Mansoor Nabawi, 309498.

DDA Ex07. 25.06.2022

- Functions, Downloading Dataset, Extracting, Making dimensions right

Initially, we check if GPU is available which lets us work faster.

```
[5] #to extract the download cifar10 file, reference: cs.toronto.edu
    def unpickle(file):
        import pickle
        with open(file, 'rb') as fo:
            dict = pickle.load(fo, encoding='latin1')
        return dict
```

```
#making an array of 10 columns and it has 1 whenever the class is there
#one hot encoding for labels/targets
def get_label(y):
  label = np.zeros((len(y), 10))
  for i, val in enumerate(y):
    label[i][val]= 1
  return label
```

- Manual data downloading, fixing dimensions

```
[7] !wget https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
    !tar -xvf cifar-10-python.tar.gz
```

```
--2022-06-25 21:59:22--  https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: 'cifar-10-python.tar.gz'

cifar-10-python.tar 100%[===================>] 162.60M  95.4MB/s    in 1.7s

2022-06-25 21:59:24 (95.4 MB/s) - 'cifar-10-python.tar.gz' saved [170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
```

- Reading batches, extracting and transforming data, and test set

There are 5 batches for the data set and one batch for the test set. We read them and concatenate them together and in the end, we fix the shape and transform tensor

```
#reading file names in the directory of extracted file.
filenames = []
for f in os.listdir("cifar-10-batches-py"):
  if f.startswith("data"):
    filenames.append(f)

#extracting from pickle and making new variables for data and label
filenames.sort()
for n, filen in enumerate(filenames):
  batch = unpickle("cifar-10-batches-py/"+filen)
  globals()['data%s' % (n+1)] = batch["data"]
  globals()['label%s' % (n+1)] = batch["labels"]

#transforming the shape of the data
train = [data1]

for i in range(2,6):
  train.append(globals()['data%s' % i])
#all bathces in one array
train = np.concatenate(train, axis=0)
#shape -> images,channels,height,width
train = train.reshape((len(train), 3, 32, 32))

#converting to Tensors
train = torch.Tensor(train)

#transforming the shape of label
label = [label1]
for i in range(2,6):
  label.append(globals()['label%s' % (i)])

label = np.concatenate(label, axis=0)
label = get_label(label)
#converting to Tensors
label_train = torch.Tensor(label)
```

For the test set.

```
#loading and transforming the shape of the test data and its labels
test = unpickle("cifar-10-batches-py/test_batch")

test_data = test["data"]
test_label = test["labels"]
#shape -> images,channels,height,width
test_data = test['data'].reshape((len(test['data']), 3, 32, 32))
test_data = torch.Tensor(test_data)

#labels
test_label = get_label(test_label)
test_label = torch.Tensor(test_label)
```

```
test_data.shape
```

```
torch.Size([10000, 3, 32, 32])
```

- Now we need to make a Tensor dataset out of our data and put them in the data loader with chosen batches.

creating dataset and dataloader

```
from torch.utils.data import Dataset, DataLoader

params = {'batch_size': 64,
          'shuffle': True}

params_test = {'batch_size': 64,
          'shuffle': False}
#training_set = Dataset(train,label_train)

#craeting a dataset
dataset = torch.utils.data.TensorDataset(train,label_train)
dataset_test = torch.utils.data.TensorDataset(test_data,test_label)

#creating our dataloader
training_generator = torch.utils.data.DataLoader(dataset, **params)
test_generator = torch.utils.data.DataLoader(dataset_test, **params_test)
```

We also make some functions for data augmentation as it is needed later.

augmentation and normalization

```
#to augment and normalize data/images
def augment(img, test=False):
    names = []

    #transformations
    if test:
      augs = torchvision.transforms.Compose([torchvision.transforms.Normalize((0.4915, 0.4823, 0.4468),(0.2470, 0.2435, 0.2616))])
    else:
      augs = torchvision.transforms.Compose([torchvision.transforms.RandomHorizontalFlip(),torchvision.transforms.Normalize((0.4915, 0.4823, 0.4468),

    #augmentation
    for n in range(1*2):
      globals()['x%s' % (n+1)] = (augs(img))
      names.append(globals()['x%s' % (n+1)])

    return torch.stack(names, dim=0).flatten().reshape(len(names),3,32,32)

#to repeat labels for data
def label_repeater(label, length):
    names = []
    for n in range(length):
      globals()['x%s' % (n+1)] = label
      names.append(globals()['x%s' % (n+1)])

    return torch.stack(names, dim=0).flatten().reshape(length,10)
```

```
#final normalizer and augmentor
def norm_aug_dataset(data, label, test=False):
  #data
  dd = [augment(d, test) for d in data]
  dd = torch.stack(dd, dim=0).flatten().reshape(-1,3,32,32)

  #label
  ll = [label_repeater(l, 2) for l in label]
  ll = torch.stack(ll, dim=0).flatten().reshape(-1,10)


  return dd, ll
```

We defined 3 functions for augmentation and normalization.
augment() function receives the data and performs random horizontal flip and normalization and then stacks them, label_repeater() function makes the many labels out of one label we need, and finally norm_aug_dataset() receives data and label use the two previous mentioned functions to perform augmentation. Here our data gets doubled.
We also need to make a dataset and put the new dataset to the data loader, so we define them.

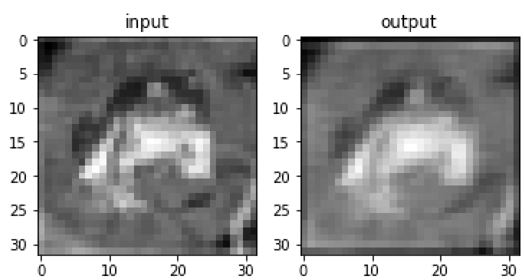
```
[18] augd, augl = norm_aug_dataset(train, label_train, test=False)
```

```
[19] augl.shape

torch.Size([100000, 10])
```

```
[ ]  params = {'batch_size': 64,
              'shuffle': True}

     #augmented
     dataset_aug = torch.utils.data.TensorDataset(augd, augl)

     #augmented datloader for training dataset
     training_generator_aug = torch.utils.data.DataLoader(dataset_aug, **params)
```

```
#We may not need this
#augmentation for test set
params_test = {'batch_size': 64,
          'shuffle': False}

norm_test, norm_label = norm_aug_dataset(test_data,test_label, test=False)
dataset_test_norm = torch.utils.data.TensorDataset(norm_test, norm_label)

#creating our dataloader
test_generator_norm = torch.utils.data.DataLoader(dataset_test_norm, **params_test)
```

We can look at the actions we perform on one of our images.
First conv result with 16 features out, 3 kernels, and 1 padding.



Defining the Network

```
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self, nclasses, img, nchans3=32, nhidden=256):
        super().__init__()
        nchannels, nrows, ncols = img.shape
        self.nchans1 = nchans3 //4
        self.nchans2 = nchans3 // 2
        self.nchans3 = nchans3
        self.nhidden = nhidden
        self.nhidden1 = nhidden//2
        self.nclasses = nclasses
        self.conv1 = nn.Conv2d(nchannels, self.nchans1, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(self.nchans1, self.nchans2, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(self.nchans2, self.nchans3, kernel_size=3, padding=1)
        # size of input to fc1 will be  32 * nrows/4 * ncols/4,
        # We divide by 4 since we apply 2 maxpooling layers with size 2
        # For a 32x32 image, this becomes 8x8 times 32 channnels.
        self.nflat = nrows // 4 * ncols // 4
        self.fc1 = nn.Linear(self.nchans3 * self.nflat, self.nhidden)
        self.fc2 = nn.Linear(self.nhidden , self.nhidden1)
        self.fc3 = nn.Linear(self.nhidden1, self.nclasses)

    def forward(self, x):
        out = F.max_pool2d(F.relu(self.conv1(x)),2)
        out = (F.relu(self.conv2(out)))
        out = F.max_pool2d(F.relu(self.conv3(out)),2)
        #flatten, we could also use torch.flatten()
        #out = out.view(-1, self.nchans3 * self.nflat)
        out = torch.flatten(out, 1) # flatten all dimensions except batch
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = F.relu(self.fc3(out))
        # out = F.log_softmax(out, dim=1)
        out = F.softmax(out, dim=1)

        return out
```

We define our network as it is asked in the exercise.
In the beginning, our class gets the number of classes and an image to find the channels, height, and width.
3 Convs.
3,8,(3,3),(1,1) - > 8,16,3,3,1,1 ->16,32,3,3,1,1
In each step it is divided by 2.
In the forward method, everything is done as it is asked.
Final softmax will decide over 10 classes

We define our training loop.

```python
import datetime


def training_loop(n_epochs, optimizer, model, loss_fn, train_loader, test_loader, l2_regularizer=0, l1_regularizer=0, print_every=1, tag_txt="first"):
    train_size = len(train_loader.dataset)
    for epoch in tqdm(range(1, n_epochs + 1), leave=True):
        loss_train = 0.0
        running_loss = 0.0

        correct = 0
        total = 0
        acc_mini = 0
        acc= 0
        for i, (data) in tqdm(enumerate(train_loader),leave=False):
            imgs, labels = data
            # print(labels)
            imgs = imgs.to(device=device)
            labels = labels.to(device=device)
            outputs = model(imgs)
            # print(outputs)
            #loss--------
            loss = loss_fn(outputs, labels)

            #regularization
            if l2_regularizer != 0:

              l2_norm = sum(p.pow(2.0).sum() for p in model.parameters())
              loss = loss + l2_regularizer * l2_norm

            if l1_regularizer != 0:
              l1_norm = torch.tensor(0., requires_grad=True)
              for name, param in model.named_parameters():
                  if 'weight' in name:
                      L1_reg = l1_norm + torch.sum(torch.abs(param))

              loss = loss + l1_regularizer*L1_reg

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            loss_train += loss.item()
            running_loss += loss.item()
            #----------

            #accuracy----------
            _, predicted = torch.max(outputs, dim=1)
            #i added for written dataset
            _, labels = torch.max(labels, dim=1)
            total += labels.shape[0]  # batch size
            correct += int((predicted == labels).sum())
            #----------------
```

```python
            #----------------

            # n_iter = epoch*len(train_dataloader)+batch

            if i % 100 == 99:    # print every 100 mini-batches
              acc_mini = correct / total
              writer.add_scalars(f'Loss_minibatch_{tag_txt}', {"Train_loss":running_loss / 100}, ((epoch)+(i + 1)))
              writer.add_scalars(f'Accuracy_minibatch_{tag_txt}', {"Train_acc":acc_mini}, ((epoch)+(i + 1)))

              running_loss = 0.0
              acc_mini = 0


        accuracy = correct / total

        if epoch == 1 or epoch % print_every == 0:

            test_size = len(test_loader.dataset)
            test_loss, test_acc = 0, 0

            with torch.no_grad():
                for X, y in tqdm(test_loader,leave = False):
                    X, y = X.to(device), y.to(device)
                    pred = model(X)
                    _, predicted = torch.max(pred, dim=1)
                    _, labels = torch.max(y, dim=1)


                    total += labels.shape[0]  # batch size

                    correct += int((predicted == labels).sum())

                    loss = loss_fn(pred, y)
                    test_loss += loss.item()


            test_acc = correct / total
            test_loss /=test_size

            writer.add_scalars(f'Train_loss', {f"{tag_txt}":loss_train / len(train_loader)}, ((epoch)))
            writer.add_scalars(f'Test_loss', {f"{tag_txt}":test_loss}, ((epoch)))

            writer.add_scalars(f'Train_Accuracy', {f"{tag_txt}":accuracy}, ((epoch)))
            writer.add_scalars(f'Test_Accuracy', {f"{tag_txt}":test_acc}, ((epoch)))
```

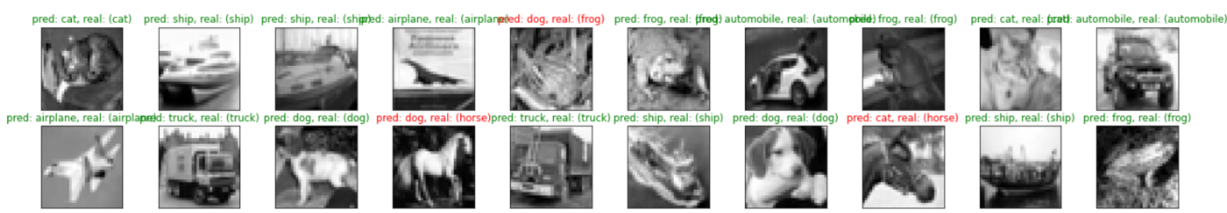
In our training loop we go through the number of epochs, here we choose 20 epochs. In each epoch we go through each batch of our train dataset and load them, we check if GPU is available. We input the images into the model and train it and get our predictions. Then we get the loss of the predicted and real labels. We check if there is L2 or L1 regularization values so we punish our model. To get the accuracy we get the max value and in each batch size we check the correct ones and some the number of them.
In each 100 minibatch we also check the accuracy and loss and write them into our tensorboard.
In the end we also check for test accuracy and loss as well as train accuracy and loss.

This is how start training our first CNN. lr=1e-2, 20 epochs. 0 regularizations

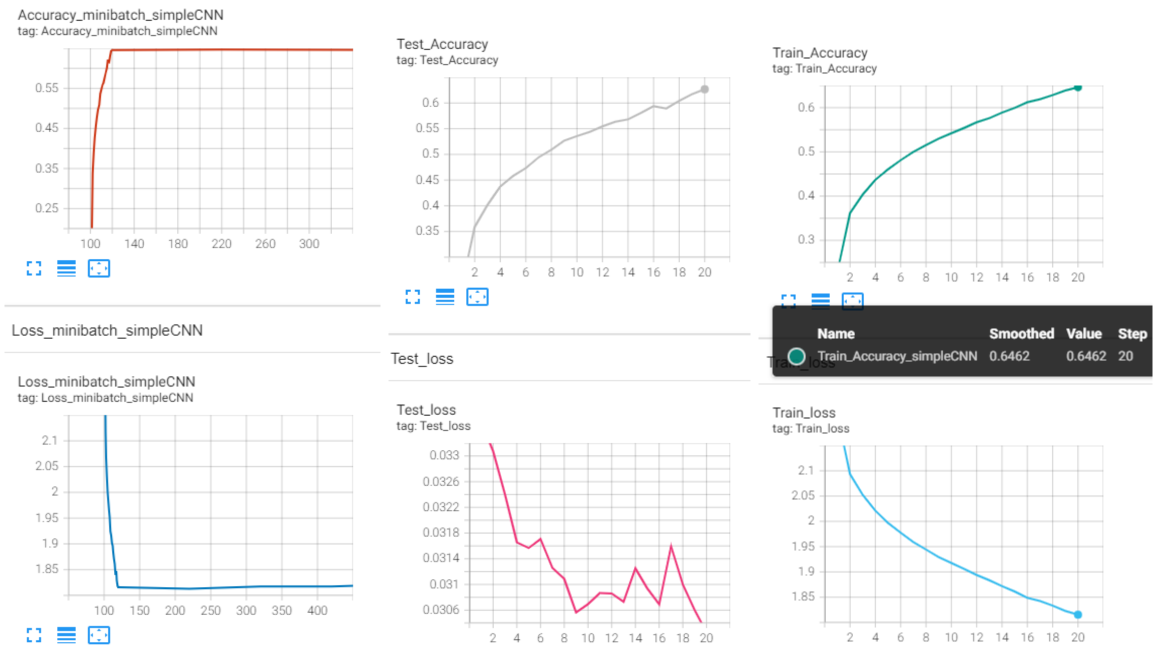
```
#Training

nclasses = 10
set_seed(0)
img_batch = img_t.to(device=device)

model1 = Net(nclasses, img_batch[0]).to(device=device)
optimizer = optim.SGD(model1.parameters(), lr=1e-2)
loss_fn = nn.CrossEntropyLoss()

training_loop(
    n_epochs=20,
    optimizer=optimizer,
    model=model1,
    loss_fn=loss_fn,
    train_loader=training_generator,
    test_loader = test_generator,
    tag_txt = "simpleCNN"
)
```

\

We can look at some predictions and their true label of this model.



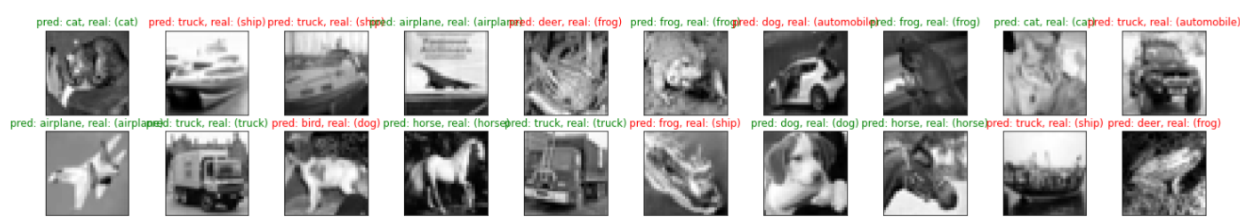
We see in the above images, out of 20 only 3 labels predicted wrongly.
We can also check all the losses and accuracies on tensorboard.



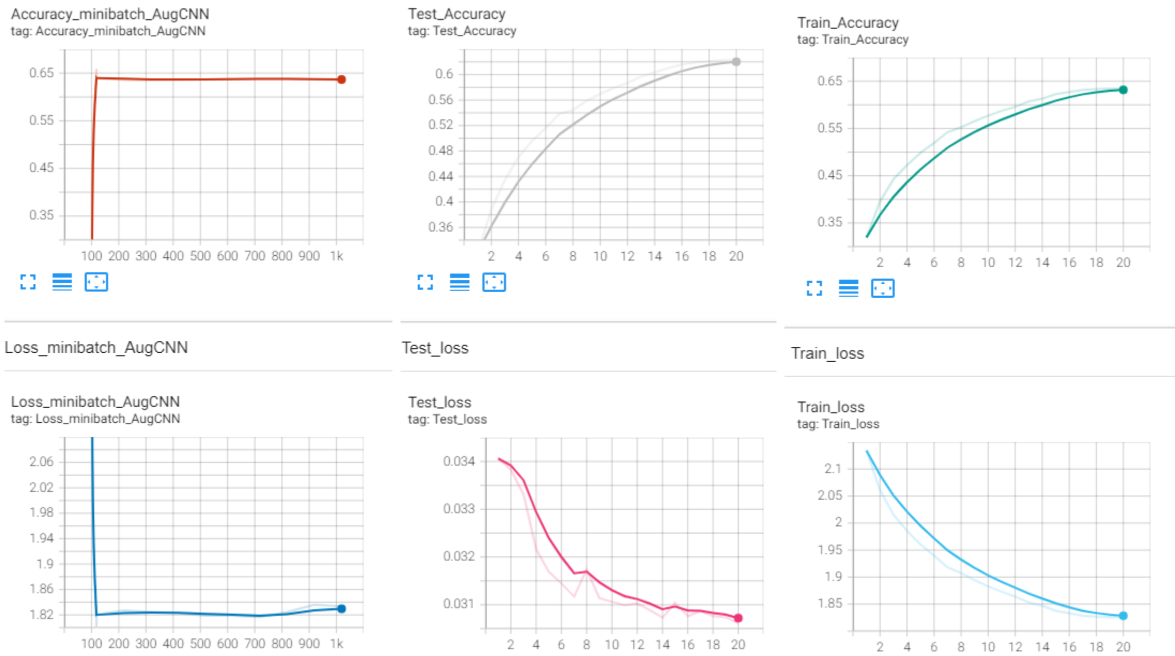
We can see that the accuracy in every 100 minibatch started from 0.22 and increased to 0.646. The loss in minibatch is also decreasing from 2.2 to 1.85.
Train accuracy and test accuracy follows the same trend train accuracy in the final epoch is 0.6462 whereas in test it is 0.62. The loss of train is always decreasing but in the test set there is fluctuations.

Using Augmented and normalized dataset to train the dataset.

The values are the same as before, the only change is our training set which is the augmented one, the test set is the original one.
Lets see the predictions



As you see we have more incorrect labels, this might be due to using augmented training set and original test set. The augmented dataset has double of the size of the original train set. But as some images are flipped horizontally randomly we may not have all the pictures, the real and augmented one (at least i guess because i did all the preprocessing step manually i am not sure of how the augmentation kept the real data.)



The accuracy in minbatches have a good value, it increased from 0.33 until 0.65 and it almost stayed the same for the next 1000 batches. The loss has a good decrease too.
Train accuracy increased from 0.35 to 0.6344 while using original dataset it was 0.6462
Test accuracy in the original dataset was 0.6267 while in augmented dataset it is 0.6232.
The loss in both train and test has a smooth decrease.

BONUS part -> regularizations and dropout.
The regularizations are already included in our training loop, as l2 and l1 regularizers punishes the weight by the value we input. Here is the snap of the regulizers.

```
#regularization
if l2_regularizer != 0:

  l2_norm = sum(p.pow(2.0).sum() for p in model.parameters())
  loss = loss + l2_regularizer * l2_norm

if l1_regularizer != 0:
  l1_norm = torch.tensor(0., requires_grad=True)
  for name, param in model.named_parameters():
      if 'weight' in name:
          L1_reg = l1_norm + torch.sum(torch.abs(param))

  loss = loss + l1_regularizer*L1_reg
```

We also put drop out of 0.2 in our network.
This is added to our initialization.

```
self.dropout = nn.Dropout(0.20)
```

And this changes made to our forward function.

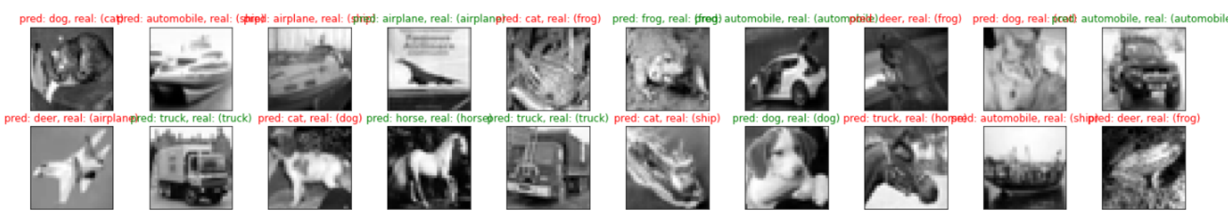
```
def forward(self, x):

  out = F.max_pool2d(F.relu(self.conv1(x)),2)
  out = (F.relu(self.conv2(out)))
  out = F.max_pool2d(F.relu(self.conv3(out)),2)
  #flatten, we could also use torch.flatten()
```

```
        out = torch.flatten(out, 1) # flatten all dimensions except batch
        #out = out.view(-1, self.nchans3 * self.nflat)
        out = self.dropout(out)
        out = F.relu(self.fc1(out))
        out = self.dropout(out)
        out = F.relu(self.fc2(out))
        out = self.dropout(out)
        out = F.relu(self.fc3(out))
        # out = F.log_softmax(out, dim=1)
        out = F.softmax(out, dim=1)

        return out
```

We can run our model with drop out and regulizers. The values of l2_regulizer is 1e-8 and lr is the same as before.

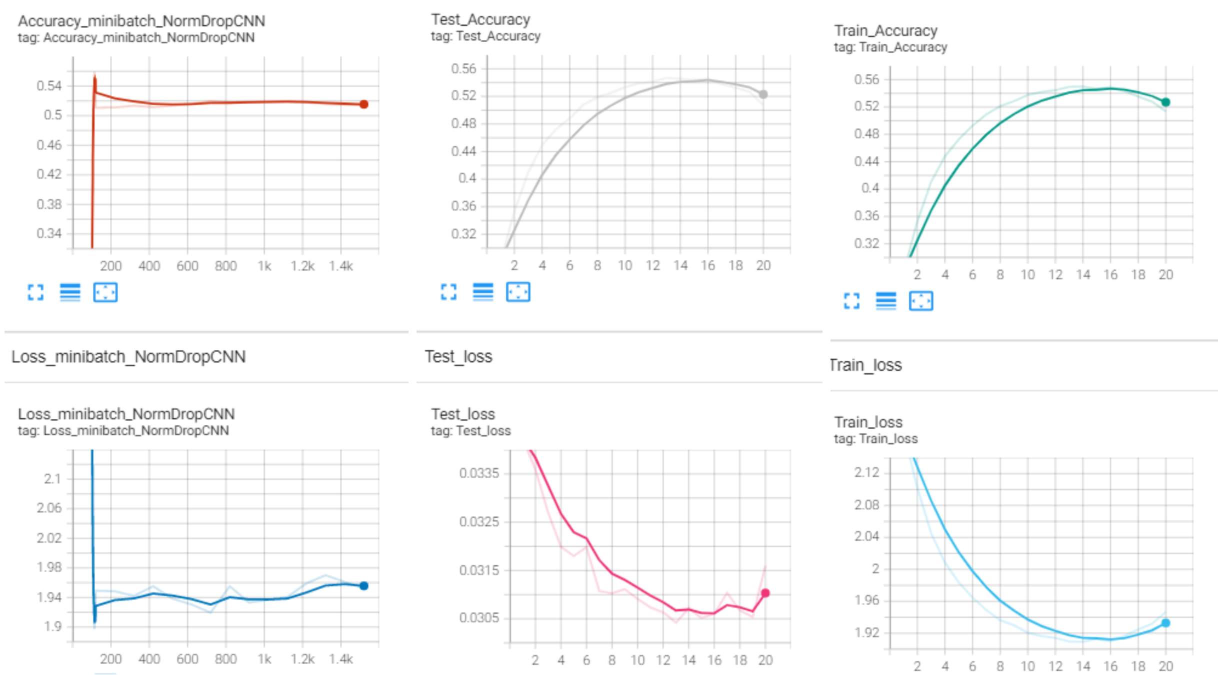
```
[45] nclasses = 10
     set_seed(0)
     img_batch = img_t.to(device=device)

     model3 = Net_b(nclasses, img_batch[0]).to(device=device)
     optimizer = optim.SGD(model3.parameters(), lr=1e-2)
     loss_fn = nn.CrossEntropyLoss()

     training_loop(
         n_epochs=20,
         optimizer=optimizer,
         model=model3,
         loss_fn=loss_fn,
         l2_regularizer=1e-8,
         train_loader=training_generator_aug,
         test_loader = test_generator,
         tag_txt = "NormDropCNN"
     )
```

Lets look at the results.



We see many errors here, and we use the augmented dataset.

Lets look at the graphs.



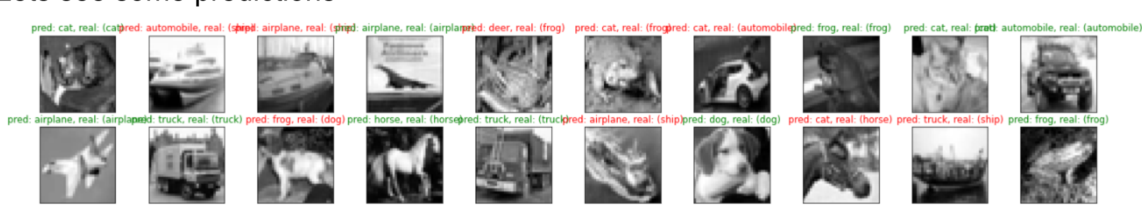
The accuracy in minibacthes increase alot in the beginning but it decrease a bit and stays the same.
The accuracy in train and test set are about 0.52 both, lower than two models before.
The losses follow the same trend.
One problem would be the learning rate! I am using 1e-2 and it might be too big, or the regulizers are too small or big.

Using original dataset and smaller regulaizers.

lr=1e2, l2_regulizer=1e-7, the dataset is different now.

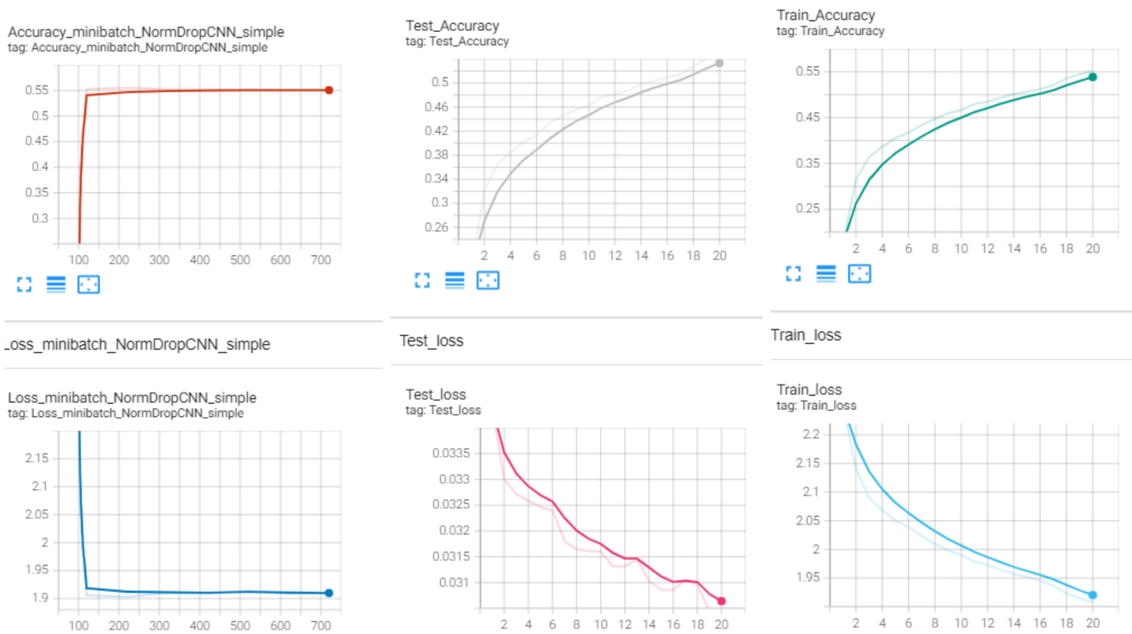
```
nclasses = 10
set_seed(0)
img_batch = img_t.to(device=device)

model4 = Net_b(nclasses, img_batch[0]).to(device=device)
optimizer = optim.SGD(model4.parameters(), lr=1e-2)
loss_fn = nn.CrossEntropyLoss()

training_loop(
    n_epochs=20,
    optimizer=optimizer,
    model=model4,
    loss_fn=loss_fn,
    l2_regularizer=1e-7,
    train_loader=training_generator,
    test_loader = test_generator,
    tag_txt = "NormDropCNN_simple"
)
```

Lets see some predictions



The prediction is a bit better than before.

Lets see the graphs.

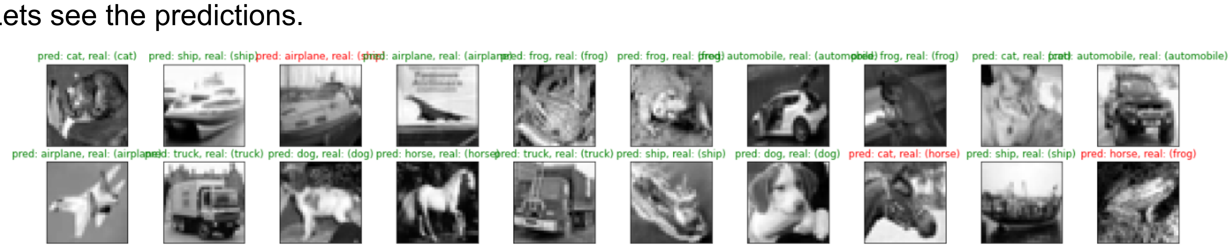


This time it has better predictions. Losses are always decreasing and accuracies are increasing. If we trained them for longer it might have a better result.

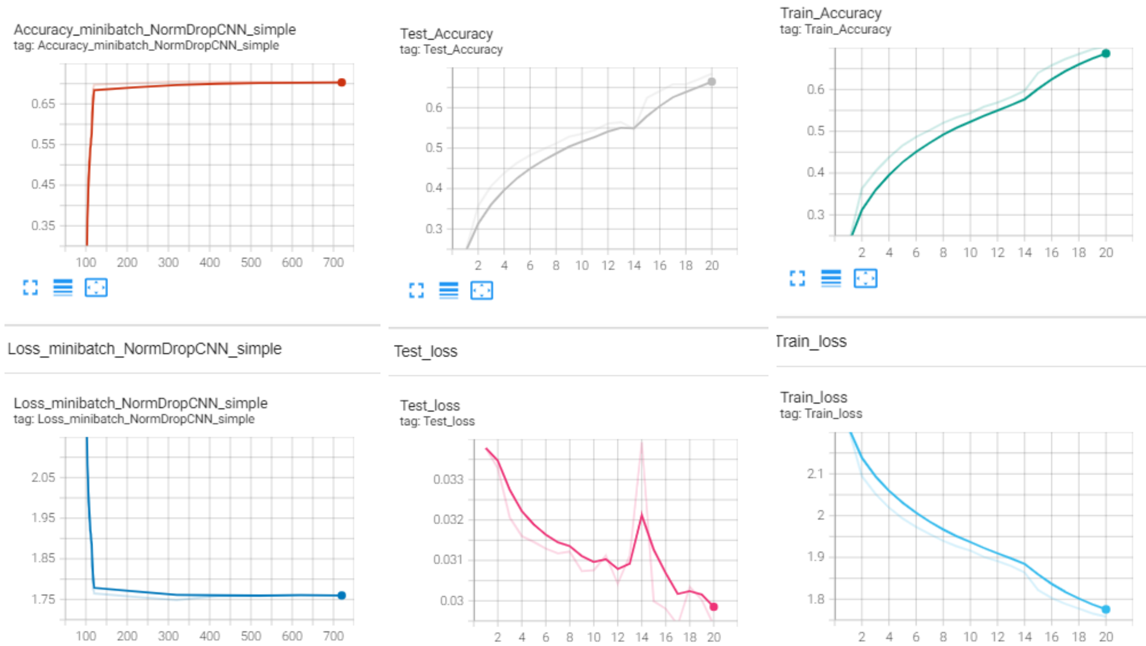
Using L1 regulizers with simple Netowrk(No dropout). Original dataset.

```
[55] nclasses = 10
     set_seed(0)
     img_batch = img_t.to(device=device)

     model5 = Net(nclasses, img_batch[0]).to(device=device)
     optimizer = optim.SGD(model5.parameters(), lr=1e-2)
     loss_fn = nn.CrossEntropyLoss()

     training_loop(
         n_epochs=20,
         optimizer=optimizer,
         model=model5,
         loss_fn=loss_fn,
         l1_regularizer=1e-8,
         train_loader=training_generator,
         test_loader = test_generator,
         tag_txt = "NormDropCNN_simple"
     )
```

Lets see the predictions.



This is really good. Only 3 worng predictions here.

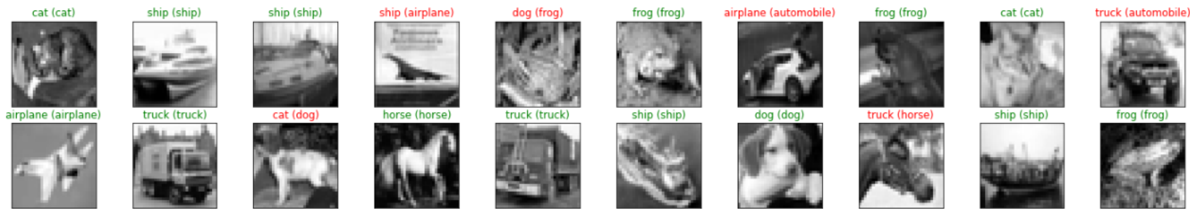


Until now we have the best accuracy and loss in both train and test.
The accuracy in train increased to 0.704 and in test it is 0.6841.
Better not to forget we are using original dataset, and we can see the regulizer effect.

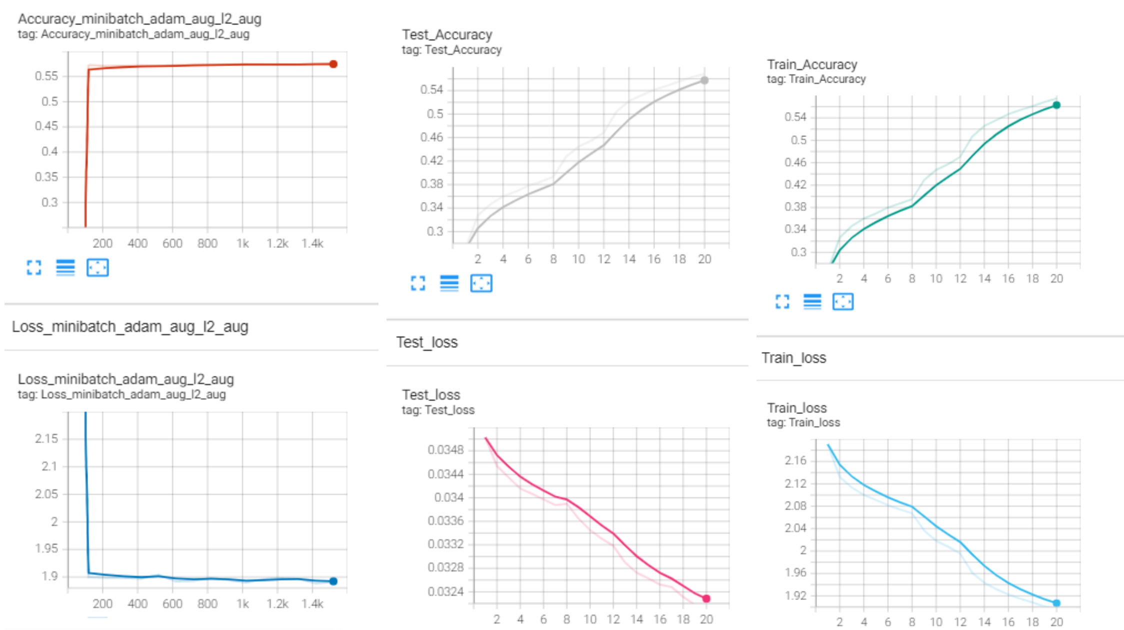
Different oprimizers.

```
nclasses = 10
set_seed(0)
img_batch = img_t.to(device=device)

model6 = Net(nclasses, img_batch[0]).to(device=device)
optimizer = optim.Adam(model6.parameters(), lr=1e-5)
loss_fn = nn.CrossEntropyLoss()

training_loop(
    n_epochs=20,
    optimizer=optimizer,
    model=model6,
    loss_fn=loss_fn,
    l1_regularizer=1e-8,
    train_loader=training_generator_aug,
    test_loader = test_generator,
    tag_txt = "adam_aug_l2_aug"
)
```

We first use Adam optimizer with lr=1e-5 and l1_regulizer=1e-8
We use the augmented dataset and 20 epochs. As our previous experience we may not get better result with augmented dataset. But using lower learning rate and regulizer might help. There is no model with drop out used here. (Wrong tag_text as I was experimenting)

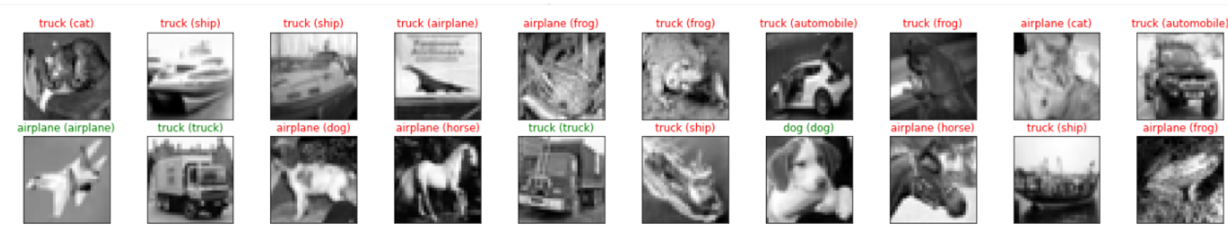


Interestingly we get a good prediction at least in this 20 images. Only 6 mistakes.
Lets see the graphs

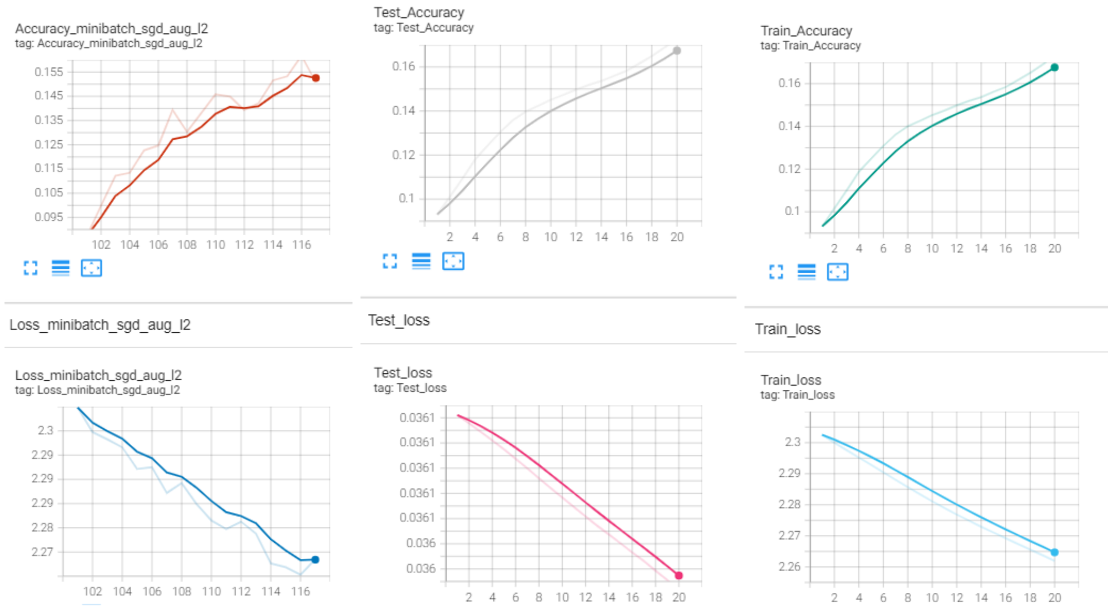


The results are not bad, almost 0.55 accuracy in both train and test and the loss trend is decreasing. Maybe if trained it for a longer time it gave us a better result.

Using SGD, with the same parameters as above.
Let's see the predictions.



Not good.
Graphs may tell us more.



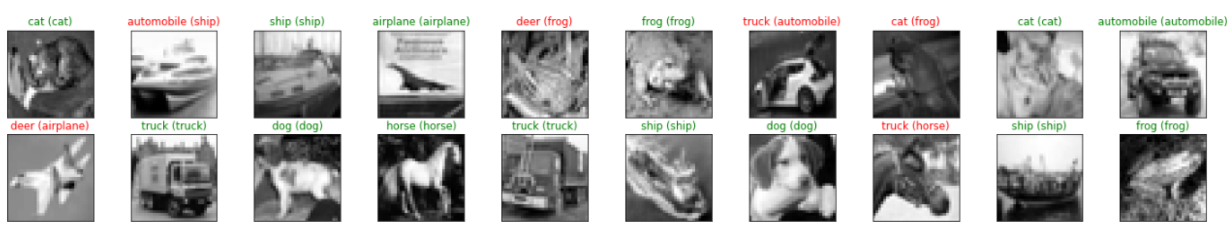
I think because our learning rate is small and the graphs are showing the accuracies are increasing we just need to train it for more epochs.

So we train both for another 50 epochs, but the learning rate is a bit bigger this time.
Adam for 50 epochs. lr=1e-4

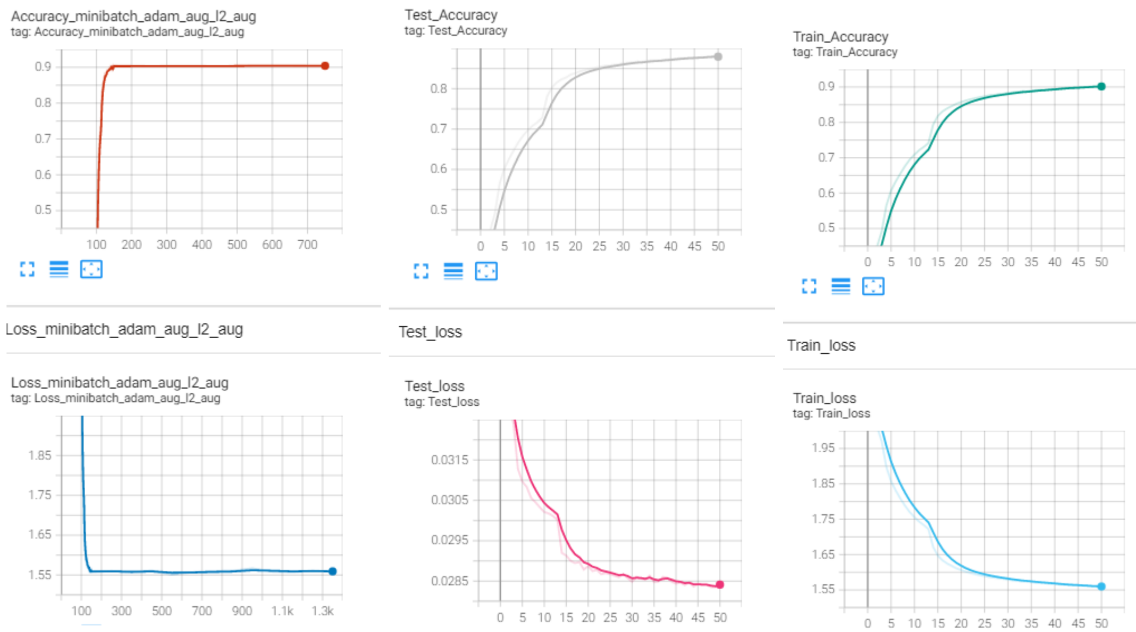
```
[79] logs = 'Logs/adam_aug_l2'
     writer = SummaryWriter(logs)
```

```
nclasses = 10
set_seed(0)
img_batch = img_t.to(device=device)

model6 = Net(nclasses, img_batch[0]).to(device=device)
optimizer = optim.Adam(model6.parameters(), lr=1e-4)
loss_fn = nn.CrossEntropyLoss()

training_loop(
    n_epochs=50,
    optimizer=optimizer,
    model=model6,
    loss_fn=loss_fn,
    l1_regularizer=1e-8,
    train_loader=training_generator_aug,
    test_loader = test_generator,
    tag_txt = "adam_aug_l2_aug"
)
```

Let’s see the result.



6 mistakes, like previous time but different labels.
Let’s see the graphs

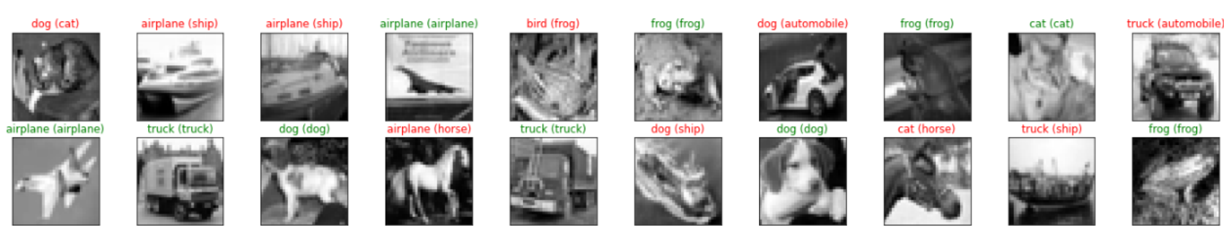


interestingly we have good accuracies and losses which is even visible in the minibatches.
As the train accuracy is increasing but very slowly there might be a hope of change but i think 50 epochs is enough.
Compared to previous time which both accuracies were almost 0.55. Here we have 0.9026 for train accuracy and 0.8799 for test accuracy. The loss has a good trend as well.

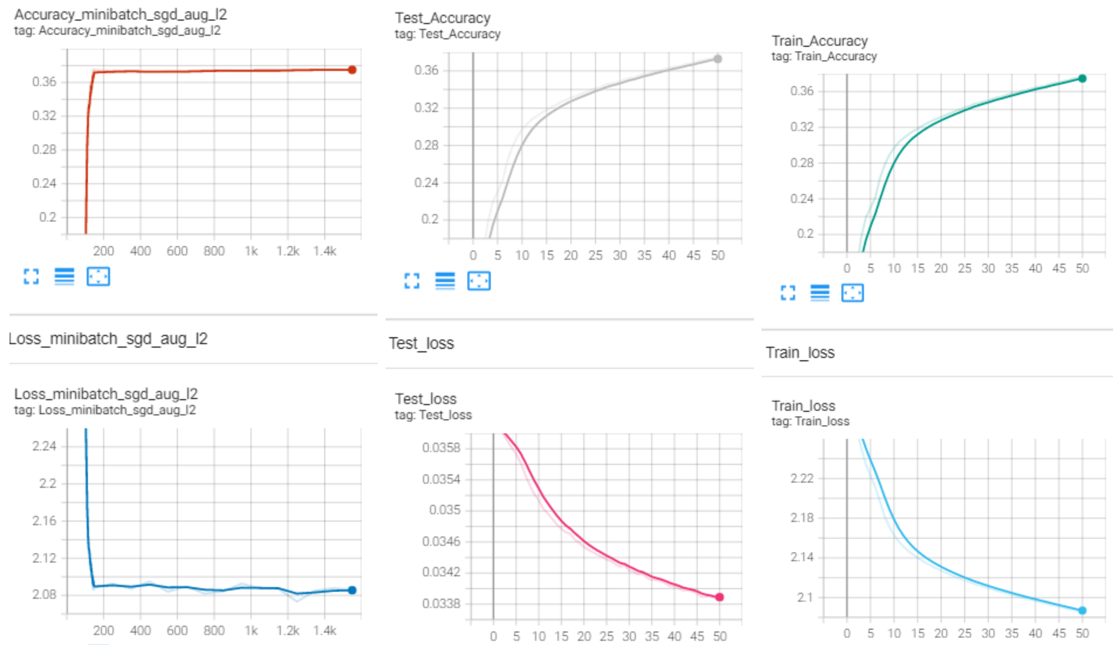
Let's run SGD optimizer for 50 epochs and lr=1e-4 and l1_regulizer=1e-8.

```
nclasses = 10
set_seed(0)
img_batch = img_t.to(device=device)

model7 = Net(nclasses, img_batch[0]).to(device=device)
optimizer = optim.SGD(model7.parameters(), lr=1e-4)
loss_fn = nn.CrossEntropyLoss()

training_loop(
    n_epochs=50,
    optimizer=optimizer,
    model=model7,
    loss_fn=loss_fn,
    l1_regularizer=1e-8,
    train_loader=training_generator_aug,
    test_loader = test_generator,
    tag_txt = "sgd_aug_l2"
)
```

Prediction



Previous time there was only 4 correct predictions but this time it is 10 .
Graphs.



Everything is better than previous run, but seems like the learning rate is too small for sgd to get better score. Otherwise there is hope for improvement because the accuracy is still increasing.

Log_softmax with augmented dataset and regulizer

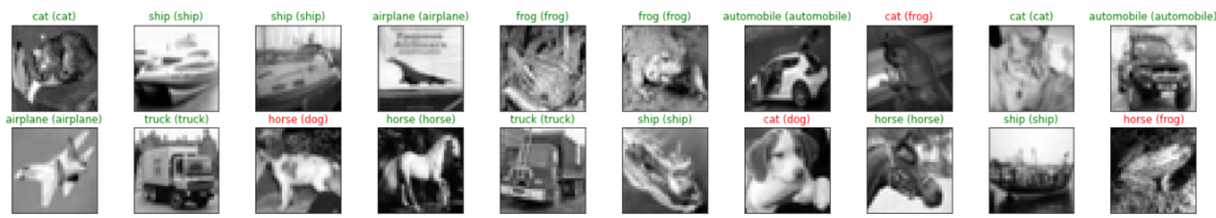
Searching for how to make this CNN led me to finding log_softmax, which i used for my final Network and you can see the result here.
To my very first model i only make one change and it is the final softmax changed to log_softmax.

```
def forward(self, x):
    out = F.max_pool2d(F.relu(self.conv1(x)),2)
    out = (F.relu(self.conv2(out)))
    out = F.max_pool2d(F.relu(self.conv3(out)),2)
    #flatten, we could also use torch.flatten()
    #out = out.view(-1, self.nchans3 * self.nflat)
    out = torch.flatten(out, 1) # flatten all dimensions except batch
    out = F.relu(self.fc1(out))
    out = F.relu(self.fc2(out))
    out = F.relu(self.fc3(out))

    return F.log_softmax(out, dim=1)
```

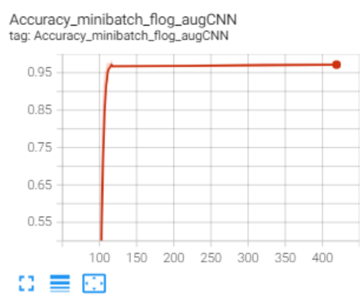
Let’s run the model

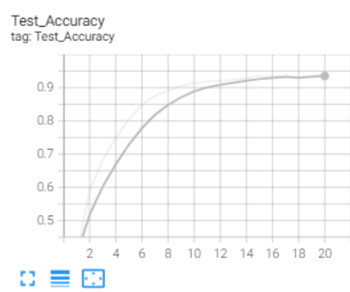
```python
nclasses = 10
set_seed(0)
img_batch = img_t.to(device=device)

model8 = Net_f(nclasses, img_batch[0]).to(device=device)
optimizer = optim.SGD(model8.parameters(), lr=1e-2)
loss_fn = nn.CrossEntropyLoss()

training_loop(
    n_epochs=20,
    optimizer=optimizer,
    model=model8,
    loss_fn=loss_fn,
    l2_regularizer=1e-8,
    train_loader=training_generator_aug,
    test_loader = test_generator,
    tag_txt = "flog_augCNN"
)
```

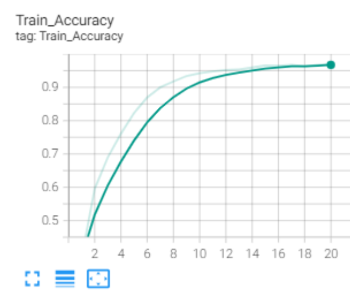
20 epochs, lr=1e-2, l1_regulizer=1e-8 using augmented dataset.
Let's see the prediction.



Only 4 mistakes.



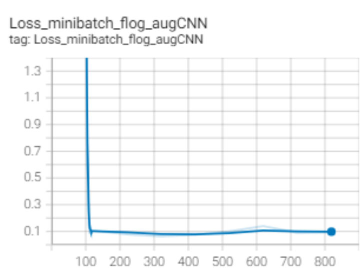


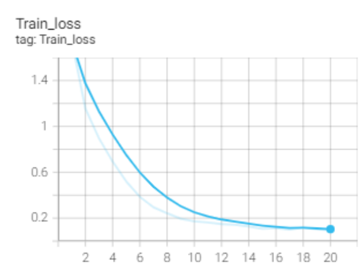


Loss_minibatch_flog_augCNN

Test_loss

Train_loss





0.9711 train accuracy rate, 0.9312 test accuracy rate.
Best result so far.