Original Problem: How do we make linear classifiers non-linear?

$$\vec{w}^T \vec{x} + b \rightarrow w^T \phi(\vec{x}) + b$$

where kernilization $\phi(\vec{x})$ is a clever way to make inner products computationally tractable.

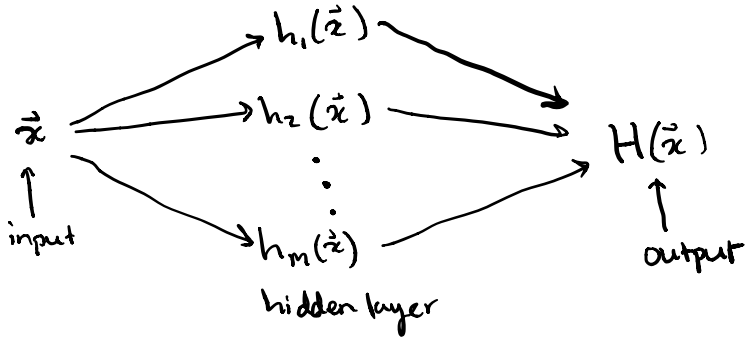
Neural networks learn ϕ:

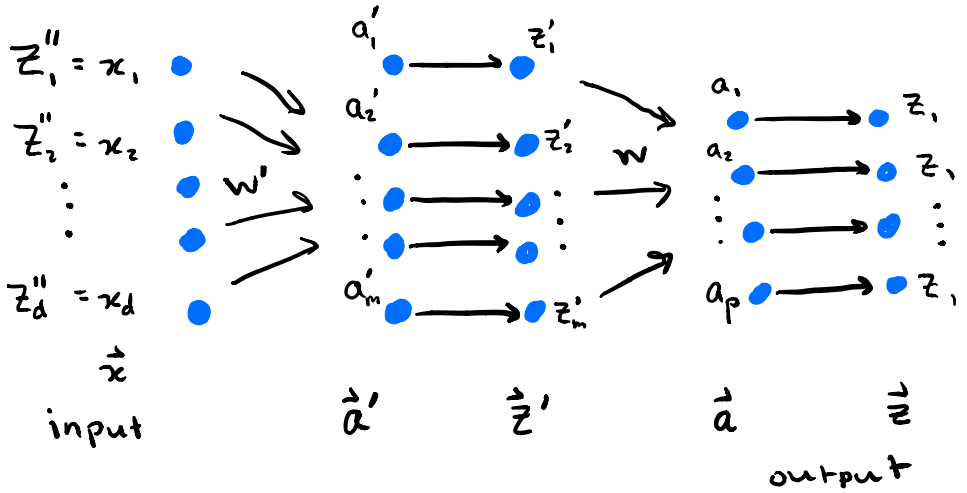
$$\phi(\vec{x}) = \begin{bmatrix} h_1(x) \\ \vdots \\ h_m(x) \end{bmatrix}$$

Each $h_i(\vec{x})$ is a linear classifier. This learns how level problems are "simpler".

An example is digit classification, each $h_i(\vec{x})$ detects something different like vertical edges, round shapes, horizontals.

Their output then becomes the input to the **main** linear classifier ($\vec{w}$).





$a'_j = \sum_k w'_{jk} + b'$

$a_j = \sum_j w_{jk} z'_j + b$

Forward Propagation

We can express $\vec{\hat{z}}$ in terms of w, w', b, b', f, y in matrix notation.

We need to learn w, w', b, b', and we can do so through gradient descent.

Back Propogation: Loss function for a single example (For the entire training set average over ALL training points)

$$L(\vec{\hat{x}}, \vec{\hat{y}}) = \frac{1}{2}\left(H(\vec{\hat{x}}) - \vec{\hat{y}}\right)^2$$

where $H(\vec{x}) = \vec{\hat{z}}$

$$L = \frac{1}{2}\left(\vec{\hat{z}} - \vec{\hat{y}}\right)^2$$

We learn W with gradient descent.

Observation (via chain rule):

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial \alpha_i}\frac{\partial \alpha_i}{\partial w_{ij}} = \frac{\partial L}{\partial \alpha_i} z'_j$$

$$\frac{\partial L}{\partial w'_{jk}} = \frac{\partial L}{\partial \alpha'_j}\frac{\partial \alpha'_j}{\partial w'_{jk}} x_k = \frac{\partial L}{\partial \alpha'_j} z''_k$$

Let $\vec{\delta} = \frac{\partial L}{\partial \vec{\alpha}}$ and $\vec{\delta}' = \frac{\partial L}{\partial \vec{\alpha}'}$ (i.e $\delta'_j = \frac{\partial L}{\partial \alpha_j}$)

Gradients are simple if we know $\vec{\delta}, \vec{\delta}', \vec{\delta}'', \vec{\delta}'''$, etc

So what is δ?

$$\delta_i = \frac{\partial L}{\partial \alpha_i} = \frac{\partial L}{\partial z_i}\frac{\partial z_i}{\partial \alpha_i} = (z_i - y_i) g'(\alpha_i) = g'(\vec{\alpha}) \circ (\vec{z} - \vec{y})$$

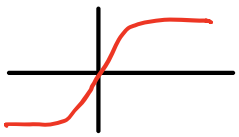
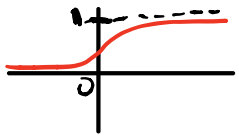
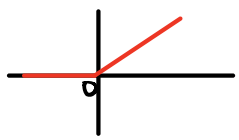
(Note that $L = \frac{1}{2}(z_i - y_i)^2$ and $z_i = g(\alpha_i)$)

$$\delta_j' = \frac{\partial L}{\partial \alpha_j} = \sum_i \underbrace{\frac{\partial L}{\partial z_i}\frac{\partial z_i}{\partial \alpha_i}}_{\delta_i} \frac{\partial \alpha_i}{\partial z_j'}\frac{\partial z_j'}{\partial \alpha_j} = \sum_i \delta_i \frac{\partial \alpha_i}{\partial z_j'}\frac{\partial z_j'}{\partial \alpha_j}$$

Note that $\alpha = \sum_j w_{ij} z_j' + b$ and $\frac{\partial z_j'}{\partial \alpha_j'} = \delta'(\alpha_j')$

$$f(\alpha_j') \sum_i \delta_i W_{ij} = \vec{f}\,'(\vec{\alpha}') \circ (W^T \delta)$$

Typical Transition Functions

Name	Function	Gradient	Expanded	Graph
tanh	$\tanh(u)$	$1 - (\tanh(u))^2$	$\tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}$	
Sigmoid	$\sigma(u)$	$\sigma(u)(1 - \sigma(u))$	$\sigma(u) = \frac{1}{1+e^{-u}}$	
Rectified Linear Unit	$\max(0,u)$	$\begin{cases} 1 & \text{if } u > 0 \\ 0 & \text{if } u < 0 \\ \text{N/A} & \text{if } u = 0 \end{cases}$		

ALGORITHMS

Forward Pass

Algorithm 1 Feed Forward Neural Network: Forward Pass

procedure ForwardPass($\vec{x}$)
  $\vec{z}_0 = \vec{x}$
  for $l = 1$ to L **do**
    $\vec{\alpha}_l = W_l \vec{z}_{l-1} + \vec{b}_l$
    $\vec{z}_l = f_l(\vec{\alpha}_l)$
  end for
  return $\vec{z}_L$
end procedure

Backward Pass

Algorithm 2 Feed Forward Neural Network: Backward Pass

procedure BackProp($\vec{x}$)
  $\vec{\delta}_L = \frac{\partial L}{\partial \vec{z}_L} \circ f_L'(\vec{\alpha}_L)$
  for $l = L:-1:1$ **do**
    $\Delta W_l = \vec{\delta}_l \vec{z}_{l-1}^T$
    $\Delta \vec{b}_l = \vec{\delta}_l$
    $\vec{\delta}_{l-1} = f_{l-1}'(\vec{\alpha}_{l-1}) \circ (W_l^T \vec{\delta}_l)$
    $\vec{W}_l = \vec{W}_l - \alpha \Delta W_l$
    $\vec{b}_l = \vec{b}_l - \alpha \Delta \vec{b}_l$
  end for
end procedure

Famous Theorem

- ANN are universal approximators
- Theoretically an ANN with one hidden layer is as expressive as one with many hidden layers in practice if many nodes are used

Overfitting in ANN

Neural networks learn a lot of parameters and thus are prone to overfitting

This isn't necessarily a problem as long as you use regularization.

Two popular regularizers are the following:

① Weight Decay: use l_2 regularization on all the weights (including bias terms)

② Dropout: For each input (or mini-batch) randomly remove each hidden node with probability p. These nodes stay removed during the backprop pass, however are included again for the next input

Avoidance of Local Minima

① Use momentum: Decline $\nabla w_t = \Delta w_t + \mu \Delta w_{t-1}$; $w = w - \alpha \nabla w_t$

i.e still use portion of the last gradient to keep pushing out of small local minima

② Initialize weights cleverly (not all that important)

③ Use Relu instead of sigmoid/tanh (weights don't saturate)

Tricks and Tips

- Rescale so all features are within $[0,1]$
- Lower learning rate
- Use mini-batch
- Use convolutional neural networks for images