# IS 584: Deep Learning for Text Analytics
# Assignment 2

**Due by 11.05.2025**

In this assignment you will add a custom masked language head to the base BERT model for predicting masked words in The Hunger Games by Suzanne Collins.

To load the BERT model architecture and pre-trained weights you can use the Transformers library from HuggingFace:

```
!pip install transformers
from transformers import BertModel
model = BertModel.from_pretrained('bert-base-uncased')
```

But you are **NOT** allowed to import the `BertForMaskedLM` class. Instead, you will add a custom masked language head at the end of `BertModel`.

Main tasks are outlined here:

1. Prepare the inputs and initiate the masked language model
2. Complete the psuedo-perplexity calculation code
3. Train the model and evaluate it with the validation set and custom examples
4. Use the model to generate new tokens, mimicking generative models

## Disclaimer

This is an individual assignment. Please refrain from collaboration. Upon any hurdle, you can contact your TA. It is important you adhere to academic integrity principles.

LLM's can only be used to paraphrase your writing or to help diagnose coding errors. You have to make decisions, write the code, execute it and report the outcomes yourself.

## Deliverables

- An ipython notebook with versions of major packages used (pytorch, torchtext etc.), cell outputs and random seeds visible.
- A pdf report. This should include high resolution exports (dpi $\geq$ 200) of any graphic and make comparisons using tables. Besides visual elements, provide all commentary on the output in the report.

## Late Submissions

Late submissions will be accepted until 13.05.2025 with a 10% grade reduction each day. (For a total of 20% at maximum)

## 1. Data Preprocessing (10p)

Download the dataset from the attached .txt file and preprocess the text. Report on the dimensionality and quality of the data. (Lengths, unique values, distributions etc.)

Look into the BERT model configuration to find it's maximum sequence length, and decide on a strategy for using short and long sentences in the book. Initiate the pre-trained BertTokenizer with the help of huggingface and use it to tokenize each sentence and pad the rest of the tokens until maximum sequence length.

## 2. Preparation (15p)

Set a seed in PyTorch. Make sure to document what that seed is so your analysis is reproducible. Separate the preprocessed data into training-validation-testing splits with 0.7-0.15-0.15 ratios.

You can use this class to initiate your dataloader if you want to keep the dictionary structure returned from the tokenizer:

```
from torch.utils.data import Dataset

class MyDataset(Dataset):
    def __init__(self, data_dict):
        self.data_dict = data_dict

    def __len__(self):
        return self.data_dict[list(self.data_dict.keys())[0]].shape[0]

    def __getitem__(self, idx):
        ''' Retrieves a single sample from the dataset and returns a dictionary.'''
        sample = {}
        for key, tensor in self.data_dict.items():
            sample[key] = tensor[idx,:]
        return sample
```

For each sentence, randomly select 20% of the tokens and replace them with the index of the [MASK] token. Be sure to not replace special tokens like [PAD], [CLS], and [SEP].

Masked version will serve as your inputs, while the original indices will be your labels. Finally, replace all words that weren't masked in the labels with -100. You will ignore this index in the loss function.

## 3. Masked Language Head (25p)

Define a separate neural network class that will take 768 dimensional embeddings from BertModel and for each word, including the masked ones, will output logits corresponding to each word in the vocabulary.

Make sure to have more than one hidden layer, utilizing an activation function and layer normalization in-between. Test the model altogether by passing in a random tokenized sentence to the model. The output should be: $B\ x\ L\ x\ V$ where $B =$ batch size, $L =$ length of text and $V =$ vocabulary size.

## 4. Metrics and Training (35p)

Define a loss function and optimizer. Initiate the learning rate at $10^{-4}$. Fill in the blanks of the function in the appendix on page 3 to calculate a psuedo-perplexity measure based on the probabilities generated by the model on texts in the validation set.

Calculate perplexity values for the original sentences (not the ones with words replaced with -100) in the validation and test set before fine tuning.

Train your neural network for at least 10 epochs. Monitor under or overfitting with loss and perplexity values on the training and validation sets. Report these values with graphs. Highlight when the model starts to overfit the data. Generate predictions for masked words in the test set and report the observed perplexities afterwards.

Then, come up with three sentences. These should not be domain specific, but rather reflect day-to-day exchanges. Mask out words at your own discretion, focusing on verbs or nouns that are crucial to the meaning of the sentence. An example is given below:

> *I would've appreciated some [MASK] given our [MASK].*

Use your fine-tuned model to make predictions for the masks in these sentences and comment on if they make sense. For at least one sentence, pass the outputs through the softmax activation and report on the top 5 words that were most likely to be predicted.

## 5. Masked Language Generation (15p)

The purpose of this section is to showcase how transformer based encoders perform in language generation, a task typically done with the help of, or sole usage of a transformer based decoder architecture.

Come up with a string between 20-30 tokens. It should end with the masked word token. This string should act like a prompt given to an LLM.

In an iterative fashion, do the following for 10 iterations:

1. Generate predictions for the masked word with the trained model.
2. Sample from the top k=20 words and replace the masked word with the prediction.
3. Append the masked word token at the end. (`sentence += " [MASK]"`)
4. Go to step 1.

Based on your knowledge and findings, what makes this version of BERT fundamentally different than generative language models that did not allow it to generate fluid text?

## Appendix

This code block was adapted from this file from the Language Model Perplexity (LM-PPL) GitHub repository. Since perplexity is originally defined for decoder only transformers where it is calculated over the last token given the rest, a custom implementation is needed for MLM.

Here, for each input, each location in the text is replaced with the masked word token, predicted, and losses over all those versions are summed only for those points. Do keep in mind that special tokens are also replaced here. It is not fixed in this version, but you can adapt it to make better conclusions.

Portions of missing code are at the very end marked by `'FILL HERE'`

```python
import math
from itertools import chain

def get_perplexity(model, device: str, input_texts: str or List, batch_size: int = None):
    """ Compute the perplexity on MLM.
    :model: PyTorch based MLM
    :device: A string. Valid values: cpu, cuda, mps
    :param input_texts: A string or list of input texts for the encoder.
    :param batch_size: Batch size
    :return: List of perplexity values.
    """
    tokens = tokenizer.tokenize('get tokenizer specific prefix')
    tokens_encode = tokenizer.convert_ids_to_tokens(tokenizer.encode('get tokenizer specific prefix'))
    sp_token_prefix = tokens_encode[:tokens_encode.index(tokens[0])]
    sp_token_suffix = tokens_encode[tokens_encode.index(tokens[-1]) + 1:]

    single_input = type(input_texts) == str
    input_texts = [input_texts] if single_input else input_texts

    def get_partition(_list):
        length = list(map(lambda o: len(o), _list))
        return list(map(lambda o: [sum(length[:o]), sum(length[:o + 1])], range(len(length))))

    # data preprocessing
    data = []
    for x in input_texts:
        x = tokenizer.tokenize(x)

        def encode_mask(mask_position: int):
            _x = x.copy()
            # get the token id of the correct token
            masked_token_id = tokenizer.convert_tokens_to_ids(_x[mask_position])
            # mask the token position
            _x[mask_position] = tokenizer.mask_token
            # convert into a sentence
            _sentence = tokenizer.convert_tokens_to_string(_x)
            # encode
            tokens_encode[:tokens_encode.index(tokens[0])]
            _e = tokenizer(
                _sentence, max_length=100, truncation=True, padding='max_length', return_tensors='pt')
            # add the correct token id as the label
            label = [-100] * _e['input_ids'].shape[1]
            label[mask_position + len(sp_token_prefix)] = masked_token_id
            _e['labels'] = torch.tensor([label], dtype=torch.long)
            return _e

        data.append([encode_mask(i) for i in range(min(100 - len(sp_token_prefix), len(x)))])
    # get partition
    partition = get_partition(data)
    data = list(chain(*data))
    # batch preparation
    batch_size = len(data) if batch_size is None else batch_size
    batch_id = list(range(0, len(data), batch_size)) + [len(data)]
    batch_id = list(zip(batch_id[:-1], batch_id[1:]))

    ce_loss = torch.nn.CrossEntropyLoss(
        reduction='none',
        ignore_index=-100
    )
    # run model
    nll = []
```

```
    with torch.no_grad():
        for s, e in batch_id:
            batch = data[s:e]
            inputs = {k: torch.cat([ex[k] for ex in batch], dim=0).to(device)
                      for k in batch[0].keys()}
            labels = inputs.pop('labels')
            # get an inference from the model
            logits = model('FILL HERE')

            # flatten to [B*L, V] and [B*L]
            B, L, V = logits.shape
            # calculate the loss with logits and labels
            loss_flat = ce_loss('FILL HERE')
            # reshape to [B, L]
            loss_per_token = 'FILL HERE'
            # sum over tokens to get per-example NLL
            nll_batch = 'FILL HERE'  # shape [B]

            nll.extend(nll_batch.detach().cpu().tolist())

    # reconstruct the nested structure
    ppl = [math.exp(sum(nll[i:j])/(j-i)) for (i,j) in partition]
    return ppl
```