Michael Siebel
Final Project: Group 5
DATS 6203
Machine Learning II
Section 10

Game of Classification
Deep Learning on Game of Thrones Screenshots

Individual Report

**Work**

I wrote an Rmd program, for the statistical language R, that webscrapped images from IMDB. Each image was named a value and the shows name, and a corresponding text file was created with the same name. After webscrapping them into folders per show name, I wrote a python script to collect all files, label the text files "1 Game of Thrones" or "0 Sitcom". The script then split the files and placed them in three folders, "train", "test", and "val" based on a 70%, 15%, and 15% split.

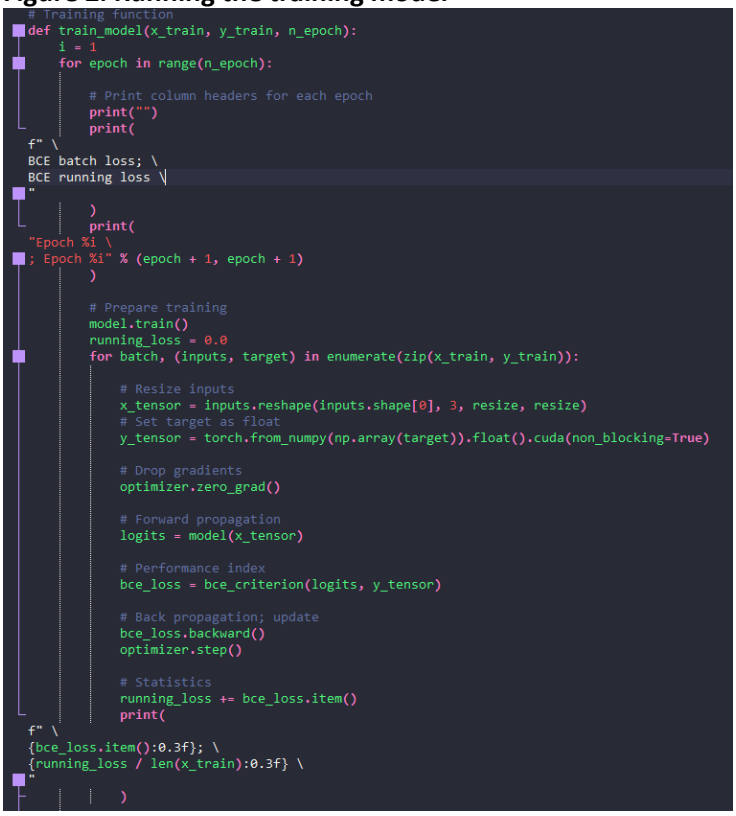
I wrote a script for each model I ran. These scripts were above 400 lines, although with plenty of spacing. They were written to take as input file paths and create lists of jpg images and corresponding text files for the target label. First, loaded only the text files for the training data into python and used label encoder to prepare them for modeling. Next, I oversampled the list of jpg file paths and shuffled the lists using the code in Figure 1.

**Figure 1. Oversampling lists of file paths and shuffling**

```
# Oversample GoT
x_ovsp = []
y_ovsp = []
for im, lab in zip(x_train, y_train):
    if lab == 1:
        x_ovsp.append(im)
        y_ovsp.append(lab)

y_ovsp = np.array(y_ovsp)

def shuffle_train(a, b):
    assert len(a) == len(b)
    shuffled_a = np.empty(a.shape, dtype=a.dtype)
    shuffled_b = np.empty(b.shape, dtype=b.dtype)
    permutation = np.random.permutation(len(a))
    for old_index, new_index in enumerate(permutation):
        shuffled_a[new_index] = a[old_index]
        shuffled_b[new_index] = b[old_index]
    return shuffled_a, shuffled_b

x_train = np.concatenate((x_train, x_ovsp), axis=0)
y_train = np.concatenate((y_train, y_ovsp), axis=0)
x_train, y_train = shuffle_train(x_train, y_train)
```

I only then read in the images and set up the data loader with the data augmentations. Finally, I completed the code on the training data by setting up the model using Pytorch syntax. Figure 2 displays the code for running the training model. It included printing the loss function on each batch as well as a running loss.

**Figure 2. Running the training model**



```
# Training function
def train_model(x_train, y_train, n_epoch):
    i = 1
    for epoch in range(n_epoch):

        # Print column headers for each epoch
        print("")
        print(
f" \
BCE batch loss; \
BCE running loss \
"
        )
        print(
"Epoch %i \
; Epoch %i" % (epoch + 1, epoch + 1)
        )

        # Prepare training
        model.train()
        running_loss = 0.0
        for batch, (inputs, target) in enumerate(zip(x_train, y_train)):

            # Resize inputs
            x_tensor = inputs.reshape(inputs.shape[0], 3, resize, resize)
            # Set target as float
            y_tensor = torch.from_numpy(np.array(target)).float().cuda(non_blocking=True)

            # Drop gradients
            optimizer.zero_grad()

            # Forward propagation
            logits = model(x_tensor)

            # Performance index
            bce_loss = bce_criterion(logits, y_tensor)

            # Back propagation; update
            bce_loss.backward()
            optimizer.step()

            # Statistics
            running_loss += bce_loss.item()
            print(
f" \
{bce_loss.item():0.3f}; \
{running_loss / len(x_train):0.3f} \
"
            )
```

Finally, I included the predict function in the same script, which removed the tensors from using GPUs. The predict function saved both the raw logit values and the prediction with the hard limit applied. Figure 3 shows how I coded the later part of the predict function output these two values.

**Figure 3. Saving logit prediction values and binary values after hard limit**

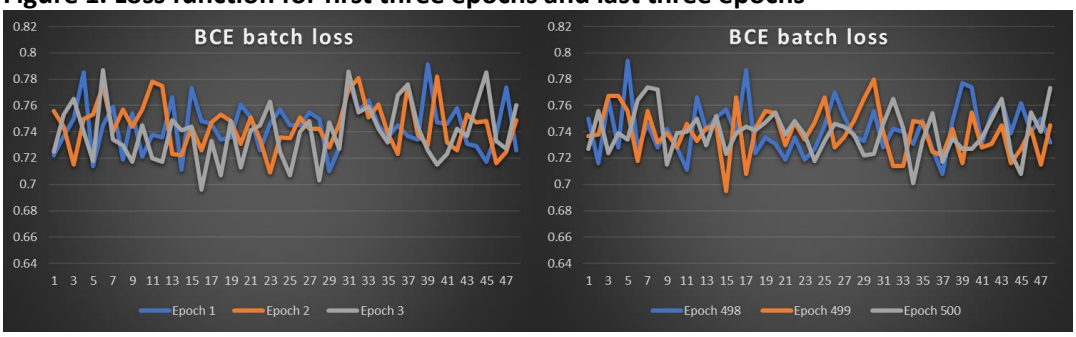
```
    # Initiate y_pred
    y_pred = torch.zeros(1, 1)
    y_logits = torch.zeros(1, 1)
    with torch.no_grad():
        for batch, inputs in enumerate(x):

            # Resize inputs
            x_tensor = inputs.reshape(inputs.shape[0]*inputs.shape[1], 3, resize, resize)

            # Model
            logits = model(x_tensor)

            # Add raw logits
            y_logits = torch.cat([y_logits, logits])

            # Hardlimit
            logits = np.array(logits)
            threshold = np.array([.5])
            np.putmask(logits, logits >= threshold, 1)
            np.putmask(logits, logits <  threshold, 0)
            y_lab = torch.FloatTensor(logits).to(device)

            # Add to y_pred
            y_pred = torch.cat([y_pred, y_lab])

    # Remove initiation row
    y_logits = y_logits[1:len(y_logits)]
    y_pred = y_pred[1:len(y_pred)]
    return y_pred, y_logits

y_pred, y_logits = predict(x_val)
y_pred = np.array(y_pred)
y_logits = np.array(y_logits)
```

As the last step, each script loaded the testing or validating labels and printed the evaluation statistics.

**Results**

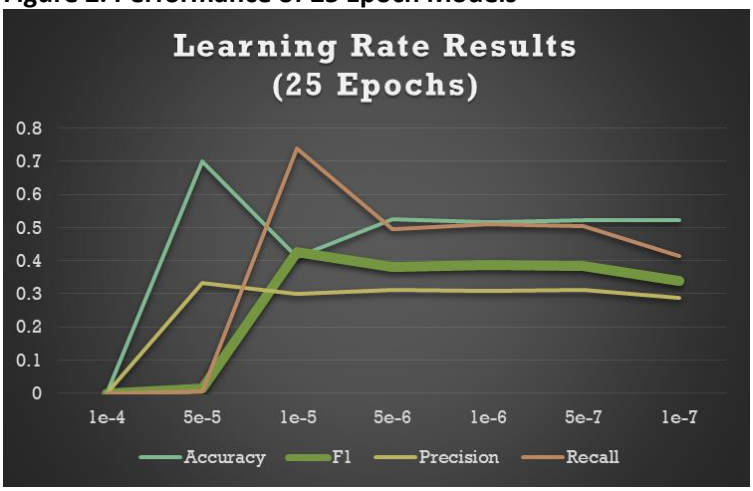
My scripts printed out the information I needed for evaluation. This included the loss function for each batch, grouped by epoch. This enabled me to provide detailed graphs of the loss function. Figure 1 shows the loss function by batch at the beginning of the run (first three epochs) to the end of the run (last three epochs). I created these graphs by printing the loss in python and pasting the output into excel to graph. This figure shows that the loss did not improve across epochs.

**Figure 1. Loss function for first three epochs and last three epochs**




Further, I printed accuracy, F1-scores, precession, and recall at the end of each run, based on a prediction function on either the testing data or training data. Again, I copied this data into excel to graph. Figure 2 shows these statistics for several models running 25 epochs. Here we can see that at learning rates of 1e-4 and 5e-5, I have lines hovering around zero for all or some of the statistics. For 1e-4, all statistics are zero. This is because the model made no predictions for the main class (Game of Thrones images). As a result the F1-score, precision, and recall contained invalid metrics. I placed "NA" for the value of these statistics. I further placed "NA" for accuracy simply to not evaluate these statistics, even the statistic is technically a valid number. In excel, "NA" is interpreted as zero, and this is how I chose to display this result.

**Figure 2. Performance of 25 Epoch Models**




With each run, I further printed a distribution of the predicted values and the confusion matrix. From this I was able to observe that as I increased epochs, lowered the learning rate, oversampled data, and

used five instead of three convolutional layers, Game of Thrones predictions increased as well as the true positives. Despite this, my F1-scores largely remained within a 0.3 to 0.4 score.

**Table 1. Confusion Matrix and Final Distributions**

| Confusion Matrix | Predicted Sitcom | Predicted Game of Thrones |
|---|---|---|
| Actual Sitcom | 533 | 930 |
| Actual Game of Thrones | 202 | 366 |


**Conclusion**

My model required more than data augmentation tricks and tuning of hyper-parameters. This was likely because the image quality was too weak for such complex images. With more time, the first thing I would try is to grab the full-size images. I would be interested to see if the issue was simply related to image quality. Second, I would investigate loss functions. For binary classifiers, I did not see much literature on how loss functions could be advantageous with certain data issues other than handling the issue of sparsity—which was not the issue in my dataset. My loss function, binary cross entropy loss (BCELoss) did not diminish greatly across epochs, raising red flags. Finally, the best solution would be to attempt an ensemble algorithm. This way, I could include pretrained model such as VGG-Face Model. As mentioned, the goal of my project is not to recognize faces but including this pretrained model could help differentiate some of the characters. More importantly, adding other models with varying architectures could help remove any bias in my main model.

**Code Usage**

27 lines taken for the basis of the model
14 lines added
6 lines modified

( (27 – 6) / (27 + 14) ) * 100 = 51%