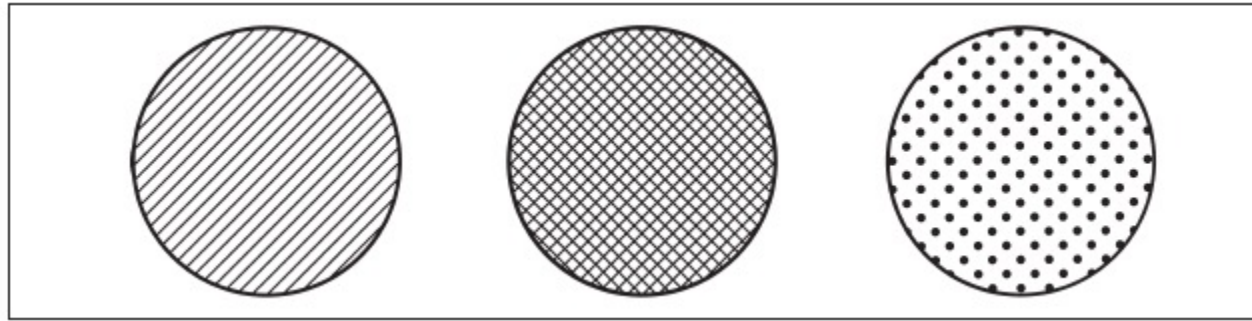


# Introduction to backpropagation

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?**

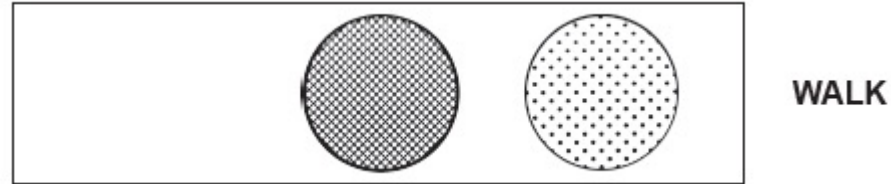
To solve this problem, you might sit at the street corner for a few minutes observing the correlation between each light combination and whether people around you choose to walk or stop. You take a seat and record the following pattern:



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."

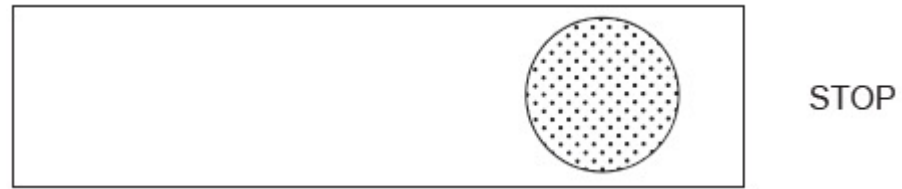
Let's take another datapoint:



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.

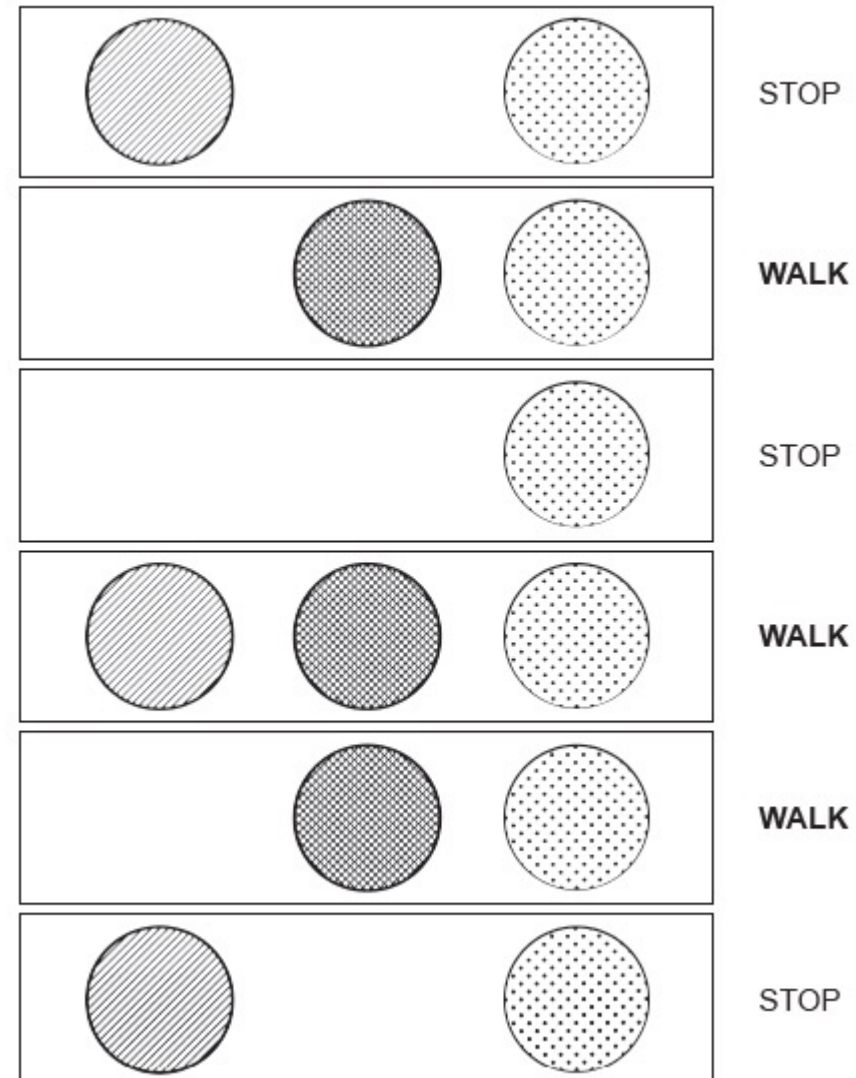
The working hypothesis is that the *middle* light indicates when people feel safe to walk.

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.

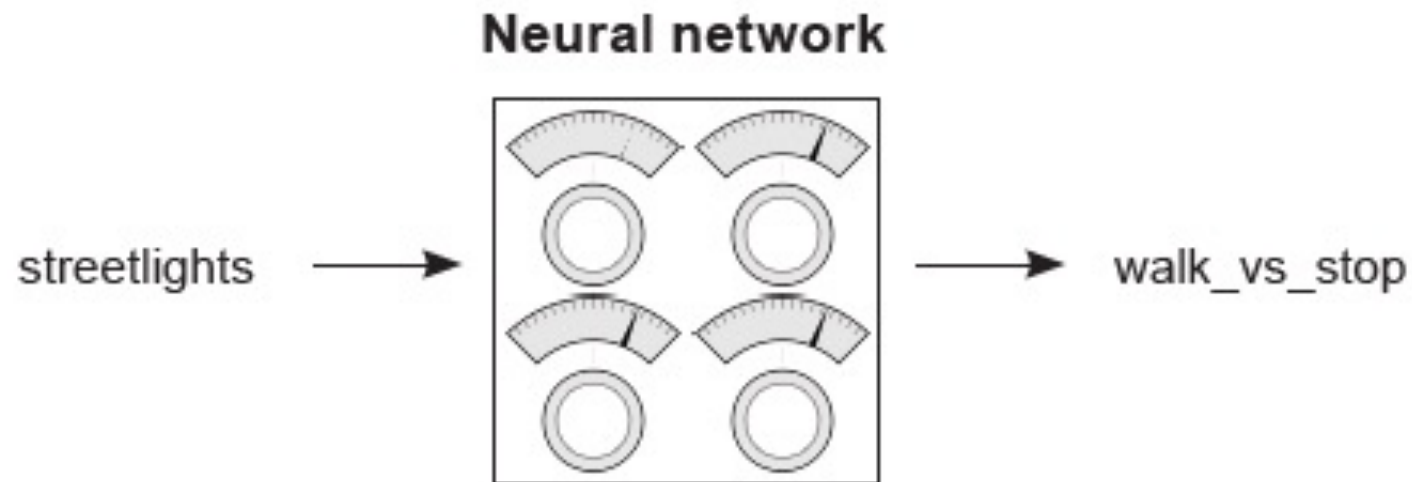
Let's create a NumPy matrix for the output data

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_allLights = 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_allLights += error

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



# Neural networks learn correlation

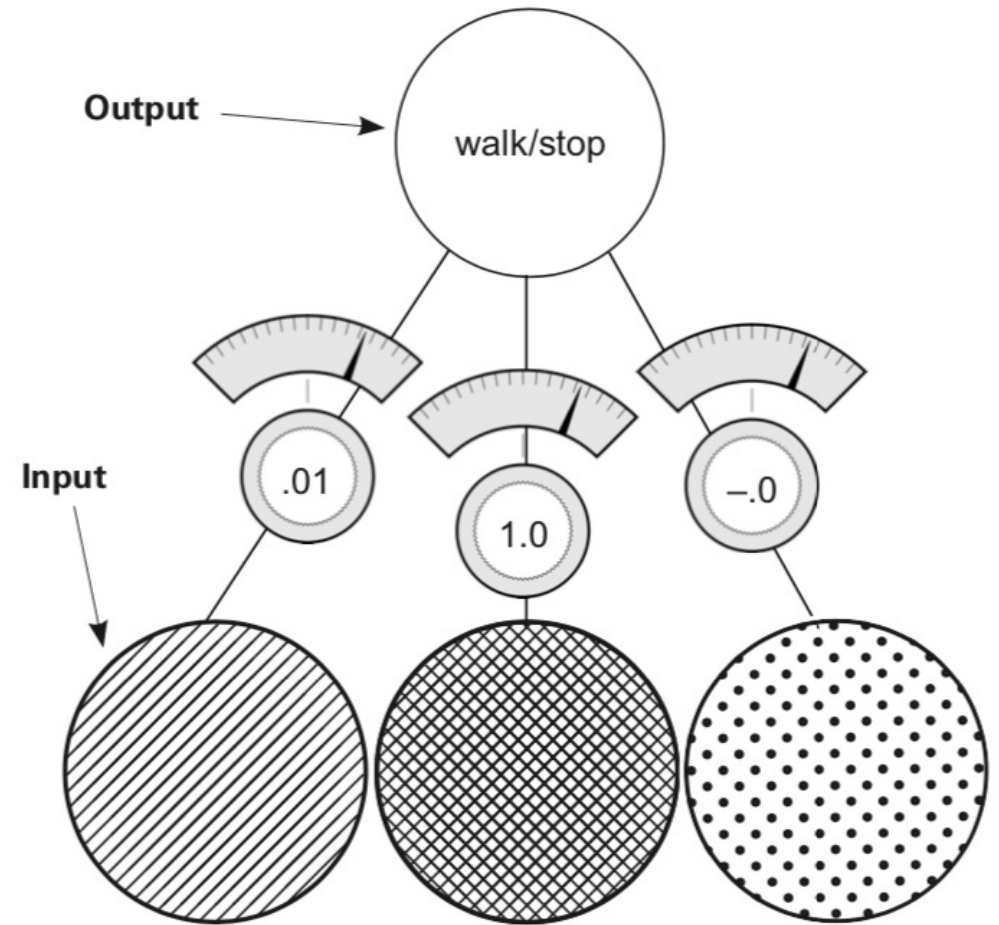
## What did the last neural network learn?

All it was trying to do was identify which input (of the three possible) correlated with the output.

It correctly identified the middle light by analyzing the final weight positions of the network.

How did the network identify correlation? Well, in the process of gradient descent, each training example asserts either *up pressure* or *down pressure* on the weights.

On average, there was more up pressure for the middle weight and more down pressure for the other weights.



# Up and down pressure

It comes from the data.

Each node is individually trying to correctly predict the output given the input.

For the most part, each node ignores all the other nodes when attempting to do so.

The only *cross communication* occurs in that all three weights must share the same error measure.

The *weight update* is nothing more than taking this shared error measure and multiplying it by each respective input.



# Up and down pressure

It comes from the data.

A key part of why neural networks learn is *error attribution*, which means given a shared error, the network needs to figure out which weights contributed (so they can be adjusted) and which weights did *not* contribute (so they can be left alone).

Training data				Weight pressure					
1	0	1	→	0	-	0	-	→	0
0	1	1	→	1	0	+	+	→	1
0	0	1	→	0	0	0	-	→	0
1	1	1	→	1	+	+	+	→	1
0	1	1	→	1	0	+	+	→	1
1	0	1	→	0	-	0	-	→	0

Consider the first training example. Because the middle input is 0, the middle weight is *completely irrelevant* for this prediction.

No matter what the weight is, it's going to be multiplied by 0 (the input).

Thus, any error at that training example (regardless of whether it's too high or too low), can be *attributed* to only the far-left and right weights.

Consider the pressure of this first training example. If the network should predict 0, and two inputs are 1s, then this will cause error, which drives the weight values *toward 0*.

# Up and down pressure

It comes from the data.

Training data				Weight pressure					
1	0	1	→	0	-	0	-	→	0
0	1	1	→	1	0	+	+	→	1
0	0	1	→	0	0	0	-	→	0
1	1	1	→	1	+	+	+	→	1
0	1	1	→	1	0	+	+	→	1
1	0	1	→	0	-	0	-	→	0

The Weight Pressure table helps describe the effect of each training example on each respective weight. + indicates that it has pressure toward 1, and – indicates that it has pressure toward 0. Zeros (0) indicate that there is no pressure because the input datapoint is 0, so that weight won't be changed.

Notice that the far-left weight has two negatives and one positive, so on average the weight will move toward 0. The middle weight has three positives, so on average the weight will move toward 1.

Training data				Weight pressure					
1	0	1	→	0	-	0	-	→	0
0	1	1	→	1	0	+	+	→	1
0	0	1	→	0	0	0	-	→	0
1	1	1	→	1	+	+	+	→	1
0	1	1	→	1	0	+	+	→	1
1	0	1	→	0	-	0	-	→	0

Each individual weight is attempting to compensate for error.

In the first training example, there's discorrelation between the far-right and far-left inputs and the desired output. **This causes those weights to experience down pressure.**

This same phenomenon occurs throughout all six training examples, rewarding correlation with pressure toward 1 and penalizing decorrelation with pressure toward 0.

On average, this causes the network to find the correlation present between the middle weight and the output to be the dominant predictive force (heaviest weight in the weighted average of the input), making the network quite accurate.

## Edge case: Overfitting

**Sometimes correlation happens accidentally.**

Consider again the first example in the training data. What if the far-left weight was 0.5 and the far-right weight was  $-0.5$ ? Their prediction would equal 0. The network would predict perfectly. But it hasn't remotely learned how to safely predict streetlights (those weights would fail in the real world). This phenomenon is known as *overfitting*.

### Deep learning's greatest weakness: Overfitting

Error is shared among all the weights. If a particular configuration of weights *accidentally* creates perfect correlation between the prediction and the output dataset (such that `error == 0`) without giving the heaviest weight to the best inputs, *the neural network will stop learning*.

## Edge case: Overfitting

Sometimes correlation happens accidentally.

### Deep learning's greatest weakness: Overfitting

Error is shared among all the weights. If a particular configuration of weights *accidentally* creates perfect correlation between the prediction and the output dataset (such that `error == 0`) without giving the heaviest weight to the best inputs, *the neural network will stop learning*.

If it wasn't for the other training examples, this fatal flaw would cripple the neural network.

What do the other training examples do? Well, let's look at the second training example. It bumps the far-right weight upward while not changing the far-left weight. This throws off the equilibrium that stopped the learning in the first example.

As long as you don't train exclusively on the first example, the rest of the training examples will help the network avoid getting stuck in these edge-case configurations that exist for any one training example.

This is *very important*. Neural networks are so flexible that they can find many, many different weight configurations that will correctly predict for a subset of training data.

If you trained this neural network on the first two training examples, it would likely stop learning at a point where it did *not* work well for the other training examples.

In essence, it memorized the two training examples instead of finding the *correlation* that will *generalize* to any possible streetlight configuration.

If you train on only two streetlights and the network finds just these edge-case configurations, it could *fail* to tell you whether it's safe to cross the street when it sees a streetlight that wasn't in the training data.

### Key takeaway

The greatest challenge you'll face with deep learning is convincing your neural network to *generalize* instead of just *memorize*. You'll see this again.

# Edge case: Conflicting pressure

## Sometimes correlation fights itself.

Consider the far-right column in the following Weight Pressure table. What do you see?

This column seems to have an equal number of upward and downward pressure moments. But the network correctly pushes this (far-right) weight down to 0, which means the downward pressure moments must be larger than the upward ones. How does this work?

Training data				Weight pressure					
1	0	1	→	0	-	0	-	→	0
0	1	1		1	0	+	+		1
0	0	1	→	0	0	0	-	→	0
1	1	1		1	+	+	+		1
0	1	1		1	0	+	+		1
1	0	1	→	0	-	0	-	→	0

The left and middle weights have enough signal to converge on their own. The left weight falls to 0, and the middle weight moves toward 1. As the middle weight moves higher and higher, the error for positive examples continues to decrease. But as they approach their optimal positions, the decorrelation on the far-right weight becomes more apparent.



Let's consider the extreme example, where the left and middle weights are perfectly set to 0 and 1, respectively. What happens to the network? If the right weight is above 0, then the network predicts too high; and if the right weight is beneath 0, the network predicts too low.

As other nodes learn, they absorb some of the error; they absorb part of the correlation. They cause the network to predict with *moderate* correlative power, which reduces the error. The other weights then only try to adjust their weights to correctly predict what's left.

In this case, because the middle weight has consistent signal to absorb all the correlation (because of the 1:1 relationship between the middle input and the output), the error when you want to predict 1 becomes very small, but the error to predict 0 becomes large, pushing the middle weight downward.

## It doesn't always work out like this.

In some ways, you kind of got lucky. If the middle node hadn't been so perfectly correlated, the network might have struggled to silence the far-right weight. Later you'll learn about *regularization*, which forces weights with conflicting pressure to move toward 0.

As a preview, regularization is advantageous because if a weight has equal pressure upward and downward, it isn't good for anything. It's not helping either direction. In essence, regularization aims to say that only weights with really strong correlation can stay on; everything else should be silenced because it's contributing noise. It's sort of like natural selection, and as a side effect it would cause the neural network to train faster (fewer iterations) because the far-right weight has this problem of both positive and negative pressure.

In this case, because the far-right node isn't definitively correlative, the network would immediately start driving it toward 0. Without regularization (as you trained it before), you won't end up learning that the far-right input is useless until after the left and middle start to figure out their patterns. More on this later.

If networks look for correlation between an input column of data and the output column, what would the neural network do with the following dataset?

Training data				Weight pressure					
1	0	1	→	1	+	0	+	→	1
0	1	1	→	1	0	+	+	→	1
0	0	1	→	0	0	0	-	→	0
1	1	1	→	0	-	-	-	→	0

There is no correlation between any input column and the output column. Every weight has an equal amount of upward pressure and downward pressure. *This dataset is a real problem for the neural network.*

Previously, you could solve for input datapoints that had both upward and downward pressure because other nodes would start solving for either the positive or negative predictions, drawing the balanced node to favor up or down. But in this case, all the inputs are equally balanced between positive and negative pressure. What do you do?

## Learning indirect correlation

**If your data doesn't have correlation, create intermediate data that does!**

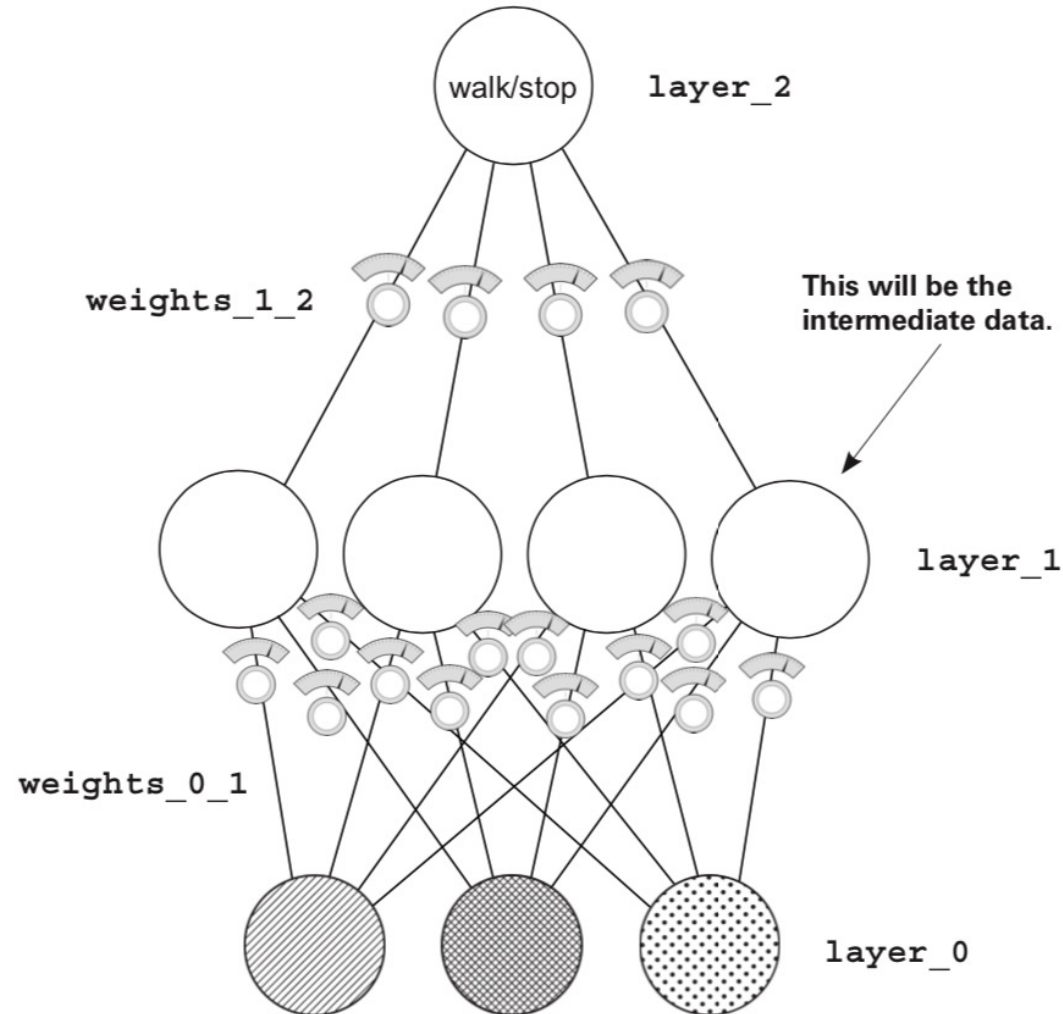
Unfortunately, this is a new streetlights dataset that has *no correlation* between the input and output. The solution is simple: use two of these networks. The first one will create an intermediate dataset that has limited correlation with the output, and the second will use that limited correlation to correctly predict the output.

Because the input dataset doesn't correlate with the output dataset, you'll use the input dataset to create an intermediate dataset that *does* have correlation with the output. It's kind of like cheating.



# Creating correlation

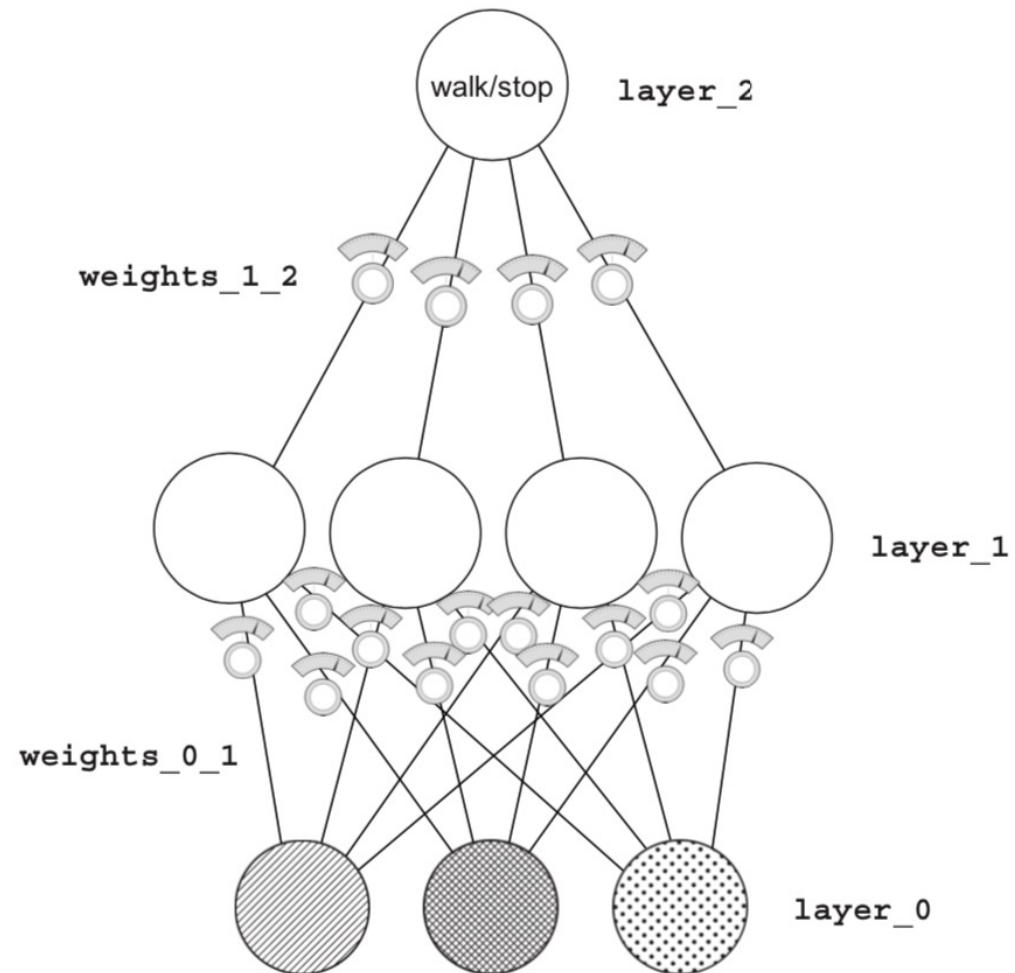
Here's a picture of the new neural network. You basically stack two neural networks on top of each other. The middle layer of nodes (`layer_1`) represents the *intermediate dataset*. The goal is to train this network so that even though there's no correlation between the input dataset and output dataset (`layer_0` and `layer_2`), the `layer_1` dataset that you create *using* `layer_0` will have correlation with `layer_2`.



# Stacking neural networks: A review

**Chapter 3 briefly mentioned stacked neural networks.  
Let's review.**

When you look at the following architecture, the prediction occurs exactly as you might expect when I say, "Stack neural networks." The output of the first lower network (`layer_0` to `layer_1`) is the input to the second upper neural network (`layer_1` to `layer_2`). The prediction for each of these networks is identical to what you saw before.



As you start to think about how this neural network learns, you already know a great deal. If you ignore the lower weights and consider their output to be the training set, the top half of the neural network (`layer_1` to `layer_2`) is just like the networks trained in the preceding chapter. You can use all the same learning logic to help them learn.

The part that you don't yet understand is how to update the weights between `layer_0` and `layer_1`. What do they use as their error measure? As you may remember from chapter 5, the cached/normalized error measure is called `delta`. In this case, you want to figure out how to know the `delta` values at `layer_1` so they can help `layer_2` make accurate predictions.