Final Year Project

Deep Learning for Symbol Detection

Aman Parary

Student ID: 18417714

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Dr Guénolé Silvestre

UCD School of Computer Science
University College Dublin

April 28, 2022

Table of Contents

Abstract

The word 'symbol' comes from a concept in digital communication systems. Symbols are multiple bits of data mapped together to remove redundancy so that we can effectively monitor transmitted symbols and observe them from the receiver to ensure that the communication system operates at the correct time and in the correct order. One of the most popular algorithms to recover symbols in a digital receiver is the iterative Viterbi. This algorithm requires the knowledge of the statistical input-output relationship between the digital transmitter and receiver (channel state information) to recover symbols with minimal errors. Nowadays, deep learning has proven to be a robust use case in digital communication systems and has experienced increasing interest over the recent years. The increasing interest is due to the neural network's sequential learning capabilities from past symbols and effectively predicting unseen symbols without knowing the channel state information. This project aims to integrate efficient deep learning technology into some parts of the Viterbi algorithm for symbol detection. The project undergoes the process of generating our dataset and then training and testing our detectors. The evaluation metric will specialise in the performance of the symbol error rate (the ratio of symbol errors to total symbols sent). The results indicate that the deep learning methods perform optimally with only small training symbols. Moreover, this method also performs optimally over noisy symbols, unlike the traditional Viterbi algorithm.

Chapter 1: Project Specification

1.1 Core

- Review literature on the Viterbi algorithm and neural network detection of symbols in communication systems.
- Generate dataset with a finite memory causal channel: An intersymbol interference (ISI) channel with additive white Gaussian noise (AWGN).
- Implement and evaluate the performance of the ViterbiNet detecting symbols in terms of the symbol error rate, compared to the traditional Viterbi algorithm.

1.2 Advanced

- Evaluate the performance using alternative models.
- Explore different noise channels and evaluate performance between Viterbi and ViterbiNet.

Chapter 2: **Introduction**

Communication is the exchange of information sources between two points. From an electronics point of view, devices and gadgets exchange information between two points far away. Using a block diagram, let us understand the fundamentals of information exchange via communication systems.

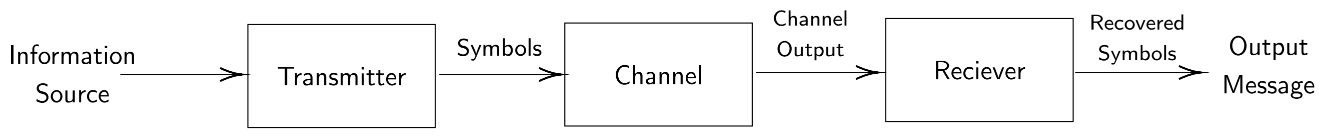

Figure 2.1: Block Diagram of Information Exchange in Communication Systems

The first block represents the information source. This information can be anything in the form of audio, image or discrete data. The source should generate electrical signals to pass down to the transmitter stage. However, if an information source does not generate electrical signals (e.g. audio), we must translate the information to generate such signals with the help of a transducer.

The transmitter stage undergoes three processes: source encoding, channel encoding, and modulation. Source encoding helps reduce the data redundancy generated by the electrical signal. In addition, this technique utilises the use of bandwidth effectively. The encoding also helps compress data using coding techniques such as Huffman or Shannon-Fano compression. Moreover, channel encoding provides noise immunity by adding redundancy data bits. In addition, adding redundancy helps handle errors in the channel stage (which contains noise), making it easier to recover such errors in the receiver stage of the block diagram. The modulation technique converts an electrical signal into high frequency modulated signal, known as symbols. This technique requires the product of two inputs: the encoded digital signal, and the carrier high-frequency signal. Finally, the modulation technique transmits symbols over the transmitter antenna.

The transmitted symbol from the antenna enters the channel stage: a medium over which the symbol is transmitted over a certain distance. This channel can be either physical (e.g., optical fibre) or wireless (e.g., radio link). The channel partly acts as a filter, which attenuates and distorts symbols. Consequently, the symbols get exposed to channel noise.

The main idea of the receiver stage is to recover symbols from the channel output effectively. This concept is known as symbol detection. The receiver stage performs the opposite process as the transmitter, i.e., demodulation, channel and source decoding. The demodulation converts the symbols back to data bits with additive redundancy data. As mentioned earlier, the redundancy data added in the channel encoding must be corrected. Therefore, we must apply error correction codes in the channel decoder to remove redundant data. Finally, the source decoder converts the data bits into an information source sent out as an output message.

Many detection algorithms require the channel's statistical input-output relationship, called the channel state information (CSI). It describes how a symbol propagates through the channel stage and represents the combined effect of elements (such as fading and power decay) over a particular time. The most popular detection method that requires CSI is the iterative Viterbi algorithm. This algorithm is efficient for recovering symbols with minimal errors. However, for this algorithm to achieve its full potential, it has to have a complete and concise CSI.

Deep learning models have become a trending topic in computer science and are widely applied in visual recognition, text analytics, cybersecurity, and many more. As a result, machine learning techniques, particularly deep learning, can help detect symbols over digital communication systems without using CSI. This project will focus on reproducing ViterbiNet [1]: A Viterbi algorithm for symbol detection based on deep learning integration, researched and implemented by the following authors: Nir Shlezinger, Yonina C. Eldar, Nariman Farsad, and Andrea J. Goldsmith. The implementation produced by Shlezinger *et al.* was completed using MATLAB with a deep learning toolbox. However, this project will reproduce the same approach with Python and PyTorch. Furthermore, we will modify the neural network architecture proposed by Shlezinger *et al.* and explore other topologies such as convolutional neural networks and attention layers and stacked layers of long-short term networks and gated recurrent units.

ViterbiNet is an extension of Viterbi, where it identifies specific parts of the algorithm that are channel-based models and applies a deep neural network to detect symbols. The benefit of applying a deep learning technique to the Viterbi algorithm is that it is independent due to its randomisation approach during learning; it can be effectively used when the channel model is unknown, or its parameters are difficult to measure. Moreover, compared to iterative model-based approaches, deep learning methods frequently result in faster convergence, even when the model is known, and can efficiently obtain and recover symbols from the observed data.

The work provided by Shlezinger *et al.* has supported that ViterbiNet is a simple network design that can learn using limited observations of training data. Moreover, according to their numerical evaluations, ViterbiNet delivers about the same performance as the ideal CSI-based Viterbi algorithm when the training data follows the same statistical model as the test data [1]. Furthermore, when ViterbiNet is trained with a range of channels, it significantly exceeds the Viterbi algorithm with the same level of CSI. Thus, proving to be an efficient and reliable neural network-based communication system.

This report will discuss various aspects of deep learning for symbol detection. For example, we will write a chapter where we research other people's work related to this project on integrating neural networks in digital communication systems. Moreover, we discuss the core implementation tasks, reproducing the work of Shlezinger *et al.*, which includes the implementation of a dataset, the Viterbi algorithm and the neural network-integrated Viterbi algorithm. Furthermore, we implement other types of neural networks as part of our advanced implementation. Finally, we will record our results, overall experiments, and performance in their respective chapters, in which we conclude our project implementation.

Chapter 3: **Related Work**

The work provided by Shlezinger *et al.* [1] has supported that ViterbiNet is a simple network design that can learn using small observations of training data. In addition, according to their numerical evaluation, the performance of ViterbiNet is comparable to the ideal CSI-based Viterbi algorithm when the training data follows the same statistical model as the test data. In addition, when ViterbiNet uses a series of channels for training, it significantly surpasses the Viterbi algorithm with the same CSI level, proving to be an efficient and reliable neural network-based communication system.

3.1 Viterbi Algorithm

The Viterbi algorithm[2] is a method of decoding convolutional codes. The central concept behind this method is to find the best path through the trellis diagram that is closest to the received data bit sequence. A trellis is a graph in which the nodes are arranged vertically to show the best paths through the states, with each node linked to at least one node at a previous branch. Moreover, the goal is to consider a distance metric to evaluate which one of the paths is better than the others. The following diagram will help understand the trellis concept clearly:

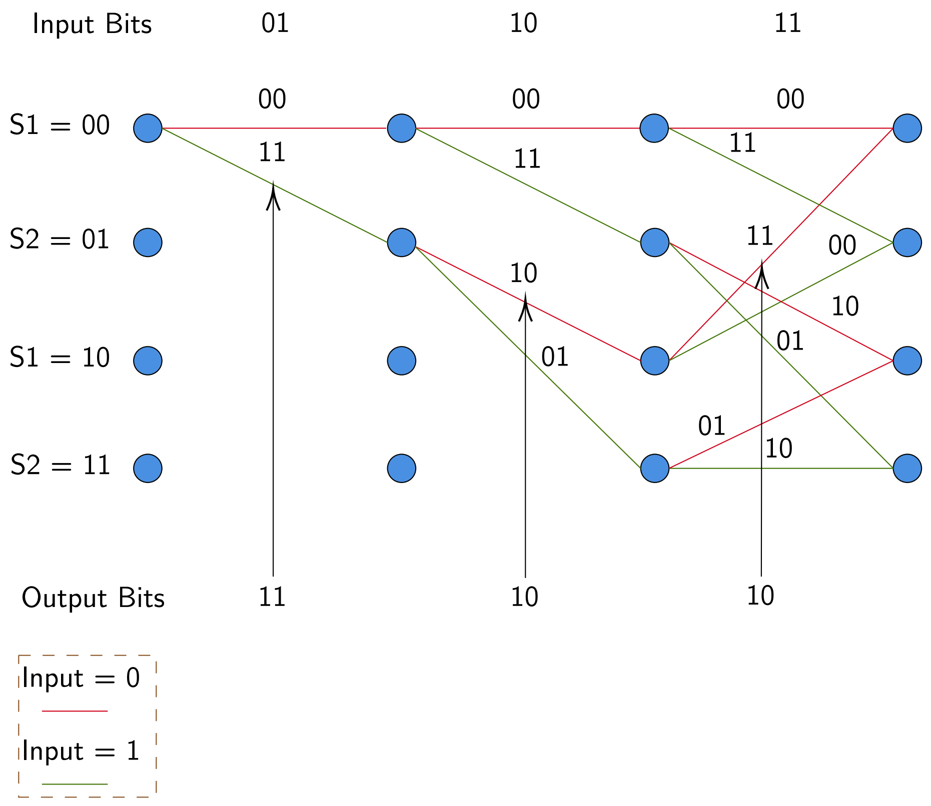

Figure 3.1: Example of a trellis diagram to find the optimal path to decode convolutional codes.

The sequences S1 to S4 represent a starting state (nodes), and the branches show that every potential input sequence corresponds to a distinct trellis path over time. For example, if we wish to decode the received data bit sequence: "01 10 11," we must match the branch's weighted bits with the potential input sequences at each node. If we use the Hamming distance to compare the branch sequence S1 (00) to 01, we get a difference of 1 bit. Then, we iteratively add the resultant bits after each node until we receive the minimum received path from the branch. Consequently, the decoded bits are: "11 10 11" as those branches have the minimum distance in the trellis over time.

The Viterbi algorithm employs trellises in communication system encoders and decoders to simplify the Markov graph sequences. These sequences show a certain number of states and probabilities that the system changes from one state to another. However, the time complexity will be enormous if the trellis has a significant length to compute. So instead, an alternate approach mentioned by Forney could be to use a "sequential trial and error" [3] method to examine the optimal path in the trellis.

3.1.1 Application

The Viterbi algorithm has many use-cases in the wireless communication field. For example, convolutional codes are the main part of the development of Viterbi. If it is not for these codes, we will use block codes to apply the Viterbi algorithm. According to Forney, the problem with block codes is that the rate of decline is much slower than that of convolutional codes with similar decoding runtime.[3].

An essential application for Viterbi is for the use of text recognition. The author mentioned the noise simulation analysis of the diagram graph statistics [3], which uses the error correction of the distorted text. The results from the simulation check suggest that the algorithm can be a viable use case in detecting ambiguous texts due to its availability of confidence intervals.

3.2 Deep MIMO Detection

Similar to our symbol detection project, Samuel *et al.* [4] discusses the application of DL-techniques to recover symbols in MIMO. MIMO is short for multiple-input and multiple-output. It is a concept in the antenna technology for wireless communication that multiple transmitters and receivers can transfer data at once. The main idea behind MIMO is that by allowing data to travel across many signal routes at once, the antennas at both ends of the commutation are merged to reduce mistakes, enhance data speed, and increase radio transmission capacity. This idea is advantageous as multiple antennas, a single device, can aggregate the signals received from each antenna and process them together to remove noise, resulting in a cleaner end signal. However, although the concept of MIMO is efficient, the time complexity to detect such input and output messages is challenging. Therefore, it is proven that machine learning, and in particular, deep learning, can be applied to MIMO [4] so that not only do we get an optimal time complexity, but also the learners can study these rules to detect unknown outputs of future inputs [4].

In a formulation manner, the authors of [4] have presented a linear MIMO detector. It should look like the following:

$$y = Hx + w \tag{3.1}$$

Where y represents a received vector and H is an mxn channel matrix randomly generated with x being an unknown vector of independent and equal probability binary symbols. Moreover, w is the noisy independent and identically distributed (i.i.d), zero-mean Gaussian variables with a variance of σ^2. The main idea with this formula is to detect x given the inputs from H and y.

However, if we apply this formula to a learning-based MIMO detector, the results would be inefficient. Therefore, to make an efficient DetNet detector, the authors alter the linear formula to prevent the receiving vector (y) from working with DetNet [4] directly:

$$\mathbf{H}^\top y = H^\top Hx + H^\top w \tag{3.2}$$

We apply a transpose to a channel matrix to represent some linear transformation that reveals some properties. In evaluating the performance, the authors compared "DetNet" with an Approximate Message Passing algorithm (AMP): a widespread detection iterative algorithm used to detect MIMO. The results indicate that the learning-based detector classifies the outputs accurately and runs "30 times faster" than AMP. Furthermore, DetNet's ability to optimise the distribution of channels rather than a single or even a large-finite set of channels makes it robust and allows it to be used in systems where the channel is not fixed.

3.3 Neural Network Detection of Data Sequences in Communication Systems

In addition to detect symbols using neural networks, Farsad & Goldsmith discusses how different deep learning methods can be applied to create detection algorithms for communication systems that learn directly from data[5]. Data in this context represents a set of messages provided to channel state information (CSI): a piece of information in wireless communication that describes how the signal propagates from the transmitter to the receiver. Therefore, the main goal of the detection algorithm in this application is to recover the messages transmitted from the receiving end at the correct time and in the correct order. The author provides a chart to intuitively explain the complete process of the detection algorithm in the communication system:

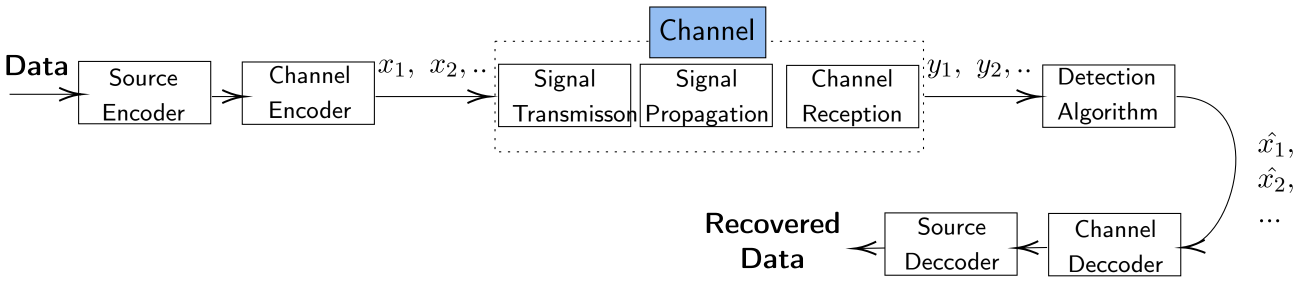

Figure 3.2: Block diagram for digital communication systems [5]

First, the data is transferred to a source encoder, which compresses the data bits. In addition, these compressed bits pass through the channel encoder, where it adds extra redundant bits to the data stream to reduce errors during transmission and reception. Second, the encoded data bits are ready to pass through channel state information (CSI), which propagates the bits from the transmitter to the receiver. Furthermore, the output y is a series of observed bits that are used by the detection algorithm to estimate such transmitted bits $\hat{X}$. Finally, these estimated bits are decoded in error-correction code (Channel decoder) and uncompress the bits to see the recovered bits (Source decoder).

3.3.1 Sliding Bi-directional Recurrent Neural Network

The detection algorithm that the authors propose in this paper is a unique architecture called a sliding bidirectional recurrent neural network (SBRNN)[5]. This category of neural network uses the predicted output from the previous step and uses it as an input to the current step. Though the length of the bits can change every time, Farsad & Goldsmith set the maximum length of BRNN to the same as the channel memory length. However, if the channel memory length is unknown, the BRNN length can be interpreted as a hyperparameter adjusted during the training phase. The working principle of the NN mechanism is to move the BRNN forward one bit after each new bit arrives from the CSI output, hence the word "sliding". Furthermore, the authors estimate the incoming detected bits by using their estimator based probability mass function formula ($\hat{P^k}$): [5]

$$\hat{P^k} = \frac{1}{|\mathcal{J}_k|} \sum_{j \in \mathcal{J}_k} \hat{P_k}^j \tag{3.3}$$

Where $\mathcal{J}_k$ is the weighted sum of a series of starting events for the BRNN to detect over a length of k bits. Finally, we will let j be the iterative slider of the BRNN over the k bits. The advantage of estimating the detected bits using this PMF is that, once the first length of bits has been received and detected, the detector estimates the next set of bits as soon as the signal corresponding to that bit arrives at the destination. Therefore, the evaluation between SBRNN and the traditional Viterbi algorithm can conclude that regardless of whether the CSI is noisy or not, SBRNN can perform better and be more efficient.

3.4 Deep Learning Methods for Improved Decoding of Linear Codes

As mentioned in section 3.3, the estimated transmitted symbols enter through the decoder channel, which is an error correction code technique. Nachmani *et al.* have proved the usefulness of deep learning techniques to reduce the run-time required for decoding linear codes. However, if we look closely, the benefit of an error-correction code is considerably reduced if the decoding method takes a long time to run. While the main idea of finding the codeword closest to the received vector is straightforward, the techniques vary significantly in terms of time and memory needs. An example of a linear correcting code for small to medium-sized blocks is the low-density parity-check (LDPC), which uses a Belief Propagation (BP) decoder algorithm. However, if we are to compute a large size of the linear block, we would need a high-density-parity-check (HDPC). Unfortunately, when using (HDPC) to calculate such a large block size, the run-time performance is poor compared to the improved learning-based decoder algorithm used for BP.

3.4.1 A Neural Belief Propagation Decoder

Nachmani *et al.* propose that deep learning-based methods can help generalise the BP decoders and improve the complexity of an error-correction code by applying weights onto the BP decoder. By applying such weights, we can train the neural networks using a stochastic gradient descent [6]. In other words, we can update the weights after each training sample for each iteration. A significant advantage of using a learning-based technique for decoding problems is that not only it requires a short number of parameters, but it can also train efficiently, even using a single codeword [6]. Furthermore, it does not matter whether or not the transmitted codewords are mixed or single bits because the model can output the same error rate. Consequently, the authors during this paper have evidenced that the learning-based BP coding technique for HDPC has an improvement of 0.9dB, compared to the normal HDPC with a BP decoder.

3.4.2 Neural Min-Sum Decoding

Another decoding algorithm that can help reduce the run-time of error correction codes is the min-sum algorithm. A min-sum algorithm is an approximation approach for decoding LDPC code that can considerably reduce the computational cost of the message-passing BP. The problem with the traditional min-sum is that it creates a significant bits length. As a result, the "propagated information seems to be more dependable than it is"[6], which can reduce the bit-error rate. Similarly to the learning-based approach for BP, the learning-based min-sum will also apply weights in order to "train the decoder as a neural network".[6] The advantage of such a learning-based approach is the reduced use-case of constant multiplication, which is the leading cause of the expensive run time.

3.5 An Artificial Neural Net Viterbi Decoder

As previously discussed in section 3.1 the Viterbi algorithm estimates the maximum likelihood of decoding convolutional codes and plays a vital role in digital communication. The main idea of applying machine learning techniques to these traditional algorithms is to optimise the time complexity while maintaining the same performance, if not better. The paper [7] written by Wang & Wicker introduces Artificial Neural Network (ANN) into the Viterbi algorithm to decode symbols at a faster rate.

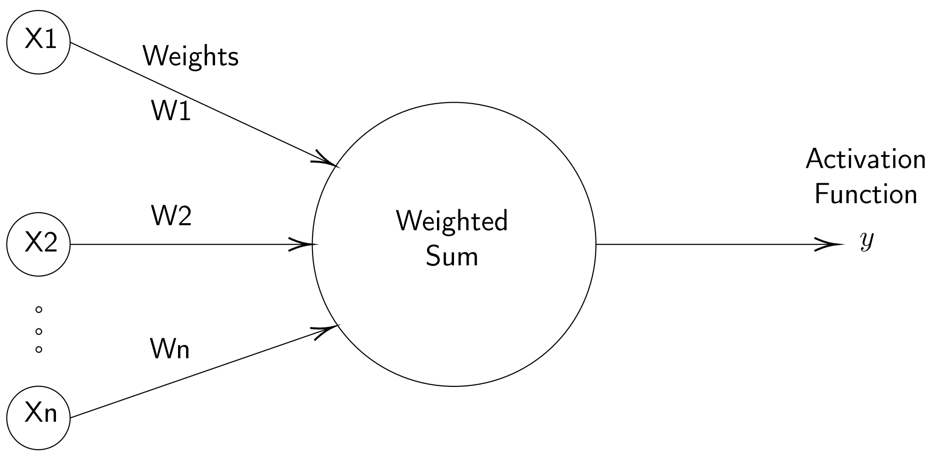

Figure 3.3: Input-output representation of a Neural Network

A neural model (shown in fig 2.3) contains inputs (x1 to xn), their corresponding weights (w1 to wn), which are aggregated into a weighted sum before applying to the activation function. Hence, the formula for an input-output process should look like this:

$$y = f(\sum_{i=1}^{N} w_i x_i) \tag{3.4}$$

Based on past implementations on ANN with the Viterbi Algorithm, performance is considered to be sub-optimal [7]. This implementation shows that the ANN's are trained to retrieve coefficients and will only be able to decode minimal codes. Consequently, the authors discuss some modifications that vastly improve the efficiency of the decoder. For example, the ANN only requires one-sixth of the number of transistors required by the digital decoder due to the analogue/digital hybrid design resulting in a high-speed and efficient decoder [7]. What's more, the analogue/digital hybrid is designed to solve the maximised dynamic range of Threshold Logic (TL) neurons by allocating bits represented by the output of each register neuron among several neurons [7].

3.6 Blind Channel Equalization using Variational Autoencoders

Blind channel equalisation is a concept in communication systems where the transmitted data bits over a channel are the same as the receiver. Its "blindness" is due to the data bits only being considered from the transmission side. The traditional algorithm used to equalise blind channels is called Constant Modulus Algorithm (CMA) [8]. However, like every other traditional algorithm mentioned in this related work section, it suffers from slow performance. Luckily, Caciularu & Burshtein [9] have introduced a deep neural network technique called the Variational Autoencoder, which learns data from a latent representation—in this case, using a learning-based technique to equalise the symbols from receiver to the transmitter.

Correspondingly to the Viterbi algorithm that estimates the maximum likelihood of symbols, the variational autoencoder is trained to estimate the same method used to equalise the blind channel. Consequently, the ML-technique shows vast improvements compared to the traditional CMA in symbol error rate. In particular, the model proposed by Caciularu & Burshtein has proved to have similar results to the non-blind adaptive linear minimum mean square error adjuster of the channel used for research [9].

3.7 Attention Neural Network for Signal Modulation Detection

Attention neural network is a mechanism to selectively concentrate on relevant things while ignoring others in deep neural networks. The main idea behind attention in neural networks, particularly RNN, is to summarise the input data into a vector that applies in the last hidden state of the RNN algorithm. As a result, all the other states are ignored because the final state is treated as the initial state. As part of our advanced concept of ViterbiNet, we will explore how attention mechanisms are used in digital communication.

The work written by Luo *et al.*, tackles the issue of poorly resulting detection of signal modulation using RNN [10]. The solution that the authors propose is the attention mechanism in the "residual block network" [10]. The integration of attention of block networks summarises the different types of signals into a vector. Thus, the neural network only focuses on the signals based on the summarised vectors. As a result, the attention-based RNN performs optimally compared to the normal RNN.

Chapter 4: Core Features

In this chapter, we will discuss the minimum features required to replicate the work of Shlezinger *et al.* in Python. The replication includes the implementation of the generative dataset, the iterative Viterbi algorithm and the learning-based Viterbi algorithm. Moreover, we will discuss the performance of both algorithms using a `semilogy` graph and the model summary of the deep neural network. Furthermore, we will intensely discuss and understand each section using figures and pseudocodes.

4.1 Data Considerations

The dataset is a generative function that entails the digital communication systems simulation process. This section will discuss the theory behind one of the channel-based datasets and how we implement this feature in Python.

A communication system requires an input and an output message. For example, sending an email to a user generates an electrical signal. The signal is an input message that passes through a transmitter modulation technique that maps the data bits to symbols. Once the modulated symbols are ready, they are transferred across a channel. A channel is a medium over which the symbol is transmitted to a receiver over a certain distance. A channel that we will use in this project is intersymbol interference (ISI) with additive white Gaussian noise (AWGN). The ISI distorts the input symbol over a finite number of times, spreading them to interfere with the adjacent symbol at the sampling instant. An example diagram is given below to understand the ISI technique intuitively:

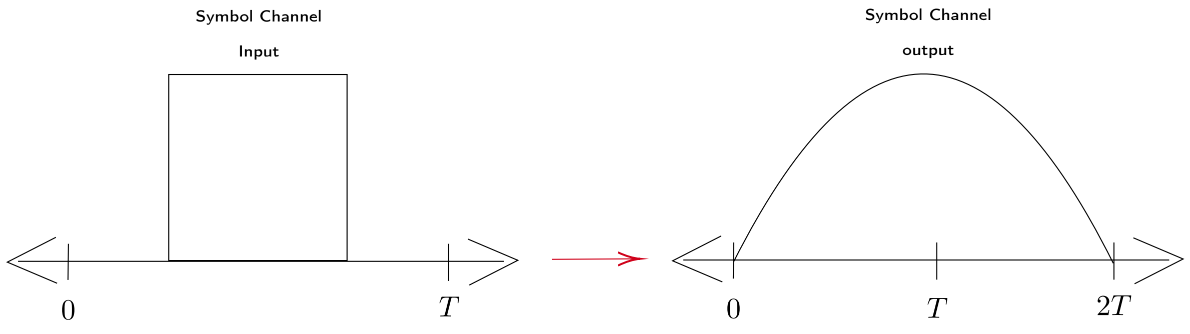

Figure 4.1: Diagram of the ISI Channel

We can see that when we send a symbol over a finite number of times, T, and by the time the symbol has gone through the channel, it spreads into a more extended time, causing distortion. Therefore, the current symbol interferes with the following symbol that we would like to transmit, starting at T and finishing at 2T. Moreover, the channel introduces a noise signal that mimics the effect of many random processes in digital communication. In this case, the noise signal we will use with the ISI channel is the AWGN. The main idea of AWGN is that it adds noise independently from time to time through a Gaussian distribution. The generator function for the ISI channel with AWGN is given in the following formula [1]:

$$Y[i] = \sqrt{p} \cdot \sum_{\tau=1}^{l} (h)_\tau \, S[i - \tau + 1] + W[i] \tag{4.1}$$

To summarise this formula, $Y[i]$ represents the output symbol at time index i. The output is derived by taking the square root of the signal-to-noise ratio $p \in [-6, 10]$ and multiplying its values with AWGN $W[i]$. Moreover, we add these resultant values to our modulated frequency signals stochastically mapped by $S[i - \tau + 1]$, where τ is the current iteration point over channel length 4 l, which is assumed to be smaller than the blocklength of symbols. The modulation technique that we will use to convert data bits to symbols is the binary phase-shift keying (BPSK) constellation, where data bits with 0 will be mapped to +1, and 1 will be mapped to -1 symbol, i.e. $S = [1, -1]$. Finally, we let h be an exponentially decaying fading channel given by $(\mathbf{h})_\tau \triangleq e^{(-\tau-1)/5}$ [1].

To evaluate the robustness of our ViterbiNet model, we assume that the DNN is trained on imperfect channel. Furthermore, this trained ViterbiNet model will estimate the results based on the test data of the perfect channel. To apply a noisy channel, we expose its estimates of the fading channel h. In particular, these estimates are corrupted with the independent and identically distributed (i.i.d.) zero-mean Gaussian noise with variance $\sigma^2 = 0.1$ [1]. Moreover, the 5000 samples we use for training noisy ViterbiNet are divided into ten subsets, each generated from a fading channel h with different noise estimates [1].

We take the average symbol error rate (SER) values per signal to noise ratio (SNR) [-6, 10] dB over 50000 Monte Carlo simulations for the ISI channel with AWGN [1]. We test these simulations on Viterbi and ViterbiNet models through perfect and imperfect channels.

Implementing the training dataset in Python is straightforward. Firstly, we discuss how the symbols are generated:

1. We generate symbols using NumPy's `randint()` method, which generates integers 1 and 2, 5000 times.
2. We take the random integers and reshape the array to a 5000x4 matrix to fit the channel length. In addition, to ensure that the symbols follow the BPSK modulation technique, we take integer 1 and map it to -1, and 2 will be mapped to 1 symbol, i.e. $\mathcal{S} = 1, -1$.
3. Moreover, we reshape symbols ($\mathcal{S}[i - \tau + 1]$) to take the first symbol in every row from the matrix and shift it to the back for every iteration.
4. Finally, to complete $\mathcal{S}[i - \tau + 1]$, we implement $e^{(-\tau-1)/5}$ using Numpy's library to complete the fading channel $(h)_\tau$. Thus, we take the fading channel and multiply it with the resulting symbols for each iteration.

Finally, we need to generate the channel SNR values ranging from -6 to $10dB$ to complete $\sqrt{p}$. However, SNR contains a vast dynamic range of values. Therefore, we must condense these values by converting SNR decibels to a standard scale. We do this by applying this expression $10log(p)$, where $p \in [-6, 10]$. Furthermore, we complete the Gaussian noise by applying the normal distribution of the symbols in each iteration.

We calculate the noise by taking 5000 samples divided into ten subsets. i.e. one subset has 500 indexes. At a given index of a frame, the receiver only has access to a noisy estimate of the fading channel, in which the entries are corrupted by i.i.d. zero-mean Gaussian noise with variance sqrt(0.1) [1], i.e. if the distribution is independent, then the covariance matrix is diagonal.

Therefore, we take the product of two inputs: the normal distribution of the fading channel size and the diagonal matrix of the fading channel. Finally, apply the noisy fading channel estimates to each subset of the training data.

4.2 Viterbi

As aforementioned in section 4.1, when we send a symbol over a finite number of times, T, and by the time the symbol has gone through the channel, it spreads into a more extended time, causing distortion. Our task is to recover the distorted symbols using decoder algorithms from the receiver stage. An algorithm that will specialise is the iterative Viterbi, which achieves minimal error probability. The implementation of this algorithm is straightforward. Before starting the Viterbi decoder, we need to get each state's conditional probability distribution function (PDF). We get the PDF by taking the values of the channel outputs in a given sample space and describing all the possible values and likelihoods that a random variable can take within a given range. The pseudocode for the Viterbi algorithm states the following:

Algorithm 1: The Viterbi Algorithm [1]

```
1 Input: Block of channel outputs $y^t$, where $t > l$;
2 Initialisation: Set $k = 1$, and $\tilde{c}_0(\tilde{s}) = 0, \forall \tilde{s} \in \mathcal{S}^l$;
3 Compute $\tilde{c}_k(\tilde{s}) = \min_{u \in \mathcal{S}^l : u^{l-1} = \tilde{s}_2{}^l} (\tilde{c}_{k-1}(u) + c_k(\tilde{s}))$;
4 If $k \geq l$, set $\hat{(s)}_{k-l+1} := (\tilde{s}^o)$, where $\tilde{s}^o = \arg\min_{\tilde{s} \in \mathcal{S}^l} \tilde{c}_k(\tilde{s})$;
5 Set $k := k + 1$. If $k \leq t$ go to step 3;
6 Output: decoded output $\hat{s}^t$ where $\hat{s}^t_{t-l+1} := \tilde{s}^o$ ;
```

The algorithm inputs the estimated channel outputs at time t (y^t), where l denotes the channel's memory, assumed to be smaller than the blocklength, i.e., $t > l$. [1]. Firstly, we initialise the algorithm by generating a trellis matrix of shape $16\text{x}2$, where $\tilde{c}_0(\tilde{s})$ is the constellation point containing the symbol transmitted at time index k. We get sixteen after calculating the number of states and two from the size of the constellation key applied in the modulation technique. The calculation considers the size of the constellation key to the power of the memory length. In this case, we have two constellation sizes and four memory taps, resulting in 16 states $(2^4 = 16)$. Moreover, elements within the trellis matrix should contain the index of the state. For example, $[0][0]$ is the first state in the matrix that contains the value 0, $[0][1]$ is the second state that contains the value 1, and we repeat this process until we reach $[15][1]$ which is the sixteenth state containing value 15. We store the values in the trellis matrix into a variable u.

In step 3, we create a vector that should store a minimum of two values in each state after adding the elements from the trellis matrix (u) and elements from the log-likelihood matrix $(\tilde{s})$. In the following steps, we obtain the minimum value of two states, store it in a vector $(\hat{s})$, and compare the following states until we reach the maximum blocklength l. Finally, we retrieve the vector $(\hat{s})$ with the shortest state path that decodes the symbol concerning the time blocklength.

The advantage of implementing the Viterbi algorithm is that the algorithm solves at a computational complexity that is linear in blocklength t [1]. However, the disadvantage is that the CSI must be fully known to compute the log-likelihood function $c_i(\tilde{s})$ and to produce results optimally. Moreover, retaining full CSI is difficult, especially in a wireless communication domain potentially saturated with noisy signals [1].

4.3 ViterbiNet

The section will discuss the approach and implementation of the DL-based symbol detector based on the Viterbi algorithm, which does not require CSI [1].

Although the Viterbi requires accurate knowledge of the CSI to compute the log-likelihood function $c_i(s)$, the function also depends on the estimates of channel outputs $y[i]$ and its transmitted symbol. Therefore, in the ViterbiNet structure, the cost function's detailed computation is replaced with a learning-based method to evaluate computation cost functions from the training data [1]. For example, the ViterbiNet will take the input of the channel outputs $y[i]$ and provide an output of the log-likelihood function $c_i(s)$ after learning and evaluating cost functions from training data. Figure 4.2 below illustrates how the Viterbi algorithm computes symbol decoding taking the learning-based log-likelihood function as the input.

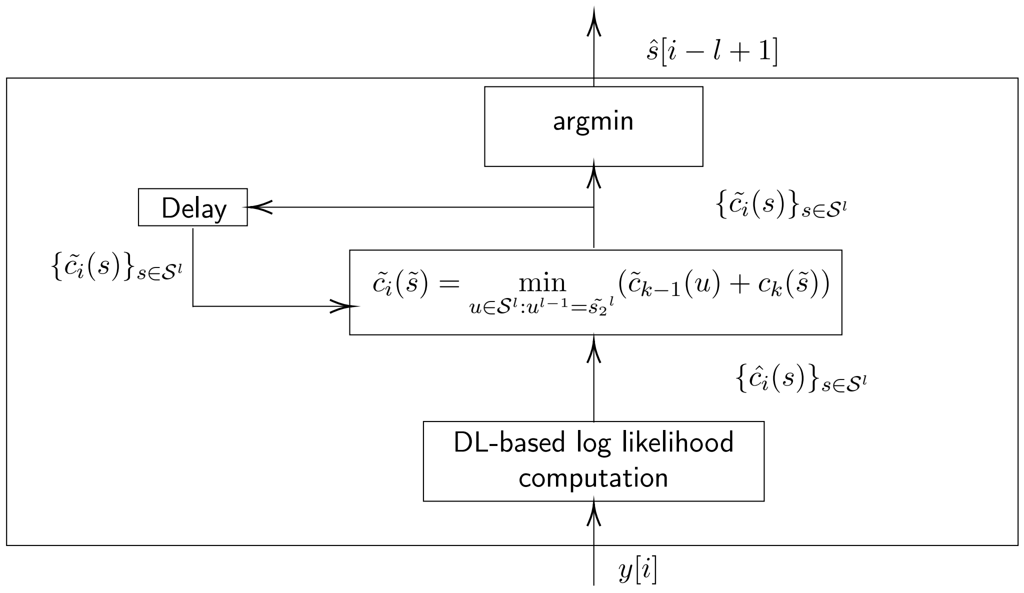
$\hat{s}[i-l+1]$
argmin
Delay
$\{\tilde{c}_i(s)\}_{s\in\mathcal{S}^l}$
$\{\tilde{c}_i(s)\}_{s\in\mathcal{S}^l}$
$\tilde{c}_i(\tilde{s}) = \min_{u\in\mathcal{S}^l:u^{l-1}=\tilde{s}_2^l}(\tilde{c}_{k-1}(u)+c_k(\tilde{s}))$
$\{\hat{c}_i(s)\}_{s\in\mathcal{S}^l}$
DL-based log likelihood computation
$y[i]$

Figure 4.2: Proposed DNN-based Viterbi decoder. [1]

The figure demonstrates how the Viterbi algorithm is carried out with the DL-based log-likelihood estimates. The channel outputs $y[i]$ are treated as an input to the ViterbiNet so that the model can understand and generalise the log-likelihood function of a symbol ($\hat{c}_i(s)$) based on the training data. Moreover, we continue the same approach as mentioned in the Viterbi algorithm. For example, consider a trellis matrix of 2 states, containing the constellation size c of two and memory length l of one i.e. 2^1 states ($c_{\tilde{k-1}}(u)$, where k is the state index): $[0][0] = 0$ and $[0][1] = 1$ and the log-likelihood estimate $c_k(\tilde{s})$ is 2.86. Firstly, we evaluate the state index $[0][1] : 1 + 2.86 = 3.86$ and in the next state $[0][0]$, we evaluate that the computation $0 + 2.86 = 2.86$. After adding our default symbol with the log-likelihood function, we consider the minimum value of two states. In this case, we know that state $[0][0]$ has the minimum value compared to $[0][1]$. Therefore, we pass the state to the next block containing the log-likelihood function of the symbol $\tilde{c}_i(s)$. In the next block, we extract the index of the minimum state to determine the shortest path to decode symbols. In this case, we know that the index is 0 ($\hat{s}[0-1+1] = 0$).

However, if we have a higher memory length l, we would send the minimum symbol to the delay block and continue to compare the minimum states of the next transmitted symbol until we reach the targeted l. For example, if we set $l = 2$, we would have four states. The first two states are compared together to find the minimum value, and then the program delays the output until we compute the minimum value for the other two states. As a result, the algorithm outputs two indexes from the "argmin" block, indicating the shortest state path to the decoded symbol.

Furthermore, we will discuss how the ViterbiNet computes the log-likelihood function with a figure, illustrating the model architecture:

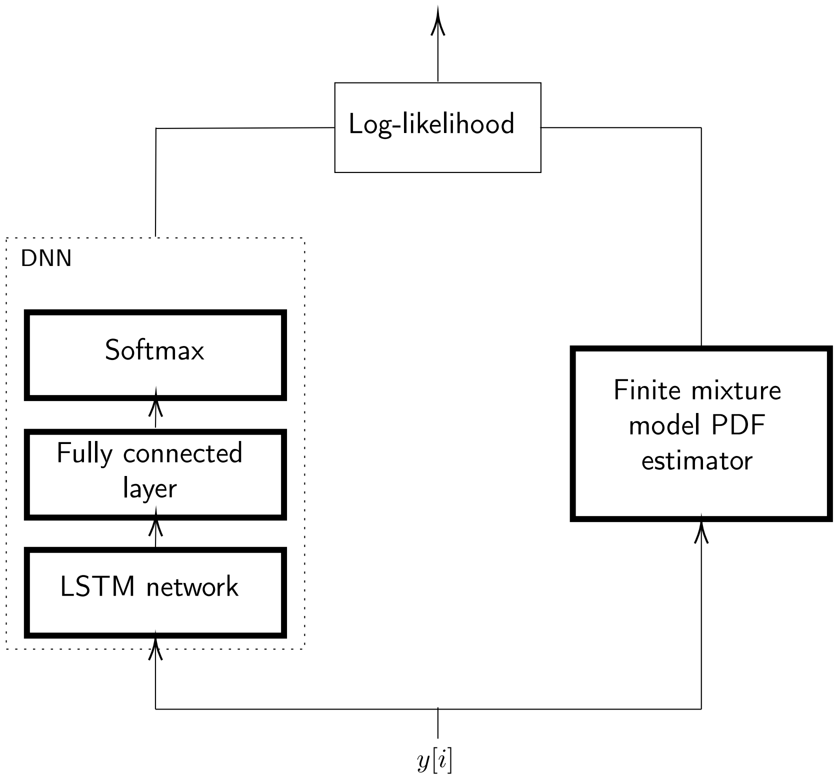

Figure 4.3: ViterbiNet log-likelihood computation. [1]

On the left-hand side of the figure, we are introduced to the DNN topology. This structure consists of a 1x100 long short term memory (LSTM), followed by a 100x50 and a 50x16 fully-connected layers, using the intermediate rectified linear unit (ReLU) and softmax activation functions, respectively [1]. This approach is proposed by Shlezinger *et al.* for an efficient sequence prediction problem. Moreover, the LSTM's capability of learning selective patterns for a long duration of time makes it an appropriate use case for learning channel outputs from the training data over a finite-memory stationary causal channel. In addition, we feed all the LSTM outputs to a fully connected layer with a ReLU function to allow the neural network to learn nonlinear dependencies so that the network can approximate a linear function when necessary, with the flexibility to also account for non-linearity. Similarly to the CSI, which outputs probabilities based on the channel's input and output, we need to apply a similar approach to our DNN topology. Therefore, we apply a softmax output layer that outputs a probabilities vector in this case.

However, DNNs use loss functions to optimise the model during training. In addition, an optimised model would rely on minimising the loss function. In this case, the optimising solver we use is Adam, with a learning rate of 0.0005. Since we are outputting probabilities, we would most likely use the cross-entropy loss function to measure the distance from the actual values. As a result, the output probability structure would be the conditional distribution of the symbols given the channel output: $P(\mathcal{S}_{i-l+1}^{i}|Y_{[i]})$ [1]. On the other hand, the Viterbi algorithm requires the conditional PDF of channel output given their symbols: $P(Y_{[i]}|\mathcal{S}_{i-l+1}^{i})$ [1]. Therefore, the outputs from the DNN architecture are not enough.

A solution to overcome the minimisation of the loss function is to apply a mixture model and, in particular, a Gaussian-based expectation maximisation algorithm [1]. The idea of a mixture model is to maximise the likelihood estimates by taking the mean and standard deviation parameters for each channel output. Finally, we get the log-likelihood of the ViterbiNet approach once we get the dot product of the estimates from both DNN and the mixture model, which should give us the condition PDF of the channel output given their symbols.

We need to replicate their approach strictly when training Shlezinger *et al.*'s DNN topology. Firstly, the network is trained using 5000 training samples to minimise the cross-entropy loss [1]. In particular, the training sample is derived from section 4.1, as discussed earlier. Then, we need to convert the NumPy array of both the channel and symbols to tensors. Moreover, we combine tensors into a tensor-dataset to apply batches (27) in the data loader. Finally, we initialise the model training, treating the channel outputs as the `"X_train"` and the symbols as `"y_train"`. We will update the loss function and apply back-propagation to improve the model training for every epoch (the maximum number of epochs is 100). We can precisely visualise the summary of our model using Pytorch's `summary()` method:

Layer (type:depth-idx)	Output Shape	Param #
ViterbiNet	–	–
LSTM: 1-1	[5000, 1, 100]	41,200
Linear: 1-2	[5000, 50]	5,050
Linear: 1-3	[5000, 16]	816

Total params: 47,066
Trainable params: 47,066
Non-trainable params: 0
Total mult-adds (M): 235.33

Table 4.1: ViterbiNet Model Summary

The summary table shows us how many parameters the model utilises for each layer. For example, we provide an input size of $(1, 5000)$, the same size as the channel output. Firstly, the LSTM layer takes the shape of $(1\text{x}100)$, indicating that the learnable parameter would match according to its input-output size, in this case, $41,200$. Then, the two fully-connected layers have learnable parameters of $5,050$ and 816. It is essential to keep track of the number of parameters present in a model to prevent over/under-fitting.

4.4 Evaluation

The previous sections discuss the implementation of Viterbi and ViterbiNet. This final section of the chapter discusses the evaluation of how these implemented models perform over a simulation. For example, figure 4.4 helps analyse both models' performance in recovering symbols over unseen channels using a `semilogy` graph. This graph is an appropriate approach due to our conversion of SNR dB to a linear scale from section 4.1.

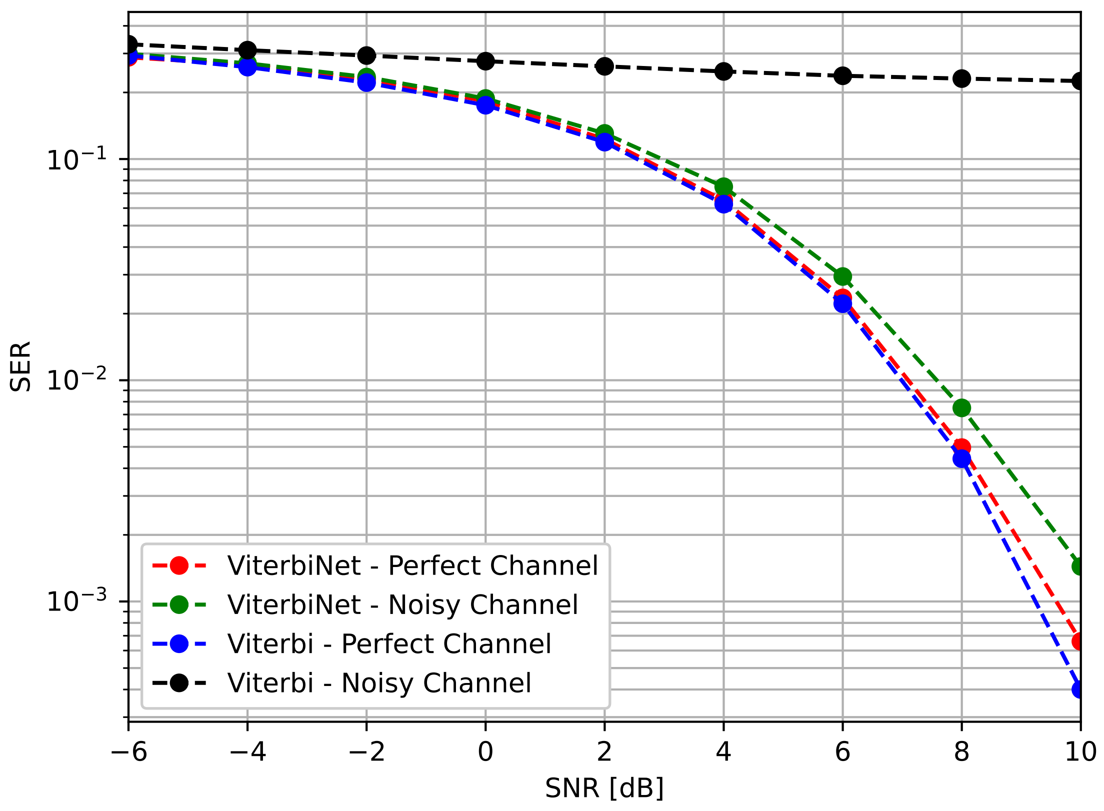

Figure 4.4: Viterbi vs ViterbiNet: Symbol Error Rate

The graph's y-axis represents the symbol error rate (SER), and the x-axis is the signal to noise ratio (SNR), measured in decibels. The blue and black lines represent the performance of the iterative Viterbi algorithm over a perfect channel and a noisy channel. Similarly, the red and green lines are the performance is the DNN integrated Viterbi algorithm. Both models have a high SER at the beginning of the SNR, and the error gradually reduces as the SNR increases. A strong signal is better than a weak one due to the low chance of errors. More precisely, the low number of errors increases with SNR because of noise. Ultimately, the red ViterbiNet line follows the same trend as the blue Viterbi. However, the red line begins to deviate after the sixth signal. In addition, the black line of the Viterbi algorithm performs poorly over noisy channels throughout the SNR. This line indicates that the symbols are incorrectly predicted regardless of the signal strength. On the other hand, the green line ViterbiNet effectively recovers symbols over noisy channels, as it almost follows the same trend as the red line.

We have successfully reproduced the work by Shlezinger *et al.* in Python/PyTorch. For example, both models performed as expected in the `semilogy` graph and are very similar to the results provided in the paper [1]. In particular, the ViterbiNet over noisy channels performs optimally compared to the traditional Viterbi. The functional results are due to the network's learning efficiency from previous trends to recover symbols, yet, predicting intuitively over noisy data.

Chapter 5: Advanced Features

The main idea of this project is to replicate Shlezinger *et al.*'s work and go above and beyond in terms of exploring other neural network topologies and different noisy channel levels to evaluate the ViterbiNet performance against Viterbi. In particular, we will apply the same mixture model and training approach as discussed in chapter 4 to these new network topologies. Therefore, we will discuss our advanced features in this chapter.

5.1 Convolutional Neural Network

Although Shlezinger *et al.* proposed LSTM as part of their model architecture, we explore convolutional neural networks (CNN) to evaluate how they would perform against LSTMs. Unlike LSTM, which is capable of predicting a sequence problem, CNN can recognize objects and patterns. Therefore, we hope to sense that the CNN can detect some patterns arising from the channel output to influence their decision. The proposed structure of a CNN is shown graphically below:

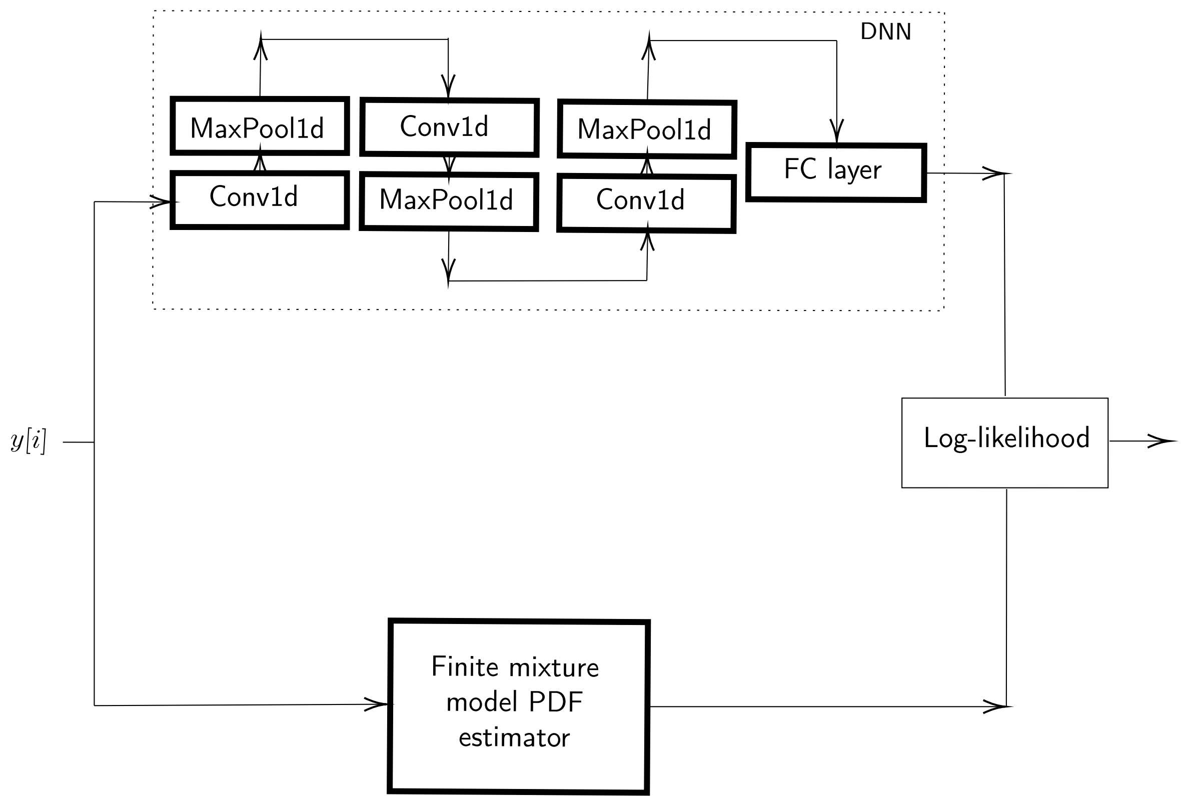

Figure 5.1: CNN log-likelihood computation

First, we will define our model as having three conv1d layers, while each layer contains a max pool of size two. We set the convolutional and max pool layers three consecutive times so that the model has a better chance of learning features from the input data. When the convolutional layer receives an input, the filter will slide over each 7x7 matrix from the input feature map until it slides over every 7x7 matrix. So when a filter convolves a given input, it gives us a resulting output.

However, the problem with feeding the resulting output into the next convolutional layer is that the model may experience the following layer recording the precise position of features in the input. This indicates that small movements in the position of the feature in the input data will result in a different feature map. Therefore, the pooling layer is essential as each filter in a convolutional layer is downsampled, which reduces the dimension in the feature map. As a result, the number of learning parameters and network processing is reduced. In this case, we use a max pool layer to calculate the maximum value in each feature map. Finally, the resulting outputs from the final convolutional layer represent high-level features in the data. Therefore, we can flatten the outputs and feed them into a fully connected layer to learn non-linear combinations of these features.

We take the first `conv1d` with an input size of 1 and an output of 32 in its implementation. Moreover, the second `conv1d` takes an input of 32 and outputs 64. Furthermore, the final `conv1d` layer takes an input of 64 and has an output size of 128. In addition, the `MaxPool` layer in between each `conv1d` layer has a size of 2. More precisely, this pooling layer should be half the output size from the prior `conv1d` layer. Thus, reducing the parameter size and the multiplication/addition computation. We can visualise the model summary to analyse the pooling feature closely:

Layer (type:CNN)	**Output Shape**	**Param #**
Conv1d: 1-1	[8, 32, 5000]	256
MaxPool1d: 1-2	[8, 32, 2500]	–
Conv1d: 1-3	[8, 64, 2500]	14,400
MaxPool1d: 1-4	[8, 64, 1250]	–
Conv1d: 1-5	[8, 128, 1250]	57,472
MaxPool1d: 1-6	[8, 128, 625]	–
Linear: 1-7	[5000, 16]	2,064
Total params: 74,192		
Trainable params: 74,192		
Non-trainable params: 0		
Total mult-adds (M): 883.28		
Input size (MB): 0.16		
Forward/backward pass size (MB): 31.36		
Params size (MB) : 0.30		
Estimated Total Size (MB): 31.82		

Table 5.1: CNN Model Summary

The outcome of the first max-pooling layer will be a single feature map with each dimension reduced from 5000 to 2500. The process continues until the final max pool layer provides an output of shape 5000x16 and then flattens the output to feed to a fully connected output layer. As a result, the total parameter size is $74,192$. This total size is a lot higher than ViterbiNet, with a difference of over $25,000$ parameters. A potential improvement to consider is to decrease the parameter size by reducing the input-output sizes to 1x16, 16x32, 32x64, and 64x16.

5.2 Stacked LSTM

As mentioned earlier in the ViterbiNet section, the architecture comprises a single LSTM layer and two fully connected layers. Moreover, the use of LSTMs is robust in sequence prediction problems such as symbol detection. Thus, this section will dive deeper into the LSTM and see how our model would perform in recovering symbols with a stacked layer of LSTMs. The stacked LSTM is a variation of this architecture that has multiple LSTM layers piled up on one another, each with numerous memory cells. As a deep learning approach, stacking an LSTM atop another makes the model deeper and more accurate. In addition, given that LSTMs operate well on sequenced data such as the ISI channel, the addition of layers adds levels of abstraction of input observations over a finite series of time. Let us see how we will set up our stacked LSTM architecture to output log-likelihood computation:

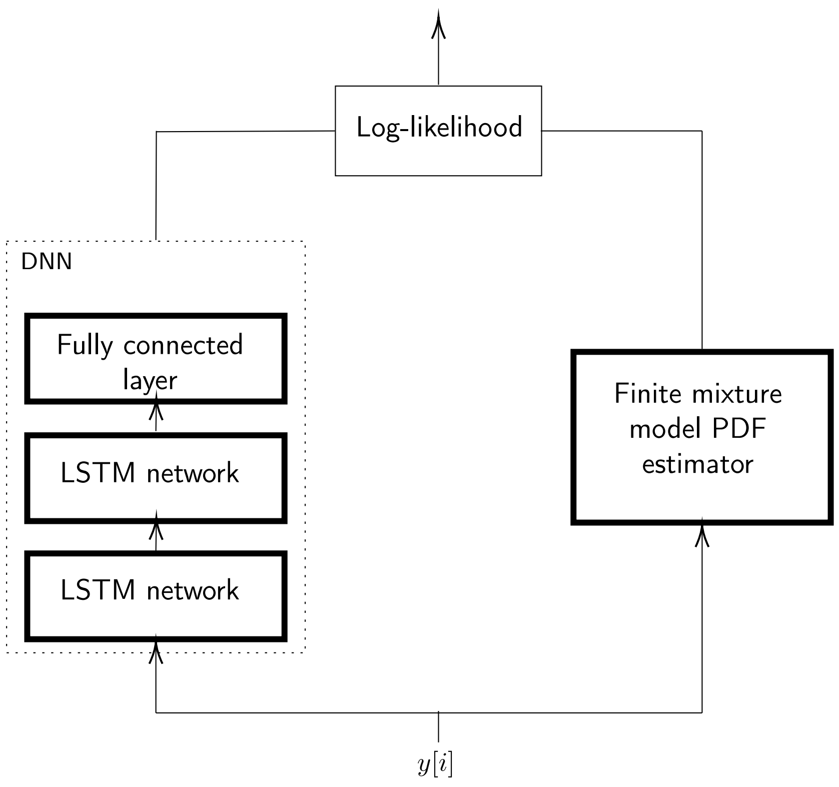

Figure 5.2: Stacked LSTM log-likelihood computation

As we can see from the figure, the DNN follows the same architecture as the ViterbiNet. In addition, however, we replace one of the fully-connected layers with an LSTM for a deeper network. We can easily create stacked LSTM models in Pytorch's deep learning library. To implement the stacked LSTM, we create hidden and internal states initialised with zeros. Then, we pass the first LSTM layer with input, hidden, and internal state at the current timestamp, which should return a new hidden state, current state, and output that we can feed to the following LSTM layer. Furthermore, we reshape the output (or flatten) so that output can pass to a fully-connected layer with a sigmoid activation function.

The training phase of the stacked LSTM will be similar to ViterbiNet, although the only difference is that we will try to explore other activation and loss functions. For example, we apply a sigmoid activation function in the fully connected layer. The results from this function should be between 0 and 1, which we can infer how confident the model is with recovering symbols from the channel outputs. Moreover, to suit the sigmoid function, we will apply a binary cross-entropy loss function which quantifies the difference between two probability distributions. More precisely, we can visualise the model summary to analyse the stacked feature closely:

Layer (type:StackedLSTM)	Output Shape	Param #
LSTM: 1-1	[5000, 1, 100]	41,200
LSTM: 1-2	[5000, 1, 100]	41,200
Linear: 1-3	[5000, 16]	1,616
Total params: 84,016		
Trainable params: 84,016		
Non-trainable params: 0		
Total mult-adds (M): 420.08		
Input size (MB): 0.02		
Forward/backward pass size (MB): 8.64		
Params size (MB) : 0.34		
Estimated Total Size (MB): 9.00		

Table 5.2: Stacked LSTM Model Summary

The model summary indicates that both LSTMs have the same 41,200 parameters to train with over 5000 symbols. Similarly, the exact number of parameters is also required for the single LSTM layer in ViterbiNet. In addition, all the outputs from the stacked LSTMs are flattened are fed into a fully connected layer. However, the stacked LSTM's total parameters doubled that of the ViterbiNet model. As a result, the multiplications and additions also doubled to 420.08 million operations.

5.3 Stacked GRU

Similarly to the stacked LSTM, we will apply the same concept and architecture to the gated recurrent units (GRU). RNNs have a short-term memory issue, which is their fundamental flaw. There is a problem transmitting information from the previous time step to the subsequent time step if the sequence is lengthy enough. For example, if the RNNs are trying to process a long sequence of channel outputs to recover symbols, RNNs may leave out important information from the beginning. As a result, networks such as LSTM and GRU are introduced to mitigate such problems. The GRU is very similar to LSTM. The sole difference between these two features is that, unlike LSTM, which uses a tuple that contains both the cell state and the hidden state to transmit information, the GRUs do not use the cell state and instead rely on the hidden state. Using figure 5.3, we can closely analyse the model's architecture and discuss the implementation of GRU in Pytorch.

This figure follows a similar architecture as the stacked LSTM concept. The only difference is that we replace LSTMs with GRUs and flatten this information to connect to an output layer. The implementation of this feature is relatively straightforward. Firstly we create a 1x100 GRU layer, which we can pass in the input and hidden state into the model. Secondly, the information gained from the initial hidden state is passed onto the following 100x100 GRU layer. This layer should help make the model more accurate in recovering symbols by gaining deeper information about the channel outputs. Finally, we can flatten the outputs from the second GRU and feed them to a 100x16 fully connected layer with a sigmoid activation function. In particular, similar training

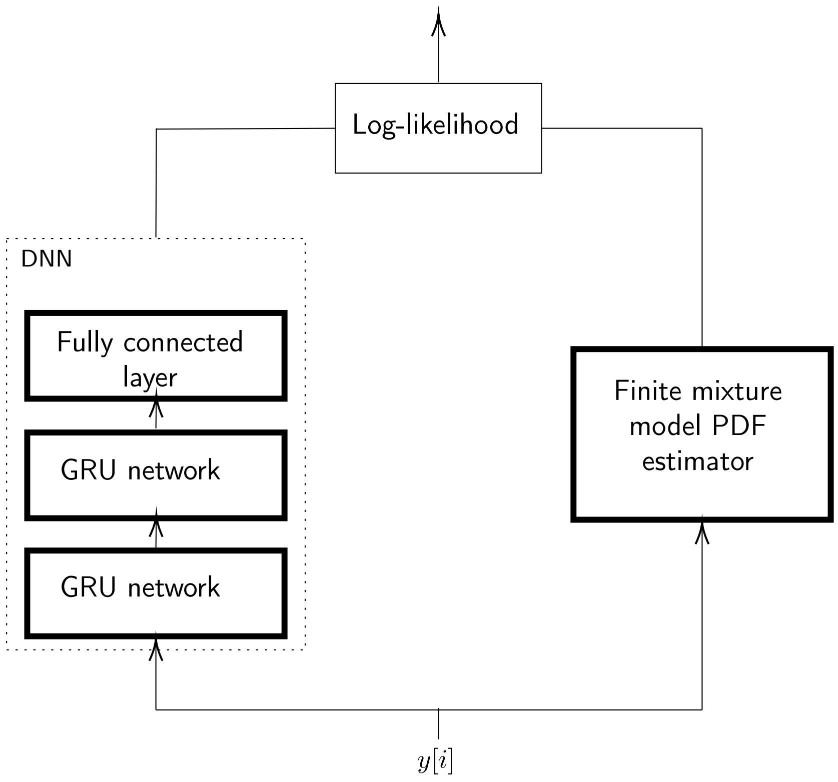

Figure 5.3: Stacked GRU log-likelihood computation

and loss methods from the stacked LSTM concept are used in this model also. We can analyse the table containing the model summary below:

Layer (type:StackedGRU)	Output Shape	Param #
GRU: 1-1	[5000, 1, 100]	30,900
GRU: 1-	[5000, 1, 100]	30,900
Linear: 1-3	[5000, 16]	1,616
Total params: 63,416		
Trainable params: 63,416		
Non-trainable params: 0		
Total mult-adds (M): 317.08		
Input size (MB): 0.02		
Forward/backward pass size (MB): 8.64		
Params size (MB) : 0.25		
Estimated Total Size (MB): 8.91		

Table 5.3: Stacked GRU Model Summary

In the stacked GRU model, it is clear that the total number of parameters is reduced significantly compared to the stacked LSTM. As mentioned earlier, the reduction of parameters could be due to the loss of the cell state and not requiring the need for memory units. However, the fully-connected layer has the same parameters as the stacked LSTM. As a result, the total multiplication-addition was significantly reduced to 317.08 million operations.

5.4 Attention Layer

In this section, we will discuss two types of attention. Firstly, an attention mechanism in the LSTM layer to search for the most relevant information. Secondly, as a self-attention model that calculates multiple inputs into one.

5.4.1 Attention-based LSTM

In this section, we will focus on implementing an attention-based LSTM model. Though the LSTMs are trained with a small number of samples, they are potentially constrained by the need to encode all channel outputs to a set length internal vector. Thus, we will explore an advanced concept in the "attention" based LSTM network. Moreover, the integration of the attention mechanism into the LSTM network shows how attention is paid to the input channel when predicting the symbol. Furthermore, this mechanism can help precisely understand what the DNN is considering and to what degree for specific input-output combinations. Finally, we can closely analyse the model's architecture and discuss the implementation of LSTM with attention in Pytorch:

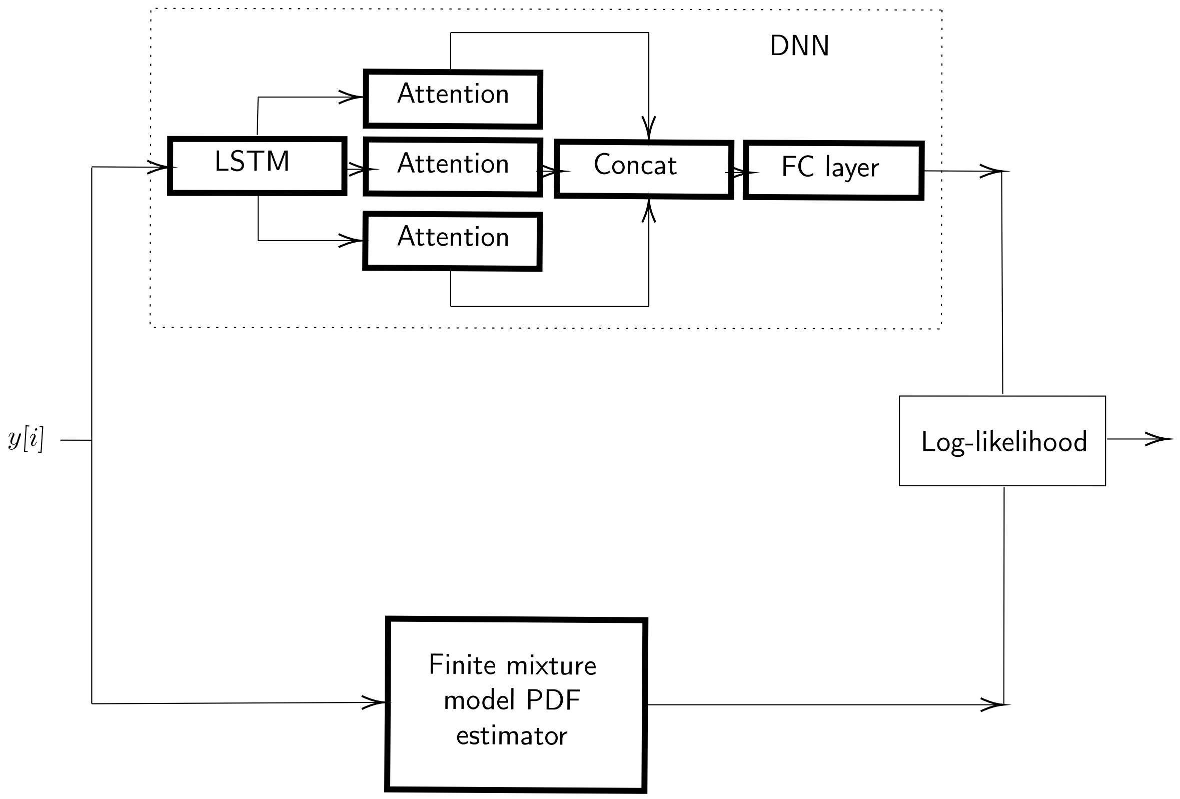

Figure 5.4: Attention + LSTM log-likelihood computation

The figure indicates that the outputs from the LSTM are fed into three different layers of attention. Moreover, the outputs from each attention layer are concatenated before feeding them into an output layer. To achieve the log-likelihood of a symbol, we must multiply our DNN estimates with the Gaussian-based mixture model.

The implementation of this model took some time to complete:

1. We initialise a 1x100 LSTM layer, where the outputs are fed into three attention layers. The attention layer we utilise in this section is the scaled dot product attention. This type

of layer requires a set of normalised attention weights multiplied by the outputs from the LSTM.

2. The resulting product is run through a softmax layer so that the probabilities add up to 1. These softmax estimates represent the attention distribution.

3. The generalised attention is computed by a weighted sum of the LSTMs output vectors.

This layer is applied three times to run through the attention mechanism. After that, the separate attention outputs are combined and fed into a fully-connected layer with a sigmoid activation function. To thoroughly examine the model, we may diagnose the following summary of the attention-based LSTM:

Layer (type:LSTM+Attention)	Output Shape	Param #
LSTM: 1-1	[5000, 1, 100]	41,200
Attention: 1-2	[5000, 100]	200
Attention: 1-3	[5000, 100]	200
Attention: 1-4	[5000, 100]	200
Linear: 1-5	[5000, 16]	1,616
Total params: 43,416		
Trainable params: 43,416		
Non-trainable params: 0		
Total mult-adds (M): 215.58		
Input size (MB): 0.02		
Forward/backward pass size (MB): 16.64		
Params size (MB) : 0.17		
Estimated Total Size (MB): 16.83		

Table 5.4: LSTM + Attention Model Summary

The total parameter numbers are slightly lower than the ViterbiNet, with a difference of 3,650 parameters. The number of parameters from the attention layers is low due to their lower computation requirements. The only computations that the layer has to go through are the multiplication of the LSTM inputs and the attention weight, which are fed through a softmax layer. As a result, the total multiplication/addition also reduces significantly to 215.58 million operations.

5.4.2 Multi-head Attention

The multi-head attention (MHA) implementation is based on the paper "Attention Is All You Need" [11]. However, the inputs are taken from three fully-connected layers (treated as query, key, value layers) instead of having the LSTM outputs from subsection 5.4.1 as the attention inputs. These layers perform a scaled dot product attention that outputs the normalised attention weights and their representation. This operation may often perform parallelly, depending on the number of heads we define. In addition, depending on the number of heads, the attention outputs are concatenated before feeding to an output fully-connected layer. Figure 5.5 below represents the architecture of MHA that computes the log-likelihood estimates for the Viterbi layer.

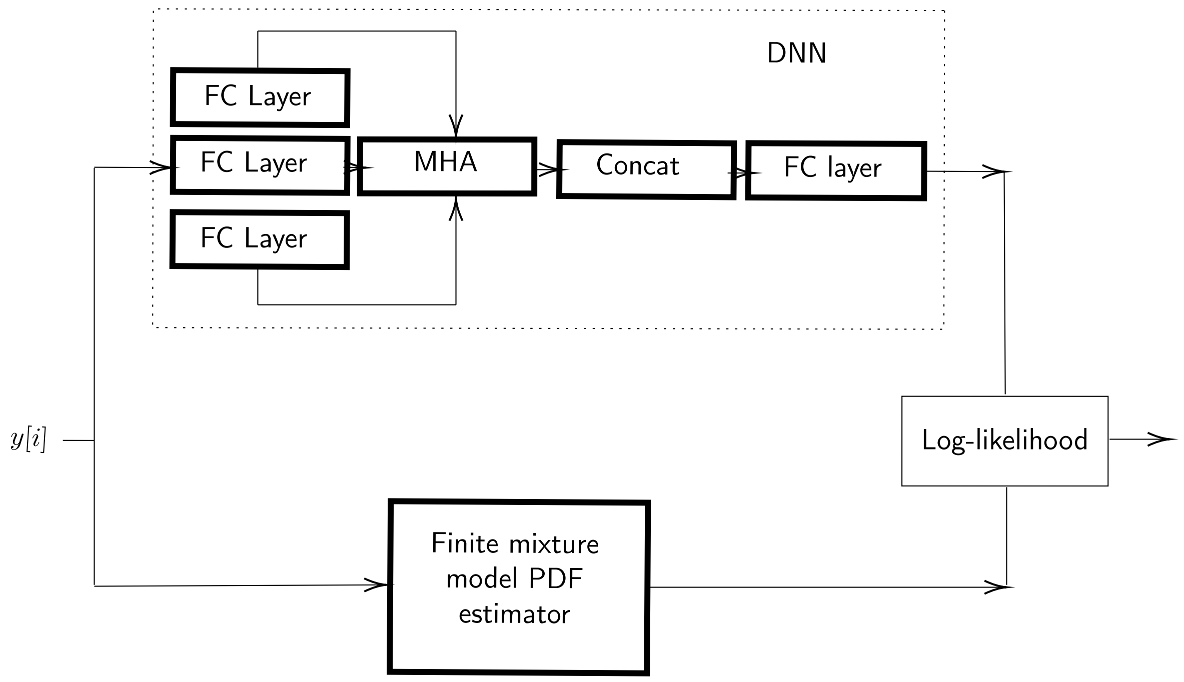

Figure 5.5: Multi Head Attention log-likelihood computation

The code to implement this feature is based on a Github repository [12]. It manually creates the attention layers and customises the concatenation function to add all the parallelised outputs to feed to an output layer. The process is very similar to the previous section.

Before starting the attention layer, the dot product of three fully-connected layers containing the channels query, key and value are required. In particular, we need to set the total dimensions of the model in these fully-connected layers. In this case, we set a dimension of 200. Moreover, these dot product tensors are split by the number of heads. In this case, set the number of heads to 5. Now, the attention computation begins:

1. Take the dot product of two inputs: the query with key-transpose to compute similarity.
2. Apply a masking function to prevent the attention mechanism of a transformer from leaking data in the decoder when training.
3. Pass these masked values through a softmax layer to compute probabilities and normalise scores.
4. The probabilities are multiplied by the value given in the FC layer.

Thus, the scaled dot product should provide a vector of attention estimates. Finally, these five attention outputs are further concatenated and fed through a final output layer with a sigmoid activation function. The summary of this model can be seen in table 5.5 below:

The total number of parameters increased significantly from section 5 by 80,400. In addition, the multiplication-addition contains 619.08 million operations. These operations mainly derive from the FC and attention layers by performing multiple dot product matrices mentioned earlier. Moreover, the model's memory is also high compared to the LSTM based attention mechanism. The multiple FC layers with 200x200 dimensions could result from the high computation of multiplication/addition calculations and high parameter numbers.

Layer (type:MultiHeadAttention)	Output Shape	Param #
Linear: 1-1	[5000, 1, 200]	40,200
Linear: 1-2	[5000, 1, 200]	40,200
Linear: 1-3	[5000, 1, 200]	40,200
ScaleDotProductAttention: 1-4	[5000, 5, 1, 40]	–
Softmax 2-1	[5000, 5, 1, 1]	–
Linear: 1-5	[5000, 16]	3,216
Total params: 123,816		
Trainable params: 123,816		
Non-trainable params: 0		
Total mult-adds (M): 619.08		
Input size (MB): 4.00		
Forward/backward pass size (MB): 24.64		
Params size (MB) : 0.50		
Estimated Total Size (MB): 29.14		

Table 5.5: Multi-head Attention Model Summary

5.5 Evaluation

The previous sections discuss the outline of the advanced DNN architecture in computing the log-likelihood computation. Moreover, the sections also examine each model's summary and discuss the total parameters, number of calculations, and memory required for the model to complete the training phase. In this section, we evaluate the performance of all discussed models by performing a Monte Carlo simulation of recovering $50,000$ symbols over unseen channel outputs. The graph that we will use to estimate the symbol error rate over the signal to noise ratio is semiology. In addition, we will present all model results (including the Viterbi and ViterbiNet as baselines) in one graph. Due to many implemented models, we split their performances into two figures. Furthermore, we will split this section into two subsections where one will focus on the performance of the features over a perfect channel, whereas the other will focus on the performance over the noisy channel.

5.5.1 Perfect Channel

Figure 5.6 compares the CNN, stacked GRU and LSTM + Attention against the two baseline models. The performance of SER indicates that all the models follow the same SER trend along with each SNR until $6dB$. It is also clear that the stacked GRU and CNN models outperform the ViterbiNet from SNR $8db$ and $10dB$. Thus, both models' performances almost reach the same as the iterative Viterbi. The high performance could be due to the increased parameters from table 5.3 and table 5.4. These high parameters are required when making accurate predictions. However, the LSTM + Attention model almost has the same performance as the ViterbiNet, with a slightly more significant error in SNR $10dB$. As mentioned earlier, these two models have almost the same parameters. Hence, it is expected for the models to perform similarly.

The second graph depicts the performance of both stacked LSTM and MHA against the baseline models. Similarly to figure 5.6, the performance of these models follows the same SER trend along with each SNR until $6dB$. However, there are slight deviations between the stacked LSTM and

MHA from $8-10dB$. For example, the LSTMs are closer to the Viterbi model, whereas the MHA has the same performance as the ViterbiNet. What is more, the parameters and calculations for the LSTMs have a smaller total number than MHA.

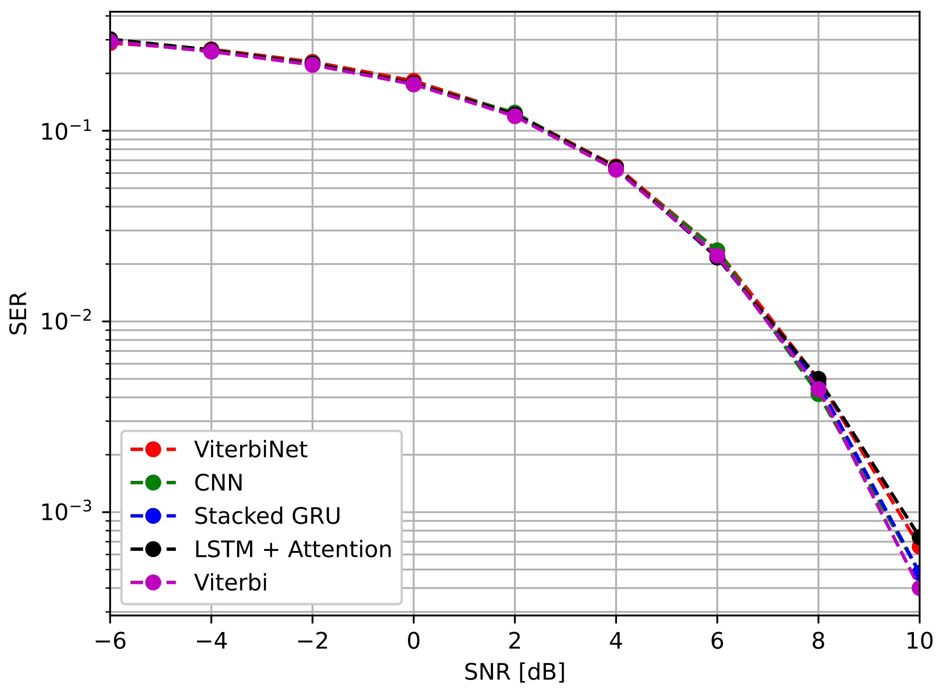

Figure 5.6: Viterbi vs DNN - Perfect Channel: Symbol Error Rate

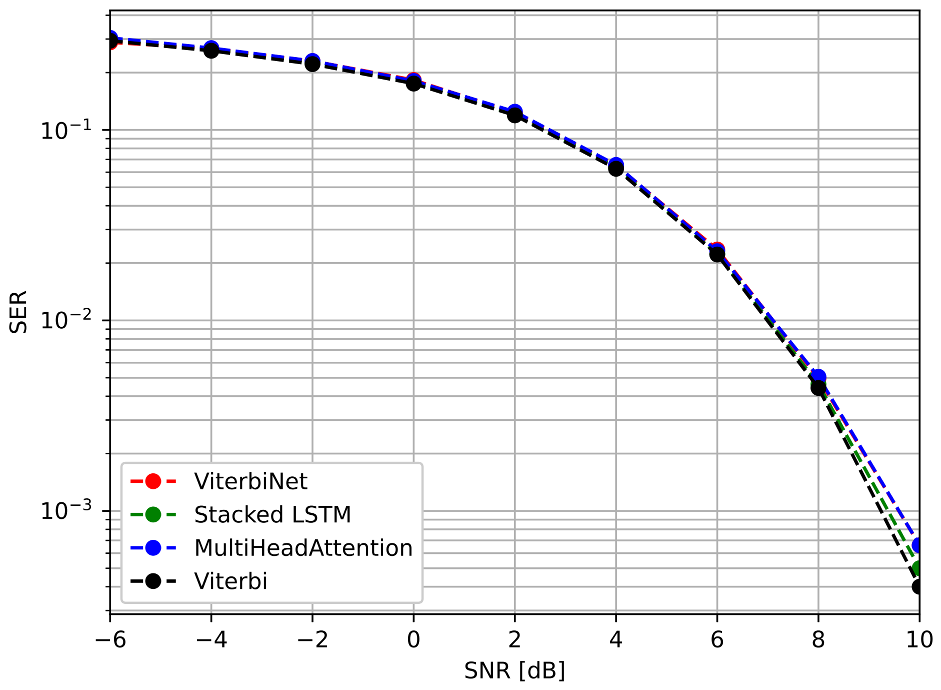

Figure 5.7: Viterbi vs DNN - Perfect Channel: Symbol Error Rate

5.5.2 Noisy Channel

Similarly to section 5.5.1, all the neural network-based Viterbi algorithms follow the same performance trend in the noisy channel. However, the conventional Viterbi model incorrectly predicts the symbols over each noisy SNR. The poor performance might be due to the CSI computation failing to produce a precise log-likelihood. In addition, the BPSK modulation of the noisy input signal and finite traceback blocklength might contribute to inaccurate log-likelihood computations. Consequently, the learning-based DNN has training data containing noisy channels and can identify patterns of noisy channels in each SNR. Hence, the accurate prediction of the log-likelihood estimates indicates that the DNNs were trained effectively.

Figure 5.8 compares the CNN, stacked GRU and LSTM + Attention against the two baseline models. The performance of SER indicates that all the models follow the same SER trend along with each SNR until $2dB$. After that, however, the ViterbiNet model seems to get more errors from $4-10dB$. The LSTM + Attention seems to follow the same SER trend as the CNN and stacked GRU until $6dB$, and then it more or less gets the same errors as the ViterbiNet. The result could be due to less training data provided for those particular SNRs so that the model can generalise the predictions effectively. Finally, it is evident in figure 5.8 that CNN outperforms all outlined models over noisy channels. It seems to have the lowest errors out of all models, particularly in $10dB$.

Figure 5.9 depicts the performance of both stacked LSTM and MHA against the baseline models. Similarly to the previous graph, the conventional Viterbi model predicts the symbols poorly over each SNR. However, notice how the multi-head attention more or less follows the same SER trend as the ViterbiNet model. Even though multi-head attention has a lot of trained parameters, it does not fall below the ViterbiNet baseline. There are three points of SNRs where the multi-head attention performs effectively. For example, the performance in $4dB - 8dB$ slightly outperforms the ViterbiNet. The better performance could be due to the model identifying more trends in those particular SNRs than ViterbiNet. Furthermore, the stacked LSTM has the best performance overall. It follows the same trend as the multi-head attention and ViterbiNet until $2dB$ and then has the lowest errors of all models until $10dB$.

This section considers the performance of both neural network-based and conventional Viterbi algorithms. Though the Viterbi itself outperforms the other models in the perfect channel conditions, it poorly performs when the channels are noisy. However, models such as the stacked GRU, CNN, and stacked LSTM come close to Viterbi's performance. In the next section, we will experiment with the performance of different noisy estimates.

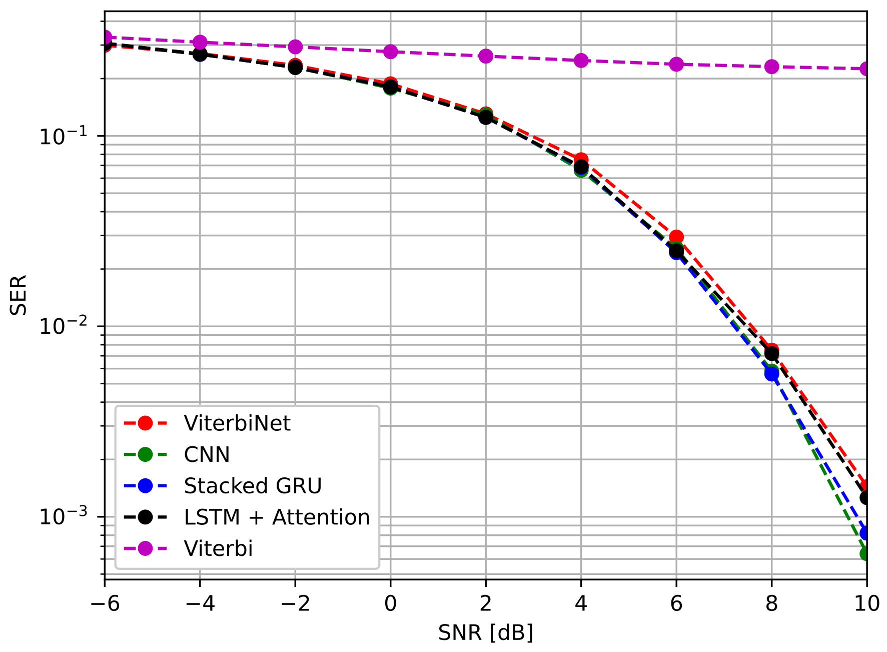

Figure 5.8: Viterbi vs DNN - Noisy Channel: Symbol Error Rate

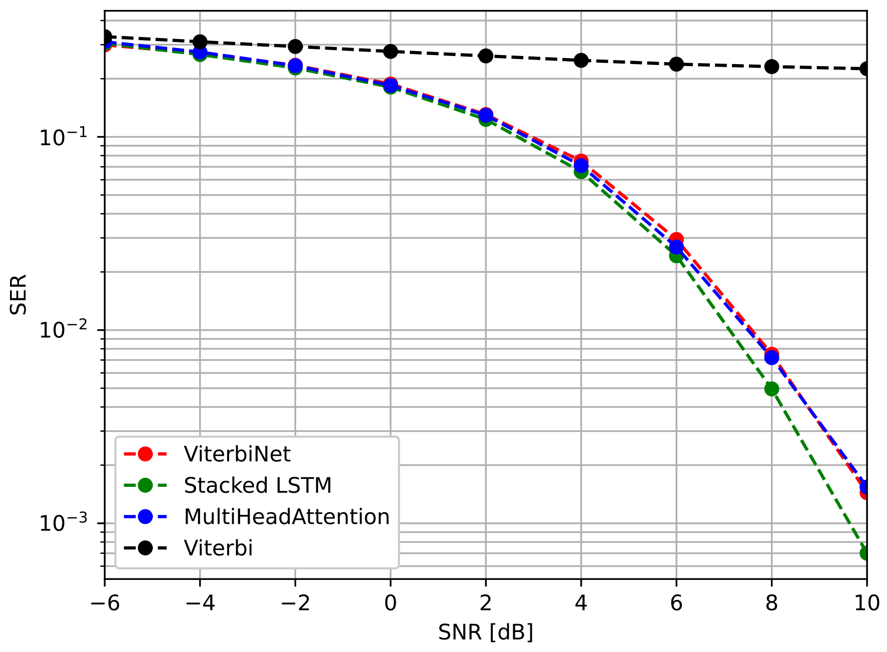

Figure 5.9: Viterbi vs DNN - Noisy Channel: Symbol Error Rate

5.6 Noisy Channel Alteration

This section explores an alternative approach to generating a noisy channel dataset. Shlezinger *et al.* have conducted a similar study called DeepSIC, the same concept as the ViterbiNet [13]. However, given that the Viterbi focuses on a particular input-output channel, DeepSIC specialises in recovering symbols over multiple input-output channels with the help of the soft interference cancellation (SIC) algorithm.

Shlezinger *et al.*'s numerical study introduces the same noisy channel dataset approach as section 4.1. However, they apply two different error variances: $\sigma^2 = 0.1$ (the same variance approach as section 4.1) and $\sigma^2 = 0.75$ [13]. The high variance indicates that the channels are thoroughly saturated with noise. Moreover, the high variance should deteriorate the performance of recovering symbols over different signals. This discussion will take the same error variance ($\sigma^2 = 0.75$) concept as DeepSIC and apply it to our Viterbi and ViterbiNet models mentioned in the core features of chapter 4. In addition, we will evaluate the performance of the two models over the perfect and noisy channel. We show the plots of the perfect channel conditions to analyse how far off the models are from the noisy channels.

Figure 5.10 indicates that the Viterbi performs even worse in noisy channels when $\sigma^2 = 0.75$ than $\sigma^2 = 0.1$. Consequently, the high variance constantly predicts inaccurate symbols over each SNR. In addition, the noisy ViterbiNet has slightly higher symbol errors in the earlier signals than the perfect channel. However, the errors gradually increase from $0dB - 8dB$ as there is a vast space between the green and red lines. The saturation of noisy channel conditions in ViterbiNet still outperforms the traditional Viterbi algorithm. More precisely, the model can more or less generalise the recovery of symbols effectively regardless of the error variance. Hence, the learning-based model is still more robust than the iterative Viterbi model.

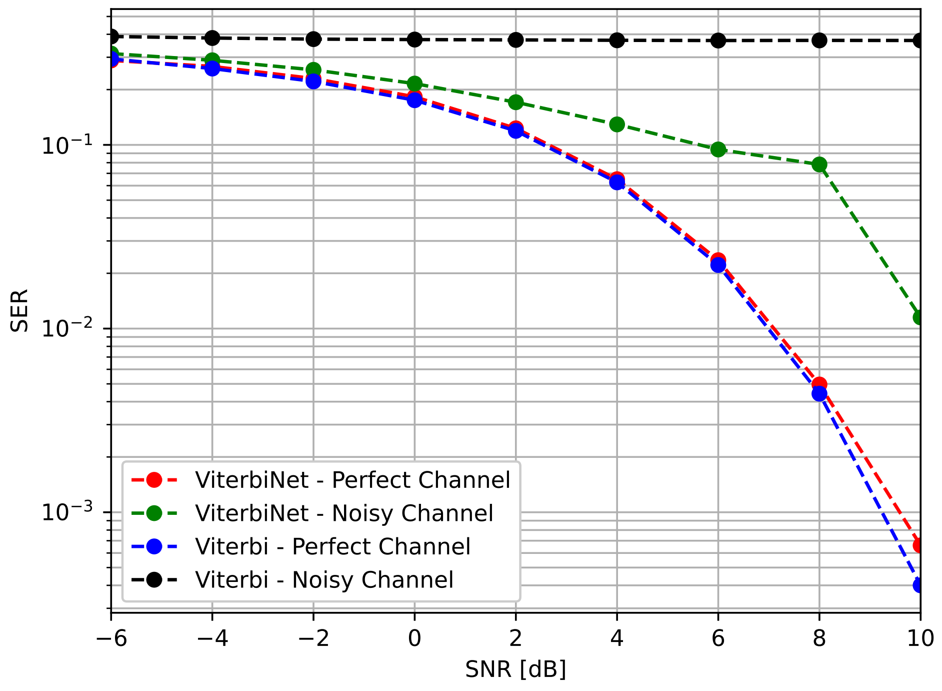

Figure 5.10: Viterbi vs ViterbiNet - Noisy Channel ($\sigma^2 = 0.75$): Symbol Error Rate

Chapter 6: Summary and Conclusions

We presented a brief review of digital communication systems and proposed a ViterbiNet model that replaces the receiver's CSI-based symbol recovery procedure with a deep neural network. Previously to the ViterbiNet, we used the log-likelihood computation as a part of the Viterbi algorithm, which needs complete knowledge of the underlying channel input-output statistical relationship. The approach is then reworked to include an DL-based framework for computing log-likelihood estimates. The final architecture combines deep learning with the Viterbi symbol detection algorithm. In summary, the ViterbiNet has the following characteristics:

- The ViterbiNet can learn from the data, which better matches the conventional Viterbi algorithm experienced by the communication system.
- The ViterbiNet architecture incorporates two factors. Firstly, a neural network topology consisting of an LSTM followed by two fully connected layers, where the final layer contains the softmax activation function. Secondly, the outputs from the network are multiplied with a mixture model based on an expectation maximisation fitting algorithm. As a result, we receive a log-likelihood estimate sent to the Viterbi algorithm.
- The simulation test results have validated that with using a small amount of training data, the performance of the ViterbiNet approaches the optimal performance of the CSI based Viterbi algorithm. Moreover, ViterbiNet can adapt to non-ideal conditions such as noise and thus significantly outperforms the traditional Viterbi.
- The efficient symbol detection over the channel's thorough noise saturation indicates ViterbiNet's robustness in the digital communication field.

In addition, we explore other deep neural network architectures that perform similar log-likelihood computations. The simulation results indicate that the stacked LSTM, GRU, and CNN outperform ViterbiNet. These topologies are closer to the optimal performance of the CSI-based Viterbi algorithm. On the other hand, the attention layers perform similarly to the ViterbiNet.

6.1 Possible Improvements

Although this project showcases a reproduction of ViterbiNet, new network topologies, and noisy channel changes, some minor changes can be considered to improve the project further. For example, we could reduce the number of training parameters in our advanced features to match the ViterbiNet. In addition, layers such as an average or max pool after a stacked LSTM and GRU could reduce the model's size and potentially match the numbers in ViterbiNet. Moreover, we could apply a pooling feature in the attention mechanisms before feeding them to an output layer. On the other hand, we could reduce the input-output dimension size of CNN to 1x16, 16x32, 32x64 and 64x16. These ideas can help reduce the model size while the performance probably diverges.

6.2 Future Work

Although ViterbiNet performs best with limited training samples, it may not work effectively if the communication system changes its configuration. Thus, the model might need to retrain samples based on the newly configured systems. However, the training phase will take much time and possibly be inefficient if these systems require constant reconfiguration. A solution to overcome such a problem is transfer learning. This concept ensures that a new ViterbiNet model learned on a newly configured system uses one or more layers from the pre-trained ViterbiNet model. Consequently, the ViterbiNet model's training time is significantly reduced, and the generalisation error is also reduced when transfer learning is used in this domain.

6.3 GitLab

The source code for the entire project is available on GitLab. In addition, the repository contains a structured directory of all neural network implementations, a channel-based dataset as Python scripts and performance figures of all models. Moreover, a Jupyter notebook contains only the network implementation to view the model summaries.

Acknowledgements

I would like to thank Dr Guénolé Silvestre, my project supervisor, for guiding me through my final year project. I would also like to express my gratitude to my friends and family for being a continual source of inspiration.

Bibliography

1. Shlezinger, N., Eldar, Y. C., Farsad, N. & Goldsmith, A. J. ViterbiNet: Symbol Detection Using a Deep Learning Based Viterbi Algorithm. *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)* (2019).
2. Viterbi, A. A personal history of the Viterbi algorithm. *IEEE Signal Processing Magazine* **23,** 120–142 (2006).
3. Forney, G. The viterbi algorithm. *Proceedings of the IEEE* **61,** 268–278 (1973).
4. Samuel, N., Diskin, T. & Wiesel, A. Deep Mimo Detection. *2017 IEEE 18th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)* (2017).
5. Farsad, N. & Goldsmith, A. Neural Network Detection of Data Sequences in Communication Systems. *IEEE Transactions on Signal Processing* **66,** 5663–5678 (2018).
6. Nachmani, E. *et al.* Deep Learning Methods for Improved Decoding of Linear Codes. *IEEE Journal of Selected Topics in Signal Processing* **12,** 119–131 (2018).
7. Wang, X.-A. & Wicker, S. An artificial neural net viterbi decoder. *IEEE Transactions on Communications* **44,** 165–171 (1996).
8. Moshirian, S., Ghadami, S. & Havaei, M. Blind Channel Equalization. *CoRR* **abs/1208.2205.** http://dblp.uni-trier.de/db/journals/corr/corr1208.html#abs-1208-2205 (2012).
9. Caciularu, A. & Burshtein, D. Blind channel equalization using variational autoencoders. *2018 IEEE International Conference on Communications Workshops (ICC Workshops)* (2018).
10. Luo, R. *et al. A radio signal modulation recognition algorithm based on residual networks and attention mechanisms* Sept. 2019. https://arxiv.org/abs/1909.12472.
11. Vaswani, A. *et al. Attention Is All You Need* Dec. 2017. https://arxiv.org/pdf/1706.03762.pdf..
12. Hyunwoongko. *Hyunwoongko/Transformer: "Attention is all you need"* https://github.com/hyunwoongko/transformer.
13. Shlezinger, N., Fu, R. & Eldar, Y. C. DeepSIC: Deep soft interference cancellation for multiuser MIMO detection. *IEEE Transactions on Wireless Communications* **20,** 1349–1362 (2021).