Assignment 3: Predicting Mapping Penalties with a From-Scratch ANN

Due: June 5, 2025, 11:59 PM

1 Introduction and Learning Objectives

In this assignment, you will implement a feed-forward artificial neural network (ANN) *from scratch* (no TensorFlow, PyTorch, or other high-level libraries) to predict the penalty score of a mapping between tasks and employees.

By the end of this assignment, you will:

1. Preprocess and encode structured data into a fixed-length input vector.
2. Design and implement two ANN architectures:
   - **Model A**: single hidden layer with 256 neurons.
   - **Model B**: two hidden layers, each with 128 neurons.
3. Implement forward propagation, backpropagation, and parameter updates manually.
4. Train your networks using Mean Squared Error (MSE) loss.
5. Evaluate and compare model performance across architectures and hyperparameters.
6. Generate and interpret three key comparison graphs in your report.

2 Problem Context and Data Description

A company has 10 tasks and 5 employees. Each mapping—an assignment of every task to an employee—carries a real-valued penalty score reflecting five constraint violations. You can generate 100 such mappings using the codes of your Assessment Task 1, and save them in a file, with the penalty score calculated.

2.1 Task Data and Employee Data (synthetic example)

Task ID	Time (hrs)	Difficulty	Deadline (hrs)	Required Skill
T1	4	3	8	A
T2	6	5	12	B
T3	2	2	6	A
T4	5	4	10	C
T5	3	1	7	A
T6	8	6	15	B
T7	4	3	9	C
T8	7	5	14	B
T9	2	2	5	A
T10	6	4	11	C

Table 1: Synthetic Task Data

Employee ID	Available Hrs	Skill Level	Skills
E1	10	4	A,C
E2	12	6	A,B,C
E3	8	3	A
E4	15	7	B,C
E5	9	5	A,C

Table 2: Synthetic Employee Data

2.2 Sample Mapping and Penalty

```
Mapping: { (T1→E2), (T2→E3), (T3→E1), (T4→E4), (T5→E2), (T6→E5), (T7→E1), (T8→E3), (T9→E5), (T10→E4) }
Penalty: 5.00
```

3 Preprocessing, Encoding & Input Vector

Each input to the network must be a fixed-length numeric vector. Follow these steps:

1. **Numeric features (time, difficulty, deadline)**: Directly use as numeric inputs.
2. **Categorical features (skill)**: Use one-hot encoding. For skills 'A', 'B', 'C', the encodings would be:
   - 'A' $\rightarrow [1, 0, 0]$
   - 'B' $\rightarrow [0, 1, 0]$
   - 'C' $\rightarrow [0, 0, 1]$
   - 'A' & 'C' $\rightarrow [1, 0, 1]$

3.1 Constructing the Input Vector

To train your neural network, you need to convert your task-employee assignments into numerical vectors. Follow these two clear steps:

1. **Create a Feature Vector for Each Task-Employee Pair:**
   For each task T_i assigned to an employee E_j, construct an 11-dimensional numeric vector consisting of:
   - **Task Features (6 elements):**
     - Estimated Time Required (`Time`)
     - Difficulty Level (`Diff`)
     - Deadline (`Dead`)
     - Required Skill (One-hot encoded for Skills A, B, C)
   - **Employee Features (5 elements):**
     - Available Hours (`Avail`)
     - Skill Level (`Level`)
     - Employee Skills (One-hot encoded for Skills A, B, C)

   Thus, each pair (T_i, E_j) is represented as:

$$\mathbf{f}_{ij} = [\text{Time}_i, \text{Diff}_i, \text{Dead}_i, \text{ReqSkillOneHot}_i^{(3)}, \text{Avail}_j, \text{Level}_j, \text{EmpSkillsOneHot}_j^{(3)}]$$

2. **Combine All Pair-Vectors into One Input Vector:**
   Concatenate the feature vectors of all 10 task-employee pairs in the fixed order from Task T_1 to T_{10} into a single vector:

$$\mathbf{x} = [\mathbf{f}_{1\,a_1}, \mathbf{f}_{2\,a_2}, \ldots, \mathbf{f}_{10\,a_{10}}]$$

   Since each pair vector has 11 elements, the total input size is:

$$10 \text{ tasks} \times 11 \text{ features per pair} = 110 \text{ features}$$

Example:
Consider the following two task-employee assignments:

$$T1 \to E2 : [4, 3, 8,\ 1, 0, 0,\ 12, 6, 1, 1, 1],$$

$$T2 \to E3 : [6, 5, 12,\ 0, 1, 0,\ 8, 3, 1, 0, 0],$$

$$\vdots$$

Repeat this for all 10 task-employee pairs to form your full input vector.

4 ANN Implementation

For this assignment, you need to implement two different feed-forward architectures. In both cases, the input layer size is determined by the flattened input vector length, and the output layer has one neuron (predicting the penalty score).

- **Model A:**

$$\underbrace{110}_{\text{input}} \longrightarrow \underbrace{256}_{\text{hidden}_1} \longrightarrow \underbrace{1}_{\text{output}}$$

  - *Example dimensions:*

$$W^{(1)} \in \mathbb{R}^{256\times 110}, \quad b^{(1)} \in \mathbb{R}^{256}, \quad W^{(2)} \in \mathbb{R}^{1\times 256}, \quad b^{(2)} \in \mathbb{R}^{1}.$$

- **Model B:**

$$\underbrace{110}_{\text{input}} \longrightarrow \underbrace{128}_{\text{hidden}_1} \longrightarrow \underbrace{128}_{\text{hidden}_2} \longrightarrow \underbrace{1}_{\text{output}}$$

  - *Example dimensions:*

$$W^{(1)} \in \mathbb{R}^{128\times 110},\ b^{(1)} \in \mathbb{R}^{128},\ W^{(2)} \in \mathbb{R}^{128\times 128},\ b^{(2)} \in \mathbb{R}^{128},\ W^{(3)} \in \mathbb{R}^{1\times 128},\ b^{(3)} \in \mathbb{R}^{1}.$$

4.1 Activation Functions

We need non-linear activations in the hidden layers to learn complex mappings. Two common choices:

- **Sigmoid:**

$$\sigma(x) = \frac{1}{1+e^{-x}}, \quad \sigma'(x) = \sigma(x)\big(1-\sigma(x)\big).$$

  Example: If $z = 0.5$, then

$$\sigma(0.5) = \frac{1}{1+e^{-0.5}} \approx 0.62, \quad \sigma'(0.5) \approx 0.62\,(1-0.62) = 0.24.$$

- **ReLU:**

$$\mathrm{ReLU}(x) = \max(0, x), \quad \mathrm{ReLU}'(x) = \begin{cases} 1, & x > 0, \\ 0, & x \le 0. \end{cases}$$

  Example: If $z = -1$, $\mathrm{ReLU}(-1) = 0$, and $\mathrm{ReLU}'(-1) = 0$; if $z = 2$, $\mathrm{ReLU}(2) = 2$, and $\mathrm{ReLU}'(2) = 1$.

For the **output layer**, since we predict a real-valued penalty, we use a *linear* activation:

$$f(x) = x, \quad f'(x) = 1.$$

4.2 Feed-Forward Propagation

Denote:

$$a^{(0)} \in \mathbb{R}^{110} \quad \text{(flattened input)}, \quad \text{and for each layer } l = 1, \ldots, L: \quad z^{(l)} = W^{(l)}\, a^{(l-1)} + b^{(l)}, \quad a^{(l)} = g^{(l)}\big(z^{(l)}\big).$$

Step-by-step example (Architecture 1):

1. *Input:* Suppose
$$a^{(0)} = [\,0.2,\ 0.5,\ \ldots,\ 0.1\,]^T \in \mathbb{R}^{110}.$$
2. *Layer 1 (hidden, 256 neurons):*
$$z^{(1)} = W^{(1)}\, a^{(0)} + b^{(1)}, \quad a^{(1)} = g\big(z^{(1)}\big),$$
where $W^{(1)}$ has shape (256×110). Concretely, for neuron k:
$$z_k^{(1)} = \sum_{i=1}^{110} W_{k,i}^{(1)}\, a_i^{(0)} \ + \ b_k^{(1)}, \quad a_k^{(1)} = g\big(z_k^{(1)}\big).$$
3. *Layer 2 (output, 1 neuron):*
$$z^{(2)} = W^{(2)}\, a^{(1)} + b^{(2)}, \quad \hat{y} = a^{(2)} = z^{(2)} \quad \text{(linear)}.$$

4.3 Backpropagation Algorithm

We use Mean Squared Error (MSE) as the loss:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \big(y_i - \hat{y}_i\big)^2.$$

For a single example, the loss is $(y - \hat{y})^2$.

Error at the output layer ($L = 2$):

$$\delta^{(2)} = \frac{\partial \mathcal{L}}{\partial z^{(2)}} = 2\,(\hat{y} - y) \ \cdot\ f'\big(z^{(2)}\big).$$

Since $f'(z^{(2)}) = 1$ (linear), we get

$$\delta^{(2)} = 2\,(\hat{y} - y).$$

Example: if $y = 3.5$ and $\hat{y} = 2.8$, then

$$\delta^{(2)} = 2\,(2.8 - 3.5) = -1.4.$$

Error in hidden layer ($l = 1$):

$$\delta^{(1)} = \big(W^{(2)}\big)^T \delta^{(2)} \ \odot\ g'\big(z^{(1)}\big),$$

where $\odot$ is element-wise multiplication.

Example (first hidden neuron):

$$\delta_1^{(1)} = W_{1,\,1}^{(2)}\, \delta^{(2)} \ \times\ g'\big(z_1^{(1)}\big).$$

Gradients for weights and biases:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)}\ (a^{(l-1)})^T, \quad \frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)}.$$

Thus each entry

$$\frac{\partial \mathcal{L}}{\partial W_{k,i}^{(l)}} = \delta_k^{(l)}\ a_i^{(l-1)}, \qquad \frac{\partial \mathcal{L}}{\partial b_k^{(l)}} = \delta_k^{(l)}.$$

Gradient descent update:

$$W^{(l)} \leftarrow W^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial W^{(l)}}, \quad b^{(l)} \leftarrow b^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial b^{(l)}},$$

where α is the learning rate (e.g. 0.01).

Numeric example for one weight:

If $W_{1,1}^{(2)} = 0.5$, $\delta^{(2)} = -1.4$, and $a_1^{(1)} = 0.62$, then

$$\frac{\partial \mathcal{L}}{\partial W_{1,1}^{(2)}} = -1.4 \times 0.62 = -0.868, \quad W_{1,1}^{(2)} \leftarrow 0.5 - 0.01\,(-0.868) = 0.5087.$$

5 Training & Evaluation Protocol

Data Splitting Given approximately 100 labelled mappings, we partition the data into three non-overlapping sets:

$$\text{Training} : \text{Validation} : \text{Test} = 70\% : 15\% : 15\%.$$

- Assign the first 70 mappings to the **training set**, the next 15 to the **validation set**, and the final 15 to the **test set**.

Hyperparameter Grid We will perform a grid search over the following hyperparameters:

Hyperparameter	**Values**	**Notes**
Learning rate α	$\{0.01,\ 0.001,\ 0.0001\}$	Controls step size in gradient descent.
Batch size	$\{8,\ 16,\ 32\}$	Number of samples per weight update.
# Epochs	100 to 200	Full passes over the training set.
Activation function	{sigmoid, ReLU}	Hidden-layer nonlinearity.

Table 3: Hyperparameter search space.

Training Loop For each combination of (α, batch size, activation):

1. Initialize all weights $W^{(l)}$ with small random values (e.g. Gaussian $\mathcal{N}(0,\ 0.01)$) and biases $b^{(l)} = 0$.
2. **For** epoch = 1 to E **do**
   (a) *Shuffle* the training set.
   (b) Partition into mini-batches of the chosen size.
   (c) **For each** mini-batch:
      - Perform *forward propagation* to compute predictions.
      - Compute batch loss (MSE).
      - Perform *backpropagation* to compute gradients.
      - Update weights and biases via gradient descent with learning rate α.
   (d) At the end of the epoch:
      - Evaluate *training loss* on the entire training set.
      - Evaluate *validation loss* on the validation set.
      - Measure and record the *epoch time* (e.g. wall-clock seconds).
3. After all epochs, evaluate final model on the **test set**.

Recorded Metrics For each epoch e, log:

$$\left(e,\ \mathcal{L}_{\text{train}}^{(e)},\ \mathcal{L}_{\text{val}}^{(e)},\ t_{\text{epoch}}^{(e)}\right).$$

- $\mathcal{L}_{\text{train}}^{(e)}$: Mean Squared Error on training set.
- $\mathcal{L}_{\text{val}}^{(e)}$: Mean Squared Error on validation set.
- $t_{\text{epoch}}^{(e)}$: Time (in seconds) to complete epoch e.

Activation Functions Compare two choices in the hidden layers:

$$\text{Sigmoid}(x) = \frac{1}{1+e^{-x}}, \quad \text{ReLU}(x) = \max(0, x).$$

- *Sigmoid* can saturate for large $|x|$, slowing convergence.
- *ReLU* is sparse (zero for $x \leq 0$) and often yields faster training.
- Keep the *output layer* activation *linear* (identity).

Required Comparison Graphs

Include the following graphs for both Model A and B:

1. Epoch vs. Loss (train & val) [on specific learning rate, batch size and activation function]
2. Learning Rate vs Loss (train & val) [on specific epoch, batch size and activation function]
3. Activation Function vs Loss (train & val) [on specific epoch, batch size and learning rate]
4. Batch Size vs. Epoch Time [on specific learning rate and activation function]

Report Template

Introduction

Context and motivation for predicting mapping penalties.

Methodology

Data Description

Overview of task & employee features. Brief description of how 100 sample mappings were generated.

Preprocessing & Encoding

Numeric vs. one-hot encoding. Construction of the 110-dimensional input vector.

Model Architectures

Model A, Model B.

Training Procedure

Loss function, activation functions, hyperparameter grid, data split (70/15/15).

Results

Model A:

- **Figure 1:** Epoch vs. Loss (train & val)
- **Figure 2:** Learning Rate vs. Loss
- **Figure 3:** Activation Function vs. Loss
- **Figure 4:** Batch Size vs. Epoch Time

Model B: (same four figures)
Each figure should include a caption and a comprehensive discussion.

Discussion

Compare Model A vs. Model B based on the figures. Highlight trade-offs and insights.

Implementation Details

Google Colab Link: `https://colab.research.google.com/your-notebook-link` Environment & dependencies (NumPy, pandas, matplotlib, etc.). Instructions for running cells.

Formatting & Appendices

Appendix A: Sample Mappings

List of 100 mappings (task→employee + penalty).

Appendix B: Additional Tables or Code Snippets

Code Template

Listing 1: Google Colab Notebook Structure

```python
# 1. Notebook Setup
# Title, assignment info, and markdown overview.

# 2. Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# (No TensorFlow/PyTorch!)

# 3. Data Generation/Load
# - Generate or load the 100 mappings CSV
# - Load into pandas DataFrame

# 4. Preprocessing
def one_hot_encode_skill(skills):
    # returns a 3-element vector
    ...

def construct_input_vector(mapping_row):
    # builds 110-dim vector for one example
    ...

# 5. Model Definitions
class NeuralNetwork:
    def __init__(self, layer_dims, activation='relu'):
        ...
    def forward(self, x):
        ...
    def backward(self, x, y_true):
        ...
    def update_params(self, lr):
        ...

# 6. Training Loop
def train(model, X_train, y_train, params):
    # implement mini-batch SGD, record loss
    ...

# 7. Evaluation & Plots
# - Generate the eight required figures
# - Save each via plt.savefig()

# 8. Save & Export
# - Download figures
# - Optionally, pickle model parameters
```

Once complete, click File → Download .zip to produce your ZIP upload.

Marking Rubric

Component	Criteria	Marks
Report (60 marks)		
Figures & Discussion	8 figures $\times$ 5 marks each, captions & discussion	40
Formatting & Presentation	Clear layout and consistency	10
Appendix & Dataset Listing	Complete list of 100 mappings	10
Code (40 marks)		
Notebook Functionality	Runs end-to-end; reproduces figures	15
Implementation Correctness	All required functions & specs implemented	15
Code Quality & Comments	Structure & descriptive comments	10
Penalties & Deductions		
AI-Generated Content	Turnitin similarity >20%: deduct (similalrity-20)% of 60 report marks	See note
Disallowed Libraries	Any TensorFlow/PyTorch import: zero for Code component	Severe

AI Penalty Note: If Turnitin reports X% similarity and X > 20%, then

$$\text{Marks Deducted} = (X - 20)\% \times 60.$$

Therefore, please do not use AI chatbots to write your report