INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Masters's Thesis

Reducing Computational Cost of Multilinguality in LLMs via Universal Subword Embeddings

Tanvin Sharma

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Masters's Thesis in Informatics

Reducing Computational Cost of Multilinguality in LLMs via Universal Subword Embeddings

Reduzierung des Rechenaufwands von Mehrsprachigkeit in LLMs mittels Universeller Subword Embeddings

Author:	Tanvin Sharma
Supervisor:	Apl. Prof. Dr. Georg Groh
Advisor:	Jeremias Bohn
Submission Date:	08.10.2024

I confirm that this masters's thesis is my own work and I have documented all sources and material used.

Munich, 08.10.2024 Tanvin Sharma

Acknowledgments

I would like to express my gratitude to my supervisor, Jeremias Born, for his strong support throughout this thesis. I had the pleasure of working with him for over a year now, including a practical project and this thesis, and his support and guidance is the reason I was motivated throughout this project to work harder. I am also thankful for the discussions we had throughout the duration of this project. It was not only a learning experience but also a fun way to spend Wednesday afternoons, where we would tackle our problems with innovative ideas.

I would like to thank Professor Groh, my examiner, for instilling the interest of NLP in me. I had only messed around a bit with NLP, that too on a very low level, till I went to his lecture in my first semester. The lecture was good enough reason to pursue NLP for the rest of my Master's degree, in the form of various projects, another subject, a seminar, and finally, this thesis project.

I would also like to thank the Technical University of Munich for granting me the opportunity to pursue this journey over the course of two years.

Abstract

Natural Language Processing (NLP) has evolved in recent years due to the advancement of Transformers trained on large amounts of data. These Transformers based models called Large Language Models (LLMs) have revolutionized this field showing exceptional versatility and efficiency across diverse tasks. However, a key factor contributing, and also inhibiting, to NLP research is the computational resource limitations and substantial environment footprint.

This thesis explores a novel approach to reduce the computational cost of multilinguality in LLMs by using universal subword embeddings. Focussing on Hindi and English as two example languages, this work introduces a custom architecture that combines a curated Hindi-English dictionary and a BERT model fine-tuned using Low-Rank Adaptation (LoRA). The proposed method aims to preserve morphological information while efficiently adapting pre-trained English language models to Hindi. Experiments demonstrate the performance of our model on various NLP downstream tasks from the IndicGLUE benchmark.

This approach shows a promising direction for bridging the gap between different languages in NLP, offering a flexible framework that could potentially extend to multiple languages with minimal computational overhead.

Contents

1 Introduction

The Transformer architecture (Vaswani, Shazeer, Parmar, et al. 2017) has become the standard architecture for LLMs. Based on them, models such as BERT (Devlin, Chang, Lee, and Toutanova 2019), GPT (Brown, Mann, Ryder, et al. 2020) and many others have shown their ability to learn information about a language and task it was trained on. However, the majority of these models are primarily trained on English or a handful of high-resource languages, leaving a significant gap in their ability to handle the world's linguistic diversity.

Multilinguality in LLMs is a critical area of research, aiming to create models that can understand and generate text in multiple languages. Multilingual models like mBERT (Devlin, Chang, Lee, and Toutanova 2019), T5 (Raffel, Shazeer, Roberts, et al. 2019), and other multilingual models have demonstrated the ability of these models and also the potential in this field. Nonetheless, the conventional approach of training separate models for each language or using massive multilingual models is computationally expensive and resource-intensive. Our goal in this work is to devise an approach which can eliminate this problem using morphological subword embeddings. Previous work (Chimalamarri, Sitaram, and Jain 2020) has shown how embeddings trained on subword representations yield better task performance than those trained on whole words. Classical subword tokenization algorithms like Byte-Pair Encoding (BPE) (Sennrich, Haddow, and Birch 2016) and Wordpiece (Wu, Schuster, Z. Chen, et al. 2016) provide the advantage of having a fixed size vocabulary by splitting them into smaller units, but they tend to disregard morphological properties of the language it is dealing with (Hofmann, Pierrehumbert, and Schütze 2021).

Our work provides an approach to reduce the computational cost of multilinguality in LLMs while preserving morphological information, focussing on Hindi and English. The core idea is to leverage morphologically informed embeddings that can capture the semantic and syntactic features across languages. The proposed method introduces the following key elements:

- Custom Hindi-English dictionary: A comprehensive dictionary is created by compiling various sources. This dictionary serves as the foundation for our custom tokenization process.
- Morphological tokenization: Instead of relying on subword tokenization algo-

rithms, we employ a morphological tokenizer from the IndicNLP library (Kunchukuttan 2020).

- Embedding bridge: To address the vocabulary mismatch between the custom tokenizer and the pre-trained BERT embeddings, an embedding bride is implemented with the goal to learn custom embeddings to a space compatible with BERT's subsequent layers.
- LoRA: We use LoRA (Hu, Shen, Wallis, et al. 2021) to fine-tune the BERT model efficiently with a minimal increase in parameters.

We conduct various experiments to optimize various aspects of the proposed architecture, including hyperparameter tuning, comparison of tokenization strategies and exploration of different LoRA configurations. The resulting models are evaluated on multiple tasks from the IndicGLUE benchmark, providing insights into their performance across different aspects of language understanding. This research contributes to the field of multilingual NLP by offering an efficient approach to adapting pre-trained English language models to morphologically rich languages. This method has the potential to be extended to other language pairs, providing a flexible framework for learning embedding representations over multiple languages with merely a bilingual dictionary, a dataset and some LoRA adapters.

The following chapters delve into the background of neural networks, strategies we implement to train them efficiently, transformers and multilingual NLP. We then move to our architecture and explain briefly how individual components work. We also demonstrate experiments to show why we chose certain parameters for our components in our current architecture. Further experiments demonstrate interesting results leading to the limitations of our current architecture, and directions for future work.

2 Background

2.1 Neural Networks

Neural Networks are the backbone of NLP. They are modeled as a network of basic computation units called *neurons*, which are connected with each other in specific layouts similar to the human brain. The following section covers the fundamentals of Neural Networks (Jurafsky and Martin 2024, Chapter 7).

2.1.1 Neurons

The building block is a single neural unit, a neuron. It is, at heart, taking a weighted sum of its real valued inputs with an additional term in the sum called the **bias** term. Given an input vector x with a corresponding weight vector w and a scalar bias term b, the weighted sum z can be represented as

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

Instead of z being a linear function of x, neural units apply a non-linear activate function f to z. This output is called the **activation** value for the unit, a.

The most popular non-linear activation functions are the sigmoid, tanh and the Rectified Linear Unit (ReLU). The sigmoid function has a range of (0, 1) which is useful in squashing outliers towards 0 or 1, and it is differentiable. The sigmoid function, mathematically can be represented as:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + \exp(-(\mathbf{w} \cdot \mathbf{x} + b))}$$

The tanh function which ranges from -1 to +1 can be represented as:

$$y = \tanh(z) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

The ReLU activation function can be represented as:

$$y = ReLU(z) = \max(z, 0)$$

The activation functions mentioned above have different properties that make them useful for a variety of language applications. These functions are crucial as without them, the neural network would result in a single layer network. A multilayer network would just be a notational variant of a single layer network with a different set of weights, thereby losing all the representational power of multilayer networks.

2.1.2 Feedforward Neural Networks

A feedforward network, one of the simplest kind, is a multilayer network in which the units are connected with no cycles. The core of the neural network is the **hidden layer** formed of hidden units $\mathbf{h_i}$ each of which is a neural unit. In standard architecture, each layer is **fully connected** meaning that every unit in every layer takes an input which is the output from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers.

As covered in 2.1.1, each hidden unit has as parameters a weight vector and bias. We can represent the parameters for the whole hidden layer by combining the weight vectors and bias for each unit i into a matrix $\mathbf{W}$ and a single bias vector for the whole layer. Therefore, each element $\mathbf{W_{ji}}$ of the weight matrix $\mathbf{W}$ represents the weight of the connection from the ith input unit x_i to the jth hidden unit h_j. We can multiply the weight matrix defined above, with the input vector $\mathbf{x}$ adding the bias vector $\mathbf{b}$ and applying the activation function, which can be represented as:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

The output layer is thus a vector $\mathbf{y}$ that gives a probability distribution across all the output nodes for example in the case of classification. The output layer also has a weight matrix (let's call it $\mathbf{U}$). Since some models do not include a bias vector in the output, we can simplify the equation by eliminating the bias vector. The weight matrix is multiplied by its input vector $(\mathbf{h})$ to produce the output z:

$$\mathbf{z} = \mathbf{U}\mathbf{h}$$

For the task of classification, $\mathbf{z}$ can't be the output of the classifier since it is a vector of real-valued numbers. To conveniently normalize a vector of real values, we can use the **softmax** function. Generally, for any vector $\mathbf{z}$ of dimensionality d, the softmax is defined as:

$$softmax(\mathbf{z_i}) = \frac{\exp(\mathbf{z_i})}{\sum_{j=1}^{d} \exp(\mathbf{z_j})} 1 \leq i \leq d$$

Therefore, the probability distribution $\mathbf{y}$ can be written as:

$$\mathbf{y} = softmax(\mathbf{z})$$

This completes the representation of a simple 2 layer network which is also represented visually in figure 2.1. This can be extended to deeper networks with different activation

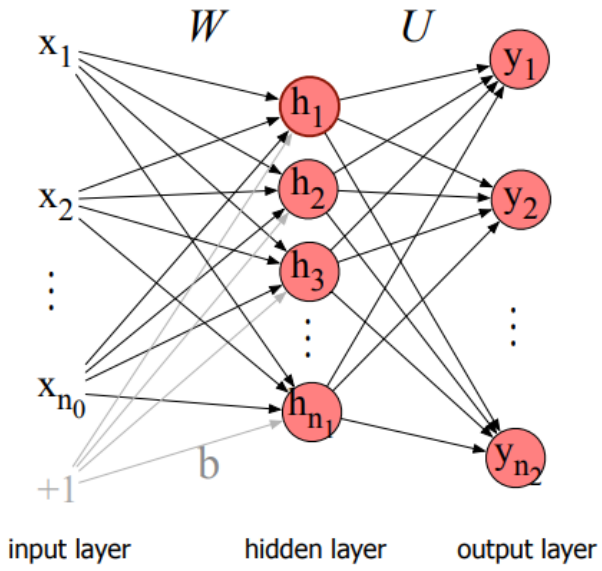


Figure 2.1: The Transformer encoder-decoder architecture. The figure is found in Jurafsky and Martin 2024, Chapter 7.

functions at different layers if needed.

2.1.3 Training Neural Networks

The goal of the training procedure in feedforward neural networks is to learn parameters $\mathbf{W}^{[i]}$ and $\mathbf{b}^{[i]}$ for each layer i to get the system's estimate of the true output y which needs to be as close to the true output as possible. We can call this estimate $\hat{y}$.

Therefore, we would need a loss function that models the distance between the gold output and the system's output.

Cross-Entropy Loss

A common choice is using the **cross-entropy loss**. Also known as log loss, it is predominantly used in classification tasks, particularly scenarios involving probability distributions. Let's assume that the system output is represented as $\hat{y}$ and the gold output is represented as y. The cross-entropy loss can be defined as:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})]$$

If we assume that we are doing a hard classification task, where only one class is the correct one, the true label $\mathbf{y}$ is then a vector with K elements, each corresponding to a class, with $\mathbf{y_c} = 1$, if the correct class is c with all other elements being 0. The loss function for a single example $\mathbf{x}$ is the negative sum of the logs of the K output classes, each weighted by their probability y_k:

$$L_{CE}(\hat{y}, y) = -\sum_{k=1}^{K} \mathbf{y_k} \log \hat{\mathbf{y}}_\mathbf{k}$$

In other words, the cross-entropy loss can be simplified and represented as the negative log of the output probabilities corresponding to the correct class. We can therefore call this the negative log likelihood loss:

$$L_{CE}(\hat{y}, y) = -\log \hat{\mathbf{y}}_\mathbf{c}$$

where c is the correct class.

Mean Squared Error (MSE)

Also known as L2 loss, it measures the average squared difference between the predicted and actual values. Like Cross-Entropy being used for classification, MSE loss is usually used for regression. Mathematically, it can be represented as:

$$MSE = \frac{1}{n} \sum_{i} (\mathbf{y_i} - \hat{\mathbf{y}}_\mathbf{i})^2$$

where $\mathbf{y_i}$ is the true value, $\hat{\mathbf{y}}_\mathbf{i}$ is the predicted value, and n is the number of samples

AdamW Optimizer

Also, to find the parameters that minimize this loss function, we will use a certain optimization algorithm. In this thesis, we use the **AdamW** optimizer which was an extension of the popular Adam incorporating some modifications, as explained Loshchilov and Hutter 2017. The 'W' in AdamW stands for decoupled weight decay, which is the key innovation of this optimizer.

To understand AdamW, its essential to first understand how Adam works. Adam (Kingma and J. Ba 2017) calculates adaptive learning rates for each parameter by estimating the first and second moments of gradients. It maintains running averages of both the gradients (first moment) and the squared gradients (second moment). These averages are used to adjust the learning rate for each parameter individually, thereby allowing the optimizer to take larger steps for infrequent parameters and smaller steps for frequent ones.

The update rule for Adam can be summarized as follows:

$$\mathbf{m}_{t+1} = \beta_1 \cdot \mathbf{m}_t + (1 - \beta_1) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}_t)$$
$$\mathbf{v}_{t+1} = \beta_2 \cdot \mathbf{v}_t + (1 - \beta_2) \cdot (\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}_t))^2$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon}$$

where $\mathbf{m}_{t+1}$ and $\mathbf{v}_{t+1}$ are the estimates of the first and second moments of the gradients, $\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}_t)$ is the gradient at time t, α is the learning rate, β_1 and β_2 are the decay rates for the moment estimates, and ϵ is a small constant for numerical stability.

AdamW builds upon this by addressing a key issue with the way Adam interacted with weight decay regularization. In standard Adam, weight decay is typically implemented as an addition term in the loss function, therefore becoming a part of the gradient computation. This approach can lead to suboptimal regularization especially with adaptive learning rates.

Instead of incorporating weight decay into gradient computations, AdamW applies it directly to the weight update step. Therefore, the modified update rule for AdamW can be expressed as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon} + \lambda \cdot \theta_{t-1}$$

where λ is the weight decay coefficient. This decoupling ensures that weight decay is applied consistently regardless of the adaptive learning rates, leading to more effective regularization.

Finally, the optimization algorithm needs to know the gradient of the loss function. For neural networks, with millions of parameters, if not billions, it can become cumbersome to see how to compute the partial derivative of some weight in layer 1 when the loss is attached to some later layer. This is where **backpropagation** (Rumelhart, Durbin, Golden, and Chauvin 2013) comes into play.

2.1.4 Improving the Training Process

The model is trained for a fixed number of iterations and in every step, the loss value is calculated. This value is used to determine how the parameters should be updated according to a certain optimization algorithm. Although, there are a few problems with learning these weights as the model can *memorize* and become predictive of the outcome. This problem is called **overfitting**. Ideally, the model should be able to **generalize** well for the training data to the unseen test set, which is a crucial challenge in machine learning. Typically, the validation loss should decrease with the training loss. In the case of overfitting, the training loss keeps decreasing till the minimum possible value, but the validation loss begins rising after a certain number of iterations. This shows

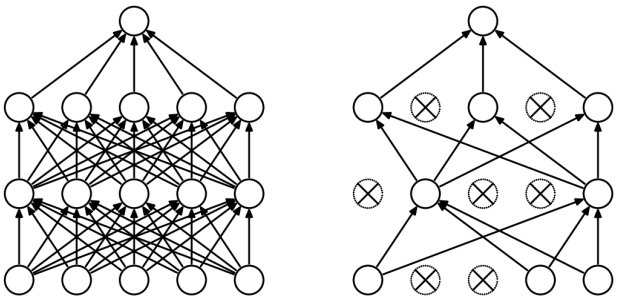
how the model is losing its ability to generalize to unseen data and a simple way to ensure generalization would be to stop training if the validation loss is at its lowest value.

Conversely, the model can also **underfit** the training data, which means the model is not learning enough from the training data. Both of these cases are dependent on the model's **capacity**, which in simple words, is how much a model can learn. If the capacity is too large, the model tends to learn the noise and overfit. Similarly, if the capacity is too small, the model might not learn enough and underfit.

Dropout

To keep the model's capacity in check, one can implement regularization. They are a collection of strategies that aim to reduce the validation loss at the possible expense of the training loss (Goodfellow, Bengio, and Courville 2016). An important one is dropout where some units and their connections are randomly dropped from the network during training (Srivastava, G. Hinton, Krizhevsky, et al. 2014). The proportion of neurons that are dropped out is referred to as the dropout rate which is a hyperparameter that can be tuned.

By removing neurons, dropout forces the network to learn more robust features that are useful in combination with different subsets of other neurons. Dropout can also be understood as a form of 'model averaging'. Each training iteration effectively trains a different subnetwork and the final model can be seen as an ensemble of these subnetworks. During inference, dropout is typically turned off, and the weights are scaled according to the dropout rate to compensate for the full network being active. Dropout is visually represented in 2.2



(a) Standard Neural Net (b) After applying dropout.

Figure 2.2: Dropout applied to a standard neural network (Srivastava, G. Hinton, Krizhevsky, et al. 2014).

Linear Learning Rate Scheduler

This is a technique used to adjust the learning rate during the training process. The learning rate if a crucial hyperparameter in neural network training, controlling the step size at each iteration while moving toward a minimum of the loss function. A poorly chosen learning rate can lead to slow convergence or even divergence.

It involves a **warmup phase** where the learning rate is linearly increased from a small initial value to a predefined maximum value. The warmup period helps to stabilize early stages of training, particularly when using optimizers with adaptive learning rates like Adam. The next phase is the **decay phase**, where after reaching the maximum learning rate, it is then linearly decreased over the remaining training iterations. This reduction allows the model to make fine-grained updates. Mathematically, these phases can be expressed as:

Warmup phase:

$$lr(t) = lr_0 + \frac{t \cdot (lr_{max} - lr_0)}{warmup\_steps}$$

Decay phase:

$$lr(t) = lr_{max} - \frac{(t - warmup\_steps) \cdot (lr_{max} - lr_{final})}{total\_steps - warmup\_steps}$$

where t is the current training step, lr_0 is the starting learning rate, lr_{max} is the maximum learning rate after warmup, lr_{final} is the learning rate at the end of training, $warmup\_steps$ is the number of warmup steps and the $total\_steps$ is the total number of training steps.

The combination of dropout for regularization and a linear learning rate scheduler for optimization often results in more stable training.

Gradient Clipping

This is a technique is used in training deep neural networks particularly for addressing the challenge of exploding gradients. The core idea behind gradient clipping is to limit the magnitude of gradients during the optimization process. Thus, gradients are prevented from becoming too large in magnitude which can cause drastic updates to the model parameters and potentially derail the training process. This technique acts as a form of regularization, ensuring the model updates remain within a reasonable range.

There are two primary methods of gradient clipping:

- Gradient Norm Clipping: This approach scales down the gradient when its L2 norm exceeds a predetermined threshold. More formally, if $\|g\| > c$, then $\left(\frac{c}{\|g\|}\right)$.

- Gradient Value Clipping: This approach includes defining a minimum and maximum clip value. If the gradient exceeds some threshold value, we clip that gradient to the threshold, and vice versa.

For the purpose of training, finding the right set of *hyperparameters* is also crucial. The parameters of a simple neural network include weights W and biases b which are learned by the optimization algorithm. The hyperparameters are a set of parameters that are chosen by the developer, and they include, but not limited to, the learning rate, mini-batch size, model architecture, regularization technique.

2.1.5 Transformers Architecture

The transformer architecture, as introduced by (Vaswani, Shazeer, Parmar, et al. 2017) has proved to be one of the central components of state-of-the-art NLP models. The motivation lies behind the idea that language is sequential and fully connected neural networks do not provide the ideal functionality for language related tasks.

Recurrent neural networks (Rumelhart, G. E. Hinton, and Williams 1986) were introduced to overcome this, and the idea was that they feed the network's hidden output back into parts of the input thus creating this network which contain cycles. The goal was to have a network which could be queried token by token, i.e., low level language units ranging from letter to words, and have access to the information from previous tokens as well. A problem with these networks was its inability to model long-term dependencies within a sequence. This gave rise to more elaborate approaches like *Long Short-Term Memory* (LSTM) (Hochreiter and Schmidhuber 1996). They imported recurrent information flow but still suffered from the long-term dependency problem and also computational performance issues due to it not being well paralleled. Transformers are also a neural architecture that can handle distant information but unlike LSTMs, they are not based on recurrent connections, therefore being more efficient to build at scale. They used a mechanism called **self-attention**, the key innovation of transformers. Self-attention allows a network to extract and use information from arbitrarily large contexts. This allows the Transformer to understand the context of each word in relation to others in the sequence, capturing dependencies and relationships across long distances. Before we get into the architecture of transformers, we can briefly cover how self-attention and *multi-head-attention* works.

An attention function maps a query and a set of key-value pairs to an output. Let's assume out input embeddings of N tokens of the input sequence can be packed into a single matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$ where N is the input length and d is the dimensionality of the embedding. We multiply $\mathbf{X}$ by the key, query and value matrices all dimensionality $d \times d$ to produce $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$.

This gives us:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}; \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}}$$

Based on this, self-attention can be defined as follows:

$$\mathbf{A} = SelfAttention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{K}^{\mathbf{T}}}{\sqrt{d}}\right)\mathbf{V}$$

The division by $\sqrt{d}$ was introduced as scaling factor to counter the issue of small gradients of the softmax function which could result from the dot product becoming too large as dimensionality grows.

Since it would be difficult for a single self-attention model to capture all the different relations among its inputs, transformers addressed this issue with multi-head attention. These are sets of self-attention layers, called heads, that compute parallely while having their own weight matrices. If we define the i^{th} head to be:

$$\mathbf{head_i} = SelfAttention(\mathbf{Q}, \mathbf{K}, \mathbf{V})$$

Multi-head attention is defined as:

$$\mathbf{A} = MultiHeadAttention(\mathbf{X}) = (\mathbf{head_1} \oplus \mathbf{head_2} \cdots \oplus \mathbf{head_h})\mathbf{W}^{O}$$

Self-attention calculations being the core of the transformer block, the next parts of the transformer architecture we need to address the feedforward layer, residual connections and layer norm. The feedforward layer contains N position-wise networks. Each is a fully connected 2 layer network, i.e., one hidden layer, two weight matrices. Unlike attention, the feedforward networks are independent for each position and can be computed in parallel.

Residual connections are responsible for passing information from a lower layer to a higher layer without going through the intermediate layers. This results in an improvement in learning by giving higher level layers access to information in the lower layers (He, Zhang, Ren, and Sun 2015). In transformers, they are implemented by adding a layer's input vector to its output vector before passing it forward.

These vectors are then normalized using layer norm (J. L. Ba, Kiros, and G. E. Hinton 2016). It does so by maintaining the values of a hidden layer in a range that allows gradient based training. For a particular token position i, we have a vector of dimensionality d, which is the input to layer norm. The output produced is a normalized vector of the same dimensionality. To do, we need the mean, μ and standard deviation, σ. Given a hidden layer with dimensionality d_h, the values are

calculated as:

$$\mu = \frac{1}{d_h}\sum_{i=1}^{d_h} x_i$$
$$\sigma = \sqrt{\frac{1}{d_h}\sum_{i=1}^{d_h}(x_i - \mu)^2}$$

The vector components can be normalized as follows:

$$\hat{\mathbf{x}} = \frac{(x - \mu)}{\sigma}$$

where $\hat{\mathbf{x}}$ is a new vector with zero mean and a standard deviation of one. In the standard implementation of layer norm, two learnable parameters γ and β representing gain and offset values are used which gives us the following representation:

$$LayerNorm = \gamma\hat{\mathbf{x}} + \beta$$

Compiling all these components together, we can represent the architecture of a block of the transformer as follows:

$$\mathbf{T^1} = SelfAttention(\mathbf{X})$$
$$\mathbf{T^2} = X + \mathbf{T^1}$$
$$\mathbf{T^3} = LayerNorm(\mathbf{T^2})$$
$$\mathbf{T^4} = FFN(\mathbf{T^3})$$
$$\mathbf{T^5} = \mathbf{T^4} + \mathbf{T^3}$$
$$\mathbf{H} = LayerNorm(\mathbf{T^5})$$

The entire architecture is depicted in figure 2.3.

It is also important to discuss what exactly the input, $\mathbf{X}$ is. This matrix of shape $[N \times d]$ has an **embedding** for every word in the context and this embedding is created by combining an input token embedding and an input positional embedding. The input token embedding is a vector of dimension d is a starting representation of the input token. This embedding representation learns how to incorporate context and changes as it goes through the transformer layers. The positional embeddings are specific to each position in an input sequence. As proposed in Vaswani, Shazeer, Parmar, et al. 2017, they used sine and cosine functions of different frequencies.

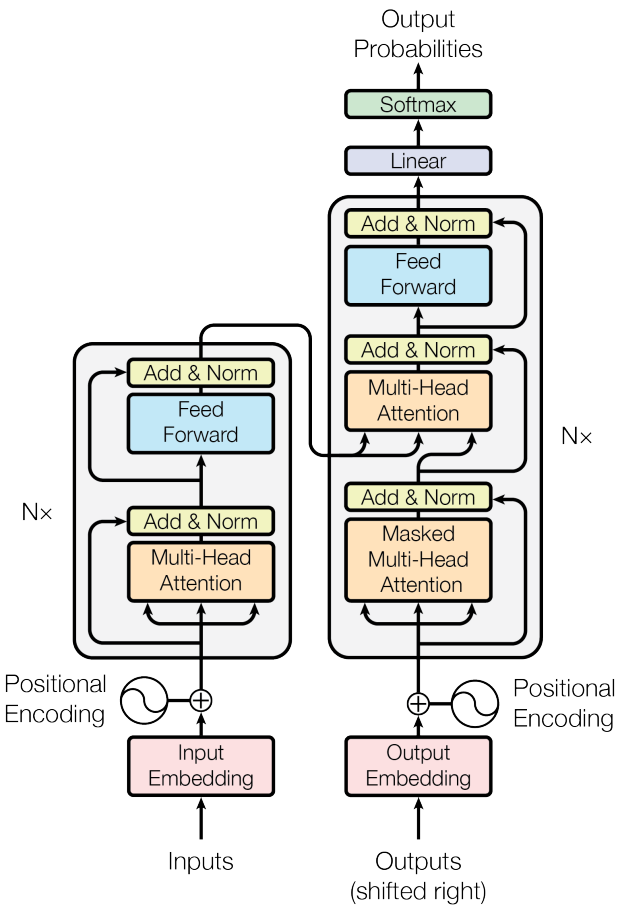


Figure 2.3: The Transformer encoder-decoder architecture (Vaswani, Shazeer, Parmar, et al. 2017).

2.2 NLP Background

This section covers how tokenization works and how the transformer model paved the way for NLP by introducing BERT. We also cover what Adapters are, how LoRA works and briefly discuss Multilingual NLP models. Finally, we see how we benchmark NLP models using IndicGLUE tasks.

2.2.1 Tokenization

Tokenization is simply the process of converting a sequence of text into smaller parts, known as tokens. Most NLP algorithms and large language models break up words

into **subword tokens**, which can be words, parts of words, or even letters. BERT specifically uses the **Wordpiece** algorithm (Wu, Schuster, Z. Chen, et al. 2016), which starts from a small vocabulary including special tokens used by the model and the alphabet. It uses a greedy longest-match-first strategy to tokenize a single word.

In our methodology, as we will see later, we do not use a subword tokenization algorithm like Wordpiece, which BERT was trained with. These tokenizers tend to disregard the morphology of the language it is dealing with (Hofmann, Pierrehumbert, and Schütze 2021). Therefore, we intend to keep the words as they are, or use morphological tokenizers to preserve the morphology of Hindi.

2.2.2 BERT

BERT introduced by Devlin et al.(2019), stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. It is designed to pre-train deep bidirectional representations by using both left and right context in all layers. The two classification objectives it was pre-trained were:

- Masked Language Modelling (MLM): This objective involves masking tokens within a given sequence and aims to predict this token based on its preceding and following tokens. For BERT, if the i^{th} token is chosen, we replace it with $'[MASK]'$ token 80% of the time, a random token 10% of the time, or leave it unchanged 10% of the time.
- Next Sentence Prediction (NSP): The goal of the task is to train the model to understand the relationship between two sentences. Specifically, when choosing the sentences A and B for each pre-training example, 50% of the time, B is the actual next sentence that follows A, and 50% of the time it is a random sentence from the corpus. The input sequence is padded by a preceding $'[CLS]'$ token. The way sentences are differentiated are by a separator token $'[SEP]'$

The model architecture implementation is almost identical to the encoder part of the transformer introduced in Vaswani, Shazeer, Parmar, et al. 2017. The BERT model used in this thesis is commonly known as the $\mathbf{BERT_{BASE}}$ which consists of 12 layers, 768 as the hidden size (embedding dimension), and 12 self-attention heads each of size 64, and a feedforward hidden size of 3072. The total parameters come out to be 110M. The model consists of an embedding layer which maps tokens to dense embeddings and adds positional encoding. Then we have the transformer encoder layer, and finally a classification head that varies based on the downstream task it was fine-tuned for. BERT itself has an MLM and NSP head, which can be replaced with a classification head for the downstream task's requirements. The BERT architecture is depicted in figure 2.4.

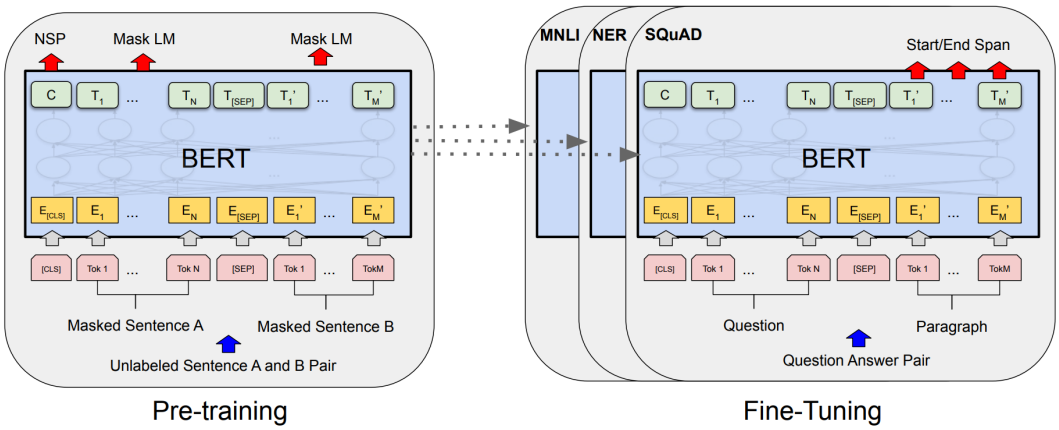


Figure 2.4: The BERT architecture (Devlin, Chang, Lee, and Toutanova 2019).

2.2.3 Adapters

Fine-tuning LLMs is parameter inefficient in the presence of many downstream tasks. Adapter modules (Houlsby, Giurgiu, Jastrzebski, et al. 2019) which are a compact and extensible model which only add a few trainable parameters per task. The parameters of the original model remain fixed therefore reducing the computational overhead.

The adapter module proposed in Houlsby, Giurgiu, Jastrzebski, et al. 2019 was a bottleneck adapter module. It involved injecting new layers into the original network. The weights of the original network are untouched, whilst the new adapter layers are initialised at random. While fine-tuning, the parameters of the original network are frozen and only the adapter parameters are learnt. Each layer of the transformer contains two primary sub-layers, an attention layer and a feedforward layer. In this implementation, two serial adapters are inserted into each of these sub-layers. The adapter is always applied directly to the output of the sub-layer, after the projection back to the input size, but before adding the skip connection back. The output of the adapter is then passed directly into the following layer normalization.

To limit the number of parameters, a bottleneck architecture was proposed. The adapters first project the original d-dimensional features into a smaller dimension, m, apply a non-linearity, then project back to d dimensions. The total number of parameters added per layer, including biases is $2md + d + m$. By setting $m \ll d$, the number of parameters were limited per task. The bottleneck dimension, m provides a simple means to trade-off performance with parameter efficiency. The adapter module itself has a skip-connection. With this, if the parameters of the projection layers are initialised to near-zero, the module's initialisation approximates to an identity function. Alongside the layers in the adapter module, new layer normalization parameters were also trained

per task. The adapter module architecture is depicted in 2.5. The green layers are trained on the downstream data, which includes the adapter, layer normalization parameters and the final classification layer (not shown in the figure).

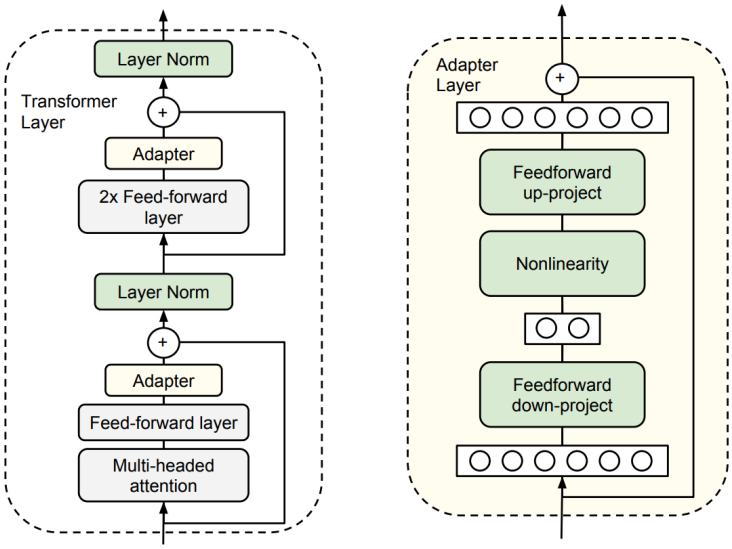


Figure 2.5: Architecture of the adapter module (Houlsby, Giurgiu, Jastrzebski, et al. 2019).

Poth, Sterz, Paul, et al. 2023 introduced a library that unifies parameter-efficient and modular transfer learning in large language models. Bottleneck adapters are one of the early methods in the family of transfer learning methods. Subsequent designs have adapted a transformer model's self attentions (Li and Liang 2021), bias terms (Ben Zaken, Goldberg, and Ravfogel 2022), or embeddings (Pfeiffer, Vulić, Gurevych, and Ruder 2021). Other methods have also focussed on optimizing the parameter efficiency (Karimi Mahabadi, Henderson, and Ruder 2021) and runtime efficiency (Hu, Shen, Wallis, et al. 2021). In the following section, we will cover a method proposed by Hu, Shen, Wallis, et al. 2021 which adds trainable rank decomposition matrices as adapters.

2.2.4 LoRA

Mathematical Background

Before understanding how LoRA works, it is crucial to cover some linear algebra concepts which will be useful to understand LoRA.

The **rank** is the dimension of the vector space generated by its columns. This is given by the number of *linearly independent* columns (or rows) in a given matrix. It can be proved that the column rank is always equal to the row rank. Hence, for a matrix A with m rows and n columns:

$$rank(A) \leq \min(m, n)$$

Based on this, a matrix can be classified into mainly two types- Full-rank matrix and Rank-Deficient matrix. A matrix A_{mn} is called a **full-rank matrix** if

$$rank(A) = \min(m, n)$$

A matrix is **rank deficient** if

$$rank(A) \ll \min(m, n)$$

A low-rank matrix is defined as a rank-deficient matrix if its rank is significantly lower than the minimum number of rows and columns.

Another key concept is **rank decomposition**. Consider a matrix A_{mn}, its rank decomposition is of the form $A = C_{mr}F_{rn}$ where $rank(A) = r$. It can be proven that every (finite) matrix has a rank decomposition.

Methodology

Low-**R**ank **A**daptation (Hu, Shen, Wallis, et al. 2021), or LoRA , freezes the pre-trained model weights and adds trainable rank decomposition matrices into each layer of the Transformer architecture. Using this, a pre-trained model can be shared and used to build many small LoRA modules for different tasks. We can freeze the shared model and efficiently switch tasks by simply replacing the matrices A and B. This greatly reduces the storage requirement and task-switching over-head significantly. Moreover, it makes training more efficient and reduces the hardware barrier as we don't need to calculate gradients for most parameters. Instead, we only optimize the low-rank matrices.

Since a neural network contains many dense layers which perform matrix multiplication, these matrices typically have full-rank. Inspired by Aghajanyan, Zettlemoyer, and S. Gupta 2020, the hypothesis LoRA is based on is that the updates to the weights also have a low 'intrinsic rank' during adaptation. Since large models are trained to capture representations of a certain domain, they capture a variety of features which allow them to be used for a variety of tasks with reasonable accuracy. However, when adapting these models to a specific downstream task, only a few features need to be re-learnt. This would mean that the update matrix, which we refer to as ΔW can be

a low-rank matrix. So for a pre-trained weight matrix $W_0 \in \mathbb{R}^{d\times k}$, it is represented as with a low rank decomposition $W_0 + \Delta W = W_0 + BA$ where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and the rank $r \ll \min(d, k)$. During training, W_0 is frozen while A and B contain trainable parameters, which is depicted in 2.6. For $h = W_0 x$, the forward pass yields:

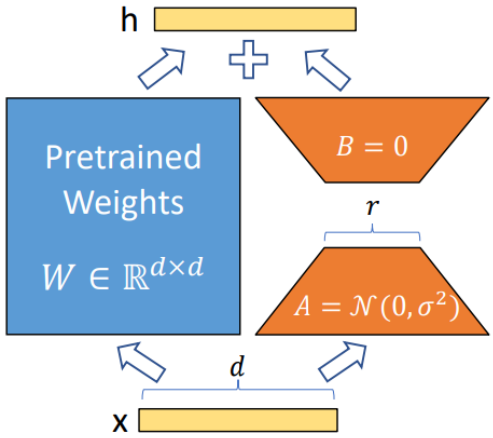


Figure 2.6: Reparameterization in LoRA (Hu, Shen, Wallis, et al. 2021).

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

A and B are initialised with a random Gaussian initialisation and zero, respectively. Therefore, $\Delta W = BA$ is zero at the beginning and $\Delta W x$ is scaled by $\frac{\alpha}{r}$, where α is a constant in r.

Now lets look at how we can apply LoRA to Transformers. We can apply LoRA to any subset of weight matrices in a neural network to reduce the number of trainable parameters. In the self-attention module (W_q, W_k, W_v, W_O) and the two in the feedforward module. In the work presented by Hu, Shen, Wallis, et al. 2021, they limited their research to adapting the attention weights and freezing the multi-layer perceptron modules, but one could add LoRA matrices to the dense and output layers as well.

2.2.5 Multilingual Large Language Models

Most of the state-of-the-art models were pre-trained solely on English text which limits their use given roughly 80% of the world population does not speak English (Crystal 2008). Multilingual models are a type of neural architecture designed to handle multiple languages at one time. These models aim to learn representations that generalize across languages, therefore allowing them to perform a variety of tasks in which knowledge of diverse linguistic structures is necessary.

The key feature of such models is the attempt to map words from different languages into a shared embedding space. This allows words with similar meaning in different languages to have close vector representations. To deal with variance in languages, these models typically use subword tokenization strategies to ensure rare or unseen words are broken down into subword units. These units can be shared across languages, making it easier for the model to generalize. A key advantage of multilingual models is their ability to transfer knowledge across languages. This cross-lingual transfer is achieved by learning linguistic features that capture the true meaning of these features across multiple languages. Multilingual models also often use MLM objectives to predict masked tokens in sentences across various languages, therefore improving their ability to handle linguistic diversity.

The core concept behind these models are multilingual embeddings. Just as in this work, we aim to align semantically similar words in English and Hindi into the same or at least, closely related vector spaces.

One of the earliest and most widely use multilingual models was mBERT (Devlin, Chang, Lee, and Toutanova 2019). It is based on the original BERT architecture and is pre-trained on 104 languages using a shared WordPiece tokenizer. It uses MLM and NSP as its pre-training objective just like BERT. One of its strengths is its ability to perform zero-shot transfer, where a model fine-tuned in one language on a certain task can generalize to other languages without explicit training. XLM-RoBERTa (Conneau, Khandelwal, N. Goyal, et al. 2019), another multilingual model, was built on top of RoBERTa (Liu, Ott, N. Goyal, et al. 2019) and is trained on 100 languages using a more diverse and larger dataset than mBERT. It demonstrates superior performance over mBERT, especially for low-resource languages. Finally, a more focussed model for our use case, we have IndicBERT (Doddapaneni, Aralikatte, Ramesh, et al. 2023) which was specifically designed for 12 Indian languages. It is more lightweight than models like mBERT and XLM-R, making it suitable for low-resource environments. It efficiently handles the complexities of Indian languages and also outperforms general multilingual models on tasks involving Indian languages.

2.2.6 Benchmarking Large language Models

Preparing a fine-tuned model using state-of-the-art techniques is a major part of the task, but not all of it. To evaluate the performance of the model and quantify its language understanding is also a crucial part of the task. Benchmarks such as GLUE (A. Wang, Singh, Michael, et al. 2018) are important for tracking the efficacy of NLP models across languages. Such a benchmark was missing for Indic languages. Doddapaneni, Aralikatte, Ramesh, et al. 2023 provided the Indic General Language Understanding Evaluation Benchmark (IndicGLUE) which is a collection of various NLP downstream

tasks for benchmarking. The goal is to provide an evaluation benchmark for natural language understanding capabilities of NLP models on diverse tasks and multiple Indian languages. Standard train and test splits for all datasets mentioned in this benchmark are available on the web for reproducibility. The authors provided these datasets for many languages but for our research, we chose only the Hindi datasets. Out of the many tasks provided in this evaluation benchmark, we chose the following:

- **COPA**: The Choice of Plausible Alternatives (Roemmele, Bejan, and Gordon 2011) task evaluates open-domain common sense causal reasoning. The tasks consist of a large set of two-choice questions, constructed as a *premise* and two alternatives written as sentences. The task is to select the alternative that is more plausibly the cause or effect of the situation. The dataset contained 899 rows in total.
- **Sentiment Analysis**: For this task, the benchmark offered a dataset called the IIT-Patna Movie and Product Sentiment Analysis dataset available in Hindi (M. S. Akhtar, Kumar, Ekbal, and Bhattacharyya 2016). The dataset contained 3100 rows in total
- **Category Classification**: The benchmark provided the BBC News Article classification dataset where the given text was to be classified among 14 valid values. The dataset contained 4330 entries in total.
- **Discourse Mode Classification**: Given a sentence, the task here is to classify it into one of the following discourse categories: argumentative, descriptive, dialogic, informative, narrative. The dataset used here is the MIDAS Hindi Discourse Analysis (S. S. Akhtar, A. Gupta, Vajpayee, et al. 2017) and it contains 9970 rows in total.

2.3 Related Work

In this section, we cover work which inspires our approach to learn morphological subword embeddings with our specific experiment setting. Most approaches have predominantly dealt with high resource languages and huge corpora. Extending them for generating cross lingual predictions across low resource languages is problematic. Additionally, in the case of a limited corpora available, the embeddings tend to be sparse due to the rich linguistic properties of some languages. It is often the case for Indian languages that there are a large number of morphological variants for each root word, most of which occur rarely in training sets.

Embeddings trained on subword representations are shown to yield better task performance than those trained only on whole words (Chaudhary, Zhou, Levin, et al.

2018). Further, it was shown that embeddings trained on morphological representations often outperform those trained only on whole words. Word2vec model's (Mikolov, K. Chen, Corrado, and Dean 2013) inability to generalize for out of vocabulary words that may share morphemes with in-vocabulary words was a significant problem for languages like Hindi which are morphologically rich. Previous work (Luong, Socher, and Manning 2013; Botha and Blunsom 2014) found that morphological relationship between words can be captured more directly if embeddings are trained on morphological representations. Another extension of the Word2vec model (Chimalamarri, Sitaram, and Jain 2020) had the goal to develop embeddings at a morpheme level rather than word level to improve cross lingual prediction on three Indian languages, including Hindi. By performing a simple morphological segmentation prior to generating the embeddings, they were able to achieve better similarity scores and accuracies for target language predictions. Even though their task was more in line to achieve accurate translations, their approach further proves that performing morphological segmentation can be beneficial for our task.

Another approach, for the purpose of machine translation, but still including interesting insights regarding the embedding layer. This approach (V. Goyal, Mishra, and Sharma 2020) presented efforts towards building a Hindi to English machine translation system using Transformer models via exploiting linguistic features of Hindi. This was done by generalizing the embedding layer of the encoder in the standard Transformer architecture to support the inclusion of arbitrary features, in addition to the baseline token feature. Morphological features, part-of-speech (POS) tags and lemmas were added as a combination of features to the encoder input. This work also experimented with subword tokenization algorithms as they wanted to minimize the risk of encountering an out of vocabulary word to ensure successful translations.

While incorporating morphological information has resulted in a better quality of embeddings, subword tokenization still pose an important issue. Subword tokenization algorithms have the advantage of a fixed sized vocabulary by splitting them into smaller units, they tend to do so by disregarding language morphology (Hofmann, Pierrehumbert, and Schütze 2021), and some of them favour splittings with more subword units than would be necessary (Church 2020). Knowing what the best strategy was to combine contextualized subword representations into a contextualized word level representation was a crucial research question. The impact of word splitting on semantic content was researched and also evaluated (Soler, Labeau, and Clavel 2024) to see how the quality of split-word representations compare to that of full word representations. A similar concern was researched (Kim, Choi, and Jeong 2024) regarding how implementing such morphological tokenizers can be used to train embeddings for newly added tokens. One key learning from this study was how their initialisation method for tokens which were new in the vocabulary, and they did so by

using the average embeddings of the subword tokens that makes up these tokens. To learn input embeddings of these newly added tokens, they froze the model parameters and allowed the model to adjust its recognition and processing of these tokens from the beginning. FLOTA (Hofmann, Schuetze, and Pierrehumbert 2022), a simple and effective method to improve the tokenization of pre-trained language models, further expands on the benefits of preserving the morphological structure of words during tokenization.

Different tokenization strategies including their own essence of morphological segmentation have been proposed over the years in various languages. A tokenization system for Kurdish was proposed (Rai and Borah 2021) which included word tokenization by splitting text by space and matching words with a pre-built lexicon. Then, a morphological analysis is done to retrieve prefixes and suffixes of the word. A similar approach was implemented for Turkish (Toraman, Yilmaz, Şahinuç, and Ozcelik 2023). The impact of different tokenizers was analyzed, ranging from character to word level, on various downstream tasks for Turkish language models. it was also shown that increasing the vocabulary size improves the performances of morphological tokenizers more than that of subword tokenization algorithms like BPE and Wordpiece. Experiments were conducted (Hou, Katinskaia, Vu, and Yangarber 2023) to compare subword tokenization algorithms used in GPT and BERT with unsupervised morphological tokenizers like Morfessor (Creutz, Lagus, and Virpioja 2006). Morfessor is based on the Minimum Description Length which factors lexicons with fewer and shorted sub-words. It learns the model parameters with MAP estimation using a two part cost function: a prior cost and a corpus cost. The original Morfessor has been extended to Morfessor 2.0 (Smit, Virpioja, Grönroos, and Kurimo 2014), a framework of what will also be used in our work.

Specifically for Hindi, the Indic NLP Library (Kunchukuttan 2020) provides tools for common text processing in Indian languages. Despite the diversity in Indic languages, they share a lot of similarity in terms of script, phonology, language, syntax, etc. This library provides a general solution to some commonly required toolset of Indian language text. More specifically, this library provides the unsupervised morphological analyser built using Morfessor 2.0, which has the ability to analyze a single word and return a list of component morphemes. The Indic NLP catalog provides the most comprehensive catalog of NLP datasets, models and other resources for Indian languages. This source has been our core reference for state-of-the-art models in Hindi, datasets and evaluation benchmarks.

3 Methodology

This section provides an overview and brief descriptions of the individual components in our architecture.

3.1 Overview

Let's briefly go over the whole methodology before diving into the details of the individual components. Our goal is to find multilingual word embeddings between two languages of choice. As discussed, for this thesis, we have chosen Hindi and English.

Since we want to learn meaningful embeddings between two words across different languages, we need a *dictionary* which contains one-to-one translations between Hindi and English words. The first step, therefore, is to *create the dictionary* for the two languages by compiling various sources. As we do not plan to divide our words into sub-word token representations, unknown tokens will not be handled the way they are by sub-word tokenization algorithms like WordPiece. Therefore, it is crucial to cover as many words as possible to ensure we can minimize the unknown words the model will encounter. In the case we encounter an unknown word, we will replace it with a placeholder for unknown words. During training, we select sentences according to the number of unknown tokens in them to avoid having sentences with too many unknown words.

Once we have our dictionary ready, we need to discuss *tokenization*. As we have opted not to implement a sub-word tokenizer, we need to implement a tokenizer that handles Hindi sentences and successfully tokenizes them into English words using our dictionary. Since BERT was trained on data which was tokenized by WordPiece, the model's embeddings are specifically tied to this vocabulary. Our vocabulary, being made of whole words or specific morphemes, could be different with some potential overlap. Thus, BERT might not inherently 'understand' these new tokens as BERT's embedding layer is familiar with the vocabulary it was trained on. There are two options we considered to fix this problem. One would be to simply train the whole model from scratch with our new tokenizer, so that BERT learns how to deal with tokens according to our needs. This method, despite having the best chance of success theoretically, is computationally expensive and time-consuming. The method we

decided to go with was the *Embedding bridge*. The aim here is to bridge the gap between a custom tokenizer and pre-trained BERT embeddings. We implement an approach where new embeddings are trained if the word in our vocabulary does not exist in BERT's vocabulary, otherwise we use the same embedding as BERT incase the token is found. We will cover the embedding bridge in detail in subsection 4.2.2.

Finally, with our tokenizer and embedding bridge prepared, we move on to the fine-tuning task. As mentioned before, we need multilingual embeddings for the same word in Hindi and English. Our tokenizer converts hindi sentences to its english 'translations' by replacing each word in the sentence. With this english sentence, we will perform a simple MLM task. For the MLM task, we initialise an MLM head from scratch which we will also train. We will add LoRA adapters to every block of the model, freeze the rest of the model and only update the LoRA weights, instead of fine-tuning the whole model. This would ensure that on successful training, the adapters will be specific to the Hindi-English setting of the task. Ideally, one could train such adapters on any two languages for which you want multilingual embeddings, and 'plug' them into the model.

Now, we can get into each component discussed above in detail.

3.2 Hindi-English Dictionary

The first step to our experiment is creating the one-to-one Hindi to English translation dictionary. Hindi contains more than 180,000 words (Wikitionary 2024) and covering a vocabulary of this size is a complicated task.

Initially, we tried to use publicly available API translation services like Google Translate which can give you accurate translations of a Hindi word. Unfortunately, due to rate limits imposed on the API, it is not practical to do this for a vocabulary of such size. In fact, any API that provides translation services for free was not a valid choice because of the same problem. Therefore, we had to compile various sources to curate a dictionary which covers most Hindi words and their corresponding English translation.

GitHub Repository

The first source we found was a GitHub repository (Eskilson 2015) which provided Hindi words and phrases and their corresponding English translations. From this, we extracted 68471 Hindi words with their corresponding one-to-one translations.

Hindi-English Dictionary from IIT-B

The second source we used was a dictionary provided by Indian Institute of Technology Bombay (IIT-B) (Bhattacharyya 2024b). This source provided two files:

- UW-Hindi Essential (Function) Dictionary: It is a dictionary of a closed category of words or function words, made for the purpose of UNL-based Machine Translation and Multilingual Meaning based Search Engine. This dictionary contained the words that occur frequently in Hindi text. It contained broadly two types of words: function words and commonly used symbols, punctuations, word endings, auxiliaries and some common adverbs. It contained 556 entries till date. The format of the dictionary is shown in figure 3.1:

```
[Head Word]{} 'Universal Concept (icl>restriction)' (Attributes) <H,0,0>;
                    Hindi Example Sentence
```

Figure 3.1: An example for an entry in UW-Hindi Essention Dictionary.

- Universal Word-Hindi Dictionary: This dictionary consists of content words or words belonging to the 'open' class. It has Hindi counterparts of English Nouns, Verbs, Adjectives and Adverbs along with common phrases, idioms and proverbs. It also provides the grammatical, morphological and semantic attributes of Hindi words to make each concept as unambiguous as possible, thus ensuring lexical coherence. It contained 136162 entries till date. The format of the dictionary is shown in figure 3.2:

```
[Headword]{}"UC(icl>restriction)"(Attributes)<H,0,0>;(gloss)
```

Figure 3.2: An example for an entry in Universal Word-Hindi Dictionary.

Considering the features offered by these dictionaries, as we can see from the formats, we had to carefully parse out the information we needed.

Hindi Wordnet

We realised that it would also be useful to collect synsets to expand our vocabulary. IIT-B also provides the Hindi Wordnet (Bhattacharyya 2024a) which is a system for bringing together different lexical and semantic relations between the Hindi words. It is inspired by the English Wordnet provided by Princeton (Miller 1994). They also provide a downloadable version with a graphical user interface to access the Hindi

Wordnet locally. In the file directory, there is a folder called *database* which contains text files in a similar format as the Princeton English Wordnet. We wrote a script to parse these files to collect new words and synonyms of words in our existing dictionary of words.

Most Common Hindi Words from Wikipedia

Finally, our last source is the Wikipedia page of 1900 most common Hindi words (Wiktionary 2024). We parsed out the Hindi and its corresponding English translations to populate our dictionary further with some common Hindi words.

Finally, we cleaned our dictionary and confirmed words are not being repeated, there isn't any punctuation and other measures to ensure there are no discrepancies.

3.3 Custom Tokenizer

Since we decided to not use sub-word tokenizers, we need a custom tokenizer that handles our tokenization needs properly. This implementation will be the first step in handling the gap between Hindi input text and English based NLP models.

At its core, the custom tokenizer is initialised with the dataframe of Hindi-English translations we discussed in 3.2. This foundation allows the creation of a bidirectional mapping between Hindi and English vocabularies. The tokenizer maintains several crucial dictionaries: one for Hindi-to-English word mapping, another for assigning unique IDs to English words, and a reverse mapping from IDs back to English words. We could avoid creating the reverse mapping as it is mainly useful for decoding, that is when we have the model outputs or want to convert the token IDs back into readable text. The same result could be achieved by iterating through the English word to ID dictionary, but this would be inefficient. Also, we used this dictionary extensively while testing the tokenizer, so we have decided to keep it in the implementation incase it is needed for future work. These dictionaries ensure efficient bidirectional translation and encoding processes.

Now we construct the tokenizer's vocabulary. An important point to consider is ensuring the tokenizer's vocabulary contains the special tokens that are critical especially for transformer-based models like BERT. These special tokens- [PAD], [UNK], [MASK], [CLS] and [SEP]- are added into the vocabulary. The role of these special tokens are as follows:

- Padding token [PAD]: This token is used to fill sequences to a uniform length, typically the maximum length in a batch.

- Unknown token [UNK]: This token represents words not in the vocabulary, handling out-of-vocabulary words.
- Mask token [MASK]: This token is used in the MLM task indicating the tokens to be predicted.
- Classification token [CLS]: This token is added at the start of the sequences, often used for classification tasks or to obtain a whole-sequence representation.
- Separator token [SEP]: This token marks the separation between different segments in a sequence, for example, between a question and a passage in question-answering tasks.

Following this, the English translations from the input DataFrame are added to the vocabulary, therefore creating a comprehensive token set that spans both languages and special tokens.

One of the key functionalities of the tokenizer is, obviously, the ability to tokenize Hindi text. We have the option to use two tokenizers from the IndicNLP library (Kunchukuttan 2020):

- *Trivial tokenizer*: This tokenizer simply tokenizes on the punctuation boundaries using Brahmi for Arabic scripts. Major punctuations specific to Indian languages are handled.
- *Morphological tokenizer*: This tokenizer, on top of the trivial tokenizer, uses an Unsupervised Morphological analyser built using Morfessor 2.0. This analyser takes a single word as input and returns a list of component morphemes.

Table 3.1: The output of different tokenizers on Hindi text.

English sentence	I like blue lotus very much
Hindi Sentence	मुझे नीलकमल बहुत पसंद है
Trivial Tokenizer	['मुझे', 'नीलकमल', 'बहुत', 'पसंद', 'है']
Morphological Tokenizer	['मुझे', 'नील', 'कमल', 'बहुत', 'पसंद', 'है']
English sentence	This is a sentence
Hindi Sentence	यह एक वाक्य है
Trivial Tokenizer	['यह', 'एक', 'वाक्य', 'है']
Morphological Tokenizer	['यह', 'एक', 'वाक्य', 'है']

The motivation behind using two different approaches to Hindi tokenization was to experiment with how accurate the MLM predictions would be if we used a morphological tokenizer. The difference in tokenization for two example sentences is depicted in table 3.1. In the first example given in this table, the word 'blue lotus' has been represented by one word in Hindi, which the trivial tokenizer leaves as it is, but the morphological tokenizer divides it into 'blue' and 'lotus'. As we can see in the second example though, the trivial and morphological tokenizers give the same result due to the lack of any words having specific morphological components.

The translation process from Hindi to English is handled as well by referring to the pre-established Hindi-to-English mapping. In case the tokenizer encounters a token not available in the mapping, it simply replaces it with the [UNK] token. Finally, we prepare the tokenized input for BERT. We start by tokenizing the Hindi input text and translating the Hindi tokens to English using our dictionary. We add special tokens, [CLS] and [SEP] at the beginning and end of the sequence, respectively. We convert the English tokens (including special tokens) to their corresponding IDs. If the sequence is longer than a certain pre-defined maximum length, we truncate it, otherwise we pad the sequence with the [PAD] token ID. We then create the attention masks, where 1 represents tokens to attend to and 0 represents tokens to ignore. With these steps, we ensure the input is properly formatted for BERT.

3.4 Embedding Bridge

The Embedding bridge is designed to create a learnable mapping between a custom tokenizer's vocabulary and BERT embeddings. It is initialised with our tokenizer and a specified BERT model (in our case $BERT_{BASE}$). It creates an embedding layer for the custom vocabulary and a mapper network to fine-tune these embeddings. The job of this layer is to return a corresponding vector representation for each token ID input. The mapper network is a simple linear layer with BERT's embedding dimensions as its input and output size. The job of the mapper network is to transform the initial embedding into a space that's more compatible with BERT's embedding space. This is crucial for tokens that are unique to our vocabulary as it helps align their representations with BERT's embedding space.

A key feature is its ability to initialise embeddings for known words using BERT's pre-trained weights as it is redundant to retrain the embedding from scratch for a word in both vocabularies. It does so by breaking it down to subwords and then padding them to the length of the longest sequence in the batch. The padded subword sequences are then passed through the BERT model. The output of the last hidden state which represents the contextual embedding for each subword. These embeddings are then

averaged over the subwords with which we get one embedding for the original token. Finally, the reconstructed embeddings are padded to a uniform length within the batch and normalized using L2 normalization to ensure that all embeddings have unit length.

For tokens which are not in BERT's vocabulary, new embeddings are initialised with a Normal distribution. To train the new embeddings, we implement a training loop which uses MSE loss to align the custom embeddings with BERT embeddings, focusing only on trainable tokens. In the forward pass, it applies the custom embeddings to the input tokens and then selective applies the mapper network only to trainable (custom) tokens. This approach allows the model to maintain the pre-trained BERT embeddings for known tokens while fine-tuning representations of custom tokens. The training process employs the AdamW optimizer with a linear learning rate schedule. The training process also uses attention masks to ensure that padding tokens do not contribute to the loss calculation. Additionally, gradient clipping is applied to prevent exploding gradients which is a common issue when fine-tuning deep networks.

3.5 Custom BERT Architecture

Our architecture provides a sophisticated approach to adapt the pre-trained BERT model for custom vocabularies and MLM. The model includes the trained embedding bridge which is frozen during training, preserving the learned mappings between custom tokens and BERT embeddings. We add LoRA adapters to key BERT components (query, key, value and dense layers) which is represented in figure 3.3. Finally, an MLM head, which is a linear layer, is added on top of BERT's output to predict tokens for the MLM task. BERT also has a pre-trained MLM head, but due to the difference in our vocabularies, using the MLM head BERT provides is not feasible.

For fine-tuning, the embedding bridge uses the input IDs to custom embeddings which are then passed through the BERT model with adapters. The resulting sequence output is used to generate MLM logits which are then used to calculate the loss on the masked tokens.

3.5.1 Masking Strategy

We implement a slightly more complicated strategy for masking as compared to the standard MLM approaches that use uniform masking probabilities. In our case, we employ a method that computes the frequency of each token in the dataset as a dictionary so that rarer tokens more likely to be masked. The method uses inverse token frequencies raised to a power α, which is used to calculate masking probabilities. These probabilities are normalized to ensure consistent overall masking rates.

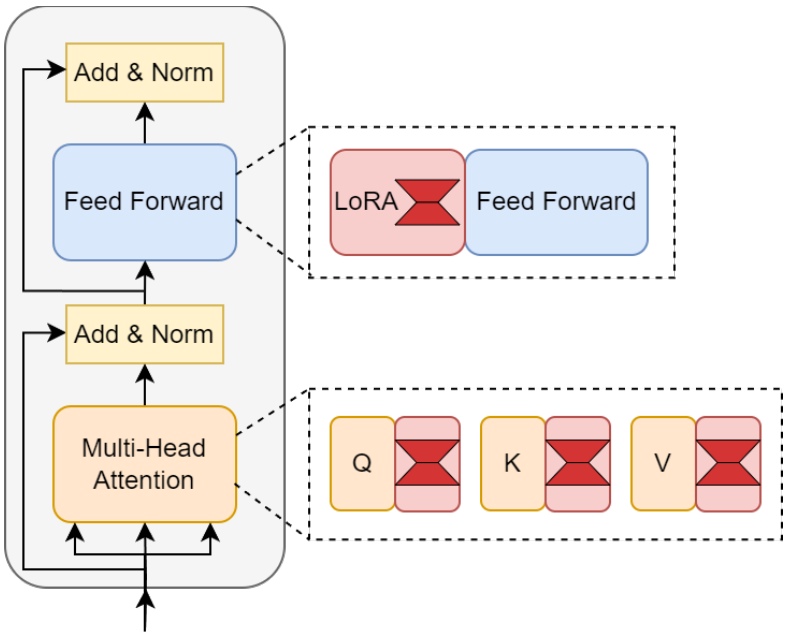


Figure 3.3: Adding LoRA adapters to the standard BERT layer. The modules in red are the parameters that are trained. The rest are frozen. The figure is adapted from Yu, Yang, Kolehmainen, et al. 2023.

The masking procedure follows the standard MLM approach: 80% of the selected tokens are replaced with the [MASK] token, 10% are replaced with random tokens, and 10% are left unchanged.

3.5.2 Fine-Tuning

For optimization, we use an AdamW optimizer with a sophisticated parameter grouping strategy. We implement layer-wise learning rate decay where separate the LoRA parameters by layer, applying a decaying learning rate for deeper layer. This was done to boost convergence and also because lower layers in transformer models often require less adaptation than higher layers when fine-tuning (You, Long, Jordan, and J. Wang 2019). The same logic is applied to the MLM head. A learning rate scheduler with warmup, gradient clipping to prevent exploding gradients and an early stopping mechanism are used here as standard deep learning techniques.

4 Experiments

Now that we have covered how our architecture looks like, in this chapter, we will start by discussing the dataset. We will also discuss the experiments that led to the current architecture of our model. To find the optimal values for the hyperparameters, we will discuss how we conducted the hyperparameter search. Finally, we will cover some experiments which were conducted to see if we could further improve the performance of our working solution. All the experiments, unless specified otherwise, were conducted on the Tesla T4 GPU available on Google Collab.

4.1 Dataset

For this experiment, we use the **HindiEnCorp** dataset, a parallel corpus of Hindi and English (Bojar, Diatka, Rychlý, et al. 2014). This corpus, originally compiled for machine translation tasks, encompasses data from 17 distinct sources. For our work, we would only need the Hindi sentences from this dataset.

The structure of an entry in the dataset was as follows:

- Source identifier: The source of the sentence
- Alignment type: The number of english segments to hindi segments.
- Alignment quality: It can be *manual* which are sentences which were manually aligned, *implied* which are sentences where one side was constructed by translating segment by segment, *float* which is a value reflecting the goodness of the automatic alignment.
- English segments
- Hindi segments

Being a dataset for translation tasks, it had some sources which added just single words or very small phrases specific to a particular domain. This would make sense for a translation task as it covers some words which would be very context specific. For our purpose, we were looking for sentences which were general, complete and sensible.

Therefore, these entries were not particularly useful and we decided to exclude these sources.

Since we are using a curated dictionary for Hindi to English translations, there were cases where multiple Hindi words could translate to the same English word, depending on the context it was used in. For example, the word 'from' in the sentence 'Where are you from?' and 'Are you from Munich?' could have two different words in Hindi. Because of this, we found out that after translation, some common English words in our dataset like 'from', 'of', 'in' ,etc., were more common than other words which would appear in specific contexts. We decided to counter this by token level balancing.

Another issue was unknown tokens as we could encounter a token in the dataset which is simply not present in our dictionary. Therefore, we also implemented a limit on the number of unknown tokens that could appear in a sentence.

To summarize, we ensured quality in our dataset by implementing the following:

- Source selection: We chose 8 out of the 17 sources, excluding those with single words or domain-specific short phrases.
- Quality filtering: Despite not using the dataset for translation purposes, we use alignment quality as a proxy for overall sentence quality. This was done because we noticed that usually sentences with good alignment were generally sentences of good quality as well. We set a threshold of 0.6 for inclusion.
- Token level balancing: Initial experiments revealed token-level imbalances after tokenizing. Therefore, we implemented a scoring function to evaluate sentences based on an overall token distribution. This ensured a more diverse and balanced dataset.
- Sentence Length Control: To main sentence length, we imposed a lower limit on the word count of 3 words and upper limit of 50 words.
- Unknown Token Management: Since our tokenization process uses a dictionary which doesn't change, there could be cases where the sentences from the dataset have many unknown words. To avoid this, we limit the number of unknown tokens per sentence to a range of 0 to 2.

These preprocessing steps significantly impacted the final dataset size. We implemented a *subset* mechanism to control the final dataset size. This was done to create datasets of different sizes as we will use this in our experiments later.

4.2 Reaching the Current Architecture

4.2.1 Creating the Dictionary

In this subsection, we will briefly discuss the experiments we conducted leading up to the final version of the dictionary. Even though in the dictionary that we use for our experiments is simply a one to one mapping between Hindi and English words, the dictionary has had various versions over the course of this project.

After realising that using translation services was not viable, we started looking for more static sources to compile our dictionary. We can start from the beginning with the GitHub repository explained in 3.2. This repository included a Hindi word, an English word and an English grammar column which gave information about the parts of speech of the English word. Quite a few of these Hindi words were symbols, phrases or numerals, which we had to filter out. We also needed to make sure that the length of each entry in this dictionary is one as in some cases, a Hindi word was accompanied by its English definition. We also included parts of speech tags from the NLTK library (Loper and Bird 2002). This was done as we were trying to implement some logic regarding which word we could choose in the case where an English word could have multiple Hindi words, and to see if using the grammar could help.

The next step was to extract data from the Hindi-English dictionary from IIT-B as explained in 3.2. This dictionary had two files, for both of them, we had to parse out valid Hindi and English text. Next, we use the Hindi Wordnet as explained in 3.2 to mainly parse out the head word and the corresponding synsets for that head word. We then created the dataframe for Hindi words and the synonyms of that word, similarly doing the same for English words. We concatenated the two dataframes with the original dictionary we were creating. This resulted in our original Hindi to English translation with two columns for the synonyms of the Hindi and English words respectively. In the current version of the project, synonyms are not being used, but this could be a future point of improvement.

As we discussed the problem of encountering English word which could have multiple Hindi translations, and vice versa, we conducted another experiment to see if we could find a workaround to solve this. Given a sentence (either in Hindi or English), we wanted to see if extracting word embeddings from a multilingual BERT model could help us to decide which Hindi word we should choose for the English word. This would incorporate some contextual information for the word which might help in choosing the right Hindi word. This experiment was conducted in the early stages of the project, and we had not decided exactly how our fine-tuning experiment would look like. Even though for the final fine-tuning task we use Hindi sentences, this experiment was done on English sentences, because of the above reason. One could

do this in the opposite setting, where you do this on Hindi sentences, and use the embeddings to see which English word to choose.

We used the 'bert-base-multilingual-cased' model for this experiment. In hindsight, if we are dealing with embeddings for Hindi words, maybe it would be better to use a model like IndicBERT as it would give better representations because of its training procedure and the fact that it is a specialised model for Hindi. We wanted to keep the embeddings for all the Hindi words in our dictionary, so we went through all the words in the Hindi column, and created a 'Hindi embedding' column. Since we would have an English sentences as input, we then created a function to find the best matching word for each English word. We would simply get the word embeddings for an English word, and in the case where this English word had multiple entries as Hindi words, we use cosine similarity to see if we can get the best Hindi word based on contextual information. As compared to naively choosing one of the entries, using embeddings did help achieving the better choice for the sentence. Nonetheless, since we didn't want to use another model for our fine-tuning task, and wanted to learn representations solely using the data we get and not an external source, we decided to keep this approach on the side.

As discussed in 3.2, we wanted to further expand our vocabulary by including the most common Hindi words. For this, we simply implemented an HTML page scraper function using Beautiful Soup (Richardson 2007).

Obviously, compiling from various sources resulted in some entries with problems. We ensured that there were no duplicates in the dictionary and also performed a regex match to ensure that the Hindi and English columns do not contain anything but their respective characters. In some cases, we saw multiword entries separated by an underscore for example, and not a whitespace. Some sources included transliterations and not translations which were also not needed for our use case. We create test function to check and fix these problems. After a few similar cleaning checks, we ensure to the best of our ability that our dictionary is ready to be used as a one to one mapping between Hindi and English words for our further experiments.

4.2.2 Implementing the Embedding Bridge

Our idea to avoid standard WordPiece tokenization and use our custom tokenizer brought problems as BERT wouldn't know how to interpret tokens from our vocabulary. One of the two ideas we had was to fine-tune the BERT model using our custom tokenizer. This process considering how time-consuming it was, did not feel justified as making updates to the dictionary or expanding our vocabulary would mean training the whole model from scratch again. For this reason, we thought of the concept of an Embedding bridge. The goal here was to somehow bridge the gap between our custom

vocabulary and BERT's vocabulary without having to train the model from scratch.

In the initial experiments, we were initialising new embeddings for every token in our vocabulary. We quickly realised that this was not the most efficient way to handle this, as it took longer to train, and often we would encounter a token in English that would also be present in BERT's vocabulary. We therefore changed our initialisation process. For tokens known to BERT, we initialised our embedding with the one from BERT, otherwise we would initialise new embeddings with a normal distribution. To avoid training these know tokens, we maintained a tensor called *trainable_tokens* which kept track of which tokens needed to be fine-tuned (custom tokens). Therefore, using this tensor, we knew which tokens need to be trained, and for which embeddings can be initialised from the BERT model.

The first key element in our embedding bridge is the *nn.Embedding* module with the size of our custom vocabulary and BERT's embedding dimensions. We plan to basically replace the need of BERT's original embedding layer with this layer. The next key element is the mapper network which is used to finetune embeddings for the *trainable_tokens*. The mapper network is a simple linear layer which learns to transform our custom embeddings into a space that's compatible with BERT's subsequent layers.

To train the embedding bridge, specifically *trainable_tokens* we implemented a training procedure to align our custom embeddings with BERT embeddings. We carefully constructed a function to guarantee the extraction of embeddings from BERT was correct. The function responsible for this functionality ensured that the tokens passed as input were correctly disintegrated into subwords using BERT's tokenizer. It breaks down tokens into subwords, pads them to the length of the longest sequence in the batch, and passes them through the BERT model. After getting the subword embeddings, we average them to obtain embeddings for the original tokens. These reconstructed embeddings are padded to a uniform length within the batch and normalized.

In the forward pass of this bridge, we generate embeddings and generate a *trainable_mask* based on which tokens are to be trained. The mapper network then learns how to transform our embeddings to BERT's embedding for the same token as closely as possible. We normalize the output embeddings to ensure they have unit L2 norm, which is consistent with BERT's embedding properties.

Experimenting with a Different Mapper Network

Our initial mapper network consists of a simple linear layer. We tried to experiment with a more complex mapper network to see its effect on the performance of the Embedding bridge. Even though the linear mapper network worked very well, it would be interesting to see how introducing non-linearity in the mapper network would affect the results.

In the second version of the mapper network, we use a sequential module which uses a linear layer to transform the input from BERT embedding dimension to twice the size, performs layer normalization, applies ReLU and then transforms the data back into the original dimension, normalizes the output of the second linear layer thus giving us the final output.

We took a dataset of 5000 sentences to train our embedding bridge, and trained both versions. The training loss statistics are available in Table 4.1

Table 4.1: Different mapper networks for the Embedding bridge

Mapper network	Loss after 1 epoch	Loss after 2 epoch
Linear	1.29e-05	0.9e-05
Sequential module	0.4e-05	0.3e-05

The complex mapper network does seem to outperform the Linear mapper. Nonetheless, we continue to use the Linear mapper in our architecture. The performance is still very good, and we can manage without introducing unnecessary complexity and more parameters.

The reader might be wondering as to how the 'replacement' of BERT's embedding layer happens. It would be correct to assume that the BERT model we will use for fine-tuning still has its embedding layer intact, and we have not explicitly removed it. It could be an improvement in the model architecture, but from a functionality perspective, it does not change anything. We do create a *CustomBERTModel*, which is a wrapper around the BERT model which has the embedding bridge, but the model itself still has its embedding layer.

In the early implementations, since we were initialising a BERT model in the Embedding bridge class, and then using this bridge in our *CustomBERTModel* wrapper, it resulted in two BERT models in our wrapper class. This was unnecessary as the BERT model for the embedding bridge was used only to retrieve subword embeddings. The model we are fine-tuning was not the same model, and there was no need for this BERT model to be included in the final architecture. Although, one could use this model itself instead, but we decided to keep it separate to not introduce complexity and keeping all our classes abstract and understandable by themselves. Even though our implementation worked, this is a confusing architecture to wrap your mind around. To fix this in a simple way, we constructed another class, called the *OptimizedEmbeddingBridge*, which simply included the embedding layer and the mapper network. This therefore reduced the size of our model and also reduced the complexity of the final architecture.

Finally, to conclude this subsection, we would like to present an example of the output from the Embedding bridge. We take a test sentence in Hindi, we tokenize

it using our custom tokenizer, and we try to find the closest 5 BERT tokens for each token in our sentence. We do this by retrieving BERT embeddings and for each custom embedding for each token in our example sentences, we calculate the cosine similarity with all BERT embeddings. We then take the top 5 values and retrieve the actual tokens. This would help us to see if the embedding bridge is actually generating meaningful embeddings for our tokens by comparing them to the actual BERT embeddings. For this example, we will show the top 3 examples to not clutter the table. The sentence we used is shown in the Table 4.2

Table 4.2: The sentences used to test the Embedding bridge.

Hindi Sentence	English Translation	English output using tokenizer
यह एक परीक्षण वाक्य है	This is a test sentence	['it', 'one', 'try', 'sentence', 'is']

The top 3 closest BERT tokens for each token is shown in Table 4.3 Now that we have

Table 4.3: The top 3 tokens for each token.

Token	Closes 3 tokens (token:str, similarity:float)
it	(it: 1.0), (this: 0.6295), (that: 0.5771)
one	(one: 1.0), (two: 0.5523), (1: 0.5319)
try	(try: 1.0), (tried: 0.6837), (tries: 0.6390)
sentence	(sentence: 1.0), (sentences: 0.8209), (sentenced: 0.5938)
is	(is: 1.0), (was: 0.7542), (are: 0.6747)

successfully covered how our tokenizer handles input data, and how the Embedding bridge ensures we have valid embeddings for the BERT model, we can now see how we implement our custom BERT model.

4.2.3 Implementing the Custom BERT Model

The initial implementation of the wrapper class included using Bottleneck Adapters as explained in 2.2.3. Even though being a very simple representation, we kept it this way to ensure that we have a working solution. Once we saw that the training loss is decreasing, we decided to experiment more. We injected these adapters in between every layer of the encoder in BERT. Therefore, for each layer's output, we would have a weight and bias for the downwards projection and the upward projection. The second implementation was using LoRA adapters with some standard parameters for the

adapters. In this case, for each layer, we had two LoRA matrices, A and B, for the query, key and value layers in the attention blocks, for the intermediate dense layers and the output dense layers. Considering how much more complex this architecture was as compared to the bottleneck adapters, we expected it to improve performance. Keeping all other parameters the same, we trained our model for 7 epochs using a dataset of 10000 sentences to evaluate difference in training loss. We were proven right, which can be seen in figure 4.1. We therefore decided to continue our experiments with LoRA adapters instead of the standard Bottleneck adapters.

Figure 4.1: Comparison in performance between Bottleneck and LoRA Adapters

4.2.4 Improving the Fine-Tuning Process

After choosing the right adapter architecture for the fine-tuning experiments, we moved on to further improve this process. Early results from fine-tuning experiments showed that the model is still leaning to predict tokens which come up more often that others as explained in 4.1. Even though we tried to fix this while sampling sentences from our dataset, and we did improve it to a certain extent, there was still a small subset of tokens which were tending to appear more often that others. To fix this problem in the fine-tuning process, we decided to change the masking strategy.

Improving the Masking Strategy

As covered briefly in 3.5.1, we initially had the masking strategy which is the most standard and commonly known MLM approach. This strategy uses a uniform probability (called MLM probability) for all tokens. Every token, regardless of its frequency in the dataset has an equal chance of being masked. The masking strategy then includes the general approach of masking, which is 80% of the times we mask the token, 10% of the times we put random token and 10% of the times we leave the token unchanged.

Since we want to consider the frequency of tokens in this strategy, we start by creating a hash table which contains token frequencies across the entire dataset before the masking is applied. We then calculate the inverse frequency for each token, which is used to determine the masking probability. Therefore, tokens which are more frequent than other are masked less often, and vice versa. We can control the impact of token frequency on masking probability by a parameter we set as α. If we increase α, the effect of the frequency in amplified, therefore increasing the probability of masking rare tokens. By adjusting this parameter, we can control the balance between learning rare words and maintaining a good representation of common words. The probability matrix is then created using the normalized inverse frequencies for each token. Finally, the same masking strategy (80-10-10) is applied, but unlike the first approach, it is applied to a non-uniform probability distribution.

More concretely, the advantages of our approach can be highlighted with the following points:

- Balanced learning and Reduced bias: The model can learn rarer tokens better, and it has a reduced bias towards tokens which appear frequently. Since the model is challenged to predict a more diverse set of tokens, this should lead to better generalization.
- Control over frequency impact using the α parameter: This parameter allows us to control the effect of the token frequency on the masking probability. If $\alpha = 0$, we revert to our old function with uniform masking probability, and as we increase α, the impact becomes more visible. We have set this parameter as $\alpha = 0.7$. Further experiments could be conducted with this parameter to see how it depends on the dataset or the task requirements.

Once we implemented this masking strategy, we instantly saw improvement in the model's predictions as it started giving diverse outputs for the masked tokens instead of just the most frequent tokens in the dataset.

Hyperparameter Search

To further improve the model performance, we decided to experiment with different values of the learning rate and warmup. This was done to see that optimal parameters for learning before we start experimenting with other components, for example the LoRA adapters.

We conducted a hyperparameter search for a total of 9 combinations formed from 3 learning rates: $1e-05$, $1e-03$ and $1e-02$, and warmup ratio values of 0.1, 0.2 and 0.3. The results of this experiment are depicted in figure 4.2.

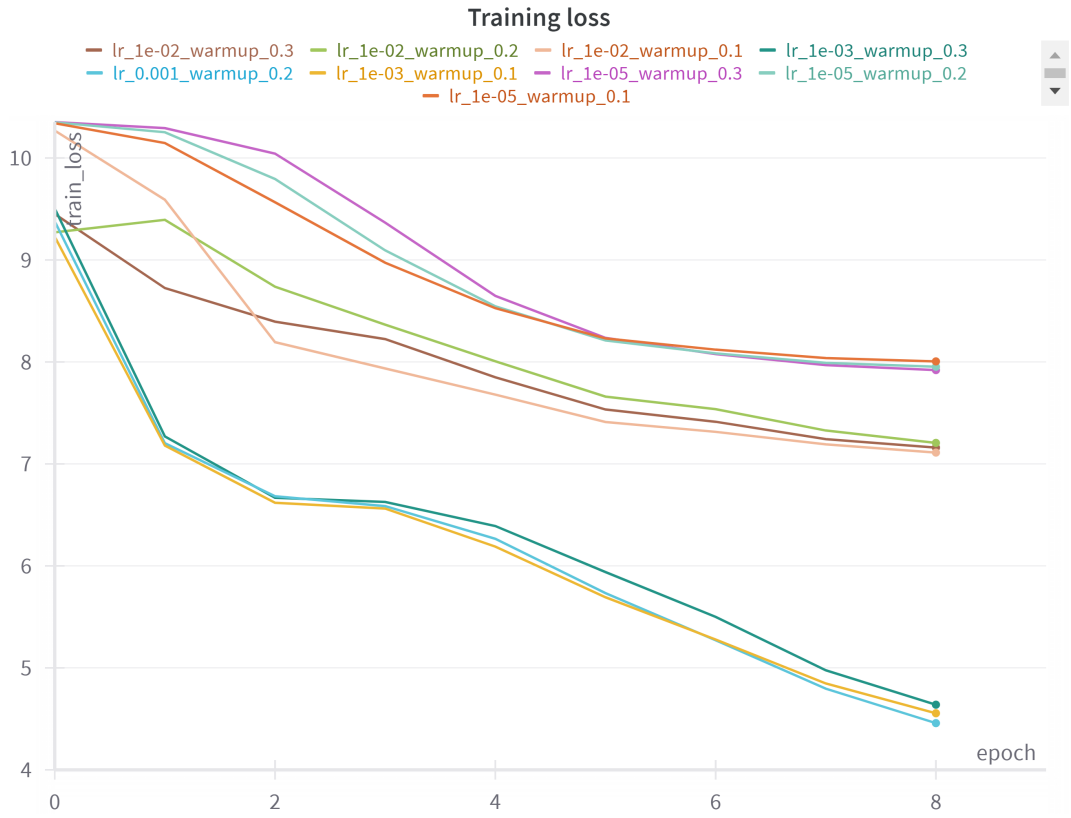


Figure 4.2: Hyperparameter Search

We learnt from this experiment that the ideal values for the learning rate is $1e-03$ and warmup ratio value of 0.2. Note that this learning rate is the same for both LoRA and MLM head. Since our MLM head is initialised from scratch, it could be possible that the MLM head might require more aggressive updates as compared to the LoRA

parameters. For this reason, we decided to have separate learning rates for the LoRA and MLM head parameters to see if we can further improve the model performance.

Optimizing the Optimizer

Instead of initialising an AdamW optimizer in our training function, we decided to separate this logic into another function responsible to create a custom optimizer. We first iterate through each layer of the BERT encoder and perform LoRA parameter and MLM head *parameter grouping*. This process is explained with the following steps which were performed for both LoRA and MLM head parameters:

- Two parameter groups are created, the first one being the LoRA weights excluding biases and LayerNorm weights, and the second group containing biases and LayerNorm weights.
- For the case of LoRA parameters, each group gets a custom learning rate, which decays for earlier layers. The custom learning rate can be represented as,

  $$lr_{current\_layer} = lora\_lr * (lora\_lr\_decay * *(num\_layers - current\_layer - 1))$$

  For the case of MLM head parameters, both groups use the mlm_lr.
- Weight decay is set to 0.01 for weights and 0 for biases and LayerNorm parameters

In the above steps, $lora\_lr$, $lora\_lr\_decay$, mlm_lr are passed as parameters to the function. Finally, all parameters are grouped and an AdamW optimizer is created.

With these steps, we ensure that we apply different learning rates to LoRA and MLM head parameters. For LoRA parameters, we implement a decay due to which earlier layers have lower learning rates than later layers (You, Long, Jordan, and J. Wang 2019). Weight decay is implemented to weights but not to biases and LayerNorm parameters, as explained in Loshchilov and Hutter 2017.

Due to our hyperparameter search, we know that a learning rate of $1e-03$ for all parameters worked well. Considering we want to update the MLM head parameters more aggressively, we thought a good starting point for further experiments would be to have a learning rate of $1e-02$ for the MLM head parameters, and $1e-03$ for the LoRA parameters. To observer the difference, we can compare the best configuration from the hyperparameter search to the starting point we discussed. The result can be seen in figure 4.3

We see a significant improvement from separating the learning rates for the different set of parameters. Based on this observation, we decided to experiment further with other values for the learning rates. For the following experiments, we use a dataset

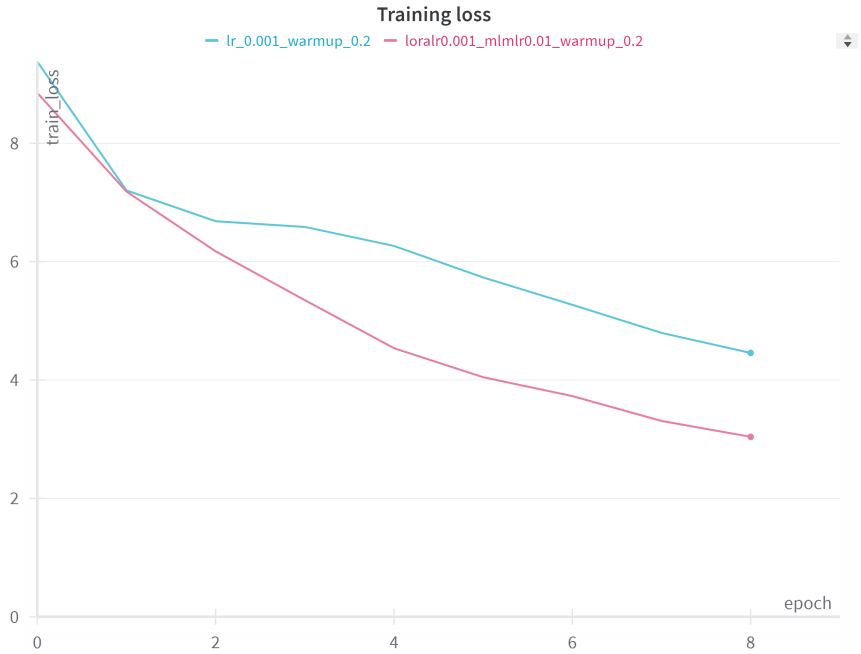


Figure 4.3: The best configuration from the Hyperparameter search with the same learning rate for LoRA and MLM head compared to having different learning rates.

of 5000 sentences and train for 15 epochs to see the performance of the model over a longer period of time. Firstly, we change the LoRA learning rate values while keeping the MLM learning rate the same ($1e-02$). We run the cases for which the LoRA learning rate values are one of the following- $1e-02$, $1e-04$, $5e-03$ and $5e-04$ with $1e-03$ being the case we have covered before. Then we also run cases where the LoRA learning rate is kept at $1e-03$, but the MLM head learning rate is either $1e-03$ or $5e-02$. The final configuration we tried was LoRA learning rate at $5e-04$ and MLM head learning rate at $5e-02$.

The results of all these configurations are presented in figure 4.4. To give a better understanding of the top 5 runs in figure 4.4, we have summarized the results in the table 4.4.

The best results come from the configuration the LoRA and MLM head learning rates being $5e-04$ and $1e-02$ respectively. Our baseline learning rates of $1e-03$ and $1e-02$ for the LoRA and MLM head respectively come in as a very close second.

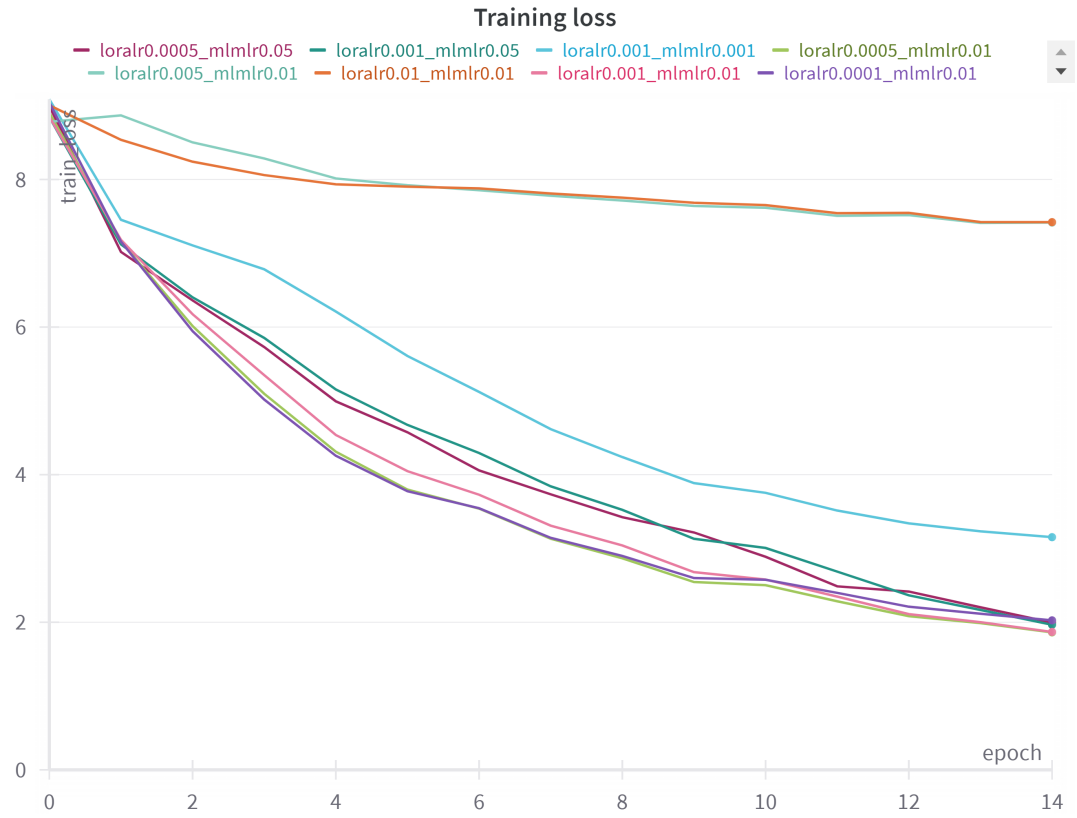


Figure 4.4: Experimenting with different values of LoRA and MLM head learning rates.

Testing Different Tokenizers

Now that we have a decent understanding of which set of hyperparameters work well, lets test the two different tokenizers for experiment. Up until now, we were using the morphological tokenizer for our experiments. We keep the LoRA and MLM head learning rates being $5e-04$ and $1e-02$ respectively from our last experiment. The difference in training loss is depicted in Figure 4.5. We can see that the trivial tokenizer actually outperforms the morphological tokenizer in training loss.

This could be due to the fact that our dictionary is composed of more complete words than words divided into their morphological segments. Even though there are a few instances of morphs being present in the dictionary, all the sources the dictionary was compiled from was mostly for full words. This would explain as to why the training loss for the model using the trivial tokenizer is lesser than the morphological tokenizer.

Table 4.4: Summarizing top 5 results from figure 4.4

Configuration (LoRA-LR, MLM-LR)	Final Training loss value
5e-04, 1e-02	1.86246
1e-03, 1e-02	1.86908
1e-03, 5e-02	1.9656
5e-04, 5e-02	1.99614
1e-04, 1e-02	2.02539

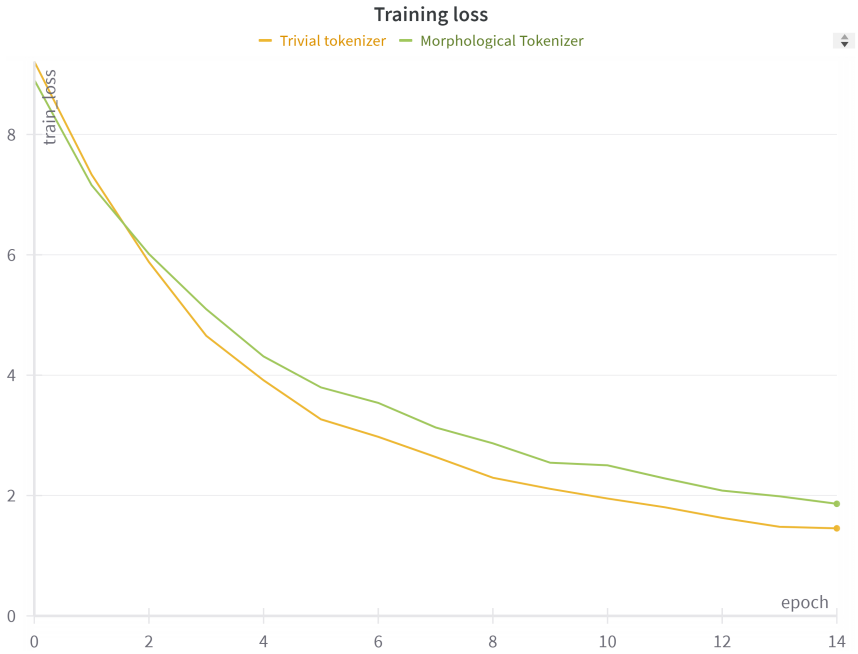


Figure 4.5: Difference in tokenizers

Nonetheless, since our goal is to evaluate results after using a morphological tokenizer, we will continue to do so in the following experiments.

Experimenting with Larger Datasets

Up until now, we have experimented on small datasets of 5000 sentences. We wanted to see how increasing the dataset size affect the training loss, and we will also observe the difference in performance of these models while evaluating them. We train the

same configuration of LoRA and MLM head learning rates being $5e-04$ and $1e-02$ respectively, using 10000 and 20000 sentences. The results are shown in figure 4.6.

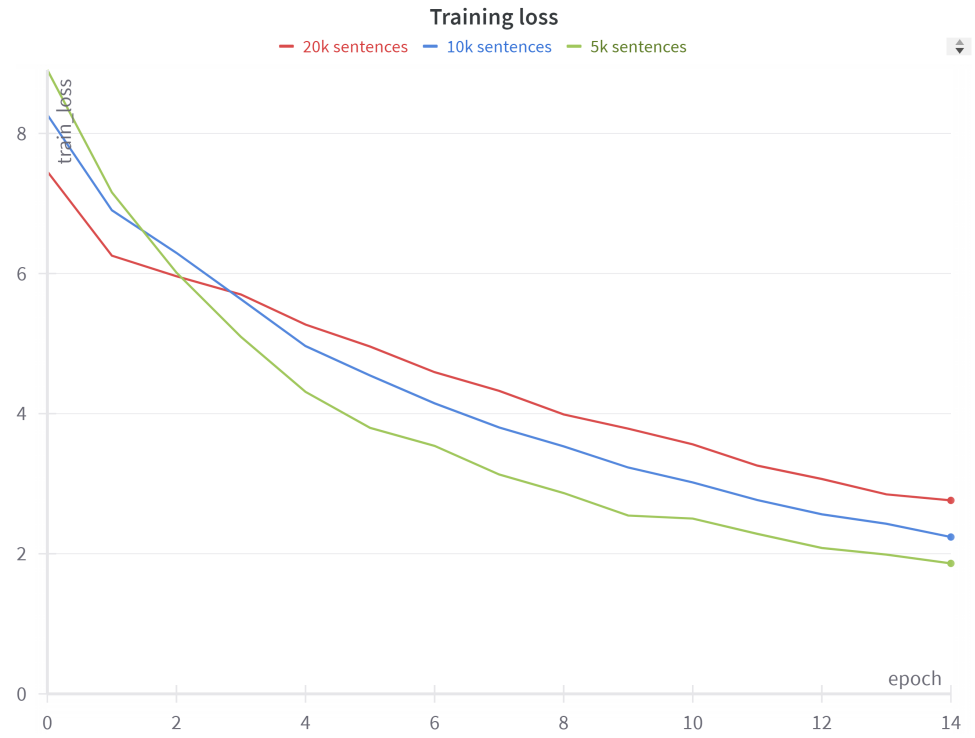


Figure 4.6: Experimenting with different dataset sizes.

As we can see, the model trained on 5000 sentences works better than the rest. But, this does not necessarily mean that this model is a better model than the others. The dataset, being a dataset prepared for MT tasks, ensures variety in sentences instead of being a balanced dataset in terms of its vocabulary. This is simply because, a highly heterogeneous dataset is beneficial for MT tasks, as it's better to cover as many words in different contexts as possible. For our case, we want to cover as much vocabulary as we want as well, but this brings up a problem for evaluating the models using validation sets. Having a validation dataset wouldn't work as it wouldn't be representative of the training dataset due to the high variety. It could also be the case that the models being trained on a larger dataset might have a better representation of the vocabulary, but their training loss is higher because they simply need more epochs.

Changing LoRA Parameters

Usually, having a higher value of r in the LoRA adapters, means more parameters to train. It allows the adapters to capture more complex transformations and potentially learn better. But, since we are increasing the number of parameters to optimize, it can also lead to suboptimal solutions and slower convergence in some cases. To see, how changing r and α would affect the model, we decided to take the model fine-tuned on 10000 sentences with different LoRA configurations. We tried three configurations (r, α): (8,16), (16,32), (32,64). The (r, α) = (16,32) is the configuration we saw in figure 4.6.

The comparison between these three configurations can be found in the figure 4.7.

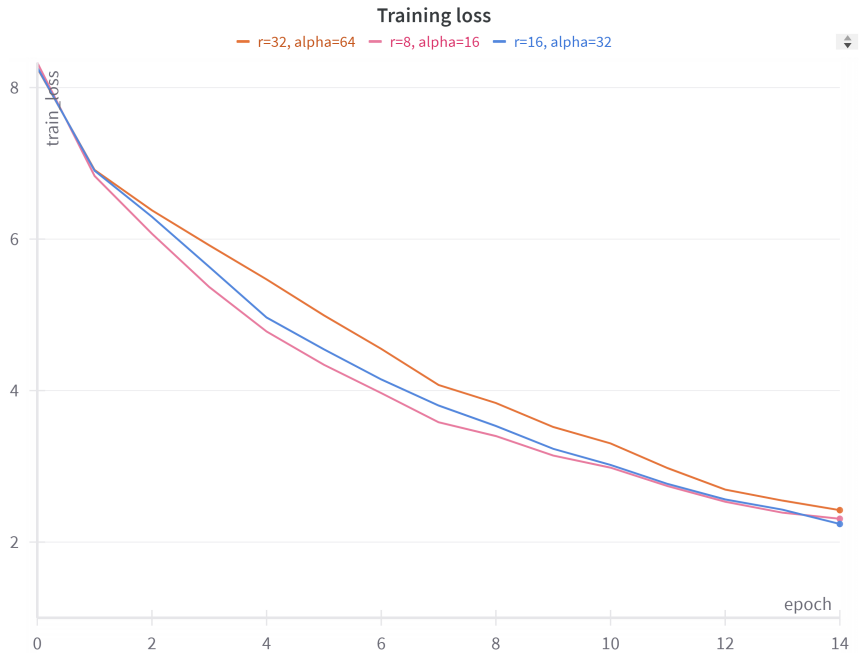


Figure 4.7: Experimenting with LoRA parameters for a model trained on 10000 sentences.

Even though the other configurations are close in performance, our base configuration works the best. Another key point to note is that the best performing configuration would change based on the dataset size, as capturing more complex features would ideally require larger low-rank matrices.

Training the MLM Head

The MLM head we have in our model is initialised from scratch. We considered using BERT's MLM head, as it has been trained on a large amount of data (Devlin, Chang, Lee, and Toutanova 2019), but due to our vocabularies not being compatible, this was not an option. Thus, we decided to take a small chunk of the original data BERT was trained on, and train our MLM head using our tokenizer to see if it would improve the performance of our model. We take 10% of the dataset, and train the MLM head with a learning rate of $1e-4$, 2500 warmup steps, and 250000 total steps. This experiment was conducted on an NVIDIA GeForce RTX 3080 Ti GPU.

Finally, after training the MLM head, we resume the process of conducting our experiment as before with LoRA and MLM head learning rates being $5e-04$ and $1e-02$ respectively. The results are depicted in figure 4.8.

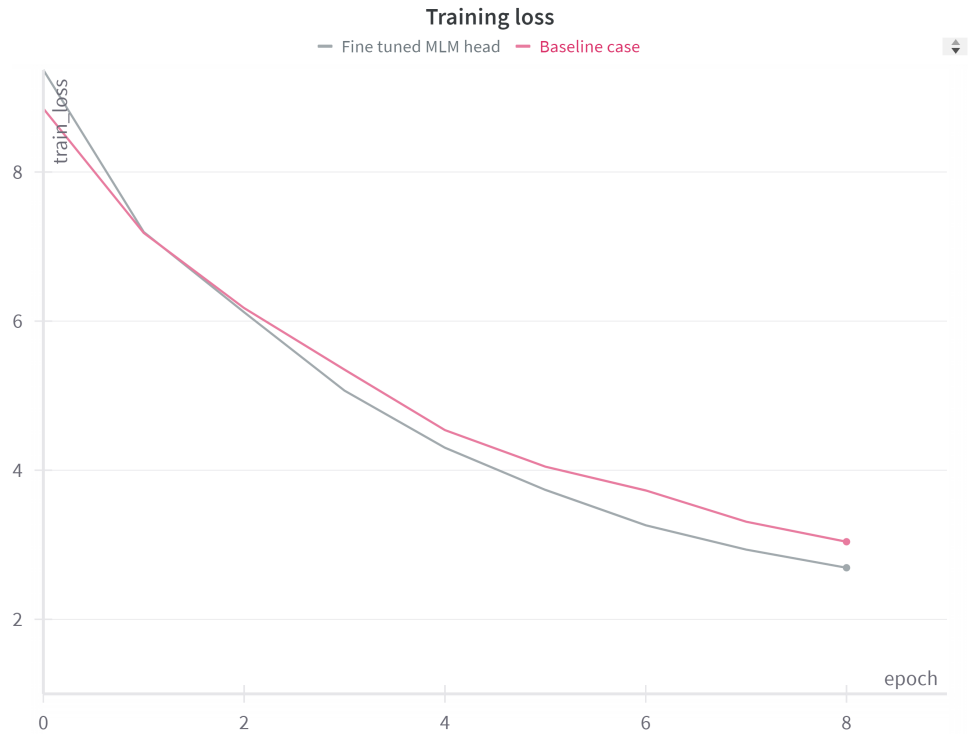


Figure 4.8: Observing the difference in performance after training the MLM head.

Training the MLM head on a very small subset of BERT's original training dataset has already improved the model's performance. Further training using more data

and longer time periods could definitely result in better performance for the same hyperparameter configuration.

Training the Model with 50000 Sentences

We tried to experiment with the learning rates we have adjusted over the past few experiments, and train the model with 50000 sentences. This experiment was conducted on an NVIDIA GeForce RTX 3080 Ti GPU. We can see the result in figure 4.9.

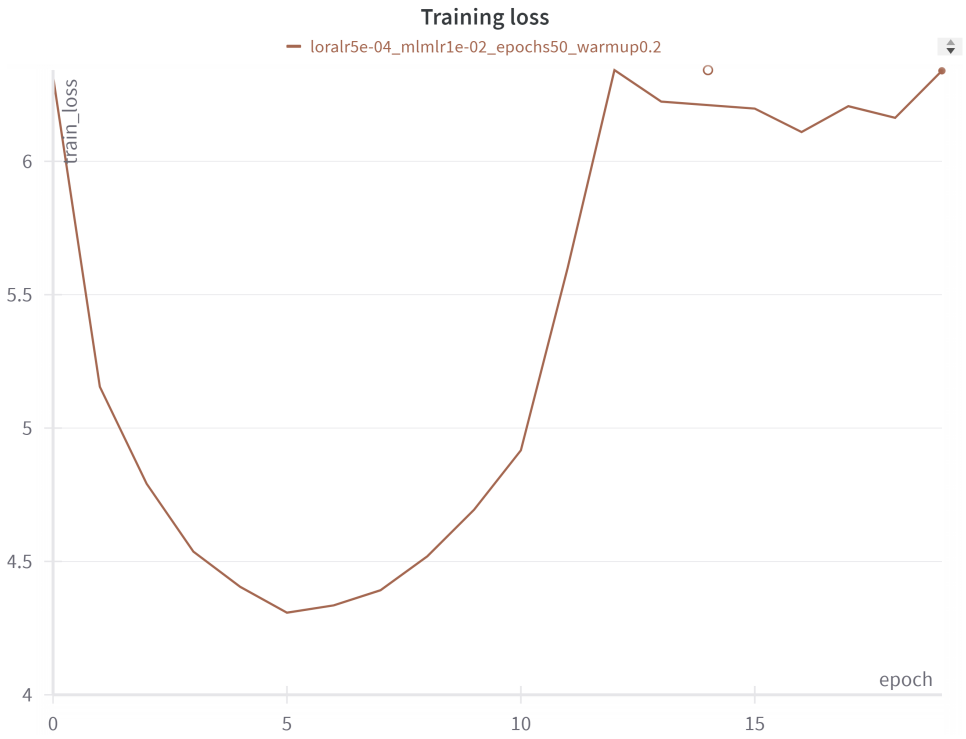


Figure 4.9: Training the model using 50000 sentences.

We saw the model learn upto a certain point, reaching a minimum training loss value of 4.30, but soon enough, the training loss increased very fast. This sharp increase is a classic sign of overshooting due to our aggressive learning rates. The model is taking steps that are too large, causing it to jump over optimal areas in the loss landscape. This patter leads to training divergence, where the model fails to converge to any stable solution, which we see after epoch 12.

The most important conclusion here is to have a separate hyperparameter search to

find the optimal learning rates when using a dataset of this size. For smaller datasets as in our experiments, larger learning rate values worked, but we can clearly see this is not the case for a larger dataset. We couldn't experiment further and investigate this failed experiment due to lack of time, but finding optimal learning rates for this experiment could definitely lead to a model with a better understanding of a more diverse vocabulary.

4.3 Final Experimental setup

After various experiments, lets quickly cover the configurations we will use for benchmarking our model. We will cover some specifics about additional features that were implemented in the existing training approaches, for the sake of completion.

Dataset

We will cover evaluation for 2 models: a model fine-tuned on 5000 sentences and one on 10000 sentences. From now on, we will refer to these models as **5k** and **10k**.

Training the Embedding Bridge

The embedding bridge is trained for 2 epochs with a learning rate of $1e-4$. We also implement a learning rate scheduler which implements a gradual reduction in learning rate. It is known to potentially improve in model convergence (You, Long, Jordan, and J. Wang 2019) and is a common practice in deep learning. For the same reason, we also implemented gradient clipping for the parameters being trained in the embedding bridge. We set the maximum L2 norm of the gradients to 1.0. This was done to prevent exploding gradients and again, is a common practice in deep learning. The MSELoss is used as the loss function for the embedding bridge.

Custom BERT Model

As we saw in subsection 4.2.3, using LoRA adapters is a better choice than Bottleneck adapters. We use LoRA adapters with $r = 16$, where r is the Rank which determines the number of parameters in the LoRA adapter. We set $\alpha = 32$, which is a scaling factor that affects the magnitude of the LoRA update. We add these adapters to the query, key, value and dense layers of the BERT model. We freeze the embedding bridge an original BERT parameters to only train the LoRA adapters and MLM head. Even though training the MLM head separately is a promising approach, for the purpose of

benchmarking, we will continue with our original approach of initialising our MLM head with a normal distribution for weights, and biases as 0.

Fine-Tuning

The masking strategy, optimizer and fine-tuning loop is implemented as explained in 4.2.4. Just like in the embedding bridge, we implement a learning rate scheduler here, but with warmup. We also implement gradient clipping for the potential problem of exploding gradients. Finally, we fine-tune the model and for our evaluation, we have chosen the two best models from 4.6, which are the 5k and 10k respectively.

4.4 Evaluation

With our 5k and 10k models, we will now begin the process of evaluation. As briefly discussed in 4.2.4, evaluation for a model where the dataset is extremely varied is complex. Since our validation set was not representative of the training examples, our models scored low on validation checks. Thus, we decided to train the model on the dataset till we see a low training loss. To gauge if the model has developed a decent understanding of the relationship between Hindi and English words, we use the IndicGLUE datasets to fine tune these models and see how the performance of our model compares to the IndicBERT model. The original evaluation results (Kakwani, Kunchukuttan, Golla, et al. 2020) are also available for the tasks we are going to include from the benchmark.

We benchmarked our model against a fine-tuned IndicBERT model on COPA, Sentiment Analysis, Category Classification and Discourse Mode Classification as covered in 2.2.6.

Table 4.5: Evaluation results

IndicGLUE Task	IndicBERT	**5k**	**10k**
COPA	58.33	47.73	42.05
IITP Movie Reviews	59.03	51.29	53.23
BBC News Classification	74.60	74.25	74.20
Midas Discourse	78.44	70.21	69.41

The evaluation results presented in Table 4.5 provides valuable insights into the performance of our 5k and 10k models compared to the IndicBERT baseline. Some key observations are as follows:

- COPA Task: Both the 5k and 10k models underperformed compared to the IndicBERT model fine-tuned on this task, with the 10k model showing slightly lower performance than the 5k model. This suggests that our models may struggle with the reasoning required for the COPA task.
- IITP Movie Reviews: Being a sentiment analysis task, and simpler than the COPA task, we see our models show promising results. While still lower than the IndicBERT model, the gap is smaller compared to the COPA task. Increasing the dataset size for the trained model gives us a slight advantage, which might indicate the benefit of training the model on more training instances.
- BBC News Classification: Both our models performed remarkably well, nearly matching IndicBERT's performance. This indicates that our architecture is capable of handling this type of classification task.
- Midas Discourse Task: Our model shows a decent performance on this task. This task involves understanding the discourse structure and relationships within text, which is a complex NLP challenge. The performance gap between our 5k and 10k models with IndicBERT suggests that while our models have developed some discourse understanding capabilities, there is still room for improvement. The 5k models interestingly slightly outperforms the 10k model which align with the trend observed in the COPA task.

The evaluation results clearly show the capabilities of our model when it comes to having a decent understanding of the language. Note that for the downstream tasks, we were using the custom tokenizer for our models. This goes to show that there is a satisfactory level of understanding between Hindi and English for our models. The strong performance on the BBC News Classification task is encouraging, suggesting that our model has developed a decent understanding of the Hindi language structure. Nonetheless, for complicated tasks (COPA and Midas), the model tends to degrade in performance. Also, the 5k model on these complicated tasks outperforms the 10k model which could indicate that increasing the dataset size may not always lead to better performance, especially for tasks requiring a deep understanding of the language. Clearly, the level of understanding is not at a level of perfection, and there is still room for improvement.

5 Discussion and Future Work

In this section, we will discuss the results of our experiments, and summarize them with a brief list of points for future work.

The experiments conducted in this thesis aimed to reduce the computational cost of multilingual in LLMs through the use of universal subword embeddings, specifically for the case of Hindi and English. Our approach involved creating a custom Hindi-English dictionary, implementing a custom tokenizer handling our tokenization needs with the support of a Hindi morphological tokenizer, designing and training an embedding bridge, and fine-tuning a BERT model using LoRA adapters.

The custom architecture, with the various components, showed promising results in terms of performance. By avoiding the classic subword tokenization algorithms and instead relying on a curated Hindi-English dictionary, the model was able to develop a decent understanding of Hindi while still handling the morphological richness of Hindi.

The process of creating comprehensive Hindi-English dictionary from various sources, including GitHub repositories, IIT-B dictionaries, and Hindi Wordnet, proved to be a crucial component in the project. The dictionary used whole-word tokenization with a hint of morphological tokenization, instead of using classical subround tokenization algorithms which presented both challenges and opportunities. While it allowed for better preservation of the morphological structure of words, our dictionary falls short of completion. As we saw in our experiments, the model with morphological tokenization performs slightly worse than the one with trivial word based tokenization. We believe that it is to do with the dictionary not having enough morphological information to accommodate for perfect tokenization.

The embedding bridge proved to be an effective solution for bridging the gap between our custom vocabulary and BERT's vocabulary. By initialising known tokens with BERT's embeddings and learning new embeddings for custom tokens, the model was able to leverage pre-existing knowledge while adapting to our specific task. Our experiment with different mapper functions showed how we can benefit from a more complicated mapper function with the trade-off of adding more parameters. Nonetheless, the results from the embedding bridge were more than satisfactory as it successfully was able to assign correct embeddings for the tokens in our vocabulary with sufficient training.

The use of LoRA adapters demonstrated clear advantages over bottleneck adapters. This confirms that LoRA can provide more efficient fine-tuning with fewer parameters (Hu, Shen, Wallis, et al. 2021). The experiments with different LoRA configurations also provides insights into the trade-offs between model capacity and performance, with the $r = 16$ $\alpha = 32$ configuration proving to be the optimal choice for this task in the case of the 10k model.

We also observed the relationship between the dataset size and the model performance. The experiments with 5000, 10000, and 20000 sentences revealed that simply increasing the dataset size did not always lead to improved performance for the same amount of training. Nonetheless, the use of HindiEnCorp dataset, originally compiled for Machine translation tasks, introduced both benefits and challenges. While it provided a diverse set of sentences, the heterogeneity of the dataset did not allow us to use standard validation checks. Therefore, we suspect that training models with a bigger dataset would need more epochs to give better results. And in doing so, it could also result in the model having a better understanding of a more diverse set of tokens. The 5k model often outperformed the 10k model on complex tasks suggesting the quality and diversity of our dataset might be more important than the quantity. Again, we would need to conduct further experiments with larger dataset to know if this is a fact.

The implementation of a frequency based masking strategy was a crucial improvement over uniform masking. By adjusting the masking probabilities based on token frequencies, our model was encouraged to learn a more diverse set of tokens. Even though the α value in our experiment to control the impact of the frequency on masking probabilities turned out to be a good choice, further experiments taking advantage of this flexibility of choosing the right value could help in getting better results.

Our decision to have different learning rates for the LoRA adapters and the MLM head proved to bring a great improvement in our training procedure. This suggests that the different layers do have different learning dynamics. Also, the implementation of layer-wise learning rate decay for LoRA parameters showed varying adaptation needs across the different layers LoRA was applied to. As mentioned above, training models with larger dataset would probably require a separate hyperparameter search. We couldn't incorporate these experiments in this thesis, but it would be interesting to find a relationship between the increasing dataset size and the change in hyperparameters. That being said, even though we found a well performing configuration for 5000 sentences, doing this for a larger dataset with different configurations could result in a model with a better understanding of Hindi and a larger vocabulary. This also applies to our experiments with LoRA parameters. For the case of 10000 sentences, we found the sweet spot to be $r = 16$ and $\alpha = 32$. Other values showed underfitting or optimization difficulties for the model. For 5000 sentences, the same configuration worked well, although future experiments with other configurations having smaller r, α

could also work.

Our experiments with training the MLM head beforehand also showed that the performance of the model can improve if we train the MLM head beforehand. Even though this process is time-consuming, it would be worth to train the MLM head on more data, and for longer, to see the extent of improvement as compared to our baseline models.

Our evaluation results proved to be a great way to understand our model's current capabilities. The lower performance on the COPA task highlights that our model faces challenges in complex reasoning. The improved performance of the 10k model in the IITP Movie reviews task suggests that increased data can benefit certain types of language understanding tasks. Our best performance was on the BBC News Classification tasks. This proves the model's strong capability in content categorization. But finally, another difficult task, the MIDAS discourse task, showed the model's inability to capture some aspects of discourse structure. We see a similar trend of the 5k model performing better than the 10 model, as seen in COPA, reinforcing the idea that data quality might more crucial than quantity. While not consistently outperforming IndicBERT, the custom model showed competitive performance especially considering the smaller pre-training dataset.

5.1 Future Work

In this section, we take insight from the discussion of our experiments, identify key limitations and propose areas for future work.

Dictionary Expansion and Tokenizer Improvement

Even though we compiled various sources for the purpose of having as many Hindi words as possible, more work can be put into incorporating translated morphemes into the dictionary. This is more complicated than it sounds, as disintegrating words into morphemes can sometimes lose information. There are multiple cases in which the word as a complete form makes sense, but its individual morphemes combined do not mean the same thing. Nonetheless, including some sort of morphological information in the dictionary, whenever available, could definitely boost the performance of the model. This would also include some improvement in the tokenizer as to if it supposed to handle some morphemes differently or not.

Another direction for future work would be incorporate the synonyms and POS tags of Hindi and English words. Finding how the grammar of these both languages could help in selecting which word to choose of one language when there are multiple translations in the other language, could definitely improve results.

Different Datasets

The HindiEnCorp was originally meant for Machine translation tasks, and the challenges of using it have been covered in 5. One direction to go in would be to find a more balanced dataset with high variety. There are ofcourse more dataset available in the Indic Suite (Kakwani, Kunchukuttan, Golla, et al. 2020), but it is crucial to go through the dataset in detail to see what value it provides. We excluded quite a few sources due to the fact that they did not provide any value, or were too context specific. Another direction would be to obviously to create a dataset. This is extremely time-consuming, but would provide the best results, as a dataset with sufficient vocabulary covered and a decent validation and test set which includes words seen in training, would be ideal. Therefore, a systematic study on how model performance scales with dataset size could help establish a set of guidelines for training models on larger datasets. This would also help to understand why the 10k model sometimes underperformed as compared to the 5k model.

Embedding Bridge

Even though our version of the embedding bridge provides good results, one could experiment with different mapper network in terms of performance, time to train, or other metrics to choose the best option.

MLM head Pre-Training

As seen in figure 4.8, training the MLM head before our fine-tuning process resulted in better performance. Nonetheless, training the MLM head on the same data BERT was trained is an extremely time-consuming and computationally expensive. Because of this reason, we couldn't conduct further experiments, but we were glad to see that the performance does improve even if the MLM head is trained for just 10% of BERT's training dataset.

Another possible solution could be transferring the knowledge of the pre-trained MLM head from BERT, similarly to what we did with the Embedding bridge. This could result in an 'MLM head bridge' which transfers the knowledge from BERT's MLM head to ours for a specific set of tokens. This solution, if viable, could possibly save alot of time, effort and computational resources. Ofcourse, research and experiments would need to be conducted to see if such a solution would be possible.

Further Experiments with Larger Datasets

The hyperparameters we used for our experiments were the best for a model fine-tuned on 5000 sentences. As we saw in 4.2.4, the performance for the 20k model and 10k model was worse than 5k. Nonetheless, training these models longer could result in better performance as they have more data to learn from. This would also require finding a certain set of hyperparameters which work best for their training process. As an experiment on the side, we tried to train the model with 50000 data entries, for 50 epochs. The performance of the model degraded very soon considering the aggressive learning rates set for the MLM head. Due to shortage of time, we couldn't experiment more, but one simple conclusion was to change the hyperparameters, especially the learning rates, based on the size of the dataset. A hyperparameter search would need to be conducted for a dataset of this size, as for smaller (subsets of the) datasets like ours, aggressive learning rates did not necessarily punish the training procedure. With the same reasoning, one could also experiment with different LoRA adapter configurations for bigger, or even smaller dataset.

6 Conclusions

This thesis presented a novel approach to reduce the computational cost of multilinguality in LLMs by using morphologically informed embeddings. By combining a custom Hindi-English dictionary, morphological tokenization, an embedding bridge and fine-tuning using LoRA adapters, the proposed method offers a promising direction for multilingual NLP.

The experimental results demonstrate the ability of this approach to achieve competitive performance on certain tasks. The success of frequency based masking strategy, the insights gained from learning rate optimization, and tuning LoRA configurations demonstrate the importance of experimenting with different pre-training and fine-tuning techniques.

The evaluation across multiple IndicGLUE tasks revealed both strengths and areas for improvement. While the model performed well in tasks like content classification, it showed room for growth in tougher tasks like complex reasoning and discourse understanding. These results provide valuable insight and direction for future research, indicating where efforts might pay off to boost the model's capabilities.

The approach's ability to leverage pre-trained English language models while adapting to a different language like Hindi, offers a flexible framework that could be extended to multiple languages. With more experiments, research and improvements, this approach could expand to various languages with just a dataset and some adapters.

In conclusion, while there is still room for improvement, the approach proposed shows significant potential for bridging the gap between high-resource and low-resource languages in NLP. By addressing specific challenges of morphologically rich languages and offering a computationally efficient approach to multilinguality, this work opens up new avenues for research and hopefully pushes further research in development in multilingual NLP.

Abbreviations

NLP Natural Language Processing

LLMs Large Language Models

LoRA Low-Rank Adaptation

BPE Byte-Pair Encoding

ReLU Rectified Linear Unit

MSE Mean Squared Error

MLM Masked Language Modelling

NSP Next Sentence Prediction

List of Figures

List of Tables

Bibliography

Aghajanyan, A., L. Zettlemoyer, and S. Gupta (2020). *Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning*. arXiv: 2012.13255 [cs.LG].

Akhtar, M. S., A. Kumar, A. Ekbal, and P. Bhattacharyya (Dec. 2016). "A Hybrid Deep Learning Architecture for Sentiment Analysis." In: *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*. Ed. by Y. Matsumoto and R. Prasad. Osaka, Japan: The COLING 2016 Organizing Committee, pp. 482–493.

Akhtar, S. S., A. Gupta, A. Vajpayee, A. Srivastava, and M. Shrivastava (Apr. 2017). "Word Similarity Datasets for Indian Languages: Annotation and Baseline Systems." In: *Proceedings of the 11th Linguistic Annotation Workshop*. Ed. by N. Schneider and N. Xue. Valencia, Spain: Association for Computational Linguistics, pp. 91–94. DOI: 10.18653/v1/W17-0811.

Ba, J. L., J. R. Kiros, and G. E. Hinton (2016). *Layer Normalization*. arXiv: 1607.06450 [stat.ML].

Ben Zaken, E., Y. Goldberg, and S. Ravfogel (May 2022). "BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models." In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Ed. by S. Muresan, P. Nakov, and A. Villavicencio. Dublin, Ireland: Association for Computational Linguistics, pp. 1–9. DOI: 10.18653/v1/2022.acl-short.1.

Bhattacharyya, P. (2024a). *Hindi Wordnet*.

– (2024b). *Universal Word-Hindi Dictionary*.

Bojar, O., V. Diatka, P. Rychlý, P. Straňák, V. Suchomel, A. Tamchyna, and D. Zeman (May 2014). "HindEnCorp - Hindi-English and Hindi-only Corpus for Machine Translation." In: *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*. Ed. by N. Calzolari, K. Choukri, T. Declerck, H. Loftsson, B. Maegaard, J. Mariani, A. Moreno, J. Odijk, and S. Piperidis. Reykjavik, Iceland: European Language Resources Association (ELRA), pp. 3550–3555.

Botha, J. and P. Blunsom (2014). "Compositional Morphology for Word Representations and Language Modelling." In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by E. P. Xing and T. Jebara. Vol. 32. Proceedings of Machine Learning Research 2. Bejing, China: PMLR, pp. 1899–1907.

Brown, T. B., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei (2020). *Language Models are Few-Shot Learners*. arXiv: 2005.14165 [cs.CL].

Chaudhary, A., C. Zhou, L. Levin, G. Neubig, D. R. Mortensen, and J. Carbonell (Oct. 2018). "Adapting Word Embeddings to New Languages with Morphological and Phonological Subword Representations." In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Ed. by E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii. Brussels, Belgium: Association for Computational Linguistics, pp. 3285–3295. DOI: 10.18653/v1/D18-1366.

Chimalamarri, S., D. Sitaram, and A. Jain (June 2020). "Morphological Segmentation to Improve Crosslingual Word Embeddings for Low Resource Languages." In: *ACM Trans. Asian Low-Resour. Lang. Inf. Process.* 19.5. ISSN: 2375-4699. DOI: 10.1145/3390298.

Church, K. W. (2020). "Emerging trends: Subwords, seriously?" In: *Natural Language Engineering* 26.3, pp. 375–382. DOI: 10.1017/S1351324920000145.

Conneau, A., K. Khandelwal, N. Goyal, V. Chaudhary, G. Wenzek, F. Guzmán, E. Grave, M. Ott, L. Zettlemoyer, and V. Stoyanov (2019). "Unsupervised Cross-lingual Representation Learning at Scale." In: *CoRR* abs/1911.02116. arXiv: 1911.02116.

Creutz, M., K. Lagus, and S. Virpioja (2006). "Unsupervised Morphology Induction Using Morfessor." In: *Finite-State Methods and Natural Language Processing*. Ed. by A. Yli-Jyrä, L. Karttunen, and J. Karhumäki. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 300–301. ISBN: 978-3-540-35469-7.

Crystal, D. (2008). "Two thousand million?" In: *English Today* 24.1, pp. 3–6. DOI: 10.1017/S0266078408000023.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (June 2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by J. Burstein, C. Doran, and T. Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

Doddapaneni, S., R. Aralikatte, G. Ramesh, S. Goyal, M. M. Khapra, A. Kunchukuttan, and P. Kumar (July 2023). "Towards Leaving No Indic Language Behind: Building Monolingual Corpora, Benchmark and Models for Indic Languages." In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by A. Rogers, J. Boyd-Graber, and N. Okazaki. Toronto, Canada: Association for Computational Linguistics, pp. 12402–12426. DOI: 10.18653/v1/2023.acl-long.693.

Eskilson, A. (2015). *English-Hindi CSV Dictionary*. https://https://github.com/bdrillard/english-hindi-dictionary.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. MIT Press.

Goyal, V., P. Mishra, and D. M. Sharma (May 2020). "Linguistically Informed Hindi-English Neural Machine Translation." English. In: *Proceedings of the Twelfth Language Resources and Evaluation Conference*. Ed. by N. Calzolari, F. Béchet, P. Blache, K. Choukri, C. Cieri, T. Declerck, S. Goggi, H. Isahara, B. Maegaard, J. Mariani, H. Mazo, A. Moreno, J. Odijk, and S. Piperidis. Marseille, France: European Language Resources Association, pp. 3698–3703. ISBN: 979-10-95546-34-4.

He, K., X. Zhang, S. Ren, and J. Sun (2015). *Deep Residual Learning for Image Recognition*. arXiv: 1512.03385 [cs.CV].

Hochreiter, S. and J. Schmidhuber (1996). "LSTM can solve hard long time lag problems." In: *Proceedings of the 9th International Conference on Neural Information Processing Systems*. NIPS'96. Denver, Colorado: MIT Press, pp. 473–479.

Hofmann, V., J. Pierrehumbert, and H. Schütze (Aug. 2021). "Superbizarre Is Not Superb: Derivational Morphology Improves BERT's Interpretation of Complex Words." In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Ed. by C. Zong, F. Xia, W. Li, and R. Navigli. Online: Association for Computational Linguistics, pp. 3594–3608. DOI: 10.18653/v1/2021.acl-long.279.

Hofmann, V., H. Schuetze, and J. Pierrehumbert (May 2022). "An Embarrassingly Simple Method to Mitigate Undesirable Properties of Pretrained Language Model Tokenizers." In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Ed. by S. Muresan, P. Nakov, and A. Villavicencio. Dublin, Ireland: Association for Computational Linguistics, pp. 385–393. DOI: 10.18653/v1/2022.acl-short.43.

Hou, J., A. Katinskaia, A.-D. Vu, and R. Yangarber (Dec. 2023). "Effects of sub-word segmentation on performance of transformer language models." In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by H. Bouamor, J. Pino, and K. Bali. Singapore: Association for Computational Linguistics, pp. 7413–7425. DOI: 10.18653/v1/2023.emnlp-main.459.

Houlsby, N., A. Giurgiu, S. Jastrzebski, B. Morrone, Q. de Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly (2019). *Parameter-Efficient Transfer Learning for NLP*. eprint: arXiv:1902.00751.

Hu, E. J., Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, and W. Chen (2021). "LoRA: Low-Rank Adaptation of Large Language Models." In: *CoRR* abs/2106.09685. arXiv: 2106.09685.

Jurafsky, D. and J. H. Martin (2024). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. Online manuscript released August 20, 2024.

Kakwani, D., A. Kunchukuttan, S. Golla, G. N.C., A. Bhattacharyya, M. M. Khapra, and P. Kumar (Nov. 2020). "IndicNLPSuite: Monolingual Corpora, Evaluation Benchmarks and Pre-trained Multilingual Language Models for Indian Languages." In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Ed. by T. Cohn, Y. He, and Y. Liu. Online: Association for Computational Linguistics, pp. 4948–4961. DOI: 10.18653/v1/2020.findings-emnlp.445.

Karimi Mahabadi, R., J. Henderson, and S. Ruder (2021). "Compacter: Efficient Low-Rank Hypercomplex Adapter Layers." In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan. Vol. 34. Curran Associates, Inc., pp. 1022–1035.

Kim, S., S. Choi, and M. Jeong (2024). *Efficient and Effective Vocabulary Expansion Towards Multilingual Large Language Models*. arXiv: 2402.14714 [cs.CL].

Kingma, D. P. and J. Ba (2017). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs.LG].

Kunchukuttan, A. (2020). *The IndicNLP Library*. https://github.com/anoopkunchukuttan/indic_nlp_library/blob/master/docs/indicnlp.pdf.

Li, X. L. and P. Liang (Aug. 2021). "Prefix-Tuning: Optimizing Continuous Prompts for Generation." In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Ed. by C. Zong, F. Xia, W. Li, and R. Navigli. Online: Association for Computational Linguistics, pp. 4582–4597. DOI: 10.18653/v1/2021.acl-long.353.

Liu, Y., M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov (2019). "RoBERTa: A Robustly Optimized BERT Pretraining Approach." In: *CoRR* abs/1907.11692. arXiv: 1907.11692.

Loper, E. and S. Bird (2002). *NLTK: The Natural Language Toolkit*. arXiv: cs/0205028 [cs.CL].

Loshchilov, I. and F. Hutter (2017). "Fixing Weight Decay Regularization in Adam." In: *CoRR* abs/1711.05101. arXiv: 1711.05101.

Luong, T., R. Socher, and C. Manning (Aug. 2013). "Better Word Representations with Recursive Neural Networks for Morphology." In: *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*. Ed. by J. Hockenmaier and S. Riedel. Sofia, Bulgaria: Association for Computational Linguistics, pp. 104–113.

Mikolov, T., K. Chen, G. Corrado, and J. Dean (2013). *Efficient Estimation of Word Representations in Vector Space*. arXiv: 1301.3781 [cs.CL].

Miller, G. A. (1994). "WordNet: A Lexical Database for English." In: *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*.

Pfeiffer, J., I. Vulić, I. Gurevych, and S. Ruder (Nov. 2021). "UNKs Everywhere: Adapting Multilingual Language Models to New Scripts." In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Ed. by M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, pp. 10186–10203. DOI: 10.18653/v1/2021.emnlp-main.800.

Poth, C., H. Sterz, I. Paul, S. Purkayastha, L. Engländer, T. Imhof, I. Vulić, S. Ruder, I. Gurevych, and J. Pfeiffer (2023). *Adapters: A Unified Library for Parameter-Efficient and Modular Transfer Learning*. arXiv: 2311.11077 [cs.CL].

Raffel, C., N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu (2019). "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." In: *CoRR* abs/1910.10683. arXiv: 1910.10683.

Rai, A. and S. Borah (2021). "Study of Various Methods for Tokenization." In: *Applications of Internet of Things*. Ed. by J. K. Mandal, S. Mukhopadhyay, and A. Roy. Singapore: Springer Singapore, pp. 193–200. ISBN: 978-981-15-6198-6.

Richardson, L. (2007). "Beautiful soup documentation." In: *April*.

Roemmele, M., C. A. Bejan, and A. S. Gordon (Mar. 2011). "Choice of Plausible Alternatives: An Evaluation of Commonsense Causal Reasoning." In: *AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*. Stanford University.

Rumelhart, D. E., R. Durbin, R. Golden, and Y. Chauvin (2013). "Backpropagation: The basic theory." In: *Backpropagation*. Psychology Press, pp. 1–34.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). "Learning internal representations by error propagation." In.

Sennrich, R., B. Haddow, and A. Birch (Aug. 2016). "Neural Machine Translation of Rare Words with Subword Units." In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by K. Erk and N. A. Smith. Berlin, Germany: Association for Computational Linguistics, pp. 1715–1725. DOI: 10.18653/v1/P16-1162.

Smit, P., S. Virpioja, S.-A. Grönroos, and M. Kurimo (Apr. 2014). "Morfessor 2.0: Toolkit for statistical morphological segmentation." In: *Proceedings of the Demonstrations at the 14th Conference of the European Chapter of the Association for Computational Linguistics*. Ed. by S. Wintner, M. Tadić, and B. Babych. Gothenburg, Sweden: Association for Computational Linguistics, pp. 21–24. DOI: 10.3115/v1/E14-2006.

Soler, A. G., M. Labeau, and C. Clavel (2024). "The Impact of Word Splitting on the Semantic Content of Contextualized Word Representations." In: *Transactions of the Association for Computational Linguistics* 12, pp. 299–320. DOI: 10.1162/tacl_a_00647.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." In: *Journal of Machine Learning Research* 15.56, pp. 1929–1958.

Toraman, C., E. H. Yilmaz, F. Şahinuç, and O. Ozcelik (Mar. 2023). "Impact of Tokenization on Language Models: An Analysis for Turkish." In: *ACM Trans. Asian Low-Resour. Lang. Inf. Process.* 22.4. ISSN: 2375-4699. DOI: 10.1145/3578707.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, Ł. Kaiser, and I. Polosukhin (2017). *Attention Is All You Need*.

Wang, A., A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman (2018). "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding." In: *CoRR* abs/1804.07461. arXiv: 1804.07461.

Wikitionary (2024). *List of dictionaries by number of words — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-October-2024].

Wu, Y., M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean (2016). "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation." In: *CoRR* abs/1609.08144. arXiv: 1609.08144.

You, K., M. Long, M. I. Jordan, and J. Wang (2019). "Learning Stages: Phenomenon, Root Cause, Mechanism Hypothesis, and Implications." In: *CoRR* abs/1908.01878. arXiv: 1908.01878.

Yu, Y., C.-H. H. Yang, J. Kolehmainen, P. G. Shivakumar, Y. Gu, S. R. R. Ren, Q. Luo, A. Gourav, I.-F. Chen, Y.-C. Liu, T. Dinh, A. G. D. Filimonov, S. Ghosh, A. Stolcke, A. Rastow, and I. Bulyko (Dec. 2023). "Low-Rank Adaptation of Large Language Model Rescoring for Parameter-Efficient Speech Recognition." In: *2023 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE. DOI: 10.1109/asru57964.2023.10389632.