Introduction to Deep Learning with R

Intermediate Predictive Analytics

Prof.Asc. Endri Raco, Ph.D.

Department of Mathematical Engineering
Polytechnic University of Tirana

November 2025

Section 1

Introduction

Why Deep Learning?

Deep learning has revolutionized predictive analytics by enabling machines to automatically discover patterns in data without extensive feature engineering.

Key Applications:

- Image recognition and computer vision
- Natural language processing
- Time series forecasting
- Customer behavior prediction
- Recommendation systems

Today's Focus: Understanding neural networks and implementing them in R using the keras package.

Course Context

This lecture is part of our **Intermediate Predictive Analytics** course sequence:

Previous Topics: Basetable construction, feature engineering, traditional ML models

Today: Introduction to neural networks and deep learning

Next Topics: Advanced architectures, model optimization, deployment

Learning Objectives:

- Understand neural network architecture and forward propagation
- Implement activation functions in R
- Build multi-layer networks
- Recognize when deep learning adds value

Section 2

Neural Networks: Core Concepts

The Motivation: Capturing Interactions

Consider predicting bank transactions based on customer characteristics.

Linear Regression View:

- Each feature contributes independently
- Interactions must be manually specified
- Limited ability to capture complex patterns

Neural Network View:

- Automatically learns feature interactions
- Hidden layers create intermediate representations
- Can model highly nonlinear relationships

Example: Bank Transaction Prediction

Scenario: Predict number of transactions next year

Input Features:

- Number of children
- Number of existing accounts
- Bank balance
- Retirement status

Question: How do these features interact?

- Retired customers with high balances behave differently
- Multiple accounts interact with number of children
- Neural networks learn these patterns automatically

Network Architecture: Building Blocks

Components:

- **Input Layer:** Raw features
- **Hidden Layers:** Learned representations
- **Output Layer:** Predictions
- **Weights:** Learned parameters

Key Insight:

- Each connection has a weight
- Weights are learned from data
- More layers = deeper networks
- "Deep" learning comes from multiple hidden layers

Section 3

Forward Propagation

What is Forward Propagation?

Forward propagation is the process of computing predictions by passing input data through the network layer by layer.

The Process:

1. Start with input values
2. Multiply by weights and sum (dot product)
3. Apply activation function
4. Pass result to next layer
5. Repeat until reaching output

Critical: This is how neural networks make predictions for a single data point.

Simple Example: Two Inputs, One Hidden Layer

Consider inputs: number of children = 2, number of accounts = 3

```r
# Input data
input_data <- c(2, 3)

# Weights for hidden layer nodes
# First hidden node
weights_node_0 <- c(1, 1)
# Second hidden node
weights_node_1 <- c(-1, 1)

# Weights for output layer
weights_output <- c(2, -1)
```

Computing Hidden Layer Values I

Node 0 calculation:

```
# Compute weighted sum for node 0
node_0_value <- sum(input_data * weights_node_0)
node_0_value
```

```
## [1] 5
```

Computing Hidden Layer Values II

Node 1 calculation:

```
# Compute weighted sum for node 1
node_1_value <- sum(input_data * weights_node_1)
node_1_value
```

```
## [1] 1
```

The hidden layer now has values: 5 and 1

Computing Final Output

```r
# Combine hidden layer values
hidden_layer_values <- c(node_0_value, node_1_value)

# Compute final output
output <- sum(hidden_layer_values * weights_output)
output
```

```
## [1] 9
```

Result: The network predicts 9 transactions

This is pure forward propagation without activation functions (coming next!)

Complete Forward Propagation Function I

```r
# Function to perform forward propagation
forward_prop <- function(input_data, weights) {
  # Calculate hidden layer
  node_0_value <- sum(input_data * weights$node_0)
  node_1_value <- sum(input_data * weights$node_1)

  # Combine hidden layer
  hidden_layer <- c(node_0_value, node_1_value)

  # Calculate output
  output <- sum(hidden_layer * weights$output)

  return(output)
}
```

Complete Forward Propagation Function II

```r
# Define weights structure
weights <- list(
  node_0 = c(1, 1),
  node_1 = c(-1, 1),
  output = c(2, -1)
)

# Test the function
result <- forward_prop(input_data, weights)
cat("Prediction:", result, "\n")
```

```
## Prediction: 9
```

Section 4

Activation Functions

Why Activation Functions?

Problem: Without activation functions, neural networks are just linear models

- Stacking linear transformations = one big linear transformation
- Cannot capture nonlinear patterns
- Limited expressiveness

Solution: Apply nonlinear activation functions

- Introduces nonlinearity into the network
- Enables learning of complex patterns
- Each layer can learn increasingly abstract features

Common Activation Functions I

1. ReLU (Rectified Linear Unit)

- Most popular in deep learning
- $f(x) = \max(0, x)$
- Fast to compute
- Helps avoid vanishing gradients

2. Tanh (Hyperbolic Tangent)

- $f(x) = \tanh(x)$
- Outputs in range $[-1, 1]$
- Centered around zero

Common Activation Functions II

3. Sigmoid

- $f(x) = \frac{1}{1+e^{-x}}$
- Outputs in range $[0, 1]$
- Good for binary classification

When to use what?

- **Hidden layers:** ReLU (default)
- **Output layer:** depends on task
  - Binary: sigmoid
  - Multi-class: softmax

ReLU: The Default Choice I

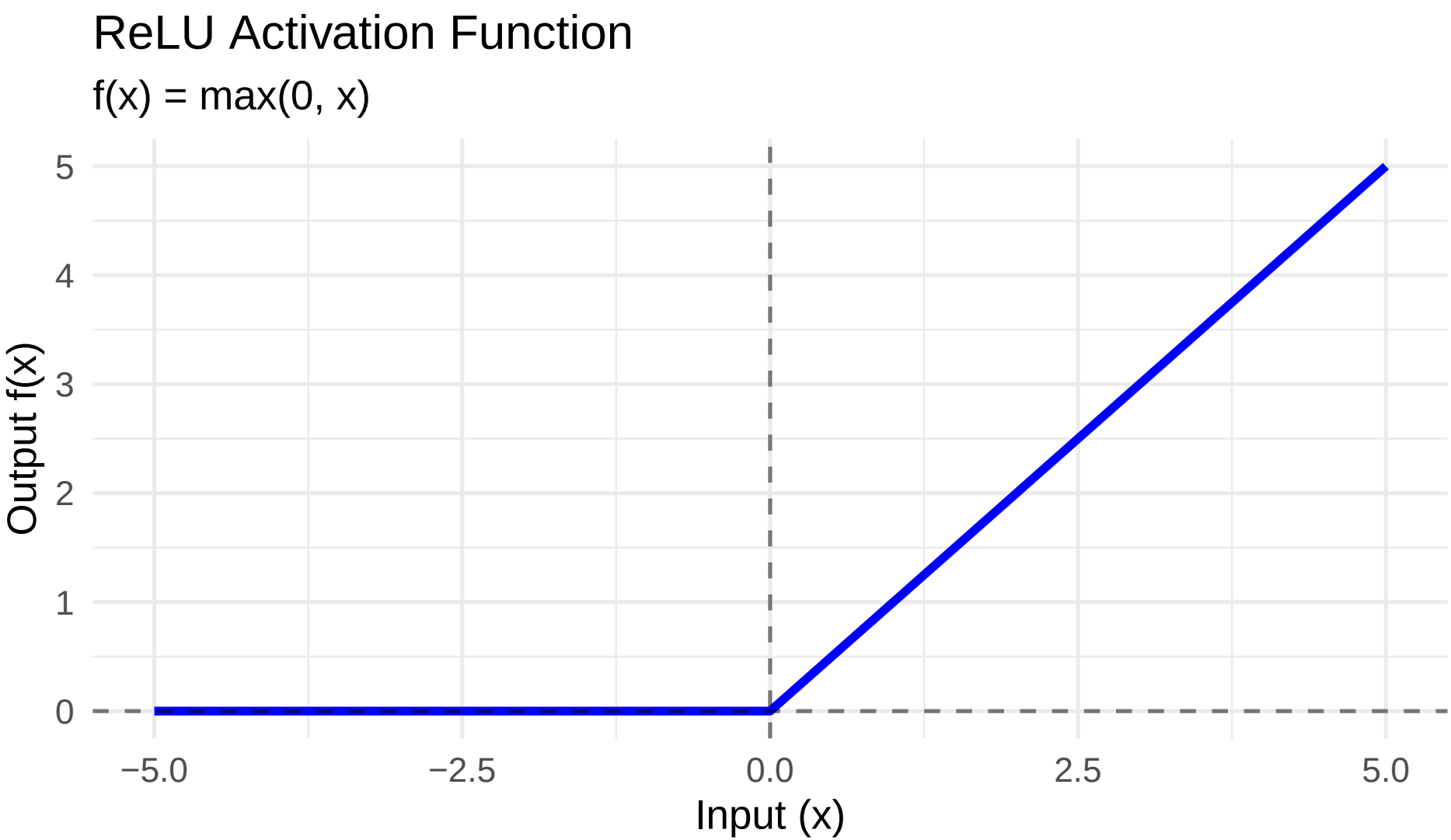
ReLU (Rectified Linear Unit) is the most widely used activation function.

ReLU: The Default Choice II

```r
# Create data for visualization
x <- seq(-5, 5, by = 0.1)
y_relu <- pmax(0, x)  # ReLU: max(0, x)

# Plot ReLU function
ggplot(data.frame(x = x, y = y_relu), aes(x, y)) +
  geom_line(color = "blue", linewidth = 1.2) +
  geom_hline(yintercept = 0, linetype = "dashed",
             alpha = 0.5) +
  geom_vline(xintercept = 0, linetype = "dashed",
             alpha = 0.5) +
  labs(title = "ReLU Activation Function",
       subtitle = "f(x) = max(0, x)",
       x = "Input (x)", y = "Output f(x)") +
  theme_minimal(base_size = 12)
```

ReLU: The Default Choice III



Implementing Activation Functions in R I

```r
# ReLU implementation
relu <- function(x) {
  # Element-wise maximum with 0
  pmax(0, x)
}

# Tanh is built-in: tanh(x)

# Sigmoid implementation
sigmoid <- function(x) {
  1 / (1 + exp(-x))
}
```

Implementing Activation Functions in R II

```r
# Test the functions
test_values <- c(-2, -1, 0, 1, 2)

results <- data.frame(
  Input = test_values,
  ReLU = relu(test_values),
  Tanh = tanh(test_values),
  Sigmoid = sigmoid(test_values)
)

print(results)
```

Implementing Activation Functions in R III

```
##   Input ReLU       Tanh   Sigmoid
## 1    -2    0 -0.9640276 0.1192029
## 2    -1    0 -0.7615942 0.2689414
## 3     0    0  0.0000000 0.5000000
## 4     1    1  0.7615942 0.7310586
## 5     2    2  0.9640276 0.8807971
```

Forward Propagation with Activation Functions

```r
# Enhanced forward propagation with ReLU
forward_prop_relu <- function(input_data, weights) {
  # Calculate hidden layer with ReLU activation
  node_0_input <- sum(input_data * weights$node_0)
  node_0_output <- relu(node_0_input)

  node_1_input <- sum(input_data * weights$node_1)
  node_1_output <- relu(node_1_input)

  # Combine hidden layer
  hidden_layer <- c(node_0_output, node_1_output)

  # Calculate output (no activation for regression)
  output <- sum(hidden_layer * weights$output)

  return(output)
}
```

Comparing Activation Functions I

Comparing Activation Functions II

```r
# Compare multiple activation functions
x <- seq(-5, 5, by = 0.1)

comparison_data <- data.frame(
  x = x,
  ReLU = relu(x),
  Tanh = tanh(x),
  Sigmoid = sigmoid(x)
) %>%
  pivot_longer(cols = -x, names_to = "Function",
               values_to = "y")

ggplot(comparison_data, aes(x, y, color = Function)) +
  geom_line(linewidth = 1.2) +
  labs(title = "Comparison of Activation Functions",
       x = "Input (x)", y = "Output f(x)") +
  theme_minimal(base_size = 12) +
```

Section 5

Deeper Networks

Why Go Deeper?

Single hidden layer networks are limited

- Can approximate many functions (universal approximation theorem)
- But may require exponentially many neurons
- Difficult to learn hierarchical features

Multiple hidden layers enable:

1. **Hierarchical feature learning** - Each layer learns progressively abstract features
2. **Improved generalization** - Better at capturing complex patterns
3. **Parameter efficiency** - Fewer total parameters for same expressiveness

Example: Multi-Layer Network Architecture I

Consider a network with two hidden layers:

`Input Layer → Hidden Layer 1 → Hidden Layer 2 → Output Layer`

Example: Multi-Layer Network Architecture II

```r
# Define weights for two-hidden-layer network
multilayer_weights <- list(
  # Input to first hidden layer (2 nodes)
  hidden1_node0 = c(2, 4),
  hidden1_node1 = c(-5, -1),

  # First hidden to second hidden layer (2 nodes)
  hidden2_node0 = c(-1, 1),
  hidden2_node1 = c(2, 4),

  # Second hidden to output
  output = c(-3, 7)
)
```

Forward Propagation Through Multiple Layers I

Forward Propagation Through Multiple Layers II

```
multilayer_forward <- function(input_data, weights) {
  # First hidden layer
  h1_node0 <- relu(sum(input_data *
                       weights$hidden1_node0))
  h1_node1 <- relu(sum(input_data *
                       weights$hidden1_node1))
  hidden1_output <- c(h1_node0, h1_node1)

  # Second hidden layer
  h2_node0 <- relu(sum(hidden1_output *
                       weights$hidden2_node0))
  h2_node1 <- relu(sum(hidden1_output *
                       weights$hidden2_node1))
  hidden2_output <- c(h2_node0, h2_node1)

  # Output layer
  output <- sum(hidden2_output * weights$output)
```

Examining Network Activations

Key Observations:

- ReLU sets negative values to zero
- Only some nodes activate for each input
- Creates sparse representations
- Different inputs activate different pathways

This **selective activation** is what enables the network to learn complex decision boundaries and feature hierarchies.

Representation Learning

What is Representation Learning?

Deep networks automatically discover useful representations of data in their hidden layers.

Layer-by-layer learning:

- **Layer 1:** Simple patterns (edges, basic features)
- **Layer 2:** Combinations of simple patterns
- **Layer 3:** High-level concepts
- **Output:** Final prediction

Benefits:

- Reduces need for manual feature engineering
- Learns task-specific representations
- Transfers knowledge between tasks
- Discovers patterns humans might miss

Section 6

Building Neural Networks in R with Keras

Introduction to Keras

Keras is a high-level neural networks API that runs on top of TensorFlow.

```r
# Install keras (one-time setup)
install.packages("keras")
library(keras)
install_keras()  # Installs TensorFlow backend

# Load keras for use
library(keras)
```

Why Keras?

- Simple, consistent interface
- Quick prototyping
- Production-ready
- Extensive pre-trained models
- Active community

Building Your First Neural Network I

```r
# Define network architecture
model <- keras_model_sequential() %>%
  # First hidden layer: 100 units, ReLU activation
  layer_dense(units = 100, activation = 'relu',
              input_shape = c(10)) %>%

  # Second hidden layer: 100 units, ReLU activation
  layer_dense(units = 100, activation = 'relu') %>%

  # Output layer: 1 unit (regression)
  layer_dense(units = 1)

# View model architecture
summary(model)
```

Building Your First Neural Network II

Key Parameters:

- `units`: Number of neurons in the layer
- `activation`: Activation function to use
- `input_shape`: Dimensions of input (first layer only)

Complete Example: Bank Transaction Prediction I

Complete Example: Bank Transaction Prediction II

```r
# Load and prepare data
library(tidyverse)

# Simulated bank customer data
set.seed(123)
n_customers <- 1000

customer_data <- tibble(
  n_children = sample(0:4, n_customers,
                      replace = TRUE),
  n_accounts = sample(1:5, n_customers,
                      replace = TRUE),
  balance = rnorm(n_customers, 10000, 5000),
  retired = sample(0:1, n_customers,
                   replace = TRUE),
  age = sample(25:75, n_customers, replace = TRUE)
) %>%
```

Preprocessing and Train/Test Split I

Preprocessing and Train/Test Split II

```r
# Separate features and target
X <- customer_data %>%
  select(-n_transactions) %>%
  as.matrix()

y <- customer_data$n_transactions

# Normalize features (important for neural networks!)
X_normalized <- scale(X)

# Train/test split (80/20)
set.seed(42)
train_indices <- sample(1:n_customers,
                        0.8 * n_customers)

X_train <- X_normalized[train_indices, ]
y_train <- y[train_indices]
```

Building and Compiling the Model I

Building and Compiling the Model II

```r
# Build the model
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = 'relu',
              input_shape = c(ncol(X_train))) %>%
  layer_dense(units = 32, activation = 'relu') %>%
  layer_dense(units = 1)

# Compile the model
model %>% compile(
  optimizer = 'adam',        # Optimization algorithm
  loss = 'mse',              # Mean squared error
  metrics = c('mae')         # Mean absolute error
)

# View architecture
summary(model)
```

Training the Model

```
# Train the model
history <- model %>% fit(
  x = X_train,
  y = y_train,
  epochs = 50,                # Number of passes through data
  batch_size = 32,            # Samples per gradient update
  validation_split = 0.2,     # Use 20% for validation
  verbose = 1                 # Show progress
)

# Plot training history
plot(history)
```

Key Parameters:

- **epochs:** How many times to iterate through entire dataset
- **batch_size:** Number of samples to use per gradient update
- **validation_split:** Fraction of training data to use for validation

Making Predictions and Evaluating I

```r
# Make predictions on test set
predictions <- model %>% predict(X_test)

# Evaluate model performance
evaluation <- model %>% evaluate(X_test, y_test)

cat("Test MSE:", evaluation$loss, "\n")
cat("Test MAE:", evaluation$mae, "\n")
```

Making Predictions and Evaluating II

```r
# Compare predictions to actual values
results <- tibble(
  Actual = y_test,
  Predicted = predictions[,1]
)

# Visualization
ggplot(results, aes(Actual, Predicted)) +
  geom_point(alpha = 0.5) +
  geom_abline(intercept = 0, slope = 1,
              color = "red", linetype = "dashed") +
  labs(title = "Predicted vs Actual Transactions",
       x = "Actual Transactions",
       y = "Predicted Transactions") +
  theme_minimal()
```

Section 7

Key Concepts Summary

What We've Learned Today

Core Concepts:

1. Neural networks learn feature interactions automatically
2. Forward propagation: computing predictions layer by layer
3. Activation functions introduce nonlinearity
4. Deeper networks learn hierarchical representations

Practical Skills:

1. Implementing forward propagation in R
2. Using common activation functions
3. Building networks with keras
4. Training and evaluating models

Next class: Optimization, backpropagation, and hyperparameter tuning

When to Use Deep Learning

Deep learning excels when:

- Large amounts of data available
- Complex nonlinear patterns exist
- Feature engineering is difficult
- Working with images, text, or sequences
- High-dimensional input data

Consider alternatives when:

- Small datasets ($n < 1000$)
- Simple linear relationships
- Interpretability is critical
- Limited computational resources
- Fast training is essential

Remember: Deep learning is a powerful tool, but not always the best tool.

Best Practices for Neural Networks

1. **Always normalize/standardize your input features**
   - Neural networks are sensitive to feature scales
   - Use `scale()` in R or preprocessing layers in keras
2. **Start with simple architectures**
   - Begin with 1-2 hidden layers
   - Add depth only if performance plateaus
3. **Use ReLU activation in hidden layers**
   - Default choice for most problems
   - Helps avoid vanishing gradient problem
4. **Monitor training and validation loss**
   - Early stopping prevents overfitting
   - Validation loss should decrease with training loss
5. **Use appropriate output activation**
   - Regression: no activation (or linear)
   - Binary classification: sigmoid
   - Multi-class: softmax

Section 8

Practice and Next Steps

Classwork Assignment

Task: Implement forward propagation for a custom network

```
# Given data
customer <- c(
  age = 45,
  income = 75000,
  accounts = 3,
  children = 2,
  balance = 50000
)
```

Network architecture:

- Input: 5 features
- Hidden layer 1: 4 nodes (ReLU)
- Hidden layer 2: 3 nodes (ReLU)
- Output: 1 node (churn probability with sigmoid)

Additional Resources

Essential Reading:

- Deep Learning with R (Chollet & Allaire) - Chapters 1-3
- Neural Networks and Deep Learning (Nielsen) - Free online book
- Keras R documentation: `https://keras.rstudio.com`

Online Tutorials:

- RStudio keras tutorials and examples
- Fast.ai Practical Deep Learning course
- DeepLearning.AI courses on Coursera

Practice Datasets:

- MNIST handwritten digits (keras built-in)
- UCI Machine Learning Repository
- Kaggle competition datasets