Lab 4: RNN - Sequence tagging

Author: Duy Nghia TRAN

Architectures and results obtained in the lab session

The very first recurrent network

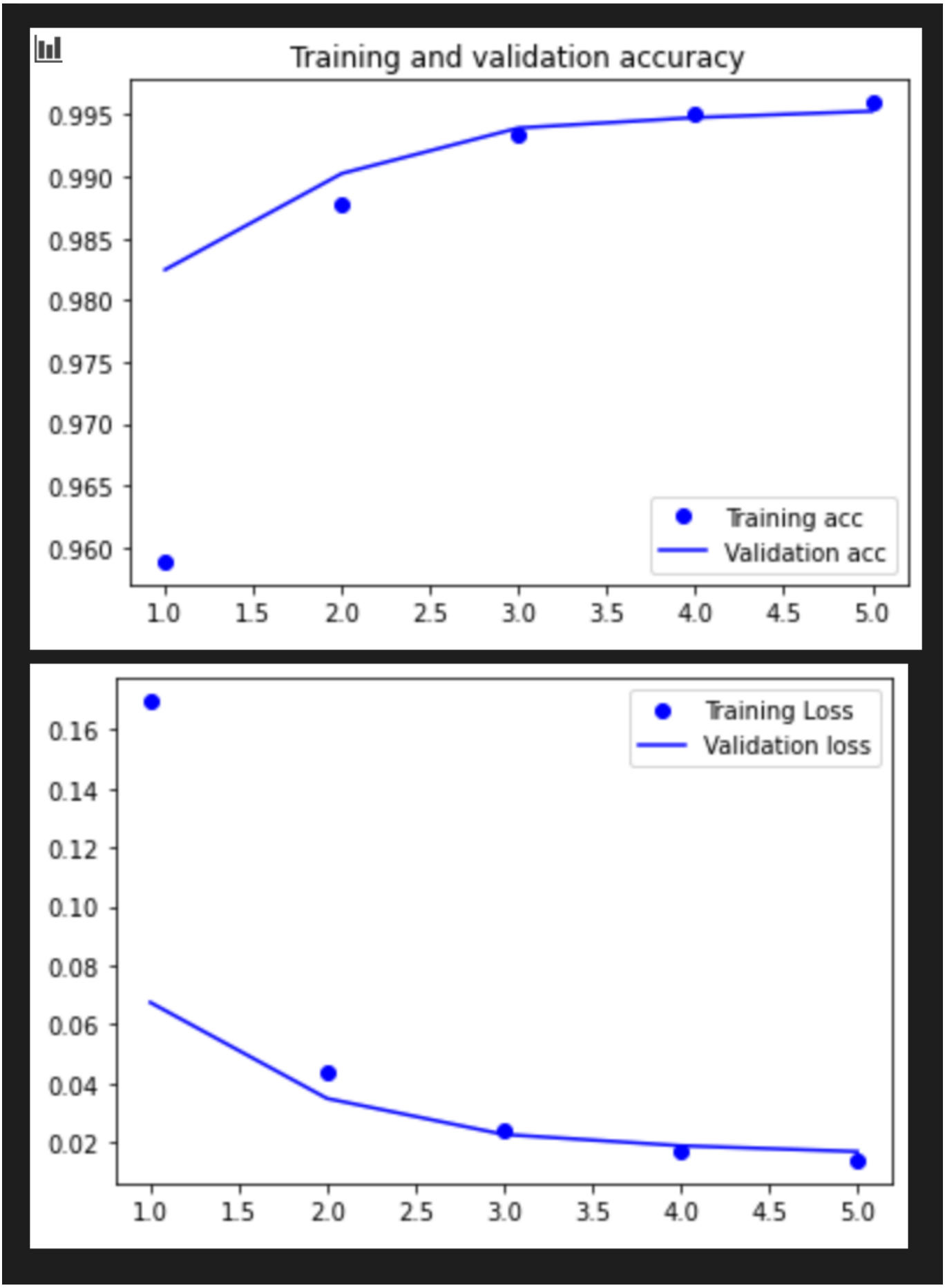
On this first simple network, I only had three layers in the model.

```
model = Sequential()
model.add(Embedding(EMBEDDING_COUNT, EMBEDDING_DIM, input_length=max_len))
model.add(SimpleRNN(EMBEDDING_DIM, return_sequences=True))
model.add(Dense(len(ners) + 2, activation='softmax'))

model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = True
model.summary()
```

The first layer is an embedding layer that has a 2D weight matrix. This matrix is set to be the one that is computed from the GloVe embeddings 6B.100d. In this model I left the first layer to be trainable, and compile it with the RMSProp optimizer.

```
model.compile(optimizer = optimizers.RMSprop(), loss=
'categorical_crossentropy', metrics=['acc'])
```



This model has reached 70% of accuracy which is pretty impressive for such a simple network.

LSTM network

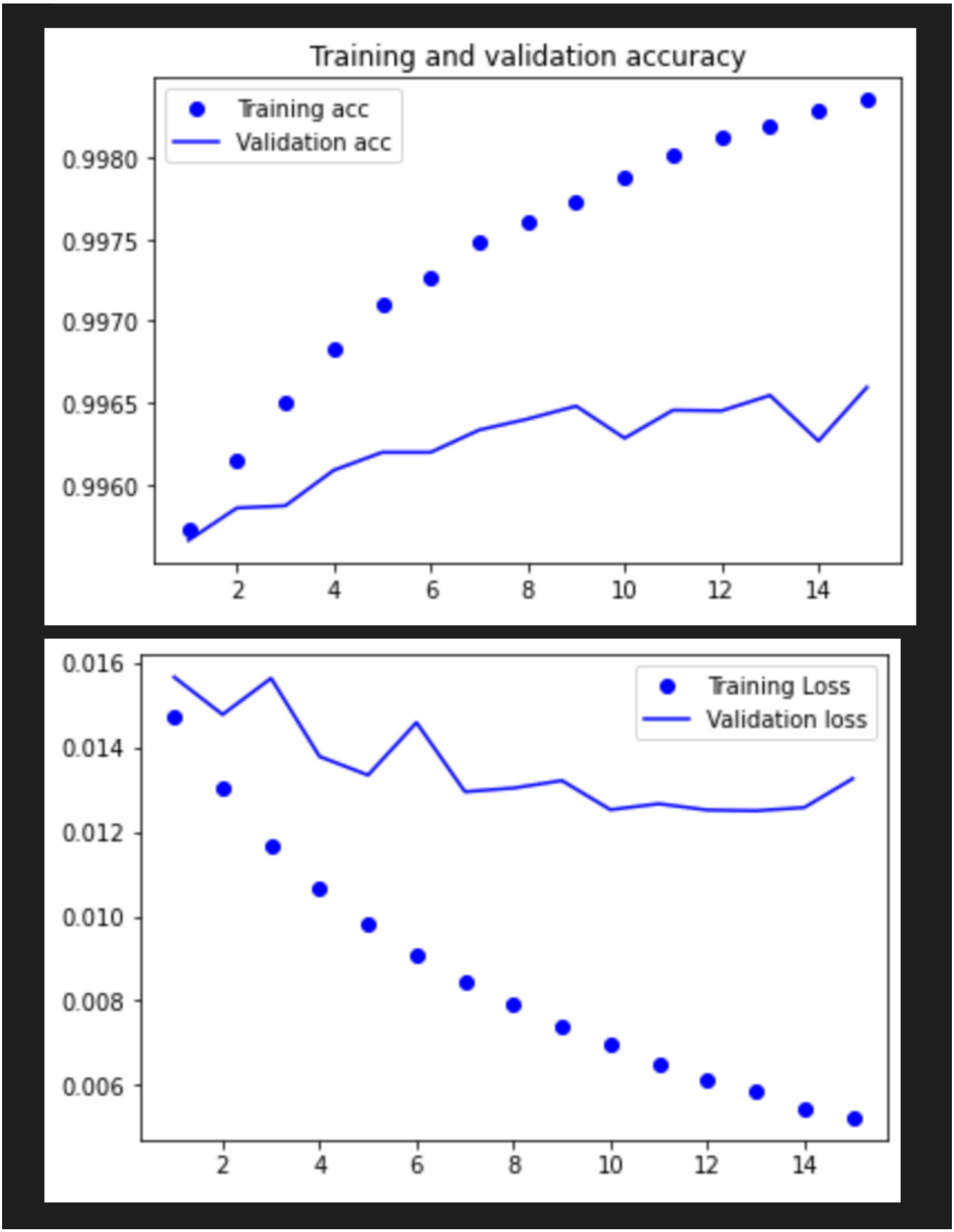
Because the SimpleRNN layer suffers the vanishing gradient problem, I decided to replace it with the LSTM which allows information to be carried along the network and be used at later timesteps.

```
model_lstm = Sequential()
model_lstm.add(Embedding(EMBEDDING_COUNT, EMBEDDING_DIM,
input_length=max_len))
model_lstm.add(LSTM(EMBEDDING_DIM, return_sequences=True))
model_lstm.add(Dropout(0.5))
model_lstm.add(Dense(len(ners) + 2, activation='softmax'))

model_lstm.layers[0].set_weights([embedding_matrix])
model_lstm.layers[0].trainable = True

model_lstm.summary()
```

A Dropout layer was also added to fight overfitting. Below is the training process.



The accuracy obtained using this model was 75%.

Bidirectional LSTM

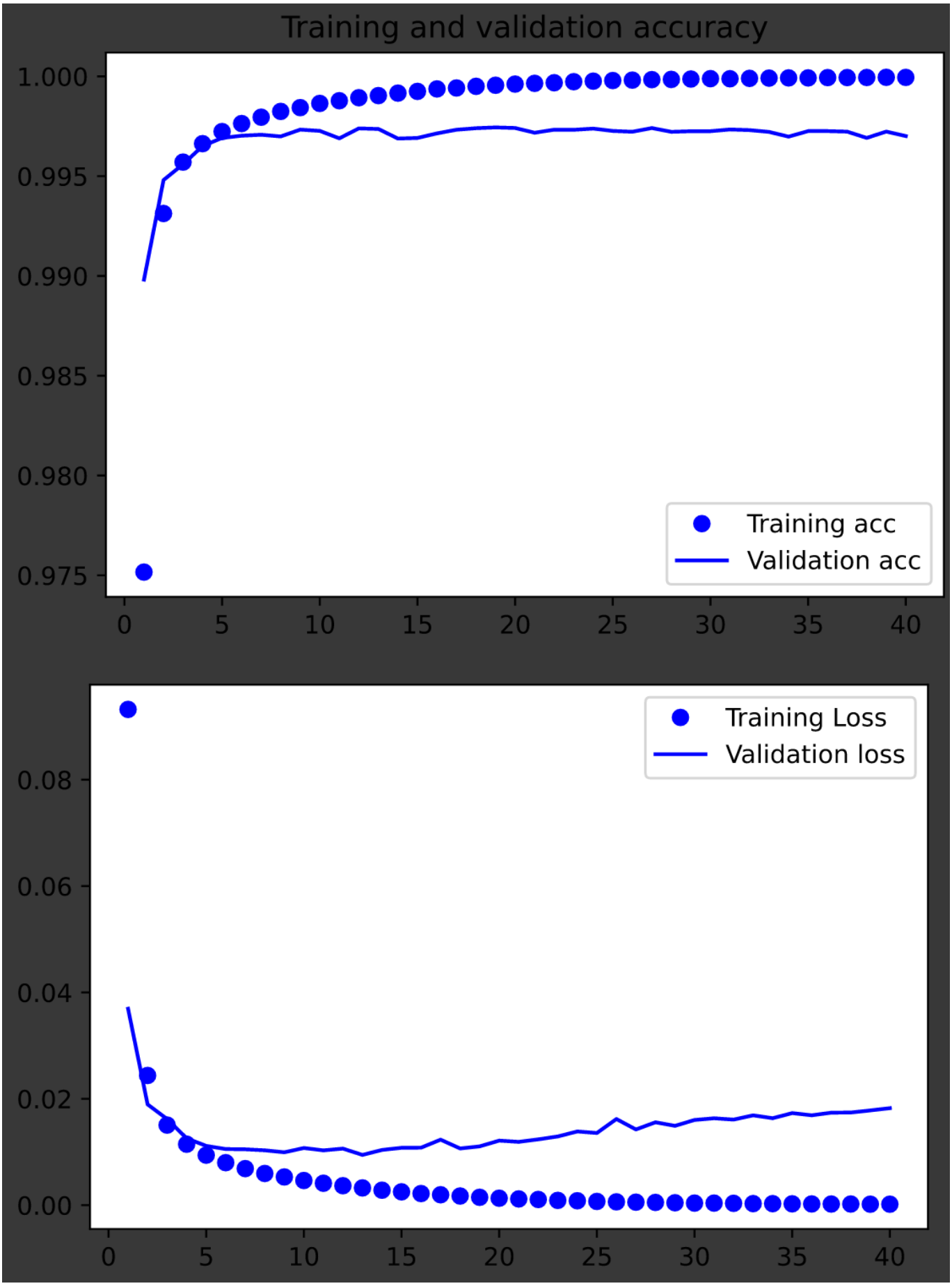
An enhancement I could include in the previous model is the Bidirectional layer which processes the information in two directions and then merges the representations. It performed slightly better than the

LSTM only model.

```
model_enhanced = Sequential()
model_enhanced.add(Embedding(EMBEDDING_COUNT, EMBEDDING_DIM,
input_length=max_len))
model_enhanced.add(Bidirectional(LSTM(EMBEDDING_DIM,
return_sequences=True)))
model_enhanced.add(Dropout(0.5))
model_enhanced.add(Dense(len(ners) + 2, activation='softmax'))

model_enhanced.layers[0].set_weights([embedding_matrix])
model_enhanced.layers[0].trainable = True

model_enhanced.summary()
```



On this model, the accuracy was 78.9%, just a bit better than the previous one.

The best model

The previous model seemed to overfit quickly so I have experimented with different combinations to solve this issue. The best performed one was the following:

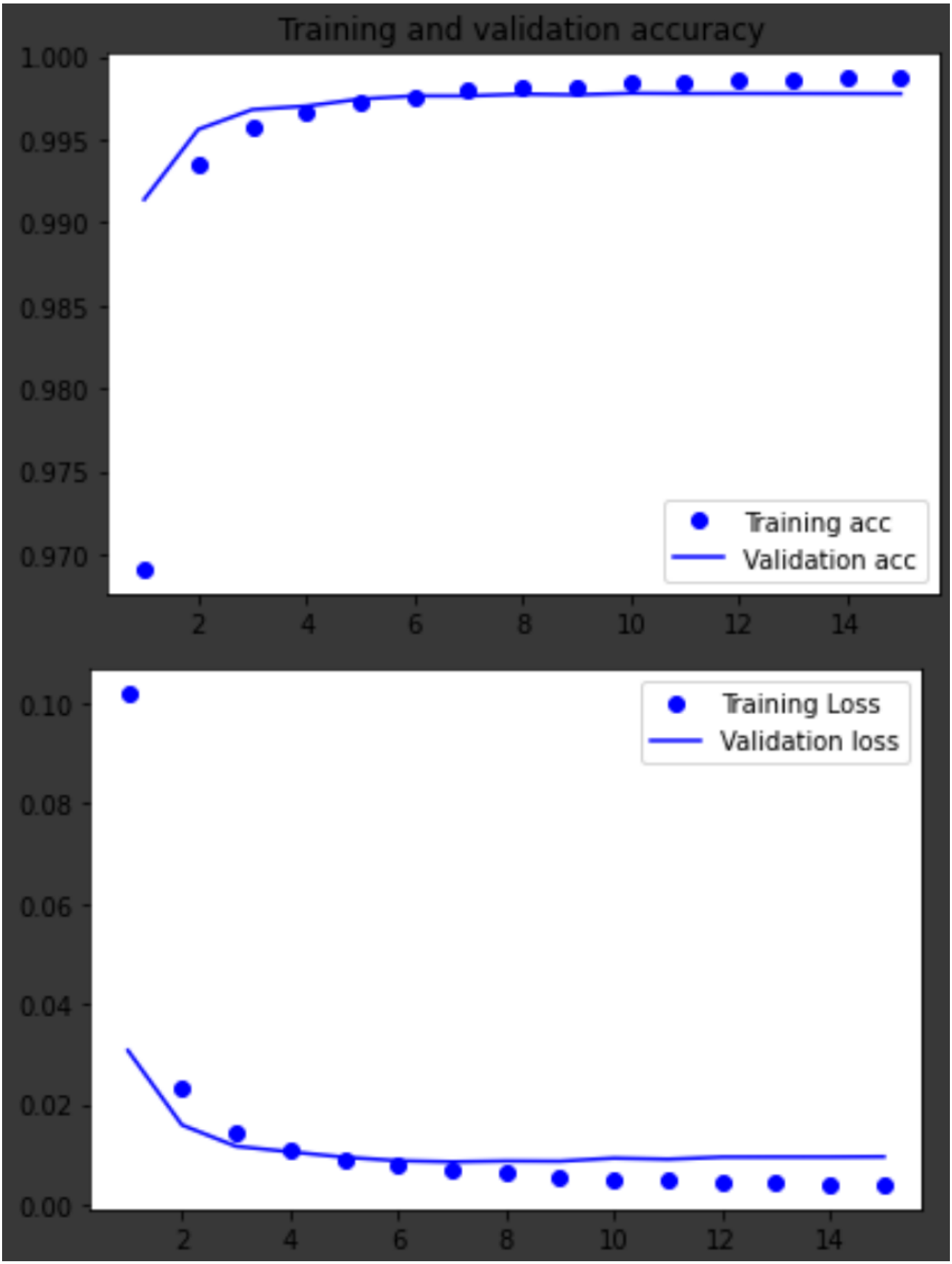
```
model_final = Sequential()
model_final.add(Embedding(EMBEDDING_COUNT, EMBEDDING_DIM,
input_length=max_len))
model_final.add(Dropout(0.2))
model_final.add(Bidirectional(LSTM(EMBEDDING_DIM, return_sequences=True)))
model_final.add(Dropout(0.2))
model_final.add(Dense(len(ners) + 2, activation='softmax'))

model_final.layers[0].set_weights([embedding_matrix])
model_final.layers[0].trainable = False

model_final.summary()
```

Several crucial changes have been made:

- There are two Dropout layers instead of one, each with a rate of 0.2 instead of 0.5. I found this to work best because not too much data is ignored.
- The embedding layer is frozen, my reasoning was because the amount of data that I worked with was relatively small so it does not worth it to retrain the embedding.



Not only the training time was drastically reduced, the accuracy is also increased to 83.86%

One interesting thing I have found out while experimenting with different models was the choice of the right hyperparameters. On my last model, I have used the RMSProp optimizer with 0.2 Dropouts but a slightly better performance was observed when using the Adam optimizer and 0.5 Dropouts.

Reflections on the article by Akbik et al. (2018)

Ihe article proposes contextual string embeddings at a character-level and its its use in sequence labeling architectures.

Contextual string embeddings

The work uses LSTM at a character-level, allowing the model to posses hidden state of each character in the sequence. Combined with a two ways forward-backward neural network, the authors' approach generates embedding that not only contains information about the words itself but also the about the surroundings (which we can refer to as context).

Embeddings stacking

In their work, the authors have combined different types of embeddings by concatenating them into a vector that is the final word vector. They have included the GloVe embedding and their own representation to achieve greater word-level semantics.

Comparison with the lab architecture

	Lab	**Authors**
Model	BiLSTM-CRF	BiLSTM-CRF
Embeddings	GloVe	Contextual string + GloVe + task-trained character

The main difference relies in the choice of embeddings. The authors' embeddings are very effective since they are able to produce different embeddings for the same word in different contexts while my embeddings are static.

Their model was more complex (2048 hidden states) and was also trained during more time and with a large amount of data (1-billion word corpus).

Performance

The authors have made a comparison table on their proposed architecture and the existing earlier approaches. They have different configurations where they concatenate different embeddings but overall they have outperformed the state-of-the-art approaches.

On the NER-English set, they have reached the f1-score of 93.09%.