Τεχνητή Νοημοσύνη ΙΙ - Εργασία 2

Μυσταχίδης Ιωάννης 1115201600113

1.

We have

$$\hat{\mathbf{y}} = Softmax(\boldsymbol{\vartheta})$$

and

$$J = CE(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{c} y_i \cdot log(\hat{y}_i)$$

where c denotes the number of different classes and the subscript i denotes the i-th element of the vector. So,

$$\frac{\partial J}{\partial \theta_j} = -\frac{\partial}{\partial \theta_j} \sum_{i=1}^{c} y_i \cdot log(\hat{y}_i) = -\sum_{i=1}^{c} y_i \cdot \frac{\partial}{\partial \theta_j} log(\hat{y}_i) = -\sum_{i=1}^{c} \frac{y_i}{\hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial \theta_j}$$

now, by plugging in the derivatives, we get

$$\frac{\partial J}{\partial \theta_j} = -\sum_{i=1}^{c} \frac{y_i}{\hat{y}_i} \cdot \hat{y}_i \cdot (1\{i = j\} - \hat{y}_j) = -\sum_{i=1}^{c} y_i \cdot (1\{i = j\} - \hat{y}_j)$$

and by expanding the product in the last term we get

$$\frac{\partial J}{\partial \theta_j} = \sum_{i=1}^{c} y_i \cdot \hat{y}_j - \sum_{i=1}^{c} y_i \cdot 1\{i = j\}$$

The indicator function $1\{\cdot\}$ takes on a value of 1 for $i = j$ and 0 everywhere else:

$$\frac{\partial J}{\partial \theta_j} = \sum_{i=1}^{c} y_i \cdot \hat{y}_j - y_j$$

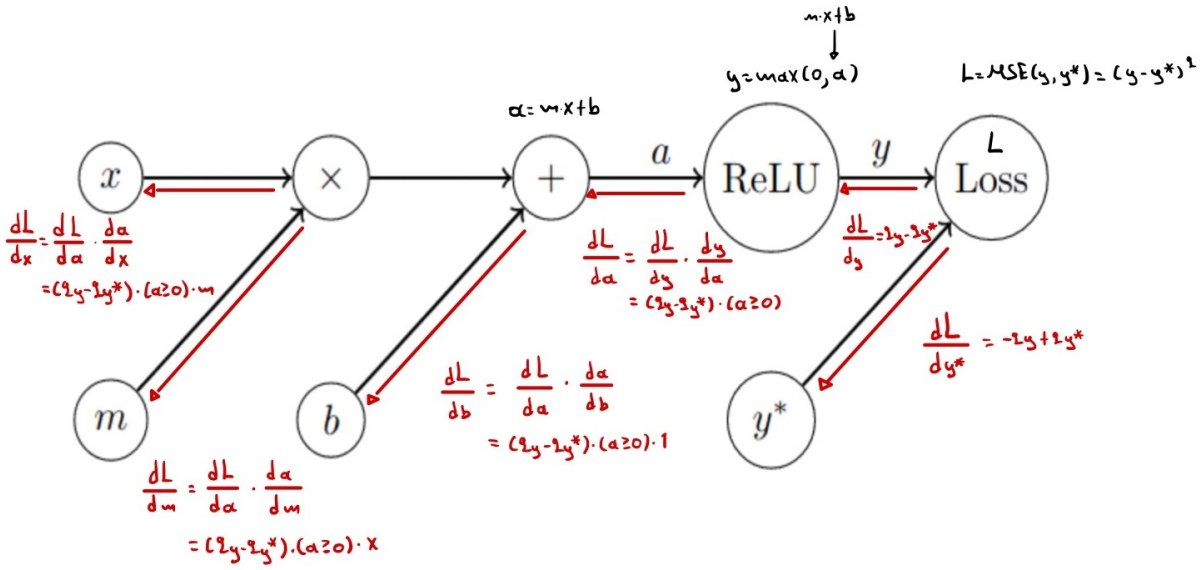
Next, we pull $\hat{y}$ out of the sum, since it does not depend on index i:

$$\frac{\partial J}{\partial \theta_j} = \hat{y}_j \cdot \sum_{i=1}^{c} y_i - y_j = \hat{y}_j - y_j$$

because the sum of $\mathbf{y}$ (with respect to the classes) equals to 1 when one-hot encoded. So, in concise vector notation we have:

$$\frac{\partial J}{\partial \theta} = \hat{\mathbf{y}} - \mathbf{y}$$

2.



where we denote the derivative of ReLU as the truth value of $(a \geq 0)$ meaning that the derivative is 1 if $(a \geq 0)$ and 0 otherwise.

3.

How to Run:

- Firstly, you have to upload the datasets on colab (the vaccine datasets). We read the training and validation data in the first cell. The paths of the datasets are simply the names of the datasets themselves, but you can change according to the path on which you upload the files yourself. You don't have to upload any word-embeddings because we download and unzip the ones we need inside the colab.
- You can either:
  - Use 'Run all'

    Which would take a little more than 1 hour to finish, because of all the plotting (not advised).

- Or **omit** some (code) cells and run the others:

  You can omit the cells that do the plotting for each hyperparameter in the 'Model Selection and Training'. That is:

  * Cell 15: which plots the metric values per epoch for each learning rate
  * Cell 16: which plots the metric values per epoch for each number of hidden layers
  * Cell 17: which plots the metric values per epoch for each number of units for each hidden layer
  * Cell 18: which plots the metric values per epoch for each activation function
  * Cell 19: which plots the metric values per epoch for each optimizer

  These cells are merely for visualization purposes only, just to visualize how each hyperparameter affects our model's bias/variance and justify our choices for the various hyperparameters (although countless more experiments did take place that are not shown). Omitting these cells will significantly reduce the running time.
  You can't omit any of the other cells because they are important for the flow of the notebook.

- Finally, in order to run the model on the test set you only have to change the path of the file in the section 'Final Model Evaluation on the Test Set' (the path is currently set to the validation set again and there are some prints and plots). Then you can run the cells in that section only to see how our model performs on the test set.

Experiments:

We conducted three different experiments. We tried Softmax Regression with Word-Embeddings, Neural Networks without Word-Embeddings and Neural Networks with Word-Embeddings. Obviously, in each of the aforementioned experiments we experimented with various hyperparameters and conducted countless (sub-)experiments. We don't provide a jupyter notebook for the first two experiments, only for the third one. This is because our python code is general enough to be used by each experiments according to its specific needs. Also, the (sub-)experiments conducted (especially regarding the second and the third experiment that use neural networks) are very similar, with only different conclusions coming out of them. So we only documented the first two experiments and provided the jupyter notebook for the third one.

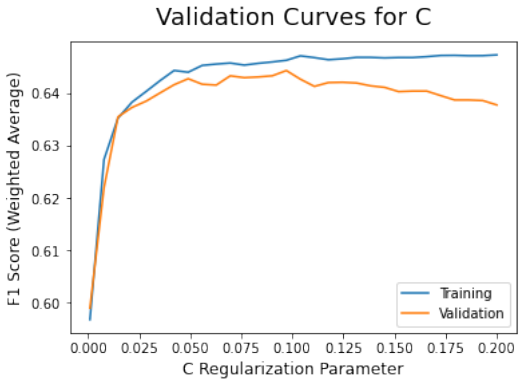
- **Recap**: First Exercise Best Model
  - Best F1_Score (Average Weighted) on Validation Set: 0.69764
  - Best Training Algorithm Hyperparameters:

    - C: 0.13
    - Penalty: l1
  - Best Preparation Steps Hyperparameters:
    - Remove URLs
    - Remove Hashtags
    - Remove Tags
    - Remove Emojis
    - Convert to Lowercase
    - Remove Punctuation
    - Don't Remove Stopwords
    - Handle Inflected Forms using Lemmatization
    - Vectorize using plain Bag of Words
    - Vectorizer ngram_range: (1, 2)
    - Vectorizer min_df: 5

- **Experiment (i)**: Softmax Regression **with** Word-Embeddings

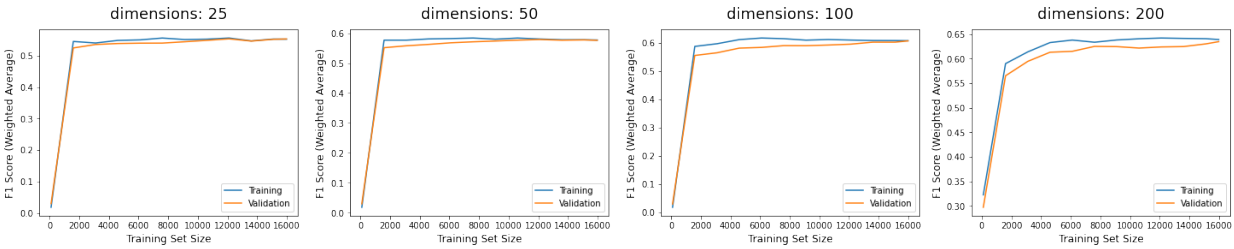
  Here we wanted to see how the use of word-embeddings as the vectorization technique for the tweets would affect the training process of the Softmax Regressor we used in the first assignment.
  We used the Twitter word-embeddings and tried out all 4 options (files) for the vector dimensions (i.e. 25,50,100,200).
  Using these files, we map every word in a tweet to a vector of 25,50,100,200 dimensions (depending on the choice) and then the whole tweet is averaged along these dimensions. So after the vectorization of the whole dataset (either training or validation) we end up with a numpy array with rows as many as the dataset's rows (i.e. number of tweets in the dataset) and columns as many as the dimensions (i.e. which one of the 4 files) chosen.
  We experimented with the hyperparamets of both the training algorithm (Softmax Regression) and the preparation steps as well.

– Validation Curves for C:



We used the default preparation steps and the 200 dimensions word-embeddings for the above plot. The plot for the other dimensions isn't that much different, the optimal value for the C hyperparameter of softmax regression always lies somewhere between 0.02 and 0.1 (similar to the first exercise). Lower than 0.02 leads to underfitting and higher than 0.1 leads to overfitting.

– Learning Curves per the different Dimensions:



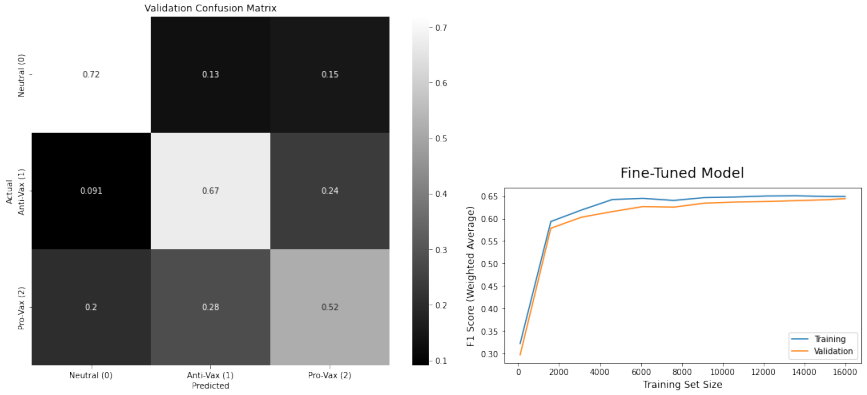
We can see that the model generalizes well no matter the dimensions we choose, but it appears that there's a little lower bias when using the 200 dimensions, which leads to better f1_scores.

– After some fine-tuning we ended up with:

  * Best Training Algorithm Hyperparameters:
    · C: 0.1
    · Penalty: l1

- Class_Weight: Balanced

* Best Preparation Steps Hyperparameters:
  - Vectorize using Pre-trained Word-Embeddings
    (This was what this experiment was all about)
  - Use the 200 Dimensions File for the Word-Embeddings
    (This choice was explained above)
  - **Don't** Remove Hashtags
    (The pre-trained word-embeddings contain hashtags like #love and therefore we don't necessarily have to remove them from our tweets. This way, useful hashtags that possibly carry some kind of sentiment like #love or #notgood are included in the final vector of the tweet and useless ones are discared because they don't exist in the word-embeddings' vocabulary)
  - **Don't** Handle Inflected Forms (Stemming was way worse than Lemmatization because it lead to words that did not exist in the word-embeddings' vocabulary and therefore the vectorized tweets were not very representative. However, the best choice was to not handle the inflected forms at all)
  - All other preparation steps remained the same as in the first exercise.
* Best F1_Score (Average Weighted) on Validation Set: 0.64433



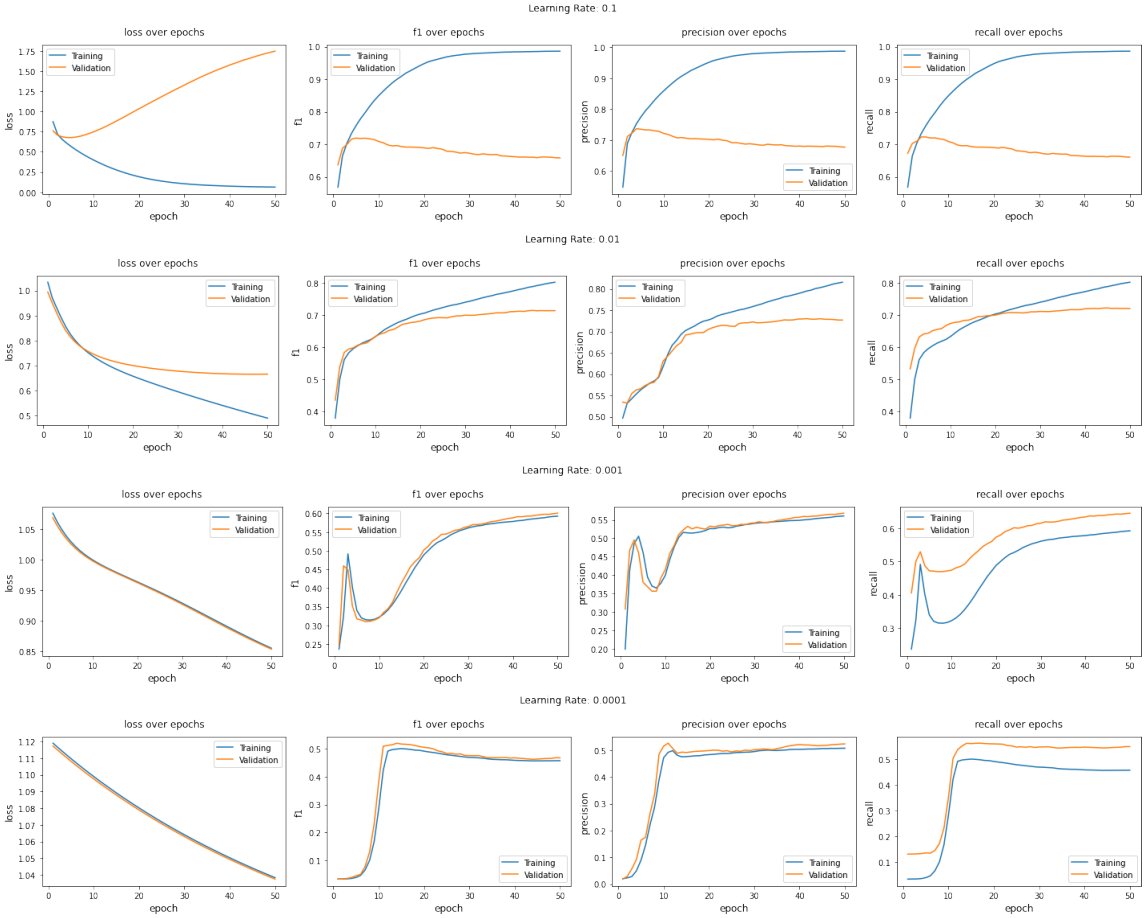
Comments on experiment (i):
As shown above, the final model generalizes pretty well. There is no underfitting/overfitting as the two curves are very close to each other. However, when compared to the first exercise's model, the word-embeddings seem

to have worsen our performance. This is quite expected when we take into consideration the fact that the averaging that takes place over the different dimensions (so that all tweets' final vectors have the same dimensions regardless of their number of words) doesn't make that much sense. Another reason would be that we didn't have the chance to exploit bi-grams, tri-grams etc. using this very elementary approach to word-embeddings. Finally, the corpus that the word-embeddings were trained on wasn't specific to tweets about vaccination, so many words were probably not taken into consideration during the vectorization process because they were not in the word-embeddings' vocabulary (whereas in the first exercise they were included, because the vocabulary depended on our own datasets). Training our own word-embeddings (on possibly larger vaccination tweets datasets) would be a better approach.

- **Experiment (ii)**: Neural Networks **without** Word-Embeddings

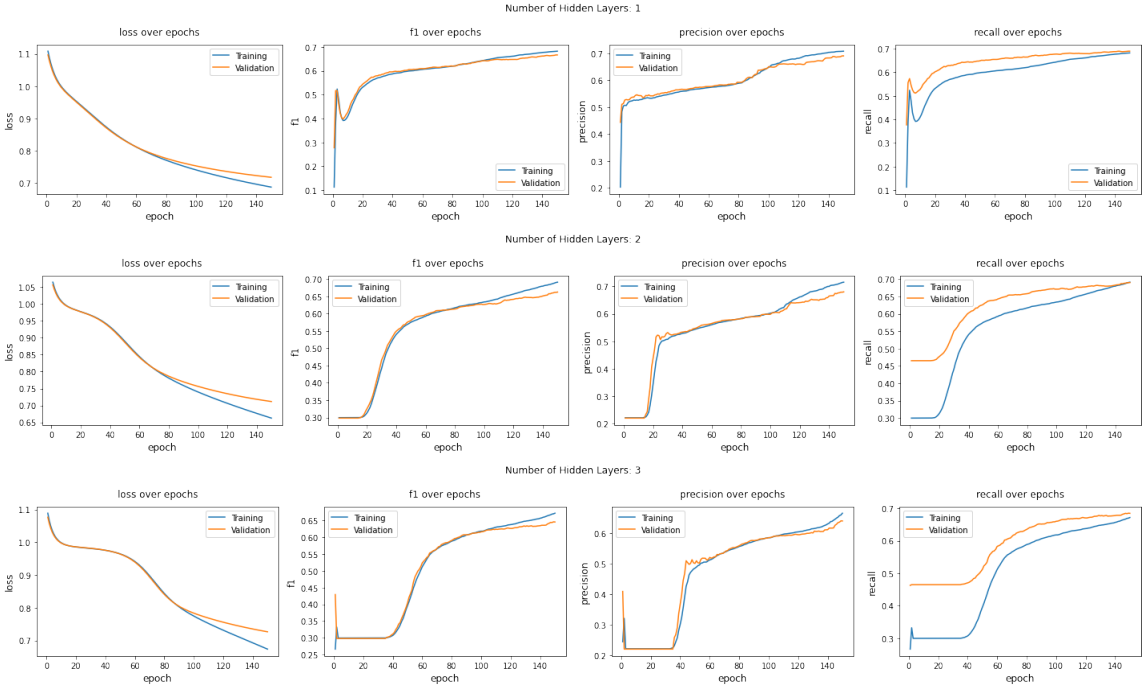
  Here we wanted to see whether the use of neural networks (without the word-embeddings) would lead to a better model than the softmax regressor of the first exercise. Therefore, we kept the best preparation pipeline we found in the first exercise and only experimented with the different hyperparameters of the neural networks (not the preparation steps), like the number of hidden layers, the number of their units etc. and show below how each one affects the bias/variance of our model (some assumptions where made for each of the hyperparameters that where not the variable of the experiment each time, but the overall effect -bias/variance increase/decrease- shown below for each hyperparameter doesn't change 360 degrees given different values for the other hyperparameters):

– Learning Rate:



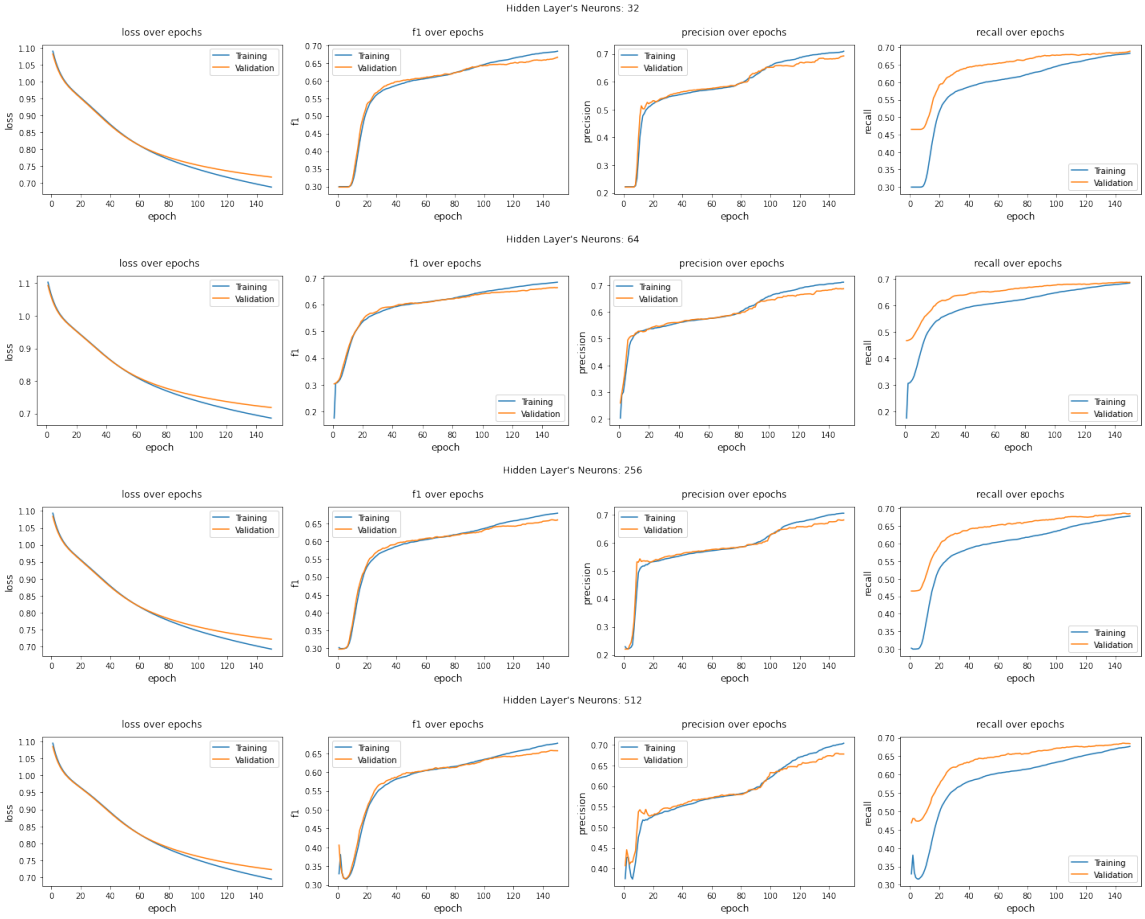
We can see in the above plots that a large learning rate like 0.1 leads to low bias but very high variance, i.e. our model overfits. Decreasing the learning rate, reduces the variance but increases the bias and therefore we get slightly worse f1 scores. Reducing the learning rate too much (0.0001) leads to undefitting, our model learns too slowly and may end up not learning at all.

– Number of Hidden Layers:



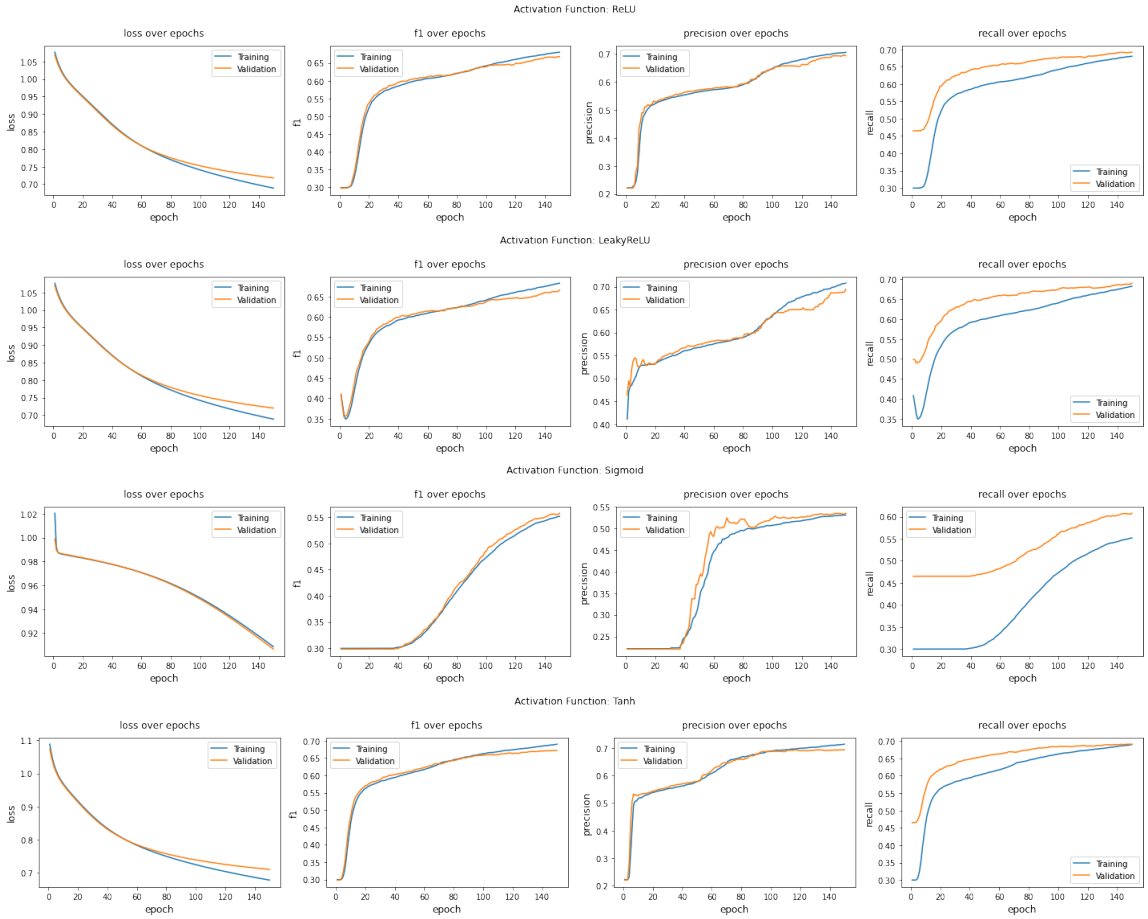
We can see in the above plots that a more complex model, as per the number of hidden layers, reduces the bias of the model but increases the variance. A very complex model, with lots of hidden layers, is prone to overfitting and therefore hurts our generalization.

– Number of Neurons for each Layer:



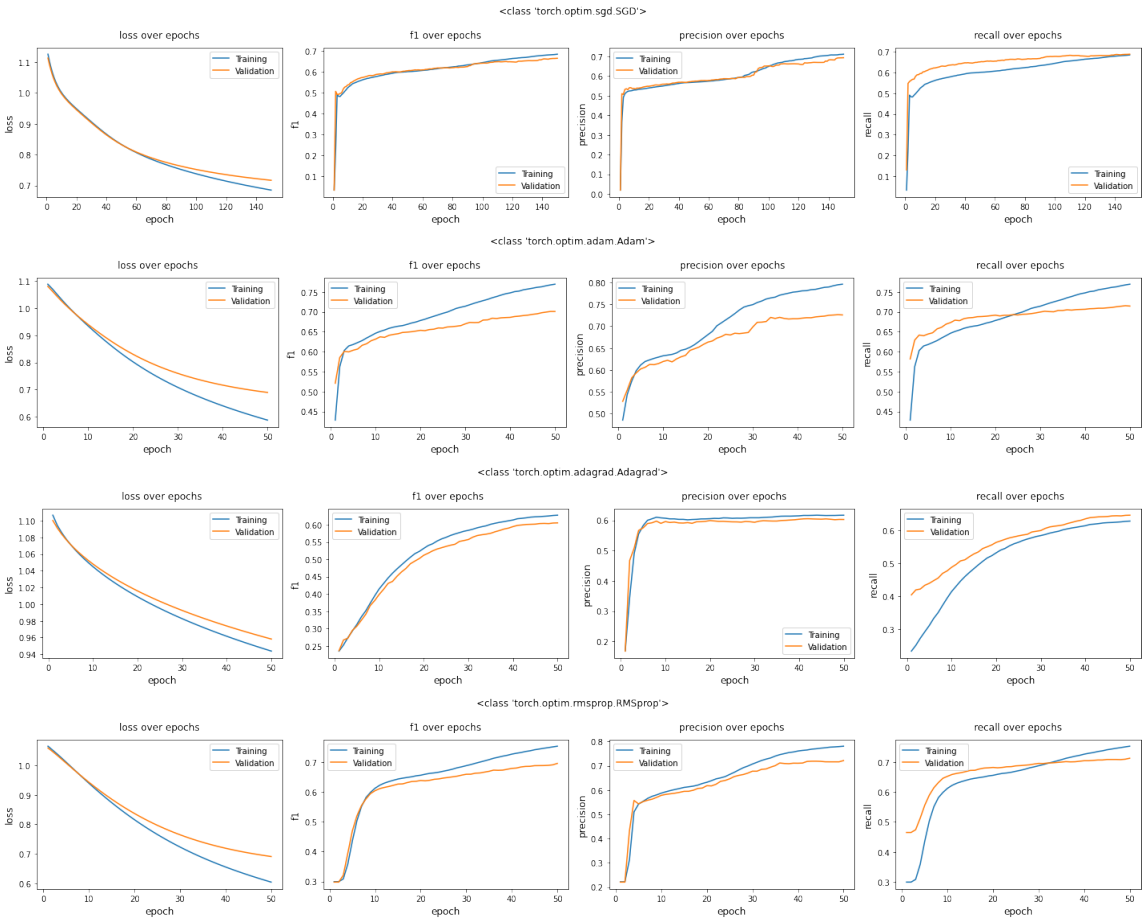
We can see in the above plots that a larger number of neurons for each hidden layer increases the complexity of the model. The overall effect, however, on the model's performance depends greatly on other hyperparameters like the the number of hidden layers. That is, a large number of neurons for a single hidden layer (as shown above) doesn't affect the performance of the model in a very obvious way. However, given three hidden layers, the number of their units would greatly affect the overall performance of our model (a high number of units would probably lead to overfitting given the large number of hidden layers).

– Activation Functions:



We can see in the above plots that, apart from the sigmoid function that doesn't look like a good fit for our case, using different activation functions for the hidden layers doesn't affect our model in an obvious way. Tanh looks promising but overall it looks like simple ReLU performs fine. The best option probably depends on the values of the other hyperparameters, but overall the effect of the activation functions given the current configuration is not that large.

– Optimizer:



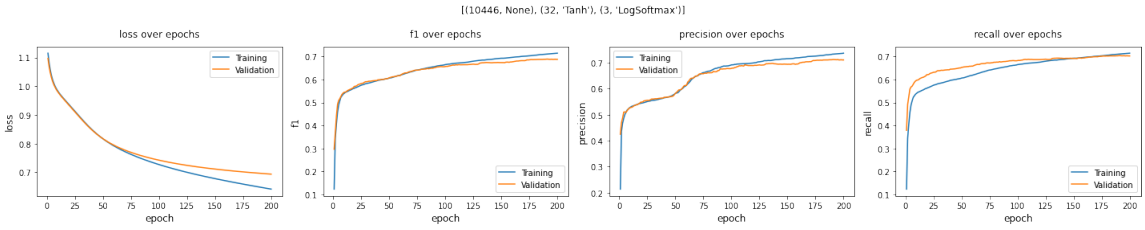
Using the same learning rate (0.001) for all the different optimizers leads to immense overfitting for all except the SGD one. Therefore, we had to adjust the learning rate for each of the optimizers, in such a way that the network would still learn and try to avoid overfitting as much as possible, to create the above plots. Apart from SGD, adagrad was the most promising one, but didn't overcome SGD's performance. The other ones look like they're overfitting given the current configuration, but this is probably because we didn't fine tune their hyperparameters (e.g. weight_decay etc.).

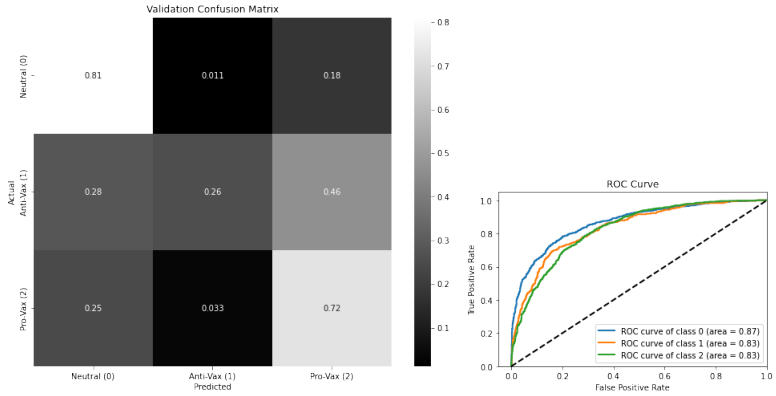
– We also carried out some other experiments regarding the loss function, the epochs, the batch size etc. Obviously, the best choice for each hyperparameter depends on the values of the other hyperparam-

eters (in the above plots we made some assumptions for the other hyperparameters each time, e.g. the hidden layer's neurons where plotted given a single hidden layers and the ReLU activation function). We showed how each hyperparameter generally affects the bias/variance of the model, so for example if we wanted a lower bias we could decrease the learning rate or increase the complexity of our model (i.e. number of hidden layers, neurons per layer), whereas if we wanted to decrease the variance we would do the opposite. The epochs and the batch size always depended on the values for the other hyperparameters. As for the loss function, given that this was a multi-class classification problem, we don't have many options, almost all of the times you apply the softmax activation function at the output layer and use CrossEntropy for the loss. We used LogSoftmax at the output layer and then NLLLoss for the loss. Instead of NLLLoss, we also tried out PoissonNLLLoss and GaussianNLLLoss. Finally, we also tried out KLDivergence with the sigmoid activation function at the output layer instead of softmax. However, overall, plain NLLLoss gave us the best results.

After some fine-tuning we ended up with the following model:

- Learning Rate: 0.001
- Number of Hidden Layers: 1
- Number of Neurons for each Layer: 32
- Activation Functions: Tanh for the hidden layer and LogSoftmax for the output layer
- Loss Function: NLLLoss
- Optimizer: SGD





Comments on experiment (ii):
Given the activation functions, the neural networks are able to identify non-linear relationships between the input and the output. This power, however, comes with the danger of overfitting. In order to avoid overfitting we had to scale down our model's complexity which also affected our bias (i.e. resulted to higher bias than the softmax regression model). Instead of that, we could have tried a more complex model, with more hidden layer's and/or units per hidden layer and regularize it using dropout and/or weight regularization (e.g. weight_decay on the optimizer). This could probably lead to a slightly better model and surpass the performance of our current (not very complex) model, because it would allow to fit our training data better (due to the higher complexity of the model) and also avoid overfitting (due to the regularization techniques). We **don't** believe, however, that the difference would be that large (e.g. reach 90 percent average f1_score) because of the small number of our data and the inherent noise they contain. In conclusion, we managed to match the performance of the first exercise using a very simple model and we would probably surpass this performance had we gone for a slightly more complex model whilst employing better regularization techniques.

- **Experiment (iii)**: Neural Networks **with** Word-Embeddings

  This is the last experiment. Here we wanted to see how the combination of Neural Networks and Word-Embeddings affects our model's training and generalization. When it comes to the preprocessing part, we kept the best preparation pipeline we found in experiment (i) for the word-embeddings so that we have a common framework for the comparison. The effect that each hyperparameter has on our model, when it comes to increasing/decreasing bias/variance, is similar to experiment (ii) in general. However, due to the small number features coming from the

word-embeddings, some of the hyperparameters, obviously, have different optimal values compared to the ones in experiment (ii). Since this is already a very large documentation, we provide the comments and the plots for each of the hyperparameters and the final model inside the notebook.