Lecture 11

NON-LINEAR PREDICTION

Probability & Statistics

Francisco Richter, Martina Boschi and Ernst Wit

As we delve into the world of statistical prediction, our exploration leads us to pivotal concepts: neural networks, generalized linear models (GLMs), and generalized additive models (GAMs). These models represent an evolution in our approach to modeling and understanding complex data relationships.

1 Foundations of Neural Networks

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:

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

In this equation, $p(X)$ represents the probability that the dependent variable equals a 'success' or 'case'. The parameters β_0 and β_1 are estimated from the data, and e is the base of the natural logarithm.
Consider a dataset with binary outcomes and multiple predictor variables. Here, logistic regression models the probability of a class being 'success'. This concept is visually represented in Figure 1.

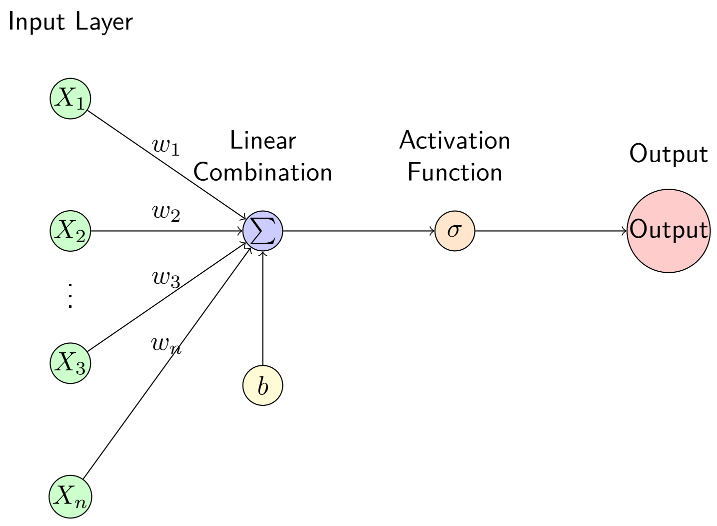


Figure 1: Logistic Regression as a One-Layer Neural Network for prediction

In the process outlined in the diagram, we consider the following framework:
The input data is represented as a vector $\mathbf{X} = (X_1, X_2, X_3, \ldots, X_n)$, where each X_i represents a feature of the input data point or batch of data points. A linear transformation is applied to the input data using a set of weights $\mathbf{w} = (w_1, w_2, w_3, \ldots, w_n)$ and a bias term b. The linear transformation is represented by the equation:

$$z = \sum_{i=1}^{n} w_i X_i + b$$

This step combines the input features linearly, using their respective weights and adding a bias. The linearly transformed result z is then passed through a sigmoid activation function to introduce non-linearity, crucial for classification tasks:

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

Here, $p(X)$ is the output of the sigmoid function, representing the probability of the input belonging to a certain class. 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 indicates the probability that the input data point belongs to a particular class, based on the logistic regression model.

1.1 Generalized Linear Models (GLMs)

Generalized Linear Models (GLMs) are an extension of traditional linear regression models that allow for response variables that have error distribution models other than a normal distribution. GLMs are widely used in statistics and are essential tools for modeling various types of data, especially when dealing with non-normal response distributions and non-linear relationships.
GLMs consist of three components: a random component that specifies the probability distribution of the response variable Y, which can be normal, binomial, Poisson, gamma, or another distribution from the exponential family; a systematic component that involves a linear combination of the explanatory variables, represented by the linear predictor:

$$\eta = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p$$

where $\beta_0, \beta_1, \ldots, \beta_p$ are the model coefficients and $X_1, X_2, \ldots, X_p$ are the predictor variables. Finally, a link function $g(\mu)$ relates the expected value of the response variable $\mu = E[Y]$ to the linear predictor to ensure predictions stay within valid limits. The choice of the link function depends on the nature of the response variable.
A key aspect of GLMs is that they assume the response variable comes from the exponential family of distributions, which includes normal, binomial, Poisson, and gamma distributions. The general form of the exponential family is:

$$f_Y(y; \theta, \phi) = \exp\left(\frac{y\theta - b(\theta)}{a(\phi)} + c(y, \phi)\right)$$

where θ is the canonical parameter, ϕ is the dispersion parameter, and $a(\cdot), b(\cdot), c(\cdot)$ are specific functions that define the distribution.
The link function $g(\cdot)$ connects the mean of the response variable to the linear predictor. Table 1 summarizes common link functions and their usage.

Link Function	**Formula**	**Description**
Identity Link	$g(\mu) = \mu$	Normal distributions in linear regression.
Logit Link	$g(\mu) = \log\left(\frac{\mu}{1-\mu}\right)$	Logistic regression for binomial data.
Probit Link	$g(\mu) = \Phi^{-1}(\mu)$	Inverse CDF of standard normal distribution.
Log Link	$g(\mu) = \log(\mu)$	Poisson regression for count data.
Reciprocal Link	$g(\mu) = \frac{1}{\mu}$	Gamma regression.

Table 1: Common Link Functions in Generalized Linear Models

For example, when the response variable is normally distributed and the identity link function is used, the GLM reduces to the traditional linear regression model:

$$Y = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p + \varepsilon, \quad \varepsilon \sim N(0, \sigma^2)$$

In contrast, for binary response variables, logistic regression is used, which employs the binomial distribution and the logit link function:

$$\log\left(\frac{\mu}{1-\mu}\right) = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p$$

Here, $\mu = E[Y]$ represents the probability of success. Similarly, for count data, the Poisson regression model uses the Poisson distribution and the log link function:

$$\log(\mu) = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p$$

However, while GLMs are flexible, they have certain limitations. They assume a linear relationship between the predictors and the transformed response variable, which may not always be appropriate. The model also assumes independence between observations and can suffer from overdispersion in models like Poisson regression, where the variance is assumed to equal the mean, which may not hold in practice.

GLMs are applied across numerous fields, including medicine (e.g., modeling the probability of disease occurrence based on risk factors), epidemiology (e.g., analyzing the number of disease cases), insurance (e.g., modeling claim counts and amounts), and economics (e.g., analyzing binary outcomes such as purchase decisions).

1.2 Generalized Additive Models (GAMs)

Generalized Additive Models (GAMs) extend GLMs by allowing for more flexibility in modeling the relationship between predictors and the response variable non-parametrically. Unlike GLMs, which assume a linear relationship, GAMs model each predictor's effect using smooth functions. A GAM is defined as:

$$g(\mu) = \beta_0 + f_1(X_1) + f_2(X_2) + \ldots + f_p(X_p)$$

where $g(\mu)$ is the link function, β_0 is the intercept, and $f_i(X_i)$ are smooth functions of the predictors that can capture complex, non-linear relationships. These functions are estimated from the data and can take the form of splines, kernel smooths, or local regression (LOESS).
Splines are piecewise polynomial functions joined at knots, which allow them to flexibly model non-linear trends in the data. Types of splines used in GAMs include cubic splines, B-splines, and penalized splines. Kernel smooths use kernel functions to weigh nearby observations, and LOESS fits simple models to localized subsets of the data to create a smooth curve.
To estimate a GAM, the smooth functions $f_i(\cdot)$ and the model parameters must be estimated simultaneously. This is typically done using methods like backfitting algorithms, where each function is estimated while holding the others fixed and iteratively updating until convergence.
The backfitting algorithm proceeds by initializing the functions $f_i(\cdot)$, and for each predictor X_j, computing the partial residuals:

$$R_j = g(\mu) - \beta_0 - \sum_{i \neq j} f_i(X_i)$$

The function $f_j(X_j)$ is then updated by smoothing R_j against X_j. This process repeats until the changes in the functions are below a specified threshold.
To prevent overfitting, penalization is applied to control the smoothness of the functions. The penalized least squares criterion is written as:

$$\text{PLS} = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} f_j(X_{ij}) \right)^2 + \sum_{j=1}^{p} \lambda_j \int [f_j''(x)]^2 dx$$

where λ_j are smoothing parameters that control the trade-off between fitting the data and the smoothness of f_j, and $f_j''(x)$ represents the second derivative of f_j, penalizing curvature.
GAMs offer significant advantages over GLMs by being able to model complex, non-linear relationships without specifying a particular functional form. This makes them highly flexible and interpretable, as the additive structure allows for easy visualization of each predictor's effect. GAMs are widely used in fields such as environmental modeling (e.g., species abundance with respect to environmental gradients), finance (e.g., modeling non-linear effects of economic indicators), and medicine (e.g., modeling patient outcomes based on biomarkers).
However, GAMs also have limitations, such as the potential for overfitting if the smoothness is not adequately controlled, computational complexity in estimating smooth functions, especially for large datasets, and unreliable extrapolation beyond the range of observed data.
GAMs can be implemented using statistical software packages like R (with the `mgcv` package), Python (with the `pyGAM` library), and other software like SAS and SPSS.
For example, consider modeling house prices (Y) based on square footage (X_1) and age of the house (X_2). A GAM for this relationship might be:

$$\log(\mu) = \beta_0 + f_1(\text{Square Footage}) + f_2(\text{Age})$$

The functions f_1 and f_2 can capture non-linear effects on the log of the expected house price. One strength of GAMs is that the estimated smooth functions can be visualized to understand the nature of the relationships.
In addition to modeling individual predictor effects, GAMs can include interaction terms, such as:

$$g(\mu) = \beta_0 + f_1(X_1) + f_2(X_2) + f_{12}(X_1, X_2)$$

where f_{12} is a smooth function of both X_1 and X_2, capturing their interaction.

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 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.

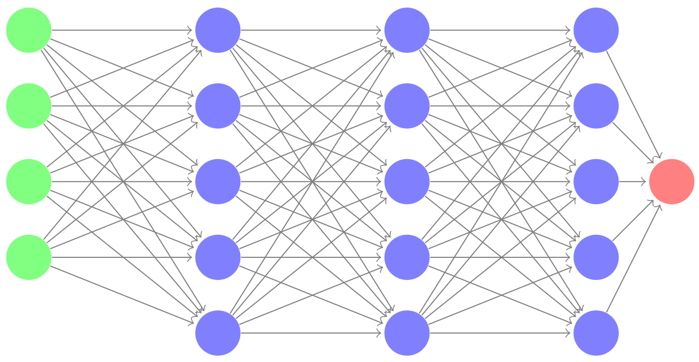


Figure 2: A Multi-Layer Neural Network

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:

- 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:

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

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:

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

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. 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.

Weights and biases are defined for simplicity:

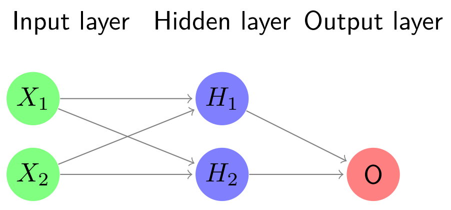


Figure 3: Simplified Neural Network Diagram

- 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:

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

□

1.3 Training and Estimation

Training, or estimation, in neural networks, GLMs, and GAMs involves adjusting the model parameters to minimize prediction error.
For GLMs, parameters are estimated using Maximum Likelihood Estimation (MLE), which involves specifying the model, formulating the likelihood function based on the assumed distribution, optimizing the likelihood using numerical methods, and assessing model fit through goodness-of-fit tests and residual analysis.
In GAMs, training involves selecting smoothing parameters for the functions $f_i(\cdot)$, with techniques like generalized cross-validation (GCV) or Akaike's Information Criterion (AIC) being used to balance model fit and smoothness. The backfitting algorithm or penalized regression splines are often used to estimate the smooth functions.
Regularization techniques such as Lasso (L1 Regularization) and Ridge (L2 Regularization) are used to prevent overfitting in both GLMs and GAMs. These methods add a penalty term to the loss function, proportional to the magnitude of the coefficients, thereby reducing overfitting.
Model selection, including choosing appropriate predictors and model complexity, can be performed using stepwise selection, information criteria like AIC or BIC, and cross-validation to evaluate model performance.
Suppose we are modeling the probability of a customer making a purchase (Y) based on the time they spend on a website (X). Using a GLM with logistic regression, we have:

$$\log\left(\frac{\mu}{1-\mu}\right) = \beta_0 + \beta_1 X$$

This assumes a linear relationship between X and the log-odds of purchase. In contrast, using a GAM:

$$\log\left(\frac{\mu}{1-\mu}\right) = \beta_0 + f(X)$$

Here, $f(X)$ is a smooth function that captures a potentially non-linear relationship, allowing for more flexibility if, for example, the probability of purchase increases rapidly with time up to a point and then levels off.

2 Image Recognition

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. 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.

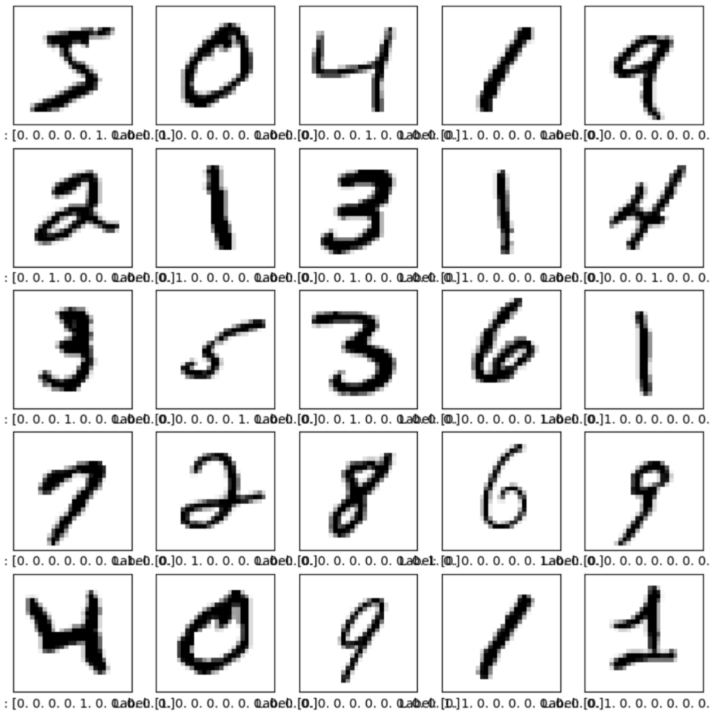
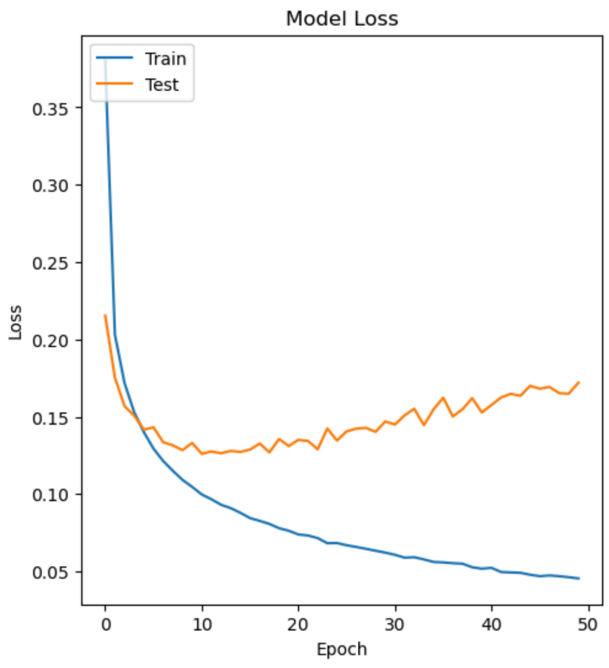


Figure 4: Selected handwritten digit from the MNIST dataset.

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.
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.



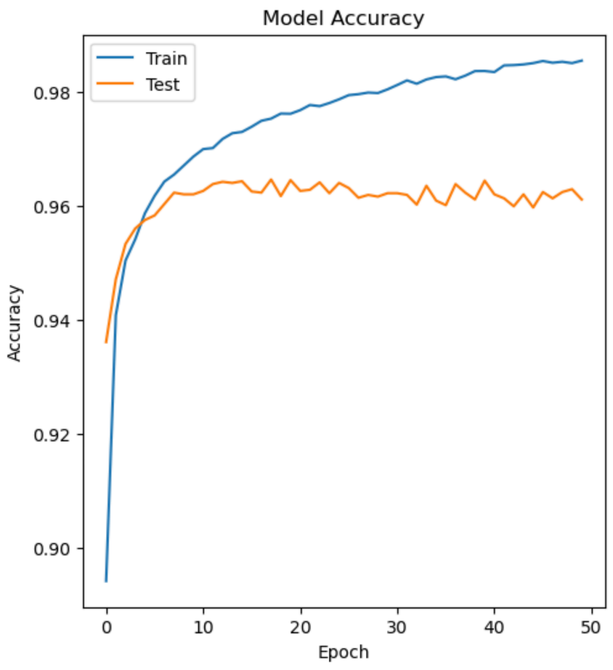


Figure 5: Training and validation loss and accuracy of the neural network on the MNIST dataset.

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.

3 Network Enhancements

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.

3.1 Activation Functions

Activation functions play a crucial role in neural networks, introducing non-linear properties to the model. The sigmoid function is commonly used, especially in binary classification tasks. Other activation functions are defined as follows:

ReLU (Rectified Linear Unit) Defined as:

$$f(x) = \max(0, x)$$

ReLU helps to alleviate the vanishing gradient problem and allows models to learn faster.

Leaky ReLU Addresses the "dying ReLU" problem by allowing a small, non-zero gradient when the unit is not active:

$$f(x) = \max(\alpha x, x)$$

where α is a small constant.

Tanh (Hyperbolic Tangent) Defined as:

$$f(x) = \tanh(x)$$

Tanh outputs values between -1 and 1, making it suitable for applications where the model needs to predict the directionality of the data.

Softmax Used in the output layer of multi-class classification neural networks. Softmax converts logits to probabilities by comparing the exponential of each logit relative to the sum of exponentials of all logits.

3.2 Optimization Algorithms

Gradient descent is a fundamental optimization algorithm in neural networks, but 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, leading to faster iterations compared to standard gradient descent. Mini-Batch Gradient Descent uses a small batch of training examples per update, balancing efficiency and speed. Adam (Adaptive Moment Estimation) combines the advantages of two other extensions of stochastic gradient descent, AdaGrad and RMSProp, by computing adaptive learning rates for each parameter. Other optimizers like RMSprop divide the learning rate for a weight by a running average of the magnitudes of recent gradients, and Momentum accelerates gradient descent by adding a fraction of the previous update to the current update.
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.

3.3 Regularization Techniques

To prevent overfitting, various regularization techniques are employed. L1 and L2 Regularization add a penalty term to the loss function proportional to the magnitude of the weights. Dropout randomly drops units (along with their connections) from the neural network during training, which prevents units from co-adapting too much. Early stopping halts training when the performance on a validation set starts to degrade, ensuring that the model does not overfit to the training data.

3.4 Batch Normalization

Batch normalization normalizes the input layer by adjusting and scaling the activations, which helps accelerate the training of deep neural networks. It reduces internal covariate shift and allows for higher learning rates.