Neural networks

Fraida Fund

Contents

In this lecture

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

From linear to non-linear

Representation as a computational graph

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

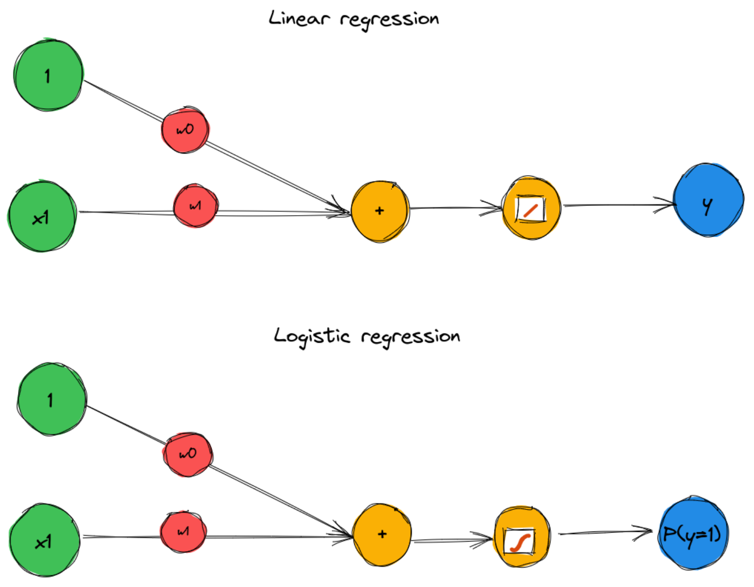


Figure 1: Regression and classification models.

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).

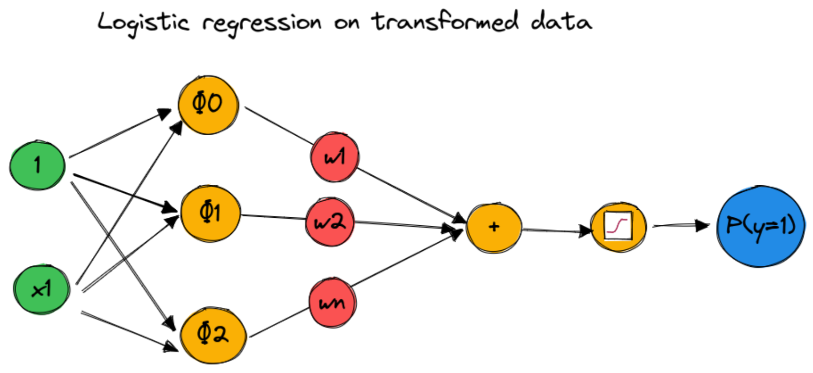


Figure 2: With a basis function transformation.

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.

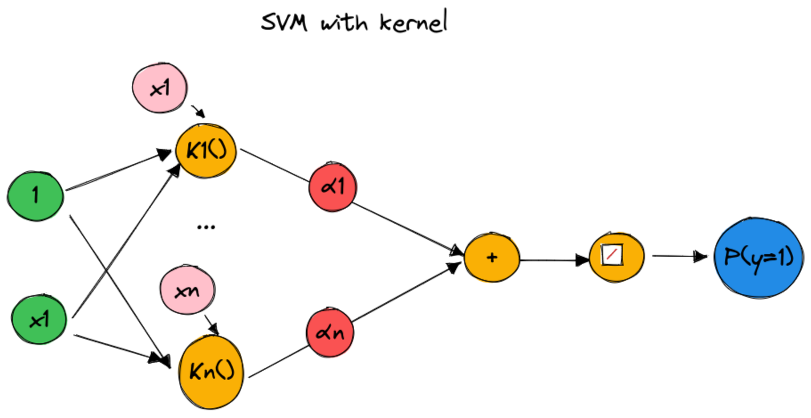


Figure 3: SVM computational graph.

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 instead learn the first part - the mapping of the features to a transformed feature space?

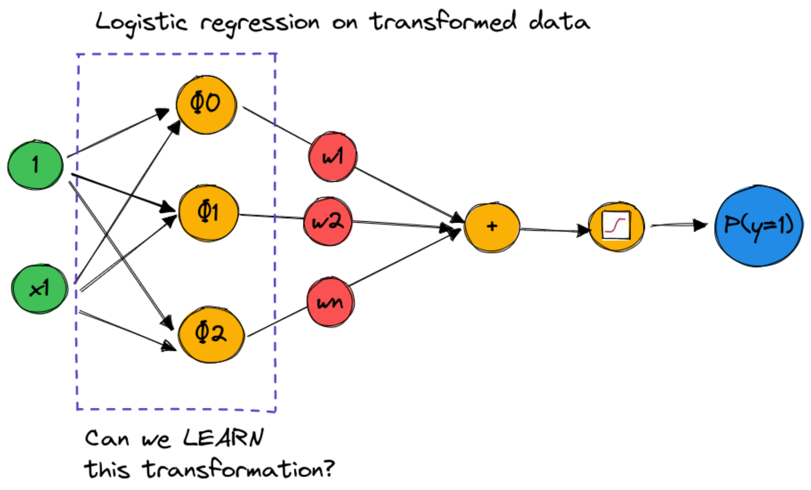

Logistic regression on transformed data
Φ0
Φ1
Φ2
1
x1
w1
w2
wn
+
P(y=1)
Can we LEARN this transformation?

Figure 4: Can we learn this transformation part?

Example: synthetic data

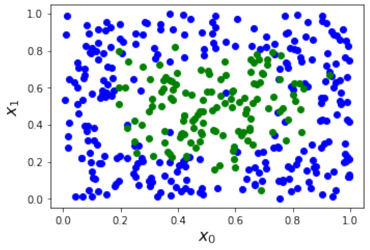


Figure 5: Example via Sundeep Rangan

Model of example two-stage network (1)

First step (*hidden layer*):

- Take $N_H = 4$ "logistic regression" nodes.
- Use (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.

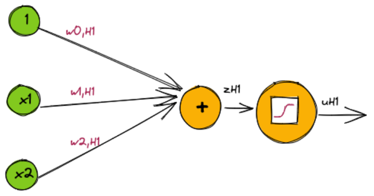


Figure 6: Computation at one hidden node.

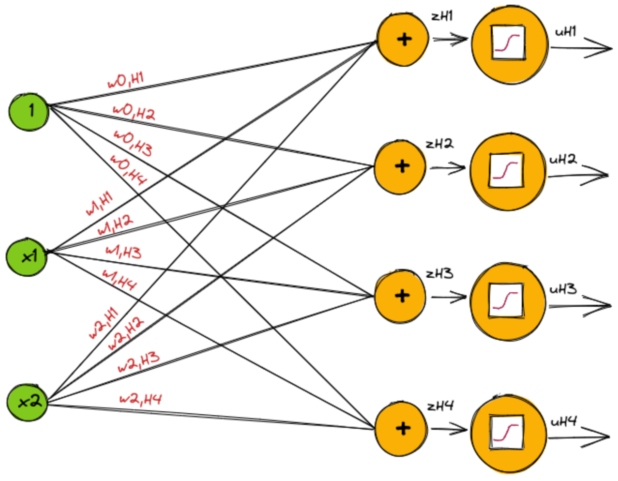


Figure 7: Computation at four hidden nodes.

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.

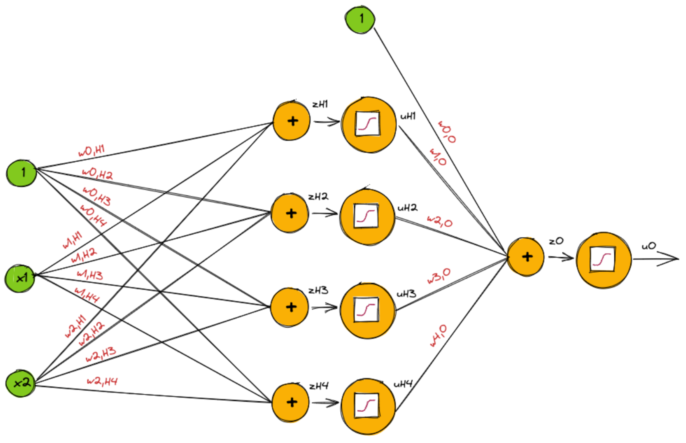


Figure 8: Two-stage network.

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

Example: output of each hidden node

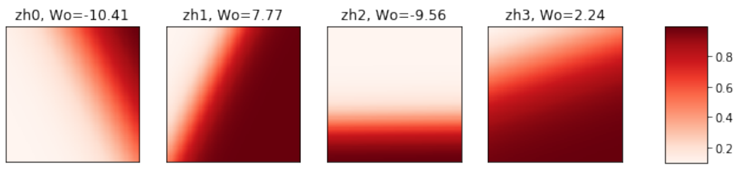


Figure 9: Example via Sundeep Rangan

Example: output of output node

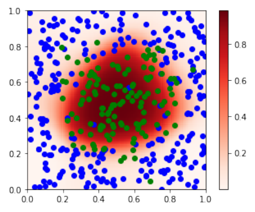


Figure 10: Via Sundeep Rangan

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

Training the two-stage network for binary classification

- From final stage: $z_o = F(\mathbf{x}, \mathbf{W})$ where parameters $\mathbf{W} = (\mathbf{W}_H, \mathbf{W}_o)$
- Given training data $(\mathbf{x}_i, y_i), i = 1, \ldots, n$
- Loss function $L(\mathbf{W}) := -\sum_{i=1}^{n} \ln P(y_i | \mathbf{x}_i, \mathbf{W})$
- Choose parameters to minimize loss: $\hat{\mathbf{W}} = \operatorname{argmin}_{\mathbf{W}} L(\mathbf{W})$

(Using negative log likelihood/binary cross-entropy loss function from the logistic regression lesson.)

How do we choose the parameters in the last step? We'll use *gradient descent* on the computational graph. More on that soon...

Neural networks

Biological inspiration

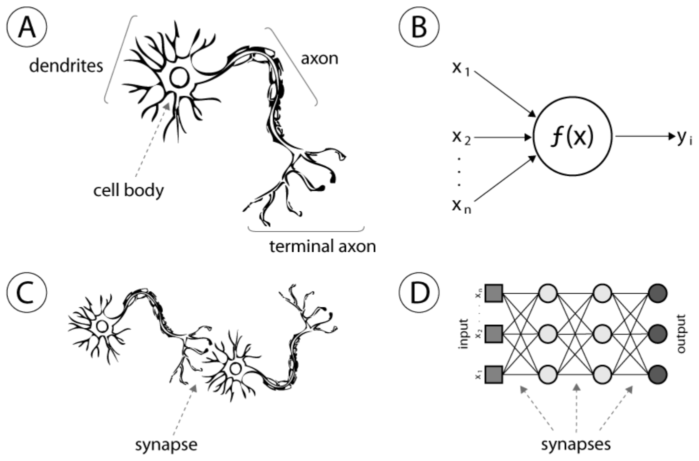


Figure 11: A biological neuron accepts inputs via dendrites, "adds" them together within cell body, and once some electrical potential is reached, "fires" via axons. Synaptic weight (the influence the firing of one neuron has on another) changes over time ("learning"). Image via: http://http://doi.org/10.5772/51275

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.

Activation function at output layer

What $g_O(z)$ to use for...

- Regression?
- Binary classification?
- Multi-class classification?

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

- z_O is scalar - need one output node
- Soft decision: $P(y = 1|x) = \frac{1}{1+e^{-z_O}}$
- Then you can apply threshold to get $\hat{y}$

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

- $\mathbf{z}_O = [z_{O,1}, \ldots, z_{O,K}]$ is a vector - need K output nodes
- Soft decision: $P(y = k|x) = \frac{e^{z_{O,k}}}{\sum_{\ell=1}^{K} e^{z_\ell}}$ (softmax)
- Then you can select label by $\hat{y} = \text{argmax}_k\, z_{O,k}$

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

- Can be a vector - need K output nodes
- Linear activation: $\hat{y} = z_O$

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.)

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

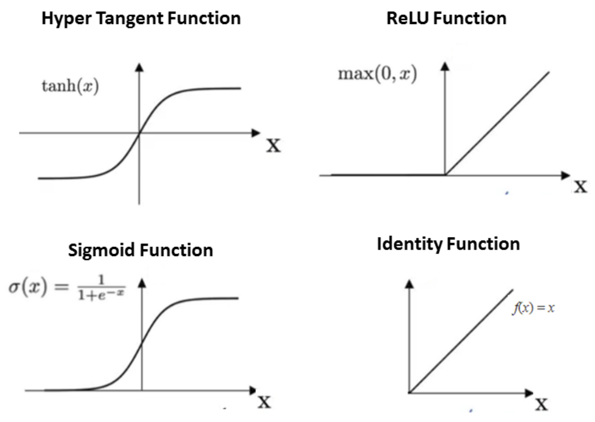


Figure 12: Most common activation functions

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)

Dimension (1)

- N_I = input dimension, number of features
- N_H = number of hidden units
- 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		$N_H(N_I + 1) + N_O(N_H + 1)$

Loss function: regression

- y_i is scalar target variable for sample i
- z_{Oi} is estimate of y_i
- Use L2 loss:

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

Loss function: regression with vector output

- For vector $\mathbf{y}_i = (y_{i1}, \ldots, y_{iK})$, use vector L2 loss:

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

Loss function: binary classification (1)

- $y_i = 0, 1$ is target variable for sample i
- z_{Oi} is scalar, called “logit score”
- Negative log likelihood loss:

$$L(\mathbf{W}) = -\sum_{i=1}^{n} \text{ln} P(y_i|\mathbf{x}_i, \mathbf{W}), \quad P(y_i = 1|\mathbf{x}_i, \mathbf{W}) = \frac{1}{1 + e^{-z_{Oi}}}$$

Loss function: binary classification (2)

Loss (binary cross entropy):

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

Loss function: multi-class classification (1)

- $y_i = 1, \ldots, K$ is target variable for sample i
- $\mathbf{z}_{Oi} = (z_{Oi1}, \ldots, z_{OiK})$ is vector with one entry per class
- Likelihood given by softmax function, class with highest score has highest probability:

$$P(y_i = k|\mathbf{x}_i, \mathbf{W}) = g_k(\mathbf{z}_{Oi}), \quad g_k(\mathbf{z}_{Oi}) = \frac{e^{z_{OiK}}}{\sum_\ell e^{z_{Oi\ell}}}$$

Loss function: multi-class classification (2)

Define “one-hot” vector - for a sample from class k, all entries in the vector are 0 except for the kth entry which is 1:

$$r_{ik} = \begin{cases} 1 & y_i = k \\ 0 & y_i \neq k \end{cases}$$

Loss function: multi-class classification (3)

Negative log likelihood

$$\ln P(y_i = k | \mathbf{x}_i, \mathbf{W}) = \sum_{k=1}^{K} \ln P(y_i = k | \mathbf{x}_i, \mathbf{W})$$

Loss function: multi-class classification (4)

Loss (categorical cross entropy):

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

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

Composite functions and computation 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:

$$\begin{tikzcd} x \arrow[r] & v = h(x) \arrow[r] & u = g(v) \arrow[r] & f(u) \end{tikzcd}$$

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 $f(u)$

Derivative of composite function

- Suppose need to compute the derivative of the composite function $f(g(h(x)))$ with respect to x
- We will use the chain rule.

Backward pass on computational graph

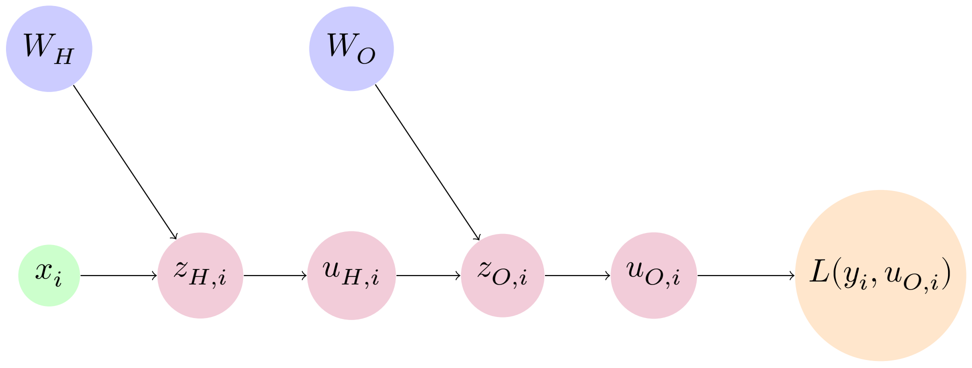
We can compute this chain rule derivative by doing a *backward pass* on the computational graph:

We just need to get the derivative of each node with respect to its inputs:

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

$$\begin{tikzcd} x & v = h(x) \arrow[l, "\frac{dh}{dx}"'] & u = g(v) \arrow[l, "\frac{dg}{dv}"'] & f(u) \arrow[l, "\frac{df}{du}"'] \end{tikzcd}$$

Neural network computation graph



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.

Backpropagation error: output unit

For output unit in *regression* network, where $u_O = z_O$,

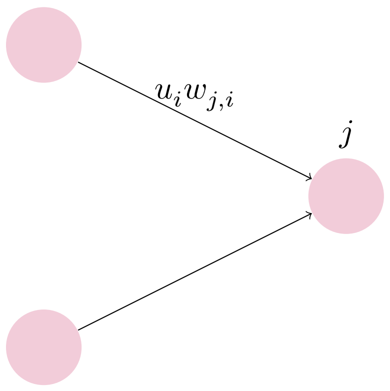
$$L = \frac{1}{2}\sum_n (y_n - z_{O,n})^2$$

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

Note: this is the error of the model, the difference between true value and network output!

More generally, if not using the identity activation function at the output, you compute $\delta_O = \frac{\partial L}{\partial u_O}\frac{\partial u_O}{\partial z_O}$.

Backpropagation error: unit with inputs illustration

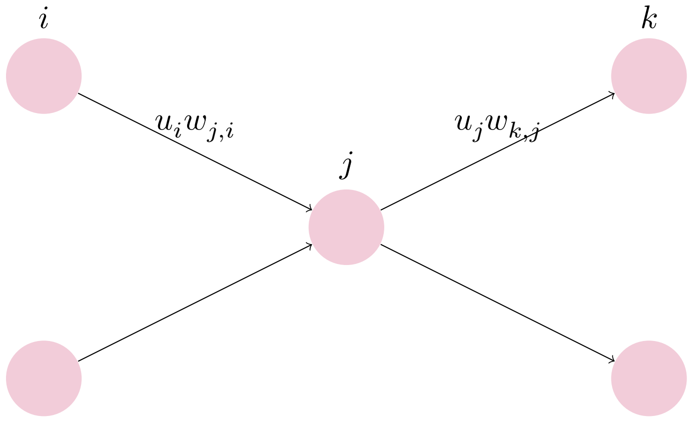


At a node j, $z_j = \sum_i w_{j,i} u_i$

Backpropagation error: unit with inputs

- Backpropagation error of node j is $\delta_j = \frac{\partial L}{\partial z_j}$
- Chain rule: $\frac{\partial L}{\partial w_{j,i}} = \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial w_{j,i}}$
- Since $z_j = \sum_i w_{j,i} u_i$, $\frac{\partial z_j}{\partial w_{j,i}} = u_i$
- Then $\frac{\partial L}{\partial w_{j,i}} = \delta_j u_i$

Backpropagation error: unit with inputs and outputs illustration



Backpropagation error: unit with inputs and outputs (1)

For a hidden unit,

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

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

using $\delta_k = \frac{\partial L}{\partial z_k}$.

Backpropagation error: unit with inputs and outputs (2)

And because $z_k = \sum_l w_{k,l} u_l = \sum_l w_{k,l} g(z_l)$,

$\frac{\partial z_k}{\partial z_j} = w_{k,j} g'(z_j)$.

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_k 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.

Derivatives for common loss functions

Squared/L2 loss:

$$L = \sum_i (y_i - z_{O,i})^2, \quad \frac{\partial L}{\partial z_{O,i}} = \sum_i -(y_i - z_{O,i})$$

Binary cross entropy loss:

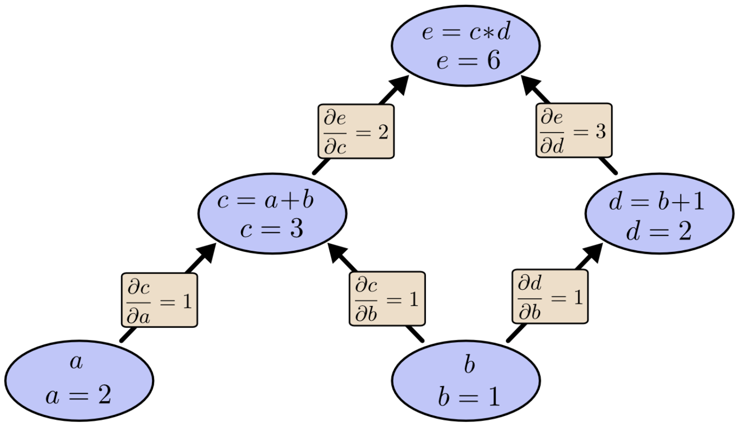
$$L = \sum_i -y_i z_{O,i} + \ln(1 + e^{y_i z_{O,i}}), \quad \frac{\partial L}{\partial z_{O,i}} = y_i - \frac{e^{y_i z_{O,i}}}{1 + e^{y_i z_{O,i}}}$$

Derivatives for common activation functions

- Sigmoid activation: $g'(x) = \sigma(x)(1 - \sigma(x))$
- Tanh activation: $g'(x) = \frac{1}{\cosh^2(x)}$

Why is backpropagation so important?

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

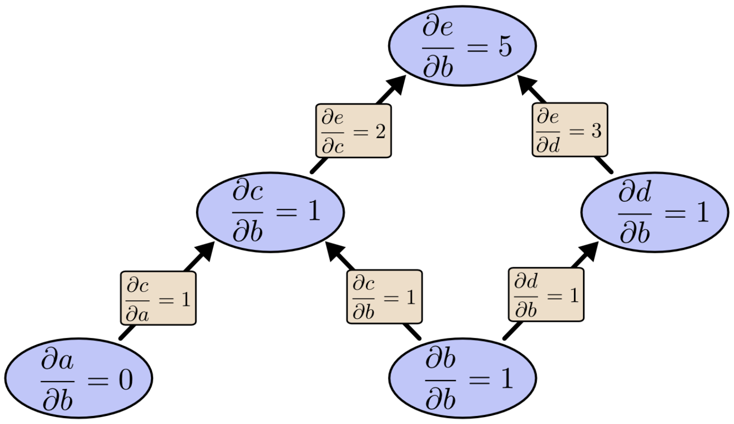


$e = c*d$
$e = 6$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$c = a+b$
$c = 3$
$d = b+1$
$d = 2$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
a
$a = 2$
b
$b = 1$

Figure 13: Derivatives on a computational graph

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

Forward-mode differentiation



$\frac{\partial e}{\partial b} = 5$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial a}{\partial b} = 0$
$\frac{\partial b}{\partial b} = 1$

Figure 14: Forward-mode differentiation from input to output gives us derivative of every node with respect to each input. Then we can compute the derivative of output with respect to input.

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

Reverse-mode differentiation

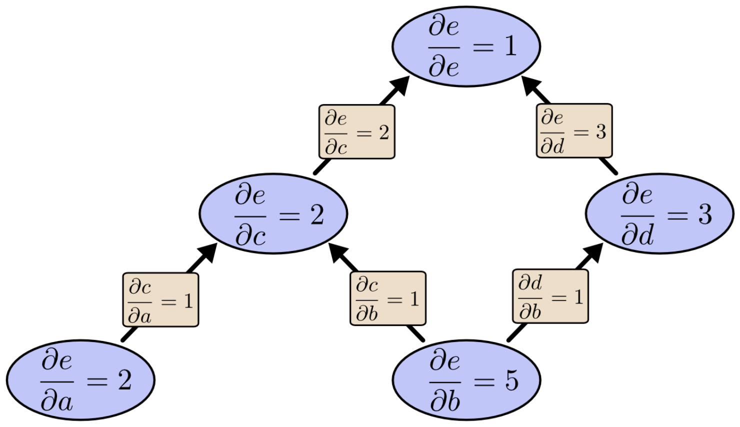

$\frac{\partial e}{\partial e} = 1$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial e}{\partial c} = 2$
$\frac{\partial e}{\partial d} = 3$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
$\frac{\partial e}{\partial a} = 2$
$\frac{\partial e}{\partial b} = 5$

Figure 15: Reverse-mode differentiation from output to input gives us derivative of output with respect to every node.

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