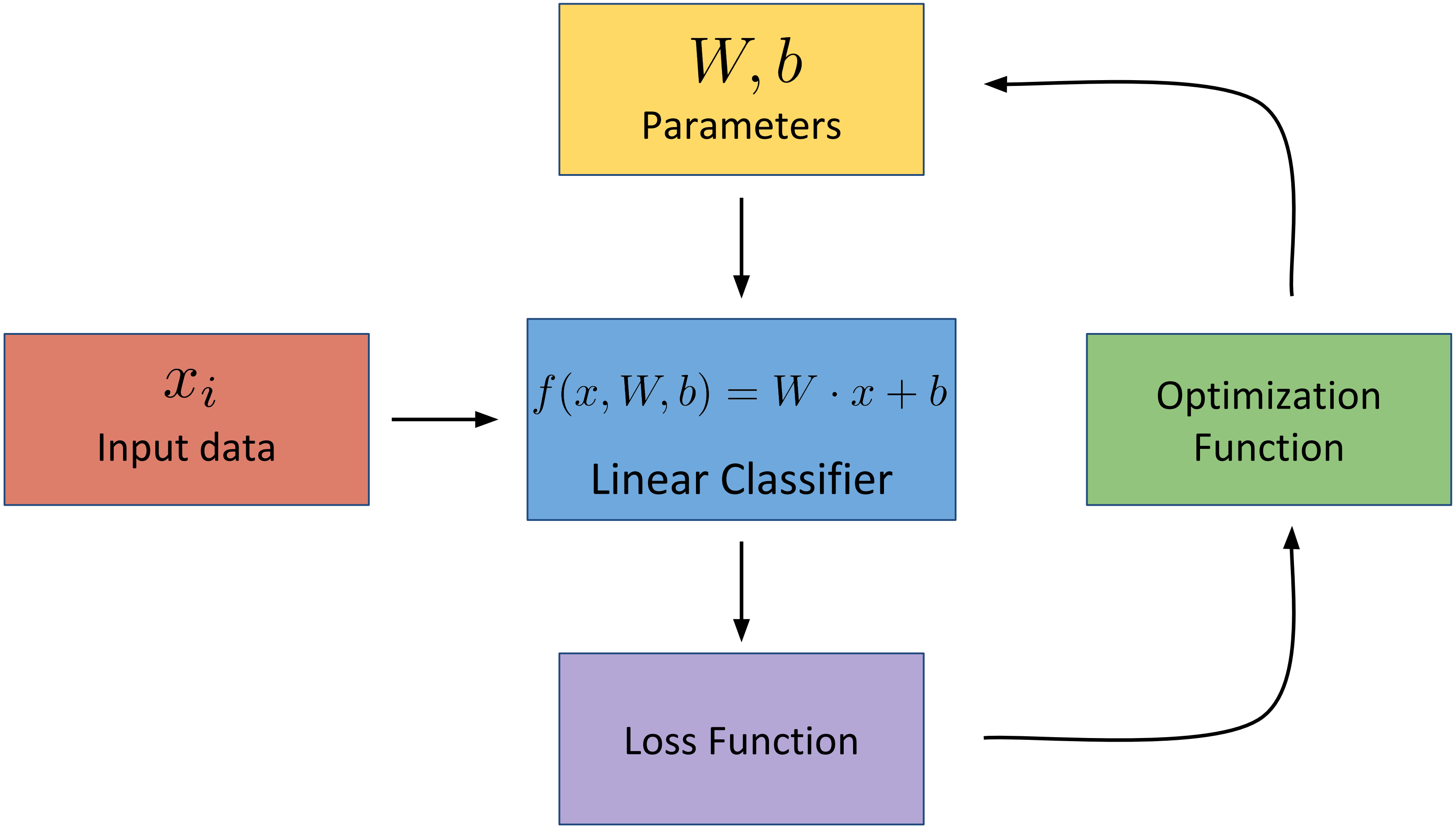
Course series: Deep Learning for Machine Translation

# Machine Learning II

Lecture # 4

Hassan Sajjad and Fahim Dalvi
Qatar Computing Research Institute, HBKU

# Recap



$W, b$

Parameters

$x_i$

Input data

$f(x, W, b) = W \cdot x + b$

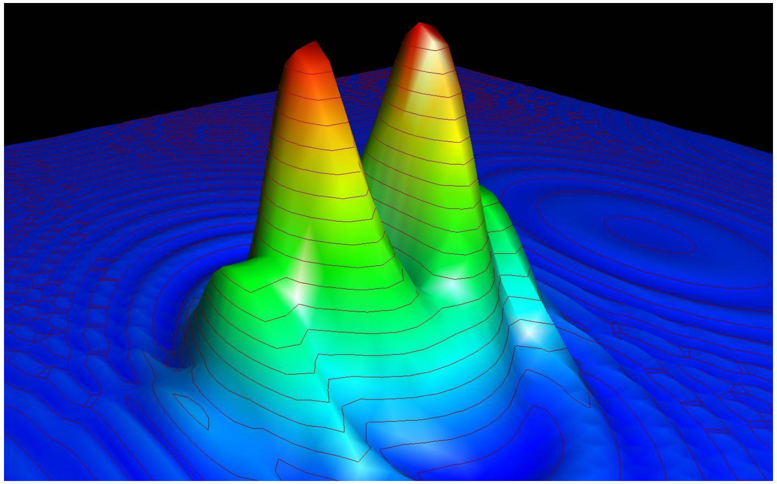
Linear Classifier

Optimization Function

Loss Function

# Optimization

- Recall from Lecture 3:
  - For every training example, we compute a loss
  - We then use the loss to adjust the parameters
  - Updated parameters should result in a lower loss
- We “adjust” by moving in the direction of the slope of the function



# Optimization

- How can we compute the slope of the function?
  - Compute gradients analytically
  - Backpropagation

# Optimization

Let us consider the *linear classifier* and *Mean Squared Error* from last lecture:

Objective Function

$$\begin{aligned} f(x, W, b) &= W \cdot x + b \\ &= w_0 \cdot x_0 + w_1 \cdot x_1 + b \end{aligned}$$

Loss Function

$$MSE(x, W, b, y) = (f(x, W, b) - y)^2$$

# Partial Derivative Rules

$$f(x) = x^n$$

$$\frac{\partial f}{\partial x} = n \cdot x^{(n-1)}$$

$$f(x) = \max(1, x^2)$$

$$\frac{\partial f}{\partial x} = \begin{cases} 2x, & \text{if } x^2 \geq 1 \\ 0, & \text{if } x^2 < 1 \end{cases}$$

$$f(x, y) = x \cdot y$$

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

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

$$f(x, y) = x + y$$

$$\frac{\partial f}{\partial x} = 1$$

$$\frac{\partial f}{\partial y} = 1$$

$$f(x) = 5x$$

$$\frac{\partial f}{\partial x} = 5$$

# Optimization

Let us compute the gradient of MSE analytically

$$\mathcal{L} = (f(x, W, b) - y)^2$$

# Optimization

Let us compute the gradient of MSE analytically

$$\mathcal{L} = (f(x, W, b) - y)^2$$

$$\frac{\partial \mathcal{L}}{\partial w_0} = \frac{\partial}{\partial w_0}(f(x, W, b) - y)^2$$

# Optimization

Let us compute the gradient of MSE analytically

$$\mathcal{L} = (f(x, W, b) - y)^2$$

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_0} &= \frac{\partial}{\partial w_0}(f(x, W, b) - y)^2 \\ &= 2 \cdot (f(x, W, b) - y) \cdot \frac{\partial}{\partial w_0}(f(x, W, b) - y) \end{aligned}$$

# Optimization

Let us compute the gradient of MSE analytically

$$\mathcal{L} = (f(x, W, b) - y)^2$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_0} &= \frac{\partial}{\partial w_0}(f(x, W, b) - y)^2 \\
&= 2 \cdot (f(x, W, b) - y) \cdot \frac{\partial}{\partial w_0}(f(x, W, b) - y) \\
&= 2 \cdot (f(x, W, b) - y) \cdot \frac{\partial}{\partial w_0}(w_0 \cdot x_0 + w_1 \cdot x_1 + b - y)
\end{aligned}$$

# Optimization

Let us compute the gradient of MSE analytically

$$\mathcal{L} = (f(x, W, b) - y)^2$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_0} &= \frac{\partial}{\partial w_0}(f(x, W, b) - y)^2 \\
&= 2 \cdot (f(x, W, b) - y) \cdot \frac{\partial}{\partial w_0}(f(x, W, b) - y) \\
&= 2 \cdot (f(x, W, b) - y) \cdot \frac{\partial}{\partial w_0}(w_0 \cdot x_0 + w_1 \cdot x_1 + b - y) \\
&= 2 \cdot (f(x, W, b) - y) \cdot (x_0 + 0 + 0 - 0)
\end{aligned}$$

# Optimization

Let us compute the gradient of MSE analytically

$$\mathcal{L} = (f(x, W, b) - y)^2$$

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_0} &= \frac{\partial}{\partial w_0}(f(x, W, b) - y)^2 \\ &= 2 \cdot (f(x, W, b) - y) \cdot \frac{\partial}{\partial w_0}(f(x, W, b) - y) \\ &= 2 \cdot (f(x, W, b) - y) \cdot \frac{\partial}{\partial w_0}(w_0 \cdot x_0 + w_1 \cdot x_1 + b - y) \\ &= 2 \cdot (f(x, W, b) - y) \cdot (x_0 + 0 + 0 - 0) \\ &= 2 \cdot x_0 \cdot (f(x, W, b) - y) \end{aligned}$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = 2 \cdot x_1 \cdot (f(x, W, b) - y)$$

$$\frac{\partial \mathcal{L}}{\partial b} = 2 \cdot (f(x, W, b) - y)$$

# Optimization

But what if the function was slightly more complicated:

$$f(x, w) = \left(\frac{e^{x \cdot w}}{x}\right)^3$$

$$\begin{aligned}
\frac{\partial}{\partial w}\left(\frac{e^{xw}}{xw}\right)^3 &= 3\left(\frac{e^{xw}}{xw}\right)^2 \cdot \frac{\partial}{\partial w}\left(\frac{e^{xw}}{xw}\right) \\
&= 3\left(\frac{e^{xw}}{xw}\right)^2 \cdot \frac{\frac{\partial}{\partial w}e^{xw} \cdot xw - e^{xw} \cdot \frac{\partial}{\partial w}xw}{x^2w^2} \\
&= 3\left(\frac{e^{xw}}{xw}\right)^2 \cdot \frac{e^{xw}x \cdot xw - e^{xw} \cdot x}{x^2w^2} \\
&= 3\left(\frac{e^{2xw}}{x^2w^2}\right) \cdot \frac{e^{xw} \cdot x^2w - e^{xw} \cdot x}{x^2w^2} \\
&= 3\left(\frac{e^{2xw}}{x^2w^2}\right) \cdot \frac{e^{xw} \cdot xw - e^{xw}}{xw^2} \\
&= 3\frac{e^{3xw} \cdot (xw - 1)}{x^3w^4}
\end{aligned}$$

# Optimization

But what if the function was slightly more complicated:

$$f(x, w) = \left(\frac{e^{x \cdot w}}{x}\right)^3$$

Analytical gradients become much more complicated and tedious to compute!

$$= 3\left(\frac{e^{2xw}}{x^2 w^2}\right) \cdot \frac{e^{xw} \cdot xw - e^{xw}}{xw^2}$$

$$= 3\frac{e^{3xw} \cdot (xw - 1)}{x^3 w^4}$$

# Optimization

But what if the function was slightly more complicated:

$$f(x, w) = \left(\frac{e^{x \cdot w}}{x}\right)^3$$

Backpropagation to the rescue!

$$= 3\left(\frac{e^{2xw}}{x^2w^2}\right) \cdot \frac{e^{xw} \cdot xw - e^{xw}}{xw^2}$$

$$= 3\frac{e^{3xw} \cdot (xw - 1)}{x^3w^4}$$

# Backpropagation

Backpropagation is a technique to compute gradients of any function with respect to a variable using the concept of a *computation graph*

# Backpropagation

Computation graph: Graphical way of describing any function:

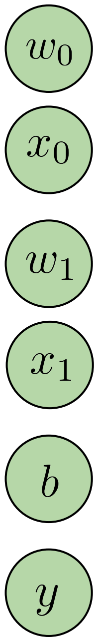
$$\mathcal{L} = (f(x, W, b) - y)^2$$

# Backpropagation

$$f(x, W, b) = w_0 \cdot x_0 + w_1 \cdot x_1 + b$$

Computation graph: Graphical way of describing any function:

$$\mathcal{L} = (f(x, W, b) - y)^2$$



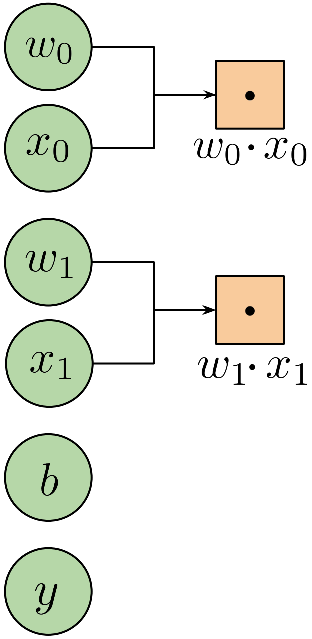
Each node in the graph is either an input, an operation or an output

# Backpropagation

$f(x, W, b) = w_0 \cdot x_0 + w_1 \cdot x_1 + b$

Computation graph: Graphical way of describing any function:

$$\mathcal{L} = (f(x, W, b) - y)^2$$



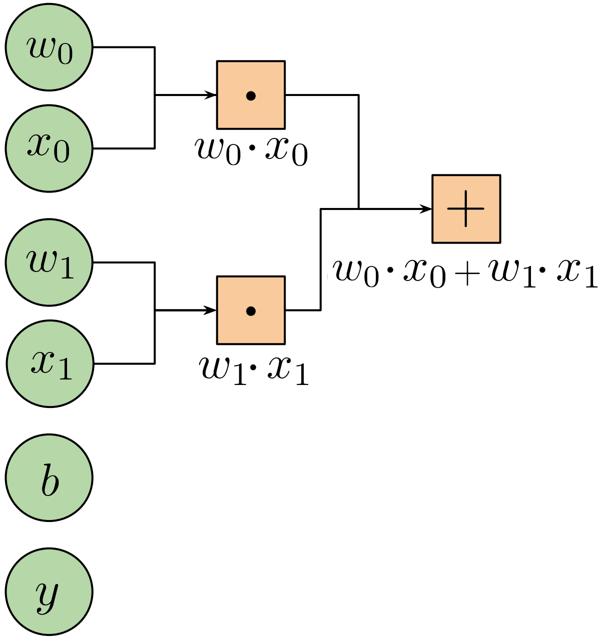
Each node in the graph is either an input, an operation or an output

# Backpropagation

$f(x, W, b) = w_0 \cdot x_0 + w_1 \cdot x_1 + b$

Computation graph: Graphical way of describing any function:

$$\mathcal{L} = (f(x, W, b) - y)^2$$

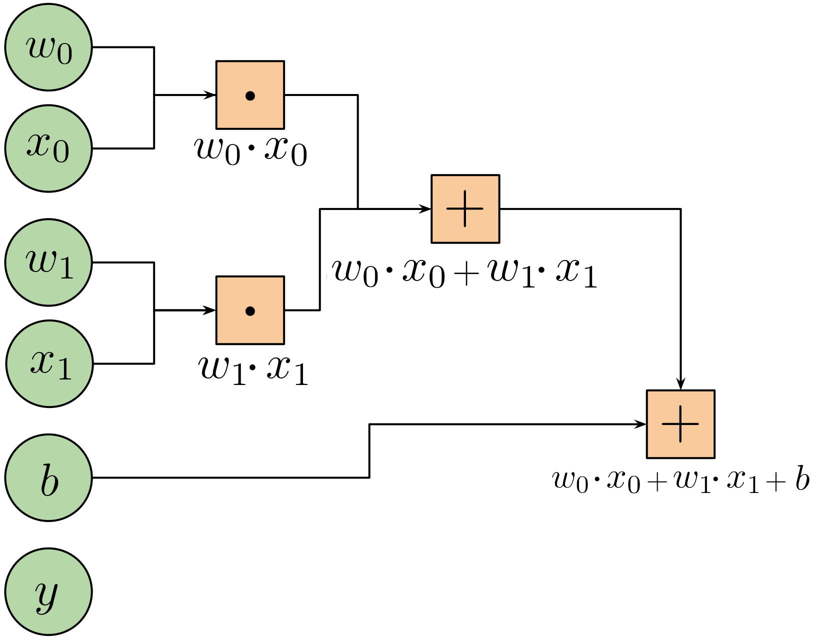


Each node in the graph is either an input, an operation or an output

# Backpropagation

Computation graph: Graphical way of describing any function:

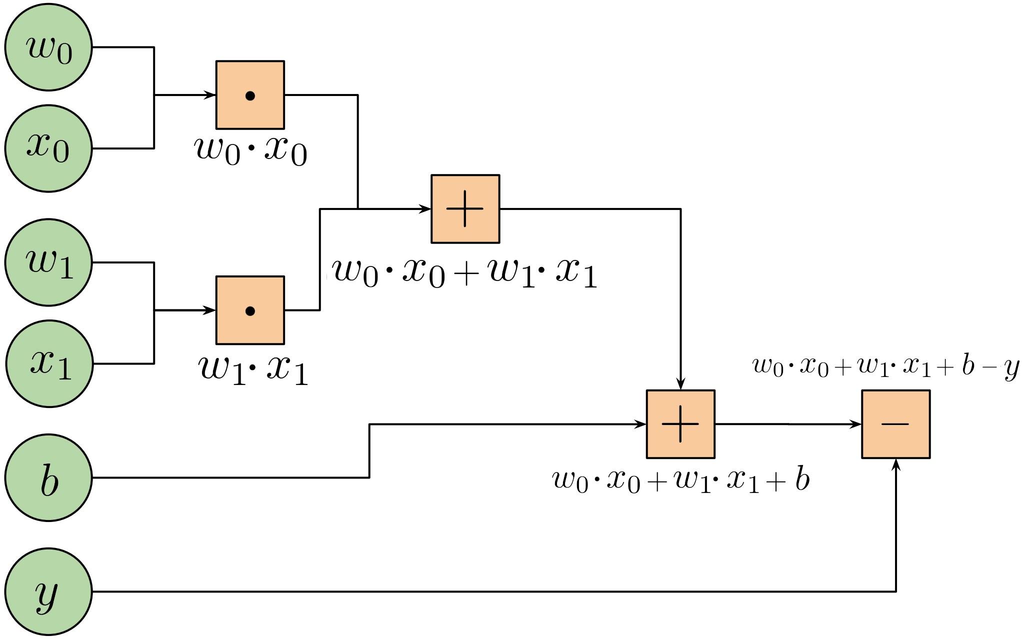
$$\mathcal{L} = (f(x, W, b) - y)^2$$



# Backpropagation

Computation graph: Graphical way of describing any function:

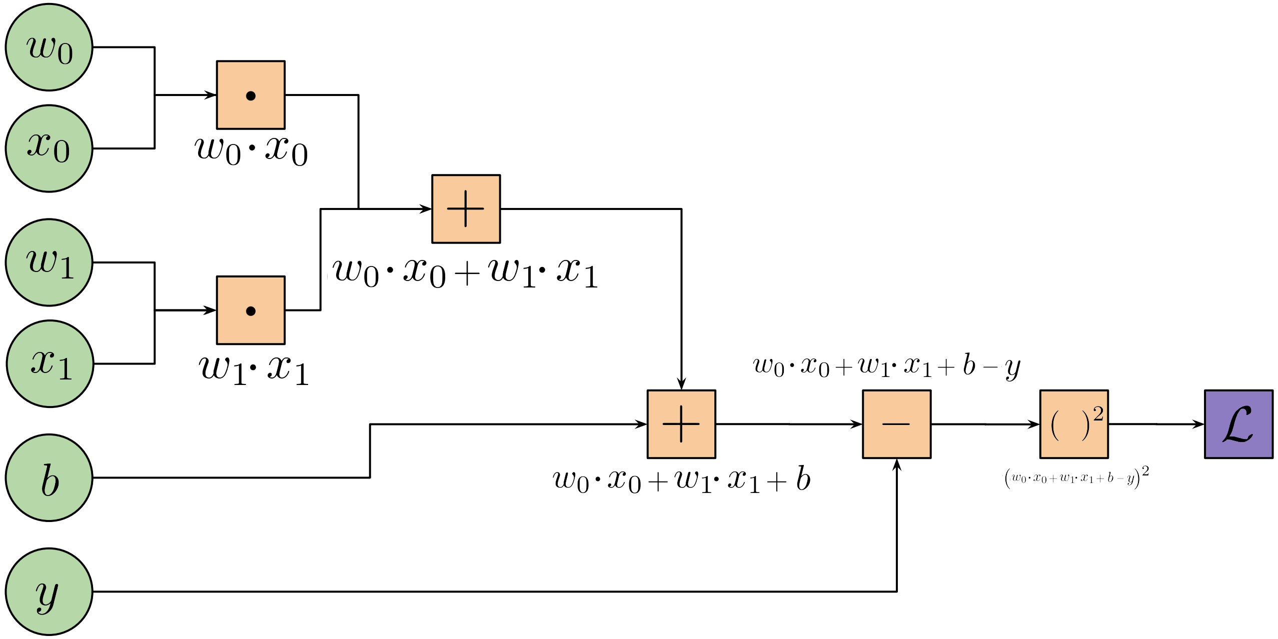
$$\mathcal{L} = (f(x, W, b) - y)^2$$



# Backpropagation

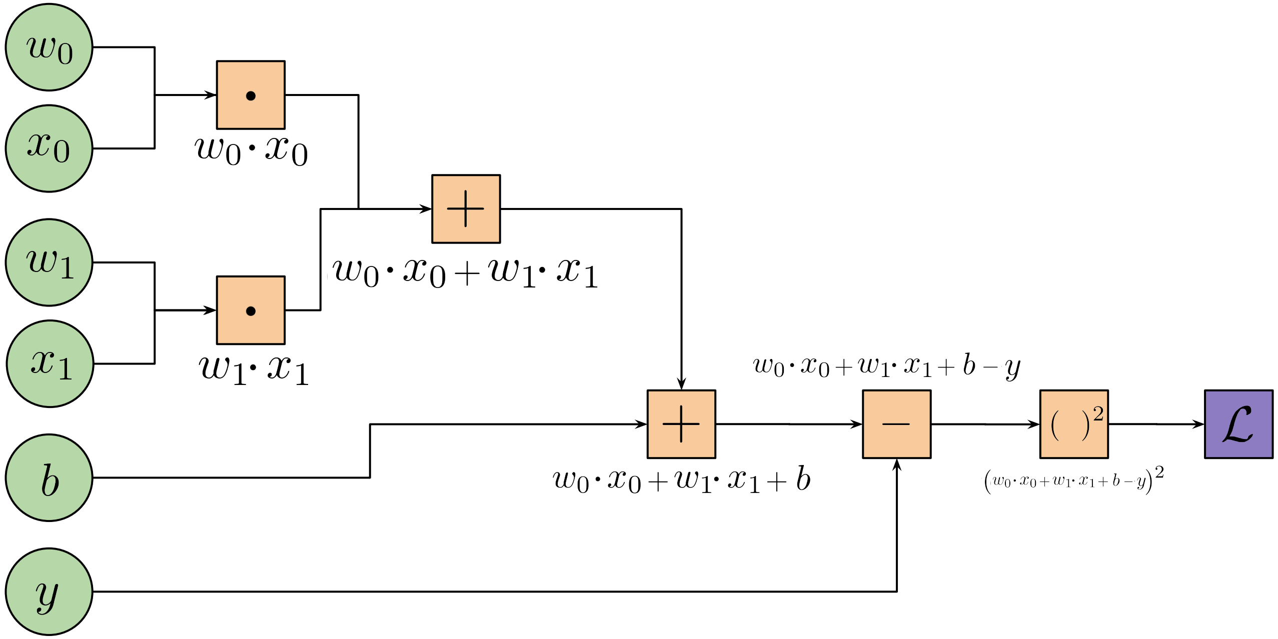
Computation graph: Graphical way of describing any function:

$$\mathcal{L} = (f(x, W, b) - y)^2$$



# Backpropagation

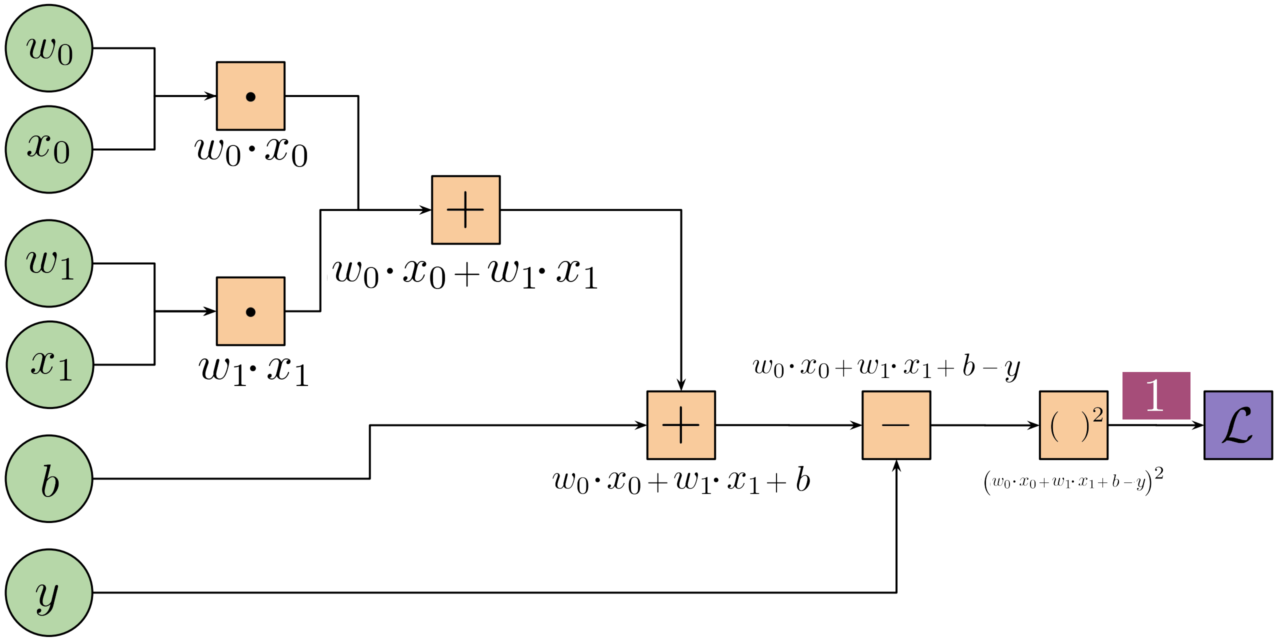
$$\mathcal{L} = (f(x, W, b) - y)^2$$



We can now use the chain rule to compute the gradient, and work our way backwards from the output to the inputs!

# Backpropagation

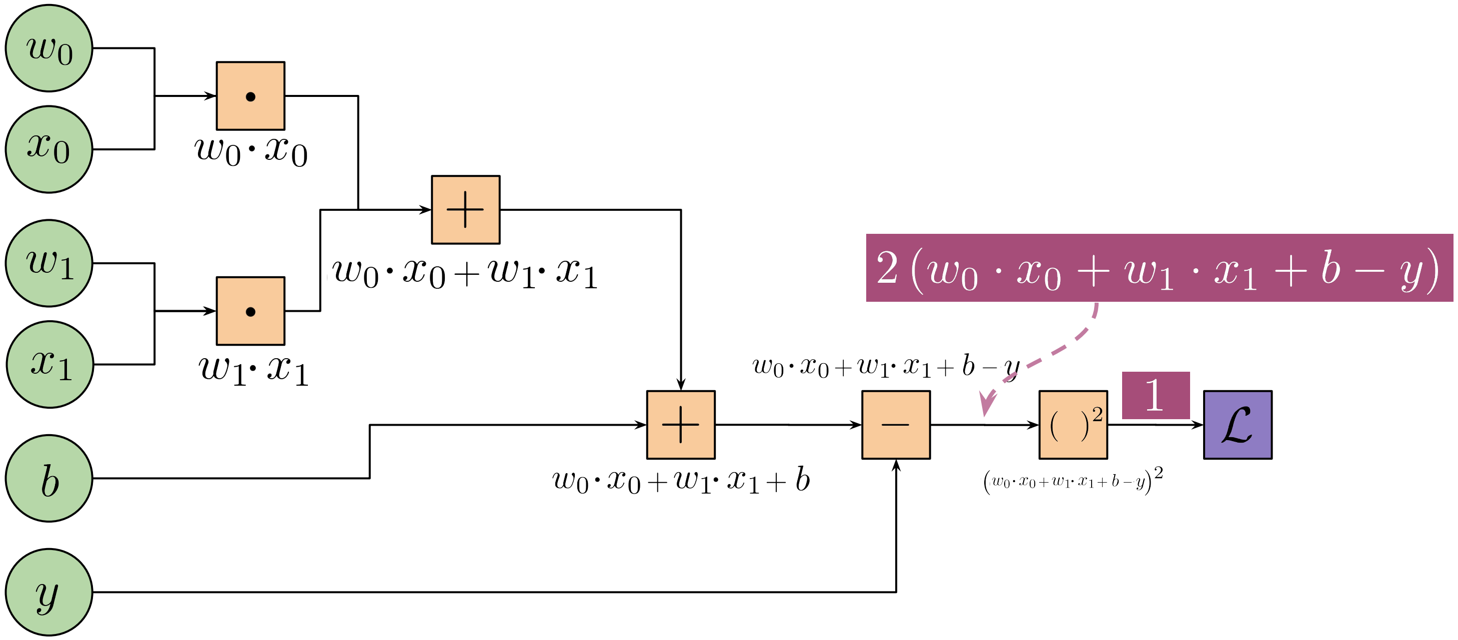
$$\mathcal{L} = (f(x, W, b) - y)^2$$



We can now use the chain rule to compute the gradient, and work our way backwards from the output to the inputs!

# Backpropagation

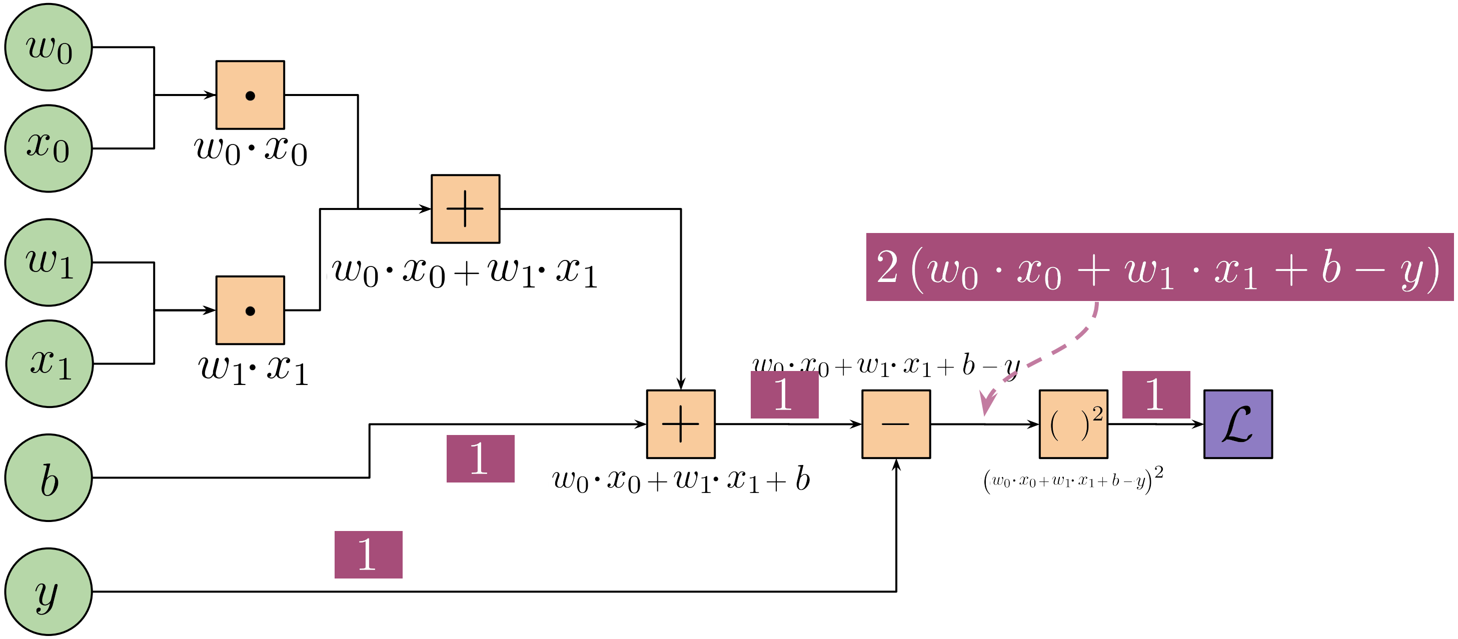
$$\mathcal{L} = (f(x, W, b) - y)^2$$



We can now use the chain rule to compute the gradient, and work our way backwards from the output to the inputs!

# Backpropagation

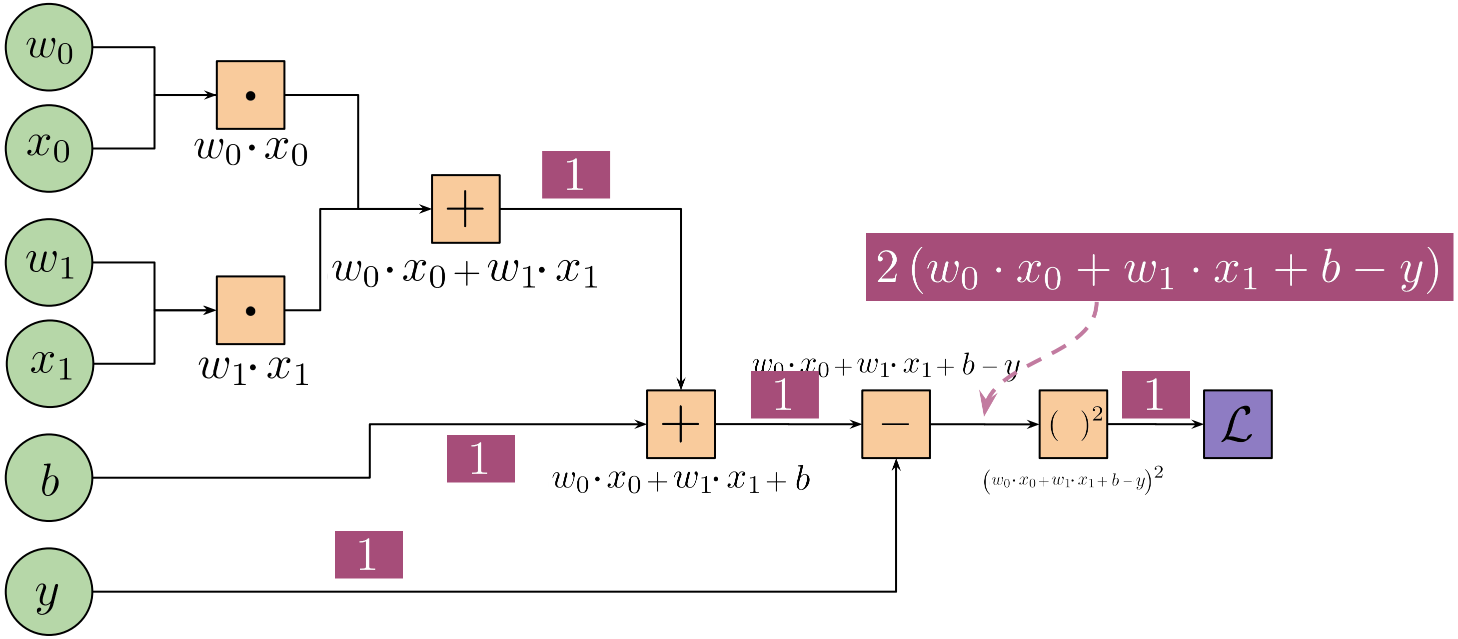
$$\mathcal{L} = (f(x, W, b) - y)^2$$



We can now use the chain rule to compute the gradient, and work our way backwards from the output to the inputs!

# Backpropagation

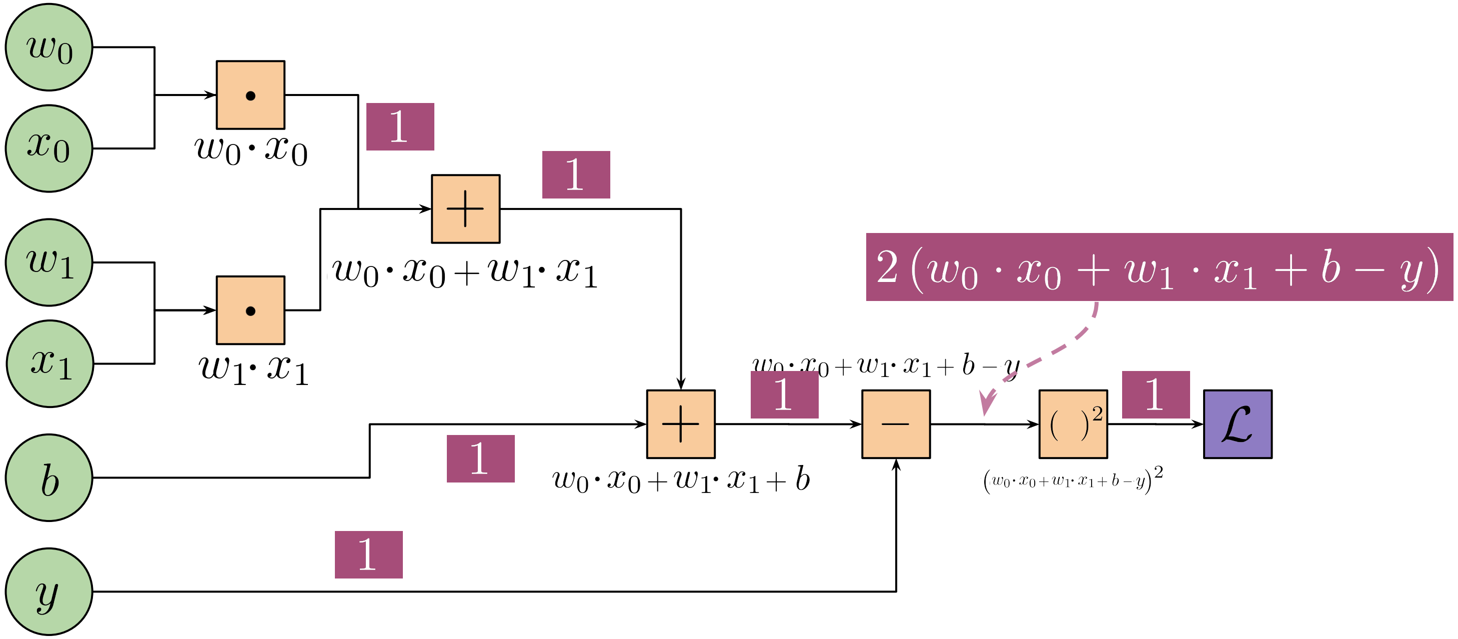
$$\mathcal{L} = (f(x, W, b) - y)^2$$



We can now use the chain rule to compute the gradient, and work our way backwards from the output to the inputs!

# Backpropagation

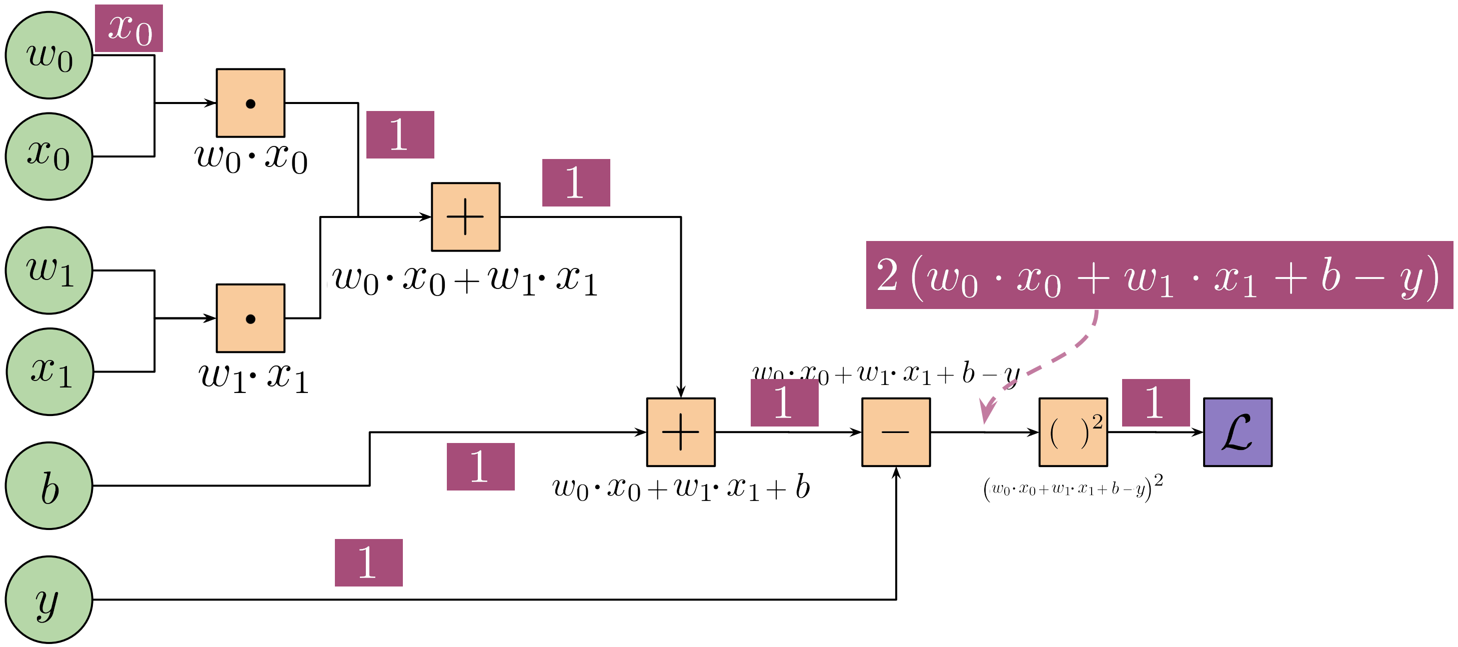
$$\mathcal{L} = (f(x, W, b) - y)^2$$



We can now use the chain rule to compute the gradient, and work our way backwards from the output to the inputs!

# Backpropagation

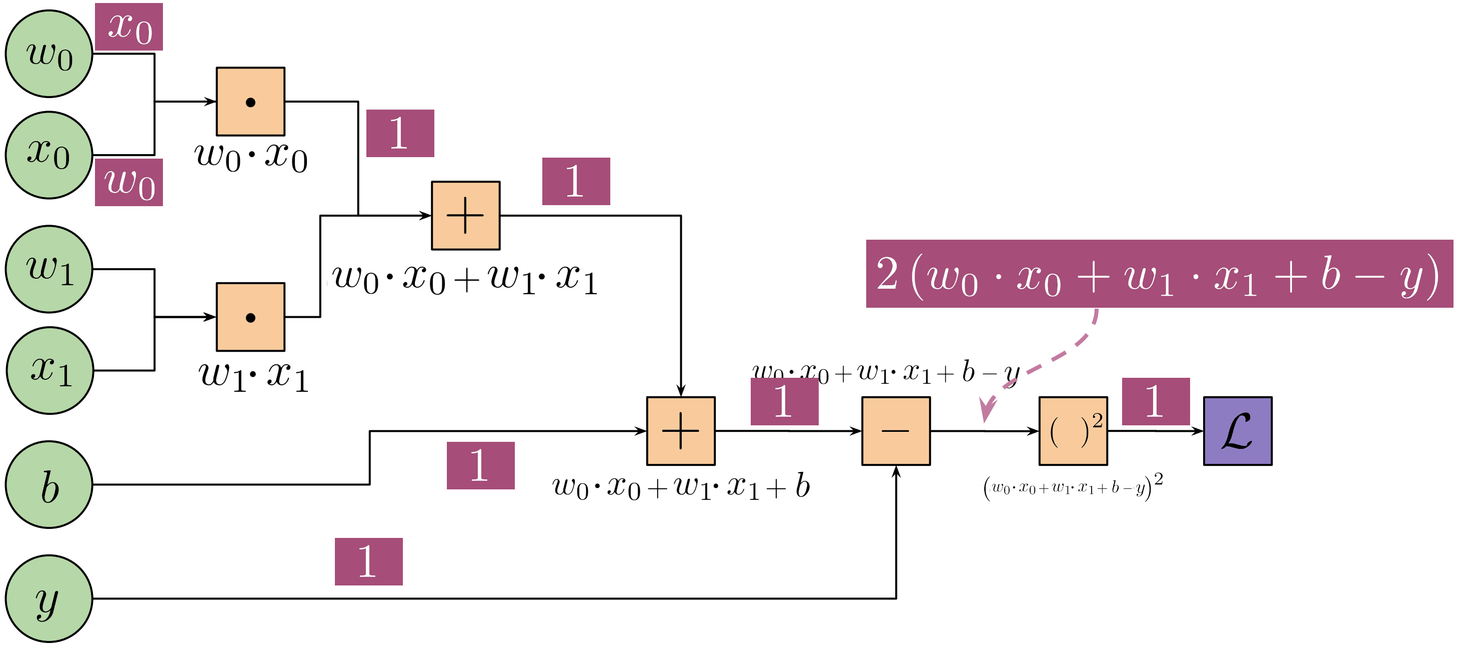
$$\mathcal{L} = (f(x, W, b) - y)^2$$



We can now use the chain rule to compute the gradient, and work our way backwards from the output to the inputs!

# Backpropagation

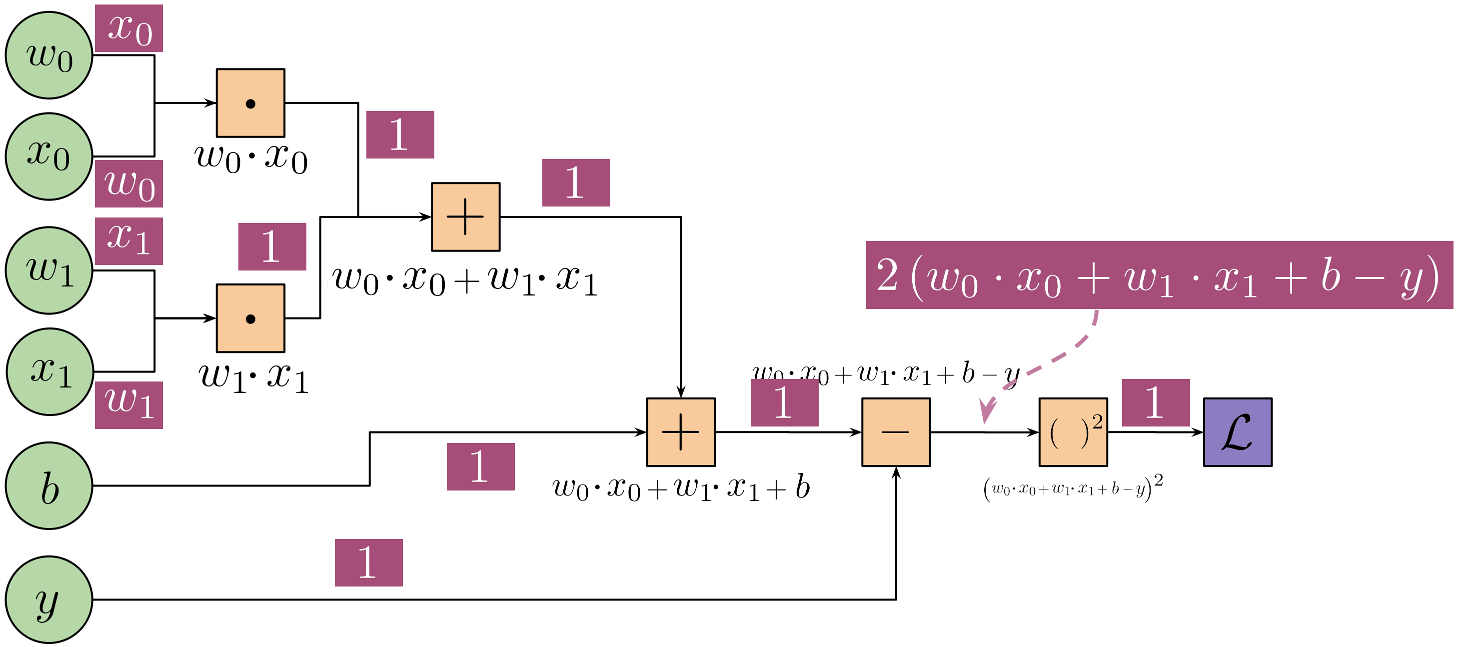
$$\mathcal{L} = (f(x, W, b) - y)^2$$



We can now use the chain rule to compute the gradient, and work our way backwards from the output to the inputs!

# Backpropagation

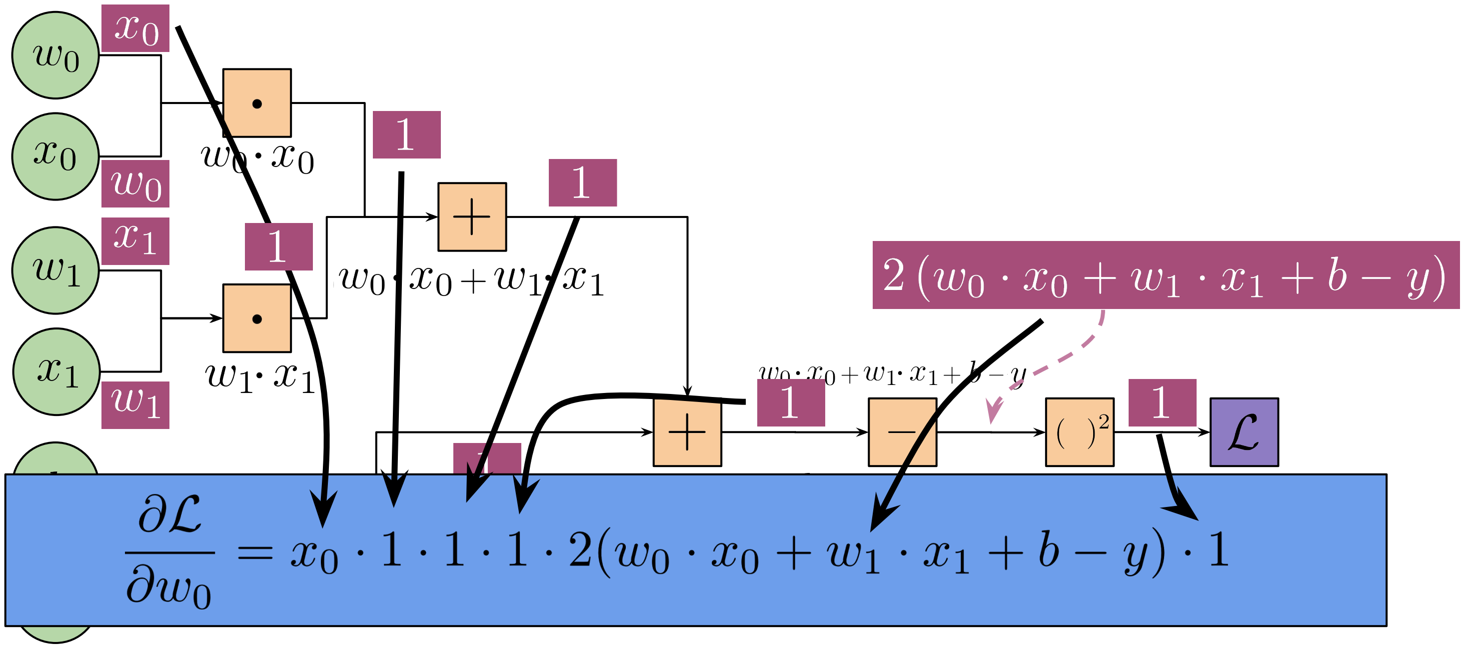
$$\mathcal{L} = (f(x, W, b) - y)^2$$



We can now use the chain rule to compute the gradient, and work our way backwards from the output to the inputs!

# Backpropagation

$$\mathcal{L} = (f(x, W, b) - y)^2$$



$$\frac{\partial \mathcal{L}}{\partial w_0} = x_0 \cdot 1 \cdot 1 \cdot 1 \cdot 2(w_0 \cdot x_0 + w_1 \cdot x_1 + b - y) \cdot 1$$

We can now use the chain rule to compute the gradient, and work our way backwards from the output to the inputs!

# Optimization

To complete the picture, we can then use the gradients to update the parameters using gradient descent

# Optimization

To complete the picture, we can then use the gradients to update the parameters using gradient descent

Recall: We want to take a “step” in the direction of the slope

# Optimization

To complete the picture, we can then use the gradients to update the parameters using gradient descent

$$w_0 = w_0 - \eta \cdot \frac{\partial \mathcal{L}}{\partial w_0}$$

$$w_1 = w_1 - \eta \cdot \frac{\partial \mathcal{L}}{\partial w_1}$$

$$b = b - \eta \cdot \frac{\partial \mathcal{L}}{\partial b}$$

# Optimization

To complete the picture, we can then use the gradients to update the parameters using gradient descent

$$w_0 = w_0 - \boxed{\eta} \cdot \frac{\partial \mathcal{L}}{\partial w_0}$$

$$w_1 = w_1 - \boxed{\eta} \cdot \frac{\partial \mathcal{L}}{\partial w_1}$$

Step size

Learning rate

$$b = b - \boxed{\eta} \cdot \frac{\partial \mathcal{L}}{\partial b}$$

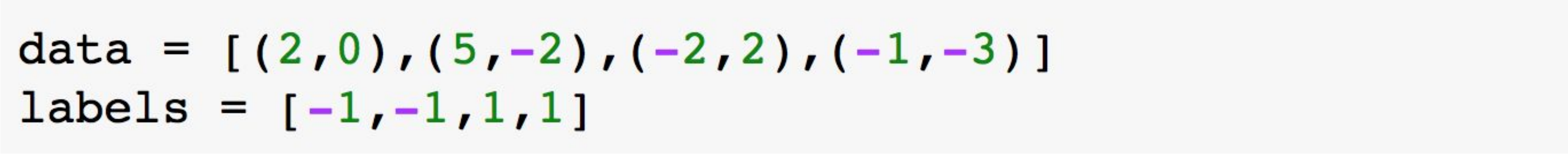
# Optimization Exercise

Let’s see a linear classifier in code!

1. Data setup
2. Defining objective and loss functions
3. Implementing gradient functions
4. Optimization
5. Bonus: Plotting results

# Optimization Exercise

## Data setup

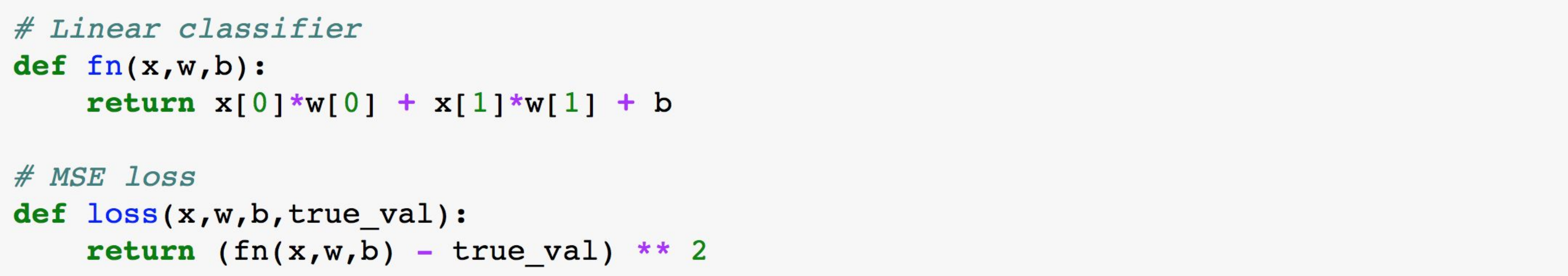


```
data = [(2,0),(5,-2),(-2,2),(-1,-3)]
labels = [-1,-1,1,1]
```

- Usually data is loaded from an external source
- Eventually, all data is represented in some structured form like in matrices
- Data for supervised learning is normally composed of the actual data points and the labels for each point

# Optimization Exercise

## Defining objective and loss functions



```
# Linear classifier
def fn(x,w,b):
    return x[0]*w[0] + x[1]*w[1] + b

# MSE loss
def loss(x,w,b,true_val):
    return (fn(x,w,b) - true_val) ** 2
```

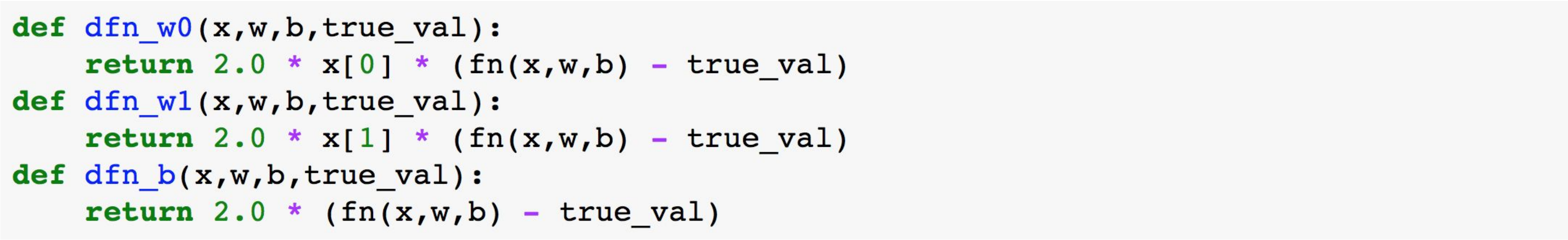
- In this case, `fn` is the objective function for a linear classifier
- `loss` computes *mean squared error*

$$f(x, W, b) = w_0 \cdot x_0 + w_1 \cdot x_1 + b$$

$$MSE(x, W, b, y) = (f(x, W, b) - y)^2$$

# Optimization Exercise

## Implementing gradient functions

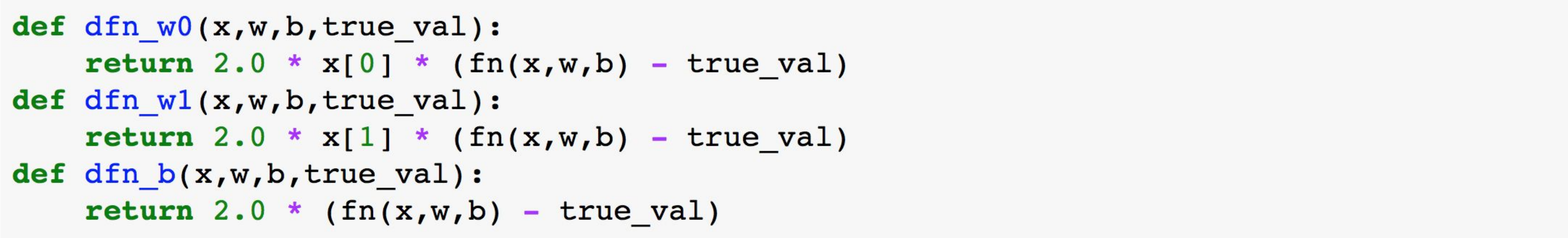


```python
def dfn_w0(x,w,b,true_val):
    return 2.0 * x[0] * (fn(x,w,b) - true_val)
def dfn_w1(x,w,b,true_val):
    return 2.0 * x[1] * (fn(x,w,b) - true_val)
def dfn_b(x,w,b,true_val):
    return 2.0 * (fn(x,w,b) - true_val)
```

- We care about the gradient of the loss function with respect to each of our trainable parameters, i.e $w_0$, $w_1$ and $b$
- Gradients can be computed analytically or using the computation graph as we've seen before - only the final form is important

# Optimization Exercise

## Implementing gradient functions



```
def dfn_w0(x,w,b,true_val):
    return 2.0 * x[0] * (fn(x,w,b) - true_val)
def dfn_w1(x,w,b,true_val):
    return 2.0 * x[1] * (fn(x,w,b) - true_val)
def dfn_b(x,w,b,true_val):
    return 2.0 * (fn(x,w,b) - true_val)
```

- We care about the gradient of the loss function with respe $w_0$, $w_1$ and
- Gradi the comp he final form

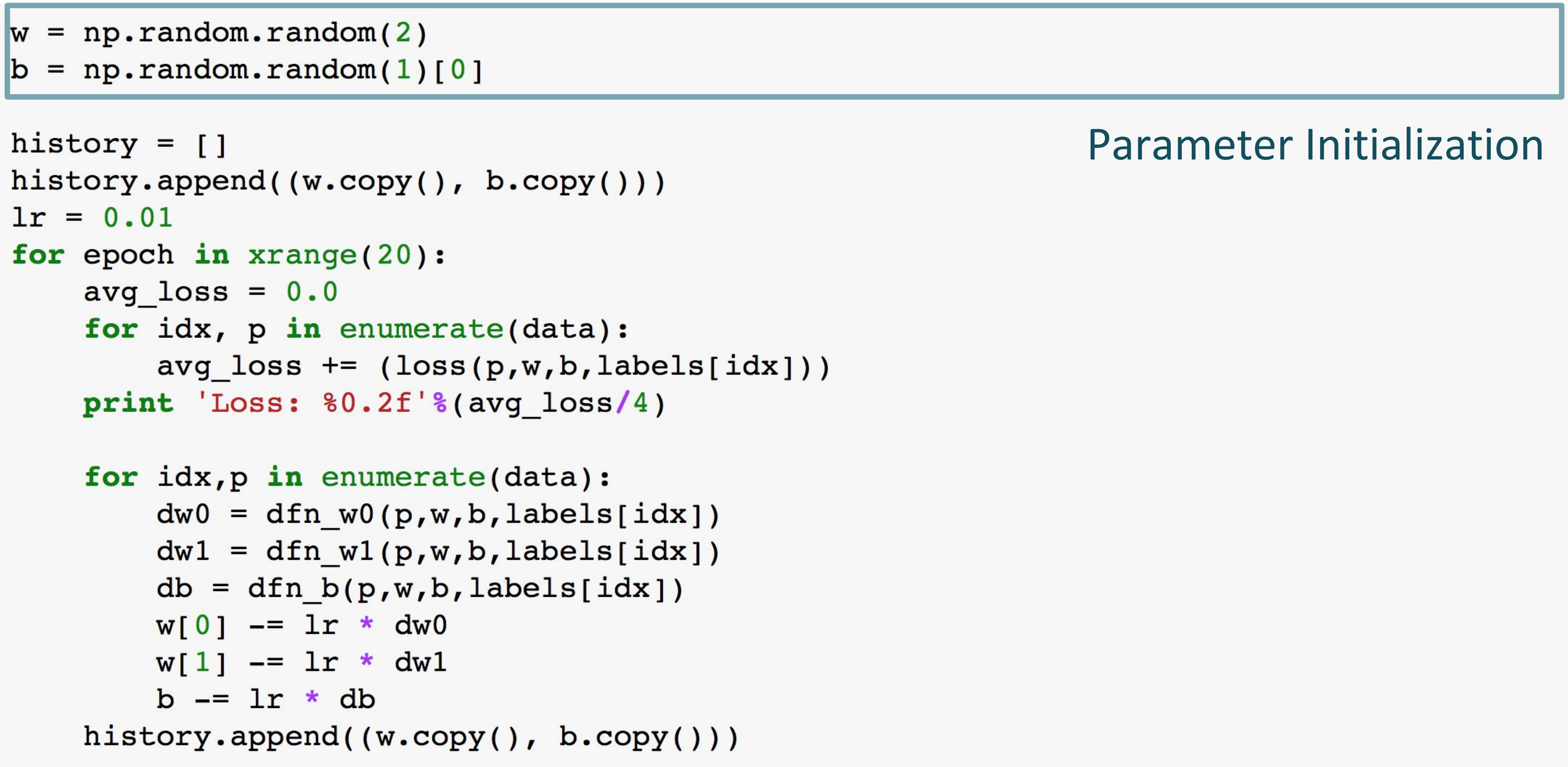
$$\frac{\partial \mathcal{L}}{\partial w_0} = 2 \cdot x_0 \cdot (f(x, W, b) - y)$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = 2 \cdot x_1 \cdot (f(x, W, b) - y)$$

$$\frac{\partial \mathcal{L}}{\partial b} = 2 \cdot (f(x, W, b) - y)$$

# Optimization Exercise

## Optimization

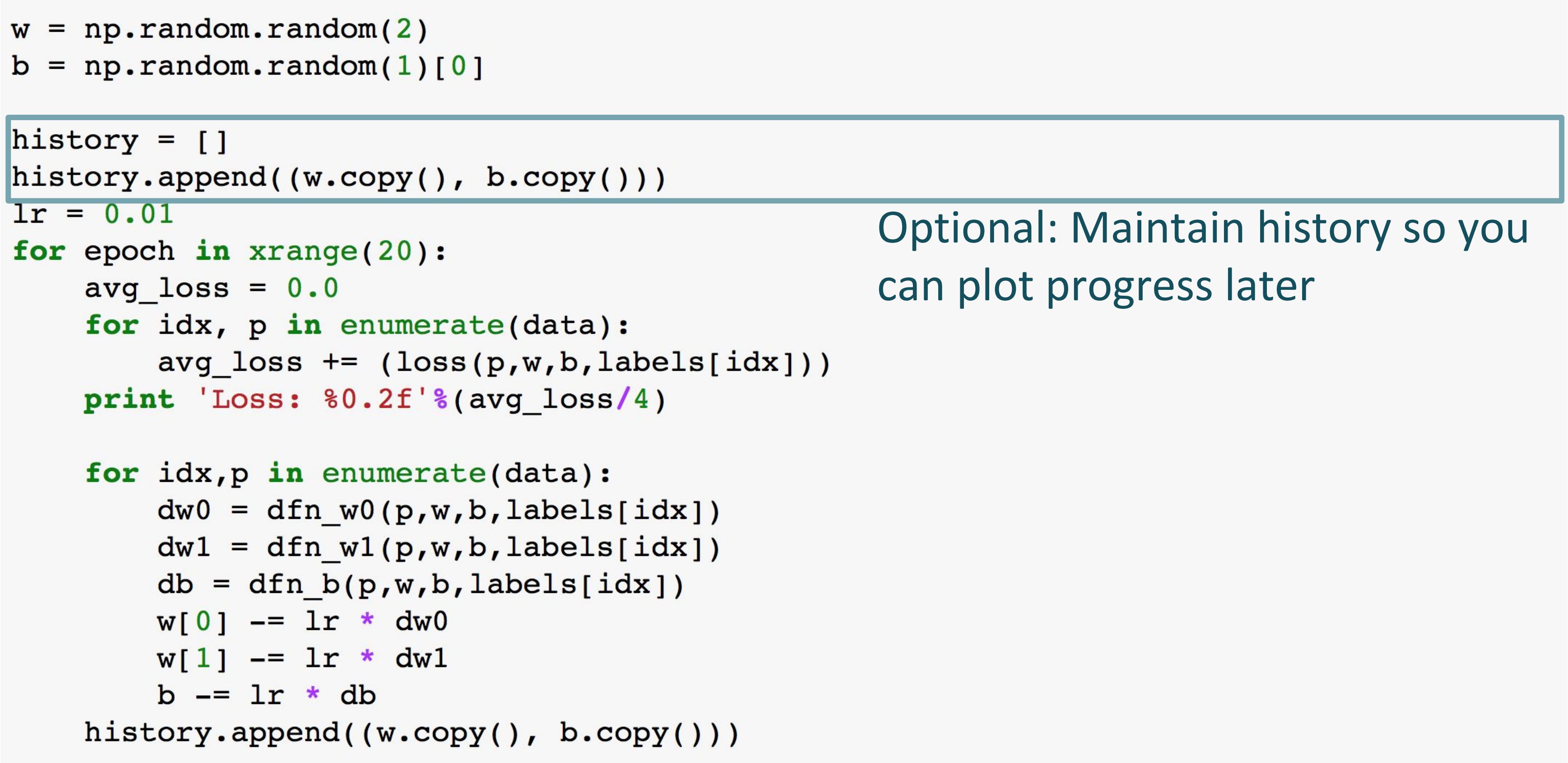


```
w = np.random.random(2)
b = np.random.random(1)[0]

history = []
history.append((w.copy(), b.copy()))
lr = 0.01
for epoch in xrange(20):
    avg_loss = 0.0
    for idx, p in enumerate(data):
        avg_loss += (loss(p,w,b,labels[idx]))
    print 'Loss: %0.2f'%(avg_loss/4)

    for idx,p in enumerate(data):
        dw0 = dfn_w0(p,w,b,labels[idx])
        dw1 = dfn_w1(p,w,b,labels[idx])
        db = dfn_b(p,w,b,labels[idx])
        w[0] -= lr * dw0
        w[1] -= lr * dw1
        b -= lr * db
    history.append((w.copy(), b.copy()))
```

Parameter Initialization

# Optimization Exercise

## Optimization

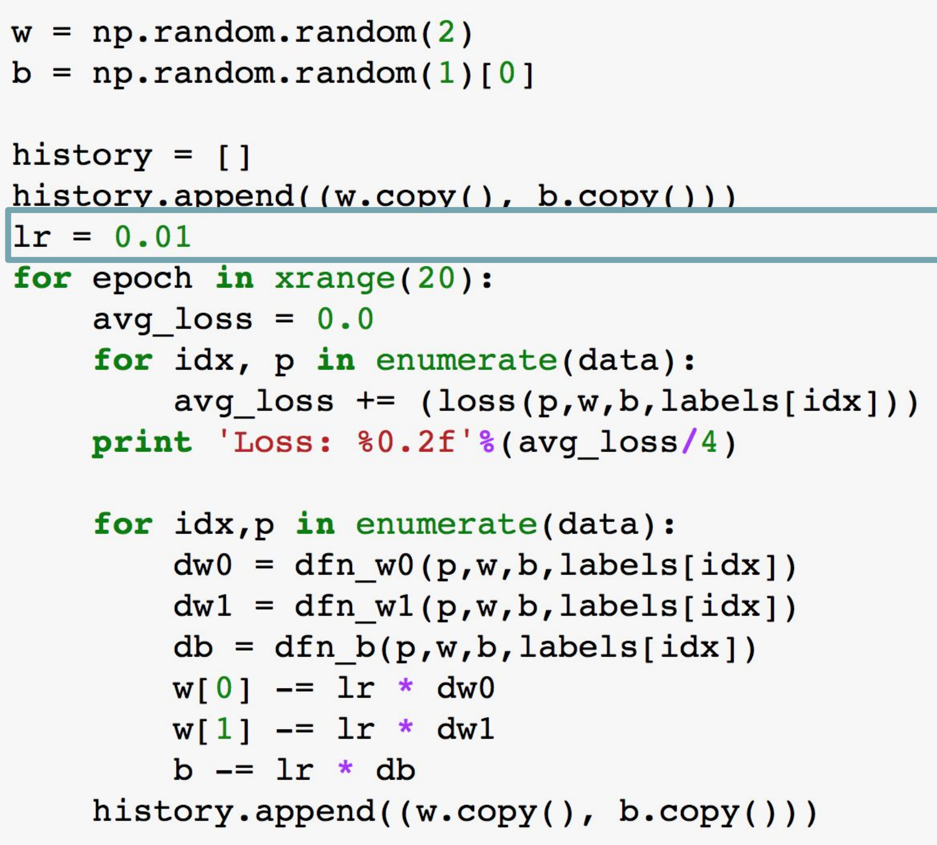


```
w = np.random.random(2)
b = np.random.random(1)[0]

history = []
history.append((w.copy(), b.copy()))
lr = 0.01
for epoch in xrange(20):
    avg_loss = 0.0
    for idx, p in enumerate(data):
        avg_loss += (loss(p,w,b,labels[idx]))
    print 'Loss: %0.2f'%(avg_loss/4)

    for idx,p in enumerate(data):
        dw0 = dfn_w0(p,w,b,labels[idx])
        dw1 = dfn_w1(p,w,b,labels[idx])
        db = dfn_b(p,w,b,labels[idx])
        w[0] -= lr * dw0
        w[1] -= lr * dw1
        b -= lr * db
    history.append((w.copy(), b.copy()))
```

Optional: Maintain history so you can plot progress later

# Optimization Exercise

## Optimization

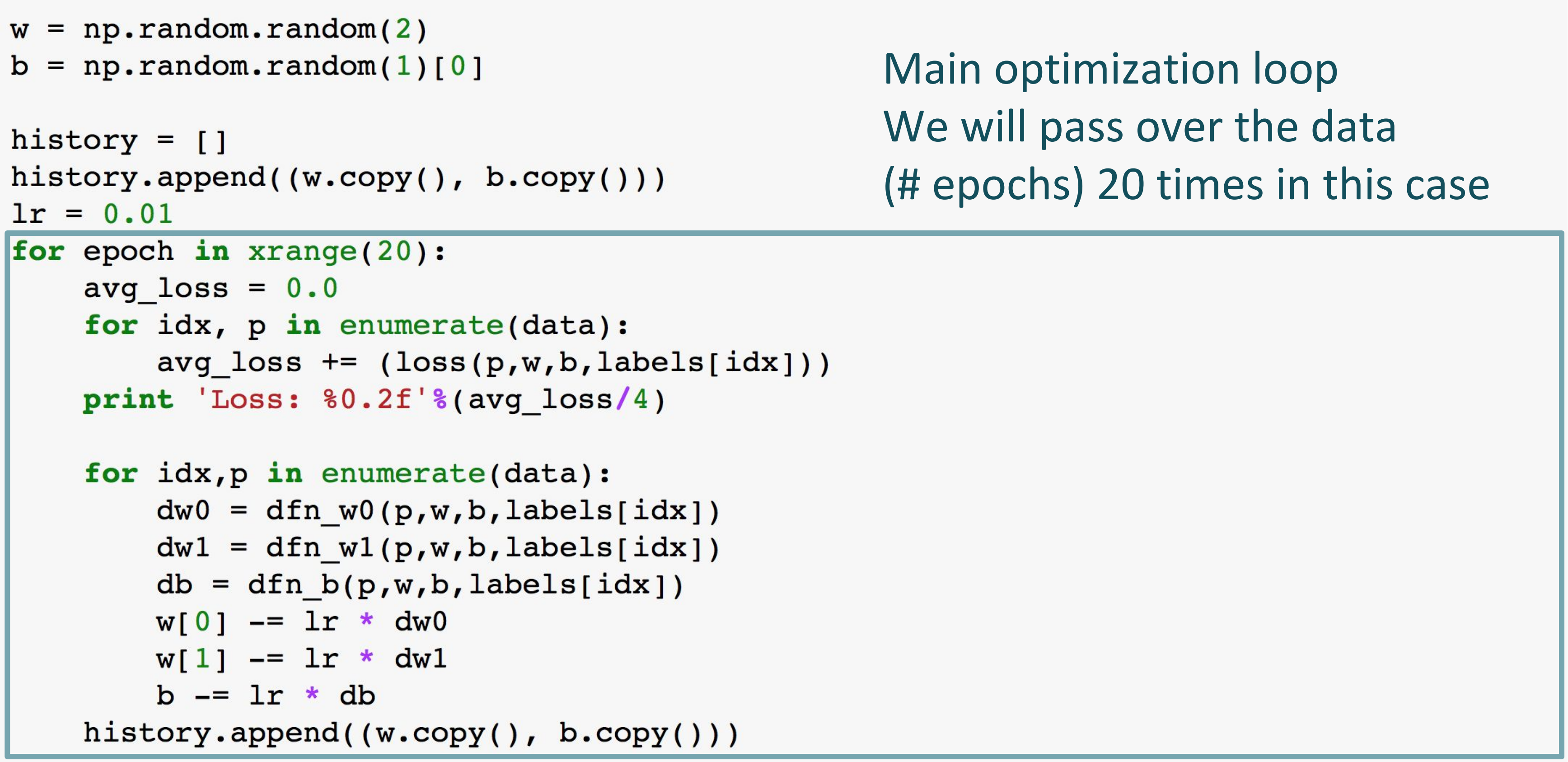


```
w = np.random.random(2)
b = np.random.random(1)[0]

history = []
history.append((w.copy(), b.copy()))
lr = 0.01
for epoch in xrange(20):
    avg_loss = 0.0
    for idx, p in enumerate(data):
        avg_loss += (loss(p,w,b,labels[idx]))
    print 'Loss: %0.2f'%(avg_loss/4)

    for idx,p in enumerate(data):
        dw0 = dfn_w0(p,w,b,labels[idx])
        dw1 = dfn_w1(p,w,b,labels[idx])
        db = dfn_b(p,w,b,labels[idx])
        w[0] -= lr * dw0
        w[1] -= lr * dw1
        b -= lr * db
    history.append((w.copy(), b.copy()))
```

Parameter initialization for optimization algorithm.
Here we are going to use SGD, so there is only one parameter to set: learning rate

# Optimization Exercise

## Optimization

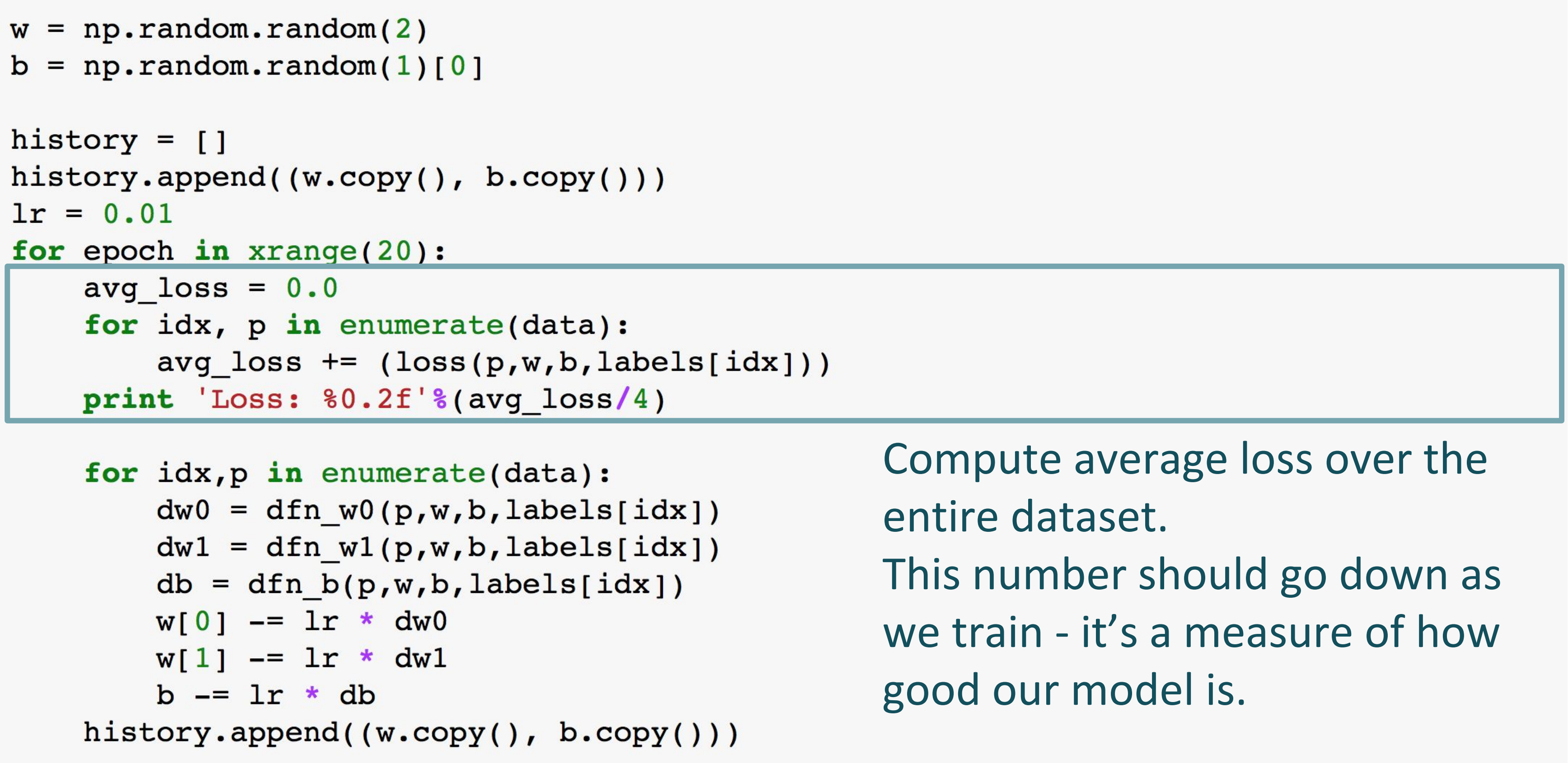


```
w = np.random.random(2)
b = np.random.random(1)[0]

history = []
history.append((w.copy(), b.copy()))
lr = 0.01
for epoch in xrange(20):
    avg_loss = 0.0
    for idx, p in enumerate(data):
        avg_loss += (loss(p,w,b,labels[idx]))
    print 'Loss: %0.2f'%(avg_loss/4)

    for idx,p in enumerate(data):
        dw0 = dfn_w0(p,w,b,labels[idx])
        dw1 = dfn_w1(p,w,b,labels[idx])
        db = dfn_b(p,w,b,labels[idx])
        w[0] -= lr * dw0
        w[1] -= lr * dw1
        b -= lr * db
    history.append((w.copy(), b.copy()))
```

Main optimization loop
We will pass over the data
(# epochs) 20 times in this case

# Optimization Exercise

## Optimization

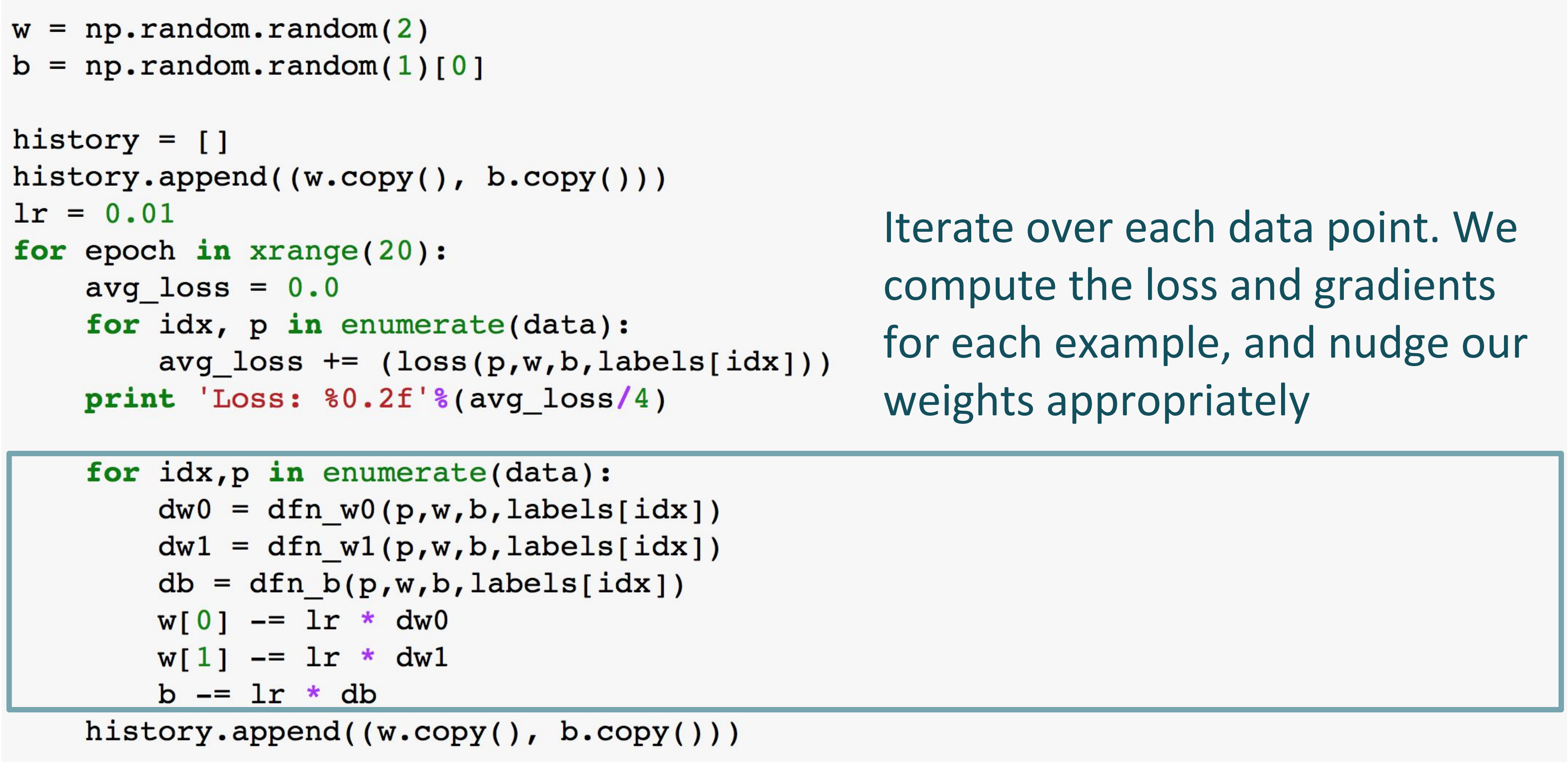


```
w = np.random.random(2)
b = np.random.random(1)[0]

history = []
history.append((w.copy(), b.copy()))
lr = 0.01
for epoch in xrange(20):
    avg_loss = 0.0
    for idx, p in enumerate(data):
        avg_loss += (loss(p,w,b,labels[idx]))
    print 'Loss: %0.2f'%(avg_loss/4)

    for idx,p in enumerate(data):
        dw0 = dfn_w0(p,w,b,labels[idx])
        dw1 = dfn_w1(p,w,b,labels[idx])
        db = dfn_b(p,w,b,labels[idx])
        w[0] -= lr * dw0
        w[1] -= lr * dw1
        b -= lr * db
    history.append((w.copy(), b.copy()))
```

Compute average loss over the entire dataset.
This number should go down as we train - it’s a measure of how good our model is.

# Optimization Exercise

## Optimization

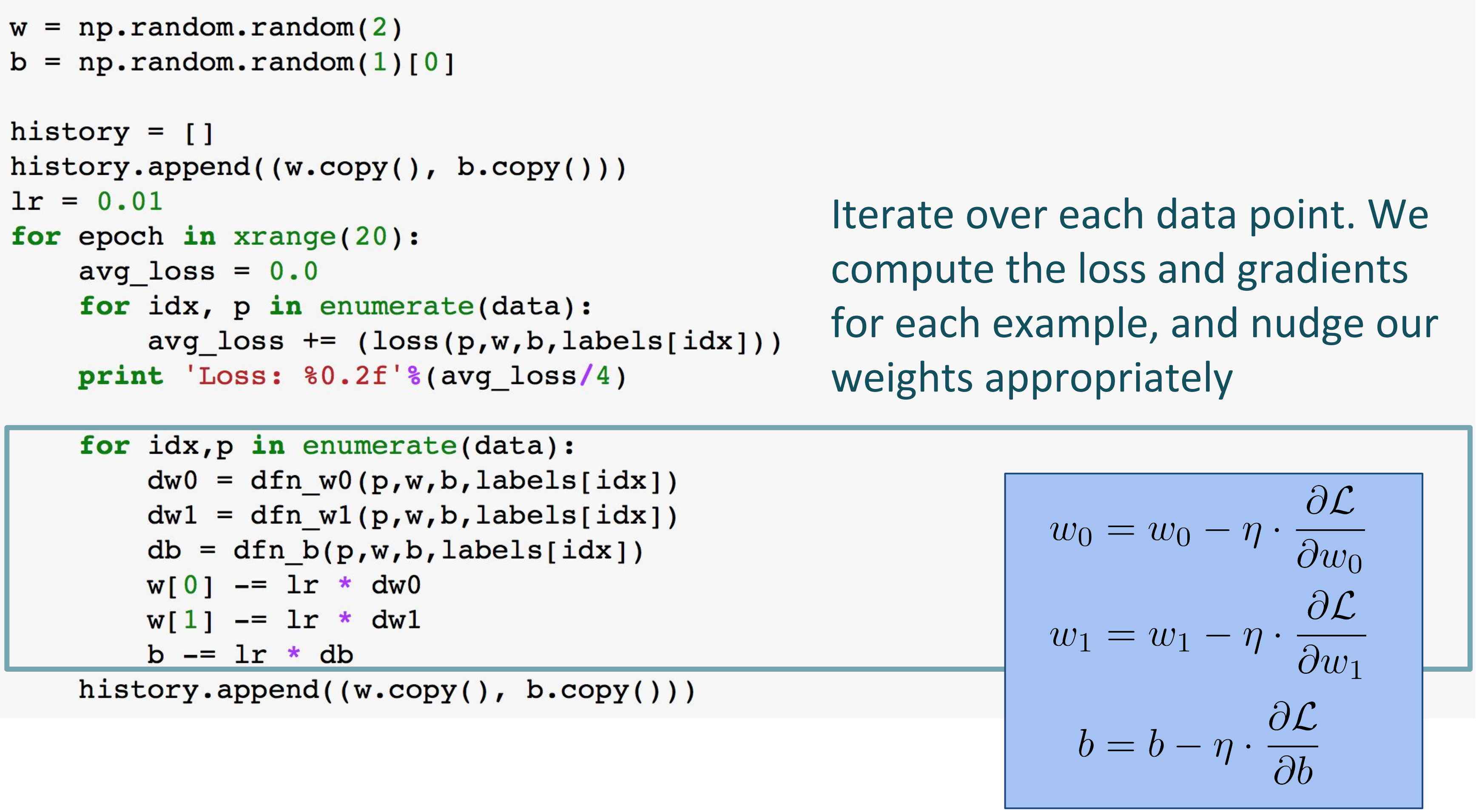


```
w = np.random.random(2)
b = np.random.random(1)[0]

history = []
history.append((w.copy(), b.copy()))
lr = 0.01
for epoch in xrange(20):
    avg_loss = 0.0
    for idx, p in enumerate(data):
        avg_loss += (loss(p,w,b,labels[idx]))
    print 'Loss: %0.2f'%(avg_loss/4)

    for idx,p in enumerate(data):
        dw0 = dfn_w0(p,w,b,labels[idx])
        dw1 = dfn_w1(p,w,b,labels[idx])
        db = dfn_b(p,w,b,labels[idx])
        w[0] -= lr * dw0
        w[1] -= lr * dw1
        b -= lr * db
    history.append((w.copy(), b.copy()))
```

Iterate over each data point. We compute the loss and gradients for each example, and nudge our weights appropriately

# Optimization Exercise

## Optimization



```
w = np.random.random(2)
b = np.random.random(1)[0]

history = []
history.append((w.copy(), b.copy()))
lr = 0.01
for epoch in xrange(20):
    avg_loss = 0.0
    for idx, p in enumerate(data):
        avg_loss += (loss(p,w,b,labels[idx]))
    print 'Loss: %0.2f'%(avg_loss/4)

    for idx,p in enumerate(data):
        dw0 = dfn_w0(p,w,b,labels[idx])
        dw1 = dfn_w1(p,w,b,labels[idx])
        db = dfn_b(p,w,b,labels[idx])
        w[0] -= lr * dw0
        w[1] -= lr * dw1
        b -= lr * db
    history.append((w.copy(), b.copy()))
```

Iterate over each data point. We compute the loss and gradients for each example, and nudge our weights appropriately

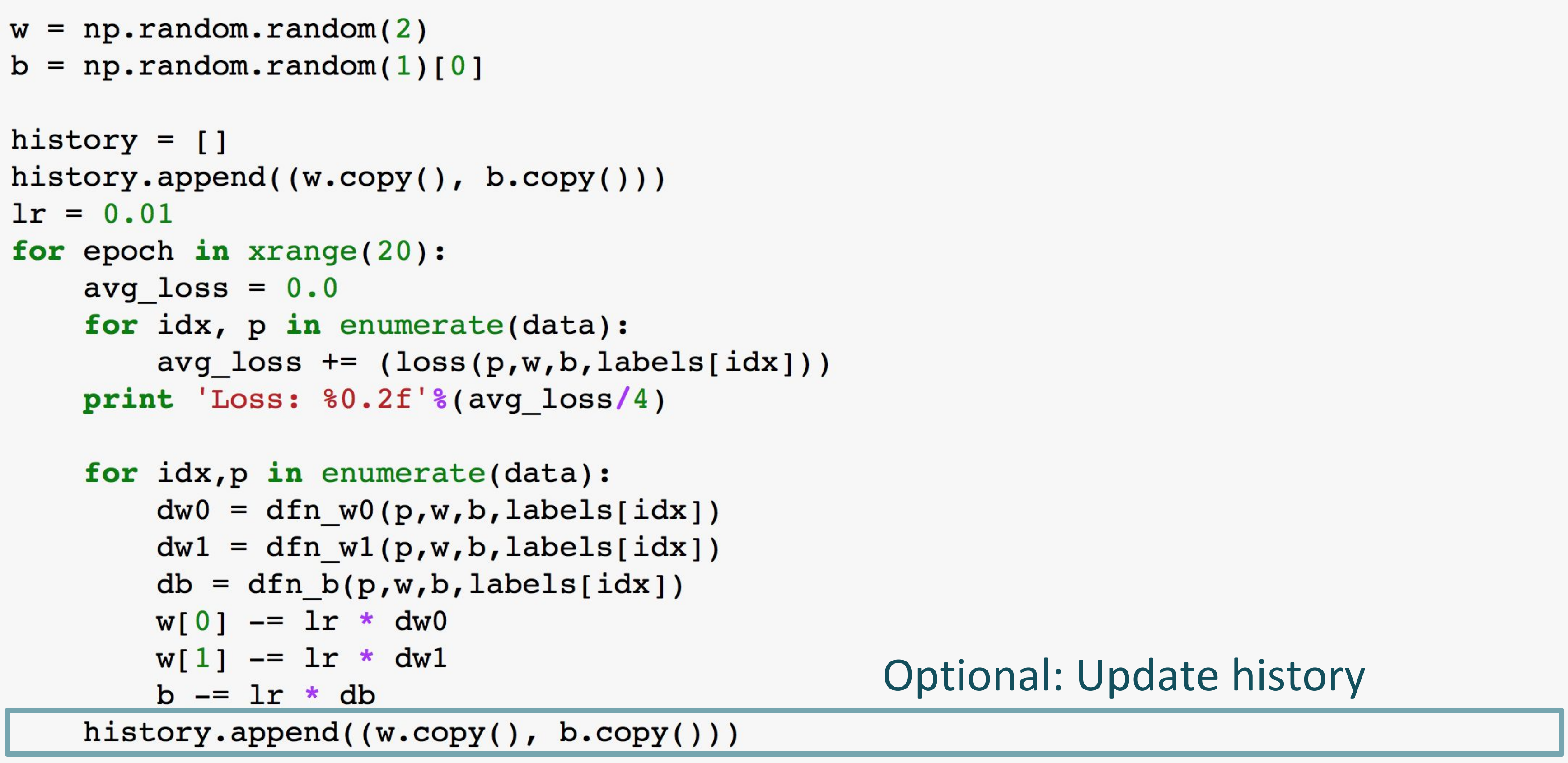
$$w_0 = w_0 - \eta \cdot \frac{\partial \mathcal{L}}{\partial w_0}$$

$$w_1 = w_1 - \eta \cdot \frac{\partial \mathcal{L}}{\partial w_1}$$

$$b = b - \eta \cdot \frac{\partial \mathcal{L}}{\partial b}$$

# Optimization Exercise

## Optimization

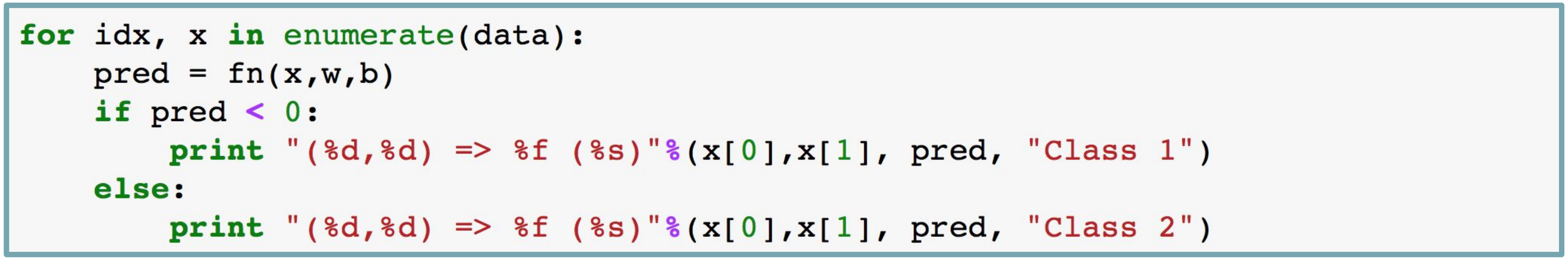


```
w = np.random.random(2)
b = np.random.random(1)[0]

history = []
history.append((w.copy(), b.copy()))
lr = 0.01
for epoch in xrange(20):
    avg_loss = 0.0
    for idx, p in enumerate(data):
        avg_loss += (loss(p,w,b,labels[idx]))
    print 'Loss: %0.2f'%(avg_loss/4)

    for idx,p in enumerate(data):
        dw0 = dfn_w0(p,w,b,labels[idx])
        dw1 = dfn_w1(p,w,b,labels[idx])
        db = dfn_b(p,w,b,labels[idx])
        w[0] -= lr * dw0
        w[1] -= lr * dw1
        b -= lr * db
    history.append((w.copy(), b.copy()))
```

Optional: Update history

# Optimization Exercise

## Prediction

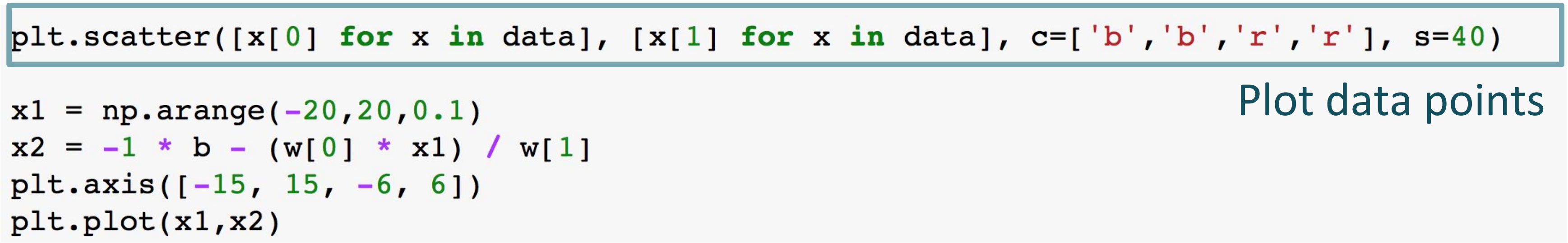


```
for idx, x in enumerate(data):
    pred = fn(x,w,b)
    if pred < 0:
        print "(%d,%d) => %f (%s)"%(x[0],x[1], pred, "Class 1")
    else:
        print "(%d,%d) => %f (%s)"%(x[0],x[1], pred, "Class 2")
```

Once we have optimized for w and b, we can just use fn to compute predictions for any data points!

# Optimization Exercise

## Bonus: Plotting

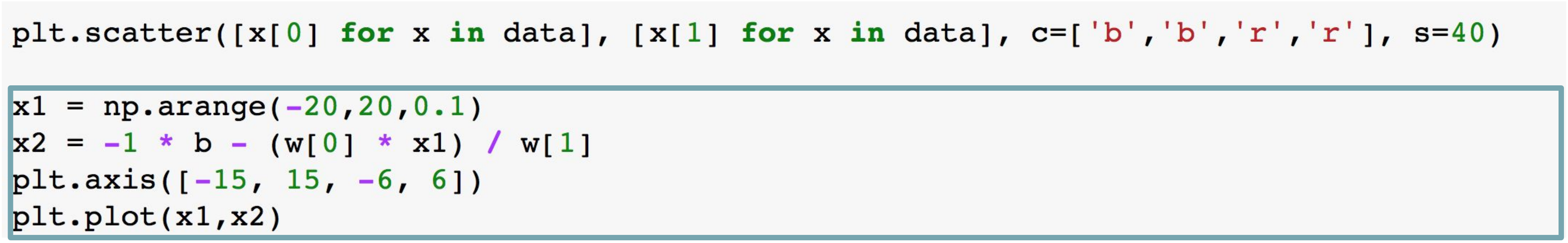


```
plt.scatter([x[0] for x in data], [x[1] for x in data], c=['b','b','r','r'], s=40)

x1 = np.arange(-20,20,0.1)
x2 = -1 * b - (w[0] * x1) / w[1]
plt.axis([-15, 15, -6, 6])
plt.plot(x1,x2)
```

Plot data points

# Optimization Exercise

## Bonus: Plotting

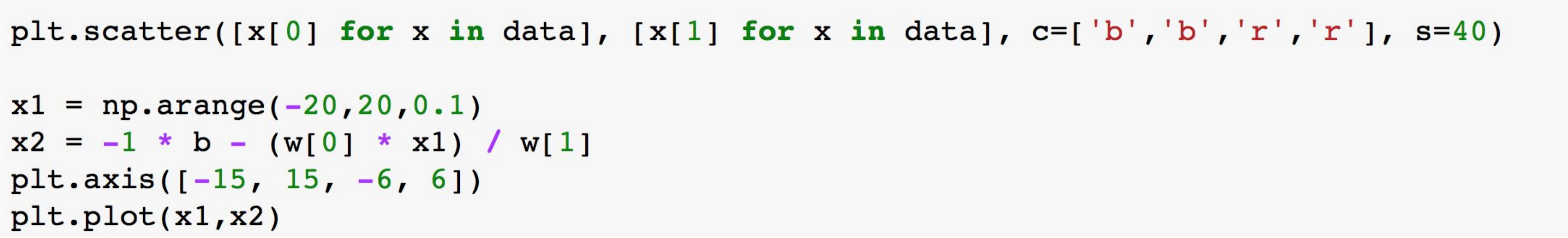


```
plt.scatter([x[0] for x in data], [x[1] for x in data], c=['b','b','r','r'], s=40)

x1 = np.arange(-20,20,0.1)
x2 = -1 * b - (w[0] * x1) / w[1]
plt.axis([-15, 15, -6, 6])
plt.plot(x1,x2)
```

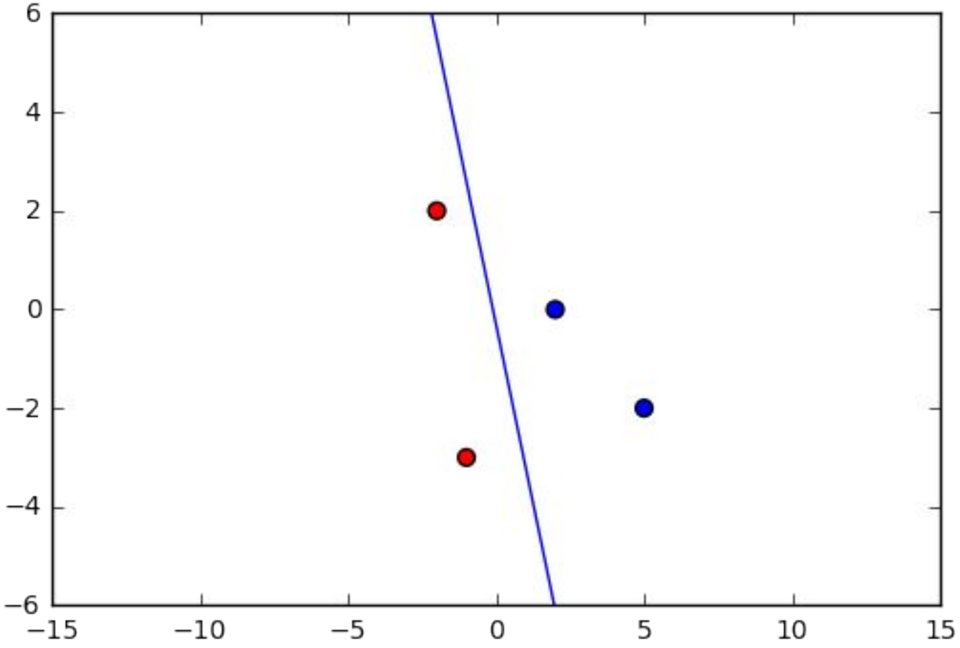
Plot decision boundary

# Optimization Exercise

## Bonus: Plotting



```
plt.scatter([x[0] for x in data], [x[1] for x in data], c=['b','b','r','r'], s=40)

x1 = np.arange(-20,20,0.1)
x2 = -1 * b - (w[0] * x1) / w[1]
plt.axis([-15, 15, -6, 6])
plt.plot(x1,x2)
```



# Optimization Exercise

Let’s put it all together and see it in action

1. Analytical gradient
2. Backproped gradient
3. Overall optimization

# Optimization Exercise

Example #2: Consider the *linear classifier* and *Hinge loss*:

Objective Function

$$\begin{aligned} f(x, W, b) &= W \cdot x + b \\ &= w_0 \cdot x_0 + w_1 \cdot x_1 + b \end{aligned}$$

Loss Function

$$\mathcal{L} = \max(0, 1 - y \cdot f(x, W, b))$$

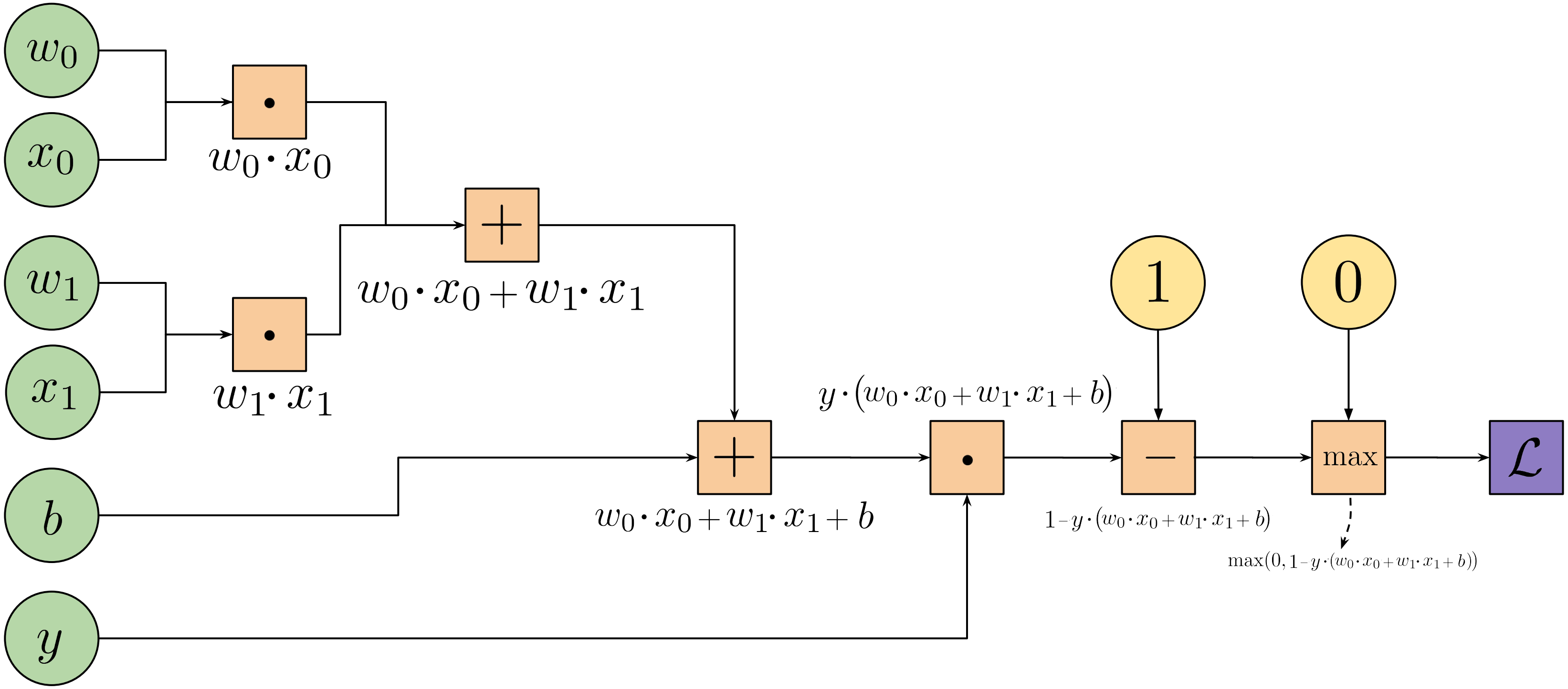
# Exercise

Derive the gradients $\frac{\partial \mathcal{L}}{\partial w_0}$, $\frac{\partial \mathcal{L}}{\partial w_1}$ and $\frac{\partial \mathcal{L}}{\partial b}$:

1. Use derivation rules to derive the gradients analytically
2. Build the computation graph to use backpropagation

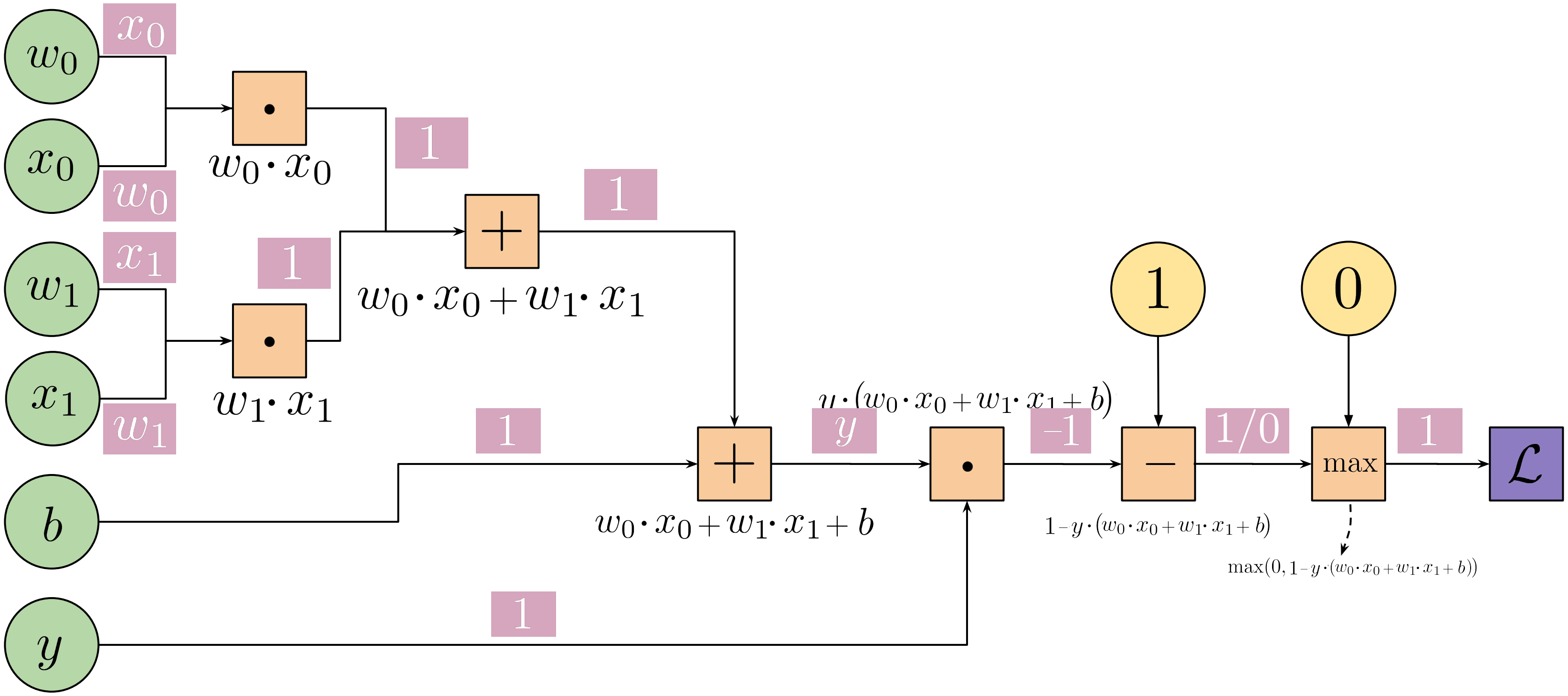
# Hinge Loss

$$\mathcal{L} = \max(0, 1 - y \cdot f(x, W, b))$$



# Hinge Loss

$$\mathcal{L} = \max(0, 1 - y \cdot f(x, W, b))$$



# Hinge Loss

Analytical gradients:

$$\frac{\partial \mathcal{L}}{\partial w_0} = \begin{cases} 0, & y \cdot f(x, W, b) \geq 1 \\ -y \cdot x_0, & y \cdot f(x, W, b) < 1 \end{cases}$$

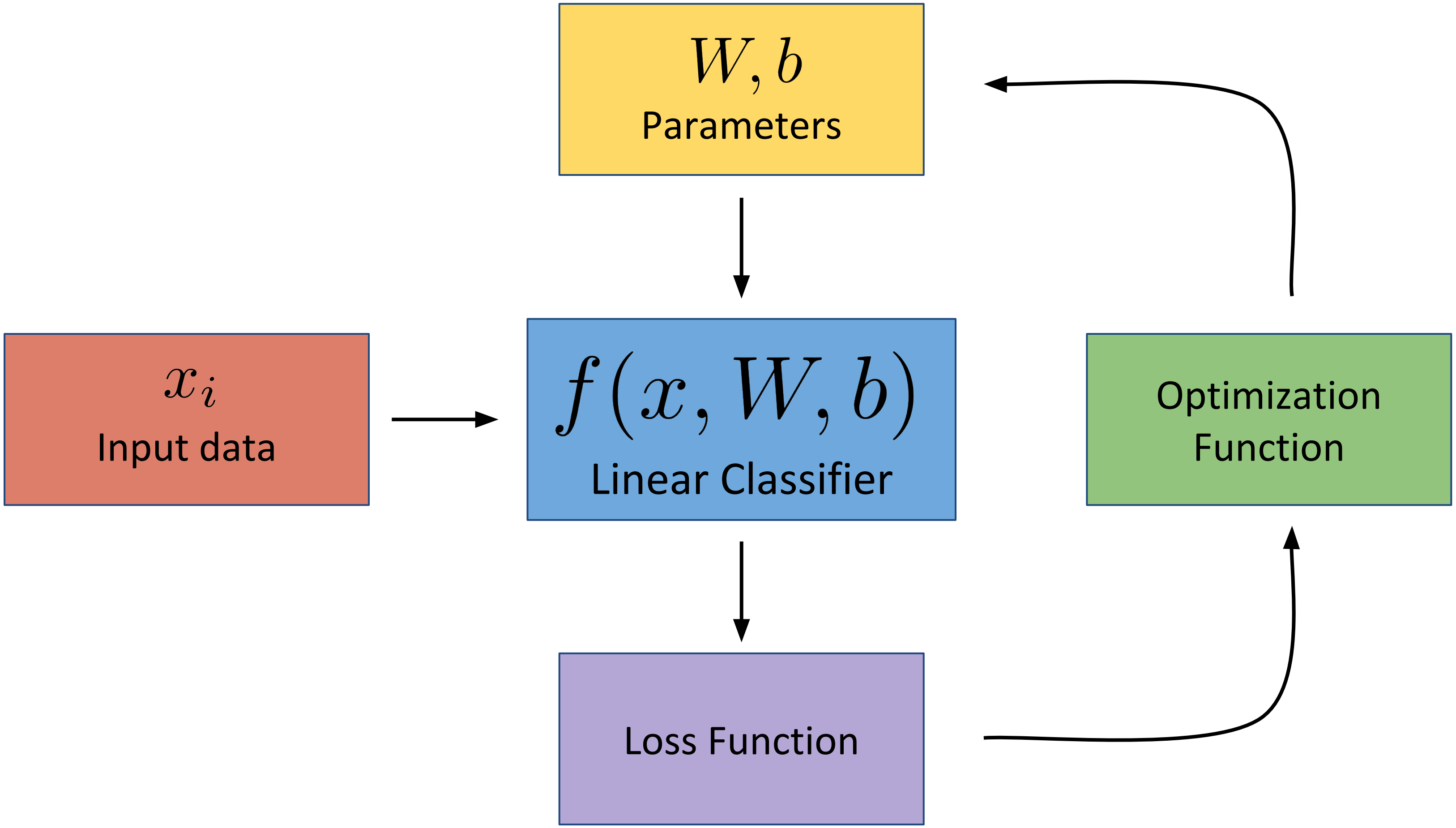
$$\frac{\partial \mathcal{L}}{\partial w_1} = \begin{cases} 0, & y \cdot f(x, W, b) \geq 1 \\ -y \cdot x_1, & y \cdot f(x, W, b) < 1 \end{cases}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \begin{cases} 0, & y \cdot f(x, W, b) \geq 1 \\ -y, & y \cdot f(x, W, b) < 1 \end{cases}$$

# Exercise

Now implement the optimization in code!

# Recap



$W, b$

Parameters

$x_i$

Input data

$f(x, W, b)$

Linear Classifier

Optimization Function

Loss Function

# Practical Considerations

## Multiclass classification

# Multiclass Classification

Recall that we have been using *linear regression* so far and making decisions based on the sign of the output

f(, W, b) = 1 Real Number

Output

If the number is less than 0, it is *accident prone*, else it is *not accident prone*

# Multiclass Classification

In general, we design our function `f` such that we output one number per class:

`f(`  `, W, b)   =   2 Numbers`

Outputs

The scores for the two classes - *accident prone* and *not accident prone*

# Multiclass Classification

In regression:

$$f(x, w, b) = w_0 \cdot x_0 + w_1 \cdot x_1 + b$$
$$= w \cdot x + b \longleftarrow \text{Output: A real number}$$

In classification:

$$f(x, W, b) = W \cdot x + b \longleftarrow \text{Output: A vector}$$

# Multiclass Classification

In regression:

$$f(x, w, b) = w_0 \cdot x_0 + w_1 \cdot x_1 + b$$

$$= \boxed{w} \cdot \boxed{x} + \boxed{b} \longleftarrow \text{Output: A real number}$$

Vector

Real number

In classification:

$$f(x, W, b) = \boxed{W} \cdot \boxed{x} + \boxed{b} \longleftarrow \text{Output: A vector}$$

Matrix

Vector

# Multiclass Classification

From now on, we will use this generalized technique, since it can be easily extended to more than two classes

f(  , W, b) = Numbers

Outputs
The scores for the classes

# Multiclass Classification

From now on, we will use this generalized technique, since it can be easily extended to more than two classes

`f(`  `, W, b)  =  Numbers (vector)`

Everything else remains the same - the loss functions now operates on vectors instead of real numbers

# Multiclass Classification

## Prediction

In regression:

f(, W, b) >= 0

In classification:

argmax(f(, W, b))

Pick the class with the highest score

# Practical Considerations

## Stochastic vs Batch gradient descent

# Gradient Descent

In the previous implementation, we compute the objective function and gradient for an example, adjusted our parameters and then continued with the next example

# Gradient Descent

In the previous implementation, we compute the objective function and gradient for an example, adjusted our parameters and then continued with the next example

This approach is known as ***stochastic gradient descent*** (SGD)

# Gradient Descent

Another approach is to accumulate the gradients over all examples, and then do a single update to the parameters - this approach is known as ***Batch gradient descent***

# Gradient Descent

Batch gradient descent leads to more “stable” updates - the direction towards the optimal parameters is computed after looking at all examples, instead of just one!

# Gradient Descent

What if you had a few outliers (bad examples)

- SGD will cause the parameters to drift farther from their optimal values when the update loop goes over these outliers
- Batch GD will drown out the effect of the outliers since there are many more good examples

# Gradient Descent

But...

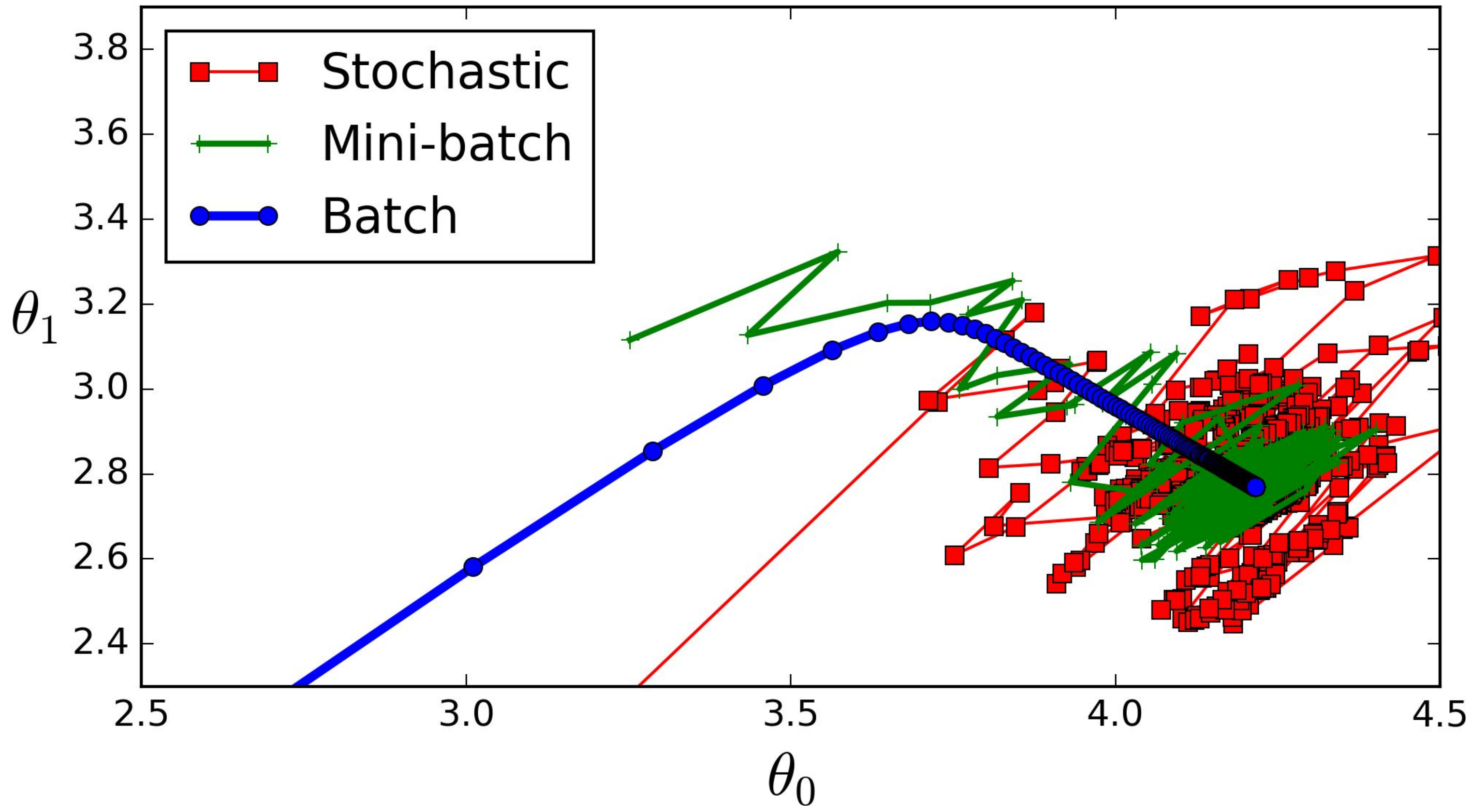
- Batch GD requires us to look over the entire dataset before making any progress - so it’s much slower
- The entire dataset may not even fit in memory, so making the code efficient would be more difficult

# Gradient Descent

Solution:

- Minibatch SGD: Perform updates after looking at a “minibatch” (e.g. 32 data points)
- Much faster than Batch GD, but largely avoids the issues with SGD

# Gradient Descent



https://stats.stackexchange.com/a/153535

# Practical Considerations

## Softmax function

# Softmax

So far, our classifier has always output some “scores”, and we just pick whichever score is higher:

f(  , W, b) = 2 Numbers

Outputs

The scores for the two classes - *accident prone* and *not accident prone*

# Softmax

However, these scores are not *interpretable.*

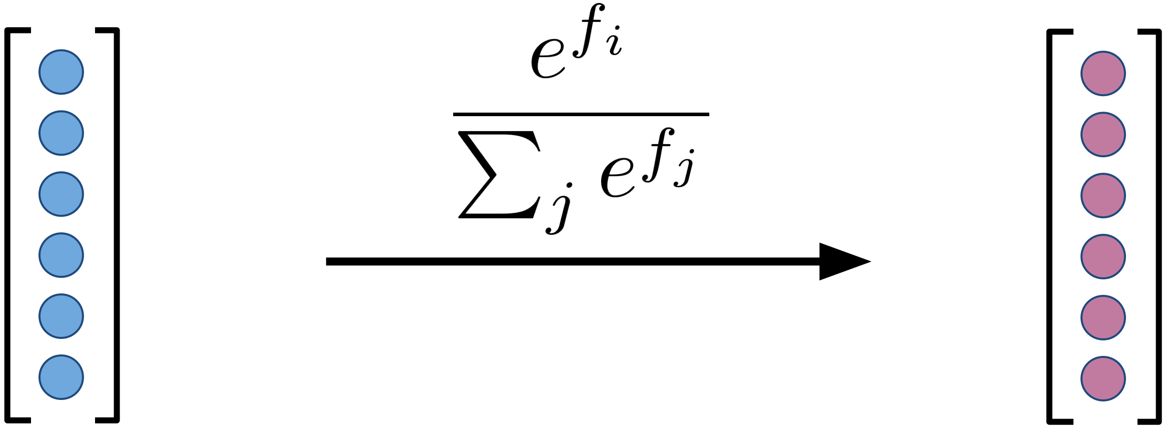
Their absolute values don’t give us any insight, we can only compare them relatively

`f(`  `, W, b) = 2 Numbers`

Outputs
The scores for the two classes - *accident prone* and *not accident prone*

# Softmax

The softmax function helps us transform these values into probability distributions:



$$\xrightarrow{\frac{e^{f_i}}{\sum_j e^{f_j}}}$$

Scores from the classifier

$f$

Scores as a probability distribution

# Softmax

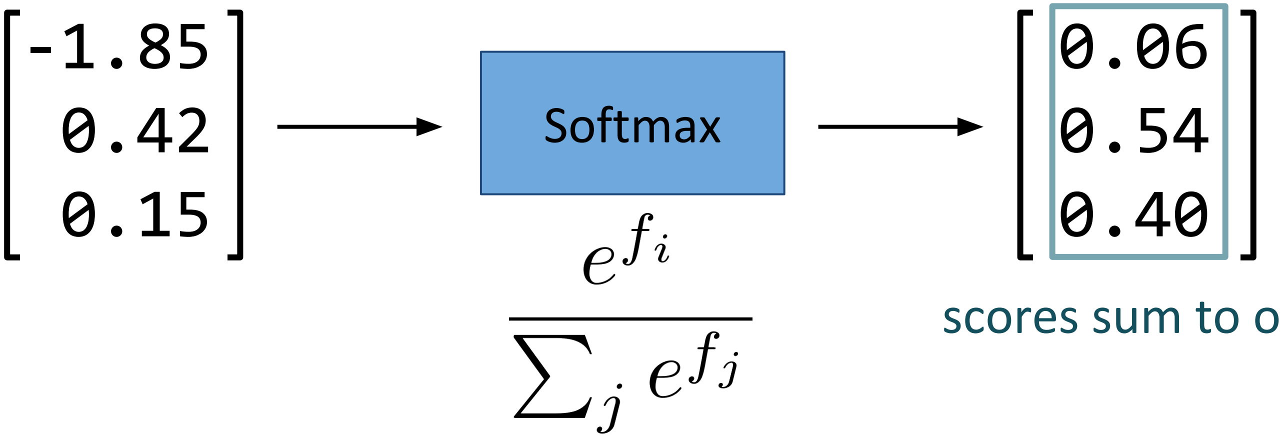
The softmax function helps us transform these values into probability distributions:

$$\begin{bmatrix}-1.85\\ 0.42\\ 0.15\end{bmatrix} \longrightarrow \boxed{\text{Softmax}} \longrightarrow \begin{bmatrix}0.06\\ 0.54\\ 0.40\end{bmatrix}$$

$$\frac{e^{f_i}}{\sum_j e^{f_j}}$$

# Softmax

The softmax function helps us transform these values into probability distributions:



$$\begin{bmatrix} -1.85 \\ 0.42 \\ 0.15 \end{bmatrix} \longrightarrow \boxed{\text{Softmax}} \longrightarrow \begin{bmatrix} \boxed{\begin{matrix} 0.06 \\ 0.54 \\ 0.40 \end{matrix}} \end{bmatrix}$$

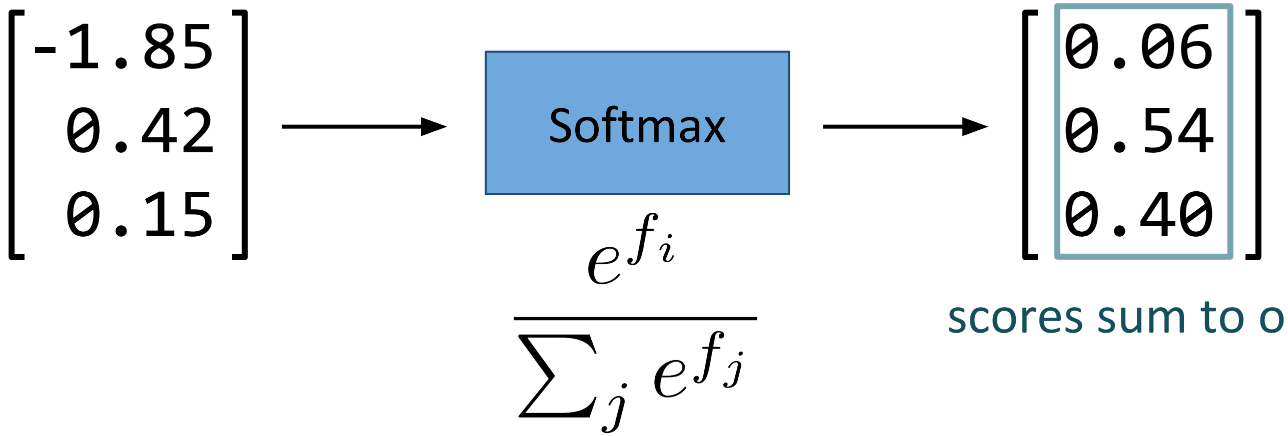
$$\frac{e^{f_i}}{\sum_j e^{f_j}}$$

each output can be treated as the probability of that class

scores sum to one

# Softmax

The softmax function helps us transform these values into probability distributions:



each output can be treated as the probability of that class

$$\begin{bmatrix} -1.85 \\ 0.42 \\ 0.15 \end{bmatrix} \longrightarrow \boxed{\text{Softmax}} \longrightarrow \begin{bmatrix} \boxed{\begin{matrix} 0.06 \\ 0.54 \\ 0.40 \end{matrix}} \end{bmatrix}$$

$$\frac{e^{f_i}}{\sum_j e^{f_j}}$$

scores sum to one

The Softmax function also acts as a *normalizer*, i.e. we can now compare scores from different models and examples directly

# Practical Considerations

## Cross Entropy loss

# Cross Entropy Loss

Recall from the previous lecture:

Mean Squared Error

$$L = \sum_{i=1}^{n} (f_i - y_i)^2$$

# Cross Entropy Loss

Recall from the previous lecture:

Mean Squared Error

We saw that MSE is better than just taking the absolute difference:

$$\boxed{L = \sum_{i=1}^{n} |f_i - y_i|}$$

# Cross Entropy Loss

Recall from the previous lecture:

Mean Squared Error

In practice, we use *Cross Entropy loss*, which generally performs better for more complex models.

# Cross Entropy Loss

$$H_y(f) = -\sum_i y_i \log(f_i)$$

Here, $y$ represents the true probability distribution (so $y_i = 1$ for the correct class $i$, and $0$ otherwise)

$f_i$ represents the score of class $i$ from our classifier

# Cross Entropy Loss

$$H_y(f) = -\sum_i y_i \log(f_i)$$

$$= -y_c \log(f_c)$$

Simplifying for our case,
if $c$ is the correct class, then $y_c = 1$, and all other $y_i$'s are $0$
Therefore, we only have one element left from the summation

# Cross Entropy Loss

$$\begin{aligned} H_y(f) &= -\sum_i y_i \log(f_i) \\ &= -y_c \log(f_c) \\ &= -\log(f_c) \end{aligned}$$

# Cross Entropy Loss

Mean Squared Error

$$L = \sum_{i=1}^{n} (f_i - y_i)^2$$

Cross Entropy

$$L = -\log(f_c)$$

# Cross Entropy Loss

## Why cross entropy?

Consider three people, Person1 is a *Democrat*, Person2 is a *Republican* and Person3 is *Other*. We have two models to classify these people:

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/

# Cross Entropy Loss

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

Both models misclassify *Person3*, but is one model better than the other?

# Cross Entropy Loss

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

**Model 2** is better, since it classifies *Person1* and *Person2* with higher scores on the correct class, and mis-classifies *Person3* with a smaller error in the scores

# Cross Entropy Loss

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

**Person1:** 0.54

**Person2:** 0.54

**Person3:** 1.34

**Model 1 Average:** 0.81

**Person1:** 0.14

**Person2:** 0.14

**Person3:** 0.74

**Model 2 Average:** 0.34

Mean Squared Error

# Cross Entropy Loss

|  | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

|  | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

**Person1:** -log(0.4) = 0.92

**Person2:** -log(0.4) = 0.92

**Person3:** -log(0.1) = 2.30

**Model 1 Average:** 1.38

**Person1:** 0.36

**Person2:** 0.36

**Person3:** 1.20

**Model 2 Average:** 0.64

Cross Entropy

# Cross Entropy Loss

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

Mean Squared Error

**Model 1 Average:** 0.81

**Model 2 Average:** 0.34

Cross Entropy

**Model 1 Average:** 1.38

**Model 2 Average:** 0.64

# Cross Entropy Loss

Mean Squared Error

**Model 1 Average:** 0.81

**Model 2 Average:** 0.34

Cross Entropy

**Model 1 Average:** 1.38

**Model 2 Average:** 0.64

Cross Entropy Loss difference between the two models is greater than the Mean Squared Error!

# Cross Entropy Loss

In general, *Mean Squared Error* penalizes incorrect predictions much more than *Cross Entropy*

# Cross Entropy Loss

A more principled reason arises from the underlying mathematics of MSE and Cross Entropy

MSE causes the gradients to become very small as the network scores become better, so learning slows down!

# Cross Entropy and Softmax

# Cross Entropy and Softmax

Cross Entropy is mathematically defined to compare two probability distributions

# Cross Entropy and Softmax

Cross Entropy is mathematically defined to compare two probability distributions

Our ground truth is already represented as a probability distribution (with all the probability mass on the correct class)

$$y = \begin{bmatrix} 0.00 \\ 1.00 \\ 0.00 \end{bmatrix}$$

# Cross Entropy and Softmax

Cross Entropy is mathematically defined to compare two probability distributions

However, the scores directly from a linear classifier do not form any such distribution:

$$f = \begin{bmatrix} -1.85 \\ 0.42 \\ 0.15 \end{bmatrix}$$

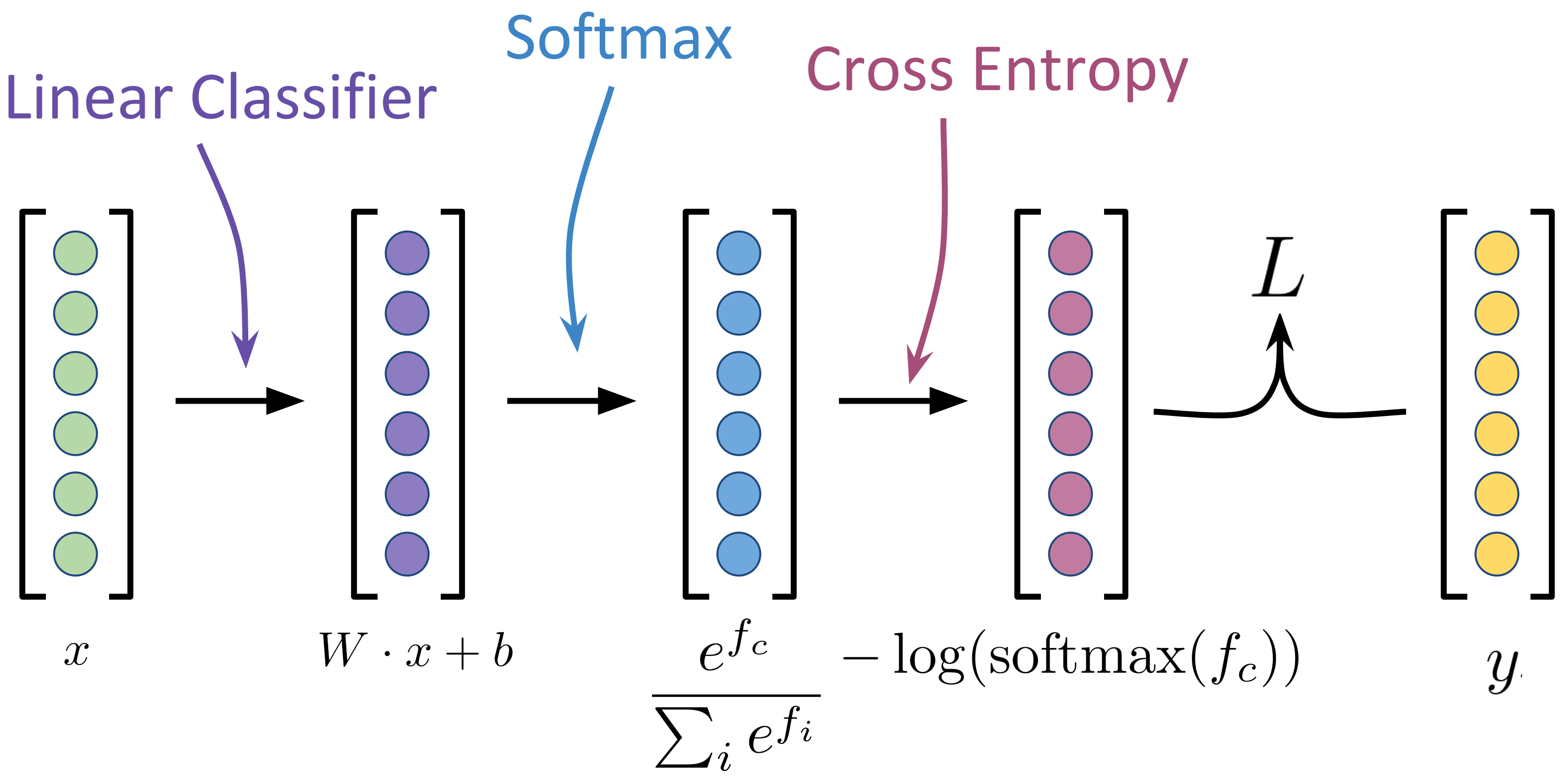
# Cross Entropy and Softmax

Cross Entropy is mathematically defined to compare two probability distributions

Solution: Use softmax!

$$\text{softmax}(f) = \begin{bmatrix} 0.06 \\ 0.54 \\ 0.40 \end{bmatrix}$$

# Putting it all together



Linear Classifier

Softmax

Cross Entropy

$L$

$x$

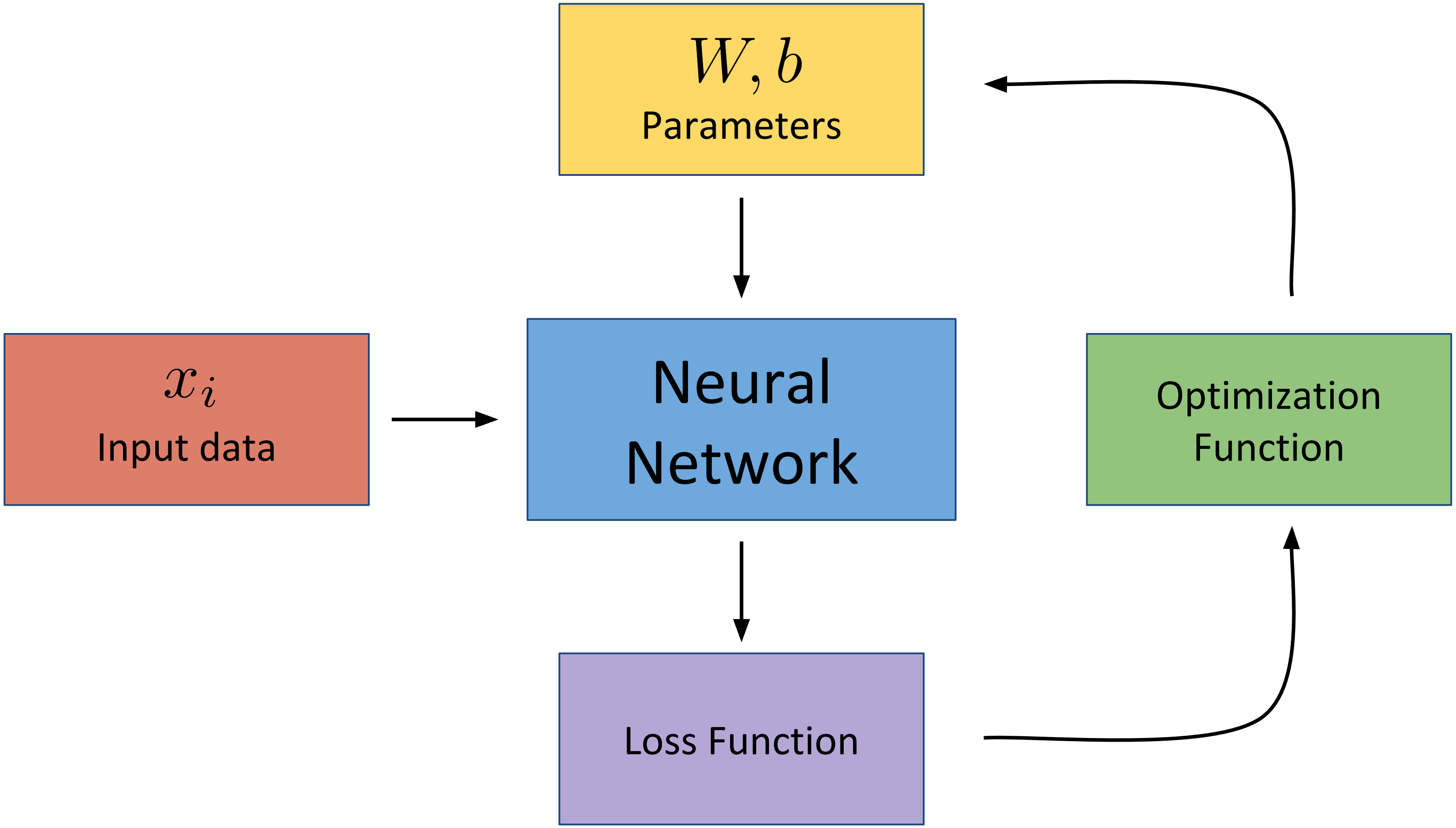
$W \cdot x + b$

$$\frac{e^{f_c}}{\sum_i e^{f_i}}$$

$-\log(\text{softmax}(f_c))$

$y$

# Overall Picture



$W, b$

Parameters

$x_i$

Input data

Neural Network

Optimization Function

Loss Function