Lossless data compression for genome sequences using recurrent neural networks

Arnoud De Jonge Ewout Vlaeminck

Abstract

Genomic data takes up a lot of storage space and more and more genomes are getting sequenced. Previous researchers have used the DeepZip algorithm to perform lossless compression on human mitochondrial genome sequences. We take a closer look at how DeepZip works and how capable it is at performing lossless compression. We compare the impact of different model architectures and sizes on compression factor, training time, compression and decompression time. We propose a few methods to potentially speed up the compression and decompression process. Finally, we test if training a model on the specific file we want to compress can outperform compression with a traditional model that is trained on multiple files, because in the latter case, the used model will be trained beforehand, which results in not having to train the model during the compression process of each file and the model doesn't need to get stored together with the compressed data.

1. Introduction

Bioinformatics is an evergrowing field in computer science that comes with a lot of challenges and opportunities. One of these challenges is storing genomic data. As a single human genome takes up a lot of storage space and more and more genomes are sequenced, storage will grow from gigabytes to petabytes to exabytes. By 2025, an estimated 40 exabytes of storage capacity will be required for human genomic data [8]. To counter this issue, people have experimented with all kinds of lossless compression algorithms to decrease the size of genomic data without losing information along the way.

In this paper we will focus on applying recurrent neural networks (RNNs) in the field of data compression. Multiple types and sizes of RNNs will be used in a lossless data compression algorithm and we will compare our results to classical lossless compression algorithms. We will also analyse if it is beneficial to train a model for each specific genome sequence or if a general trained model does the job well enough. This analysis will be based on the difference between the achieved compression factors versus the extra time it takes to train this RNN.

2. Related work

Genome sequences are generally stored in FASTA format. In this format, genome sequence characters are stored in ASCII and are represented by four different symbols (called nucleotides or bases). Sometimes, other symbols can occur if uncertainties occur during the reading or sequencing process of the genome. In our research, we will switch these other symbols out for the known general symbol N, meaning "unknown". The problem we face is how to effectively compress strings of a certain length composed of these five elements. DeepZip is an machine learning compression method that can be used on sequential data like regular text files, but it has also been used on genomic data before [9, 20]. We will be taking a closer look at this method.

2.1. DeepZip

DeepZip at its core consists of two major components. The **probability predictor block** estimates the conditional probability distribution of a symbol S_i based on the previously K observed symbols, where K is a hyperparameter. A recurrent neural network (RNN) is used for this purpose. This probability estimate is then fed into the **arithmetic coder block**. Arithmetic coding is a form of entropy encoding that encodes an entire message into a single number, an arbitrary-precision fraction q where $0.0 \leq q \leq 1.0$. It does this by splitting up an interval into smaller intervals, based on the probabilities of certain symbols appearing at that position in the sequence. The encoding-decoding operations are shown in Figure 1 and Figure 2.

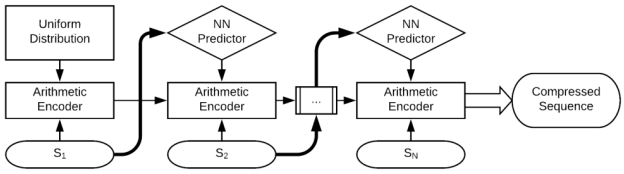

Figure 1. Encoder Framework [9]

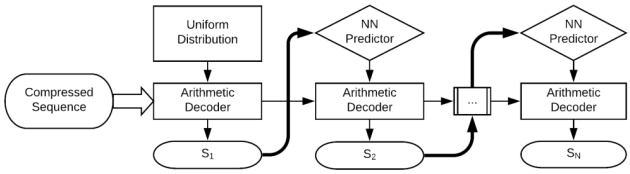

Figure 2. Decoder Framework [9]

1. The RNN model in the probability predictor block is trained and the weights are stored, to allow its usage during decompression.
2. For the initial K symbols, a uniform prior gets passed to the arithmetic encoding block. After these initial symbols, the probability predictor block uses the trained model weights to output a probability distribution over each symbol.
3. The operations of the decoder are exactly symmetrical to the encoder, as shown in Figure 2.

2.2. Static probabilities

Instead of using a recurrent neural network to predict the needed probabilities, we can also use other predictor types that are less complex. A very simple method that comes to mind straight away is to use static probabilities in every arithmetic coding step [25]. This can lead to an increase in speed, but a decrease in compression factor. Two possible ways to create decently representable probabilities are the following:

- Equal probabilities: each symbol gets a probability of $1/N$ with N being the alphabet size.
- Fair probabilities: each symbol gets a probability depending on the amount of times that symbol appears in the complete sequence.

2.3. ZIP

A last method we will be comparing with is the Windows 10 ZIP functionality [16]. The ZIP file format permits a number of compression algorithms like Store (no compression), Shrink, Reduce, Implode,... but the most commonly used compression method is Deflate, which is described in IETF RFC 1951[7, 21, 22].

3. Data

The data we will use for this project is a reference for the human genome sequence (GRCh38). We will use the FASTA file found here [2]. This dataset contains 639 sequences of different sizes.

3.1. Preprocessing

To make our data compatible with our model it has to be integer encoded. Each unique character in our data file will be represented by a unique integer. For this project we will only consider the letters A, C, G, T and N, where N stands for unknown. We will also not make a difference between lowercase and uppercase. If this is considered important we can easily increase our size of the alphabet.

4. Methods

We implemented our own version of the DeepZip algorithm. We compared the effectiveness of different model architectures that can be used in the probability predictor block.

4.1. Arithmetic coding

Figure 3 is a very good illustration for the essence of the thoughts behind arithmetic coding.

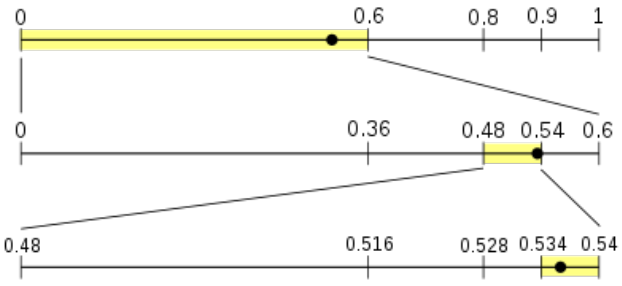

Figure 3. Inner workings of arithmetic coding [11]

It takes a stream of input symbols and replaces it with a single floating point output number. The longer (and more complex) the message, the more bits are needed in the output number. Since we are trying to compress big files, it is impractical to use a floating-point number to store our data, because we would quickly run out of space in the mantissa [11]. To resolve this issue, we can store our data in an integer and use bit shifts when we need more precision. The bits that get shifted out get written to a buffer. Later on when this buffer fills up, it gets written to a file. In our research, we use a modified version of an arithmetic coding implementation by Nayuki [17]. We modified this implementation to improve speed, readability and ease of use, but the general algorithm stays the same.

4.2. Model from predictions

As predictor we will use different recurrent neural networks. They will use their internal state as memory to process the previous K observed symbols and make a prediction for the

next symbol. We will compare the models using different input lengths, internal sizes and architectures. The models will be trained using exactly one datafile, the one it has to compress, using 10 epochs. Later we will compare the results with a traditional model that is trained on different multiple files.

All models will consist of three parts: start, middle and end as shown in Figure 4. The start will be an embedding layer [13]. This will adjust the model to suit the training data. Since our training data for each model is also the data to compress, this layer should help us increase the compression factor [3]. The middle layers will be explained in detail in subsubsection 4.2.1, subsubsection 4.2.2, subsubsection 4.2.3 and subsubsection 4.2.4. Finally we have our end part, this consists of a dense layer using relu as activation function followed by another dense layer using softmax. The softmax is used to compute our probability distribution over the possible output classes.

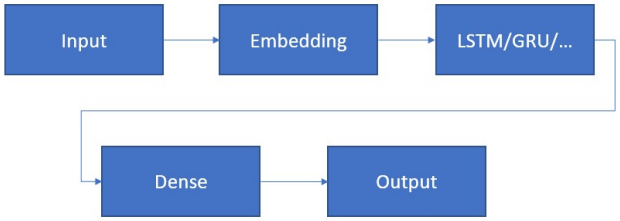

Figure 4. The architecture of our model

4.2.1. Simple RNN

The first architecture we will use is the simplest, a fully-connected RNN where the output from the previous timestep is to be fed to the next timestep [14]. This simple architecture will be compared to more complex variants.

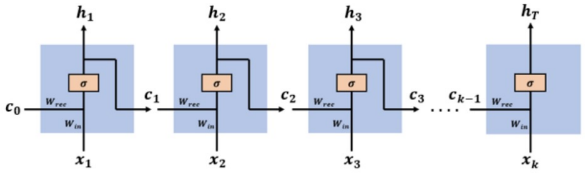

Figure 5. Simple fully connected RNN layer [1]

4.2.2. LSTM

The following architecture we will be using is Long short-term memory (LSTM) [12, 18]. LSTM has three gates: input, output and forget. It was created to solve the vanishing gradient problem that occured in classic RNNs. The addition of a forget gate will allow LSTM to decide, at every time step, that certain information should (not) be forgotten given the new input at that certain time step [1].

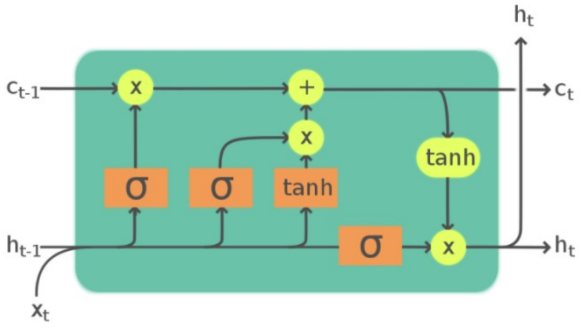

Figure 6. One LSTM cell used in our layer [24]

4.2.3. GRU

GRU or Gated recurrent unit has only two gates compared to LSTM: reset and update. [5, 15]. This architecture is computationally more efficient compared to LSTM while remaining a similar performance. The reset gate has a similar role compared to the LSTM forget gate. Essentially, this gate is used to decide how much of the past information to forget.

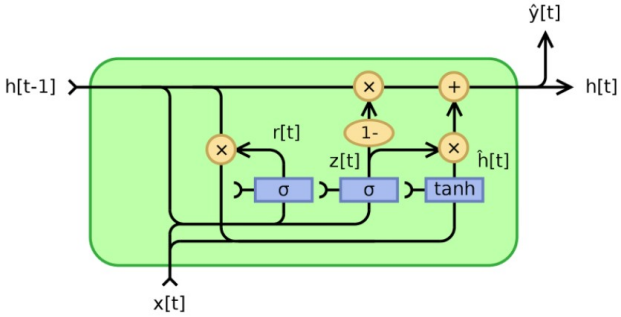

Figure 7. One GRU cell used in our layer [23]

4.2.4. BI

We will reuse the architectures from subsubsection 4.2.1, subsubsection 4.2.2 and subsubsection 4.2.3.

First we double the layer to create a stacked neural network [4]. This provides a sequence output rather than a single value output to the next layer. Specifically, one output per input time step, rather than one output time step for all input

time steps. In a study from 2013 [10] they found that the depth of the network was more important than the size of a layer.

Finally we make both layers bidirectional [19]. This means we will be processing the data in both directions with two separate hidden layers, which are then fed forwards to the same output layer.

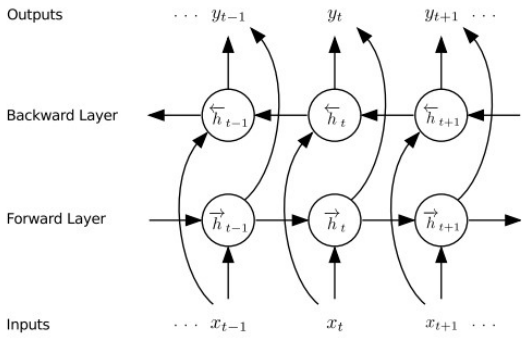

Figure 8. A bidirectional RNN layer [10]

As a result this will give us numerous additional parameters. We hope to increase the compression factor using these layers at the cost of the storage size and training time of the model.

5. Results

Since results will vary because of hardware we will list the one used in this project. We used an intel(r) core(tm) i7-6700k CPU @ 4.00gh and a NVIDIA GeForce GTX 960 GPU.

In this section we will list the results for training a model for only one genome sequence of 1023 kB. They will be compared by compression factor, training time if applicable and (de)compression time. We will also compare them with the Windows 10 ZIP functionality [16], arithmetic encoding with equal probabilities and arithmetic encoding with fair probabilities as described in section subsection 2.2.

Note that for all algorithms that use arithmetic coding, we need to store the length of the data. If we are using fair probabilities, we also need to store them. The storage that this extra information takes is negligible.

When mentioning single or double layer models, we refer to the middle part of the network as described in subsection 4.2 and not the complete neural network.

5.1. Compression factor

In the following table we will show the compressed file sizes using certain methods. The size of the saved model will also be included if applicable. Note that the saved model itself is not compressed yet, this could be done instantly using Windows ZIP.

Compression method	Size	Model
Equal probalities	297 kB	N/A
Fair probalities	256 kB	N/A
Windows ZIP	299 kB	N/A
Simple RNN single (size 32)	245 kB	467 kB
Simple RNN single (size 64)	247 kB	636 kB
Simple RNN single (size 128)	247 kB	1,09 MB
biSimple RNN double (size 32)	247 kB	2,11 MB
biSimple RNN double (size 64)	247 kB	2,87 MB
LSTM single (size 32)	239 kB	1,20 MB
LSTM single (size 64)	234 kB	1,71 MB
LSTM single (size 128)	232 kB	3,32 MB
biLSTM double (size 32)	234 kB	6,11 MB
biLSTM double (size 64)	231 kB	8,85 MB
biLSTM double (size 128)	229 kB	18,8 MB
GRU single (size 32)	238 kB	1,01 MB
GRU single (size 64)	235 kB	1,42 MB
GRU single (size 128)	240 kB	2,65 MB
biGRU double (size 32)	235 kB	5,13 MB
biGRU double (size 64)	245 kB	7,21 MB
biGRU double (size 128)	244 kB	14,7 MB

We can see that the algorithms that use a neural network give a better compression factor than the ones without a neural network. Between different models, we can see that LSTM and GRU are performing the best. Another important detail is that the bidirectional models with a double layer are performing similar to their single variant, but the storage size as well as the training time has heavily increased. More on training time in subsection 5.2.

Since we need to save the model to decompress the file, the compression isn't actually better for this file. Note that the model storage size is independent of the size of the input file because it will use a fixed amount of parameters. This means that given a large enough input file, compression will be better. Unfortunately this will come at a cost of training and de(compression) time as described in the following sections subsection 5.2 and subsection 5.3.

5.2. Training time

Since the training time is heavily dependent on the used GPU, we will give rough estimations of the training time of the models with 10 epochs. As the size of the input and model increases, so will the training time.

Model (size 32-128)	Training time
Simple RNN single	1.5-4 hours
LSTM single	1-2 hours
GRU single	0.85-1.5 hours
biSimple RNN double	10-50 hours
biLSTM double	2.25-5.5 hours
biGRU double	2-4.75 hours

We can clearly see that each bidirectional model with an extra layer heavily impacts the training time, without achieving a significant performace upgrade.

Comparing the single layer models we see that a GRU is the fastest, followed closely by LSTM. Lastly we have the Simple RNN, being the slowest of them all.

5.3. Compression time

The compression time of Windows ZIP is basically instant. For arithmetic encoding with equal or fair probabilities it takes around 15 seconds. Regarding the models, the compression time is dependent on the time it takes to predict. As a direct result we have that the time is dependent on the architecture of the model. This also means that the data used to train the model has no influence on the compression time.

During the compression we have access to the full data. In other terms, we can predict in parallel batches. This will result in a reasonable time for encoding.

Since we already realized that the bidirectional models with an extra layer have no significant improvement we will exclude them in the upcoming table and give a summary here. They will have a longer compression time than its single variant. This is because the model is more complex.

Model	Compression time
LSTM single (size 32)	1.25 minutes
GRU single (size 32)	1 minute
Simple RNN single (size 32)	3 minutes
LSTM single (size 64)	2.5 minutes
GRU single (size 64)	2.25 minutes
Simple RNN single (size 64)	4.5 minutes
LSTM single (size 128)	10 minutes
GRU single (size 128)	7.5 minutes
Simple RNN single (size 128)	8.5 minutes

5.4. Decompression time

Again, the decompression time of Windows ZIP is instant. Regarding arithmetic encoding with equal or fair probabilities, it takes a bit longer, around 20 seconds.

Unfortunately decompression with models takes a while. In the compression section subsection 5.3 we could predict our probabilities in parallel, now we only have the data that we already decoded. In other terms, we have to predict one by one which will result in a huge bottleneck.

Note that we did not decompress the file with every model to save time. The models we did not time were estimated by tqdm, a python progress bar [6].

Once again we will only display the decompression time of the single layer models.

Model	Decompression time
LSTM single (size 32)	12 hours
GRU single (size 32)	12 hours
Simple RNN single (size 32)	12 hours
LSTM single (size 64)	13 hours
GRU single (size 64)	13 hours
Simple RNN single (size 64)	14 hours
LSTM single (size 128)	18 hours
GRU single (size 128)	20 hours
Simple RNN single (size 128)	20 hours

It is now clear that we have an unacceptable bottleneck. We will discuss methods to avoid this problem in section 6

6. Discussion

In this section we will discuss some problems with the implementation and attempt to propose a solution. We will also question the effectiveness of the compression compared to a traditional model, that was trained on a training set of multiple files.

6.1. Long decompression time

As described in subsection 5.4, the decompression takes too much time because of the one by one predictions. We have to reduce the amount of predictions and/or do them in parallel.

6.1.1. Reducing the amount of predictions

In our current solution we are taking an input of size N (32 or 64) and predicting the next character X in the sequence, as shown in Figure 9. This means that the amount of predictions we have to make is equal to the length of the input file (excluding the first N characters that will be encoded using equal probabilties).

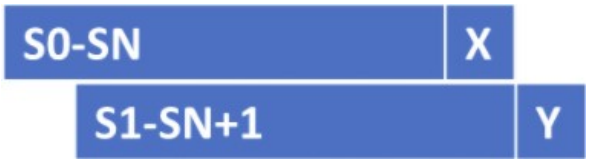

Figure 9. Model that predicts one character in each step

Instead of predicting for only one character, we can predict for the next M characters as shown in Figure 10. This will decrease in the amount of predictions by a factor of M.

Figure 10. Model that predicts next M=4 characters in each step

Note that our models input will stay the same, but it will produce a larger output. Before, our output had the size of our alphabet, which is five in our case as described in subsection 3.1. Now we will need to predict every possible combination of length M with characters in the alphabet. The output size will now be the size of our alphabet to the power of M, which can result in a memory problem if M is too big.

A final remark for this method about how to encode a sequence that is not divisible by M. There is a simple solution. Instead of encoding the first K characters with equal probabilties we can encode the first $K + R$ characters, with R being the remainder of the division of the length of the sequence by M.

6.1.2. Parallel predictions

Another way to speed up the compression and decompression procedure is to do predictions in parallel. This is sadly not possible in our current setup, because during the decompression phase, we rely on the result of the previous prediction to do the next prediction. However, if we split our data before feeding it to the encoder, we can predict in parallel batches during both the encoding and decoding phase. This method is presented in Figure 11. The operations of the decoder are exactly symmetrical to the encoder.

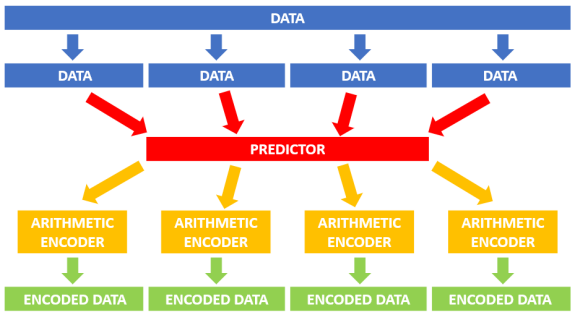

Figure 11. Parallel encoding by splitting data

6.2. Long training time

Another problem we encountered is that training a model takes too long. Reducing the amount of epochs will heavily speed up the training time, but at at the expense of the compression factor. The following table show us how much impact lowering the epochs has on training time and compression factor. We will display the model used, amount of epochs, training time and file size after compression of a 1023 kB file.

Model (size) (epochs)	time	size
LSTM single (32) (10)	60 minutes	239 kB
LSTM single (32) (2)	12 minutes	244 kB
LSTM single (32) (1)	6 minutes	245 kB
LSTM single (64) (10)	90 minutes	234 kB
LSTM single (64) (2)	18 minutes	241 kB
LSTM single (64) (1)	9 minutes	243 kB
GRU single (32) (10)	50 minutes	238 kB
GRU single (32) (2)	10 minutes	243 kB
GRU single (32) (1)	5 minutes	244 kB
GRU single (64) (10)	70 minutes	235 kB
GRU single (64) (2)	14 minutes	241 kB
GRU single (64) (1)	7 minutes	243 kB

We can see that the compression factor is similar but the time to train the model has significantly decreased. This is a big improvement for our models.

6.3. Using one model for all files

We obviously want to know if our models, trained on the single file it has to compress, can outperform a traditional model that is trained on a training set of multiple files. To compare these, we will used the same architectures from subsection 4.2. Since we have only changed the training data, the problem of slow decompression will remain. The training time will still be long, but in the case of the traditional model it is not a problem. The reason being that we will only need to train the model once.

Model (size)	Compression
LSTM single (32)	248 kB
LSTM single (64)	246 kB
GRU single (32)	246 kB
GRU single (64)	247 kB

If we compare these results with the ones with lower training time in subsection 6.2 we notice that the compression rate is a bit better in our model, but we did not optimize the traditional model, using for example k-fold cross-validation. Neither did we create a more complex, stacked model which could also increase the compression factor.

7. Conclusion

Our conclusion is that creating one model to compress exactly one file is not beneficial compared to a traditional model. It will mostly increase the total time to compress and decompress a file without a proper gain in compression factor.

Using a RNN in general results in good compression but is noticably slower compared to other methods.

8. Future Work

Since both our and the traditional model suffers from slow decompression, we have proposed multiple improvements in subsection 6.1. Further research can be done in the parameters used by these solutions. They can even be combined to create more parallelism and less predictions.

References

[1] Nir Arbel. *How LSTM networks solve the problem of vanishing gradients*. 2018. URL: https://medium.datadriveninvestor.com/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577.

[2] National Center for Biotechnology Information. *Human Genome Resources at NCBI*. URL: https://www.ncbi.nlm.nih.gov/projects/genome/guide/human/index.shtml.

[3] Jason Brownlee. *How to Use Word Embedding Layers for Deep Learning with Keras*. 2017. URL: https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/.

[4] Jason Brownlee. *Stacked Long Short-Term Memory Networks*. 2017. URL: https://machinelearningmastery.com/stacked-long-short-term-memory-networks/.

[5] Yunyoung Chung et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. URL: https://arxiv.org/pdf/1412.3555v1.pdf.

[6] Casper da Costa-Luis and Stephen Karl Larroque. *tqdm*. 2021. URL: https://github.com/tqdm/tqdm.

[7] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. 1996. URL: https://datatracker.ietf.org/doc/html/rfc1951.

[8] The Medical Futurist. *The Genomic Data Challenges Of The Future*. 2018. URL: https://medicalfuturist.com/the-genomic-data-challenges-of-the-future/.

[9] Mohit Goyal et al. *DeepZip: Lossless Data Compression using Recurrent Neural Networks*. 2018. URL: https://arxiv.org/abs/1811.08162.

[10] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. *Speech recognition with deep recurrent neural networks*. 2013. URL: https://arxiv.org/pdf/1303.5778.pdf.

[11] Mathias Helminger, Jan Schlüter, and Liang Wang. *ITM Report*. URL: https://www.cs.helsinki.fi/group/cosco/Teaching/ITProject/2009/reports/Juustohampurilainen/.

[12] Sepp Hochreiter and Jürgen Schmidhuber. *Long Short-Term Memory*. 1997. URL: https://www.bioinf.jku.at/publications/older/2604.pdf.

[13] Keras. *Embedding layer*. URL: https://keras.io/api/layers/core_layers/embedding/.

[14] Keras. *SimpleRNN layer*. URL: https://keras.io/api/layers/recurrent_layers/simple_rnn/.

[15] Simeon Kostadinov. *Understanding GRU Networks*. 2017. URL: https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be.

[16] Microsoft. *Zip and unzip files*. URL: https://support.microsoft.com/en-us/windows/zip-and-unzip-files-8d28fa72-f2f9-712f-67df-f80cf89fd4e5.

[17] Nayuki. *Reference arithmetic coding*. 2018. URL: https://www.nayuki.io/page/reference-arithmetic-coding.

[18] Christopher Olah. *Understanding LSTM Networks*. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs.

[19] Mike Schuster and Kuldip K. Paliwal. *Bidirectional recurrent neural networks*. 1997. URL: https://www.researchgate.net/publication/3316656_Bidirectional_recurrent_neural_networks.

[20] Rongjie Wang, Tianyi Zang, and Yadong Wang. *Human mitochondrial genome compression using machine learning techniques*. 2019. URL: https://humgenomics.biomedcentral.com/articles/10.1186/s40246-019-0225-3.

[21] Hans Wennborg. *Shrink, Reduce and Implode: The Legacy Zip Compression Methods*. 2021. URL: https://www.hanshq.net/zip2.html.

[22] Hans Wennborg. *Zip Files: History, Explanation and Implementation*. 2020. URL: https://www.hanshq.net/zip.html.

[23] Wikipedia. *Gated recurrent unit*. 2021. URL: https://en.wikipedia.org/wiki/Gated_recurrent_unit.

[24] Wikipedia. *Long short-term memory*. 2021. URL: https://en.wikipedia.org/wiki/Long_short-term_memory.

[25] Ian H. Witten, Radford M. Neal, and John G. Cleary. *Arithmetic coding for data compression*. 1987. URL: https://web.stanford.edu/class/ee398a/handouts/papers/WittenACM87ArithmCoding.pdf.