Dan is traveling
Midterms were shown yesterday, and will be shown tomorrow.

Once you leave with your midterms, we don’t take any regrade requests.
Gradient

What is gradient of a function $f(x)$

$$\nabla f(x) = \frac{df}{dx}$$

Rate of change of function $f(x)$ with respect to input

What is gradient of multiple variable function

$$f(x_1, x_2, x_3, \ldots, x_n)$$

$$\nabla f(x_1, x_2, \ldots, x_n) = \left[ \frac{\delta f}{\delta x_1}, \frac{\delta f}{\delta x_2}, \frac{\delta f}{\delta x_3}, \ldots, \frac{\delta f}{\delta x_n} \right]$$

This is a vector. What is the magnitude? What is the direction?

Direction: direction of steepest increase
Magnitude: rate of change in that direction
Gradient Descent

Since gradient vector gives me direction of steepest increase, we should go in the reverse direction to get lower function value.

Update: \( \mathbf{x} \leftarrow \mathbf{x} - R \nabla f(\mathbf{x}) \) where \( \mathbf{x} = [x_1, x_2, \ldots, x_n] \)

For each element, we have

\[
x_i \leftarrow x_i - R \frac{\delta f}{\delta x_i}
\]

Works well when you have convex functions, guaranteed to reach global minima.
Gradient Descent

What happens when function is non-convex?

Ohh no! Algorithm gets stuck in local minima

Till now, you have applied gradient descent on convex functions, today we will apply it on non-convex problem
A different view of LMS

For \(d^{th}\) example

\[
o_d = \sum_i w_i x_i - T
\]

Define error

\[
Err(w) = \sum_d (o_d - t_d)^2
\]

Perform gradient descent updates

\[
w = w - R \nabla Err(w)
\]

Each input \(x_i\) is connected to a unit (blue circle) by weight \(w_i\)

Think of the input to unit from \(i^{th}\) connection to be \(w_i x_i\)

The unit (blue circle) is summing all these inputs and outputting the sum
Expressivity by Depth

However linear threshold units like last slide can only deal with linear functions of input $x$.

We now want to learn **non-linear functions** of the input. You already know some methods to learn non-linear functions of the input. Can you name them?

- Decision trees, kernels

Neural networks another way of doing that, **stack** several layers of threshold elements, each layer using the output of the previous layer as input.
Basic Unit: Neuron

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]

Let's define a variable called \textbf{net}.

\[
\text{net} = \sum_i w_i x_i
\]

Then we have

\[
o = \sigma(\text{net} - T)
\]
Stack them up

\[ X_1 \rightarrow \sigma \circ \sum \rightarrow O_1 \]
\[ X_2 \rightarrow \sigma \circ \sum \rightarrow O_k \]
\[ X_n \rightarrow \sigma \circ \sum \rightarrow \text{Output Layer} \]

Input Layer \hspace{1cm} Hidden Layer

Feedforward Neural Network
Feedforward Neural Network

Let's number all circles / nodes with number ids.

Connection between node i and node j is denoted as $w_{ij}$, net input to node j is $\text{net}_j$, output from node j is $x_j$ or $o_j$.

Connection (yellow arrow) between node 3 and node 7 is denoted by $w_{37}$

Output of node 7 is $x_7$ or $o_7$

Net Input to node 7 is $\text{net}_7$
Feedforward Neural Network

For $k^{th}$ output node

$$o_k = \sigma\left( \sum_{j \in \text{hidden}} w_{jk} \sigma\left( \sum_{i \in \text{input}} w_{ij} x_i - T_j \right) - T_k \right)$$
What if we use identity function (like LMS example) instead of sigmoid function for activation?

Becomes a Linear Model

What if we use step function (like Perceptron) then?

Non-differentiable, not suitable for gradient descent
However there are other non-linear differentiable functions you can use

\[ x \]
\[ x \]
\[ x \]
\[ \text{tanh} \]
\[ \text{relu} \]
We will use squared error

\[ Err(w) = \frac{1}{2} \sum_{d \in D} \sum_{k \in K} (t_{kd} - o_{kd})^2 \]

Here D is the set of training examples, and K is the set of output units.
Remember Gradient Descent

\[ w = w - R \nabla Err(w) \]

In order to compute \( \nabla Err(w) \)

You need to compute partial derivatives w.r.t each element in \( w \) above.

Whats \( w \) here ? What are the parameters of the model you want to train ?

- Connection weights \( w_{ij} \)
- Thresholds at each unit \( T_i \)
Do you remember derivatives?

Function

\[ y = \frac{1}{2} \sum_{k \in K} (c_k - x_k)^2 \]

\[ y = \sum w_i \cdot x_i \]

\[ y = \frac{1}{1 + \exp\{-(x-T)\}} \]

Derivative

\[ \frac{\partial y}{\partial x_i} = -(c_i - x_i) \]

\[ \frac{\partial y}{\partial w_i} = x_i \]

\[ \frac{\partial y}{\partial x} = \frac{\exp\{-(x-T)\}}{(1 + \exp\{-(x-T)\})^2} = y(1 - y) \]
Some facts from real analysis

\[
\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}
\]

\[
\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \frac{\partial y_2}{\partial x}
\]

\[
\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}
\]
Reminder : Neuron

\[ o = \sigma \left( \sum_i w_i x_i - T \right) \]

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

**Diagram:**

- \( X_1 \) to \( X_n \) connected to neuron with weights \( W_1 \) to \( W_n \).
- Neuron output \( o \).

**Text:**

Let's define a variable called **net**

you can think of it as net input to a unit

Then we have

\[ net = \sum_i w_i x_i \]

\[ o = \sigma(net - T) \]
ARE YOU READY

TO COMPUTE SOME PARTIAL DERIVATIVES?
Derivation of Learning Rule

Connection $w_{ij}$ between hidden and output layer (influences output only through $\text{net}_j$)

$$
\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}
$$

$$
= -(t_j - o_j) \cdot o_j(1 - o_j) \cdot x_i
$$

$$
\text{Err}_d(\overline{w}) = \frac{1}{2} \sum_{k \in K} (t_k - o_k)^2
$$

$$
\frac{\partial o_j}{\partial \text{net}_j} = o_j(1 - o_j)
$$

$$
o_j = \frac{1}{1 + \exp\{-\text{net}_j - T_j\}}
$$

$$
\text{net}_j = \sum w_{ij}x_i
$$
Derivation of Learning Rule

Connection $w_{ij}$ between hidden and output layer is updated as

$$\Delta w_{ij} = R(t_j - o_j) o_j (1 - o_j) x_i$$

$$= R \delta_j x_i$$

where

$$\delta_j = (t_j - o_j) o_j (1 - o_j) = - \frac{\delta E_d}{\delta \text{net}_j}$$
Derivation of Learning Rule

Connection $w_{ij}$ between input and hidden layer (shown in orange)

Influences the output only through those output units which are connected to node $j$ (shown connected by yellow arrows to $j$)

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} = \sum_{k \in \text{downstream}(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$
Derivation of Learning Rule

\[
\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} = \sum_{k \in \text{downstream}(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}
\]

\[
= \sum_{k \in \text{downstream}(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i
\]

\[
\delta_j = -\frac{\delta E_d}{\delta \text{net}_j}
\]

\[
\text{net}_j = \sum w_{ij} x_i
\]
Derivation of Learning Rule

\[ \frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}} = \sum_{k \in \text{downstream}(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}} \]

\[ = \sum_{k \in \text{downstream}(j)} -\delta_k \frac{\partial net_k}{\partial net_j} x_i \]

\[ = \sum_{k \in \text{downstream}(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} x_i \]

\[ = \sum_{k \in \text{downstream}(j)} -\delta_k w_{jk} o_j (1 - o_j) x_i \]

**Diagram:**

- **Network Structure:** Showed as nodes and weighted connections.
- **Equation:**
  - \( net_j = \sum w_{ij} x_i \)
  - \( \frac{\partial o_j}{\partial net_j} = o_j (1 - o_j) \)
  - \( o_j = \frac{1}{1 + \exp\{-(net_j - T_j)\}} \)
Connection $w_{ij}$ between input and hidden layer is updated as

$$\Delta w_{ij} = Ro_j (1 - o_j) \left( \sum_{k \in \text{downstream}(j)} \delta_k w_{jk} \right)$$

where

$$\delta_j = o_j (1 - o_j) \left( \sum_{k \in \text{downstream}(j)} \delta_k w_{jk} \right)$$

First determine the error for the output units. Then, back propagate this error layer by layer through the network, changing weights appropriately in each layer.
The Backpropagation Algorithm

For each example in the training set, do:

1. Compute the network output for this example
2. Compute the error term between the output and target values
3. For each output unit $j$, compute error term:
   \[ \delta_j = (t_j - o_j) o_j (1 - o_j) \]
4. For each hidden unit, compute error term
   \[ \delta_j = o_j (1 - o_j) \left( \sum_{k \in \text{downstream}(j)} \delta_k w_{jk} \right) \]
5. Update weights by
   \[ \Delta w_{ij} = R \delta_j x_i \]
A few small things

We computed gradient for a single example. So, you can directly use it if you perform **stochastic gradient descent**

Do you know how to convert this information to gradient descent?

We only showed updated for connection weights. Do you know how to derive updates for the thresholds $T$?

Usually, for neural networks, people use stochastic gradient descent with mini-batches (and not with one example at a time)
Alternatives

You already saw that you can use a lot of other activation functions

You can also use any loss function. The example you saw uses squared loss

Squared loss is actually not that good for neural nets, due to training slowdown.

A better loss function is **cross entropy**

\[
Err(w) = - \sum_{i \in K} (t_i \log(o_i) + (1 - t_i) \log(1 - o_i))
\]

Exercise: Try to compute its back propagation terms
More Hidden Layers

Same algorithm holds for more hidden layers
Deep Learning Libraries

Tensorflow

Torch

Theano

Pytorch

Dynet

They allow automatic differentiation. You can just write the network, get the gradients for free.

Let's look at **Tensorflow** demo
Comments on Training

• No guarantee of convergence, may oscillate or reach a local minima

• In practice, many large networks can be trained on large amounts of data for realistic problems

• Termination criteria: Number of epochs; Threshold on training set error; No decrease in error; Increase error on a validation set

• To avoid local minima: several trials with different random initial weights with majority or voting techniques
Over-fitting Prevention

• Running too many epochs may over-train the network and result in overfitting.

• Keep a held out validation set and test accuracy after each epoch, maintain weights for the best performing network on the validation set, and return it when performance decreases significantly beyond that.

• Too few hidden units can prevent the data from adequately fitting the data, too many hidden units lead to overfitting. You can tune to get the best number of hidden units for your task.

• Another approach to prevent overfitting: Change error function to include a term for the sum of squares of the weights in the network.
Dropout Training

Proposed by Hinton et al 2012

Each time, decide on whether to delete a hidden unit with some probability $p$. 
Dropout Training

Dropout of 50% of hidden units, and 20% of input units
Story Time
Some History

Inspired by biological systems, but don’t take this seriously

**Computation** : McCollough and Pitts (1943) showed how linear threshold units can be used to compute logical functions

**Learning Rules**

Hebb (1949) suggested that if two units are both active (firing) then the weights between them should increase.

Rosenblatt (1959) suggested that when a target output value is provided for a single neuron with fixed input, it can incrementally change weights and learn to produce the output using the Perceptron learning rule.
Rise and Fall and Rise and Fall of Neural Nets

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1959</td>
<td>Rosenblatt’s Perceptron</td>
</tr>
<tr>
<td>1969</td>
<td>Minsky and Papert’s book</td>
</tr>
<tr>
<td>1986</td>
<td>Backpropagation work</td>
</tr>
<tr>
<td>1995</td>
<td>SVM’s and deep nets not being trainable</td>
</tr>
<tr>
<td>2006</td>
<td>Hinton’s deep belief net paper</td>
</tr>
</tbody>
</table>

Deep Learning

Revolution of Depth

Object Recognition Performance on Imagenet
Next Class

We will learn about some deep learning architectures