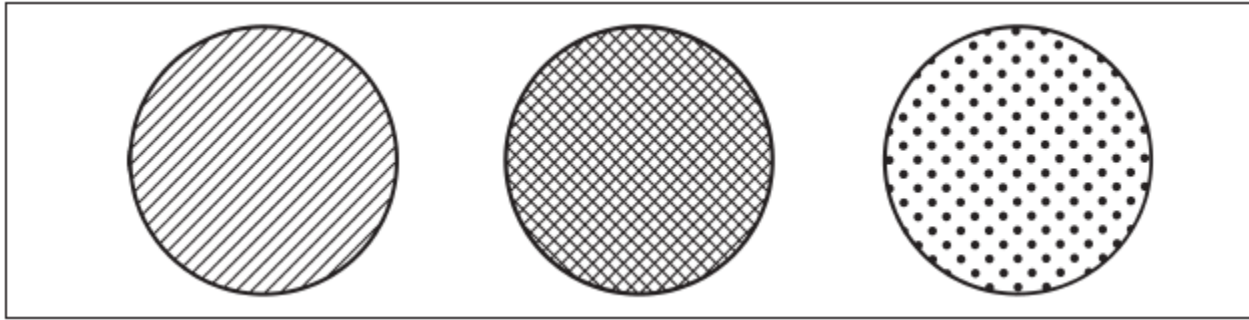


Introduction to backpropagation

Building your first deep neural network

Chapter-6

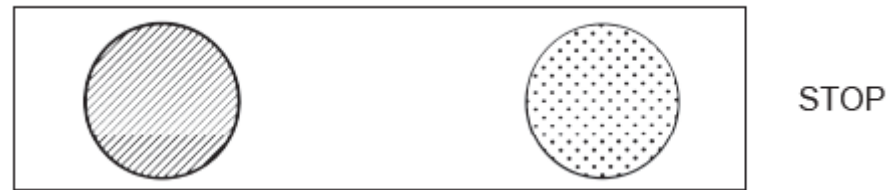
The streetlight problem



Consider yourself approaching a street corner in a foreign country. As you approach, you look up and realize that the streetlight is unfamiliar.

How can you know when it's safe to cross the street?

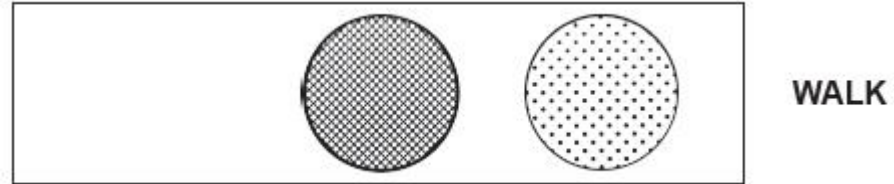
To solve this problem, you might sit at the street corner for a few minutes observing the correlation between each light combination and whether people around you choose to walk or stop. You take a seat and record the following pattern:



OK, nobody walked at the first light. At this point you're thinking, "Wow, this pattern could be anything."

The left light or the right light could be correlated with stopping, or the central light could be correlated with walking."

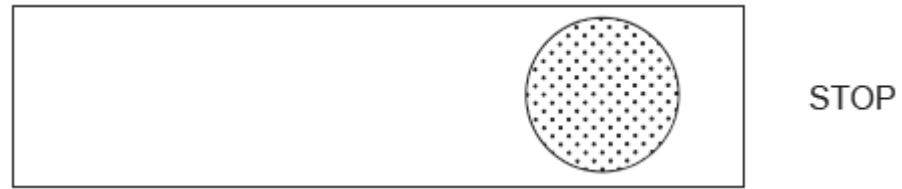
Let's take another datapoint:



People walked, so something about this light changed the signal.

The only thing you know for sure is that the far-right light doesn't seem to indicate one way or another. Perhaps it's irrelevant.

Let's take another datapoint:



Only the middle light changed this time, and you got the opposite pattern.

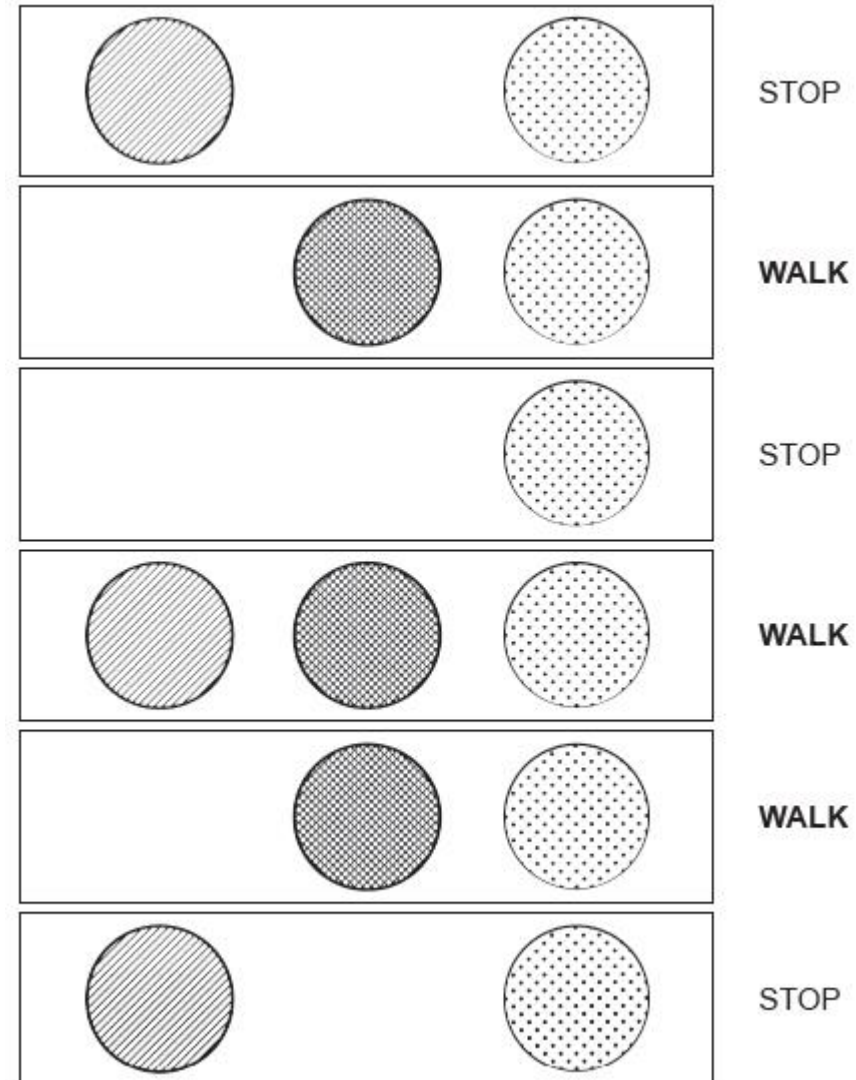
The working hypothesis is that the *middle* light indicates when people feel safe to walk.

Over the next few minutes, you record the following six light patterns, noting when people walk or stop.

As hypothesized, there is a *perfect correlation* between the middle (crisscross) light and whether it's safe to walk.

You learned this pattern by observing all the individual datapoints and *searching for correlation*.







This is what you're going to train a neural network to do.



Preparing the data







In supervised algorithms. Take a dataset of *what you know* and turn it into a dataset of *what you want to know*.

In this particular real-world example, you know the state of the streetlight at any given time, and you want to know whether it's safe to cross the street

What you know	What you want to know
	STOP
	WALK
	STOP
	WALK
	WALK
	STOP


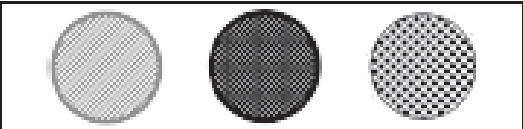

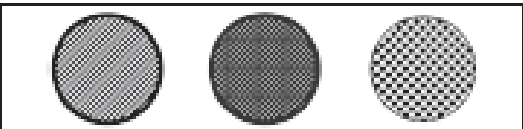
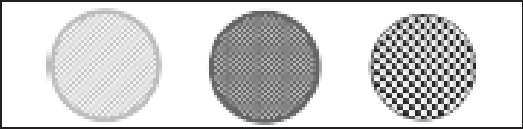
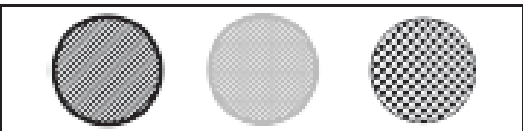
Matrices and the matrix relationship

Translate the streetlight into math.



















Streetlights		Streetlight pattern
	→	1 0 1
	→	0 1 1
	→	0 0 1
	→	1 1 1
	→	0 1 1
	→	1 0 1

In data matrices, it's convention to give each *recorded example* a single *row*. It's also convention to give each *thing being recorded* a single *column*. This makes the matrix easy to read. In this case, a column contains every on/off state recorded for a particular light. Each row contains the simultaneous state of every light at a particular moment in time.

If the streetlights were on dimmers and turned on and off at varying degrees of intensity?

Streetlights	Streetlight matrix A
	→ .9 .0 1
	→ .2 .8 1
	→ .1 .0 1
	→ .8 .9 1
	→ .1 .7 1
	→ .9 .1 0

Would the following matrix still be valid?







Streetlights		Streetlight matrix B				
			→	9	0	10
			→	2	8	10
			→	1	0	10
			→	8	9	10
			→	1	7	10
			→	9	1	0

Matrix (B) is valid. It adequately captures the relationships between various training examples (rows) and lights (columns). Note that `Matrix A * 10 == Matrix B` (`A * 10 == B`).

This means these matrices are *scalar multiples* of each other.







Creating a matrix or two in Python

Import the matrices into Python.

Streetlights	Streetlight pattern	
	→ 1 0 1	<pre>import numpy as np streetlights = np.array([[1, 0, 1], [0, 1, 1], [0, 0, 1], [1, 1, 1], [0, 1, 1], [1, 0, 1]])</pre>
	→ 0 1 1	
	→ 0 0 1	
	→ 1 1 1	
	→ 0 1 1	
	→ 1 0 1	

A matrix is just a list of lists. It's an array of arrays.

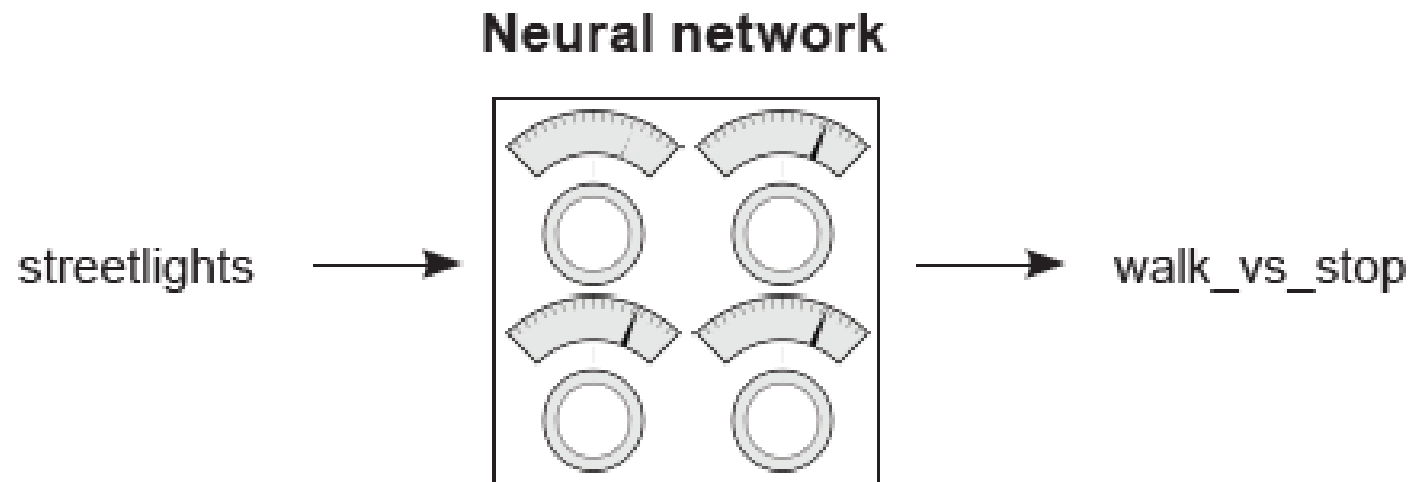
Let's create a NumPy matrix for the output data

What you know	What you want to know				
	STOP	STOP	→	0	walk_vs_stop = np.array([[0], [1], [0], [1], [1], [0]])
	WALK	WALK	→	1	
	STOP	STOP	→	0	
	WALK	WALK	→	1	
	WALK	WALK	→	1	
	STOP	STOP	→	0	

What do you want the neural network to do?

Take the `streetlights` matrix and learn to transform it into the `walk_vs_stop` matrix.

More important, you want the neural network to take *any matrix containing the same underlying pattern* as `streetlights` and transform it into a matrix that contains the underlying pattern of `walk_vs_stop`.



Building a neural network

```
import numpy as np
weights = np.array([0.5, 0.48, -0.7])
alpha = 0.1

streetlights = np.array( [ [ 1, 0, 1 ],
                           [ 0, 1, 1 ],
                           [ 0, 0, 1 ],
                           [ 1, 1, 1 ],
                           [ 0, 1, 1 ],
                           [ 1, 0, 1 ] ] )

walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

input = streetlights[0] ← [1,0,1]
goal_prediction = walk_vs_stop[0] ← Equals 0 (stop)

for iteration in range(20):
    prediction = input.dot(weights)
    error = (goal_prediction - prediction) ** 2
    delta = prediction - goal_prediction
    weights = weights - (alpha * (input * delta))

    print("Error:" + str(error) + " Prediction:" + str(prediction))
```

```

import numpy as np
weights = np.array([0.5,0.48,-0.7])
alpha = 0.1

streetlights = np.array( [ [ 1, 0, 1 ],
                           [ 0, 1, 1 ],
                           [ 0, 0, 1 ],
                           [ 1, 1, 1 ],
                           [ 0, 1, 1 ],
                           [ 1, 0, 1 ] ] )

walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

input = streetlights[0] # [1,0,1]
goal_prediction = walk_vs_stop[0] # equals 0... i.e. "stop"

for iteration in range(20):
    prediction = input.dot(weights)
    error = (goal_prediction - prediction) ** 2
    delta = prediction - goal_prediction
    weights = weights - (alpha * (input * delta))

    print("Error:" + str(error) + " Prediction:" + str(prediction))

```

Error:0.03999999999999998 Prediction:-0.19999999999999996
Error:0.025599999999999973 Prediction:-0.15999999999999992
Error:0.01638399999999997 Prediction:-0.1279999999999999
Error:0.010485759999999964 Prediction:-0.10239999999999982
Error:0.006710886399999962 Prediction:-0.08191999999999977
Error:0.004294967295999976 Prediction:-0.06553599999999982
Error:0.002748779069439994 Prediction:-0.05242879999999994
Error:0.0017592186044416036 Prediction:-0.04194304000000004
Error:0.0011258999068426293 Prediction:-0.03355443200000008
Error:0.0007205759403792803 Prediction:-0.02684354560000002
Error:0.0004611686018427356 Prediction:-0.021474836479999926
Error:0.0002951479051793508 Prediction:-0.01717986918399994
Error:0.00018889465931478573 Prediction:-0.013743895347199997
Error:0.00012089258196146188 Prediction:-0.010995116277759953
Error:7.737125245533561e-05 Prediction:-0.008796093022207963
Error:4.951760157141604e-05 Prediction:-0.007036874417766459
Error:3.169126500570676e-05 Prediction:-0.0056294995342132115
Error:2.028240960365233e-05 Prediction:-0.004503599627370569
Error:1.298074214633813e-05 Prediction:-0.003602879701896544
Error:8.307674973656916e-06 Prediction:-0.002882303761517324

Learning the whole dataset

The neural network has been learning only one streetlight. Don't we want it to learn them all?

```
import numpy as np
weights = np.array([0.5, 0.48, -0.7])
alpha = 0.1

streetlights = np.array( [ [ 1, 0, 1 ],
                           [ 0, 1, 1 ],
                           [ 0, 0, 1 ],
                           [ 1, 1, 1 ],
                           [ 0, 1, 1 ],
                           [ 1, 0, 1 ] ] )

walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

input = streetlights[0] ← [1,0,1]
goal_prediction = walk_vs_stop[0] ← Equals 0 (stop)

for iteration in range(20):
    prediction = input.dot(weights)
    error = (goal_prediction - prediction) ** 2
    delta = prediction - goal_prediction
    weights = weights - (alpha * (input * delta))

    print("Error:" + str(error) + " Prediction:" + str(prediction))
```

```
import numpy as np

weights = np.array([0.5, 0.48, -0.7])
alpha = 0.1

streetlights = np.array( [ [ 1, 0, 1 ],
                           [ 0, 1, 1 ],
                           [ 0, 0, 1 ],
                           [ 1, 1, 1 ],
                           [ 0, 1, 1 ],
                           [ 1, 0, 1 ] ] )

walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

input = streetlights[0] ← [1,0,1]
goal_prediction = walk_vs_stop[0] ← Equals 0 (stop)

for iteration in range(40):
    error_for_all_lights = 0
    for row_index in range(len(walk_vs_stop)):
        input = streetlights[row_index]
        goal_prediction = walk_vs_stop[row_index]

        prediction = input.dot(weights)

        error = (goal_prediction - prediction) ** 2
        error_for_all_lights += error

    delta = prediction - goal_prediction
    weights = weights - (alpha * (input * delta))
    print("Prediction:" + str(prediction))
    print("Error:" + str(error_for_all_lights) + "\n")
```

```

import numpy as np

weights = np.array([0.5,0.48,-0.7])
alpha = 0.1

streetlights = np.array( [[ 1, 0, 1 ],
                           [ 0, 1, 1 ],
                           [ 0, 0, 1 ],
                           [ 1, 1, 1 ],
                           [ 0, 1, 1 ],
                           [ 1, 0, 1 ] ] )

walk_vs_stop = np.array( [ 0, 1, 0, 1, 1, 0 ] )

input = streetlights[0] # [1,0,1]
goal_prediction = walk_vs_stop[0] # equals 0... i.e. "stop"

for iteration in range(40):
    error_for_all_lights = 0
    for row_index in range(len(walk_vs_stop)):
        input = streetlights[row_index]
        goal_prediction = walk_vs_stop[row_index]

        prediction = input.dot(weights)

        error = (goal_prediction - prediction) ** 2
        error_for_all_lights += error

        delta = prediction - goal_prediction
        weights = weights - (alpha * (input * delta))
        print("Prediction:" + str(prediction))
    print("Error:" + str(error_for_all_lights) + "\n")

```

Prediction:-0.199999999999999996
Prediction:-0.199999999999999996
Prediction:-0.559999999999999999
Prediction:0.616000000000000001
Prediction:0.172799999999999995
Prediction:0.17552
Error:2.6561231104

Prediction:0.140415999999999999
Prediction:0.3066464
Prediction:-0.34513824
Prediction:1.006637344
Prediction:0.4785034751999999
Prediction:0.26700416768
Error:0.9628701776715985

Prediction:0.213603334144
Prediction:0.5347420299776
Prediction:-0.26067345110016

Full, batch, and stochastic gradient descent

Stochastic gradient descent updates weights one example at a time.

As it turns out, this idea of learning one example at a time is a variant on gradient descent called *stochastic gradient descent*, and it's one of the handful of methods that can be used to learn an entire dataset.

How does stochastic gradient descent work? As you saw in the previous example, it performs a prediction and weight update for each training example separately. In other words, it takes the first streetlight, tries to predict it, calculates the `weight_delta`, and updates the weights. Then it moves on to the second streetlight, and so on. It iterates through the entire dataset many times until it can find a weight configuration that works well for all the training examples.

(Full) gradient descent updates weights one dataset at a time.

As introduced in chapter 4, another method for learning an entire dataset is gradient descent (or *average/full gradient descent*). Instead of updating the weights once for each training example, the network calculates the average `weight_delta` over the entire dataset, changing the weights only each time it computes a full average.

Batch gradient descent updates weights after n examples.

This will be covered in more detail later, but there's also a third configuration that sort of splits the difference between stochastic gradient descent and full gradient descent. Instead of updating the weights after just one example or after the entire dataset of examples, you choose a *batch size* (typically between 8 and 256) of examples, after which the weights are updated.