Introduction to Deep Learning

Georgia Tech CS 4650/7650
Fall 2020
Outline

● Deep Learning
  ○ CNN
  ○ RNN
  ○ Attention
  ○ Transformer

● Pytorch
  ○ Introduction
  ○ Basics
  ○ Examples
CNNs

Some slides borrowed from Fei-Fei Li & Justin Johnson & Serena Yeung at Stanford.
Fully Connected Layer

- **Input**: 32x32x3 image
- **Flattened image**: $32 \times 32 \times 3 = 3072$
- **Weight Matrix**: $Wx$ with $10 \times 3072$ weights
- **Output**: Activation function with 10 outputs
Convolutional Layer

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.
Convolutional Layer

At each step during the convolution, the filter acts on a region in the input image and results in a single number as output.

This number is the result of the dot product between the values in the filter and the values in the 5x5x3 chunk in the image that the filter acts on.

Combining these together for the entire image results in the activation map.
Convolutional Layer

Filters can be stacked together.

Example- If we had 6 filters of shape 5x5, each would produce an activation map of 28x28x1 and our output would be a “new image” of shape 28x28x6.
Convolutional Layer

Visualizations borrowed from Irhum Shafkat’s blog.
Convolutional Layer

Standard Convolution

Convolution with Padding

Convolution with strides

Visualizations borrowed from vdumoulin's [github repo](https://github.com/).
Convolutional Layer

Output Size: 
\[(N - F)/\text{stride} + 1\]

e.g. \(N = 7, F = 3, \text{stride} 1\)
\[=> (7 - 3)/1 + 1 = 5\]
e.g. \(N = 7, F = 3, \text{stride} 2\)
\[=> (7 - 3)/2 + 1 = 3\]
Pooling Layer

- makes the representations smaller and more manageable
- operates over each activation map independently
Max Pooling

Single depth slice

max pool with 2x2 filters and stride 2

x

y
ConvNet Layer

Image credits- Saha's blog.
NLP doesn’t use convolutional nets a lot

Some adjacent applications exist, such as graph convolutions or image-to-text

For text sequences, it sometimes helps to use 1-dimensional convolutions (because embedding dimension ordering has no intrinsic meaning)

What does this basically amount to?

N-gram features.
RNNs

Some slides borrowed from Fei-Fei Li & Justin Johnson & Serena Yeung at Stanford.
Vanilla Neural Networks

House Price Prediction

Input
- size
- #bedrooms
- zip code
- wealth

Hidden Layers

Output
- y

Vanilla Neural Networks
How to model sequences?

- Text Classification: Input Sequence → Output label
- Translation: Input Sequence → Output Sequence
- Image Captioning: Input image → Output Sequence
RNN - Recurrent Neural Networks

- **Vanilla Neural Networks**
  - One to one: Image captioning
  - One to many: Text classification
  - Many to one: Translation
  - Many to many: POS tagging
RNN - Representation

Output Vector

Hidden state fed back into the RNN cell

Input Vector
RNN - Recurrence Relation

The RNN cell consists of a hidden state that is updated whenever a new input is received. At every time step, this hidden state is fed back into the RNN cell.

\[ h_t = f_W(h_{t-1}, x_t) \]

- \( h_t \): new state
- \( h_{t-1} \): old state
- \( x_t \): input vector at some time step
- \( f_W \): some function with parameters \( W \)

Output Vector

Hidden state fed back into the RNN cell

Input Vector
RNN - Rolled out representation

$h_0 \xrightarrow{f_W} h_1 \xrightarrow{f_W} h_2 \xrightarrow{f_W} h_3 \xrightarrow{\ldots} h_T$

$y_1 \xleftarrow{f_W} x_1$

$y_2 \xleftarrow{f_W} x_2$

$y_3 \xleftarrow{f_W} x_3$
RNN - Rolled out representation

\[ y_1 \rightarrow L_1 \quad y_2 \rightarrow L_2 \quad y_3 \rightarrow L_3 \quad y_T \rightarrow L_T \]

\[ h_0 \rightarrow f_W h_1 \rightarrow f_W h_2 \rightarrow f_W h_3 \rightarrow \ldots \rightarrow h_T \]

\[ x_1 \rightarrow f_W x_2 \rightarrow f_W x_3 \rightarrow \ldots \rightarrow f_W x_T \]

Same Weight matrix - W
RNN - Backpropagation Through Time

Forward pass through entire sequence to produce intermediate hidden states, output sequence and finally the loss. Backward pass through the entire sequence to compute gradient.
Running Backpropagation through time for the entire text would be very slow. Switch to an approximation—Truncated Backpropagation Through Time
RNN - Truncated Backpropagation Through Time

Run forward and backward through chunks of the sequence instead of whole sequence
RNN - Truncated Backpropagation Through Time

Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps.
RNN Types

The 3 most common types of Recurrent Neural Networks are:

1. Vanilla RNN
2. LSTM (Long Short-Term Memory)
3. GRU (Gated Recurrent Units)

Some good resources:

Understanding LSTM Networks

An Empirical Exploration of Recurrent Network Architectures

Recurrent Neural Network Tutorial, Part 4 – Implementing a GRU/LSTM RNN with Python and Theano

Stanford CS231n: Lecture 10 | Recurrent Neural Networks
Attention

Some slides borrowed from Sarah Wiegreffe at Georgia Tech and Abigail See, Stanford CS224n.
RNN

Encoding of the source sentence. This needs to capture all information about the source sentence. Information bottleneck!

Source sentence (input)

Target sentence (output)

Decoder RNN
RNN - Attention
RNN - Attention
RNN - Attention
RNN - Attention

On this decoder timestep, we're mostly focusing on the first encoder hidden state ("he")

Take softmax to turn the scores into a probability distribution

Source sentence (input)
RNN - Attention

Use the attention distribution to take a **weighted sum** of the encoder hidden states.

The attention output mostly contains information from the hidden states that received high attention.
RNN - Attention

- Encoder RNN
  - Attention distribution
  - Attention scores

- Decoder RNN

Source sentence (input): il a m' entarté

Concatenate attention output with decoder hidden state, then use to compute $\hat{y}_1$ as before.
Sometimes we take the **attention output** from the previous step, and also feed it into the decoder (along with the usual decoder input).
RNN - Attention

Sequence-to-sequence with attention
RNN - Attention

Attention is a way to obtain a fixed-size representation of an arbitrary set of representations (the values), dependent on some other representation (the query).
Attention

- For query vector $q$, key vector $k_i$ representing value $v_i$
  - $s_i$ is the similarity score between $q$ and $k_i$
- Normalize the similarity scores to sum to 1
  - $p_i = \text{Softmax}(s_i)$
- Compute $z$ as the weighted sum of the value vectors $v_i$
  weighted by their scores $p_i$
- In Machine Translation & Image Captioning, the keys and
  values are the same.
  - But, they could be different.
  
  $$z = \sum_{i=1}^{L} p_i v_i$$
Attention is great

- Attention significantly **improves performance** (in many applications)
  - It’s very useful to allow decoder to focus on certain parts of the source
- **Attention solves the bottleneck problem**
  - Attention allows decoder to look directly at source; bypass bottleneck
- **Attention helps with vanishing gradient problem**
  - Provides shortcut to faraway states
- **Attention provides some interpretability**
  - By inspecting attention distribution, we can see what the decoder was focusing on
Drawbacks of RNN

- RNNs involve sequential computation
  - can’t parallelize = time-consuming
- RNNs “forget” past information
- No explicit modeling of long and short range dependencies
Transformer

Some slides borrowed from Sarah Wiegreffe at Georgia Tech and “The Illustrated Transformer”
https://jalammar.github.io/illustrated-transformer/
Transformer

“Attention is All You Need”  
(Vaswani et. al 2017)
Self-Attention
Self-Attention
Self-Attention

Input
Embedding
Queries
Keys
Values
Score
Divide by $8 \cdot \sqrt{d_k}$
Softmax
Softmax X Value
Sum

Thinking

Machines

$q_1 \cdot k_1 = 112$
$q_1 \cdot k_2 = 96$

14
0.88

12
0.12

$v_1$
$v_2$

$z_1$
$z_2$
Self-Attention

$$A(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V$$
Multi-Head Self-Attention

Parallel attention layers with different linear transformations on input and output.
Retaining Hidden State Size
Details of Each Attention Sub-Layer of Transformer Encoder

1) This is our input sentence*
2) We embed each word*
3) Split into 8 heads. We multiply X or R with weight matrices
4) Calculate attention using the resulting Q/K/V matrices
5) Concatenate the resulting Z matrices, then multiply with weight matrix W⁰ to produce the output of the layer

* In all encoders other than #0, we don’t need embedding. We start directly with the output of the encoder right below this one.
Each Layer of Transformer Encoder
Positional Encoding
Each Layer of Transformer Decoder
Transformer Decoder - Masked Multi-Head Attention

Problem of Encoder self-attention: we can’t see the future!
Transformer

“Attention is All You Need”
(Vaswani et. al 2017)

Encoder →  Decorder
Thank you!