CS 4644-DL / 7643-A: LECTURE 10
DANFEI XU

Topics:

- Convolutional Neural Networks Architectures (cont.)
- Training Neural Networks (Part 1)

Administrative

- PS2/HW2 out : **Most difficult assignment. Start early!**
- Project proposal due Sep 27th

CNN Architectures

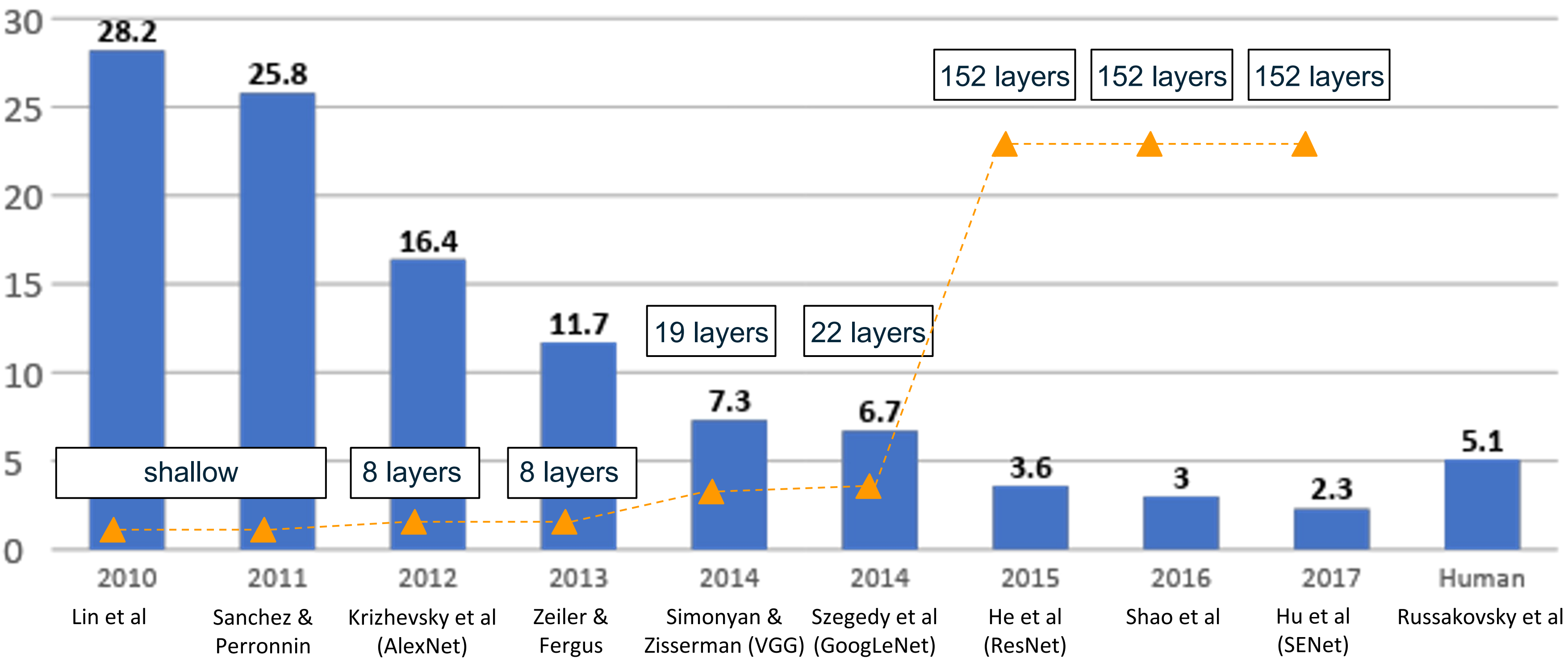
Case Studies

- AlexNet
- VGG
- GoogLeNet
- ResNet

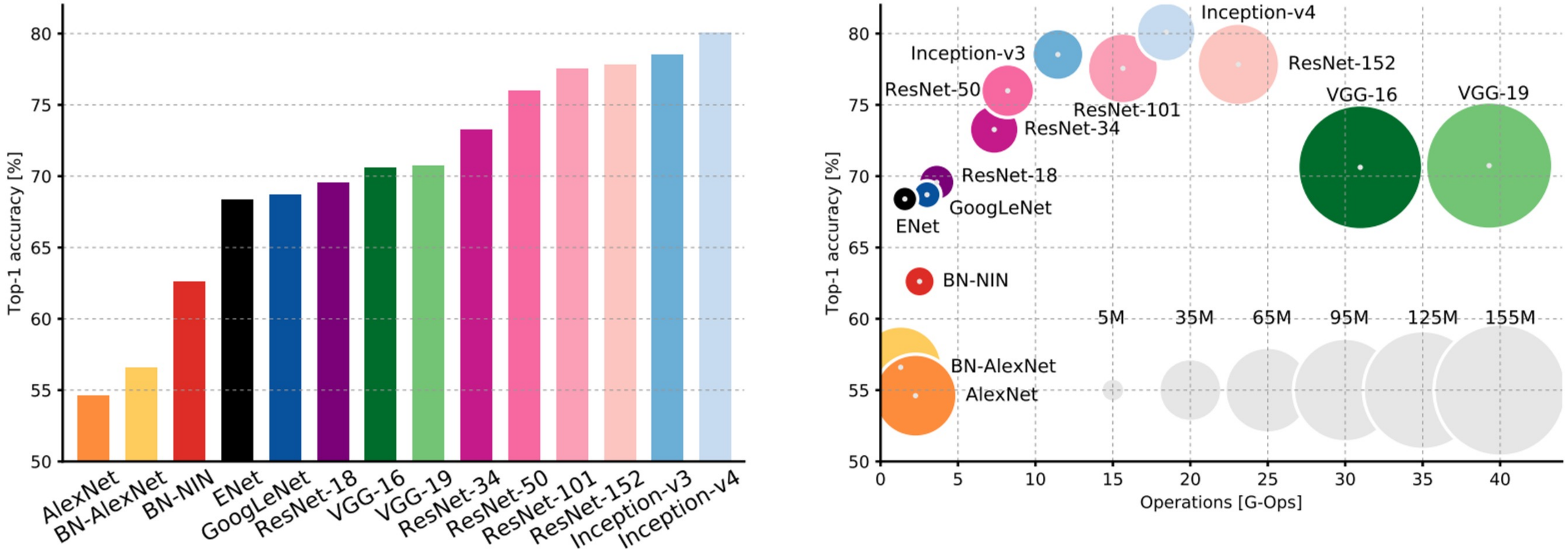
Also....

- SENet
- Wide ResNet
- ResNeXT
- DenseNet
- MobileNets
- NASNet
- EfficientNet

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



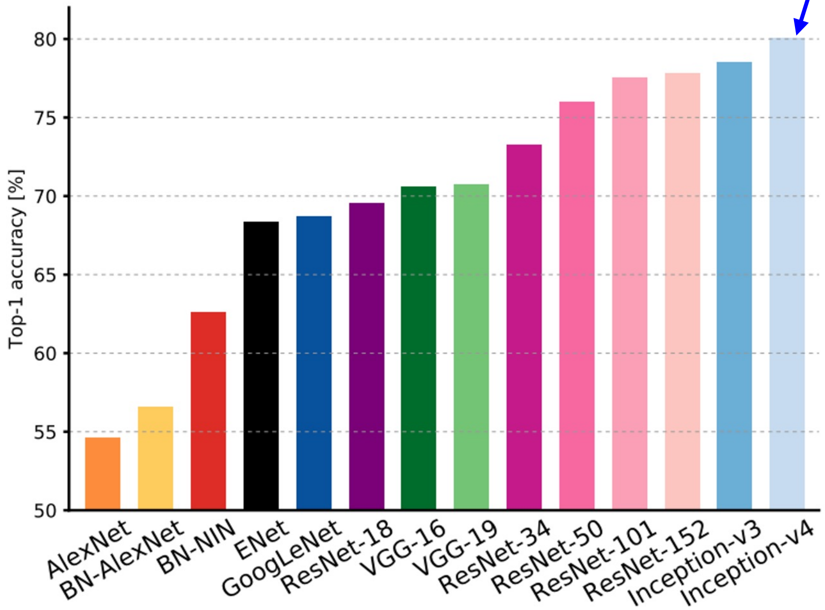
Comparing complexity...

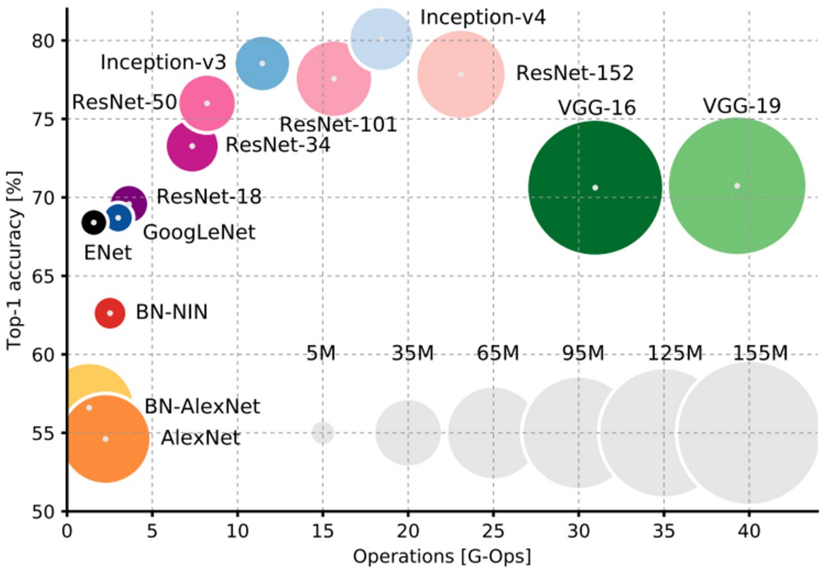


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Comparing complexity...

Inception-v4: Resnet + Inception!

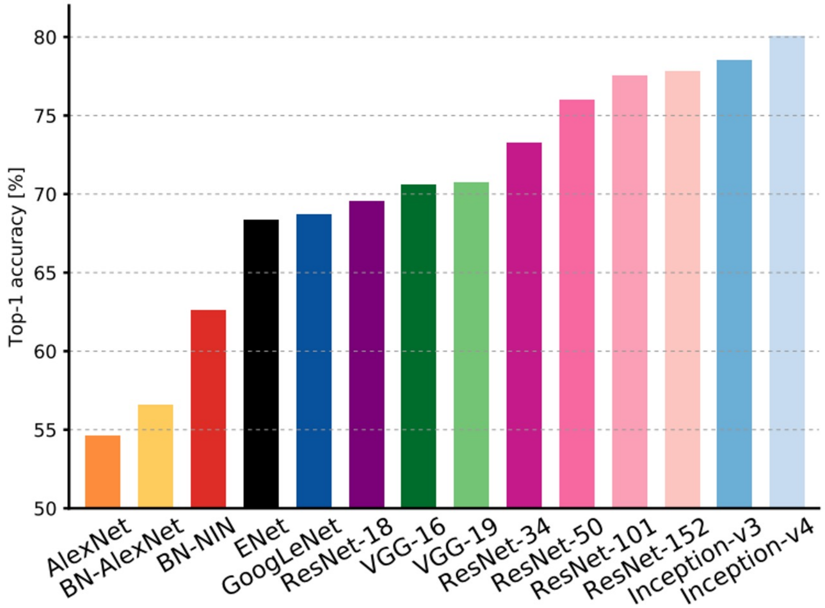


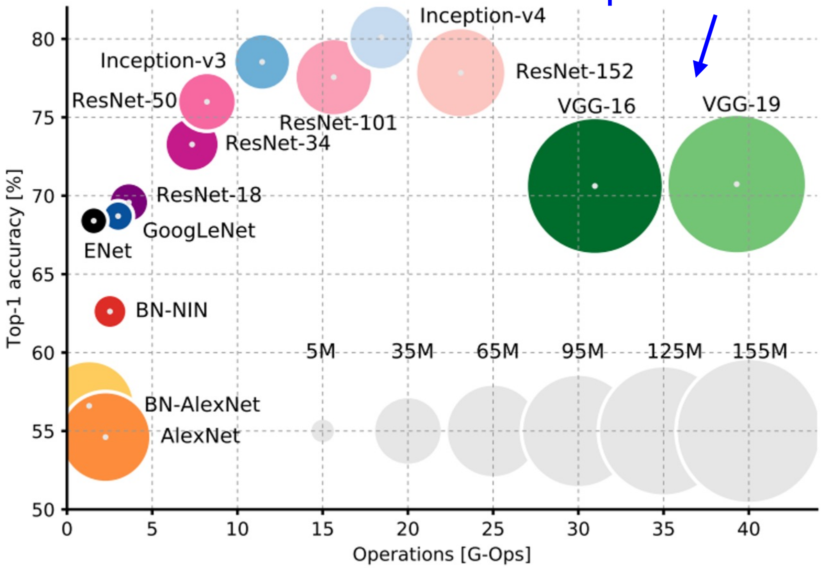


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Comparing complexity...

VGG: most parameters, most operations

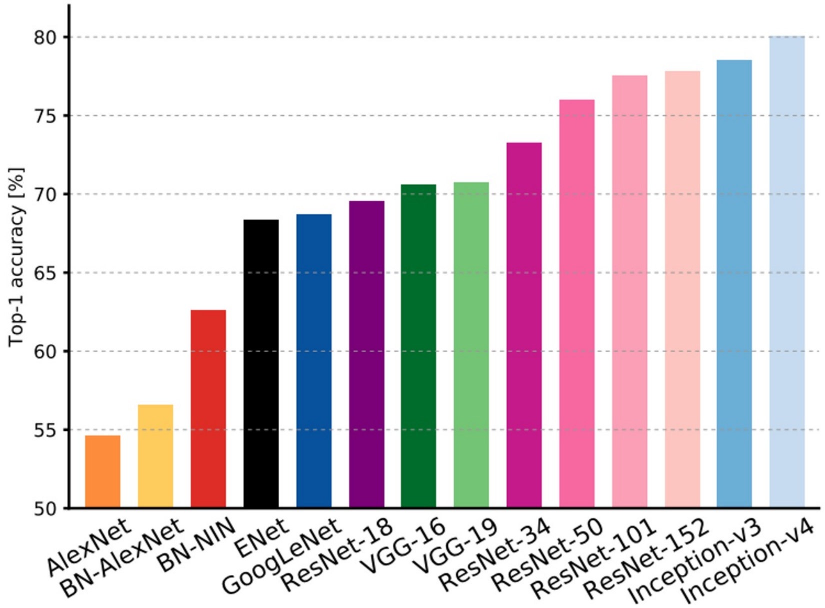


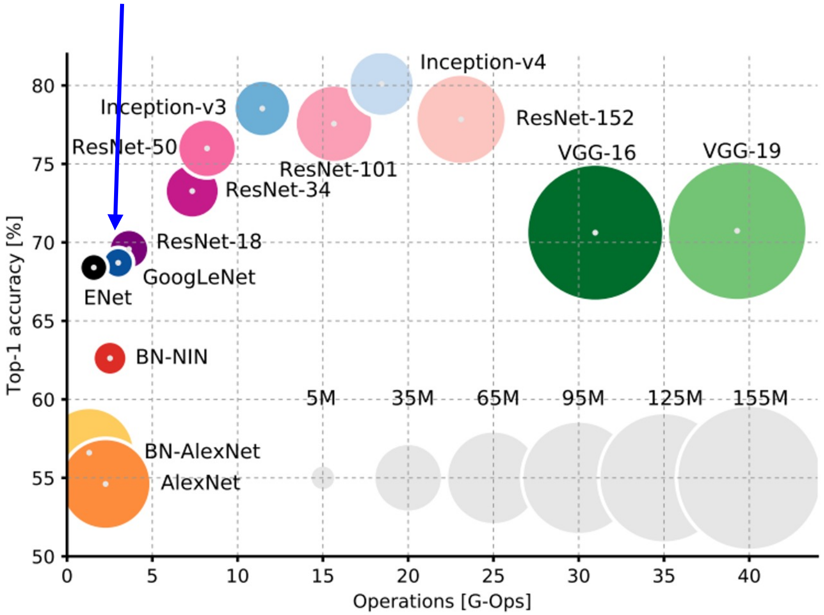


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Comparing complexity...

GoogLeNet:
most efficient

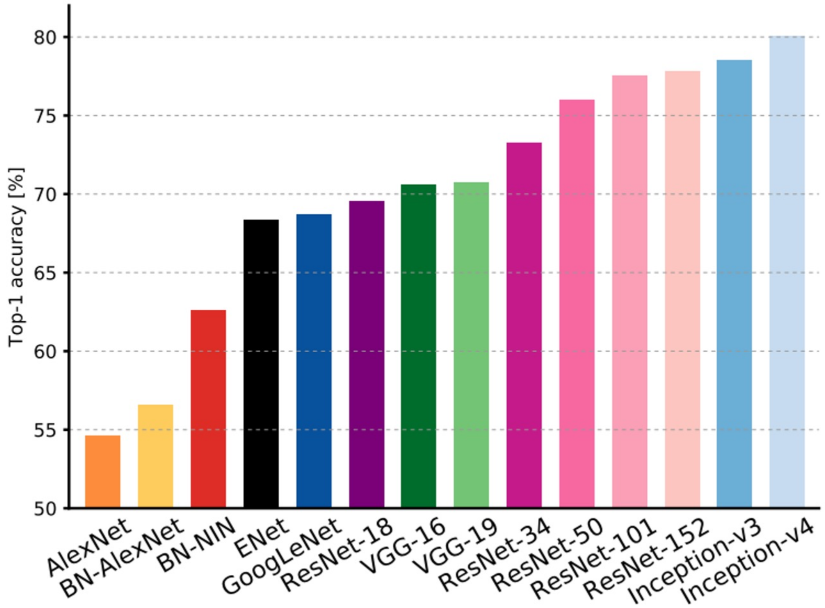


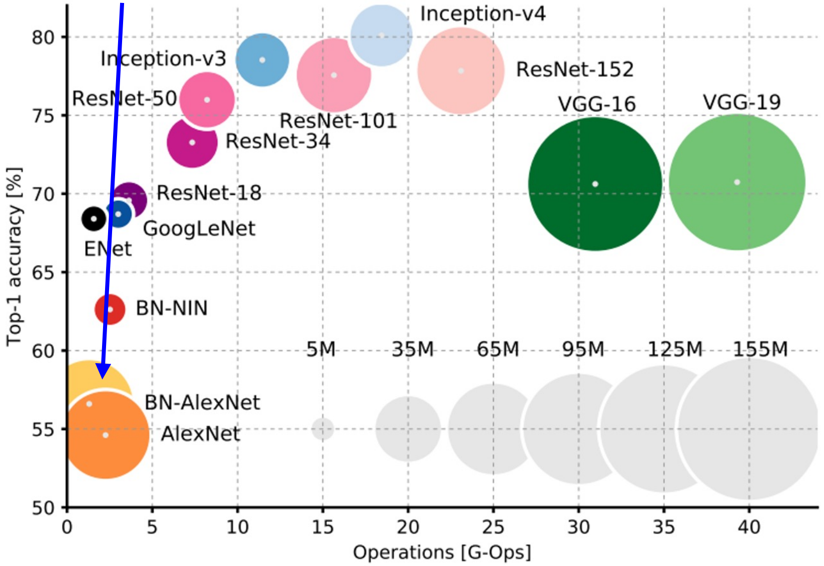


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Comparing complexity...

AlexNet:
Smaller compute, still memory heavy, lower accuracy

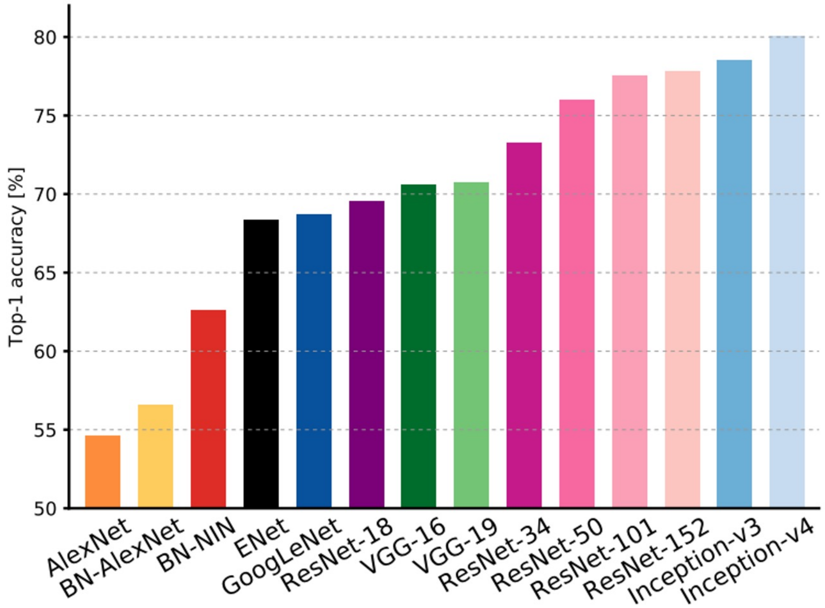


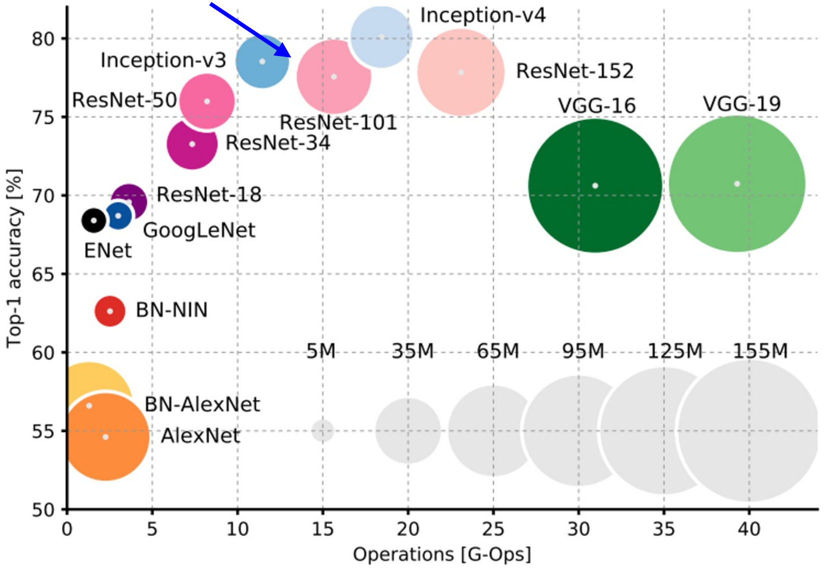


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Comparing complexity...

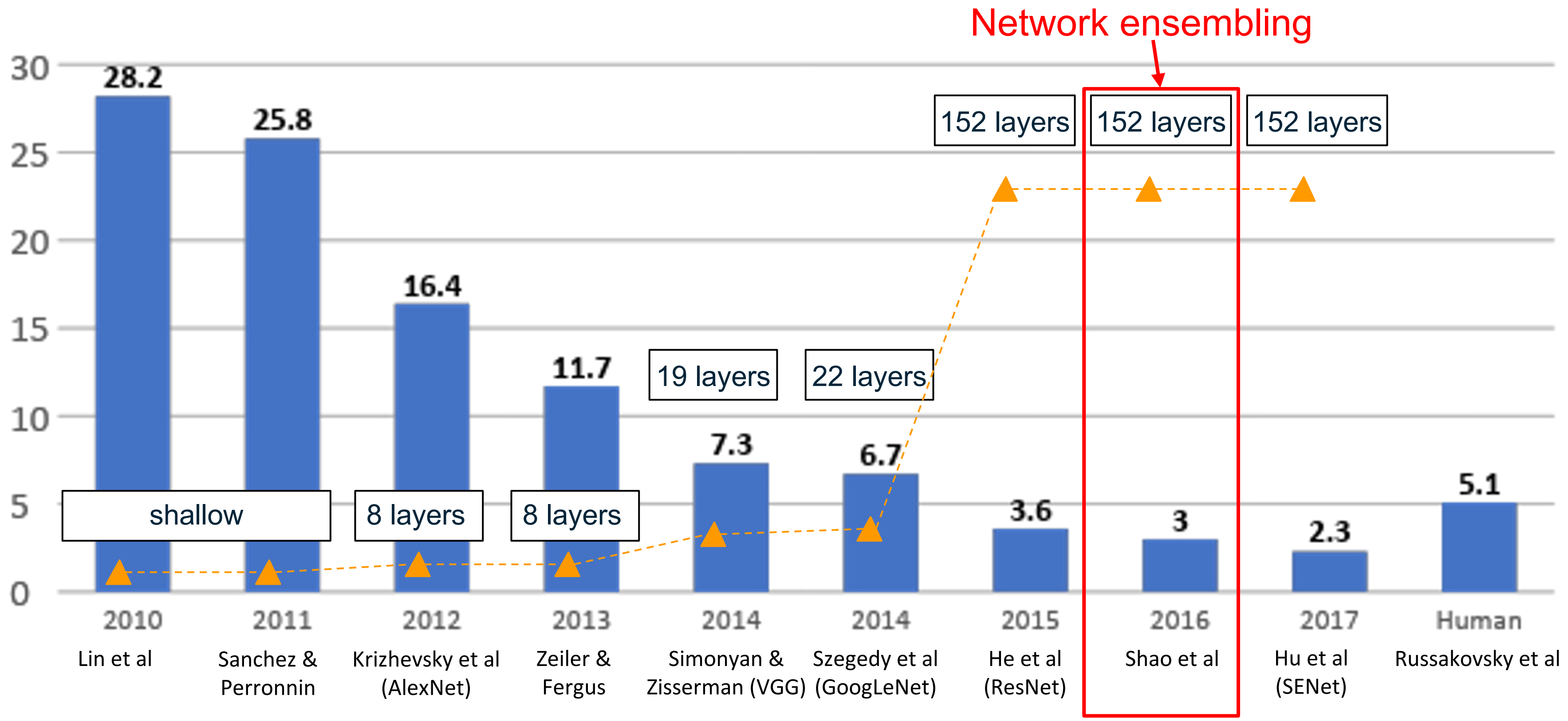
ResNet:
Moderate efficiency depending on model, highest accuracy





An Analysis of Deep Neural Network Models for Practical Applications, 2017.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Improving ResNets...

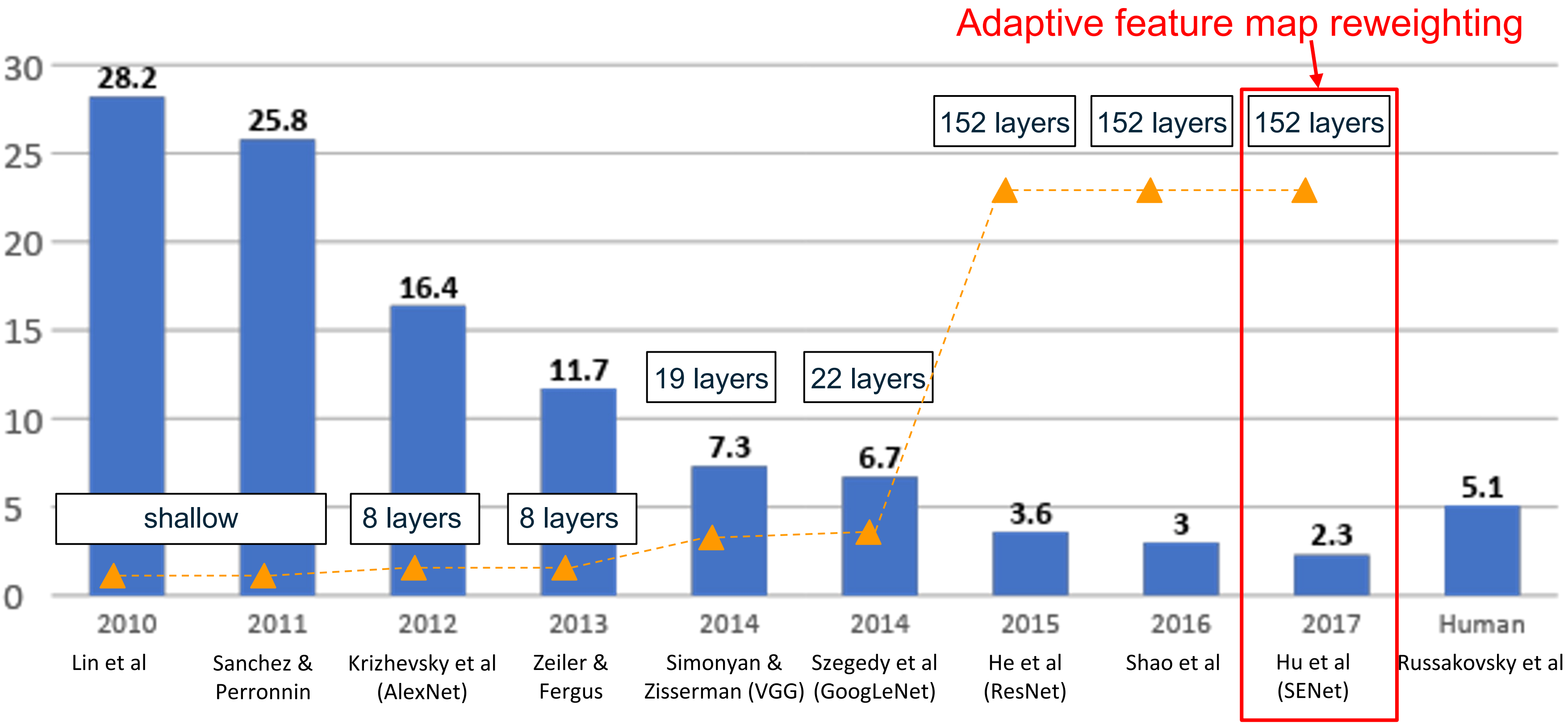
“Good Practices for Deep Feature Fusion”

[Shao et al. 2016]

- Multi-scale ensembling of Inception, Inception-Resnet, Resnet, Wide Resnet models
- ILSVRC’16 classification winner

	Inception-v3	Inception-v4	Inception-Resnet-v2	Resnet-200	Wrn-68-3	Fusion（Val.）	Fusion（Test）
Err. (%)	4.20	4.01	3.52	4.26	4.65	2.92 (-0.6)	2.99

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

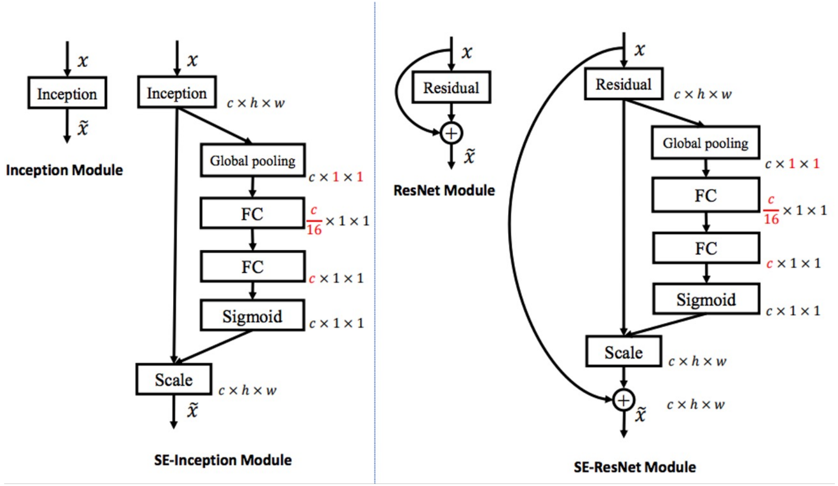


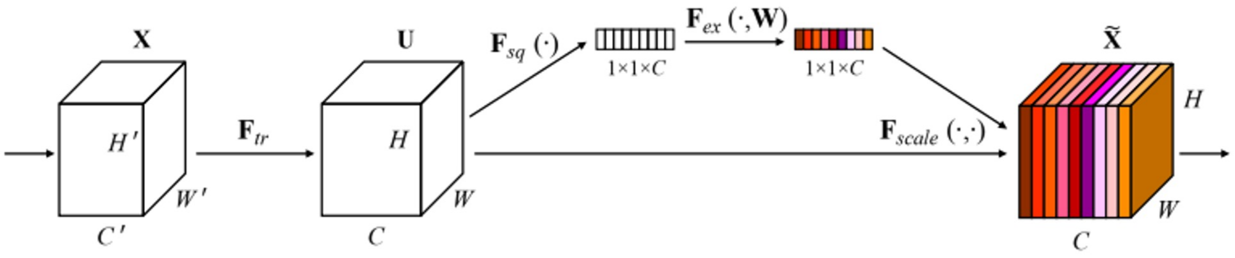
Improving ResNets...

Squeeze-and-Excitation Networks (SENet)

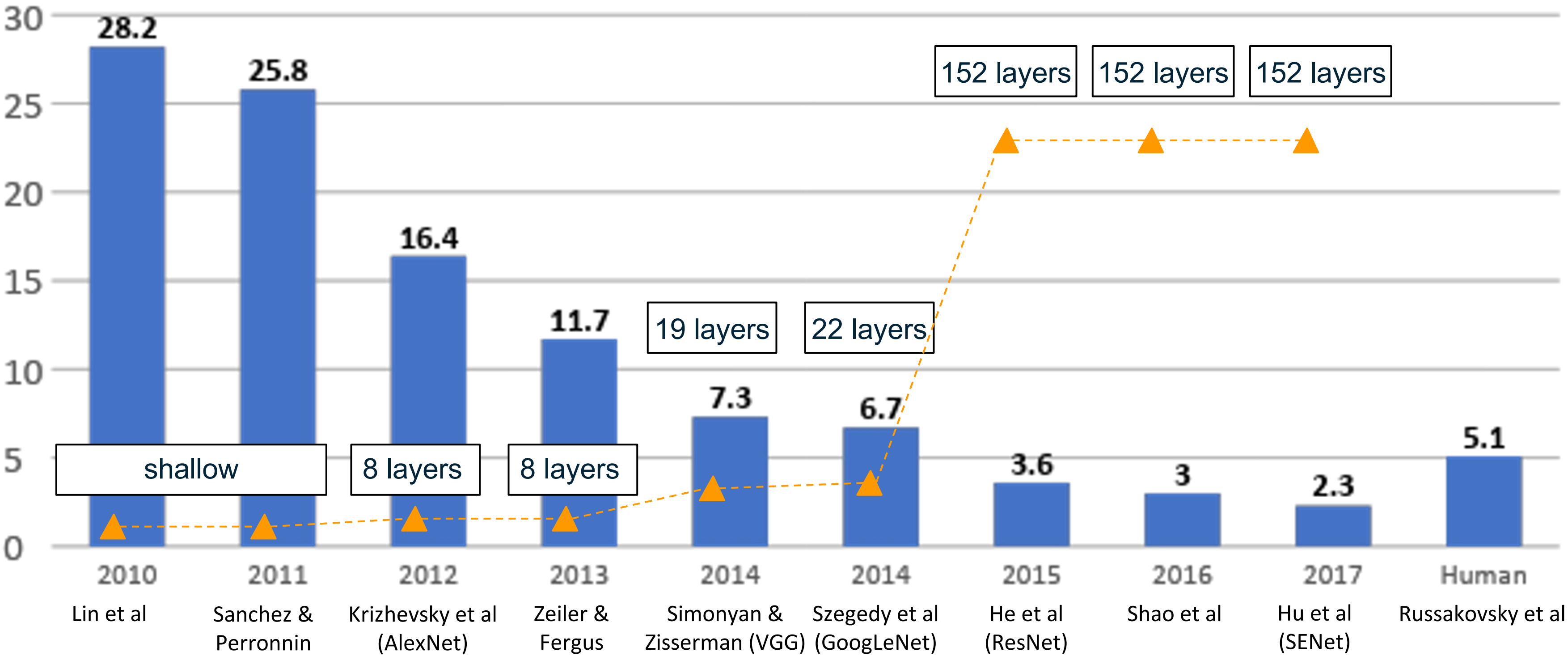
[Hu et al. 2017]

- Add a "feature recalibration" module that learns to adaptively reweight feature maps
- Global information (global avg. pooling layer) + 2 FC layers used to determine feature map weights
- ILSVRC'17 classification winner (using ResNeXt-152 as a base architecture)

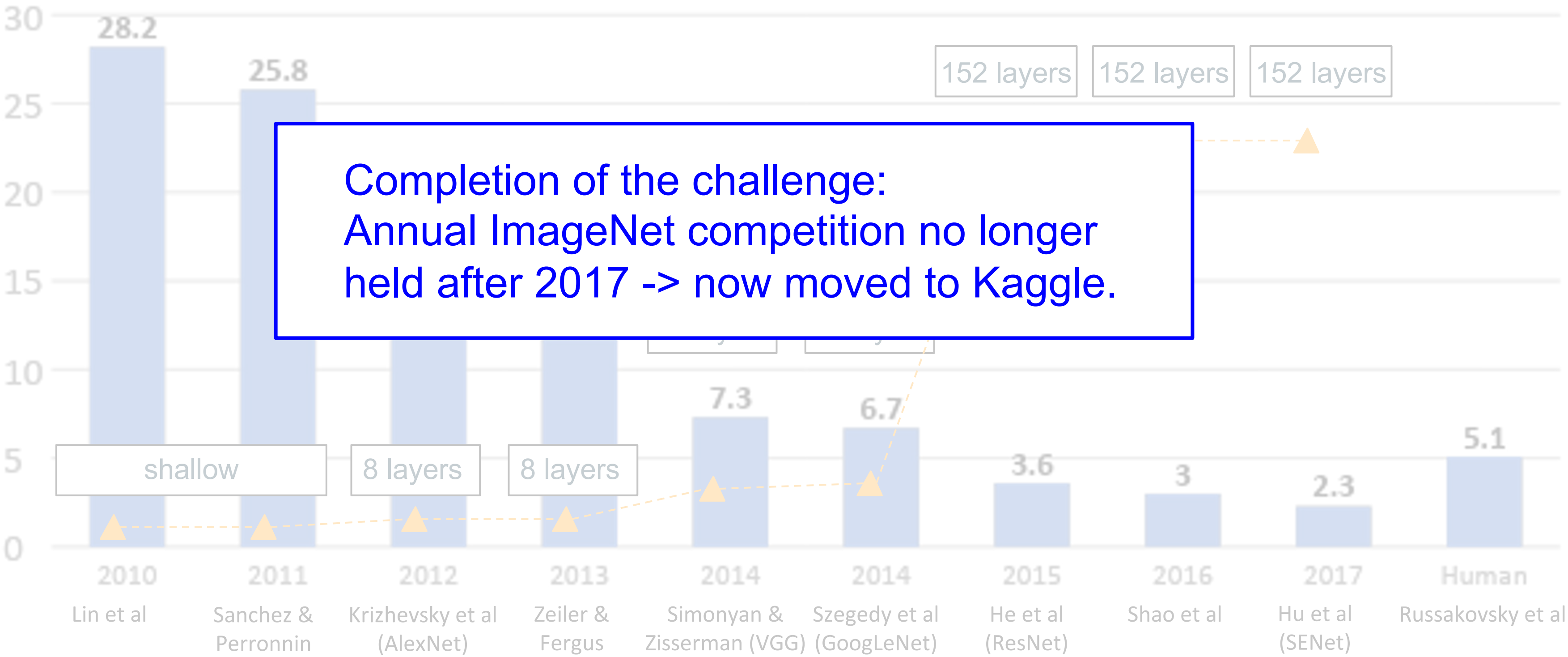




ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



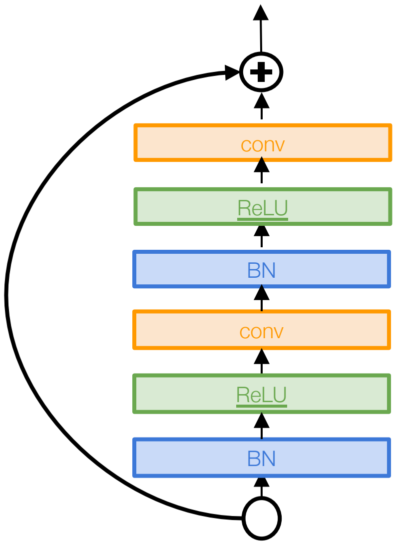
Completion of the challenge:
Annual ImageNet competition no longer held after 2017 -> now moved to Kaggle.

But research into CNN architectures is still flourishing

Identity Mappings in Deep Residual Networks

[He et al. 2016]

- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network
- Gives better performance

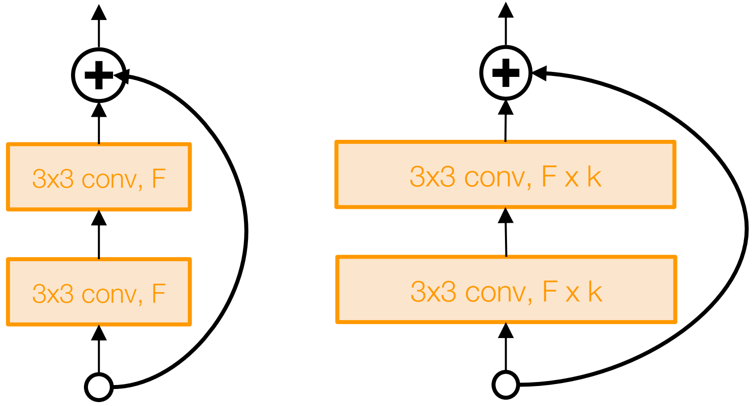


Improving ResNets...

Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- User wider residual blocks (F x k filters instead of F filters in each layer)
- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth more computationally efficient (parallelizable)



Basic residual block

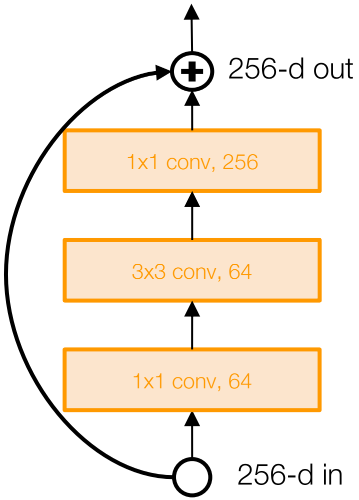
Wide residual block

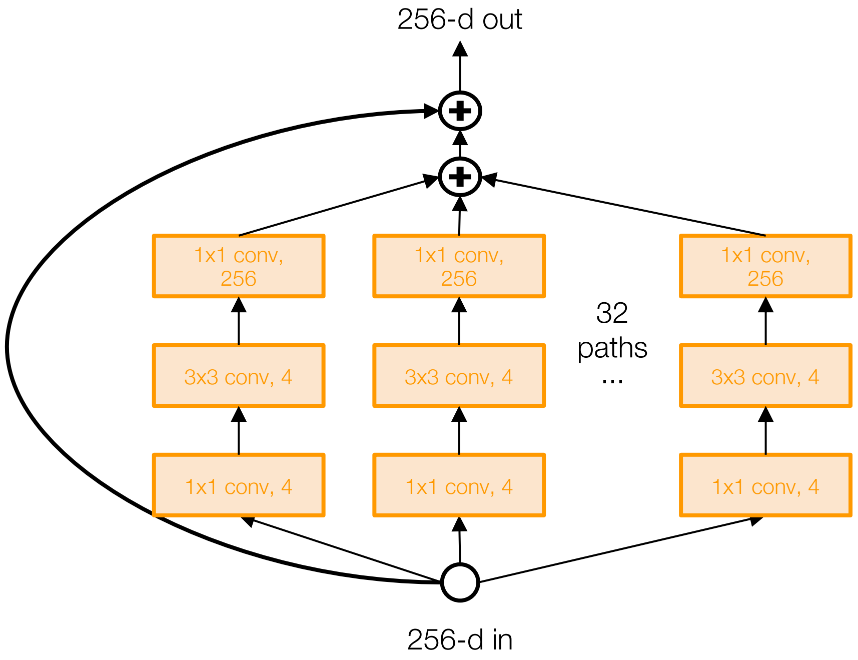
Improving ResNets...

Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways (“cardinality”)
- Parallel pathways similar in spirit to Inception module



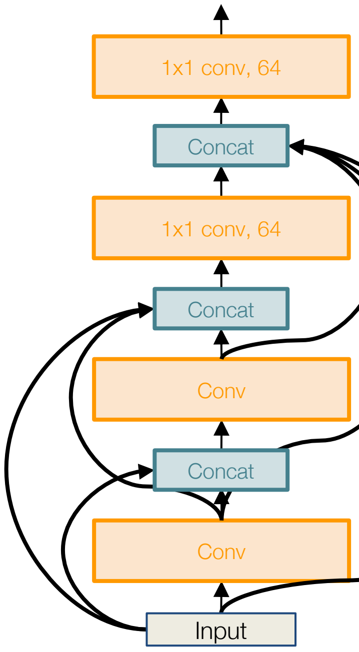


Other ideas...

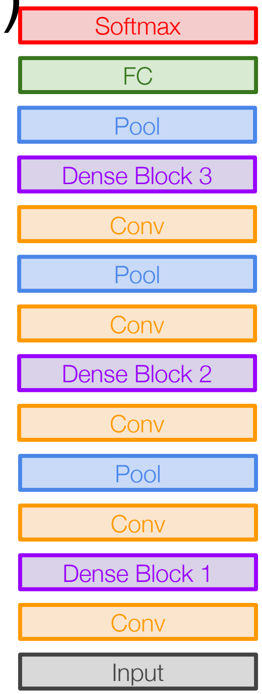
Densely Connected Convolutional Networks (DenseNet)

[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse
- Showed that shallow 50-layer network can outperform deeper 152 layer ResNet



Dense Block

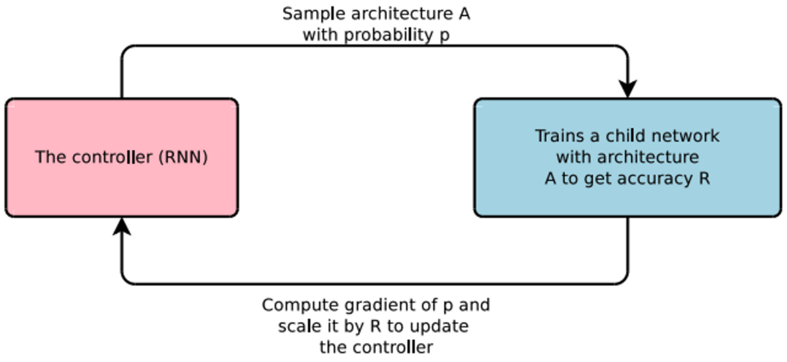


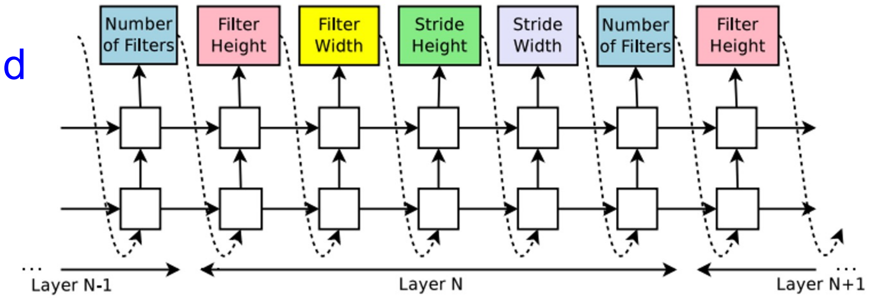
Learning to search for network architectures...

Neural Architecture Search with Reinforcement Learning (NAS)

[Zoph et al. 2016]

- "Controller" network that learns to design a good network architecture (output a string corresponding to network design)
- Iterate:
  1) Sample an architecture from search space
  2) Train the architecture to get a "reward" R corresponding to accuracy
  3) Compute gradient of sample probability, and scale by R to perform controller parameter update (i.e. increase likelihood of good architecture being sampled, decrease likelihood of bad architecture)



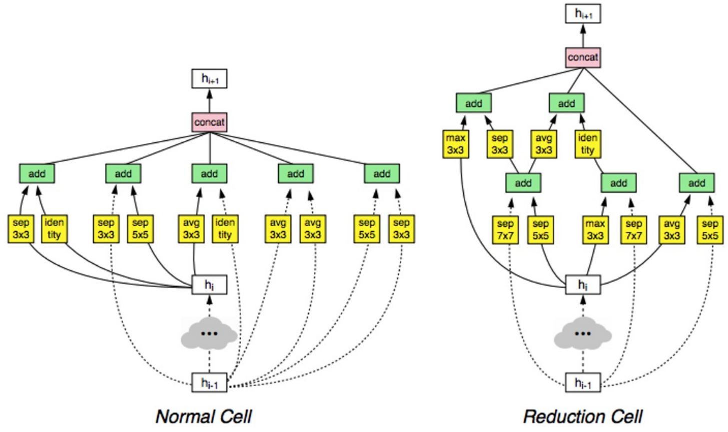


Learning to search for network architectures...

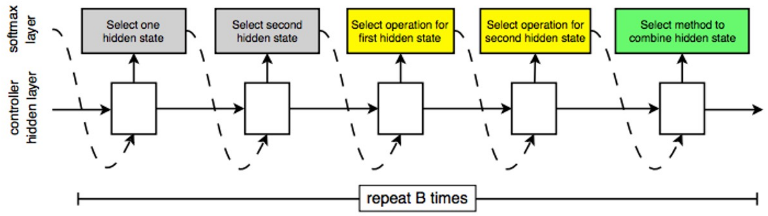
Learning Transferable Architectures for Scalable Image Recognition

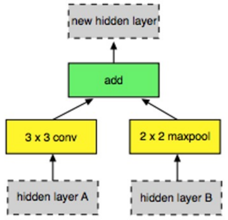
[Zoph et al. 2017]

- Applying neural architecture search (NAS) to a large dataset like ImageNet is expensive
- Design a search space of building blocks (“cells”) that can be flexibly stacked
- NASNet: Use NAS to find best cell structure on smaller CIFAR-10 dataset, then transfer architecture to ImageNet



- Many follow-up works in this space e.g. AmoebaNet (Real et al. 2019) and ENAS (Pham, Guan et al. 2018)





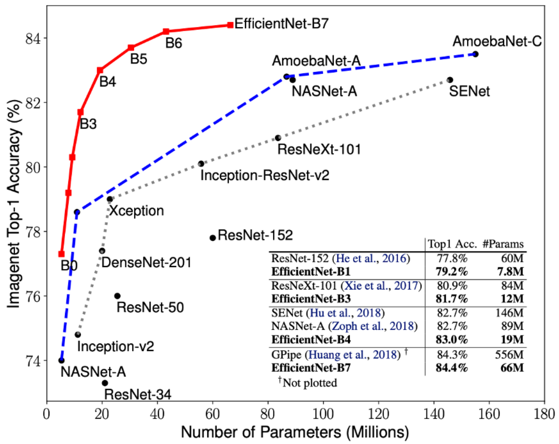
But sometimes smart heuristic is better than NAS ...

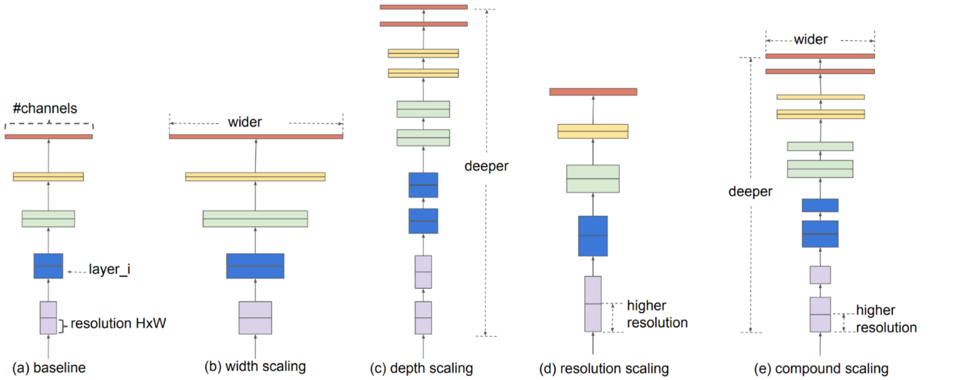
EfficientNet: Smart Compound Scaling

[Tan and Le. 2019]

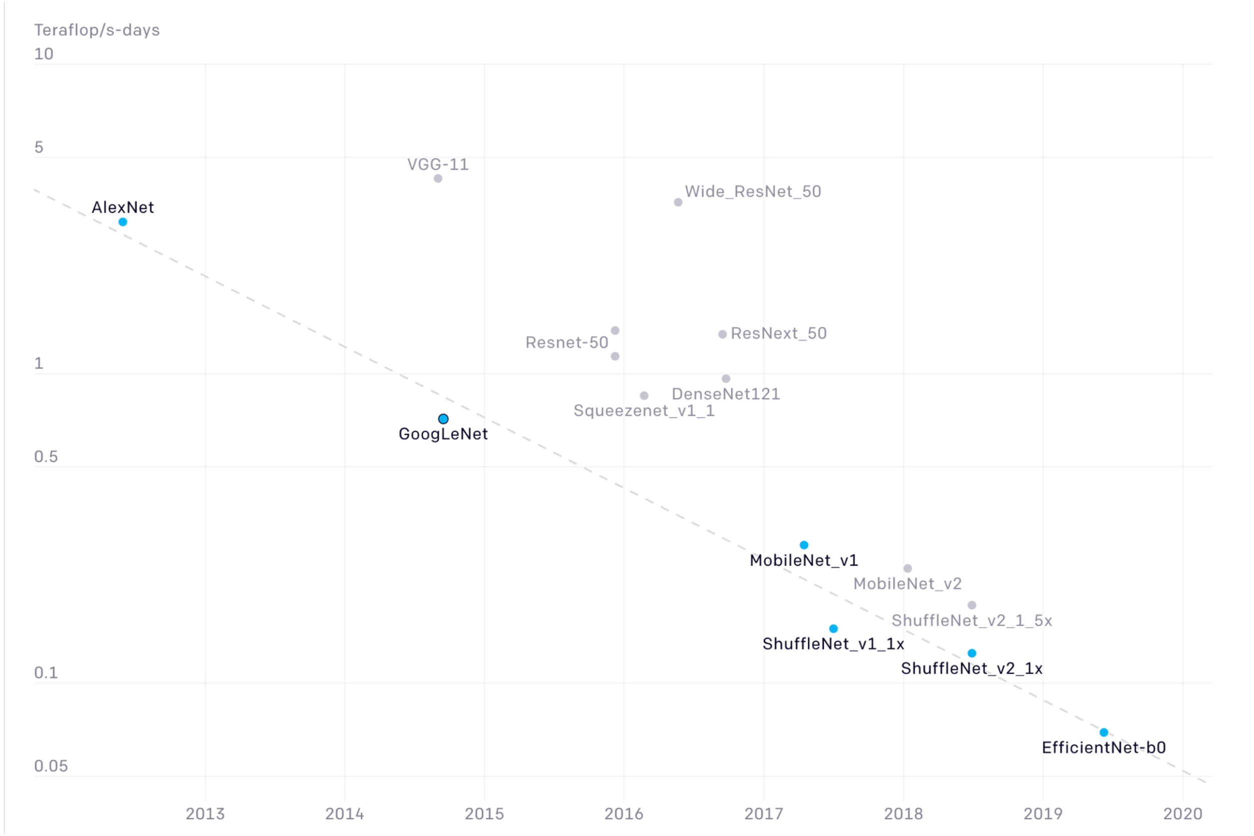
- Increase network capacity by scaling width, depth, and resolution, while balancing accuracy and efficiency.
- Search for optimal set of compound scaling factors given a compute budget (target memory & flops).
- Scale up using smart heuristic rules

$$
\begin{aligned}
\text{depth: } & d = \alpha^{\phi} \\
\text{width: } & w = \beta^{\phi} \\
\text{resolution: } & r = \gamma^{\phi} \\
\text{s.t. } & \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \\
& \alpha \geq 1, \beta \geq 1, \gamma \geq 1
\end{aligned}
$$





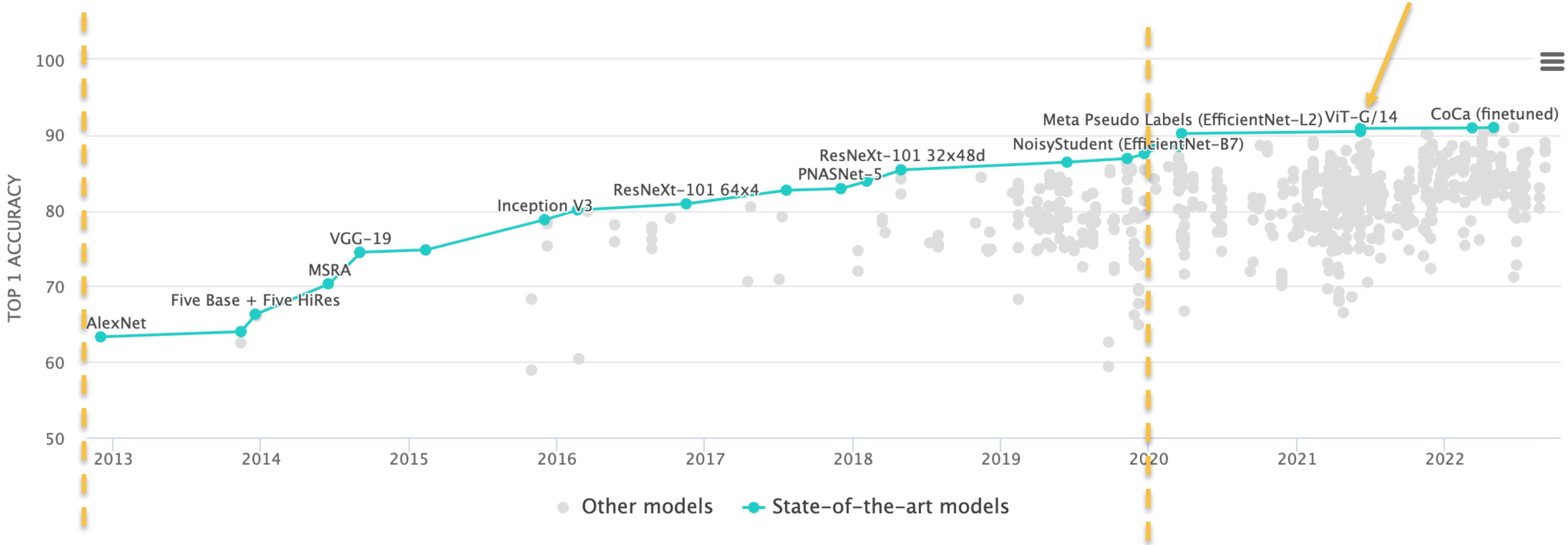
Efficient networks...



https://openai.com/blog/ai-and-efficiency/

Today's Lecture

Transformer



https://paperswithcode.com/sota/image-classification-on-imagenet

What we have learned so far …

Deep Neural Networks:

- What they are (composite parametric, non-linear functions)
- Where they come from (biological inspiration, brief history of ANN)
- How they are optimized, in principle (analytical gradient via computational graphs, backpropagation)
- What they look like in practice (Deep ConvNets for vision)

Next few lectures:

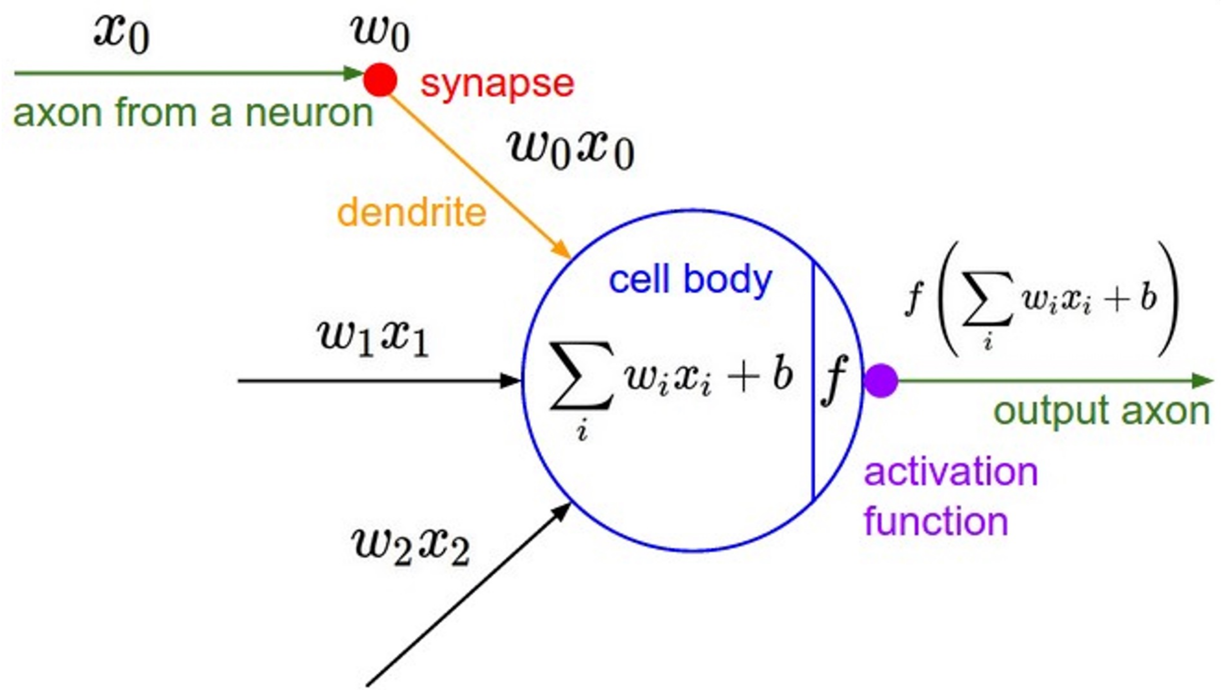
Training Deep Neural Networks

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Regularization
- Advanced Optimization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
- Model Ensemble

Today: Training Deep NNs (Part 1)

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization

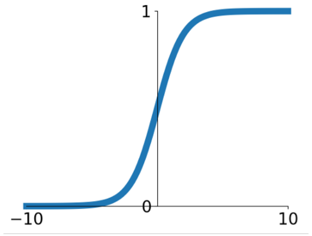
Activation Functions



Activation Functions

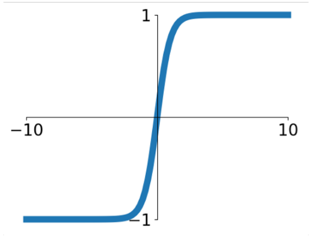
Sigmoid

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



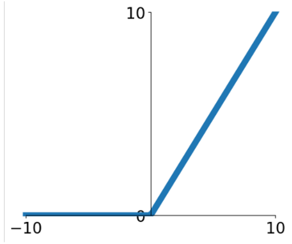
tanh

$\tanh(x)$



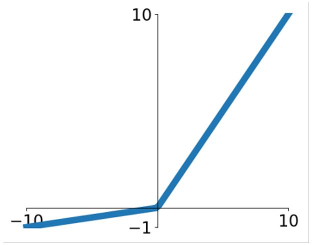
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

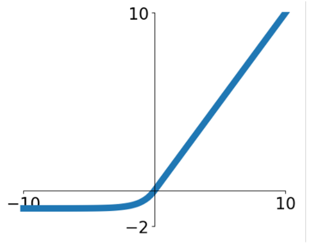


Maxout

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

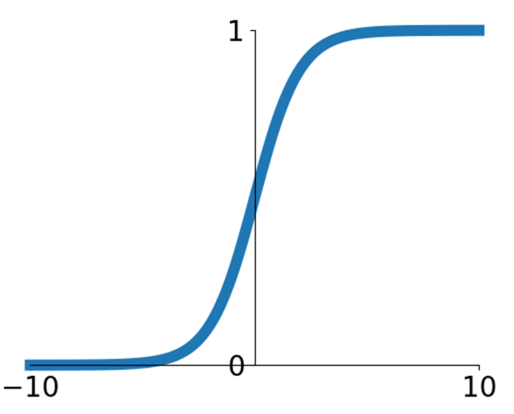
ELU

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



Activation Functions

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

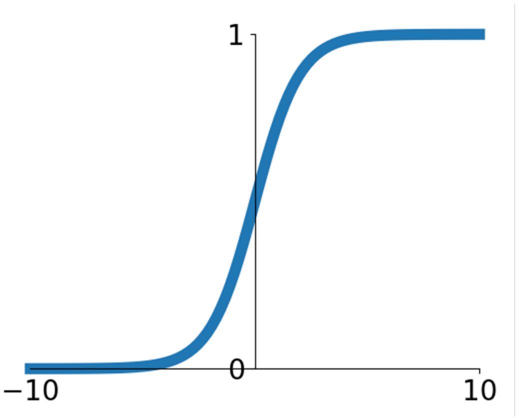


Sigmoid

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

Activation Functions

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

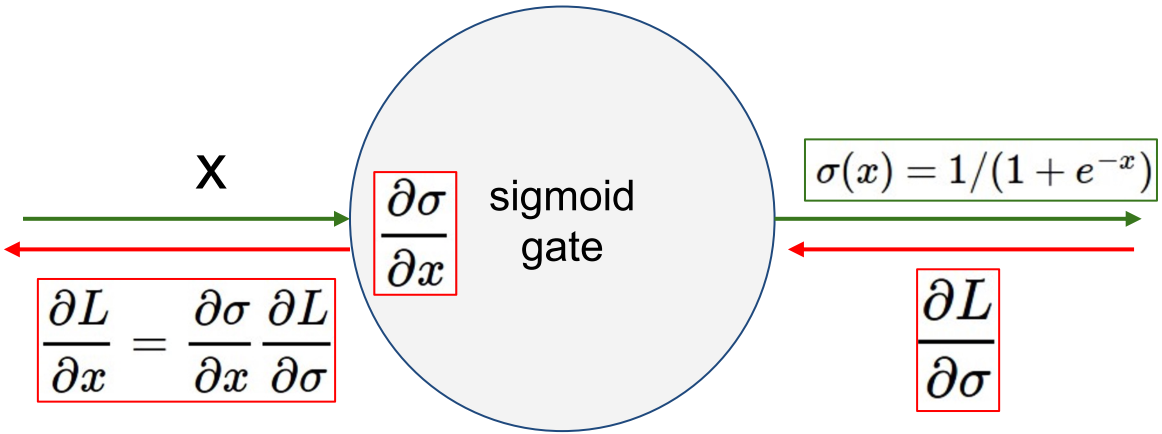


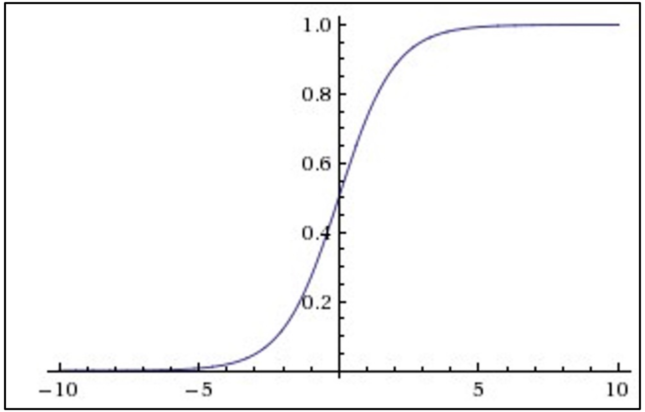
Sigmoid

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

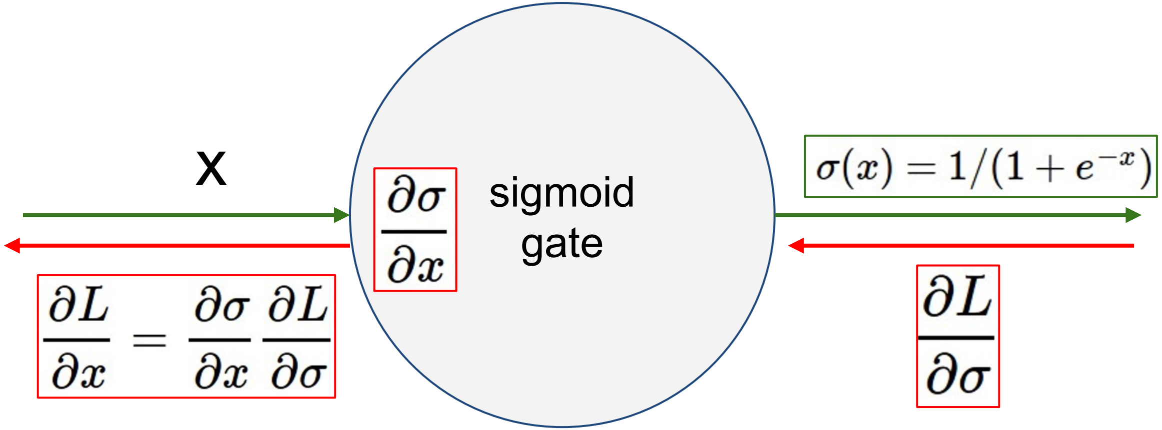
3 problems:

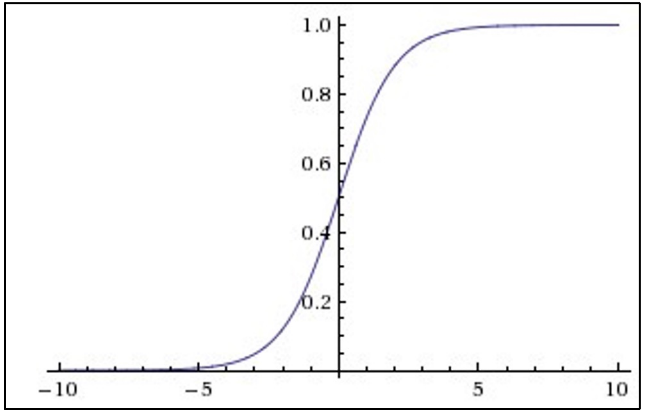
1. Saturated neurons “kill” the gradients





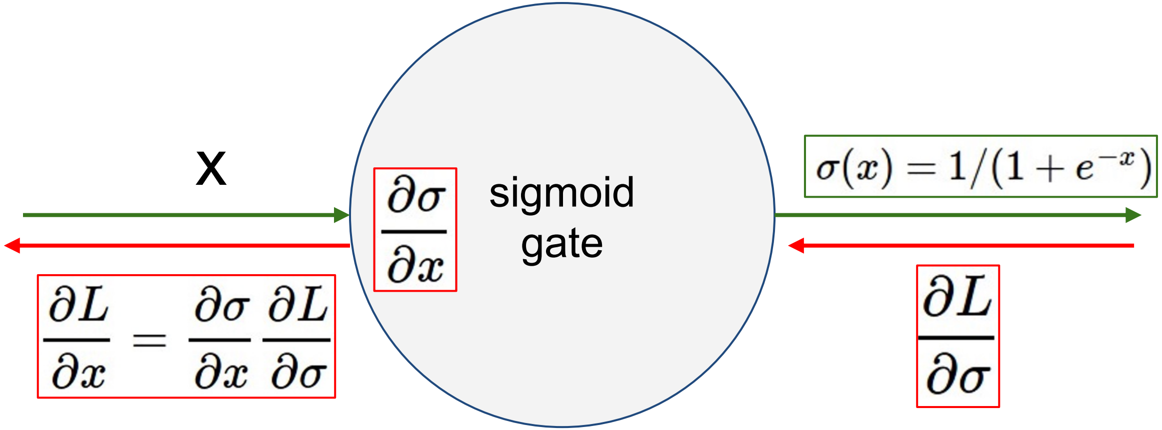
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x))$$

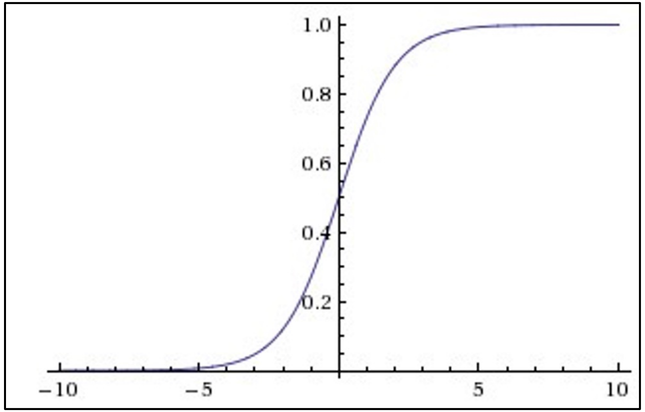




What happens when x = -10?

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



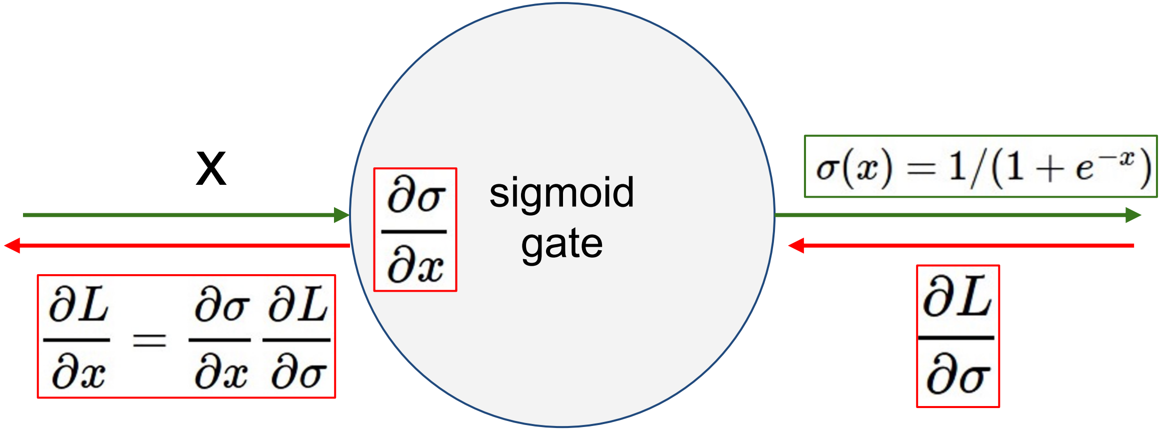


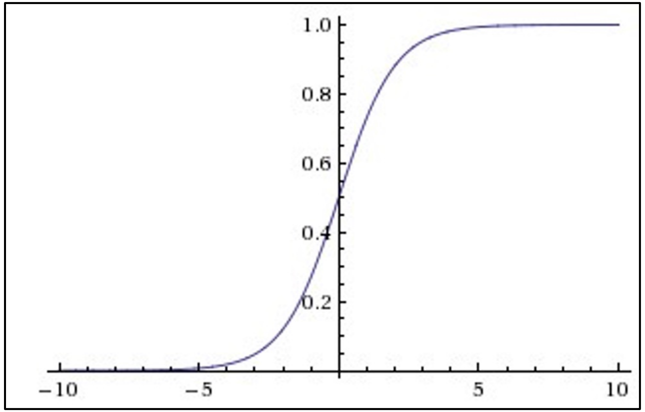
What happens when x = -10?

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

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

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



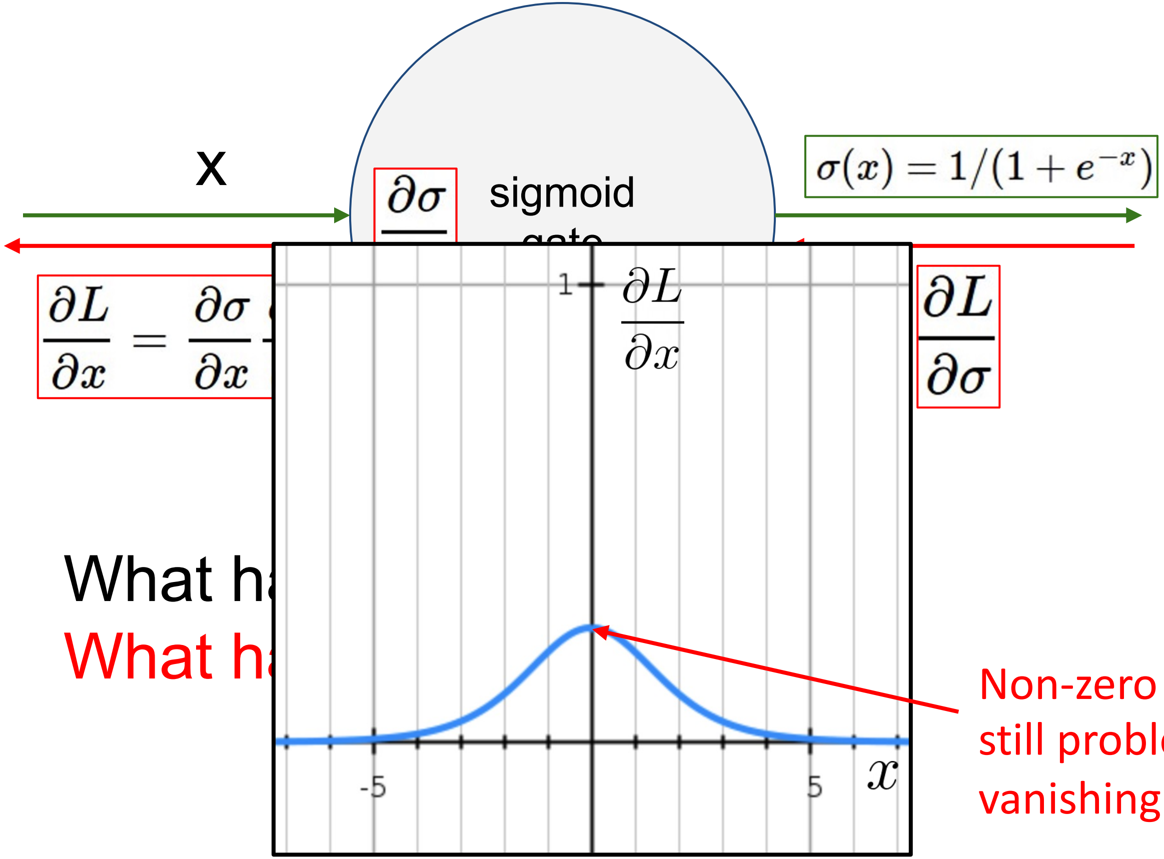


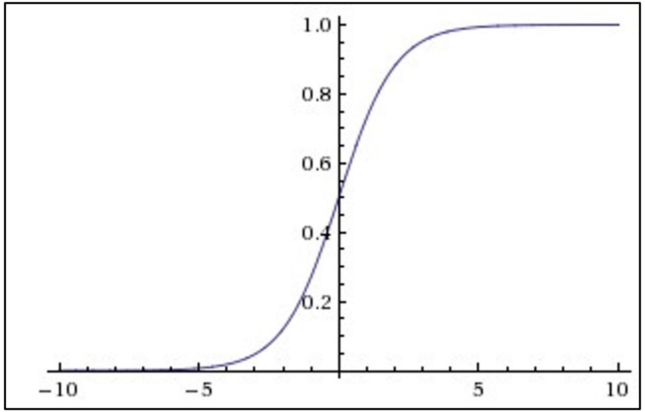
What happens when x = -10?

What happens when x = 10?

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

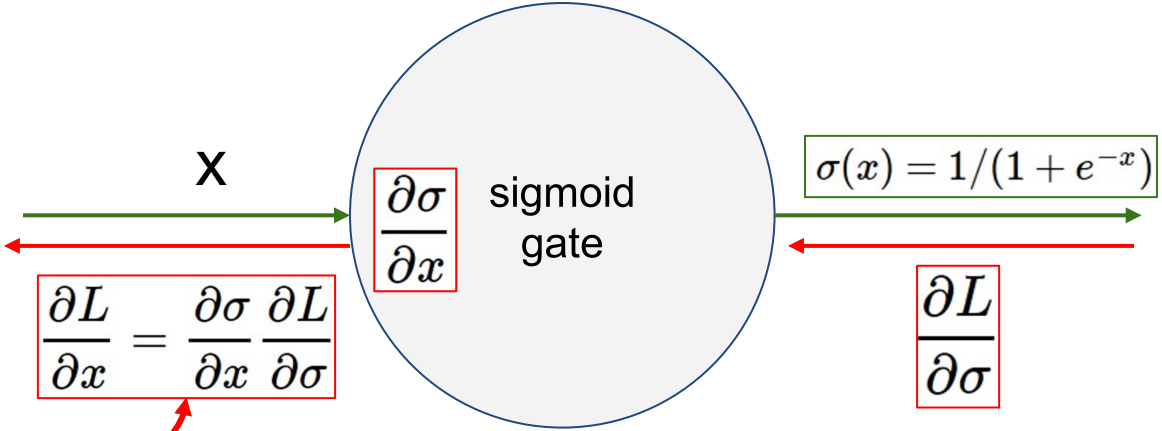
$$\sigma(x) \;= \sim 1 \qquad \frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)\,) = 1(1-1) \;=\; 0$$

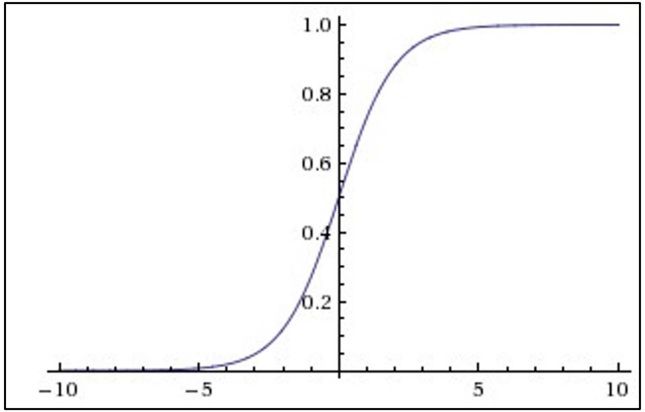




$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)\,)$$

Non-zero but small:
still problematic, causes
vanishing gradient





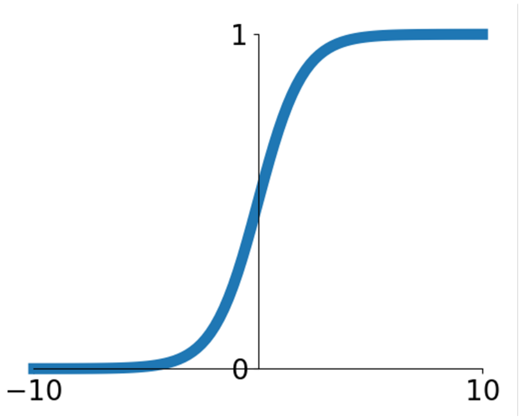
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x))$$

Why is this a problem?

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

Activation Functions

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



Sigmoid

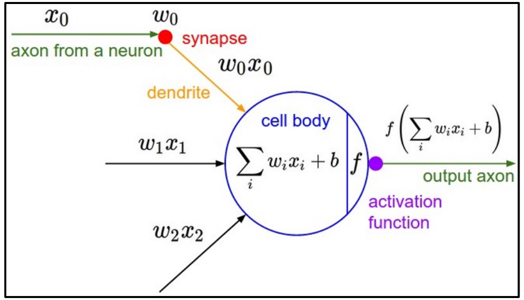
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients
2. **Sigmoid outputs are not zero-centered**

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

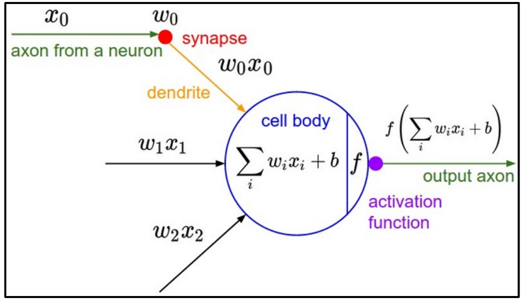
$$f\left(\sum_i w_i x_i + b\right)$$



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

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

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

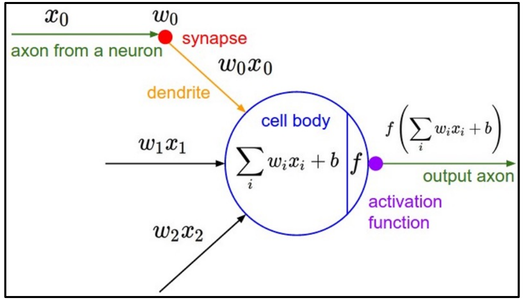


What can we say about the gradients on **w**?

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

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

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



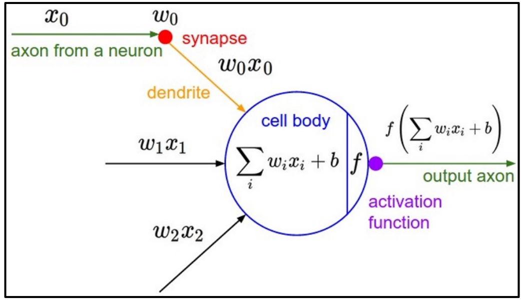
What can we say about the gradients on **w**?

We know that local gradient of sigmoid is always positive

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

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

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



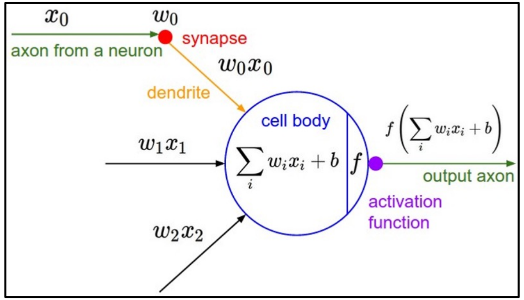
What can we say about the gradients on **w**?

We know that local gradient of sigmoid is always positive
We are assuming x is always positive

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

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**?

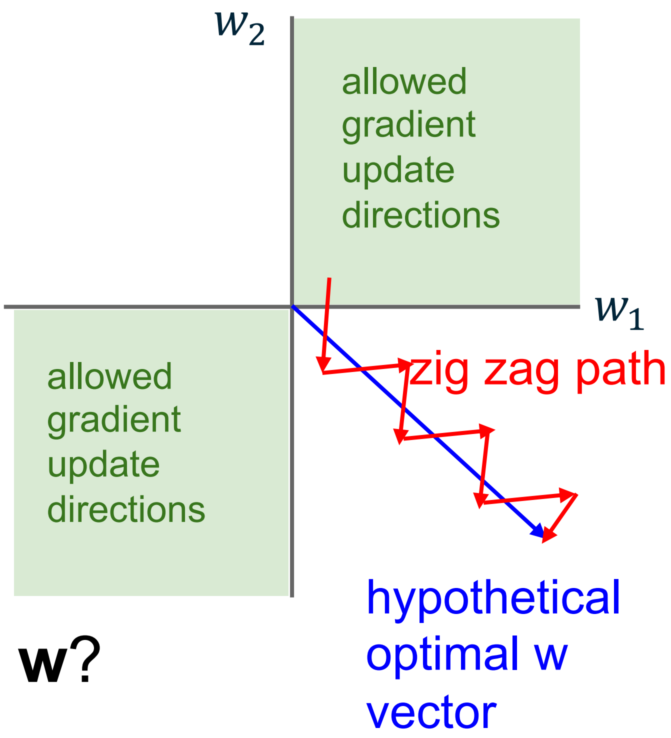
We know that local gradient of sigmoid is always positive
We are assuming x is always positive

So!! Sign of gradient **for all w_i** is the same as the sign of upstream **scalar** gradient!
(local gradient cannot change the sign of global gradient)

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

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

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

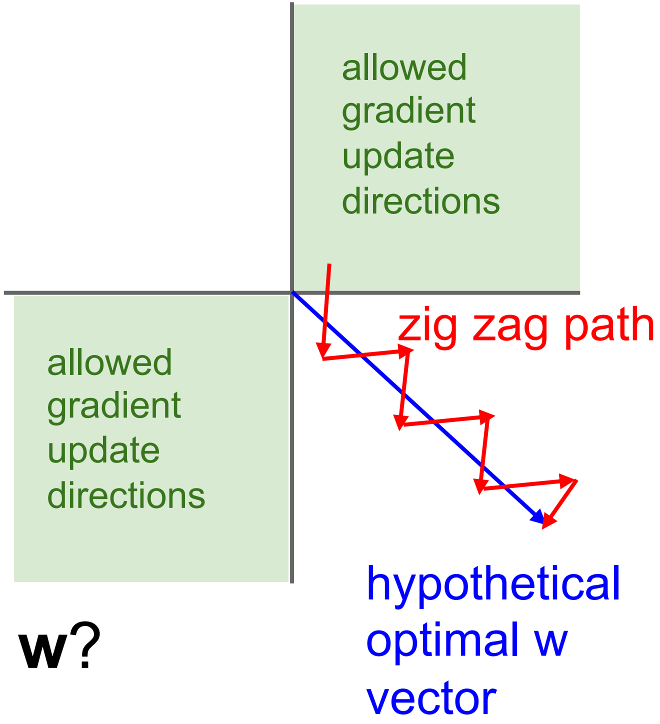


What can we say about the gradients on **w**?

Always all positive or all negative :(

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

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



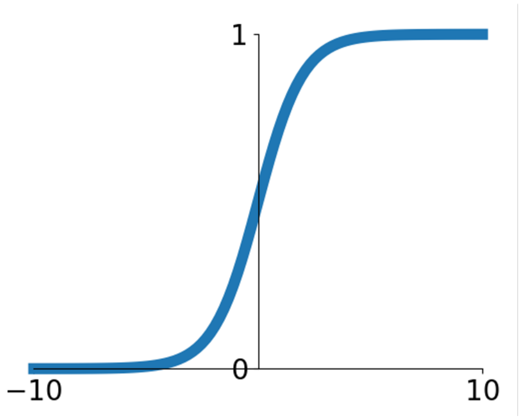
What can we say about the gradients on **w**?

Always all positive or all negative :(

(Minibatches help to average out the gradient, but still not great)

Activation Functions

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



Sigmoid

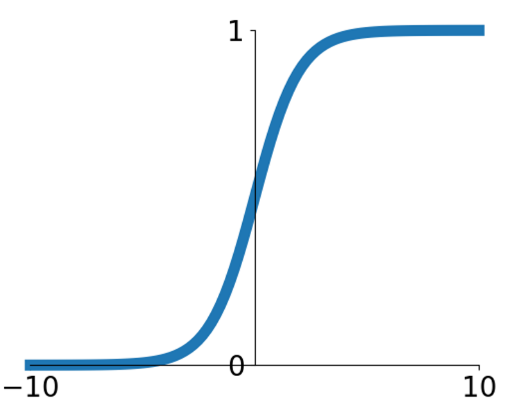
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

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

Activation Functions

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

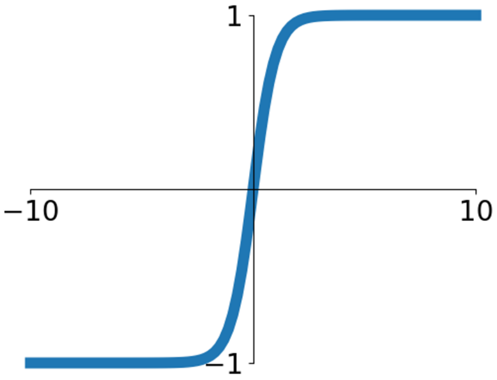


Sigmoid

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

Worst problem in practice: Saturated neurons “kill” the gradients / vanishing gradient

Activation Functions

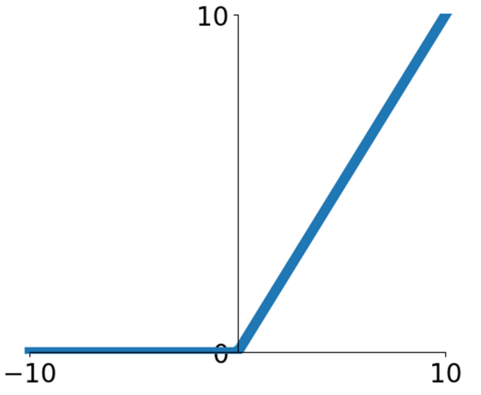


tanh(x)

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

[LeCun et al., 1991]

Activation Functions



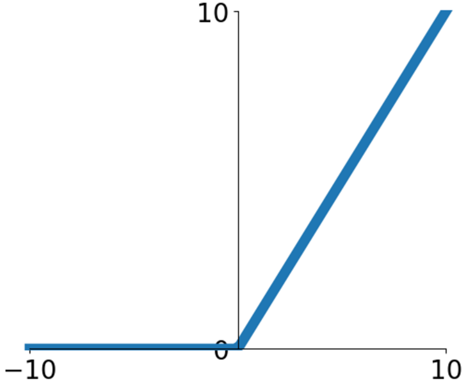
ReLU
(Rectified Linear Unit)

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

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

[Krizhevsky et al., 2012]

Activation Functions



ReLU

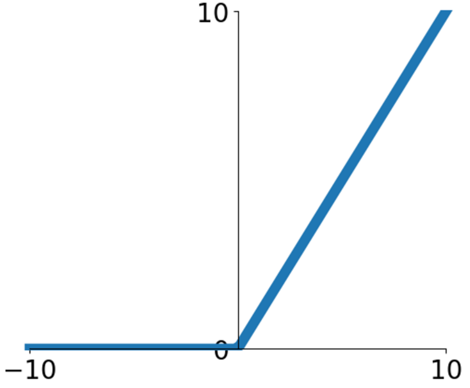
(Rectified Linear Unit)

- 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)

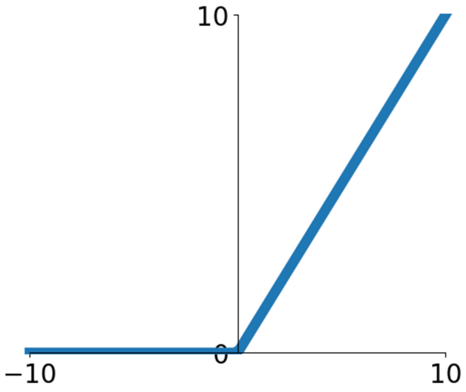
- Computes **f(x) = max(0,x)**

- 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 $x < 0$?

Activation Functions



ReLU
(Rectified Linear Unit)

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

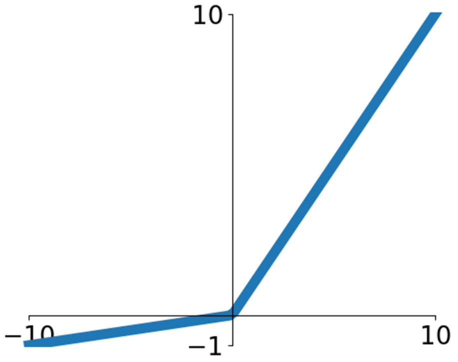
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?
Always 0, A.K.A. “dead ReLU”

Activation Functions

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



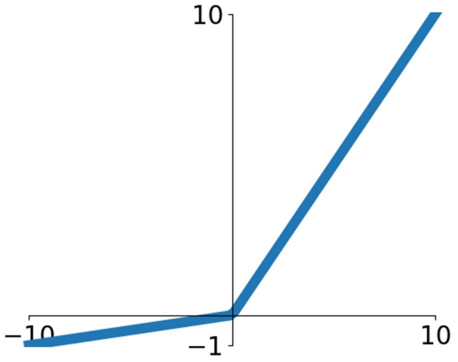
Leaky ReLU

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

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

Activation Functions

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



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

Leaky ReLU

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

Parametric Rectifier (PReLU)

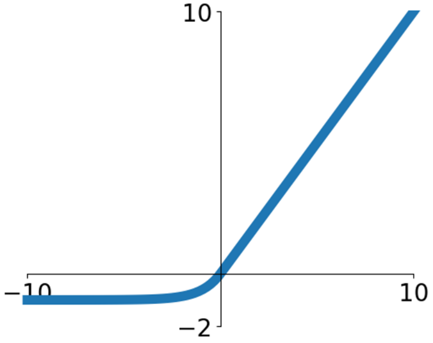
$$f(x) = \max(\alpha x, x)$$

backprop into \alpha
(parameter)

Activation Functions

[Clevert et al., 2015]

Exponential Linear Units (ELU)



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

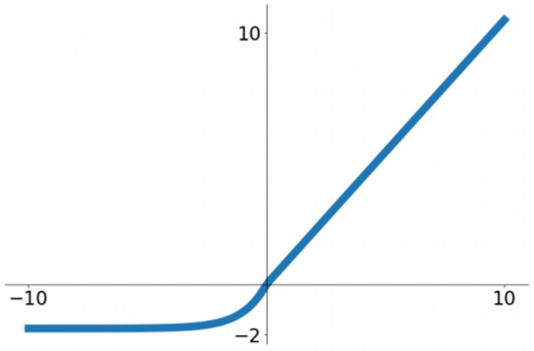
(Alpha default = 1)

- All benefits of ReLU
- Negative saturation encodes presence of features (all goes to -\alpha), not magnitude
- Same in backprop
- Compared with Leaky ReLU: more robust to noise

Activation Functions

[Klambauer et al. ICLR 2017]

Scaled Exponential Linear Units (SELU)



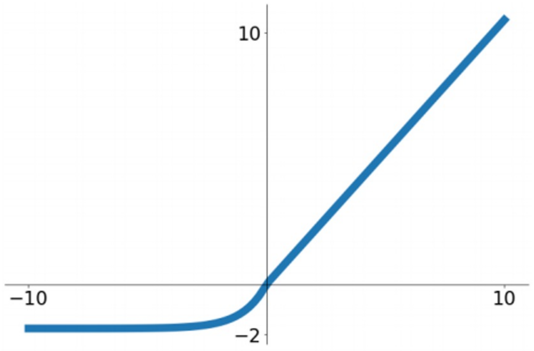
- Scaled version of ELU that works better for deep networks
- "Self-normalizing" property

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

Activation Functions

[Klambauer et al. ICLR 2017]

Scaled Exponential Linear Units (SELU)



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

α = 1.6732632423543772848170429916717
λ = 1.0507009873554804934193349852946

- Scaled versionof ELU that works better for deep networks
- "Self-normalizing" property;
- Can train deep SELU networks without BatchNorm
  - (will discuss more later)

Derivation takes 91 pages of math in appendix…
(Klambauer et al, Self-Normalizing Neural Networks, ICLR 2017)

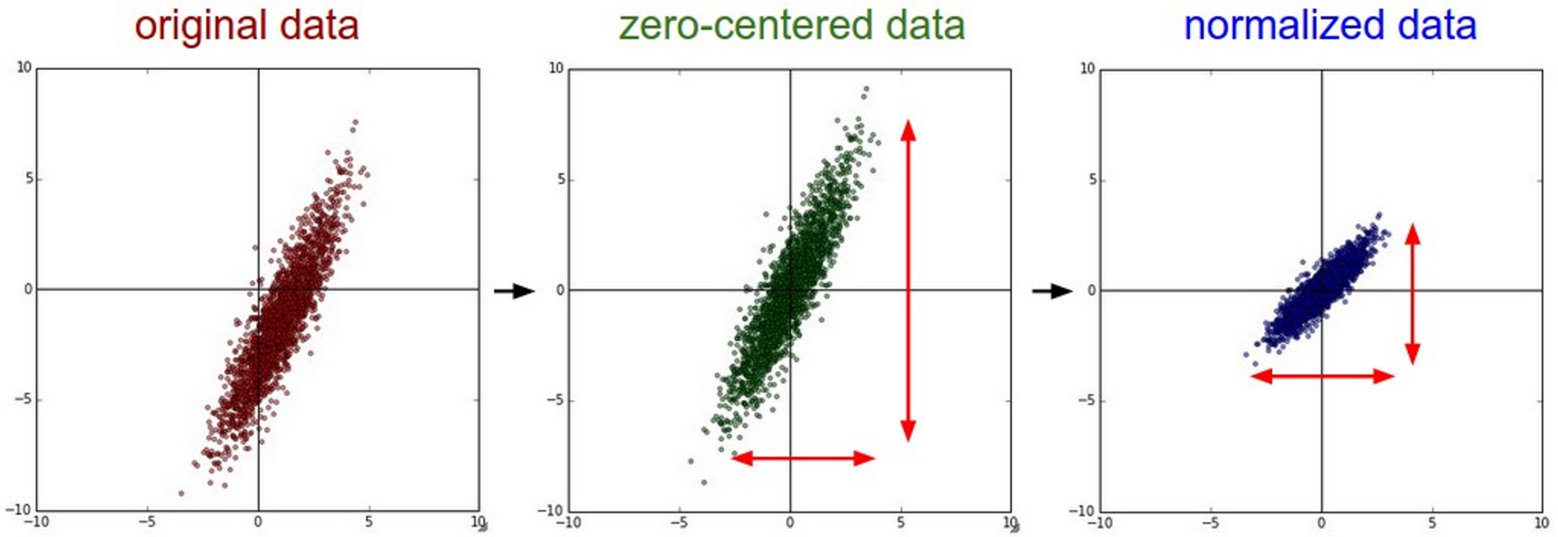
TLDR: In practice:

- Many possible choices beyond what we've talked here, but …

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

Data Preprocessing

Data Preprocessing



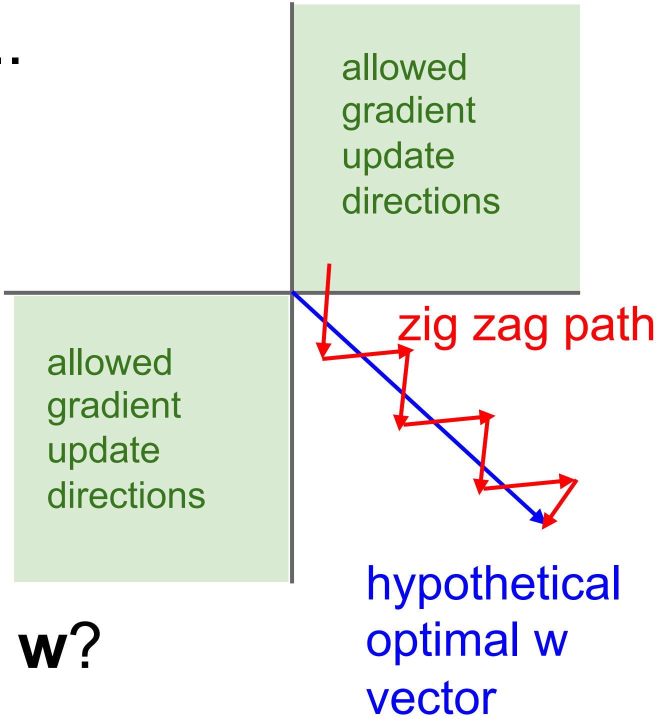

```
X -= np.mean(X, axis = 0)
```

```
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...

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

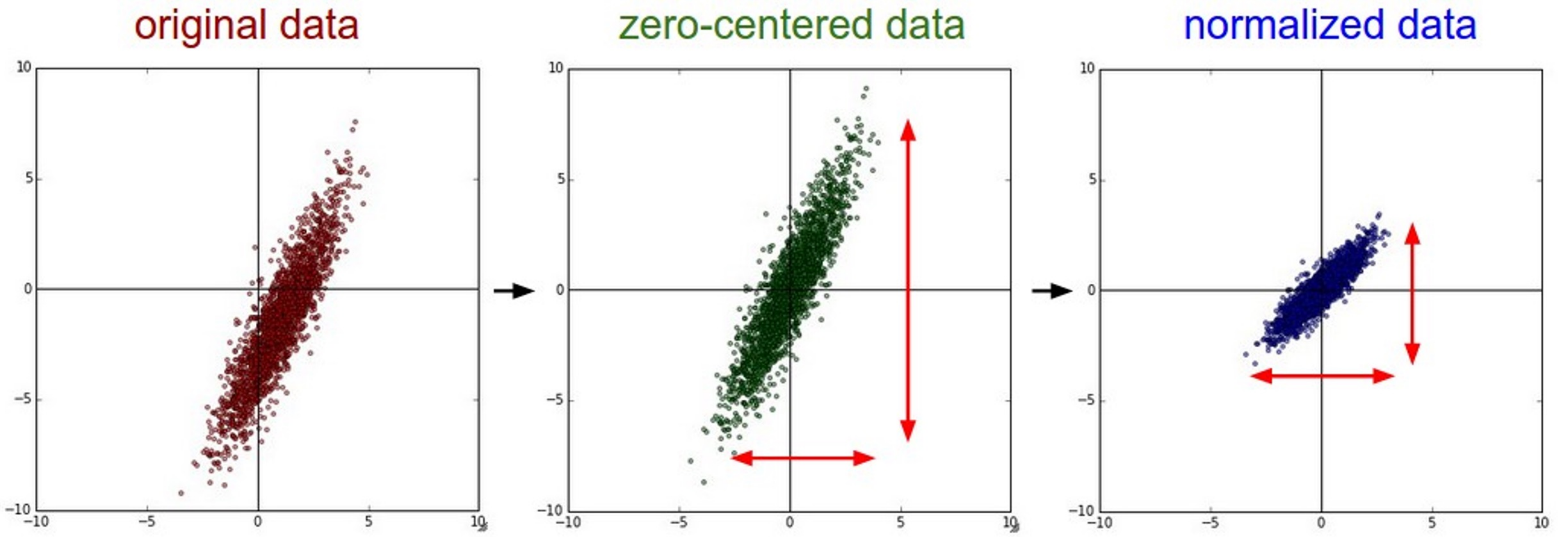


What can we say about the gradients on **w**?

Always all positive or all negative :(

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

Data Preprocessing



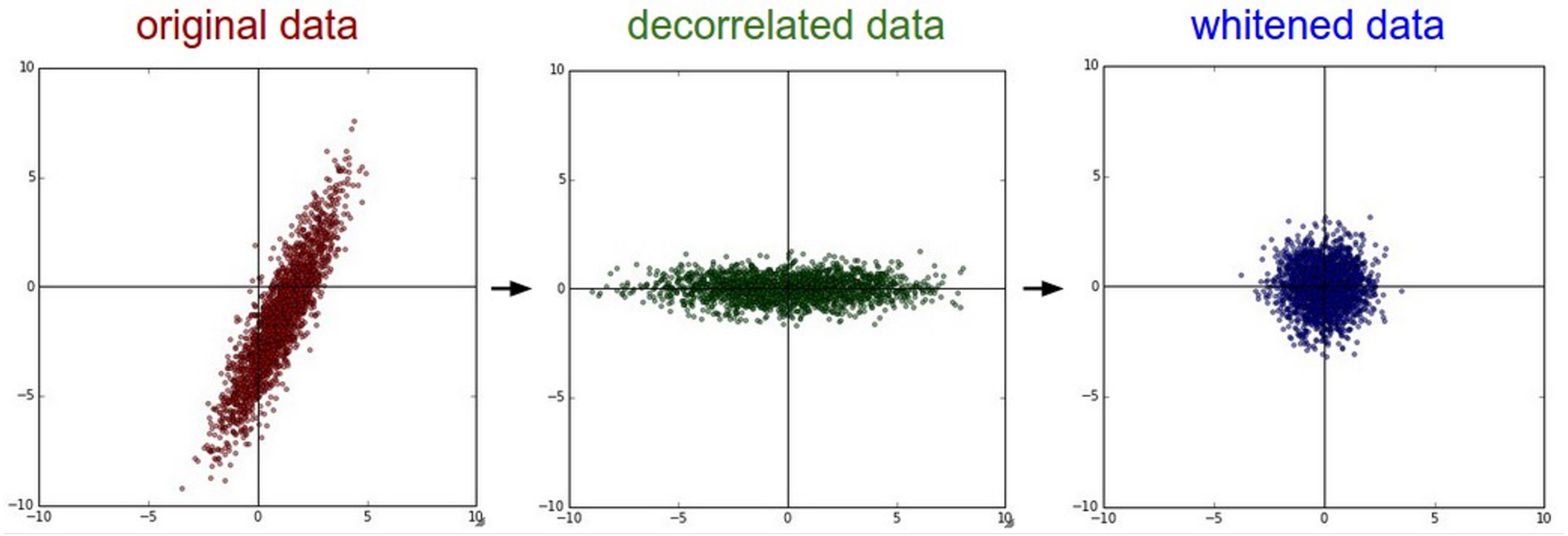

```
X -= np.mean(X, axis = 0)
```

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

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

Data Preprocessing

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

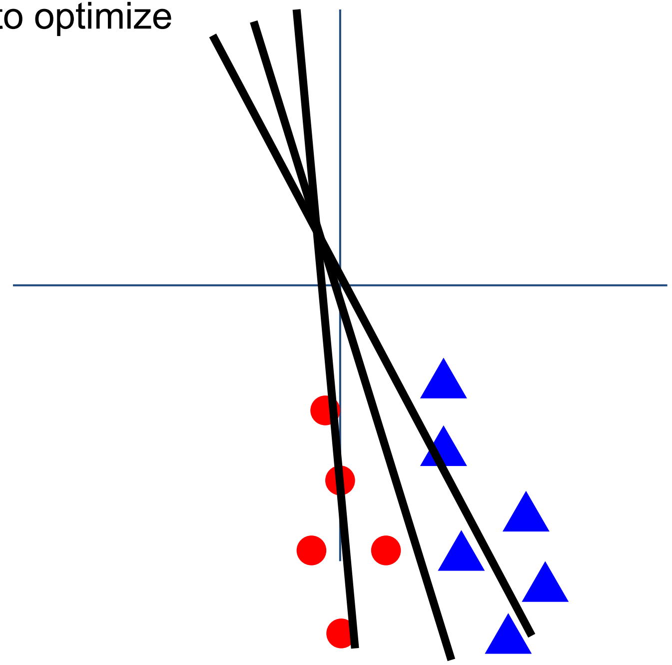


(data has diagonal covariance matrix)

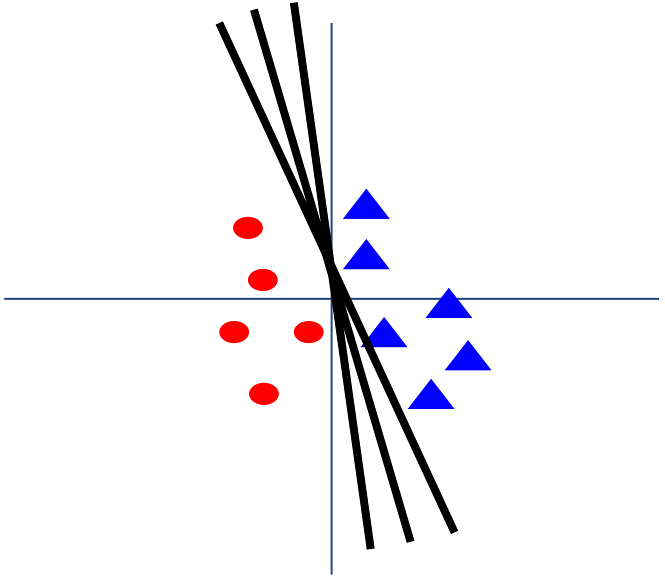
(covariance matrix is the identity matrix)

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



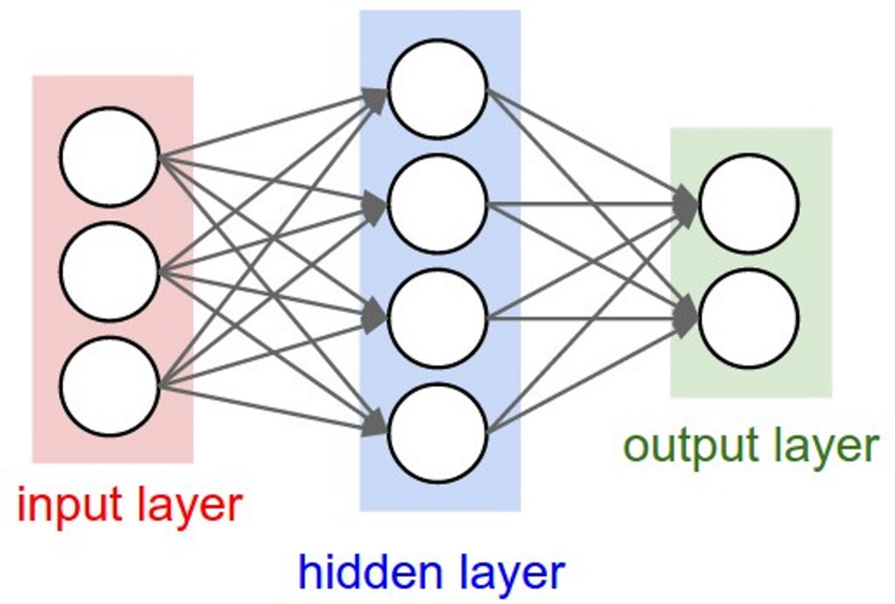
TLDR: In practice for Images: center only

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

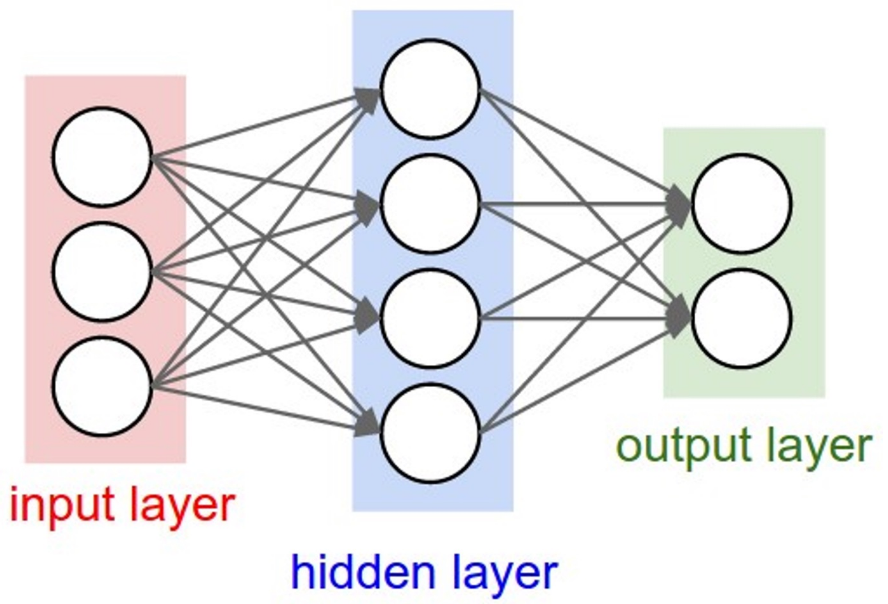
- Subtract the per-pixel mean(e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers,)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

Weight Initialization

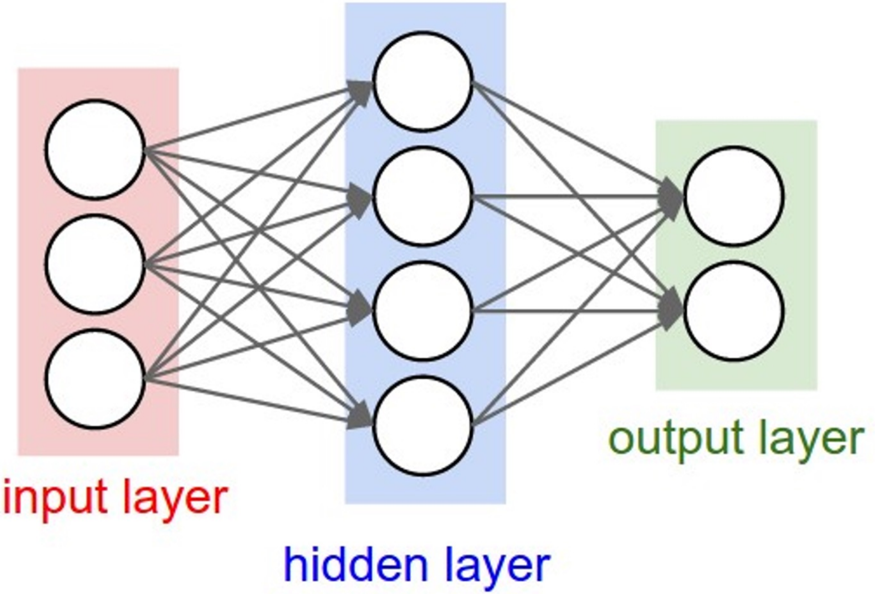
- Q: what happens when W=same initial value is used?



- Q: what happens when W=same initial value is used?
- A: All output will be the same! $w_1^T x = w_2^T x$ if $w_1 = w_2$



- Q: what happens when W=same initial value is used?
- A: All output will be the same! $w_1^T x = w_2^T x$ if $w_1 = w_2$
- Want to **maintain variance** through the layers.



- 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
hs = []
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)
```

Forward pass for a 6-layer net with hidden size 4096

What will happen to the activations for the last layer?

Weight Initialization: Activation statistics

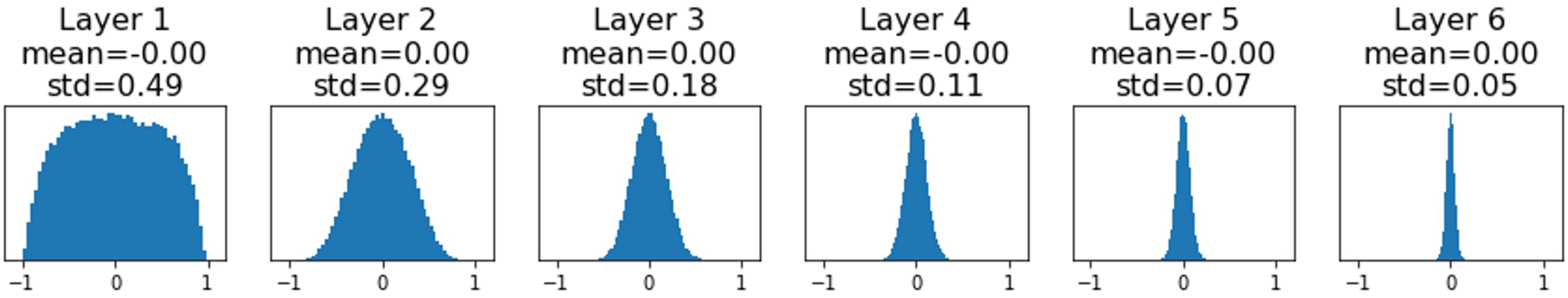
```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

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

Hint: $\frac{\partial L}{\partial w} = x^T \left(\frac{\partial L}{\partial y} \right)$



Weight Initialization: Activation statistics

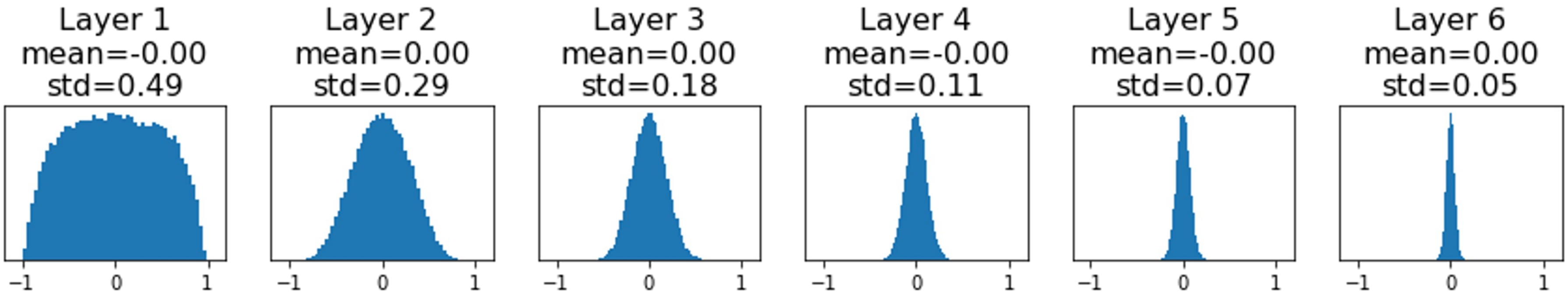
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

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
hs = []
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)
```

Increase std of initial weights from 0.01 to 0.05

Initialize with higher values

What will happen to the activations for the last layer?

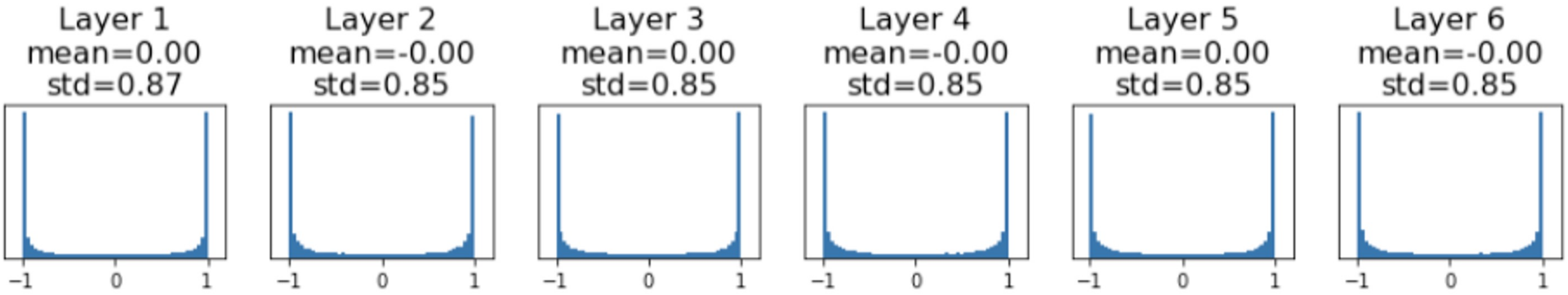
Weight Initialization: Activation statistics

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

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?



Weight Initialization: Activation statistics

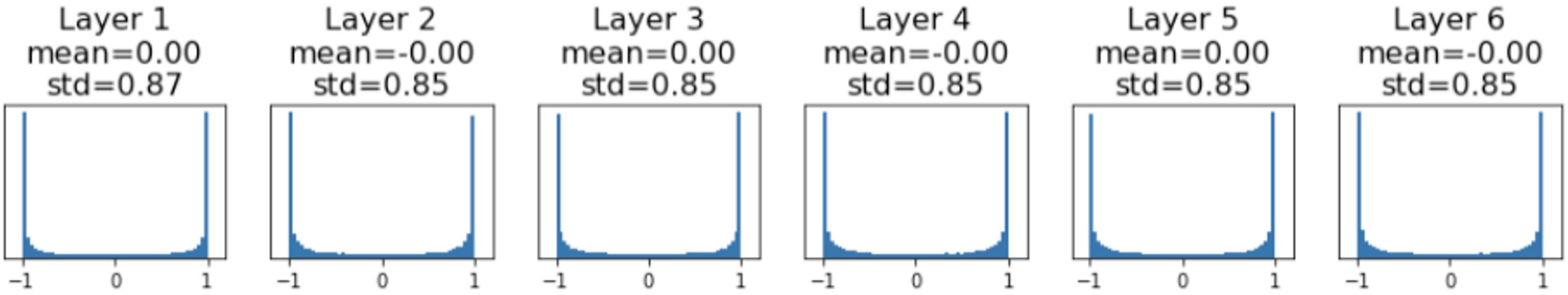
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(



Weight Initialization: Activation statistics

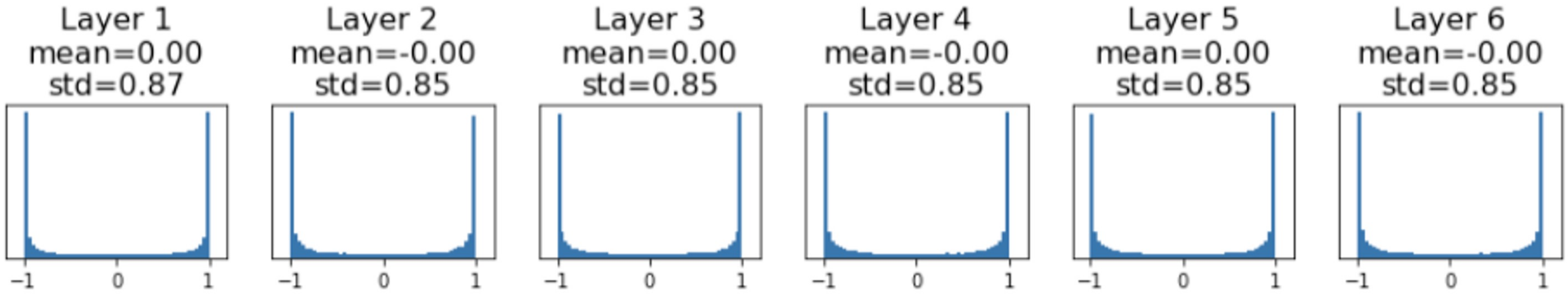
```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

More generally, gradient explosion.



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(Din)

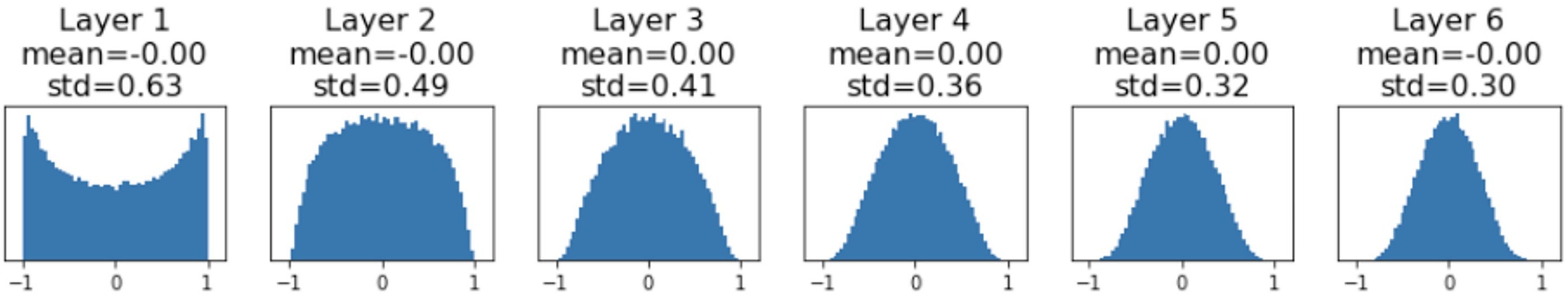
Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

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(Din)

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



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

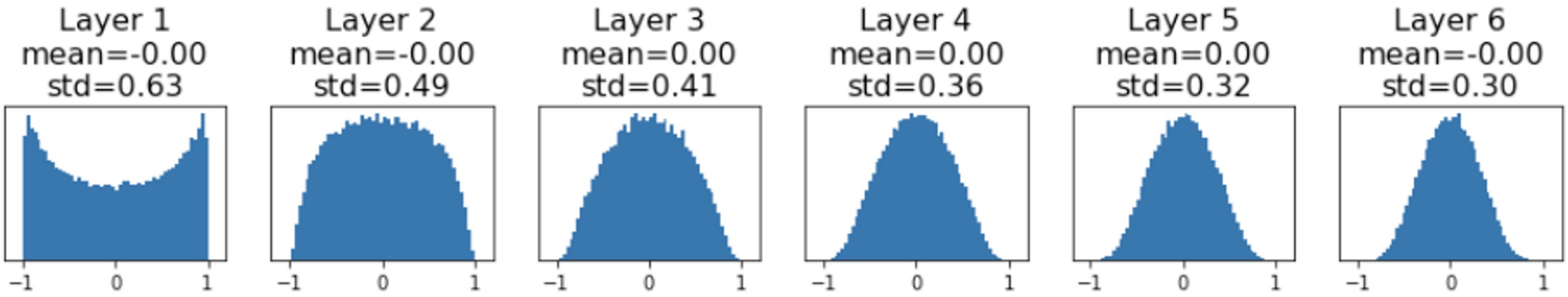
Weight Initialization: “Xavier” Initialization

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

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

For conv layers, Din is filter_size^2 * input_channels



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

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(Din)

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

For conv layers, Din is $\text{filter\_size}^2 * \text{input\_channels}$

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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

Weight Initialization: “Xavier” Initialization

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

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

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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

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

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(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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

We want: $Var(y) = Var(x_i)$

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

Weight Initialization: "Xavier" Initialization

```python
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(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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

We want: $Var(y) = Var(x_i)$

$Var(y) = Var(x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din})$
[substituting value of y]

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

Weight Initialization: “Xavier” Initialization

```python
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(Din)

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

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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

We want: $Var(y) = Var(x_i)$

$Var(y) = Var(x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din})$

$= \Sigma\, Var(x_iw_i) = Din\, Var(x_iw_i)$

[Assume all x_i, w_i are iid] $\sigma^2_{X+Y} = \sigma^2_X + \sigma^2_Y$

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

Weight Initialization: "Xavier" Initialization

```python
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(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is $\text{filter\_size}^2 * \text{input\_channels}$

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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

We want: $Var(y) = Var(x_i)$

$Var(y) = Var(x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din})$
$= Din\ Var(x_iw_i)$
$= Din\ Var(x_i)\ Var(w_i)$
[Assume all x_i, w_i are zero mean]

$$\mathrm{Var}(XY) = E(X^2Y^2) - (E(XY))^2 = \mathrm{Var}(X)\mathrm{Var}(Y) + \cancel{\mathrm{Var}(X)(E(Y))^2} + \cancel{\mathrm{Var}(Y)(E(X))^2}$$

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

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(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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

We want: $Var(y) = Var(x_i)$

$$Var(y) = Var(x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din})$$
$$= Din\ Var(x_iw_i)$$
$$= Din\ Var(x_i)\ Var(w_i)$$

[Assume all x_i, w_i are iid]

So, $Var(y) = Var(x_i)$ only when $Var(w_i) = 1/Din$

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

Weight Initialization: What about ReLU?

```
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.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

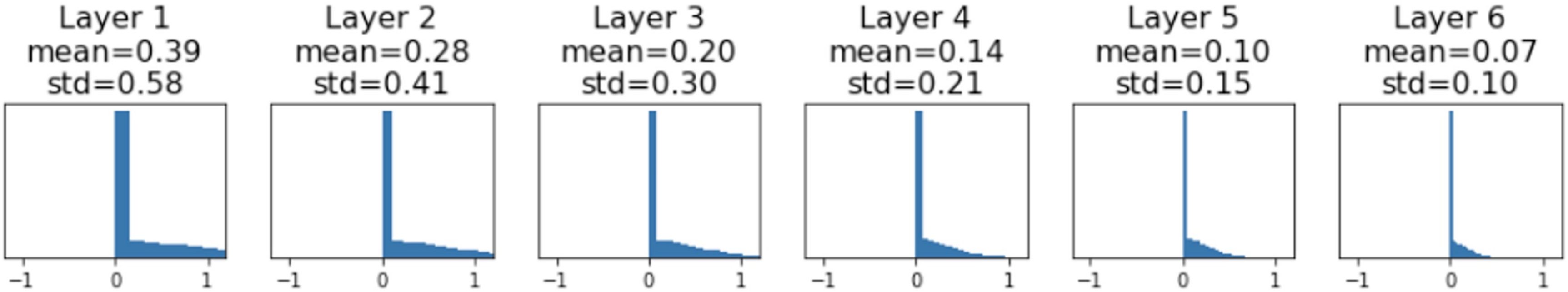
Weight Initialization: What about ReLU?

```python
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.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(



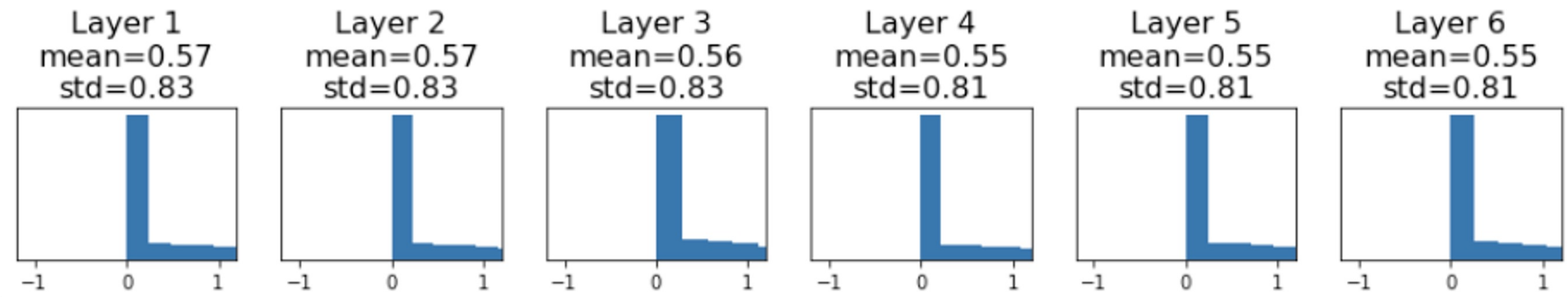
Weight Initialization: Kaiming / MSRA Initialization

```
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(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

Issue: Half of the activation get killed.

Solution: make the non-zero output variance twice as large as input



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

Summary

Training Deep Neural Networks

- Details of the non-linear activation functions
  - Sigmoid, Tanh, ReLU, LeakyRELU, ELU, SELU
- Data normalization
  - Zero-centering, decorrelation, image normalization
- Weight Initialization
  - Constant init, random init, Xavier Init, Kaiming Init

Next time:

Training Deep Neural Networks

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
- Model Ensemble