Lecture 14: Neural Networks

Applied Machine Learning

Volodymyr Kuleshov, Jin Sun
Cornell Tech

Part 1: An Artifical Neuron

In this lecture, we will learn about a new class of machine learning algorithms inspired by the brain.

We will start by defining a few building blocks for these algorithms, and draw connections to neuroscience.

Review: Binary Classification

In supervised learning, we fit a model of the form
$$f : \mathcal{X} \to \mathcal{Y}$$
that maps inputs $x \in \mathcal{X}$ to targets $y \in \mathcal{Y}$.

In classification, the space of targets $\mathcal{Y}$ is *discrete*. Classification is binary if $\mathcal{Y} = \{0, 1\}$

Review: Logistic Regression

Logistic regression fits a model of the form

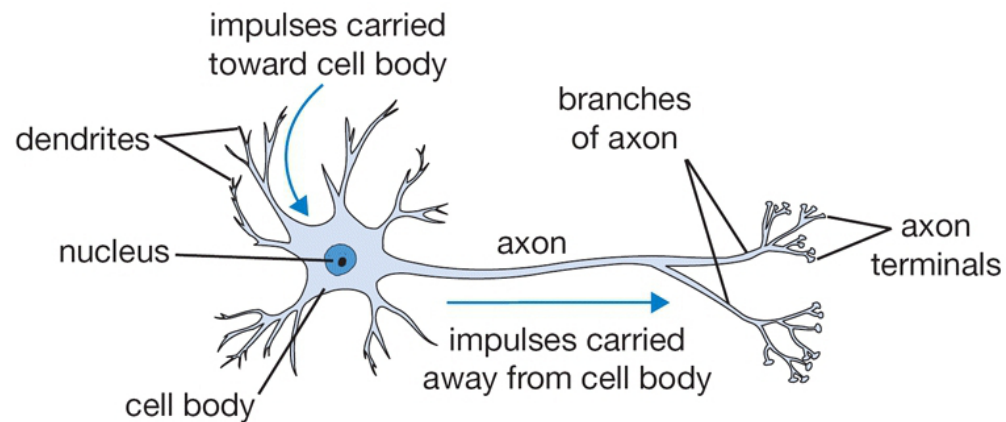$$f(x) = \sigma(\theta^\top x) = \frac{1}{1 + \exp(-\theta^\top x)},$$

where

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

is known as the *sigmoid* or *logistic* function.

A Biological Neuron

In order to define an artifical neuron, let's look first at a biological one.

- Each neuron receives input signals from its dendrites
- If input signals are strong enough, neuron fires output along its axon, which connects to the dendrites of other neurons.

An Artificial Neuron: Example

We can imitate this machinery using an idealized artifical neuron.

- Dendrite j gets signal x_j; modulates multiplicatively to $w_j \cdot x_j$.
- The body of the neuron sums the modulated inputs: $\sum_{j=1}^{d} w_j \cdot x_j$.
- These go into the activation function that produces an ouput.
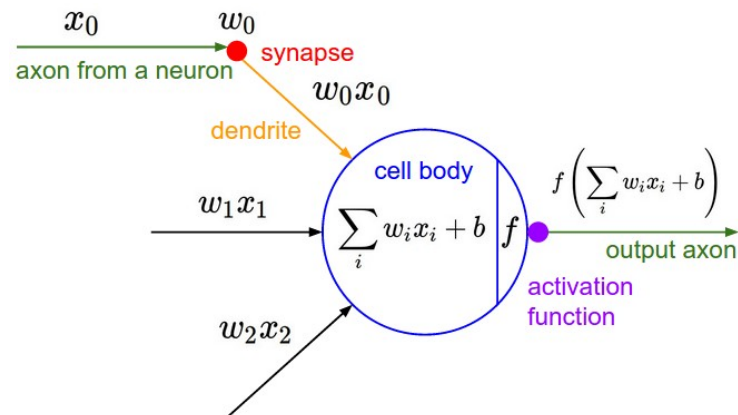
An Artificial Neuron: Notation

More formally, we say that a neuron is a model $f : \mathbb{R}^d \rightarrow [0, 1]$, with the following components:

- Inputs $x_1, x_2, \ldots, x_d$, denoted by a vector x.
- Weight vector $w \in \mathbb{R}^d$ that modulates input x as $w^\top x$.
- An activation function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ that computes the output $\sigma(w^\top x)$ of the neuron based on the sum of modulated features $w^\top x$.

Perceptron

If we use a step function as the activation function, we obtain the classic Perceptron model:

$$f(x) = \begin{cases} 1 & \text{if } \theta^\top x > 0, \\ 0 & \text{otherwise} \end{cases}$$
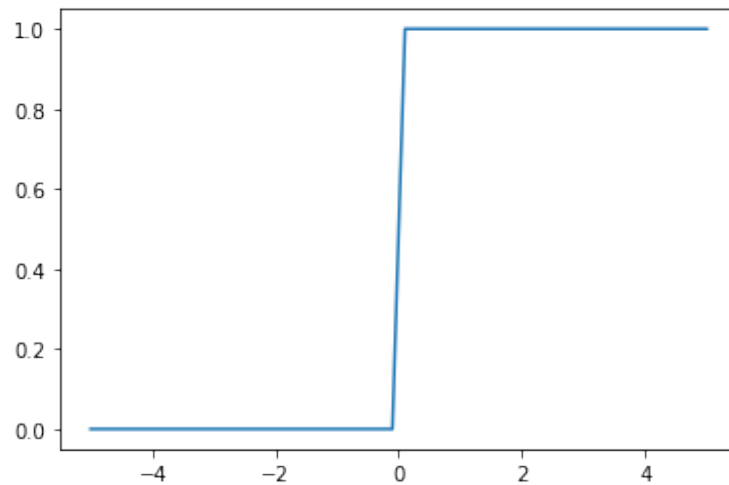
This models a neuron that fires if the inputs are sufficiently large, and doesn't otherwise.

We can visualize the activation function of the Perceptron.

In [2]:
```python
step_fn = lambda z: 1 if z > 0 else 0
plt.plot(z, [step_fn(zi) for zi in z])
```

Out[2]:
`[<matplotlib.lines.Line2D at 0x120c11978>]`

Logistic Regression as an Artifical Neuron

Logistic regression is a model of the form

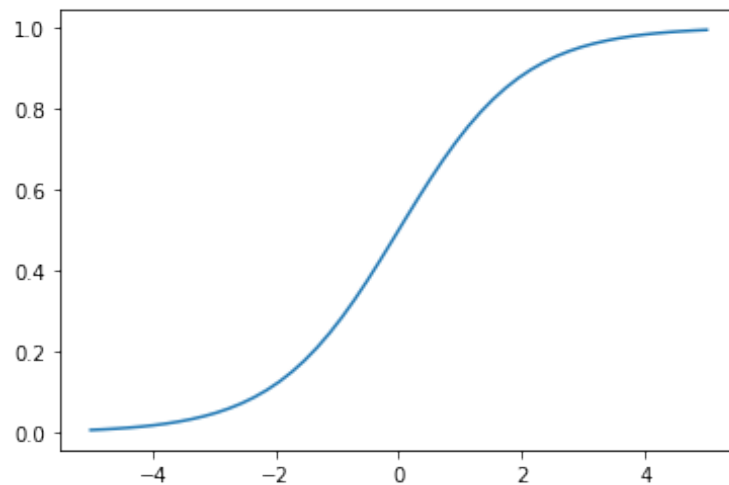$$f(x) = \sigma(\theta^\top x) = \frac{1}{1 + \exp(-\theta^\top x)},$$

that can be interpreted as a neuron that uses the *sigmoid* as the activation function.

The sigmoid activation function encodes the idea of a neuron firing if the inputs exceed a threshold, makes make the activation function "smooth".

In [3]:
```
z = np.linspace(-5, 5)
sigma = 1/(1+np.exp(-z))

plt.plot(z, sigma)
```

Out[3]: [<matplotlib.lines.Line2D at 0x120c832e8>]

Activation Functions

There are many other activation functions that can be used. In practice, these two work better than the sigmoid:

- Hyperbolic tangent (`tanh`): $\sigma(z) = \tanh(z)$
- Rectified linear unit (`ReLU`): $\sigma(z) = \max(0, z)$

We can easily visualize these.

```
In [78]:  %matplotlib inline
          import matplotlib.pyplot as plt
          plt.rcParams['figure.figsize'] = [12, 4]

          plt.subplot(121)
          plt.plot(z, np.tanh(z))
          plt.subplot(122)
          plt.plot(z, np.maximum(z, 0))
```

Out[78]: [<matplotlib.lines.Line2D at 0x1333eb668>]
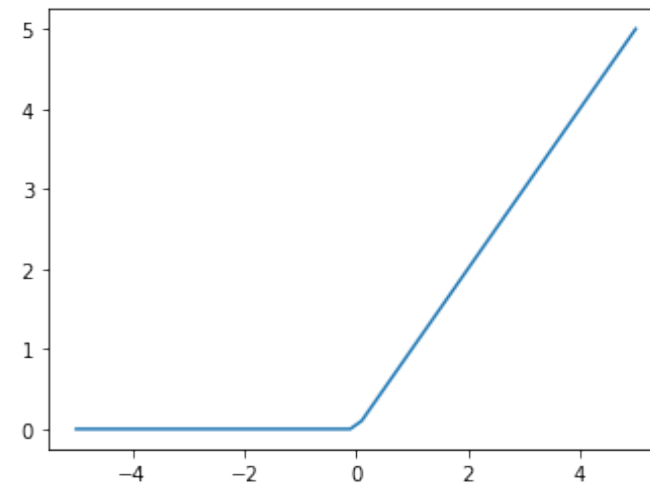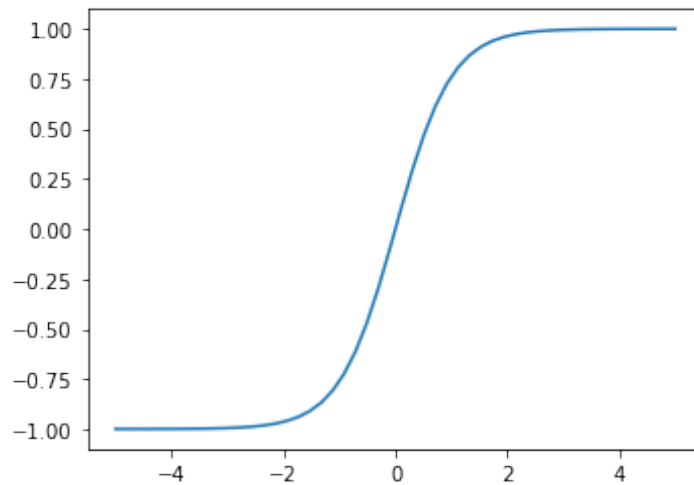
Classification Dataset: Iris Flowers

To demonstrate classification algorithms, we are going to use the Iris flower dataset.

We are going to define an artificial neuron for the binary classification problem (class-0 vs the rest).

```python
In [60]:  # https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.ht
          ml
          import numpy as np
          import pandas as pd
          from sklearn import datasets

          # Load the Iris dataset
          iris = datasets.load_iris(as_frame=True)
          iris_X, iris_y = iris.data, iris.target

          # rename class two to class one
          iris_y2 = iris_y.copy()
          iris_y2[iris_y2==2] = 1

          X = iris_X.to_numpy()[:,:2]
          Y = iris_y2
```
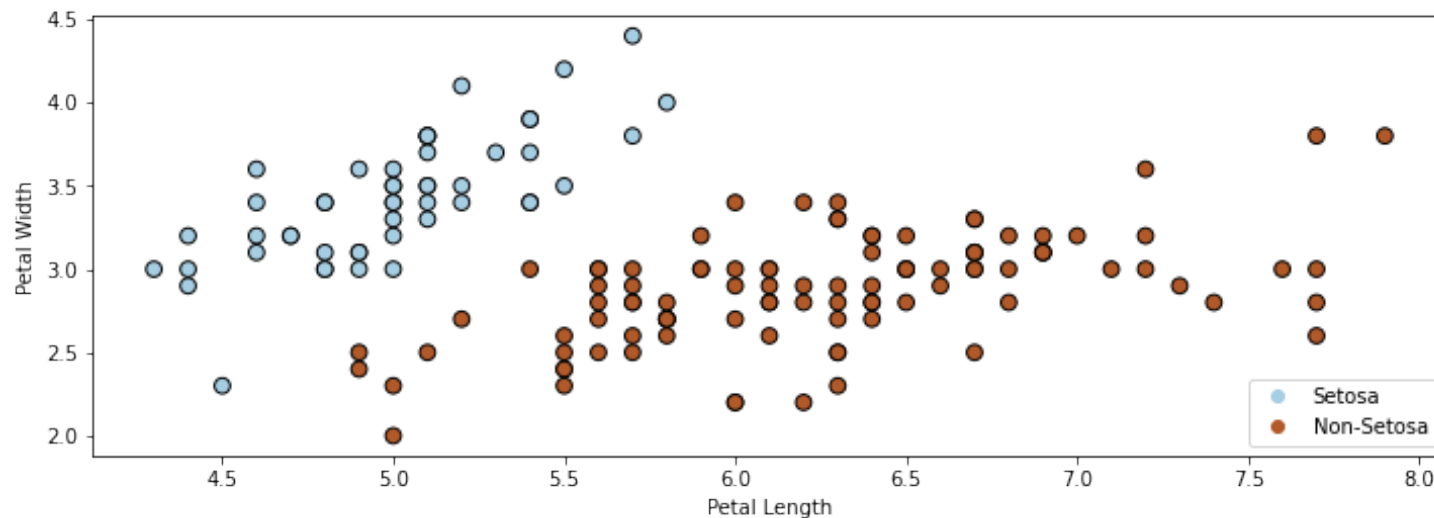
This is a visualization of the dataset.

```
In [61]:    # Plot also the training points
            p1 = plt.scatter(X[:,0], X[:,1], c=iris_y2, edgecolor='k', s=60, cmap=plt.cm.Pai
            red)
            plt.xlabel('Petal Length')
            plt.ylabel('Petal Width')
            plt.legend(handles=p1.legend_elements()[0], labels=['Setosa', 'Non-Setosa'], loc
            ='lower right')
```

Out[61]:   <matplotlib.legend.Legend at 0x12f4f45c0>

Below, we define neuron with a sigmoid activation function (and its gradient).

```
In [69]:  def neuron(X, theta):
              activation_fn = lambda z: 1/(1+np.exp(-z))
              return activation_fn(X.dot(theta))

          def gradient(theta, X, y):
              return np.mean((y - neuron(X, theta)) * X.T, axis=1)
```

We can optimize is using gradient descent.

```python
threshold = 5e-5
step_size = 1e-1

iter, theta, theta_prev = np.zeros((3,)), np.ones((3,)), 0
iris_X['one'] = 1 # add a vector of ones for the bias
X_train = iris_X.iloc[:,[0,1,-1]].to_numpy()
y_train = iris_y2.to_numpy()

while np.linalg.norm(theta - theta_prev) > threshold:
    if iter % 50000 == 0:
        print('Iteration %d.' % iter)
    theta_prev = theta
    grad = gradient(theta, X_train, y_train)
    theta = theta_prev + step_size * grad
    iter += 1
```

```
Iteration 0.
Iteration 50000.
Iteration 100000.
Iteration 150000.
Iteration 200000.
```

This neuron learns a linear decision boundary that separates the data.

```python
# generate predictions over a grid:
xx, yy = np.meshgrid(np.arange(3.3, 8.9, 0.02), np.arange(1.0, 5.4, 0.02))
Z = neuron(np.c_[xx.ravel(), yy.ravel(), np.ones(xx.ravel().shape)], theta)
Z[Z<0.5] = 0
Z[Z>=0.5] = 1

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Paired)

# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=Y, edgecolors='k', cmap=plt.cm.Paired)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
```
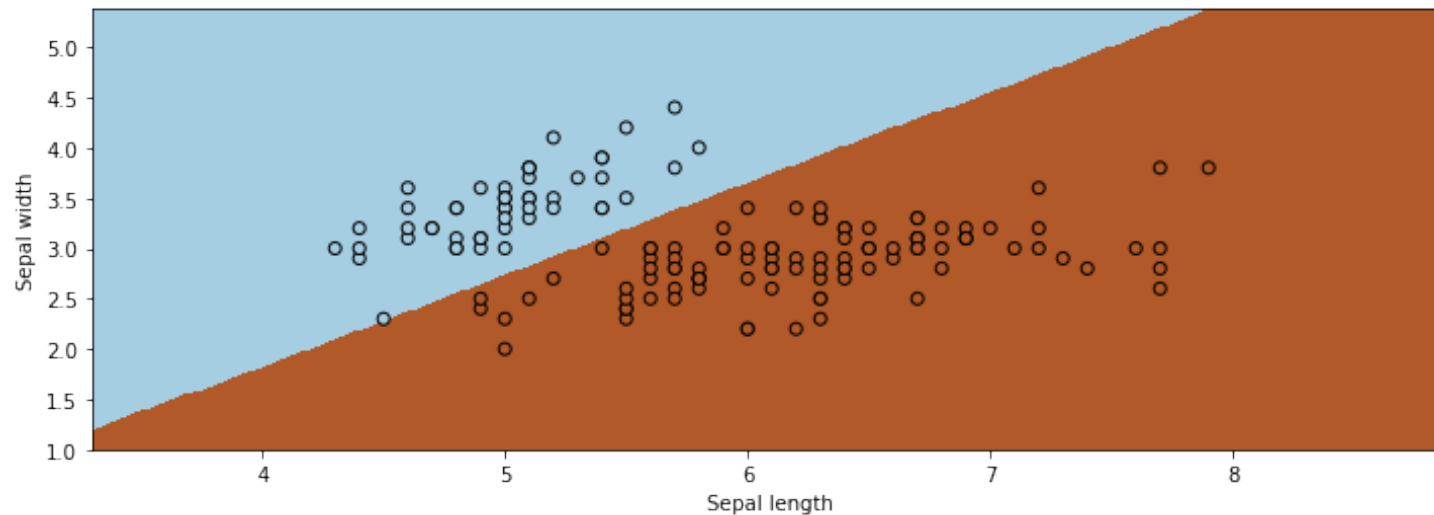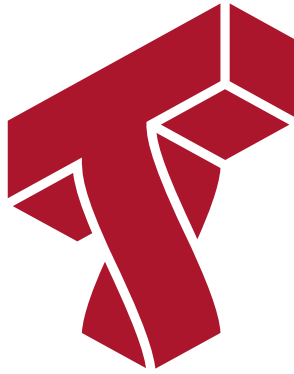
Text(0, 0.5, 'Sepal width')

Algorithm: Artificial Neuron

- **Type**: Supervised learning (regression and classification).
- **Model family**: Linear model followed by non-linear activation.
- **Objective function**: Any differentiable objective.
- **Optimizer**: Gradient descent.
- **Special Cases**: Logistic regression, Perceptron

Part 2: Artificial Neural Networks

Let's now see how we can connect neurons into networks that form complex models that further mimic the brain.

Review: Artificial Neuron

We say that a neuron is a model $f : \mathbb{R}^d \to [0, 1]$, with the following components:

- Inputs $x_1, x_2, \ldots, x_d$, denoted by a vector x.
- Weight vector $w \in \mathbb{R}^d$ that modulates input x as $w^\top x$.
- An activation function $\sigma : \mathbb{R} \to \mathbb{R}$ that computes the output $\sigma(w^\top x)$ of the neuron based on the sum of modulated features $w^\top x$.

Review: Logistic Regression as Neuron

Logistic regression is a model of the form
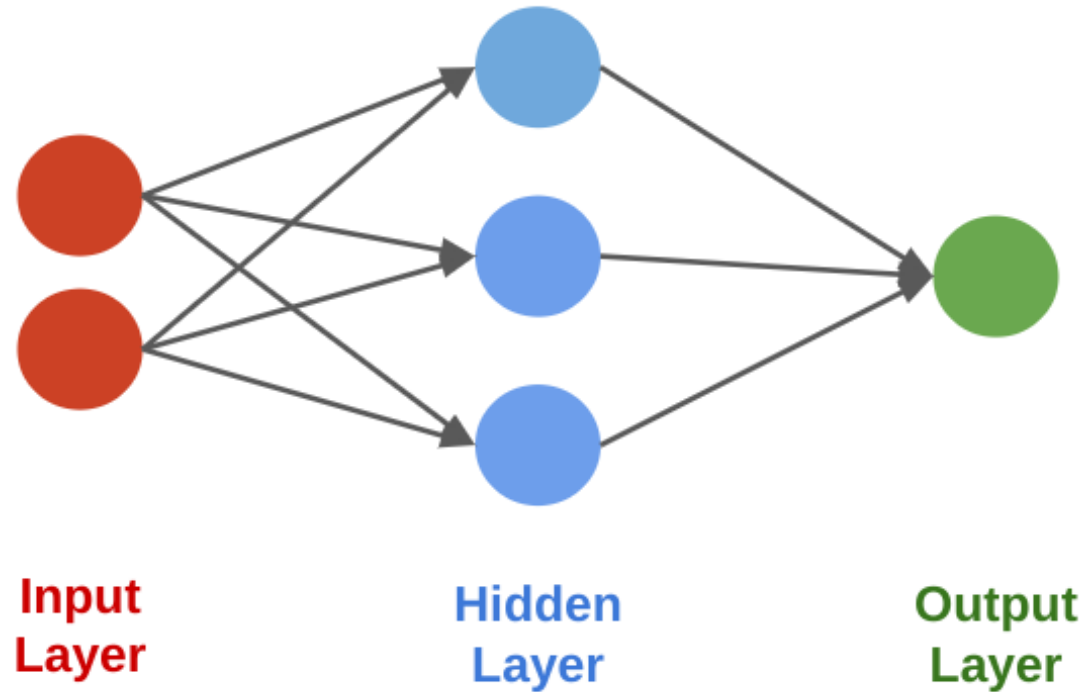
$$f(x) = \sigma(\theta^\top x) = \frac{1}{1 + \exp(-\theta^\top x)},$$

that can be interpreted as a neuron that uses the *sigmoid* as the activation function.

Neural Networks: Intuition

A neural network is a directed graph in which a node is a neuron that takes as input the outputs of the neurons that are connected to it.

Networks are typically organized in layers.

Neural Networks: Layers

A neural network layer is a model $f : \mathbb{R}^d \rightarrow \mathbb{R}^p$ that applies p neurons in parallel to an input x.

$$f(x) = \begin{bmatrix} \sigma(w_1^\top x) \\ \sigma(w_2^\top x) \\ \vdots \\ \sigma(w_p^\top x) \end{bmatrix}.$$

where each w_k is the vector of weights for the k-th neuron. We refer to p as the *size* of the layer.

The first output of the layer is a neuron with weights w_1 :

$$w_1^\mathsf{T}$$

$$x$$

$$\sigma(w_1^\mathsf{T} x)$$
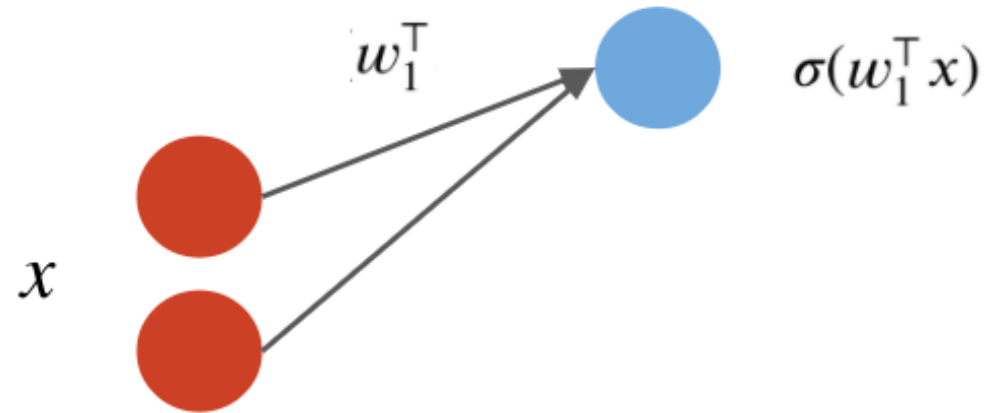
The second neuron has weights w_2 :

The third neuron has weights w_3 :
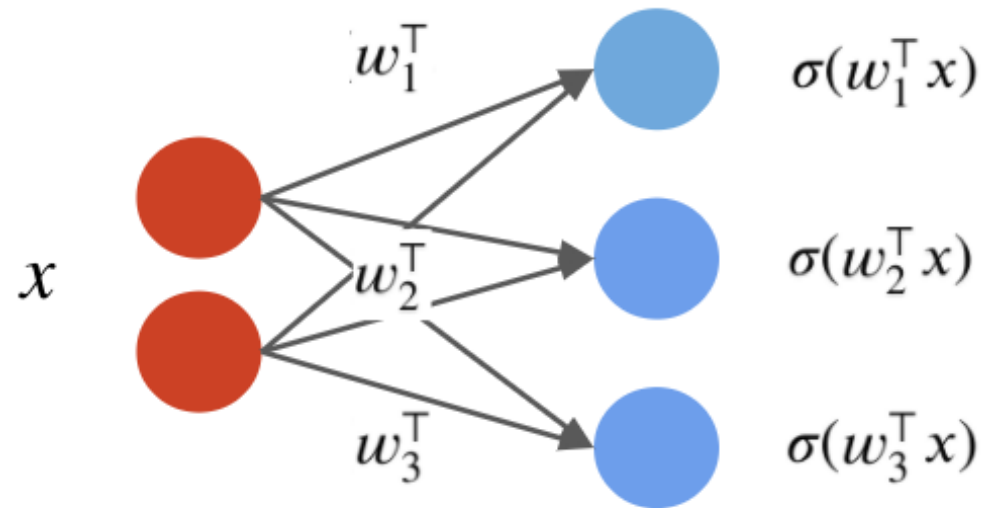
The parameters of the layer are w_1 , w_2 , w_3 .

By combining the w_k into one matrix W, we can write in a more succinct vectorized form:

$$f(x) = \sigma(W \cdot x) = \begin{bmatrix} \sigma(w_1^\top x) \\ \sigma(w_2^\top x) \\ \vdots \\ \sigma(w_p^\top x) \end{bmatrix},$$

where $\sigma(W \cdot x)_k = \sigma(w_k^\top x)$ and $W_{kj} = (w_k)_j$.

Visually, we can represent this as follows:
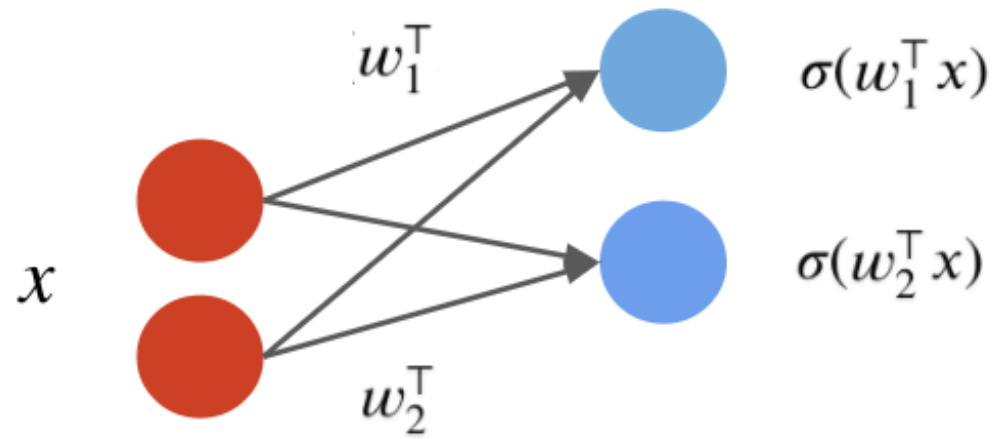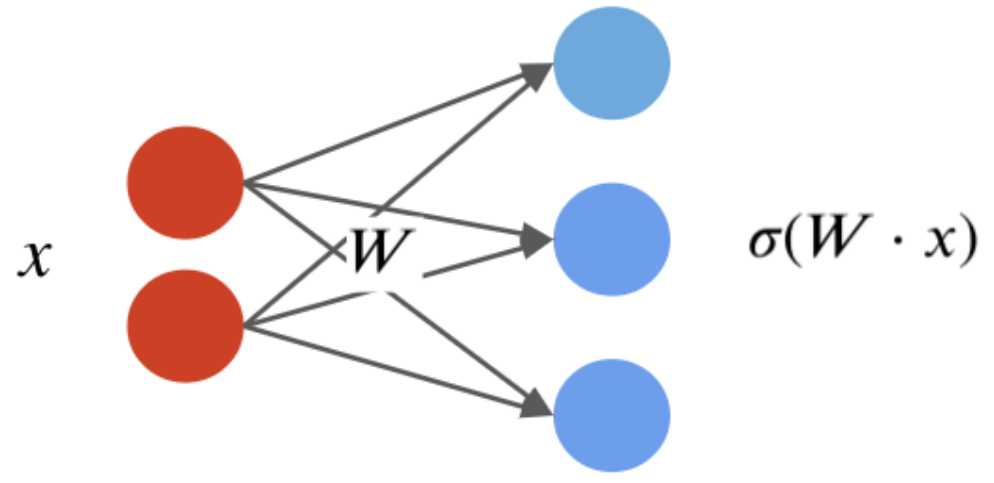
Neural Networks: Notation

A neural network is a model $f : \mathbb{R}^d \to \mathbb{R}$ that consists of a composition of L neural network layers:

$$f(x) = f_L \circ f_{L-1} \circ \ldots f_l \circ \ldots f_1(x).$$

The final layer f_L has size one (assuming the neural net has one ouput); intermediary layers f_l can have any number of neurons.

The notation $f \circ g(x)$ denotes the composition $f(g(x))$ of functions.

We can visualize this graphically as follows.
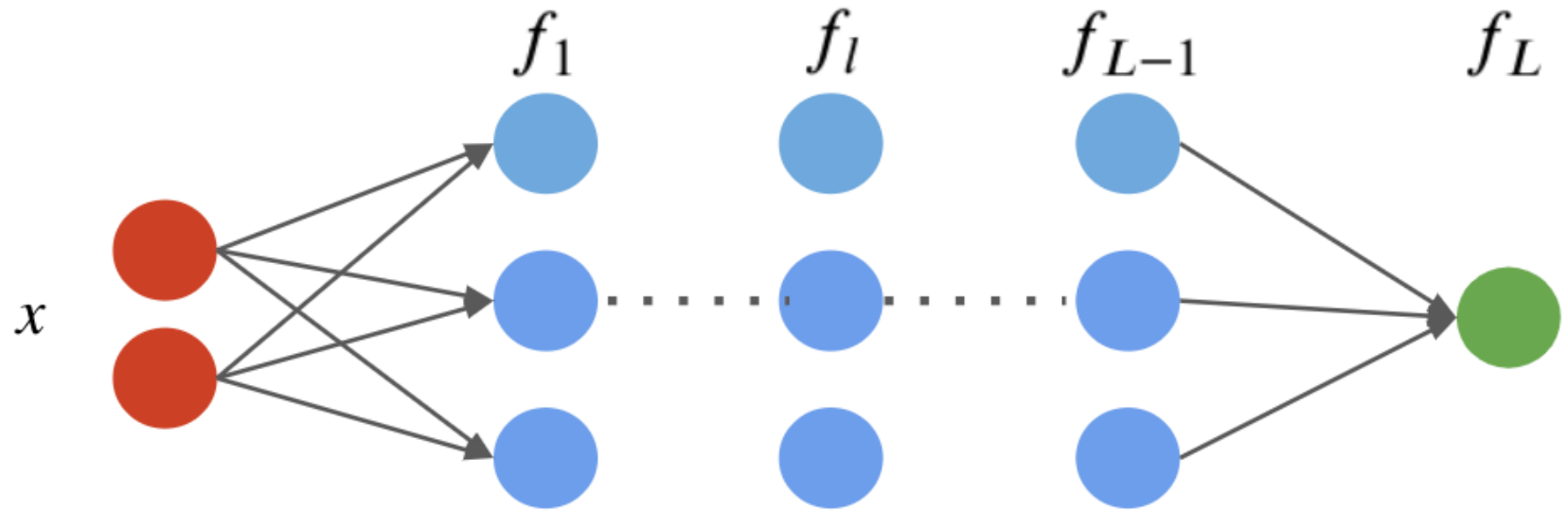
Example of a Neural Network

Let's implement a small two layer neural net with 3 hidden units.
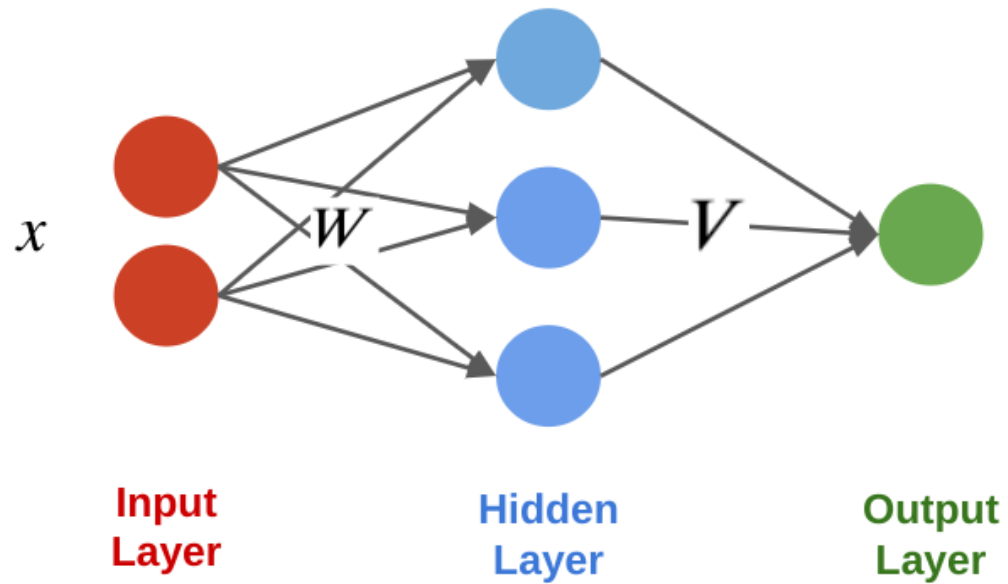
This implementation looks as follows.

```python
In [202]:  # a two layer network with logistic function as activation
           class Net():
               def __init__(self, x_dim, W_dim):
                   # weight matrix for layer 1
                   self.W = np.random.normal(size=(x_dim, W_dim))
                   # weight matrix for layer 2, also the output layer
                   self.V = np.random.normal(size=(W_dim, 1))
                   # activation function
                   self.afunc = lambda x: 1/(1+np.exp(-x))

               def predict(self, x):
                   # get output of the first layer
                   l1 = self.afunc(np.matmul(x, self.W))
                   # get output of the second layer, also the output layer
                   out = self.afunc(np.matmul(l1, self.V))
                   return out
```

Later in this lecture, we will see how to train this model using gradient descent.

Types of Neural Network Layers

There are many types of neural network layers that can exist. Here are a few:

- Ouput layer: normally has one neuron and special activation function that depends on the problem
- Input layer: normally, this is just the input vector x.
- Hidden layer: Any layer between input and output.

- Dense layer: A layer in which every input is connected to every neuron.
- Convolutional layer: A layer in which the operation $w^\top x$ implements a mathematical [convolution (https://en.wikipedia.org/wiki/Convolution)](https://en.wikipedia.org/wiki/Convolution).
- Recurrent Layer: A layer in which a neuron's output is connected back to the input.

Algorithm: (Fully-Connected) Neural Network

- **Type**: Supervised learning (regression and classification).
- **Model family**: Compositions of layers of artificial neurons.
- **Objective function**: Any differentiable objective.
- **Optimizer**: Gradient descent.

Pros and Cons of Neural Nets

Neural networks are very powerful models.

- They are flexible, and can approximate any function.
- They work well over unstructured inputs like audio or images.
- They can achieve state-of-the-art perfomrance.

They also have important drawbacks.

- They can also be slow and hard to train.
- Large neworks require a lot of data.
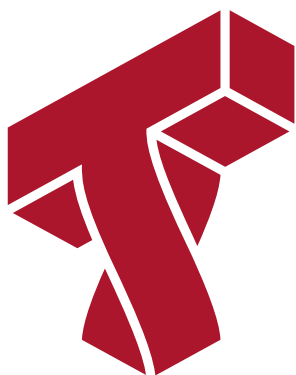
Part 3: Backpropagation

We have defined what is an artificial neural network.

Let's now see how we can train it so that it performs well on given tasks.

Review: Neural Network Layers

A neural network layer is a model $f : \mathbb{R}^d \to \mathbb{R}^p$ that applies p neurons in parallel to an input x.

$$f(x) = \sigma(W \cdot x) = \begin{bmatrix} \sigma(w_1^\top x) \\ \sigma(w_2^\top x) \\ \vdots \\ \sigma(w_p^\top x) \end{bmatrix},$$

where each w_k is the vector of weights for the k-th neuron and $W_{kj} = (w_k)_j$. We refer to p as the *size* of the layer.

Review: Neural Networks

A neural network is a model $f : \mathbb{R} \to \mathbb{R}$ that consists of a composition of L neural network layers:

$$f(x) = f_L \circ f_{L-1} \circ \ldots f_1(x).$$

The final layer f_L has size one (assuming the neural net has one ouput); intermediary layers f_l can have any number of neurons.

The notation $f \circ g(x)$ denotes the composition $f(g(x))$ of functions

We can visualize this graphically as follows.
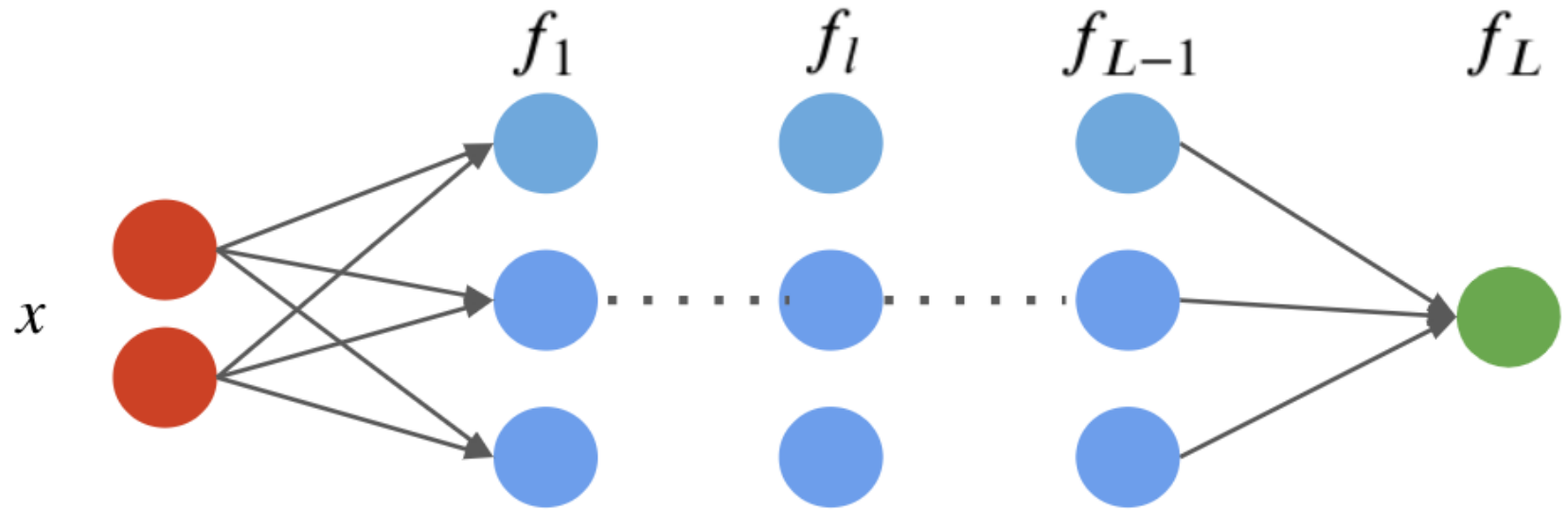
Review: The Gradient

The gradient $\nabla_\theta f$ further extends the derivative to multivariate functions $f : \mathbb{R}^d \to \mathbb{R}$, and is defined at a point θ as

$$\nabla_\theta f(\theta) = \begin{bmatrix} \frac{\partial f(\theta)}{\partial \theta_1} \\ \frac{\partial f(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(\theta)}{\partial \theta_d} \end{bmatrix}.$$

In other words, the j-th entry of the vector $\nabla_\theta f(\theta)$ is the partial derivative $\frac{\partial f(\theta)}{\partial \theta_j}$ of f with respect to the j-th component of θ.

Review: Gradient Descent

If we want to optimize an objective $J(\theta)$, we start with an initial guess θ_0 for the parameters and repeat the following update until the function is no longer decreasing:
$$\theta_i := \theta_{i-1} - \alpha \cdot \nabla_\theta J(\theta_{i-1}).$$

As code, this method may look as follows:

```python
theta, theta_prev = random_initialization()
while norm(theta - theta_prev) > convergence_threshold:
    theta_prev = theta
    theta = theta_prev - step_size * gradient(theta_prev)
```

Backpropagation

To apply gradient descent, we need to calculate gradients for every parameter in a neural network model f:

$$\frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1}, \cdots, \frac{\partial J}{\partial \theta_d}$$

It might be possible to do it manually when the network is small. But it is nearly impossible and very much error-prone to compute gradients for larger networks.

Backpropagation is a way of calculating gradients efficiently for neural network models with arbitrary number of layers and neurons.

The core idea of it is something we are actually very familiar with: the chain rule.

Review: Chain Rule of Calculus

If we have two differentiable functions $f(x)$ and $g(x)$, and
$$F(x) = f \circ g(x)$$
then the derivative of $F(x)$ is:
$$F'(x) = f'(g(x)) \cdot g'(x).$$

Let $y = f(u)$ and $u = g(x)$, we also have:
$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}.$$

Chain Rule in Neural Nets

A neural network is a model $f : \mathbb{R} \to \mathbb{R}$ that consists of a composition of L neural network layers:

$$f(x) = f_L \circ f_{L-1} \circ \ldots f_1(x).$$

Let y_l denote the output $f_l \circ f_{-1} \circ f_1(x)$ of layer l.

The chain rule tells us to compute $\frac{\partial J}{\partial \theta_l}$ for all parameters θ_l in layer l. We can break the computation down as:

$$\frac{\partial J}{\partial \theta_l} = \frac{\partial J}{\partial y_L} \frac{\partial y_L}{\partial y_{L-1}} \cdots \frac{\partial y_{l+1}}{\partial y_l} \frac{\partial y_l}{\partial \theta_l},$$

where $y_L, y_{L-1} \cdots y_l$ are the outputs from each layer.

Note that the computation of $\frac{\partial J}{\partial y_l}$ can be re-used for computing gradients for all θ in layers before l.

This is the key idea of backpropagation: local gradients computation for each layer can be 'chained' to obtain gradients.

Illustration:

Let's start with a single layer $y = f(x; \theta)$:

$$x \longrightarrow \boxed{f(\theta)} \longrightarrow y$$
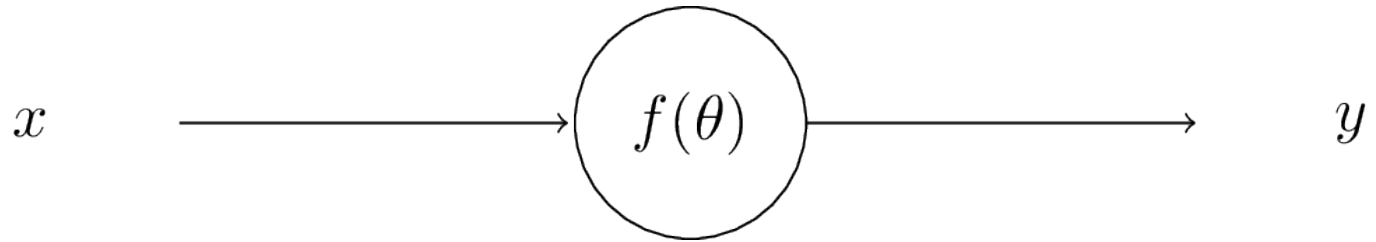
With the output y, target label $\hat{y}$, and loss function J, we can compute the loss (error) of the prediction.

$$x \longrightarrow f(\theta) \longrightarrow y$$

$$J(y, \hat{y})$$
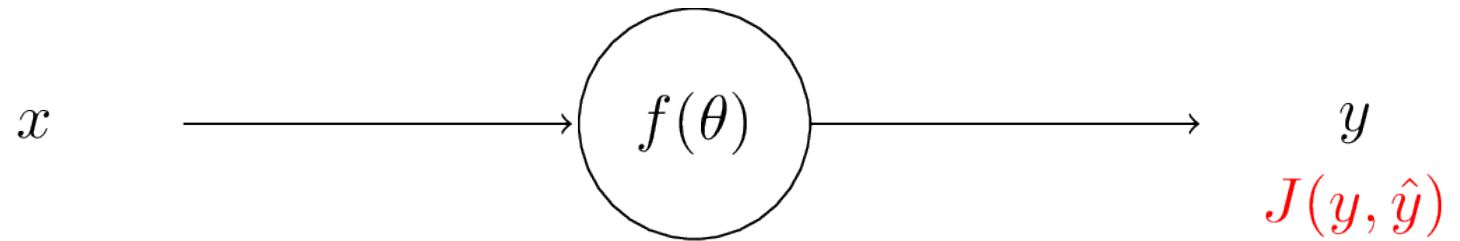
The backpropagation starts from the output layer and moves backwards.

We first need to compute the gradients of the loss to the output.

$$x \longrightarrow f(\theta) \longrightarrow y$$

$$\frac{\partial J}{\partial y} \qquad J(y, \hat{y})$$
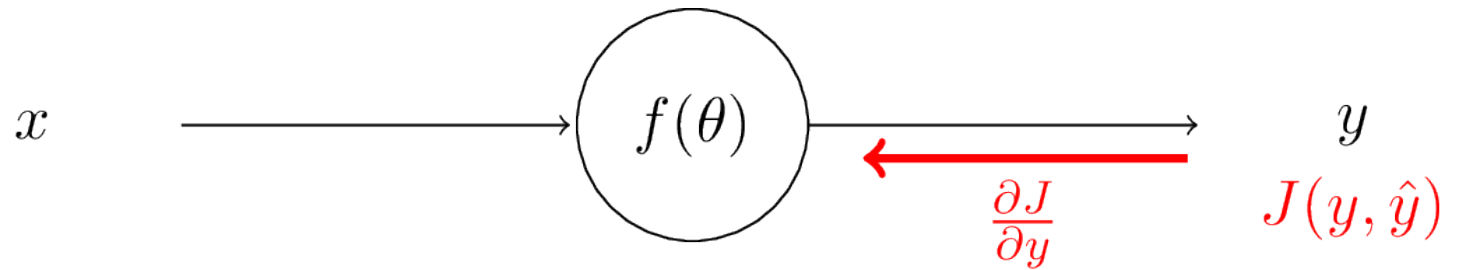
After we have those, then using the chain rule, we can compute the gradients with respect to the network parameters θ.

$$x \qquad \longrightarrow \qquad f(\theta) \qquad \longrightarrow \qquad y$$

$$\frac{\partial J}{\partial y} \qquad\qquad J(y, \hat{y})$$

$$\frac{\partial J}{\partial \theta} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial \theta}$$
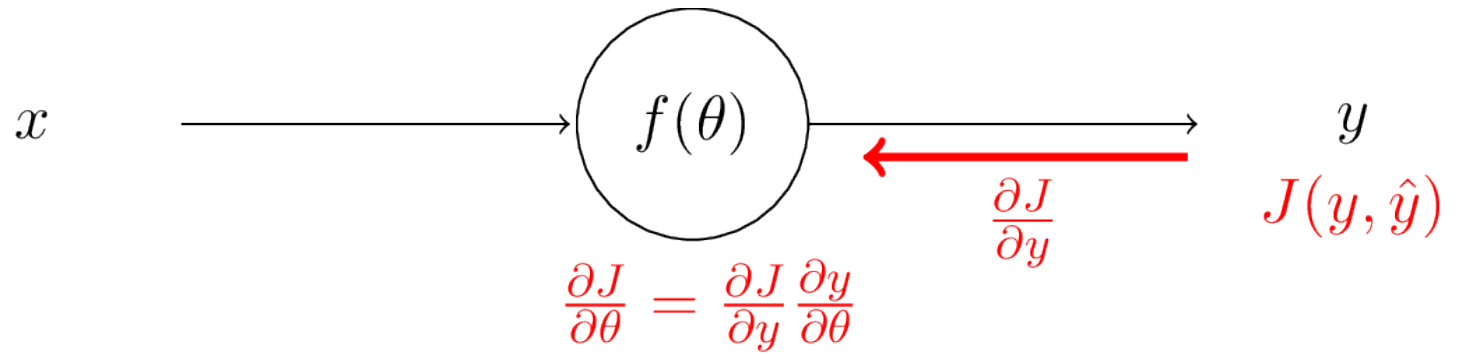
We can keep working upstream and compute gradients to the input. After that we finish the backpropagation in this layer.

$$x \qquad\qquad \xrightarrow{\hspace{3cm}} \quad f(\theta) \quad \xrightarrow{\hspace{3cm}} \qquad y$$

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x}$$

$$\frac{\partial J}{\partial \theta} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial \theta}$$

$$\frac{\partial J}{\partial y}$$

$$J(y, \hat{y})$$
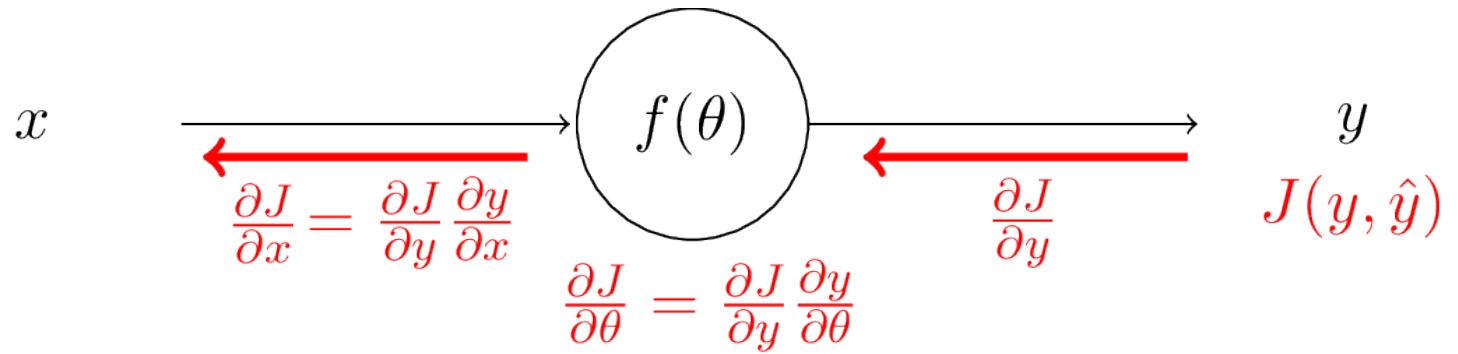
We can apply this process recursively to obtain derivatives for any number of layers.

$$x \quad \longrightarrow \cdots \longrightarrow \left(f_{l-1} \right) \longrightarrow \left(f_l \right) \longrightarrow \left(f_{l+1} \right) \longrightarrow \cdots \longrightarrow \quad y$$

$$\frac{\partial J}{\partial x} \qquad\qquad \frac{\partial J}{\partial y_{l-1}} \quad \frac{\partial J}{\partial y_l} \qquad\qquad \frac{\partial J}{\partial y} \quad J(y, \hat{y})$$
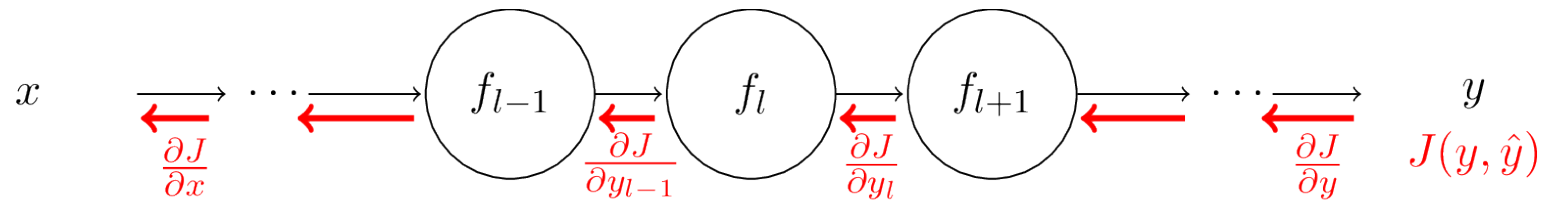
Backprogragation by Hand

Let's work out by hand what backpropagation would do on our two layer neural network.
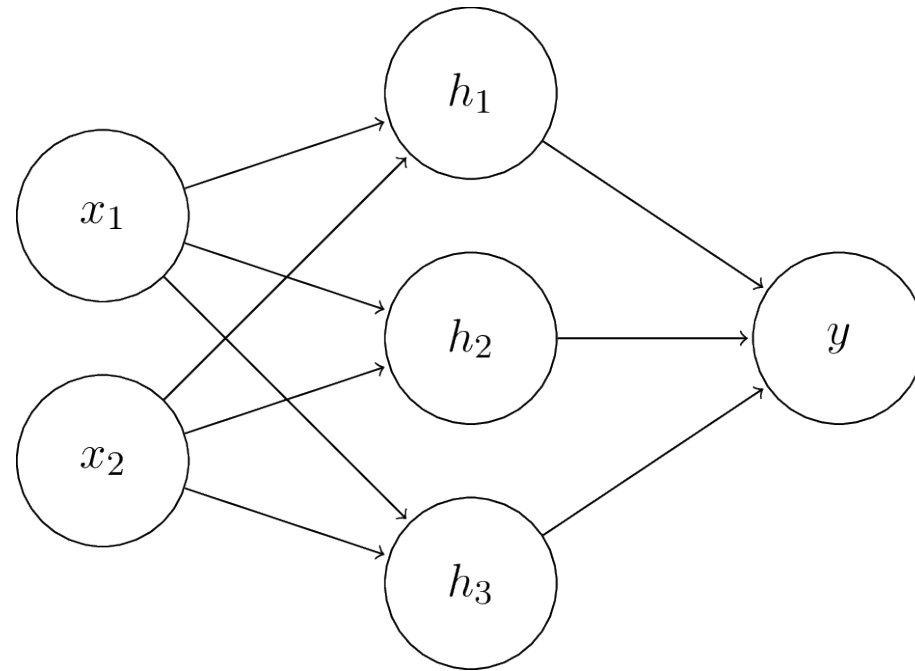
For our two layer fully connected network with sigmoid activation, the network is composed of following functions:

$$\mathbf{h} = \sigma(\mathbf{W}^T \mathbf{x})$$

In our example, we have the following values:

$$\mathbf{x} = [5.0, 3.0]^T, \quad \hat{y} = 1 \text{ means it is positive class.}$$

$$\mathbf{W} = \begin{bmatrix} 1.0 & -1.0 & 3.0 \\ 2.0 & 2.0 & -1.0 \end{bmatrix}$$

$$\mathbf{V} = [0.1, 0.5, -0.1]^T$$

node value 0.9999833

1.0

h_1

5.0 x_1 2.0

h_2 y

3.0 x_2

h_3
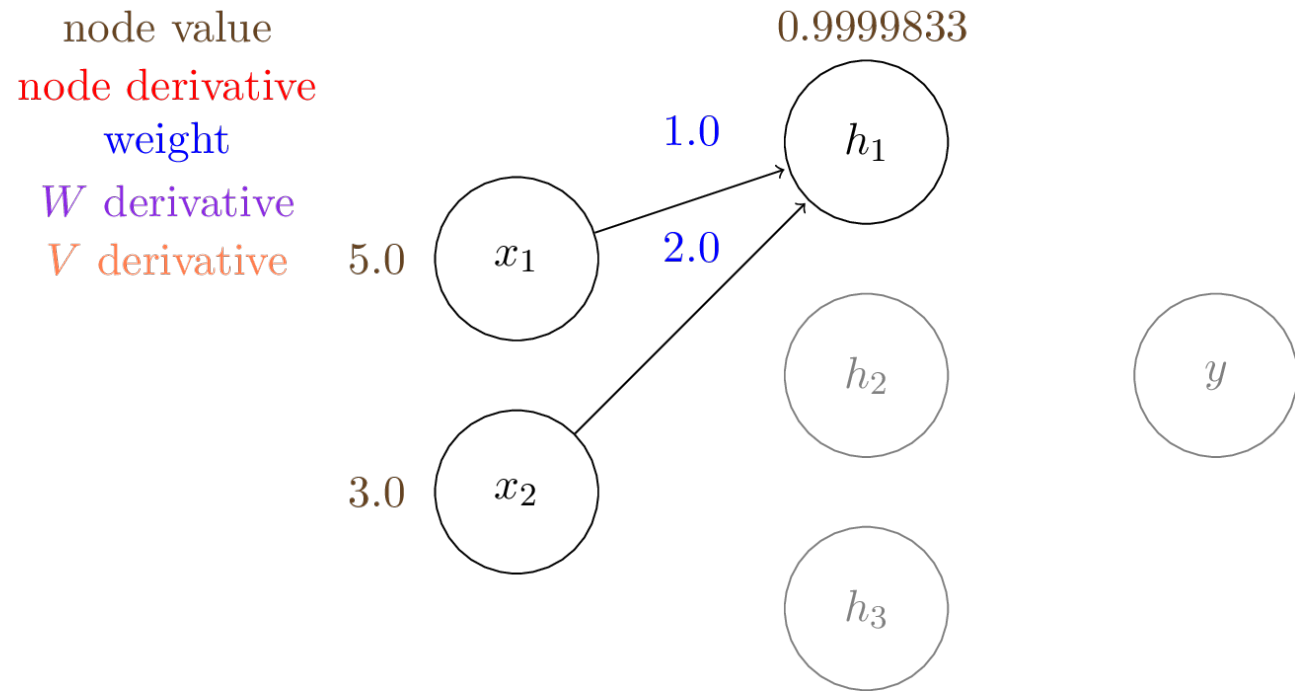
We can compute the output of the hidden layer, **h**:

$$h_1 = \sigma(W_{11} \cdot x_1 + W_{21} \cdot x_2) = \sigma(1.0 \times 5.0 + 2.0 \times 3.0) = 0.99998329857$$
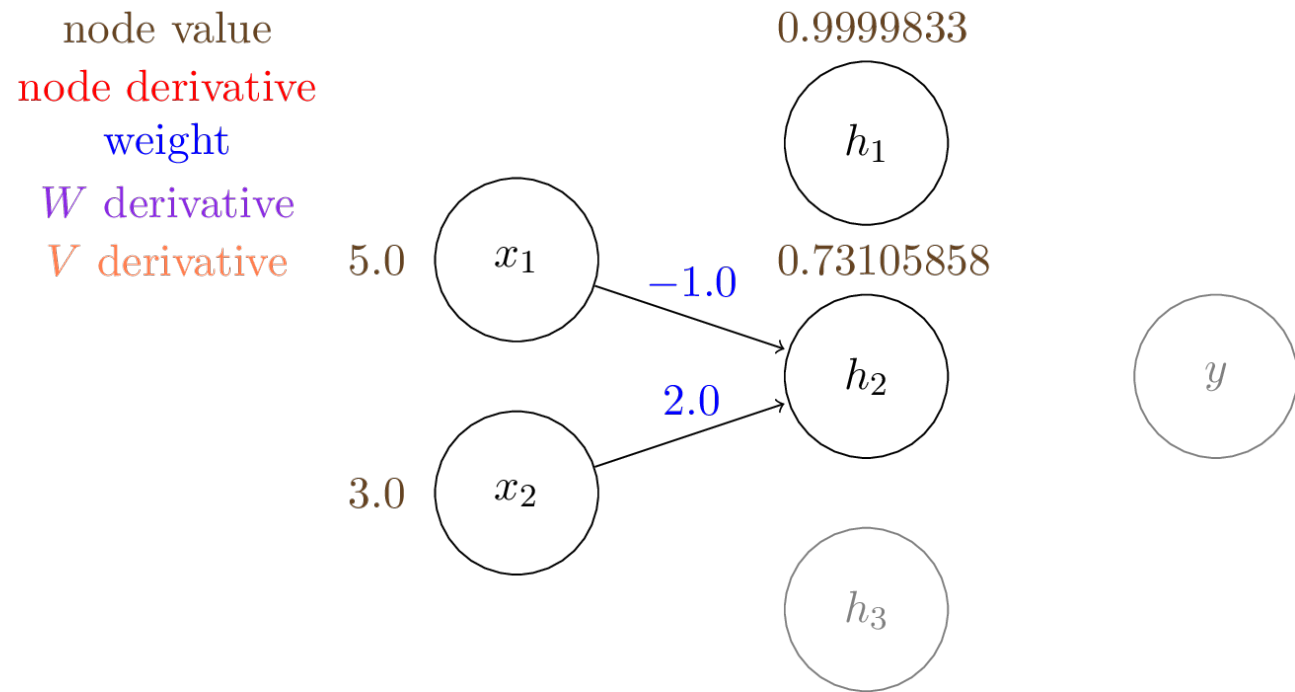
node value   0.9999833
node derivative
weight h_1
W derivative
V derivative   5.0 x_1 0.73105858
                          -1.0
                                                  h_2 y
                     2.0
           3.0 x_2

                                                  h_3

We can compute the output of the hidden layer, **h**:

$$h_1 = \sigma(W_{11} \cdot x_1 + W_{21} \cdot x_2) = \sigma(1.0 \times 5.0 + 2.0 \times 3.0) = 0.9999$$

node value

node value

node derivative

weight

W derivative

V derivative

0.9999833

h_1

5.0 x_1

0.73105858

h_2

y

3.0

3.0 x_2 -1.0 0.99999386

h_3

We can compute the output of the hidden layer, **h**:

node value
node derivative
weight
W derivative
V derivative

0.9999833

h_1

0.1

5.0 x_1 0.73105858

0.5

h_2 y 0.590378

-0.1

3.0 x_2 0.99999386

h_3
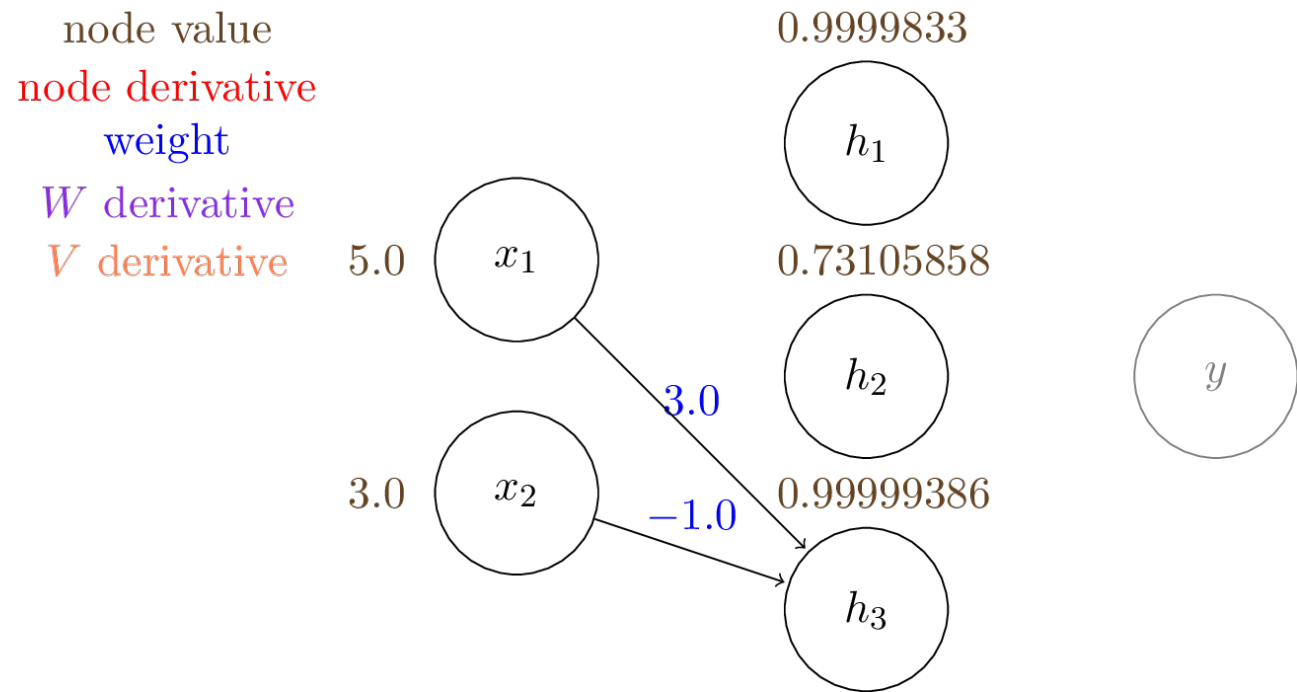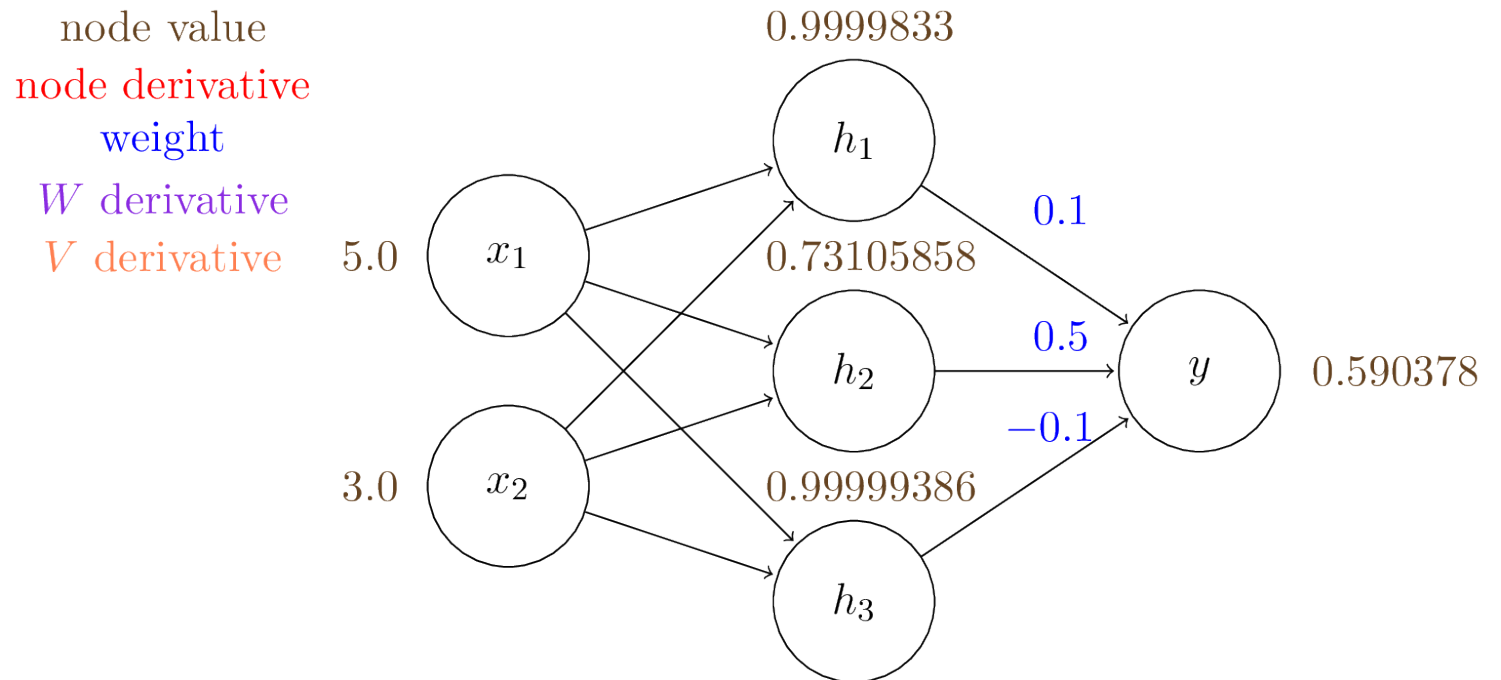
Similarly we can get the output of y:

$$y = \sigma(V_1 \cdot h_1 + V_2 \cdot h_2 + V_3 \cdot h_3) = 0.590378$$

node value
node derivative
weight
W derivative
V derivative
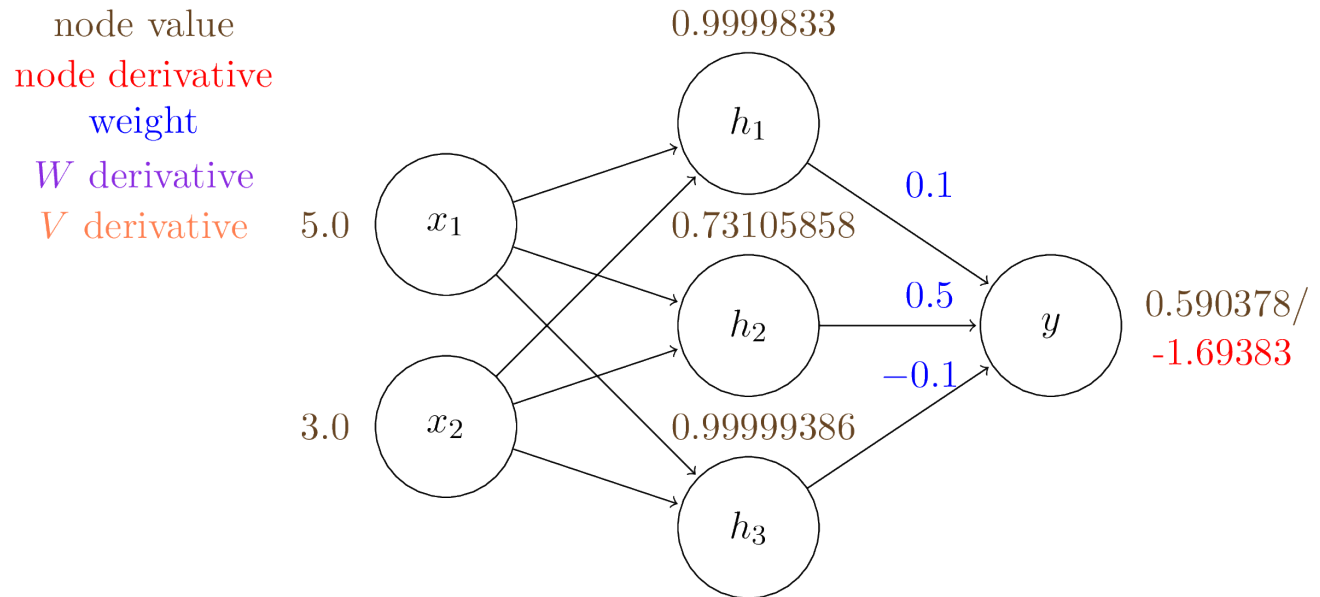
Next, we compute the binary cross entropy loss with the target label 1:
$J(y, \hat{y}) = - \log(y) = 0.52699$

We can also compute the gradient (shown in red): $\frac{\mathrm{d}J}{\mathrm{d}y} = -1/y = -1.69383$

node value
node derivative
weight
W derivative
V derivative

0.9999833

h_1

0.1

5.0 x_1 0.73105858

0.5

h_2 y $0.590378/$
-1.69383

-0.1

3.0 x_2 0.99999386

h_3
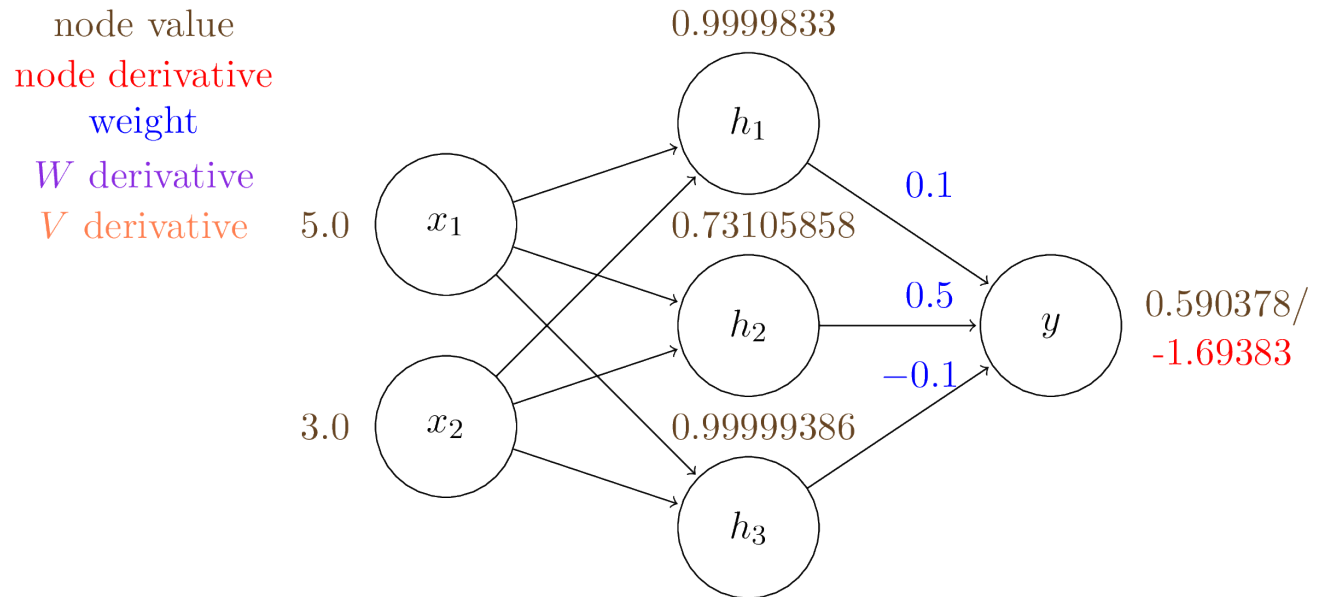
We are going to compute the gradients to the weights in the output layer:

$$\frac{\mathrm{d}J}{\mathrm{d}V} = \frac{\mathrm{d}J}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}V}$$

Recall $y = \sigma(\mathbf{V}^T\mathbf{h}) = \sigma(V_1 \cdot h_1 + V_2 \cdot h_2 + V_3 \cdot h_3)$ and $\sigma' = \sigma(1 - \sigma)$:

node value
node derivative
weight
W derivative
V derivative

0.9999833

h_1

$0.1/-0.41$

5.0 x_1 0.73105858

$0.5/-0.3$ y $0.59/$
h_2 -1.69

3.0 x_2 0.99999386

$-0.1/-0.41$

h_3
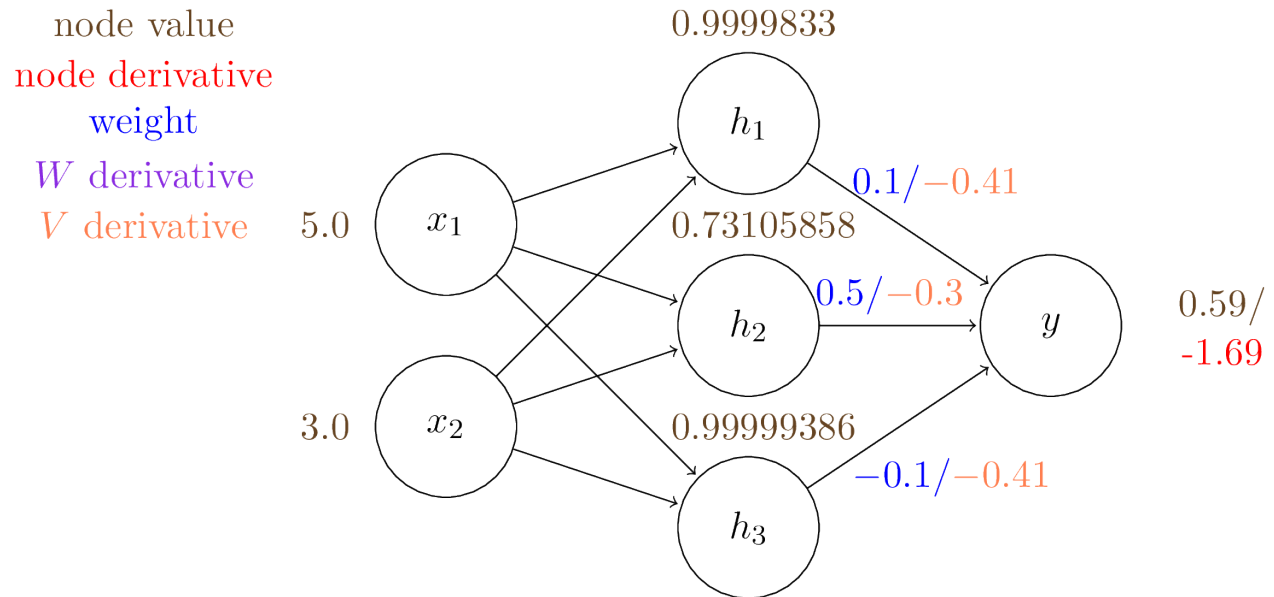
Applying these formulas, we obtain the gradients of $\mathbf{V}$:

$$\frac{\partial J}{\partial V_1} = \frac{\mathrm{d}J}{\mathrm{d}y}\frac{\partial y}{\partial V_1} = -1.69 \times 0.59 \times (1 - 0.59) \times 0.99998 = -0.41$$

node value
node derivative
weight
W derivative
V derivative

5.0 x_1

3.0 x_2

0.9999833

h_1

0.73105858

h_2

0.99999386

h_3

0.1/−0.41

0.5/−0.3

−0.1/−0.41

y

0.59/
-1.69
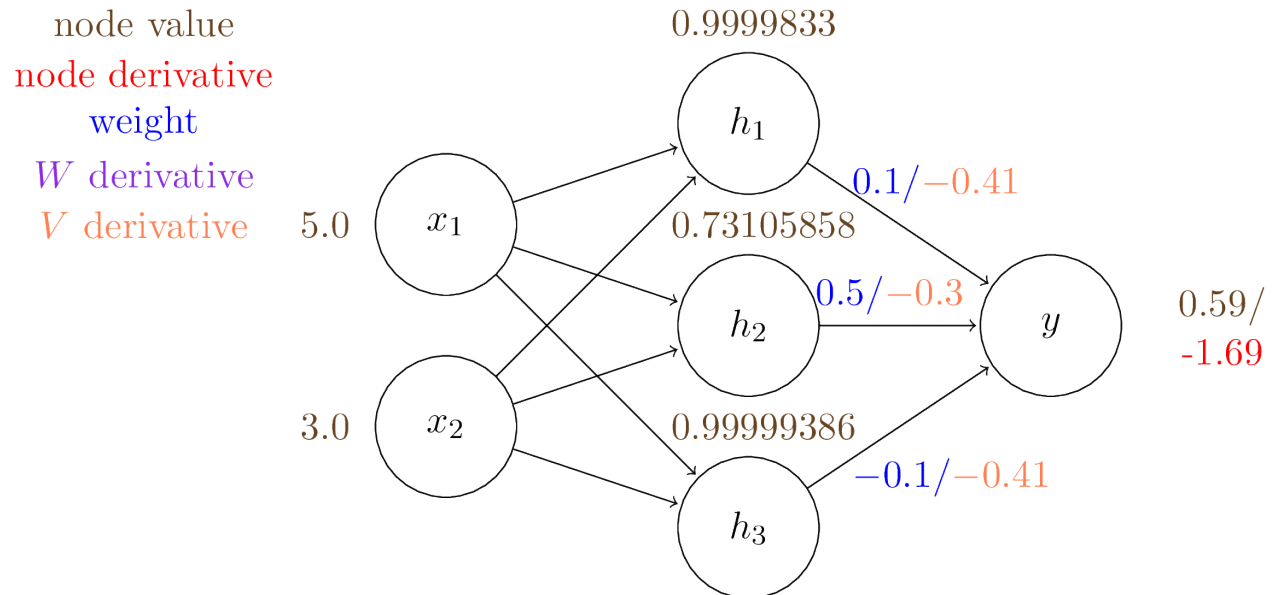
Next, we want to compute gradients at the hidden layer:

$$\frac{\mathrm{d}\boldsymbol{J}}{\mathrm{d}h} = \frac{\mathrm{d}\boldsymbol{J}}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}h}$$
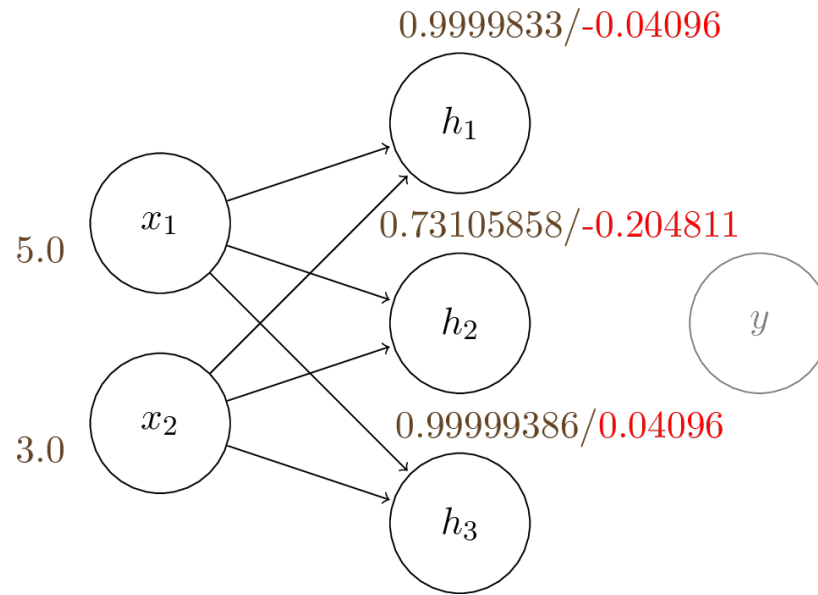
Similarly to the previous slide:
$$\frac{\partial J}{\partial h_1} = \frac{\mathrm{d}J}{\mathrm{d}y}\frac{\partial y}{\partial h_1} = -1.69 \times 0.59 \times (1 - 0.59) \times 0.1 = -0.04096$$

node value
node derivative
weight
W derivative
V derivative

5.0

3.0

$0.9999833/\text{-}0.04096$

$1.0/\text{-}3.\text{e-}6$

h_1

x_1

$2.0/\text{-}2\text{e-}6$

h_2

y

x_2

h_3

Since we have another linear layer with sigmoid activation, the way we compute gradients will be the same as in the output layer.
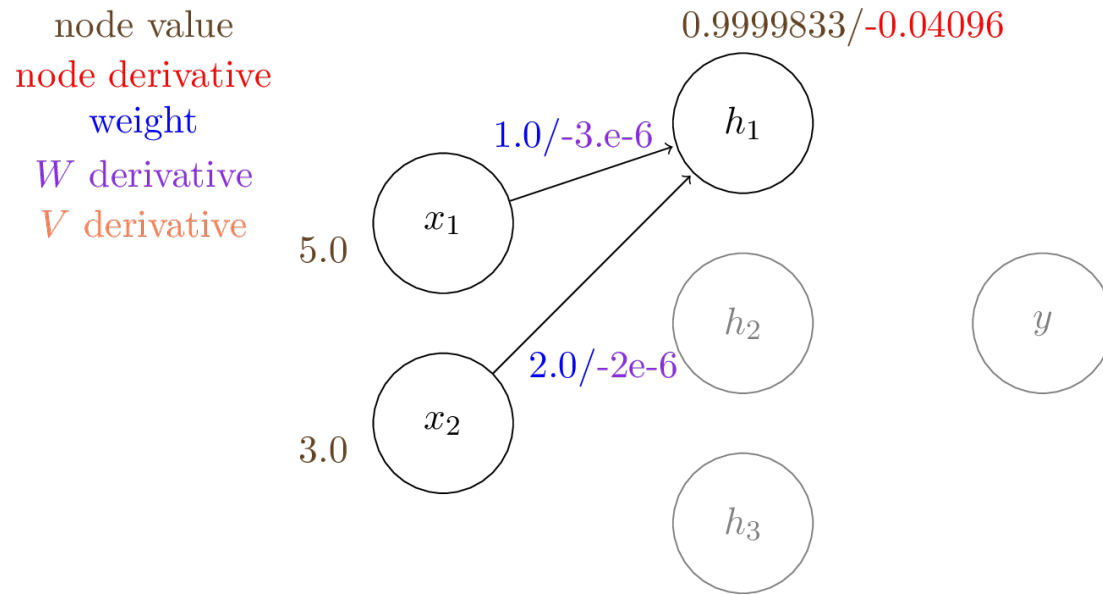
node value
node derivative
weight
W derivative
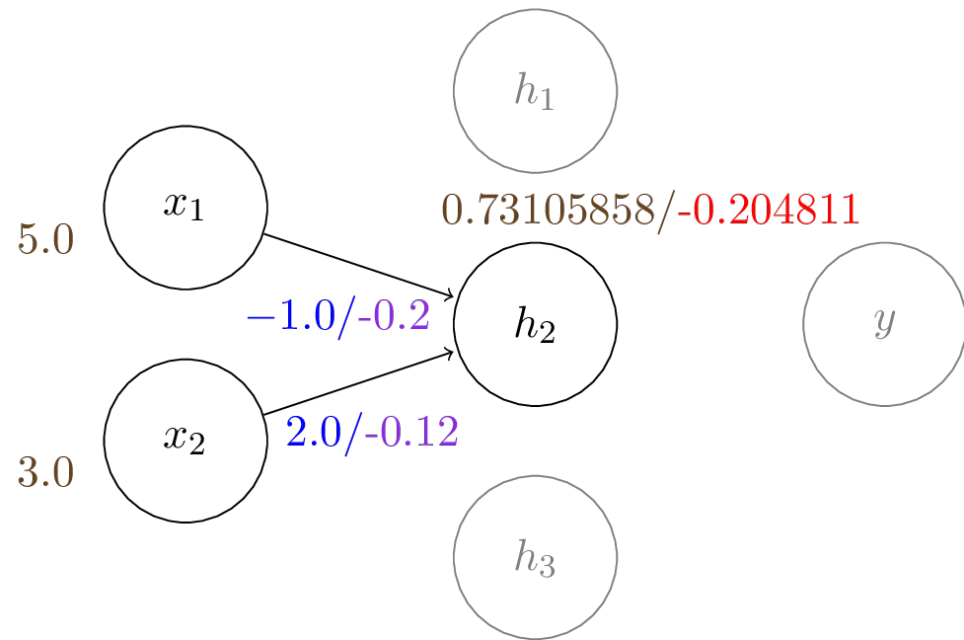V derivative

h_1

x_1

5.0

0.73105858/-0.204811

y

-1.0/-0.2

h_2

x_2

2.0/-0.12

3.0

h_3

We can compute the rest the same way.

node value
node derivative
weight
W derivative
V derivative

h_1

x_1

5.0

3.0/1e-6

h_2

y

x_2

3.0

0.99999386/0.04096

$-1.0/7\text{e-}7$

h_3
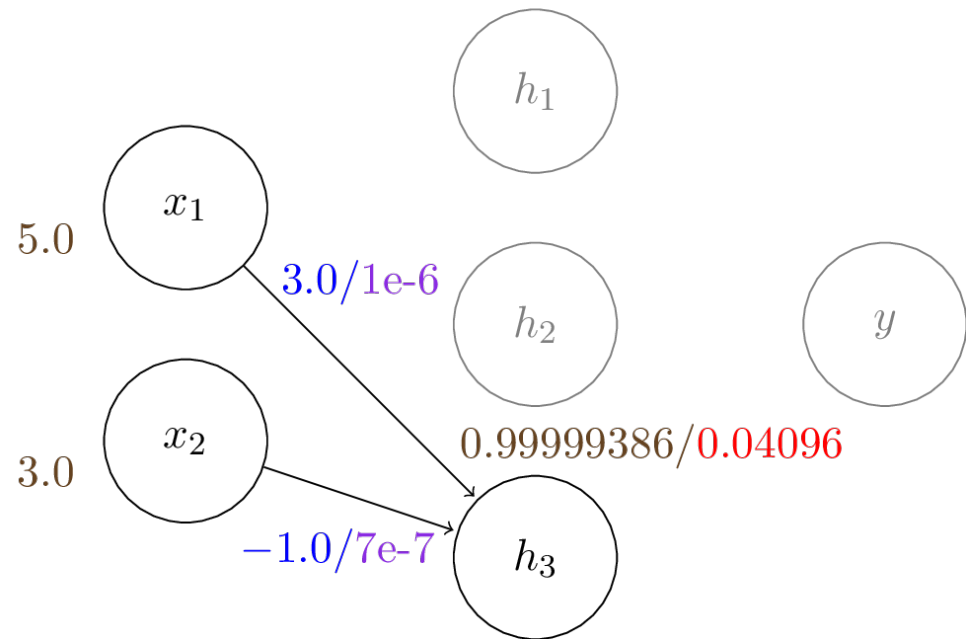
And now we have the gradients to all the learnable weights in this two layer network and
we can tune the weights by gradient descenet.

Now let's implement backprop with the simple neural network model we defined earlier.

We start by implementing the building block of our network: a linear layer with sigmoid activation.

```
In [81]:  import numpy as np

          # a single linear layer with sigmoid activation
          class LinearSigmoidLayer():
              def __init__(self, in_dim, out_dim):
                  self.W = np.random.normal(size=(in_dim,out_dim))
                  self.W_grad = np.zeros_like(self.W)

                  self.afunc = lambda x: 1. / (1. + np.exp(-x))

              # forward function to get output
              def forward(self, x):
                  Wx = np.matmul(x, self.W)
                  self.y = self.afunc(Wx)
                  self.x = x
                  return self.y

              # backward function to compute gradients
              def backward(self, grad_out):
                  self.W_grad = np.matmul(
                      self.x.transpose(),
                      self.y * (1-self.y) * grad_out,
```

Then we can stack the single layers to construct a two layer network.

```
In [82]:  # a two layer network with logistic function as activation
          class Net():
              def __init__(self, x_dim, W_dim):
                  self.l1 = LinearSigmoidLayer(x_dim, W_dim)
                  self.l2 = LinearSigmoidLayer(W_dim, 1)

              # get output
              def predict(self, x):
                  h = self.l1.forward(x)
                  self.y = self.l2.forward(h)
                  return self.y

              # backprop
              def backward(self, label):
                  # binary cross entropy loss, and gradients
                  if label == 1:
                      J = -1*np.log(self.y)
                      dJ = -1/self.y
                  else:
                      J = -1*np.log(1-self.y)
                      dJ = 1/(1-self.y)
```

We can run with our previous example to check if the results are consistent with our
manual computation.

```
In [85]:   model = Net(2, 3)
           model.l1.W = np.array([[1.0,-1.0,3.0],[2.0,2.0,-1.0]])
           model.l2.W = np.array([[0.1],[0.5],[-0.1]])

           x = np.array([5.0, 3.0])[np.newaxis,...]
           x_label = 1

           # forward
           out = model.predict(x)

           # backward
           loss = model.backward(label=x_label)

           print('loss: {}'.format(loss))
           print('W grad: {}'.format(model.l1.W_grad))
           print('V grad: {}'.format(model.l2.W_grad))
```

```
loss: [[0.52699227]]
W grad: [[-3.42057777e-06 -2.01341432e-01  1.25838681e-06]
 [-2.05234666e-06 -1.20804859e-01  7.55032084e-07]]
V grad: [[-0.40961516]
 [-0 29945768]
```

Another sanity check is to perform gradient descent on the single sample input and see if we can achieve close to zero loss.

You can try to change the target label below to see the network is able to adapt in either case.

```
In [95]:  ## gradient descent
          loss = []
          score = []
          for i in range(100):
              out = model.predict(x)
              loss.append(model.backward(label=1)) # 1 for positive, 0 for negative
              model.grad_step(lr=1e-1)
              score.append(out)

          import matplotlib.pyplot as plt
          plt.plot(np.array(loss).squeeze(),'-')
          plt.plot(np.array(score).squeeze(),'.')
```

Out[95]:   [<matplotlib.lines.Line2D at 0x7f8c0ed09f10>]
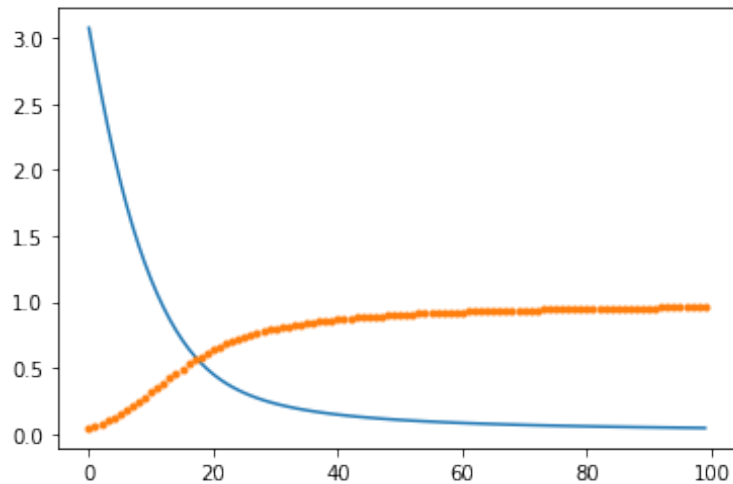
Summary

- Neural networks are powerful models that can approximate any function.
- They are trained using gradient descent.
- In order to compute gradients, we use an efficient algorithm called backpropagation.