CPE 486/586: Deep Learning for Engineering Applications

03 Introducing Deep Learning: Neural Networks

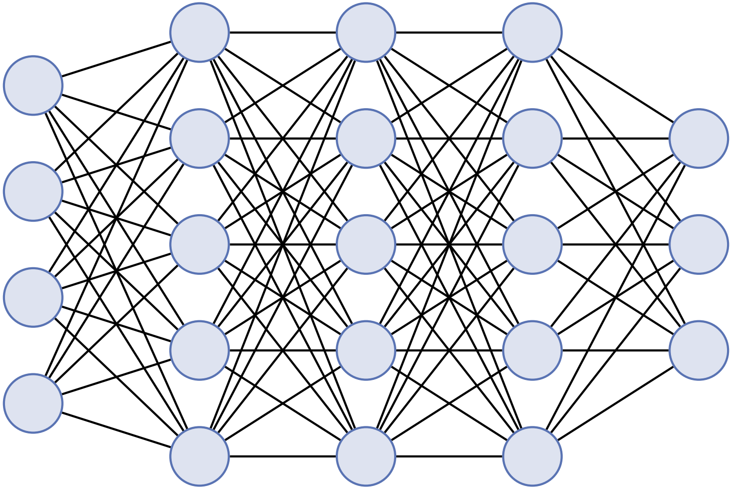
Spring 2026

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

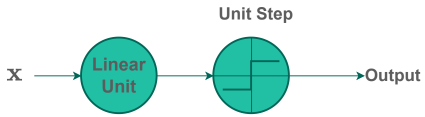
$$y = \sum_{i=0}^{n} w_i x_i$$

or

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

Perceptron

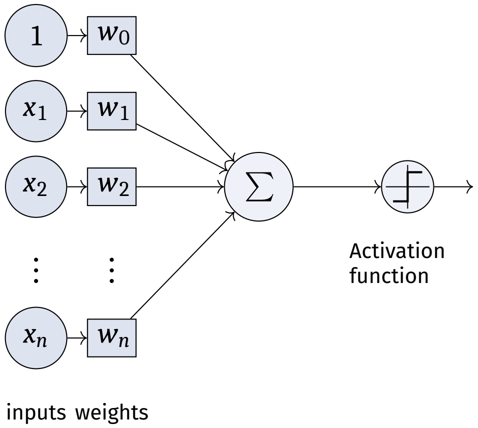
A neuron with unit-step for decision making:



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):

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

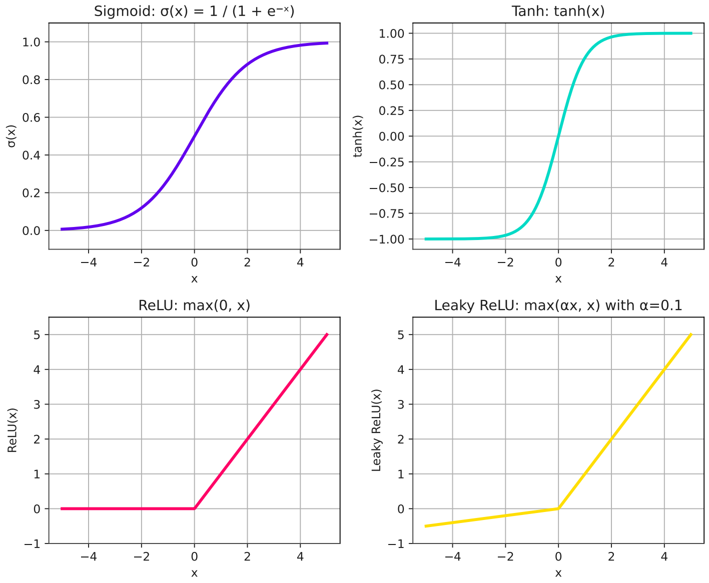
Suitable for regression problems. Range between 0 to ∞.

2. Leaky ReLU:

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

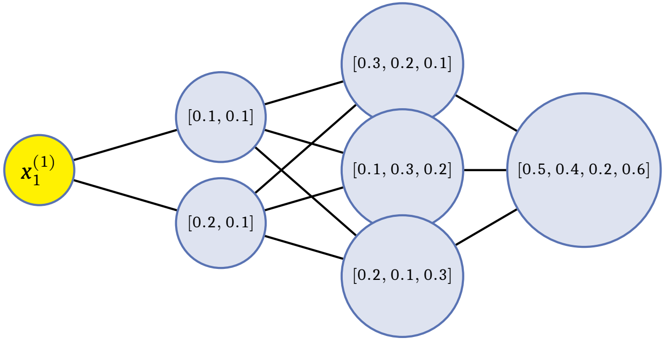
Suitable for regression problems. It solves the problem of dead neuron suffered while ReLU is used, i.e., if a neuron receives only negative inputs, it outputs zero and its gradient becomes zero. This means it stops learning. Leaky ReLU is a modified version of ReLU designed to fix the problem of dead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

$$\begin{aligned}
\frac{\partial L}{\partial a_1^{[1]}} &= \sum_{j=1}^{3} \frac{\partial L}{\partial z_j^{[2]}} \cdot w_{1j}^{[2]} \\
&= (-0.0214 \times 0.2) + (-0.0109 \times 0.3) + (-0.0323 \times 0.1) \\
&= -0.00428 - 0.00327 - 0.00323 = -0.01078 \\
\frac{\partial L}{\partial a_2^{[1]}} &= (-0.0214 \times 0.1) + (-0.0109 \times 0.2) + (-0.0323 \times 0.3) \\
&= -0.00214 - 0.00218 - 0.00969 = -0.01401
\end{aligned}$$

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:

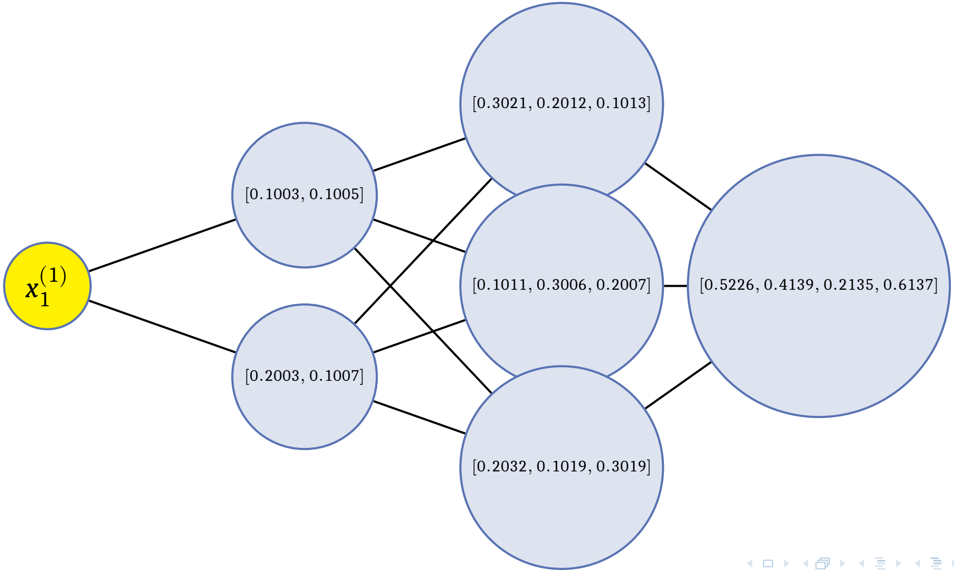
$$\Delta L = 0.2563 - 0.2455 = 0.0108 \quad \text{(4.23\% reduction)}$$

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!

Gradient

Definition

Vector of partial derivatives:

$$\nabla f(x,y) = \left[\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}\right]$$

This is about vector calculus, but in a Neural Network, we not only have one function of many variables but many functions of many variables, $\rightarrow$ Matrix calculus.

Jacobian

If we have two functions f and g, then

$$J = \begin{bmatrix} \nabla f(x, y) \\ \nabla g(x, y) \end{bmatrix} = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} & \frac{\partial f(x,y)}{\partial y} \\ \frac{\partial g(x,y)}{\partial x} & \frac{\partial g(x,y)}{\partial y} \end{bmatrix}$$

Generalization

Assume $f(x, y, z) \Rightarrow f(\mathbf{x})$ Or in general, $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \end{bmatrix}$

- Scalar-valued functions can be written as vectors $\mathbf{y} = \mathbf{f}(\mathbf{x})$.

So,

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \nabla f_1(\mathbf{x}) \\ \nabla f_2(\mathbf{x}) \\ \dots \\ \nabla f_m(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial \mathbf{x}} f_1(\mathbf{x}) \\ \frac{\partial}{\partial \mathbf{x}} f_2(\mathbf{x}) \\ \dots \\ \frac{\partial}{\partial \mathbf{x}} f_m(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1} f_1(\mathbf{x}) & \frac{\partial}{\partial x_2} f_1(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n} f_1(\mathbf{x}) \\ \frac{\partial}{\partial x_1} f_2(\mathbf{x}) & \frac{\partial}{\partial x_2} f_2(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n} f_2(\mathbf{x}) \\ \dots & \dots & \dots & \dots \\ \frac{\partial}{\partial x_1} f_m(\mathbf{x}) & \frac{\partial}{\partial x_2} f_m(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n} f_m(\mathbf{x}) \end{bmatrix}$$

Some Notes on Generalization

- $\frac{\partial}{\partial \mathbf{x}} f_i(\mathbf{x})$ is a horizontal (row) n-vector.
- Width of Jacobian is n and have m rows.

Classwork

Derive the Jacobian of identify function $\mathbf{f}(\mathbf{x}) = \mathbf{x}$, i.e. $f_i(\mathbf{x}) = x_i$ that has n functions and each function has n parameters held in a single vector $\mathbf{x}$.

Element-wise Binary Operators

By ”element-wise binary operations” we mean applying an operator to the first item of each vector to get the first item of the output, then to the second items of the inputs for the second item of the output, and so forth.

Element-wise binary operation: $\mathbf{y} = \mathbf{f}(\mathbf{w}) \bigcirc \mathbf{g}(\mathbf{x})$
Consider $m = n$ for this case.
Essentially, in scalar format:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x}) \\ f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x}) \end{bmatrix}$$

Elementwise implies that f_i is purely a function of w_i and g_i is purely a function of x_i.

General Case of Element-wise Binary Operators

Jacobian with respect to $\mathbf{w}$ can be expanded as

$$J_{\mathbf{w}} = \frac{\partial \mathbf{y}}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial}{\partial w_1}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \frac{\partial}{\partial w_2}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \dots & \frac{\partial}{\partial w_n}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) \\ \frac{\partial}{\partial w_1}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \frac{\partial}{\partial w_2}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \dots & \frac{\partial}{\partial w_n}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) \\ \dots & \dots & \dots & \dots \\ \frac{\partial}{\partial w_1}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \frac{\partial}{\partial w_2}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \dots & \frac{\partial}{\partial w_n}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) \end{bmatrix}$$

General Case of Element-wise Binary Operators

Jacobian with respect to $\mathbf{x}$ can be expanded as

$$J_{\mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial}{\partial x_1}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \frac{\partial}{\partial x_2}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \dots & \frac{\partial}{\partial x_n}(f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) \\ \frac{\partial}{\partial x_1}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \frac{\partial}{\partial x_2}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \dots & \frac{\partial}{\partial x_n}(f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) \\ \dots & \dots & \dots & \dots \\ \frac{\partial}{\partial x_1}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \frac{\partial}{\partial x_2}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \dots & \frac{\partial}{\partial x_n}(f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) \end{bmatrix}$$

The good news that most of the time, Jacobian will be a diagonal matrix that greatly simplifies the calculation!!

Classwork

Under what conditions, are the off-diagonal elements of a Jacobian zero?

Example

Consider $\mathbf{f}(\mathbf{w}) = \mathbf{w}$ and $\mathbf{g}(\mathbf{x}) = \mathbf{x}$. $\mathbf{f}(\mathbf{w}) + \mathbf{g}(\mathbf{x})$ has scalar equations $y_i = f_i(\mathbf{w}) + g_i(\mathbf{x})$ which reduces to $y_i = f_i(w_i) + g_i(x_i) = w_i + x_i$. We can get partial derivatives as

$$\frac{\partial}{\partial w_i}(f_i(w_i) + g_i(x_i)) = \frac{\partial}{\partial w_i}(w_i + x_i) = 1 + 0 = 1$$
$$\frac{\partial}{\partial x_i}(f_i(w_i) + g_i(x_i)) = \frac{\partial}{\partial x_i}(w_i + x_i) = 0 + 1 = 1$$

This gives us $\frac{\partial(\mathbf{w}+\mathbf{x})}{\partial \mathbf{w}} = \frac{\partial(\mathbf{w}+\mathbf{x})}{\partial \mathbf{x}} = \mathbf{I}$, the identity matrix.

Results of Common Elementwise Binary Operations

Operation	**Partial with respect to w**	**Result**
$+$	$\frac{\partial(\mathbf{w}+\mathbf{x})}{\partial\mathbf{w}} = \text{diag}\left(\ldots \frac{\partial(w_i+x_i)}{\partial w_i} \ldots\right) = \text{diag}(\vec{1})$	I
$-$	$\frac{\partial(\mathbf{w}-\mathbf{x})}{\partial\mathbf{w}} = \text{diag}\left(\ldots \frac{\partial(w_i-x_i)}{\partial w_i} \ldots\right) = \text{diag}(\vec{1})$	I
$\otimes$	$\frac{\partial(\mathbf{w}\otimes\mathbf{x})}{\partial\mathbf{w}} = \text{diag}\left(\ldots \frac{\partial(w_i\times x_i)}{\partial w_i} \ldots\right)$	$\text{diag}(\mathbf{x})$
$\oslash$	$\frac{\partial(\mathbf{w}\oslash\mathbf{x})}{\partial\mathbf{w}} = \text{diag}\left(\ldots \frac{\partial(w_i/x_i)}{\partial w_i} \ldots\right)$	$\text{diag}\left(\ldots \frac{1}{x_i} \ldots\right)$

$\otimes$ is also called the Hadamard product.

Vector-Scalar Operation

What about adding scalar z to vector $\mathbf{x}$?

Vector-Scalar Operation

What about adding scalar z to vector $\mathbf{x}$?

z gets expanded to vector with appropriate length to add to elements of $\mathbf{x}$.

In essence,

$$\begin{aligned} z = \mathbf{g}(z) &= \vec{1}z \\ \mathbf{y} = \mathbf{f}(\mathbf{x}) &+ \mathbf{g}(z) \end{aligned}$$

Vector-Scalar Operation

What about adding scalar z to vector $\mathbf{x}$?

z gets expanded to vector with appropriate length to add to elements of $\mathbf{x}$.

In essence,

$$\begin{aligned} z &= \mathbf{g}(z) = \vec{1}z \\ \mathbf{y} &= \mathbf{f}(\mathbf{x}) + \mathbf{g}(z) \end{aligned}$$

And then you can apply our previous discussion. $\vec{1}$ means a vertical vector (column vector of all 1s).

Note: Computing the partial derivative with respect to the scalar parameter however results in a vertical vector and not a diagonal matrix.

Vector Sum Reduction and Calculus

Consider

$$y = \sum_{i=1}^{n} f_i(\mathbf{x})$$

The sume is over the results of the function, and not the parameter itself. Then,

$$\begin{aligned}
\frac{\partial y}{\partial \mathbf{x}} &= \left[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_n}\right] \\
&= \left[\frac{\partial}{\partial x_1}\sum_i f_i(\mathbf{x}), \frac{\partial}{\partial x_2}\sum_i f_i(\mathbf{x}), \ldots, \frac{\partial}{\partial x_n}\sum_i f_i(\mathbf{x})\right] \\
&= \left[\sum_i \frac{\partial f_i(\mathbf{x})}{\partial x_1}, \sum_i \frac{\partial f_i(\mathbf{x})}{\partial x_2}, \ldots, \sum_i \frac{\partial f_i(\mathbf{x})}{\partial x_n}\right] \quad \text{(move derivative inside } \Sigma\text{)}
\end{aligned}$$

Single Variable Chain Rule

$$
\begin{aligned}
y &= f(g(x)) \\
y' &= f'(g(x))g'(x)
\end{aligned}
$$

We also rewrite by introducing an intermediate variable $u = g(x)$, so

$$
\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}
$$

Example: Use chain rule for $\sin(x^2)$ to find its derivative wrt x.

Law of Total Derivatives

To compute $\frac{dy}{dx}$, we need to sum up all possible contributions from changes in x to the change in y.

Assume $u_1 = g(x)$, and $y = f(x) = u_2(x, u_1)$, then

$$\begin{aligned}\frac{dy}{dx} = \frac{\partial f(x)}{\partial x} &= \frac{\partial u_2(x, u_1)}{\partial x} \\ &= \frac{\partial u_2}{\partial x}\frac{\partial x}{\partial x} + \frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial x} \\ &= \frac{\partial u_2}{\partial x} + \frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial x}\end{aligned}$$

Law of Total Derivatives

In general

$$\begin{aligned}\frac{\partial f(x, u_1, \ldots, u_n)}{\partial x} &= \frac{\partial f}{\partial x} + \frac{\partial f}{\partial u_1}\frac{\partial u_1}{\partial x} + \frac{\partial f}{\partial u_2}\frac{\partial u_2}{\partial x} + \cdots + \frac{\partial f}{\partial u_n}\frac{\partial u_n}{\partial x} \\ &= \frac{\partial f}{\partial x} + \sum_{i=1}^{n} \frac{\partial f}{\partial u_i}\frac{\partial u_i}{\partial x}\end{aligned}$$

This is called **Single-variable Total-derivative Chain Rule**.

The total derivative assumes all variables are potentially codependent whereas the partial derivative assumes all variables but x are constants.

Could we simplify the above expression further?

Vector Chain Rule

$$\begin{bmatrix} y_1(x) \\ y_2(x) \end{bmatrix} = \begin{bmatrix} f_1(x) \\ f_2(x) \end{bmatrix} = \begin{bmatrix} ln(x^2) \\ sin(3x) \end{bmatrix}$$

Consider $\mathbf{y} = \mathbf{f}(\mathbf{g}(x))$ with g_1 and g_2 as two intermediate variables:

$$\begin{bmatrix} g_1(x) \\ g_2(x) \end{bmatrix} = \begin{bmatrix} x^2 \\ 3x \end{bmatrix}$$

$$\begin{bmatrix} f_1(\mathbf{g}) \\ f_2(\mathbf{g}) \end{bmatrix} = \begin{bmatrix} ln(g_1) \\ sin(g_2) \end{bmatrix}$$

$$\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial f_1(\mathbf{g})}{\partial x} \\ \frac{\partial f_2(\mathbf{g})}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial g_1}\frac{\partial g_1}{\partial x} + \frac{\partial f_1}{\partial g_2}\frac{\partial g_2}{\partial x} \\ \frac{\partial f_2}{\partial g_1}\frac{\partial g_1}{\partial x} + \frac{\partial f_2}{\partial g_2}\frac{\partial g_2}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{1}{g_1}2x + 0 \\ 0 + \cos(g_2)3 \end{bmatrix} = \begin{bmatrix} \frac{2x}{x^2} \\ 3\cos(3x) \end{bmatrix} = \begin{bmatrix} \frac{2}{x} \\ 3\cos(3x) \end{bmatrix}$$

Vector Chain Rule

Still, the goal is convert group of scalar operations into a vector operations.

$$\begin{bmatrix} \frac{\partial f_1}{\partial g_1}\frac{\partial g_1}{\partial x} + \frac{\partial f_1}{\partial g_2}\frac{\partial g_2}{\partial x} \\ \frac{\partial f_2}{\partial g_1}\frac{\partial g_1}{\partial x} + \frac{\partial f_2}{\partial g_2}\frac{\partial g_2}{\partial x} \end{bmatrix}$$

could be written as

$$\begin{bmatrix} \frac{\partial f_1}{\partial g_1} & \frac{\partial f_1}{\partial g_2} \\ \frac{\partial f_2}{\partial g_1} & \frac{\partial f_2}{\partial g_2} \end{bmatrix} \begin{bmatrix} \frac{\partial g_1}{\partial x} \\ \frac{\partial g_2}{\partial x} \end{bmatrix} \overset{\text{set}}{=} \frac{\partial \mathbf{f}}{\partial \mathbf{g}} \frac{\partial \mathbf{g}}{\partial x}$$

by matrix multiplication. So the Jacobian is the multiplication of two other Jacobians.

Vector Chain Rule

Instead of a scalar x, if we consider $\mathbf{x}$,

colback=powderGreen, colframe=cpelime

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{f}(\mathbf{g}(\mathbf{x})) = \frac{\partial \mathbf{f}}{\partial \mathbf{g}}\frac{\partial \mathbf{g}}{\partial \mathbf{x}}$$

Note: matrix multiply doesn't commute; order of $\frac{\partial \mathbf{f}}{\partial \mathbf{g}}\frac{\partial \mathbf{g}}{\partial \mathbf{x}}$ matters.

Vector Chain Rule

Expanding $\frac{\partial}{\partial \mathbf{x}}\mathbf{f}(\mathbf{g}(\mathbf{x}))$,

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{f}(\mathbf{g}(\mathbf{x})) = \begin{bmatrix} \frac{\partial f_1}{\partial g_1} & \frac{\partial f_1}{\partial g_2} & \cdots & \frac{\partial f_1}{\partial g_k} \\ \frac{\partial f_2}{\partial g_1} & \frac{\partial f_2}{\partial g_2} & \cdots & \frac{\partial f_2}{\partial g_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial g_1} & \frac{\partial f_m}{\partial g_2} & \cdots & \frac{\partial f_m}{\partial g_k} \end{bmatrix} \begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_k}{\partial x_1} & \frac{\partial g_k}{\partial x_2} & \cdots & \frac{\partial g_k}{\partial x_n} \end{bmatrix}$$

where $m = |f|$, $n = |x|$, and $k = |g|$, the resulting Jacobian matrix has dimensions $m \times n$ (derived from the multiplication of an $m \times k$ matrix by a $k \times n$ matrix). $|\cdot|$ denotes cardinality.

When will the off-diagonal entries be zero for both matrices? How does that simplify our calculations?

Matrix Calculus in Action

Gradient of Neuron Activation

We consider ReLU activation for simplicity.

$$activation(x) = max(0, \mathbf{w} \cdot \mathbf{x} + b)$$

Focusing on $\frac{\partial(\mathbf{w} \cdot \mathbf{x} + b)}{\partial \mathbf{w}}$, the dot product $\mathbf{w} \cdot \mathbf{x}$ is elementwise multiplication that can be written as $sum(\mathbf{w} \otimes \mathbf{x})$.

Gradient of Neuron Activation

$$\boldsymbol{u} = \boldsymbol{w} \otimes \boldsymbol{x}$$
$$y = \text{sum}(\boldsymbol{u})$$

Once we've rephrased y, we recognize two subexpressions with known partial derivatives:

$$\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{w}} = \frac{\partial}{\partial \boldsymbol{w}}(\boldsymbol{w} \otimes \boldsymbol{x}) = \text{diag}(\boldsymbol{x})$$
$$\frac{\partial y}{\partial \boldsymbol{u}} = \frac{\partial}{\partial \boldsymbol{u}}\text{sum}(\boldsymbol{u}) = \boldsymbol{1}^T$$

The vector chain rule says to multiply the partials:

$$\frac{\partial y}{\partial \boldsymbol{w}} = \frac{\partial y}{\partial \boldsymbol{u}}\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{w}} = \boldsymbol{1}^T\text{diag}(\boldsymbol{x}) = \boldsymbol{x}^T$$

Verification via Scalar Expansion

To check our results, we can grind the dot product down into a pure scalar function:

$$y = \mathbf{w} \cdot \mathbf{x} = \sum_i^n (w_i x_i)$$

$$\frac{\partial y}{\partial w_j} = \frac{\partial}{\partial w_j} \sum_i (w_i x_i) = \sum_i \frac{\partial}{\partial w_j} (w_i x_i) = \frac{\partial}{\partial w_j} (w_j x_j) = x_j$$

Then:

$$\frac{\partial y}{\partial \mathbf{w}} = [x_1, \ldots, x_n] = \mathbf{x}^T$$

Hooray! Our scalar results match the vector chain rule results.

Full Linear Layer with Bias

Now, let $y = \boldsymbol{w} \cdot \boldsymbol{x} + b$, the full expression within the $\max$ activation function call. We have two different partials to compute, but we don't need the chain rule:

$$\frac{\partial y}{\partial \boldsymbol{w}} = \frac{\partial}{\partial \boldsymbol{w}} \boldsymbol{w} \cdot \boldsymbol{x} + \frac{\partial}{\partial \boldsymbol{w}} b = \boldsymbol{x}^T + \boldsymbol{0}^T = \boldsymbol{x}^T$$

$$\frac{\partial y}{\partial b} = \frac{\partial}{\partial b} \boldsymbol{w} \cdot \boldsymbol{x} + \frac{\partial}{\partial b} b = 0 + 1 = 1$$

Including Nonlinearity of the Activation Function

Let's tackle the partials of the neuron activation, $\max(0, \boldsymbol{w} \cdot \boldsymbol{x} + b)$. The use of the $\max(0, z)$ function call on scalar z just says to treat all negative z values as 0.

The derivative of the max function is a piecewise function:

- When $z \leq 0$, the derivative is 0 because z is a constant
- When $z > 0$, the derivative of the max function is just the derivative of z, which is 1

$$\frac{\partial}{\partial z} \max(0, z) = \begin{cases} 0 & z \leq 0 \\ \frac{dz}{dz} = 1 & z > 0 \end{cases}$$

Broadcasting Functions Across Vectors

An aside on broadcasting functions across scalars. When one or both of the $\max$ arguments are vectors, such as $\max(0, \boldsymbol{x})$, we broadcast the single-variable function $\max$ across the elements. This is an example of an element-wise unary operator.

$$\max(0, \boldsymbol{x}) = \begin{bmatrix} \max(0, x_1) \\ \max(0, x_2) \\ \vdots \\ \max(0, x_n) \end{bmatrix}$$

Derivative of Broadcast ReLU

For the derivative of the broadcast version then, we get a vector of zeros and ones where:

$$\frac{\partial}{\partial x_i} \max(0, x_i) = \begin{cases} 0 & x_i \leq 0 \\ \frac{dx_i}{dx_i} = 1 & x_i > 0 \end{cases}$$

$$\frac{\partial}{\partial \boldsymbol{x}} \max(0, \boldsymbol{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} \max(0, x_1) \\ \frac{\partial}{\partial x_2} \max(0, x_2) \\ \vdots \\ \frac{\partial}{\partial x_n} \max(0, x_n) \end{bmatrix}$$

Applying the Chain Rule to Activation

To get the derivative of the activation$(\boldsymbol{x})$ function, we need the chain rule because of the nested subexpression, $\boldsymbol{w} \cdot \boldsymbol{x} + b$.

Following our process, let's introduce intermediate scalar variable z to represent the affine function giving:

$$\begin{aligned} z(\boldsymbol{w}, b, \boldsymbol{x}) &= \boldsymbol{w} \cdot \boldsymbol{x} + b \\ \text{activation}(z) &= \max(0, z) \end{aligned}$$

Applying the Chain Rule to Activation

$$\frac{\partial \text{activation}}{\partial \boldsymbol{w}} = \frac{\partial \text{activation}}{\partial z}\frac{\partial z}{\partial \boldsymbol{w}}$$

which we can rewrite as follows:

$$\frac{\partial \text{activation}}{\partial \boldsymbol{w}} = \begin{cases} 0 \cdot \frac{\partial z}{\partial \boldsymbol{w}} = \boldsymbol{0}^T & z \leq 0 \\ 1 \cdot \frac{\partial z}{\partial \boldsymbol{w}} = \frac{\partial z}{\partial \boldsymbol{w}} = \boldsymbol{x}^T & z > 0 \end{cases} \quad \text{(we computed } \frac{\partial z}{\partial \boldsymbol{w}} = \boldsymbol{x}^T \text{ previously)}$$

and then substitute $z = \boldsymbol{w} \cdot \boldsymbol{x} + b$ back in:

$$\frac{\partial \text{activation}}{\partial \boldsymbol{w}} = \begin{cases} \boldsymbol{0}^T & \boldsymbol{w} \cdot \boldsymbol{x} + b \leq 0 \\ \boldsymbol{x}^T & \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \end{cases}$$

When the activation function clips affine function output z to 0, the derivative is zero with respect to any weight w_i. When $z > 0$, it's as if the $\max$ function disappears and we get just the derivative of z with respect to the weights.

Derivative with Respect to Bias

Turning now to the derivative of the neuron activation with respect to b, we get:

$$\frac{\partial \text{activation}}{\partial b} = \begin{cases} 0 \cdot \frac{\partial z}{\partial b} = 0 & \boldsymbol{w} \cdot \boldsymbol{x} + b \leq 0 \\ 1 \cdot \frac{\partial z}{\partial b} = 1 & \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \end{cases}$$

Gradient of Neural Network Loss

Feature matrix: $X = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \cdots, \mathbf{x}_N]^\top$
$\mathbf{y} = [y_1, y_2, \cdots, y_N]$.
where y_i is a scalar.

Cost function:

$$C(\boldsymbol{w}, b, X, \boldsymbol{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \text{activation}(\boldsymbol{x}_i))^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \max(0, \boldsymbol{w}\cdot\boldsymbol{x}_i + b))^2$$

Following our chain rule process we introduce these intermediate variables:

$$\begin{aligned}
u(\boldsymbol{w}, b, \boldsymbol{x}) &= \max(0, \boldsymbol{w} \cdot \boldsymbol{x} + b) \\
v(y, u) &= y - u \\
C(v) &= \frac{1}{N} \sum_{i=1}^{N} v^2
\end{aligned}$$

Gradient Computation

Recall

$$\frac{\partial}{\partial \boldsymbol{w}} u(\boldsymbol{w}, b, \boldsymbol{x}) = \begin{cases} \boldsymbol{0}^T & \boldsymbol{w} \cdot \boldsymbol{x} + b \leq 0 \\ \boldsymbol{x}^T & \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \end{cases}$$

and

$$\frac{\partial v(y, u)}{\partial \boldsymbol{w}} = \frac{\partial}{\partial \boldsymbol{w}}(y - u) = \boldsymbol{0}^T \frac{\partial u}{\partial \boldsymbol{w}} = -\frac{\partial u}{\partial \boldsymbol{w}} = \begin{cases} \boldsymbol{0}^T & \boldsymbol{w} \cdot \boldsymbol{x} + b \leq 0 \\ -\boldsymbol{x}^T & \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \end{cases}$$

Gradient Computation

$$\frac{\partial C(v)}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}} \frac{1}{N} \sum_{i=1}^{N} v_i^2 = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial}{\partial \mathbf{w}} v_i^2 = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial v_i^2}{\partial v} \frac{\partial v}{\partial \mathbf{w}} = \frac{1}{N} \sum_{i=1}^{N} 2v_i \frac{\partial v}{\partial \mathbf{w}}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \begin{cases} 2v_i \mathbf{0}^T = \mathbf{0}^T & \mathbf{w} \cdot \mathbf{x}_i + b \leq 0 \\ -2v_i \mathbf{x}^T & \mathbf{w} \cdot \mathbf{x}_i + b > 0 \end{cases}$$

Gradient Computation

$$\frac{\partial C(v)}{\partial \boldsymbol{w}} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}\mathbf{0}^T & \boldsymbol{w}\cdot\boldsymbol{x}_i + b \leq 0\\ -2(y_i - u)\boldsymbol{x}_i^T & \boldsymbol{w}\cdot\boldsymbol{x}_i + b > 0\end{cases} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}\mathbf{0}^T & \boldsymbol{w}\cdot\boldsymbol{x}_i + b \leq 0\\ -2(y_i - \max(0, \boldsymbol{w}\cdot\boldsymbol{x}_i + b))\boldsymbol{x}_i^T & \boldsymbol{w}\cdot\boldsymbol{x}_i + b > 0\end{cases}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\begin{cases}\mathbf{0}^T & \boldsymbol{w}\cdot\boldsymbol{x}_i + b \leq 0\\ -2(y_i - (\boldsymbol{w}\cdot\boldsymbol{x}_i + b))\boldsymbol{x}_i^T & \boldsymbol{w}\cdot\boldsymbol{x}_i + b > 0\end{cases} = \begin{cases}\mathbf{0}^T & \boldsymbol{w}\cdot\boldsymbol{x}_i + b \leq 0\\ -\frac{2}{N}\sum_{i=1}^{N}(y_i - (\boldsymbol{w}\cdot\boldsymbol{x}_i + b))\boldsymbol{x}_i^T & \boldsymbol{w}\cdot\boldsymbol{x}_i + b > 0\end{cases}$$

$$= \begin{cases}\mathbf{0}^T & \boldsymbol{w}\cdot\boldsymbol{x}_i + b \leq 0\\ \frac{2}{N}\sum_{i=1}^{N}(\boldsymbol{w}\cdot\boldsymbol{x}_i + b - y_i)\boldsymbol{x}_i^T & \boldsymbol{w}\cdot\boldsymbol{x}_i + b > 0\end{cases}$$

We can substitute an error term $e_i = \boldsymbol{w}\cdot\boldsymbol{x}_i + b - y_i$ yielding:

$$\frac{\partial C}{\partial \boldsymbol{w}} = \frac{2}{N}\sum_{i=1}^{N} e_i\boldsymbol{x}_i^T \quad \text{(for the nonzero activation case)}$$

Gradient Computation

What about the bias?
Intermediate variables:

$$\begin{aligned} u(\boldsymbol{w}, b, \boldsymbol{x}) &= \max(0, \boldsymbol{w} \cdot \boldsymbol{x} + b) \\ v(y, u) &= y - u \\ C(v) &= \frac{1}{N} \sum_{i=1}^{N} v^2 \end{aligned}$$

From previous result:

$$\frac{\partial u}{\partial b} = \begin{cases} 0 & \boldsymbol{w} \cdot \boldsymbol{x} + b \leq 0 \\ 1 & \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \end{cases}$$

Gradient Computation for Bias

For v, the partial is:

$$\frac{\partial v(y,u)}{\partial b} = \frac{\partial}{\partial b}(y-u) = 0 - \frac{\partial u}{\partial b} = -\frac{\partial u}{\partial b} = \begin{cases} 0 & \boldsymbol{w}\cdot\boldsymbol{x}+b \leq 0 \\ -1 & \boldsymbol{w}\cdot\boldsymbol{x}+b > 0 \end{cases}$$

Gradient Computation for Bias

$$\frac{\partial C(v)}{\partial b} = \frac{\partial}{\partial b}\frac{1}{N}\sum_{i=1}^{N} v_i^2 = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial}{\partial b}v_i^2 = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial v_i^2}{\partial v}\frac{\partial v}{\partial b}$$

$$= \frac{1}{N}\sum_{i=1}^{N} 2v_i\frac{\partial v}{\partial b}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\begin{cases} 0 & \boldsymbol{w}\cdot\boldsymbol{x} + b \leq 0 \\ -2v_i & \boldsymbol{w}\cdot\boldsymbol{x} + b > 0 \end{cases}$$

Gradient Computation for Bias

$$\frac{\partial C(v)}{\partial b} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases} 0 & \boldsymbol{w}\cdot\boldsymbol{x}+b \le 0 \\ -2(y_i - \max(0, \boldsymbol{w}\cdot\boldsymbol{x}_i + b)) & \boldsymbol{w}\cdot\boldsymbol{x}+b > 0 \end{cases}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\begin{cases} 0 & \boldsymbol{w}\cdot\boldsymbol{x}+b \le 0 \\ 2(\boldsymbol{w}\cdot\boldsymbol{x}_i + b - y_i) & \boldsymbol{w}\cdot\boldsymbol{x}+b > 0 \end{cases}$$

$$= \begin{cases} 0 & \boldsymbol{w}\cdot\boldsymbol{x}+b \le 0 \\ \frac{2}{N}\sum_{i=1}^{N}(\boldsymbol{w}\cdot\boldsymbol{x}_i + b - y_i) & \boldsymbol{w}\cdot\boldsymbol{x}+b > 0 \end{cases}$$

Error Term Substitution for Bias Gradient

As before, we can substitute an error term:

$$\frac{\partial C}{\partial b} = \frac{2}{N} \sum_{i=1}^{N} e_i \quad \text{(for the nonzero activation case)}$$

where $e_i = \boldsymbol{w} \cdot \boldsymbol{x}_i + b - y_i$ is the prediction error.

The bias gradient is proportional to the mean prediction error across all samples when the neuron is active.

Scalar vs. Vectorized Operations

Operation	**Scalar (Loop)**	**Vectorized**
Affine transformation	$z_i = \mathbf{x}_i^T \boldsymbol{w} + b$ One sample at a time	$\boldsymbol{z} = X\boldsymbol{w} + b$ All n samples at once
Activation	$u_i = \max(0, z_i)$ Scalar function	$\boldsymbol{u} = \max(0, \boldsymbol{z})$ Element-wise operation
Loss	$C = \frac{1}{N}\sum_i v_i^2$ Accumulate in loop	$C = \frac{1}{N}\lVert\boldsymbol{v}\rVert_2^2$ Vectorized norm
Gradient $\boldsymbol{w}$	$\frac{\partial C}{\partial w_j} = \frac{2}{N}\sum_i x_{ij} e_i$ Nested loops	$\frac{\partial C}{\partial \boldsymbol{w}} = \frac{2}{N} X^T \boldsymbol{e}$ Single matrix product
Gradient b	$\frac{\partial C}{\partial b} = \frac{2}{N}\sum_i e_i$ Single loop	$\frac{\partial C}{\partial b} = \frac{2}{N}\mathbf{1}^T\boldsymbol{e}$ Dot product

N: Number of samples.

References/Readings

Mandatory Reading

1. The Matrix Calculus You Need For Deep Learning https://arxiv.org/pdf/1802.01528

Recommended Reading

1. https://lrjconan.github.io/UBC-CPEN455-DL/assets/slides_2025/mlp.pdf
2. https://www.csie.ntu.edu.tw/~cjlin/papers/autodiff/autodiff.pdf
3. https://comp6248.ecs.soton.ac.uk/handouts/autograd-handouts.pdf
4. https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2022/tutorials/tut01.pdf
5. https://xiucheng.org/assets/pdfs/matrix-calculus4ml.pdf
6. https://robotchinwag.com/posts/gradient-of-matrix-multiplicationin-deep-learning/