frqs

May 18, 2021

[1]:
```
#pytorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import time
#MNIST data
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
#DogSet
from data.dogs import DogsDataset
```

[2]:
```
#local functions
from data.load_data import load_mnist_data
from data.my_dataset import MyDataset
from src.run_model import run_model
from src.run_model import _test
#Classifiers
from src.models import Digit_Classifier
from src.models import Dog_Classifier_FC
from src.models import Dog_Classifier_Conv
#plotting
import matplotlib.pyplot as plt
#reduce
from functools import reduce
```

0.1 Training on MNIST

[3]:
```
#hoppers
training_sizes = [500, 1000, 1500, 2000]
losses_tr = dict((key, None) for key in training_sizes)
models = dict((key, None) for key in training_sizes)
#in seconds
time_elapsed = dict((key, None) for key in training_sizes)
#in percentage
accuracies_tr = dict((key, None) for key in training_sizes)
```

[4]:
```
params = {'batch_size': 10,
          'shuffle': True
         }
#models[idx], losses[idx], accuracies[idx]
```

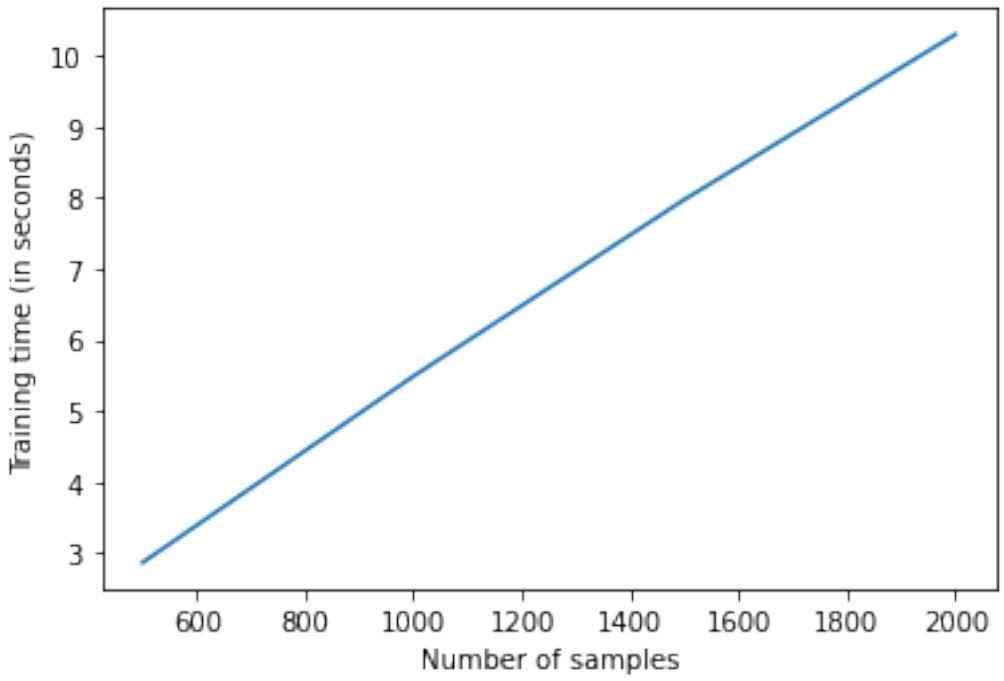
[5]:
```
for idx, size in enumerate(training_sizes):
    # data prep
    tr_f, _, tr_t, _ = load_mnist_data(10, 1, size/10)
    trainingset = MyDataset(tr_f, tr_t)
    #build model outside
    models[size] = Digit_Classifier()
    #params
    params = {
        "model": models[size],
        "running_mode": "train",
        "train_set": trainingset,
        "batch_size": 10,
        "learning_rate": 0.01,
        "n_epochs": 100
    }
    start = time.time()
    models[size], losses_tr[size], accuracies_tr[size] = run_model(**params)
    end = time.time()
    time_elapsed[size] = (end-start)
```

```
Number of epochs ran: 100
Number of epochs ran: 100
Number of epochs ran: 100
Number of epochs ran: 100
```

0.1.1 Q1

[6]:
```
plt.plot(training_sizes, list(time_elapsed.values()))
plt.xlabel("Number of samples")
plt.ylabel("Training time (in seconds)")
```

[6]:
```
Text(0, 0.5, 'Training time (in seconds)')
```



0.1.2 Q2

Training time increases linearly with the number of training examples. To train the full MNIST training set of 60K images, it would take (60K/2K)*10seconds/60s/min = 5 min

0.1.3 Q3

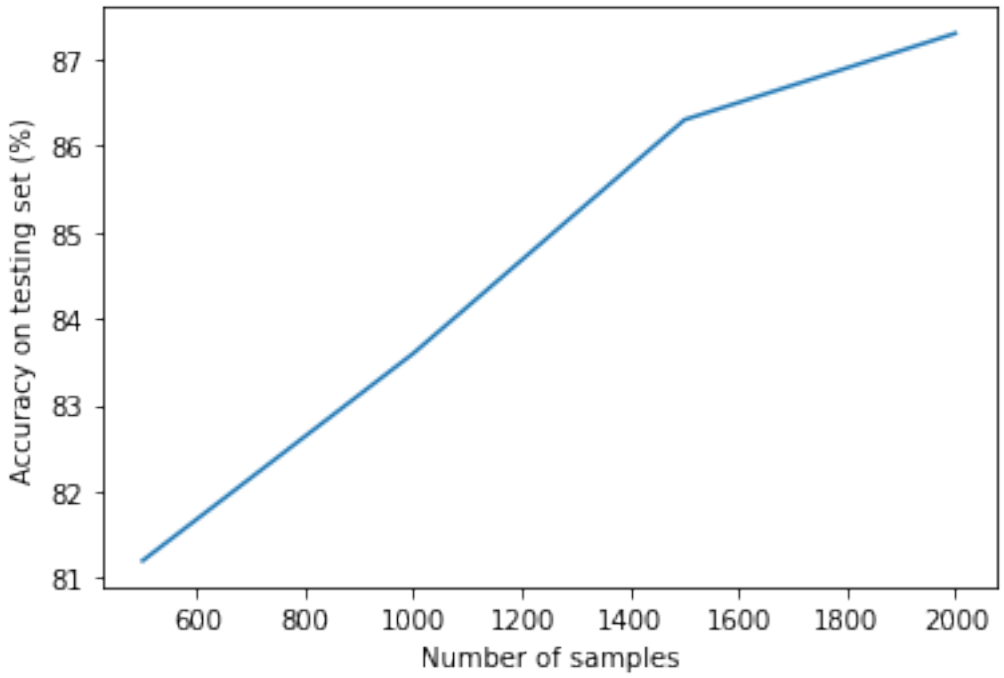
[7]:
```
#testing set + DataLoader
_, te_f, _, te_t = load_mnist_data(10, 0, 100)
testingset = MyDataset(te_f, te_t)
testing_generator = DataLoader(testingset, 1000)
```

[8]:
```
#hoppers
losses_te = dict((key, None) for key in training_sizes)
accuracies_te = dict((key, None) for key in training_sizes)
```

[9]:
```
for idx, size in enumerate(training_sizes):
    losses_te[size], accuracies_te[size] = _test(models[size],␣
 ↪testing_generator)
```

[10]:
```
#plotting
plt.plot(training_sizes, list(accuracies_te.values()))
plt.xlabel("Number of samples")
plt.ylabel("Accuracy on testing set (%)")
```

[10]: `Text(0, 0.5, 'Accuracy on testing set (%)')`



0.1.4 Q4

The training accuracy pretty much also increases linearly wrt number of training examples. Though This would mean that very soon the model will have a very high accuracy and probably this is overfitting.

0.2 Exploring DogSet

0.2.1 Q5

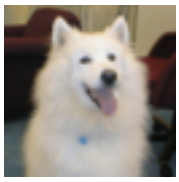
7665 in train, 555 in test, 2000 in valid partition, 3 color channels, and 10 dog breeds.

0.2.2 Q6

I think Collies (122) are hard for the classifier to get right. It can be of many different colors and its facial features may be mistaken with other mid-large sized dogs such as saint bernard (143). The photos also have the dogs'faces at different angles which may not be captured well with our current approach.

Picture 1 

Picture 2 

Picture 3 

0.3 Training a model on DogSet

0.3.1 Q7

Number of weights: fully connected so we have 64 * 64 * 3 * 128 weights and 64 * 64 * 3 bias terms from input to 1st hidden layer, 128 * 64 weights and 128 bias terms from 1st hidden to 2nd hidden, and finally 64 * 10 weights and 64 bias terms from 2nd hidden to linear output layer for a grand total of 1,594,176 parameters.

[11]:
```
dogs = DogsDataset('data/DogSet')
dog_trainingX, dog_trainingY = dogs.get_train_examples()
dog_validX, dog_validY = dogs.get_validation_examples()
dog_testX, dog_testY = dogs.get_test_examples()

dogtrainset = MyDataset(dog_trainingX, dog_trainingY)
dogvalidset = MyDataset(dog_validX, dog_validY)
dogtestset = MyDataset(dog_testX, dog_testY)
```

```
loading train…
loading valid…
loading test…
Done!
```

0.3.2 Q8

[12]:
```
#init
dogfc = Dog_Classifier_FC()
#params
dogfc_params = {
    "model": dogfc,
    "running_mode": "train",
    "train_set": dogtrainset,
    "valid_set": dogvalidset,
    "batch_size": 10,
    "learning_rate": 1e-5,
    "n_epochs": 100,
    "stop_thr": 1e-4,
```

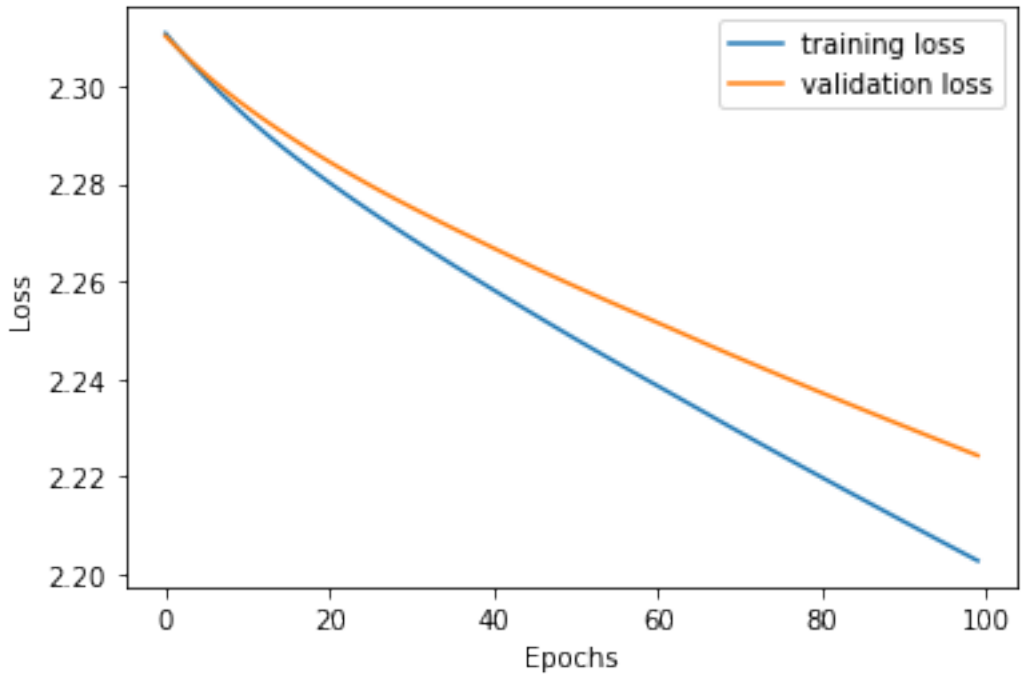
```
    "shuffle": "True"
}
```

[13]:
```
dogfc, dogfc_loss, dogfc_acc = run_model(**dogfc_params)
```

```
Number of epochs ran: 100
```

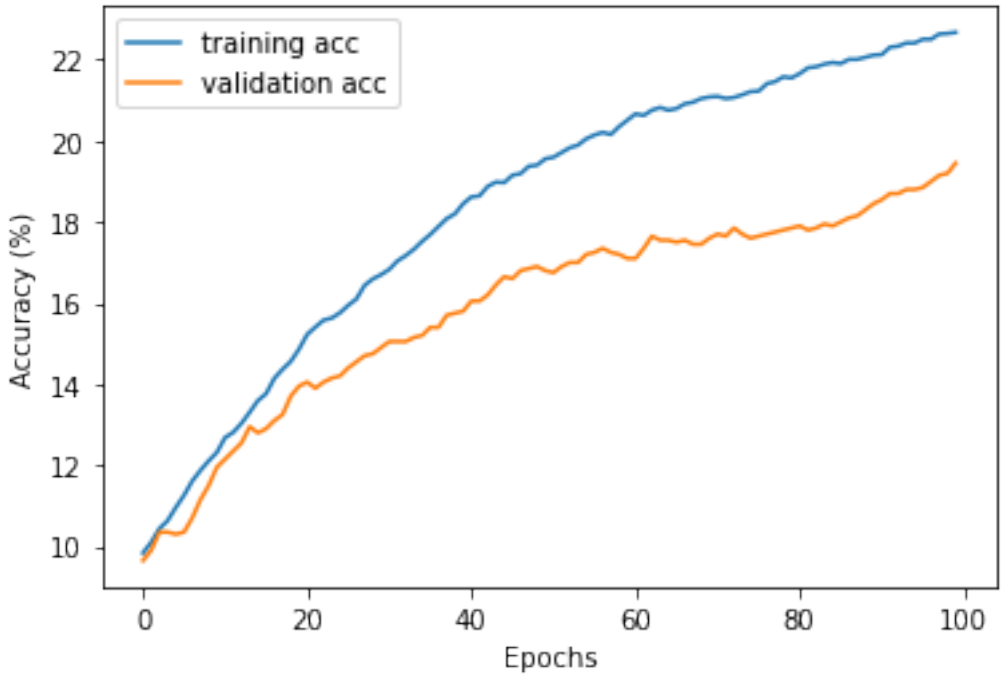
[14]:
```
#losses
plt.plot(range(100),dogfc_loss["train"], label='training loss')
plt.plot(range(100),dogfc_loss["valid"], label='validation loss')
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Loss")
```

[14]:
```
Text(0, 0.5, 'Loss')
```



[15]:
```
#accuracies
plt.plot(range(100),dogfc_acc["train"], label='training acc')
plt.plot(range(100),dogfc_acc["valid"], label='validation acc')
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Accuracy (%)")
```

[15]:
```
Text(0, 0.5, 'Accuracy (%)')
```



[16]:
```
# Accuracy of model on testing set
#test params
dogfc_test_params = {
    "model": dogfc,
    "running_mode": "test",
    "test_set": dogtestset,
    "batch_size": 10,
    "shuffle": "True"
}
```

[17]:
```
testing_loss, testing_acc = run_model(**dogfc_test_params)
```

[18]:
```
#Testing accuracy
print("Testing accuracy {0} %".format(round(testing_acc, 3)))
```

```
Testing accuracy 17.297 %
```

0.3.3 Q9

With these parameters, the network stopped learning only when max epoch was reached. Similarly, we see that the accuracy of the model does not improve much in the span of 100 epochs, accuracy is around 20%. This may be due to the fact that fully connected networks are not suitable for learning image classification.

0.4 Convolutional layers

0.4.1 Q10

Output: (16, Cout, Hout, Wout) [(W−K+2P)/S]+1

= 17 x 23

[19]:
```
#sanity check
in_channels = 3
out_channels = 3
kernel_size = (8,4)
stride = (2,2)
padding= (4, 8)
dilation = (1,1)
conv2d1 = nn.Conv2d(in_channels, out_channels,
                    kernel_size, stride=stride, padding=padding,
                     dilation=dilation
                   )
```

[20]:
```
inputs = torch.Tensor(0.1 * np.ones((16,3,32,32)))
inputs.shape
```

[20]:
```
torch.Size([16, 3, 32, 32])
```

[21]:
```
a = conv2d1(inputs)
a.shape
```

[21]:
```
torch.Size([16, 3, 17, 23])
```

0.4.2 Q11

[22]:
```
#init
dogconv = Dog_Classifier_Conv(kernel_size = [(5,5),(5,5)],
                              stride = [(1,1),(1,1)]
                             )
```

[23]:
```
#even better attempt
pytorch_total_params = sum(p.numel() for p in dogconv.parameters() if p.
 ↪requires_grad)
```

[24]:
```
#my attempt
def num_params(model):
    print(reduce((lambda x, y: x+y),
            [reduce((lambda x, y: x*y), layer.shape) for layer in model.
 ↪parameters() if layer.requires_grad]))
```

[25]:
```
# Num of params:
num_params(dogconv)
```

68138

0.4.3 Q12

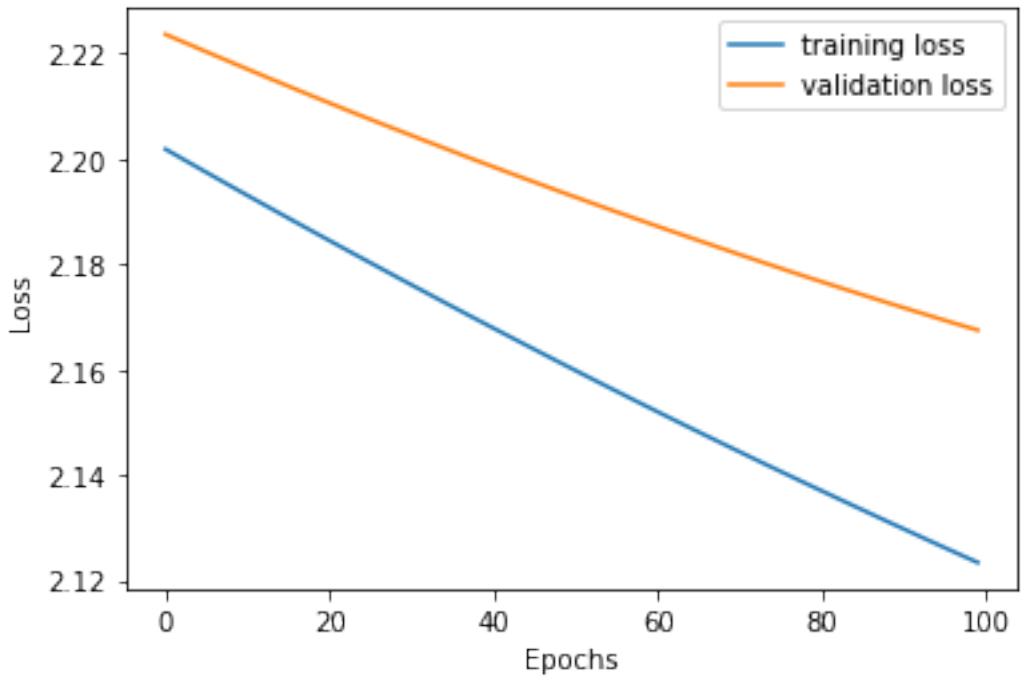
[26]:
```
#params
dogconv_params = {
    "model": dogconv,
    "running_mode": "train",
    "train_set": dogtrainset,
    "valid_set": dogvalidset,
    "batch_size": 10,
    "learning_rate": 1e-5,
    "n_epochs": 100,
    "stop_thr": 1e-4,
    "shuffle": "True"
}
```

[27]:
```
dogconv, dogconv_loss, dogconv_acc = run_model(**dogfc_params)
```

```
Number of epochs ran: 100
```

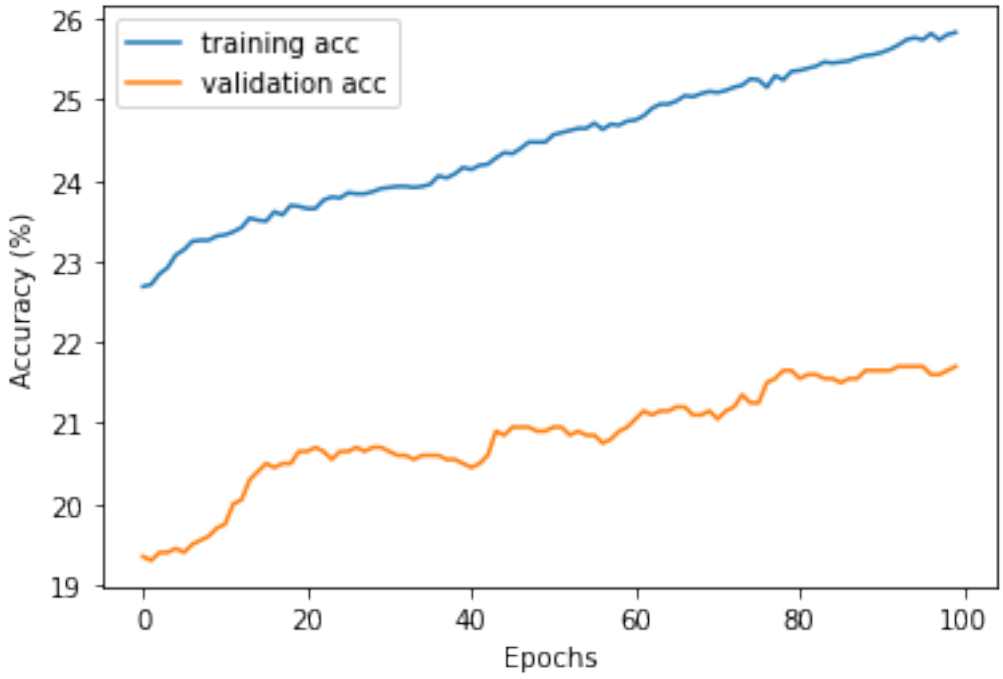
[28]:
```
#losses
plt.plot(range(100),dogconv_loss["train"], label='training loss')
plt.plot(range(100),dogconv_loss["valid"], label='validation loss')
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Loss")
```

[28]:
```
Text(0, 0.5, 'Loss')
```



[29]:
```
#accuracies
plt.plot(range(100),dogconv_acc["train"], label='training acc')
plt.plot(range(100),dogconv_acc["valid"], label='validation acc')
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Accuracy (%)")
```

[29]:
```
Text(0, 0.5, 'Accuracy (%)')
```



[30]:
```
# Accuracy of model on testing set
#test params
dogconv_test_params = {
    "model": dogconv,
    "running_mode": "test",
    "test_set": dogtestset,
    "batch_size": 10,
    "shuffle": "True"
}
```

[31]:
```
testing_loss, testing_acc = run_model(**dogconv_test_params)
```

[32]:
```
#Testing accuracy
print("Testing accuracy {0} %".format(round(testing_acc, 3)))
```

```
Testing accuracy 20.36 %
```

0.5 Digging more deeply into conv nets:

(in the CNN-Free-Response Doc)

[33]:
```
from src.models import Large_Dog_Classifier
```

[34]:
```
# Set up
#create instance
```

```
model = Large_Dog_Classifier()
model_weights_path = 'experiments/large-CNN'
# load in the DogSet dataset
model.load_state_dict(torch.load(model_weights_path))
# Print out the layers of the network
model.eval()
```

[34]:
```
Large_Dog_Classifier(
  (conv1): Conv2d(3, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(4, 6, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(6, 8, kernel_size=(3, 3), stride=(1, 1))
  (conv4): Conv2d(8, 10, kernel_size=(3, 3), stride=(1, 1))
  (conv5): Conv2d(10, 12, kernel_size=(3, 3), stride=(1, 1))
  (conv6): Conv2d(12, 14, kernel_size=(3, 3), stride=(1, 1))
  (conv7): Conv2d(14, 16, kernel_size=(3, 3), stride=(2, 2))
  (fc1): Linear(in_features=11664, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=10, bias=True)
)
```

[35]:
```
#loading in images
dogs = DogsDataset('data/DogSet')
dog_trainingX, dog_trainingY = dogs.get_train_examples()
dog_validX, dog_validY = dogs.get_validation_examples()
dog_testX, dog_testY = dogs.get_test_examples()

dogtrainset = MyDataset(dog_trainingX, dog_trainingY)
dogvalidset = MyDataset(dog_validX, dog_validY)
dogtestset = MyDataset(dog_testX, dog_testY)
```

```
loading train…
loading valid…
loading test…
Done!
```

[36]:
```
# grab an image from the testing set, it doesn't have to be random.
# if you're having trouble finding a good image we found 86,
img_num = np.random.randint(555)
filter_image = dog_testX[img_num]

# we need to put the image into a tensor since the network expects input to␣
 ↪come in batches
# our batch size will be 1
filter_image = torch.tensor([filter_image])

# function to plot our image
def imshow(img):
    npimg = img.numpy()
```

```
    plt.imshow(npimg)
    plt.axis('off')
    plt.show()
filter_image = filter_image/2 +.5

# display the image, students will write this
imshow(filter_image[0])
print(filter_image.shape)
```

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

```
torch.Size([1, 64, 64, 3])
```

[37]:
```
# permute the image just as you did in the homework
filter_image = filter_image.permute(0, 3, 1, 2)



# pass the image through the network and save the output to a variable
filter1 = model.conv1(filter_image)  # what img looks like after 1st layer␣
 ↪without applying relu yet
filter2 = model.conv2(F.relu(filter1)) # apply relu to img, pass to 2nd layer
filter3 = model.conv3(F.relu(filter2)) # and so on
filter4 = model.conv4(F.relu(filter3))
filter5 = model.conv5(F.relu(filter4))
filter6 = model.conv6(F.relu(filter5))
```

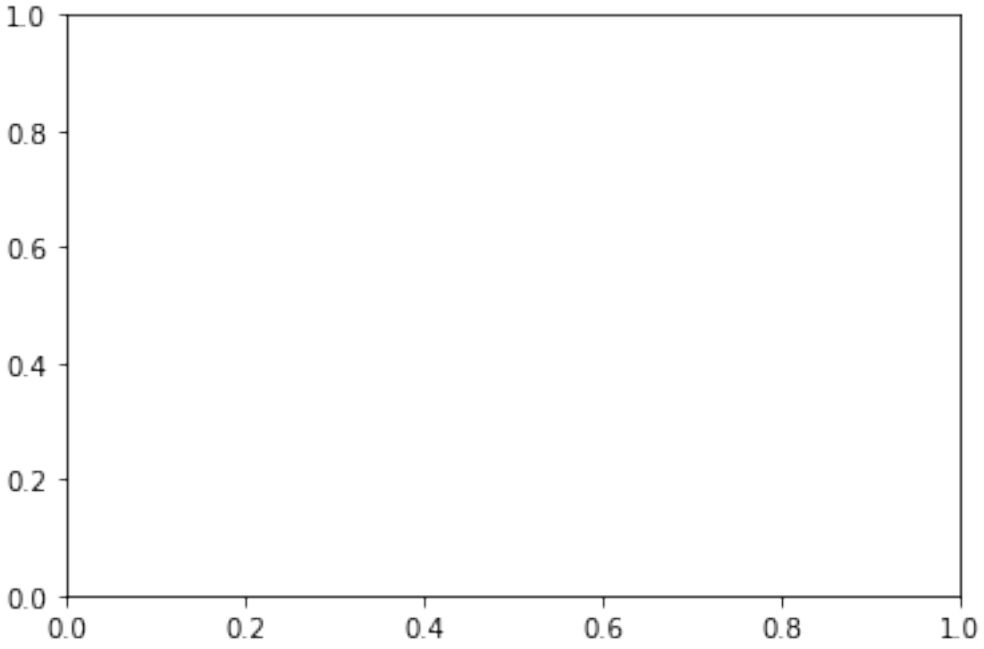
```
filter7 = model.conv7(F.relu(filter6))
#fcs
fc1 = F.relu(model.fc1(F.relu(filter7).view(-1, 11664)))
fc2 = model.fc2(fc1)


#we need to detach the gradient variable and convert the tensors to numpy arrays
filter1 = filter1.detach().numpy()
filter2 = filter2.detach().numpy()
filter3 = filter3.detach().numpy()
filter4 = filter4.detach().numpy()
filter5 = filter5.detach().numpy()
filter6 = filter6.detach().numpy()
filter7 = filter7.detach().numpy()
#fcs
fc1 = fc1.detach().numpy()
fc2 = fc2.detach().numpy()

filters = [filter1, filter2, filter3, filter4, filter5, filter6, filter7]
```

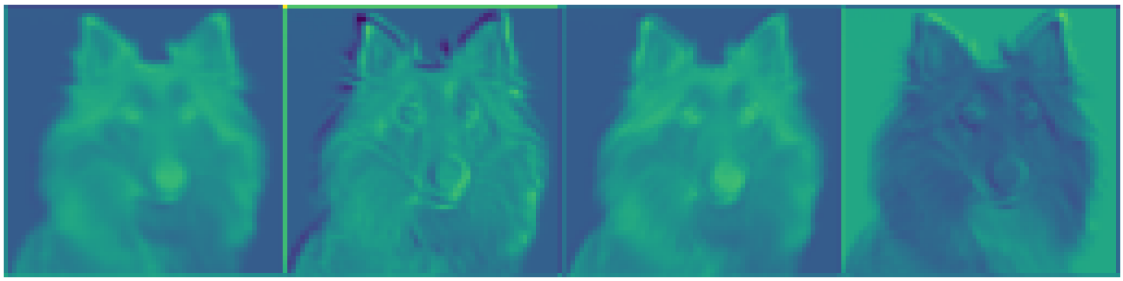
[38]:
```
def graph_filters(filters):
    """
        graph_filter - graphs a list of images which have been taken out at␣
↪various stages of a CNN

        args:
            filters (list) - list containg the output of a convolutional layer␣
↪in the network

    """
    for filter_mat in filters:
        channels = filter_mat.shape[1]
        display_grid = np.zeros((filter_mat.shape[3], channels * filter_mat.
↪shape[3]))
        print(display_grid.shape)
        for i in range(channels):
            x = filter_mat[0, i, :, :]
            x -= x.mean()
            x /= x.std()
            x *= 64
            x += 128
            display_grid[:, i * filter_mat.shape[3] : (i + 1) * filter_mat.
↪shape[3]] = x

        scale = 40. / channels
        plt.grid(False)
        plt.figure(figsize=(scale * channels, scale))
```

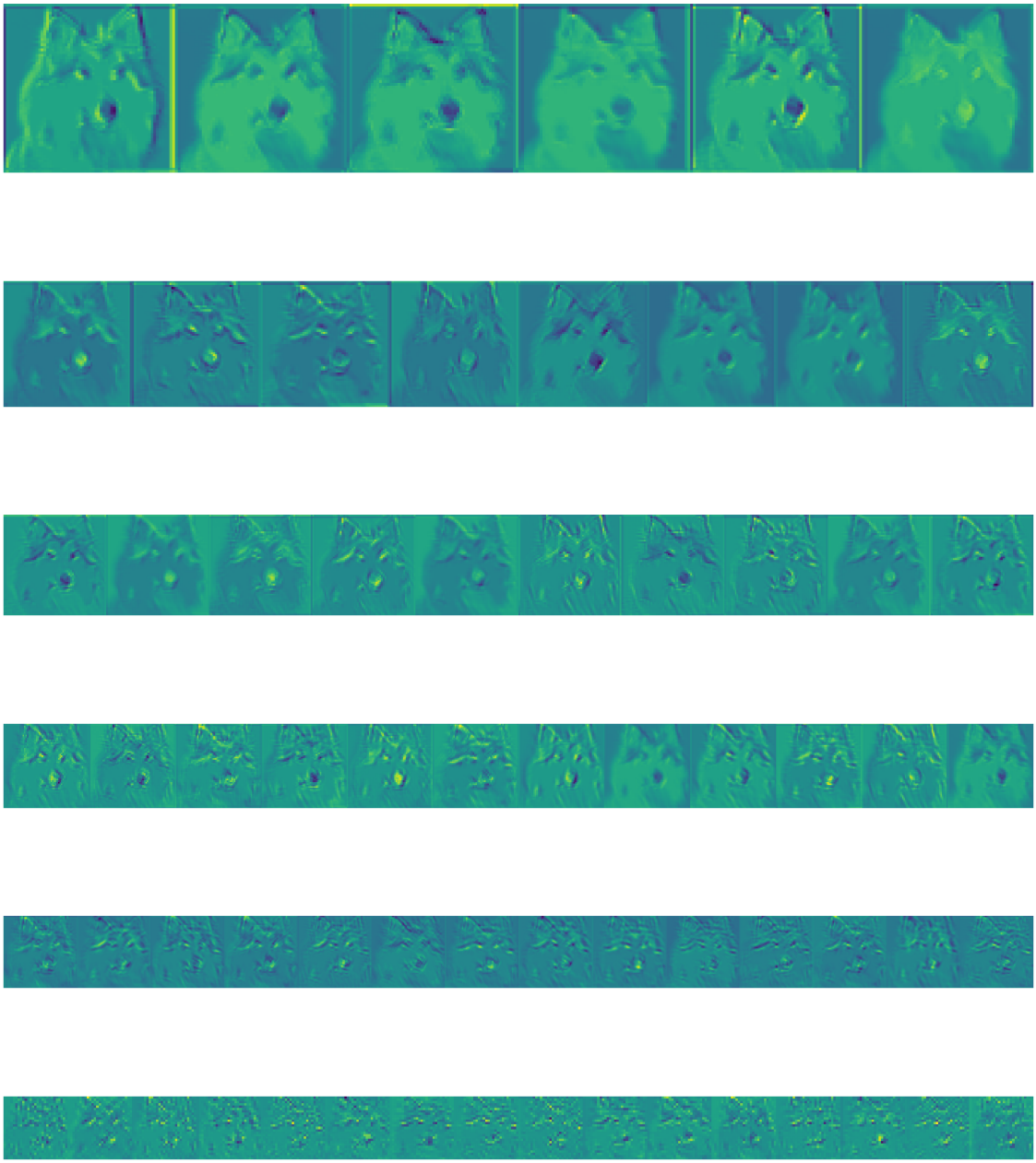
```
        plt.axis('off')
        plt.imshow(display_grid, aspect='auto', cmap='viridis')
```

[39]:
```
# use this function to view the iamges as they are passed through the CNN
graph_filters(filters)
```

```
(64, 256)
(64, 384)
(62, 496)
(60, 600)
(58, 696)
(56, 784)
(27, 432)
```







0.5.1 Q 15

(a) Each of the seven graphs represents what the outputting vector is like after the respective convolution layer. E.g. 3rd graph = the output of the image after the 3rd convolution layer. The images are repeated several times because as we can see when we print out the shape after each convolution layer, the 2nd dimension element increases. Meaning, the channels are increased from the original 3->4->6->8 and so on throughout the layers.

(b) I think the light regions represent the areas in the image that is sensitive to some filter that's being applied. Since lighter colors means higher values in the vector, it means applying some

filter on the vector results in these specific pixel values to be returned. I think it's part of feature identification/selection throughout the layers.

(c)

[40]:
```
# output dims of each CNN layer
print(f'filter1: {filter1.shape}')
print(f'filter2: {filter2.shape}')
print(f'filter3: {filter3.shape}')
print(f'filter4: {filter4.shape}')
print(f'filter5: {filter5.shape}')
print(f'filter6: {filter6.shape}')
print(f'filter7: {filter7.shape}')
print(f'fc1: {fc1.shape}')
print(f'fc2: {fc2.shape}')
```

```
filter1: (1, 4, 64, 64)
filter2: (1, 6, 64, 64)
filter3: (1, 8, 62, 62)
filter4: (1, 10, 60, 60)
filter5: (1, 12, 58, 58)
filter6: (1, 14, 56, 56)
filter7: (1, 16, 27, 27)
fc1: (1, 1024)
fc2: (1, 10)
```

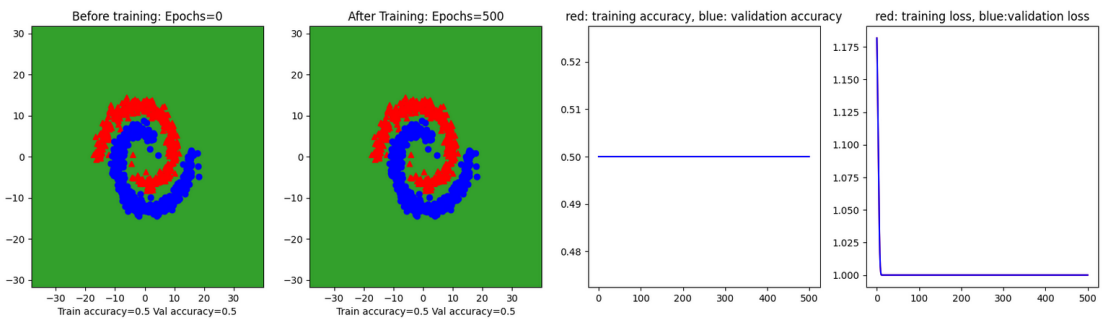
0.6 Thinking about deep models

0.6.1 Q 16

More layers and more nodes in each layer helped but was not the defining factor of whether or not the model improved its performance or not. It really comes down to the application. For instance, I made a network with multiple layers and multiple nodes and not so many layers with multiple nodes and not so many layers and not so many nodes. The most important aspect for me was whether or not I used the tanh activation function at the last output layer. I think this is important because tanh goes from [-1, 1] which is what the actual targets are whereas relu does not go < 0 nor does sigmoid. With the same structured model, if I used the tanh in the last layer the results were great.

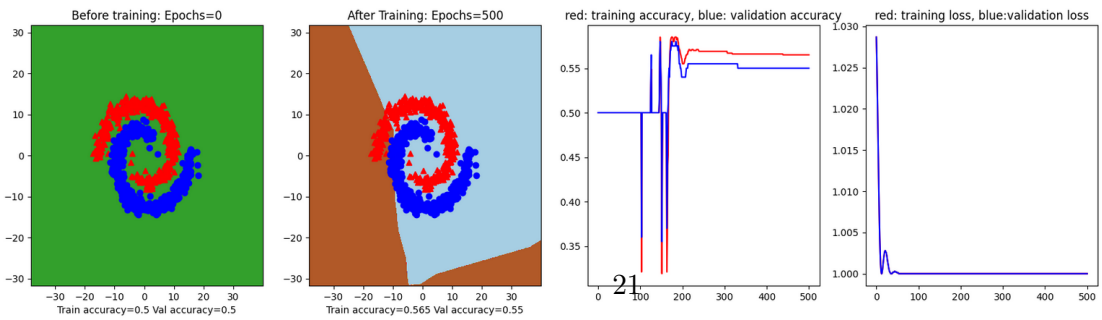
0.6.2 Many layers, many nodes, all relu (#1)

```
def __init__(self):
  super(Net2, self).__init__() # has to be here
  self.fc1 = nn.Linear(2, 4)
  self.fc2 = nn.Linear(4, 6)
  self.fc3 = nn.Linear(6, 8)
  self.fc4 = nn.Linear(8, 10)
  self.fc5 = nn.Linear(10, 12)
  self.fc6 = nn.Linear(12, 10)
  self.fc7 = nn.Linear(10, 8)
  self.fc8 = nn.Linear(8, 6)
  self.fc9 = nn.Linear(6, 4)
  self.fc10 = nn.Linear(4, 2)
  self.fc11 = nn.Linear(2, 1)

def forward(self, inputs):
  #inputs = inputs.view(inputs.shape[0], -1)
  inputs = F.relu(self.fc1(inputs))
  inputs = F.relu(self.fc2(inputs))
  inputs = F.relu(self.fc3(inputs))
  inputs = F.relu(self.fc4(inputs))
  inputs = F.relu(self.fc5(inputs))
  inputs = F.relu(self.fc6(inputs))
  inputs = F.relu(self.fc7(inputs))
  inputs = F.relu(self.fc8(inputs))
  inputs = F.relu(self.fc9(inputs))
  inputs = F.relu(self.fc10(inputs))
  inputs = F.relu(self.fc11(inputs))
  #inputs = F.tanh(self.fc3(inputs))

  return inputs
```



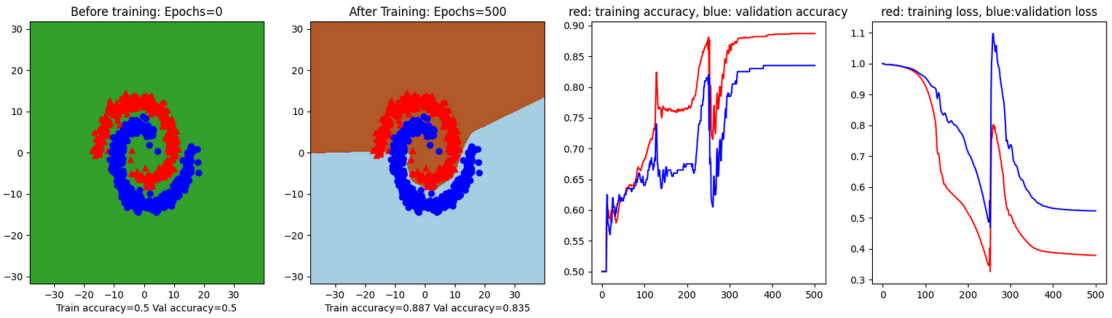
0.6.3 Many layers, many nodes, last tanh (#2)

```
def __init__(self):
  super(Net2, self).__init__() # has to be her
  self.fc1 = nn.Linear(2, 4)
  self.fc2 = nn.Linear(4, 6)
  self.fc3 = nn.Linear(6, 8)
  self.fc4 = nn.Linear(8, 10)
  self.fc5 = nn.Linear(10, 12)
  self.fc6 = nn.Linear(12, 10)
  self.fc7 = nn.Linear(10, 8)
  self.fc8 = nn.Linear(8, 6)
  self.fc9 = nn.Linear(6, 4)
  self.fc10 = nn.Linear(4, 2)
  self.fc11 = nn.Linear(2, 1)

def forward(self, inputs):
  #inputs = inputs.view(inputs.shape[0], -1)
  inputs = F.relu(self.fc1(inputs))
  inputs = F.relu(self.fc2(inputs))
  inputs = F.relu(self.fc3(inputs))
  inputs = F.relu(self.fc4(inputs))
  inputs = F.relu(self.fc5(inputs))
  inputs = F.relu(self.fc6(inputs))
  inputs = F.relu(self.fc7(inputs))
  inputs = F.relu(self.fc8(inputs))
  inputs = F.relu(self.fc9(inputs))
  inputs = F.relu(self.fc10(inputs))
  #inputs = F.relu(self.fc1(inputs))
  inputs = F.tanh(self.fc11(inputs))

  return inputs
```



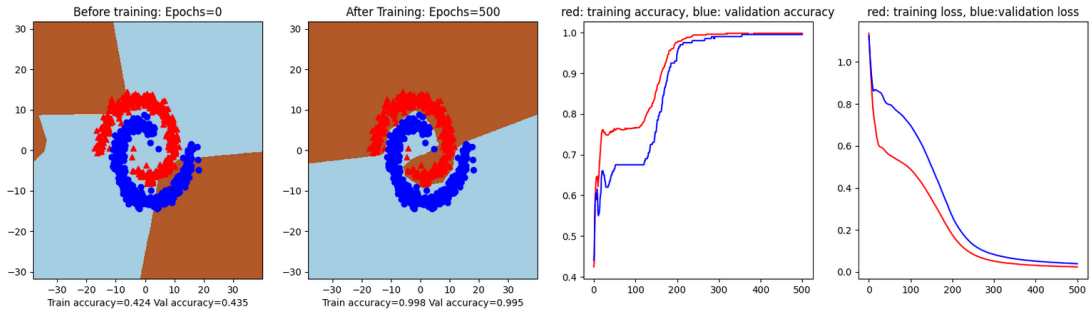
0.6.4 Less layers, many nodes, last tanh (#3)

```
def __init__(self):
  super(Net2, self).__init__() # has to be here
  self.fc1 = nn.Linear(2, 8)
  self.fc2 = nn.Linear(8, 12)
  self.fc3 = nn.Linear(12, 8)
  self.fc4 = nn.Linear(8, 2)
  self.fc5 = nn.Linear(2, 1)

def forward(self, inputs):
  #inputs = inputs.view(inputs.shape[0], -1)
  inputs = F.relu(self.fc1(inputs))
  inputs = F.relu(self.fc2(inputs))
  inputs = F.relu(self.fc3(inputs))
  inputs = F.relu(self.fc4(inputs))
  inputs = F.tanh(self.fc5(inputs))

  return inputs
```



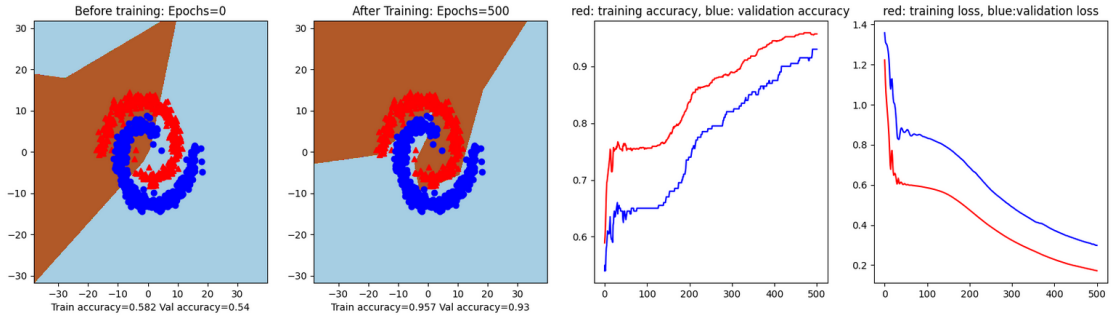
0.6.5 Even less layers, many nodes, last tanh (#4)

```
def __init__(self):
  super(Net, self).__init__() # has to be here
  self.fc1 = nn.Linear(2, 12)
  self.fc2 = nn.Linear(12, 8)
  self.fc3 = nn.Linear(8, 6)
  self.fc4 = nn.Linear(6, 1)

def forward(self, inputs):
  #inputs = inputs.view(inputs.shape[0], -1)
  inputs = F.relu(self.fc1(inputs))
  inputs = F.relu(self.fc2(inputs))
  inputs = F.relu(self.fc3(inputs))
  inputs = F.tanh(self.fc4(inputs))

  return inputs
```



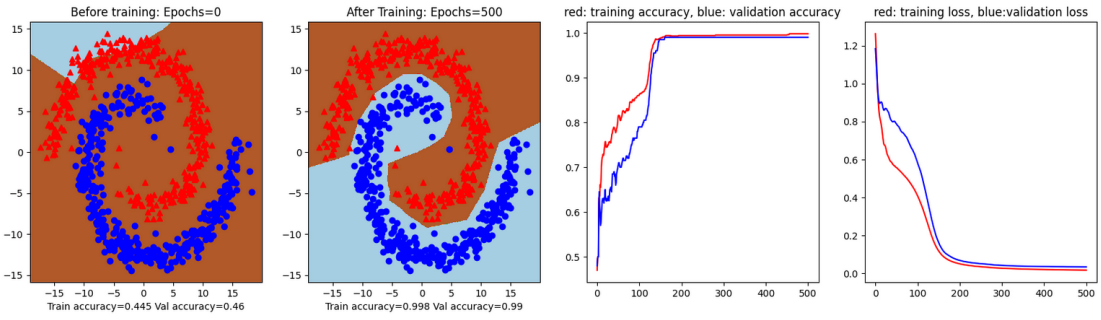
0.6.6 2 Layers, 4 nodes, last tanh (#5)

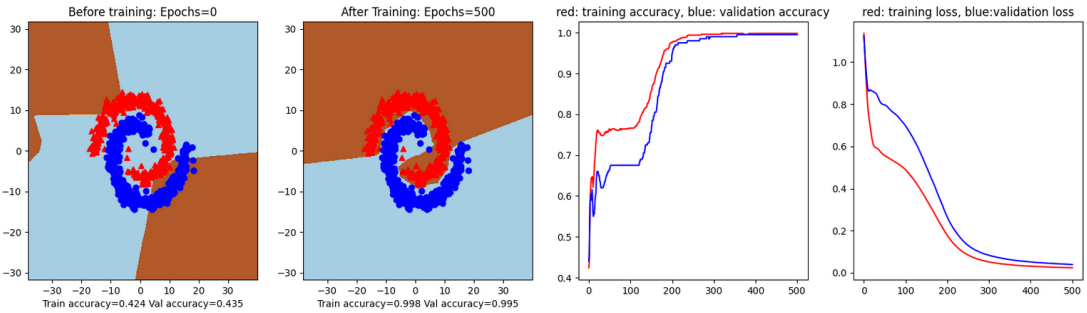
```
def __init__(self):
  super(Net, self).__init__() # has to be here
  self.fc1 = nn.Linear(2, 4)
  self.fc3 = nn.Linear(4, 1)

def forward(self, inputs):
  #inputs = inputs.view(inputs.shape[0], -1)
  inputs = F.relu(self.fc1(inputs))
  inputs = F.tanh(self.fc3(inputs))
```

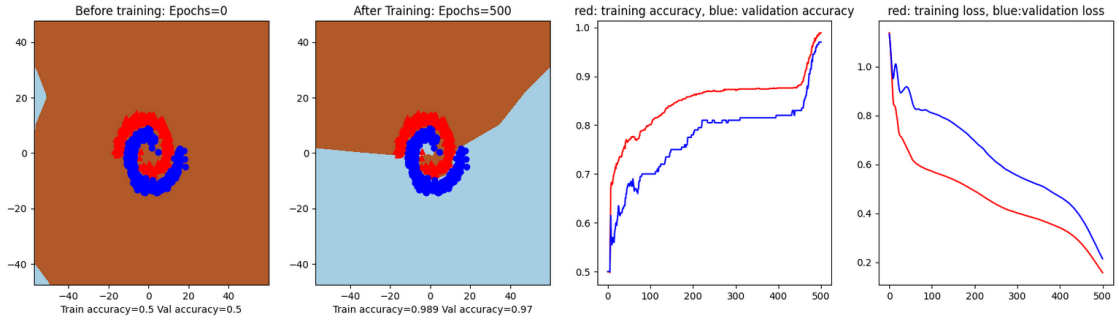


0.6.7 Q17

Besting performing model: res_mmin_tanh (#4 from above) with training acc: 99.8%, validation acc: 99.5%.







As scale increases, we zoom out more, the validation accuracy also slightly decreases. From the decision boundaries, we see that at scale=1, the boundary covers all the training points but at scale=3, the boundary covers a lot of excess space that the model "thinks" is some class. I think as we scale out the proportion of significant data decreases (since more of the background is used as input data) and hence the learning gets harder. If the spiral's pattern were to be extended to fit the canvas of a scale=3, our trained model would perform very poorly beyond the given data. This means that neural networks have a hard time extrapolating its learning beyond its given data in certain types of learning objectives such as that with a spiral-like dataset.

[ ]: