CS4395 Assignment 2

https://github.com/adibarra/hlt

Alec Ibarra
adi220000

1 Introduction and Dataset

In this assignment, we implemented and compared two neural network architectures for sentiment classification of Yelp reviews: a Feedforward Neural Network (FFNN) and a Recurrent Neural Network (RNN). The task is a 5-class sentiment classification problem, where each review is labeled with an integer score from 1 (very negative) to 5 (very positive) indicating its sentiment.

Our experiments involved building models using PyTorch, training them on the provided dataset, and evaluating their performance based on accuracy and loss. The FFNN uses a bag-of-words representation, while the RNN uses word embeddings and processes the sequence of words in the review.

Dataset Statistics

The Yelp dataset is divided into training, validation, and test sets. Table 1 shows the number of samples in each split:

Split	Number of Reviews
Training Set	16,000
Validation Set	800
Test Set	800

Table 1: Dataset statistics for Yelp sentiment classification.

Each review is pre-processed by tokenizing the text into words, converting them to lowercase, and mapping them to indices using a vocabulary built from the training set. In the RNN, we use pre-trained word embeddings to initialize word vectors, while the FFNN uses bag-of-words vectors. Analysis of the dataset shows a balanced distribution for each label.

2 Implementations

The models share key implementation elements. Both use the `Adam` optimizer and the `ReduceLROnPlateau` scheduler, which adjusts learning rates based on performance to aid convergence. They also include early stopping with patience, which halts training if combined training and validation accuracy stagnates over several epochs. Though the training approach is similar, primary differences between the two lie in their architecture.

Both designs balance expressiveness and generalization, using simple yet effective architectures tailored to their inputs. Regularization techniques like dropout help prevent overfitting, while shallow networks and efficient training promote robust generalization. These choices emphasize stability, convergence, and generalization.

The implementations are almost entirely custom, with lots of refactoring to improve efficiency and eliminate deprecated functions. Refactoring also enabled GPU/CUDA support for faster training. Common tasks such as data loading, vectorization, and vocabulary management are centralized in `utils.py`, simplifying reuse and maintenance.

Dependencies

The dependencies required to run the models are minimal and were provided as part of the initial starter package. They are listed in the `requirements.txt` file and include `numpy`, `torch`, and `tqdm`.

Debugging and Resources

During the implementation process, we encountered several challenges, including deprecated functions, tensor shape mismatches, and the need for efficient data handling. To resolve these issues, we relied heavily on the official PyTorch documentation, which helped us identify correct usage patterns and appropriate alternatives to outdated APIs.

We also referred to tutorials on deep learning with PyTorch to guide our use of key components such as `nn.Linear`, `nn.ReLU`, `nn.CrossEntropyLoss`, `Adam`, and `ReduceLROnPlateau`. For debugging, we primarily used print statements and monitored model outputs during training to ensure that each stage of the pipeline was functioning as expected.

While we consulted standard design patterns, the architecture, training loop, and utilities were developed from scratch and tailored to the project.

FFNN

The Feedforward Neural Network (FFNN) was implemented to classify Yelp reviews into one of five sentiment categories. This model uses a bag-of-words representation of the reviews, which means that each review is represented as a vector where each element corresponds to the frequency or presence of a specific word from a vocabulary built from the training data.

Model Architecture. The FFNN takes as input a bag-of-words vector for each review. Each review is represented as a sparse vector, where each element corresponds to the presence or absence of a specific word from the vocabulary. The model consists of:

- **Input Layer**: The input layer takes the bag-of-words vector as input, which is the same size as the vocabulary.
- **Hidden Layers**: The hidden layers are implemented by using `nn.Linear(input_dim, hidden_dim)` followed by a ReLU activation `nn.ReLU()`. These layers also have a `0.3 Dropout Rate` applied to prevent overfitting and encourage generalization.
- **Output Layer**: The output layer is implemented by `nn.Linear(hidden_dim, output_dim)`, where `output_dim` is set to 5 (corresponding to the 5 sentiment classes).

Model Definition. The code for the FFNN model is as follows:

```
import torch
import torch.nn as nn

class FFNN(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int, output_dim: int
    ↪  = 5) -> None:
        super().__init__()
        self.ffnn = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden_dim, output_dim),
        )
        self.loss_fn = nn.CrossEntropyLoss()

    def forward(self, input_vector: torch.Tensor) -> torch.Tensor:
        return self.ffnn(input_vector)
```

```
    def compute_loss(self, predicted_vector: torch.Tensor, gold_label:
    ↪  torch.Tensor) -> torch.Tensor:
        return self.loss_fn(predicted_vector, gold_label)
```

Design Motivations. The architectural design of the FFNN was driven by the simplicity of the bag-of-words input representation and the need for computational efficiency. Since bag-of-words vectors discard word order and focus purely on word presence, a relatively shallow architecture was selected to avoid unnecessary complexity. A single hidden layer with ReLU activation was sufficient to learn meaningful patterns while enabling fast training and inference.

The training strategy—including the choice of loss function (`nn.CrossEntropyLoss`), optimizer (`Adam`), learning rate scheduler (`ReduceLROnPlateau`), dropout rate, and early stopping—was consistent across both models, and is further elaborated in the general implementation section above. These shared strategies reflect a design focus on training stability, efficient convergence, and robust generalization across different input representations.

Optimizer and Scheduler Code. The optimizer and learning rate scheduler are set up as follows:

```
from torch import optim
from torch.optim.lr_scheduler import ReduceLROnPlateau

optimizer = optim.Adam(model.parameters(), lr=3e-5, weight_decay=1e-6)
scheduler = ReduceLROnPlateau(optimizer, mode="max", min_lr=1e-6,
↪  factor=0.5, patience=3)
```

The optimizer used is `Adam`, with a learning rate of 3e-5 and weight decay for regularization. The learning rate scheduler `ReduceLROnPlateau` adjusts the learning rate when the validation performance plateaus. This helps improve convergence and stabilize training, especially when the model is not improving after several epochs. The patience parameter is set to 3, meaning the learning rate will be reduced if no improvement is observed for 3 consecutive validation steps.

RNN

The Recurrent Neural Network (RNN) was implemented to classify Yelp reviews into one of five sentiment categories, leveraging the sequential nature of the text. Unlike the Feedforward Neural Network (FFNN), the RNN processes the sequence of words in a review, capturing the temporal relationships between words. The model utilizes pre-trained word embeddings to initialize word vectors, enabling the network to leverage semantic information from the text.

Model Architecture. The RNN model processes reviews as sequences of word embeddings, which are passed through an RNN layer followed by a fully connected layer. The architecture consists of:

- **Input Layer**: The input layer takes word embeddings as input, with each review represented as a sequence of word vectors. The word embeddings are initialized from pre-trained embeddings.
- **RNN Layers**: The RNN layers are implemented by `nn.RNN(input_dim, hidden_dim, num_layers, batch_first=True, bidirectional=True)`. These layers processes the sequence of word embeddings and outputs a hidden state at each time step. The bidirectional nature allows the model to capture information from both past and future words in the sequence.
- **Hidden Layer**: After the RNN layers, the hidden states from both directions are concatenated to preserve information from both ends of the sequence. A `Dropout(0.3)` is applied to the concatenated hidden states to prevent overfitting.

- **Output Layer**: The output layer is implemented by `nn.Linear(hidden_dim * 2, output_dim)`, where `output_dim` is set to 5 (corresponding to the 5 sentiment classes).

Model Definition. The code for the RNN model is as follows:

```
import torch
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int, output_dim: int
    ↪  = 5) -> None:
        super().__init__()
        self.rnn = nn.RNN(input_dim, hidden_dim, batch_first=True,
        ↪  bidirectional=True)
        self.dropout = nn.Dropout(0.3)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.loss_fn = nn.CrossEntropyLoss()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        _, hidden = self.rnn(x)
        hidden_forward = hidden[-2]
        hidden_backward = hidden[-1]
        hidden_combined = torch.cat((hidden_forward, hidden_backward),
        ↪  dim=1)
        dropped = self.dropout(hidden_combined)
        return self.fc(dropped)

    def compute_loss(self, predictions: torch.Tensor, labels:
    ↪  torch.Tensor) -> torch.Tensor:
        return self.loss_fn(predictions, labels)
```

Design Motivations. The architectural design of the RNN was motivated by the need to capture the sequential dependencies inherent in the input data. Unlike the FFNN, which uses a bag-of-words representation, the RNN processes the review text in sequence, preserving the order of words. To leverage both past and future context, a bidirectional RNN was chosen, allowing the model to capture richer contextual information from both directions of the sequence.

The training strategy—including the choice of loss function (`nn.CrossEntropyLoss`), optimizer (`Adam`), learning rate scheduler (`ReduceLROnPlateau`), dropout rate, and early stopping—was consistent across both models, and is further elaborated in the general implementation section above. These shared strategies reflect a design focus on training stability, efficient convergence, and robust generalization across different input representations.

Optimizer and Scheduler Code. The optimizer and learning rate scheduler for the RNN are set up as follows:

```
from torch import optim
from torch.optim.lr_scheduler import ReduceLROnPlateau

optimizer = optim.Adam(model.parameters(), lr=3e-4, weight_decay=1e-6)
    scheduler = ReduceLROnPlateau(optimizer, mode="max", min_lr=1e-6,
    ↪  factor=0.5, patience=3)
```

The optimizer used is `Adam`, with a learning rate of 3e-4 and weight decay for regularization. The learning rate scheduler `ReduceLROnPlateau` is used to adjust the learning rate based on validation performance. This helps stabilize training and improve convergence, especially when there is no

improvement in performance for multiple epochs. The patience parameter is set to 3, meaning the learning rate will be reduced if no improvement is observed for 3 consecutive validation steps.

3 Results and Analysis

Evaluations

Model performance is evaluated primarily using **accuracy**, which measures the percentage of correct predictions. During training, we compute both training and validation accuracy, as well as training and validation loss after each epoch. To guide early stopping and learning rate adjustments more effectively, we use a combined accuracy metric that places greater emphasis on generalization:

$$\text{Combined Accuracy} = 0.95 \times \text{Validation Accuracy} + 0.05 \times \text{Training Accuracy}$$

This metric prioritizes performance on the validation set while still lightly accounting for training accuracy, helping to identify models that generalize well and avoid overfitting. Accuracy is calculated by taking the `argmax` of the predicted logits and comparing them to the ground truth labels.

Due to this, in practice, validation accuracy is the primary signal for both early stopping and the learning rate scheduler. Training loss is also tracked throughout to monitor convergence behavior and identify potential training instability.

Results

The performance of each model variant on the validation and test sets is summarized in the tables below. The following hyperparameters were kept constant across all models to isolate the effects of hidden layer count, learning rate, and scheduler factor:

- **Optimizer**: The optimizer used is `Adam`, with a variable learning rate (as specified in Table 3 and Table 3), and a weight decay of **1e-6** for regularization.
- **Scheduler**: `ReduceLROnPlateau`, with a reduction factor of **0.5**, a minimum learning rate of **1e-6**, and a patience of **3 epochs**.
- **Dropout Rate**: **0.3** for both **FFNN** and **RNN** models.
- **Early Stopping Patience**: **7 epochs**, based on the combined accuracy metric, as described earlier.

These settings were kept the same for all models to isolate the effect of hidden layer count, learning rate, and scheduler factor.

Model	Hidden Layers	Hidden Dim	Learn. Rate	Train Acc.	Val. Acc.
FFNN	1	16	3e-4	0.7568	0.5850
FFNN	1	16	3e-5	0.7473	0.5813
FFNN	1	32	3e-4	0.7498	0.5837
FFNN	1	32	3e-5	0.8743	0.5700
FFNN	1	64	3e-4	0.8037	0.5700
FFNN	1	64	3e-5	0.8754	0.5725
FFNN	1	128	3e-4	0.7223	0.5813
FFNN	1	128	3e-5	0.8532	0.5750
FFNN	1	256	3e-4	0.7492	0.5563
FFNN	1	256	3e-5	0.9539	0.5625
FFNN	2	16	3e-4	0.7547	0.5913
FFNN	2	16	3e-5	0.7240	0.5825
FFNN	2	32	3e-4	0.7840	0.5713
FFNN	2	32	3e-5	0.7728	0.5813
FFNN	2	64	3e-4	0.8610	0.5663
FFNN	2	64	3e-5	0.7616	0.5825

Model	Hidden Layers	Hidden Dim	Learn. Rate	Train Acc.	Val. Acc.
FFNN	2	128	3e-4	0.9178	0.5513
FFNN	2	128	3e-5	0.8687	0.5813
FFNN	2	256	3e-4	0.7208	0.5650
FFNN	2	256	3e-5	0.7208	0.5650

Table 2: Validation and test accuracy for **FFNN** models. **Hidden Layers** refers to the number of layers in the model and **Factor** is the learning rate scheduler reduction factor.

Model	Hidden Layers	Hidden Dim	Learn. Rate	Train Acc.	Val. Acc.
RNN	1	16	3e-4	0.3847	0.4537
RNN	1	16	3e-5	0.1983	0.2575
RNN	1	32	3e-4	0.4454	0.5162
RNN	1	32	3e-5	0.2116	0.3400
RNN	1	64	3e-4	0.4281	0.5325
RNN	1	64	3e-5	0.3612	0.4600
RNN	1	128	3e-4	0.4246	0.5038
RNN	1	128	3e-5	0.3616	0.4487
RNN	1	256	3e-4	0.3162	0.4537
RNN	1	256	3e-5	0.3694	0.4500
RNN	2	16	3e-4	0.4017	0.4988
RNN	2	16	3e-5	0.3479	0.3762
RNN	2	32	3e-4	0.4760	0.5487
RNN	2	32	3e-5	0.3407	0.4300
RNN	2	64	3e-4	0.4011	0.5175
RNN	2	64	3e-5	0.3609	0.4625
RNN	2	128	3e-4	0.3474	0.4400
RNN	2	128	3e-5	0.3480	0.4650
RNN	2	256	3e-4	0.3217	0.4587
RNN	2	256	3e-5	0.4078	0.5025

Table 3: Validation and test accuracy for **RNN** models. **Hidden Layers** refers to the number of layers in the model and **Factor** is the learning rate scheduler reduction factor.

Observations

From the results presented in Tables 3 and 3, several trends and insights can be observed regarding the impact of hidden layer count, hidden dimension size, and learning rate on model performance.

Effect of Hidden Layers. For both FFNN and RNN models, increasing the number of hidden layers from 1 to 2 showed marginal improvements in validation accuracy. The best performing FFNN configuration used 2 hidden layers with 16 hidden units, achieving a validation accuracy of 0.5913. Similarly, the best RNN model also had 2 hidden layers with 32 hidden units, reaching a validation accuracy of 0.5487. However, adding more layers did not consistently improve performance and in some cases led to overfitting or instability during training.

Effect of Hidden Dimension. Larger hidden dimensions generally increased training accuracy, indicating higher capacity. However, validation accuracy did not always improve, and in some FFNN configurations (e.g., 256 hidden units), a clear overfitting pattern emerged: models achieved high training accuracy (up to 0.95) but much lower validation accuracy (as low as 0.5563). This suggests that excessively large hidden dimensions may hurt generalization, especially when not accompanied by adequate regularization.

Effect of Learning Rate. Learning rate had a substantial impact on model performance. In both FFNN and RNN models, a higher learning rate (3e-4) typically led to better validation accuracy, while lower learning rates (3e-5) often resulted in underfitting. Notably, RNN models trained with a learning rate of 3e-5 consistently underperformed across all configurations, showing

low training and validation accuracy. In contrast, FFNNs managed to achieve high training accuracy at lower learning rates, but without consistent improvements in validation accuracy.

Comparison of FFNN and RNN Models. Overall, FFNN models outperformed RNN models in terms of maximum validation accuracy. The top FFNN configuration achieved 0.5913 validation accuracy, while the best RNN reached only 0.5487. FFNNs also converged more reliably and exhibited higher training accuracy across the board. RNNs showed more difficulty in training, potentially due to their sequential nature and increased complexity. Moreover, FFNNs appeared more prone to overfitting with larger hidden sizes, whereas RNNs often suffered from underfitting, especially at low learning rates.

These observations suggest that, for this task and dataset, FFNN models provide a more effective and stable architecture under the tested hyperparameters.

4 Analysis

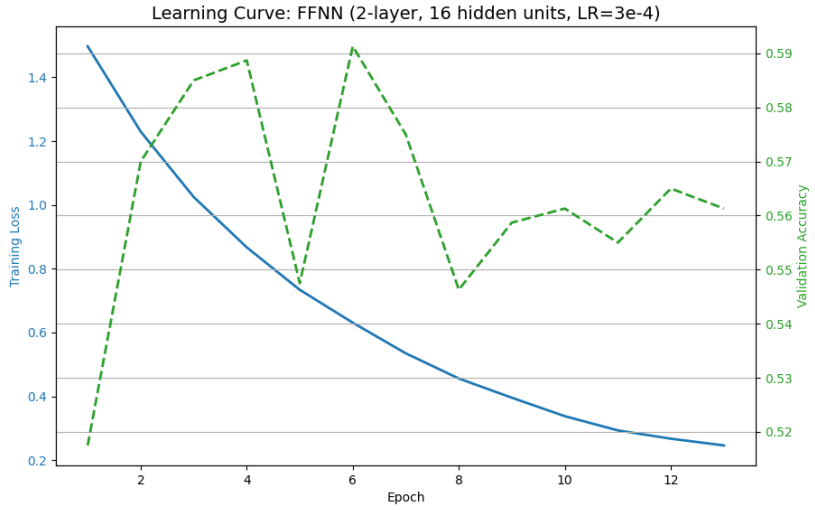

Figure 1: Learning curve for FFNN (2-layer, 16 hidden units, LR = 3e-4).

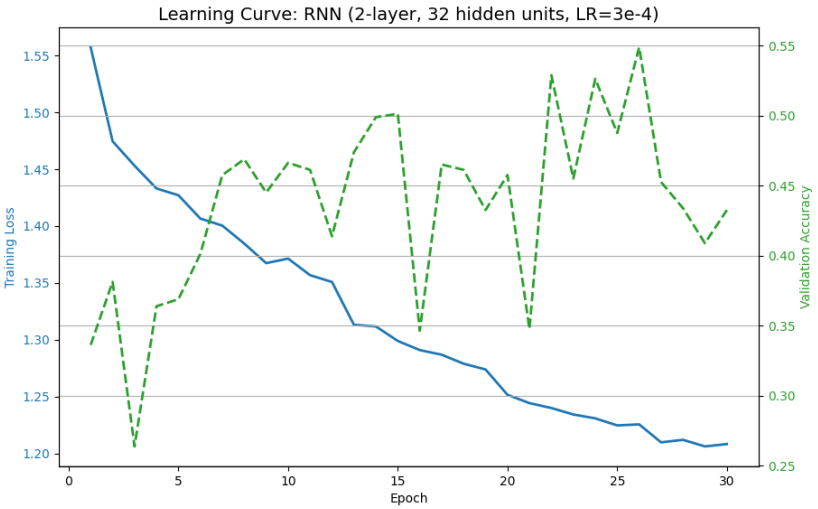

Figure 2: Learning curve for RNN (2-layer, 32 hidden units, LR = 3e-4).

Learning Curve

The learning curve of our best-performing models shows distinct behaviors for the FFNN and the RNN. For the FFNN (with 2 hidden layers and 16 hidden units, LR = 3e-4), the training loss steadily decreases, and the validation accuracy plateaus around epoch 10, after which early stopping was triggered. This indicates stable training and good generalization without overfitting.

On the other hand, the RNN (with 2 layers and 32 hidden units, LR = 3e-4) exhibits a more erratic learning curve, with some fluctuations in both training loss and validation accuracy. These fluctuations suggest that the RNN had some instability during training, which may have contributed to lower validation accuracy, but it still showed improvement over time, indicating the model was learning meaningful patterns despite the noise.

Error Analysis

One common error observed was the misclassification of ambiguous or out-of-distribution inputs, especially for rare or noisy tokens. RNNs tended to make more inconsistent predictions, likely due to unstable training. In contrast, FFNNs handled consistent patterns better but struggled with long-range dependencies. Improving the feature representation (e.g., using pre-trained embeddings) or applying more regularization could help reduce such errors.

5 Conclusion and Feedback

Conclusion

Our experiments demonstrated that FFNN models outperformed RNNs on this classification task, achieving higher and more consistent validation accuracy. In particular, FFNNs with two hidden layers and smaller hidden dimensions (e.g., 16 or 32) combined with a learning rate of 3e-4 yielded the best results. RNN models showed lower performance and were more sensitive to hyperparameter choices, often struggling with underfitting. Overall, FFNNs provided a more reliable architecture under the tested conditions.

Feedback

The assignment was engaging and helped reinforce our understanding of model training, tuning, and evaluation. However, it was quite time-consuming, taking approximately 20–25 hours in total. The tuning process—particularly for RNNs—was challenging due to long training times and unstable convergence. It might be helpful to provide a clearer guide or tighter constraints on the hyperparameter search space. Overall, though, the assignment was valuable and realistic, offering practical insight into how models behave under different settings.