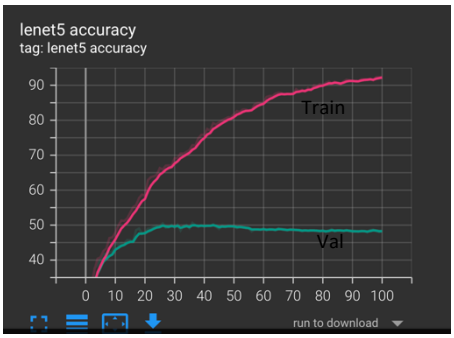
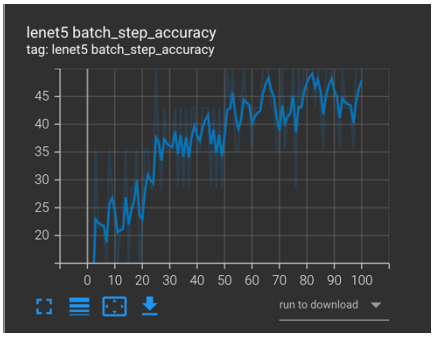
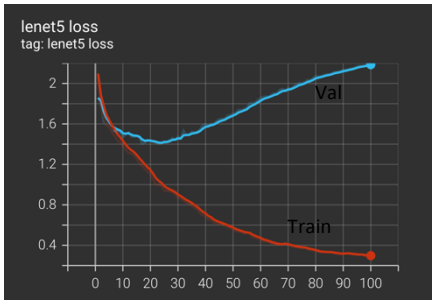
1. For your main experiment setting, show the evolution of training losses and validation losses with multiple steps.

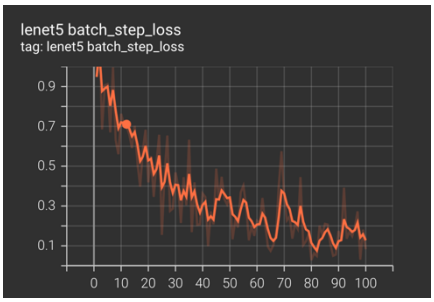
X-axis for all graphs: Epochs

Lenet 5

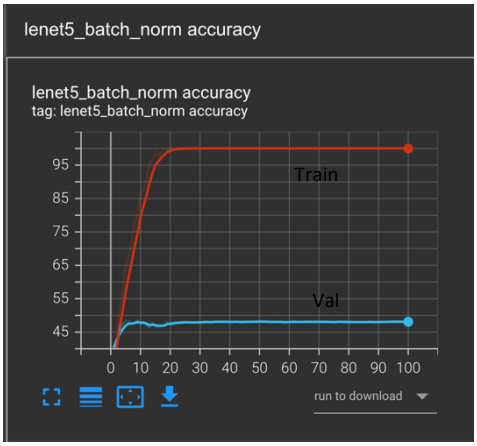


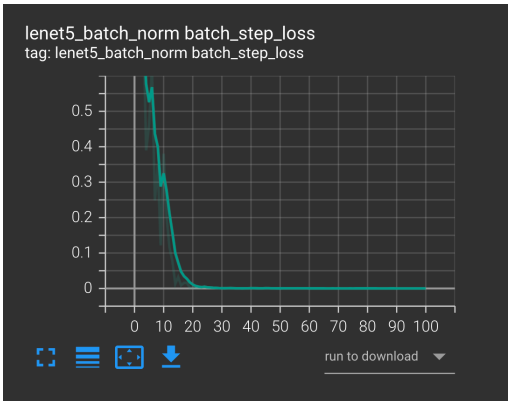


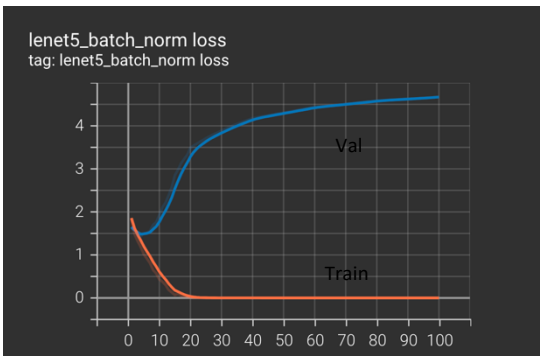


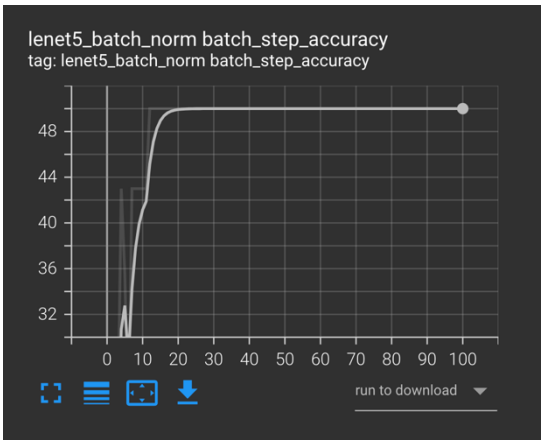


Lenet5 Batch-norm

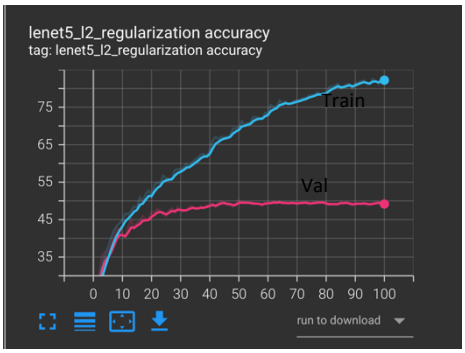


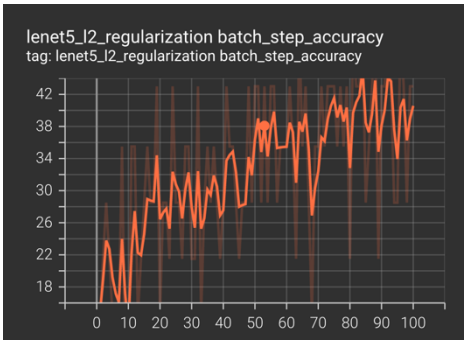


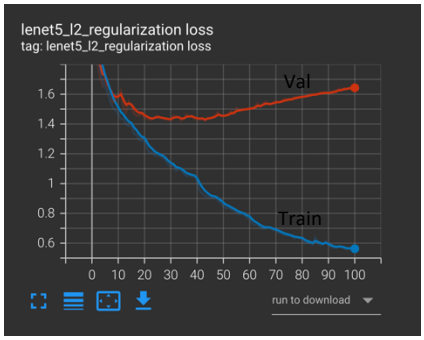


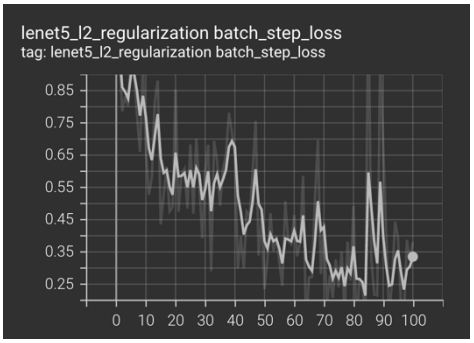


Lenet5 L2 Regularization







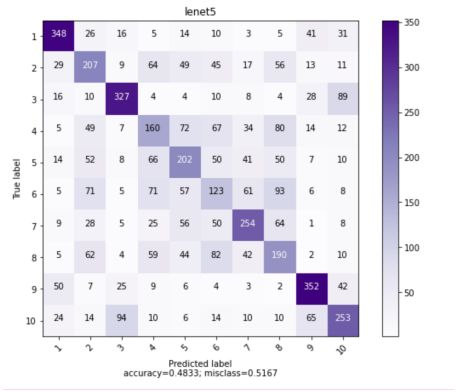


2. Show the confusion matrix and per-class classification accuracy for this setting.

Label and corresponding object

1:"airplane"
2: "bird"
3: "car"
4: "cat"
5: "deer"
6: "dog"
7: "horse"
8: "monkey"
9: "ship"
10: "truck"

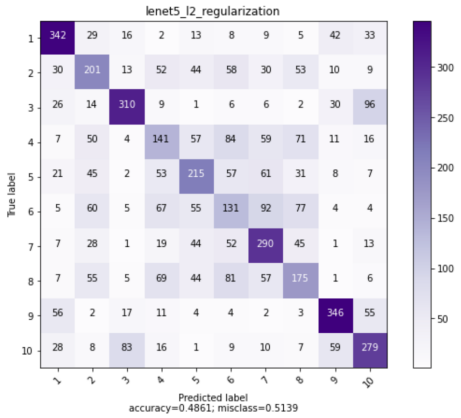
Lenet5



Accuracy by each class for lenet5

	class	accuracy
0	1	69.739479
1	2	41.400000
2	3	65.400000
3	4	32.000000
4	5	40.400000
5	6	24.600000
6	7	50.800000
7	8	38.000000
8	9	70.400000
9	10	50.600000

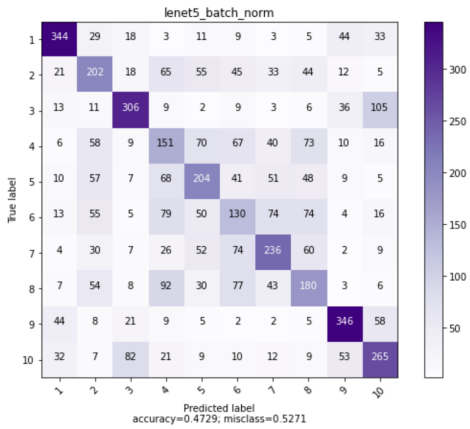
Lenet5_l2_reg



Accuracy by each class for lenet5_l2_regularization

	class	accuracy
0	1	68.537074
1	2	40.200000
2	3	62.000000
3	4	28.200000
4	5	43.000000
5	6	26.200000
6	7	58.000000
7	8	35.000000
8	9	69.200000
9	10	55.800000

Lenet5_batch_norm

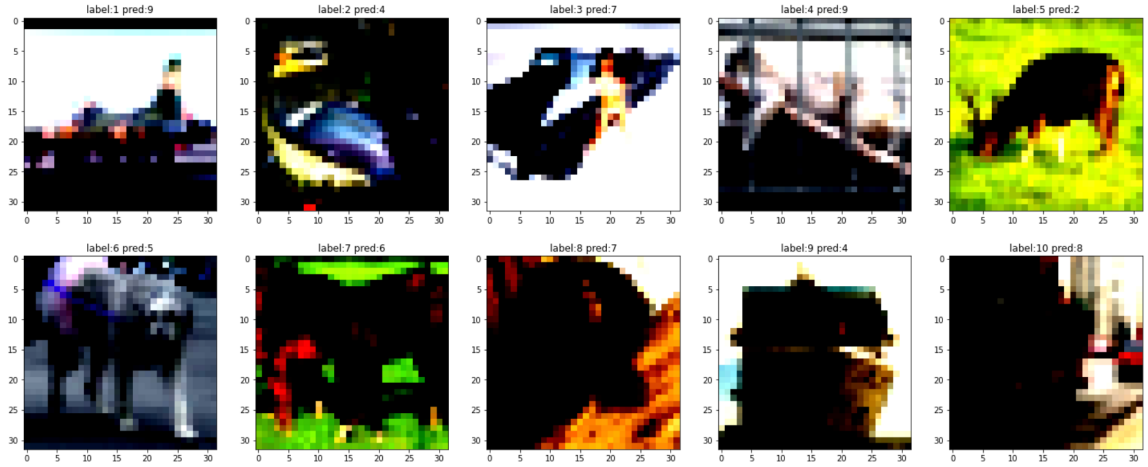


Accuracy by each class for lenet5_batch_norm

	class	accuracy
0	1	68.937876
1	2	40.400000
2	3	61.200000
3	4	30.200000
4	5	40.800000
5	6	26.000000
6	7	47.200000
7	8	36.000000
8	9	69.200000
9	10	53.000000

3. Show some examples of failed cases, with some analysis if feasible.

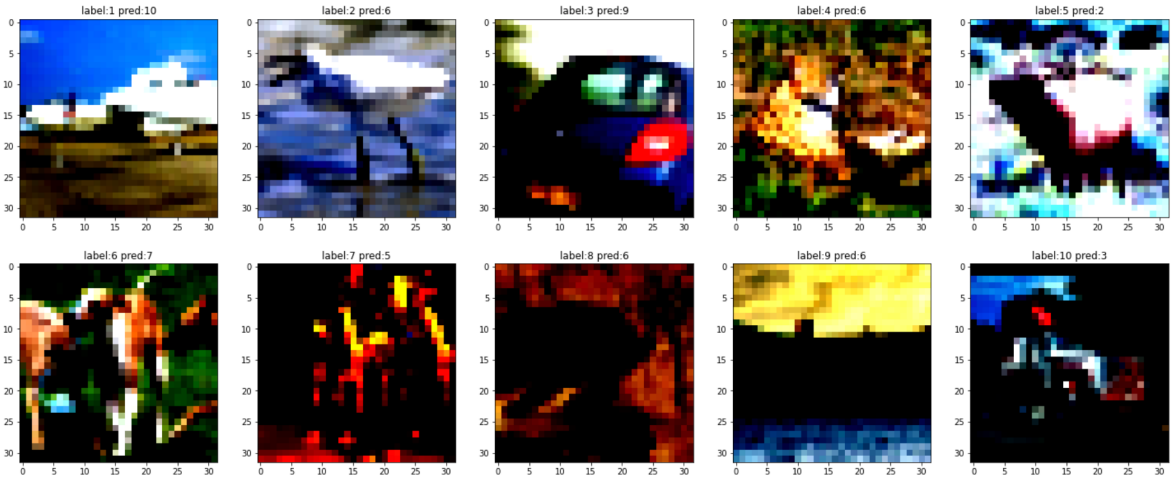
Lenet 5 failed cases:



Pic 1: Label:Airplane(1) and Pred:Ship(9) the model has wrongly classified an airplane as a ship in this pic. This could be because visually at the resized level the object ships and planes can be easily confused with.
As seen in the Confusion Matrix for lenet 5, 50 image belonging to airplane class are wrongly misclassified as ship.

Pic 6: Label:dog (6) and Pred:deer (5) the model has wrongly classified a dog as a deer in this pic. This could be because visually at the resized level if the model is looking for features like 4 legs, it can get confused between dog and deer
As seen in the Confusion Matrix for lenet 5, 57 image belonging to dog class are wrongly misclassified as deer.

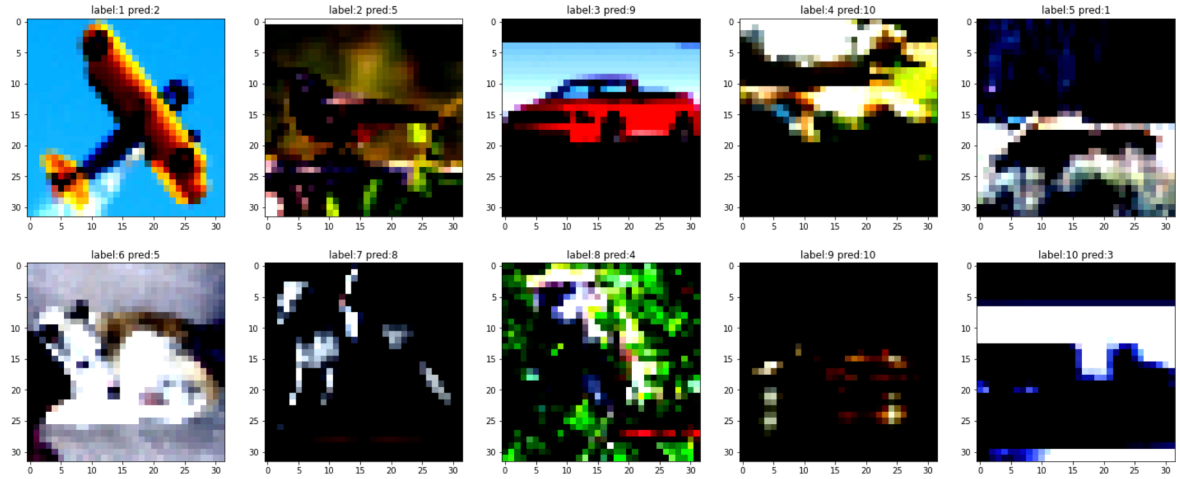
Lenet 5 with l2 regularization failed cases:



Pic 3: Label:Car (3) and Pred:Ship(9) the model has wrongly classified an car as a ship in this pic. This could be because the image of the car is zoomed in and the features look more closer to that of a ship than a car.
As seen in the Confusion Matrix for lenet5_l2_regularization, 30 image belonging to car class are wrongly misclassified as ship.

Pic 7: Label:horse (7) and Pred:deer (5) the model has wrongly classified a horse as a deer in this pic. This could be because visually at the resized level the image is very unclear and is randomly predicting it as a deer.

Lenet 5 with batch norm failed cases:



Pic 1: Label:Airplane (1) and Pred:Bird (3) the model has wrongly classified an Airplane as a bird in this pic. This could be because the model might be looking for features like wings and is wrongly classifying the airplane as a bird due to this.
As seen in the Confusion Matrix for lenet5_batch_norm 44 image belonging to airplane class are wrongly misclassified as birds.

Pic 6: Label:horse (7) and Pred:deer (5) the model has wrongly classified a horse as a deer in this pic. This could be because the dog is sitting like a deer and has the same color as a deer.
As seen in the Confusion Matrix for lenet5_batch_norm 52 image belonging to horse class are wrongly misclassified as deer.

4. Compare your results for the variations with the main experiment setting.

Accuracies of each Model:

Lenet5 Model: 48.32966593318664

Lenet5 l2 Regularization Model: 48.60972194438888

Lenet5 Batch Normalization Model: 47.289457891578316

In comparison model with batch normalization had lesser accuracy than the other two models

LeNet5

November 5, 2021

[60]:
```
import torch
from  torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn.functional as F
import torch.nn as nn
import pandas as pd
from tqdm import tqdm
import os
from PIL import Image
from torch.optim.lr_scheduler import StepLR
from torch.utils.tensorboard import SummaryWriter
from sklearn.metrics import confusion_matrix
import itertools
writer = SummaryWriter()
```

[61]:
```
class stl10_dataset(torch.utils.data.Dataset):

    def __init__(self, text_file, root_dir, transform):
        """
        Args:
            text_file(string): path to text file
            root_dir(string): directory with all train images
        """
        self.name_frame = pd.read_csv(text_file,sep=" ",usecols=range(1))
        self.label_frame = pd.read_csv(text_file,sep=" ",usecols=range(1,2))
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.name_frame)

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir, self.name_frame.iloc[idx, 0])
        image = Image.open(img_name)
        image = self.transform(image)
        labels = self.label_frame.iloc[idx, 0]
```

```
        return image,labels

data_transform = transforms.Compose([ transforms.Resize((32,32)), transforms.
 ↪ToTensor(),
                                     transforms.Normalize(mean=[0,0,0],␣
 ↪std=[1,1,1])])

trainSet = stl10_dataset(text_file ='splits/train.txt', root_dir = '',␣
 ↪transform=data_transform)
stl10_TrainLoader = torch.utils.data.DataLoader(trainSet, batch_size=16,␣
 ↪shuffle=True)
```

[62]:
```
# Find mean and standard deviation for the data.

def compute_mean_std(image_set, image_loader):

    psum    = torch.tensor([0.0, 0.0, 0.0])
    psum_sq = torch.tensor([0.0, 0.0, 0.0])

    # loop through images and find mean
    for inputs,_ in tqdm(image_loader):
        psum    += inputs.sum(axis = [0, 2, 3])
        psum_sq += (inputs ** 2).sum(axis = [0, 2, 3])

    # pixel count in a batch
    count = len(image_set) * 32 * 32

    # mean and std dev calculations
    total_mean = psum / count
    total_var  = (psum_sq / count) - (total_mean ** 2)
    total_std  = torch.sqrt(total_var)

    # print data stats
    print('mean: '  + str(total_mean))
    print('std:  '  + str(total_std))
    return total_mean, total_std

# compute mean and standard deviation
train_mean, train_std = compute_mean_std(trainSet, stl10_TrainLoader)
```

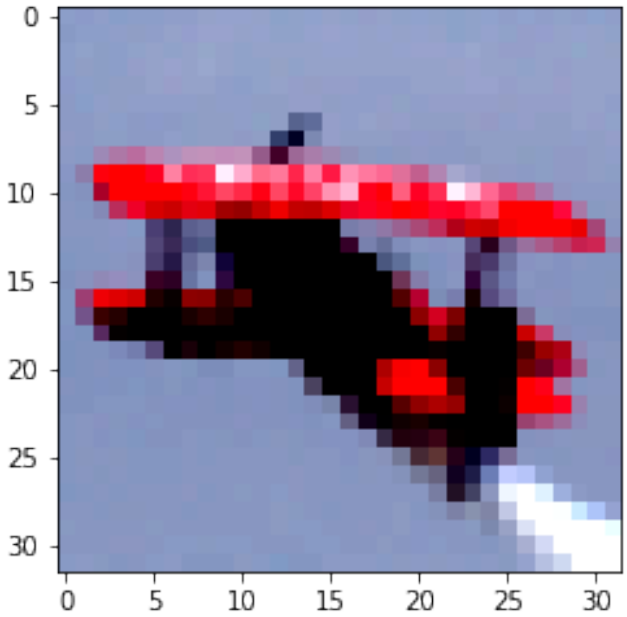
```
100%|                    | 313/313 [00:10<00:00, 30.40it/s]

mean: tensor([0.4468, 0.4399, 0.4067])
std:  tensor([0.2419, 0.2384, 0.2541])
```

[63]:
```
#re-load the train test and val data with newly calculated mean and std dev

data_transform2 = transforms.Compose([ transforms.Resize((32,32)), transforms.
 ↪ToTensor(),
                                      transforms.Normalize(mean=train_mean,␣
 ↪std=train_std)])


trainSet = stl10_dataset(text_file ='splits/train.txt', root_dir = '',␣
 ↪transform=data_transform2)
testSet = stl10_dataset(text_file ='splits/test.txt', root_dir = '',␣
 ↪transform=data_transform2)
valSet = stl10_dataset(text_file ='splits/val.txt', root_dir = '',␣
 ↪transform=data_transform2)

stl10_TrainLoader = torch.utils.data.DataLoader(trainSet, batch_size=128,␣
 ↪shuffle=True)
stl10_TestLoader = torch.utils.data.DataLoader(testSet, batch_size=1,␣
 ↪shuffle=True)
stl10_ValLoader = torch.utils.data.DataLoader(valSet, batch_size=1,␣
 ↪shuffle=True)
```

[64]:
```
#plotting sample image
dataiter = iter(stl10_TrainLoader)
images, labels = dataiter.next()
plt.imshow(np.transpose(images[0].numpy(), (1, 2, 0)))
print(labels)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
tensor([0, 3, 3, 6, 6, 9, 0, 9, 1, 4, 5, 1, 6, 2, 5, 5, 2, 0, 3, 8, 1, 8, 1, 4,
        2, 0, 5, 8, 0, 9, 1, 7, 3, 3, 8, 4, 0, 2, 7, 8, 7, 1, 2, 1, 7, 5, 4, 3,
        8, 8, 7, 0, 2, 4, 2, 7, 6, 6, 1, 5, 6, 2, 0, 3, 1, 5, 6, 0, 3, 8, 3, 6,
        1, 9, 5, 4, 6, 6, 4, 2, 9, 3, 7, 1, 1, 4, 3, 9, 3, 4, 7, 2, 3, 9, 0, 1,
        8, 2, 2, 9, 1, 1, 7, 0, 5, 4, 5, 3, 1, 8, 8, 1, 1, 9, 1, 9, 8, 6, 5, 7,
        1, 8, 3, 9, 3, 7, 8, 9])
```



[65]:
```
#lenet5 with batch normalization

class LeNet5_bn(nn.Module):
    def __init__(self):
        super(LeNet5_bn, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, stride = 1, kernel_size=(5,5))
        self.conv1_bn = nn.BatchNorm2d(6)
        self.pool1 = nn.MaxPool2d(2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, stride = 1, kernel_size=(5,5))
        self.conv2_bn = nn.BatchNorm2d(16)
        self.pool2 = nn.MaxPool2d(2, stride=2)
        self.fc1 = nn.Linear(400, 120)
        self.fc1_bn = nn.BatchNorm1d(120)
        self.fc2 = nn.Linear(120, 84)
        self.fc2_bn = nn.BatchNorm1d(84)
        self.fc3 = nn.Linear(84,10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.conv1_bn(x)
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.conv2_bn(x)
        x = self.pool2(x)
        x = torch.flatten(x,1)
```

```
        x = F.relu(self.fc1(x))
        x = self.fc1_bn(x)
        x = F.relu(self.fc2(x))
        x = self.fc2_bn(x)
        x = self.fc3(x)
        return x
```

[66]:
```
#lenet5
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, stride = 1, kernel_size=(5,5))
        self.pool1 = nn.MaxPool2d(2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, stride = 1, kernel_size=(5,5))
        self.pool2 = nn.MaxPool2d(2, stride=2)
        self.fc1 = nn.Linear(400, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84,10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = torch.flatten(x,1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

[67]:
```
# multi-class accuracy function

def multi_accuracy(y_pred, y_test):
    y_pred_softmax = torch.log_softmax(y_pred, dim = 1)
    _, y_pred_tags = torch.max(y_pred_softmax, dim = 1)
    correct_pred = (y_pred_tags == y_test).float()
    acc = correct_pred.sum() / len(correct_pred)
    acc = torch.round(acc * 100)
    return acc
```

[68]:
```
def train(model, epochs, criterion, optimizer, scheduler,model_name=""):
    model.train()
    min_valid_loss = float("inf")

    for epoch in range(1,epochs+1):  # loop over the dataset multiple times
        running_loss = 0.0
        running_accuracy = 0

        for i, data in enumerate(stl10_TrainLoader, 0):
```

```
            batch_loss = 0
            batch_accuracy = 0

            inputs, labels = data    # get the inputs; data is a list of␣
↪[inputs, labels]

            optimizer.zero_grad()    # zero the parameter gradients


            outputs = model(inputs)  # forward + backward + optimize
            loss = criterion(outputs, labels) # calculate loss
            loss.backward()                   # accumulate gradient
            optimizer.step()                  # update weights

            # add statistics calculated
            running_loss += loss.item()
            running_accuracy += multi_accuracy(outputs,labels)
            batch_loss += loss.item()
            batch_accuracy += multi_accuracy(outputs,labels)

            # log statistics on tensorboard
            writer.add_scalars(model_name+' batch_step_loss', {'training loss':
↪(batch_loss/len(data))}, epoch)
            writer.add_scalars(model_name+' batch_step_accuracy', {'training␣
↪accuracy':(batch_accuracy/len(data))}, epoch)


        # advance the scheduler for lr decay
        scheduler.step()


        # calculate loss and accuracy on validation set
        r_valid_loss = 0.0
        r_valid_accuracy = 0
        model.eval()
        for data, labels in stl10_ValLoader:
            target = model(data)
            loss = criterion(target,labels)
            r_valid_loss += loss.item()
            r_valid_accuracy += multi_accuracy(target, labels)


        #print Loss and Accuracy every 5 epochs
        if epoch%5 ==0:
            print("Epoch", epoch," Training Loss", running_loss/
↪len(stl10_TrainLoader)," Training Accuracy", running_accuracy/
↪len(stl10_TrainLoader))
```

```
            print("Epoch", epoch," Validation Loss", r_valid_loss/
↪len(stl10_ValLoader)," Validation Accuracy", r_valid_accuracy/
↪len(stl10_ValLoader))

        # log training vs validation loss and accuracy
        writer.add_scalars(model_name+' loss', {'training loss':(running_loss/
↪len(stl10_TrainLoader)),'validation loss':(r_valid_loss/
↪len(stl10_ValLoader))}, epoch)
        writer.add_scalars(model_name+' accuracy', {'training accuracy':
↪(running_accuracy/len(stl10_TrainLoader)),'validation accuracy':
↪(r_valid_accuracy/len(stl10_ValLoader))}, epoch)

    print('Finished Training')
    return model
```

[69]:
```
# print class wise accuracy

def print_class_wise_acc(cm, model_name):
    print("Accuracy by each class for {}".format(model_name))
    accuracy_multiclass = []
    index = 0
    for row in cm:
        accuracy_multiclass.append([index+1, (row[index]/sum(row))*100])
        index += 1
    df = pd.DataFrame(accuracy_multiclass, columns=['class', 'accuracy'])
    print(df)
```

[70]:
```
def evaluate_model(model,model_name=""):
    model.eval()

    y_pred_list = []
    y_test_list = []
    correct_pred = 0
    failed_dict = {}

    with torch.no_grad():
        for X_batch, y_batch in stl10_TestLoader:            # loop through the␣
↪test batch
            X_batch = X_batch.type(torch.FloatTensor)        # convert to tensor
            y_batch.type(torch.FloatTensor)
            y_test_pred = model(X_batch)                     # get prediction

            y_pred_softmax = torch.log_softmax(y_test_pred, dim = 1)
            _, y_pred_tags = torch.max(y_pred_softmax, dim = 1)    # get class␣
↪tag

            y_pred_list.append(y_pred_tags.item())
```

```
            y_test_list.append(y_batch.item())
            if y_batch != y_pred_tags:                       # save one example␣
↪of each wrongly predicted class
                if y_batch.item() not in failed_dict:
                    failed_dict[y_batch.item()] = (X_batch, y_pred_tags)

            correct_pred += multi_accuracy(y_test_pred, y_batch)
        print(correct_pred.item()/len(stl10_TestLoader))


        y_test_list, y_pred_list = np.array(y_test_list), np.array(y_pred_list)
        y_unique = np.array([1,2,3,4,5,6,7,8,9,10])          # unique label␣
↪names for confusion matrix plot
        cm = confusion_matrix(y_test_list, y_pred_list)      # sklearn's␣
↪confusion matrix

        plot_confusion_matrix(cm, y_unique,title=model_name) # plot confusion␣
↪matrix
        print_class_wise_acc(cm, model_name)                 # print classwise␣
↪accuracy

        # plot misclassified images

        i=0
        fig, axs = plt.subplots(2, 5, figsize=(25, 10))
        failed_dict = dict(sorted(failed_dict.items()))
        for label, (image, pred) in failed_dict.items():
            image = torch.squeeze(image, axis=0)

            if i<5:
                axs[0, i].imshow(np.transpose(image.numpy(), (1, 2, 0)))
                axs[0, i].set_title("label:"+str(label+1)+" pred:"+str(pred.
↪item()+1))
            else:
                axs[1, i-5].imshow(np.transpose(image.numpy(), (1, 2, 0)))
                axs[1, i-5].set_title("label:"+str(label+1)+" pred:"+str(pred.
↪item()+1))

            i+=1
```

[71]:
```
def plot_confusion_matrix(cm,
                          target_names,
                          title='Confusion matrix',
```

```
                          cmap=None,
                          normalize=False):

    accuracy = np.trace(cm) / float(np.sum(cm))
    misclass = 1 - accuracy

    if cmap is None:
        cmap = plt.get_cmap('Purples')

    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()

    if target_names is not None:
        tick_marks = np.arange(len(target_names))
        plt.xticks(tick_marks, target_names, rotation=45)
        plt.yticks(tick_marks, target_names)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]


    thresh = cm.max() / 1.5 if normalize else cm.max() / 2
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        if normalize:
            plt.text(j, i, "{:0.4f}".format(cm[i, j]),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")
        else:
            plt.text(j, i, "{:,}".format(cm[i, j]),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")


    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label\naccuracy={:0.4f}; misclass={:0.4f}'.
↪format(accuracy, misclass))
    plt.show()
```

1 lenet5 main experiment

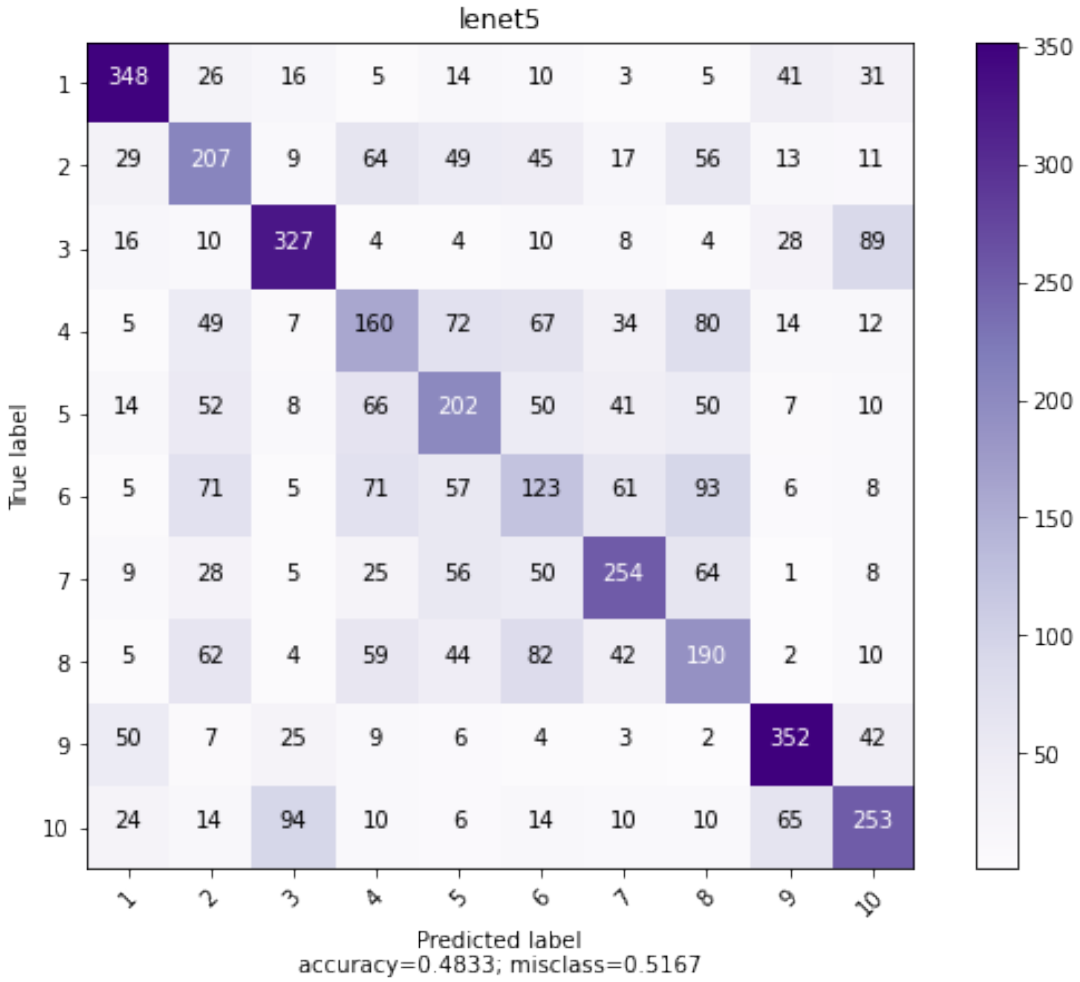
[72]:
```
lenet = LeNet5()
EPOCHS = 100
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(lenet.parameters(), lr=0.001)
scheduler = StepLR(optimizer, step_size=20, gamma=0.5)
lenet = train(lenet, EPOCHS, criterion, optimizer,␣
 ↪scheduler,model_name="lenet5")
writer.flush()
```

```
Epoch 5  Training Loss 1.575697299838066  Training Accuracy tensor(40.7750)
Epoch 5  Validation Loss 1.5698190864123387  Validation Accuracy tensor(40.1801)
Epoch 10  Training Loss 1.3818290472030639  Training Accuracy tensor(47.8250)
Epoch 10  Validation Loss 1.4684835151305267  Validation Accuracy
tensor(44.9817)
Epoch 15  Training Loss 1.2527966618537902  Training Accuracy tensor(53.5250)
Epoch 15  Validation Loss 1.4814216553644415  Validation Accuracy
tensor(45.6485)
Epoch 20  Training Loss 1.1106121674180032  Training Accuracy tensor(58.4500)
Epoch 20  Validation Loss 1.4325014029500225  Validation Accuracy
tensor(47.8826)
Epoch 25  Training Loss 0.9411104649305344  Training Accuracy tensor(66.3250)
Epoch 25  Validation Loss 1.4308740707124026  Validation Accuracy
tensor(49.6499)
Epoch 30  Training Loss 0.8682149231433869  Training Accuracy tensor(69.1250)
Epoch 30  Validation Loss 1.476574780098051  Validation Accuracy tensor(49.3164)
Epoch 35  Training Loss 0.7914263606071472  Training Accuracy tensor(71.4500)
Epoch 35  Validation Loss 1.5232982737680563  Validation Accuracy
tensor(49.0830)
Epoch 40  Training Loss 0.6944576188921928  Training Accuracy tensor(75.8250)
Epoch 40  Validation Loss 1.59869904123073  Validation Accuracy tensor(49.6499)
Epoch 45  Training Loss 0.61368098706007  Training Accuracy tensor(79.4000)
Epoch 45  Validation Loss 1.6481357428658285  Validation Accuracy
tensor(48.8830)
Epoch 50  Training Loss 0.5598262213170528  Training Accuracy tensor(81.6500)
Epoch 50  Validation Loss 1.700431093317772  Validation Accuracy tensor(49.6165)
Epoch 55  Training Loss 0.5210699081420899  Training Accuracy tensor(82.7750)
Epoch 55  Validation Loss 1.761116014614465  Validation Accuracy tensor(48.9830)
Epoch 60  Training Loss 0.45923717096447947  Training Accuracy tensor(85.0750)
Epoch 60  Validation Loss 1.8553456196684404  Validation Accuracy
tensor(48.9830)
Epoch 65  Training Loss 0.41152731962502004  Training Accuracy tensor(87.9000)
Epoch 65  Validation Loss 1.904910587746458  Validation Accuracy tensor(49.1164)
Epoch 70  Training Loss 0.40353403985500336  Training Accuracy tensor(87.6250)
Epoch 70  Validation Loss 1.9576027167229801  Validation Accuracy
tensor(48.6496)
Epoch 75  Training Loss 0.3709789726883173  Training Accuracy tensor(89.2500)
```

```
Epoch 75  Validation Loss 2.001226163027382  Validation Accuracy tensor(48.3494)
Epoch 80  Training Loss 0.34382734894752504  Training Accuracy tensor(90.0250)
Epoch 80  Validation Loss 2.0738771795275226  Validation Accuracy
tensor(48.4495)
Epoch 85  Training Loss 0.3319267462939024  Training Accuracy tensor(90.5750)
Epoch 85  Validation Loss 2.091185016211298  Validation Accuracy tensor(48.5495)
Epoch 90  Training Loss 0.3158308815211058  Training Accuracy tensor(91.2500)
Epoch 90  Validation Loss 2.1341774685329136  Validation Accuracy
tensor(48.0827)
Epoch 95  Training Loss 0.3058752354234457  Training Accuracy tensor(91.6750)
Epoch 95  Validation Loss 2.1691040179315215  Validation Accuracy
tensor(47.9493)
Epoch 100  Training Loss 0.2926258221268654  Training Accuracy tensor(92.4250)
Epoch 100  Validation Loss 2.1975043401674457  Validation Accuracy
tensor(48.2828)
Finished Training
```

[73]:
```
evaluate_model(lenet, model_name="lenet5")
```

```
48.32966593318664
```

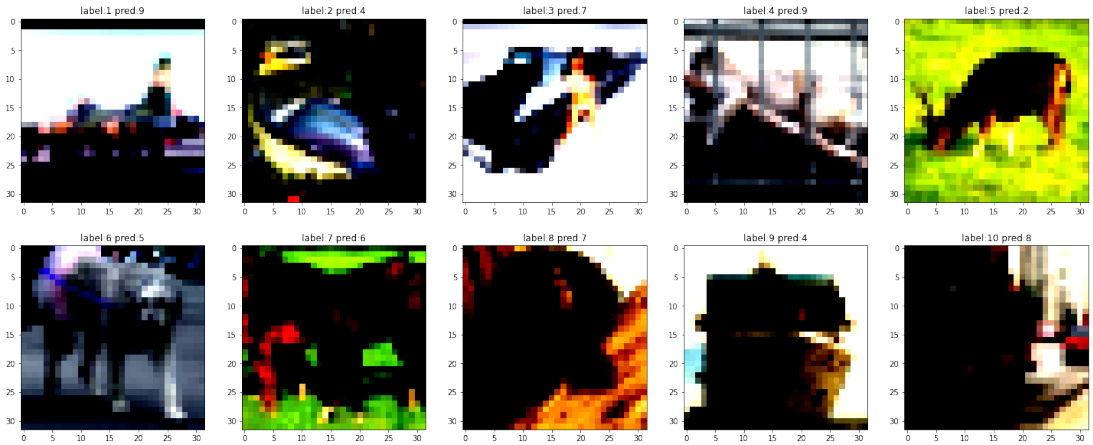



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
Accuracy by each class for lenet5
   class   accuracy
0      1  69.739479
1      2  41.400000
2      3  65.400000
3      4  32.000000
4      5  40.400000
```

```
5       6  24.600000
6       7  50.800000
7       8  38.000000
8       9  70.400000
9      10  50.600000
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



2 lenet5 with l2 regularization

[74]:
```
lenet_l2 = LeNet5()
EPOCHS = 100
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(lenet_l2.parameters(), lr=0.001, weight_decay=0.01)
scheduler = StepLR(optimizer, step_size=20, gamma=0.5)
lenet_l2 = train(lenet_l2, EPOCHS, criterion, optimizer, scheduler, model_name␣
 ↪="lenet5_l2_regularization")
writer.flush()
```

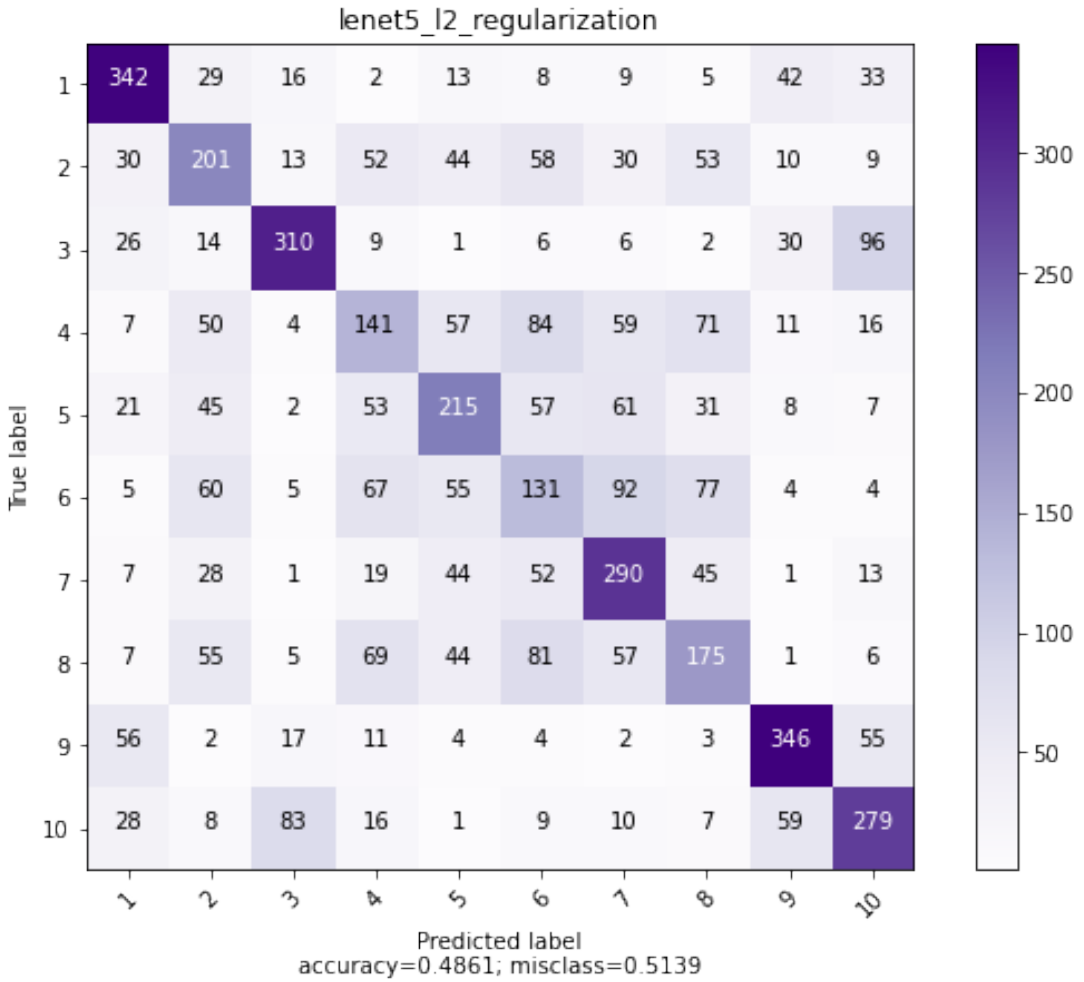
```
Epoch 5  Training Loss 1.6420347273349762  Training Accuracy tensor(37.5750)
Epoch 5  Validation Loss 1.708211713826713  Validation Accuracy tensor(36.2454)
```

```
Epoch 10  Training Loss 1.4685825616121293  Training Accuracy tensor(44.8250)
Epoch 10  Validation Loss 1.587516279833372  Validation Accuracy tensor(40.7803)
Epoch 15  Training Loss 1.3916505768895149  Training Accuracy tensor(48.3000)
Epoch 15  Validation Loss 1.5191796361411283  Validation Accuracy
tensor(44.2481)
Epoch 20  Training Loss 1.2978912353515626  Training Accuracy tensor(51.2250)
Epoch 20  Validation Loss 1.4379625811569037  Validation Accuracy
tensor(46.9823)
Epoch 25  Training Loss 1.1904519021511077  Training Accuracy tensor(56.1000)
Epoch 25  Validation Loss 1.4550223189659326  Validation Accuracy
tensor(45.8486)
Epoch 30  Training Loss 1.1166773736476898  Training Accuracy tensor(58.2000)
Epoch 30  Validation Loss 1.4283744809205912  Validation Accuracy
tensor(47.4825)
Epoch 35  Training Loss 1.0651883035898209  Training Accuracy tensor(61.0500)
Epoch 35  Validation Loss 1.4388275674831252  Validation Accuracy
tensor(47.8159)
Epoch 40  Training Loss 1.0106955140829086  Training Accuracy tensor(63.4750)
Epoch 40  Validation Loss 1.433052927490014  Validation Accuracy tensor(48.9830)
Epoch 45  Training Loss 0.9035082712769509  Training Accuracy tensor(67.0750)
Epoch 45  Validation Loss 1.4437636285423912  Validation Accuracy
tensor(49.5165)
Epoch 50  Training Loss 0.8509379647672176  Training Accuracy tensor(69.5750)
Epoch 50  Validation Loss 1.4506518061003364  Validation Accuracy
tensor(50.2167)
Epoch 55  Training Loss 0.8053811475634575  Training Accuracy tensor(72.4250)
Epoch 55  Validation Loss 1.5086992580843983  Validation Accuracy
tensor(49.0830)
Epoch 60  Training Loss 0.7704974010586738  Training Accuracy tensor(73.4750)
Epoch 60  Validation Loss 1.5075913826771128  Validation Accuracy
tensor(49.5165)
Epoch 65  Training Loss 0.6968361765146256  Training Accuracy tensor(76.1000)
Epoch 65  Validation Loss 1.5386339542981458  Validation Accuracy
tensor(49.2497)
Epoch 70  Training Loss 0.6864814922213555  Training Accuracy tensor(76.8500)
Epoch 70  Validation Loss 1.5545678885834158  Validation Accuracy
tensor(49.1497)
Epoch 75  Training Loss 0.649484645575285  Training Accuracy tensor(78.1750)
Epoch 75  Validation Loss 1.569956872859134  Validation Accuracy tensor(49.2831)
Epoch 80  Training Loss 0.6333191588521003  Training Accuracy tensor(78.8000)
Epoch 80  Validation Loss 1.5863696949912869  Validation Accuracy
tensor(49.0497)
Epoch 85  Training Loss 0.6371806062757969  Training Accuracy tensor(79.9500)
Epoch 85  Validation Loss 1.600475216804388  Validation Accuracy tensor(49.7833)
Epoch 90  Training Loss 0.5840219862759113  Training Accuracy tensor(81.2750)
Epoch 90  Validation Loss 1.6080025716637671  Validation Accuracy
tensor(49.3831)
Epoch 95  Training Loss 0.5757010236382485  Training Accuracy tensor(81.1250)
```

```
Epoch 95  Validation Loss 1.6294508721079994  Validation Accuracy
tensor(48.9163)
Epoch 100  Training Loss 0.5617842935025692  Training Accuracy tensor(82.5250)
Epoch 100  Validation Loss 1.6480250835286523  Validation Accuracy
tensor(48.9496)
Finished Training
```

[75]:
```
evaluate_model(lenet_l2, model_name ="lenet5_l2_regularization")
```

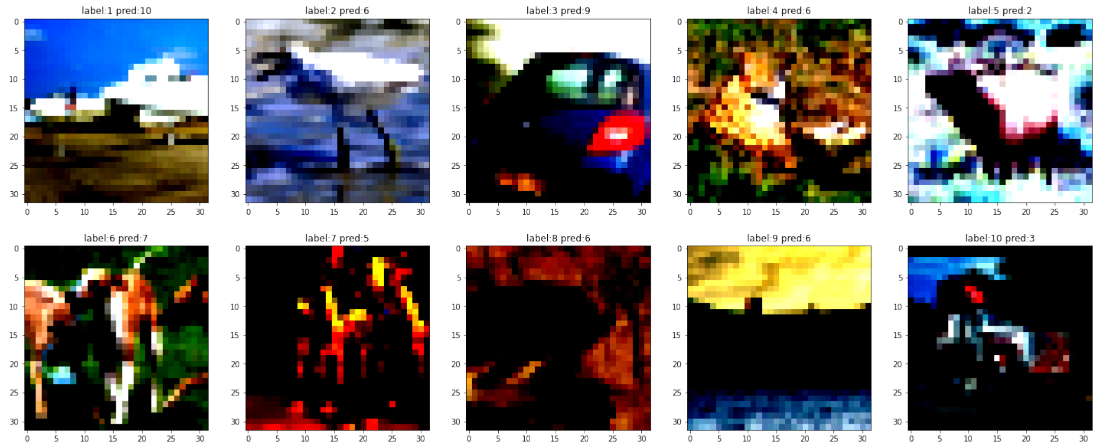
```
48.60972194438888
```




```
Accuracy by each class for lenet5_l2_regularization
   class   accuracy
0      1  68.537074
1      2  40.200000
2      3  62.000000
3      4  28.200000
```

```
4      5  43.000000
5      6  26.200000
6      7  58.000000
7      8  35.000000
8      9  69.200000
9     10  55.800000
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



3 lenet5 with batch normalization

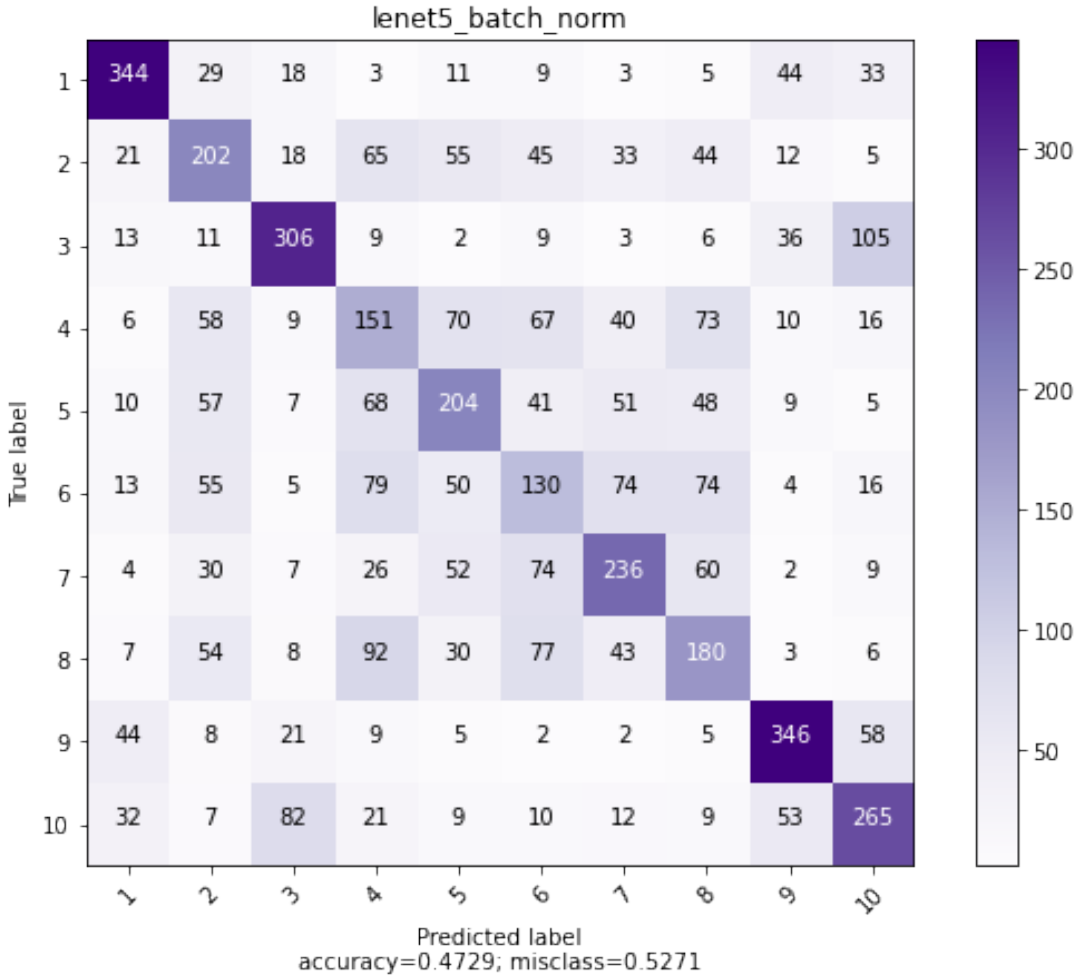
[76]:
```
lenet_bn = LeNet5_bn()
EPOCHS = 100
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(lenet_bn.parameters(), lr=0.001)
scheduler = StepLR(optimizer, step_size=20, gamma=0.5)
lenet = train(lenet_bn, EPOCHS, criterion, optimizer, scheduler,␣
 ↪model_name="lenet5_batch_norm")
writer.flush()
```

```
Epoch 5  Training Loss 1.0169735640287398  Training Accuracy tensor(64.2750)
Epoch 5  Validation Loss 1.5002265332678877  Validation Accuracy tensor(48.2828)
Epoch 10  Training Loss 0.4304424747824669  Training Accuracy tensor(86.5750)
Epoch 10  Validation Loss 1.9248181056759333  Validation Accuracy
tensor(47.4825)
Epoch 15  Training Loss 0.06895617498084902  Training Accuracy tensor(98.9500)
Epoch 15  Validation Loss 2.852661964290283  Validation Accuracy tensor(46.7156)
Epoch 20  Training Loss 0.007315184199251234  Training Accuracy tensor(100.)
Epoch 20  Validation Loss 3.4910287384480845  Validation Accuracy
tensor(47.5492)
Epoch 25  Training Loss 0.0029795408278005197  Training Accuracy tensor(100.)
Epoch 25  Validation Loss 3.7111832981714694  Validation Accuracy
tensor(48.0494)
Epoch 30  Training Loss 0.0019057502591749652  Training Accuracy tensor(100.)
Epoch 30  Validation Loss 3.886864897923908  Validation Accuracy tensor(47.9493)
Epoch 35  Training Loss 0.0013907953776651993  Training Accuracy tensor(100.)
Epoch 35  Validation Loss 4.04381811339142  Validation Accuracy tensor(48.1494)
Epoch 40  Training Loss 0.0010937311191810295  Training Accuracy tensor(100.)
Epoch 40  Validation Loss 4.176086124637156  Validation Accuracy tensor(48.0494)
Epoch 45  Training Loss 0.0009348673309432342  Training Accuracy tensor(100.)
Epoch 45  Validation Loss 4.242625395243457  Validation Accuracy tensor(48.0827)
Epoch 50  Training Loss 0.0007906935847131535  Training Accuracy tensor(100.)
Epoch 50  Validation Loss 4.3074973868896125  Validation Accuracy
tensor(48.1827)
Epoch 55  Training Loss 0.0007077229107380845  Training Accuracy tensor(100.)
Epoch 55  Validation Loss 4.3729505263246065  Validation Accuracy
tensor(48.0160)
Epoch 60  Training Loss 0.000609176728175953  Training Accuracy tensor(100.)
Epoch 60  Validation Loss 4.439311746580169  Validation Accuracy tensor(48.0494)
Epoch 65  Training Loss 0.0005621202843030914  Training Accuracy tensor(100.)
Epoch 65  Validation Loss 4.473744677766031  Validation Accuracy tensor(48.1160)
Epoch 70  Training Loss 0.0005396161330281756  Training Accuracy tensor(100.)
Epoch 70  Validation Loss 4.509700376752205  Validation Accuracy tensor(48.0827)
Epoch 75  Training Loss 0.0004883895075181499  Training Accuracy tensor(100.)
Epoch 75  Validation Loss 4.547178981526407  Validation Accuracy tensor(47.9827)
Epoch 80  Training Loss 0.0004573613565298729  Training Accuracy tensor(100.)
Epoch 80  Validation Loss 4.585343365932771  Validation Accuracy tensor(48.0160)
```

```
Epoch 85  Training Loss 0.00042736648210848214  Training Accuracy tensor(100.)
Epoch 85  Validation Loss 4.607082829464927  Validation Accuracy tensor(48.0160)
Epoch 90  Training Loss 0.0004128491629671771  Training Accuracy tensor(100.)
Epoch 90  Validation Loss 4.627614226111002  Validation Accuracy tensor(48.0160)
Epoch 95  Training Loss 0.00039616950962226836  Training Accuracy tensor(100.)
Epoch 95  Validation Loss 4.650277558316599  Validation Accuracy tensor(48.0494)
Epoch 100  Training Loss 0.0003721174212842016  Training Accuracy tensor(100.)
Epoch 100  Validation Loss 4.677111216709572  Validation Accuracy
tensor(48.0827)
Finished Training
```

[77]:
```
evaluate_model(lenet_bn, model_name="lenet5_batch_norm")
```

```
47.289457891578316
```

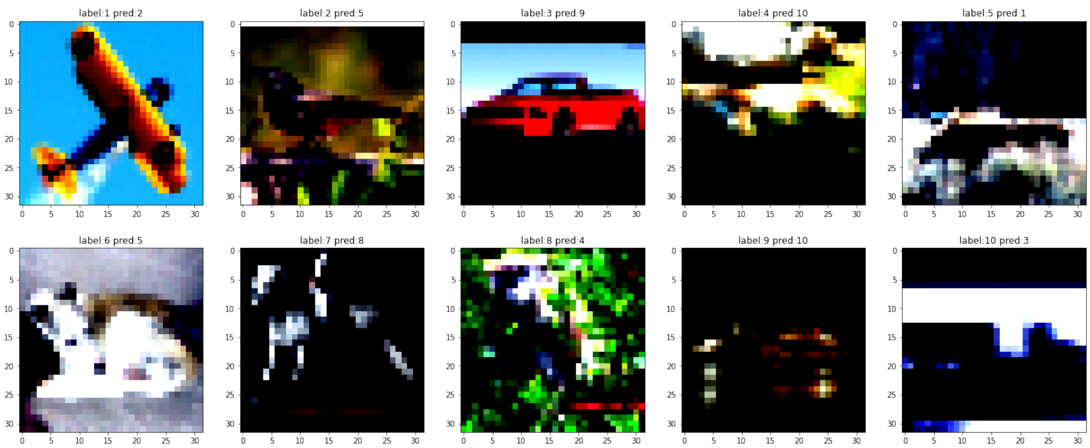



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

Accuracy by each class for lenet5_batch_norm

	class	accuracy
0	1	68.937876
1	2	40.400000
2	3	61.200000
3	4	30.200000
4	5	40.800000
5	6	26.000000
6	7	47.200000
7	8	36.000000
8	9	69.200000
9	10	53.000000

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



4 Accuracies for each model

Lenet5 Model: 48.32966593318664

Lenet5 l2 Regularization Model: 48.60972194438888

Lenet5 Batch Normalization Model: 47.289457891578316

[ ]: