Contents

List of Figures

List of Tables

Compression of Basecalling Models in Genome Sequencing

Abstract

The genome of an organism consists of a few million to billions of base pairs. Oxford Nanopore sequencing works by monitoring changes in electrical current and the resulting signal is basecalled using a neural network to produce the Deoxyribonucleic Acid (DNA) sequence.

Deep Learning operations are computationally intensive because they involve multiplying tensors (which are multi-dimensional matrices). We use methods like pruning and quantization to lower the amount of computation and reduce the size of the model. Neural networks consist of weight and bias tensors, and with the help of pruning, the insignificant weights are removed leading to a lesser number of multiplications. Normally these weights and biases are stored in 32-bit format, but with the help of quantization, this can be brought down to 8-bits. Using both techniques simultaneously we achieved an overall size reduction of 75% without significant loss in accuracy.

We also managed to reduce the number of multiplications significantly after pruning and quantization.

Acknowledgements

We would like to thank our mentor Dr. Madhura Purnaprajna, Professor, Department of ECE, for guiding us throughout the course of this project.

We would also like to express our deepest gratitude to Dr. Ajey SNR, Chairperson, Department of ECE, for giving us constant support and encouragement throughout this project.

We would also like to extend our gratitude to Dr. Subhash Kulkarni, Principal, PES University Electronic City Campus, for giving us this great opportunity to work on this project.

1 Introduction

A genome is an organism's complete set of DNA and Genomics is the branch of molecular biology concerned with the structure, function, evolution and mapping of genomes. With the help of genomics, we can determine complete DNA sequences and perform genetic mapping to help understand diseases and identify genetic mutations.

Oxford Nanopore's devices measure the changes in electrical current. As a single stranded molecule of DNA passes through the nanopore, it helps in determining the nucleotide sequence of the DNA strand with the help of neural networks. This process is known as basecalling. Since the genome of an organism consists of a few million to billion base pairs, genome sequencing takes a significant amount of time to complete.

In this project, we aim to compress the basecalling models in order to perform sequencing in a faster manner by reducing the number of multiplications, without compromising the accuracy of the sequence obtained.

2 Literature Survey

Genome sequencing is increasingly used in healthcare and research to identify genetic variations. A DNA strand essentially has 4 nucleotide bases, namely - Adenine (A), Guanine (G), Cytosine (C) and Thymine (T). These 4 bases are arranged in a double helix structure to form a strand of DNA.

Oxford Nanopore Technologies (ONT) [1] has developed a new generation of DNA/Ribonucleic acid (RNA) sequencing technology that offers real time analysis. This technology is fully scalable, can analyse DNA or RNA and sequence any length of fragment to achieve short to ultra long reads. They also have an open source tool called taiyaki [2] from which we obtained the basecalling model.

During our research, we found that in the process of genome sequencing, most of the time is consumed by basecalling. So in order to bring this time down we can reduce the amount of computations involved in the neural network. We mainly focused on reducing this heavy network to obtain a more light-weight form of the same. But there is a

trade-off between the amount of reduction achievable and the accuracy of the model.

There are two key types of accuracy in DNA sequencing technologies - Read Accuracy and Consensus Accuracy [3]. A good consensus accuracy but a lower read accuracy is fine, because in the end we want the entire sequence to be constructed so the individual read alignments won't matter a lot. Oxford Nanopore's model has a consensus identity of 99.90%.

Long Short Term Memory (LSTM) [4] networks are a type of Recurrent Neural Network (RNN) capable of learning order dependence in sequence prediction problems. This cell remembers values over arbitrary time intervals and has 3 gates which regulate the flow of information into and out of the cell.

Gated Recurrent Units (GRU) [5] are an improved version of RNN's i.e., they can be trained to retain information obtained from long ago, without removing information which is relevant to the prediction. These perform faster compared to LSTM networks because GRU uses lesser training parameters and hence uses lesser memory.

The neural network that we will be using throughout this project can be obtained in both LSTM and GRU variants. Since GRU models use lesser memory we chose it to perform the reduction techniques.

3 Main work

First it is important to learn how the genomics pipeline works and all the different terms involved. Then we move on to the two types of size reduction techniques that were applied to reduce the size of the neural network.

3.1 Pipeline

We have followed 2 different Pipelines for this project [6]-

1. Raw Reads → Basecalling → Assembly → Polishing
2. Raw Reads → Basecalling → Assembly → Variant Calling

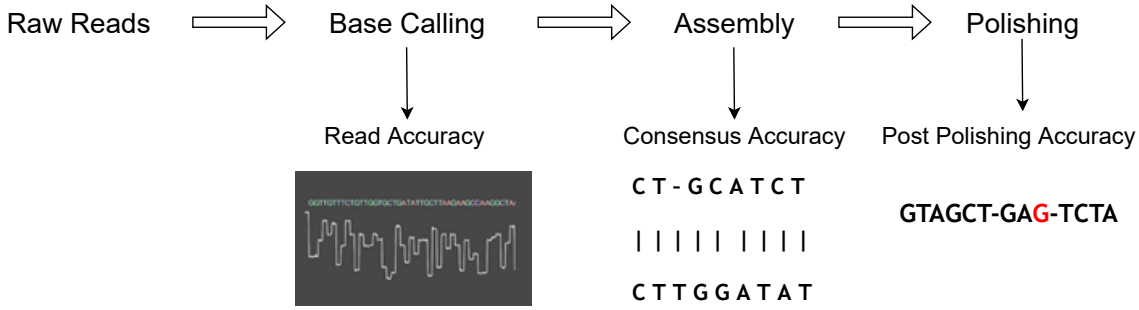

Figure 3.1: Pipeline 1 with accuracy metric after each stage

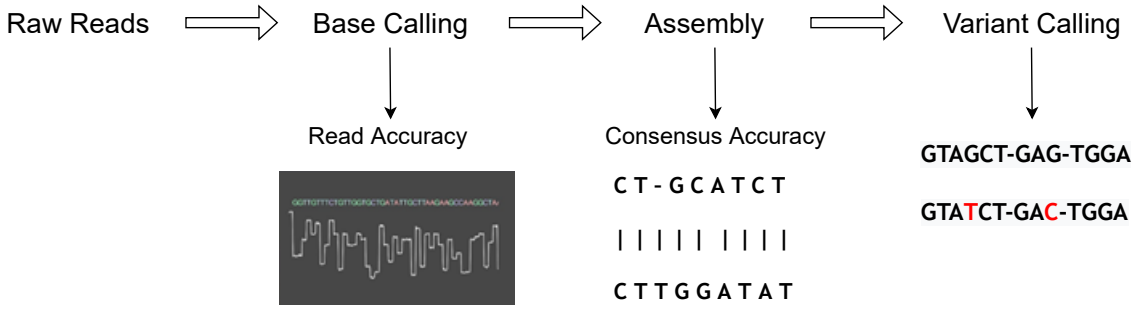

Figure 3.2: Pipeline 2 with accuracy metric after each stage

Raw Reads-

Raw Reads are the raw electrical signals that were produced from the sequencing machine. These are usually stored in .fast5 file format. The .fast5 files that were used in this project were from the organism - *Klebsiella pneumoniae*.

Basecalling- Basecalling is a process which translates the raw electrical signals from the ONT sequencer into nucleotide bases using RNN's. The tool used for basecalling was Guppy. We obtain a .fastq file once basecalling is completed.

Assembly- During sequencing, the DNA is chopped up into small fragments called reads. The assembler then takes these small sequences and reassembles them to reconstruct the original sequence by aligning and merging. This process is known as assembly. The aligners used to perform assembly were minimap, Racon and Rebaler [7]. We obtain a BAM or SAM file after assembly is completed. There are 2 ways to perform assembly -

1. **Reference Assembly-**
    - This method assembles reads into larger units by mapping the reads against a reference sequence/genome.

2. **De Novo Assembly-**
    - This method assembles reads without using any kind of template or reference sequence/genome.

Polishing-

Polishing is the process of removing/correcting the inconsistencies/errors that are formed in the assembled sequence. This process is also know as genome finishing. The output of polishing is a .fasta file. The polisher used was nanopolish [7].

Variant Calling-

A variant is a different version/form of the same organism. The process by which we can identify variants from the sequence data is known as variant calling. The output of this process is a .vcf file. The variant caller used was Clair [8].

3.2 Basecalling model

Guppy is a data processing toolkit that contains ONT's production basecalling algorithms and several bioinformatic post-processing features. For this project, Guppy basecaller has been used. Guppy models are based on RNN's. Guppy comes with three models - 'Fast', 'HAC' and 'sup'. These models are not open-source.

Taiyaki [2] is an open-source research software used to train the basecalling models that are used by Guppy. Taiyaki provides pre-trained (GRU) and (LSTM) models along with all the tools and data necessary to perform training/retraining on the models. The reduction techniques were performed on the GRU model as the memory footprint is lower than LSTM.

3.2.1 Network Structure

The GRU model consists of a total of 7 layers in the following order (from input side):

1. Convolution
2. Backward GRU
3. Forward GRU
4. Backward GRU

5. Forward GRU
6. Backward GRU
7. Linear

The convolution layer has an insize of 1, size of 96, kernel_size of 19 and a stride of 4. Each GRU layer has been defined with an input and hidden size of 96. The linear layer has an input size of 96 and output size of 40

3.3 Pruning

Pruning is a process by which we can reduce the size of the network by removing parameters like weights or neurons as shown in Figure 3.3 [9]. Pruning neurons means that we are reducing the size of the dimensions of the layers. This would mean the number of weights reduce. In weight pruning we set the insignificant weights from the weight matrix to 0. The most popular type of weight pruning is magnitude pruning where the weights with the smallest magnitude are set to 0 [10]. By doing so, we are introducing sparsity into the weight matrices. If efficient sparse $\times$ dense matrix multiplication techniques are used, then this can lead to a reduction in computation time as well.

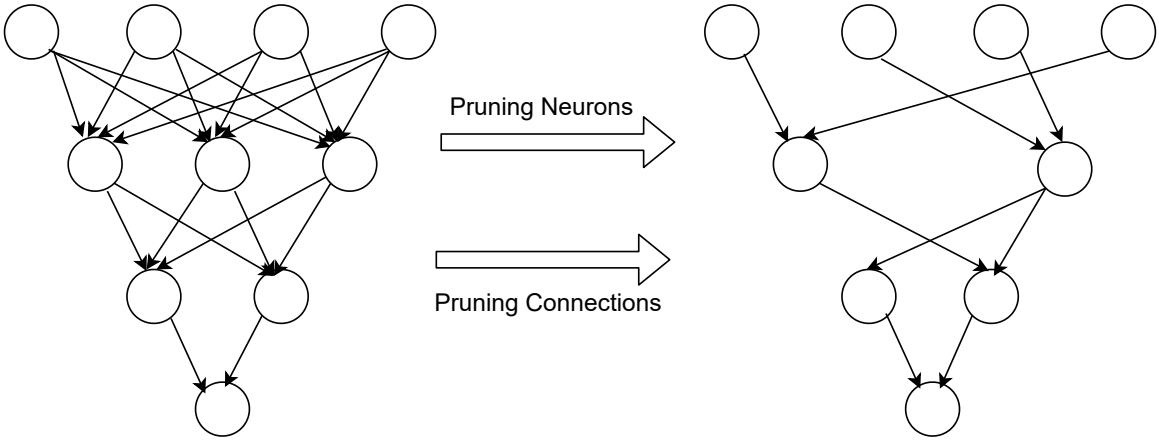

Figure 3.3: Pruning of neurons and connections

In most neural networks the majority of the weights are concentrated very close to 0. This can be observed in Figure 3.4 which shows the distribution of weights. The amount of sparsity that can be introduced into a network depends on its sensitivity. Some layers are less sensitive than others and can be pruned to a higher degree. Generally the layers closer to the input are the most sensitive and the layers close to the output are the least sensitive.

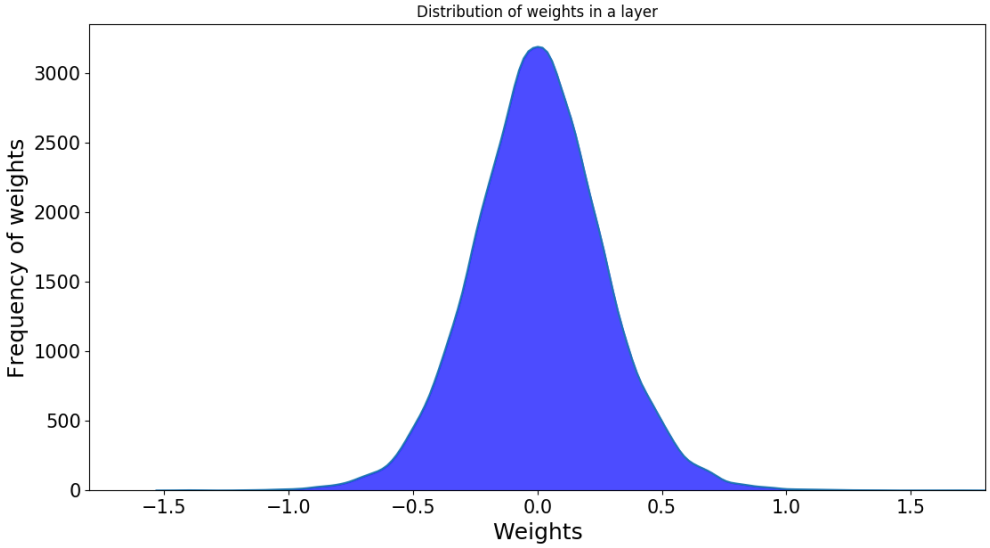

Figure 3.4: Distribution of weights in a layer

PyTorch provides functions to perform pruning and generating masks. A mask is a matrix which has the same dimension as a weight matrix that is to be pruned. It consists only of 0's and 1's. So if an element wise multiplication is performed between the mask and weight matrix, the locations in the weight matrix, where a 0 is present in the mask, will also become 0 [11]. As shown in Figure 3.5, when the weight tensor is multiplied with the mask, we get the pruned weight tensor.

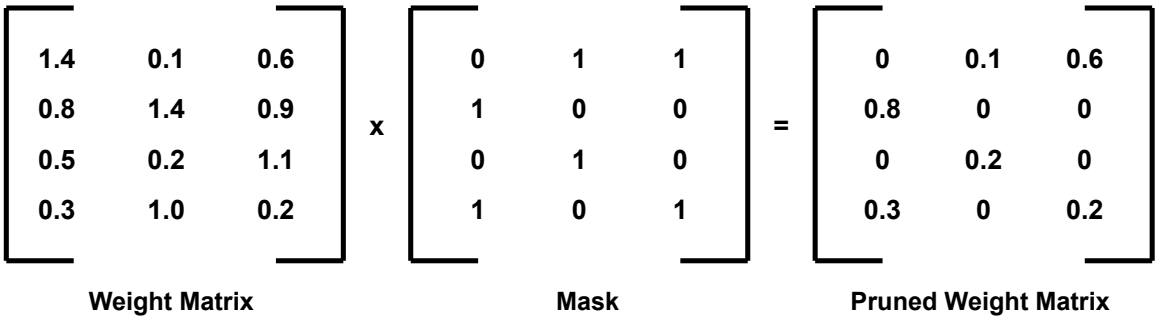

Figure 3.5: Multiplying a mask with the weights (element wise) introduces sparsity

The pseudocode for applying the mask is given below -

```
L = list()
for weight in weight_tensor:
  L.append(abs(weight))
L.sort()
limit = L[ceil(sparsity*len(L))]
```

```
for weight in weight_tensor:
  if abs(weight) < limit:
    weight = 0
```

3.3.1 Retraining with pruning

When pruning was performed on the input layer or the output layer, a huge loss in accuracy was observed, so pruning was performed only on the intermediate layers (GRU layers). When any layer was pruned and retrained, all the other layers were frozen. Starting from the last GRU layer we moved backwards to the first GRU layer.
During training/retraining, there is the forward propagation phase where the inputs are given to the network and then there is the back propagation phase where the weights and biases are updated based on the gradient calculated. This will cause the 0's to become non-zero, thereby removing sparsity. To overcome this, the mask needs to be multiplied with the weights when retraining is taking place.

3.4 Quantization

The dominant precision format used for storing the weights and biases of a neural network is 32-bit floating point. Quantization is a process by which we can store these weights and biases in lower precision formats, which will lead to an overall reduction in the size of the model without any significant loss in accuracy.
PyTorch provides multiple techniques to quantize a deep neural network. The two most popular modes of quantization that are provided by the 'torch.quantization' module are:

1. **Dynamic Quantization**-

   Dynamic Quantization converts a network to lower precision format to save on the model size [12]. It converts all the floating point weights and activations of the pre-trained model to integer (8-bit) format using 'torch.quantization.quantize_dynamic' API. This built-in API decides the scale factor for the weights and activations dynamically based on the data range and ensures that the chosen scale factor is tuned to preserve the accuracy of the quantized model.

   ```
   quantized_model = torch.quantization.quantize_dynamic(model,
   dtype = int8)
   ```

Although the above quantization technique resulted in a reduction of the size of the model parameters by a factor of 4, we could not save/load the quantized model for further operations using 'torch.save' and 'torch.load' (Which is the built-in function provided by PyTorch to save and load deep neural network models).

2. **Static Quantization**-
   In static quantization, compression takes place by fusing the modules of a pretrained model using 'torch.quantization.fuse_modules' API [12]. The resulting precision format of the weights and activations of the fused modules is 8-bit integer.

```
model_fused = torch.quantization.fuse_modules(model, [['conv',
'linear']])
```

   The above technique can help in fusing the 'Convolution' and 'Linear' modules but not the GRU layers of our model. Hence we wrote a code to do 8-bit quantization on all the weights and biases of our pre-trained model, which includes all the GRU layers as well.

Quantization can be performed post training or during training of neural network models. The different post training quantization methods that we have used are as follows:

3.4.1 16-bit Floating Point Quantization

Before diving into integer quantization, we worked towards achieving a significant reduction in the size of the model by keeping the precision of the weights and biases as floating point. We used a built-in function called 'half()' to convert all the 32-bit floating point weight and bias tensors of the pre-trained model to 16-bit floating point tensors. The 'half()' function reduces the size of the model (upto 50%) while having minimal to no impact on the accuracy of the model. The 32-bit model has a size of 1158 KB and the 16-bit model has a size of 579 KB.

```
quantized_model = model.half()
```

3.4.2 Integer Quantization

After achieving around 50% reduction in the size of our model with 16-bit floating point quantization, we worked on lowering the bit-widths of weights and biases to 8-bit integer format to gain more amount of compression. We used Scale Quantization [13] [14]

to map between the quantized and real number line. In this quantization scheme, for a set of real values, we map the minimum and maximum real values in the range (min, max) to minimum and maximum integer values (-$2^{(N-1)}$, $2^{(N-1)}$ -1) respectively (where *N* is the number of bits) as shown in Figure 3.6. Scale and zero point are used to scale and shift the real number line to the quantized number line. Zero point 'z' is an integer which will always map to the zero of the quantized number range.

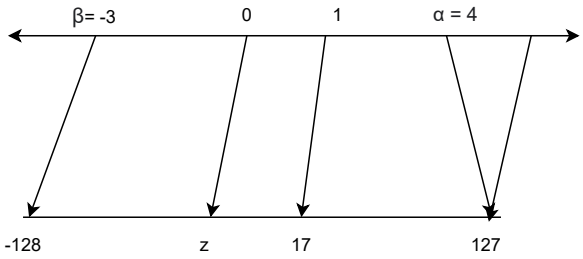

Figure 3.6: Scale Quantization

Some of the operations in the genomics pipeline require the model to be in floating point format. Since we know the scale and zero-point used in quantization, we can dequantize the model which will return values that lie very close to the original floating point values. We use equation 3.1 to quantize the tensor and equation 3.3 to dequantize the tensor.

$$W_Q = \frac{W_f}{scale} + zeropoint \tag{3.1}$$

$$scale = \frac{max - min}{q_{max} - q_{min}} \tag{3.2}$$

$$W_f = (scale * W_Q) - zeropoint \tag{3.3}$$

In equation 3.2, max and min are the maximum and minimum values of the weight tensor (before quantizing) and q_{max} and q_{min} represent the range of integer values. For example if we are using 8-bits then the range is (-128, +127). Zero point is the integer value to which the real value zero is mapped as shown in Figure 3.6. W_Q and W_f are the quantized tensor and floating point tensor respectively.

```
qmin = -2.**(num_bits - 1)
qmax = 2.**(num_bits - 1) - 1
scale = (max_val - min_val) / (qmax - qmin)
initial_zero_point = qmin - min_val / scale
```

```
zero_point = 0
if initial_zero_point < qmin:
    zero_point = qmin
elif initial_zero_point > qmax:
    zero_point = qmax
else:
    zero_point = initial_zero_point
Quantized_tensor = zero_point + fp_tensor / scale
fp_tensor = scale * (Quantized_tensor - zero_point)
```

The 32-bit model has a size of 1158 KB and the 8-bit model has a size of 289.5 KB.

We worked towards further lowering the bit-widths beyond 8-bits. With respect to the original training loss of our model (0.08), the loss of the quantized models before and after retraining are given in Table 3.1 The maximum compression with 4-bit integer quantization was 87.50%, but none of the reads could be aligned with the reference sequence (for any bit-width below 8) during assembly, which resulted in significant loss in accuracy.

Bit-width	Loss before retraining	Loss after retraining	Compression
7	1.536	0.320	78.13%
6	1.073	0.672	81.25%
5	0.593	0.593	84.38%
4	1.135	0.745	87.50%

Table 3.1: Training loss for lower bit-widths

3.4.3 Ternary Connect Quantization

Ternary Connect Quantization is a higher compression quantization technique which reduces the bit-width to 2-bits. In this method we map a tensor with 32-bit precision weights to a tensor with values in the range (-1, 0, 1) [15].
Weights are mapped according to the following thresholds-

$$W = \begin{cases} -1, & w \leq -0.5 \\ 0, & -0.5 < w \leq 0.5 \\ 1, & w > 0.5 \end{cases} \tag{3.4}$$

According to Figure 3.4, we can see that most of the weights of our model lie in the range

of -1 to 1. Hence we used this deterministic approach to quantize all the weights and biases of our model to 2-bits. This resulted in higher loss in accuracy since it involves direct mapping of the floating point weight values to -1, 0 and 1. The loss for this model was found to be 1.271 upon retraining whereas the original loss of the model is 0.08.

4 Results

In order to assess the network after pruning or quantization we need to look at the accuracy. Values for accuracy can be obtained after basecalling, assembly and polishing. These accuracy values are known as 'Read Identity', 'Assembly Identity' and 'Post-Nanopolish Identity' respectively [7]. The results have been summarized in Table 4.1.

Pruned version 1 or Pruned (v1) has a sparsity of 25% on all the input layer weights (of GRU) and a sparsity of 50% on the hidden layer weights. Pruned version 2 or Pruned (v2) has a sparsity of 40% on all the input layer weights (of GRU) and a sparsity of 60% on the hidden layer weights.

We can see from Table 4.1 that the identity/accuracy values obtained are lesser for the pruned model when compared to the original model. This was expected since higher the sparsity, lower the accuracy. This is because during pruning, some of the weights become 0 due to which some information is lost as the input passes through the different layers. So when the sequence is formed during basecalling, it has some gaps and inconsistencies. This makes it difficult to reconstruct the sequence during assembly and results in a slightly lesser accuracy.

Pruned(v1) + Q16 and Pruned(v2) + Q16 are the models where both pruning and 16-bit floating point quantization was performed. We can see that the results obtained for these two models show a similar trend as seen in Pruned(v1) and Pruned(v2) models.

The Quantized (8-bit) model takes up only a fourth of the memory when compared to the original model, but most of the operations in the genomics pipeline require the model parameters to be in floating point format. So the weights were dequantized back to floating point values before basecalling was started (as a one time operation). Upon dequantiza-

tion, the values of weights and biases obtained were very close to the original weight and bias values and it resulted in good read and assembly identity values as shown in Table 4.1.

We had also performed pruning on the 8-bit quantized model (Pruned + Q8) but it resulted in higher loss in accuracy as none of the reads could be aligned with the reference.

4.1 Accuracy Plots

We have used violin plots to represent the accuracy obtained from the different compression techniques/models. A violin plot is a method of plotting numeric data which shows the full distribution of data. The horizontal width is proportional to the frequency of values that lie in the corresponding range shown by the y-axis.

4.1.1 Read Accuracy

Read accuracy [3] is the error rate of individual measurements (reads) from a DNA sequencing technology. This is obtained after basecalling. In Figure 4.1, we have plotted the different models vs their read accuracy. As sparsity increases, the accuracy decreases. Even though read accuracy is lower than the original model, we get a high consensus and post-nanopolish accuracy.

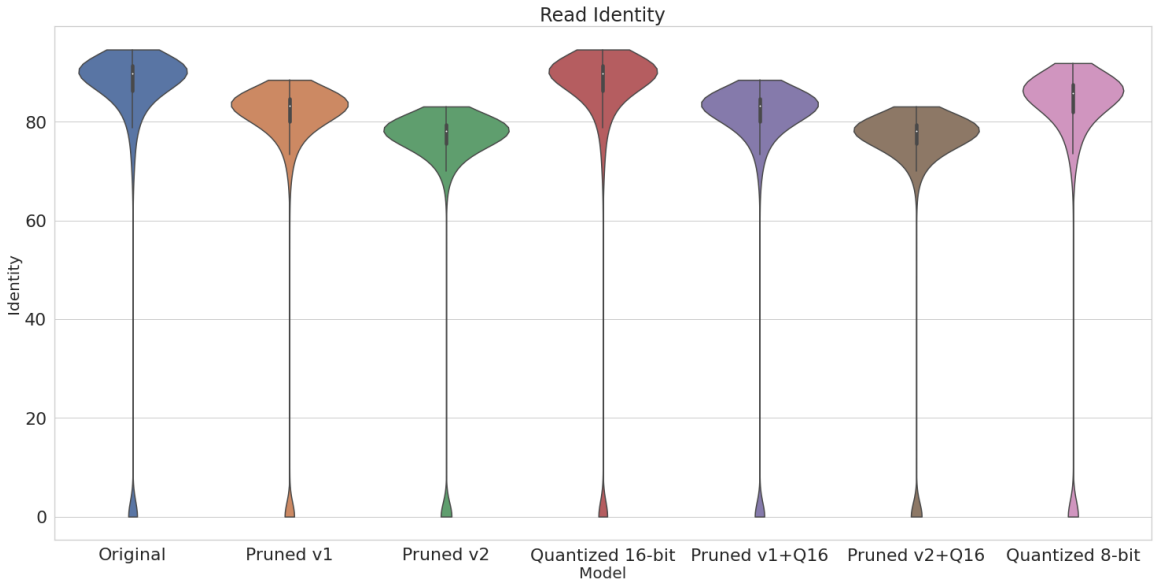

Figure 4.1: Read Accuracy

Model	Sparsity	Compression	Median Accuracy
Original	-	-	Read - 89.75% Assembly - 99.58% Nanopolish - 99.90%
Pruned (v1)	36.36%	36.36%	Read - 83.18% Assembly - 98.52% Nanopolish - 99.76%
Pruned (v2)	48.48%	48.48%	Read - 78.10% Assembly - 96.13% Nanopolish - 99.21%
Quantization (16-bit)	-	50%	Read - 89.75% Assembly - 99.58% Nanopolish - 99.90%
Pruned (v1) + Q16	36.36%	68.18%	Read - 83.18% Assembly - 98.52% Nanopolish - 99.76%
Pruned (v2) + Q16	48.48%	74.24%	Read - 78.10% Assembly - 96.06% Nanopolish - 99.22%
Quantization (8-bit)	-	75%	Read - 85.75% Assembly - 96.06% Nanopolish - 99.85%

Table 4.1: Accuracy comparison for different compression techniques

4.1.2 Assembly Accuracy

Consensus accuracy (also known as Assembly Accuracy) [3] is determined by combining information from multiple reads in a data set, which eliminates any random errors in individual reads. This is obtained after assembly and alignment. In Figure 4.2, we can see that the accuracy values lie between 80% - 100% for all except the Pruned v2 models.

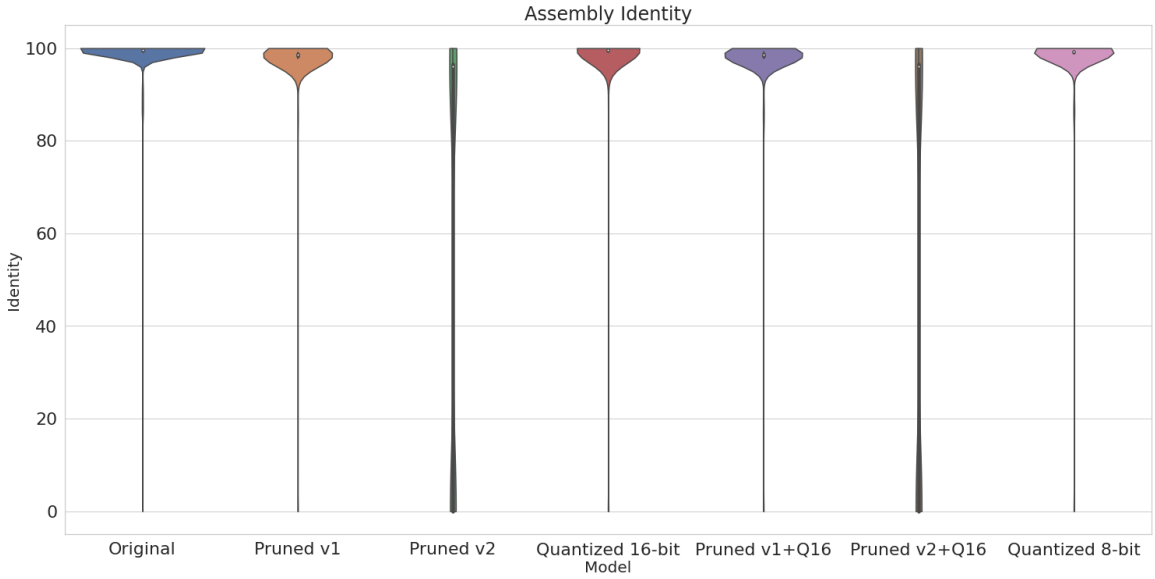

Figure 4.2: Assembly Accuracy

4.1.3 Post-Nanopolish Accuracy

Post-Nanopolish Accuracy is the accuracy obtained after polishing. In Figure 4.3, we can see that the accuracy values lie between 95%-100% even though the read accuracy was lower for the pruned models.

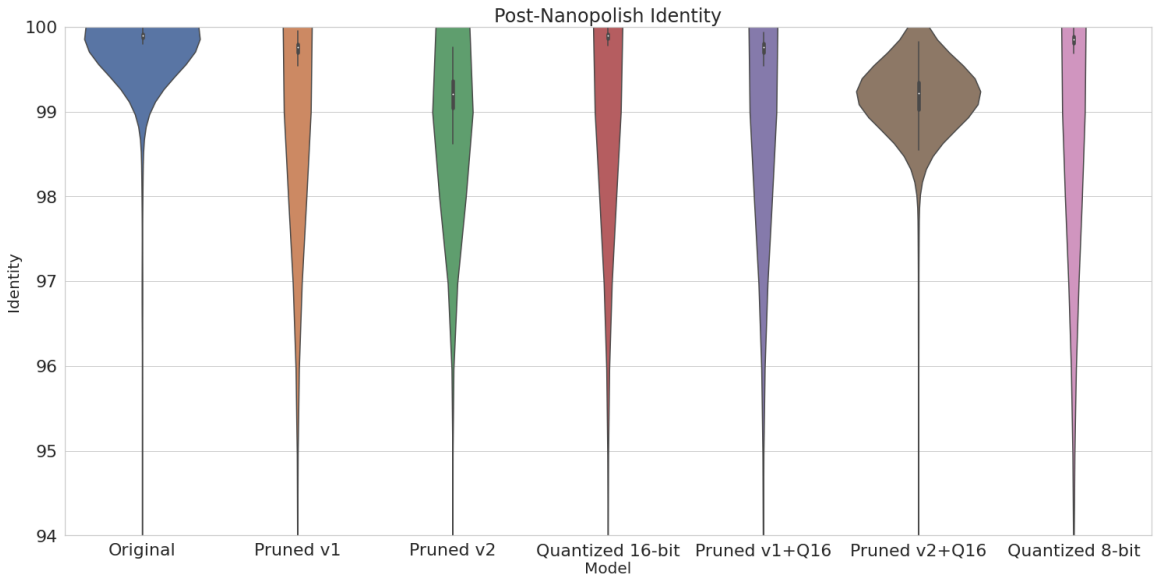

Figure 4.3: Post-Nanopolish Accuracy

4.1.4 Size Compression

We can see the size compression obtained for different models compared to the original model in Figure 4.4.

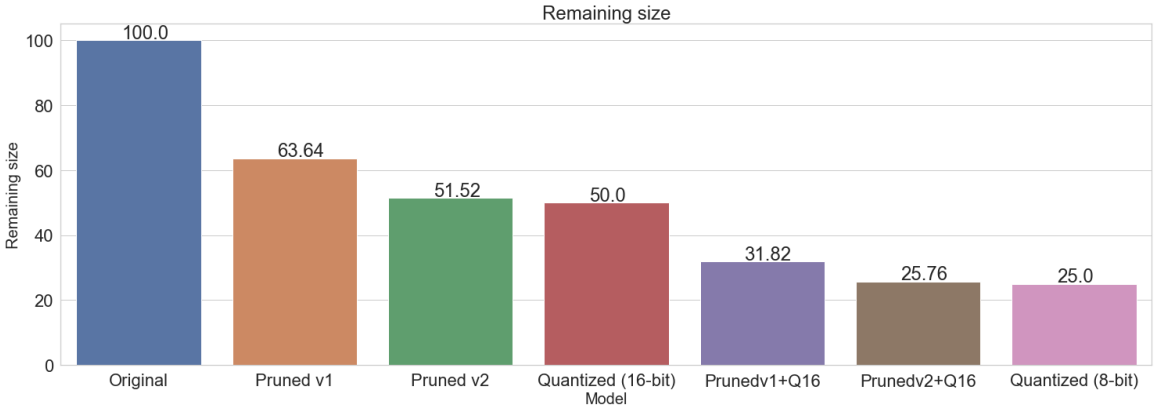

Figure 4.4: Size of the model after different reduction techniques

4.2 Hardware Implementation

As mentioned before, neural networks involve a lot of matrix-matrix multiplication. In order to understand the impact on computation we need to understand what operations take place in the network. Since only the GRU layers were pruned we will understand how a GRU cell works.

GRU networks are an improvement over standard recurrent neural networks and can be considered as a variation of LSTM due to their similarities. GRU uses two gates, the 'Update Gate' and the 'Reset Gate' [5]. GRU's can be trained to retain information that is relevant and discard the irrelevant information.

A diagrammatic representation of a GRU cell is shown in Figure 4.5. Each cell takes in two inputs, one is the hidden state from the previous time step h_{t-1} and the other is the input data for the current time step x_t. For the first time step, h_{t-1} will be zero. Equations 4.1, 4.2 and 4.3 are used to compute the output of reset gate (r_t), update gate (z_t) and hidden state (h_t) respectively.

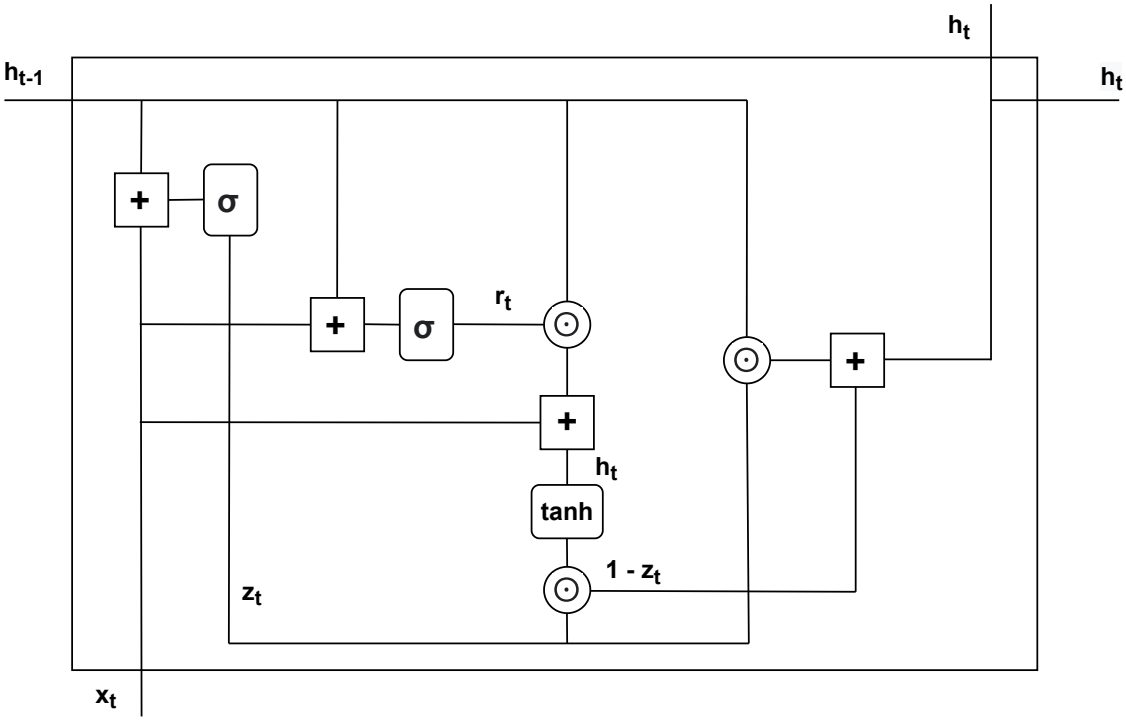

Figure 4.5: A GRU cell

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) \tag{4.1}$$

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}) \tag{4.2}$$

$$h_t = tanh(Wx_t + r_t \odot Uh_{t-1}) \tag{4.3}$$

Where W and U are the weight matrices and $\odot$ represents the Hadamard product (element wise matrix multiplication).

$W^{(z)}$ and $U^{(z)}$ are the weight matrices corresponding to the update gate z_t, and $W^{(r)}$ and $U^{(r)}$ are the weight matrices corresponding to the reset gate r_t.

Now we need to look at how the GRU cell is implemented in PyTorch [16] as taiyaki uses PyTorch. The weights are first broadly divided into 'input weights' (GRU.weight_ih_l[k]) and 'hidden weights' (GRU.weight_hh_l[k]). The input weight tensor has a size of (3*hidden size, input size) and the hidden weight tensor has a size of (3*hidden size, hidden size). In our model both were (288, 96) as the input and hidden size were 96.

Efficient sparse $\times$ dense matrix multiplication techniques can now be used on the pruned model to reduce the number of multiplications. Pruned (v1) shows a reduction of 32% in the number of multiplications when compared to the original model and Pruned (v2) shows a reduction of 45%.

5 Conclusion and Future Scope

From Table 4.1 we can see that in the case of Pruned v1 and Pruned v2, there is some difference in Read Accuracy, but once the whole pipeline is completed, we get a high Post-Nanopolish Accuracy.

16-bit floating point quantized model showed no change in Read and Assembly identity values with respect to the original model. With this we can conclude that 32-bit precision floating point format is not required to store the weights and biases of the model.

We had also performed variant calling using the variant caller 'Clair3' on the different models and got the list of variants present in the sequence. Further, we evaluated each of the .vcf files and obtained the F1 scores, precision and recall. These are the metrics used to evaluate the variants in a sequence. But we got a lower F1 score than expected. Evaluation was done with the help of a python script known as 'hap.py' [17].

As mentioned previously, the processes in the pipeline have been performed on the organism - *Klebsiella pneumoniae*. The whole pipeline is not restricted only to this organism but can also be applied to genomes of other organisms.

The neural network (after pruning) can be stored on the memory of an accelerator (Such as a Field Programmable Gate Array (FPGA)) in sparse matrix formats such as Compressed Sparse Row (CSR) or Coordinate Format (COO). This way only the non-zero weights will get multiplied and the entire computation will take lesser number of clock cycles. If the quantized model is stored on chip, then it would require only a fourth of the memory when compared to the unquantized model. By using the pruned + quantized model, we lower the memory and the number of clock cycles required to compute the output.

On the FPGA, the neural network can be stored in the on-chip Block Random Access Memory (BRAM) which can be implemented using an Intellectual Property (IP). The multipliers and adders which will be required can also be implemented using IP's. Multipliers usually use Digital Signal Processing (DSP) slices, and there are a finite number of slices on an FPGA board. Most of the slices can be utilized in the design which would allow for maximum parallelism and lower latency.

References

[1] (2021) Oxford nanopore technologies. [Online]. Available: https://nanoporetech.com

[2] Taiyaki. [Online]. Available: https://github.com/nanoporetech/taiyaki

[3] (2020) Sequencing 101 : Understanding accuracy in dna sequencing. [Online]. Available: https://www.pacb.com/blog/understanding-accuracy-in-dna-sequencing/

[4] (2020) A gentle introduction to long short-term memory networks. [Online]. Available: https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/

[5] (2017) Understanding gru networks. [Online]. Available: https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be

[6] (2021) Common workflows in genomics. [Online]. Available: https://genomicsbench.eecs.umich.edu/

[7] Performance of neural network basecalling tools for oxford nanopore sequencing. [Online]. Available: https://github.com/rrwick/Basecalling-comparison

[8] Clair3. [Online]. Available: https://github.com/HKU-BAL/Clair3

[9] (2020) Experiments in neural network pruning (in pytorch). [Online]. Available: https://olegpolivin.medium.com/experiments-in-neural-network-pruning-in-pytorch-c18d5b771d6d

[10] V. Sanh, T. Wolf, and A. M. Rush, "Movement pruning : Adaptive sparsity by fine-tuning," 2020.

[11] Experiments in neural network pruning (in pytorch). [Online]. Available: https://github.com/olegpolivin/pruningExperiments

[12] (2020) Pytorch documentation(v1.90). [Online]. Available: https://pytorch.org/docs/1.9.0/torch.quantization.html

[13] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," pp. 3–4, 2020.

[14] K. Chahal. (2019) Mnist 8-bit quantization pytorch. [Online]. Available: https://karanbirchahal.medium.com/

[15] N. M. Rezk, M. Purnaprajna, T. Nordström, and Z. Ul-Abdin, "Recurrent neural networks: An embedded computing perspective," pp. 12–13, 2019.

[16] (2020) Pytorch documentation(v1.90). [Online]. Available: https://pytorch.org/docs/1.9.0/generated/torch.nn.GRU.html

[17] Haplotype comparison tools. [Online]. Available: https://github.com/Illumina/hap.py#happy