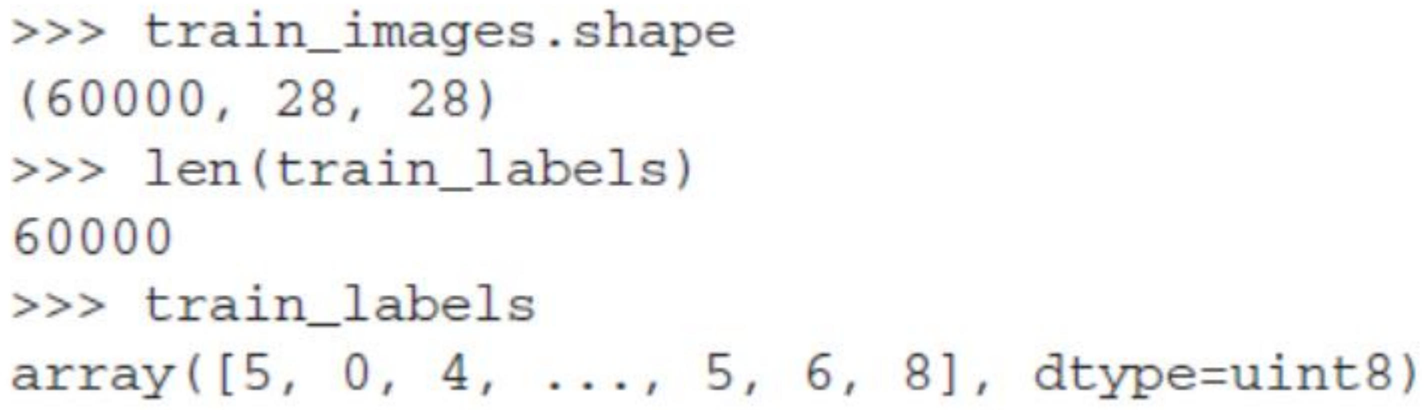
Deep Learning with Python

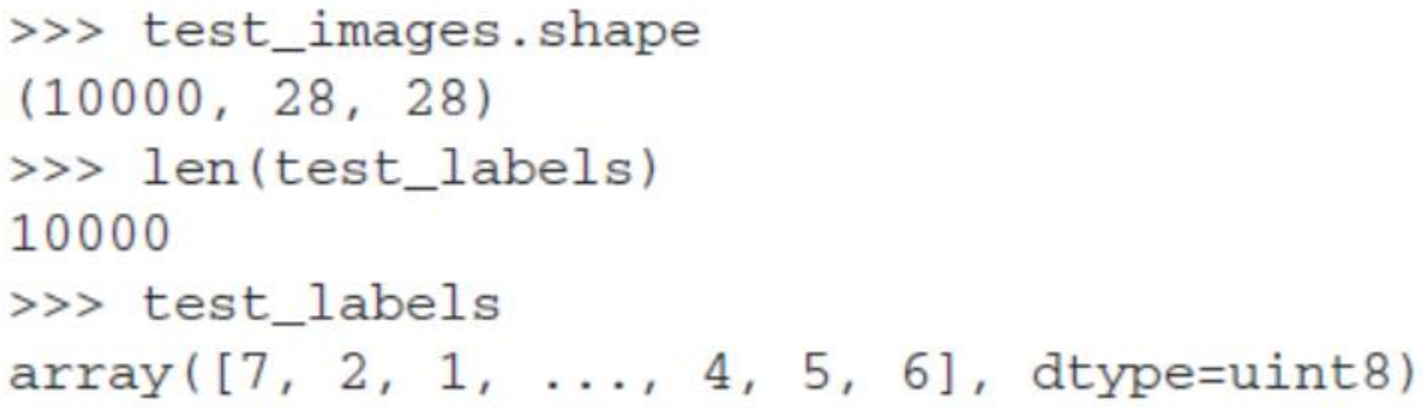
Chapter 2

Figure 2.1 MNIST sample digits

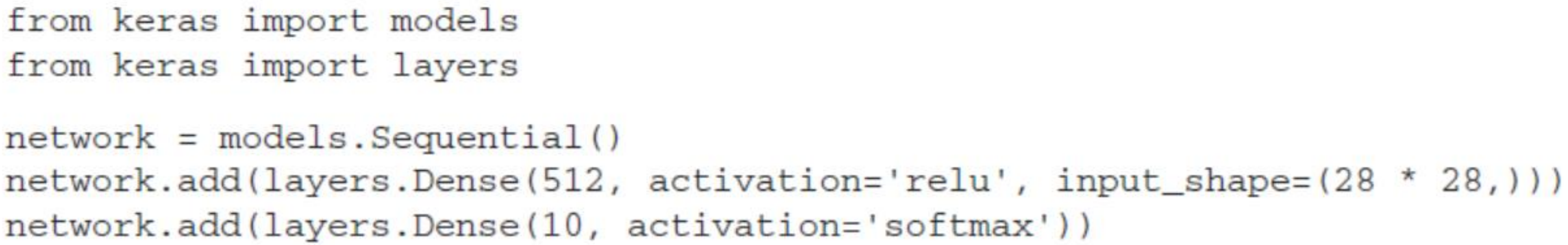
```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```



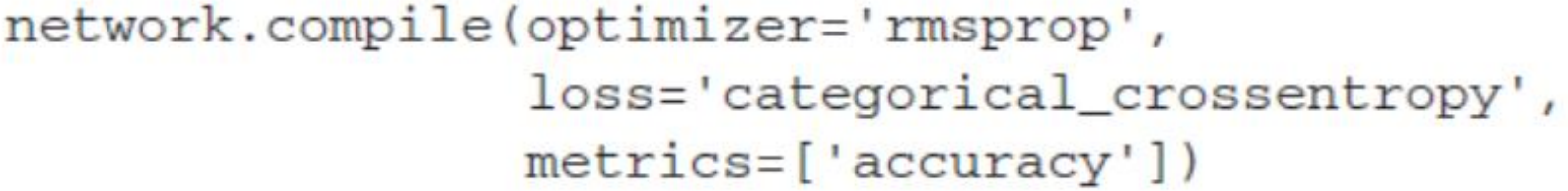
```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```



```
>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```



```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```



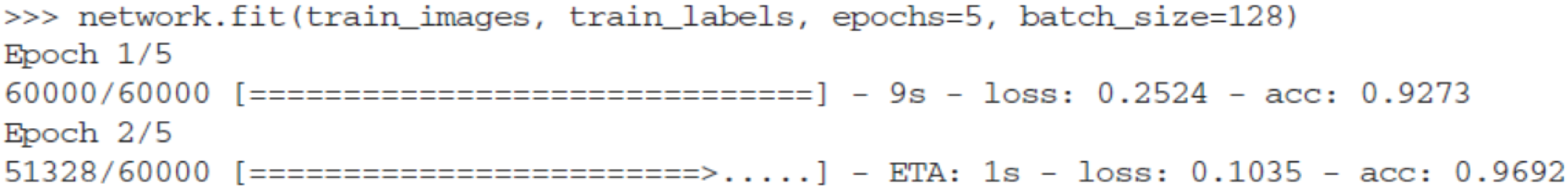
```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

Listing 2.4 Preparing the image data

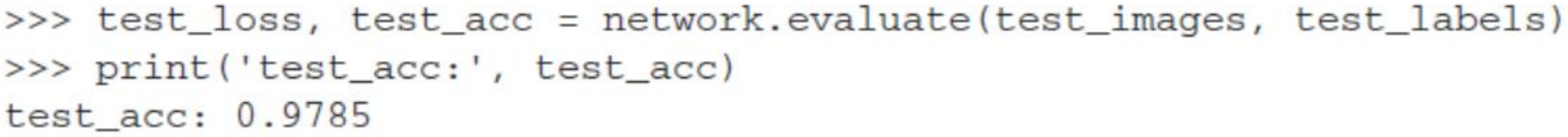
```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Listing 2.5 Preparing the labels

```
from keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```



```
>>> network.fit(train_images, train_labels, epochs=5, batch_size=128)
Epoch 1/5
60000/60000 [=============================] - 9s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [=======================>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```



```
>>> test_loss, test_acc = network.evaluate(test_images, test_labels)
>>> print('test_acc:', test_acc)
test_acc: 0.9785
```

Tensors

Scalars (0D tensors)

```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

Vectors (1D tensors)

```
>>> x = np.array([12, 3, 6, 14])
>>> x
array([12, 3, 6, 14])
>>> x.ndim
1
```

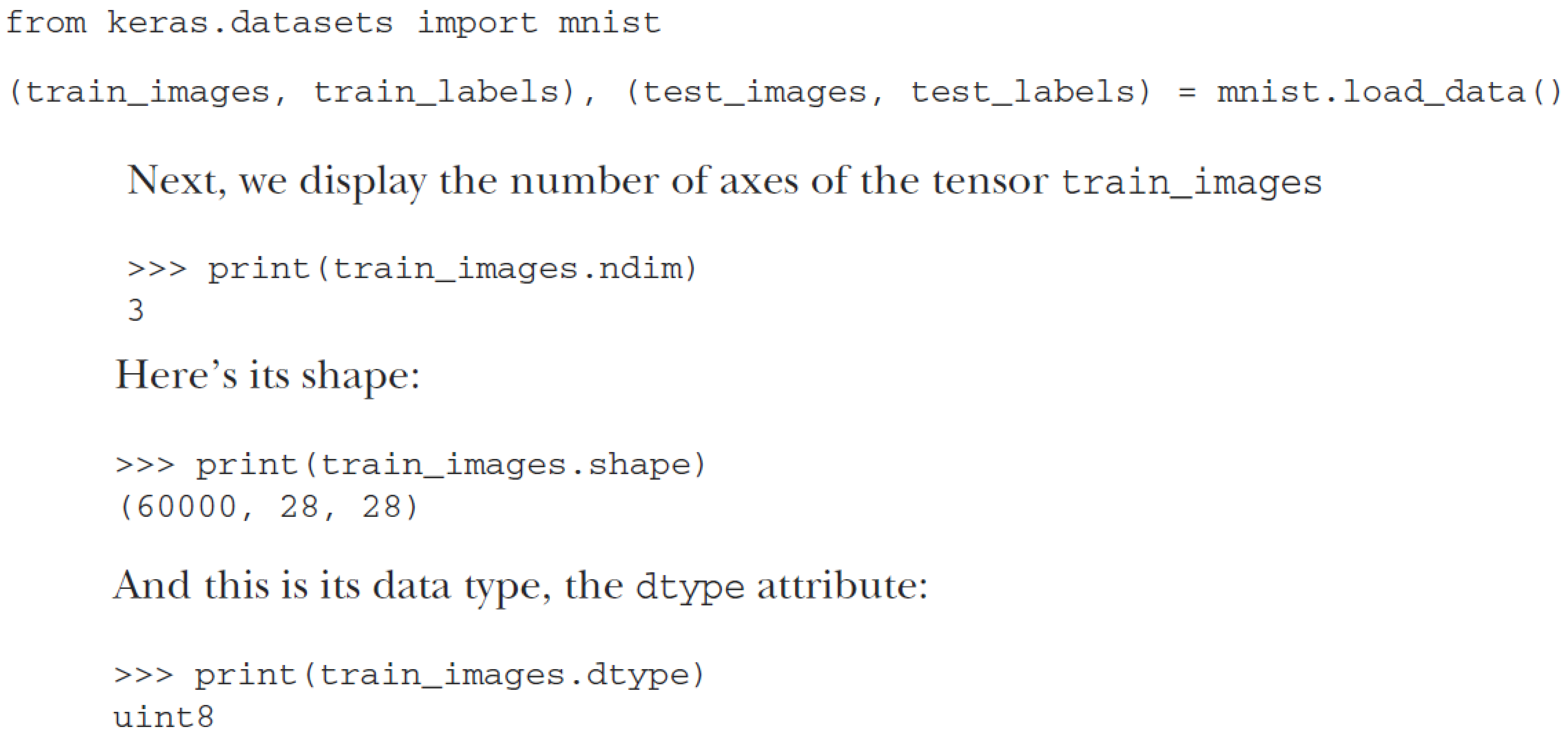
Matrices (2D tensors)

```
>>> x = np.array([[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]])
>>> x.ndim
2
```

3D tensors and higher-dimensional tensors

```
>>> x = np.array([[[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]]])
>>> x.ndim
3
```

Example



```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Next, we display the number of axes of the tensor `train_images`

```
>>> print(train_images.ndim)
3
```

Here’s its shape:

```
>>> print(train_images.shape)
(60000, 28, 28)
```

And this is its data type, the `dtype` attribute:

```
>>> print(train_images.dtype)
uint8
```

Example

Listing 2.6 Displaying the fourth digit

```
digit = train_images[4]

import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

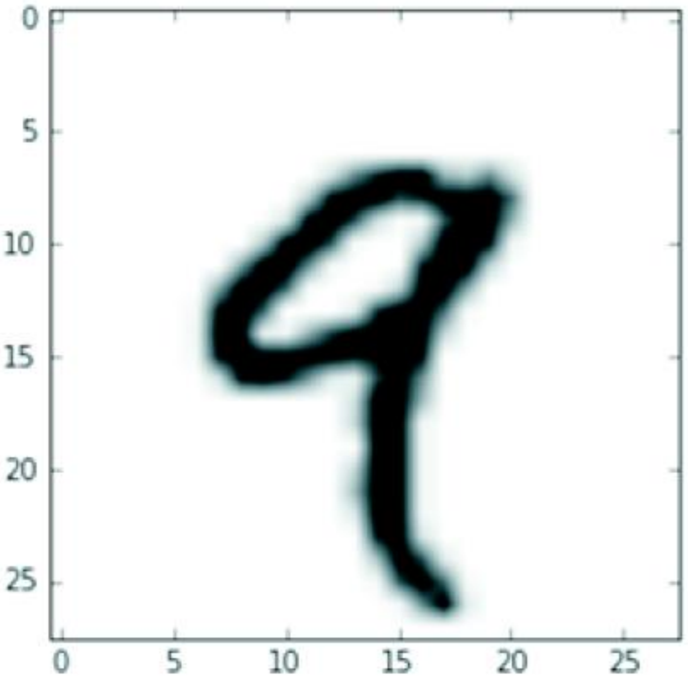
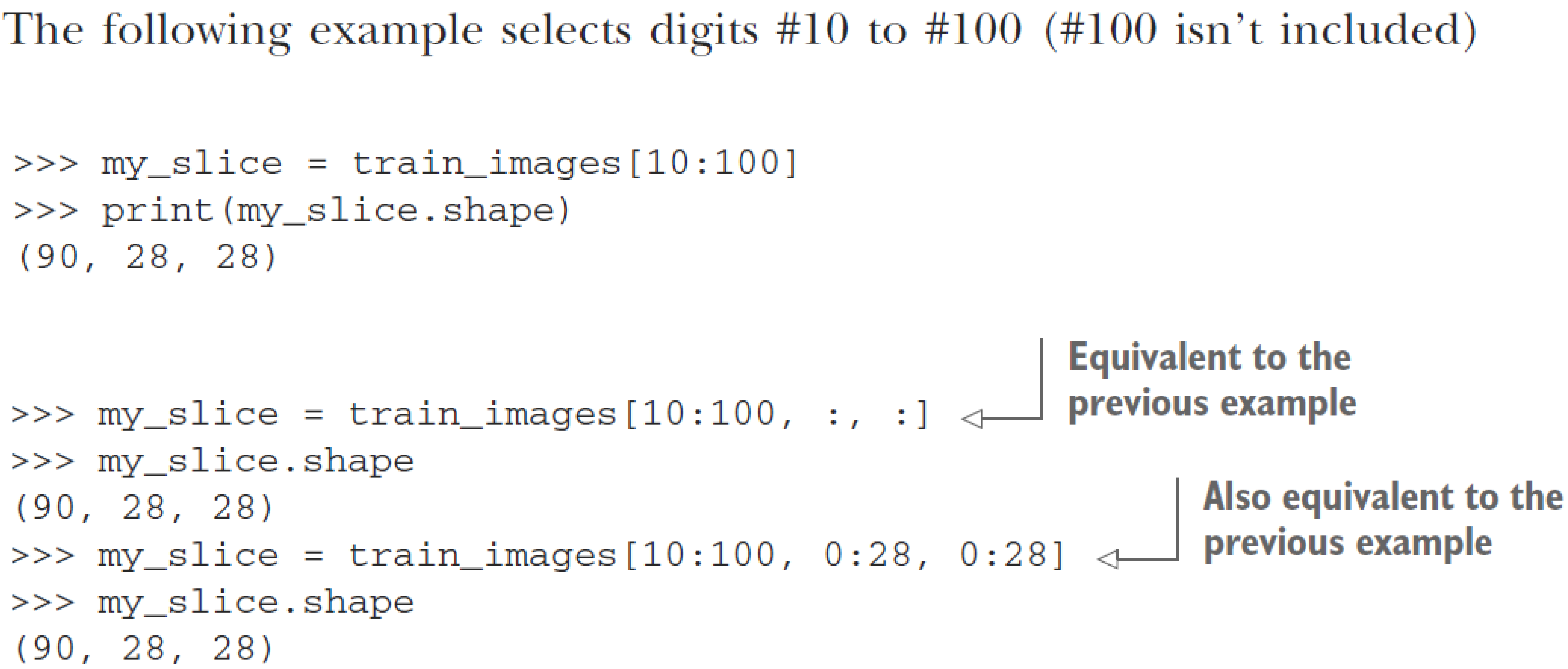


Figure 2.2 The fourth sample in our dataset

Tensor slicing



The following example selects digits #10 to #100 (#100 isn't included)

```
>>> my_slice = train_images[10:100]
>>> print(my_slice.shape)
(90, 28, 28)
```

```
>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

Equivalent to the previous example

Also equivalent to the previous example

in order to select 14 × 14 pixels in the bottom-right corner of all images

```
my_slice = train_images[:, 14:, 14:]
```

In order to crop the images to patches of 14 × 14 pixels centered in the middle, you do this:

```
my_slice = train_images[:, 7:-7, 7:-7]
```

Data batches

Deep-learning models don't process an entire dataset at once; rather, they break the data into small batches. Concretely, here's one batch of our MNIST digits, with batch size of 128:

```
batch = train_images[:128]
```

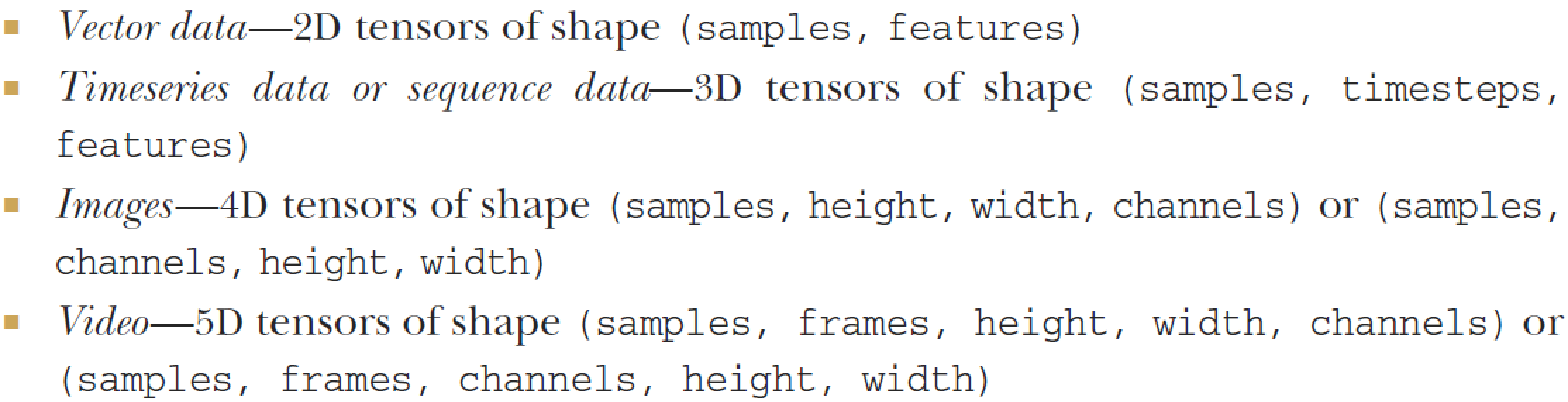
And here's the next batch:

```
batch = train_images[128:256]
```

And the *n*th batch:

```
batch = train_images[128 * n:128 * (n + 1)]
```

Real-world examples of data tensors



- *Vector data*—2D tensors of shape `(samples, features)`
- *Timeseries data or sequence data*—3D tensors of shape `(samples, timesteps, features)`
- *Images*—4D tensors of shape `(samples, height, width, channels)` or `(samples, channels, height, width)`
- *Video*—5D tensors of shape `(samples, frames, height, width, channels)` or `(samples, frames, channels, height, width)`

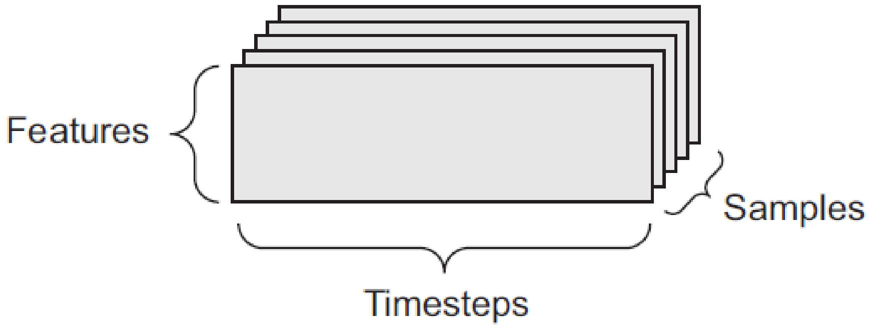


Figure 2.3 A 3D timeseries data tensor

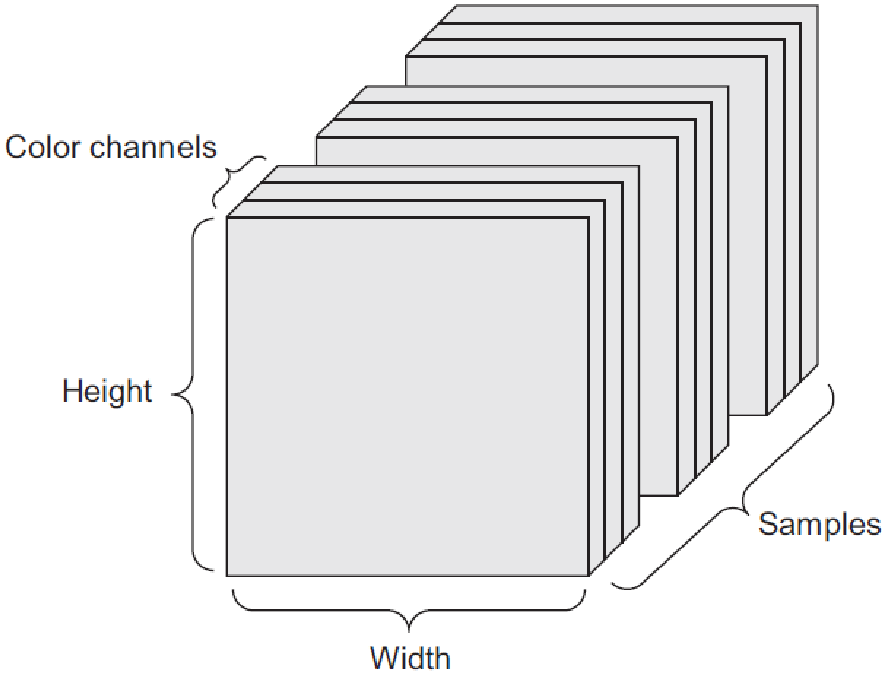
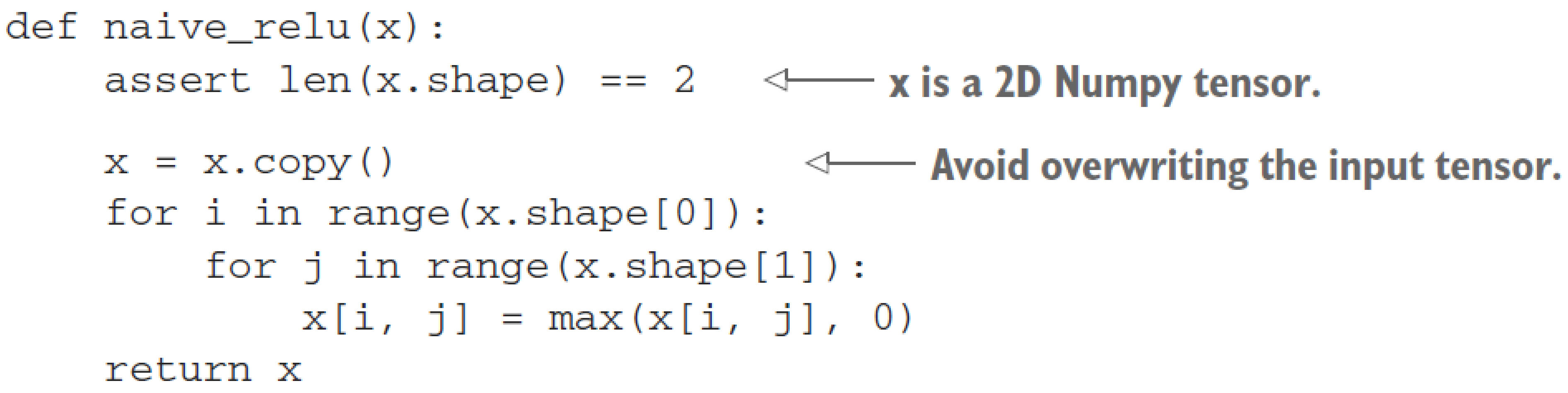


Figure 2.4 A 4D image data tensor (channels-first convention)

Tensor operations

```
keras.layers.Dense(512, activation='relu')
```

```
output = relu(dot(W, input) + b)
```

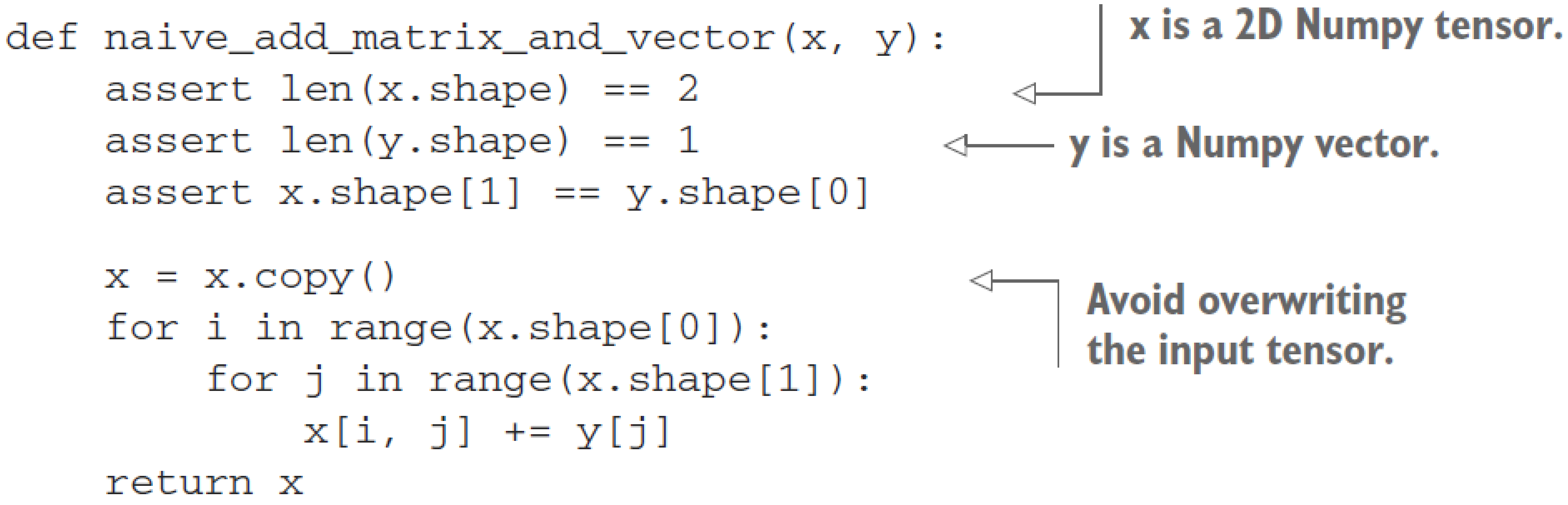


```
def naive_relu(x):
    assert len(x.shape) == 2

    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

x is a 2D Numpy tensor.

Avoid overwriting the input tensor.

tensor operations: Broadcasting



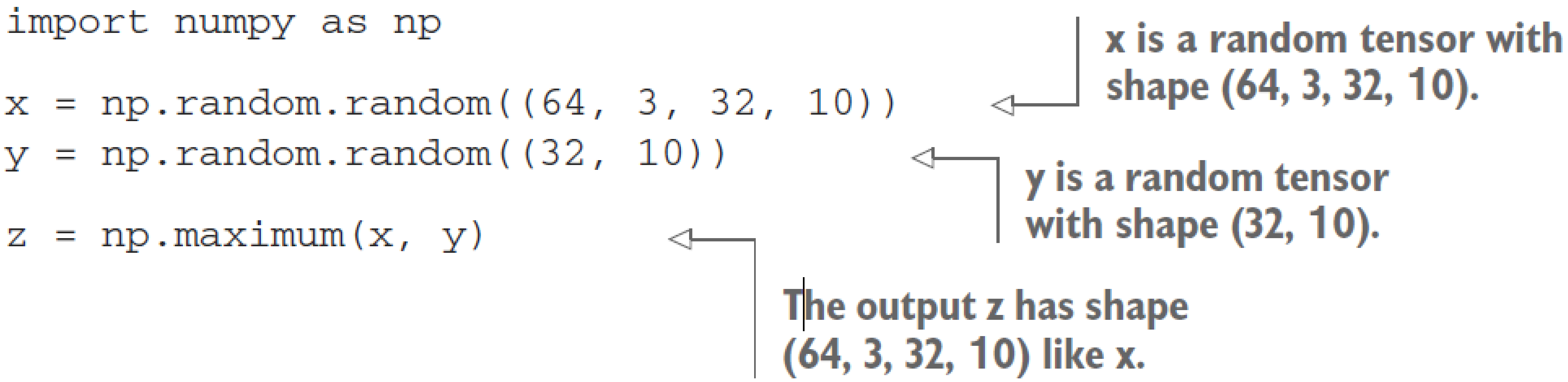
```
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]

    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

x is a 2D Numpy tensor.

y is a Numpy vector.

Avoid overwriting the input tensor.

Tensor operations: Broadcasting



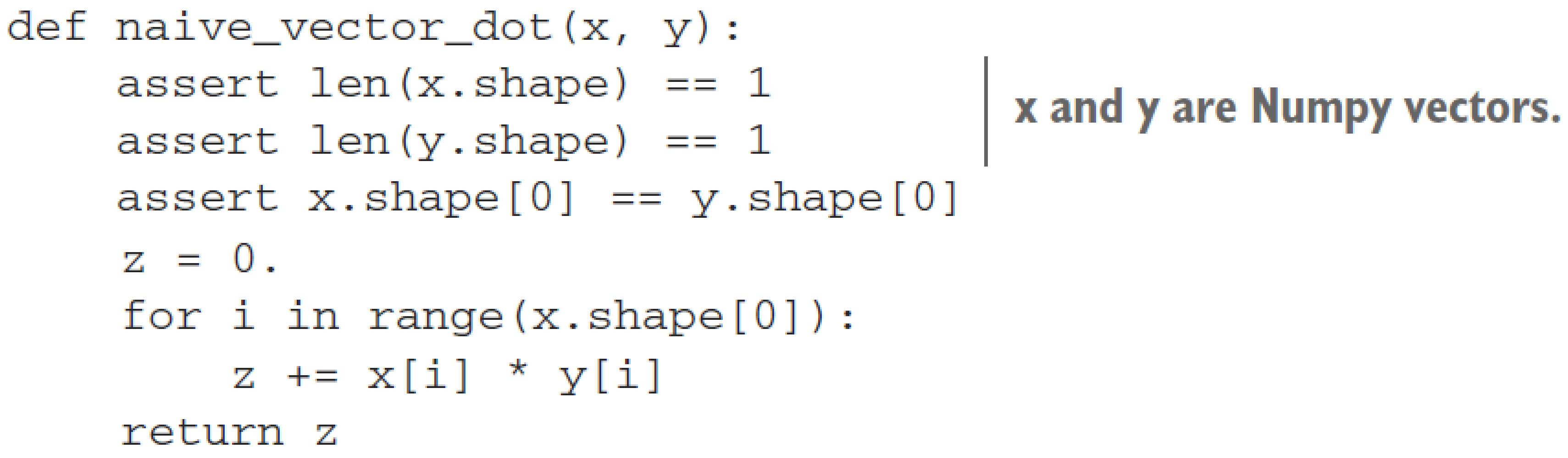
```
import numpy as np

x = np.random.random((64, 3, 32, 10))
y = np.random.random((32, 10))

z = np.maximum(x, y)
```

x is a random tensor with shape (64, 3, 32, 10).

y is a random tensor with shape (32, 10).

The output z has shape (64, 3, 32, 10) like x.

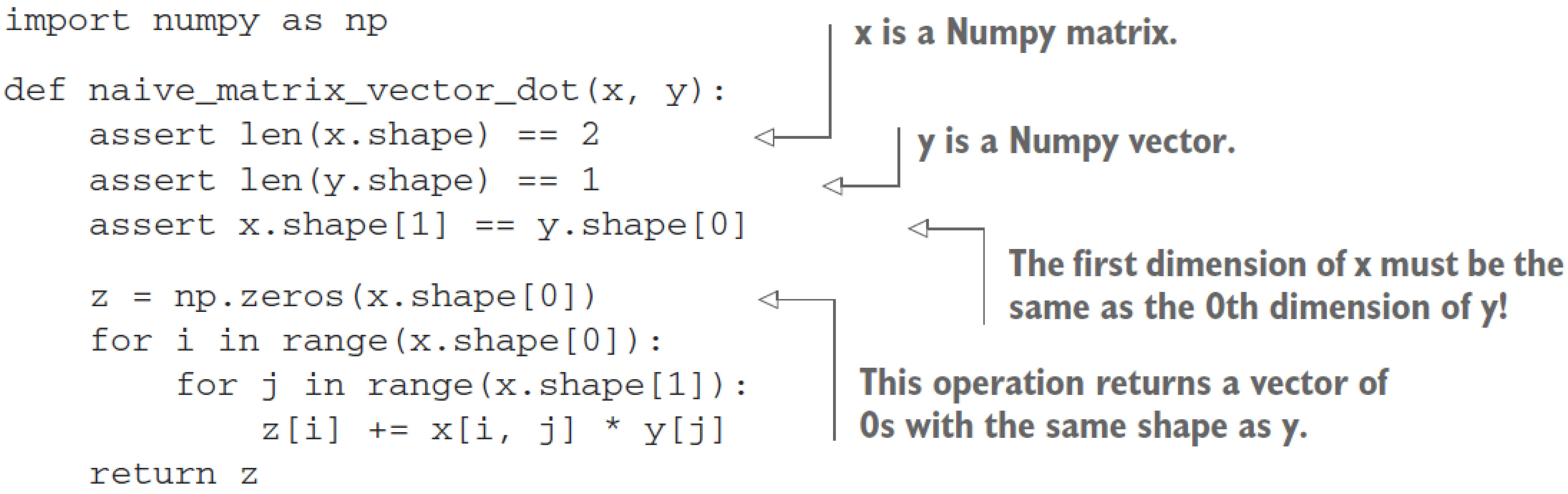
Tensor operations: Dot



```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
    z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

x and y are Numpy vectors.

Tensor operations: Dot



```
import numpy as np

def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]

    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
    return z
```

x is a Numpy matrix.

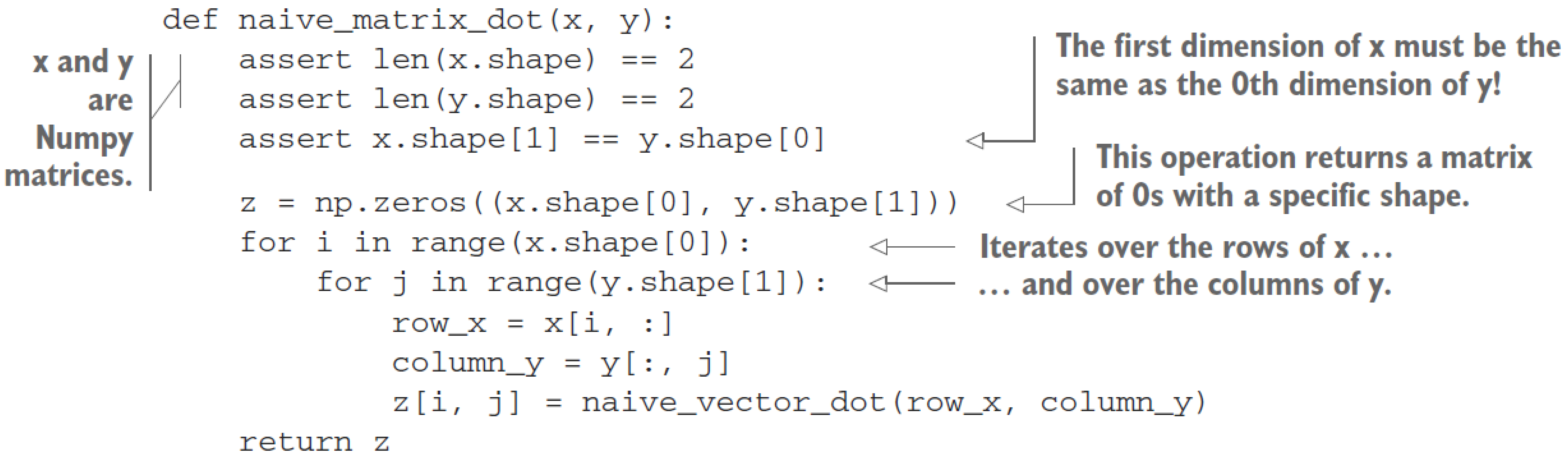
y is a Numpy vector.

The first dimension of x must be the same as the 0th dimension of y!

This operation returns a vector of 0s with the same shape as y.

```
def naive_matrix_vector_dot(x, y):
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        z[i] = naive_vector_dot(x[i, :], y)
    return z
```

Tensor operations: Dot



```
def naive_matrix_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]

    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            row_x = x[i, :]
            column_y = y[:, j]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```

x and y are Numpy matrices.

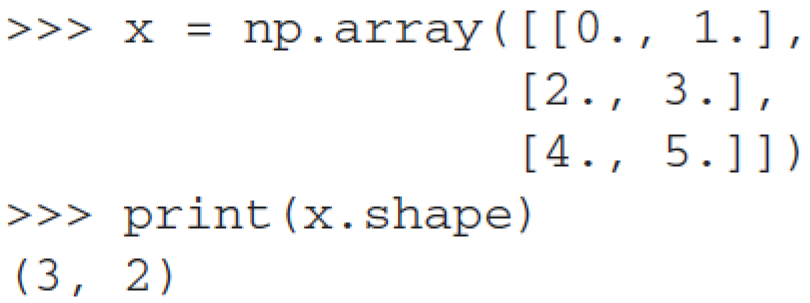
The first dimension of x must be the same as the 0th dimension of y!

This operation returns a matrix of 0s with a specific shape.

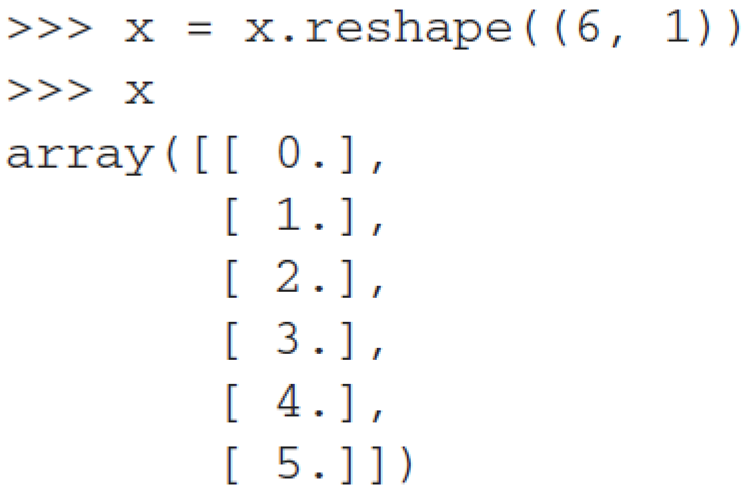
Iterates over the rows of x …

… and over the columns of y.

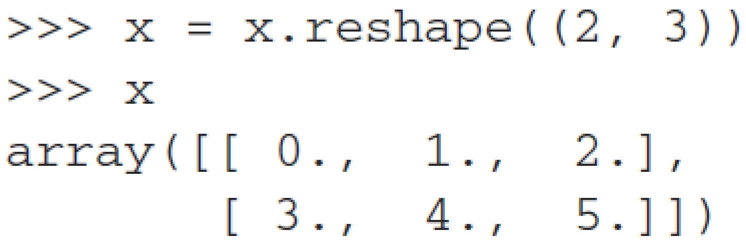
Tensor operations: Reshaping



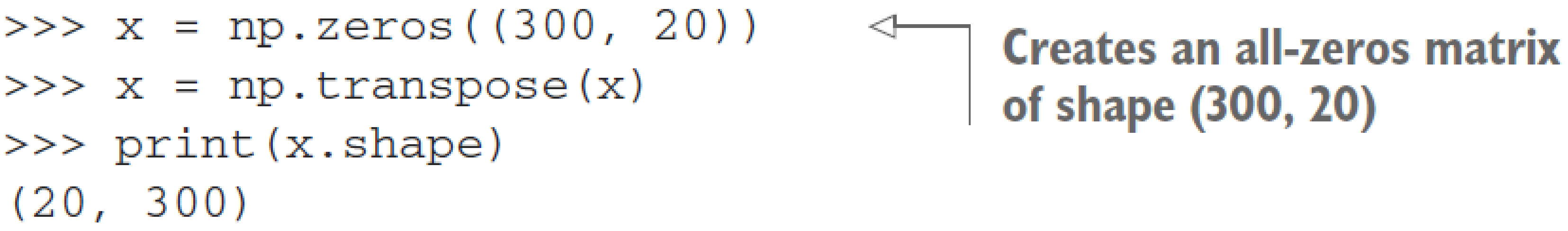
```
>>> x = np.array([[0., 1.],
                  [2., 3.],
                  [4., 5.]])
>>> print(x.shape)
(3, 2)
```



```
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
```



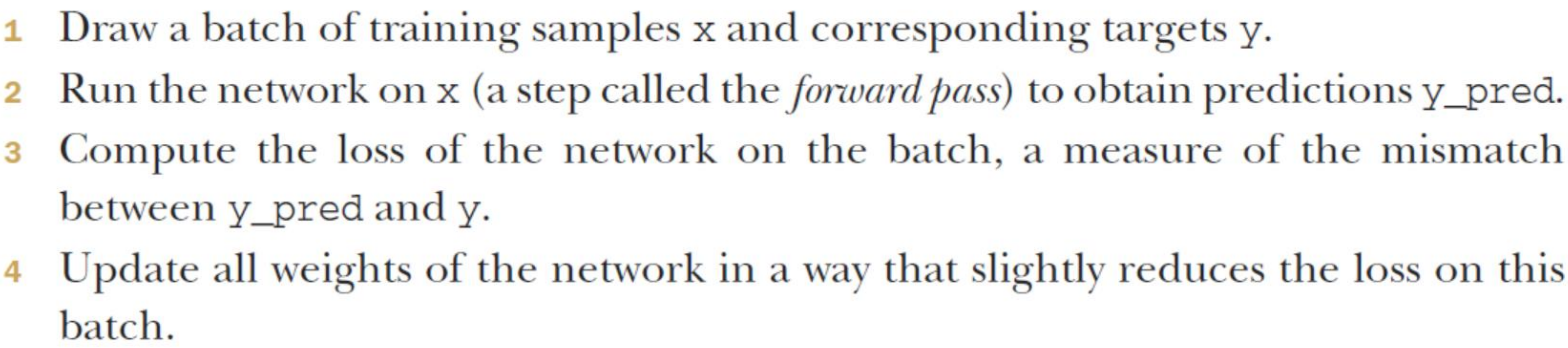
```
>>> x = x.reshape((2, 3))
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
```



```
>>> x = np.zeros((300, 20))
>>> x = np.transpose(x)
>>> print(x.shape)
(20, 300)
```

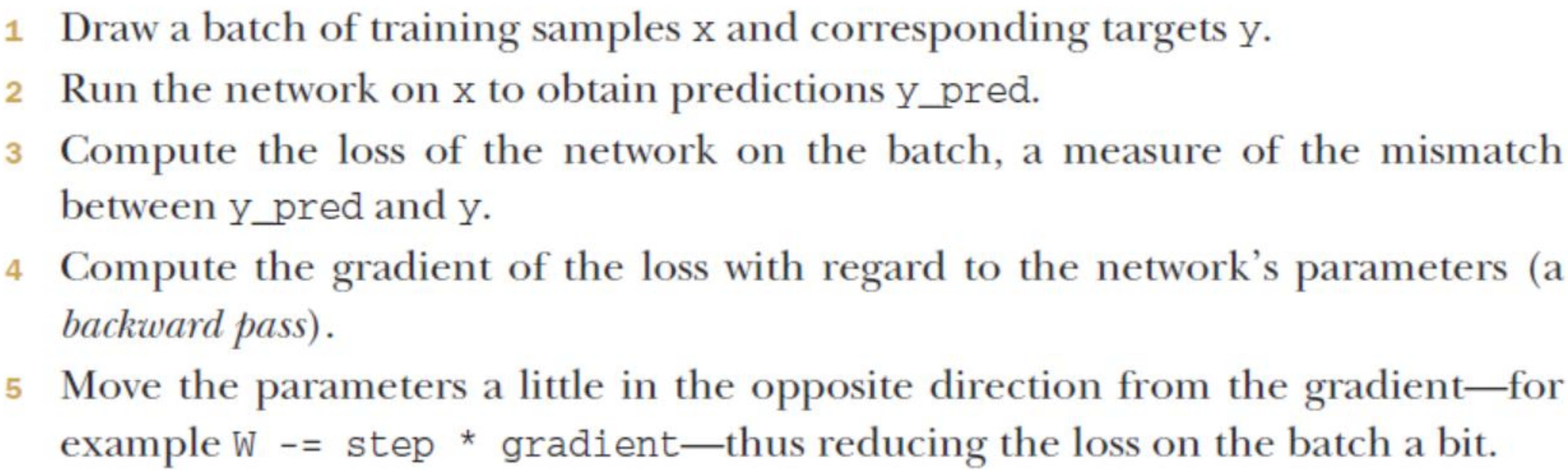
Creates an all-zeros matrix of shape (300, 20)

Training Loop



1. Draw a batch of training samples `x` and corresponding targets `y`.
2. Run the network on `x` (a step called the *forward pass*) to obtain predictions `y_pred`.
3. Compute the loss of the network on the batch, a measure of the mismatch between `y_pred` and `y`.
4. Update all weights of the network in a way that slightly reduces the loss on this batch.

Training Loop



1. Draw a batch of training samples `x` and corresponding targets `y`.
2. Run the network on `x` to obtain predictions `y_pred`.
3. Compute the loss of the network on the batch, a measure of the mismatch between `y_pred` and `y`.
4. Compute the gradient of the loss with regard to the network's parameters (a *backward pass*).
5. Move the parameters a little in the opposite direction from the gradient—for example `W -= step * gradient`—thus reducing the loss on the batch a bit.

mini-batch SGD vs *true SGD vs batch SGD*

SGD with momentum

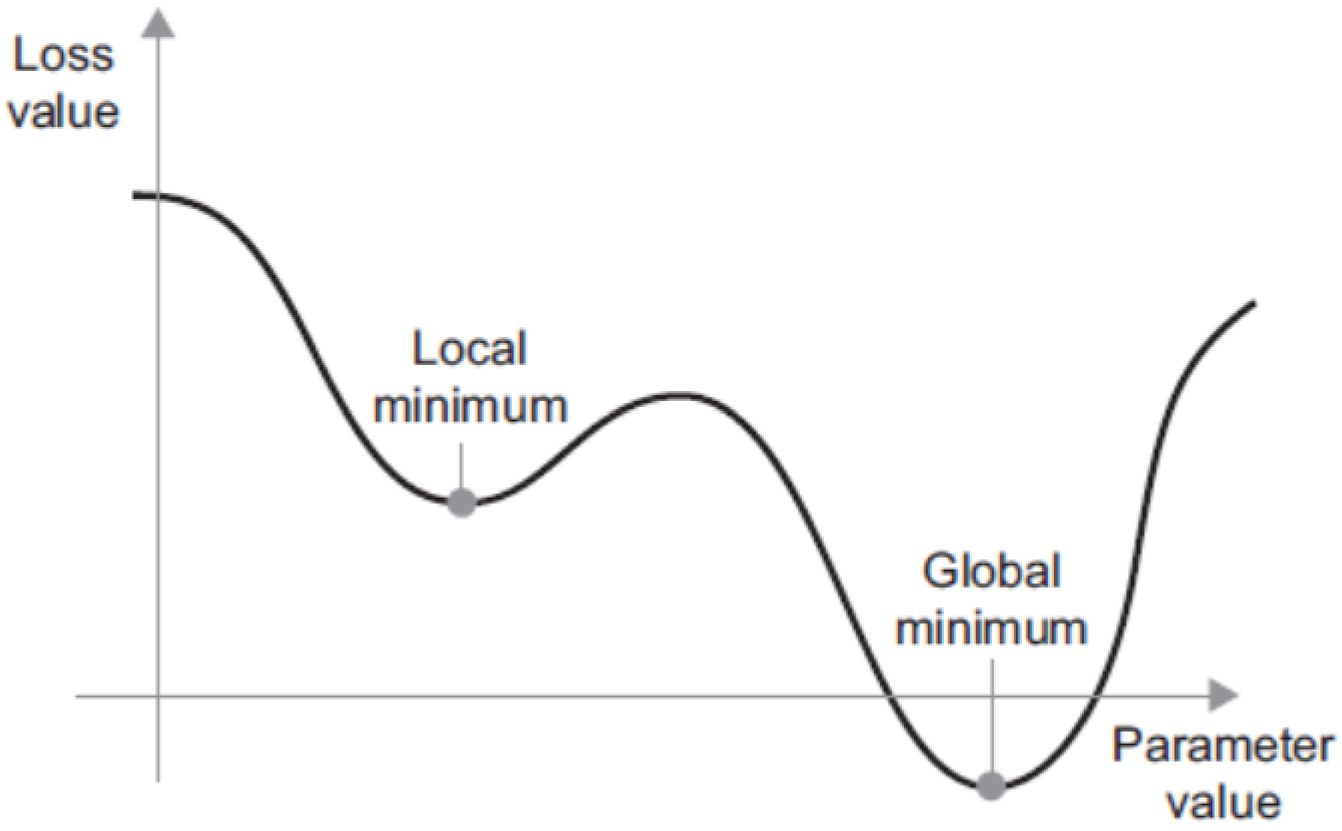
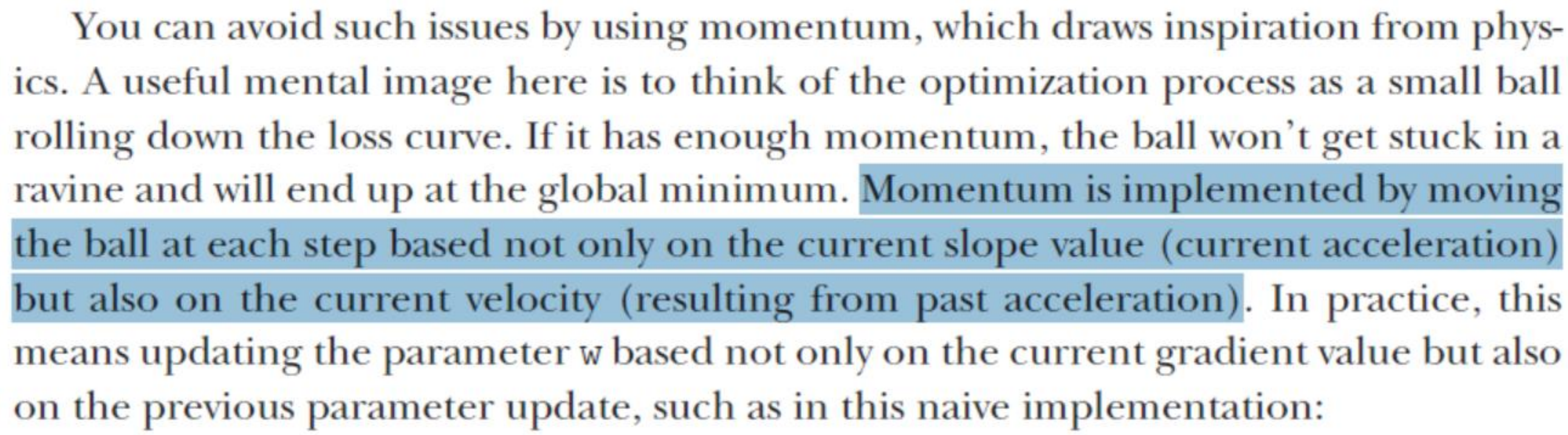


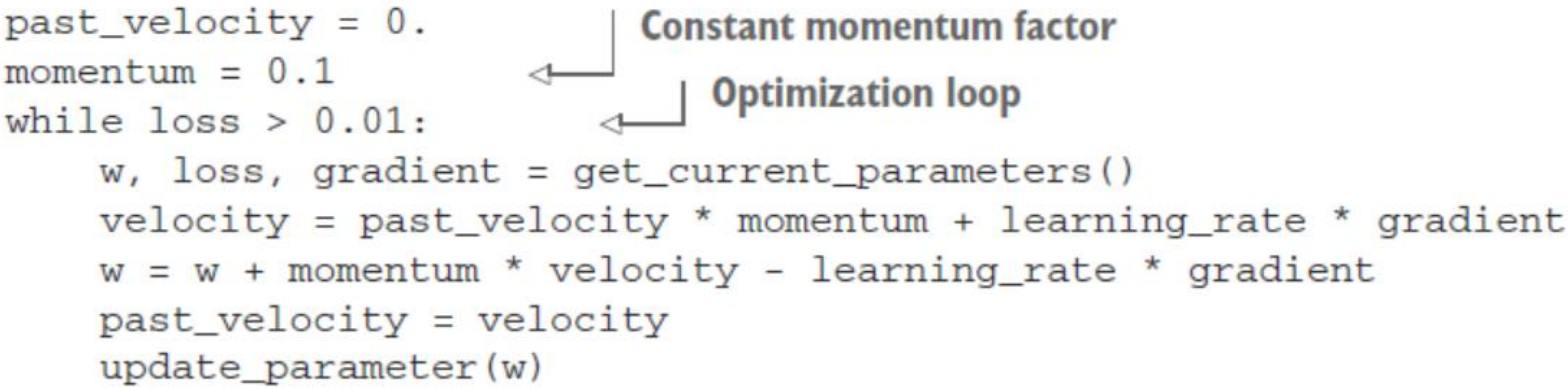
Figure 2.13 **A local minimum and a global minimum**

SGD with momentum



You can avoid such issues by using momentum, which draws inspiration from physics. A useful mental image here is to think of the optimization process as a small ball rolling down the loss curve. If it has enough momentum, the ball won't get stuck in a ravine and will end up at the global minimum. Momentum is implemented by moving the ball at each step based not only on the current slope value (current acceleration) but also on the current velocity (resulting from past acceleration). In practice, this means updating the parameter `w` based not only on the current gradient value but also on the previous parameter update, such as in this naive implementation:

SGD with momentum



```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum + learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

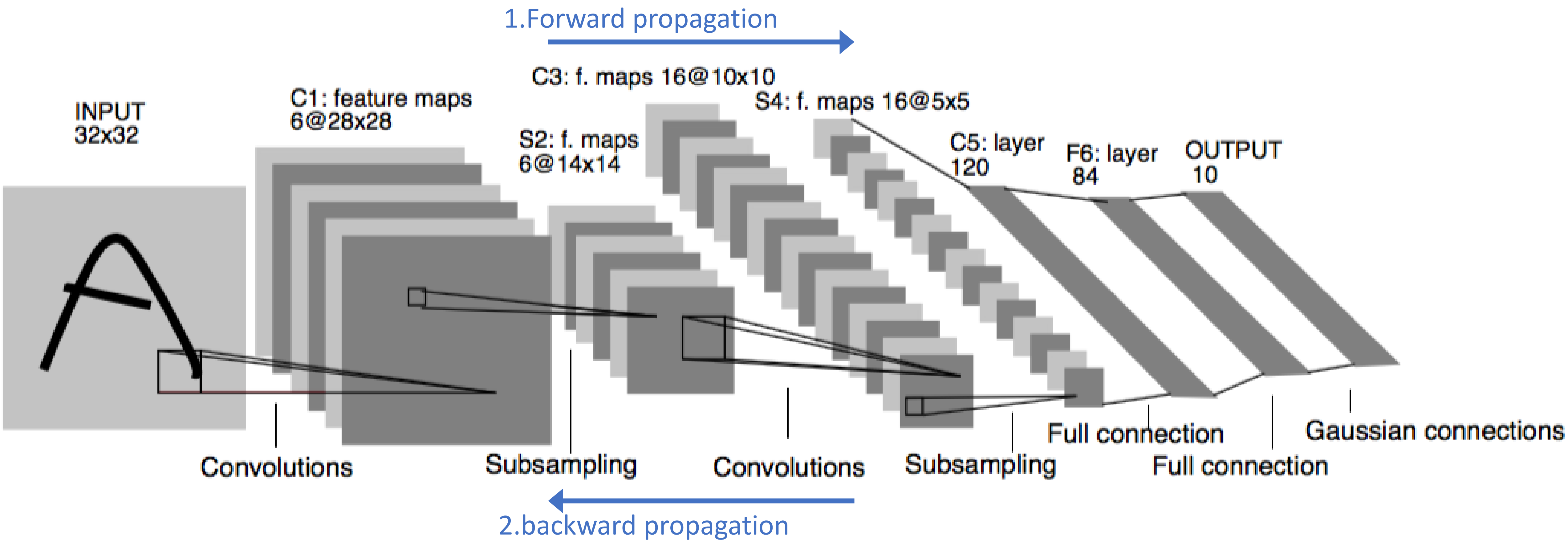
Constant momentum factor

Optimization loop

Backpropagation

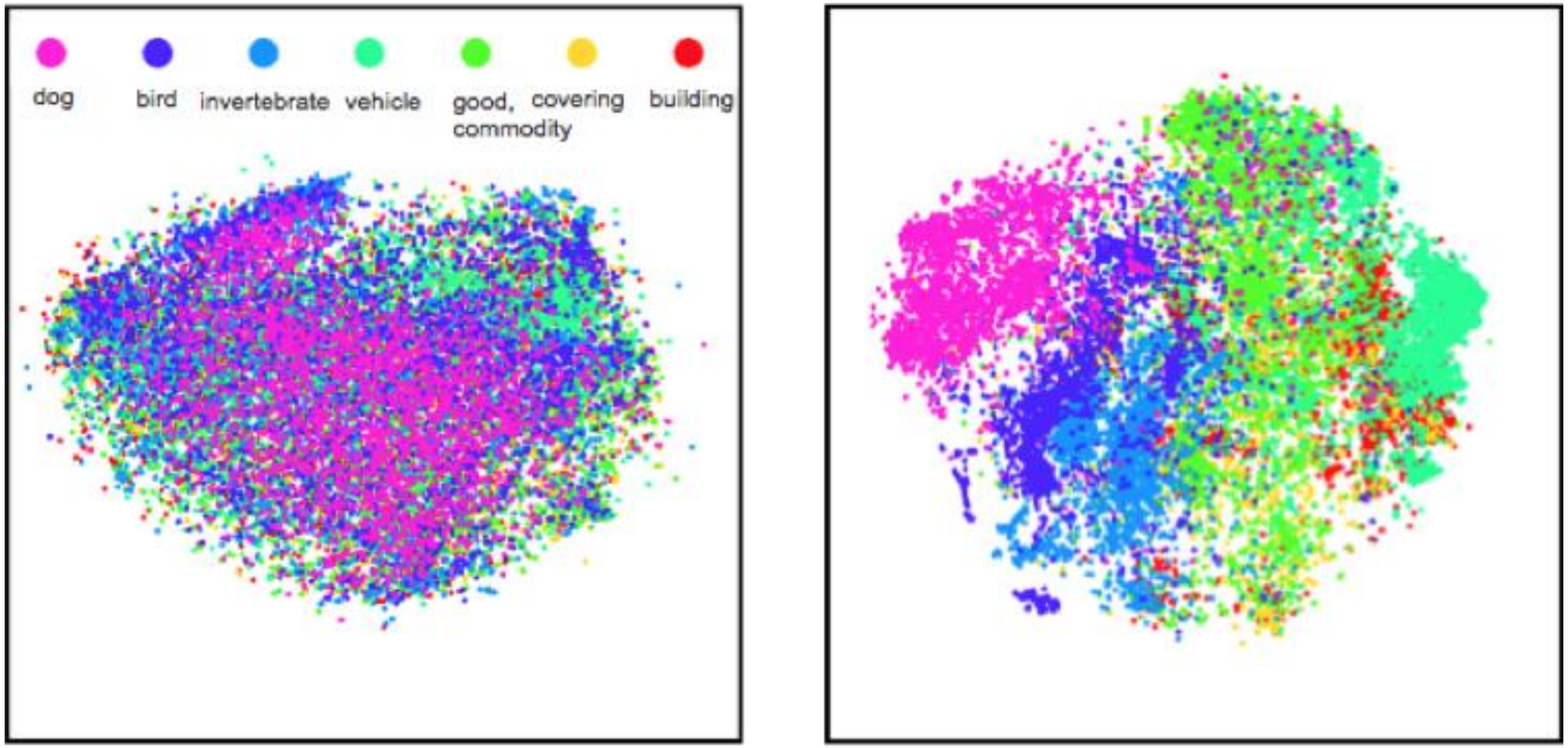
LeNet-5 [Lecun98]

- Lecun, et al. use the gradient-based learning method on MNIST dataset.



Why Deep Learning?

The Unreasonable Effectiveness of Deep Features-*Caffe descripts.*



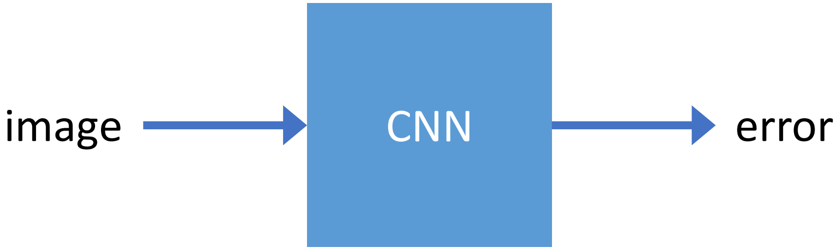
Low-level: Pool_1

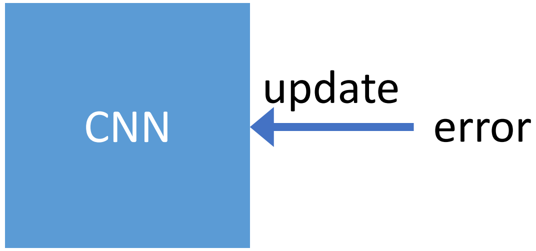
High-level: FC_6

Classes separate in the deep representations and transfer to many tasks.
[DeCAF] [Zeiler-Fergus]

Basic Layer in LeNet-5

- The input of forward propagation
  - An image
  - Pre-defined label
- The output of forward propagation
  - Loss error, square error, error rate, probability of each classes
- The input of backward propagation
  - Loss error
- Training methods
  - Gradient-based, dynamic programming

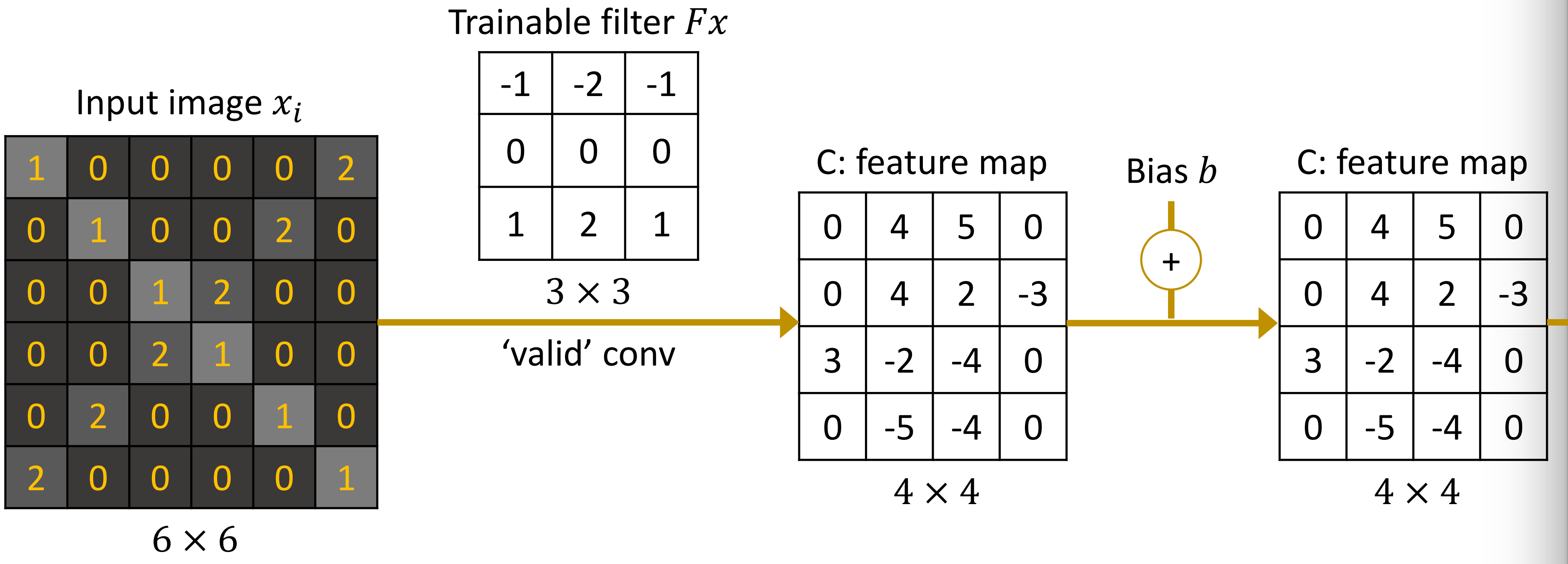




Basic Layer in LeNet-5

- Forward propagation for basic layer
  - Convolutional layer
  - Pooling
  - ReLU
  - Fully connected layer
  - Softmax

The convolutional layer



Input image x_i

1	0	0	0	0	2
0	1	0	0	2	0
0	0	1	2	0	0
0	0	2	1	0	0
0	2	0	0	1	0
2	0	0	0	0	1

6×6

Trainable filter Fx

-1	-2	-1
0	0	0
1	2	1

3×3

'valid' conv

C: feature map

0	4	5	0
0	4	2	-3
3	-2	-4	0
0	-5	-4	0

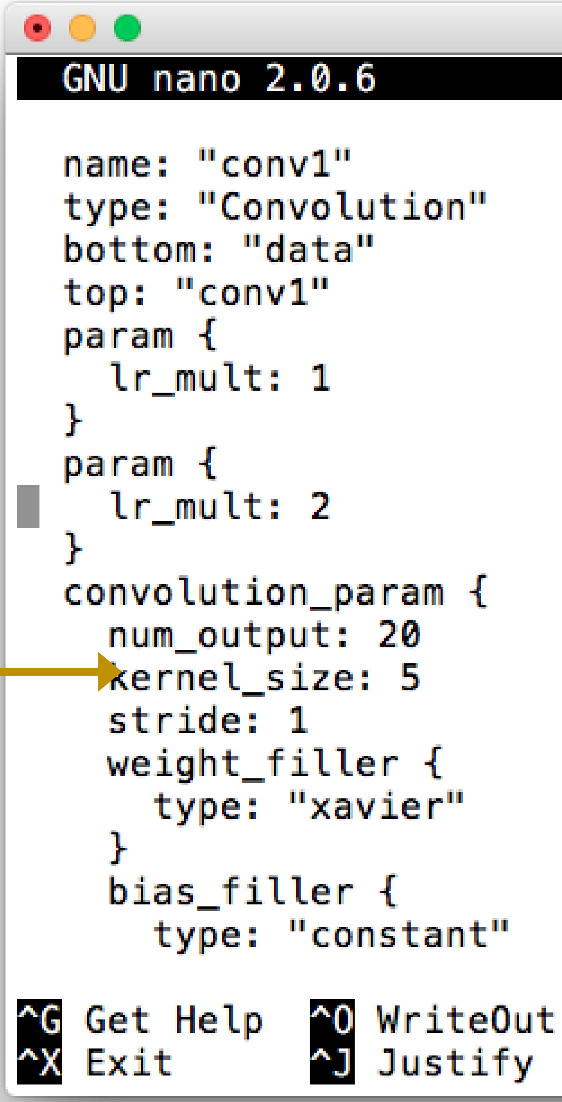
4×4

Bias b

C: feature map

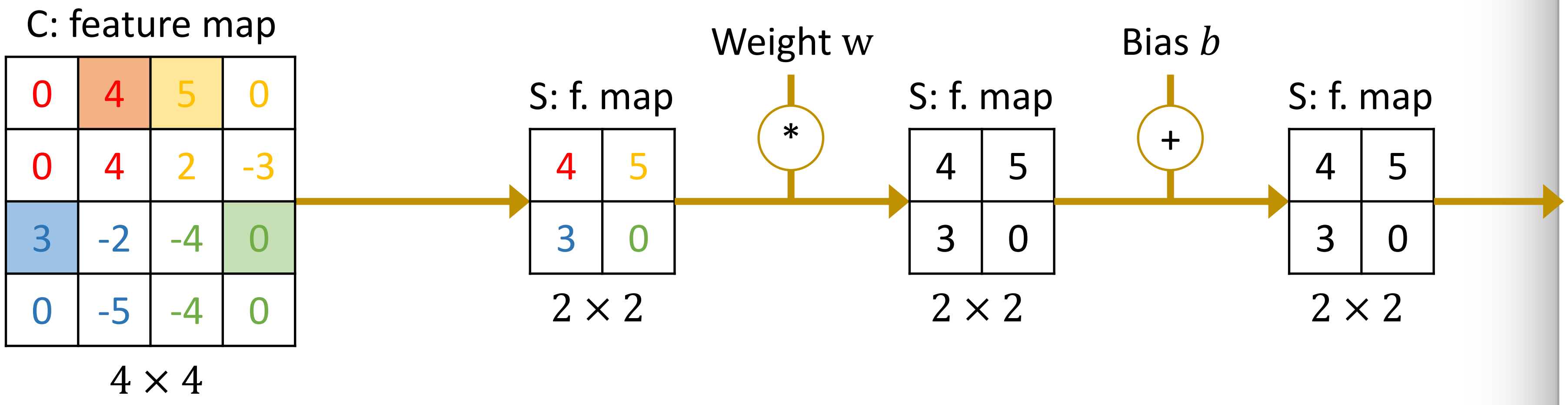
0	4	5	0
0	4	2	-3
3	-2	-4	0
0	-5	-4	0

4×4



```
name: "conv1"
type: "Convolution"
bottom: "data"
top: "conv1"
param {
  lr_mult: 1
}
param {
  lr_mult: 2
}
convolution_param {
  num_output: 20
  kernel_size: 5
  stride: 1
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
```

Pooling Layer



C: feature map

0	4	5	0
0	4	2	-3
3	-2	-4	0
0	-5	-4	0

4×4

S: f. map

4	5
3	0

2×2

Weight w

S: f. map

4	5
3	0

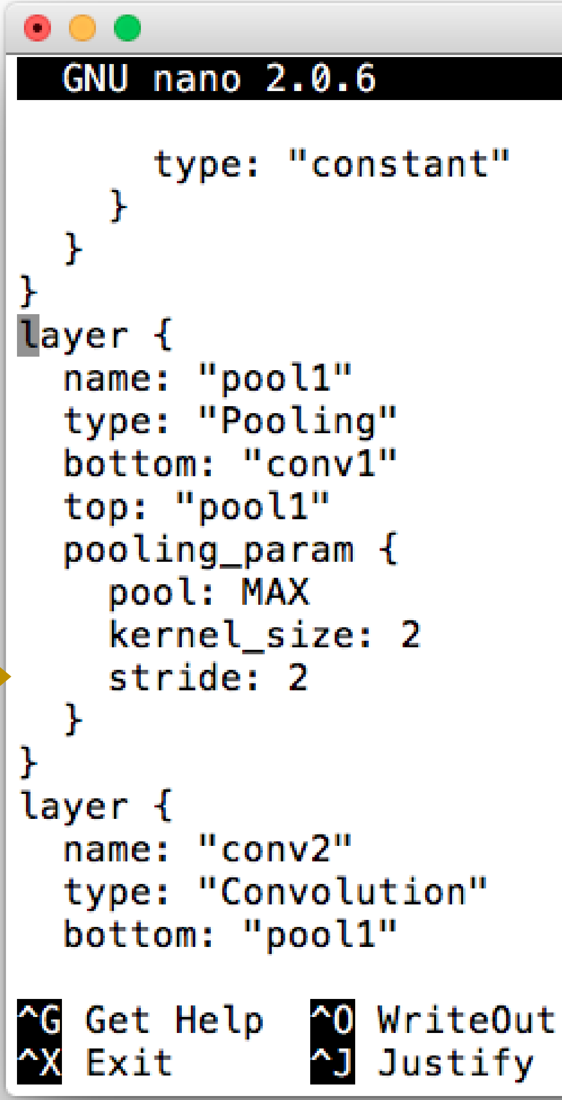
2×2

Bias b

S: f. map

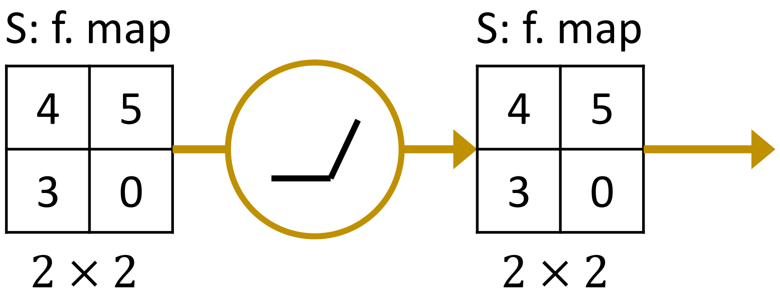
4	5
3	0

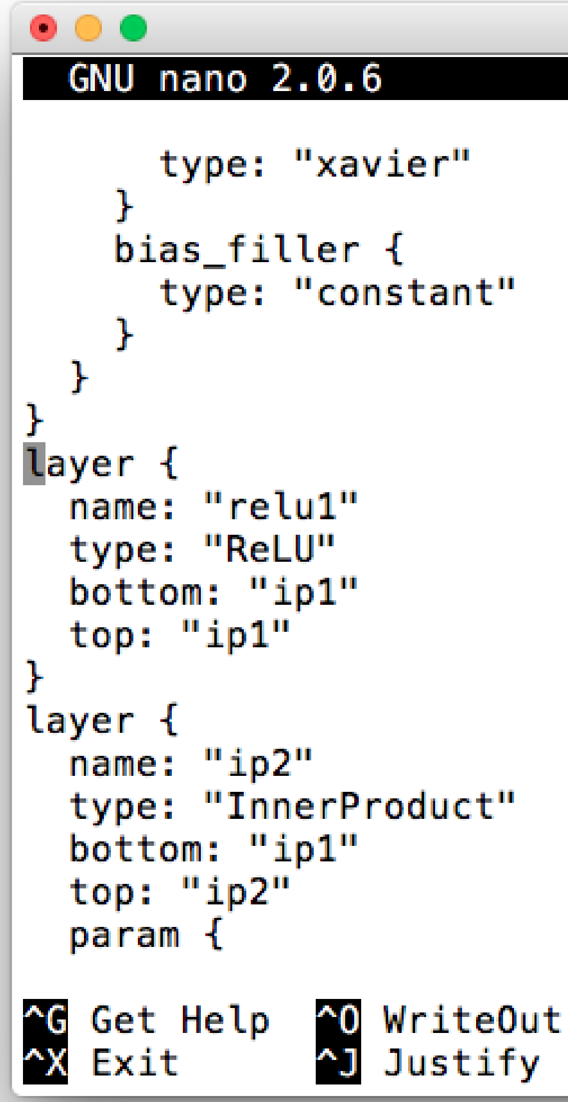
2×2



```
        type: "constant"
      }
    }
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
```

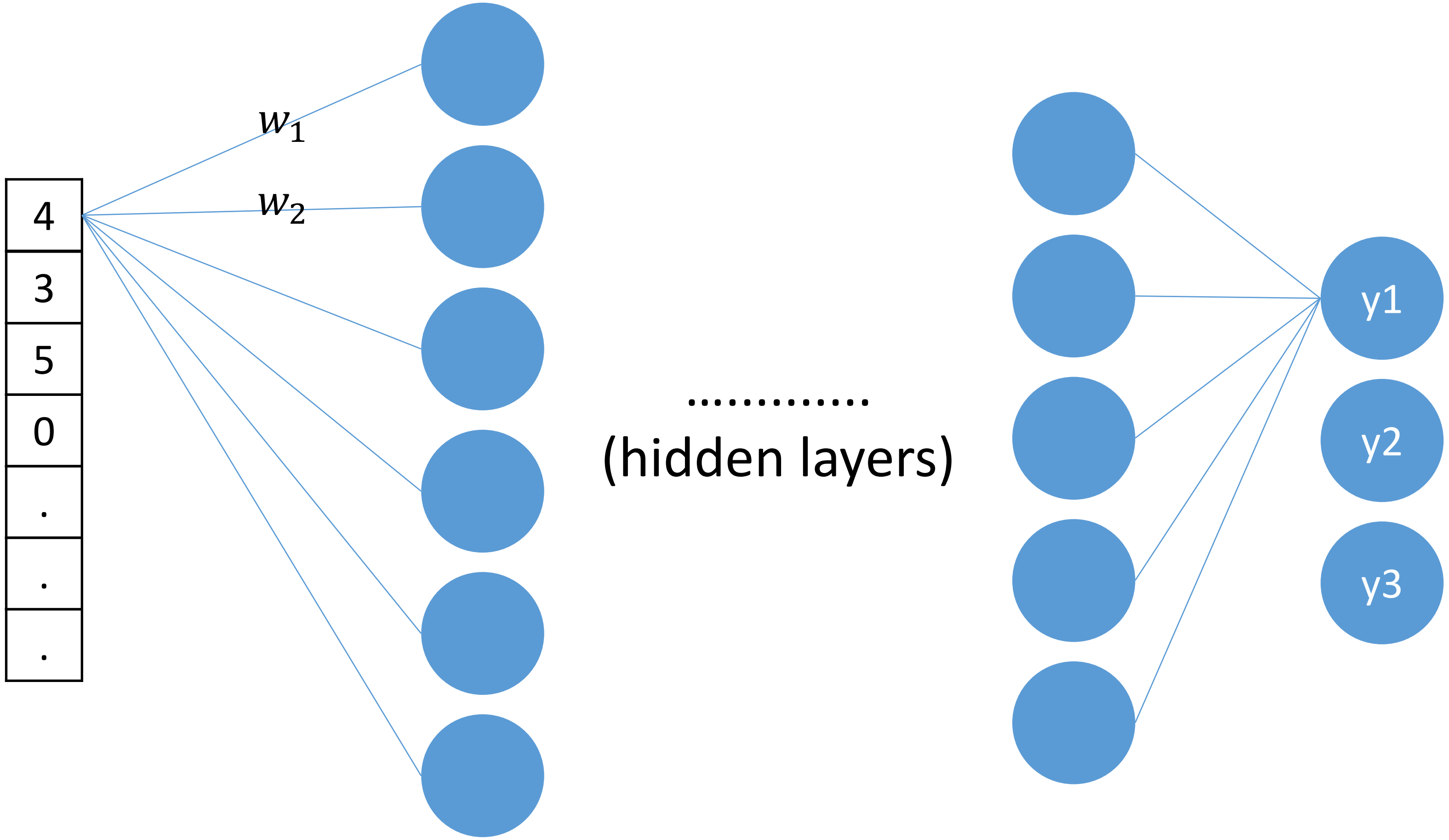
ReLU (Rectified Linear Units)

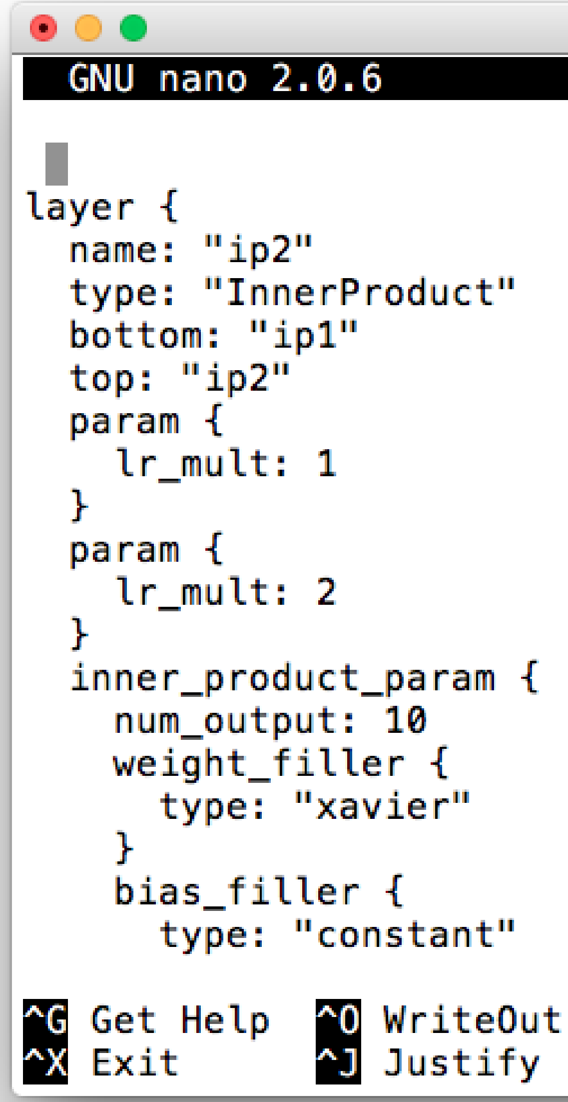




```
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
```

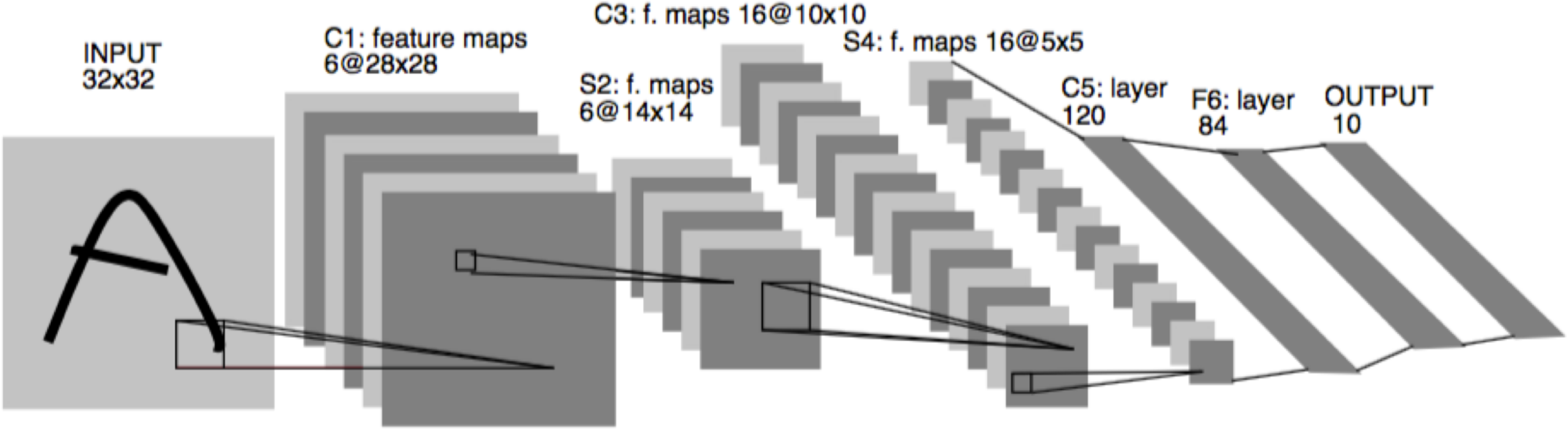
Fully Connected Layer





```
  GNU nano 2.0.6

layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"

^G Get Help  ^O WriteOut
^X Exit      ^J Justify
```

How Many Trainable Parameters in LeNet-5



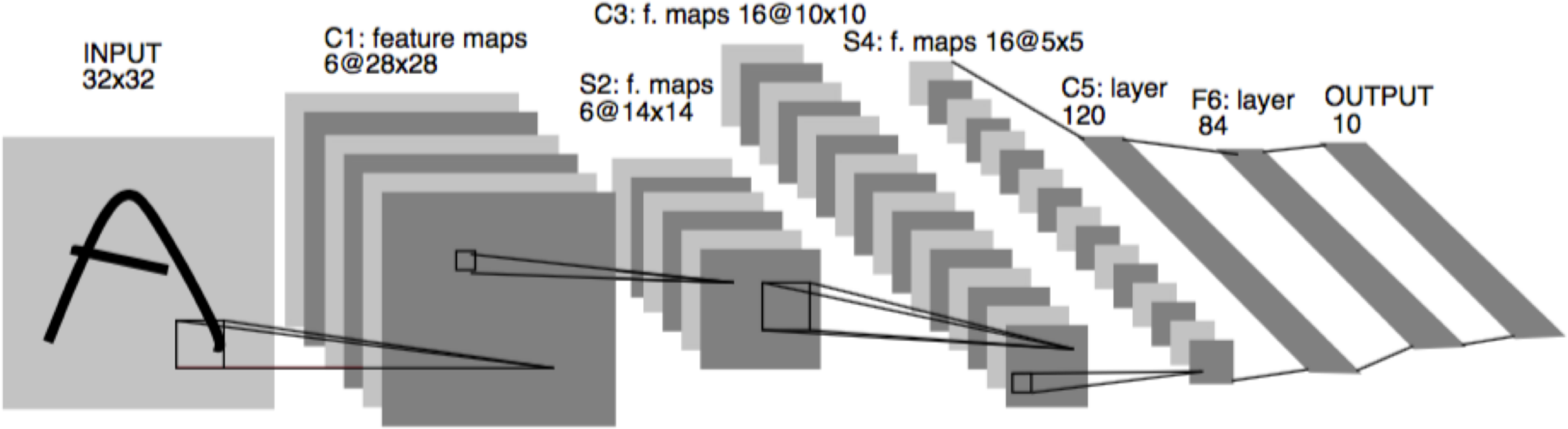
C1: $6|_{kernels} \times (5 \times 5 + 1)|_{size} = 156$ parameters

S2: $6|_{f.maps} \times \left(1|_{weight} + 1|_{bias}\right) = 12$ parameters

C3: $16|_{f.maps} \times (6 \times 5 \times 5 + 1|_{bias}) = 2416$ parameters

S4: $16|_{f.maps} \times (1|_{weight} + 1|_{bias}) = 32$ parameters

How Many Trainable Parameters in LeNet-5



C5: $\left(16|_{f.maps} \times (5 \times 5)|_{size} + 1|_{bias}\right) * 120|_{neurons} = 48120$

F6: $120|_{neurons} \times 85|_{neurons} = 10200$

OUT: $84|_{neurons} \times 11|_{neurons} = 924$

There are about 61,860 trainable parameters

Gradient-Based Training

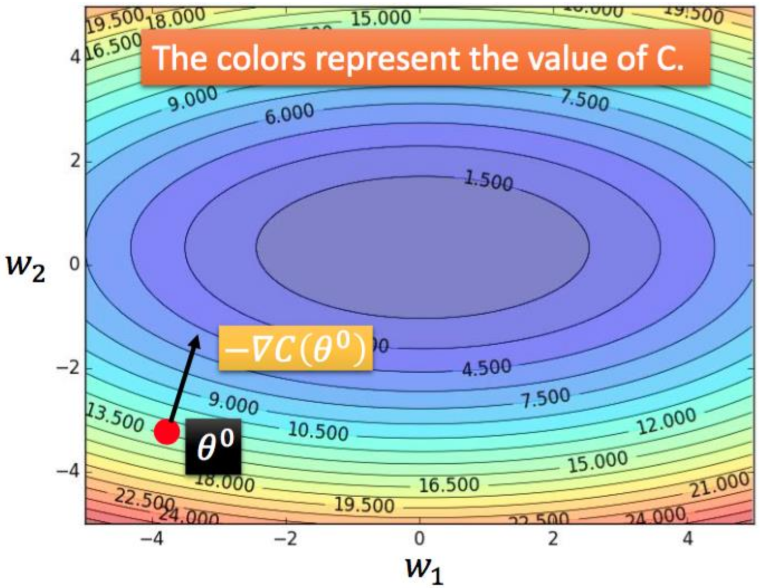
- We want to find parameters W to minimize an error

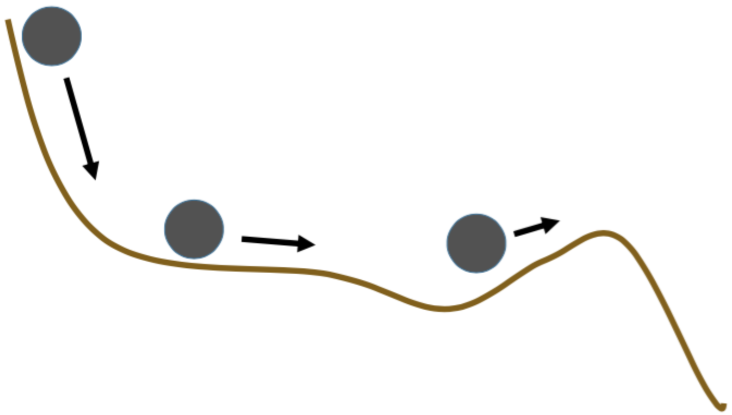
$$E(f(X,W),Z)$$

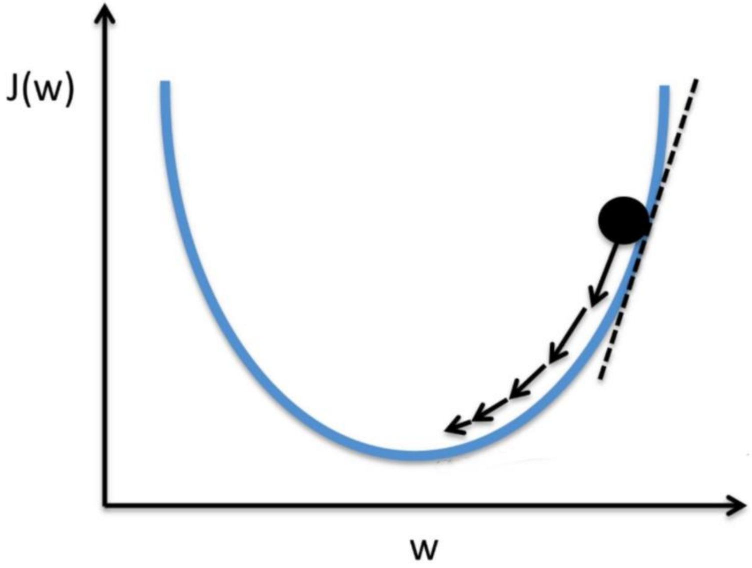
- For this, we will do iterative gradient descent

$$W(t) = W(t-1) - \eta \frac{\partial E}{\partial W}(t)$$

X: input image
W: weights
Z: pre-defined labels







Basic Layer in LeNet-5

- Back-propagation for basic layer
  - Fully connected layer
  - ReLU
  - Pooling
  - Convolutional layer

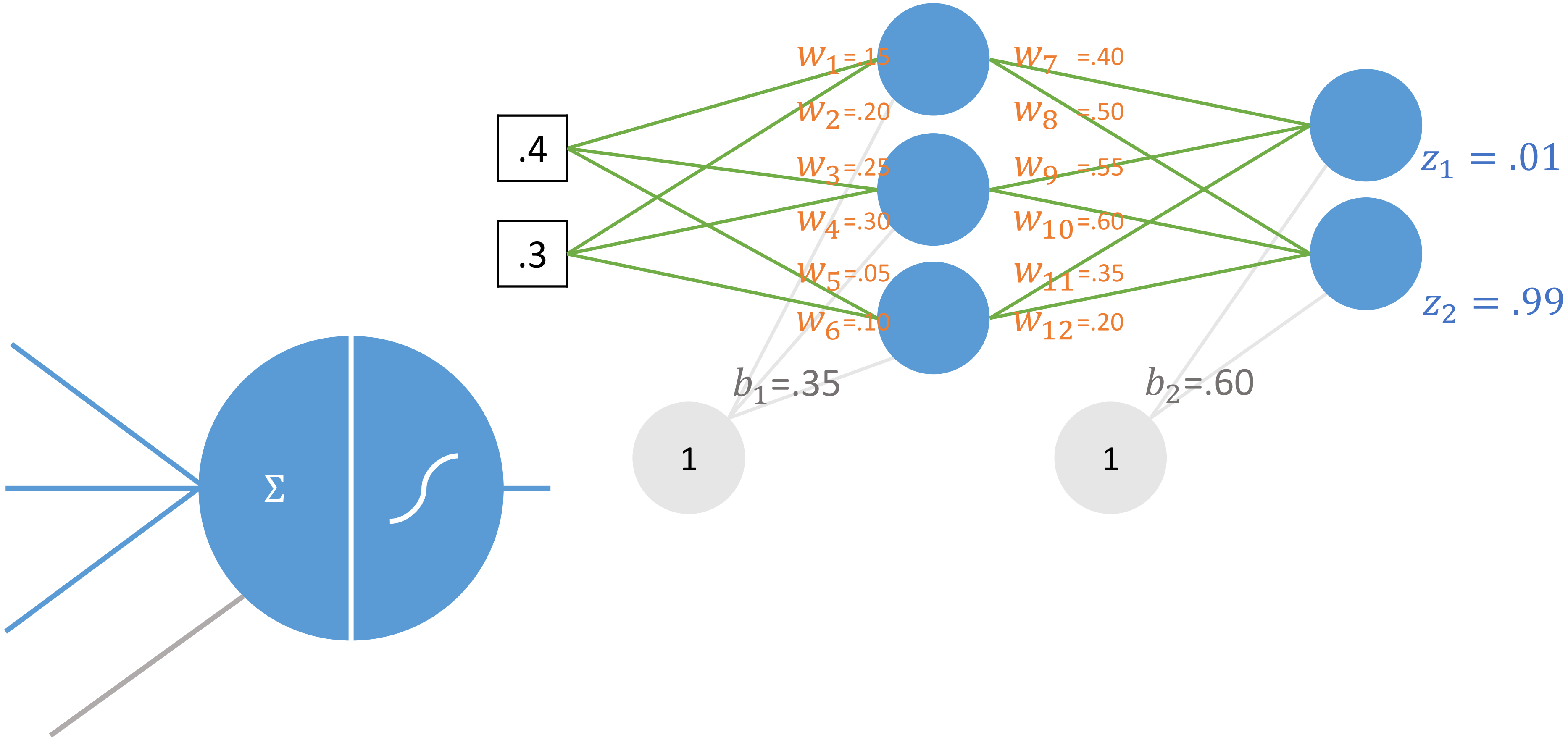
Fully Connected Layer

$$\underline{\text{Weight vector}} = W = \{w_i\}_{i=1\ldots N}$$
$$= w_{i,j}^{c,k}$$

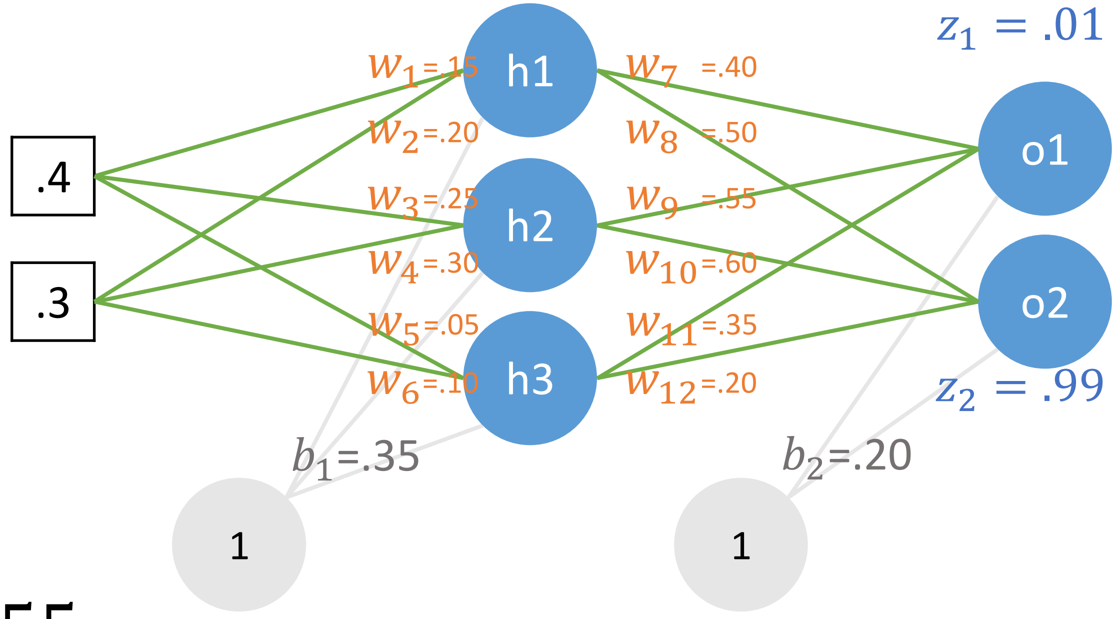
$$\text{Input scales} = X = \{x_j\}_{j=1\ldots M}$$

$$\text{Labels} = Z = \{z_k\}_{k=1\ldots C}$$

$$\text{Loss} = \mathrm{F}(\mathrm{F}(\mathrm{F}(X; w_1); w_2); w_N)$$



Fully Connected Layer



- Forward pass-hidden layer

$$net_{h1} = w_1x_1 + w_2x_2 + b_1$$

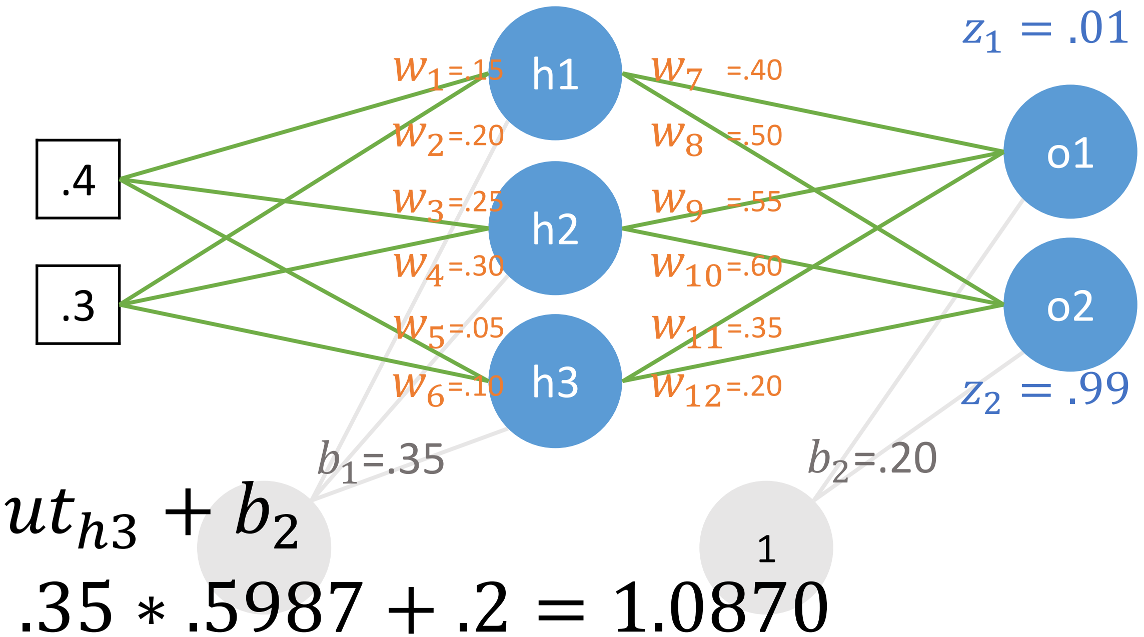
$$= .15 * .4 + .20 * .3 + .35 = 1.55$$

- Using the activation function and we got the output of hidden layer

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-1.55}} = .8249$$

$$out_{h2} = .6318, out_{h3} = .5987$$

Fully Connected Layer



- Forward pass-output layer

$$net_{o1} = w_7 out_{h1} + w_9 out_{h2} + w_{11} out_{h3} + b_2$$
$$= .40 * .8249 + .55 * .6318 + .35 * .5987 + .2 = 1.0870$$

- Using the activation function and we got the output of hidden layer

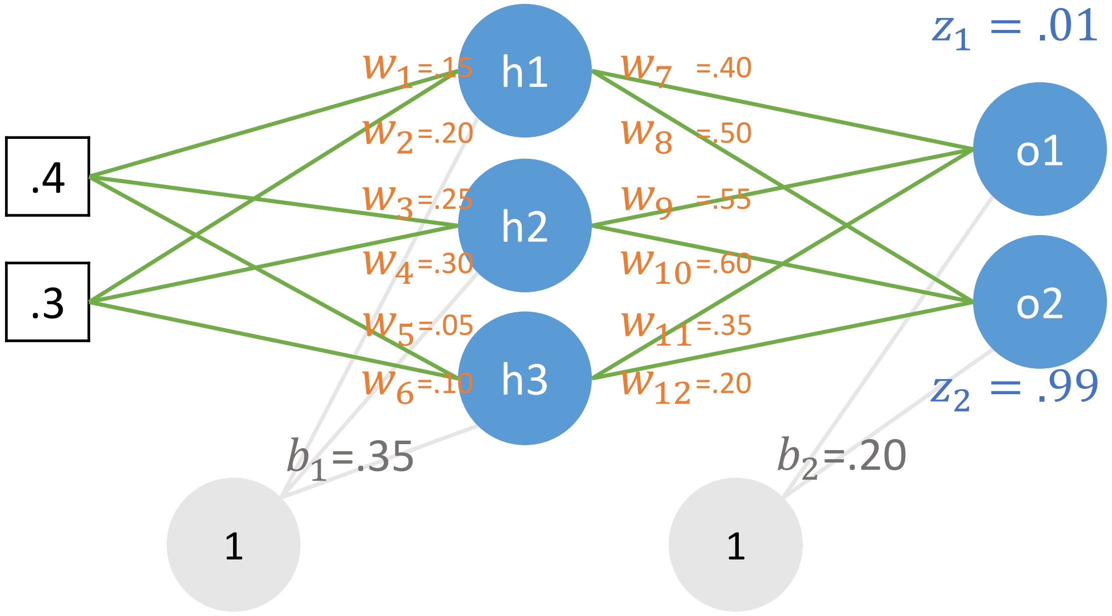
$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.0870}} = .7478$$
$$out_{o2} = .7524$$

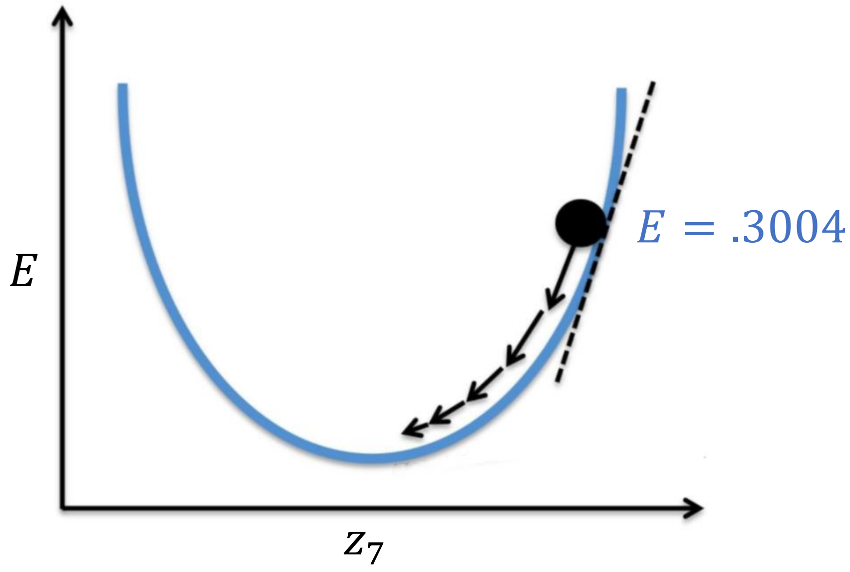
- And we got square error $E_{total} = \sum \frac{1}{2}(target - output)^2$

$$E_{total} = \frac{1}{2}(.01 - .7478)^2 + \frac{1}{2}(.99 - .7524)^2 = .3004$$

The Backwards Pass

- The goal with back-propagation is to update each of the weights in the network so that they cause the actual output to be closer the target output.

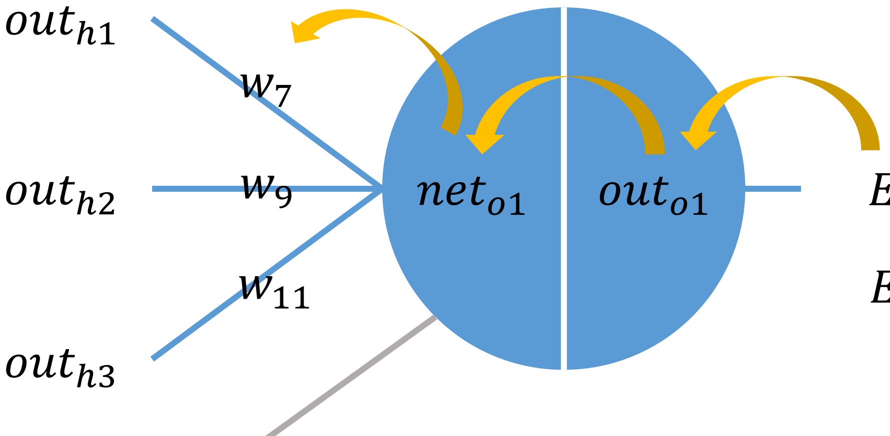




The Backwards Pass

- Consider w_7. We want to know how much a change in w_7, affects the total error, $\frac{\partial E_{total}}{\partial w_7}$.
- The gradient with respect to w_7 by applying the chain rule, we know that

$$\frac{\partial E_{total}}{\partial w_7} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_7}$$



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

$$E_{total} = E_{o1} + E_{o2}$$

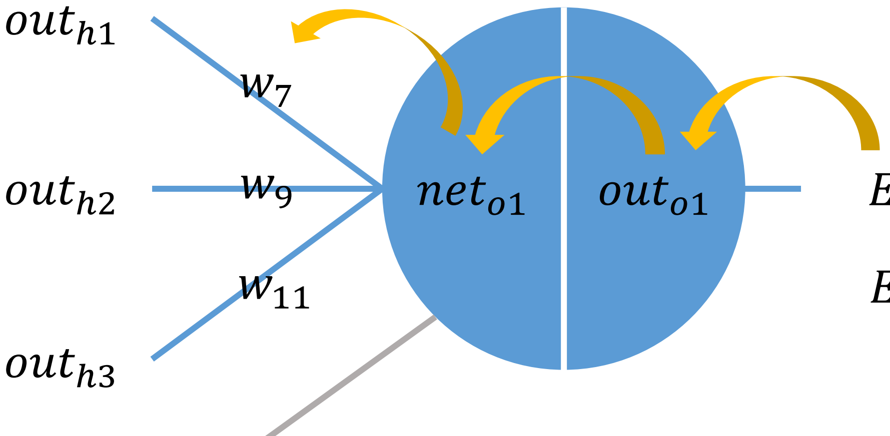
The Backwards Pass

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

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1}) * -1 + 0$$

$$= -(target_{o1} - out_{o1}) = -(.01 - .7478) = .7378$$

$$\frac{\partial E_{total}}{\partial w_7} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_7}$$



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

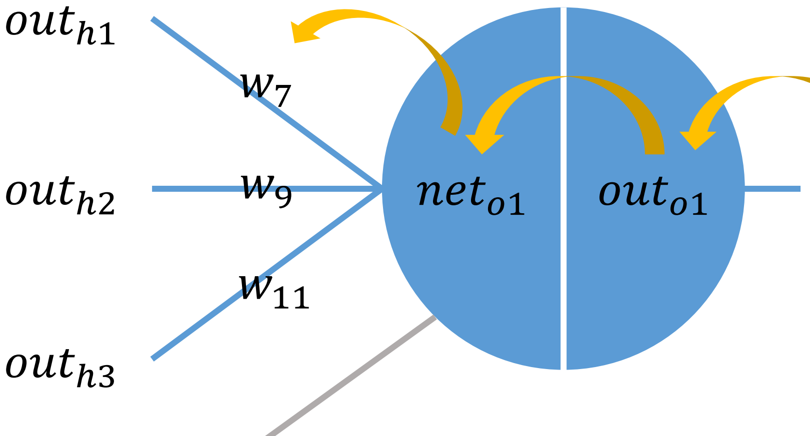
$$E_{total} = E_{o1} + E_{o2}$$

The Backwards Pass

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

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = .7478(1 - .7478) = .1886$$

$$\frac{\partial E_{total}}{\partial w_7} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_7}$$



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

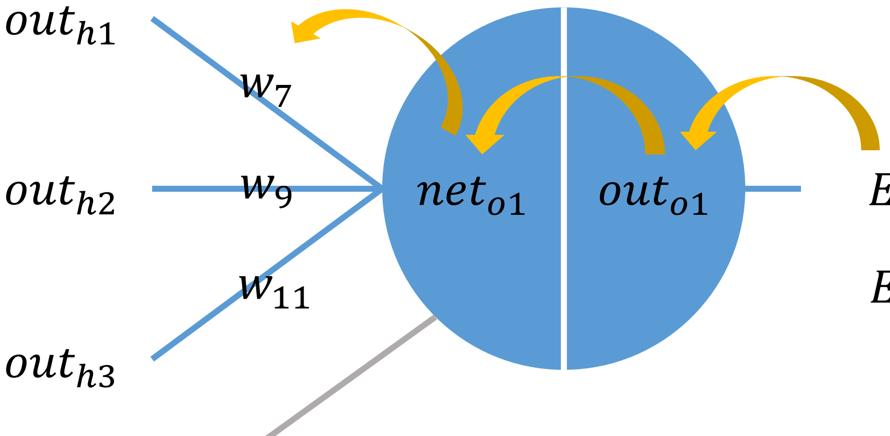
$$E_{total} = E_{o1} + E_{o2}$$

The Backwards Pass

$$net_{o1} = w_7 out_{h1} + w_9 out_{h2} + w_{11} out_{h3}$$

$$\frac{\partial net_{o1}}{\partial w_7} = out_{h1} + 0 + 0 = .8249$$

$$\frac{\partial E_{total}}{\partial w_7} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_7}$$



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

$$E_{total} = E_{o1} + E_{o2}$$

The Backwards Pass

$$\frac{\partial E_{total}}{\partial w_7} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_7}$$

$$\frac{\partial E_{total}}{\partial w_7} = .7378 * .1886 * .8249 = .1148$$

$$\frac{\partial E_{total}}{\partial w_7} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}$$

- The above form often called delta rule

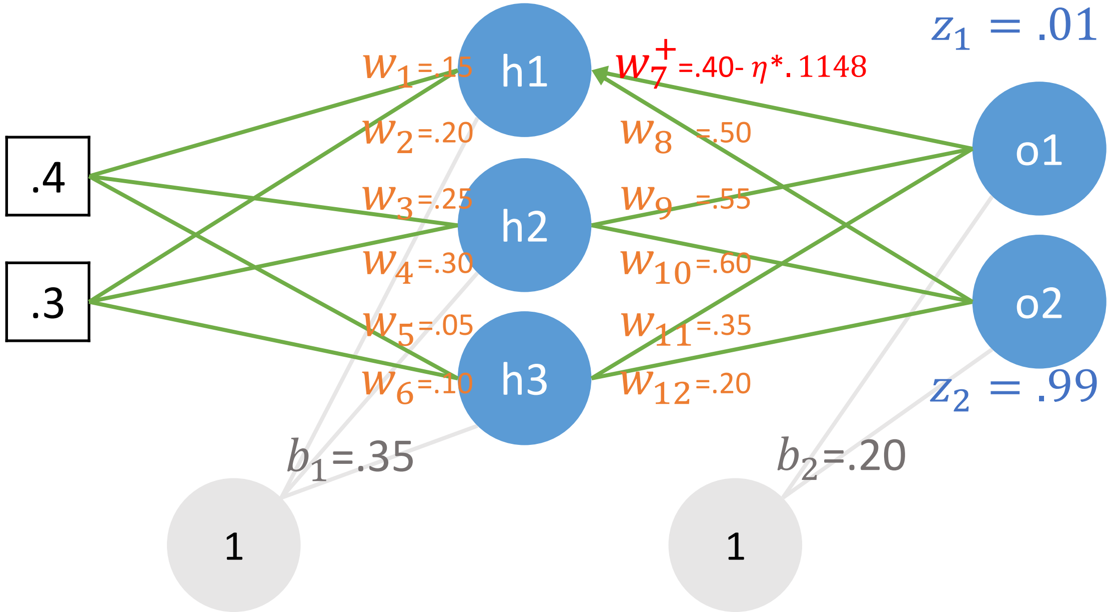
Update The Weight

- The weight w_7 can be updated

$$w_7^+ = w_7 - \eta \frac{\partial E_{total}}{\partial w_7}$$

  where η is the learning rate.

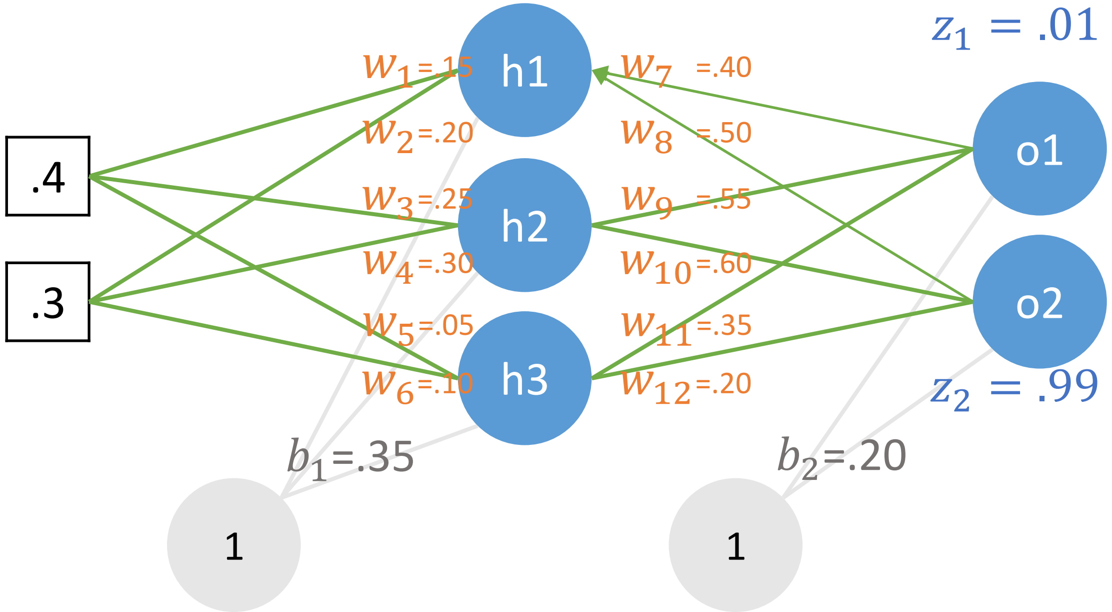
- So the others $w_8, w_9, w_{10}, w_{11}, w_{12}$



The Hidden Layer

- The weight w_1 can be updated

$$w_1^+ = w_1 + \eta \frac{\partial E_{total}}{\partial w_1}$$

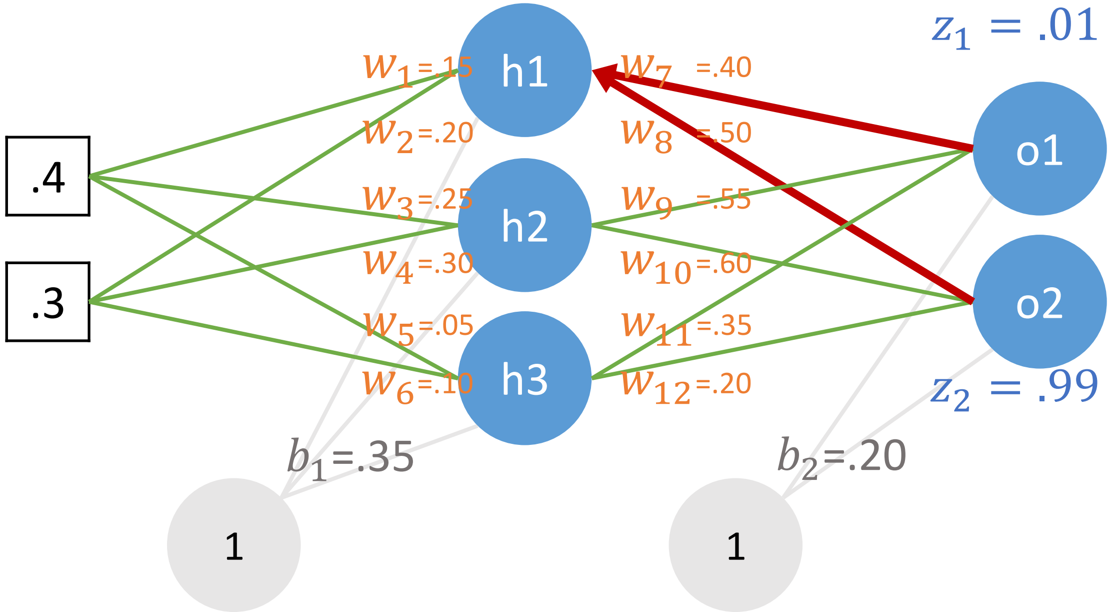


The Hidden Layer

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

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

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



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

The Hidden Layer

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

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

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} \frac{\partial out_{o1}}{\partial net_{o1}} = .7378 * .1886 = .1391$$

$$where\ \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} = -(target_{o1} - out_{o1})$$

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

The Hidden Layer

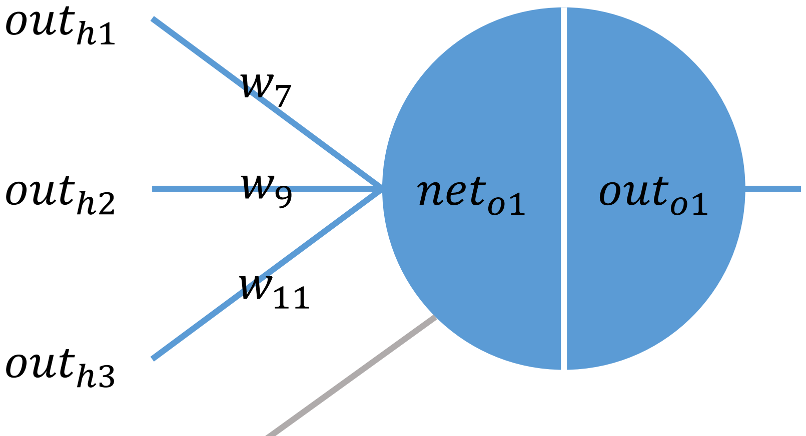
$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

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

$$net_{o1} = w_7 out_{h1} + w_9 out_{h2} + w_{11} out_{h3}$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_7 = .40$$

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



The Hidden Layer

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

$$\frac{\partial E_{o1}}{\partial out_{h1}} = .1391 * .40 = .0556$$

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -.0443 * .50 = -.0221$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = .0556 - .0221 = .0335$$

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

The Hidden Layer

- Now we have $\frac{\partial E_{total}}{\partial out_{h1}}$ and we need to figure out $\frac{\partial out_{h1}}{\partial net_{h1}}$ and $\frac{\partial net_{h1}}{\partial w_1}$

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

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

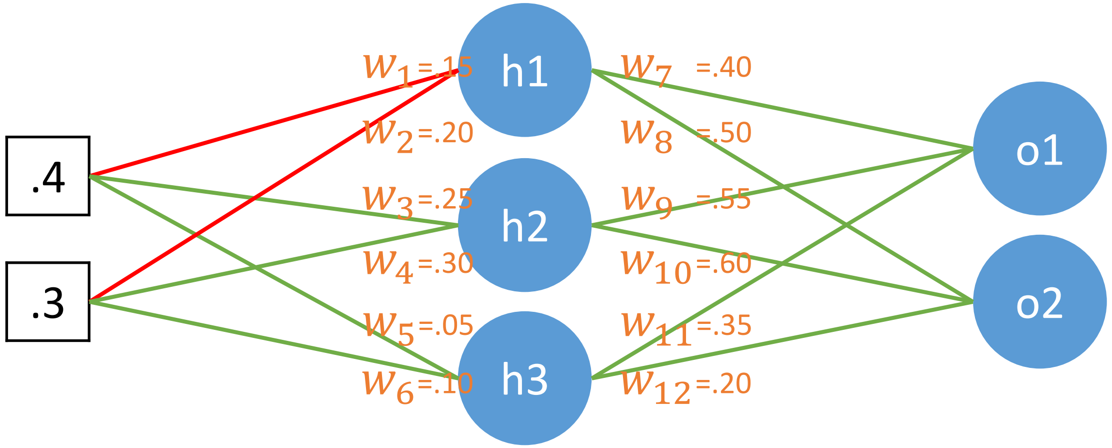
$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = .8249(1 - .8249) = .1444$$

The Hidden Layer

- Now we have $\frac{\partial E_{total}}{\partial out_{h1}}$ and $\frac{\partial out_{h1}}{\partial net_{h1}}$, we need to figure out $\frac{\partial net_{h1}}{\partial w_1}$

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

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



$$\frac{\partial net_{h1}}{\partial w_1} = x_1 = .4$$

The Hidden Layer

- Finally, we have $\frac{\partial E_{total}}{\partial w_1}$

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

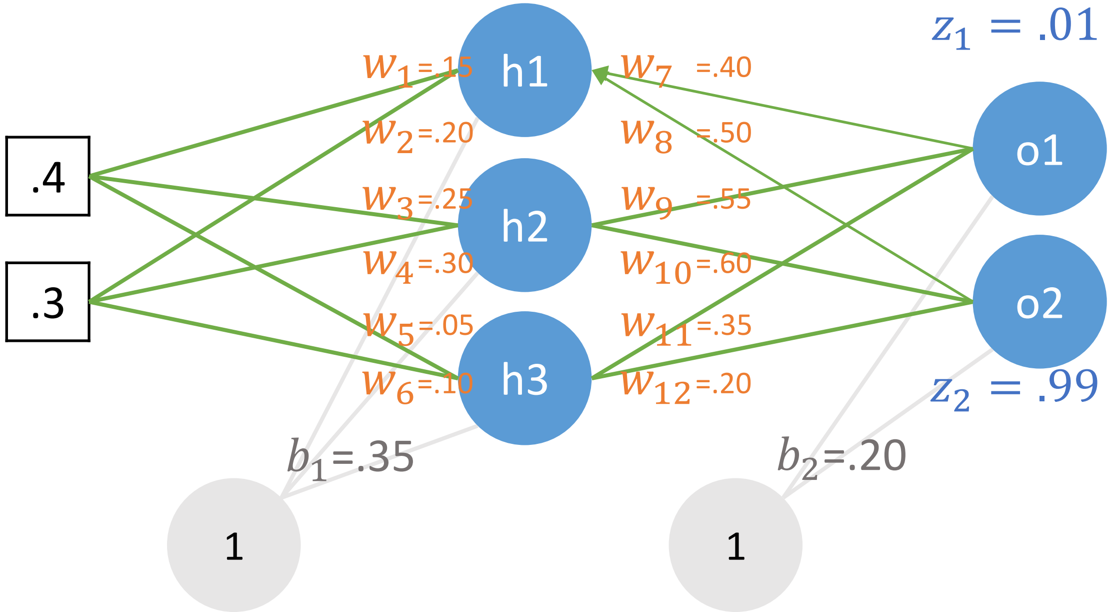
$$\frac{\partial E_{total}}{\partial w_1} = .0335 * .1444 * .4 = .0019$$

The Hidden Layer

- Thus the weight w_1 can be updated

$$w_1^+ = w_1 - \eta \frac{\partial E_{total}}{\partial w_1}$$

- Repeating this for the rest of the weights w_2, w_3, w_4, w_5, w_6



Tools

NVIDIA CUDA

TensorFlow

theano

Caffe

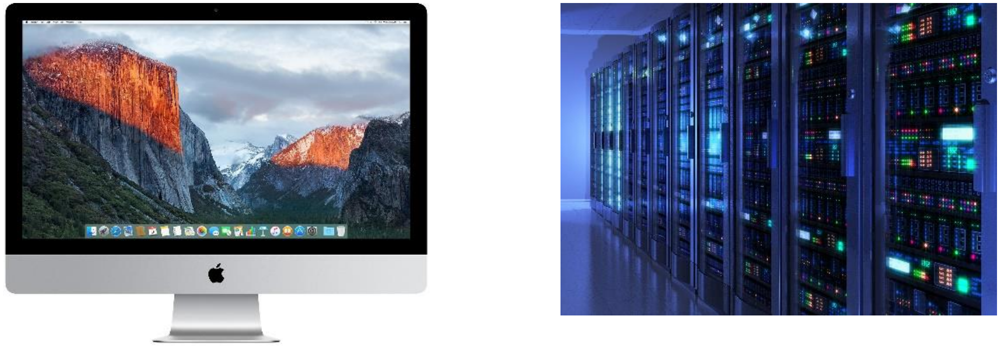
Microsoft CNTK

torch

ML

What is Caffe?

- **Open framework, models, and worked examples** for deep learning
- Pure C++ / CUDA library for deep learning
- Command line, Python, MATLAB interfaces
- Seamless switch between CPU and GPU



Caffeinated Companies

Chelsea Kohler

Sunday night splurge

Image may contain: pizza, food

Manohar Paluri

With my college buddies in my favorite place - ready for a great weekend!

Image may contain: tree, sky, outdoor

Peter Cottle

We finally made it :)

Image may contain: two people, smiling, sunglasses, sky, outdoor, water

Automatic Alt Text
recognize photo content
for accessibility

Your Memories on Facebook

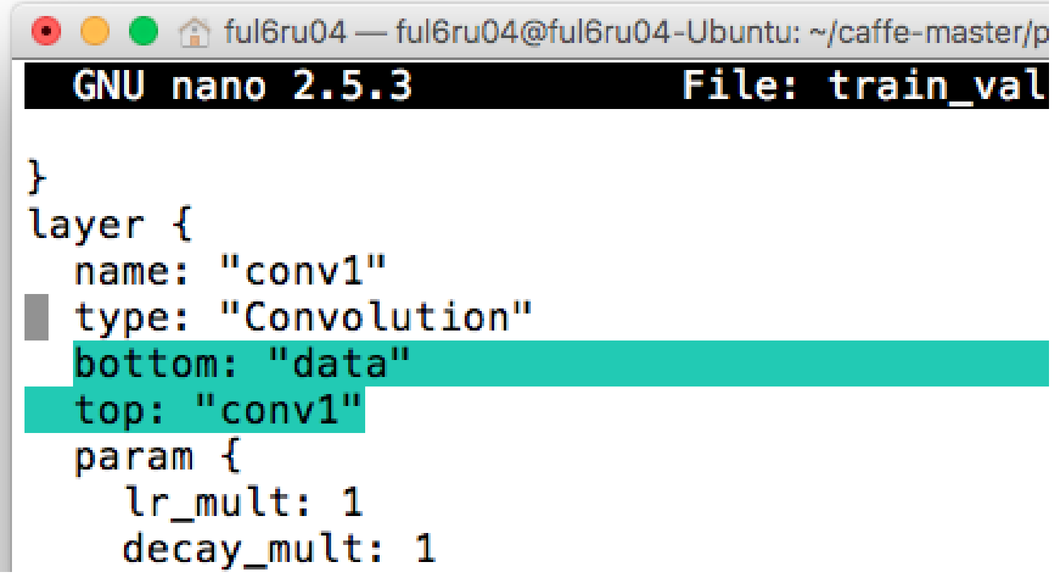
Allan, we care about you and the memories you share here. We thought you'd like to look back on this post from 2 years ago.

On This Day
highlight content

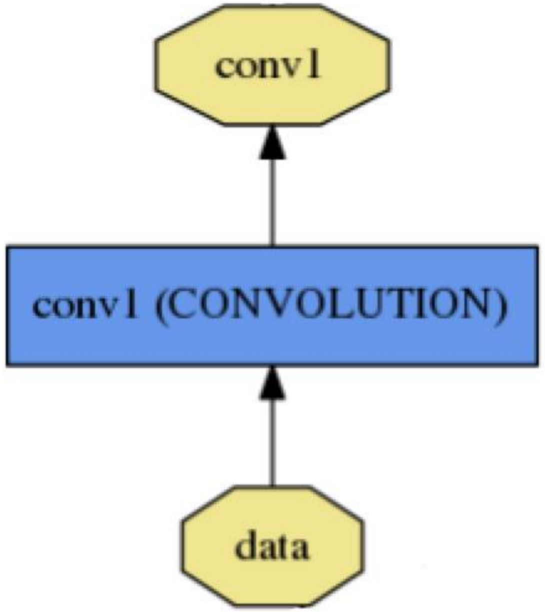
News Image Recommendation
select and crop images for news

Blob is Everything

- Blobs are N-D arrays for storing and communicating information.
    - hold data, derivatives, and parameters
    - lazily allocate memory
    - shuttle between CPU and GPU

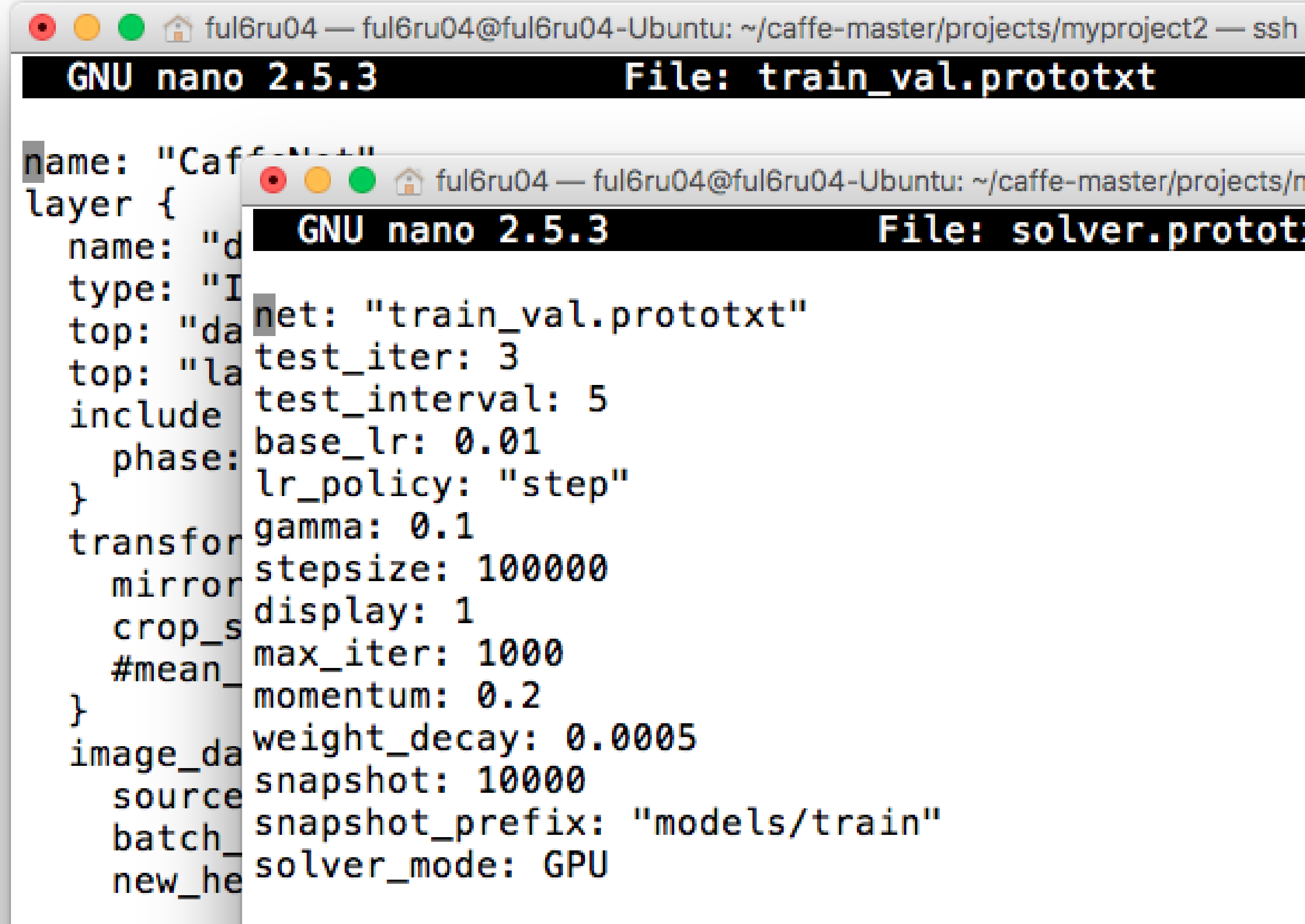


```
  GNU nano 2.5.3            File: train_val

}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
```



Protobuf Model Format

- Auto-generates code
- Developed by Google
- Defines Net / Layer / Solver schemas in ***.proto**



```
  GNU nano 2.5.3             File: train_val.prototxt

name: "Caf
layer {
  name: "d
  type: "I
  top: "da
  top: "la
  include
    phase:
  }
  transfor
    mirror
    crop_s
    #mean_
  }
  image_da
    source
    batch_
    new_he
```

```
  GNU nano 2.5.3             File: solver.protot

net: "train_val.prototxt"
test_iter: 3
test_interval: 5
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 1
max_iter: 1000
momentum: 0.2
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "models/train"
solver_mode: GPU
```

Gradient Descent

Review: Gradient Descent

- In step 3, we have to solve the following optimization problem:

$$\theta^* = \arg\min_{\theta} L(\theta)$$ L: loss function θ: parameters

---

Suppose that θ has two variables {θ_1, θ_2}

Randomly start at $\theta^0 = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix}$

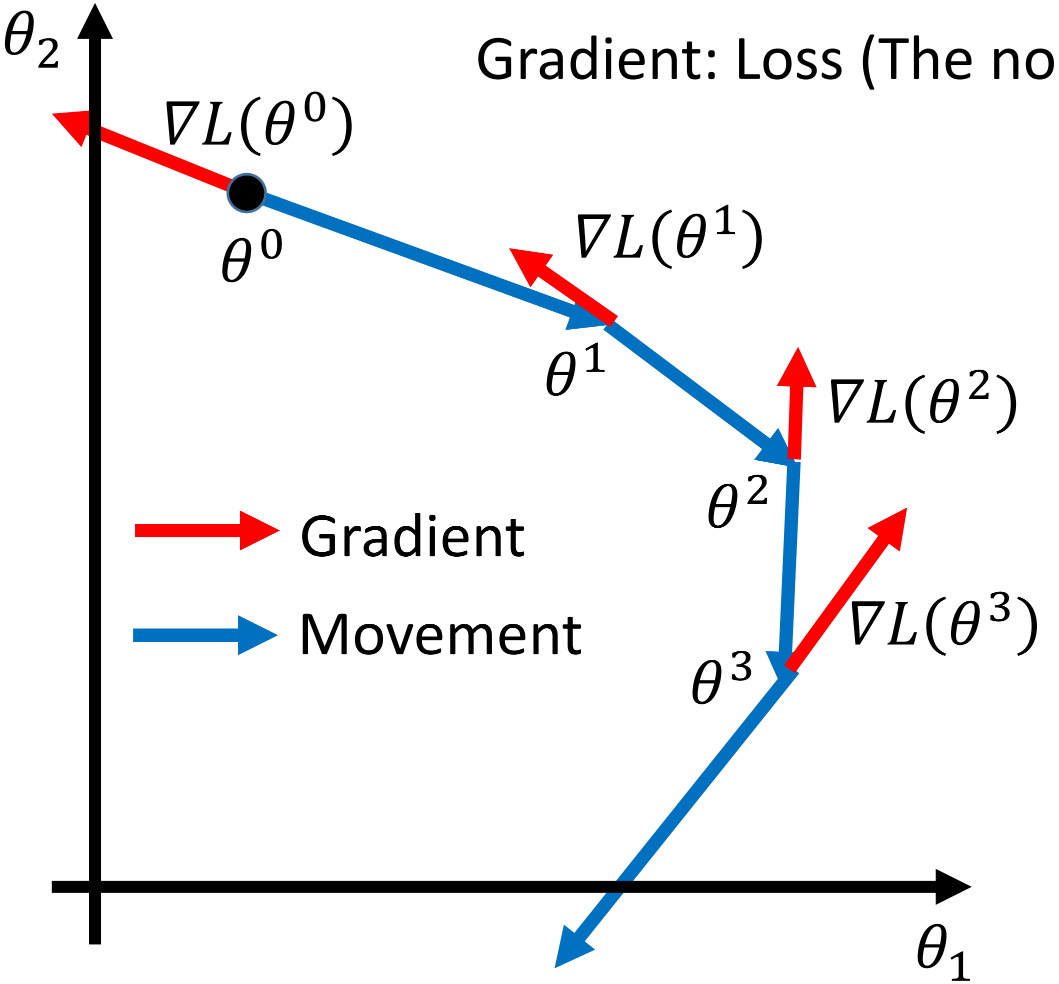
$$\nabla L(\theta) = \begin{bmatrix} \partial L(\theta_1)/\partial \theta_1 \\ \partial L(\theta_2)/\partial \theta_2 \end{bmatrix}$$

$$\begin{bmatrix} \theta_1^1 \\ \theta_2^1 \end{bmatrix} = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix} - \eta \begin{bmatrix} \partial L(\theta_1^0)/\partial \theta_1 \\ \partial L(\theta_2^0)/\partial \theta_2 \end{bmatrix} \Rightarrow \theta^1 = \theta^0 - \eta \nabla L(\theta^0)$$

$$\begin{bmatrix} \theta_1^2 \\ \theta_2^2 \end{bmatrix} = \begin{bmatrix} \theta_1^1 \\ \theta_2^1 \end{bmatrix} - \eta \begin{bmatrix} \partial L(\theta_1^1)/\partial \theta_1 \\ \partial L(\theta_2^1)/\partial \theta_2 \end{bmatrix} \Rightarrow \theta^2 = \theta^1 - \eta \nabla L(\theta^1)$$

Review: Gradient Descent

Gradient: Loss (The normal direction of the contour)



Start at position θ^0

Compute gradient at θ^0

Move to $\theta^1 = \theta^0 - \eta\nabla L(\theta^0)$

Compute gradient at θ^1

Move to $\theta^2 = \theta^1 - \eta\nabla L(\theta^1)$

⋮

Gradient Descent

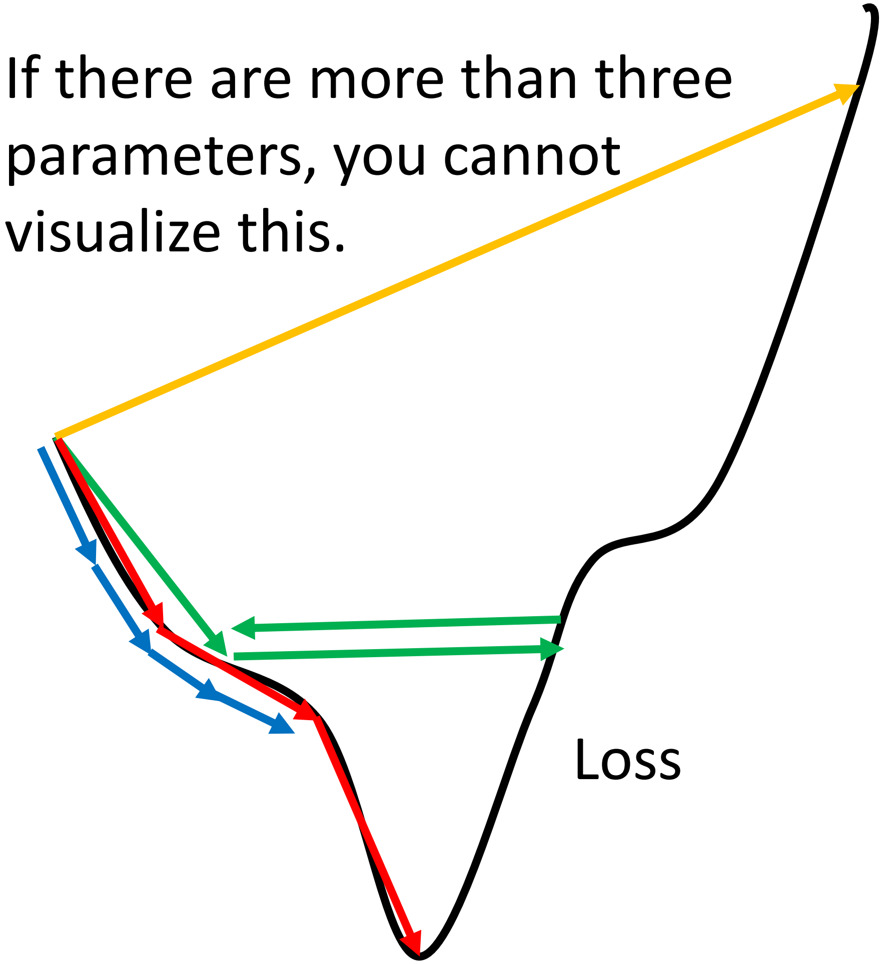
Tip 1: Tuning your learning rates

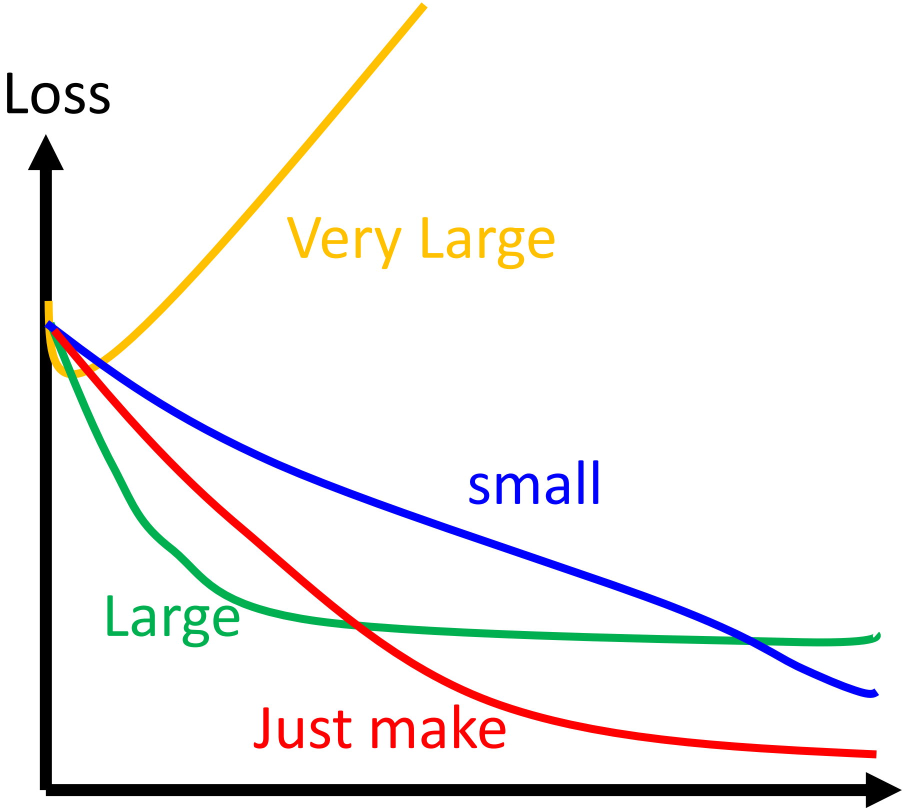
Learning Rate

$$\theta^{i} = \theta^{i-1} - \eta \nabla L\left(\theta^{i-1}\right)$$

Set the learning rate η carefully

If there are more than three parameters, you cannot visualize this.





No. of parameters updates

But you can always visualize this.

Adaptive Learning Rates

- Popular & Simple Idea: Reduce the learning rate by some factor every few epochs.
    - At the beginning, we are far from the destination, so we use larger learning rate
    - After several epochs, we are close to the destination, so we reduce the learning rate
    - E.g. 1/t decay: $\eta^t = \eta / \sqrt{t+1}$
- Learning rate cannot be one-size-fits-all
    - Giving different parameters different learning rates

Adagrad

$$\eta^t = \frac{\eta}{\sqrt{t+1}} \qquad g^t = \frac{\partial L(\theta^t)}{\partial w}$$

- Divide the learning rate of each parameter by the ***root mean square of its previous derivatives***

Vanilla Gradient descent

$$w^{t+1} \leftarrow w^t - \eta^t g^t$$

w is one parameters

Adagrad

$$w^{t+1} \leftarrow w^t - \frac{\eta^t}{\sigma^t} g^t$$

σ^t: ***root mean square*** of the previous derivatives of parameter w

Parameter dependent

Adagrad

σ^t: ***root mean square*** of the previous derivatives of parameter w

$$w^1 \leftarrow w^0 - \frac{\eta^0}{\sigma^0} g^0 \qquad \sigma^0 = \sqrt{(g^0)^2}$$

$$w^2 \leftarrow w^1 - \frac{\eta^1}{\sigma^1} g^1 \qquad \sigma^1 = \sqrt{\frac{1}{2}[(g^0)^2 + (g^1)^2]}$$

$$w^3 \leftarrow w^2 - \frac{\eta^2}{\sigma^2} g^2 \qquad \sigma^2 = \sqrt{\frac{1}{3}[(g^0)^2 + (g^1)^2 + (g^2)^2]}$$

$$\vdots$$

$$w^{t+1} \leftarrow w^t - \frac{\eta^t}{\sigma^t} g^t \qquad \sigma^t = \sqrt{\frac{1}{t+1}\sum_{i=0}^{t}(g^i)^2}$$

Adagrad

- Divide the learning rate of each parameter by the ***root mean square of its previous derivatives***

$$w^{t+1} \leftarrow w^t - \frac{\eta^t}{\sigma^t} g^t$$

$$\eta^t = \frac{\eta}{\sqrt{\cancel{t+1}}}$$

1/t decay

$$\sigma^t = \sqrt{\frac{1}{\cancel{t+1}} \sum_{i=0}^{t} (g^i)^2}$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^{t} (g^i)^2}} g^t$$

Contradiction?

$$\eta^t = \frac{\eta}{\sqrt{t+1}} \qquad g^t = \frac{\partial L(\theta^t)}{\partial w}$$

Vanilla Gradient descent

$$w^{t+1} \leftarrow w^t - \eta^t \underline{g^t}$$

Larger gradient, larger step

Adagrad

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\underline{\sum_{i=0}^{t}(g^i)^2}}} \underline{g^t}$$

Larger gradient, larger step

Larger gradient, smaller step

Intuitive Reason

$$\eta^t = \frac{\eta}{\sqrt{t+1}} \quad g^t = \frac{\partial L(\theta^t)}{\partial w}$$

- How surprise it is

Contrast

Very High

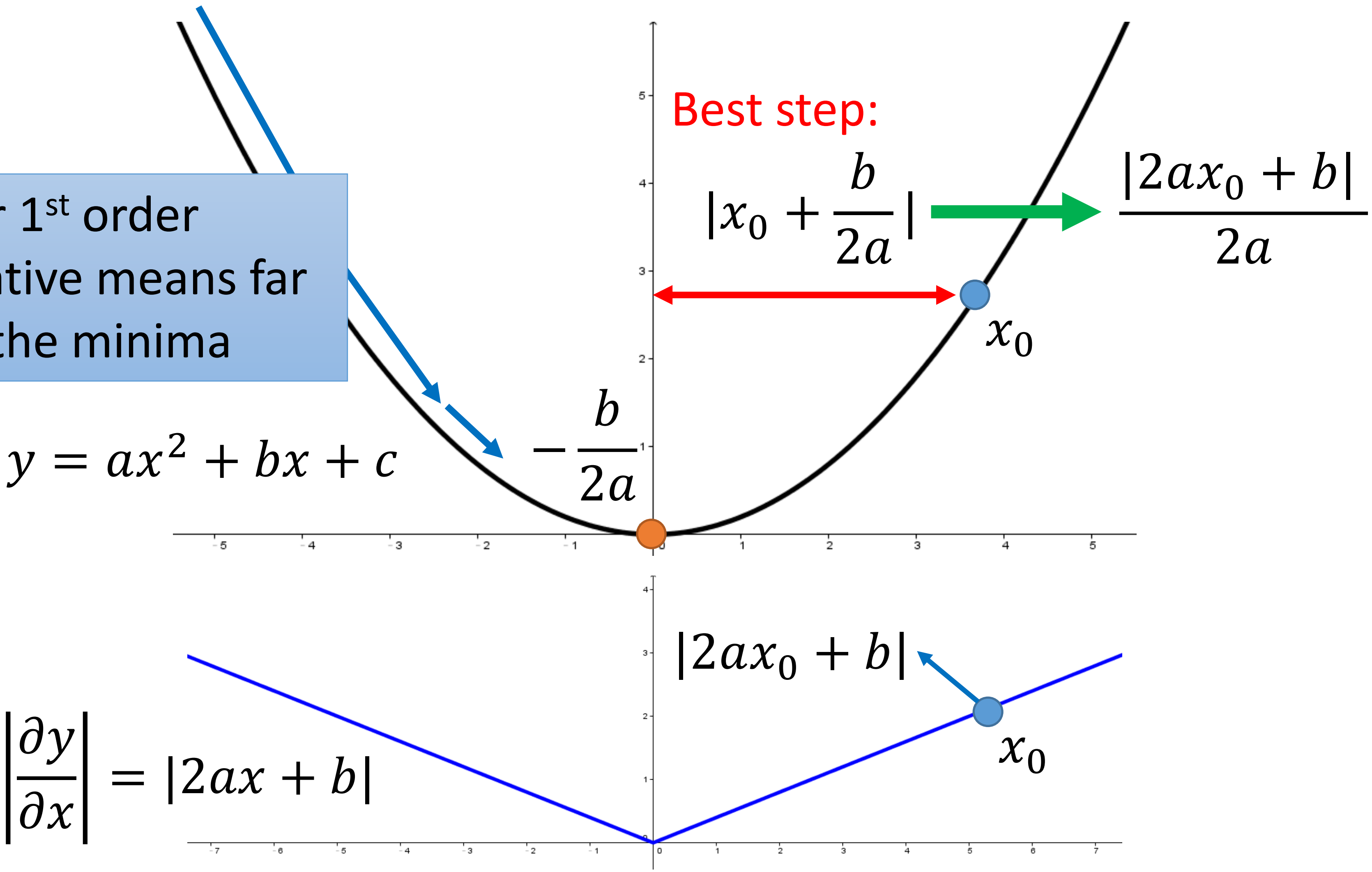
g^0	g^1	g^2	g^3	g^4	……
0.001	0.001	0.003	0.002	0.1	……

g^0	g^1	g^2	g^3	g^4	……
10.8	20.9	31.7	12.1	0.1	……

Very Small

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\underline{\sum_{i=0}^{t}(g^i)^2}}} g^t \longrightarrow \text{Contrasting Effect}$$

Larger gradient, larger steps?



Larger 1st order derivative means far from the minima

Best step:

$$\left|x_0 + \frac{b}{2a}\right| \rightarrow \frac{|2ax_0 + b|}{2a}$$

$$-\frac{b}{2a}$$

$$y = ax^2 + bx + c$$

x_0

$|2ax_0 + b|$

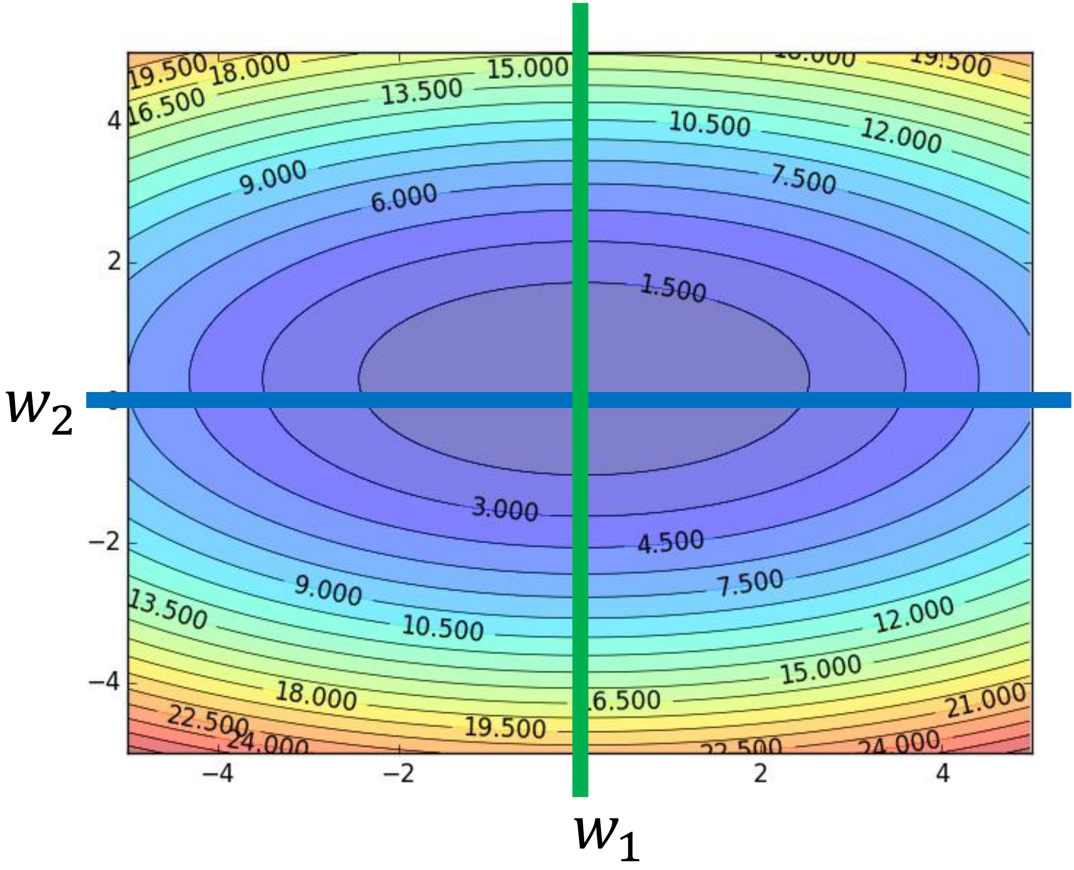
x_0

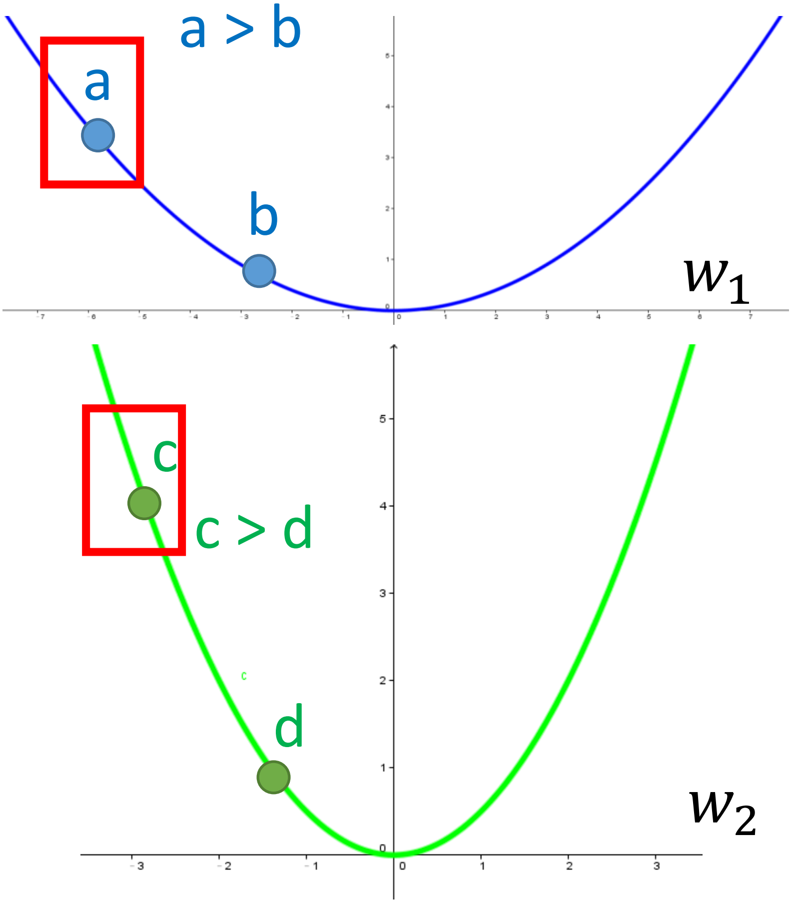
$$\left|\frac{\partial y}{\partial x}\right| = |2ax + b|$$

Comparison between different parameters

~~Larger 1st order derivative means far from the minima~~

Do not cross parameters

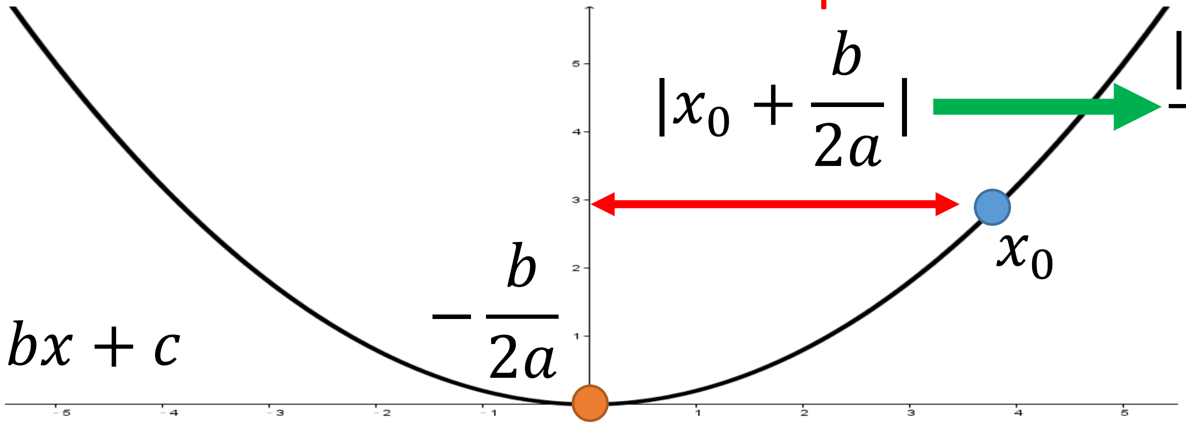




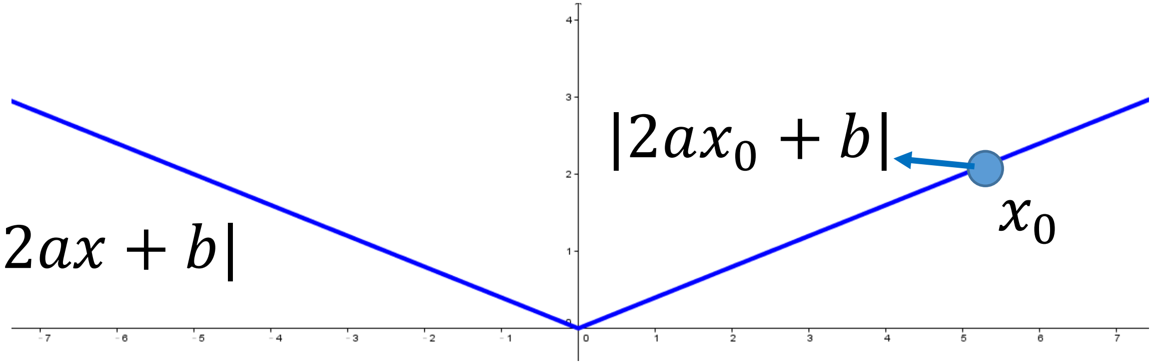
Second Derivative

Best step:

$$\left|x_0 + \frac{b}{2a}\right| \rightarrow \frac{|2ax_0 + b|}{\boxed{2a}}$$



$$y = ax^2 + bx + c$$



$$\left|\frac{\partial y}{\partial x}\right| = |2ax + b|$$

$$\frac{\partial^2 y}{\partial x^2} = 2a$$

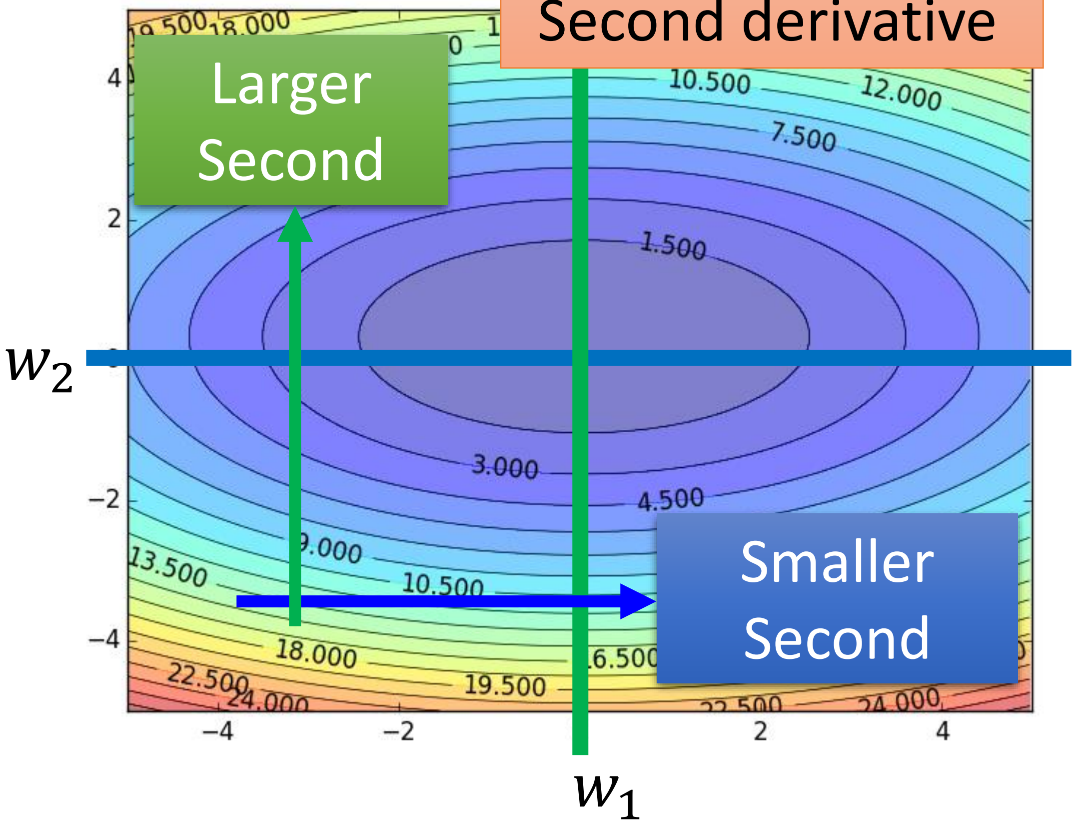
The best step is $\frac{|\text{First derivative}|}{\text{Second derivative}}$

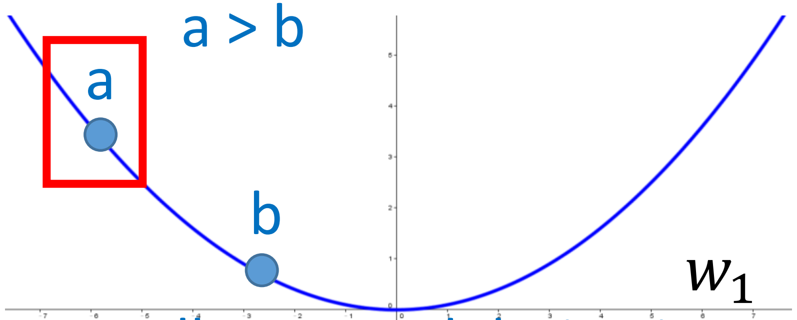
Comparison between different parameters

~~Larger 1st order derivative means far from the minima~~

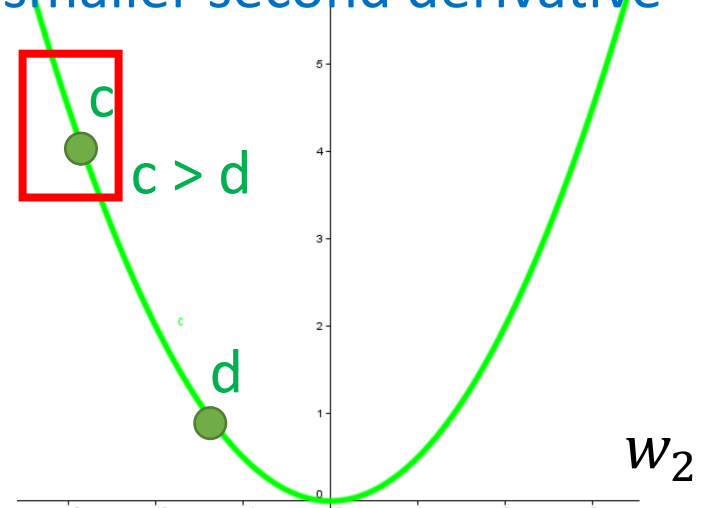
Do not cross parameters

The best step is $\frac{|\text{First derivative}|}{\text{Second derivative}}$





smaller second derivative



Larger second derivative

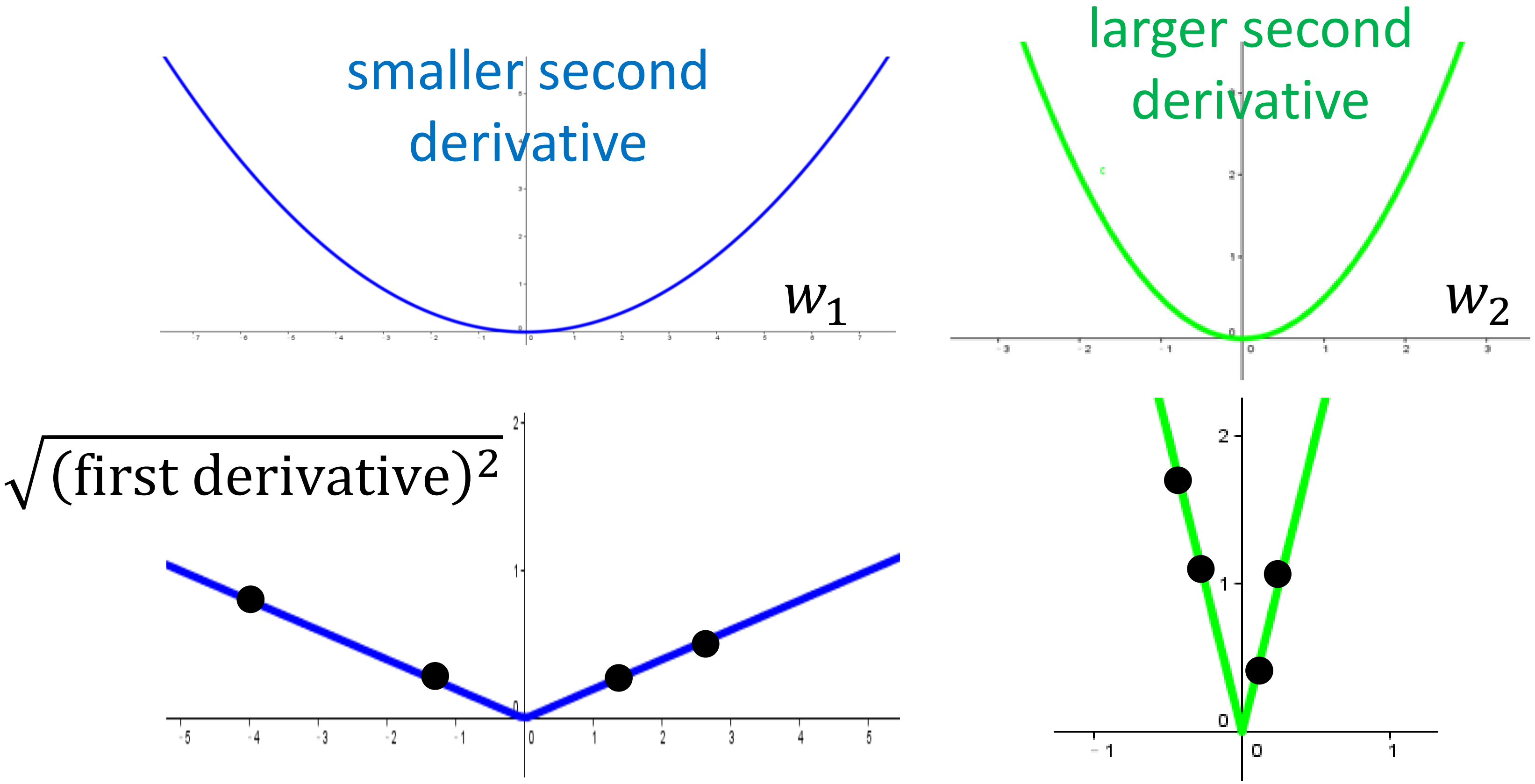
$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2}}\, g^t$$

The best step is

$$\frac{|\text{First derivative}|}{\text{Second derivative}}$$

?

Use *first derivative* to estimate *second derivative*



Gradient Descent

Tip 2: Stochastic Gradient Descent

Make the training faster

Stochastic Gradient Descent

$$L = \sum_n \left(\hat{y}^n - \left(b + \sum w_i x_i^n \right) \right)^2$$

Loss is the summation over all training examples

◆ ***Gradient Descent*** $\theta^i = \theta^{i-1} - \eta \nabla L\left(\theta^{i-1}\right)$

◆ ***Stochastic Gradient Descent***

Faster!

Pick an example x^n

$$L^n = \left(\hat{y}^n - \left(b + \sum w_i x_i^n \right) \right)^2 \qquad \theta^i = \theta^{i-1} - \eta \nabla L^n\left(\theta^{i-1}\right)$$

Loss for only one example

Stochastic Gradient Descent

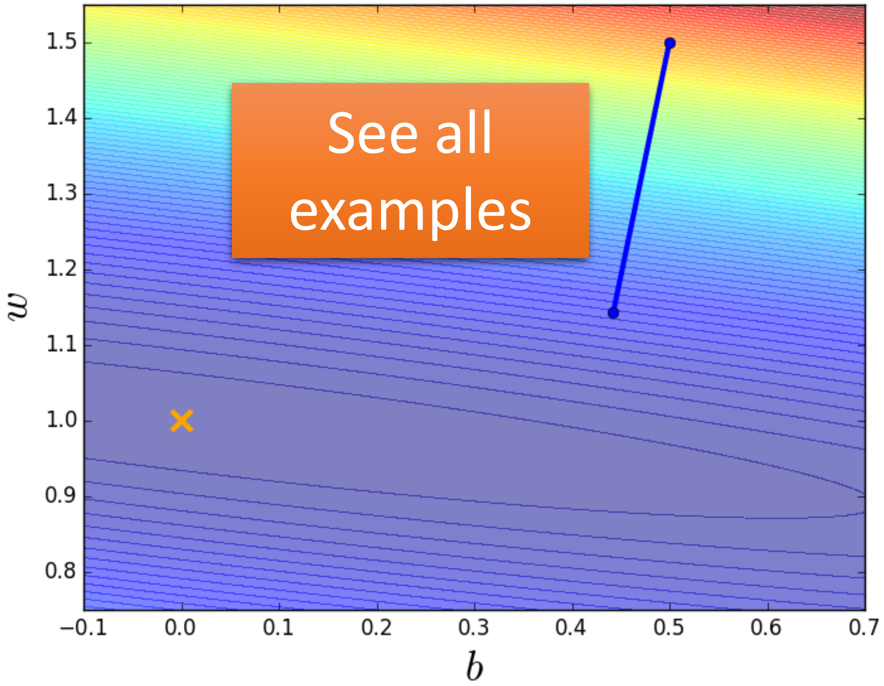
Gradient Descent

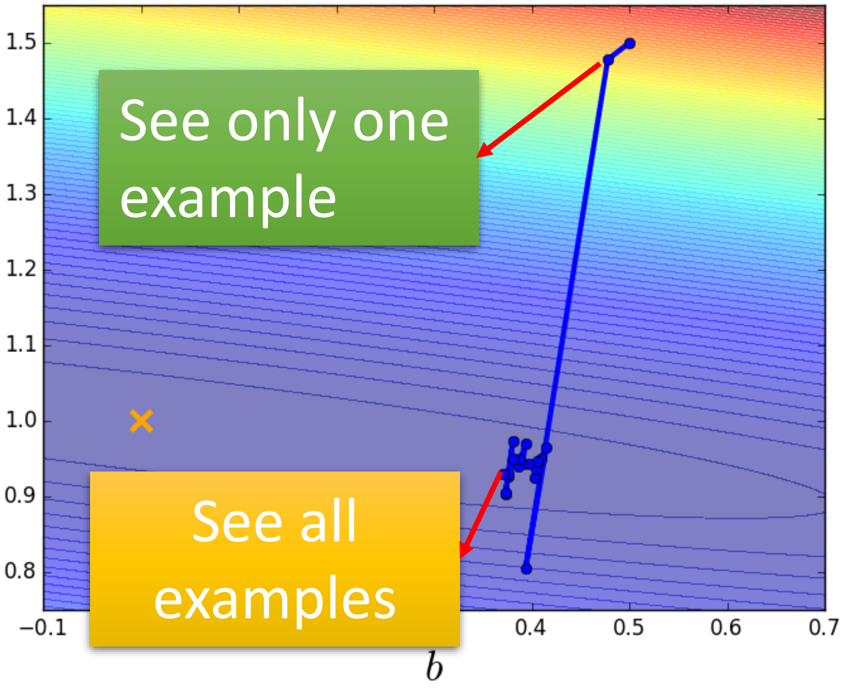
Update after seeing all examples

Stochastic Gradient Descent

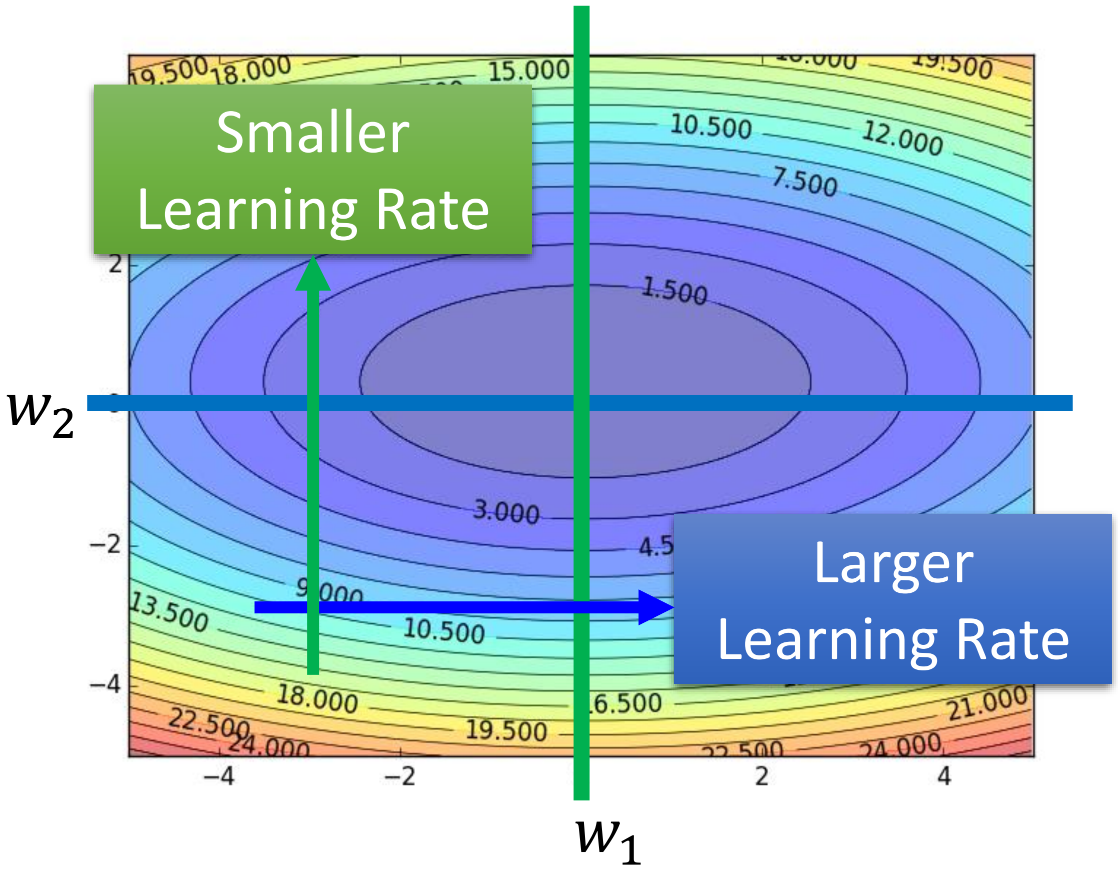
Update for each example

If there are 20 examples, 20 times faster.





Review



Smaller Learning Rate

Larger Learning Rate

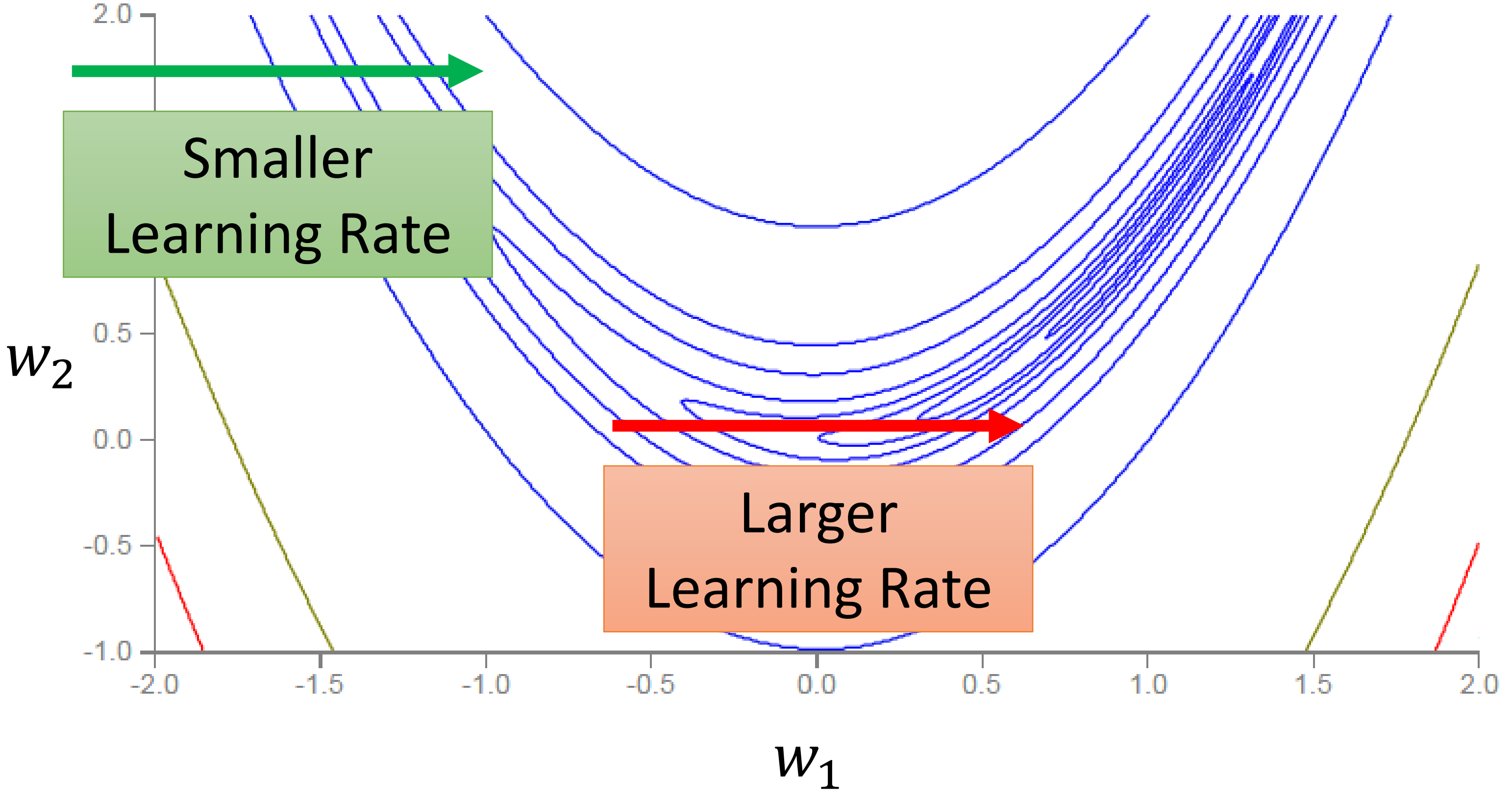
Adagrad

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2}} g^t$$

Use first derivative to estimate second derivative

RMSProp

Error Surface can be very complex when training NN.



Smaller
Learning Rate

Larger
Learning Rate

RMSProp

$$w^1 \leftarrow w^0 - \frac{\eta}{\sigma^0} g^0 \qquad \sigma^0 = g^0$$

$$w^2 \leftarrow w^1 - \frac{\eta}{\sigma^1} g^1 \qquad \sigma^1 = \sqrt{\alpha(\sigma^0)^2 + (1-\alpha)(g^1)^2}$$

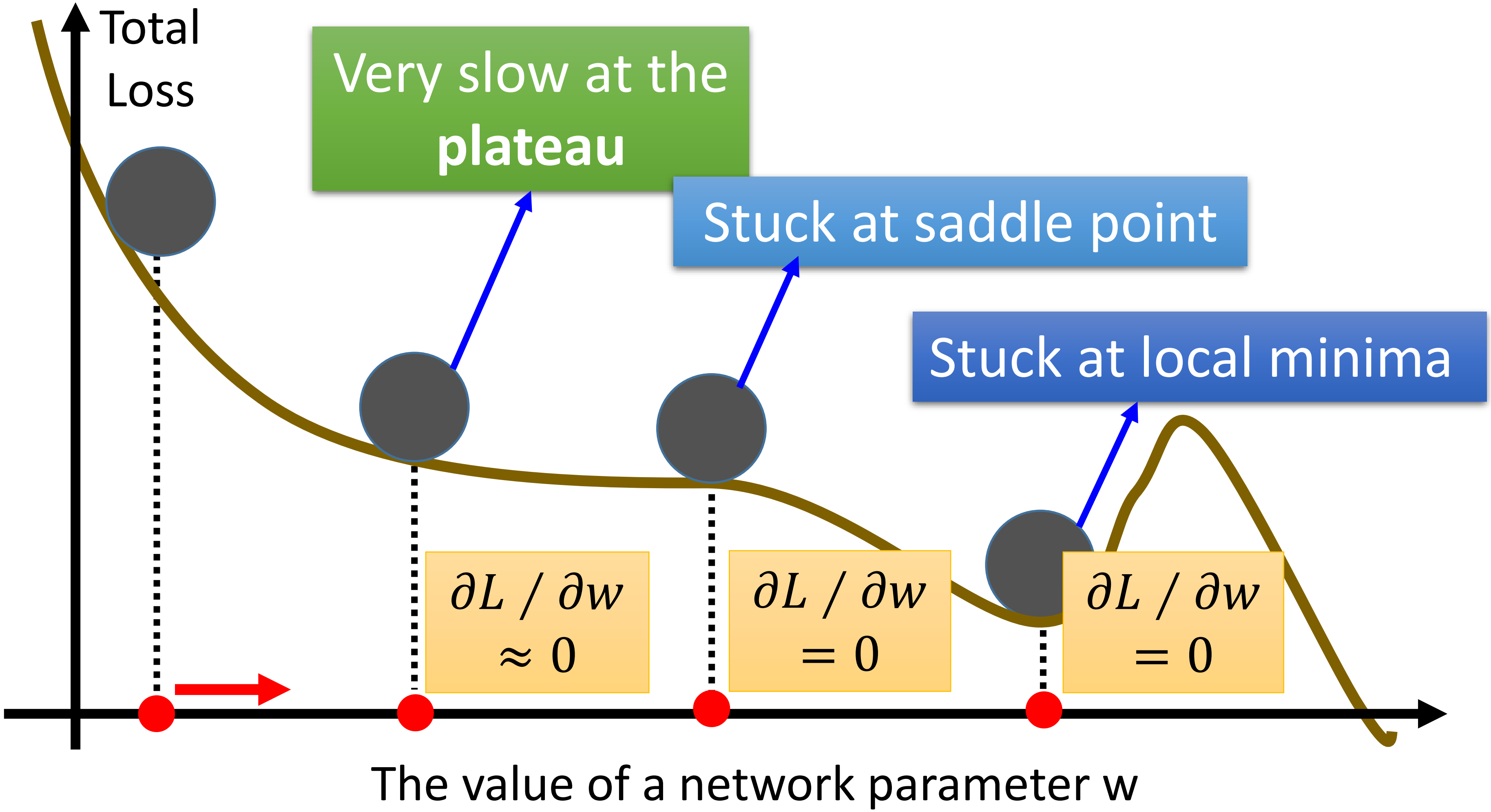
$$w^3 \leftarrow w^2 - \frac{\eta}{\sigma^2} g^2 \qquad \sigma^2 = \sqrt{\alpha(\sigma^1)^2 + (1-\alpha)(g^2)^2}$$

$$\vdots$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sigma^t} g^t \qquad \sigma^t = \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2}$$

Root Mean Square of the gradients
with previous gradients being decayed

Hard to find
optimal network parameters



Very slow at the **plateau**

Stuck at saddle point

Stuck at local minima

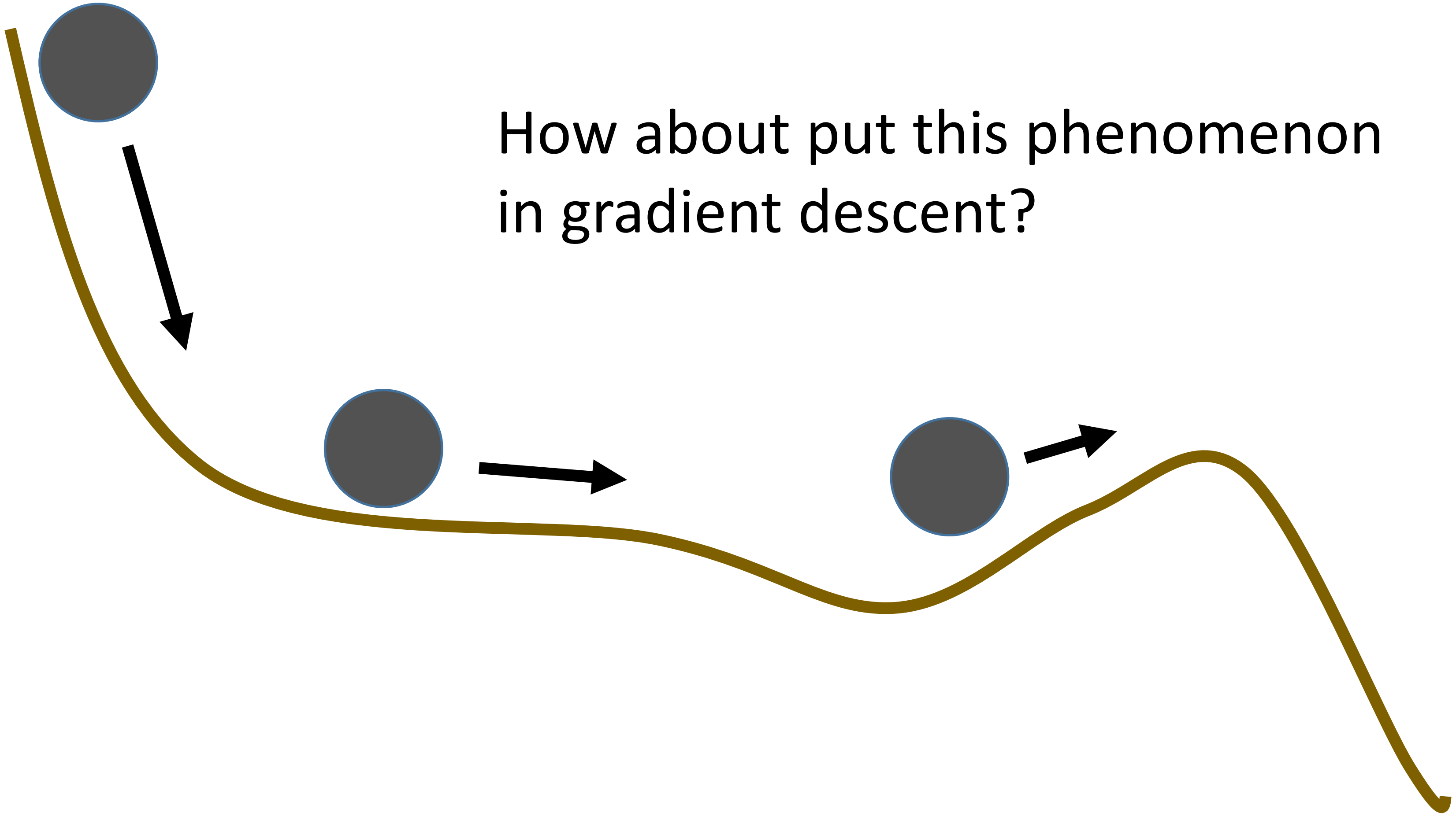
$\partial L \,/\, \partial w \approx 0$

$\partial L \,/\, \partial w = 0$

$\partial L \,/\, \partial w = 0$

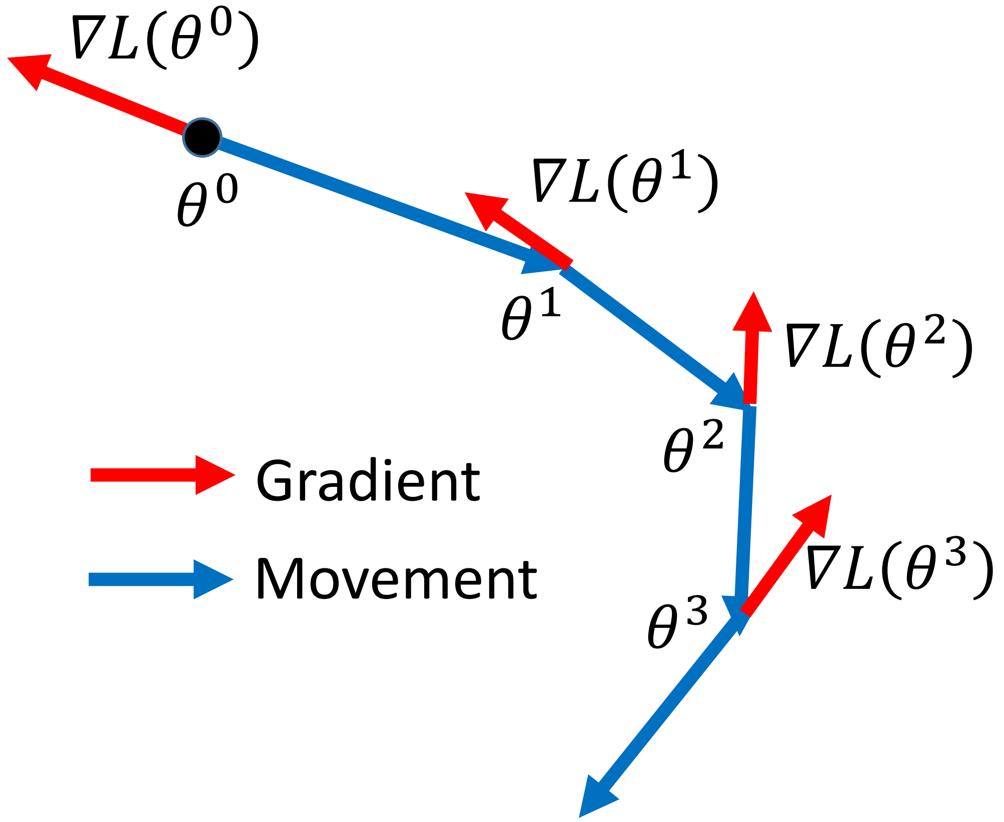
In physical world ……

- Momentum



How about put this phenomenon in gradient descent?

Review: Vanilla Gradient Descent



Start at position θ^0

Compute gradient at θ^0

Move to $\theta^1 = \theta^0 - \eta \nabla L(\theta^0)$

Compute gradient at θ^1

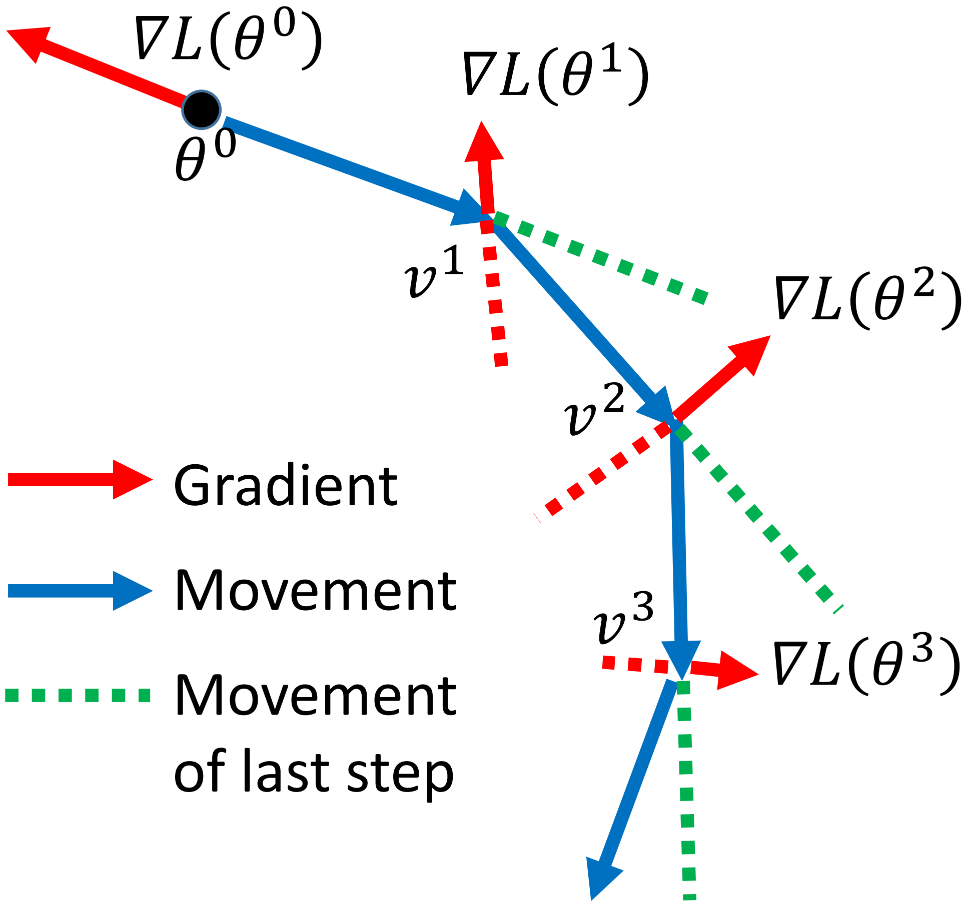
Move to $\theta^2 = \theta^1 - \eta \nabla L(\theta^1)$

⋮

Stop until $\nabla L(\theta^t) \approx 0$

Momentum

Movement: movement of last step minus gradient at present



Start at point θ^0

Movement $v^0=0$

Compute gradient at θ^0

Movement $v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

Compute gradient at θ^1

Movement $v^2 = \lambda v^1 - \eta \nabla L(\theta^1)$

Move to $\theta^2 = \theta^1 + v^2$

Movement not just based on gradient, but previous movement.

Momentum

Movement: movement of last step minus gradient at present

v^i is actually the weighted sum of all the previous gradient: $\nabla L(\theta^0), \nabla L(\theta^1), \ldots \nabla L(\theta^{i-1})$

$v^0 = 0$

$v^1 = -\eta \nabla L(\theta^0)$

$v^2 = -\lambda \eta \nabla L(\theta^0) - \eta \nabla L(\theta^1)$

$\vdots$

Start at point θ^0

Movement $v^0=0$

Compute gradient at θ^0

Movement $v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

Compute gradient at θ^1

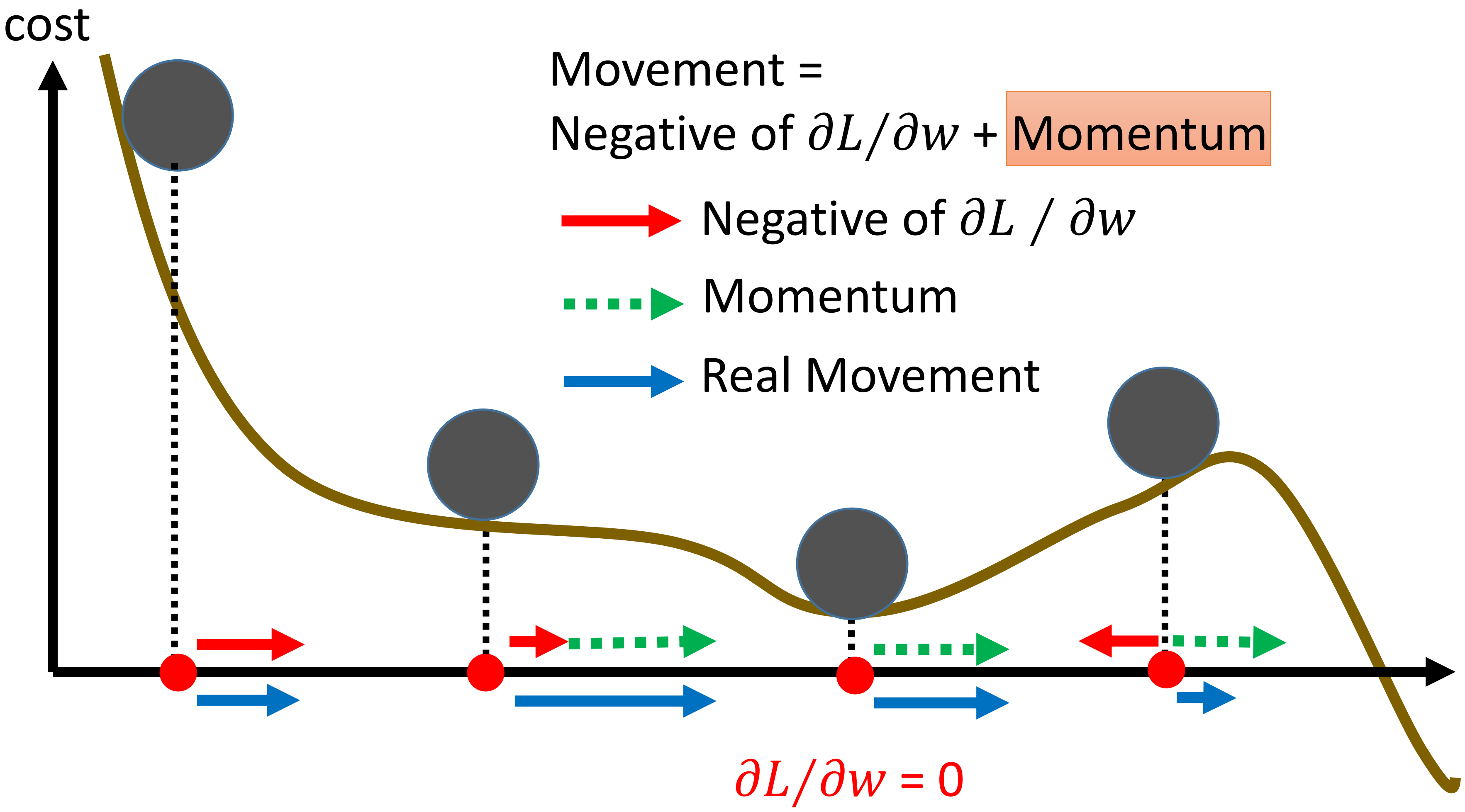
Movement $v^2 = \lambda v^1 - \eta \nabla L(\theta^1)$

Move to $\theta^2 = \theta^1 + v^2$

Movement not just based on gradient, but previous movement

Momentum

Still not guarantee reaching global minima, but give some hope ……



Movement =
Negative of $\partial L/\partial w$ + **Momentum**

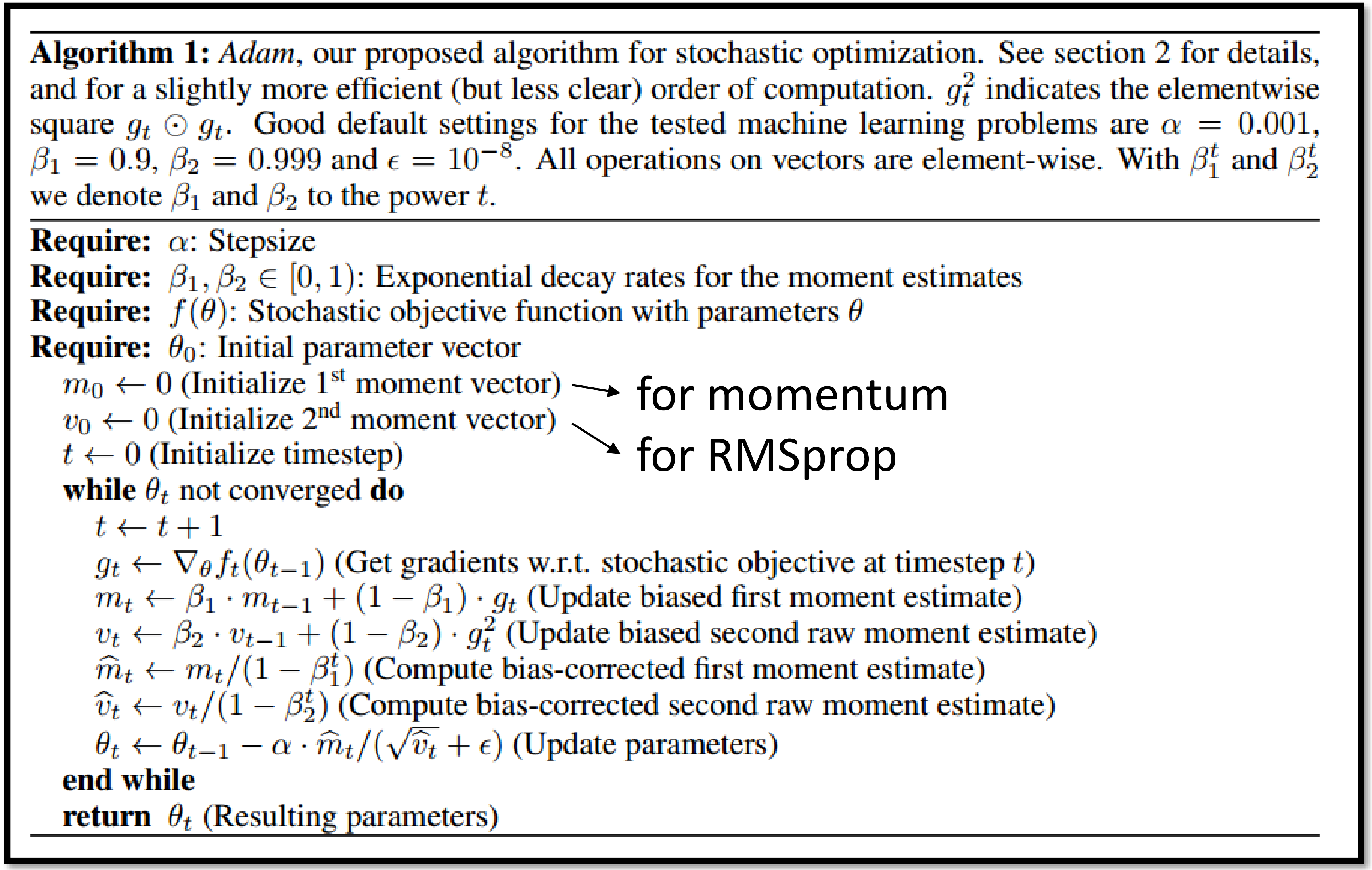
→ Negative of $\partial L \,/\, \partial w$

⇢ Momentum

→ Real Movement

Adam

RMSProp + Momentum



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

for momentum

for RMSprop