UiO - V21: IN5550

Fábio Rodrigues Pereira - fabior@uio.no
Per Morten Halvorsen - pmhalvor@uio.no
Eivind Grønlie Guren - eivindgg@ifi.uio.no
UiO GitHub repository: https://github.uio.no/fabior/IN5550/tree/master/Oblig2

March 13th, 2021

Contents

1 Introduction

The purpose of this project was to get comfortable with word embeddings, integrate pre-trained embeddings in neural architecture, and implement a recurrent neural network for a classification task.

1.1 Source code

All code made for this project is written in Python v.3.9, and found in the GitHub repository at https://github.uio.no/fabior/IN5550/tree/master/Oblig2. The repository contains:

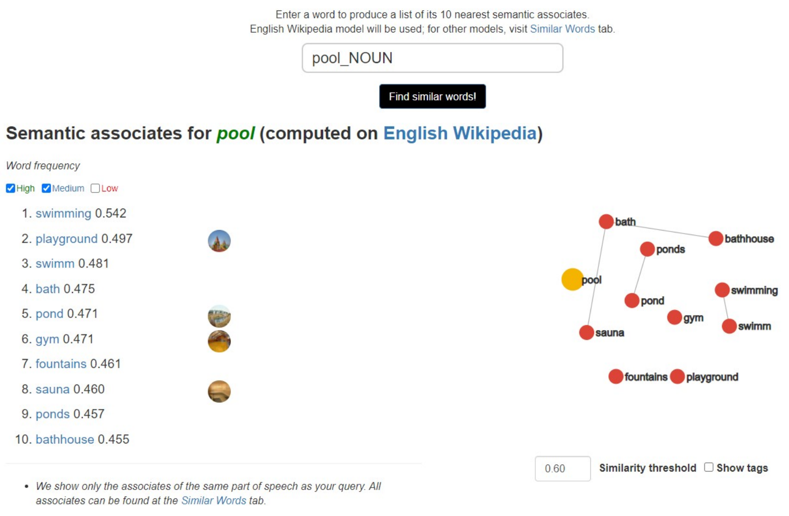
- `evaluation.py` - Python file provided the instructors will use to compare the prediction results with their gold standards.
- `ex4.py` - Code for exercise 4
- `ex5.py` - Code for exercise 5
- `ex6.py` -Code for exercise 6
- `data/` - Directory containing the data utilized in this project.
- `outputs/` - Directory containing the collection of tuning results.
- `packages/` - Directory containing the collection of Python methods utilized in the project.
- `packages/ann_models.py` - Python file containing both the feed neural network and the recurrent neural network models.
- `packages/studies.py` - Python file containing the studies used for tuning hyper-parameters on the MLP model.
- `packages/plot.py` - Python file used for plotting the results found in tuning step.
- `packages/preprocessing.py` - Python file containing classes that load, split, and preprocess the data.
- `packages/rnn_studies.py` - Python file containing studies written to tune the parameters of the RNN model.
- `predict_on_test.py` - The file used to generate the results for evaluation
- `report/` - Directory containing the written report in latex and pdf.

2 Discussion

2.1 WebVectors

The first task in this project was to play around with the *WebVectors* web service [1]. This web-app takes in a word and outputs a graph with 11 nodes, representing the 10 most similar words to the main input node. Edges are drawn between each of the nodes that are also very similar. The tool attempts to visualize the hyper-dimensional embedding space generated by LGT at UiO [2] through similarities.

[1] http://vectors.nlpl.eu/explore/embeddings/en/
[2] http://vectors.nlpl.eu/explore/embeddings/en/models/


Enter a word to produce a list of its 10 nearest semantic associates.
English Wikipedia model will be used; for other models, visit Similar Words tab.

pool_NOUN

Semantic associates for *pool* (computed on English Wikipedia)

Word frequency

☑ High ☑ Medium ☐ Low

1. swimming 0.542
2. playground 0.497
3. swimm 0.481
4. bath 0.475
5. pond 0.471
6. gym 0.471
7. fountains 0.461
8. sauna 0.460
9. ponds 0.457
10. bathhouse 0.455

0.60 Similarity threshold ☐ Show tags

- *We show only the associates of the same part of speech as your query. All associates can be found at the Similar Words tab.*

Figure 1: WebVectors

2.2 Working with pre-trained models locally

The next step was to load and play with the embeddings locally. This meant, reading them from the shared directory `/cluster/shared/nlpl/data/vectors/latest` on our saga login nodes. More specifically, we were to use the embedding models found at `200.zip` and `29.zip` to generate similarity tables for each word in the sentence:

> *Almost all current dependency parsers classify based on millions of sparse indicator features.* [3]

The provided file `play_with_gensim.py` gave a skeleton of how these embedding should be loaded and how the results should be presented. Script `ex4.py` shows our implementation of a solution for this task.

When running this file, the user will be asked for the directory path where each of the models `200.zip` and `29.zip` lies. If the paths are correct, the output of the script can be found in `outputs/ex4/`.

The ids `200` and `29` refer to the embedding models trained on the English Wikipedia corpus and the Gigaword corpus [4], respectively. They both span over 300 dimensions and have respective vocabulary sizes of about 250,000 and 300,000.

One notable difference between the two embedders is the apparent, expected case-folding of `29`, that is not present in embedder `200`. This prevents the same word from appearing in the similarity table for a specific input word, just with a capitalized first letter. This could be viewed as helpful for returning more relevant results, naturally eliminating cluttering repeats, although it would require a slightly different preprocessing technique than what was asked for here. Consequently, this tells us that the Gigaword embedding model should have more unique words and generate results for a larger span of input vocabularies.

However, from the words tested in this example, embedder `29` actually has *fewer* matches of inputs to words in its vocabulary than embedder `200`. This hints at a difference in the domain the corpora focus on. While Gigaword has more unique words in the vocab than the Wikipedia corpus, the latter seems like a better fit for the particular domain of scientific articles.

Further, when discussing domain adaptability, neither model was able to produce results for highly technical words like `parser`, with means both should be fine-tuned if they are to be used on this type of input data.

[3] Chen and Manning, 2014
[4] http://vectors.nlpl.eu/repository/

2.3 Document classification with word embeddings

Moving on, it came time to build a model using these embeddings as inputs. A precise formulation of the task at hand can be as follows:

Given a list of words in a document, **predict the sentiment** of the document.

The documents are classified as either positive or negative, which makes this a binary classification task. As it is the binary classification, we expect much higher accuracy than the last assignment (multi-class classification) as there are only two possible outputs. We reused the MLP model and used the best parameters from the last assignment as the baseline for this task. The model was modified slightly to utilize word embeddings, but the architecture remains mostly the same.

The provided code in `play_with_gensim.py` was refactored to the functions `load_embedding` and `load_embedded_model` in `packages/preprocessing.py`.

2.3.1 Baseline model

The values for these initial parameters were chosen from the best performing model for last assignment to use as a baseline.

Hyper-parameters					
Hidden Layers	**Units**	**Epoch**	**Mini-batch**	**Dropout**	`Bias`
1	25	10	32	0.2	0.1
Learning Rate	**Momentum**	**Loss-Function**	**HL-Act. Funct.**	**Out-Act. Funct.**	
0.01	0.9	Cross-Entropy	tanh	relu	

Table 1: Last assignment's best performing model.

2.3.2 Tuning

In the first assignment we made studies to determine the optimal hyperparameters for our classifier. These studies were:

- `Activation function for hidden layer` × `Activation function for output layer` $\in ["sigmoid","tanh","relu","softmax"]$
- `Number of epochs` × `Batch size` $\in [5,10,15,20,30,40,50] \times \in [32,35,40,45,50,500]$
- `Number of hidden layers` × `Number of units per layer` $\in [1,2,3,4] \times \in [5,10,25,50,100,250,500,1000,2000]$
- `Learning rate` × `Momentum` $\in [0.5,0.1,0.05,0.01,0.005,0.001] \times \in [0.9,0.7,0.5,0.3,0.1,0]$

We opted to make the tuning more automized this time, with a dynamic evaluation function `ex5.py`. This function swaps out the default parameter with the one yielding the best result from the studies in `packages/studies.py`. This module consists of multiple different `Study` classes for each of the different parameter pairs this group decided to test. The encapsulation of the classes helped maintain efficient memory usage and made for straightforward and understandable tuning.

A `packages/ex5.py` script executed the studies for each of the tuning steps. It was this file that generated the results presented below. These results are stored in the `output/ex5` directory. For this assignment, the most important metric we looked at was accuracy.

Before running the studies from the last assignment, we first had to find a vocabulary suited for our task. Thus additional tuning was required, as there were many pre-trained vocabularies to choose from, and we had observed spikes in accuracy when choosing the word embedding model arbitrarily. We created two more studies:

- Testing pretrained vocabularies against different loss functions.
- Testing different word embedding types against the pretrained vocabularies.

This was mainly to find an optimal vocabulary to use, as well as asserting that cross entropy was a good loss function to utilize.

The combination that yielded the best result was vocabulary 40, a Word2Vec Continous Skipgram without POS tagging, and cross-entropy. Thus we continued with vocabulary 40 and cross-entropy. Once we had found our preferred word embedding model, we decided on the type of embedding we were going to use.

As we can see in the heatmap, the mean was by the far superior embedding type, and the model was finally producing actual results. After finding the appropriate vocabulary and embedding type, we could tune the model in the same manner as in the previous assignment.

Study	Param. 1	Param. 2	Heatmap
`VocabLossFunct`	`Loss function: cross-entropy`	`Vocabulary: 40`	Apendix 14
`EmbedTypeNewVocab`	`Embedding type: mean`		Apendix 15
`ActFunct`	`Hidden: tanh`	`Output: relu`	Apendix 16
`EpochsBatches`	`Epochs:30`	`Batch size:32`	Apendix 17
`HLU`	`Hidden layers:2`	`Units:5`	Apendix 18
`LrMmt`	`Learning rate: 0.01`	`Momentum: 0.9`	Apendix 19

Table 2: Results from studies of exercise 5

2.3.3 Optimal parameters for MLP model

The best parameters for the MLP model can be seen in the table below which received an accuracy score of 0.694.

Hyper-parameters for MLP model				
Hidden Layers	**Units**	**Epoch**	**Mini-batch**	**Dropout**
1	25	10	32	0.2
Learning Rate	**Momentum**	**Loss-Function**	**HL-Act. Funct.**	
0.01	0.9	Cross-Entropy	tanh	
Embedding type	`Vocabulary`	**Bias**	**Out-Act. Funct.**	
mean	40	0.1	relu	

Table 3: The best hyperparameters found for the MLP model.

These were used as the default values for the upcoming steps in our analysis.

2.4 Document classification using recurrent neural networks

The next part of this project was to implement a RNN as a classifier. We were to continue the task from part 2.3, where the goal was to classify the sentiment polarity of an input sentence. We chose to only focus on fine-tuning the RNN and feeding it forward to a single linear output layer for simplicity. This way, we can directly compare the performance of recurrent networks with simple, feed-forward networks, as implemented above, getting a precise measurement of the increase of performance these architectures bring.

Recurrent neural networks work well on sentiment analysis because they maintain the inherent, sequential information of input sentences. The order in which words appear in a sentence can change the sentence's overall polarity. For example, take the two sentences:

In contrast to the **poor** title, the movie was overall **interesting**.

In contrast to the **interesting** title, the movie was overall **poor**.

The two sentences carry opposite polarities, yet a bag-of-words model would classify them exactly the same. Through element-wise updating of states, RNN's consider each word concerning all of the preceding words. This allows the model to capture some of the sentence's structural information, making it more likely to predict the correct sentiment of the sentence.

Specific RNN flavors (LSTM and GRU) incorporate *gated architectures*. Simply put, these learn to separate between the input features that the model needs to focus on from those that are best left untouched. The gates then filter and feed-forward only the features that need tuning, sending the others directly to the output, thus preventing ambiguous computation and over-fitting.

To get a feel of both gated and non-gated RNN's, we were asked to experiment with PyTorch's three RNN types [5], and report the results on the variations of performance.

2.4.1 Building the model

The underlying structure of the `MLPModel` was reused, but with some slight alterations. The `self.model` attribute was set to an instance of a PyTorch RNN, dependent upon the hyperparameter `rnn_type:str="rnn"` of the `RNNModel` class. The three possible alternatives here were: `"rnn"`, `"lstm`, or `"gru"`.

Another part of the model that changed from that used in part 2.3 was the output layer. For the RNN architectures, a `self._linear` attribute was used, instead of the `self.out` from above. This allowed for the model to predict the two classes independently of each other, which translates to higher model flexibility. Additionally, limiting the complexity of the output layer allowed for more direct observations on the performance of RNNs.

The linear layer's input size depended on the size of the output the RNN layer would be serving. In particular, if the RNN layer was given the parameter `bidirectional=True`, the output size would double, moreover the linear layer's input size would need to be doubled. Bidirectional recurrent network outputs are two times larger than regular recurrent networks because they return updated state values from reading the input both forward and backward.

It was still preferable to convert the input text to word embeddings since we saw an increase in performance from the bag of words model in classification with word embeddings. This meant that again, the model needed a `torch.nn.Embedding` attribute to convert input data into embedding representations. This was kept open as a tunable parameter since the optimal embedding vocabulary for a simple feed-forward network is not necessarily the optimal vocabulary for an RNN. The optimal embedder is something that needs to be found for each architecture.

One last key element of our implementation of `RNNModel` was the incorporation of a learning rate scheduler. While there are many different solutions to adaptive learning rates for neural architectures, we chose to utilize the `torch.optim.lr_scheduler.ReduceLROnPlateau` variant. This tool's idea is to adjust the learning rate every time learning seems to stagnate, making sure the model is always learning as much as possible without overfitting. We felt this was necessary because we noticed the loss would often decrease for a few epochs and then jump back up. This told us the optimizer was overstepping, which meant the learning rate was too high. The scheduler fixed this problem. A vital drawback here is that the scheduler only works in one direction and cannot self-correct if an adjustment occurs prematurely.

2.4.2 Tuning studies

Similar to those developed for the `MLPModel`, a set of studies were created to test different variations of architectures of recurrent neural networks found in `rnn_studies.py`. While some of the studies were able to be reused as-is, quite a few new studies needed to be written. The new study implementations here included:

- `RnnTypeBiDirectional`

[5] Recurrent Layers in PyTorch (1.7.1)

- `VocabRnnType`
- `PoolTypeRnnType`
- `BestEpochRnnType`
- `FactorPatience`

We tested one more parameter but did not develop a study for the freezing/unfreezing of the model's embeddings. Analysis of this parameter was done using the bidirectional study, manually changing the value for this parameter. The freezing of embeddings means that they will *not* be updated according to how much the model learned for each epoch. This means they will not be fine-tuned for the particular vocabulary being worked on, thus simplifying the model's complexity. It can often be beneficial to fine-tune a model's embeddings if the task at hand includes many unique words particular to the task's respective domain, which seemed to not happen in this assignment. Word embedding models train to span over as general of a vocabulary as possible to prevent the need for future fine-tuning, as computation time should always be minimized.

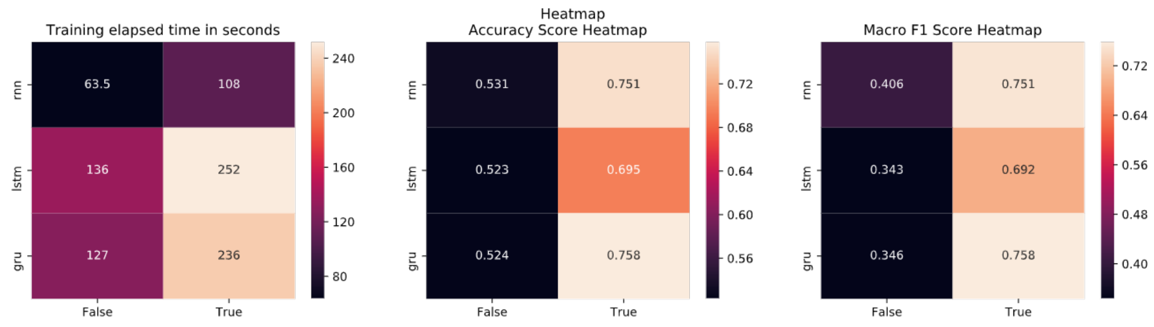


Figure 2: Freeze = True

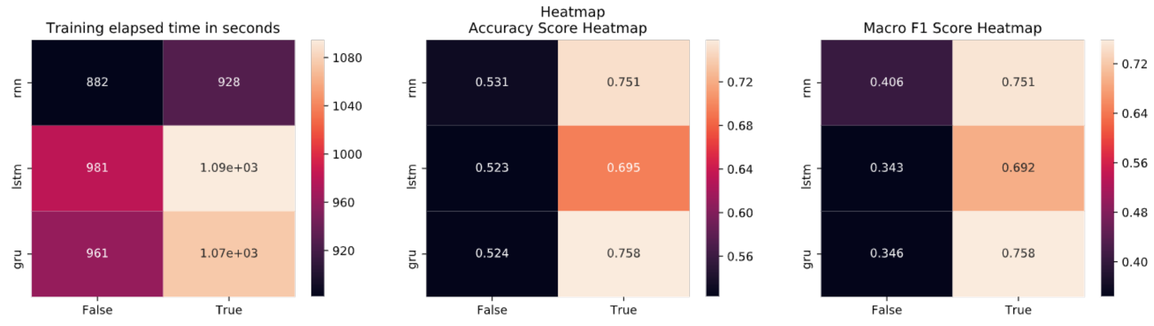


Figure 3: Freeze = False

As can be seen when comparing figures 2 and 3, the only major difference between frozen and non-frozen embeddings was the computation time. Our model could not truly fine-tune the embeddings (LGT did too good a job when training them). For the rest of the experiments, we kept `freeze=True` to lower computation times.

RNN type versus Bidirectional Some of the results from the first study have already been shown. The `True`/`False` axes in figures 2 and 3 represent the `bidirectional` parameter. As can be seen, only reliable results were found when bidirectional was activated. This tells us that a lot can be learned from these particular data about the sentence structure by feeding it to the model both forward and backward. It is important to note here that this study only intended on finding the optimal value for bidirectional. We wanted to test the different RNN types with some of the other variables as well, so no optimal RNN was derived from these analyses.

Embedding vocabulary versus RNN type Next, we wanted to find which embedding vocabulary worked best for each of the three RNN types. We assume that these optimal vocabularies would be different from those found for the MLP model but needed empirical evidence to support this claim.

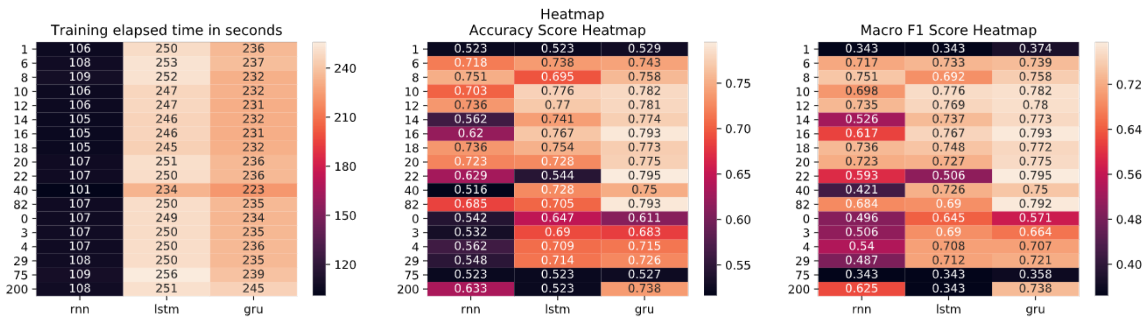


Figure 4: Embedding vocab versus RNN type

Figure 4 shows the results of this study, although the individual vocabularies are only specified by their ID. The 3 top performing vocabularies were as followed:

Rank	1st	2nd	3rd
Algorithm	fastText Skipgram	fastText Skipgram	Global Vectors
ID	22	16	82
Corpus	Wikipedia'17 + Gigaword 5th	Gigaword 5th	English Common Crawl
Vocab. Size	291392	292967	2000000
Accuracy	79.5%	79.3%	79.3%
F-Score 1	79.5%	79.3%	79.2%

An interesting point to note here is that the top 2 embedding models were built using the fastText skipgram algorithm. The Global Vectors algorithm also achieved scores close to that of fastText, across the board, barely missing out on that top spot. Comparatively, the vocabulary size difference between these two embedding algorithms highlights the efficiency of the former. This makes sense since the latter is one of the more rudimentary embedding techniques in the field, built on count matrices of individual words and how often they co-occur together. FastText uses a more robust approach, focusing on n-grams of all shapes, not just singular words. This opens the possibility of representing higher-level concepts and abstractions than previously possible, which is intuitively quite important in language analysis since language is a higher-level abstraction of the human experience.

Another thing to note from this study is the absence of the POS vocabularies as the top performers. All of the English embedders present in the embedding repository [4], including all of those which incorporated part-of-speech tags, were tested here. However, only a few of the POS tagged embedders gave results indicating that the model *actually* learned something. This could have been because the POS tagging technique we used might not have been 100% similar to that which trained those models. Regardless, the models that returned the poorest scores were excluded from the heatmaps for visibility, but their results can still be found in the `outputs/` directory.

Finally, since the difference between performances of 22 and 16 were quite small, we decided to test both of them in some of the upcoming studies to see if any variations in performance would reveal themselves in the later optimization stages.

Pooling strategy versus RNN types The next natural parameter to study was the pooling strategy applied after the RNN. Simply put, this step converted the 3-dimensional output from the

RNN to a 2-dimensional (and linearly interpretable) tensor through some pooling technique. The three techniques we limited our analysis to were slicing the first layer of weights (states) from the RNN output, the last layer, and a concatenation of the two. We also looked slightly into max-pooling (finding the weights with the highest values for each node position, through all the layers of the RNN) and average pooling (average values of each node position through all the layers), but the results seemed too unreliable to include in this report.

We needed to look at both the front slice and the back slice of the output because of bidirectional being set to true (as stated above). Technically, after the model is finished learning, the last spot that would have been updated is that at the first index of the network, since information flowed through the model twice, once from front-to-back and again back-to-front. So, intuitively, this layer provides the most condensed version of the information learned by the model.

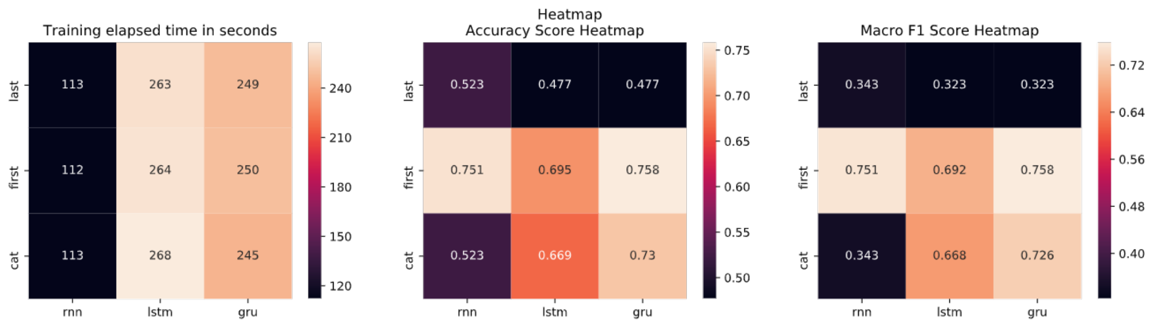


Figure 5: Pooling strategy versus RNN type

The evidence in the heatmaps shown in Figure 5 prove the point made in the previous paragraph. The scores for the models using the `last` pooling strategy gave very poor results (basically random guessing). The technique that read only the `first` slice scored the best, with the concatenation of the two in a close second place (surprisingly enough).

RNN type and best epoch One study that wouldn't look so pretty in a heat map was the one used to determine the actual best RNN type. Using the optimal hyperparameters found so far, this study also looked at the number of epochs it took each architecture type to reach it's highest performance. Noise could play a very large factor here, so heatmaps were disregarded and the results are therefore presented in Table 4:

type	Elman	LSTM	GRU
accuracy	72.5%	80.7%	80.2%
epochs	34	86	58

Table 4: RNN type study

This study was quite interesting, especially with the learning rate scheduler enabled because every bounce up of the loss, indicating over-fitting, resulted in the learning rate being lowed by the scheduler. With a very shallow learning rate, near 0, the model's learning ability does not change anymore, staying in the highest accuracy without over-fitting.

For all three architectures, the performance pretty much reached the converging range after around 30 epochs. However the best performances weren't actually reached until much later (two to three times as many epochs later). This insight was applied to the later studies, testing only on 30 epochs for lighter computing resources, with the intentions to train the final model on many more.

We're also confident the model wasn't over-fitting (at least not by a substantial amount) because of the scheduler. In fact, when studying the output for this study, it was observed that every time there was an increase in loss for a few steps, the model seemed to correct itself, and start learning again.

This output was stored in outputs/ex6/slurm_BestEpochRnnType_vocab22.out, and gives a nice insight in the optimal ranges for epochs for these architectures on these data.

In this study, the chosen RNN type was GRU because of its efficiency, relatively high score with fewer epochs. Indeed, the score difference between LSTM and GRU was not significant; therefore, the lower number of epochs was considered an efficiency parameter.

Hidden layer versus units The exact same hidden layers versus units study was used from previous experiments. The only maps that showed a true trend were the time maps.

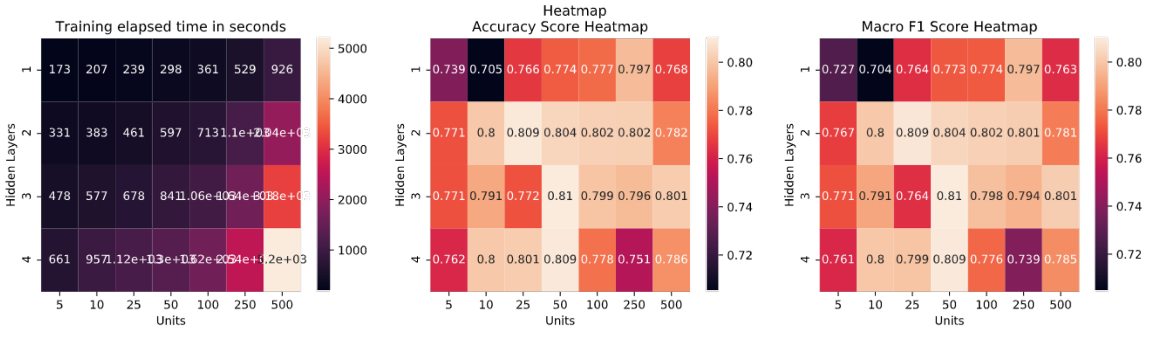


Figure 6: Hidden layers vs. Units (vocab. 16)

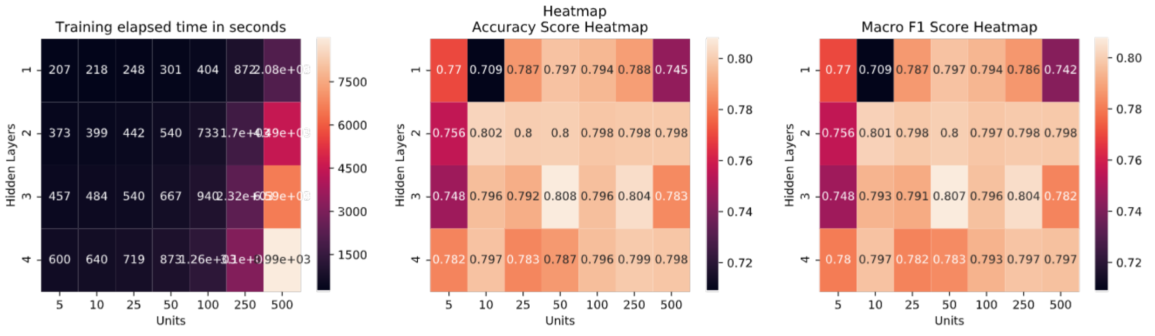


Figure 7: Hidden layers vs. Units (vocab. 22)

However, it is interesting to point out that both embedding 16 and 22 gave the best score with 3 hidden layers and 50 units. Figure 6 shows a very close second place at 2 hidden layers and only 25 units. Again, to spare excess computations, these values were deemed good enough for future experiments (with intentions of using the 3 hidden and 50 units on the final model).

Learning rate versus momentum The next study was also one that has been used previously, namely comparing learning rates against momentum. Recall the learning rate scheduler implemented here, that only *reduces* the learning rate on stagnation. This tells us that larger start values for learning rate are probably better than small, since they give the scheduler more control and ability to work properly.

Momentum is obviously chosen the test with learning rate, since the two parameters are highly correlated. Lower learning rates require higher momentum, ensuring the model learns enough each epoch to produce a noticeable change in performance.

Again, both vocabulary 16 and 22 were tested simultaneously.

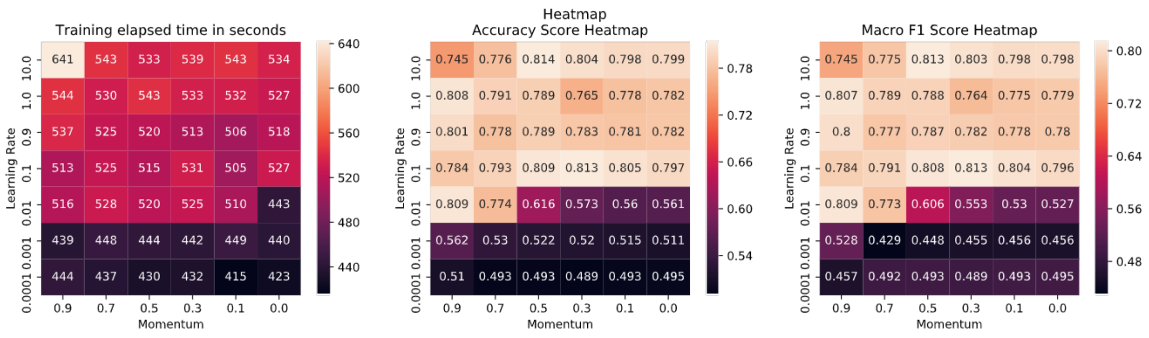


Figure 8: Learning rate vs. Momentum (vocab. 16)

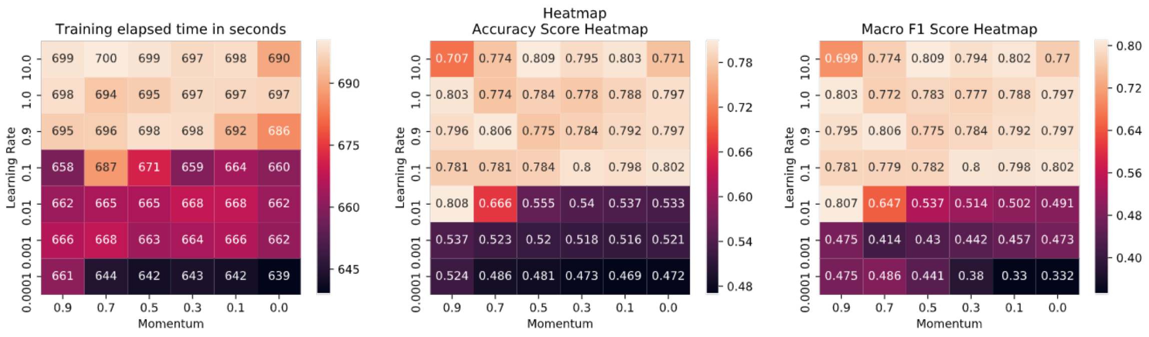


Figure 9: Learning rate vs. Momentum (vocab. 22)

Just to see what would happen, we tested a learning rate of 10, even though this does not quite make theoretical sense. To our surprise, this actually gave some of the better results. Most likely, this elevated performance came from the ultimate flexibility we are giving the scheduler here. The current factor being tested was 0.1, so after the scheduler's first reduction, the learning rate would be back down to "normal" values.

We decided to shoot safe and stick to the best performing learning rate between 0 and 1, which was 0.1. This occurred with a momentum of 0.3, which is much lower than found for previous neural architectures' implementations. This is likely due to the inherent complexity of RNNs, which do not rely as much on previous performance improvement than the vanilla MLP model from above.

Factor versus patience Two completely new parameters that needed to be tuned here were `factor` and `patience` of the learning rate scheduler. Factor refers to how much the scheduler should reduce the learning rate when called. Patience is how many epochs are allowed after stagnation is first noticed, before calling for a reduction.

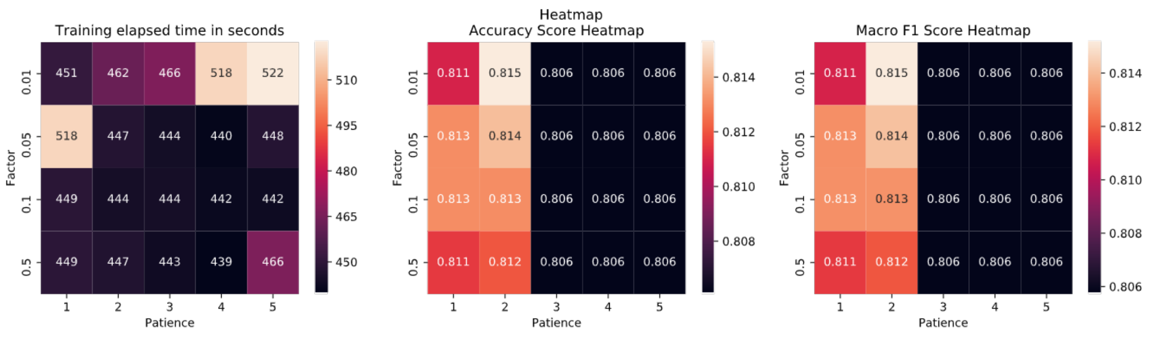


Figure 10: Factor vs. Patience (vocab. 16)

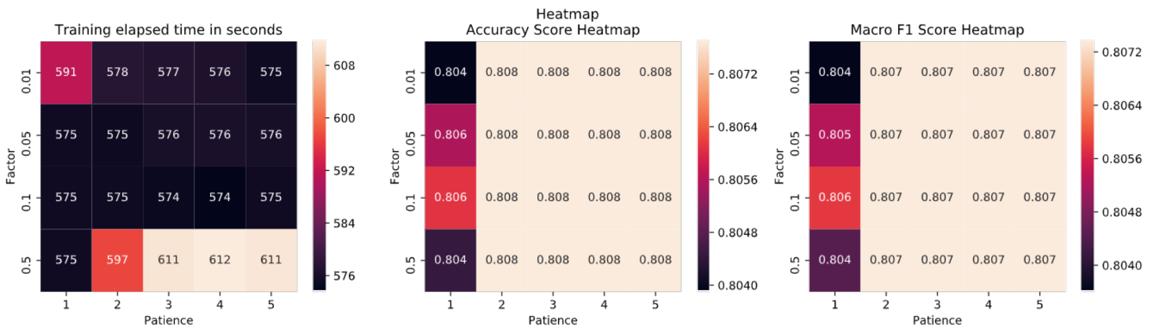


Figure 11: Factor vs. Pateience (vocab. 22)

Figures 10 and 11 show very little variation in performance when tuning of these values. However, a factor of 0.01 and patience of 2 is the top scoring configuration, when considering both embedders.

Batch size versus number of epochs The final study run in this analysis tested different batch sizes with number of epochs (repeated to reuse previous study). We wanted to make sure the batch size used for these tests were defendable.

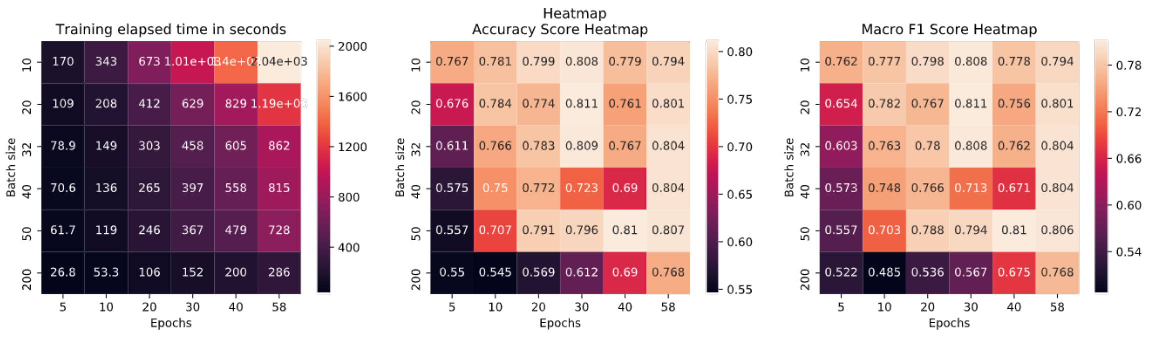


Figure 12: Batch size vs. Epochs (Vocab. 16)

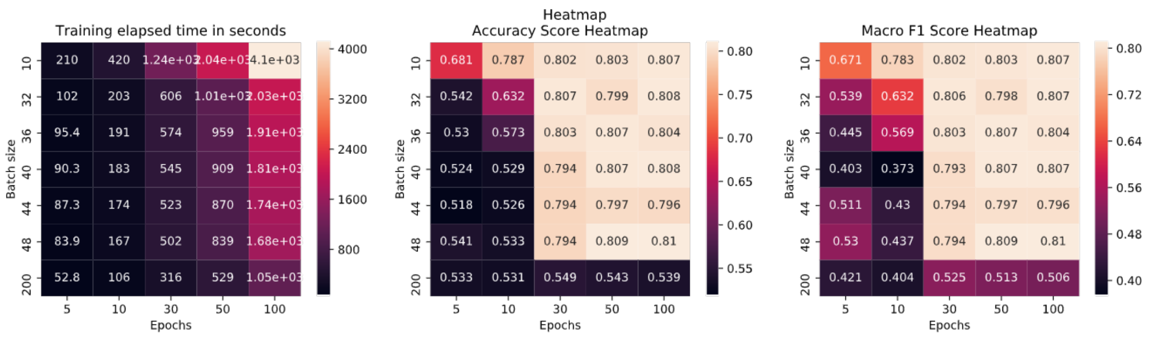


Figure 13: Batch size vs. Epochs (Vocab. 22)

A key insight to note here is the *poor* performance with too few epochs and too large of batch sizes. Other than that, it seemed we were already testing pretty close to the optimal parameters, with both around 30.

2.5 Final hyperparameters

As a result of our experimenting, our setup gave the following optimal hyperparameters:

Vocab	**RNN**	**Bidirection/Freeze**	**Pool**	**Batch size**	**Epoch**
22	GRU	True / True	First	20	30
Learning Rate	**Momentum**	**Factor**	**Patience**	**Hidden Layers**	**Units**
0.01	0.9	0.01	2	3	50

Table 5: Optimized model parameters

3 Conclusion

This project looked at word embeddings, through a web-app, through local experimenting, and finally as input features for a more complex model, incorporating transfer-learning. We also detailed the steps to build and tune a simplistic RNN model that could read arbitrary lengths of sequences and produce reliable binary classifications.

One of the main hurdles that needed to be overcome when representing sentences as tensors of word embedding was standardizing the models' input feature size. As shown, there are various ways to achieve this standardization, whether it be taking the mean, the sum, or focusing on a specific slice of the input.

The puzzle of building a recurrent architecture requires finding the right pieces that work well together. The parameter that had the biggest impact on our scores was `bidirectional`, although the use of gated architectures also proved to be quite helpful compared to the vanilla Elman RNN.

At the beginning of this project, it was assumed that RNNs would achieve better scores than MLP, since they consider sequential information. We observed a 12.2 percentage-point increase in performance between the two models through our experimenting, confirming our hypothesis. The best score for the recurrent setup was 81.5%. For binary classification, this is well beyond randomly guessing, especially considering the data set was more or less balanced (47% negative and 52% positive). To prove the model's robustness, different random seeds were set, causing different partitions of training and validation data. The results are presented in Table 6:

Score type	**Run 1**	**Run 2**	**Run 3**
F1-score	79.60%	78.71%	81.72%
Accuracy	80.08%	79.63%	80.28%

Table 6: Scores for three RNN builds on different splits

Table 6 shows the stability of the optimized parameters for our final model, using 3 different train-validation splits, and measuring the performance.

We feel that accuracies and F1-scores around 80% were sufficient for this task at hand, and expect the evaluation on the hold-out set should return similar results.

4 Bibliography

[1] T. Hastie, R. Tibshirani, J. Friedman. *The Elements of Statistical Learning data mining, inference, and prediction*; 2nd edition; Springer; New York, USA. 2017.

[2] Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning*; 2nd edition; Packt; Birmingham, UK. 2017.

[3] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*; O'Reilly Media; Sebastopol, CA. 2017.

[4] Jeffery Pennington, Richard Socher, Christopher D. Manning. *GloVe: Global Vectors for Word Representation*; Computer Science Department, Stanford University; Stanford, CA 2014

[5] Murhaf Fares, Andrei Kutuzov, Stephan Oepen, Erik Velldal. *NLPL word embedding repository*; Language Technology Group, University of Oslo; Oslo, Norway 2017

5 Appendix

5.1 Exercise 5

5.1.1 Vocab versus Loss Function Study

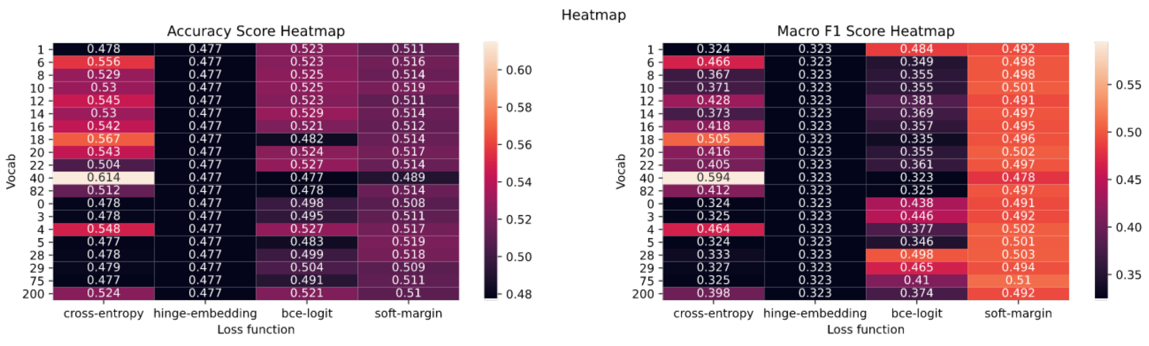


Figure 14: Vocab vs Loss Function

5.1.2 Embedding Type Study

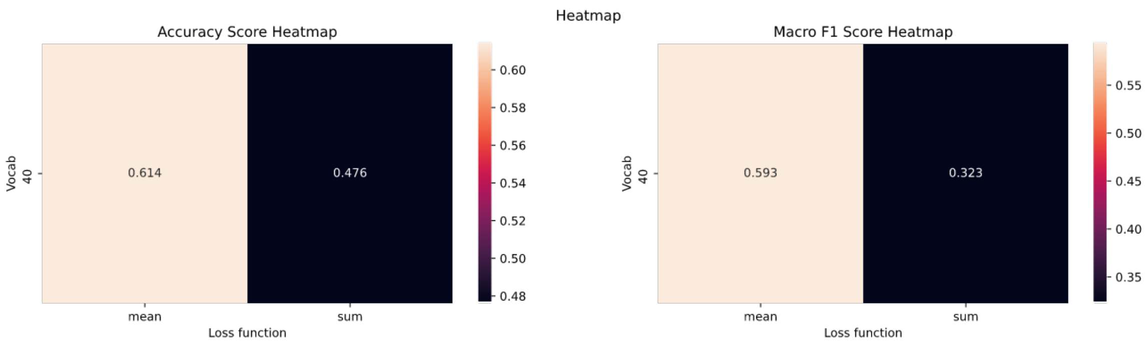


Figure 15: Embedding Pool type

5.1.3 Hidden versus Output Activation Functions Study

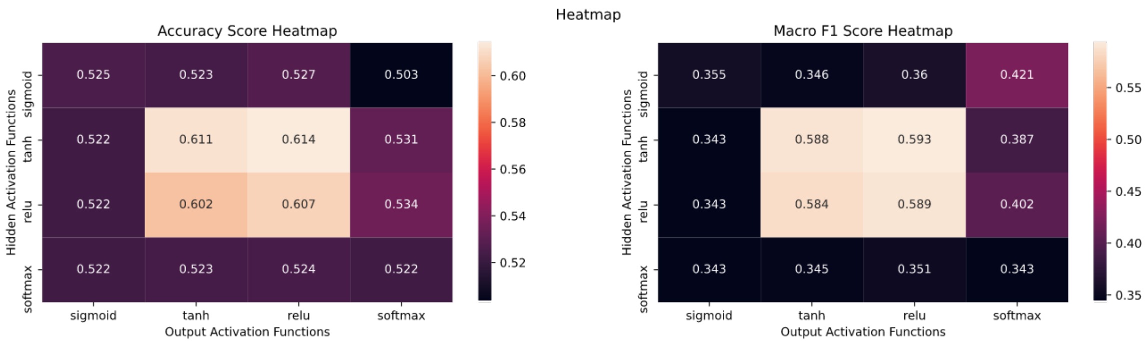


Figure 16: Activation function study

5.1.4 Epochs versus Batch Size Study

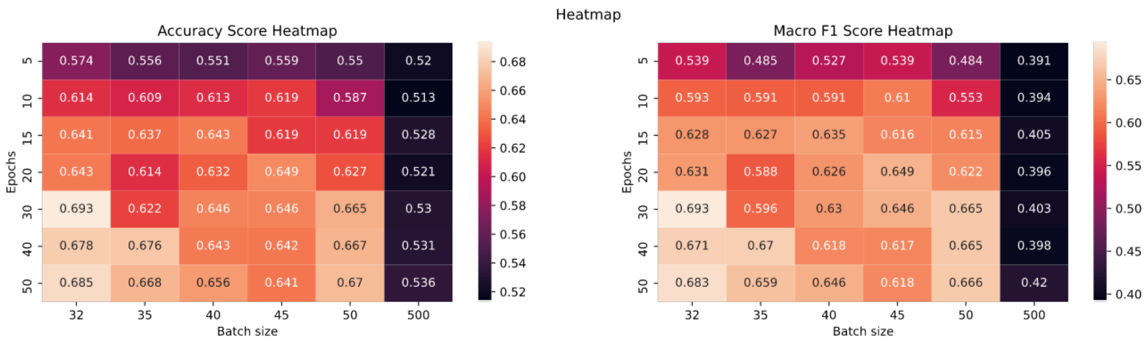


Figure 17: Epochs vs batch size

5.1.5 Hidden Layers versus Units Study

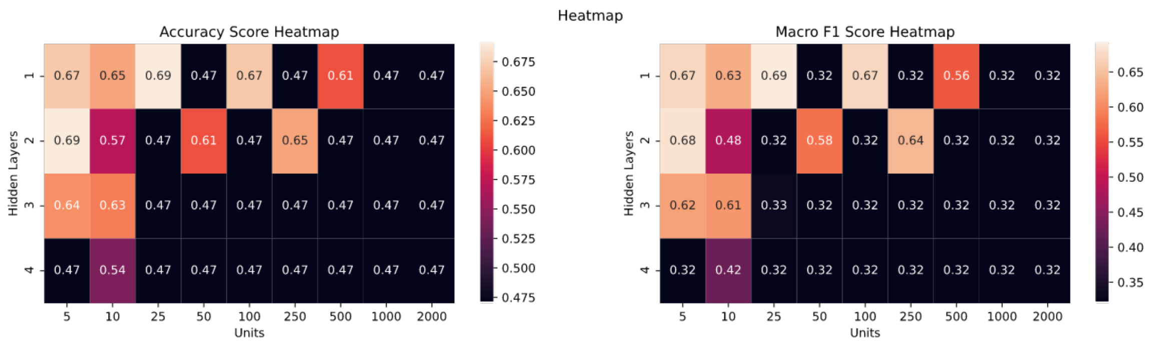


Figure 18: Hidden layers vs units per layer

5.1.6 Learning Rate versus Momentum Study

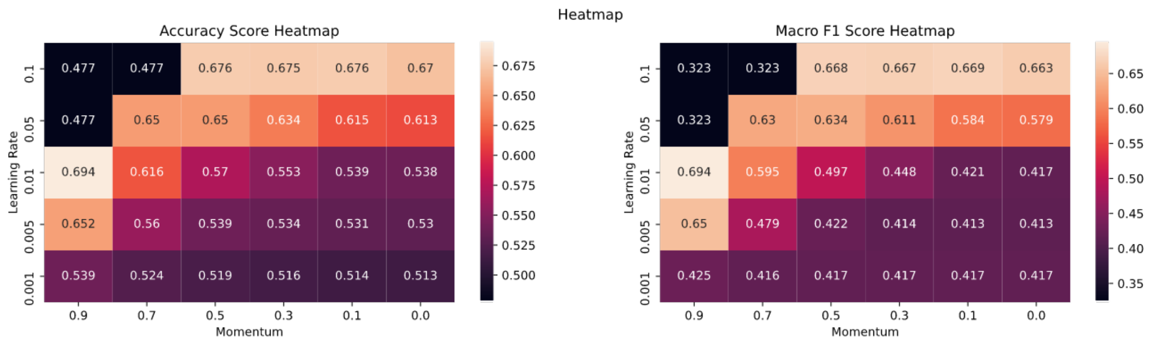


Figure 19: Learning rate vs momentum