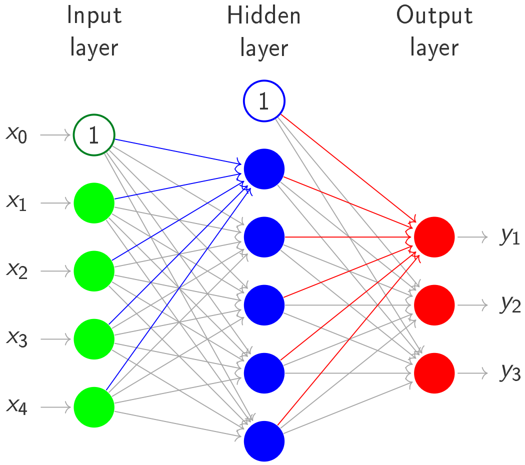
CS-563 Deep Learning

Training Neural Networks: Forward and Backward Propagation

Nazar Khan
Department of Computer Science
University of the Punjab

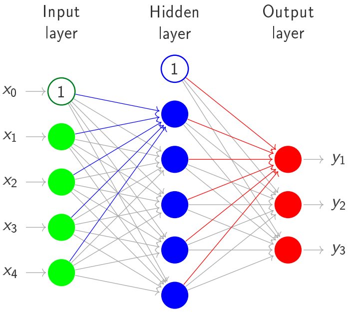
Neural Networks



Output of a neural network can be visualised graphically as *forward propagation of information*.

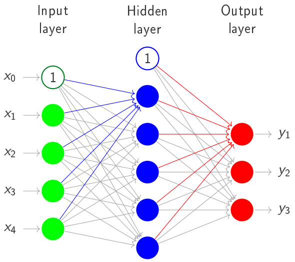
Neural Networks
Notation

- Input layer neurons will be indexed by i.
- Hidden layer neurons will be indexed by j.
- Next hidden layer or output layer neurons will be indexed by k.
- Weights of j-th hidden neuron will be denoted by the vector $\mathbf{w}_j^{(1)} \in \mathbb{R}^D$.
- Weight between i-th input neuron and j-th hidden neuron is $w_{ji}^{(1)}$.
- Weights of k-th output neuron will be denoted by the vector $\mathbf{w}_k^{(2)} \in \mathbb{R}^M$.
- Weight between j-th hidden neuron and k-th output neuron is $w_{kj}^{(2)}$.



Neural Networks
Forward Propagation

- For input $\mathbf{x}$, denote output of hidden layer as the vector $\mathbf{z}(\mathbf{x}) \in \mathbb{R}^M$.
- Model $z_j(\mathbf{x})$ as a non-linear function $h(a_j)$ where *pre-activation* $a_j = \mathbf{w}_j^{(1)T}\mathbf{x}$ with adjustable parameters $\mathbf{w}_j^{(1)}$.



- So the k-th output can be written as

$$y_k(\mathbf{x}) = f(a_k) = f(\mathbf{w}_k^{(2)T}\mathbf{z}(\mathbf{x}))$$
$$= f\left(\sum_{j=1}^{M} w_{kj}^{(2)} z_j(\mathbf{x}) + w_{k0}^{(2)}\right) = f\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right) + w_{k0}^{(2)}\right)$$

where we have prepended $x_0 = 1$ to to absorb bias input and $w_{j0}^{(1)}$ and $w_{k0}^{(2)}$ represent biases.

Neural Networks
Forward Propagation

- The computation

$$y_k(\mathbf{x}, \mathbf{W}) = f\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right) + w_{k0}^{(2)}\right)$$

can be viewed in two stages:

1. $z_j = h(\mathbf{w}_j^{(1)\,T}\mathbf{x})$ for $j = 1, \ldots, M$.
2. $y_k = f(\mathbf{w}_k^{(2)\,T}\mathbf{z})$.

Neural Networks
Forward Propagation

- If we define the matrices

$$\mathbf{W}^{(1)} = \underbrace{\begin{bmatrix} \longleftarrow & \mathbf{w}_1^{(1)T} & \longrightarrow \\ \longleftarrow & \mathbf{w}_2^{(1)T} & \longrightarrow \\ & \vdots & \\ \longleftarrow & \mathbf{w}_M^{(1)T} & \longrightarrow \end{bmatrix}}_{M\times(D+1)} \text{ and } \mathbf{W}^{(2)} = \underbrace{\begin{bmatrix} \longleftarrow & \mathbf{w}_1^{(2)T} & \longrightarrow \\ \longleftarrow & \mathbf{w}_2^{(2)T} & \longrightarrow \\ & \vdots & \\ \longleftarrow & \mathbf{w}_K^{(2)T} & \longrightarrow \end{bmatrix}}_{K\times(M+1)}$$

  then forward propagation constitutes
  1. $\mathbf{z} = h(\mathbf{W}^{(1)}\mathbf{x})$.
  2. Prepend 1 to $\mathbf{z}$.
  3. $\mathbf{y} = f(\mathbf{W}^{(2)}\mathbf{z})$.

Neural Networks for Regression

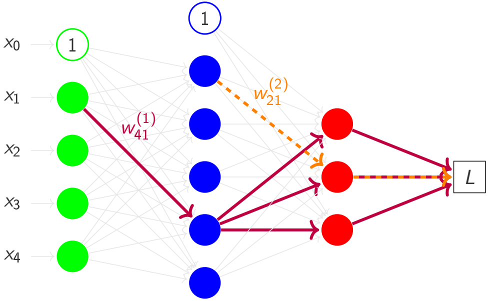
Gradients

- Regression requires continuous output $y_k \in \mathbb{R}$.
- So use *identity* activation function $y_k = f(a_k) = a_k$.
- Loss can be written as

$$L(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}) = \frac{1}{2} \sum_{n=1}^{N} \underbrace{\|\mathbf{y}_n - \mathbf{t}_n\|^2}_{L_n} = \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} (y_{nk} - t_{nk})^2$$

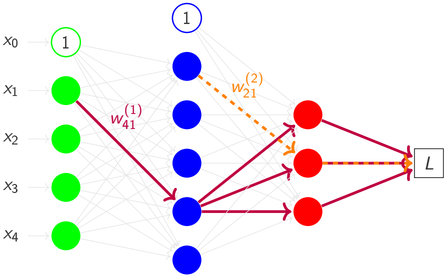
- Loss L depends on sum of individual losses L_n.
- In the following, we will focus on loss L_n for the n-th training sample.
- We will drop n for notational clarity and refer to L_n simply as L.

How do weights influence loss?



- $w_{kj}^{(2)}$ influences $a_k^{(2)}$ which influences y_k which influences L.
- For scalar dependencies, use chain rule.
- $w_{ji}^{(1)}$ influences $a_j^{(1)}$ which influences z_j which influences $a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$ which influence y_1, y_2, y_3 which influence L.
- For vector/multivariate dependencies, use multivariate chain rule.

How do weights influence loss?



- Layer 2: $L \leftarrow y_k \leftarrow a_k^{(2)} \leftarrow w_{kj}^{(2)}$.

$$L(y_k(a_k^{(2)}(w_{kj}^{(2)})))$$

- Layer 1: $L \leftarrow \mathbf{y} \leftarrow \mathbf{a}^{(2)} \leftarrow z_j \leftarrow a_j^{(1)} \leftarrow w_{ji}^{(1)}$.

$$L(\underbrace{y_1(a_1^{(2)}(z_j(a_j^{(1)}(w_{ji}^{(1)}))))}_{y_1(w_{ji}^{(1)})}, \underbrace{y_2(a_2^{(2)}(z_j(a_j^{(1)}(w_{ji}^{(1)}))))}_{y_2(w_{ji}^{(1)})}, \ldots, \underbrace{y_k(a_k^{(2)}(z_j(a_j^{(1)}(w_{ji}^{(1)}))))}_{y_k(w_{ji}^{(1)})})$$

Multivariate Chain Rule

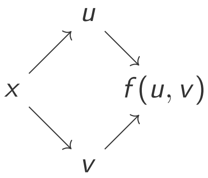
- The chain rule of differentiation states

$$\frac{df(u(x))}{dx} = \frac{df}{du}\frac{du}{dx}$$

$$x \longrightarrow u \longrightarrow f$$

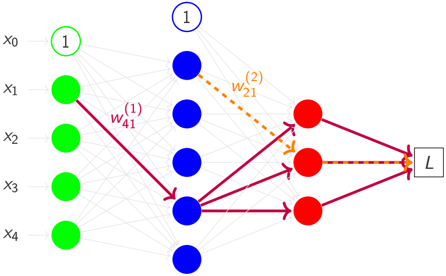
- The *multivariate* chain rule of differentiation states

$$\frac{df(u(x), v(x))}{dx} = \frac{\partial f}{\partial u}\frac{du}{dx} + \frac{\partial f}{\partial v}\frac{dv}{dx}$$



- The multivariate chain rule applied to compute derivatives w.r.t weights of hidden layers has a special name – *backpropagation*.

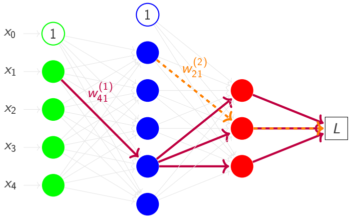
Backpropagation



- For the output layer weights

$$\frac{\partial L(y_k(a_k^{(2)}(w_{kj}^{(2)})))}{\partial w_{kj}^{(2)}} = \frac{\partial L}{\partial a_k^{(2)}} \frac{\partial a_k^{(2)}}{\partial w_{kj}^{(2)}} = \delta_k z_j$$

Backpropagation



- For the hidden layer weights, using the multivariate chain rule

$$\frac{\partial}{\partial w_{ji}^{(1)}} L(y_1(a_1^{(2)}(z_j(a_j^{(1)}(w_{ji}^{(1)})))), y_2(a_2^{(2)}(z_j(a_j^{(1)}(w_{ji}^{(1)})))), \ldots, y_k(a_k^{(2)}(z_j(a_j^{(1)}(w_{ji}^{(1)})))))$$

$$= \frac{\partial L}{\partial a_j^{(1)}} \frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}} = \underbrace{\sum_{k=1}^{K} \underbrace{\frac{\partial L}{\partial a_k^{(2)}}}_{\delta_k} \underbrace{\frac{\partial a_k^{(2)}}{\partial z_j}}_{w_{kj}^{(2)}} \underbrace{\frac{\partial z_j}{\partial a_j^{(1)}}}_{h'(a_j^{(1)})}}_{\frac{\partial L}{\partial a_j^{(1)}} = \delta_j} \underbrace{\frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}}}_{x_i} = \delta_j x_i$$

Backpropagation

- It is important to note that

$$\delta_j = h'(a_j) \sum_{k=1}^{K} \delta_k w_{kj}$$

  yields the error δ_j at hidden neuron j by *backpropagating* the errors δ_k from all output neurons that use the output of neuron j.
- More generally, compute error δ_j at a layer by *backpropagating* the errors δ_k from next layer.
- Hence the names *error backpropagation*, *backpropagation*, or simply *backprop*.
- Very useful machine learning technique that is *not limited to neural networks*.

Backpropagation

$$\delta_j^{(1)} = h'(a_j) \sum_{k=1}^{K} \delta_k^{(2)} w_{kj}$$

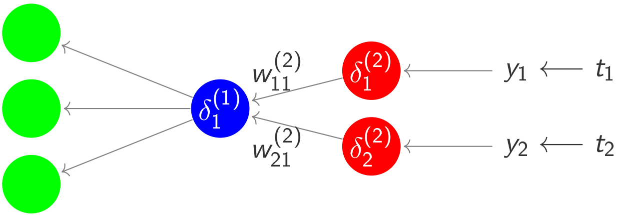


Figure: Visual representation of backpropagation of delta values of layer $l+1$ to compute delta values of layer l.

Backpropagation
Learning Algorithm

1. Forward propagate the input vector $\mathbf{x}_n$ to compute *and store* activations and outputs of every neuron in every layer.
2. Evaluate $\delta_k = \frac{\partial L_n}{\partial a_k}$ for every neuron in output layer.
3. Evaluate $\delta_j = \frac{\partial L_n}{\partial a_j}$ for every neuron in *every* hidden layer via backpropagation.

$$\delta_j = h'(a_j) \sum_{k=1}^{K} \delta_k w_{kj}$$

4. Compute derivative of each weight $\frac{\partial L_n}{\partial w}$ via $\delta \times$input.
5. Update each weight via gradient descent $w^{\tau+1} = w^{\tau} - \eta \frac{\partial L_n}{\partial w}$.

Summary

- Forward propagation from inputs to output can be modeled via matrix-vector products.
- Backpropagation is merely an implementation of the multivariate chain rule from calculus.