Neural Networks

Neural networks are semi-supervised models capable of both classification and regression. A #neural-network is effectively a graph made up of "layers", each one of which contains many nodes/neurons. Each of these neurons is a perceptron.

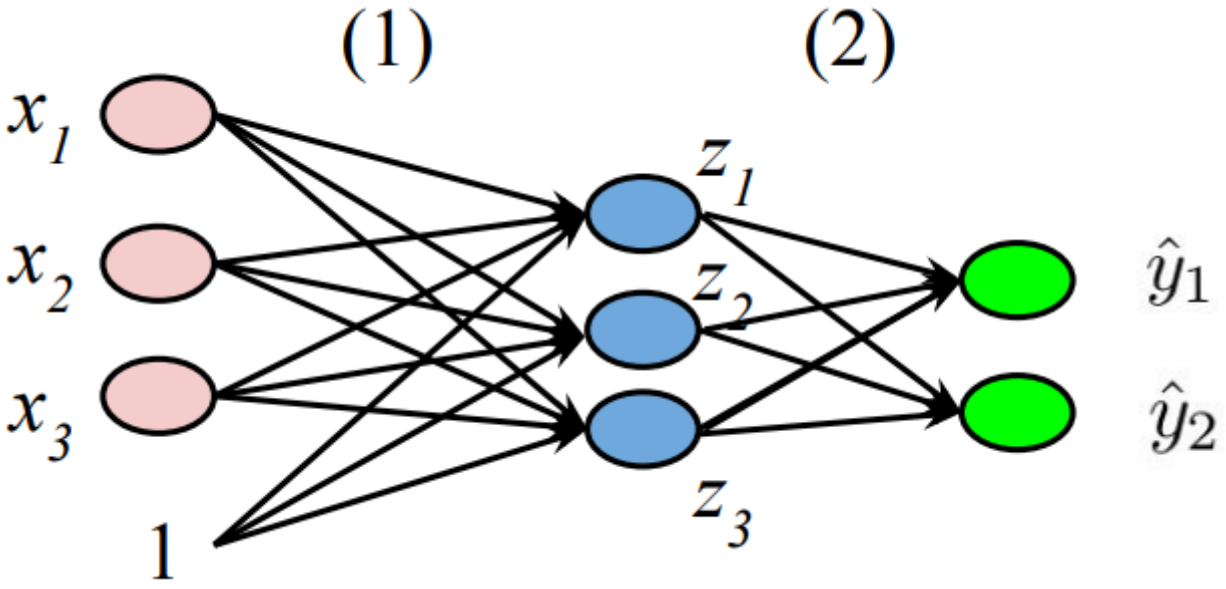
Before #neural-networks were invented, non-linear problems were unsolvable with simple, #linear-models. It seems counter-intuitive that the #neural-network should be our saviour in this scenario as they are made up of #perceptrons, a famously linear model, but the activation function of the #perceptron is precisely what allows us to introduce non-linearity.

Feed-Forward Neural Networks

Information is propagated from the input node(s) to the output node(s) through a directed, acyclic graph[1]. In each node the information is mutated by algebraic functions implemented using the connections, weights, and biases of the hidden and output layers.

> **Note**
>
> The hidden layers compute intermediate representations, which are then developed upon by either further hidden layers or the output layer.

Here we can see a simple, single layer, #feed-forward #neural-network. The formula to calculate each of the different z values is $z_j = \sum_i w_{i,j}^{(1)} x_i + w_{0,j}^{(1)}$ and the formula to calculate each of the various $\hat{y}$ output values is $\hat{y}_k = f(\sum_i w_{i,k}^{(2)} h(z_i) + w_{0,k}^{(2)})$



Limitations

Problem

Unfortunately, we can't learn a model using the techniques we've seen thus far. For the output layers we have direct supervision, that is the ground-truth labels, but for the hidden layers we cannot know the desired target and therefore the perceptrons can't be trained.

Back-propagation

Solution

The technique that comes in to save the day this time is #back-propagation . This technique consists of the following three steps:

- Forward propagation
  - Sum inputs
  - Produce activations
  - Feed-forward
- Error estimation
- #Back-propagation
  - Back-propagate the error signal and use it to update the weights

The Main Idea

Given training samples $T = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$ adjust all the weights of the network Θ such that a cost function is minimised: $min_\Theta \sum_i L(y_i, f(x_i; \Theta))$

- Choose a loss function
- Update the weights of each layer using gradient descent
- Use #back-propagation of the error signal to compute the gradient efficiently

Further Problems

Problem

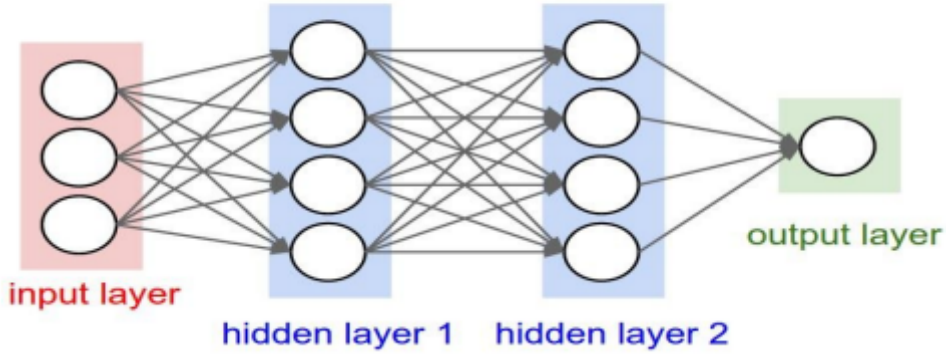
In this form, neural networks are unable to exploit the benefits that may come from having many layers as they risk not only overfitting but more importantly they suffer from the vanishing gradient problem. This is because in neural networks you have to multiply a lot of small numbers, and the more layers there are the more numbers there are. Multiplying so many small numbers causes them to get very small very quickly, and therefore the gradient gets shallow early on in the training, thus slowing it down considerably.

Basic Elements

Our goal is to approximate some ideal function $f^* : X \to Y$. A #feed-forward network requires defining a parametric mapping $f(x_i; \Theta)$ and learning parameters to get a good approximation of f^* given the available data samples.

The function f that we're learning is actually a composition of multiple functions which can be described by a DAG, with $f^{(1)}$ being the first layer, $f^{(2)}$ being the second layer, and so on. The depth is the maximum i in the function composition chain.

$$f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$$



Training

In training, we want to optimise Θ to drive $f(x; \Theta)$ closer to $f^*(x)$. To do this, we obtain training data such that each expected output value is equal to $f^*(x)$ for various instances of x. Note that this only specifies what we want the output layers to output; we leave the output of the intermediate layers unspecified.

Designing and training a neural network isn't all that different from training any other machine learning model with gradient descent.

Modelling Choices

Cost Function

The #cost-function measures how well the #neural-network performs on training data. We can apply any of the loss functions seen in Linear Models > Loss-Functions. For classification it is common to convert outputs into probabilities[2], that is, the outputs become values explaining how likely the #neural-network thinks it is that the current example belongs to each class.

> **Note**
>
> The choice of the loss used is related to the choice of the output unit.

Output Units - Linear

Given features h, a layer of linear output units gives $\hat{y} = W^T h + b$. Linear units do not saturate, which can cause some difficulty for gradient-based optimisation algorithms.

> **Note**
>
> Naturally, if the gradient of the output of the model is close to 0 it could be problematic.

Output Units - Softmax

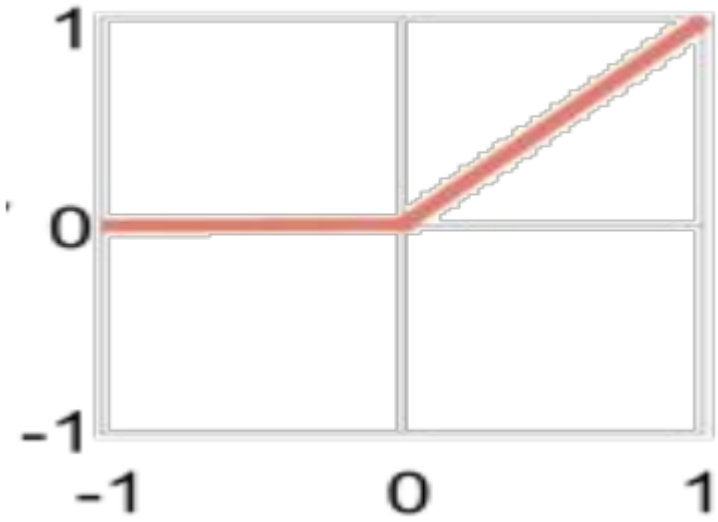
Sometimes we want to produce normalised probabilities in the output layer, but with linear units we produce unnormalised log probabilities. One solution to this problem is softmax: $S(l_i) = \frac{e^{l_i}}{\sum_k e^{l_k}}$ where the values of l_i are the scores, or logits.

Hidden Units

In a hidden unit, we have to accept an input x, compute an affine transformation $z = W^T x + b$, apply an elementwise non-linear function $h(z)$, and then finally obtain the output $h(z)$.

> **Note**
>
> The design of hidden units is an active area of research.

Rectified Linear Units (RELUs)



Here we can see a #RELU with equation $h = max(0, x)$.

In the case of a #RELU, the gradient is always either 0 or 1, making it similar to linear units and therefore easy to optimise as they give large and consistent gradients when active.

Unfortunately, they're not differentiable everywhere, although this isn't really a problem in practice as we can return one-sided derivatives at $z = 0$[3].

A large problem is that the units die when the gradient is 0[4], which people have tried to resolve by using the sigmoid[5] and hyperbolic tangent[6] functions, but they suffer from the same problem in that they saturate across most of the domain and are only strongly sensitive when the input is close to zero.

Back-propagation Continued

Key Ideas

For #back-propagation we need error derivatives for all the weights in the net.

Example

This is a possible formula for the weights of the first layer:

$$w_{11}^{(1)} = w_{11}^{(1)} - \eta \frac{\partial L}{\partial w_{11}^{(1)}}$$

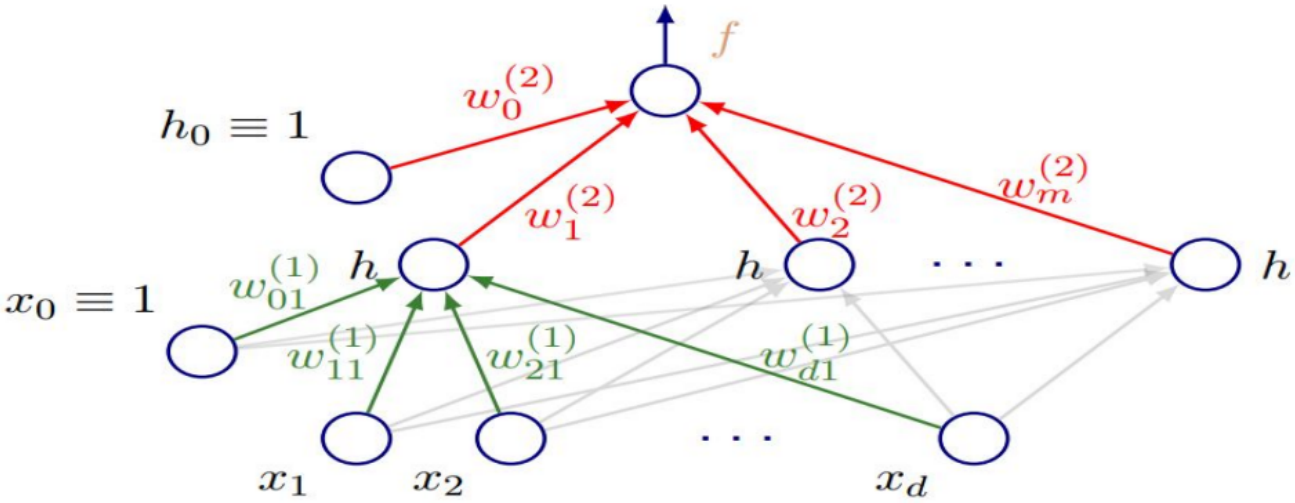
From the training data we can't tell what the hidden units should do, but we can compute how quickly the overall error of the network changes when we change a hidden activity. Each hidden unit can affect many output units and have various effects on the error, which we then combine.

Note

Computing error derivatives for hidden units is very efficient. Once we have error derivatives for a given hidden activity it's easy to get error derivatives for the weights leading to the said hidden activity.

Step 1 - Feedforward

Given the following neural network, a formula to represent it's output $\hat{y}$ given input x and weights w could be: $\hat{y}(x; w) = f(\sum_{j=1}^{m} w_j^{(2)} h(\sum_{i=1}^{d} w_{ij}^{(1)} x_i + w_{0j}) + w_0^{(2)})$ where f and h are functions.



Step 2 - Computing the Error and Training

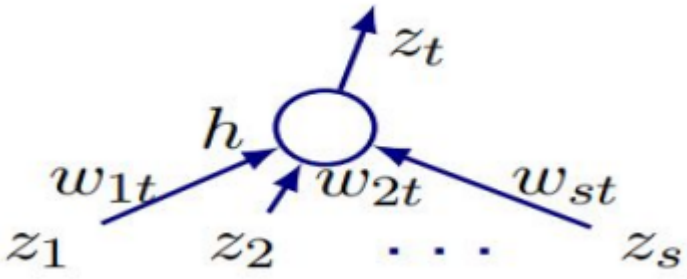
The error of the network on the training set is $L(X;w) = \sum_{i=1}^{N} \frac{1}{2}(y_i - \hat{y}(x_i;w))^2$.

As part of this function isn't linear there is no closed-form solution, meaning we have to resort to gradient-descent for the training.

We now need to evaluate the derivative of L on a single example which, using the simple linear model for output $\hat{y} = \sum_j w_j x_{ij}$, can be done like so: $\frac{\partial L(x_i)}{\partial w_j} = (\hat{y}_i - y_i)x_{ij}$ where $\hat{y}_i - y_i$ is the error of the model.

Step 3 - #Back-propagation

The output of a unit with activation function h like in the image below can be calculated as $z_t = h(\sum_j w_{jt} z_j)$ where t refers to the layer of the network the unit belongs to.



In forward propagation, we calculate $a_t = \sum_j w_{jt} z_j$ for each unit. The loss L then depends on w_{jt} only through the use of a_t: $\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t}\frac{\partial a_t}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j$

> **Note**
>
> Note that $\frac{\partial a_t}{\partial w_{jt}} = \frac{\partial \sum_j w_{jt} z_j}{\partial w_{jt}}$

In the final equation $\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j$ we can set the variable $\delta_t = \hat{y} - y$, which is the linear activation of the output unit.

In the case of a hidden unit t sending output to units in the set S, we have the following: $\delta_t = \sum_{s \in S} \frac{\partial L}{\partial a_s} \frac{\partial a_s}{\partial a_t} = h^{'}(a_t) \sum_{s \in S} w_{ts} \delta_s$ where $a_s = \sum_{j:j \to s} w_{js} h(a_j)$.

Gradient

The gradient is the vector of partial derivatives with respect to all the coordinates of the weights: $\Delta_w L = [\frac{\partial L}{\partial w_1} \frac{\partial L}{\partial w_2} \dots \frac{\partial L}{\partial w_M}]$

Each partial derivative measures how fast the loss changes in one direction, so when the gradient is zero[7] the loss is not changing in any direction.

Problems

This approach suffers from the classic problems of local minima and saddle points.

Optimisation Methods Based on Gradient Descent

Batch Gradient Descent

Algorithm

```
def BSD(eta_k, w):
        while stopping criteria not met:
                # compute gradient estimate over N examples
```

$$g \leftarrow \frac{1}{N} \Delta_w \sum_{i=1}^{N} L(f(x_i; w), y_i)$$

```
                # Apply update
```

$$w \leftarrow w - \eta_k g$$

The learning rate changes at each step, and typically decays linearly. This means that gradient estimates are stable, but for each update we need to calculate gradients over the entire training set.

Stochastic Gradient Descent

Algorithm

```
def SGD(eta_k, w):
        while stopping criteria not met:
                # Take one point from training set
```

$$(x_i, y_i) \; for \; some \; i$$

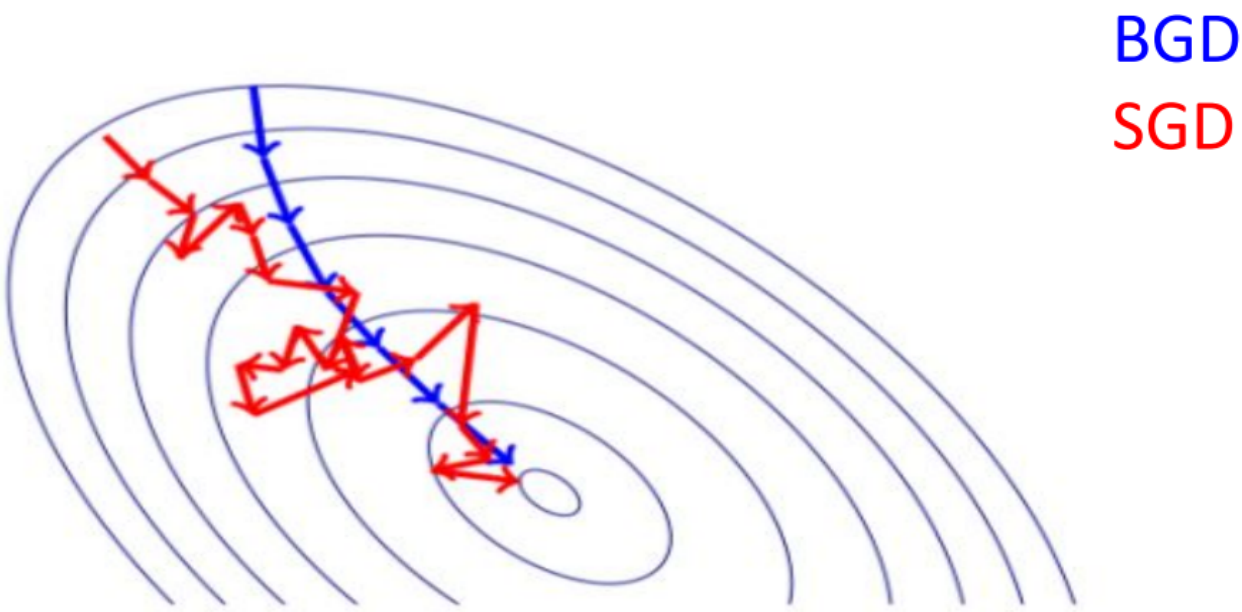
```
                # Compute gradient estimate
```

$$g \leftarrow \Delta_w L(f(x_i; w), y_i)$$

```
                # Apply update
```

$$w \leftarrow w - \eta_k g$$

While SGD logically gets to the same final result as BGD, the path it takes to get there is drastically different. As we are taking a single point from the training set to base our gradient estimate on, we don't generally know exactly which direction the endpoint is in despite always making progress in one dimension or another. The result of this is that the path we take isn't optimal, and appears sporadic in comparison to the path that BGD takes.



Adaptive Learning Rate Methods

A problem with the methods explained above is that we assign the same learning rate to every feature, which isn't a good idea if the features vary in importance.

> **Note**
>
> The learning rate is one of the most difficult to set hyperparameters in neural networks.

Easier: all the features important ***Harder***

Convolutional Neural Networks

A #convolutional-neural-network is a #neural-network that uses #convolution instead of matrix multiplication in at least one of its layers.

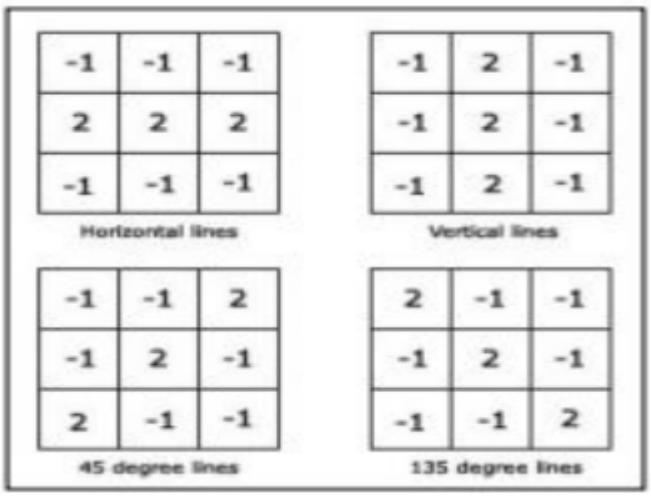
Convolution

#Convolution is a general purpose filter operation for images. It works by determining the value of a central pixel by adding the weighted values of all its neighbours together. The matrix of these weights is called a kernel matrix. The output is a new, modified, filtered image.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

[8]

We can use this commutative operation to smooth, sharpen, or even enhance images. The following kernels could be used for line detection:



---

1. DAG ↩

2. generally $[0, 1]$ ↩
3. gradient-based optimisation is prone to numerical error anyway ↩
4. saturation ↩
5. $h(x) = \frac{1}{1+e^{-x}}$ ↩
6. $h(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ ↩
7. all the partial derivatives are zero ↩
8. where I refers to the image and K refers to the kernel matrix ↩