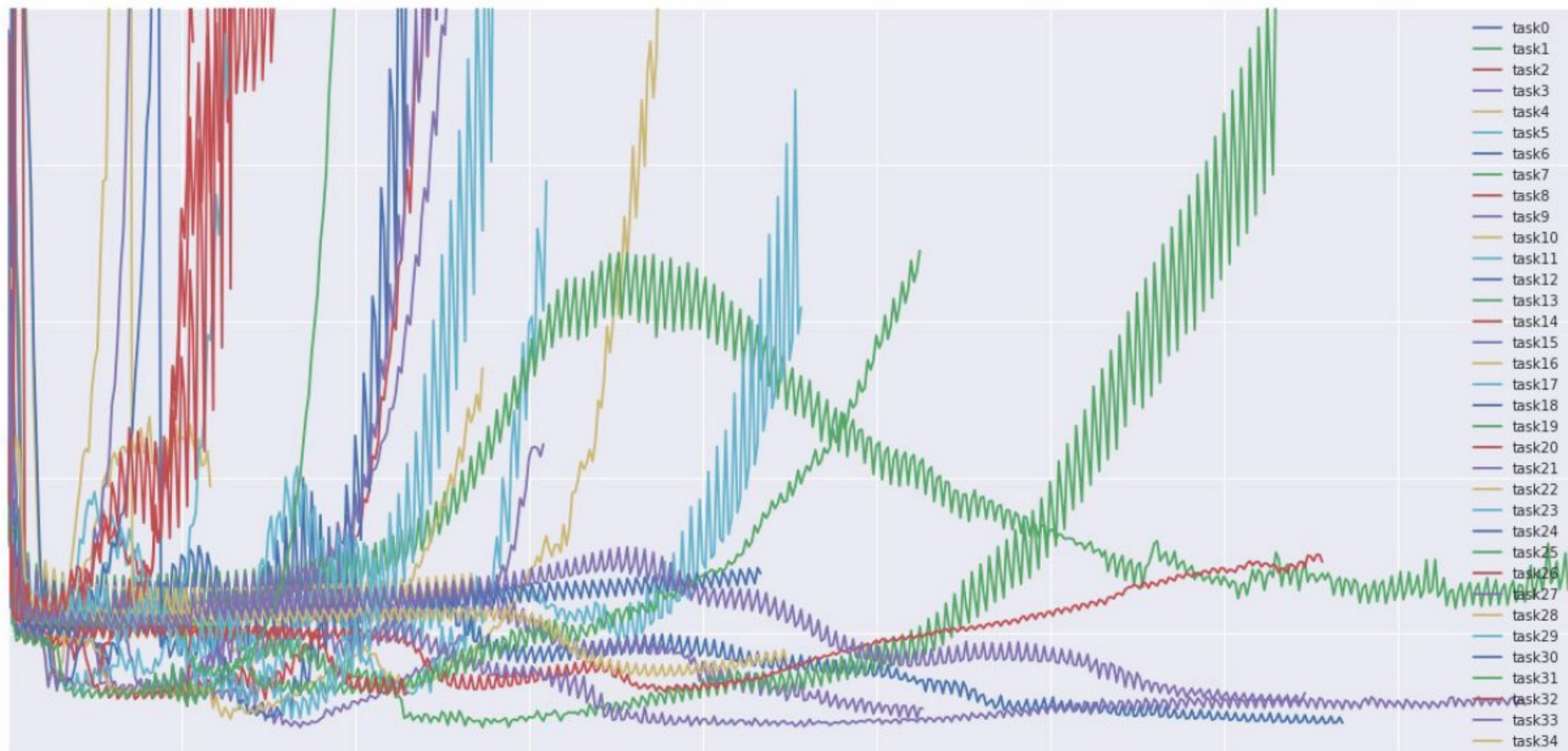


Look at learning curves



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

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



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**

Hyperparameter selection

- Finding a good set of hyperparameters to provide better convergence
 - initial **learning rate** α
 - regularization strength (L2 penalty, dropout strength)
 - # of hidden units and # of layers
 - mini-batch size
 - decay schedule (such as the decay constant), update type
 - parameters of optimization algorithms (momentum, adam, ...)
 - These are usually fixed to $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ or 10^{-7}

Babysitting one model vs. training models in parallel

- When we do not have sufficient computational resources to train a lot of models
 - Watching performance of one model during the time and tune its parameters by nudging them up and down
- Train many different models in parallel (with different hyperparameters) and just pick the one that works best
- Babysitting one model called **Panda** while training many models in parallel called **Cavier** approach for selecting hyperparameters

Summary

- Activation Functions (use ReLU)
- Data Preprocessing (subtract mean and some times scale according to standard deviation)
- Weight Initialization (use Xavier init)
- Hyper-parameter Optimization (random sample hyperparams, in log space when appropriate)
- Babysitting the learning process

Resource

- Please see the following notes:
 - <http://cs231n.github.io/neural-networks-2/>
 - <http://cs231n.github.io/neural-networks-3/>