Review of Lecture 14

- **GoogLeNet’s Inception module**
  - Multiple filter sizes in one module

- Dimensionality reduction to reduce ops

Inception module with dimensionality reduction

Naïve Inception module

With dimensionality reduction

1x1 convolutions for projection

- Preserves spatial dims but reduces depth!
- Projects depth to lower dimension (essentially combining activation maps)
Review of Lecture 14

- **Full GoogLeNet architecture**
  - Eliminated expensive FC layers for classification
  - Global Average Pooling

**GAP idea:**
1. Generate 1 activation map per class
2. Take the average of each map and feed this vector to softmax

- Used auxiliary classification units to inject gradient
- Better choice: Batch Normalization

\[ B = \{x_1, \ldots, x_m\} \quad \text{(a batch of activations)} \]

\[ \text{BN}_{\gamma, \beta}(x) : x_1, \ldots, x_m \rightarrow y_1, \ldots, y_m \quad \text{(desired transform)} \]

Input: Values of \( x \) over a mini-batch: \( B = \{x_1, \ldots, x_m\} \)

Parameters to be learned: \( \gamma, \beta \)

Output: \( \{y_i = \text{BN}_{\gamma, \beta}(x_i)\} \)

\[ \mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad // \text{mini-batch mean} \]

\[ \sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad // \text{mini-batch variance} \]

\[ \bar{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad // \text{normalize} \]

\[ y_i \leftarrow \gamma \bar{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift} \]

Let model learn the best distribution! \hspace{1cm} \text{(Ioffe and Szegedy 2015)}

(not forced to have mean 0, var of 1, but still standardized)
Review of Lecture 14

- Implementing Batch Norm in PyTorch

  ```python
  self.layer3 = nn.Sequential(  
    nn.Conv2d(60, 60, kernel_size=3, stride=1),  
    nn.BatchNorm2d(60),  
    nn.ReLU(),  
    nn.MaxPool2d(kernel_size=2, stride=1)
  )
  ```

- By convention, normalize the signal at neural input.

  **ResNet: Residual connections, deep networks**

  - Very deep networks using residual connections
    - 152-layer model for ImageNet
    - ILSVRC’15 classification winner (3.57% top-5 error)
    - Swept all classification and detection competitions in ILSVRC’15 and COCO’15

- Fit a residual mapping to combat optimization issues

  **ResNet**

  ![ResNet diagram]

  **Solution:** Use network layers to fit a ‘residual mapping’ instead of directly trying to fit a desired underlying mapping.

  \[ H(x) = F(x) + x \]

  - Also added “bottleneck” layers for dimensionality reduction
Other practicalities

- Xavier, Xavier/2 initialization
  1. Zero mean
  2. Want same variance into and out of neuron
  3. Gaussian or uniform distribution

- Initialize the weights with this variance and zero mean:

\[
\text{var}(w_i) = \frac{1}{n_{in}} \quad (n_{in} \text{ number of inputs})
\]  

(Xavier)

- Want this to hold for both forward and backward pass
- Take average of fan-in/fan-out to include backprop

\[
\text{var}(w_i) = \frac{2}{n_{in} + n_{out}}
\]  

(Glorot & Bengio)

- Multiply by 2 for ReLU (since 0 for half of input)

\[
\text{var}(w_i) = \frac{2}{n_{in}}
\]  

(He et al. 2015)

See torch.nn.init!
Review of Lecture 14

- Rapid advancement in accuracy and efficiency
- Work needed on processing rates and power

An Analysis of Deep Neural Network Models for Practical Applications (Canziani et al. 2017)
Review of Lecture 14

Many other notable architectures and advancements...

- Network in Network: Influenced both GoogLeNet and ResNet
- More general use of “identity mappings” from ResNet
- Wider and aggregated networks
- Networks with stochastic depth
- Fractal networks
- Densely connected convolutional networks
- Efficient networks (compression)

... to name a few!
Today’s Lecture

• Brief introduction to recurrent neural networks

• Course project!

(Many slides adapted from Stanford’s excellent CS231n course. Thank you Fei-Fei Li, Justin Johnson, and Serena Young!)
Feedforward neural networks

One input

One output
Recurrent neural networks process sequences

One-to-one

One-to-many

E.g., **image captioning**
Image to **sequence** of words
Recurrent neural networks process sequences

One-to-one  One-to-many  Many-to-one

E.g., sentiment classification
Sequence of words to sentiment
Recurrent neural networks process sequences

- One-to-one
- One-to-many
- Many-to-one
- Many-to-many

E.g., machine translation sequence of words to sequence of words
Recurrent neural networks process sequences

- One-to-one
- One-to-many
- Many-to-one
- Many-to-many

E.g., video classification at frame-level
Sequential processing of non-sequential data

Classify images by “sequential spatial attention”

Reading MNIST

(Gregor et al., “DRAW: A Recurrent Neural Network For Image Generation”, ICML 2015)
Sequential processing of non-sequential data

Generate images one portion at a time!

Generating MNIST

(Gregor et al., “DRAW: A Recurrent Neural Network For Image Generation”, ICML 2015)
Recurrent neural network

Want to predict a vector at time steps
Can process a sequence of vectors $x_t$ by applying a recurrence formula at each time step:

$$h_t = f_W(h_{t-1}, x_t)$$

The same function and parameters are used at every time step!
A “plain” recurrent neural network

The state consists of a single “hidden” vector $\mathbf{h}$:

$$ h_t = f_W(h_{t-1}, x_t) $$

$$ h_t = \tanh \left( W_{hh} h_{t-1} + W_{xh} x_t \right) $$

$$ y_t = W_{hy} h_t $$
Recurrent feedback can be ‘unrolled’ into a computation graph.
The RNN computation graph

Recurrent feedback can be ‘unrolled’ into a computation graph.
The RNN computation graph

Recurrent feedback can be ‘unrolled’ into a computation graph.
Reuse the **same** weight matrix at each time step!

When thinking about backprop, separate gradient will flow back for each time step.
The RNN computation graph (many-to-many)

A second network can generate the output that we desire.
Can compute an individual loss at every time step as well (e.g., softmax loss of labels)
The RNN computation graph (many-to-many)

Total loss for entire training step is sum of losses (compute gradient of loss w.r.t. W)
The RNN computation graph (many-to-one)

Final hidden state summarizes all context (can best predict, e.g., sentiment)
The RNN computation graph (one-to-many)

Fixed size input, variably sized output (e.g., image captioning)
**Sequence-to-sequence**

**Many-to-one:** Encode input sequence into single vector...

**One-to-many:** Decode output sequence from single input vector....

**Variably-sized input and output (e.g., machine translation)**
Example:
Character-level language model

Predicting the next character...

Vocabulary:
[h,e,l,o]

Training sequence: "hello"
**Example:**
Character-level language model

Predicting the next character...

**Vocabulary:** [h,e,l,o]

**Training sequence:** “hello”

\[ h_t = \tanh (W_{hh}h_{t-1} + W_{xh}x_t) \]
Example: Character-level language model

Predicting the next character...

Vocabulary: [h,e,l,o]

Training sequence: “hello”
Example:
Character-level language model sampling

Vocabulary: [h,e,l,o]

At test time, sample characters one at a time and feed back to model.
Example:
Character-level language model sampling

Vocabulary: [h,e,l,o]

At test time, sample characters one at a time and feed back to model
Example:
Character-level language model sampling

Vocabulary: [h,e,l,o]

At test time, sample characters one at a time and feed back to model.
Example: Character-level language model sampling

Vocabulary: [h, e, l, o]

At test time, sample characters one at a time and feed back to model.
Backpropagation through time

Run forward through entire sequence to compute loss, then backward through entire sequence to compute gradient.
Truncated backpropagation through time

Run forward and backward through ‘chunks’ of the sequence instead of entire sequence.
**Truncated backpropagation through time**

Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps.
Truncated backpropagation through time
Further reading


