18-661 Introduction to Machine Learning

Neural Networks-II

Spring 2020

ECE – Carnegie Mellon University

Announcements

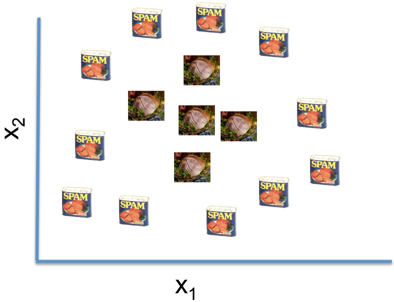
- All campuses have moved to remote lectures, recitations and office hours.
- Homework 4 is due today, March 23 at 11:59pm ET.
- Homework 5 released and due April 1.
- Recitation on Friday will cover neural networks. Same Zoom link as the lectures and same times as before.
- The final exam will also be conducted online – more details to follow
- Re: asking questions via Zoom chat – We very much encourage and appreciate your participation! I will try to answer as many questions as possible, but might miss some questions while teaching the lecture. If your question is not answered, please unmute yourself and ask, or attend the post-class office hours.

Outline

1. Review: Neural networks Motivation
2. Review: Single Neuron Models
3. Review: Multi-layer Neural Network
4. Inference using a Trained Network: Forward Propagation
5. Training a Neural Network: Backpropagatiopn
6. Optimizing SGD Parameters for Faster Convergence

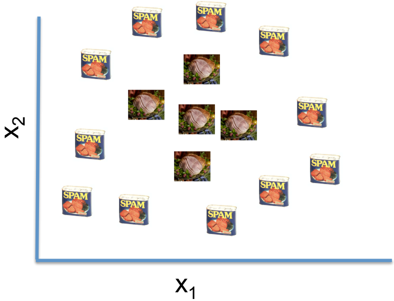
Review: Neural networks
Motivation

Logistic Regression: How to Handle Complex Boundaries?



- This data is not linear separable

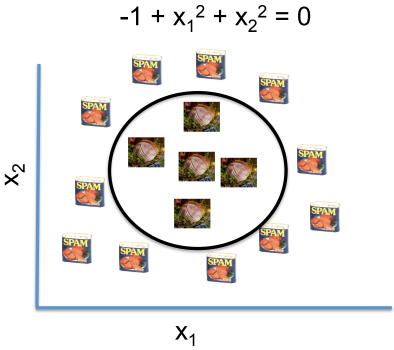
Logistic Regression: How to Handle Complex Boundaries?



- 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



But what if we had a large number of features?

Each feature x_i is one pixel in an 100×100 input image

- Adding polynomial features would result in an enormous $\phi(\mathbf{x})$

But what if we had a large number of features?

Each feature x_i is one pixel in an 100×100 input image

- Adding polynomial features would result in an enormous $\phi(\mathbf{x})$
- 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 100×100 input image

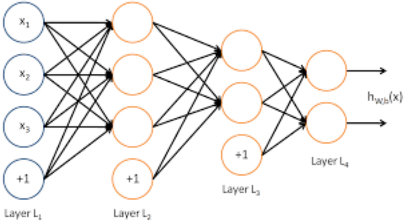
- Adding polynomial features would result in an enormous $\phi(\mathbf{x})$
- Can we somehow only retain the important features?
- We will need to carefully hand-pick them, which can be hard and tedious

But what if we had a large number of features?

Each feature x_i is one pixel in an 100×100 input image

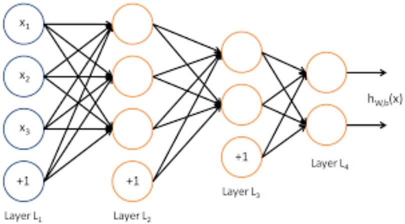
- Adding polynomial features would result in an enormous $\phi(\mathbf{x})$
- Can we somehow only retain the important features?
- We will need to carefully hand-pick them, which can be hard and tedious
- Neural networks automate this for us!

Neural Networks Compress the Set of Features



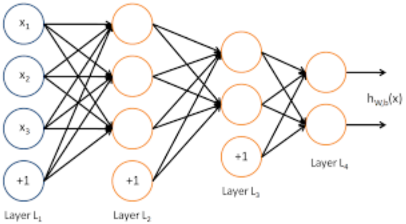
- Start with feature vector **x** containing all pixels in the image

Neural Networks Compress the Set of Features



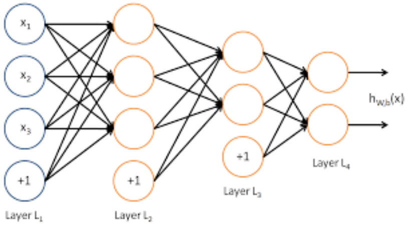
- Start with feature vector **x** containing all pixels in the image
- Layer 1: distill the edges of the image

Neural Networks Compress the Set of Features



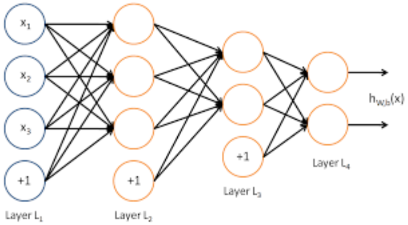
- Start with feature vector $\mathbf{x}$ containing all pixels in the image
- Layer 1: distill the edges of the image
- Layer 2: distill triangles, circles, etc.

Neural Networks Compress the Set of Features



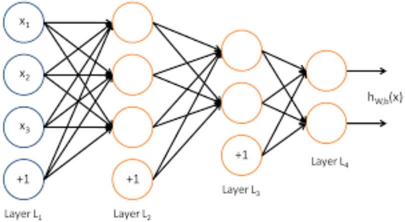
- Start with feature vector **x** containing all pixels in the image
- Layer 1: distill the edges of the image
- Layer 2: distill triangles, circles, etc.
- Layer 3: recognize pointy ears, fur style etc.

Neural Networks Compress the Set of Features



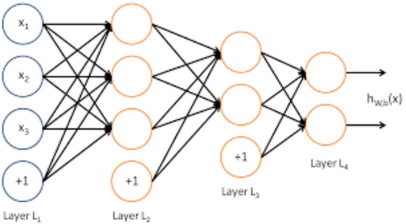
- Start with feature vector **x** containing all pixels in the image
- Layer 1: distill the edges of the image
- Layer 2: distill triangles, circles, etc.
- Layer 3: recognize pointy ears, fur style etc.
- Layer 4: performs logistic regression on the features in layer 3

Neural Networks 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 Networks Compress the Set of Features



- Start with feature vector **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

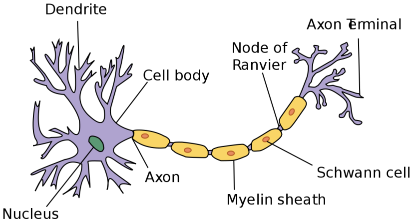
Inspiration from Biology: How does our brain work?

Each feature x_i is one pixel in an 100×100 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 – can be viewed as the transformed feature
- Other neurons use this output as the input signal



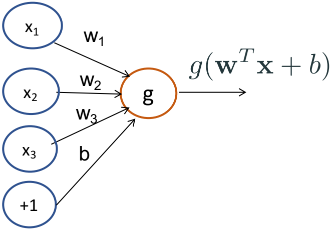
Neuron in the brain

An average human brain has $\sim$ 100 billion neurons!

Artificial Neuron Model

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

Mimicking the human brain

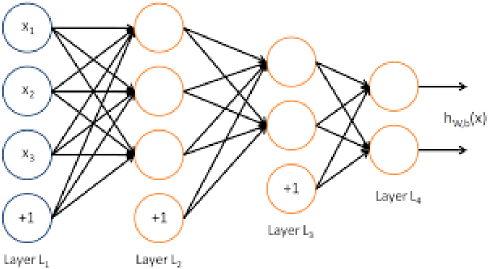
Pass inputs through a "network" of neurons to obtain outputs.

- Neural networks are very good at handling large-scale data.
- They can learn very complex relationships.
- Requires careful configuration: what does this network look like?

Mimicking the human brain

Pass inputs through a “network” of neurons to obtain outputs.

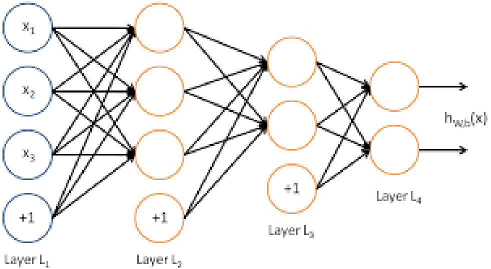
- Neural networks are very good at handling large-scale data.
- They can learn very complex relationships.
- Requires careful configuration: what does this network look like?



Mimicking the human brain

Pass inputs through a "network" of neurons to obtain outputs.

- Neural networks are very good at handling large-scale data.
- They can learn very complex relationships.
- Requires careful configuration: what does this network look like?



Each function is sometimes called a "node" in the network. We group functions into "layers" depending on how many functions their inputs have passed through since the original inputs.

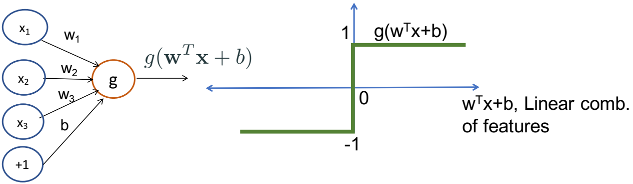
Review: Single Neuron Models

Example 1: Perceptron, Rosenblatt (1957)

- The perception is a single-unit neural network with the activation function $g(x) = \text{sign}(x)$

Example 1: Perceptron, Rosenblatt (1957)

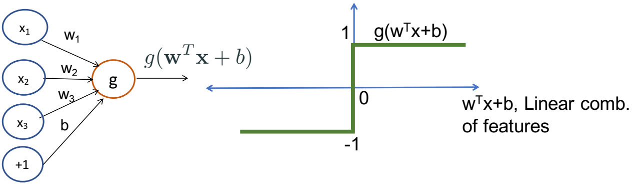
- The perception is a single-unit neural network with the activation function $g(x) = \text{sign}(x)$
- It considers a linear binary classification problem to distinguish between two classes $\{-1, +1\}$.



Perceptron

Example 1: Perceptron, Rosenblatt (1957)

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

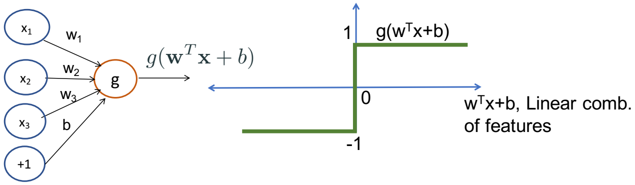


Perceptron

- Assign label $\text{sign}(\boldsymbol{w}^\top \boldsymbol{x} + b)$ to a new sample

Example 1: Perceptron, Rosenblatt (1957)

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



Perceptron

- Assign label $\text{sign}(\boldsymbol{w}^\top \boldsymbol{x} + b)$ 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

$$\varepsilon = \sum_n \mathbb{I}[y_n \neq \text{sign}(\boldsymbol{w}^\top \boldsymbol{x}_n)]$$

Algorithm: For a randomly chosen data point $(\mathbf{x}_n, y_n)$ make small changes to $\boldsymbol{w}$ so that

$$y_n = \text{sign}(\boldsymbol{w}^\top \boldsymbol{x}_n)$$

Two cases

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

$$\boldsymbol{w}^{\text{NEW}} \leftarrow \boldsymbol{w}^{\text{OLD}} + y_n \boldsymbol{x}_n$$

Why would it work?

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

$$y_n(\boldsymbol{w}^\top \boldsymbol{x}_n) < 0$$

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

$$y_n[(\boldsymbol{w} + y_n \boldsymbol{x}_n)^\top \boldsymbol{x}_n] = y_n \boldsymbol{w}^\top \boldsymbol{x}_n + y_n^2 \boldsymbol{x}_n^\top \boldsymbol{x}_n$$

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

$$y_n(\boldsymbol{w}^{\text{NEW}\top} \boldsymbol{x}_n) > 0$$

i.e., we are more likely to classify correctly

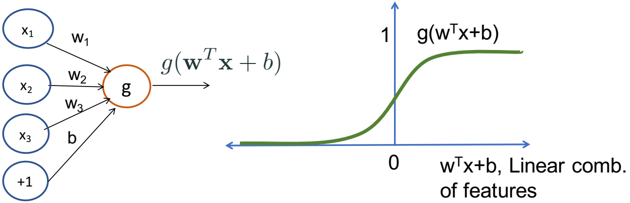
Properties

- 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 $\boldsymbol{w}_0 = 0$).
- We don't need to set a learning rate

The perceptron algorithm was used in old times to train $\mathbf{w}$ by hand, without a computer.

Example 2: Binary Logistic Regression

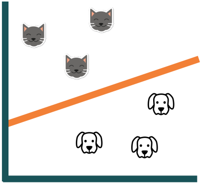
- Suppose g is the sigmoid function $\sigma(\mathbf{w}^T\mathbf{x} + b) = \frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}$
- We can find a linear decision boundary separating two classes. The output is the probability of $\mathbf{x}$ belonging to class 1.



Neuron with Sigmoid activation

Example 2: Binary Logistic Regression

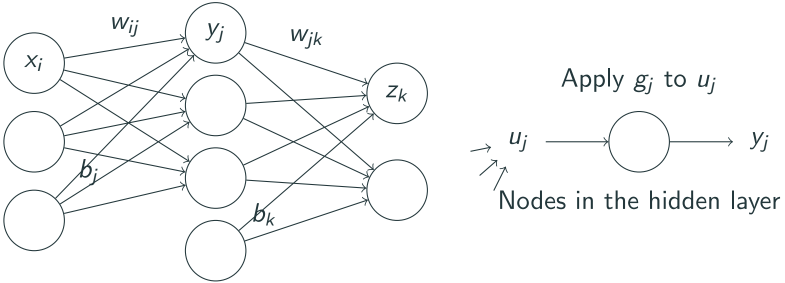
- Suppose g is the sigmoid function $\sigma(\mathbf{w}^T\mathbf{x} + b) = \frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}$
- We can find a linear decision boundary separating two classes. The output is the probability of $\mathbf{x}$ belonging to class 1.
- This is binary logistic regression, which we already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



linear decision boundary

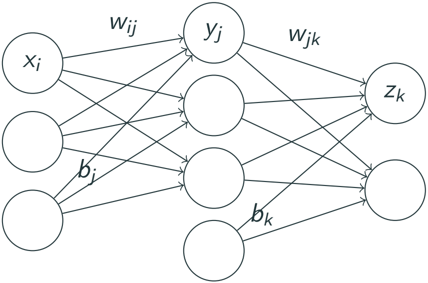
Review: Multi-layer Neural Network

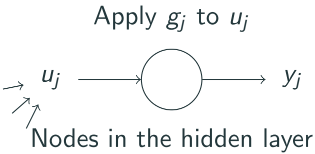
Multi-layer Neural Network



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

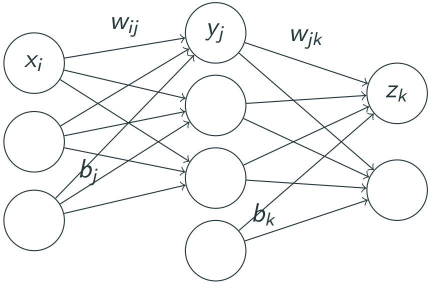
Multi-layer Neural Network

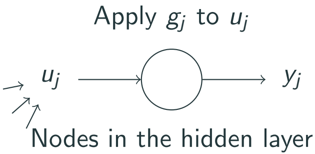




- w_{ij}: weights connecting node i in layer $(\ell - 1)$ to node j in layer ℓ.
- b_j, b_k: bias for nodes in layers j and k respectively.
- u_j, u_k: inputs to nodes j and k (where $u_j = b_j + \sum_i x_i w_{ij}$).

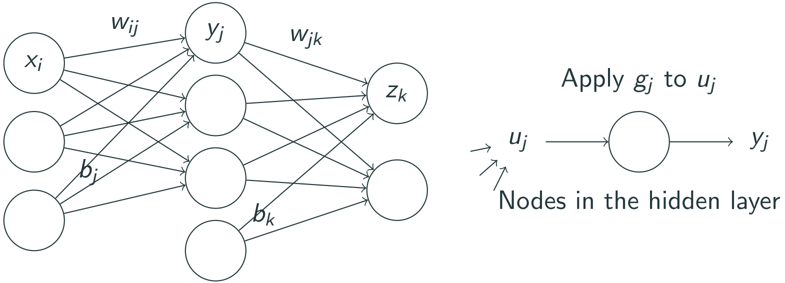
Multi-layer Neural Network





- w_{ij}: weights connecting node i in layer $(\ell - 1)$ to node j in layer ℓ.
- b_j, b_k: bias for nodes in layers j and k respectively.
- 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.

Multi-layer Neural Network

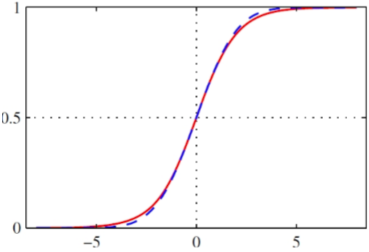


- w_{ij}: weights connecting node i in layer $(\ell - 1)$ to node j in layer ℓ.
- b_j, b_k: bias for nodes in layers j and k respectively.
- u_j, u_k: inputs to nodes j and k (where $u_j = b_j + \sum_i x_i w_{ij}$).
- g_j, g_k: activation function for node j (applied to u_j) and node k.
- $y_j = g_j(u_j)$,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

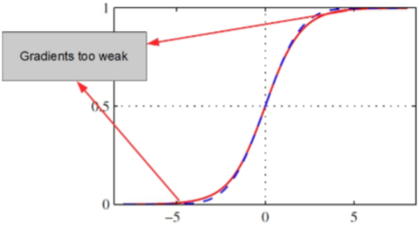
Sigmoid Activation Function

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

- Squashing type non-linearity: pushes output to range [0,1]

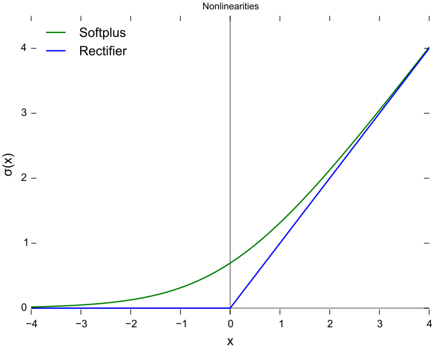


The vanishing gradients problem



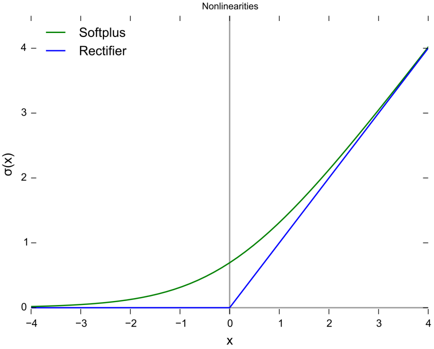
- Problem: Near-constant value across most of their domain, strongly sensitive only when z is closer to zero
- Saturation makes gradient based learning difficult

Rectified Linear Units



- Approximates the softplus function which is $\log(1 + e^z)$
- ReLu Activation function is $g(z) = max(0, z)$ with $z \in R$

Rectified Linear Units



- Approximates the softplus function which is $\log(1 + e^z)$
- ReLu Activation function is $g(z) = max(0, z)$ with $z \in R$
- Similar to linear units. Easy to optimize!
- Give large and *consistent* gradients when active

Activation Choices for Each Layer

Output layer produces a classification decision.

- Probabilities of the input being in each class.
- Often uses sigmoid, softmax, or tanh activations.

Activation Choices for Each Layer

Output layer produces a classification decision.

- Probabilities of the input being in each class.
- Often uses sigmoid, softmax, or tanh activations.

Hidden layers convert activated inputs to classification features.

- ReLU, Leaky ReLU, ELU and variants are popular choices.

Activation Choices for Each Layer

Output layer produces a classification decision.

- Probabilities of the input being in each class.
- Often uses sigmoid, softmax, or tanh activations.

Hidden layers convert activated inputs to classification features.

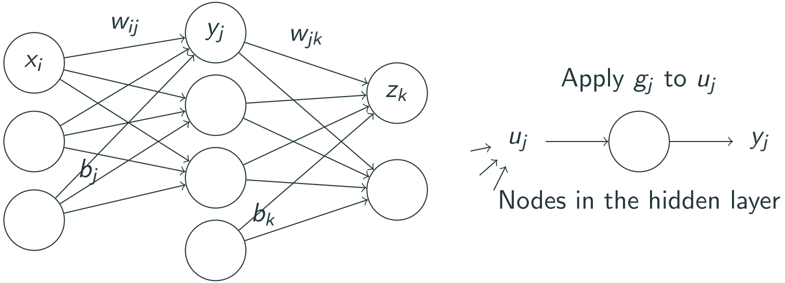
- ReLU, Leaky ReLU, ELU and variants are popular choices.

Input layer initially transforms the features.

Often uses linear, sigmoid, or tanh activations.

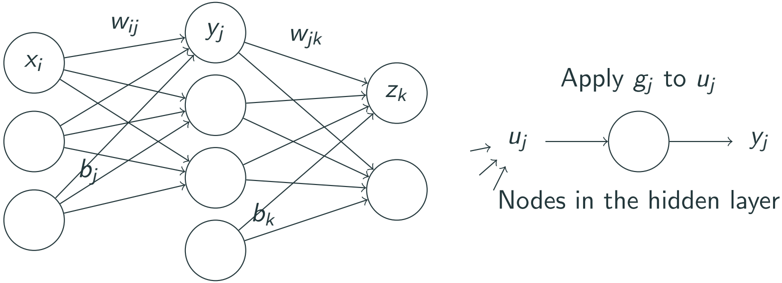
Inference using a Trained Network: Forward Propagation

How do you perform inference using a trained neural network?



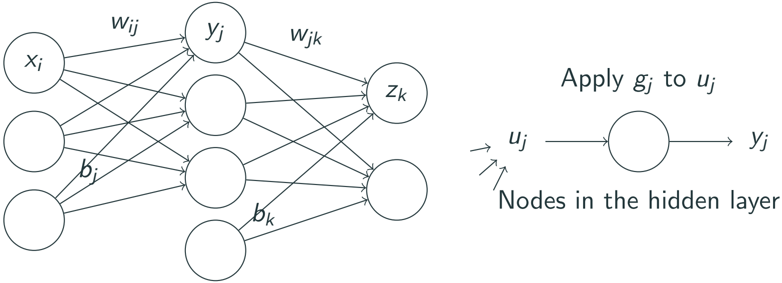
- Expressing outputs z in terms of inputs x is called forward-propagation.

How do you perform inference using a trained neural network?



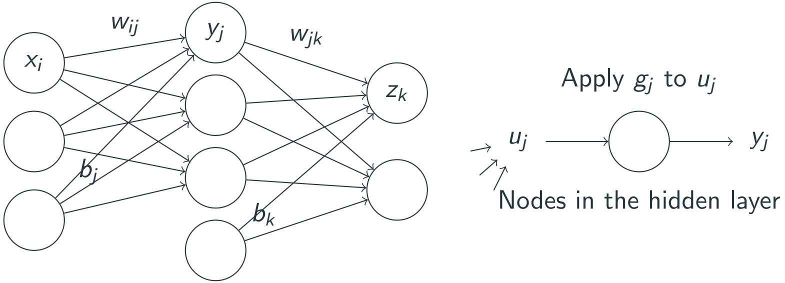
- Expressing outputs z in terms of inputs x is called forward-propagation.
  - Express inputs u_j to the hidden layer in terms of x

How do you perform inference using a trained neural network?



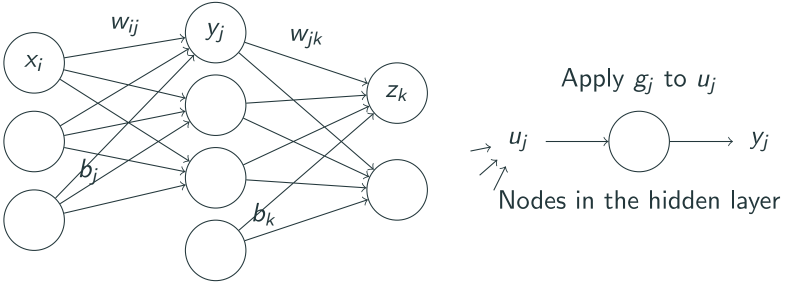
- Expressing outputs z in terms of inputs x is called forward-propagation.
  - Express inputs u_j to the hidden layer in terms of x

How do you perform inference using a trained neural network?



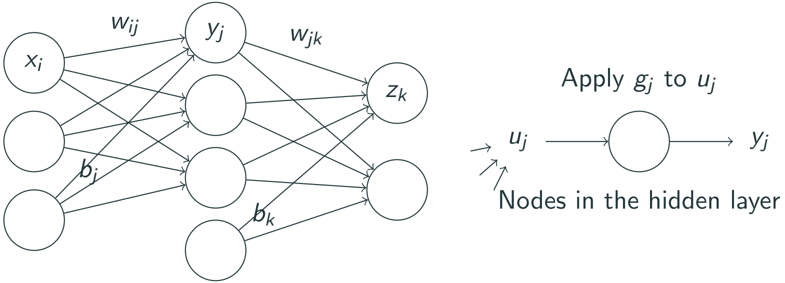
- Expressing outputs z in terms of inputs x is called forward-propagation.
  - Express inputs u_j to the hidden layer in terms of x: $u_j = \sum_i w_{ij} x_i + b_j$
  - Express outputs y_j of the hidden layer in terms of x

How do you perform inference using a trained neural network?



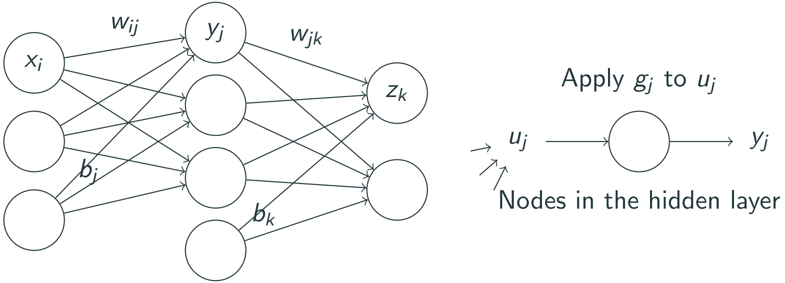
- Expressing outputs z in terms of inputs x is called forward-propagation.
    - Express inputs u_j to the hidden layer in terms of x: $u_j = \sum_i w_{ij} x_i + b_j$
    - Express outputs y_j of the hidden layer in terms of x

How do you perform inference using a trained neural network?



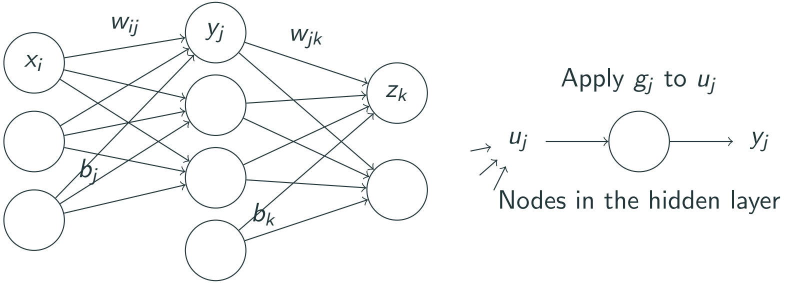
- Expressing outputs z in terms of inputs x is called forward-propagation.
  - Express inputs u_j to the hidden layer in terms of x: $u_j = \sum_i w_{ij} x_i + b_j$
  - Express outputs y_j of the hidden layer in terms of x: $y_j = g(\sum_i w_{ij} x_i + b_j)$
  - Express inputs to the final layer in terms of x

How do you perform inference using a trained neural network?



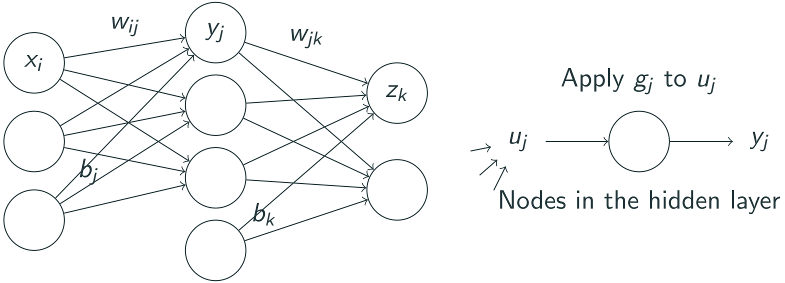
- Expressing outputs z in terms of inputs x is called forward-propagation.
  - Express inputs u_j to the hidden layer in terms of x: $u_j = \sum_i w_{ij}x_i + b_j$
  - Express outputs y_j of the hidden layer in terms of x: $y_j = g(\sum_i w_{ij}x_i + b_j)$
  - Express inputs to the final layer in terms of x
  - Express outputs z_k of the final layer in terms of x

How do you perform inference using a trained neural network?



- Expressing outputs z in terms of inputs x is called forward-propagation.
  - Express inputs u_j to the hidden layer in terms of x: $u_j = \sum_i w_{ij} x_i + b_j$
  - Express outputs y_j of the hidden layer in terms of x: $y_j = g(\sum_i w_{ij} x_i + b_j)$
  - Express inputs to the final layer in terms of x
  - Express outputs z_k of the final layer in terms of x

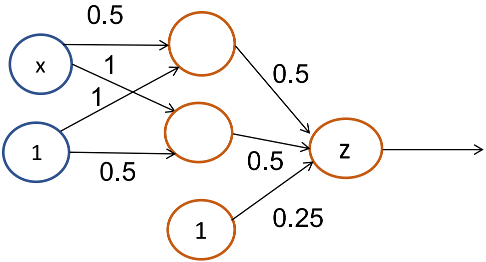
How do you perform inference using a trained neural network?



- Expressing outputs z in terms of inputs x is called forward-propagation.
  - Express inputs u_j to the hidden layer in terms of x: $u_j = \sum_i w_{ij} x_i + b_j$
  - Express outputs y_j of the hidden layer in terms of x: $y_j = g(\sum_i w_{ij} x_i + b_j)$
  - Express inputs to the final layer in terms of x
  - Express outputs z_k of the final layer in terms of x: $z_k = g(\sum_j w_{jk} y_j + b_k)$

Exercise: Forward-Propagation

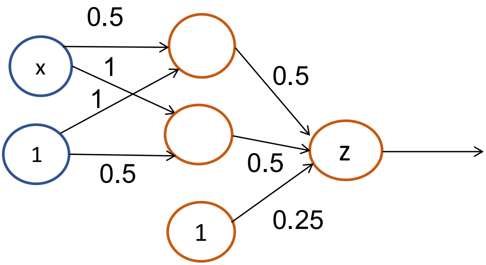
Assume that we are using the sigmoid $\sigma(x) = 1/(1 + e^{-x})$ activation function.



- Outputs of the hidden layer are

Exercise: Forward-Propagation

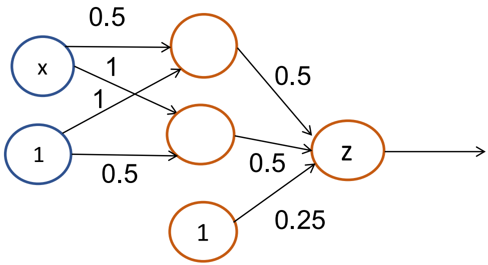
Assume that we are using the sigmoid $\sigma(x) = 1/(1 + e^{-x})$ activation function.



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

Exercise: Forward-Propagation

Assume that we are using the sigmoid $\sigma(x) = 1/(1+e^{-x})$ activation function.



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

Training a Neural Network: Backpropagatiopn

Learning Parameters

How to learn the parameters?

- Choose the right loss function

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (today's class)

$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (today's class)

$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (today's class)

$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

  - Classification: cross-entropy loss (in the homework)

$$min - \sum_n \sum_k t_{nk} \log f_k(\mathbf{x}_n) + (1 - t_{nk}) \log(1 - f_k(\mathbf{x}_n))$$

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (today's class)

$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

  - Classification: cross-entropy loss (in the homework)

$$min - \sum_n \sum_k t_{nk} \log f_k(\mathbf{x}_n) + (1 - t_{nk}) \log(1 - f_k(\mathbf{x}_n))$$

- Hard optimization problem because of f (the output of the neural network) is a complicated function of $\mathbf{x}_n$

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (today's class)

$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

  - Classification: cross-entropy loss (in the homework)

$$min - \sum_n \sum_k t_{nk} \log f_k(\mathbf{x}_n) + (1 - t_{nk}) \log(1 - f_k(\mathbf{x}_n))$$

- Hard optimization problem because of f (the output of the neural network) is a complicated function of $\mathbf{x}_n$

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (today's class)

$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

  - Classification: cross-entropy loss (in the homework)

$$min - \sum_n \sum_k t_{nk} \log f_k(\mathbf{x}_n) + (1 - t_{nk}) \log(1 - f_k(\mathbf{x}_n))$$

- Hard optimization problem because of f (the output of the neural network) is a complicated function of $\mathbf{x}_n$
  - Stochastic gradient descent is commonly used

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (today's class)

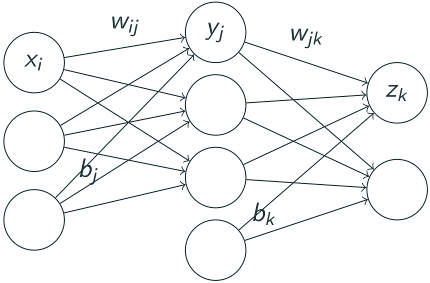
$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

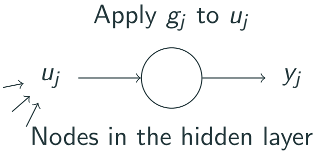
  - Classification: cross-entropy loss (in the homework)

$$min - \sum_n \sum_k t_{nk} \log f_k(\mathbf{x}_n) + (1 - t_{nk}) \log(1 - f_k(\mathbf{x}_n))$$

- Hard optimization problem because of f (the output of the neural network) is a complicated function of $\mathbf{x}_n$
  - Stochastic gradient descent is commonly used
  - Many optimization tricks are applied

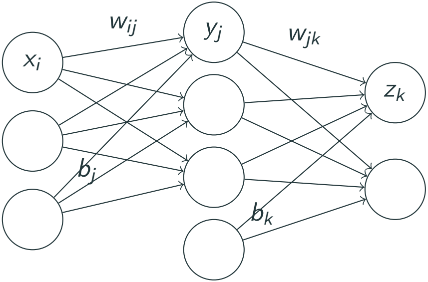
Stochastic gradient descent

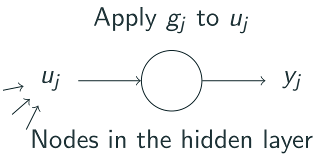




- Randomly pick a data point $(\mathbf{x}_n, t_n)$

Stochastic gradient descent

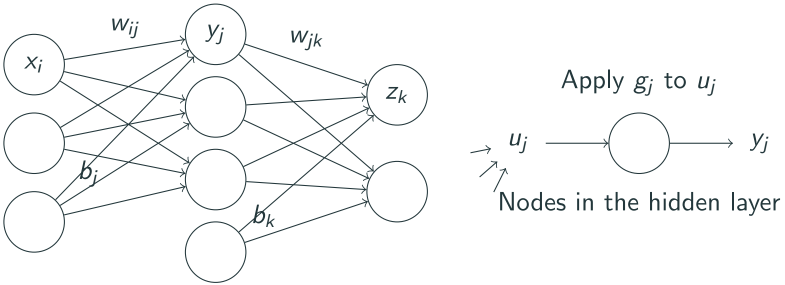




- Randomly pick a data point $(\mathbf{x}_n, t_n)$
- Compute the gradient using only this data point, for example,

$$\Delta = \frac{\partial [f(\mathbf{x}_n) - t_n]^2}{\partial w}$$

Stochastic gradient descent

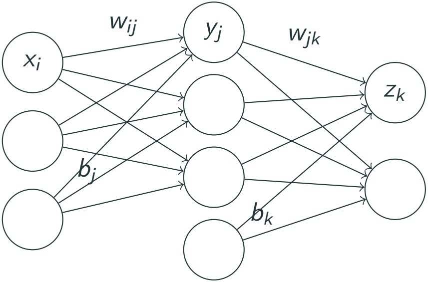


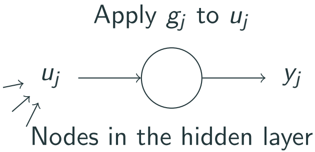
- Randomly pick a data point $(\mathbf{x}_n, t_n)$
- Compute the gradient using only this data point, for example,

$$\Delta = \frac{\partial [f(\mathbf{x}_n) - t_n]^2}{\partial w}$$

- Update the parameters: $\mathbf{w} \leftarrow \mathbf{w} - \eta\Delta$

Stochastic gradient descent





- Randomly pick a data point $(\mathbf{x}_n, t_n)$
- Compute the gradient using only this data point, for example,

$$\Delta = \frac{\partial [f(\mathbf{x}_n) - t_n]^2}{\partial w}$$

- Update the parameters: $\mathbf{w} \leftarrow \mathbf{w} - \eta \Delta$
- Iterate the process until some (pre-specified) stopping criteria

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").

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.

Updating the parameter values

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

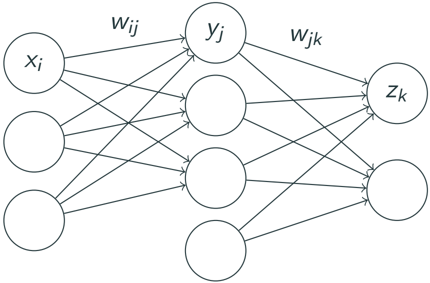
- Step 1: Forward-propagate to find z_k in terms of the input (the "feed-forward signals").
- Step 2: Calculate output error E by comparing the predicted output z_k to its true value t_k.
- Step 3: Back-propagate E by weighting it by the gradients of the associated activation functions and the weights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

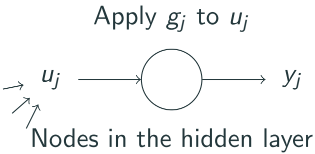
Updating the parameter values

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.
- 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 η is the step size.

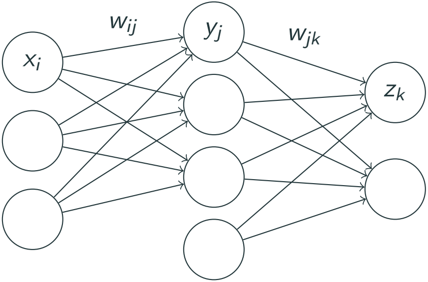
Illustrative example

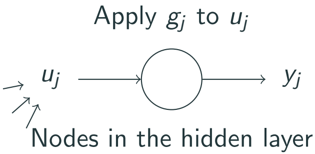




- w_{ij}: weights connecting node i in layer $(\ell - 1)$ to node j in layer ℓ.
- b_j, b_k: bias for nodes j and k.
- u_j, u_k: inputs to nodes j and k (where $u_j = b_j + \sum_i x_i w_{ij}$).
- g_j, g_k: activation function for node j (applied to u_j) and node k.
- $y_j = g_j(u_j)$, $z_k = g_k(u_k)$: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

Illustrative example (steps 1 and 2)

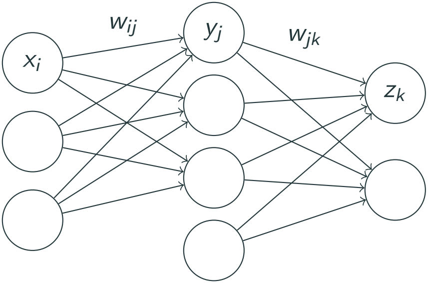


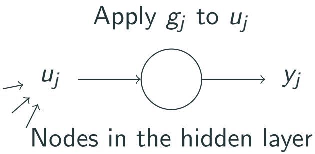


- Step 1: Forward-propagate for each output z_k.

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

Illustrative example (steps 1 and 2)



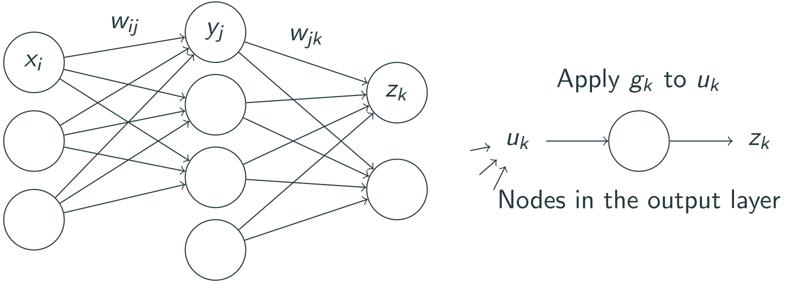


- Step 1: Forward-propagate for each output z_k.

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

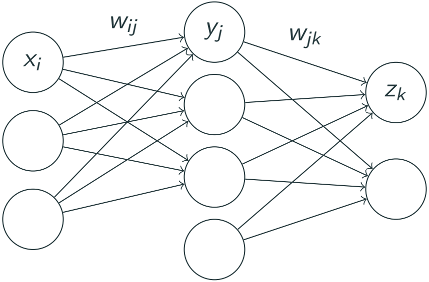
- Step 2: Find the error. Let's assume that the error function is the sum of the squared differences between the target values t_k and the network output z_k: $E = \frac{1}{2} \sum_{k \in K} (z_k - t_k)^2$.

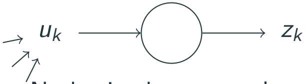
Illustrative example (step 3, output layer)



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

Illustrative example (step 3, output layer)

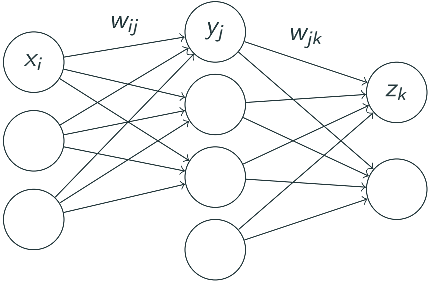


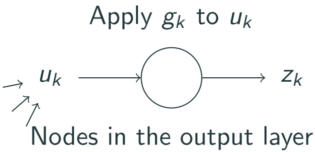


Step 3: Backpropagate the error. Let's start at the output layer with weight w_{jk}, recalling that $E = \frac{1}{2}\sum_{k \in K}(z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial u_k} \frac{\partial u_k}{\partial w_{jk}}$$

Illustrative example (step 3, output layer)

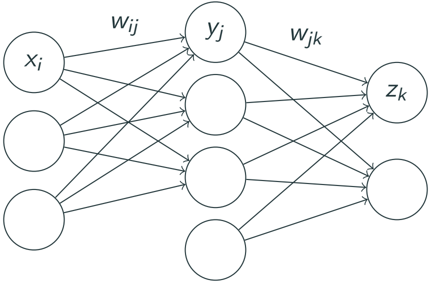


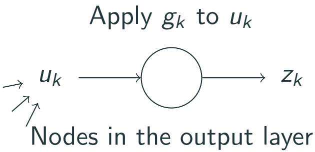


Step 3: Backpropagate the error. Let's start at the output layer with weight w_{jk}, recalling that $E = \frac{1}{2}\sum_{k \in K}(z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$:

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

Illustrative example (step 3, output layer)

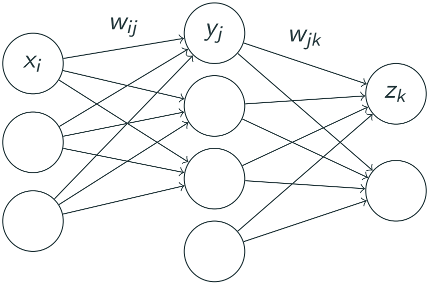


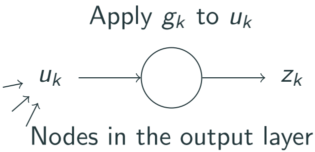


Step 3: Backpropagate the error. Let's start at the output layer with weight w_{jk}, recalling that $E = \frac{1}{2}\sum_{k \in K}(z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$:

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

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



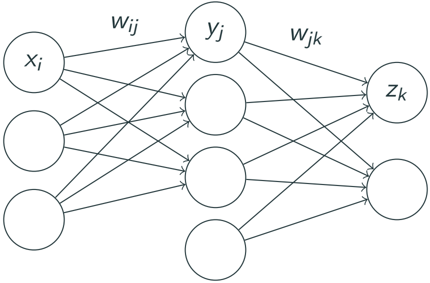


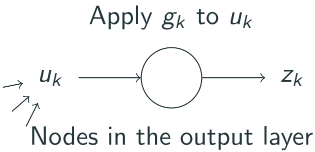
Step 3: Backpropagate the error. Let's start at the output layer with weight w_{jk}, recalling that $E = \frac{1}{2}\sum_{k \in K}(z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$:

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

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

Illustrative example (step 3, output layer)





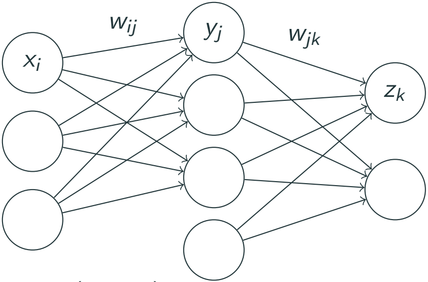
Step 3: Backpropagate the error. Let's start at the output layer with weight w_{jk}, recalling that $E = \frac{1}{2}\sum_{k\in K}(z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk}y_j$:

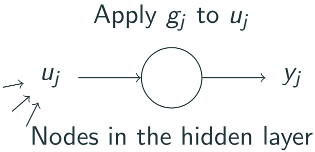
$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial w_{jk}} = (z_k - t_k)\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial w_{jk}}$$

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

where $\delta_k = (z_k - t_k)g_k'(u_k)$ is called the error in u_k.

Illustrative example (step 3, hidden layer)

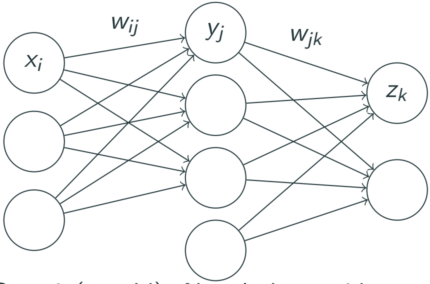


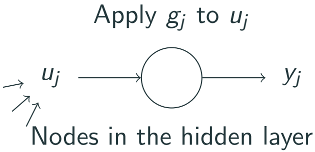


Step 3 (cont'd): Now let's consider w_{ij} in the hidden layer, recalling $u_j = b_i + \sum_i x_i w_{ij}$, $u_k = b_k + \sum_j g_j(u_j) w_{jk}$, $z_k = g_k(u_k)$:

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

Illustrative example (step 3, hidden layer)

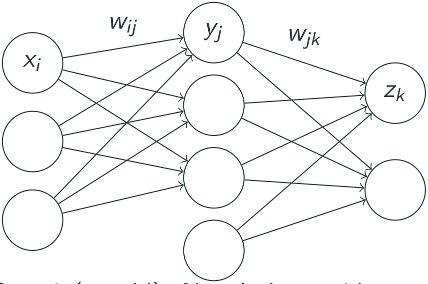


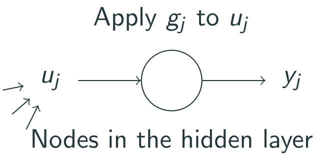


Step 3 (cont'd): Now let's consider w_{ij} in the hidden layer, recalling $u_j = b_i + \sum_i x_i w_{ij}$, $u_k = b_k + \sum_j g_j(u_j) w_{jk}$, $z_k = g_k(u_k)$:

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

Illustrative example (step 3, hidden layer)



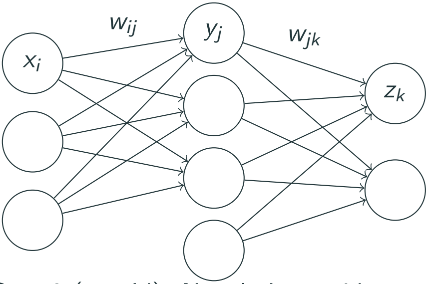


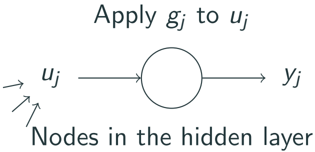
Step 3 (cont'd): Now let's consider w_{ij} in the hidden layer, recalling $u_j = b_i + \sum_i x_i w_{ij}$, $u_k = b_k + \sum_j g_j(u_j) w_{jk}$, $z_k = g_k(u_k)$:

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

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

Illustrative example (step 3, hidden layer)





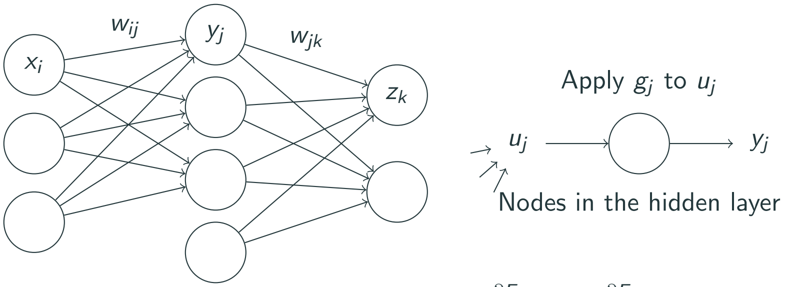
Step 3 (cont'd): Now let's consider w_{ij} in the hidden layer, recalling $u_j = b_i + \sum_i x_i w_{ij}$, $u_k = b_k + \sum_j g_j(u_j) w_{jk}$, $z_k = g_k(u_k)$:

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

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

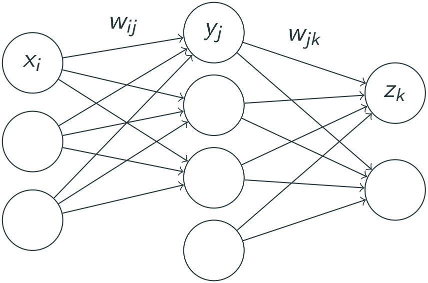
where we substituted $\delta_j = g_j'(u_j) \sum_{k \in K} (z_k - t_k) g_k'(u_k) w_{jk}$, the error in u_j.

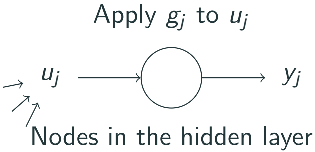
Illustrative example (steps 3 and 4)



- Step 3 (cont'd): We similarly find that $\frac{\partial E}{\partial b_k} = \delta_k$, $\frac{\partial E}{\partial b_j} = \delta_j$.

Illustrative example (steps 3 and 4)



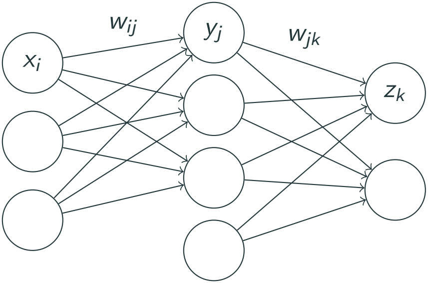


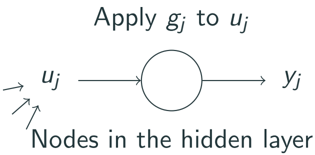
- Step 3 (cont'd): We similarly find that $\frac{\partial E}{\partial b_k} = \delta_k$, $\frac{\partial E}{\partial b_j} = \delta_j$.
- Step 4: Calculate the gradients. We have found that

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

where $\delta_k = (z_k - t_k) g_k'(u_k)$, $\delta_j = g_j'(u_j) \sum_{k \in K} (z_k - t_k) g_k'(u_k) w_{jk}$.

Illustrative example (steps 3 and 4)



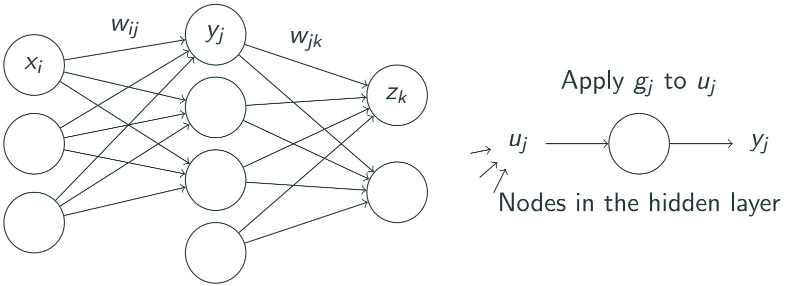


- Step 3 (cont'd): We similarly find that $\frac{\partial E}{\partial b_k} = \delta_k$, $\frac{\partial E}{\partial b_j} = \delta_j$.
- Step 4: Calculate the gradients. We have found that

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

where $\delta_k = (z_k - t_k)g_k'(u_k)$, $\delta_j = g_j'(u_j)\sum_{k \in K}(z_k - t_k)g_k'(u_k)w_{jk}$. Now since we know the z_k, y_j, x_i, u_k and u_j for a given set of parameter values w, b, we can use these expressions to calculate the gradients at each iteration and update them.

Illustrative example (steps 4 and 5)



- Step 4: Calculate the gradients. We have found that

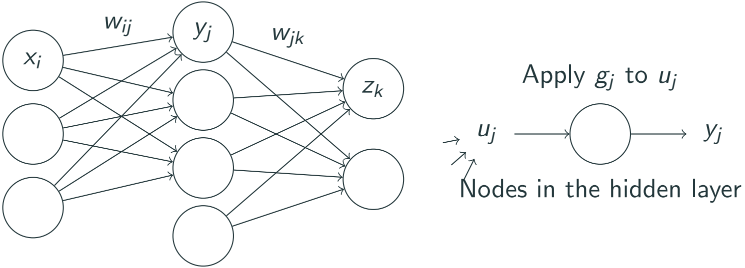
$$\frac{\partial E}{\partial w_{ij}} = \delta_j x_i \text{ and } \frac{\partial E}{\partial w_{jk}} = \delta_k y_j.$$

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

- Step 5: Update the weights and biases with learning rate η. For example

$$w_{jk} \leftarrow w_{jk} - \eta \frac{\partial E}{\partial w_{jk}} \text{ and } w_{ij} \leftarrow w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}$$

High-level Procedure: Can be Used with More Hidden Layers



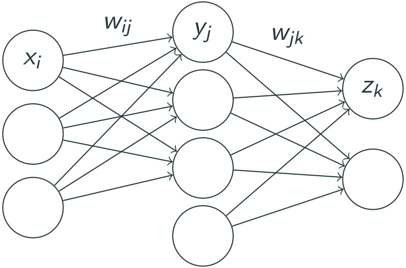
Final Layer

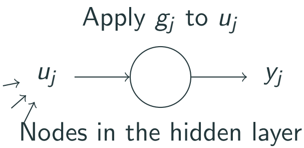
- Error in each of its outputs is $z_k - t_k$.
- Error in input u_k to the final layer is $\delta_k = g_k'(u_k)(z_k - t_k)$

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

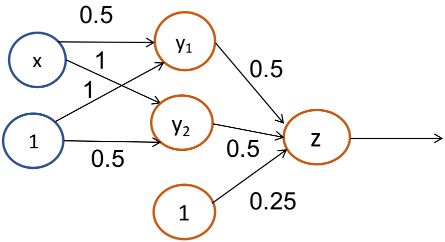
- Error in each of its outputs is $z_k - t_k$.
- Error in input u_k to the final layer is $\delta_k = g_k'(u_k)(z_k - t_k)$

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

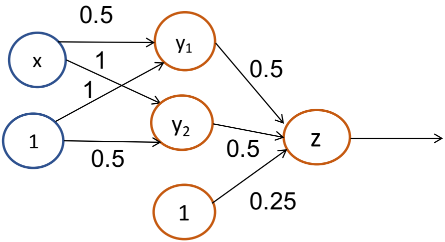
The gradient w.r.t. w_{ij} is $x_i \delta_j$.

Exercise: Back-Propagation



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.

Exercise: Back-Propagation

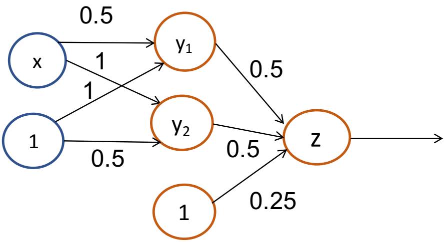


Suppose the output $z = 0.9$ but the target is 1. Perform backpropagation and compute the gradient of error w.r.t. the weight connecting x and y_2.

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 $z = \sigma(u)$

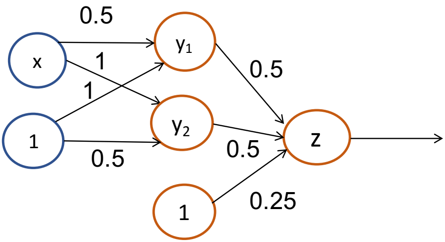
Exercise: Back-Propagation



Final layer

- Error in output z is $0.9 - 1 = -0.1$

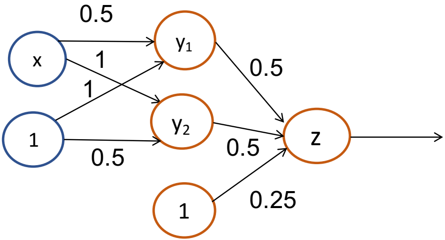
Exercise: Back-Propagation



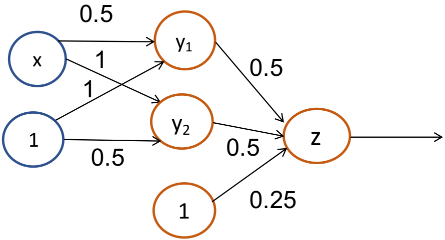
Final layer

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

Exercise: Back-Propagation



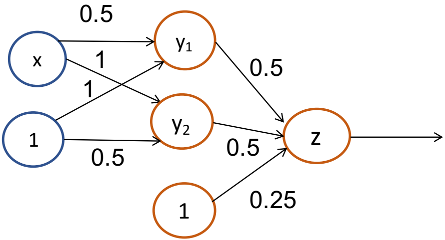
Exercise: Back-Propagation



Hidden Layer

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

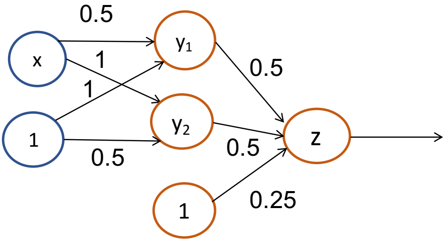
Exercise: Back-Propagation



Hidden Layer

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

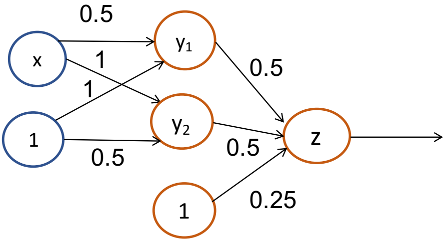
Exercise: Back-Propagation



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 $-0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2)$

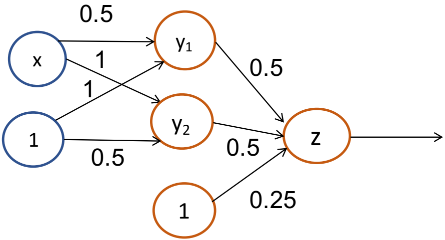
Exercise: Back-Propagation



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 $-0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2)$

Exercise: Back-Propagation

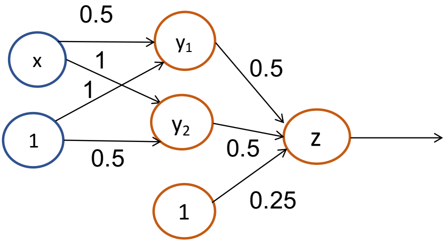


Hidden Layer

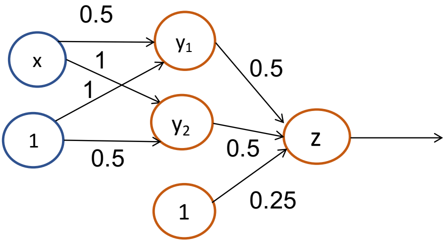
- Error in y_1 is $-0.1 \times \sigma'(u) \times 0.5$
- Error in y_2 is $-0.1 \times \sigma'(u) \times 0.5$
- Error in u_2 is $-0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2)$

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

Exercise: Back-Propagation



Exercise: Back-Propagation

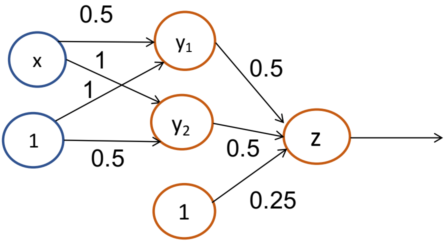


Hidden Layer

- The gradient w.r.t. the weight connecting x and y_2 is

$$\frac{\partial E}{\partial w} = -0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2) \times x$$

Exercise: Back-Propagation



Hidden Layer

- The gradient w.r.t. the weight connecting x and y_2 is

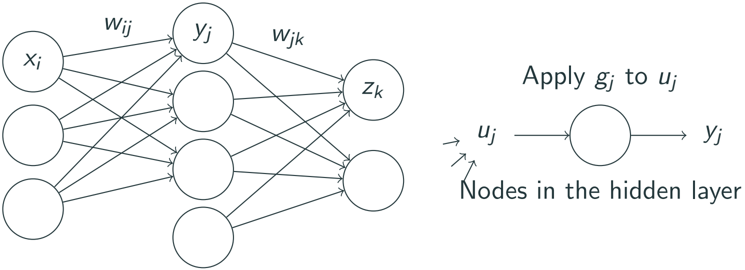
$$\frac{\partial E}{\partial w} = -0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2) \times x$$

- Thus, we will update the weight as

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer

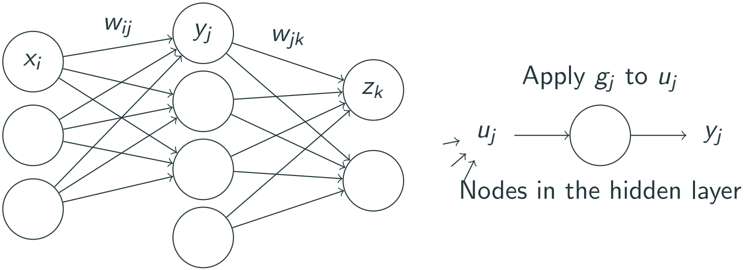


Forward-Propagation

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

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer

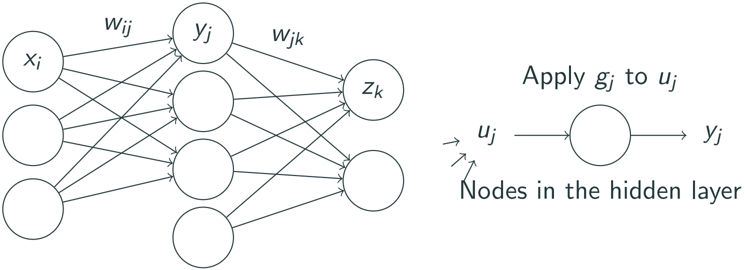


Forward-Propagation

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

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer

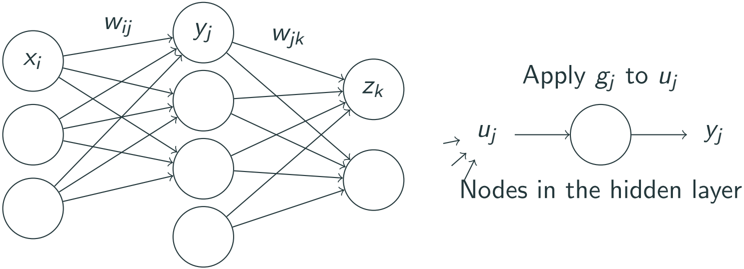


Forward-Propagation

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

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer

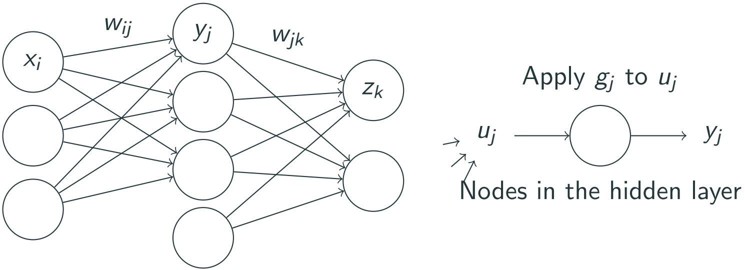


Back-Propagation

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

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer

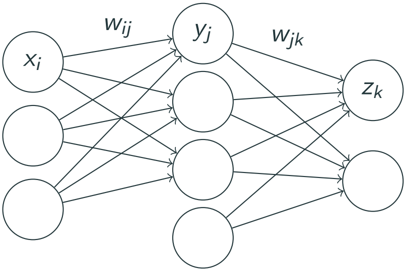


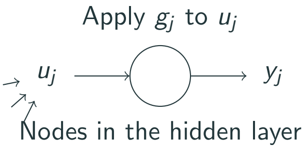
Back-Propagation

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

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer





Back-Propagation

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

Optimizing SGD Parameters for Faster Convergence

Mini-batch SGD

- Recall the empirical risk loss function that we considered for the backpropagation discussion

$$E = \sum_{n=1}^{N} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

Mini-batch SGD

- Recall the empirical risk loss function that we considered for the backpropagation discussion

$$E = \sum_{n=1}^{N} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

- For large training datasets (large N), then computing gradients with respect to each datapoint is expensive. For example, for the last year, the batch gradients are

$$\frac{\partial E}{\partial w_{jk}} = \sum_{n=1}^{N}(z_k - t_k)$$

Mini-batch SGD

- Recall the empirical risk loss function that we considered for the backpropagation discussion

$$E = \sum_{n=1}^{N} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

- For large training datasets (large N), then computing gradients with respect to each datapoint is expensive. For example, for the last year, the batch gradients are

$$\frac{\partial E}{\partial w_{jk}} = \sum_{n=1}^{N}(z_k - t_k)$$

- Therefore we use stochastic gradient descent (SGD), where we choose a random data point $\mathbf{x}_n$ and use $E = \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$ instead of the entire sum

Mini-batch SGD

- Mini-batch SGD is in between these two extremes

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

for backpropagation

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

  for backpropagation
- Small m saves per-iteration computing cost, but increases noise in the gradients and yields worse error convergence

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

  for backpropagation
- Small m saves per-iteration computing cost, but increases noise in the gradients and yields worse error convergence
- Large m reduces gradient noise and gives better error convergence, but increases computing cost per iteration

Mini-batch SGD

- Mini-batch SGD is in between these two extremes

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

for backpropagation

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

  for backpropagation
- Small m saves per-iteration computing cost, but increases noise in the gradients and yields worse error convergence

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

  for backpropagation
- Small m saves per-iteration computing cost, but increases noise in the gradients and yields worse error convergence
- Large m reduces gradient noise and typically gives better error convergence, but increases computing cost per iteration

How to Choose Mini-batch size

- Small training datasets – use batch gradient descent $m = N$
- Large training datasets – typical m are 64, 128, 256 ... whatever fits in the CPU/GPU memory
- Mini-batch size is another hyperparameter that you have to tune

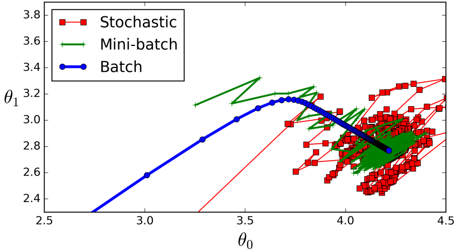


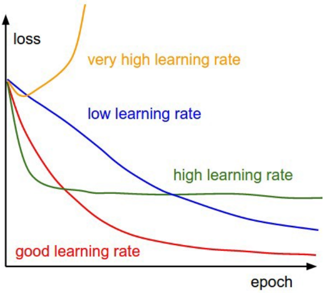
Image source: `https://github.com/buomsoo-kim/Machine-learning-toolkits-with-python`

Learning Rate

- SGD Update Rule

$$w^{(t+1)} = w^{(t)} - \eta \frac{\partial E}{\partial w^{(t)}} = w^{(t)} - \eta \nabla E(w^{(t)})$$

- Large η; Faster convergence, but higher error floor (the flat portion of each curve)
- Small η: Slow convergence, but lower error floor (the blue curve will eventually go below the red curve)
- To get the best of both worlds, decay η over time



Summary

You should know:

- Multi-layer neural network architecture – typical choices of activation functions and loss functions.
- How to perform inference on a trained network using forward propagation
- How to train a neural network using the back-propagation algorithm.
- Effect of learning rate and mini-batch size on training speed and accuracy

Next class

- More optimizing neural network training
- Other types of neural networks