Lecture 10: Training Neural Networks (Part 1)
Reminder: A3

• Due Monday, October 14 (1 week from today!)

• Remember to run the validation script!
Midterm Exam

• Monday, October 21 (two weeks from today!)
• Location: Chrysler 220 (NOT HERE!)
• Format:
  • True / False, Multiple choice, short answer
  • Emphasize concepts – you don’t need to memorize AlexNet!
  • Closed-book
  • You can bring 1 page "cheat sheet" of handwritten notes (standard 8.5” x 11” paper)
• Alternate exam times: Fill out this form: https://forms.gle/uiMpHdg9752p27bd7
  • Conflict with EECS 551
  • SSD accommodations
  • Conference travel for Michigan
Last Time: Hardware and Software

CPU

GPU

TPU

Static Graphs vs Dynamic Graphs

PyTorch vs TensorFlow
Overview

1. **One time setup**
   - Activation functions, data preprocessing, weight initialization, regularization

2. **Training dynamics**
   - Learning rate schedules; large-batch training; hyperparameter optimization

3. **After training**
   - Model ensembles, transfer learning
Overview

1. One time setup
   - Activation functions, data preprocessing, weight initialization, regularization

2. Training dynamics
   - Learning rate schedules; large-batch training; hyperparameter optimization

3. After training
   - Model ensembles, transfer learning

Today

Next time
Activation Functions
Activation Functions

$x_0$ \rightarrow w_0 \rightarrow$ synapse

axon from a neuron

dendrite

$w_0 x_0$

$w_1 x_1$ \rightarrow $\sum_i w_i x_i + b$ \rightarrow f

cell body

activation function

output axon

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

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

**tanh**
\[ \tanh(x) \]

**ReLU**
\[ \max(0, x) \]

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

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

**ELU**
\[ \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases} \]
Activation Functions: Sigmoid

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

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

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

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

3 problems:  

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

\[ f \left( \sum_i \mathbf{w}_i \mathbf{x}_i + b \right) \]

What can we say about the gradients on \( \mathbf{w} \)?

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 :(
(For a single element! Minibatches help)
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
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. \(\text{exp()}\) is a bit compute expensive
Activation Functions: Tanh

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

\[ f(x) = \max(0, x) \]

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

\[ f(x) = \max(0,x) \]

- 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

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

ReLU gate

$\sigma(x) = \max(0, x)$

$\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x} \frac{\partial L}{\partial \sigma}$

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

active ReLU

dead ReLU will never activate => never update
DATA CLOUD

active ReLU

dead ReLU

will never activate

=> never update

=> Sometimes initialize ReLU neurons with slightly positive biases (e.g. 0.01)
Activation Functions: Leaky ReLU

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

Leaky ReLU

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

Maas et al, “Rectifier Nonlinearities Improve Neural Network Acoustic Models”, ICML 2013
Activation Functions: Leaky ReLU

- 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)

Maas et al, “Rectifier Nonlinearities Improve Neural Network Acoustic Models”, ICML 2013

Activation Functions: Exponential Linear Unit (ELU)

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

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

(Default alpha=1)

- Computation requires \exp()
Activation Functions: Scaled Exponential Linear Unit (SELU)

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

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

\[
\alpha = 1.6732632423543772848170429916717 \\
\lambda = 1.05070098735548049341933349852946
\]

Klambauer et al, Self-Normalizing Neural Networks, ICLR 2017
Activation Functions: Scaled Exponential Linear Unit (SELU)

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

Derivation takes 91 pages of math in appendix...

\[ \alpha = 1.673263242354377284170429916717 \]
\[ \lambda = 1.0507009873554804934193349852946 \]
Accuracy on CIFAR10

<table>
<thead>
<tr>
<th>Network</th>
<th>ReLU</th>
<th>Leaky ReLU</th>
<th>Parametric ReLU</th>
<th>Softplus</th>
<th>ELU</th>
<th>SELU</th>
<th>GELU</th>
<th>Swish</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet</td>
<td>93.8</td>
<td>94.2</td>
<td>94.1</td>
<td>94.3</td>
<td>93</td>
<td>94.1</td>
<td>94.3</td>
<td>94.1</td>
</tr>
<tr>
<td></td>
<td>94.6</td>
<td>94.1</td>
<td>94.1</td>
<td>95.3</td>
<td>95.1</td>
<td>94.9</td>
<td>95.6</td>
<td>95.6</td>
</tr>
<tr>
<td>Wide ResNet</td>
<td>94.8</td>
<td>94.7</td>
<td>94.5</td>
<td>95.5</td>
<td>94.7</td>
<td>94.7</td>
<td>94.7</td>
<td>94.7</td>
</tr>
<tr>
<td>DenseNet</td>
<td>94.8</td>
<td>94.7</td>
<td>94.4</td>
<td>94.8</td>
<td>94.8</td>
<td>94.8</td>
<td>94.8</td>
<td>94.8</td>
</tr>
</tbody>
</table>

Ramachandran et al., “Searching for activation functions”, ICLR Workshop 2018
Activation Functions: Summary

- Don’t think too hard. Just use ReLU
- Try out Leaky ReLU / ELU / SELU / GELU if you need to squeeze that last 0.1%
- Don’t use sigmoid or tanh
Data Preprocessing
Data Preprocessing

original data  zero-centered data  normalized data

(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

original data  zero-centered data  normalized data

\[ X -= \text{np.mean}(X, \text{axis} = 0) \]
\[ X /= \text{np.std}(X, \text{axis} = 0) \]

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

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

- **original data**
- **decorrelated data** (data has diagonal covariance matrix)
- **whitened data** (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
Data Preprocessing for Images

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

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)

- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

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

Not common to do PCA or whitening
Weight Initialization
Q: What happens if we initialize all $W=0, b=0$?
Weight Initialization

Q: What happens if we initialize all $W=0$, $b=0$?

A: All outputs are 0, all gradients are the same! No “symmetry breaking”
Weight Initialization

Next idea: \textbf{small random numbers}
(Gaussian with zero mean, std=0.01)

\begin{equation}
W = 0.01 \times \text{np.random.randn}(\text{Din}, \text{Dout})
\end{equation}
Weight Initialization

Next idea: small random numbers
(Gaussian with zero mean, std=0.01)

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

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)
```
**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?
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 😞
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)
```
Weight Initialization: Activation Statistics

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

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

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

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

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

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

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

"Xavier" initialization: std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

\[ y = Wx \]

\[ y_i = \sum_{j=1}^{Din} x_j w_j \]
Weight Initialization: Xavier Initialization

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

**Derivation:** Variance of output = Variance of input

\[ y = Wx \]

\[ y_i = \sum_{j=1}^{\text{Din}} x_j w_j \]

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

[Assume \( x, w \) are iid]
Weight Initialization: Xavier Initialization

"Xavier" initialization: std = 1/sqrt(Din)

**Derivation:** Variance of output = Variance of input

\[ y = Wx \]

\[ y_i = \sum_{j=1}^{Din} x_j w_j \]

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

\[ = \text{Din} \times (E[x_i^2]E[w_i^2] - E[x_i]^2 E[w_i]^2) \]

[Assume x, w are iid]

[Assume x, w independent]
**Weight Initialization: Xavier Initialization**

“Xavier” initialization: \( \text{std} = 1/\sqrt{\text{Din}} \)

**Derivation:** Variance of output = Variance of input

\[
y = Wx \quad y_i = \sum_{j=1}^{\text{Din}} x_j w_j
\]

\[
\text{Var}(y_i) = \text{Din} \times \text{Var}(x_i w_i) \\
= \text{Din} \times (E[x_i^2]E[w_i^2] - E[x_i]^2 E[w_i]^2) \\
= \text{Din} \times \text{Var}(x_i) \times \text{Var}(w_i)
\]

[Assume \( x, w \) are iid]

[Assume \( x, w \) independent]

[Assume \( x, w \) are zero-mean]
Weight Initialization: Xavier Initialization

"Xavier" initialization: std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

\[ y = Wx \]
\[ y_i = \sum_{j=1}^{Din} x_j w_j \]

\[ \text{Var}(y_i) = \text{Din} \times \text{Var}(x_i w_i) \]
\[ = \text{Din} \times (E[x_i^2]E[w_i^2] - E[x_i]^2 E[w_i]^2) \]
\[ = \text{Din} \times \text{Var}(x_i) \times \text{Var}(w_i) \]

[Assume x, w are iid]

[Assume x, w independent]

[Assume x, w are zero-mean]

If \( \text{Var}(w_i) = 1/\text{Din} \) then \( \text{Var}(y_i) = \text{Var}(x_i) \)
Weight Initialization: What about ReLU?

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

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

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(
Weight Initialization: Kaiming / MSRA Initialization

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

”Just right” – activations nicely scaled for all layers

Weight Initialization: Residual Networks

If we initialize with MSRA: then $\text{Var}(F(x)) = \text{Var}(x)$
But then $\text{Var}(F(x) + x) > \text{Var}(x)$ – variance grows with each block!
Weight Initialization: Residual Networks

If we initialize with MSRA: then $\text{Var}(F(x)) = \text{Var}(x)$
But then $\text{Var}(F(x) + x) > \text{Var}(x)$ – variance grows with each block!

**Solution**: Initialize first conv with MSRA, initialize second conv to zero. Then $\text{Var}(x + F(x)) = \text{Var}(x)$

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
Now your model is training ... but it overfits!

Regularization
Regularization: Add term to the loss

\[ L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_{j} - f(x_i; W)_{y_i} + 1) + \lambda R(W) \]

In common use:

**L2 regularization**

\[ R(W) = \sum_{k} \sum_{l} W_{k,l}^2 \quad \text{(Weight decay)} \]

**L1 regularization**

\[ R(W) = \sum_{k} \sum_{l} |W_{k,l}| \]

**Elastic net (L1 + L2)**

\[ R(W) = \sum_{k} \sum_{l} \beta W_{k,l}^2 + |W_{k,l}| \]
Regularization: Dropout

In each forward pass, randomly set some neurons to zero. Probability of dropping is a hyperparameter; 0.5 is common.

Regularization: Dropout

\[ p = 0.5 \] # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p  # first dropout mask
    H1 *= U1  # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p  # second dropout mask
    H2 *= U2  # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
Regularization: Dropout

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

- has an ear
- has a tail
- is furry
- has claws
- mischievous look

X

X

X

X

X

X

cat score
Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...
Want to “average out” the randomness at test-time

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

But this integral seems hard ...
Want to approximate the integral

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

Consider a single neuron.
Want to approximate the integral

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

Consider a single neuron.

At test time we have:

$$E[a] = w_1x + w_2y$$
Dropout: Test Time

Want to approximate the integral

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

Consider a single neuron.

At test time we have:

\[ E[a] = w_1 x + w_2 y \]

During training we have:

\[
\begin{align*}
E[a] &= \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\
& \quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\
& \quad = \frac{1}{2}(w_1 x + w_2 y)
\end{align*}
\]
Dropout: Test Time

Want to approximate the integral

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

Consider a single neuron.

At test time we have:
\[ E[a] = w_1 x + w_2 y \]

During training we have:
\[ E[a] = \frac{1}{4} (w_1 x + w_2 y) + \frac{1}{4} (w_1 x + 0y) + \frac{1}{4} (0x + 0y) + \frac{1}{4} (0x + w_2 y) \]
\[ = \frac{1}{2} (w_1 x + w_2 y) \]

At test time, drop nothing and **multiply** by dropout probability.
At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time
""" Vanilla Dropout: Not recommended implementation (see notes below) """

\[ p = 0.5 \] # probability of keeping a unit active. higher = less dropout

```python
def train_step(X):
    \"\"\" X contains the data \"\"\"

    # forward pass for example 3-layer neural network
    \n    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p  # first dropout mask
    H1 *= U1  # drop!
    \n    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p  # second dropout mask
    H2 *= U2  # drop!
    out = np.dot(W3, H2) + b3
    \n    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    \# ensembled forward pass
    \n    H1 = np.maximum(0, np.dot(W1, X) + b1) * p  # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p  # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

- drop in forward pass
- scale at test time
More common: “Inverted dropout”

\[ p = 0.5 \] # probability of keeping a unit active. higher = less dropout

```python
def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

Drop and scale during training

Test time is unchanged!
Recall AlexNet, VGG have most of their parameters in **fully-connected layers**; usually Dropout is applied there.
Recall AlexNet, VGG have most of their parameters in **fully-connected layers**; usually Dropout is applied there.

Later architectures (GoogLeNet, ResNet, etc) use global average pooling instead of fully-connected layers: they don’t use dropout at all!
Regularization: A common pattern

**Training:** Add some kind of randomness

\[ y = f_W(x, z) \]

**Testing:** Average out randomness (sometimes approximate)

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]
Regularization: A common pattern

**Training:** Add some kind of randomness

\[ y = f_W(x, z) \]

**Testing:** Average out randomness (sometimes approximate)

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

**Example:** Batch Normalization

**Training:** Normalize using stats from random minibatches

**Testing:** Use fixed stats to normalize
Regularization: A common pattern

**Training:** Add some kind of randomness

\[ y = f_W(x, z) \]

For ResNet and later, often L2 and Batch Normalization are the only regularizers!

**Testing:** Average out randomness (sometimes approximate)

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

**Example:** Batch Normalization

**Training:** Normalize using stats from random minibatches

**Testing:** Use fixed stats to normalize
Data Augmentation

Load image and label

“cat”

CNN

Compute loss

This image by Nikita is licensed under CC-BY 2.0
Data Augmentation

Load image and label → "cat" → Transform image → CNN → Compute loss
Data Augmentation: Horizontal Flips
Data Augmentation: Random Crops and Scales

Training: sample random crops / scales

ResNet:
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch
Data Augmentation: Random Crops and Scales

**Training**: sample random crops / scales

ResNet:
1. Pick random $L$ in range $[256, 480]\]
2. Resize training image, short side = $L$
3. Sample random $224 \times 224$ patch

**Testing**: average a fixed set of crops

ResNet:
1. Resize image at 5 scales: $\{224, 256, 384, 480, 640\}$
2. For each size, use $10$ $224 \times 224$ crops: 4 corners + center, + flips
Data Augmentation: Color Jitter

Simple: Randomize contrast and brightness

More Complex:
1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(Used in AlexNet, ResNet, etc)
Data Augmentation: Get creative for your problem!

Random mix/combinations of:
- translation
- rotation
- stretching
- shearing,
- lens distortions, ... (go crazy)
Regularization: A common pattern

**Training:** Add some randomness

**Testing:** Marginalize over randomness

**Examples:**
- Dropout
- Batch Normalization
- Data Augmentation

Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013
Regularization: DropConnect

Training: Drop random connections between neurons (set weight=0)
Testing: Use all the connections

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect

Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013
Regularization: Fractional Pooling

**Training:** Use randomized pooling regions

**Testing:** Average predictions over different samples

**Examples:**
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling

Graham, “Fractional Max Pooling”, arXiv 2014
Regularization: Stochastic Depth

**Training:** Skip some residual blocks in ResNet

**Testing:** Use the whole network

**Examples:**
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth

**Regularization:** Stochastic Depth

**Training:** Set random images regions to 0  
**Testing:** Use the whole image

**Examples:**  
Dropout  
Batch Normalization  
Data Augmentation  
DropConnect  
Fractional Max Pooling  
Stochastic Depth  
Cutout

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

DeVries and Taylor, “Improved Regularization of Convolutional Neural Networks with Cutout”, arXiv 2017
Regularization: Mixup

**Training:** Train on random blends of images

**Testing:** Use original images

**Examples:**
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout
- Mixup

Zhang et al., “mixup: Beyond Empirical Risk Minimization”, ICLR 2018

Sample blend probability from a beta distribution $\text{Beta}(a, b)$ with $a=b\approx0$ so blend weights are close to $0/1$.

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

**Target label:**
- cat: 0.4
- dog: 0.6

CNN
Regularization: Mixup

**Training**: Train on random blends of images

**Testing**: Use original images

**Examples**:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout
- Mixup

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

Target label:
- cat: 0.4
- dog: 0.6

Zhang et al, “mixup: Beyond Empirical Risk Minimization”, ICLR 2018
Regularization: Mixup

Training: Train on random blends of images
Testing: Use original images

Examples:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth
- Cutout
- Mixup

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

Zhang et al, “mixup: Beyond Empirical Risk Minimization”, ICLR 2018
Summary

1. **One time setup**
   Activation functions, data preprocessing, weight initialization, regularization

2. **Training dynamics**
   Learning rate schedules; large-batch training; hyperparameter optimization

3. **After training**
   Model ensembles, transfer learning

Today

Next time
Next time:
Training Neural Networks
(part 2)