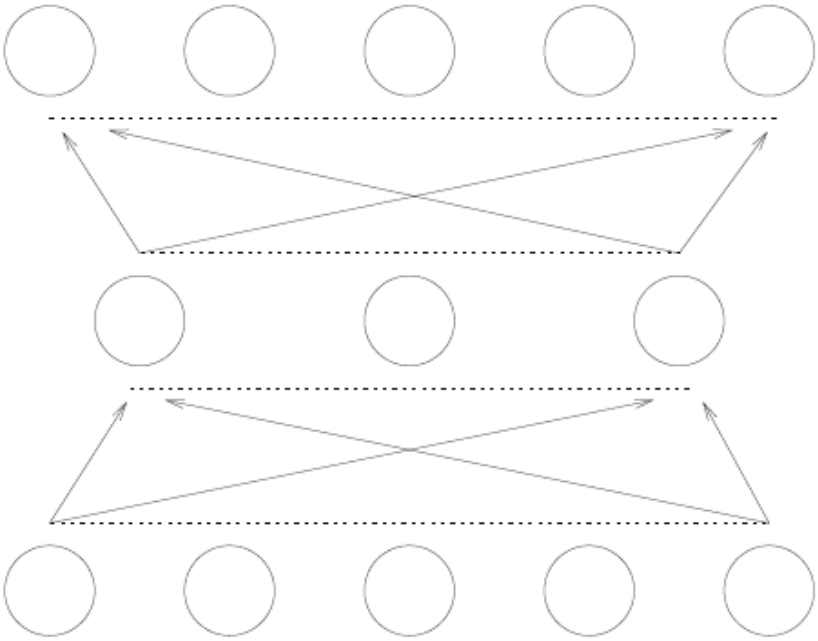
2c: Hidden Unit Dynamics

Hidden Unit Dynamics

Encoder Networks

The $N-K-N$ Encoder task is a simple supervised learning task which has been designed to help us understand the hidden unit dynamics of neural networks, and also serves as a simplified version of the Autoencoders we will meet in Week 6.



Inputs	Outputs
10000	10000
01000	01000
00100	00100
00010	00010
00001	00001

For this task, the jth item follows a **one-hot** encoding, with its jth input equal to 1 and all other inputs equal to 0. The target output is exactly the same as the input. However, the challenge lies in the fact that the input must be "compressed" to fit through the bottleneck of the K hidden nodes where $K < N$, and then reconstructed from this lower dimensional space.

Run this code, and then click on "encoder.png" to visualize the hidden unit space for a $9-2-9$ Encoder network.

```
# encoder_main.py
# COMP9444, CSE, UNSW

from __future__ import print_function
import torch
import torch.utils.data
import torch.nn.functional as F
import matplotlib.pyplot as plt
#import numpy as np
```

```
class EncModel(torch.nn.Module):
    # fully connected two-layer network
    def __init__(self, num_input, num_hid, num_out):
        super(EncModel, self).__init__()
        self.in_hid  = torch.nn.Linear(num_input, num_hid)
        self.hid_out = torch.nn.Linear(num_hid, num_out)
    def forward(self, input):
        hid_sum = self.in_hid(input)
        hidden  = torch.tanh(hid_sum)
        out_sum = self.hid_out(hidden)
        output  = torch.sigmoid(out_sum)
        return(output)

def plot_hidden(net):
    # plot the hidden unit dynamics of the network
    plt.xlim(-1,1), plt.ylim(-1,1) # limits of x and y axes

    # input to hidden weights and biases
    weight = net.in_hid.weight.data.cpu()
    bias   = net.in_hid.bias.data.cpu()

    num_in  = net.in_hid.weight.data.size()[1]
    num_out = net.hid_out.weight.data.size()[0]

    # draw a dot to show where each input is mapped to in hidden unit space
    P = torch.tanh(weight + bias.unsqueeze(1).repeat(1,num_in))
    plt.plot(P[0,:],P[1,:],'bo')

    # draw a line interval to show the decision boundary of each output
    for i in range(num_out):

        A = net.hid_out.weight.data.cpu()[i,0]
        B = net.hid_out.weight.data.cpu()[i,1]
        C = net.hid_out.bias.data.cpu()[i]

        j = 0;
        if A == 0:
            if B != 0:
                y0 = -C/B
                if -1 < y0 and y0 < 1:
                    j = 2
                    plt.plot([-1,1],[y0,y0])
        elif B == 0:
            if A != 0:
                x0 = -C/A
                if -1 < x0 and x0 < 1:
                    plt.plot([x0,x0],[-1,1])
        else:
            x = torch.zeros(2)
            y = torch.zeros(2)
            y0 = (A-C)/B
            if -1 <= y0 and y0 <= 1:
                x[j] = -1
```

```
                y[j] =  y0
                j = j+1
            y0 = (-A-C)/B
            if -1 <= y0 and y0 <= 1:
                x[j] =  1
                y[j] =  y0
                j = j+1
            x0 = (B-C)/A
            if j < 2 and -1 <= x0 and x0 <= 1:
                x[j] =  x0
                y[j] = -1
                j = j+1
            x0 = (-B-C)/A
            if j < 2 and -1 <= x0 and x0 <= 1:
                x[j] =  x0
                y[j] =  1
                j = j+1
            if j > 1:
                plt.plot(x,y)

def main():
    target = torch.eye(9)

    num_in  = target.size()[0]
    num_out = target.size()[1]

    # input is one-hot with same number of rows as target
    input  = torch.eye(num_in)
    target = torch.eye(num_in)

    xor_dataset  = torch.utils.data.TensorDataset(input,target)
    train_loader = torch.utils.data.DataLoader(xor_dataset,batch_size=num_in)

    # create neural network according to model specification
    net = EncModel(num_in,2,num_out)

    # initialize weights, but set biases to zero
    net.in_hid.weight.data.normal_(0,0.001)
    net.hid_out.weight.data.normal_(0,0.001)
    net.in_hid.bias.data.zero_()
    net.hid_out.bias.data.zero_()

    # SGD optimizer
    optimizer = torch.optim.SGD(net.parameters(),lr=0.4,momentum=0.9)

    loss = 1.0
    epoch = 0

    while epoch < 100000 and loss > 0.02:
        epoch = epoch+1
        for batch_id, (data,target) in enumerate(train_loader):
            #data, target = data.to(device), target.to(device)
            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
            if epoch % 100 == 0:
                print('ep%3d: loss = %7.4f' % (epoch, loss.item()))

    plot_hidden(net)
    plt.savefig("encoder.png")

if __name__ == '__main__':
    main()
```

Each dot in this image corresponds to a particular input pattern, and shows us the activations of the two hidden nodes (horizontal and vertical axis) when this input pattern is fed to the network. Each line in the image corresponds to a particular output node, and shows the dividing line between those points in hidden unit space for which this output is less than 0.5, and those for which it is greater than 0.5 .

Optional video

Weight Space Symmetry

- for any two hidden nodes in the same layer, if we swap the connections coming in and out from those nodes, the overall function will be the same
- assuming the activation function is symmetric, if we reverse the sign of all incoming and outgoing weights for a particular node (and adjust the subsequent bias, if necessary) the overall

function will be the same
- hidden nodes with identical input-to-hidden weights in theory would never separate; so, they all have to begin with different (small) random weights
- in practice, all hidden nodes try to do similar job at first, then gradually specialize

Controlled Nonlinearity

- for small weights, each layer implements an approximately linear function, so multiple layers also implement an approximately linear function
- for large weights, transfer function approximates a step function, so computation becomes digital and learning becomes very slow
- with typical weight values, two-layer neural network implements a function which is close to linear, but takes advantage of a limited degree of nonlinearity

Optional video

Further Reading

Lister, R., 1993. Visualizing weight dynamics in the N-2-N encoder. In *IEEE International Conference on Neural Networks* (pp. 684-689).

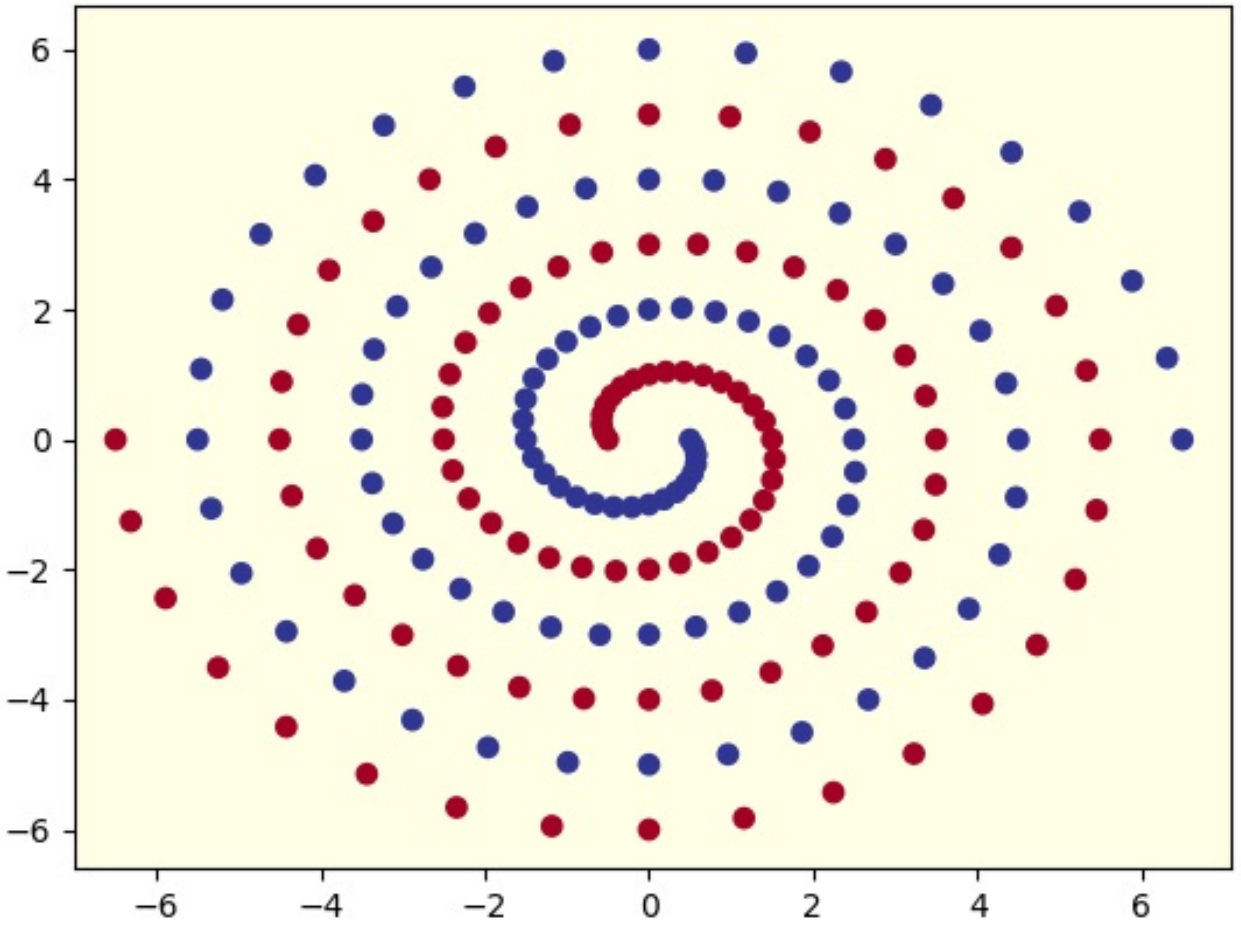
Textbook Deep Learning (Goodfellow, Bengio, Courville, 2016):

- Geometry of hidden unit activations (8.2)

Deeper Networks

Vanishing / Exploding Gradients

In general, neural networks with more hidden layers are able to implement a wider class of functions.



For example, this Twin Spirals problem cannot be learned with a 2-layer sigmoidal network, but it can be learned with a 3-layer network (Lang & Witbrock, 1988). The first hidden layer learns features that are linearly separable, the second hidden layer learns features that we could informally describe as "convex", and the third (output) layer learns the target function, which we could say has a more "concave" appearance.

It is tempting to think that any function could be learned, simply by adding more hidden layers to the network. However, it turns out that training networks with many hidden layers by backpropagation is not so easy, due to the problem of **Vanishing or Exploding Gradients**.

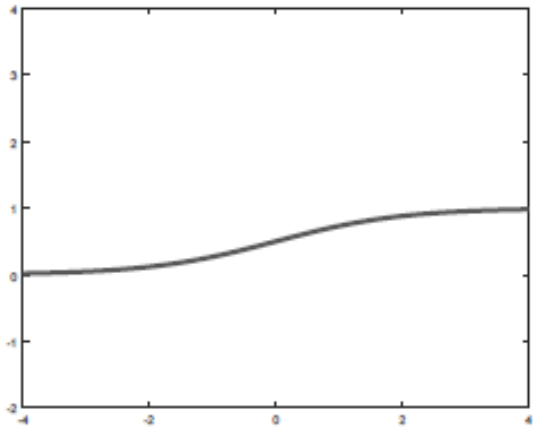
When the weights are small, the differentials become smaller and smaller as we backpropagate through the layers and end up having no effect. When the weights are large, the activations in the higher layers may saturate to extreme values. As a result, the gradients at those layers would become very small, and would not be propagated to the earlier layers. When the weights have intermediate values, the differentials can sometimes get amplified in places where the transfer function is steep, causing them to blow up to large values.

Dealing with Deep Networks

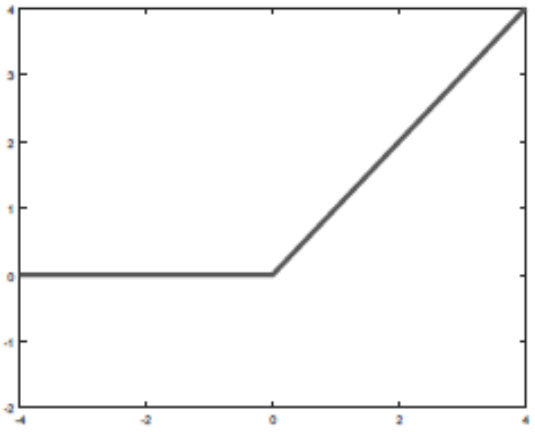
We will explore a number of enhancements which have been introduced in recent years to allow successful training of deeper networks; the main ones are summarised in this table:

4 - 9 layers:	New Activation Functions (ReLU, SeLU)
10-30 layers:	Weight Initialization, Batch Normalisation
30-100 layers:	Skip Connections (Residual Networks)
more than 100 layers:	Identity Skip Connections, Dense Networks

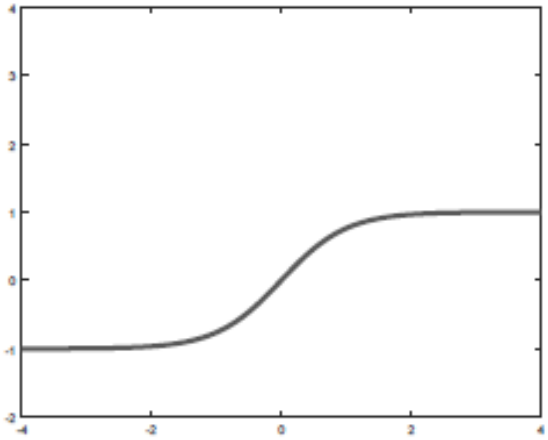
Activation Functions



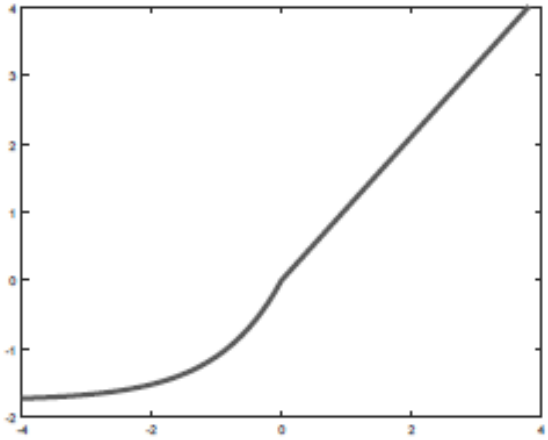
Sigmoid



Rectified Linear Unit (ReLU)



Hyperbolic Tangent



Scaled Exponential Linear Unit (SELU)

The sigmoid and hyperbolic tangent were traditionally used for 2-layer networks, but suffer from the vanishing gradient problem in deeper networks. Rectified Linear Units (ReLUs) have become popular since 2012 for deep networks, including convolutional networks. The gradients are multiplied by either 0 or 1 at each node (depending on the activation) and are therefore less likely to vanish or explode. Scaled Exponential Linear Units (SELUs) are a more recent innovation, which seem to also work well for deep networks.

Optional video

References

Lang, K.J., & Witbrock, M.J., 1988. Learning to tell two spirals apart. In *Proceedings of the 1988 connectionist models summer school* No. 1989 (pp. 52-59).

Further Reading

Textbook Deep Learning (Goodfellow, Bengio, Courville, 2016):

- Activation Functions (6.3)

Exercise: Hidden Unit Dynamics

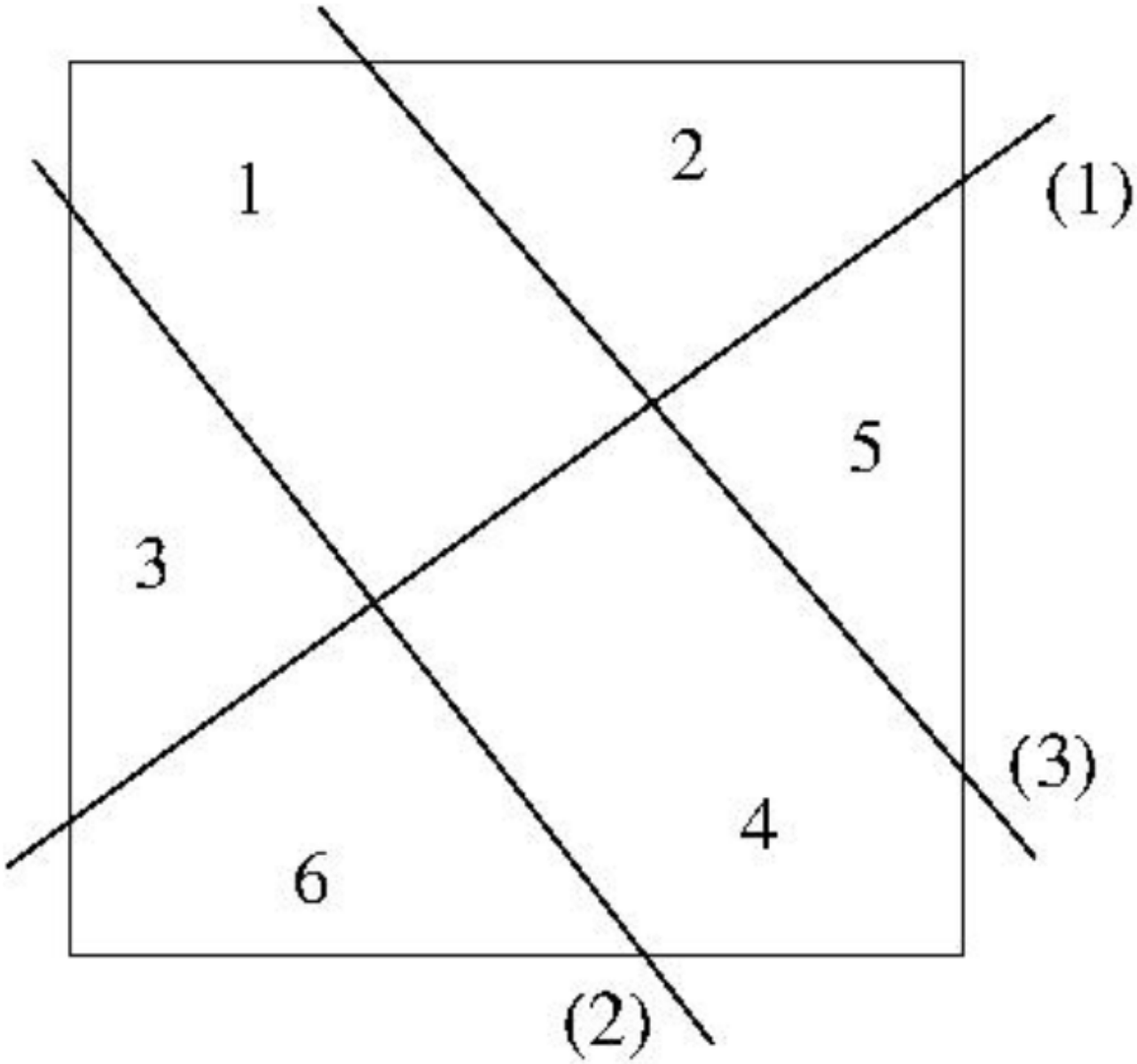
Question 1 *Submitted Sep 7th 2022 at 11:54:31 pm*

Consider a fully connected feedforward neural network with 6 inputs, 2 hidden units and 3 outputs, using $\tanh$ activation at the hidden units and $\mathrm{sigmoid}$ at the outputs. Suppose this network is trained on the following data, and that the training is successful.

Item	Inputs	Outputs
	123456	123
1.	100000	000
2.	010000	001
3.	001000	010
4.	000100	100
5.	000010	101
6.	000001	110

Draw a diagram showing:

- for each input, a point in hidden unit space corresponding to that input, and
- for each output, a line dividing the hidden unit space into regions for which the value of that output is greater/less than one half.



Question 2 *Submitted Sep 7th 2022 at 11:54:17 pm*

Consider an $8-3-8$ encoder with its three-dimensional hidden unit space. What shape would be formed by the 8 points representing the input-to-hidden weights for the 8 input units? What shape would be formed by the planes representing the hidden-to-output weights for each output unit?

Hint: think of two platonic solids, which are “dual” to each other.

Solution

The 8 points would move toward the 8 corners of a cube. Each of the 8 planes representing the hidden-to-output weights would effectively "chop off" one corner of the cube, to form a shape known as a cuboctahedron.