Outline

1. Review: Neural networks Motivation
2. Review: Perceptron Algorithm
3. Review: Neural Network Architecture
4. The Back-propagation Algorithm
5. Choice of Activation Function
Review: Neural networks
Motivation
Logistic Regression: How to Handle Complex Boundaries?

- This data is not linear separable.
- Use non-linear basis functions to add more features.
Logistic Regression: How to Handle Complex Boundaries?

- This data is not linear separable
- Use non-linear basis functions to add more features
Adding polynomial features

- New feature vector is $\mathbf{x} = [1, x_1, x_2, x_1^2, x_2^2]$
- $\Pr(y = 1 | \mathbf{x}) = \sigma(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2)$
- If $\mathbf{w} = [-1, 0, 0, 1, 1]$, the boundary is $-1 + x_1^2 + x_2^2 = 0$
  - If $-1 + x_1^2 + x_2^2 \geq 0$ declare spam
  - If $-1 + x_1^2 + x_2^2 < 0$ declare ham
Adding polynomial features

- New feature vector is \( x = [1, x_1, x_2, x_1^2, x_2^2] \)
- \( \Pr(y = 1|x) = \sigma(w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2) \)
- If \( w = [-1, 0, 0, 1, 1] \), the boundary is \(-1 + x_1^2 + x_2^2 = 0\)
  - If \(-1 + x_1^2 + x_2^2 \geq 0\) declare spam
  - If \(-1 + x_1^2 + x_2^2 < 0\) declare ham

\[-1 + x_1^2 + x_2^2 = 0\]
But what if we had a large number of features?

Each feature $x_i$ is one pixel in an $100 \times 100$ input image

- Adding polynomial features would result in an enormous $\phi(x)$
But what if we had a large number of features?

Each feature $x_i$ is one pixel in an $100 \times 100$ input image

- Adding polynomial features would result in an enormous $\phi(x)$
- Can we somehow only retain the important features?
But what if we had a large number of features?

Each feature $x_i$ is one pixel in an $100 \times 100$ input image

- Adding polynomial features would result in an enormous $\phi(x)$
- Can we somehow only retain the important features?
- We will need to carefully hand-pick them, which can be hard and tedious
But what if we had a large number of features?

Each feature $x_i$ is one pixel in an $100 \times 100$ input image

- Adding polynomial features would result in an enormous $\phi(x)$
- Can we somehow only retain the important features?
- We will need to carefully hand-pick them, which can be hard and tedious
- Neural networks automate this for us!
Neural Network Compress the Set of Features

- Start with feature vector $\mathbf{x}$ containing all pixels in the image
- Layer 1: distill the edges of the image
- Layer 2: distill triangles, circles, etc.
- Layer 3: recognize pointy ears, fur style etc.
- Layer 4: performs logistic regression on the features in layer 3
Neural Network Compress the Set of Features

- Start with feature vector \( \mathbf{x} \) containing all pixels in the image
- Layer 1: distill the edges of the image
- Layer 2: distill triangles, circles, etc.
- Layer 3: recognize pointy ears, fur style etc.
- Layer 4: performs logistic regression on the features in layer 3

We cannot directly control what each layer learns; this depends on the training data
Inspiration from Biology: How does our brain work?

Each feature $x_i$ is one pixel in an $100 \times 100$ input image

- Humans easily perform such complex image or speech recognition tasks
- We cannot exactly describe a set of rules by which we distinguish cats vs. dogs, but we almost always know the correct answers when a new image is presented to us
- How do our brains learn these complex tasks?
Neurons in the Brain

- Each neuron is a non-linear computing unit
- It collects input signals from neighboring neurons
- Output of its computation is transmitted through the axon
- Other neurons use this output as the input signal

An average human brain has \( \sim 100 \) billion neurons!
• Different parts of our brain are specialized for different tasks
• But they can be re-wired to adapt to different inputs
• The auditory cortex can be taught to see and the visual cortex can be taught to listen!
• Based on these observations, scientists concluded that all the neurons are structurally homogeneous
Based on the biological insights, a mathematical model for an ’artificial’ neuron was developed

- Each input $x_i$ is multiplied by weight $w_i$
- Add a +1 input neuron which is multiplied by the bias $b$
- Apply a non-linear function $g$ to the weighted combination of the inputs, $\mathbf{w}^T \mathbf{x} + b$
- Different candidates for $g$: heaviside function, sigmoid, tanh, rectified linear unit, etc.

![Single Artificial Neuron Diagram]
1. Review: Neural networks Motivation

2. Review: Perceptron Algorithm

3. Review: Neural Network Architecture

4. The Back-propagation Algorithm

5. Choice of Activation Function
Review: Perceptron Algorithm
• The perception is a single-unit neural network with the heavyside activation function or $\text{sign}(x)$
Perceptron: Rosenblatt (1957)

- The perception is a single-unit neural network with the heavyside activation function or $\text{sign}(x)$.
- It considers a linear binary classification problem to distinguish between two classes $\{-1, +1\}$.

$$g(w^T x + b), \text{Linear comb. of features}$$
The perception is a single-unit neural network with the heavyside activation function or $\text{sign}(x)$.

It considers a linear binary classification problem to distinguish between two classes $\{-1, +1\}$.

Assign label $\text{sign}(\mathbf{w}^T \mathbf{x} + b)$ to a new sample.
• The perception is a single-unit neural network with the heavyside activation function or \( \text{sign}(x) \)

• It considers a linear binary classification problem to distinguish between two classes \( \{-1, +1\} \).

\[
\begin{align*}
\mathbf{x}^T \mathbf{w} + b, \text{ Linear comb. of features} \\
g(\mathbf{w}^T \mathbf{x} + b)
\end{align*}
\]

Perceptron

• Assign label \( \text{sign}(\mathbf{w}^T \mathbf{x} + b) \) to a new sample

• Notation change: Merge \( b \) into the vector \( \mathbf{w} \) and append 1 to the vector \( \mathbf{x} \)
How to learn the weights \( w \)?

The objective is to learn \( w \) that minimizes the number of errors on the training dataset. That is, minimize

\[
\varepsilon = \sum_{n} \mathbb{1}[y_n \neq \text{sign}(w^T x_n)]
\]

**Algorithm**: For a randomly chosen data point \((x_n, y_n)\) make small changes to \( w \) so that

\[
y_n = \text{sign}(w^T x_n)
\]
How to learn the weights $w$?

The objective is to learn $w$ that minimizes the number of errors on the training dataset. That is, minimize

$$
\varepsilon = \sum_n \mathbb{1}[y_n \neq \text{sign}(w^\top x_n)]
$$

**Algorithm**: For a randomly chosen data point $(x_n, y_n)$ make small changes to $w$ so that

$$
y_n = \text{sign}(w^\top x_n)
$$

**Two cases**

- If $y_n = \text{sign}(w^\top x_n)$, do nothing.
- If $y_n \neq \text{sign}(w^\top x_n)$,

$$
\begin{align*}
    w^{\text{NEW}} &\leftarrow w^{\text{OLD}} + y_n x_n
\end{align*}
$$
Why would it work?

If \( y_n \neq \text{sign}(\mathbf{w}^\top \mathbf{x}_n) \), then

\[
y_n(\mathbf{w}^\top \mathbf{x}_n) < 0
\]
Why would it work?

If \( y_n \neq \text{sign}(\mathbf{w}^\top \mathbf{x}_n) \), then

\[
y_n(\mathbf{w}^\top \mathbf{x}_n) < 0
\]

What would happen if we change to new \( \mathbf{w}^{\text{NEW}} = \mathbf{w} + y_n \mathbf{x}_n \)?

\[
y_n[(\mathbf{w} + y_n \mathbf{x}_n)^\top \mathbf{x}_n] = y_n \mathbf{w}^\top \mathbf{x}_n + y_n^2 \mathbf{x}_n^\top \mathbf{x}_n
\]
If \( y_n \neq \text{sign}(w^\top x_n) \), then

\[
y_n(w^\top x_n) < 0
\]

What would happen if we change to new \( w^{\text{NEW}} = w + y_n x_n \)?

\[
y_n[(w + y_n x_n)^\top x_n] = y_n w^\top x_n + y_n^2 x_n^\top x_n
\]

We are adding a positive number, so it is possible that

\[
y_n(w^{\text{NEW}}^\top x_n) > 0
\]

i.e., we are more likely to classify correctly
Iteratively solving one case at a time

- REPEAT
- Pick a data point $x_n$ (can be a fixed order of the training instances)
- Make a prediction $y = \text{sign}(w^\top x_n)$ using the current $w$. 
Iteratively solving one case at a time

- REPEAT
- Pick a data point $x_n$ (can be a fixed order of the training instances)
- Make a prediction $y = \text{sign}(w^T x_n)$ using the current $w$.
- If $y = y_n$, do nothing.
  Else,
    $$w \leftarrow w + y_n x_n.$$ 
- UNTIL converged.
Properties

• This is an online algorithm (works when data is arriving sequentially as a stream)
• If the training data is linearly separable, the algorithm stops in a finite number of steps.
• The parameter vector is always a linear combination of training instances (requires initialization of $w_0 = 0$).
• We don’t need to set a learning rate

The perceptron algorithm was used in old times to train $w$ by hand, without a computer.
1. Review: Neural networks Motivation

2. Review: Perceptron Algorithm

3. Review: Neural Network Architecture

4. The Back-propagation Algorithm

5. Choice of Activation Function
Review: Neural Network Architecture
• Suppose $g$ is the sigmoid function $\sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}$

• We can find a linear decision boundary separating two classes. The output is the probability of $x$ belonging to class 1.

Neuron with Sigmoid activation
Binary Logistic Regression

- Suppose $g$ is the sigmoid function $\sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}$
- We can find a linear decision boundary separating two classes. The output is the probability of $x$ belonging to class 1.
- This is binary logistic regression, which we already know.
Learning weights $w$ using SGD

Cross-entropy Error Function

$$E(w) = -\sum_n \{y_n \log \sigma(w^\top x_n) + (1 - y_n) \log [1 - \sigma(w^\top x_n)]\}$$
Learning weights $w$ using SGD

Cross-entropy Error Function

$$\mathcal{E}(w) = - \sum_{n} \{y_n \log \sigma(w^\top x_n) + (1 - y_n) \log [1 - \sigma(w^\top x_n)]\}$$

Gradients

$$\frac{\partial \mathcal{E}(w)}{\partial w} = \sum_{n} \{\sigma(w^\top x_n) - y_n\} x_n$$

Gradient descent for logistic regression

- Choose a proper step size $\eta > 0$
- Iteratively update the parameters following the negative gradient to minimize the error function

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \sum_{n} \{\sigma(w^\top x_n) - y_n\} x_n$$
Multi-class Regression

- If the target is to take $C$ possible values
- If $y$ belongs to the first class, the outputs should be $[1, 0, \ldots, 0]$
- Need to produce a vector $\hat{y}$ with $\hat{y}_i = p(y = i|x)$
- Linear output layer ($g(x) = x$) first produces un-normalized log probabilities:

$$z = w^T x + b$$

Multiclass Regression for $C = 3$
Multi-class Regression

- Softmax:
  \[ \text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \]

- Log of the softmax (since we wish to maximize \( p(y = i; z) \)):
  \[ \log \text{softmax}(z)_i = z_i - \log \sum_j \exp(z_j) \]

Multiclass Regression for \( C = 3 \)
Multi-layer Neural Network

- \( w_{ij} \): weights connecting node \( i \) in layer \((\ell - 1)\) to node \( j \) in layer \( \ell \).

Nodes in the hidden layer

Apply \( g_j \) to \( u_j \)

\[ y_j = g_j(u_j), \quad z_k = g_k(u_k) \]: output/activation of nodes \( j \) and \( k \).

\[ t_k \]: target value for node \( k \) in the output layer.
Multi-layer Neural Network

- \( w_{ij} \): weights connecting node \( i \) in layer \((\ell - 1)\) to node \( j \) in layer \( \ell \).
- \( b_j, b_k \): bias for nodes \( j \) and \( k \).
- \( u_j, u_k \): inputs to nodes \( j \) and \( k \) (where \( u_j = b_j + \sum_i x_i w_{ij} \)).
Multi-layer Neural Network

- $w_{ij}$: weights connecting node $i$ in layer $(\ell - 1)$ to node $j$ in layer $\ell$.
- $b_j, b_k$: bias for nodes $j$ and $k$.
- $u_j, u_k$: inputs to nodes $j$ and $k$ (where $u_j = b_j + \sum_i x_i w_{ij}$).
- $g_j, g_k$: activation function for node $j$ (applied to $u_j$) and node $k$.
- $y_j = g_j(u_j), z_k = g_k(u_k)$: output/activation of nodes $j$ and $k$. 

Apply $g_j$ to $u_j$ 

Nodes in the hidden layer

$y_j$
Multi-layer Neural Network

- $w_{ij}$: **weights** connecting node $i$ in layer $(\ell - 1)$ to node $j$ in layer $\ell$.
- $b_j$, $b_k$: **bias** for nodes $j$ and $k$.
- $u_j$, $u_k$: **inputs to nodes $j$ and $k** (where $u_j = b_j + \sum_i x_i w_{ij}$).
- $g_j$, $g_k$: **activation function** for node $j$ (applied to $u_j$) and node $k$.
- $y_j = g_j(u_j)$, $z_k = g_k(u_k)$: **output/activation** of nodes $j$ and $k$.
- $t_k$: **target value** for node $k$ in the output layer.
Expressing outputs $z$ in terms of inputs $x$ is called forward-propagation. This is the operation that is performed when doing inference with a trained neural network.

Exercise: Perform forward propagation for the 1-hidden layer neural network shown above. Assume that we are using the sigmoid activation function.
Exercise: Forward-Propagation

- Outputs of the hidden layer are

\[ z = \sigma(0.5x + 1) + 0.5\sigma(x + 0.5) + 0.25 \]
Exercise: Forward-Propagation

- Outputs of the hidden layer are $\sigma(0.5x + 1)$ and $\sigma(x + 0.5)$
- Input to the last layer is
Exercise: Forward-Propagation

- Outputs of the hidden layer are $\sigma(0.5x + 1)$ and $\sigma(x + 0.5)$
- Input to the last layer is $0.5\sigma(0.5x + 1) + 0.5\sigma(x + 0.5) + 0.25$
- $z = \sigma(0.5\sigma(0.5x + 1) + 0.5\sigma(x + 0.5) + 0.25)$
Neural Networks are Very Powerful

- With enough neurons and layers we can represent very complex input-output relationships
- Can be used for regression, classification, embedding, and many other ML applications
1. Review: Neural networks Motivation

2. Review: Perceptron Algorithm

3. Review: Neural Network Architecture

4. The Back-propagation Algorithm

5. Choice of Activation Function
The Back-propagation Algorithm
How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (today’s class)
    \[ \min_n \sum (f(x_n) - t_n)^2 \]
  - Classification: cross-entropy loss (in the homework)
    \[ \min - \sum_n \sum_k t_{nk} \log f_k(x_n) + (1 - t_{nk}) \log (1 - f_k(x_n)) \]
- Very hard optimization problem
  - Stochastic gradient descent is commonly used
  - Many optimization tricks are applied
Stochastic gradient descent

- Randomly pick a data point \((x_n, t_n)\)
- Compute the gradient using only this data point, for example,

\[
g = \frac{\partial[f(x_n) - t_n]^2}{\partial w}
\]

- Update the parameter

\[
w \leftarrow w - \eta g
\]
- Iterate the process until some stop criteria
Back-propagate the error. Given parameters $w, b$:

- Step 1: **Forward-propagate to find** $z_k$ in terms of the input (the “feed-forward signals”).
Updating the parameter values

Back-propagate the error. Given parameters $w, b$:

- Step 1: **Forward-propagate to find** $z_k$ in terms of the input (the “feed-forward signals”).
- Step 2: **Calculate output error** $E$ by comparing the predicted output $z_k$ to its true value $t_k$. 
Updating the parameter values

Back-propagate the error. Given parameters $w, b$:

- **Step 1:** *Forward-propagate to find* $z_k$ *in terms of the input (the “feed-forward signals”).*
- **Step 2:** *Calculate output error* $E$ *by comparing the predicted output* $z_k$ *to its true value* $t_k$.
- **Step 3:** *Back-propagate* $E$ *by weighting it by the gradients of the associated activation functions and the weights in previous layers.*
Back-propagate the error. Given parameters $w, b$:

- **Step 1**: **Forward-propagate to find** $z_k$ in terms of the input (the “feed-forward signals”).
- **Step 2**: **Calculate output error** $E$ by comparing the predicted output $z_k$ to its true value $t_k$.
- **Step 3**: **Back-propagate** $E$ by weighting it by the gradients of the associated activation functions and the weights in previous layers.
- **Step 4**: **Calculate the gradients** $\frac{\partial E}{\partial w}$ and $\frac{\partial E}{\partial b}$ for the parameters $w, b$ at each layer based on the backpropagated error signal and the feedforward signals from the inputs.

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}, \quad b \leftarrow b - \eta \frac{\partial E}{\partial b}$$
Back-propagate the error. Given parameters $w, b$:

- **Step 1**: **Forward-propagate to find** $z_k$ in terms of the input (the “feed-forward signals”).
- **Step 2**: **Calculate output error** $E$ by comparing the predicted output $z_k$ to its true value $t_k$.
- **Step 3**: **Back-propagate** $E$ by weighting it by the gradients of the associated activation functions and the weights in previous layers.
- **Step 4**: **Calculate the gradients** $\frac{\partial E}{\partial w}$ and $\frac{\partial E}{\partial b}$ for the parameters $w, b$ at each layer based on the backpropagated error signal and the feedforward signals from the inputs.
- **Step 5**: **Update the parameters** using the calculated gradients $w \leftarrow w - \eta \frac{\partial E}{\partial w}$, $b \leftarrow b - \eta \frac{\partial E}{\partial b}$ where $\eta$ is the step size.
Back-propagation Example

- $w_{ij}$: weights connecting node $i$ in layer $(\ell - 1)$ to node $j$ in layer $\ell$.
- $b_j$, $b_k$: bias for nodes $j$ and $k$.
- $u_j$, $u_k$: inputs to nodes $j$ and $k$ (where $u_j = b_j + \sum_i x_i w_{ij}$).
- $g_j$, $g_k$: activation function for node $j$ (applied to $u_j$) and node $k$.
- $y_j = g_j(u_j)$, $z_k = g_k(u_k)$: output/activation of nodes $j$ and $k$.
- $t_k$: target value for node $k$ in the output layer.
Back-propagation example (steps 1 and 2)

- Step 1: **Network output** for each \( k \).

\[ z_k = g_k(u_k) \]
Back-propagation example (steps 1 and 2)

- Step 1: **Network output** for each $k$.

$$z_k = g_k(u_k) = g_k(b_k + \sum_j y_j w_{jk})$$

- Step 2: Find the error. Let's assume that the error function is the sum of the squared differences between the target values $t_k$ and the network output $z_k$:

$$E = \frac{1}{2} \sum_{k \in K} (z_k - t_k)^2.$$
Back-propagation example (steps 1 and 2)

- Step 1: **Network output** for each $k$.

$$z_k = g_k(u_k) = g_k(b_k + \sum_j y_j w_{jk}) = g_k(b_k + \sum_j g_j(b_j + \sum_i x_i w_{ij}) w_{jk})$$

- Step 2: Find the error. Let's assume that the error function is the sum of the squared differences between the target values $t_k$ and the network output $z_k$:

$$E = \frac{1}{2} \sum_k (z_k - t_k)^2.$$
Back-propagation example (steps 1 and 2)

- **Step 1:** Network output for each $k$.

  
  \[ z_k = g_k(u_k) = g_k(b_k + \sum_j y_j w_{jk}) = g_k(b_k + \sum_j g_j(b_j + \sum_i x_i w_{ij}) w_{jk}) \]

- **Step 2:** Find the error. Let’s assume that the error function is the sum of the squared differences between the target values $t_k$ and the network output $z_k$: 

  \[ E = \frac{1}{2} \sum_{k \in K} (z_k - t_k)^2. \]
Step 3: **Backpropagate the error.** Let's start at the output layer with weight $w_{jk}$, recalling that $E = \frac{1}{2} \sum_{k \in K} (z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$.
Step 3: **Backpropagate the error.** Let's start at the output layer with weight $w_{jk}$, recalling that $E = \frac{1}{2} \sum_{k \in K} (z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial u_k} \frac{\partial u_k}{\partial w_{jk}}$$
Step 3: **Backpropagate the error.** Let's start at the output layer with weight $w_{jk}$, recalling that $E = \frac{1}{2} \sum_{k \in K} (z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial u_k} \frac{\partial u_k}{\partial w_{jk}} = (z_k - t_k) \frac{\partial z_k}{\partial u_k} \frac{\partial u_k}{\partial w_{jk}}$$
Step 3: **Backpropagate the error.** Let's start at the output layer with weight $w_{jk}$, recalling that $E = \frac{1}{2} \sum_{k \in K} (z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk}y_j$:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial u_k} \frac{\partial u_k}{\partial w_{jk}} = (z_k - t_k) \frac{\partial z_k}{\partial u_k} \frac{\partial u_k}{\partial w_{jk}}$$

$$= (z_k - t_k)g_k'(u_k) \frac{\partial u_k}{\partial w_{jk}}$$
Step 3: **Backpropagate the error.** Let’s start at the output layer with weight $w_{jk}$, recalling that $E = \frac{1}{2} \sum_{k \in K} (z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial u_k} \frac{\partial u_k}{\partial w_{jk}} = (z_k - t_k) \frac{\partial z_k}{\partial u_k} \frac{\partial u_k}{\partial w_{jk}}$$

$$= (z_k - t_k) g'_k(u_k) \frac{\partial u_k}{\partial w_{jk}} = (z_k - t_k) g'_k(u_k) y_j$$
Step 3: **Backpropagate the error.** Let’s start at the output layer with weight $w_{jk}$, recalling that $E = \frac{1}{2} \sum_{k \in K} (z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial u_k} \frac{\partial u_k}{\partial w_{jk}} = (z_k - t_k) \frac{\partial z_k}{\partial u_k} \frac{\partial u_k}{\partial w_{jk}}$$

$$= (z_k - t_k) g'_k(u_k) \frac{\partial u_k}{\partial w_{jk}} = (z_k - t_k) g'_k(u_k) y_j = \delta_k y_j$$

where $\delta_k = (z_k - t_k) g'_k(u_k)$, which we refer to as the error in $u_k$. 

Nodes in the output layer

Apply $g_k$ to $u_k$
Step 3 (cont’d): Now let’s consider \( w_{ij} \) in the hidden layer, recalling
\[
    u_j = b_i + \sum_i x_i w_{ij}, \quad u_k = b_k + \sum_j g_j(u_j) w_{jk}, \quad z_k = g_k(u_k):
\]
\[
    \frac{\partial E}{\partial w_{ij}} = \sum_{k \in K} \frac{\partial E}{\partial u_k} \frac{\partial u_k}{\partial y_j} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}}
\]
Step 3 (cont’d): Now let’s consider $w_{ij}$ in the hidden layer, recalling $u_j = b_i + \sum_i x_i w_{ij}$, $u_k = b_k + \sum_j g_j(u_j) w_{jk}$, $z_k = g_k(u_k)$:

$$\frac{\partial E}{\partial w_{ij}} = \sum_{k \in K} \frac{\partial E}{\partial u_k} \frac{\partial u_k}{\partial y_j} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}} = \sum_{k \in K} \delta_k w_{jk} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}}$$
Step 3 (cont’d): Now let’s consider $w_{ij}$ in the hidden layer, recalling $u_j = b_i + \sum_i x_i w_{ij}$, $u_k = b_k + \sum_j g_j(u_j) w_{jk}$, $z_k = g_k(u_k)$:

$$\frac{\partial E}{\partial w_{ij}} = \sum_{k \in K} \frac{\partial E}{\partial u_k} \frac{\partial u_k}{\partial y_j} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}} = \sum_{k \in K} \delta_k w_{jk} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}}$$

$$= \sum_{k \in K} \delta_k w_{jk} g_j'(u_j) x_i$$
Step 3 (cont’d): Now let’s consider $w_{ij}$ in the hidden layer, recalling

$$u_j = b_i + \sum_i x_i w_{ij}, \quad u_k = b_k + \sum_j g_j(u_j) w_{jk}, \quad z_k = g_k(u_k):$$

$$\frac{\partial E}{\partial w_{ij}} = \sum_{k \in K} \frac{\partial E}{\partial u_k} \frac{\partial u_k}{\partial y_j} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}} = \sum_{k \in K} \delta_k w_{jk} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}}$$

$$= \sum_{k \in K} \delta_k w_{jk} g_j'(u_j) x_i = \delta_j x_i$$

where $\delta_j = g_j'(u_j) \sum_{k \in K} \delta_k w_{jk}$, the error in $u_j$. 

Nodes in the hidden layer
• Step 3 (cont’d): We similarly find that \( \frac{\partial E}{\partial b_k} = \delta_k \), \( \frac{\partial E}{\partial b_j} = \delta_j \).
• Step 3 (cont’d): We similarly find that $\frac{\partial E}{\partial b_k} = \delta_k$, $\frac{\partial E}{\partial b_j} = \delta_j$.
• Step 4: **Calculate the gradients.** We have found that

$$\frac{\partial E}{\partial w_{ij}} = \delta_j x_i \text{ and } \frac{\partial E}{\partial w_{jk}} = \delta_k y_j.$$ 

where $\delta_k = (z_k - t_k)g'_k(u_k)$ and 
$\delta_j = g'_j(u_j) \sum_{k \in K} \delta_k w_{jk}$. 

Apply $g_j$ to $u_j$

Nodes in the hidden layer
Back-propagation example (steps 3 and 4)

- Step 3 (cont’d): We similarly find that $\frac{\partial E}{\partial b_k} = \delta_k$, $\frac{\partial E}{\partial b_j} = \delta_j$.
- Step 4: **Calculate the gradients.** We have found that

$$\frac{\partial E}{\partial w_{ij}} = \delta_j x_i \quad \text{and} \quad \frac{\partial E}{\partial w_{jk}} = \delta_k y_j.$$ 

where $\delta_k = (z_k - t_k)g'_k(u_k)$ and $\delta_j = g'_j(u_j) \sum_{k \in K} \delta_k w_{jk}$. 

Nodes in the hidden layer

Apply $g_j$ to $u_j$
Step 4: **Calculate the gradients.** We have found that

\[
\frac{\partial E}{\partial w_{ij}} = \delta_j x_i \quad \text{and} \quad \frac{\partial E}{\partial w_{jk}} = \delta_k y_j.
\]
• **Step 4: Calculate the gradients.** We have found that

\[
\frac{\partial E}{\partial w_{ij}} = \delta_j x_i \quad \text{and} \quad \frac{\partial E}{\partial w_{jk}} = \delta_k y_j.
\]

• **Step 5: Update Parameters.**

\[
w_{jk} \leftarrow w_{jk} - \eta \frac{\partial E}{\partial w_{jk}}; \quad \text{and} \quad w_{ij} \leftarrow w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}
\]

Similarly update \(b_j\) and \(b_k\).
Final Layer

- Error in each of its outputs is $z_k - t_k$. 

Nodes in the hidden layer

Apply $g_j$ to $u_j$
**Final Layer**

- Error in each of its outputs is $z_k - t_k$.
- Error in input $u_k$ to the final layer is $\delta_k = g'_k(u_k)(z_k - t_k)$
High-level Procedure: Can be Used with More Hidden Layers

Nodes in the hidden layer

**Final Layer**

- Error in each of its outputs is $z_k - t_k$.
- Error in input $u_k$ to the final layer is $\delta_k = g'_k(u_k)(z_k - t_k)$
High-level Procedure: Can be Used with More Hidden Layers

Final Layer
- Error in each of its outputs is $z_k - t_k$.
- Error in input $u_k$ to the final layer is $\delta_k = g'_k(u_k)(z_k - t_k)$

Hidden Layer
- Error in output $y_j$ is $\sum_{k \in K} \delta_k w_{jk}$.
High-level Procedure: Can be Used with More Hidden Layers

Final Layer

- Error in each of its outputs is $z_k - t_k$.
- Error in input $u_k$ to the final layer is $\delta_k = g'_k(u_k)(z_k - t_k)$

Hidden Layer

- Error in output $y_j$ is $\sum_{k \in K} \delta_k w_{jk}$.
- Error in the input $u_j$ of the hidden layer is $\delta_j = g'_j(u_j) \sum_{k \in K} \delta_k w_{jk}$
High-level Procedure: Can be Used with More Hidden Layers

Final Layer

- Error in each of its outputs is $z_k - t_k$.
- Error in input $u_k$ to the final layer is $\delta_k = g'_k(u_k)(z_k - t_k)$

Hidden Layer

- Error in output $y_j$ is $\sum_{k \in K} \delta_k w_{jk}$.
- Error in the input $u_j$ of the hidden layer is $\delta_j = g'_j(u_j) \sum_{k \in K} \delta_k w_{jk}$
**High-level Procedure: Can be Used with More Hidden Layers**

**Final Layer**
- Error in each of its outputs is $z_k - t_k$.
- Error in input $u_k$ to the final layer is $\delta_k = g'_k(u_k)(z_k - t_k)$

**Hidden Layer**
- Error in output $y_j$ is $\sum_{k \in K} \delta_k w_{jk}$.
- Error in the input $u_j$ of the hidden layer is $\delta_j = g'_j(u_j) \sum_{k \in K} \delta_k w_{jk}$

The gradient w.r.t. $w_{ij}$ is $x_i \delta_j$. 
Suppose the output $z = 0.9$ but the target is 1. Perform backpropagation and compute the gradient of error w.r.t. the weight connecting $x$ and $y_2$. 
Suppose the output $z = 0.9$ but the target is 1. Perform backpropagation and compute the gradient of error w.r.t. the weight connecting $x$ and $y_2$.

**Forward-propagation**

- $y_1 = \sigma(0.5x + 1)$ and $y_2 = \sigma(x + 0.5)$
- Input to last layer $u = 0.5y_1 + 0.5y_2 + 0.25$
- Final Output $z = \sigma(u)$
Final layer

- Error in output $z$ is $0.9 - 1 = -0.1$
Exercise: Back-Propagation

Final layer

- Error in output $z$ is $0.9 - 1 = -0.1$
- Error in input $u$ is $-0.1 \times \sigma'(u)$
Exercise: Back-Propagation

The gradient w.r.t. the weight connecting $x$ and $y_2$ is $-0.1 \times \sigma'(u_2) \times 0.5 \times \sigma'(u_2) \times x$. 

Error in $y_1$ is $-0.1 \times \sigma'(u)$ \times 0.5. 

Error in $y_2$ is $-0.1 \times \sigma'(u) \times 0.5$. 

Error in $u_2$ is $-0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2) \times 0.25$. 

Hidden Layer
Exercise: Back-Propagation

Hidden Layer

- Error in $y_1$ is $-0.1 \times \sigma'(u) \times 0.5$
Exercise: Back-Propagation

Hidden Layer

- Error in $y_1$ is $-0.1 \times \sigma'(u) \times 0.5$
- Error in $y_2$ is $-0.1 \times \sigma'(u) \times 0.5$
Hidden Layer

- Error in $y_1$ is $-0.1 \times \sigma'(u) \times 0.5$
- Error in $y_2$ is $-0.1 \times \sigma'(u) \times 0.5$
- Error in $u_2$ is $-0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2)$
Exercise: Back-Propagation

Hidden Layer

- Error in $y_1$ is $-0.1 \times \sigma'(u) \times 0.5$
- Error in $y_2$ is $-0.1 \times \sigma'(u) \times 0.5$
- Error in $u_2$ is $-0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2)$
Hidden Layer

- Error in $y_1$ is $-0.1 \times \sigma'(u) \times 0.5$
- Error in $y_2$ is $-0.1 \times \sigma'(u) \times 0.5$
- Error in $u_2$ is $-0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2)$

The gradient w.r.t. the weight connecting $x$ and $y_2$ is $-0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2) \times x$
Vectorized Implementation

Forward-Propagation

- Represent the weights between layers $l - 1$ and $l$ as a matrix $W^{(l)}$
Forward-Propagation

- Represent the weights between layers $l-1$ and $l$ as a matrix $W^{(l)}$.
- Outputs of layer $l-1$ are in a row vector $y^{(l-1)}$. Then we have $u^{(l)} = y^{(l-1)}W^{(l)}$. 

Back-Propagation

- For each layer $l$ find $\Delta^{(l)}$, the vector of errors in $u^{(l)}$ in terms of the final error.
- Update weights $W^{(l)}$ using $\Delta^{(l)}$.
- Recursively find $\Delta^{(l-1)}$ in terms of $\Delta^{(l)}$. 

Vectorized Implementation
Vectorized Implementation

Forward-Propagation

- Represent the weights between layers $l - 1$ and $l$ as a matrix $W^{(l)}$.
- Outputs of layer $l - 1$ are in a row vector $y^{(l-1)}$. Then we have $u^{(l)} = y^{(l-1)}W^{(l)}$.
- Outputs of layer $l$ are in the row vector $y^{(l)} = g(u^{(l)})$. 

Back-Propagation

- For each layer $l$ find $\Delta^{(l)}$, the vector of errors in $u^{(l)}$ in terms of the final error.
- Update weights $W^{(l)}$ using $\Delta^{(l)}$.
- Recursively find $\Delta^{(l-1)}$ in terms $\Delta^{(l)}$. 
Vectorized Implementation

Forward-Propagation

- Represent the weights between layers \(l - 1\) and \(l\) as a matrix \(W^{(l)}\).
- Outputs of layer \(l - 1\) are in a row vector \(y^{(l-1)}\). Then we have \(u^{(l)} = y^{(l-1)}W^{(l)}\).
- Outputs of layer \(l\) are in the row vector \(y^{(l)} = g(u^{(l)})\).
Vectorized Implementation

**Forward-Propagation**

- Represent the weights between layers $l - 1$ and $l$ as a matrix $W^{(l)}$.
- Outputs of layer $l - 1$ are in a row vector $y^{(l-1)}$. Then we have $u^{(l)} = y^{(l-1)}W^{(l)}$.
- Outputs of layer $l$ are in the row vector $y^{(l)} = g(u^{(l)})$.

**Back-Propagation**

- For each layer $l$ find $\Delta^{(l)}$, the vector of errors in $u^{(l)}$ in terms of the final error.
Vectorized Implementation

**Forward-Propagation**

- Represent the weights between layers $l - 1$ and $l$ as a matrix $W^{(l)}$.
- Outputs of layer $l - 1$ are in a row vector $y^{(l-1)}$. Then we have $u^{(l)} = y^{(l-1)}W^{(l)}$.
- Outputs of layer $l$ are in the row vector $y^{(l)} = g(u^{(l)})$.

**Back-Propagation**

- For each layer $l$ find $\Delta^{(l)}$, the vector of errors in $u^{(l)}$ in terms of the final error.
- Update weights $W^{(l)}$ using $\Delta^{(l)}$. 
Vectorized Implementation

**Forward-Propagation**

- Represent the weights between layers \( l - 1 \) and \( l \) as a matrix \( W^{(l)} \)
- Outputs of layer \( l - 1 \) are in a row vector \( y^{(l-1)} \). Then we have \( u^{(l)} = y^{(l-1)}W^{(l)} \).
- Outputs of layer \( l \) are in the row vector \( y^{(l)} = g(u^{(l)}) \).

**Back-Propagation**

- For each layer \( l \) find \( \Delta^{(l)} \), the vector of errors in \( u^{(l)} \) in terms of the final error
- Update weights \( W^{(l)} \) using \( \Delta^{(l)} \)
- Recursively find \( \Delta^{(l-1)} \) in terms \( \Delta^{(l)} \)
Choice of Activation Function
Sigmoid Units

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

- Squashing type non-linearity: pushes output to range \([0,1]\)
Sigmoid Units

- Problem: Saturate across most of their domain, strongly sensitive only when $z$ is closer to zero
- Saturation makes gradient based learning difficult
**Tanh Units**

\[ \text{tanh}(z) = \frac{1 - e^{-2z}}{1 + e^{-2z}} \]

- Related to sigmoid: \[ \text{tanh}(z) = 2\sigma(2z) - 1 \]
- **Positive:** Squashes output to range \([-1,1]\), outputs are zero-centered
- **Negative:** Also saturates
Hard Tanh and Hard Sigmoid

- To avoid the problem of vanishing gradients we can use piece-wise linear approximations to these functions.
- This significantly reduces the computation complexity because gradients can take only a few values.
Rectified Linear Units

- Approximates the softplus function which is $\log(1 + e^z)$
- ReLu Activation function is $g(z) = \max(0, z)$ with $z \in \mathbb{R}$
- Similar to linear units. Easy to optimize!
- Give large and consistent gradients when active
• Not everywhere differentiable. Is this a problem?
  • In practice not a problem. Return one sided derivatives at $z = 0$
  • Gradient based optimization is subject to numerical error anyway
Rectified Linear Units

- **Positives:**
  - Gives large and *consistent* gradients (does not saturate) when active
  - Efficient to optimize, converges much faster than sigmoid or tanh

- **Negatives:**
  - Non zero centered output
  - Units “die i.e when inactive they will never update
Generalized Rectified Linear Units

- Get a non-zero slope when $z_i < 0$
- $g(z, a)_i = \max(0, z_i) + a_i \min(0, z_i)$
  - **Leaky ReLU**: (Mass et al., 2013) Fix $a_i$ to a small value e.g. 0.01
  - **Parametric ReLU** (He et al., 2015) Learn $a_i$
  - **Randomized ReLU** (Xu et al., 2015) Sample $a_i$ from a fixed range during training, fix during testing