Announcements

- HW4 available. Due Friday 12/8. Note will take a while to run some things.
- Final Exam: Thursday 12/14 at 7pm-10pm
- Review session: TBD
- Discussion of Final on wed class.

Where have we been so far

- Supervised classification
  - Feature space
  - Nearest Neighbor (kth nearest neighbor)
  - Bayesian (MAP) classifier
  - Support Vector Machines (didn’t talk about this, but important)
- Curse of dimensionality
- Dimensionality reduction
  - Principal component analysis
  - Fisher’s linear discriminant

Accuracy of PCA + K-NN

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>KNN + 3.072 Features</td>
<td>33.86</td>
</tr>
<tr>
<td>KNN + 200 PCA Comp.</td>
<td>36.54</td>
</tr>
<tr>
<td>KNN + 75 PCA Comp.</td>
<td>39.77</td>
</tr>
<tr>
<td>KNN + 50 PCA Comp.</td>
<td>40.12</td>
</tr>
<tr>
<td>KNN + 40 PCA Comp.</td>
<td>40.93</td>
</tr>
<tr>
<td>KNN + 30 PCA Comp.</td>
<td>41.78</td>
</tr>
<tr>
<td>KNN + 25 PCA Comp.</td>
<td>41.57</td>
</tr>
<tr>
<td>KNN + 15 PCA Comp.</td>
<td>38.75</td>
</tr>
<tr>
<td>KNN + 10 PCA Comp.</td>
<td>34.93</td>
</tr>
</tbody>
</table>

Example of Feature Extraction

**Step 2B. Filtering**

\[ I_P \in \mathbb{R}^3 \]

**Step 3. Each Pixel is Mapped to a Visual Word**

\[ I_P = \arg \min_{i} ||a_i - F_P||_2 \]

**Step 4B. Histograms at Multiple Scales**

Neural Networks

**NEW NAVY DEVICE LEARNS BY DOING**

Wendell Willkie, ex-president and chairman of the board of CSX Corporation, connects the first pilot model of the Mark I Perceptron to a camera that produces 20x20 pixels of image data. The Mark I Perceptron was the first implementation of the perceptron algorithm, which is a linear classifier that is able to learn from data. The machine was connected to a camera that produced 400 pixel images. The main visible feature is a patchboard that allowed experimentation with different combinations of input features. To the right of that are arrays of potentiometers that implemented the adaptive weights.

The Mark I Perceptron machine

- The Mark I Perceptron machine was the first implementation of the perceptron algorithm. The machine was connected to a camera that used 20x20 cadmium sulfide photocells to produce a 400-pixel image. The main visible feature is a patchboard that allowed experimentation with different combinations of input features. To the right of that are arrays of potentiometers that implemented the adaptive weights.

**1958 New York Times...**

In today’s demonstration, the “TN” was fed two cards, one with black circles on the left side and the other with squares on the right side.

The Perceptron machine was connected to a camera that produced 20x20 pixel images. The main visible feature is a patchboard that allowed experimentation with different combinations of input features. To the right of that are arrays of potentiometers that implemented the adaptive weights.

**From Wikipedia**
For a Network, even as simple as a single perceptron, we can ask questions:

1. What can be represented with it?
2. How do we evaluate it?
3. How do we train it?

Note: For \( x = (x_1, \ldots, x_n) \), \( x_i \) can be binary or a real number.

\[ o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ 0 & \text{otherwise}. \end{cases} \]

How powerful is a perceptron?

- Inverter
- Boolean AND
- Boolean OR
- Boolean XOR

Increasing Expressiveness: Multi-Layer Neural Networks

But where did those weights come from?

Stay tuned.
The nodes of multilayered network

\[ f(\mathbf{x}; \mathbf{w}) = S(\mathbf{w}^T \mathbf{x} + w_0) \]

Nonlinearities: \( S(x) \)
- Threshold (perceptron)
- Sigmoid
- Rectified Linear Unit (ReLU)

Rectified Linear Unit (ReLU)

\[ g(z) = \max(0, z) \]

Feedforward Networks

- Let \( y = f^*(x) \) be some function we are trying to approximate
- This function could be assignment of an input to a category as in a classifier
- Let a feedforward network approximate this mapping \( y = f(\mathbf{x}; \mathbf{w}) \) by learning parameters \( \mathbf{w} \)
Feedforward Networks

- Feedforward networks have **NO** feedback
- These networks can be represented as directed acyclic graphs describing the composition of functions
- These networks are composed of functions represented as "layers" \( f(x) = h^1(h^2(\cdots h^t(x))) \)
- The length of the chain of compositions gives the "depth" of the network

Universal Approximation Theorem

- **tldr:** if we have enough hidden units we can approximate "any" function! … but we may not be able to train it.
- **Universal Approximation Theorem:** A feedforward neural network with a linear output layer and one or more hidden layers with ReLU [Leshno et al. '93], or sigmoid or some other "squashing" activation function [Hornik et al. '89, Cybenko '89] can approximate any continuous function on a closed and bounded subset of \( \mathbb{R}^n \). This holds for functions mapping finite dimensional discrete spaces as well.

Universal Approximation Theorem: Caveats

- Optimization may fail to find the parameters needed to represent the desired function.
- Training might choose the wrong function due to overfitting.
- The network required to approximate this function might be so large as to be infeasible.

So even though "any" function can be approximated with a network as described with single hidden layer, the network may fail to train, fail to generalize, or require so many hidden units as to be infeasible.

- This is both encouraging and discouraging!
- However, [Montufar et al. 2014] showed that **deeper networks are more efficient** in that a deep rectified net can represent functions that would require an exponential number of hidden units in a shallow one hidden layer network.
- Deep networks composed of many rectified hidden layers are good at approximating functions that can be composed from simpler functions. And lots of tasks such as image classification may fit nicely into this space.

High level view of evaluation and training

- Training data: \( \{(x_i, y_i): 1 \leq i \leq n\} \)
- Training: Find \( w \) that minimizes the total loss.
- \( x \rightarrow f(x,w) \rightarrow y \rightarrow \text{Loss} \rightarrow L(y,\hat{y}) \)
- \( \hat{y} \rightarrow \text{Loss} \rightarrow L(y,\hat{y}) \)
- **Total Loss:** \( \sum_c l(\hat{y}(x,w), c) \)
The loss function

- The loss function is really important. It determines how we compare what the network produces to our labels.
- Common ones:
  - Regression problems:
    - Distance: \( L(y, \hat{y}) = ||y - \hat{y}||^p \), usually \( p = 1 \) or \( 2 \)
  - Classification
    - Cross entropy (See homework)
    - Softmax
      \[
      p(y = i|x) = \frac{e^{x_i}}{\sum_j e^{x_j}}
      \]
- Common ones:
  - Regression problems:
    - Distance: \( L(y, \hat{y}) = ||y - \hat{y}||^p \), usually \( p = 1 \) or \( 2 \)
  - Classification
    - Cross entropy (See homework)
    - Softmax
      \[
      p(y = i|x) = \frac{e^{x_i}}{\sum_j e^{x_j}}
      \]

Gradient-Based Optimization

Gradient Descent

Critical Points

What if our input is a vector?

- Let \( f(x) : \mathbb{R}^n \rightarrow \mathbb{R} \)
  - The **directional derivative** is the slope of the function in direction \( u \)
  - We can find this as \( \frac{\partial}{\partial \eta} f(x + \eta u) \) at \( \eta = 0 \)
  - ...or after the chain rule yields \( \nabla_x f(x)^T u \)
PLEASE DON'T FORGET

Gradient Descent

- So if we move in direction \( \mathbf{u} \) the slope is \( \nabla_x f(x)^T \mathbf{u} \)
- So in what direction is the slope most negative?
- Clearly in the OPPOSITE direction of the gradient!
- And if we traverse the fcn this way then we are doing steepest descent or gradient descent.

\[
x_{i+1} = x_i - \eta \nabla_x f(x_i)
\]

Optimization for Deep Nets

- Deep learning optimization is a type of global optimization where the optimization is usually expressed as a loss summed over all the training samples.
- Our goal is not so much find the parameters (or weights) that minimize the loss but rather to find parameters that produce a network with the desired behavior.
- Note that there are LOTS of solutions to which our optimization could converge to—with very different values for the weights—but each producing a model with very similar behavior on our sample data.
- For example, consider all the permutations of the weights in a hidden layer that produce the same outputs.

Optimization for Deep Nets

- Although there is a seemingly endless literature on global optimization, here we consider only gradient descent-based methods.
- Our optimizations for deep learning are typically done in very high-dimensional spaces, where the parameters we are optimizing can run into the millions.
- And for these optimizations, when starting the training from scratch (i.e., some random initialization of the weights) we will need LOTS of labeled training data.
- The process of learning our model from this labeled data is referred to as supervised learning. Although, supervised learning is more general than the deep learning algorithms we will consider.

Back propagation

- Basically another name for gradient descent
- Because of the nature of the network \( l_j(\{l_i(x;w_i);w_j\};w_k) \), gradients with respect to \( w_i \) are determined by chain rule
- Can be thought of as “propagating” from loss function to input.
- You’ll use adaptive step size method called ADAM.

Training and Validation Sets

Labeled Data

Training Data

Validation Data
The goal of regularization is to prevent overfitting the training data with the hope that this improves generalization, i.e., the ability to correctly handle data that the network has not trained on.
**Dropout** [Srivastava et al., 2014]

- For every training batch through the network, dropout 0.5 of hidden units and 0.2 of input units. You can choose the probabilities as you like.
- Train as you normally would using SGD, but each time you impose a random dropout that essentially trains for that batch on a random sub-network.
- When you are done training, you use for your model the complete network with all its learned weights, except multiply the weight by the probability of including its parent unit.
- This is called the **weight scaling inference rule** [Hinton et al., 2012].

**Early Stopping**

- Typical deep neural networks have millions and millions of weights!
- With so many parameters at their disposal how to we prevent them from overfitting?
- Clearly we can use some of the other regularization techniques that have been mentioned…
- …but given enough training time, our network will eventually start to overfit the data.

**Deterministic vs. Stochastic Methods**

- If we performed our gradient descent optimization using all the training samples to compute each step in our parameter updates, then our optimization would be **deterministic**.
- Confusingly, deterministic gradient descent algorithms are sometimes referred to as **batch** algorithms.
- In contrast, when we use a subset of randomly selected training samples to compute each update, we call this **stochastic gradient descent** and refer to the subset of samples as a **mini-batch**.
- And even more confusingly, we often call this mini-batch the “batch” and refer to its size as the “batch size.”

**Stochastic Gradient Descent**

The SGD algorithm could not be any simpler:

1. Choose a learning rate schedule $\eta_k$.
2. Choose stopping criterion.
3. Choose batch size $m$.
4. Randomly select mini-batch $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$
5. Forward and backpropagation
6. Update $\theta_{k+1} = \theta_k - \eta_k \nabla L(z^{(i)}, y^{(i)})$
7. Repeat 4, 5, 6 until the stopping criterion is satisfied.

Finally, we get to images…
What if we just vectorized images and stuffed these into a MLP?

Too many weights and connections!

- This fully connected hidden layer might have 75 million weights!
- And this is just for a thumbnail image and a two layer net.

Convolutional Neural Networks

Fully Connected (FC) Layer

Every input unit is connected to every output unit.

Fully Connected (FC) Layer

Consider a hidden unit: it connects to all units from the previous layer.

Convolutional Layer: Local Connections

Here the connections are spatially local and governed by the kernel size.
Convolutional Layer: Local Connections

Here the connections are spatially local and governed by the kernel size.

Convolutional Layer: Shared Weights

The weights are shared. They are the same for each position. So, this is like a convolution kernel.

Convolution in 2D

Convolution with 2D Kernel
Convolution with 2D Kernel

Convolutional Layer: Shared Weights

Convolutional Layer: Stride

We can skip input pixels by choosing a \( stride > 1 \).

Convolutional Layer: Stride

We can skip input pixels by choosing a \( stride > 1 \).

Convolutional Layer: Stride

The output dim = (input dim - kernel size) / stride + 1.

ReLU used with ConvNets

- Just like with our fully connected layers, for our convolutional layers we will follow the linear operation (convolution) with a non-linear squashing function.
- Again the fcn to use for now is ReLU.
- But we are not done…there’s one more thing!
Pooling

- We can spatially pool the output from the ReLU to reduce the size of subsequent layers in our network.
- This reduces both computation and the number of parameters that need to be fit and helps prevent overfitting.
- The pooling operation is often the max value in the region, but it could be average, or median, etc.
- The pooling has a stride associated with it that determines the downsampling of the input.

Pooling Layer

The pooling layer pools values in regions of the conv layer.

Convolutional Layer: Shared Weights

The weights for the kernel are shared. They are the same for each position.

Conv Layer

Color images

Conv layer features

Oops. Just one more thing... Recall

Each kernel finds just one type of feature.

If a kernel shares weights then it can only extract one type of feature.

Why not allow for many kernels and many features!
A Convolutional Net

• Let’s assume we have 28x28 grayscale images as input to our conv net. So we will input 28x28x1 samples into the net.
• Let’s fix our kernel size at 5x5 and, to make this simple, pad our images with zeros and use a stride = 1.
• Let’s use max pooling on the output, with a 2x2 pooling region and a stride of 2.
• Let's extract 32 features after the first layer.
• So the output from this layer will be 14x14x32.

A Convolutional Net

• Now let’s make a second layer, also convolutional.
• Let’s fix our kernel size at 5x5, pad our images with zeros and use a stride = 1.
• Let’s use max pooling on the output again, with a 2x2 pooling region and a stride of 2.
• Let’s extract 64 features after the second layer.
• So the output from this layer will be 7x7x64.

A Convolutional Net

• Our third layer will be a fully connected layer mapping our convolutional features to a 1024 dimensional feature space.
• This layer is just like any of the hidden layers you have created before. It is a linear transformation followed by ReLU.
• So the output from this layer will be 1x1x1024.
A Convolutional Net

- Finally, will map this feature space to a 10 class output space and use a softmax with a MLE/cross entropy loss function.
- And...we’re done!

Some Famous Deep Nets and Data sets

AlexNet [Krizhevsky et al., 2012]

VGG16 [Simonyan and Zisserman, 2014]
GoogLeNet [Szegedy et al., 2014]

ResNet [He et al., 2016]

PASCAL VOC

20 categories in 11,530 images with 27,450 ROIs and 6,929 segmentations

[Everingham et al. 2005—2012]

ImageNet

20,000+ categories x ~1000 instances = 14,000

[ Deng et al. 2009]
Classification: ImageNet Challenge top-5 error