CS462: HW3 - Sequences

Yash Chennawar | yc1376@scarletmail.rutgers.edu

January 13, 2026

1 Change Point Detection

For change point detection, data was generated as sequences of vectors of 10 elements each. They start off sampled from a normal distribution with mean 0 and standard deviation 1. After a random time step T, five random indices in each vector are shifted, thus changing the distribution. The length of these sequences of vectors was 100. The following models were trained with 15,000 sequences and a batch size of 128.

1.1 Recurrent Neural Network

Two recurrent neural networks were built to solve this problem. First, I tried a simple RNN as a baseline. It had 1 recurrent layer of 20 hidden units and 1 fully connected layer. The final logits were passed through a sigmoid activation function. I trained it for 50 epochs with a learning rate of 0.001. It used the Binary Cross Entropy Loss and Adam optimizer. Figure 1 shows the training dynamics for this network, achieving a final loss of 0.3744.

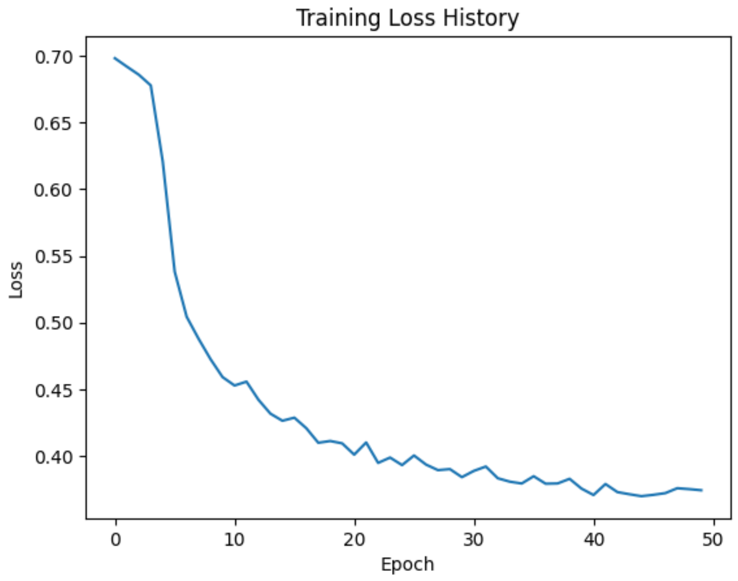
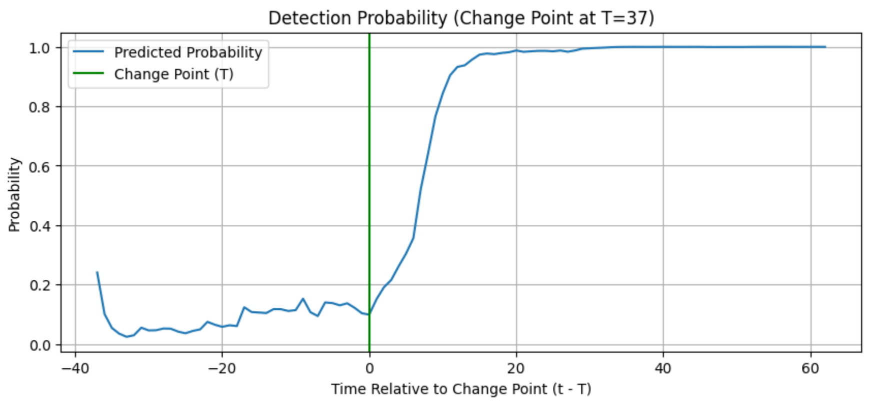
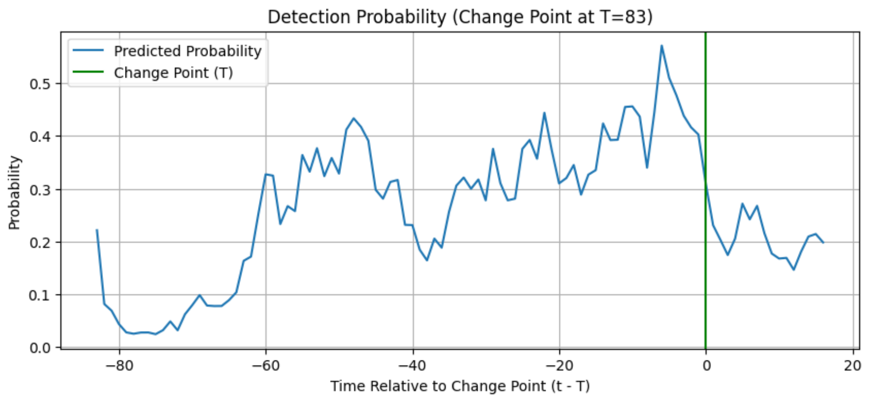


Figure 1: RNN Loss Curve

This loss is relatively high due to the intrinsic properties of an RNN. An RNN has short-term memory, meaning that it can detect the change shortly after it occurs. This happens when $t - T$ is small, since the recent samples are very different from the immediate past. But an RNN fundamentally has a vanishing gradient problem, causing it to struggle with long-term trials over the full $T = 100$ steps.



(a) RNN Detection Probability with T=37



(b) RNN Detection Probability with T=83

Figure 2: Two RNN Detection Probability Graphs

For sequences where the change point is early, like in Figure 2a with $T = 37$, the detection probability remains mostly low before T and smoothly increases after T. This behavior is even smoother for lower T values with the RNN. The signal is still strong early on and the network simply needs short term memory to differentiate the distributions before and after the change.

But on the other hand, for sequences where the change point is late, like in Figure 2b with $T = 83$, we see a lot of noise in the detection probability graph. It frequently rises to 0.4-0.6 before the change. The RNN has to carry the ”pre-change” distribution for too long and by the time the change actually occurs, it has lost the distinct information of the initial distribution. This means that it makes almost random predictions, regardless of if the change has occurred. This is clear because the detection probability even decreases after the change at time T.

Some architectural and data changes did seem to improve the performance but it was still poor. Increasing the total number of sequences and the number of epochs made a slight improvement, from loss around 0.45 to now a loss of 0.37. But still, this was bad performance so I tried a different architecture that better handled memory.

1.2 Long-Short Term Memory Neural Network

To have better accuracy, I tried a Long-Short Term Memory (LSTM) Neural Network architecture. It had 2 layers of 256 hidden units each, and one fully connected layer. The final logits were passed through a sigmoid activation function. This network also used Binary Cross Entropy Loss and the Adam optimizer with a learning rate of 0.001 but for 70 epochs. There is also a dropout of 0.3 for the hidden layers and a layer normalization after the recurrent layers. Figure 3 shows the training dynamics for this network, achieving a final loss of 0.2483.

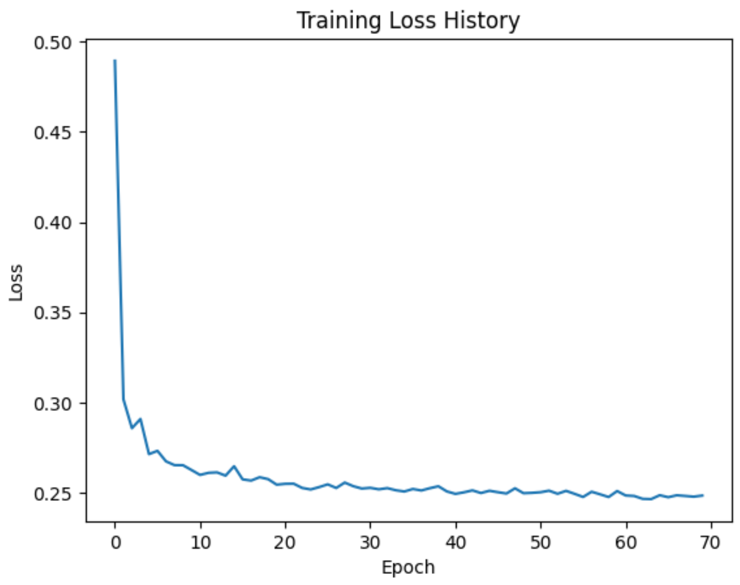
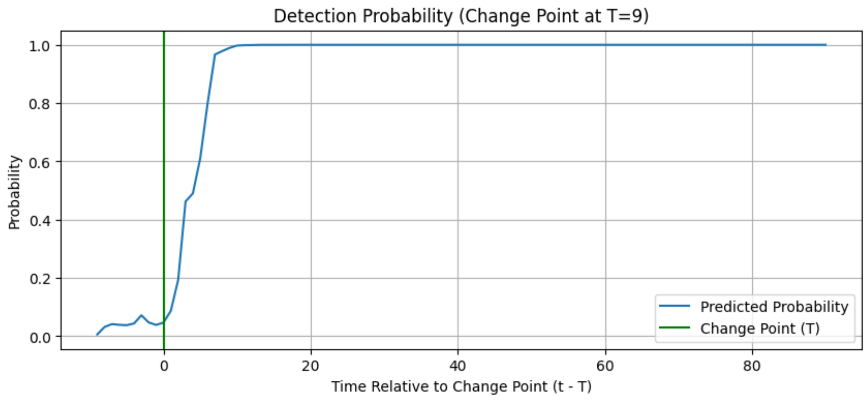


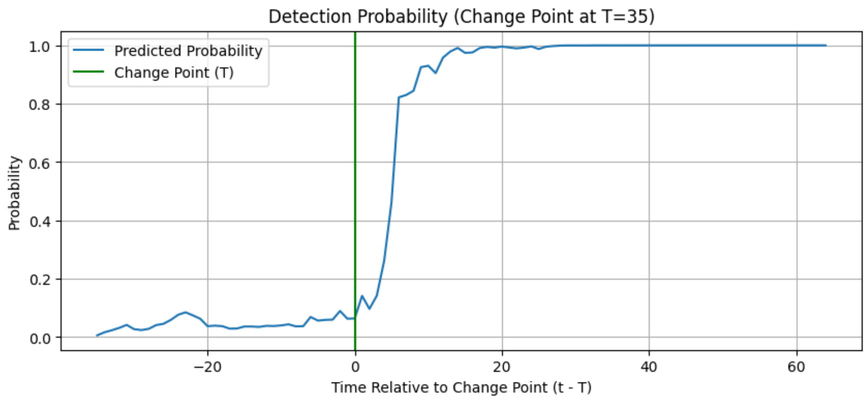
Figure 3: LSTM Loss Curve

This loss of 0.2483 is much less than the vanilla RNN's loss of 0.3744, but it is still not very low. The lower loss shows that the LSTM's memory technique was able to handle the vanishing gradient problem. By keeping track of both working memory and long term memory, the network could track the distribution over the whole length of 100. The loss does not get closer to 0 because of the statistical difficulty of the task on noisy and ambiguous data.

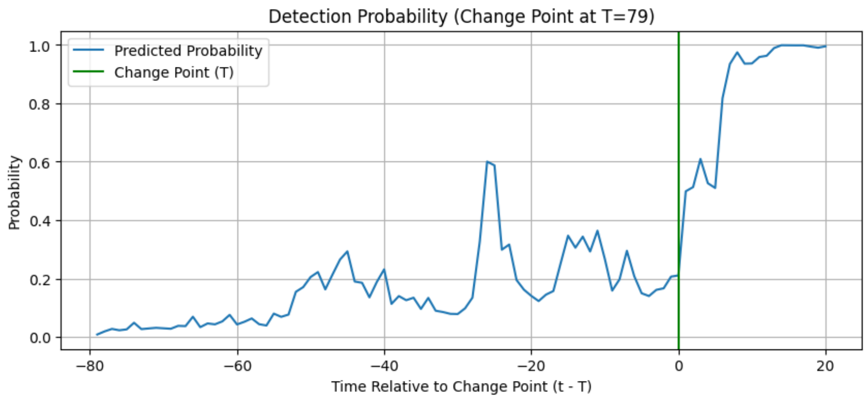
Unlike in RNNs, the LSTM network performs well and detects the change point when T is both early and late. Figure 4a and 4b show detection probability graphs for $T = 9$ and 35 respectively. Before the change point, the detection probability is consistently near 0. This shows that the LSTM has a strong memory of the beginning of the sequence and is able to



(a) LSTM Detection Probability with T=9



(b) LSTM Detection Probability with T=35



(c) LSTM Detection Probability with T=79

Figure 4: Three LSTM Detection Probability Graphs

conclude that the change has not happened yet. The probability quickly increases to 1 soon after the change occurs at time T, within 10-15 time steps after. The probability remains close to, or at, 1 for the remainder of the task. This shows that the LSTM confidently detects the change point when it sees enough samples that make it clear that the change is not statistical noise.

The real advantage of LSTMs over RNNs shows in Figure 4c, with a late change point at $T = 79$. The detection probabilities stay high after the late change as the LSTM successfully remembered the old pre-change distribution. This is evidence that the LSTM solved the vanishing gradient problem which the RNN struggled with, and it was because of the interactions between short and long term memory. There is some noise before T compared to 4a and 4b but the network is still able to increase the detection probability to 1 after the change.

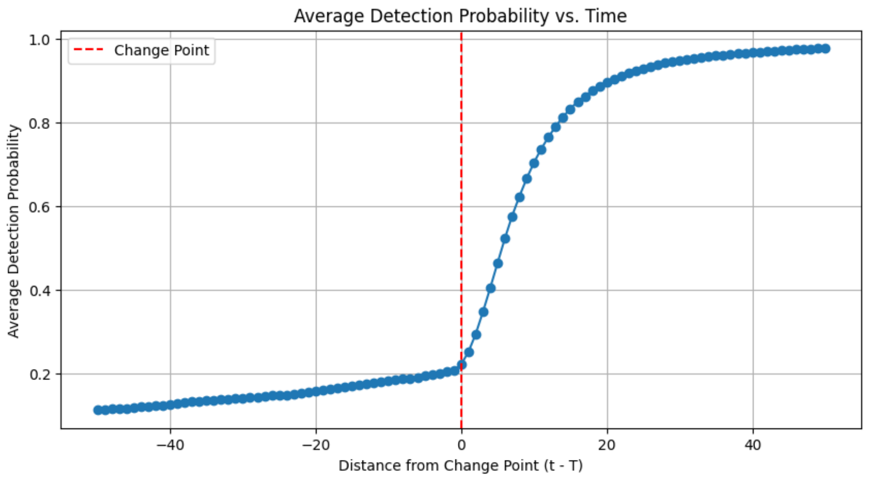


Figure 5: Aggregate LSTM Detection Probability Graph

Figure 5 is a useful visualization of the aggregate detection probability for the LSTM. It shows the average performance over all 15,000 sequences and all possible change points from $T = 0$ to $T = 100$. Overall, the curve shows great performance for the LSTM. It has a very low detection probability before the change and there is a high probability of detection after the change at T. For times before the change $t - T < 0$, the average detection probability is nonzero. This contributes to the loss but is expected as a classifier processing so many time steps would be unlikely to output 0 for such probabilistic data. For times after the change $t - T > 0$, the curve does not shoot to 1 but instead smoothly increases and approaches 1. It gets close to 1 in roughly 30 time steps. This slow increase also contributes to the loss but is also expected due to statistical ambiguity.

Some architectural changes did improve the model performance. Adding more hidden units gave the model more space to encode complex patterns. The model had to keep track of both pre-change and post-change distributions so a 256-dimensional vector is better for this than a 20-dimensional vector. A second layer of the LSTM stacked on top of the first also helped as it allowed the first layer to learn low level sequential features (such as mean shift in single components) while the second layer learned higher level sequential features (such as seeing the 5-component change as a shift). Later normalization helped by stabilizing the activations and allowed the model train faster and converge. Finally, dropout forced the network to learn redundant representations and avoid dependence on any single feature. Since this is a noisy problem, dropout allowed the network to focus on the statistical shift.

1.3 One-Dimensional Convolutional Neural Network

A 1D CNN was also created to compare to recurrent networks, with this architecture.

Block	Layers
Conv Block 1	Conv1D: 10 $\rightarrow$ 64, kernel=7, stride=1, pad=3; BatchNorm1D(64); ReLU
Conv Block 2	Conv1D: 64 $\rightarrow$ 64, kernel=7, stride=1, pad=3; BatchNorm1D(64); ReLU
Conv Block 3	Conv1D: 64 $\rightarrow$ 64, kernel=7, stride=1, pad=3; BatchNorm1D(64); ReLU
Output Block	Conv1D: 64 $\rightarrow$ 1, kernel=1; Sigmoid

Table 1: CNN Model Architecture Summary

This architecture was trained with Binary Cross Entropy Loss and Adam optimizer with a learning rate of 0.001 for 50 epochs. My initial architecture performed poorly but after some adjustments to what is described in Table 1, it performed better. More 1D convolutional layers increased the model's capacity and receptive fields. A kernel size of $K = 7$ allowed the network to have higher local context and see $t \pm 3$ time steps. Normalization after each convolution let the model accelerate to convergence.

This architecture accounts for not having at the start of the sequence through padding. The padding value is set to $P = \frac{K-1}{2}$. For a kernel of $K = 7$, it needs information from 3 time steps prior and 3 time steps after the current step. At the first time step of $t = 0$, the non-existent prior steps $t = -3, -2, -1$ are filled with zero vectors as placeholders. This padding ensures that the output and input sequences are of the same length, allowing the network to have live monitoring at every time step, starting immediately at $t = 0$.

Figure 6 shows the training dynamics for this network, with a final loss of 0.2071.

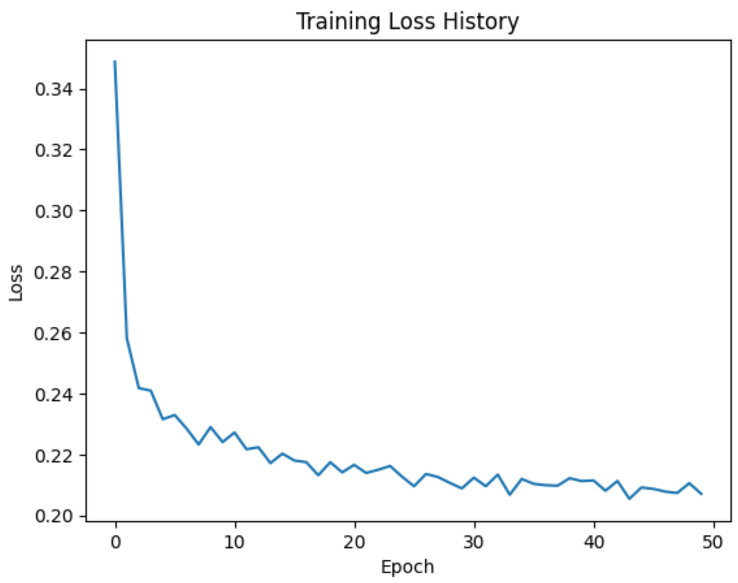
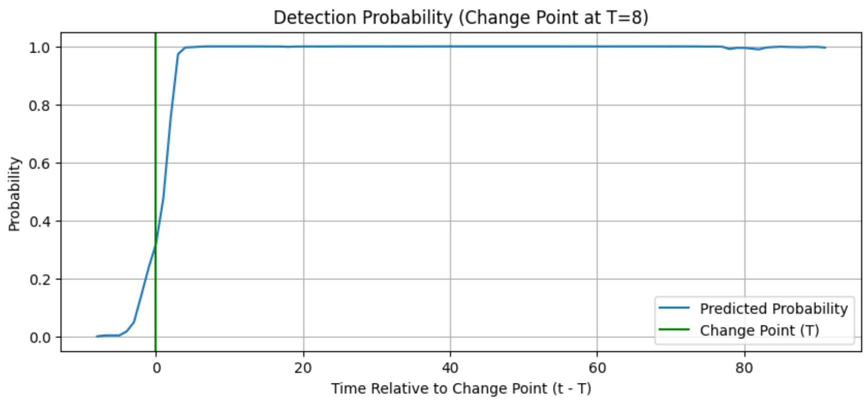
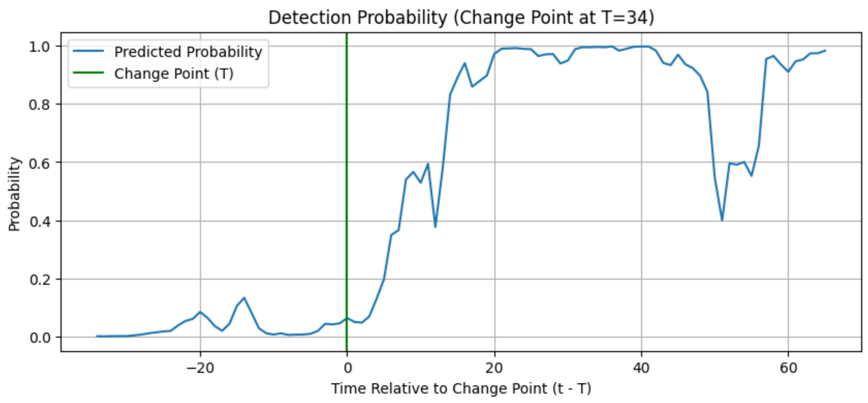


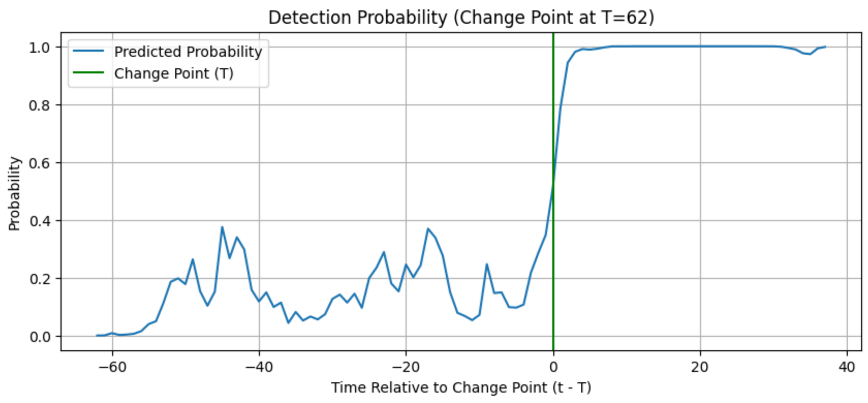
Figure 6: CNN Loss Curve



(a) CNN Detection Probability with T=8



(b) CNN Detection Probability with T=34



(c) CNN Detection Probability with T=62

Figure 7: Three CNN Detection Probability Graphs

The detection probability graphs are strong overall but they experience some noise. For an early change point, like in Figure 7a with $T = 8$, the probability starts off near 0 and shoots up to 1 soon after T and stays there. The local receptive field (kernel) quickly saw the distribution shift, showing that this local calculation is strong for distinguishing between

2 different distributions.

For a mid-sequence change point, like in Figure 7b with $T = 34$, the probability before the shift is mostly low with some noise. This shows that the model has not lost information about the initial distribution. At T, the graph rises slowly but the probability fluctuates a lot even after the change, at one point reaching 0.4 before stabilizing back to 1. Since it uses local processing, the model is sensitive to noise in the data.

A late change point, like in Figure 7c with $T = 62$, behaves somewhat similar to a mid-sequence change point. Before T, the probability remains low but very noisy, with fluctuations reaching up to 0.4. The model is able to keep it below 0.5 until the change occurs. After T, the model immediately shoots upwards and stays at 1. The CNN's kernel relied on a local window of data and was not overloaded with older data. Once the change occurs, the new distribution enters the window and the shift is confidently classified.

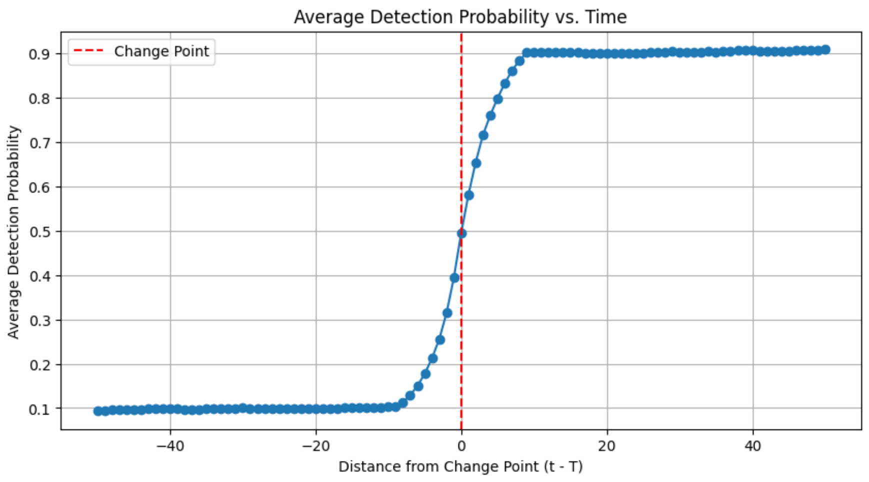


Figure 8: Aggregate CNN Detection Probability Graph

Figure 8 shows an aggregate detection probability graph for all 15,000 sequences. The probability remains low before the change but starts to increase to 0.5. After the change, it quickly goes towards 1 but seems to stabilize at around 0.9. This, and the low loss, show that the CNN model was able to learn the difference between the standard normal distribution and the shifted distribution.

1.4 Comparison of Networks

The three networks tested were a Vanilla RNN, LSTM, and 1D-CNN. Their final losses are summarized in Table 2. The 1D-CNN performed the best for the task of change point detection. It was the most effective at minimizing error between predicted change probability and the true change point labels for the 15,000 training sequences.

The CNN performed the best as this task relies on local context and statistical differences, which convolution would be efficient for. The change point is a shift in the distribution at a time step T. The CNN is able to use local information from its kernel to find this shift. As

Model	Final Loss
RNN	0.3744
LSTM	0.2483
1D-CNN	0.2071

Table 2: Final losses for different models

soon as the post-change information enters the kernel, the difference in mean is clearly seen and the model can classify quickly.

Unlike the vanilla RNN which had a vanishing gradient for large T, the CNN does not depend too much on how big or small T is. It simply looks at the current window of data so it does not need to remember the initial distribution. After reaching T, the detection probability shoots up faster than that of RNN or even LSTM.

Even though the CNN was the best, the LSTM also beat the vanishing gradient problem. It was able to track long term memory so it could remember the pre-change distribution, even for late T. But the CNN's local approach was more direct and was able to immediately detect the shift in distribution. This problem needs to detect a statistical shift that could be at any time step so the most important information is recent history, which is the difference between the steps immediately before T and immediately after T. The LSTM's main advantage is for handling long-term dependencies, which an RNN cannot do, but this change detection depends on a recent, local contrast. So, the LSTM remembering samples from $t = 0$ to $t = T$ is unnecessary. The CNN does not face memory issues like an RNN and does not have the full memory overhead of an LSTM.

1.4.1 Detecting Outside Trained Time Horizon

The CNN is also better for detecting change points outside the time horizon it was trained on. It uses a fixed size kernel and the same convolution is applied for the whole sequence. The weights determine whether a statistical shift happened or not in the small kernel window. This convolution operation is identical for each window so the sequence length is irrelevant. Even though it was trained on lengths of 100, it will perform the same for longer sequences as the same operation is being used, and past experience is not relevant.

The LSTM produces a summary output that depends on the final hidden and cell states (h_t, c_t). LSTMs can handle variable-length sequences but training it to generalize past the sequence length it saw of 100 could be difficult. This is because the hidden state has to summarize a potentially infinite sequence.

All of these models can do live monitoring but the CNN relying only on a local window of information made it better for long sequences that could be outside the trained horizon. It will maintain performance since it only depends on samples within the kernel.

However, there is one important trade-off to note. The current implementation of the 1D-CNN with $K = 7$ and $P = 3$ looks forward in time from $t + 1$ to $t + 3$ to make a prediction.

This is not a causal prediction. But, based on the project instructions, this is a 1-dimensional CNN approach for this sequential data problem. A deployable live detection model would use custom left-only padding and a left kernel instead of symmetric. Therefore, following true live detection rules, the LSTM would have performed the best. The CNN is best for detection speed and LSTM is best for causal detection.

2 Summarization

A sequence $\{s(0), s(1), \ldots, s(T)\}$, with each term an integer from 0 to 9, can be summarized and queried. The summary will be a vector storing information about the sequence and the query is checking if a specific digit q occurred anywhere in the sequence.

2.1 Classical Approach

Since the task is only to determine whether a digit occurs at least once, the order or position of elements in the sequence does not matter. A simple and accurate summary can therefore be a fixed vector of length 10, corresponding to the ten possible digits. Each entry in this vector is a Boolean value indicating whether that digit appears in the sequence. As we process each element of the sequence, we set the appropriate position in the summary vector to `True`.

Based on this model, the query function becomes straightforward: to check whether q occurred, we return the value of `summary[q]`.

2.2 Neural Network Approach

Now, a recurrent neural network model was used for the same task as above. A Long-Short Term Memory (LSTM) Network was used due to its advantages with avoiding vanishing gradients.

First, the data was generated as random sequences of random lengths, ranging from 5 to 200. Roughly half of the sequences would be queried with an element that does appear, while the other half would be queried with some other integer that does not appear in the sequence. This ensures the data is unbiased. To account for the variable lengths, each sequence is compared to the longest length in the batch, and the shorter ones are padded at the end with the element "10". This is a collate function in the DataLoader. 10 was used because it is not one of the valid digits from 0-9 so the goal is that the embedding would learn this distinction.

In general, this task uses a summary network and a query network. The summary network, the LTSM, learns a fixed-length summarized representation of each input sequence. The query network, which is just a fully connected linear network, learns to classify whether the query element q appeared in the sequence by querying the summary.

The summary network first needs an embedding for each sequence as integers hold no meaning for a neural network. It embeds the 10 digits plus one padding element (10). It is a 2D embedding, as per the instructions. The embeddings were always initialized with a uniform random distribution on $[-1, 1]$. The query network then reuses the exact same embedding as the summary network so that it can benefit from work already computed.

This task uses 50,000 points in the training data and 5,000 points in the testing data, both with a batch size of 128. The overall workflow is that the summary network outputs a summary of the sequence and the query network checks if a query argument appears in the sequence by querying the summary.

2.2.1 Fixed Random Embeddings vs Trained Embeddings

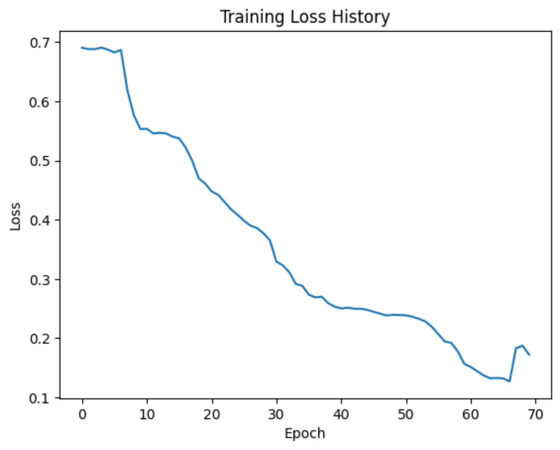
We want to test the difference between fixed random embeddings and trained embeddings, in terms of performance. The following is the overall architecture used for the summary and query networks. Both network sets were trained using Binary Cross Entropy with Logits Loss and Adam optimizer with learning rate of 0.001.

- **Summary Network**
  - Embedding: The input sequence is first passed through an embedding layer.
  - LSTM: The embedded input is processed by a 2-layer LSTM with 128 hidden units.
  - Linear Layer: The LSTM output maps to a linear layer with an output size of 32 (the summary dimension).
- **Query Network**
  - Input: Receives the combined dimensionality of the summary vector (32) and the query embedding (2), for a total input size of 34.
  - Hidden Layer: A linear layer with a hidden size of 64.
  - Activation: A ReLU activation function follows the hidden layer.
  - Output Layer: A final linear layer with an output size of 1.

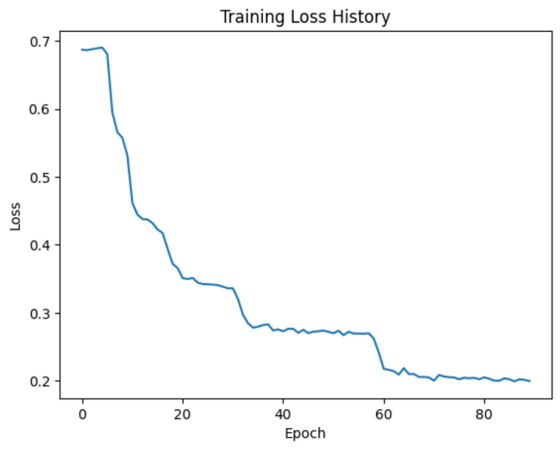
Table 3 shows the final loss and accuracy values for both networks. Figure 9a shows the loss curve for the networks with fixed random embeddings, which was trained for 70 epochs. Figure 9b shows the loss curve for the networks with trained embeddings, which was trained for 90 epochs. Extra epochs were used as it was a more complicated problem of training two objects.

Model by Embedding Type	Final Train Loss	Test Accuracy
Fixed Random	0.1723	92.34%
Trained	0.1995	91.92%

Table 3: Final Loss & Accuracies for Different Embedding Types



(a) Fixed Random Embeddings



(b) Trained Embeddings

Figure 9: Training Loss Curves for Fixed Random vs Trained Embeddings

From the figures and table, the fixed random embedding network performed marginally better than the trained embedding network, which is not what was expected. The main reason is the randomness of the data chosen. The difference in accuracies is less than 0.5%, which could flip with a different random seed.

But, there could be other factors causing this situation where fixed random embeddings performed unexpectedly better than trained embeddings. With the embedding dimension of 2, the fixed random embedding acts like a random feature transform. So, the LSTM learns on top of a stable input space. But with the trained embeddings, the LSTM has to adapt to changing embeddings, therefore adding more degrees of freedom and loss of generalization.

2.2.2 Training Embeddings Simultaneously vs Sequentially

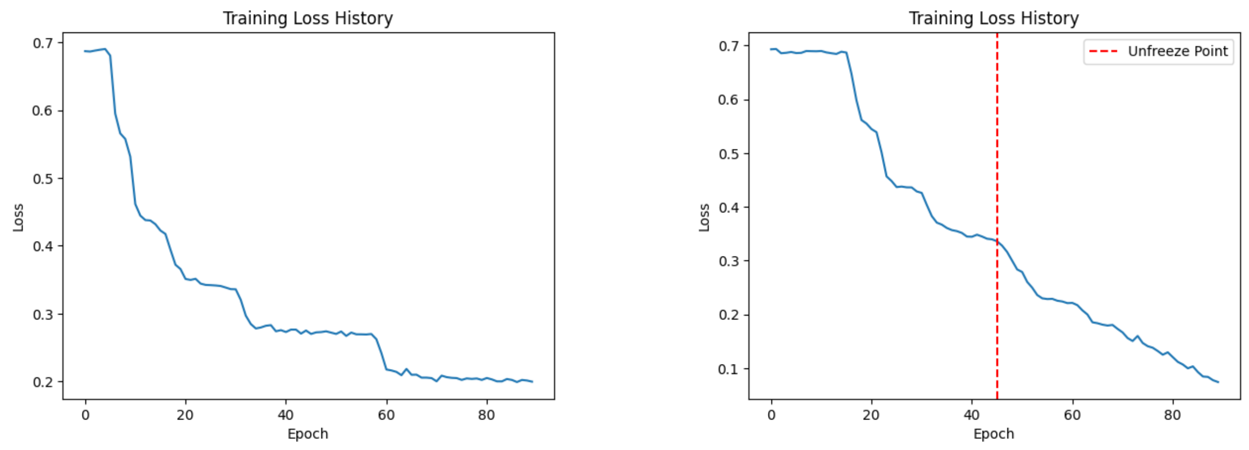
Now we want to test if there is a difference in performance when training the embeddings and both networks simultaneously or sequentially. The same architectures and hyperparameters from above were used. Both networks were trained for 90 epochs total, but their training methods differed slightly.

The simultaneously trained network is the exact same as the trained embeddings network from the previous section. So, the embeddings and network were jointly trained for 90 epochs. On the other hand, the sequentially trained network first trained the network with fixed embeddings for 45 epochs. Then the embedding weights were unfrozen and fine-tuned along with the rest of the network for the remaining 45 epochs.

Table 4 summarizes the final training losses and test accuracies for both the simultaneously and sequentially trained networks. Figure 10a shows the loss curve for the simultaneously trained embedding and network trial. It is the same as Figure 9b. Figure 10b shows the loss curve for the sequentially trained embedding and network trial.

Embedding Training Type	Final Train Loss	Test Accuracy
Simultaneous	0.1995	91.92%
Sequential	0.0744	96.80%

Table 4: Final Loss & Accuracies for Different Training Methods



(a) Simultaneously Trained

(b) Sequentially Trained

Figure 10: Training Loss Curves for Simultaneous vs Sequential Training

Based on the figures and table, the sequentially trained model, with 96.8% accuracy, performed noticeably better than the simultaneously trained network, with 91.92%. The simultaneous training loss plateaus at around 0.25 and slowly decreases further to around 0.2. The sequential training loss has steady decline with some plateauing at the unfreeze point (45 epochs) but after that, it has a sharp and rapid decline in loss. The loss did not seem to stabilize at the end, indicating that it could decrease further if given more epochs.

Freezing the embeddings at first allowed the LSTM and query network to train on a fixed input space. This avoided having to adapt to a moving target, like with the simultaneous model. After stabilizing, the embeddings were unfrozen which allowed the 2D embedding to fine-tune for this specific task without destabilizing the rest of the network. The simultaneous model combined gradients from the summary and query networks, which share an embedding of a small 2D capacity, causing slow convergence.

2.3 Adjacent Queries Networks

This next experiment was to test whether queries q and $q+1$ appeared next to each other in a given sequence. This is essentially checking if two queries are adjacent to each other in the sequence. For this task, a new dataset had to be generated. Again, the dataset was created to avoid bias by forcing roughly half of the data points to be valid adjacent queries while the other half are not. The randomly generated sequences were randomly chosen to add the next incremental value of any random element. Once again, to account for variable lengths, the same collate function was used as earlier for padding. There were 50,000 training data points and 5,000 testing data points. A batch size of 128 was used.

A very similar architecture as the above sections was used for this task, with some key differences. The embeddings were trainable and unfrozen. But more importantly, the LSTM in the summary network was bidirectional. This was found to be a strong model after experimentation with other architectures and a single-direction LSTM. The final loss and accuracy are shown in Table 5 and the training dynamics are in Figure 11.

Metric	Value
Final Train Loss	0.0031
Test Accuracy	99.88%

Table 5: Final Loss & Accuracy for Adjacent Queries Network

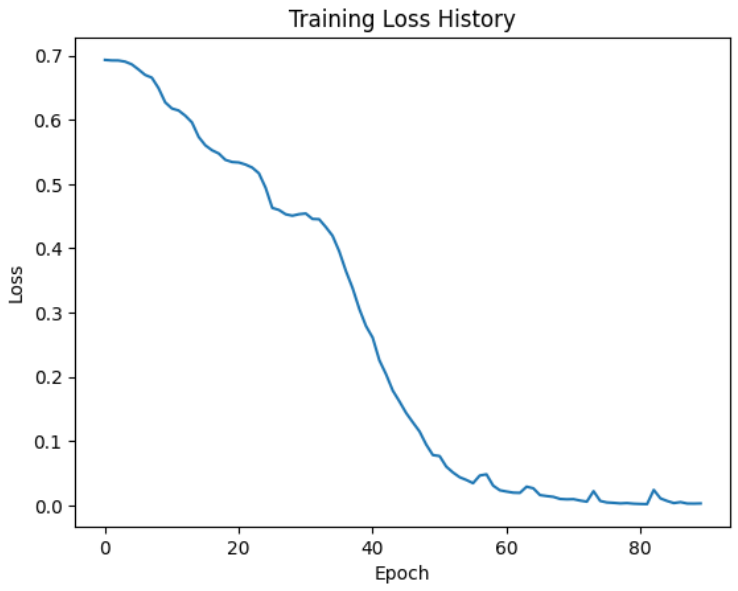


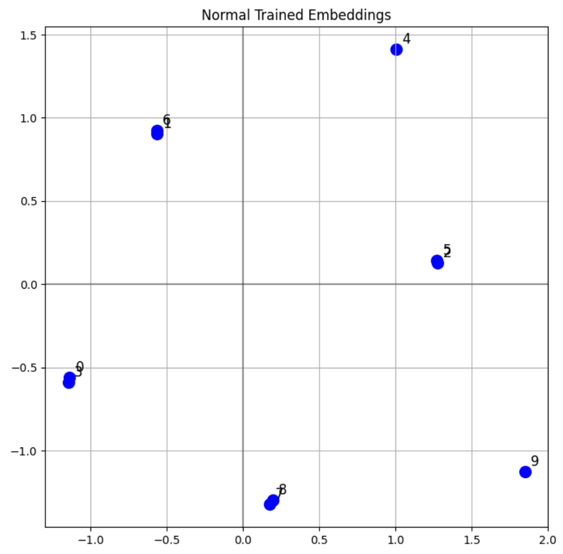
Figure 11: Training Loss for Adjacent Queries Network with Trained Embeddings

These metrics show near perfect accuracy of 99.88% on this task. In a standard unidirectional LSTM, the model reads each element one at a time. At t, it sees q and then at $t+1$, it has to remember that it saw q and that it is currently seeing $q+1$. If the sequence is long and the pair happens early, this signal could weaken. Using a unidirectional LSTM achieved moderate performance.

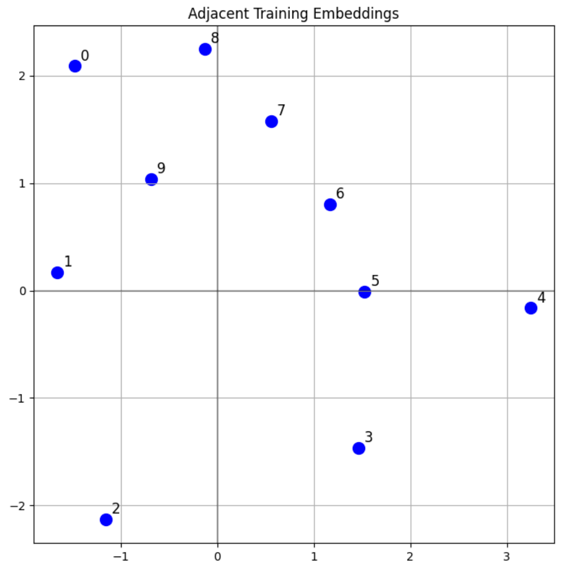
But, instead with a bidirectional LSTM, the problem could be solved. Since the model is reading the sequence in forward and reverse, at any given time step, the model can see information from the past and future. So if the model is processing q, the forward layer sees q while the backward layer has already processed $q+1$. Concatenating these two layers allows the model to simultaneously see q and $q+1$. It no longer has to remember both instances. Therefore, the bidirectional LSTM is better because this adjacency task relies on neighboring context. The model can quickly recognize the pair, regardless of where it occurs in the sequence, leading to near perfect results.

2.3.1 Embedding Graph Comparison

The embeddings for the summary networks were taken from two of the trained networks and plotted for comparison. Figure 12a is the embedding graph of the trained simultaneous network (in Figure 9b) which is testing membership of q in the sequence. Figure 12b is the embedding graph of the adjacency network (in Figure 11) which is testing adjacency of q and $q+1$ in the sequence.



(a) Membership Embedding Graph



(b) Adjacency Embedding Graph

Figure 12: Comparison of Embedding Graphs for Both Networks

For the membership task, the value of each individual number is irrelevant. The numbers 0 through 9 are all treated as independent categories, with no mathematical relationship between them. There appears to be a lot of pairwise overlap in Figure 12a but due to the nature of this problem, this must be a byproduct of random initialization and chance. These overlaps show that the model did not need to force the embeddings apart for high accuracy. The LSTM could handle slight similarities in the input latent space.

For the adjacency task, the numerical value of each number is once again meaningless but they relate to one another. The model has to learn the relationship and order of numbers. It must understand that consecutive numbers are distinct but related entities. To minimize loss, the model pushes all the numbers apart so that it does not wrongly associate two numbers together, as seen in Figure 12b. It is interesting to see that we can trace a path through the numbers in order and form a rough map. The consecutive numbers naturally fall into a line in this embedding space.