Lecture 2: Introduction to Neural Networks; From Single Layer to Deep Networks

Justo E. Karell

September 19, 2024

CS 583: Deep Learning

Key Components of Neural Networks

A neural network is a computational model composed of several interconnected layers of nodes (or neurons). Each node represents a mathematical function that processes inputs, applies transformations, and produces outputs. The layers of a neural network are as follows:

1. **Input Layer:** The input layer receives the raw input data as vectors. The input vector $\mathbf{x}$ has dimensions $\mathbb{R}^n$, where n is the number of input features.

2. **Hidden Layers:** These layers process inputs from the previous layer through a series of transformations. Each hidden layer is composed of multiple nodes, each of which applies a weighted sum to the input and then passes it through an activation function to introduce non-linearity.

3. **Output Layer:** The final layer outputs the predictions or classifications of the network. Its structure depends on the task (e.g., regression or classification) and can vary in the number of output nodes.

4. **Weights and Biases:** Each connection between nodes in adjacent layers has an associated weight w, and each node has a bias b. These parameters are learned during training through backpropagation and gradient descent.

5. **Activation Function:** The activation function $\sigma(z)$ introduces non-linearity into the model. Common activation functions include the sigmoid function, hyperbolic tangent, and ReLU (Rectified Linear Unit).

6. **Loss Function:** The loss function measures the difference between the predicted output and the actual target. Common loss functions are Mean Squared Error (MSE) for regression and Cross-Entropy Loss for classification.

7. **Backpropagation and Gradient Descent:** The parameters (weights and biases) are updated through backpropagation. The gradients of the loss function with respect to the parameters are computed and used to update the weights via gradient descent:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}, \quad b \leftarrow b - \eta \frac{\partial L}{\partial b}$$

where η is the learning rate and L is the loss function.

Mathematically, for a neural network with L layers, the output of the i-th node in layer l is given by:

$$h_i^{(l)} = \sigma \left(\sum_{j=1}^{n_{l-1}} w_{ij}^{(l)} h_j^{(l-1)} + b_i^{(l)} \right)$$

Where:

- $h_j^{(l-1)}$ is the output from node j in the previous layer $(l-1)$.
- $w_{ij}^{(l)}$ is the weight from node j in layer $(l-1)$ to node i in layer l.
- $b_i^{(l)}$ is the bias for node i in layer l.
- σ is the activation function, applied element-wise.

Introduction to Nodes in Neural Networks

The most fundamental building block of a neural network is a node. In a mathematical sense, a node is a computational unit that processes data, combining the input with weights, adding a bias, and passing the result through an activation function. Importantly, nodes introduce non-linearity into the system via activation functions, making neural networks powerful tools for modeling complex relationships.

A node is often compared to a function in traditional machine learning models (e.g., linear regression). However, unlike traditional models, nodes in neural networks can incorporate non-linear transformations and dynamic updates to their parameters, which enables them to solve more complex problems.

Node in a Neural Network

Mathematically, a node in a neural network is defined as:

$$h = \sigma(\mathbf{w}^T \mathbf{x} + b)$$

Where:

- $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T \in \mathbb{R}^n$ is the input vector, a real-valued vector containing n features.
- $\mathbf{w} = [w_1, w_2, \ldots, w_n]^T \in \mathbb{R}^n$ is the weight vector, a real-valued vector of the same dimension as the input.
- $b \in \mathbb{R}$ is the bias term, a scalar.
- σ is the activation function, such as ReLU, sigmoid, or others, applied to the weighted sum.
- $h \in \mathbb{R}$ is the output of the node, a scalar value.

Representing NNs: From Linear Models to Nodes to Layers

In traditional linear regression, the model for predicting an output y is:

$$y = \mathbf{w}^T\mathbf{x} + b$$

This is a simple weighted sum of inputs and is inherently linear. In contrast, a neural network node extends this idea by adding an activation function σ, introducing non-linearity:

$$h = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

This allows neural networks to capture non-linear relationships between inputs and outputs.

Now let's extend this to show a complete hidden layer of multiple nodes, each processing the input in parallel. Suppose we have a hidden layer with 3 nodes:

For Node 1:

$$h_1 = \sigma(w_{11}x_1 + w_{12}x_2 + \cdots + w_{1n}x_n + b_1)$$

For Node 2:

$$h_2 = \sigma(w_{21}x_1 + w_{22}x_2 + \cdots + w_{2n}x_n + b_2)$$

For Node 3:

$$h_3 = \sigma(w_{31}x_1 + w_{32}x_2 + \cdots + w_{3n}x_n + b_3)$$

We represent a layer has just this vector of nodes, or those functions representing the nodes.

$$\mathbf{h} = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}$$

Taking it a step further, we can represent our layer as the product of a matrix of weights with an input vector plus some bias vector:

$$\mathbf{h} = \sigma\left(\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ w_{31} & w_{32} & \cdots & w_{3n} \end{bmatrix} \mathbf{x} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right)$$

Where $\mathbf{h}$ is the output vector of the hidden layer, representing the output of each node in the layer after applying the activation function.

Thus, this demonstrates how multiple nodes, working in parallel, process the same input vector $\mathbf{x}$ and produce their outputs, which are then passed to the next layer (or the final output layer if this is the last hidden layer).

Representing NNs: Multiple Layers in Vector Notation

In the previous section, we saw how a single hidden layer in a neural network can be represented using matrix and vector notation. This same approach extends to neural networks with multiple layers, where the output of one layer becomes the input to the next layer.

A Layer l
For each layer l, we compute the output using the weight matrix $\mathbf{W}^{(l)}$, the input from the previous layer $\mathbf{h}^{(l-1)}$, the bias vector $\mathbf{b}^{(l)}$, and the activation function σ:

$$\mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

Where:

- $\mathbf{W}^{(l)}$ is the weight matrix for layer l, where each row corresponds to the weights for a node in layer l.
- $\mathbf{h}^{(l-1)}$ is the input vector from the previous layer, and for the first layer, this is the input data $\mathbf{x}$.
- $\mathbf{b}^{(l)}$ is the bias vector for layer l, with one bias per node.
- σ is the activation function, applied element-wise.

Multiple Layers $l > 1$
When we have multiple layers, the output of layer l is the input to the next layer $l + 1$. We can stack these computations for each layer as follows:

For the input layer:

$$\mathbf{h}^{(0)} = \mathbf{x}$$

For the first hidden layer (layer 1):

$$\mathbf{h}^{(1)} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$

For the second hidden layer (layer 2):

$$\mathbf{h}^{(2)} = \sigma\left(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right)$$

And so on, until we reach the output layer.

The output layer applies the same transformation as any other layer, but the structure of the output (e.g., regression or classification) dictates the number of output nodes and the activation function used. For example, for binary classification, we might use a sigmoid activation function in the output layer:

$$\hat{y} = \sigma\left(\mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}\right)$$

Where:

- $\hat{y}$ is the final prediction/output of the network.
- L is the total number of layers in the network.

Thus, this shows how each layer in a deep neural network transforms the input vector $\mathbf{x}$ step by step, until the final prediction $\hat{y}$ is produced at the output layer.

Counting Parameters in Neural Networks

Parameters in a Linear Equation

In traditional linear regression, a linear model for predicting an output y is given by:

$$y = \mathbf{w}^T\mathbf{x} + b$$

Where:

- $\mathbf{x} \in \mathbb{R}^n$ is the input vector with n features,
- $\mathbf{w} \in \mathbb{R}^n$ is the weight vector,
- $b \in \mathbb{R}$ is the bias (a scalar).

Counting Parameters in the Linear Equation:
The number of parameters in this linear equation is:

- n weights (one for each feature in $\mathbf{x}$),
- 1 bias term.

Thus, the total number of parameters is:

$$\text{Total parameters} = n + 1$$

For example, if $\mathbf{x}$ has 3 features (i.e., $n = 3$), the total number of parameters in the linear equation is:

$$\text{Total parameters} = 3 + 1 = 4$$

Parameters in a Single Neuron Layer

In a neural network, a single neuron (or node) operates similarly to a linear model, but with an added activation function:

$$h = \sigma(\mathbf{w}^T \mathbf{x} + b)$$

Where:

- $\mathbf{x} \in \mathbb{R}^n$ is the input vector,
- $\mathbf{w} \in \mathbb{R}^n$ is the weight vector,
- $b \in \mathbb{R}$ is the bias term,
- σ is the activation function (such as sigmoid, ReLU, etc.).

Counting Parameters in a Single Neuron:

Adding the activation function does not introduce any additional parameters. Therefore, the total number of parameters in a single neuron layer is the same as in a linear equation:

$$\text{Total parameters in a single neuron} = n + 1$$

For example, if $\mathbf{x}$ has 3 features (i.e., $n = 3$), the total number of parameters in a single neuron layer is:

$$\text{Total parameters in a single neuron} = 3 + 1 = 4$$

Parameters in a Layer with Multiple Neurons

When we extend this idea to a layer with multiple neurons, each neuron in the layer has its own set of weights and bias. For a layer with m neurons, the total number of parameters is calculated as follows:

For each neuron i in the layer:

$$h_i = \sigma\left(\sum_{j=1}^{n} w_{ij} x_j + b_i\right)$$

Where:

- w_{ij} represents the weight from input j to neuron i,
- b_i is the bias term for neuron i,
- σ is the activation function.

Counting Parameters in a Layer:
Each neuron has n weights and 1 bias. Therefore, for a layer with m neurons, the total number of parameters is:

$$\text{Total parameters in a layer} = m \times (n + 1)$$

Example: A Layer with 3 Neurons and Input Vector of Size 3:
If the input vector $\mathbf{x}$ has 3 features (i.e., $n = 3$) and the layer has 3 neurons (i.e., $m = 3$):

- Each neuron has 3 weights and 1 bias.
- Therefore, the total number of parameters for this layer is:

$$\text{Total parameters} = 3 \times (3 + 1) = 3 \times 4 = 12$$

Parameters in a Network with Two Layers

Now let's consider a neural network with two layers:

- The first layer has m_1 neurons and receives an input vector of size n_0.
- The second layer has m_2 neurons and receives an input from the first layer.

Counting Parameters in Two Layers:
The total number of parameters in the two-layer network is the sum of the parameters in both layers.

First Layer:

For each neuron i in the first layer:

$$h_i^{(1)} = \sigma\left(\sum_{j=1}^{n_0} w_{ij}^{(1)} x_j + b_i^{(1)}\right)$$

Where:

- Each of the m_1 neurons in the first layer has n_0 weights and 1 bias.
- Therefore, the total number of parameters in the first layer is:

$$\text{Total parameters in the first layer} = m_1 \times (n_0 + 1)$$

Second Layer:

For each neuron i in the second layer:

$$h_i^{(2)} = \sigma\left(\sum_{j=1}^{m_1} w_{ij}^{(2)} h_j^{(1)} + b_i^{(2)}\right)$$

Where:

- Each of the m_2 neurons in the second layer has m_1 weights (since the input comes from the m_1 neurons of the first layer) and 1 bias.
- Therefore, the total number of parameters in the second layer is:

$$\text{Total parameters in the second layer} = m_2 \times (m_1 + 1)$$

Total Parameters in the Network:

The total number of parameters in the two-layer network is the sum of the parameters from both layers:

$$\text{Total parameters in the network} = [m_1 \times (n_0 + 1)] + [m_2 \times (m_1 + 1)]$$

Example: Two-Layer Network

Consider a network where:

- The input vector $\mathbf{x}$ has 3 features (i.e., $n_0 = 3$),
- The first hidden layer has 4 neurons (i.e., $m_1 = 4$),
- The second hidden layer has 2 neurons (i.e., $m_2 = 2$).

First Layer: For each neuron i in the first layer:

$$h_i^{(1)} = \sigma\left(w_{i1}x_1 + w_{i2}x_2 + w_{i3}x_3 + b_i^{(1)}\right)$$

Where:

- Each neuron has 3 weights (since the input vector $\mathbf{x}$ has 3 features) and 1 bias.
- Therefore, the total number of parameters in the first layer is:

$$\text{Total parameters in the first layer} = 4 \times (3 + 1) = 4 \times 4 = 16$$

Second Layer: For each neuron i in the second layer:

$$h_i^{(2)} = \sigma\left(w_{i1}^{(2)} h_1^{(1)} + w_{i2}^{(2)} h_2^{(1)} + w_{i3}^{(2)} h_3^{(1)} + w_{i4}^{(2)} h_4^{(1)} + b_i^{(2)}\right)$$

Where:

- Each neuron in the second layer has 4 weights (since the input comes from the 4 neurons of the first layer) and 1 bias.
- Therefore, the total number of parameters in the second layer is:

$$\text{Total parameters in the second layer} = 2 \times (4 + 1) = 2 \times 5 = 10$$

The total number of parameters in the two-layer network is $16 + 10 = 26$

Summary

- The number of parameters in a neural network grows rapidly as we add more neurons and layers.
- For a single layer, the total parameters are $m \times (n + 1)$.
- For a two-layer network, the total parameters are $m_1 \times (n_0 + 1) + m_2 \times (m_1 + 1)$.

Example 1

In the example below, we compute the result of one iteration or one pass through the neural network. In a full training process, many such passes through the data would be performed. An epoch refers to a full pass over the entire dataset. During an epoch, the model would adjust its weights through backpropagation to minimize the loss function. Let's consider a basic input vector with two features:

$$\mathbf{x} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \in \mathbb{R}^2$$

With weights:

$$\mathbf{w} = \begin{bmatrix} 0.5 \\ -0.4 \end{bmatrix} \in \mathbb{R}^2$$

And bias:

$$b = 1 \in \mathbb{R}$$

For a simple node with a sigmoid activation function, the output is:

$$z = \mathbf{w}^T\mathbf{x} + b = 0.5(2) + (-0.4)(3) + 1 = 1.0$$

Applying the sigmoid function:

$$h = \frac{1}{1 + e^{-1.0}} \approx 0.731$$

The output $h \in \mathbb{R}$ is the node's result after one pass through this basic neural network node.

Example 2: One Iteration Through a Single Layer With One Neuron

Let's do the same thing as we just did: perform a forward pass through a simple single layer with only neuron :

- Input vector: $\mathbf{x} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \in \mathbb{R}^2$ - Weights: $\mathbf{w} = \begin{bmatrix} 0.7 \\ -0.5 \end{bmatrix} \in \mathbb{R}^2$ - Bias: $b = 0.1 \in \mathbb{R}$
- Activation function: Sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$

Calculation:

$$z = \mathbf{w}^T\mathbf{x} + b = (0.7 \times 2) + (-0.5 \times 3) + 0.1 = 1.4 - 1.5 + 0.1 = 0$$

$$h = \sigma(0) = \frac{1}{1+e^0} = 0.5$$

The output $h = 0.5 \in \mathbb{R}$.

Example 3: Two Neuron Layer (One Pass)

Consider a neural network with two neurons in the hidden layer:

- Input vector: $\mathbf{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \in \mathbb{R}^2$ - Weights for each neuron:

$$\mathbf{w}_1 = \begin{bmatrix} 0.6 \\ 0.3 \end{bmatrix} \in \mathbb{R}^2, \quad \mathbf{w}_2 = \begin{bmatrix} -0.4 \\ 0.9 \end{bmatrix} \in \mathbb{R}^2$$

- Biases: $b_1 = 0.2 \in \mathbb{R}, \quad b_2 = -0.1 \in \mathbb{R}$ - Activation function: ReLU

Calculation for neuron 1:

$$z_1 = \mathbf{w}_1^T \mathbf{x} + b_1 = 0.6(1) + 0.3(2) + 0.2 = 1.4$$

$$h_1 = \max(0, 1.4) = 1.4$$

Calculation for neuron 2:

$$z_2 = \mathbf{w}_2^T \mathbf{x} + b_2 = (-0.4)(1) + 0.9(2) - 0.1 = 1.3$$

$$h_2 = \max(0, 1.3) = 1.3$$

The output of the layer is:

$$\mathbf{h} = \begin{bmatrix} 1.4 \\ 1.3 \end{bmatrix} \in \mathbb{R}^2$$

Example 4: Input Vector of Length 5 (One Pass)

Consider an input vector of length 5 and three neurons in the layer:

- Input vector: $\mathbf{x} = \begin{bmatrix} 1 \\ 0.5 \\ -1 \\ 2 \\ 0 \end{bmatrix} \in \mathbb{R}^5$ - Weights for each neuron:

$$\mathbf{w}_1 = \begin{bmatrix} 0.1 \\ -0.2 \\ 0.5 \\ 0.3 \\ 0.7 \end{bmatrix} \in \mathbb{R}^5, \quad \mathbf{w}_2 = \begin{bmatrix} -0.3 \\ 0.8 \\ -0.1 \\ 0.4 \\ 0.2 \end{bmatrix} \in \mathbb{R}^5, \quad \mathbf{w}_3 = \begin{bmatrix} 0.5 \\ -0.4 \\ 0.6 \\ 0.1 \\ -0.2 \end{bmatrix} \in \mathbb{R}^5$$

- Biases: $b_1 = 0.1 \in \mathbb{R}$, $b_2 = -0.2 \in \mathbb{R}$, $b_3 = 0.05 \in \mathbb{R}$ - Activation function: Sigmoid

Calculation for neuron 1:

$$z_1 = \mathbf{w}_1^T \mathbf{x} + b_1 = 0.1(1) - 0.2(0.5) + 0.5(-1) + 0.3(2) + 0.7(0) + 0.1 = 0.3$$

$$h_1 = \sigma(0.3) = \frac{1}{1 + e^{-0.3}} \approx 0.574$$

Calculation for neuron 2:

$$z_2 = \mathbf{w}_2^T \mathbf{x} + b_2 = -0.3(1) + 0.8(0.5) - 0.1(-1) + 0.4(2) + 0.2(0) - 0.2 = 1.0$$

$$h_2 = \sigma(1.0) = \frac{1}{1 + e^{-1.0}} \approx 0.731$$

Calculation for neuron 3:

$$z_3 = \mathbf{w}_3^T \mathbf{x} + b_3 = 0.5(1) - 0.4(0.5) + 0.6(-1) + 0.1(2) - 0.2(0) + 0.05 = 0.0$$

$$h_3 = \sigma(0.0) = 0.5$$

The output of the layer is:

$$\mathbf{h} = \begin{bmatrix} 0.574 \\ 0.731 \\ 0.5 \end{bmatrix} \in \mathbb{R}^3$$

What Makes Neural Networks Better than Linear Models?

Neural networks provide key advantages over linear models by incorporating non-linear activation functions and allowing for complex relationships between input and output. The primary benefits include:

1. Non-Linearity:

Linear models can only approximate linear relationships, while neural networks can learn non-linear patterns through non-linear activation functions like ReLU, Sigmoid, and Tanh.

2. Higher Capacity:

A neural network with multiple layers can model far more intricate functions, allowing for more flexible decision boundaries.

3. Layered Structure:

Deep networks can decompose complex input features layer by layer, with each layer learning increasingly abstract representations. For example, in image recognition, early layers might learn to detect edges, while deeper layers identify complex objects.

4. Universality:

Neural networks are universal function approximators. This means that given enough data and computational resources, they can approximate any function to arbitrary accuracy.

What Makes Neural Networks Better than Non-Linear Models?

While traditional non-linear models (like polynomial regression) can model some non-linearities, neural networks surpass them in several key ways:

1. Flexibility:

Neural networks can adapt to arbitrary complex non-linearities in the data by adjusting weights across many layers, whereas non-linear models are often constrained by their specific functional form (e.g., polynomial degree).

2. Scalability:

Neural networks can handle large datasets and large feature spaces efficiently using parallel computing and modern hardware like GPUs.

3. Feature Engineering:

Non-linear models often require manual feature engineering (e.g., creating polynomial or interaction terms). Neural networks automate this process, learning hierarchical feature representations during training.

4. Depth and Learning:

Deep neural networks can stack multiple non-linear transformations, learning more abstract and useful representations of the data at each layer.