Advanced Computer Architectures – Final Project

Nadav Yosef-Zada 208825257, Chen Kruphman 323101311

Introduction:

As life evolves, DNA sequences accumulate changes over time, leading to the diversification of species and the emergence of distinct lineages. This principle extends even to the realm of viruses, entities that straddle the boundary between living and non-living. Viruses, too, possess genetic material that mutates as they traverse hosts and environments. Understanding the dynamic changes in viral DNA is paramount, especially in the case of rapidly evolving pathogens like viruses. Classifying DNA sequences of a virus into different lineages is essential for multiple reasons. It provides insights into the virus's evolutionary trajectory, aiding in tracking its origin, spread, and potential mutations. This classification assists researchers in monitoring the effectiveness of vaccines, treatments, and containment strategies. By deciphering the intricacies of DNA lineages within viruses, scientists gain a crucial vantage point to combat emerging threats and safeguard global health.

Our Project:

We will classify SARS-COV-19's genomic sequences to different existing lineages, using 2 CNN models:

1. CNN
2. CNN with Embedding layer
3. CNN using forward-forward algorithm

We will compare the results of those networks.

About our Data:

The data we used contains a few thousands of accessions from 5 different lineages of SARS-COV-19. The lineages we took are – "Beta", "Eta", "Zeta", "Lambda", "Mu". We chose these specific lineages since they have some distinction between them (several stages of evolution of SARS-COV-19) which indicates more mutations and differences, making it easier for a network to identify unique features, and train on. Another reason was that each lineage has a few thousands of accessions, which is enough for the network to train on and validate, but not too much data that will be too hard to store and use on a general computer. The way the data is built allows us to perform better training of the network, and the amount of data gives us the option to validate and prevent overfitting for a certain lineage (since the scale of data taken for each lineage is not very different).

Data Processing:

The data we used comes in a FASTA file format, which contains a header for the accession the sequence belongs to, and a long string of nucleotides, which are the building blocks of DNA sequences. The nucleotides are A,C,G,T, but in reality, due to errors made in the sequencing process there can also be N (which signifies inconclusive result over what is the nucleotide that was supposed to be there). Since convolutional networks get Tensors as inputs, we had to perform pre-processing steps on our data:

1. Parsing the FASTA files in order to extract the genomic sequences
2. Saving the sequences, their lengths and their labels as a data frame (to make the other parts simpler to perform)
3. Finding the longest sequence and padding the rest of the sequences with ‘N’ to its length
4. For the CNN and Embedding net:
    a. Encoding the padded sequence by assigning them an integer from the range [0,4]
    b. Encoding the labels by assigning them an integer value (using LabelEncoder from sklearn.preprocessing)
5. For the Forward-Forward net:
    a. Encoding the padded sequences using binary One-Hot Encoding
    b. Creating a grayscale image using the encoding , with dimensions seq_lenX5 (5 being the number of different nucleotides), where in each row of the image only a single pixel is 1 (according to the encoding)
    c. Transforming the image to Tensor for the neural network
6. Splitting the processed data to train and test classes for the network training and testing (80% for training and 20% for testing)
7. Batching the data, each batch of size 32

Our models:

CNN – using the pytorch model, we built a neural network that consists of:

1. Convolutional Layers: Two convolutional layers are defined in the constructor. These layers are responsible for learning features from the input DNA sequences. conv1 is the first convolutional layer with 5 input channels (assuming one-hot encoding for the nucleotides), 32 output channels (the filters), and a kernel size of 3. conv2 is the second convolutional layer with 32 input channels, 64 output channels, and a kernel size of 3.

2. Calculation of Convolutional Output Size: It calculates the size of the output feature maps after applying the convolutional and max-pooling layers. This is necessary to determine the input size for the fully connected layers.

3. Fully Connected Layers: Two fully connected (linear) layers are defined. These layers are used for classification based on the features extracted by the convolutional layers. fc1 is the first fully connected layer with input features determined by the flattened convolutional output and 128 output features. fc2 is the second fully connected layer with 128 input features and 5 output features, since we have 5 possible classes for classification.

4. Forward Pass: The “forward” method defines the forward pass of the neural network. It specifies how the input data should be processed through the layers. Its propagation steps:

    a. Apply the first convolutional layer with ReLU activation and max-pooling.
    b. Apply the second convolutional layer with ReLU activation and max-pooling.
    c. Flatten the output and pass it through the fully connected layers.
    d. Apply ReLU activation to the first fully connected layer.
    e. Apply the final fully connected layer without activation.
    f. Apply a log softmax function to produce class probabilities.

CNN with Embedding layer - using the pytorch model, this neural network that consists of:

1. Parameters: The constructor initializes the model and its layers, gets 4 parameters:
   a. "vocab_size": The number of unique elements in the input data
   b. "embedding_dim": The dimension of the embedding layer
   c. "num_filters": The number of filters (output channels) in the convolutional layer
   d. "output_dim": The dimension of the output layer (number of classes for classification)

2. Embedding Layer: An embedding layer is created with "vocab_size" as the number of unique elements in the input data, and "embedding_dim" as the dimension of the embeddings. This layer is used to learn dense representations of the input sequences.
   <u>What Is an Embedding?</u> - An embedding is a dense vector representation of a categorical value. Instead of using binary vectors like in one-hot encoding, embeddings use real-valued vectors with a lower dimension. These vectors are learned during training and capture relationships and similarities between categories. The idea is to place similar categories closer together in the embedding space.
   embedding layer is a crucial component in neural networks when working with categorical data, such as text or DNA sequences. Its importance stems from:

   a. Representing Categorical Data: When dealing with categorical data, like words in natural language processing or nucleotides in DNA sequences, you cannot directly use these categories as input to a neural network. Neural networks work with numerical data, so you need a way to convert these categories into numerical representations.
   b. One-Hot Encoding vs. Embeddings:
   c. Benefits of Embeddings:
      - Reduced Dimensionality: One way to represent categorical data is through one-hot encoding. In one-hot encoding, each category is represented as a binary vector where one element is set to 1 and all others are set to 0. However, one-hot encoding can result in very high-dimensional data, and it doesn't capture any semantic relationships between categories. Embeddings offer a more efficient and meaningful way to represent categorical data. Embeddings typically have a lower dimensionality compared to one-hot encoding, which reduces the computational complexity of the model.
      - Efficient Learning: Neural networks can learn embeddings as part of the training process, optimizing them to best suit the task at hand.
   d. Using Embeddings in the Model: The embedding layer is used to convert integer sequences (representing one-hot encoded DNA sequences) into dense embeddings. These embeddings are then processed by subsequent layers in the neural network to learn patterns and make predictions.

3. Convolutional Layer: A 1D convolutional layer is defined with the following settings:

   a. "in_channels": Set to "embedding_dim", indicating the number of input channels.
   b. "out_channels": Set to "num_filters", indicating the number of output channels or filters.
   c. "kernel_size": Set to 3, indicating a 1D convolution with a kernel size of 3.

4. Fully Connected Layer: The fully connected (linear) layer is defined with "num_filters" as the input features and "output_dim" as the output features. This layer is used for the final classification.

5. Forward Method: The "forward" method defines the forward pass of the model, specifying how input data should be processed through the layers. Its layers:

   a. Embedding: The input data is passed through the embedding layer to convert it into dense embeddings.

b. Transpose: The output from the embedding layer is transposed to change the dimensions.
c. Convolution and Activation: The 1D convolution is applied with ReLU activation.
d. Max Pooling: Max-pooling is performed over the output of the convolutional layer.
e. Fully Connected: The output of the max-pooling layer is passed through the fully connected layer, which produces the final classifications.

CNN using forward-forward algorithm – Using pytorch, this network is a custom neural network in which each layer contains several stages. The network is built as below:

1. Defining global variables:
   a. “threshold”: A threshold value used in the “Layer” class.
   b. “epochs”: The number of training epochs.
   c. “lr”: The learning rate for the optimizer.
   d. “log_interval”: The interval for logging training progress.

2. Define utility functions:
   a. “get_y_neg(y)”: Generates a random negative label for each input label.
      - This function takes a tensor “y” as input, which contains labels for a dataset.
      - It creates a new tensor “y_neg” with the same values as “y” initially.
      - Then, for each label in “y”, it generates a random negative label by removing the current label from the list of allowed labels.
      - The new negative label is selected randomly from the remaining labels.
      - Finally, it returns the tensor “y_neg” with the negative labels.

      This function is used to create negative labels for a contrastive loss.

   b. “overlay_y_on_x(x, y)”: Overlay the label “y” on input “x” to create a new tensor.
      - This function takes two tensors as input: “x”, which is a batch of data samples, and “y”, which are labels.
      - It creates a new tensor “x_” by cloning the input “x”.
      - It then sets all values in “x_” to 0 to clear any existing values.
      - Next, it overlays the label information from “y” onto “x_”. For each sample in the batch, it sets the value in “x_” corresponding to the label “y” to the maximum value in the original “x”. This operation effectively highlights the features corresponding to the true label in the data.
      - Finally, it returns the modified tensor “x_”.
3. Define the custom neural network model “Net”, which consists of multiple layers of the custom “Layer” class.
   a. Initialization:
      - The “__init__” method is the constructor for the “Net” class.
      - It takes one argument, “dims”, which is a list representing the dimensions of the layers in the network.
      - It then creates a list called “self.layers” to store the layers of the neural network.
   b. Layer Initialization:
      - Within the “__init__” method, a loop iterates over the dimensions in the “dims” list. It creates instances of the custom “Layer” class and appends them to the “self.layers” list.
   c. Prediction (“predict” method):
      - The “predict” method takes a tensor “x” as input, which is a batch of data samples.

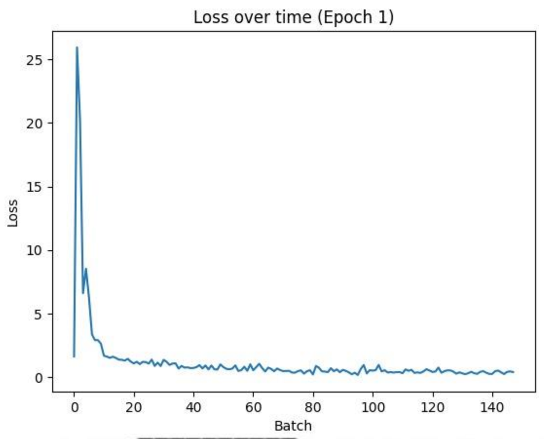
- It initializes an empty list called “goodness_per_label” to store the goodness values for each label.
- It then iterates over labels, and for each label, it calls the “overlay_y_on_x” function to overlay the label onto the input “x”, creating a new tensor “h”.
- Inside the inner loop, it iterates over the layers in “self.layers”, applying each layer to the tensor “h” and computing the goodness values for that label.
- The goodness values are stored in a list called “goodness” for each layer.
- The sum of these goodness values for all layers is computed, and a tensor containing these sums is stored in “goodness_per_label”.
- Finally, the label with the highest goodness value is selected for each sample, and the indices of the maximum values are returned. This effectively predicts the label for each input sample.

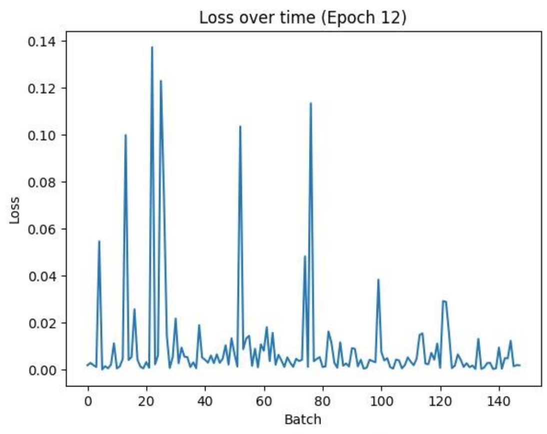
d. Training (“train” method):
- The “train” method is used for training the network.
- It takes two tensors, “x_pos” and “x_neg”, which are batches of positive and negative examples, respectively.
- It iterates through the layers in “self.layers”, and for each layer, it calls the “train” method of that layer.

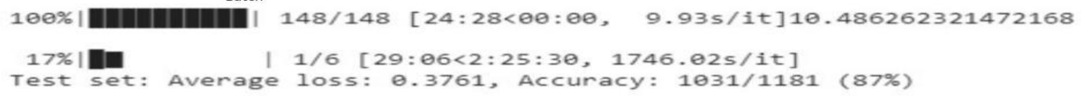
4. Define the “Layer” class:
   a. This class represents a layer in the neural network.
   b. It includes a 1D convolutional layer, ReLU activation, an Adam optimizer, and a training function.
   c. The “forward” method applies convolution and activation functions - responsible for performing a forward pass through this layer. It takes an input tensor “x” and computes the forward pass by:
      - Normalizing “x” along dimension 1 (along the batch dimension).
      - Applying the linear transformation to the normalized “x”, which involves matrix multiplication with the layer's weights and adding the bias term.
      - Applying the ReLU activation function to the result of the linear transformation.
      - The result of this forward pass is returned as the output of the layer.
   d. The “train” method trains the layer for a specified number of epochs:
      - It takes two input tensors, “x_pos” and “x_neg”, which are batches of positive and negative examples used for training.
      - Forward pass through the layer for both positive and negative examples.
      - Computes the goodness values (“g_pos” and “g_neg”) by taking the mean of the squared values of the output of the layer.
      - Calculates the loss using a contrastive loss function, which involves a log-sum-exp operation.
      - Zeroes the gradients of the layer's parameters (“self.opt.zero_grad()”).
      - Computes gradients of the loss with respect to the layer's parameters (“loss.backward()”).
      - Updates the layer's parameters using the Adam optimizer (“self.opt.step()”).

Results:

CNN – applying 12 epochs of training we got the following results:

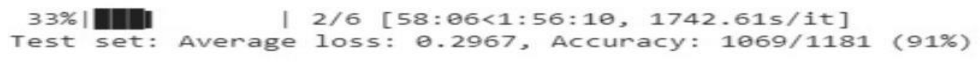




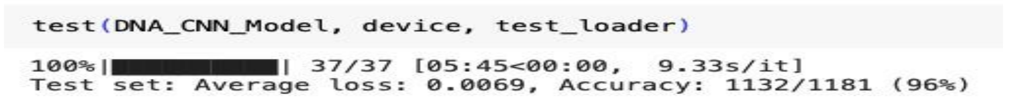


```
100%|██████████| 148/148 [24:28<00:00,  9.93s/it]10.486262321472168

 17%|█▋        | 1/6 [29:06<2:25:30, 1746.02s/it]
Test set: Average loss: 0.3761, Accuracy: 1031/1181 (87%)
```



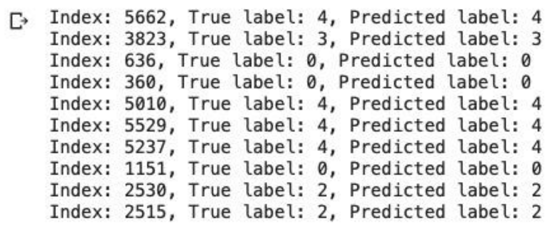
```
 33%|███▎      | 2/6 [58:06<1:56:10, 1742.61s/it]
Test set: Average loss: 0.2967, Accuracy: 1069/1181 (91%)
```



```
test(DNA_CNN_Model, device, test_loader)

100%|██████████| 37/37 [05:45<00:00,  9.33s/it]
Test set: Average loss: 0.0069, Accuracy: 1132/1181 (96%)
```

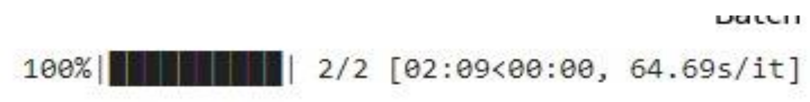
We can see that after the first epoch the loss reduces, and in the 12th epoch we can see it keeps decreasing and becoming very small. Regarding The Accuracy, we tested it throughout the training process, and we can see it improves from 87% up to 96% at the end of the training loop.
We then tried to test 10 random samples from the test data with our model and got:



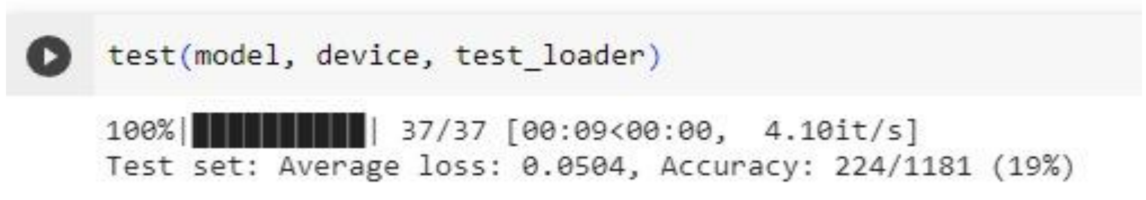
```
Index: 5662, True label: 4, Predicted label: 4
Index: 3823, True label: 3, Predicted label: 3
Index: 636, True label: 0, Predicted label: 0
Index: 360, True label: 0, Predicted label: 0
Index: 5010, True label: 4, Predicted label: 4
Index: 5529, True label: 4, Predicted label: 4
Index: 5237, True label: 4, Predicted label: 4
Index: 1151, True label: 0, Predicted label: 0
Index: 2530, True label: 2, Predicted label: 2
Index: 2515, True label: 2, Predicted label: 2
```

Meaning that the model guessed correctly, and the model can classify the test data to the correct label.

CNN with Embedding layer – after training this model for 10 epochs we got:

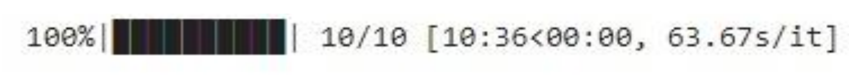


```
100%|██████████| 2/2 [02:09<00:00, 64.69s/it]
```

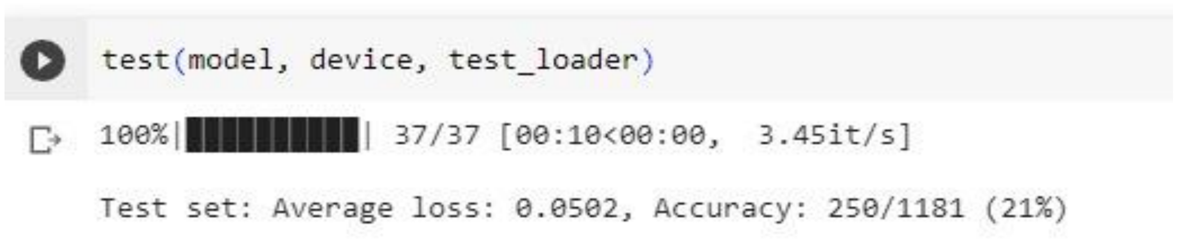


```
test(model, device, test_loader)
```

```
100%|██████████| 37/37 [00:09<00:00,  4.10it/s]
Test set: Average loss: 0.0504, Accuracy: 224/1181 (19%)
```



```
100%|██████████| 10/10 [10:36<00:00, 63.67s/it]
```



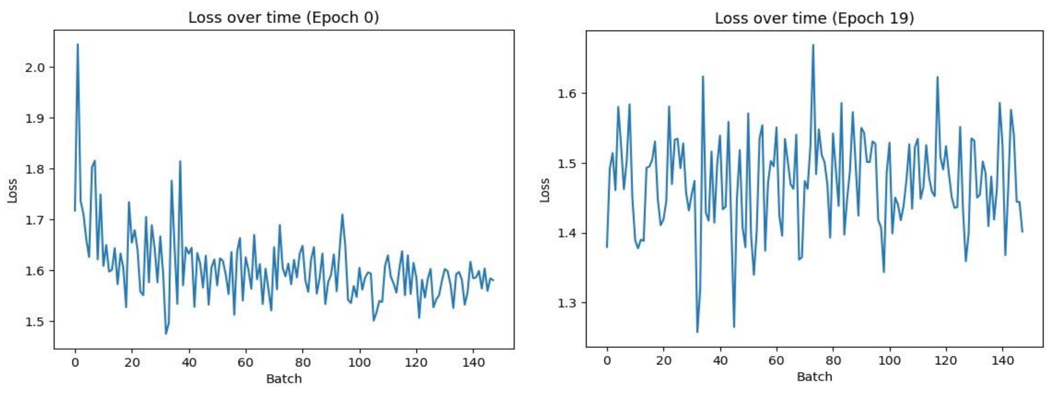
```
test(model, device, test_loader)
```

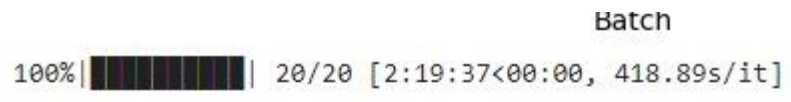
```
100%|██████████| 37/37 [00:10<00:00,  3.45it/s]

Test set: Average loss: 0.0502, Accuracy: 250/1181 (21%)
```

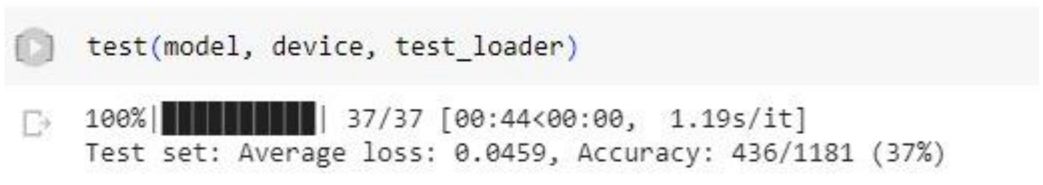
We have noticed that the accuracy we got is very low, and the loss values during the training process stayed constant and didn't decrease.
Then, we tried changing loss functions and optimizers, also changing the model parameters, but we still didn't achieve improvement.
Finally, we managed to improve the results and seeing a learning process where the loss reduces, by improving the model structure and changing its parameters.

We then trained for 20 epochs and got the following results:





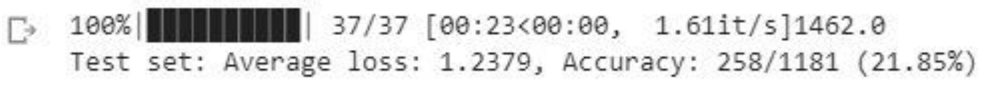
```
100%|██████████| 20/20 [2:19:37<00:00, 418.89s/it]
```



```
test(model, device, test_loader)
```

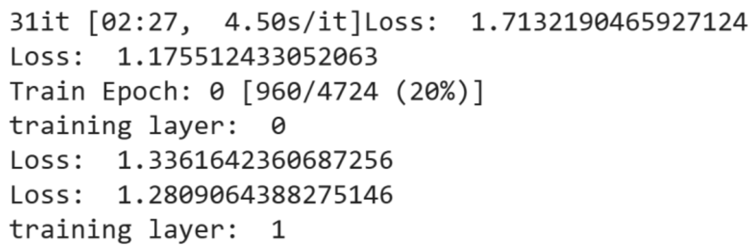
```
100%|██████████| 37/37 [00:44<00:00,  1.19s/it]
Test set: Average loss: 0.0459, Accuracy: 436/1181 (37%)
```

CNN using Forward-Forward algorithm – we wanted to see if the network performs well, so we first tried to run it on 3 epochs and got the result:

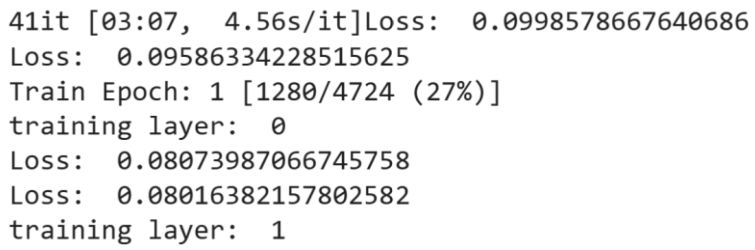


```
100%|██████████| 37/37 [00:23<00:00,  1.61it/s]1462.0
Test set: Average loss: 1.2379, Accuracy: 258/1181 (21.85%)
```

We saw that the loss reduced and training occurs, so we tried to train the network with more epochs.
We tried to run more epochs, and after a while the loss decreases, so it indicates that there is training that happens.



```
31it [02:27,  4.50s/it]Loss:  1.7132190465927124
Loss:  1.175512433052063
Train Epoch: 0 [960/4724 (20%)]
training layer:  0
Loss:  1.3361642360687256
Loss:  1.2809064388275146
training layer:  1
```



```
41it [03:07,  4.56s/it]Loss:  0.0998578667640686
Loss:  0.09586334228515625
Train Epoch: 1 [1280/4724 (27%)]
training layer:  0
Loss:  0.08073987066745758
Loss:  0.08016382157802582
training layer:  1
```

But running the test, the accuracy of the model didn't increase.
This can point that using network for image classification may not be the best option for working with sequential data (sequence of nucleotides for example).

Conclusions:

1) We have noticed that CNN's with 1D convolution layers, and the encoding we did, fit the data more so than other encodings we have tried and 2D convolutions.

2) We have managed to train and test a model with embedding layers. We believe that optimizations of the model and its parameters, better results can be achieved.

3) 2D convolution networks that gets an image as an input doesn't work very well with our data, since our data is a sequence of nucleotides, and transforming it to an image changes its structure, and can cause for misinterpretation of the connections between the nucleotides.

4) The parameters that influence the results of the model are – its structure, the model parameters, the optimizer used and the loss function that is calculated. Also, the type of data we use and its compatibility to the model we chose. We noticed that when getting low results we can try and change the parameters above, and perform different experiments with their values until achieving better results.