Announcements

- HW4 due Wed 3/18 midnight. Maximum one day late. Note. I had said that it would be due on Friday 3/13. Use that as a personal deadline, unless you are slammed.
- Final Exam: Wed 3/18 at 7pm-10pm, in this room
- Discussion of Final during wed class
Introduction

- The sea bass/salmon example
  - State of nature is a random variable, \( \omega \)
    - \( \omega_1 \) - the fish is a salmon
    - \( \omega_2 \) - the fish is a sea bass
  - Prior Probabilities
    - \( P(\omega_1), P(\omega_2) \)
    - \( P(\omega_1) > 0 \)
    - \( P(\omega_1) + P(\omega_2) = 1 \) (exclusivity and exhaustivity)
  - Example prior: bass & salmon are equally likely
    - \( P(\omega_1) = P(\omega_2) = \frac{1}{2} \) (uniform priors)

- Decision rule with only the prior information
  - Decide \( \omega_1 \) if \( P(\omega_1) > P(\omega_2) \) otherwise decide \( \omega_2 \)

- Use of the class-conditional information

- Let \( \mathbf{x} \) be a vector of features

- \( P(\mathbf{x} \mid \omega_1) \) and \( P(\mathbf{x} \mid \omega_2) \) describe the distribution in features (say lightness) between populations of sea-bass and salmon.
Computing the Posterior

• Posterior \( P(\omega_j | x) \), likelihood \( P(x | \omega_j) \), prior \( P(x) \)

\[
P(\omega_j | x) = \frac{P(x | \omega_j)P(\omega_j)}{P(x)} \quad \text{(BAYES RULE)}
\]

where in case of two categories the probability density \( P(x) \) is

\[
P(x) = \sum_{j=1}^{2} P(x | \omega_j)P(\omega_j)
\]

• Given a measurement \( x \), one can evaluate \( P(\omega_j | x) \) for all classes

Pattern Classification.
Maximum a posteriori classifier (MAP)

\[ g_j(x) = P(\omega_j | x) = \frac{P(x | \omega_j)P(\omega_j)}{P(x)} \]

Class conditional density \( P(x | \omega_i) \) : Prior of class \( j \)

\[ \hat{j} = \arg \max_j g_j(x) \]

“Conventional” Vision Technique

The Mark I Perceptron machine was the first implementation of the perceptron algorithm. The machine was connected to a camera that used $20 \times 20$ 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 "704" was fed two cards, one with squares marked on the left side and the other with squares on the right side.

Learns by Doing

In the first fifty trials, the machine made no distinction between them. It then started registering a "Q" for the left squares and "O" for the right squares.

Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a "self-induced change in the wiring diagram."

The first Perceptron will have about 1,000 electronic "association cells" receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.
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?
How powerful is a perceptron?

Inverter

<table>
<thead>
<tr>
<th>input x1</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Boolean AND

<table>
<thead>
<tr>
<th>input x1</th>
<th>input x2</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean OR

<table>
<thead>
<tr>
<th>input x1</th>
<th>input x2</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean XOR

<table>
<thead>
<tr>
<th>input x1</th>
<th>input x2</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Concept Space & Linear Separability

Linear Separability

Linear Separability

Linear Separability

Linear Separability

Eek!
Increasing Expressiveness: Multi-Layer Neural Networks

**Boolean XOR**

<table>
<thead>
<tr>
<th>input x₁</th>
<th>input x₂</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

2-layer Perceptron Net

Any Boolean function can be represented by a two layer network!

Why?
The nodes of multilayered network

\[ y(x; w) = a(w^T x + w_0) \]

- \( x \): input vector
- \( w \): weights
- \( w_0 \): bias term
- \( a \): activation function

Activation Function: Tanh

- As \( x \) goes from \(-\infty\) to \(\infty\), \( \tanh(x) \) goes from \(-1\) to \(1\)
- It has a “sigmoid” or S-like shape
- \( \tanh(0) = 0 \)
### Activation Function

**Rectified Linear Unit (ReLU)**

- **Equation:**
  
  \[ g(z) = \max(0, z) \]

- **Derivative:**
  
  \[ \frac{dg}{dz} = \begin{cases} 
  0, & \text{if } z < 0 \\
  1, & \text{if } z \geq 0 
  \end{cases} \]

- **Let** \( f = \frac{dg}{dz} \)

### Two Layer Network

- **Equation:**
  
  \[ y(x; W) = a_2(W_2 a_1(W_1 x + w_0)) \]

- **Characteristics:**
  
  - Two sets of weights
  - Two activation functions
Two Layer Network

\[ y(x, w) = a_2(W_2a_1(W_1x + w_{1,0}) + w_{2,0}) \]

- Two sets of weights \( W_1 \) and \( W_2 \)
- Two activation functions \( a_1 \) and \( a_2 \)

Feedforward Networks

- These networks are composed of functions represented as "layers"

\[ y(x) = a_i(a_{i-1}(x; w_i); w_{i-1}); w_0 \]

with weights \( w_i \) associated with layer \( i \) and \( a_i \) is the activation function for layer \( i \).

- \( y(x) \) can be a scalar or a vector function.
Classification Networks & Softmax

- To classify input $\mathbf{x}$ into one of $c$ classes, we have $c$ outputs.
- Output $O_i(\mathbf{x})$ can be viewed as an estimate of $p(\omega_i | \mathbf{x})$. That is the posterior probability of the class, given the input. Recognition decision is $\text{arg max}_i O_i(\mathbf{x})$.
- If the network were certain about the class, one output would be 1 and the rest would be zero.
- More generally, $\sum_{i=1}^c p(\omega_i | \mathbf{x}) = 1$, the $c$ outputs must sum to 1.
- This can be implemented with a softmax layer where $z_i$ are the outputs of the previous layer.

$$O_i = \frac{e^{z_i}}{\sum_{j=1}^c e^{z_j}}$$

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 \[\text{[Leshno et al. '93]}, \] or sigmoid or some other “squashing” activation function \[\text{[Hornik et al. '89, Cybenko '89]}\] 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.
High level view of evaluation and training

- **Training data:** \{ \langle \mathbf{x}^{(i)}, \mathbf{y}^{(i)} \rangle : 1 \leq i \leq n \}

- **Network:** \( \mathbf{f}(\mathbf{x}, \mathbf{w}) \)

\[
L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{f}(\mathbf{x}^{(i)}; \mathbf{w}), \mathbf{y}^{(i)})
\]

- **Total Loss:** \( TL(\mathbf{w}) = \sum_{i=1}^{n} L(\mathbf{f}(\mathbf{x}^{(i)}; \mathbf{w}), \mathbf{y}^{(i)}) \)

- **Training:** Find \( \mathbf{w} \) that minimizes the total loss.

---

Training Feed Forward Networks

- **Given a training set** \{\( \langle \mathbf{x}^{(1)}, \mathbf{y}^{(1)} \rangle, \langle \mathbf{x}^{(2)}, \mathbf{y}^{(2)} \rangle, \ldots, \langle \mathbf{x}^{(n)}, \mathbf{y}^{(n)} \rangle \)\}, **estimate** (learn) \( \mathbf{w} \) by making \( TL(\mathbf{w}) = \sum_{i=1}^{n} L(\mathbf{f}(\mathbf{x}^{(i)}; \mathbf{w}), \mathbf{y}^{(i)}) \) small.

- **Back propagation using Stochastic Gradient Descent**

- **Adagrad, RMSprop, ADAM**

- **Regularization:** Dropout, Batch/Group/Instance Normalization

- **Early Stopping**
Gradient-Based Optimization

When $x$ is a vector?

- Use gradient

$$\nabla_x f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

- and make step in opposite direction

$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$
Deep learning optimization is usually expressed as a loss summed over all the training samples.

Our goal is not so much to find the weights that globally 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.

Although there is a large literature on global optimization, gradient descent-based methods are used in practice.

Our optimizations for deep learning are typically done in very high dimensional spaces, where the number of weights 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.
Back propagation

- Basically another name for gradient descent
- Because of nature of network $a_3(a_2(a_1(x;w_1);w_2);w_3)$, gradients with respect to $w_i$ are determined by chain rule
- Can be thought of as “propagating” from loss function to input.
- In homework, you’ll use adaptive step size method called ADAM.

Training and Validation Sets

- Given a bunch of labelled data, divide into Training Set and Validation Set.
- Often 90%-10% or 80%-20% split.
- Often shuffle data before splitting
Training and Validation Sets

Labeled Data

Training Data

Validation Data

NEVER TRAIN ON YOUR VALIDATION SET!

38
Training and Validation Sets

Labeled Data

Training Data

Validation Data

NEVER TRAIN ON YOUR VALIDATION SET!
Regularization

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.

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.
EarlyStopping

More Data / Data Augmentation

• Possibly the best way to prevent overfitting is to get more training data!

• When this isn’t possible, you can often perform data augmentation where training samples are randomly perturbed or jittered to produce more training samples.

• This is usually easy to do when the samples are images and one can crop, rotate, etc. the samples to produce new samples.

• And if the data can be generated synthetically using computer graphics, we can produce an endless supply.
The SGD algorithm could not be any simpler:

1. Choose a learning rate schedule $\eta_t$. 
2. Choose stopping criterion. 
3. Choose batch size $m$. 
4. Randomly select mini-batch $\{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$ 
5. Forward and backpropagation 
6. Update $\theta_{t+1} = \theta_t - \eta_t g$ 
   $$g = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\theta} L(x^{(i)}, y^{(i)})$$ 
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?

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden1</th>
<th>Hidden2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>100x100x3 inputs</td>
<td>50x50 hidden units</td>
<td>25x25 hidden units</td>
<td>output units</td>
</tr>
</tbody>
</table>

- 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

- Every input unit is connected to every output unit.
- \( m \times n \) weights
**Fully Connected (FC) Layer**

- Consider a hidden unit: it connects to all units from the previous layer
- $n$ weights per hidden unit

**Convolutional Layer: Local Connections**

- The connections are spatially local and governed by the kernel size.
Convolutional Layer: Local Connections & Shared Weights

- The connections are spatially local and governed by the kernel size.

Convolutional Layer: Shared Weights

- The connections are spatially local and governed by the kernel size.
- 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

\[
\begin{array}{c}
\begin{array}{ccc}
-1 & -1 & -1 \\
-1 & 8 & -1 \\
-1 & -1 & -1 \\
\end{array}
\end{array}
\ast
\begin{array}{c}
\begin{array}{c}
\text{Image}
\end{array}
\end{array}
= 
\begin{array}{c}
\begin{array}{c}
\text{Output Image}
\end{array}
\end{array}
\]

62

Convolution with 2D Kernel

\[
\begin{array}{c}
\begin{array}{ccc}
1 & 2 & 1 \\
2 & 4 & 2 \\
1 & 2 & 1 \\
\end{array}
\end{array}
\ast
\begin{array}{c}
\begin{array}{c}
\text{Image}
\end{array}
\end{array}
= 
\begin{array}{c}
\begin{array}{c}
\text{Output Image}
\end{array}
\end{array}
\]

63
Convolutional Layer: Shared Weights

Convolutional Layer: Stride

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

The output dim = (input dim - kernel size) / stride + 1.
Convolutional Layer: Padding + Stride

Output dimension = \((\text{input dim} - \text{kernel size} + 2 \times \text{padding}) / \text{stride} + 1\)

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.

- The function to use for now is ReLU.

- But we are not done…there’s one more thing!
• 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.
Oops. Just one more thing…Recall

Convolutional Layer: Shared Weights

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

- Grayscale image
  - 1 channel
- Convolution kernel is 3-D: Goes across image dimensions & across channels
- Size = width x height x # input channels

Multiple input channels

- Color images
  - 3 channels
- Convolution kernel is 3-D: Goes across image dimensions & across channels
- Size = width x height x # input channels
Each kernel finds just one type of feature.

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

Convolution with 2D Kernel
Many kernels yielding many features!

- Convolution kernel is 4-D: For each output channel, kernel goes across input dimensions and channels.
- Size = width x height x # input channels x # output channels.

An example deep convolution network

- Input: 28x28 grayscale image
- Output: 10 classes. One output per class.
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, 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’ve seen 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!