Emotion Recognition using Convolutional Neural Networks, Convoluted Feelings

Abstract
The goal of this exercise was to Understand Convolutional Neural Networks (CNNs) and deepen our understanding of the role of hyperparameters, regularisation, and handling class imbalance. In addition we should be able to implement CNNs in PyTorch (or PyTorch Lightening) and perform an error analysis of machine learning models. In this report we present the implementation and results of our CNN in the context of emotion recognition. We used data from Twitter. After preprocessing the data and initializing our model, we trained eight models using different hyper-parameters to evaluate which is the best combination. To validate our results, the combination resulting in the highest accuracy (model 3 with a test accuracy of 0.9148), was run on the second data set, which resulted in an accuracy of 0.8443.

Keywords
CNN — Emotion Recognition — Natural Language Processing

Contents

1. Introduction

This is our report for the third assignment of the Machine Learning for Natural Language Processing 1 course. The task was to train ... The goal of this exercise is to:

- Understand CNNs.
- Be able to implement CNNs in PyTorch or PyTorch Lightning.
- Deepen your understanding of the role of hyper-parameters, regularisation, and handling class imbalance.
- Perform an error analysis of machine learning models.

2. Data & Task Description

This section discusses the data we worked with and the task we completed. First the data is discussed and then the task is elaborated.

2.1 Data

For this exercise, you will be working on the Twitter Emotion Recognition task. The goal of this task is to infer the affectual state of a person from their tweet. You will be using the Tweeteval dataset to train your model. You can access the related GitHub repositories from the links below:

- Tweeteval Repository:
  https://github.com/cardiffnlp/tweeteval/tree/main/datasets/emotion
- Emotion Detection (our task):
  https://github.com/cardiffnlp/tweeteval/tree/main/datasets/emotion

The above URL can be used to load the data directly in the Colab notebook - as seen in Exercise 1 and 2.

2.2 Emotion Recognition with a CNN

Implement an emotion recognition classifier in PyTorch or PyTorch Lightning. We suggest reusing and adjusting the class structure from exercise 2 (which may be inspired by Rao and McMahan). However, you are free to create your own, new class structure. Keep in mind that for emotion prediction we work on the word level instead of character level in Exercise 1. Thus, your Vocabulary class (that is, if you have one) will not hold a vocabulary of characters. Exercise 3 will cover binary text classification with CNNs, so your model will predict between two classes at most. Remember to document your code with docstrings and/or comments and/or text cells.

1. Pick two different emotion classes for your model to predict (e.g. anger and joy). Load/filter your dataset to include only the related class data. Create another dataset and change only one of the classes (e.g. anger and sadness) this time.

2. Your goal is to find the optimal model architecture and training regime for your CNN classifier. Pick one of the datasets you created and start experimenting. Try out at least three different combinations of the following hyper-parameters, which you consider well-performing, and report the combinations and corresponding results (accuracy and F1 -macro) on the development set in a table: a) optimizer, b) learning rate, c) dropout, d) # of filters, e) different strides, f) different kernel sizes, g) different pooling strategies, h) batch sizes, i) any other thing you want to test

3. Use your best performing model settings to train another model on the second dataset this time. Report the model performance (accuracy and F1-macro) on the test-set of both datasets.

4. Reason about the observed effects of your 3 best hyperparameter settings and dataset variations on model performance. You do not need to be sure that the reasons you provide are correct – the goal is to provide educated (or well- reasoned) guesses.

Keep track of the loss to interrupt early if a model does not converged.

3. Data Preprocessing

The Emotion Detection data set contains several .txt files. As they are clearly labeled it is easy to know which set: 1) training, 2) testing and 3) validation, they belong to and what kind of data they contain. The text files contain numerous sentences composed of different words, special symbols, punctuation, and emojis, whereas the label files contain several labels that correspond to different types of emotions. The mapping file tells us the specific meaning of different emotion labels.

The data is encapsulated in a loader class. This class is used to load the data from the provided link (see chapter 2.1) and preprocess it according to the steps that follow below. The 'TextVector' class is then composed into a dataset class, which inherits from PyTorch Dataset. The dataset class was then composed into a dataloader class, which allowed for batched training.

The following steps were taken to preprpocess the data:

1. **Remove all HTML tags.**
2. **Convert all characters to lowercase.**
3. **All emojis were converted to their textual meaning.** This decision was made because emojis reflect what an writer is feeling in a moment. While they are irrelevant for tasks like language recognition in the context of emotion recognition they play a very important role.
4. **Delete all numbers.** To explain why we made this choice one can consider the following example: The number 2022 can be used in many different ways, each with a different meaning. For example the questions: "What is the result of 4044/2?" and "In which year are we right now?" result in the same answer, but are completely different from a contextual point of view. Word embeddings have great difficulty with such ambiguity. There are two ways to deal with the problem: Either one simply deletes the numbers, or one can replace them with a tag (e.g. [NUMBER])). We have chosen the first option!
5. **Remove all punctuation.**

Then we need to convert the processed texts to vectors that represent the different words. To achieve this, we apply the following steps:

1. **Tokenize:** We use the function "tokenize" to tokenize our sentences in our training and validation datasets, in order to build a vocabulary and find the maximum sentence length. The function "encode" takes in the outputs of tokenize, performs sentence padding and stores input ids as a numpy array in the class.
2. **Load Pretrained Vectors:** We load the pre-trained vectors for each token in our vocabulary. For tokens without pre-trained vectors, we initialize random word vectors with identical length and variance.
3. **Create PyTorch DataLoader for Training, Validation and Testing Datasets:** We create an iterator for our datasets using the Torch DataLoader class. This will help us minimize memory-use during training and boost the training speed.

4. Network Structure

As recommended in the instructions, we applied an object oriented approach. For each data set, we used a custom loader class, which automatically downloads and reads the data and then applies the preprocessing steps discussed in chapter 3. This loader class is then composed into dataset class, which inherits from PyTorch's Dataset module. The class was then converted to a DataLoader class, which makes it easier to train the model in batches. The classes both used tensor representations to store the data. This comes with the advantage of CUDA acceleration.
The training loop is packed into a class called ModelTrainer. This class makes it easy to train our models and avoid code duplication. Here are the references to the notebook and the article, we used to build our notebook. The aforementioned resources were essential in enabling us to complete our task.

5. Theory - Convolutional Neural Networks

Suppose that we are classifying a sentence which has N=7 tokens and the dimensionality of word vectors is d=5. After applying the embedding layer on the input token ids, the sample sentence is represented as a 2D tensor with shape (7, 5) like an image where $x_{emb} \in R^{7*5}$.
We then use a 1-dimesional convolution to extract the features

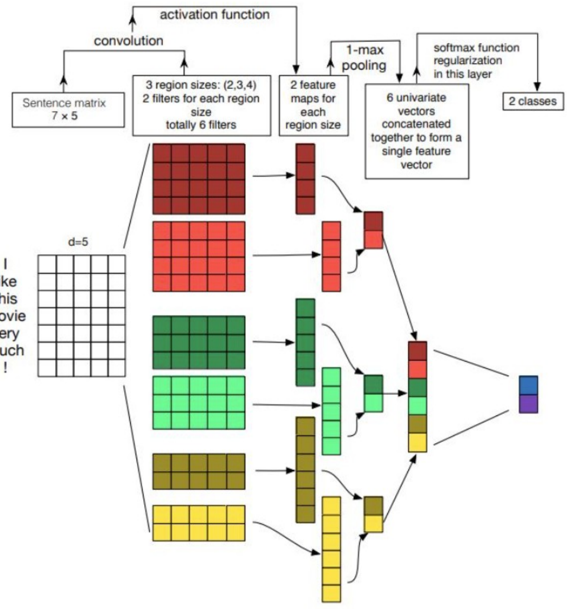

activation function
convolution
1-max pooling
softmax function regularization in this layer
Sentence matrix 7 × 5
3 region sizes: (2,3,4) 2 filters for each region size totally 6 filters
2 feature maps for each region size
6 univariate vectors concatenated together to form a single feature vector
2 classes
d=5
I like this movie very much !

Figure 1. Example where N=7 and d=5, using a 2D tensor with the shape (7,5).

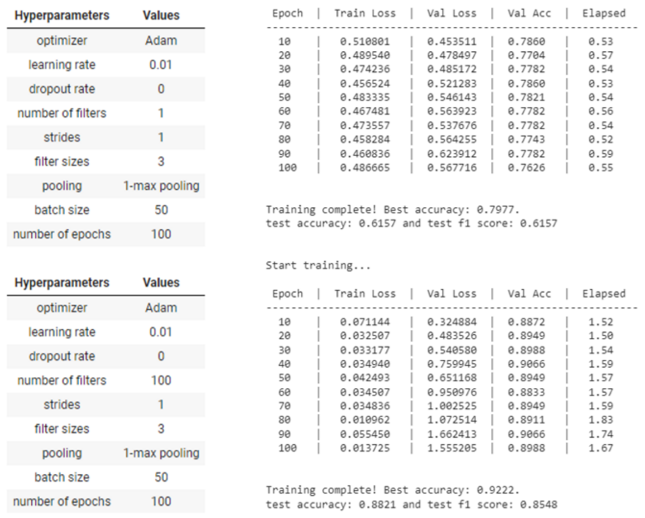
from the sentence. Each filter will then scan over x_{emb} and return a feature map: $x_{conv_i} = Conv1D(x_{emb})$
Next, we apply the ReLU activation to x_{conv_i} and use a pooling strategy to reduce each feature map to a single scalar value. Then we concatenate these scalar values to a vector which will be fed to a fully connected layer to compute the final scores of our classes.
$x_{pool_i} = Pool(ReLU(x_{conv_i}))$
$X_{fc} = concat(x_{pool_i})$
Finally, we use a fully connected layer with the weight matrix W_{fc} and dropout to compute $logits$, which is a vector of length 2 that keeps the scores for our two classes.
$logits = Dropout(W_{fc}x_{fc})$
For this task, we tried to simplify the model in training, by just using one convolutional layer to deal with the task.

6. Model Training

The two data sets we generated contain two pairs of emotions. The first emotion- pair is: **(anger, joy)** and the second is **(anger, optimism)**. As mentioned, we simplified our model in training, by just using one convolutional layer to deal with the task. During each training, we choose the parameters which get the highest accuracy on validation data as the parameter of the best model, in order to avoid over-fitting. And between models, we choose the model which gets the highest accuracy and f1-score as the best model to train the new dataset.

7. Results

The three Figures 2, 3 and 4 show the hyper-parameter combinations we used and the results we obtained. We achieved the best performance using the third hyper- parameter combination. Using the same hyper-parameters on the second dataset, we got accuracy levels of 0.8443 and an F1-score of 0.2933, which can be seen in Figure 5.



Hyperparameters	Values
optimizer	Adam
learning rate	0.01
dropout rate	0
number of filters	1
strides	1
filter sizes	3
pooling	1-max pooling
batch size	50
number of epochs	100

```
 Epoch  |  Train Loss  |  Val Loss  |  Val Acc  |  Elapsed
------------------------------------------------------------
  10    |   0.510801   |  0.453511  |  0.7860   |   0.53
  20    |   0.489540   |  0.478497  |  0.7704   |   0.57
  30    |   0.474236   |  0.485172  |  0.7782   |   0.54
  40    |   0.456524   |  0.521283  |  0.7860   |   0.53
  50    |   0.483335   |  0.546143  |  0.7821   |   0.54
  60    |   0.467481   |  0.563923  |  0.7782   |   0.56
  70    |   0.473557   |  0.537676  |  0.7782   |   0.54
  80    |   0.458284   |  0.564255  |  0.7743   |   0.52
  90    |   0.460836   |  0.623912  |  0.7782   |   0.59
  100   |   0.486665   |  0.567716  |  0.7626   |   0.55


Training complete! Best accuracy: 0.7977.
test accuracy: 0.6157 and test f1 score: 0.6157


Start training...

 Epoch  |  Train Loss  |  Val Loss  |  Val Acc  |  Elapsed
------------------------------------------------------------
  10    |   0.071144   |  0.324884  |  0.8872   |   1.52
  20    |   0.032507   |  0.483526  |  0.8949   |   1.50
  30    |   0.033177   |  0.540580  |  0.8988   |   1.54
  40    |   0.034940   |  0.759945  |  0.9066   |   1.59
  50    |   0.042493   |  0.651168  |  0.8949   |   1.57
  60    |   0.034507   |  0.950976  |  0.8833   |   1.57
  70    |   0.034836   |  1.002525  |  0.8949   |   1.59
  80    |   0.010962   |  1.072514  |  0.8911   |   1.83
  90    |   0.055450   |  1.662413  |  0.9066   |   1.74
  100   |   0.013725   |  1.555205  |  0.8988   |   1.67


Training complete! Best accuracy: 0.9222.
test accuracy: 0.8821 and test f1 score: 0.8548
```

Hyperparameters	Values
optimizer	Adam
learning rate	0.01
dropout rate	0
number of filters	100
strides	1
filter sizes	3
pooling	1-max pooling
batch size	50
number of epochs	100

Figure 2. The hyper-parameter combinations number 1 and 2 and their results.

8. Discussion

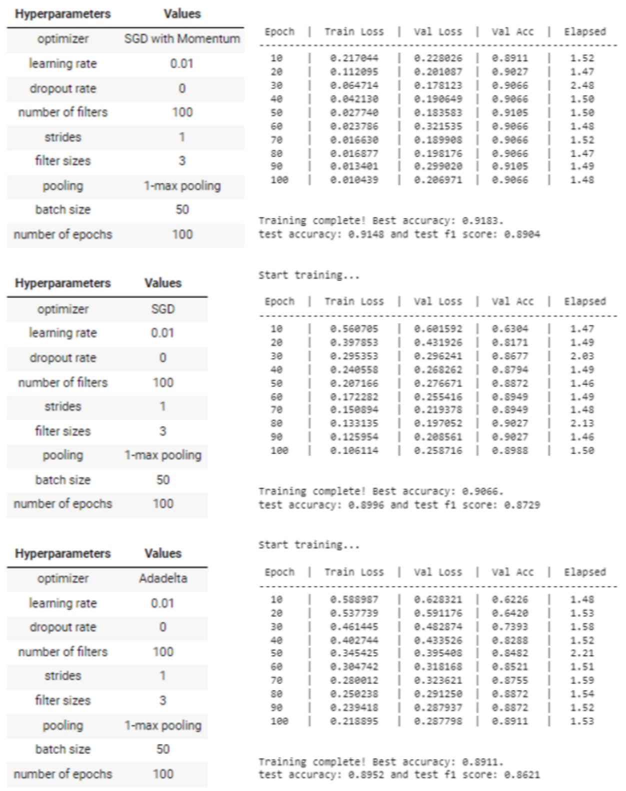
As mentioned, the different combinations turned out to exhibit different levels of accuracy. We determined the main reasons behind the third combination achieving the best performance to be the following:

1. When the stride equals 1 the network can consider the relations between words better, as the model may miss important relations as the value gets larger.
2. The filter size of the convolutional layer seems to fit well for this task. As it gets larger the model will consider the embedding of words with larger size, which may give more weight to useless information.
3. The number of filters is just enough to make sure the model is not under-fitted.

When reviewing the results of the performance of our model on the second data set, we noticed that the F1 score is really low. This could be due to the volume of different labels being severely unbalanced. For example only 294 out of 1694 samples are labeled ”optimism” in the training dataset.

9. Conclusion

The different hyper-parameter combinations delivered different results. We chose the most accurate performing combination (number 3) as the ”best” and tested it on a second data set to validate our choice. Our evaluation resulted in a low F1 score, which we attributed to the dataset being unbalanced. In future it may be wise to perform different up-sampling



Hyperparameters	Values
optimizer	SGD with Momentum
learning rate	0.01
dropout rate	0
number of filters	100
strides	1
filter sizes	3
pooling	1-max pooling
batch size	50
number of epochs	100

```
Epoch  |  Train Loss  |  Val Loss  |  Val Acc  |  Elapsed
------------------------------------------------------------
 10    |   0.217044   |  0.228026  |  0.8911   |   1.52
 20    |   0.112095   |  0.201087  |  0.9027   |   1.47
 30    |   0.064714   |  0.178123  |  0.9066   |   2.48
 40    |   0.042130   |  0.190649  |  0.9066   |   1.50
 50    |   0.027740   |  0.183583  |  0.9105   |   1.50
 60    |   0.023786   |  0.321535  |  0.9066   |   1.48
 70    |   0.016630   |  0.189908  |  0.9066   |   1.52
 80    |   0.016877   |  0.198176  |  0.9066   |   1.47
 90    |   0.013401   |  0.299020  |  0.9105   |   1.49
 100   |   0.010439   |  0.206971  |  0.9066   |   1.48


Training complete! Best accuracy: 0.9183.
test accuracy: 0.9148 and test f1 score: 0.8904
```

Hyperparameters	Values
optimizer	SGD
learning rate	0.01
dropout rate	0
number of filters	100
strides	1
filter sizes	3
pooling	1-max pooling
batch size	50
number of epochs	100

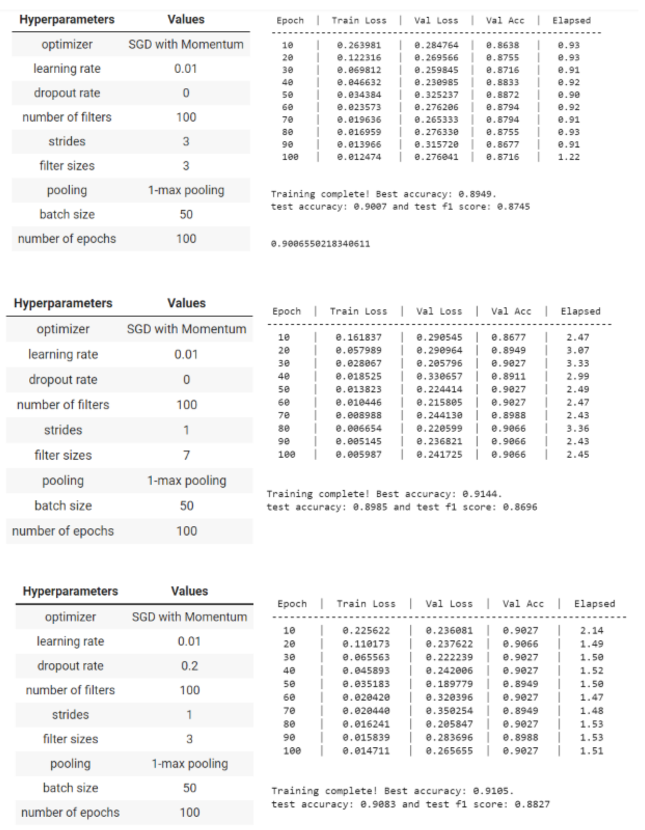
```
Start training...

Epoch  |  Train Loss  |  Val Loss  |  Val Acc  |  Elapsed
------------------------------------------------------------
 10    |   0.560705   |  0.601592  |  0.6304   |   1.47
 20    |   0.397853   |  0.431926  |  0.8171   |   1.49
 30    |   0.295353   |  0.296241  |  0.8677   |   2.03
 40    |   0.240558   |  0.268262  |  0.8794   |   1.49
 50    |   0.207166   |  0.276671  |  0.8872   |   1.46
 60    |   0.172282   |  0.255416  |  0.8949   |   1.49
 70    |   0.150894   |  0.219378  |  0.8949   |   1.48
 80    |   0.133135   |  0.197052  |  0.9027   |   2.13
 90    |   0.125954   |  0.208561  |  0.9027   |   1.46
 100   |   0.106114   |  0.258716  |  0.8988   |   1.50


Training complete! Best accuracy: 0.9066.
test accuracy: 0.8996 and test f1 score: 0.8729
```

Hyperparameters	Values
optimizer	Adadelta
learning rate	0.01
dropout rate	0
number of filters	100
strides	1
filter sizes	3
pooling	1-max pooling
batch size	50
number of epochs	100

```
Start training...

Epoch  |  Train Loss  |  Val Loss  |  Val Acc  |  Elapsed
------------------------------------------------------------
 10    |   0.588987   |  0.628321  |  0.6226   |   1.48
 20    |   0.537739   |  0.591176  |  0.6420   |   1.53
 30    |   0.461445   |  0.482874  |  0.7393   |   1.58
 40    |   0.402744   |  0.433526  |  0.8288   |   1.52
 50    |   0.345425   |  0.395408  |  0.8482   |   2.21
 60    |   0.304742   |  0.318168  |  0.8521   |   1.51
 70    |   0.280012   |  0.323621  |  0.8755   |   1.59
 80    |   0.250238   |  0.291250  |  0.8872   |   1.54
 90    |   0.239418   |  0.287937  |  0.8872   |   1.52
 100   |   0.218895   |  0.287798  |  0.8911   |   1.53


Training complete! Best accuracy: 0.8911.
test accuracy: 0.8952 and test f1 score: 0.8621
```

Figure 3. The hyper-parameter combinations number 3, 4 and 5 and their results.

strategies on certain labels that are underrepresented. Overall the project was successful and all tasks have been completed. Further we report the successful implementation of a CNN in PyTorch and the importance of choosing the right hyper-parameter combination has become clear.



Hyperparameters	Values
optimizer	SGD with Momentum
learning rate	0.01
dropout rate	0
number of filters	100
strides	3
filter sizes	3
pooling	1-max pooling
batch size	50
number of epochs	100

```
Epoch  |  Train Loss  |  Val Loss  |  Val Acc  |  Elapsed
------------------------------------------------------------
 10    |   0.263981   |  0.284764  |  0.8638   |   0.93
 20    |   0.122316   |  0.269566  |  0.8755   |   0.93
 30    |   0.069812   |  0.259845  |  0.8716   |   0.91
 40    |   0.046632   |  0.230985  |  0.8833   |   0.92
 50    |   0.034384   |  0.325237  |  0.8872   |   0.90
 60    |   0.023573   |  0.276206  |  0.8794   |   0.92
 70    |   0.019636   |  0.265333  |  0.8794   |   0.91
 80    |   0.016959   |  0.276330  |  0.8755   |   0.93
 90    |   0.013966   |  0.315720  |  0.8677   |   0.91
 100   |   0.012474   |  0.276041  |  0.8716   |   1.22


Training complete! Best accuracy: 0.8949.
test accuracy: 0.9007 and test f1 score: 0.8745


0.9006550218340611
```

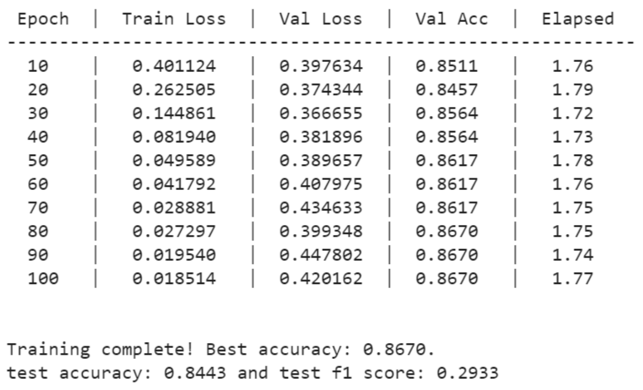
Hyperparameters	Values
optimizer	SGD with Momentum
learning rate	0.01
dropout rate	0
number of filters	100
strides	1
filter sizes	7
pooling	1-max pooling
batch size	50
number of epochs	100

```
Epoch  |  Train Loss  |  Val Loss  |  Val Acc  |  Elapsed
------------------------------------------------------------
 10    |   0.161837   |  0.290545  |  0.8677   |   2.47
 20    |   0.057989   |  0.290964  |  0.8949   |   3.07
 30    |   0.028067   |  0.205796  |  0.9027   |   3.33
 40    |   0.018525   |  0.330657  |  0.8911   |   2.99
 50    |   0.013823   |  0.224414  |  0.9027   |   2.49
 60    |   0.010446   |  0.215805  |  0.9027   |   2.47
 70    |   0.008988   |  0.244130  |  0.8988   |   2.43
 80    |   0.006654   |  0.220599  |  0.9066   |   3.36
 90    |   0.005145   |  0.236821  |  0.9066   |   2.43
 100   |   0.005987   |  0.241725  |  0.9066   |   2.45


Training complete! Best accuracy: 0.9144.
test accuracy: 0.8985 and test f1 score: 0.8696
```

Hyperparameters	Values
optimizer	SGD with Momentum
learning rate	0.01
dropout rate	0.2
number of filters	100
strides	1
filter sizes	3
pooling	1-max pooling
batch size	50
number of epochs	100

```
Epoch  |  Train Loss  |  Val Loss  |  Val Acc  |  Elapsed
------------------------------------------------------------
 10    |   0.225622   |  0.236081  |  0.9027   |   2.14
 20    |   0.110173   |  0.237622  |  0.9066   |   1.49
 30    |   0.065563   |  0.222239  |  0.9027   |   1.50
 40    |   0.045893   |  0.242006  |  0.9027   |   1.52
 50    |   0.035183   |  0.189779  |  0.8949   |   1.50
 60    |   0.020420   |  0.320396  |  0.9027   |   1.47
 70    |   0.020440   |  0.350254  |  0.8949   |   1.48
 80    |   0.016241   |  0.205847  |  0.9027   |   1.53
 90    |   0.015839   |  0.283696  |  0.8988   |   1.53
 100   |   0.014711   |  0.265655  |  0.9027   |   1.51


Training complete! Best accuracy: 0.9105.
test accuracy: 0.9083 and test f1 score: 0.8827
```

Figure 4. The hyper-parameter combinations number 6, 7 and 8 and their results.



```
Epoch  |  Train Loss  |  Val Loss  |  Val Acc  |  Elapsed
------------------------------------------------------------
 10    |   0.401124   |  0.397634  |  0.8511   |   1.76
 20    |   0.262505   |  0.374344  |  0.8457   |   1.79
 30    |   0.144861   |  0.366655  |  0.8564   |   1.72
 40    |   0.081940   |  0.381896  |  0.8564   |   1.73
 50    |   0.049589   |  0.389657  |  0.8617   |   1.78
 60    |   0.041792   |  0.407975  |  0.8617   |   1.76
 70    |   0.028881   |  0.434633  |  0.8617   |   1.75
 80    |   0.027297   |  0.399348  |  0.8670   |   1.75
 90    |   0.019540   |  0.447802  |  0.8670   |   1.74
 100   |   0.018514   |  0.420162  |  0.8670   |   1.77


Training complete! Best accuracy: 0.8670.
test accuracy: 0.8443 and test f1 score: 0.2933
```

Figure 5. Results of running combination 3 on the second data set.