Lecture 23: Neural Networks (cont), CNNs
Menu for Today (April 2, 2020)

Topics:

- Backpropagation
- Convolutional Layers
- Pooling Layer
- R-CNN

Readings:

- **Today’s** Lecture: N/A
- **Next** Lecture: N/A

Reminders:

- **Assignment 6**: Deep Learning due **Tuesday, April 7th**
Please fill out Student Evaluations (on Canvas)
Lecture 22: Re-cap

— The basic unit of computation in a neural network is a neuron.

— A neuron accepts some number of input signals, computes their weighted sum, and applies an activation function (or non-linearity) to the sum.

— Common activation functions include sigmoid and rectified linear unit (ReLU)
A neural network comprises neurons connected in an acyclic graph.
The outputs of neurons can become inputs to other neurons.
Neural networks typically contain multiple layers of neurons.

Example of a neural network with three inputs, a single hidden layer of four neurons, and an output layer of two neurons.

Figure credit: Fei-Fei and Karpathy
Neural Network

A neural network comprises neurons connected in an acyclic graph. The outputs of neurons can become inputs to other neurons. Neural networks typically contain multiple layers of neurons.

**Note:** each neuron will have its own vector of weights and a bias, it’s easier to think of all neurons in a layer as a single entity with a matrix of weights (size = number of inputs x number of neurons) and a vector of biases (size = number of neurons).

**Figure credit:** Fei-Fei and Karpathy
Neural Network

A neural network comprises neurons connected in an acyclic graph. The outputs of neurons can become inputs to other neurons. Neural networks typically contain multiple layers of neurons.

Note: each neuron will have its own vector of weights and a bias, it’s easier to think of all neurons in a layer as a single entity with a matrix of weights (size = number of inputs x number of neurons) and a vector of biases (size = number of neurons).

\[
\hat{y} = f(x, W_1, W_2, b_1, b_2) = \sigma \left( W_2^{(2 \times 4)} \sigma \left( W_1^{(4 \times 3)} x + b_1^{(4)} \right) + b_2^{(2)} \right)
\]

Figure credit: Fei-Fei and Karpathy
Activation Function

Why can’t we have \textit{linear} activation functions? Why have non-linear activations?
Activation Function

\[ \hat{y} = f(x, W_1, W_2, b_1, b_2) = \sigma \left( W_2^{(2 \times 4)} \sigma \left( W_1^{(4 \times 3)} x + b_1^{(4)} \right) + b_2^{(2)} \right) \]
\[ \hat{y} = f(x, W_1, W_2, b_1, b_2) = \sigma \left( W_2^{(2 \times 4)} \sigma \left( W_1^{(4 \times 3)} x + b_1^{(4)} \right) + b_2^{(2)} \right) \]

\[ = W_2^{(2 \times 4)} \left( W_1^{(4 \times 3)} x + b_1^{(4)} \right) + b_2^{(2)} \]
 Activation Function

\[ \hat{y} = f(x, W_1, W_2, b_1, b_2) = \sigma \left( W_2^{(2\times4)} \sigma \left( W_1^{(4\times3)} x + b_1^{(4)} \right) + b_2^{(2)} \right) \]

\[ = W_2^{(2\times4)} \left( W_1^{(4\times3)} x + b_1^{(4)} \right) + b_2^{(2)} \]

\[ = W_2^{(2\times4)} W_1^{(4\times3)} x + W_2^{(2\times4)} b_1^{(4)} + b_2^{(2)} \]
**Activation Function**

\[
\hat{y} = f(x, W_1, W_2, b_1, b_2) = \sigma \left( W_2^{(2\times4)} \sigma \left( W_1^{(4\times3)} x + b_1^{(4)} \right) + b_2^{(2)} \right)
\]

\[
= W_2^{(2\times4)} \left( W_1^{(4\times3)} x + b_1^{(4)} \right) + b_2^{(2)}
\]

\[
= W_2^{(2\times4)} W_1^{(4\times3)} x + W_2^{(2\times4)} b_1^{(4)} + b_2^{(2)}
\]

\[
\begin{aligned}
\hat{y} &= W_2^{(2\times4)} W_1^{(4\times3)} x + b_2^{(2)}
\end{aligned}
\]

**Figure credit:** Fei-Fei and Karpathy
Activation Function

Non-linear activation is required to provably make the Neural Net a **universal function approximator**

**Intuition:** with ReLU activation, we effectively get a linear spline approximation to any function.

Optimization of neural net parameters = finding slopes and transitions of linear pieces

The quality of approximation depends on the number of linear segments

Number of linear segments for large input dimension: $\Omega(2^{\frac{2}{3}Ln})$
**Light Theory**: Neural Network as Universal Approximator

**Universal Approximation Theorem**: Single hidden layer can approximate any continuous function with compact support to arbitrary accuracy, when the width goes to infinity.

[ Hornik et al., 1989 ]

**Universal Approximation Theorem (revised)**: A network of infinite depth with a hidden layer of size $d + 1$ neurons, where $d$ is the dimension of the input space, can approximate any continuous function.

[ Lu et al., NIPS 2017 ]

**Universal Approximation Theorem (further revised)**: ResNet with a single hidden unit and infinite depth can approximate any continuous function.

[ Lin and Jegelka, NIPS 2018 ]
Neural Network

How many neurons?
Neural Network

How many neurons?  \[4 + 2 = 6\]
How many neurons? 4 + 2 = 6

Neural Network

How many weights?
How many neurons? \[ 4 + 2 = 6 \]

How many weights? \[ (3 \times 4) + (4 \times 2) = 20 \]
Neural Network

How many neurons? 4 + 2 = 6

How many weights? 

(3 x 4) + (4 x 2) = 20

How many learnable parameters?
How many neurons? \(4 + 2 = 6\)

How many weights? \((3 \times 4) + (4 \times 2) = 20\)

How many learnable parameters? \(20 + 4 + 2 = 26\)

Neural Network
Modern \textbf{convolutional neural networks} contain 10-20 layers and on the order of 100 million parameters.

\textbf{Training} a neural network requires estimating a large number of parameters.
Backpropagation

When training a neural network, the final output will be some loss (error) function
— e.g. cross-entropy loss: \[ L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}} \right) \]

which defines loss for i-th training example with true class index \( y_i \); and \( f_j \) is the j-th element of the vector of class scores coming from neural net.
Backpropagation

When training a neural network, the final output will be some loss (error) function
— e.g. cross-entropy loss: \[ L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}} \right) \]

which defines loss for i-th training example with true class index \( y_i \); and \( f_j \) is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector \( x_i \) and predicts scores for 3 classes, with true class being class 3:
Backpropagation

When training a neural network, the final output will be some loss (error) function
— e.g. cross-entropy loss: \[ L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}} \right) \]

which defines loss for i-th training example with true class index \( y_i \); and \( f_j \) is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector \( x_i \) and predicts scores for 3 classes, with true class being class 3:

\[
\begin{align*}
  f \\
  c_1 &= -2.85 \\
  c_2 &= 0.86 \\
  c_3 &= 0.28
\end{align*}
\]
Backpropagation

When training a neural network, the final output will be some loss (error) function
— e.g. cross-entropy loss: \[ L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}} \right) \]
which defines loss for i-th training example with true class index \( y_i \); and \( f_j \) is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector \( x_i \) and predicts scores for 3 classes, with true class being class 3:

\[
\begin{align*}
  f & \\
  c_1 &= -2.85 & \exp & 0.058 \\
  c_2 &= 0.86 & \exp & 2.36 \\
  c_3 &= 0.28 & & 1.32
\end{align*}
\]
When training a neural network, the final output will be some loss (error) function
— e.g. cross-entropy loss: \[ L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}} \right) \]
which defines loss for i-th training example with true class index \( y_i \); and \( f_j \) is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector \( \mathbf{x}_i \) and predicts scores for 3 classes, with true class being class 3:

\[
\begin{align*}
f & \\
c_1 = -2.85 & \rightarrow \text{exp} \rightarrow 0.058 & \text{Normalize to sum to 1} \rightarrow 0.016 \\
c_2 = 0.86 & \rightarrow 2.36 & \rightarrow 0.631 \\
c_3 = 0.28 & \rightarrow 1.32 & \rightarrow 0.353
\end{align*}
\]
Backpropagation

When training a neural network, the final output will be some loss (error) function
— e.g. cross-entropy loss: \[ L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}} \right) \]
which defines loss for i-th training example with true class index \( y_i \); and \( f_j \) is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector \( \mathbf{x}_i \) and predicts scores for 3 classes, with true class being class 3:

\[
\begin{align*}
f & \\
c_1 &= -2.85 & \exp & 0.058 & \text{Normalize to sum to 1} & 0.016 \\
c_2 &= 0.86 & \exp & 2.36 & & 0.631 \\
c_3 &= 0.28 & \exp & 1.32 & & 0.353
\end{align*}
\]

probability of a class
When training a neural network, the final output will be some loss (error) function
— e.g. cross-entropy loss: 
\[ L_i = - \log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}} \right) \]
which defines loss for i-th training example with true class index \( y_i \); and \( f_j \)
is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector \( x_i \) and predicts scores for 3 classes, with true class being class 3:

\[
\begin{align*}
  \text{probability of a class} \\
  f & \\
  c_1 &= -2.85 \quad \text{exp} \quad 0.058 \quad \text{Normalize to sum to 1} \quad 0.016 \\
  c_2 &= 0.86 \quad 2.36 \quad 0.631 \\
  c_3 &= 0.28 \quad 1.32 \quad 0.353
\end{align*}
\]
Backpropagation

When training a neural network, the final output will be some loss (error) function
— e.g. cross-entropy loss: \( L_i = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}} \right) \)

which defines loss for i-th training example with true class index \( y_i \); and \( f_j \) is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector \( x_i \) and predicts scores for 3 classes, with true class being class 3:

\[
\begin{align*}
  f & \\
c_1 &= -2.85 & \exp & 0.058 & \text{Normalize to sum to 1} & 0.016 \\
c_2 &= 0.86 & \exp & 2.36 & & 0.631 \\
c_3 &= 0.28 & \exp & 1.32 & & 0.353 \\
\end{align*}
\]

\( L_i = -\log(0.353) = 1.04 \)
Backpropagation

When training a neural network, the final output will be some loss (error) function
— e.g. cross-entropy loss: \( L_i = - \log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}} \right) \)

which defines loss for i-th training example with true class index \( y_i \); and \( f_j \) is the j-th element of the vector of class scores coming from neural net.

We want to compute the **gradient** of the loss with respect to the network parameters so that we can incrementally adjust the network parameters
Gradient Descent

*slide adopted from V. Ordonex*
Gradient Descent

1. Start from random value of $W_0, b_0$

*slide adopted from V. Ordonex*
1. Start from random value of $W_0, b_0$
1. Start from random value of $W_0, b_0$

For $k = 0$ to max number of iterations

2. Compute gradient of the loss with respect to previous (initial) parameters:

$$\nabla L(W, b)|_{W=W_k, b=b_k}$$

*slide adopted from V. Ordonex*
Gradient Descent

1. Start from random value of $W_0, b_0$

2. Compute gradient of the loss with respect to previous (initial) parameters:

$$\nabla \mathcal{L}(W, b)|_{W=W_k, b=b_k}$$

For $k = 0$ to max number of iterations

*slide adopted from V. Ordonex*
1. Start from random value of $W_0, b_0$

For $k = 0$ to max number of iterations

2. Compute gradient of the loss with respect to previous (initial) parameters:

$$\nabla L(W, b)|_{W=W_k, b=b_k}$$

3. Re-estimate the parameters

$$W_{k+1} = W_k - \lambda \frac{\partial L(W, b)}{\partial W} \bigg|_{W=W_k, b=b_k}$$

$$b_{k+1} = b_k - \lambda \frac{\partial L(W, b)}{\partial b} \bigg|_{W=W_k, b=b_k}$$

*slide adopted from V. Ordonex*
**Gradient Descent**

1. Start from random value of \( W_0, b_0 \)

For \( k = 0 \) to max number of iterations

2. Compute gradient of the loss with respect to previous (initial) parameters:

\[
\nabla \mathcal{L}(W, b)\big|_{W=W_k, b=b_k}
\]

3. Re-estimate the parameters

\[
W_{k+1} = W_k - \lambda \frac{\partial \mathcal{L}(W, b)}{\partial W} \bigg|_{W=W_k, b=b_k}
\]

\[
b_{k+1} = b_k - \lambda \frac{\partial \mathcal{L}(W, b)}{\partial b} \bigg|_{W=W_k, b=b_k}
\]

*slide adopted from V. Ordonex*
1. Start from random value of $W_0, b_0$

For $k = 0$ to max number of iterations

2. Compute gradient of the loss with respect to previous (initial) parameters:

$$\nabla \mathcal{L}(W, b)|_{W=W_k, b=b_k}$$

3. Re-estimate the parameters

$$W_{k+1} = W_k - \lambda \frac{\partial \mathcal{L}(W, b)}{\partial W}|_{W=W_k, b=b_k}$$

$$b_{k+1} = b_k - \lambda \frac{\partial \mathcal{L}(W, b)}{\partial b}|_{W=W_k, b=b_k}$$

*slide adopted from V. Ordonex*
Gradient Descent

1. Start from random value of $W_0, b_0$

For $k = 0$ to max number of iterations

2. Compute gradient of the loss with respect to previous (initial) parameters:

$$\nabla \mathcal{L}(W, b) |_{W=W_k, b=b_k}$$

3. Re-estimate the parameters

$$W_{k+1} = W_k - \lambda \left. \frac{\partial \mathcal{L}(W, b)}{\partial W} \right|_{W=W_k, b=b_k}$$

$$b_{k+1} = b_k - \lambda \left. \frac{\partial \mathcal{L}(W, b)}{\partial b} \right|_{W=W_k, b=b_k}$$

$\lambda$ - is the learning rate

*slide adopted from V. Ordonex*
Re-cap

**Loss:**

\[ \mathcal{L}(y, \hat{y}) = \|y - \hat{y}\| = \|y - f(x, W_1, W_2, b_1, b_2)\| \]

\[ \hat{y} = f(x, W_1, W_2, b_1, b_2) = \sigma \left( W_2^{(2\times4)} \sigma \left( W_1^{(4\times3)} x + b_1^{(4)} \right) + b_2^{(2)} \right) \]
**Re-cap**

**Neural Network**

**Loss:**

\[ \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = ||\mathbf{y} - \hat{\mathbf{y}}|| = ||\mathbf{y} - f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2)|| \]

**Gradient Descent**

\[
\begin{align*}
\mathbf{W}_{1,i,j} &= \mathbf{W}_{1,i,j} - \lambda \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{W}_{1,i,j}} \\
\mathbf{b}_{1,i} &= \mathbf{b}_{1,i} - \lambda \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{b}_{1,i}}
\end{align*}
\]

\[
\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma \left( \mathbf{W}_2^{(2 \times 4)} \sigma \left( \mathbf{W}_1^{(4 \times 3)} \mathbf{x} + \mathbf{b}_1^{(4)} \right) + \mathbf{b}_2^{(2)} \right)
\]

**Figure credit:** Fei-Fei and Karpathy
Backpropagation

The parameters of a neural network are learned using backpropagation, which computes gradients via recursive application of the chain rule from calculus.
Backpropagation

The parameters of a neural network are learned using **backpropagation**, which computes gradients via recursive application of the **chain rule** from calculus.

Suppose $f(x, y) = xy$. What is the partial derivative of $f$ with respect to $x$? What is the partial derivative of $f$ with respect to $y$?
The parameters of a neural network are learned using backpropagation, which computes gradients via recursive application of the chain rule from calculus.

Suppose $f(x, y) = xy$. What is the partial derivative of $f$ with respect to $x$? What is the partial derivative of $f$ with respect to $y$?

\[
\frac{\partial f}{\partial x} = y \quad \text{and} \quad \frac{\partial f}{\partial y} = x
\]
Backpropagation

Suppose $f(x, y) = x + y$. What is the partial derivative of $f$ with respect to $x$? What is the partial derivative of $f$ with respect to $y$?
Suppose $f(x, y) = x + y$. What is the partial derivative of $f$ with respect to $x$? What is the partial derivative of $f$ with respect to $y$?

\[
\frac{\partial f}{\partial x} = 1 \quad \frac{\partial f}{\partial y} = 1
\]
A trickier example:  $f(x, y) = \max(x, y)$
Backpropagation

A trickier example:  \( f(x, y) = \max(x, y) \)

\[
\frac{\partial f}{\partial x} = 1(x \geq y) \quad \frac{\partial f}{\partial y} = 1(y \geq x)
\]

That is, the (sub)gradient is 1 on the input that is larger, and 0 on the other input

— For example, say \( x = 4, y = 2 \). Increasing \( y \) by a tiny amount does not change the value of \( f \) (\( f \) will still be 4), hence the gradient on \( y \) is zero.
Backpropagation

We can compose more complicated functions and compute their gradients by applying the **chain rule** from calculus.
Backpropagation

We can compose more complicated functions and compute their gradients by applying the **chain rule** from calculus.

Suppose \( f(x, y, z) = (x + y)z \). What are the partial derivatives of \( f \) with respect to \( x \)? \( y \)? \( z \)?
We can compose more complicated functions and compute their gradients by applying the \textbf{chain rule} from calculus.

Suppose \( f(x, y, z) = (x + y)z \). What are the partial derivatives of \( f \) with respect to \( x \)? \( y \)? \( z \)?

For illustration we break this expression into \( q = x + y \) and \( f = qz \). This is a sum and a product, and we have just seen how to compute partial derivatives for these.
We can compose more complicated functions and compute their gradients by applying the **chain rule** from calculus.

Suppose $f(x, y, z) = (x + y)z$. What are the partial derivatives of $f$ with respect to $x$, $y$, and $z$?

For illustration we break this expression into $q = x + y$ and $f = qz$. This is a sum and a product, and we have just seen how to compute partial derivatives for these.

By the chain rule

$$
\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = z \cdot 1 = z
$$
Backpropagation

We can compose more complicated functions and compute their gradients by applying the **chain rule** from calculus.

Suppose $f(x, y, z) = (x + y)z$. What are the partial derivatives of $f$ with respect to $x$, $y$, and $z$?

For illustration we break this expression into $q = x + y$ and $f = qz$. This is a sum and a product, and we have just seen how to compute partial derivatives for these.

By the chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = z \cdot 1 = z \quad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = z \cdot 1 = z \quad \frac{\partial f}{\partial z} = q$$
Backpropagation

\[ f(x, y, z) = (x + y)z \]
Backpropagation

\[ f(x, y, z) = (x + y)z \]

**Computational graph** (a DAG) with variable ordering from topological sort, where each **node** is an input, intermediate, or output variable.
Suppose the network input is: \((x, y, z) = (-2, 5, -4)\)

Then: \(q = x + y = 3\) \hspace{1cm} f = qz = -12\)  \textbf{(forward pass)}
Backpropagation

\[ f(x, y, z) = (x + y)z \]

Suppose the network input is: \((x, y, z) = (-2, 5, -4)\)

Then: \[ q = x + y = 3 \quad f = qz = -12 \quad \text{(forward pass)} \]

\[ \frac{\partial f}{\partial q} = z = -4 \quad \text{(backward pass)} \]
Backpropagation

\[ f(x, y, z) = (x + y)z \]

\[
\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = \frac{\partial f}{\partial q} \cdot 1
\]

Suppose the network input is: \((x, y, z) = (-2, 5, -4)\)

Then: \(q = x + y = 3\) \hspace{1cm} \(f = qz = -12\) \hspace{1cm} (forward pass)

\(\frac{\partial f}{\partial q} = z = -4\) \hspace{1cm} (backward pass)
Backpropagation

\[ f(x, y, z) = (x + y)z \]

\[
\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = \frac{\partial f}{\partial q} \cdot 1
\]

Suppose the network input is: \((x, y, z) = (-2, 5, -4)\)

Then: \(q = x + y = 3\) \hspace{1cm} \(f = qz = -12\) \hspace{1cm} (forward pass)

\[
\frac{\partial f}{\partial q} = z = -4 \hspace{1cm} \frac{\partial f}{\partial x} = -4
\]

(backward pass)
Backpropagation

\[ f(x, y, z) = (x + y)z \]

- \[ \frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = \frac{\partial f}{\partial q} \cdot 1 \]
- \[ \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = \frac{\partial f}{\partial q} \cdot 1 \]
- \[ \frac{\partial f}{\partial z} = q \]

Suppose the network input is: \((x, y, z) = (-2, 5, -4)\)

Then: \( q = x + y = 3 \quad f = qz = -12 \quad \text{(forward pass)} \)

- \[ \frac{\partial f}{\partial q} = z = -4 \]
- \[ \frac{\partial f}{\partial x} = -4 \]
- \[ \frac{\partial f}{\partial y} = -4 \]
- \[ \frac{\partial f}{\partial z} = 3 \quad \text{(backward pass)} \]
Example: Let’s Build (world smallest) Neural Network

Let's create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

![Pattern 1](image1.png)

![Pattern 2](image2.png)

![Pattern 3](image3.png)
Example: Let’s Build (world smallest) Neural Network

Let's create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

We will need some labeled data
**Example:** Let’s Build (world smallest) Neural Network

Let's create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images.
Example: Let’s Build (world smallest) Neural Network

Let's create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images.
Example: Let’s Build (world smallest) Neural Network

Let's create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images.

![Image of neural network and patterns]
Example: Let’s Build (world smallest) Neural Network

Let’s create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images.

What do we need to do?

First, let’s re-formulate the problem.
Example: Let’s Build (world smallest) Neural Network

Let’s create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images.

What do we need to do?

First, let’s re-formulate the problem.
Example: Let’s Build (world smallest) Neural Network

Let’s create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

Now, let’s build a network!

How many inputs should the network have? How neuron outputs?
Example: Let’s Build (world smallest) Neural Network

Let's create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images.

Input Layer

Output Layer

What else is missing for us to train it?
Example: Let’s Build (world smallest) Neural Network

Let’s create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

\[ L_i = - \log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}} \right) \]
Example: Let’s Build (world smallest) Neural Network

Let's create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

\[
L_1 = -\log \left( \frac{e^{\sum_{i=1}^{9} \sigma(w_{1i}x_i+b_1)}}{\sum_{j=1}^{3} e^{\sum_{i=1}^{9} \sigma(w_{1i}x_i+b_1)}} \right)
\]
Fully Connected Layer

Example: 200 x 200 image (small) x 40K hidden units

* slide from Marc'Aurelio Renzato
Fully Connected Layer

Example: 200 x 200 image (small) x 40K hidden units = \(~ 2 \text{ Billion} \) parameters (for one layer!)

* slide from Marc’Aurelio Renzato
Fully Connected Layer

Example: 200 x 200 image (small)
  x 40K hidden units

= ~ 2 Billion parameters (for one layer!)

Spatial correlations are generally local

Waste of resources + we don’t have enough data to train networks this large

* slide from Marc’Aurelio Renzato
Locally Connected Layer

**Example:** 200 x 200 image (small) 
× 40K hidden units

**Filter size:** 10 x 10

= ~ **4 Million** parameters

* slide from Marc’Aurelio Renzato*
Locally Connected Layer

**Example:** 200 x 200 image (small) x 40K hidden units

**Filter size:** 10 x 10

= ~ 4 Million parameters

**Stationarity** — statistics is similar at different locations

*slide from Marc’Aurelio Renzato*
**Convolutional Layer**

*Example:* 200 x 200 image (small) x 40K hidden units

**Filter size:** 10 x 10

= ~ 4 Million parameters

Share the same parameters across the locations (assuming input is stationary)

* slide adopted from Marc’Aurelio Renzato
Convolutional Layer

**Example:** 200 x 200 image (small) x 40K hidden units

**Filter size:** 10 x 10

= \sim 4 \text{ Million} \text{ parameters}

= 100 parameters

Share the same parameters across the locations (assuming input is stationary)

* slide adopted from Marc’Aurelio Renzato*
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc'Aurelio Renzato
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc'Aurelio Renzato
Convolutional Layer

* slide from Marc’Aurelio Renzato
Convolutional Layer

* slide from Marc'Aurelio Renzato
Convolution Layer

\[
\begin{bmatrix}
-1 & 0 & 1 \\
-1 & 0 & 1 \\
-1 & 0 & 1 \\
\end{bmatrix}
\]
Convolution Layer

\[
\begin{bmatrix}
0.11 & 0.11 & 0.11 \\
0.11 & 0.11 & 0.11 \\
0.11 & 0.11 & 0.11 \\
\end{bmatrix}
\]
Convolutional Layer

Example: 200 x 200 image (small) x 40K hidden units

Filter size: 10 x 10

# of filters: 20

Learn multiple filters

* slide from Marc’Aurelio Renzato
Convolutional Layer

Example: 200 x 200 image (small) x 40K hidden units

Filter size: 10 x 10

# of filters: 20

= 2000 parameters

Learn multiple filters

* slide from Marc’Aurelio Renzato
Convolutional Layer

32 x 32 x 3 image (note the image preserves spatial structure)

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Layer

32 x 32 x 3 image

32 height

5 x 5 x 3 filter

32 width

3 depth

Convolve the filter with the image (i.e., “slide over the image spatially, computing dot products”)

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Layer

32 x 32 x 3 image

5 x 5 x 3 filter

Convolve the filter with the image (i.e., “slide over the image spatially, computing dot products”)

Filters always extend the full depth of the input volume

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Layer

32 x 32 x 3 image

5 x 5 x 3 filter \((W)\)

1 number: the result of taking a dot product between the filter and a small 5 x 5 x 3 part of the image

\[W^T x + b, \text{ where } W, x \in \mathbb{R}^{75}\]
Convolutional Layer

32 x 32 x 3 image

5 x 5 x 3 filter (W)

1 number: the result of taking a dot product between the filter and a small 5 x 5 x 3 part of the image

\[ W^T x + b, \text{ where } W, x \in \mathbb{R}^{75} \]

How many parameters does the layer have?

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Layer

32 x 32 x 3 image

5 x 5 x 3 filter \( (W) \)

1 number: the result of taking a dot product between the filter and a small 5 x 5 x 3 part of the image

\[ W^T x + b, \text{ where } W, x \in \mathbb{R}^{75} \]

How many parameters does the layer have? 76

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Layer

32 x 32 x 3 image

5 x 5 x 3 filter \((W)\)

32 width

convolve (slide) over all spatial locations

activation map

28 height

1 depth

5 x 5 x 3 filter

32 width

3 depth

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Layer

32 x 32 x 3 image

5 x 5 x 3 filter \((W)\)

32 width

3 depth

convolve (slide) over all spatial locations

activation map

28 height

28 width

1 depth

consider another green filter

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Layer

If we have 6 5x5 filters, we’ll get 6 separate activation maps: activation map

32 height

convolutional layer

28 height

32 width

6 depth

28 width

this results in the “new image” of size 28 x 28 x 6!

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Layer

The number of neurons in a layer is determined by depth and stride parameter — also affected by zero-padding

**Depth**: Controls number of neurons that connect to the same region of the input layer
— a set of neurons connected to the same region is called a **depth column**

**Stride**: Controls spatial density. How far apart are depth columns?
Convolutional Layer: Closer Look at **Spatial Dimensions**

32 x 32 x 3 image

5 x 5 x 3 filter (W)

convolve (slide) over all spatial locations

activation map

32 width

1 depth

28 width

28 height

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Neural Network (ConvNet)

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Neural Network (ConvNet)

32 width 3 depth

CONV, ReLU

e.g. 6 5x5x3 filters

28 width 6 depth

28 height

32 height

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Neural Network (ConvNet)

- **32** height
- **32** width
- **6** depth

CONV, ReLU
- e.g. **6 5x5x3**
- **10 5x5x6**

*slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford*
Convolutional Neural Network (ConvNet)

CONV, ReLU

- e.g. 6x5x3 filters

32 width 32 height

6 depth

CONV, ReLU

- e.g. 10x5x6 filters

28 width 28 height

10 depth

24 width 24 height

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Neural Network (ConvNet)

32 height

CONV,
ReLU
e.g. **6 5x5x3** filters

28 height

CONV,
ReLU
e.g. **10 5x5x6** filters

24 height

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional Neural Network (ConvNet)

With padding we can achieve no shrinking (32 -> 28 -> 24); shrinking quickly (which happens with larger filters) doesn’t work well in practice

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Convolutional neural networks can be seen as learning a hierarchy of filters. As we go deeper in the network, filters learn and respond to increasingly specialized structures — The first layers may contain simple orientation filters, middle layers may respond to common substructures, and final layers may respond to entire objects.
What **filters** do networks learn?

[Zeiler and Fergus, 2013]
What **filters** do networks learn?

[ Zeiler and Fergus, 2013 ]
Pooling Layer

Let us assume the filter is an “eye” detector

How can we make detection spatially invariant (insensitive to position of the eye in the image)
Let us assume the filter is an “eye” detector

How can we make detection spatially invariant (insensitive to position of the eye in the image)

By “pooling” (e.g., taking a max) response over a spatial locations we gain robustness to position variations
Pooling Layer

- Makes representation smaller, more manageable and spatially invariant
- Operates over each activation map independently
Pooling Layer

- Makes representation smaller, more manageable and spatially invariant
- Operates over each activation map independently

How many parameters?

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Pooling Layer

- Makes representation smaller, more manageable and spatially invariant
- Operates over each activation map independently

How many parameters? None!

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Max Pooling

activation map

max pool with 2 x 2 filter and stride of 2

* slide from Fei-Dei Li, Justin Johnson, Serena Yeung, cs231n Stanford
Average Pooling

activation map

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

avg pool with 2 x 2 filter and stride of 2

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.25</td>
<td>5.25</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
**Object Classification**

![Leopard Image]

<table>
<thead>
<tr>
<th>Category</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dog</td>
<td>No</td>
</tr>
<tr>
<td>Cat</td>
<td>No</td>
</tr>
<tr>
<td>Couch</td>
<td>No</td>
</tr>
<tr>
<td>Flowers</td>
<td>No</td>
</tr>
<tr>
<td>Leopard</td>
<td>Yes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Problem:** For each image predict which category it belongs to out of a fixed set
Problem: For each image predict which category it belongs to out of a fixed set
**Problem:** For each image predict which category it belongs to out of a fixed set.
R-CNN

Input Image

* image from Ross Girshick

[ Girshick et al, CVPR 2014 ]
R-CNN

[ Girshick et al, CVPR 2014 ]

* image from Ross Girshick

Input Image

Regions of Interest from a proposal method (~2k)
R-CNN

[ Girshick et al, CVPR 2014 ]

* image from Ross Girshick
R-CNN

Input Image

Regions of Interest from a proposal method (~2k)

Warped image regions

Forward each region through a CNN
R-CNN

* image from Ross Girshick

**Input Image**

- Warped image regions

- Regions of Interest from a proposal method (~2k)

- Forward each region through a **CNN**

- Classify regions with SVM
R-CNN

**Linear Regression** for bounding box offsets

Classify regions with SVM

Forward each region through a **CNN**

Warped image regions

Regions of Interest from a proposal method (~2k)

Input Image

[ Girshick et al, CVPR 2014 ]

* image from Ross Girshick
R-CNN (Regions with CNN features) algorithm:
  — Extract promising candidate regions using an object proposals algorithm
  — Resize each proposal window to the size of the input layer of a trained
    convolutional neural network
  — Input each resized image patch to the convolutional neural network

**Implementation detail:** Instead of using the classification scores of the
network directly, the output of the final fully-connected layer can be used as an
input feature to a trained support vector machine (SVM)
The parameters of a neural network are learned using backpropagation, which computes gradients via recursive application of the chain rule.

A convolutional neural network assumes inputs are images, and constrains the network architecture to reduce the number of parameters.

A convolutional layer applies a set of learnable filters.

A pooling layer performs spatial downsampling.

A fully-connected layer is the same as in a regular neural network.

Convolutional neural networks can be seen as learning a hierarchy of filters.
Please fill out Student Evaluations (on Canvas)