Artificial Intelligence II

Homework 2

Student: Argiro Lisi
Id: 111520200054

This here is my presentation for Homework2 AI2, I know it's not the standard, usual template I should have followed, however it is my own personal presentation step by step as I followed it myself at every moment. Thanks for watching and accepting my presentation.

Final settings:

1) Data

Removal of stopwords, ~~Lemmatization~~, removal of punctuation, ~~correcting~~ words, removal urls, mentions emojis, spelling corrections, lowercasing, tokenization, removal of empty tweets.

2) Embeddings

Word2vec, method='mean', unk strategy='zero'

3) Model

input dim=300, hidden dims=[300, 150, 75], activation='gelu', optimizer='rmsprop', lr=0.0005, loss function=CrossEntropyLoss

AI2 Homework 2 - Results

On this page we are going to represent the results of our experiments

Exp no 1

Data : lemmatization

Embeddings : w2v_model, emb_dim = 300, method = mean, unk_strategy = zero

Model : input_dim = 300, hidden_dims = [256, 128], output_dim = 2, activation = relu, dropout_rate = 0.4, batchnorm = True

Opti : adam lr : 0,001 loss : CrossEntropyLoss epochs : 10

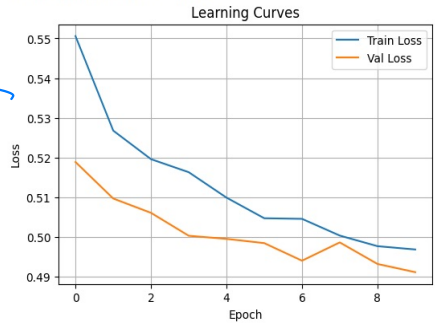
Results

Epoch 1 → 10:

- Train loss: 0.5214 → 0.4524
- Val loss 0.4978 → 0.4767
- Acc : 0.7546 → 0.7741
- F1 : 0.7630 → 0.7687
- AUC : 0.8375 → 0.8542

Results seem good. Both Train Loss and Val loss gradually decrease which is a good sign

Created submission.csv for Kaggle.



Exp no2

Data : same

Embeddings : same

model : input_dim = 300 hidden_dims = [512, 256, 128], output_dim = 2, activation = gelu, dropout = 0.4

Opti : rmsprop lr : 0.0005 loss : CrossEntropyloss epochs : 20

Results

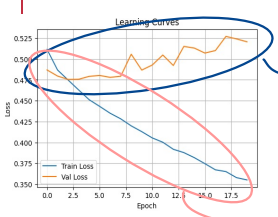
Epoch 1 → 10 → 20 :

- Train loss : 0.5115 → 0.4130 → 0.3551
- Val loss : 0.4869 → 0.4867 → 0.5206
- Acc : 0.7625 → 0.7718 → 0.7617
- F1 : 0.7630 → 0.7764 → 0.7557
- AUC : 0.8452 → 0.8533 → 0.8430

Conclusion : Overfitting. Early dropout (epoch 8 → 10) is more likeble. ~~Train loss~~ was okay but val loss went high.

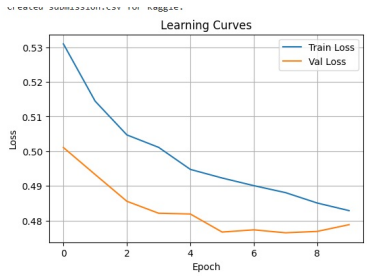
we should do : capacity ↓
dropout ↑
early dropping

↳ overfitting
↳ big capacity
↳ maybe dropout was not enough for regularization



→ Val loss is increased dramatically

→ Optimizer worked well on Train Loss



Exp no3

Data : same

Embeddings : same

Model : hidden_dims = [128, 64], dropout = 0.5

opti = same lr = same loss = same epochs = 10

→ train loss went well → optimizer fits

Results

Epoch 1 → 10 :

- Train loss : 0.5286 → 0.4760
- Val loss : 0.4965 → 0.4754
- Acc : 0.7576 → 0.7726
- F1 : 0.7637 → 0.7776
- AUC : 0.8391 → 0.8543

Better view. It needs more work because even if both curves are decreasing they don't get close to each other

Conclusion : way better. The diagram is very good

Best experiment so far, so :

- Early dropout is more likeble
- maybe gelu > relu

Dropout worked better and less hidden units

Exp no 4 → stick to the preview model and change the embeddings

Data: saul

Model: saul

Embeddings: let's try glove 100d

Second try: glove 200d

Third try: glove 300d

glove 300d

Results

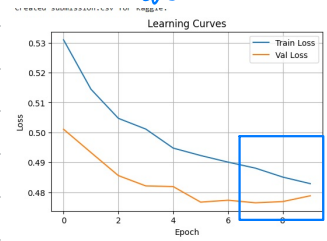
	glove 100d	glove 200d	glove 300d
Train loss:	0.5750 → 0.5352	0.5574 → 0.5128	0.5506 → 0.4968
Val loss:	0.5522 → 0.5228	0.5325 → 0.5024	0.5188 → 0.4911
Acc:	0.7154 → 0.7355	0.7289 → 0.7503	0.7390 → 0.7596
F1:	0.7326 → 0.7256	0.7387 → 0.7472	0.7434 → 0.7576
AUC:	0.7955 → 0.8170	0.8090 → 0.8335	0.8200 → 0.8419

Conclusion: As we expected. 300d > 200d > 100d. More dimension → richer semantic representation. Less information loss in similar words. Better generalization.

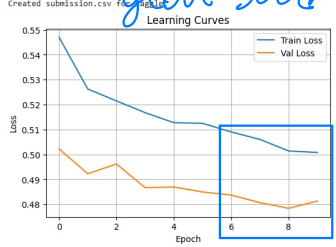
So is it better w2v or glove 300d?
w2v is the winner?

	w2v	glove
Train:	0.52 → 0.47	0.55 → 0.49
Val:	0.49 → 0.47	0.51 → 0.49
Acc:	0.75 → 0.77	0.73 → 0.75
F_1:	0.76 → 0.77	0.74 → 0.75
AUC:	0.83 → 0.85	0.82 → 0.84

w2v



glove 300d



So we keep w2v.

In both (w2v, glove), method was 'mean' unk strategy 'zero'

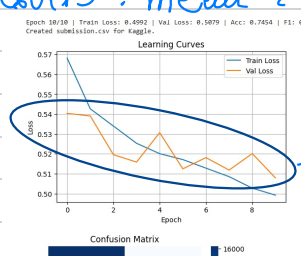
Exp no 5

Let's try w2v but method 'max,

Results

Train loss: 0.5632 → 0.4992
Val loss: 0.5403 → 0.5079
Acc : 0.7225 → 0.7454
F_1 : 0.7390 → 0.7405
AUC : 0.8044 → 0.8283

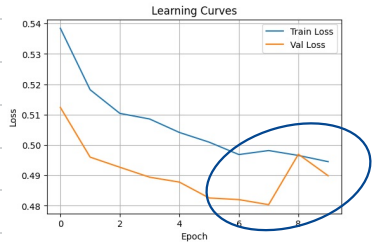
Results : mean > max Let's try 'mean' & 'random'
↳ unk strategy



→ Val loss doesn't seem to be decreased smoothly

Exp no 6

w2v, 'mean', 'random'



Results

Train loss : 0.5896 → 0.5412
Val loss : 0.5649 → 0.5481
Acc : 0.7039 → 0.7143
F1 : 0.7002 → 0.7209
AUC : 0.7791 → 0.7929

In comparison to 'zero' 'random' doesn't work well

Conclusion. It went horrible. So we stick to:
w2v, 'mean', 'zero' ← Final embedding settings.

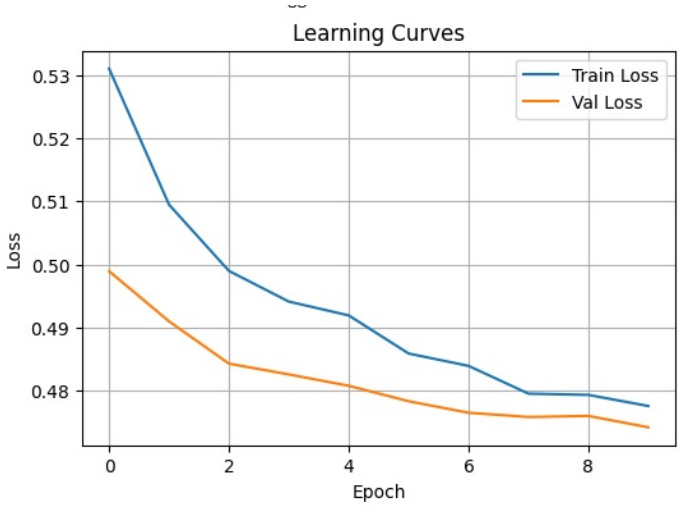
Exp with batch size & learning rate.

Below is an array of all the test I tried:

Batch size	learn. rate
16	0.0005 0.0001
32	0.01 0.0005 0.001
64	0.001 0.0005
128	0.001, 0.0008, 0.0003

	B.S.	L.R.	B.S.	L.R.
	1) 16	& 0.0005	2) 16	& 0.0001
Train loss:	0.5470 →	0.5008	0.5657 →	0.5120
Val loss:	0.5029 →	0.4813	0.5043 →	0.4831
Acc :	0.7533 →	0.7711	0.7531 →	0.7654
F1 :	0.7649 →	0.7751	0.7497 →	0.7735
AUC :	0.8361 →	0.8526	0.8333 →	0.8504
	3) 32	& 0.01	4) 32	& 0.0005
Train loss:	0.5385 →	0.4945	0.5325 →	0.4838
Val loss:	0.5124 →	0.4899	0.5001 →	0.4777
Acc :	0.7520 →	0.7575	0.7552 →	0.7701
F1 :	0.7300 →	0.7785	0.7437 →	0.7630
AUC :	0.8396 →	0.8533	0.8383 →	0.8536
	5) 32	& 0.001	6) 64	& 0.001
Train loss:	0.5309 →	0.4829	0.5252 →	0.4744
Val loss:	0.5011 →	0.4788	0.4983 →	0.4806
Acc :	0.7573 →	0.7719	0.7572 →	0.7719
F1 :	0.7575 →	0.7627	0.7477 →	0.7794
AUC :	0.8383 →	0.8542	0.8397 →	0.8535
	7) 64	& 0.0005	8) 128	& 0.001
Train loss:	0.5310 →	0.4776	0.5247 →	0.4711
Val loss:	0.4989 →	0.4742	0.4989 →	0.4753
Acc :	0.7535 →	0.7731	0.7538 →	0.7718
F1 :	0.7613 →	0.7721	0.7394 →	0.7763
AUC :	0.8380 →	0.8541	0.8391 →	0.8541
	9) 128	& 0.0008	10) 128	& 0.003
Train loss:	0.5280 →	0.4676	0.5201 →	0.4672
Val loss:	0.4963 →	0.4748	0.5024 →	0.4779
Acc :	0.7567 →	0.7716	0.7521 →	0.7698
F1 :	0.7577 →	0.7696	0.7441 →	0.7772
AUC:	0.8389 →	0.8541	0.8358 →	0.853

Winner!



Winner's diagram.

Both Train loss and Val loss decrease steadily over the epochs, which means that the model learns effectively

Val loss remains always lower than Train loss which means there is no overfitting.

Both curves continuously decline = good generation and smooth learning

And the winner is : batch_size = 64
learning_rate = 0.0005

- Batch size = 64 offers a great balance between randomness and stability. It doesn't overshoot in loss surface. Is GPU-friendly and it doesn't overfit
- learning rate = 0.0005 is not too aggressive, nor slow and works best with batch size 64

Best so far

- Embeddings : w2v 'mean' 'zero'
- Dataloader : bs = 64, lr = 0.0005

Let's move back to model:

- Let's try some tests for hidden-di
- [256 128]
- [300, 150]
- [128, 64]
- [256, 128, 64]
- [512, 256, 128]
- [300, 150, 75]
- [128, 128, 64]

Results

	(1) [256 128]	(2) [300 150]	(3) [128, 64]
Train :	0.5227 → 0.4644	0.5232 → 0.4582	0.5291 → 0.4789
Val :	0.4957 → 0.4717	0.5025 → 0.4771	0.5020 → 0.4743
Acc :	0.7551 → 0.7745	0.7511 → 0.7743	0.7537 → 0.7725
F1 :	0.7655 → 0.7734	0.7639 → 0.7702	0.7382 → 0.7769
Auc :	0.8406 → 0.8564	0.8373 → 0.8555	0.8382 → 0.8544

	(4) [256, 128, 64]	(5) [512, 256, 128]	(6) [300, 150, 75]
Train :	0.5312 → 0.4647	0.5216 → 0.4525	0.5286 → 0.4608
Val :	0.4966 → 0.4711	0.4969 → 0.4766	0.4973 → 0.4708
Acc :	0.7564 → 0.7739	0.7556 → 0.7684	0.7561 → 0.7748
F1 :	0.7456 → 0.7714	0.7671 → 0.7547	0.7675 → 0.7745
Auc :	0.8394 → 0.8565	0.8408 → 0.8557	0.8408 → 0.8560

↳ it's a winner!

	(7) [128, 128, 64]
Train :	0.5337 → 0.4757
Val :	0.4983 → 0.4768
Acc :	0.7556 → 0.7712
F1 :	0.7514 → 0.7606
Auc :	0.8373 → 0.8544

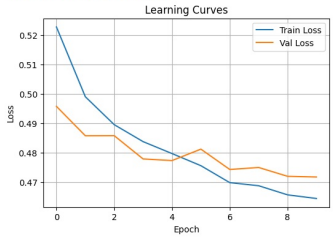
[300 150, 75] worked really well. This pair of hidden dims was combined with word2vec, which is 300 dimensions. So first layer and the dimensions of w2v worked each.

*All tests were with:
'gelu'
'drop_rate' = 0.5
'batch_norm' = True
'input_dims' = 300
'output_dims' = 2

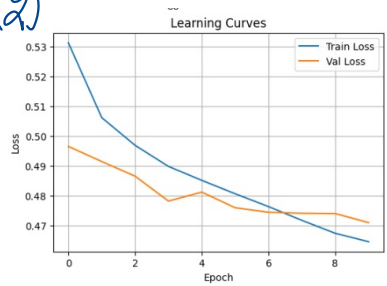
There is smooth compression of information, meaning each layer progressively compressed the data. This aids in the natural emergence of critical patterns. With 3 hidden dims, the model can learn more abstract representations without overfitting. In the end, there is a balanced between capacity and regularization

(1)

Created submission.csv for Kaggle.

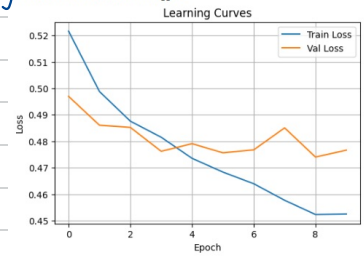


(2)

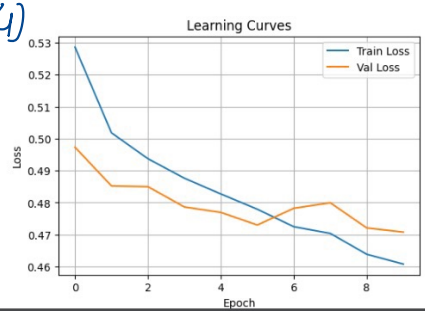


(3)

Created submission.csv for Kaggle.

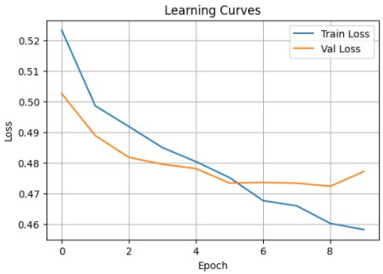


(4)



Epoch 10/10 | Train Loss: 0.4582 | Val Loss: 0.4771 | Acc: 0.7743 | F1: 0.7702 | AUC: 0
Created submission.csv for Kaggle.

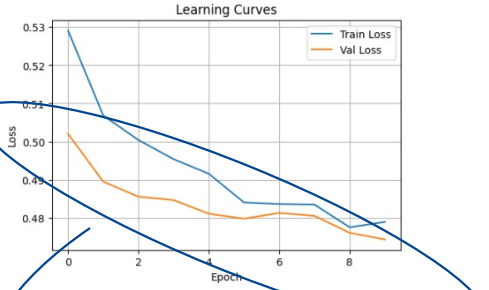
(5)



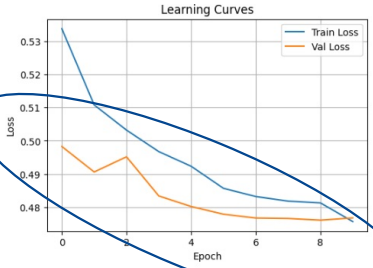
Confusion Matrix

(6)

Created submission.csv for Kaggle.



(7)



Winner's diagram

In all the other diagrams val loss is higher than train loss which is not a good sign

Val loss on (6) decrease smoother than (7) which indicates that the model is learning & generating better

Let's stop and evaluate our options so far:

First try:

- Train loss: 0.5214 → 0.4524
- Val loss 0.4978 → 0.4767
- Acc : 0.7546 → 0.7741
- F1 : 0.7630 → 0.7687
- AUC : 0.8375 → 0.8542

After changing optimizer, hidden_dims & epochs:

- Train loss : 0.5286 → 0.4760
- Val loss : 0.4965 → 0.4754
- Acc : 0.7576 → 0.7726
- F1 : 0.7637 → 0.7776
- AUC : 0.8391 → 0.8543

After choosing best batch size & learn. rate:

- Train loss: 0.5310 → 0.4776
- Val loss: 0.4989 → 0.4742
- Acc : 0.7535 → 0.7731
- F1 : 0.7613 → 0.7721
- AUC : 0.8380 → 0.8541

After choosing best hidden dims

- Train loss : 0.5286 → 0.4608
- Val loss : 0.4973 → 0.4708
- Acc : 0.7561 → 0.7748
- F1 : 0.7675 → 0.7745
- AUC : 0.8408 → 0.8560

Best so far

Next try: optimizers & loss functions & activations

So far: optimizer: rmsprop
loss: CrossEntropyLoss
activ: gelu

First, let's test activators

Results of 'gelu':

- Train loss: 0.5986 → 0.4608
- Val loss: 0.4973 → 0.4708
- Acc: 0.7561 → 0.7748
- F1: 0.7675 → 0.7745
- AUC: 0.8408 → 0.8560

Is a winner?

GeLU: Gaussian Error Linear Unit

Gelu retains smaller values with less weight while gradually allowing larger ones to pass

Results of 'relu':

- Train loss: 0.5439 → 0.4867
- Val loss: 0.5051 → 0.4772
- Acc: 0.7583 → 0.7726
- F1: 0.7601 → 0.7720
- AUC: 0.8342 → 0.8524

- It is smoother
- It is suitable for NLP

Finally it handles the continuous subconsious patterns of language better than behaviors of Relu and Tanh

Results of 'tanh':

- Train loss: 0.5462 → 0.5116
- Val loss: 0.5178 → 0.4994
- Acc: 0.7486 → 0.7575
- F1: 0.7418 → 0.7641
- AUC: 0.8271 → 0.8385

Let's try optimizers

Results of rmsprop:

- Train loss : 0.5986 → 0.4608
- Val loss : 0.4973 → 0.4708
- Acc : 0.7561 → 0.7748
- F1 : 0.7675 → 0.7745
- AUC : 0.8408 → 0.8560

Results of adam:

- Train loss : 0.5432 → 0.4786
- Val loss : 0.4978 → 0.4739
- Acc : 0.7560 → 0.7739
- F1 : 0.7518 → 0.7703
- AUC : 0.8374 → 0.8552

Results of sgd:

- Train loss: 0.6638 → 0.5260
- Val loss: 0.5743 → 0.5012
- Acc: 0.7164 → 0.7540
- F1 : 0.7064 → 0.7520
- AUC : 0.7910 → 0.8361

Winner:

RMSprop is an adaptive optimizer that adjusts the learning rate individually for each parameter. It focuses on recent gradient values using exponential moving averages, avoiding abrupt increases or decreases. Compared to Sgd or Adam, it is adaptive with controlled decay, smooth training and stable convergence.

Let's try loss functions

Results of CrossEntropyLoss — Is a winner!

- Train loss : 0.5986 → 0.4608
- Val loss : 0.4973 → 0.4708
- Acc : 0.7561 → 0.7748
- F1 : 0.7675 → 0.7745
- AUC : 0.8408 → 0.8560

Results of BCEwithlogitsLoss (output = L)

- Train loss : 0.5315 → 0.4785
- Val loss : 0.4991 → 0.4783
- Acc : 0.7550 → 0.7688
- F1 : 0.7651 → 0.7781
- AUC : 0.8389 → 0.8519

Does batchnorm work?

In all our tries we used batchnorm.

I tried once to use_batchnorm = False and reduced the speed of all values. The reason is batch-norm normalizes the activations in each batch, which reduces internal covariate shift. This leads to faster convergence and smoother loss curves. Additionally it has a regularising effect. Reduces sensitivity to initialization and help gradients flow more predictably

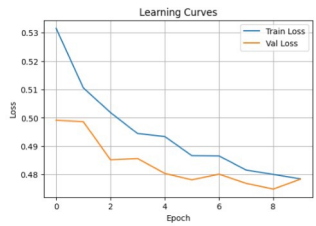
Okay ... we tried to excel embeddings and model ... now it is time for data!

Data until now worked well. On page 1 I mentioned all the settings I applied on my data. Here are some more experiments:

Now let's try lemmatization & 2 types of stemming

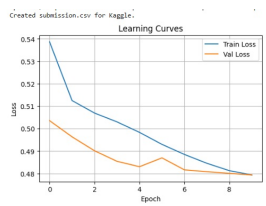
Without lemmatization:

- Train loss : 0.5347 → 0.4764
- Val loss : 0.4957 → 0.4737
- Acc : 0.7582 → 0.7727
- F1 : 0.7544 → 0.7667
- AUC : 0.8387 → 0.8553



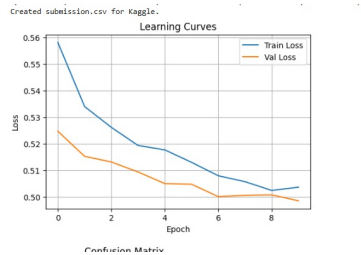
With lemmatization:

- Train loss : 0.5387 → 0.4793
- Val loss : 0.5036 → 0.4794
- Acc : 0.7525 → 0.7699
- F1 : 0.7520 → 0.7738
- AUC : 0.8355 → 0.8512



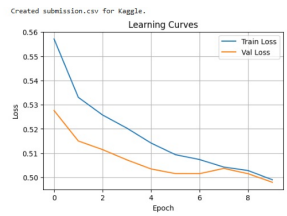
With Porter Stemmer:

- Train loss : 0.5580 → 0.5036
- Val loss : 0.5247 → 0.4985
- Acc: 0.7376 → 0.7540
- F_1: 0.7444 → 0.7581
- AUC: 0.8185 → 0.8366



With Snowball Stemmer:

- Train loss: 0.5572 → 0.4990
- Val loss: 0.5276 → 0.4980
- Acc : 0.7340 → 0.7569
- F_1 : 0.7193 → 0.7527
- AUC : 0.8168 → 0.8379



Stemmers didn't work really well.
And lemmatization did not change the outcome dramatically

So after a lot of research on embeddings and model I knew that I needed to clean the data manually.

So I created the following functions:

def frequent_dict = it gets as an input a dataset and it reads it in batches. The purpose of it is to create a .txt file with all the words of a dataset and the frequency of them

- train_df : 120.378 words
- val_df : 47.349 words
- test_df : 28.580 words

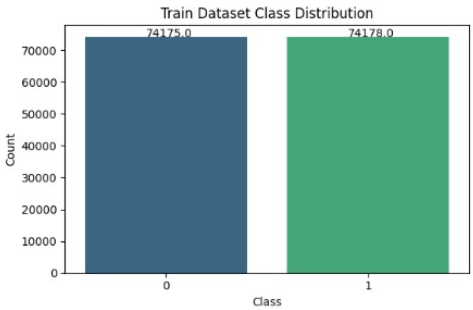
def build_correction_dict & def save_dict_as_py_file:
Basically they read all the words of a txt file and they create a .py file in the form of a py dictionary:

correct_dict : {"word1" : " ", "word2" : " ", ... }

So basically me to change the words.
Of course I couldn't change 120.378 words.
So I chose the test_df with the 28.580 words

Did I changed 28.580 words?

No I changed only 5.000. But it didn't work as I expected. Let's see the results



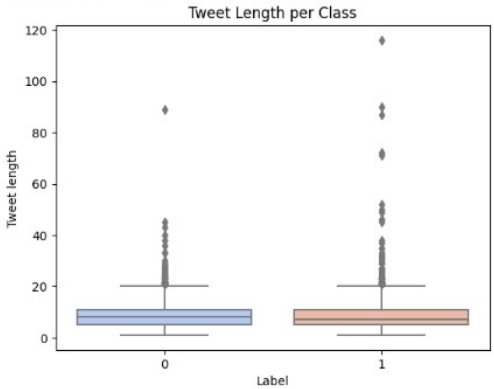
Class 0 count : 74,175
Class 1 count : 74,178

3-sample difference
Very balanced

Class 0 count : 21,193
Class 1 count : 21,197

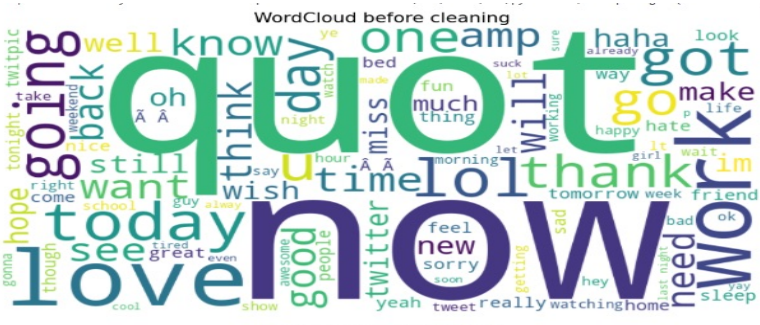
4-sample difference
Very balanced

Very long tweets in train (≥50 words): 8



Long words : only 2
Short words : 8,189 tweets (5.5%)

The majority of tweets are quite short, with 75% being 11 words or fewer.



The quality has significally improved:

- Terms like quot, amp, Â, Ã and other irrelevant tokens have been removed.

- The words are much clearer and more representative.

- There is an increased presence of thematic words

submission (13).csv Complete · 18s ago	0.77060	☐
submission (12).csv Complete · 8m ago	0.76998	☐
submission (11).csv Complete · 16m ago	0.77310	☐
submission (10).csv Complete · 5h ago	0.77682	☑

without lemma correcting words and applying fast_correct_text

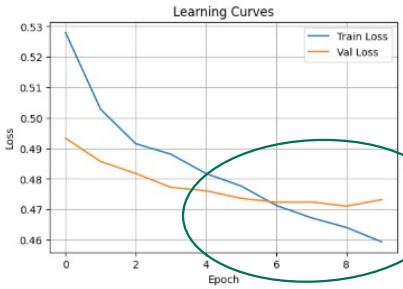
with def correct_words

with def correct_words & lemmatization

with def correct_words & lemmatization & fast_correct_words

Results: without anything my model works better!

```
Epoch 1/10 | Train Loss: 0.5280 | Val Loss: 0.4933 | Acc: 0.7593 | F1: 0.7592 | AUC: 0.8413
Epoch 2/10 | Train Loss: 0.5028 | Val Loss: 0.4857 | Acc: 0.7632 | F1: 0.7644 | AUC: 0.8459
Epoch 3/10 | Train Loss: 0.4915 | Val Loss: 0.4817 | Acc: 0.7677 | F1: 0.7627 | AUC: 0.8490
Epoch 4/10 | Train Loss: 0.4881 | Val Loss: 0.4772 | Acc: 0.7696 | F1: 0.7752 | AUC: 0.8526
Epoch 5/10 | Train Loss: 0.4817 | Val Loss: 0.4761 | Acc: 0.7712 | F1: 0.7753 | AUC: 0.8535
Epoch 6/10 | Train Loss: 0.4776 | Val Loss: 0.4736 | Acc: 0.7717 | F1: 0.7704 | AUC: 0.8544
Epoch 7/10 | Train Loss: 0.4713 | Val Loss: 0.4723 | Acc: 0.7723 | F1: 0.7699 | AUC: 0.8554
Epoch 8/10 | Train Loss: 0.4672 | Val Loss: 0.4724 | Acc: 0.7749 | F1: 0.7741 | AUC: 0.8556
Epoch 9/10 | Train Loss: 0.4640 | Val Loss: 0.4710 | Acc: 0.7734 | F1: 0.7713 | AUC: 0.8564
Epoch 10/10 | Train Loss: 0.4593 | Val Loss: 0.4732 | Acc: 0.7715 | F1: 0.7784 | AUC: 0.8563
```



So that would be my final submission. I would love to try grid search and optuna to find best hyperparameter tuning. But I was lack of time. My last experiment was: early stopping.

Why early stopping? Because of this

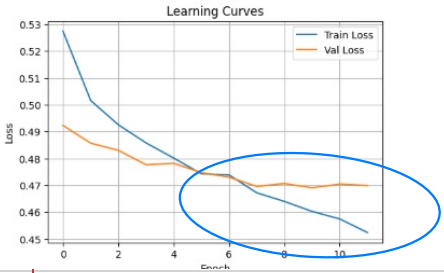
It seems that after epoch 9-10, model doesn't generate that well. So I try:

→ early stopping
→ dropout adjustment

Early stopping

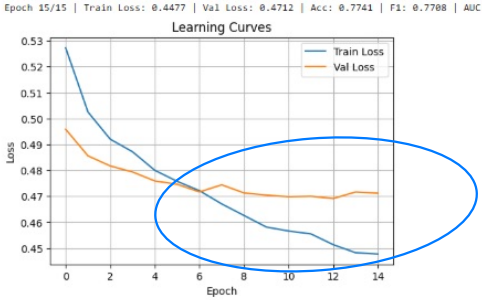
```
Epoch 1/12 | Train Loss: 0.5274 | Val Loss: 0.4923 | Acc: 0.7604 | F1: 0.7579 | AUC: 0.8410
Epoch 2/12 | Train Loss: 0.5016 | Val Loss: 0.4857 | Acc: 0.7646 | F1: 0.7698 | AUC: 0.8467
Epoch 3/12 | Train Loss: 0.4925 | Val Loss: 0.4830 | Acc: 0.7654 | F1: 0.7718 | AUC: 0.8488
Epoch 4/12 | Train Loss: 0.4858 | Val Loss: 0.4776 | Acc: 0.7693 | F1: 0.7658 | AUC: 0.8519
Epoch 5/12 | Train Loss: 0.4801 | Val Loss: 0.4782 | Acc: 0.7692 | F1: 0.7591 | AUC: 0.8534
Epoch 6/12 | Train Loss: 0.4743 | Val Loss: 0.4747 | Acc: 0.7721 | F1: 0.7717 | AUC: 0.8539
Epoch 7/12 | Train Loss: 0.4739 | Val Loss: 0.4730 | Acc: 0.7737 | F1: 0.7754 | AUC: 0.8550
Epoch 8/12 | Train Loss: 0.4672 | Val Loss: 0.4696 | Acc: 0.7759 | F1: 0.7720 | AUC: 0.8574
Epoch 9/12 | Train Loss: 0.4640 | Val Loss: 0.4706 | Acc: 0.7771 | F1: 0.7775 | AUC: 0.8567
Epoch 10/12 | Train Loss: 0.4603 | Val Loss: 0.4691 | Acc: 0.7763 | F1: 0.7782 | AUC: 0.8579
Epoch 11/12 | Train Loss: 0.4575 | Val Loss: 0.4704 | Acc: 0.7771 | F1: 0.7794 | AUC: 0.8578
Epoch 12/12 | Train Loss: 0.4524 | Val Loss: 0.4699 | Acc: 0.7780 | F1: 0.7769 | AUC: 0.8583
```

patience = 3
epochs = 12



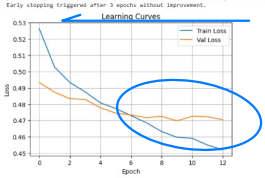

```
Epoch 1/15 | Train Loss: 0.5273 | Val Loss: 0.4958 | Acc: 0.7565 | F1: 0.7494 | AUC: 0.8392
Epoch 2/15 | Train Loss: 0.5025 | Val Loss: 0.4856 | Acc: 0.7654 | F1: 0.7602 | AUC: 0.8465
Epoch 3/15 | Train Loss: 0.4920 | Val Loss: 0.4817 | Acc: 0.7662 | F1: 0.7636 | AUC: 0.8492
Epoch 4/15 | Train Loss: 0.4872 | Val Loss: 0.4794 | Acc: 0.7671 | F1: 0.7739 | AUC: 0.8518
Epoch 5/15 | Train Loss: 0.4800 | Val Loss: 0.4759 | Acc: 0.7709 | F1: 0.7687 | AUC: 0.8535
Epoch 6/15 | Train Loss: 0.4757 | Val Loss: 0.4746 | Acc: 0.7703 | F1: 0.7750 | AUC: 0.8539
Epoch 7/15 | Train Loss: 0.4721 | Val Loss: 0.4716 | Acc: 0.7727 | F1: 0.7730 | AUC: 0.8558
Epoch 8/15 | Train Loss: 0.4670 | Val Loss: 0.4744 | Acc: 0.7708 | F1: 0.7791 | AUC: 0.8557
Epoch 9/15 | Train Loss: 0.4626 | Val Loss: 0.4713 | Acc: 0.7752 | F1: 0.7735 | AUC: 0.8568
Epoch 10/15 | Train Loss: 0.4581 | Val Loss: 0.4704 | Acc: 0.7756 | F1: 0.7768 | AUC: 0.8571
Epoch 11/15 | Train Loss: 0.4566 | Val Loss: 0.4698 | Acc: 0.7758 | F1: 0.7763 | AUC: 0.8572
Epoch 12/15 | Train Loss: 0.4555 | Val Loss: 0.4700 | Acc: 0.7747 | F1: 0.7716 | AUC: 0.8573
Epoch 13/15 | Train Loss: 0.4513 | Val Loss: 0.4691 | Acc: 0.7767 | F1: 0.7773 | AUC: 0.8579
Epoch 14/15 | Train Loss: 0.4482 | Val Loss: 0.4716 | Acc: 0.7747 | F1: 0.7773 | AUC: 0.8566
Epoch 15/15 | Train Loss: 0.4477 | Val Loss: 0.4712 | Acc: 0.7741 | F1: 0.7708 | AUC: 0.8568
```

patience = 3
epochs = 15




```
Epoch 1/20 | Train Loss: 0.5264 | Val Loss: 0.4933 | Acc: 0.7588 | F1: 0.7609 | AUC: 0.8413
Epoch 2/20 | Train Loss: 0.5020 | Val Loss: 0.4874 | Acc: 0.7635 | F1: 0.7726 | AUC: 0.8465
Epoch 3/20 | Train Loss: 0.4933 | Val Loss: 0.4834 | Acc: 0.7664 | F1: 0.7658 | AUC: 0.8489
Epoch 4/20 | Train Loss: 0.4876 | Val Loss: 0.4830 | Acc: 0.7654 | F1: 0.7519 | AUC: 0.8508
Epoch 5/20 | Train Loss: 0.4808 | Val Loss: 0.4778 | Acc: 0.7701 | F1: 0.7605 | AUC: 0.8536
Epoch 6/20 | Train Loss: 0.4774 | Val Loss: 0.4745 | Acc: 0.7720 | F1: 0.7751 | AUC: 0.8541
Epoch 7/20 | Train Loss: 0.4730 | Val Loss: 0.4734 | Acc: 0.7713 | F1: 0.7665 | AUC: 0.8551
Epoch 8/20 | Train Loss: 0.4687 | Val Loss: 0.4717 | Acc: 0.7744 | F1: 0.7655 | AUC: 0.8563
Epoch 9/20 | Train Loss: 0.4632 | Val Loss: 0.4726 | Acc: 0.7740 | F1: 0.7780 | AUC: 0.8568
Epoch 10/20 | Train Loss: 0.4598 | Val Loss: 0.4698 | Acc: 0.7758 | F1: 0.7744 | AUC: 0.8573
Epoch 11/20 | Train Loss: 0.4591 | Val Loss: 0.4726 | Acc: 0.7739 | F1: 0.7687 | AUC: 0.8564
Epoch 12/20 | Train Loss: 0.4550 | Val Loss: 0.4723 | Acc: 0.7766 | F1: 0.7796 | AUC: 0.8569
Epoch 13/20 | Train Loss: 0.4521 | Val Loss: 0.4705 | Acc: 0.7758 | F1: 0.7746 | AUC: 0.8573
Early stopping triggered after 3 epochs without improvement.
```

patience = 3
epochs = 20.



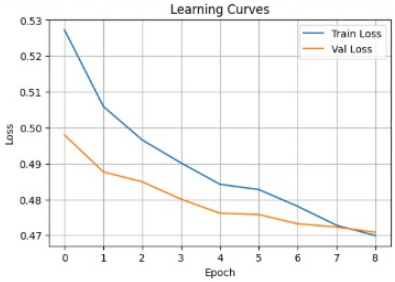
Epochs ↑ → doesn't work well,
so I chose epochs = 9

Last but not least : dropout adjustment.

We can see : train loss ↓ val loss ↘ : dropout ↑

Dropout : 0.55 - epochs 9

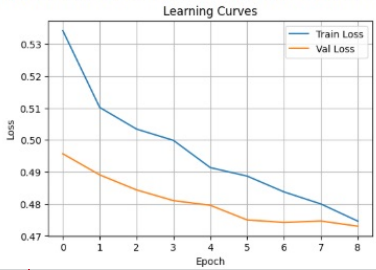
```
Epoch 1/9 | Train Loss: 0.5273 | Val Loss: 0.4980 | Acc: 0.7551 | F1: 0.7469 | AUC: 0.8378
Epoch 2/9 | Train Loss: 0.5059 | Val Loss: 0.4877 | Acc: 0.7640 | F1: 0.7687 | AUC: 0.8452
Epoch 3/9 | Train Loss: 0.4966 | Val Loss: 0.4849 | Acc: 0.7653 | F1: 0.7656 | AUC: 0.8479
Epoch 4/9 | Train Loss: 0.4902 | Val Loss: 0.4801 | Acc: 0.7689 | F1: 0.7633 | AUC: 0.8509
Epoch 5/9 | Train Loss: 0.4842 | Val Loss: 0.4762 | Acc: 0.7709 | F1: 0.7739 | AUC: 0.8531
Epoch 6/9 | Train Loss: 0.4828 | Val Loss: 0.4758 | Acc: 0.7721 | F1: 0.7673 | AUC: 0.8536
Epoch 7/9 | Train Loss: 0.4781 | Val Loss: 0.4732 | Acc: 0.7729 | F1: 0.7712 | AUC: 0.8549
Epoch 8/9 | Train Loss: 0.4728 | Val Loss: 0.4723 | Acc: 0.7744 | F1: 0.7773 | AUC: 0.8558
Epoch 9/9 | Train Loss: 0.4699 | Val Loss: 0.4709 | Acc: 0.7756 | F1: 0.7762 | AUC: 0.8564
```



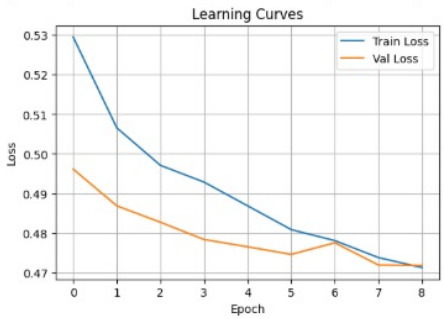
← slightly better!

Dropout : 0.6 - epochs 9

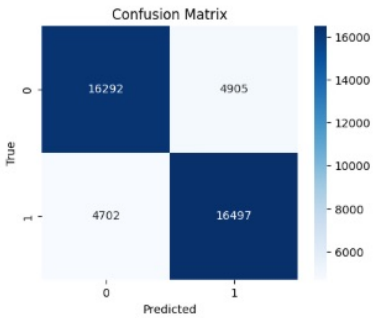
```
Epoch 1/9 | Train Loss: 0.5342 | Val Loss: 0.4957 | Acc: 0.7567 | F1: 0.7480 | AUC: 0.8397
Epoch 2/9 | Train Loss: 0.5102 | Val Loss: 0.4891 | Acc: 0.7624 | F1: 0.7621 | AUC: 0.8442
Epoch 3/9 | Train Loss: 0.5034 | Val Loss: 0.4844 | Acc: 0.7666 | F1: 0.7664 | AUC: 0.8477
Epoch 4/9 | Train Loss: 0.4999 | Val Loss: 0.4810 | Acc: 0.7689 | F1: 0.7696 | AUC: 0.8498
Epoch 5/9 | Train Loss: 0.4913 | Val Loss: 0.4795 | Acc: 0.7696 | F1: 0.7626 | AUC: 0.8513
Epoch 6/9 | Train Loss: 0.4887 | Val Loss: 0.4750 | Acc: 0.7730 | F1: 0.7739 | AUC: 0.8536
Epoch 7/9 | Train Loss: 0.4837 | Val Loss: 0.4742 | Acc: 0.7721 | F1: 0.7726 | AUC: 0.8541
Epoch 8/9 | Train Loss: 0.4800 | Val Loss: 0.4746 | Acc: 0.7737 | F1: 0.7736 | AUC: 0.8542
Epoch 9/9 | Train Loss: 0.4746 | Val Loss: 0.4730 | Acc: 0.7742 | F1: 0.7787 | AUC: 0.8554
```



Final diagrams

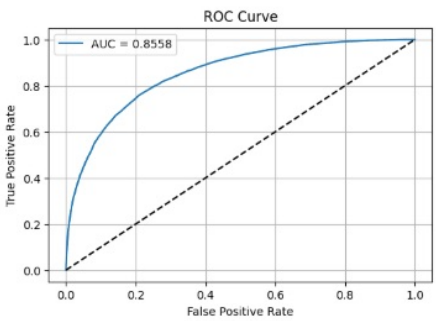


Train loss decreases which means model is effectively learning. Val loss also decreases which means the model is generating well and not overfit.
Both are close: stable learning and balanced model capacity



True Positives : 16,497 → 1
True Negatives : 16,292 → 0
False Positives : 4905 → 1
False Negatives : 4,702 → 0

Fairly balanced. Numbers are close. Similar accuracy, not biased model



ROC : 0.8558 : is excellent

The model distinguishes between the two classes very effectively.

The model demonstrates very good training behavior balanced classification and strong ROC performance

So, final settings

- **Data:** stopwords, punctuation, urls, mentions, emojis, spelling corrections, lowercasing, tokenization
- **Embeddings:** word2vec, 'mean', 'zero'
- **Model:** hid_dims = [300, 150, 75], activ = 'gelu', dropout rate = 0.55, optimizer = 'rmsprop', lr = 0.0005, epochs = 9, loss = CrossEntropy

Thank you so much for your time and appreciate accepting my assignment.