Information Retrieval and Data Mining Coursework 2

Abstract

This project aims to evaluate a set of models for passage retrieval as part of an information retrieval pipeline. The primary objective is to rank relevant passages for a given query. The initial step involves implementing a BM25-based scoring system and evaluating it with Mean Average Precision (MAP) and Normalised Discounted Cumulative Gain (NDCG) on validation data. All subsequently built models are evaluated using these metrics, trained on train data.tsv, and assessed on validation data.tsv. First, a Logistic Regression (LR) model is introduced, where GloVe-based average embeddings represent queries and passages. Different learning rates are tested to optimise training, and the loss curves are analysed. After identifying the learning rate with the highest MAP and NDCG, that rate is selected for the final LR model. Subsequently, a LambdaMART (LM) model is implemented using the XGBoost framework. Embedding-based feature vectors are generated, and a randomised hyperparameter search identifies the best-performing settings and the top-performing model. Finally, two neural networks—a BiLSTM-based ranker and a CNN-based ranker—are designed and trained using token-level embeddings initialised with GloVe. Both models are evaluated on the validation set with MAP and NDCG, and the better network is chosen. After determining the best model in each class (LR, LM, NN), all three are applied to rank the top 100 passages from candidate passages top1000.tsv for each query in test queries.tsv. The outputs are provided as `LR.txt` for logistic regression, `LM.txt` for LambdaMART, and `NN.txt` for the neural network model. Due to limited computational resources, extensive hyperparameter tuning and deeper neural network experiments were not performed.

1 Introduction

Information Retrieval (IR) systems depend on retrieving relevant passages for user queries. This coursework focuses on designing and evaluating several passage retrieval approaches, beginning with a BM25 baseline. We then enhance this baseline using machine-learning ranking methods, including a Logistic Regression model with GloVe embeddings, a LambdaMART model employing tree boosting, and two neural networks (BiLSTM and CNN) for re-ranking passages. We utilise Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (NDCG) as performance metrics for model evaluation. Following hyperparameter tuning and feature engineering, we compare the models to identify the best in each category, which are subsequently used for final rankings on the test set. The report is organised as follows: Section 2 reviews the BM25 baseline and evaluation metrics. Section 3 details the logistic regression approach, including strategies for negative downsampling, GloVe embeddings, training, and results. Section 4 focuses on the LambdaMART model, feature engineering, hyperparameter tuning, and performance outcomes. Section 5 examines the BiLSTM and CNN architectures for re-ranking, data preparation and results. Section 6 concludes with an overview of limitations and suggestions for future research.

2 Retrieval Quality Evaluation

Multiple metrics can measure how effectively our system ranks relevant passages among all candidates. This coursework focuses on *Average Precision (AP)* and *Normalised Discounted Cumulative Gain (NDCG)*.

2.1 Average Precision (AP)

Average Precision (AP) measures retrieval quality by examining how well relevant passages are distributed throughout the ranked list, by approximating the area under the precision-recall curve. For a single query q, let $r(k) = 1$ if the passage at rank k is relevant and $r(k) = 0$ otherwise, and let $P(k)$ be the precision at rank k. The formula for AP is:

$$\mathrm{AP}(q) \;=\; \frac{1}{R}\sum_{k=1}^{K} P(k)\, r(k),$$

where K is the total number of retrieved passages (or the maximum rank examined), and R is the total number of relevant passages for q. The *Mean Average Precision (MAP)* is obtained by averaging $\mathrm{AP}(q)$ over all queries.

Implementation Summary (AP). All model predictions are first grouped by their query ID (`qid`), forming a dictionary of passage IDs (`pid`) mapped to predicted scores. The passages are then sorted in *descending* order for each query based on the predicted score. We check each ranked passage against the ground truth to see if it is truly relevant ($r(k) = 1$) and record the fraction $\frac{\text{num\_relevant\_so\_far}}{k}$ whenever a relevant passage is encountered at rank k. Summing these fractions for all relevant passages and dividing by R yields $\mathrm{AP}(q)$. We average $\mathrm{AP}(q)$ over all queries to obtain MAP.

2.2 Normalised Discounted Cumulative Gain (NDCG)

Normalised Discounted Cumulative Gain (NDCG) emphasises the rank positions of relevant items, penalising relevant passages placed lower in the list. While NDCG can handle multi-level relevance, we use binary relevance in this coursework. The *Discounted Cumulative Gain* (DCG) for rank k is

$$\mathrm{DCG}(k) \;=\; \sum_{i=1}^{k} \frac{r(i)}{\log_2(i+1)},$$

where $r(i)$ is the relevance at rank i. To normalise and bound scores between 0 and 1, we compute the *Ideal DCG* (IDCG) by ranking all relevant passages at the top and re-evaluating DCG. Hence,

$$\mathrm{NDCG}(k) \;=\; \frac{\mathrm{DCG}(k)}{\mathrm{IDCG}(k)}.$$

Like AP, we group passages by `qid` and sort them in descending order of predicted score. We compute DCG by iterating over the top k passages in this sorted list, adding $\frac{1}{\log_2(k+1)}$ for each relevant passage at rank k. Using the same DCG formula, the IDCG is computed hypothetically placing all relevant passages at the top. We then calculate $\frac{\mathrm{DCG}}{\mathrm{IDCG}}$ for each query, defaulting to 0 if there

are no relevant passages. Finally, the mean NDCG is computed by averaging NDCG across all queries. Additionally, the implementation explicitly handles cases where IDCG = 0 to avoid division the query's NDCG is set to 0 by zero errors in such scenarios.

2.3 BM25 Performance

The same BM25 method from the previous coursework is used here, providing a benchmark for subsequent models. It should be mentioned Table 1 shows the MAP and NDCG of BM25 on the `validation_data.tsv`.

Table 1: Performance of BM25

Model	MAP	NDCG
BM25	0.239	0.382

3 Logistic Regression (LR)

This section explains how we implemented a Logistic Regression model to predict the relevance of each passage given a query. The approach treats passage relevance as a binary classification problem, where each *(query, passage)* pair receives a feature vector, and the model estimates $P(\text{relevant} \mid \text{query}, \text{passage})$.

3.1 Negative Downsampling

Due to computational limitations and the highly imbalanced nature of the dataset, we *downsample* the negative class. Specifically, we keep all positive samples (relevant passages) but limit the negatives per query to 10. This approach reduces bias toward the negative class and avoids significant class imbalance. Consequently, each query has a more manageable ratio of positive to negative examples.

3.2 Feature Representation: GloVe Embeddings

All text is converted into numerical vectors using GloVe (Global Vectors) embeddings. First, we tokenise and clean each query and passage (e.g., removing stopwords, optionally lemmatising). Next, each token is mapped to its 100-dimensional GloVe embedding from the pre-trained "glove.6B.100d" model (originally trained on Wikipedia). We manually loaded GloVe 100d embeddings by reading the file line by line, parsing each word and its corresponding embedding vector.

After obtaining the token embeddings, we *average* them per query (and similarly per passage) to get a single 100-dimensional embedding. We then concatenate the query and passage embeddings and append an intercept term of 1, resulting in a 201-dimensional feature vector.

Handling Out-of-Vocabulary Tokens. When a token does not exist in the GloVe vocabulary, we can adopt one of two strategies:

(1) Drop the token if fallback is not enabled (i.e., it contributes no information).
(2) If fallback *is* enabled, assign a random 100D vector with values sampled in the range $[-1, 1]$.

These methods ensure every *(query, passage)* pair has a well-defined feature vector, preventing empty embeddings in case all tokens are out-of-vocabulary. For this coursework, fallback was enabled, so tokens missing in the GloVe embeddings were consistently represented by random vectors rather than dropped.

3.3 Logistic Regression Implementation

Because relevance is either 0 or 1, we treat passage ranking as a binary classification problem. For each feature vector $\mathbf{x}$, logistic regression estimates:

$$\hat{y} \;=\; \sigma\big(\mathbf{w}^\top \mathbf{x} + b\big) \;=\; \frac{1}{1 + e^{-\left(\mathbf{w}^\top \mathbf{x} + b\right)}},$$

where $\mathbf{w}$ is the weight vector and b is a bias term. We define the binary cross-entropy loss function:

$$L(\mathbf{w}, b) \;=\; -\frac{1}{m} \sum_{i=1}^{m} \Big[y^{(i)} \ln\big(\hat{y}^{(i)}\big) + \big(1 - y^{(i)}\big) \ln\big(1 - \hat{y}^{(i)}\big) \Big],$$

The parameters are updated via gradient descent as follows:

$$w \leftarrow w - \alpha \frac{\partial L(w, b)}{\partial w}$$
$$b \leftarrow b - \alpha \frac{\partial L(w, b)}{\partial b}$$

and optimise $\mathbf{w}$ and b via gradient descent with α as the learning rate. To accelerate training, we employ *mini-batches* of size 5000 samples, updating the model in batches each epoch until all training samples are seen.

We tested multiple learning rates $\alpha \in \{0.001,\ 0.005,\ 0.01,\ 0.05,\ 0.1\}$ to identify which provides the best performance. Table 2 presents the MAP and NDCG for each learning rate, indicating that $\alpha = 0.01$ (MAP = 0.0167, NDCG = 0.1389) had the best performance. Figure 1 shows that higher rates converge quickly but exhibit small oscillations.

Table 2: MAP and NDCG for different LR learning rates

Learning Rate	MAP	NDCG
0.001	0.0145	0.1362
0.005	0.0166	0.1387
0.01	0.0167	0.1389
0.05	0.0164	0.1387
0.1	0.0166	0.1388

Table 3: Final logistic regression performance

Model	MAP	NDCG
LR	0.0167	0.1391

3.4 Performance of Logistic Regression

Table 3 shows the final performance of the LR model on the validation set. After training on the entire downsampled dataset, we use this LR model with a learning rate of 0.01 to re-rank the 1000 candidate passages file for each test query. The resulting top 100 passages (along with their ranks and scores) are saved to `LR.txt`.

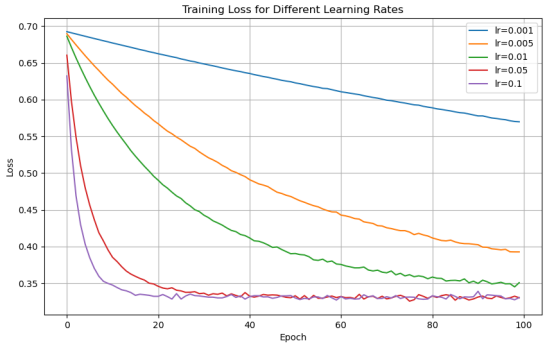

Figure 1: Training loss vs. epochs for different logistic regression learning rates. Higher rates converge quickly but may oscillate slightly. A learning rate of 0.01 provided stable convergence and good validation performance.

4 LambdaMART (LM)

4.1 LambdaMART Introduction

LambdaMART is a ranking model that evaluates the degree of relevance between pairs of samples in the training set, producing a relevance score for each (query, passage) pair. This method is suitable for re-ranking tasks because it optimises the order in which candidate passages appear. In this project, we use the XGBoost gradient boosting library with a ranking objective to re-rank passages based on learned features.

4.2 Input Processing and Feature Representation

The first step is to *downsample* the training data to address limited computational resources and the large number of negative passages. As in the previous task, we retain all positive samples and keep only a limited number of negatives (up to 10 per query) due to computational constraints and class imbalance. After this downsampling, each (query, passage) pair is processed to form a feature vector.

We reuse the GloVe embedding strategy from Task 2 to construct the feature vectors: we tokenise the query and passage text, convert each token into its 100-dimensional GloVe representation, and compute the *average* embedding for both query and passage. These two 100-dimensional vectors, plus a bias term of 1, yield a 201-dimensional feature vector (1 + 100 + 100). This embedding-based approach captures semantic information from queries and passages. This results in a 201-dimensional vector if a bias term is only concatenated, as well as the query and passage embedding. This is referred to as the "simple" feature approach. However, in the "enhanced" approach, BM25 and four overlap-based features are incorporated as described below.

4.2.1 Enhanced Feature Construction. In addition to the embedding-based vector (1 + 100 + 100 = 201D), (1) a BM25 score, (2) the query length, (3) the passage length, (4) the count of common tokens, and (5) an overlap ratio between common tokens and query length are added. These signals capture basic lexical alignment and frequency cues that can complement the distributional semantics from embeddings. Consequently, the final feature vector grows to 206 dimensions (201D + 1 BM25 dimension + 4 overlap-based dimensions). Thus, the final vector is 206D for the enhanced approach.

4.3 Setting Up XGBoost for Ranking

To train LambdaMART, we group training samples by query ID, collecting (query, passage) pairs from the downsampled dataset to build:

- A feature matrix X
- A label vector y (binary: 1 = relevant, 0 = irrelevant)
- A list of group sizes (indicating how many pairs belong to each query)

XGBoost ranker is instantiate with `objective="rank:ndcg"`, `tree_method="auto"`,`random_state=42`, and perform two separate hyperparameter searches. One search only uses the embedding-based feature vector (*simple*), while the other uses additional overlap-based signals (*enhanced*) feature vector that includes extra features. Both searches loop over all parameter combinations specified in Table 4. For each combination, the model is trained on the downsampled data and evaluated on MAP and NDCG on the validation set, ultimately selecting the model that achieves the highest MAP and NDCG.

Table 4: LambdaMART Hyperparameter Grid Search

Parameter	Grid Values	Optimal Value
learning_rate	{0.1, 1}	0.1
max_depth	{3, 5, 7}	5
n_estimators	{100, 200}	200

The resulting model is evaluated on the entire validation set using MAP and NDCG, recording the best-performing hyperparameter configurations. Table 5 compares the performance of two models: one using *simple* embedding features ("LM (Simple)"), and one with extra features ("LM (Enhanced)").

Table 5: Performance of LambdaMART Model

Model	mAP	NDCG
LM (Simple)	0.0255	0.1517
LM (Enhanced)	0.2529	0.3919

The model with extra features (Enhanced) outperforms the simple embedding-based model, so it was selected for the final submission (`LM.txt`). The hyperparameter search revealed that including the extra features greatly improves ranking quality: the best model without extra features achieved MAP = 0.0255 and NDCG = 0.1517, whereas including the extra features significantly boosted performance to MAP = 0.2529 and NDCG = 0.3919. Thus, combining extra features is beneficial. In conclusion, the enhanced feature model with hyperparameters learning rate: 0.1, max depth: 5, n estimators: 100, was chosen as the final LambdaMART model. This

model was then used to re-rank the passages from candidate passages top1000.tsv for each query in test queries.tsv, storing results in `LM.txt`.

5 Neural Network

In this final part, neural network-based models are designed and trained for reranking passages and unlike pointwise methods or tree-based ensembles, neural networks, in general, can learn to capture richer, sequential dependencies in the text potential, improving how queries and passages are matched. All input processing, including tokenisation and embedding assignment, follows the methods outlined in previous tasks, but here, these are applied to deeper architectures designed for NLP tasks.

A BiLSTM and a CNN model are designed and trained for this task. Bidirectional LSTM is chosen because it can capture the sequential relationships in the text, considering that each token's meaning can depend on surrounding tokens. LSTM units help address the vanishing gradient issues that often plague vanilla RNNs, and the bi-directionality allows the model to look forward and backwards within a token sequence. At the same time, convolutional neural networks can efficiently detect n-gram-like features in text, often converging faster and requiring fewer parameters than RNNs. By sliding convolutional filters across the combined query-passage sequence, the network can learn local matching word overlaps, partial phrase alignments, etc, that are useful for rankings. At the same time, both BiLSTM and CNN architectures handle variable-length text by summarising relevant patterns in the sequence. Unlike simpler models, for example, average embeddings alone, these networks can potentially learn more complex interactions between query and passage words. Hence, each architecture is well suited for the re-ranking task, where precise semantic matches can influence the most relevant passage. Ultimately, these two architectures were selected explicitly due to their complementary strengths—BiLSTM's ability to model sequential dependencies and CNN's efficiency in capturing local semantic patterns—thus providing a broad basis for comparative analysis.

5.1 Input Processing and Representation

As mentioned, the downsampling and keeping all the positive samples while limiting negatives per query to mitigate data imbalance and reduce computational cost strategy is also kept in place for this task. As part of data preparation, each query and passage pair is tokenised by removing stop words and applying stemming, then combined into a single sequence separated by a special [SEP] token. For example, a query's tokens first appear, followed by [SEP], and then the passage's tokens appear. To feed these to a neural model, a vocabulary is made first from all the tokens appearing in the training set. Any token below a minimum frequency threshold (in this case, 1) is replaced with `<UNK>` token to reduce sparse noise. Then, each token is converted to indices, and each word is mapped to a unique integer ID in the vocabulary dictionary. To ensure a uniform input is fed into the neural network, padding and truncation are done to the inputs for a fixed length, in this case, 200 tokens. This ensures a uniform input shape is fed to the neural network. Tokens beyond 200 are discarded, and shorter ones are padded with `<PAD>`. This approach allows flexible handling of queries and passages of varying lengths, helping with memory usage. By assembling all tokens for a given query and passage in one sequence, the neural network can learn how words in the query might interact with words in the passage.

5.2 Model Architectures

5.2.1 RNN with LSTM. A Recurrent Neural Network (RNN) processes sequences of tokens incrementally, maintaining a hidden state to capture temporal dependencies. However, standard RNNs struggle with long-range dependencies due to vanishing gradients. To address this, Long-Short-Term Memory (LSTM) networks are utilised. These networks feature gating mechanisms: input gate, forget gate, and output gate, enabling selective retention and removal of information from the internal cell state. This enhances the preservation of critical information and mitigates the short-term memory limitations inherent in conventional RNNs. The bi-directional LSTM (BiLSTM) architecture enhances context capture by processing input in both forward and backward directions, thus combining the hidden states to gain context from preceding and succeeding words. This approach proves advantageous for tasks such as passage reranking, where subtle contextual cues in both directions may offer valuable insights. In the final stage, the aggregated hidden states are processed through fully connected layers, complemented by dropout and ReLU activations to mitigate overfitting and capture non-linear relationships. The network outputs a sigmoid probability y, indicating the relevance of a passage to a given query. In summary, the rationale for employing RNNs with LSTM for this task includes: 1. Sequential Data: The inherent word order dependencies of queries and passages align well with the capabilities of LSTMs. 2. Long-term Context: LSTM gating mechanisms effectively retain important information from earlier tokens, which standard RNNs may overlook. 3. Bidirectionality: Incorporating contextual information from both directions enhances the alignment and nuance recognition between query and passage pairs.

5.2.2 Convolutional Neural Network (CNN). While RNNs process tokens sequentially, a 1D CNN extracts local features using sliding filters. In this coursework, each token is mapped to a 100D GloVE vector, resulting in a (batch_size × sequence_length × embedding_dim) tensor. A 1D convolution (kernel size = 3) slides across the embedding dimension for each sequence position, capturing n-gram patterns. After applying a ReLU non-linearity, adaptive max-pooling aggregates the highest activations from each filter, compressing the sequence length into a single feature per channel. This makes the representation robust to small positional shifts and variations in passage length. Finally, dropout and dense layers lead to a linear output with a sigmoid activation, providing probability relevance. CNNs are favoured for their faster training times compared to RNNs, as convolutions and pooling are easily parallelised, and they effectively capture short phrase matches without the sequential overhead of LSTMs, while max pooling ensures the model prioritises pattern presence over specific locations in the text.

5.3 Pre-Trained GloVe Embeddings

Before training, each model's `nn.Embedding` weight matrix is initialised using GloVe vectors loaded from glove.6B.100d.txt with the same method as explained in section 2.2. This step gives our models

a head start on learning semantically useful representations rather than relying on random initialisation alone.

5.4 Training Setup

PyTorch was used to train both models and binary cross-entropy as the loss function and Adam as the optimiser. The learning rate is set to 10^{-2} with a batch size of 32, and the training is run for 5 epochs. Beyond 5 epochs, CPU runtimes became long, so we settled on 5 for this coursework. Each LSTM layer contained 64 hidden units, chosen for memory constraints, while the CNN used 64 filters with a kernel size of 3. Due to computational limitations, more extensive hyperparameter tuning was skipped, such as adjusting learning rates, network depths, or adding attention mechanisms. Each model's loss per epoch is tracked, as shown in Figure 2. The Bi-LSTM tends to capture sequential relationships more effectively, but training takes longer. While CNN is faster to train and often converges faster, it is less sensitive to long-range context unless deeper layers or additional features are added.

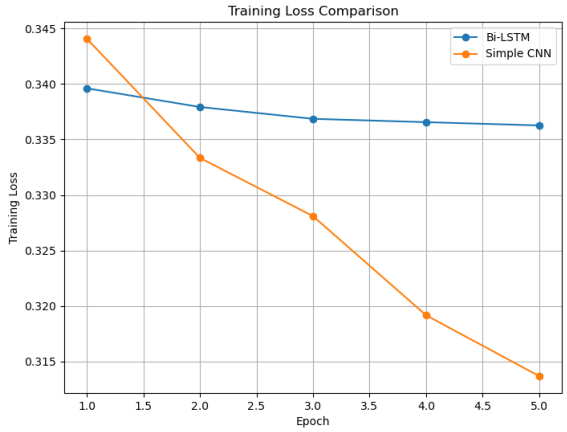

Figure 2: Training loss comparison between Bi-LSTM and Simple CNN over 5 epochs. The CNN converges faster.

5.5 Validation Results

The next step is to validate both data sets using the validation data set as used for previous tasks, converting each query and passage to the exact single sequence representation. A predicted relevance score via the network's forward pass is obtained for each sample. By grouping these scores by query and sorting in descending order, we compute MAP and NDCG, shown in the table below:

Table 6: Neural Model Validation Performance

Model	MAP	NDCG
Bi-LSTM	0.0091	0.1256
CNN	0.0104	0.1303

Although both neural approaches underperformed relative to the BM25 baseline, they demonstrate that deeper architectures can be integrated into an IR pipeline. We could explore extensive hyperparameter tuning (including deeper architectures, learning rate scheduling, and attention mechanisms) with more computational resources, potentially closing the gap with BM25.

5.6 Choosing the Best Neural Model and Final Test Ranking

Since the CNN model slightly outperformed the Bi-LSTM on the validation set, the CNN was explicitly chosen as the final neural network model. It was applied to rerank the candidate passages from `candidate_passages_top1000.tsv` for each query from `test_queries.tsv`, and the final reranked results were saved in `NN.txt`. All final rankings submitted (`LR.txt`, `LM.txt`, `NN.txt`) were generated explicitly from models selected based on their highest validation of MAP and NDCG scores.

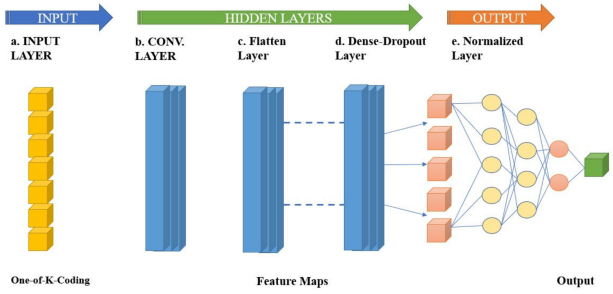

Figure 3: CNN architecture for text classification. Reproduced from [1].

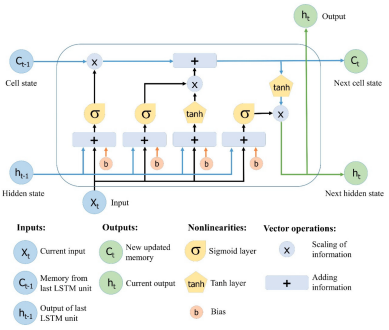

Figure 4: LSTM cell architecture. Reproduced from [2].

6 Limitations

Despite demonstrating a range of retrieval models—from classic BM25 to neural networks— this project faces several constraints that limit the overall performance and scope of the experiments. First, the dataset size and its imbalance required downsampling, which may discard potentially informative negative samples. Second, limited computational resources (e.g., absence of GPU acceleration)

constrained hyperparameter tuning and the depth of the neural architectures we could explore. Third, the feature representations relied heavily on average GloVe embeddings, which may lose finer semantic nuances. More advanced methods (e.g., transformer-based encoders) could capture richer context. Finally, the project's time constraints and the small training data size limited the ability to incorporate additional signals (like passage-specific lexical features or context-aware re-ranking strategies). Overcoming these limitations—through larger-scale data, stronger computational resources, and advanced embedding techniques could improve ranking effectiveness.

References

[1] Eman Aldakheel, Mohammed Zakariah, Ghada Gashgari, F A Almarshad, and Abdullah Alzahrani. 2023. A Deep Learning-Based Innovative Technique for Phishing Detection in Modern Security with Uniform Resource Locators. *Sensors* 23, 9 (2023), 4403. doi:10.3390/s23094403

[2] Sheng Yan. 2016. Understanding LSTM and Its Diagrams. https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714. Accessed: 2025-04-04.