Eötvös Loránd University
Faculty of Informatics

Multimodal Classification Using Deep Neural Networks

Supervisor
Tsegaye Misikir Tashu (Ph.D)
Data Science and Engineering Department

Author
Sara Fattouh
Computer Science

June 2021, Budapest

To my Father .. in loving memory
to whom I promised to dedicate
this dissertation before he left this world

To my Mother .. for unconditional love
for inspiring me to take nothing for granted

To my supervisor, Prof. Tsegaye Misikir
for valuable time, patience and guidance

To the laughs of friends
through the hardest times

To innocent children victims of aggression
anywhere on earth

To anyone who believed in me .. and helped me along the way

Statement
of thesis submission and originality

I hereby confirm the submission of the Master Thesis Work on the Computer Science MSc course with author and title:

Name of Student: **Sara Fattouh**
Code of Student: **G9UHTN**
Title of Thesis: **Multimodal Classification Using Deep Neural Networks**
Supervisor: **Tsegaye Misikir Tashu**

at Eötvös Loránd University, Faculty of Informatics

In consciousness of my full legal and disciplinary responsibility I hereby claim that the submitted thesis work is my own original intellectual product, the use of referenced literature is done according to the general rules of copyright.
I understand that in the case of thesis works the following acts are considered plagiarism:

- literal quotation without quotation marks and reference;
- citation of content without reference;
- presenting others' published thoughts as own thoughts.

Budapest, May 15, 2021

Sara Fattouh

EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS

Thesis Registration Form

Student's Data:
Student's Name: Fattouh Sara
Student's Neptun code: G9UHTN

Course Data:
Student's Major: Computer Science MSc

I have an internal supervisor

Internal Supervisor's Name: *Tashu, Tsegaye Misikir*

Supervisor's Home Institution:	*ELTE Faculty of Informatics*
Address of Supervisor's Home Institution:	*1117 Budapest, Pázmány Péter sétány 1/C.*
Supervisor's Position and Degree:	*Position: Research assistant, Degree: PhD Candidate*

Thesis Title: Multimodal Classification Using Deep Neural Networks

Topic of the Thesis:
(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

Multimodal data has become increasingly popular in people's daily life with the advent of the internet. Modality refers to the way something happens or is expressed. We say that a research problem is multimodal when it includes multiple such modalities or types of information. Human beings have a multimodal perception of the world. We see objects, hear sounds, touch textures, taste flavors and smell odors. As artificial intelligence applications aim to mimic the human brain and behavior, there is a need for the models to combine and process such multimodal signals together, where different modalities have different properties and statistical characteristics. Multimodal learning involves relating information from multiple sources, for example, images are usually associated with captions, text documents include images for illustration and to more clearly and quickly convey the essence of the article.
Deep neural networks have achieved satisfactory results when applied for single modalities. We will investigate and propose a novel application of deep neural networks to learn features from different modalities using a dedicated neural network for each modality and then combining them to output the final result. Though combining different modalities to achieve state-of-the-art-results sounds appealing, but in practice, it is challenging to combine different representations and levels of noise between modalities. Moreover, modalities have a different quantitative contribution to the final output. Therefore, giving equal importance to all modalities is highly unlikely to be true in real-life problems. All that makes this recent advance in deep learning an important and attractive area of research.

Budapest, 2020.09.28.

Contents

List of Figures

List of Tables

List of Abbreviations

DL	Deep Learning
AI	Artificial Intelligence
ML	Machine Learning
ANN	Artificial Neural Network
NN	Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Networks
LSTM	Long Short-term Memory
ConvS2S	Convolutional Sequence-to-Sequence
BERT	Bidirectional Encoder Representations from Transformers
RoBERTA	Robustly Optimized BERT Pre-training Approach
ELMo	Embeddings from Language Models
XLM	Cross-lingual Language Model
GPT	Generative Pre-trained Transformer
MLM	Masked Language Model
NLP	Next Sentence Prediction
MLM	The Masked Language Modeling
POS	Part of Speech
NLI	Natural Language Inference

ResNet	Residual Neural Network
GELU	Gaussian Error Linear Unit
ReLU	Rectified Linear Unit
LR	Linear Regression
CE	Cross-Entropy
GD	Gradient Descent
DCNN	Deep Convolutional Neural Network
DA	Data Augmentation
TP	True Positives
FP	False Positives
TN	True Negatives
FN	False Negatives

Chapter 1

Introduction

1.1 Motivation

With the invention of the internet, multimodal data has become increasingly common in people's everyday lives. The way something occurs or is expressed is referred to as a modality. We say that a research problem is multimodal when it includes multiple modalities or types of information. Human beings have a multimodal perception of the world. We can see objects, hear noises, feel texture, taste flavors, and smell odors. Artificial intelligence algorithms attempt to mimic the human brain's way of processing input data. Therefore, models should integrate and process multi-modal inputs, each of which has its own set of properties and statistical characteristics. Multi-modal learning involves integrating information from various sources; for example, images are often accompanied by captions, and text articles often contain images for illustration and to express the meaning of the article more accurately and clearly.

Deep neural networks have achieved satisfactory results when applied to single modalities and promising results in multimodal classification. In this study, We will investigate and propose a transfer learning-based architecture that learns features from different modalities using a dedicated neural network for each modality and then fuse them to make a proper classification.

1.1.1 Problem Definition

Commercial products are grouped in hundreds, if not thousands of categories. This makes manual and rule-based methods of categorization unscalable and lacks generalizability when products need to be classified into a large number of classes. Developing a multi-modal deep learning architecture solution for product classification will not only help the company in efficiently scaling it for multiple classes but also save a large amount of time and rule-based generalization errors. It is often observed that not only the text of the product's title and description but also the images associated with it are helpful. Advances in this area of research have been limited due to the lack of real product data from actual commercial catalogs.

1.2 Objective

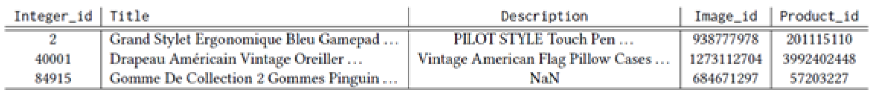
The goal of this research is to build a model that classifies previously unseen products into their corresponding product type codes. We propose a deep Multi-Modal Multi-Level Fusion Learning Framework used to categorize large-scale multi-modal (text and image) product data into product type codes. Basically, we solve a fairly large-scale multi-modal product classification. We present our solution for the "Multi-modal Product Classification" task as a part of the 2020 SIGIR challenge on e-commerce (ECOM20).

For example, a product with a French title *" Klarstein Présentoir 2 Montres Optique Fibre "* is associated with an image and sometimes with an additional description. This product is categorized with a product type code of 1500. There are other products with different titles, images, and possible descriptions, which are under the same product type code. Given this information on the products, we built the proposed model. Other examples can be seen in Figure 1.1.

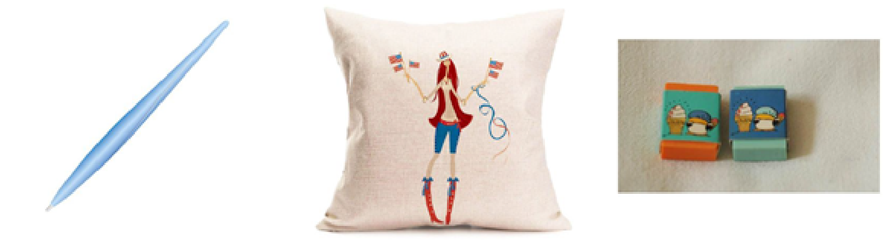
1.3 Thesis Outline

The structure of the thesis is described as follows:

- Chapter 2: provides an overview of BERT, a transformer-based machine learning technique for natural language processing pre-training. Moreover, BERT attention's analysis is presented.



Integer_id	Title	Description	Image_id	Product_id
2	Grand Stylet Ergonomique Bleu Gamepad …	PILOT STYLE Touch Pen …	938777978	201115110
40001	Drapeau Américain Vintage Oreiller …	Vintage American Flag Pillow Cases …	1273112704	3992402448
84915	Gomme De Collection 2 Gommes Pinguin …	NaN	684671297	57203227



(a) image_938777978_product_201115110.jpg; Category: Entertainment

(b) image_1273112704_product_3992402448.jpg; Category: Household

(c) image_684671297_product_57203227.jpg; Category: Books

Figure 1.1: Images of the three example products

- Chapter 3: describes BERT's variations that were used in our study to process the text modality.
- Chapter 4: includes an overview of ResNet and Squeeze-and-Excitation Networks. The latter architecture was used to process visual data (image modality)
- Chapter 5: gives an overview of model fusion architectures as well as transfer learning importance and methods.
- Chapter 6: Presents our proposed hierarchical fusion architecture, experiments, and results.
- Chapter 7: is a conclusion about the proposed model with the future work of this research.

Chapter 2

Bidirectional Encoder Representations from Transformers

2.1 Introduction

Recurrent neural networks (RNN), long short-term memory(LSTM), and gated recurrent neural networks, in particular, have been well-developed for sequence modeling and transduction problems such as language modeling and machine translation. Since then, several attempts to push the boundaries of recurrent language models and encoder-decoder architectures have been made. Recurrent models typically do computations along with the symbol positions of the input and output sequences. Aligning the positions to steps in computation time, they generate a sequence of hidden states h_t , as a function of the previous hidden state h_{t-1} and the input for position t. When processing input examples, the procedure's sequential structure forbids parallelization. Longer input sequences necessitate this because memory limitations prevent batching through examples. Using factorization tricks, recent work has achieved significant advances in computational efficiency [30]

Attention mechanisms have become an integral part of compelling sequence modeling in various tasks, enabling modeling of dependencies regardless of their distance in the input or output sequences [56]. More parallelization is possible with the Transformer architecture, which was proposed in a paper published by Google Research in 2017. To construct global dependencies between input and output, this model relies on the attention mechanism.

2.2 Transformer Model Architecture

In sequential computation, the number of operations that are needed to relate two input and output positions grows linearly for ConvS2S [25] and logarithmically for ByteNet [26] This makes learning dependencies between distant positions more difficult. The number of operations in the Transformer architecture is constant. However, this is achieved at the cost of the effective resolution due to averaging attention-weighted positions. Self-attention is a method for calculating a representation of a sequence by relating the positions of one input sequence. The transformer is the first model to represent the sequence of input and output using just self-attention rather than sequence-aligned RNNs and convolution. Transformer model has an encoder-decoder structure, whereas the encoder maps an input sequence of symbol representations ($\mathrm{x}_1; \ldots; \mathrm{x}_n$) to a sequence of continuous representations z = ($\mathrm{z}_1; \ldots;$ z_n). Given z, the decoder then generates an output sequence ($\mathrm{y}_1; \ldots; \mathrm{y}_n$) of symbols one element at a time. At each step, the model consumes the previously generated symbol as an additional input [56].

2.2.1 Encoder and Decoder Stacks

Encoder: The encoder is made up of a stack of six equal layers, each of which is divided into two sub-layers. The first is a multi-head self-attention mechanism. A position-wise feed-forward network is the second. A residual connection is also present around each of the two sub-layers, accompanied by a normalization layer. In addition to the embedding layers, all sub-layers in the model produce output with dimension $\mathrm{d}_m odel = 512$ [56].

Decoder: A stack of six identical layers also makes up the decoder. In addition to the two sub-layers in an encoder layer, it has a third sub-layer that performs multi-head attention over the encoder stack's output. Like the encoder, there are residual connections across each sub-layer and a normalization layer then comes the next.

2.2.2 Attention

The authors of the Encoder-Decoder network proposed attention as an extension. The Encoder-Decoder model's restriction of encoding the input sequence to one fixed-length vector from which to decode each output time step is proposed to be overcome. This dilemma is thought to be more problematic when decoding long sequences. Generally speaking, attention is a way for a model to assign weight to input features, depending on their relevance to some task. For example, a language model attempting to predict the masked word in a sentence may choose to pay more attention to certain background words relevant to the subject in order to predict the next word. BERT's attention mechanism is straightforward. It works by mapping an input to a set of key-value pairs, with the query, key, value, and output all being vectors. The result is a weighted sum of values, with the weight assigned to each value determined by the query's compatibility function with the corresponding key.

Scaled dot-product

In the BERT model, attention is referred to as "Scaled Dot-Product Attention." The input consists of queries and keys of size d$_k$, as well as values of size d$_v$. The output is calculated by taking the dot-product of the query with all keys and dividing each by the squared root of d$_k$. Finally, the Softmax function is used to calculate the weights' values. As mentioned in the introduction, unlike traditional sequence models such as RNNs, BERT allows the weights of the input vectors to be calculated in parallel. As a result, the focus function on multiple input vectors is computed concurrently in operation. A matrix Q is used to store the input vectors. In the same way, the keys and values are packed into the matrices K and V. The following formula summarizes the computation of the output matrix in BERT:

$$Attention(Q;K;V) = softmax(\frac{(Q.K^T)}{\sqrt{dk}})V \tag{2.1}$$

The two most commonly used attention functions are additive attention and dot-product. The dot-product is similar to the algorithm which is used in the BERT model except for the scaling factor $\frac{1}{\sqrt{dk}}$.

Although the potential time complexity of the two functions is the same, dot-product attention is much faster and takes up less space in practice since it can be applied using highly optimized matrix multiplication code. Scaled Dot-Product Attention vs Multi-Head Attention is illustrated in Figure 2.1 [56]

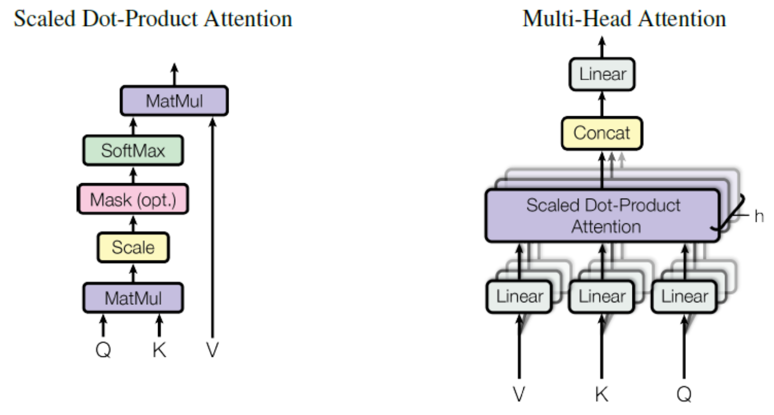


Figure 2.1: Scaled Dot-Product Attention vs Multi-Head Attention

Multi-Head Attention

Instead of applying a single attention function with d$_model$-dimensional keys, values, and queries, BERT projection queries, keys, and values to dk, dk, and dv dimensions h times using different, learned linear projections. We then perform the attention function in parallel on each of these projected versions of queries, keys, and values, yielding dv-dimensional output values. These are concatenated and projected once more, yielding the final values, as depicted in Figure 2.1. Intuitively, multiple attention heads allow for attending to parts of the sequence differently (e.g. longer-term dependencies versus shorter-term dependencies). The attention module splits its query, key, and value parameters N times and sends each break to a different Head. The results of both of these related Attention estimates are then combined to create a final Attention score. This allows the Transformer to encode several relationships for each word with greater ease. Below is the step-by-step procedure for calculating multi-headed self-attention:

- Take each word of the input sentence and generate the embedding from it.
- In this mechanism, h different attention heads (for example, h $= 8$) are created, each head has different weight matrices (W$^Q, W^K, W^V$).
- To produce the key, value, and query matrices for each attention head, multiply the input matrix with each of the weight matrices (W$^Q, W^K, W^V$).

- Apply the attention mechanism to these query, key, and value matrices, this gives us an output matrix from each attention head.
- Finally, concatenate the output matrix obtained from each attention head and dot product with the weight W$^{Q} togeneratetheoutputofthemulti-headedattentionlayer.$

Mathematically, multi-head attention can be represented by:

$$MultiHead(Q; K; V) = Concat(head1; \ldots ; headh).W^{Q} \tag{2.2}$$

where head$_i$ is Attention and the projections are parameter matrices W^{Q}i ∈R$^{dmodel \times dk}$ W^{K}i ∈R$^{dmodel \times dk}$ W^{V}i ∈R$^{dmodel \times dv}$ and W^{Q} ∈R$^{hdv \times dmodel}$.

Complexity

The advantage of using self-consideration layers in the NLP tasks is that it is less computationally expensive to execute than other operations. Table 2.1 depicts the difficulty of various operations.

Layer Type	**Complexity per Layer**
Self-Attention	O (n^2.d)
Recurrent	O (n. d^2)
Convolutional	O (k. n . d^2)
Self-Attention	O (r. n. d)

Table 2.1: Per-layer Complexity for Different Layer Types.

where n is the sequence length, d is the representation dimension, k is the kernel size of convolutions, and r the size of the neighborhood in restricted self-attention

2.2.3 Position-wise Feed-Forward Networks

Each of the encoder and decoder layers in BERT has a fully connected feed-forward network that is applied to each position. This is made up of two linear transformations separated by a ReLU activation [56]

$$FFN(x) = max(0; xW_1 + b_2) * W_2 + b_2 \tag{2.3}$$

Although the linear transformations are the same in all positions, the parameters used for each layer vary.

2.2.4 Positional Encoding

Since the BERT model includes no recurrence and no convolution, to allow the model to exploit the order of the sequence, information about the location must be used, whether these be the relative or absolute position of the token in the sequence. The "positional encodings" are injected into the input embeddings at the bottom of both the encoder and decoder stacks for this purpose. Since the positional encodings and embeddings have the same dimension dmodel , the two can be combined. There are several positional encoding alternatives that can be learned and fixed [25]. The sine and cosine functions are used in the basic BERT model published in 2017 [56].

$$PE(pos; 2_i) = sin(pos/1000^{Q_{2i}/d_{model}}) \tag{2.4}$$

$$PE(pos; 2_{i+1}) = cos(pos/1000^{Q_i/d_{model}}) \tag{2.5}$$

The location is pos, and the dimension is i. As a result, a sinusoid responds to each dimension of the position encoding. It's worth noting that this function was selected because researchers believed it would make it easier for the model to learn to attend to relative positions because PE_{pos+k} can be expressed as a linear function of PE_{pos} for any fixed offset k.

Why Self-attention?

In many sequence transduction tasks, learning long-range dependencies is a major obstacle. The length of the paths forward and backward signals must travel in the network is one important factor impacting the ability to learn those dependencies. The shorter these paths between any combination of positions in the input and output sequences, the easier it is to learn long-range dependencies [51]. Following that, we compare and evaluate the maximal path length between any two input and output positions in networks made up of various layer types. A self-attention layer connects all positions with a fixed number of sequentially executed operations, while a recurrent layer allows O(n) sequential operations to bind all positions. Self-attention should be reduced to only considering a neighborhood of size r in the input sequence centered around the respective output position to increase computational efficiency for tasks involving processing very long sequences. This would increase the maximum path length to O(n=r).

It is important to remember the fact that the attention mechanism allows output to concentrate attention on input while producing output while the self-attention model allows inputs to interact with each other (for example, measure the attention of all other inputs with respect to one input) (for example, calculate attention of all other inputs with respect to one input).

2.3 Analysis of BERT's Attention

When fine-tuning large pre-trained language models on supervised tasks, they attain very high accuracy [5], [39], [45], but the reason for this is unknown. The clear findings show that pre-training teaches the models about language structure, but they don't know which basic linguistic features they learn. This topic has recently been explored by looking at the outputs of language models on carefully selected input sentences [29] or looking at the internal vector representations of the model using methods like probing classifiers [29]. Other researchers looked at the attention maps of a pre-trained model that was a popular neural network component. Since an attention weight has a simple meaning: how much a certain word would be weighted when calculating the next representation for the current word, it is easily interpretable. BERT has multiple attention heads, all of whom performed admirably in a variety of roles. They have typical patterns of behavior, such as paying attention to fixed positional offsets or paying enough attention to the whole sentence. This study finds that specific heads correspond surprisingly well to specific relations, despite the fact that no single head performs well on all relations. Heads that find direct objects of verbs, determiners of nouns, objects of prepositions, and objects of possessive pronouns with >75 % were discovered in the analysis. The behavior of the attention heads resulted from self-supervised training on unlabeled data, which was interesting.

Transformers, as discussed earlier in this chapter, is made up of several layers, each with several attention heads. The attention head receives a list of vectors that refer to the input tokens as input. Via different linear transformations, each vector is converted into a question, key, and value vector. Between all pairs of terms, the head calculates focus weights. Each weight is a soft-max normalized dot product between the query and key vectors. The output o is a weighted sum of the value vectors.

$$a_{i,j} = \frac{\exp(qi^T k_)}{\sum_{l=1}^{n} \exp(q_i^T k)} \tag{2.6}$$

$$o_i = \sum_{k=0}^{n} a_{i,j} v_i \tag{2.7}$$

When creating the next representation for the current token, attention weights can be thought of as governing how "important" any other token is. It's important to note that BERT is pre-trained to perform the tasks on 3.3 billion tokens of unlabeled text. The preprocessing of the input text is an important feature of BERT. A special token [CLS] is inserted at the start of the document, and another token [SEP] is added at the end; these special tokens play an important role in BERT's attention [27].

Figure 2.3 [27] shows Examples of heads exhibiting the patterns The darkness of a line indicates the strength of the attention weight (some attention weights are so low they are invisible)

2.4 Surface-Level Patterns in Attention

A research was conducted to examine surface-level patterns in the behavior of BERT's attention heads. Figure 4 shows several examples of heads with these shapes, where each point corresponds to the average attention a particular BERT attention head puts toward a token type. Above: heads often attend to "special" tokens. Early heads attend to [CLS], middle heads attend to [SEP], and deep heads attend to periods and commas. Often more than half of a head's total attention is to these

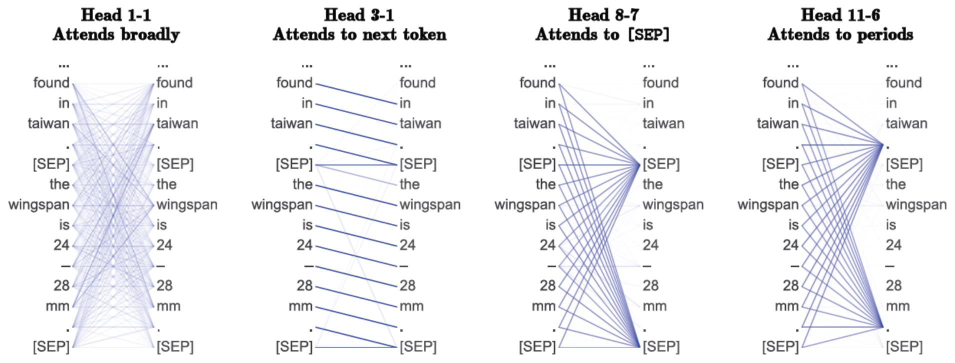


Figure 2.2: Examples of Heads and Attention Weights

tokens. Below in Figure 2.4: heads attend to [SEP] tokens, even more, when the current token is [SEP] itself [27]

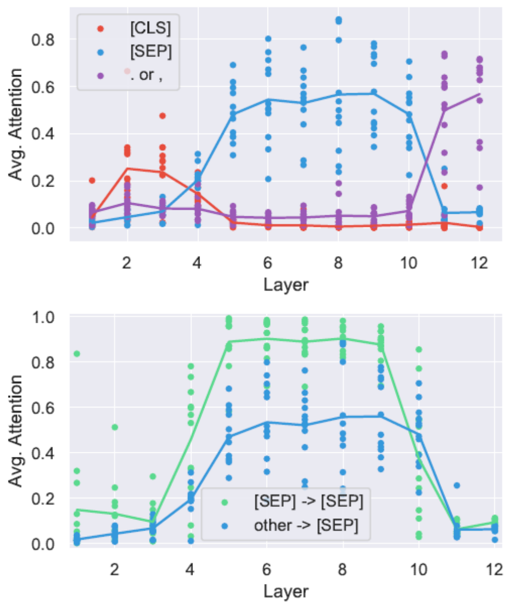


Figure 2.3: Attention of Heads to Different Tokens

2.4.1 Relative Position

Researchers looked at how much BERT attention heads pay attention to the current token, previous token, and next token [25], and discovered that the current token received the most of the heads' attention. Specifically, four attention heads (layers

2, 4, 7, and 8) devote more than 50% of their attention to the previous token, while five attention heads (layers 1, 2, 2, 3, and 6) devote more than 50% of their attention to the next token.

2.4.2 Attending to Separator Tokens

Surprisingly, the same analysis [27] discovered that a large proportion of BERT's attention is focused on a few tokens; for example, in layers 6-10, over half of BERT's attention is focused on [SEP]. [CLS] and [SEP] tokens are always guaranteed to be present and are never masked out, while periods and commas are the most common tokens in the data (aside from "the"), which may explain why the model handles them differently. The uncased BERT model shows a similar trend, implying that the attention to special tokens is due to a systematic cause rather than being an artifact of stochastic training. Another possibility is that [SEP] is used to aggregate segment-level data that can then be read by other heads. However, further investigation leads one to believe that this is not the case. If this theory is right, attention heads processing [SEP] should pay attention to the whole segment in order to build up these representations. Instead, they devote nearly all of their attention (more than 90%) to themselves and the other [SEP] token. In addition, qualitative analysis (see Figure 6) reveals that heads with specific functions pay attention to [SEP] even when the function is not called for. In the head, for example, 8-10 direct objects pay attention to their verbs. non-nouns mostly attend to [SEP].

2.4.3 Focused vs Broad Attention

Researchers have looked at whether attention heads pay attention to a few words or a large number of words. They did this by calculating the average entropy of the attention distribution of each head (see Figure 2.4). They discovered that certain attention heads, especially those in the lower layers, have a wide range of attention. These attention heads with high entropy spend no more than 10% of their attention mass on any single word. These heads generate a bag-of-vectors representation of the sentence as their output. They also used only the [CLS] token to measure entropies for all attention heads. Although the average [CLS] entropies for most layers are very similar to those in Figure 5, the last layer has a high [CLS] entropy of 3.89 nats, indicating very large attention. This finding makes sense because the [CLS] token's representation is used as input for the "next sentence prediction" task during pre-training, so it pays attention to aggregate a representation for the whole input in the last layer. In the first layer, there are particularly high-entropy heads that produce bag-of-vector-like representations. This is illustrated in Figure 2.4.3 [27].

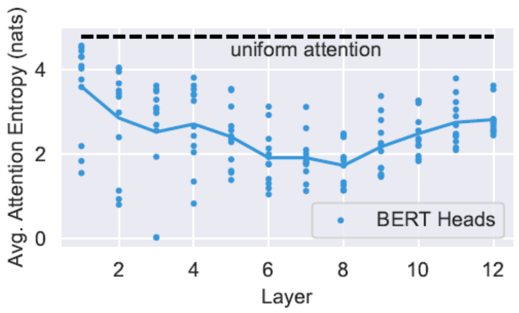


Figure 2.4: Entropies of Attention Distributions

Figure 2.5 shows some examples of attention behavior. While the similarity between learned attention weights and human-defined syntactic relations is striking, we note these are relations for which attention heads do particularly well. There are many relations for which BERT only slightly improves over the simple baseline, so we would not say individual attention heads capture dependency structure as a whole. In the example attention maps Figure2.4.3 [27], the darkness of a line indicates the strength of the attention weight. All attention to/from red words is colored red; these colors are there to highlight certain parts of the attention heads' behaviors.

2.5 Probing Individual Attention Heads

Specific attention heads were investigated as a next step in the research review of BERT's attention to see what aspects of language they had learned. This thesis used a named dataset to evaluate attention heads for tasks like dependency parsing.

Byte-pair Tokenization

This algorithm was created to aid data compression by identifying common byte pair combinations. It can also be used in natural language processing (NLP) to find the most effective way to represent text. It works by substituting a byte that does not appear in the data of common pairs of consecutive bytes. It is usually applied on rare words such as athazagoraphobia to be split up into more frequent subwords such as ['ath', 'az', 'agor', 'aphobia']. This algorithm achieves the ideal balance between character-level and word-level hybrid representations, allowing it to handle massive corpora. This action also allows any uncommon words in the language to be encoded with acceptable sub-word tokens without adding any "unknown" tokens. This is particularly true in foreign languages like German, where the presence of a large number of compound words can make understanding a large vocabulary difficult. Any word will now overcome their fear of being lost thanks to this tokenization algorithm.

Used Method

The researchers sought to analyze attention heads on a word-by-word basis, but BERT uses a byte-pair tokenization algorithm [18]. As a result, several words are divided into multiple tokens. As a result, they switched from token-token to word-to-word attention charts. They summed the attention weights over the tokens for a split-up word's attention. For attention from a split-up word, we take the mean of the attention weights over its tokens.

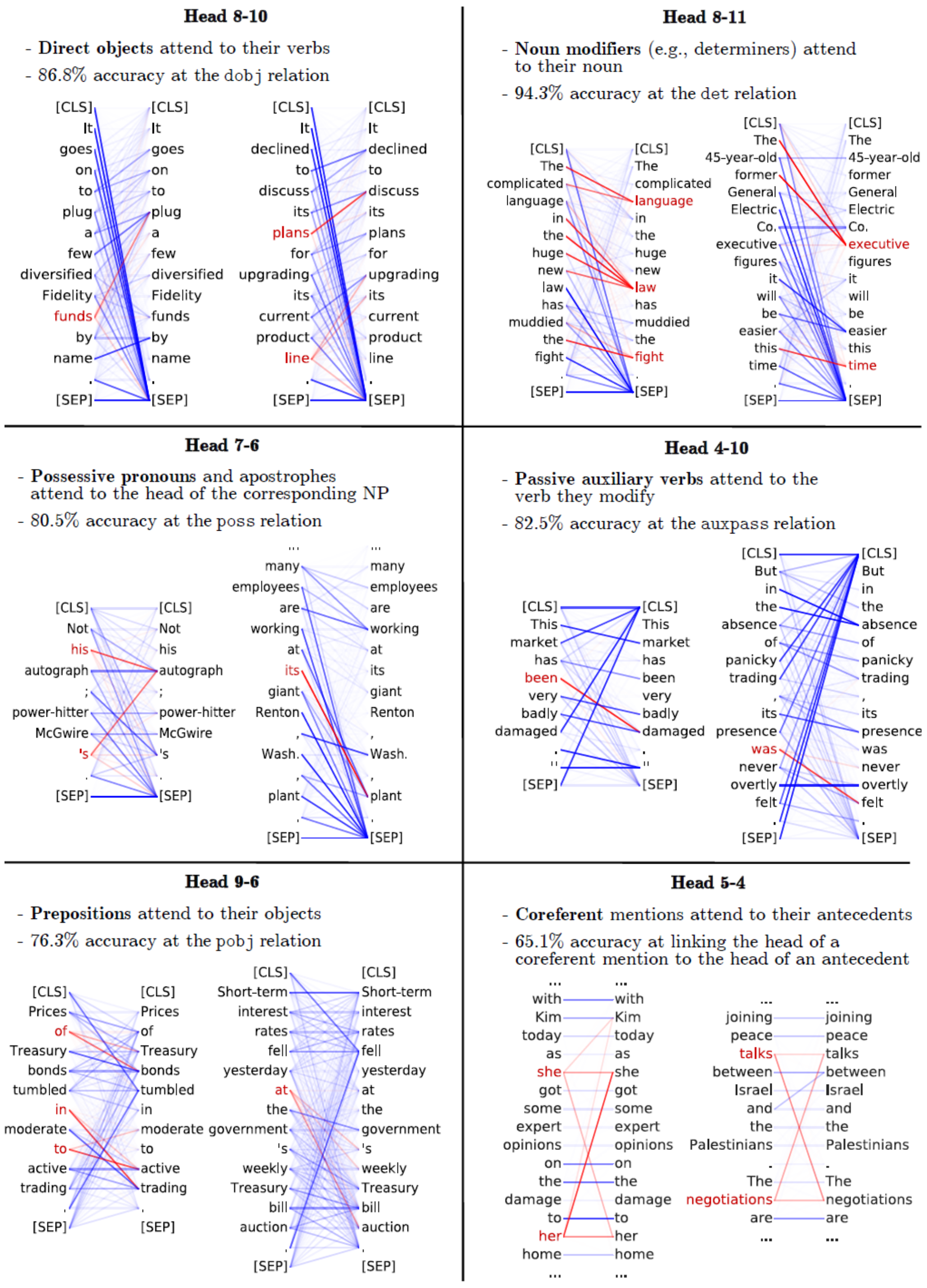

Head 8-10
- Direct objects attend to their verbs
- 86.8% accuracy at the dobj relation
Head 8-11
- Noun modifiers (e.g., determiners) attend to their noun
- 94.3% accuracy at the det relation
Head 7-6
- Possessive pronouns and apostrophes attend to the head of the corresponding NP
- 80.5% accuracy at the poss relation
Head 4-10
- Passive auxiliary verbs attend to the verb they modify
- 82.5% accuracy at the auxpass relation
Head 9-6
- Prepositions attend to their objects
- 76.3% accuracy at the pobj relation
Head 5-4
- Coreferent mentions attend to their antecedents
- 65.1% accuracy at linking the head of a coreferent mention to the head of an antecedent

Figure 2.5: BERT Attention heads that Correspond to Linguistic Phenomena

Chapter 3

BERT Variations

3.1 RoBERTA

3.1.1 Introduction

Self-training methods like Embeddings from Language Models (ELMo) [36], Generative Pre-trained Transformer (GPT) [4], BERT [10], Cross-lingual Language Model (XLM) [33] and XLNet [59] have resulted in substantial performance improvements, but determining the aspects of the methods contribute the most can be difficult. RoBERTa is a replication analysis of BERT in which the results of hyperparameter tuning and training set scale are carefully evaluated [61]. To put it another way, it's a better recipe for training BERT models that can equal or even outperform all post-BERT approaches. The changes are as follows:

- Training the model longer, with bigger batches, over more data.
- Removing the next sentence prediction objective.
- Training on longer sequences.
- Dynamically changing the masking pattern applied to the training data.

To better account for training set size impact, researchers obtained a large new dataset (CC-NEWS) of comparable size to other privately used datasets. When training data is controlled for, the enhanced training technique outperforms reported BERT findings on both the GLUE and SQuAD benchmarks. On the public GLUE leaderboard, the RoBERTa model achieves an 88.5 score after being trained for longer over more results. It's worth noting that the CNN NEWS dataset was used. It has been shown that and the amount of data used for pretraining increases efficiency on downstream activities.

3.1.2 Training Objectives

During pretraining, BERT uses two objectives: masked language modeling and next sentence prediction.

- **Masked Language Model (MLM)** The special token [MASK] is substituted for a random sample of the tokens in the input set. The MLM goal is to estimate the masked tokens with a cross-entropy loss. BERT chooses 15% of the input tokens for future substitution in a consistent manner. 80% of the tokens are replaced with [MASK], ten percent are left unchanged, and 10% are replaced by a vocabulary token chosen at random. Random masking and substitution are done once in the beginning and saved for the remainder of training in the initial implementation.
- **Next Sentence Prediction (NSP)** NSP is a binary classification loss that predicts whether two segments in the original text will match each other. Consecutive sentences from the text corpus are used to construct positive examples. Negative examples are made by combining sections from various documents. Of the same probability, both positive and negative examples are sampled. The NSP aimed to boost efficiency on downstream tasks.

3.1.3 Optimization

BERT is optimized with Adam [28] using the following parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e - 6$ and L2 weight decay of 0.01. The learning rate is warmed up over the first 10,000 steps to a peak value of 1e-4, and then linearly decayed. BERT trains with a dropout of 0.1 on all layers and attention weights, and a GELU activation function [20]. Mini batches of B = 256 sequences of maximum length T = 512 tokens are used to pre-train models for S = 1,000,000 updates.

Data

BERT is trained on 16GB of uncompressed text from a mixture of BOOKCORPUS [63] and English WIKIPEDIA. Five English-language corpora of varying sizes and domains were used to train RoBERTa, totaling over 160GB of uncompressed content. In a study [6], it was shown that increasing data size would increase end-task performance. Several efforts have used datasets that are larger and more diverse than the original BERT to prepare. Unfortunately, not all of the extra databases can be made available. The following are the corpora that were used:

1. BOOKCORPUS plus English WIKIPEDIA: This is the original data used to train BERT(16GB) [63].
2. CC-NEWS: which we collected from the English portion of the CommonCrawl News Dataset (76GB after filtering) [38].
3. OPENWEBTEXT: an open-source The text which is web content extracted from URLs shared on Reddit with at least three upvotes. (38GB) [17]
4. Stories: a dataset containing a subset of CommonCrawl data filtered to match the story-like style of Winograd schemas. (31GB) [55]

Evaluation

The pre-trained models were tested on downstream tasks using the following three benchmarks after the previous work and preparation:

1. GLUE: The GLUE benchmark [16] is a list of nine datasets for analyzing natural language understanding systems. The procedure for finetuning is based on the original BERT paper.[57]
2. SQuAD: The Stanford Question Answering Dataset (SQuAD) provides a paragraph of context and a question. The task is to answer the question by extracting the relevant span from the context. Researchers evaluated on two versions of SQuAD: V1.1 and V2.0[43]
3. RACE: is a large-scale reading comprehension dataset with more than 28,000 passages and over 100,000 questions [32]. The data was gathered from Chinese

English tests for middle and high school students. Each passage in RACE is linked to several questions. The challenge is to choose one correct answer from four choices for each question.

3.1.4 Training Procedure Analysis

In this part, we'll look at and measure which choices are crucial for pretraining BERT models successfully. It's important to note that the model architecture stayed stable. BERT models were first trained with the same parameters as BERTBASE (L = 12, H = 768, A = 12, 110M params).

Static vs. Dynamic Masking

BERT works by masking and guessing tokens at random. Masking was done only once during data preprocessing in the original BERT implementation, resulting in a single static mask. To prevent using the same mask for each training instance in each epoch, training data was duplicated ten times, resulting in each series being masked in ten different ways over the 40 training epochs. During rehearsal, each training series was shown four times with the same mask. This technique was compared in RoBERTa to dynamic masking, which generates a masking pattern every time a sequence is fed to the model. When pretraining for further phases or larger datasets, this becomes critical.
Table 3.1.4 shows a comparison between static and dynamic masking for $BERT_{BASE}$. F1 reported for SQuAD and accuracy for MNLI-m and SST-2. Reported results are medians over 5 random initializations (seeds)

Masking	**SQuAD 2.0**	**MNLI-m**	**SST-2**
Reference – BERT$_{BASE}$	76.3	84.3	92.8
RoBERTa - static	78.3	84.3	92.5
RoBERTa - dynamic	78.7	84.0	92.9

Table 3.1: Comparison Between Static and Dynamic Masking for $BERT_{BASE}$.

Static masking reimplementation performs similarly to the original BERT model, and dynamic masking performs similarly to or marginally better than static masking. Dynamic masking was used in the experiments of training the model with RoBERTa because of these effects and the additional performance advantages of dynamic masking.

Training with Large Batches

Where the learning rate is increased appropriately, previous work in Neural Machine Translation has shown that training with very large mini-batches will boost both optimization speed and end-task performance. BERT can also be trained in big batches, according to recent research. BERTBASE was originally trained for 1 million phases with a batch size of 256 sequences. This is the statistical cost of training for 125K steps with a batch size of 2K sequences, or 31K steps with a batch size of 8K sequences, using gradient accumulation [40].

Table 3.2 presents and compares perplexity and end-task performance of $BERT_{BASE}$ as batch size is increased when adjusting for the number of passes through the training data. Perplexity on held-out training data (ppl) and development set accuracy for base models trained over BOOKCORPUS and WIKIPEDIA with varying batch sizes (bsz). The learning rate (LR) was tuned for each setting. Models make the same number of passes over the data (epochs) and have the same computational cost.[61]

bsz	**steps**	**lr**	**ppl**	**MNLI-m**	**SST-2**
256	1M	1e-4	3.99	84.7	92.7
2K	125K	7e-4	3.68	85.2	92.9
8K	31K	1e-3	3.77	84.6	92.8

Table 3.2: Perplexity on Held-out Training Data (ppl) and Development Set Accuracy for Base Models.

Training with large batches improves perplexity for the masked language modeling objective, as well as end-task accuracy. Large batches are also easier to parallelize using distributed data-parallel training, and the model was trained with batches of 8K sequences in the experiments.

Text Encoding

Byte-Pair Encoding (BPE) [50] is a combination of character- and word-level representations that enable natural language corpora to handle large vocabulariesBPE uses subword units instead of complete words, which are derived by statistical analysis of the training corpus. The scale of a BPE vocabulary usually ranges from 10K to 100K subword units. When designing large and diverse corpora, such as the ones considered in this study, Unicode characters will account for a significant portion of the vocabulary.

In their article, Radolf et al proposed a clever implementation of BPE that uses bytes as the base subword units instead of Unicode characters. Bytes allow for the learning of a small subword vocabulary (50K units) that can encode any input text without introducing any "unknown" tokens. The original BERT implementation employs a 30K character-level BPE vocabulary that is taught after the data is preprocessed using heuristic tokenization rules. There are only minor differences between these encodings, with bytes BPE reaching marginally lower end-task performance on some tasks.

3.1.5 Summary

RoBERTa proposes BERT pretraining procedure modifications that increase end-task efficiency. We are now integrating these enhancements and assessing their combined impact. This configuration is known as RoBERTa, which stands for the Robustly Configured BERT Approach. It is also trained with an eight times larger batch size with half as many optimization phases, resulting in four times the number of sequences used in pretraining compared to BERT.

3.2 CamemBERT

3.2.1 Introduction

Leveraging the huge amount of unlabeled texts nowadays available, they provide an efficient way to pre-train continuous word representations that can be fine-tuned for a downstream task, along with their contextualization at the sentence level. This has been illustrated several times in the sense of English using contextualized representations. There are about 7.5 billion people who live in about 200 countries around the world. Just 1.2 billion of them speak English as their first language. As a result, there is a large amount of unstructured non-English textual files. The majority of the studies and articles in English show how to use BERT-based architectures to conduct tasks like text classification, emotion interpretation, query answering, and text generation models. In Natural Language Processing, pertained language models are now common. Despite their popularity, the majority of available models were either trained on English data or data from multiple languages concatenated together. This limits the functional use of such models in all languages except English. To fix this problem, French versions of the Bi-directional Encoders for Transformers (BERT) such as CamemBERT and Flaubert were released.

3.2.2 CamemBERT Architecture

CamemBERT is based on RoBERTa, which replicates and improves the initial BERT by identifying key hyper-parameters for more robust performance. CamemBERT differs from RoBERTa mainly with the addition of whole-word masking and the usage of Sentence-Piece tokenization, CamemBERT is a multi-layer bidirectional Transformer, similar to RoBERTa and BERT. CamemBERT employs the original $BERT_{BASE}$ configuration, which includes 12 layers, 768 hidden lengths, and 12 attention heads for a total of 110M parameters.

3.2.3 Pretraining Objective

The Masked Language Modeling (MLM) task was used to train the CamemBERT model. 15% of tokens were chosen from an input text series made up of N tokens $\mathrm{x}_1, ..., \mathrm{x}_N$ for potential substitution. 80% Using cross-entropy loss, the model is then learned to predict the initial masked tokens. Tokens were masked dynamically during training instead of statically for the entire dataset during preprocessing, as recommended by RoBERTa. When preparing for multiple epochs, this increases variability and makes the model more robust. Since the input sentence is segmented into subwords using Sentence Piece, the input tokens to the models can be sub words [35].

Optimization

The model was optimized using Adam $\beta_1 = 0.9, \beta_2 = 0.98$ for 100k steps. large batch sizes of 8192 sequences were used. Each sequence contains at most 512 tokens. Each sequence is enforced to only contain complete sentence.

Segmentation into Sub-word Units

Input text was segmented into subword units using SentencePiece. SentencePiece is a BPE extension that does not need pre-tokenization (at the word or token level), eliminating the need for language-specific tokenizers [35]

Pretraining Data

By adding more data to pre-trained language models, they can be greatly enhanced. CamemBERT was trained on Common Craw [23] French text. The OSCAR corpus (a pre-classified and pre-filtered version of the November 2018 Popular Craw snapshot) was used specifically [1].

OSCAR is a set of monolingual corpora extracted from Common Crawl, specifically the plain text WET format, which eliminates all HTML tags and transforms all text encodings to UTF-8. The unshuffled version of the French OSCAR corpus, which contains 138GB of uncompressed text and 32.7 billion SentencePiece tokens, was used.

Evaluation

CamemBERT was evaluated on different NLP tasks, which we'll go over in the following sections.

Part-of-speech Tagging and Dependency Parsing

CamemBERT was tested on part-of-speech (POS) tagging and dependency parsing, all of which are downstream tasks. POS tagging is a syntactic task that involves assigning a grammatical category to each word. The goal of dependency parsing is to predict a labeled syntactic tree that captures the syntactic relations between words. The research was performed on the four freely available French UD treebanks in UD v2.2: GSD, Sequoia, Spoken, and ParTUT [35].

Natural Language Inference

The aim of NLI is to figure out whether a hypothesis sentence is implied, neutral, or opposes a proposition sentence. The French portion of the XNLI dataset was used to assess this task. There are 122k train, 2490 valid, and 5010 test examples in the dataset.

It turned out that CamemBERT improves the state of the art for multiple downstream tasks in French. It is also lighter than other BERT-based approaches such as mBERT or XLM.

3.3 FlauBERT

FlauBERT is a BERT variant for the French language. In the sections below, we will go through the training corpus, text preprocessing pipeline, model architecture, and training settings used to create FlauBERTBASE and BERTLARGE, $FlauBERT_{BASE}$. and FlauBERT$_{LARGE}$

3.3.1 Training Data

Data Collection

FlauBERT was trained using a French text corpus that consists of 24 sub-corpora collected from various sources, covering a wide range of topics and writing styles, from formal to well-written text. The information was gathered from three major sources:

1. Monolingual data for French provided in WMT19 shared tasks.
2. French text corpora offered in the OPUS collection.
3. Datasets available in the Wikimedia projects. WikiExtractor tool was used to extract the text from Wikipedia. The total size of the uncompressed text before preprocessing is 270 GB [34].

Data preprocessing

Quite short sentences, as well as repeated and non-meaningful text, such as phone/-fax numbers, email addresses, and so on, were filtered out of all sub-corpora. Before being tokenized with Moses tokenizer [24], all of the data was unicode-normalized. The total size of the training corpus is 71 GB.

3.3.2 Models and Training Configurations

Model Architecture

The model architecture of FlauBERT is the same as that of BERT, which is a multi-layer bidirectional transformer [34]

- $FlauBERT_{BASE}$: warmup steps of 24k, peak learning rate of 6e4,$\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 1e - 6$ weight decay of 0.01.
- $FlauBERT_{LARGE}$: warmup steps of 30k, peak learning rate of 6e4,$\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 1e - 6$ weight decay of 0.01.

The Byte Pair Encoding (BPE) algorithm is used to build a vocabulary of 50K sub-word units. The only distinction between our work and RoBERTa is that the training data is preprocessed and tokenized using a simple tokenizer for French (Moses). $FlauBERT_{BASE}$ is trained on 32 GPUs Nvidia V100 in 410 hours and $FlauBERT_{LARGE}$ is trained on 128 GPUs in 390 hours, both with the effective batch size of 8192 sequences.

Other Training Details

3.3.3 FLAU

FLAU is a French NLP test benchmark (French Language Understanding Evaluation). Three of the six tasks (Text Classification, Paraphrase, and Natural Language Inference) are from cross-lingual datasets, as the aim was to provide findings from a monolingual pre-trained model to aid possible analyses of cross-lingual models, which have recently sparked a lot of interest.

3.3.4 Experiments and Results

The effects of FlauBERT fine-tuning on the FLUE benchmark are presented in this section. On all tasks, the performance of FlauBERT was compared to that of Multilingual BERT and CamemBERT. In addition, we provide the best non-BERT model for each task for comparison.

Text Classification

CamemBERT and FlauBERT outperform mBERT by a wide margin, demonstrating the importance of a monolingual French model for text classification. $FlauBERT_{BASE}$ performs moderately better than CamemBERT in the books dataset, while its results on the two remaining datasets of DVD and music are lower than those of CamemBERT. FlauBERTLARGE achieves the best results in all categories.

Paraphrasing

Table 3.3 summarizes the final accuracy of each model. The monolingual French models do just marginally better than the multilingual variant mBER.

Model	**Accuracy**
ESIM [27]	66.20
mBERT	89.30
CamemBERT	90.14
FlauBERT$_{BASE}$	89.49
FlauBERT$_{LARGE}$	89.34

Table 3.3: Paraphrasing Results on the French PAWS-X Dataset.

Natural Language Inference

On this task, the results confirm the dominance of the French models over the multilingual model mBERT. $FlauBERT_{LARGE}$ outperforms CamemBERT by a small margin. Both of them clearly outperform XLM_{RBASE} , while cannot surpass $XLMR_{LARGE}$.

Despite being trained on almost twice as few text data as CamemBERT, FlauBERT is efficient. Hugging Face's transformers library supports FlauBERT as well.

Chapter 4

Convolutional Neural Networks

Convolutional Neural Network or as it is called sometimes as Deep Convolutional Neural Network (DCNN) has been at the heart of advanced methods in Deep Learning, it is considered also a very useful tool for machine learning practitioners. CNNs have been used since the nineties to solve the basic problem of character recognition, while now their usage becomes wider and popular especially for image classification tasks and other complex Computer Vision tasks [13].

In general, the CNNs are some short Feed-Forward Neural Networks. Beside the similar architecture between the CNNs and the traditional Feed-Forward Networks, the forward and backward passes in both have the same concept, also that applies to the learning algorithms and the weights updating concept, but the CNNs are more advanced and dominant in the image classification and the Computer Vision tasks over the traditional Feed-Forward Networks [44].

In contrast to the traditional Feed-Forward Networks, the CNNs begin with a stack of convolutional layers, as it is referred to by the name, these layers perform some convolutional operations on the network's input which preserves the input spatial information and leads to more robust, generalized and less sensitive to some input's transformations model.

The convolutional layers have a smaller number of learnable parameters compared with the dense layers, and the added pooling layers - which are usually added right after the convolutional layers - reduce the input dimensions, these two factors make the learning process faster and easier to converge [44].

4.1 Overview of CNNs Architecture

There are different variations of the CNN architecture, but in general, as it is shown in Figure 4.1 [44], the CNN architecture consists of one or more repeated blocks of convolutional and pooling - sometimes is called sub-sampling - layers, that form a deep module, followed by either one or more dense layers just like in a standard Feed-Forward Neural Network.

Besides the fact that this is the most popular base architecture found in the literature, there are many other architectures that have been introduced aiming to improve the image classification accuracy and robustness, and also to reduce the computational requirements [44].

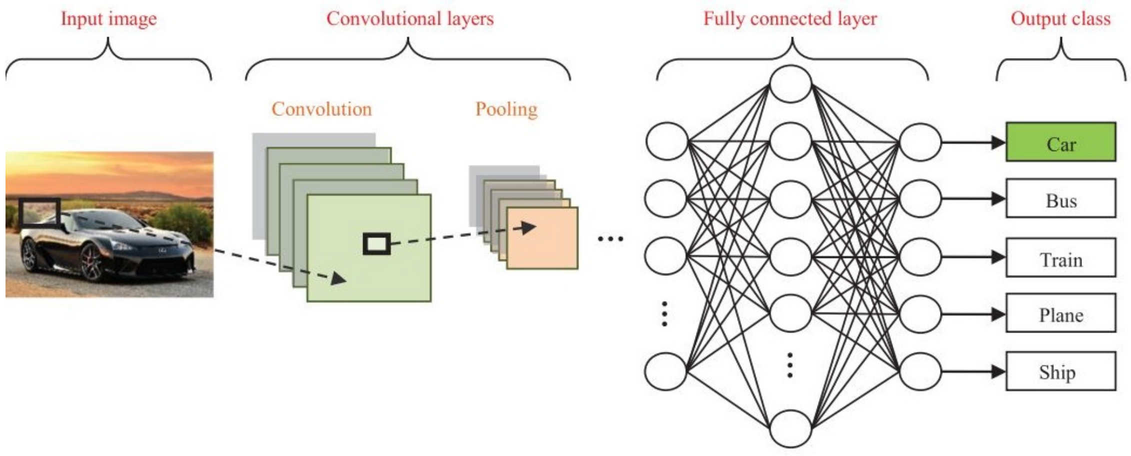


Figure 4.1: The CNN Pipeline

4.1.1 Convolutional Layers

In CNNs, the convolutional layers serve as feature extractors which learn the feature representations from the input images. The neurons in this kind of layers are organized as feature maps, and each neuron in these maps is connected to a set of neighbor neurons in the previous layer using a set of trainable weights [44]. In contrast to the neurons in the dense layers which are connected to every neuron in the previous layer, this fact not only makes the convolutional layer sparse, where few inputs contribute to a given output, but also it makes the treanable weights reusable and could be used in different locations over the input [13].

The inputs to the convolutional layer are convolved using a set of learned weights - namely, kernels - in order to produce a new set of feature maps, which are sent to the next layer after passing it through a nonlinear transformation [44].

Within one convolutional layer, all the neurons have to have the same size, however, each neuron with the same convolutional layer produces a separated feature map using different learned kernels, so multiple features can be learned in the same location [44].

The output of each convolutional layer is a set of one or more feature maps, the number of these feature maps - which is called the depth - depends on the number of kernels or neurons in the layer, where each neuron produces a single feature map based on the input set of maps [44].

The k^{th} output feature map y_k from one layer is calculated as the following:

$$y_k = \varphi(W_k * x) \tag{4.1}$$

Where x is the input image or the feature map set from the previous layer, W_k is the weights set or kernel related to the k^{th} feature map of the layer, φ is a non-linear activation function, finally, the multiplication sign $*$ refers to the convolutional operator which is used to calculate the inner product between the kernel W and each location of the input [44].

Figure 4.2 [13] shows an example of calculating the convolutional operation between an input feature map and a given kernel. The operation is done by sliding the kernel over each single location in the input and calculating the weighted sum between the kernel weights and the aligned locations of the input, this operation produces a single value goes to the equivalent location in the output feature map.

The mentioned example in Figure 4.2 shows a 2D convolutional operation, while in the convolutional layer, this operation is extended to 3D and it is constrained that the depth of each kernel in the layer is the same as the number of the input feature maps, while the number of the output feature maps is the same as the number of kernels in that layer [13].

As is shown in Figure 4.2, the convolutional operation reduces the input size, this reduction depends on two factors, the kernel size and the sliding amount or stride. Sometimes, to avoid this reduction, the input images or feature maps are padded so the produced feature maps can have the same size as the input [13].

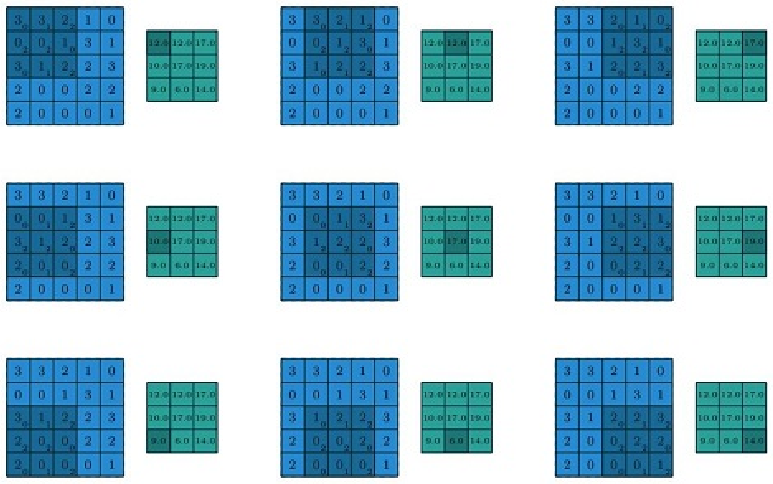


Figure 4.2: Example of Computing the Convolution Output

4.1.2 Pooling Layers

The pooling layers usually take place after a stack of one or more convolutional layers, this pooling step aims to reduce the spatial information of the input feature maps, which leads to less training parameters, and this reduction makes the model more robust against the spatial invariance and translations of the input [44].

The pooling operation works as a function to summarize the subregions of the input [13], in the beginning of CNNs, it was more common to use the average pooling layer to pass the average of the input values, computed from a small neighborhood in the image or the feature map to the next layer, but in the recent architectures the max and min pooling layers are in use too to propagate either the input minimum or maximum to the next layer [44].

As it is shown in Figure 4.3 [44], the pooling works by moving a window over the input feature map with applying the pooling function on this window, this function could be a minimum, maximum or average pooling function [13]

The pooling operation works similarly to the convolutional operation except that the convolutional function is replaced, also the pooling results do not pass through any nonlinear transformations. Usually, a 2×2 sliding window is used for pooling, which reduces the input width and height by half, while the pooling does not affect the input depth due to the fact that it operates in each channel or feature map separately in contrast to the convolutional layers [13].

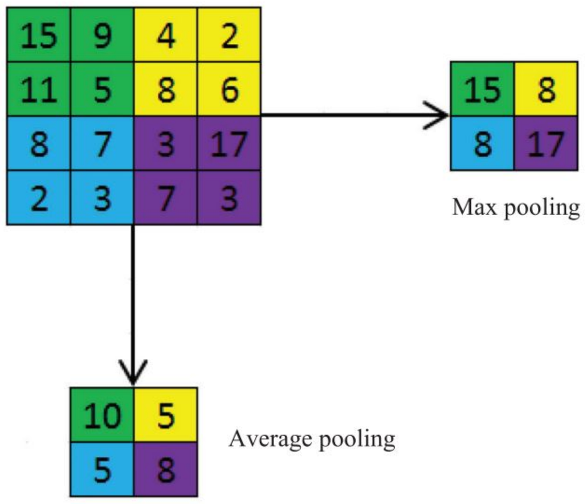


Figure 4.3: Example of Computing the Max and Average Pooling of Input

4.1.3 Fully Connected Layers

After stacking several blocks of convolutional and pooling layers on top of each other, which are used to extract the feature representations of the input, comes the role of the dense layers, the dense or fully connected layers are stacked right after these blocks to interpret the feature representations and perform the high level function of reasoning and decision making [44].

A flattening stage is placed between the last convolutional block and the first fully connected layer, the purpose of this stage is to convert the 3D stack of feature maps into 1D vector, which is fed to the fully connected layers as an input just like in the traditional Feed-Forward Networks. [44].

4.2 ResNet Architecture

The Residual Neural Network (ResNet) is one of the most popular state-of-art CNN architectures, the ResNet was introduced with the concept of Residual Blocks or Residual Learning to solve the problems that appear while training a Deep Neural Networks [52].

In deep networks, the network integrates multi-level features low, mid and high with a classifier in an end-to-end multi-layer fashion the level of these features depends on the number of stacked layers or the depth of the network [18]. It has been proven that the depth of the network is critical and an important factor in the network performance [52], and many experiments using deep networks have shown great benefits [18, 54].

As the deep models were showing significant performance on solving some problems, it started to be noticeable that just stacking more layers on top of each other does not always lead to better performance [18]. The reason of this is the problem of the gradients vanishing or exploding that could be a big obstacle in front of the model convergence [16]. However, a solution was introduced to solve this problem by normalizing the initial weights of the network and by applying the normalization between layers too, this normalization makes the deep networks able to converge even with tens of layers [24].

When the deep networks were able to converge, another problem has appeared which is the degradation problem, this problem appears when increasing the depth of the network, but the accuracy gets stuck and then degrades [18]. As was not expected this was not an over-fitting problem and this problem was not solved by making the network deeper [18].

The training accuracy degradation problem brings the idea that not all networks are similarly easy to optimize, even if the network architectures are similar. Kaiming He in his paper [18] trained a shallow model and its deeper counterpart on the same problem, the deeper version showed a higher training and testing error. Therefore a solution was proposed by constructing the deep model in such way that the added layers are identity mapped and copied from the shallower learned layers. This will constrain that the deeper model will not produce a higher error than the shallower model [18].

4.2.1 Residual Learning

The architecture of deep residual networks consists of many stacked residual blocks or units, where these blocks help to eliminate the degradation problem [19]. In contrast to the traditional CNN architecture, ResNet architecture does not stack the layers directly on top of each other to learn the desired mapping, instead, the layers in ResNet are supposed to explicitly learn a residual mapping [18], using the residual blocks and it is constrained that all the layers within a block have the same output shape [19].

Formally speaking, referring to the residual block presented in Figure **??** [18], and referring to the desired mapping function by $H(x)$, this function is supposed to be learned by some stacked layers and not necessarily by the whole network, where x is the input to the first layer of this stack. Here, if a stack of non-linear layers can learn the mapping of a complex function, then this stack of layers can learn the residual function $H(x) - x$, where both the input and the output have the same dimensions. Therefore, instead of making the stack of layers learn the mapping function $H(x)$ it can learn an approximate residual function $F(x) = H(x) - x$ and then it is referred to the original function by $F(x) + x$ where both forms should be able to learn the desired mapping [18].

Rewriting the mapping in this formula is motivated by the fact that the deeper models produce a higher training error than the shallower models. As it is difficult to learn the identity mappings using multiple non-linear layers, with this reformulation, when the identity mapping is optimal then the solver can simply push the weights of the nonlinear layers to zero in order to get the identity mapping. However, in the real-world, the optimal identity mapping is hard to learn, but this reformulation can help the pre-defined condition to become true [18]. Residual block is shown in Figure 4.4 [23].

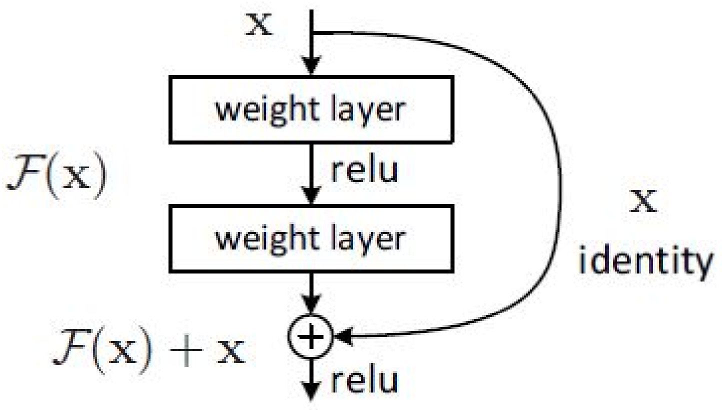


Figure 4.4: Residual Block

4.2.2 Shortcuts for Identity Mapping

In ResNet, the residual learning is applied to every stack of few layers among the entire network, where the network is divided into a group of building blocks or residual blocks as it is shown in Figure **??**. The output of each residual block y is defined as the following:

$$y = F(x, \{W_i\}) + x \tag{4.2}$$

Where x is the input of the residual block and $F(x, \{W_i\})$ is the residual mapping to be learned through the block. The addition $F(x, \{W_i\}) + x$ is implemented by a skip or shortcut connection and addition operation, while another nonlinearity could be applied after the addition.

This skip connection implies some proprieties that could be mentioned as the following [19]:

- The input feature representations fed to a deeper block can be the same as feature representations fed to shallower blocks plus the residual mapping. Which indicates that residual mapping happens between the blocks in the model.
- The input to any deeper block is the summation of the output of all the previous residual mappings in contrast to the traditional feedforward networks.
- The signals can be directly passed from one block to another in either forward or backward passes.

The input and the output of the residual block must have the same dimensions, otherwise a linear projection is applied to the input x in order to match the output dimensions and perform the addition operation [18].

The skip connection has no trained parameters, also it does not add any complexity to the model. This fact helps in practice to compare two models with and without the skip connection but with the same network architecture like the number of parameters, layers, and depth [18].

4.3 Squeeze-and-Excitation Networks

The convolution operator, which enables networks to create informative features by fusing both spatial and channel-wise information within local receptive fields at each layer, is the core building block of convolutional neural networks (CNNs). A wide variety of previous research has examined the spatial aspect of this relationship, with the goal of improving the consistency of spatial encoding within the function hierarchy of a CNN to improve its representational power. The "Squeeze-and-Excitation" (SE) block focuses instead on the channel, this block adaptively recalibrates channel-wise feature responses by explicitly modelling the interdependencies between channels. These blocks can be stacked to create SENet architectures that generalize well over a wide range of datasets. SE blocks increase the efficiency of current state-of-the-art CNNs by a considerable margin at a low computational cost.

4.3.1 Overview

For any given transformation F_{tr} mapping the input X to the feature maps U where U $\in$R$_{HxWxC}$, e.g., a convolution, we can construct a corresponding SE block to perform feature recalibration. The features U are first put through a squeeze process, which aggregates feature maps over their spatial dimensions H_W to generate a channel descriptor. This descriptor's purpose is to create an embedding of the global distribution of channel-wise feature responses, enabling all of the network's layers to use information from the global receptive field. Following the aggregation, an excitation operation is performed, which takes the form of a simple self-gating mechanism that takes the embedding as input and outputs a set of per-channel modulation weights. These weights are applied to the function map U to produce the SE block's output, which can be directly fed into the network's subsequent layers [23]. The structure of the SE building block is depicted in Figure 4.5 [23].

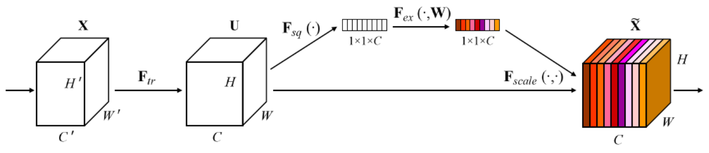


Figure 4.5: Squeeze-and-Excitation Block

By simply stacking a set of SE blocks, a SE network (SENet) can be created. Furthermore, at various depths in the network architecture, these SE blocks may be used as a drop-in substitute for the original node. Although the building block's template is generic, the function it plays at various depths varies across the network.

- It strengthens the shared low-level representations by excitating insightful features in earlier layers in a class-agnostic manner.
- As the SE blocks progress through the layers, they become more advanced and react to various inputs in a highly class-specific manner.

As a result, the advantages of SE blocks' function recalibration can be accumulated throughout the network.

4.3.2 Squeeze-and-Excitation block

A Squeeze-and-Excitation block is a computational unit which can be built upon a transformation For mapping an input X RH0xW0xC0 to feature maps U RHxWxC.

4.3.3 Squeeze: Global Information Embedding

To begin addressing the problem of channel dependencies, we look at the signal to each channel in the output features. As a result, each unit of the transformation output U is unable to exploit contextual information outside of this region because each of the trained filters works with a local receptive field. To mitigate this problem, we propose to squeeze global spatial information into a channel descriptor. This is achieved by using global average pooling to generate channel-wise statistics.

$$Z_c = F_{sq}(u, c) = \frac{1}{W * H} \sum_{i=1}^{H} \sum_{j=1}^{W} uc(i, j) \tag{4.3}$$

The transformation's result U can be thought of as a collection of local descriptors whose statistics are representative of the entire image.

4.3.4 Adaptive Recalibration

The second operation attempts to completely collect channel-wise dependencies in order to make use of the information gathered in the squeeze operation. The feature must follow two conditions to achieve this goal:

- First and foremost, it must be adaptable (in particular, it must be capable of learning a nonlinear interaction between channels).
- second, it must learn a non-mutually-exclusive relationship since we would like to ensure that multiple channels are allowed to be emphasized. A simple gating mechanism was employed with a sigmoid activation

$$s = Fex(z, W) = s(g(z, W)) = s(W_2 d(W_{1z})) \tag{4.4}$$

where d refers to the ReLU function, W$_1$∈ $\mathbb{R}^{C_r/C}$ and W$_2$ ∈ $\mathbb{R}^{C/C_r}$

Schema of original residual and SE-ResNet modules is illustrated in Figure 4.6 [23].

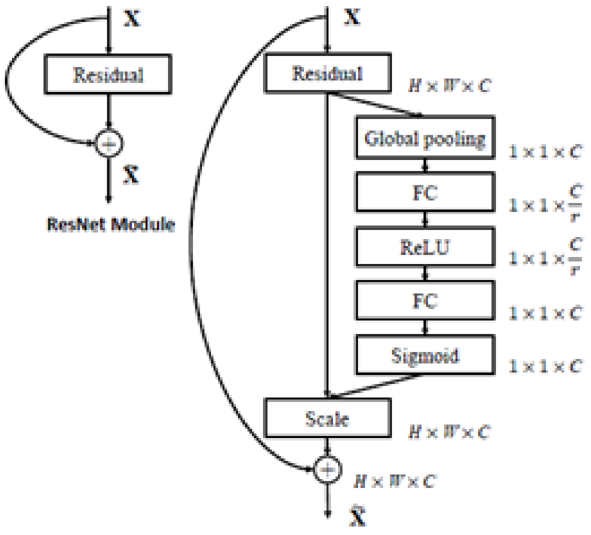


Figure 4.6: The Schema of Original Residual and SE-ResNet Modules.

The excitation operator converts the z descriptor in the input to a set of channel weights. In this regard, SE blocks implement input-dependent dynamics, which can be viewed as a self-attention mechanism on channels whose relationships are not limited to the local receptive field that the convolutional filters respond to.

4.3.5 SE Block Integration Strategy

The SE block is compatible with current architectures. We consider three variations in addition to the SE design:

- SE-PRE block: in which the SE block is moved before the residual unit.
- SE-POST block: in which the SE unit is moved after the summation with the identity branch (after ReLU).
- SE-Identity block: in which the SE unit is placed on the identity connection in parallel with the residual unit.

These variants are illustrated in Figure 4.7 [23].

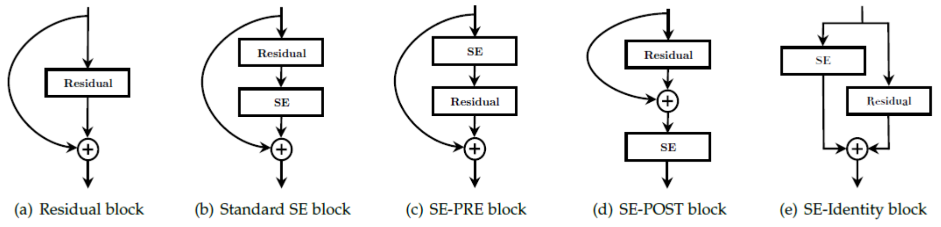


Figure 4.7: SE bBock Integration Designs

4.3.6 Conclusion

The SE block is an architectural unit that allows a network to perform dynamic channel-wise feature recalibration to increase its representational power. SENets have been shown to be successful in a variety of experiments, achieving state-of-the-art efficiency through various datasets and tasks. Furthermore, SE blocks shed light on previous architectures' failure to properly model channel-wise feature dependencies. We hope that this knowledge can be helpful in other tasks that require powerful discriminative features. Finally, the feature importance values produced by SE blocks can be useful for other tasks such as network pruning for model compression.

4.4 Performance Measures

The performance measures are used to assess and analyse the performance of a Deep Learning or Machine Learning model. These measures vary depending on the problem type, where each problem has its own applicable measures, for example, the regression performance measures are different from those which are used for assessing the performance for a classification problem [53, 7]. Since this research concerns a classification problem, it is discussed in the following some classification performance measures like accuracy, precision and recall.

The Accuracy, Precision, and Recall are all calculated based on some terms observed from the model predictions, these terms are listed as True Positives (TP), False Positives (FP), True Negatives (TN) and False Negatives (FN), and they are defined as the following taking into consideration a binary classification problem of negative class and positive class [53, 7]:

- True Positives: the number of positive samples which were correctly classified as positive class.
- False Positives: the number of negative samples which were incorrectly classified as positive class.
- True Negatives: the number of negative samples which were correctly classified as negative class.
- False Negatives: the number of positive samples which were incorrectly classified as negative class.

4.4.1 Accuracy

The Accuracy is considered as the most popular performance measure in the domain of classification problems, it is also defined as the total effectiveness of a given classifier or the ratio of the total number of correctly classified samples to the total number of samples as Equation 4.5 shows [7].

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.5}$$

Sometimes, measuring the performance by using the accuracy only might not be so realistic. These cases happen when the classes are highly imbalanced, where the samples are more likely to be classified to the largest class, however, some other measures can be combined with the Accuracy [22].

4.4.2 Precision

The Precision, as presented in Equation 4.6, can be interpreted as the ratio of correctly classified positive samples to the total number of samples which were classified as positive. The Precision is useful to measure the prediction confidence, especially when the classes are highly imbalanced [7, 22].

$$Precision = \frac{TP}{TP + FP} \tag{4.6}$$

4.4.3 Recall

The Recall measures the effectiveness or sensitivity of a classifier to identify positive samples, as presented in Equation 4.8, it is calculated as the ratio correctly classified positive sample to the total number of positive samples. The Recall is usually used side by side with precision to assess the model performance [53].

$$Recall = \frac{TP}{TP + FN} \tag{4.7}$$

4.4.4 F1-Score

This is the harmonic mean of precision and recall and gives a better measure of the incorrectly classified cases than the Accuracy Metric. We use the Harmonic Mean since it penalizes the extreme values.

$$F1 - Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4.8}$$

Chapter 5

Models Fusion and Transfer Learning

5.1 Introduction

In various disciplines, information about the same phenomenon can be acquired from different types of detectors, at different conditions, in multiple experiments or subjects, among others. We use the term “modality” for each such acquisition framework. Due to the rich characteristics of natural phenomena, it is rare that a single modality provides complete knowledge of the phenomenon of interest. The increasing the availability of several modalities reporting on the same system introduces new degrees of freedom, which raise questions beyond those related to exploiting each modality separately. As we argue, many of these questions, or “challenges”, are common to multiple domains [1]. This chapter deals with two key questions: “why we need data fusion” and “how we perform it”.

5.2 Diversity and Data Fusion

We will concentrate on configurations in which various instruments, measuring systems, or framework procedures are used to observe a system. Each acquisition framework is referred to as a modality and is correlated with a single dataset in this case. Multimodal refers to the entire setup in which one has access to data collected from various modalities. Complementarity, in and of itself, is a central property of multimodality.

5.2.1 Diversity

"Diversity is a property that allows to enhance the uses, benefits and insights in a way that cannot be achieved with a single modality." [9]

5.2.2 Data Fusion

"Data fusion is the analysis of several datasets such that different datasets can interact and inform each other" [9]

Multimodality is defined as a structure that can be observed using several sensors. The goal of multimodality is to collect and combine essential data from individual sensors, then use this combined function to solve a problem. As a result, the predicted output would have a richer representation.

5.3 Data Fusion Techniques

There are three methods that are used to fuse multimodal data [9]. We will go through them in the following sections.

5.3.1 Early Fusion or Data-Level Fusion

This approach is used traditionally to fuse multiple data before conducting the analysis. It is known as "input level fusion" [9]. Researchers [60] proposed two different methods for early fusion strategy. The first method involves integrating data by eliminating the correlation between two sensors. The second method involves fusing data in a lower-dimensional common space. Principal component analysis (PCA), canonical correlation analysis, and independent component analysis.

Early fusion could be applied to raw or preprocessing data obtained from sensors. Data features should be extracted from the data before fusion, otherwise, the process would be difficult, especially if the data sources have different sampling rates between modalities. Synchronization of data sources is not a trivial task when one data source is discrete and the others are continuous. Therefore, converting data sources into a single feature vector is an essential prerequisite in early data fusion. This approach is illustrated in Figure 5.1. The Concept of early data fusion is the conditional independence between multiple data sources. According to Sebe et al.

[49]), this concept is not always valid since different modalities can have highly correlated features, for example, video and depth cues. Another paper [2] also shows that multiple modalities can contain information that is correlated with each other at a higher level. Poria et al. [47] used early-stage data fusion which implemented concatenation of the features in a multimodal stream, this can be considered as the most basic and simplest form of early-stage data fusion as demonstrated in Figure 5.1 [48]

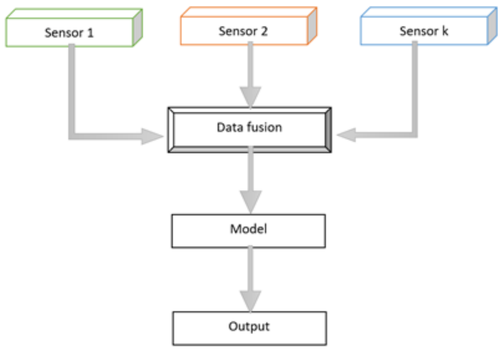


Figure 5.1: Early Fusion or Data-level Fusion

The use of early-stage data fusion has two drawbacks. One of the major drawbacks of this approach is that a significant amount of data would be deducted from the modalities before fusion to provide a common ground. A machine learning algorithm is used to analyze the data until it has common matrices. Another drawback to this approach is the inability to synchronize the timestamps of the various modalities. Collecting data or signals at a common sampling rate is a common approach to solve this drawback. Martinez et al. [?] recommend other mitigation strategies such as training, pooling, and convolution fusion. The proposed approaches were created by combining concurrent discrete event simulations.

5.3.2 Late Fusion or Decision Level Fusion

Late fusions make use of data sources individually followed by fusion at a decision-making stage Figure 2. The success of ensemble classifiers [31] motivated late data fusion. This method is much easier to use than the early fusion method, especially when the data sources differ significantly in terms of sampling rate, data dimensionality, and measurement units. Late fusion also produces improved results when errors from different models are dealt with separately, resulting in uncorrelated errors.

Ramachandram et al. [11] contend, however, that there is no definitive proof that late fusion outperforms early fusion. Despite this, a number of researchers [?] [15] use late or judgment stage fusion to analyze multimodal data issues. There are a variety of guidelines for determining the best way to integrate each of the individually trained models. Four of the most widely used late fusion rules are Bayes rule, max-fusion, and average-fusion. Late fusion is a simplified and more scalable solution where the input data sources are substantially different in terms of dimensionality and sampling rate. This fusion fashion is illustrated in Figure 5.2 [48]

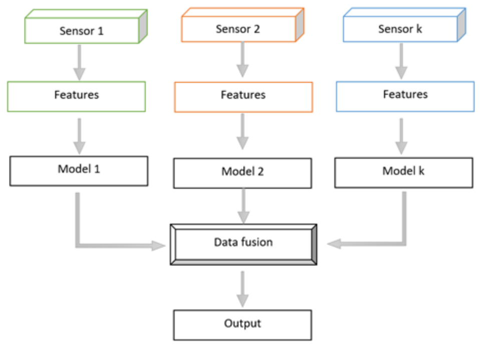


Figure 5.2: Late Fusion or Decision Fusion

5.3.3 Intermediate (Hierarchical) Fusion

The intermediate fusion architecture is based on the well-known deep neural network. This approach is the most flexible, allowing for data fusion at various levels of model training. The efficiency of neural network-based multimodal data fusion has significantly improved. Intermediate fusion changes input data into a higher level of representation (features) through multiple layers. Each layer uses linear and nonlinear functions to change the size, skew, and swing of the input data, resulting in a new representation of the original data. In a deep learning multimodal sense, intermediate fusion is the fusion of several modality representations into a single hidden layer, allowing the model to learn a joint representation of all modalities. Different types of layers, such as 2D convolution, 3D convolution, and fully connected layers, may be used to learn features. A fusion layer, also known as a shared representation layer, is the layer where various modality elements have been fused.

Different modalities may be fused into a single shared representation layer at the same time, or one or more modalities can be fused at a time (Figure 5.3). While fusing multiple modality features or weights in a single layer is feasible, it can result in model overfitting or the network failing to learn the relationship between each modality. Reducing the data's dimensionality is one way to increase deep multimodal fusion efficiency. After building a fusion layer, Li et al [8] use principal component analysis (PCA) and Ding et al [12] use autoencoders to reduce the network's dimensionality. Intermediate fusion, as opposed to early fusion level and late level fusion, allows for the fusion of features at varying depths. Karpathy et al. [1] use a "slow-fusion" network in which training video stream features are fused gradually through several fusion layers. In a large-scale video stream classification problem, this method performs better. Other research [37] demonstrates a gradual fusion approach in which highly correlated input modalities are fused first, followed by less correlated ones. This fusion method is shown in Figure 5.3 [48]

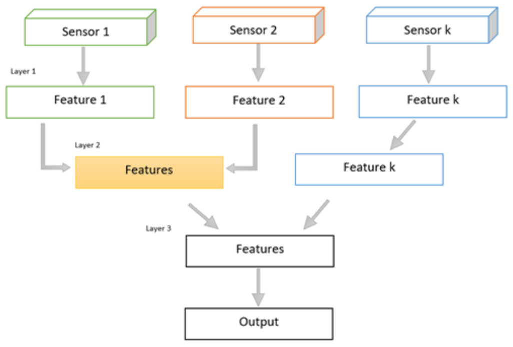


Figure 5.3: Intermediate Fusion

In our proposed model, we used the hierarchical fusion. Having used three different pertained models on text and image data, we fused the properties for each model individually and obtained corresponding weights for each of the fusion processes. These weights are again used in a second layer of fusion where we fused the three embeddings together to get the weights of the final model that is used for the product type code classification. Further explanation of the structural details could be viewed in chapter 6.

5.4 Transfer Learning

Humans are born with the capacity to transfer information from one task to another. What we learn as experience when learning about one task, we apply to other tasks in the same manner. The more related the tasks, the easier it is for us to transfer, or cross-utilize our knowledge. When we seek to understand new aspects or subjects, we do not start from the scratch. We use what we have learned in the past to transfer and leverage our expertise. "Transfer learning is the idea of overcoming the isolated learning paradigm and utilizing knowledge acquired for one task to solve related ones."In their famous book, Deep Learning, Goodfellow et al [48] refer to transfer learning in the context of generalization. Their definition is as follows: "Situation where what has been learned in one setting is exploited to improve generalization in another setting."

5.4.1 Motivation for Transfer Learning

The main motivation, particularly in the sense of deep learning, is that most models that solve complex problems need a large amount of data, and obtaining large amounts of labeled data for supervised models can be extremely difficult, given the time and effort required to label data points. The ImageNet dataset, for example, has millions of images relating to various categories, thanks to years of hard work at Stanford. Obtaining such a dataset for each domain, however, is difficult. Furthermore, most deep learning models are extremely focused on a single domain or even a single task. Although these models could be cutting-edge, with very high precision and the ability to outperform all benchmarks, they will only be trained on very specific datasets and would experience a substantial loss in the performance when applied to a new task that was close to the one it was trained on. This is the motivation of transfer learning, which looks at existing tasks and domains to see if information from pre-trained models can be applied to new problems

5.4.2 Transfer Learning Concept

Transfer learning is not a concept that is unique to deep learning. The traditional approaches to build and train machine learning models based on transfer learning theory are vastly different. Figure 5.4 [48] compares traditional and transfer learning.

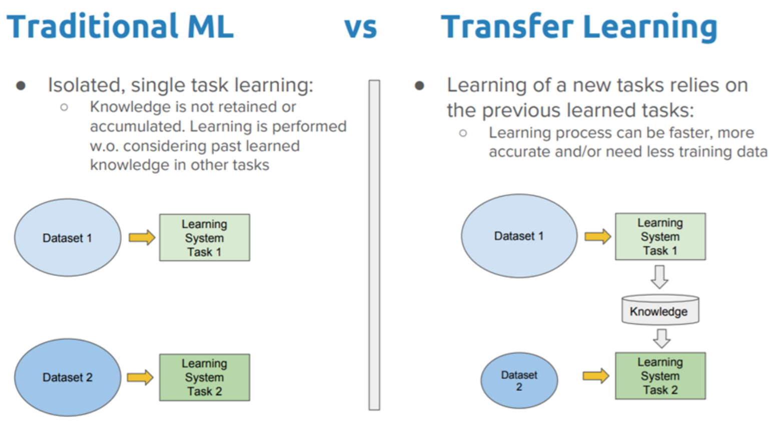

Traditional ML
vs
Transfer Learning
Isolated, single task learning:
Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks
Dataset 1
Learning System Task 1
Dataset 2
Learning System Task 2
Learning of a new tasks relies on the previous learned tasks:
Learning process can be faster, more accurate and/or need less training data
Dataset 1
Learning System Task 1
Knowledge
Dataset 2
Learning System Task 2

Figure 5.4: Traditional Learning VS Transfer Learning

Traditional learning is isolated, and it happens only on specific tasks, datasets, and the training of different isolated models on them. There is no knowledge that can be passed from one model to another. Transfer learning allows you to use previously trained models' experience (features, weights, etc.) to train newer models and even solve challenges like getting less data for the newer challenge.

5.5 Applications of Transfer Learning

5.5.1 Learning From Simulations

Gathering data and training a model in the real world is either costly, time-consuming, or just too risky for many machine learning applications that rely on hardware for interaction. As a result, it is preferable to collect data in a less dangerous manner. The preferred method for this is simulation, which is used to enable many advanced ML systems in the real world. Transfer learning scenario is where you learn from a simulation and then incorporate what you have learned in the real world. The feature spaces in the source and goal domains are similar (both depend on pixels), but the marginal probability distributions between simulation and reality are different, i.e. objects in the simulation and the source look different. When simulations get more practical, the gap between them shrinks. Simulated and real-world conditional probability distributions can vary because simulations cannot fully reproduce all reactions in the real world, for example, a physics engine cannot perfectly imitate the complicated interactions of real-world objects. Learning from simulations has the advantage of facilitating data gathering while also allowing for rapid training by allowing learning to be parallelized over many instances. As a result, consequently, it is a prerequisite for large-scale machine learning projects that would communicate with the physical world, such as self-driving cars.

Robotics is another field where learning from simulations is critical: training models on a real robot is too sluggish, and robots are costly to train. This dilemma is solved by learning from a simulation and passing the information to a real-world robot. An example of a data manipulation task in the real world and in a simulation can be seen in Figure 5.5 [46].

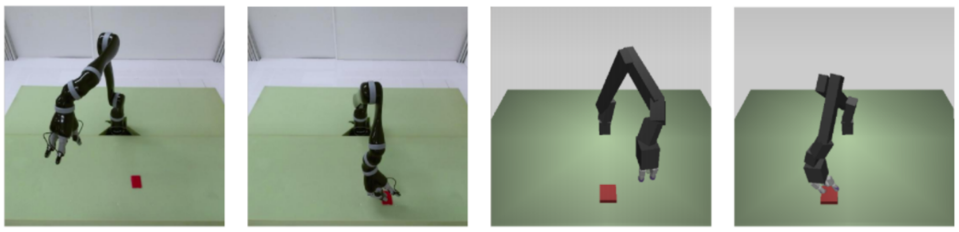


Figure 5.5: Robot and Simulation images

5.5.2 Adapting to New Domains

Although learning from simulations is one type of domain adaptation, there are a few other types of domain adaptation to consider. Adapting to various document styles is another general domain adaptation scenario: Standard NLP tools including part-of-speech taggers and parsers are usually trained on news data like the Wall Street Journal, which has traditionally been used to test these models. Models trained on news data, on the other hand, struggle to deal with more novel text sources like social media messages and the difficulties they pose. People use different terms and phrases to articulate the same viewpoint even within the same domain, such as product reviews. To avoid being surprised by the change in domain, a model trained on one form of review should be able to separate the common and domain-specific opinion terms that people use.

5.5.3 Transferring Knowledge Across Languages

Another fantastic application of transfer learning is learning from one language and adapting what we have learned to another. Reliable cross-lingual adaptation methods would allow us to leverage the vast amounts of labeled data we have in English and apply them to any language, particularly underserved and truly low-resource languages. With the current state of the art, this still seems utopian, but recent advancements in this field, such as zero-shot translation [24], promise rapid change.

5.6 Adaptation

There are several adapting ways we can decide on when applying a pre-trained model to a target task: architectural modifications, optimization schemes, and whether to receive more signal.

5.6.1 Architectural Modifications

For architectural modifications, the two general options we have are the following:

Keep the Pre-Trained Model Internals Unchanged

Adding one or more linear layers on top of a pre-trained model, as is usually done with BERT, is an easy way to accomplish this. Instead, we can feed the model output into a different model, which is useful when a target task requires interactions that are not possible in the pre-trained embedding, such as span representations or modeling cross-sentence relations.

Modify the Pre-Trained Model Internals Architecture

One reason why we would want to do this is to adapt to a structurally different target task, such as one with several input sequences. We will use the pre-trained model to initialize as much of a structurally different target model as possible in this situation. We will also choose to make task-specific adjustments and modifications, such as adding skip or residual connections or attention. “Finally, modifying the target task parameters may reduce the number of parameters that need to be fine-tuned by adding bottleneck modules ("adapters") between the layers of the pre-trained model" [41].

5.6.2 Optimization Schemes

When it comes to improving the model, we can select which weights to update, as well as how and when to do so. For updating the weights, we can either tune or not tune (the pre-trained weights).

Do Not Change the Pre-Trained Weights (Feature Extraction)

"In practice, a linear classifier is trained on top of the pre-trained representations. The best performance is typically achieved by using the representation not just of the top layer, but learning a linear combination of layer representations" . In a downstream model, pre-trained representations can also be used as features. Only the adapter layers are educated when connecting adapters.(Peters et al., 2018) [41].

5.6.3 Change the Pre-Trained Weights (Fine-tuning)

The downstream model's parameters are initialized using the pre-trained weights. During the adaptation process, the entire pre-trained architecture is then trained.

5.6.4 How and When to Update the Weights?

The main motivation for choosing the order and how to update the weights is that we want to avoid overwriting useful pretrained information and maximize positive transfer. Related to this is the concept of catastrophic forgetting. (McCloskey Cohen) [41]. This happens when a model forgets the role it was trained on. In most cases, we just think about the performance on the target task, but this varies depending on the application. A guiding principle for updating the parameters of our model is to update them progressively from top-to-bottom in time, in intensity, or compared to a pre-trained model:

Progressively in Time (Freezing)

The basic idea is that training all layers at the same time on data with varying distributions and tasks would lead in instability and bad results. Instead, we train the layers one at a time to allow them to adapt to the new role and data. This goes back to layer-wise training of early deep neural networks. Recent approaches mostly vary in the combinations of layers that are trained together; all train all parameters jointly in the end. Unfreezing has not been investigated in detail for transformer models.

Progressively in Intensity (Lower Learning Rates)

We want to use lower learning rates to avoid overwriting useful information. Lower learning rates are particularly important in lower layers (as they capture more general information), early in training (as the model still needs to adapt to the target distribution), and late in training (when the model is close to convergence). To this end, we can use discriminative fine-tuning (Howard and Ruder) [21]

Progressively vs. a Pre-Trained Model (Regularization)

Using a regularization word, one approach to reduce forgetting is to enable the target model parameters to remain close to the parameters of the pretrained model (Wiese et al., CoNLL 2017)[58].

5.7 Trade-offs and Practical Considerations

The more parameters you would train from scratch, the slower your training would be. Feature extraction necessitates adding more parameters than fine-tuning (Peters et al., 2019) [42] so is typically slower to train. When a model has to be adapted to a variety of tasks, however, feature extraction is more space-efficient as it only requires storing one copy of the pre-trained model in memory. Adapters strike a balance by adding a small number of additional parameters per task. No adaptation approach is simply superior in any setting in terms of performance. Feature extraction appears to be preferable where the source and target tasks activities are dissimilar. Otherwise, feature extraction and fine-tuning also work similarly, but this is dependent on the hyper-parameter tuning budget available (fine-tuning may often require a more extensive hyper-parameter search). Anecdotally, Transformers are easier to fine-tune (less sensitive to hyper-parameters) than LSTMs and may achieve better performance with fine-tuning. When fine-tuned to tasks with limited training sets, large pre-trained models (e.g. BERT-Large) are vulnerable to degenerate performance. In reality, the observable behavior is always "on-off": the model either works very well or does not work at all. Understanding the conditions and causes of this behavior is an open research question.

5.8 Conclusion

Though pre-trained models showed interesting results empirically on a wide spectrum of machine learning tasks, the shortcomings of pretrained language models are still there when considering some tasks. Pre-trained language models are still bad at fine-grained linguistic tasks, hierarchical syntactic reasoning (Kuncoro et al., 2019), and common sense (when you actually make it difficult; Zellers et al., 2019) [62]. They still fail at natural language generation, in particular maintaining long-term dependencies, relations, and coherence. They also tend to overfit to surface form information when fine-tuned and can still mostly be seen as ‘rapid surface learners’.

Chapter 6

Proposed Model and Experiments

6.1 Introduction

In this chapter, we will present our approach for a 'Multi-modal Product Classification' task. The specific objective is to build a model that classifies previously unseen products into their corresponding product type codes. We proposed a deep Multi-Modal Multi-level Fusion Learning Framework used to categorize large-scale multi-modal (text and image) product data into product type codes using a transfer learning approach.

6.2 Proposed Model: Hierarchical Fusion

The hierarchical model made use of state-of-the-art architectures in the base models (pre-trained models) . When building the fusion model, we tried to keep it as simple as possible since a simple architecture may show a close performance with sparing complexity and cost.

The fusion architecture is a simple trainable module. The building process took into consideration the careful analysis of the dataset besides the computational power shortage and it was developed in two phases with considering the evaluation on the validation and test (when available) sets as the following:

- Running multiple experiments with different combination methods in both intermediate and final fusion layers, these methods were : concatenation, addition and average. The model with the best performance was selected for the next phase.

- Stacking more layers after fusing the logits of the base models, including fully connected, non-linearity and dropout layers with different probabilities, and analysing the performance.

6.2.1 Uni-Modal Base Models

The proposed model made use of the following pre-trained image and text models.

SE-ResNeXt Model

We fine-tuned SE-ResNeXt with 50 layers [23] pre-trained on the ImageNet weights using the product images. Visual features from SE-ResNeXt model, $Z_{im} \in R^{p \times p \times d_1}$ where $p \times p$ denotes the size of the output feature maps and d_1 represents the dimension of feature channel, are passed to an adaptive average pooling layer to output X_m where $X_m \in R^{d_1}$. A linear layer is added on top with output dimension equal to the number of classes using a linear layer.

FlauBERT Model

We extracted the first token of the FlauBERT's hidden-states output sequence [34] from the last layer, $Z_{t_1} \in R^{m \times d_2}$ where m denotes the sequence length of the text inputs and d_2 represents the model's hidden size. Extracted token embedding, represented by $X_{t_1} \in R^{d_2}$ is projected to the output dimension equal to the number of classes using a linear layer. Flaubert model along with the classification head is fine-tuned for the task of product classification.

CamemBERT Model

We fine-tuned CamemBERT [35] for product classification by adding classification head on top of the hidden states output of the last layer of the model, $Z_{t_2} \in R^{m \times d_2}$. Output Z_{t_2} is flattened and passed through a linear layer which outputs $X_{t_2} \in R^{d_2}$ followed by tanh activation and a linear layer of output dimension equal to the number of classes.

The above-mentioned intermediate vector and tensor representations were used as inputs to learn simultaneously from image and textual features for the task of product classification. End-to-end multimodal methodology is described in the section below.

6.2.2 Multi-modal Joint Representation Learning

In this section, we describe the multi-modal joint learning of the visual and textual features derived from the data for the product classification task.

The visual feature extracted by passing the input images through the SE-ResNeXt-50 model is represented by X_{im} where $X_{im} \in R^{d_1}$. Similarly, the textual features extracted by passing the textual titles and descriptions through CamemBERT and FlauBERT models are represented by X_{t_1} and $X_{t_2} \in R^{d_2}$. We use a Convolution 1D layer to project the image embeddings from $X_{im} \in R^{d_1}$ to $X'_{im} \in R^{d_2}$ in the same dimensional space as the text embeddings R^{d_2}.

We tried two different multi-modal fusion architectures: baseline fusion and hierarchical fusion. In those architectures, we used different operations to fuse the embeddings as follows:

- **Addition Ensemble:** this operation is defined as the sum of input representations, whereas, in our configuration the input size $\in R^{d_2}$, where $d_2 = 768$.
  $X_{add} = X_{t_1} + X_{t_2}$, where $X_{add} \in R^{d_2}$.
- **Concatenation Ensemble:** this operation is defined as the representations concatenation across a dimension. In terms of the base models:
  $X_{con} =< X_{t_1} + X_{t_2} >$, where $X_{con} \in R^{d_2 \times 2}$ and $<>$ is the concatenation.
- **Average Ensemble:** this operation is defined as the mean of the input representations. $X_{avg} = mean(X_{t_1}, X_{t_2})$, where $X_{avg} \in R^{d_2}$.

6.2.3 Architecture

In the proposed architecture, we fed the products' title and description features individually to the textual base models, resulting in four distinct textual embeddings. In an intermediate fusing layer, logits of textual base models (product title and description) corresponding to CamemBERT were combined. The same was applied for FlauBERT model, resulting in extracted embeddings represented by $X_{t_1}, X_{t_2} \in R^{d_2}$ respectively. Where $d_2 = 768$ is model hidden size.

When training the fusion model, the uni-modal network layers were freezed , a concatinated field of product title and description was used as textual input to the final fusion module along with the reduced image representation. Thus, the input to the final fusion module was $X_{t_1}, X_{t_2}, X'_{im} \in R^{d_2}$.

We trained our proposed multi-modal architecture by combining $X_{t_1}, X_{t_2}, X'_{im}$ in different ways, then projecting the resulting embeddings to the output dimension equals to the number of classes with a linear layer.
The operations used in the fusion layers resulted in different variations of the proposed hierarchical architecture. Figure 6.1 showcases the hierarchical architecture using addition operation in the intermediate fusion layer and concatenation operation in the final fusion layer.

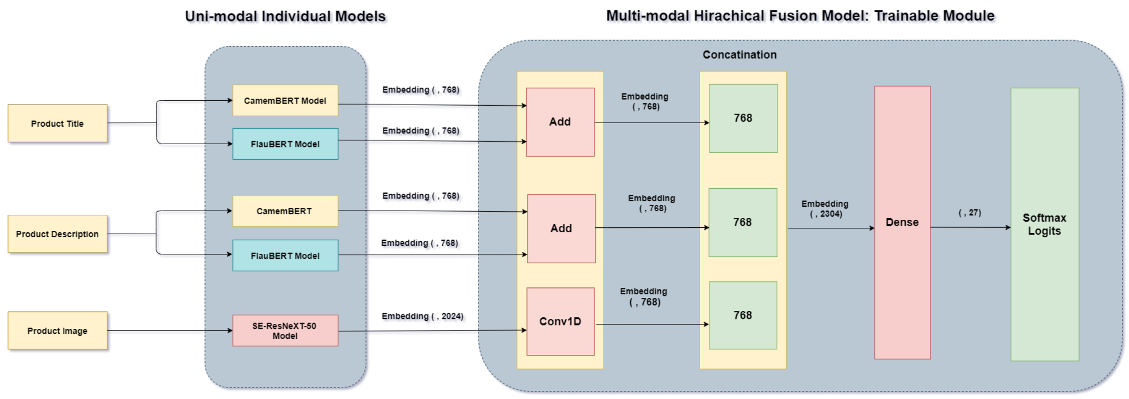


Figure 6.1: Multi-Modal Representation: Hierarchical Fusion

6.3 Experimental Settings

6.3.1 Dataset Overview

Text and image dataset is provided with 27 different product type codes and a total of 84,916 unique products. Child, Household, Entertainment, and Books are the four top-level divisions that group the whole product catalog. The text data is in French and consists of two fields: a title and a description with a median length of 35 words and a limit of 2068 words. All the products have a title field, but 29,800 has no description present. The product images are all squares with a size of 500 x 500 pixels with either white or black borders. There is an image associated with each product.

Preprocessing

For text data, we performed the initial experiments with product title and descriptions as separate fields to train distinct models for text classification. Later, we combined both into one field by concatenating description and title together for each product and feed it to the fusion model.

In a baseline experiment this concatenated field was used as the input for all text based models. We used the text tokenizers corresponding to the pre-trained model.

- Text data was cleaned by stripping product title and description fields, whereas HTML tags were removed.
- Records with null description values were deleted.
- Images were resized to a uniform size of 224×224 and normalized per channel.
- We augment the training data for image classification by randomly rotating, flipping, and extracting random crops from the original images.

6.3.2 Implementation Details

The data preparation, data pre-processing and models are all implemented under Python 3.6, this choice was made due to the fact that Python is the most popular and major language for AI, ML and DL. Python has a lot of advantages like flexibility, readability, platform independence, and a great integration of important and useful libraries, we mention the following used libraries:

- **PyTorch:** is used for building-up and training DL models and handling massive datasets, primarily developed by Facebook's AI Research lab.
- **Huggingface Transformers Package:** is an immensely popular Python library providing pretrained models that are extraordinarily useful for a variety of natural language processing (NLP) tasks.

- **Pretrained Model Package:** provides access to pretrained ConvNets with a unique interface/API inspired by torchvision.
- **NumPy:** is used for creating and handling large and multi-dimensional arrays and matrices by a set of associated functions and operations.
- **Pandas:** is used for applying high-level data structures - like numerical tables and time series - operations and analysis.

All the trainings have been run in **Google Colaboratory**. Google Colaboratory or Google Colab as a short-term, is a tool provided by Google for writing and executing Python codes, it has many useful features like Graphics Processing Units (GPUs), easy sharing and environment for building and executing DL models.

6.3.3 Dataset Splits

We report and compare the performance of our uni-modal text and image baseline networks along with multi-modal joint learning models on different splits of the cleaned dataset which comprises 55,116 product instances. 90/10 training/validation split was the first attempt. However, using the same data for both training and testing of a machine learning model refrains us from detecting if a model is overfitted or not, as it will certainly behave accurate because of being tested on something it has already seen while training. For that reason, we split the dataset into train and test sets in a ratio of 80–20%, and the names are self-explanatory. Splitting the dataset only into train and test sets is not always enough, mainly when we are balancing between various hypotheses of a certain ML algorithm. So , we split the resulted train set into train and validation sets with split size 15%. After getting the trained fusion model, it was essential to test it out with the obtained test set.

6.3.4 Parameters Setting

Our implementation is based on pytorch framework [3]. We used $FlauBERT_{base}$ architecture with cased vocabulary which has 138M parameters and $CamemBERT_{base}$ architecture which contains 110M parameters. FlauBERT and CamemBERT models were trained with a batch size of 32 and a sequence length of 256 for 10 epochs. SE-ResNeXt-50 model was trained with a batch size of 32 for 20 epochs.

The multimodal representation learning module was fine-tuned using the pre-trained weights from the trained uni-modal baseline models. The uni-modal network layers were freezed and we trained our proposed multi-modal architecture using the feature fusion methods that will be described in the next sections, followed by the softmax layer to perform product classification. We used categorical cross entropy minimization objective as loss function, AdamW optimizer with the learning rate value of 2e-5 for the text models, and Adam optimizer with 1e-3 for the image model.

6.4 Baseline Model

6.4.1 Architecture

This architecture was build based on a paper work (Verma et al., 2020) [14]. The general approach is to project the uni-modal representations from image X'_{im} and text X_{t_1}, X_{t_2} into a joint representation space and learn a classifier in this joint space. i.e,

$$y = f(Z(X'_{im}, X_{t_1}, X_{t_2})) \tag{6.1}$$

where Z is a learned function that combines the text and image features, and f is a neural network classifier that maps the features from the joint feature space to a class label y.

For text data, initial experiments were performed by using concatenating description and title for each product. This field was fed to text-based uni-modal models (CamemBERT and FlauBERT). The Extracted token embeddings of CamemBERT and FlauBERT models are represented by $X_{t_1}, X_{t_2} \in R^{d_2}$. Where $d_2 = 768$ is model hidden size. These embeddings were projected to the output dimension equals to the number of classes using a linear layer. The image uni-modal model resulted in image embeddings X$_{im} \in R^{d_1}$. When training the fusion model, the uni-modal network layers were freezed, the concatinated field of product title and description was used as textual input to the final fusion module along with the reduced image representation. Thus, the input to the final fusion module is $X_{t_1}, X_{t_2}, X'_{im} \in R^{d_2}$. Figure 6.2 illustrates the structure of this experimental baseline model.

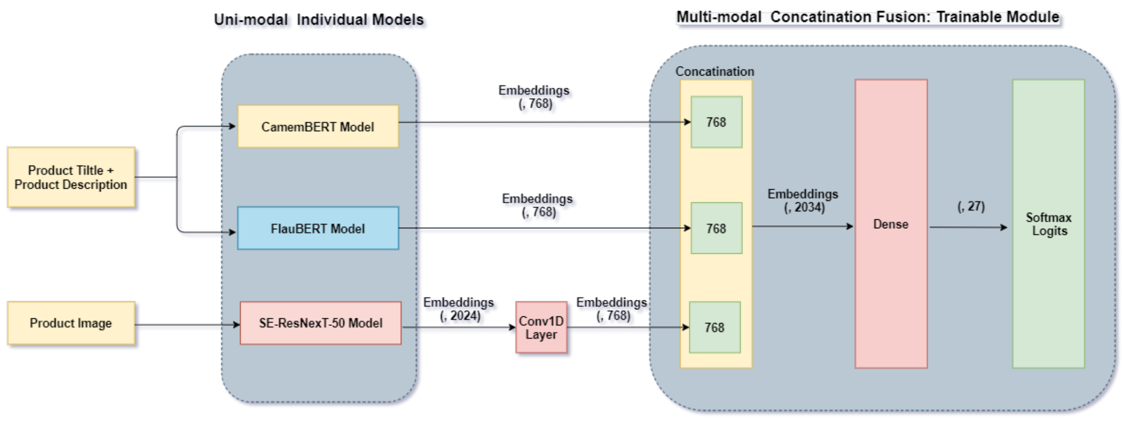


Figure 6.2: Multi-Modal Representation: Baseline Architecture

6.5 Results and Discussion

The aim of multi-modal models was to combine the predictions of the uni-modal models to get more accurate, confidant and robust predictions. Multiple models have been produced so far, these models vary in terms of overall performance. We present our findings in the following sections.

6.5.1 90/10, Train/Validation Split

In the first trial, a 90/10 train/validation split was used. Table 6.1 shows the performance of each uni-modal model, namely, CamemBERT-title, CamemBERT-description, FlauBERT-title, FlauBERT-description and SE-ResNeXt individually, as well as the hierarchical fusion models on the validation set using addition/concatenation fusion in the first fusion layer and different combination methods in the final fusion layer.

It is noticeable in the hierarchical architecture that the model that used average combination in both fusion layers has a good performance on the validation set.

6.5.2 Introducing More Layers

After running different experiments, the model with the average fusion method in both fusion layers showed the best performance in comparison with other models trained using a 90/10 dataset split. This model was made up of base uni-models

Model Type	**Model**	**Accuracy**	**F1 Score**
Validation Set			
Uni-Modal	CamemBERT$_{description}$	86.98%	84.83%
	CamemBERT$_{title}$	90.19%	87.38%
	FlauBERT$_{description}$	84.43%	84.43%
	FlauBERT$_{title}$	92.31%	90.32%
	SE-ResNeX-50	59.25%	54.30 %
Multi-Modal (Concatenation Base)	Concatenation Ensemble	92.80%	92.47%
	Addition Ensemble	92.75%	92.45%
	Average Ensemble	92.86%	92.47%
Multi-Modal (Average Base)	**Average Ensemble**	**93.20%**	**92.67%**

Table 6.1: Hierarchical Fusion Models Performance on Validation Set - 90/10 Split

Model Type	**Model**	**Accuracy**	**F1 Score**
Validation Set			
Uni-Modal	FlauBERT	92.38%	90.10%
	CamemBERT	92.09%	91.91%
	SE-ResNeX-50	59.25%	54.30 %
Multi-Modal	Addition Ensemble	92.15%	88.00%
	Concatenation Ensemble	93.16%	91.36%
	Average Ensemble	**93.46%**	**91.83%**

Table 6.2: Baseline Models Performance on Validation Set - 90/10 Split

merged partially in an intermediate fusion layer, where the average operation was used. The final average fusion layer was the following layer. Finally, a dense layer was used to project the fused logits with a dimension equals to the number of classes. This composed the input to the final layer, the softmax.

We started an incremental model building process aiming to further improve the performance of this model as follows:

- Adding a dropout layer after the final fusion layer.
- Stacking a dropout layer followed by FC layer and non-linearity before the final projection FC layer as shown in Figure 6.3

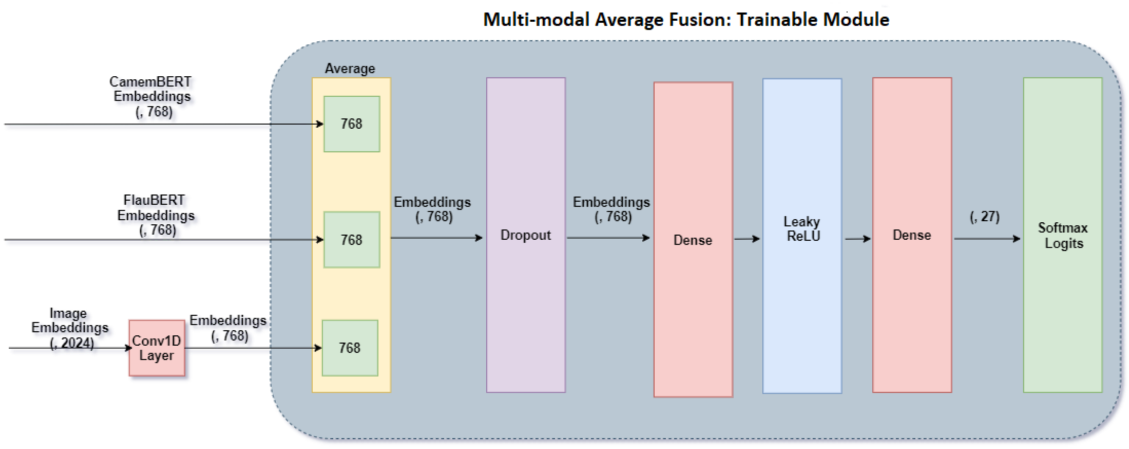


Figure 6.3: Multi-Modal Representation: Hierarchical Fusion with More Layers

Table 6.3 compares the performance of the basic proposed architecture and the modified architecture having stacked dropout out, FC and non-linearity after the final fusion layer.

Model Type	**Model**	**Accuracy**	**F1 Score**
Validation Set			
Multi-Modal	**Basic Architecture**	**93.20%**	**92.67%**
	With Dropout	92.49%	90.00%
	With More Layers	92.42%	91.11%

Table 6.3: Average Fusion performance with Different Architectures - 90/10 Split

Having looked at Table 6.3, a drop in Accuracy and F1 Score was introduced. Therefore, the modified model with more layers did not seem to improve the results, while the simplest architecture (Figure 6.1) showed better overall performance and reduced the training time, which gave room for launching more experiments.

It is noticeable in the simplest hierarchical architecture that the model that used average combination in both fusion layers has a good performance on the validation set. However, to obtain a robust model, further investigation is needed to verify that the model is not overfitted. Therefore, we split the dataset in a way that allows us to obtain a test set.

6.5.3 80/20, Train-Validation/Test Split

This spilt was performed for the sake of models testing. Table 6.4 compares the proposed hierarchical model variations in terms of accuracy and F1 Score on the given dataset split.

The best performance can be observed in the concatenation and average combination manner in the last fusion layer, where Average Ensemble outperforms the Concatenation Ensemble when evaluated on both validation and test sets. Detailed performance metrics of concatenation and average fusion models on test set are presented in Tables 6.8 and 6.9 respectively. Table 6.5 shows the performance of baseline models corresponding to the given dataset split. Best achieved models in terms of F1 Score are highlighted.

6.5.4 Comparison

In Tables 6.6 and 6.7 we present a comparison of the fused models architectures on 90/10 and 80/20 dataset splits.

It is clear that the proposed hierarchical model outperforms the baseline model by a reasonably wide margin in terms of F1-Score in all discussed experiments, while the baseline model does just marginally better in a few experiments in terms of Accuracy.
The best model has achieved an overall F1 Score of 92.67% while the worst has achieved an overall F1 Score of 88.00%. These models vary in terms of the individual classes' performance too, some models have a more balanced classification, while other models are biased for some classes.

Loss Manipulation

In both experiments which achieved the best performance on the 80-20 split, whose performance on test set is presented in Tables 6.8 and 6.9, it is noticeable that Class 1, Class 11, Class 14 and Class 16 have the lowest recall value, which means that these classes are more likely to be missed classified than other classes. To solve this problem, a potential solution could be proposed as: the loss of these classes could be given a higher weight.i.e. the classifier is punished more when it misses these classes.

Model Type	Model	Accuracy	F1 Score
Validation Set			
Uni-Modal	CamemBERT$_{description}$	86.40%	83.13%
	CamemBERT$_{title}$	89.64%	84.01%
	FlauBERT$_{description}$	85.81%	81.08%
	FlauBERT$_{title}$	88.98%	83.10%
	SE-ResNeX-50	57.20%	52.32%
Multi-Modal (Concatenation Base)	Addition Enseble	91.97%	89.39%
	Concatenation Ensemble	92.36%	90.03%
	Average Ensemble	**92.86%**	**92.46%**
Multi-Modal (Average Base)	Average Ensemble	92.25%	89.94%
Test Set			
Uni-Modal	CamemBERT$_{description}$	86.03%	84.31%
	CamemBERT$_{title}$	90.03%	86.06%
	FlauBERT$_{description}$	85.29%	83.27%
	FlauBERT$_{title}$	88.81%	84.24%
	SE-ResNeX-50	51.00%	41.50%
Multi-Modal (Concatenation Base)	Addition Ensemble	91.94%	87.29%
	Concatenation Ensemble	91.64%	90.39%
	Average Ensemble	**92.23%**	**90.45%**
Multi-Modal (Average Base)	Average Ensemble	92.03%	87.70%

Table 6.4: Hierarchical Fusion Models Performance on Validation and Test Sets-80/20 Split

Model Type	**Model**	**Accuracy**	**F1 Score**
Validation Set			
Uni-Modal	FlauBERT	92.10%	90.18%
	CamemBERT	92.28%	88.54%
	SE-ResNeX-50	57.20%	52.32%
Multi-Modal	Addition Ensemble	92.16%	88.01%
	Concatination Ensemble	92.17%	88.36%
	Average Ensemble	**92.09%**	**88.46%**
Test Set			
Uni-Modal	FlauBERT	92.21%	87.77%
	CamemBERT	92.07%	90.73%
	SE-ResNeX-50	51.00%	41.50%
Multi-Modal	Addition Ensemble	92.22%	90.38%
	Average Ensemble	92.28%	90.31%
	Concatenation Ensemble	**92.08%**	**90.79%**

Table 6.5: Baseline Models Performance on Validation and Test Sets - 80/20 Split

Fusion Operation	**Model**	**Accuracy**	**F1 Score**
Validation Set			
Addition	Baseline	92.15%	88.00%
	Hierarchical	92.75%	92.45%
Concatenation	Baseline	93.16%	91.36%
	Hierarchical	92.80%	92.47%
Average	Baseline	92.38%	90.10%
	Hierarchical	92.86%	92.47%
Average	**Hierarchical (average base)**	**93.20%**	**92.67%**

Table 6.6: Comparison of Baseline and Hierarchical Models Performance on Validation Set - 90/10 Split

Fusion Operation	**Model**	**Accuracy**	**F1 Score**
Validation Set			
	Baseline	92.16%	88.01%
Addition	Hierarchical	91.97%	89.39%
	Baseline	92.17%	88.36%
Concatenation	Hierarchical	92.36%	90.03%
	Baseline	92.09%	88.46%
Average	**Hierarchical**	**92.86%**	**92.46%**
Average	Hierarchical (average base)	92.25%	89.94%
Test Set			
	Baseline	92.22%	90.38%
Addition	Hierarchical	91.94%	87.29%
	Baseline	92.22%	90.34%
Concatenation	Hierarchical	91.64%	90.39%
	Baseline	92.28%	90.31%
Average	**Hierarchical**	**91.23%**	**90.45%**
Average	Hierarchical (average base)	92.03%	87.70%

Table 6.7: Comparison of Baseline and Hierarchical Models Performance on Validation and test Set using 80/20 Split

Class Name	**Precision**	**Recall**	**F1 Score**
Class 1	0.84	0.71	0.77
Class 2	0.82	0.91	0.86
Class 3	0.91	0.97	0.94
Class 4	0.85	0.83	0.84
Class 5	0.97	0.99	0.98
Class 6	0.97	0.96	0.96
Class 7	0.92	0.90	0.91
Class 8	0.92	0.90	0.91
Class 9	0.80	0.83	0.82
Class 10	0.95	0.96	0.95
Class 11	0.69	0.88	0.77
Class 12	0.89	0.85	0.87
Class 13	0.99	0.99	0.99
Class 14	0.83	0.65	0.73
Class 15	0.99	0.99	0.99
Class 16	0.80	0.73	0.76
Class 17	0.93	1.00	0.97
Class 18	0.90	0.89	0.90
Class 19	0.96	0.91	0.93
Class 20	0.97	0.98	0.98
Class 21	0.87	0.90	0.88
Class 22	0.91	0.91	0.91
Class 23	0.97	0.97	0.97
Class 24	0.99	0.99	0.99
Class 25	0.97	0.94	0.95
Class 26	0.96	0.99	0.98
Class 27	1.00	0.99	1.00
Overall	0.91	0.91	0.93

Table 6.8: Performance Metrics of Hierarchical Concatenation Model on Test Set

Class Name	**Precision**	**Recall**	**F1 Score**
Class 1	0.87	0.68	0.76
Class 2	0.86	1.00	0.93
Class 3	0.91	0.94	0.93
Class 4	0.83	0.81	0.82
Class 5	0.98	0.99	0.99
Class 6	0.96	0.97	0.96
Class 7	0.88	0.86	0.87
Class 8	0.90	0.89	0.89
Class 9	0.76	0.79	0.77
Class 10	0.93	0.94	0.93
Class 11	0.69	0.73	0.71
Class 12	0.80	0.81	0.81
Class 13	0.98	1.00	0.99
Class 14	0.73	0.44	0.55
Class 15	1.00	0.99	0.99
Class 16	0.60	0.50	0.55
Class 17	0.95	0.98	0.96
Class 18	0.89	0.89	0.89
Class 19	0.88	0.87	0.87
Class 20	0.99	0.95	0.97
Class 21	0.86	0.89	0.88
Class 22	0.89	0.88	0.88
Class 23	0.92	0.95	0.93
Class 24	0.98	1.00	0.99
Class 25	0.94	0.92	0.93
Class 26	0.98	0.96	0.97
Class 27	0.93	0.96	0.95
Overall	92	0.88	0.92

Table 6.9: Performance Metrics of Hierarchical Average Model on Test Set

Chapter 7

Conclusion and Future Work

Through this research, multiple Deep Learning models were applied and investigated, with the goal of producing a robust and accurate classification, taking into consideration the data shortage and imbalance problems.

On the one hand, the baseline experiment was done using concatenated textual features and the best variation used average fusion operation and achieved an overall performance of 91.83% and 88.46% in terms of F1-Score on the validation set given 90/10 and 80/20 splits respectively. On the other hand, textual features were fed individually to the proposed hierarchical models. The best overall performance was recorded using the average fusion operation in both fusion layers, where it reached an F1-Score of 92.67% on the validation set given 90/10 dataset split. While when it came to 80/20 dataset split, the best achieved model was concatenation-based when considering the textual features, and average-based in the final fusion layer with F1-Score of 92.46%.

A careful analysis of the performance on individual classes showed that the dataset is imbalanced. Therefore, some techniques like loss manipulation and batch balancing could be helpful to further improve the obtained results. Moreover, gradient boosting multi-class classification model could be used to learn the weighted contribution of class probability outputs from each uni-modal model, whereas late fusion method could be the second level model on top of the trained uni-modal baselines and multi-modal fusion methods.

All experiments in this research were done on a dataset of 27 classes that contains a total of 55, 116 unique products. The research intended to solve a real-world problem in the domain of e-commerce classification, thus, many classes of products can be included in the dataset in order to make the classification process wider and more useful. The future plan aims to collect or generate more data in order to increase the existing dataset size, besides adding more classes. Finally, adapt the existing models to the extended dataset.

Acknowledgement

We would like to thank the Tempus Public Foundation for the financial support through the Stipendium Hungaricum Scholarship Programme and the staff of Computer Science for Autunomous Systems specialization and Data Science and Engineering Department, Faculty of Informatics, Eötvös Loránd University, for their efforts and assistance with this research.

Bibliography

[1] S. Shetty T. Leung R. Sukthankar A. Karpathy, G. Toderici and F. F. Li. Large-scale video classification with convolutional neural networks, 2014.

[2] J. H. McDermott W. T. Freeman A. Owens, J. Wu and A. Torralba. Ambient sound provides supervision for visual learning, 2016.

[3] Soumith Chintala Gregory Chanan Edward Yang Zachary DeVito Zeming Lin Alban Desmaison Luca Antiga Adam Paszke, Sam Gross and Adam Lerer. Automatic differentiation in pytorch., 2017.

[4] Time Salimans Alec Radford, Karthik Narasimhan and Ilya Sutskever. Improving language understanding with unsupervised learning. 2018.

[5] AndrewMDai and Quoc V Le. Semi-supervised sequence learning. 2015.

[6] Alexei Baevski, Sergey Edunov, Yinhan Liu, Luke Zettlemoyer, and Michael Auli. Cloze-driven pretraining of self-attention networks, 2019.

[7] Gürol Canbek, Seref Sagiroglu, Tugba Taskaya Temizel, and Nazife Baykal. Binary classification performance measures/metrics: A comprehensive visualized roadmap to gain new insights. pages 821–826, 10 2017.

[8] Z. Lei D. Yi and S. Z. Li. Shared representation learning for heterogenous face recognition, 2015.

[9] Christian Jutten Dana Lahat, Tülay Adalı. Multimodal data fusion: An overview of methods, challenges and prospects., 2015.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina N. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018.

[11] R. Dhanesh and T. Graham W. Deep multimodal learning: A survey on recent advances and trends, 2017.

[12] Changxing Ding and Dacheng Tao. Robust face recognition via multimodal deep face representation. *IEEE Transactions on Multimedia*, 17(11):2049–2058, Nov 2015.

[13] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2016.

[14] Vinodh Motupalli Ekansh Verma, Souradip Chakraborty. Deep multi-level boosted fusion learning framework for multi-modal product classification, 2020.

[15] D. Wu et al. Deep dynamic neural networks for multimodal gesture segmentation and recognition, 2013.

[16] Xavier Glorot and Y. Bengio. Understanding the difficulty of training deep feed-forward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010.

[17] Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. 2019.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks, July 2016.

[20] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2020.

[21] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification, 2018.

[22] Bao-Gang Hu and Wei-Ming Dong. A study on cost behaviors of binary classification measures in class-imbalanced problems, 2014.

[23] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-excitation networks, 2019.

[24] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[25] David Grangier Denis Yarats Jonas Gehring, Michael Auli and Yann N. Dauphin. Convolutional sequence to sequence learning. 2017.

[26] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aäron van den Oord, Alexander Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. 2016.

[27] Omer Levy Christopher D. Manning Kevin Clark, Urvashi Khandelwal. What does bert look at? an analysis of bert's attention. 2019.

[28] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[29] Edouard Grave Tal Linzen Kristina Gulordava, Piotr Bojanowski and Marco Baroni. Colorless green recurrent networks dream hierarchically. 2018.

[30] Oleksii Kuchaiev and Boris Ginsburg. Factorization tricks for lstm networks. 2017.

[31] L. I. Kuncheva. Combining pattern classifiers: Methods and algorithms, 2014.

[32] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. Race: Large-scale reading comprehension dataset from examinations, 2017.

[33] Guillaume Lample and Alexis Conneau. Crosslingual language model pretraining. 2019.

[34] Hang Le, Loïc Vial, Jibril Frej, Vincent Segonne, Maximin Coavoux, Benjamin Lecouteux, Alexandre Allauzen, Benoît Crabbé, Laurent Besacier, and Didier Schwab. Flaubert: Unsupervised language model pre-training for french, 2020.

[35] Pedro Javier Ortiz Su´are Yoann Dupont Laurent Romary Eric Villemonte de la Clergerie Djam´e Seddah2 Beno^ıt Sagot Louis Martin, Benjamin Muller. Camembert: a tasty french language model, 2020.

[36] Mohit Iyyer Matt Gardner Christopher Clark Kenton Lee Matthew Peters, Mark Neumann and Luke Zettlemoyer. Deep contextualizedword representations. 2018.

[37] G. Taylor N. Neverova, C. Wolf and F. Nebout. Moddrop: Adaptive multimodal gesture recognition, 2016.

[38] Sebastian Nagel. Cc-news. 2016.

[39] Yonatan Belinkov M. Peters Nelson F. Liu, Matt Gardner and Noah A. Smith. Linguistic knowledge and transferability of contextual representations. 2019.

[40] Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. Scaling neural machine translation, 2018.

[41] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018.

[42] Matthew E. Peters, Sebastian Ruder, and Noah A. Smith. To tune or not to tune? adapting pretrained representations to diverse tasks, 2019.

[43] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad, 2018.

[44] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *The MIT Press Journals*, 29(9), August 2017.

[45] Barry Haddow Rico Sennrich and Alexandra Birch. Neural machine translation of rare words with subword units. 2016.

[46] Andrei A. Rusu, Mel Vecerik, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets, 2016.

[47] E. Cambria S. Poria and A. Gelbukh. Deep convolutional neural network textual features and multiple kernel learning for utterance-level multimodal sentiment analysis, 2015.

[48] Dipanjan Sarkar. Guide to transfer learning with real-world applications in deep learning, 2018.

[49] Cohen I. Garg A. Huang Th.S Sebe, N. Machine learning in computer vision, 2005.

[50] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016.

[51] Paolo Frasconi Sepp Hochreiter, Yoshua Bengio and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. 2001.

[52] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[53] Marina Sokolova and Guy Lapalme. systematic analysis of performancemeasures for classification tasks. 2015.

[54] Tsegaye Misikir Tashu and Tomáš Horváth. Attention-based multi-modal emotion recognition from art. In *Pattern Recognition. ICPR International Workshops and Challenges: Virtual Event, January 10–15, 2021, Proceedings, Part III*, pages 604–612. Springer International Publishing, 2021.

[55] Trieu H. Trinh and Quoc V. Le. A simple method for commonsense reasoning, 2019.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2017.

[57] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems, 2020.

[58] Georg Wiese, Dirk Weissenborn, and Mariana Neves. Neural domain adaptation for biomedical question answering, 2017.

[59] Jonathan Hseu Xiaodan Song James Demmel Yang You, Jing Li and Cho-Jui Hsieh. Reducing bert pre-training time from 3 days to 76 minutes. 2019.

[60] Dong Yi, Zhen Lei, and Stan Z. Li. Shared representation learning for heterogenous face recognition. In *2015 11th IEEE International Conference and Workshops on Automatic Face and Gesture Recognition (FG)*, volume 1, pages 1–7, 2015.

[61] Naman Goyal Jingfei Du Mandar Joshi Danqi Chen Omer Levy Mike Lewis Luke Zettlemoyer Veselin Stoyanov Yinhan Liu, Myle Ott. Roberta: A robustly optimized bert pretraining approach. 2019.

[62] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019.

[63] Yukun Zhu, Ryan Kiros, Richard Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books, 2015.