AN ENCODER-DECODER BASED BASECALLER FOR NANOPORE DNA SEQUENCING

MAHDIEH ABBASZADEGAN

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
FEB 2019

©MAHDIEH ABBASZADEGAN, 2019

Abstract

Nanopore DNA sequencing is a method in which DNA bases are determined (base-called) using electric current signals generated by passing DNA through nanopore sensors. The raw measured signals can be aggregated into event data presenting new bases entering the nanopore. This thesis has two contributions. First, we implemented RNN-based single- and double-strand basecallers for simulated event data to analyze the effect of signal noise. As the SNR decreased from 20 dB to 5 dB, the accuracy of the single-strand basecaller dropped 9% while the accuracy of double-strand basecaller only dropped 0.5%. Second, we implemented an end-to-end single-strand basecaller, directly processing the raw signal using an encoder-decoder model with attention instead of the CTC-style approach used in available basecallers. We achieved an accuracy of 81.9% for a viral sample and an accuracy of 90.9% for a bacterial sample. Our accuracy is comparable to state-of-the-art basecallers with a considerably smaller model.

Table of Contents

List of Tables

List of Figures

Abbreviations

ANN	Artificial Neural Network
BPTT	Back-propagation Through Time
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CTC	Connectionist Temporal Classification
DC	Direct Current
DNA	Deoxyribonucleic Acid
FNN	Feed-forward Neural Network
GPU	Graphics Processing Unit
GRU	Gated Recurrent Units
HMM	Hidden Markov Model
LSTM	Long Short-Term Memory
NCBI	National Center for Biotechnology Information
NGS	Next-generation Sequencing

NMT Neural Machine Translation

ONT Oxford Nanopore Technologies

PacBio Pacific Biosciences Sequencers

PCR Polymerase Chain Reaction

ReLU Rectified Linear Unit

RNN Recurrent Neural Network

SGD Stochastic Gradient Descent

SGS Second-generation Sequencers

SMRT Single Molecule Real-Time

SNR Signal-to-noise Ratio

ZMW Zero-Mode Waveguides

1 Introduction

1.1 Introduction and Motivation

DNA or deoxyribonucleic acid is an inherited molecule that has the unique genetic code of each organism. DNA is formed by chemical components called nucleotides that are made of three groups: a base, a sugar and a phosphate. The four nucleotide bases of a DNA are Adenine (A), Guanine (G), Cytosine (C) and Thymine (T). DNA nucleotides pair with their complement (A-T, C-G) and build two joined long strands with a spiral form called the double helix. The sequence of bases and their order contain the information on developing and maintaining the organism.

Study of DNA is valuable for microbiology, plant and animal research and many other applications in medicine and the life sciences. Human genome sequencing has helped in monitoring cancer and other diseases.

DNA sequencing is the process by which DNA molecules are measured utilizing DNA sequencers. Subsequently, the raw measurements are reconstructed with algorithms into summary equivalents, such as the text representation of the exact order

of nucleotides (bases) in a DNA molecule. Typically, this text equivalent consists of the 4-element alphabet which denotes the first letter of the chemical name for each possible base (A, C, G and T).

DNA sequencers were introduced in the mid-70's [4]. Due to the limitations of their sensory modalities, DNA sequencers only operate on fragments of DNA samples, perhaps 100-200 bases in length in established sequencing machines. The text-equivalent of these segments is first determined via a process referred to as "basecalling", followed by the application of ensuing bioinformatics algorithms to re-assemble the aforementioned fragments into the total genome or some biologically relevant subset of such. This reconstruction effort is computationally very expensive.

DNA sequencers have undergone profound improvements in processing speed, cost and read length in three generations. The first generation of DNA sequencers are large, time-consuming and expensive instruments. Second generation DNA sequencers are cheaper and faster by processing multiple short reads in parallel.

Recently, third-generation DNA sequencers have been released that are portable, more affordable and work in real-time. The two leading companies in this domain are Oxford Nanopore Technologies (ONT) and Pacific Biosciences sequencers (PacBio). PacBio [5] uses SMRT (Single Molecule Real-Time) and light detection technology while ONT uses nanopores and electrical current for sequencing. This

thesis focuses on nanopore-based sequencing.

Nanopores are very small holes made of proteins arranged as an array (of thousands) in ONT devices. A DC voltage is applied across nanopores, resulting in a DC electrical current flowing in the nanopore. Translocation is the process of passing strands of DNA through the nanopore. The current alterations generated by a DNA molecule translocation, can be used to determine the base makeup of the DNA.

The MinION is a particular palm-sized portable third-generation sequencer from ONT that can operate on DNA samples at least 100X longer than traditional sequencers, a property with the promise of greatly easing the aforementioned reconstruction challenge [6]. Perhaps even more importantly, the MinION, via its size and real-time operation, offers the possibility of ubiquitous sequencing, bringing a traditionally resource-limited diagnostic (i.e. DNA sequencing) to point-of-care and environmental sensing scenarios. However, due to its output signal's relatively poor signal-to-noise ratio (SNR), the MinION suffers from poor accuracy during its basecalling stage. To endow this device with the necessary accuracy our motivation is to reduce the MinION's basecalling error rate via improved algorithms.

In essence, basecalling is a sequence-to-sequence mapping problem. The basecaller's input data is a sequence of electronic current signals generated by translocation of DNA through a nanopore sensor and its output is a sequence of equivalent

molecule labels (i.e. the bases: A, C, G, T). Single-strand basecallers process the input data of only one strand of DNA. And double-strand basecallers process the input data of both strands of DNA comprising the double-helix structure.

Basecallers are divided into raw and event basecallers based on their input data. Raw basecallers directly perform on the current signal generated from MinION. For event basecalling, the raw data is first processed by an event detection algorithm and then the event data is used for basecalling. Each event should present a new base entering the nanopore. However, the event detection process is error-prone and may result in events presenting less or more than one base.

In event basecalling, the input and output sequences have the same length and their alignment is known. In this special sequence-to-sequence mapping case, a Recurrent Neural Network (RNN) can model the problem. An RNN is a type of neural network that is used for processing time-series data.

In raw basecalling, the length of the input and output sequences are not the same (and their alignment is unknown) as each base is the output of a variable number of current input signals. Thus, sequence-to-sequence mapping techniques should be used. The sequence-to-sequence models include connectionist temporal classification (CTC), the encoder-decoder model and the attention-based encoder-decoder model.

1.2 Contributions

The low SNR of the nanopore current signal is the major cause for the low accuracy of nanopore-based basecalling. To investigate the effect of noise and analyze it on single- and double-strand basecalling, we use approximated nanopore characteristics to simulate nanopore-based event data. Each simulated event signal is affected by three bases (3-mer) passing the nanopore. We impose different input SNR conditions upon the simulated data.

We implemented RNN-based single- and double-strand basecallers for the simulated event data. We study the behavior of our basecallers as a function of the time-series SNR.

A number of basecallers have been developed for raw nanopore data by ONT and individual researchers. A CTC-style approach is used in the implementation of all of these basecallers. Using the attention-based encoder-decoder model has shown a great improvement in end-to-end speech recognition compared to a CTC-style approach [7, 8]. This thesis implements and evaluates an open source raw basecaller using the encoder-decoder model with attention.

1.3 Overview of Thesis

Chapter 2 provides the background of machine learning models and deep learning concepts used for building and training basecallers. In Chapter 3, the background of DNA sequencing in general and third-generation sequencing, in particular are reviewed. Available basecallers for nanopore sequencing data are then discussed and compared. In Chapter 4, we propose an event data simulation process and implement single- and double-strand basecallers for the simulated data. We investigate the accuracy of our single- and double-strand basecallers for simulated data with different noise levels. Chapter 5 presents an encoder-decoder based single-strand basecaller for raw nanopore data. Conclusions and future work are presented in Chapter 6.

2 Machine Learning Models

In this chapter, we review the fundamentals of machine learning models used for the basecalling problem. In basecalling, the input sequence is a time-series signal generated by nanopore sensors and the output is a sequence of DNA bases.

As a sequence classification problem, appropriate machine learning approaches for basecalling include Hidden Markov Models (HMMs), Recurrent Neural Networks (RNNs) and temporal Convolutional Neural Networks (CNNs).

In section 2.1, we review different types of neural networks including Feed-forward Neural Networks (FNN), RNN, CNN and the algorithms used for training these networks. We also discuss sequence-to-sequence mapping techniques such as Connectionist Temporal Classification (CTC), encoder-decoder models and attention mechanisms. In section 2.2 we briefly introduce Hidden Markov Models, discussing elements of an HMM, the training step and the decoding step.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are machine learning models inspired by the biological neural networks in the human brain. A neural network can be used for nonlinear classification and regression problems. Three main types of neural networks are feed-forward neural networks, convolutional neural networks and recurrent neural networks.

2.1.1 Feed-forward neural network

The feed-forward neural network is the simplest type of artificial neural network and is built by layers of interconnected neurons. A neuron or node or unit pictured in Fig. 2.1 is the smallest structure comprising neural networks. Each node has a set of weighted input connections. The neuron weights (w_n) are the parameters of the network and are learned by a training process. The sum of weighted inputs are subject to a nonlinear transformation, a so-called activation function.

A feed-forward neural network, as shown in Fig. 2.2, has an input layer, an output layer and can have one or more hidden layers. Feed-forward neural networks don't have loops and information flows only from the input to the output layer.

A feed-forward network with only input and output layers is called a single-layer perceptron. And a feed-forward network with at least three layers (input

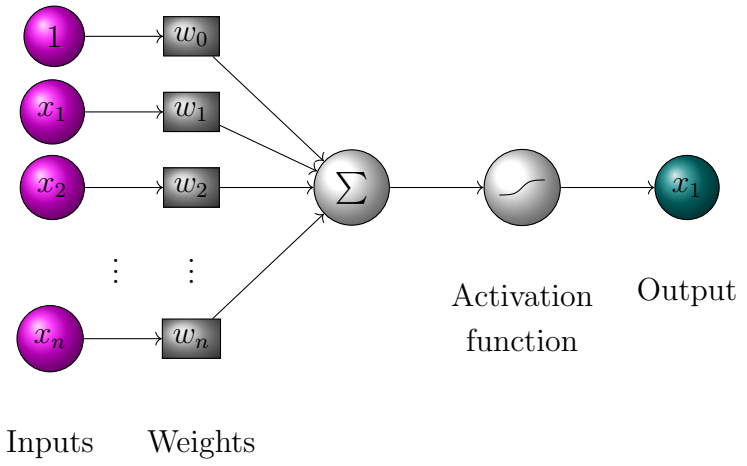


Figure 2.1: A single neuron used for building neural networks.

layer, hidden layer and output layer) is called a multilayer perceptron. In a fully-connected or dense layer (the most usual type of NN layer), all of the nodes in adjacent layers are connected.

2.1.1.1 Activation Function

The activation function imposes nonlinearity to the output of a node. Well-known activation functions include Sigmoid, Tanh, ReLU and Softmax.

Softmax Softmax is usually used for the output layer of classification problems with mutually exclusive classes. Each output is in the range $[0, 1]$, and the sum of all the outputs is 1. So, the output may be interpreted as the probability of each class being the correct output.

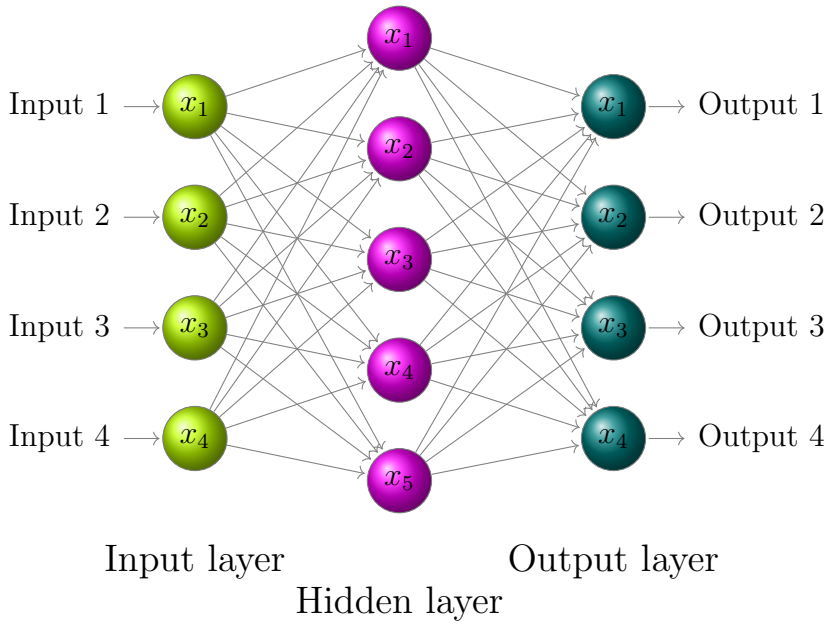
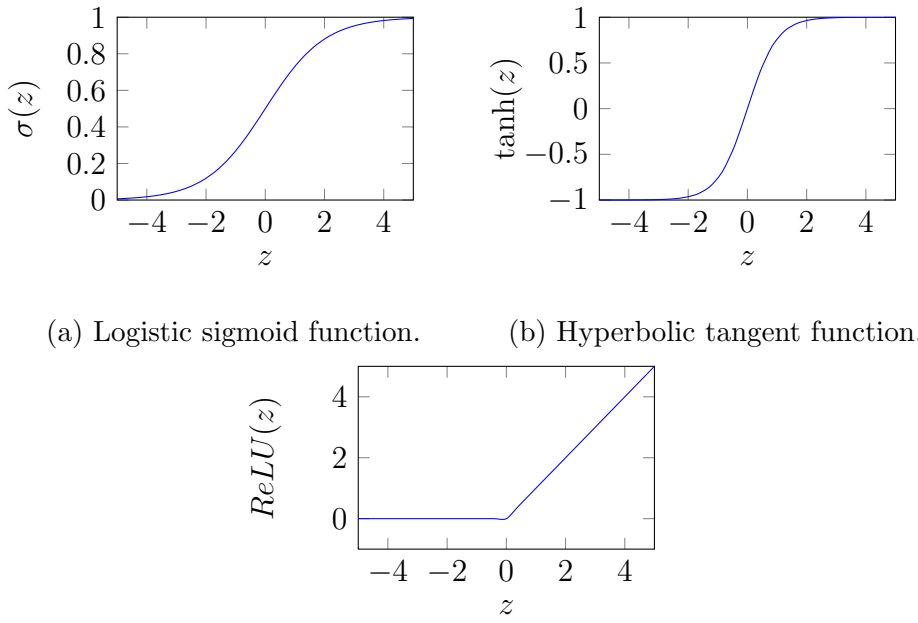


Figure 2.2: A feed-forward neural network with an input layer, a hidden layer and an output layer.

Sigmoid The Sigmoid function (σ) is shown in Fig.2.3a and its mathematical expression is $\sigma_x = \frac{1}{1+e^{-x}}$. The Sigmoid compresses the value of each output to the range $[0, 1]$, with no constraint on the sum of the outputs. So, it can be used for non mutually exclusive classification problems or binary classification.

Rectified Linear Unit Most recent neural networks use the Rectified Linear Unit (ReLU) activation function. ReLU has the mathematical form $ReLU_x = max(x, 0)$. When the input is smaller than 0, the output is 0. And when the input is 0 or larger, the output is the same as the input. The graph of the ReLU function



(a) Logistic sigmoid function.

(b) Hyperbolic tangent function.

(c) ReLU function.

Figure 2.3: Common used activation functions.

is shown in Fig. 2.3c.

Tanh The Tanh activation function is similar to the Sigmoid function, but its range is $[-1, 1]$. Mathematically, this function is expressed as $\tanh_x = \frac{2}{1+e^{-2*x}} - 1$. The graph of the Tanh function is shown in Fig. 2.3b.

2.1.2 Training

In a supervised learning problem, the network is trained (i.e. the weights are learned) using a set of correctly labeled data, the so-called training data. Each

training input data is labeled with the correct target value. In a regression problem, the target value is a continuous value. But in a classification problem, the target is a class label from a discrete number of possible classes. As basecalling is a classification problem, we will focus on classification techniques.

A neural network (i.e. its weight parameters) is first initialized by small random numbers, and at each step of the training, the error of the network is calculated using the "Loss function" explained in 2.1.2.1. Back-propagation is the process of associating the error with the preceding network nodes and fixing their weights to reduce it. An optimization algorithm is used to adjust the parameters based on the calculated error.

2.1.2.1 Loss Function

The loss function is a mathematical function used to determine the error of a neural network compared to the correct output. A loss function gives a measure of how well our neural network is working. When all of the outputs are correct, the loss value is 0. The loss is calculated based on the learnable parameters (w) of the network. An error surface is an $(n + 1)$-dimensional surface that presents error of a network depending on its n parameters. Fig. 2.4 shows the error surface of a network with two parameters. The minimum point on this surface shows the optimal set of parameters of the network.

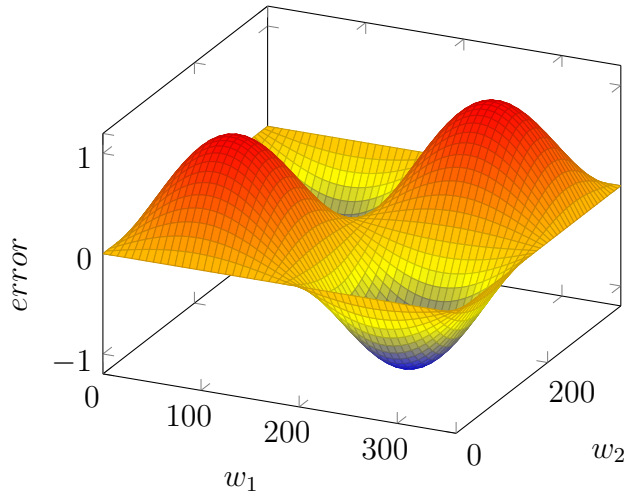


Figure 2.4: Error surface of a neural network with two units.

Cross-entropy or log loss is a popular loss function used for classification problems. Binary cross-entropy is used for binary classification and categorical cross-entropy is used for multiclass classification. Basecalling is a multiclass classification problem.

2.1.2.2 Optimization Algorithms

An optimization algorithm is used to adjust the parameters of the network based on the loss function. The goal of an optimizer is to minimize the loss function, finding the best set of network parameters. The error surface can have multiple local minimums in addition to a global minimum and a number of saddle points as pictured in Fig. 2.4.

Some common optimizers are:

Gradient Descent Gradient descent is the most popular optimizer. As the name suggests, we calculate the gradient of the loss function and at each time we move to the direction of (negative of) the gradient proportional to the learning rate.

The learning rate is a hyperparameter that controls the size of our steps in each parameter update. If the learning rate is small, we take smaller steps in the direction of (negative of) the gradient, so we move with higher precision but consequently may take a long time to converge on an error minimum.. There is a trade-off in choosing the learning rate and it usually should decrease as we move closer to the minimum.

In gradient descent, all of the training data is used for calculating the gradient for each weight update and this makes the algorithm very slow.

Stochastic Gradient Descent In Stochastic Gradient Descent (SGD), the network weights are updated for each training data, instead of observing all of the data as in gradient descent. SGD is much faster than gradient descent, but has a very high parameter update variance, changing the directions quickly and making the convergence unstable.

Mini-Batch Gradient Descent In mini-batch gradient descent, a constant number of samples called a batch is used for calculating the gradient. This algorithm works better than both gradient descent and SGD algorithms.

Momentum The momentum technique proposes a new approach on updating the network parameters. In this technique, instead of moving in the direction of the gradient and updating the position of network parameters on the error surface, the gradient is used for calculating the velocity. The position is then adjusted based on this velocity. Nesterov momentum is a newer version of momentum. The position is first adjusted based on the velocity, and at each step the gradient is calculated in the updated position instead of the old position as in momentum.

AdaGrad The previously mentioned optimizers, change all of the parameters by the same amount. AdaGrad [9] presents an adaptive learning rate for each parameter. An accumulated sum of squared gradients is tracked for each parameter and is used to calculate the learning rate (learning rate is divided by the square root of this accumulated sum). The learning rate is always decreasing in this algorithm as the gradient sum is always increasing, so the training may stop too early.

RMSProp RMSProp presented by Geoffry Hinton [10] solves the monotonically decreasing learning rate of AdaGrad. RMSProp calculates an exponentially moving

average of the gradients for updating the learning rate.

Adam Adam (adaptive moment estimation) [11] also implements an adaptive learning rate for each parameter. Adam is very similar to RMSProp but also implements Momentum [12]. Adam is the recommended default optimizer.

2.1.2.3 Generalization

Generalization of a model refers to how well the model acts for new unobserved data with the same distribution as the training data. For evaluating generalization, we divide the available labeled data into training and validation data. Only the training data is used to learn the model parameters. We then measure the loss and accuracy of both training and validation sets after each epoch. An epoch is completed when the network has observed all of the training samples once.

Generalization curves show loss and accuracy of both training and validation sets after each epoch. The model is "overfitting" when it works well on the training data, but performs poorly on the validation data. The model is "underfitting" when it performs poorly on both training data and validation data. A good model has high accuracy and low loss for both training and validation data sets.

2.1.2.4 Regularization

Regularization is used to avoid overfitting. The three most popular regularization methods are the L1 norm, L2 norm and dropout.

L1 and L2 regularizations add a regularization term (or penalty term) to the loss function, preventing the parameters from becoming too large and this way they avoid overfitting. L2 regularization adds the squared magnitude of all the parameters (multiplied by a regularization strength λ) to the loss function but L1 regularization adds the absolute magnitude of the parameters (multiplied by a regularization strength λ) to the loss function [12].

Dropout [13] is a recent regularization technique and works very efficiently. In this method, network inputs are randomly set to 0 in the training process. Fig. 2.5 represents a visualization of dropout.

2.1.3 Recurrent Neural Networks

Recurrent neural networks (RNN) are a generalization of the feed-forward neural networks that consider the previously observed data as part of their decision making (inference) process. RNNs pictured in Fig. 2.6 possess feedback loops across their neuron structures effectively providing a memory of the already seen data. Thus, the hidden state of the network has the information of previous inputs. RNNs can

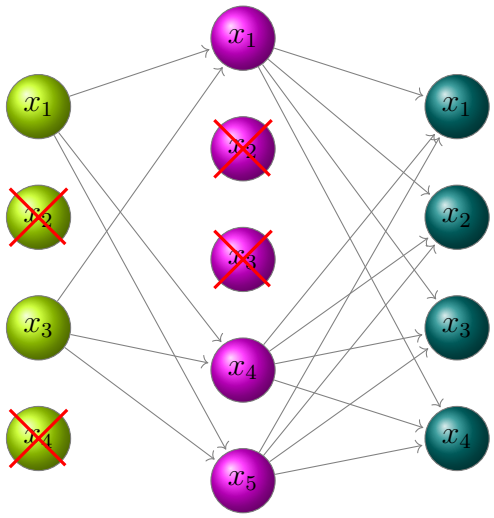


Figure 2.5: Illustration of the dropout technique.

process variable length inputs and are powerful models for sequence data analysis.

We can unfold an RNN through time as shown in Fig. 2.7 to better see the flow of information. The network looks like a feed-forward neural network but as the same unit is expanded through time, all of these units have the same parameters. Back-propagation through time (BPTT) is used for training a recurrent neural network. In this process, the regular back-propagation is applied on the unfolded network with this constraint that the parameters of the network are the same in all time-steps.

Different RNN structures such as simple RNN, LSTM, GRU and bidirectional RNN units have been developed. We describe these four RNN types in the following sections.

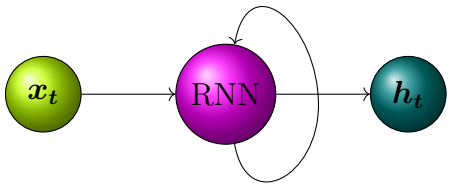


Figure 2.6: A recurrent neural network.

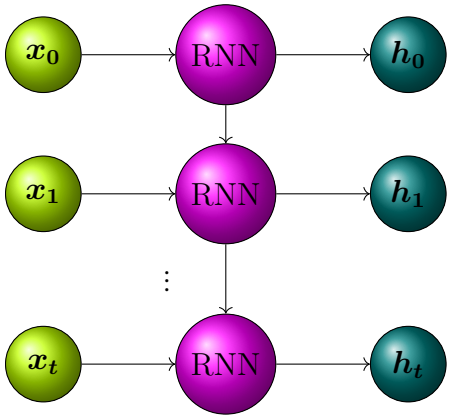


Figure 2.7: Illustration of unfolding an RNN through time.

2.1.3.1 Simple RNN

The simple RNN presented in Fig. 2.8, uses a $tanh$ function to compute the hidden state. The hidden state (h_t) is the output of the unit and is computed as $h_t = tanh(W \cdot [h_{t-1}, x_t])$. W is the learnable parameter of the unit, x_t is the network input and h_{t-1} is the hidden state of the previous time-step.

Training a simple RNN is difficult due to the vanishing gradient problem [14]. The vanishing gradient problem occurs in training neural networks using gradient-based optimization algorithms. By multiplying the same small weights in back-propagation through time, the product becomes so small that it vanishes over multiple iterations. Simple RNN also lacks the ability to model long-term dependencies in the inputs of a sequence.

2.1.3.2 Long-Short Term Memory

The Long Short-Term Memory method presented by Hochreiter and Schmidhuber [15] is a popular approach for solving the vanishing gradient problem. This structure presented in Fig. 2.9 has three gates and an internal memory cell instead of a single tanh unit as used in a simple RNN. The LSTM is capable of modeling long-term dependencies.

Gates control the flow of information in and out of memory. The memory cell

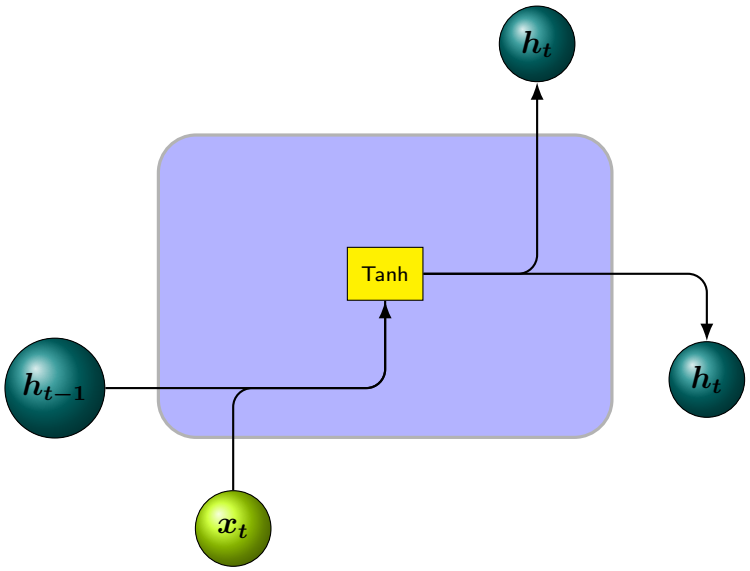


Figure 2.8: Illustration of a simple RNN node.

(C) is the internal memory of the LSTM. The hidden state (h) is the hidden output of the node. The forget gate (f) controls the amount of previous memory (C) that we forget. The input gate (i) controls how much the new cell is updated by the cell candidate ($\tilde{C}$). And the output gate (o) determines how much the cell state (after going through a $tanh$) is passed as the output (h).

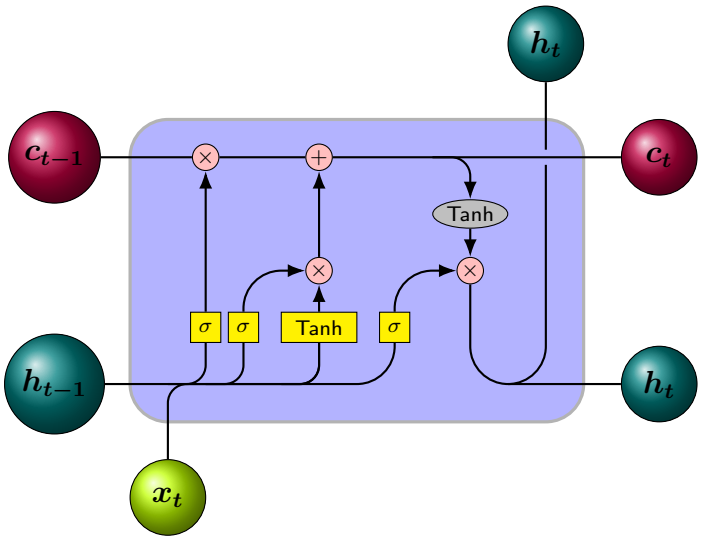


Figure 2.9: Illustration of an LSTM node (adapted from [2])

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * tanh(C_t)$$

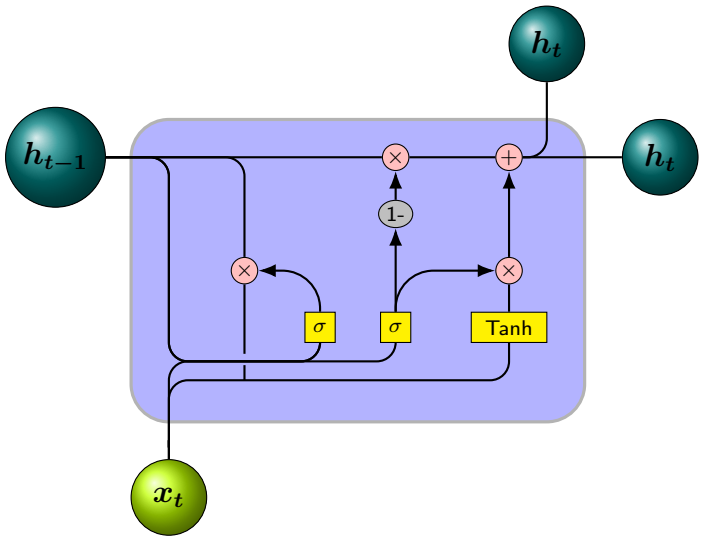


Figure 2.10: Illustration of a GRU node (adapted from [2])

2.1.3.3 Gated Recurrent Units

The so-called Gated Recurrent Unit (GRU) is the most popular variation of the LSTM. It is simpler in structure as it has only two unit gates (update and reset) and no separate memory cell (C). Forget and input gates normally present in the LSTM are mixed into the update gate of the GRU and the reset gate is similar to the LSTM output gate. A GRU unit is presented in Fig. 2.10.

In an empirical study on polyphonic music modeling and speech signal modeling, Chung *et al.* [16] showed that the GRU is comparable to the LSTM and with the same number of parameters, GRU performs better than LSTM both in training time and generalization. LSTM and GRU can be chosen based on each task and dataset.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h}_t = tanh(W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

2.1.3.4 Bidirectional RNN

In a sequential problem, the output of a time-step can depend on both previous and future time-steps. In a bidirectional RNN presented in Fig. 2.11, the output of each input is connected to two networks. The first network is trained on the original sequence data and the second network is trained with the inverse sequence data. All three previously mentioned RNN versions can be extended to a bidirectional RNN and can learn some problems faster and better than regular RNNs.

2.1.4 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a form of feed-forward neural network and employ the convolution operation. Convolutional neural networks are mainly used for computer vision problems but they also perform well on sequencing problems. A simple ConvNet pictured in Fig. 2.12 has convolutional layers, pooling

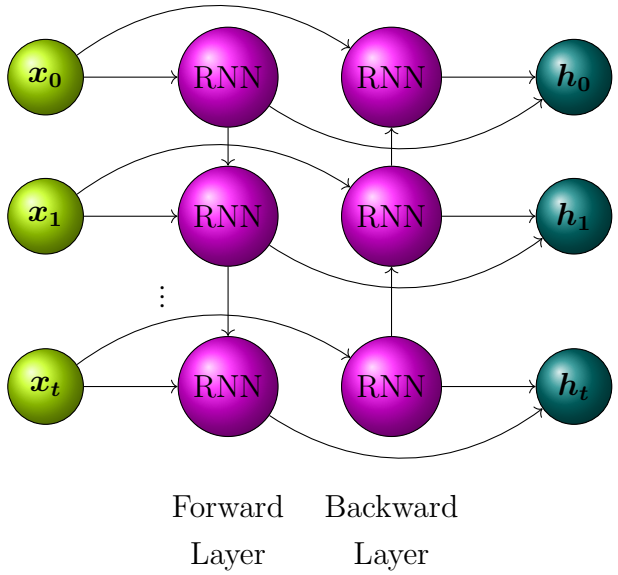


Figure 2.11: Illustration of a bidirectional recurrent neural network.

layers and fully connected (dense) layers.

2.1.4.1 Convolutional Layer

The convolutional layer is the main layer in CNN. Each convolutional layer has a number of small windows called filters or kernels that convolve on the input data. In the convolving process, the dot product of the filter parameters and the input values in that region are calculated. Usually, the ReLU activation function is applied to the output of the convolutional layer. Each filter has different learnable weights and results in different outputs (called feature maps) on the same input data.

Hyperparameters of a convolutional layer are the filter size (or receptive field),

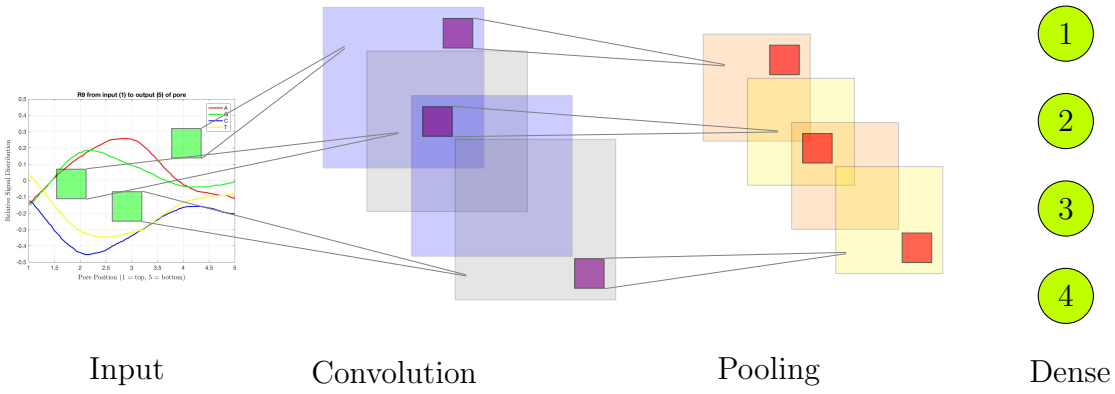


Figure 2.12: A Convolutional Neural Network for image input.

the stride and the type of padding. The stride value specifies the amount by which we slide the filter on the input. If the stride is 1, we move the filter one pixel at a time and if the stride is 2, we move the filter by two pixels at each time. When convolving a filter on an input, the feature map is smaller than the input. By padding the input by a default value (zeros) around the borders, we can preserve the size of the input after convolution. Two common types of padding are valid and same. The valid padding means no padding, and in the same padding, the input is padded in a way that the output of the convolutional layer has the same dimension as the input.

1D convolutional layer A 1D or temporal convolution layer is used for 1D sequence inputs such as speech data, temperature time-series data and the nanopore sequencing data. A temporal convolutional layer is especially useful for extracting

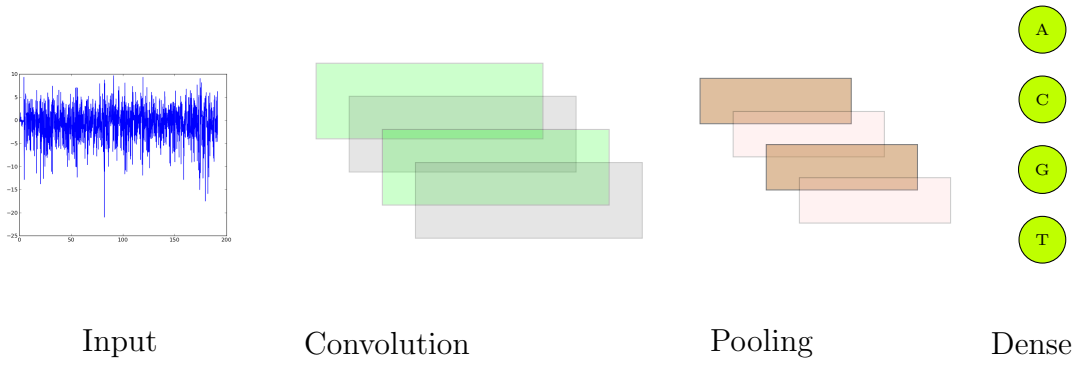


Figure 2.13: A Convolutional Neural Network for signal input.

data features from an input sequence. In this structure, filter size is a number defining the length of the 1D kernel window. Fig. 2.13 presents a 1D convolutional neural network with a 1D convolutional layer, a pooling layer and a fully connected Softmax layer.

2D convolutional layer A 2D or spatial convolutional layer is the most popular type of convolutional layers and is used for 2-dimensional input data like an image. The kernel window is 2-dimensional and the filter size is a list of two numbers defining the height and width of the window. In the case of image input, different filters represent different image processing techniques such as edge finding and blurring.

3D convolutional layer A 3D or volumetric convolutional layer is used for 3-dimensional input data such as video or 3D images. The kernel window is 3D and

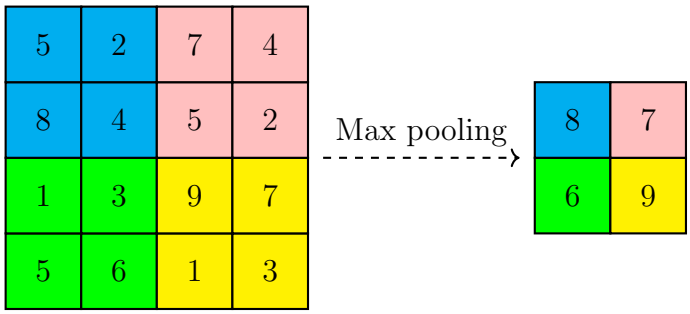


Figure 2.14: Illustration of a max pooling layer with pool size (2,2).

the filter size is a list of three numbers defining the depth, height and width of the kernel window. Each filter convolves on the input data in three dimensions to produce the feature maps for the next layer.

2.1.4.2 Pooling Layer

A pooling layer performs a downsampling operation [17]. By using a pooling layer, we only keep important information from the previous layer and reduce the size. The pooling layer doesn't have learnable parameters but has a set of hyperparameters that should be assigned. These hyperparameters are pool size, padding and stride.

A pooling layer can perform a max, an average or a sum operation [17]. The max pooling layer presented in Fig. 2.14 down-samples the input by only keeping the max value in the pool window. The output dimension depends on the pool size, padding and stride.

2.1.5 Sequence-to-sequence Mapping

In sequence-to-sequence (seq2seq) learning, an input sequence from one domain is translated to an output sequence in another domain [18]. In a particular case that each input time-step is aligned to an output time-step (input and output sequences have same lengths), a recurrent neural network or a 1-dimensional convolutional neural network can be trained to model the problem.

In cases that input and output sequences have different lengths and we don't know their exact alignments such as speech recognition and Neural Machine Translation (NMT), a regular neural network requires much more complexity to model the problem. The sequence-to-sequence models include connectionist temporal classification (CTC), the encoder-decoder model and the attention-based encoder-decoder model.

2.1.5.1 Connectionist Temporal Classification

The Connectionist Temporal Classification (CTC) algorithm proposed by Graves et al. [19] is an approach for dealing with unsegmented label data. With unsegmented label data, we have the output (label) sequence represented by an input sequence but the exact alignments are unknown. An example is an input sequence from the speech domain and an output sequence from the text domain as shown in Fig. 2.15.

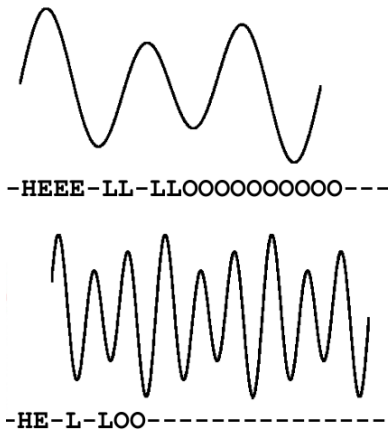


Figure 2.15: Two voice signals representing the Hello text (adapted from [3]).

We have both the speech signal and the text (Hello), but not the exact alignment of the two. Some parts of the input signal may not represent any characters. The CTC labels these parts by a "blank" token.

A CTC is a scoring function and is used after the Softmax output layer of the network. The Softmax layer has an extra unit for a "blank" class. The blank class or token is used for segmenting the output sequence and deleting replicate outputs of the network. The blank tokens detach different slices of the output. In each piece, we remove repetitive tokens and then delete blank tokens to get the output sequence.

The constraint of the CTC decoder is that the length of the output sequence can't be more than the length of the input sequence [20]. The encoder-decoder model does not have this constraint.

2.1.5.2 Encoder Decoder

The encoder-decoder model [21] is used for end-to-end training input and output sequences from different domains with different lengths. An encoder-decoder model has two sub-models, the encoder model and the decoder model.

The "encoder" model shown in Fig. 2.16 receives the input sequence and produces a representation of the whole input. The encoder network is usually implemented by an RNN (an LSTM or a GRU network). The last hidden state of the encoder is the description of the input sequence and is called the context vector.

The decoder model is also implemented by an RNN network that takes in the context vector as the initial hidden state. The decoder network generates the next output of a sequence given the previous outputs. Two new classes are introduced to the vocabulary for start-of-sequence and end-of-sequence tokens. In the beginning, the initial state of the decoder is the last state of the encoder and the decoder input is the start-of-sequence token. In the training step, the decoder input is generated statistically, but in the inference step, it is fed back dynamically.

While training, we feed the target label of the network delayed by one time-step (l_{t-1}) as the input to the decoder as shown in Fig. 2.17. This technique is called teacher-forcing and helps with faster convergence of the network.

In the time of inference, we don't have the correct output sequence, so the

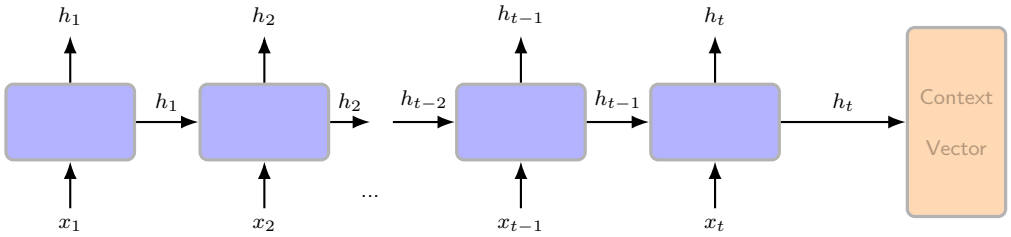


Figure 2.16: Illustration of the encoder network.

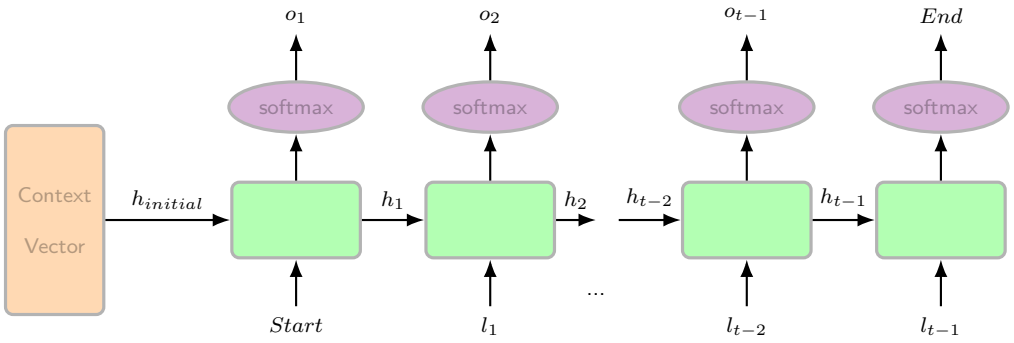


Figure 2.17: Illustration of the decoder network in the training step.

actual output of the decoder is fed back to the decoder in the next time-step (o_{t-1}) as shown in Fig. 2.18. We continue until the output is the end-of-sequence class.

2.1.5.3 Attention Mechanism

In the encoder-decoder model, the whole input sequence is encrypted and represented by a fixed size context vector. The size of the context vector can be a bottleneck for accuracy, especially for longer input sequences (a very long input is represented by a fixed size vector). Increasing the size of the context vector adds

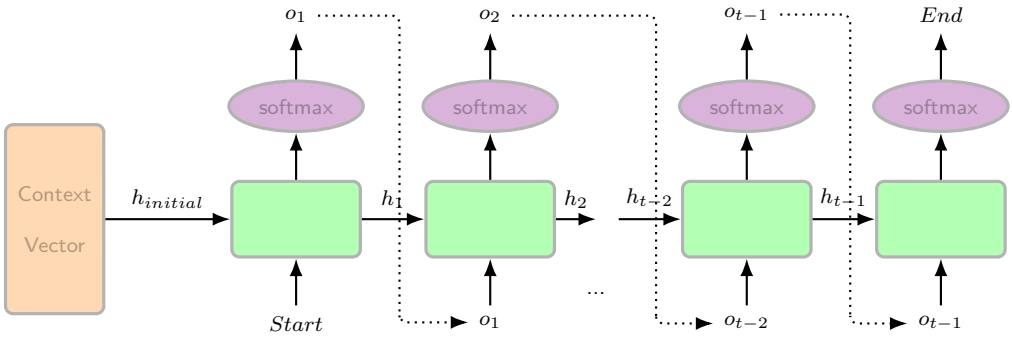


Figure 2.18: Illustration of the decoder network in the inference step.

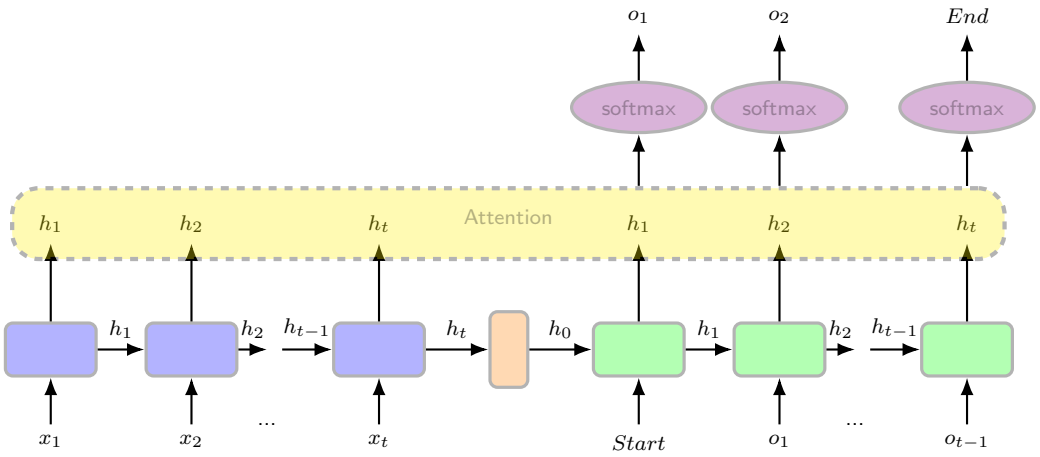


Figure 2.19: Illustration of the attention mechanism

complexity to training the network. The attention mechanism solves this problem by paying attention to a specific part of the input sequence at each time. The network architecture is shown in Fig. 2.19. Multiple designs have been proposed for attention mechanism. Bahdanau et al. [22], presented an additive attention structure and Lung et al. [23] proposed a multiplicative attention structure.

2.2 Hidden Markov Models

The Markov chain is a stochastic process assuming probability independence of a state from earlier states given one previous state (first-order Markov assumption). Hidden Markov Models (HMM) are statistical models employing the Markov assumption. The HMM is applied when a sequence of observations is the outcome of hidden underlying states of our interest and our goal is to describe the progress of the hidden states [24]. The observations can be discrete or continuous values.

2.2.1 Elements of an HMM

Each HMM with continuous observations is defined with these elements [25]:

1. The number of states in the HMM. States of an HMM are the set $\{s_1, ..., s_N\}$.

2. Emission probability of each state $b_j(e_i)$, describing the probability of observing an event e_i in this state s_j. If observations are continuous, the emission probability is shown with a probability distribution function.

3. Transition probability $\tau(i, j)$, describing the probability of transitioning from one state s_i to another state s_j. The transition probability is 0 if there is no transition between the two states.

4. The initial probability of each state $\pi(s_i)$, describing the probability of starting from state s_i.

Three main problems of an HMM model are [25]:

Evaluation problem: Given a model and a sequence of observations, calculate the probability that the observation sequence is related to this HMM model. The evaluation problem is solved using the forward-backward algorithm.

Decoding problem: Given a model and a sequence of observations, extract the most likely sequence of hidden states. The solution to this problem is a dynamic programming approach called the Viterbi algorithm.

Training problem: Like many other machine learning models, we should train HMMs to find the most optimal set of parameters (emission probabilities, initial probabilities and transition probabilities) that best describe the sequence of observations. Baum-Welch algorithm is used for training and finding a local optimum of the HMM.

3 Nanopore Sequencing: Background

This chapter provides the background of DNA sequencing and more specifically nanopore DNA sequencing. In section 3.1, we review the history of DNA sequencing and DNA sequencer generations. Section 3.2 explains nanopore DNA sequencing basics, devices and input data used for basecalling. In section 3.3, we review available Neural Network (NN) based and Hidden Markov Model (HMM) based basecallers, and compare their architectures, accuracy and other features.

3.1 DNA Sequencing

First-generation DNA sequencers developed by Sanger et al. [26] and Maxam and Gilbert [27] in 1977 are time-consuming and expensive machines. The Maxam-Gilbert technique also known as chemical sequencing uses toxin and radioactive chemicals and is considered dangerous [28]. Sanger technology, also known as the shotgun sequencing has better accuracy and more popularity compared to Maxam-Gilbert. The Sanger method is still used for smaller projects. The first human

genome was partially constructed using Sanger sequencing and cost around 100 million dollars.

Second-generation sequencers (SGS) process many parallel short reads and this increases their speed and lowers their cost compared to first generation sequencers. Main second-generation sequencing companies are 454/Roche, Illumina, Thermo Fisher and ABI. 454/Roche released the first second-generation sequencer in 2005 but Illumina is the most popular technology [28].

In an Illumina preparation kit, DNA samples are randomly fragmented and each DNA fragment is PCR amplified. PCR (Polymerase chain reaction) amplification produces thousands to millions of copies of one DNA read. The maximum read length basecalled by second generation sequencers is $\sim$400 bp [29]. Because of the constraint on the read length, second generation sequencers are not able to directly capture recurrent patterns in the genome.

Recently released third-generation DNA sequencers are portable, real-time, more affordable machines and can operate on DNA samples at least 100X longer than traditional sequencers. The two leading companies in this domain are Oxford Nanopore Technologies (ONT) and Pacific Biosciences sequencers (PacBio). As of November 2018, PacBio is an Illumina company.

PacBio uses SMRT (Single Molecule Real-Time) technology for sequencing [5]. SMRT technology uses light detection for sequencing and consists of tens of thou-

sands of Zero-Mode Waveguides (ZMWs) within which individual DNA samples are captured and sensed one base at-a-time [30]. Single molecule sequencing is an amplification-free technique.

Oxford Nanopore Technologies (ONT) uses nanopores and electrical current for sequencing. Nanopores are very small holes (on the order of a DNA helix diameter) arranged as an array (of thousands) in ONT devices. A DC voltage is applied across the nanopores, resulting in a DC electrical current. This current will change as a DNA molecule passes through the pore. These alterations can be used to determine the bases in the DNA.

Second and third generation sequencers are called next-generation sequencers (NGS).

3.2 Nanopore Sequencing

Nanopore sequencing can be performed using both nanopore categories, solid-state and biological nanopores [31]. Biological nanopores are naturally built by bacteria, have well-defined size and structure, are easy to produce and can easily be changed by genetic engineering [32]. The drawback of biological nanopores is their lack of stability, meaning that they can work over a narrower range of applied voltage and temperatures [31]. Solid-state nanopores are more stable but much harder to build at large scales. They can be produced using different materials such as silicon and

aluminum with different sizes [32].

ONT [33] is the first company to commercially release nanopore-based DNA sequencers. Available ONT sequencers are MinION, GridION and PromethION shown in Fig. 3.1.

MinION, pictured in Fig. 3.1a is a pocket-sized, portable and lightweight (90 grams) DNA sequencer. MinION plugs into the computer via a USB port. The read length can be specified to up to hundreds of kbp [6]. The MinION has 512 channels, each having 4 nanopores (2048 nanopores total) over an area of about 10×15 mm^2. Fragments of DNA are pushed into the device and flowed through the nanopore array. The nanopore is immersed in a conducting fluid through which a DC current, I_{DC}, is made to flow. As a DNA strand passes through the pore, it modulates this DC current related to the actual structure of the DNA. The flow of current is sampled over time which is the observable signal of the system.

The GridION, pictured in Fig. 3.1b is a benchtop system having the same technology as MinION. GridION has up to five MinION flowcells that can be used independently or jointly [34].

PromethION, pictured in Fig. 3.1c uses the same workflow as the MinION but on a much larger scale. PromethION has 48 modular independent flow cells, each having up to 3000 channels [35].

(a) MinION (b) GridION (c) PromethION

Figure 3.1: Oxford Nanopore Technologies DNA sequencers. (adapted from `nanoporetech.com/products`)

3.2.1 Nanopore Sequencing Data

The operational principles of nanopore sequencing involve the translocation of a DNA fragment through the nanopore and the ensuing measurements of resulting current alterations. The MinION device is connected to a computer via a USB port. MinKNOW, the desktop software of the ONT obtains data from the MinION and stores the sampled signal (a read) in a Fast5 file (a data format based on HDF5 format) [36]. MinKNOW can perform local basecalling on the streamed data. Main data levels stored in a Fast5 file are raw data, event data and base level data.

MinION Versions ONT has released five MinION chemistry versions (R6.0, R7.0, R7.3, R9 and R9.4) [31]. First versions of MinION used $\alpha - HL$ (hemolysin) biological nanopores and the latest versions use CsgG nanopores [31]. Jain et al. [37] compared R7.3 and R9 versions using *Escherichia coli* K-12. The error rate

of single-strand basecalling decreased from 26.7% in R7.3 to 14.5% in R9. And the error rate of double-strand basecalling decreased from 9.1% in R7.3 to 7.5% in R9. Translocation speed has increased from 30 bps in R7.3 to 250 bps in R9.0 and to 450 bps for R9.4 pore [31, 37].

Before the release of MinION version 9, the default basecaller model was based on a Hidden Markov Model (HMM) and with the release of version 9, ONT has released a number of Recurrent Neural Network (RNN) based basecallers.

Nanopore State As the length of a practical nanopore (i.e. the dimension parallel to the DNAs direction of translocation) is bigger than the size of one base, a set of bases pass through the nanopore at any one moment and together affect the measured current signal. If we assume that at any moment, k bases are present in the pore, the nanopore state may be associated with a sequence of k bases (k-mer). For example, if a 5-mer is responsible for generating the current from a nanopore at any one moment then we may think of the nanopore as being in any one of $4^5 = 1024$ possible states at any one moment.

Raw Data The time-series signal generated by DNA translocation through the nanopore is called the raw data. Each time step is stored as a 16-bit integer value [38]. Measurements are stored at a rate of 4 kHz. With a translocation rate of 450 base pair per second (bp/s), there is approximately one base per nine data

samples [38].

The raw data in a Fast5 file is shown in Fig. 3.2.

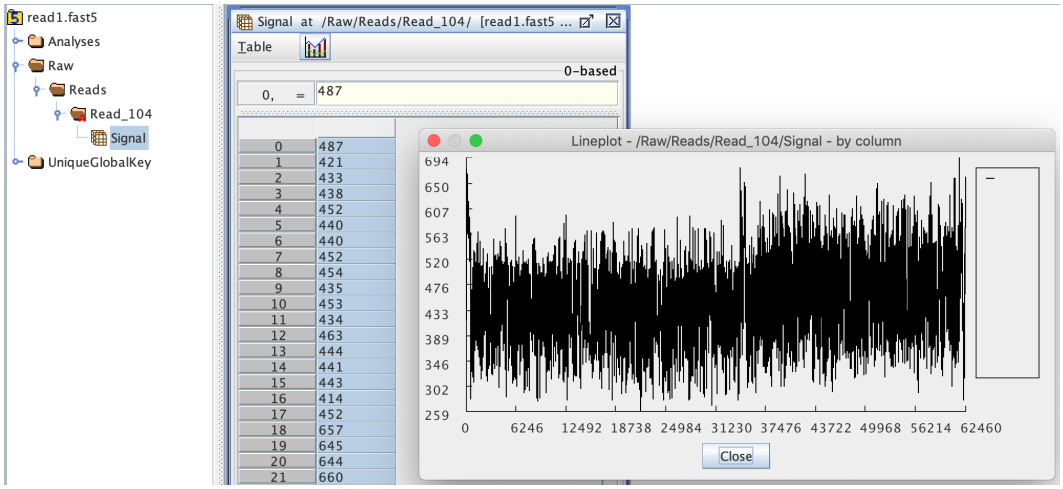


Figure 3.2: Raw data from a Fast5 file

Event Data Event detectors aggregate raw measurement samples and segment the data into a time sequence of piece-wise constant feature approximations, the so-called events. Each event segment consists of mean, standard deviation, start time and length. Ideally, one event should be produced for each new base entering the pore called a *step* condition. But because of the randomness in translocation, event detection is noisy and two error cases can occur [39]. First, a *stay* condition or insertion may occur when raw data points corresponding to one set of bases in the pore are segmented into two separate events (producing two events for just one base passing through the nanopore). Second, a *skip* condition or deletion may

occur when the event detection algorithm generates one event from raw data points corresponding to two sets of bases (missing a base passing the nanopore).

The event data in a Fast5 file is shown in Fig. 3.3:

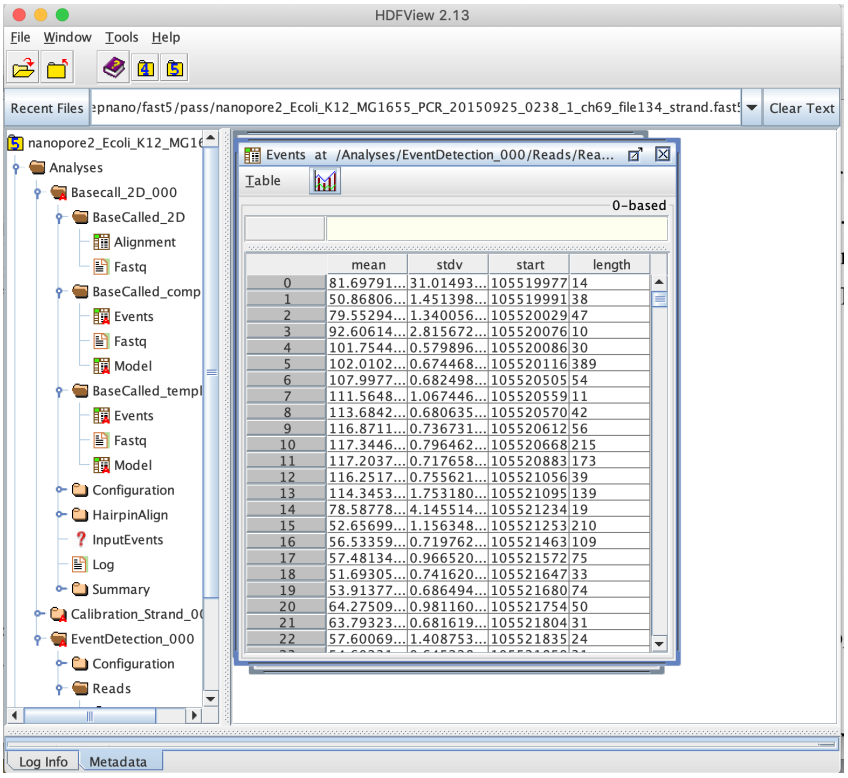


Figure 3.3: Events data from a Fast5 file

After event basecalling, the Events section of a Fast5 file may also contain the nanopore state at each event, the probability of the state, the move (stay, skip or step) and a Fasta file (a text file containing a letter-coded nucleotide sequence, starting with a description line) containing the base sequence.

$1D$ and $1D^2$ Data $1D$ and $1D^2$ (1D squared) are the two MinION sequencing workflows. In the $1D$ workflow, only one strand of DNA (template or complement) passes through the nanopore. In the $1D^2$ workflow, both DNA strands (template and complement) are sequenced by using a hairpin [31]. Single-strand and double-strand basecalling are used respectively for $1D$ and $1D^2$ sequenced data. Since the release of the higher resolution R9 version of the nanopore, the importance of $1D^2$ sequencing has decreased.

3.3 Basecalling

Basecalling is a sequence-to-sequence mapping problem. The input of a basecaller is the raw or event time-series data produced by nanopore sequencing. And the output is the text base sequence presumed (by the basecaller) to have originated the input data. Based on the input, basecallers are divided into raw and event basecallers.

In event basecalling, the length of input and output sequences are the same (each event represents a nanopore state or the corresponding base) and an RNN, a 1D-CNN or an HMM can be used for implementation. But in raw basecalling, the input sequence is longer than the output sequence as each base is the output of a set of current input signals. Thus, sequence-to-sequence mapping techniques should be used for implementation.

Since the release of the MinION sequencer in 2014, a variety of basecalling options have emerged and evolved for different MinION nanopore versions resulting in longer reads, higher yield and better basecalling accuracy. Machine learning models have also progressed from simpler HMMs to elaborate RNNs and CNNs.

Well-known basecallers are summarized in Table 3.1. Basecallers developed by ONT are Metrichor, Nanonet, Scrappie, Albacore and Guppy. Metrichor and Guppy have a proprietary license. Albacore is only available to users signed into an ONT company-forum. Nanonet and Scrappie are open source and available on Github. Open source basecallers provided by individual researchers include Nanocall, DeepNano, Basecrawller and Chiron.

Albacore and Scrappie have the ability to process both raw and event data. Basecrawller and Nanonet only take raw data points and internally perform event segmentation. Chiron performs basecalling on the raw signal without the segmentation step. DeepNano and Nanocall can only work with event data.

Basecallers have different speed and accuracies based on their underlying algorithm. The accuracy of a basecaller also depends on the quality of its signal. Read identity is the ratio of correctly identified bases to the number of bases in the reference genome for the sample. And consensus accuracy is the accuracy after post-processing the reads and aligning multiple output sequences. If the basecalling error is random, the consensus accuracy should be perfect. But in case of a

systematic error, taking the average doesn't correct the errors.

In a study by Wick et al. [40], a $1D$ R9.4 dataset of a bacterial genome (native DNA *Klebsiella pneumoniae*) was used to compare read identity and assembly identity of RNN-based basecallers. The Racon [41] module is used for the assembly in this study. Read identity and assembly identity values reported in Table 3.1 are the median accuracy and the highest accuracy of each basecaller reported in the mentioned comparison. The accuracy values for Basecrawller are not reported because of its poor performance on bacterial data (Basecrawller is only trained on human data). Albacore, Guppy and Scrappie have the best read identity. Chiron has the best assembly identity.

In some cases, Nanonet produces drastically shorter reads compared to other basecallers. And it may be the reason for its very high maximum accuracy in the comparison. This accuracy is not reliable as it is just for a very small fraction of reads [40].

HMM-based basecallers such as Nanocall and Metrichor are the older generation of basecallers and were not included in this comparison. These basecallers achieved accuracies in the range of $65\% - 85\%$ on version R7.3 MinION data [1].

Each basecaller was run on different hardware in the study, so their speed cannot be compared. But Chiron seems to be the slowest, while Guppy is the fastest basecaller. Basecalling a read set of size 1.2 Gbp took two weeks to complete on

Chiron. While basecalling the same read on the same hardware took 30 minutes to finish on Guppy.

Other information summarized in Table 3.1 is the development language, libraries used for the implementation, the input signal of the basecaller and the neural network architecture.

Table 3.1: Available nanopore sequencing basecallers.

Name	Author	License	Model	Language	Library	Data	Architecture	Read Id	Assembly Id
Metrichor	ONT	P	HMM, RNN	-	-	Event	-	-	-
Nanocall	David et al.	O	HMM	C++	-	Event	4096 State HMM	-	-
DeepNano	Boža et al.	O	RNN	Python	Theano	Event	Single-strand 3 (100) BGRU + 2 (5) Softmax Double-strand 4 (250) BGRU + 2 (5) Softmax	MAX 87% MED 80%	MAX 98.8% MED -
Nanonet	ONT	O	RNN	Python	Current	Raw → Event	2 (128) BLSTM + 1 (1024) Softmax	MAX 99% Not Reliable MED 84%	MAX 99.4% MED 98.9%
Scrappie	ONT	O	RNN, CNN	C++	-	Event, Raw	Event 5 BGRU + 1 (1025) Softmax	MAX 95% MED 88%	MAX 99.6% MED 99.3%

Albacore	ONT	C	RNN, CNN	Python	-	Event, Raw	Event 2 (192) BLSTM + 1 (1025) Softmax Raw 1 (96) CNN +2 (96) BGRU + 1 (1025) Softmax + CTC	MAX 95% MED 88%	MAX 99.8% MED 99.5%
Guppy	ONT	P	-	-	-	-	-	MAX 95% MED 88%	MAX 99.8% MED 99.5%
Basecrawller	Stoiber et al.	O	RNN	Python	Tensorflow	Raw	3 (75/100/50) LSTM + 1 (256) Softmax + Segmentation + 2 (200/75) LSTM + 1 (21) Softmax	-	-
Chiron	Teng et al.	O	CNN, RNN	Python	Tensorflow	Raw	5 (256) CNN + 3 (200) BLSTM + 1 Softmax + CTC	MAX 95% MED 87%	MAX 99.9% MED 99.8%

HMM-based and RNN-based basecallers are described chronologically in the following sections.

3.3.1 HMM-based Basecalling

The Hidden Markov Model (HMM) is the former default model used for ONT's basecalling. In the basecalling problem, the state of the HMM is the combination of bases in the nanopore at some measurement instant. If we assume a 6-mer condition for the nanopore (at each time, 6 bases are present in the nanopore), then the HMM has $N = 4^6 = 4096$ states. If we choose to model one skipped base with the HMM, then the number of transitions from each state is at most: $1(stay) + 4(step) + 16(skip) = 21$ transitions. And the transition probability is a function of p_{stay}, p_{step}, and p_{skip}.

The HMM is trained several times using the Baum-Welch algorithm to find an optimal set of parameters for initial probabilities, emission probabilities and transition probabilities. The goal of the basecaller is to find the most likely path that the HMM has passed while observing the sequence of nanopore events (x_i, y_i). The Viterbi algorithm is used for decoding and finding the most likely sequence of states for each event sequence. Metrichor and Nanocall are two well-known HMM-based basecallers.

Some drawbacks of using HMMs are as follows [1]: HMMs can only model short-

range dependencies, but there are long-range dependencies in nanopore data that are hard for an HMM to capture. A prior model of DNA is a part of the HMM model, and this can be a problem when dealing with an unknown DNA.

3.3.1.1 Metrichor

Metrichor was the first MinION basecaller and is no longer available [40]. Metrichor was only available as a cloud-based service with a proprietary license. Metrichor used an HMM-based approach to achieve accuracies in the low 70% for R7 nanopore [43]. Metrichor was developed for both single- and double-strand basecalling. The details of Metrichor's implementation are not available because of its proprietary license.

With the release of MinION version R9, Metrichor was integrated into ONT's EPI2ME platform and updated to an RNN-based basecaller. The EPI2ME platform allows local single-strand basecalling [31].

3.3.1.2 Nanocall

Nanocall is the first open-source basecaller and is available at `https://github.com/mateidavid/nanocall`. Nanocall is developed in C++ using HMMs for modeling the 6-mer nanopore data for the R7 nanopore. Nanocall implements single-strand basecalling.

The HMM initial emission probabilities come from the pore models provided by ONT. Each state is connected to at most 21 other states (16 states for a skip condition, 4 states for a step condition and 1 state for a stay condition). The HMM is trained using the Baum Welch algorithm. The identity rate of Nanocall is 68% for bacteria and human genome, which is comparable to HMM-based Metrichor.

3.3.2 Neural Networks Based Basecalling

Since the release of MinION version 9, RNNs have been the main model used in basecallers. In an event basecaller, an RNN network (LSTM or GRU) is used to process the time-series event input data. A Softmax output layer is then used for classification.

The output layer of basecallers can be different in a sense that some basecallers directly output the probability of each base, and some calculate a logit probability for the state of the pore and then perform an extra step to find the corresponding base sequence. In case of predicting nanopore state, the number of classes is 4^k, when the model state is represented by a k-mer. However, when generating the probability of bases, the output layer has four units, one for each of the four bases (A, C, G, T).

The most recent basecallers focus on processing raw data, eliminating the event segmentation step. A separate RNN or CNN is first used for feature detection.

An RNN network is again used for processing the time-series data. Available raw basecallers use the CTC decoder for aligning the raw signal and the base sequence output. In this case, the Softmax output layer has an extra unit for the "blank" class (required for segmentation in the CTC approach).

Below is an outline of various RNN-based basecallers developed by both ONT and individual developers.

3.3.2.1 DeepNano

DeepNano is an RNN-based basecaller developed by Bož a et al. [1]. The source code is publicly available on `https://bitbucket.org/vboza/deepnano/overview`. The development language is Python and the Theano deep learning library [44] is used for training the network. DeepNano only implements event basecalling and considers two versions of MinION data (R7 and R9). It is trained for single- and double-strand basecalling for the R7 version data but is only trained for single-strand basecalling for R9 version data.

Input features used from event data are mean, standard deviation, length of the event and the auxiliary feature of mean squared. A deep GRU network with 3 bidirectional GRU layers, each having 100 nodes is used for single-strand basecalling.

For double-strand basecalling, template and complement event sequences are aligned into one signal as (e_t, e_c). Two options are available for aligning the events.

The first option is to use the alignment information available in the already base-called Fast5 file. The second option is to use the DeepNano dynamic programming algorithm (implemented in an external C code) over the output probabilities of the single-strand basecaller. A deep GRU network with 4 bidirectional GRU layers, each having 250 nodes is then trained for basecalling this aligned signal. Double-strand basecalling can be performed by first single-strand basecalling template and complement sequences and then aligning the output base sequences by dynamic programming. But this approach doesn't perform well [1].

The output layer is implemented as two five class classifiers. This structure is robust to deletion (stay) and one insertion (skip) error. If the event segmentation represents exactly one new base (that is, the transition is a step), the first classifier shows the base and the second one reports an N (none). But in a skip situation, the second classifier shows the skipped base. And if the transition is a stay, both of the classifiers show N. The output sequence is generated by concatenating all of the outputs except the N outputs. Insertion and deletion transitions can also be represented using a 21-class classifier, four classes representing the step condition (A, C, G, T), one class for a stay (N) and 16 classes for a skip condition (AA, AC, ..., TT). Two smaller classifiers (the first approach) showed better results than one bigger classifier [1].

For constructing the training data, the signal is first basecalled using an existing

basecaller (Metrichor in this case). The already basecalled sequence is then aligned to its reference genome using a simple heuristic to correct any errors [1]. While training, events are realigned to the reference genome after every 100 steps using the DeepNano output probabilities.

DeepNano trained for R7 single-strand data is compared to Nanocall and Metrichor for E.coli and K. pneumoniae samples in Table 3.2. DeepNano has a higher accuracy than both Nanocall and Metrichor for single-strand basecalling. Table 3.3 compares the accuracy of DeepNano and Metrichor for double-strand basecalling (Nanocall doesn't support double-strand basecalling). DeepNano also has higher accuracy than Metrichor for double-strand basecalling . DeepNano for version R9 is compared to Nanonet from ONT in Table 3.4 and they have similar accuracy but DeepNano has higher speed.

3.3.2.2 Nanonet

Nanonet is an open source RNN-based basecaller provided by ONT and is available on Github at `github.com/nanoporetech/nanonet`. Nanonet implements both single- and double-strand basecalling.

The Nanonet input data is the raw data streaming out of MinKNOW. Nanonet performs event detection on the raw signal before basecalling. Input features are mean, standard deviation, length of the event and the auxiliary feature mean diff

Table 3.2: Accuracy of single-strand basecalling on two R7.3 data sets (adapted from [1]).

Basecaller	Accuracy	
	E. coli	K. pneumoniae
DeepNano	77.2%	76%
Nanocall	68.4%	67%
Metrichor	71.4%	68.8%

Table 3.3: Accuracy of double-strand basecalling on two R7.3 data sets (adapted from [1]).

Basecaller	Accuracy	
	E. coli	K. pneumoniae
DeepNano	88.5%	86.7%
Metrichor	86.8%	84.8%

Table 3.4: Accuracy of single-strand basecalling on an R9 E.coli sample (adapted from [1]).

Basecaller	Accuracy	Speed
DeepNano	83.2%	2057 event/s
Nanonet	81%	4716 event/s

that is the difference between consecutive event means.

The RNN has 2 bidirectional LSTM layers, each having 128 nodes. The output layer is a Softmax layer with 1024 units presenting the nanopore states (the state of the model is a 5-mer, so there are $4^5 = 1024$ states). As Nanonet outputs the state of the network, an extra pass on the output is needed to generate the sequence of bases.

Nanonet doesn't use any libraries for basecalling but does use the Currennt library for training on GPUs. The Currennt library is a Python library for RNNs available at `sourceforge.net/projects/currennt`. It uses NVIDIA graphics cards to accelerate computing. Currennt implements uni- and bidirectional LSTM layers but it doesn't support the GRU layer.

There is also an OpenCL version of single- and double-strand basecallers in Nanonet. OpenCL is a framework for accelerating code performance on different

devices.

3.3.2.3 Albacore

Albacore is the first stand-alone ONT basecaller (i.e. not a cloud service), albeit with its source code not publicly available. Albacore implements single- and double-strand basecalling for both raw and event data.

3.3.2.4 Scrappie

Scrappie is the research basecaller from ONT in C++ and is available on Github at `https://github.com/nanoporetech/scrappie`. Scrappie can process both raw and event data. A Python binding for the raw Scrappie basecaller is available on the same Github page. For R9.4 and R9.5 versions, Scrappie can basecall raw data without requiring an event detection step [45]. Successful modifications in Scrappie get implemented in Albacore [40].

3.3.2.5 Guppy

Guppy is the newest basecaller from ONT and its source code is not publicly available. Guppy runs faster than Albacore as it can use GPUs in addition to CPUs.

3.3.2.6 BasecRAWller

Basecrawller is an open-source basecaller developed by Stoiber and Brown [38]. It is developed in Python using the Tensorflow library. Basecrawller only implements raw single-strand basecalling.

Basecrawller processes raw data using two unidirectional RNNs [38]. Having only unidirectional networks make basecrawller open to online basecalling.

The first RNN network, called the raw net, takes in the normalized raw measurements and produces a logit probability for each 4-mer (4 combinations of 4 bases, total 256 states) and a second output showing if this input represents a new base. The raw net has 3 LSTM layers with 75, 100 and 50 nodes respectively. The first output is a 256 node Softmax layer and the second output is a 1 unit dense layer. Based on the raw net output probabilities for each 4-mer and the probability of each data point representing a new base, the segmentation step segments data points representing a 4-mer.

The second RNN network, called fine-tune net, takes in the average of 4-mer probability outputs of the raw net within each segment, so the input is a vector of probabilities for each 4-mer. Fine-tune net generates a logit probability for between 0 and 2-mer length states. The 0 length shows that this input should have been merged with other inputs and does not represent a different base. A 1-mer for the

case segmentation was accurate and the input is only derived from one base. And a 2-mer if each input represents more than one base. The fine-tune net has 2 LSTM layers with 200 and 75 nodes respectively and a 21 node Softmax layer is used as the output.

3.3.2.7 Chiron

Chiron only implements raw single-strand basecalling and directly basecalls from raw data with no segmentation step. Chiron has a more complex architecture compared to other basecallers, stacking five 256 unit 1D convolutional layers for feature detection followed by three 200 unit bidirectional LSTM layers for sequence processing. A fully connected layer with Softmax activation function and a CTC decoder layer to finish the sequence classification [42]. Chiron is very slow because of its complex architecture.

Chiron uses a sliding window with length 300 to traverse the raw signal and slide the window by 30 data points. Overlapped windows get stacked and build a consensus sequence. Windows can be processed in parallel for better speed.

3.3.3 Training Data Preparation

A set of signals (raw data) or events (event data) and their equivalent base sequences are required for training a basecaller. The output of MinION is available in a Fast5

file produced by the MinKNOW software. The file reads may or may not have been processed by an ONT basecaller. In order to label the nanopore signals/events, we need to find the alignment of the output sequence and the signals/events.

Raw data For constructing raw signal training data, Nanoraw software is used to segment the raw data and the corresponding base sequence. Basecrawller and Chiron both use this software to build their training data [42, 38]. The signal is first basecalled using an existing basecaller. Then with a genomic alignment (comparing the basecalled sequence with the reference genome), errors will be corrected. Nanoraw aligns this base sequence to the signal, providing the boundary (start time, end time) of the raw signal for each base. The *genome_resquiggle* algorithm from Nanoraw is used for this purpose. The detailed algorithm and the process is described in the manuscript [46]. Nanoraw also has the ability to identify chemically modified nucleotides. This software is available at `https://nanoraw.readthedocs.io/en/latest/`. We used the training data provided by the Chiron team for training our raw basecaller presented in chapter 5.

Event data Preparing a training dataset for Nanonet was explained in Oxford Nanopore London Calling 2017 conference by Scott Gigante [47]. Nanopolish software available at `https://github.com/jts/nanopolish` is used to associate the k-mers of the reference with each nanopore event.

4 Single- and Double-strand Basecalling for Simulated Event Data

In this chapter, we present the design and implementation of two RNN-based basecallers for $1D$ and $1D^2$ simulated event data.

The electric current signal generated by the translocation of DNA through the nanopore experiences noise. This noise is caused by both the nanopore sensor and the measurement circuitry. The weak current signal along with the noise (i.e. low signal-to-noise ratio (SNR)) is the major cause for the low accuracy of basecalling using nanopore-based data. To investigate the effect of noise, we simulate a 3-mer nanopore-based signal and impose different input SNR conditions upon it. We did not introduce event detection errors (insertions and deletions) into our simulation. We study the behavior of our basecallers as a function of the time-series SNR.

In section 4.1, we describe our data simulation. In section 4.2, the implementation of our RNN-based single-strand basecaller along with the analysis of its performance on data with different noise levels is represented. In section 4.3, we

present our double-strand basecaller and its accuracy for event data with different SNR values.

4.1 Event Data Simulation

To exert better control over the study and design of our basecalling algorithms and to investigate the influence of noise on the accuracy, we simulate the nanopore event data.

In particular, we use approximated R9 nanopore sensor characteristics as shown in Fig. 4.1 [48]. This graph shows the relative influence that a DNA base has on the input signal depending on its position in the pore (1 is close to the entry into the pore and 5 is close to the exit from the pore). We can model this continuous graph with a discrete representation. This discrete representation effectively localizes the impact of the translocating bases to a smaller-subset of intra-pore positions. By integrating the continuous graph over 3 time-steps, we obtained the 3-mer discrete representation shown in Fig. 4.2. Such a model may effectively serve as a look-up-table equivalent for the output signals from the nanopore under study.

We assume three bases (a 3-mer) are present in the pore at a time in our simulations [49]. We use three look-up-tables for simulating mean, standard deviation and length of an event, respectively. We generate a random sequence of bases and create the event features per every three bases using the look-up-tables. The ap-

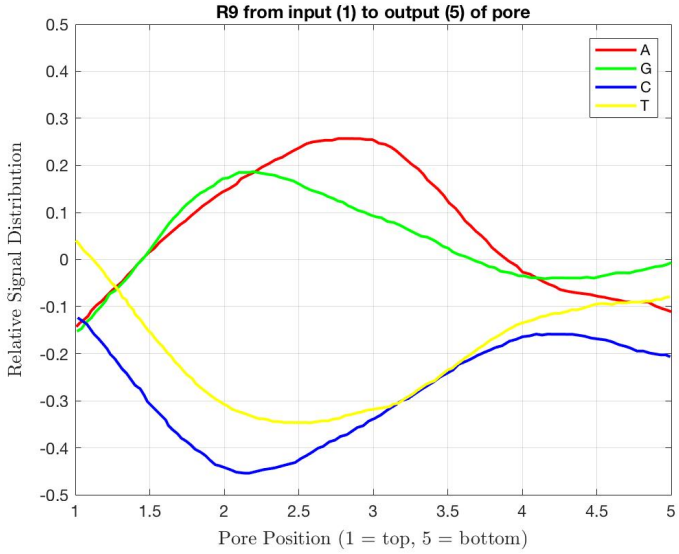


Figure 4.1: The relative influence of DNA bases on the current signal depending on their position in the pore.

proximate nature of this pore is only meant to convey the relative complexity of the model and hence only generally guide an appropriate basecaller design. Specific basecallers will need to work on data obtained from specific nanopore sensors.

4.2 Single-strand basecaller

So-called "single-strand" basecallers use nanopore sequencing data obtained from only one (of the two) strands comprising the traditional double-helical DNA structure. The basecaller input data is the sequence of events generated by the sequencer, which we simulated using the approximated nanopore characteristics noted above.

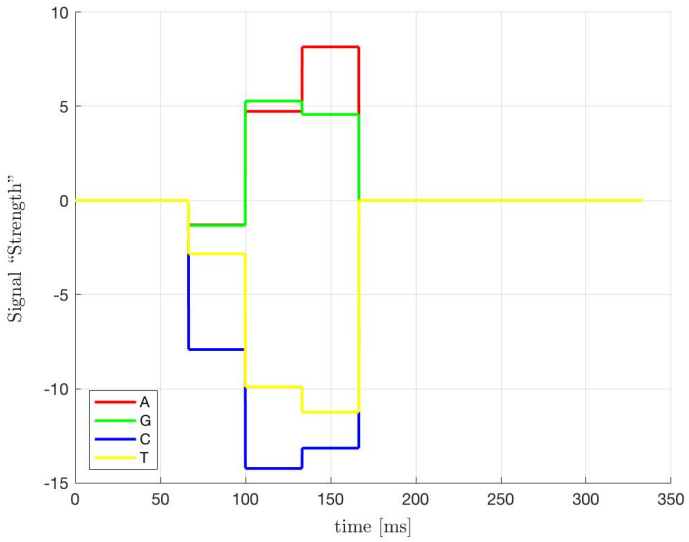


Figure 4.2: Discrete R9 impulse response for a 3-mer model.

The output sequence can either be the sequence of bases having the major contribution on the input (the last base) or the sequence of nanopore states (all of the bases present in the pore). The latter makes the network bigger and requires an extra decoding step after basecalling. We implemented our single-strand basecaller in Python using the Keras library.

4.2.1 RNN Basecaller Architecture

Input Layer The network input is the sequence of simulated events for a DNA strand. Event features are the three simulated features (mean, standard deviation, length) on top of which we create mean squared and difference of consecutive means

as auxiliary features. Mean squared is an auxiliary feature in DeepNano and difference of consecutive means is used in Nanonet and Scrappie. Both of the helper features resulted in accuracy improvement. Thus, each event has five continuous features. Fig. 4.3 shows a number of simulated events along with their base label.

	A	B	C	D	E	F	G
1	Mean	Standard Dev	Length	Mean Diff	Mean Square	Label	Base
2	-0.026403	0.78087	0.40215	0	0.00069709	2	G
3	20.216	0.34176	0.24862	20.242	408.68	3	T
4	1.617	0.975	0.46216	-18.599	2.6146	3	T
5	-12.579	0.08918	0.013104	-14.196	158.23	0	A
6	1.9577	0.65908	0.12993	14.537	3.8325	0	A
7	19.674	0.57566	0.27879	17.716	387.06	3	T
8	6.6638	0.11012	0.14227	-13.01	44.406	0	A
9	0.46917	0.49781	0.43893	-6.1946	0.22012	3	T
10	6.5156	0.0035425	0.27337	6.0464	42.453	2	G
11	2.3907	0.3226	0.41298	-4.1249	5.7155	2	G
12	11.753	0.39304	0.081374	9.3626	138.14	1	C
13	-2.8923	0.0043334	0.32509	-14.646	8.3653	3	T
14	-14.846	0.26052	0.21413	-11.954	220.4	2	G

Figure 4.3: Simulated event features and the event label.

Hidden Layers We searched through different network architectures and achieved the best generalization and performance using 2 stacked GRU layers with 30 units.

Output Layer The output layer can be a four unit fully-connected layer with Softmax activation function, determining the logit probability of the four bases (four class classification). The output layer can also be a 64 unit fully-connected

layer with a Softmax activation function, determining the probability of each 3-mer state (64 class classification). We examined both output layers and achieved the same accuracy. We decided to implement our basecaller to output the probability of the four bases, as it needs fewer parameters.

Architecture Search We evaluated different neural network layer types appropriate for sequence input data such as GRU, LSTM, bidirectional GRU and 1-dimensional CNN. The configurations are summarized in Table 4.1. Tanh kernel activation and Sigmoid recurrent activation functions are used for all RNN layers. The ReLU activation function is used for CNN layers.

A number of the hidden layer architectures upon which we experimented, along with the number of learnable parameters of each network, are listed in Table 4.2. The number of parameters are calculated with a five feature input layer and a four unit fully-connected output layer. The accuracy is reported for a dataset with an SNR of 10 dB. We achieved the same accuracy using GRU layers and 1-dimensional CNN layers. Bidirectional GRU and LSTM layers did not show a performance improvement.

Table 4.1: Investigated configurations for single-strand event basecaller

Layer Type	Number of Layers	Nodes
BGRU	$\in \{1, 2\}$	$\in \{20, 30\}$
GRU	$\in \{1, 2\}$	$\in \{20, 30, 50, 100\}$
LSTM	$\in \{1, 2\}$	$\in \{20, 30\}$
		Filters $\in \{32\}$
CNN	$\in \{1, 2\}$	Filter Size $\in \{1, 2, 3\}$
		Stride 1
		CNN Filter 32
CRNN	2	Filter Size, Stride 1
		GRU 30

Table 4.2: A number of evaluated architectures for single-strand event basecaller with their accuracy and number of parameters

Type	Layers	Nodes	Parameters	Accuracy
BGRU	2	20	10,124	87%
LSTM	2	30	11,284	85%
CNN	2	32 (1)	1,252	94%
GRU	2	20	4,104	91%
GRU	2	30	8,854	95%
GRU	2	50	23,754	95%

4.2.2 Noise Level Analysis

We trained a deep GRU network for five noise levels and compared their accuracy and loss. Experimental results are listed in Table 4.3. SNR values are in dB. These results were obtained using the five event input features noted above (mean, standard deviation, length, mean squared and mean difference of consecutive events), two stacked GRU layers (30 nodes each) and a four unit Softmax layer. The model is shown in Fig. 4.4. Accuracy dropped from 99.5% for SNR 40 to 90% for SNR 5.

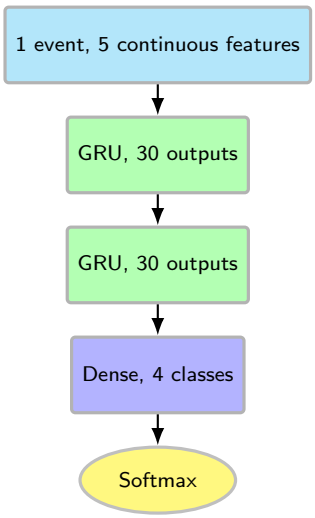


Figure 4.4: Model used for single-strand basecalling.

Table 4.3: Loss and accuracy of single-strand basecalling on data with different noise levels.

SNR (dB)	Validation Loss	Validation Accuracy
40	0.02	99.5%
20	0.03	98.5%
15	0.065	97.5%
10	0.12	95%
5	0.23	90%

4.3 Double-strand basecaller

In double-strand basecalling, we use the nanopore sequencing data of both (template and complement) strands of DNA. The two strands can get basecalled simultaneously in order to get better accuracy given the fact that the strands are complements of each other (i.e. A/C/G/T on one strand always couples to a T/G/C/A on the other) and thus introduce the possibility of effectively raising the signal strength by employing methods of joint detection.

Double-strand basecalling is usually done by first aligning the template and complement strand events into a joint signal (e_t, e_c). But our network takes the template and complement events as two separate inputs to the network. We used the network architecture shown in Fig. 4.5. We implemented our double-strand basecaller in Python using the Keras library.

4.3.1 RNN Basecaller Architecture

Input layer The network input is two sequences of simulated events from both DNA strands. Each simulated event has five features (mean, standard deviation, length, mean squared and difference of consecutive means) as mentioned in 4.2.1. So, the network has two inputs, each having five continuous features.

Hidden Layers As the network has two separate inputs, the flow of information should get concatenated for the output layer. We evaluated GRU and bidirectional GRU networks with different architectures for before and after the merge. A model having 2 stacked GRU layers with 5 and 10 hidden units respectively showed the best performance and generalization.

Output Layer The double-strand basecaller outputs 4 probabilities for each base group (A-T, C-G, G-C, T-A). The output layer is a four unit fully-connected dense layer with Softmax activation function.

Architecture Search Some of the hidden layer architectures upon which we experimented, along with the number of learnable parameters of each network, are listed in Table 4.4. The number of parameters are calculated with two separate five-feature input layers and a four unit fully-connected output layer. The accuracy is presented for a dataset with an SNR of 10 dB.

4.3.2 Noise Level Analysis

We evaluated our basecaller on the simulated data with five different noise levels and the results are shown in Table 4.5. The effect of redundancy caused by double-strand basecalling is significant as we increase the noise. The accuracy only dropped to 99.3% for an SNR of 5 dB compared to 90% for single-strand basecall-

Table 4.4: Evaluated architectures for the double-strand event basecaller

Before Merge			After Merge			Parameters	Accuracy
Type	Layers	Node	Type	Layers	Node		
BGRU	1	20	GRU	1	20	12,384	99.2%
GRU	1	30	GRU	1	30	14,794	99.4%
GRU	1	20	GRU	1	20	6,864	99.3%
GRU	1	10	GRU	1	10	1,934	99.3%
GRU	1	5	GRU	1	10	1,004	99.5%

ing at the same noise level. The number of network parameters has also decreased significantly.

4.4 Conclusion

We simulated nanopore sequencing data using base-relative influence curves presented by ONT. Each event has 5 features: mean, standard deviation, length, mean squared and consecutive mean difference. We implemented a single-strand basecaller that receives a sequence of events from one strand of DNA and outputs four probabilities for the four bases. The accuracy dropped significantly as we decreased the signal to noise ratio of the data, 99.5% accuracy for an SNR of 40 dB and 90%

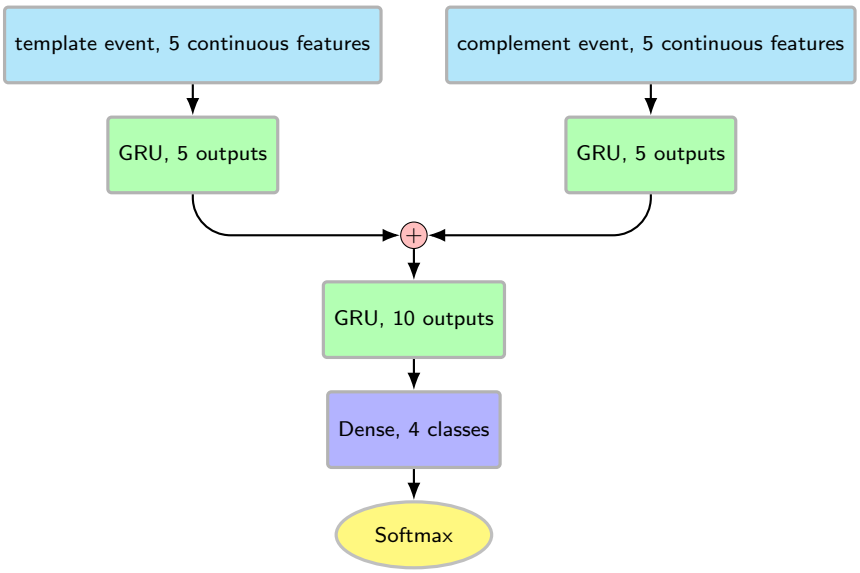


Figure 4.5: Model used for double-strand basecalling.

Table 4.5: Loss and accuracy of double-strand basecalling on data with different noise levels.

SNR (dB)	Validation Loss	Validation Accuracy
40	0	100%
20	0.004	99.85%
15	0.01	99.7%
10	0.02	99.5%
5	0.02	99.3%

accuracy for an SNR of 5 dB. We implemented a double-strand basecaller, receiving two event sequences from both strands of DNA and outputting four probabilities for each of the possible four base-couples (A-T, C-G, G-C, T-A). With a much smaller network, we could achieve 99.3% accuracy instead of the 90% achieved in single-strand under an SNR of 5 dB.

5 Single-strand Basecalling Raw Nanopore Data

The speed of DNA translocation through a nanopore sensor is variable and this results in error-prone event detection. The newer generation of basecallers focus on directly processing the raw nanopore signal eliminating the event detection step. Chiron, the first basecaller that directly processes the raw nanopore data and other available end-to-end basecallers, use a CTC-style approach for translating the raw input signal to the equivalent base sequence. The deep learning encoder-decoder model has shown great results in speech recognition and neural machine translation compared to the CTC network [7, 8]. Our goal is to advance the end-to-end basecalling by employing the encoder-decoder model.

In this chapter, we describe the design and implementation of our encoder-decoder based basecaller for single-strand raw nanopore input data. In section 5.1, we address our data preparation. In section 5.2, we present our encoder-decoder model architecture. Section 5.3 and 5.4 describe our training and inference steps. Section 5.5 shows our results and the performance of our basecaller.

5.1 Data Preparation

We process raw data files produced by the MinION device. A read length in these files can be up to hundreds of kbp (the lengths are random with constraints based on the specific experimental conditions employed). Learning very long input sequences is very hard for RNNs (because of the vanishing gradient problem through the back-propagation process), requires a large amount of memory and is very slow.

We use a sliding window on the raw input signal to process smaller fragments of each read to decrease memory requirements and increase the training speed. We process 300 raw measurements and slide the window by 30 raw measurements at each time. After basecalling, we assemble the output of each window and generate a consensus sequence for each read.

5.1.1 Data Preprocessing

We rescale the read signal values so that the mean is 0 and the standard deviation is 1 (i.e. standardization).

5.1.2 Training Data Preparation

For developing the training data, we first basecall a raw signal sample with an available basecaller. Then we correct basecalling errors by aligning the sequence to

the sample reference genome. Nanoraw software (specifically its *genome_resquiggle* algorithm) [46] is then used to align the raw data and the corresponding base sequence. Nanoraw segments this base sequence to the signal, providing the boundary (start time, end time) of the raw signal for each base.

Encoder Input The encoder input is a 300 long frame of the raw nanopore signal. So, each encoder input sample has 300 time-steps and each time-step has 1 continuous feature.

Output Label Using the signal boundary for each base produced by Nanoraw, we select the sequence of bases that originated the encoder input. We add an end-of-sequence label to the target output sequence.

Decoder Input We employ the teacher-forcing training technique as described in 2.1.5.2. Instead of feeding back the actual output of the decoder network, we deliver the output label of the previous time-step (the correct decoder output) for faster convergence.

While training, the decoder input is the target output sequence delayed by one time-step, starting with the start-of-sequence label.

Batching Because of the variance in DNA translocation speed through a nanopore, the number of bases corresponding to every 300 raw signals is different. Thus, the

length of the output labels (the base sequence represented by each input sliding window) are variable.

In order to train on batches of training samples, the length of the output sequences (and the input sequences) in a batch should be of the same length. We need to pad or truncate output labels to a fixed maximum length. We pad shorter sequences by the end-of-sequence token and truncate longer sequences. In our implementation, we set the maximum length to 40. Meaning that on average, for every 300 current signals generated by the MinION, 40 DNA bases has passed through the nanopore.

5.2 Network Architecture

We use the encoder-decoder model described in 2.1.5.2 with attention mechanism for developing our basecaller (Mauler). We implemented Mauler in Python using the Keras library.

5.2.1 Encoder

The encoder model processes a sequence of raw inputs and produces an internal representation of it. We tested different encoder model architectures including only RNN (GRU, LSTM, bidirectional layers) and CNN-RNN architectures with different numbers of layers. The best performance was achieved using the model

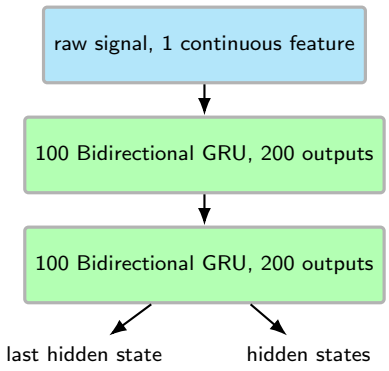


Figure 5.1: The encoder sub-model.

pictured in Fig. 5.1. This model has 241,800 parameters.

Encoder Input The encoder input is a sequence of raw signals (1 continuous feature) with a fixed length of 300.

CNN Layers We examined using 1D-CNN layers with different number of units as a feature detector for different RNN network architectures. This resulted in higher number of network parameters and overfitting the training data. We achieved a better performance using only RNN layers.

RNN Layers We examined different architectures of uni-directional and bidirectional LSTM and GRU networks. For choosing the RNN layer type (GRU and LSTM), we selected the units so that networks have roughly the same number of parameters. The GRU layers generalized better to this problem. We gained the

best accuracy using 2 stacked bidirectional GRU layers with 100 units.

Encoder Output The hidden state of the last bidirectional GRU layer for all of the input time-steps is used for the attention mechanism. The decoder model uses the hidden state of the last bidirectional GRU layer for the last input time-step.

5.2.2 Decoder

The decoder network decodes the encoder representation of the input to the output sequence. This network generates the next output of a sequence given the previous outputs. We investigated different decoder model architectures with different numbers of GRU layers. The best performance was achieved using the model pictured in Fig. 5.2. This model has 123,205 parameters.

Decoder Input The decoder input is a sequence of labels (1 discrete feature): {start-of-sequence, A, C, G, T, end-of-sequence} with a fixed length of 40 (padded or truncated for batching). The start-of-sequence label is the first decoder input.

The decoder model also receives the last hidden state of the encoder model.

RNN Layers We examined different GRU networks and achieved the best accuracy using one GRU layer with 200 units. The last hidden state of the encoder model is used to set the initial hidden state of this RNN network.

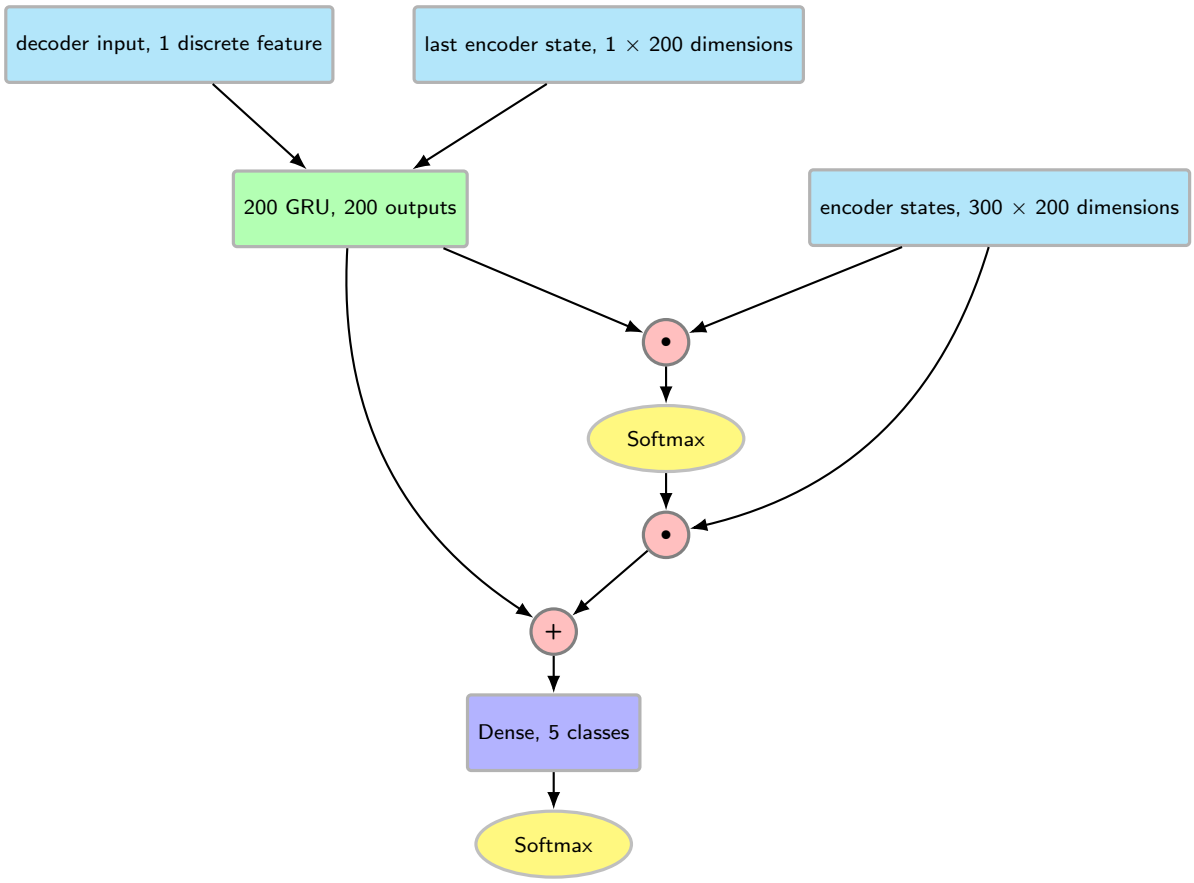


Figure 5.2: The decoder sub-model with the dot-based global attention mechanism.

Decoder Output The output of the decoder network is the last hidden state of the GRU layer.

5.2.3 Attention

The attention mechanism is an addition to the encoder-decoder model that results in higher performance when dealing with long input sequences. We employed the attention mechanism proposed by Luong et al. [23] called the "global attention". Different scoring functions are presented in this paper. We used the dot-based score in our implementation [50]. The attention mechanism is pictured in Fig. 5.2.

We first calculate the attention score. The attention score determines the effect of each input time-step on the output and is calculated as the dot product of hidden states of the encoder network and the last hidden state of the decoder network. We normalize the attention score with a Softmax activation function. The context vector is then calculated as the dot product of the attention score and the encoder hidden states. The context vector is concatenated with the decoder hidden state for decision making in the output layer.

5.2.4 Output Layer

The output layer is a five unit fully-connected layer with Softmax activation function. The output classes are: {A, C, G, T, end-of-sequence}.

5.2.5 Architecture Search

We explored different encoder networks including RNN and CNN-RNN architectures in combination with different RNN architectures for the decoder model.

A number of the examined models that employed the CNN-RNN encoder architecture are reported in Table 5.1 with their accuracy and number of parameters. Raw basecaller models with an RNN encoder architecture are reported in Table 5.2 along with their accuracy and the number of parameters they require. Tanh kernel activation and Sigmoid recurrent activation functions are used for all RNN layers. The ReLU activation function is used for CNN layers.

Table 5.1: Raw basecaller models with CNN-RNN encoder architecture

Encoder CNN				Encoder RNN			Decoder			Parameters	Train Acc	Val Acc
Layers	Filters	Filter Size	Stride	Type	Layers	Nodes	Type	Layers	Nodes			
1	100	9	1	LSTM	1	200	LSTM	1	200	405,805	75%	55%
1	100	9	1	LSTM	1	150	LSTM	1	150	244,705	73%	60%
1	96	19	7	GRU	2	100	GRU	1	100	153,309	65%	55%
2	96	19	7	BGRU	1	200	GRU	1	400	1,020,693	60%	60%
1	96	19	7	BGRU	1	100	GRU	1	200	243,709	60%	55%
1	50	9	1	BGRU	1	150	GRU	1	300	456,405	80%	58%
1	50	9	1	BGRU	2	100	GRU	1	200	395,105	70%	50%
1	50	9	1	BGRU	1	100	GRU	1	200	214,505	84%	65%
1	100	9	1	BGRU	2	100	GRU	1	200	425,805	85%	73%
1	100	9	1	BGRU	1	200	GRU	1	400	849,005	83%	63%
1	100	9	1	BGRU	2	200	GRU	1	400	1,570,205	86%	65%

Table 5.2: Raw basecaller models with RNN encoder architecture

Encoder			Decoder			Parameters	Train Acc	Val Acc
Type	Layers	Nodes	Type	Layers	Nodes			
BGRU	1	100	GRU	1	200	184,405	80%	79%
BGRU	2	100	GRU	1	200	365,005	89%	88%
BGRU	1	120	GRU	1	240	264,485	80%	76%
BGRU	1	150	GRU	1	300	411,605	83%	83%
BGRU	2	75	GRU	1	150	206,255	84%	75%
BGRU	1	200	GRU	1	400	728,805	88%	88%

5.3 Network Training

A training dataset of 4000 reads (signals and labels) and a test dataset of 4000 reads were prepared using Metrichor and Nanoraw for training the Chiron [42] basecaller. 2000 reads were randomly chosen from a set of 34,383 reads from a phage Lambda virus sample provided by ONT and 2000 reads were chosen between 15,012 reads from Escherichia coli sequenced by MinION (nanopore version R9.4). The test dataset also contains 2000 reads from each sample. The data files are available at `http://gigadb.org/dataset/100425`. We used the datasets for training and evaluating our basecaller.

We train encoder and decoder sub-models together as a whole model (end-to-end training) pictured in Fig. 5.3. The complete model has 365,005 trainable parameters. We then build encoder and decoder sub-models from the trained layers for the inference step. We used Adam optimizer for minimizing the loss function. We trained the model for 50 steps and used 10000 samples per step.

5.4 Inference

In the inference step, we first use the encoder sub-model to encode the whole input sequence. We then set up the initial state of the decoder network with the last hidden state of the encoder network and start the decoding by inputting the start-

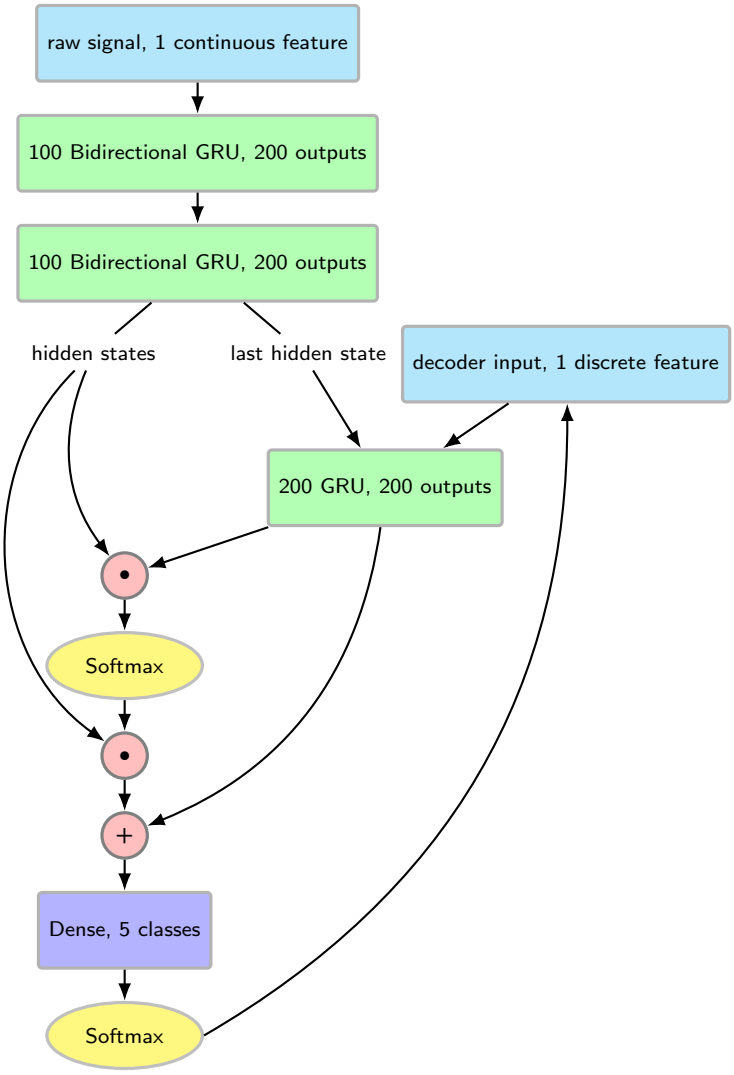


Figure 5.3: The encoder-decoder model with global attention.

of-sequence label. We then feedback the output of the network until the output sequence is complete. We can either use the greedy search or the beam search for decoding.

Greedy Search In the greedy search, at each time we select and feedback the output with the highest probability. As soon as the network outputs an end-of-sequence token, we stop the decoding and return the generated output sequence.

Beam Search In the beam search, instead of only considering the best output (as in greedy search), we calculate and track the k best outputs. k is a beam search parameter, called beam width.

We maintain an incomplete beam list with size k. Each unit of the beam list has three values: score, the incomplete sequence and the last state. We continue decoding all of the units and only preserve k beam units having the highest scores.

We also keep a list of complete sequences. As soon as the network outputs an end-of-sequence token, we add the generated sequence to the complete beam list. When the number of complete sequences reaches the beam width, we stop the decoding process and return the sequence with the highest score.

We use the greedy decoder in our basecaller. The beam search decoder may improve accuracy but reduces the basecalling speed.

Table 5.3: Basecalling accuracy of different basecallers for an E.coli sample.

Basecaller	Read identity
Albacore	90.9
BasecRAWller	82.5
Chiron	90.6
Mauler	90.9
Metrichor	88.6

5.5 Results

To evaluate the accuracy of our model, we basecalled 2000 E. coli reads (K12 MG1655 reference sequence) and 2000 Lambda reads (National Center for Biotechnology Information (NCBI) reference sequence NC_001416.1). We used minimap [51] software to align the Fasta output of our basecaller to the reference genome and calculate the accuracy for each sample.

In Table 5.3, we compare the accuracy of Albacore, BasecRAWller, Chiron, Mauler (our basecaller) and Metrichor for the E. coli sample. Table 5.4 summarizes the accuracy of mentioned basecallers for the Lambda sample. Read identity is the number of matched bases divided by the length of the aligned reference genome.

Table 5.5 compares the speed of Albacore, BasecRAWller, Chiron and Mauler.

Table 5.4: Basecalling accuracy of different basecallers for a Lambda sample.

Basecaller	Read identity
Albacore	89.8
BasecRAWller	81.5
Chiron	87.8
Mauler	81.9
Metrichor	86.5

The CPU rate is the total number of base pairs basecalled divided by the total CPU time. The reported CPU rate of Albacore, BasecRAWller, and Chiron are adapted from [42]. The CPU rate of Mauler was evaluated on a 2018 MacBook Pro with 2.2 GHz Intel Core i7 processor.

5.6 Conclusion

We implemented a raw single-strand basecaller using an encoder-decoder model with attention mechanism. We process 300 long signal windows from the raw input data and shift the window by 30 data points. The encoder network has 2 stacked bidirectional GRU layers (100 units). The decoder network has 1 GRU layer (200 units). We use the dot-based global attention mechanism. The model classifies

Table 5.5: Basecalling speed and network parameters of different basecallers. CPU rate is the ratio of the basecalled nucleotides to the total CPU time for the basecalling. The CPU rate of Albacore, BasecRAWller and Chiron is adapted from [42]. The CPU rate of Mauler was evaluated on a 2018 MacBook Pro with 2.2 GHz Intel Core i7 processor.

Basecaller	CPU Rate	Number of Parameters
Albacore	2,975	379,265
BasecRAWller	81	136,756 + segmentation + 245,996
Chiron	21	2,657,028
Mauler	1,580	365,005

raw input signal to 5 labels: {A, C, G, T, end-of-sequence}. We trained our model (end-to-end) using two datasets (E.coli and Lambda) segmented by the Nanoraw software. For basecalling, we use greedy decoding to find the output sequence with the highest likelihood. We evaluated our basecaller for E.coli and Lambda samples and achieved an inference accuracy of 81.9% for the Lambda sample and an accuracy of 90.9% for the E.coli sample. Our accuracy is comparable to Chiron and Albacore, the best available raw basecallers. Our basecaller has seven times less number of parameters compared to Chiron and approximately the same number of parameters as Albacore.

6 Conclusions and Future Work

The purpose of this thesis was to optimize nanopore-based basecalling. In particular, we focused on studying the effect of noise on single- and double-strand basecallers and proposed and implemented a new approach for performing end-to-end basecalling using the deep learning encoder-decoder model.

We simulated nanopore event data using approximated R9 nanopore characteristics provided by ONT to analyze the effect of noise on single- and double-strand basecalling. We implemented RNN-based single- and double-strand basecallers for the simulated event data. We investigated the behavior of our basecallers as a function of the time-series SNR. The accuracy of the single-strand basecaller decreased significantly from 99% to 90% as the SNR decreased from 20 dB to 5 dB. The accuracy only decreased from 99.8% to 99.3% from an SNR of 20 dB to an SNR of 5 dB for the double-strand basecaller.

All available raw basecallers employ CTC to directly process the raw measured current signal. We implemented a single-strand raw basecaller using attention-

based encoder-decoder model. The encoder network architecture is two stacked bidirectional GRU layers with 100 nodes. We used one GRU layer with 200 nodes for implementing the decoder model. And a global attention mechanism was used to improve accuracy. We trained our model on a bacterial (E. coli) dataset and a viral (Lambda) dataset sequenced by the MinION. We process sliding windows of the read and then assemble the windows in order to improve the basecalling speed and accuracy. To evaluate our model, we basecalled 2000 E. coli reads and 2000 Lambda reads. We used the minimap module to align the basecaller output and the sample reference genome to calculate the accuracy. We achieved an accuracy of 81.9% for the Lambda sample. We achieved 90.9% accuracy for the E. coli data set comparable to the accuracy of Chiron and Albacore. Our basecaller has seven times less number of parameters compared to Chiron.

In the future, we would like to train our basecaller further on more species including a human data set. A valuable future work is the implementation of an encoder-decoder based double-strand basecaller. A dual encoder network [52] can be used for implementing an alignment-free double-strand basecaller.

Another direction is to experiment using both event and raw data in the decision making process using the twin encoder approach proposed by Xiao et al. [53].

Bibliography

[1] Vladimir Boza, Brona Brejova, and Tomas Vinar. Deepnano: Deep recurrent neural networks for base calling in MinION nanopore reads. *Plos One*, 12(6):1–13, 2017.

[2] Christopher Olah. Understanding LSTM networks, Aug 2015. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[3] Harald Scheidl. What is connectionist temporal classification (CTC)?, Sep 2018. accessed on 9 Dec 2018 from `https://stats.stackexchange.com/questions/320868/what-is-connectionist-temporal-classification-ctc`.

[4] James M. Heather and Benjamin Chain. The sequence of sequencers: The history of sequencing DNA. *Genomics*, 107(1):1–8, 2016.

[5] Anthony Rhoads and Kin Fai Au. PacBio sequencing and its applications. *Genomics Proteomics Bioinformatics*, 13:278–289, 2015.

[6] Oxford Nanopore Technologies. MinION, Jan 2018. Retrieved Jan 15, 2018 from `nanoporetech.com/products/minion/`.

[7] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. volume abs/1508.04395, 2015.

[8] Liang Lu, Xingxing Zhang, and Steve Renals. On training the recurrent neural network encoder-decoder for large vocabulary end-to-end speech recognition. *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5060–5064, 2016.

[9] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[10] Geoffrey Hinton. Neural networks for machine learning, Sep 2014. Retrieved Oct 1, 2017 from http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[12] Andrej Karpathy. CS231n convolutional neural networks for visual recognition, April 2018. Retrieved April 1, 2018 from http://cs231n.github.io/neural-networks-2/.

[13] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[14] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowlege-Based Systems*, 6(2):107–116, 1998.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[16] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. pages 1–10. arXiv, 2014.

[17] Andrej Karpathy. Convolutional neural networks (CNNs / ConvNets), April 2018. Retrieved April 1, 2018 from http://cs231n.github.io/convolutional-networks/.

[18] François Chollet. A ten-minute introduction to sequence-to-sequence learning in keras, Sep 2017. https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html.

[19] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 369–376, New York, NY, USA, 2006. ACM.

[20] Awni Hannun. Sequence modeling with CTC, Nov 2017. https://distill.pub/2017/ctc/.

[21] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[22] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[23] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.

[24] Byung-Jun Yoon. Hidden markov models and their applications in biological sequence analysis. *Bioinformatics*, 2009.

[25] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *IEEE*, 77(2):257–286, 1989.

[26] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977.

[27] A M Maxam and W Gilbert. A new method for sequencing DNA. *Proceedings of the National Academy of Sciences*, 74(2):560–564, 1977.

[28] Mehdi Kchouk, Jean-Francois Gibrat, and Mourad Elloumi. Generations of sequencing technologies: From first to next generation. *Biology and Medicine*, 9, 2017.

[29] Mari Miyamoto, Daisuke Motooka, Kazuyoshi Gotoh, Takamasa Imai, Kazutoshi Yoshitake, Naohisa Goto, Tetsuya Iida, Teruo Yasunaga, Toshihiro Horii, Kazuharu Arakawa, Masahiro Kasahara, and Shota Nakamura. Performance comparison of second- and third-generation sequencers using a bacterial genome with two chromosomes. *BMC genomics*, 15:699, 08 2014.

[30] Pacific Biosciences of California. SMRT sequencing, May 2018. Retrieved May 5, 2018 from `www.pacb.com/SMRT-science/smrt-sequencing/`.

[31] Alberto Magi, Roberto Semeraro, Alessandra Mingrino, Betti Giusti, and Romina D'Aurizio. Nanopore sequencing data analysis: state of the art, applications and challenges. 06 2017.

[32] Yanxiao Feng, Yuechuan Zhang, Cuifeng Ying, Deqiang Wang, and Chunlei Du. Nanopore-based fourth-generation DNA sequencing technology. *Genomics, Proteomics and Bioinformatics*, 13(1):4 – 16, 2015.

[33] Oxford Nanopore Technologies. Products, Oct 2018. Retrieved Oct 12, 2018 from `nanoporetech.com/products`.

[34] Oxford Nanopore Technologies. Products, Oct 2018. Retrieved Oct 12, 2018 from `nanoporetech.com/products/gridion`.

[35] Oxford Nanopore Technologies. Products, Oct 2018. Retrieved Oct 12, 2018 from `nanoporetech.com/products/promethion`.

[36] Oxford Nanopore Technologies. Primary data analysis, Oct 2018. Retrieved Oct 12, 2018 from `nanoporetech.com/analyse#complete`.

[37] Jain M, Tyson JR, and Loose M. MinION analysis and reference consortium: Phase 2 data release and analysis of R9.0 chemistry. May 2017.

[38] Marcus Stoiber and James Brown. Basecrawller: Streaming nanopore basecalling directly from raw signal. pages 1–15. bioRxiv, 2017.

[39] Matei David. Packing ont fast5 files, Feb 2018. Retrieved Feb 1, 2018 from `http://simpsonlab.github.io/2017/02/27/packing_fast5/`.

[40] Ryan R. Wick, Louise M. Judd, and Kathryn E. Holt. Comparison of Oxford nanopore basecalling tools, May 2018. Retrieved May 1, 2018 from `github.com/rrwick/Basecalling-comparison`.

[41] Robert Vaser, Ivan Sovic, Niranjan Nagarajan, and Mile Sikic. Fast and accurate de novo genome assembly from long uncorrected reads. *bioRxiv*, 2016.

[42] Haotian Teng, Minh Duc Cao, and Michael B. Hall. Chiron: Translating nanopore raw signal directly into nucleotide sequence using deep learning. pages 1–10. bioRxiv, 2017.

[43] Matei David, L. J. Dursi, Delia Yao, and Paul C. Boutros. Nanocall: An open source basecaller for Oxford nanopore sequencing data. *Bioinformatics*, 33(1):49–55, 2017.

[44] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[45] Senol Cali, Kim JS, Ghose S, Alkan C, and Mutlu O. Nanopore sequencing data analysis: state of the art, applications and challenges. *Briefings in bioinformatics*, 04 2018.

[46] Marcus H Stoiber, Joshua Quick, Rob Egan, Ji Eun Lee, Susan E Celniker, Robert Neely, Nicholas Loman, Len Pennacchio, and James B Brown. De novo identification of DNA modifications enabled by genome-guided nanopore signal processing. *bioRxiv*, 2016.

[47] Scott Gigante. In-house training of the nanonet local basecaller: opportunities and challenges, May 2017.

[48] Clive Brown. Remarks by Oxford nanopore technologies CTO Clive Brown at the London Calling conference on nanopore sensing., May 2015. Retrieved May, 2015 from `events.nanoporetech.com/events/london-calling-2015/videos/view/244/`.

[49] Winston Timp, Jeffrey Comer, and Aleksei Aksimentiev. DNA basecalling from a nanopore using a Viterbi algorithm. *Biophysical*, 102(10):L37 – L39, 2012.

[50] Wanasit. Attention-based sequence-to-sequence in keras, Sep 2017. `https://wanasit.github.io/attention-based-sequence-to-sequence-in-keras.html`.

[51] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[52] S. SharathT., Shubhangi Tandon, and Ryan Bauer. A dual encoder sequence to sequence model for open-domain dialogue modeling. *CoRR*, abs/1710.10520, 2017.

[53] Shuai Xiao, Junchi Yan, Mehrdad Farajtabar, Le Song, Xiaokang Yang, and Hongyuan Zha. Joint modeling of event sequence and time series with attentional twin recurrent neural networks. *CoRR*, abs/1703.08524, 2017.