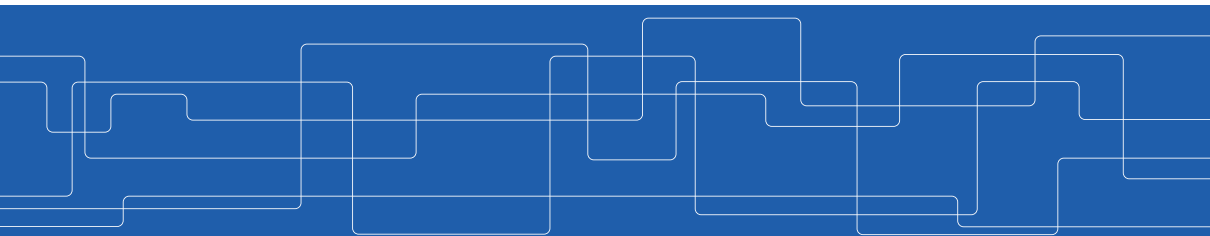




Deep Feedforwards Networks

Amir H. Payberah
payberah@kth.se
28/11/2018



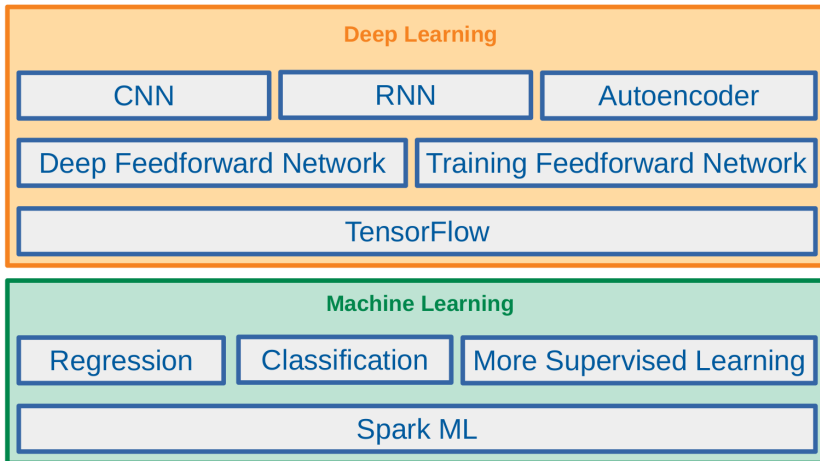


The Course Web Page

`https://id2223kth.github.io`

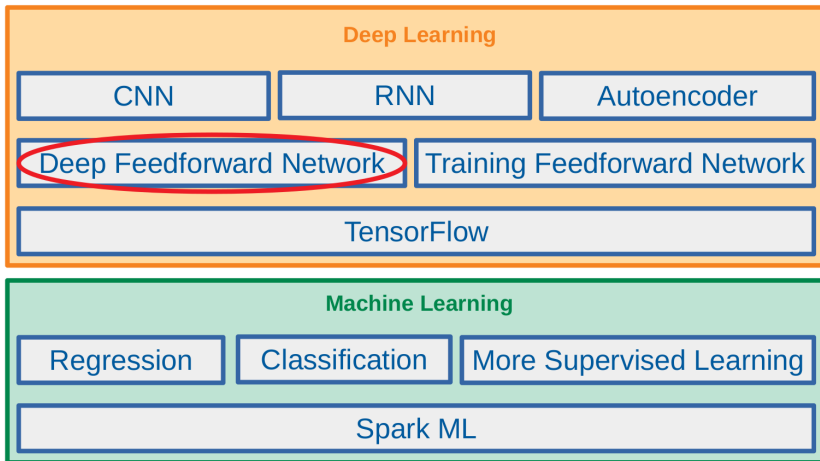


Where Are We?





Where Are We?



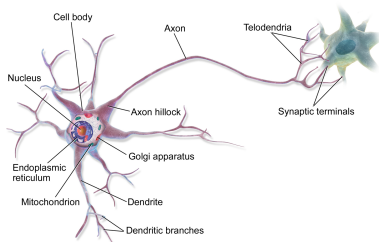
Nature ...

- ▶ **Nature** has inspired many of our **inventions**
 - Birds inspired us to fly
 - Burdock plants inspired velcro
 - Etc.



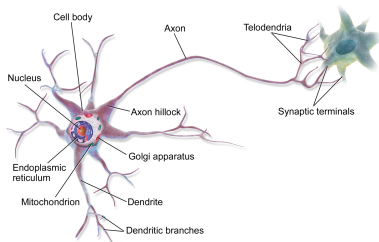
Biological Neurons (1/2)

- ▶ Brain architecture has inspired artificial neural networks.



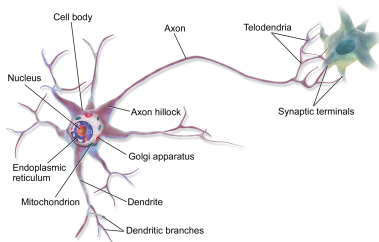
Biological Neurons (1/2)

- ▶ Brain architecture has inspired artificial neural networks.
- ▶ A biological neuron is composed of
 - Cell body, many dendrites (branching extensions), one axon (long extension), synapses



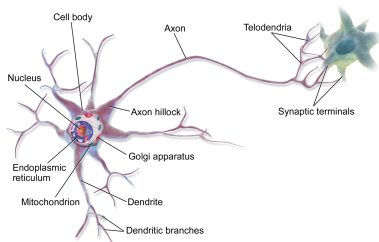
Biological Neurons (1/2)

- ▶ Brain architecture has inspired artificial neural networks.
- ▶ A biological neuron is composed of
 - Cell body, many dendrites (branching extensions), one axon (long extension), synapses
- ▶ Biological neurons receive signals from other neurons via these synapses.



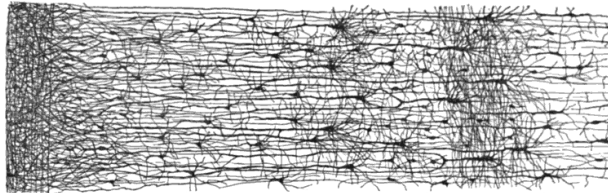
Biological Neurons (1/2)

- ▶ Brain architecture has inspired artificial neural networks.
- ▶ A biological neuron is composed of
 - Cell body, many dendrites (branching extensions), one axon (long extension), synapses
- ▶ Biological neurons receive signals from other neurons via these synapses.
- ▶ When a neuron receives a sufficient number of signals within a few milliseconds, it fires its own signals.



Biological Neurons (2/2)

- ▶ Biological neurons are organized in a vast **network of billions of neurons**.
- ▶ Each neuron typically is **connected** to **thousands of other neurons**.



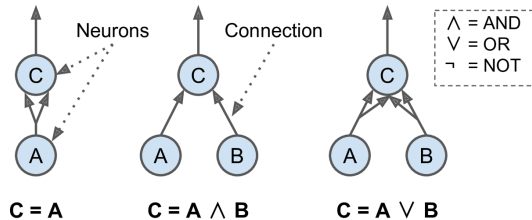


A Simple Artificial Neural Network

- ▶ One or more binary inputs and one binary output
- ▶ Activates its output when more than a certain number of its inputs are active.

A Simple Artificial Neural Network

- ▶ One or more binary inputs and one binary output
- ▶ Activates its output when more than a certain number of its inputs are active.



[A. Geron, O'Reilly Media, 2017]



The Linear Threshold Unit (LTU)

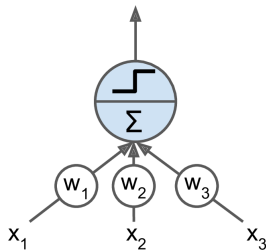
- ▶ Inputs of a LTU are **numbers** (not binary).

The Linear Threshold Unit (LTU)

- ▶ Inputs of a LTU are **numbers** (not binary).
- ▶ Each **input connection** is associated with a **weight**.
- ▶ Computes a **weighted sum of its inputs** and applies a **step function** to that **sum**.

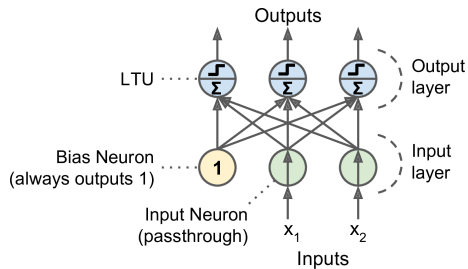
- ▶ $z = w_1x_1 + w_2x_2 + \dots + w_nx_n = \mathbf{w}^T\mathbf{x}$

- ▶ $\hat{y} = \text{step}(z) = \text{step}(\mathbf{w}^T\mathbf{x})$



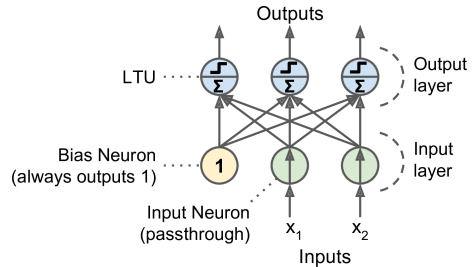
The Perceptron

- ▶ The **perceptron** is a **single layer** of LTUs.



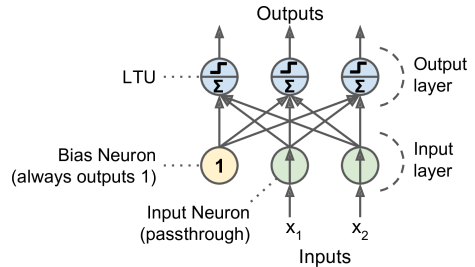
The Perceptron

- ▶ The **perceptron** is a **single layer** of LTUs.
- ▶ The **input neurons** output whatever **input they are fed**.



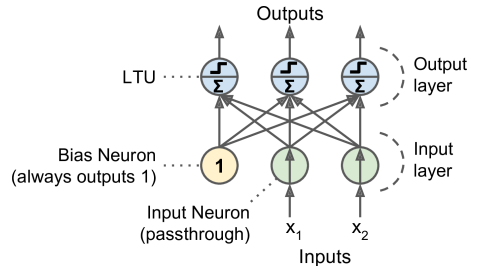
The Perceptron

- ▶ The **perceptron** is a **single layer** of LTUs.
- ▶ The **input neurons** output whatever **input they are fed**.
- ▶ A **bias neuron**, which just **outputs 1** all the time.



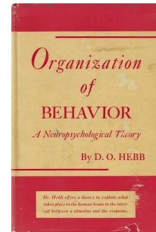
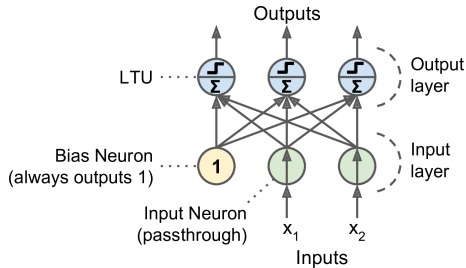
The Perceptron

- ▶ The **perceptron** is a **single layer** of LTUs.
- ▶ The **input neurons** output whatever **input they are fed**.
- ▶ A **bias neuron**, which just **outputs 1 all the time**.
- ▶ If we use **logistic function (sigmoid)** instead of a **step** function, it computes a **continuous** output.



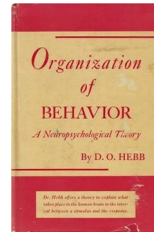
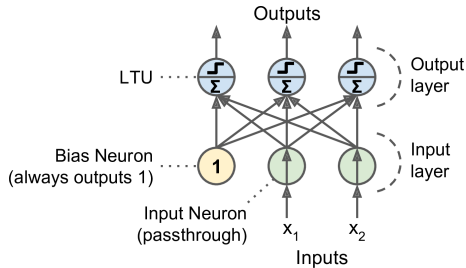
How is a Perceptron Trained? (1/2)

- ▶ The **Perceptron training algorithm** is inspired by **Hebb's rule**.



How is a Perceptron Trained? (1/2)

- ▶ The **Perceptron training algorithm** is inspired by **Hebb's rule**.
- ▶ When a **biological neuron** often **triggers another neuron**, the **connection** between these two neurons grows **stronger**.





How is a Perceptron Trained? (2/2)

- ▶ Feed one training instance \mathbf{x} to each neuron j at a time and make its prediction \hat{y} .
- ▶ Update the connection weights.

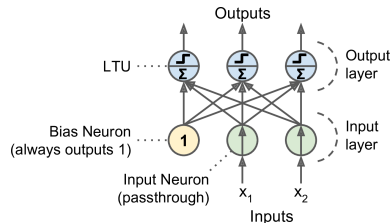
How is a Perceptron Trained? (2/2)

- ▶ Feed one training instance \mathbf{x} to each neuron j at a time and make its prediction \hat{y}_j .
- ▶ Update the connection weights.

$$\hat{y}_j = \sigma(\mathbf{w}_j^T \mathbf{x} + b)$$

$$J(\mathbf{w}_j) = \text{cross_entropy}(y_j, \hat{y}_j)$$

$$\mathbf{w}_{i,j}^{(\text{next})} = \mathbf{w}_{i,j} - \eta \frac{\partial J(\mathbf{w}_j)}{\partial w_i}$$



How is a Perceptron Trained? (2/2)

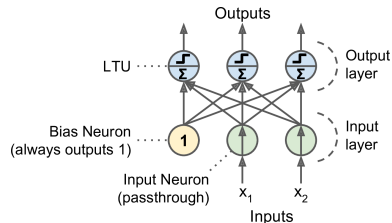
- ▶ Feed one training instance \mathbf{x} to each neuron j at a time and make its prediction \hat{y}_j .
- ▶ Update the connection weights.

$$\hat{y}_j = \sigma(\mathbf{w}_j^T \mathbf{x} + b)$$

$$J(\mathbf{w}_j) = \text{cross_entropy}(y_j, \hat{y}_j)$$

$$\mathbf{w}_{i,j}^{(\text{next})} = \mathbf{w}_{i,j} - \eta \frac{\partial J(\mathbf{w}_j)}{\partial w_i}$$

- ▶ $w_{i,j}$: the weight between neurons i and j .
- ▶ x_i : the i th input value.
- ▶ \hat{y}_j : the j th predicted output value.
- ▶ y_j : the j th true output value.
- ▶ η : the learning rate.



Perceptron in TensorFlow



Perceptron in TensorFlow - First Implementation (1/3)

- ▶ `n_neurons`: number of neurons in a layer.
- ▶ `n_features`: number of features.

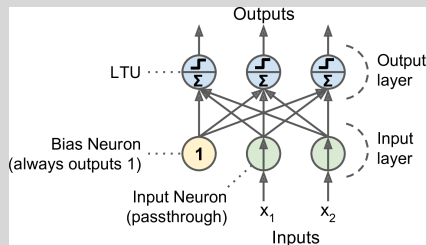
```

n_neurons = 3
n_features = 2

# placeholder
X = tf.placeholder(tf.float32, shape=(None, n_features),
                  name="X")
y_true = tf.placeholder(tf.int64, shape=(None),
                       name="y")

# variables
W = tf.get_variable("weights", dtype=tf.float32,
                    initializer=tf.zeros((n_features, n_neurons)))
b = tf.get_variable("bias", dtype=tf.float32,
                    initializer=tf.zeros((n_neurons)))

```





Perceptron in TensorFlow - First Implementation (2/3)

$$\hat{y}_j = \sigma(\mathbf{w}_j^T \mathbf{x} + b)$$

```
# make the network  
z = tf.matmul(X, W) + b  
y_hat = tf.nn.sigmoid(z)
```



Perceptron in TensorFlow - First Implementation (2/3)

$$\hat{y}_j = \sigma(\mathbf{w}_j^T \mathbf{x} + b)$$

```
# make the network  
z = tf.matmul(X, W) + b  
y_hat = tf.nn.sigmoid(z)
```

$$J(\mathbf{w}_j) = \text{cross_entropy}(y_j, \hat{y}_j) = - \sum_i^m y_j^{(i)} \log(\hat{y}_j^{(i)})$$

```
# define the cost  
cross_entropy = -y_true * tf.log(y_hat)  
cost = tf.reduce_mean(cross_entropy)
```



Perceptron in TensorFlow - First Implementation (2/3)

$$\hat{y}_j = \sigma(\mathbf{w}_j^T \mathbf{x} + b)$$

```
# make the network  
z = tf.matmul(X, W) + b  
y_hat = tf.nn.sigmoid(z)
```

$$J(\mathbf{w}_j) = \text{cross_entropy}(y_j, \hat{y}_j) = - \sum_i^m y_j^{(i)} \log(\hat{y}_j^{(i)})$$

```
# define the cost  
cross_entropy = -y_true * tf.log(y_hat)  
cost = tf.reduce_mean(cross_entropy)
```

$$\mathbf{w}_{i,j}^{(\text{next})} = \mathbf{w}_{i,j} - \eta \frac{\partial J(\mathbf{w}_j)}{\partial \mathbf{w}_i}$$

```
# train the model  
# 1. compute the gradient of cost with respect to W and b  
# 2. update the weights and bias  
learning_rate = 0.1  
new_W = W.assign(W - learning_rate * tf.gradients(xs=W, ys=cost))  
new_b = b.assign(b - learning_rate * tf.gradients(xs=b, ys=cost))
```



Perceptron in TensorFlow - First Implementation (3/3)

- ▶ Execute the network.

```
# execute the model
init = tf.global_variables_initializer()

n_epochs = 100
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        sess.run([new_W, new_b, cost], feed_dict={X: training_X, y_true: training_y})
```



Perceptron in TensorFlow - Second Implementation (1/2)

$$\hat{y}_j = \sigma(\mathbf{w}_j^T \mathbf{x} + b)$$

```
# make the network  
z = tf.matmul(X, W) + b  
y_hat = tf.nn.sigmoid(z)
```



Perceptron in TensorFlow - Second Implementation (1/2)

$$\hat{y}_j = \sigma(\mathbf{w}_j^T \mathbf{x} + b)$$

```
# make the network  
z = tf.matmul(X, W) + b  
y_hat = tf.nn.sigmoid(z)
```

$$J(\mathbf{w}_j) = \text{cross_entropy}(y_j, \hat{y}_j) = - \sum_i^m y_j^{(i)} \log(\hat{y}_j^{(i)})$$

```
# define the cost  
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(z, y_true)  
cost = tf.reduce_mean(cross_entropy)
```



Perceptron in TensorFlow - Second Implementation (1/2)

$$\hat{y}_j = \sigma(\mathbf{w}_j^T \mathbf{x} + b)$$

```
# make the network  
z = tf.matmul(X, W) + b  
y_hat = tf.nn.sigmoid(z)
```

$$J(\mathbf{w}_j) = \text{cross_entropy}(y_j, \hat{y}_j) = - \sum_i^m y_j^{(i)} \log(\hat{y}_j^{(i)})$$

```
# define the cost  
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(z, y_true)  
cost = tf.reduce_mean(cross_entropy)
```

$$\mathbf{w}_{i,j}^{(\text{next})} = \mathbf{w}_{i,j} - \eta \frac{\partial J(\mathbf{w}_j)}{\partial \mathbf{w}_i}$$

```
# train the model  
learning_rate = 0.1  
optimizer = tf.train.GradientDescentOptimizer(learning_rate)  
training_op = optimizer.minimize(cost)
```




Perceptron in TensorFlow - Second Implementation (2/2)

- ▶ Execute the network.

```
# execute the model
init = tf.global_variables_initializer()

n_epochs = 100
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        sess.run(training_op, feed_dict={X: training_X, y_true: training_y})
```



Perceptron in Keras

- ▶ Build and execute the network.

```
n_neurons = 10
y_hat = tf.keras.Sequential([layers.Dense(n_neurons, activation="sigmoid")])

y_hat.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="binary_crossentropy",
              metrics=["accuracy"])

n_epochs = 100
y_hat.fit(training_X, training_y, epochs=n_epochs)
```



Multi-Layer Perceptron (MLP)



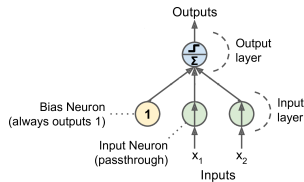
Perceptron Weakness (1/2)

- ▶ Incapable of solving some **trivial problems**, e.g., **XOR** classification problem. **Why?**

Perceptron Weakness (1/2)

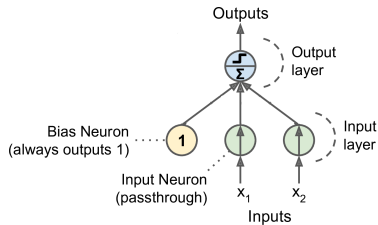
- ▶ Incapable of solving some trivial problems, e.g., XOR classification problem. Why?

| A | B | A XOR B |
|---|---|---------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Perceptron Weakness (2/2)



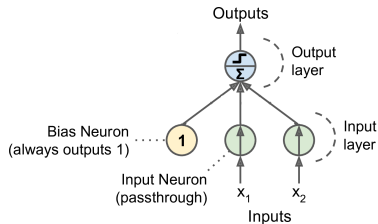
$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$$\hat{y} = \text{step}(z), z = w_1x_1 + w_2x_2 + b$$

$$J(\mathbf{w}) = \frac{1}{4} \sum_{x \in \mathbf{X}} (\hat{y}(x) - y(x))^2$$

Perceptron Weakness (2/2)

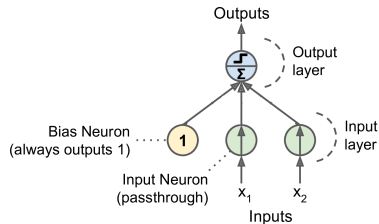


$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \hat{y} = \text{step}(z), z = w_1x_1 + w_2x_2 + b$$

$$J(\mathbf{w}) = \frac{1}{4} \sum_{x \in \mathbf{X}} (\hat{y}(x) - y(x))^2$$

- If we minimize $J(\mathbf{w})$, we obtain $w_1 = 0$, $w_2 = 0$, and $b = \frac{1}{2}$.

Perceptron Weakness (2/2)



$$\mathbf{x} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \hat{y} = \text{step}(z), z = w_1x_1 + w_2x_2 + b$$

$$J(\mathbf{w}) = \frac{1}{4} \sum_{x \in \mathbf{X}} (\hat{y}(x) - y(x))^2$$

- ▶ If we minimize $J(\mathbf{w})$, we obtain $w_1 = 0$, $w_2 = 0$, and $b = \frac{1}{2}$.
- ▶ But, the model outputs **0.5 everywhere**.

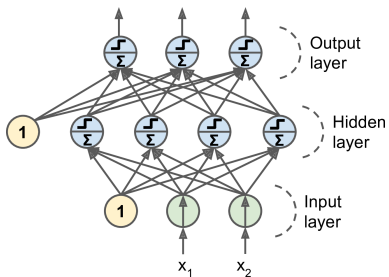


Multi-Layer Perceptron (MLP)

- ▶ The **limitations** of Perceptrons can be eliminated by **stacking multiple Perceptrons**.
- ▶ The resulting network is called a **Multi-Layer Perceptron (MLP)** or **deep feedforward neural network**.

Feedforward Neural Network Architecture

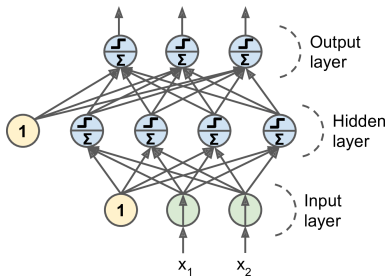
- ▶ A **feedforward neural network** is composed of:
 - One **input layer**
 - One or more **hidden layers**
 - One final **output layer**



Feedforward Neural Network Architecture

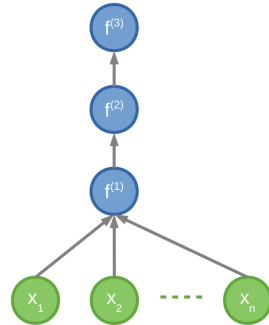
- ▶ A **feedforward neural network** is composed of:
 - One **input layer**
 - One or more **hidden layers**
 - One final **output layer**

- ▶ Every layer except the output layer includes a **bias neuron** and is **fully connected** to the **next layer**.



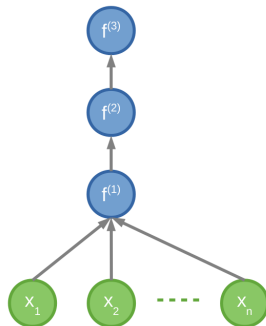
How Does it Work?

- ▶ The **model** is associated with a **directed acyclic graph** describing how the functions are **composed together**.



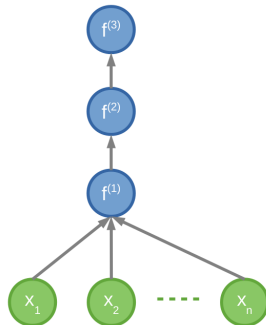
How Does it Work?

- ▶ The **model** is associated with a **directed acyclic graph** describing how the functions are **composed together**.
- ▶ E.g., assume a network with just a **single neuron** in **each layer**.
- ▶ Also assume we have **three functions** $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain: $\hat{y} = f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$



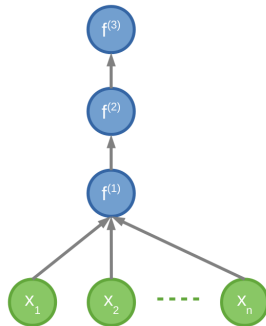
How Does it Work?

- ▶ The **model** is associated with a **directed acyclic graph** describing how the functions are **composed together**.
- ▶ E.g., assume a network with just a **single neuron** in **each layer**.
- ▶ Also assume we have **three functions** $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain: $\hat{y} = f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$
- ▶ $f^{(1)}$ is called the **first layer** of the network.
- ▶ $f^{(2)}$ is called the **second layer**, and so on.

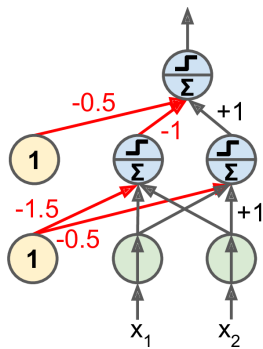


How Does it Work?

- ▶ The **model** is associated with a **directed acyclic graph** describing how the functions are **composed together**.
- ▶ E.g., assume a network with just a **single neuron** in **each layer**.
- ▶ Also assume we have **three functions** $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain: $\hat{y} = f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$
- ▶ $f^{(1)}$ is called the **first layer** of the network.
- ▶ $f^{(2)}$ is called the **second layer**, and so on.
- ▶ The **length of the chain** gives the **depth of the model**.



XOR with Feedforward Neural Network (1/3)



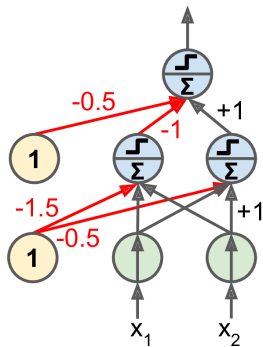
$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$$\mathbf{W}_x = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{b}_x = \begin{bmatrix} -1.5 \\ -0.5 \end{bmatrix}$$

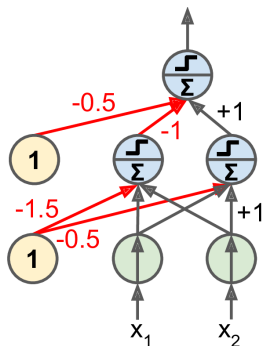
XOR with Feedforward Neural Network (2/3)



$$\text{out}_h = \mathbf{XW}_x^T + \mathbf{b}_x = \begin{bmatrix} -1.5 & -0.5 \\ -0.5 & 0.5 \\ -0.5 & 0.5 \\ 0.5 & 1.5 \end{bmatrix} \quad \mathbf{h} = \text{step}(\text{out}_h) = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{w}_h = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad \mathbf{b}_h = -0.5$$

XOR with Feedforward Neural Network (3/3)



$$\text{out} = \mathbf{w}_h^T \mathbf{h} + b_h = \begin{bmatrix} -0.5 \\ 0.5 \\ 0.5 \\ -0.5 \end{bmatrix} \quad \text{step}(\text{out}) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$



How to Learn Model Parameters \mathbf{W} ?



Feedforward Neural Network - Cost Function

- ▶ We use the **cross-entropy** (minimizing the negative log-likelihood) between the training data \mathbf{y} and the model's predictions $\hat{\mathbf{y}}$ as the **cost function**.

$$\text{cost}(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_j y_j \log(\hat{y}_j)$$

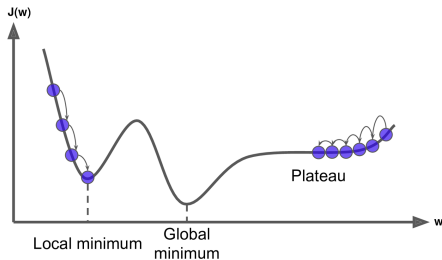


Gradient-Based Learning (1/2)

- ▶ The **most significant difference** between the **linear models** we have seen so far and **feedforward neural network**?

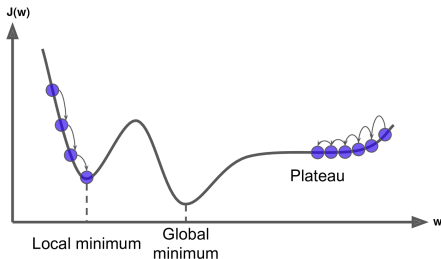
Gradient-Based Learning (1/2)

- ▶ The **most significant difference** between the **linear models** we have seen so far and **feedforward neural network**?
- ▶ The **non-linearity** of a neural network causes its **cost functions** to become **non-convex**.



Gradient-Based Learning (1/2)

- ▶ The **most significant difference** between the **linear models** we have seen so far and **feedforward neural network**?
- ▶ The **non-linearity** of a neural network causes its **cost functions** to become **non-convex**.
- ▶ Linear models, with **convex cost function**, **guarantee** to find **global minimum**.
 - Convex optimization converges starting from **any initial parameters**.





Gradient-Based Learning (2/2)

- ▶ Stochastic gradient descent applied to **non-convex cost functions** has no such convergence guarantee.



Gradient-Based Learning (2/2)

- ▶ Stochastic gradient descent applied to **non-convex cost functions** has no such convergence guarantee.
- ▶ It is **sensitive** to the values of the **initial parameters**.



Gradient-Based Learning (2/2)

- ▶ Stochastic gradient descent applied to **non-convex cost functions** has no such convergence guarantee.
- ▶ It is **sensitive** to the values of the **initial parameters**.
- ▶ For **feedforward neural networks**, it is important to **initialize** all **weights** to small random values.

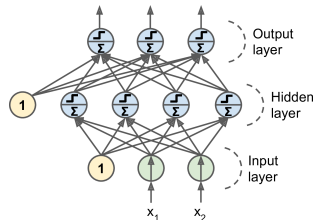


Gradient-Based Learning (2/2)

- ▶ Stochastic gradient descent applied to **non-convex cost functions** has no such convergence guarantee.
- ▶ It is **sensitive** to the values of the **initial parameters**.
- ▶ For **feedforward neural networks**, it is important to **initialize** all **weights** to small random values.
- ▶ The **biases** may be **initialized** to **zero** or to small **positive values**.

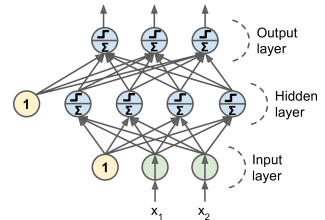
Training Feedforward Neural Networks

- ▶ How to train a feedforward neural network?



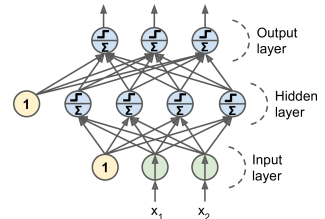
Training Feedforward Neural Networks

- ▶ How to **train** a **feedforward neural network**?
- ▶ For each training instance $\mathbf{x}^{(i)}$ the algorithm does the following **steps**:



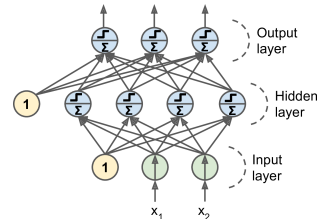
Training Feedforward Neural Networks

- ▶ How to **train** a **feedforward neural network**?
- ▶ For each training instance $\mathbf{x}^{(i)}$ the algorithm does the following **steps**:
 1. **Forward pass**: make a **prediction** (compute $\hat{\mathbf{y}}^{(i)} = \mathbf{f}(\mathbf{x}^{(i)})$).



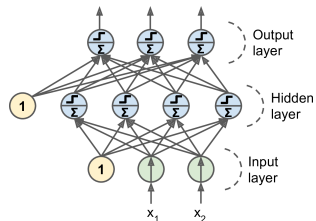
Training Feedforward Neural Networks

- ▶ How to **train** a **feedforward neural network**?
- ▶ For each training instance $\mathbf{x}^{(i)}$ the algorithm does the following **steps**:
 1. **Forward pass**: make a **prediction** (compute $\hat{y}^{(i)} = f(\mathbf{x}^{(i)})$).
 2. Measure the **error** (compute $\text{cost}(\hat{y}^{(i)}, y^{(i)})$).



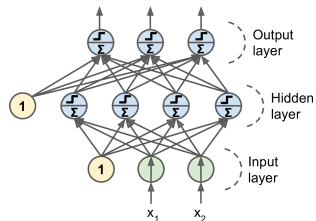
Training Feedforward Neural Networks

- ▶ How to **train** a **feedforward neural network**?
- ▶ For each training instance $\mathbf{x}^{(i)}$ the algorithm does the following **steps**:
 1. **Forward pass**: make a **prediction** (compute $\hat{\mathbf{y}}^{(i)} = \mathbf{f}(\mathbf{x}^{(i)})$).
 2. Measure the **error** (compute $\text{cost}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$).
 3. **Backward pass**: go through each layer in **reverse** to measure the **error contribution** from **each connection**.



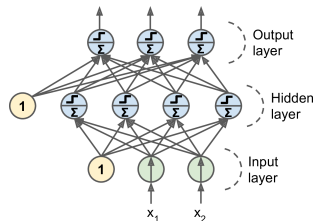
Training Feedforward Neural Networks

- ▶ How to **train** a **feedforward neural network**?
- ▶ For each training instance $\mathbf{x}^{(i)}$ the algorithm does the following **steps**:
 1. **Forward pass**: make a **prediction** (compute $\hat{\mathbf{y}}^{(i)} = \mathbf{f}(\mathbf{x}^{(i)})$).
 2. Measure the **error** (compute $\text{cost}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$).
 3. **Backward pass**: go through each layer in **reverse** to measure the **error contribution** from **each connection**.
 4. **Tweak the connection weights** to **reduce the error** (update \mathbf{W} and \mathbf{b}).



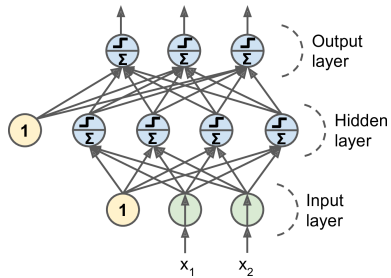
Training Feedforward Neural Networks

- ▶ How to **train** a **feedforward neural network**?
- ▶ For each training instance $\mathbf{x}^{(i)}$ the algorithm does the following **steps**:
 1. **Forward pass**: make a **prediction** (compute $\hat{\mathbf{y}}^{(i)} = \mathbf{f}(\mathbf{x}^{(i)})$).
 2. Measure the **error** (compute $\text{cost}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$).
 3. **Backward pass**: go through each layer in **reverse** to measure the **error contribution** from **each connection**.
 4. **Tweak the connection weights** to **reduce the error** (update \mathbf{W} and \mathbf{b}).
- ▶ It's called the **backpropagation** training algorithm



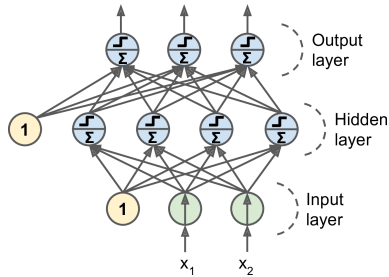
Output Unit (1/3)

- ▶ **Linear units** in neurons of the **output layer**.



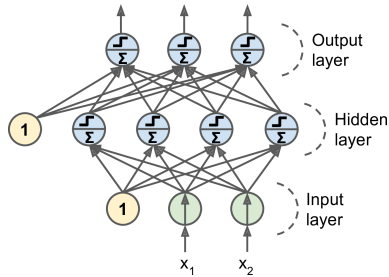
Output Unit (1/3)

- ▶ **Linear units** in neurons of the **output layer**.
- ▶ Given **\mathbf{h}** as the output of neurons in the **layer before the output layer**.
- ▶ Each neuron **j** in the **output layer** produces $\hat{y}_j = \mathbf{w}_j^\top \mathbf{h} + b_j$.



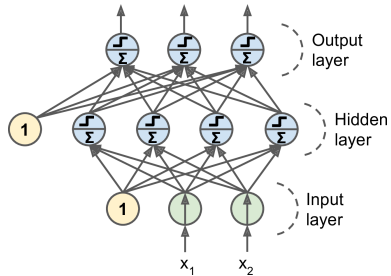
Output Unit (1/3)

- ▶ **Linear units** in neurons of the **output layer**.
- ▶ Given **\mathbf{h}** as the output of neurons in the **layer before the output layer**.
- ▶ Each neuron **j** in the **output layer** produces $\hat{y}_j = \mathbf{w}_j^\top \mathbf{h} + b_j$.
- ▶ **Minimizing the cross-entropy** is then equivalent to **minimizing the mean squared error**.



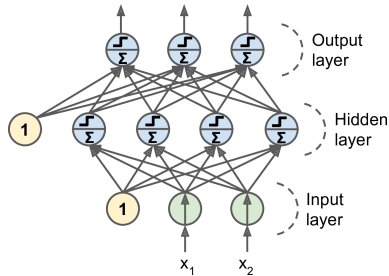
Output Unit (2/3)

- **Sigmoid units** in neurons of the **output layer** (**binomial** classification).



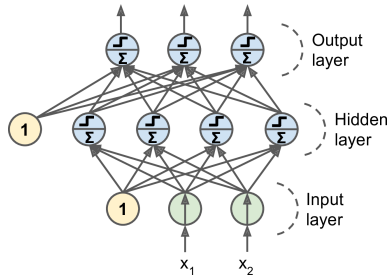
Output Unit (2/3)

- ▶ **Sigmoid units** in neurons of the **output layer** (**binomial** classification).
- ▶ Given **\mathbf{h}** as the output of neurons in the **layer before the output layer**.
- ▶ Each neuron **j** in the **output layer** produces $\hat{y}_j = \sigma(\mathbf{w}_j^T \mathbf{h} + b_j)$.



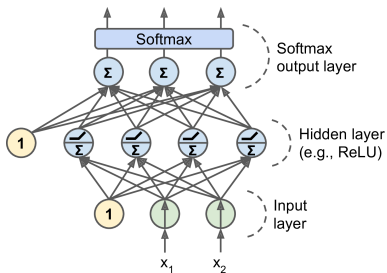
Output Unit (2/3)

- ▶ **Sigmoid units** in neurons of the **output layer** (**binomial** classification).
- ▶ Given **\mathbf{h}** as the output of neurons in the **layer before the output layer**.
- ▶ Each neuron **j** in the **output layer** produces $\hat{y}_j = \sigma(\mathbf{w}_j^T \mathbf{h} + b_j)$.
- ▶ **Minimizing the cross-entropy.**



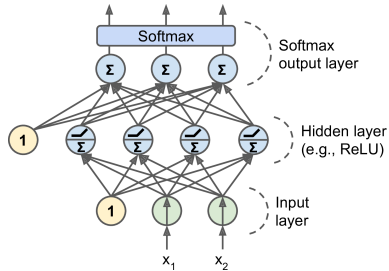
Output Unit (3/3)

- **Softmax units** in neurons of the **output layer** (**multinomial** classification).



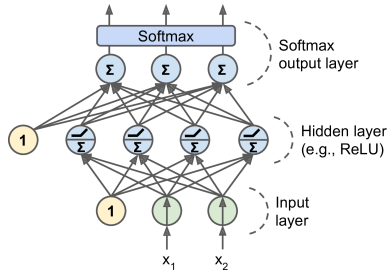
Output Unit (3/3)

- ▶ **Softmax units** in neurons of the **output layer** (**multinomial** classification).
- ▶ Given **\mathbf{h}** as the output of neurons in the **layer before the output layer**.
- ▶ Each neuron **j** in the **output layer** produces $\hat{y}_j = \text{softmax}(\mathbf{w}_j^T \mathbf{h} + b_j)$.



Output Unit (3/3)

- ▶ **Softmax units** in neurons of the **output layer** (**multinomial** classification).
- ▶ Given **\mathbf{h}** as the output of neurons in the **layer before the output layer**.
- ▶ Each neuron **j** in the **output layer** produces $\hat{y}_j = \text{softmax}(\mathbf{w}_j^T \mathbf{h} + b_j)$.
- ▶ **Minimizing the cross-entropy.**



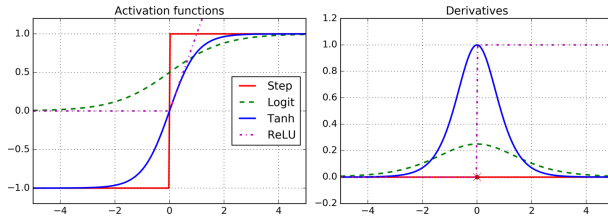


Hidden Units

- ▶ In order for the **backpropagation** algorithm to work properly, we need to **replace the step function** with **other activation functions**. **Why?**

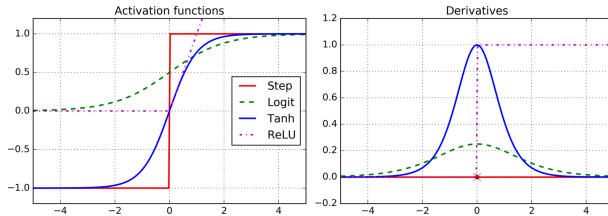
Hidden Units

- ▶ In order for the **backpropagation** algorithm to work properly, we need to **replace the step function** with **other activation functions**. **Why?**
- ▶ Alternative activation functions:



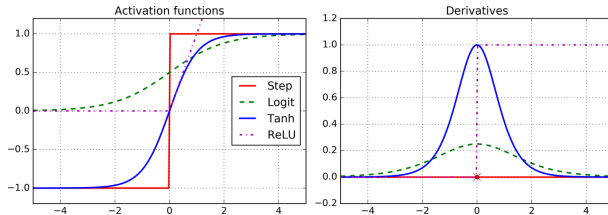
Hidden Units

- ▶ In order for the **backpropagation** algorithm to work properly, we need to **replace the step function** with **other activation functions**. **Why?**
- ▶ Alternative activation functions:
 1. **Logistic function (sigmoid)**: $\sigma(z) = \frac{1}{1+e^{-z}}$



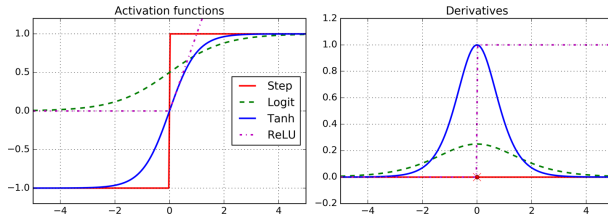
Hidden Units

- ▶ In order for the **backpropagation** algorithm to work properly, we need to **replace the step function** with **other activation functions**. **Why?**
- ▶ Alternative activation functions:
 1. **Logistic function (sigmoid)**: $\sigma(z) = \frac{1}{1+e^{-z}}$
 2. **Hyperbolic tangent function**: $\tanh(z) = 2\sigma(2z) - 1$



Hidden Units

- ▶ In order for the **backpropagation** algorithm to work properly, we need to **replace the step function** with **other activation functions**. **Why?**
- ▶ Alternative activation functions:
 1. **Logistic function (sigmoid)**: $\sigma(z) = \frac{1}{1+e^{-z}}$
 2. **Hyperbolic tangent function**: $\tanh(z) = 2\sigma(2z) - 1$
 3. **Rectified linear units (ReLU)**: $\text{ReLU}(z) = \max(0, z)$





Feedforward Network in TensorFlow



Feedforward in TensorFlow - First Implementation (1/3)

- ▶ `n_neurons_h`: number of neurons in the hidden layer.
- ▶ `n_neurons_out`: number of neurons in the output layer.
- ▶ `n_features`: number of features.

```
n_neurons_h = 4
n_neurons_out = 3
n_features = 2

# placeholder
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
y_true = tf.placeholder(tf.int64, shape=(None), name="y")

# variables
W1 = tf.get_variable("weights1", dtype=tf.float32,
                    initializer=tf.zeros((n_features, n_neurons_h)))
b1 = tf.get_variable("bias1", dtype=tf.float32, initializer=tf.zeros((n_neurons_h)))

W2 = tf.get_variable("weights2", dtype=tf.float32,
                    initializer=tf.zeros((n_neurons_h, n_neurons_out)))
b2 = tf.get_variable("bias2", dtype=tf.float32, initializer=tf.zeros((n_neurons_out)))
```

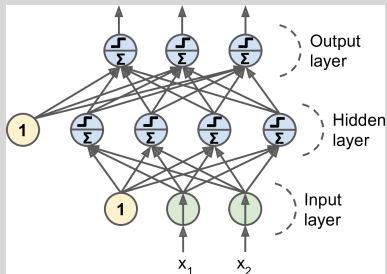
Feedforward in TensorFlow - First Implementation (2/3)

- Build the network.

```
# make the network
h = tf.nn.sigmoid(tf.matmul(X, W1) + b1)
z = tf.matmul(h, W2) + b2
y_hat = tf.nn.sigmoid(z)

# define the cost
cross_entropy =
    tf.nn.sigmoid_cross_entropy_with_logits(z, y_true)
cost = tf.reduce_mean(cross_entropy)

# train the model
learning_rate = 0.1
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
training_op = optimizer.minimize(cost)
```





Feedforward in TensorFlow - First Implementation (3/3)

- ▶ Execute the network.

```
# execute the model
init = tf.global_variables_initializer()

n_epochs = 100
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        sess.run(training_op, feed_dict={X: training_X, y_true: training_y})
```

Feedforward in TensorFlow - Second Implementation

```

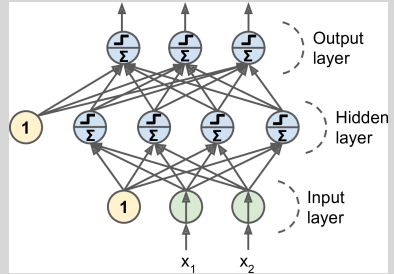
n_neurons_h = 4
n_neurons_out = 3
n_features = 2

# placeholder
X = tf.placeholder(tf.float32, shape=(None, n_features),
                  name="X")
y_true = tf.placeholder(tf.int64, shape=(None),
                       name="y")

# make the network
h = tf.layers.dense(X, n_neurons_h, name="hidden",
                   activation=tf.sigmoid)
z = tf.layers.dense(h, n_neurons_out, name="output")

# the rest as before

```





Feedforward in Keras

```
n_neurons_h = 4
n_neurons_out = 3
n_epochs = 100
learning_rate = 0.1

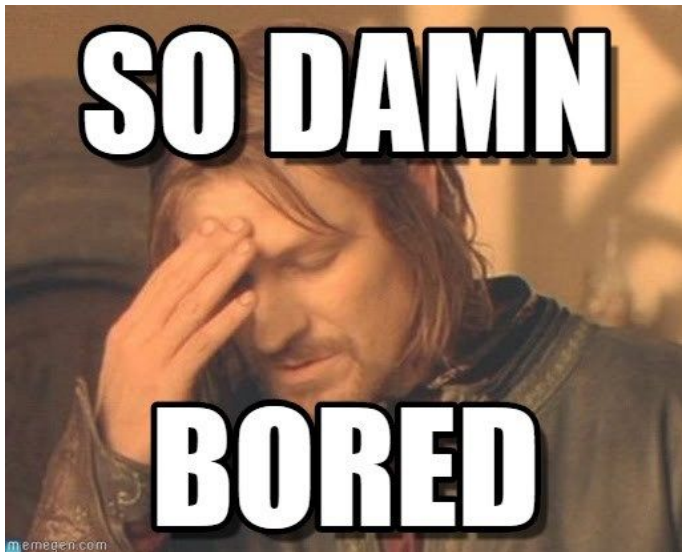
model = tf.keras.Sequential()
model.add(layers.Dense(n_neurons_h, activation="sigmoid"))
model.add(layers.Dense(n_neurons_out, activation="sigmoid"))

model.compile(optimizer=tf.train.GradientDescentOptimizer(learning_rate.001),
              loss="binary_crossentropy", metrics=["accuracy"])

model.fit(training_X, training_y, epochs=n_epochs)
```



Dive into Backpropagation Algorithm



[<https://i.pining.com/originals/82/d9/2c/82d92c2c15c580c2b2fce65a83fe0b3f.jpg>]



Chain Rule of Calculus (1/2)

- ▶ Assume $x \in \mathbb{R}$, and two functions f and g , and also assume $y = g(x)$ and $z = f(y) = f(g(x))$.



Chain Rule of Calculus (1/2)

- ▶ Assume $x \in \mathbb{R}$, and two functions f and g , and also assume $y = g(x)$ and $z = f(y) = f(g(x))$.
- ▶ The **chain rule of calculus** is used to compute the **derivatives of functions**, e.g., z , formed by **composing other functions**, e.g., g .



Chain Rule of Calculus (1/2)

- ▶ Assume $x \in \mathbb{R}$, and two functions f and g , and also assume $y = g(x)$ and $z = f(y) = f(g(x))$.
- ▶ The **chain rule of calculus** is used to compute the **derivatives of functions**, e.g., z , formed by **composing other functions**, e.g., g .
- ▶ Then the **chain rule** states that $\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$



Chain Rule of Calculus (1/2)

- ▶ Assume $x \in \mathbb{R}$, and two functions f and g , and also assume $y = g(x)$ and $z = f(y) = f(g(x))$.
- ▶ The **chain rule of calculus** is used to compute the **derivatives of functions**, e.g., z , formed by **composing other functions**, e.g., g .
- ▶ Then the **chain rule** states that $\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$
- ▶ Example:

$$z = f(y) = 5y^4 \text{ and } y = g(x) = x^3 + 7$$



Chain Rule of Calculus (1/2)

- ▶ Assume $x \in \mathbb{R}$, and two functions f and g , and also assume $y = g(x)$ and $z = f(y) = f(g(x))$.
- ▶ The **chain rule of calculus** is used to compute the **derivatives of functions**, e.g., z , formed by **composing other functions**, e.g., g .
- ▶ Then the **chain rule** states that $\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$
- ▶ Example:

$$z = f(y) = 5y^4 \text{ and } y = g(x) = x^3 + 7$$

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$



Chain Rule of Calculus (1/2)

- ▶ Assume $x \in \mathbb{R}$, and two functions f and g , and also assume $y = g(x)$ and $z = f(y) = f(g(x))$.
- ▶ The **chain rule of calculus** is used to compute the **derivatives of functions**, e.g., z , formed by **composing other functions**, e.g., g .
- ▶ Then the **chain rule** states that $\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$
- ▶ Example:

$$z = f(y) = 5y^4 \text{ and } y = g(x) = x^3 + 7$$

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

$$\frac{dz}{dy} = 20y^3 \text{ and } \frac{dy}{dx} = 3x^2$$



Chain Rule of Calculus (1/2)

- ▶ Assume $x \in \mathbb{R}$, and two functions f and g , and also assume $y = g(x)$ and $z = f(y) = f(g(x))$.
- ▶ The **chain rule of calculus** is used to compute the **derivatives of functions**, e.g., z , formed by **composing other functions**, e.g., g .
- ▶ Then the **chain rule** states that $\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$
- ▶ Example:

$$z = f(y) = 5y^4 \text{ and } y = g(x) = x^3 + 7$$

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

$$\frac{dz}{dy} = 20y^3 \text{ and } \frac{dy}{dx} = 3x^2$$

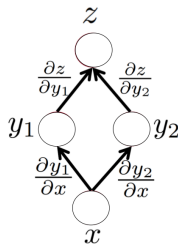
$$\frac{dz}{dx} = 20y^3 \times 3x^2 = 20(x^3 + 7) \times 3x^2$$

Chain Rule of Calculus (2/2)

- ▶ Two paths chain rule.

$$z = f(y_1, y_2) \text{ where } y_1 = g(x) \text{ and } y_2 = h(x)$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \frac{\partial y_2}{\partial x}$$



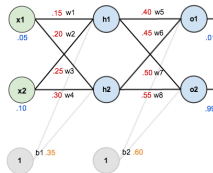


Backpropagation

- ▶ Backpropagation training algorithm for MLPs
- ▶ The algorithm repeats the following steps:
 1. Forward pass
 2. Backward pass

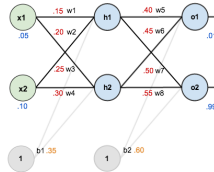
Backpropagation - Forward Pass

- ▶ Calculates outputs given input patterns.



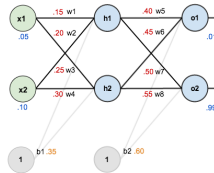
Backpropagation - Forward Pass

- ▶ Calculates outputs given input patterns.
- ▶ For each training instance



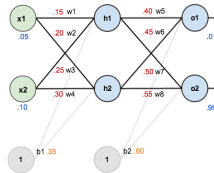
Backpropagation - Forward Pass

- ▶ Calculates outputs given input patterns.
- ▶ For each training instance
 - Feeds it to the network and computes the output of every neuron in each consecutive layer.



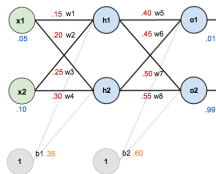
Backpropagation - Forward Pass

- ▶ Calculates outputs given input patterns.
- ▶ For each training instance
 - Feeds it to the network and computes the output of every neuron in each consecutive layer.
 - Measures the network's output error (i.e., the difference between the true and the predicted output of the network)



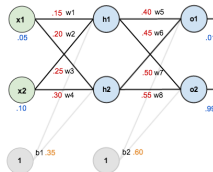
Backpropagation - Forward Pass

- ▶ Calculates outputs given input patterns.
- ▶ For each training instance
 - Feeds it to the network and computes the output of every neuron in each consecutive layer.
 - Measures the network's output error (i.e., the difference between the true and the predicted output of the network)
 - Computes how much each neuron in the last hidden layer contributed to each output neuron's error.



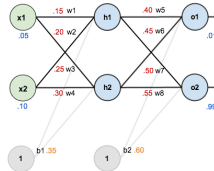
Backpropagation - Backward Pass

- Updates weights by calculating gradients.



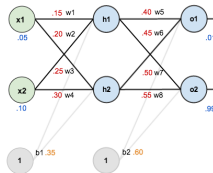
Backpropagation - Backward Pass

- ▶ Updates weights by calculating gradients.
- ▶ Measures how much of these error contributions came from each neuron in the previous hidden layer
 - Proceeds until the algorithm reaches the input layer.



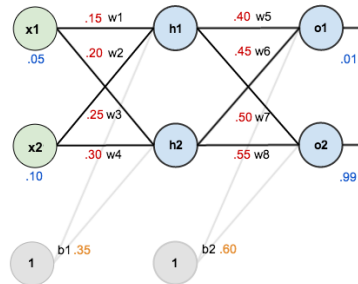
Backpropagation - Backward Pass

- ▶ Updates weights by calculating gradients.
- ▶ Measures how much of these error contributions came from each neuron in the previous hidden layer
 - Proceeds until the algorithm reaches the input layer.
- ▶ The last step is the gradient descent step on all the connection weights in the network, using the error gradients measured earlier.



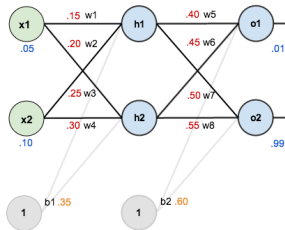
Backpropagation Example

- ▶ Two **inputs**, two **hidden**, and two **output** neurons.
- ▶ Bias in **hidden** and **output** neurons.
- ▶ Logistic activation in all the neurons.
- ▶ Squared error function as the cost function.



Backpropagation - Forward Pass (1/3)

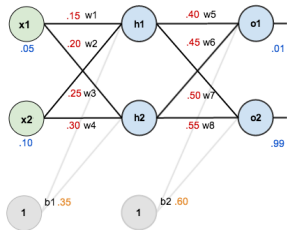
- ▶ Compute the **output** of the **hidden layer**



$$\text{net}_{h1} = w_1x_1 + w_2x_2 + b_1 = 0.15 \times 0.05 + 0.2 \times 0.1 + 0.35 = 0.3775$$

Backpropagation - Forward Pass (1/3)

- Compute the **output** of the **hidden layer**



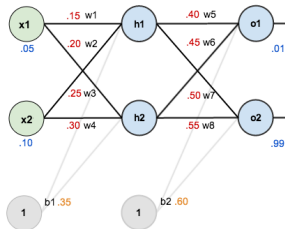
$$\text{net}_{h1} = w_1x_1 + w_2x_2 + b_1 = 0.15 \times 0.05 + 0.2 \times 0.1 + 0.35 = 0.3775$$

$$\text{out}_{h1} = \frac{1}{1 + e^{\text{net}_{h1}}} = \frac{1}{1 + e^{0.3775}} = 0.59327$$

$$\text{out}_{h2} = 0.59688$$

Backpropagation - Forward Pass (2/3)

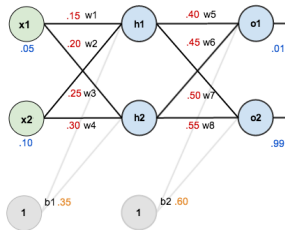
- ▶ Compute the **output** of the **output layer**



$$\text{net}_{o_1} = w_5 \text{out}_{h_1} + w_6 \text{out}_{h_2} + b_2 = 0.4 \times 0.59327 + 0.45 \times 0.59688 + 0.6 = 1.1059$$

Backpropagation - Forward Pass (2/3)

- Compute the **output** of the **output layer**



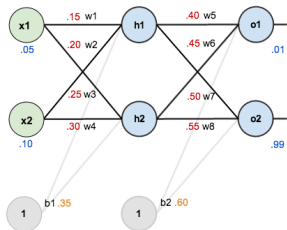
$$\text{net}_{o_1} = w_5 \text{out}_{h_1} + w_6 \text{out}_{h_2} + b_2 = 0.4 \times 0.59327 + 0.45 \times 0.59688 + 0.6 = 1.1059$$

$$\text{out}_{o_1} = \frac{1}{1 + e^{\text{net}_{o_1}}} = \frac{1}{1 + e^{1.1059}} = 0.75136$$

$$\text{out}_{o_2} = 0.77292$$

Backpropagation - Forward Pass (3/3)

- Calculate the **error** for each output



$$E_{o1} = \frac{1}{2}(\text{target}_{o1} - \text{output}_{o1})^2 = \frac{1}{2}(0.01 - 0.75136)^2 = 0.27481$$

$$E_{o2} = 0.02356$$

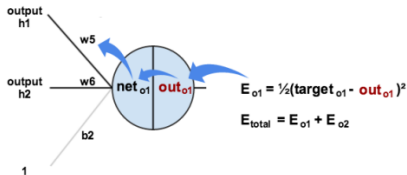
$$E_{\text{total}} = \sum \frac{1}{2}(\text{target} - \text{output})^2 = E_{o1} + E_{o2} = 0.27481 + 0.02356 = 0.29837$$



[<http://marimancusi.blogspot.com/2015/09/are-you-book-dragon.html>]

Backpropagation - Backward Pass - Output Layer (1/6)

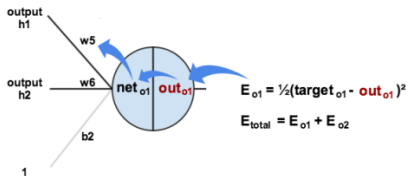
- ▶ Consider w_5
- ▶ We want to know how much a **change** in w_5 affects the **total error** ($\frac{\partial E_{total}}{\partial w_5}$)
- ▶ Applying the **chain rule**



$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial \text{out}_{o1}} \times \frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} \times \frac{\partial \text{net}_{o1}}{\partial w_5}$$

Backpropagation - Backward Pass - Output Layer (2/6)

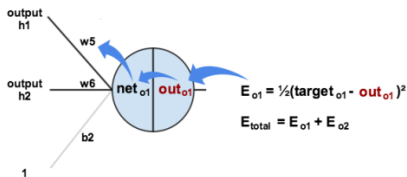
- ▶ First, how much does the **total error** change with **respect to the output**? ($\frac{\partial E_{total}}{\partial out_{o1}}$)



$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial w_5}$$

Backpropagation - Backward Pass - Output Layer (2/6)

- First, how much does the **total error** change with **respect to the output**? ($\frac{\partial E_{total}}{\partial out_{o1}}$)



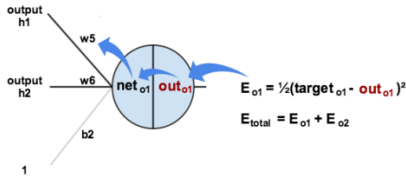
$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial w_5}$$

$$E_{total} = \frac{1}{2}(\text{target}_{o1} - \text{out}_{o1})^2 + \frac{1}{2}(\text{target}_{o2} - \text{out}_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -2 \frac{1}{2}(\text{target}_{o1} - \text{out}_{o1}) = -(0.01 - 0.75136) = 0.74136$$

Backpropagation - Backward Pass - Output Layer (3/6)

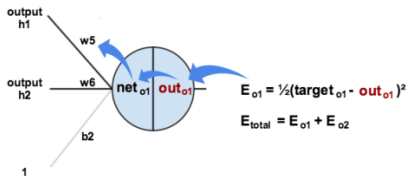
- ▶ Next, how much does the out_{o1} change with respect to its total input net_{o1} ?
 $(\frac{\partial out_{o1}}{\partial net_{o1}})$



$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial w_5}$$

Backpropagation - Backward Pass - Output Layer (3/6)

- Next, how much does the out_{o1} change with respect to its total input net_{o1} ?
 $\left(\frac{\partial out_{o1}}{\partial net_{o1}}\right)$



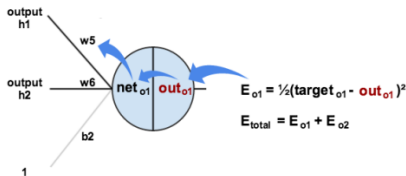
$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial w_5}$$

$$out_{o1} = \frac{1}{1 + e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136(1 - 0.75136) = 0.18681$$

Backpropagation - Backward Pass - Output Layer (4/6)

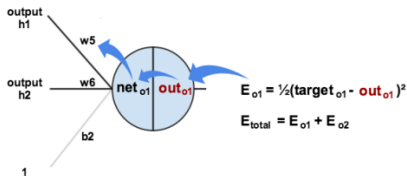
- ▶ Finally, how much does the total net_{o1} change with respect to w_5 ? ($\frac{\partial net_{o1}}{\partial w_5}$)



$$\frac{\partial E_{\text{total}}}{\partial w_5} = \frac{\partial E_{\text{total}}}{\partial \text{out}_{o1}} \times \frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} \times \frac{\partial \text{net}_{o1}}{\partial w_5}$$

Backpropagation - Backward Pass - Output Layer (4/6)

- ▶ Finally, how much does the total net_{o1} change with respect to w_5 ? ($\frac{\partial \text{net}_{o1}}{\partial w_5}$)



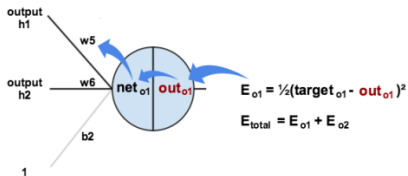
$$\frac{\partial E_{\text{total}}}{\partial w_5} = \frac{\partial E_{\text{total}}}{\partial \text{out}_{o1}} \times \frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} \times \frac{\partial \text{net}_{o1}}{\partial w_5}$$

$$\text{net}_{o1} = w_5 \times \text{out}_{h1} + w_6 \times \text{out}_{h2} + b_2$$

$$\frac{\partial \text{net}_{o1}}{\partial w_5} = \text{out}_{h1} = 0.59327$$

Backpropagation - Backward Pass - Output Layer (5/6)

- ▶ Putting it all together:

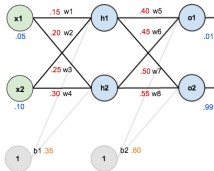


$$\frac{\partial E_{\text{total}}}{\partial w_5} = \frac{\partial E_{\text{total}}}{\partial \text{out}_{o1}} \times \frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} \times \frac{\partial \text{net}_{o1}}{\partial w_5}$$

$$\frac{\partial E_{\text{total}}}{\partial w_5} = 0.74136 \times 0.18681 \times 0.59327 = 0.08216$$

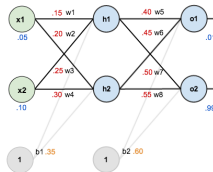
Backpropagation - Backward Pass - Output Layer (6/6)

- ▶ To decrease the error, we subtract this value from the current weight.



Backpropagation - Backward Pass - Output Layer (6/6)

- ▶ To **decrease the error**, we subtract this value from the **current weight**.
- ▶ We assume that the **learning rate** is $\eta = 0.5$.

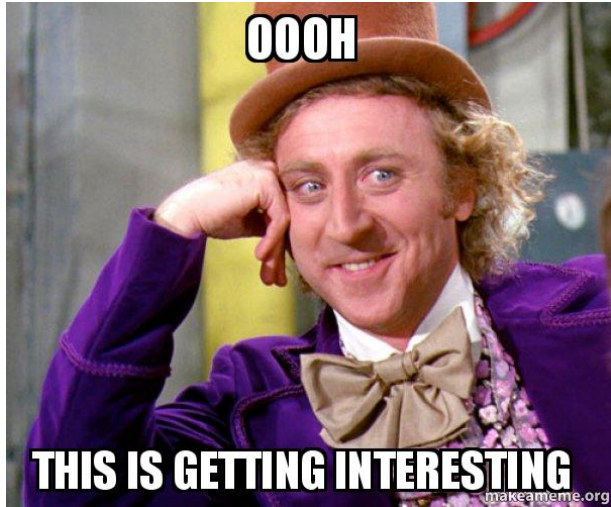


$$w_5^{(\text{next})} = w_5 - \eta \times \frac{\partial E_{\text{total}}}{\partial w_5} = 0.4 - 0.5 \times 0.08216 = 0.35891$$

$$w_6^{(\text{next})} = 0.40866$$

$$w_7^{(\text{next})} = 0.5113$$

$$w_8^{(\text{next})} = 0.56137$$

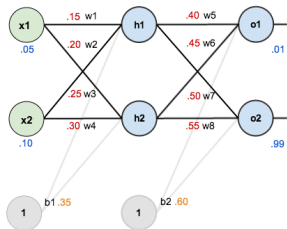


[<https://makeameme.org/meme/oooh-this>]

Backpropagation - Backward Pass - Hidden Layer (1/8)

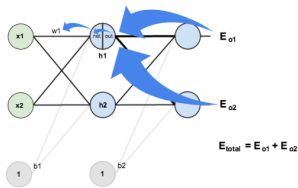
- ▶ Continue the **backwards pass** by calculating new values for w_1 , w_2 , w_3 , and w_4 .
- ▶ For w_1 we have:

$$\frac{\partial E_{\text{total}}}{\partial w_1} = \frac{\partial E_{\text{total}}}{\partial \text{out}_{h1}} \times \frac{\partial \text{out}_{h1}}{\partial \text{net}_{h1}} \times \frac{\partial \text{net}_{h1}}{\partial w_1}$$



Backpropagation - Backward Pass - Hidden Layer (2/8)

- ▶ Here, the **output of each hidden layer neuron** contributes to the **output of multiple output neurons**.
- ▶ E.g., out_{h1} affects both out_{o1} and out_{o2} , so $\frac{\partial E_{total}}{\partial out_{h1}}$ needs to take into consideration its effect on the both output neurons.

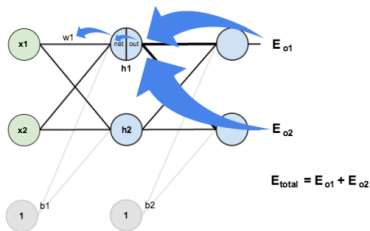


$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} \times \frac{\partial out_{h1}}{\partial net_{h1}} \times \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

Backpropagation - Backward Pass - Hidden Layer (3/8)

► Starting with $\frac{\partial E_{o1}}{\partial out_{h1}}$



$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial out_{h1}}$$

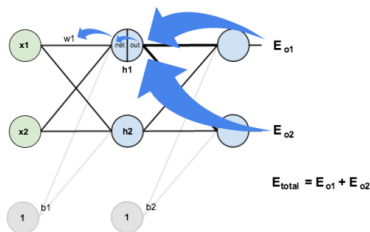
$$\frac{\partial E_{o1}}{\partial out_{o1}} = 0.74136, \quad \frac{\partial out_{o1}}{\partial net_{o1}} = 0.18681$$

$$net_{o1} = w_5 \times out_{h1} + w_6 \times out_{h2} + b_2$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

Backpropagation - Backward Pass - Hidden Layer (4/8)

- ▶ Plugging them together.

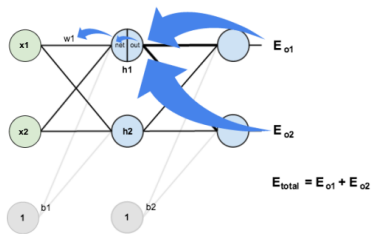


$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial out_{h1}} = 0.74136 \times 0.18681 \times 0.40 = 0.0554$$

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.01905$$

Backpropagation - Backward Pass - Hidden Layer (4/8)

- ▶ Plugging them together.



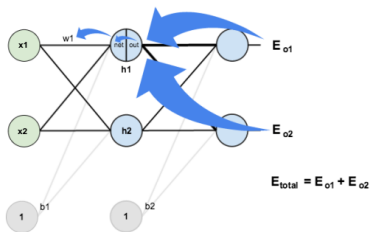
$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial out_{h1}} = 0.74136 \times 0.18681 \times 0.40 = 0.0554$$

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.01905$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.0554 + -0.01905 = 0.03635$$

Backpropagation - Backward Pass - Hidden Layer (5/8)

- ▶ Now we need to figure out $\frac{\partial out_{h1}}{\partial net_{h1}}$.



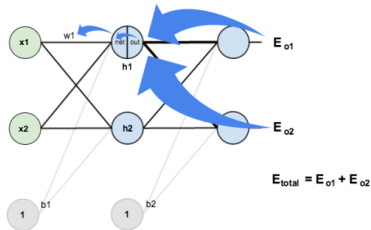
$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} \times \frac{\partial out_{h1}}{\partial net_{h1}} \times \frac{\partial net_{h1}}{\partial w_1}$$

$$out_{h1} = \frac{1}{1 + e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59327(1 - 0.59327) = 0.2413$$

Backpropagation - Backward Pass - Hidden Layer (6/8)

- And then $\frac{\partial \text{net}_{h1}}{\partial w_1}$.



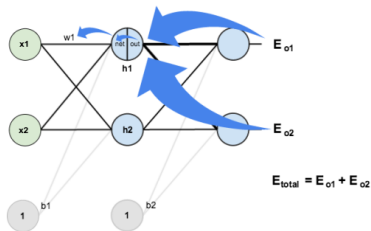
$$\frac{\partial E_{\text{total}}}{\partial w_1} = \frac{\partial E_{\text{total}}}{\partial \text{out}_{h1}} \times \frac{\partial \text{out}_{h1}}{\partial \text{net}_{h1}} \times \frac{\partial \text{net}_{h1}}{\partial w_1}$$

$$\text{net}_{h1} = w_1 x_1 + w_2 x_2 + b_1$$

$$\frac{\partial \text{net}_{h1}}{\partial w_1} = x_1 = 0.05$$

Backpropagation - Backward Pass - Hidden Layer (7/8)

- ▶ Putting it all together.

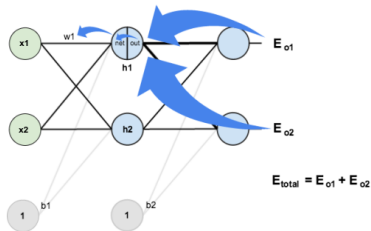


$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} \times \frac{\partial out_{h1}}{\partial net_{h1}} \times \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.03635 \times 0.2413 \times 0.05 = 0.00043$$

Backpropagation - Backward Pass - Hidden Layer (8/8)

- ▶ We can now update w_1 .
- ▶ Repeating this for w_2 , w_3 , and w_4 .



$$w_1^{(next)} = w_1 - \eta \times \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 \times 0.00043 = 0.14978$$

$$w_2^{(next)} = 0.19956$$

$$w_3^{(next)} = 0.24975$$

$$w_4^{(next)} = 0.2995$$

Summary



Summary

- ▶ LTU
- ▶ Perceptron
- ▶ Perceptron weakness
- ▶ MLP and feedforward neural network
- ▶ Gradient-based learning
- ▶ Backpropagation: forward pass and backward pass
- ▶ Output unit: linear, sigmoid, softmax
- ▶ Hidden units: sigmoid, tanh, relu



Reference

- ▶ Ian Goodfellow et al., Deep Learning (Ch. 6)
- ▶ Aurélien Géron, Hands-On Machine Learning (Ch. 10)

Questions?