Data & Donuts

Friday, February 17
3:00-4:00 pm
164 Angell Street, 3rd floor

Join us this week with Peter Hull, Professor of Economics

Organized by the DSI DUG.
Gourmet donuts, fruit, and coffee will be served.
CSCI 1470/2470
Spring 2023

Ritambhara Singh

February 17, 2023
Friday

Deep Learning

DALL-E 2 prompt “a painting of deep underwater with a yellow submarine in the bottom right corner”
Recap

Stacking multiple layers

More layers ▶ more complicated function
Linear layers are not sufficient!
Need non-linearity

Activation functions

Exploding gradients
Vanishing gradients
ReLU, Leaky ReLU

Diagram showing a sequence of operations: input → linear layer → activation function → output.
Recap: Reasons to use other activation functions

• Bounding network outputs to a particular range
  • Tanh: [-1, 1]
  • Sigmoid: [0,1]
  • Softplus: [0, ∞]

• Example: Predicting a person’s age from other biological features
  • Age is a strictly positive quantity
  • We can help our network learn by restricting it to output only positive numbers
  • Use a Softplus activation on the output
Today’s goal – continue to learn about multi-layer networks and learn about convolution

(1) What are hidden layers and hyperparameters?

(2) Universal approximate theorem – what a one-hidden layer network can learn?

(3) Intro to CNNs – Convolution
Recap: Consequences of adding activation layers

• Previously:
  - 1x784
  - 784x10
  - 10x1

• Now:
  - 1x784
  - 784x?
  - ?x10
  - 10x1

What dimension to use here??
“Hidden Layers”

• The output of a function that doesn’t feed into the output layer (like softmax) is called a **hidden layer**
• Have to set the size $h$ of these hidden layers
• More linear units $\rightarrow$ more hidden layer sizes
Hyperparameters

• Hidden layer sizes are a hyperparameter — configuration external to model, value usually set before training begins
  • Number of epochs, batch size, etc.
  • Contrast this with a learnable parameter, we keep talking about

• Rule of thumb
  • Start out making hidden layers the same size as the input
  • Then, tweak it to see the effect

• There are more principled (and time-consuming) ways to set them
  • Grid search, random search, Bayesian optimization...
  • See here for an overview and more references
What a multi-layer neural network could look like?

```
<table>
<thead>
<tr>
<th>Input</th>
<th>$1 \times 784$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 1</td>
<td>$784 \times h_1$</td>
</tr>
<tr>
<td>Layer 2</td>
<td>$h_1 \times h_2$</td>
</tr>
<tr>
<td>Layer 3</td>
<td>$h_2 \times h_3$</td>
</tr>
<tr>
<td>Layer 4</td>
<td>$h_3 \times 10$</td>
</tr>
<tr>
<td>Output</td>
<td>$10 \times 1$</td>
</tr>
</tbody>
</table>
```

$\sigma$
What functions can a one-hidden-layer neural net learn?
Universal Approximation Theorem [Cybenko ‘89]

- Remarkably, a one-hidden-layer network can actually represent any function (under the following assumptions):
  - Function is continuous
  - We are modeling the function over a closed, bounded subset of $\mathbb{R}^n$
  - Activation function is sigmoidal (i.e. bounded and monotonic)

- The proof of this theorem is an existence proof
  - i.e. it tells us that a network exists which can approximate any function, not how to actually learn it
A “Proof By Picture”
Universal Approximation Theorem “Proof”

• Start with a complex one dimensional function that relates some input $x$ to some output $y$
• We don’t know what the function that relates $x$ and $y$ is

$f(x)$

Universal Approximation Theorem “Proof”

• We can build up this function using simpler functions, i.e. box functions.
Universal Approximation Theorem “Proof”
How does this relate to activation functions?

- We can subtract two sigmoids to create these box functions
Universal Approximation Theorem “Proof”

• Summing up these simpler functions can do a pretty good job of approximating the actual function
Universal Approximation Theorem “Proof”

• Using more functions lets us model a complex function more accurately
  • Up to an arbitrary degree of accuracy, if we want
Universal Approximation Theorem “Proof”

• *Very* inefficient way to approximate
  - Need *lots* of box functions 🡪 *lots* of sigmoids 🡪 very large hidden layer

• Real networks trained with gradient descent can’t even learn these kinds of approximations
  - They **find smooth approximations**, require more hidden layers to get this same level of complexity.

• Nevertheless, the theorem is often cited to back up claims that a sufficiently complex neural net “can learn any function”

Do you remember what function a perceptron could not learn?
Can a multi-layer network learn XOR?

\[ w_1 \cdot x_1 + w_2 \cdot x_2 + b > 0 \]
Let’s find out

Google Tensorflow Playground
What kind of datasets CNNs are popularly applied to?

Convolution and CNNs
Does a network have to be fully connected?

Fully Connected

Partially Connected?

Why would you ever want to do this?
Partially Connected Networks?

• Fewer connections == Worse results? ...right?

• Advantages of Partial Connections
  • Fewer connections ⇨ fewer weights to learn
    • Faster training; more compact models; better generalization performance
  • Can design connectivity pattern that exploits knowledge of the data
    (like connecting patterns in features)

What’s a data type where we can do this?
Images!
When partially connected networks are useful

• **Observation:** Nearby pixels are more likely to be related

• **Assumption:** It is okay to only connect the nearby pixels
Limitations of Full Connections for MNIST

Suppose we’ve got a well-trained MNIST classifier...

#1 encoded as 🡪
Suppose we’ve got a well-trained MNIST classifier...
Limitations of Full Connections for MNIST

If we shift the digit to the right, then a different set of weights becomes relevant, network might have trouble classifying this as a 1...

Can you tell this is a 1?
This would **not** be a problem for the human visual system

Our eyes don’t look at absolute intensity values...

#1 encoded as 🡪

- this pixel has a low intensity
- this pixel has a high intensity
- this pixel has a low intensity
This would **not** be a problem for the human visual system

...but rather *local differences* in intensities

\[ \text{#1 encoded as } \square \]

- this intensity difference is large
- this intensity difference is zero
Translational Invariance

- To make a neural net $f$ robust in this same way, it should ideally satisfy *translational invariance*: $f(T(x)) = f(x)$, where
  - $x$ is the input image
  - $T$ is a translation (i.e. a horizontal and/or vertical shift)
Fully Connected Nets are **not** Translationally Invariant

How to make the network translationally invariant?

Focus on local differences/patterns

![Diagram showing pixel weights before and after translation](image)

- This pixel gets weight 0.6 before translation.
- This pixel gets weight 0.9 before translation.
- This pixel gets weight 0.1 before translation.
- This pixel gets weight 0.6 after translation.
- This pixel gets weight 0.9 after translation.
- This pixel gets weight 0.1 after translation.

**Sum of these three:** $0.6 \cdot 0.8 + 0.1 \cdot 0 + 0.9 \cdot 1 = 1.38$

**Sum of these three:** $0.6 \cdot 0 + 0.1 \cdot 0.4 + 0.9 \cdot 0 = 0.4$
Focusing on local patterns = partial connections

Fully Connected

Partially Connected

How do we do that?

Any questions?
The Main Building Block: Convolution

Convolution is an operation that takes two inputs:

1. An image (2D – B/W)
2. A filter (also called a kernel)

2D array of numbers; could be any values
What Convolution Does (Visually)

(We use this symbol for convolution)
(The verb form is “convolve”)

image

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

filter/kernel

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
What Convolution Does (Visually)

Overlay the filter on the image
What Convolution Does (Visually)

Sum up multiplied values to produce output value

\[2 \times 1 + 0 \times 1 + 1 \times 1 + 7 \times 0 + 0 \times 0 + 1 \times 0 + 0 \times -1 + 2 \times -1 + 5 \times -1 = -4\]
What Convolution Does (Visually)

Move the filter over by one pixel

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

output

-4
What Convolution Does (Visually)

Move the filter over by one pixel

<table>
<thead>
<tr>
<th>image</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 1 1</td>
<td>-4</td>
</tr>
<tr>
<td>7 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 -1 -1 -1</td>
<td></td>
</tr>
<tr>
<td>0 5 1 4</td>
<td></td>
</tr>
</tbody>
</table>
What Convolution Does (Visually)

Repeat (multiply, sum up)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output: \(-4\)
What Convolution Does (Visually)

Repeat (multiply, sum up)

Image:

<table>
<thead>
<tr>
<th></th>
<th>0 x1</th>
<th>1 x1</th>
<th>3 x1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1 x0</td>
<td>1 x0</td>
<td>0 x0</td>
</tr>
<tr>
<td>0</td>
<td>2 x-1</td>
<td>5 x-1</td>
<td>0 x-1</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Output:

$$0x1 + 1x1 + 3x1 + 0x0 + 1x0 + 0x0 + 2x-1 + 5x-1 + 0x-1$$

Output matrix:

$$\begin{bmatrix} -4 & -3 \\ -4 & -3 \end{bmatrix}$$
### What Convolution Does (Visually)

Repeat...

<table>
<thead>
<tr>
<th>Image</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-4</td>
</tr>
<tr>
<td>0</td>
<td>-3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

```
7x1 + 1x1 + 1x1 + 0x0 + 2x0 + 5x0 + 0x-1 + 5x-1 + 1x-1
```
What Convolution Does (Visually)

Repeat...

image

<table>
<thead>
<tr>
<th>2</th>
<th>0</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1\times 1</td>
<td>1\times 1</td>
<td>0\times 1</td>
</tr>
<tr>
<td>0</td>
<td>2\times 0</td>
<td>5\times 0</td>
<td>0\times 0</td>
</tr>
<tr>
<td>0</td>
<td>5\times -1</td>
<td>1\times -1</td>
<td>4\times -1</td>
</tr>
</tbody>
</table>

output

\[1\times 1 + 1\times 1 + 0\times 1 + 2\times 0 + 5\times 0 + 0\times 0 + 5\times -1 + 1\times -1 + 4\times -1\]

\[
\begin{array}{cc}
-4 & -3 \\
3 & -8 \\
\end{array}
\]
What Convolution Does (Visually)

In summary:

<table>
<thead>
<tr>
<th>image</th>
<th>filter/kernel</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 0 1 3</td>
<td>1 1 1 0 0 -1 -1 -1</td>
<td>-4 -3 3 -8</td>
</tr>
</tbody>
</table>
Try it out yourself!

Convolve this image

<table>
<thead>
<tr>
<th>2</th>
<th>0</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

With this filter

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

\[ 2 \times 2 \]
\[
\begin{array}{cccc}
2 & 0 & 3 & 1 \\
1 & 1 & 0 & 0 \\
1 & 0 & 2 & 0 \\
1 & 0 & 1 & 2 \\
\end{array}
\times
\begin{array}{ccc}
1 & 0 & -1 \\
2 & 0 & -2 \\
1 & 0 & -1 \\
\end{array}
=
\begin{array}{c}
\end{array}
\]
Recap

Building multi-layer neural networks

- Hidden layers
- What a one-hidden layer network can learn
- What a multi-layer network can learn
- Partially connected networks are useful (e.g., for images!)
- Fully connected networks are not transitionally invariant
- Convolutional filter

Introduction to CNNs