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

Section 1: Introduction

Why Deep Learning?

Deep learning enables machines to **automatically discover patterns** in data without extensive manual 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 network fundamentals and implementing them in R using the keras package.

Course Context

This lecture fits into our course progression:

Previous Topics:

- Basetable construction
- Feature engineering
- Traditional ML models

Today:

- Neural networks
- Deep learning
- Keras in R

Next Topics:

- Advanced architectures
- Model optimization
- Deployment

Learning Objectives

By the end of this lecture, you will be able to:

1. Understand neural network architecture and forward propagation
2. Implement activation functions in R
3. Build multi-layer neural networks
4. Use keras to train and evaluate models
5. Recognize when deep learning adds value

Section 2

Section 2: Neural Network Fundamentals

The Core Problem: Feature Interactions

Scenario: Predict bank transactions based on customer characteristics

Table 1: Sample Customer Data

Customer	Children	Accounts	Balance	Retired
A	0	1	5000	No
B	2	3	50000	Yes
C	3	2	15000	No

The Challenge: How do retired customers with high balances differ from young families with multiple accounts?

Linear Regression vs Neural Networks

Linear Regression:

- Each feature contributes independently
- Interactions must be manually specified
- Limited expressiveness
- Formula: $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots$

Neural Networks:

- Automatically learns interactions
- Hidden layers discover patterns
- Highly flexible
- Can model complex nonlinear relationships

Key Insight: Neural networks create intermediate representations that capture feature interactions automatically.

Neural Network Architecture

Basic Components:

1. **Input Layer:** Raw features (e.g., age, balance, accounts)
2. **Hidden Layer(s):** Learned intermediate representations
3. **Output Layer:** Final predictions
4. **Weights:** Parameters learned during training
5. **Activation Functions:** Introduce nonlinearity

Key Terminology:

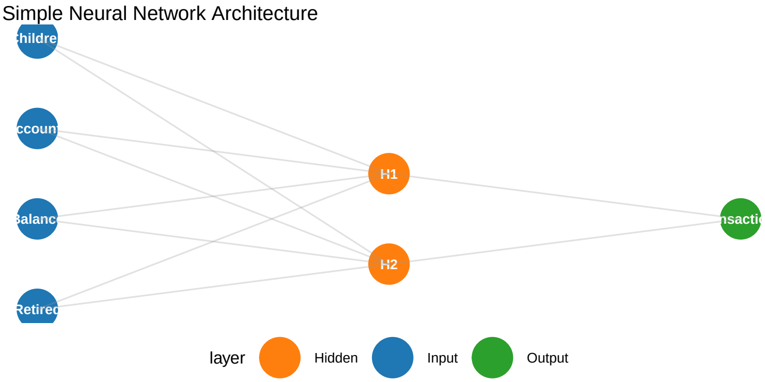
- **Neurons/Nodes:** Individual computational units
- **Layers:** Collections of neurons
- **Deep Learning:** Multiple hidden layers (depth)

Simple Network Example

Consider a network to predict transactions:

Architecture:

$$\text{Input (4 features)} \rightarrow \text{Hidden (2 nodes)} \rightarrow \text{Output (1 value)}$$



Section 3

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 Algorithm:

1. Start with input values
2. For each layer:
    - Multiply inputs by weights
    - Sum the weighted inputs
    - Apply activation function
    - Pass result to next layer
3. Output final prediction

This is how neural networks make predictions for a single observation.

Mathematical Notation

For a simple two-layer network:

Input to Hidden Layer:

$$h_j = f\left(\sum_{i=1}^{n} w_{ij}^{(1)} x_i + b_j^{(1)}\right)$$

where:

- x_i are input features
- $w_{ij}^{(1)}$ are weights from input i to hidden node j
- $b_j^{(1)}$ is the bias term
- f is the activation function
- h_j is the hidden layer activation

Hidden to Output Layer:

$$\hat{y} = g\left(\sum^{m} w_j^{(2)} h_j + b^{(2)}\right)$$

Forward Propagation Example: Setup

Given: A customer with 2 children and 3 accounts

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

# Weights for first hidden node
weights_node_0 <- c(1, 1)

# Weights for second hidden node
weights_node_1 <- c(-1, 1)

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

Task: Compute the predicted number of transactions

Forward Propagation: Hidden Layer

Computing Hidden Node 0:

```
# Weighted sum for node 0
node_0_input <- sum(input_data * weights_node_0)
cat("Node 0 input:", node_0_input, "\n")
```

```
## Node 0 input: 5
```

```
# No activation yet (linear)
node_0_value <- node_0_input
```

Computing Hidden Node 1:

```
# Weighted sum for node 1
node_1_input <- sum(input_data * weights_node_1)
cat("Node 1 input:", node_1_input, "\n")
```

```
## Node 1 input: 1
```

```
# No activation yet (linear)
```

Forward Propagation: Output Layer

Computing Final Prediction:

```
# Combine hidden layer outputs
hidden_layer <- c(node_0_value, node_1_value)
cat("Hidden layer values:", hidden_layer, "\n")
```

```
## Hidden layer values: 5 1
```

```
# Compute final output
prediction <- sum(hidden_layer * weights_output)
cat("\nFinal prediction:", prediction, "transactions\n")
```

```
##
## Final prediction: 9 transactions
```

The network predicts **9 transactions** based on our manually specified weights.

Generalizing: Forward Propagation Function

```r
# Display the forward propagation function
forward_propagate
```

```
## function (input_data, weights)
## {
##     node_0 <- sum(input_data * weights$node_0)
##     node_1 <- sum(input_data * weights$node_1)
##     hidden_layer <- c(node_0, node_1)
##     output <- sum(hidden_layer * weights$output)
##     return(output)
## }
```

Testing Forward Propagation Function

```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_propagate(input_data, weights)
cat("Prediction:", result, "\n")
```

```
## Prediction: 9
```

Section 4

Section 4: Activation Functions

The Need for Nonlinearity

Problem: Without activation functions, neural networks collapse to linear models.

Consider stacking two linear transformations:

$$y = W_2(W_1x) = (W_2W_1)x = W_{combined}x$$

This is equivalent to a single linear transformation!

Solution: Apply nonlinear activation functions after each layer:

$$y = f_2(W_2f_1(W_1x))$$

Now the network can learn complex, nonlinear patterns.

Common Activation Functions

1. ReLU (Rectified Linear Unit) - Most Popular

$$f(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

2. Tanh (Hyperbolic Tangent)

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

3. Sigmoid (Logistic)

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

Implementing Activation Functions

```r
# Display activation functions
cat("ReLU function:\n")
```

```
## ReLU function:
```

```r
relu
```

```
## function (x)
## {
##     pmax(0, x)
## }
```

```r
cat("\nSigmoid function:\n")
```

```
##
## Sigmoid function:
```

```r
sigmoid
```

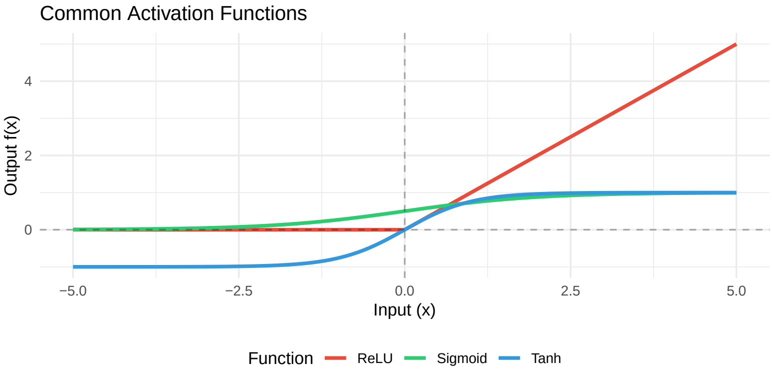
```
## function (x)
```

Testing Activation Functions

```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)
)

kable(results, digits = 3, booktabs = TRUE)
```

Input	ReLU	Tanh	Sigmoid
-2	0	-0.964	0.119
-1	0	-0.762	0.269
0	0	0.000	0.500
1	1	0.762	0.731

Visualizing Activation Functions



Activation Function Properties

Function	Range	Zero-Centered	Computational Cost	Typical Use
ReLU	[0, Inf)	No	Low	Hidden layers
Tanh	(-1, 1)	Yes	Medium	Hidden layers
Sigmoid	(0, 1)	No	Medium	Output (binary)

Rule of thumb: Use ReLU for hidden layers unless you have a specific reason not to.

Forward Propagation with ReLU

```r
# Display the function with activation
forward_prop_activated
```

```
## function (input_data, weights, activation = relu)
## {
##     node_0_input <- sum(input_data * weights$node_0)
##     node_0_output <- activation(node_0_input)
##     node_1_input <- sum(input_data * weights$node_1)
##     node_1_output <- activation(node_1_input)
##     hidden_layer <- c(node_0_output, node_1_output)
##     output <- sum(hidden_layer * weights$output)
##     return(list(hidden = hidden_layer, output = output))
## }
```

Comparing Linear vs ReLU Networks

```r
# Without activation (linear)
linear_result <- forward_propagate(input_data, weights)

# With ReLU activation
relu_result <- forward_prop_activated(input_data, weights, rel

cat("Linear network output:", linear_result, "\n")
```

```
## Linear network output: 9
```

```r
cat("ReLU network output:", relu_result$output, "\n")
```

```
## ReLU network output: 9
```

```r
cat("Hidden layer activations:", relu_result$hidden, "\n")
```

```
## Hidden layer activations: 5 1
```

Notice how ReLU sets negative values to zero, creating sparse

Section 5

Section 5: Deeper Networks

Why Add More Layers?

Single Hidden Layer Networks:

- Can theoretically approximate any function (Universal Approximation Theorem)
- May require exponentially many neurons
- Difficult to learn hierarchical patterns

Multiple Hidden Layers:

- Learn hierarchical representations
- More parameter efficient
- Better generalization on complex tasks
- Each layer captures different abstraction levels

Hierarchical Feature Learning

Image Recognition Example:

Layer	Learns
Layer 1	Edges, corners, basic textures
Layer 2	Simple shapes, object parts
Layer 3	Object components (eyes, wheels)
Layer 4	Complete objects (faces, cars)
Output	Object categories

Each layer builds upon representations from previous layers.

Multi-Layer Network Implementation

```r
# Weights for a 2-hidden-layer network
multilayer_weights <- list(
  # Input (2) to Hidden Layer 1 (2 nodes)
  h1_node0 = c(2, 4),
  h1_node1 = c(-5, -1),

  # Hidden Layer 1 (2) to Hidden Layer 2 (2 nodes)
  h2_node0 = c(-1, 1),
  h2_node1 = c(2, 4),

  # Hidden Layer 2 (2) to Output (1)
  output = c(-3, 7)
)
```

Multi-Layer Forward Propagation

```r
# Display the multi-layer function
multilayer_forward
```

```
## function (input_data, weights)
## {
##     h1_0 <- relu(sum(input_data * weights$h1_node0))
##     h1_1 <- relu(sum(input_data * weights$h1_node1))
##     hidden1 <- c(h1_0, h1_1)
##     h2_0 <- relu(sum(hidden1 * weights$h2_node0))
##     h2_1 <- relu(sum(hidden1 * weights$h2_node1))
##     hidden2 <- c(h2_0, h2_1)
##     output <- sum(hidden2 * weights$output)
##     return(list(h1 = hidden1, h2 = hidden2, output = output
## }
```

Testing Multi-Layer Network

```r
# Test the multi-layer network
result <- multilayer_forward(input_data, multilayer_weights)

cat("Hidden Layer 1:", result$h1, "\n")
```

```
## Hidden Layer 1: 16 0
```

```r
cat("Hidden Layer 2:", result$h2, "\n")
```

```
## Hidden Layer 2: 0 32
```

```r
cat("Output:", result$output, "\n")
```

```
## Output: 224
```

Network Depth Analysis

Table 4: Network Depth Comparison

Architecture	Output	Parameters
No Hidden Layers	5	2
1 Hidden Layer	9	6
2 Hidden Layers	224	10

Deeper networks can learn more complex representations with similar parameter counts.

Section 6

Section 6: Building Networks with Keras

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

Installation (one-time setup):

```r
install.packages("keras")
library(keras)
install_keras()  # Installs TensorFlow backend
```

Why Keras?

- User-friendly API
- Fast prototyping
- Production-ready
- Extensive documentation
- Large community support

Creating a Simple Keras Model

```r
library(keras)

# Define model architecture
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu",
              input_shape = c(4)) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1)

# View model structure
summary(model)
```

Architecture:

- Input: 4 features
- Hidden layer 1: 64 neurons, ReLU
- Hidden layer 2: 32 neurons, ReLU

Generating Synthetic Bank Data

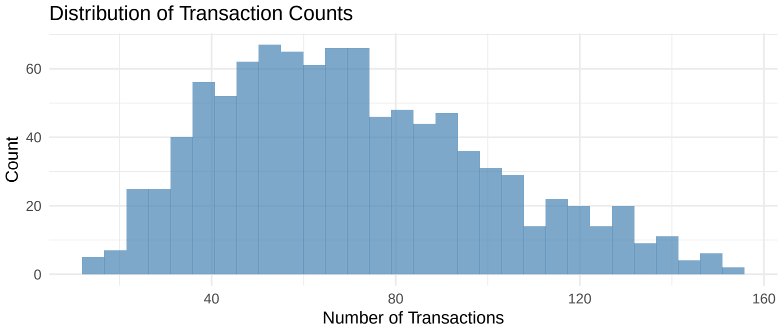
```
# Create synthetic customer dataset
set.seed(123)
n <- 1000

bank_data <- tibble(
  children = sample(0:4, n, replace = TRUE),
  accounts = sample(1:5, n, replace = TRUE),
  balance = rnorm(n, 25000, 15000),
  age = sample(25:75, n, replace = TRUE),
  retired = ifelse(age > 65, 1, 0)
) %>%
  mutate(
    # Simulate complex interaction for transactions
    transactions = 20 +
      5 * children +
      8 * accounts +
      0.0003 * balance +
```

Exploring the Data

Table 5: Bank Customer Data Summary

children	accounts	balance	age	retired	transactions
Min. :0.000	Min. :1.000	Min. :-20718	Min. :25.00	Min. :0.000	Min. : 14.31
1st Qu.:1.000	1st Qu.:2.000	1st Qu.: 15215	1st Qu.:37.00	1st Qu.:0.000	1st Qu.: 48.24
Median :2.000	Median :3.000	Median : 24646	Median :50.00	Median :0.000	Median : 67.41
Mean :1.985	Mean :2.987	Mean : 24931	Mean :50.02	Mean :0.206	Mean : 71.05
3rd Qu.:3.000	3rd Qu.:4.000	3rd Qu.: 35420	3rd Qu.:63.00	3rd Qu.:0.000	3rd Qu.: 90.68
Max. :4.000	Max. :5.000	Max. : 74358	Max. :75.00	Max. :1.000	Max. :153.37

Data Distribution



Data Preprocessing

```r
# Separate features and target
X <- bank_data %>%
  select(children, accounts, balance, age, retired) %>%
  as.matrix()

y <- bank_data$transactions

# Standardize features (critical for neural networks!)
X_scaled <- scale(X)

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

X_train <- X_scaled[train_idx, ]
y_train <- y[train_idx]
```

Building the Keras Model

```r
library(keras)

# Define architecture
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 model
model %>% compile(
  optimizer = "adam",           # Adaptive learning rate
  loss = "mse",                 # Mean squared error
  metrics = c("mae")            # Mean absolute error
)

# View architecture
```

Model Architecture Summary

When you run `summary(model)`, you'll see:

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	384
dense_1 (Dense)	(None, 32)	2080
dense_2 (Dense)	(None, 1)	33
Total params		**2,497**

Parameter calculation:

$$(5 \times 64 + 64) + (64 \times 32 + 32) + (32 \times 1 + 1) = 2,497$$

Training the Model

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

# Plot training history
plot(history)
```

Training Parameters Explained

Key hyperparameters:

- **epochs**: Number of complete passes through training data
  - Too few: underfitting
  - Too many: overfitting
- **batch_size**: Samples processed before weight update
  - Smaller: more updates, noisier gradients
  - Larger: fewer updates, smoother gradients
- **validation_split**: Fraction for monitoring overfitting
  - Typical: 0.1 to 0.3

Making Predictions

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

# Evaluate model
evaluation <- model %>% evaluate(X_test, y_test, verbose = 0)

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

# Create results dataframe
results <- tibble(
  Actual = y_test,
  Predicted = as.vector(predictions)
)

head(results, 10)
```

Visualizing Model Performance

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

Residual Analysis

```r
# Calculate residuals
results <- results %>%
  mutate(Residual = Actual - Predicted)

# Residual plot
ggplot(results, aes(x = Predicted, y = Residual)) +
  geom_point(alpha = 0.5) +
  geom_hline(yintercept = 0, color = "red",
             linetype = "dashed") +
  labs(title = "Residual Plot",
       x = "Predicted Values",
       y = "Residuals (Actual - Predicted)") +
  theme_minimal()
```

Section 7

Section 7: Best Practices

When to Use Deep Learning

Deep learning excels when:

- Large datasets $(n > 10,000)$
- Complex nonlinear relationships
- High-dimensional inputs
- Feature engineering is difficult
- Images, text, or sequential data

Consider alternatives when:

- Small datasets $(n < 1,000)$
- Simple linear relationships
- Interpretability is critical
- Limited computational resources
- Fast training is essential

Neural Network Best Practices

1. **Always normalize input features**
   - Use `scale()` in R or keras preprocessing layers
   - Neural networks are sensitive to feature scales
2. **Start simple, then add complexity**
   - Begin with 1-2 hidden layers
   - Add layers/neurons only if needed
3. **Use appropriate activation functions**
   - Hidden layers: ReLU (default)
   - Binary output: sigmoid
   - Multi-class output: softmax
   - Regression output: none (linear)

Training Best Practices

4. Monitor training and validation loss

- Both should decrease together
- Divergence indicates overfitting
- Use early stopping

5. Use appropriate batch sizes

- Too small: unstable training
- Too large: poor generalization
- Typical: 32, 64, 128

6. Tune learning rate carefully

- Too high: divergence
- Too low: slow convergence
- Use adaptive optimizers (Adam)

Common Pitfalls to Avoid

1. **Forgetting to normalize features** - Leads to poor convergence
2. **Using too many parameters** - Causes overfitting on small datasets
3. **Not using validation data** - Cannot detect overfitting
4. **Training for too many epochs** - Overfits to training noise
5. **Inappropriate activation for output** - Wrong predictions
6. **Not setting random seeds** - Non-reproducible results

Section 8

Section 8: Summary and Practice

Key Concepts Review

Core Concepts:

1. Neural networks learn feature interactions automatically
2. Forward propagation computes predictions layer-by-layer
3. Activation functions introduce essential nonlinearity
4. Deeper networks learn hierarchical representations
5. Keras provides a high-level API for building models

Practical Skills:

1. Implementing forward propagation manually
2. Using activation functions correctly
3. Building and training keras models
4. Evaluating model performance

Classwork Assignment

Task: Build a customer churn prediction model

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

Requirements:

1. Implement manual forward propagation (2 hidden layers, ReLU)
2. Output layer uses sigmoid for churn probability
3. Build equivalent keras model
4. Compare manual vs keras predictions

Assignment Details

Architecture:

- Input: 5 features (age, income, accounts, children, balance)
- Hidden Layer 1: 4 nodes, ReLU activation
- Hidden Layer 2: 3 nodes, ReLU activation
- Output: 1 node, sigmoid activation (churn probability)

Deliverables:

1. R script with manual implementation
2. Keras model code
3. Comparison of results
4. Brief interpretation (2-3 sentences)

Due: Next class session

Assignment Grading Rubric

Component	**Points**
Manual forward propagation (correct)	30
Proper activation function usage	20
Keras model architecture (correct)	20
Model training and evaluation	15
Code quality and documentation	10
Interpretation of results	5
Total	**100**

Additional Resources

Essential Reading:

- *Deep Learning with R* (Chollet & Allaire) - Chapters 1-4
- *Neural Networks and Deep Learning* (Nielsen) - Free online
- Keras R documentation: keras.rstudio.com

Online Resources:

- TensorFlow tutorials: tensorflow.rstudio.com
- Fast.ai Practical Deep Learning course
- DeepLearning.AI specialization (Coursera)

Practice Datasets:

- MNIST digits (keras built-in)
- UCI Machine Learning Repository
- Kaggle competitions