Lab Report #6: Neural Networks
Course: RSM338H1S, Winter 2026

Instructions:

- This assignment may be completed in **groups of up to 3 students**. Group members may be from either section. If you work in a group, submit one report with all names listed.
- Submit your lab report as a **PDF** to Crowdmark via Quercus. Export your Jupyter Notebook to PDF before uploading. Only one group member should upload the submission, being sure to select their group mates at the time of submission.
- There is no page limit, but be concise. A good report is thorough but not padded.

Marking:

- **75%** — Coding and results (correct implementation, complete answers to all parts, appropriate choice of methods, accurate numerical output, properly formatted tables and figures)
- **25%** — Overall quality (clear and professional writing, thoughtful interpretation of results, demonstrated understanding of the underlying concepts, logical flow and narrative structure)

Writing Expectations: Your report should read as a **coherent narrative**, not just code with scattered comments. Use section headers to indicate which problem you're working on. Before each code block, briefly explain what you are about to do and why. After results appear, interpret what you see. A reader should be able to understand your analysis even if they skipped the code cells.

You may use AI coding assistants (ChatGPT, Copilot, Claude, etc.) to help write code, but you must be able to explain what every line does. The text you write around the code is what demonstrates your understanding. You are ultimately responsible for your own work. **If you use an AI tool, you must disclose this in a note at the end of your report. Mention which tool you used, which tasks you asked it to complete, and discuss your (dis)satisfaction with its assistance.**

Assignment: This assignment has two parts. In Part A, you revisit the Lending Club loan default problem from Lab Report #5—this time using a neural network built in PyTorch. In Part B, you use a neural network to learn the Black–Scholes option pricing function from synthetic data. Together, the two parts illustrate both the limitations and the strengths of neural networks: they struggle on small tabular datasets where simpler models already work well, but they excel at learning complex nonlinear functions when given enough data.

Part A: Classification

You will use your cleaned Lending Club dataset from Lab Report #5. If you are working with a different group than HW5, pick one group member's cleaned data and use it consistently.

Problem 1: Data Preparation

Reuse your data preparation pipeline from Lab Report #5: load the data, define the target variable, select features, handle missing values and categorical encoding, and standardize.

Tasks:

(a) Include your data preparation code and confirm the final dimensions of your training and test sets.

(b) Convert your data to **PyTorch tensors**. Split your data 60/20/20 into training, validation, and test sets. Shuffle before splitting. Report the number of observations in each set. Explain the rationale for using three sets instead of two.

Problem 2: Build and Train a Baseline Neural Network

Build a feed-forward neural network in PyTorch to classify loan defaults. Use the following architecture:

- Two hidden layers, each with 64 neurons
- ReLU activation after each hidden layer
- Sigmoid activation on the output (single neuron)
- Binary cross-entropy loss (`nn.BCELoss`)
- Adam optimizer with learning rate 0.001
- Batch size 64
- 500 epochs

During training, track the best model weights—the weights from the epoch with the lowest validation loss.

Tasks:

(a) Report the total number of trainable parameters in your model. How does this compare to the number of training observations?

(b) Train the model for 500 epochs. Plot the **training and validation loss curves** (both on the same figure, loss on the y-axis, epoch on the x-axis).

(c) Describe what you see in the loss curves. At roughly what epoch does the validation loss start increasing? What is this phenomenon called, and what does it mean?

(d) Load the best model weights (from the epoch with the lowest validation loss) and evaluate on the test set. Report **accuracy**, **recall**, and **AUC**. Display the confusion matrix.

Problem 3: Experimentation

The baseline model from Problem 2 almost certainly overfits. Try to improve its generalization performance. You have freedom to experiment—here are some directions to consider:

- **Architecture:** Fewer or more neurons, fewer or more layers
- **Regularization:** Dropout (`nn.Dropout`), weight decay (the `weight_decay` parameter in Adam)
- **Learning rate and batch size**
- **Number of epochs** and early stopping

Tasks:

(a) Try at least **three** different configurations. For each one, state what you changed and why, show the loss curves, and report the test set metrics (accuracy, recall, AUC).

(b) Which configuration performed best? Summarize your findings in a table.

Problem 4: Comparison to Lab Report #5

(a) Re-fit **Logistic Regression**, **Random Forest**, and **XGBoost** on the same training data (you can reuse your HW5 code and hyperparameters). Evaluate them on the same test set using accuracy, recall, and AUC.

(b) Produce a **summary table** comparing all models side by side: the three `sklearn` models and your best neural network.

(c) Plot the **ROC curves** for all four models on a single figure.

(d) Does the neural network outperform, match, or underperform the `sklearn` models? Why do you think this is the case? In your answer, discuss what characteristics of a dataset and problem make neural networks more or less likely to outperform simpler methods.

Part B: Regression

In Part A, the neural network was applied to a small tabular dataset where simpler models already perform well. In Part B, we move to a setting where neural networks have a clear advantage: learning a known nonlinear function from a large dataset.

Problem 5: Learning the Black–Scholes Formula

A **call option** is a contract that gives its holder the right (but not the obligation) to buy a stock at a predetermined price K (the *strike price*) at a future date T (the *time to maturity*). If the stock price ends up above K, the option is valuable; if it ends up below K, the option expires worthless. The question is: what should this contract be worth *today*?

The **Black–Scholes formula** answers this question. It is a function of five inputs—the current stock price S_0, the strike price K, the time to maturity T, the risk-free interest rate r, and the stock's volatility σ—that outputs the fair price of the option:

$$C(S_0, K, T, r, \sigma) = S_0\,\Phi(d_1) - K\,e^{-rT}\,\Phi(d_2)$$

where:

$$d_1 = \frac{\ln(S_0/K) + (r + \sigma^2/2)\,T}{\sigma\sqrt{T}}, \qquad d_2 = d_1 - \sigma\sqrt{T}$$

and $\Phi(\cdot)$ is the standard normal CDF (`scipy.stats.norm.cdf` in Python).

Don't worry if you haven't seen this formula before—the details don't matter for this problem. The point is that it is a **known nonlinear function**: five numbers go in, one number comes out, and the relationship between them is complex. We are going to use it as a convenient test case to see how well a neural network can learn a nonlinear function when we have plenty of data and know the true answer.

Your task is to train a neural network C_{NN} that approximates the Black–Scholes function:

$$C_{\text{NN}}(S_0, K, T, r, \sigma) \;\approx\; C(S_0, K, T, r, \sigma)$$

The inputs to the network are (S_0, K, T, r, σ) and the output is a single number: the predicted call price.

Tasks:

(a) **Generate training data.** Write a function that computes Black–Scholes call prices. Then generate 100,000 random option contracts by sampling the inputs uniformly (following Hull, Table 6.6):

- $S_0 \in [40, 60]$
- $K \in [0.5\,S_0,\ 1.5\,S_0]$

- $T \in [0.25, 2.0]$ (years)
- $r \in [0, 0.05]$
- $\sigma \in [0.10, 0.40]$

Note that K depends on S_0—sample S_0 first, then sample K relative to it. This keeps the moneyness range realistic.

Compute the Black–Scholes price for each contract. Split the data into training (60%), validation (20%), and test (20%) sets. Standardize the inputs using the training set statistics.

(b) **Fit a linear regression** (`sklearn.linear_model.LinearRegression`) as a baseline. Report the test set R^2 and mean squared error.

(c) **Build and train a neural network** to predict call prices. Use the following setup:

- Mean squared error loss (`nn.MSELoss`)
- Adam optimizer
- **Tanh** activations in the hidden layers (`nn.Tanh`)
- **Softplus** activation on the output layer (`nn.Softplus`)
- You choose the number of layers, neurons per layer, learning rate, and number of epochs

Two activation functions are new here. In Part A we used ReLU, which is piecewise linear—fine for classification, but a poor match for approximating a smooth function like Black–Scholes. Instead:

Tanh (hidden layers) is a smooth S-shaped function that outputs values between -1 and $+1$:

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

Because tanh is smooth, the network's output is also smooth—it can approximate the curvature of the Black–Scholes surface naturally, without needing a huge number of neurons.

Softplus (output layer) ensures the network's output is always positive:

$$\text{softplus}(z) = \ln(1 + e^z)$$

This is a smooth approximation of ReLU that is always > 0. We use it on the output because option prices are never negative—a call option is worth at least zero. Without this constraint, the network could predict negative prices for deep out-of-the-money options.

Choose an architecture and train the model, plotting the training and validation loss curves.

(d) Report the test set R^2 and MSE for the neural network. How does it compare to linear regression?

(e) Create a scatter plot of **predicted vs. actual** Black–Scholes prices on the test set, for both linear regression and the neural network. Include the 45-degree line for reference. What do these plots tell you about each model's ability to learn the pricing function?

(f) Create **partial dependence plots** for both models: for each of the five inputs (S_0, K, T, r, σ), vary that input across its range while holding the other four at their median values. Plot the predicted call price against the varying input. Overlay the true Black–Scholes price for comparison. Discuss: which inputs does the neural network capture well? Where does linear regression break down? Do the shapes match your intuition about how option prices depend on each input?

(g) The Black–Scholes formula is a known, deterministic function—there is no noise. In principle, a sufficiently flexible model could achieve perfect predictions ($R^2 = 1$). How close does your neural network get? In reality, why might this approach still be insufficient for real options traders, and how would you propose adjusting the approach?

