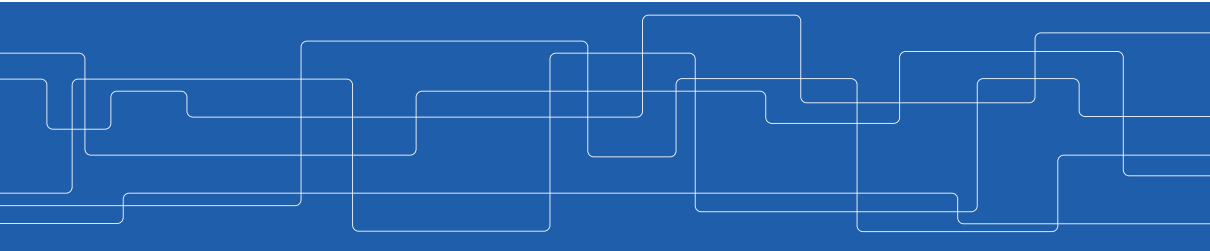

# Deep Learning for Poets (Part III)

Amir H. Payberah
payberah@kth.se
20/12/2018

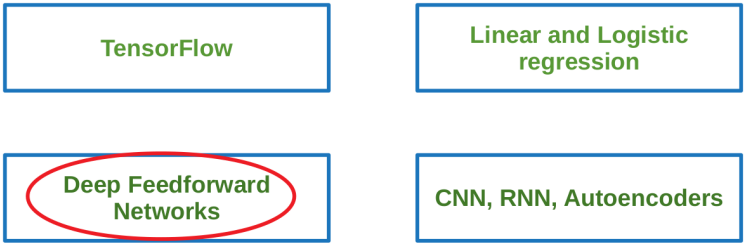



TensorFlow

Linear and Logistic regression

Deep Feedforward Networks

CNN, RNN, Autoencoders


TensorFlow
Linear and Logistic regression
Deep Feedforward Networks
CNN, RNN, Autoencoders

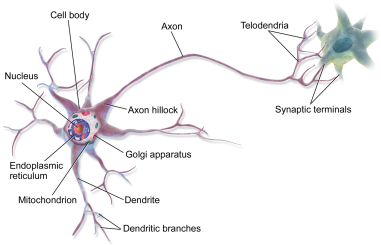
# Nature ...

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



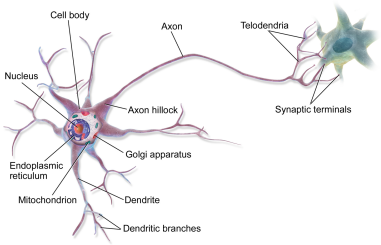

# Biological Neurons (1/2)

- Brain architecture has inspired artificial neural networks.



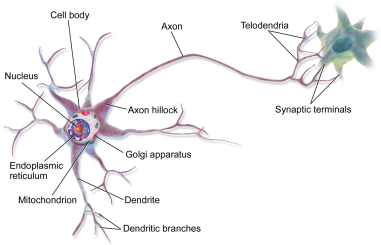
# Biological Neurons (1/2)

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



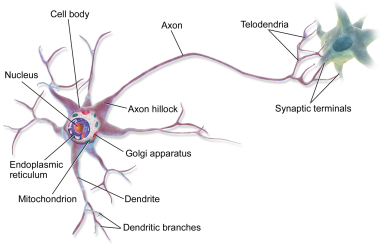
# Biological Neurons (1/2)

- 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.



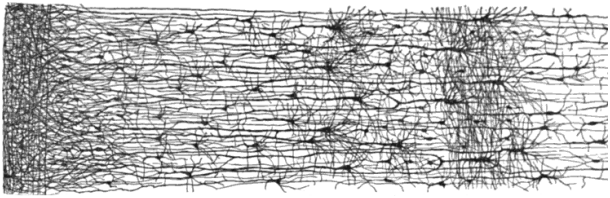
# Biological Neurons (1/2)

- 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.

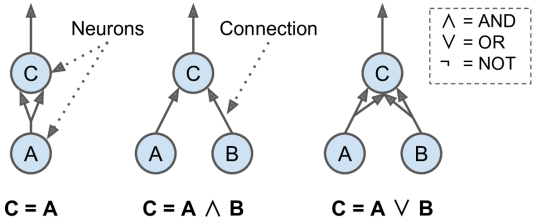



# A Simple Artificial Neural Network

- 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]

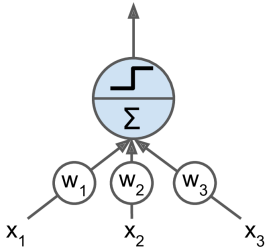

# The Linear Threshold Unit (LTU)

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

# The Linear Threshold Unit (LTU)

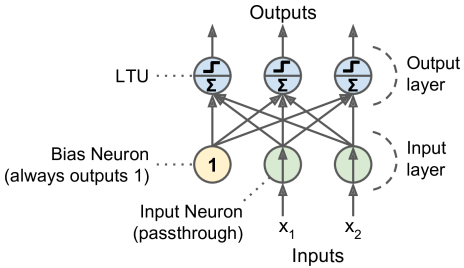
- Inputs of a LTU are numbers (not binary).
- Each input connection is associated with a weight.
- Computes a weighted sum of its inputs and applies a step function to that sum.

- $z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w}^\intercal \mathbf{x}$
- $\hat{y} = \text{step}(z) = \text{step}(\mathbf{w}^\intercal \mathbf{x})$



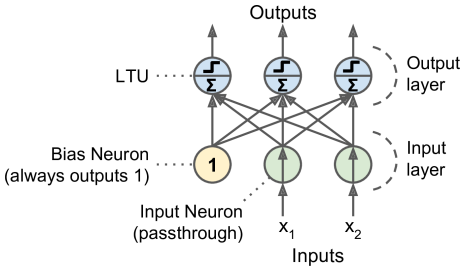
# The Perceptron

- The perceptron is a single layer of LTUs.



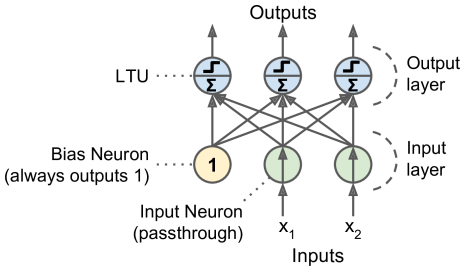
# The Perceptron

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



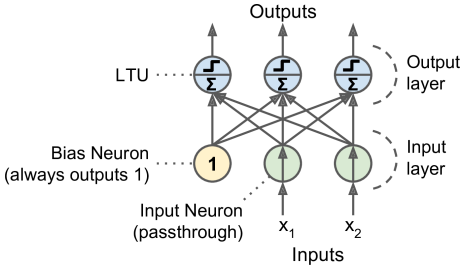
# The Perceptron

- 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.



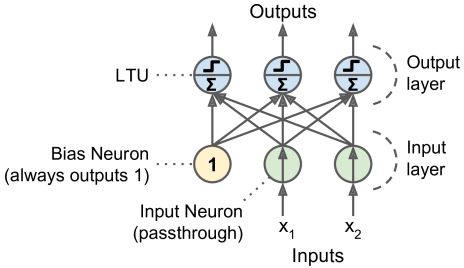
# The Perceptron

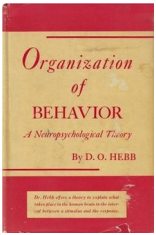
- The perceptron is a single layer of LTUs.
- The input neurons output whatever input they are fed.
- A bias neuron, which just outputs 1 all the time.
- If we use logistic function (sigmoid) instead of a step function, it computes a continuous output.



# How is a Perceptron Trained? (1/2)

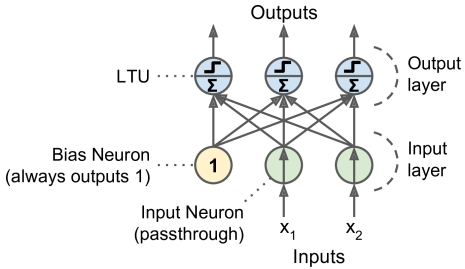
- The Perceptron training algorithm is inspired by Hebb's rule.

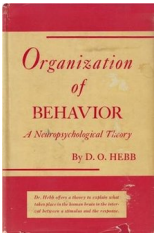


# How is a Perceptron Trained? (1/2)

- 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.



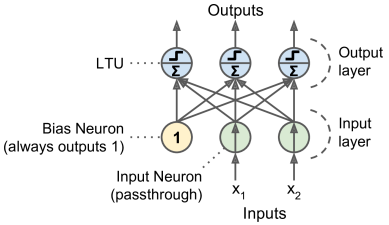
# How is a Perceptron Trained? (2/2)

- 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 $\mathtt{j}$ at a time and make its prediction $\hat{\mathtt{y}}$.
- Update the connection weights.

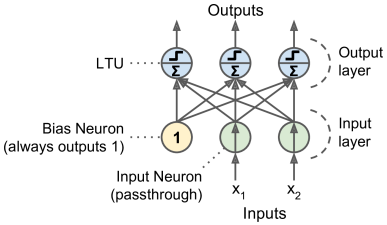
$$\hat{\mathtt{y}}_{\mathtt{j}} = \sigma(\mathbf{w}_{\mathtt{j}}^{\mathsf{T}}\mathbf{x} + \mathtt{b})$$
$$\mathtt{J}(\mathbf{w}_{\mathtt{j}}) = \mathtt{cross\_entropy}(\mathtt{y}_{\mathtt{j}}, \hat{\mathtt{y}}_{\mathtt{j}})$$
$$\mathtt{w}_{\mathtt{i},\mathtt{j}}^{(\mathtt{next})} = \mathtt{w}_{\mathtt{i},\mathtt{j}} - \eta \frac{\partial \mathtt{J}(\mathbf{w}_{\mathtt{j}})}{\mathtt{w}_{\mathtt{i}}}$$



# How is a Perceptron Trained? (2/2)

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

$$\hat{y}_j = \sigma(\mathbf{w}_j^\intercal \mathbf{x} + b)$$
$$J(\mathbf{w}_j) = \texttt{cross\_entropy}(y_j, \hat{y}_j)$$
$$w_{i,j}^{(\texttt{next})} = w_{i,j} - \eta \frac{\partial J(\mathbf{w}_j)}{w_i}$$



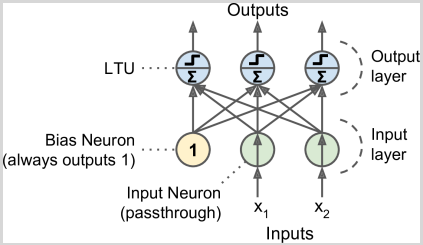

- $w_{i,j}$: the weight between neurons $i$ and $j$.
- $x_i$: the $i$th input value.
- $\hat{y}_j$: the $j$th predicted output value.
- $y_j$: the $j$th true output value.
- $\eta$: the learning rate.

# 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)
```

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

```
# 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)
```

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

```
# 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})
```

# Multi-Layer Perceptron (MLP)

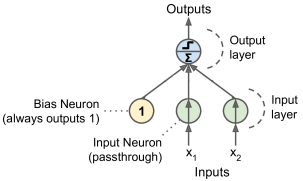
# Perceptron Weakness (1/2)

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

# Perceptron Weakness (1/2)

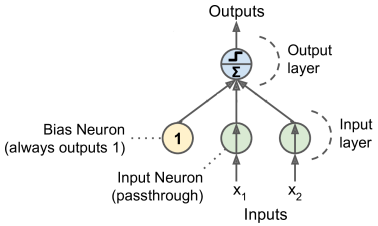
- Incapable of solving some trivial problems, e.g., XOR classification problem. Why?

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




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

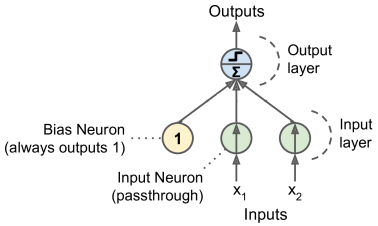
# Perceptron Weakness (2/2)




$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad \hat{y} = \texttt{step}(z), z = w_1 x_1 + w_2 x_2 + b$$

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

# Perceptron Weakness (2/2)

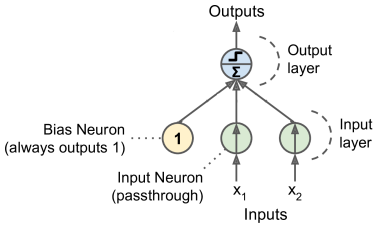



$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad \hat{y} = \mathtt{step}(z), z = w_1 x_1 + w_2 x_2 + b$$

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

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

# Perceptron Weakness (2/2)




$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad \hat{y} = \texttt{step}(z), z = w_1 x_1 + w_2 x_2 + b$$

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

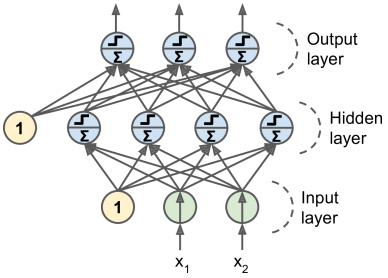
- If we minimize $J(\mathbf{w})$, we obtain $w_1 = 0$, $w_2 = 0$, and $b = \frac{1}{2}$.
- But, the model outputs 0.5 everywhere.


# 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.

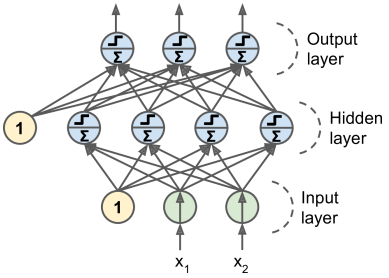
# Feedforward Neural Network Architecture

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



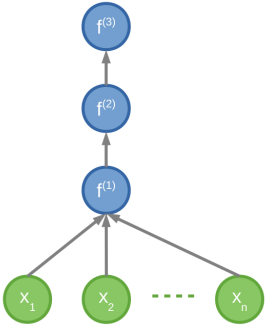
# Feedforward Neural Network Architecture

- 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.



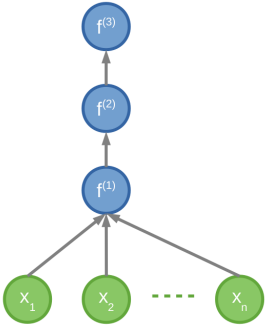
# How Does it Work?

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



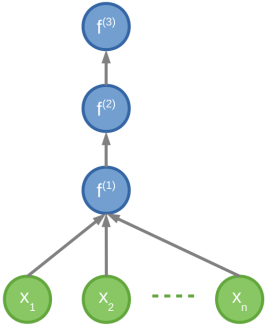
# How Does it Work?

- 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 $\mathbf{f}^{(1)}$ , $\mathbf{f}^{(2)}$, and $\mathbf{f}^{(3)}$ connected in a chain: $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}) = \mathbf{f}^{(3)}(\mathbf{f}^{(2)}(\mathbf{f}^{(1)}(\mathbf{x})))$



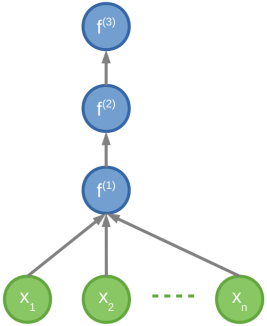
# How Does it Work?

- 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 $\mathbf{f}^{(1)}$ , $\mathbf{f}^{(2)}$, and $\mathbf{f}^{(3)}$ connected in a chain: $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}) = \mathbf{f}^{(3)}(\mathbf{f}^{(2)}(\mathbf{f}^{(1)}(\mathbf{x})))$
- $\mathbf{f}^{(1)}$ is called the first layer of the network.
- $\mathbf{f}^{(2)}$ is called the second layer, and so on.

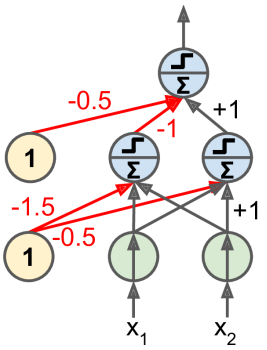


# How Does it Work?

- 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 $\mathbf{f}^{(1)}$ , $\mathbf{f}^{(2)}$, and $\mathbf{f}^{(3)}$ connected in a chain: $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}) = \mathbf{f}^{(3)}(\mathbf{f}^{(2)}(\mathbf{f}^{(1)}(\mathbf{x})))$
- $\mathbf{f}^{(1)}$ is called the first layer of the network.
- $\mathbf{f}^{(2)}$ is called the second layer, and so on.
- The length of the chain gives the depth of the model.

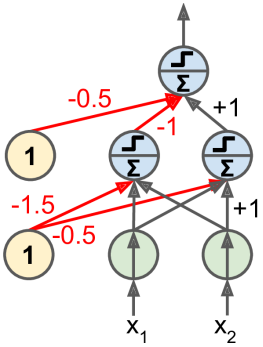


# XOR with Feedforward Neural Network (1/3)




$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad \mathbf{W}_x = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \mathbf{b}_x = \begin{bmatrix} -1.5 \\ -0.5 \end{bmatrix}$$

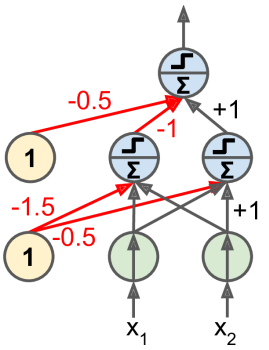
# XOR with Feedforward Neural Network (2/3)




$$\mathbf{out}_h = \mathbf{X}\mathbf{W}_x^T + \mathbf{b}_x = \begin{bmatrix} -1.5 & -0.5 \\ -0.5 & 0.5 \\ -0.5 & 0.5 \\ 0.5 & 1.5 \end{bmatrix} \qquad \mathbf{h} = \mathtt{step}(\mathbf{out}_h) = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{w}_h = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \qquad b_h = -0.5$$

# XOR with Feedforward Neural Network (3/3)




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

# 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 $\mathtt{y}$ and the model's predictions $\hat{\mathtt{y}}$ as the cost function.

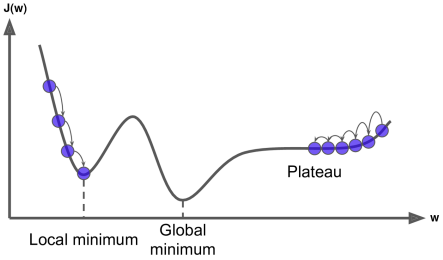
$$\mathtt{cost}(\mathtt{y}, \hat{\mathtt{y}}) = -\sum_{\mathtt{j}} \mathtt{y}_{\mathtt{j}} \mathtt{log}(\hat{\mathtt{y}}_{\mathtt{j}})$$

# Gradient-Based Learning (1/2)

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

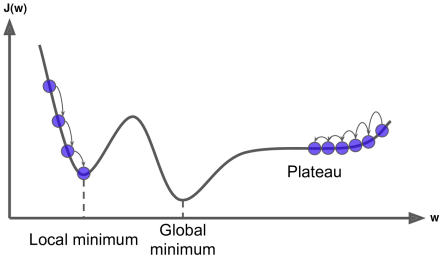
# Gradient-Based Learning (1/2)

- 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)

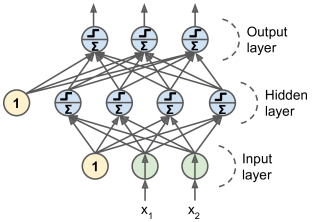
- Stochastic gradient descent applied to non-convex cost functions has no such convergence guarantee.
- It is sensitive to the values of the initial parameters.
- For feedforward neural networks, it is important to initialize all weights to small random values.

# 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.

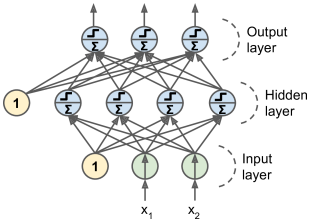
# Training Feedforward Neural Networks

- How to train a feedforward neural network?



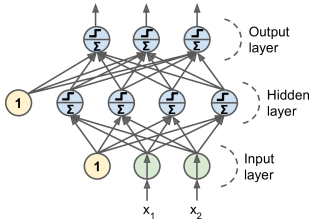
# Training Feedforward Neural Networks

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



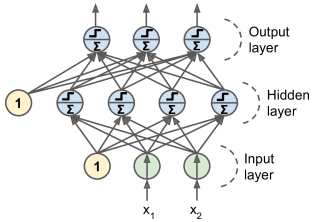
# Training Feedforward Neural Networks

- 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 $\hat{\mathbf{y}}^{(i)} = \mathbf{f}(\mathbf{x}^{(i)})$).



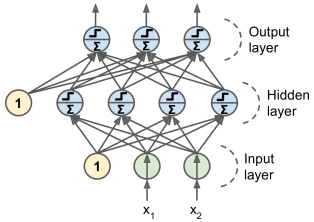
# Training Feedforward Neural Networks

- 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 $\hat{\mathbf{y}}^{(i)} = \mathtt{f}(\mathbf{x}^{(i)})$).
  2. Measure the error (compute $\mathtt{cost}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$).



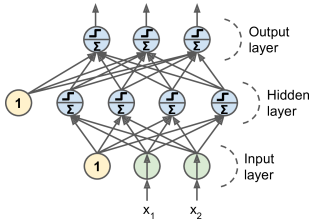
# Training Feedforward Neural Networks

- 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 $\hat{y}^{(i)} = f(\mathbf{x}^{(i)})$).
  2. Measure the error (compute $\texttt{cost}(\hat{y}^{(i)}, y^{(i)})$).
  3. Backward pass: go through each layer in reverse to measure the error contribution from each connection.



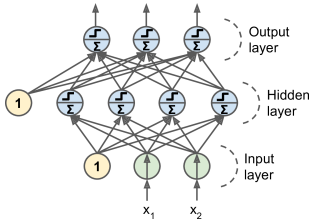
# Training Feedforward Neural Networks

- 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 $\hat{\mathbf{y}}^{(i)} = \mathtt{f}(\mathbf{x}^{(i)})$).
  2. Measure the error (compute $\mathtt{cost}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$).
  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 $\mathbf{b}$).



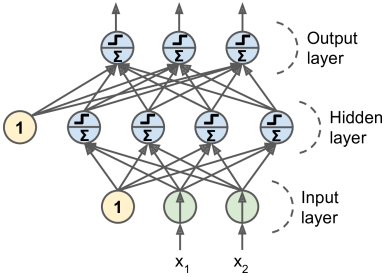
# Training Feedforward Neural Networks

- 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 $\hat{\mathbf{y}}^{(i)} = \mathtt{f}(\mathbf{x}^{(i)})$).
  2. Measure the error (compute $\mathtt{cost}(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$).
  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



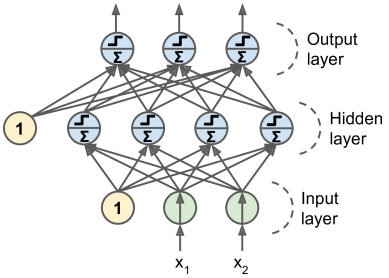
# Output Unit (1/3)

- Linear units in neurons of the output layer.



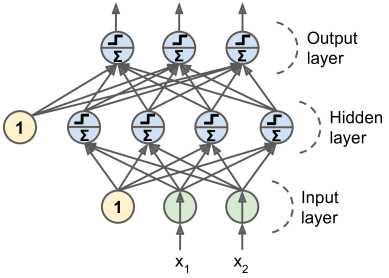
# Output Unit (1/3)

- 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 $\hat{y}_j = \mathbf{w}_j^T \mathbf{h} + b_j$.



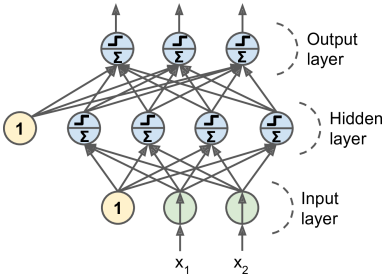
# Output Unit (1/3)

- 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 $\hat{y}_j = \mathbf{w}_j^T \mathbf{h} + b_j$.
- Minimizing the cross-entropy is then equivalent to minimizing the mean squared error.



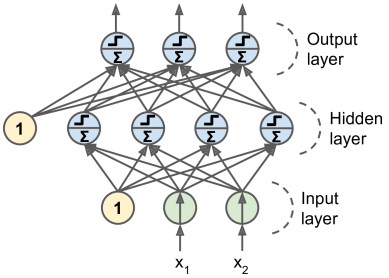
# Output Unit (2/3)

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



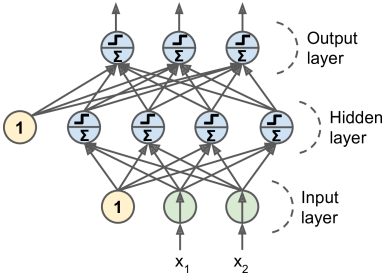
# Output Unit (2/3)

- 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 $\mathtt{j}$ in the output layer produces $\hat{\mathtt{y}}_\mathtt{j} = \sigma(\mathbf{w}_\mathtt{j}^\intercal \mathbf{h} + \mathtt{b}_\mathtt{j})$.



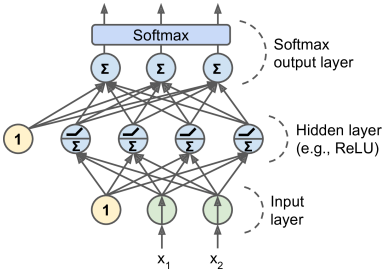
# Output Unit (2/3)

- 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.



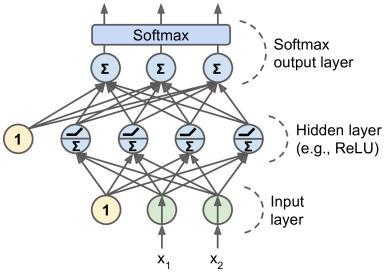
# Output Unit (3/3)

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



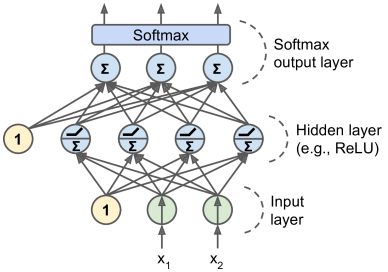
# Output Unit (3/3)

- 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 $\mathtt{j}$ in the output layer produces $\hat{\mathtt{y}}_\mathtt{j} = \mathtt{softmax}(\mathbf{w}_\mathtt{j}^\mathsf{T}\mathbf{h} + \mathtt{b}_\mathtt{j})$.



# 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 $\mathtt{j}$ in the output layer produces $\hat{\mathtt{y}}_\mathtt{j} = \mathtt{softmax}(\mathbf{w}_\mathtt{j}^\mathsf{T}\mathbf{h} + \mathtt{b}_\mathtt{j})$.
- Minimizing the cross-entropy.

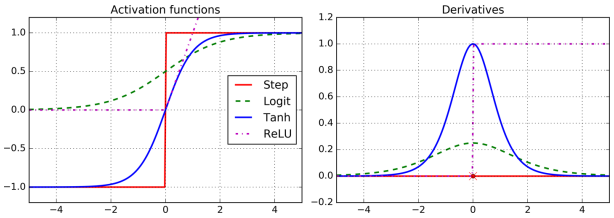


# Hidden Units

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

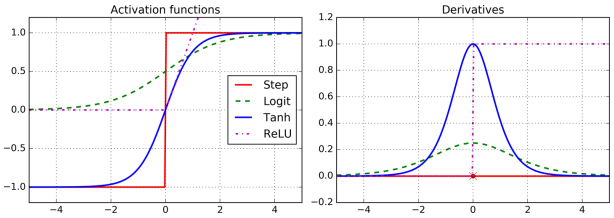
# Hidden Units

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



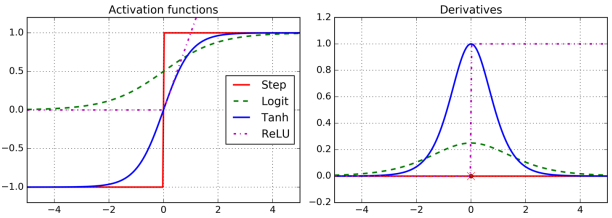
# Hidden Units

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



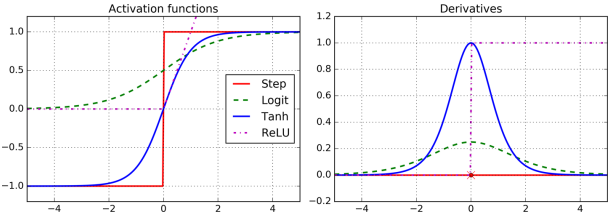
# Hidden Units

- 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 (ReLUs): $\mathrm{ReLU}(z) = \max(0, z)$




# Feedforward Network in TensorFlow

# Feedforward in TensorFlow - First Implementation (1/3)

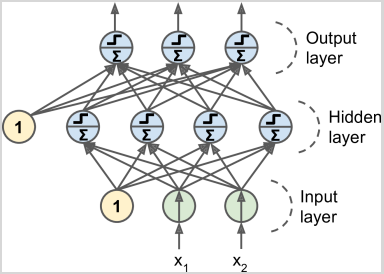
- `n_neurons_h`: number of neurons in the hidden layer.
- `n_neurons_out`: number of neurons in the output layer.
- `n_features`: number of features.

```
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.zero((n_neurons_h)))

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



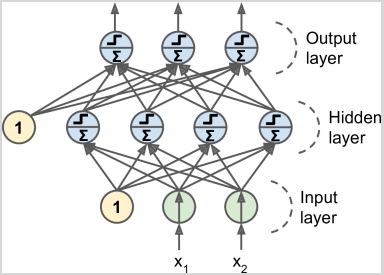
# Feedforward in TensorFlow - First Implementation (3/3)

- 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 Network in Keras


# Keras

- Keras is a high-level API to build and train deep learning models.
- To get started, import `tf.keras` to your program.

```
import tensorflow as tf
from tensorflow.keras import layers
```


# Keras Layers (1/2)

- In Keras, you assemble layers `tf.keras.layers` to build models.
- A model is (usually) a graph of layers.
- There are many types of layers, e.g., `Dense`, `Conv2D`, `RNN`, ...

# Keras Layers (2/2)

- Common constructor parameters:

```
layers.Dense(64, activation=tf.sigmoid, kernel_regularizer=tf.keras.regularizers.l1(0.01),
             bias_initializer=tf.keras.initializers.constant(2.0))
```

# Keras Layers (2/2)

- Common constructor parameters:
  - `activation`: the activation function for the layer.

```
layers.Dense(64, activation=tf.sigmoid, kernel_regularizer=tf.keras.regularizers.l1(0.01),
             bias_initializer=tf.keras.initializers.constant(2.0))
```

# Keras Layers (2/2)

- Common constructor parameters:
  - `activation`: the activation function for the layer.
  - `kernel_initializer` and `bias_initializer`: the initialization schemes of the layer's weights.

```
layers.Dense(64, activation=tf.sigmoid, kernel_regularizer=tf.keras.regularizers.l1(0.01),
             bias_initializer=tf.keras.initializers.constant(2.0))
```

# Keras Layers (2/2)

- Common constructor parameters:
  - `activation`: the activation function for the layer.
  - `kernel_initializer` and `bias_initializer`: the initialization schemes of the layer's weights.
  - `kernel_regularizer` and `bias_regularizer`: the regularization schemes of the layer's weights, e.g., L1 or L2.

```
layers.Dense(64, activation=tf.sigmoid, kernel_regularizer=tf.keras.regularizers.l1(0.01),
             bias_initializer=tf.keras.initializers.constant(2.0))
```


# Keras Models

- There are two ways to build Keras models: sequential and functional.


# Keras Models

- There are two ways to build Keras models: sequential and functional.
- The sequential API allows you to create models layer-by-layer.


# Keras Models

- There are two ways to build Keras models: sequential and functional.
- The sequential API allows you to create models layer-by-layer.
- The functional API allows you to create models that have a lot more flexibility.
  - You can define models where layers connect to more than just their previous and next layers.

# Keras Models - Sequential Models

- You can use `tf.keras.Sequential` to build a sequential model.

```
from tensorflow.keras import layers

model = tf.keras.Sequential()

model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(10, activation="softmax"))
```

# Keras Models - Functional Models

- You can use `tf.keras.Model` to build a functional model.

```
from tensorflow.keras import layers

inputs = tf.keras.Input(shape=(32,))
x = layers.Dense(64, activation="relu")(inputs)
x = layers.Dense(64, activation="relu")(x)
predictions = layers.Dense(10, activation="softmax")(x)

model = tf.keras.Model(inputs=inputs, outputs=predictions)
```

# Training Keras Models

- Call the `compile` method to configure the learning process.
- `tf.keras.Model.compile` takes three important arguments.

```
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="mse", metrics=["mae"])

model.fit(training_data, training_labels, epochs=10, batch_size=32)
```

# Training Keras Models

- Call the `compile` method to configure the learning process.
- `tf.keras.Model.compile` takes three important arguments.
  - `optimizer`: specifies the training procedure.

```
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="mse", metrics=["mae"])

model.fit(training_data, training_labels, epochs=10, batch_size=32)
```

# Training Keras Models

- Call the `compile` method to configure the learning process.
- `tf.keras.Model.compile` takes three important arguments.
  - `optimizer`: specifies the training procedure.
  - `loss`: the cost function to minimize during optimization, e.g., mean squared error (`mse`), `categorical_crossentropy`, and `binary_crossentropy`.

```
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="mse", metrics=["mae"])

model.fit(training_data, training_labels, epochs=10, batch_size=32)
```

# Training Keras Models

- Call the `compile` method to configure the learning process.
- `tf.keras.Model.compile` takes three important arguments.
  - `optimizer`: specifies the training procedure.
  - `loss`: the cost function to minimize during optimization, e.g., mean squared error (`mse`), `categorical_crossentropy`, and `binary_crossentropy`.
  - `metrics`: used to monitor training.

```
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="mse", metrics=["mae"])

model.fit(training_data, training_labels, epochs=10, batch_size=32)
```

# Training Keras Models

- Call the `compile` method to configure the learning process.
- `tf.keras.Model.compile` takes three important arguments.
  - `optimizer`: specifies the training procedure.
  - `loss`: the cost function to minimize during optimization, e.g., mean squared error (`mse`), `categorical_crossentropy`, and `binary_crossentropy`.
  - `metrics`: used to monitor training.
- Call the `fit` method to fit the model the training data.

```
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="mse", metrics=["mae"])

model.fit(training_data, training_labels, epochs=10, batch_size=32)
```

# Evaluate and Predict

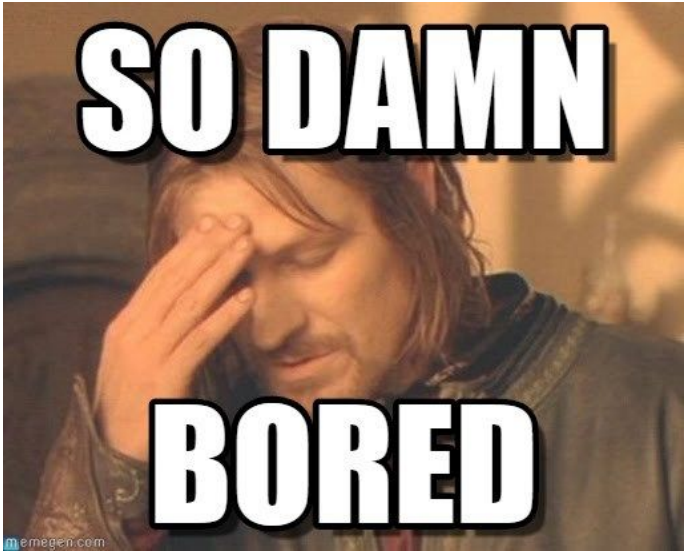
- `tf.keras.Model.evaluate`: evaluate the cost and metrics for the data provided.
- `tf.keras.Model.predict`: predict the output of the last layer for the data provided.

```
model.evaluate(test_data, test_labels, batch_size=32)

model.predict(test_data, batch_size=32)
```

# Feedforward Network 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.pinimg.com/originals/82/d9/2c/82d92c2c15c580c2b2fce65a83fe0b3f.jpg]

# Chain Rule of Calculus (1/2)

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

# Chain Rule of Calculus (1/2)

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

# 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:

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

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

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

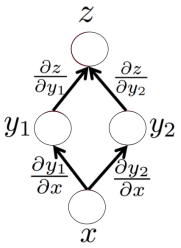
$$\frac{dz}{dx} = 20y^3 \times 3x^2 = 20(x^3 + 7) \times 3x^2$$

# Chain Rule of Calculus (2/2)

- Two paths chain rule.

$$\mathtt{z} = \mathtt{f}(\mathtt{y}_1, \mathtt{y}_2) \text{ where } \mathtt{y}_1 = \mathtt{g}(\mathtt{x}) \text{ and } \mathtt{y}_2 = \mathtt{h}(\mathtt{x})$$

$$\frac{\partial \mathtt{z}}{\partial \mathtt{x}} = \frac{\partial \mathtt{z}}{\partial \mathtt{y}_1}\frac{\partial \mathtt{y}_1}{\partial \mathtt{x}} + \frac{\partial \mathtt{z}}{\partial \mathtt{y}_2}\frac{\partial \mathtt{y}_2}{\partial \mathtt{x}}$$

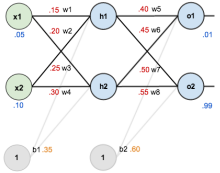



# Backpropagation

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

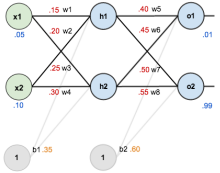
# Backpropagation - Forward Pass

- Calculates outputs given input patterns.



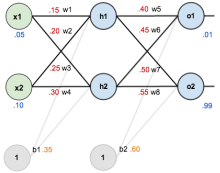
# Backpropagation - Forward Pass

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



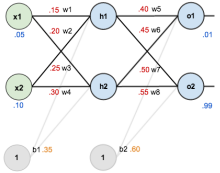
# Backpropagation - Forward Pass

- 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.



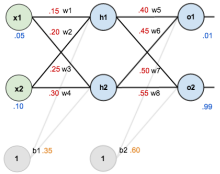
# Backpropagation - Forward Pass

- 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)



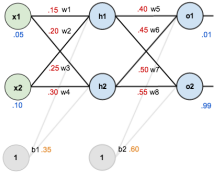
# Backpropagation - Forward Pass

- 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.



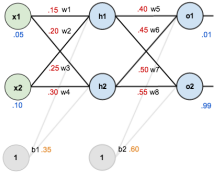
# Backpropagation - Backward Pass

- Updates weights by calculating gradients.



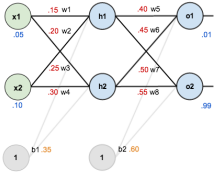
# Backpropagation - Backward Pass

- 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.



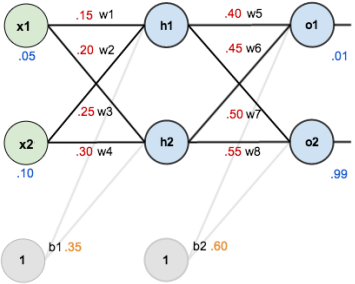
# Backpropagation - Backward Pass

- 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.



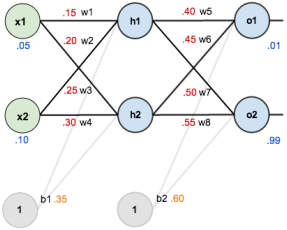
# Backpropagation Example

- 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)

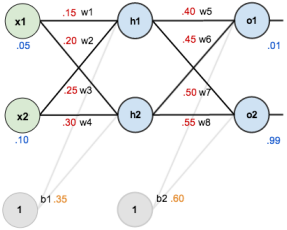
- Compute the output of the hidden layer




$$\texttt{net}_{\texttt{h1}} = \texttt{w}_1\texttt{x}_1 + \texttt{w}_2\texttt{x}_2 + \texttt{b}_1 = 0.15 \times 0.05 + 0.2 \times 0.1 + 0.35 = 0.3775$$

# Backpropagation - Forward Pass (1/3)

- Compute the output of the hidden layer



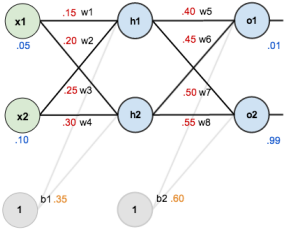

$$\mathrm{net}_{h1} = w_1 x_1 + w_2 x_2 + b_1 = 0.15 \times 0.05 + 0.2 \times 0.1 + 0.35 = 0.3775$$

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

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

# Backpropagation - Forward Pass (2/3)

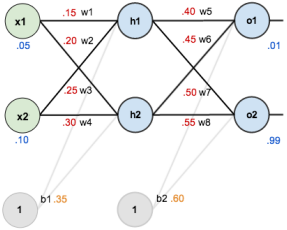
- Compute the output of the output layer




$$\texttt{net}_{o1} = \texttt{w}_5\texttt{out}_{h1} + \texttt{w}_6\texttt{out}_{h2} + \texttt{b}_2 = 0.4 \times 0.59327 + 0.45 \times 0.59688 + 0.6 = 1.1059$$

# Backpropagation - Forward Pass (2/3)

- Compute the output of the output layer



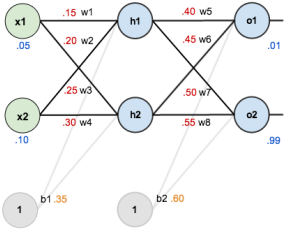

$$\texttt{net}_{o1} = \texttt{w}_5\texttt{out}_{h1} + \texttt{w}_6\texttt{out}_{h2} + \texttt{b}_2 = 0.4 \times 0.59327 + 0.45 \times 0.59688 + 0.6 = 1.1059$$

$$\texttt{out}_{o1} = \frac{1}{1 + \texttt{e}^{\texttt{net}_{o1}}} = \frac{1}{1 + \texttt{e}^{1.1059}} = 0.75136$$

$$\texttt{out}_{o2} = 0.77292$$

# Backpropagation - Forward Pass (3/3)

- Calculate the error for each output




$$\mathrm{E_{o1}} = \frac{1}{2}(\mathrm{target_{o1}} - \mathrm{output_{o1}})^2 = \frac{1}{2}(0.01 - 0.75136)^2 = 0.27481$$

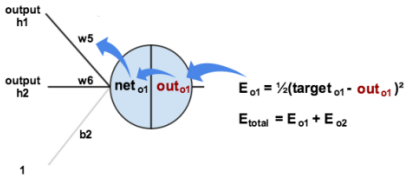
$$\mathrm{E_{o2}} = 0.02356$$

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


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

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

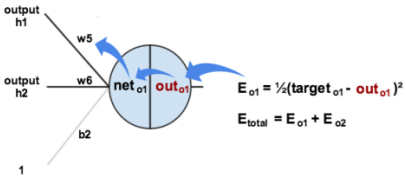
- Consider $w_5$
- We want to know how much a change in $w_5$ affects the total error ($\frac{\partial E_{total}}{\partial w_5}$)
- Applying the chain rule




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

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

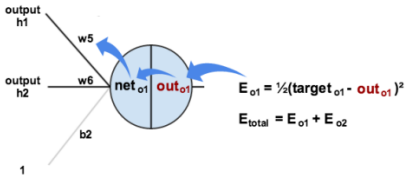
- First, how much does the total error change with respect to the output? ($\frac{\partial E_{total}}{\partial out_{o1}}$)




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

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

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



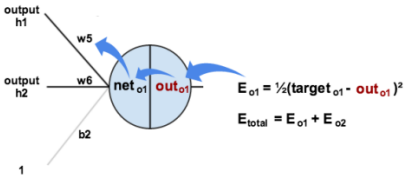

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial w_5}$$

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -2\frac{1}{2}(target_{o1} - out_{o1}) = -(0.01 - 0.75136) = 0.74136$$

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

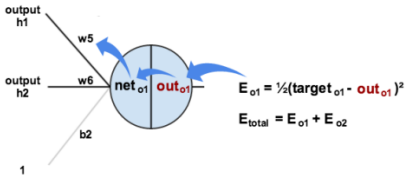
- Next, how much does the $\mathrm{out}_{o1}$ change with respect to its total input $\mathrm{net}_{o1}$? $(\frac{\partial \mathrm{out}_{o1}}{\partial \mathrm{net}_{o1}})$




$$\frac{\partial \mathrm{E}_{\mathrm{total}}}{\partial \mathrm{w}_5} = \frac{\partial \mathrm{E}_{\mathrm{total}}}{\partial \mathrm{out}_{o1}} \times \frac{\partial \mathrm{out}_{o1}}{\partial \mathrm{net}_{o1}} \times \frac{\partial \mathrm{net}_{o1}}{\partial \mathrm{w}_5}$$

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

- Next, how much does the $\text{out}_{o1}$ change with respect to its total input $\text{net}_{o1}$? $(\frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}})$



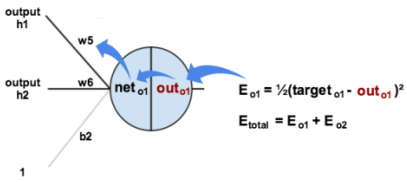

$$\frac{\partial \text{E}_{\text{total}}}{\partial \text{w}_5} = \frac{\partial \text{E}_{\text{total}}}{\partial \text{out}_{o1}} \times \frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} \times \frac{\partial \text{net}_{o1}}{\partial \text{w}_5}$$

$$\text{out}_{o1} = \frac{1}{1 + \text{e}^{-\text{net}_{o1}}}$$

$$\frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} = \text{out}_{o1}(1 - \text{out}_{o1}) = 0.75136(1 - 0.75136) = 0.18681$$

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

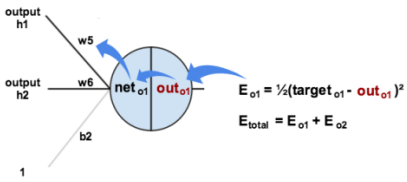
- Finally, how much does the total $\mathtt{net}_{o1}$ change with respect to $\mathtt{w}_5$? ($\frac{\partial \mathtt{net}_{o1}}{\partial \mathtt{w}_5}$)




$$\frac{\partial \mathtt{E}_{\mathtt{total}}}{\partial \mathtt{w}_5} = \frac{\partial \mathtt{E}_{\mathtt{total}}}{\partial \mathtt{out}_{o1}} \times \frac{\partial \mathtt{out}_{o1}}{\partial \mathtt{net}_{o1}} \times \frac{\partial \mathtt{net}_{o1}}{\partial \mathtt{w}_5}$$

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

- Finally, how much does the total $\mathtt{net}_{o1}$ change with respect to $\mathtt{w}_5$? ($\frac{\partial \mathtt{net}_{o1}}{\partial \mathtt{w}_5}$)



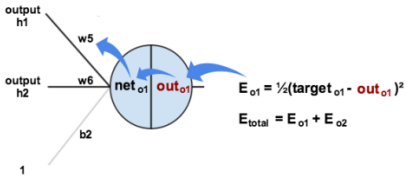

$$\frac{\partial \mathtt{E}_{\mathtt{total}}}{\partial \mathtt{w}_5} = \frac{\partial \mathtt{E}_{\mathtt{total}}}{\partial \mathtt{out}_{o1}} \times \frac{\partial \mathtt{out}_{o1}}{\partial \mathtt{net}_{o1}} \times \frac{\partial \mathtt{net}_{o1}}{\partial \mathtt{w}_5}$$

$$\mathtt{net}_{o1} = \mathtt{w}_5 \times \mathtt{out}_{h1} + \mathtt{w}_6 \times \mathtt{out}_{h2} + \mathtt{b}_2$$

$$\frac{\partial \mathtt{net}_{o1}}{\partial \mathtt{w}_5} = \mathtt{out}_{h1} = 0.59327$$

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

- Putting it all together:

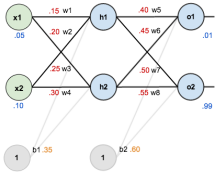



$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial w_5}$$

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

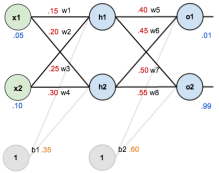
# Backpropagation - Backward Pass - Output Layer (6/6)

- 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 $\eta = 0.5$.

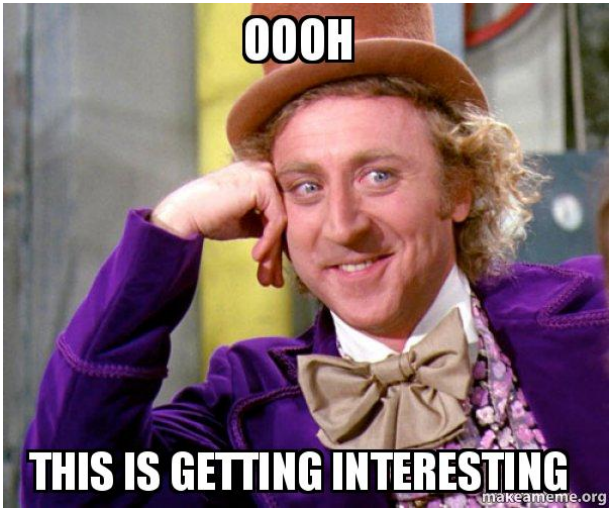



$$w_5^{(next)} = w_5 - \eta \times \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 \times 0.08216 = 0.35891$$

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

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

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

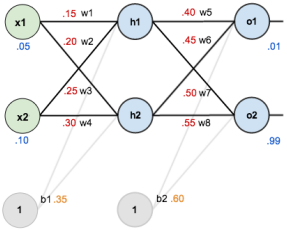



[https://makeameme.org/meme/oooh-this]

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

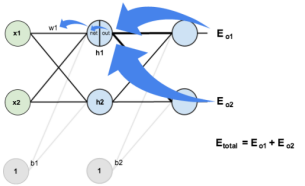
- Continue the backwards pass by calculating new values for $w_1$, $w_2$, $w_3$, and $w_4$.
- For $w_1$ we have:

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



# 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., $\mathtt{out}_{\mathtt{h1}}$ affects both $\mathtt{out}_{\mathtt{o1}}$ and $\mathtt{out}_{\mathtt{o2}}$, so $\frac{\partial \mathtt{E}_{\mathtt{total}}}{\partial \mathtt{out}_{\mathtt{h1}}}$ needs to take into consideration its effect on the both output neurons.

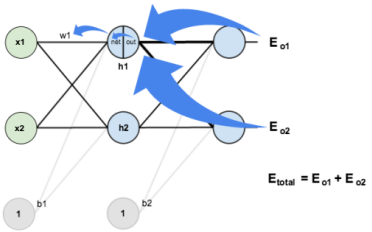



$$\frac{\partial \mathtt{E}_{\mathtt{total}}}{\partial \mathtt{w}_1} = \frac{\partial \mathtt{E}_{\mathtt{total}}}{\partial \mathtt{out}_{\mathtt{h1}}} \times \frac{\partial \mathtt{out}_{\mathtt{h1}}}{\partial \mathtt{net}_{\mathtt{h1}}} \times \frac{\partial \mathtt{net}_{\mathtt{h1}}}{\partial \mathtt{w}_1}$$

$$\frac{\partial \mathtt{E}_{\mathtt{total}}}{\partial \mathtt{out}_{\mathtt{h1}}} = \frac{\partial \mathtt{E}_{\mathtt{o1}}}{\partial \mathtt{out}_{\mathtt{h1}}} + \frac{\partial \mathtt{E}_{\mathtt{o2}}}{\partial \mathtt{out}_{\mathtt{h1}}}$$

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

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




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

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

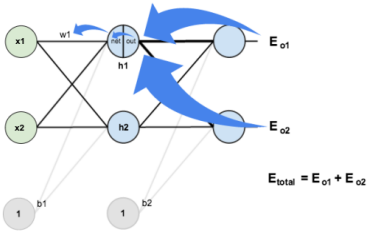
$$\frac{\partial E_{o1}}{\partial out_{o1}} = 0.74136, \frac{\partial out_{o1}}{\partial net_{o1}} = 0.18681$$

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

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

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

- Plugging them together.

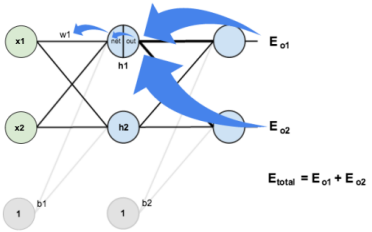



$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{o1}} \times \frac{\partial out_{o1}}{\partial net_{o1}} \times \frac{\partial net_{o1}}{\partial out_{h1}} = 0.74136 \times 0.18681 \times 0.40 = 0.0554$$

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

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

- Plugging them together.



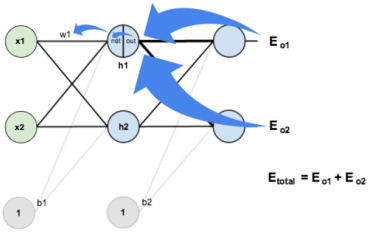

$$\frac{\partial \mathrm{E_{o1}}}{\partial \mathrm{out_{h1}}} = \frac{\partial \mathrm{E_{o1}}}{\partial \mathrm{out_{o1}}} \times \frac{\partial \mathrm{out_{o1}}}{\partial \mathrm{net_{o1}}} \times \frac{\partial \mathrm{net_{o1}}}{\partial \mathrm{out_{h1}}} = 0.74136 \times 0.18681 \times 0.40 =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

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

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

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

- Now we need to figure out $\frac{\partial \text{out}_{h1}}{\partial \text{net}_{h1}}$.



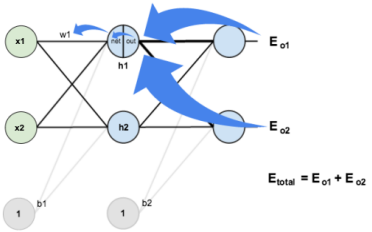

$$\frac{\partial E_{\text{total}}}{\partial w_1} = \frac{\partial E_{\text{total}}}{\partial \text{out}_{h1}} \times \frac{\partial \text{out}_{h1}}{\partial \text{net}_{h1}} \times \frac{\partial \text{net}_{h1}}{\partial w_1}$$

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

$$\frac{\partial \text{out}_{h1}}{\partial \text{net}_{h1}} = \text{out}_{h1}(1 - \text{out}_{h1}) = 0.59327(1 - 0.59327) = 0.2413$$

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

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



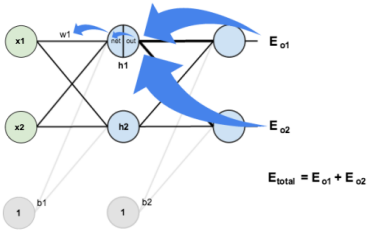

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} \times \frac{\partial out_{h1}}{\partial net_{h1}} \times \frac{\partial net_{h1}}{\partial w_1}$$

$$net_{h1} = w_1 x_1 + w_2 x_2 + b_1$$

$$\frac{\partial net_{h1}}{\partial w_1} = x_1 = 0.05$$

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

- Putting it all together.

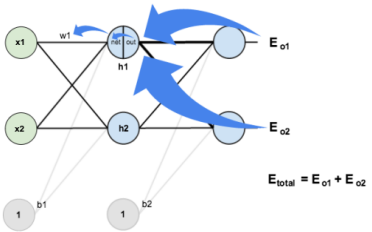



$$\frac{\partial \mathrm{E_{total}}}{\partial \mathrm{w_1}} = \frac{\partial \mathrm{E_{total}}}{\partial \mathrm{out_{h1}}} \times \frac{\partial \mathrm{out_{h1}}}{\partial \mathrm{net_{h1}}} \times \frac{\partial \mathrm{net_{h1}}}{\partial \mathrm{w_1}}$$

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

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

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




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

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

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

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


# Challenges

# Challenges of Training Feedforward Neural Networks

- Challenges ...

# Challenges of Training Feedforward Neural Networks

- Challenges ...
- Overfitting: risk of overfitting a model with large number of parameters.

# Challenges of Training Feedforward Neural Networks

- Challenges ...
- Overfitting: risk of overfitting a model with large number of parameters.
- Vanishing/exploding gradients: hard to train lower layers.

# Challenges of Training Feedforward Neural Networks

- Challenges ...
- Overfitting: risk of overfitting a model with large number of parameters.
- Vanishing/exploding gradients: hard to train lower layers.
- Training speed: slow training with large networks.

# Overfitting

# High Degree of Freedom and Overfitting Problem

- With large number of parameters, a network has a high degree of freedom.
- It can fit a huge variety of complex datasets.

# High Degree of Freedom and Overfitting Problem

- With large number of parameters, a network has a high degree of freedom.
- It can fit a huge variety of complex datasets.
- This flexibility also means that it is prone to overfitting on training set.

# High Degree of Freedom and Overfitting Problem

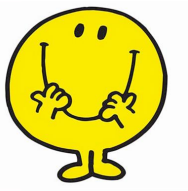
- With large number of parameters, a network has a high degree of freedom.
- It can fit a huge variety of complex datasets.
- This flexibility also means that it is prone to overfitting on training set.
- Regularization: a way to reduce the risk of overfitting.
- It reduces the degree of freedom a model.

# Avoiding Overfitting Through Regularization

- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation

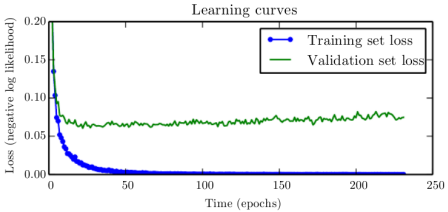
# Avoiding Overfitting Through Regularization

- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation



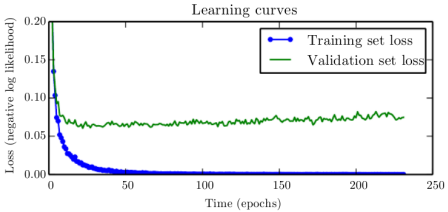
# Early Stopping

- As the training steps go by, its prediction error on the training/validation set naturally goes down.



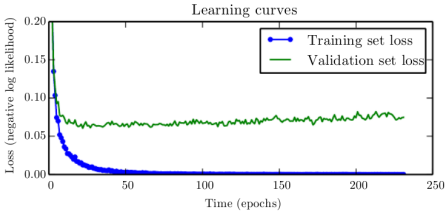
# Early Stopping

- As the training steps go by, its prediction error on the training/validation set naturally goes down.
- After a while the validation error stops decreasing and starts to go back up.
  - The model has started to overfit the training data.



# Early Stopping

- As the training steps go by, its prediction error on the training/validation set naturally goes down.
- After a while the validation error stops decreasing and starts to go back up.
  - The model has started to overfit the training data.
- In the early stopping, we stop training when the validation error reaches a minimum.



# Avoiding Overfitting Through Regularization

- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation

# $l1$ and $l2$ Regularization (1/4)

- Penalize large values of weights $w_j$.

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda R(\mathbf{w})$$

- Two questions:
  1. How should we define $R(\mathbf{w})$?
  2. How do we determine $\lambda$?

# $l1$ and $l2$ Regularization (2/4)

- $l1$ regression: $R(\mathbf{w}) = \lambda \sum_{i=1}^{n} |w_i|$ is added to the cost function.

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda \sum_{i=1}^{n} |w_i|$$

- $l2$ regression: $R(\mathbf{w}) = \lambda \sum_{i=1}^{n} w_i^2$ is added to the cost function.

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda \sum_{i=1}^{n} w_i^2$$

# $l1$ and $l2$ Regularization (3/4)

- Manually implement it in TensorFlow.

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

# $l1$ and $l2$ Regularization (3/4)

- Manually implement it in TensorFlow.

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

```
# extract the weights of layers
W1 = tf.get_default_graph().get_tensor_by_name("hidden/kernel:0")
W2 = tf.get_default_graph().get_tensor_by_name("output/kernel:0")
```

# $l1$ and $l2$ Regularization (3/4)

- Manually implement it in TensorFlow.

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

```
# extract the weights of layers
W1 = tf.get_default_graph().get_tensor_by_name("hidden/kernel:0")
W2 = tf.get_default_graph().get_tensor_by_name("output/kernel:0")
```

```
# l1 regularization
reg_cost = tf.reduce_sum(tf.abs(W1)) + tf.reduce_sum(tf.abs(W2))
```

# $l1$ and $l2$ Regularization (3/4)

- Manually implement it in TensorFlow.

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

```
# extract the weights of layers
W1 = tf.get_default_graph().get_tensor_by_name("hidden/kernel:0")
W2 = tf.get_default_graph().get_tensor_by_name("output/kernel:0")
```

```
# l1 regularization
reg_cost = tf.reduce_sum(tf.abs(W1)) + tf.reduce_sum(tf.abs(W2))
```

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

l1_param = 0.001
cost = base_cost + l1_param * reg_cost

# the rest is as before
```

# $l1$ and $l2$ Regularization (5/5)

- Alternatively, we can pass a regularization function to the `tf.layers.dense()`.

```
# make the network
l1_param = 0.001 # l1 regularization hyperparameter

hidden = tf.layers.dense(X, n_neurons_h, activation=tf.sigmoid, name="hidden",
  kernel_regularizer=tf.contrib.layers.l1_regularizer(l1_param))
logit = tf.layers.dense(hidden, n_neurons_out, name="output",
  kernel_regularizer=tf.contrib.layers.l1_regularizer(l1_param))
```

# $l1$ and $l2$ Regularization (5/5)

- Alternatively, we can pass a regularization function to the `tf.layers.dense()`.

```
# make the network
l1_param = 0.001 # l1 regularization hyperparameter

hidden = tf.layers.dense(X, n_neurons_h, activation=tf.sigmoid, name="hidden",
  kernel_regularizer=tf.contrib.layers.l1_regularizer(l1_param))
logit = tf.layers.dense(hidden, n_neurons_out, name="output",
  kernel_regularizer=tf.contrib.layers.l1_regularizer(l1_param))
```

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

base_cost = tf.reduce_mean(cross_entropy)
reg_cost = tf.losses.get_regularization_loss()

cost = base_cost + reg_cost

# the rest is as before
```

# Avoiding Overfitting Through Regularization

- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation

# Max-Norm Regularization (1/3)

- Max-norm regularization: constrains the weights $\mathbf{w}_j$ of the incoming connections for each neuron $j$.
  - Prevents them from getting too large.

# Max-Norm Regularization (1/3)

- Max-norm regularization: constrains the weights $\mathbf{w}_j$ of the incoming connections for each neuron $j$.
  - Prevents them from getting too large.

- After each training step, clip $\mathbf{w}_j$ as below:

$$\mathbf{w}_j \leftarrow \mathbf{w}_j \frac{r}{||\mathbf{w}_j||_2}$$

# Max-Norm Regularization (1/3)

- Max-norm regularization: constrains the weights $\mathbf{w}_j$ of the incoming connections for each neuron $j$.
  - Prevents them from getting too large.
- After each training step, clip $\mathbf{w}_j$ as below:

$$\mathbf{w}_j \leftarrow \mathbf{w}_j \frac{r}{||\mathbf{w}_j||_2}$$

- We have $||\mathbf{w}_j||_2 \leq r$.
  - $r$ is the max-norm hyperparameter
  - $||\mathbf{w}_j||_2 = (\sum_i w_{i,j}^2)^{\frac{1}{2}} = \sqrt{w_{1,j}^2 + w_{2,j}^2 + \cdots + w_{n,j}^2}$

# Max-Norm Regularization (2/3)

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

# Max-Norm Regularization (2/3)

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

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

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

# Max-Norm Regularization (3/3)

- Use `tf.clip_by_norm`.

```
# max-norm regularization - hidden layer
threshold = 1.0

weights = tf.get_default_graph().get_tensor_by_name("hidden/kernel:0")
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = weights.assign(clipped_weights)
```

# Max-Norm Regularization (3/3)

- Use `tf.clip_by_norm`.

```
# max-norm regularization - hidden layer
threshold = 1.0

weights = tf.get_default_graph().get_tensor_by_name("hidden/kernel:0")
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = weights.assign(clipped_weights)
```

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

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})
    clip_weights.eval()
```

# Avoiding Overfitting Through Regularization

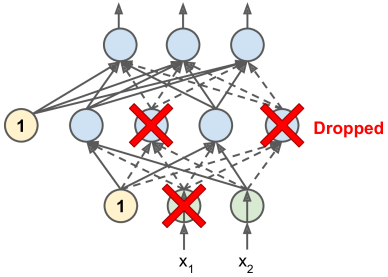
- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation

# Dropout (1/4)

- Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work?

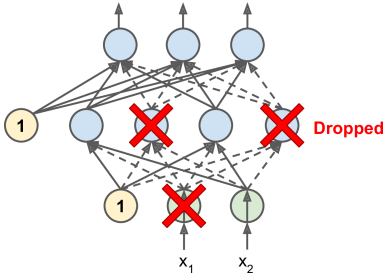
# Dropout (2/4)

- At each training step, each neuron drops out temporarily with a probability p.



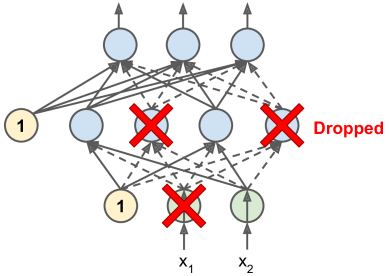
# Dropout (2/4)

- At each training step, each neuron drops out temporarily with a probability p.
  - The hyperparameter p is called the dropout rate.



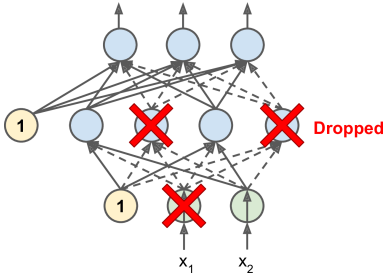
# Dropout (2/4)

- At each training step, each neuron drops out temporarily with a probability p.
  - The hyperparameter p is called the dropout rate.
  - A neuron will be entirely ignored during this training step.



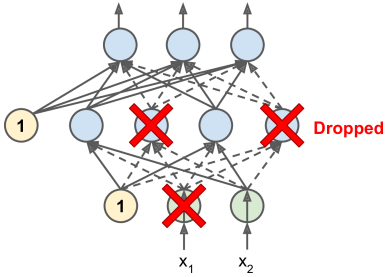
# Dropout (2/4)

- At each training step, each neuron drops out temporarily with a probability p.
  - The hyperparameter p is called the dropout rate.
  - A neuron will be entirely ignored during this training step.
  - It may be active during the next step.



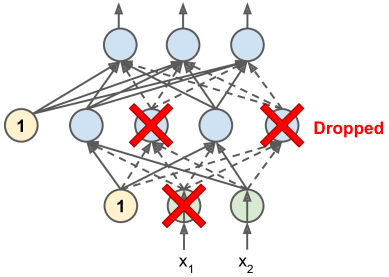
# Dropout (2/4)

- At each training step, each neuron drops out temporarily with a probability p.
  - The hyperparameter p is called the dropout rate.
  - A neuron will be entirely ignored during this training step.
  - It may be active during the next step.
  - Exclude the output neurons.



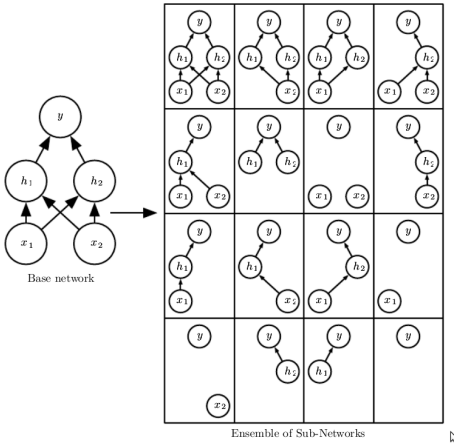
# Dropout (2/4)

- At each training step, each neuron drops out temporarily with a probability p.
  - The hyperparameter p is called the dropout rate.
  - A neuron will be entirely ignored during this training step.
  - It may be active during the next step.
  - Exclude the output neurons.
- After training, neurons don't get dropped anymore.



# Dropout (3/4)

- Each neuron can be either present or absent.
- $2^N$ possible networks, where $N$ is the total number of droppable neurons.
  - $N = 4$ in this figure.



# Dropout (4/4)

- Use `tf.layers.dropout`: specify the dropout rate rather than the keep probability.

```
# make the network
dropout_rate = 0.5  # == 1 - keep_prob
training = tf.placeholder_with_default(False, shape=(), name="training")

X_drop = tf.layers.dropout(X, dropout_rate, training=training)
hidden = tf.layers.dense(X_drop, n_neurons_h, activation=tf.sigmoid, name="hidden")
hidden_drop = tf.layers.dropout(hidden, dropout_rate, training=training)
logit = tf.layers.dense(hidden_drop, n_neurons_out, name="output")
```

# Dropout (4/4)

- Use `tf.layers.dropout`: specify the dropout rate rather than the keep probability.

```
# make the network
dropout_rate = 0.5  # == 1 - keep_prob
training = tf.placeholder_with_default(False, shape=(), name="training")

X_drop = tf.layers.dropout(X, dropout_rate, training=training)
hidden = tf.layers.dense(X_drop, n_neurons_h, activation=tf.sigmoid, name="hidden")
hidden_drop = tf.layers.dropout(hidden, dropout_rate, training=training)
logit = tf.layers.dense(hidden_drop, n_neurons_out, name="output")
```

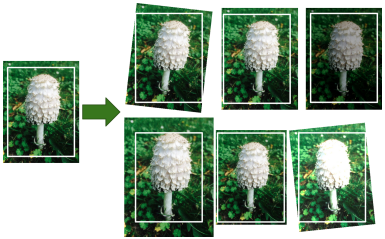
```
# executing the model
init = tf.global_variables_initializer()

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, training: True})
```

# Avoiding Overfitting Through Regularization

- Early stopping
- $l1$ and $l2$ regularization
- Max-norm regularization
- Dropout
- Data augmentation

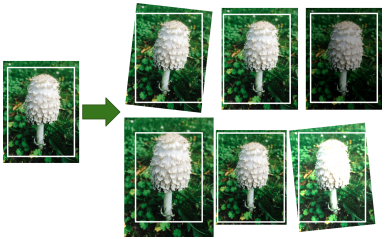
# Data Augmentation

- One way to make a model generalize better is to train it on more data.
- This will reduce overfitting.



# Data Augmentation

- One way to make a model generalize better is to train it on more data.
- This will reduce overfitting.
- Create fake data and add it to the training set.
  - E.g., in an image classification we can slightly shift, rotate and resize an image.
  - Add the resulting pictures to the training set.



# Vanishing/Exploding Gradients

# Vanishing/Exploding Gradients Problem (1/4)

- The backpropagation goes from output to input layer, and propagates the error gradient on the way.

$$\mathtt{w}^{(\mathtt{next})} = \mathtt{w} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}}$$

# Vanishing/Exploding Gradients Problem (1/4)

- The backpropagation goes from output to input layer, and propagates the error gradient on the way.

$$\mathtt{w}^{(\mathtt{next})} = \mathtt{w} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}}$$

- Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
- As a result, the gradient descent update leaves the lower layer connection weights virtually unchanged.

# Vanishing/Exploding Gradients Problem (1/4)

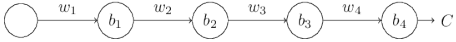
- The backpropagation goes from output to input layer, and propagates the error gradient on the way.

$$\mathtt{w}^{(\mathtt{next})} = \mathtt{w} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}}$$

- Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
- As a result, the gradient descent update leaves the lower layer connection weights virtually unchanged.
- This is called the vanishing gradients problem.

# Vanishing/Exploding Gradients Problem (2/4)

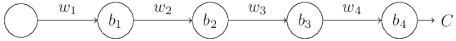
- Assume a network with just a single neuron in each layer.




  - $w_1, w_2, \cdots$ are the weights
  - $b_1, b_2, \cdots$ are the biases
  - $C$ is the cost function

# Vanishing/Exploding Gradients Problem (2/4)

- Assume a network with just a single neuron in each layer.




  - $w_1, w_2, \cdots$ are the weights
  - $b_1, b_2, \cdots$ are the biases
  - $C$ is the cost function

- The output $a_j$ from the $j$th neuron is $\sigma(z_j)$.
  - $\sigma$ is the sigmoid activation function
  - $z_j = w_j a_{j-1} + b_j$
  - E.g., $a_4 = \sigma(z_4) = \mathtt{sigmoid}(w_4 a_3 + b_4)$

# Vanishing/Exploding Gradients Problem (3/4)

- Let's compute the gradient associated to the first hidden neuron ($\frac{\partial C}{\partial b_1}$).


$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial z_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial z_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial z_1}{\partial b_1}$$

# Vanishing/Exploding Gradients Problem (3/4)

- Let's compute the gradient associated to the first hidden neuron ($\frac{\partial C}{\partial b_1}$).


$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial z_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial z_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial z_1}{\partial b_1}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial w_4 a_3 + b_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial w_3 a_2 + b_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial w_2 a_1 + b_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial w_1 a_0 + b_1}{\partial b_1}$$

# Vanishing/Exploding Gradients Problem (3/4)

- Let's compute the gradient associated to the first hidden neuron ($\frac{\partial C}{\partial b_1}$).


$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial z_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial z_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial z_1}{\partial b_1}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \frac{\partial a_4}{\partial z_4} \times \frac{\partial w_4 a_3 + b_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3} \times \frac{\partial w_3 a_2 + b_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial w_2 a_1 + b_2}{\partial a_1} \times \frac{\partial a_1}{\partial z_1} \times \frac{\partial w_1 a_0 + b_1}{\partial b_1}$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \sigma^{'}(z_4) \times w_4 \times \sigma^{'}(z_3) \times w_3 \times \sigma^{'}(z_2) \times \times w_2 \times \sigma^{'}(z_1) \times 1$$

# Vanishing/Exploding Gradients Problem (4/4)

- Now, consider $\frac{\partial \mathtt{C}}{\partial \mathtt{b}_3}$.


$$\frac{\partial \mathtt{C}}{\partial \mathtt{b}_3} = \frac{\partial \mathtt{C}}{\partial \mathtt{a}_4} \times \sigma^{'}(\mathtt{z}_4) \times \mathtt{w}_4 \times \sigma^{'}(\mathtt{z}_3)$$

# Vanishing/Exploding Gradients Problem (4/4)

- Now, consider $\frac{\partial C}{\partial b_3}$.


$$\frac{\partial C}{\partial b_3} = \frac{\partial C}{\partial a_4} \times \sigma'(z_4) \times w_4 \times \sigma'(z_3)$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \sigma'(z_4) \times w_4 \times \sigma'(z_3) \times w_3 \times \sigma'(z_2) \times w_2 \times \sigma'(z_1) \times 1$$

# Vanishing/Exploding Gradients Problem (4/4)

- Now, consider $\frac{\partial C}{\partial b_3}$.


$$\frac{\partial C}{\partial b_3} = \frac{\partial C}{\partial a_4} \times \sigma'(z_4) \times w_4 \times \sigma'(z_3)$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \times \sigma'(z_4) \times w_4 \times \sigma'(z_3) \times w_3 \times \sigma'(z_2) \times w_2 \times \sigma'(z_1) \times 1$$

- Assume $w_3\sigma'(z_2) < \frac{1}{4}$ and $w_2\sigma'(z_1) < \frac{1}{4}$
  - The gradient $\frac{\partial C}{\partial b_1}$ be a factor of 16 (or more) smaller than $\frac{\partial C}{\partial b_3}$.
  - This is the essential origin of the vanishing gradient problem.

# Overcoming the Vanishing Gradient

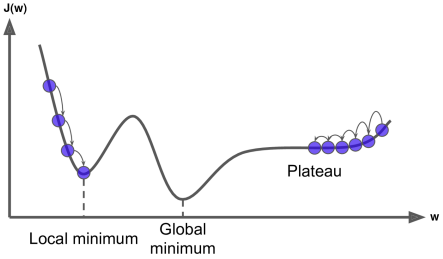
- Parameter initialization strategies
- Nonsaturating activation function
- Batch normalization

# Overcoming the Vanishing Gradient

- Parameter initiazlization strategies
- Nonsaturating activation function
- Batch normalization

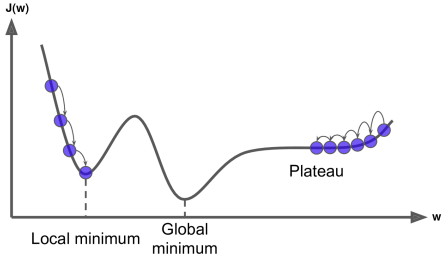
# Parameter Initialization Strategies (1/2)

- The non-linearity of a neural network causes the cost functions to become non-convex.



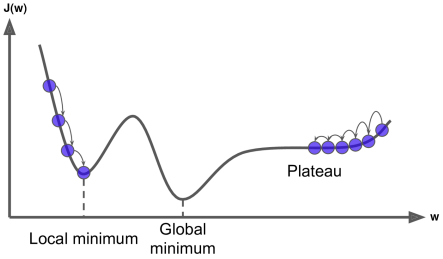
# Parameter Initialization Strategies (1/2)

- The non-linearity of a neural network causes the cost functions to become non-convex.
- The stochastic gradient descent on non-convex cost functions performs is sensitive to the values of the initial parameters.



# Parameter Initialization Strategies (1/2)

- The non-linearity of a neural network causes the cost functions to become non-convex.
- The stochastic gradient descent on non-convex cost functions performs is sensitive to the values of the initial parameters.
- Designing initialization strategies is a difficult task.



# Parameter Initialization Strategies (2/2)

- The initial parameters need to break symmetry between different units.

# Parameter Initialization Strategies (2/2)

- The initial parameters need to break symmetry between different units.
- Two hidden units with the same activation function connected to the same inputs, must have different initial parameters.

# Parameter Initialization Strategies (2/2)

- The initial parameters need to break symmetry between different units.
- Two hidden units with the same activation function connected to the same inputs, must have different initial parameters.
  - The goal of having each unit compute a different function.

# Parameter Initialization Strategies (2/2)

- The initial parameters need to break symmetry between different units.
- Two hidden units with the same activation function connected to the same inputs, must have different initial parameters.
  - The goal of having each unit compute a different function.
- It motivates random initialization of the parameters.
  - Typically, we set the biases to constants, and initialize only the weights randomly.

# Overcoming the Vanishing Gradient

- Parameter initiazlization strategies
- Nonsaturating activation function
- Batch normalization

# Nonsaturating Activation Functions (1/4)

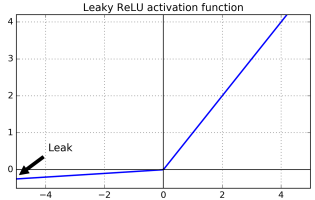
- $\texttt{ReLU}(z) = \max(0, z)$
- The dying ReLUs problem.

# Nonsaturating Activation Functions (1/4)

- $\texttt{ReLU}(z) = \max(0, z)$
- The dying ReLUs problem.
  - During training, some neurons stop outputting anything other than 0.
  - E.g., when the weighted sum of the neuron's inputs is negative, it starts outputting 0.

# Nonsaturating Activation Functions (1/4)

- $\texttt{ReLU}(z) = \max(0, z)$
- The dying ReLUs problem.
  - During training, some neurons stop outputting anything other than 0.
  - E.g., when the weighted sum of the neuron's inputs is negative, it starts outputting 0.
- Use leaky ReLU instead: $\texttt{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$.
  - $\alpha$ is the slope of the function for $z < 0$.



# Nonsaturating Activation Functions (2/4)

- Randomized Leaky ReLU (RReLU)
  - $\alpha$ is picked randomly during training, and it is fixed during testing.

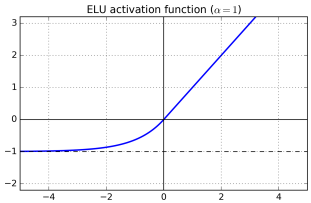
# Nonsaturating Activation Functions (2/4)

- Randomized Leaky ReLU (RReLU)
  - $\alpha$ is picked randomly during training, and it is fixed during testing.
- Parametric Leaky ReLU (PReLU)
  - Learn $\alpha$ during training (instead of being a hyperparameter).

# Nonsaturating Activation Functions (2/4)

- Randomized Leaky ReLU (RReLU)
  - $\alpha$ is picked randomly during training, and it is fixed during testing.
- Parametric Leaky ReLU (PReLU)
  - Learn $\alpha$ during training (instead of being a hyperparameter).
- Exponential Linear Unit (ELU)

$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

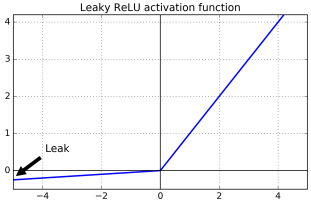


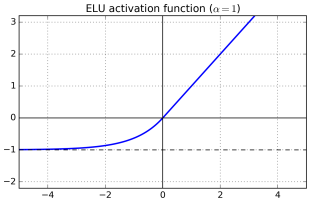
# Nonsaturating Activation Functions (3/4)

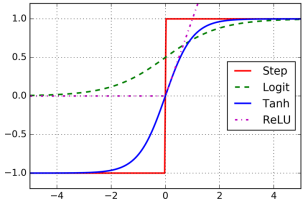
- Which activation function should we use?

# Nonsaturating Activation Functions (3/4)

- Which activation function should we use?
- In general logistic $<$ tanh $<$ ReLU $<$ leaky ReLU (and its variants) $<$ ELU

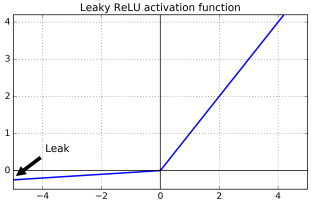


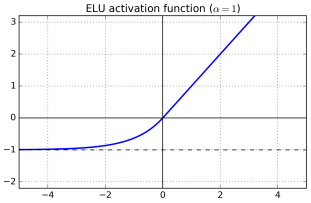



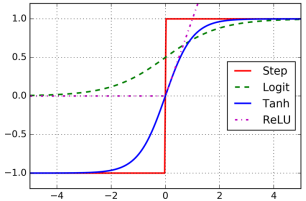



# Nonsaturating Activation Functions (3/4)

- Which activation function should we use?
- In general logistic $<$ tanh $<$ ReLU $<$ leaky ReLU (and its variants) $<$ ELU
- If you care about runtime performance, then leaky ReLUs works better than ELUs.



# Nonsaturating Activation Functions (4/4)

```
# leaky relu
def leaky_relu(z, name=None):
  alpha = 0.01
  return tf.maximum(alpha * z, z, name=name)

hidden = tf.layers.dense(X, n_neurons_h, activation=leaky_relu, name="hidden")
```

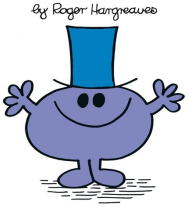
# Nonsaturating Activation Functions (4/4)

```
# leaky relu
def leaky_relu(z, name=None):
  alpha = 0.01
  return tf.maximum(alpha * z, z, name=name)

hidden = tf.layers.dense(X, n_neurons_h, activation=leaky_relu, name="hidden")
```

```
# elu
hidden = tf.layers.dense(X, n_neurons_h, activation=tf.nn.elu, name="hidden")
```

# Overcoming the Vanishing Gradient

- Parameter initiazlization strategies
- Nonsaturating activation function
- Batch normalization



# Batch Normalization (1/5)

- The gradient tells how to update each parameter, under the assumption that the other layers do not change.

# Batch Normalization (1/5)

- The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.

# Batch Normalization (1/5)

- The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.
- Batch normalization makes the learning of layers in the network more independent of each other.

# Batch Normalization (1/5)

- The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.
- Batch normalization makes the learning of layers in the network more independent of each other.
  - It is a technique to address the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.

# Batch Normalization (1/5)

- The gradient tells how to update each parameter, under the assumption that the other layers do not change.
  - In practice, we update all of the layers simultaneously.
  - However, unexpected results can happen.
- Batch normalization makes the learning of layers in the network more independent of each other.
  - It is a technique to address the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.
- The technique consists of adding an operation in the model just before the activation function of each layer.

# Batch Normalization (2/5)

- It's zero-centering and normalizing the inputs, then scaling and shifting the result.

# Batch Normalization (2/5)

- It's zero-centering and normalizing the inputs, then scaling and shifting the result.
  - Estimates the inputs' mean and standard deviation of the current mini-batch.

# Batch Normalization (2/5)

- It's zero-centering and normalizing the inputs, then scaling and shifting the result.
  - Estimates the inputs' mean and standard deviation of the current mini-batch.

$$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

$$\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$$

- $\mu_B$: the empirical mean, evaluated over the whole mini-batch B.
- $\sigma_B$: the empirical standard deviation, also evaluated over the whole mini-batch.
- $m_B$: the number of instances in the mini-batch.

# Batch Normalization (3/5)

$$\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$
$$\mathbf{z}^{(i)} = \gamma \hat{\mathbf{x}}^{(i)} + \beta$$

- $\hat{\mathbf{x}}^{(i)}$: the zero-centered and normalized input.
- $\gamma$: the scaling parameter for the layer.
- $\beta$: the shifting parameter (offset) for the layer.
- $\epsilon$: a tiny number to avoid division by zero.
- $\mathbf{z}^{(i)}$: the output of the BN operation, which is a scaled and shifted version of the inputs.

# Batch Normalization (4/5)

- Use `tf.layers.batch_normalization`

```
# make the network
training = tf.placeholder_with_default(False, shape=(), name="training")

hidden = tf.layers.dense(X, n_neurons_h, name="hidden")
bn = tf.layers.batch_normalization(hidden, training=training)
bn_act = tf.sigmoid(bn)

logits_before_bn = tf.layers.dense(bn_act, n_outputs, name="output")
logits = tf.layers.batch_normalization(logits_before_bn, training=training)
```

# Batch Normalization (4/5)

- Use `tf.layers.batch_normalization`

```
# make the network
training = tf.placeholder_with_default(False, shape=(), name="training")

hidden = tf.layers.dense(X, n_neurons_h, name="hidden")
bn = tf.layers.batch_normalization(hidden, training=training)
bn_act = tf.sigmoid(bn)

logits_before_bn = tf.layers.dense(bn_act, n_outputs, name="output")
logits = tf.layers.batch_normalization(logits_before_bn, training=training)
```

```
# define the cost
cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
cost = tf.reduce_mean(cross_entropy)

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

# Batch Normalization (5/5)

- We need to explicitly run the extra update operations needed by batch normalization `sess.run([training_op, extra_update_ops], ...)`.

```
extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
init = tf.global_variables_initializer()

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

# Training Speed

# Regular Gradient Descent Optimization (1/2)

- Gradient descent optimization algorithm
- It updates the weights $\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$
- Better optimization algorithms to improve the training speed

# Regular Gradient Descent Optimization (2/2)

```
# 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)
```

# Optimization Algorithms

- Momentum
- AdaGrad
- RMSProp
- Adam Optimization

# Optimization Algorithms

- Momentum
- AdaGrad
- RMSProp
- Adam optimization

# Momentum (1/4)

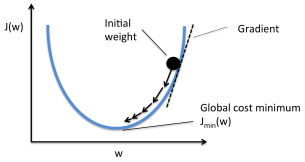
- Momentum is a concept from physics: an object in motion will have a tendency to keep moving.
- It measures the resistance to change in motion.
  - The higher momentum an object has, the harder it is to stop it.

# Momentum (2/4)

- This is the very simple idea behind momentum optimization.

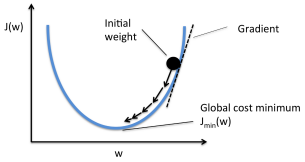
# Momentum (2/4)

- This is the very simple idea behind momentum optimization.
- We can see the change in the parameters $\mathbf{w}$ as motion: $\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$



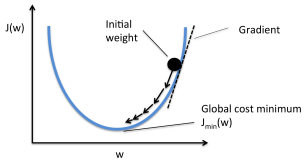
# Momentum (2/4)

- This is the very simple idea behind momentum optimization.
- We can see the change in the parameters $\mathbf{w}$ as motion: $\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$
- We can thus use the concept of momentum to give the update process a tendency to keep moving in the same direction.



# Momentum (2/4)

- This is the very simple idea behind momentum optimization.
- We can see the change in the parameters $\mathbf{w}$ as motion: $\mathtt{w}_\mathtt{i}^{(\mathtt{next})} = \mathtt{w}_\mathtt{i} - \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$
- We can thus use the concept of momentum to give the update process a tendency to keep moving in the same direction.
- It can help to escape from bad local minima pits.



# Momentum (3/4)

- Momentum optimization cares about what previous gradients were.

# Momentum (3/4)

- Momentum optimization cares about what previous gradients were.
- At each iteration, it adds the local gradient to the momentum vector **m**.

$$\mathtt{m_i} = \beta \mathtt{m_i} + \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w_i}}$$

# Momentum (3/4)

- Momentum optimization cares about what previous gradients were.
- At each iteration, it adds the local gradient to the momentum vector $\mathbf{m}$.

$$\mathtt{m}_\mathtt{i} = \beta \mathtt{m}_\mathtt{i} + \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_\mathtt{i}}$$

- $\beta$ is called momentum, ans it is between 0 and 1.

# Momentum (3/4)

- Momentum optimization cares about what previous gradients were.
- At each iteration, it adds the local gradient to the momentum vector **m**.

$$\mathtt{m}_i = \beta \mathtt{m}_i + \eta \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w}_i}$$

- $\beta$ is called momentum, ans it is between 0 and 1.
- Updates the weights by subtracting this momentum vector.

$$\mathtt{w}_i^{(\mathtt{next})} = \mathtt{w}_i - \mathtt{m}_i$$

# Momentum (4/4)

```
# train the model

optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate, momentum=0.9)
```

# Optimization Algorithms

- Momentum
- AdaGrad
- RMSProp
- Adam optimization

# AdaGrad (1/3)

- AdaGrad keeps track of a learning rate for each parameter.
- Adapts the learning rate over time (adaptive learning rate).

# AdaGrad (2/3)

- For each feature $\mathtt{w_i}$, we do the following steps:

$$\mathtt{s_i} = \mathtt{s_i} + (\frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w_i}})^2$$

$$\mathtt{w_i^{(next)}} = \mathtt{w_i} - \frac{\eta}{\sqrt{\mathtt{s_i} + \epsilon}} \frac{\partial \mathtt{J}(\mathbf{w})}{\partial \mathtt{w_i}}$$

- Parameters with large partial derivative of the cost have a rapid decrease in their learning rate.
- Parameters with small partial derivatives have a small decrease in their learning rate.

# AdaGrad (3/3)

```
# train the model

optimizer = tf.train.AdagradOptimizer(learning_rate=learning_rate)
```

# Optimization Algorithms

- Momentum
- AdaGrad
- RMSProp
- Adam optimization


# RMSProp (1/3)

- AdaGrad often stops too early when training neural networks.
- The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum.

# RMSProp (2/3)

- The RMSProp fixed the AdaGrad problem.
- It is like the AdaGrad problem, but accumulates only the gradients from the most recent iterations (not from the beginning of training).
- For each feature $\mathtt{w_i}$, we do the following steps:

$$\mathtt{s_i} = \beta \mathtt{s_i} + (1-\beta)\left(\frac{\partial \mathtt{J(w)}}{\partial \mathtt{w_i}}\right)^2$$

$$\mathtt{w_i^{(next)}} = \mathtt{w_i} - \frac{\eta}{\sqrt{\mathtt{s_i} + \epsilon}} \frac{\partial \mathtt{J(w)}}{\partial \mathtt{w_i}}$$

# RMSProp (3/3)

```
# train the model

optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate, momentum=0.9,
  decay=0.9, epsilon=1e-10)
```

# Optimization Algorithms

- Momentum
- AdaGrad
- RMSProp
- Adam optimization

# Adam Optimization (1/3)

- Adam (Adaptive moment estimation) combines the ideas of Momentum optimization and RMSProp.

# Adam Optimization (1/3)

- Adam (Adaptive moment estimation) combines the ideas of Momentum optimization and RMSProp.
- Like Momentum optimization, it keeps track of an exponentially decaying average of past gradients.

# Adam Optimization (1/3)

- Adam (Adaptive moment estimation) combines the ideas of Momentum optimization and RMSProp.
- Like Momentum optimization, it keeps track of an exponentially decaying average of past gradients.
- Like RMSProp, it keeps track of an exponentially decaying average of past squared gradients.

# Adam Optimization (2/3)

$$\text{1. } \mathbf{m}^{(\text{next})} = \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_{\mathbf{w}} J(\mathbf{w})$$
$$\text{2. } \mathbf{s}^{(\text{next})} = \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\mathbf{w}} J(\mathbf{w}) \otimes \nabla_{\mathbf{w}} J(\mathbf{w})$$
$$\text{3. } \mathbf{m}^{(\text{next})} = \frac{\mathbf{m}}{1 - \beta_1^T}$$
$$\text{4. } \mathbf{s}^{(\text{next})} = \frac{\mathbf{s}}{1 - \beta_2^T}$$
$$\text{5. } \mathbf{w}^{(\text{next})} = \mathbf{w} - \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$$

- $\otimes$ and $\oslash$ represents the represents the element-wise multiplication and division.
- Steps 1, 2, and 5: similar to both Momentum optimization and RMSProp.
- Steps 3 and 4: since $\mathbf{m}$ and $\mathbf{s}$ are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost $\mathbf{m}$ and $\mathbf{s}$ at the beginning of training.

# Adam Optimization (3/3)

```
# train the model

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```


# 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

# Summary

- Overfitting
  - Early stopping, $l1$ and $l2$ regularization, max-norm regularization
  - Dropout, data augmentation
- Vanishing gradient
  - Parameter initialization, nonsaturating activation functions
  - Batch normalization
- Training speed
  - Momentum, AdaGrad
  - RMSProp, Adam optimization

# Reference


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


# Questions?