SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.040 Natural Language Processing

NLP Final Project

Cheng Wei Xuan	1006040
Onn Hui Ning	1006132
Vinny Koh	1006036

Members	Contribution	Percentage
Cheng Wei Xuan	Question 1, Design Challenge (Hierarchical Attention Network and XLNet)	33.3%
Onn Hui Ning	Question 2, Design Challenge (FLAIR)	33.3%
Vinny Koh	Question 3, Design Challenge (Roberta)	33.3%

Design Challenge Best Model - RoBERTa

Model Architecture

This section covers an overview of the RoBERTa model architecture that is introduced here.

Overview

RoBERTa (Robustly Optimised BERT Approach) is an optimised version of BERT, addressing key limitations in its training methodology. RoBERTa has the same architecture as BERT but with improvements to its training process, such as using larger batch sizes, larger datasets (CC-NEWS), the removal of next sentence prediction, training on longer sequences, as well as dynamic masking. This results in the production of a better pretrained BERT model that can be further finetuned to cater to specific tasks like sentiment analysis.

BERT Architecture

BERT's model architecture is a multi-layer bidirectional transformer encoder.

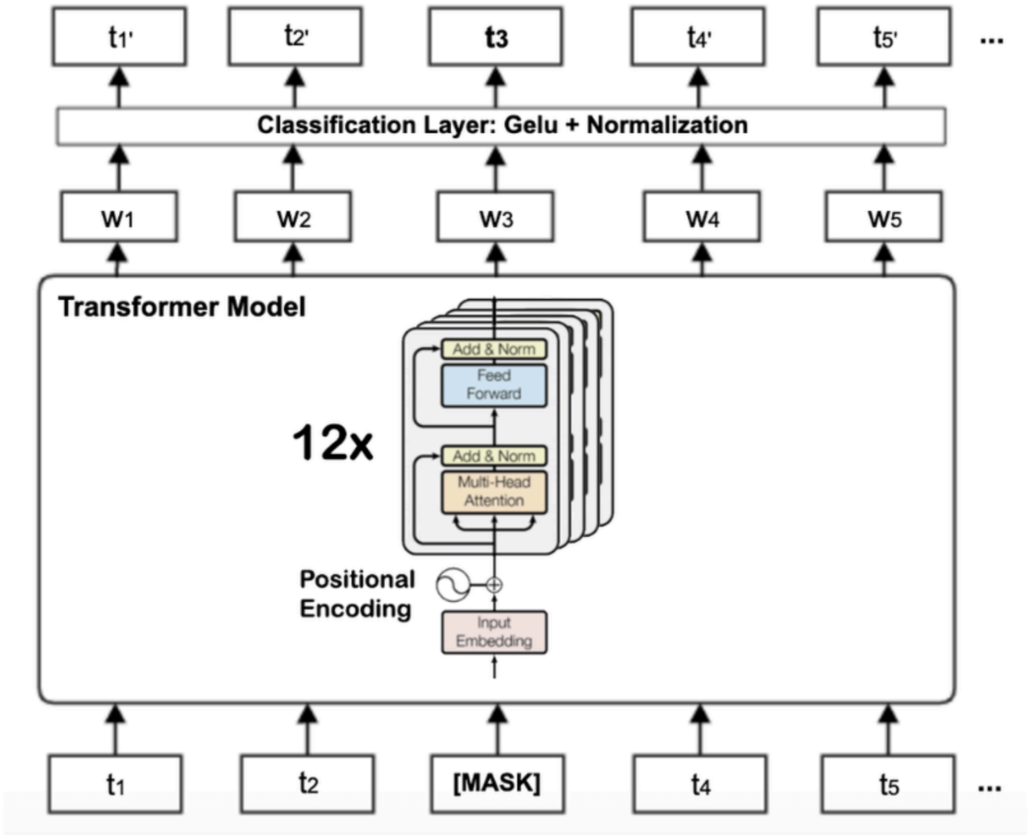


Figure 1. Model architecture of Bert (link)

BERT architecture consists of the following parts:

1. *Input embeddings*: Sum of token embeddings (where words are converted into dense vectors), segment embeddings and positional embeddings
2. *Multi-layer transformer encoder*: Stacks of identical transformer encoder layers where each transformer layer consists of **Multi-Head Self-Attention Mechanism, Feed-Forward Neural Network, Residual Connections and Layer Normalisation**. The number of layers of encoders can be either 12 in Bert-Base or 24 in Bert-Large.

In each encoder layer:

1. Multihead attention

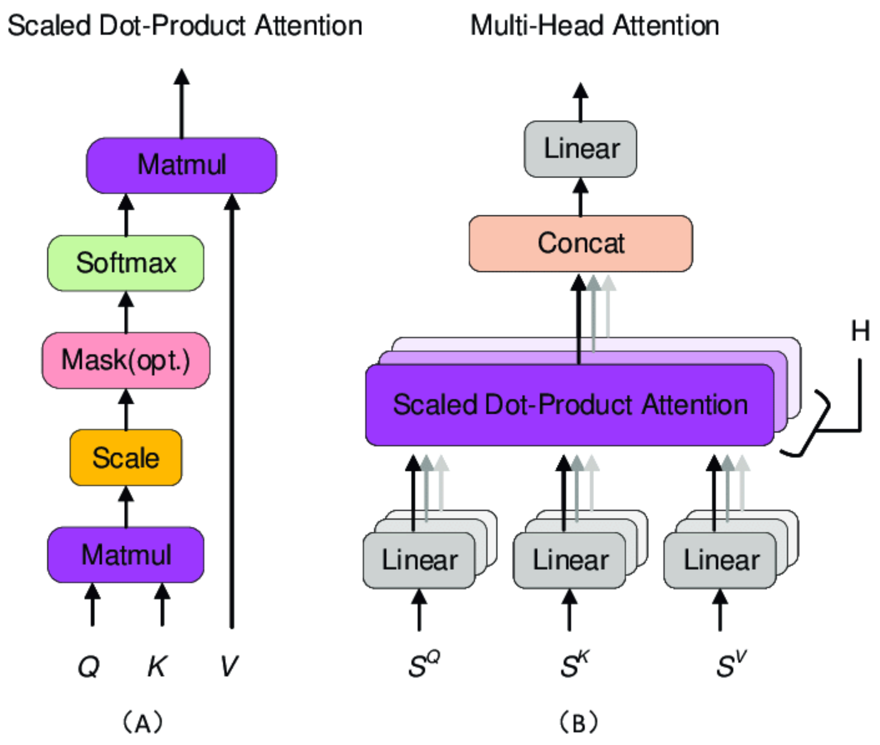


Figure 2. Multihead attention architecture (link)

Each input embedding undergoes linear transformation to form the query, key, and value vector for each position using learnt weight matrices. At each input position i, the output is a weighted sum of all the value vectors in the sentence.

The attention weights for each value vector are the dot product of its key vector and the query vector at position i, which is then passed through softmax to derive a probability. In multi-head attention, this process is performed multiple times in parallel with different learnable projections of the input to capture different aspects of attention. The outputs of all the attention heads are then concatenated and passed through a linear transformation to form the final output.

2. Residual Connections and Normalisation

Residual connections help mitigate vanishing gradients during backpropagation and preserve some parts of the original information, while normalisation helps the model to converge more quickly by reducing internal covariate shifts.

3. Feed forward

The feed forward layer applies non-linear transformations to the embeddings, helping the model capture more complex patterns.

The output of the above transformer encoder layer are the hidden states, which are then passed through task specific layers. For instance, the classification layer maps the hidden states to output labels.

Pre-training of BERT

During pre-training, BERT uses two objectives: masked language modelling and next sentence prediction.

1. *Masked Language Model (MLM)*: A random sample of the tokens in the input sequence is selected and replaced with the special token [MASK]. The MLM objective is to predict the masked tokens with a cross-entropy loss. BERT uniformly selects 15% of the input tokens for possible replacement. Of the selected tokens, 80% are replaced with [MASK], 10% are left unchanged, and 10% are replaced by a randomly selected vocabulary token (link). The MLM objective helps BERT learn contextualised word representations by masking random tokens and predicting them using both left and right context. The use of randomly selected vocabulary tokens helps the model to handle noise and prevent overfitting.

2. *Next Sentence Prediction (NSP)*: NSP aims to predict whether two sentences follow each other in the original text with a binary classification loss. Positive examples are created by taking consecutive sentences from the text corpus, while negative examples are created by pairing segments from different documents. Positive and negative examples are sampled with equal probabilities (link). NSP helps BERT learn about sentence coherence and relationships between sentences, enhancing its performance on tasks that require understanding the connections between text segments.

ROBERTA Architecture

RoBERTa is an optimised version of BERT with the same model architecture but with a better training methodology, as mentioned below:

1. *Dynamic masking*: The original BERT implementation performed masking once during data preprocessing, resulting in a single static mask. However, in RoBERTa, dynamic masking is used, where a different masking pattern is generated every time a sequence is fed to the model to avoid using the same mask for each training instance in every epoch. This increases the amount of training data, enhancing generalisability.

2. *Training without NSP:* While NSP was hypothesised to be an important factor in training the original BERT model, experiments conducted in the research paper (link) found that training without NSP and training with blocks of text from a single document (DOC-SENTENCES) outperform the originally published BERTBASE results.

3. *Larger mini batches*: Bert base was originally trained for 1 million steps with a batch size of 256 sequences. RoBERTa was subsequently trained with larger mini batches of 8K sequences, as previous research in Neural Machine Translation demonstrated that using significantly larger mini-batches during training with a suitably increased learning rate can enhance both optimisation efficiency and final task performance (link).

4. *Byte-level Byte-Pair Encoding (BPE)*: BPE is a method used to break down text into subwords, helping the models handle rare words and large vocabularies more efficiently. BPE vocabulary ranges from 10-100 thousand subword units with unicode characters used as base subword units. By using bytes as base subword units instead, it is possible to encode any text without introducing “unknown” tokens with a modest vocabulary size of 50K units. The original BERT implementation uses a character-level BPE vocabulary of size 30K, which is learnt after preprocessing the input with heuristic tokenisation rules (splitting by punctuation, lowercasing, and whitespace), while RoBERTa uses a larger byte-level BPE vocabulary of 50K subword units without requiring any preprocessing of the input token, thus making it flexible and robust for diverse text formats (link).

5. *Larger dataset corpus*: RoBERTa is trained with a significantly larger dataset that is 10 times more than BERT, totalling over 160 GB of uncompressed text. The larger dataset improved RoBERTa’s model performance.

Model Setup (Code Explanation)

As training RoBERTa from scratch is computationally intensive and requires a large amount of compute resources, finetuning was done on a pre-trained RoBERTa model from Hugging Face.

Custom Dataset and Dataloader

The dataset class **ImdbData** has the **_readImdb** method, which reads the dataset into a dataframe of review and its tag. This dataframe is used in the **_getitem** method, which tokenises the data in the review column of the dataframe. The tokeniser used is the RobertaTokenizer from the pretrained roberta-base and padded to the declared maximum token length, which is a hyperparameter that can be finetuned later.

The expected output for each iteration from the train/test iter includes the following:

1. Input Ids (tokenised sequence): [batch_size, max_token _length]
2. Attention Masks (Mask to avoid performing attention on padding token indices): [batch_size, max_token_length]
3. Token Type Ids (Segment token indices to indicate whether it is the first or second part of sequence): [batch_size, max_token_length]
4. Label tensors: [batch_size]

The Dataset is then loaded into the **createDataLoaders** method, where the train dataset will be shuffled while the test dataset will not. The batch size of the datasets is adjustable for hyperparameter tuning later.

Modified Model Architecture

A pre-trained RoBERTa model from Hugging Face is retrieved, with the weights of the RoBERTa base model frozen by default.

After the hidden states are produced from the pre-trained RoBERTa model, they will be passed through 2 additional linear layers (which are bolded), which will be finetuned for sentiment analysis.

These layers are the pre_classifier layer with ReLU activation to add extra complexity to deal with nonlinear data, the dropout layer to enable better generalisation, as well as the final classification layer to produce the binary label of sentiment either being positive or negative, respectively. The dropout rate is a hyperparameter that can be used for fine tuning later.

During experimentations later, we also tried to unfreeze the entire RoBERTa model alongside the linear classification layers to be fine-tuned.

The overarching structure can be seen in the following, where the bolded layers are the ones that will be fine-tuned (regardless if RoBERTa is frozen or not):

```
RobertaClass(
  (l1): RobertaModel(
    (embeddings): RobertaEmbeddings(
      (word_embeddings): Embedding(50265, 768, padding_idx=1)
      (position_embeddings): Embedding(514, 768, padding_idx=1)
      (token_type_embeddings): Embedding(1, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): RobertaEncoder(
      (layer): ModuleList(
        (0-11): 12 x RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): RobertaPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (pre_classifier): Linear(in_features=768, out_features=768, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
)
```

Model Performance

The model was trained with the following fixed parameters:

1. Epochs: 5 (except for the first)
2. Loss function: Cross-entropy loss
3. Optimizer: Adam

These are the possible hyperparameters to be explored:

1. Batch size: 8,32,64
2. Learning rate (LR): 1e-5, 3e-5
3. Dropout rate: 0.1, 0.3, 0.4
4. Max token length: 128,256

The performance of different explorations can be seen below.

Index	Dropout	Max token length	Batch Size	LR	Epochs	Unfreeze roberta	Test IMDB Accuracy	Released Test Accuracy
1	0.1	128	32	1e-5	10	F	0.805	0.808
2	0.3	128	32	1e-5	5	F	0.793	0.794
3	0.3	128	8	1e-5	5	F	0.799	0.802
4	0.3	128	64	1e-5	5	F	0.78	0.761
5	**0.3**	**256**	**8**	**1e-5**	**5**	**T**	**0.940**	0.950
6	0.3	256	8	3e-5	5	T	0.932	0.959
7	**0.4**	**256**	**8**	**1e-5**	**5**	**T**	0.939	**0.967**
8	0.4	128	8	1e-5	5	T	0.907	0.950

Model Result

The best model was the **7th model**, with an accuracy of 0.967 (3sf), recall of 0.970 (3sf), precision of 0.964 (3sf), and f1 score of 0.967 (3sf) on the released dataset.

Performance Analysis on IMDB test dataset

The best model for the IMDB test dataset is the 5th model, with an accuracy of 0.940.

From experiment 1, it is found that the best model test accuracy at the 10th epoch is 0.805 which is only slightly better than that at the 5th epoch which is 0.794. Since the model training process is time-consuming and the improvements are very small; future experiments will cut short the number of epochs to 5 in order to carry out more exploration on other hyperparameters.

Comparing the test accuracy of the 5th epoch for experiment 1, which is 0.794, to that of experiment 2, which is 0.793, it is shown that the dropout rate does not significantly affect the results.

From experiments 2 to 4 , it can be seen that batch size of 8 performs slightly better than batch sizes of 32 and 64. Furthermore, training times were faster for a smaller batch size; thus, experiments 5 onwards will be carried out with a batch size of 8.

Comparing experiments 3 and 5, the unfreezing of RoBERTa layers leads to significantly better performance; hence, all experiments from 6 onwards will also unfreeze the RoBERTa base model layers.

We noted later, during the release of the test set, that we should have unfrozen the pooler layers in RoBERTa.

```
(pooler): RobertaPooler(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (activation): Tanh()
)
```

RoBERTa originally comes with a randomly initialised linear layer for finetuning; however, we assumed this layer was already pre-trained and thus frozen it. This would explain why unfreezing the RoBERTa model significantly improved model performance, as the randomly initialised linear layer would be unfrozen and finetuned.

However, due to the lack of time, we did not retrain the models with just the unfrozen pooler and linear layers but instead relied on our already trained unfrozen robertas.

Comparing experiments 5 and 6, the increase in learning rate leads to a fall in accuracy. Thus, we will stick to a learning rate of 1e-5 for the remaining experiments.

Furthermore, when observing the training graphs from experiments 5 and 6, as the model trains more, it overfits more to the training data and generalises less well on the test data, leading to falling test accuracy. Hence, we decided to experiment with larger dropout rates of 0.4 in hopes of mitigating overfitting. However, as enforced from experiments 1 and 2, dropout rates had little impact on performance when comparing experiments 5 and 7.

Lastly, comparing experiments 7 and 8, it is shown that the reduction in token length leads to a significant decrease in performance, which may be due to some sentences being longer than 128 tokens long, thus resulting in missing information which hinders model performance.

Therefore, the ideal configuration would be an unfrozen RoBERTa with a max token length of 256, a batch size of 8, a learning rate of 1e-5, and a dropout of 0.3.

Possible areas of exploration include using a learning rate scheduler that can dynamically adjust the learning rate during the training process as well as training for a greater number of epochs. Furthermore, from analysing the dataset, we know that most of the token length in the dataset is larger than 256; however, we were constrained by available computational resources. Given more time and compute power, exploration for larger token length and batch size should be done.

Performance Analysis on released test dataset

Generally, the model performed similarly on both the IMDB test dataset and the newly released dataset for all the models which have RoBERTa layers frozen.

However, the model performed significantly better on the newly released dataset for all the models which did not have their RoBERTa layers frozen.

The best model was from experiment 7, which had the highest dropout rate. This could suggest that dropout rates did play a role in improving the overall generalisability of the model, which resulted in the best accuracy for the new test set.

Overall, the impact of each hyperparameter remains the same, with batch size of 8 providing the best results and a maximum token length of 256 being consistently better than 128, likely because the released test dataset also contained long sentences beyond 128 tokens.

However, it is interesting to note that the model with a higher learning rate (experiment 6) performed better (experiment 5). Unfortunately, as we only had 1 model, we are unable to confirm if this was simply because of the initialisations of the model or if a higher learning rate does improve the model's performance on the test dataset.

Regardless, these results highlight the prowess of transformer models for natural language processing tasks, significantly outperforming any of the other models tried as showcased in the Appendix.

Appendix

This section contains all the reports for models that we also explored for the design challenge, but did not perform as well as our best model.

3 other model architectures were explored:

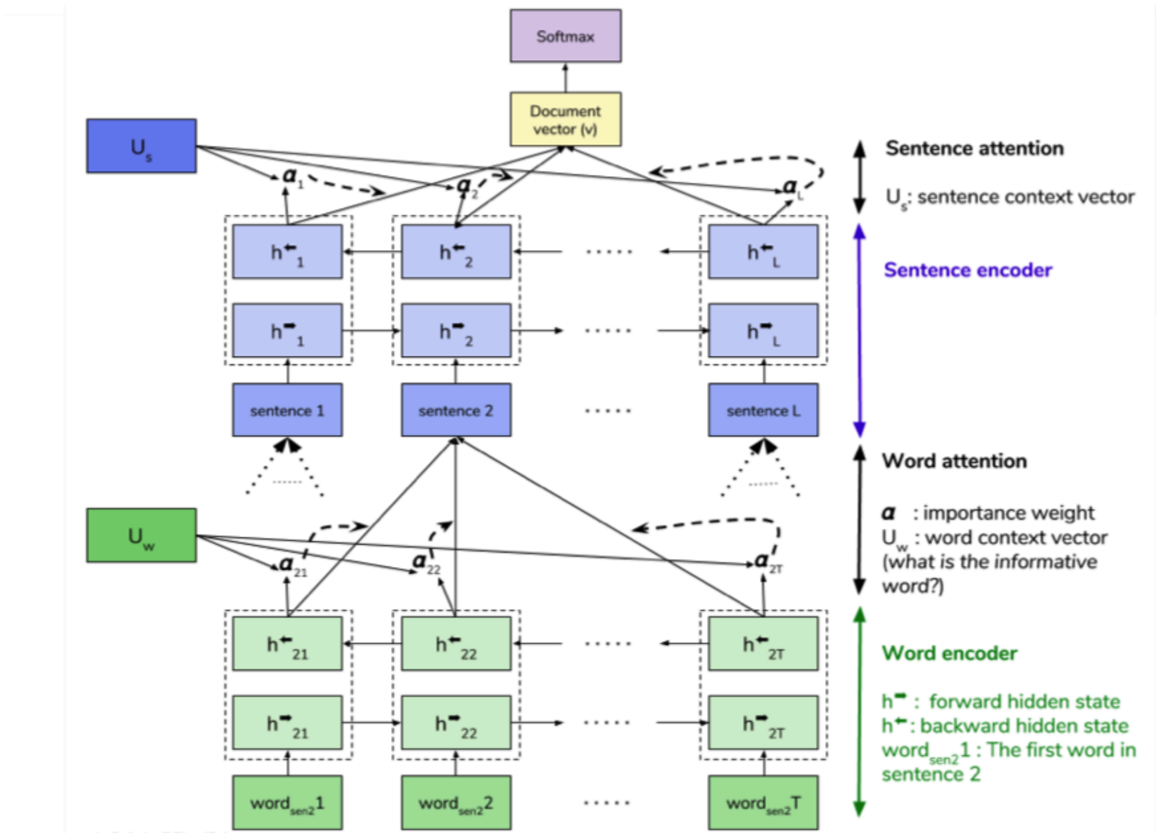
1. Hierarchical Attention Networks
2. XLNet
3. Flair

The reports of these other models will be found in the following pages.

Appendix A - Hierarchical Attention Networks (HAN)

Model Architecture

This section covers an overview of the chosen model architecture (HAN), according to the research paper here. Minor modifications were made to the model and will be described in the later section *Model Setup*.



Softmax
Document vector (v)
U_s
Sentence attention
U_s: sentence context vector
Sentence encoder
sentence 1
sentence 2
sentence L
Word attention
a : importance weight
U_w : word context vector (what is the informative word?)
U_w
Word encoder
h⃗ : forward hidden state
h⃖ : backward hidden state
word_sen2 1 : The first word in sentence 2
word_sen2 1
word_sen2 2
word_sen2 T

HAN is an extension of the existing RNN models, integrating both bidirectional RNN and Attention Networks. The underlying idea behind HAN is that documents have a hierarchical structure ("Words form sentences and sentences form documents"). By constructing HAN to follow this same representation, encoding the document on a word level with word attention, followed by the sentence level with sentence attention, HAN believes this will provide a better document embedding, which will improve the document classification process.

The model can thus be broken down into 3 main components:

1. Word Level Embedding
2. Sentence Level Embedding
3. Classification of Document Embedding

Word Level Embedding

The word level embedding aims to generate the sentence vector by embedding each word in the sentence. The word level embedding is made out of both a bidirectional RNN and a Additive

Attention Mechanism. The bidirectional RNN will embed every word in the sentence, providing context in both the forward and backward direction, and word attention is retrieved to highlight important words in the sentence.

After embedding each word in the sentence, the sentence embedding is retrieved as the weighted sum of the word embeddings * word attention.

Sentence Level Embedding

The sentence level embedding aims to generate the document vector by embedding each sentence vector (generated from word level embedding) in the document. The sentence level embedding component is also made out of both a bidirectional RNN and a Additive Attention Mechanism. Following the same concept as the word level embedding, the bidirectional RNN will embed each sentence vector in the document, incorporating context in both the forward and backward direction. Sentence attention is also retrieved to highlight important sentences in the document.

After embedding each sentence in the document, the document vector is retrieved as the weighted sum of the sentence embeddings * sentence attention.

Classification of Document Embedding

The document vector is a high level representation of the document and is thus used for document classification. The classification component is made out of linear layers, with a softmax activation function.

The model will be trained on negative log likelihood (Cross Entropy Loss).

Model Setup (Code Explanation)

The previous section provided a brief overview of the HAN model architecture, however, modifications were made to the model based on the use case. This section covers the general code architecture and the changes made to the HAN model for sentiment analysis.

Dataset and Custom Dataloader

The HAN model requires the following input:

1. The document
2. The label of the document (0 for negative, 1 for positive)
3. The number of sentences in the document
4. The number of words in each sentence in the document

A custom Dataset was thus written to create the training and testing dataset. Utilising pytorch inbuilt Dataset class, the **IMDBDataset** will read the text files from the data using the same read_imdb function from the final project notebook.

Upon reading each document, data preprocessing is done (self._transform_text()) to split each document into a nested list of sentences, where each sentence is also tokenized into a list of words. Each word is then tokenized using d2l.Vocab class on the train dataset, with a minimum frequency of 5 and '<pad>' as the reserved token.

The document will then be sliced to ensure that each sentence in the document does not exceed maximum sentence length (max word per sentence), and each document does not exceed maximum document length (max sentence per document).

Upon creating the Dataset, pytorch's DataLoader was used to create the train and testing iteration for training and evaluation. For the train_iter, shuffle was set to True to allow the shuffling of the training data.

An additional class padAnnotations was created to pad the documents, ensuring that each sentence in the document is maximum sentence length long and each document is maximum document length long. d2l.Vocab <pad> token was used for padding. The class was called in the DataLoader as the collate_fn to pad the dataset for uniform data.

The expected output for each iteration from train/test iter includes the following:

1. Document tensors: [batch_size, max_document_length, max_sentence_length]
2. Label tensors: [batch_size]
3. Document Lengths (before padding): [batch_size]
4. Sentence Lengths (before padding): [batch_size, max_document_length]

Modified Model Architecture

As HAN model has 3 components, 3 separate classes were coded (**WordAttention, SentenceAttention, HANModel**). I took reference from this github to develop my code.

WordAttention

The WordAttention class takes in the document tensors, document lengths and sentence lengths. As an RNN accepts variable input size, to optimise computation, we will use PyTorch inbuilt pack_padded_sequence to remove the padded words in the sentences.

Each word will then be embedded with **Glove.6b.100** embeddings (frozen), with a dropout of 0.2. Dropout was used to improve the robustness of the model and to prevent overfitting.

After embedding, each word will then be passed into a RNN to retrieve the word vector. In the original HAN paper, a bidirectional GRU was proposed. However, due to the long sentences in this particular IMDB dataset, the team proposed using a bidirectional LSTM instead, as LSTMs can maintain and propagate information over longer time lags than GRU. Both options were integrated into the model, with the default RNN being a LSTM.

After retrieving the word vectors from the LSTM, word attention was calculated.

$$\begin{aligned} u_i &= \tanh(W_w \cdot x_w) \\ u_w &= W_c \cdot u_i \\ \text{val} &= \max(u_w) \\ att &= \frac{\exp(u_w - \text{val})}{\sum \exp(u_w - \text{val})} \end{aligned}$$

W_w and W_c are 2 different linear layers that will be finetuned by the model. U_i was used to capture high level features, which were then passed through W_c to retrieve the attention scores of each word. The max score was used to stabilise the attention, preventing any potential overflow before softmax was applied to convert the attention score into probabilities.

After retrieving the word attention weights and the word vectors, they were multiplied and summed to obtain the sentence embedding.

SentenceAttention

SentenceAttention follows extremely closely to WordAttention, but instead of taking in document tensors, SentenceAttention takes in the sentence vector outputted by WordAttention. Following the same concept, PyTorch inbuilt pack_padded_sequence was used to remove the padded sentences in the document.

Each sentence embedding passes through a dropout of 0.2 before being encoded by a bidirectional LSTM to retrieve the sentence vector. Afterwhich, the same attention formula as the one in WordAttention was applied on the sentence vectors, just with a new W_s (replacing W_w) and W_c.

After retrieving the sentence attention weights and the sentence vectors, they were multiplied and summed to obtain the document embedding.

HANModel

HANModel is the overarching class, containing both WordAttention and SentenceAttention classes. The inputs of the train_iter will be passed to WordAttention to retrieve the sentence embeddings, which were then passed to Sentence Attention to retrieve the document vector.

Upon retrieving the document vector, the document vector will be passed through 1 linear layer and the final classification linear layer of output [batch_size, num_classes]. As we are only doing sentiment analysis, there are 2 classes (0 for negative, 1 for positive). Therefore, instead of softmax as the activation function, ReLU was used as the activation function for the binary classes.

The overarching HANModel architecture can be seen here for LSTM and GRU.

LSTM	GRU
HANModel((word_attention): WordAttention((embedding): Embedding(49347, 100) (encoder): LSTM(100, 128, batch_first=True, dropout=0.2, bidirectional=True) (dropout): Dropout(p=0.2, inplace=False) (word_weight): Linear(in_features=256, out_features=256, bias=True) (context_weight): Linear(in_features=256, out_features=1, bias=True)) (sentence_attention): SentenceAttention((encoder): LSTM(256, 128, batch_first=True, dropout=0.2, bidirectional=True) (dropout): Dropout(p=0.2, inplace=False) (sentence_weight): Linear(in_features=256, out_features=256, bias=True) (sentence_context_weight): Linear(in_features=256, out_features=1, bias=True)) (linear): Linear(in_features=256, out_features=50, bias=True) (fc): Linear(in_features=50, out_features=2, bias=True) (relu): ReLU())	HANModel((word_attention): WordAttention((embedding): Embedding(49347, 100) (encoder): GRU(100, 128, batch_first=True, dropout=0.2, bidirectional=True) (dropout): Dropout(p=0.2, inplace=False) (word_weight): Linear(in_features=256, out_features=256, bias=True) (context_weight): Linear(in_features=256, out_features=1, bias=True)) (sentence_attention): SentenceAttention((encoder): GRU(256, 128, batch_first=True, dropout=0.2, bidirectional=True) (dropout): Dropout(p=0.2, inplace=False) (sentence_weight): Linear(in_features=256, out_features=256, bias=True) (sentence_context_weight): Linear(in_features=256, out_features=1, bias=True)) (linear): Linear(in_features=256, out_features=50, bias=True) (fc): Linear(in_features=50, out_features=2, bias=True) (relu): ReLU())

Model Performance

The model was trained with the following parameters:

1. CrossEntropyLoss as the Criterion (supports binary classes)
2. Adam Optimiser
3. Maximum Document Length of 24 was used based on the dataset not having any documents longer than 24 sentences
4. 25 Epochs
5. Gradient Clip at 1

During training, it was noted that the model had a tendency to result in exploding gradients when the learning rate is high. Therefore, to ensure this does not happen, a gradient clip at 1 was set to stabilise the training.

Exploration was done mainly on the learning rate (LR), batch size and maximum sentence length. Despite removing padding for the RNN, some of the sentences are longer than 256 words, thus exploration was done if cutting the sentences short significantly impacted the model's performance.

None of the sentences in the dataset had lengths greater than 512, thus testing was done between the results of maximum sentence length of 256 and 512. All exploration and the results are found in the following table.

RNN	LR	LR Scheduler	Batch Size	Max Sent Len	Test IMDB Acc	Released Test Acc
LSTM	0.005	Linear, scale to 0.0005 in 25 iterations	32	256	0.90236	0.93950
LSTM	0.005	Linear, scale to 0.0005 in 25 iterations	64	256	0.90276	0.93603
LSTM	0.005	Linear, scale to 0.0005 in 25 iterations	32	512	0.90020	0.93895
LSTM	**0.005**	**Linear, scale to 0.0005 in 25 iterations**	**64**	**512**	**0.90308**	0.93275
LSTM	**0.005**	**None**	**64**	**512**	0.90200	**0.940525**
LSTM	0.01	Linear, scale to 0.001 in 25 iterations	64	512	0.90220	0.940250
LSTM	0.001	Linear, scale to 0.0001 in 25 iterations	64	512	0.88788	0.896425
LSTM	0.005	Linear, scale to 0.0005 in 25 iterations	128	512	0.90052	0.928975

GRU	0.005	Linear, scale to 0.0005 in 25 iterations	64	256	0.90024	0.93205
GRU	0.005	Linear, scale to 0.0005 in 25 iterations	64	512	0.89804	0.93285

Model Result

The best model was the LSTM with 0.005 learning rate, 64 batch size and 512 max sentence length. The model had an accuracy of 0.941 (3sf), recall of 0.944 (3sf), precision of 0.938 (3sf) and f1 score of 0.941 (3sf).

Performance Analysis on IMDB test dataset

All models performed similarly, with little significant change in their accuracy for the IMDB test dataset. This highlights that perhaps the model has reached the limit for the provided dataset and is unable to improve further.

If we were to compare, the model performed best when the maximum sentence length was 512 (where none of the sentences were cut off) and a batch size of 64 was used. However, due to how small the differences are, it is unconfirmed that batch size and maximum sentence length have any significant impact on the model performance, or if it was simply due to the random initialisation. Furthermore, higher batch sizes required a significantly larger training time and compute resources; thus, little exploration was done.

However, the model did perform best for a learning rate of 0.005, as 0.001 was too small to impact training, and 0.01 was too large. Furthermore, using a learning rate scheduler to reduce the learning rate as training continued also improved the model performance, as the model could finetune itself with smaller learning steps.

Lastly, LSTM models performed better than GRU models as suspected, as the longer reviews in the dataset would require a RNN that can maintain long term dependencies.

Future improvements of the model can look into using different embeddings in the WordAttention embedding layer, such as testing out other Glove Embeddings or even utilising Bert Embeddings.

Performance Analysis on released test dataset

All models performed slightly better on the new released test set, having a consistent increment by 0.3. The best model for the released test dataset is not the same as the IMDB test dataset, which highlights that having a learning rate scheduler has little impact on the model's accuracy.

Likewise, LSTM models still performed better than GRU models. Furthermore, due to an inconsistent trend, it is difficult to pinpoint if smaller batch sizes are better and further testing can be done in the future.

Appendix B - XLNet

Model Architecture

This section covers an overview of the XLNet model architecture that is introduced here.

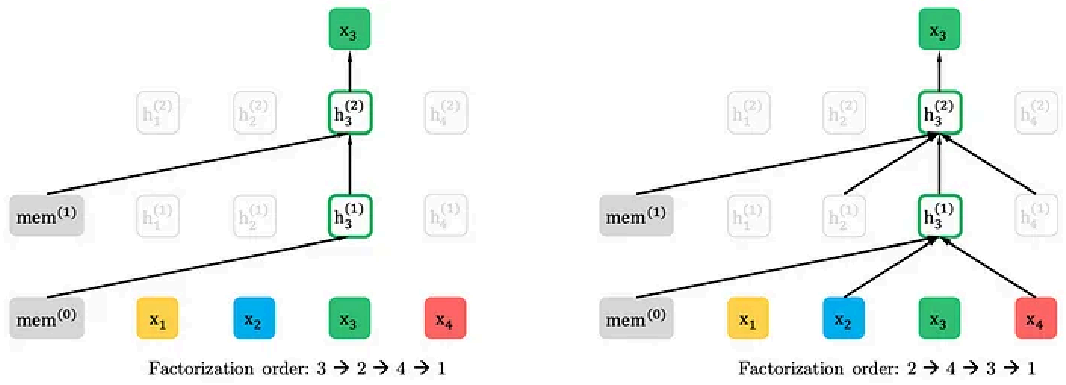

x_3
h_1^(2)
h_2^(2)
h_3^(2)
h_4^(2)
mem^(1)
h_1^(1)
h_2^(1)
h_3^(1)
h_4^(1)
mem^(0)
x_1
x_2
x_3
x_4
Factorization order: 3 → 2 → 4 → 1
Factorization order: 2 → 4 → 3 → 1
Factorization order: 1 → 4 → 2 → 3
Factorization order: 4 → 3 → 1 → 2

Figure 1: Illustration of the permutation language modeling objective for predicting x_3 given the same input sequence x but with different factorization orders.

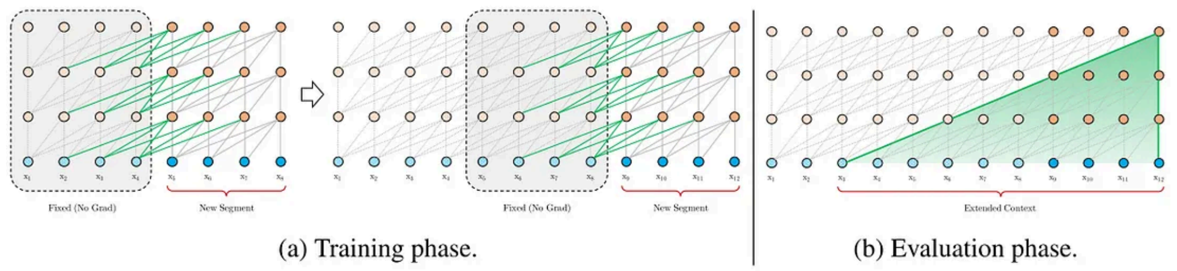
Overview

XLNet is a generalised autoregressive model that aims to overcome the disadvantages of BERT, mainly the neglect of dependency between masked positions when training. Instead of relying on masking to achieve bidirectionality in BERT, XLNet utilises **Permutation Language Modelling** to achieve bidirectional context instead.

Permutation Language Modelling can be defined as training an autoregressive model on all possible permutations of words in a sentence as seen in the above image. XLNet aims to maximise the expected log likelihood over all possible permutations of the sequence to output the best permutation of words (most likely sentence). Doing so, the model learns to utilise contextual information from all positions, capturing bidirectional context. This method does not require masking the input, and thus does not corrupt the input data unlike BERT.

Transformer-XL Architecture

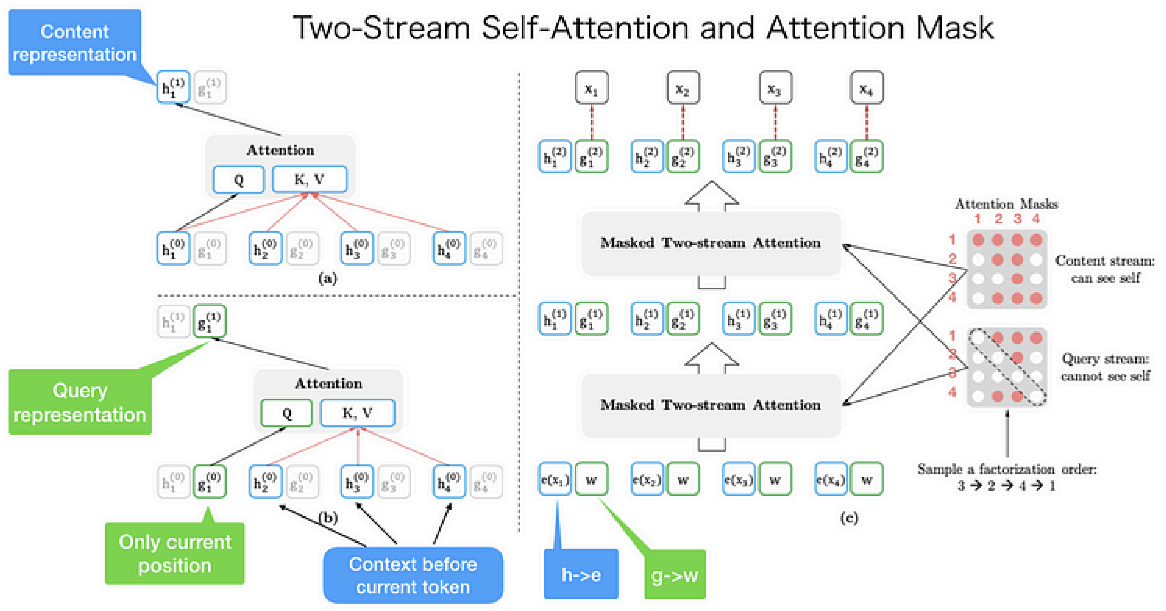
XLNet utilises Transformer-XL, a modification of the transformer model. Transformer-XL is a decoder-only model, with multi-head attention and feedforward linear layers, however, it also includes a segment-level recurrent mechanism and relative positional encoding.



The segment-level recurrent mechanism allows the transformer-xl to utilise previous hidden states with the existing hidden states. This allows the context of past training data to be carried over, removing the fixed length constraint of normal transformer models. The modified hidden state (the concatenated previous and current hidden states) will be used as the **Key** and **Value** for self attention, while the current hidden states will be used as the **Query**. This essentially effectively extends the context for the attention mechanism.

To avoid temporal confusion due to the presence of previous hidden states, relative positional encodings replaced transformers' absolute positional encodings. Relative positional encodings encode **positional information biases**, teaching the model the spatial relationship between tokens, rather than simply knowing the absolute position of the token. Relative positional encodings are calculated by the relative distance between each Key and Query vector.

XL-Net Architecture


Two-Stream Self-Attention and Attention Mask
Content representation
Attention
Q
K, V
(a)
Query representation
Attention
Q
K, V
(b)
Only current position
Context before current token
Masked Two-stream Attention
Masked Two-stream Attention
h->e
g->w
(c)
Attention Masks
Content stream: can see self
Query stream: cannot see self
Sample a factorization order: 3 → 2 → 4 → 1

To apply Permutation Language Modelling instead of BERT's bidirectional context masking onto the transformer-xl, attention masking is applied. When calculating attention, the Query will be the predicted word, and the Keys and Values will be the preceding words in the sequence. This ensures that attention is not applied to future tokens, and XLNet cannot 'cheat' but rather learns the contextual relationships through permutations.

However, the attention mask does not provide the content representation for prediction, requiring a second hidden state to provide the global content representation. Hence, a two-stream self attention is integrated into XL-Net, where the **content stream** acts similar to the hidden states in transformers, encoding all tokens, while the **query stream** will apply attention masking to only encode the preceding context.

The content stream thus captures the bidirectional information across permutations. improving the context building, while the *query stream* ensures predictions only rely on preceding tokens, enforcing the autoregressive nature of the model. The **content stream** will be used as the **Key** and **Value** for the attention, while the **query stream** provides the **Query** for prediction.

Model Setup (Code Explanation)

As XLNet is an extremely large model, training from scratch is computationally intensive and requires a large amount of compute resources. Thus, finetuning was done on a pre-trained XLNet model from Hugging Face.

Dataset and Custom Dataloader

XLNet requires the following token pattern:
Sentence_A + [SEP] + Sentence_B + [SEP] + [CLS]

However, utilising the pretrained XLNet tokenizer will automatically add the [SEP] and [CLS] tokens at the end of each data. Furthermore, as the IMDB dataset is expecting the entire review as a single input sequence (and thus a single label), adding [SEP] within each sentence can be considered unnecessary as XLNet will be assuming the entire review as a single sequence.

Therefore, a custom Dataset was written to create the training and testing data. Utilising pytorch inbuilt Dataset class, the **XLimdbDataset** will read the text files from the data using the same read_imdb function from the final project notebook. Afterwhich, each data will be tokenised by the pretrained XLNet tokeniser from hugging face and padded to the defined maximum token length. The vocabulary used will be XLNet's tokenizer in-built vocabulary.

The Dataset is then loaded into pytorch Dataloader class, where the train dataset will be shuffled, while the test dataset will not.

The expected output for each iteration from train/test iter includes the following:

1. Document Input Ids (tokenised document): [batch_size, max_token _length]
2. Document Attention Masks: [batch_size, max_token_length]
3. Document Token Type Ids: [batch_size, max_token_length]
4. Label tensors: [batch_size]

Modified Model Architecture

A pre-trained XLNet model from hugging face is retrieved, with the weights of the XLNet base model frozen.

However, the weights and biases on the linear layers after the transformer-xl model that are used to classify the document embeddings are randomly intialised and will be fine-tuned. The fine-tuning process will thus be on the **sequence_summary.summary** and the **logits_proj**.

The overarching structure can be seen in the following, where the bolded layers are the ones that will be fine-tuned:

```
XLNetFineTunedModel(
  (model): XLNetForSequenceClassification(
    (transformer): XLNetModel(
      (word_embedding): Embedding(32000, 768)
      (layer): ModuleList(
        (0-11): 12 x XLNetLayer(
          (rel_attn): XLNetRelativeAttention(
            (layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (ff): XLNetFeedForward(
            (layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (layer_1): Linear(in_features=768, out_features=3072, bias=True)
            (layer_2): Linear(in_features=3072, out_features=768, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
            (activation_function): GELUActivation()
          )
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (sequence_summary): SequenceSummary(
      (summary): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
      (first_dropout): Identity()
      (last_dropout): Dropout(p=0.1, inplace=False)
    )
    (logits_proj): Linear(in_features=768, out_features=2, bias=True)
  )
)
```

Model Performance

The model was trained with the following parameters:

1. BinaryCrossEntropyLoss that is inbuilt inside the Hugging Face XLNet model
2. AdamW Optimiser with weight decay of 0.01
3. 10 Epochs

Due to the extremely long training time, only 10 epochs were used to finetune the model.

Exploration was done mainly on the learning rate (LR), batch size and maximum token length. Learning rates of 1e-5, 2e-5, 5e-5 and 5e-4 were explored, as they are the most commonly used learning rates for transformer models.

Furthermore, due to computational limitation, only batch size of 32 and 64 were explored, with batch size of 128 being too computational intensive.

Examining the dataset also highlighted that the majority of the data had token lengths larger than 256 but smaller than 512, with some exceptions having token lengths larger than 1000. Thus, exploration was also done on maximum token length, with only the best configuration being tested on a token length of 1024 due to the computational intensiveness.

The performance of different exploration can be seen below.

LR	LR Scheduler	Batch Size	Max Token Len	Test IMDB Acc	Released Test Acc
2e-5	Linear, scale to 2e-6 in 10 iterations	32	256	0.85752	0.858025
2e-5	Linear, scale to 2e-6 in 10 iterations	64	256	0.85064	0.851500
5e-4	Linear, scale to 5e-5 in 10 iterations	32	256	0.87944	0.881200
5e-5	Linear, scale to 5e-6 in 10 iterations	32	256	0.87308	0.874100
1e-5	Linear, scale to 1e-6 in 10 iterations	32	256	0.84432	0.844625
5e-4	None	32	256	0.88060	0.884050
5e-5	None	32	256	0.87952	0.881350
5e-4	None	32	512	0.90520	0.909325
5e-5	None	32	512	0.90164	0.903100
5e-4	**None**	**32**	**1024**	**0.91276**	**0.914000**

Model Result

The best model was XLNet trained on 1024 tokens. The model had an accuracy of 0.914 (3sf), recall of 0.921 (3sf), precision of 0.908 (3sf) and f1 score of 0.915 (3sf).

Performance Analysis on IMDB test dataset

The best model for the IMDB test dataset was the model trained on 1024 tokens. This is expected, as the model was receiving the full sentences in the dataset.

Most of the exploration was done on the model with 256 token length as it has the fastest training timing. Due to the similarity in accuracy for 5e-4 and 5e-5 learning rates, we tried both for the dataset with 512 tokens, where we noted that 5e-4 was the best learning rate.

The ideal configuration was thus determined to be 5e-4 with no learning rate scheduler and a batch size of 32. This configuration was then used to explore the fine-tuning of XLNet with different token lengths. It is important to note that the ideal configuration for the dataset with 256 token length may not be the same as 512 or 1024 tokens. However, due to the computational resources and time taken, we decided to assume that the ideal configuration is consistent.

Future works thus should consider exploring the different learning rates and batch sizes for the 512 and 1024 token lengths.

Performance Analysis on released test dataset

Generally, the model performed similarly on both the IMDB test dataset and the newly released dataset, with the accuracy for the newly released dataset to be slightly higher. Therefore, the best model on the released test dataset was the same as the IMDB test dataset.

The consistent results also showed that the model was able to generalise across unseen data, highlighting the power of transformer models.

Appendix C - Flair

Model Architecture

This section delves into the exploration of using Flair (link) for sentiment analysis with various embedding strategies for fine-tuning a classification model.

Flair Framework

Flair is a powerful natural language processing (NLP) library built on PyTorch, designed for tasks such as named entity recognition (NER), part-of-speech tagging (PoS), and sentiment analysis. Its modular design allows integration of diverse embeddings and models to address the wide array of NLP tasks.

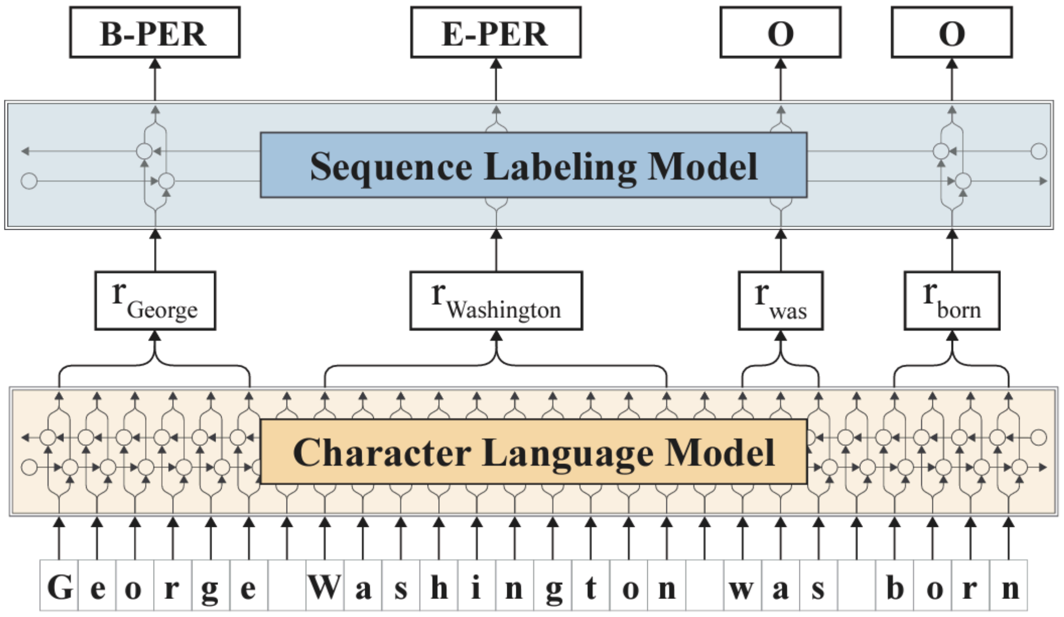


Figure 1. Overview of Contextual Embeddings with BiLSTM-CRF

As illustrated in Figure 1, Flair contains 2 key components:

1. Input embedding (Flair Embeddings) through the use of a Character Language Model
2. Classification through the use of a Sequence Labeling Model

Each segment will be discussed in detail in the following sections.

Flair Embeddings

Flair embeddings (link) are the contextual text embeddings derived from character-level language models. By employing bidirectional LSTMs, Flair embeddings contextualise words based on their surrounding text, effectively capturing semantic and syntactic nuances.

For example, as shown in Figure 2, the embedding for "Washington" differs based on its context, distinguishing between its usage as a location or a person's name. This dual modelling allows Flair embeddings to adapt dynamically to linguistic variations.

The Key Characteristics of Flair Embeddings can be broken down into the following:

1. *Bi-directional Character-level LSTMs*: The forward model predicts the next character in a sequence, while the backward model predicts the preceding character.
2. *Hidden State Extraction*: Outputs from these models are concatenated to produce contextualised word embeddings.
3. *Robustness*: Character-level approach ensures that the embeddings are able to handle rare, misspelled or out-of-vocabulary (OOV) words, making them suitable for morphologically rich languages and diverse datasets.

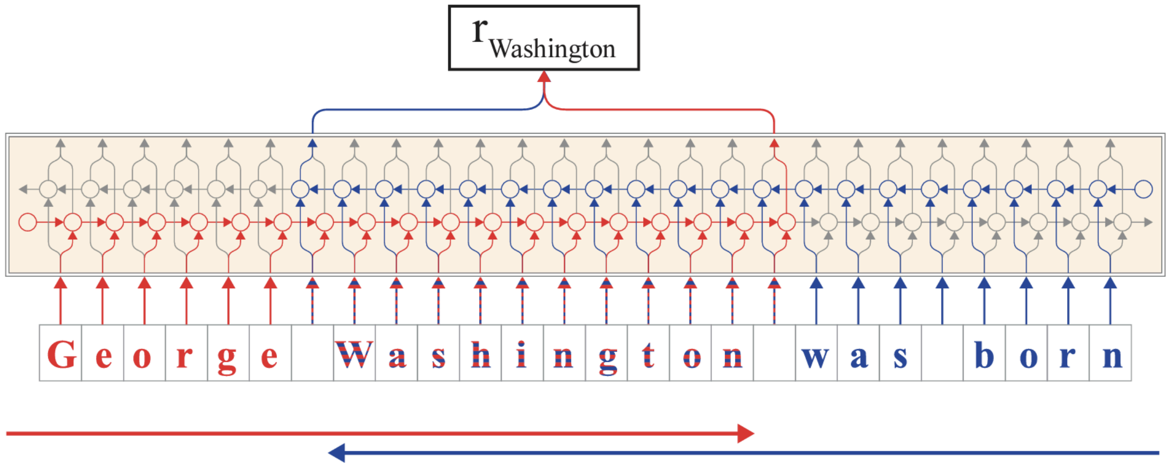


Figure 2. Bi-directional Character-level LSTMs

Stacked Embeddings

Other than relying on Flair to generate the embeddings, an alternative approach is to use Stacked embeddings. Stacked embeddings combine multiple embedding types from Flair and transformer models to leverage their complementary strengths. For instance, Flair embeddings capture contextual character-level information, while transformer embeddings such as RoBERTa or DistilBERT encapsulate sentence-level semantics.

Transformer Embeddings

Transformer-based embeddings leverage transformer architectures, such as DistilBERT and RoBERTA, to generate word-level embeddings. Unlike Flair embeddings, which focus on character-level modelling, transformer embeddings specialise in sentence-level contextualisation.

Two versions were used in our exploration:

1. DistilBERT: A lighter version of BERT, optimised for speed and memory efficiency with minimal performance trade-offs.
2. RoBERTA: A robustly optimised version of BERT, finetuned on a larger corpus and with improved hyperparameter settings.

Transformer embeddings represent each word based on its context within the entire sentence, making them powerful for understanding semantic nuances.

Stacked Embedding Architecture

Utilising both Flair Embeddings and Transformer Embeddings, the following architecture was then used to create the stacked embedding:

1. *Input*: Token embeddings (e.g., Flair forward/backward, transformer embeddings).
2. *RNN Layer*: Processes token embeddings sequentially to create a document representation.
3. *Reprojection Layer (Linear layer)*: Optionally reduces embedding dimension to prevent computational bottlenecks employing PyTorch linear transformation. The PyTorch Linear applies affine linear transformations to the document embedding given using the formula:

$$y = xA^T + b$$

where x is the input embedding, A is the weight matrix, and b is the bias vector.

In our implementation, we used Flair's `DocumentRNNEmbeddings` to wrap the stacked embeddings by aggregating token embeddings into fixed-length document vectors. This process relies on an RNN (typically BiLSTM) to encode sequential dependencies, and the final hidden state is used as the document embedding.

Classifier Architecture

After embedding the input and retrieving the document representation, the classifier handles the downstream NLP tasks. For Named Entity Recognition, the sequence labelling model used is a BiLSTM-Conditional Random Field (CRF). However, as we are implementing a sentiment analysis model, a linear layer was used instead to classify the embeddings into its outputs.

This architecture is implemented using Flair's TextClassifier, which applies a linear layer on top of the document embeddings produced by the embedding layer. For single-label classification tasks like sentiment analysis, the linear layer outputs raw logits, which are unnormalised scores representing the model's confidence for each class. These logits are then converted into probabilities during loss computation through the Pytorch CrossEntropyLoss function.

The `CrossEntropyLoss` function combines two critical steps:

1. Apply the softmax function internally to convert the raw logits into probabilities. The softmax ensures that the probabilities across all classes sum to 1, enabling them to be interpretable as a distribution over classes.

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

2. Thereafter, it calculates the negative log-likelihood of the true class's probability. This penalises the model more heavily for confident but incorrect predictions.

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n} \cdot 1\{y_n \neq \text{ignore\_index}\}} l_n, & \text{if reduction} = \text{'mean'}; \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{'sum'}. \end{cases}$$

Overarching Architecture

The overarching architecture thus comprises:

1. Embedding Layer: Generates token or document-level embeddings using Flair, transformers, or stacked embeddings.
2. RNN Layer (optional): Sequentially processes embeddings in DocumentRNNEmbeddings to create a document vector.
3. Linear Layer: Maps the document vector to class probabilities using a PyTorch linear transformation; the softmax activation is subsequently applied to compute sentiment class probabilities.

The final architecture depends on the choice of embeddings. For example:

- Flair-only embeddings: Word embeddings → RNN → Linear Layer
- Transformer embeddings: Sentence embeddings → Linear Layer
- Stacked embeddings: Combined word embeddings → RNN → Linear Layer

Model Setup (Code Explanation)

Dataset

We used the **IMDB dataset**, modified to align with Flair's required input format. Each sentence was prefixed with the label (__label__POS or __label__NEG). This format ensures compatibility with Flair's ClassificationCorpus class, which splits the dataset into training, validation, and test sets. From the training set, 10% of the original data was allocated to the validation set. This split was chosen to strike a balance between retaining sufficient data for training and retaining enough data for evaluation at each epoch. Additionally, a smaller validation set helps reduce computational overhead during training and evaluation, particularly given the already long runtime required for the models.

The validation set played a critical role in monitoring model performance throughout the training process. It enabled early stopping to prevent overfitting—whereby a degradation in validation performance, despite continued improvement on the training set, served as an indication of overfitting. By halting training at this stage, the model's generalisation capability was preserved. Lastly, a label dictionary map was computed to ensure all possible class labels (NEG and POS) were correctly represented in the training pipeline.

Expected input format:

```
__label__POS A woman who hates cats (Alice Krige) and her son…

__label__NEG Beast Wars is a show that is over-hyped, overpraised and…
```

TextClassifier

The TextClassifier is the core architecture used for classification tasks in Flair. It combines the embeddings generated by the embedding layer with an RNN (such as BiLSTM) or transformer architecture to produce a document-level representation.

```
classifier = TextClassifier(

    embeddings=document_embeddings,

    label_dictionary=label_dict,

    label_type="sentiment"

)
```

In the above code snippet, the document_embeddings refers to the various combinations of document embeddings created using Flair and transformer embeddings. The label dictionary maps sentiment labels to numerical indices, while label_type aims to specify the task's output labels.

Model Performance

Finetuning the sentiment analysis classifier was conducted using Flair's ModelTrainer, a versatile utility for orchestrating training, validation, and logging processes. Key hyperparameters were carefully selected to optimise the training process within the constraints of computational efficiency and memory limitations:

1. *Learning Rate:* A learning rate of 5e-5 was chosen to enable effective finetuning of pre-trained embeddings. This value helps balance between adapting to the new task and retaining the pre-trained knowledge, avoiding issues like catastrophic forgetting, where previously acquired knowledge is overwritten during the finetuning process.
2. *Batch Size*: A small batch size of 4, balancing computational demands against the capacity of the hardware used for training.
3. *Optimiser:* The AdamW optimiser was employed for its efficiency in handling pre-trained transformer embeddings, with weight decay minimising overfitting.
4. *Loss Function:* Cross-entropy loss was used

The maximum number of epochs varied across different configurations, reflecting the trade-offs between computational efficiency and model performance:

- For *Stacked Embeddings (Flair and RoBERTa-large)*, only 2 epochs were completed, as each epoch exceeded 30 hours of GPU computation. Early results showed underwhelming accuracy (~0.5 after the first epoch), leading to the decision to halt further training and explore alternative approaches.

- For *Stacked Embeddings (Flair and DistilBERT)*, despite requiring 3 days to train over 10 epochs, results were promising, warranting the full training schedule.

- For *Flair embeddings*, computational demands were similarly high, and training was limited to 5 epochs, given its comparatively modest performance.

- For *RoBERTa-large embeddings*, test accuracy stabilised at 5 epochs. Hence, training was terminated early to minimise unnecessary computational expense.

We examine 4 embedding configurations:

- Stacked Embeddings using DocumentRNNEmbeddings
    - Flair and DistilBERT
    - Flair and RoBERTa-large
- Flair embeddings
- RoBERTa-large embeddings

For RoBERTa-large embeddings, the team acknowledges that Flair was not involved and we are simply finetuning RoBERTA. These configurations were designed to assess the impact of combining Flair embeddings with transformers and to benchmark the performance of transformers (RoBERTa) against Flair.

The following table summarises the results of different configurations on the IMDB test set and additional test set:

Index	Embedding Configurations	Number of Epochs	Test IMDB Accuracy	Released Test Accuracy
1	Stacked (Flair and DistilBERT)	10	0.9290	0.9610
2	Stacked (Flair and RoBERTa-large)	2	0.7372	0.7409
3	Flair	5	0.8182	0.8219
4	RoBERTa-large	5	0.9432	0.9674

Model Result

The best model was RoBERTa-large, with an accuracy of 0.967 (3sf), precision of 0.966 (3sf), recall of 0.969 (3sf), and f1 score of 0.967 (3sf).

Performance Analysis on IMDB test dataset

The experiments highlighted the superior performance of RoBERTa-large embeddings, which achieved the highest test accuracy among all configurations. This result underscores the power of transformers for sentiment analysis.

Stacked embeddings demonstrated incremental improvements over Flair-only embeddings, leveraging the complementary strengths of Flair's contextual string embeddings and transformers. However, these gains were offset by significantly prolonged training durations, especially for large datasets like IMDB. For instance, training with Stacked (Flair and RoBERTa-large) embeddings proved computationally prohibitive, requiring over 30 hours per epoch with suboptimal performance.

While Flair's embeddings offered a lightweight and robust approach, their performance fell short when compared to transformer embeddings. The findings suggest that Flair may not be the most effective architecture for sentiment analysis, particularly when higher-performance transformer models like RoBERTa are readily available.

Future explorations will focus on optimising parameters for various embedding combinations, a process that is anticipated to require substantial time due to the long computational hours involved for each configuration. While we recognise that different embedding combinations could benefit from unique optimisations of key parameters, this was not fully explored in the current study. The extensive computational demands necessitated prioritising broader comparisons, balancing depth of analysis with the practical limitations of time and resources.

Performance Analysis on released test dataset

The trends observed on the IMDB test set were largely mirrored on the released dataset, with a proportionate increase in accuracy across configurations. This consistency indicates that the models generalise effectively to new data. As on the IMDB test set, RoBERTa-large embeddings outperformed all other configurations, achieving an impressive accuracy of 0.9674 on the released test dataset.

Stacked (Flair and DistilBERT) embeddings also performed commendably, reaching a released test accuracy of 0.9610—only marginally trailing RoBERTa-large and surpassing Flair alone. However, the significant computational costs associated with stacked embeddings remained a notable drawback.

Flair embeddings alone demonstrated stable performance across both datasets, achieving a released test accuracy of 0.8219. This highlights Flair's capacity to generalise to unseen data but underscores its limitations in sentiment analysis tasks where a deeper contextual understanding, as provided by transformer models, proves critical.

These results reinforce the transformative impact of RoBERTa-large, both as a standalone embedding and as a benchmark for future comparisons. While Flair offers a computationally efficient alternative, transformer models like RoBERTa-large continue to set the standard for high-performance sentiment analysis.