HW1_Q3

February 17, 2019

1 HW1 Q3 - Convolutional Neural Networks

In this question, we set up a CNN in order to perform the 'Cats vs Dogs' Kaggle challenge
First, we import some PyTorch libraries and direct them to use Google Colab's GPU:

```
In [0]: import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.utils.data
        import torchvision
        import torchvision.transforms as transforms
        from torch.utils.data import DataLoader, Sampler, Dataset
        from torchvision import datasets
        from torch.utils.data.sampler import SubsetRandomSampler, SequentialSampler
        from torch.autograd import Variable

        import os
        import numpy as np
        import PIL.Image
        import glob
        import matplotlib.pyplot as plt
```

```
In [0]: # Use the GPU
        device = torch.device('cuda')
```

In order to import the data for Kaggle from Google Drive, we need to jump through a few hoops, as seen below

```
In [0]: # Import the data

        from google.colab import drive

        drive.mount('/content/drive/')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6

Enter your authorization code:
ůůůůůůůůůů
Mounted at /content/drive/
```

```
In [0]: PATH = '/content/drive/My Drive/DL_A1/'


        # apply several different transforms, as necessary
        rotate = [-90, 90]
        transform = transforms.Compose([
            transforms.ToPILImage(),
            transforms.RandomRotation(rotate),
            transforms.RandomHorizontalFlip(p=0.25),
            transforms.RandomResizedCrop(64, scale=(0.75, 1.0)),
            transforms.ToTensor()
        ])

        # for the hyperparameter tunning, no transforms were applied

        # transform = transforms.Compose([
        #     transforms.ToPILImage(),
        #     transforms.ToTensor()
        # ])

        # load the data, which was downloaded and saved
        Data = np.load(PATH +'data.npz')
```

```
In [0]: # we define the dataset class for the train, valid and test data
        class dataset(Dataset):
          def __init__(self, X, Y, transform=None):
            self.X = X
            self.Y = Y
            self.transform = transform

          def __len__(self):
            return self.X.shape[0]

          def __getitem__(self, index):
            #temp = self.transform(self.X[index,:])
            #return temp, self.Y[index]\
            sequence, target = self.X[index], self.Y[index]
            if self.transform is not None:
              sequence = self.transform(sequence)
            if target is None:
              return sequence
            return sequence, target
```

```
In [0]: # define the labels and datasets

        test_labels = [None]*Data['Xte'].shape[0]
        train_dataset = dataset(Data['Xtr'], Data['y'], transform)
        test_dataset = dataset(Data['Xte'], test_labels, transform)
```

```
testIDS_list = np.hsplit(Data['testIds'], 1)
testIDS = testIDS_list[0]
```

In [0]:
```
# we will define this information now
batch_size = 64
learning_rate = 0.005
num_epochs = 100

num_classes = 2
num_of_workers = 4
```

To split the training data into a training and validation set, we need to establish a `sampler` for each that will pull out certain samples when called from the dataloader. In our case, we have a 20% validation split, with 16000 training examples and 4000 validation examples.

In [0]:
```
# Define a shuffled train and valid index list (non-overlapping)

a = np.arange(19998)
np.random.shuffle(a)

num_train_samples = 16000
num_valid_samples = 19998 - num_train_samples

train_index = a[:num_train_samples]
valid_index = a[num_train_samples:
                num_train_samples+num_valid_samples]
```

In [0]:
```
# Fix seed
np.random.seed(42)
torch.manual_seed(42)

# Define samplers for the data (to get validation split)

train_sampler = SubsetRandomSampler(train_index)

valid_sampler = SubsetRandomSampler(valid_index)
```

As a last step before training, we define training, validation, and test dataloaders that sample from the above `samplers` according to the defined batch size:

In [0]:
```
# Get dataloaders for train/valid/test
train_loader = torch.utils.data.DataLoader(train_dataset,
                                           batch_size=batch_size,
                                           sampler=train_sampler)

valid_loader = torch.utils.data.DataLoader(train_dataset,
                                           batch_size=batch_size,
                                           sampler=valid_sampler)
```

1.1 CNN architecture definition

Now we define the class for our CNN.

```
In [0]: # this is just a function to help with input/output size
        def output_size(input_size, kernel_size, stride, padding):
            '''
            Helper function to determine output size of a convolution/pooling.
            '''
            output = int((input_size - kernel_size + 2*(padding)) / stride) + 1
            return output
```

```
In [0]: # hyperparameter tunning
        # Define the CNN class

        class CNN(nn.Module):
          def __init__(self):
            super(CNN, self).__init__()
            self.conv1 = nn.Conv2d(3, 40, kernel_size=3, stride=1, padding=1)
            self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

            self.conv2 = nn.Conv2d(40, 40, kernel_size=3, stride=1, padding=1)
            self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

            self.fc1 = nn.Linear(10240, 175)
            self.fc2 = nn.Linear(175, 2)

          def forward(self, x):
            x = self.conv1(x)
            x = F.relu(x)
            x = self.pool1(x)
            x = F.relu(self.conv2(x))
            x = self.pool2(x)
            x = x.view(len(x), 10240)
            x = self.fc1(x)
            x = F.relu(x)
            x = self.fc2(x)
        #     x = F.softmax(x)nYT `yyb5

            return x
```

This CNN has two convolutional layers, each with a ReLU activation, a kernel size of 3, a stride of 1, and a padding level of 1. The first convolutional layer learns 18 feature maps and the second learns 64 feature maps. Both convolutional layers use max pooling after their operation, with a kernel size of 2 and stride of 2. During hyperparameter tuning, these values were modified

Finally, the data will pass through two fully connected layers and output to 2 units representing either a 'Cat' or a 'Dog'.

Below, we see an outline of this architecture. Note that the total number of trainable parameters is around 1.8 million per sample

```
In [0]: model = CNN().to(device)
        print(model)

        model_parameters = filter(lambda p: p.requires_grad, model.parameters())
        params = sum([np.prod(p.size()) for p in model_parameters])
        print('Number of parameters: {}'.format(params))
```

```
CNN(
  (conv1): Conv2d(3, 40, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(40, 40, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=10240, out_features=175, bias=True)
  (fc2): Linear(in_features=175, out_features=2, bias=True)
)
Number of parameters: 1808087
```

1.2 CNN training

Now we will define our loss function and stochastic gradient descent optimizer, and run over the training loop for our CNN.

```
In [0]: # Define loss and optimizer

        # depending on the method, either NLL or cross entropy loss was used (the latter inclu
        # the former allows for easy visualization of the data
        criterion = nn.CrossEntropyLoss()
        # criterion = nn.NLLLoss()
        optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

seed weight for reproducibility Please note: originally, the code was run without a seed. We do understand that initial values are important in order to allow for reproducibility, but it wasn't done early enough to allow for seeding. As a result, while the code is set up for seeding the weights, running the code might not give the desired results on the first run.

```
In [0]: # initialize seed weights for reproducability
        if type(CNN) in [nn.Conv2d, nn.Linear, nn.MaxPool2d ]:
          CNN.weight.data = normal_(1.0, 0.02)
          CNN.bias.data = fill_(0)
```

2 Below is the training and validation loop

```
In [0]: # Make a nice progress bar - tqdm on colab is a bit buggy though
        from tqdm import tqdm
```

```python
best_val_acc = 0
# Train
print('\nBeginning training.')
total_step = len(train_loader)
train_losses, train_accuracies = [], []
valid_losses, valid_accuracies = [], []

for epoch in range(num_epochs):
  print('Epoch {}/{}'.format(epoch+1, num_epochs))
  # Training set
  model.train()
  total_tr, correct_tr = 0, 0
  total_val, correct_val = 0, 0
  mean_train_loss = 0.0
  mean_valid_loss = 0.0
  mean_train_acc = 0.0
  mean_valid_acc = 0.0

  output_clear = []
  output_ambig = []
  misclass = []
  for i, (images, labels) in enumerate(tqdm(train_loader, position=0)):

    labels = torch.max(labels, 1)[1]
    images, labels = images.to(device), labels.long().to(device)
    optimizer.zero_grad()


    # Forward
    outputs = model(images)

    # Check if classifications are close (ambiguous) or far apart (clear)
    for j in range(outputs.shape[0]):
      outputs_cpu = outputs.cpu().detach().numpy()
      if abs(outputs_cpu[j,1]-outputs_cpu[j,0] > 0.4):
        output_clear.append(images[j].cpu().numpy())
      if abs(outputs_cpu[j,1]-outputs_cpu[j,0] < 0.1):
        output_ambig.append(images[j].cpu().numpy())


    loss = criterion(outputs, labels)




# try this implementation
    lambda1 = 0.5
    lambda2 = 2.75
```

```
    l1_regularization, l2_regularization = torch.tensor(0), torch.tensor(0)
#     for param in model.parameters():
#       l1_regularization += torch.norm(param, 1).long()
#       l2_regularization += torch.norm(param, 2).long()
#     loss = criterion(outputs, labels) + lambda2 * l2_regularization
    mean_train_loss += loss

    # Backward
    loss.backward()
    optimizer.step()

#     modify learning rate, if desired
#     if epoch%200 == 0:
#       learning_rate = learning_rate/5
#       optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

    # Track loss and accuracy

    total_tr += labels.size(0)
    _, predicted_tr = torch.max(outputs.data, 1)
    correct_tr += (predicted_tr == labels).sum().item()
    incorrect_tr = (predicted_tr != labels).cpu().detach().numpy()

    # Keep track of wrongly classified items
    for k in range(outputs.shape[0]):
      if incorrect_tr[k] == 1: # If misclassified
        misclass.append(images[k].cpu().numpy())
    acc_tr = correct_tr/total_tr
    mean_train_acc += acc_tr


  mean_train_loss /= i+1
  mean_train_acc /= i+1
  pred_total = 0
  labels_total = 0

  # Validation set
  model.eval()
  for i, (images, labels) in enumerate(valid_loader):

    labels = torch.max(labels, 1)[1]
    images, labels = images.to(device), labels.long().to(device)
    valid_outputs = model(images)
    valid_loss = criterion(valid_outputs, labels)
    mean_valid_loss += valid_loss

    total_val += labels.size(0)
    _, predicted_val = torch.max(valid_outputs.data, 1)
```

```
          correct_val += (predicted_val == labels).sum().item()
          valid_acc = correct_val/total_val
          mean_valid_acc += valid_acc
        mean_valid_loss /= i+1
        mean_valid_acc /= i+1



        train_losses.append(mean_train_loss.item())
        train_accuracies.append(mean_train_acc)
        valid_losses.append(mean_valid_loss.item())
        valid_accuracies.append(mean_valid_acc)
        # Mod this when num epochs is too big
        print('\ntrain_loss: {}, train_acc: {}, valid_loss: {}, valid_acc: {}'.format(
            mean_train_loss.item(), mean_train_acc*100, mean_valid_loss.item(), mean_valid_ac
```

```
  0%|          | 0/250 [00:00<?, ?it/s]

Beginning training.
Epoch 1/100

100%|| 250/250 [00:17<00:00, 14.30it/s]


train_loss: 0.6930142641067505, train_acc: 49.36486159234167, valid_loss: 0.6922606229782104, v
Epoch 2/100

100%|| 250/250 [00:17<00:00, 14.40it/s]


train_loss: 0.6922444105148315, train_acc: 52.8340007855049, valid_loss: 0.6912415027618408, va
Epoch 3/100

100%|| 250/250 [00:17<00:00, 14.50it/s]


train_loss: 0.6914874911308289, train_acc: 54.95353264798657, valid_loss: 0.69083571434021, val
Epoch 4/100

100%|| 250/250 [00:17<00:00, 14.51it/s]
```

```
train_loss: 0.6907480359077454, train_acc: 55.19348766675375, valid_loss: 0.6905655860900879, v
Epoch 5/100

100%|| 250/250 [00:17<00:00, 14.36it/s]


train_loss: 0.6896104216575623, train_acc: 55.34413896172211, valid_loss: 0.6903660893440247, v
Epoch 6/100

100%|| 250/250 [00:17<00:00, 14.50it/s]


train_loss: 0.6883096694946289, train_acc: 56.18287752933874, valid_loss: 0.6873754858970642, v
Epoch 7/100

100%|| 250/250 [00:17<00:00, 14.53it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6874707341194153, train_acc: 56.514254546765564, valid_loss: 0.6864235401153564,
Epoch 8/100

100%|| 250/250 [00:17<00:00, 14.56it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6858648657798767, train_acc: 57.16382512194684, valid_loss: 0.6853871941566467, v
Epoch 9/100

100%|| 250/250 [00:17<00:00, 14.60it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6842032670974731, train_acc: 56.20940964915641, valid_loss: 0.683220624923706, va
Epoch 10/100

100%|| 250/250 [00:17<00:00, 14.70it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6820480823516846, train_acc: 56.76872600315477, valid_loss: 0.6794880628585815, v
Epoch 11/100
```

```
100%|| 250/250 [00:17<00:00, 14.74it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.6799564957618713, train_acc: 57.269154100209555, valid_loss: 0.6772283315658569,
Epoch 12/100


100%|| 250/250 [00:17<00:00, 14.68it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.677191436290741, train_acc: 58.33161262297411, valid_loss: 0.6742280721664429, va
Epoch 13/100


100%|| 250/250 [00:17<00:00, 13.78it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.6744272112846375, train_acc: 58.664599682431714, valid_loss: 0.670024037361145, v
Epoch 14/100


100%|| 250/250 [00:17<00:00, 14.76it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.6713566184043884, train_acc: 59.25405371184442, valid_loss: 0.6723098754882812, v
Epoch 15/100


100%|| 250/250 [00:16<00:00, 14.82it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.6683833599090576, train_acc: 59.477089042439516, valid_loss: 0.6641493439674377,
Epoch 16/100


100%|| 250/250 [00:17<00:00, 14.79it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.6645286083221436, train_acc: 59.6185137949265, valid_loss: 0.65889972448349, vali
Epoch 17/100


100%|| 250/250 [00:17<00:00, 14.54it/s]
  0%|          | 0/250 [00:00<?, ?it/s]
```

```
train_loss: 0.6617603898048401, train_acc: 61.03695473268134, valid_loss: 0.6610053777694702, v
Epoch 18/100

100%|| 250/250 [00:17<00:00, 14.46it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6592349410057068, train_acc: 60.78294433662011, valid_loss: 0.6555015444755554, v
Epoch 19/100

100%|| 250/250 [00:17<00:00, 14.52it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6559522747993469, train_acc: 60.68106177824072, valid_loss: 0.6497381925582886, v
Epoch 20/100

100%|| 250/250 [00:17<00:00, 14.76it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.652971625328064, train_acc: 61.3318756091554, valid_loss: 0.6482619047164917, val
Epoch 21/100

100%|| 250/250 [00:17<00:00, 14.69it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6491773724555969, train_acc: 61.71482844314302, valid_loss: 0.6507822871208191, v
Epoch 22/100

100%|| 250/250 [00:17<00:00, 14.51it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6485158205032349, train_acc: 61.78846818135355, valid_loss: 0.6665711998939514, v
Epoch 23/100

100%|| 250/250 [00:17<00:00, 14.60it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6440988779067993, train_acc: 61.927212259849576, valid_loss: 0.6365282535552979,
Epoch 24/100
```

```
100%|| 250/250 [00:17<00:00, 14.72it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6386667490005493, train_acc: 63.47673544492009, valid_loss: 0.6365890502929688, v
Epoch 25/100

100%|| 250/250 [00:17<00:00, 14.71it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6338826417922974, train_acc: 64.18960594301632, valid_loss: 0.6466444730758667, v
Epoch 26/100

100%|| 250/250 [00:17<00:00, 14.48it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6327885389328003, train_acc: 63.71404607730034, valid_loss: 0.626064658164978, va
Epoch 27/100

100%|| 250/250 [00:17<00:00, 14.41it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6274076700210571, train_acc: 64.48343149515847, valid_loss: 0.6323862075805664, v
Epoch 28/100

100%|| 250/250 [00:17<00:00, 14.70it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6246305704116821, train_acc: 65.94733448286837, valid_loss: 0.6230015158653259, v
Epoch 29/100

100%|| 250/250 [00:17<00:00, 14.44it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.6215811967849731, train_acc: 65.29609120900301, valid_loss: 0.6184007525444031, v
Epoch 30/100

100%|| 250/250 [00:17<00:00, 14.65it/s]
  0%|          | 0/250 [00:00<?, ?it/s]
```

```
train_loss: 0.6183581948280334, train_acc: 65.9617543236825, valid_loss: 0.6223486065864563, va
Epoch 31/100


100%|| 250/250 [00:17<00:00, 14.63it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.6158603429794312, train_acc: 66.02656035852038, valid_loss: 0.6143268942832947, v
Epoch 32/100


100%|| 250/250 [00:17<00:00, 14.73it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.614260196685791, train_acc: 67.02422142149942, valid_loss: 0.6176616549491882, va
Epoch 33/100


100%|| 250/250 [00:17<00:00, 14.67it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.6106559634208679, train_acc: 66.24181773197915, valid_loss: 0.6119679808616638, v
Epoch 34/100


100%|| 250/250 [00:17<00:00, 14.80it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.609559178352356, train_acc: 66.94976790583601, valid_loss: 0.6042861342430115, va
Epoch 35/100


100%|| 250/250 [00:17<00:00, 14.47it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.6079224348068237, train_acc: 66.61432287014311, valid_loss: 0.601917564868927, va
Epoch 36/100


100%|| 250/250 [00:17<00:00, 14.65it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.6046432256698608, train_acc: 67.851929275642, valid_loss: 0.6102263331413269, val
Epoch 37/100
```

```
100%|| 250/250 [00:17<00:00, 14.54it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.6040030121803284, train_acc: 66.98571870837252, valid_loss: 0.6275973916053772, v
Epoch 38/100


100%|| 250/250 [00:17<00:00, 14.73it/s]
  1%|          | 2/250 [00:00<00:16, 14.78it/s]


train_loss: 0.6004275679588318, train_acc: 68.06119648453677, valid_loss: 0.6017123460769653, v
Epoch 39/100


100%|| 250/250 [00:17<00:00, 14.74it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.6010342240333557, train_acc: 68.16332785004413, valid_loss: 0.6173083186149597, v
Epoch 40/100


100%|| 250/250 [00:17<00:00, 14.59it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5991964936256409, train_acc: 67.79102951454881, valid_loss: 0.6073956489562988, v
Epoch 41/100


100%|| 250/250 [00:17<00:00, 14.53it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5985295176506042, train_acc: 68.46334260297033, valid_loss: 0.6038658618927002, v
Epoch 42/100


100%|| 250/250 [00:17<00:00, 14.65it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.598051130771637, train_acc: 67.64258619453877, valid_loss: 0.6016388535499573, va
Epoch 43/100


100%|| 250/250 [00:17<00:00, 14.79it/s]
  0%|          | 0/250 [00:00<?, ?it/s]
```

```
train_loss: 0.5968722701072693, train_acc: 68.58369244422457, valid_loss: 0.5983814597129822,
Epoch 44/100


100%|| 250/250 [00:17<00:00, 14.79it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5956230163574219, train_acc: 68.12071870355969, valid_loss: 0.6024935841560364,
Epoch 45/100


100%|| 250/250 [00:16<00:00, 14.81it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5946987867355347, train_acc: 68.30726933676709, valid_loss: 0.6064804792404175,
Epoch 46/100


100%|| 250/250 [00:16<00:00, 14.93it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5921802520751953, train_acc: 68.90593811303553, valid_loss: 0.5953626036643982,
Epoch 47/100


100%|| 250/250 [00:16<00:00, 14.86it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5894041657447815, train_acc: 69.11685014943917, valid_loss: 0.5967847108840942,
Epoch 48/100


100%|| 250/250 [00:17<00:00, 14.64it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5897251963615417, train_acc: 69.10082912841649, valid_loss: 0.5906380414962769,
Epoch 49/100


100%|| 250/250 [00:16<00:00, 14.58it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5877701640129089, train_acc: 68.66142026484599, valid_loss: 0.5932918787002563,
Epoch 50/100
```

```
100%|| 250/250 [00:17<00:00, 14.63it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5878379940986633, train_acc: 69.47558335868574, valid_loss: 0.5921817421913147, v
Epoch 51/100

100%|| 250/250 [00:17<00:00, 14.76it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5890380144119263, train_acc: 69.56559688117349, valid_loss: 0.5901303887367249, v
Epoch 52/100

100%|| 250/250 [00:17<00:00, 14.83it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5854870676994324, train_acc: 69.113152057261, valid_loss: 0.5888407826423645, val
Epoch 53/100

100%|| 250/250 [00:16<00:00, 14.60it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5832918882369995, train_acc: 69.05465779720448, valid_loss: 0.5896896123886108, v
Epoch 54/100

100%|| 250/250 [00:17<00:00, 14.76it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5833215713500977, train_acc: 69.49019245520809, valid_loss: 0.5848877429962158, v
Epoch 55/100

100%|| 250/250 [00:17<00:00, 13.93it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5828086137771606, train_acc: 69.10205661352929, valid_loss: 0.5880361795425415, v
Epoch 56/100

100%|| 250/250 [00:17<00:00, 14.49it/s]
  0%|          | 0/250 [00:00<?, ?it/s]
```

```
train_loss: 0.581347644329071, train_acc: 69.05186397303, valid_loss: 0.5888998508453369, valid
Epoch 57/100


100%|| 250/250 [00:17<00:00, 14.70it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5802642107009888, train_acc: 69.62820056124082, valid_loss: 0.5893005728721619, v
Epoch 58/100


100%|| 250/250 [00:17<00:00, 14.59it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5812596678733826, train_acc: 69.09458398963424, valid_loss: 0.5822047591209412, v
Epoch 59/100


100%|| 250/250 [00:17<00:00, 14.80it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5772011876106262, train_acc: 69.59484254985644, valid_loss: 0.58188796043396, val
Epoch 60/100


100%|| 250/250 [00:17<00:00, 14.77it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5780349373817444, train_acc: 69.9226445132165, valid_loss: 0.5814523100852966, va
Epoch 61/100


100%|| 250/250 [00:17<00:00, 14.61it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.574999988079071, train_acc: 69.37279546092803, valid_loss: 0.5815266370773315, va
Epoch 62/100


100%|| 250/250 [00:17<00:00, 14.70it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5756095051765442, train_acc: 70.60843480027106, valid_loss: 0.5766737461090088, v
Epoch 63/100
```

```
100%|| 250/250 [00:17<00:00, 14.69it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5738035440444946, train_acc: 70.0028434845498, valid_loss: 0.5781043171882629, va
Epoch 64/100


100%|| 250/250 [00:17<00:00, 14.71it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5741413831710815, train_acc: 70.55229594798197, valid_loss: 0.576945960521698, va
Epoch 65/100


100%|| 250/250 [00:17<00:00, 14.62it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5719600915908813, train_acc: 70.8690256832013, valid_loss: 0.5910728573799133, va
Epoch 66/100


100%|| 250/250 [00:17<00:00, 14.86it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5701560378074646, train_acc: 70.16188890584246, valid_loss: 0.571110725402832, va
Epoch 67/100


100%|| 250/250 [00:17<00:00, 14.54it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5709902048110962, train_acc: 70.40747837291951, valid_loss: 0.5847728252410889, v
Epoch 68/100


100%|| 250/250 [00:16<00:00, 14.90it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5712817311286926, train_acc: 70.36311949269383, valid_loss: 0.5765557289123535, v
Epoch 69/100


100%|| 250/250 [00:16<00:00, 14.89it/s]
  0%|          | 0/250 [00:00<?, ?it/s]
```

```
train_loss: 0.5696108341217041, train_acc: 70.02251258043613, valid_loss: 0.5711027383804321, v
Epoch 70/100


100%|| 250/250 [00:16<00:00, 14.71it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5694392323493958, train_acc: 70.5985007781696, valid_loss: 0.5756178498268127, va
Epoch 71/100


100%|| 250/250 [00:16<00:00, 14.65it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.564736545085907, train_acc: 70.99057362424281, valid_loss: 0.5705124735832214, va
Epoch 72/100


100%|| 250/250 [00:17<00:00, 14.51it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5640832185745239, train_acc: 71.03041590765147, valid_loss: 0.5730744004249573, v
Epoch 73/100


100%|| 250/250 [00:16<00:00, 14.64it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.562656819820404, train_acc: 71.8631107150642, valid_loss: 0.5697087049484253, val
Epoch 74/100


100%|| 250/250 [00:16<00:00, 14.53it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5632374882698059, train_acc: 70.93599194889649, valid_loss: 0.5672594308853149, v
Epoch 75/100


100%|| 250/250 [00:16<00:00, 14.74it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.562930166721344, train_acc: 71.74039209335551, valid_loss: 0.5661502480506897, va
Epoch 76/100
```

```
100%|| 250/250 [00:16<00:00, 14.43it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5633465647697449, train_acc: 71.36235354958073, valid_loss: 0.5710119605064392, v
Epoch 77/100

100%|| 250/250 [00:16<00:00, 14.52it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5632562041282654, train_acc: 70.84514313409731, valid_loss: 0.5615735650062561, v
Epoch 78/100

100%|| 250/250 [00:17<00:00, 14.56it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5623751282691956, train_acc: 71.26489849170353, valid_loss: 0.5676506161689758, v
Epoch 79/100

100%|| 250/250 [00:17<00:00, 14.72it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5586408376693726, train_acc: 71.25325177132191, valid_loss: 0.5629934668540955, v
Epoch 80/100

100%|| 250/250 [00:17<00:00, 14.42it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5597710013389587, train_acc: 71.69856740150352, valid_loss: 0.5627267360687256, v
Epoch 81/100

100%|| 250/250 [00:17<00:00, 14.74it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5564095377922058, train_acc: 71.19056164303844, valid_loss: 0.5609859228134155, v
Epoch 82/100

100%|| 250/250 [00:17<00:00, 14.77it/s]
  0%|          | 0/250 [00:00<?, ?it/s]
```

```
train_loss: 0.5560643672943115, train_acc: 71.81286287379395, valid_loss: 0.5622586011886597, v
Epoch 83/100


100%|| 250/250 [00:17<00:00, 14.84it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5557202696800232, train_acc: 71.58107468499819, valid_loss: 0.561116635799408, va
Epoch 84/100


100%|| 250/250 [00:17<00:00, 14.82it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5578325986862183, train_acc: 71.09968812204488, valid_loss: 0.567905604839325, va
Epoch 85/100


100%|| 250/250 [00:16<00:00, 14.86it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.556057333946228, train_acc: 71.42337739314037, valid_loss: 0.5609319806098938, va
Epoch 86/100


100%|| 250/250 [00:17<00:00, 14.77it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5535361170768738, train_acc: 72.533884153439, valid_loss: 0.5579771995544434, val
Epoch 87/100


100%|| 250/250 [00:17<00:00, 14.56it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5526137351989746, train_acc: 71.96640178070241, valid_loss: 0.5544003248214722, v
Epoch 88/100


100%|| 250/250 [00:17<00:00, 14.65it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5523031949996948, train_acc: 71.739056878272, valid_loss: 0.5543074011802673, val
Epoch 89/100
```

```
100%|| 250/250 [00:17<00:00, 14.77it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5512797832489014, train_acc: 72.81931330620029, valid_loss: 0.570824086661072, v
Epoch 90/100

100%|| 250/250 [00:17<00:00, 14.85it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5512949228286743, train_acc: 72.26880424748289, valid_loss: 0.5508379340171814, v
Epoch 91/100

100%|| 250/250 [00:16<00:00, 14.79it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.551581621170044, train_acc: 72.17026775843843, valid_loss: 0.5466482043266296, va
Epoch 92/100

100%|| 250/250 [00:17<00:00, 14.82it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5496463775634766, train_acc: 72.69515602521541, valid_loss: 0.5543608665466309, v
Epoch 93/100

100%|| 250/250 [00:16<00:00, 14.83it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.5496641993522644, train_acc: 72.70877922905835, valid_loss: 0.548102855682373, va
Epoch 94/100

100%|| 250/250 [00:16<00:00, 14.82it/s]
  0%|          | 0/250 [00:00<?, ?it/s]

train_loss: 0.54891437292099, train_acc: 71.95688047271528, valid_loss: 0.5483821034431458, val
Epoch 95/100

100%|| 250/250 [00:16<00:00, 14.89it/s]
  0%|          | 0/250 [00:00<?, ?it/s]
```

```
train_loss: 0.5465601086616516, train_acc: 71.23897404432243, valid_loss: 0.5500037670135498, v
Epoch 96/100


100%|| 250/250 [00:16<00:00, 14.84it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5469673871994019, train_acc: 72.41770994167301, valid_loss: 0.5437818765640259, v
Epoch 97/100


100%|| 250/250 [00:17<00:00, 14.97it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5439240336418152, train_acc: 72.58893193942164, valid_loss: 0.5469570159912109, v
Epoch 98/100


100%|| 250/250 [00:17<00:00, 14.83it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5442822575569153, train_acc: 72.2527614570947, valid_loss: 0.5547399520874023, va
Epoch 99/100


100%|| 250/250 [00:16<00:00, 14.86it/s]
  0%|          | 0/250 [00:00<?, ?it/s]


train_loss: 0.5428784489631653, train_acc: 72.55627415438018, valid_loss: 0.544049859046936, va
Epoch 100/100


100%|| 250/250 [00:16<00:00, 14.87it/s]



train_loss: 0.5423505902290344, train_acc: 72.40680750136235, valid_loss: 0.5456769466400146, v
```

3 In order to plot the images, we will use the following code. Please note that the loops will take a while to run.

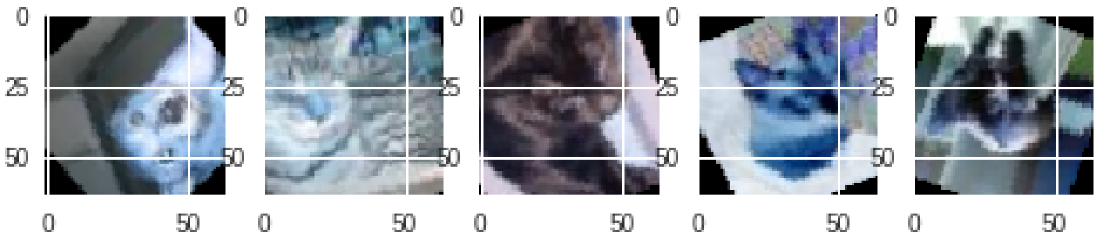
In [0]:
```
output_ambig = np.array(output_ambig)
output_clear = np.array(output_clear)
```

```
misclass = np.array(misclass)

# This is inefficient, but works!
ambig_and_wrong, clear_and_wrong = [], []
for item in misclass:
  if item in output_ambig:
    ambig_and_wrong.append(item)
  elif item in output_clear:
    clear_and_wrong.append(item)

print('We have',len(ambig_and_wrong),' samples that were misclassified and were ambigu
print('We have', len(clear_and_wrong),' samples that were clearly misclassified')
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-2-87c9f051dfde> in <module>()
----> 1 output_ambig = np.array(output_ambig)
      2 output_clear = np.array(output_clear)
      3 misclass = np.array(misclass)


NameError: name 'output_ambig' is not defined
```

4 We can plot the images below. Please note that only some images were chosen for the report

```
In [0]: w=20
        h=20
        fig=plt.figure(figsize=(8, 8))
        columns = 5
        rows = 1
        for i in range(5):
          img = np.transpose(ambig_and_wrong[i])
          fig.add_subplot(rows, columns, i + 1)
          plt.imshow(img)
        plt.show()
```

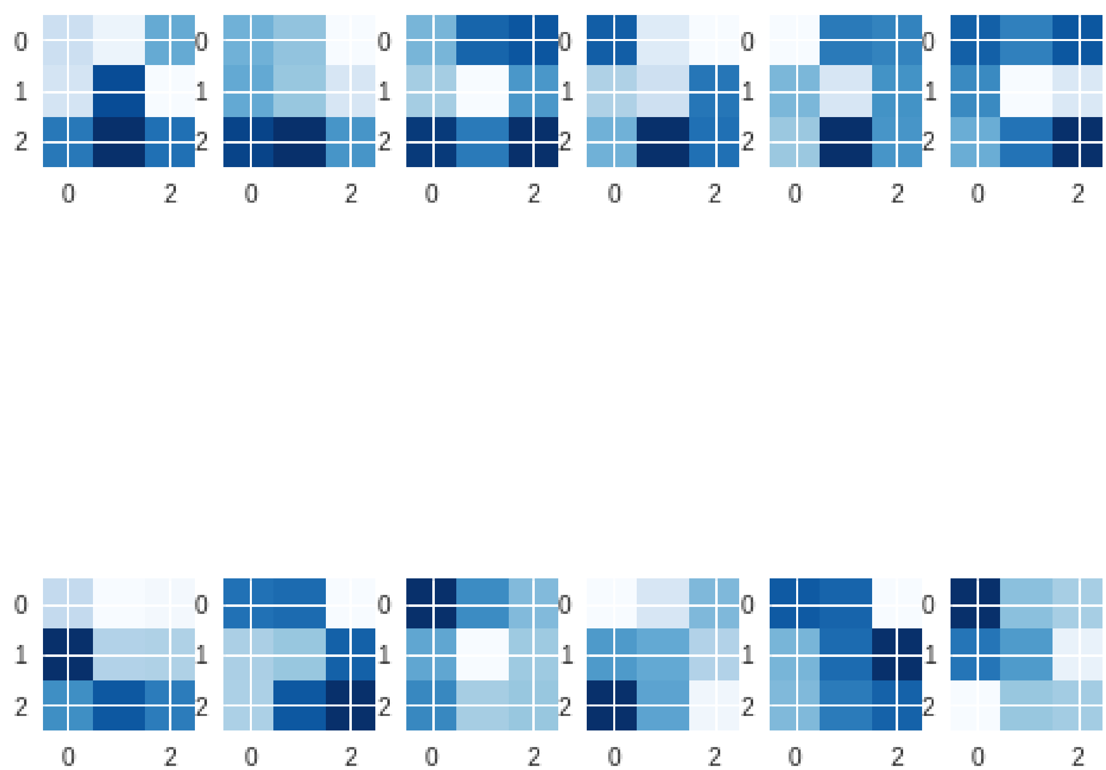


5 We can also plot the kernels too, although interpretation is more difficult

```
In [0]: # we can plot the kernels too

        def plot_kernels():
          w = 12
          h = 12
          fig=plt.figure(figsize=(8, 8))
          columns = 8
          rows = 6
          weight_cpu = model.conv2.weight.data.cpu()
          weight_np = weight_cpu.detach().numpy()
          for i in range(40):
            img = weight_np[i,0,:,:]
            fig.add_subplot(rows, columns, i + 1)
            plt.imshow(img, cmap = 'Blues')
          plt.show()

        plot_kernels()
```

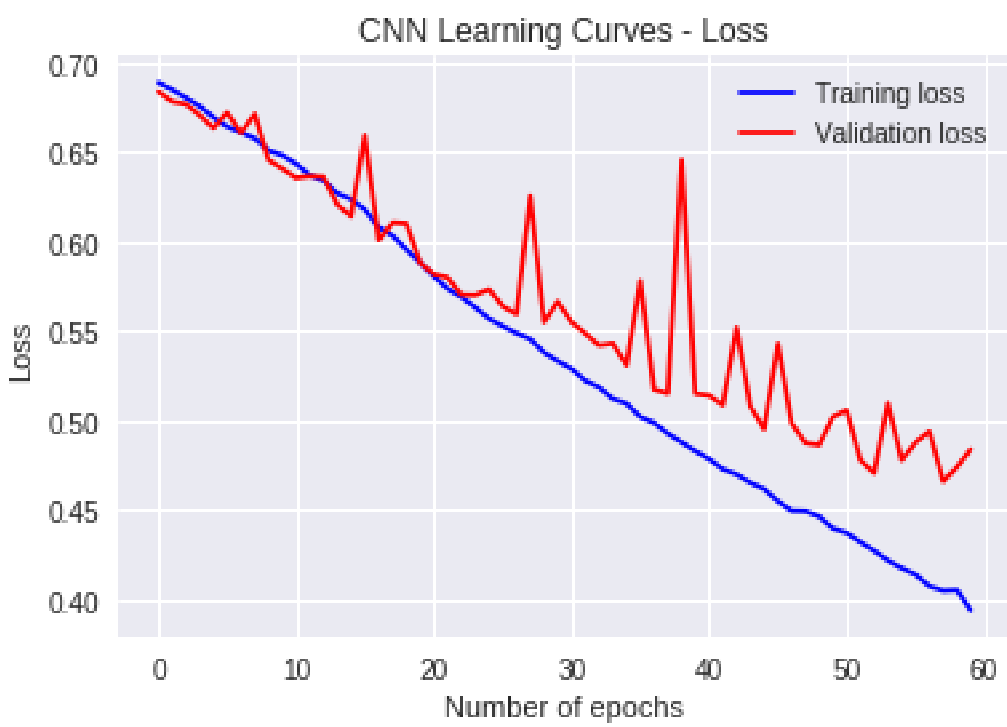
```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-87-e2570a5be5fb> in <module>()
         14   plt.show()
         15
    ---> 16 plot_kernels()


        <ipython-input-87-e2570a5be5fb> in plot_kernels()
         10   for i in range(40):
         11     img = weight_np[i,0,:,:]
    ---> 12     fig.add_subplot(rows, columns, i + 1)
```

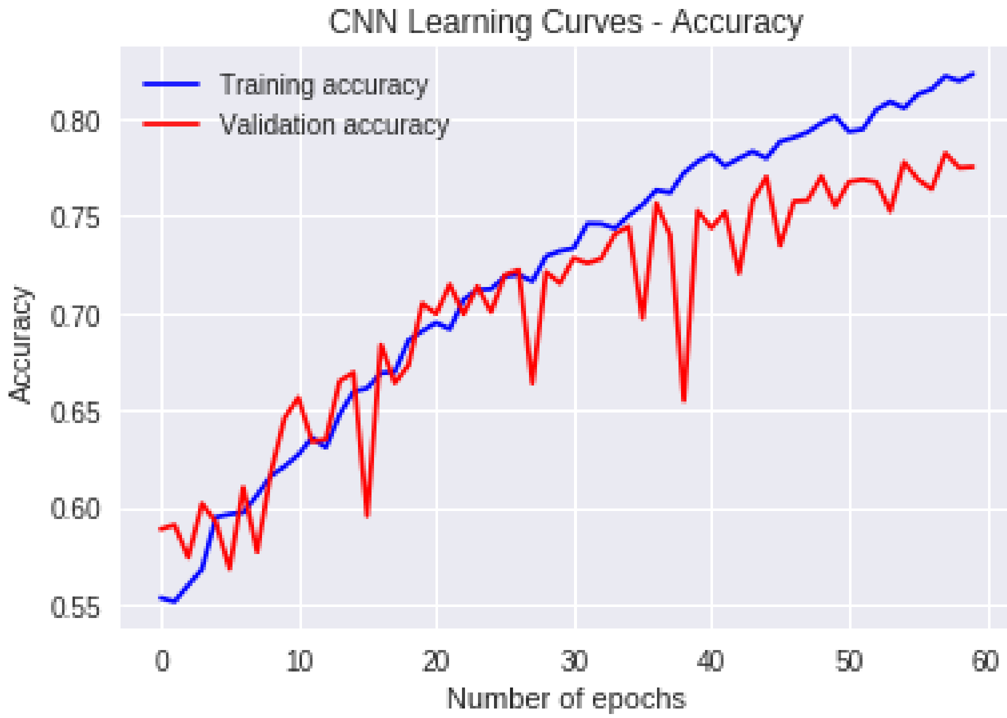
```
     13     plt.imshow(img, cmap = 'Blues')
     14   plt.show()


    /usr/local/lib/python3.6/dist-packages/matplotlib/figure.py in add_subplot(self, *args
   1365                         self._axstack.remove(ax)
   1366
-> 1367                 a = subplot_class_factory(projection_class)(self, *args, **kwargs)
   1368             self._axstack.add(key, a)
   1369             self.sca(a)


    /usr/local/lib/python3.6/dist-packages/matplotlib/axes/_subplots.py in __init__(self, :
     58                         raise ValueError(
     59                             ("num must be 1 <= num <= {maxn}, not {num}"
---> 60                             ).format(maxn=rows*cols, num=num))
     61                     self._subplotspec = GridSpec(
     62                             rows, cols, figure=self.figure)[int(num) - 1]


    ValueError: num must be 1 <= num <= 12, not 13
```



6 Finally, we can plot the learning curves for our training and validation run

In [0]:
```
%matplotlib inline
import matplotlib.pyplot as plt

plt.title('CNN Learning Curves - Loss')
plt.plot(range(num_epochs), train_losses, color='blue', label='Training loss')
plt.plot(range(num_epochs), valid_losses, color='red', label='Validation loss')
plt.xlabel('Number of epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.title('CNN Learning Curves - Accuracy')
plt.plot(range(num_epochs), train_accuracies, color='blue', label='Training accuracy')
plt.plot(range(num_epochs), valid_accuracies, color='red', label='Validation accuracy'
plt.xlabel('Number of epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```





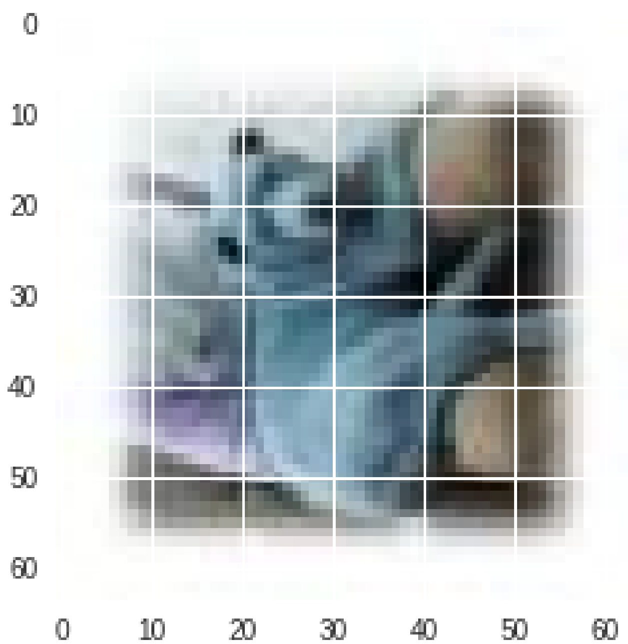
6.1 CNN Testing

Now we evaluate the test performance of our CNN.

```
In [0]: # define the transforms and dataloaders for the test set
        transform = transforms.Compose([
            transforms.ToPILImage(),
            transforms.ToTensor()
        ])

        test_sampler = SequentialSampler(np.arange(len(testIDS)))

        test_loader = torch.utils.data.DataLoader(test_dataset,
                                                  batch_size=batch_size,
                                                  sampler=test_sampler)

        # Test the model
        model.eval()
        with torch.no_grad():
          count = 1
          for images in test_loader:

            just_image = images[0,:,:,:]
            a = np.transpose(just_image)
```

```
        plt.imshow(a)


        images = images.to(device)
        outputs = model(images)

#       convert and stack outputs
        outputs_cpu = outputs.cpu()
        temp = outputs_cpu.detach().numpy()
        if count == 1:
          test_out = temp
        else:
          test_out = np.vstack((test_out, temp))

        count += 1
    print('Beginning testing')
    print(len(test_out))
```

```
Beginning testing
4999
```



In [0]:
```
# pickle and save results to submit for kaggle

import pandas as pd
import numpy as np
```

```python
import pickle
import os
import csv

currPath = os.getcwd()
prediction = []
# convert softmax to cat/dog
for i in range(len(test_out)):
  if test_out[i, 0] > test_out[i, 1]:
    prediction.append('Cat')
  else:
    prediction.append('Dog')




# build csv and submit
with open(PATH +'test_predicted.csv', 'w') as csvfile:
    # defined by the sample csv
    fieldnames = ['Id', 'label']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    for i in range(len(prediction)):
      writer.writerow({'Id': testIDS[i], 'label':prediction[i]})
```