# Backpropagation in MLP

Tirtharaj Dash

Dept. of CS & IS and APPCAIR
BITS Pilani, Goa Campus

September 04, 2021

# Where we are:

Summary of the previous lecture:

- Deep Feedforward Nets and MLPs
- Forward computation in MLPs
- Role of activation functions
- Re-doing XOR with ReLU activation
- Role of hidden representations

Today, we will know:

- How the error is backpropagated via usage of chain rule in calculus.
- That is, we will derive the steps in the backpropagation procedure.
- This will help us understand the tensor-level operation that I will show in the next lecture.

# Introduction I

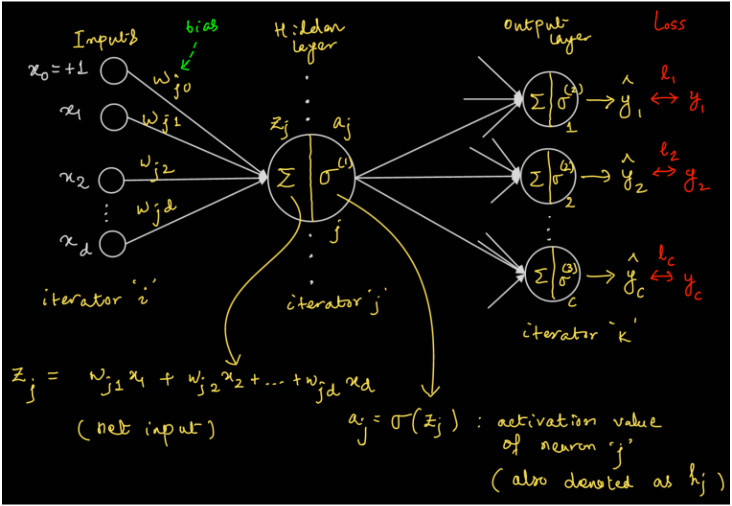
- We are concerned with MLPs (deep fully-connected feedforward neural networks).
- The parameters of an MLP are its synaptic strenghts or interconnection weights.
- There are two main computations in an MLP:
  - Forward: Activations flow from inputs to output layer (leading upto: the calculation loss)
  - Backward: Gradient of the loss (or error) is backpropagated from the output layer towards input via the hidden layers (leading upto: the update of weights)

# Introduction II

- The parameters (weights) play a significant role in forward and backward computation.
- The method adopted to update the parameters in a deep network is known as the Backpropagation procedure [1].

[1] D.E. Rumelhart, G. Hinton, R.J. Williams (1986): Learning representations by back-propagating errors. Nature, 323(6088), 533–536.

- A portion of a 1-hidden layer MLP showing 1 hidden unit:



# Notations I

We will adopt the following for our derivations:

- For convenience, we will consider a 1-hidden layer MLP.
- $L$ denotes the loss function: At this point, we will not define what $L$ is, but, it is some function $L(\mathbf{y}, \hat{\mathbf{y}})$.
- $\mathbf{y}$: vector of target outputs ($y_k$ denotes the output of $k$th output neuron).
- $\hat{\mathbf{y}}$: vector of computed outputs
- There are $c$ output neurons (one neuron for each target). For instance:
  - In binary classification, $c = 1$
  - In multiclass classification, $c =$ no. of classes
  - In regression, $c = 1$

# Notations II

- Some examples of $L(\mathbf{y}, \hat{\mathbf{y}})$:
  - Binary cross-entropy: $L = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$
  - Cross-entropy: $L = -\sum_{k=1}^{c} y_k \log \hat{y}_k$
  - SSE: $L = \frac{1}{2} \sum_{k=1}^{c} (y_k - \hat{y}_k)^2$ (for a single instance)
  - MSE for a batch of instances: $L = \frac{1}{2m} \sum_{i=1}^{m} \sum_{k=1}^{c} (y_k^{(i)} - \hat{y}_k^{(i)})^2$

# Notations III

- The subsctipt $i$ corresponds to the inputs.
- There are $d$ input units, each corresponding to a feature vector in $\mathbf{x}$.
- The subscript $j$ corresponds to the hidden layer.
- There are $m$ units in the hidden layer.
- The subscript $k$ corresponds to the output layer.
- There are $c$ units in the output layer.

# Notations IV

- Weights (see the footnote below):
  - $w_{ji}$: weight parameter for the connection from $i$th input to $j$th neuron in the hidden layer.
  - $w_{kj}$: weight parameter for the connection from $j$th hidden neuron to $k$th neuron in the output layer.

Note: In our forward propagation lecture (also in our Textbook 1), we were using a different notation. That is, $w_{ij}$ and $w_{jk}$ respectively. Don't get confused.

# Notations V

- $z$ denotes the net input to any neuron $j$. For example,
  - $z_j$: for the hidden neuron $j$
  - $z_k$: for the output neuron $k$
- $a$ denotes the activation value. That is: $a = \sigma(z)$, where $\sigma$ is some activation function (preferrably, continuous).
  - $a_j$: for the hidden neuron $j$
  - $a_k$: for the output neuron $k$. But, this is equal to $\hat{y}_k$.

# GD and Backprop I

Gradient Descent (GD) on loss:

- The update to any parameter $w_{ij}$ at any iteration $t$ is carried out using the step

$$w_{ij}^{(t)} \leftarrow w_{ij}^{(t-1)} - \eta \left. \frac{\partial L}{\partial w_{ij}} \right|_{\text{at } w_{ij}=w_{ij}^{(t-1)}}$$

- Next, we will see how to derive $\frac{\partial L}{\partial w_{ij}}$ for the hidden layer parameters and the output layer parameters.

Note: In this particular slide and in the next one the subscript in $w$ (i.e. $i$ and $j$) are just local variables. They represent any weight in a network.

# GD and Backprop II

What exactly is $\frac{\partial L}{\partial w_{ij}}$?

- This is calculated using chain-rule of (partial) derivative. This turns out to be:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial \text{ activation}} \frac{\partial \text{ activation}}{\partial \text{ net-input}} \frac{\partial \text{ net-input}}{\partial w_{ij}}$$

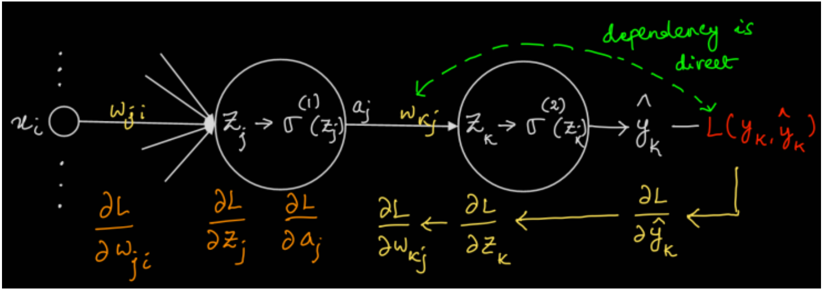
- or,

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial a} \frac{\partial a}{\partial z} \frac{\partial z}{\partial w_{ij}}$$

Ignoring the subscript on $a$ and $z$ here. In the next slides, these will be clearer.

# GD and Backprop III

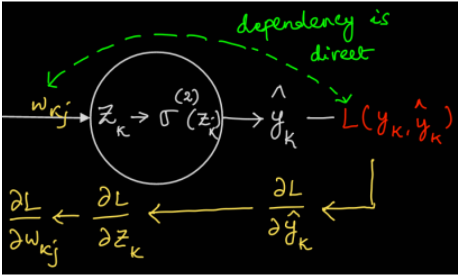
A figure showing flow of gradient from $L$ towards the inputs:



# Gradients at output layer I

We want to compute $\frac{\partial L}{\partial w_{kj}}$ for all $j$ and $k$:

- Let us see the dependecy graph (chain) first:




- Now, the expression for the derivative based on the above chain:

$$\frac{\partial L}{\partial w_{kj}} = \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_k} \frac{\partial z_k}{\partial w_{kj}}$$

- We want to compute these individual quantities. See next.

# Gradients at output layer II

- Derivative of the loss w.r.t. the output (activation), $\frac{\partial L}{\partial \hat{y}_k}$:
  - Since we have not defined our function $L$: Let us keep it as it is, and call it: $L'$
  - If $L$ is SSE, this would be:

$$\frac{\partial L}{\partial w_{kj}} = -(y_k - \hat{y}_k)$$

# Gradients at output layer III

- Derivative of the output (activation) w.r.t. the net input, $\frac{\partial \hat{y}_k}{\partial z_k}$:
  - We know that $\hat{y}_k = \sigma^{(2)}(z_k)$, where $\sigma^{(2)}$ is the output layer activation function, we have then:

$$\frac{\partial \hat{y}_k}{\partial z_k} = \frac{\partial \sigma^{(2)}(z_k)}{\partial z_k} = \sigma^{(2)'}(z_k)$$

# Gradients at output layer IV

- Derivative of the net input w.r.t. the weight, $\frac{\partial z_k}{\partial w_{kj}}$:
  - We know that $z_k = \sum_{j=1}^{m} w_{kj} a_j$ (for simplicity we are not using the bias term; however, including this is staraightforward: the summation will start at $j = 0$). Now, the derivative is:

$$\frac{\partial \hat{y}_k}{\partial z_k} = a_j$$

# Gradients at output layer V

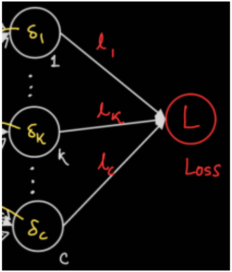
- Substituting the quantities obtained earlier we get our $\frac{\partial L}{\partial w_{kj}}$ expression:

$$\frac{\partial L}{\partial w_{kj}} = L'(\mathbf{y}, \hat{\mathbf{y}})\sigma^{(2)'}(z_k)a_j$$

- The blue expression is called the *local gradient* computed at the output neuron $k$, and we denote it as $\delta_k$.

# Gradients at output layer VI

- A simple diagram shown as:

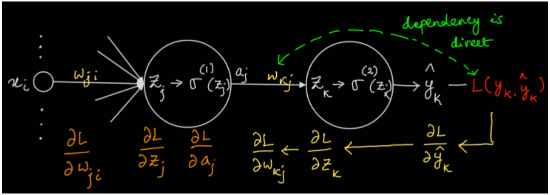



$$\delta_k = L'(\mathbf{y}, \hat{\mathbf{y}})\sigma^{(2)'}(z_k)$$

- Notice that we do not need to re-calculate this $\delta_k$ while calculating some other weight (i.e. different $j$) for the same $k$. We can re-use the computed $\delta_k$.
- Also, recall that $a_j$ was already computed during our forward computation.

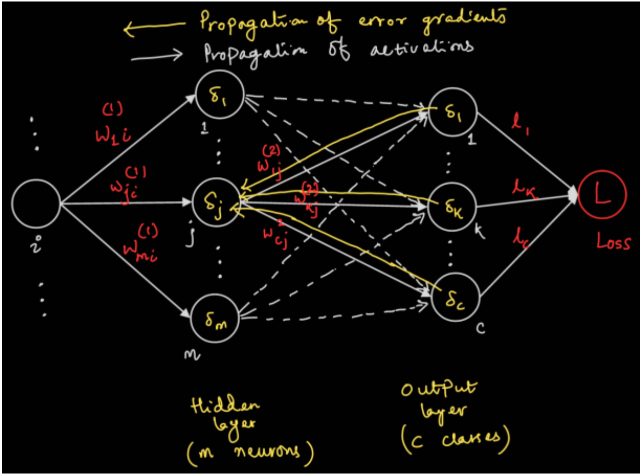
We want to compute $\frac{\partial L}{\partial w_{ji}}$ for all $i$ and $j$:

- Let us first see the dependency graph:




The dependency graph does not show that the weight $w_{ji}$ leads to error at all the neurons at the next layer (outputs). We show this next.

- All the paths from $w_{ji}$ to $L$:




The yellow-lines show the paths.

# Gradients at hidden layers III

- What is this really?
  - $L$ depends on $\hat{\mathbf{y}}$
  - $\hat{\mathbf{y}}$ depends on $a_j$
- For the hidden unit $j$, we know these following from forward computation:

$$z_j = \sum_{i=1}^{d} w_{ji} x_i$$

and

$$a_j = \sigma^{(1)}(z_j)$$

- For the output unit $k$, we know that:

$$z_k = \sum_{j=1}^{m} w_{kj} a_j$$

# Gradients at hidden layers IV

- In the end, $\frac{\partial L}{\partial w_{ji}}$ turns out to be:

$$
\begin{aligned}
\frac{\partial L}{\partial w_{ji}} &= \left( \sum_{k=1}^{c} \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_k} \frac{\partial z_k}{\partial a_j} \right) \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ji}} \\
&= \left( \sum_{k=1}^{c} \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_k} w_{kj} \right) \sigma^{(1)'}(z_j) x_i \\
&= \left( \sum_{k=1}^{c} L'(\mathbf{y}, \hat{\mathbf{y}}) \sigma^{(2)'}(z_k) w_{kj} \right) \sigma^{(1)'}(z_j) x_i \\
&= \left( \sum_{k=1}^{c} \delta_k w_{kj} \right) \sigma^{(1)'}(z_j) x_i \\
&= \delta_j x_i
\end{aligned}
$$

# Gradients at hidden layers V

- $\delta_j$ is the local gradient computed at the hidden unit $j$:

$$\delta_j = \left( \sum_{k=1}^{c} \delta_k w_{kj} \right) \sigma^{(1)'}(z_j)$$

- Notice the first term in parenthesis. This is saying weigh all the local gradients computed at the output layer by the corresponding interconnection weights.

# Generalisation to Deeper Nets I

More on $\delta_j$ computed in the last slide:

- This is a beautiful finding, which takes all our worries of computing gradients for deeper network.
- This allows us to generalise to any layers, since we are travelling from $L$ towards the inputs, on the way, we are calculating these local gradients.

# Generalisation to Deeper Nets II

- Then the local gradient of any neuron $j$ at any hidden layer $\ell$ (where $\ell_{max} > \ell \geq 1$) can be easily computed as:

$$\delta_j^{(\ell)} = \left( \sum_{k=1}^{m^{(\ell+1)}} \delta_k^{(\ell+1)} w_{kj}^{(\ell+1)} \right) \sigma^{(\ell)\prime}(z_j^{(\ell)})$$

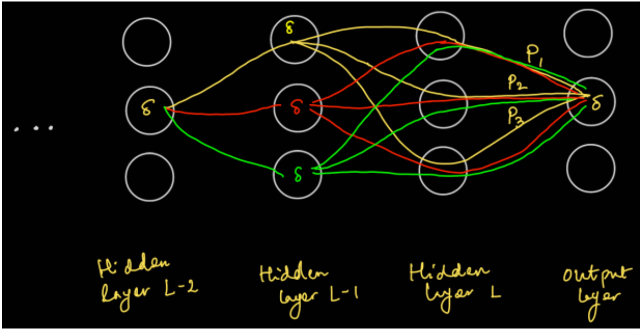
- Then the gradient of the loss $L$ w.r.t to any weight at layer $\ell$ is simply:

$$\frac{\partial L}{\partial w_{ji}^{(\ell)}} = \delta^{(\ell)} a_i^{(\ell-1)}$$

Notations: $(\ell)$ is used to denote a layer index; $\ell = 0$ corresponds to the inputs; $\ell_{max}$ corresponds to the output layer; $m^{(\ell)}$ denotes the number of neurons (units) at layer $\ell$; $\mathbf{a}^{(0)} = \mathbf{x}$.

# Generalisation to Deeper Nets III

- This can be clearly shown as a bunch of computational (flow) paths in the network:



# Parameter updates I

- We have now computed the derivatives of $L$ w.r.t. all the parameters in the network.
- The update to the parameters based on the GD procedure for any network layer $\ell$ is then:

$$w_{ij}^{(\ell)} = w_{ij}^{(\ell)} - \eta \frac{\partial L}{\partial w_{ij}^{(\ell)}}$$

- That is, the gradient of $L$ w.r.t. all the parameters of the network have to be computed **first** and then the update is done. **Otherwise** the computation of the local gradients will be wrong.

# Parameter updates II

- Common mistake:

  For layer $\ell \in \{1, \ldots, \ell_{max}\}$:

  Compute gradient of $L$ w.r.t. weights at layer $\ell$

  Update the weights at layer $\ell$

- Correct (writing again):

  For layer $\ell \in \{1, \ldots, \ell_{max}\}$:

  Compute gradient of $L$ w.r.t. weights at layer $\ell$

  For layer $\ell \in \{1, \ldots, \ell_{max}\}$:

  Update the weights at layer $\ell$

# Parameter updates III

- The derivations so far has been to show how the backpropagation procedure operates at the scalar-level.
- However, you will notice that many computations such as the local gradints at neurons at any layer can be computed in parallel (i.e. computation of $\delta_i^{(\ell)}$ and $\delta_j^{(\ell)}$ is independent) allowing tensor-level computation.
- Further, the quantities computed at the forward propagation such as $a_j$s can be reused for backpropagation. This requires some kind of *recor keeping*. This results in requirement of more memory storage for backpropagation.

# Homework

These two questions will make you comfortable with the derivations we did here:

1. Derive the complete backpropagation procedure for a classification problem with $c$ classes. Here, $L$ is the cross-entropy loss.
2. Derive the complete backpropagation procedure for a regression problem. Here $L$ is the SSE.
3. What happens to the derivations if we have a batch-size greater than 1 in our training? What would change in our derivations?