COMP4211 PA2 Report

Taewoo Kim

2021 3873

tkimae@connect.ust.hk

1. Build a CNN Classifier from scratch

a) Screen shot of the code

```
class CnnClassifier(nn.Module):
  def __init__(self, n_hidden):
    super(CnnClassifier, self).__init__()

    # in_data size: (batch_size, 1, 28, 28)
    self.cnn_layers = nn.Sequential(
      nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3, stride=1, padding=0),
      nn.ReLU(),
      nn.Conv2d(in_channels=4, out_channels=8, kernel_size=3, stride=2, padding=0),
      nn.ReLU(),
      nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, stride=2, padding=0),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=3, stride=1, padding=0),
      nn.Conv2d(in_channels=16, out_channels=32,kernel_size=3,stride = 1, padding=0),
      nn.Sigmoid()
    )
    # linear layers transforms flattened image features into logits before the softmax layer
    self.linear = nn.Sequential(
      nn.Linear(32, n_hidden),
      nn.ReLU(),
      nn.Linear(n_hidden, 47)
    )

    self.softmax = nn.Softmax(dim=1)
    self.loss_fn = nn.CrossEntropyLoss(reduction='sum')

  def forward(self, in_data):
    img_features = self.cnn_layers(in_data).view(in_data.size(0), 32)# in_data.size(0) == batch_size
    logits = self.linear(img_features)
    return logits

  def loss(self, logits, labels):
    preds = self.softmax(logits) # size (batch_size, 10)
    return self.loss_fn(preds, labels) / logits.size(0) # divided by batch_size

  def top1_accuracy(self, logits, labels):
    # get argmax of logits along dim=1 (this is equivalent to argmax of predicted probabilites)
    predicted_labels = torch.argmax(logits, dim=1, keepdim=False) # size (batch_size,)
    n_corrects = predicted_labels.eq(labels).sum(0) # sum up all the correct predictions
    return int(n_corrects) / float(logits.size(0)) * 100. # in percentage
  #reference: https://discuss.pytorch.org/t/imagenet-example-accuracy-calculation/7840
  def top3_accuracy(self, logits, labels,topk=(3,)):
    maxk = max(topk)
    batch_size = labels.size(0)

    _, pred = logits.topk(maxk, 1, True, True)
    pred = pred.t()
    correct = pred.eq(labels.view(1, -1).expand_as(pred))

    res = []
    for k in topk:
        correct_k = correct[:k].view(-1).float().sum(0)
        res.append(correct_k.mul_(100.0 / batch_size))
    return res
#
```

Reference:

1. comp4211 tutorial6
2. https://discuss.pytorch.org/t/imagenet-example-accuracy-calculation/7840

b) Holdout Validation

Fixed parameters:

batch size: 32

number of epochs: 10

Parameters to be tested:

1. hidden layer = 32, optimizer = ADAM, learning rate = 0.001
2. hidden layer = 32, optimizer = SGD, learning rate = 0.1
3. hidden layer = 32, optimizer = SGD, learning rate = 0.01
4. hidden layer = 64, optimizer = ADAM, learning rate = 0.001
5. hidden layer = 64, optimizer = SGD, learning rate = 0.1
6. hidden layer = 64, optimizer = SGD, learning rate = 0.01

Train data has been randomly divided into training and validation set with the function: **train_test_split()** from sklearn

```
def splitvalidation(trainingd):
  train_ds,eval_ds = train_test_split(trainingd,test_size=0.2,random_state=42)
  return train_ds, eval_ds
```

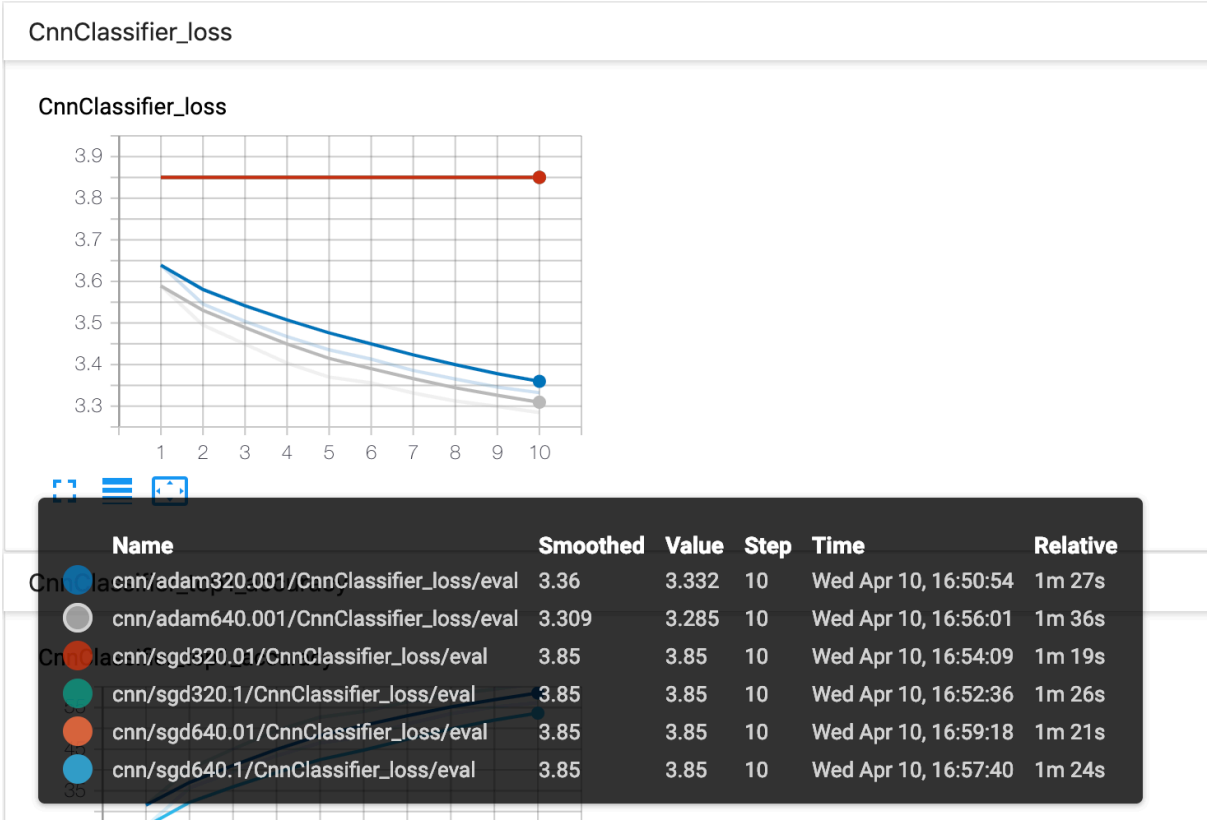
With the divided data, each set of parameters were ran for 10 epochs and best parameter set with the lowest cross entry loss was selected. Cross entropy loss for each parameter set was obtained by recording the lowest loss among 10 epochs(optimal loss).

Results are shown below:

```
validation for cnnClassifer
hidden_layer-32, optimizer - adam, learning rate-0.001 : train_loss3.3160686183120704  eval_loss3.332328377523683
hidden_layer-32, optimizer - sgd, learning rate-0.1 : train_loss3.850087045536215  eval_loss3.850153565044461
hidden_layer-32, optimizer - sgd, learning rate-0.01 : train_loss3.8501717814196206  eval_loss3.8501374960670356
hidden_layer-64, optimizer - adam, learning rate-0.001 : train_loss3.276669450806267  eval_loss3.2846206973751264
hidden_layer-64, optimizer - sgd, learning rate-0.1 : train_loss3.8501038551330566  eval_loss3.8501525932532314
hidden_layer-64, optimizer - sgd, learning rate-0.01 : train_loss3.8501514758020186  eval_loss3.850156218809922
parameters chosen are: 64 adam 0.001
```

As you can see from the output screen, parameter set with hidden layer: 64, optimizer: adam, and learning rate: 0.001 was picked as the best parameter, since it has the lowest evaluation loss: 3.2846206973751264.

In order to visualize the results of holdout validation, I have attached the screen shot of the cross entropy loss plotted for each epoch. Visualization also shows that classifier with optimizer: adam, hidden_layer:64, and learning rate: 0.001 shows the best performance. This supports the selection of the parameter.



c) Testing phase

With the optimal parameters picked in the previous stage, the model was trained with full training data set and tested with test data set. However, for better performance, number of epochs was increased to 50.

Parameter setting:

batch size: 32

number of epochs: 50

hidden layer = 64

optimizer = ADAM,

learning rate = 0.001

This procedure was repeated for 5 times with shuffling train data. For each run, optimal values for cross entropy loss, top 1 accuracy and top 3 accuracy were calculated, and mean and standard deviation were calculated. For each iteration, train data was shuffled by setting torch.utils.data.DataLoader()'s shuffle parameter as True.

Output screen is as shown as below:

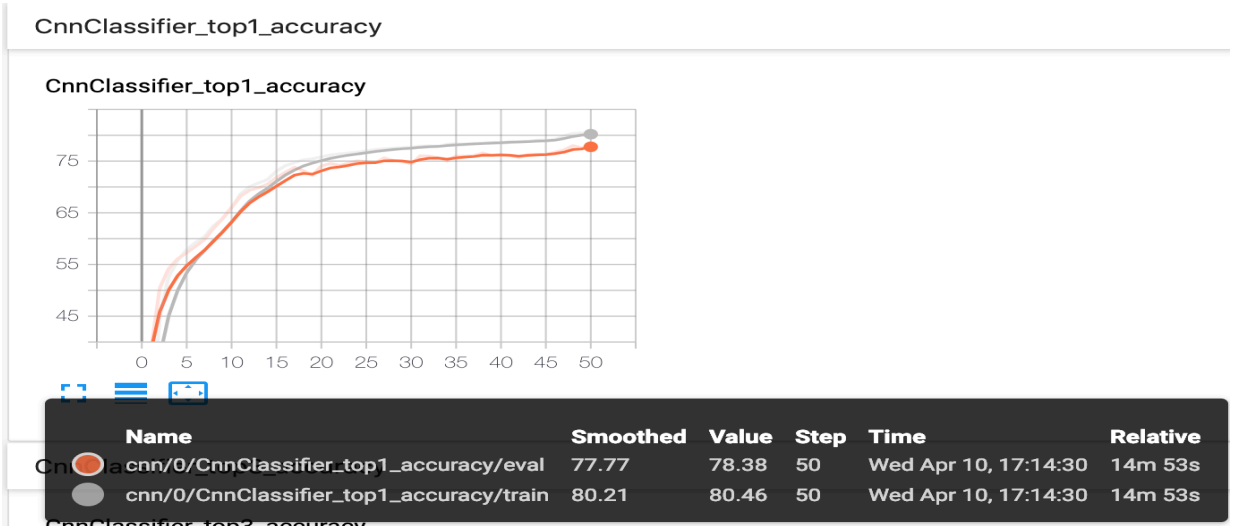
```
Minimum train loss mean is 3.1289192221737196 and standard deviation is 0.026861690840993997
Minimum evaluation loss mean is 3.1519691492747337 and standard deviation is 0.029371315612370764
Maximum train top 1 accuracy Mean is 75.9798632218845 and standard deviation is 2.696102883843837
Maximum evaluation top 1 accuracy Mean is 73.62864077669903 and standard deviation is 2.9483975645568
Maximum train top 3 accuracy Mean is 84.17705167173251 and standard deviation is 3.332596825025465
Maximum evaluation top 3 accuracy Mean is 83.48149271844662 and standard deviation is 3.3859541038261893
```

This output screen shows the mean and standard deviation for train loss, evaluation loss, top 1 accuracy for training, and evaluation, and top 3 accuracy for training and evaluation.

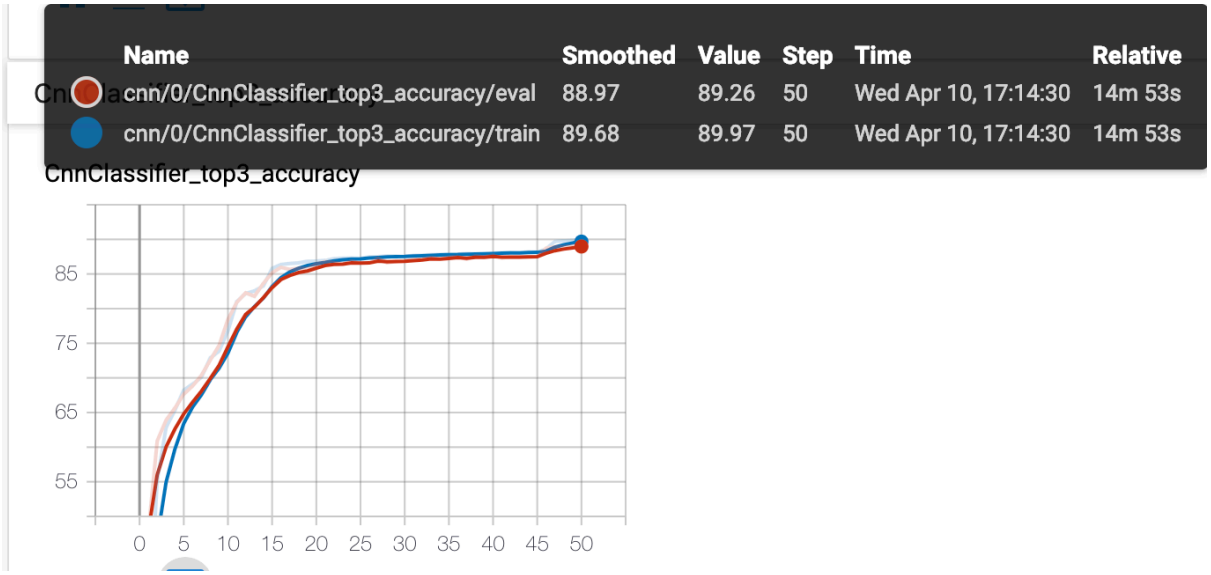
For the presentation of learning curve, I have selected the first iteration results and the curve is shown below. Smoothing option was selected as 0.6 for the better visualization of graph:

a) Cross entropy loss learning curve for both train and test data

b) Top 1 accuracy learning curve for both train and test data



c) Top 3 accuracy learning curve for both train and test data



These results including mean, standard deviation of metrics and time taken to run will be used to compare two classifiers in the section 3.

2. Build a CNN Classifier with pretrained encoder

a) Code

```
class pretrainedCNN(nn.Module):
  def __init__(self, n_hidden):
    super(pretrainedCNN, self).__init__()
    self.cnn_layers = torch.load('pretrained_encoder.pt')['model']
    self.linear = nn.Sequential(
      nn.Linear(32, n_hidden),
      nn.ReLU(),
      nn.Linear(n_hidden, 47)
    )

    self.softmax = nn.Softmax(dim=1)
    self.loss_fn = nn.CrossEntropyLoss(reduction='sum')
    #transferring weights from pretrained encoder
    #reference: https://discuss.pytorch.org/t/copy-weights-only-from-a-networks-parameters/5841
    beta = 0.5 #The interpolation parameter
    params1 = torch.load('pretrained_encoder.pt')['model'].named_parameters()
    params2 = self.named_parameters()
    dict_params2 = dict(params2)
    for name1, param1 in params1:
      if name1 in dict_params2:
        dict_params2[name1].data.copy_(beta*param1.data + (1-beta)*dict_params2[name1].data)

    self.load_state_dict(dict_params2)
  def forward(self, in_data):
    img_features = self.cnn_layers(in_data).view(in_data.size(0), 32)# in_data.size(0) == batch_size
    logits = self.linear(img_features)
    return logits

  def loss(self, logits, labels):
    preds = self.softmax(logits) # size (batch_size, 10)
    return self.loss_fn(preds, labels) / logits.size(0) # divided by batch_size
  def top1_accuracy(self, logits, labels):
    # get argmax of logits along dim=1 (this is equivalent to argmax of predicted probabilites)
    predicted_labels = torch.argmax(logits, dim=1, keepdim=False) # size (batch_size,)
    n_corrects = predicted_labels.eq(labels).sum(0) # sum up all the correct predictions
    return int(n_corrects) / float(logits.size(0)) * 100.
  #reference: https://discuss.pytorch.org/t/imagenet-example-accuracy-calculation/7840
  def top3_accuracy(self, logits, labels,topk=(3,)):
    maxk = max(topk)
    batch_size = labels.size(0)

    _, pred = logits.topk(maxk, 1, True, True)
    pred = pred.t()
    correct = pred.eq(labels.view(1, -1).expand_as(pred))

    res = []
    for k in topk:
        correct_k = correct[:k].view(-1).float().sum(0)
        res.append(correct_k.mul_(100.0 / batch_size))
    return res
```


Reference:

1. comp4211 tutorial6
2. https://discuss.pytorch.org/t/imagenet-example-accuracy-calculation/7840
3. https://discuss.pytorch.org/t/copy-weights-only-from-a-networks-parameters/5841

b) Holdout Validation

Fixed parameters:

batch size: 32

number of epochs: 10

Parameters to be tested:

1. hidden layer = 32, optimizer = ADAM, learning rate = 0.001
2. hidden layer = 32, optimizer = SGD, learning rate = 0.1
3. hidden layer = 32, optimizer = SGD, learning rate = 0.01
4. hidden layer = 64, optimizer = ADAM, learning rate = 0.001
5. hidden layer = 64, optimizer = SGD, learning rate = 0.1
6. hidden layer = 64, optimizer = SGD, learning rate = 0.01

Train data has been randomly divided into training and validation set with the function: **train_test_split()** from sklearn

```
def splitvalidation(trainingd):
  train_ds,eval_ds = train_test_split(trainingd,test_size=0.2,random_state=42)
  return train_ds, eval_ds
```

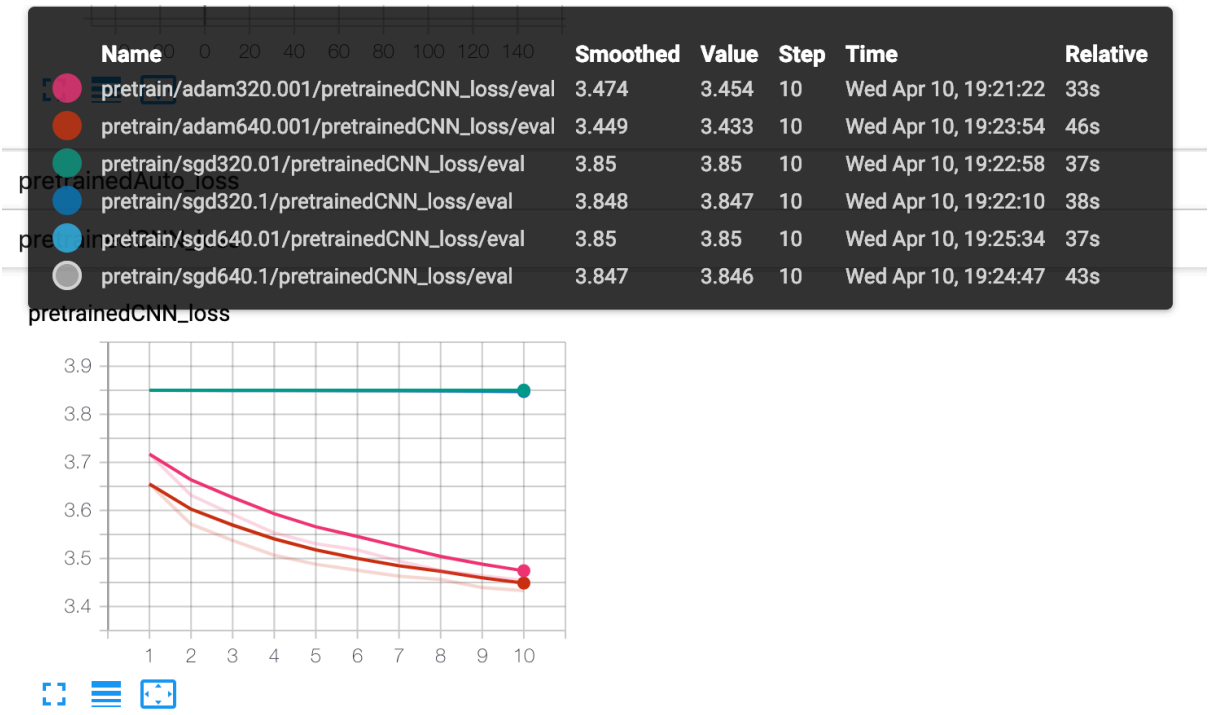
With the divided data, each set of parameters were ran for 10 epochs and best parameter set with the lowest cross entry loss was selected. Cross entropy loss for each parameter set was obtained by recording the lowest loss among 10 epochs(optimal loss).

Results are shown below:

```
hidden_layer-32, optimizer - adam, learning rate-0.001 : train_loss3.4524870074628695  eval_loss3.4540573180990015
hidden_layer-32, optimizer - sgd, learning rate-0.1 : train_loss3.847366875247028  eval_loss3.8469467250047122
hidden_layer-32, optimizer - sgd, learning rate-0.01 : train_loss3.849948032467561  eval_loss3.8499973639166463
hidden_layer-64, optimizer - adam, learning rate-0.001 : train_loss3.432827696430647  eval_loss3.432939998284662
hidden_layer-64, optimizer - sgd, learning rate-0.1 : train_loss3.846494455649135  eval_loss3.8457552299673434
hidden_layer-64, optimizer - sgd, learning rate-0.01 : train_loss3.8500206220476096  eval_loss3.8500277764166744
parameters chosen are: 64 adam 0.001
```

As you can see from the output screen, parameter set with hidden layer: 64, optimizer: adam, and learning rate: 0.001 was picked as the best parameter, since it has the lowest evaluation loss : 3.432939998284662.

In order to visualize the results of holdout validation, I have attached the screen shot of the cross entropy loss plotted for each epoch. Visualization also shows that classifier with optimizer: adam, hidden_layer:64, and learning rate: 0.001 shows the best performance. This supports the selection of the parameter.



c) Testing Phase

With the optimal parameters picked in the previous stage, the model was trained with full training data set and tested with test data set. However, for better performance, number of epochs was increased to 50.

Parameter setting:

batch size: 32

number of epochs: 50

hidden layer = 64

optimizer = ADAM,

learning rate = 0.001

This procedure was repeated for 5 times with shuffling train data. For each run, optimal values for cross entropy loss, top 1 accuracy and top 3 accuracy were calculated, and mean and standard deviation were calculated. For each iteration, train data was shuffled by setting torch.utils.data.DataLoader()'s shuffle parameter as True.

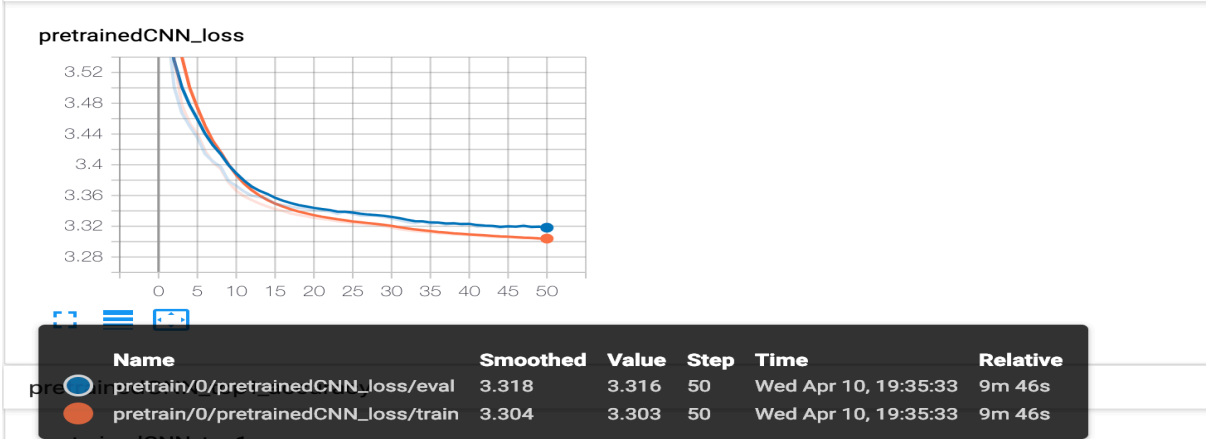
Output screen is as shown as below:

```
train loss mean is 3.3231959520357366 and standard deviation is 0.017195453456792933
evaluation loss mean is 3.3354663006310323 and standard deviation is 0.01744540496409072
train top 1 accuracy Mean is 57.06079027355622 and standard deviation is 1.7751042878828314
evaluation top 1 accuracy Mean is 55.747876213592235 and standard deviation is 1.7563003823494796
train top 3 accuracy Mean is 67.19946808510637 and standard deviation is 2.459346900757243
evaluation top 3 accuracy Mean is 66.2302791262136 and standard deviation is 2.4783496640663056
```

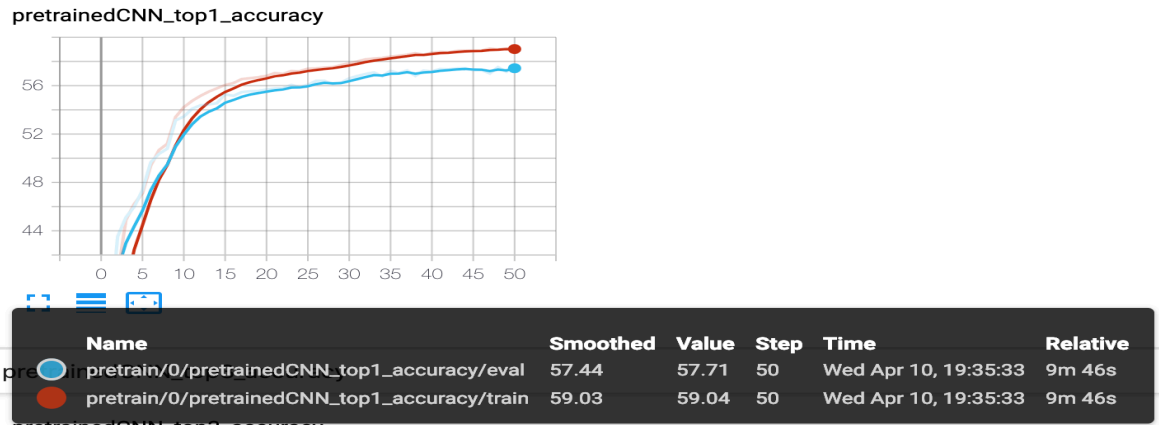
This output screen shows the mean and standard deviation for train loss, evaluation loss, top 1 accuracy for training, and evaluation, and top 3 accuracy for training and evaluation.

For the presentation of learning curve, I have selected the first iteration results and the curve is shown below. Smoothing option was selected as 0.6 for the better visualization of graph:

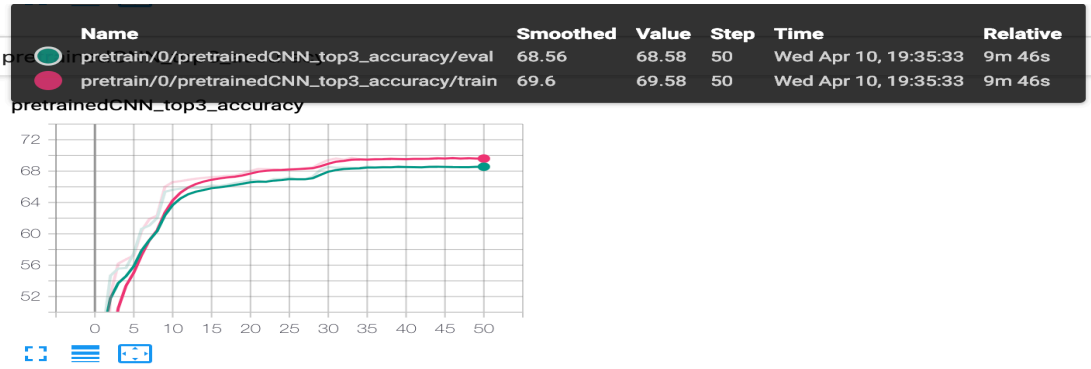
a) Cross entropy loss learning curve for both train and test data



b) Top 1 accuracy learning curve for both train and test data



c) Top 3 accuracy learning curve for both train and test data



These results including mean, standard deviation of metrics and time taken to run will be used to compare two classifiers in the section 3.

3) Compare performance of CNN classifiers

a) Time taken

Time taken for each classifiers differ a lot.
As you can see from the learning curves presented on the testing phases of both CNN classifier from scratch and pretrained CNN classifier is 14minutes 53seconds and 9minutes 46seconds respectively. So there is 5minutes 7seconds of difference in running time for running 50 epochs. If epochs were increased further, bigger difference in time taken is expected.

b) Number of epochs

For the fair comparison, number of epochs ran for both classifiers are the same: 50 epochs

c) Cross entropy loss

for the comparison of cross entropy loss, I am going to use mean and standard deviation calculated on testing set for each classifiers.
CNN from scratch:
Mean= 3.1519691492747337
Std= 0.029371315612370764
Pretrained CNN:
Mean = 3.3354663006310323
Std = 0.01744540496409072

From these values, we can see that CNN classifier built from scratch has the lower mean value for cross entropy loss than pretrained CNN classifier. The difference is about 0.18. However, pretrained CNN showed the more consistency since std is lower than CNN from scratch's classifier.

d) Top 1 accuracy

CNN from scratch:
Mean= 73.62864077669903
Std= 2.9483975645568
Pretrained CNN:
Mean = 55.747876213592235
Std = 1.7563003823494796

From these values, we can see that CNN classifier built from scratch has the higher top 1 accuracy value than pretrained CNN classifier. The difference is about 17.88 which is quite huge. However, pretrained CNN showed the more consistency since std is much lower than CNN from scratch's classifier.

e) Top 3 accuracy

CNN from scratch:
Mean= 83.4814927184466
Std= 3.3859541038261893
Pretrained CNN:
Mean = 66.2302791262136
Std = 2.478349664066305

From these values, we can see that CNN classifier built from scratch has the higher top 3 accuracy value than pretrained CNN classifier. The difference is about 17.25 which is quite huge. However, pretrained CNN showed the more consistency since std is much lower than CNN from scratch's classifier.

f) Conclusion

Working environment for this assignment was as shown below:

Since my laptop does not have GPU, it does not support CUDA operations. Hence time taken is longer than other computing devices with GPU, so time taken for execution is really important. My experiment took only 50 epochs, but if more number of epochs were taken, it will be more serious matter. Hence in my case(with low computer specification without GPU), using pretrained encoder to build CNN classifier will be the better choice, since the time taken for operation is much shorter. Moreover, it showed better consistency in result when it was repeated for five times(proven by smaller std in metrics).

However, for people with high specification of the computing device(with GPU) it will be a better choice to build CNN from scratch, because although it takes longer time, the top 1 accuracy and the top 3 accuracy shown is way more higher compared to pretrained CNN classifier's.

4) Build CAE with pretrained encoder

a) code:

```
class pretrainedAuto(nn.Module):
  def __init__(self):
    super().__init__()
    model33 = torch.load('pretrained_encoder.pt')['model']
    self.encoder =  model33
    self.decoder = nn.Sequential(
      nn.ConvTranspose2d(in_channels=32, out_channels=16, kernel_size=3, stride=1),
      nn.ReLU(),
      nn.ConvTranspose2d(in_channels=16, out_channels=8, kernel_size=3, stride=1),
      nn.ReLU(),
      nn.ConvTranspose2d(in_channels=8, out_channels=8, kernel_size=3, stride=2),
      nn.ReLU(),
      nn.ConvTranspose2d(in_channels=8, out_channels=4,kernel_size=3, stride=1),
      nn.ReLU(),
      nn.ConvTranspose2d(in_channels=4, out_channels=1, kernel_size=4, stride=2),
      nn.Sigmoid()
      )
    self.encoder = model33
    self.softmax = nn.Softmax(dim=1)

    self.loss_fn = nn.MSELoss(reduction='sum')

    beta = 0.5 #The interpolation parameter
    params1 = torch.load('pretrained_encoder.pt')['model'].named_parameters()
    params2 = self.named_parameters()
    dict_params2 = dict(params2)
    for name1, param1 in params1:
      if name1 in dict_params2:
        dict_params2[name1].data.copy_(beta*param1.data + (1-beta)*dict_params2[name1].data)

    self.load_state_dict(dict_params2)

  def forward(self,in_data):
    img_features = self.encoder(in_data)
    logits = self.decoder(img_features)
    return logits

  def loss(self, logits, labels):
    return self.loss_fn(logits, labels) / logits.size(0)
```

References:

1. https://discuss.pytorch.org/t/imagenet-example-accuracy-calculation/7840
2. https://discuss.pytorch.org/t/copy-weights-only-from-a-networks-parameters/5841

b) Validation phase

Fixed parameters:

batch size: 32

number of epochs: 10

Parameters to be tested:

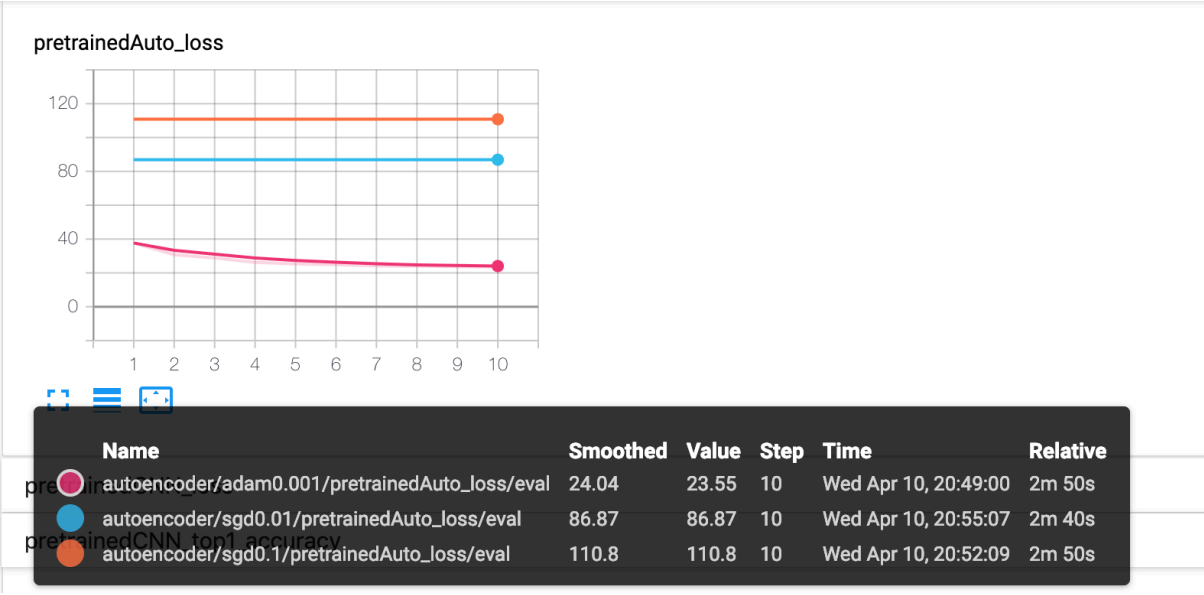
1. optimizer = ADAM, learning rate = 0.001
2. optimizer = SGD, learning rate = 0.1
3. optimizer = SGD, learning rate = 0.01

With each parameter set specified above, 10 epochs had been executed and results are shown below:

```
validation for Convolutional Auto Encoder
optimizer - adam, learning rate-0.001 : train_loss22.502794589242676  eval_loss22.55358330254416
optimizer - sgd, learning rate-0.1 : train_loss111.03904871607261  eval_loss110.80975052916888
optimizer - sgd, learning rate-0.01 : train_loss87.0387103559036  eval_loss86.86300442519698
parameters chosen are: adam 0.001
```

For the selection of the best parameters, first I examined the parameter that had lowest MSE loss during the 10 epochs. So for this step, best parameter selected so far is:

`optimizer - adam, learning rate-0.001 : train_loss22.502794589242676 eval_loss22.55358330254416`



However, to be accurate, I have also checked the reconstructed image with each parameter set to prevent parameter set which cannot reconstruct image well be selected:

Reconstructed image for optimizer = ADAM, learning rate = 0.001

Reconstructed image for optimizer = SGD, learning rate = 0.1

Reconstructed image for optimizer = SGD, learning rate = 0.01

Fortunately, parameter set with the lowest MSE loss had the best quality of image reconstructed. To conclude, parameter set with lowest MSE loss and the best reconstructed image is:

```
optimizer – adam, learning rate–0.001 : train_loss22.502794589242676
eval_loss22.55358330254416
```

and this set has been used for testing phase

c) Testing phase

For the testing phase, I have used parameter sets chosen from testing phase with 20 epochs.

parameters:

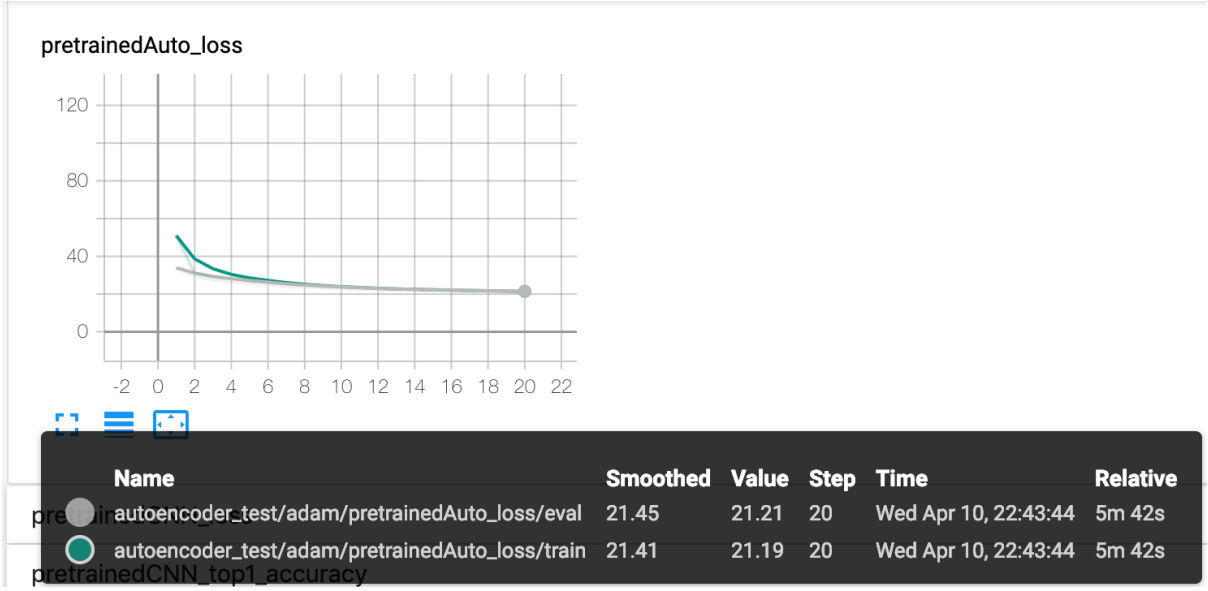
batch size: 32

number of epochs: 20

optimizer = ADAM

learning rate = 0.001

learning curve:



Out of 20 epochs, epoch with the minimum MSE loss was recorded as shown below:

```
minimum value of mse loss is 21.188006204529735 at 20 th epoch
```

Reconstructed images at the epoch with lowest MSE loss is shown below: