Neural networks

Fraida Fund

Contents

In this lecture

- Neural network
- Structure of a neural network
- Training a neural network

Math prerequisites for this lecture: You should know about:

- derivatives and especially the chain rule (Appendix C in Boyd and Vandenberghe)

From linear to non-linear

Representation as a computational graph

Let's represent the linear regression and logistic regression models using a computational graph.

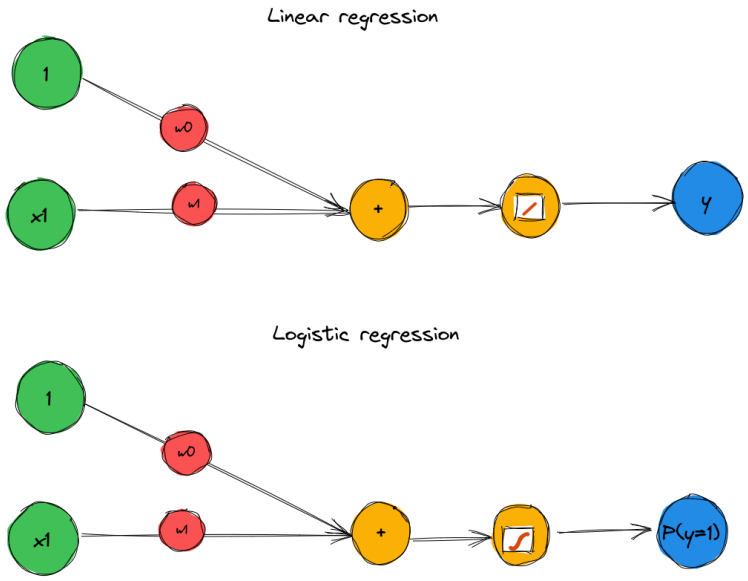

Figure 1: Regression and classification models.

We can use also do a linear regression or logistic regression with a basis function transformation applied to the data first. Here, we have one "transformation" node for each basis function, and then the output of those "transformation" nodes become the input to the logistic regression (or linear regression).

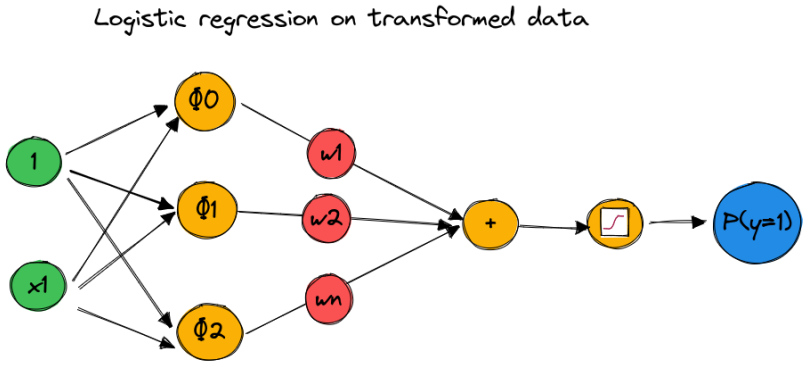

Figure 2: With a basis function transformation.

We can also represent the SVM with a linear or non-linear kernel using a computational graph.

Here, we have one "transformation node" for each training sample! (The "transformation" is the kernel function, which is computed over the input sample and the training sample).

Then the weighted sum of those nodes (weighted by α_i, which is learned by the SVM, and which is zero for every non-support vector training sample and non-zero for every support vector training sample) is used to compute the class label.

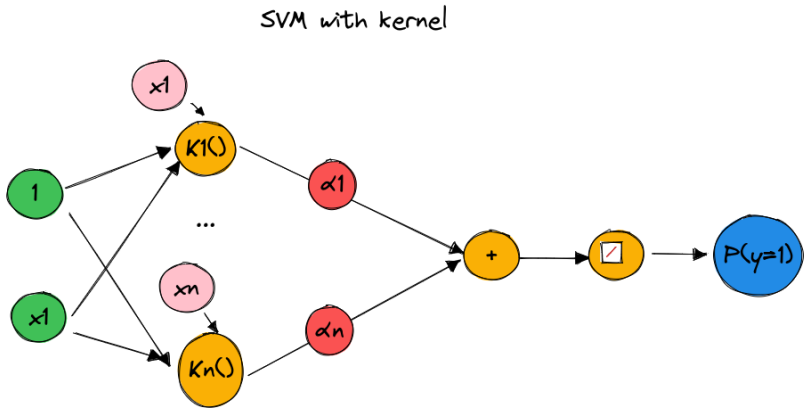

Figure 3: SVM computational graph.

In those regression and classification models, we use a fixed basis function to transform features. We only learned the weights to map the transformed features to a continuous output (regression) or to a class label (classification).

Would it be possible to also learn the first part - the mapping of the features to a transformed feature space?

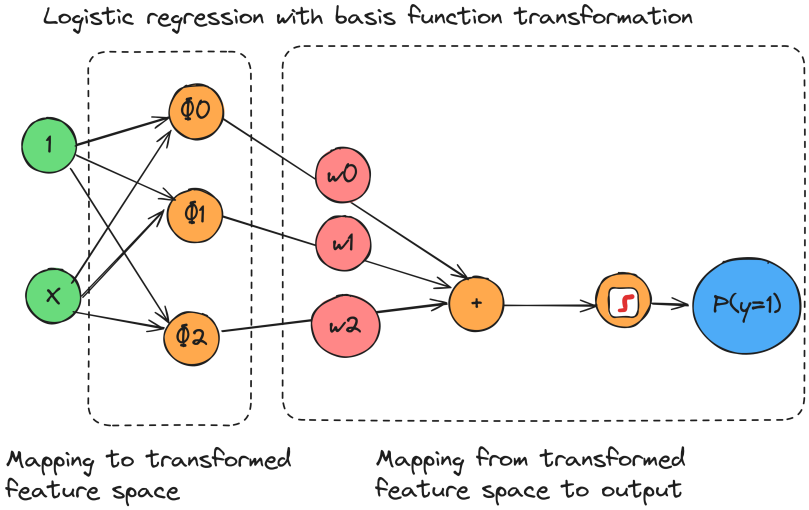

Figure 4: Can we learn this mapping to transformed feature space?

Example: synthetic data

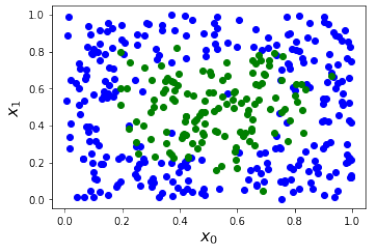

Figure 5: Example via Sundeep Rangan

Model of example two-stage network (1)

First step (*hidden layer*):

- Take $N_H = 4$ "logistic regression" nodes.
- Use $\mathbf{x}$ as input to each node.
- At each node m, first compute: $z_{H,m} = \mathbf{w}_{H,m}^T \mathbf{x}$
- Then, at each node, apply a sigmoid: $u_{H,m} = g_H(z_{H,m}) = \frac{1}{1+e^{-z_{H,m}}}$

Note: assume a 1s column was added to the data matrix, so we don't need a separate intercept term.

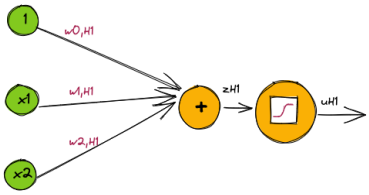

Figure 6: Computation at one hidden node.

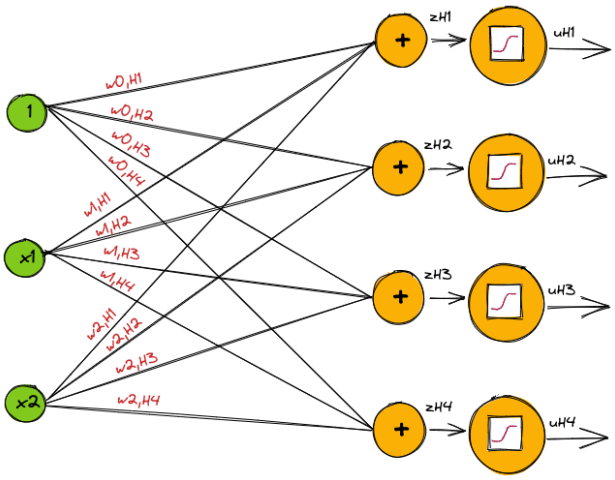

Figure 7: Computation at four hidden nodes.

At this point, we have some representation of the data in $\mathbb{R}^4$.

Model of example two-stage network (2)

Second step (*output layer*):

- At output node, first compute: $z_O = \mathbf{w}_O^T[1, \mathbf{u}_H]$
- Then, compute: $u_O = g_O(z_O) = \frac{1}{1+e^{-z_O}}$
- (Not in the graph): apply a threshold to get $\hat{y}$

Notes:

- we use the output of the previous layer as input to this layer
- as with the first layer, we add a 1s column to the input, to take care of the intercept term.

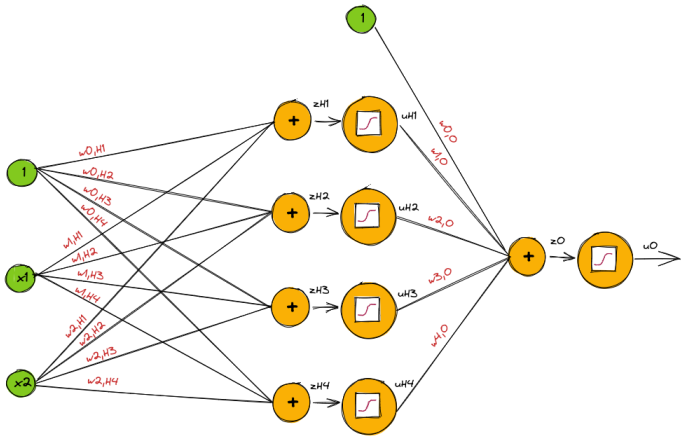

Figure 8: Two-stage network.

What does the output look like (over the feature space) at each node?

Example: output of each hidden node

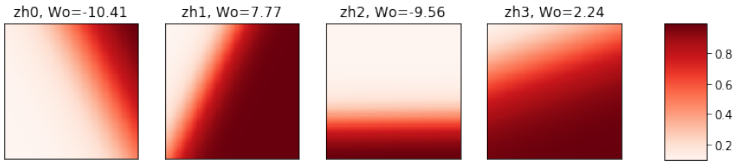

Figure 9: Example via Sundeep Rangan

Example: output of output node

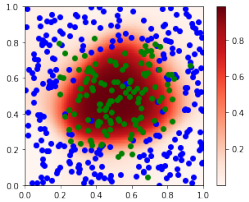

Figure 10: Via Sundeep Rangan

Matrix form of two stage network

- Hidden layer: $\mathbf{z}_H = \mathbf{W}_H^T \mathbf{x}, \quad \mathbf{u}_H = g_H(\mathbf{z}_H)$
- Output layer: $z_O = \mathbf{W}_O^T [1, \mathbf{u}_H], \quad u_O = g_O(\mathbf{z}_O)$

Neural networks

Terminology

- **Hidden variables**: the variables $\mathbf{z}_H, \mathbf{u}_H$, which are not directly observed.
- **Hidden units**: the nodes that compute the hidden variables.
- **Activation function**: the function $g(z)$
- **Output units**: the node(s) that compute z_O.

Setting up a neural network - givens

For a particular problem, these are "given":

- the number of inputs
- the number of outputs
- the activation function to use at the output
- the loss function

The number of inputs comes from the data - what is the size of each training sample?

The number of outputs is dictated by the type of problem -

- binary classification: we need to predict $P(y = 1)$ which is a single quantity, so we have one output unit.
- multi-class classification: we will predict $P(y = k)$ for each class k, so we need an output unit for each class.
- regression: if we need to predict a single target, we will have one output unit. If we need to predict multiple values in the same problem (vector output), we will have as many output units as there are values in the output.

Similarly, the activation function at the output unit and the loss function are dictated by the problem!

Binary classification...

For binary classification, $y \in [0, 1]$:

- Use sigmoid activation, output will be: $u_O = P(y = 1|x) = \frac{1}{1+e^{-z_O}}$
- u_O is scalar - need one output node
- Use binary cross entropy loss:

$$L(\mathbf{W}) = \sum_{i=1}^{n} -y_i z_{Oi} + \ln(1 + e^{z_{Oi}})$$

Then you select the predicted label by applying a threshold to the output u_O.

The mapping from transformed feature space to output is just like a logistic regression - we haven't changed *that* part!

Multi-class classification...

For multi-class classification, $y = 1, \ldots, K$:

- Use softmax activation, output will be: $u_{O,k} = P(y = k|x) = \frac{e^{z_{O,k}}}{\sum_{\ell=1}^{K} e^{z_\ell}}$
- u_O is vector $[u_0, \ldots, u_K]$ - need K output nodes
- Use categorical cross entropy loss:

$$L(\mathbf{W}) = \sum_{i=1}^{n} \left[\ln \left(\sum_k e^{z_{Oik}} \right) - \sum_k r_{ik} z_{Oik} \right]$$

Then you can select predicted label by $\hat{y} = \text{argmax}_k \, u_{O,k}$

Regression with one output...

For regression, $y \in R^1$:

- Use linear activation, output will be: $u_O = z_O$
- u_O is scalar - need one output node
- Use L2 loss:

$$L(\mathbf{W}) = \sum_{i=1}^{n} (y_i - z_{Oi})^2$$

Regression with multiple outputs...

For regression, $y \in R^K$:

- Use linear activation, output will be: $u_{O,k} = z_{O,k}$
- u_O is vector $[u_0, \ldots, u_K]$ - need K output nodes
- Use vector L2 loss:

$$L(\mathbf{W}) = \sum_{i=1}^{n} \sum_{k=1}^{K} (y_{ik} - z_{Oik})^2$$

Setting up a neural network - decisions

We still need to decide:

- the number of hidden units
- the activation function to use at hidden units

Dimension (1)

- N_I = input dimension, number of features
- N_H = number of hidden units, you decide!
- N_O = output dimension, number of outputs

Dimension (2)

Parameter	Symbol	Number of parameters
Hidden layer: weights	W_H	$N_H(N_I+1)$
Output layer: weights	W_O	$N_O(N_H+1)$
Total		$N_H(N_I+1)+N_O(N_H+1)$

Activation functions at hidden layer: identity?

- Suppose we use $g(z)=z$ (identity function) as activation function throughout the network.
- The network can only achieve linear decision boundary!
- To get non-linear decision boundary, need non-linear activation functions.

Universal approximation theorem: under certain conditions, with enough (finite) hidden nodes, can approximate *any* continuous real-valued function, to any degree of precision. But only with non-linear decision boundary! (See this post for a convincing demonstration.)

(The more hidden units we have, the more complex a function we can represent.)

By scaling, shifting, and adding a bunch of "step" or "step-like" functions, you can approximate a complicated function. What step-like function can you use?

Activation functions at hidden layer: binary step

- Not differentiable at $x=0$, has 0 derivative everywhere else.
- Not useful for gradient-based optimization methods.

Activation functions at hidden layer: some choices

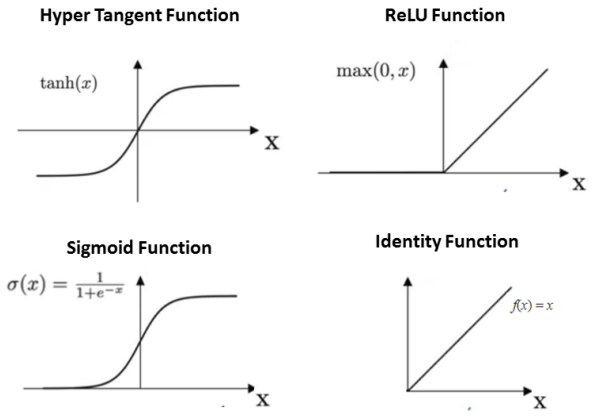

Figure 11: Most common activation functions

What do they have in common?

- Differentiable (at least from one side)
- Non-linear (except for the linear one, which is only used as the output function for a regression)

Neural network - summary

Things that are "given"

For a particular problem, these are "given":

- the number of inputs
- the number of outputs
- the activation function to use at the output
- the loss function

Things that we decide

We still need to decide:

- the number of hidden units
- the activation function to use at hidden units

Training the network

- Still need to find the $\mathbf{W}$ that minimizes $L(\mathbf{W})$.
- How?

Backpropagation

How to compute gradients?

- Gradient descent requires computation of the gradient $\nabla L(\mathbf{W})$
- Backpropagation is key to efficient computation of gradients

We need to compute the gradient of the loss function with respect to *every* weight, and there are $N_H(N_I + 1) + N_O(N_H + 1)$ weights!

The key to efficient computation will be:

- saving all the intermediate (hidden) variables on the forward pass, to reuse in the computation of gradients
- computing the gradients in a *backwards* pass - going from output to input, and accumulating local derivatives along the path (you'll see!)

Composite functions and computational graphs

Suppose we have a composite function $f(g(h(x)))$

We can represent it as a computational graph, where each connection is an input and each node performs a function or operation:

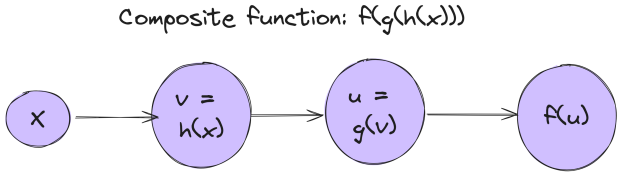

Figure 12: Composite function.

Forward pass on computational graph

To compute the output $f(g(h(x)))$, we do a *forward pass* on the computational graph:

- Compute $v = h(x)$
- Compute $u = g(v)$
- Compute $f(u)$

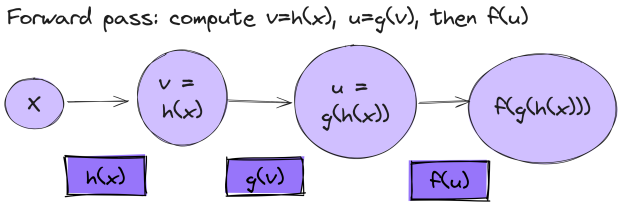

Figure 13: Forward pass.

Note that we *accumulate* results in the forward direction - at each node, we use the output of the previous node, which depends on the output of *all* the previous nodes. But, we don't need to repeat the steps of *all* the previous nodes each time, since the output is "accumulated" forward.

Derivative of composite function

Suppose we need to compute the derivative of the composite function $f(g(h(x)))$ with respect to x.

We will use the chain rule:

$$\frac{df}{dx} = \frac{df}{du}\frac{dg}{dv}\frac{dh}{dx}$$

Backward pass on computational graph

We can compute this chain rule derivative by doing a *backward pass* on the computational graph:

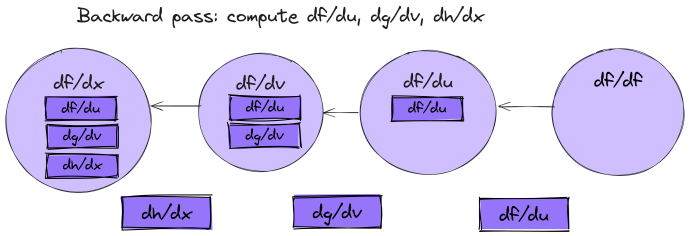

Figure 14: Backward pass.

As in the forward pass, in the backward pass, we do a "local" operation at each node: the "local" derivative of each node with respect to its inputs (shown in rectangles on each edge).

As in the forward pass, we *accumulate* results, but now in the backward direction. At each node, the derivative of the output with respect to the value computed at that node is:

- The product of all the "local" derivatives along the path between the node and the output.
- or equivalently, the product of the derivative at the previous node in the backward pass, and the "local" derivative along the path from that node.

For example: when we compute $\frac{df}{dx}$ at the last "stop" along the backwards pass, we don't need to compute all the parts of $\frac{df}{du}\frac{dg}{dv}\frac{du}{dx}$ *again*. We just need:

- compute "local" gradient $\frac{dh}{dx}$
- and multiply it by the "accumulated" gradient computed at the previous node, $\frac{df}{dv}$

This seems obvious... but when we apply it to a neural network, we will see why it is so important.

Neural network computational graph

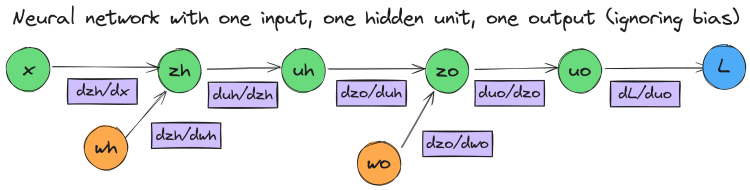

Figure 15: Neural network as a computational graph.

What about when we have multiple inputs, multiple hidden units?

Backpropagation error: definition

Denote the backpropagation error of node j as

$$\delta_j = \frac{\partial L}{\partial z_j}$$

the derivative of the loss function, with respect to the input to the activation function at that node.

Spoiler: this δ_j is going to be the "accumulated" part of the derivative.

Output unit: backpropagation error (accumulated)

Generally, for output unit j:

$$\delta_j = \frac{\partial L}{\partial z_j} = \frac{\partial L}{\partial u_j}\frac{\partial u_j}{\partial z_j}$$

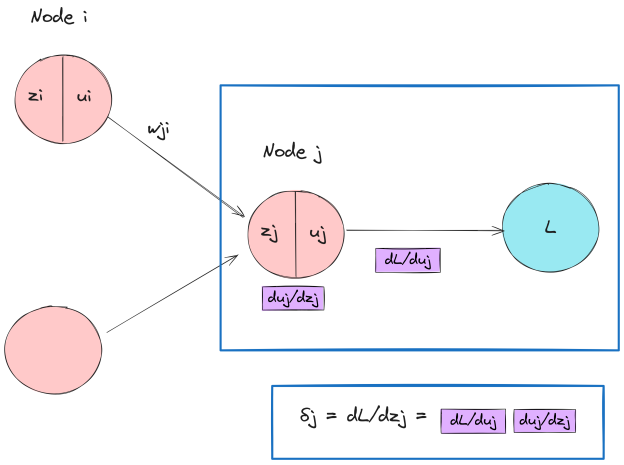

Figure 16: Computing backpropagation error at output unit.

For example, in a regression network:

$$L = \frac{1}{2}\sum_n (y_n - z_{O,n})^2$$

Then $\delta_O = \frac{\partial L}{\partial z_O} = -(y - z_O)$.

Output unit: derivative vs input weights (local)

- At a node j, $z_j = \sum_i w_{j,i} u_i = w_{j,i} u_i + \ldots$ (sum over inputs to the node)
- When taking $\frac{\partial z_j}{\partial w_{j,i}}$ the only term left is $w_{j,i} u_i$
- So $\frac{\partial z_j}{\partial w_{j,i}} = u_i$

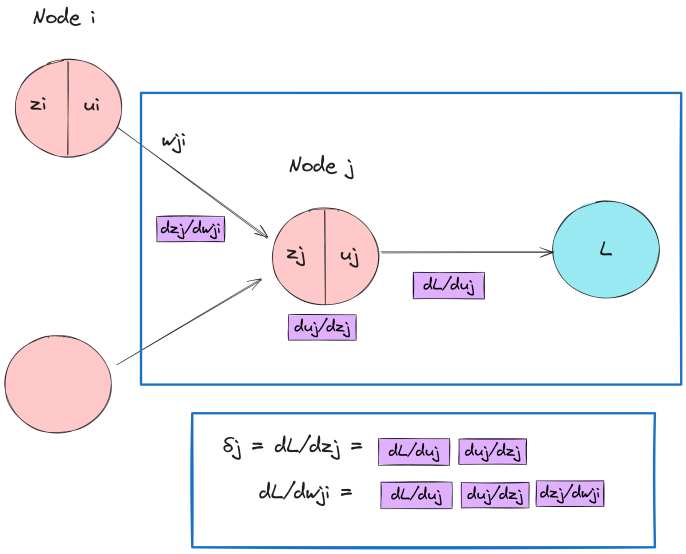

Figure 17: Computing gradient with respect to weight at output unit.

The derivative of the loss with respect to a weight $w_{j,i}$ input to the node, $\frac{\partial L}{\partial w_{j,i}}$, is the product of:

- $\delta_j = \frac{\partial L}{\partial z_j}$ (the "accumulated" part)
- $u_i = \frac{\partial z_j}{\partial w_{j,i}}$ (the "local" part)

so finally, $\frac{\partial L}{\partial w_{j,i}} = \delta_j u_i$.

(We save the computations of all the u_i values from the forward pass, so that we can reuse them for backpropagation.)

Hidden unit: backpropagation error (accumulated)

Sum the accumulated gradient along output paths:

$$\begin{aligned}\delta_j &= \frac{\partial L}{\partial z_j} \\ &= \sum_k \frac{\partial L}{\partial z_k}\frac{\partial z_k}{\partial z_j} \\ &= \sum_k \delta_k \frac{\partial z_k}{\partial z_j} \\ &= \sum_k \delta_k w_{k,j} g_j'(z_j)\end{aligned}$$

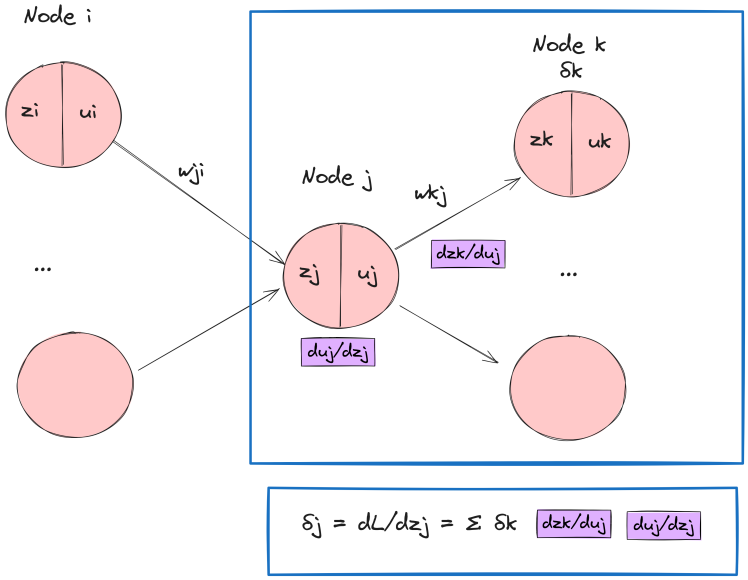

Figure 18: Computing backpropagation error at hidden unit.

Note: since we move from the output end of the network toward its input, we have already accumulated δ_k when we "visited" node k. So we the "new" computation is just $\frac{\partial z_k}{\partial z_j}$.

We compute the next "accumulated" gradient using the previous "accumulated" gradient and a "local" derivative.

Since

$$z_k = \sum_l w_{k,l} u_l$$

(sum over inputs to node k), but for the derivative with respect to z_j the only term left is $w_{k,j}u_j$. So,

$$\begin{aligned}\frac{\partial z_k}{\partial z_j} &= \frac{\partial}{\partial z_j} w_{k,j} u_j \\ &= \frac{\partial}{\partial z_j} w_{k,j} g_j(z_j) \\ &= w_{k,j} g'_j(z_j)\end{aligned}$$

where $g'_j()$ is the derivative of the activation function. (We save z_j from the forward pass, so we can reuse it here to compute $g'_j(z_j)$.)

Hidden unit: derivative vs input weights (local)

Same as output unit - $\frac{\partial z_j}{\partial w_{j,i}} = u_i$

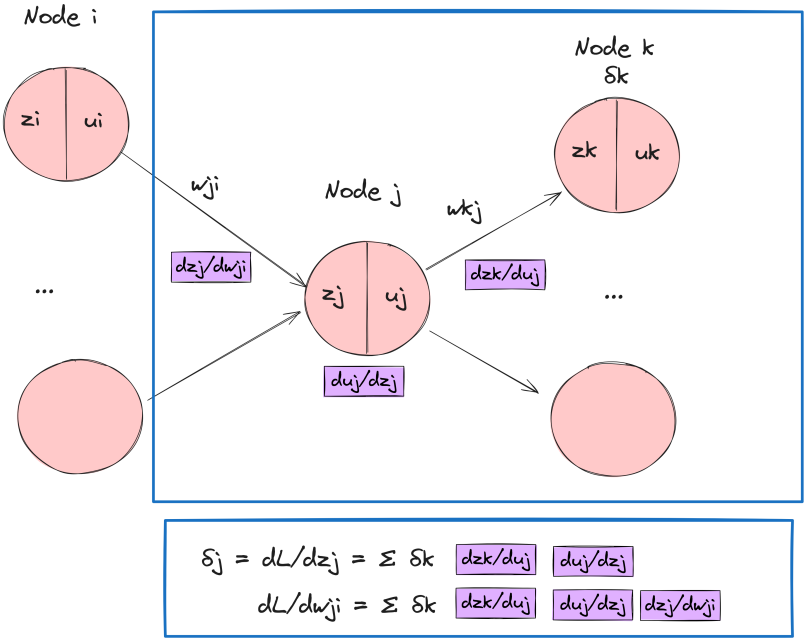

Figure 19: Computing gradient with respect to weight at hidden unit.

As at output unit, the derivative of the loss with respect to a weight $w_{j,i}$ input to the node, $\frac{\partial L}{\partial w_{j,i}}$, is the product of:

- $\delta_j = \frac{\partial L}{\partial z_j}$ (the "accumulated" part)
- $u_i = \frac{\partial z_j}{\partial w_{j,i}}$ (the "local" part)

so for a hidden unit, too, $\frac{\partial L}{\partial w_{j,i}} = \delta_j u_i$

Backpropagation + gradient descent algorithm (1)

1. Start with random (small) weights. Apply input x_n to network and propagate values forward using $z_j = \sum_i w_{j,i} u_i$ and $u_j = g(z_j)$. (Sum is over all inputs to node j.)
2. Evaluate δ_j for all output units.

Backpropagation + gradient descent algorithm (2)

3. Backpropagate the δs to get δ_j for each hidden unit. (Sum is over all outputs of node j.)

$$\delta_j = g'(z_j) \sum_k w_{k,j} \delta_k$$

Backpropagation + gradient descent algorithm (3)

4. Use $\frac{\partial L_n}{\partial w_{j,i}} = \delta_j u_i$ to evaluate derivatives.
5. Update weights using gradient descent.

Why is backpropagation so important?

Example: $e = (a + b) \times (b + 1)$

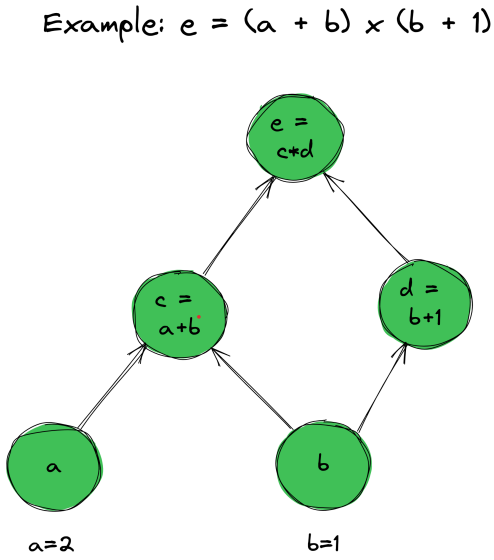

Figure 20: Derivatives on a computational graph

Example via https://colah.github.io/posts/2015-08-Backprop/.

Forward-mode differentiation

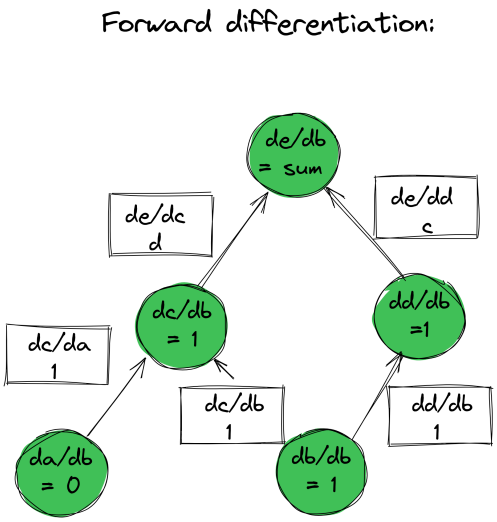

Figure 21: Forward-mode differentiation.

With forward-mode differentiation, we take the derivative of the outupt with respect to one input (e.g. $\frac{de}{db}$), by starting at the *input* and "accumulating" the gradients toward the output.

However, if we want to take derivatives with respect to a *different* input (e.g. input a), these accumulated gradients don't help - we need to compute all of the derivatives again.

Reverse-mode differentiation

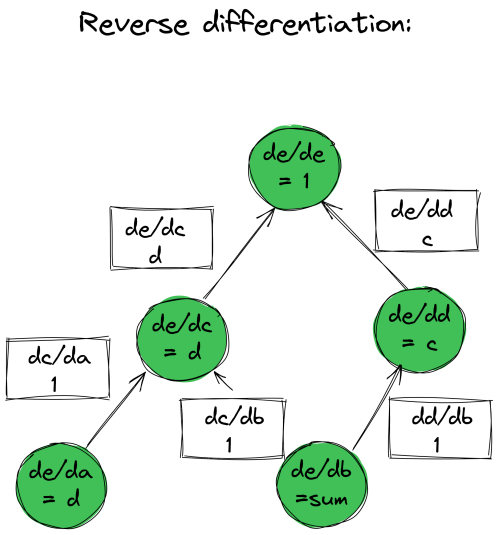

Figure 22: Reverse-mode differentiation.

With reverse mode differentiation, we take the derivative of the outupt with respect to one input (e.g. $\frac{de}{db}$), by starting at the *output* and "accumulating" the gradients toward the input.

If we want to take derivatives with respect to a *different* input (e.g. input a), we already have most of the accumulated gradients - we would just need to compute one more local derivative near that input ($\frac{dc}{da}$).

For a problem where you need derivative of one output (loss) with respect to many inputs (many weights), reverse mode differentiation is very efficient because the accumulated gradients (δ values) are computed once and then reused many times.