Machine Learning
Part IV

A Biswas

Syllabus

Introduction

Learning Problems, Well-posed learning problems, Designing learning systems.

Concept Learning

Concept learning task, Inductive hypothesis, Ordering of Hypothesis, General-to-specific ordering of hypotheses. Version spaces, Inductive Bias.

Learning Rule Sets

Sequential Covering Algorithm, First Order Rules, Induction, First Order Resolution, Inverting Resolution.

Regression

Linear regression, Notion of cost function, Logistic Regression, Cost function for logistic regression, application of logistic regression to multi-class classification.

Continued …

Syllabus (continued)

Supervised Learning

Support Vector Machine, Decision tree Learning, Representation, Problems, Decision Tree Learning Algorithm, Attributes, Inductive Bias, Overfitting.

Bayes Theorem, Bayesian learning, Maximum likelihood, Least squared error hypothesis, Gradient Search, Naive Bayes classifier, Bayesian Network, Expectation Maximization Algorithm.

Unsupervised learning

Clustering, K-means clustering, hierarchical clustering.

Instance-Based Learning

k-Nearest Neighbour Algorithm, Radial Basis Function**, Locally Weighted Regression, Locally Weighted Function.**

Neural networks

Linear threshold units, Perceptrons, Multilayer networks and back-propagation, recurrent networks. Probabilistic Machine Learning, Maximum Likelihood Estimation.

Regularization, Preventing Overfitting, Ensemble Learning: Bagging and Boosting, Dimensionality reduction

Unsupervised Learning

Unsupervised learning: dataset doesn’t have labels.

Very problematic for many practical applications.

The absence of labels which represent the desired behavior for your model means the absence of a solid reference point to judge the quality of your model.

Unsupervised Learning

We will see unsupervised learning methods that allow building models that can be evaluated based on data as opposed to human judgment.

Unsupervised Learning

Clustering:

Clustering is a problem of learning to assign a label to examples by leveraging an unlabeled dataset.

Because the dataset is completely unlabeled, deciding on whether the learned model is optimal is much more complicated than in supervised learning.

Unsupervised Learning

Clustering:

There is a variety of clustering algorithms, and, unfortunately, it’s hard to tell which one is better in quality for your dataset.

Usually, the performance of each algorithm depends on the unknown properties of the probability distribution the dataset was drawn from.

Unsupervised Learning

K-Means Clustering:

Steps:

1. The analyst has to choose k - the number of classes (or clusters).
2. Then we randomly put k feature vectors, called centroids, to the feature space.
3. Compute the distance from each example x to each centroid c using some metric, like the Euclidean distance.

Some variants of k-means compute the initial positions of centroids based on some properties of the dataset.

Unsupervised Learning

K-Means Clustering:

Steps:

4. Then we assign the closest centroid to each example (say each example with a centroid id as the label).

5. For each centroid, we calculate the average feature vector of the examples labeled with it. These average feature vectors become the new locations of the centroids.

Unsupervised Learning

K-Means Clustering:

Steps:

6. We recompute the distance from each example to each centroid, modify the assignment and repeat the procedure until the assignments don’t change after the centroid locations were recomputed. The model is the list of assignments of centroids IDs to the examples.

Unsupervised Learning

K-Means Clustering:

Steps:

7. The initial position of centroids influence the final positions, so two runs of k-means can result in two dierent models.

8. The value of k, the number of clusters, is a hyperparameter that has to be tuned by the data analyst.

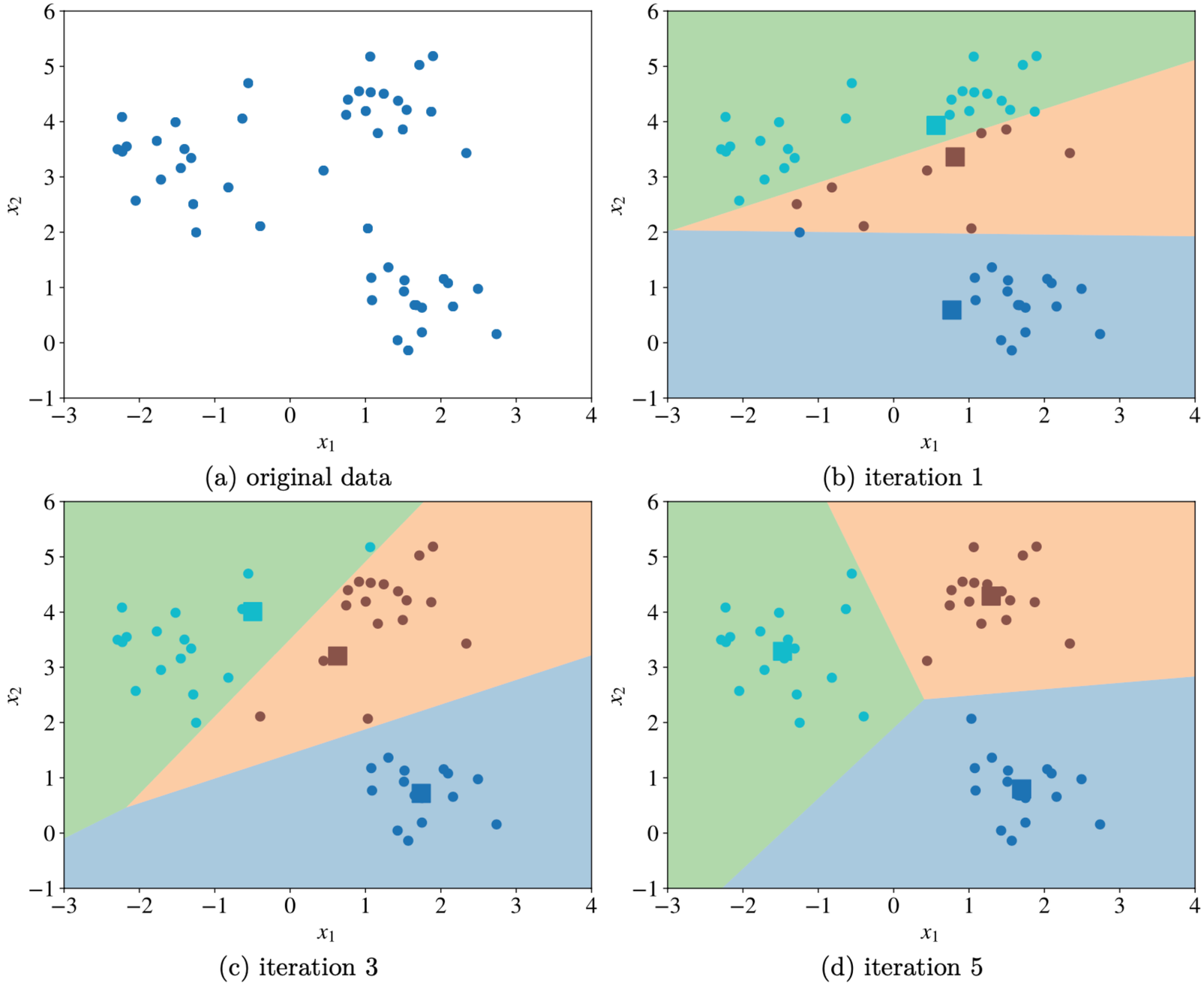
Unsupervised Learning

K-Means Clustering:

Steps:

9. There are some techniques for selecting k. None of them is proven optimal.

10. Most of them require from the analyst to make an “educated guess” by looking at some metrics or by examining cluster assignments visually.



(a) original data (b) iteration 1

(c) iteration 3 (d) iteration 5

Figure 2: The progress of the kmeans algorithm for $k = 3$. The circles are two-dimensional feature vectors; the squares are moving centroids.

The k-means algorithm works by initializing k different centroids $\{\boldsymbol{\mu}^{(1)}, \ldots, \boldsymbol{\mu}^{(k)}\}$ to different values, then alternating between two different steps until convergence. In one step, each training example is assigned to cluster i, where i is the index of the nearest centroid $\boldsymbol{\mu}^{(i)}$. In the other step, each centroid $\boldsymbol{\mu}^{(i)}$ is updated to the mean of all training examples $\boldsymbol{x}^{(j)}$ assigned to cluster i.

Unsupervised Learning

Determining the Number of Clusters:

The most important question is how many clusters does your dataset have?

When the feature vectors are one-, two- or three-dimensional, you can look at the data and see “clouds” of points in the feature space. Each cloud is a potential cluster.

However, for D-dimensional data, with D > 3, looking at the data is problematic.

Unsupervised Learning

Determining the Number of Clusters:

One approach based on prediction strength:

1. Split the data into training (S_{tr} of size N_{tr}) and test set (S_{te} of size N_{te}).
2. Fix k, the number of clusters.
3. Run a clustering algorithm C on S_{tr} and S_{te} to get the clustering results $C(S_{tr}, k)$ and $C(S_{te}, k)$.

Unsupervised Learning

Determining the Number of Clusters:

Let A be the clustering $C(S_{tr},k)$ built using the training set.

Note that the clusters in A can be defined by some regions.

If an example falls within one of those regions, then this example belongs to some specific cluster.

Unsupervised Learning

Determining the Number of Clusters:

4. Determine the $N_{te} \times N_{te}$ co-membership function $\mathbf{D}[A, S_{te}]$ as:

$D[A, S_{te}]^{i,i'} = 1$ if and only if examples x_i *and* $x_{i'}$ from the test set belongs to the same cluster according to the clustering A.

Otherwise, $D[A, S_{te}]^{i,i'} = 0$

Unsupervised Learning

Determining the Number of Clusters:

So far,

we have built, using the training set of examples, the clustering A that has k clusters.

Then we have built the co-membership matrix that indicates whether two examples from the test set belong to the same cluster in A.

Unsupervised Learning

Determining the Number of Clusters:

Intuitively, if the quantity k is the reasonable number of clusters, then two examples that belong to the same cluster in clustering $C(S_{te},k)$ will most likely belong to the same cluster in clustering $C(S_{tr},k)$.

On the other hand, if k is not reasonable (too high or too low), then training data-based and test data-based clusterings will likely be less consistent.

Unsupervised Learning

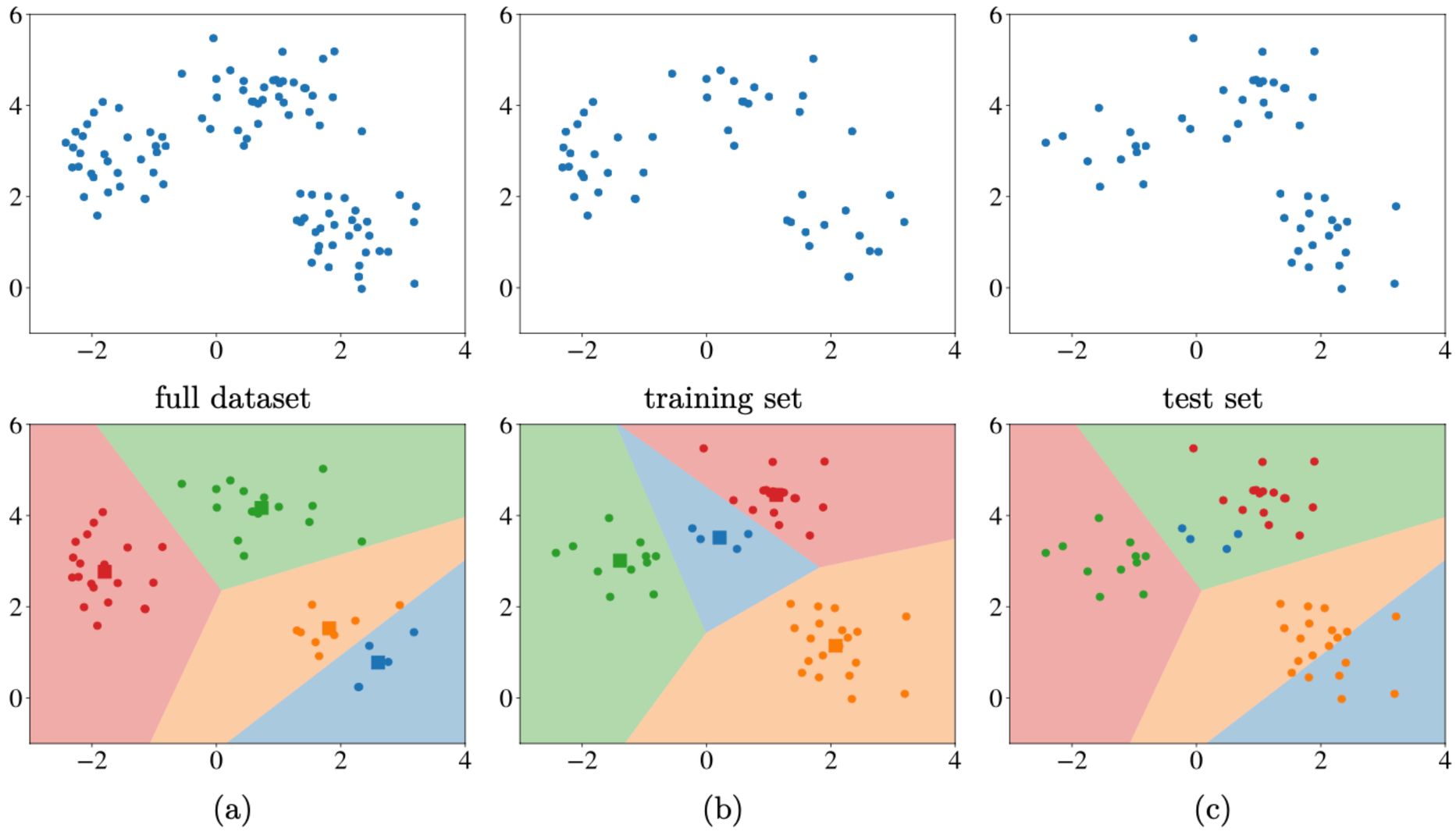


Figure 3: The clustering for $k = 4$: (a) training data clustering; (b) test data clustering; (c) test data plotted over the training clustering.

You can see in 3c that orange test examples don't belong anymore to the same cluster according to the clustering regions obtained from the training data. This will result in many zeroes in the matrix **D**[A,Ste] which, in turn, is an indicator that k = 4 is likely not the best number of clusters.

Unsupervised Learning

Determining the Number of Clusters:

Formally, the prediction strength for the number of clusters k is given by,

$$\mathrm{ps}(k) \stackrel{\text{def}}{=} \min_{j=1,\ldots,k} \frac{1}{|A_j|(|A_j|-1)} \sum_{i,i' \in A_j} \mathbf{D}[A, \mathcal{S}_{te}]^{(i,i')},$$

where $A \stackrel{\text{def}}{=} C(\mathcal{S}_{tr}, k)$, A_j is j^{th} cluster from the clustering $C(\mathcal{S}_{te}, k)$ and $|A_j|$ is the number of examples in cluster A_j.

Unsupervised Learning

Determining the Number of Clusters:

5. Given a clustering $C(S_{tr},k)$, for each test cluster, we compute the proportion of observation pairs in that cluster that are also assigned to the same cluster by the training set centroids.

The prediction strength is the minimum of this quantity over the k test clusters.

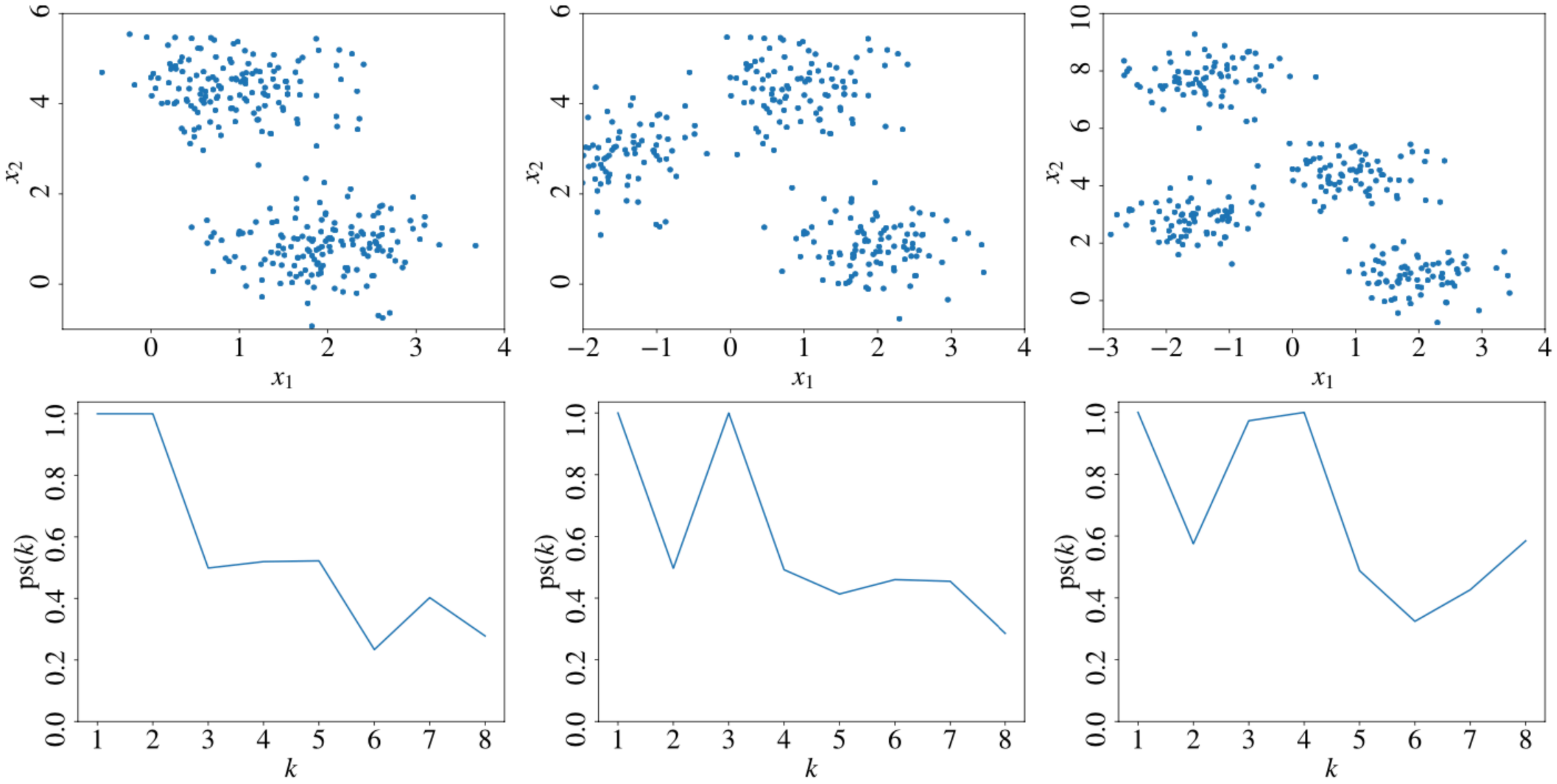


Figure 4: Predictive strength for different values of k for two-, three- and four-cluster data.

Experiments suggest that a reasonable number of clusters is the largest k such that ps(k) is above 0.8.

Unsupervised Learning

Dimensionalty Reduction:

Many modern machine learning algorithms, such as ensemble algorithms and neural networks handle well very high-dimensional examples, up to millions of features.

Unsupervised Learning

Dimensionalty Reduction:

With modern computers and graphical processing units (GPUs), dimensionality reduction techniques are used much less in practice than in the past. The most frequent use case for dimensionality reduction is data visualization: humans can only interpret on a plot the maximum of three dimensions.

Unsupervised Learning

Dimensionalty Reduction:

Another situation in which you could benefit from dimensionality reduction is when you have to build an interpretable model and to do so you are limited in your choice of learning algorithms.

Unsupervised Learning

Dimensionalty Reduction:

For example, you can only use decision tree learning or linear regression. By reducing your data to lower dimensionality and by figuring out which quality of the original example each new feature in the reduced feature space reflects, one can use simpler algorithms. Dimensionality reduction removes redundant or highly correlated features; it also reduces the noise in the data — all that contributes to the interpretability of the model.

Unsupervised Learning

Dimensionalty Reduction: Principal Component Analysis

Consider a two-dimensional data

Principal components are vectors that define a new coordinate system in which the first axis goes in the direction of the highest variance in the data.

The second axis is orthogonal to the first one and goes in the direction of the second highest variance in the data.

Unsupervised Learning

Dimensionalty Reduction: Principal Component Analysis

If our data was three-dimensional, the third axis would be orthogonal to both the first and the second axes and go in the direction of the third highest variance, and so on.

Unsupervised Learning

Dimensionalty Reduction: Principal Component Analysis

Now, if we want to reduce the dimensionality of our data to $D_{new} < D$, we pick D_{new} largest principal components and project our data points on them.

Unsupervised Learning

Dimensionalty Reduction: Principal Component Analysis

For our two-dimensional illustration, we can set $D_{new} = 1$ and project our examples to the first principal component to obtain the orange points in fig. 6c.

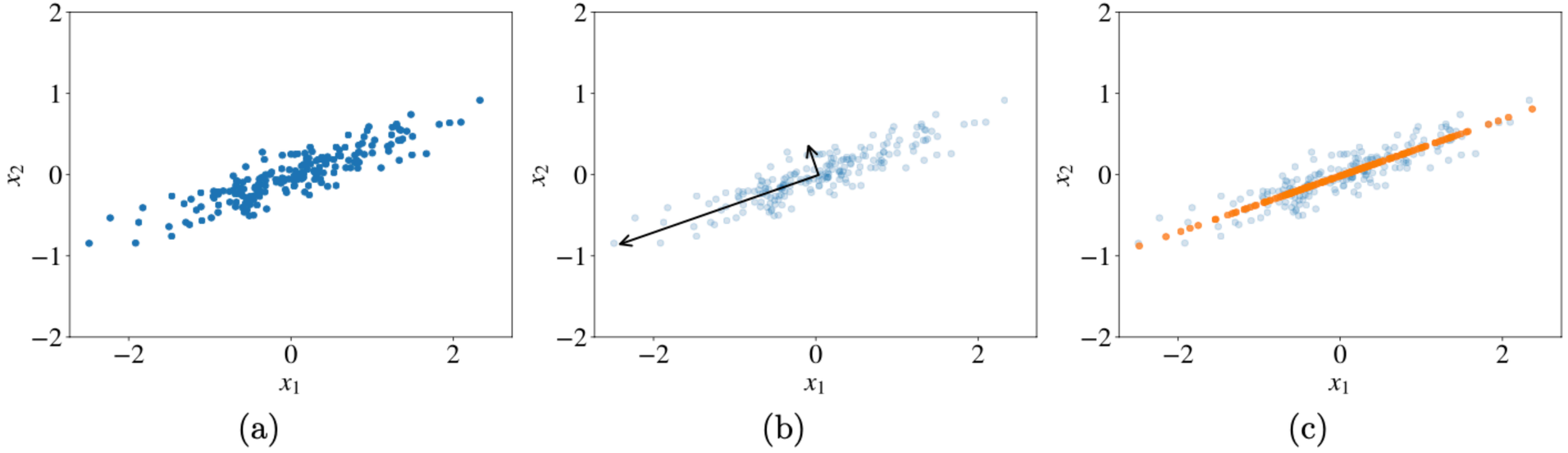


Figure 6: PCA: (a) the original data; (b) two principal components displayed as vectors; (c) the data projected on the first principal component.

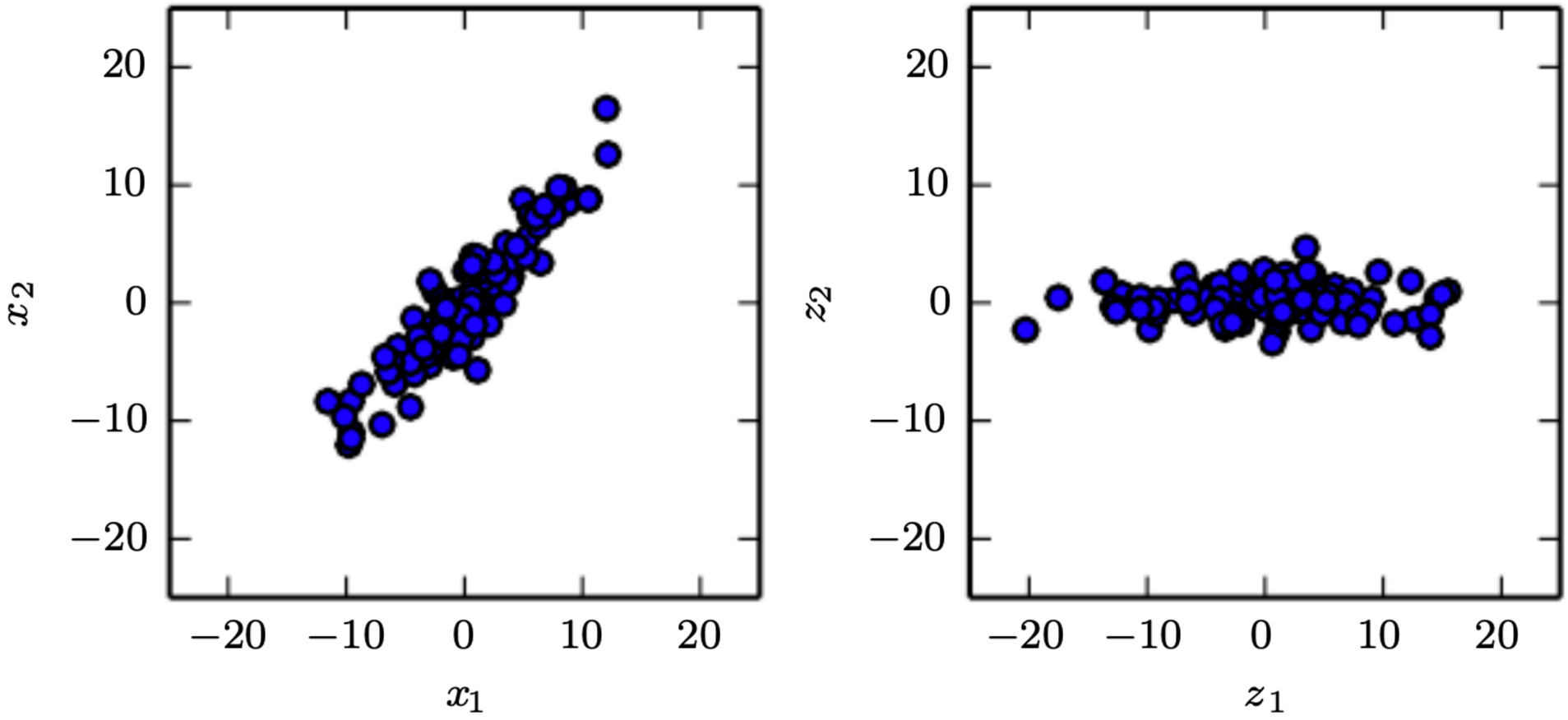


Figure 5.8: PCA learns a linear projection that aligns the direction of greatest variance with the axes of the new space. *(Left)*The original data consists of samples of $\boldsymbol{x}$. In this space, the variance might occur along directions that are not axis-aligned. *(Right)*The transformed data $\boldsymbol{z} = \boldsymbol{x}^\top \boldsymbol{W}$ now varies most along the axis z_1. The direction of second most variance is now along z_2.

Neural Network

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Algorithms such as BACKPROPAGATION use gradient descent to tune network parameters to best fit a training set of input-output pairs.

Neural Network

ANN learning is robust to errors in the training data and has been successfully applied to problems such as interpreting visual scenes, speech recognition, and learning robot control strategies.

Neural Network

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions.

Neural Network

For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known.

Neural Network

The BACKPROPAGATION algorithm has proven surprisingly successful in many practical problems such as
learning to recognize handwritten characters (LeCun et al. 1989),
learning to recognize spoken words (Lang et al. 1990), and
learning to recognize faces (Cottrell 1990)

Neural Network

One survey of practical applications is provided by Rumelhart et al. (1994).

Neural Network

Biological Motivation:

Artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

Neural Networks

Biological Motivation:

Neurobiology:

The human brain, for example, is estimated to contain a densely interconnected network of approximately 10^{11} neurons, each connected, on average, to 10^4 others.

Neural Networks

Biological Motivation: Neurobiology:

Neuron activity is typically excited or inhibited through connections to other neurons.
The fastest neuron switching times are known to be on the order of 10^{-3} seconds--quite slow compared to computer switching speeds of 10^{-10} seconds.

Yet humans are able to make surprisingly complex decisions, surprisingly quickly.

Neural Network

While ANNs are loosely motivated by biological neural systems, there are many complexities to biological neural systems that are not modeled by ANNs, and many features of the ANNs are known to be inconsistent with biological systems.

For example, we consider here ANNs whose individual units output a single constant value, whereas biological neurons output a complex time series of spikes.

Neural Network

Two groups of researchers

- using ANNs to study and model biological learning processes.
- obtaining highly effective machine learning algorithms, independent of whether these algorithms mirror biological processes.

Neural Networks: A prototypical example

ALVINN:

- to steer an autonomous vehicle driving at normal speeds on public highway.
- by Pomerleau's (1993)

Neural Networks: A prototypical example

ALVINN:

Input: 30 x 32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle

Output: the direction in which the vehicle is steered.

Training: to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes.

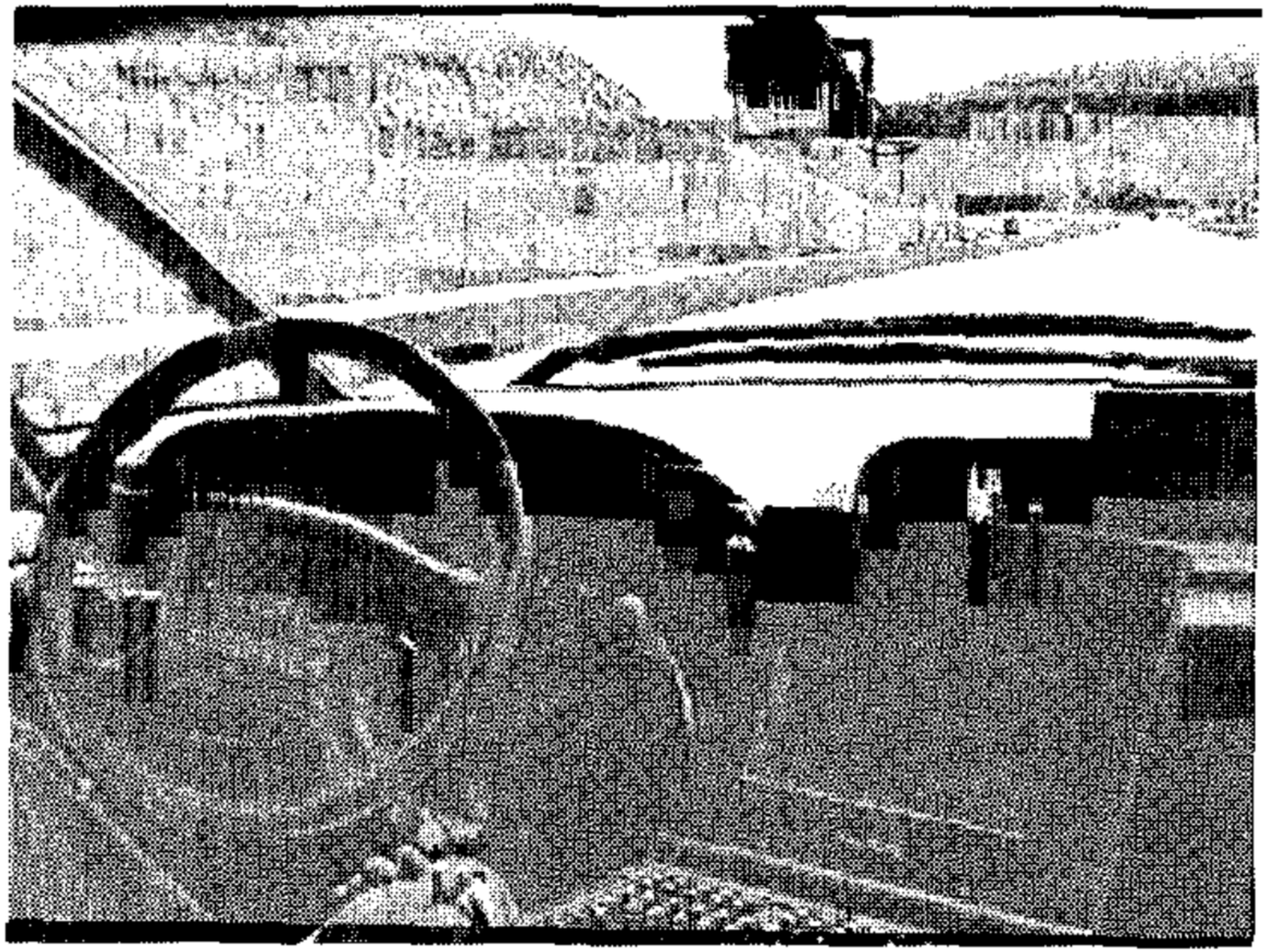
Neural Networks: A prototypical example

ALVINN:

ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways (driving in the left lane of a divided public highway, with other vehicles present).

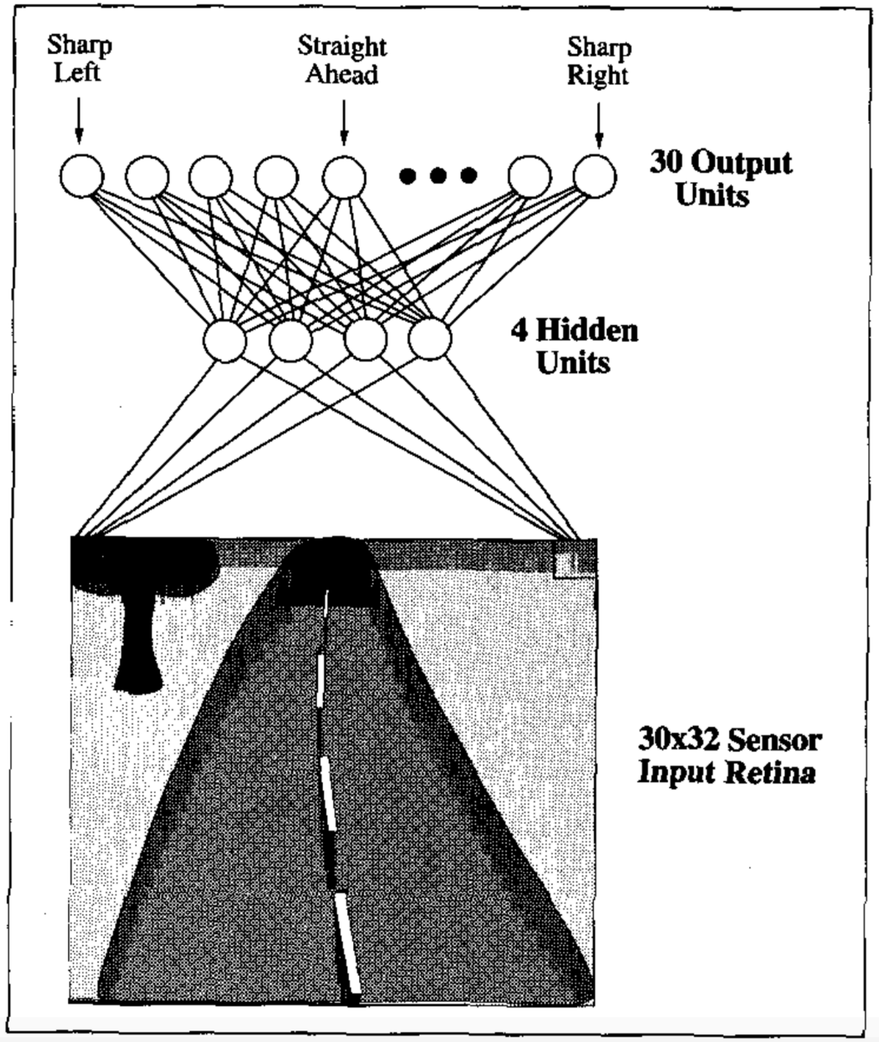
Neural Networks: A prototypical example

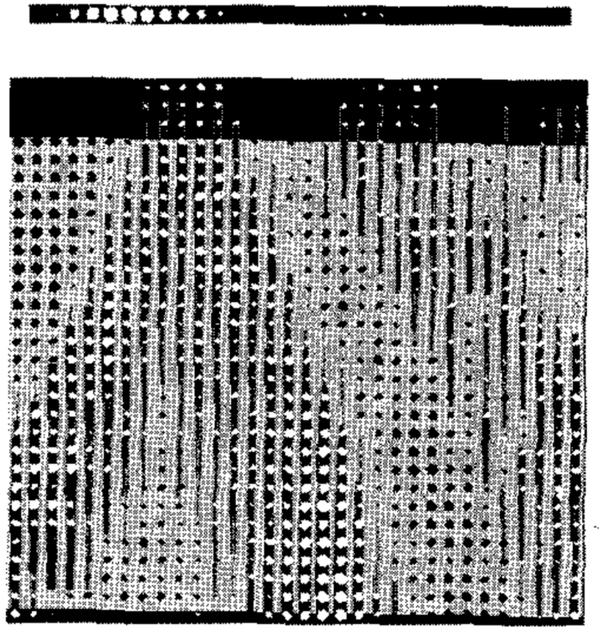
ALVINN:



Neural Networks: A prototypical example

ALVINN:

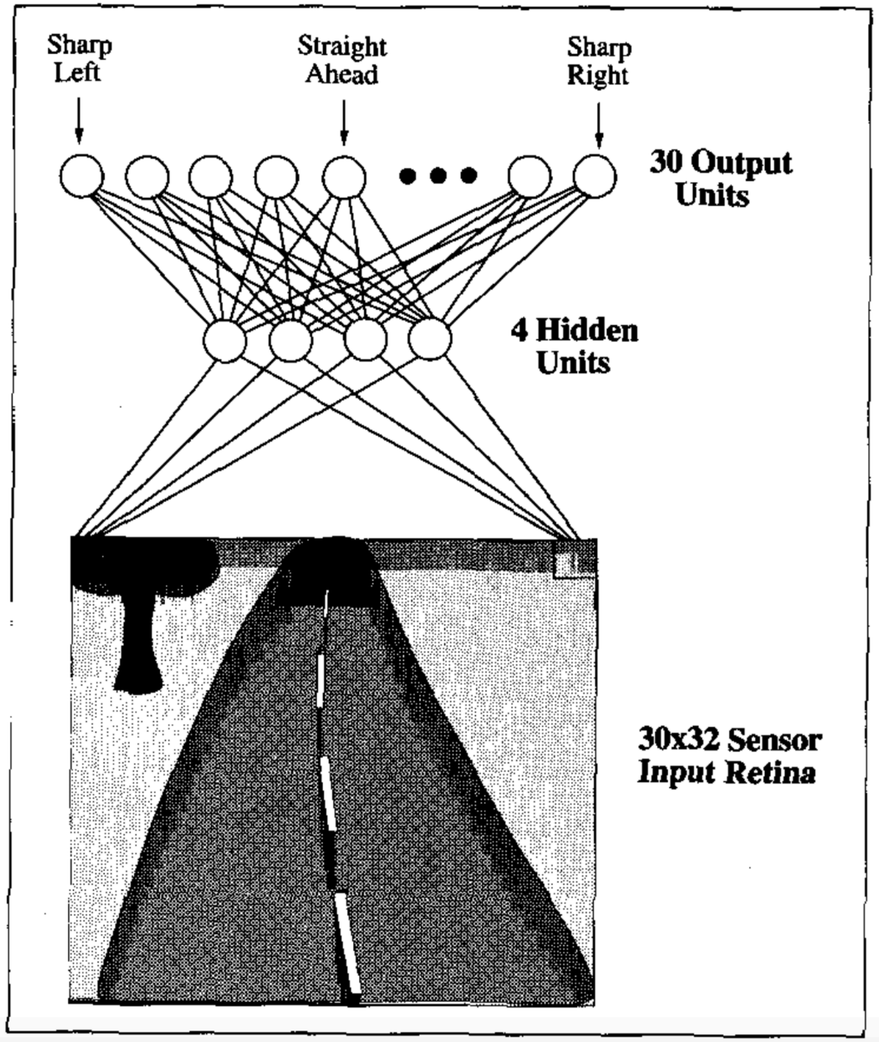


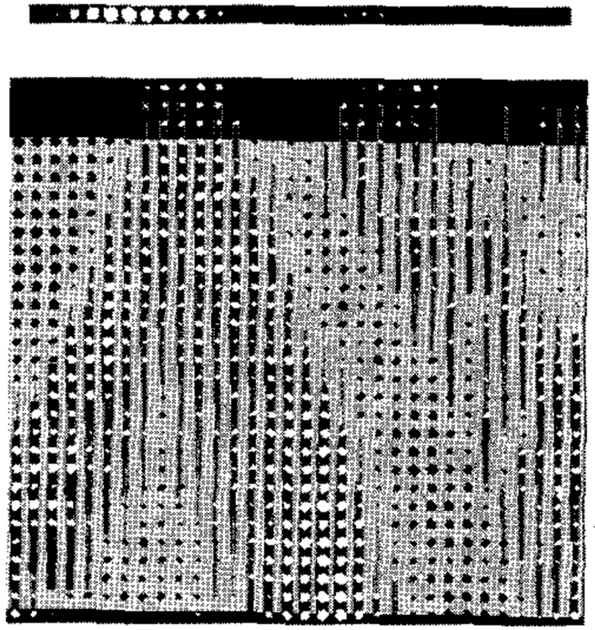


Left: shows how the image of a forward-mounted camera is mapped to **960** neural network inputs, which are fed forward to 4 hidden units, connected to **30** output units.

Neural Networks: A prototypical example

ALVINN:

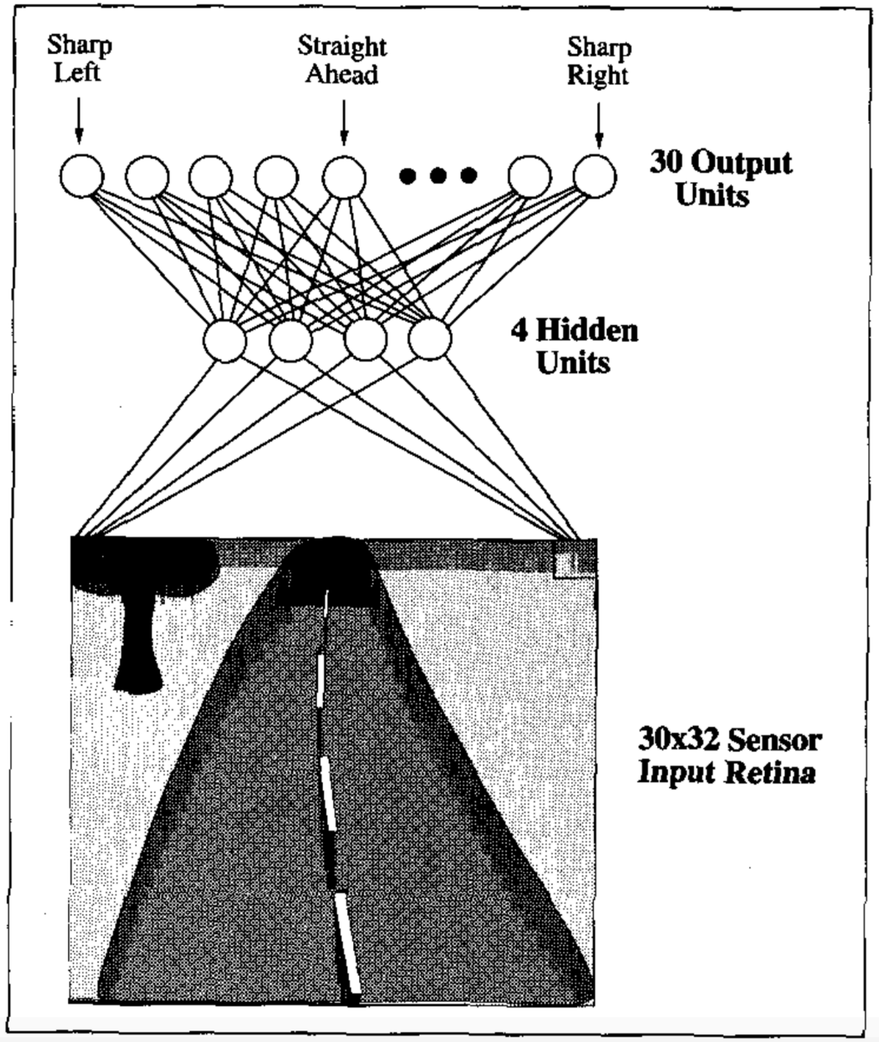


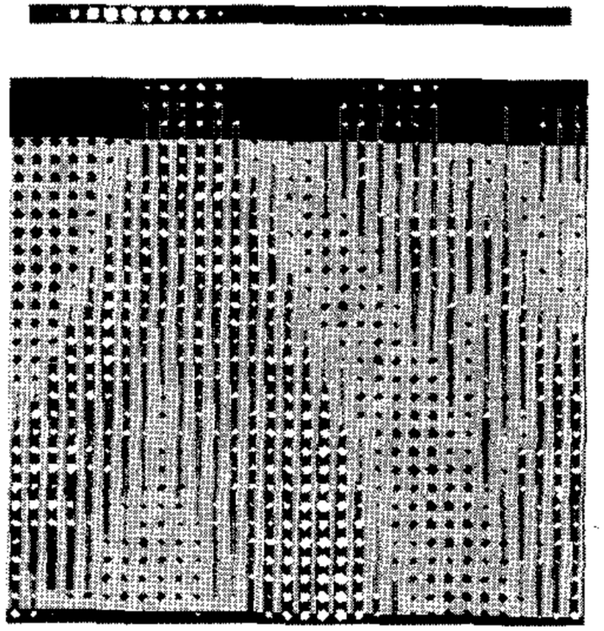


Network outputs encode the commanded steering direction.

Neural Networks: A prototypical example

ALVINN:

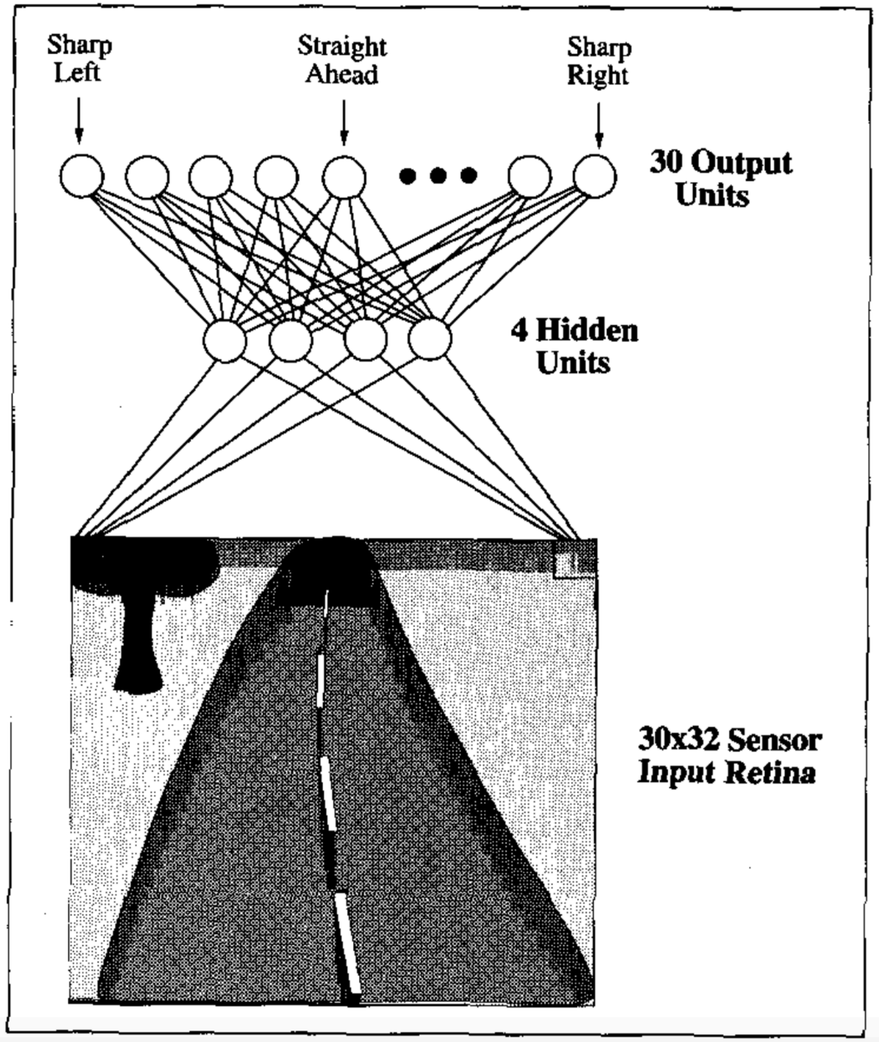


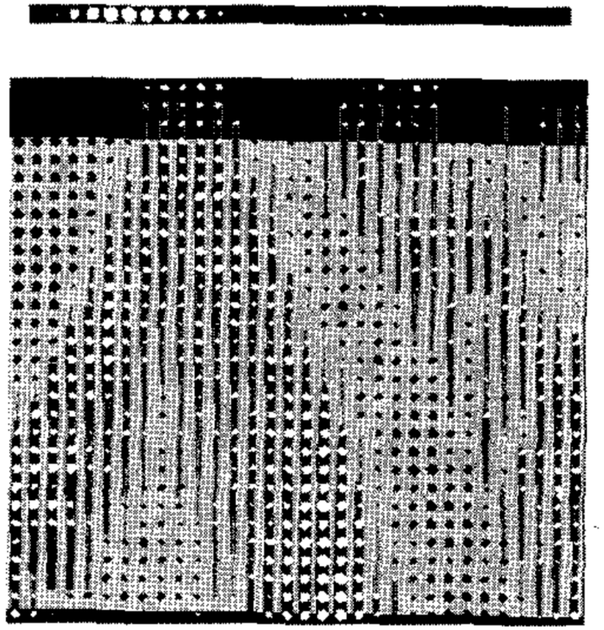


Right: shows weight values for one of the hidden units in this network. The **30 x** 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights

Neural Networks: A prototypical example

ALVINN:





The weights from this hidden unit to the **30** output units **are** depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

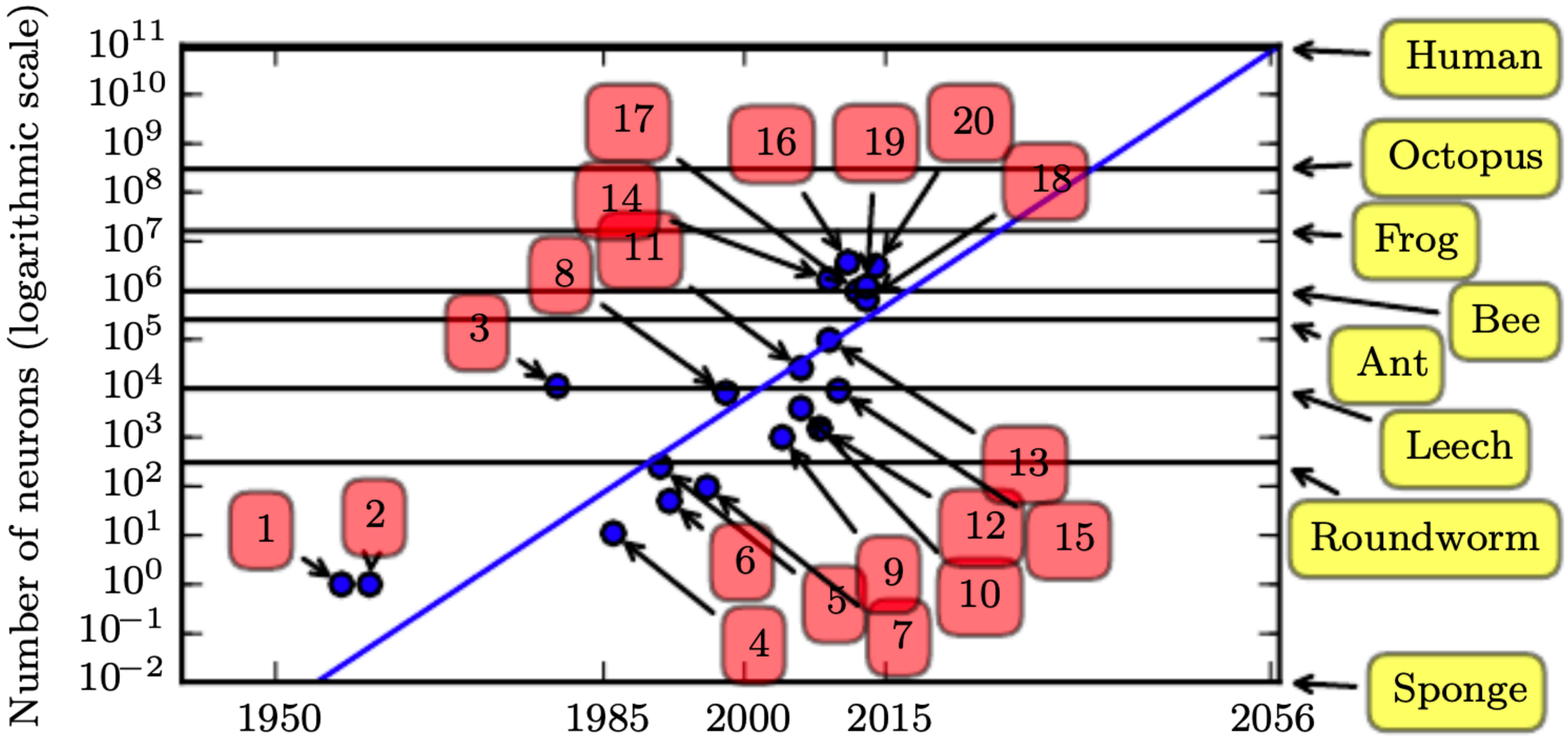


Figure 1.11: Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. Biological neural network sizes from Wikipedia (2015).

1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart *et al.*, 1986b)
5. Recurrent neural network for speech recognition (Robinson and Fallside, 1991)
6. Multilayer perceptron for speech recognition (Bengio *et al.*, 1991)
7. Mean field sigmoid belief network (Saul *et al.*, 1996)
8. LeNet-5 (LeCun *et al.*, 1998b)
9. Echo state network (Jaeger and Haas, 2004)
10. Deep belief network (Hinton *et al.*, 2006)
11. GPU-accelerated convolutional network (Chellapilla *et al.*, 2006)
12. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina *et al.*, 2009)
14. Unsupervised convolutional network (Jarrett *et al.*, 2009)
15. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
16. OMP-1 network (Coates and Ng, 2011)
17. Distributed autoencoder (Le *et al.*, 2012)
18. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012)
19. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
20. GoogLeNet (Szegedy *et al.*, 2014a)

Perceptrons

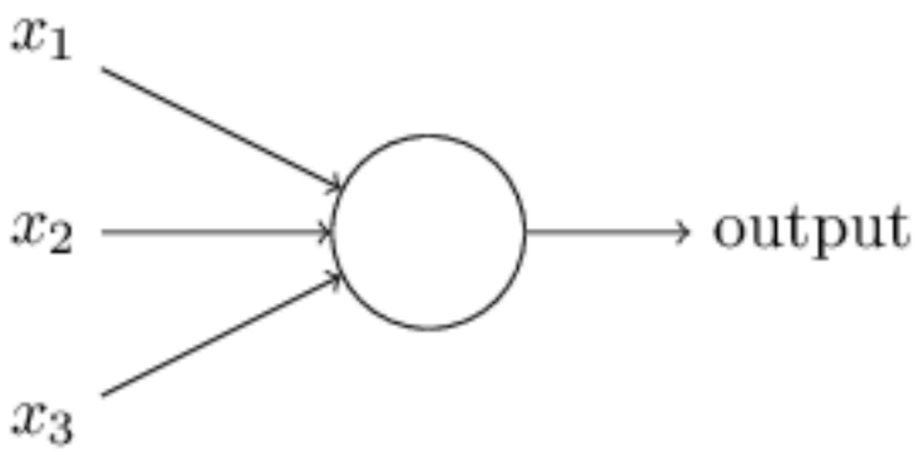
- a type of artificial neuron called a perceptron.
- 1950-1960 - by Frank Rosenblatt.
- modern neurons are Sigmoid neuron.

Perceptrons

How do perceptrons work?

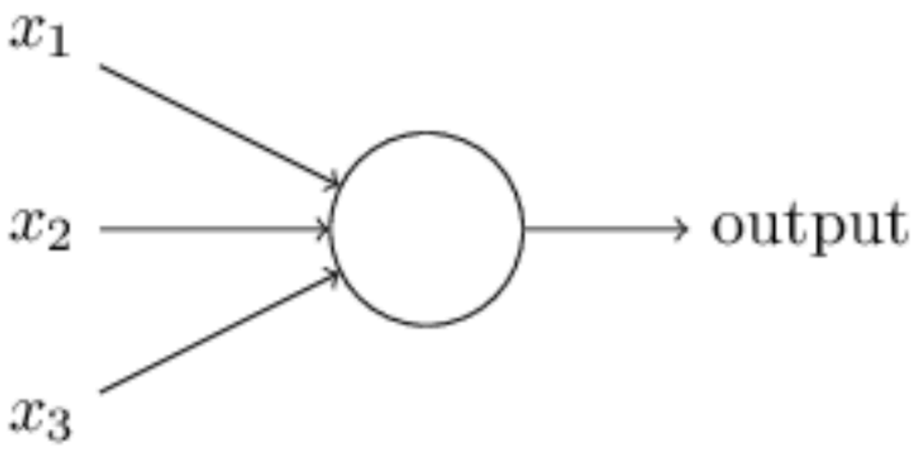
A perceptron takes several binary inputs,

$x_1, x_2, \ldots, x_n$, and produces a single binary output.



Perceptrons

How do perceptrons work?



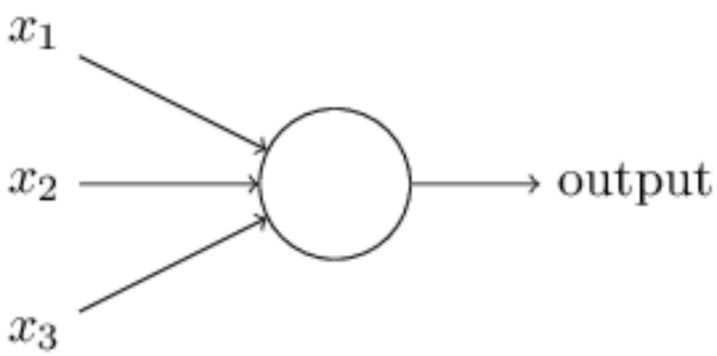
$$\text{output} = \begin{cases} 0 & \text{if} \quad \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if} \quad \sum_j w_j x_j > \text{threshold} \end{cases}$$

Three binary **inputs** x_1, x_2, x_3.

Weights: w_1, w_2, w_3, real numbers expressing the importance of the respective inputs to the output.

Threshold:a real number - a parameter of the neuron

Perceptrons



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

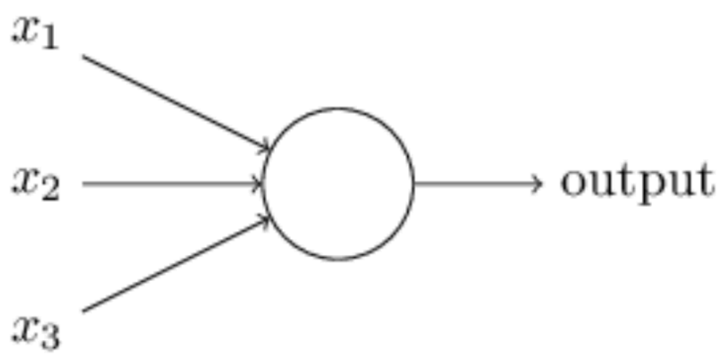
Example: Cheese festival at weekend

To decide to go or not to go to the cheese festival.

Decision based on:

1. Weather is good?
2. Boyfriend or girlfriend wants to accompany you?
3. Festival near the public transit? (Own no car)

Perceptrons



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

You love cheese.

Weights set: $w_1 = 6, w_2 = 2, w_3 = 2$ - weather matters to you.

Threshold chosen: 5.

Drop the threshold - you are more willing to go.

Perceptrons

By varying the weights and the threshold, we can get different models of decision-making. For example, suppose we instead chose a threshold of 3. Then the perceptron would decide that you should go to the festival whenever the weather was good or when both the festival was near public transit and your boyfriend or girlfriend was willing to join you. In other words, it'd be a different model of decision-making. Dropping the threshold means you're more willing to go to the festival.

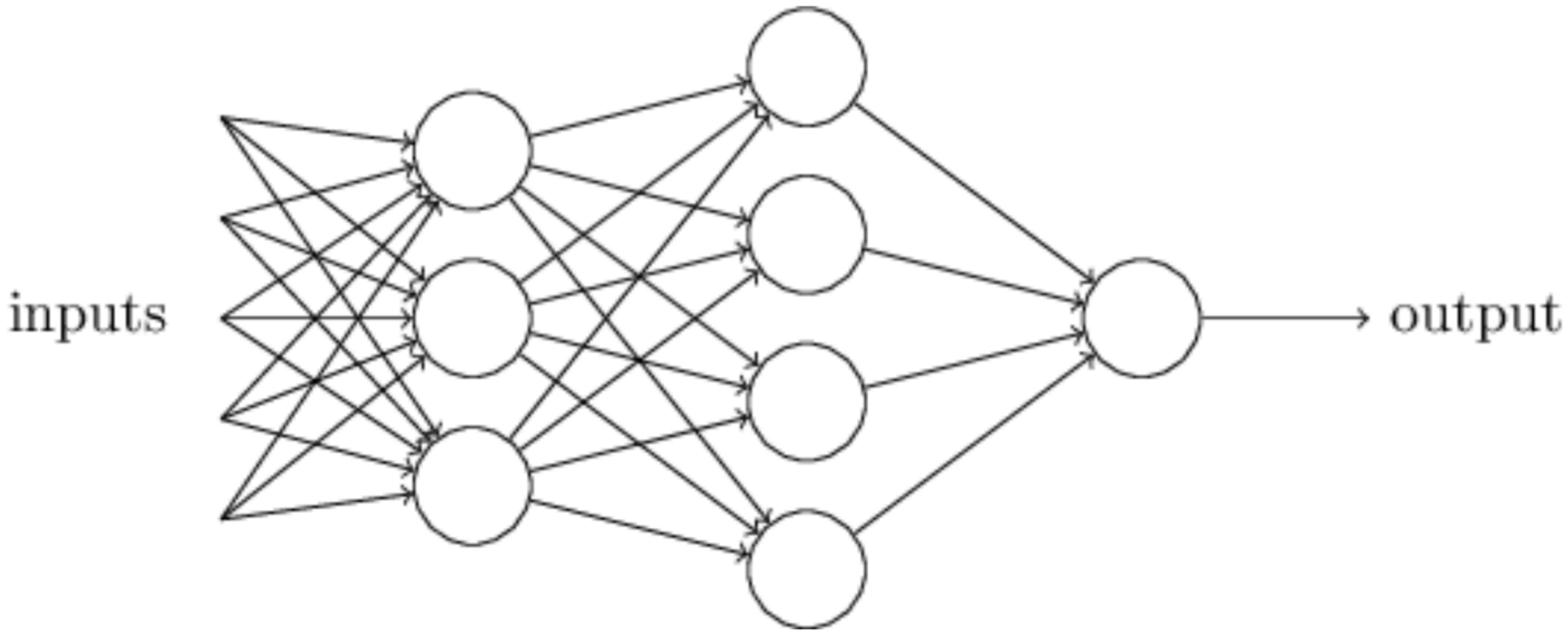
Perceptrons

Obviously, the perceptron isn't a complete model of human decision-making!

But the example illustrates is how a perceptron can weigh up different kinds of evidence in order to make decisions.

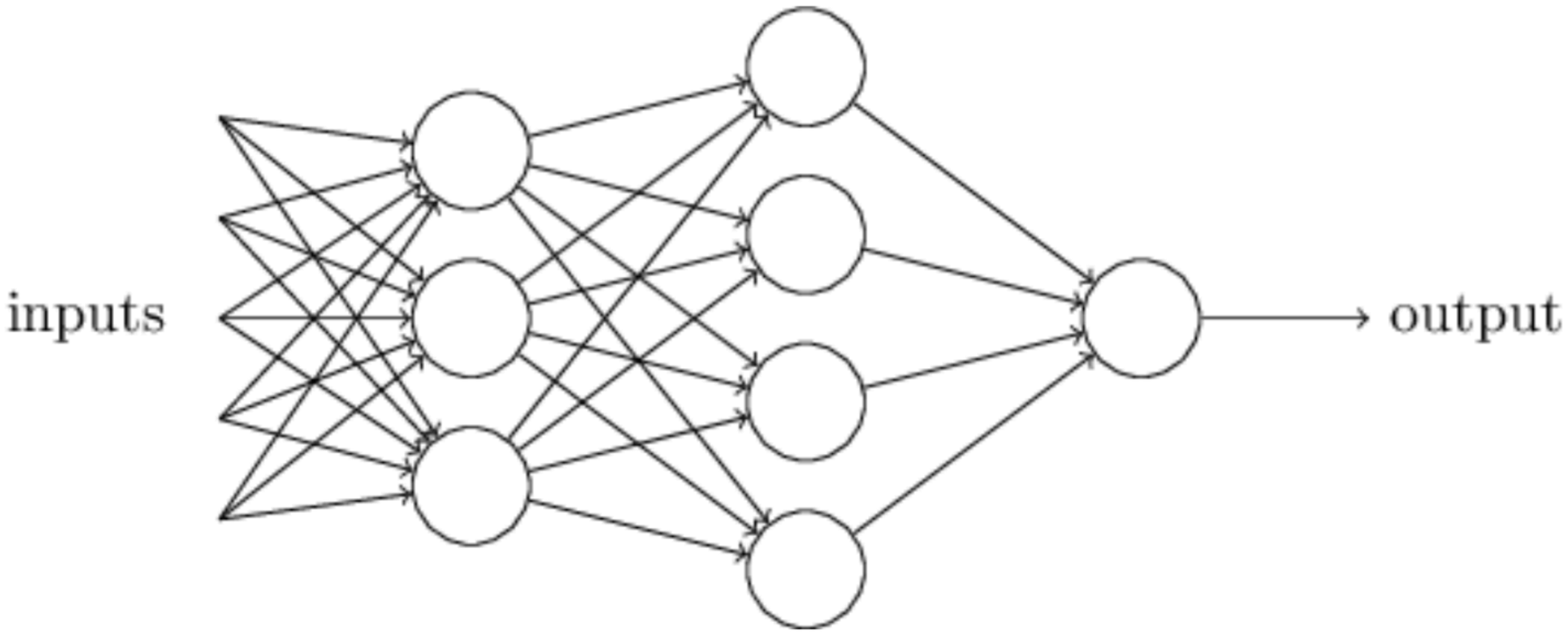
And it should seem plausible that a complex network of perceptrons could make quite subtle decisions.

Perceptrons



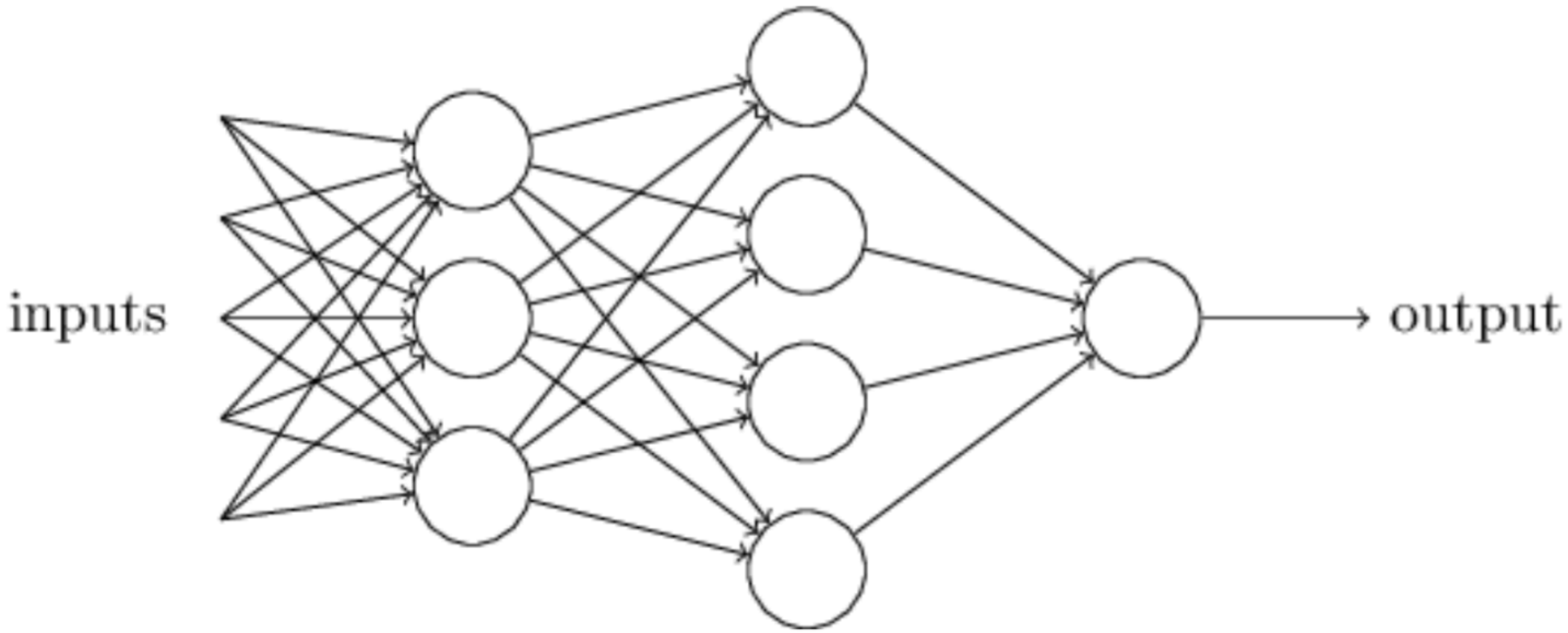
the first column of perceptrons – what we'll call the first layer of perceptrons – is making three very simple decisions, by weighing the input evidence.

Perceptrons



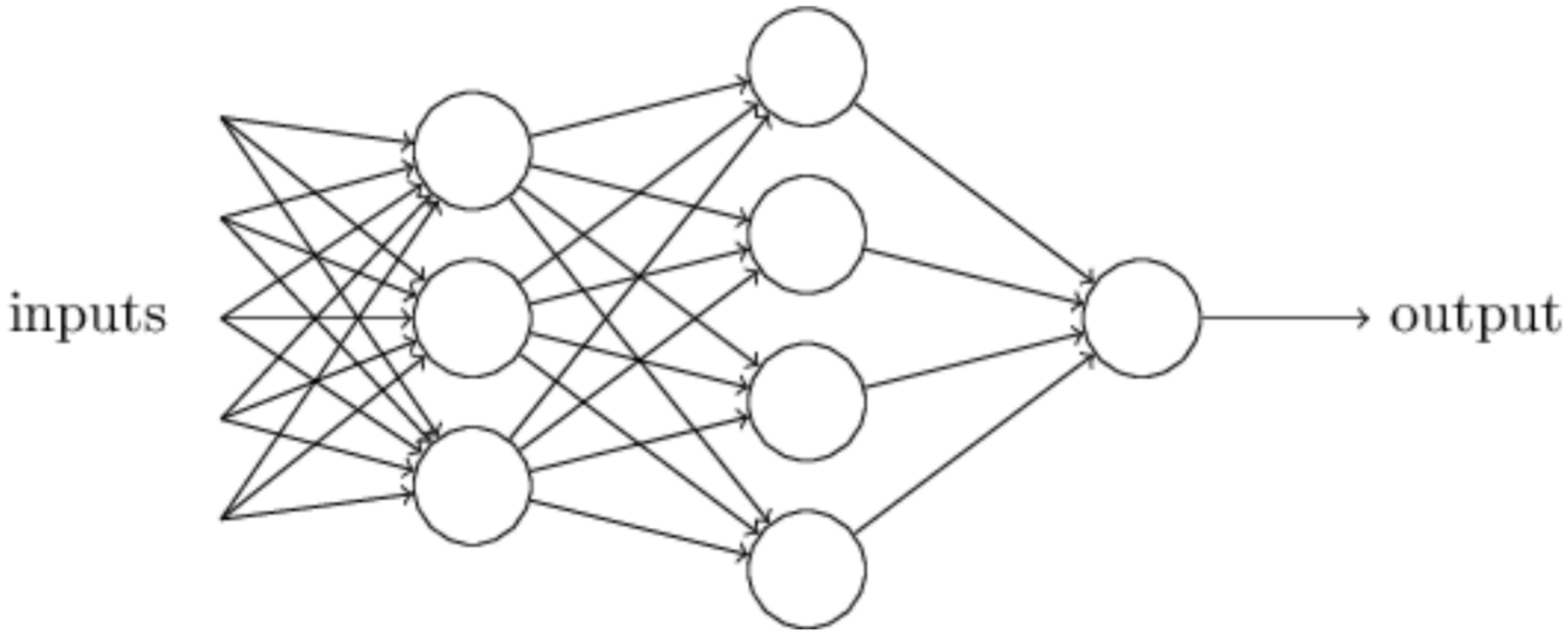
What about the perceptrons in the second layer? Each of those perceptrons is making a decision by weighing up the results from the first layer of decision-making.

Perceptrons



In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer

Perceptrons



And even more complex decisions can be made by the perceptron in the third layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

Perceptrons

Let's simplify

1. Change $\sum_j w_j x_j$ to a dot product;

$$\mathbf{w} \cdot \mathbf{x} = \sum_j w_j x_j$$

2. Move the threshold to the other side, replace by the Perceptron's bias, b= - threshold.

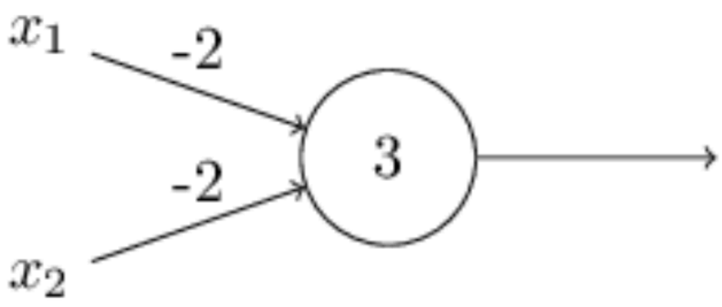
$$\text{output} = \begin{cases} 0 & \text{if} \quad w \cdot x + b \leq 0 \\ 1 & \text{if} \quad w \cdot x + b > 0 \end{cases}$$

Perceptrons

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

The bias is a measure of how easy it is to get the perceptron to fire.
For a perceptron with a really big bias, it's extremely easy for the perceptron to output a 1. But if the bias is very negative, then it's difficult for the perceptron to output a 1.

Perceptrons to compute the elementary logical functions such as AND, OR, and NAND

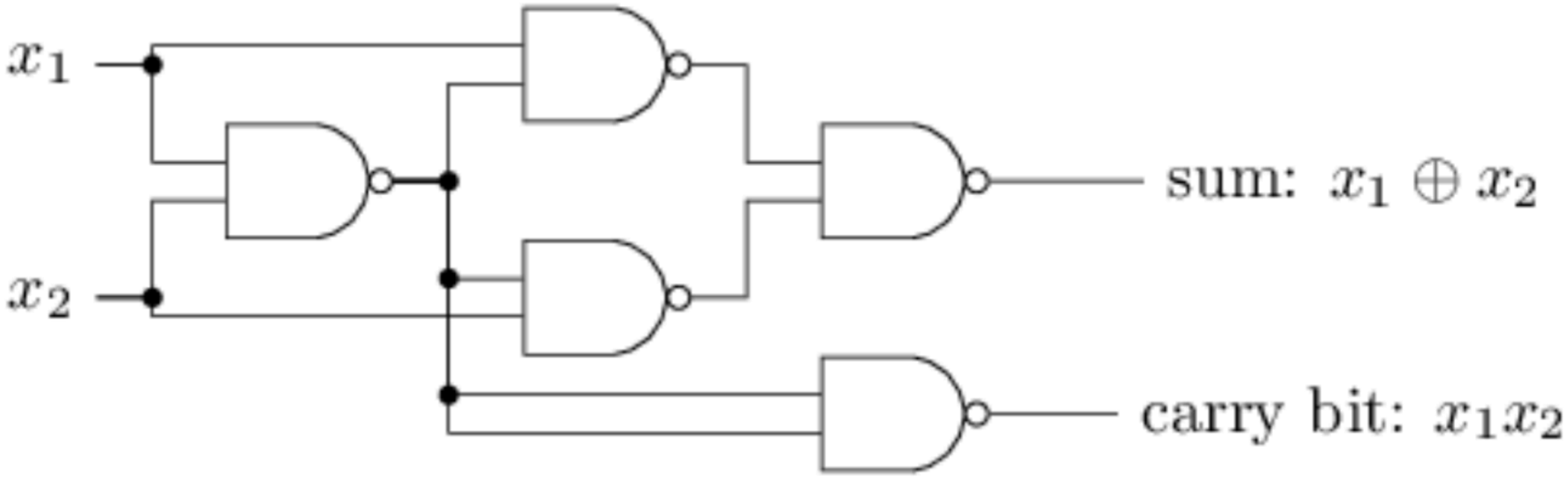


$$\text{output} = \begin{cases} 0 & \text{if} \quad w \cdot x + b \leq 0 \\ 1 & \text{if} \quad w \cdot x + b > 0 \end{cases}$$

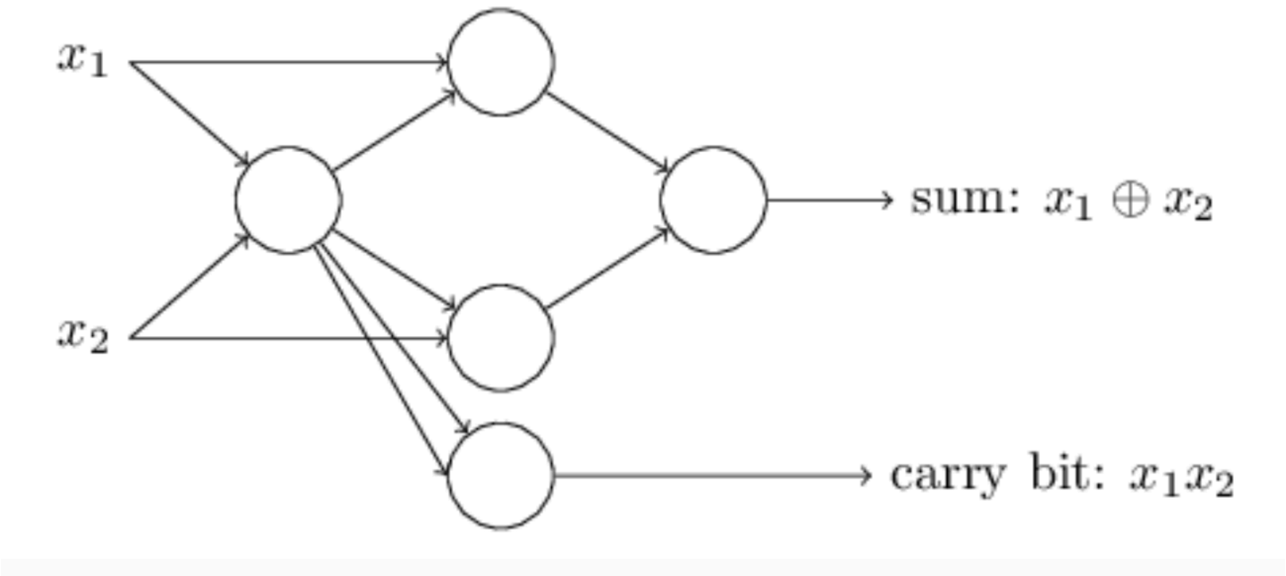
Which gate it is?

Perceptrons to compute the elementary logical functions such as AND, OR, and NAND

NAND gate is universal, we can build any computation up out of NAND gates.

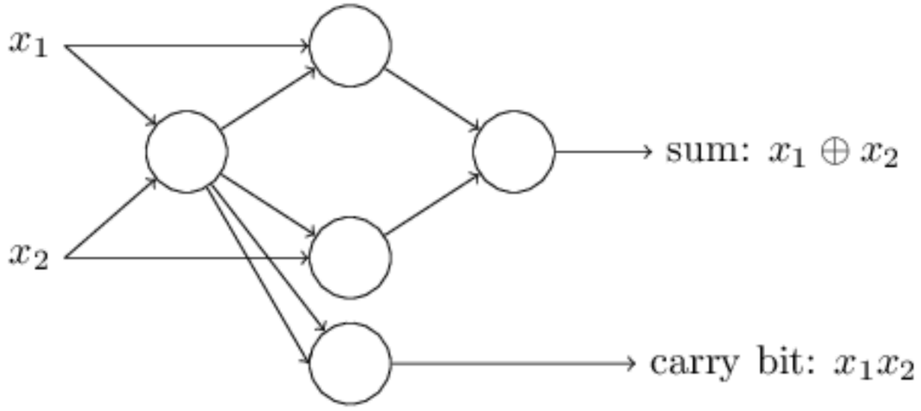


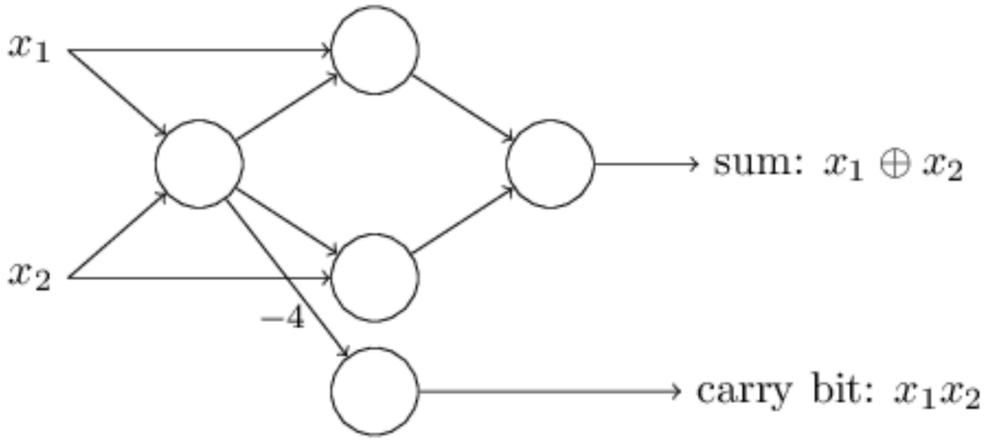
Perceptrons to compute the elementary logical functions such as AND, OR, and NAND

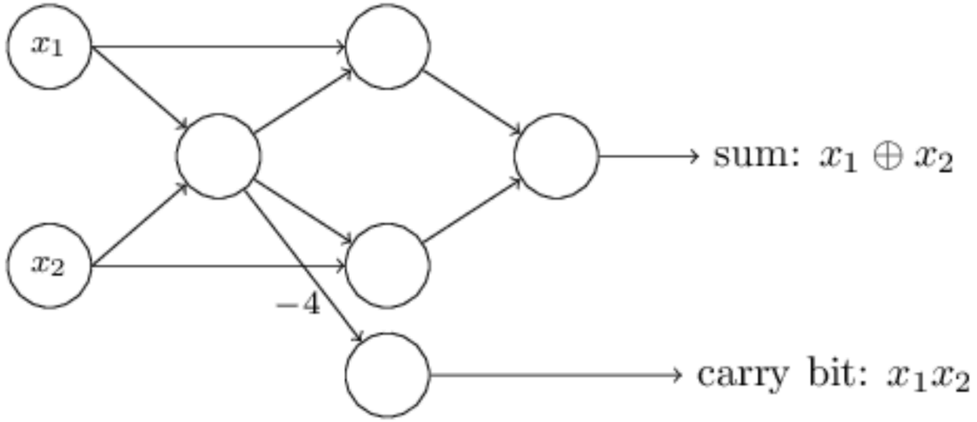


Each input with weights -2 and overall bias 3.

Perceptrons to compute the elementary logical functions such as AND, OR, and NAND







Perceptrons to compute the elementary logical functions such as AND, OR, and NAND

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!

Perceptrons to compute the elementary logical functionssuch as AND, OR, and NAND

We can devise learning algorithms which can automatically tune the weights and biases of a network of artificial neurons.

This tuning happens in response to external stimuli, without direct intervention by a programmer.

Perceptrons to compute the elementary logical functionssuch as AND, OR, and NAND

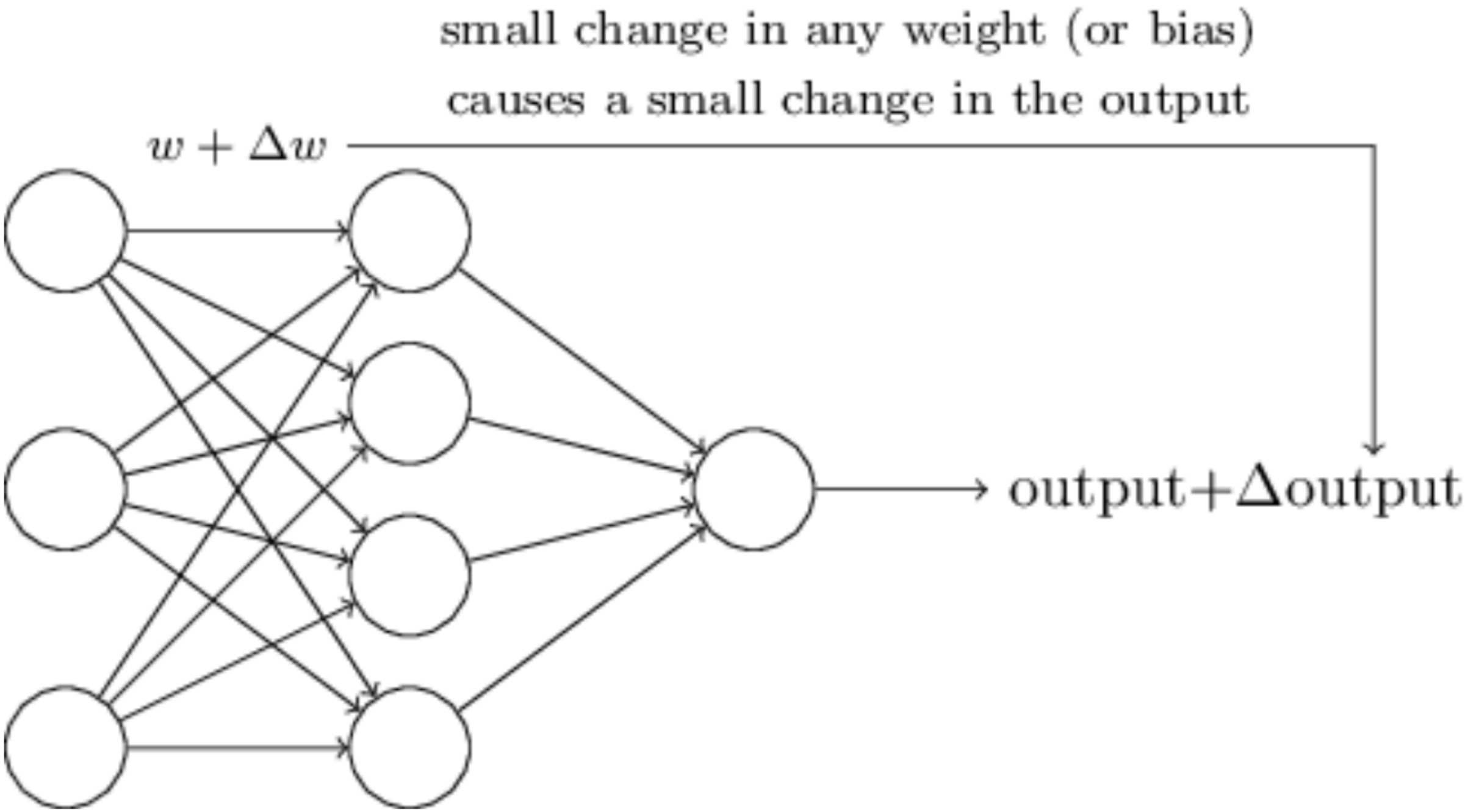
These learning algorithms enable us to use artificial neurons in a way which is radically different to conventional logic gates.

Instead of explicitly laying out a circuit of NAND and other gates, our neural networks can simply learn to solve problems, sometimes problems where it would be extremely difficult to directly design a conventional circuit.

Sigmoid Neurons

Example: the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit.

Goal: the network will learn weights and biases so that the output from the network correctly classifies the digit.


small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

Sigmoid Neurons

If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want.

Sigmoid Neurons

For example, suppose the network was mistakenly classifying an image as an “8” when it should be a “9”.

We could figure out how to make a small change in the weights and biases so the network gets a little closer to classifying the image as a “9”.

Sigmoid Neurons

And then we'd repeat this, changing the weights and biases over and over to produce better and better output. The network would be learning.

Sigmoid Neurons

A network with perceptrons, can it learn?

A small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1.

The flip may cause the behaviour of the rest of the network to completely change in some complicated manner.

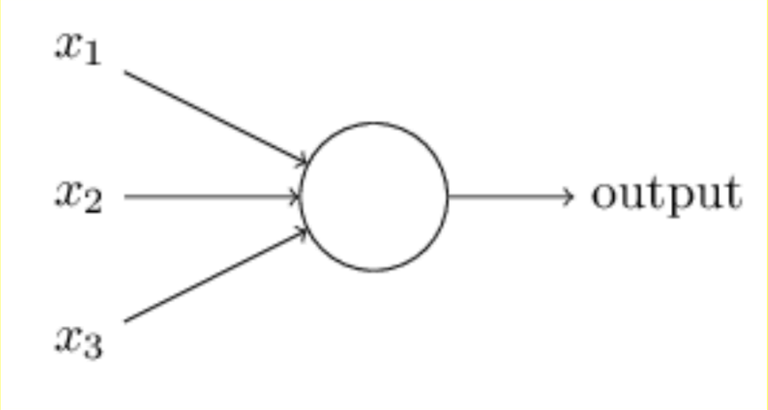
Sigmoid Neurons

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

Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output.

Sigmoid Neurons

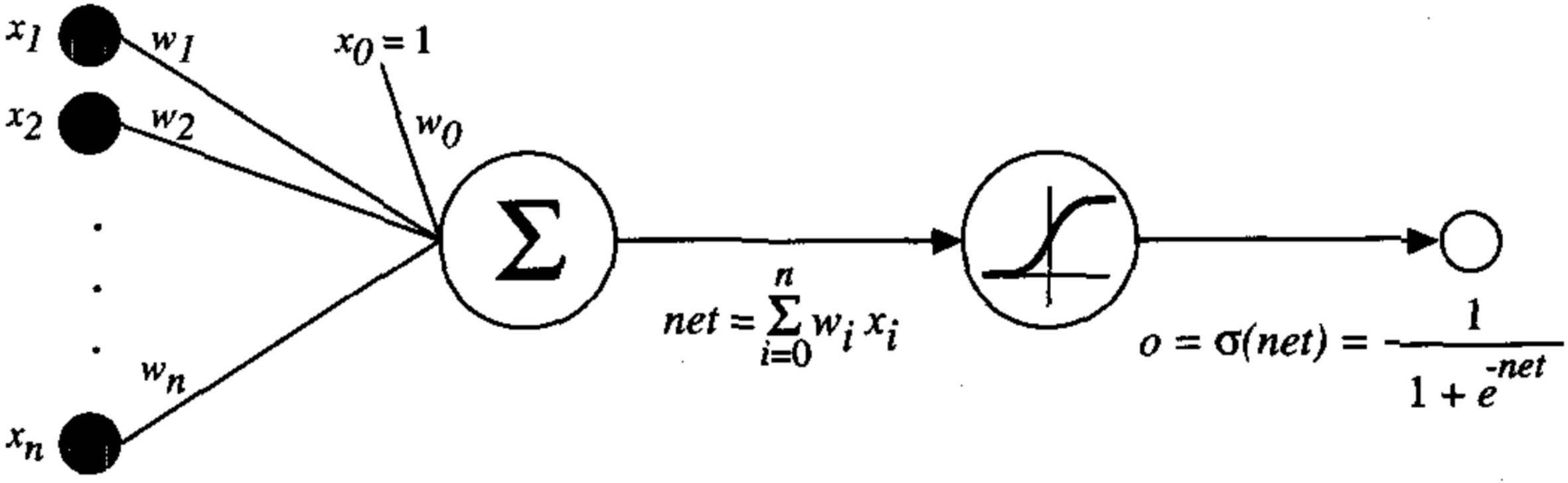
Depiction similar to perceptron:



Note: Inputs $x_1, x_2, x_3, \ldots$ may take any value between 0 and 1 and the output is $\sigma(\mathbf{w} \,.\, \mathbf{x} + b)$

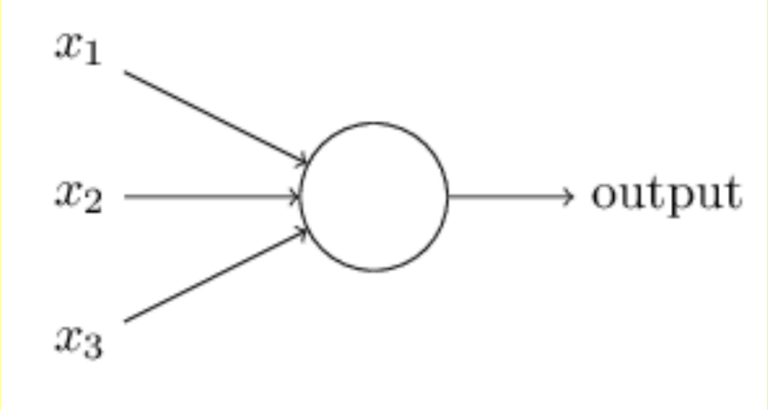
where σ is the sigmoid function

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


x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

Sigmoid Neurons

Depiction similar to perceptron:



A sigmoid neuron with inputs $x_1, x_2, x_3, \ldots$, weights

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

$w_1, w_2, w_3, \ldots$ and bias b:

Sigmoid Neurons

Similarities with perceptron model:

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

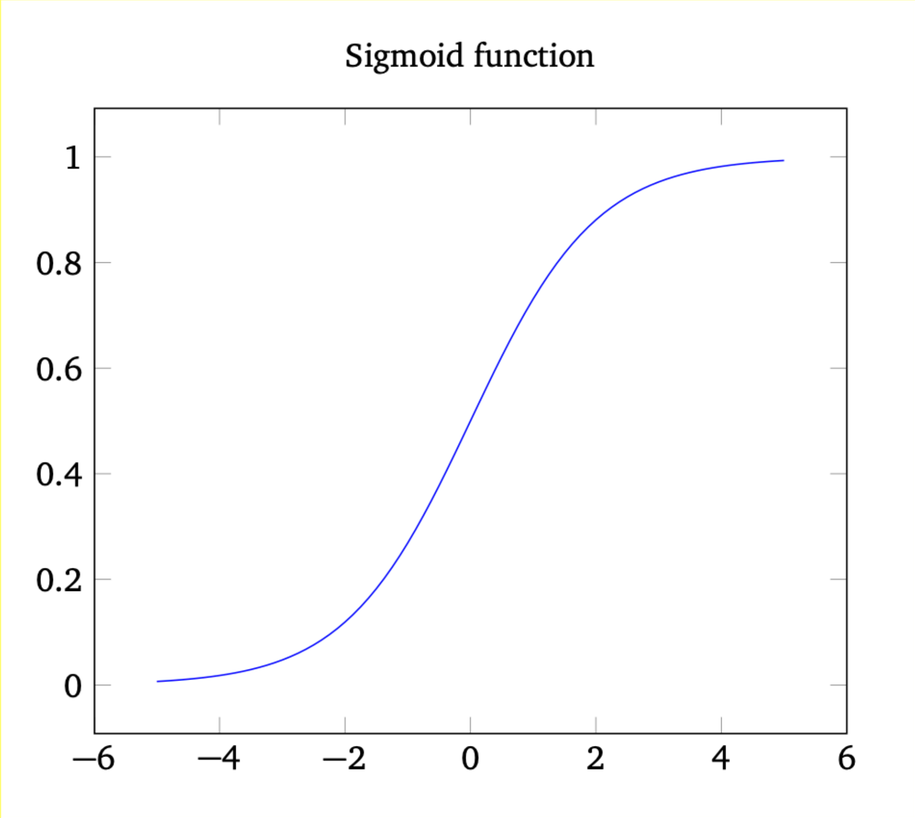
If z=w.x+b is very large, then —> $e^{-z} \equiv 0$ -> $\sigma(z) \equiv 1$

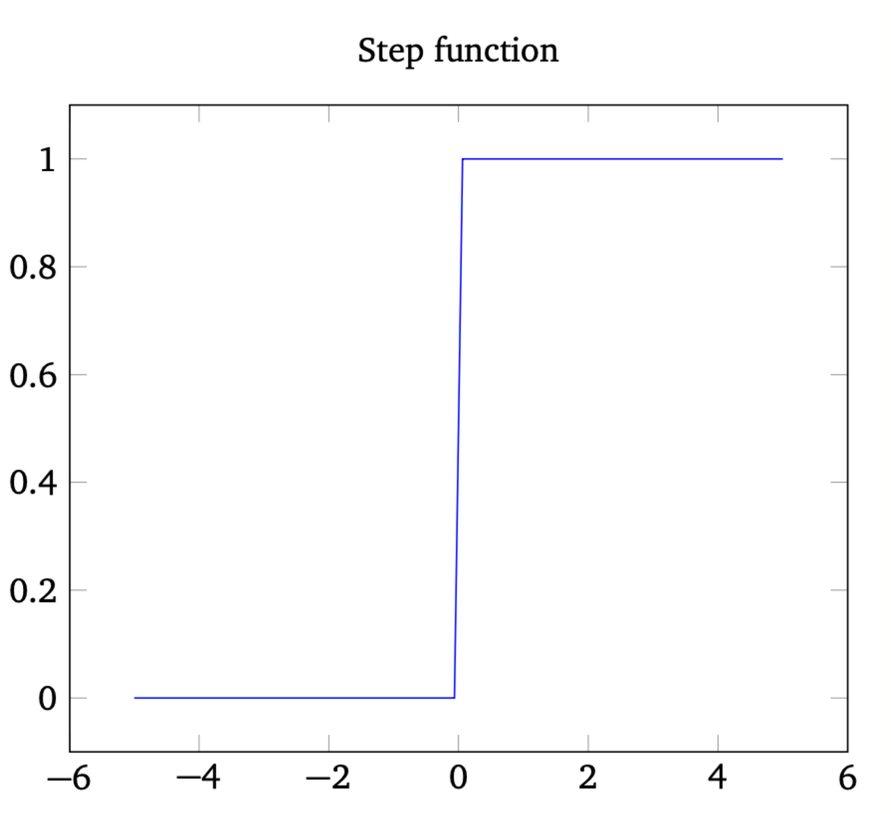
If z=w.x+b is very negative,then -> $e^{-z} \equiv \inf \sigma(z) \equiv 0$

It's only when w · x + b is of modest size that there's much deviation from the perceptron model.

Sigmoid Neurons

Shape of sigmoid is a smoothed out version of a step function.





Sigmoid Neurons

By using the actual σ function we get a smoothed out perceptron.

The smoothness of σ means that small changes Δw_j in the weights and Δb in the bias will produce a small change Δoutput in the output from the neuron.

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

Sigmoid Neurons

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

where the sum is over all the weights, w_j, and $\partial \text{output}/\partial w_j$ and $\partial \text{output}/\partial b$ denote partial derivatives of the output with respect to w_j and b, respectively.

Sigmoid Neurons

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

Δoutput is a linear function of the changes Δw_j and Δb in the weights and bias.

This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output.

Sigmoid Neurons

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

So while sigmoid neurons have much of the same qualitative behavior as perceptrons, they make it much easier to figure out how changing the weights and biases will change the output.

Sigmoid Neurons

Why σ ?

Why this particular form of function?

So, let us call a function, activation function,

$$f(w.x+b) \longrightarrow f(.)$$

When we use a different activation function is that the particular values for the partial derivatives.

Sigmoid Neurons

It turns out that when we compute those partial derivatives later, using σ will simplify the algebra, simply because exponentials have beautiful properties when differentiated.

In any case, σ is commonly-used in work on neural nets.

Sigmoid Neurons

How should we interpret the output from a sigmoid neuron?

One big difference is that sigmoid neurons don't just output 0 or 1.

Any real number between 0 and 1, so values such as 0.173. . . and 0.689. . . are legitimate outputs.

Useful, say, if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network.

Sigmoid Neurons

But, suppose we want the output from the network to indicate either “the input image is a 9” or “the input image is not a 9”.

Obviously, if we set up a convention to deal with this, for example, by deciding to interpret any output of at least 0.5 as indicating a “9”, and any output less than 0.5 as indicating “not a 9”.

Sigmoid Neurons

Note: Because it maps a very large input domain to small range of outputs, it is often referred to as the squashing function of the unit.

The sigmoid function has the useful property that its derivative is easily expressed in terms of its output [in particular, $\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$].

The gradient descent learning rule makes use of this derivative.

Sigmoid Neurons

Note:

Other differentiable functions with easily calculated derivatives are sometimes used in place of σ.

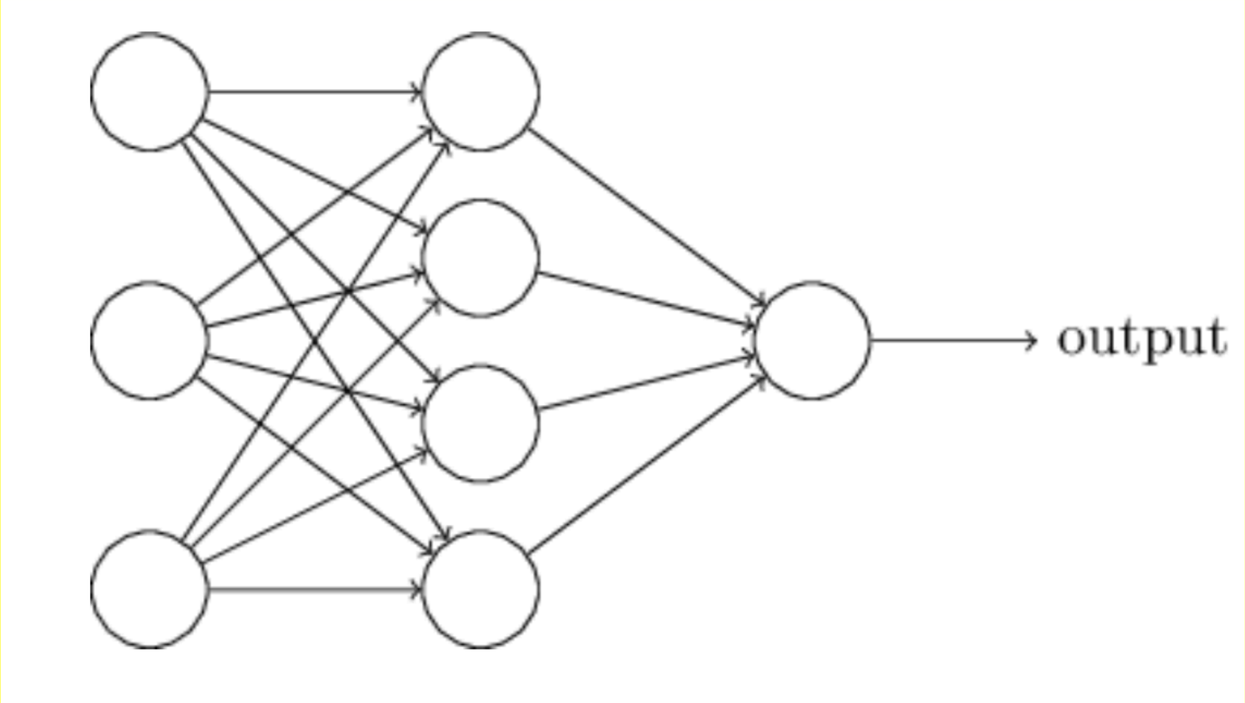
The function tanh is sometimes used in place of the sigmoid function.

The architecture of neural networks

Example: a neural network that can classify handwritten digits.

The architecture of neural networks

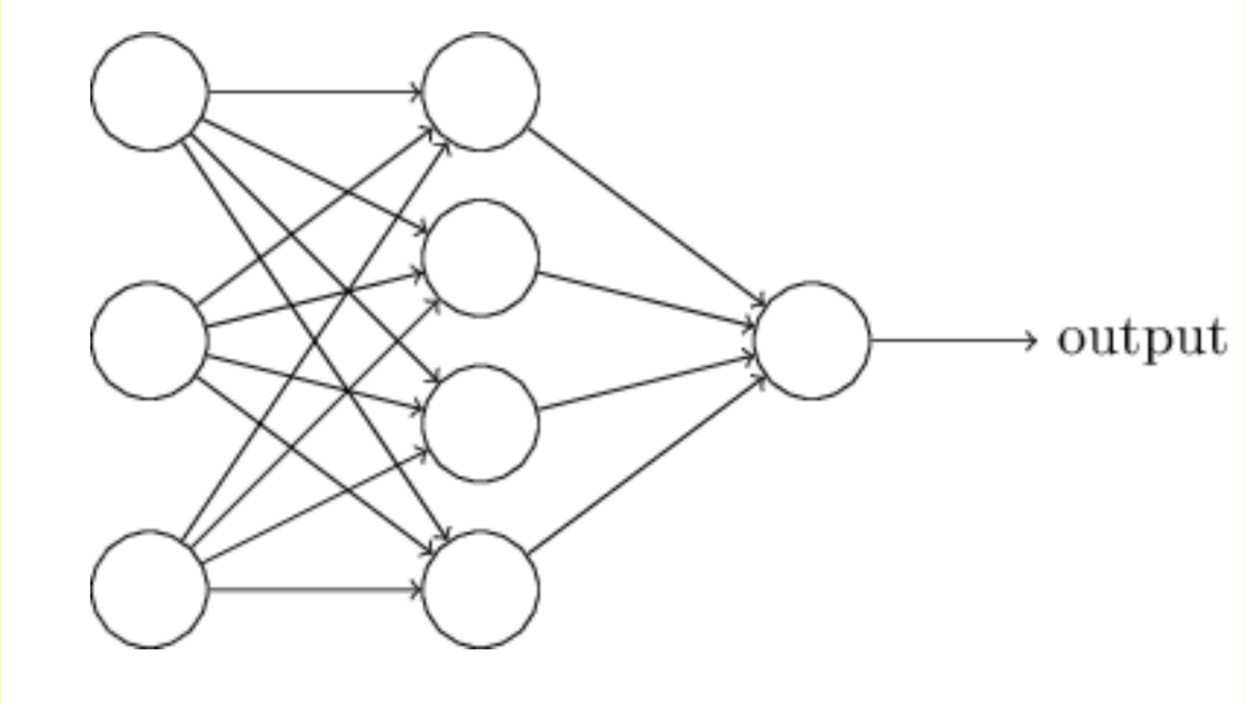
Preparatory:



The leftmost layer is called the input layer, and the neurons within the layer are called input neurons.

The architecture of neural networks

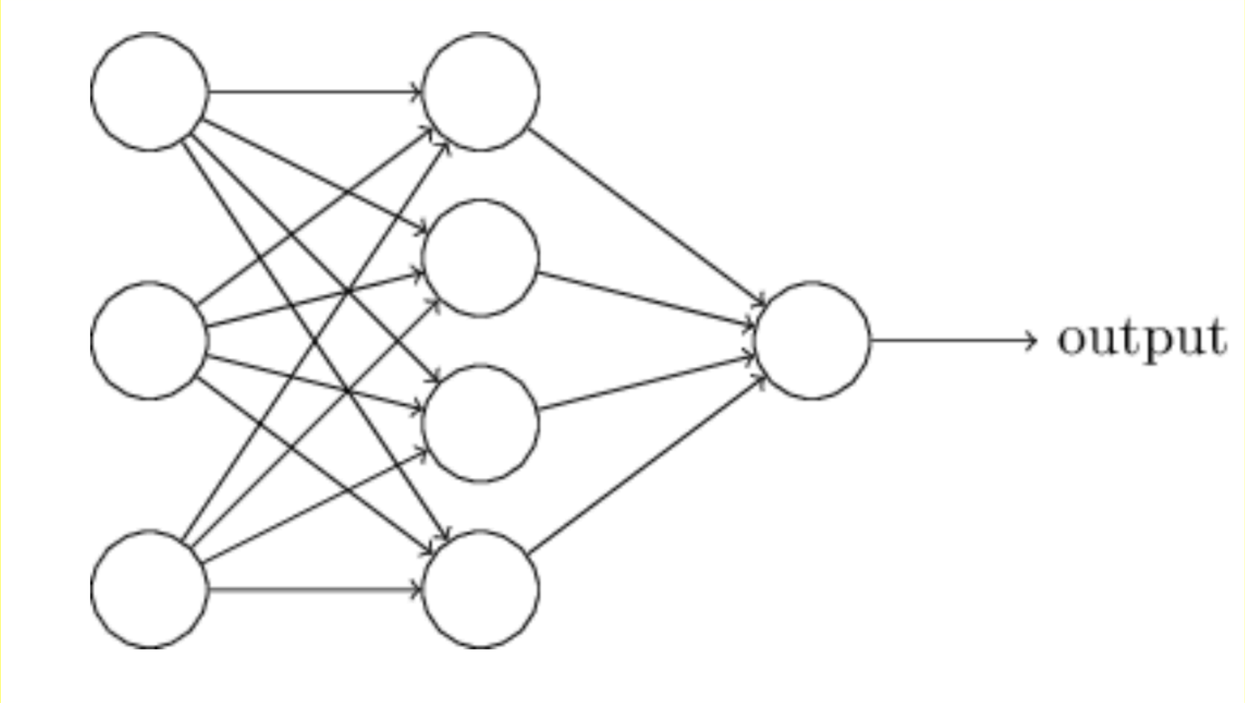
Preparatory:



The rightmost or output layer contains the output neurons, or, as in this case, a single output neuron.

The architecture of neural networks

Preparatory:



The middle layer is called a **hidden** layer, since the neurons in this layer are neither inputs nor outputs.

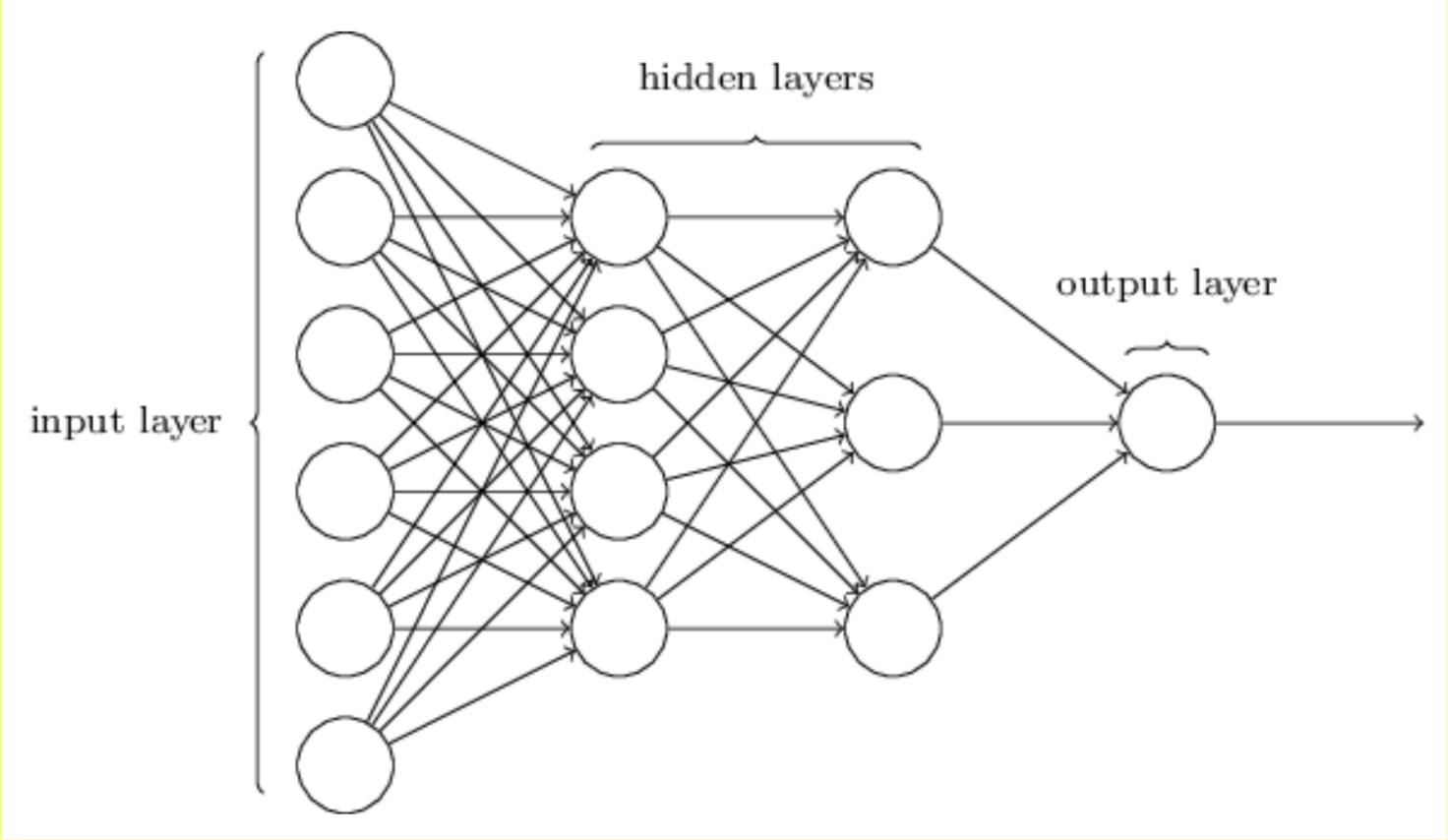
The architecture of neural networks

Preparatory:

The term “hidden” perhaps sounds a little mysterious – the first time I heard the term I thought it must have some deep philosophical or mathematical significance – but it really means nothing more than “not an input or an output”. The network above has just a single hidden layer, but some networks have multiple hidden layers.

The architecture of neural networks

Preparatory:



Four-layer network with two hidden layers.

The architecture of neural networks

Preparatory:

Somewhat confusingly, and for historical reasons, such multiple layer networks are sometimes called multilayer perceptrons or MLPs, despite being made up of sigmoid neurons, not perceptrons.

The architecture of neural networks

Some design thoughts:

- Design of the input and output layers is often straightforward.
- whether a handwritten image depicts a “9” or not?
  - encode the intensities of the image pixels into the input neurons - a natural way.

The architecture of neural networks

Some design thoughts:

- whether a handwritten image depicts a “9” or not?

- if the image is a 64 by 64 greyscale image, then we’d have 4, 096 = 64 × 64 input neurons, with the intensities scaled appropriately between 0 and 1.

- The output layer will contain just a single neuron, with output values of less than 0.5 indicating “input image is not a 9”, and values greater than 0.5 indicating “input image is a 9”.

The architecture of neural networks

Some design thoughts:

- not possible to sum up the design process for the hidden layers with a few simple rules of thumb.
- Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behaviour they want out of their nets.

The architecture of neural networks

Some design thoughts:

- For example, such heuristics can be used to help determine how to trade off the number of hidden layers against the time required to train the network.

The architecture of neural networks

Some design thoughts:

- So far, discussed neural networks where the output from one layer is used as input to the next layer. Such networks are called feedforward neural networks.

The architecture of neural networks

Some design thoughts:

This means there are no loops in the network – information is always fed forward, never fed back.

If we did have loops, we'd end up with situations where the input to the σ function depended on the output. That'd be hard to make sense of, and so we don't allow such loops.

The architecture of neural networks

Some design thoughts:

However, there are other models of artificial neural networks in which feedback loops are possible. These models are called recurrent neural networks.

The architecture of neural networks

Some design thoughts:

The idea is to have neurons which fire for some limited duration of time, before becoming quiescent.

That firing can stimulate other neurons, which may fire a little while later, also for a limited duration. That causes still more neurons to fire, and so over time we get a cascade of neurons firing.

The architecture of neural networks

Some design thoughts:

Loops don't cause problems in such a model, since a neuron's output only affects its input at some later time, not instantaneously.

The architecture of neural networks

Some design thoughts:

Recurrent neural nets have been less influential than feedforward networks, in part because the learning algorithms for recurrent nets are (at least to date) less powerful. But recurrent networks are still extremely interesting. They're much closer in spirit to how our brains work than feedforward networks.

The architecture of neural networks

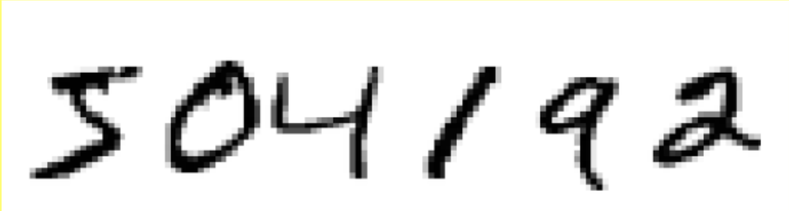
Some design thoughts:

And it's possible that recurrent networks can solve important problems which can only be solved with great difficulty by feedforward networks.

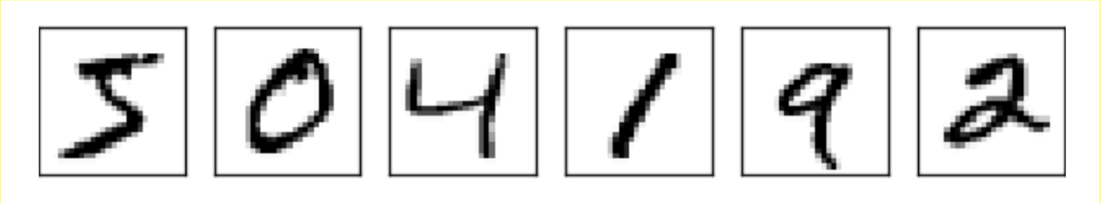
A network to classify hand-written digits

Handwriting recognition:

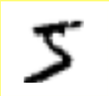
- a. breaking an image containing many digits into a sequence of separate images
- each containing single digit (segmentation problem)



→

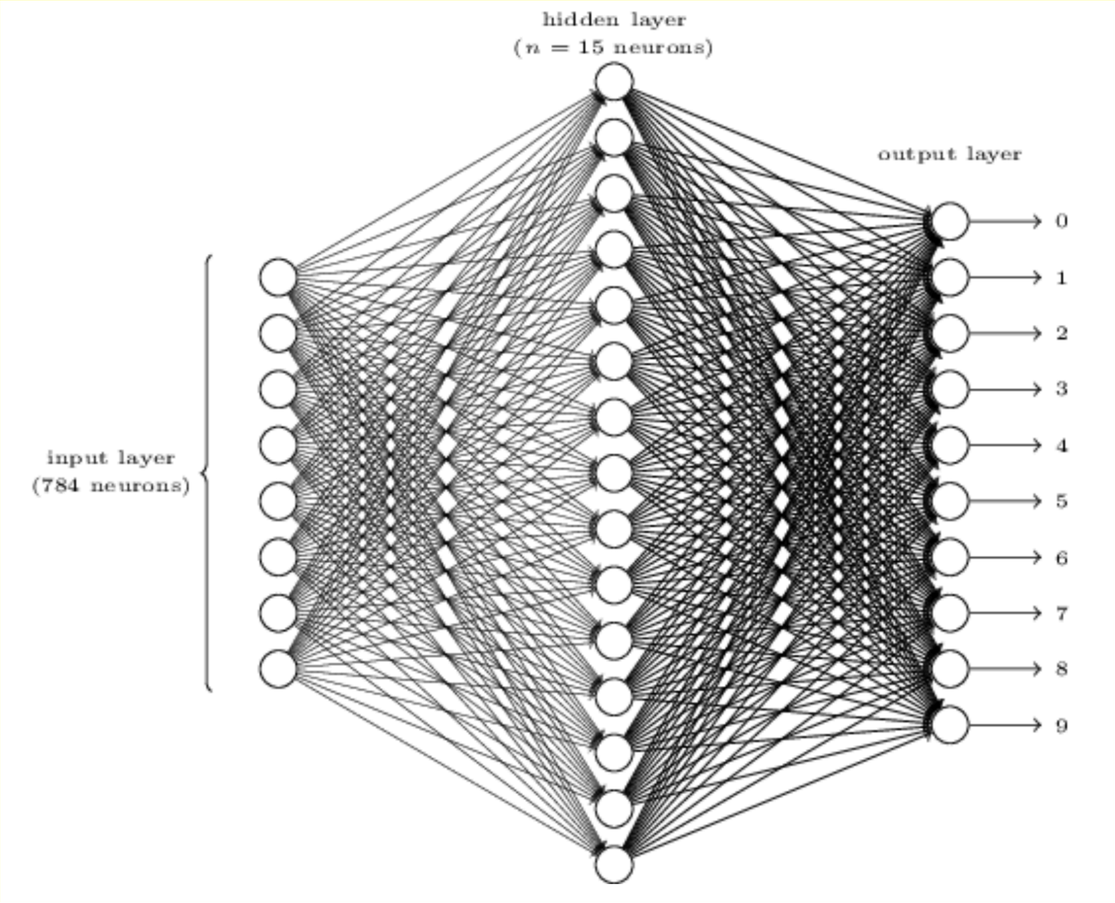


- b. to classify each individual digit

 is a 5.

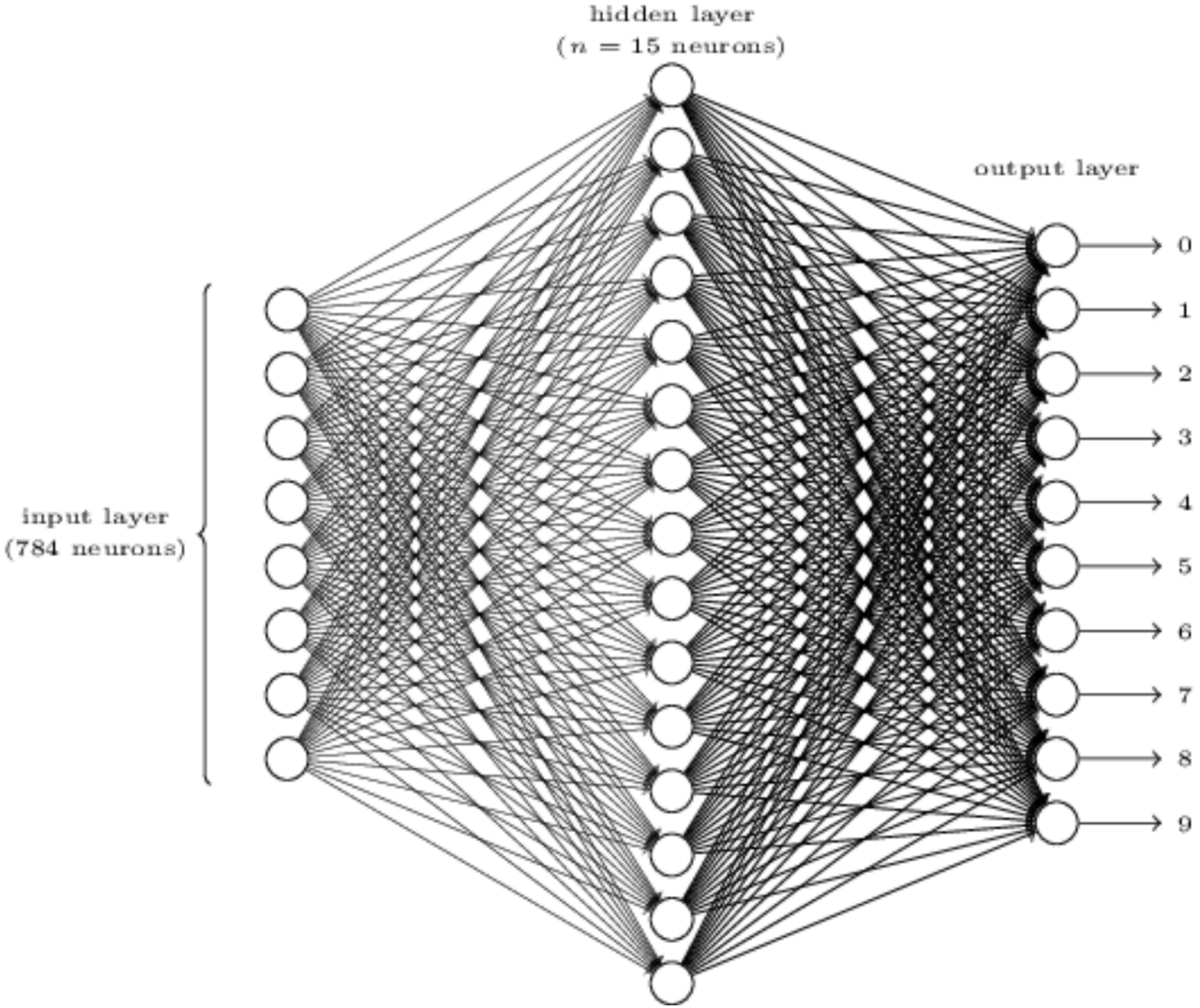
A network to classify hand-written digits

A three-layer neural network to recognise individual digits:



A netw s

A thre dividual digits:



A network to classify hand-written digits

Input layer:

— neurons encoding the values of the input pixels.

— our training data for the network will consist of many 28 x 28 pixel images of scanned handwritten digits, and so the input layer contains $784 = 28 \times 28$ neurons.

— The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0 representing black and in between values.

A network to classify hand-written digits

Hidden Layer

— n, number of neutrons in this layer.

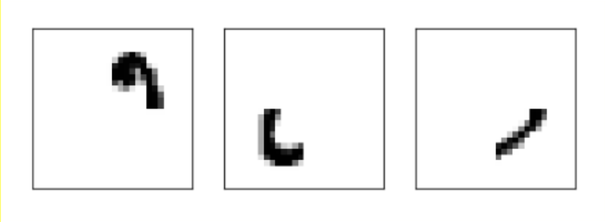
— we can experiment with different values for n.

The example shown illustrates a small hidden layer, containing just n = 15 neurons.

A network to classify hand-written digits

A very crude way of assimilating the job being done by the neural network:

First neutron in hidden layer detects:

Second, third, and fourth neutron in hidden layer detects: 

——————————>

A network to classify hand-written digits

Learning with gradient descent:

We need a data set to learn from – a so-called **training data set**.

MNIST data set, which contains tens of thousands of scanned images of handwritten digits, together with their correct classifications.

Modified subset of two data sets collected by NIST, National Institute of Standards and Technology, US.

Learning with gradient descent:

MNIST has two parts:

First: 60,000 images to be used as training data. These images are scanned handwriting samples from 250 people, half of whom were US Census Bureau employees, and half of whom were high school students. The images are greyscale and 28 by 28 pixels in size.

Learning with gradient descent:

MNIST has two parts:

Second: 10,000 (28 x 28) images as test data.

We'll use the test data to evaluate how well our neural network has learned to recognize digits.

— The test data was taken from a different set of 250 people than the original training data (albeit still a group split between Census Bureau employees and high school students).

Learning with gradient descent:

Let us use x to denote a training input.

Each training input x is 28 × 28 = 784-dimensional vector.

Each entry in the vector represents the grey value for a single pixel in the image.

Corresponding desired output by y = y(x), where y is a 10-dimensional vector. For example, x, depicts a 6, then $y(x) = (0,0,0,0,0,0,1,0,0,0)^T$ is the desired output.

Learning with gradient descent:

Goal: To design an algorithm which finds weights and biases so that the output from the network approximates y(x) for all training inputs x.

Cost function (loss or objective function):

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

Learning with gradient descent:

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

C - the quadratic cost function; mean squared error or just MSE.

C(w, b) is non-negative, since every term in the sum is non-negative. The cost C(w, b) becomes small, i.e., C(w, b) ≈ 0, when y(x) is approximately equal to the output, a, for all training inputs, x.

Learning with gradient descent:

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

So our training algorithm has done a good job if it can find weights and biases so that $C(w, b) \approx 0$. By contrast, it's not doing so well when $C(w, b)$ is large – that would mean that $y(x)$ is not close to the output a for a large number of inputs.

Learning with gradient descent:

So the aim of our training algorithm will be to minimize the cost C(w, b) as a function of the weights and biases.

— In other words, we want to find a set of weights and biases which make the cost as small as possible.

—We'll do that using an algorithm known as gradient descent.

Learning with gradient descent:

Why quadratic cost?

If we use a smooth cost function like the quadratic cost it turns out to be easy to figure out how to make small changes in the weights and biases so as to get an improvement in the cost.

That's why we focus first on minimizing the quadratic cost, and only after that will we examine the classification accuracy.

Learning with gradient descent:

Our goal in training a neural network is to find weights and biases which minimize the quadratic cost function $C(w, b)$.

This is a well-posed problem, but it's got a lot of distracting structure as currently posed – the interpretation of w and b as weights and biases, the σ function lurking in the background, the choice of network architecture, MNIST, and so on.

Learning with gradient descent:

Forget all about specific form of the cost function, the connection to neural networks, and so on.

Instead, imagine that we've simply been given a function of many variables and we want to minimize that function.

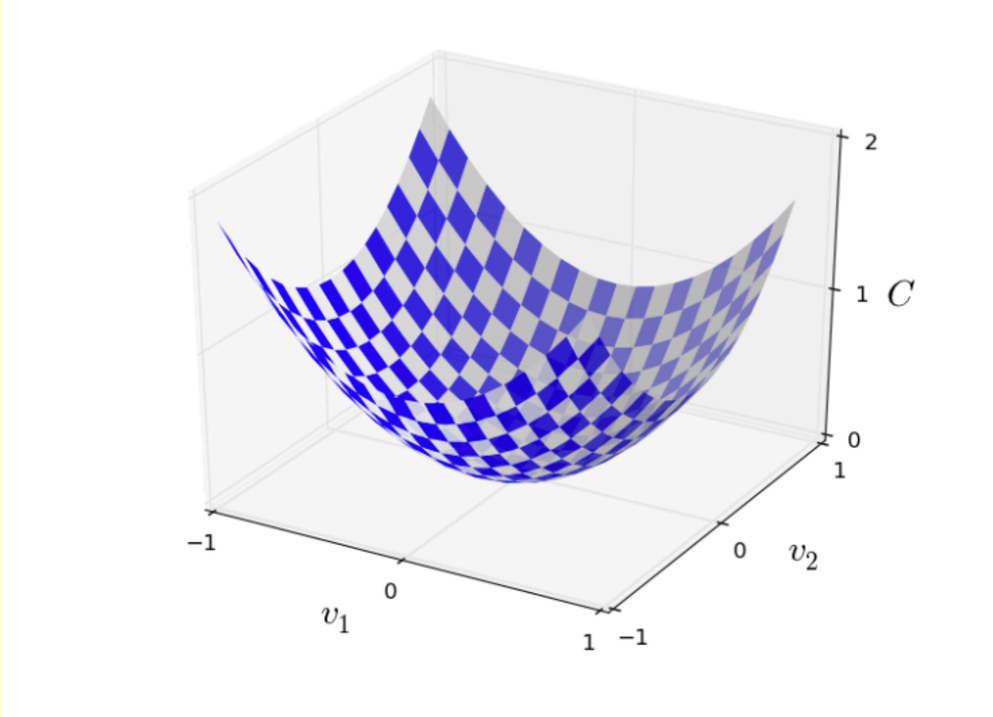
We will develop a technique called gradient descent which can be used to solve such minimization problems.

Learning with gradient descent:

To minimize C(v) it helps to imagine C as a function of just two variables, $v_1 \ and \ v_2$.

To find where C achieves its global minimum .

Learning with gradient descent:



A general function, C, may be a complicated function of many variables, and not possible to just eyeball the graph to find the minimum.

Learning with gradient descent:

Use calculus to try to find the minimum analytically.

We could compute derivatives and then try using them to find places where C is an extremum.

With some luck that might work when C is a function of just one or a few variables.

But very difficult when we have many more variables.

Learning with gradient descent:

And for neural networks we'll often want far more variables – the biggest neural networks have cost functions which depend on billions of weights and biases in an extremely complicated way. Using calculus to minimize that just won't work!

Learning with gradient descent:

So calculus doesn't work.

— We start by thinking of our function as a valley.

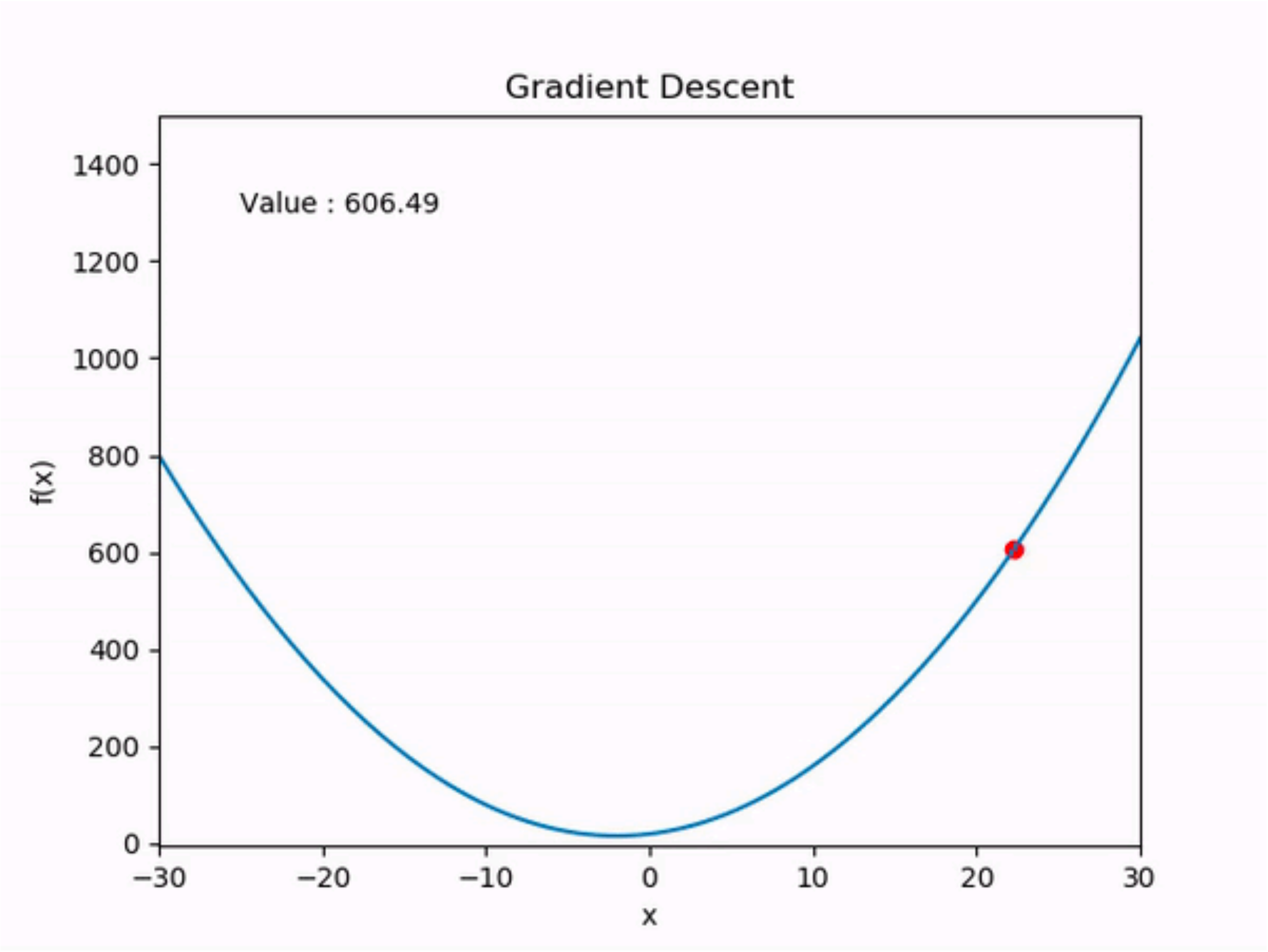
— And we imagine a ball rolling down the slope of the valley.

— Our everyday experience tells us that the ball will eventually roll to the bottom of the valley.

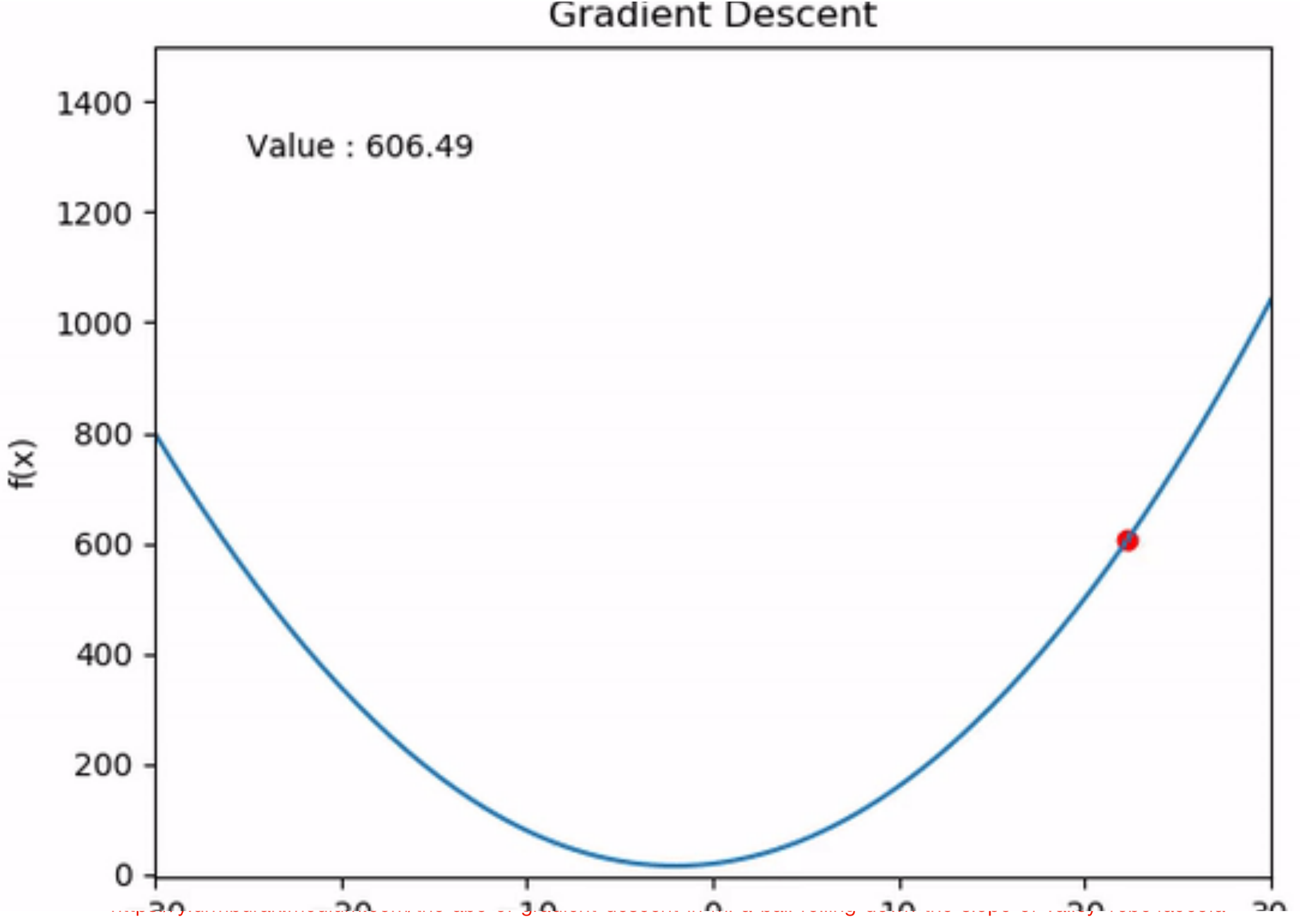
— Perhaps we can use this idea as a way to find a minimum for the function?

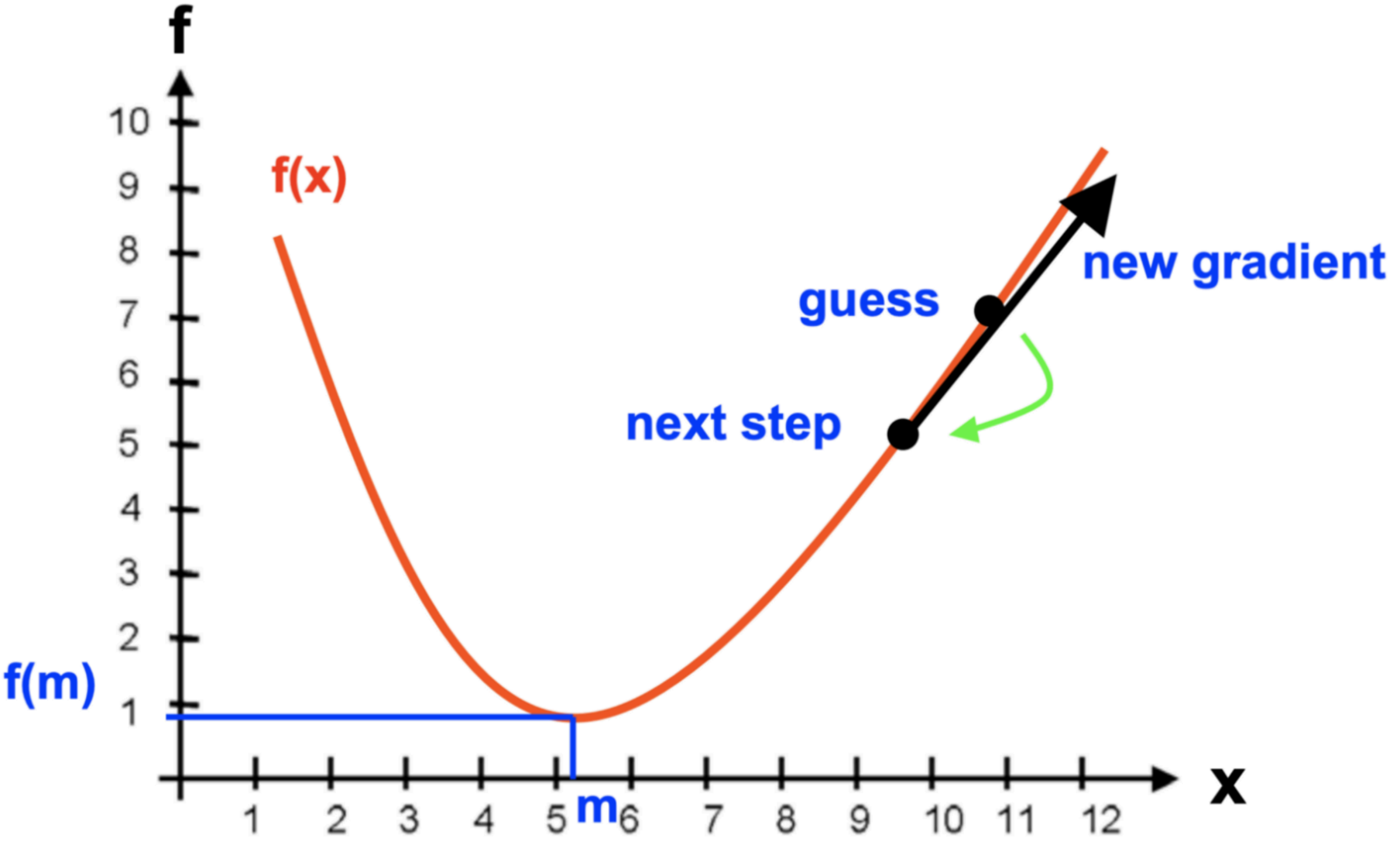
Learning with gradient descent:

We'd randomly choose a starting point for an (imaginary) ball, and then simulate the motion of the ball as it rolled down to the bottom of the valley. We could do this simulation simply by computing derivatives (and perhaps some second derivatives) of C – those derivatives would tell us everything we need to know about the local "shape" of the valley, and therefore how our ball should roll.



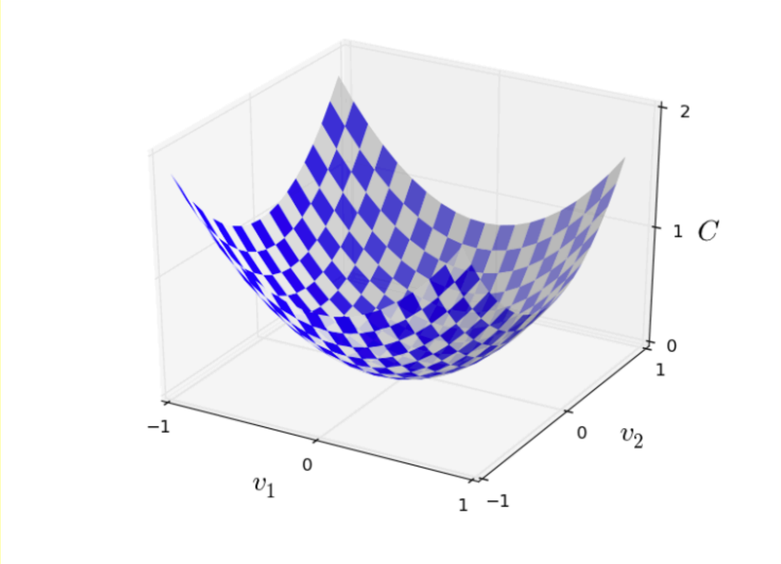
https://yldrmburak.medium.com/the-abc-of-gradient-descent-in-ml-a-ball-rolling-down-the-slope-of-valley-1eb64a9c8fa


Gradient Descent
Value : 606.49
f(x)
1400
1200
1000
800
600
400
200
0



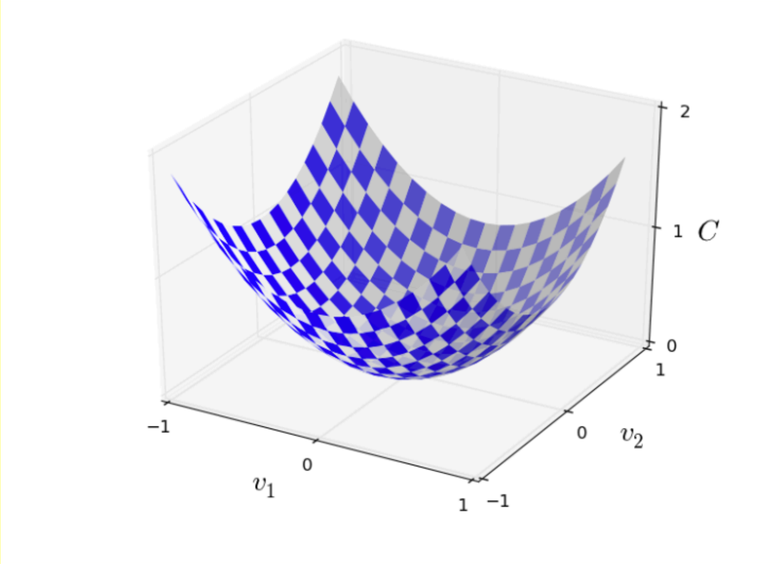
https://yldrmburak.medium.com/the-abc-of-gradient-descent-in-ml-a-ball-rolling-down-the-slope-of-valley-1eb64a9c8fa

Learning with gradient descent:



Let's think about what happens when we move the ball a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction.

Learning with gradient descent:



Calculus says that C changes as follows:

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

Learning with gradient descent:

We need to find a way to choose Δv_1 and Δv_2 in such a way that ΔC is negative.

Let us represent the changes Δv as vector of changes in v, $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$ where T is the transpose operation, turns row vector into column vector.

Learning with gradient descent:

Let us also define the gradient of C to be the vector of partial derivatives:

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

Well, just be happy with, " ∇C is a gradient vector"

Learning with gradient descent:

From $\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$, we can write,

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

Learning with gradient descent:

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

How to choose Δv to make ΔC negative?

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

Then, $$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$$ guarantees that $\Delta C \leqslant 0$.

Learning with gradient descent:

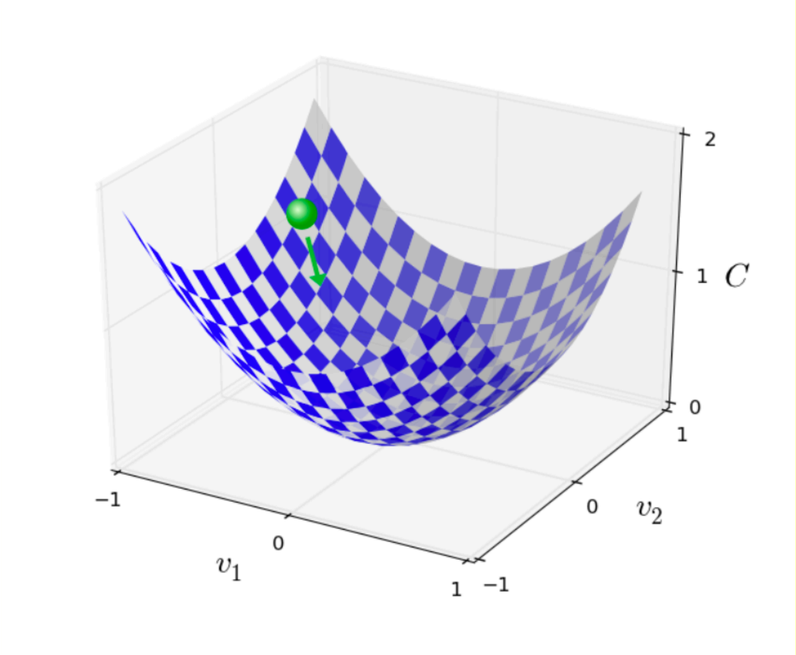
Let us use $\Delta v = -\eta \nabla C,$ and move the balls position by that amount giving the new position as

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

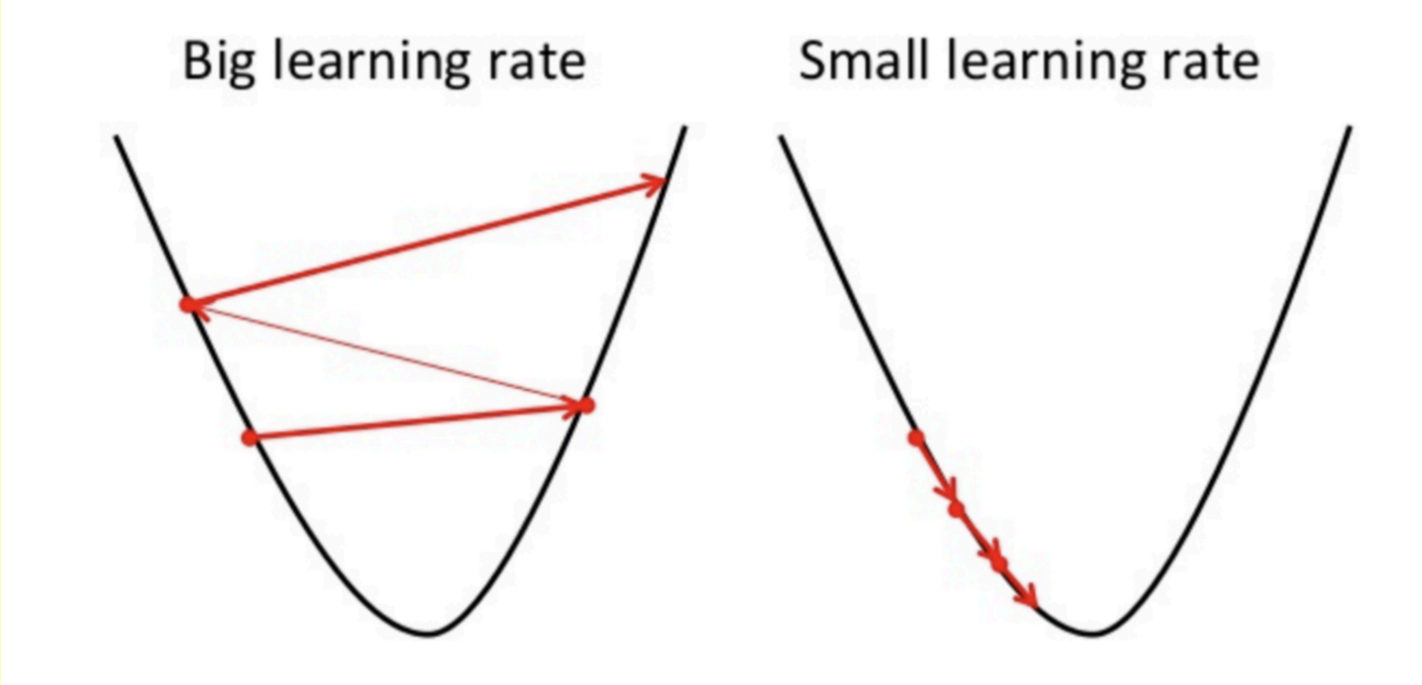
Then we'll use this update rule again, to make another move. If we keep doing this, over and over, we'll keep decreasing C until – we hope – we reach a global minimum.

Learning with gradient descent:

Repeatedly compute the gradient ∇C, and then to move in the opposite direction, “falling down” the slope of the valley.



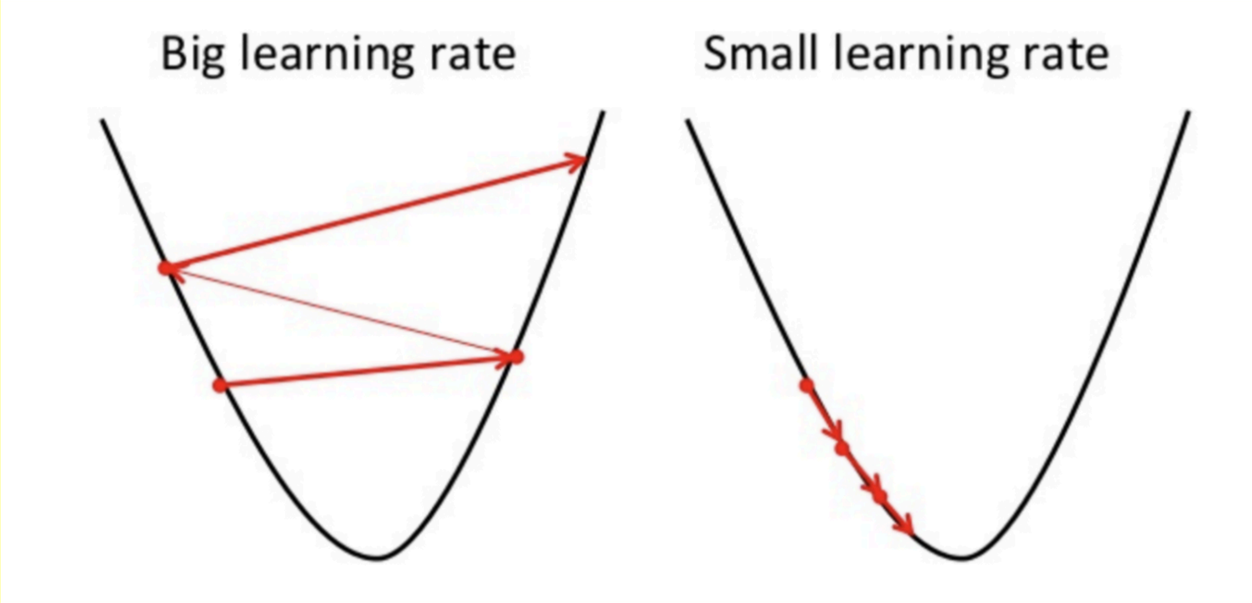
Learning with gradient descent:



To make gradient descent work correctly, choose the learning rate η to be small enough that Equation

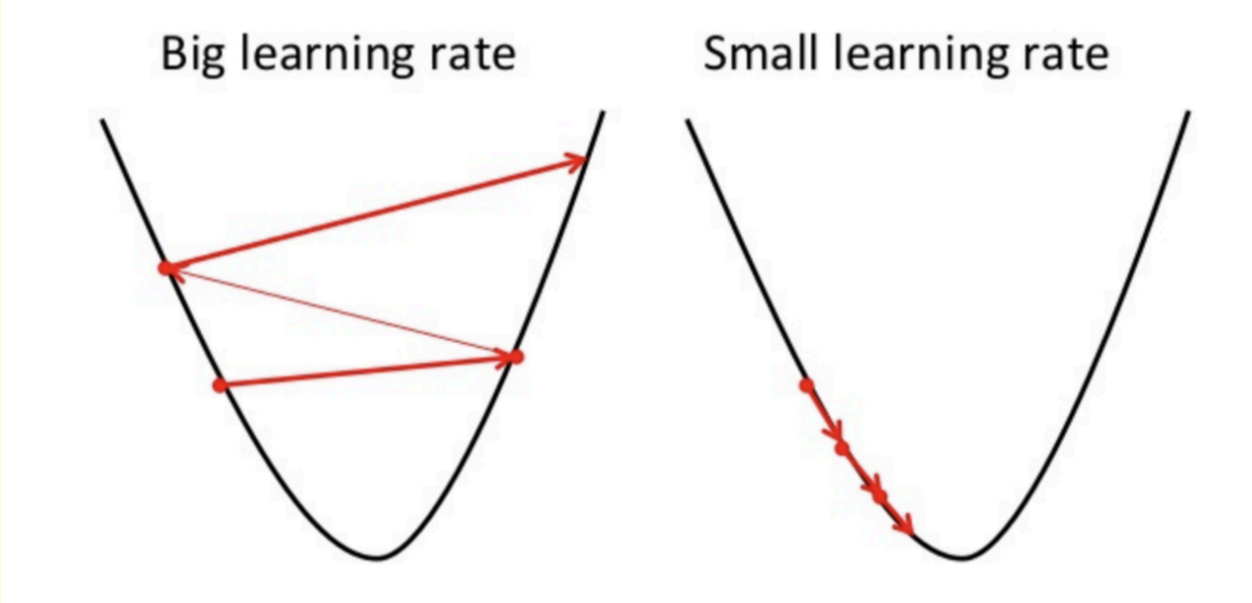
$\Delta C \approx \nabla C \cdot \Delta v$ is a good approximation.

Learning with gradient descent:



If η to be too small, since that will make the changes Δv tiny, and thus the gradient descent algorithm will work very slowly. (Right side)

Learning with gradient descent:



η is often varied so the Equation remains a good approximation, but the algorithm isn't too slow.

Learning with gradient descent:

C is a function of many more variables.

C is a function of m variables, $v_1, v_2, \ldots, v_m$

Then the change ΔC produced due to a small change in variables, $\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$ is

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

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

Learning with gradient descent:

C is a function of many more variables.

…

Now, we can use

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

and repeatedly applying the update rule

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

we can reach the minimum gradient.

Learning with gradient descent:

C is a function of many more variables.

Update rule

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

defines the gradient descent algorithm.

Learning with gradient descent:

The rule doesn't always work – several things can go wrong and prevent gradient descent from finding the global minimum of C.

But, in practice gradient descent often works extremely well, and in neural networks, it's a powerful way of minimizing the cost function, and so helping the net learn.

Learning with gradient descent:

So gradient descent can be viewed as a way of taking small steps in the direction which does the most to immediately decrease C.

Learning with gradient descent:

Challenge:

Finding alternatives to gradient descent is an active area of investigation.

Gradient Descent and Neural Network:

To find the weights w_k and biases b_l such that

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

The above cost function is minimised.

Gradient Descent and Neural Network:

Let's restate the gradient descent update rule, with the weights and biases replacing the variables v_j.

The gradient vector ∇C has components

$$\partial C/\partial w_k$$

$$\partial C/\partial b_l$$

Gradient Descent and Neural Network:

Gradient descent update rules now:

$$w_k \quad \rightarrow \quad w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \quad \rightarrow \quad b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Gradient Descent and Neural Network:

Note:

The quadratic cost function is

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

It is an average over costs $C_x \equiv \frac{||y(x) - a||^2}{2}$ **for individual training examples.**

Gradient Descent and Neural Network:

Note:

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

It is an average over costs $C_x \equiv \frac{||y(x) - a||^2}{2}$ **for individual training examples.**

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

Average over C_x (each training example):

$$C = \frac{1}{n}\sum_x C_x \text{ with } C_x \equiv \frac{||y(x) - a||^2}{2}$$

So, to compute gradient ∇C,

we have to compute, ∇C_x for each training input x

Then, average them $\nabla C = \frac{1}{n}\sum_x \nabla C_x$

With large number of training inputs —> slow.

Stochastic Gradient Descent

Used to speed up learning.

To estimate the gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs.

By averaging over this small sample we can quickly get a good estimate of true gradient ∇C.

1. Randomly picking out a small number m of randomly chosen training inputs.

Mini-batch: $X_1, X_2, \ldots, X_m$

2. If sample size m is large enough then the average value of the ∇C_x will be roughly equal to the average over all ∇C_x.

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

where the second sum is over the entire set of training data.

Swapping sides: $$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}$$

So, we can estimate the overall gradient by computing gradients just for the randomly chosen mini-batch.

In the context of learning in neural networks:

Suppose, wk and bl are the weights and biases.

Then stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs, and training with those.

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

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

where the sums are over all the training examples X_j in the current mini-batch.

Epoch:

Then we pick out another randomly chosen mini-batch and train with those. And so on, until we've exhausted the training inputs, which is said to complete an epoch of training. At that point we start over with a new training epoch.

Task: Prepare a note on online or incremental learning.

Hint: Mini-batch size = 1

Toward deep learning

While our neural network gives impressive performance, that performance is somewhat mysterious. The weights and biases in the network were discovered automatically. And that means we don't immediately have an explanation of how the network does what it does.

In early days of AI research people hoped that the effort to build an AI would also help us understand the principles behind intelligence and, maybe, the functioning of the human brain. But perhaps the outcome will be that we understand neither the brain nor how AI works.

An image shows a human face or not?

- Use the pixels in the image as input
- Output: either “Yes, it’s a face” or “No, it’s not a face”

Let us design a network by hand.

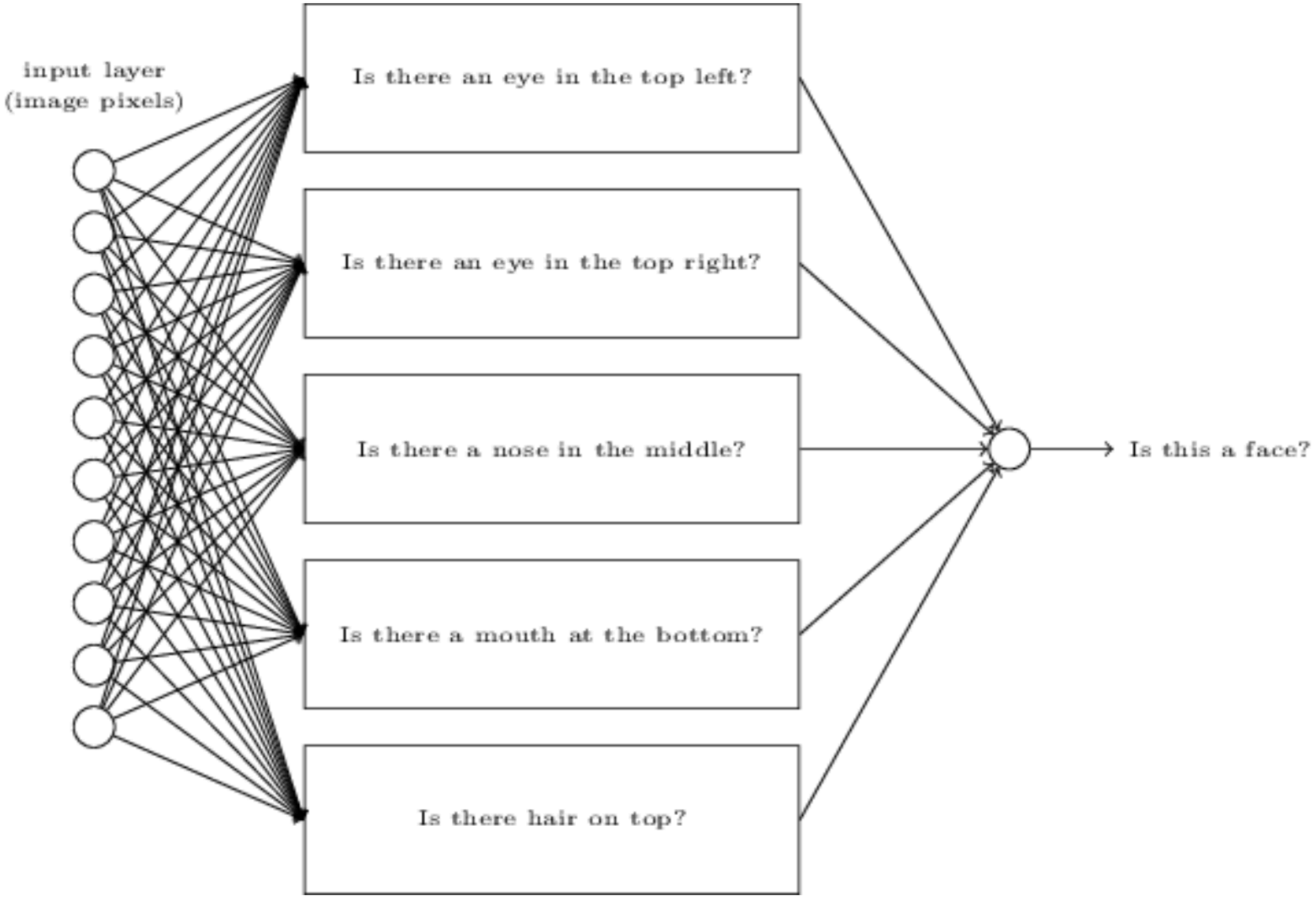
By choosing appropriate weights and biases.

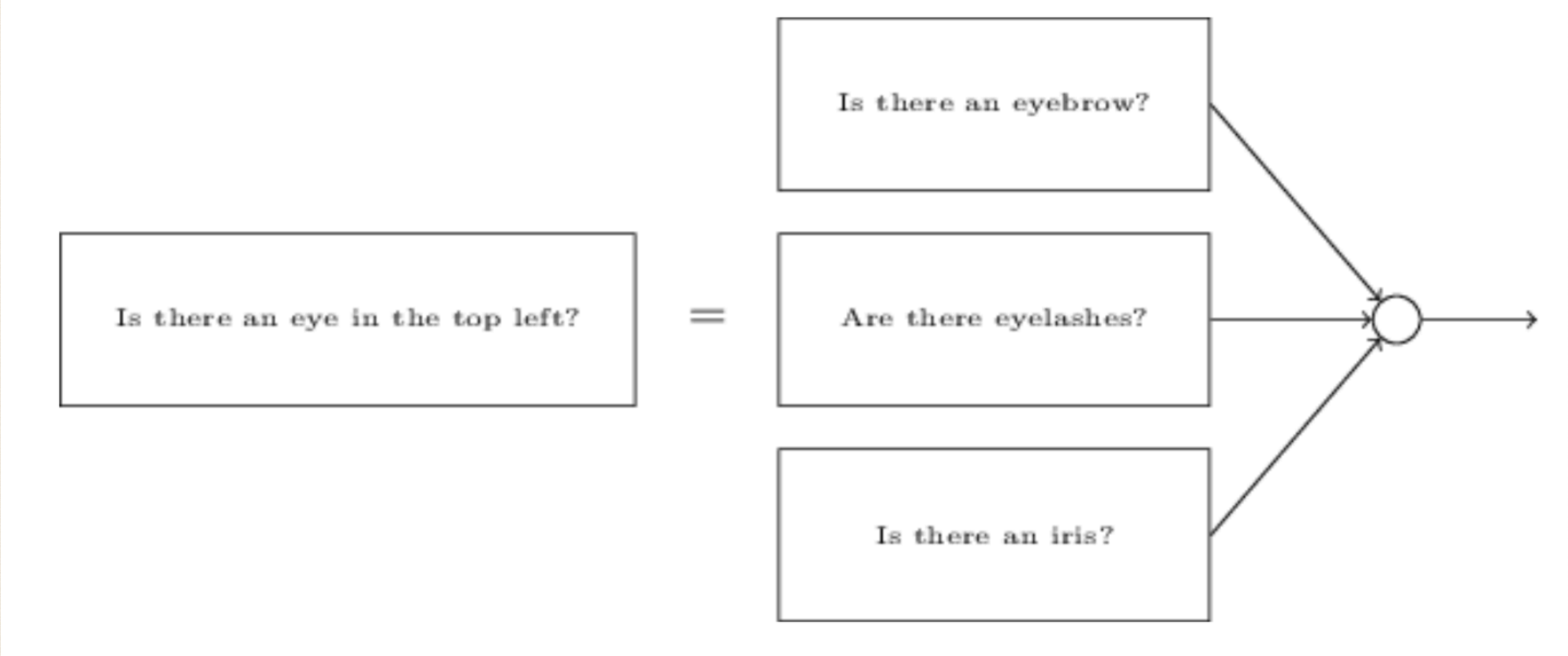
Decompose the problem into sub-problems:

- Does the image have an eye in the top left?
- Does it have an eye in the top right?
- Does it have a nose in the middle?
- Does it have a mouth in the bottom middle?
- Is there hair on top?

If the answers to several of these questions are “yes”, or even just “probably yes”, then the image is likely to be a face.

Conversely, if the answers to most of the questions are “no”, then the image probably isn’t a face.


input layer
(image pixels)
Is there an eye in the top left?
Is there an eye in the top right?
Is there a nose in the middle?
Is there a mouth at the bottom?
Is there hair on top?
Is this a face?

- It's also plausible that the sub-networks can be decomposed.
- 

- Those questions too can be broken down, further and further through multiple layers.

The end result is a network which breaks down a very complicated question – does this image show a face or not – into very simple questions answerable at the level of single pixels.

It does this through a series of many layers, with early layers answering very simple and specific questions about the input image, and later layers building up a hierarchy of ever more complex and abstract concepts.

- Networks with this kind of many-layer structure – two or more hidden layers – are called *deep neural networks*.

Researchers in the 1980s and 1990s tried using stochastic gradient descent and backpropagation to train deep networks. Unfortunately, except for a few special architectures, the networks would learn, but very slowly, and in practice often too slowly to be useful.

How the backpropagation algorithm works

We saw how neural networks can learn their weights and biases using the gradient descent algorithm.

Now, we will see a fast algorithm for computing such gradients, an algorithm known as *backpropagation*.

The backpropagation algorithm was originally introduced in the 1970s, but its importance wasn't fully appreciated until a famous 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams.

That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural nets to solve problems which had previously been insoluble.

Today, the backpropagation algorithm is the workhorse of learning in neural networks.

At the heart of backpropagation is an expression for the partial derivative

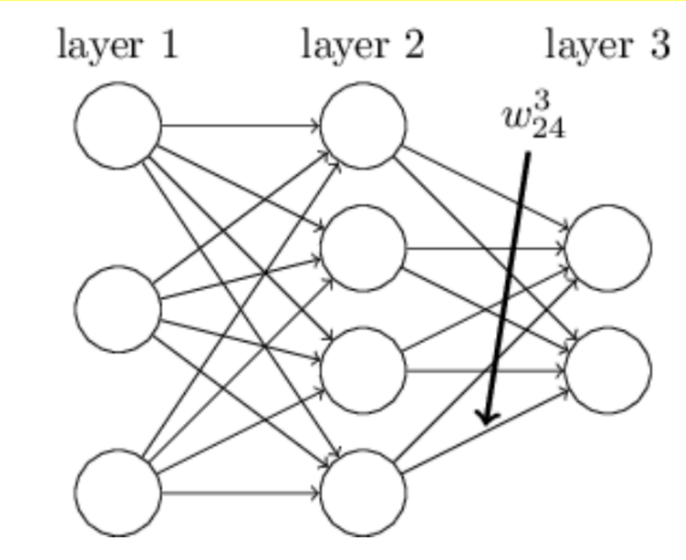
$$\partial C / \partial w$$

of the cost function C with respect to any weight w (or bias b) in the network.

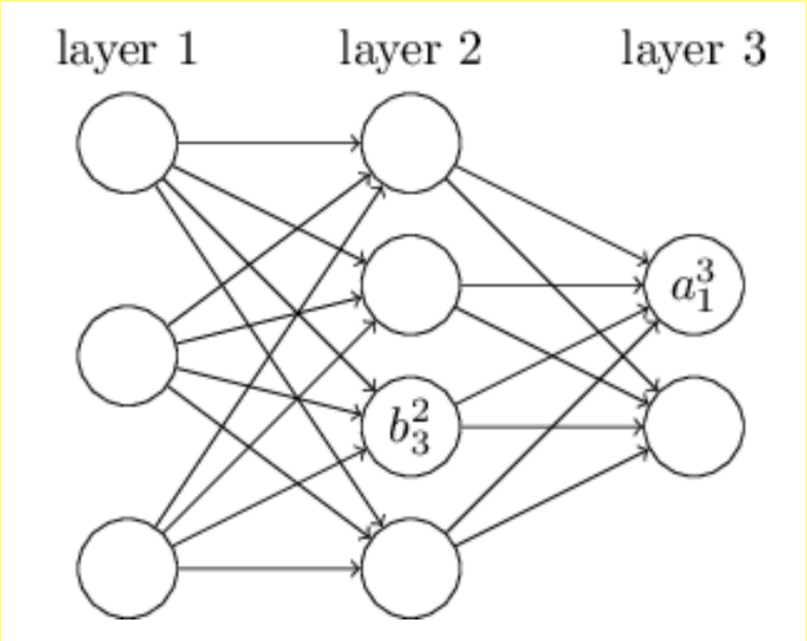
The expression tells us how quickly the cost changes when we change the weights and biases.

It actually gives us detailed insights into how changing the weights and biases changes the overall behaviour of the network.

Notations:



w^l_{jk} is the weight from the kth neuron in the $(l-1)^{\text{th}}$ layer to the jth neuron in the lth layer



b^l_j = the bias of j-th neuron in l-th layer

a^l_j=activation of j-th neuron in l-th layer

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

Compact form: $a^l = \sigma(w^l a^{l-1} + b^l)$

Notations:

To compute a_l, we compute the intermediate quantity:

$$z^l \equiv w^l a^{l-1} + b^l$$

z^l is the weighted input vector to the neurone in layer l.

z^l has the components:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

More compact expression for activation:

$$a^l = \sigma(z^l)$$

Assumptions about the cost function:

$$C = \frac{1}{2n} \sum_x \left\| y(x) - a^L(x) \right\|^2$$

where: n - total number of training examples; sum is over individual training examples, x; $y = y(x)$ is corresponding desired output; L - number of layers; $a^L = a^L(x)$ vector of activations output from the network when x is input.

1. The cost function can be written as an average $C = \frac{1}{n} \sum_x C_x$ over cost functions Cx for individual training examples, x

For quadratic cost function $C_x = \frac{1}{2} \| y - a^L \|^2$

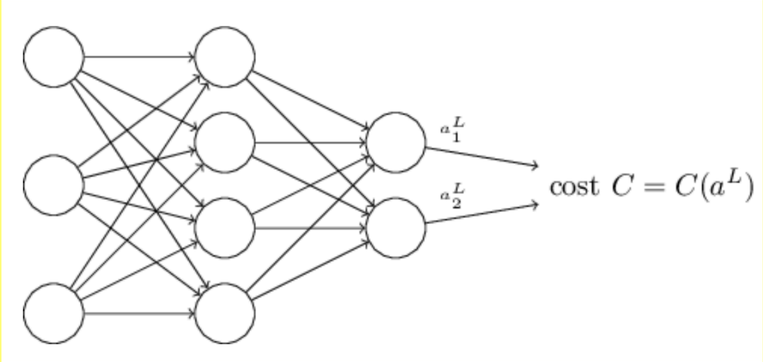
Reason: Backpropagation computes the partial derivatives,

$\partial C_x / \partial w$ *and* $\partial C_x / \partial w$ for a single training example x

Then it recovers $\partial C / \partial w$ *and* $\partial C / \partial w$ by averaging over training examples.

X is fixed, lets write C for Cx.

2. It can be written as a function of the outputs from the neural network:



The quadratic function satisfies this, since:

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2$$

And so it makes sense to regard C as a function of the output activations aL alone, with y merely a parameter that helps define that function.

Hadamard product, $s \circ t$

Elementwise product of the two vectors

$$(s \odot t)_j = s_j t_j$$

Example

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

Four fundamental equations behind backpropagation:

How changing the weights and biases in a network changes the cost function.

Hence, we need to compute

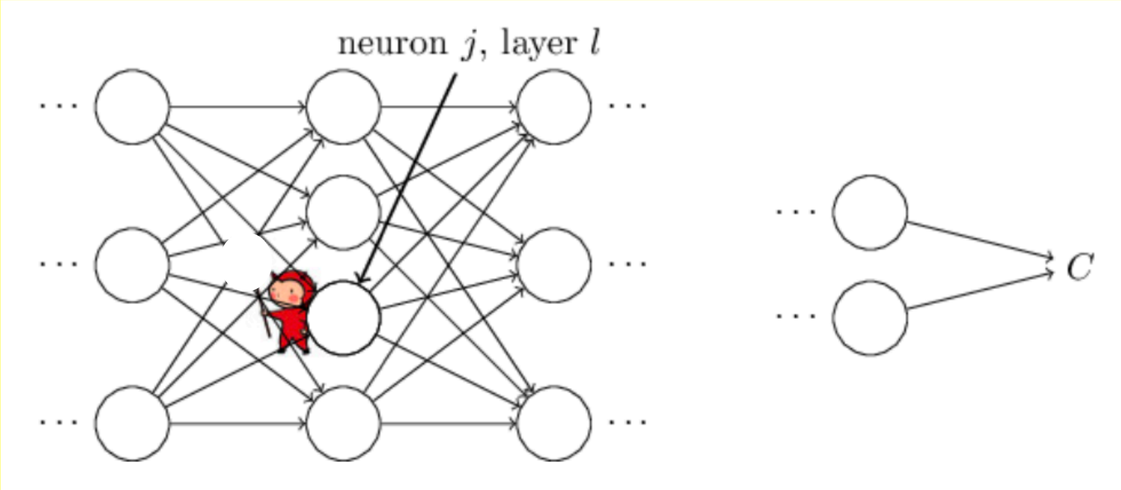
$$\frac{\partial C}{\partial w_{jk}^l} \text{ and } \frac{\partial C}{\partial b_j^l}$$

We introduce:

δ_j^l = error in j-th neuron in l-th layer.

Backpropagation will compute δ_j^l and then relate δ_j^l to $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$

A demon at 4th neuron of layer l:



Input comes in and demon messes with the neutron operation:

Adds a little Δz_j^l to the neutron weighted input so that the neutron outputs

$\sigma(z_j^l + \Delta z_j^l)$ instead of $\sigma(z_j^l)$.

This change propagates through later layers in the network, finally causing the overall cost to change by an amount

$$\frac{\partial C}{\partial z_j^l} \Delta z_j^l$$

Demon is a good demon:

trying to improve the cost

Trying to find a δz_j^l which makes the cost smaller.

Suppose $\partial C / \partial z_j^l$ has a large value (+ve or -ve)

The demon can lower the cost by choosing Δz_j^l to have opposite sign of $\partial C / \partial z_j^l$.

If $\partial C / \partial z_j^l$ is close to zero, then the demon cannot do much.

"The neuron is already pretty near optimal"

With this in the background, we define error of neuron j in layer l as

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$$

δ^l is the vector of errors of layer l.

Backpropagation will give us a way of computing δ^l for every layer, and then relating those errors to the quantities of real interest,

$$\frac{\partial C}{\partial w_{jk}^l} \text{ and } \frac{\partial C}{\partial b_j^l}$$

1. An equation for the error in the output layer, δ^L

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L). \tag{BP1}$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{BP1a}$$

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \tag{2.8}$$

2. An equation for the error δ^l in terms of the error in the next layer, δ^{l+1}

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{BP2}$$

3. An equation for the rate of change of the cost with respect to any bias in the network:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{BP3}$$

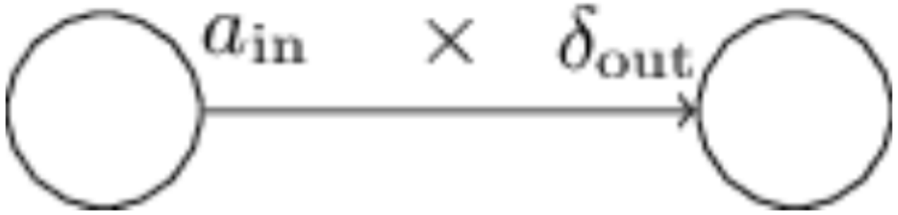
$$\frac{\partial C}{\partial b} = \delta, \tag{2.9}$$

4. An equation for the rate of change of the cost with respect to any weight:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \tag{BP4}$$

$$\frac{\partial C}{\partial w} = a_{\text{in}} \delta_{\text{out}}, \tag{2.10}$$

$$\frac{\partial C}{\partial w} =$$



Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

1. **Input** x: Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z_l)$.
3. **Output error** δ^L: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

We compute the error vectors δ backward, starting from the final layer. It may seem peculiar that we're going through the network backward. But if you think about the proof of backpropagation, the backward movement is a consequence of the fact that the cost is a function of outputs from the network.

To understand how the cost varies with earlier weights and biases we need to repeatedly apply the chain rule, working backward through the layers to obtain usable expressions.

1. Equation for error in output layer, δ^L

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L). \tag{BP1}$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{BP1a}$$

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \tag{2.8}$$

First term:

$\frac{\partial C}{\partial a_j^L}$ = how fast the cost is changing as a function of the j-th output activation.

(δ_j^L = error due to j-th neuron)

If C does not depend much on the output neuron j, then δ_j^L will be less.

Second term:

$\sigma'(z_j^L)$ = how fast the activation σ is changing with z_j^L

We compute z_j^L normally, then $\sigma'(z_j^L)$ with small overhead.

Also, if $C = 1/2 \sum_j (y_j - a_j^L)^2$, then

$$\frac{\partial C}{\partial a_j^L} = a_j^L - y_j \rightarrow \text{easily computable.}$$

1. Equation for error in output layer, δ^L

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L). \qquad \text{(BP1)}$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \qquad \text{(BP1a)}$$

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \qquad (2.8)$$

Proof: By definition:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}$$

Applying the chain rule, we can rewrite the partial derivative above in terms of partial derivatives with respect to the output activations,

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L},$$

where the sum is over all neurons k in the output layer.

The output activation a_k^L of the k-th neuron depends on the weighted input z_j^L for the j-th neuron when k=j.

So, $\frac{\partial a_k^L}{\partial z_j^L}$ vanishes when $k \neq j$.

1. Equation for error in output layer, δ^L

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L). \qquad \text{(BP1)}$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \qquad \text{(BP1a)}$$

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \qquad (2.8)$$

Proof: … hence

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L}$$

simplifies to

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}$$

Recall that $a_j^L = \sigma(z_j^L)$

So, the above equation becomes

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

2. An equation for the error δ^l in terms of the error in the next layer, δ^{l+1}

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \qquad \text{(BP2)}$$

Proof:

To prove this, we have to write error δ^l in terms of the error δ^{l+1}.

That is, we have rewrite

$\delta_j^l = \partial C / \partial z_j^l$ in terms of

$\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$

Using chain rule:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}.$$

Now, we know that

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}.$$

Differentiate

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$

Hence,

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l).$$

, the proof.

1. Equation for error in output layer, δ^L

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L). \tag{BP1}$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{BP1a}$$

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \tag{2.8}$$

2. An equation for the error δ^l in terms of the error in the next layer, δ^{l+1}

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{BP2}$$

3. An equation for the rate of change of the cost with respect to any bias in the network:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{BP3}$$

$$\frac{\partial C}{\partial b} = \delta, \tag{2.9}$$

4. An equation for the rate of change of the cost with respect to any weight:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \tag{BP4}$$

$$\frac{\partial C}{\partial w} = a_{\text{in}} \delta_{\text{out}}, \tag{2.10}$$

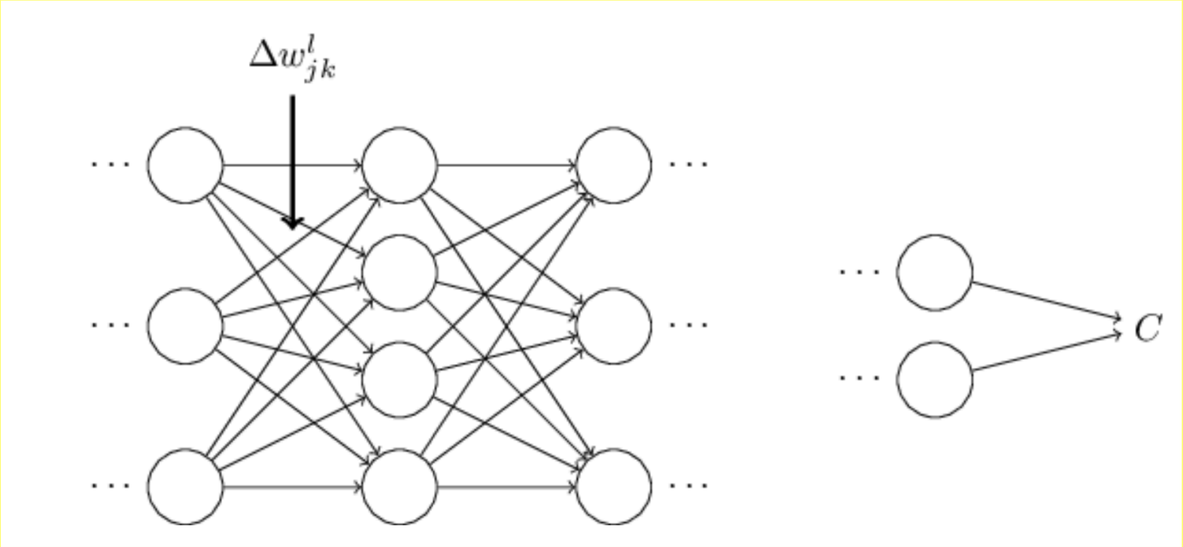
1. **Input** x: Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z_l)$.
3. **Output error** δ^L: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

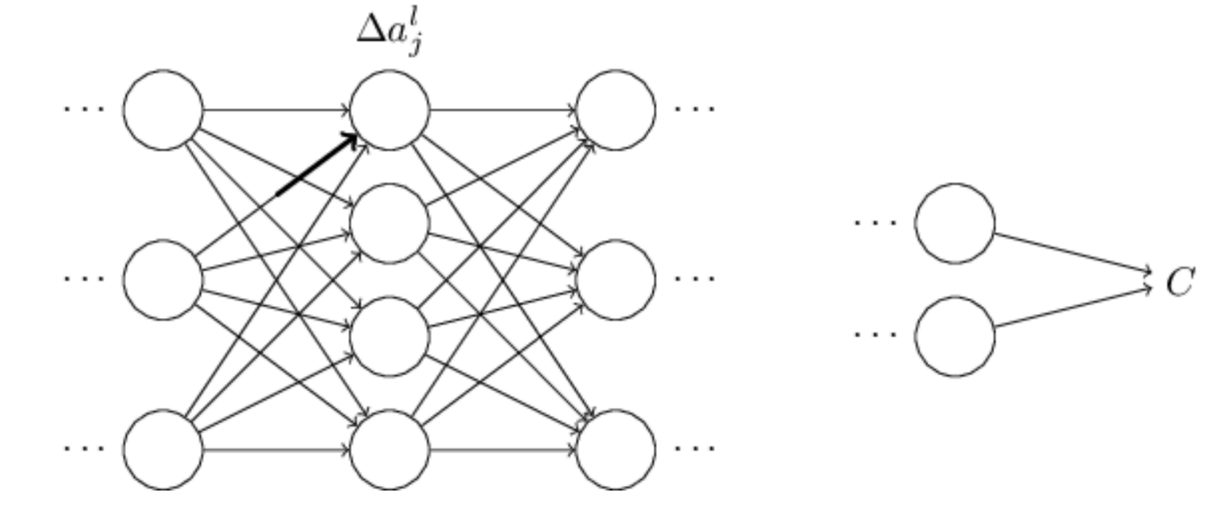
We compute the error vectors δ backward, starting from the final layer. It may seem peculiar that we're going through the network backward. But if you think about the proof of backpropagation, the backward movement is a consequence of the fact that the cost is a function of outputs from the network.

To understand how the cost varies with earlier weights and biases we need to repeatedly apply the chain rule, working backward through the layers to obtain usable expressions.

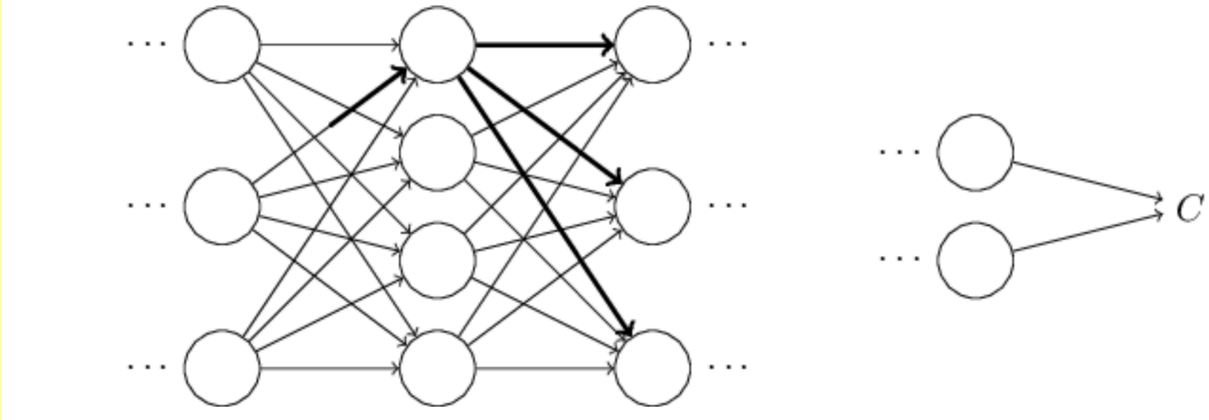
Backpropagation - a discussion

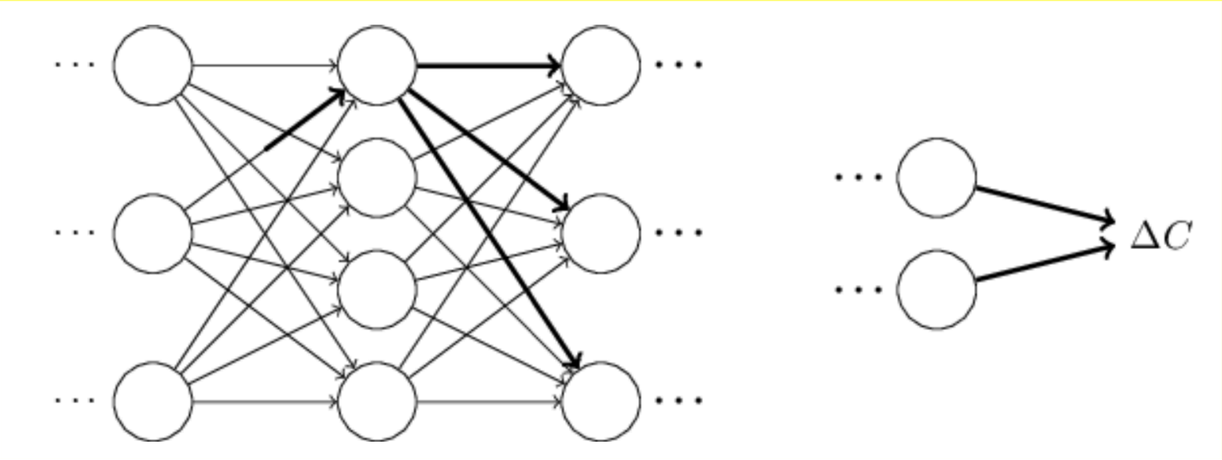
Let us make a small change $\triangle w_{jk}^{l}$ which will cause a change in output activation from that neuron,





That, in turn, will cause a change in all the activations in the next layer:





The change ΔC in the cost is related to the change $\triangle w_{jk}^{l}$ in the weight by the equation

$$\Delta C \approx \frac{\partial C}{\partial w_{jk}^{l}} \Delta w_{jk}^{l}$$

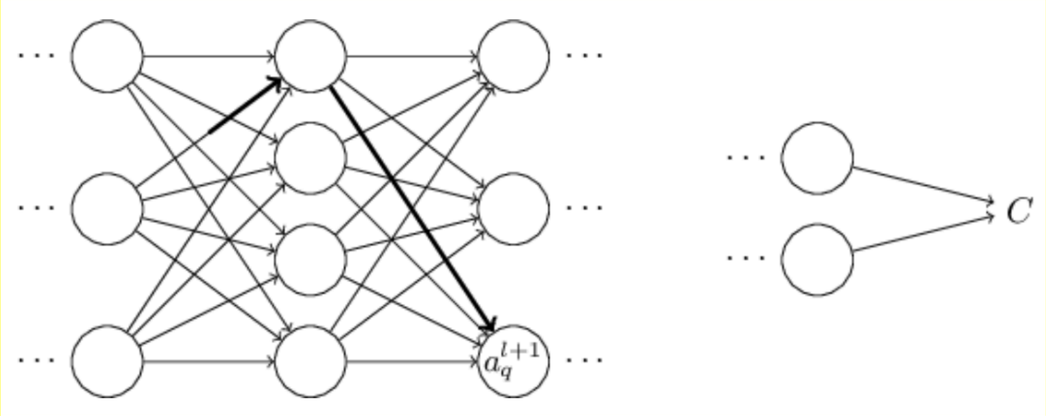
- Suggests that a possible approach to computing $\partial C / \partial w_{jk}^{l}$ is to carefully track how a small change in w_{jk}^{l} propagates to cause a small change in C. If we can do that, being careful to express everything along the way in terms of easily computable quantities, then we should be able to compute $\partial C / \partial w_{jk}^{l}$.

The change $\triangle w_{jk}^{l}$ causes a small change $\triangle a_{j}^{l}$ in the activation of the j-th neuron in the l-th layer.

$$\Delta a_{j}^{l} \approx \frac{\partial a_{j}^{l}}{\partial w_{jk}^{l}} \Delta w_{jk}^{l}$$

The change in activation, $\triangle a_j^l$, will cause changes in all the activations in the next layer, l+1 th layer.

Let us see how a single one of these activations will be affected, a_q^{l+1}:



It will cause the following change:

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \Delta a_j^l.$$

Earlier we have got:

$$\Delta a_j^l \approx \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

So,

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

The change in activation, $\triangle a_q^{l+1}$, will cause changes in all the activations in the next layer.

Imagine a path all the way through the network from w_{jk}^l to C.

Let the path go through $a_j^l, a_q^{l+1}, \cdots, a_n^{L-1}, a_m^L$

$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

This represents the change in C due to changes in the activations along this particular path through the network. Of course, there's many paths by which a change in w_{jk}^l can propagate to affect the cost, and we've been considering just a single path.

To compute the total change in C it is plausible that we should sum over all the possible paths between the weight and the final cost, i.e.,

$$\Delta C \approx \sum_{mnp\ldots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

Compare $\Delta C \approx \frac{\partial C}{\partial w^l_{jk}} \Delta w^l_{jk}$ with

$$\Delta C \approx \sum_{mnp\ldots q} \frac{\partial C}{\partial a^L_m} \frac{\partial a^L_m}{\partial a^{L-1}_n} \frac{\partial a^{L-1}_n}{\partial a^{L-2}_p} \cdots \frac{\partial a^{l+1}_q}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^l_{jk}} \Delta w^l_{jk}$$

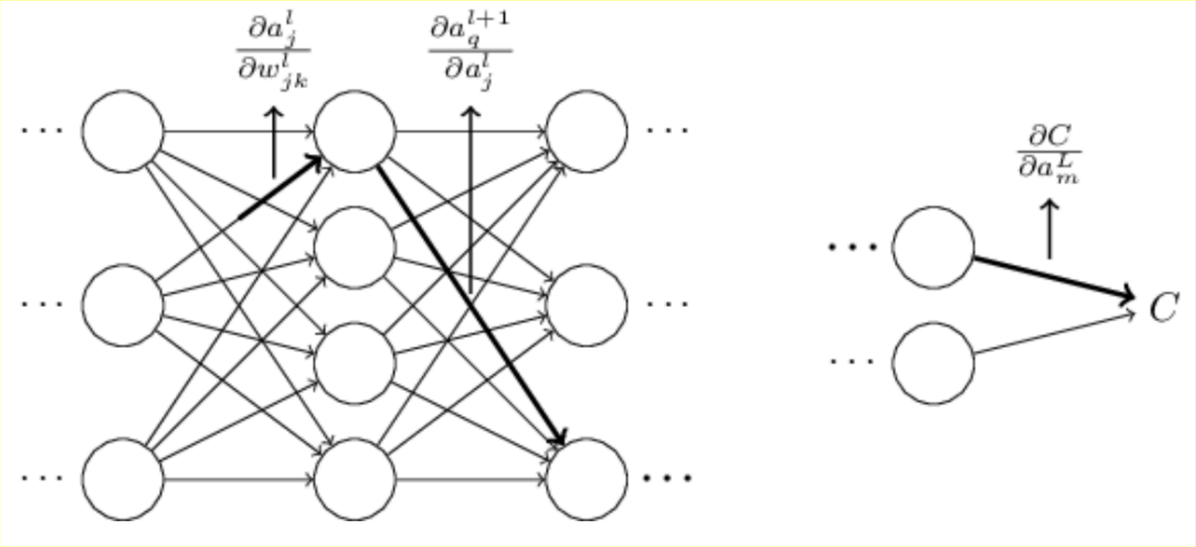
$$\longrightarrow \quad \frac{\partial C}{\partial w^l_{jk}} = \sum_{mnp\ldots q} \frac{\partial C}{\partial a^L_m} \frac{\partial a^L_m}{\partial a^{L-1}_n} \frac{\partial a^{L-1}_n}{\partial a^{L-2}_p} \cdots \frac{\partial a^{l+1}_q}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^l_{jk}}$$

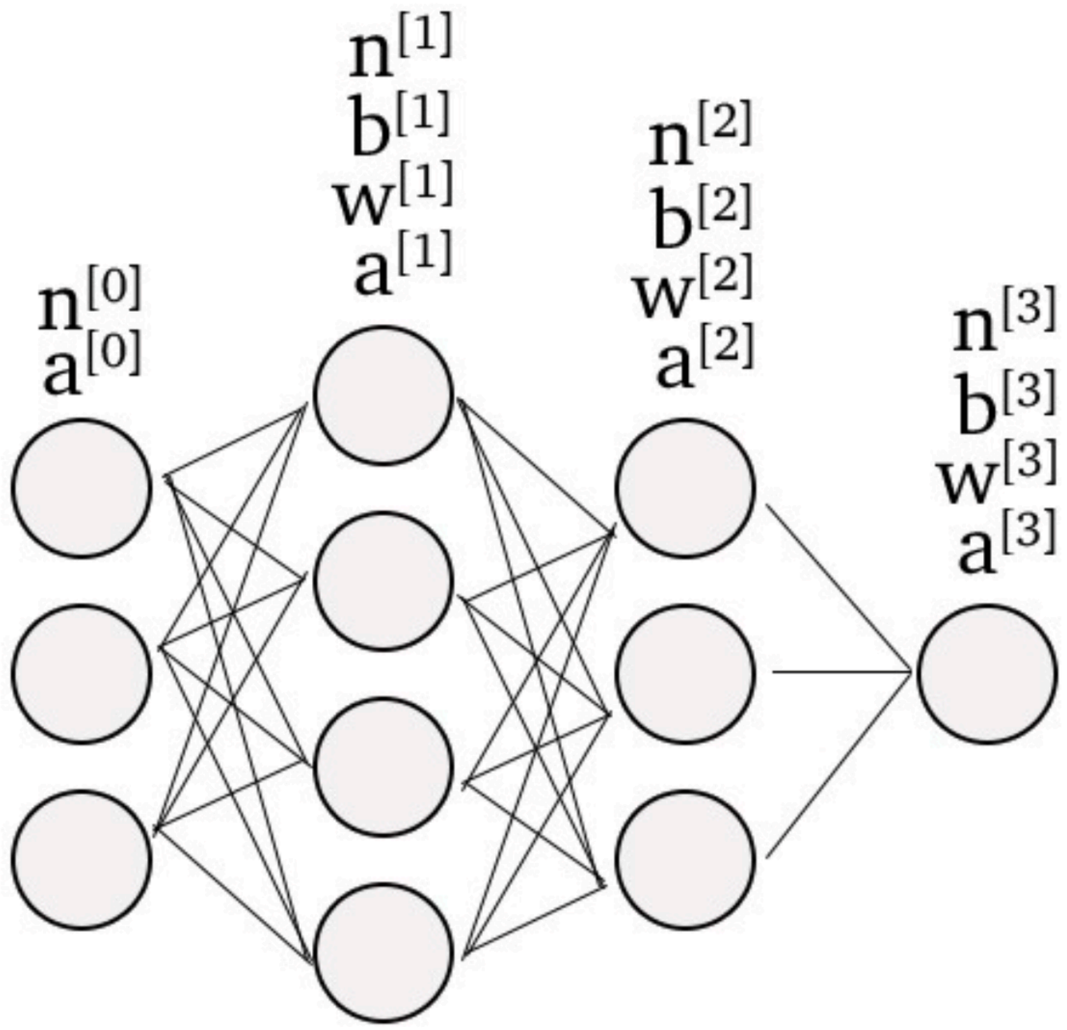
The edge from the first weight to the first neuron has a rate factor $\partial a^i_j / \partial w^l_{jk}$.

The rate factor for a path is just the product of the rate factors along the path.

And the total rate of change $\partial C / \partial w^l_{jk}$ is the sum of the rate factors of all paths from the initial weight to the final cost.

Illustrated for a single path:





Neural Network Architecture for reference

$n^{[l]}$ - The number of neurons in the layer l.

$w^{[l]}$ - The weight matrix associated with the layers l and l-1, of the size $(n^{[l]} \times n^{[l-1]})$.
$w_{jk}^{[l]}$ refers to the weight associated with the neuron j in the layer l and the neuron k in the layer l-1.

$b^{[l]}$ - The vector of biases associated with the layer l, of the size $(n^{[l]} \times 1)$.

$a^{[l]}$ - The vector of activations of the neurons in the layer l, of the size $(n^{[l]} \times 1)$.

$z^{[l]}$ - The vector of the weighted output of the neurons in the layer l, of the size $(n^{[l]} \times 1)$.

$g^{[l]}$ - The activation function applied to the output of the neurons in the layer l, $a^{[l]} = g^{[l]}(z^{[l]})$.

Notations used in the derivations

https://medium.com/binaryandmore/beginners-guide-to-deriving-and-implementing-backpropagation-e3c1a5a1e536

- Forward propagation:
- $$z^{[l]} = w^{[l]}.a^{[l-1]} + b^{[l]}$$

Equation for calculating the weighted input to the neurons in layer l

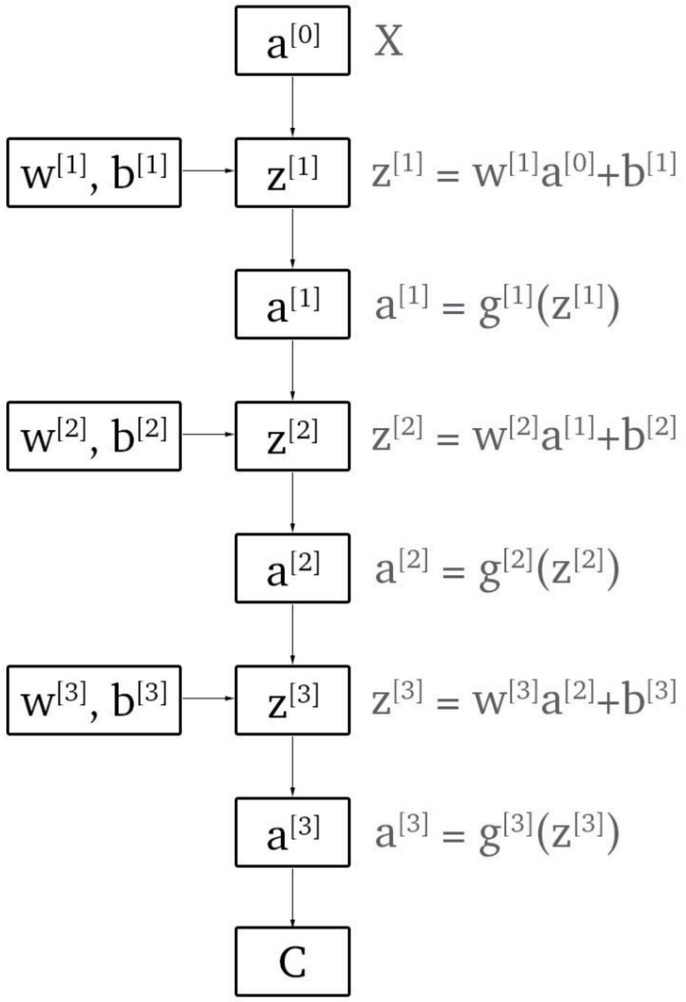
- Update rules for back propagation:

$$w^{[l]} = w^{[l]} - \alpha \frac{\partial C}{\partial w^{[l]}}$$

$$b^{[l]} = b^{[l]} - \alpha \frac{\partial C}{\partial b^{[l]}}$$

Equations of the update rules for the weights and biases of the layer l

- Forward computation graph



$C = -(y\log a^{[3]}+(1 - y)\log(1 - a^{[3]}))$

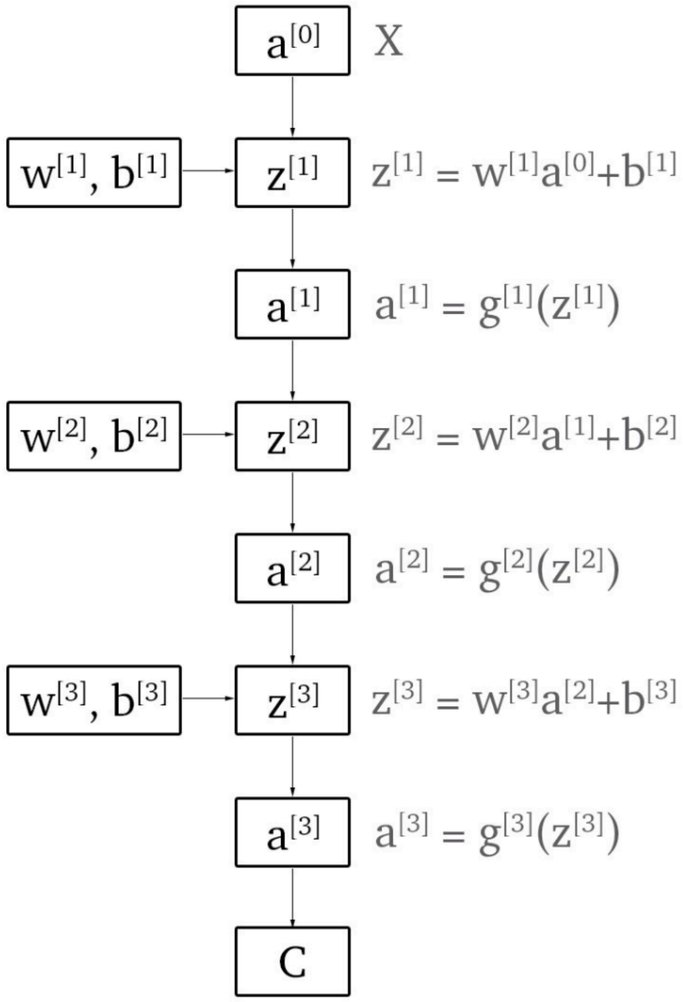
Forward Propagation Computation Graph

$$\frac{\partial C}{\partial w^{[3]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}}$$

$$\frac{\partial C}{\partial b^{[3]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial b^{[3]}}$$

Equations for calculating the partial derivative of the cost function with respect to the weights and biases of layer 3

- Forward computation graph



$$C = -(y\log a^{[3]}+(1 - y)\log(1 - a^{[3]}))$$

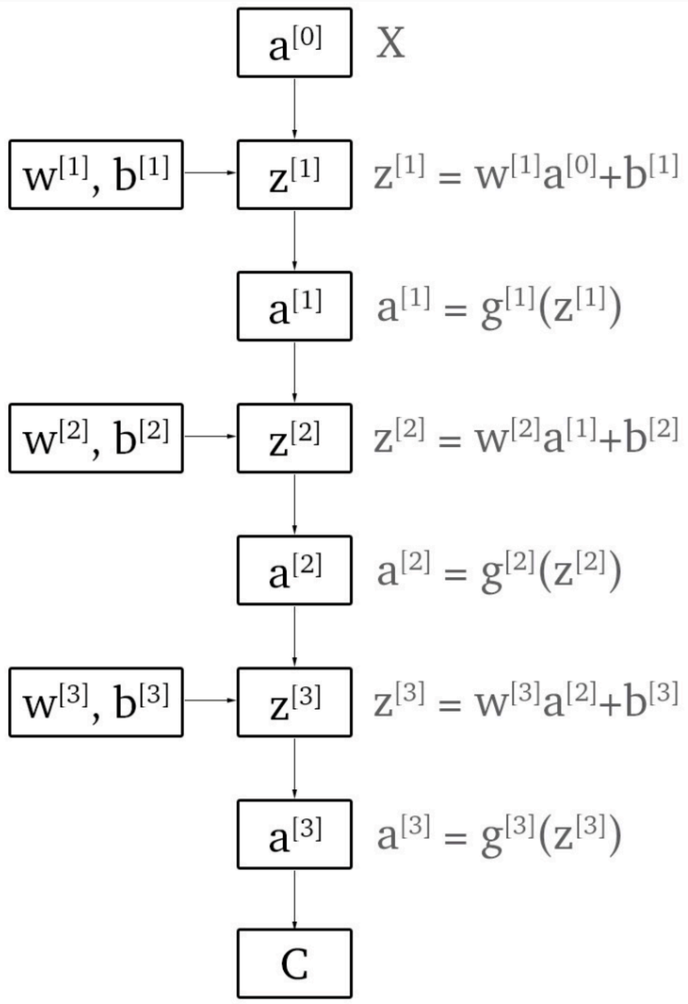
Forward Propagation Computation Graph

$$\frac{\partial C}{\partial w^{[2]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}}$$

$$\frac{\partial C}{\partial b^{[2]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial b^{[2]}}$$

Equations for calculating the partial derivative of the cost function with respect to the weights and biases of layer 2

- Forward computation graph



$C = -(y\log a^{[3]}+(1 - y)\log(1 - a^{[3]}))$

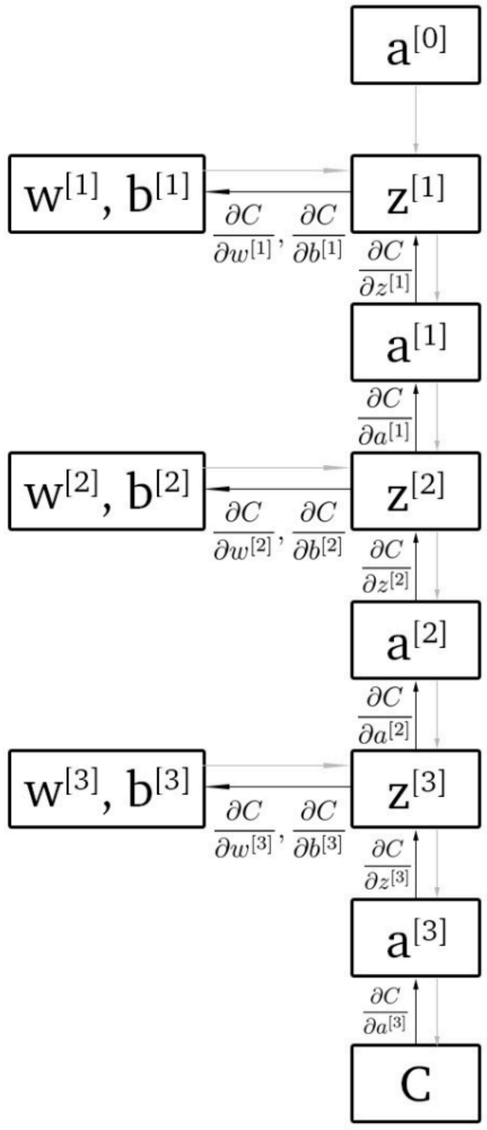
Forward Propagation Computation Graph

$$\frac{\partial C}{\partial w^{[1]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial w^{[1]}}$$

$$\frac{\partial C}{\partial b^{[1]}} = \frac{\partial C}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial b^{[1]}}$$

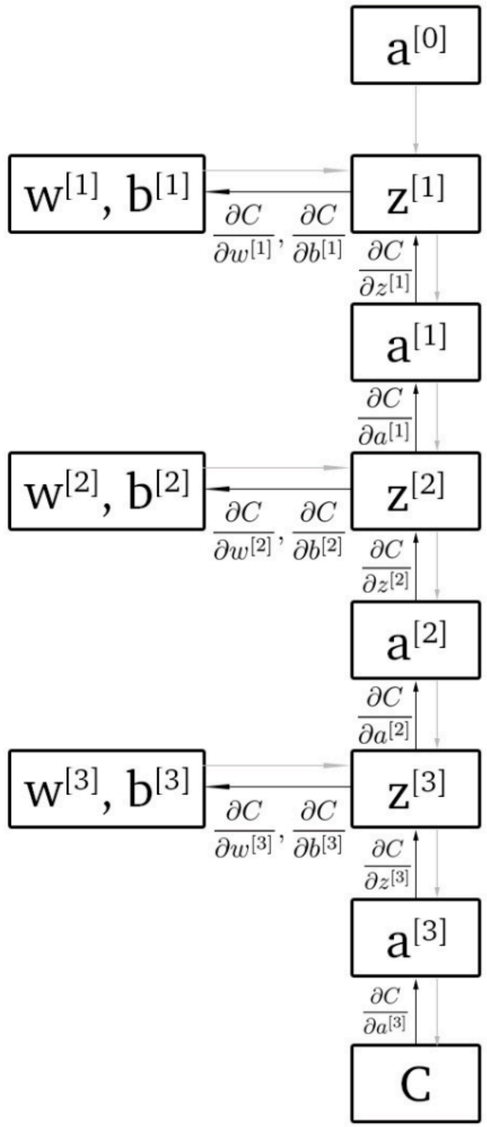
Equations for calculating the partial derivative of the cost function with respect to the weights and biases of layer 1

- Backpropagation Computation Graph



Backpropagation Computation Graph

- Backpropagation Computation Graph



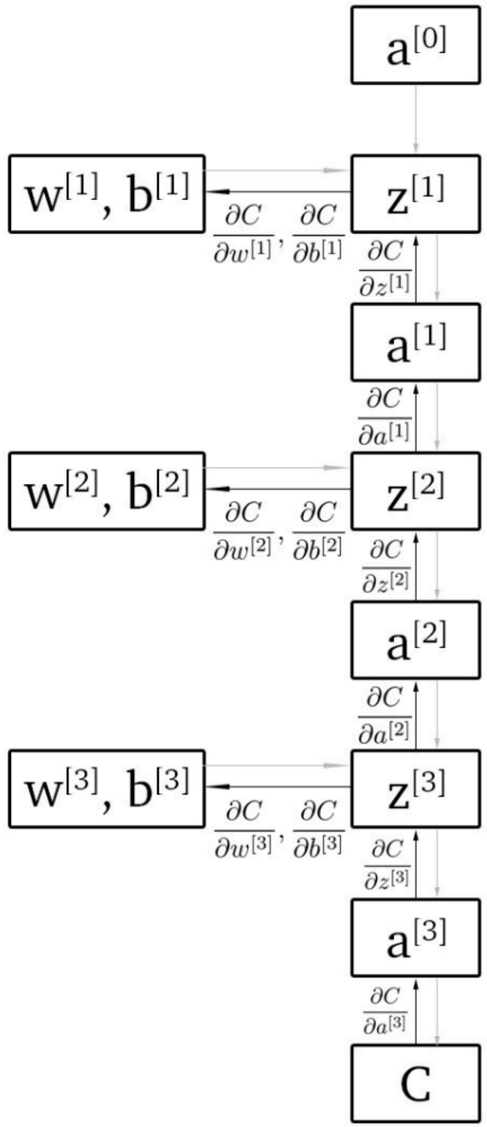
Backpropagation Computation Graph

$$\frac{\partial C}{\partial w^{[l]}} = \frac{\partial C}{\partial z^{[l]}} \cdot \frac{\partial z^{[l]}}{\partial w^{[l]}}$$

$$\frac{\partial C}{\partial b^{[l]}} = \frac{\partial C}{\partial z^{[l]}} \cdot \frac{\partial z^{[l]}}{\partial b^{[l]}}$$

Equations for calculating the partial derivative of the cost function with respect to the weights and biases of any layer l

- Backpropagation Computation Graph



Backpropagation Computation Graph

So for calculating the partial derivatives of C with respect to w[l], b[l], we need to calculate

$$\frac{\partial C}{\partial z^{[L]}}, \frac{\partial C}{\partial z^{[l]}}, \frac{\partial z^{[l]}}{\partial w^{[l]}}, \frac{\partial z^{[l]}}{\partial b^{[l]}}$$

Four fundamental partial derivatives required to perform backpropagation

where L is the last layer.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid function

- The partial derivative of C with respect to z[L] can be calculated as follows:

$$\frac{\partial C}{\partial z^{[L]}} = \frac{\partial C}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} = \frac{\partial C}{\partial a^{[L]}} . * \sigma'(z[L])$$

$$\boxed{\frac{\partial C}{\partial z^{[L]}} = \frac{\partial C}{\partial a^{[L]}} . * \sigma'(z[L])}$$

where,

$$\frac{\partial C}{\partial a^{[L]}} = \frac{\partial(-(yloga^{[L]} + (1-y)log(1-a^{[L]})))}{\partial a^{[L]}}$$

using

$$\frac{\partial yloga^{[L]}}{\partial a^{[L]}} = \frac{y}{a^{[L]}} \quad \text{and} \quad \frac{\partial(1-y)log(1-a^{[L]})}{\partial a^{[L]}} = \frac{1-y}{1-a^{[L]}}(-1)$$

we get:

$$\frac{\partial(-(yloga^{[L]} + (1-y)log(1-a^{[L]})))}{\partial a^{[L]}} = -(\frac{y}{a^{[L]}} - \frac{1-y}{1-a^{[L]}})$$

$$\therefore \frac{\partial C}{\partial a^{[L]}} = -(\frac{y}{a^{[L]}} - \frac{1-y}{1-a^{[L]}}) \quad \ldots 1$$

Derivation of the partial derivative of the cost function with respect to the activation of the last layer

Derivative of the activation function:

$$\sigma(z^{[L]}) = \frac{1}{1+e^{-z^{[L]}}}$$

$$\sigma'(z^{[L]}) = \frac{d}{dz^{[L]}}\left(\frac{1}{1+e^{-z^{[L]}}}\right)$$

$$= -\frac{1}{(1+e^{-z^{[L]}})^2}\cdot\frac{d}{dz^{[L]}}(1+e^{-z^{[L]}})$$

$$= -\frac{e^{-z^{[L]}}}{(1+e^{-z^{[L]}})^2}\cdot\frac{d}{dz^{[L]}}(-z^{[L]})$$

$$= \frac{e^{-z^{[L]}}}{(1+e^{-z^{[L]}})^2}$$

$$= \frac{e^{-z^{[L]}}+1-1}{(1+e^{-z^{[L]}})^2}$$

$$= \frac{1}{1+e^{-z^{[L]}}}.\left(\frac{e^{-z^{[L]}}+1-1}{1+e^{-z^{[L]}}}\right)$$

$$= \frac{1}{1+e^{-z^{[L]}}}.\left(\frac{1+e^{-z^{[L]}}}{1+e^{-z^{[L]}}} - \frac{1}{1+e^{-z^{[L]}}}\right)$$

$$= \frac{1}{1+e^{-z^{[L]}}}.\left(1 - \frac{1}{1+e^{-z^{[L]}}}\right)$$

$$= \sigma(z^{[L]}).(1-\sigma(z^{[L]}))$$

$$\sigma'(z^{[L]}) = \sigma(z^{[L]}).(1-\sigma(z^{[L]}))$$

$$\therefore\ \sigma'(z^{[L]}) = a^{[L]}.(1-a^{[L]}) \quad \ldots 2$$

Derivation of the derivative of the sigmoid activation function

$$\frac{\partial C}{\partial z^{[L]}} = -(\frac{y}{a^{[L]}} - \frac{1-y}{1-a^{[L]}}).a^{[L]}.(1-a^{[L]})$$

$$= -(\frac{y(1-a^{[L]}) - (1-y)a^{[L]}}{a^{[L]}.(1-a^{[L]})}).a^{[L]}.(1-a^{[L]})$$

$$= -(y(1-a^{[L]}) - (1-y)a^{[L]})$$

$$= -(y - ya^{[L]} - (a^{[L]} - ya^{[L]}))$$

$$= -(y - ya^{[L]} - a^{[L]} + ya^{[L]})$$

$$= -(y - a^{[L]})$$

$$= a^{[L]} - y$$

$$\therefore \boxed{\frac{\partial C}{\partial z^{[L]}} = a^{[L]} - y}$$

Derivation of the partial derivative of the cost function with respect to z[L]

Partial derivative of C with respect to z[l]

We want the partial derivative of C with respect to z[l] in terms of the partial derivative of C with respect to the layer l+1, so that once we have z[L], we can calculate z[L-1], z[L-2].. and so on.

We can express C as a function of z[l + 1] for any given l.

$$\frac{\partial C}{\partial z^{[l]}} = \frac{\partial C}{\partial z^{[l+1]}} \cdot \frac{\partial z^{[l+1]}}{\partial a^{[l]}} \cdot \frac{\partial a^{[l]}}{\partial z^{[l]}}$$

Equation for the partial derivative of the cost function with respect to z[l] in terms of the partial derivative of the cost function with respect to z[l + 1]

To calculate ∂z[*l*+1]/∂a[*l*] and ∂a[*l*]/∂z[*l*].

$$z^{[l+1]} = w^{[l+1]}.a^{[l]} + b^{[l+1]}$$

$$\frac{\partial z^{[l+1]}}{\partial a^{[l]}} = \frac{\partial}{\partial a^{[l]}}(w^{[l+1]}.a^{[l]} + b^{[l+1]})$$

$$= w^{[l+1]}$$

$$\therefore \frac{\partial z^{[l+1]}}{\partial a^{[l]}} = w^{[l+1]}$$

$$\frac{\partial a^{[l]}}{\partial z^{[l]}} = g^{[l]\prime}(z^{[l]})$$

$$\frac{\partial a^{[l]}}{\partial z^{[l]}} = \sigma'(z^{[l]})$$

Putting together,

$$\boxed{\frac{\partial C}{\partial z^{[l]}} = (w^{[l+1]^T} \cdot \frac{\partial C}{\partial z^{[l+1]}}).*\sigma'(z^{[l]})}$$

. = matrix multiplication and * = element-wise multiplication

Partial derivative of z[l] w.r.t. w[l],

$\partial z^{[l]} / \partial w^{[l]}$

$$z^{[l]} = w^{[l]}.a^{[l-1]} + b^{[l]}$$

$$\frac{\partial z^{[l]}}{\partial w^{[l]}} = \frac{\partial}{\partial w^{[l]}}(w^{[l]}.a^{[l-1]} + b^{[l]})$$

$$= a^{[l-1]}$$

$$\therefore \boxed{\frac{\partial z^{[l]}}{\partial w^{[l]}} = a^{[l-1]}}$$

Derivation of partial derivative of z[l] with respect to w[l]

Partial derivative of C with respect to w[l]

$$\boxed{\frac{\partial C}{\partial w^{[l]}} = \frac{\partial C}{\partial z^{[l]}}.a^{[l-1]^T}}$$

Equation for the partial derivative of the cost function with respect to w[l]

Partial derivative of z[*l*] with respect to b[*l*]

$$z^{[l]} = w^{[l]}.a^{[l-1]} + b^{[l]}$$

$$\frac{\partial z^{[l]}}{\partial b^{[l]}} = \frac{\partial}{\partial b^{[l]}}(w^{[l]}.a^{[l-1]} + b^{[l]})$$

$$\therefore \boxed{\frac{\partial z^{[l]}}{\partial b^{[l]}} = 1}$$

Derivation of the partial derivative of z[l] with respect to b[l]

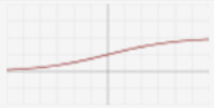
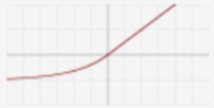
Partial derivative of C with respect to b[l]

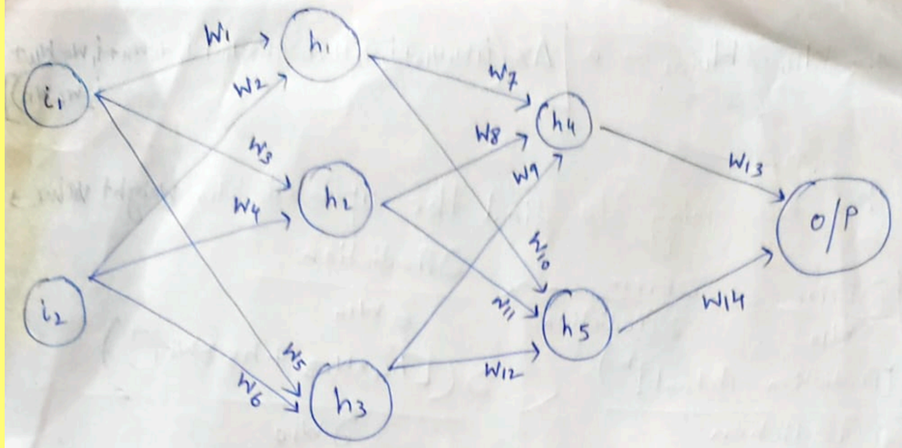
$$\boxed{\frac{\partial C}{\partial b^{[l]}} = \frac{\partial C}{\partial z^{[l]}}}$$

Equation for the partial derivative of the cost function with respect to b[l]

Note that these derivations consider input X as only one training example, for a batch of m examples, we have to average the derivatives over the m examples.

Some activation functions

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for} \quad x \neq 0 \\ ? & \text{for} \quad x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$



$$\text{Error} = \frac{1}{2}\left[\text{Prediction} - \text{Actual}\right]^2$$

$$W_{x\,new} = W_{x\,old} - a\left[\frac{\partial \text{Error}}{\partial W_{x\,old}}\right]$$

{ a = learning rate
$W_{x\,old}$ = old weight value
$W_{x\,new}$ = New weight value }

Now we are going to find the updated W_{14} weight value →

So,

$$\frac{\partial \text{Error}}{\partial W_{14}} = \frac{\partial \text{Error}}{\partial \text{Prediction}} \times \frac{\partial \text{Prediction}}{\partial W_{14}}$$

$$= \frac{\partial\left(\frac{1}{2}\left[\text{Prediction} - \text{Actual}\right]^2\right)}{\partial \text{Prediction}} \times \frac{\partial\left(\left[h_4 \times W_{13}\right] + \left[h_5 \times W_{14}\right]\right)}{\partial W_{14}}$$

$$= \frac{1}{2} \times 2\left(\text{Prediction} - \text{Actual}\right) \times \frac{\partial\left[\left(\text{Prediction} - \text{Actual}\right)\right]}{\partial \text{Prediction}} \times \frac{\partial\left(h_5 \times W_{14}\right)}{\partial W_{14}}$$

$$= \left(\text{Prediction} - \text{Actual}\right) \times h_5$$

$$= \Delta \times \left[\left(h_1 \times W_{10} + h_2 \times W_{11} + h_3 \times W_{12}\right)\right] \quad \left[\because \left(\text{Prediction} - \text{Actual}\right) = \Delta\right]$$

$$= \Delta \times \left[\left\{(i_1 W_1 + i_2 W_2) \times W_{10}\right\} + \left\{(i_1 W_3 + i_2 W_4) \times W_{11}\right\} + \left\{(i_1 W_5 + i_2 W_6) \times W_{12}\right\}\right]$$

$$= \Delta \times \left[i_1 W_1 W_{10} + i_2 W_2 W_{10} + i_1 W_3 W_{11} + i_2 W_4 W_{11} + i_1 W_5 W_{12} + i_2 W_6 W_{12}\right]$$

Now, We are going to find the updated W_{10} weight value →

So,

$$\frac{\partial \text{Error}}{\partial W_{10}} = \frac{\partial \text{Error}}{\partial \text{Prediction}} \times \frac{\partial \text{Prediction}}{\partial W_{10}}$$

$$= \frac{\partial\left(\frac{1}{2}\left[\text{Prediction} - \text{Actual}\right]^2\right)}{\partial \text{Prediction}} \times \frac{\partial\left(\left[h_4 \times W_{13} + h_5 \times W_{14}\right]\right)}{\partial W_{10}}$$

$$= \frac{1}{2} \times 2\left(\text{Prediction} - \text{Actual}\right) \times \frac{\partial\left[\left(\text{Prediction} - \text{Actual}\right)\right]}{\partial \text{Prediction}} \times \frac{\partial\left(\left[\{h_1 \times W_7 + h_2 \times W_8 + h_3 W_9\} \times W_{13} + \{h_1 W_{10} + h_2 W_{11} + h_3 W_{12}\} \times W_{14}\right]\right)}{\partial W_{10}}$$

$$= \left(\text{Prediction} - \text{Actual}\right) \times \frac{\partial\left(h_1 W_{10} W_{14}\right)}{\partial W_{10}}$$

$$= \Delta \times h_1 W_{14} \quad \left[\because \left(\text{Prediction} - \text{Actual}\right) = \Delta\right]$$

$$= \Delta \times \left[\left(i_1 W_1 + i_2 W_2\right) W_{14}\right]$$

$$= \Delta \times \left[i_1 W_1 W_{14} + i_2 W_2 W_{14}\right]$$

So,

$$\text{New } W_{10} = W_{10\,old} - a\left[\Delta \times \left(i_1 W_1 W_{14} + i_2 W_2 W_{14}\right)\right]$$

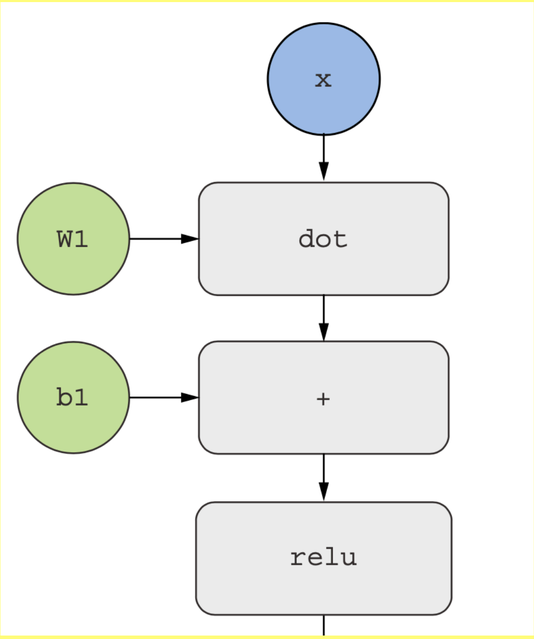
Chaining derivatives: The Backpropagation algorithm

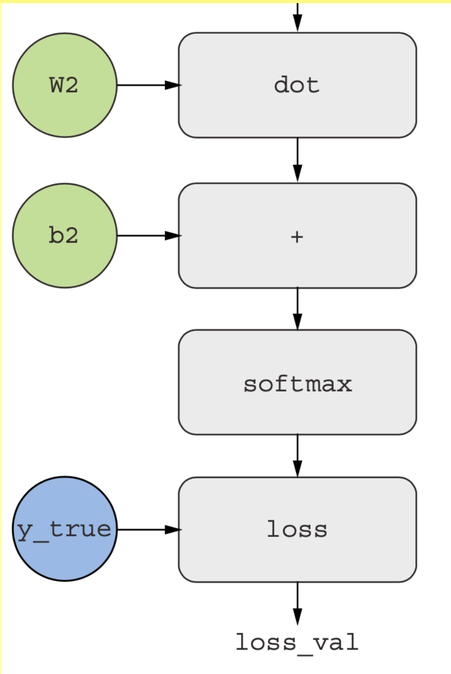
Backpropagation is a way to use the derivatives of simple operations (such as addition, relu, or tensor product) to easily compute the gradient of arbitrarily complex combi- nations of these atomic operations.

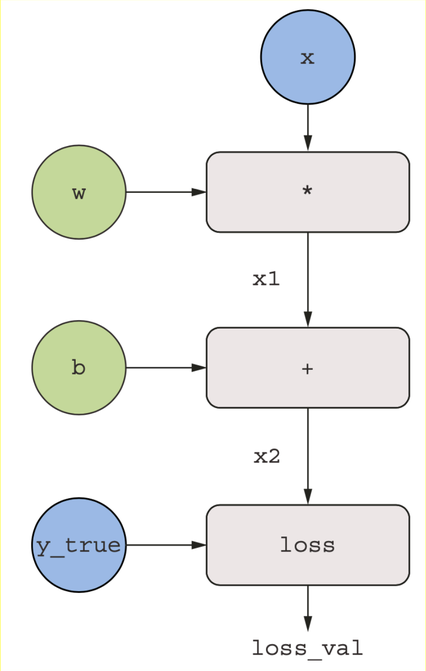
Crucially, a neural network consists of many tensor operations chained together, each of which has a simple, known derivative.

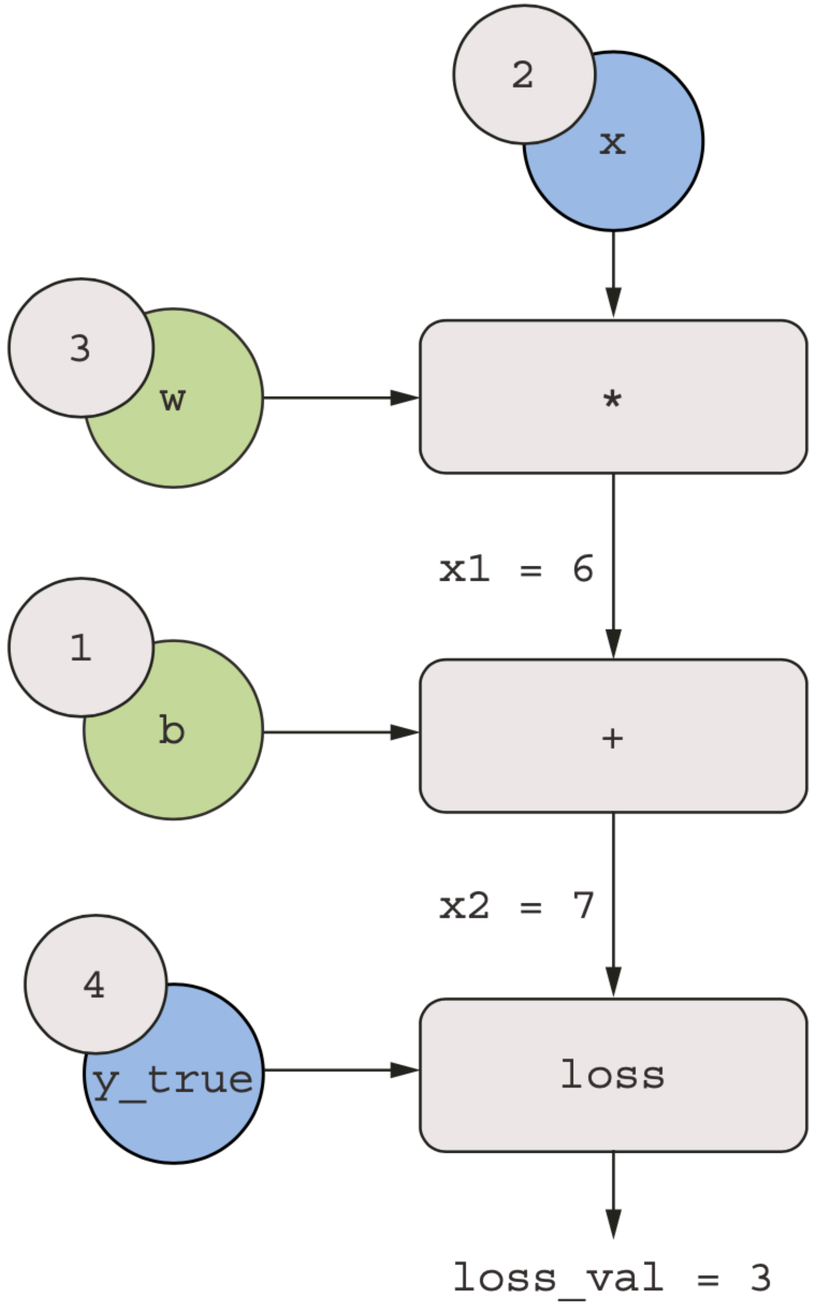
AUTOMATIC DIFFERENTIATION WITH COMPUTATION GRAPHS

The Computation Graph representation of our two-layer model








2
x
3
w
*
x1 = 6
1
b
+
x2 = 7
4
y_true
loss
loss_val = 3

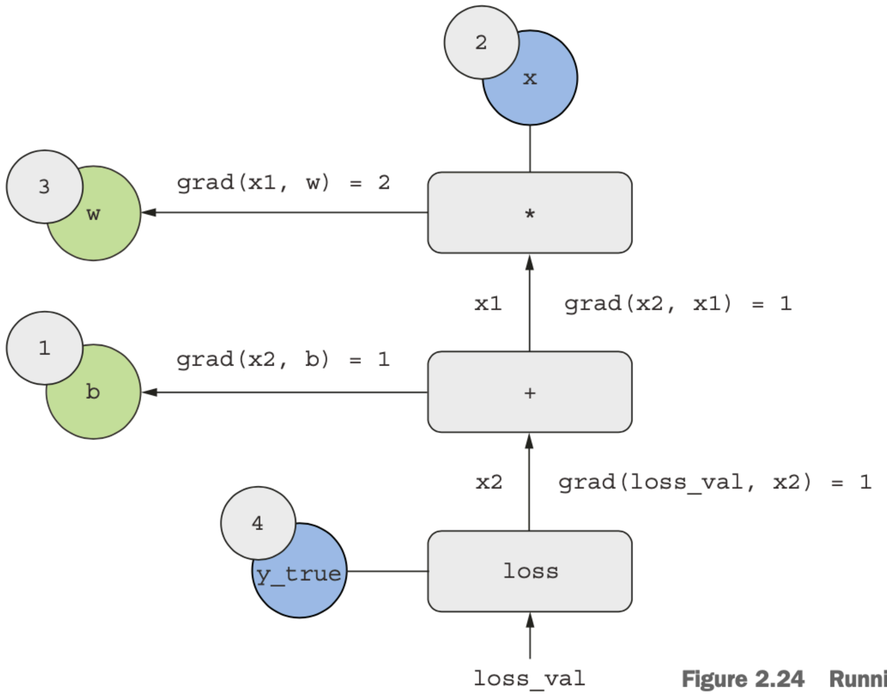
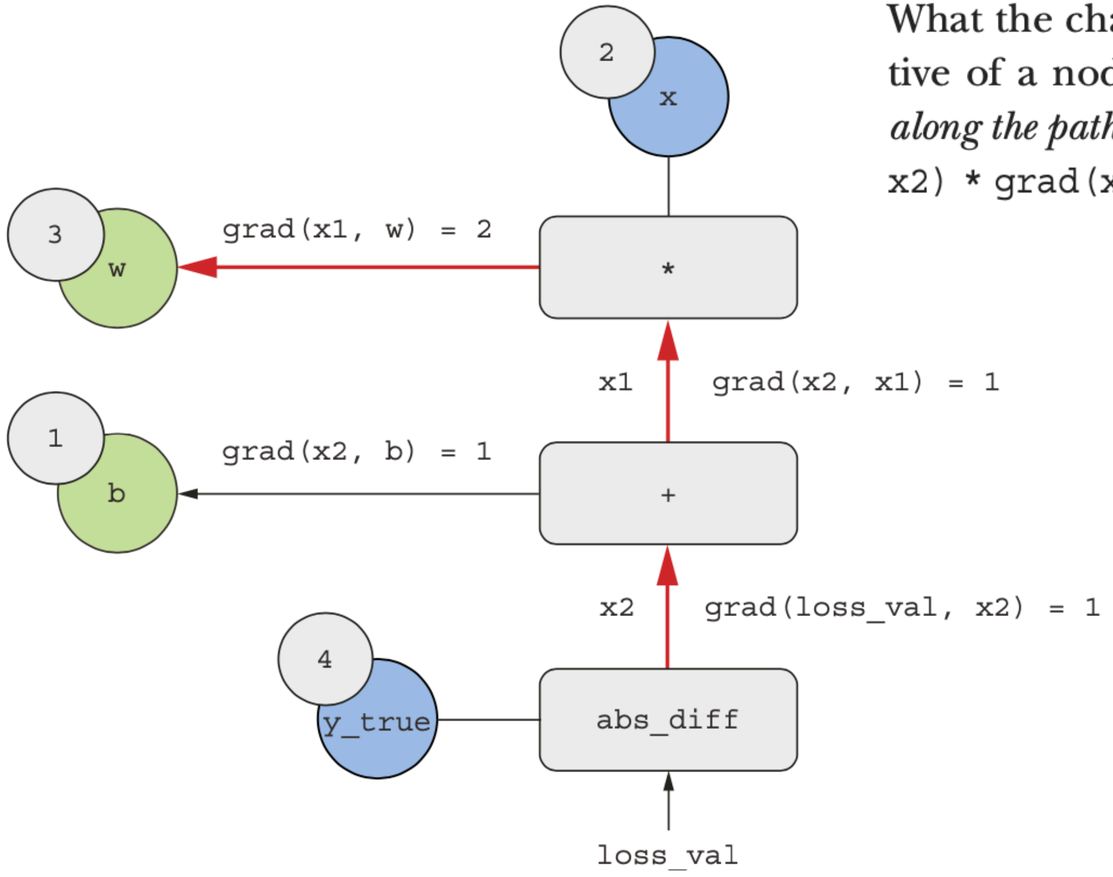


Figure 2.24 Running a backward pass

- `grad(loss_val, x2) = 1`, because as `x2` varies by an amount epsilon, `loss_val = abs(4 - x2)` varies by the same amount.
- `grad(x2, x1) = 1`, because as `x1` varies by an amount epsilon, `x2 = x1 + b = x1 + 1` varies by the same amount.
- `grad(x2, b) = 1`, because as `b` varies by an amount epsilon, `x2 = x1 + b = 6 + b` varies by the same amount.
- `grad(x1, w) = 2`, because as `w` varies by an amount epsilon, `x1 = x * w = 2 * w` varies by `2 * epsilon`.



What the chain rule says about this backward graph is that you can obtain the derivative of a node with respect to another node by *multiplying the derivatives for each edge along the path linking the two nodes.* For instance, `grad(loss_val, w) = grad(loss_val, x2) * grad(x2, x1) * grad(x1, w)` (see figure 2.25).

By applying the chain rule to our graph, we obtain what we were looking for:

- `grad(loss_val, w) = 1 * 1 * 2 = 2`
- `grad(loss_val, b) = 1 * 1 = 1`

Warm up: a fast matrix-based approach to computing the output from a neural network.

At the heart of backpropagation is an expression for the partial derivative

$$\partial C / \partial w$$

of the cost function C with respect to any weight w (or bias b) in the network.

The expression tells us how quickly the cost changes when we change the weights and biases.

It actually gives us detailed insights into how changing the weights and biases changes the overall behaviour of the network.

Gradient Descent and the Delta Rule

Use gradient descent to search the hypothesis space of possible weight vectors to find the weights that best fit the training example.

Gradient descent provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units.

It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

In order to derive a weight learning rule: Let us specify the training error of a hypothesis (weight vector).

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

D-Training set, t_d - target output, o_d - output

VISUALIZING THE HYPOTHESIS SPACE

ides the basis for
N algorithm,
ks with many

It is also impo
can serve as th
that must sear
containing ma
continuously p

ight learning rule:
ing error of a
tor).

et output, o_d -

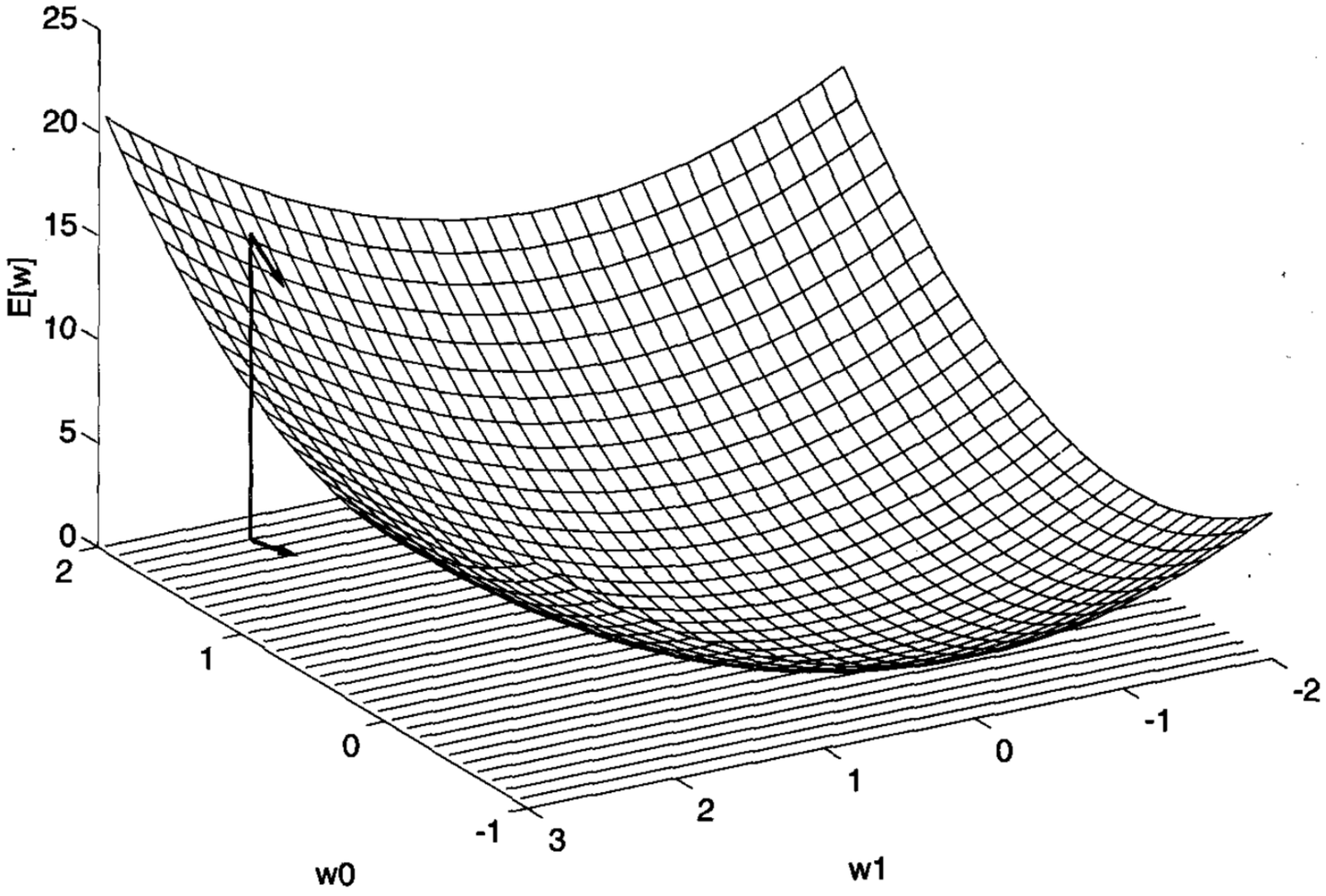


FIGURE 4.4
Error of different hypotheses. For a linear unit with two weights, the hypothesis space H is the w_0, w_1 plane. The vertical axis indicates the error of the corresponding weight vector hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.

Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps.

At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface. This process continues until the global minimum error is reached.

Gradient of E w.r.t. $\vec{w}$ is a vector:

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right]$$

The training rule for gradient descent:

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$ where $$\Delta\vec{w} = -\eta\nabla E(\vec{w})$$

In component form:

$$w_i \leftarrow w_i + \Delta w_i$$ where $$\Delta w_i = -\eta\frac{\partial E}{\partial w_i}$$

How to calculate the gradient in each step?

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$$

Differentiate the eqn.

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_{d\in D}(t_d - o_d)(-x_{id})
\end{aligned}$$

Gradient of E w.r.t. $\vec{w}$ is a vector:

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d)$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id})$$

where x_{id} denotes the single input component x_i for training example d.

From eqn. $$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Weight update of gradient descent:

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id}$$

Summary gradient descent:

- Pick an initial random weight vector
- Apply the linear unit to all training examples
- then compute Δw_i for each weight by

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id}$$

- Update each weight w_i by adding Δw_i

Then repeat this process.

GRADIENT-DESCENT(*training_examples*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \tag{T4.1}$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i \tag{T4.2}$$

TABLE 4.1

GRADIENT DESCENT algorithm for training a linear unit. To implement the stochastic approximation to gradient descent, Equation (T4.2) is deleted, and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

Practical difficulties in gradient descent:

- converging to a local minimum can sometimes be quite slow and

- if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

Incremental gradient descent or stochastic gradient descent:

Gradient descent training computes weight updates after summing over all the training examples in D

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id}$$

The idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for each individual example.

Modified training rule:

$$\Delta w_i = \eta (t - o)\, x_i$$

Modified training rule:

$$\Delta w_i = \eta(t - o)\ x_i$$

where t, o, and x_i are the target value, unit output, and ith input for the training example in question.

Key differences betn. standard gradient descent and stochastic gradient descent:

1. In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.

2. Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.

3. In cases where there are multiple local minima with respect to $E(\vec{w})$ stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$ to guide its search.

The Backpropagation Algorithm

It learns the weights for a multilayer network.

It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs

Lets consider a network with multiple output units:

E redefined:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

t_{kd} and o_{kd} are the target and output values of the k-th output input.

BACKPROPAGATION($training_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training_examples$, Do

    Propagate the input forward through the network:

    1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

    Propagate the errors backward through the network:

    2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

    3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

    4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

    where

$$\Delta w_{ji} = \eta\, \delta_j\, x_{ji} \tag{T4.5}$$

```
while(training examples not exhausted)
    For (each training example)
        Apply the network
        Compute the output
        Calculate error
        Compute the gradient w.r.t. error
        Update all weights in the network
```

This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.