# CS-568 Deep Learning

**Nazar Khan**

PUCIT

Training Multilayer Perceptrons: Backpropagation

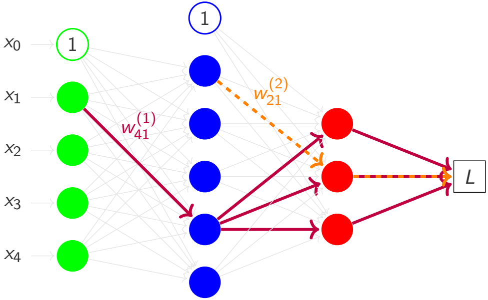
# Neural Networks for Regression
## *Gradients*

- 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

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

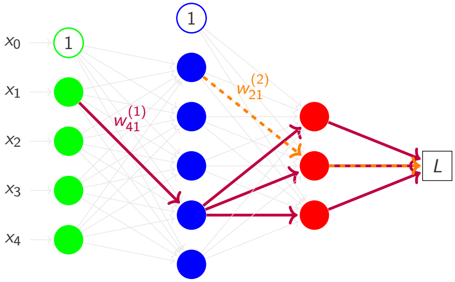
- Loss $L$ depends on sum of individual losses $L_n$.
- In the following, we will focus on loss $L_n$ for the $n$-th training sample.
- We will drop $n$ for notational clarity and refer to $L_n$ simply as $L$.

# 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: $L \leftarrow y_k \leftarrow a^{(2)}_k \leftarrow w^{(2)}_{kj}$.

$$L(y_k(a^{(2)}_k(w^{(2)}_{kj})))$$

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

$$L(\underbrace{y_1(a^{(2)}_1(z_j(a^{(1)}_j(w^{(1)}_{ji}))))}_{y_1(w^{(1)}_{ji})}, \underbrace{y_2(a^{(2)}_2(z_j(a^{(1)}_j(w^{(1)}_{ji}))))}_{y_2(w^{(1)}_{ji})}, \dots, \underbrace{y_k(a^{(2)}_k(z_j(a^{(1)}_j(w^{(1)}_{ji}))))}_{y_k(w^{(1)}_{ji})})$$

# Multivariate Chain Rule

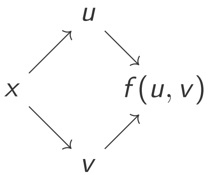
- The chain rule of differentiation states

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

$$x \longrightarrow u \longrightarrow f$$

- 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

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

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

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

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

- For each layer, notice the familiar form of gradient = error×input.

# Backpropagation

- It is important to note that

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

  yields the error $\delta_j$ at hidden neuron $j$ by *backpropagating* the errors $\delta_k$ from all output neurons that use the output of neuron $j$.
- More generally, compute error $\delta_j$ at a layer by *backpropagating* the errors $\delta_k$ 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

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

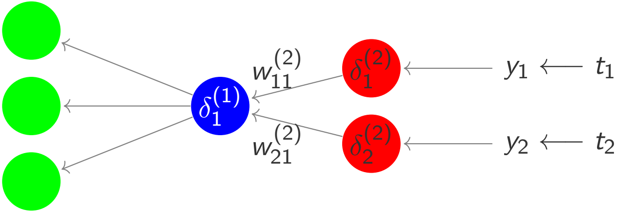



**Figure:** Visual representation of backpropagation of delta values of layer $l+1$ to compute delta values of layer $l$.

# Backpropagation

## *Learning Algorithm*

1. Forward propagate the input vector $\mathbf{x}_n$ to compute activations and outputs of every neuron in every layer.
2. Evaluate $\delta_k$ for every neuron in output layer.
3. Evaluate $\delta_j$ for every neuron in *every* hidden layer via backpropagation.
4. Compute derivative of each weight $\frac{\partial L_n}{\partial w}$ via $\delta \times$input.
5. Update each weight via gradient descent $w^{\tau+1} = w^{\tau} - \eta \frac{\partial L_n}{\partial w}$.

# Background Math
## *A $(-1, 1)$ sigmoidal function*

- Since range of logistic sigmoid $\sigma(a)$ is $(0, 1)$, we can obtain a function with $(-1, 1)$ range as $2\sigma(a) - 1$.
- Another related function with $(-1, 1)$ range is the tanh function.

$$\tanh(a) = 2\sigma(2a) - 1 = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

  where $\sigma$ is applied on $2a$.
- Preferred[1] over logistic sigmoid as activation function $h(a)$ of hidden neurons.
- Just like the logistic sigmoid, derivative of $\tanh(a)$ is simple: $1 - \tanh^2(a)$. (Prove it.)

[1] LeCun et al., 'Efficient backprop'.

# A Simple Example

- Two-layer MLP for multivariate regression from $\mathbb{R}^D \longrightarrow \mathbb{R}^K$.
- Linear outputs $y_k = a_k$ with half-SSE $L = \frac{1}{2}\sum_{k=1}^K (y_k - t_k)^2$.
- $M$ hidden neurons with $\tanh(\cdot)$ activation functions.

Forward propagation

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i$$

$$z_j = \tanh(a_j)$$

$$z_0 = 1$$

$$y_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j$$

$$\delta_k = y_k - t_k$$

Backpropagate

$$\delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{kj}^{(2)} \delta_k$$

- Compute derivatives $\frac{\partial L}{\partial w_{ji}^{(1)}} = \delta_j x_i$ and $\frac{\partial L}{\partial w_{kj}^{(2)}} = \delta_k z_j$.

# Backpropagation

## *Verifying Correctness*

- *Numerical derivatives* can be computed via finite *central differences*

$$\frac{\partial L_n}{\partial w_{ji}} = \frac{L_n(w_{ji} + \epsilon) - L_n(w_{ji} - \epsilon)}{2\epsilon} + O(\epsilon^2)$$

- *Analytical derivatives* computed via backpropagation **must be compared** with numerical derivatives for a few examples to verify correctness.
- Any implementation of analytical derivatives (not just backpropagation) must be compared with numerical derivatives.
- Notice that we could have avoided backpropagation and computed all required derivatives numerically.
  - But cost of numerical differentiation is $O(W^2)$ while that of backpropagation is $O(W)$ where $W$ is the total number of weights (and biases) in the network. (Why?)

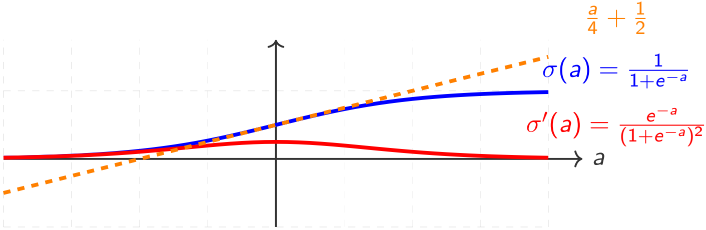
# Neural Network training finds local minimum

- For optimisation, we notice that $\mathbf{w}^*$ must be a *stationary point* of $E(\mathbf{w})$.
  - Minimum, maximum, or saddle point.
  - A saddle point is where gradient vanishes but point is not an extremum (Example).
- The goal in neural network minimisation is to find a local minimum.
- A global minimum, *even if found*, cannot be verified as globally minimum.
- Due to symmetry, there are multiple equivalent local minima. Reaching *any suitable* local minimum is the goal of neural network optimisation.
- Since there are no analytical solutions for $\mathbf{w}^*$, we use iterative, numerical procedures.

# Optimisation Options

- Options for iterative optimisation
  - Online methods
    - Stochastic gradient descent
    - Stochastic gradient descent using mini-batches
  - Batch methods
    - Batch gradient descent
    - Conjugate gradient descent
    - Quasi-Newton methods
- Online methods
  - converge faster since parameter updates are more frequent, and
  - have greater chance of escaping local minima because stationary point w.r.t to whole data set will generally not be a stationary point w.r.t an individual data point.
- Batch methods: Conjugate gradient descent and quasi-Newton methods
  - are more robust and faster than batch gradient descent, and
  - decrease the error function at each iteration until arriving at a minimum.

# Problems with sigmoidal neurons




- For large $|a|$, sigmoid value approaches either 0 or 1. This is called *saturation*.
- When the sigmoid saturates, the gradient approaches zero.
- Neurons with sigmoidal activations stop learning when they saturate.
- When they are not saturated, they are almost linear.
- There is another reason for the gradient to approach zero during backpropagation.

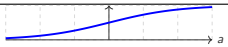
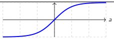
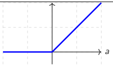
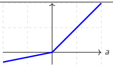
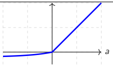
# Vanishing Gradients

- Notice that gradient of the sigmoid is always between 0 and $\frac{1}{4}$.
- Now consider the backpropagation equation.

$$\delta_j = \underbrace{h'(a_j)}_{\leq \frac{1}{4}} \sum_{k=1}^{K} w_{kj}\delta_k$$

where $\delta_k$ will also contain *at least* one factor of $\leq \frac{1}{4}$.
- This means that values of $\delta_j$ keep getting smaller as we backpropagate towards the early layers.
- Since gradient $= \delta \times$input, the gradients also keep getting smaller for the earlier layers. Known as the *vanishing gradients* problem.
- *Therefore, while the network might be deep, learning will not be deep.*

# Better Activation Functions

| Name | Formula | Plot | Derivative | Comments |
|---|---|---|---|---|
| Logistic sigmoid | $\frac{1}{1+e^{-a}}$ |  | $f(a)(1-f(a))$ | Vanishing gradients |
| Hyperbolic tangent | $\tanh(a)$ |  | $1-\tanh^2(a)$ | Vanishing gradients |
| Rectified Linear Unit (ReLU) | $\begin{cases} a & \text{if } a>0 \\ 0 & \text{if } a\leq 0 \end{cases}$ |  | $\begin{cases} 1 \\ 0 \end{cases}$ | Dead neurons. Sparsity. |
| Leaky ReLU | $\begin{cases} a & \text{if } a>0 \\ ka & \text{if } a\leq 0 \end{cases}$ |  | $\begin{cases} 1 \\ k \end{cases}$ | $0<k<1$ |
| Exponential Linear Unit (ELU) | $\begin{cases} a & \text{if } a>0 \\ k(e^a-1) & \text{if } a\leq 0 \end{cases}$ |  | $\begin{cases} 1 \\ f(a)-k \end{cases}$ | $k>0$. |

- Saturated sigmoidal neurons stop learning. Piecewise-linear units keep learning by avoiding saturation.
- ELU leads to better accuracy and faster training.
- *Take home message*: Use a member of the LU family. They avoid *i*) saturation and *ii*) the vanishing gradient problem.