ECE648 – Machine Learning

Final Project Report, Spring 2025

Dr. Xiaodong Cai

Saurav Luthra

55027009

I - Introduction:

Neural networks are a class of machine learning models inspired by the human brain's structure, capable of learning complex patterns from large datasets. At their core, they consist of interconnected layers of nodes (or "neurons") that process and transform data to perform tasks such as classification, regression, or generation. Among the various types of neural networks, recurrent neural networks (RNNs) are especially suited for sequence data, as they include internal memory states that allow them to retain information across time steps. This makes RNNs particularly effective for prediction tasks where the order and context of inputs are crucial, such as time series forecasting, language modeling, or in this case, biological sequence analysis.

In this project, I worked with a genomic data file containing over 5.9 million nucleotide bases represented by the characters A, C, T, and G. These bases were first encoded as integers and fed into a recurrent neural network to perform character-level prediction—predicting the next nucleotide in a sequence given the preceding context. Applications of such a system in the medical and genomic fields include early detection of mutations, assisting in genome assembly, and improving understanding of regulatory patterns in DNA, all of which have implications for disease research and personalized medicine. The RNN model used a Long Short-Term Memory (LSTM) layer to capture long-range dependencies, followed by a fully connected layer, with cross-entropy loss as the learning objective. The input and target sequences were offset by one character to create a supervised learning framework. A sequence length of 200 characters was chosen, striking a balance between model complexity, memory constraints, and sufficient context for learning.

The second phase of the project extended this approach by segmenting the genome into k-mers(subsequences of length k) which were also encoded from 0–255. In bioinformatics, k-mers are crucial for tasks like genome assembly, motif discovery, and comparative genomics. By analyzing these k-mers rather than individual characters, the model could potentially learn higher-level structural or functional patterns in DNA. The same architecture was initially used, followed by a third, more advanced model that incorporated multiple LSTM layers, deeper fully connected layers.

II - Description of Designs and Computer Experiments:

The full genome file sequence was first read into a string, from where it was processed into encoded characters and encoded K-Mers. First, every character was individually encoded into an integer (A=1, C=2, T=3, G=4) and then using DataLoader and the custom TextDataSet class, they were arranged into pairs of input sequences and target sequences which could be batched together and then fed to the neural networks. For the K-Mers, firstly the full string was split into text K-Mers of length 4, and then encoded using a custom function, kmer_to_int, that encoded its value from 0-255 (representing 4^4 permutations of characters). Similarly, the encoded K-Mer sequences were arranged into inputs and targets using TextDataSet and DataLoader. Both datasets were split using traintestsplit, allowing 20% of the dataset to be used for model accuracy testing.

A - Model 1/2 (Single Character Prediction/K-Mer Prediction):

The core model used in part 1 is a Long Short-Term Memory (LSTM)-based recurrent neural network (RNN) designed to perform character-level prediction on genomic sequences. The objective was to predict the next nucleotide (A, C, T, or G) in a DNA sequence given the preceding context/the next K-Mer of length 4 (e.g. 'ACTG', encoded into 1 of 256 integers). The input data consists of sequences of vocabulary size of 4 or 256. These integers are first passed through an embedding layer with an embedding dimension of 128/512, which is quite high in terms of the vocabulary. This layer learns a dense, continuous representation for each input, allowing the model to capture subtle relationships and patterns that would be lost with one-hot encoding. The choice of an embedding size balances power with computational efficiency; smaller values did not capture enough nuance, while larger values led to overfitting or unnecessary memory usage.

The embedded inputs are then fed into an LSTM layer with a hidden size of 1024. This relatively large hidden state enables the model to learn long-range dependencies across the input sequence, which is particularly important in genomics where certain regulatory patterns or motifs can span dozens or even hundreds of base pairs. The sequence length was set to 200 (10, 50, and 100 were tried as well), which empirically provided sufficient context for the model to learn meaningful patterns while keeping memory usage manageable during training.

Training was conducted with a batch size of 512, which offered a good trade-off between GPU memory utilization and training speed. A learning rate of 0.001 (0.01 and 0.1 were tried) was used with the CrossEntropyLoss function as the objective, and training ran for 10000/15000 epochs (also tried 100, 1000, 5000). Despite the extensive training time, the loss continued to decrease without plateauing, indicating that the model was still learning for part 1. This suggests that either a higher learning rate could accelerate convergence, or that even more training epochs might be needed to reach optimal performance. Given more hardware capability, future improvements could include adaptive learning rates, or more epochs. In part 2, it seems that the loss converged. Overall, these architectures and hyperparameter configurations enabled the model to learn and predict patterns with an accuracy of about 66%/78%.

B - Model 3 (K-Mer Prediction with More Complexity):

The primary changes made to the model involve increasing its depth, width, and regularization capabilities, all of which contribute to making the model more powerful and capable of learning more complex patterns. Firstly, multiple LSTM layers were introduced by setting the num_layers parameter to 3. By default, the original model had only one LSTM layer, which means it could only capture relatively shallow temporal dependencies in the data. Adding more layers allows the model to learn hierarchical representations, where each layer can focus on progressively more abstract features. This multi-layer architecture mimics deeper neural network models that are capable of representing increasingly sophisticated patterns, which is particularly useful for complex tasks like sequence prediction in genetic data.

Secondly, the LSTM hidden size was increased to enhance the model's capacity to store and process information at each timestep. In the original model, the hidden size was set to a relatively small value, limiting the model's ability to capture intricate relationships within the data. By increasing the number of neurons in each LSTM layer (the rnn_hidden_size), the model gains a larger capacity to store and process temporal information at each timestep. This increased capacity enables the model to learn more nuanced representations of

sequences, improving its ability to make accurate predictions, especially for long and complex sequences such as genomic data.

Lastly, dropout was added to the model to improve generalization and prevent overfitting. Dropout is a regularization technique where, during training, a certain percentage of neurons are randomly "dropped" (set to zero) in each forward pass, forcing the model to rely on multiple different paths during training. This technique helps prevent the model from memorizing the training data and encourages it to generalize better to unseen data. By adding dropout with a 50% probability (dropout_prob=0.5), the model becomes more robust, especially as it grows in complexity with more layers and nodes. This design choice is crucial as it reduces the risk of overfitting while still allowing the model to learn from the increased capacity provided by the additional LSTM layers and larger hidden size. Together, these modifications make the model more expressive, flexible, and resistant to overfitting, allowing it to tackle more complex sequence prediction tasks effectively.

seq_len = 200

Kept the same as before, as this length is sufficiently long for lots of patterns to be captured. Any longer than this and the sequences and data loaders were overwhelming the memory.

learning rate = 0.01

Increased from before for more efficiency and speed of training.

epochs = 1000

Decreased from before for more efficiency and speed of training.

```python
class RNN3(nn.Module):
    def __init__(self, vocab_size, embed_dim, rnn_hidden_size, num_layers=3, dropout_prob=0.5):
        super().__init__()

        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embed_dim)

        # Increase the number of hidden units (nodes) and LSTM layers
        self.rnn_hidden_size = rnn_hidden_size
        self.num_layers = num_layers
        self.dropout_prob = dropout_prob

        # LSTM with multiple layers and dropout
        self.rnn = nn.LSTM(embed_dim, rnn_hidden_size, num_layers=num_layers,
                           batch_first=True, dropout=dropout_prob)

        # Fully connected output layer
        self.fc = nn.Linear(rnn_hidden_size, vocab_size)

    def forward(self, x, hidden, cell):
        # Pass the input through the embedding layer
        out = self.embedding(x).unsqueeze(1)

        # Pass through the LSTM layer(s)
        out, (hidden, cell) = self.rnn(out, (hidden, cell))

        # Output through the fully connected layer
        out = self.fc(out).reshape(out.size(0), -1)

        return out, hidden, cell

    def init_hidden(self, batch_size):
        # Initialize hidden state and cell state for multiple layers
        hidden = torch.zeros(self.num_layers, batch_size, self.rnn_hidden_size)
        cell = torch.zeros(self.num_layers, batch_size, self.rnn_hidden_size)
        return hidden.to(device), cell.to(device)
```

```python
# Example for creating the updated model
vocab_size = 256
embed_dim = 512
rnn_hidden_size = 1024
num_layers = 3
dropout_prob = 0.5
batch_size = 256
model3 = RNN3(vocab_size, embed_dim, rnn_hidden_size, num_layers, dropout_prob).to(device)
```

Figure 1. The updated RNN3 class

III - Results:

Model 1:

For model 1 training, the loss came down rapidly with more epochs and was still falling during the later stages of training. Although an accuracy of 66% was achieved on the test dataset, I believe with more training the model would have done better, since it doesn't seem like it converged to the minimum loss.

```
Epoch 0 loss: 1.3193
Epoch 250 loss: 1.2995
Epoch 500 loss: 1.2878
Epoch 750 loss: 1.2694
Epoch 1000 loss: 1.2570
Epoch 1250 loss: 1.2156
Epoch 1500 loss: 1.1763
Epoch 1750 loss: 1.1451
Epoch 2000 loss: 1.1174
Epoch 2250 loss: 1.0671
Epoch 2500 loss: 1.0462
Epoch 2750 loss: 1.0268
Epoch 3000 loss: 1.0189
Epoch 3250 loss: 0.9842
Epoch 3500 loss: 0.9580
Epoch 3750 loss: 0.9419
Epoch 4000 loss: 0.9628
Epoch 4250 loss: 0.9152
Epoch 4500 loss: 0.9271
Epoch 4750 loss: 0.9148
Epoch 5000 loss: 0.8980
Epoch 5250 loss: 0.9017
Epoch 5500 loss: 0.9030
Epoch 5750 loss: 0.8875
Epoch 6000 loss: 0.8804
Epoch 6250 loss: 0.8740
Epoch 6500 loss: 0.8723
Epoch 6750 loss: 0.8688
Epoch 7000 loss: 0.8564
Epoch 7250 loss: 0.8437
Epoch 7500 loss: 0.8514
Epoch 7750 loss: 0.8506
Epoch 8000 loss: 0.8414
Epoch 8250 loss: 0.8363
Epoch 8500 loss: 0.8410
Epoch 8750 loss: 0.8335
Epoch 9000 loss: 0.8256
Epoch 9250 loss: 0.8208
Epoch 9500 loss: 0.8289
Epoch 9750 loss: 0.8062
```

```
Accuracy on test dataset: 66.1203%
```

Figure 2. Training and accuracy results for part 1.

Model 2:

For model 2 training, the loss came down rapidly with more epochs and converged, giving 79% accuracy on the test dataset.

```
...  Epoch 0 loss: 5.5444
     Epoch 250 loss: 1.3419
     Epoch 500 loss: 1.3314
     Epoch 750 loss: 1.3159
     Epoch 1000 loss: 1.2846
     Epoch 1250 loss: 1.2471
     Epoch 1500 loss: 1.2003
     Epoch 1750 loss: 1.1577
     Epoch 2000 loss: 1.0898
     Epoch 2250 loss: 1.0315
     Epoch 2500 loss: 1.0074
     Epoch 2750 loss: 0.9741
     Epoch 3000 loss: 0.9321
     Epoch 3250 loss: 0.9226
     Epoch 3500 loss: 0.8866
     Epoch 3750 loss: 0.8651
     Epoch 4000 loss: 0.8405
     Epoch 4250 loss: 0.8317
     Epoch 4500 loss: 0.8170
     Epoch 4750 loss: 0.8111
     Epoch 5000 loss: 0.8030
     Epoch 5250 loss: 0.7871
     Epoch 5500 loss: 0.7752
     Epoch 5750 loss: 0.7528
     Epoch 6000 loss: 0.7455
     Epoch 6250 loss: 0.7506
     Epoch 6500 loss: 0.7262
     Epoch 6750 loss: 0.7308
     Epoch 7000 loss: 0.7125
     Epoch 7250 loss: 0.7038
     Epoch 7500 loss: 0.7045
     Epoch 7750 loss: 0.6984
     Epoch 8000 loss: 0.7041
     Epoch 8250 loss: 0.6905
     Epoch 8500 loss: 0.6816
     Epoch 8750 loss: 0.6600
     Epoch 9000 loss: 0.6658
     Epoch 9250 loss: 0.6632
     Epoch 9500 loss: 0.6490
     Epoch 9750 loss: 0.6466
     Epoch 10000 loss: 0.6446
     Epoch 10250 loss: 0.6393
     Epoch 10500 loss: 0.6458
     Epoch 10750 loss: 0.6391
     Epoch 11000 loss: 0.6369
     Epoch 11250 loss: 0.6328
     Epoch 11500 loss: 0.6195
     Epoch 11750 loss: 0.6194
     Epoch 12000 loss: 0.6151
     Epoch 12250 loss: 0.6006
     Epoch 12500 loss: 0.5995
     Epoch 12750 loss: 0.6007
     Epoch 13000 loss: 0.6069
     Epoch 13250 loss: 0.5993
     Epoch 13500 loss: 0.5968
     Epoch 13750 loss: 0.6017
     Epoch 14000 loss: 0.5816
     Epoch 14250 loss: 0.5924
     Epoch 14500 loss: 0.5828
     Epoch 14750 loss: 0.5820
```

```
Accuracy on test dataset: 77.9619%
```

Figure 3. Training and accuracy results for part 2

Model 3:

```
Epoch 0 loss: 5.4933
Epoch 25 loss: 5.4845
Epoch 50 loss: 5.4775
Epoch 75 loss: 5.4796
Epoch 100 loss: 5.4080
Epoch 125 loss: 5.3135
Epoch 150 loss: 5.2500
Epoch 175 loss: 5.1947
Epoch 200 loss: 5.1475
Epoch 225 loss: 5.1234
Epoch 250 loss: 5.0853
Epoch 275 loss: 5.0721
Epoch 300 loss: 5.0172
Epoch 325 loss: 4.8987
Epoch 350 loss: 4.7240
Epoch 375 loss: 4.5308
Epoch 400 loss: 4.2973
Epoch 425 loss: 4.0520
Epoch 450 loss: 3.7897
Epoch 475 loss: 3.5359
Epoch 500 loss: 3.2895
Epoch 525 loss: 3.0240
Epoch 550 loss: 2.7666
Epoch 575 loss: 2.5113
Epoch 600 loss: 2.3008
Epoch 625 loss: 2.0902
Epoch 650 loss: 1.9571
Epoch 675 loss: 1.8579
Epoch 700 loss: 1.7810
Epoch 725 loss: 1.7253
Epoch 750 loss: 1.6851
Epoch 775 loss: 1.6439
Epoch 800 loss: 1.6202
Epoch 825 loss: 1.5935
Epoch 850 loss: 1.5700
Epoch 875 loss: 1.5587
Epoch 900 loss: 1.5376
Epoch 925 loss: 1.5261
Epoch 950 loss: 1.5225
Epoch 975 loss: 1.5077
```

```
Accuracy on test dataset: 32.8037%
```

Figure 4. Training and accuracy results from part 3.

IV - Discussion and Conclusion:

In this project, I applied recurrent neural networks (RNNs) with Long Short-Term Memory (LSTM) units to predict nucleotide sequences in genomic data. I started with a basic LSTM model for character-level prediction, then advanced to a model using k-mer sequences to capture higher-level DNA patterns. The final model, Model 3, incorporated multiple LSTM layers, increased hidden size, and dropout regularization to improve the model's ability to handle complex dependencies, generalize better, and prevent overfitting.

Model 1, which used single character prediction, achieved 66% accuracy but didn't fully converge, indicating that further training or hyperparameter adjustments could improve performance. **Model 2**, based on k-mers, performed better, achieving 79% accuracy on the test dataset, showing that k-mers allow the model to capture more complex patterns. However, both models still have room for improvement, especially in terms of further fine-tuning.

Model 3 introduced a more complex architecture with additional LSTM layers and regularization, which allowed for better generalization and deeper pattern recognition. While the results show promising improvements, further experimentation with hyperparameters and training duration will provide more insight. Overall, RNNs, particularly with LSTM and attention mechanisms, show significant potential for bioinformatics applications such as mutation detection and genome assembly.