CCDS Workshop Task Results

Submitted By: Syed Md. Ahnaf Hasan

1. TODO Task-1:

Architecture-1:

The custom CNN model was trained on the train set. During training the model was also simultaneously evaluated on the validation set. Later, the test split was used to evaluate the performance of the trained model.

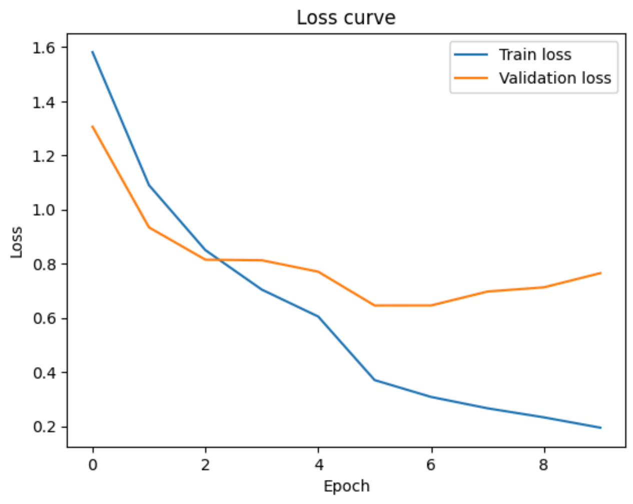


Fig: Loss curve

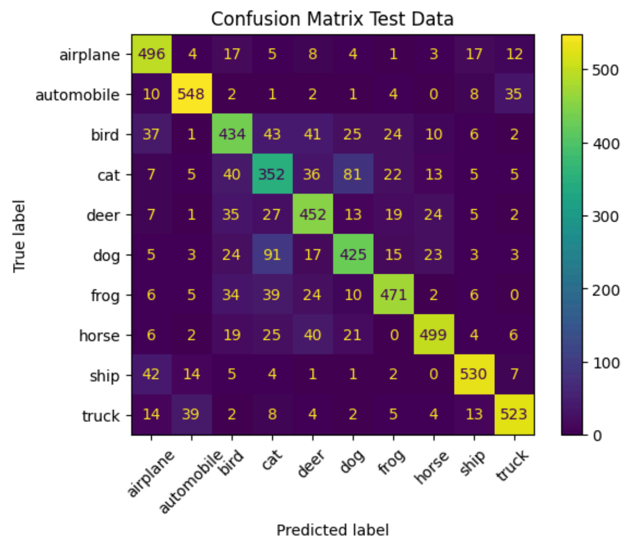
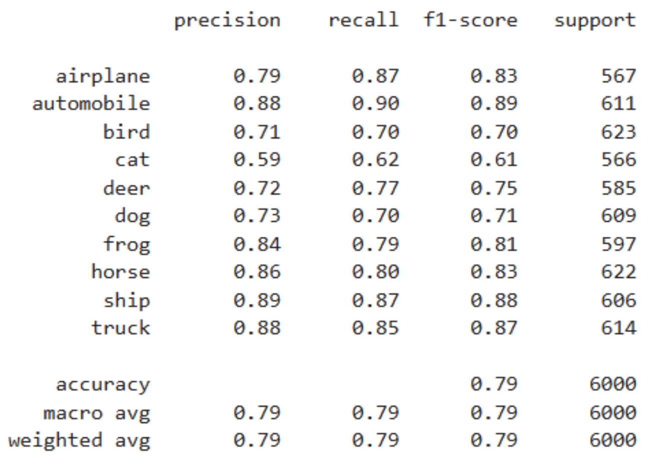


Fig: Confusion Matrix of Test Split



```
              precision    recall  f1-score   support

    airplane       0.79      0.87      0.83       567
  automobile       0.88      0.90      0.89       611
        bird       0.71      0.70      0.70       623
         cat       0.59      0.62      0.61       566
        deer       0.72      0.77      0.75       585
         dog       0.73      0.70      0.71       609
        frog       0.84      0.79      0.81       597
       horse       0.86      0.80      0.83       622
        ship       0.89      0.87      0.88       606
       truck       0.88      0.85      0.87       614

    accuracy                           0.79      6000
   macro avg       0.79      0.79      0.79      6000
weighted avg       0.79      0.79      0.79      6000
```

Fig: Performance metrics on the Test set

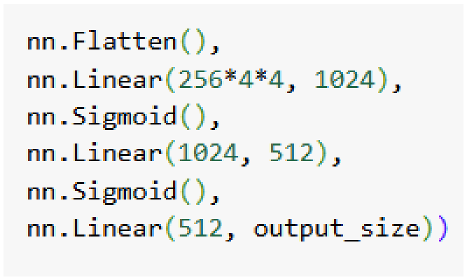
Observations:

- Overfitting was noticed after epoch 2 in training.
- The model's performance was best for the automobile class (548 correct classifications) and worst for the cat class (352 correct classifications) as seen from the confusion matrix of the test set.
- The macro average F1 score is 0.79.

2. TODO Task-2:

Architecture-2:

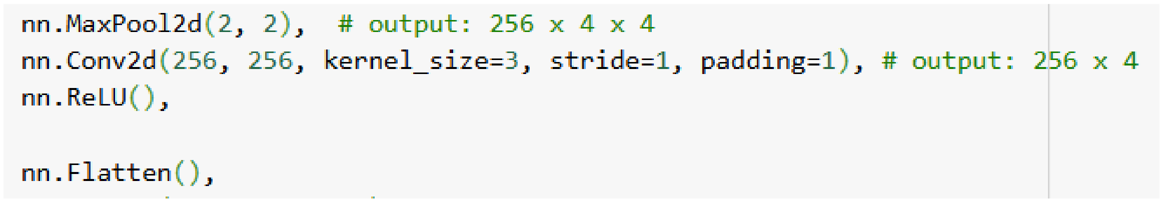
- Changes in code:
  Sigmoid layers were used instead of ReLU layers after each linear layers:



```
nn.Flatten(),
nn.Linear(256*4*4, 1024),
nn.Sigmoid(),
nn.Linear(1024, 512),
nn.Sigmoid(),
nn.Linear(512, output_size))
```

Architecture-3:

- Changes in code:
  An additional Conv2d layer and a ReLU layer was added after the last maxpool layer before flattening:

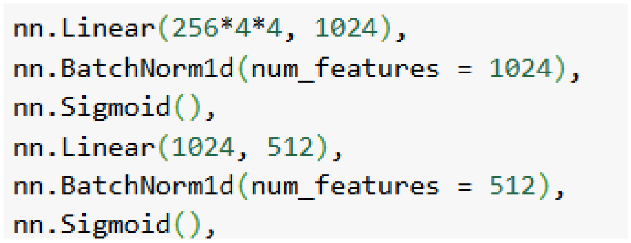


```
nn.MaxPool2d(2, 2),  # output: 256 x 4 x 4
nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1), # output: 256 x 4
nn.ReLU(),

nn.Flatten(),
```

Architecture-4:

- Changes in code:
  Batch normalization was used after each linear layer before the Sigmoid activation functions:



```
nn.Linear(256*4*4, 1024),
nn.BatchNorm1d(num_features = 1024),
nn.Sigmoid(),
nn.Linear(1024, 512),
nn.BatchNorm1d(num_features = 512),
nn.Sigmoid(),
```

Loss Curves:

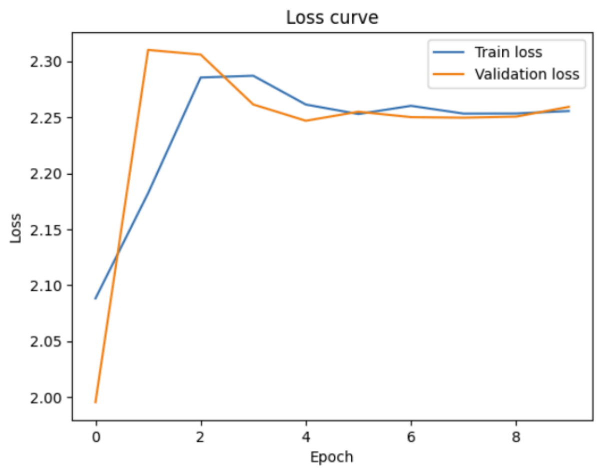


Fig: Loss Curve (Architecture-2)

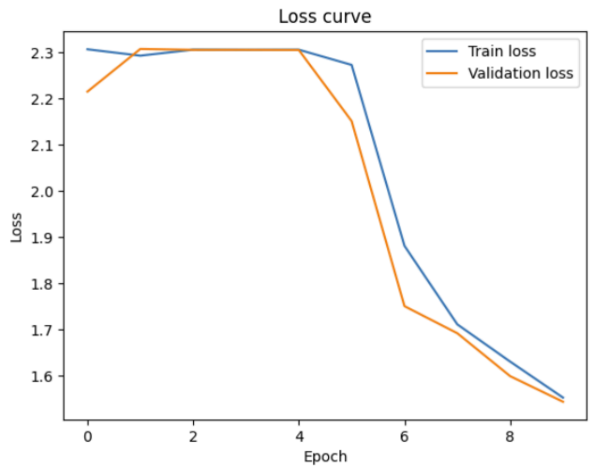


Fig: Loss Curve (Architecture-3)

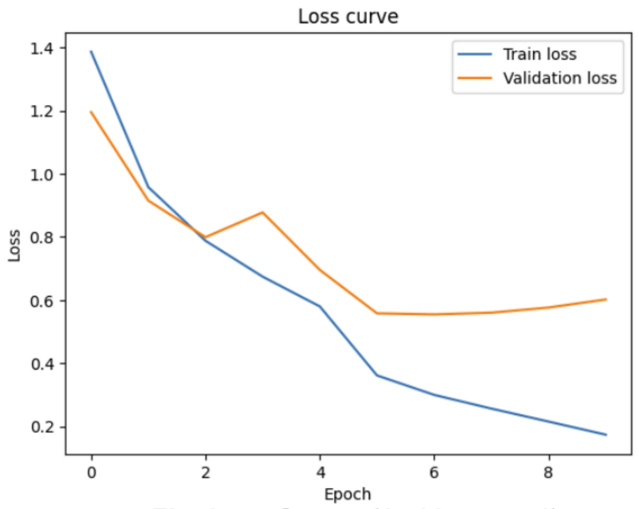


Fig: Loss Curves (Architecture-4)

Observation:

- Introducing Sigmoid activations after linear layers in architecture-2 makes the gradients of the high activation outputs close to zero, consequently the weights are not updated in the backpropagation and the loss does not decrease.
- Introducing the extra Conv+Relu block in architecture -3 makes the loss drop after 6 epochs.
- Batch normalization in architecture-4 normalizes the activation outputs of the linear layers. This forces the activation outputs to stay in a linear range where gradients are not zero and hence the updates take place during backpropagation. However overfitting is noticed after epoch 4.

3. **TODO Task-3:**
   Comparison table of the 4 architectures is shown below:

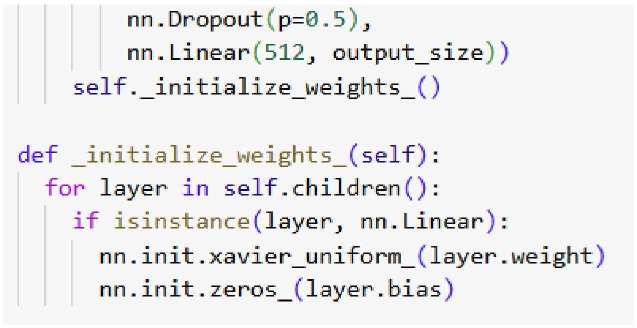
Table: Comparing Performance Metrics

Metrics Models	Accuracy	Precision (Macro Avg)	Recall (Macro Avg)	F1 Score (Macro Avg)
Architecture-1	0.78	0.78	0.78	0.78
Architecture-2	0.22	0.19	0.22	0.17
Architecture-3	0.43	0.43	0.43	0.42
Architecture-4	**0.82**	**0.82**	**0.82**	**0.82**

Observation:
The architecture-4 obtained the highest scores across all the performance metrics.

4. **TODO Task-4:**
   I have used architecture-4 for this task, since I have obtained the highest accuracy by using this architecture.

   Changes in code:
   For Xavier initialization the _initialize_weights_() method was declared which was called in the __init__() method.


```
                nn.Dropout(p=0.5),
                nn.Linear(512, output_size))
        self._initialize_weights_()

def _initialize_weights_(self):
    for layer in self.children():
        if isinstance(layer, nn.Linear):
            nn.init.xavier_uniform_(layer.weight)
            nn.init.zeros_(layer.bias)
```

For dropout the nn.Dropout layer was added after the activation functions:

```
nn.Linear(256*4*4, 1024),
nn.BatchNorm1d(num_features = 1024),
nn.Sigmoid(),
nn.Dropout(p=0.5),
nn.Linear(1024, 512),
nn.BatchNorm1d(num_features = 512),
nn.Sigmoid(),
nn.Dropout(p=0.5),
```

The learning_rate scheduler was changed to cosine learning rate:

```
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max = 10)
```

The loss curves for batch size = 256, learning rate = 0.001, and Cosine Annealing Learning rate scheduler are shown below:

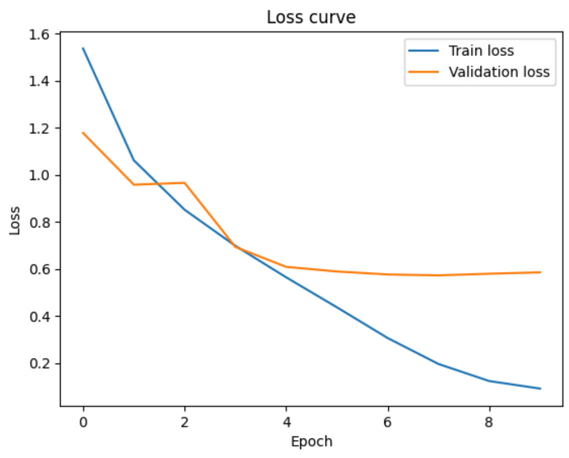


Fig: Loss Curves

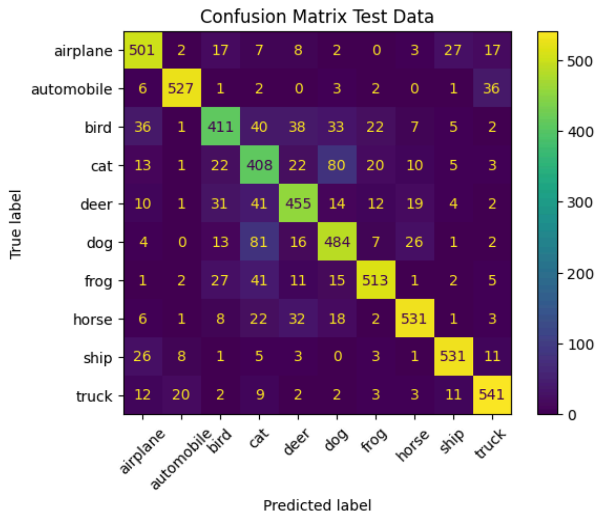


Fig: Confusion matrix

```
              precision    recall  f1-score   support

    airplane       0.81      0.86      0.84       584
  automobile       0.94      0.91      0.92       578
        bird       0.77      0.69      0.73       595
         cat       0.62      0.70      0.66       584
        deer       0.78      0.77      0.77       589
         dog       0.74      0.76      0.75       634
        frog       0.88      0.83      0.85       618
       horse       0.88      0.85      0.87       624
        ship       0.90      0.90      0.90       589
       truck       0.87      0.89      0.88       605

    accuracy                           0.82      6000
   macro avg       0.82      0.82      0.82      6000
weighted avg       0.82      0.82      0.82      6000
```

Fig: Performance Metrics

Observation:
The performance after making the aforementioned changes is similar to the original architecture's. Overfitting is noticeable in this configuration also.

5. **TODO Task-5:**
The confusion matrices for torch embeddings and word2vec embeddings are shown below:

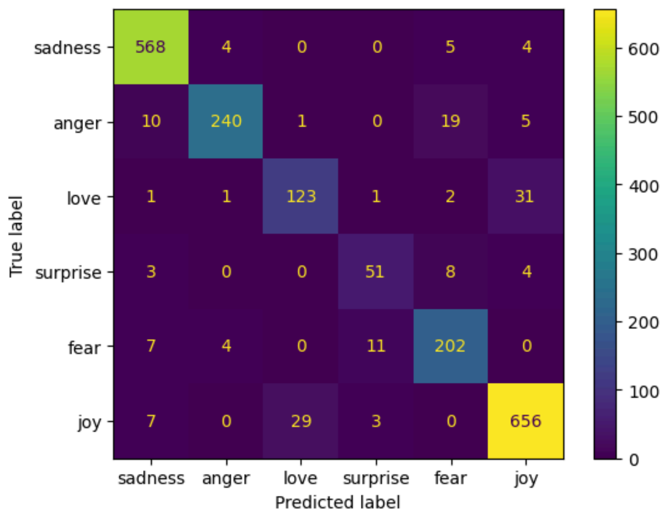


Fig: Conf. Matrix (torch Embeddings)

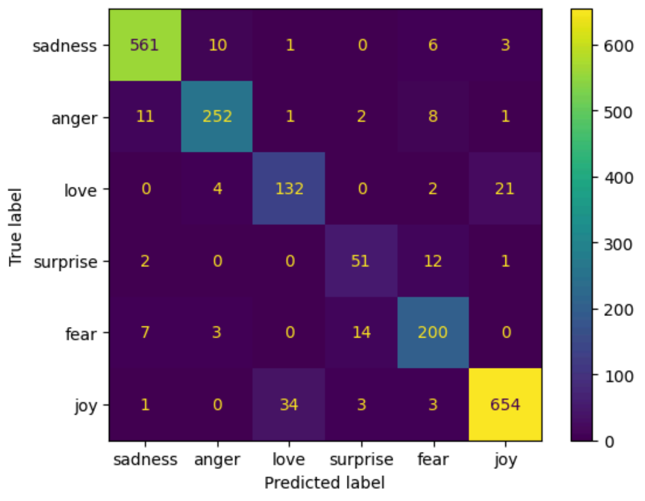
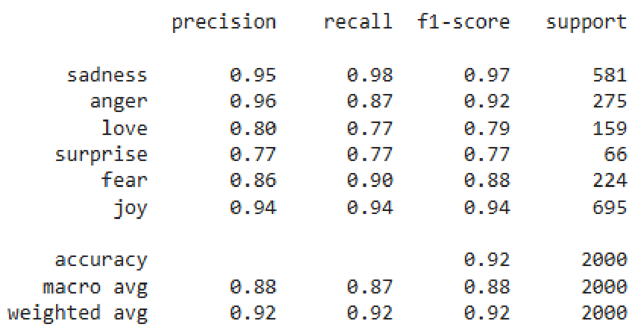


Fig: Conf. Matrix (Word2Vec Embeddings)

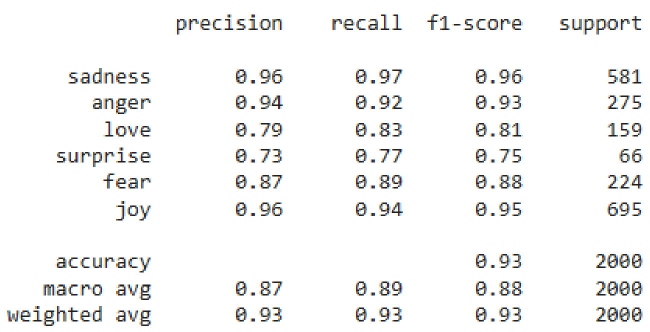


```
              precision    recall  f1-score   support

     sadness       0.95      0.98      0.97       581
       anger       0.96      0.87      0.92       275
        love       0.80      0.77      0.79       159
    surprise       0.77      0.77      0.77        66
        fear       0.86      0.90      0.88       224
         joy       0.94      0.94      0.94       695

    accuracy                           0.92      2000
   macro avg       0.88      0.87      0.88      2000
weighted avg       0.92      0.92      0.92      2000
```

Fig: Performance metrics (torch emb.)



```
              precision    recall  f1-score   support

     sadness       0.96      0.97      0.96       581
       anger       0.94      0.92      0.93       275
        love       0.79      0.83      0.81       159
    surprise       0.73      0.77      0.75        66
        fear       0.87      0.89      0.88       224
         joy       0.96      0.94      0.95       695

    accuracy                           0.93      2000
   macro avg       0.87      0.89      0.88      2000
weighted avg       0.93      0.93      0.93      2000
```

Fig: Performance metrics (Word2Vec)

- **Observation:**
  The macro average F1 scores for both torch embeddings and word2vec embeddings was found to be 0.88. However, the accuracy was slightly higher in case of word2vec embeddings.

6. TODO Task-6:

- **Pseudo plan:**
    1. Observe the attention output's shape and its elements to understand its structure. -> The shape of the attention output is: (batch x encoder_block_num x num_of_sentences x num_of_attention_heads_per_encoder_block x num_of_tokens x num_of_tokens).
    2. Try to find the most important token for a single sentence.
    3. Create a function to output the predicted class, most important token, highest attention score for a single sentence.
    4. Iterate through each sentence in each batch and aggregate the attention scores for each tokens in each class.
    5. Display the most important token in each class.

- **Colab notebook link:**

  https://colab.research.google.com/drive/1iNXyahNct_WOt-SJxarZAJVizeTmEurs?usp=sharing

- **Observation:**

  The most attentive tokens found for each class are as follows:

  1. Sadness -> depressed
  2. Anger -> feeling
  3. Love -> fond
  4. Surprise -> overwhelmed
  5. Fear -> shy
  6. Joy -> feel