



20T2

Comp9444

Hw 1



# Comp9444

## Requirements

**conda install scikit-learn**

<https://scikit-learn.org/stable/install.html#installation-instructions>

**conda install pytorch torchvision -c pytorch**

**conda install torchvision**

<https://pytorch.org/get-started/locally/>

**conda install pillow**

<https://pillow.readthedocs.io/en/stable/installation.html>




# Comp9444

## Requirements

<https://pytorch.org/docs/stable/index.html>

1.5.1 ▼

 Search Docs

Notes [Expand]

Language Bindings

C++

Javadoc

Python API

torch

torch.nn

torch.nn.functional

torch.Tensor

Tensor Attributes

Tensor Views

Docs > PyTorch documentation

PYTORCH DOCUMENTATION

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.

Notes

- [Automatic Mixed Precision examples](#)
- [Autograd mechanics](#)
- [Broadcasting semantics](#)
- [CPU threading and TorchScript inference](#)
- [CUDA semantics](#)
- [Distributed Data Parallel](#)
- [Extending PyTorch](#)
- [Frequently Asked Questions](#)
- [Features for large-scale deployments](#)
- [Multiprocessing best practices](#)
- [Reproducibility](#)



# Comp9444

## Part 1



kuzu\_main.py

```
def main():
    # Training settings
    parser = argparse.ArgumentParser()
    parser.add_argument('--net', type=str, default='full', help='lin, full or conv')
    parser.add_argument('--lr', type=float, default=0.01, help='learning rate')
    parser.add_argument('--mom', type=float, default=0.5, help='momentum')
    parser.add_argument('--epochs', type=int, default=10, help='number of training epochs')
    parser.add_argument('--no_cuda', action='store_true', default=False, help='disables CUDA')
    args = parser.parse_args()

    use_cuda = not args.no_cuda and torch.cuda.is_available()

    device = torch.device('cuda' if use_cuda else 'cpu')

    kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}

    # Define a transform to normalize the data
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (0.5,))])

    # Fetch and load the training data
    trainset = datasets.KMNIST(root='./data', train=True, download=True, transform=transform)
    train_loader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=False)

    # Fetch and load the test data
    testset = datasets.KMNIST(root='./data', train=False, download=True, transform=transform)
    test_loader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)
```



# Comp9444

## Part 1



kuzu\_main.py

```
if args.net == 'lin':
    net = NetLin().to(device)
elif args.net == 'full':
    net = NetFull().to(device)
else:
    net = NetConv().to(device)

if list(net.parameters()):
    optimizer = optim.SGD(net.parameters(), lr=args.lr, momentum=args.mom)

    for epoch in range(1, args.epochs + 1):
        train(args, net, device, train_loader, optimizer, epoch)
        test(args, net, device, test_loader)
```



# Comp9444

## Part 1



kuzu\_main.py

```
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

# Comp9444

## Part 1



kuzu\_main.py

```
def test(args, model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    conf_matrix = np.zeros((10,10)) # initialize confusion matrix
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            # sum up batch loss
            test_loss += F.nll_loss(output, target, reduction='sum').item()
            # get the index of the max log-probability
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
            conf_matrix = conf_matrix + metrics.confusion_matrix(
                pred.cpu(), target.cpu(), labels=[0,1,2,3,4,5,6,7,8,9])
    np.set_printoptions(precision=4, suppress=True)
    print(type(conf_matrix))
    print(conf_matrix)

test_loss /= len(test_loader.dataset)

print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%) \n'.format(
    test_loss, correct, len(test_loader.dataset),
    100. * correct / len(test_loader.dataset)))
```



# Comp9444

## Part 1

**CLASS** `torch.nn.Linear(in_features, out_features, bias=True)`

[\[SOURCE\]](#)

Applies a linear transformation to the incoming data:  $y = xA^T + b$

### Parameters

- **in\_features** – size of each input sample
- **out\_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

### Shape:

- Input:  $(N, *, H_{in})$  where  $*$  means any number of additional dimensions and  $H_{in} = \text{in\_features}$
- Output:  $(N, *, H_{out})$  where all but the last dimension are the same shape as the input and  $H_{out} = \text{out\_features}$ .



kuzu.py





# Comp9444

## Part 1

**CLASS** `torch.nn.LogSoftmax(dim=None)`

[\[SOURCE\]](#) 

Applies the  $\log(\text{Softmax}(x))$  function to an n-dimensional input Tensor. The LogSoftmax formulation can be simplified as:

$$\text{LogSoftmax}(x_i) = \log \left( \frac{\exp(x_i)}{\sum_j \exp(x_j)} \right)$$

Shape:

- Input:  $(*)$  where  $*$  means, any number of additional dimensions
- Output:  $(*)$ , same shape as the input

Parameters

**dim** (*int*) – A dimension along which LogSoftmax will be computed.



kuzu.py



# Comp9444

## Part 1

**CLASS** `torch.nn.Tanh`

[\[SOURCE\]](#) 

Applies the element-wise function:

$$\text{Tanh}(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

Shape:

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input



kuzu.py



# Comp9444

## Part 1

```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,  
dilation=1, groups=1, bias=True, padding_mode='zeros')
```

[\[SOURCE\]](#)



kuzu.py

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size  $(N, C_{\text{in}}, H, W)$  and output  $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$  can be precisely described as:



# Comp9444

## Part 1

**CLASS** `torch.nn.ReLU(inplace=False)`

[\[SOURCE\]](#)

Applies the rectified linear unit function element-wise:

$$\text{ReLU}(x) = (x)^+ = \max(0, x)$$

Parameters

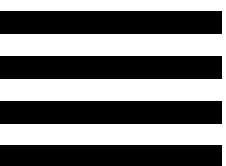
**inplace** – can optionally do the operation in-place. Default: `False`

Shape:

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input



kuzu.py



# Comp9444

## Part 1



kuzu.py

```
class NetLin(nn.Module):
    # linear function followed by log_softmax
    def __init__(self):
        super(NetLin, self).__init__()
        self.linear = nn.Linear(28*28, 10)
        self.log_softmax = nn.LogSoftmax()

    def forward(self, x):
        x_1 = x.view(x.shape[0], -1)
        x_2 = self.linear(x_1)
        x_3 = self.log_softmax(x_2)
        return x_3
```

1. [1 mark] Implement a model `NetLin` which computes a linear function of the pixels in the image, followed by log softmax. Run the code by typing:

```
python3 kuzu_main.py --net lin
```

Copy the final accuracy and confusion matrix into your report. Note that the **columns** of the confusion matrix indicate the target character, while the **rows** indicate the one chosen by the network. (0="o", 1="ki", 2="su", 3="tsu", 4="na", 5="ha", 6="ma", 7="ya", 8="re", 9="wo"). More examples of each character can be found [here](#).



# Comp9444

## Part 1



kuzu.py

```
[[771.  7.  7.  4.  59.  8.  4.  16.  10.  8.]
 [  5. 661. 59. 36.  50. 28. 23. 28. 38. 52.]
 [  9. 110. 693. 58.  79. 125. 147. 28. 97. 84.]
 [ 12.  18.  26. 760.  19.  16.  10.  12. 43.  3.]
 [ 31.  30.  25.  15. 629.  20.  25. 82.  8. 57.]
 [ 61.  23.  21.  58.  17. 726.  24.  17. 28. 30.]
 [  2.  61.  47.  13.  33.  27. 723.  53. 43. 20.]
 [ 62.  12.  37.  19.  36.  8.  19. 624.  6. 32.]
 [ 30.  26.  48.  26.  20.  31.  10.  91. 704. 38.]
 [ 17.  52.  37.  11.  58.  11.  15.  49.  23. 676.]]
```



# Comp9444

## Part 1



kuzu.py

```
class NetFull(nn.Module):
    # two fully connected tanh layers followed by log softmax
    def __init__(self):
        super(NetFull, self).__init__()
        # INSERT CODE HERE

    def forward(self, x):
        return 0 # CHANGE CODE HERE
```

2. [2 marks] Implement a fully connected 2-layer network `NetFull`, using tanh at the hidden nodes and log softmax at the output node. Run the code by typing:

```
python3 kuzu_main.py --net full
```

Try different values (multiples of 10) for the number of hidden nodes and try to determine a value that achieves high accuracy on the test set. Copy the final accuracy and confusion matrix into your report.



# Comp9444

## Part 1



kuzu.py

```
class NetConv(nn.Module):
    # two convolutional layers and one fully connected layer,
    # all using relu, followed by log_softmax
    def __init__(self):
        super(NetConv, self).__init__()
        # INSERT CODE HERE

    def forward(self, x):
        return 0 # CHANGE CODE HERE
```

3. [2 marks] Implement a convolutional network called `NetConv`, with two convolutional layers plus one fully connected layer, all using `relu` activation function, followed by the output layer. You are free to choose for yourself the number and size of the filters, metaparameter values, and whether to use max pooling or a fully convolutional architecture. Run the code by typing:

```
python3 kuzu_main.py --net conv
```

Your network should consistently achieve at least 93% accuracy on the test set after 10 training epochs. Copy the final accuracy and confusion matrix into your report.





# Comp9444

## Part 1



kuzu.py

4. [7 marks] Discuss what you have learned from this exercise, including the following points:
- a. the relative accuracy of the three models,
  - b. the confusion matrix for each model: which characters are most likely to be mistaken for which other characters, and why?
  - c. you may wish to experiment with other architectures and/or metaparameters for this dataset, and report on your results; the aim of this exercise is not only to achieve high accuracy but also to understand the effect of different choices on the final accuracy.



# Comp9444

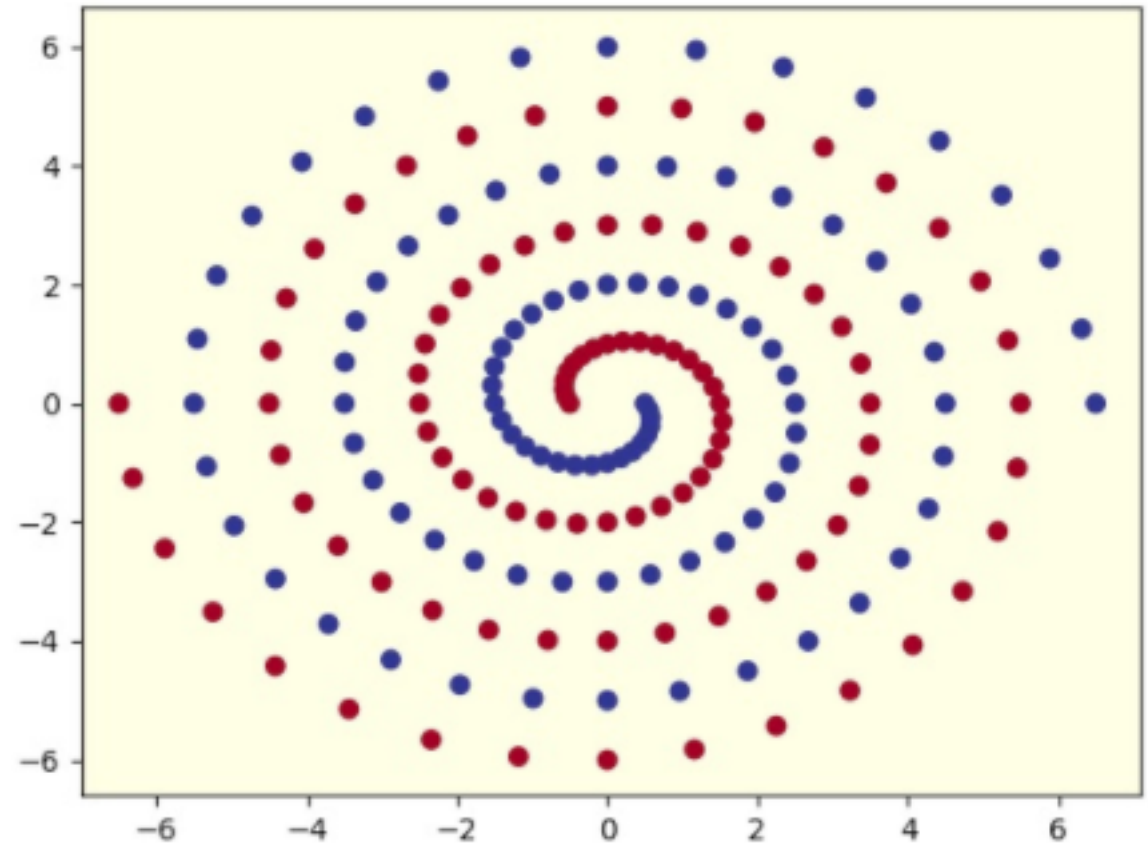
## Part 2



spiral\_main.py



spiral.py





# Comp9444

## Part 2



spiral\_main.py

```
parser = argparse.ArgumentParser()
parser.add_argument('--net', type=str, default='raw', help='polar, raw or short')
parser.add_argument('--init', type=float, default=0.1, help='initial weight size')
parser.add_argument('--hid', type=int, default='10', help='number of hidden units')
parser.add_argument('--lr', type=float, default=0.01, help='learning rate')
parser.add_argument('--epochs', type=int, default='100000', help='max training epochs')
args = parser.parse_args()

#device = 'cpu'

df = pd.read_csv('spirals.csv')

data = torch.tensor(df.values, dtype=torch.float32)

num_input = data.shape[1] - 1

full_input = data[:, 0:num_input]
full_target = data[:, num_input:num_input+1]

train_dataset = torch.utils.data.TensorDataset(full_input, full_target)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=97)
```



# Comp9444

## Part 2



spiral\_main.py

```
# create neural network
if args.net == 'polar':
    net = PolarNet(args.hid)
elif args.net == 'short':
    net = ShortNet(args.hid)
else:
    net = RawNet(args.hid)

if list(net.parameters()):
    # initialize weight values
    for m in list(net.parameters()):
        m.data.normal_(0, args.init)

    optimizer = torch.optim.Adam(net.parameters(), eps=0.000001, lr=args.lr,
                                  betas=(0.9, 0.999), weight_decay=0.0001)

    for epoch in range(1, args.epochs):
        accuracy = train(net, train_loader, optimizer)
        if epoch % 100 == 0 and accuracy == 100:
            break
```



# Comp9444

## Part 2



spiral\_main.py

```
def train(net, train_loader, optimizer):
    total=0
    correct=0
    for batch_id, (data,target) in enumerate(train_loader):
        optimizer.zero_grad()    # zero the gradients
        output = net(data)        # apply network
        loss = F.binary_cross_entropy(output,target)
        loss.backward()           # compute gradients
        optimizer.step()          # update weights
        pred = (output >= 0.5).float()
        correct += (pred == target).float().sum()
        total += target.size()[0]
        accuracy = 100*correct/total

    if epoch % 100 == 0:
        print('ep:%5d loss: %6.4f acc: %5.2f' %
              (epoch, loss.item(), accuracy))

    return accuracy
```



# Comp9444

## Part 2



spiral.py

```
torch.norm(input, p='fro', dim=None, keepdim=False, out=None, dtype=None)
```

[SOURCE]

Returns the matrix norm or vector norm of a given tensor.

### Parameters

- **input** (*Tensor*) – the input tensor
- **p** (*int, float, inf, -inf, 'fro', 'nuc', optional*) – the order of norm. Default: 'fro' The following norms can be calculated:



# Comp9444

## Part 2

`torch.atan2(input, other, out=None) → Tensor`



Element-wise arctangent of  $\text{input}_i / \text{other}_i$  with consideration of the quadrant. Returns a new tensor with the signed angles in radians between vector  $(\text{other}_i, \text{input}_i)$  and vector  $(1, 0)$ . (Note that  $\text{other}_i$ , the second parameter, is the x-coordinate, while  $\text{input}_i$ , the first parameter, is the y-coordinate.)

The shapes of `input` and `other` must be **broadcastable**.

### Parameters

- **input** (*Tensor*) – the first input tensor
- **other** (*Tensor*) – the second input tensor
- **out** (*Tensor, optional*) – the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.9041,  0.0196, -0.3108, -2.4423])
>>> torch.atan2(a, torch.randn(4))
tensor([ 0.9833,  0.0811, -1.9743, -1.4151])
```



spiral.py



# Comp9444

## Part 2

**CLASS** `torch.nn.Sigmoid`

[\[SOURCE\]](#)

Applies the element-wise function:

$$\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$$



spiral.py

Shape:

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input



# Comp9444

## Part 2



spiral.py

```
class PolarNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(PolarNet, self).__init__()
        # INSERT CODE HERE

    def forward(self, input):
        output = 0*input[:,0] # CHANGE CODE HERE
        return output
```

```
r = torch.norm(input, 2, dim=-1).unsqueeze(-1)
a = torch.atan2(input[:,0], input[:,1]).unsqueeze(-1)
x = torch.cat((r,a), -1)
```

1. [2 marks] Provide code for a Pytorch Module called `PolarNet` which operates as follows: First, the input  $(x,y)$  is converted to polar co-ordinates  $(r,a)$  with  $r=\sqrt{x^2 + y^2}$ ,  $a=\text{atan2}(y,x)$ . Next,  $(r,a)$  is fed into a fully connected neural network with one hidden layer using `tanh` activation, followed by a single output using `sigmoid` activation. The conversion to polar coordinates should be included in your `forward()` method, so that the Module performs the entire task of conversion followed by network layers.
2. [1 mark] Run the code by typing

```
python3 spiral_main.py --net polar --hid 10
```

Try to find the minimum number of hidden nodes required so that this PolarNet learns to correctly classify all of the training data within 20000 epochs, on almost all runs. The `graph_output()` method will generate a picture of the function computed by your PolarNet called `polar_out.png`, which you should include in your report.



# Comp9444

## Part 2



spiral.py

```
class RawNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(RawNet, self).__init__()
        # INSERT CODE HERE

    def forward(self, input):
        output = 0*input[:,0] # CHANGE CODE HERE
        return output
```

3. [1 mark] Provide code for a Pytorch Module called `RawNet` which operates on the raw input  $(x, y)$  without converting to polar coordinates. Your network should consist of two fully connected hidden layers with `tanh` activation, plus the output layer, with `sigmoid` activation. You should **not** use `Sequential` but should instead build the network from individual components as shown in the program `xor.py` from Exercises 5 (repeated in slide 4 of lecture slides 3b on PyTorch). The number of neurons in both hidden layers should be determined by the parameter `num_hid`.
4. [1 mark] Run the code by typing

```
python3 spiral_main.py --net raw
```

Keeping the number of hidden nodes in each layer fixed at 10, try to find a value for the size of the initial weights (`--init`) such that this `RawNet` learns to correctly classify all of the training data within 20000 epochs, on almost all runs. Include in your report the number of hidden nodes, and the values of any other metaparameters. The `graph_output()` method will generate a picture of the function computed by your `RawNet` called `raw_out.png`, which you should include in your report.



# Comp9444

## Part 2



spiral.py

```
class ShortNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(ShortNet, self).__init__()
        # INSERT CODE HERE

    def forward(self, input):
        output = 0*input[:,0] # CHANGE CODE HERE
        return output
```

5. [1 mark] Provide code for a Pytorch Module called `ShortNet` which again operates on the raw input  $(x, y)$  without converting to polar coordinates. This network should again consist of two hidden layers (with tanh activation) plus the output layer (with sigmoid activation), but this time should include short-cut connections between every pair of layers (input, hid1, hid2 and output) as depicted on slide 10 of lecture slides 3a on Hidden Unit Dynamics. The number of neurons in both hidden layers should be determined by the parameter `num_hid`.
6. [1 mark] Run the code by typing

```
python3 spiral_main.py --net short
```

You should experiment to find a good value for the initial weight size, and try to find the minimum number of hidden nodes per layer so that this ShortNet learns to correctly classify all of the training data within 20000 epochs, on almost all runs. Include in your report the number of hidden nodes per layer, as well as the initial weight size and any other metaparameters. The `graph_output()` method will generate a picture of the function computed by your ShortNet called `short_out.png`, which you should include in your report.



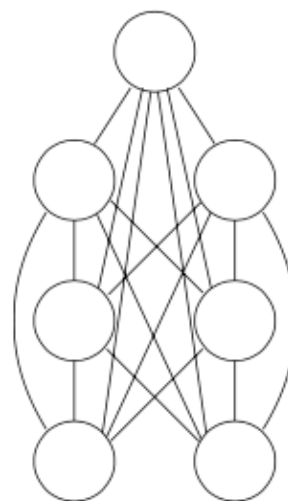
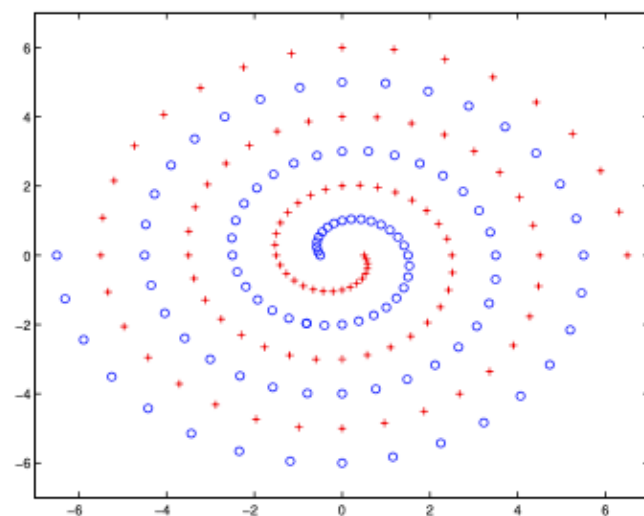
# Comp9444

## Part 2

Some functions cannot be learned with a 2-layer sigmoidal network.



spiral.py



For example, this Twin Spirals problem cannot be learned with a 2-layer network, but it can be learned using a 3-layer network if we include shortcut connections between non-consecutive layers.

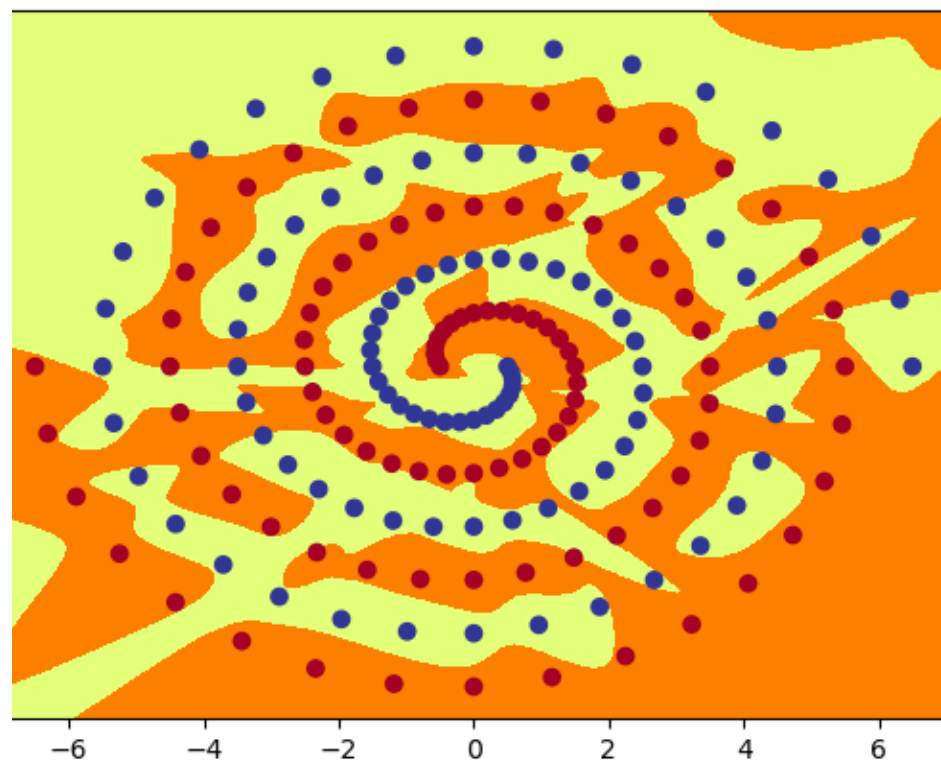
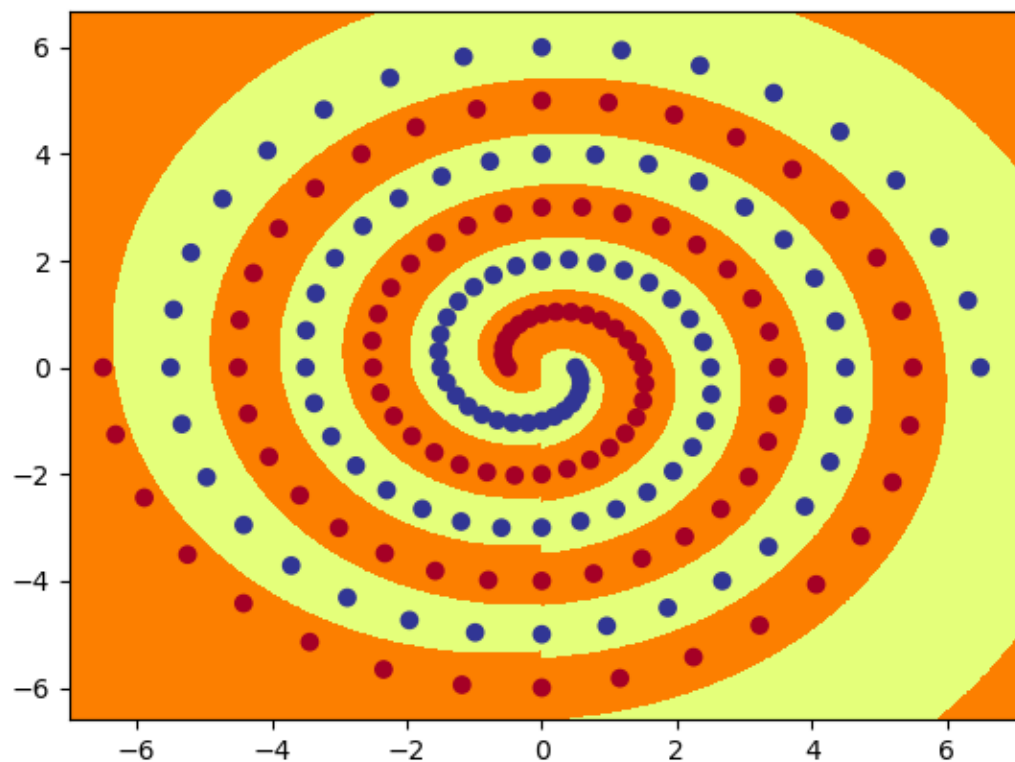


# Comp9444

## Part 2



spiral.py







# Comp9444

## Part 2

7. [2 marks] Using `graph_output()` as a guide, write a method called `graph_hidden(net, layer, node)` which plots the activation (after applying the `tanh` function) of the hidden node with the specified number (`node`) in the specified `layer` (1 or 2). (Note: if `net` is of type `PolarNet`, `graph_output()` only needs to behave correctly when `layer` is 1).

Hint: you might need to modify `forward()` so that the hidden unit activations are retained, i.e. replace `hid1 = torch.tanh(...)` with `self.hid1 = torch.tanh(...)`

Use this code to generate plots of all the hidden nodes in `PolarNet`, and all the hidden nodes in both layers of `RawNet` and `ShortNet`, and include them in your report.



spiral.py

8. [9 marks] Discuss what you have learned from this exercise, including the following points:
- the qualitative difference between the functions computed by the hidden layer nodes of the three models, and a brief description of how the network uses these functions to achieve the classification
  - the effect of different values for initial weight size on the speed and success of learning, for both `RawNet` and `ShortNet`
  - the relative "naturalness" of the output function computed by the three networks, and the importance of representation for deep learning tasks in general
  - you may like to also experiment with other changes and comment on the result - for example, changing batch size from 97 to 194, using SGD instead of Adam, changing `tanh` to `relu`, adding a third hidden layer, etc.



# Comp9444

## Part 2

```
def graph_hidden(net, layer, node):  
    plt.clf()  
    # INSERT CODE HERE
```



spiral\_main.py

```
for layer in [1,2]:  
    if layer == 1 or args.net != 'polar':  
        for node in range(args.hid):  
            graph_hidden(net, layer, node)  
            plt.scatter(full_input[:,0],full_input[:,1],  
                        c=1-full_target[:,0],cmap='RdYlBu')  
            plt.savefig('%s%d_%d.png' % (args.net, layer, node))  
  
graph_output(net)  
plt.scatter(full_input[:,0],full_input[:,1],  
            c=1-full_target[:,0],cmap='RdYlBu')  
plt.savefig('%s_out.png' % args.net)
```



# Comp9444

## Part 2



spiral\_main.py

```
def graph_output(net):
    xrange = torch.arange(start=-7, end=7.1, step=0.01, dtype=torch.float32)
    yrange = torch.arange(start=-6.6, end=6.7, step=0.01, dtype=torch.float32)
    xcoord = xrange.repeat(yrange.size()[0])
    ycoord = torch.repeat_interleave(yrange, xrange.size()[0], dim=0)
    grid = torch.cat((xcoord.unsqueeze(1), ycoord.unsqueeze(1)), 1)

    with torch.no_grad(): # suppress updating of gradients
        net.eval()        # toggle batch norm, dropout
        output = net(grid)
        net.train()       # toggle batch norm, dropout back again

    pred = (output >= 0.5).float()

    # plot function computed by model
    plt.clf()
    plt.pcolormesh(xrange, yrange, pred.cpu().view(yrange.size()[0], xrange.size()[0]), cmap='Wistia')
```



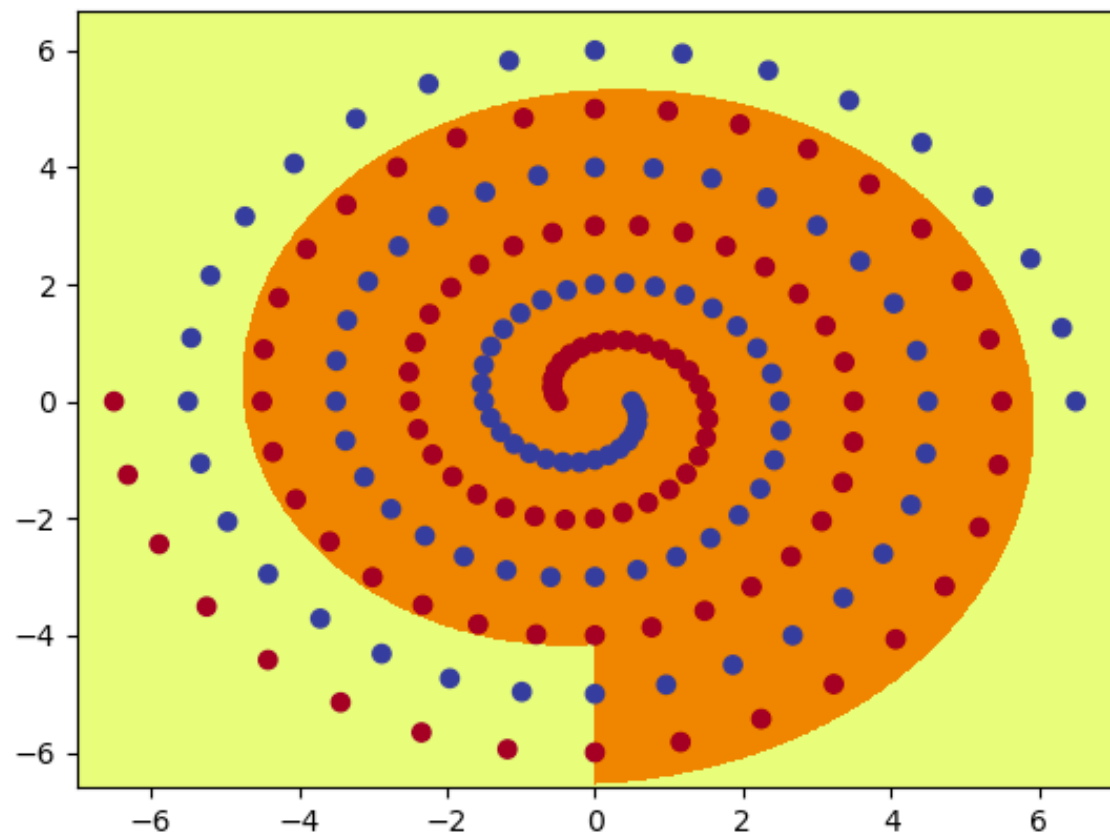


# Comp9444

## Part 2



spiral.py



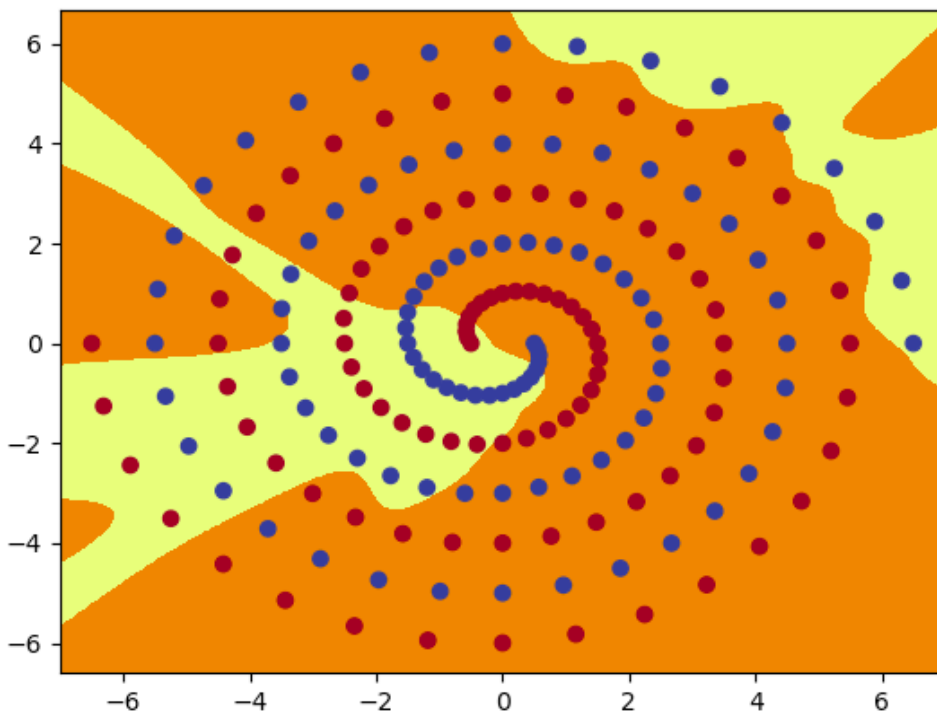
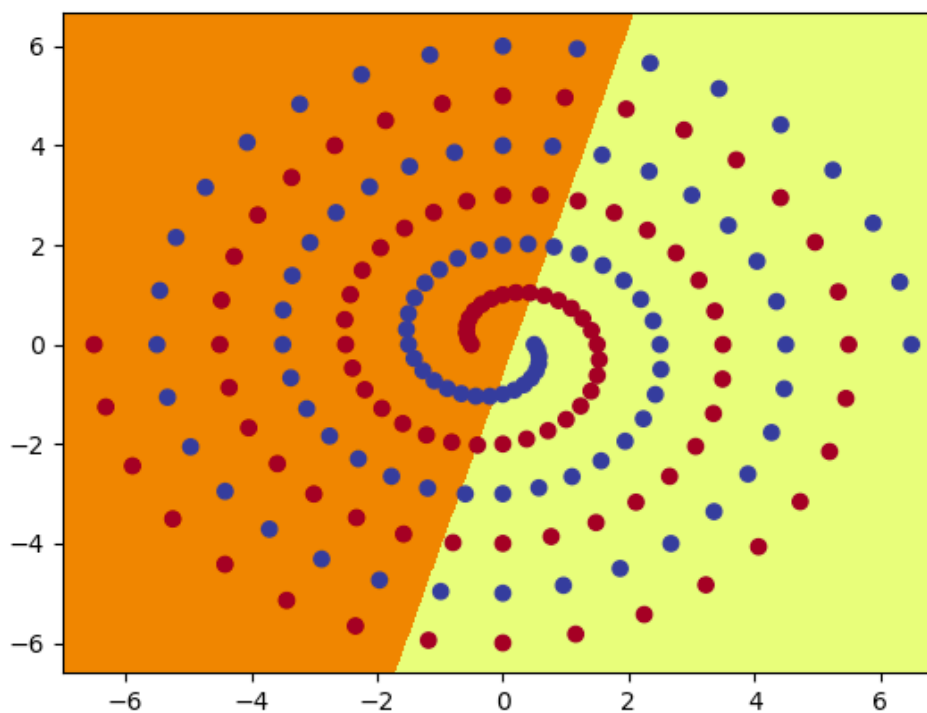


# Comp9444

## Part 2



spiral.py





# Comp9444

## Part 2



spiral.py

