CS 330 - Artificial Intelligence

- Introduction to Neural Network I

Instructor: Renzhi Cao
Computer Science Department
Pacific Lutheran University
Fall 2018

Special appreciation to Geoffrey Hinton, Ian Goodfellow, Joshua Bengio, Aaron Courville, Michael Nielsen, Andrew Ng, Katie Malone, Sebastian Thrun, Ethem Alpaydin, Christopher Bishop, Tom Mitchell.

Announcement

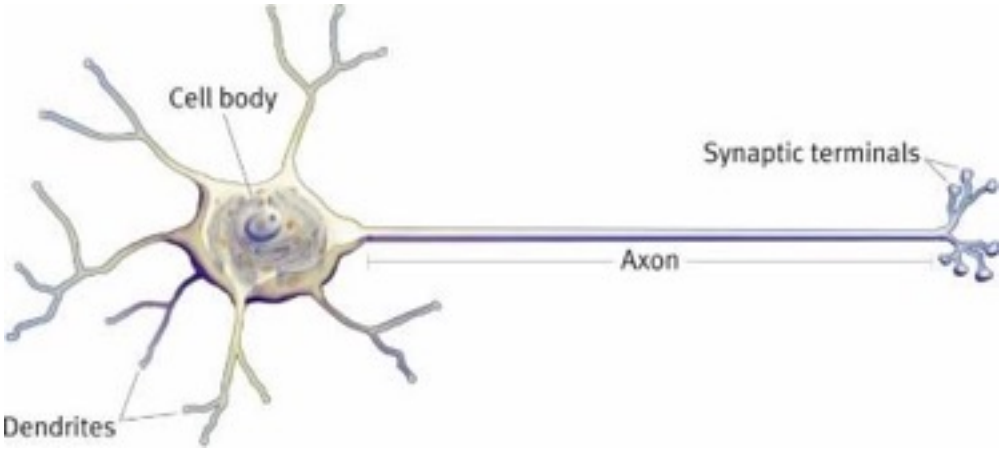
- Go over quiz 3 and home works, check Sakai!
- Literature report comments.
- Second round posted on Sakai. (You need to have the data ready for approval of your final project).

Neural Network

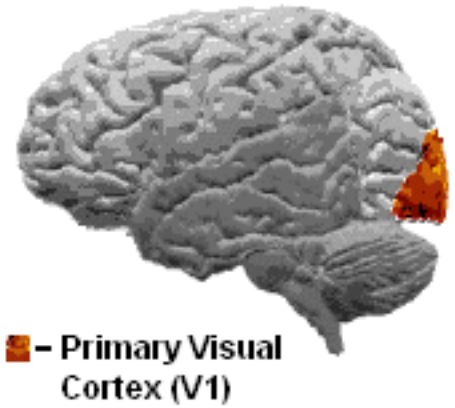
Reference book:

Neural networks and deep learning, Michael Nielsen, Jan 2017.

504/92



140 millions neurons



https://en.wikipedia.org/wiki/Visual_cortex

- The **MNIST** database (Mixed National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems

- We're focusing on handwriting recognition because it's an excellent prototype problem for learning about neural networks in general.

- We'll write a computer program implementing a neural network that learns to recognize handwritten digits

Overview of Neural Network Architecture

- Feed-forward Neural Network
- Recurrent Neural Network

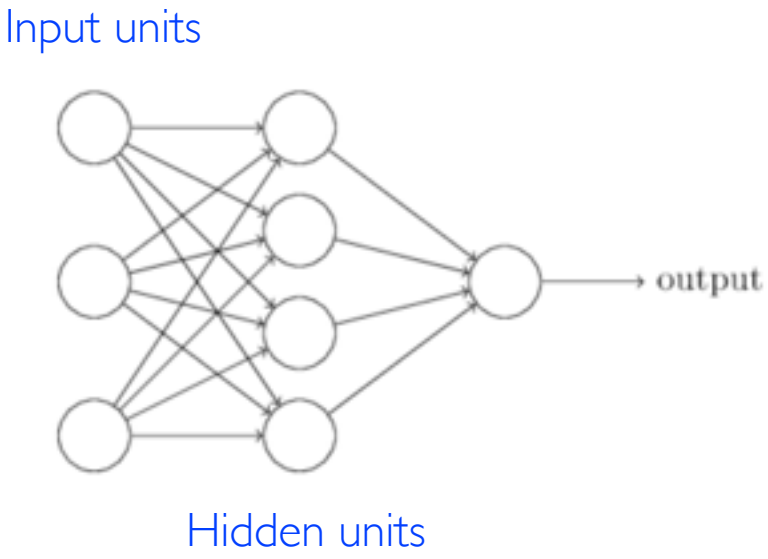
Feed-forward Neural Network

This is the commonest type of neural network in practical application

- The first layer is the input and last layer is the output
- If there is more than one hidden layer, we call them "deep" neural network

They compute a series of transformations that change the similarities between cases.

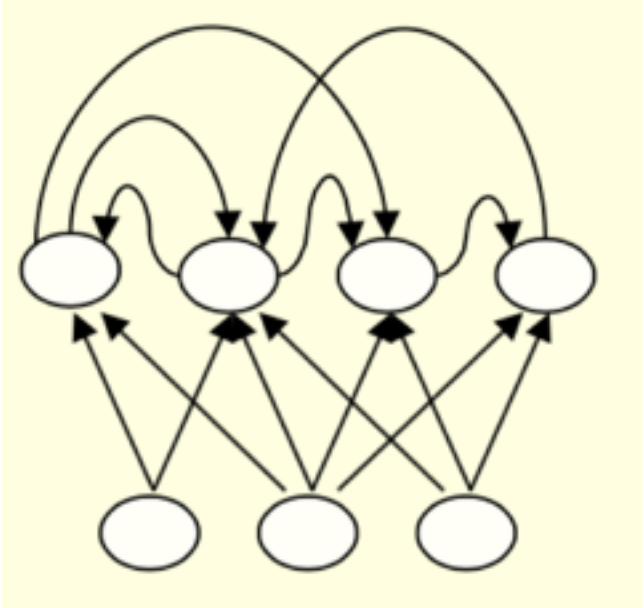
- The activities of the neurons in each layer are a non-linear function of the activities in the previous layer



Recurrent Neural Network

This has directed cycles in their connection graph

- That means you can sometimes get back to where you started by following the arrow.



They can have complicated dynamics and this can make them very difficult to train

- There is a lot of interest at present in finding efficient ways of training recurrent nets.

They are more biologically realistic

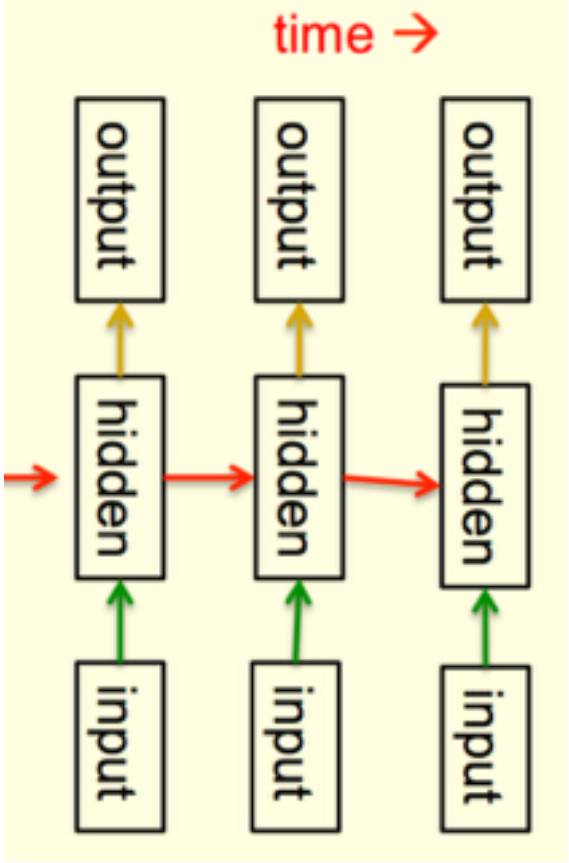
Recurrent Neural Network

Recurrent neural networks are a very natural way to model sequential data

- They are equivalent to very deep nets with one hidden layer per time slice.
- Except that they use the same weights at every time slice and they get input at every time slice

They have the ability to remember information in their hidden state for a long time

- But it's very hard to train them use this potential.



Example of Recurrent Neural Network

Andrej Karpathy released code for Multi-layer Recurrent Neural Networks (LSTM, GRU, RNN) for character-level language models in Torch

- Here is the link on Github: https://github.com/karpathy/char-rnn

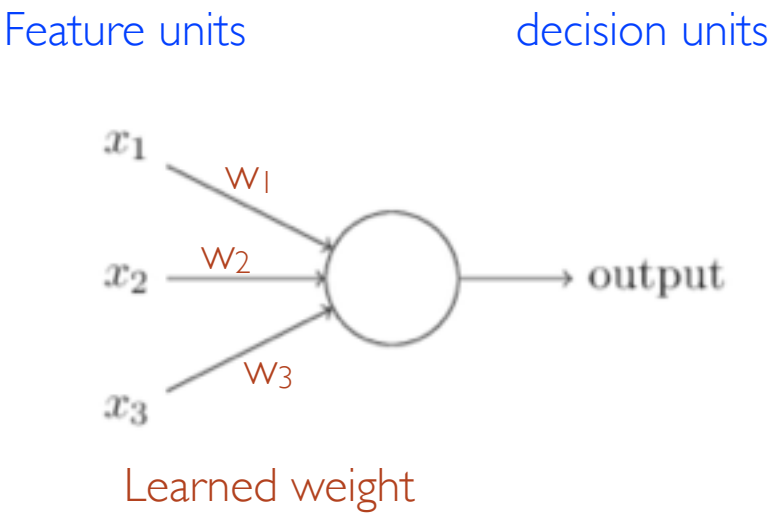
After training for several hours on a subset of works of Shakespeare, we could use the trained model to sample new text (start with “The meaning of life is “):

- th sample.lua cv/lm_lstm_epoch14.18_1.4002.t7 -gpuid -1 -primetext "the meaning of life is" -temperature 0.6 -length 100
- Demo by Dr. Cao on /home/caora/GitHub/char-rnn

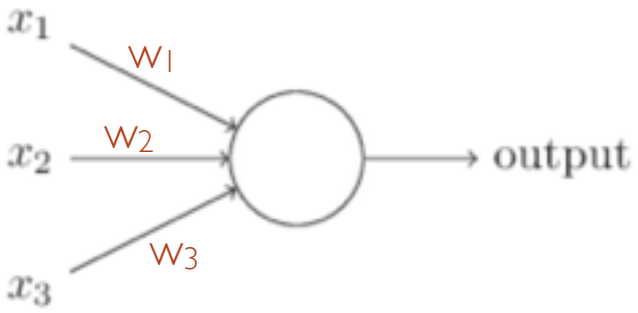
The first generation of neural network

Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts.

- 1. Convert raw input vector into a feature vector
- 2. Learn how to weigh each of the feature activations o get a single scalar quantity
- 3. If this quantity is above some threshold, decide that the input vector is a positive example of the target class



Feature units decision units



Learned weight

1. They appeared to have a very powerful learning algorithm in 1960's.

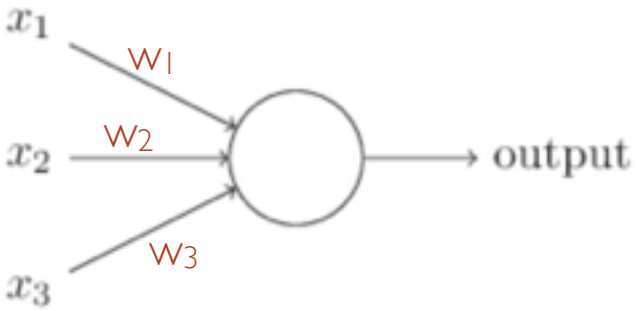
2. Lots of grand claims were made for that they could learn to do.

3. In 1969, Minsky and Papert published a book called "Perceptrons" that analyzed what they could do and showed their limitations

- Many people thought these limitations applied to all neural network models.

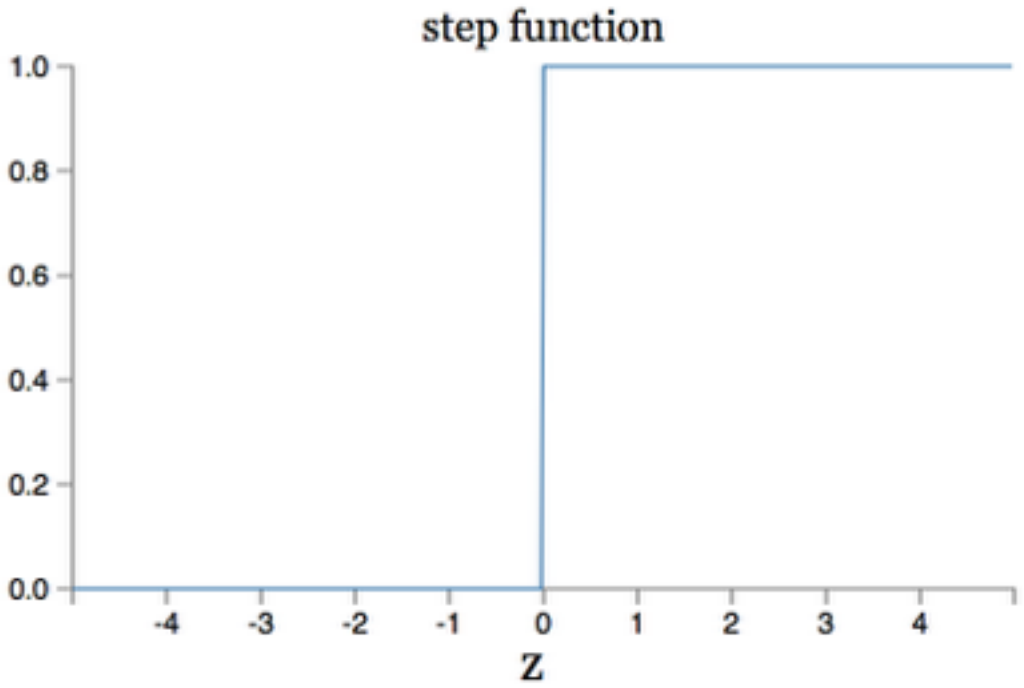
Feature units

decision units



$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Learned weight

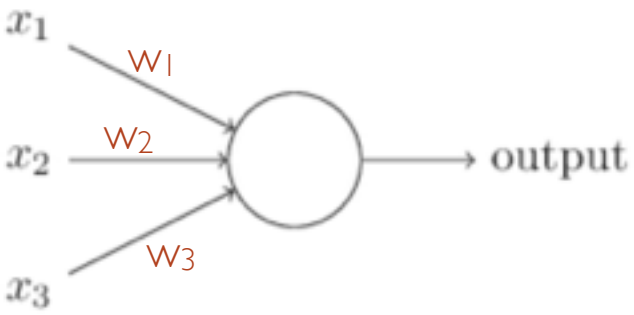


Suppose the weekend is coming up, and you've heard that there's going to be a cheese festival in your city. You like cheese, and are trying to decide whether or not to go to the festival. You might make your decision by weighing up three factors:

1. Is the weather good?
2. Does your partner want to accompany you?
3. Is the festival near public transit? (You don’t own a car).

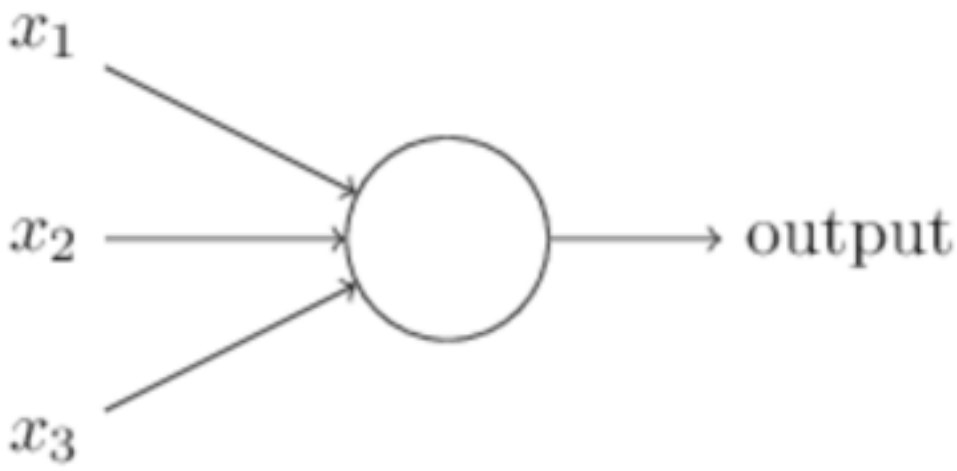
- Can you use decision tree?

Feature units decision units

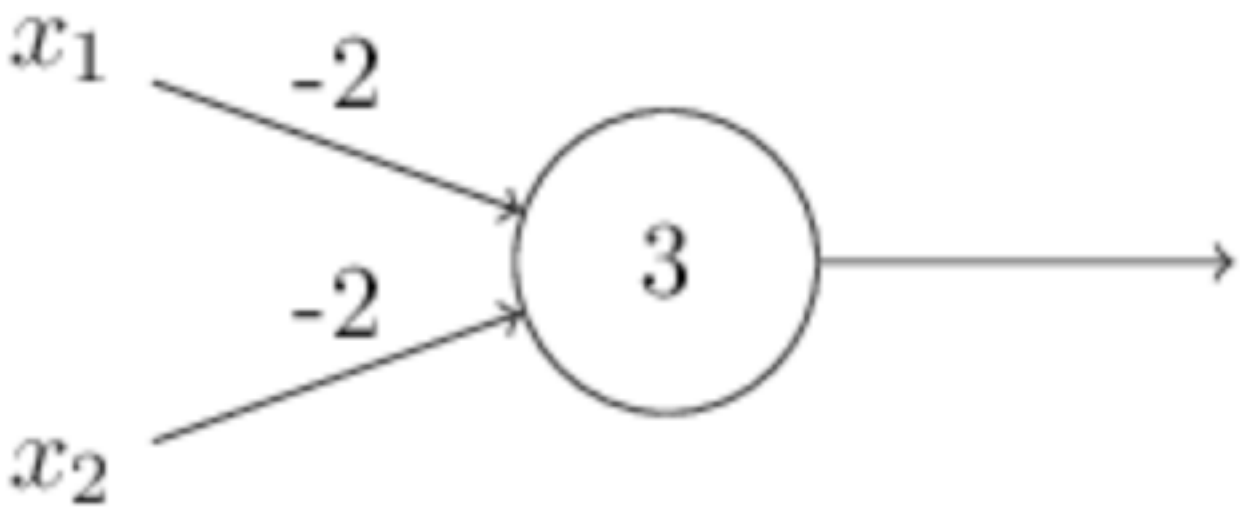


Learned weight

We can represent these three factors by corresponding binary variables x_1, x_2, and x_3. For instance, we'd have $x_1 = 1$ if the weather is good, and $x_1 = 0$ if the weather is bad. Similarly, $x_2 = 1$ if your boyfriend or girlfriend wants to go, and $x_2 = 0$ if not. And similarly again for x_3 and public transit.



$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$



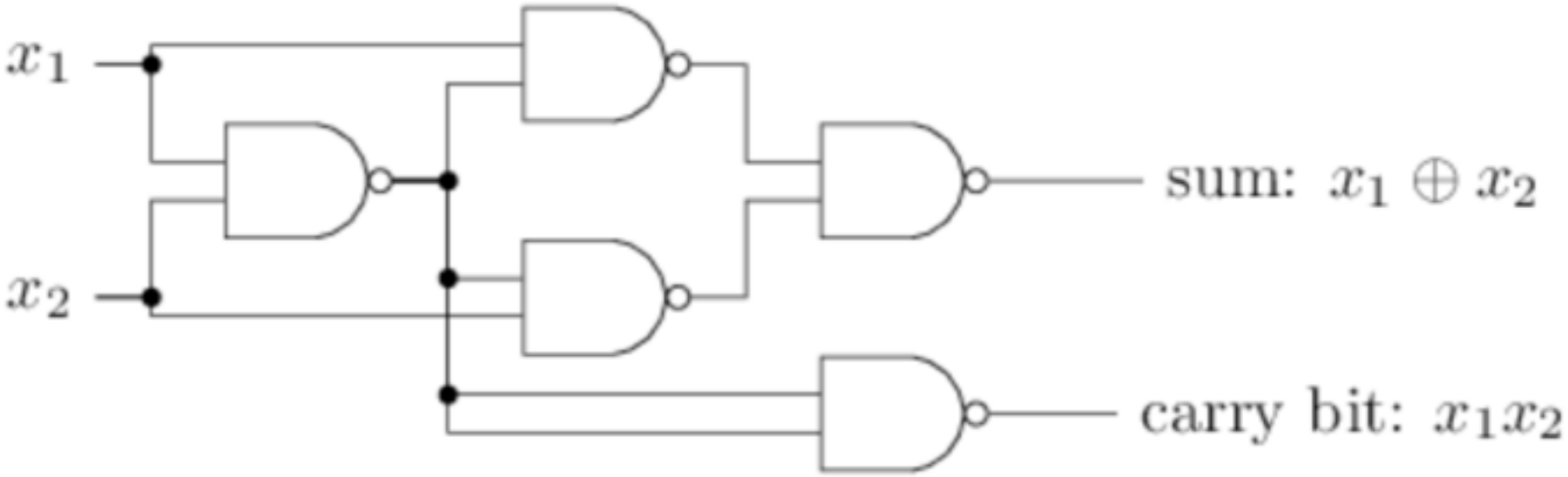
What (0,0) will produce?

1, since 0*(-2) + 0*(-2) + 3 = 3 > 0

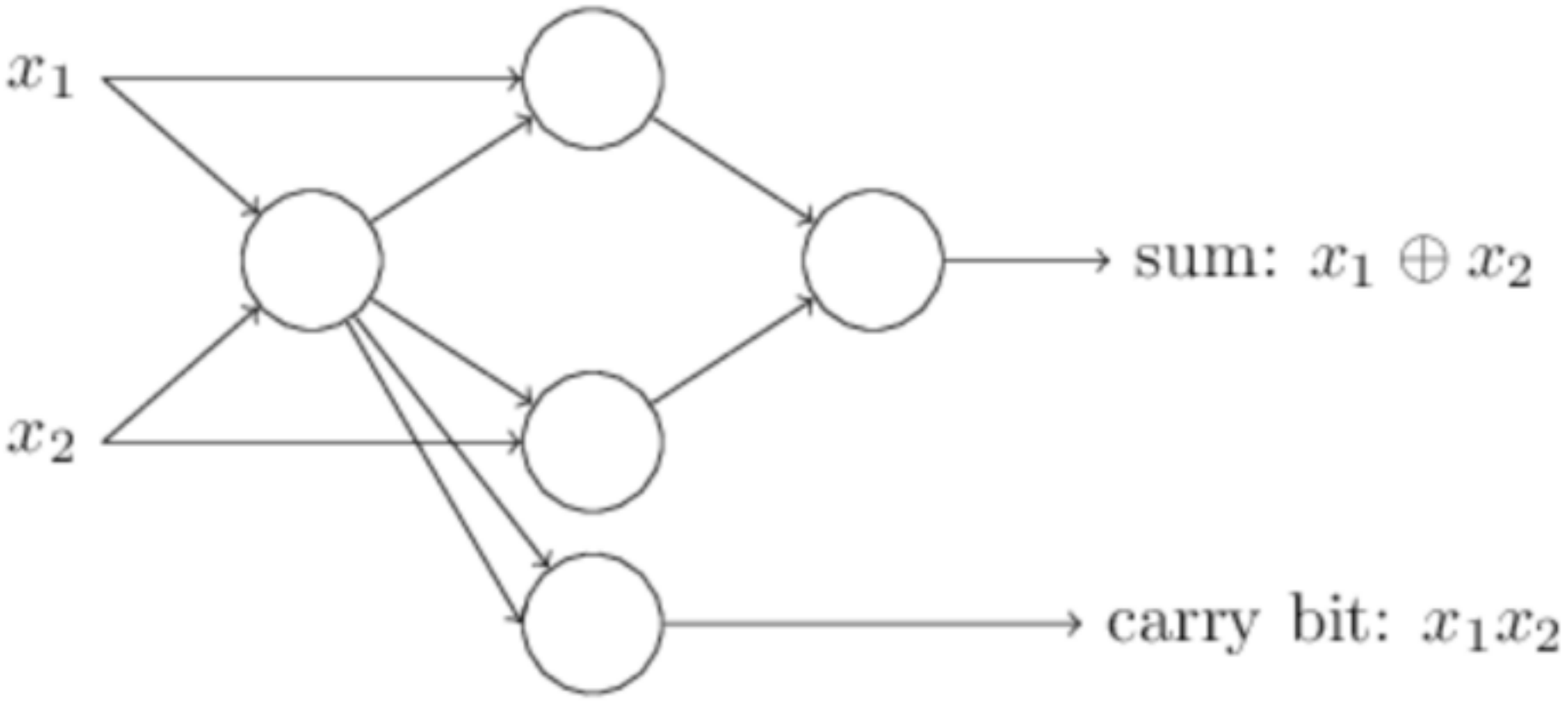
NAND gate

What (1,1) will produce?

0, since 1*(-2) + 1*(-2) + 3 = -1 < 0



use `NAND` gates to build a circuit which adds two bits x1 and x2


x_1
x_2
sum: $x_1 \oplus x_2$
carry bit: $x_1 x_2$

The computational universality of perceptrons is simultaneously reassuring and disappointing. It's reassuring because it tells us that networks of perceptrons can be as powerful as any other computing device. But it's also disappointing, because it makes it seem as though perceptrons are merely a new type of NAND gate. That's hardly big news!

The limitations of Perceptrons

You have two single bit features, can you use perceptrons to tell if they are the same?

- Positive cases (same): (1,1) -> 1; (0,0) ->1
- Negative cases (same): (1,0) -> 0; (0,1) ->0

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

We could use b = -threshold, and rewrite the formula

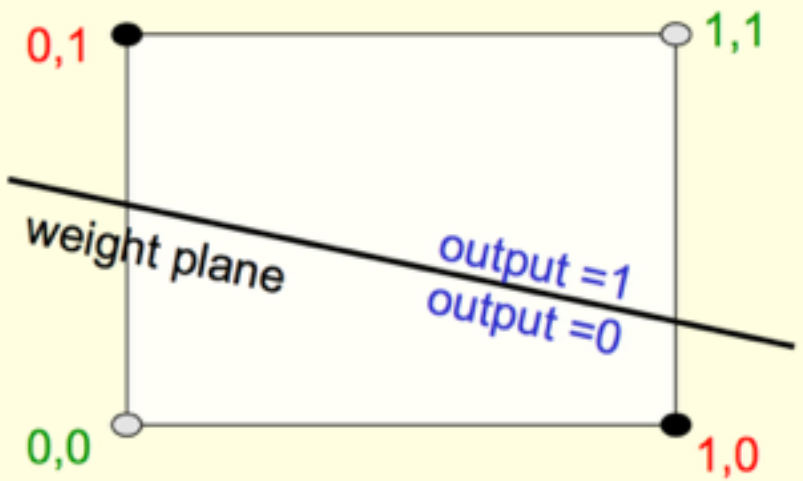
$1 * w_1 + 1 * w_2 + b > 0$, $0 * w_1 + 0 * w_2 + b > 0$

$1 * w_1 + 0 * w_2 + b <= 0$, $0 * w_1 + 1 * w_2 + b <= 0$

The limitations of Perceptrons

Imagine "data-space" in which the axes correspond to components of an input vector.

- Each input vector is a point in this space
- A weight vector defines a plane in data-space
- The weight plan is perpendicular to the weight vector and misses the origin by a distance equal to the threshold



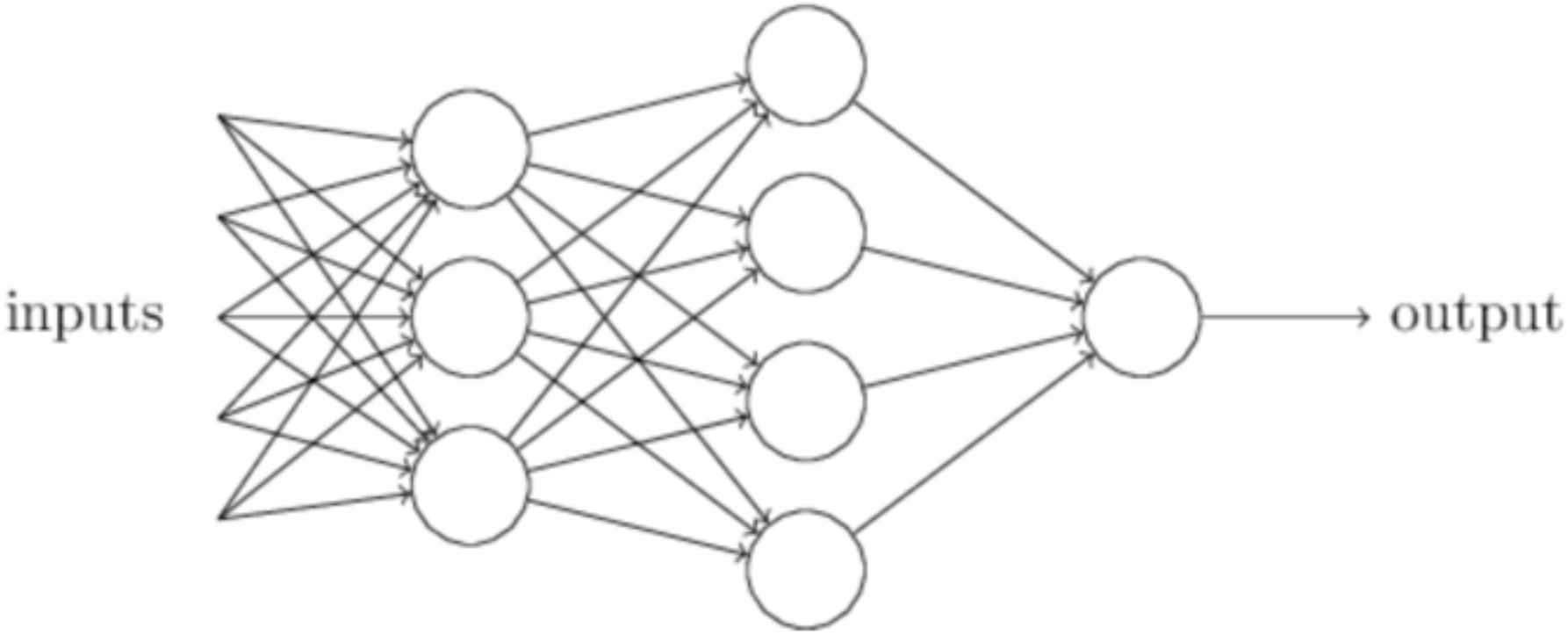
The limitations of Perceptrons

Networks without hidden units are very limited in the input-output mappings they can learn to model

- More layers of linear units do not help. Its still linear

We need multiple layers of adaptive, non-linear hidden units. But how can we train such nets?

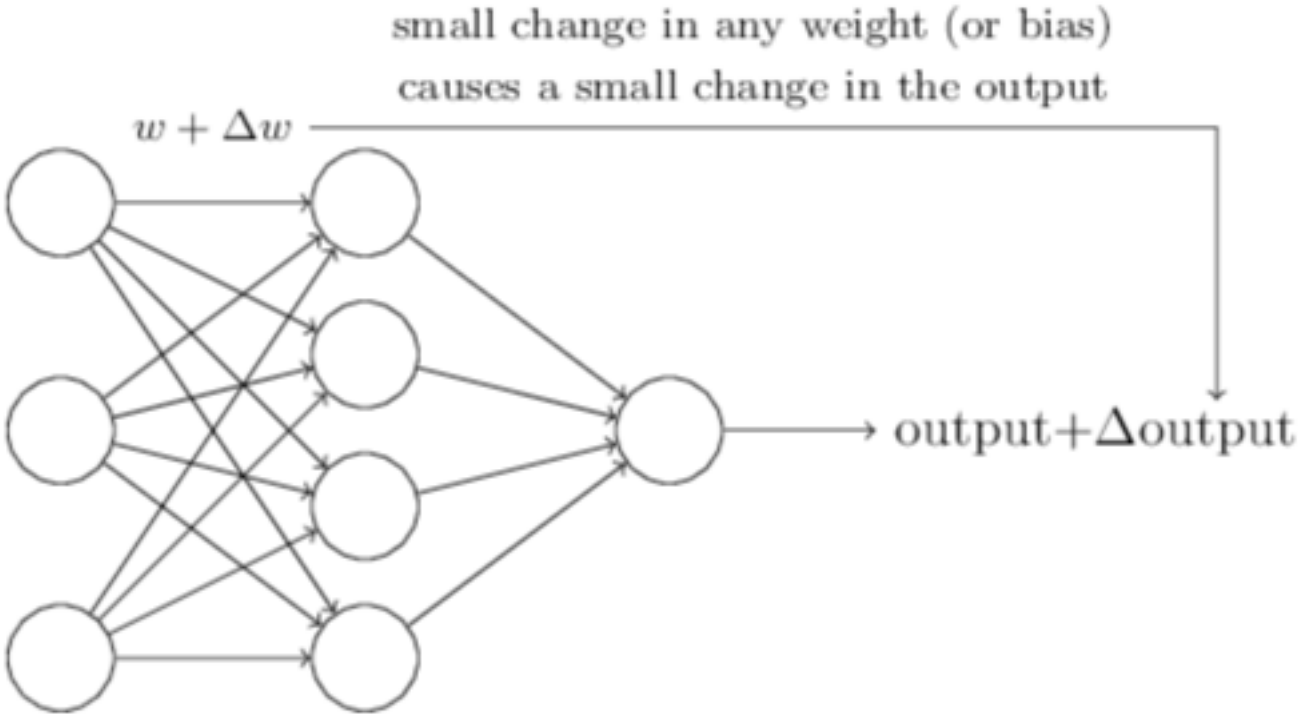
- We need an efficient way of adapting all the weights, not just the last layer. This is hard.
- Learning the weights going into hidden units is equivalent to learning features.
- This is difficult because nobody is telling us directly what the hidden units should do.


inputs
output

Sigmoid neurons

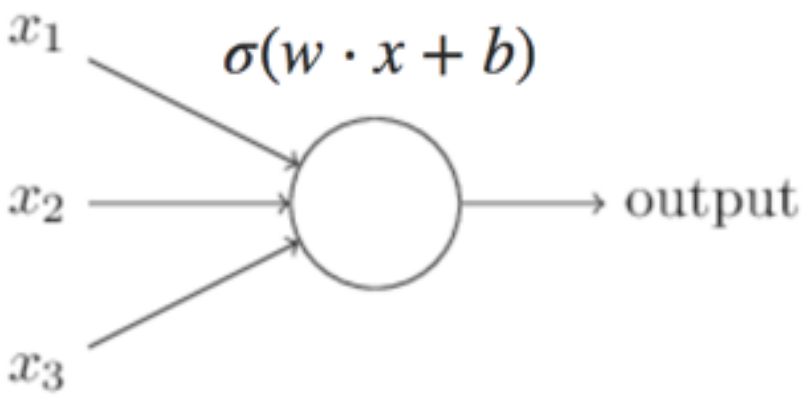
We want networks as follows, when small change in any weight causes small change in output.

- But this isn't what happens when our network contains perceptrons. Small change in the weights or bias of perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1



Sigmoid neurons

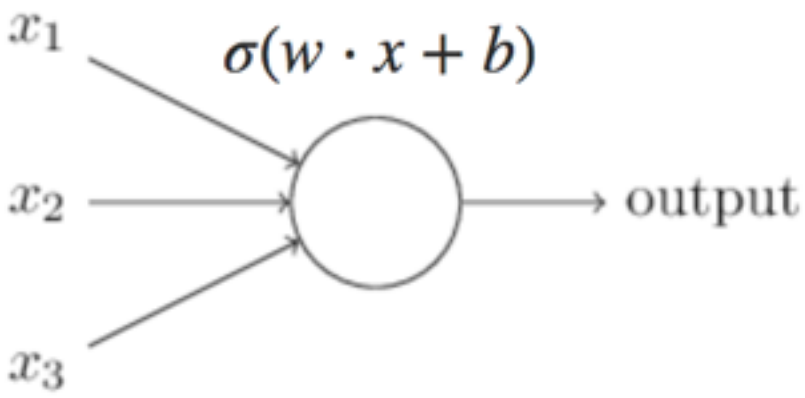
We can overcome this problem by introducing a new type of artificial neuron called a *sigmoid* neuron.



Sigmoid function

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

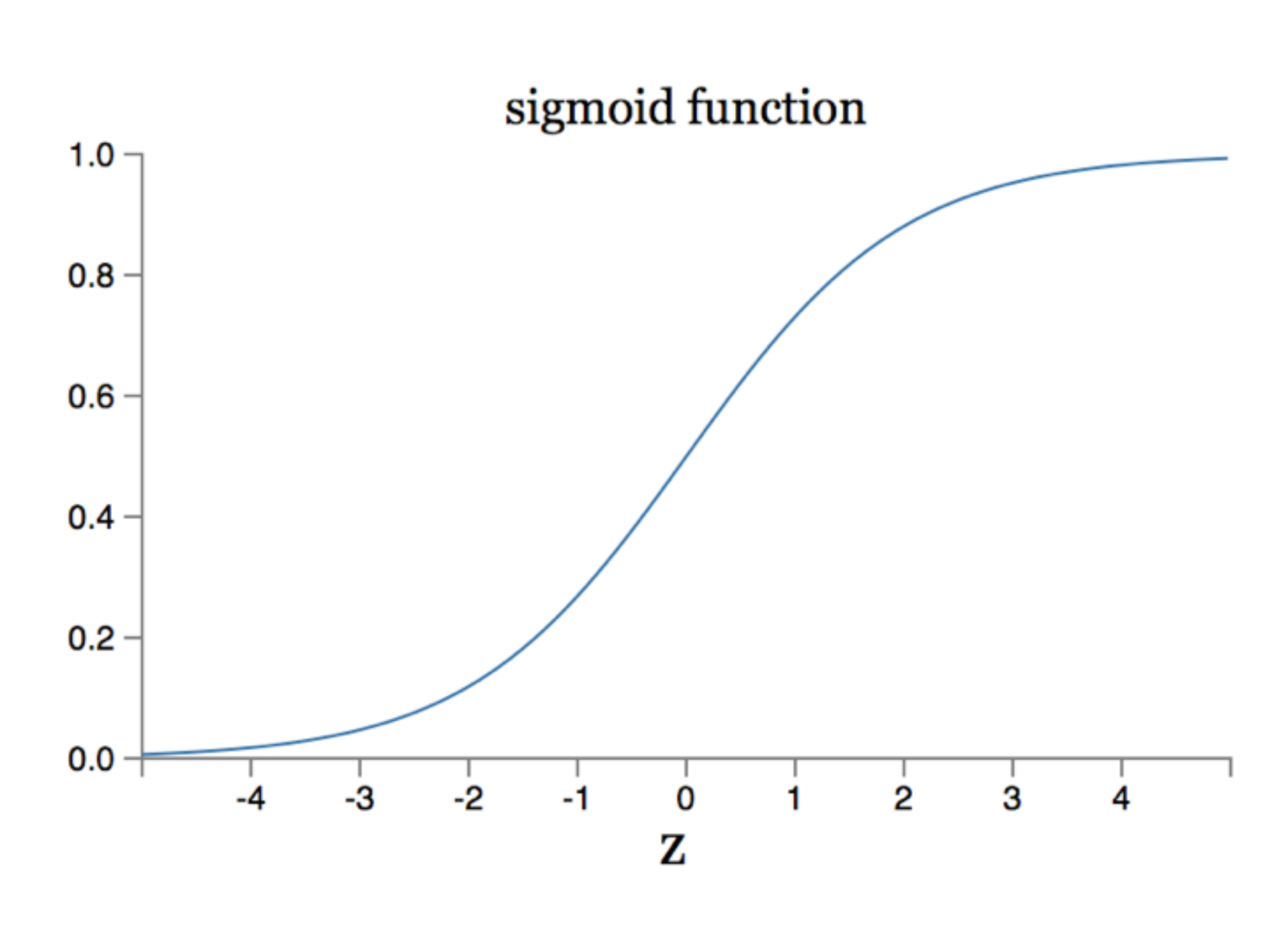
just like a perceptron, the sigmoid neuron has weights for each input, and bias. But the output is not 0 or 1.

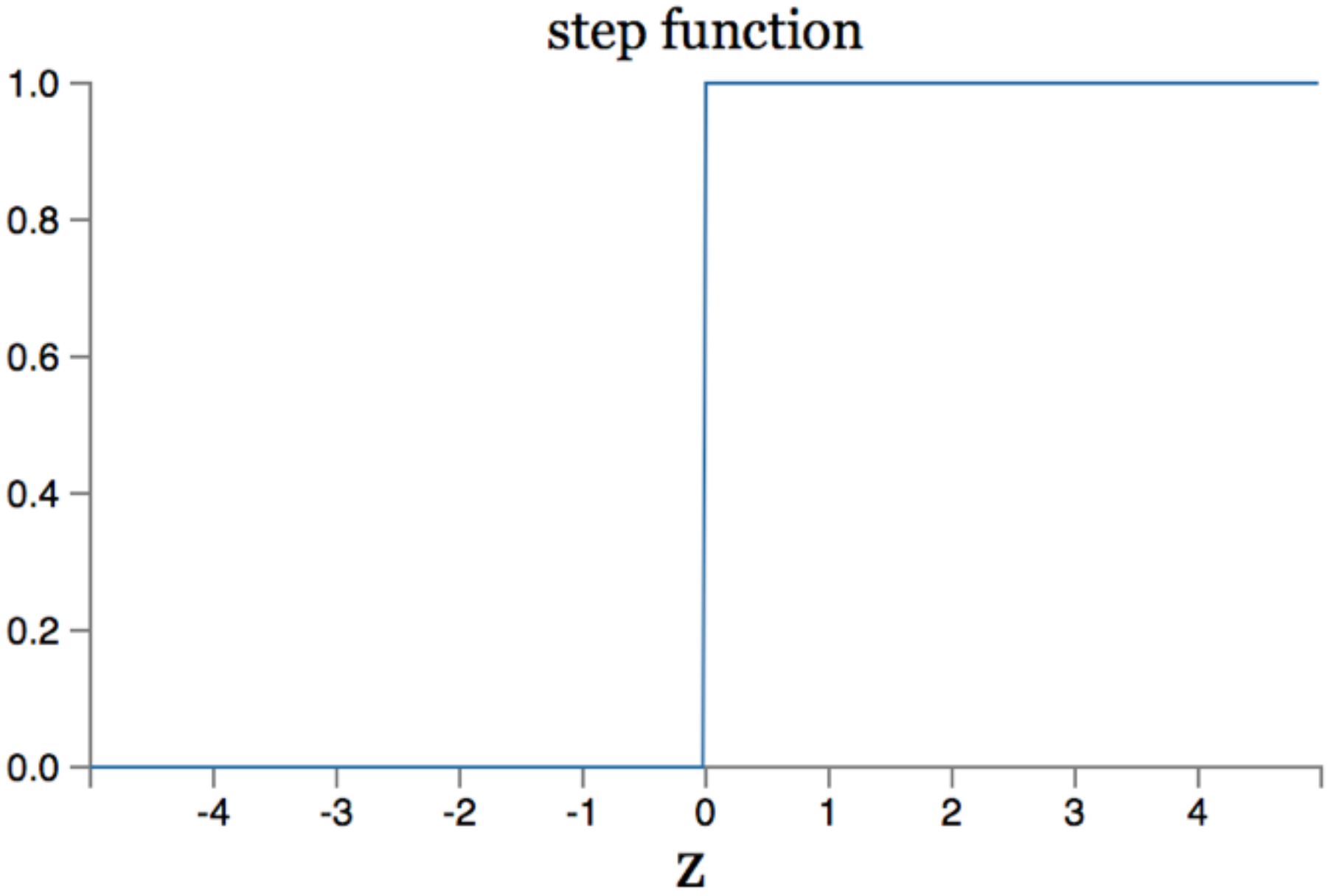


Sigmoid function

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}$$

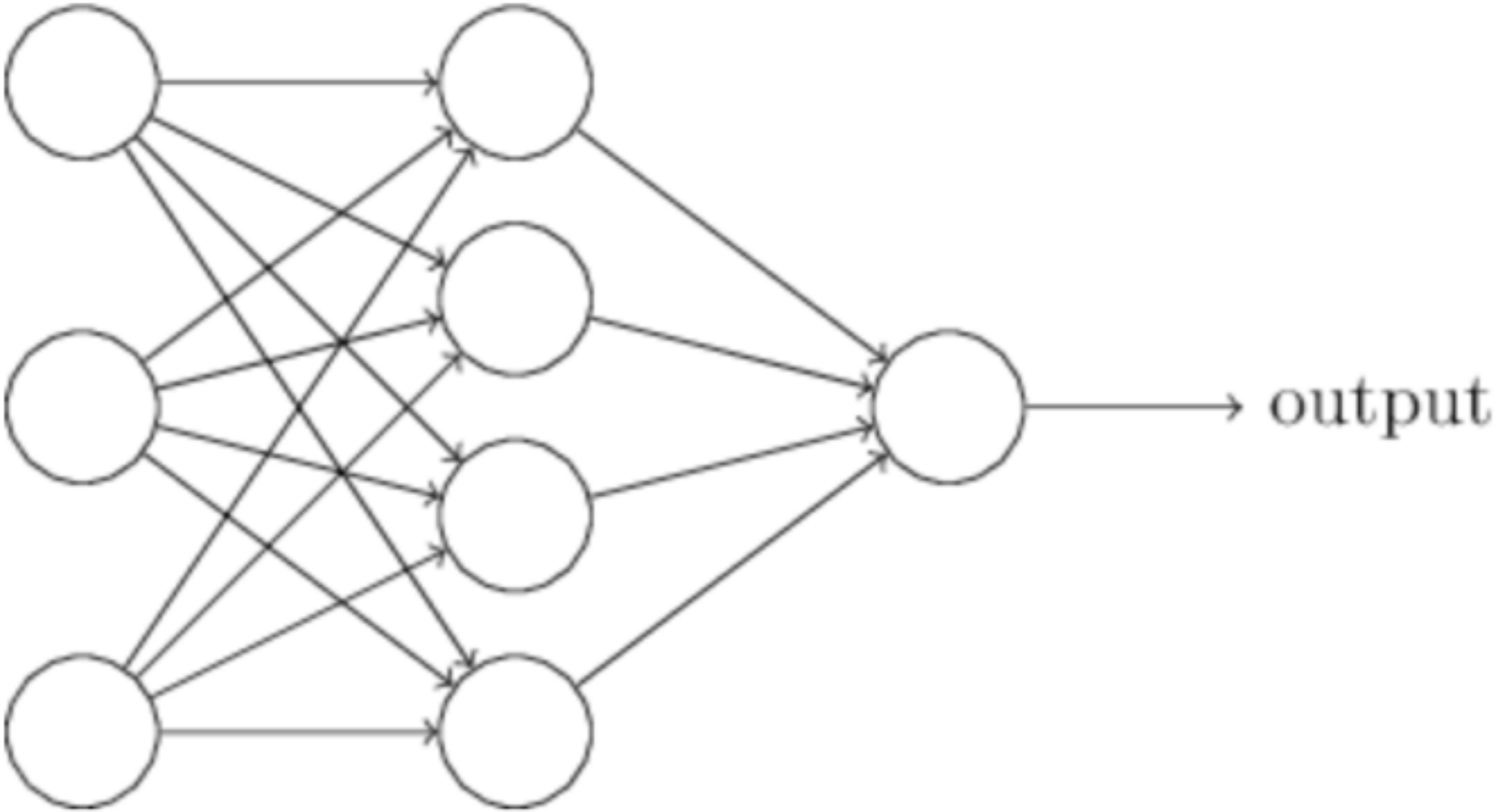

sigmoid function
1.0
0.8
0.6
0.4
0.2
0.0
-4
-3
-2
-1
0
1
2
3
4
Z

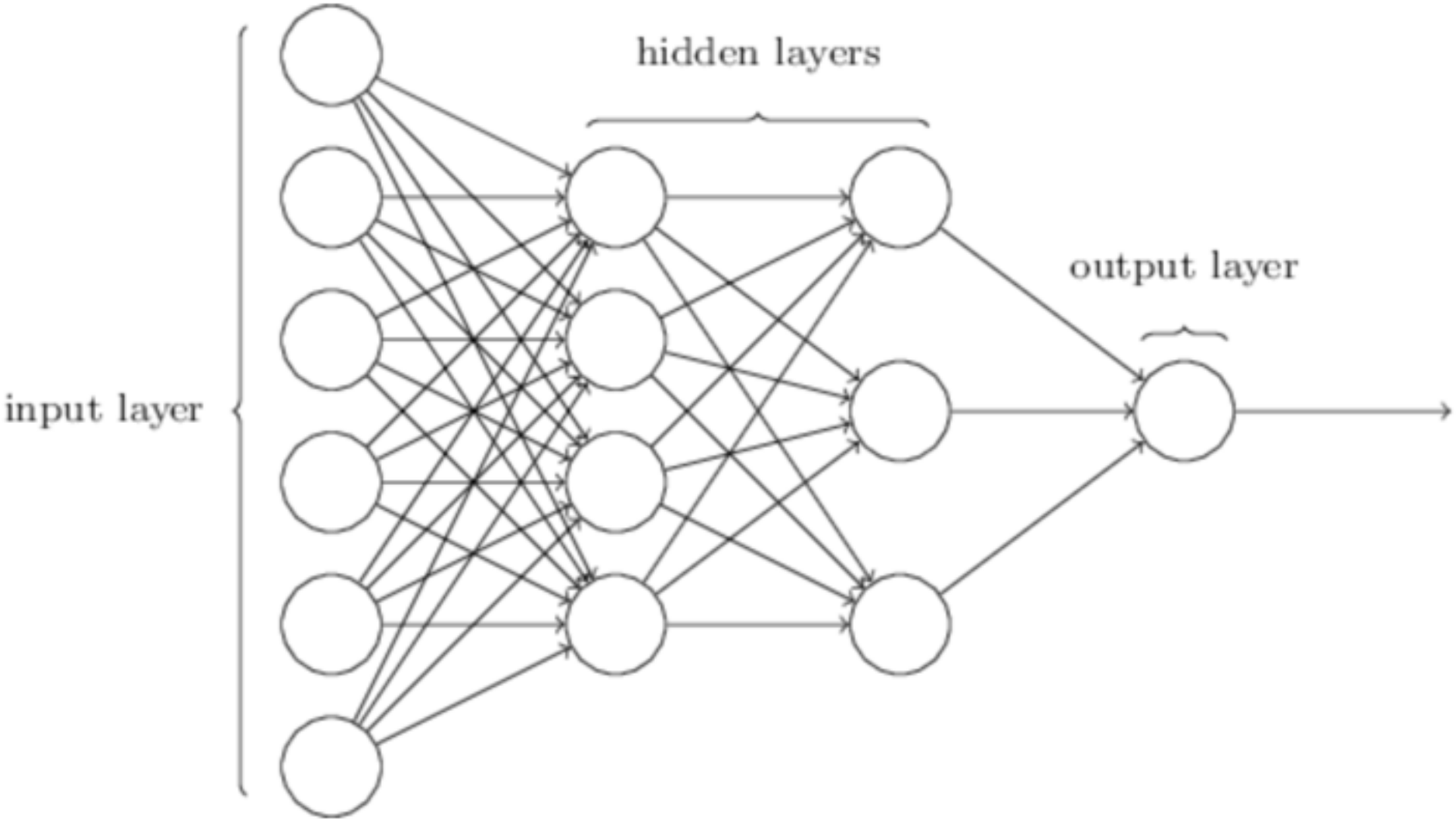

step function
1.0
0.8
0.6
0.4
0.2
0.0
-4
-3
-2
-1
0
1
2
3
4
z

$$\Delta \text{output} \approx \sum_j \frac{\partial \, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \, \text{output}}{\partial b} \Delta b$$

partial derivatives of the output with respect to weight and bias.

The architecture of neural networks





multiple layer networks are sometimes called *multilayer perceptrons* or *MLPs*
But the neurons are sigmoid neurons

1 2 3 … 64

1
2
3

…

64

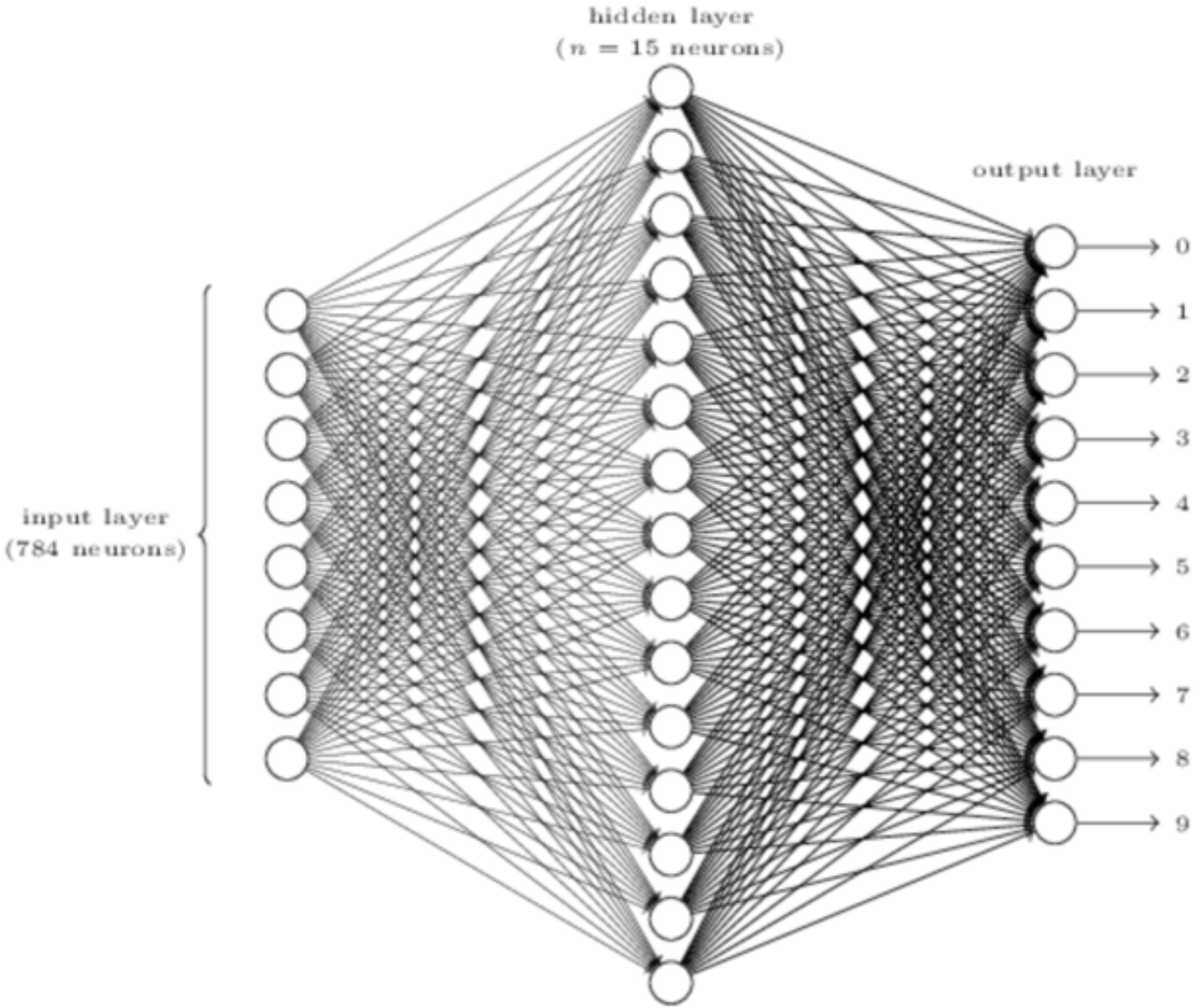
64 * 64 = 4096

A simple network to classify handwritten digits

- Each contains 28 * 28 pixel images of scanned handwritten digits

  28 * 28 = 784

- The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0 representing black.


hidden layer
(n = 15 neurons)
output layer
input layer
(784 neurons)
0
1
2
3
4
5
6
7
8
9

Learning with gradient descent

We have designed the network, what is the next?

training data set MNIST data set

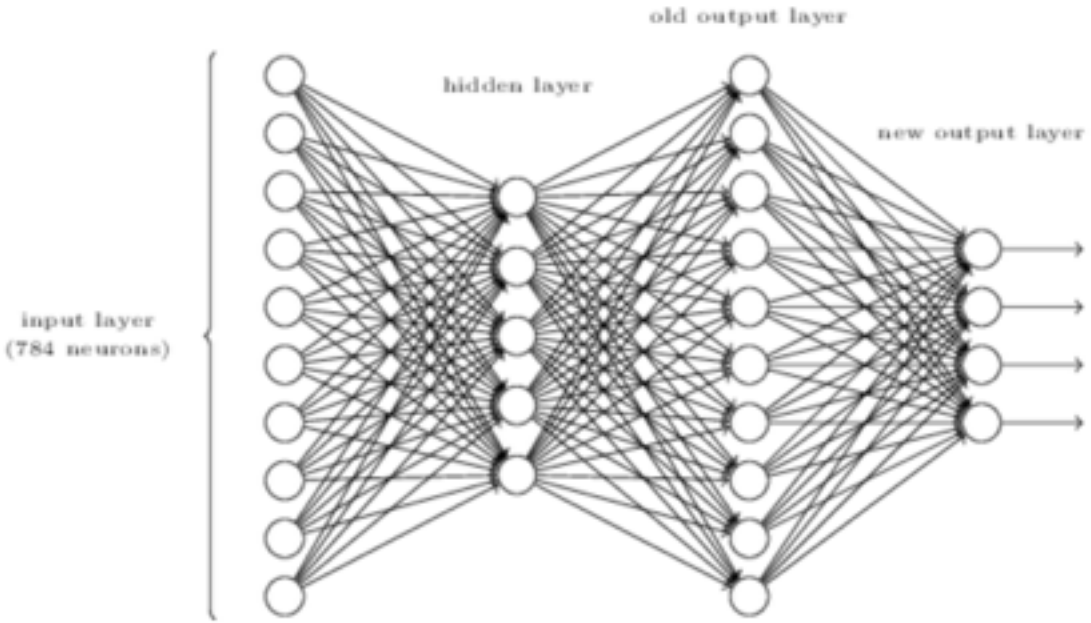
The MNIST data comes in two parts.

- The first part contains 60,000 images to be used as training data
- The second part of the MNIST data set is 10,000 images to be used as test data.

Learning with gradient descent

x as 28×28=784 dimensional vector.

$$y(x) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$$



Learning with gradient descent

Cost function to quantify how well we achieve our goal:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

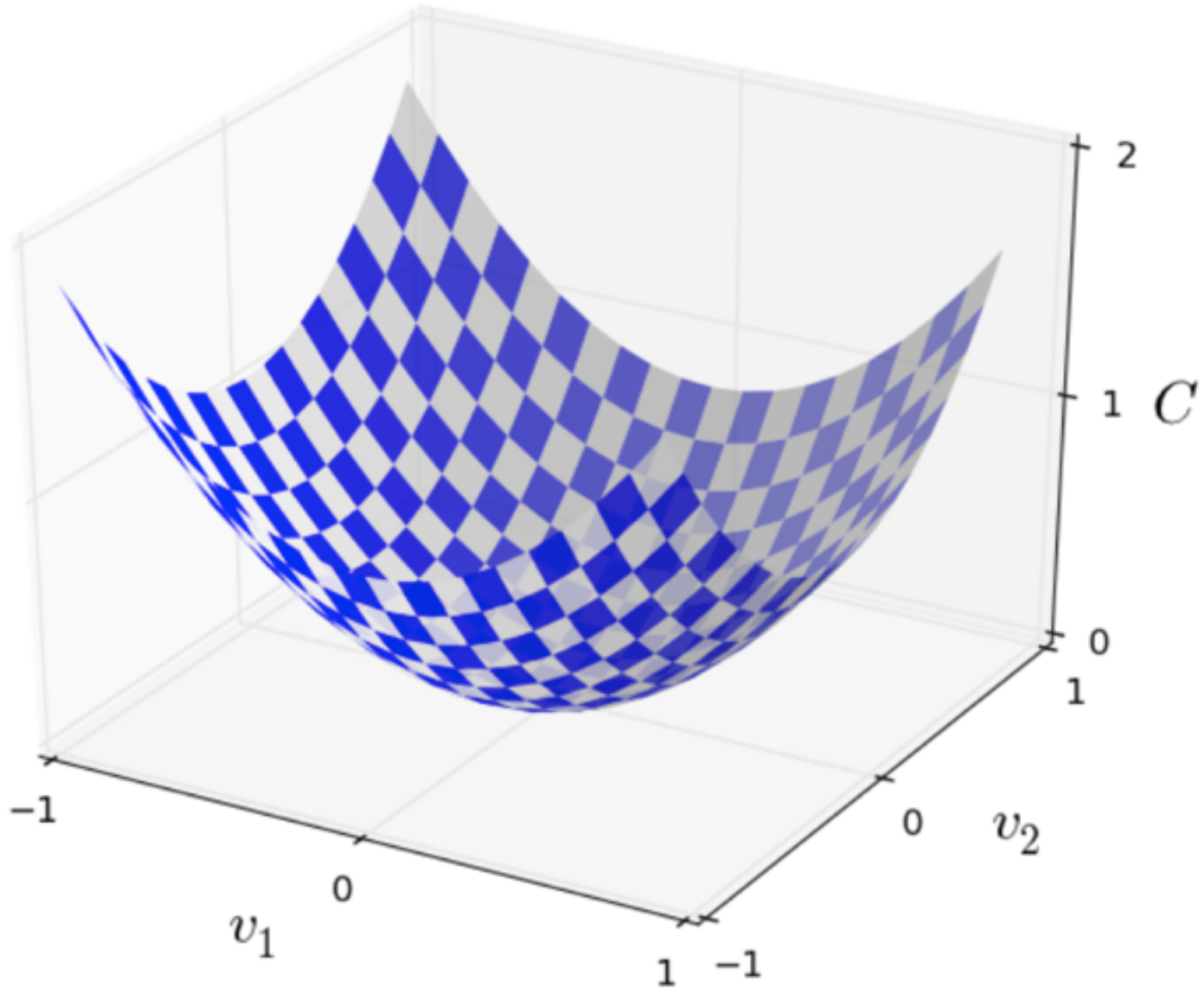
We'll call C the *quadratic* cost function; it's also sometimes known as the *mean squared error* or just *MSE*.

We would like to minimize the cost function, and to minimize C, we use algorithm known as *gradient descent*

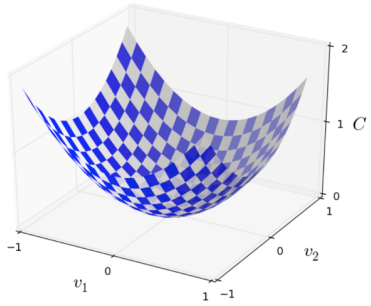
Learning with gradient descent

Now we can forget about neural network, and focus on *minimizing some function C(v)!* And v could be any real-valued function of many variable: $v = v_1, v_2, v_3 \ldots$

We could imagine C as function of just two variables: v_1 *and* v_2.


2
1
C
0
1
0
v_2
−1
−1
0
v_1
1

Learning with gradient descent



How to find the minimum?

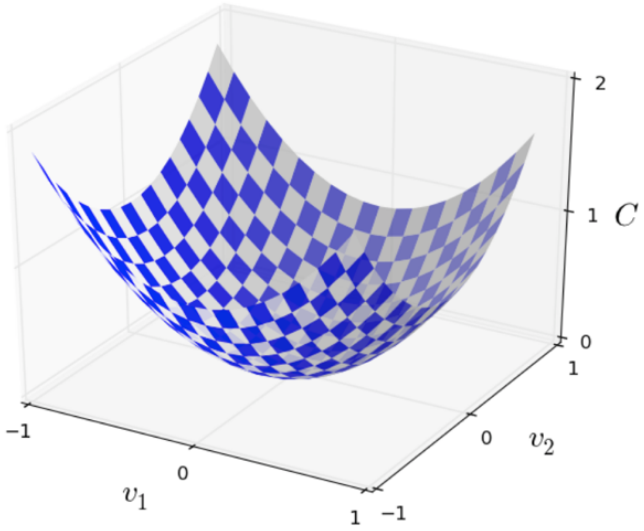
- We could eyeball the graph and find the minimum.
- We could compute derivatives and then try using them to find places where C is an extremum
- Fortunately there is a way to do it.

Imagine a ball rolling down the slope of the valley!

Learning with gradient descent

The calculus tells us:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$



We are going to find some Δv_1 *and* Δv_2, so that ΔC *is negative!*

We denote the gradient vector by ∇C *as:*

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T$$

Learning with gradient descent

$$\Delta C \approx \nabla C \cdot \Delta v.$$

$$\Delta v = -\eta \nabla C$$

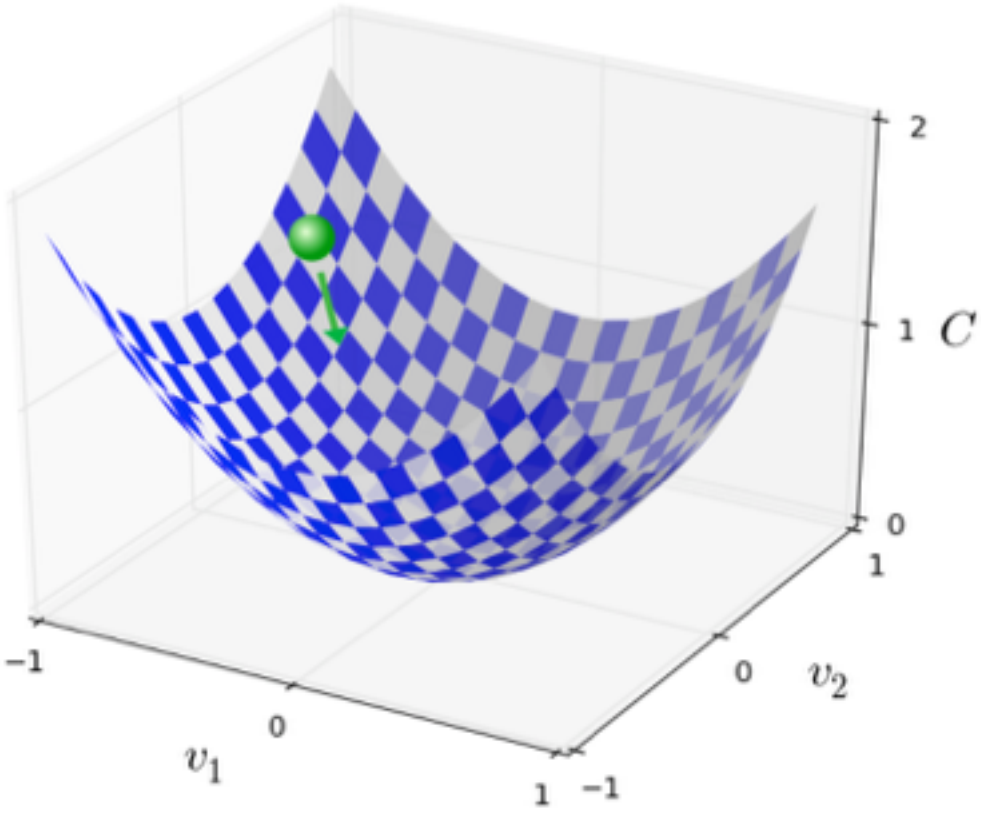
η is learning rate, a small positive number

By doing this, we can assure that ΔC is negative! Why?

*ΔC now is -η * ∇C * ∇C, and it will be negative!*

Learning with gradient descent

$$v \to v' = v - \eta \nabla C$$



Learning with gradient descent

It also works when there are more than two variables.

$\Delta v=(\Delta v_1,\ldots,\Delta v_m)^T$

$\Delta C \approx \nabla C \cdot \Delta v$, *where the gradient* ∇C *is the vector:*

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m} \right)^T$$

$$\Delta v = -\eta \nabla C$$

Learning with gradient descent

How can we apply gradient descent to learn in a neural network?

The idea is to use gradient descent to find the weights w and biases b, which minimize the cost

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Learning with gradient descent

Notice that this cost function has the form $C = \frac{1}{n} \sum_x C_x$, that is, it's an average over costs $C_x \equiv \frac{\|y(x)-a\|^2}{2}$ for individual training examples. In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then average them, $\nabla C = \frac{1}{n} \sum_x \nabla C_x$. Unfortunately, when the number of training inputs is very large this can take a long time, and learning thus occurs slowly.

Learning with gradient descent

Stochastic gradient descent

- The idea is to estimate the gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs
- Mini-batch: randomly picking m training inputs: X1,X2,...,Xm

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

Learning with gradient descent

Stochastic gradient descent

- Suppose w_k and b_l denote the weights and biases in our neural network. For a random m mini-batch:

$$w_k \to w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \to b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

- We finish this m training data, and pick up another m training data, until we've exhausted the training inputs, which is said to complete an **epoch** of training!
- For MNIST database, if we have target training set of size n = 60,000, and if mini-batch size m = 10, we will get a factor of 6,000 speedup in estimating the gradient!

Demo of short python program

Let's simply learn a short python program with less than 100 lines.

- You can download the code on simon server by:
  git clone https://github.com/mnielsen/neural-networks-and-deep-learning.git

The MINIST dataset:

- In this example, we use 50,000 image data set for training, and 10,000 for validation.
- Python library Numpy is used for fast linear algebra.

Demo of short python program

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

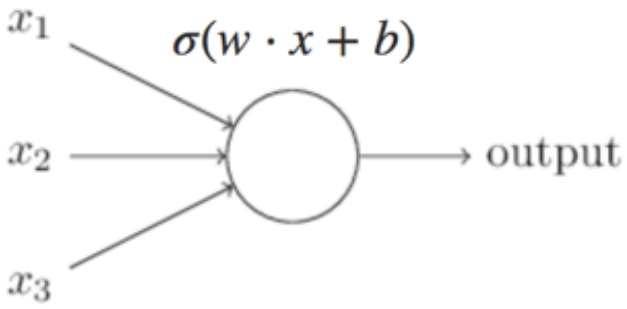
So

- net = Network([2,3,1]) will create a network with 2 neurons in the first layer, 3 neurons in the second layer and 1 neuron in the last layer.
- net.weight[1] is Numpy matrix storing weight connecting second and third layer.

Demo of short python program

Calculating the output:

- First we need to define the sigmoid function
- Calculating the output is straight forward, we only need to calculate formula:



Sigmoid

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}$$

Demo of short python program

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

```
    def feedforward(self, a):
        """Return the output of the network if "a" is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
```

How is the learning process?

- The stochastic gradient decent (SGD)

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):
    """Train the neural network using mini-batch stochastic
    gradient descent.  The "training_data" is a list of tuples
    "(x, y)" representing the training inputs and the desired
    outputs.  The other non-optional parameters are
    self-explanatory.  If "test_data" is provided then the
    network will be evaluated against the test data after each
    epoch, and partial progress printed out.  This is useful for
    tracking progress, but slows things down substantially."""
    if test_data: n_test = len(test_data)
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test)
        else:
            print "Epoch {0} complete".format(j)
```

- Training_data is tuples, input and output
- epochs and mini_batch_size is used for SGD
- eta is learning rate
- If test_data is supplied, we are going to evaluate the performance after each epoch
- This function applies a single step of gradient decent for each mini_batch.

Gradient decent on mini_batch

$$w_k \to w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \to b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

```
def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch.
    The "mini_batch" is a list of tuples "(x, y)", and "eta"
    is the learning rate."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

- most job is done by backprop

Demo of short python program

We currently skip back_prop function, and will go back to it after we learn the back propagation algorithm!

Except for that, the whole program is now understandable.
Let's try it!

```
>>>import mnist_loader
>>>training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
```

```
>>> import network
>>> net = network.Network([784, 30, 10])
```

```
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Demo of short python program

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

the trained network gives us a classification rate of about 95 percent - 95.42 percent at its peak ("Epoch 28")!

Demo of short python program

Can you try to change the number of hidden neurons to 100 and rerun the program?

```
>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Can you try to change the learning rate to 0.001 and rerun the program?

```
>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 0.001, test_data=test_data)
```

Questions to think:

- What's your performance now?
- Do you think we should increase the learning rate?
- What is the best learning rate?
- What if you set learning rate to 100?

In-class exercise

- Read and understand the code in src folder: mnist_loader.py
- Try different learning rate, number of nodes and hidden layers
- Think about simply consider **how dark** to do the job, like 2 is darker


- If you are interested, try the code at: https://github.com/mnielsen/neural-networks-and-deep-learning/blob/master/src/mnist_average_darkness.py

Homework

1. Create account on Gitlab, and each group should have one project named (CS330_fall2018_groupX) for your final project and share to me.

- Here is the link: https://gitlab.com/
- Add me in your project: caora@plu.edu

2. Login Google and Intel platform or use simon server for your experiment.

Homework

3. Install Torch and try to use NN on it.
- Here is the link: http://torch.ch

4. Download Leo Tolstoy's War and Peace at the following link, and use Torch to train a RNN model, then do sampling to generate new text.
- Here is the link: http://cs.stanford.edu/people/karpathy/char-rnn/
- Train and sampling details: https://github.com/karpathy/char-rnn
- Check Demo on: http://simon.cs.plu.edu/MLFun//index.php

5. The homework is due on Nov. 13 (Tuesday). Submit your output or code used.