02456 DEEP LEARNING, DTU COMPUTE, AUTUMN 2020

GENERATIVE MODELLING OF INSULIN PROTEINS

Jonas Vestergaard Jensen, s162615 & Emma Ahrensbach Rørbeck, s173733

Technical University of Denmark

ABSTRACT

Design of proteins with desired properties and functions is a complex problem, but advances in this field could have a large impact in biology and medicine. In this paper, we address the problem of generating insulin protein using different deep learning architectures. The models used are recurrent neural networks with long short-term memory (LSTM) cells and gated recurrent units (GRU) as well as a transformer model and a model based on WaveNet by Google DeepMind [1]. The GRU model showed the best average test perplexity (3.5) and both the GRU and LSTM model generated proteins more similar to the insulin test proteins than the Transformer and WaveNet models.

Index Terms— Protein generation, insulin, recurrent neural network, LSTM, GRU, transformer, WaveNet

1. INTRODUCTION

Being able to freely design proteins with specific properties and functions is a complex problem and considerable advances in this field could have a large impact in biology and medicine [2].

In this paper, four different deep learning model architectures for modelling sequential data have been applied to the problem of generating insulin protein sequences. The first two models are variations of the basic recurrent neural network (RNN), namely RNNs with long short-term memory (LSTM) cells and gated recurrent units (GRU) cells respectively. The third model used is a transformer model making use of an attention mechanism to gain an infinite reference window [3]. The last model is an auto-regressive generative model, based on the WaveNet model by Google DeepMind [1].

The joint probability $p(\boldsymbol{x})$ of a protein sequence, $\boldsymbol{x} = (\boldsymbol{c}, \boldsymbol{a}) = (x_1, x_2, \ldots, x_{n_c+n_a})$ with conditioning tags (properties) $\boldsymbol{c} = (c_1, c_2, \ldots, c_{n_c})$ and amino acids $\boldsymbol{a} = (a_1, a_2, \ldots, a_{n_a})$ can be factorised using the chain rule of conditional probabilities

$$p(\boldsymbol{x}) = \prod_{i=1}^{n_c+n_a} p(x_i | x_{i-1}, x_{i-2}, \ldots, x_1)\,, \tag{1}$$

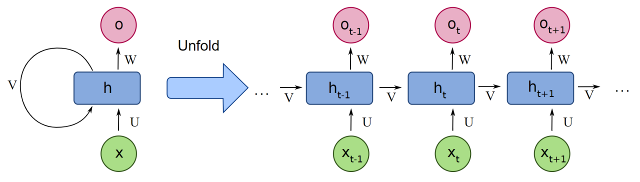
and thus the problem of generating insulin proteins can be described as a next-token prediction problem [2]. During training the negative log likelihood is minmized, which for a data set of N proteins is

$$\mathcal{L} = -\sum_{n=1}^{N} \log p(\boldsymbol{x}^{(n)})\,. \tag{2}$$

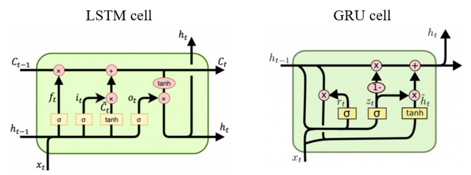
2. MODEL ARCHITECTURES

2.1. Recurrent Neural Networks

The RNN is successful in processing sequential data due to its cyclic computations, which means the RNN can recall previous calculations as a variant of memory. In Figure 1a, the cyclic nature of an RNN is seen.



(a) A representation of an RNN. Reprinted from [4].



(b) GRU and LSTM cells. Reprinted from [5].

Fig. 1. Architectures of the RNN and the GRU and LSTM cells.

RNNs operate on a sequence containing vectors $\boldsymbol{x}^{(t)}$, with t indexing the time steps from 1 to τ. The time step may refer to the actual passage of time, or may simply denote the

GitHub repository: `https://github.com/jonasvj/protein-generation`

Notebook: `https://github.com/jonasvj/protein-generation/blob/master/notebooks/results.ipynb`

Poster: `https://github.com/jonasvj/protein-generation/blob/master/reports/poster.pdf` position in a sequence. The hidden state and output of the RNN at time t are respectively

$$\boldsymbol{h}_t = \phi\left(U\boldsymbol{x}_t + V\boldsymbol{h}_{t-1}\right) \tag{3}$$
$$\boldsymbol{o}_t = \text{softmax}\left(W\boldsymbol{h}_t\right) , \tag{4}$$

where U, V and W are learnable weight matrices and ϕ denotes an activation function.

An RNN is, however, slow to train, and the training time cannot be reduced by parallelization as the forward propagation is sequential, and each time step can only be computed after the previous step [6]. The RNN also suffers from the vanishing or exploding gradients problem. And while the RNN does have the ability to retain information, it is unable to keep it for long [4]. This last complication is remediated by the usage of GRU and LSTM cells, which both have a longer short-term memory than the simple RNN, and are seen in Figure 1b.

2.1.1. LSTM

The LSTM cell introduces different gates; an input gate, an forget gate, an cell gate and a output gate. Gated RNNs create paths through time that have derivatives that neither vanish or explode, thereby fixing the problem of the regular RNN [6].

The memory of the LSTM cell is its cell state, which is updated by the forget gate and the input gate. The forget gate controls what information the cell retains, and the input gate adds the new information from the current input. The cell state is added as input to the next cell, which therefore retains the information learned by the current cell. The hidden state of the cell is updated by the output gate, and is afterwards used to calculate the output of the current cell and used as input for the next cell [6]. The equations of the input, forget, cell and output gates are respectively

$$\boldsymbol{i}_t = \sigma\left(W_{ii}\boldsymbol{x}_t + W_{hi}\boldsymbol{h}_{t-1}\right) \tag{5}$$
$$\boldsymbol{f}_t = \sigma\left(W_{if}\boldsymbol{x}_t + W_{hf}\boldsymbol{h}_{t-1}\right) \tag{6}$$
$$\tilde{\boldsymbol{C}}_t = \tanh\left(W_{ic}\boldsymbol{x}_t + W_{hc}\boldsymbol{h}_{t-1}\right) \tag{7}$$
$$\boldsymbol{o}_t = \sigma\left(W_{io}\boldsymbol{x}_t + W_{ho}\boldsymbol{h}_{t-1}\right) , \tag{8}$$

where σ is the sigmoid function, $\boldsymbol{h}_t$ is the hidden state at time t and W_{kk} are learnable weight matrices. The cell state and hidden state are updated with

$$\boldsymbol{C}_t = \boldsymbol{f}_t \odot \boldsymbol{C}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{C}}_t \tag{9}$$
$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh\left(\boldsymbol{C}_t\right) , \tag{10}$$

where $\odot$ is the Hadamard product and $\boldsymbol{C}_t$ is the cell state at time t.

Using an RNN with LSTM cells adds quite a few computations, and is therefore slower to train relative to the regular RNN.

2.1.2. GRU

The GRU cell is quite similar to the LSTM cell, with the primary difference being that a single gating unit in the GRU simultaneously controls the forgetting factor, and the decision to update the start unit [6].

The gates of the GRU are instead defined as the update gate, the reset gate and the new gate. The update gate can linearly gate any dimension, and the reset gate controls which parts of the state that is used to compute the next target state [6]. The equations for the reset, update and new gates are respectively

$$\boldsymbol{r}_t = \sigma\left(W_{ir}\boldsymbol{x}_t + W_{hr}\boldsymbol{h}_{t-1}\right) \tag{11}$$
$$\boldsymbol{z}_t = \sigma\left(W_{iz}\boldsymbol{x}_t + W_{hz}\boldsymbol{h}_{t-1}\right) \tag{12}$$
$$\tilde{\boldsymbol{h}}_t = \tanh\left(W_{ih}\boldsymbol{x}_t + \boldsymbol{r}_t \odot W_{hh}\boldsymbol{h}_{t-1}\right) , \tag{13}$$

where W_{kk} are learnable weight matrices as previously. The hidden state is updated with

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t . \tag{14}$$

The GRU therefore has fewer computations than the LSTM, and should train faster, albeit still slower than the regular RNN, which is a trade-off for larger precision.

2.2. Transformer

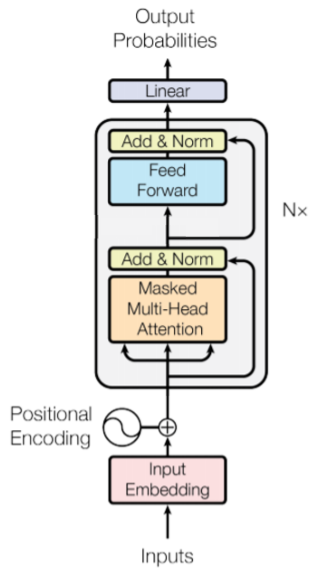


Fig. 2. The transformer architecture. The marked block can be repeated N amount of times. Adapted from [3].

The Transformer is a model architecture, which can be used for sequential data, but without the use of recurrence, instead relying entirely on an attention mechanism to draw dependencies between the input and the output. This allows for parallelization, and therefore much faster computing than the RNN [3].

The Transformer can be used as a translational tool, in which case it would make use of both an encoder and decoder structure. For this project, though, the encoder will not be necessary, and simply the decoder will be used. The decoder structure used is seen in Figure 2.

In the Transformer, the input is embedded, which includes a positional encoding. As the attention used by the Transformer is translation invariant, the positional encoding is needed to retain the sequence information. The positional encoding is done using sine and cosine functions

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}}) \tag{15}$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})\,, \tag{16}$$

where d_{model} is the embedding dimensionality, pos is the sequence position and i is the embedding dimension [3].

After the encoding layer follows the attention layer. The attention function can briefly be described as mapping a query, and a set of key-value pairs to an output [3]. The queries, keys, and values are denoted Q, K, and V, and the attention is calculated as follows

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V\,, \tag{17}$$

where d_k is the dimension of the keys.

For the Transformer, multi-headed attention is used, which calculates the attention multiple times simultaneously. This ensures, that different information can be attended to for each attention head, and therefore different patterns can be noted by each head. These different attention heads are then concatenated before addition and normalization in the next layer. To ensure that the next token prediction is not based on tokens further down the sequence, the attention layer is masked, which is done by setting all illegal connections to $-\infty$ [3].

The Transformer makes use of self-attention, specifically. In self-attention, the queries, keys, and values are all derived from the output of the previous layer. The self-attention lets the Transformer learn which words in an input sentence, for example, are associated with, or references, each other [3].

A fully connected feed forward layer follows the attention layer. The output probabilities are calculated by letting the output of the fully connected layer pass through a linear layer and a softmax function. Based on this, the next-token prediction is made.

2.3. WaveNet

The WaveNet architecture was developed to generate raw audio signals such as speech and music [1]. The key elements of the architecture are stacked dilated causal convolutions, residual connections and skip connections [1].

The use of causal convolutions ensures that the model only has access to information at sequence positions preceding the position being predicted, i.e. the prediction is $p(x_{t+1}|x_t, x_{t-1}, x_{t-2}, \dots)$ [1].

The WaveNet architecture does not have any recurrent elements and relies only on causal convolutions and are therefore typically faster to train than RNNs [1]. However, the receptive field of a network with a stack of causal convolutions is only $\#layers + filter\,length - 1$, and therefore many layers are needed to gain an adequately large receptive field [1]. A stack of dilated convolutions results in a more coarse network but can increase the receptive field by orders of magnitude without increasing the computational cost much [1].

An example of a stack of dilated causal convolutions is shown in Figure 3. In Figure 3 the dilation is doubled for every layer until it reaches a limit (in this case four) whereafter the pattern is repeated. The exponential increase in dilation factor increases the receptive exponentially and the repetition of the pattern both increases the model capacity and the receptive field [1].

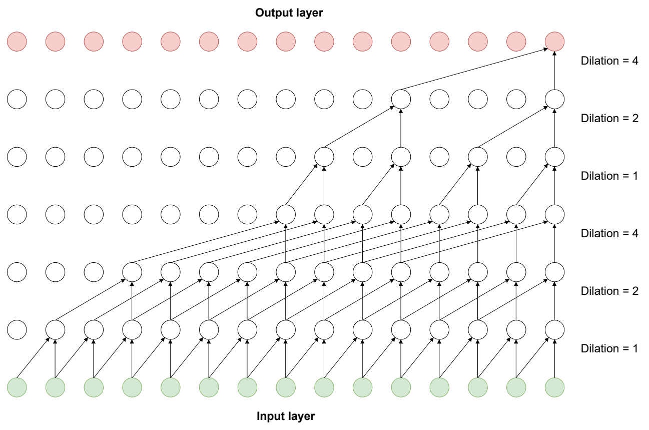


Fig. 3. An example of stacked dilated causal convolutions. Only connections to the right-most output are shown. Adapted from [1].

As mentioned, WaveNet utilizes skip connections and residual connections. A residual connection aims to learn the deviation from an identity mapping of the input, i.e. it aims to learn the function defined by $\mathcal{F}(\boldsymbol{x}) := \mathcal{H}(\boldsymbol{x}) - \boldsymbol{x}$, where $\mathcal{H}(\boldsymbol{x})$ is the actual desired underlying mapping that should be learned [7]. The desired mapping is then obtained by $\mathcal{F}(\boldsymbol{x}) + \boldsymbol{x}$ [7]. This reformulation of the learning problem has shown to be advantageous empirically and allows for faster convergence and training of deeper models [1, 7].

The following gated activation function is used in the WaveNet model

$$\boldsymbol{z} = \tanh\left(W_{f,k} * \boldsymbol{x} + V_{f,k}^T \boldsymbol{h}\right) \odot \sigma\left(W_{g,k} * \boldsymbol{x} + V_{g,k}^T \boldsymbol{h}\right)\,, \tag{18}$$

where $*$ denotes the convolution operator, $W_{f,k}$ and $W_{g,k}$ are learnable convolution filters, $V_{f,k}$ and $V_{g,k}$ are learnable linear projections and $\boldsymbol{h}$ is a global input available at all positions in the sequence, such as protein properties. The full WaveNet architecture is shown in Figure 4.

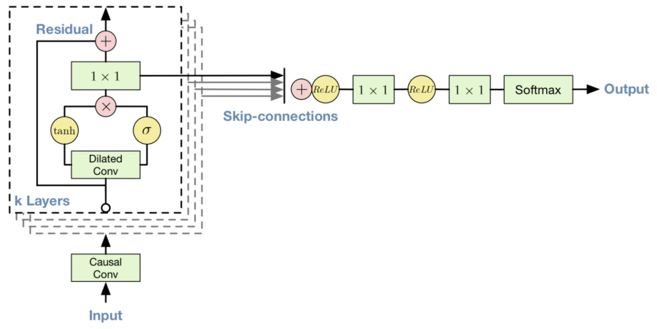


Fig. 4. The full WaveNet architecture. Reprinted from [1].

3. METHODS

3.1. Data

The data used for modelling insulin proteins were amino acid sequences of proteins from the Insulin/IGF/Relaxin protein family (Pfam ID: PF00049 [8]) gathered from Swiss-Prot and TrEMBL [9]. UniProt keywords in the categories "Biological process", "Cellular component" and "Molecular function" were also gathered for each protein as well as the organism that each protein originated from. The UniProt keywords and organism labels were included in the modelling as conditioning tags by prepending them to the amino acid sequence (except for the WaveNet model where they were included as the $\mathbf{h}$ vector).

The mean length of the included amino acids sequences were 154 but some were as short as 24 amino acids and some were as long as 1793 amino acids. For that reason, the most deviant proteins (by length) were filtered out by only including proteins with an amino acid sequence length between the 2.5th percentile and 97.5th percentile. This filtering step resulted in proteins with amino acid sequence lengths between 51 and 273. If several proteins had identical amino acid sequences only one of these protein were kept in the data set.

Besides the insulin proteins, a separate set of proteins not in the Insulin/IGF/Relaxin family (with same range of sequence lengths) were gathered from Swiss-Prot to possibly include in the training set. We hypothesised that these non-insulin proteins could help the models learn general protein features common to most types of proteins including insulin proteins. In total, the processed data set contained 3272 insulin proteins and 2617 non-insulin proteins.

It has been shown previously that the performance of generative protein models can be improved by including the reversed amino acid sequences during training [2]. For that reason, we also experimented with inclusion of the reversed sequences during training.

Many proteins had several keywords annotated to the same keyword category, for example one protein might be involved in several biological processes. To deal with this ambiguity, the data was preprocessed in two different ways; 1) one keyword was chosen randomly (among the annotated keywords) for each category per protein, 2) All combinations of the annotated keywords were included for each protein such that the same amino acid sequence appeared several times in the data set (but with different keyword combinations).

3.2. Training

All models were implemented in Pytorch [10] and trained on the high-performance computing cluster at the Technical University of Denmark using NVIDIA Tesla V100 GPUs. The data set described in Section 3.1 was split into a training set, validation set and test set with respective proportions of the original data set of 80%, 10% and 10%. The validation set was used to manually search for the most optimal hyper-parameters for the different models.

The optimizations were done using the Adam optimizer [11] with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 1\text{e-}8$ and the cross-entropy loss function. During training a learning rate scheduler was used that reduced the learning rate by a factor of 10 if the relative validation perplexity had not improved by 0.01% during the latest 10 epochs. Furthermore, a learnable encoding layer was included in each model to learn encodings for the amino acids and keywords.

After selection of hyper-parameters (based on the manual hyper-parameter search), the models with the selected hyper-parameters were trained and validated again but by using the two different ways of preprocessing the keywords (described in Section 3.1) as well as with and without non-insulin proteins and with and without including the reversed amino acid sequences. The final models and their respective data set structure were selected based on their average perplexity on the validation data.

3.3. Protein Representations

A latent representation of a given (test) protein can be obtained by feeding the amino acid sequence of the protein (and its conditioning tags) to one of the trained models and extracting the output of the last hidden layer for each sequence position. The size of this representation is dependent on the protein length, but can be fixed by for example taking the mean or max over the sequence dimension.

Using this procedure with mean pooling, a latent representation of each test protein was generated with the Transformer model and WaveNet model. For the LSTM model and GRU model, the latent representations generated for each test protein were simply the hidden layer at the last sequence position, as this layer is dependent on the hidden layers at previous sequence positions.

To visualise the latent representations, the dimensionality of the representations were reduced to two using t-distributed stochastic neighbour embedding (t-SNE) [12].

3.4. Protein Generation

Given a context sequence, the models output a probability distribution over the different amino acids. A new amino acid is then generated by sampling from this distribution. Thus, the models generate one amino acid at a time and a sequence can be generated by repeatedly generating an amino acid and updating the context sequence. In this paper, top-k sampling with $k = 1$ is used to sample amino acids and generation is continued until an end-of-sequence token is sampled or the sequence reaches a length that is twice that of the longest training sequence.

To test how well the models generate proteins, the models were fed with varying proportions of each test protein as context and thereafter used to generate the rest of the given protein.

The generated protein segments were then aligned to the reference, i.e. the part of the test protein not used as context. The global pairwise alignments were made with Biopython [13] using the Needleman-Wunsch algorithm [14] and the biologically informed BLOSUM62 substitution matrix [15], that takes into account that some amino acid substitutions are less severe than others. The gap opening penalty was set to -0.5 and the gap continuation penalty to -0.1. The resulting alignment scores were normalized by the alignment lengths. Since the alignment scores do not have a direct interpretation a 50% mutation baseline was included for comparison. For the 50% mutation baseline, the generated protein is simply the test protein with 50% of its amino acids being randomly mutated (substituted).

4. RESULTS AND ANALYSIS

4.1. Hyper-parameter and data selection

Table 1 shows the selected hyper-parameters for each model and Table 2 shows the data set structure selected for each model. As described in Section 3.2, all selections were based on the validation data.

Table 2 reveals that inclusion of non-insulin proteins were only beneficial for the LSTM model. This result might be explained by the LSTM model having the largest number of parameters (See Section 4.2) and therefore the LSTM model can perhaps better handle the increased diversity of the data.

The table also shows that none of the models benefited from inclusion of reversed amino acid sequences, contrary to the results of Madani et al. [2]. Finally, sampling one annotated keyword per keyword category for each protein was sufficient for the LSTM and GRU models while the Transformer and WaveNet models benefited from including all combinations of keywords for each protein.

	Model			
Hyper-parameter	LSTM	GRU	Transformer	WaveNet
Epochs	300	300	250	200
Initial learning rate	1e-4	1e-3	1e-3	1e-3
Mini-batch size	64	64	64	64
Encoding size	16	16	32	16
Weight-decay	0	0	0	3.5e-3
Drop out rate	0.5	0.5	0.1	-
Number of hidden layers	4	4	6	11
Size of hidden layers	1024	1024	1024	-
Number of heads	-	-	16	-
Kernel size	-	-	-	2
Dilations	-	-	-	1, ..., 16, 1, ..., 16
Residual channels	-	-	-	256
Dilation channels	-	-	-	256
Skip channels	-	-	-	128
Final channels	-	-	-	64

Table 1. Hyper-parameters selected for the different models based on a manual hyper-parameter search using the validation data.

	Model			
	LSTM	GRU	Transformer	WaveNet
Keyword method	Sample	Sample	Combination	Combination
Include non-insulin	Yes	No	No	No
Include reverse	No	No	No	No

Table 2. Data set structure selected for each model based on the validation data. See Section 3.1 for details.

4.2. Performances

Table 3 lists the average perplexity on the test data for the selected models. The table also lists the number of parameters in each model.

	Model			
	LSTM	GRU	Transformer	WaveNet
Number of parameters	31,340,560	22,930,788	548,868	3,656,602
Avg. test perplexity	4.1	3.5	4.6	4.6

Table 3. Performance of the different models.

From Table 3 it can be seen that the selected GRU model showed the best performance with an average test perplexity of 3.5, while the LSTM had a perplexity of 4.1 and both the Transformer model and WaveNet model had a perplexity of 4.6. However, Table 3 also shows that LSTM and GRU model had significantly more parameters than the Transformer and WaveNet model and perhaps these models could be improved by increasing their capacity.

4.3. Protein Representations

The results of generating embeddings of the test proteins as decribed in Section 3.3 are showcased in Figure 5, where each test protein has been colored by its molecular function.

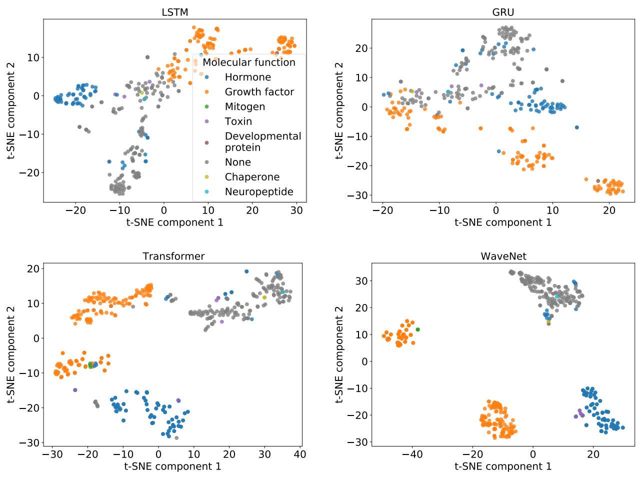


Fig. 5. Embeddings of test proteins produced by each model reduced to two dimensions by t-SNE. Test proteins are colored according to their molecular function.

From Figure 5 it can be seen that each model produces protein embeddings that give rise to clusters of proteins. These clusters are most distinct for the WaveNet model and the least distinct for the GRU model. Furthermore, these clusters generally coincide with the molecular function of the proteins, which is especially true for the WaveNet and Transformer models. Interestingly, the embeddings generated by the Transformer and WaveNet model produce two distinct clusters of proteins both containing growth factor proteins, suggesting that there might in fact be two types of growth factors in the data set. Figure 5 could also suggest that the LSTM and GRU models does not fully utilize the conditioning tags, hinting that their memories are indeed limited.

4.4. Protein Generation

The results of the protein generation described in Section 3.4 are showcased in Figure 6. All models are seen as being generally better than the 50% mutation baseline, suggesting that all models have learned an overall structure of insulin (more specifically proteins from the Insulin/IGF/Relaxin family). As expected, the similarity of the generated proteins to the test proteins generally increase when a greater context is given. There is, however, a large variation in the similarity of the generated proteins to the test proteins as indicated by the large shaded blue regions in Figure 6.

For the LSTM and GRU model the average (normalized) alignment score is roughly in the range 3.25 to 4 while the range is roughly 3 to 3.5 for the Transformer and WaveNet

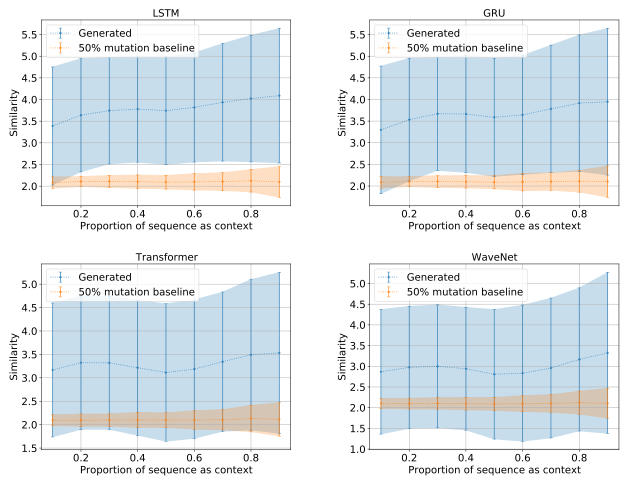


Fig. 6. Sequence similarities between the model-generated sequences and a 50% mutation baseline, based on differing amounts of context. Points are average (normalized) alignment scores for a given context proportion and shaded regions indicate one standard deviation.

model, indicating that the LSTM and GRU models are better at generating proteins similar to insulin.

Furthermore, the similarity of the mutated proteins to the test proteins lies within one standard deviation of the similarity of the generated proteins by the Transformer and WaveNet models to the test proteins. This is generally not the case for the LSTM and GRU models, which again favors these models.

5. CONCLUSION

Four different models, i.e. the LSTM, GRU, Transformer, and WaveNet models, have been successfully implemented and trained to be able to do next-token predictions of amino acids in protein sequences of the Insulin/IGF/Relaxin family and thereby also generation of these types of proteins. Exactly which of the models produced the best results is not straightforward. The GRU model had the lowest average test perplexity while the Transformer and WaveNet models appeared to generate better protein embeddings. Lastly, the LSTM and GRU models generated proteins more similar to the test proteins. This could suggest, that the GRU model is the overall best model of those presented in this paper for the problem of generating insulin proteins. A drawback of the GRU model is, however, its long training time compared to the Transformer and WaveNet models. The results presented do depend on the initializations and selected hyper-parameters, and it is therefore possible that the other models could do better with different initializations and/or hyper-parameters, for example by increasing the number of parameters in the Transformer and WaveNet models.

6. REFERENCES

[1] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu, "Wavenet: A generative model for raw audio," 2016.

[2] Ali Madani, Bryan McCann, Nikhil Naik, Nitish Shirish Keskar, Namrata Anand, Raphael R. Eguchi, Po-Ssu Huang, and Richard Socher, "Progen: Language modeling for protein generation," 2020.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[4] Pedro Torres Perez, "Deep learning: Recurrent neural networks," 2018, Available at: `https://medium.com/deeplearningbrasilia/deep-learning-recurrent-neural-networks-f9482a24d010`. Last accessed 01 January 2021.

[5] "RNN, LSTM & GRU," 2019, Available at: `http://dprogrammer.org/rnn-lstm-gru`. Last accessed 01 January 2021.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*, chapter 10, The MIT Press, 2016.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep residual learning for image recognition," 2015.

[8] Sara El-Gebali, Jaina Mistry, Alex Bateman, Sean R Eddy, Aurélien Luciani, Simon C Potter, Matloob Qureshi, Lorna J Richardson, Gustavo A Salazar, Alfredo Smart, Erik L L Sonnhammer, Layla Hirsh, Lisanna Paladin, Damiano Piovesan, Silvio C E Tosatto, and Robert D Finn, "The Pfam protein families database in 2019," *Nucleic Acids Research*, vol. 47, no. D1, pp. D427–D432, 10 2018.

[9] The UniProt Consortium, "UniProt: a worldwide hub of protein knowledge," *Nucleic Acids Research*, vol. 47, no. D1, pp. D506–D515, 11 2018.

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., pp. 8024–8035. Curran Associates, Inc., 2019.

[11] Diederik P. Kingma and Jimmy Ba, "Adam: A method for stochastic optimization," 2017.

[12] Laurens van der Maaten and Geoffrey Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008.

[13] Peter J. A. Cock, Tiago Antao, Jeffrey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, and Michiel J. L. de Hoon, "Biopython: freely available Python tools for computational molecular biology and bioinformatics," *Bioinformatics*, vol. 25, no. 11, pp. 1422–1423, 03 2009.

[14] Saul B. Needleman and Christian D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443 – 453, 1970.

[15] S Henikoff and J G Henikoff, "Amino acid substitution matrices from protein blocks," *Proceedings of the National Academy of Sciences*, vol. 89, no. 22, pp. 10915–10919, 1992.