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

- Lin et al
- Sanchez & Perronnin
- Krizhevsky et al (AlexNet)
- Zeiler & Fergus
- Simonyan & Zisserman (VGG)
- Szegedy et al (GoogLeNet)
- He et al (ResNet)
- Shao et al
- Hu et al (SENet)
- Russakovsky et al

**Year**

- 2010: 28.2
- 2011: 25.8
- 2012: 16.4
- 2013: 11.7
- 2014: 7.3
- 2014: 6.7
- 2015: 3.6
- 2016: 3
- 2017: 2.3
- Human: 5.1

**Layers**

- Shallow: 8 layers
- 19 layers
- 22 layers
- 152 layers
- 152 layers
- 152 layers
Comparing complexity...


Comparing complexity...

Inception-v4: Resnet + Inception!


Comparing complexity...

VGG: most parameters, most operations


Comparing complexity...


Comparing complexity...

**AlexNet:**
Smaller compute, still memory heavy, lower accuracy


Comparing complexity...

**ResNet:**
Moderate efficiency depending on model, highest accuracy


ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- Lin et al (2010, 28.2)
- Sanchez & Perronnin (2011, 25.8)
- Krizhevsky et al (AlexNet, 2012, 16.4)
- Zeiler & Fergus (2013, 11.7)
- Simonyan & Zisserman (VGG, 2014, 7.3)
- Szegedy et al (GoogLeNet, 2014, 6.7)
- He et al (ResNet, 2015, 3.6)
- Shao et al (2016, 3)
- Hu et al (SENet, 2017, 2.3)
- Russakovsky et al (Human, 5.1)

Network ensembling

- 152 layers (2015, 3.6)
- 152 layers (2016, 3)
- 152 layers (2017, 2.3)
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

<table>
<thead>
<tr>
<th></th>
<th>Inception-v3</th>
<th>Inception-v4</th>
<th>Inception-Resnet-v2</th>
<th>Resnet-200</th>
<th>Wrn-68-3</th>
<th>Fusion (Val.)</th>
<th>Fusion (Test)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err. (%)</td>
<td>4.20</td>
<td>4.01</td>
<td>3.52</td>
<td>4.26</td>
<td>4.65</td>
<td>2.92 (-0.6)</td>
<td>2.99</td>
</tr>
</tbody>
</table>
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- Lin et al
- Sanchez & Perronnin
- Krizhevsky et al (AlexNet)
- Zeiler & Fergus
- Simonyan & Zisserman (VGG)
- Szegedy et al (GoogLeNet)
- He et al (ResNet)
- Russakovsky et al
- Hu et al (SENet)

Adaptive feature map reweighting

- 8 layers
- 8 layers
- 19 layers
- 22 layers
- 152 layers
- 152 layers
- 152 layers
- 152 layers

shallow
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

- **2010**: Lin et al., shallow, 8 layers
- **2011**: Sanchez & Perronnin, 16.4
- **2012**: Krizhevsky et al. (AlexNet), 11.7
- **2013**: Zeiler & Fergus, 7.3
- **2014**: Simonyan & Zisserman (VGG), 6.7
- **2014**: Szegedy et al. (GoogLeNet), 3.6
- **2015**: He et al. (ResNet), 3
- **2016**: Shao et al., 2.3
- **2017**: Hu et al. (SENet), 5.1
- **2018**: Russakovsky et al., 152 layers
- **2019**: 152 layers
- **2020**: 152 layers
- **2021**: 152 layers
- **Human**: 5.1
Completion of the challenge: Annual ImageNet competition no longer held after 2017 -> now moved to Kaggle.
But research into CNN architectures is still flourishing
Improving ResNets...

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)
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
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

\[
\text{depth: } d = \alpha^\phi \\
\text{width: } w = \beta^\phi \\
\text{resolution: } r = \gamma^\phi \\
\text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \\
\alpha \geq 1, \beta \geq 1, \gamma \geq 1
\]
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**
\[ \text{max}(0, x) \]

**Leaky ReLU**
\[ \text{max}(0.1x, x) \]

**Maxout**
\[ \text{max}(w_1^T x + b_1, w_2^T x + b_2) \]

**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

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]
Activation Functions

Sigmoid

Formula: \[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

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

3 problems:

1. Saturated neurons “kill” the gradients
The image illustrates the sigmoid gate with the following equation:

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

The derivative of the loss function with respect to the input is given by:

\[ \frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x} \cdot \frac{\partial L}{\partial \sigma} \]

And the derivative of the sigmoid function with respect to the input is:

\[ \frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x)) \]
What happens when x = -10?

\[
\frac{\partial \sigma(x)}{\partial x} = \sigma(x) \left(1 - \sigma(x)\right)
\]
What happens when $x = -10$?

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

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x)) = 0(1 - 0) = 0$$
What happens when $x = -10$?
What happens when $x = 10$?

\[ \sigma(x) = \sim 1 \]
\[ \frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x)) = 1(1 - 1) = 0 \]
What happens when $x = -10$?

What happens when $x = 10$?

Non-zero but small: still problematic, causes vanishing gradient
Why is this a problem?
If all the gradients flowing back will be zero and weights will never change
Activation Functions

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

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

3 problems:

1. Saturated neurons “kill” the gradients
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 \( w \)?

\[
\frac{\partial L}{\partial w} = \sigma \left( \sum_i w_i x_i + b \right) \left( 1 - \sigma \left( \sum_i w_i x_i + b \right) \right) x \times \text{upstream\_gradient}
\]
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

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

\[
\frac{\partial L}{\partial w} = \sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))x \times \text{upstream\_gradient}
\]
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...

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

- 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

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]
Activation Functions

Sigmoid

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

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

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

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

$tanh(x)$

[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)

[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**

- 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 $x < 0$?

**ReLU**
(Rectified Linear Unit)
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

- 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

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 \( \alpha \) (parameter)

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

Exponential Linear Units (ELU)

- 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

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

(Alpha default = 1)
Activation Functions

Scaled Exponential Linear Units (SELU)

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

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

[ Klambauer et al. ICLR 2017 ]
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
  - (will discuss more later)

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

\[
\alpha = 1.6732632423543772848170429916717 \\
\lambda = 1.0507009873554804934193349852946
\]

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

(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

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

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

---

**original data**  
(data has diagonal covariance matrix)

**decorrelated data**  
(covariance matrix is the identity matrix)

**whitened 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

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.

![Diagram of a neural network with input, hidden, and output layers](image)
- First idea: **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)

\[ W = 0.01 \times \text{np.random.randn}(\text{Din, Dout}) \]
- First idea: **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)

\[ W = 0.01 \times \text{np.random.randn}(\text{Din}, \text{Dout}) \]

Works ~okay for small networks, but problems with deeper networks.
Weight Initialization: Activation statistics

```python
dims = [4096] * 7  # Forward pass for a 6-layer net with hidden size 4096
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)
```

What will happen to the activations for the last layer?
Weight Initialization: Activation statistics

```python
dims = [4096] * 7  # Forward pass for a 6-layer net with hidden size 4096
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)
```

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

```python
dims = [4096] * 7  # Forward pass for a 6-layer net with hidden size 4096
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)
```

All activations tend to zero for deeper network layers

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

**A**: All zero, no learning =(

![Activation histograms for different layers](image_url)
Weight Initialization: Activation statistics

```python
dims = [4096] * 7  # Increase std of initial weights from 0.01 to 0.05
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)
```

Initialize with higher values
What will happen to the activations for the last layer?
Weight Initialization: Activation statistics

```
dims = [4096] * 7  # Increase std of initial weights from 0.01 to 0.05
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)
```

All activations saturate

Q: What do the gradients look like?
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)
```

All activations saturate

**Q:** What do the gradients look like?

**A:** Local gradients all zero, no learning =((
Weight Initialization: Activation statistics

```
dims = [4096] * 7  # Increase std of initial weights from 0.01 to 0.05
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)
```

All activations saturate

Q: What do the gradients look like?

More generally, gradient explosion.
Weight Initialization: “Xavier” Initialization

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

“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: std = 1/sqrt(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(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_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  # "Xavier" initialization: std = 1/sqrt(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(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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

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

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

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

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

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_size^2 * input_channels

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

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

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

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

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

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  
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)
```

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

For conv layers, Din is \(\text{filter}_\text{size}^2 \times \text{input_channels}\)

Let: \(y = x_1w_1 + x_2w_2 + \ldots + x_{\text{Din}}w_{\text{Din}}\)

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

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

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

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

```
dims = [4096] * 7  # "Xavier" initialization: std = 1/sqrt(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(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_size^2 * input_channels

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

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

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

\[
\text{Var}(y) = \text{Var}(x_1w_1 + x_2w_2 + \ldots + x_{Din}w_{Din}) \\
= \sum \text{Var}(x_iw_i) = \text{Din} \text{Var}(x_iw_i)
\]

[Assume all \( x_i, w_i \) are iid] \( \sigma_{x+y}^2 = \sigma_x^2 + \sigma_y^2 \)

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

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

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

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

**Xavier** initialization:  
\[
\text{std} = \frac{1}{\sqrt{\text{Din}}}
\]

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

For conv layers, Din is \(\text{filter}\_\text{size}^2 \times \text{input\_channels}\)

```
# Assume: Var(x1) = Var(x2) = ... = Var(xDin)

Let: \(y = x_1w_1 + x_2w_2 + ... + x_{\text{Din}}w_{\text{Din}}\)

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

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

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

```python
import numpy as np

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)
```
Change from tanh to 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)
Weight Initialization: What about ReLU?

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(

```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)
```
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)
```

Issue: Half of the activation get killed.
Solution: make the non-zero output variance twice as large as input

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