Lecture Notes: Neural Networks

As we delve into the world of statistical prediction, our exploration leads us to a pivotal concept: neural networks. These networks represent an evolution in our approach to modeling and understanding complex data relationships. Building on the foundation laid by logistic regression in our previous class, we see this familiar model not just as a tool for classification, but as the first step towards the sophisticated realm of neural networks.

Imagine logistic regression as a fundamental unit, a single neuron, in a much larger system. This perspective helps us transition from the simplicity of logistic regression to the multi-layered complexity of neural networks. Each neuron within these networks acts much like our logistic model, processing inputs and contributing to the overall predictive capability.

As we progress, we'll explore how layers of these neurons interconnect, creating a tapestry of statistical learning. In this intricate setup, every layer refines the predictions, transforming simple inputs into nuanced outputs. This process is akin to a mathematical and statistical ballet, where each step and turn brings us closer to more accurate and insightful predictions.

Our journey through neural networks is not just a technical exploration; it's a narrative about the evolution of statistical prediction. From the singular function of logistic regression to the orchestrated complexity of neural networks, each concept we uncover adds to our understanding of predictive modeling. This exploration is a testament to how far we've come in harnessing the power of statistics to interpret and predict the nuances of data.

So, let's embark on this enlightening journey, starting from the well-trodden path of logistic regression and venturing into the dynamic and intricate world of neural networks. Here, the confluence of statistics and prediction opens up a new chapter in our understanding of data-driven insights.

Logistic Regression as a Neural Network Foundation

Logistic regression, a model we're already familiar with, can be seen as the simplest form of a neural network, essentially representing a network with a single layer of computation. It models the probability that the input belongs to a particular class. The logistic regression equation is given by:

$$p(X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}} \tag{1}$$

where:

- $p(X)$ is the probability of the dependent variable equaling a 'success' or 'case'.
- β_0 and β_1 are the parameters to be estimated.
- e is the base of the natural logarithm.

Consider a dataset with binary outcomes and a single predictor variable. Here, logistic regression models the probability of a class being 'success'. This concept is visually represented In Figure 1.

In the process outlined in the diagram, we consider the following framework:

1. **Input Data**: Consider the input data as a vector $\mathbf{X} = (X_1, X_2, X_3, \ldots)$, where each X_i represents a feature of the input data point or batch of data points.

2. **Linear Transformation**: In the depicted model, a linear transformation is applied to the input data. This is performed using a set of weights $\mathbf{w} = (w_1, w_2, w_3, \ldots)$ and a bias term b. The linear transformation is represented by the equation:

$$z = \sum w_i X_i + b$$

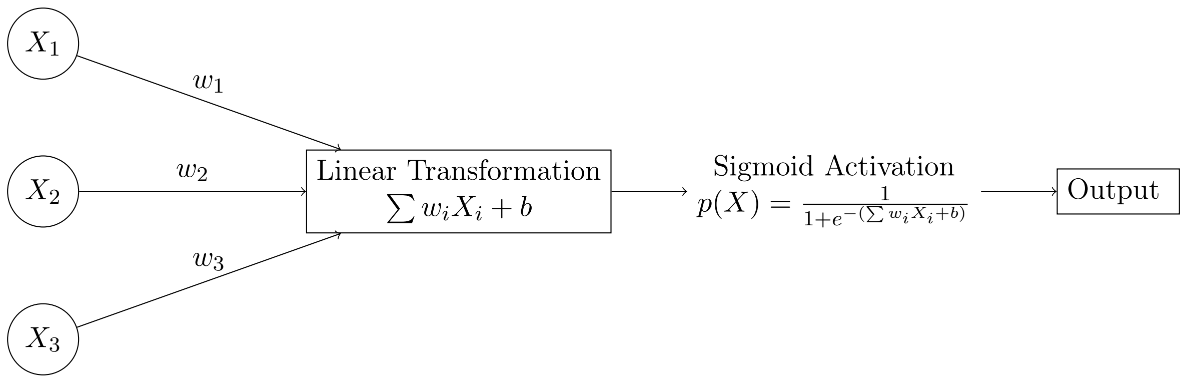

Figure 1: Logistic Regression as a One-Layer Neural Network for prediction

This step combines the input features linearly, using their respective weights and adding a bias.

3. **Sigmoid Activation Function**: The linearly transformed result z is then passed through a sigmoid activation function to introduce non-linearity, crucial for classification tasks. This is depicted by the equation:

$$p(X) = \frac{1}{1 + e^{-z}}$$

Here, $p(X)$ is the output of the sigmoid function, representing the probability of the input belonging to a certain class.

4. **Output**: The final step in the process is the output generation, where the activated value $p(X)$ is outputted as the final prediction of the model. This is indicative of the probability that the input data point belongs to a particular class, based on the logistic regression model.

Training, or estimation, in neural networks is a critical process that involves adjusting the model's weights and biases to minimize prediction error. This process typically consists of the following key steps:

1. **Initialization**: Initially, the weights $\mathbf{w}$ and biases b are assigned random or small values. This is the starting point for the optimization process.

2. **Forward Propagation**: For a given input data vector $\mathbf{X}$, the network performs forward propagation (as previously described), which involves computing the output of the network. This output is then compared to the actual target values to calculate the error or loss.

3. **Loss Function**: A loss function L quantifies the difference between the network's predictions and the actual target values. Common loss functions include Mean Squared Error (MSE) for regression tasks and Cross-Entropy Loss for classification tasks.

4. **Backward Propagation (Gradient Descent)**: In this step, the gradient of the loss function with respect to each weight and bias is computed. This involves applying the chain rule of calculus and is known as backpropagation. The gradients indicate the direction in which the weights and biases need to be adjusted to reduce the loss.

5. **Update Weights and Biases**: Using the gradients computed in the backward propagation step, the weights and biases are updated. This is typically done using an optimization

algorithm like Gradient Descent. The update equations are:

$$W_{\text{new}} = W_{\text{old}} - \alpha \frac{\partial L}{\partial W},$$
$$b_{\text{new}} = b_{\text{old}} - \alpha \frac{\partial L}{\partial b},$$

where α is the learning rate, a hyperparameter that controls the size of the update steps.

6. **Iteration**: The above steps are repeated for a number of iterations or until the loss converges to a minimum. Each iteration of forward and backward propagation followed by the update is called an epoch.

Training a neural network is an iterative process aimed at finding the optimal set of weights and biases that minimize the prediction error. The quality of the trained model is typically evaluated using a separate validation dataset to ensure the model generalizes well to new, unseen data.

Example: Designing a Simple Neural Network for Shape Recognition

Consider the problem of pattern recognition within the scope of machine learning, particularly in the context of image analysis. In this example, we address a simplified scenario where our input is a set of 16-pixel images, each represented as a 4x4 grid. The aim is to develop a model capable of discerning whether these images contain a specific pattern.

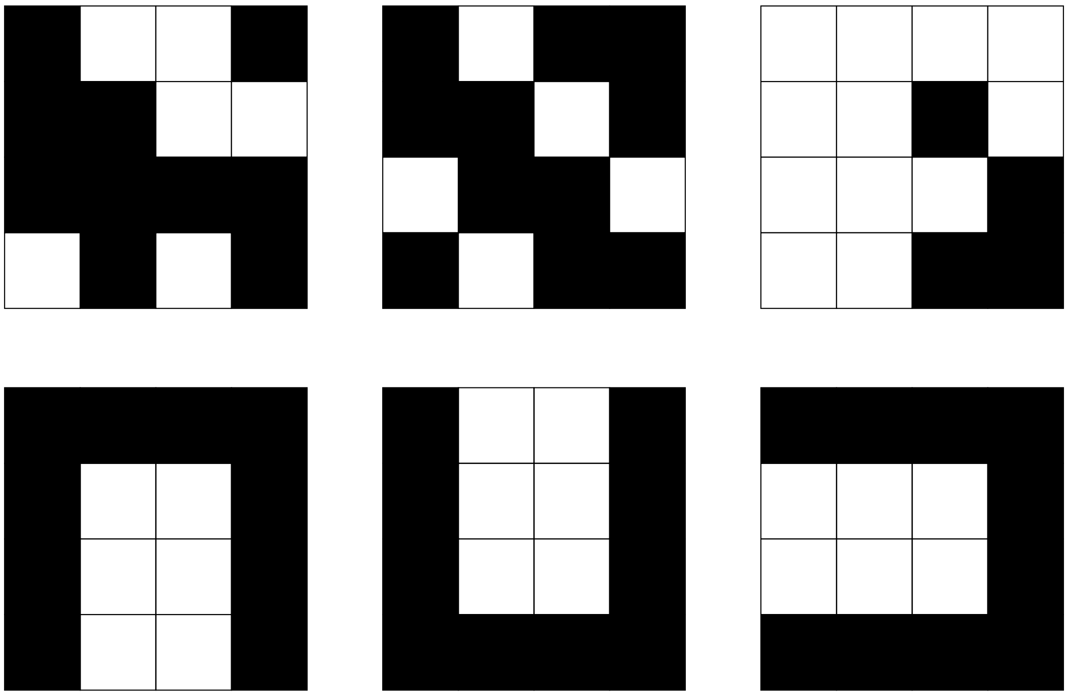

Figure 2: Top row: Three random 4x4 pixel patterns not containing the 'pi' shape. Bottom row: Three 4x4 pixel patterns showing different rotations of the 'pi' shape.

To tackle this problem, we decided to train a neural network model. The model takes as input a 16-dimensional binary vector—each element corresponding to a pixel in the 4x4 grid—and outputs the probability that the given image represents the specified pattern. The central task is to determine the set of 16 weights that best facilitate accurate pattern prediction.

The binary nature of the input vector simplifies the visual data to a format where each pixel is either 'on' or 'off'. The specific pattern we focus on is the "pi" shape, a distinct configuration within our 4x4 grid. Our training process involves adjusting the neural network's parameters to recognize this shape amidst various other pixel arrangements.

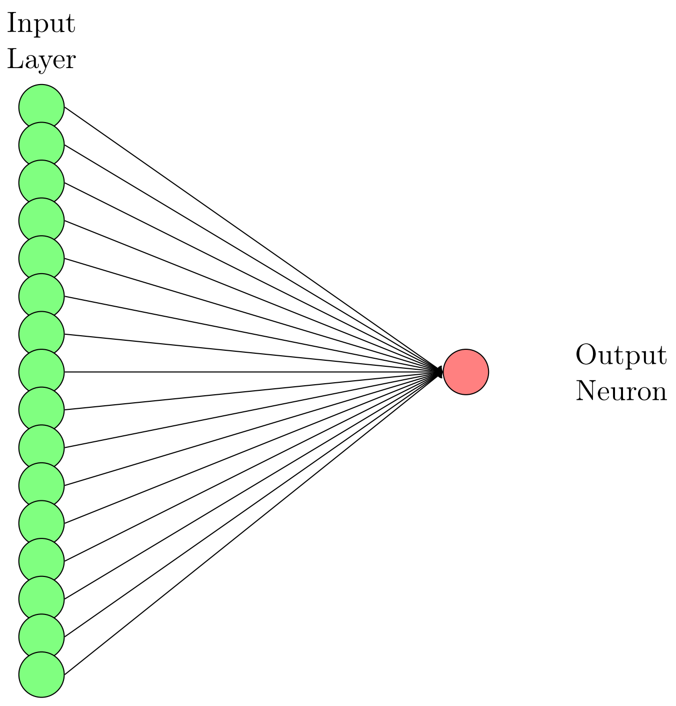

Figure 3: Arquitecture of the linear regresion neural network

Notations:

- X: Input matrix of dimensions $m \times n$ (where m is the number of examples, and n is the number of features).
- W: Weights matrix of dimensions $n \times 1$.
- b: Bias (a scalar).
- Z: Linear combination of weights and inputs plus the bias.
- A: Activation (output of the network) using the sigmoid function.

The equations for forward propagation are:

$$Z = XW + b$$

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

For binary classification, the binary cross-entropy loss is used:

$$L = -\frac{1}{m} \sum_{i=1}^{m} [Y_i \log(A_i) + (1 - Y_i) \log(1 - A_i)]$$

where Y is the vector of true labels.

Backpropagation involves computing the gradient of the loss function w.r.t each parameter (weights and biases).

Gradient of the Loss Function w.r.t Z (dZ):

$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial A} \cdot \frac{\partial A}{\partial Z}$$
$$\frac{\partial A}{\partial Z} = A \cdot (1 - A)$$
$$\frac{\partial L}{\partial A} = -\frac{Y}{A} + \frac{1-Y}{1-A}$$
$$\frac{\partial L}{\partial Z} = A - Y$$

Gradient w.r.t Weights W (dW):

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Z} \cdot \frac{\partial Z}{\partial W}$$
$$\frac{\partial Z}{\partial W} = X$$
$$dW = \frac{1}{m} X^T (A - Y)$$

Gradient w.r.t Bias b (db):

$$db = \frac{1}{m} \sum (A - Y)$$

In summary, in backpropagation, we compute:

- $dZ = A - Y$
- $dW = \frac{1}{m} X^T dZ$
- $db = \frac{1}{m} \sum dZ$

These gradients are used to update the weights and biases in the gradient descent step, iteratively minimizing the loss function during training.

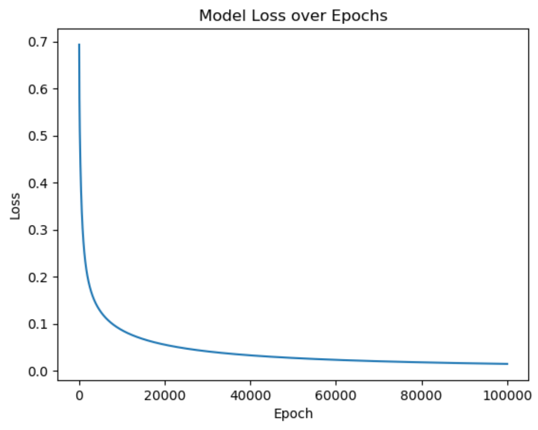

Figure 4: Loss vs. Epochs

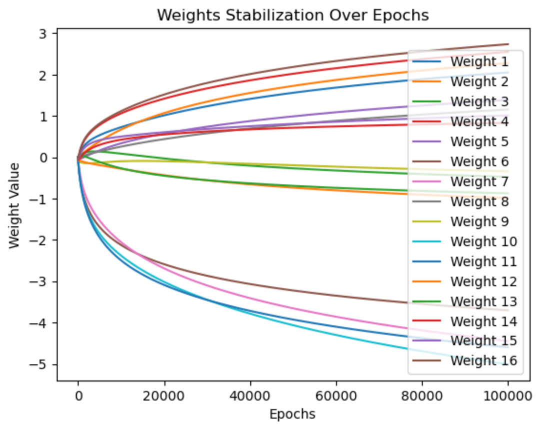

Figure 5: Weights vs. Epochs

Deep Neural networks

A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. It comprises

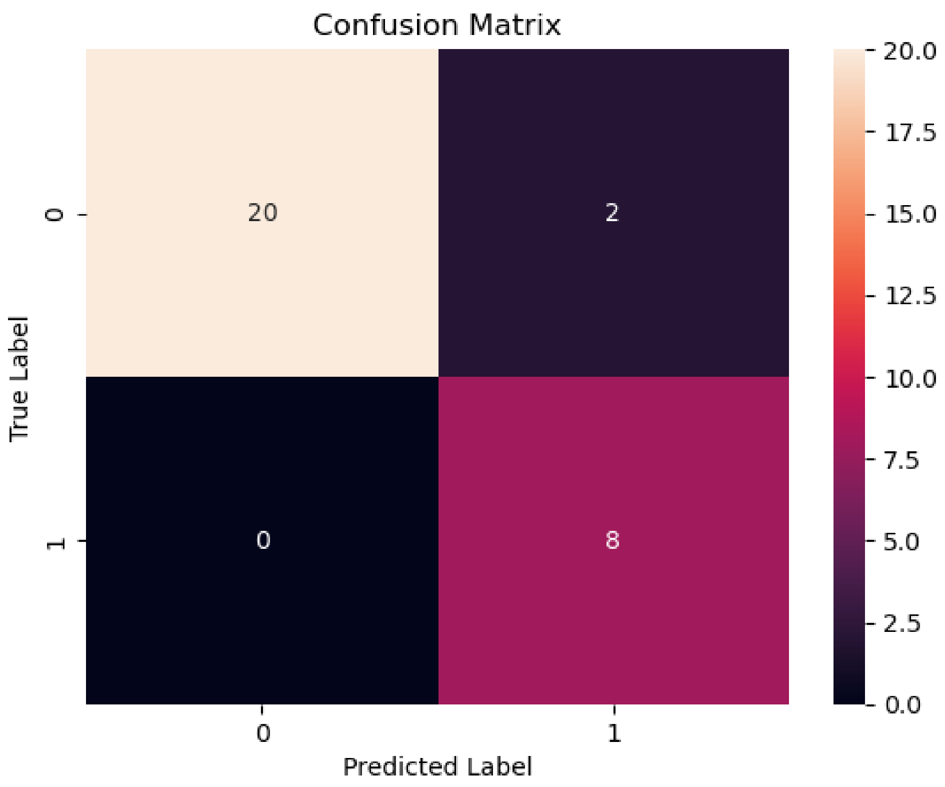

Figure 6: Loss vs. Epochs

layers of interconnected nodes or neurons, where each connection has an associated weight that gets adjusted during training. The network learns from the data by adjusting these weights to minimize the error between the predicted output and the actual target values.

In forward propagation, data moves from the input layer through hidden layers to the output layer. After this, a loss function L calculates the difference between the network's predictions $\hat{y}$ and actual target values y. For classification, a common loss function is the cross-entropy loss:

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

Backpropagation computes the gradients of the loss function with respect to the network's weights and biases. This process involves:

1. Computing the gradient at the output layer.
2. Propagating gradients backward through the network, calculating the gradient of the loss with respect to the weights and biases of each layer.
3. Applying the chain rule of calculus to compute these gradients.

Gradient descent, an optimization algorithm, minimizes the loss function by updating weights and biases using the equation:

$$w_{\text{new}} = w_{\text{old}} - \alpha \cdot \nabla_w L, \quad b_{\text{new}} = b_{\text{old}} - \alpha \cdot \nabla_b L$$

where α is the learning rate.

The initial step involves setting up the weights and biases. For example:

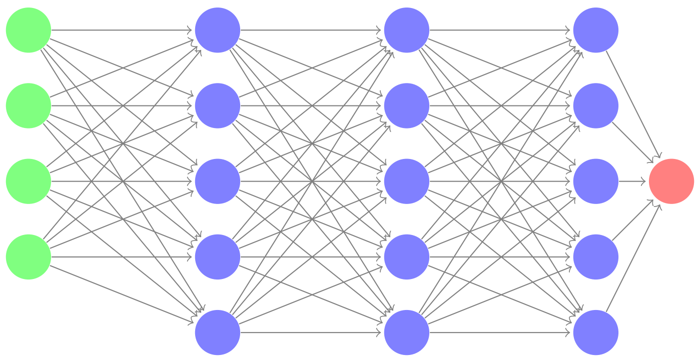

Figure 7: A Multi-Layer Neural Network

- Weights from Input to Hidden Layer ($W1$): $W1 = \begin{pmatrix} 0.1 & 0.4 \\ 0.2 & 0.5 \\ 0.3 & 0.6 \end{pmatrix}$
- Bias for Hidden Layer ($b1$): $b1 = \begin{pmatrix} 0.01 \\ 0.01 \end{pmatrix}$
- Weights from Hidden to Output Layer ($W2$): $W2 = \begin{pmatrix} 0.7 \\ 0.8 \end{pmatrix}$
- Bias for Output Layer ($b2$): $b2 = 0.02$

For forward propagation with an input $x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$, the hidden and output layer values are:

- Hidden Layer: $h = \sigma(W1^T \cdot x + b1)$
- Output Layer: $o = \sigma(W2^T \cdot h + b2)$

Backward propagation calculates gradients of the loss with respect to weights and biases, and these are used to update the parameters:

$$\begin{aligned} W1_{\text{new}} &= W1 - \alpha \cdot \nabla_{W1}\text{Loss}, \\ b1_{\text{new}} &= b1 - \alpha \cdot \nabla_{b1}\text{Loss}, \\ W2_{\text{new}} &= W2 - \alpha \cdot \nabla_{W2}\text{Loss}, \\ b2_{\text{new}} &= b2 - \alpha \cdot \nabla_{b2}\text{Loss}. \end{aligned}$$

The network undergoes iterative cycles of forward and backward propagation, adjusting parameters to enhance predictive accuracy. This systematic approach exemplifies model training in neural networks.

In neural networks, the equivalent of computing the loss function in logistic regression is termed 'forward propagation.' The process is defined as:

$$\text{Forward Propagation:} \quad \begin{aligned} Z^{[l]} &= W^{[l]} A^{[l-1]} + b^{[l]}, \\ A^{[l]} &= g^{[l]}(Z^{[l]}), \end{aligned}$$

where $A^{[0]}$ is the input data, $W^{[l]}$ and $b^{[l]}$ are the weights and biases at layer l, and $g^{[l]}$ is the activation function. The output layer yields the final prediction, and the loss function L quantifies the difference between this prediction and the actual targets.

The subsequent phase, 'backward propagation', analogous to gradient descent, involves calculating the gradients of the loss function with respect to each network parameter. This phase is expressed as:

$$\text{Backward Propagation:} \quad \frac{\partial L}{\partial W^{[l]}} = \frac{1}{m}\frac{\partial L}{\partial Z^{[l]}} A^{[l-1]T}, \quad \frac{\partial L}{\partial b^{[l]}} = \frac{1}{m}\sum \frac{\partial L}{\partial Z^{[l]}},$$

where m is the number of training examples. The gradients guide the update of weights and biases, thereby refining the model's predictions.

For instance, in the case of a network with two layers the backward propagation involves calculating the gradients:

$$\begin{aligned}
\frac{\partial L}{\partial W^{[2]}} &= \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[2]}}, \\
\frac{\partial L}{\partial b^{[2]}} &= \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial b^{[2]}}, \\
\frac{\partial L}{\partial W^{[1]}} &= \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial A^{[1]}} \cdot \frac{\partial A^{[1]}}{\partial Z^{[1]}} \cdot \frac{\partial Z^{[1]}}{\partial W^{[1]}}, \\
\frac{\partial L}{\partial b^{[1]}} &= \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial A^{[1]}} \cdot \frac{\partial A^{[1]}}{\partial Z^{[1]}} \cdot \frac{\partial Z^{[1]}}{\partial b^{[1]}}.
\end{aligned}$$

The gradients are used to update the weights and biases, refining the model's predictions through successive iterations.

Expanding upon this concept, a neural network consists of an input layer, several hidden layers, and an output layer. Each neuron in these layers is akin to the logistic model, applying a linear transformation followed by a non-linear activation function.

Through iterative cycles of forward and backward propagation, the neural network is trained. Parameters are adjusted in each iteration, improving the model's ability to predict accurately. This process exemplifies the systematic approach to model training in neural networks.

Example 1. *Consider a neural network comprising three layers: an input layer, a hidden layer, and an output layer. The architecture of this network is as follows:*

- *Input layer: 2 neurons (X_1 and X_2).*
- *Hidden layer: 2 neurons.*
- *Output layer: 1 neuron.*

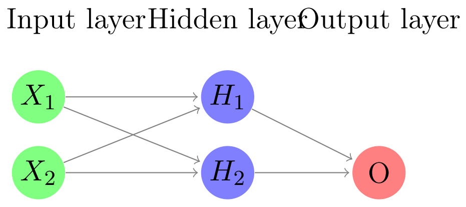

Figure 8: Simplified Neural Network Diagram

Weights and biases are defined for simplicity:

- *Weights from input layer to hidden layer:* $W^{[1]} = \begin{pmatrix} 0.15 & 0.25 \\ 0.20 & 0.30 \end{pmatrix}$

- *Bias for hidden layer:* $b^{[1]} = \begin{pmatrix} 0.35 \\ 0.35 \end{pmatrix}$
- *Weights from hidden layer to output layer:* $W^{[2]} = \begin{pmatrix} 0.40 \\ 0.45 \end{pmatrix}$
- *Bias for output layer:* $b^{[2]} = 0.60$

The activation function for all neurons is the sigmoid function, $\sigma(z) = \frac{1}{1+e^{-z}}$*.*
Given an input $X = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} 0.05 \\ 0.10 \end{pmatrix}$*, forward propagation is computed as follows:*

$$\begin{aligned}
Z^{[1]} &= W^{[1]}X + b^{[1]} = \begin{pmatrix} 0.15 & 0.25 \\ 0.20 & 0.30 \end{pmatrix} \begin{pmatrix} 0.05 \\ 0.10 \end{pmatrix} + \begin{pmatrix} 0.35 \\ 0.35 \end{pmatrix}, \\
A^{[1]} &= \sigma(Z^{[1]}), \\
Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} = \begin{pmatrix} 0.40 \\ 0.45 \end{pmatrix}^T A^{[1]} + 0.60, \\
A^{[2]} &= \sigma(Z^{[2]}) \text{ (Output of the network).}
\end{aligned}$$

Neural Network Prediction on MNIST Dataset

The MNIST dataset, consisting of handwritten digits, is a benchmark dataset in machine learning for evaluating classification algorithms. After training a neural network on this dataset, we perform a prediction using the trained model. The process involves selecting a sample image from the dataset, preprocessing it to match the input format of the network, and then using the model to predict the digit represented in the image.

Sample Selection and Preprocessing

We select a sample image from the MNIST dataset. This image is a 28x28 pixel grayscale image of a handwritten digit. The selected image is then preprocessed to be compatible with the input requirements of the neural network. This preprocessing includes flattening the image into a 784-element vector (since $28 \times 28 = 784$) and normalizing the pixel values.

The preprocessed image is fed into the trained neural network, which outputs a probability distribution over the ten digit classes. The predicted class is the one with the highest probability.

Training and Validation Loss and Accuracy

The performance of the neural network during the training phase can be visualized through the loss and accuracy plots. These plots show the training and validation loss and accuracy at each epoch and are crucial for understanding the learning behavior of the model, such as detecting overfitting or underfitting.

The left plot illustrates the loss on the training and validation sets over epochs, while the right plot shows the accuracy. An ideal scenario is where both the training and validation loss decrease over time and the accuracy increases, indicating that the model is learning effectively.

Modifications and Enhancements in Neural Network Frameworks

In developing neural networks, several modifications and enhancements can be made to the basic framework. These adjustments can significantly impact the performance, efficiency, and applicability of the neural network to various problems.

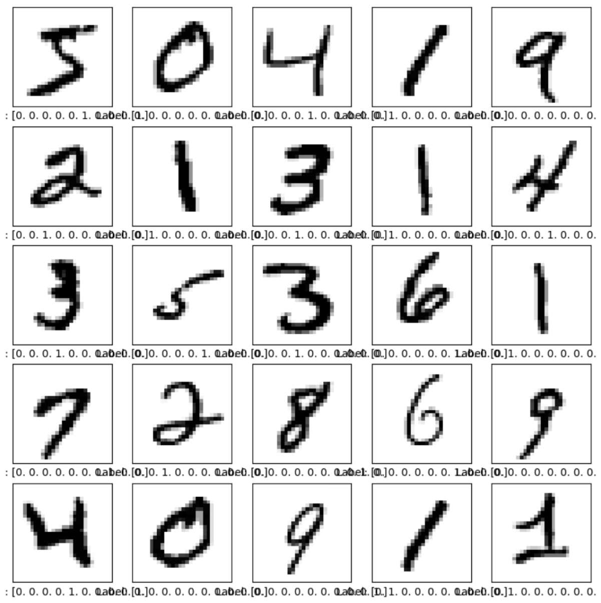

Figure 9: Selected handwritten digit from the MNIST dataset.

Activation Functions

Activation functions play a crucial role in neural networks, introducing non-linear properties to the model. While the sigmoid function is commonly used, especially in binary classification tasks, there are several other activation functions, each with its characteristics and use cases.

- **ReLU (Rectified Linear Unit):** One of the most popular activation functions for deep neural networks. It is defined as $f(x) = \max(0, x)$. ReLU helps to alleviate the vanishing gradient problem and allows models to learn faster.
- **Tanh (Hyperbolic Tangent):** This function outputs values between -1 and 1, making it suitable for applications where the model needs to predict the directionality of the data. It is defined as $f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$.
- **Softmax:** Often used in the output layer of a multi-class classification neural network, the softmax function converts logits to probabilities by comparing the exponential of each logit relative to the sum of exponentials of all logits. It is defined as $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$.

The choice of activation function can significantly affect the estimation process. For instance, ReLU is often preferred in hidden layers of deep networks due to its computational efficiency and ability to mitigate the vanishing gradient problem.

Optimization Algorithms

While gradient descent is a fundamental optimization algorithm in neural networks, several variations and alternatives offer different advantages:

- **Stochastic Gradient Descent (SGD):** Involves updating the model's parameters using only a single training example at a time. It can lead to faster iterations compared to standard gradient descent.
- **Adam (Adaptive Moment Estimation):** Combines the advantages of two other extensions of stochastic gradient descent, AdaGrad and RMSProp. Adam computes adaptive learning rates for each parameter and is efficient in terms of memory.

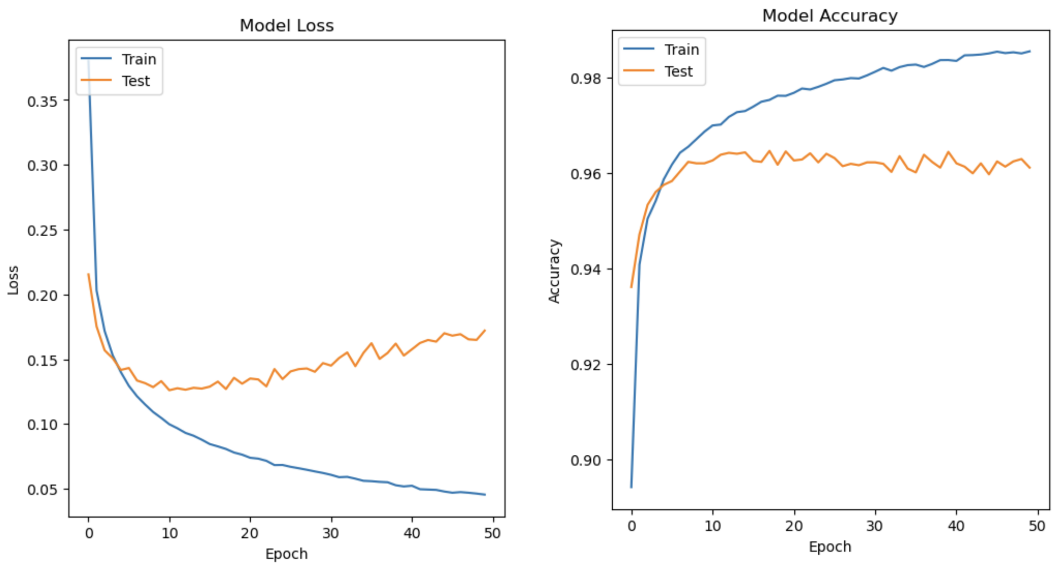

Figure 10: Training and validation loss and accuracy of the neural network on the MNIST dataset.

- **RMSprop (Root Mean Square Propagation):** An unpublished, adaptive learning rate method proposed by Geoff Hinton in his course. It divides the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.

Different optimization algorithms can impact the speed and quality of the training process. For instance, Adam is widely used in training deep learning models due to its robustness and effectiveness in handling sparse gradients.