Topics:

• Training Neural Networks (Part 2)
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

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

- 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
3. \(\exp()\) is a bit compute expensive
Activation Functions

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

Leaky ReLU

\[ f(x) = \max(0.01x, x) \]

Parametric Rectifier (PReLU)

\[ f(x) = \max(\alpha x, x) \]

[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;
- 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)
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

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

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

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

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 $\frac{dL}{dW}$ look like?

**Hint:**

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

Visualize distribution of activations
Weight Initialization: Activation statistics

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

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)

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?
A: All zero, no learning =(  

Visualize distribution of activations
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)
```

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)
```

Q: What do the gradients look like?

All activations saturate

Visualize distribution of activations
Weight Initialization: Activation statistics

```python
dims = [4096] * 6
hs = []
x = np.random.randn(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 =( 

Visualize distribution of activations
Weight Initialization: Activation statistics

In general
- Small weights -> small output -> vanishing gradient in backpropagation.
- Large weights -> large output -> exploding gradient in backpropagation.

How do we initialize the weights “just right”?

```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)
```

Q: What do the gradients look like?

More generally, gradient explosion (high w-> high output -> high gradient).
Weight Initialization: “Xavier” Initialization

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

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

```
dims = [4096] * 7
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

Visualize distribution of activations
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 \(\text{filter_size}^2 \times \text{input_channels}\)

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

Visualize distribution of activations
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)
```

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

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

\[
\text{dims} = [4096] \times 7 \\
\text{hs} = [] \\
x = \text{np.random.randn}(16, \text{dims}[0]) \\
\text{for } \text{Din}, \text{Dout} \text{ in } \text{zip(dims[:-1], dims[1:])}: \\
W = \text{np.random.randn(Din, Dout)} / \text{np.sqrt(Din)} \\
x = \text{np.tanh}(x \cdot \text{dot}(W)) \\
\text{hs.append(x)}
\]

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

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

Let: \(y = x_1 w_1 + x_2 w_2 + \ldots + x_{D_i} w_{D_i}\)

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

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

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

\[
\text{std} = \frac{1}{\sqrt{\text{Din}}}
\]

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)
```

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

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

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

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

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

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

[substituting value of y]

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

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

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

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

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

[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

```
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 \( \text{filter_size}^2 \times \text{input_channels} \)
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: \( y = x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din} \)

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

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

\[
\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din})
= \text{Din} \text{Var}(x_i w_i)
= \text{Din} \text{Var}(x_i) \text{Var}(w_i)
\]

[Assume all \( x_i, w_i \) are zero mean]

\[
\text{Var}(XY) = E(X^2Y^2) - (E(XY))^2 = \text{Var}(X)\text{Var}(Y) + \text{Var}(X)(E(Y))^2 + \text{Var}(Y)(E(X))^2
\]

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

<table>
<thead>
<tr>
<th>dims = [4096] * 7</th>
<th>“Xavier” initialization: std = 1/sqrt(Din)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hs = []</td>
<td></td>
</tr>
<tr>
<td>x = np.random.randn(16, dims[0])</td>
<td></td>
</tr>
<tr>
<td>for Din, Dout in zip(dims[:-1], dims[1:]):</td>
<td></td>
</tr>
<tr>
<td>W = np.random.randn(Din, Dout) / np.sqrt(Din)</td>
<td></td>
</tr>
<tr>
<td>x = np.tanh(x.dot(W))</td>
<td></td>
</tr>
<tr>
<td>hs.append(x)</td>
<td></td>
</tr>
</tbody>
</table>

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

For conv layers, Din is filter_size^2 * input_channels

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

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

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

\[
\text{Var}(y) = \text{Var}(x_1w_1 + x_2w_2 + ... + x_{Din}w_{Din}) \\
= \text{Din} \text{Var}(x_iw_i) \\
= \text{Din} \text{Var}(x_i) \text{Var}(w_i) \\
\]

[Assume all \( x_i, w_i \) are iid]

So, \( \text{Var}(y) = \text{Var}(x_i) \) only when \( \text{Var}(w_i) = 1/\text{Din} \)
**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: \( y = x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din} \)

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

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

\[
\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din}) \\
= \text{Din} \cdot \text{Var}(x_i w_i) \\
= \text{Din} \cdot \text{Var}(x_i) \cdot \text{Var}(w_i) \\
\]

[Assume all \( x_i, w_i \) are iid]

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

Scaling a normal distribution (std=1) to have Var=1/Din -> multiply by \( \sqrt{1/\text{Din}} \)
“Xavier” initialization:

\[ \text{std} = \frac{1}{\sqrt{\text{Din}}} \]

Weight Initialization: “Xavier” Initialization

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

\[ = \text{Din Var}(x_1w_1) \]
\[ = \text{Din Var}(x_i) \text{ Var}(w_i) \]

[Assume all \( x_i, w_i \) are iid]

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

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

In practice, use \( \text{Var} = 2 / (\text{Din} + \text{Dout}) \) to account for both forward and backward pass

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

For conv layers, Din is \( \text{filter_size}^2 \times \text{input_channels} \)
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)
```
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)
```

Visualize distribution of activations
Weight Initialization: Kaiming / MSRA Initialization


Visualize distribution of activations
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
Batch Normalization
Recall: Input Normalization

```
X -= np.mean(X, axis = 0)
X /= np.std(X, axis = 0)
```
Recall: Input Normalization

Problem: Only for input to the first layer. Input for later layers are longer normalized!
But can’t do dataset normalization for intermediate layers! Activation distribution changes as the training progresses.
Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”

consider a **batch of activations** $x$ at some layer. To make each dimension zero-mean unit-variance, apply:

\[
\hat{x} = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x]}}
\]

this is a vanilla differentiable function…
Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”

\[
\hat{x} = \frac{x - \text{E}[x]}{\sqrt{\text{Var}[x]}}
\]
**Batch Normalization**

### Input:

\[ x : N \times D \]

### Equations:

\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}
\]

Per-channel mean, shape is D

\[
\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2
\]

Per-channel var, shape is D

\[
\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}
\]

Normalized x, Shape is N x D

(Prevent div by 0 err)

[Ioffe and Szegedy, 2015]
Batch Normalization

**Input:** \( x : N \times D \)

\[ \mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j} \]  
Per-channel mean, shape is D

\[ \sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 \]  
Per-channel var, shape is D

\[ \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \]  
Normalized x, Shape is N x D

Problem: What if zero-mean, unit variance is too hard of a constraint? E.g., inserting a BN before sigmoid will constrain it to (mostly) linear regime

[ioffe and Szegedy, 2015]
**Batch Normalization**

**Input:** \( x : N \times D \)

\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j} \quad \text{Per-channel mean, shape is D}
\]

\[
\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 \quad \text{Per-channel var, shape is D}
\]

\[
\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \quad \text{Normalized x, Shape is N x D}
\]

**Problem:** What if zero-mean, unit variance is too hard of a constraint? E.g., inserting a BN before sigmoid will constrain it to (mostly) linear regime. Can we learn the normalization parameters?

[ioffe and Szegedy, 2015]
Batch Normalization

Input: \( x : N \times D \)

Learnable scale and shift parameters:

\( \gamma, \beta : \mathbb{R}^D \)

We want to give the model a chance to adjust batchnorm if the default is not optimal. Learning \( \gamma = \sigma \) and \( \beta = \mu \) will recover the identity function!

\[
\begin{align*}
\mu_j &= \frac{1}{N} \sum_{i=1}^{N} x_{i,j} \\
\sigma_j^2 &= \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 \\
\hat{x}_{i,j} &= \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \\
y_{i,j} &= \gamma_j \hat{x}_{i,j} + \beta_j
\end{align*}
\]

Per-channel mean, shape is D

Per-channel var, shape is D

Normalized x, Shape is N x D

Output, Shape is N x D

[Ioffe and Szegedy, 2015]
Batch Normalization: Test-Time

**Input:** \( x : N \times D \)

**Learnable scale and shift parameters:**
\( \gamma, \beta : \mathbb{R}^D \)

**Per-channel mean,** shape is D
\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}
\]

**Per-channel var,** shape is D
\[
\sigma^2_j = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2
\]

**Normalized x,** Shape is N x D
\[
\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma^2_j + \varepsilon}}
\]

**Output,** Shape is N x D
\[
y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j
\]

Estimates depend on minibatch; can’t do this at test-time!
Batch Normalization: Test-Time

**Input:** \( x : N \times D \)

Learnable scale and shift parameters:
\[
\gamma, \beta : \mathbb{R}^D
\]

Activations become fixed after training. Can calculate training set-wide statistics for inference-time normalization.

At training time, do moving average to save compute.

**Per-channel mean,** shape is D
\[
\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}
\]

**Per-channel var,** shape is D
\[
\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2
\]

**Normalized x,** Shape is N x D
\[
\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}
\]

**Output,** Shape is N x D
\[
y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j
\]

Estimates depend on minibatch; can’t do this at test-time!
Batch Normalization: Test-Time

**Input:** \( x : N \times D \)

Learnable scale and shift parameters:
\( \gamma, \beta : \mathbb{R}^D \)

\[
\begin{align*}
\mu_j &= \text{(Moving) average of values seen during training} \\
\sigma_j^2 &= \text{(Moving) average of values seen during training} \\
\hat{x}_{i,j} &= \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \\
y_{i,j} &= \gamma_j \hat{x}_{i,j} + \beta_j
\end{align*}
\]

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer.

**Per-channel mean,**
shape is D

**Per-channel var,**
shape is D

**Normalized x,**
Shape is \( N \times D \)

**Output,**
Shape is \( N \times D \)
Batch Normalization

Q: Should you put batchnorm before or after ReLU?
A: Topic of debate. Original paper says BN->ReLU. Now most commonly ReLU->BN. If BN-> ReLU and zero mean, ReLU kills half of the activations, but in practice makes insignificant differences.

Q: Should you normalize the input (e.g., images) with batchnorm?
A: No, you already have the fixed & correct dataset statistics, no need to do batchnorm.

Q: How many parameters does a batchnorm layer have?
A: Input dimension * 4: beta, gamma, moving average mu, moving average sigma. Only beta and gamma are trainable parameters.
Batch Normalization

- Makes deep networks **much** easier to train!
  - If you are interested in the theory, read [https://arxiv.org/abs/1805.11604](https://arxiv.org/abs/1805.11604)
  - TL;DR: makes optimization landscape smoother
- Allows higher learning rates, faster convergence
- More useful in deeper networks
- Networks become more robust to initialization
- More robust to range of input
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!
- Needs large batch size to calculate accurate stats
Batch Normalization for ConvNets

**Batch Normalization for fully-connected networks**

\[ x: N \times D \]

Normalize

\[ \mu, \sigma: 1 \times D \]

\[ \gamma, \beta: 1 \times D \]

\[ y = \gamma (x - \mu) / \sigma + \beta \]

**Batch Normalization for convolutional networks**

(Spatial Batchnorm, BatchNorm2D)

\[ x: N \times C \times H \times W \]

Normalize

\[ \mu, \sigma: 1 \times C \times 1 \times 1 \]

\[ \gamma, \beta: 1 \times C \times 1 \times 1 \]

\[ y = \gamma (x - \mu) / \sigma + \beta \]
Layer Normalization

**Batch Normalization** for fully-connected networks

\[ x: N \times D \]

- \( \mu, \sigma: 1 \times D \)
- \( \gamma, \beta: 1 \times D \)
- \( y = \gamma (x - \mu) / \sigma + \beta \)


**Layer Normalization** for fully-connected networks

Same behavior at train and test!

\[ x: N \times D \]

- \( \mu, \sigma: N \times 1 \)
- \( \gamma, \beta: 1 \times D \)
- \( y = \gamma (x - \mu) / \sigma + \beta \)

More flexible (can use \( N = 1! \)), works well with sequence models (RNN, Transformers)
Instance Normalization

Batch Normalization for convolutional networks

\[ x: N \times C \times H \times W \]

Normalize \[ \mu, \sigma: 1 \times C \times 1 \times 1 \]
\[ \gamma, \beta: 1 \times C \times 1 \times 1 \]
\[ y = \gamma (x - \mu) / \sigma + \beta \]

Instance Normalization for convolutional networks
Same behavior at train / test!

\[ x: N \times C \times H \times W \]

Normalize \[ \mu, \sigma: N \times C \times 1 \times 1 \]
\[ \gamma, \beta: 1 \times C \times 1 \times 1 \]
\[ y = \gamma (x - \mu) / \sigma + \beta \]

Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017
Comparison of Normalization Layers

Wu and He, “Group Normalization”, ECCV 2018
Group Normalization

Wu and He, “Group Normalization”, ECCV 2018
(Fancier) Optimizers
Optimization

```python
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += -step_size * weights_grad  # perform parameter update
```
Optimization: Problem #1 with SGD

- Stochastic minibatch gives a noisy estimate of the true gradient direction. Very problematic when the batch size is small (e.g., due to compute resource limit).
- Poorly-selected learning rate makes the oscillation worse (overshoot)

Optimization: Problem #2 with SGD

What if the loss function has a local minima or saddle point?
What if the loss function has a local minima or saddle point?

Zero gradient, gradient descent gets stuck
Optimization: Problem #2 with SGD

What if the loss function has a local minima or saddle point?

Saddle points much more common in high dimension

Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014

SGD + Momentum

Intuitions:
• Think of a ball (set of parameters) moving in space (loss landscape), with momentum keeping it going in a direction.
• Individual gradient step may be noisy, the general trend accumulated over a few steps will point to the right direction.
• Momentum can “push” the ball over saddle points or local minima.
SGD + Momentum

Intuitions:
• Think of a ball (set of parameters) moving in space (loss landscape), with momentum keeping it going in a direction.
• Individual gradient step may be noisy, the general trend accumulated over a few steps will point to the right direction.
• Momentum can “push” the ball over saddle points or local minima.
SGD: the simple two line update code

\[ x_{t+1} = x_t - \alpha \nabla f(x_t) \]

```python
while True:
    dx = compute_gradient(x)
    x = learning_rate * dx
```
**SGD + Momentum:**
continue moving in the general direction as the previous iterations

**SGD**

\[ x_{t+1} = x_t - \alpha \nabla f(x_t) \]

**SGD+Momentum**

\[
\begin{align*}
v_{t+1} &= \rho v_t + \nabla f(x_t) \\
x_{t+1} &= x_t - \alpha v_{t+1}
\end{align*}
\]

- Build up “velocity/momentum” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
SGD + Momentum:

continue moving in the general direction as the previous iterations

**SGD**

\[ x_{t+1} = x_t - \alpha \nabla f(x_t) \]

**SGD+Momentum**

\[ v_{t+1} = \rho v_t + \nabla f(x_t) \]
\[ x_{t+1} = x_t - \alpha v_{t+1} \]

- Build up “velocity/momentum” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
SGD + Momentum: alternative equivalent formulation

\[ v_{t+1} = \rho v_t - \alpha \nabla f(x_t) \]
\[ x_{t+1} = x_t + v_{t+1} \]

\[ v_{t+1} = \rho v_t + \nabla f(x_t) \]
\[ x_{t+1} = x_t - \alpha v_{t+1} \]

\[ vx = 0 \]
while True:
    dx = compute_gradient(x)
    vx = rho * vx - learning_rate * dx
    x += vx

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
SGD+Momentum

Momentum update:

Combine gradient at current point with velocity to get step used to update weights

Nesterov, “A method of solving a convex programming problem with convergence rate $O(1/k^2)$”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013
Momentum update:

Combine gradient at current point with velocity to get step used to update weights

Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Nesterov Momentum

“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction
Nesterov Momentum

- SGD
- SGD+Momentum
- Nesterov
Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another? What does gradient descent do? Very slow progress along shallow dimension, jitter along steep direction.

Assume each contour line has the same loss.
Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?
Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:

Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?
Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:


Loss function has high **condition number**: ratio of largest to smallest eigen value \((\lambda_{\text{max}}/\lambda_{\text{min}})\) of the Hessian matrix of a loss function is large
Small condition number in loss Hessian -> circular contour
Large condition number in loss Hessian -> skewed contour
Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?
Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:

Loss function has high **condition number**: ratio of largest to smallest eigen value ($\lambda_{max}/\lambda_{min}$) of the Hessian matrix of a loss function is large
Small condition number in loss Hessian -> circular contour
Large condition number in loss Hessian -> skewed contour

Can we enable SGD to adapt to this skew-ness?

AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

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

“Per-parameter learning rates”
or “adaptive learning rates”

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011
Q: What happens with AdaGrad?
Q: What happens with AdaGrad?

Progress along “steep” directions is damped; progress along “flat” directions is accelerated.

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```
AdaGrad

grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)

Q2: What happens to the step size over long time?
Q2: What happens to the step size over long time?

Decays to zero 😞
RMSProp: “Leaky AdaGrad”

AdaGrad

\[
\text{grad\_squared} = 0 \\
\text{while True:} \\
\quad \text{dx} = \text{compute\_gradient}(x) \\
\quad \text{grad\_squared} += \text{dx} \times \text{dx} \\
\quad x -= \text{learning\_rate} \times \text{dx} / (\text{np.sqrt(grad\_squared)} + 1\text{e-7})
\]

RMSProp

\[
\text{grad\_squared} = 0 \\
\text{while True:} \\
\quad \text{dx} = \text{compute\_gradient}(x) \\
\quad \text{grad\_squared} = \text{decay\_rate} \times \text{grad\_squared} + (1 - \text{decay\_rate}) \times \text{dx} \times \text{dx} \\
\quad x -= \text{learning\_rate} \times \text{dx} / (\text{np.sqrt(grad\_squared)} + 1\text{e-7})
\]

Tieleman and Hinton, 2012
RMSProp

- SGD
- SGD+Momentum
- RMSProp
- AdaGrad (stuck due to decaying lr)
Adam (almost)

```python
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Typical hyperparams: beta1=0.9, beta2=0.999

Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

Momentum
AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Typical hyperparams: beta1=0.9, beta2=0.999

Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Typical hyperparams: $beta_1=0.9$, $beta_2=0.999$

**Momentum**

**AdaGrad / RMSProp**

Q: What happens at first timestep?

Small -> divide by small number -> bad initial step

Adam (full form)

```python
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7)
```

Typical hyperparams: beta1=0.9, beta2=0.999

Bias correction for the fact that first and second moment estimates start at zero

Adam (full form)

Bias correction for the fact that first and second moment estimates start at zero

Adam with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\text{learning\_rate} = 1e-3$ or $5e-4$ is a great starting point for many models!

Adam

- SGD
- SGD+Momentum
- RMSProp
- Adam
Learning rate schedules
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

Q: Which one of these learning rates is best to use?
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter.

Q: Which one of these learning rates is best to use?

A: In reality, all of these are good learning rates.

Need finer adjustment closer to convergence, so we want to reduce learning rate over time to keep making progress.
Learning rate decays over time

Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** 
\[ \alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos\left(\frac{t\pi}{T}\right)\right) \]

- $\alpha_0$: Initial learning rate
- $\alpha_t$: Learning rate at epoch $t$
- $T$: Total number of epochs

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** \[ \alpha_t = \frac{1}{2} \alpha_0 \left( 1 + \cos \left( \frac{t \pi}{T} \right) \right) \]

- \( \alpha_0 \): Initial learning rate
- \( \alpha_t \): Learning rate at epoch \( t \)
- \( T \): Total number of epochs

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Learning Rate Decay

Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** \[ \alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos\left(t\pi/T\right)\right) \]

**Linear:** \[ \alpha_t = \alpha_0 \left(1 - t/T\right) \]

\( \alpha_0 \) : Initial learning rate  
\( \alpha_t \) : Learning rate at epoch \( t \)  
\( T \) : Total number of epochs
Learning Rate Decay

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** \[ \alpha_t = \frac{1}{2} \alpha_0 \left(1 + \cos\left(\frac{t \pi}{T}\right)\right) \]

**Linear:** \[ \alpha_t = \alpha_0 (1 - t/T) \]

**Inverse sqrt:** \[ \alpha_t = \frac{\alpha_0}{\sqrt{t}} \]

\( \alpha_0 \) : Initial learning rate  
\( \alpha_t \) : Learning rate at epoch \( t \)  
\( T \) : Total number of epochs

Vaswani et al, “Attention is all you need”, NIPS 2017
First-Order Optimization
First-Order Optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation
Second-Order Optimization

(1) Use gradient and Hessian to form quadratic approximation
(2) Step to the minima of the approximation
Second-Order Optimization

second-order Taylor expansion:

\[ J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_\theta J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^\top H(\theta - \theta_0) \]

Solving for the critical point we obtain the Newton parameter update:

\[ \theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0) \]

Q: Why is this bad for deep learning?
Second-Order Optimization

second-order Taylor expansion:

\[ J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_\theta J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^\top H(\theta - \theta_0) \]

Solving for the critical point we obtain the Newton parameter update:

\[ \theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0) \]

Hessian has \( O(N^2) \) elements
Inverting takes \( O(N^3) \)
\( N = \text{Millions} \)

Q: Why is this bad for deep learning?
Second-Order Optimization

Quasi-Newton methods (**BGFS** most popular): instead of inverting the Hessian ($O(n^3)$), approximate inverse Hessian with rank 1 updates over time ($O(n^2)$ each).

- **L-BFGS** (Limited memory BFGS): Does not form/store the full inverse Hessian.
L-BFGS

- Usually works very well in full batch, deterministic mode i.e. if you have a single, deterministic $f(x)$ then L-BFGS will probably work very nicely

- Does not transfer very well to mini-batch setting. Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Ba et al, “Distributed second-order optimization using Kronecker-factored approximations”, ICLR 2017
In practice:

- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate

- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule
  - Try cosine schedule, very few hyperparameters!

- If you can afford to do full batch updates (very rare for deep learning applications) then try out **L-BFGS** (and don’t forget to disable all sources of noise)
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