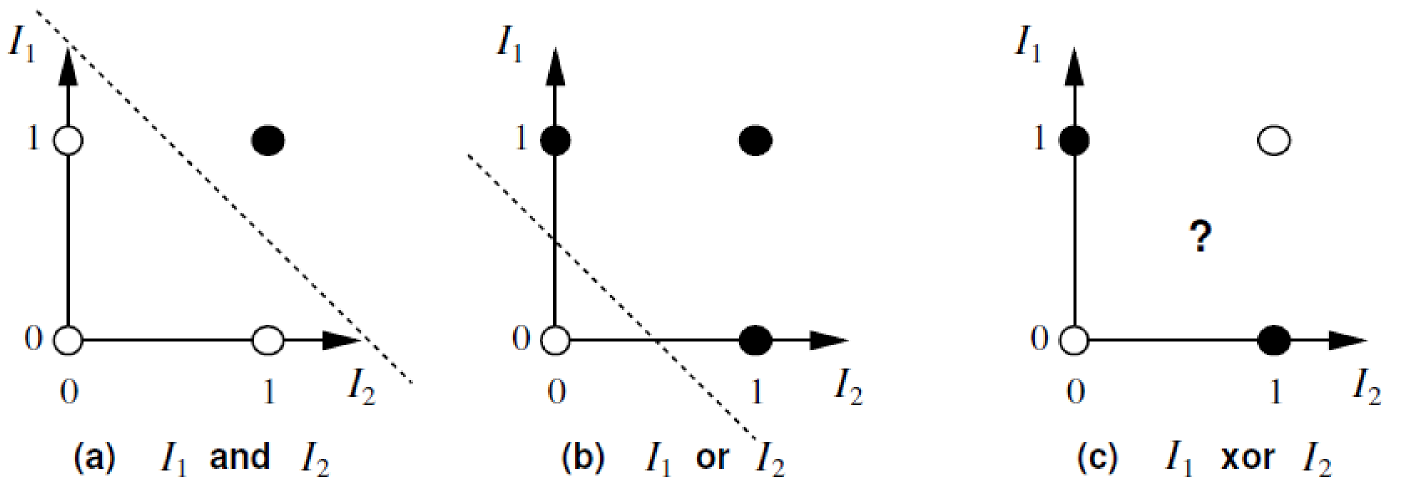
1b: Multi Layer Networks and Backpropagation

Multi Layer Perceptrons

Limitations of Perceptrons

The main problem with Perceptrons is that many useful functions are not linearly separable.

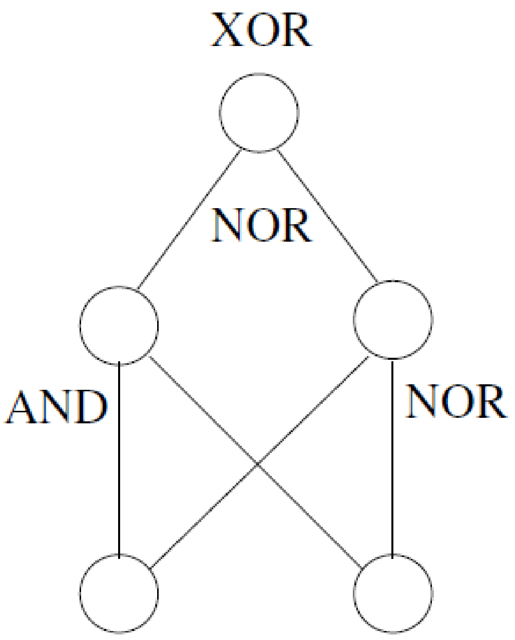


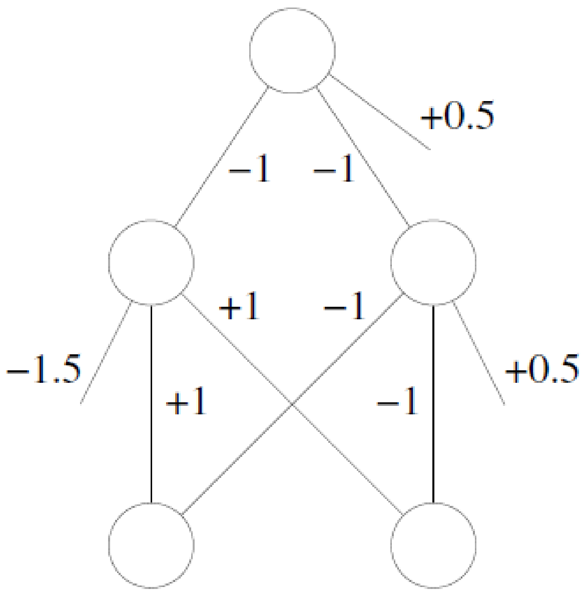
The simplest example of a logical function that is not linearly separable is the Exclusive OR or XOR function. Some languages have distinct words for inclusive and exclusive OR — for example, Latin has "aut" and "vel". In other languages, like English, this distinction needs to be inferred from the context. Consider these two sentences:

1. "All of his movies are either too long or too silly." [They could be both too long and too silly]
2. "Either you give me the money, or I will punch you in the face." [We understand from the context that one of these things will happen, but not both]

Multi Layer Perceptrons

One solution to this problem is to rewrite XOR in terms of linearly separable functions like AND, OR, NOR and then arrange several Perceptrons into a network which combines them in the same way. For example,
x_1 XOR x_2 can be written as: (x_1 AND x_2) NOR (x_1 NOR x_2). We can therefore arrange three perceptrons in the following way to compute XOR:





In the following exercise, you will be showing how this method can be used to construct a two-layer neural network which computes any given logical function.

Gradient Descent

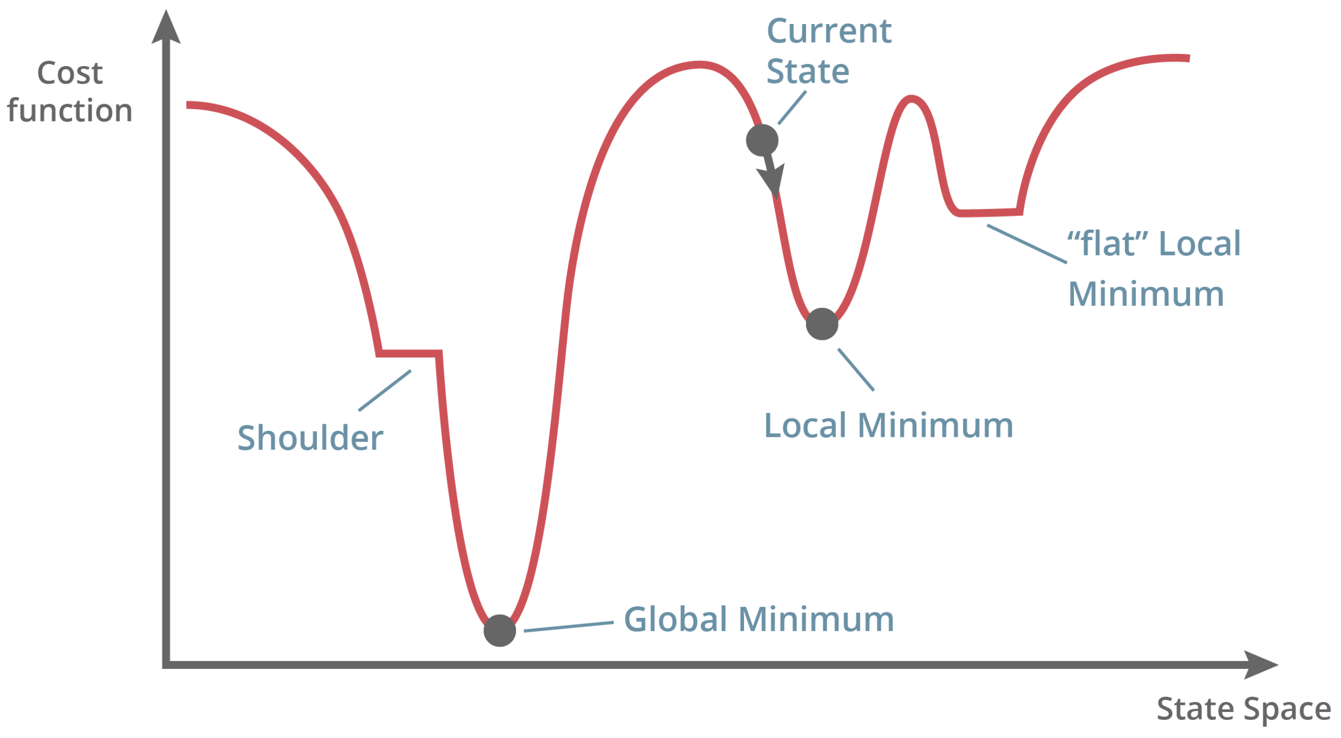
We saw previously how a Multi Layer Perceptron can be built to implement any logical function. However, normally we need to deal with raw data rather than an explicit logical expression. What we really want is a method, analogous to the perceptron learning algorithm, which can learn the weights of a neural network, based on a set of training items.

As early as the 1960's, engineers understood how to use Gradient Descent to optimize over a family of functions that are continuous and differentiable. The basic idea is this:

We define an error function or **loss** function E to be (half) the sum over all input items of the square of the difference between the actual output and target output:

$$\mathrm{E} = \frac{1}{2}\sum_i (z_i - t_i)^2$$

If we think of E as height, this defines an error landscape on the weight space. The aim is to find a set of weights for which E is very low.



If the functions involved are smooth, we can use multi-variable calculus to adjust the weights in such a way as to take us in the steepest downhill direction.

$$w \leftarrow w - \eta\frac{\partial E}{\partial w}$$

Parameter η is called the **learning rate**.

Gradient Descent for Neural Networks

Although Gradient Descent was already a well-established technique in the 1960s, somehow no-one thought to apply it to Neural Networks until many years later. One reason for this was politics. Marvin Minsky and Seymour Papert criticised neural networks in their 1969 book "Perceptrons" and lobbied the US Government to redirect research funding away from neural networks and into symbolic methods such as expert systems.

The other reason was a technical obstacle. If we use the step function as our transfer function, the landscape will not be smooth but will instead consist almost entirely of flat local regions and "shoulders", with occasional discontinuous jumps.

For the perceptron this didn't matter, because it only had one layer, but for networks with two or more layers, it becomes a big problem.

In order for gradient descent to be applied successfully, neural networks would need to be redesigned so that the function from input to output becomes smooth and differentiable. This was achieved by Paul Werbos in 1975 and, more famously, in (Rumelhart, 1986).

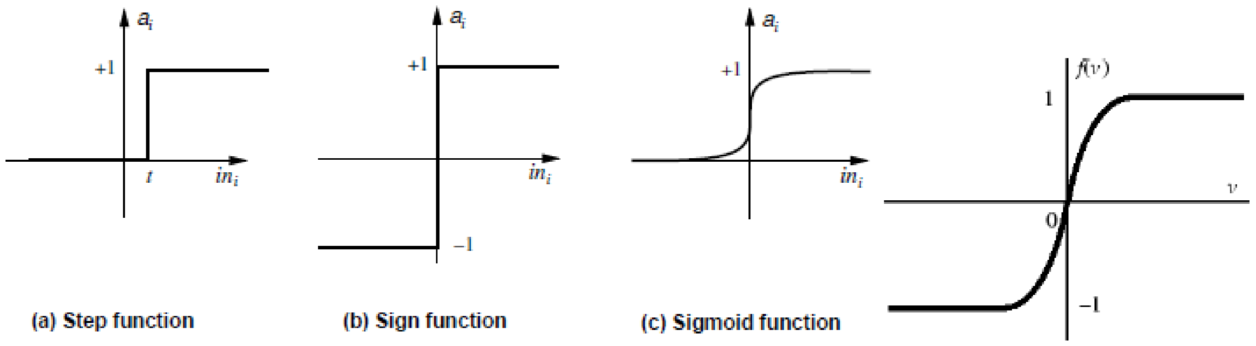
Continuous Activation Functions

The key idea is to replace the (discontinuous) step function with a differentiable function, such as the sigmoid:

$$g(s) = \frac{1}{1+e^{-s}}$$

or hyperbolic tangent:

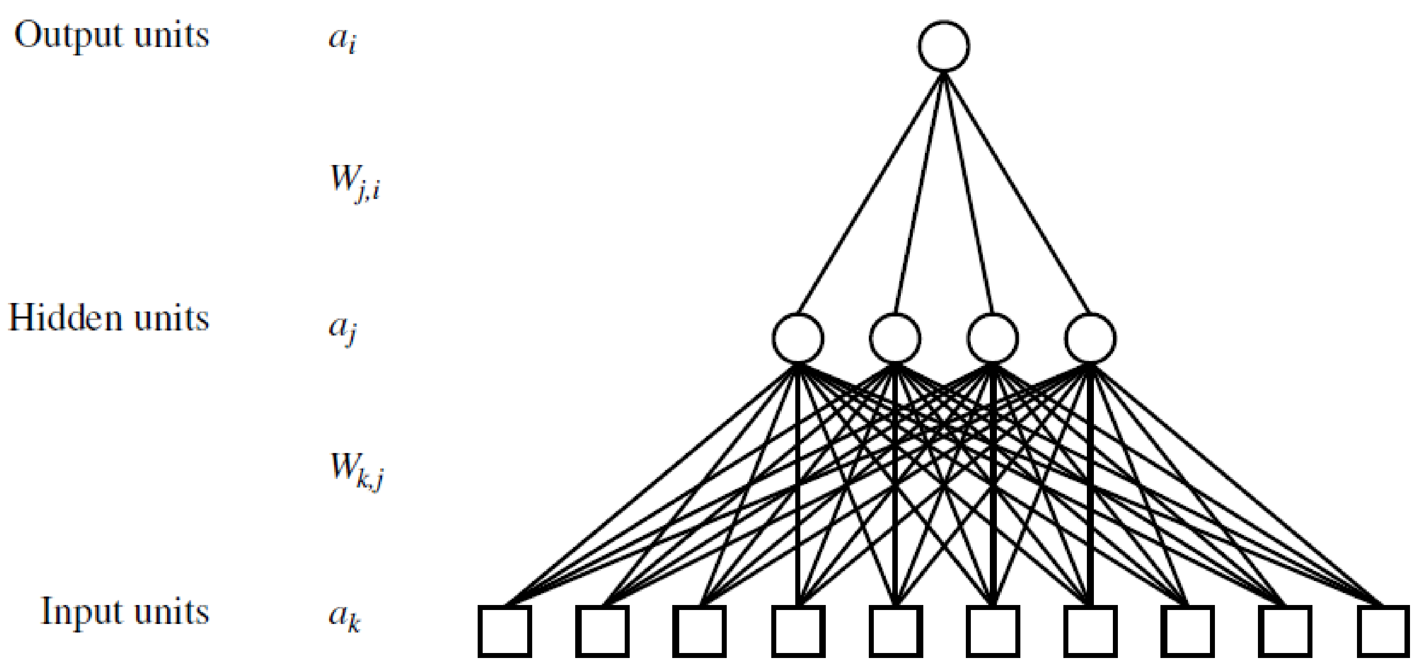
$$g(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} = 2\Big(\frac{1}{1+e^{-2s}}\Big) - 1$$



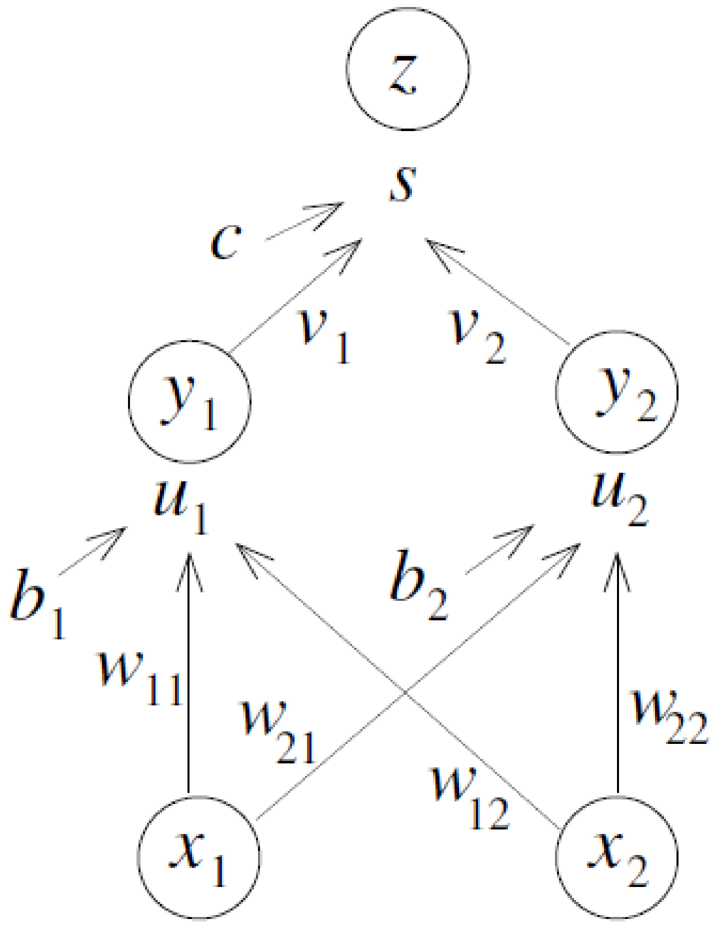
(a) Step function (b) Sign function (c) Sigmoid function

Backpropagation

We now describe how the partial derivatives of the loss function with respect to each weight can be computed. We consider the case of a 2-layer neural network with sigmoid activation at the hidden layers, as shown in this diagram.



For simplicity, we present the case with 2 inputs, 2 hidden units and 1 output.



$$\begin{aligned} u_1 &= b_1 + w_{11}x_1 + w_{12}x_2 \\ y_1 &= g\left(u_1\right) \\ s &= c + v_1 y_1 + v_2 y_2 \\ z &= g(s) \end{aligned}$$

We sometimes use w as a shorthand for any of the trainable weights $c, v_1, v_2, b_1, b_2, w_{11}, w_{21}, w_{12}, w_{22}$.

Chain Rule (6.5.2)

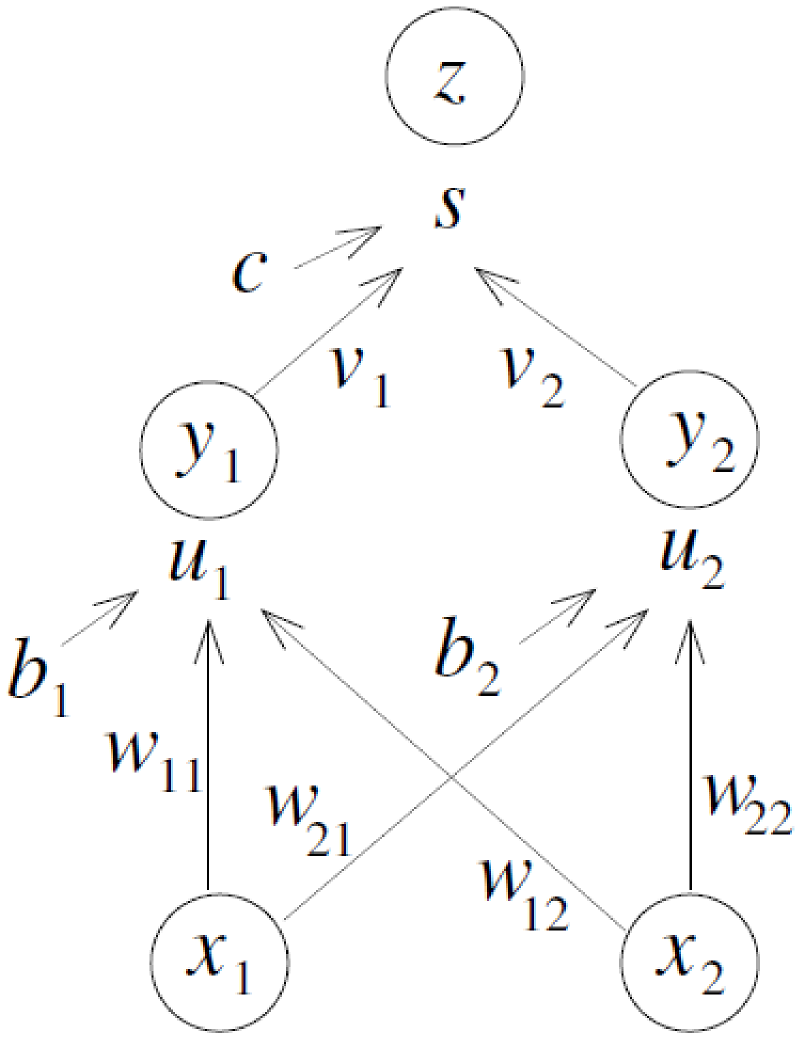
If $y = y(u, v)$ where $u = u(x)$ and $v = v(x)$ then

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial y}{\partial v}\frac{\partial v}{\partial x}$$

This principle can be used to compute the partial derivatives in an efficient and localized manner. Note that the transfer function must be differentiable (usually $\mathrm{sigmoid}$, or $\tanh$).

$$\begin{aligned} \text{Note: if } z(s) &= \frac{1}{1+e^{-s}}, \quad z'(s) = z(1-z) \\ \text{if } z(s) &= \tanh(s), \quad z'(s) = 1 - z^2 \end{aligned}$$

Forward Pass



$$\begin{aligned} u_1 &= b_1 + w_{11}x_1 + w_{12}x_2 \\ y_1 &= g\left(u_1\right) \\ s &= c + v_1 y_1 + v_2 y_2 \\ z &= g(s) \\ E &= \frac{1}{2}\sum(z-t)^2 \end{aligned}$$

Backward Pass

Partial Derivatives

$$\begin{aligned}
\frac{\partial E}{\partial z} &= z - t \\
\frac{dz}{ds} &= g'(s) = z(1-z) \\
\frac{\partial s}{\partial y_1} &= v_1 \\
\frac{dy_1}{du_1} &= y_1\left(1-y_1\right)
\end{aligned}$$

Useful notation

$$\delta_{\text{out}} = \frac{\partial E}{\partial s} \quad \delta_1 = \frac{\partial E}{\partial u_1} \quad \delta_2 = \frac{\partial E}{\partial u_2}$$

Then

$$\begin{aligned}
\delta_{\text{out}} &= (z-t)z(1-z) \\
\frac{\partial E}{\partial v_1} &= \delta_{\text{out}}\, y_1 \\
\delta_1 &= \delta_{\text{out}}\, v_1 y_1\left(1-y_1\right) \\
\frac{\partial E}{\partial w_{11}} &= \delta_1 x_1
\end{aligned}$$

Partial derivatives can be calculated efficiently by packpropagating deltas through the network.

References

Rumelhart, D.E., Hinton, G.E., & Williams, R.J., 1986. Learning representations by back-propagating errors. N*ature*, *323*(6088), 533-536.

Further Reading

Textbook Deep Learning (Goodfellow, Bengio, Courville, 2016):

- Continuous Activation Functions (3.10)
- Gradient Descent (4.3)
- Backpropagation (6.5.2)

Neural Network Training

Example: Pima Indians Diabetes Dataset

One example of the kind of task for which two-layer neural networks might be applied is the Pima Indians Diabetes Dataset. For each patient, the 8 input attributes shown in this table are provided. The network is trained to output a number between 0 and 1 indicating the probability that the patient is positive for diabetes.

	Attribute	mean	stdv
1.	Number of times pregnant	3.8	3.4
2.	Plasma glucose concentration	120.9	32.0
3.	Diastolic blood pressure (mm Hg)	69.1	19.4
4.	Triceps skin fold thickness (mm)	20.5	16.0
5.	2-Hour serum insulin (mu U/ml)	79.8	115.2
6.	Body mass index (weight in kg/(height in m) ^{2})	32.0	7.9
7.	Diabetes pedigree function	0.5	0.3
8.	Age (years)	33.2	11.8

Online, Batch and Minibatch Learning

You will notice that the loss function includes a summation over training items:

$$\mathrm{E} = \frac{1}{2}\sum_i (z_i - t_i)^2$$

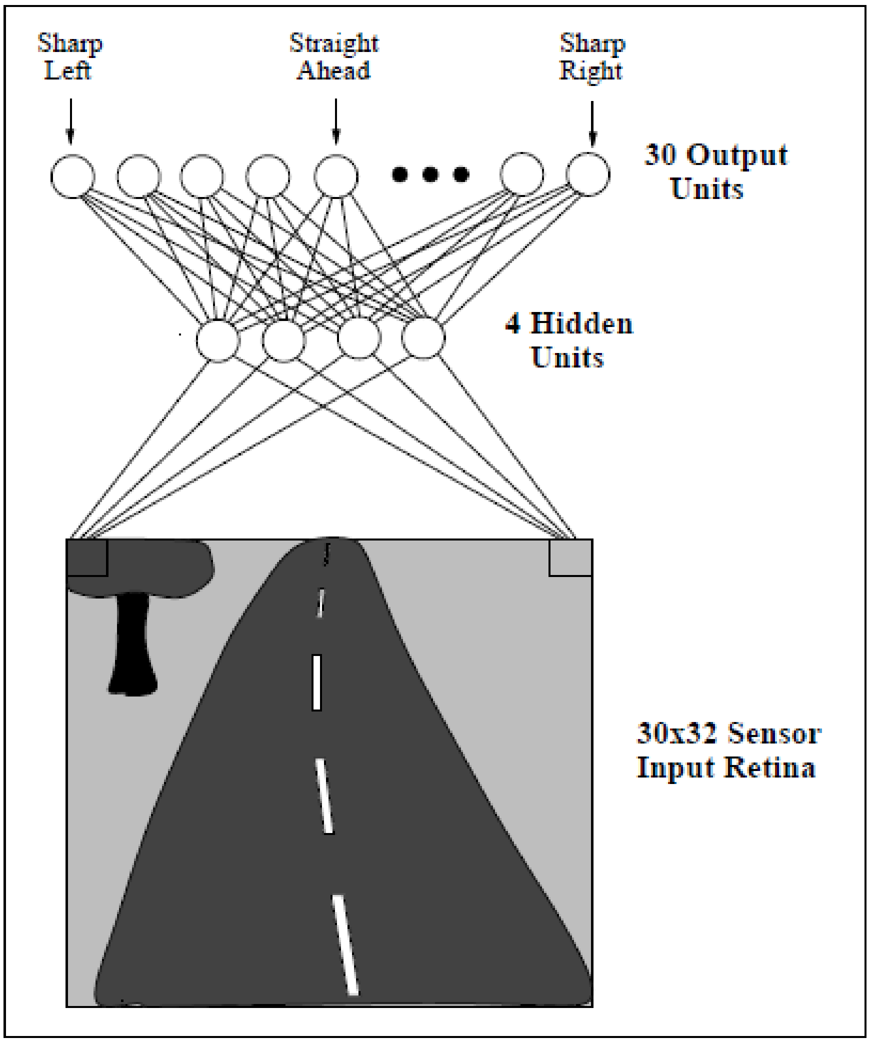
In some cases, the gradients for all training items are computed and added together, and this overall gradient is then used to update the weights in one step. This is known as **Batch Learning**. At the other extreme, we could compute the gradient for one item at a time as they are presented, and update the weights as we go, based on the gradient for each individual item. This is called **Online Learning**. To take advantage of parallel hardware, an intermediate approach known as **Minibatch Learning** is often employed, where the training items are divided randomly into minibatches of roughly equal size. The gradients are computed in parallel for all items in the minibatch, and the weights are updated accordingly. Because each minibatch represents only a portion of the full loss function, its gradient can be thought of as an approximation to the full gradient, with some implicit noise added. For this reason, online and minibatch learning are sometimes collectively referred to as **Stochastic Gradient Descent** (SGD).

In order for learning to be successful, we should try to avoid **temporal correlations** in the training process. In other words, each minibatch (or a set of consecutively presented items in the case of online learning) should contain a variety of different inputs and corresponding outputs, rather than containing very similar inputs with the same target output.

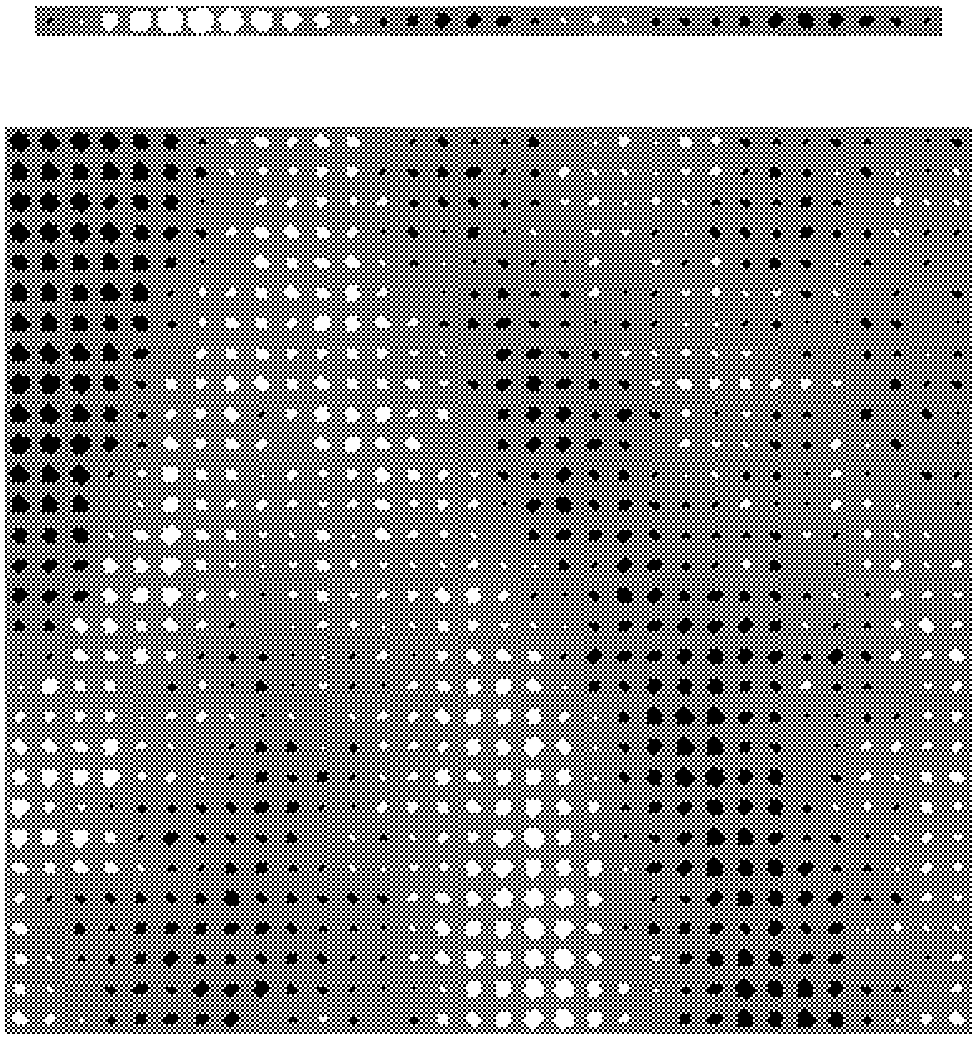
Autonomous Driving and Hinton Diagrams

Neural networks with one or two hidden layers found a diverse range of applications throughout the 1990's including medical diagnosis, game playing, credit card fraud detection, handwriting recognition, financial prediction, and many others.

We will describe ALVINN (Autonomous Land Vehicle in a Neural Network — Pomeleau, 1990) because it is a good early example of Behavioural Cloning, hidden unit visualization, Data Augmentation and Experience Replay. ALVINN was an early autonomous driving system which used a 2-layer neural network to determine the steering angle. The network takes as input a 30×32 sensor input retina and computes 30 outputs corresponding to different steering directions ranging from Sharp Left to Sharp Right. When the trained network is driving the vehicle, the steering direction is chosen as a weighted average over these 30 outputs.



For image processing or spatial tasks like this, the weights coming into and out of a particular hidden node can be visualized using a Hinton Diagram.



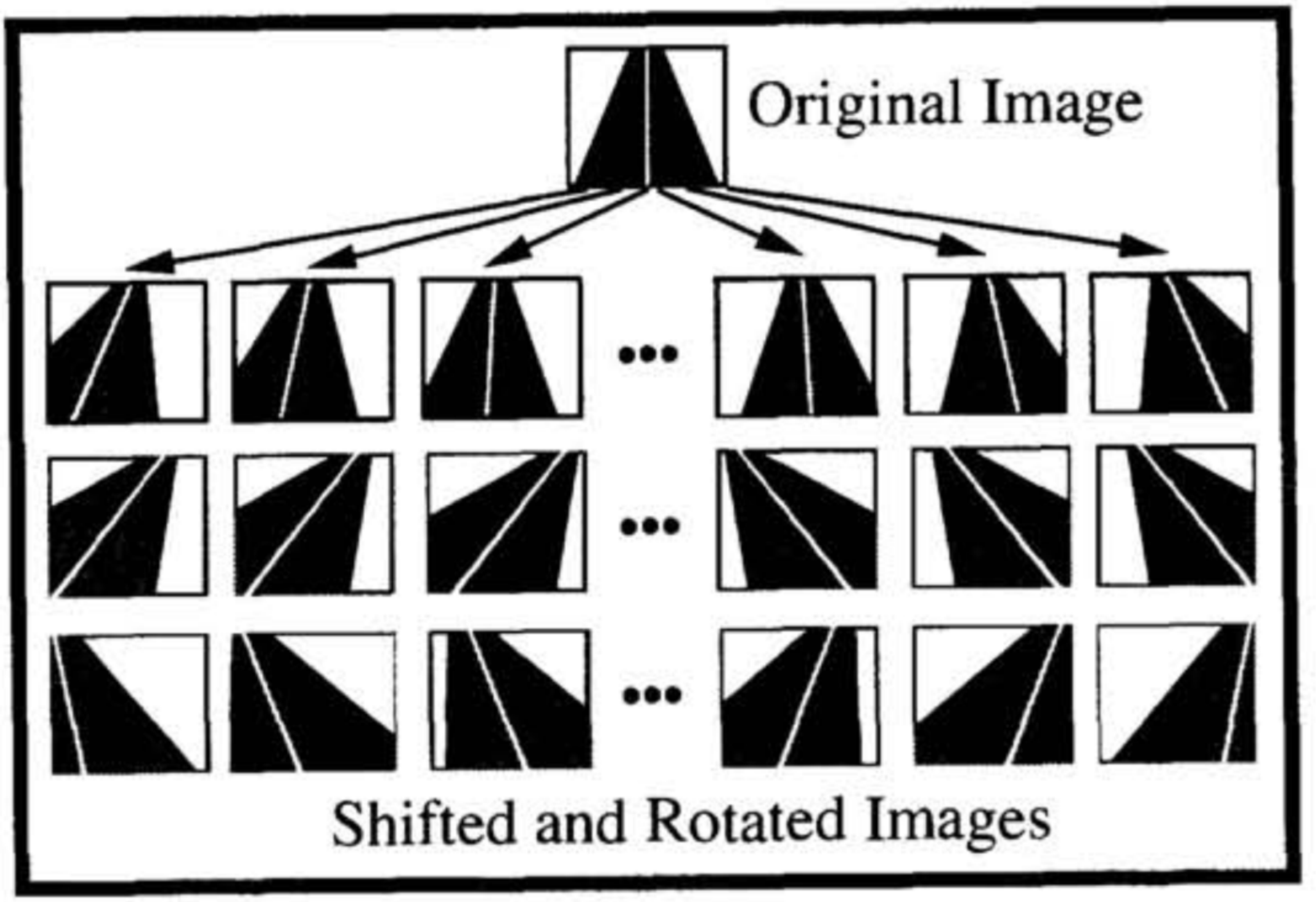
The dots in the top row correspond to hidden-to-output weights; those in the lower square correspond to input-to-hidden weights. Black and white dots represent positive and negative weight values, respectively.

Behavioral Cloning, Data Augmentation and Experience Replay

ALVINN was initially trained by a process known as Behavioral Cloning. The vehicle is driven for some time by a human, and a database is collected of sensor inputs and the corresponding actions chosen by the human driver. The network is trained on this database, and is then invited to take over the controls.

It was found that the network could not achieve proficiency in the task when trained on the human data alone. This is because, when the network was driving, the vehicle would start to veer off the road, and would then encounter situations that never occurred when the human was driving and therefore did not appear in the database.

(Pomerleau, 1990) solved this problem by using **Data Augmentation**, which means using domain knowledge to create additional training data. Every original image (collected while the human is driving) gets shifted and rotated in 14 different ways to create 14 additional training items. The steering angle is adjusted to compensate for the shift and rotation.



The other problem they encountered is that many similar images occur in rapid succession, with similar steering angles, leading to the temporal correlations discussed above. To avoid this, an **Experience Replay** buffer of 200 items is maintained. At each timestep, 15 old items are removed from the replay buffer and replaced with 15 newly generated items. The network is then trained on all 200 items currently in the replay buffer.

With these enhancements, ALVINN was able to steer autonomously across the United States in 1995 (although, for safety reasons, the accelerator and brake were still human controlled).

In Week 8, we will see how Reinforcement Learning can be used as an alternative to Behavioral Cloning for autonomous control tasks, or for learning to play board games or video games.

References

Pomerleau, D.A., 1990. Rapidly adapting artificial neural networks for autonomous navigation. In *Proceedings of the 3rd International Conference on Neural Information Processing Systems* (pp. 429-435).

Momentum and Adam

Momentum

It is often helpful to maintain a running average of the gradient for each weight, and use this running average to update that weight:

$$\begin{aligned}\delta w &\leftarrow \alpha\delta w - \eta\frac{\partial E}{\partial w}\\ w &\leftarrow w + \delta w\end{aligned}$$

The parameter $\alpha \in [0, 1)$ is called the **Momentum**.

This is helpful in two situations. Firstly, if the weights travel through a flat region in the landscape, momentum will help to speed up the learning in this region. Secondly, if the landscape is shaped like a “rain gutter”, weights will tend to oscillate without much improvement. Momentum will theoretically dampen the sideways oscillations but amplify the downhill motion by a factor of $\frac{1}{1-\alpha}$.

When momentum is introduced, we generally reduce the learning rate at the same time, in order to compensate for this implicit factor of $\frac{1}{1-\alpha}$.

Adaptive Moment Estimation (Adam)

Adaptive Moment Estimation or **Adam** maintains a running average of the gradients (m_t) and the squared gradients (v_t) for each weight in the network (Kingma & Ba, 2015).

$$\begin{aligned}m_t &= \beta_1 m_{t-1} + (1-\beta_1)g_t\\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)g_t^2\end{aligned}$$

To speed up the training in the early stages, compensating for the fact that m_t, g_t are initialized to zero, we rescale as follows:

$$\begin{aligned}\hat{m}_t &= \frac{m_t}{1-\beta_1^t}\\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t}\end{aligned}$$

Finally, each parameter is adjusted according to:

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon}\hat{m}_t$$

Adam has been found to work well across a wide variety of task domains.

As with momentum, when switching from normal gradient descent to Adam we often need to reduce the learning rate η in order to compensate for the factor of $\frac{1}{\sqrt{\hat{v}_t}+\varepsilon}$.

Second Order Methods

For smaller networks, optimisation methods such as Conjugate Gradients or Natural Gradients are sometimes used, which compute the second derivative of the loss function with respect to every pair of weights in the network. However, these methods are generally not practical in the context of deep learning, because the computation time is proportional to the square of the number of weights, which may be in the millions or even billions.

Adam is seen as a very cost effective method which, in practice, provides a similar benefit to second order methods, but with far less computation.

References

Kingma, D.P., & Ba, J., 2015. Adam: A method for stochastic optimization, *International Conference on Learning Representations* (poster).

Further Reading

https://ruder.io/optimizing-gradient-descent/index.html

Textbook Deep Learning (Goodfellow, Bengio, Courville, 2016):

- Momentum (8.3) and Adam (8.5)

Quiz 1: Perceptrons and Backprop

This is a Quiz to test your understanding of the material from Week 1.

You must attempt to answer each question yourself, before looking at the sample answer.

Question 1

What class of functions can be learned by a Perceptron?

No response

Question 2

Explain the difference between Perceptron Learning and Backpropagation.

No response

Question 3

When training a Neural Network by Backpropagation, what happens if the Learning Rate is too low? What happens if it is too high?

No response

Question 4

Explain why rescaling of inputs is sometimes necessary for Neural Networks.

No response

Question 5

What is the difference between Online Learning, Batch Learning, Mini-Batch Learning and Experience Replay? Which of these methods are referred to as "Stochastic Gradient Descent"?

No response

Question 6

Briefly explain the concept of Momentum, as an enhancement for Gradient Descent.

No response

Week 2 Tutorial

Week 2 Tutorial Questions