CPE 486/586: Deep Learning for Engineering Applications

03 Intorducing Deep Learning: Neural Networks

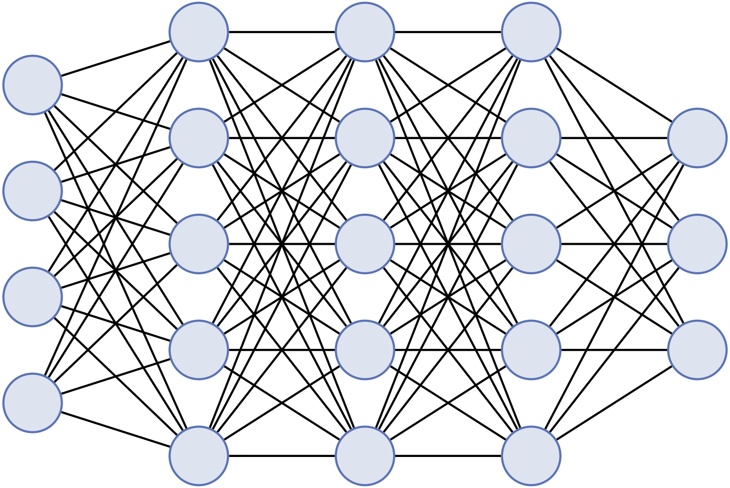
Spring 2026

Rahul Bhadani

Outline

A Single Perceptron: Build Blocks of Neural Network

A Neural Network



Linear Model

$$y = wx + b$$

For n features, and introducing unit feature $x_0 = 1$ to incorporate b into summation as w_0

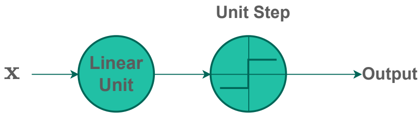
$$y = \sum_{i=0}^{n} w_i x_i$$

or

$$y = \mathbf{w}^\top \mathbf{x}$$

Perceptron

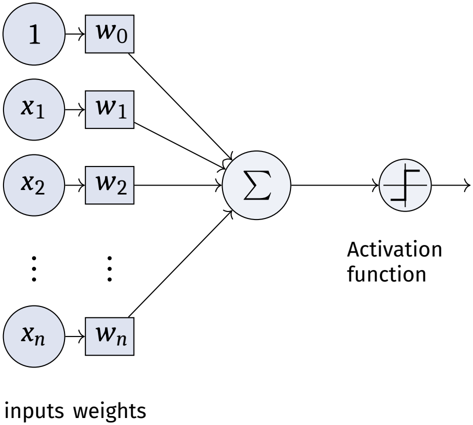
A neuron with unit-step for decision making:



No hidden layer present.
Unit step is heaviside function

$$g(z) = \begin{cases} 1, & \text{if } z \geq \theta \\ 0, & \text{otherwise} \end{cases}$$

So what could be the problem here?



Need for multi-layer perceptron

1. A single-layer perceptron is good for linearly separable classes.
2. Optimization does not converge for nonlinearly separable datasets.
3. Non-differentiability of unit-step prevents using gradient descent.
4. A single-layer perceptron lacks generalizability.

Activation function with differentiability

1. Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

   preferred for predicting probabilities as range is between 0 and 1.

2. Tanh or Hyperbolic tangent:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

   Suitable for being used as activation functon for hidden layers. Ranges between -1 and +1.

Activation function with differentiability

1. Rectified Linear Unit (ReLU):

$$ReLU(x) = max(0, x)$$

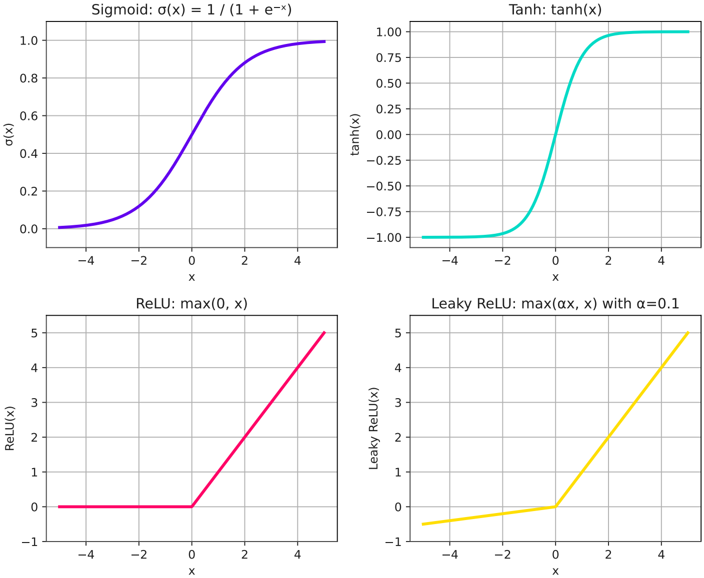
   Suitable for regression problems. Range between 0 to ∞.

2. Leaky ReLU:

$$LeakyReLU(x) = max(0.01 * x, x)$$

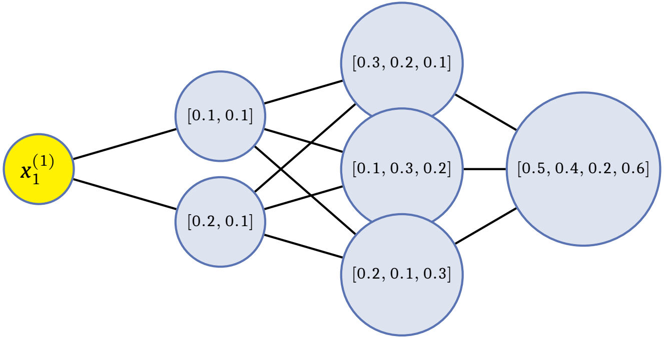
   Suitable for regression problems. It solves the problem of dead neuron suffered while ReLU is used, i.e., if a neuron receives only negative inputs, it outputs zero and its gradient becomes zero. This means it stops learning. Leaky ReLU is a modified version of ReLU designed to fix the problem of dead neurons. Instead of returning zero for negative inputs it allows a small, non-zero value.

Activation function with differentiability



Training a Simple Neural Network with One Hidden Layer

A Simple Neural Network



Each neuron is a denoted by circle consisting of affine part (Linear Model) and activation function.

1. Single feature, the first layer contains two neurons, one hidden layer with three neurons, and one output.
2. Consider classification problem either 0 or 1.
3. Superscript (1) means first training sample, subscript 1 feature index.
4. Intermediate nodes $\mathbf{w} = [w_0, w_1, \ldots]$.

Forward Propagation

Consider the first training sample $x_1^{(1)} = 2.0$. True label as 1. Also consider Sigmoid as the activation function, i.e. $\sigma(z) = \frac{1}{1+e^{-z}}$.

Layer 1:

1. Neuron 1: $z_1^{[1]} = 0.1 + 0.1 \times 2.0 = 0.3$. $a_1^{[1]} = \frac{1}{1+e^{-0.3}} = 0.5744$.
2. Neuron 2: $z_2^{[1]} = 0.2 + 0.1 \times 2.0 = 0.4$. $a_2^{[1]} = \frac{1}{1+e^{-0.4}} = 0.5987$.

Here, in an essence, the first layer expands one feature to two features, and basically each neurons will learn something different from the training data.

Forward Propagation

Layer 2:

1. Neuron 1: $z_1^{[2]} = 0.3 + 0.2 \times 0.5744 + 0.1 \times 0.5987 = 0.4748.$
   $a_1^{[2]} = \sigma(0.4748) = 0.6165.$
2. Neuron 2: $z_2^{[2]} = 0.1 + 0.3 \times 0.5744 + 0.2 \times 0.5987 = 0.3921.$
   $a_2^{[2]} = \sigma(0.3921) = 0.5968.$
3. Neuron 3: $z_3^{[2]} = 0.2 + 0.1 \times 0.5744 + 0.3 \times 0.5987 = 0.4371.$
   $a_3^{[2]} = \sigma(0.4371) = 0.6076.$

Forward Propagation

Layer 3 (Output):

1. Output Neuron: $z_1^{[3]} = 0.5 + 0.4 \times 0.6165 + 0.2 \times 0.5968 + 0.6 \times 0.6075 = 1.2305.$
2. $a_1^{[3]} = \sigma(1.2305) = 0.7739.$

The final output $\hat{y} = 0.7739.$

Cross-Entropy Loss Calculation

Given the true label $y = 1$ and predicted output $\hat{y} = 0.7739$:

$$
\begin{aligned}
\text{Cross-Entropy Loss} &= -[y \log(\hat{y}) + (1-y)\log(1-\hat{y})] \\
&= -[1 \cdot \log(0.7739) + 0 \cdot \log(1 - 0.7739)] \\
&= -\log(0.7739) \\
&= -(-0.2563) \\
&= 0.2563
\end{aligned}
$$

Where $\log$ denotes natural logarithm, and $\log(0.7740) \approx -0.2563$.

- The loss quantifies how well the network's prediction matches the true label.
- Lower loss indicates better prediction (perfect prediction would give loss = 0).
- For binary classification with sigmoid activation, cross-entropy loss is commonly used.

Cross-Entropy Loss and Its Derivative

For binary classification with true label $y \in \{0, 1\}$ and predicted probability $\hat{y} \in (0, 1)$, the cross-entropy loss is:

$$L(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

Its derivative:

$$\begin{aligned}
\frac{\partial L}{\partial \hat{y}} &= \frac{\partial}{\partial \hat{y}}[-y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})] \\
&= -y \cdot \frac{1}{\hat{y}} \cdot 1 - (1 - y) \cdot \frac{1}{1 - \hat{y}} \cdot (-1) \\
&= -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}}
\end{aligned}$$

- When $y = 1$: $\frac{\partial L}{\partial \hat{y}} = -\frac{1}{\hat{y}} + 0 = -\frac{1}{\hat{y}}$
- When $y = 0$: $\frac{\partial L}{\partial \hat{y}} = 0 + \frac{1}{1-\hat{y}} = \frac{1}{1-\hat{y}}$
- The gradient encourages $\hat{y}$ to move toward y (negative gradient pushes $\hat{y}$ up when $y = 1$)

Sigmoid Activation Function and Its Derivative

The sigmoid function and its derivative have a nice property:

$$\sigma(z) = \frac{1}{1+e^{-z}} = \hat{y}$$

$$\frac{d\sigma}{dz} = \sigma(z)[1-\sigma(z)] = \hat{y}(1-\hat{y})$$

Derivation:

$$\begin{aligned}
\frac{d\sigma}{dz} &= \frac{d}{dz}\left(1+e^{-z}\right)^{-1} \\
&= -(1+e^{-z})^{-2}\cdot(-e^{-z}) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}}\cdot\frac{e^{-z}}{1+e^{-z}} \\
&= \sigma(z)\cdot\left(1-\frac{1}{1+e^{-z}}\right) \\
&= \sigma(z)[1-\sigma(z)]
\end{aligned}$$

The derivative is expressed in terms of the function value itself . No need to recompute e^{-z}.

Chain Rule in Backpropagation

Backpropagation uses the chain rule from calculus:
If $L = f(g(h(x)))$, then:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial x}$$

In Neural Networks:

$$\begin{aligned}
L &\leftarrow \text{Loss function} \\
\hat{y} &\leftarrow \sigma(z^{[\ell]}) \quad \text{(output activation), } \ell \text{ denotes layer index} \\
z^{[\ell]} &\leftarrow w^{[\ell]} a^{[\ell-1]} + w_0^{[\ell]} \quad \text{(linear combination)} \\
a^{[\ell-1]} &\leftarrow \sigma(z^{[\ell-1]}) \quad \text{(previous layer activation)}
\end{aligned}$$

Gradient flow:

$$\begin{aligned}
\frac{\partial L}{\partial w^{[\ell]}} &= \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[\ell]}} \cdot \frac{\partial z^{[\ell]}}{\partial w^{[\ell]}} \\
&= \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \right) \cdot \hat{y}(1-\hat{y}) \cdot a^{[\ell-1]}
\end{aligned}$$

Note: $\hat{y}(1-\hat{y})$ cancels when $\hat{y}$ is not 0 or 1, simplifying computation!

These Derivatives Are Well-Behaved

1. Numerical Stability:

- Cross-entropy with sigmoid avoids extreme gradients
- For $\hat{y} \approx 0$ or $\hat{y} \approx 1$: $\hat{y}(1-\hat{y}) \approx 0$
- This prevents huge weight updates when confident but wrong

2. Gradient Behavior:

$$\frac{\partial L}{\partial z^{[\ell]}} = \hat{y} - y \quad \text{(after simplification!)}$$

Derivation:

$$\begin{aligned} \frac{\partial L}{\partial z^{[\ell]}} &= \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right) \cdot \hat{y}(1-\hat{y}) \\ &= -y(1-\hat{y}) + (1-y)\hat{y} = -y + y\hat{y} + \hat{y} - y\hat{y} = \hat{y} - y \end{aligned}$$

Beautiful Result: The gradient is simply the prediction error!

- When $\hat{y} > y$: Positive gradient decreases weights
- When $\hat{y} < y$: Negative gradient increases weights
- Intuitive and numerically stable

Backpropagation: Gradient Calculation

We'll compute gradients using chain rule. Recall: $\hat{y} = 0.7740$, $y = 1$.

Step 1: Output Layer Gradient

$$\frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = -\frac{1}{0.7740} + 0 = -1.2921$$

$$\frac{\partial \hat{y}}{\partial z^{[3]}} = \hat{y}(1-\hat{y}) = 0.7739 \times 0.2261 = 0.1750$$

$$\frac{\partial L}{\partial z^{[3]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[3]}} = -1.2922 \times 0.1750 = -0.2261$$

Backpropagation: Gradient Calculation

Step 2: Layer 3 Weight Gradients

$$\frac{\partial z^{[3]}}{\partial w_0^{[3]}} = 1, \quad \frac{\partial L}{\partial w_0^{[3]}} = -0.2261 \times 1 = -0.2261$$

$$\frac{\partial z^{[3]}}{\partial w_1^{[3]}} = a_1^{[2]} = 0.6165, \quad \frac{\partial L}{\partial w_1^{[3]}} = -0.2261 \times 0.6165 = -0.1394$$

$$\frac{\partial z^{[3]}}{\partial w_2^{[3]}} = a_2^{[2]} = 0.5968, \quad \frac{\partial L}{\partial w_2^{[3]}} = -0.2261 \times 0.5968 = -0.1349$$

$$\frac{\partial z^{[3]}}{\partial w_3^{[3]}} = a_3^{[2]} = 0.6075, \quad \frac{\partial L}{\partial w_3^{[3]}} = -0.2261 \times 0.6075 = -0.1374$$

Backpropagation: Layer 2 Gradients

Step 3: Backpropagate to Layer 2

$$\frac{\partial L}{\partial a_1^{[2]}} = \frac{\partial L}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a_1^{[2]}} = -0.2261 \times 0.4 = -0.0904$$

$$\frac{\partial L}{\partial a_2^{[2]}} = -0.2261 \times 0.2 = -0.0452$$

$$\frac{\partial L}{\partial a_3^{[2]}} = -0.2261 \times 0.6 = -0.1357$$

Backpropagation: Gradient Calculation

Step 4: Layer 2 Neuron Gradients For each neuron i in layer 2: $\frac{\partial a_i^{[2]}}{\partial z_i^{[2]}} = a_i^{[2]}(1 - a_i^{[2]})$

$$\frac{\partial L}{\partial z_1^{[2]}} = \frac{\partial L}{\partial a_1^{[2]}} \cdot a_1^{[2]}(1 - a_1^{[2]}) = -0.0904 \times (0.6165 \times 0.3835) = -0.0904 \times 0.2364 = -0.0214$$

$$\frac{\partial L}{\partial z_2^{[2]}} = -0.0452 \times (0.5968 \times 0.4032) = -0.0452 \times 0.2406 = -0.0109$$

$$\frac{\partial L}{\partial z_3^{[2]}} = -0.1356 \times (0.6075 \times 0.3925) = -0.1356 \times 0.2384 = -0.0323$$

Backpropagation: Layer 2 Weight Gradients

Step 5: Layer 2 Weight Gradients For neuron 1 in layer 2:

$$\frac{\partial L}{\partial w_{01}^{[2]}} = \frac{\partial L}{\partial z_1^{[2]}} \times 1 = -0.0214$$
$$\frac{\partial L}{\partial w_{11}^{[2]}} = \frac{\partial L}{\partial z_1^{[2]}} \times a_1^{[1]} = -0.0214 \times 0.5744 = -0.0123$$
$$\frac{\partial L}{\partial w_{21}^{[2]}} = \frac{\partial L}{\partial z_1^{[2]}} \times a_2^{[1]} = -0.0214 \times 0.5987 = -0.0128$$

For neuron 2 in layer 2:

$$\frac{\partial L}{\partial w_{02}^{[2]}} = -0.0109, \quad \frac{\partial L}{\partial w_{12}^{[2]}} = -0.0109 \times 0.5744 = -0.0063, \quad \frac{\partial L}{\partial w_{22}^{[2]}} = -0.0109 \times 0.5987 = -0.0065$$

For neuron 3 in layer 2:

$$\frac{\partial L}{\partial w_{03}^{[2]}} = -0.0323, \quad \frac{\partial L}{\partial w_{13}^{[2]}} = -0.0323 \times 0.5744 = -0.0186, \quad \frac{\partial L}{\partial w_{23}^{[2]}} = -0.0323 \times 0.5987 = -0.0194$$

Backpropagation: Layer 1 Gradients

Step 6: Backpropagate to Layer 1

$$\frac{\partial L}{\partial a_1^{[1]}} = \sum_{j=1}^{3} \frac{\partial L}{\partial z_j^{[2]}} \cdot w_{1j}^{[2]}$$

$$= (-0.0214 \times 0.2) + (-0.0109 \times 0.3) + (-0.0323 \times 0.1)$$

$$= -0.00428 - 0.00327 - 0.00323 = -0.01078$$

$$\frac{\partial L}{\partial a_2^{[1]}} = (-0.0214 \times 0.1) + (-0.0109 \times 0.2) + (-0.0323 \times 0.3)$$

$$= -0.00214 - 0.00218 - 0.00969 = -0.01401$$

Step 7: Layer 1 Neuron Gradients

$$\frac{\partial L}{\partial z_1^{[1]}} = \frac{\partial L}{\partial a_1^{[1]}} \cdot a_1^{[1]}(1 - a_1^{[1]}) = -0.01078 \times (0.5744 \times 0.4256) = -0.01078 \times 0.2445 = -0.00264$$

$$\frac{\partial L}{\partial z_2^{[1]}} = -0.01401 \times (0.5987 \times 0.4013) = -0.01401 \times 0.2403 = -0.00337$$

Backpropagation: Layer 1 Weight Gradients

Step 8: Layer 1 Weight Gradients For neuron 1 in layer 1 (input $x_1^{(1)} = 2.0$):

$$\frac{\partial L}{\partial w_{01}^{[1]}} = \frac{\partial L}{\partial z_1^{[1]}} \times 1 = -0.00264$$

$$\frac{\partial L}{\partial w_{11}^{[1]}} = \frac{\partial L}{\partial z_1^{[1]}} \times x_1^{(1)} = -0.00264 \times 2.0 = -0.00528$$

For neuron 2 in layer 1:

$$\frac{\partial L}{\partial w_{02}^{[1]}} = \frac{\partial L}{\partial z_2^{[1]}} \times 1 = -0.00337$$

$$\frac{\partial L}{\partial w_{12}^{[1]}} = \frac{\partial L}{\partial z_2^{[1]}} \times x_1^{(1)} = -0.00337 \times 2.0 = -0.00674$$

We've computed all gradients needed for weight updates using gradient descent:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial L}{\partial \mathbf{w}}$$

where η is the learning rate.

Weight Updates with Gradient Descent

Using gradient descent with learning rate $\eta = 0.1$:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \eta \cdot \frac{\partial L}{\partial \mathbf{w}}$$

Output Layer Updates:

$$w_{0\,\text{new}}^{[3]} = 0.5 - 0.1 \times (-0.2261) = 0.5 + 0.0226 = 0.5226$$
$$w_{1\,\text{new}}^{[3]} = 0.4 - 0.1 \times (-0.1393) = 0.4 + 0.01393 = 0.4139$$
$$w_{2\,\text{new}}^{[3]} = 0.2 - 0.1 \times (-0.1349) = 0.2 + 0.01349 = 0.2135$$
$$w_{3\,\text{new}}^{[3]} = 0.6 - 0.1 \times (-0.1373) = 0.6 + 0.01373 = 0.6137$$

Layer 2 Weight Updates

Neuron 1 in Layer 2:

$$w_{01}^{[2]\text{new}} = 0.3 - 0.1 \times (-0.0214) = 0.3 + 0.00214 = 0.3021$$
$$w_{11}^{[2]\text{new}} = 0.2 - 0.1 \times (-0.0123) = 0.2 + 0.00123 = 0.2012$$
$$w_{21}^{[2]\text{new}} = 0.1 - 0.1 \times (-0.0128) = 0.1 + 0.00128 = 0.1013$$

Neuron 2 in Layer 2:

$$w_{02}^{[2]\text{new}} = 0.1 - 0.1 \times (-0.0109) = 0.1 + 0.00109 = 0.1011$$
$$w_{12}^{[2]\text{new}} = 0.3 - 0.1 \times (-0.0063) = 0.3 + 0.00063 = 0.3006$$
$$w_{22}^{[2]\text{new}} = 0.2 - 0.1 \times (-0.0065) = 0.2 + 0.00065 = 0.2007$$

Neuron 3 in Layer 2:

$$w_{03}^{[2]\text{new}} = 0.2 - 0.1 \times (-0.0323) = 0.2 + 0.00323 = 0.2032$$
$$w_{13}^{[2]\text{new}} = 0.1 - 0.1 \times (-0.0186) = 0.1 + 0.00186 = 0.1019$$
$$w_{23}^{[2]\text{new}} = 0.3 - 0.1 \times (-0.0193) = 0.3 + 0.00193 = 0.3019$$

Layer 1 Weight Updates

Neuron 1 in Layer 1:

$$w_{01}^{[1]\text{new}} = 0.1 - 0.1 \times (-0.00264) = 0.1 + 0.000264 = 0.1003$$
$$w_{11}^{[1]\text{new}} = 0.1 - 0.1 \times (-0.00528) = 0.1 + 0.000528 = 0.1005$$

Neuron 2 in Layer 1:

$$w_{02}^{[1]\text{new}} = 0.2 - 0.1 \times (-0.00337) = 0.2 + 0.000337 = 0.2003$$
$$w_{12}^{[1]\text{new}} = 0.1 - 0.1 \times (-0.00674) = 0.1 + 0.000674 = 0.1007$$

- All weights increased slightly because gradients were negative
- Negative gradient means loss decreases as weight increases
- Largest updates: Output layer weights (especially bias $w_0^{[3]}$)
- Smallest updates: First layer weights (vanishing gradient effect)
- After one training step: Loss should decrease on next forward pass

Verification: Forward Pass with New Weights

Let's verify the network improves by doing one forward pass with updated weights:

Layer 1 with new weights:

$$z_1^{[1]} = 0.1003 + 0.1005 \times 2.0 = 0.3013 \qquad a_1^{[1]} = 0.5748$$
$$z_2^{[1]} = 0.2003 + 0.1007 \times 2.0 = 0.4017 \qquad a_2^{[1]} = 0.5991$$

Layer 2 with new weights:

$$z_1^{[2]} = 0.3021 + 0.2012 \times 0.5748 + 0.1013 \times 0.5991 = 0.4785 \qquad a_1^{[2]} = 0.6174$$
$$z_2^{[2]} = 0.1011 + 0.3006 \times 0.5748 + 0.2007 \times 0.5991 = 0.3941 \qquad a_2^{[2]} = 0.5973$$
$$z_3^{[2]} = 0.2032 + 0.1019 \times 0.5748 + 0.3019 \times 0.5991 = 0.4427 \qquad a_3^{[2]} = 0.6089$$

Output Layer with new weights:

$$z_1^{[3]} = 0.5226 + 0.4139 \times 0.6147 + 0.2135 \times 0.5973 + 0.6137 \times 0.6089 = 1.2794$$
$$\hat{y}_{\text{new}} = \sigma(1.2794) = 0.7823$$

Verification: New Loss Calculation

Original prediction: $\hat{y} = 0.7739$, New prediction: $\hat{y}_{\text{new}} = 0.7823$

Original Loss:

$$L = -\log(0.7739) = 0.2563$$

New Loss:

$$L_{\text{new}} = -\log(0.7823) = 0.2455$$

Improvement:

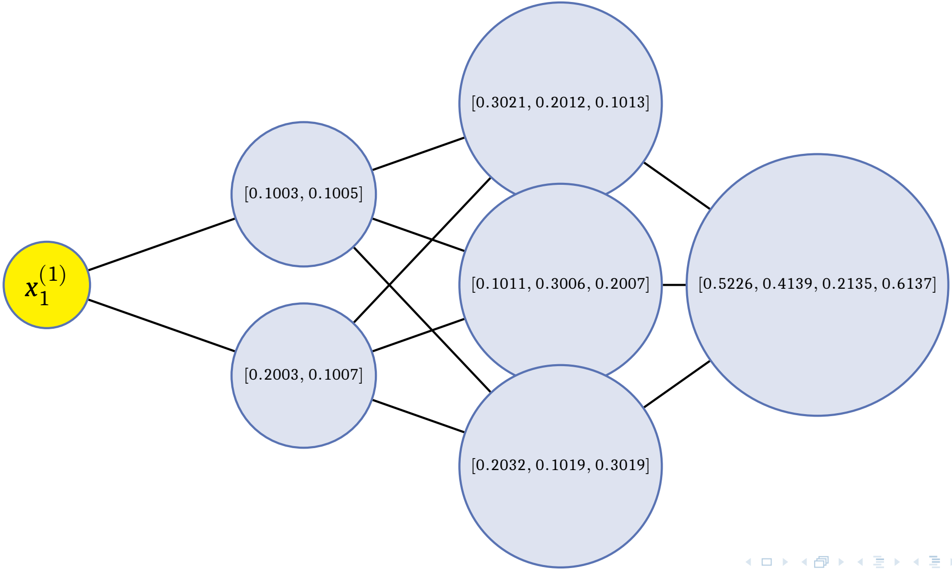
$$\Delta L = 0.2563 - 0.2455 = 0.0108 \quad \text{(4.23\% reduction)}$$

Observations:

- The network output increased from 0.7739 to 0.7823 (closer to target 1.0)
- Loss decreased from 0.2563 to 0.2455
- Gradient descent successfully reduced the loss
- Multiple epochs (repetitions) would further optimize weights

Learning Progress: The network is learning to better predict the true label!

Updated Network Visualization



At the end of first iteration:

- All weights slightly increased (negative gradients)
- Output layer bias $w_0^{[3]}$ increased most (0.5 → 0.5226)
- First layer weights changed minimally (vanishing gradient)
- Network now predicts 0.7862 (improved from 0.7740)

Next Steps: Repeat process for entire training dataset over many epochs!

A Brief Matrix Calculus

Derivatives in Neural Network Training

1. Training means choose suitable $\mathbf{w}$ iteratively.
2. i.e. minimizing a loss function.

Minimize loss $\rightarrow$ Gradient Descent $\rightarrow$ partial derivative wrt $\mathbf{w}$.

$\mathbf{w}$ is a vector.
For neuron, scalar version can work, for multiple neurons, and multiple inputs, it is cumbersome, we need a better frame work, aka Matrix!