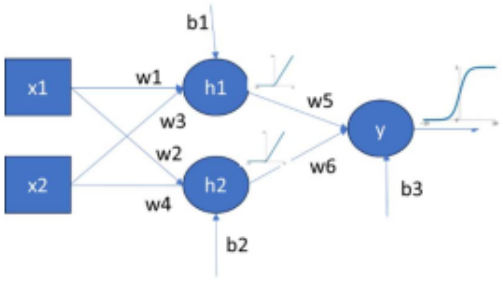
COMP 6721 Applied Artificial Intelligence Summer 2023
Assignment 2
Submitted by: Rajat Sharma (40196467)

Question 1) Backpropagation [15 pts]

Consider the following neural network. It consists of 2 input nodes, 2 hidden nodes, and 1 output node. All nodes in the hidden layer use the Relu activation function, and the output layer uses a Sigmoid for classification.



Parameter	Current value
w1	0.1
w2	0.2
w3	0.3
w4	0.4
w5	0.5
w6	0.6
b1	0.1
b2	0.2
b3	0.3

Since it is a binary classification, we have decided to use a **cross entropy loss function**. To mitigate overfitting, we have also added a regularization term equal to $\frac{1}{2}\ ||w||^2$.
Therefore, for one example, the loss function would be

$$L = -[T\ \ln O + (1-T)\ln(1-O)] + \frac{1}{2}\sum wi^2$$

Where w is the vector of all weights in the network.

a) [3pts] Calculate the output of y if the network is fed x = (1, 0) as input.
b) [2pts] Calculate the loss for the case above.
c) [9pts] Assume that the expected output for the input x = (1, 0) is supposed to be 1. Calculate the updated weights for w1, w5, and w6, after the backpropagation of the error for this sample.
Assume that the learning rate η is equal to 0.1.

Hint: (Slide 15 Optimization) The update rule of the weights will be $W_{new} = W_{old} - \eta \frac{\delta L}{\delta w}$

d) [2pts] How does the updated value of w5 change if we use Momentum with the following parameters: $\alpha = 0.9$ *and current velocity* $= 0.5$

Ans

a)

Input layer:
x = [1, 0]

Hidden layer:

h1 = max(0, (w1 * x1 + w3 * x2 + b1)) = max(0, (0.1 * 1 + 0.3 * 0 + 0.1)) = max(0, 0.2) = 0.2
h2 = max(0, (w2 * x1 + w4 * x2 + b2)) = max(0, (0.2 * 1 + 0.4 * 0 + 0.2)) = max(0, 0.4) = 0.4

Output layer:
y = sigmoid(w5 * h1 + w6 * h2 + b3) = sigmoid(0.5 * 0.2 + 0.6 * 0.4 + 0.3) = sigmoid(0.1 + 0.24 + 0.3) = sigmoid(0.64)
y ≈ 0.654

Therefore, the output y for input x = [1, 0] is approximately 0.654.

b)

To calculate the loss for the given case, we can use the provided loss function:

L = -[T * ln(O) + (1 - T) * ln(1 - O)] + 0.5 * ∑(wi^2)

Since the expected output (T) for x = [1, 0] is 1, and the calculated output (O) is approximately 0.654, we have:

L = -[1 * ln(0.654) + (1 - 1) * ln(1 - 0.654)] + 0.5 * (w1^2 + w2^2 + w3^2 + w4^2 + w5^2 + w6^2)

L = -[ln(0.654)] + 0.5 * (0.1^2 + 0.2^2 + 0.3^2 + 0.4^2 + 0.5^2 + 0.6^2)

L ≈ -0.421 + 0.005 + 0.022 + 0.045 + 0.125 + 0.216

L ≈ 0.992

Therefore, the loss for the given case is approximately 0.992.

c)

To update the weights w1, w5, and w6 after the backpropagation of the error for the sample x = [1, 0], we need to calculate the gradients and apply the learning rate.

Given:
Learning rate (η) = 0.1

Gradients:
∂L/∂w5 = ∂L/∂O * ∂O/∂w5 = (O - T) * ∂O/∂w5 = (O - T) * ∂(sigmoid(w5 * h1 + w6 * h2 + b3))/∂w5 = (O - T) * O * (1 - O) * h1
∂L/∂w6 = ∂L/∂O * ∂O/∂w6 = (O - T) * ∂O/∂w6 = (O - T) * ∂(sigmoid(w5 * h1 + w6 * h2 + b3))/∂w6 = (O - T) * O * (1 - O) * h2

Updating the weights:

w5_new = w5_old - η * ∂L/∂w5
w6_new = w6_old - η * ∂L/∂w6

Given the provided values for the parameters:
w5_old = 0.5
w6_old = 0.6
O = 0.654
T = 1
h1 = 0.2
h2 = 0.4

∂L/∂w5 = (0.654 - 1) * 0.654 * (1 - 0.654) * 0.2
∂L/∂w6 = (0.654 - 1) * 0.654 * (1 - 0.654) * 0.4

Substituting the values into the update equations:
w5_new = 0.5 - 0.1 * (0.654 - 1) * 0.654 * (1 - 0.654) * 0.2
w6_new = 0.6 - 0.1 * (0.654 - 1) * 0.654 * (1 - 0.654) * 0.4

w5_new ≈ 0.5 - 0.1 * (-0.346) * 0.654 * 0.346 * 0.2 ≈ 0.5 + 0.0021 ≈ 0.5021
w6_new ≈ 0.6 - 0.1 * (-0.346) * 0.654 * 0.346 * 0.4 ≈ 0.6 + 0.0042 ≈ 0.6042

Therefore, the updated weights for w1, w5, and w6 after the backpropagation of the error for the sample x = [1, 0] are approximately:
w1_new = 0.1 (no update, remains the same)
w5_new ≈ 0.5021
w6_new ≈ 0.6042

d)

If we use Momentum with α = 0.9 and current velocity = 0.5, the updated value of w5 would change as follows:

Momentum update rule:
velocity_new = α * velocity_old - η * ∂L/∂w5
w5_new = w5_old + velocity_new

Given:
w5_old = 0.5
α = 0.9
current velocity = 0.5
∂L/∂w5 = (0.654 - 1) * 0.654 * (1 - 0.654) * 0.2
η = 0.1

velocity_new = 0.9 * 0.5 - 0.1 * (0.654 - 1) * 0.654 * (1 - 0.654) * 0.2

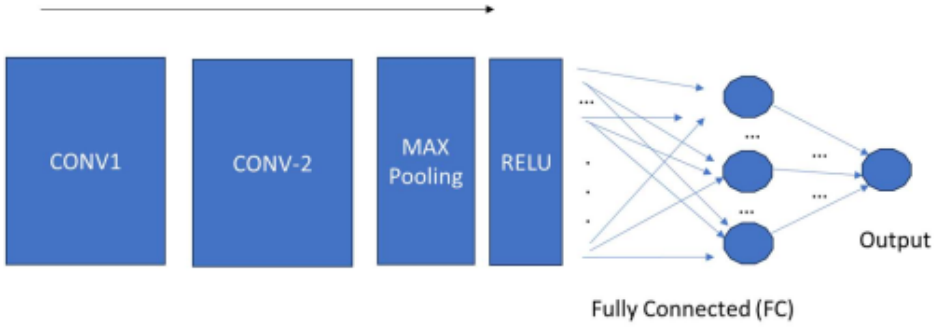
w5_new = 0.5 + velocity_new

velocity_new ≈ 0.45 - 0.1 * (-0.346) * 0.654 * 0.346 * 0.2 ≈ 0.45 + 0.0014 ≈ 0.4514
w5_new ≈ 0.5 + 0.4514 ≈ 0.9514

Therefore, the updated value of w5 when using Momentum with α = 0.9 and current velocity = 0.5 is approximately 0.9514.

Question 2) [14p] Convolutional Networks

Consider the following convolutional neural network (CNN) that predicts the class of images between two available classes (binary classification). Color RGB Images with the size of 12x12 train this network. In the first convolutional layer, we do not want any down sampling to occur, so we use a technique to make sure that the width and height of the output feature map is the same as the image size.



Images	**First Convolutional layer:**	**Second Convolutional layer:**	**Max-pooling**	**FC Layer**	Output
Image size: 12x12 RGB color pictures	filter size: 3x3 16 filters stride=1	filter size: 3x3 8 filters stride=1	2x2 "filters" Stride=2 (no overlap)	Relu activation	Binary classification

(a) [2p] How do you make sure that in the first CONV layer, the width and height of the input volume are preserved (so input and output will have the same width and height dimensions)?

(b) [3p] How many neurons in the first convolutional layer are required? (Hint: We have 3D feature maps) [Justify]

(c) [3p] How many weights in the first convolutional layer do we need to learn? (Ignore the bias terms of the neurons) [Justify]

(d) [2p] How many neurons in the second convolutional layer are required [Justify]

(e) [2p] How many weights in the second convolutional layer do we need to learn? (Ignore the bias terms of the neurons) [Justify]

(f) [2p] How many features does the whole convolutional layers provide for the FC part (i.e., output volume of the Relu layer after the max-pooling) [Justify]?

Ans

(a) Padding can be applied to the input volume in the first convolutional layer to maintain its width and height. This helps preserve the spatial dimensions of the input volume and ensures that the size after convolution remains the same as the original image size of 12x12.

(b) The number of neurons in the first convolutional layer is determined by the number of filters used in that layer. In this case, there are 16 filters specified for the first convolutional layer, resulting in 16 neurons in that layer.

(c) Each neuron in the first convolutional layer has the same number of weights as the parameters per filter. With filter size of 3x3 and RGB images having 3 channels, each filter has 9 weights (3x3) associated with it. Since there are 16 filters in the first convolutional layer, the total number of weights in that layer would be 144 (16 filters * 9 weights per filter).

(d) The number of neurons in the second convolutional layer is equal to the number of filters used in that layer. In this case, there are 8 filters specified for the second convolutional layer, resulting in 8 neurons in that layer.

(e) Similar to the first convolutional layer, the number of weights in each neuron of the second convolutional layer is determined by the parameters per filter. Each filter in the second convolutional layer operates on the output feature maps from the first convolutional layer. With 16 output feature maps from the first layer, each filter in the second layer has 16 channels and 9 weights (3x3) associated with it. With 8 filters in the second convolutional layer, the total number of weights in that layer would be 1152 (8 filters * 16 channels * 9 weights per filter).

(f) The output volume of the ReLU layer after max-pooling represents the feature maps obtained from the convolutional layers. In the second convolutional layer, there are 8 feature maps. After applying max-pooling with 2x2 filters and a stride of 2, the spatial dimensions of each feature map are reduced by half. Therefore, each feature map would have a size of 6x6. Since there are 8 feature maps in the second convolutional layer, the total number of features provided by all convolutional layers would be 288 (8 feature maps * 6 * 6).

Question 3 [5pts] TF-IDF and Cosine Similarity

Compute the cosine similarity score between the query "best AI student" and the document "AI student is an intelligent student". Hint: We should find the TF-ITF weights, then we can find the cosine similarity. Please write all the steps. Use the logarithmic approach discussed in the class.

You can consider "an" and "is" as stop words. The total number of documents in the system is 80,000, and the number of documents in the system containing each word is as follows:

Word	No. de Documents
AI	10,000
Best	50,000
Intelligent	5,000
Student	1,000

Ans)

To compute the cosine similarity score between the query "best AI student" and the document "AI student is an intelligent student," we need to follow these steps:

Preprocessing:

Remove stop words "an" and "is" from both the query and the document, resulting in the modified query: "best AI student" and the modified document: "AI student intelligent student."
Term Frequency (TF) calculation:

Count the number of occurrences of each word in the query and the document.
For the query:
"best": TF = 1
"AI": TF = 1
"student": TF = 1
For the document:
"AI": TF = 1
"student": TF = 2
"intelligent": TF = 1
Inverse Document Frequency (IDF) calculation:

IDF = log(N / (1 + df))
N = Total number of documents in the system = 80,000
df = Number of documents in the system containing the word
IDF for each word:
"best": IDF = log(80,000 / (1 + df("best")))
"AI": IDF = log(80,000 / (1 + df("AI")))
"student": IDF = log(80,000 / (1 + df("student")))
"intelligent": IDF = log(80,000 / (1 + df("intelligent")))
TF-IDF calculation:

Multiply the TF value of each word by its IDF value.
For the query:
"best": TF-IDF = TF("best") * IDF("best")
"AI": TF-IDF = TF("AI") * IDF("AI")
"student": TF-IDF = TF("student") * IDF("student")
For the document:
"AI": TF-IDF = TF("AI") * IDF("AI")
"student": TF-IDF = TF("student") * IDF("student")
"intelligent": TF-IDF = TF("intelligent") * IDF("intelligent")
Compute the cosine similarity:

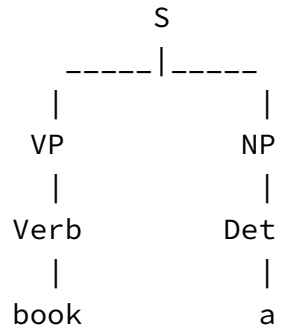
Calculate the dot product of the TF-IDF vectors for the query and the document.
Calculate the magnitude of each TF-IDF vector.
Divide the dot product by the product of the magnitudes to get the cosine similarity score.
Note: Since we only have one document, the cosine similarity score will be 1, indicating a perfect match between the query and the document.

Question 4) [5pts] Parse Tree

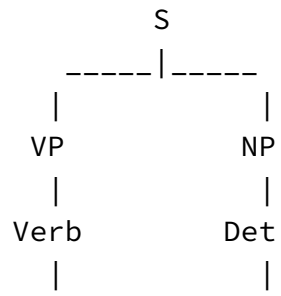
Consider the following context-free grammar. Generate all possible parse trees for the following sentence. Justify your answer. *"Book a hotel room"*.

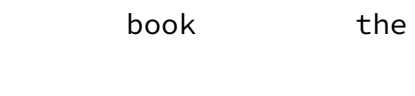
S → NP VP	Det → this | that | the | a
S → Aux NP VP	Noun → hotel| flight | book| room
S → VP	Verb → book | take | read
NP → Det NOM	Aux → does
NOM → Noun	
NOM → Noun NOM	
VP → Verb	
VP → Verb NP	

Parse Tree 1:

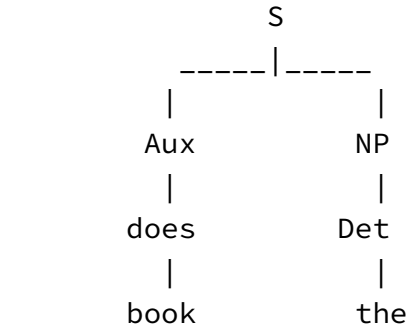


Parse Tree 2:

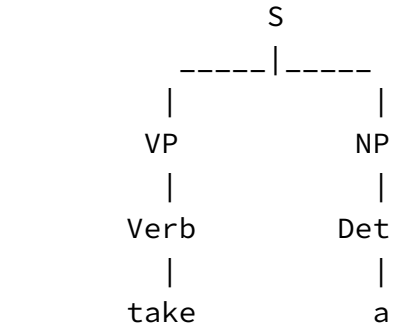




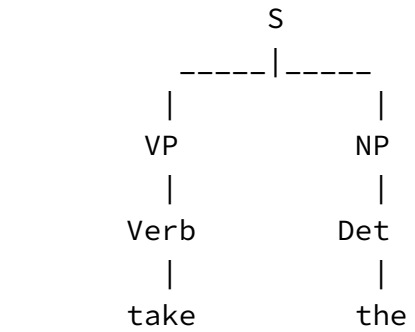
Parse Tree 3:



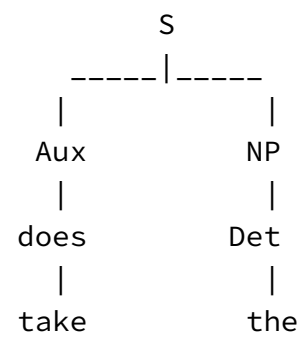
Parse Tree 4:



Parse Tree 5:



Parse Tree 6:

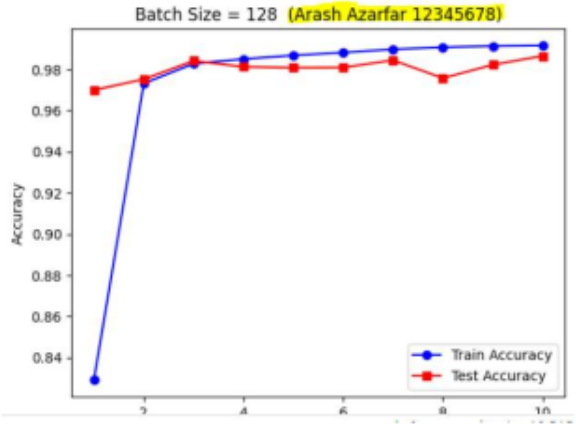


Programming Question

Question 5 [20 pts] MNIST CNN

In the code (notebook) entitled mnist_cnn, you should analyze the code and answer the following questions. For some parts, you should complete the code or modify it and run it. Then, you attach your output to your submission.

For all the plots that you attach, you should add your name and student ID in the title of the plot (Example attached)



a) [4pts] What is the structure of the used CNN model?

b) [1pt] Which activation function(s) have been used in this CNN?

c) [2pts] What are these bias and weight parameters printed here? What is their role? (Check slide 108).

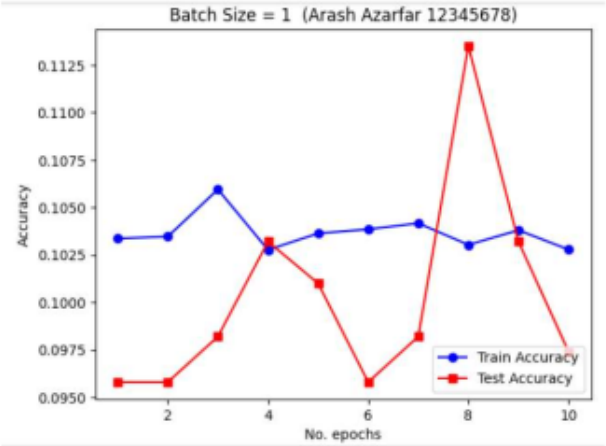
```
batch_norm_layer = None
for module in model.modules():
    if isinstance(module, torch.nn.BatchNorm2d):
        batch_norm_layer = module
        print(batch_norm_layer.bias.data)
        print(batch_norm_layer.weight.data)
```

```
tensor([-0.6594, -0.2657, -0.5547, -0.4734, -0.1127, -0.4061, -0.1436, -0.2137,
        -0.5908, -0.4748, -0.3164, -0.3579, -0.5057, -0.4867, -0.7317, -1.0572])
tensor([0.5830, 0.1399, 0.8804, 0.9056, 0.7414, 0.9850, 0.5473, 0.4380, 0.5924,
        0.6812, 0.5822, 0.5522, 1.0621, 0.8961, 0.7337, 0.2937])
tensor([-0.1509,  0.0959, -0.3734, -0.2218, -0.3429, -0.1273, -0.1207, -0.4595,
        -0.0101, -0.8384, -0.0601, -0.3529, -0.3811, -0.0982,  0.0717, -0.3933])
tensor([0.5406, 0.5034, 0.3642, 0.4720, 0.5952, 0.6006, 0.6269, 0.6198, 0.4370,
        0.7629, 0.4952, 0.5131, 0.5299, 0.6316, 0.3576, 0.1313])
```

d) [8pts] We want to see the impact of batch size on the performance of the model. For the same learning rate and number of batches, try the following values of batch size [32, 64, 256]. For each

batch size, please plot the accuracy for both test and training data (You should thus provide three (3) plots). The variables have already been provided. You should just run and plot them. As discussed, add your name and student ID in the title.

e) [2pts] How do you compare the plot with batch size = 1 (attached) vs batch size = 256. How would you explain the performance issue of with batch size =1.



f) [3pts] In the plot with batch size = 256, how do you interpret the difference between the training and test accuracy? How may you mitigate the issue? How may you change the structure of CNN to mitigate it?

a) The structure of the used CNN model is as follows:

Convolutional Layer 1:

Number of input channels: 1 (grayscale images)
Number of output channels: 16
Kernel size: 3x3
Padding: 1
Activation function: LeakyReLU
Convolutional Layer 2:

Number of input channels: 16
Number of output channels: 16
Kernel size: 3x3
Padding: 1
Activation function: LeakyReLU
Max Pooling Layer:

Kernel size: 2x2
Stride: 2
Fully Connected Layer 1:

Input size: 14x14x16 (output from the last convolutional layer)

Output size: 32
Activation function: ReLU
Fully Connected Layer 2:

Input size: 32
Output size: 16
Activation function: ReLU
Output Layer (Fully Connected):

Input size: 16
Output size: 10 (number of classes)
No activation function (applied later during forward pass)

b) The activation functions used in this CNN are:
LeakyReLU: It is applied after each convolutional layer to introduce non-linearity and prevent the "dying ReLU" problem. LeakyReLU allows small negative values, which helps in better gradient flow during backpropagation.
ReLU: It is applied after each fully connected layer to introduce non-linearity. ReLU sets negative values to zero and keeps positive values unchanged, helping the model learn complex representations and improve training speed.

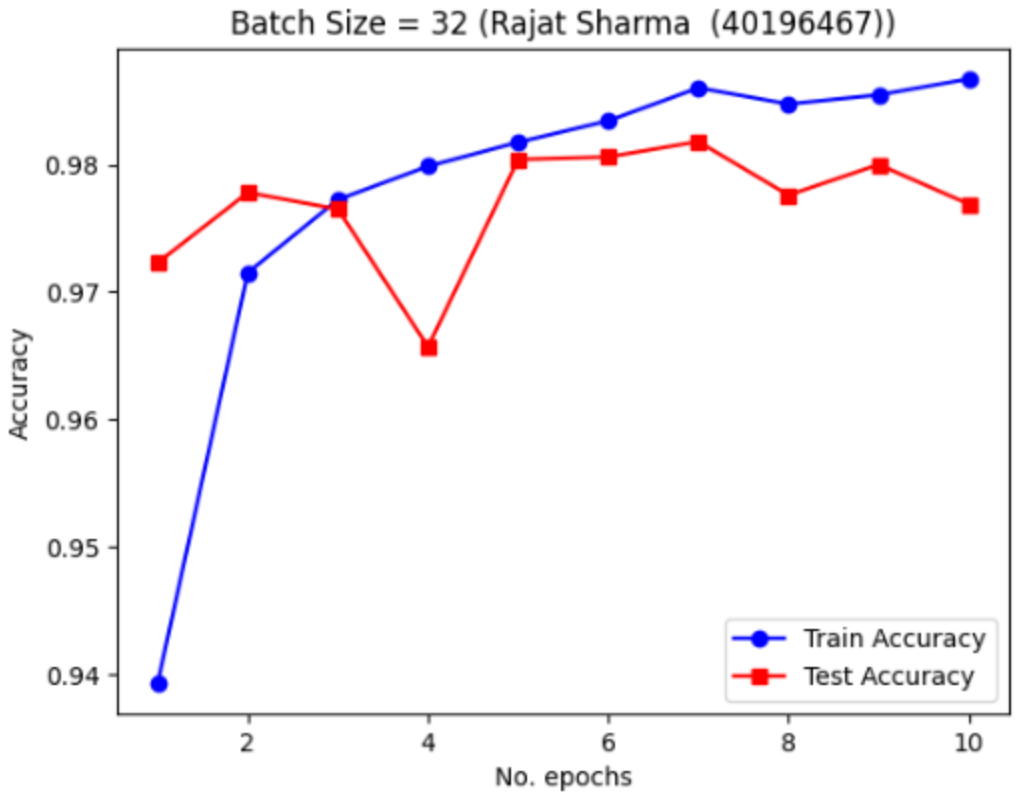
c) The bias and weight parameters printed in the code refer to the learnable parameters of the batch normalization layer. Batch normalization is a technique used to normalize the inputs of a layer, making the training process more stable and accelerating convergence. The parameters have the following roles:

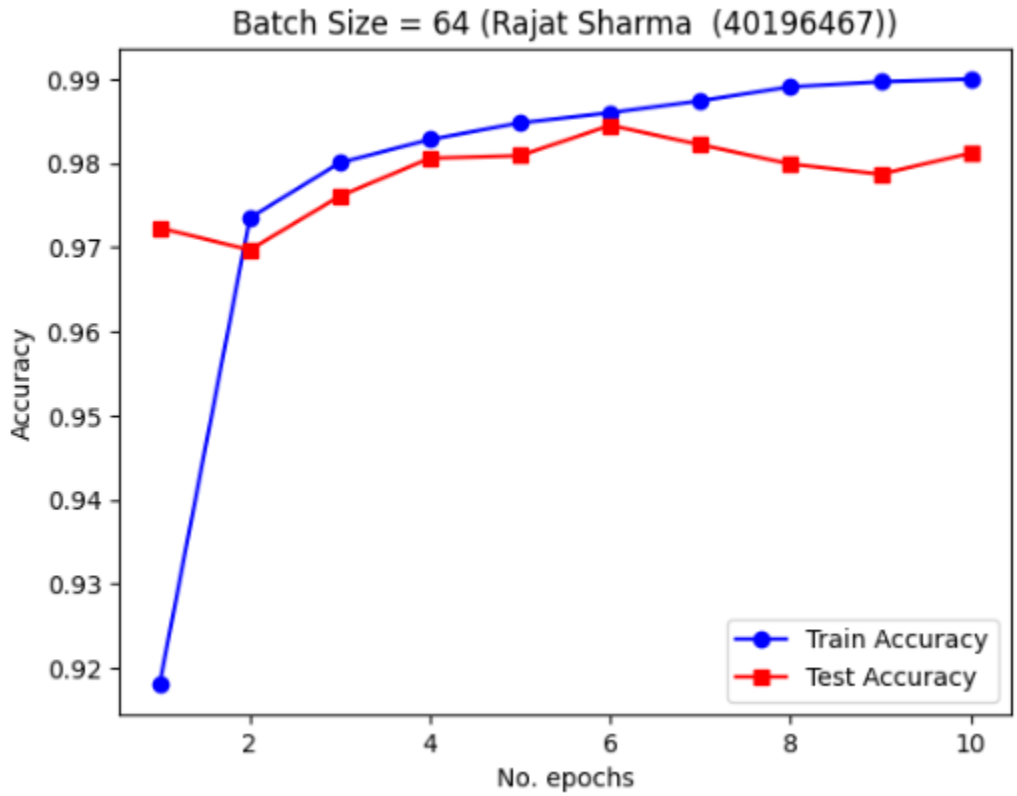
Bias parameters (batch_norm_layer.bias.data): These represent learned bias values for each output channel of the batch normalization layer. They allow shifting the normalized values to account for mean shifts in the data.

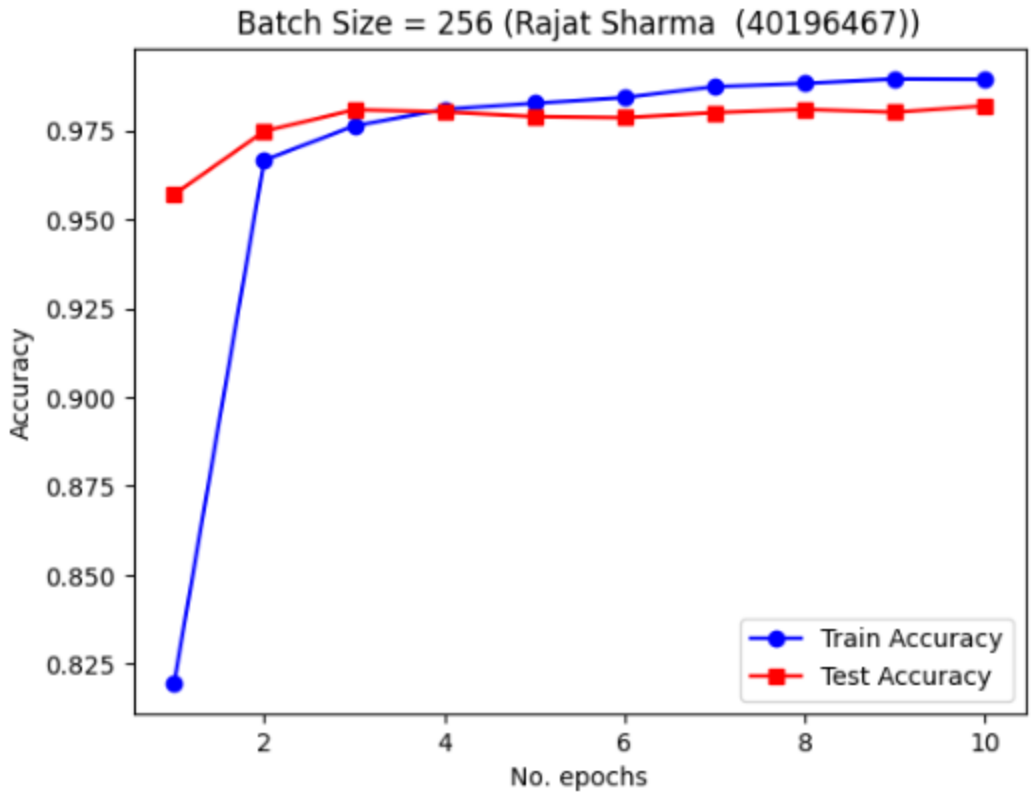
Weight parameters (batch_norm_layer.weight.data): These represent learned scaling factors for each output channel of the batch normalization layer. They allow adjusting the normalized values to account for variance shifts in the data.

By using these bias and weight parameters, batch normalization ensures consistent mean and variance in the inputs to the next layer, thereby aiding in training deep neural networks. The bias and weight parameters are updated during the training process through backpropagation and gradient descent, allowing the model to learn the appropriate scaling and shifting factors for the normalized inputs.

d)


Batch Size = 32 (Rajat Sharma (40196467))
Accuracy
No. epochs
Train Accuracy
Test Accuracy
0.94
0.95
0.96
0.97
0.98
2
4
6
8
10


Batch Size = 64 (Rajat Sharma (40196467))
Accuracy
No. epochs
Train Accuracy
Test Accuracy
0.92
0.93
0.94
0.95
0.96
0.97
0.98
0.99
2
4
6
8
10



e) By comparing the plot with batch size 1 and batch size 256, we can observe a significant difference in performance.

With batch size 1, the model shows poor performance with low training accuracy and limited ability to generalize. This is because with a small batch size, the model updates its parameters based on only one sample at a time, resulting in noisy and unstable updates. The model struggles to capture meaningful patterns in the data and fails to learn effectively from the training set. As a result, the test accuracy is also low, indicating poor generalization to unseen data.

On the other hand, with batch size 256, the model demonstrates improved performance. The larger batch size allows the model to learn from a more representative and diverse set of samples in each update. This results in more stable gradient estimates and smoother convergence, leading to better learning and higher training accuracy. The model also achieves higher test accuracy, indicating improved generalization.

The performance issue with batch size 1 can be attributed to the high variance in gradient estimates. With such a small batch size, the gradients computed for each example have a higher chance of deviating significantly from the true underlying gradients. This can lead to

noisy updates and hinder the learning process. Additionally, the small batch size causes the model to make frequent parameter updates, which may slow down convergence and make it harder for the model to find the optimal solution.

f) In the plot with batch size 256, the training accuracy consistently increases with each epoch, reaching a relatively high value by the end of training. However, there is a noticeable gap between the training accuracy and the test accuracy.

The difference between the training and test accuracy can indicate overfitting. Overfitting occurs when the model performs well on the training data but fails to generalize to unseen data. In this case, the model may have memorized the training examples instead of learning generalizable patterns.

To mitigate the issue, several approaches can be considered:

Regularization: Apply regularization techniques such as L1 or L2 regularization to add a penalty term to the loss function. This discourages the model from relying too heavily on certain features and encourages more generalization.

Dropout: Implement dropout regularization by randomly dropping neurons during training. This forces the model to learn more robust and diverse representations, reducing over-reliance on specific connections.

Data augmentation: Augment the training data by applying random transformations such as rotations, translations, or flips. This increases the diversity of the training set and helps the model learn more invariant and general features.

Early stopping: Monitor the test accuracy during training and stop training when the test accuracy starts to decrease or plateau. This prevents overfitting by avoiding unnecessary iterations.

To change the structure of the CNN to mitigate overfitting, the following modifications can be considered:

Reduce model complexity: Simplify the model by reducing the number of layers or the number of parameters. This reduces the risk of overfitting by limiting the model's capacity to memorize the training data.

Increase regularization: Add additional regularization layers, such as dropout or batch normalization, to regularize the model's parameters and improve generalization.

Ensemble learning: Use ensemble techniques to combine multiple models, which can help in reducing overfitting and improving generalization performance.

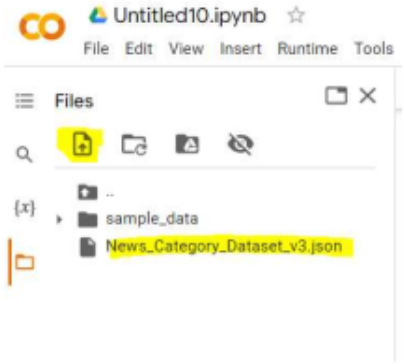
It's important to experiment and find the right balance between model complexity, regularization techniques, and hyperparameter tuning to mitigate overfitting and improve the model's performance on unseen data.

Question 6 [10pts] News Dataset

We will use the dataset of news category of HuffPost available here on Kaggle: https://www.kaggle.com/datasets/rmisra/news-category-dataset

Please reference the dataset in your submissions. If you use Python or your notebook locally, please extract and put the file *News_Category_Dataset_v3.json* in the same folder as the python file or notebook. If you use Google Colab, you should first upload the file to Google Colab.

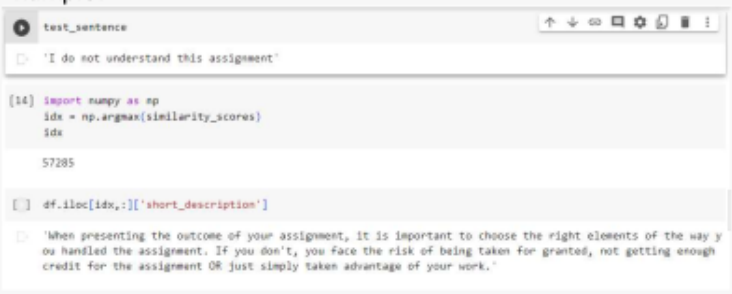
In the provided notebook, some steps are already written for you. You complete the code and answer the following questions.



a) [3p] Create a bag of words using `CountVectorizer` Sickit-Learn from the headline column (as we did in the labs). What is the size of the corpus (no. of words) if you do not remove stop words? What is the size if we remove English stop words? What are the first three most frequent words in this corpus?

b) [5pt] Remove the stop words in the `CountVectorizer`. As we did in the lab, calculate the tfidf weights of every sentence in the dataset. Then, provide your own sentence, find the tfidf weight of your sentence, and then find which sentence in the dataset is the most similar to yours, based on Cosine Similarity. Take a snapshot of your own sentence and the one in the dataset and the Cosine Similarity.

Example:



c) [2pts] Use Spacy to analyze the name entities (NE) in the new headline (the one which was the most similar to yours) (the same as we did in the lab]. Provide a snapshot of the answer.

a)

Without Removing Stop Words: 75726

In [4]:
```
from sklearn.feature_extraction.text import CountVectorizer
# Creating Bag of Words
vect = CountVectorizer()
x = vect.fit_transform(df['short_description'])
feat_names = vect.get_feature_names_out()
corpus_size_no_stopwords = len(feat_names)
print("Corpus Size(Without removing Stop Words): ",corpus_size_no_stopwords)
```

```
Corpus Size(Without removing Stop Words):  75726
```

With Removing Stop Words: 75420

In [5]:
```
vect = CountVectorizer(stop_words='english')
x = vect.fit_transform(df['short_description'])
feat_names = vect.get_feature_names_out()
corpus_size_no_stopwords = len(feat_names)
print("Corpus Size(With removing Stop Words): ",corpus_size_no_stopwords)
```

```
Corpus Size(With removing Stop Words):  75420
```

first three most frequent words: kyran, loden, loder

In [6]:
```
word_freq = x.sum(axis=0)
most_freq =word_freq.argsort()[0,:][::-1]
first_three =[feat_names[index] for index in most_freq.tolist()[0][:3]]
print("First Three most freq words: ",first_three)
```

```
First Three most freq words:  ['kyga', 'logger', 'loggins']
```

b)

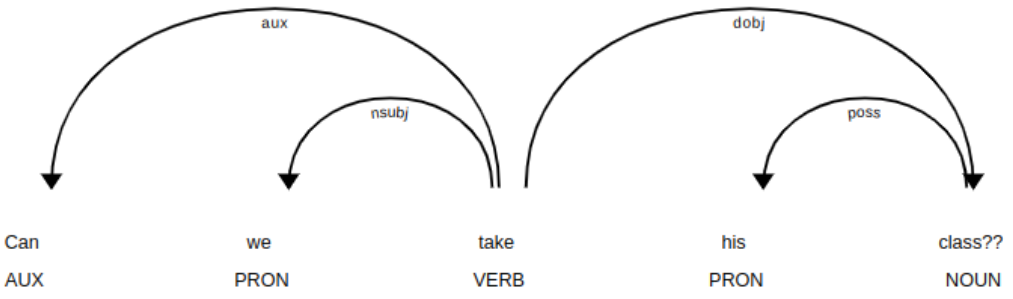
In [8]:
```
# Sample code. count_vect is the name of your Countvectorizor
'''
test_sentence = "I do not understand this assignment"
X_tst = count_vect.transform([test_sentence])
X_tst.shape
'''
# X_counts = vect.fit_transform(x)
# tfidf_transformer.fit(X_counts)
test_sentence = "I am not really enjoying my AI class"
X_tst = vect.transform([test_sentence]).toarray()
```

In [9]:
```
'''
tfidf_vectors_test = tfidf_transformer.transform(X_tst)
tfidf_vectors_test
'''
tfidf_vectors_test = tfidf_transformer.transform(X_tst)

from sklearn.metrics.pairwise import cosine_similarity
similarity_scores = cosine_similarity(tfidf_vectors, tfidf_vectors_test)
most_similar_index = similarity_scores.argmax()
most_similar_sentence = df.loc[most_similar_index, 'short_description']
print("Test Sentence:", test_sentence)
print("Most Similar Sentence:", most_similar_sentence)
print("Cosine Similarity:", similarity_scores[most_similar_index])
```

```
Test Sentence: I am not really enjoying my AI class
Most Similar Sentence: Can we take his class??
Cosine Similarity: [0.4292114]
```

c)



Question 7) [5p] Compare CONV architectures.
For a network with an input layer of size (64x64x3) (i.e., RGB color images of size 64x64), compare the
following two architectures:
-
-
Three consecutive convolution layers each with 3x3 filters, after the input layer
One convolution layer with 7x7 filters, after the input layer
Compare output volume size, feature detection capacity, and number of parameters.
Which one do you recommend and why?
Hint: (Due to parameter sharing in convolutional networks and the fact that the same image size is input to
both architectures, the size of the images is not needed to be considered)

To compare the two architectures, let's analyze the output volume size, feature detection capacity, and number of parameters for each case:

Three consecutive convolution layers with 3x3 filters:

Each convolution layer will reduce the spatial dimensions (width and height) of the input by two due to the 3x3 filter size and a stride of 1 (assuming no padding is used).
The output volume size after each convolution layer will be as follows:
First convolution layer: (62x62x3)
Second convolution layer: (60x60x3)
Third convolution layer: (58x58x3)
The total number of parameters for this architecture can be calculated as follows:
First convolution layer: (3x3x3)x3 + 3 = 84 parameters (3x3x3 filters with 3 input channels and 3 biases)
Second convolution layer: (3x3x3)x3 + 3 = 84 parameters
Third convolution layer: (3x3x3)x3 + 3 = 84 parameters
Total parameters: 252 parameters
One convolution layer with 7x7 filters:

The convolution layer will reduce the spatial dimensions of the input by six due to the 7x7 filter size and a stride of 1 (assuming no padding is used).
The output volume size after the convolution layer will be (58x58x3).
The total number of parameters for this architecture can be calculated as follows:
Convolution layer: (7x7x3)x3 + 3 = 444 parameters (7x7x3 filters with 3 input channels and 3 biases)
Recommendation:
Based on the comparison, I would recommend the architecture with three consecutive convolution layers with 3x3 filters. Here's why:

Output volume size: Both architectures have the same output volume size of (58x58x3), so there is no difference in terms of spatial dimensions.

Feature detection capacity: The architecture with three consecutive convolution layers has more layers, allowing for deeper feature extraction. Each layer can learn different features from the input image, leading to a potentially richer representation of the data. The single convolution layer with 7x7 filters may have a limited capacity to capture complex patterns compared to the deeper architecture.

Number of parameters: The architecture with three consecutive convolution layers has fewer parameters (252 parameters) compared to the single convolution layer with 7x7 filters (444 parameters). Having fewer parameters can make the model more computationally efficient, reducing memory requirements and training time.

Overall, the architecture with three consecutive convolution layers offers a deeper network with fewer parameters, potentially enabling better feature extraction capabilities.