CS4375 Assignment 2

https://github.com/carlopizzuto/4375-HW2
Carlo Pizzuto
cxp200021

1 Introduction and Data (5 pt)

In this project, there are two neural networks that need their forward pass functions complete. The first to implement is a Feedforward Neural Network (FFNN), and the second is a Recurrent Neural Network. Both models will be trained for the task of performing a 5-class Sentiment Analysis.

The data being used for this project comes from a set of Yelp reviews. There are three distinct datasets included – Training, Testing, and Validation. Each of the datasets comes in a .json file, with each object (data point) in all datasets having a '*text*' variable and '*star*' variable. The '*text*' (independent) variable is the Yelp review, which will be passed to each Neural Network (NN) to predict the '*star*' (target) variable, which ranges from 1 to 5, inclusive.

The **Train** dataset, which will be used to train both Neural Networks, consists of 16,000 data samples, each with its '*text*' and '*star*' variables. In this dataset, all 5 possible target variables are represented equally. This means that each unique '*star*' value has an equal representation in the dataset.

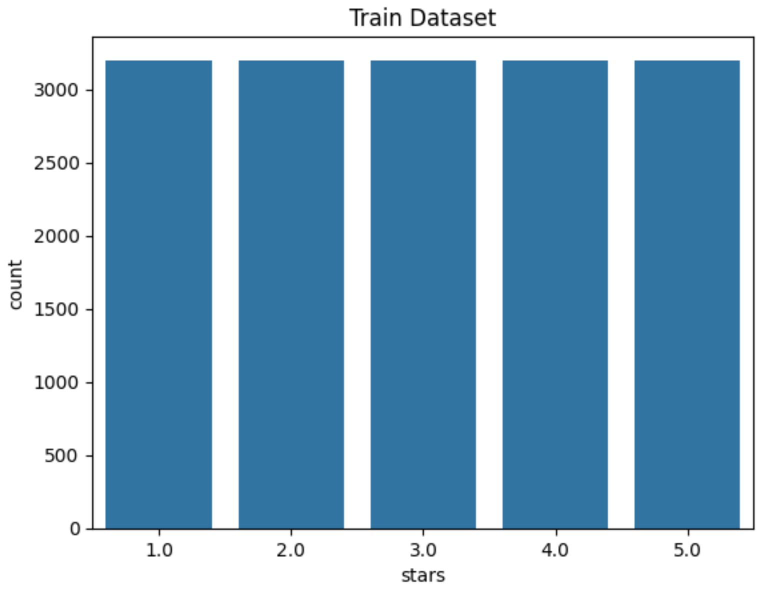


Figure 1.1 – distribution of target variable **'*stars*'** on the **Train** dataset.

The **Test** dataset consists of 800 data samples, each with its '*text*' and '*star*' variables. In this dataset, only there are only '*star*' values of 3, 4, and 5. The '*star*' value 3 is the least present, with 20% of values having a 3 '*star*' value. 4 and 5 on the other hand each have 40% representation in this dataset.

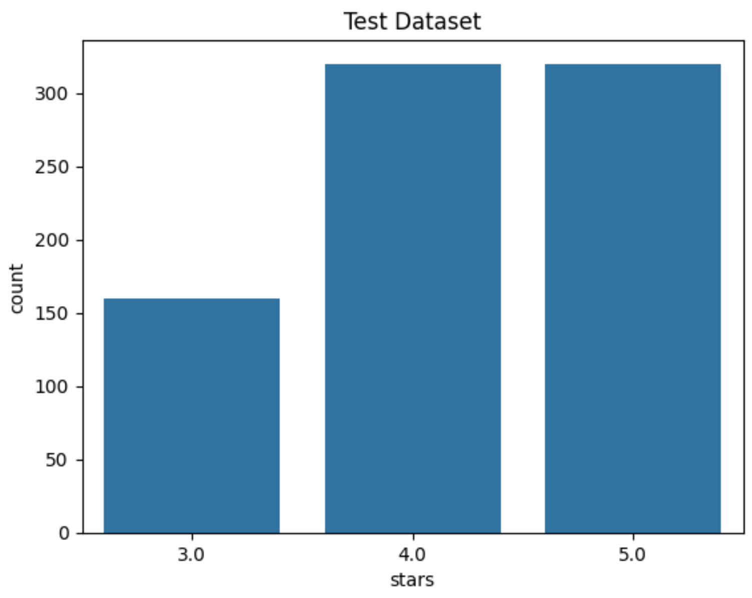


Figure 1.2 – distribution of target variable ***'stars'*** on the **Test** dataset.

The **Validation** dataset consists of 800 data samples, each with its '*text'* and '*star'* variables. In this dataset, only there are only '*star'* values of 1, 2, and 3. The '*star'* value 3 is the least present, with 20% of values having a 3 '*star'* value. 1 and 2 on the other hand each have 40%.

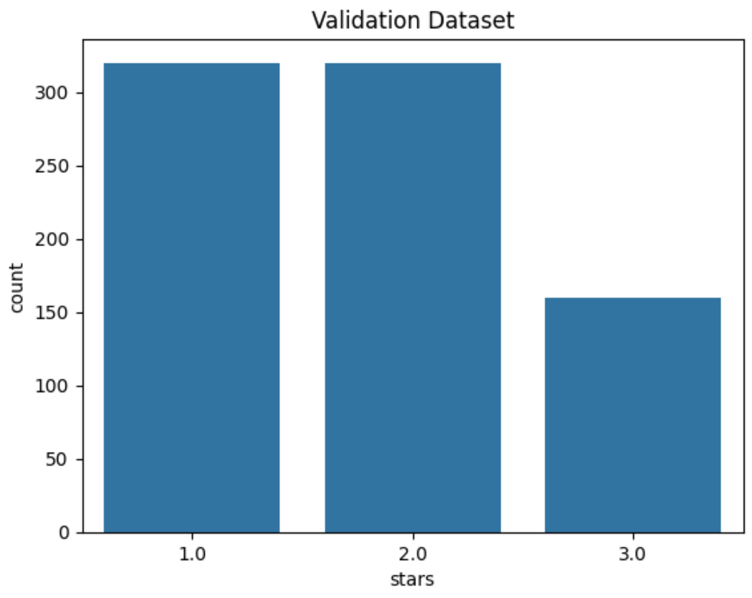


Figure 1.3 – distribution of target variable **'stars'** on the **Validation** dataset.

2 Implementations (45 pt)

2.1 FFNN (20 pt)

A Feed-Forward Neural Network (FFNN) is a type of Neural Network in which the data flows in one direction – from the input to the output – without any feedback loops. It takes the data and moves it form the input layer X through one or more hidden layers to produce an output. Each neuron in the hidden layer applies the linear transformation function $W \cdot X + b$ to the input X and passes the result through an activation function like Rectified Linear Unit *(ReLU)*. The output of the activation function is then passed to the next hidden layer's neurons as the input to the linear transformation function. This process continues for all hidden layers, and the last one (output layer) giving the output of the network.

The FFNN model I implemented only had one hidden layer, which uses the ReLU activation function, and an output layer which uses the SoftMax activation function. The SoftMax function is critical to the output since it is a category (a star from 1 to 5) and not a continuous number (e.g the price of a house). My task was to implement the forward() function for the model, which initially had the following structure.

```
def forward(self, input_vector):
      # [to fill] obtain first hidden layer representation
      # [to fill] obtain output layer representation
      # [to fill] obtain probability dist.
     return predicted_vector
```

Figure 2.1.1 – Initial forward() function structure of the FFNN model.

To fill in this function, I referred to the model's __init__() function, which initializes a two-layer Feed-Forward Neural Network. It has h hidden units provided as an argument to the file, a hidden layer with the ReLU activation function, an output layer with the softmax activation function, and uses the negative log-likelihood loss function for training.

```
def __init__(self, input_dim, h):
      super(FFNN, self).__init__()
      self.h = h # number of hidden units (neurons)
      #======== Hidden Layer Start ========
      self.W1 = nn.Linear(input_dim, h) # input layer to hidden layer
      self.activation = nn.ReLU() # relu - hidden layer activation function
      #======== Hidden Layer End ========
      self.output_dim = 5 # number of classes
      #======== Output Layer Start ========
      self.W2 = nn.Linear(h, self.output_dim) # hidden layer to output layer
      self.softmax = nn.LogSoftmax() # softmax - output layer activation function
      #======== Output Layer End ========
      self.loss = nn.NLLLoss() # cross-entropy/negative log likelihood loss function
```

Figure 2.1.2 – FFNN model's __init__() function.

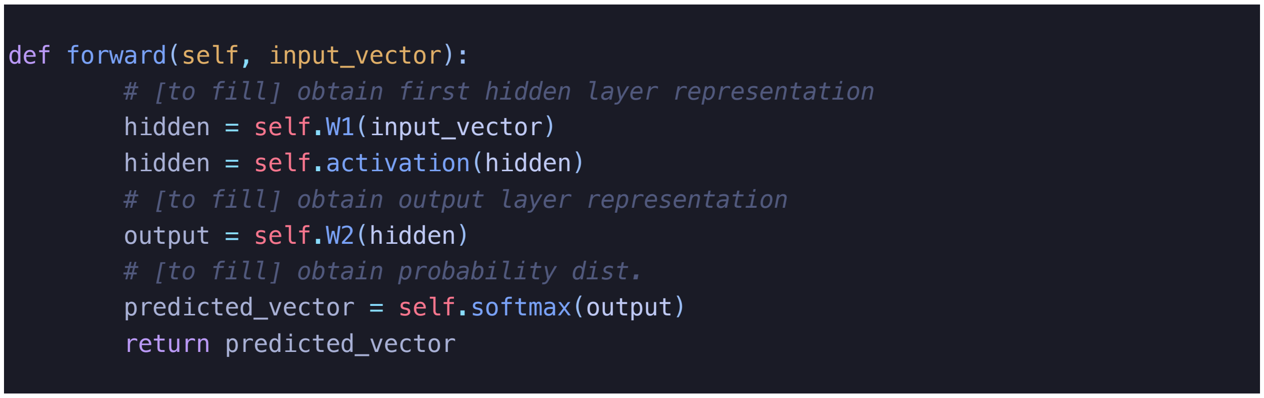
- **input_dim:** size of the input vectors (word embedding dimensions).
- **self.h:** number of neurons in the hidden layer.
- **self.W1:** computes the hidden layer's linear transformation and transforms the data from input_dim (vocabulary size) dimensions to h dimensions. It is implemented with the PyTorch module, which has the nn.Linear() function.
- **self.activation:** hidden layer's activation function Rectified Linear Unit (ReLU). It is also implemented with the PyTorch module, which has the nn.ReLU() function. This function returns:
    - 0 if x is less than 0.
    - x if x is greater than or equal to 0.
- **self.output_dim:** number of possible output classes.
- **self.W2:** same as self.W1 but transforms the data from h dimensions to 5 (unique star ratings) dimensions.
- **self.softmax:** output layer's activation function SoftMax. It is also implemented with the PyTorch module, which has the nn.LogSoftmax() function. This function takes an input vector x and computes the softmax probabilities for each element in x. It is defined as:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

- **self.loss:** computes the loss, or error, between the predicted output and the actual value. It uses the negative log likelihood loss function, which I will cover in section 3.1 Evaluations.

Given the __init__() function, I could now fill out the forward() function. This function

i. gets an argument *input_vector* of size n,
ii. applies the linear transformation function from **self.W1** to the input *input_vector* (size n) and saves the result in *hidden* (size h),
iii. passes *hidden* (size h) through the **self.activation** function *ReLU* and saves the result in the same variable *hidden* (size h),
iv. applies the linear transformation function from **self.W2** to the input *hidden* (size h), and saves the result in *output* (size 5),
v. applies softmax from **self.softmax** to *output* (size 5) to predict the probabilities of each class and saves it in *predicted_vector* (size 5),
vi. returns *predicted_vector*.

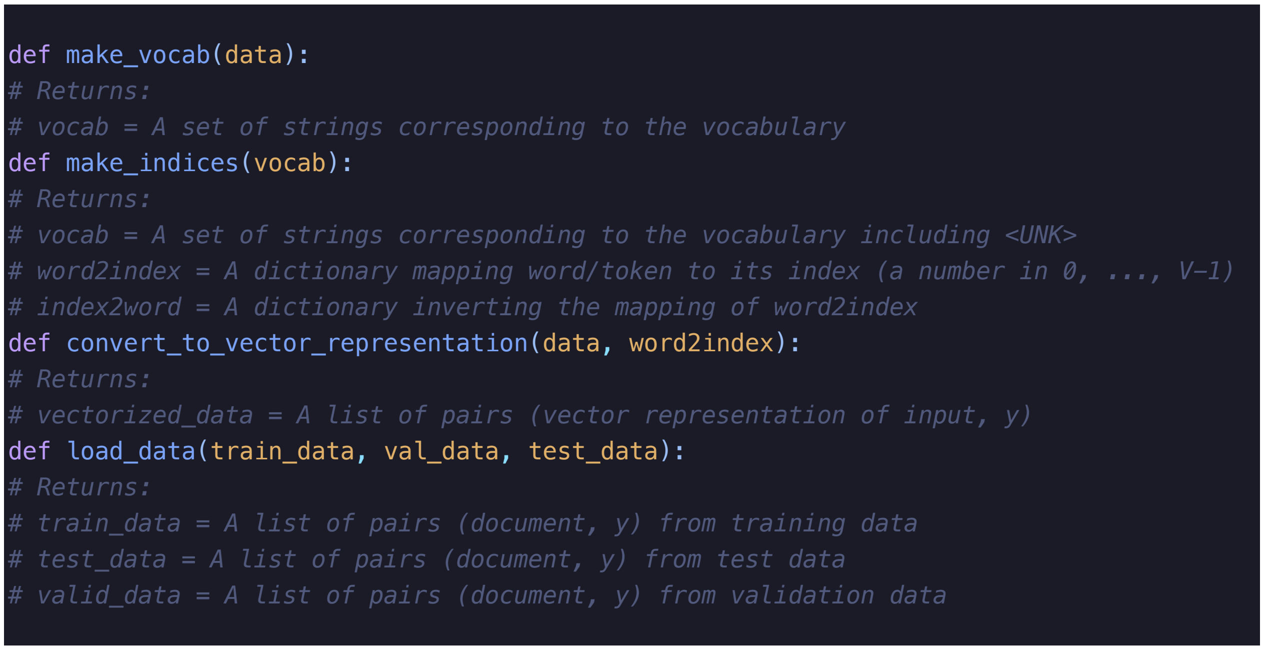


```python
def forward(self, input_vector):
        # [to fill] obtain first hidden layer representation
        hidden = self.W1(input_vector)
        hidden = self.activation(hidden)
        # [to fill] obtain output layer representation
        output = self.W2(hidden)
        # [to fill] obtain probability dist.
        predicted_vector = self.softmax(output)
        return predicted_vector
```

Figure 2.1.3 – Final forward() function for FFNN model.

Besides the __init__() and forward() functions, the FFNN model also contains an additional method `compute_Loss(self, predicted_vector, gold_label)` which takes in a prediction vector *prediction_vector* and the actual value vector *gold_label* and computes the loss with the activation function at self.loss (negative log likelihood loss function).

The file also includes some additional methods, each with their relevant comments.



```python
def make_vocab(data):
# Returns:
# vocab = A set of strings corresponding to the vocabulary
def make_indices(vocab):
# Returns:
# vocab = A set of strings corresponding to the vocabulary including <UNK>
# word2index = A dictionary mapping word/token to its index (a number in 0, ..., V-1)
# index2word = A dictionary inverting the mapping of word2index
def convert_to_vector_representation(data, word2index):
# Returns:
# vectorized_data = A list of pairs (vector representation of input, y)
def load_data(train_data, val_data, test_data):
# Returns:
# train_data = A list of pairs (document, y) from training data
# test_data = A list of pairs (document, y) from test data
# valid_data = A list of pairs (document, y) from validation data
```

Figure 2.1.4 – Additional functions present in the FFNN file.

The __main__() function present in the file is used to train the model. It takes in several command line arguments, including the number of hidden dimensions (--hidden_dim), number of epochs (--epochs), paths to the data (--val_data, …), and Boolean flags to specify training or inference (my addition, --do_train, --do_infer). After parsing the arguments into the *args* Namespace variable (like a python dictionary), the function fixes random seeds and loads and vectorizes the data using the functions in Figure 2.1.4.

After handling the data, the FFNN model is initialized with the arguments input_dim being the length of vocab variable (*len(vocab)*, with vocab = *make_vocab(train_data)*), and h to the argument from the parser (*args.hidden_dim*). This, under the hood, calls the __init__(input_dim, h) function from Figure 2.1.2. Then, a PyTorch Stochastic Gradient Descent (SGD) optimizer is initialized with

- the model parameters - which tell the optimizer which parameter o update in training,
- learning rate of 0.01 - which controls the step size during gradient descent,
- momentum of 0.9 - which helps accelerate the optimizer in the relevant direction.

If the do_train flag is present in the arguments, the function proceeds with training the model. For each epoch in range(args.epochs): The model is set to training mode and the optimizer's gradients are zeroed. The data is then shuffled randomly and split into minibatches of 16 samples. Next, the model makes predictions and gets the training loss. After this, the model predicts on the validation dataset to get the validation accuracy.

If the do_infer flag is present, the model chooses 10 samples from the test dataset and for each sample, it makes a prediction and prints it, as well as the actual target variable and the loss for that sample. At the end, it gives an average loss.

2.2 RNN (25 pt)

A Recurrent Neural Network (RNN) is a type of neural network designed to process sequential data. It does so by including loops in its architecture, allowing information to be retained across time steps. Unlike FFNNs, RNNs pass data not only from the input to the output, but also back to the previous layers. Each hidden layer at time step t calculates the hidden state h_t using the formula $h_t = f(\, W_x \cdot x_t + \, W_h \cdot h_{t-1} + b\,)$, where x_t is the input at time step t, h_{t-1} is the hidden state from the previous time step, W_x and W_h are the weight matrices, b is the bias term, and f is an activation function – typically Tanh or Sigmoid.

The hidden state h_t then feeds into the next layer and influences future time steps. The final output y_t at time step t is calculated as $y_t = g(\, W_y \cdot h_t + c\,)$, where W_y is the weight matrix for the output layer, W_x is the bias term for the output and W_x is an output specific activation function – SoftMax for classification in our case. My task was to implement the forward() function for this model, which initially had the following structure.

```python
def forward(self, inputs):
        # [to fill] obtain hidden layer representation
        # [to fill] obtain output layer representations
        # [to fill] sum over output
        # [to fill] obtain probability dist.
        return predicted_vector
```

Figure 2.2.1 – Initial forward() function for the RNN model

To implement this function, I again referred to the model's __init__() function. Here, a two-layer RNN model is defined, which has a single RNN layer with h neurons that uses the Tanh activation function. This layer is followed by a linear output layer which uses the LogSoftMax for classification and the negative log-likelihood loss function for training.

```python
def __init__(self, input_dim, h):
        super(RNN, self).__init__()
        self.h = h  # number of hidden units (neurons)
        self.numOfLayer = 1  # number of RNN layers stacked together
        #======== RNN Layer Start ========
        # input_dim: size of input vectors
        # h: number of hidden units
        # nonlinearity: activation function for hidden layer
        self.rnn = nn.RNN(input_dim, h, self.numOfLayer, nonlinearity='tanh')
        #======== RNN Layer End ========
        self.output_dim = 5  # number of classes for star ratings 1-5
        #======== Output Layer Start ========
        self.W = nn.Linear(h, self.output_dim)  # hidden layer to output layer
        self.softmax = nn.LogSoftmax(dim=1)  # log softmax for output probabilities
        #======== Output Layer End ========
        self.loss = nn.NLLLoss()  # negative log likelihood loss function for training
```

Figure 2.2.2 – RNN model's __init__() function.

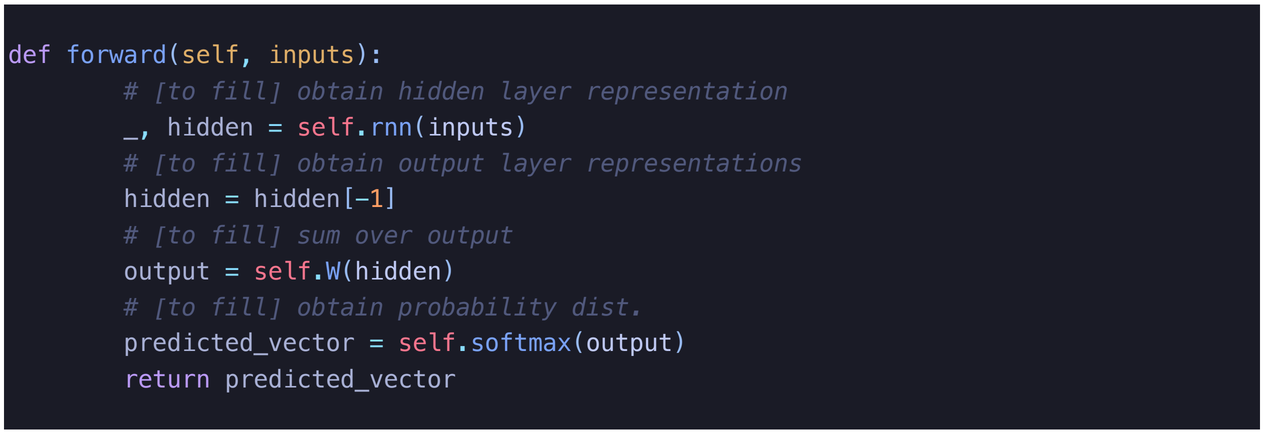
- **input_dim:** size of the input vectors (word embedding dimensions).
- **self.h:** number of neurons in the hidden layer.
- **self.numIfLayer:** number of RNN layers stacked together
- **self.RNN:** core RNN layer implemented with PyTorch's nn.RNN module. It takes in the input vector size, the number of hidden units, the number of stacked RNN layers and the nonlinearity function Tanh – which maps the values to the range [-1, 1].
- **self.output_dim:** number of possible output classes (5).
- **self.W:** linear transformation output layer that maps from hidden state dimensions (h) to output dimension (output_dim), implemented using PyTorch's nn.Linear module.
- **self.softmax:** output layer's activation function LogSoftMax. It is also implemented with the PyTorch's nn.LogSoftmax() module. This function takes an input vector x and computes the log of softmax probabilities for each element in x. It is defined as:

$$\text{Log}SoftMax(x_i) = x_i \text{ - log}\left(\sum_{j=1}^{n} e^{x_j}\right)$$

- **self.loss:** computes the loss, or error, between the predicted output and the actual value. It uses the negative log likelihood loss function, which I will cover in section 3.1 Evaluations.

Given the __init__() function, I could now fill out the forward() function. This function

i. gets an argument *input_vector* of size n,
ii. applies the RNN transformation from **self.RNN** (including the Tanh activation function) to the input *input_vector* (size n) and saves the result in *hidden* (size h),
iii. extracts the final hidden state at hidden[-1] (last element) and saves it in the same variable *hidden*,
iv. passes *hidden* (size h) through the linear transformation layer **self.W** and saves the result in the variable *output* (size 5),
v. applies softmax from **self.softmax** to *output* (size 5) to predict the probabilities of each class and saves it in *predicted_vector* (size 5),
vi. returns *predicted_vector*.



```
def forward(self, inputs):
        # [to fill] obtain hidden layer representation
        _, hidden = self.rnn(inputs)
        # [to fill] obtain output layer representations
        hidden = hidden[-1]
        # [to fill] sum over output
        output = self.W(hidden)
        # [to fill] obtain probability dist.
        predicted_vector = self.softmax(output)
        return predicted_vector
```

Figure 2.2.3 – Final forward() function of the RNN model

Besides the __init__() and forward() functions, the RNN model also contains an additional method `compute_Loss(self, predicted_vector, gold_label)`, which is the same as the FFNN model – computes the loss of the predicted vector using the *negative log likelihood loss function*.

The only other function besides __main__() is load_data(test, val, train), which processes the data from JSON files into more appropriate Python lists for training.

3 Experiments and Results (45 pt)

3.1 Evaluations (15 pt)

The Feed-Forward Neural Network (FFNN) is evaluated using both the accuracy and loss metrics. For each training epoch, the model processes data in minibatches of 16 samples and tracks the number of correct predictions and total samples. These tracked values are then used to compute the accuracy – correct / total. The loss is computed using the Negative Log Likelihood (NLL) loss function after the LogSoftMax activation, implementing cross-entropy loss. After each epoch, the model performs validation using the same metrics (but not updating the weights). This validation step helps monitor for overfitting and gives a sense of the model's generalization performance.

The Recurrent Neural Network (RNN) uses a similar evaluation method but also implements an early stop feature. Like the FFNN, this model processes the training data in minibatches of 16 samples tracks accuracy (correct / total) as well as the NLL loss. However, the RNN adds an overfitting prevention mechanism, which compares the current validation accuracy with the previous epoch's validation accuracy. If the validation accuracy decreases but training increases, the training stops to prevent overfitting. This way, the best model is the one from the previous epoch, which has the highest validation accuracy.

3.2 Results (30 pt)

FFNN

4 Analysis (bonus: 10 pt)

5 Conclusion and Other (bonus: 5 pt)