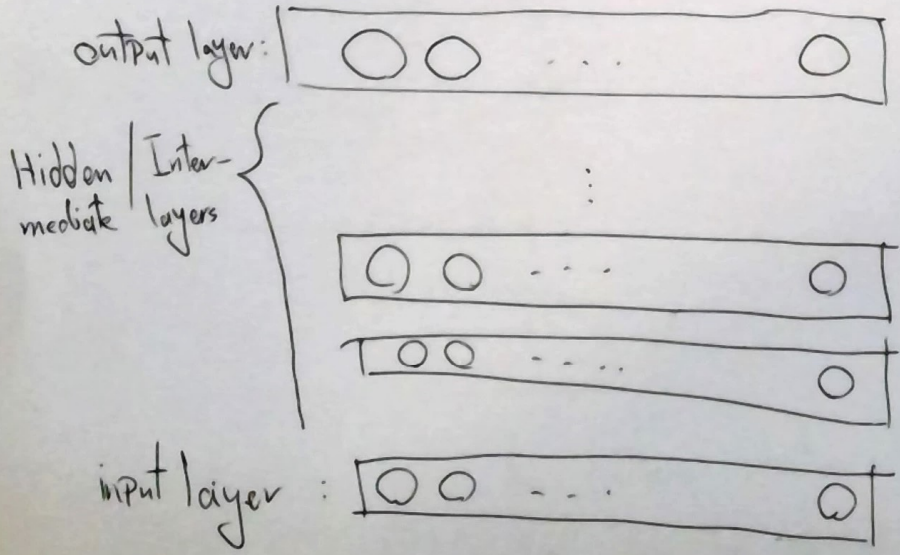
Network structure and forward Pass

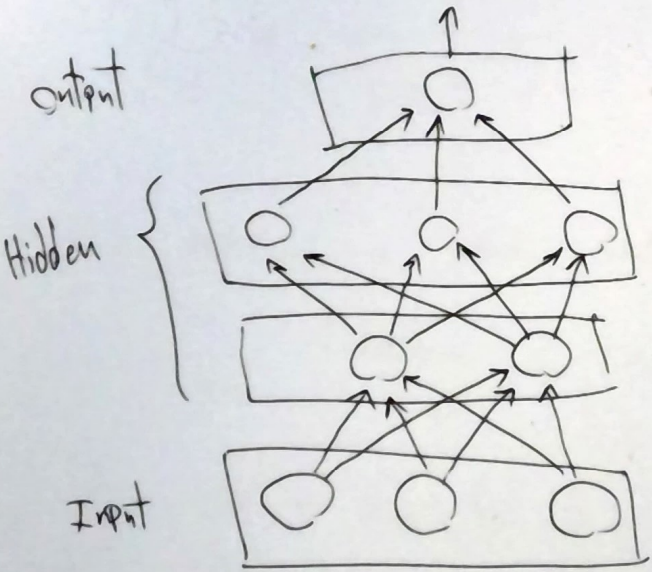
the simplest form of Network structure described by the Paper is,

" layered Networks which have a layer of input units at the bottom; any number of inter mediate layers; and a layer of outputs at the top "



Further, connections between units in the same layer is forbidden, and connections from upper layer back to lower layers is also forbidden.

with these limitations in place our Network will resemble "Feed forward Neural Networks"



It is interesting that the Authors mention that "intermediate forward connections can be skipped!" which is an interesting note since that is what we do randomly and systematically when we use Dropout!

Connections are described as following,

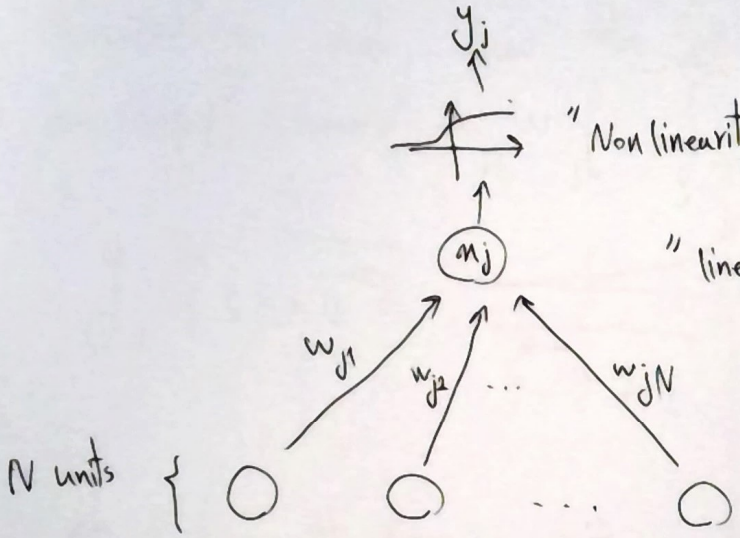
$$x_j = \sum_i y_i w_{ji} \qquad (1)$$

$$y_j = \frac{1}{1 + e^{-x_j}} \qquad (2)$$

(1) represents a lineur combination of states y_i combining to create the value x_j.

This value is then feed to a non-linearity function (2) which prodces y_j, the state of the new unit

Pictorially we can represent these operations as connection between units in the network



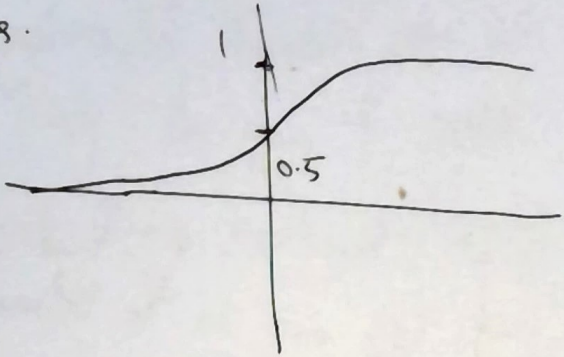
we can also add a bias to this unit by introducing a new input with a value of 1

$$a_j = \sum_i y_i w_{ji} = 1 \times w_{j0} + y_1 w_{j1} + \cdots + y_n w_{jn}$$

$$a_j = \underbrace{w_{j0}}_{\text{bias}} + y_1 w_{j1} + \cdots + y_n w_{jn}$$

This linear function is then fed into (2) a non linear function which introduces soft thresholding and allow the network to learn non linear representations.

$$y_j = \frac{1}{1+e^{-x_j}}$$



At this point the Perper mentions that (1) and (2) can be any function with bounded derivatives. but applying a linear function and then introducing non linearity is a good way to learn representation of input data.

In fact what is described here is the regular Dense layer with sigmoid activation function used for non linearity.

Now the goal is to find a set of weights to ensure that given a set of input data $\{(x_i, y_i) \mid i = 1, \dots m\}$ the output of the network for any x_i is sufficiently close to y_i.

If there is a fixed and finite set of samples the total error in the performance of the network is introduced as

$$E = \frac{1}{2} \sum_c \sum_j (y_{j,c} - d_{j,c})^2 \quad (3)$$

where j is index over output units, c is index over training example and d_c is the true output of x_c. (y_c is the output of Network for input x_c)

This loss function is summing over Norm-2 squared of difference between Predicted and desired output of the Network.

Backward Pass and finding derivatives

In order to minimize E with gradient descent we need Partial derivatives of E with respect to weights which is simply the sum of Partial derivative for any given traing example w.R.t. weights.

derivatives are computed in 2 steps:

1- Forward Pass, which we have already done in which states of units are determined layer by layer until we get output and compute the total Error.

2- Propagates derivative backward from top layer down.

First calculate $\frac{\partial E}{\partial y}$ for each unit of output

$$\frac{\partial}{\partial y_j}\left(\frac{1}{2}\sum_c\sum_j (y_{j,c} - d_{j,c})^2\right)$$

$$= \frac{1}{2} \cdot 2 (y_{j,c} - d_{j,c}) \quad ; \text{ suppressing } c$$

$$\frac{\partial E}{\partial y_j} = y_j - d_j \qquad (4)$$

By applying the chain rule,

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \cdot \frac{d y_j}{d x_j}$$

by differentiating (2)

$$y_j = \frac{1}{1+e^{-x_j}}$$

$$\frac{dy_j}{dx_j} = \frac{-(-e^{-x_j})}{(1+e^{-x_j})^2} = \frac{e^{-x_j}}{1+e^{-x_j}} \cdot \frac{1}{1+e^{-x_j}}$$

$$= \left(1 - \frac{1}{1+e^{-x_j}}\right)\left(\frac{1}{1+e^{-x_j}}\right)$$

$$\frac{dy_j}{dx_j} = (1-y_j)\, y_j$$

So, by using (4) and above we get,

$$\frac{dE}{dx_j} = \frac{dE}{dy_j} \cdot (1-y_j)\, y_j \qquad (5)$$

x_j is a linear function of its inputs and weights, to find $\frac{\partial x_j}{\partial w_{ji}}$ for a weight from unit i to j,

$$\frac{\partial x_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \left(\sum_i y_i w_{ji} \right) = y_i$$

So to find $\frac{\partial E}{\partial w_{ji}}$,

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial x_j} \cdot \frac{\partial x_j}{\partial w_{ji}} = \frac{\partial E}{\partial x_j} \cdot y_i \quad (6)$$

We can now compute $\frac{\partial E}{\partial w}$ for all w in this layer.

Now if we look at contribution of ith unit in previous layer to $\frac{\partial E}{\partial y_i}$ resulting from effect of i on j is

$$\frac{\partial E}{\partial y_i}_{\substack{\text{from} \\ i \to j}} = \frac{\partial E}{\partial x_j} \cdot \frac{\partial x_j}{\partial y_i} = \frac{\partial E}{\partial x_j} \cdot w_{ji}$$

to find $\frac{\partial E}{\partial y_i}$ we need to sum over all units in the last layer

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} \cdot w_{ji} \qquad (7)$$

we can now compute $\frac{\partial E}{\partial y}$ for units in layer $n-1$ given $\frac{\partial E}{\partial y}$ for units in layer n, and calculate $\frac{\partial E}{\partial w}$ for weight, in layer n.

We can therefore move backward through layers calculating $\frac{\partial E}{\partial w}$ as we go!

with $\frac{\partial E}{\partial w}$ in hand we can use gradient descent to update Parameters of the network to minimize Error

we can use the simple update rule

$$\Delta w = - \varepsilon \frac{\partial E}{\partial w}$$

where ε is a small hyper Parameter to be tuned for specific case called learning rate.

we can either update after observing any single train example, or accumulate them and update after we see all of the train example.

It is interesting that the two methods Presented cover Batch gradient descent and Stochastic gradient descent but doesn't explore the option of random mini-batches which is the most common Practice nowadays!

It is also of note that authors mention speeding up training by adding an acceleration term to the update rule.

$$\Delta w(t) = -\varepsilon\left(\frac{\partial E}{\partial w}\right)(t) + \alpha \Delta w\,(t-1)$$

and this is another formulation of gradient descent with Momentum which uses exponentially weighted average to smooth out oscilations in gradient descent

$$\Delta w(t) = -\varepsilon \frac{\partial E}{\partial w}(t) + \alpha \Delta w(t-1)$$

Momentum:

$$v = \beta v + (1-\beta)\left(\frac{\partial E}{\partial w}\right)$$

$$\Delta w = -\varepsilon v$$

Modern formulation

$$\Delta w = -\varepsilon\beta v - \varepsilon(1-\beta)\left(\frac{\partial E}{\partial w}\right)$$

$$\Delta w = -\varepsilon'\left(\frac{\partial E}{\partial w}\right) - \varepsilon\beta v$$

$$\Delta w = -\varepsilon'\left(\frac{\partial E}{\partial w}\right) + \beta \Delta w$$

$$\Delta w = -\varepsilon'\left(\frac{\partial E}{\partial w}\right) + \alpha \Delta w$$

1986 Paper

Finally there are two amazing observations by the author,

1. we can use random small values to initialize weights to break symmetry of units in one layer!

which is exactly what we used to date with methods such as Xavier Initialization!

2. Authors mention local minima as a drawback of such networks!

but as we know now, in very high dimensional surfaces such as loss function in neural networks, critical points are saddle points most of the time and local minimum is not a big Problem in neural networks!