NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Assignment for SC4002 / CE4045 / CZ4045

AY2023-2024

Group ID: 56

Group members:

Name	Matric No.	Contribution
Clarence Kway Wei Liang	U2020556A	Question 1
Foo Zhi Kai	U2022416G	Question 2
Teo Han Hua	U2022463B	Question 1
Xie Zijian	U2022580L	Question 2

Question 1.1 Word Embedding

Based on word2vec embeddings that we have downloaded, we made use of the function *most_similiar* from KeyedVectors in the *gensim l*ibrary and found out the most similar word to each of these words and its corresponding cosine similarity to be:

a) "student"
   - The most similar word to student is : students
   - Its cosine similarity is: 0.7294867038726807

b) "Apple"
   - The most similar word to Apple is: Apple_AAPL
   - Its cosine similarity is: 0.7456986308097839

c) "apple"
   - The most similar word to apple is: apples
   - Its cosine similarity is: 0.720359742641449

Question 1.2 Data

a) After preprocessing the dataset such that each of them contains a training file, a development file and a test file. We found out that the size (number of sentences) of the training development and test files for CoNLL2003 to be:

Number of sentences for train data is : 14041
Number of sentences for development data is : 3250
Number of sentences for test data is : 3453

We have decided to work with the IO tagging scheme and the set of all possible tag labels are:

1. I-PER
2. I-ORG
3. I-MISC
4. I-LOC
5. O

b) *To choose an example sentence from the training set of CoNLL2003 that has at least two named entities with more than one word, we created a function has_multi_word_entities that accepts two parameters: sentences and their corresponding labels.*

This function helps to check for sentences that have at least two named entities with more than one word for it.

```
def has_multi_word_entities(sentence, labels):
    current_entity = []
    multi_word_entities_count = 0

    for word, label in zip(sentence, labels):
        if label.startswith('I-'):
            current_entity.append(word)
        elif current_entity:
            if len(current_entity) > 1:
                multi_word_entities_count += 1
            current_entity = []

    if current_entity and len(current_entity) > 1:
        multi_word_entities_count += 1

    return multi_word_entities_count >= 2
```

The function will then be called in the *sentences_with_criteria* function which collates a list of sentences that fulfils the criteria.

```
def sentences_with_criteria(dataset):
    result = []

    for sentence, labels in dataset:
        if has_multi_word_entities(sentence, labels):
            result.append((sentence, labels))

    return result
```

Afterwards, we randomly picked a sentence to analyse (this is a randomly picked sentence, running the cell again will change the sentence):
“ " I don't normally do this but can you please sign, " he said thrusting an ornate white book in front of Americans Harrison Dillard (1948) , Lindy Remigino (1952) , Jim Hines (1968) , Trinidad 's Hasely Crawford (1976) and Britain 's Allan Wells (1980).”

To construct named entities based on the IO-labels assigned to each word, we developed an algorithm implemented in the e*xtract_entities_io* function. This function accepts two parameters: sentences and their corresponding labels.

```python
def extract_entities_io(words, labels):

    entities = []
    current_entity_words = []
    current_entity_type = None

    for word, label in zip(words, labels):
        if label.startswith("I-"):
            label_type = label.split("-")[1]

            # If we have a current entity and the label type has changed, or an unexpected label is encountered
            if current_entity_words and (not current_entity_type or current_entity_type != label_type):
                entities.append((current_entity_type, " ".join(current_entity_words)))
                current_entity_words = []

            # Update the current entity type and add the word to the current entity
            current_entity_type = label_type
            current_entity_words.append(word)

        else:  # For 'O' labels or any other unexpected label
            if current_entity_words:
                entities.append((current_entity_type, " ".join(current_entity_words)))
                current_entity_words = []
                current_entity_type = None

    # Handle any remaining entity at the end of the sequence
    if current_entity_words:
        entities.append((current_entity_type, " ".join(current_entity_words)))

    return entities
```

1. We start with the first word in the sequence.
2. If we encounter a word with an I- label, it's part of a named entity. Collect the words with consecutive I- labels of the same type to form multi-word entities.
3. If we encounter a word with an O label, or an I- label, or an I- label that is of a different type (e.g. 'I-MISC', 'I-PER') , it indicates the end of the current entity and possibly the start of a new one.
4. We continue this process until we've traversed the entire sequence.

As a result, this are the named entities in the sentence that we have picked:
1. Americans, TAG = MISC
2. Harrison Dillard, TAG = PER
3. Lindy Remigino, TAG = PER
4. Jim Hines, TAG = PER
5. Trinidad, TAG = LOC
6. Hasely Crawford, TAG = PER
7. Britain, TAG = LOC
8. Allan Wells, TAG = PER

Question 1.3 Model

a) Discuss how you deal with new words in the training set which are not found in the pretrained dictionary. Likewise, how do you deal with new words in the test set which are not found in either the pretrained dictionary or the training set? Show the corresponding code snippet.

For new words in the training set that are not found in the pretrained dictionary:

Embedded Initialization: If we are using pre-trained embeddings (like word2vec, GloVe, etc.), one common strategy is to initialise the embeddings of these Out-Of-Vocab (OOV) words with random values or with the average of all embeddings in the pretrained dictionary. This allows the model to adjust these embeddings to better fit the task during training.

```
# Create embedding matrix
embedding_dim = word_vectors.vector_size
embedding_matrix = np.zeros((len(word2idx) + 1, embedding_dim))  # +1 for padding token

for word, i in word2idx.items():
    try:
        embedding_matrix[i] = word_vectors[word]
    except KeyError:
        # Word not in pretrained embeddings; initialize randomly
        embedding_matrix[i] = np.random.normal(scale=0.6, size=(embedding_dim, ))

```

Special Token: Another approach is to map all OOV words to a special <UNK> (unknown) token. This way, the model learns a representation for unknown words. However, this can sometimes be suboptimal as it treats all OOV words as having the same meaning.

b) Describe what neural network you used to produce the final vector representation of each word and what are the mathematical functions used for the forward computation (i.e., from the pre-trained word vectors to the final label of each word). Give the detailed setting of the network including which parameters are being updated, what are their sizes, and what is the length of the final vector representation of each word to be fed to the softmax classifier.

A Recurrent Neural Network (RNN) was used to produce the final vector presentation of each word. The RNN architecture is as follows:

1. **Embedding Layer**: This is the first layer, and it is responsible for converting input word indices into dense vectors of fixed size, 'embedding_dim'. In this case, these embeddings are pre-trained, given by the 'embedding_matrix'.
2. **Bidirectional LSTM Layer**: After the word embeddings, the model uses a Bidirectional LSTM (Long Short-Term Memory) layer. LSTMs are a type of RNN that can capture long-range dependencies in a sequence (like a sentence). Being bidirectional means the LSTM processes the sequence from start to end from end to start, effectively giving each word a representation based on its past and future context.
3. **Dropout Layer**: This is a regularisation technique where randomly selected neurons are ignored during training, which helps in preventing overfitting.
4. **Dense Layer (Fully Connected Layer)**: The output of the LSTM is then passed through two fully connected layers (or dense layers). The first one ('self.fc') refines the LSTM output representation, and the second one ('self.fc2') produces the final tag predictions.

The mathematical functions for forward computations are:

1. **Embedding Layer**:
$$embedded = EmbeddingMatrix[text]$$
2. **Bidirectional LSTM**:
Input: embedded
Output: LSTM_out (concatenation of outputs from both the forward and backward LSTMs)
3. **Dropout**:
dropped = Dropout(LSTM_out)
4. **First Dense (Fully Connected) Layer**:
Input: dropped
Weights: W1 (size = 2 * 'hidden_dim' * 'hidden_dim')
Bias: b1
Computation: dense_out = dropped * W1 + b1
5. **Second Dense (Fully Connected) Layer**:
Input: dense_out
Weights: W2 (size = 'hidden_dim' * 'hidden_dim')
Bias: b2
Computation: tag_space = dense_out * W2 + b2

Report how many epochs you used for training, as well as the running time.

Number of epochs used for training: 28 epochs due to early stopping, 100 epochs without early stopping
Run time: 794.75 seconds ~= 13.24 minutes

Report the f1_score on the test set, as well as the f1_score on the development set for each epoch during training.

F1_score (test set) = 0.57
F1_score (development set for each epoch during training):

```
Epoch: 1/100, Train Loss: 0.2391, Validation Loss: 0.2434, Validation F1: 0.7645
Epoch: 2/100, Train Loss: 0.1695, Validation Loss: 0.2256, Validation F1: 0.7834
Epoch: 3/100, Train Loss: 0.1409, Validation Loss: 0.2134, Validation F1: 0.7863
Epoch: 4/100, Train Loss: 0.1221, Validation Loss: 0.1963, Validation F1: 0.8016
Epoch: 5/100, Train Loss: 0.1104, Validation Loss: 0.1908, Validation F1: 0.8017
Epoch: 6/100, Train Loss: 0.1043, Validation Loss: 0.1901, Validation F1: 0.8038
Epoch: 7/100, Train Loss: 0.0987, Validation Loss: 0.1828, Validation F1: 0.8075
Epoch 00007: reducing learning rate of group 0 to 5.0000e-04.
Epoch: 8/100, Train Loss: 0.0897, Validation Loss: 0.1825, Validation F1: 0.8112
Epoch: 9/100, Train Loss: 0.0867, Validation Loss: 0.1769, Validation F1: 0.8098
Epoch: 10/100, Train Loss: 0.0844, Validation Loss: 0.1759, Validation F1: 0.8109
Epoch: 11/100, Train Loss: 0.0822, Validation Loss: 0.1780, Validation F1: 0.8112
Epoch: 12/100, Train Loss: 0.0802, Validation Loss: 0.1748, Validation F1: 0.8115
Epoch: 13/100, Train Loss: 0.0792, Validation Loss: 0.1777, Validation F1: 0.8132
Epoch 00013: reducing learning rate of group 0 to 2.5000e-04.
Epoch: 14/100, Train Loss: 0.0759, Validation Loss: 0.1741, Validation F1: 0.8148
Epoch: 15/100, Train Loss: 0.0743, Validation Loss: 0.1738, Validation F1: 0.8146
Epoch: 16/100, Train Loss: 0.0736, Validation Loss: 0.1749, Validation F1: 0.8150
Epoch: 17/100, Train Loss: 0.0722, Validation Loss: 0.1754, Validation F1: 0.8154
Epoch: 18/100, Train Loss: 0.0719, Validation Loss: 0.1743, Validation F1: 0.8179
Epoch: 19/100, Train Loss: 0.0709, Validation Loss: 0.1752, Validation F1: 0.8157
Epoch 00019: reducing learning rate of group 0 to 1.2500e-04.
Epoch: 20/100, Train Loss: 0.0694, Validation Loss: 0.1758, Validation F1: 0.8149
Epoch: 21/100, Train Loss: 0.0687, Validation Loss: 0.1751, Validation F1: 0.8152
Epoch: 22/100, Train Loss: 0.0684, Validation Loss: 0.1757, Validation F1: 0.8168
Epoch: 23/100, Train Loss: 0.0681, Validation Loss: 0.1757, Validation F1: 0.8159
Epoch: 24/100, Train Loss: 0.0675, Validation Loss: 0.1755, Validation F1: 0.8170
Epoch: 25/100, Train Loss: 0.0671, Validation Loss: 0.1763, Validation F1: 0.8135
Epoch 00025: reducing learning rate of group 0 to 6.2500e-05.
Epoch: 26/100, Train Loss: 0.0668, Validation Loss: 0.1758, Validation F1: 0.8151
Epoch: 27/100, Train Loss: 0.0668, Validation Loss: 0.1759, Validation F1: 0.8144
Epoch: 28/100, Train Loss: 0.0661, Validation Loss: 0.1754, Validation F1: 0.8152
Early stopping due to no improvement in validation f1.
run time is : 794.75 seconds
```

Question 2

a) *Specify all the 5 classes you used after converting from the original label set to the new setting.*

In our current study, we have approached this by first selecting a random sample of four unique classes from the 'label-coarse' field of our training dataset. This selection process was performed using Python's numpy library, ensuring that the sample was drawn without replacement to maintain class uniqueness.

Once selected, we instituted a binary classification scheme for each sentence whereby labels not included in our chosen subset were reassigned to a catch-all 'OTHERS' category. This was achieved through the definition and application of the adjust_labels function, which takes a label as input and returns it unchanged if it is among the selected classes, or reassigns it to 'OTHERS' if not. This transformation was applied to our training, development, and test datasets.

As a result of this process, our classification task was defined by five distinct classes. The original labels were adjusted to include our randomly selected classes, which were 0, 5, 4, and 3, alongside the 'OTHERS' class.

```
array(['OTHERS', 0, 5, 4, 3], dtype=object)
```

To facilitate computational efficiency and clarity, we then mapped these five classes to a new set of contiguous integers. The mapping was as follows: 'OTHERS' was assigned 0, and the remaining classes were mapped to integers 1 through 4 in ascending order according to their label number. This mapping was applied to all instances in our datasets, thereby converting our labels into a format more suitable for machine learning algorithms.

The adjusted and mapped label sets, along with the corresponding text instances, are now ready for the next stages of our methodology, which will include feature extraction and the training of classification models. The table below illustrates the result of our label adjustment and mapping process on the first few rows of the training dataframe:

```
                                                text label-adjusted  \
0      What is Mikhail Gorbachev 's middle initial ?         OTHERS
1    How does the tail affect the flight of a kite ?              0
2  What were the first three cities to have a pop...              5
3   What is the movie Jonathan Livingstone Seagull ?         OTHERS
4              What is a fear of home surroundings ?         OTHERS

   label-mapped
0             0
1             1
2             2
3             0
4             0  )
```

b) Describe what aggregation methods you have tried and which is finally adopted (and why). Explain the detailed function of the aggregation method you used. If you have tested different aggregation methods, list their accuracy results to support your claim.

In our research, we assessed three aggregation methods: Mean, Max, and Last word, each applied across different neural network architectures to determine their impact on sentence classification tasks.

For the **Feedforward Neural Network (FFN)** and the **Convolutional Neural Network (CNN)**, we applied Mean and Max pooling. The Mean pooling method calculates the average of the representations across the sequence, providing an overall summary that smooths out the impact of any single vector. Max pooling, on the other hand, captures the most significant features by taking the maximum value across the sequence, which can be particularly useful in highlighting distinctive features that are crucial for classification.

For the **Recurrent Neural Network (RNN)**, which is inherently suited to sequence data due to its sequential processing nature, we applied Mean and Max pooling and also introduced Last word pooling. This method takes the final output vector in the sequence as the representation.

Here's how each method functions in the context of the code:

- **Mean pooling**:
    - For both the FFN/CNN and RNN, the mean pooling is implemented by averaging the output vectors across the first dimension (the sequence length), resulting in a single vector that summarizes the entire sequence.
- **Max pooling**:
    - Similar to Mean pooling, Max pooling takes the maximum value over the first dimension. This method is particularly useful for capturing the most salient features that are important for making a prediction.

- **Last word** (specific to RNN):
    - This method takes the output vector from the last time step of the RNN. In many sequence processing tasks, it is assumed that the last output contains information about the entire input sequence due to the recurrent connections.

Results:

Network Type	Aggregation Method	Test Accuracy
RNN	Mean	87.20%
RNN (Improved)	Mean	90.40%
RNN	Max	90.60%
RNN (Improved)	Max	93.80%
RNN	Last Word	91.80%
RNN (Improved)	Last Word	91.40%
FFN	Mean	85.40%
FFN (Leaky ReLU)	Mean	87.40%
FFN	Max	86.20%
FFN (Leaky ReLU)	Max	86.40%
CNN	Mean	89.40%
CNN	Max	92.40%

Given these empirical results, for the RNN, we finally adopted Max aggregation in conjunction with the improved RNN architecture due to its highest accuracy. In FFN, the standard Mean aggregation was preferred given its superior performance, and for CNN, Max aggregation was chosen for its top results.

c) *Describe what neural network you used to produce the final vector representation of each word and what are the mathematical functions used for the forward computation (i.e., from the pretrained word vectors to the final label of each word). Give the detailed setting of the network including which parameters are being updated, what are their sizes, and what is the length of the final vector representation of each word to be fed to the softmax classifier.*

The final vector representation of each word in our neural network models is produced using different architectures, including Feedforward Neural Networks (FFN), Recurrent Neural Networks (RNN), and Convolutional Neural Networks (CNN).

Feedforward Neural Network

We utilised two variants of a feedforward neural network (FFN) to generate the final vector representation for each word in a sentence before classification. The base model, QuestionClassifier, employs a three-layer architecture with ReLU activation functions, while the ImprovedQuestionClassifier integrates additional complexity, including more layers, leaky ReLU activation, and batch normalisation.

The QuestionClassifier model processes the input through a sequence of fully connected layers: the first layer (fc1) expands the input to a 128-dimensional space, followed by a dropout layer to mitigate overfitting, and then through a second fully connected layer (fc2) that further compresses the representation to a 64-dimensional space. A final fully connected layer (fc3) produces the logits for the number of classes (num_classes). The ImprovedQuestionClassifier differs by initially expanding the input to a 256-dimensional space and incorporating batch normalisation after each fully connected layer to stabilise learning. This variant also uses a leaky ReLU activation function, which allows for a small gradient when the unit is not active and helps maintain the flow of gradients during training.

The forward computation involves applying the ReLU or leaky ReLU activation functions following the linear transformations. The dropout is applied after the activation function.

In both models, an aggregation step follows the sequence of transformations to condense the entire sequence's information into a single vector, employing either mean or max pooling. The mean aggregation computes the average of the sequence of vectors, while the max aggregation takes the maximum value over each dimension of the sequence vectors.

The parameters updated during training include all the weights and biases of the fully connected layers (fc1, fc2, fc3, and so on) and the parameters of the batch

normalization layers in the improved model. Specifically, the sizes of the weight matrices correspond to the input and output dimensions of each layer, with the biases matching the output dimensions.

In the base FFN model (QuestionClassifier), the final vector representation length that is fed to the softmax classifier corresponds to the number of classes, as it represents the logits obtained from the last fully connected layer. In the case of the ImprovedQuestionClassifier, the final output before aggregation is also of length corresponding to the number of classes. After the aggregation step, the output vector is of dimension [batch_size, num_classes], which provides the basis for the probability distribution across classes after the softmax function is applied.

Recurrent Neural Network

In our exploration of neural network architectures for sentence classification, we have employed Recurrent Neural Network (RNN) models, specifically leveraging Long Short-Term Memory (LSTM) units for their ability to capture long-range dependencies in sequence data. The RNN models we developed consist of the RNNQuestionClassifier and an enhanced version, the ImprovedRNNQuestionClassifier.

The RNNQuestionClassifier utilises a standard LSTM layer to process the sequence of pre-trained word vectors of a given input_dim. The LSTM transforms these inputs into a sequence of hidden states of dimension hidden_dim, with the number of stacked LSTM layers specified by num_layers. After processing the sequence, the LSTM's output can be aggregated using different strategies—mean or max pooling across the time dimension, or taking the last hidden state as the representation of the entire sequence.

For classification, a fully connected layer (fc) transforms the aggregated LSTM output into logits corresponding to the num_classes. The softmax function is then applied to these logits to generate a probability distribution over the class labels for each word in the sequence.

The ImprovedRNNQuestionClassifier builds upon this architecture by incorporating bidirectional LSTM layers, which process the sequence in both forward and reverse directions, effectively doubling the hidden state dimensionality. This model also includes an additional dropout layer to prevent overfitting, especially in scenarios where multiple LSTM layers are stacked (num_layers > 1). The forward computation in the improved model is similar, with the addition of a fully connected layer (fc1) that transforms the bidirectional LSTM outputs before applying the ReLU activation

function, followed by dropout, and then a final fully connected layer (fc2) that computes the logits for classification.

Both models use parameter sets that include weights and biases for the LSTM layers and the fully connected layers. In the improved model, the weights and biases are doubled for the bidirectional LSTM. The parameters are updated during training using backpropagation through time (BPTT).

The final vector representation of each word, which is fed to the softmax classifier, is of length hidden_dim for the RNNQuestionClassifier and 2 * hidden_dim for the ImprovedRNNQuestionClassifier (due to the bidirectionality). The length of this vector is determined by the aggregation method used: if mean or max pooling is employed, the final vector represents the entire sequence in a fixed-length format; if the last hidden state is used, the vector represents the sequence as captured in the final LSTM unit.

Convolutional Neural Network

In our study, Convolutional Neural Networks (CNN) have been employed for sentence classification due to their proficiency in capturing local context and identifying salient features across different regions of the input data. The CNN models, CNNQuestionClassifier and MeanCNNQuestionClassifier, differ primarily in their pooling strategies but share a common structure that leverages convolutional layers to process word embeddings derived from pre-trained vectors.

The CNNQuestionClassifier begins by embedding input word indices into a continuous vector space, using an embedding layer with a specified vocab_size and embedding_dim. These embeddings are initialized with pre-trained vectors and are set to be non-trainable within the network to preserve their pre-learned semantic content. The embedded words are then passed through multiple 1D convolutional layers, each corresponding to different kernel sizes (filter_sizes). These layers serve to extract feature maps by convolving the embeddings with learnable filters of size [num_filters, embedding_dim, kernel_size].

The forward pass through the network is as follows:

1. **Embedding:** The input sequences are transformed into dense vectors using the pre-trained embeddings.
2. **Convolution:** The embedded vectors are convolved with multiple sets of filters, each of a different size. This operation is performed by the nn.Conv1d layers, capturing local features within different n-gram contexts.

3. **Activation (ReLU):** The convolution output is passed through a non-linear ReLU activation function to introduce non-linearity into the model.
4. **Pooling:** In CNNQuestionClassifier, max pooling is used to aggregate the feature maps, extracting the most prominent features. Conversely, MeanCNNQuestionClassifier uses average pooling to summarize the features by calculating the mean response across the feature map.
5. **Concatenation:** The pooled outputs from each convolutional layer, representing the most salient features detected by each filter, are concatenated into a single vector.
6. **Dropout:** A dropout layer with a rate of 0.5 is applied to the concatenated features to prevent overfitting.
7. **Fully Connected Layer:** The concatenated and regularized feature vector is then passed through a fully connected layer (fc) which transforms it to the size of num_classes, producing the logits for classification.
8. **Softmax:** Finally, a softmax function is applied to the logits to obtain a probability distribution over the class labels.

The CNNQuestionClassifier produces a final vector representation for each word that is the concatenated result of the pooled feature maps, with the length determined by the product of the number of filters and the number of filter sizes. The size of the fully connected layer's weight matrix is [len(filter_sizes) * num_filters, num_classes]. During training, the weights and biases of the convolutional and fully connected layers are updated.

d) Report how many epochs you used for training, as well as the running time

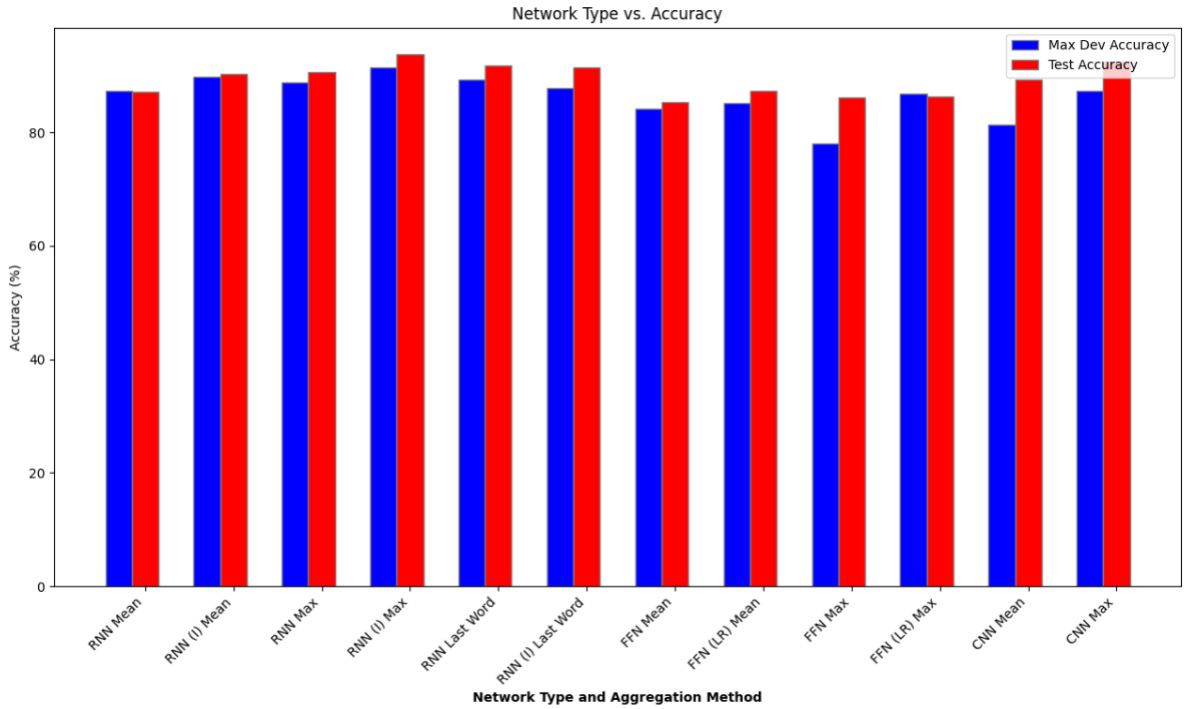
Network Type	Aggregation Method	Number of Epochs	Training Time (s)
RNN	Mean	61	1128.06
RNN (Improved)	Mean	14	508.76
RNN	Max	36	670.66
RNN (Improved)	Max	18	728.31
RNN	Last Word	40	782.68
RNN (Improved)	Last Word	16	602.17
FFN	Mean	77	92.1

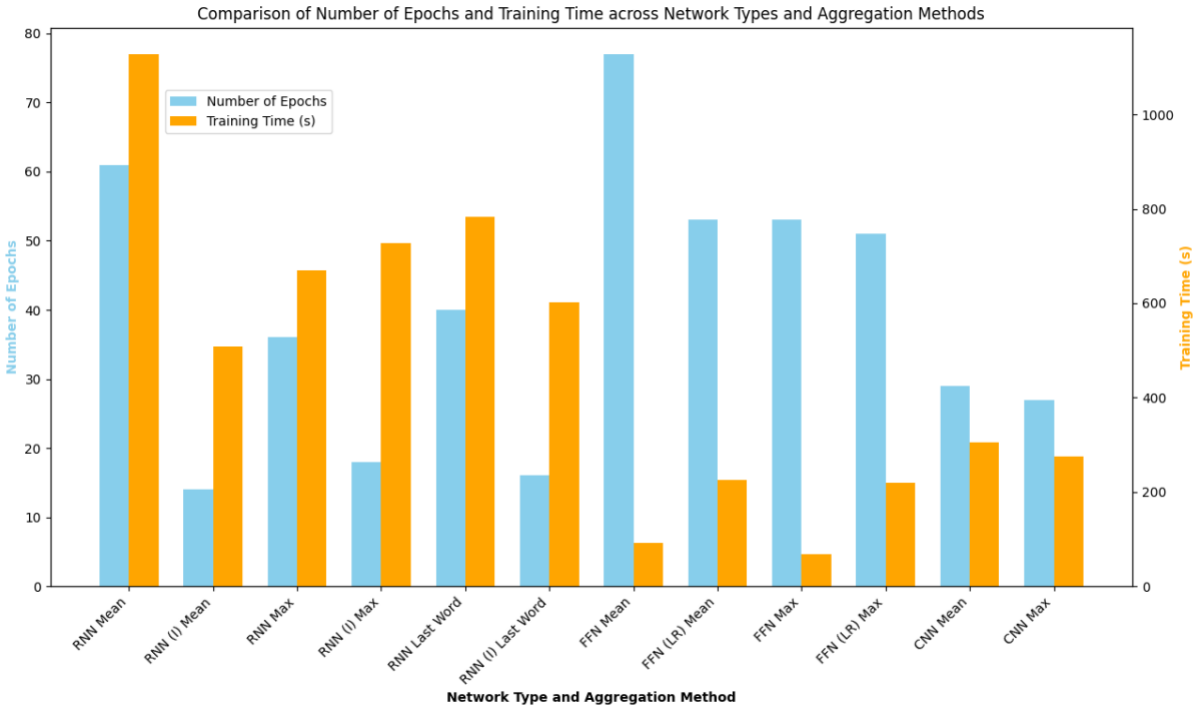
FFN (Leaky ReLU)	Mean	53	226.21
FFN	Max	53	66.98
FFN (Leaky ReLU)	Max	51	218.62
CNN	Mean	29	305.28
CNN	Max	27	275.10

e) *Report the accuracy on the test set, as well as the accuracy on the development set for each epoch during training.*

Network Type	**Aggregation Method**	**Max Dev Accuracy**	**Test Accuracy**
RNN	Mean	87.40%	87.20%
RNN (Improved)	Mean	89.80%	90.40%
RNN	Max	88.80%	90.60%
RNN (Improved)	Max	91.40%	93.80%
RNN	Last Word	89.40%	91.80%
RNN (Improved)	Last Word	87.80%	91.40%
FFN	Mean	84.20%	85.40%

FFN (Leaky ReLU)	Mean	85.20%	87.40%
FFN	Max	78.00%	86.20%
FFN (Leaky ReLU)	Max	86.80%	86.40%
CNN	Mean	81.40%	89.40%
CNN	Max	87.40%	92.40%





In conclusion, the empirical results from our series of experiments suggest that the Improved Recurrent Neural Network (RNN) using Max pooling as the aggregation method outperforms the other models, achieving the highest test accuracy of 93.8%. This model also shows a strong development accuracy of 91.4%, indicating robustness and generalizability from the development set to unseen test data.

Several factors may contribute to the superiority of the Improved RNN with Max pooling. First, RNNs are inherently well-suited for sequence data, capable of capturing temporal dependencies and nuances in sentence structures that are critical for classification tasks. The improvement in the RNN architecture likely enhances this ability, potentially through the introduction of additional layers, bidirectionality, or more sophisticated gating mechanisms like those found in LSTMs or GRUs.

Max pooling as an aggregation method may be particularly effective in this context because it focuses on the most salient features extracted by the RNN, which could be crucial in distinguishing between different classes. This suggests that the most active neurons in the network are carrying significant predictive information, which Max pooling effectively captures.

The use of Mean pooling in both the standard and improved RNN models also yields high test accuracies, with the improved RNN reaching 90.4%. This indicates that averaging features across the sequence can still capture enough information for high performance, though it is slightly less effective than Max pooling.

The Last Word aggregation method, which intuitively would capture the most recent and possibly the most informative representation in a sequence, performs well but does not outperform Max pooling. This could be due to the dispersion of relevant information throughout the sequence rather than its concentration at the end.

Feedforward Neural Networks (FFN), both with standard ReLU and leaky ReLU activations, show lower performance compared to RNNs, which might be attributed to their inability to model sequence information inherently. The introduction of leaky ReLU does not significantly enhance the performance, suggesting that the problem may not lie in the activation function but rather in the model's architecture.

The Convolutional Neural Network (CNN), while traditionally strong in capturing local and positional features, does not match the performance of the RNN models. This could be due to the nature of the task, which may rely more on understanding long-term dependencies and the overall context of the sentence rather than local patterns that CNNs excel at capturing.

We have included the dev accuracy for every epoch and model in the appendix

Appendix

Convolutional Neural Network with Max aggregation:

Epoch 1/1000, Loss: 1.4924, Dev Accuracy: 48.40%
Epoch 2/1000, Loss: 1.3342, Dev Accuracy: 61.80%
Epoch 3/1000, Loss: 1.2528, Dev Accuracy: 65.20%
Epoch 4/1000, Loss: 1.2129, Dev Accuracy: 66.40%
Epoch 5/1000, Loss: 1.1838, Dev Accuracy: 69.20%
Epoch 6/1000, Loss: 1.1483, Dev Accuracy: 77.80%
Epoch 7/1000, Loss: 1.0771, Dev Accuracy: 83.00%
Epoch 8/1000, Loss: 1.0333, Dev Accuracy: 85.00%
Epoch 9/1000, Loss: 1.0119, Dev Accuracy: 85.60%
Epoch 10/1000, Loss: 0.9989, Dev Accuracy: 86.20%
Epoch 11/1000, Loss: 0.9866, Dev Accuracy: 86.40%
Epoch 12/1000, Loss: 0.9791, Dev Accuracy: 86.40%
Epoch 13/1000, Loss: 0.9706, Dev Accuracy: 86.20%
Epoch 14/1000, Loss: 0.9630, Dev Accuracy: 86.60%
Epoch 15/1000, Loss: 0.9583, Dev Accuracy: 86.40%
Epoch 16/1000, Loss: 0.9543, Dev Accuracy: 86.20%
Epoch 17/1000, Loss: 0.9510, Dev Accuracy: 86.00%
Epoch 18/1000, Loss: 0.9482, Dev Accuracy: 87.20%
Epoch 19/1000, Loss: 0.9454, Dev Accuracy: 87.20%
Epoch 20/1000, Loss: 0.9428, Dev Accuracy: 86.40%
Epoch 21/1000, Loss: 0.9410, Dev Accuracy: 87.00%
Epoch 22/1000, Loss: 0.9401, Dev Accuracy: 87.40%
Epoch 23/1000, Loss: 0.9382, Dev Accuracy: 87.00%
Epoch 24/1000, Loss: 0.9366, Dev Accuracy: 87.20%
Epoch 25/1000, Loss: 0.9348, Dev Accuracy: 87.00%
Epoch 26/1000, Loss: 0.9331, Dev Accuracy: 87.00%
Epoch 27/1000, Loss: 0.9327, Dev Accuracy: 86.60%
Stopping training early! Best Dev Accuracy: 87.40%
Training complete for Max Convolutional Neural Network. Time taken: 275.10 seconds.

Convolutional Neural Network with Mean aggregation:

Epoch 1/1000, Loss: 1.1709, Dev Accuracy: 68.40%
Epoch 2/1000, Loss: 1.1561, Dev Accuracy: 71.80%
Epoch 3/1000, Loss: 1.1436, Dev Accuracy: 72.80%
Epoch 4/1000, Loss: 1.1312, Dev Accuracy: 74.00%
Epoch 5/1000, Loss: 1.1250, Dev Accuracy: 75.40%
Epoch 6/1000, Loss: 1.1137, Dev Accuracy: 75.40%
Epoch 7/1000, Loss: 1.1041, Dev Accuracy: 76.00%
Epoch 8/1000, Loss: 1.1008, Dev Accuracy: 75.60%
Epoch 9/1000, Loss: 1.0947, Dev Accuracy: 76.80%
Epoch 10/1000, Loss: 1.0873, Dev Accuracy: 77.80%
Epoch 11/1000, Loss: 1.0806, Dev Accuracy: 78.40%
Epoch 12/1000, Loss: 1.0736, Dev Accuracy: 78.60%
Epoch 13/1000, Loss: 1.0686, Dev Accuracy: 79.20%
Epoch 14/1000, Loss: 1.0645, Dev Accuracy: 79.60%
Epoch 15/1000, Loss: 1.0587, Dev Accuracy: 79.00%
Epoch 16/1000, Loss: 1.0557, Dev Accuracy: 80.00%
Epoch 17/1000, Loss: 1.0526, Dev Accuracy: 80.20%
Epoch 18/1000, Loss: 1.0489, Dev Accuracy: 80.40%
Epoch 19/1000, Loss: 1.0435, Dev Accuracy: 79.80%
Epoch 20/1000, Loss: 1.0413, Dev Accuracy: 80.00%
Epoch 21/1000, Loss: 1.0409, Dev Accuracy: 80.20%
Epoch 22/1000, Loss: 1.0378, Dev Accuracy: 80.60%
Epoch 23/1000, Loss: 1.0340, Dev Accuracy: 80.80%
Epoch 24/1000, Loss: 1.0308, Dev Accuracy: 81.40%
Epoch 25/1000, Loss: 1.0304, Dev Accuracy: 80.40%
Epoch 26/1000, Loss: 1.0273, Dev Accuracy: 80.60%
Epoch 27/1000, Loss: 1.0265, Dev Accuracy: 81.00%
Epoch 28/1000, Loss: 1.0220, Dev Accuracy: 81.20%
Epoch 29/1000, Loss: 1.0168, Dev Accuracy: 81.00%
Stopping training early! Best Dev Accuracy: 81.40%

Training complete for Mean Convolutional Neural Network. Time taken: 305.28 seconds.

Feedforward Neural Network with Mean Aggregation

Epoch 1/100, Loss: 1.5630, Dev Accuracy: 38.00%
Epoch 2/100, Loss: 1.4549, Dev Accuracy: 52.00%
Epoch 3/100, Loss: 1.3848, Dev Accuracy: 53.20%
Epoch 4/100, Loss: 1.3678, Dev Accuracy: 53.00%
Epoch 5/100, Loss: 1.3465, Dev Accuracy: 55.80%
Epoch 6/100, Loss: 1.3259, Dev Accuracy: 58.80%
Epoch 7/100, Loss: 1.2976, Dev Accuracy: 63.20%
Epoch 8/100, Loss: 1.2713, Dev Accuracy: 64.00%
Epoch 9/100, Loss: 1.2553, Dev Accuracy: 66.20%
Epoch 10/100, Loss: 1.2390, Dev Accuracy: 65.40%
Epoch 11/100, Loss: 1.2325, Dev Accuracy: 66.60%
Epoch 12/100, Loss: 1.2194, Dev Accuracy: 66.40%
Epoch 13/100, Loss: 1.2183, Dev Accuracy: 66.20%
Epoch 14/100, Loss: 1.2070, Dev Accuracy: 67.20%
Epoch 15/100, Loss: 1.2068, Dev Accuracy: 65.40%
Epoch 16/100, Loss: 1.2023, Dev Accuracy: 67.00%
Epoch 17/100, Loss: 1.1985, Dev Accuracy: 67.40%
Epoch 18/100, Loss: 1.1948, Dev Accuracy: 67.80%
Epoch 19/100, Loss: 1.1884, Dev Accuracy: 67.00%
Epoch 20/100, Loss: 1.1831, Dev Accuracy: 67.80%
Epoch 21/100, Loss: 1.1803, Dev Accuracy: 67.00%

Epoch 22/100, Loss: 1.1781, Dev Accuracy: 67.60%
Epoch 23/100, Loss: 1.1756, Dev Accuracy: 67.80%
Epoch 24/100, Loss: 1.1773, Dev Accuracy: 67.60%
Epoch 25/100, Loss: 1.1710, Dev Accuracy: 67.60%
Epoch 26/100, Loss: 1.1707, Dev Accuracy: 68.20%
Epoch 27/100, Loss: 1.1647, Dev Accuracy: 68.60%
Epoch 28/100, Loss: 1.1671, Dev Accuracy: 67.60%
Epoch 29/100, Loss: 1.1611, Dev Accuracy: 68.20%
Epoch 30/100, Loss: 1.1647, Dev Accuracy: 68.00%
Epoch 31/100, Loss: 1.1647, Dev Accuracy: 67.40%
Epoch 32/100, Loss: 1.1588, Dev Accuracy: 68.00%
Epoch 33/100, Loss: 1.1591, Dev Accuracy: 68.80%
Epoch 34/100, Loss: 1.1595, Dev Accuracy: 68.00%
Epoch 35/100, Loss: 1.1533, Dev Accuracy: 68.20%
Epoch 36/100, Loss: 1.1522, Dev Accuracy: 69.00%
Epoch 37/100, Loss: 1.1514, Dev Accuracy: 68.00%
Epoch 38/100, Loss: 1.1474, Dev Accuracy: 68.40%
Epoch 39/100, Loss: 1.1568, Dev Accuracy: 68.60%
Epoch 40/100, Loss: 1.1533, Dev Accuracy: 68.60%
Epoch 41/100, Loss: 1.1443, Dev Accuracy: 69.20%
Epoch 42/100, Loss: 1.1443, Dev Accuracy: 69.00%
Epoch 43/100, Loss: 1.1469, Dev Accuracy: 67.80%
Epoch 44/100, Loss: 1.1449, Dev Accuracy: 68.80%
Epoch 45/100, Loss: 1.1430, Dev Accuracy: 68.00%
Epoch 46/100, Loss: 1.1423, Dev Accuracy: 69.60%
Epoch 47/100, Loss: 1.1348, Dev Accuracy: 68.40%
Epoch 48/100, Loss: 1.1403, Dev Accuracy: 67.80%
Epoch 49/100, Loss: 1.1358, Dev Accuracy: 68.80%
Epoch 50/100, Loss: 1.1342, Dev Accuracy: 68.00%
Epoch 51/100, Loss: 1.1386, Dev Accuracy: 70.80%
Epoch 52/100, Loss: 1.1330, Dev Accuracy: 70.60%
Epoch 53/100, Loss: 1.1265, Dev Accuracy: 70.80%
Epoch 54/100, Loss: 1.1277, Dev Accuracy: 72.40%
Epoch 55/100, Loss: 1.1060, Dev Accuracy: 76.40%
Epoch 56/100, Loss: 1.0835, Dev Accuracy: 78.20%
Epoch 57/100, Loss: 1.0638, Dev Accuracy: 79.00%
Epoch 58/100, Loss: 1.0512, Dev Accuracy: 80.60%
Epoch 59/100, Loss: 1.0430, Dev Accuracy: 82.00%
Epoch 60/100, Loss: 1.0357, Dev Accuracy: 82.60%
Epoch 61/100, Loss: 1.0304, Dev Accuracy: 82.60%

Epoch 62/100, Loss: 1.0285, Dev Accuracy: 82.40%
Epoch 63/100, Loss: 1.0261, Dev Accuracy: 82.20%
Epoch 64/100, Loss: 1.0182, Dev Accuracy: 82.40%
Epoch 65/100, Loss: 1.0131, Dev Accuracy: 83.20%
Epoch 66/100, Loss: 1.0129, Dev Accuracy: 83.80%
Epoch 67/100, Loss: 1.0127, Dev Accuracy: 84.20%
Epoch 68/100, Loss: 1.0096, Dev Accuracy: 83.80%
Epoch 69/100, Loss: 1.0083, Dev Accuracy: 83.60%
Epoch 70/100, Loss: 1.0072, Dev Accuracy: 84.00%
Epoch 71/100, Loss: 1.0017, Dev Accuracy: 84.20%
Epoch 72/100, Loss: 1.0051, Dev Accuracy: 84.00%
Epoch 73/100, Loss: 1.0017, Dev Accuracy: 84.00%
Epoch 74/100, Loss: 1.0000, Dev Accuracy: 84.00%
Epoch 75/100, Loss: 0.9946, Dev Accuracy: 83.40%
Epoch 76/100, Loss: 0.9948, Dev Accuracy: 83.60%
Epoch 77/100, Loss: 0.9982, Dev Accuracy: 84.00%
Stopping training early! Best Dev Accuracy: 84.20%
Training complete for Mean Feedforward Neural Network. Time taken: 92.10 seconds.

Feedforward Neural Network (Leaky ReLu) with Mean Aggregation

Epoch 2/100, Loss: 1.4022, Dev Accuracy: 38.00%
Epoch 3/100, Loss: 1.3851, Dev Accuracy: 38.00%
Epoch 4/100, Loss: 1.3711, Dev Accuracy: 38.00%
Epoch 5/100, Loss: 1.3424, Dev Accuracy: 38.20%

Epoch 6/100, Loss: 1.2791, Dev Accuracy: 50.20%
Epoch 7/100, Loss: 1.1771, Dev Accuracy: 52.20%
Epoch 8/100, Loss: 1.1079, Dev Accuracy: 52.80%
Epoch 9/100, Loss: 1.0756, Dev Accuracy: 49.40%
Epoch 10/100, Loss: 1.0347, Dev Accuracy: 54.20%
Epoch 11/100, Loss: 1.0208, Dev Accuracy: 59.00%
Epoch 12/100, Loss: 0.9382, Dev Accuracy: 61.80%
Epoch 13/100, Loss: 0.8451, Dev Accuracy: 65.40%
Epoch 14/100, Loss: 0.7650, Dev Accuracy: 67.80%
Epoch 15/100, Loss: 0.7229, Dev Accuracy: 70.00%
Epoch 16/100, Loss: 0.6676, Dev Accuracy: 72.80%
Epoch 17/100, Loss: 0.6587, Dev Accuracy: 74.40%
Epoch 18/100, Loss: 0.6185, Dev Accuracy: 75.20%
Epoch 19/100, Loss: 0.5870, Dev Accuracy: 76.40%
Epoch 20/100, Loss: 0.5638, Dev Accuracy: 79.20%
Epoch 21/100, Loss: 0.5347, Dev Accuracy: 79.40%
Epoch 22/100, Loss: 0.5170, Dev Accuracy: 81.00%
Epoch 23/100, Loss: 0.4878, Dev Accuracy: 83.00%
Epoch 24/100, Loss: 0.4744, Dev Accuracy: 81.80%
Epoch 25/100, Loss: 0.4534, Dev Accuracy: 81.20%
Epoch 26/100, Loss: 0.4395, Dev Accuracy: 82.80%
Epoch 27/100, Loss: 0.4276, Dev Accuracy: 83.40%
Epoch 28/100, Loss: 0.4172, Dev Accuracy: 82.40%
Epoch 29/100, Loss: 0.4116, Dev Accuracy: 83.60%
Epoch 30/100, Loss: 0.4014, Dev Accuracy: 82.60%
Epoch 31/100, Loss: 0.3859, Dev Accuracy: 82.20%
Epoch 32/100, Loss: 0.3706, Dev Accuracy: 82.60%
Epoch 33/100, Loss: 0.3632, Dev Accuracy: 83.80%
Epoch 34/100, Loss: 0.3625, Dev Accuracy: 82.20%
Epoch 35/100, Loss: 0.3601, Dev Accuracy: 81.80%
Epoch 36/100, Loss: 0.3353, Dev Accuracy: 83.00%
Epoch 37/100, Loss: 0.3228, Dev Accuracy: 83.00%
Epoch 38/100, Loss: 0.3242, Dev Accuracy: 84.20%
Epoch 39/100, Loss: 0.3287, Dev Accuracy: 83.80%
Epoch 40/100, Loss: 0.3014, Dev Accuracy: 83.60%
Epoch 41/100, Loss: 0.3015, Dev Accuracy: 84.40%
Epoch 42/100, Loss: 0.2944, Dev Accuracy: 83.40%
Epoch 43/100, Loss: 0.3113, Dev Accuracy: 85.20%
Epoch 44/100, Loss: 0.2941, Dev Accuracy: 83.80%
Epoch 45/100, Loss: 0.2848, Dev Accuracy: 84.60%

Epoch 46/100, Loss: 0.2725, Dev Accuracy: 84.80%
Epoch 47/100, Loss: 0.2765, Dev Accuracy: 83.80%
Epoch 48/100, Loss: 0.2619, Dev Accuracy: 84.80%
Epoch 49/100, Loss: 0.2466, Dev Accuracy: 84.60%
Epoch 50/100, Loss: 0.2497, Dev Accuracy: 84.20%
Epoch 51/100, Loss: 0.2552, Dev Accuracy: 84.80%
Epoch 52/100, Loss: 0.2638, Dev Accuracy: 84.80%
Epoch 53/100, Loss: 0.2445, Dev Accuracy: 84.80%
Stopping training early! Best Dev Accuracy: 85.20%
Training complete for Improved Mean Feedforward Neural Network. Time taken: 226.21 seconds.

Feedforward Neural Network with Max Aggregation

Epoch 1/100, Loss: 1.5306, Dev Accuracy: 46.40%
Epoch 2/100, Loss: 1.4157, Dev Accuracy: 48.40%
Epoch 3/100, Loss: 1.3942, Dev Accuracy: 51.00%
Epoch 4/100, Loss: 1.3736, Dev Accuracy: 53.40%
Epoch 5/100, Loss: 1.3480, Dev Accuracy: 57.40%
Epoch 6/100, Loss: 1.3070, Dev Accuracy: 61.80%
Epoch 7/100, Loss: 1.2734, Dev Accuracy: 61.60%
Epoch 8/100, Loss: 1.2551, Dev Accuracy: 61.80%
Epoch 9/100, Loss: 1.2402, Dev Accuracy: 63.00%
Epoch 10/100, Loss: 1.2316, Dev Accuracy: 62.60%
Epoch 11/100, Loss: 1.2232, Dev Accuracy: 65.20%
Epoch 12/100, Loss: 1.2244, Dev Accuracy: 64.80%
Epoch 13/100, Loss: 1.2124, Dev Accuracy: 66.20%
Epoch 14/100, Loss: 1.2041, Dev Accuracy: 68.00%
Epoch 15/100, Loss: 1.1969, Dev Accuracy: 66.40%
Epoch 16/100, Loss: 1.1917, Dev Accuracy: 67.40%
Epoch 17/100, Loss: 1.1884, Dev Accuracy: 68.20%
Epoch 18/100, Loss: 1.1899, Dev Accuracy: 67.60%
Epoch 19/100, Loss: 1.1805, Dev Accuracy: 67.60%
Epoch 20/100, Loss: 1.1781, Dev Accuracy: 66.80%
Epoch 21/100, Loss: 1.1789, Dev Accuracy: 68.20%
Epoch 22/100, Loss: 1.1758, Dev Accuracy: 66.60%
Epoch 23/100, Loss: 1.1729, Dev Accuracy: 67.20%
Epoch 24/100, Loss: 1.1734, Dev Accuracy: 68.60%
Epoch 25/100, Loss: 1.1674, Dev Accuracy: 70.60%
Epoch 26/100, Loss: 1.1577, Dev Accuracy: 72.20%

Epoch 27/100, Loss: 1.1487, Dev Accuracy: 72.80%
Epoch 28/100, Loss: 1.1470, Dev Accuracy: 72.00%
Epoch 29/100, Loss: 1.1437, Dev Accuracy: 73.20%
Epoch 30/100, Loss: 1.1323, Dev Accuracy: 72.80%
Epoch 31/100, Loss: 1.1249, Dev Accuracy: 73.00%
Epoch 32/100, Loss: 1.1199, Dev Accuracy: 72.20%
Epoch 33/100, Loss: 1.1187, Dev Accuracy: 73.00%
Epoch 34/100, Loss: 1.1168, Dev Accuracy: 73.20%
Epoch 35/100, Loss: 1.1177, Dev Accuracy: 74.20%
Epoch 36/100, Loss: 1.1118, Dev Accuracy: 72.60%
Epoch 37/100, Loss: 1.1092, Dev Accuracy: 72.80%
Epoch 38/100, Loss: 1.1105, Dev Accuracy: 73.20%
Epoch 39/100, Loss: 1.1111, Dev Accuracy: 72.40%
Epoch 40/100, Loss: 1.1047, Dev Accuracy: 73.80%
Epoch 41/100, Loss: 1.1048, Dev Accuracy: 74.40%
Epoch 42/100, Loss: 1.0931, Dev Accuracy: 77.40%
Epoch 43/100, Loss: 1.0734, Dev Accuracy: 78.00%
Epoch 44/100, Loss: 1.0698, Dev Accuracy: 77.60%
Epoch 45/100, Loss: 1.0716, Dev Accuracy: 76.60%
Epoch 46/100, Loss: 1.0684, Dev Accuracy: 76.40%
Epoch 47/100, Loss: 1.0709, Dev Accuracy: 77.00%
Epoch 48/100, Loss: 1.0637, Dev Accuracy: 77.20%
Epoch 49/100, Loss: 1.0653, Dev Accuracy: 77.00%
Epoch 50/100, Loss: 1.0631, Dev Accuracy: 77.00%
Epoch 51/100, Loss: 1.0646, Dev Accuracy: 77.20%
Epoch 52/100, Loss: 1.0556, Dev Accuracy: 76.80%
Epoch 53/100, Loss: 1.0566, Dev Accuracy: 76.60%
Stopping training early! Best Dev Accuracy: 78.00%
Training complete for Max Feedforward Neural Network. Time taken: 66.98 seconds.

Feedforward Neural Network (Leaky ReLu) with Max Aggregation

Epoch 1/100, Loss: 1.4622, Dev Accuracy: 38.00%
Epoch 2/100, Loss: 1.4220, Dev Accuracy: 38.00%
Epoch 3/100, Loss: 1.4144, Dev Accuracy: 38.00%
Epoch 4/100, Loss: 1.4121, Dev Accuracy: 38.00%
Epoch 5/100, Loss: 1.4095, Dev Accuracy: 38.00%
Epoch 6/100, Loss: 1.4128, Dev Accuracy: 38.00%
Epoch 7/100, Loss: 1.4068, Dev Accuracy: 38.00%
Epoch 8/100, Loss: 1.4092, Dev Accuracy: 38.00%

Epoch 9/100, Loss: 1.3439, Dev Accuracy: 49.00%
Epoch 10/100, Loss: 1.1979, Dev Accuracy: 49.60%
Epoch 11/100, Loss: 1.1252, Dev Accuracy: 51.60%
Epoch 12/100, Loss: 1.0825, Dev Accuracy: 52.00%
Epoch 13/100, Loss: 1.0559, Dev Accuracy: 52.00%
Epoch 14/100, Loss: 1.0233, Dev Accuracy: 52.80%
Epoch 15/100, Loss: 1.0002, Dev Accuracy: 53.40%
Epoch 16/100, Loss: 0.9824, Dev Accuracy: 54.20%
Epoch 17/100, Loss: 0.9291, Dev Accuracy: 62.20%
Epoch 18/100, Loss: 0.8851, Dev Accuracy: 61.60%
Epoch 19/100, Loss: 0.8501, Dev Accuracy: 62.00%
Epoch 20/100, Loss: 0.8242, Dev Accuracy: 70.80%
Epoch 21/100, Loss: 0.7823, Dev Accuracy: 72.80%
Epoch 22/100, Loss: 0.7057, Dev Accuracy: 78.40%
Epoch 23/100, Loss: 0.6382, Dev Accuracy: 82.40%
Epoch 24/100, Loss: 0.6005, Dev Accuracy: 80.60%
Epoch 25/100, Loss: 0.5622, Dev Accuracy: 82.80%
Epoch 26/100, Loss: 0.5290, Dev Accuracy: 83.40%
Epoch 27/100, Loss: 0.5436, Dev Accuracy: 84.20%
Epoch 28/100, Loss: 0.5168, Dev Accuracy: 86.40%
Epoch 29/100, Loss: 0.4882, Dev Accuracy: 86.20%
Epoch 30/100, Loss: 0.4739, Dev Accuracy: 86.40%
Epoch 31/100, Loss: 0.4455, Dev Accuracy: 86.20%
Epoch 32/100, Loss: 0.4604, Dev Accuracy: 86.00%
Epoch 33/100, Loss: 0.4557, Dev Accuracy: 87.20%
Epoch 34/100, Loss: 0.4203, Dev Accuracy: 85.60%
Epoch 35/100, Loss: 0.4238, Dev Accuracy: 85.60%
Epoch 36/100, Loss: 0.4404, Dev Accuracy: 86.80%
Epoch 37/100, Loss: 0.4174, Dev Accuracy: 86.80%
Epoch 38/100, Loss: 0.4131, Dev Accuracy: 86.60%
Epoch 39/100, Loss: 0.3875, Dev Accuracy: 86.00%
Epoch 40/100, Loss: 0.3804, Dev Accuracy: 86.40%
Epoch 41/100, Loss: 0.3747, Dev Accuracy: 87.80%
Epoch 42/100, Loss: 0.3770, Dev Accuracy: 86.60%
Epoch 43/100, Loss: 0.3513, Dev Accuracy: 87.20%
Epoch 44/100, Loss: 0.3581, Dev Accuracy: 87.20%
Epoch 45/100, Loss: 0.3429, Dev Accuracy: 86.80%
Epoch 46/100, Loss: 0.3652, Dev Accuracy: 87.00%
Epoch 47/100, Loss: 0.3489, Dev Accuracy: 87.00%
Epoch 48/100, Loss: 0.3452, Dev Accuracy: 86.20%

Epoch 49/100, Loss: 0.3368, Dev Accuracy: 86.80%
Epoch 50/100, Loss: 0.3384, Dev Accuracy: 86.80%
Epoch 51/100, Loss: 0.3326, Dev Accuracy: 87.40%
Stopping training early! Best Dev Accuracy: 87.80%
Training complete for Improved Max Feedforward Neural Network. Time taken: 218.62 seconds.

Recurrent Neural Network with Mean Aggregation

Epoch 1/100, Loss: 1.5662, Dev Accuracy: 34.00%
Epoch 2/100, Loss: 1.5575, Dev Accuracy: 34.00%
Epoch 3/100, Loss: 1.5478, Dev Accuracy: 34.40%
Epoch 4/100, Loss: 1.4734, Dev Accuracy: 49.40%
Epoch 5/100, Loss: 1.4059, Dev Accuracy: 49.80%
Epoch 6/100, Loss: 1.3855, Dev Accuracy: 57.80%
Epoch 7/100, Loss: 1.2857, Dev Accuracy: 67.40%
Epoch 8/100, Loss: 1.2160, Dev Accuracy: 77.20%
Epoch 9/100, Loss: 1.1633, Dev Accuracy: 78.40%
Epoch 10/100, Loss: 1.1277, Dev Accuracy: 79.80%
Epoch 11/100, Loss: 1.1131, Dev Accuracy: 82.20%
Epoch 12/100, Loss: 1.0909, Dev Accuracy: 80.80%
Epoch 13/100, Loss: 1.0832, Dev Accuracy: 80.60%
Epoch 14/100, Loss: 1.0702, Dev Accuracy: 83.20%
Epoch 15/100, Loss: 1.0679, Dev Accuracy: 82.80%
Epoch 16/100, Loss: 1.0810, Dev Accuracy: 82.60%
Epoch 17/100, Loss: 1.0560, Dev Accuracy: 84.20%
Epoch 18/100, Loss: 1.0458, Dev Accuracy: 84.80%
Epoch 19/100, Loss: 1.0375, Dev Accuracy: 83.80%
Epoch 20/100, Loss: 1.0355, Dev Accuracy: 83.00%
Epoch 21/100, Loss: 1.0307, Dev Accuracy: 83.60%
Epoch 22/100, Loss: 1.0284, Dev Accuracy: 83.60%
Epoch 23/100, Loss: 1.0265, Dev Accuracy: 85.20%
Epoch 24/100, Loss: 1.0205, Dev Accuracy: 83.40%
Epoch 25/100, Loss: 1.0147, Dev Accuracy: 85.40%
Epoch 26/100, Loss: 1.0119, Dev Accuracy: 84.00%
Epoch 27/100, Loss: 1.0084, Dev Accuracy: 83.80%
Epoch 28/100, Loss: 1.0001, Dev Accuracy: 84.20%
Epoch 29/100, Loss: 1.0020, Dev Accuracy: 85.20%
Epoch 30/100, Loss: 1.0004, Dev Accuracy: 85.40%
Epoch 31/100, Loss: 0.9989, Dev Accuracy: 86.00%

Epoch 32/100, Loss: 1.0018, Dev Accuracy: 84.60%
Epoch 33/100, Loss: 1.0019, Dev Accuracy: 85.00%
Epoch 34/100, Loss: 0.9961, Dev Accuracy: 85.40%
Epoch 35/100, Loss: 0.9926, Dev Accuracy: 85.20%
Epoch 36/100, Loss: 0.9947, Dev Accuracy: 85.00%
Epoch 37/100, Loss: 1.0140, Dev Accuracy: 83.40%
Epoch 38/100, Loss: 0.9961, Dev Accuracy: 86.40%
Epoch 39/100, Loss: 0.9861, Dev Accuracy: 85.80%
Epoch 40/100, Loss: 0.9819, Dev Accuracy: 86.00%
Epoch 41/100, Loss: 0.9772, Dev Accuracy: 86.60%
Epoch 42/100, Loss: 0.9734, Dev Accuracy: 86.20%
Epoch 43/100, Loss: 0.9745, Dev Accuracy: 86.00%
Epoch 44/100, Loss: 0.9715, Dev Accuracy: 85.20%
Epoch 45/100, Loss: 0.9699, Dev Accuracy: 86.00%
Epoch 46/100, Loss: 0.9690, Dev Accuracy: 86.40%
Epoch 47/100, Loss: 0.9702, Dev Accuracy: 85.40%
Epoch 48/100, Loss: 0.9703, Dev Accuracy: 86.60%
Epoch 49/100, Loss: 0.9696, Dev Accuracy: 86.00%
Epoch 50/100, Loss: 0.9664, Dev Accuracy: 87.00%
Epoch 51/100, Loss: 0.9638, Dev Accuracy: 87.40%
Epoch 52/100, Loss: 0.9614, Dev Accuracy: 86.80%
Epoch 53/100, Loss: 0.9621, Dev Accuracy: 86.00%
Epoch 54/100, Loss: 0.9624, Dev Accuracy: 86.40%
Epoch 55/100, Loss: 0.9648, Dev Accuracy: 85.80%
Epoch 56/100, Loss: 0.9643, Dev Accuracy: 87.00%
Epoch 57/100, Loss: 0.9599, Dev Accuracy: 86.80%
Epoch 58/100, Loss: 0.9628, Dev Accuracy: 87.20%
Epoch 59/100, Loss: 0.9601, Dev Accuracy: 86.40%
Epoch 60/100, Loss: 0.9731, Dev Accuracy: 85.60%
Epoch 61/100, Loss: 0.9751, Dev Accuracy: 86.80%
Stopping training early! Best Dev Accuracy: 87.40%
Training complete for Mean Recurrent Neural Network. Time taken: 1128.06 seconds.

Recurrent Neural Network with Max Aggregation

Epoch 1/100, Loss: 1.4721, Dev Accuracy: 61.20%
Epoch 2/100, Loss: 1.2102, Dev Accuracy: 71.80%
Epoch 3/100, Loss: 1.1237, Dev Accuracy: 80.60%
Epoch 4/100, Loss: 1.0735, Dev Accuracy: 81.00%

Epoch 5/100, Loss: 1.0480, Dev Accuracy: 83.20%
Epoch 6/100, Loss: 1.0402, Dev Accuracy: 83.80%
Epoch 7/100, Loss: 1.0225, Dev Accuracy: 85.40%
Epoch 8/100, Loss: 1.0135, Dev Accuracy: 85.40%
Epoch 9/100, Loss: 0.9965, Dev Accuracy: 87.20%
Epoch 10/100, Loss: 0.9903, Dev Accuracy: 86.60%
Epoch 11/100, Loss: 0.9811, Dev Accuracy: 86.60%
Epoch 12/100, Loss: 0.9948, Dev Accuracy: 21.00%
Epoch 13/100, Loss: 1.0505, Dev Accuracy: 86.80%
Epoch 14/100, Loss: 0.9846, Dev Accuracy: 87.20%
Epoch 15/100, Loss: 0.9729, Dev Accuracy: 86.20%
Epoch 16/100, Loss: 0.9696, Dev Accuracy: 86.60%
Epoch 17/100, Loss: 0.9624, Dev Accuracy: 86.60%
Epoch 18/100, Loss: 0.9590, Dev Accuracy: 87.40%
Epoch 19/100, Loss: 0.9567, Dev Accuracy: 88.40%
Epoch 20/100, Loss: 0.9550, Dev Accuracy: 88.00%
Epoch 21/100, Loss: 0.9508, Dev Accuracy: 87.80%
Epoch 22/100, Loss: 0.9481, Dev Accuracy: 87.80%
Epoch 23/100, Loss: 0.9467, Dev Accuracy: 87.60%
Epoch 24/100, Loss: 0.9471, Dev Accuracy: 87.80%
Epoch 25/100, Loss: 0.9468, Dev Accuracy: 88.20%
Epoch 26/100, Loss: 0.9446, Dev Accuracy: 88.80%
Epoch 27/100, Loss: 0.9451, Dev Accuracy: 88.20%
Epoch 28/100, Loss: 0.9442, Dev Accuracy: 87.80%
Epoch 29/100, Loss: 0.9442, Dev Accuracy: 88.00%
Epoch 30/100, Loss: 0.9436, Dev Accuracy: 87.20%
Epoch 31/100, Loss: 0.9434, Dev Accuracy: 86.60%
Epoch 32/100, Loss: 0.9438, Dev Accuracy: 87.40%
Epoch 33/100, Loss: 0.9436, Dev Accuracy: 87.60%
Epoch 34/100, Loss: 0.9437, Dev Accuracy: 88.00%
Epoch 35/100, Loss: 0.9421, Dev Accuracy: 88.40%
Epoch 36/100, Loss: 0.9421, Dev Accuracy: 87.60%
Stopping training early! Best Dev Accuracy: 88.80%
Training complete for Max Recurrent Neural Network. Time taken: 670.66 seconds.

Recurrent Neural Network with Last Word Representation

Epoch 1/100, Loss: 1.5529, Dev Accuracy: 40.40%
Epoch 2/100, Loss: 1.4710, Dev Accuracy: 44.20%
Epoch 3/100, Loss: 1.3829, Dev Accuracy: 57.60%
Epoch 4/100, Loss: 1.2475, Dev Accuracy: 68.40%
Epoch 5/100, Loss: 1.1683, Dev Accuracy: 75.80%
Epoch 6/100, Loss: 1.1091, Dev Accuracy: 80.00%
Epoch 7/100, Loss: 1.0717, Dev Accuracy: 82.00%
Epoch 8/100, Loss: 1.0495, Dev Accuracy: 84.40%
Epoch 9/100, Loss: 1.0312, Dev Accuracy: 83.20%
Epoch 10/100, Loss: 1.0224, Dev Accuracy: 84.40%
Epoch 11/100, Loss: 1.0140, Dev Accuracy: 85.60%
Epoch 12/100, Loss: 1.0161, Dev Accuracy: 85.20%
Epoch 13/100, Loss: 1.0023, Dev Accuracy: 86.80%
Epoch 14/100, Loss: 0.9908, Dev Accuracy: 85.80%
Epoch 15/100, Loss: 0.9858, Dev Accuracy: 85.80%
Epoch 16/100, Loss: 0.9893, Dev Accuracy: 83.40%
Epoch 17/100, Loss: 0.9929, Dev Accuracy: 86.80%
Epoch 18/100, Loss: 0.9788, Dev Accuracy: 86.40%
Epoch 19/100, Loss: 0.9785, Dev Accuracy: 86.40%
Epoch 20/100, Loss: 0.9715, Dev Accuracy: 86.40%
Epoch 21/100, Loss: 0.9641, Dev Accuracy: 87.40%
Epoch 22/100, Loss: 0.9666, Dev Accuracy: 86.80%
Epoch 23/100, Loss: 0.9658, Dev Accuracy: 88.00%
Epoch 24/100, Loss: 0.9753, Dev Accuracy: 87.20%
Epoch 25/100, Loss: 0.9638, Dev Accuracy: 84.80%
Epoch 26/100, Loss: 0.9609, Dev Accuracy: 87.80%
Epoch 27/100, Loss: 0.9539, Dev Accuracy: 87.60%
Epoch 28/100, Loss: 0.9513, Dev Accuracy: 88.20%
Epoch 29/100, Loss: 0.9495, Dev Accuracy: 88.80%
Epoch 30/100, Loss: 0.9456, Dev Accuracy: 89.40%
Epoch 31/100, Loss: 0.9446, Dev Accuracy: 88.00%
Epoch 32/100, Loss: 0.9424, Dev Accuracy: 88.80%
Epoch 33/100, Loss: 0.9416, Dev Accuracy: 88.60%
Epoch 34/100, Loss: 0.9382, Dev Accuracy: 88.80%
Epoch 35/100, Loss: 0.9378, Dev Accuracy: 88.80%
Epoch 36/100, Loss: 0.9370, Dev Accuracy: 89.00%

Epoch 37/100, Loss: 0.9391, Dev Accuracy: 89.00%
Epoch 38/100, Loss: 0.9368, Dev Accuracy: 87.40%
Epoch 39/100, Loss: 0.9405, Dev Accuracy: 88.00%
Epoch 40/100, Loss: 0.9377, Dev Accuracy: 88.60%
Stopping training early! Best Dev Accuracy: 89.40%
Training complete for Last Word Recurrent Neural Network. Time taken: 782.68 seconds.

Improved Recurrent Neural Network with Mean Aggregation

Epoch 1/100, Loss: 1.4393, Dev Accuracy: 44.40%
Epoch 2/100, Loss: 1.0957, Dev Accuracy: 67.20%
Epoch 3/100, Loss: 0.7056, Dev Accuracy: 78.00%
Epoch 4/100, Loss: 0.5142, Dev Accuracy: 81.80%
Epoch 5/100, Loss: 0.3912, Dev Accuracy: 87.20%
Epoch 6/100, Loss: 0.3082, Dev Accuracy: 84.80%
Epoch 7/100, Loss: 0.2607, Dev Accuracy: 86.40%
Epoch 8/100, Loss: 0.1765, Dev Accuracy: 87.80%
Epoch 9/100, Loss: 0.1293, Dev Accuracy: 89.80%
Epoch 10/100, Loss: 0.0788, Dev Accuracy: 89.40%
Epoch 11/100, Loss: 0.0661, Dev Accuracy: 88.20%
Epoch 12/100, Loss: 0.0440, Dev Accuracy: 88.80%
Epoch 13/100, Loss: 0.0377, Dev Accuracy: 88.00%
Epoch 14/100, Loss: 0.0322, Dev Accuracy: 88.60%
Stopping training early! Best Dev Accuracy: 89.80%
Training complete for Improved Mean Recurrent Neural Network. Time taken: 508.76 seconds.

Improved Recurrent Neural Network with Max Aggregation

Epoch 1/100, Loss: 1.2051, Dev Accuracy: 66.20%
Epoch 2/100, Loss: 0.6123, Dev Accuracy: 80.60%
Epoch 3/100, Loss: 0.3959, Dev Accuracy: 85.00%
Epoch 4/100, Loss: 0.2377, Dev Accuracy: 87.00%
Epoch 5/100, Loss: 0.1414, Dev Accuracy: 89.00%
Epoch 6/100, Loss: 0.0771, Dev Accuracy: 90.00%
Epoch 7/100, Loss: 0.0498, Dev Accuracy: 90.60%

Epoch 8/100, Loss: 0.0293, Dev Accuracy: 89.40%
Epoch 9/100, Loss: 0.0329, Dev Accuracy: 87.60%
Epoch 10/100, Loss: 0.0178, Dev Accuracy: 90.80%
Epoch 11/100, Loss: 0.0118, Dev Accuracy: 90.20%
Epoch 12/100, Loss: 0.0080, Dev Accuracy: 90.40%
Epoch 13/100, Loss: 0.0072, Dev Accuracy: 91.40%
Epoch 14/100, Loss: 0.0068, Dev Accuracy: 90.60%
Epoch 15/100, Loss: 0.0062, Dev Accuracy: 89.00%
Epoch 16/100, Loss: 0.0047, Dev Accuracy: 90.20%
Epoch 17/100, Loss: 0.0018, Dev Accuracy: 89.60%
Epoch 18/100, Loss: 0.0023, Dev Accuracy: 90.20%
Stopping training early! Best Dev Accuracy: 91.40%
Training complete for Improved Max Recurrent Neural Network. Time taken: 728.31 seconds.

Improved Recurrent Neural Network with Last Word Representation

Epoch 1/100, Loss: 1.4302, Dev Accuracy: 46.20%
Epoch 2/100, Loss: 1.1143, Dev Accuracy: 56.80%
Epoch 3/100, Loss: 0.8210, Dev Accuracy: 70.80%
Epoch 4/100, Loss: 0.5400, Dev Accuracy: 82.00%
Epoch 5/100, Loss: 0.3533, Dev Accuracy: 82.60%
Epoch 6/100, Loss: 0.2576, Dev Accuracy: 85.40%
Epoch 7/100, Loss: 0.1765, Dev Accuracy: 84.80%
Epoch 8/100, Loss: 0.1391, Dev Accuracy: 86.80%
Epoch 9/100, Loss: 0.0904, Dev Accuracy: 86.60%
Epoch 10/100, Loss: 0.0587, Dev Accuracy: 85.80%
Epoch 11/100, Loss: 0.0478, Dev Accuracy: 87.80%
Epoch 12/100, Loss: 0.0619, Dev Accuracy: 87.20%
Epoch 13/100, Loss: 0.0434, Dev Accuracy: 86.80%
Epoch 14/100, Loss: 0.0217, Dev Accuracy: 87.40%
Epoch 15/100, Loss: 0.0122, Dev Accuracy: 87.60%
Epoch 16/100, Loss: 0.0062, Dev Accuracy: 87.40%
Stopping training early! Best Dev Accuracy: 87.80%
Training complete for Improved Last Word Recurrent Neural Network. Time taken: 602.17 seconds.