Deep Learning models for Sentence Classification, Assingment 1, course CE7455

Adam Barla
n2308836j@e.ntu.edu.sg
NTU, Singapore

1 Configuration Optimization

1.1

Implement the `pack_padded_sequence` function in PyTorch's RNN library. Report results under the default setting and discuss the benefits of this function.

The function `pack_padded_sequence` takes input sequences of varying lengths and packs them into a compact form, before they are fed into the RNN. By using pack_padded_sequence, the model can skip the padded areas, focusing only on the actual data, which can lead to more accurate and faster training.

I created a class `PackedRNN` that used this function. Then, I conducted 10 training runs consisting of 100 epochs for the model with and without the `pack_padded_sequence`. I compared the models on validation accuracy so the test accuracy can still serve as an indicator of the performance on unseen data in the end. The average validation accuracy of the baseline model was 67.17 % while the model with `pack_padded_sequence` reached 68.53 %, which is a minor improvement. The average duration of one epoch went up by ~3.5 seconds from 17.5s to 21s. This can be improved by packing the sequences only once and not every time the `forward` method is called.

1.2

Experiment with different configurations (optimizers, learning rates, batch sizes, sizes of hidden embedding) and report the best configuration's performance on the validation and test sets.

For this purpose, I decomposed the notebook into a project. Using `hydra` and `wandb` I created a sweep over hyperparameters. I chose to test using a grid search over these parameters based on some empirical test runs:

```
lr:
  values: [ 0.00001, 0.0001, 0.001, 0.01, 0.1 ]
batch_size:
  values: [ 32, 64, 128 ]
optimizer:
  values: [ sgd, adam, adadelta, adagrad, rmsprop ]
model.hidden_dim:
  values: [ 50, 100, 200, 400 ]
```

I tested each combination of parameters with the `PackedRNN` class from Section 1.1 for 100 epochs. The best performance on validation accuracy was achieved when using these parameters:

```
hidden_dim: 200
optimizer: adadelta
batch_size: 32
lr: 0.1
```

The model with these parameters reached validation and test accuracy 77.71% and 83.8% respectively.

1.2.1 Parameter interaction analysis

Figure 1 shows an average validation accuracy when two parameters are kept at a specific value. We can observe trends that help us better understand which hyperparameters are optimal. For example, we can see a general correlation between hidden embedding size and validation accuracy, which is not surprising. Adam and RMSprop optimizers seem to generally dominate, even

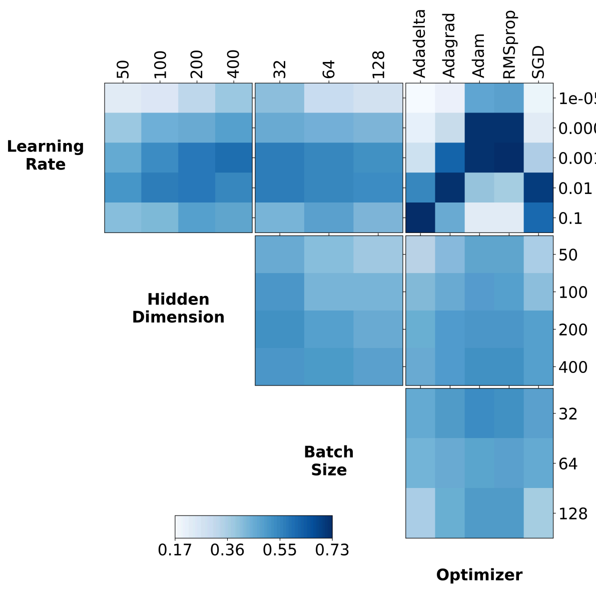

Figure 1: Each square represents a mean validation accuracy of runs with two parameters set to a particular value.

though Adadelta reached the highest validation accuracy. Each optimizer has a learning rate at which it works best as it seems. The trend towards smaller batch sizes seems curious. Given more time I would explore it further.

For the next experiments, I will use Adam with a hidden dimension of 200, and a batch size of 32. The learning rate may vary in the future but a good value for this configuration seems 0.001.

1.3

Implement regularization techniques, describe them, and report accuracy results after application.

1.3.1 Dropout

During training, randomly zeroes some of the elements of the input tensor with probability p given by a parameter, which greatly reduces overfitting. This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the [1]. Dropout has the effect of training and using an ensemble of models and promotes learning of a sparse representation.

1.3.2 L1/2 Regularization

We can include a regularization parameter to the loss, which is computed based on the parameters of the model.
For L1 it is

$$\alpha \sum_{\omega \in \Omega} |\omega|$$

and for L2 it is

$$\alpha \sum_{\omega \in \Omega} \omega^2$$

where Ω is a set of all parameters of the model and α is the weight of the regularization parameter.

This regularization results in sparsity.

1.3.3 Gradient clipping

To counteract exploding gradient we can employ gradient clipping. It scales the gradient g down if its norm $\|g\|$ is larger than some threshold t.

$$\text{if } \|g\| \geq t : g \leftarrow t * \left(\frac{g}{\|g\|}\right)$$

For this I used the function `torch.nn.utils.clip_grad_norm`.

1.3.4 Early stopping

Up to a point, training improves the learner's performance on the validation set. Past that point, however, improving on the training data comes at the expense of increased generalization error.

I implemented early stopping with patience. If validation accuracy doesn't improve in `patience` epochs, the run is terminated. By using patience we can increase the number of epochs to a large value because models stop by themselves.

1.3.5 Batch normalization

Batch normalization makes the training of NNs faster and more stable through the normalization of the layers' inputs by re-centring and re-scaling. It can mitigate the problem of internal covariate shift, where parameter initialization and changes in the distribution of the inputs of each layer affect the learning rate of the network.

It isn't common to use batch norm with RNNs but it can be used later with more complex classifiers in later sections.

1.3.6 Results of regularization

I conducting another sweep over the new hyperparameters. I kept the previously found parameters and I additionally set `patience` to 10, number of epochs to 500 and regularization to l2. I picked l2 over l1 by conducting smaller sweeps, where l2 always outperformed l1. I searched over these parameters:

```
lr:
    min: 0.00001
    max: 0.001
dropout:
    min: 0.0
    max: 0.5
regularizer.alpha:
    min: 0.00001
    max: 0.01
grad_clip_threshold:
    min: 1.0
    max: 10.0
```

I searched over the learning rate because regularization techniques might result in a different optimal learning rate. The best parameters found were:

```
regularizer.alpha: 0.0006332825849228081
lr: 0.0008726644867850513
grad_clip_threshold: 3.409241906075233
dropout: 0.014878368933965491
```

and the model with these parameters achieved validation and test accuracy 86.88% and 89.2% respectively, which is an improvement.

2 Input Embedding

Switch from randomly initialized input word embeddings to pre-trained word2vec embeddings. Report accuracy on the validation set and compare performance.

- You can refer to the here on how to install gensim to to work with word2vec. [2]
- Pretrained word2vec models can be downloaded following link. Use `word2vec-google-news-300` as the pre-trained word2vec embeddings.

I created a new class `GensimPackedRNN`. The `nn.Embedding` in this class is initialized through the function `from_pretrained`. This function requires an matrix of vocab_size × embedding_size. I created it for the vocabulary used before by iterating through the words and finding a corresponding embedding in the `word2vec-google-news-300`. For words that are not in the vocabulary of the word2vec model I just insert an empty vector as an embedding. In total, 419 out of 7687 weren't found. These include for example `<unk>` and `<pad>` tokens, symbols such as `?` and `-`, numbers, names and complex words such as `hendecasyllabic`.

The model, with same hyperparameters as in Section 1.3.6, reached accuracy of 83.76% and 86.6% on validation and test data, which is a slight decrease that may have been caused by the missing embeddings.

3 Output Embedding

Explore options for computing sentence embedding beyond the final hidden representation. Implement the best option(s) and report accuracy on the validation set, comparing it to the performance in Task 2.

3.1 Average of Word Embeddings

One of the simplest methods is to take the average of the word embeddings of all words in the sentence. This method is easy to implement but does not take into account the order of the words in the sentence.

3.2 Max/Average Pooling of hidden states

Max or average pooling operations can be applied to the hidden states of an RNN to create a sentence embedding. Max pooling captures the most important features, while average pooling captures the average features of the hidden states.

3.3 Self-Attention

To calculate the sentence embedding we can compute a weighted sum of all hidden states in the sentence. The weights are learned through self-attention during training and determine the importance of each word to the meaning of the sentence.

I opted to use this technique. I implemented `AttentionGensimPackedRNN` class that uses multiheaded attention:

```python
def forward(self, text, text_lengths):
  ...
  packed_output, _ = self.rnn(packed_input)
  padded_output, _ = pad_packed_sequence(packed_output)
  attention_output, _ = self.attention(
    padded_output, padded_output, padded_output
  )
  sentence_embedding = torch.mean(
    attention_output, dim=0
  )
  ...
```

Maintaining hyperparameters from before and using only 1 attention head the model reached accuracy of 85.32% and 87.2% on validation and test data, which is a slight increase compared to model form Section 2, but it still worse than model from Section 1.3.6. This could be possibly improved with further hyperparameter tuning.

4 Architecture Optimization

The original code base uses the most simple RNN architecture. Now consider more complex architectures in the following.

- GRU with a single hidden layer
- LSTM with a single hidden layer
- Bidirectional simple RNN with a single hidden layer
- Simple RNN with 2 hidden layers

Run experiments by replacing the original RNN with each of the following architectures. Report the accuracy of the validation set and compare these results. Discuss your findings.

I implemented changing of base method through Hydra configs so the new configuration could be run by this command:

```
python3.10 -m training model=attention_rnn base=rnn
base.bidirectional=true base.num_layers=1
```

the possible options for `base` are `rnn`, `gru` and `lstm`. My results were as follows:

model	**validation accuracy**	**test accuracy**
GRU	85.688%	87.8%
LSTM	86.514%	88%
RNN two layers	**86.605%**	88.4%
RNN bidirectional	86.330%	**90%**

The highest test accuracy was achieved by the bidirectional RNN, which could be attributed to better architecture or just to the higher (double) dimension of the RNN output caused by the concatenation of forward and backward outputs. The highest validation accuracy was achieved by the RNN with two layers. Both these results could suggest that increasing the model size leads to better outcomes. Another possibility is that these results for options that use RNN are good because the hyperparameters were selected using the RNN. For a more fair comparison, we would have to fine-tune all models.

5 Critical Thinking

Based on the best setting and architecture you find from all the above 4 tasks, describe one more modification (other than the above optimization options) you can think of that could potentially further improve the performance. Conduct experiments, report the accuracy on the validation set and discuss your findings. (Hint: additional component to the model, integration with different models, etc.)

I creadted a new class `FC` representing a fully connected classifier. I then conducted experiments to determine optimal number of layers and their hidden dimensions. I used the one directional RNN with two layers that proved efective in Section 4. Unfortunatelly I wasn't able to achieve better performance then before. The best configuration I found was

```
classifier.hidden_dims: [200,200]
dropout:0.061442548711177714
lr:0.000832086482041085
```

and it reached 89.8% test accuracy and 86.88% validation accuracy. I tuned the learning rate togetehr with dropout. Dropout was used in the new classifier together with batch norm so I found it importatnt to tune it.

References

[1] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *CoRR*, 2012, [Online]. Available: http://arxiv.org/abs/1207.0580

[2] R. Řehůřek and P. Sojka, “Software Framework for Topic Modelling with Large Corpora,” in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, Valletta, Malta: ELRA, May 2010, pp. 45–50. [Online]. Available: http://is.muni.cz/publication/884893/en