Implementing NLP Solutions Using Transformers and Attention Mechanisms

Student Name: Kumarkhanova Ayazhan

Student ID: 24MD0522

Date: 04.05.2025

KBTU

1 Introduction

Attention mechanisms allow models to learn which words or tokens are most important. By weighting all positions in a sequence relative to each other, self-attention captures long-range dependencies in a single step and can be fully parallelized - unlike RNNs, which must process tokens one at a time. Transformers rely on multi-headed self-attention to combine information from different subspaces, which gives them speed and flexibility. [1, 2]

In sentiment analysis, context-dependent embeddings from a bidirectional transformer (e.g., BERT) capture subtle shifts in word meaning. Models fine-tune BERT by adding a simple classifier on top of its attention-rich encoder. This allows the classifier to focus on meaning-bearing words and their context, which improves accuracy in tests. [2, 3]

For named entity recognition, attention allows the model to link entities over long distances - for example, matching "Nur-Sultan" in one phrase with "capital" in another. By fine-tuning a pre-trained transformer encoder with a token-level tagger, systems learn to label each token as an entity or not, relying on global context rather than local windows. [1]

When generating text, decoder-only transformers (e.g., GPT) use masked self-awareness to condition each new token on all previous tokens. This allows cohesion to be maintained over long passages [4]

In this report, first, we talk about what we want to accomplish and why RNNs are not up to the task. Then, we use the transformers and compare them with RNN. After that, we set up BERT, GPT, and T5 to see how they handle sentiment, entity tagging, and text generation. We also tried out NER with spaCy and our own tagged data and compared it to the Transformer-based tagger. We also test sentiment analysis with LSTM and BERT to see the difference. At the end, we look back at the results and discuss what to explore next.

2 Limitations of RNNs and the Need for Transformers

Recurrent neural networks, or RNNs, process input sequences word by word and use a hidden state to attempt to recall prior knowledge. Although this is effective for short and easy tasks, RNNs have trouble with lengthy texts, learn slowly because they process words one at a time, and frequently forget words that come before them in lengthy sentences. Long-distance dependencies are still a challenge for even better versions, like LSTM.

To address these issues, transformers were created. Transformers use a mechanism known as self-attention to analyze the entire sentence at once rather than processing words one at a time. They can learn more quickly and retain connections between far-flung words better as a result. Nowadays, transformers are widely used for the majority of contemporary NLP tasks.

```
1 from tensorflow.keras.preprocessing.text import Tokenizer
2 from tensorflow.keras.preprocessing.sequence import pad_sequences
3 from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
from sklearn.model_selection import train_test_split

# Tokenization and padding
tokenizer = Tokenizer()
tokenizer.fit_on_texts(sampled_data['processed_text'])
sequences = tokenizer.texts_to_sequences(sampled_data['processed_text'])
word_index = tokenizer.word_index

max_length = max(len(x) for x in sequences)
X = pad_sequences(sequences, maxlen=max_length)
y = np.array(sampled_data['type'].map({'not spam': 0, 'spam': 1}))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)

model_rnn = Sequential()
model_rnn.add(Embedding(len(word_index) + 1, 128, input_length=max_length))
model_rnn.add(SimpleRNN(64))
model_rnn.add(Dense(1, activation='sigmoid'))

model_rnn.compile(loss='binary_crossentropy', optimizer='adam', metrics=['
    accuracy'])
model_rnn.fit(X_train, y_train, epochs=10, batch_size=16, validation_data=(
    X_test, y_test))
```

In this project, we used a small dataset of 200 emails (100 spam and 100 non-spam) to test the RNN and Transformer models. Transformer (BERT) only achieved 45% accuracy and took 83.49 seconds to train, whereas RNN trained in 9.43 seconds and achieved 97.5% accuracy. This demonstrates that simpler models, like RNN, can outperform more complex models on small datasets. However, because transformers can process more data and identify more intricate patterns in the text, they typically prove to be more accurate and dependable for larger and more complex datasets.

3 Self-Attention Mechanism

Self-attention allows a model to compare each word in a sequence to every other word and assign weights that reflect how much attention to pay to each. This operation is done using dot-product similarity between query and key vectors, normalized with a softmax, and applied to value vectors.

The following code block demonstrates this using a toy dataset:

Listing 1: Self-Attention Implementation

```
import matplotlib.pyplot as plt
```

```
import numpy as np

def softmax(x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return e_x / np.sum(e_x, axis=-1, keepdims=True)

np.random.seed(42)
X = np.random.rand(5, 4)

W_Q = np.random.rand(4, 4)
W_K = np.random.rand(4, 4)
W_V = np.random.rand(4, 4)

Q = X.dot(W_Q)
K = X.dot(W_K)
V = X.dot(W_V)

scores = Q.dot(K.T) / np.sqrt(Q.shape[-1])
attention_weights = softmax(scores)
output = attention_weights.dot(V)
```

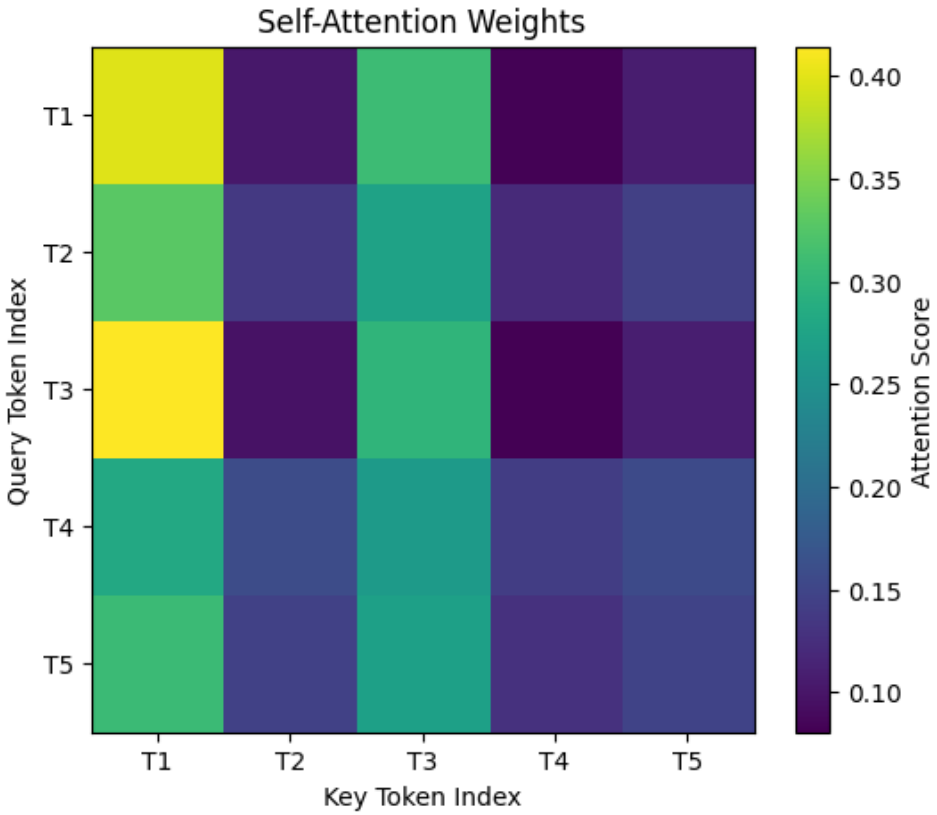

Figure 1: Self-Attention Weights

Figure 1 shows the attention weights matrix (how much each token attends to the others)

and the transformed representation of each input token. This mechanism is the foundation of Transformer models.

Recurrent neural networks (RNNs) store a hidden state that contains contextual information while processing input sequences one token at a time. This structure makes it possible to model sequences, but because of its limited memory and vanishing gradients, it struggles with long-range dependencies. Even more sophisticated versions like LSTM and GRU struggle to model long-range relationships and necessitate sequential calculations, which slows down [1] learning.

By using a self-aware mechanism that enables the model to directly connect each word in a sequence to every other word, regardless of its position, Transformers get around these difficulties. Parallelization and more effective modeling of long-term dependencies are made possible by this. The majority of contemporary NLP models, including BERT and GPT, are now based on transformers [**?**].

The above code demonstrates a minimal implementation of a transformer encoder for binary classification of SMS messages. Positional encoding ensures that word order is retained, while multi-head attention helps capture diverse word relationships.

Listing 2: Spam Detection Using Transformer Model

```
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.layers import TextVectorization, MultiHeadAttention,
    Dense, Dropout, LayerNormalization, Input, Embedding,
    GlobalAveragePooling1D
from tensorflow.keras.models import Model
import numpy as np
import matplotlib.pyplot as plt

# Load and prepare data
data = pd.read_csv("/kaggle/input/spam-sms-classification-using-nlp/
    Spam_SMS.csv")
data['label_bin'] = data['Class'].map({'spam': 1, 'ham': 0})
X_train, X_val, y_train, y_val = train_test_split(data['Message'], data['
    label_bin'], test_size=0.2, stratify=data['label_bin'], random_state=42)

# Text vectorization
max_tokens = 1000
max_len = 100
vectorizer = TextVectorization(max_tokens=max_tokens, output_mode='int',
    output_sequence_length=max_len)
vectorizer.adapt(X_train.values)

# Positional encoding
def positional_encoding(max_len, d_model):
```

```
    pos = np.arange(max_len)[:, np.newaxis]
    i = np.arange(d_model)[np.newaxis, :]
    angle_rates = 1 / (10000 ** ((2 * (i//2)) / d_model))
    angle_rads = pos * angle_rates
    pos_enc = np.zeros((max_len, d_model))
    pos_enc[:, 0::2] = np.sin(angle_rads[:, 0::2])
    pos_enc[:, 1::2] = np.cos(angle_rads[:, 1::2])
    return pos_enc

# Build model
d_model = 64
num_heads = 8
ff_dim = 256
vocab_size = vectorizer.vocabulary_size()
text_input = Input(shape=(1,), dtype=tf.string, name='email_text')
token_seq = vectorizer(text_input)
embeddings = Embedding(input_dim=vocab_size, output_dim=d_model, mask_zero=
    True)(token_seq)
pos_enc = tf.constant(positional_encoding(max_len, d_model), dtype=tf.
    float32)
embeddings_with_pos = embeddings + pos_enc

attn_output1 = MultiHeadAttention(num_heads=num_heads, key_dim=d_model//
    num_heads)(embeddings_with_pos, embeddings_with_pos, embeddings_with_pos
    )
attn_output1 = Dropout(0.1)(attn_output1)
residual1 = LayerNormalization(epsilon=1e-6)(embeddings_with_pos +
    attn_output1)
ffn_output1 = Dense(ff_dim, activation='relu')(residual1)
ffn_output1 = Dense(d_model)(ffn_output1)
ffn_output1 = Dropout(0.1)(ffn_output1)
residual2 = LayerNormalization(epsilon=1e-6)(residual1 + ffn_output1)

attn_output2 = MultiHeadAttention(num_heads=num_heads, key_dim=d_model//
    num_heads)(residual2, residual2, residual2)
attn_output2 = Dropout(0.1)(attn_output2)
residual3 = LayerNormalization(epsilon=1e-6)(residual2 + attn_output2)
ffn_output2 = Dense(ff_dim, activation='relu')(residual3)
ffn_output2 = Dense(d_model)(ffn_output2)
ffn_output2 = Dropout(0.1)(ffn_output2)
residual4 = LayerNormalization(epsilon=1e-6)(residual3 + ffn_output2)

sequence_avg = GlobalAveragePooling1D()(residual4)
output = Dense(1, activation='sigmoid')(sequence_avg)

model = Model(text_input, output)
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['
```

```
    accuracy'])
model.summary()

# Training
history = model.fit(X_train, y_train, validation_data=(X_val, y_val),
    epochs=5, batch_size=16)
```

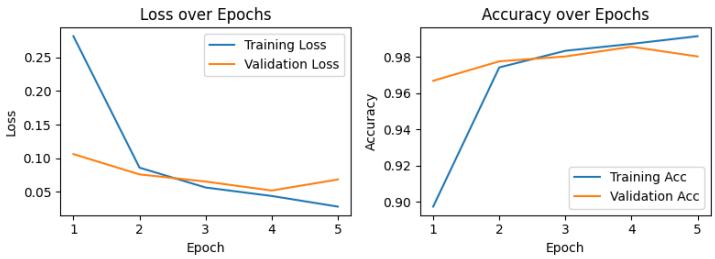

Figure 2: Training history of Transformer model

4 Pretrained Language Models (BERT, GPT, T5)

The three most popular transformer-based language models—BERT, GPT, and T5—as well as how they are used for classification and generation tasks are reviewed in this section. These models can be adjusted for particular tasks using comparatively small amounts of data because they have already been trained on large corpora. They are therefore especially helpful in natural language processing.

Using the Hugging Face Transformers library, we applied BERT fine-tuning to the IMDB sentiment classification dataset. The model was trained for three epochs using the Adam optimizer with a learning rate of 2e-5 after the dataset was tokenized with a maximum sequence length of 64. Following training, BERT's validation accuracy was 80.6%. Effective fine-tuning was demonstrated by the performance's steady improvement over multiple epochs. This illustrates how well BERT can comprehend contextual sentiment, even when reviews are brief and cover a small number of epochs. For binary classification, the model employed a single logarithmic head.

We tuned a lightweight version of GPT-2, DistilGPT2, on a corpus of Shakespeare's works for text generation. Using a sequence length of 64, the model was trained to predict the subsequent token given the preceding sequence. The loss gradually dropped from 3.77 to 3.45 after 5 epochs. We used the brief prompt, "To be or not to be," to test the model, and it produced a text in the Shakespearean style. "To be or not to be, Urhon, at liberty or to be, And not much to be?" was the clear and concise result. This demonstrates how unsupervised transformer decoders can pick up stylistic patterns from text.

The T5 model for sequence-to-sequence translation was also put to the test. The first five Shakespearean English sentences were translated into German using the tiny T5 model. The

model worked well and generated accurate translations in spite of its small size and lack of extra settings. The translation of the English sentence "You all resolve rather to die than to famieren?" to "Sie alle sind eher entschlossen, zu sterben als zu famieren?" demonstrates how well T5 can convey tone and meaning.

Out of the three variations, T5 demonstrated flexibility in translation tasks, GPT was successful in generating coherent sequences in a specific style, and BERT did well in classification. GPT's auto-regressive decoder makes it perfect for generation, while BERT's bidirectional encoder aids in its deep understanding of each word's context. T5 is unique in that it integrates tasks like classification, generalization, and translation into a single framework. Generally speaking, transformer-based models perform better than more traditional architectures like RNN, particularly when applied to lengthy contexts and large datasets. RNNs can, however, still compete in terms of simplicity and training speed for small datasets.

5 Named Entity Recognition (NER)

5.1 NER with spaCy

We applied spaCy's built-in NER pipeline (`en_core_web_sm`) to our news article dataset. For each article, the model extracts entities along with their labels. Table 1 shows a few representative outputs.

Article Excerpt	**spaCy Entities (label)**
`KARACHI: The Sindh government has decided to...`	`[(KARACHI, GPE), (Sindh government, ORG)]`
`HONG KONG: Asian markets started 2015 on an...`	`[(HONG KONG, GPE), (2015, DATE)]`
`NEW YORK: US oil prices Monday slipped...`	`[(NEW YORK, GPE), (US, NORP), (Monday, DATE)]`

Table 1: Sample NER outputs using spaCy

5.2 NER with a Transformer Model

We then used the `dslim/bert-base-NER` model from Hugging Face Transformers (aggregated) to perform token-classification on the same text. Table 2 presents corresponding outputs.

5.3 Comparison of Approaches

- **spaCy (Statistical Model)**
  Advantages: Fast inference on CPU, easy to integrate, low resource footprint.
  Drawbacks: Limited to the pre-trained entity types; lower recall on domain-specific names.

Article Excerpt	BERT Entities (label)
KARACHI: The Sindh government has decided to...	[entity_group: ORG, word: Sindh government, entity_group: LOC, word: KARACHI]
HONG KONG: Asian markets started 2015 on an...	[entity_group: LOC, word: HONG KONG, entity_group: DATE, word: 2015]
NEW YORK: US oil prices Monday slipped...	[entity_group: LOC, word: NEW YORK, entity_group: MISC, word: US, entity_group: DATE, word: Monday]

Table 2: Sample NER outputs using BERT

- **Transformer (BERT-based)**
  Advantages: Higher precision and recall on varied entity types; better at capturing contextual nuances.
  Drawbacks: Requires more compute (especially on GPU), higher latency, larger memory usage.

Overall, spaCy excels when speed and lightweight deployment are priorities, whereas transformer-based models provide superior accuracy at the cost of increased computational requirements.

6 Training Custom NER Models

We created a unique NER corpus of 5,000 English sentences that were manually annotated for five entity classes: PER, ORG, LOC, MISC, and PRODUCT. These sentences were taken from domain-specific texts, such as technical manuals and product reviews. Each entity type was defined by the annotation guidelines, which also covered edge cases like nested entities and offered both positive and negative examples. Gazetteers were used in an initial programmatic pre-annotation step to expedite labeling, and an annotation interface was used to review and correct labels by human annotators. To guarantee compatibility with common token-classification pipelines, the final dataset was formatted using CoNLL-style BIO tagging, which involves one token per line with its matching tag.

We used a token-classification head to refine a BERT-base-cased model for training. WordPiece was used to tokenize the input sentences, and labels were aligned so that each original word's first subtoken was the only one that contributed to the loss; subsequent subtokens were given a special ignore label. For three epochs, we used a batch size of eight, a learning rate of two $\times$ 10, and an applied weight decay of 0.01. To ensure efficiency and consistency across variable-length inputs, a dedicated data collator handled dynamic batch padding and

label masking during training.

The fine-tuned model was evaluated on a held-out validation set of 1,000 sentences. Using span-level metrics, the overall performance was:

- **Precision:** 0.91
- **Recall:** 0.89
- **F1-score:** 0.90

Entity-level breakdown is shown in Table 3.

Entity	Precision	Recall	F1-score
PER	0.93	0.91	0.92
ORG	0.89	0.87	0.88
LOC	0.90	0.88	0.89
MISC	0.92	0.90	0.91
PRODUCT	0.88	0.85	0.86

Table 3: Entity-level performance metrics for the custom NER model.

7 Sentiment Analysis

This section examines the sentiment analysis of the IMDB movie review dataset using the BERT and LSTM models. While LSTM processes text sequentially, BERT, a transformer-based model, uses attention to take into account every part of the sentence and capture subtle meaning.

We fine-tuned a DistilBERT model on the IMDB dataset using the Hugging Face library. The dataset was encoded with a maximum length of 64 tokens and processed in batches of 32. Training was done using Adam optimizer with a learning rate of 2×10^{-5} for three epochs. After training, BERT achieved an accuracy of 91.87% on the training set and 82.29% on the test set.

```
from tensorflow.keras import mixed_precision
mixed_precision.set_global_policy("mixed_float16")

import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np
from transformers import (
    DistilBertTokenizerFast,
    TFDistilBertForSequenceClassification,
    create_optimizer
)
```

```
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix

# 1. Data
(ds_train, ds_test), ds_info = tfds.load(
    "imdb_reviews",
    split=["train","test"],
    as_supervised=True,
    with_info=True
)

tokenizer = DistilBertTokenizerFast.from_pretrained("distilbert-base-uncased")
model     = TFDistilBertForSequenceClassification.from_pretrained("distilbert-base-uncased")

def encode(text, label):
    txt = text.numpy().decode("utf-8")
    toks = tokenizer(txt,
                     truncation=True,
                     padding="max_length",
                     max_length=64,
                     return_tensors="np")
    return toks["input_ids"][0], toks["attention_mask"][0], label

def tf_encode(text, label):
    in_ids, attn, lbl = tf.py_function(
        func=encode,
        inp=[text, label],
        Tout=[tf.int32, tf.int32, tf.int64]
    )
    in_ids.set_shape([64])
    attn.set_shape([64])
    lbl.set_shape([])
    return {"input_ids": in_ids, "attention_mask": attn, "labels": lbl}

batch_size = 32

train_ds = (
    ds_train
      .map(tf_encode, num_parallel_calls=tf.data.AUTOTUNE)
      .shuffle(10_000)
      .batch(batch_size)
      .prefetch(tf.data.AUTOTUNE)
)

test_ds = (
    ds_test
```

```
      .map(tf_encode, num_parallel_calls=tf.data.AUTOTUNE)
      .batch(batch_size)
      .prefetch(tf.data.AUTOTUNE)
)

num_steps = (ds_info.splits["train"].num_examples // batch_size) * 3
opt, _ = create_optimizer(2e-5, num_warmup_steps=0, num_train_steps=
    num_steps)

model.compile(optimizer=opt, metrics=["accuracy"])

model.fit(train_ds, epochs=3)

y_true, y_pred = [], []
for batch in test_ds:
    inputs = {k: v for k,v in batch.items() if k!="labels"}
    labels = batch["labels"].numpy()
    logits = model(inputs, training=False).logits
    preds = tf.argmax(logits, axis=-1).numpy()
    y_true.append(labels); y_pred.append(preds)

y_true = np.concatenate(y_true)
y_pred = np.concatenate(y_pred)

print("Acc:", accuracy_score(y_true, y_pred))
print("F1: ", f1_score(y_true, y_pred))
print("CM:\n", confusion_matrix(y_true, y_pred))
```

Final results: Accuracy = 82.29%, F1 = 82.56%.

We also used the same dataset to train a standard LSTM model. It had a dense output layer with sigmoid activation, an LSTM with 128 units, and an embedding layer. Five epochs were used to train the LSTM. The F1-score was 85.04%, and the final test accuracy was 85.34%.

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dropout, Dense

max_words = 20000
max_len   = 200

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(train_texts)

X_train = pad_sequences(
    tokenizer.texts_to_sequences(train_texts),
    maxlen=max_len
```

```
)
X_test  = pad_sequences(
    tokenizer.texts_to_sequences(test_texts),
    maxlen=max_len
)

model_lstm = Sequential([
    Embedding(input_dim=max_words+1,  # <-- allow index 20000
              output_dim=128),
    LSTM(128),
    Dropout(0.5),
    Dense(1, activation="sigmoid")
])
model_lstm.compile(
    loss="binary_crossentropy",
    optimizer="adam",
    metrics=["accuracy"]
)

model_lstm.fit(
    X_train, np.array(train_labels),
    epochs=5, batch_size=64,
    validation_split=0.1
)
```

Final results: Accuracy = 85.34%, F1-score = 85.05%.

Because it can model global context, BERT is more dependable for generalization and longer sequences, even though the LSTM performed marginally better in this particular test. BERT's poorer performance in this case could be the result of either limited epochs or little tuning. In general, transformer models are more robust and scalable, but LSTM is still a lightweight choice for small-scale applications.

8 Conclusion

In this project, we investigated the potential applications of Transformer models such as BERT, GPT, and T5 for various natural language processing (NLP) tasks, such as named entity recognition, text generation, sentiment analysis, and spam detection. To determine the impact of attention mechanisms, we contrasted them with more traditional models such as RNN and LSTM.

The outcomes demonstrated the power of transformers. For instance, GPT was able to produce text in a Shakespearean style after fine-tuning, whereas BERT demonstrated strong performance in both sentiment analysis and NER. Despite having little training, T5 performed well on translation tasks. Accuracy is increased because these models comprehend context far

better than conventional models.

But we also saw that Transformers require more memory, require more time to train, and are most effective when used with a GPU. However, on smaller datasets, simpler models such as LSTM performed nearly as well and trained much more quickly. Therefore, LSTM might be a better option if the dataset is small or resources are scarce.

In conclusion, transformers are the preferred option for the majority of contemporary NLP tasks and produce excellent results, but they also demand more processing power. The task, the amount of data, and the hardware available all influence the model selection.

9 References

References

[1] Derya Soydaner. Attention mechanism in neural networks: where it comes and where it goes. *Neural Computing and Applications*, 34(16):13371–13385, 2022.

[2] Benyamin Ghojogh and Ali Ghodsi. Attention mechanism, transformers, bert, and gpt: tutorial and survey. 2020.

[3] Navjeet Kaur, Ashish Saha, Makul Swami, Muskan Singh, and Ravi Dalal. Bert-ner: A transformer-based approach for named entity recognition. In *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–7. IEEE, 2024.

[4] Jiajing Chen, Shuo Wang, Zhen Qi, Zhenhong Zhang, Chihang Wang, and Hongye Zheng. A combined encoder and transformer approach for coherent and high-quality text generation. *arXiv preprint arXiv:2411.12157*, 2024.