Red neuronal simple con una capa oculta

Objetivo

- Crear una red neuronal simple en pytorch.

Tabla de contenido

Usará una red neuronal de una única capa para clasificar datos no linealmente separables.

Preparación

In [11]:
```
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

In [12]:
```
# Import the libraries we need for this lab

import torch
import torch.nn as nn
from torch import sigmoid
import matplotlib.pylab as plt
import numpy as np
torch.manual_seed(0)
```

Out[12]:
```
<torch._C.Generator at 0x1f282db6250>
```

Para graficar:

In [13]:
```
# The function for plotting the model

def PlotStuff(X, Y, model, epoch, leg=True):

    plt.plot(X.numpy(), model(X).detach().numpy(), label=('epoch ' + str(epoch)))
    plt.plot(X.numpy(), Y.numpy(), 'r')
    plt.xlabel('x')
    if leg == True:
        plt.legend()
    else:
        pass
```

Módulo Neural Network Module y función de entrenamiento

Definimos las activaciones y la salida de la primer capa lineal como un atributo. No obstante observe que esto no es una buena práctica.

In [14]:
```
# Define the class Net

class Net(nn.Module):

    # Constructor
    def __init__(self, D_in, H, D_out):
        super(Net, self).__init__()
        # hidden layer
        self.linear1 = nn.Linear(D_in, H)
        self.linear2 = nn.Linear(H, D_out)
        # Define the first linear layer as an attribute, this is not good practice
        self.a1 = None
        self.l1 = None
        self.l2=None

    # Prediction
    def forward(self, x):
        self.l1 = self.linear1(x)
        self.a1 = sigmoid(self.l1)
        self.l2=self.linear2(self.a1)
        yhat = sigmoid(self.linear2(self.a1))
        return yhat
```

Definimos la función de entrenamiento:

In [15]:
```
# Define the training function

def train(Y, X, model, optimizer, criterion, epochs=1000):
    cost = []
    total=0
    for epoch in range(epochs):
        total=0
        for y, x in zip(Y, X):
            yhat = model(x)
            loss = criterion(yhat, y)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            #cumulative loss
            total+=loss.item()
        cost.append(total)
        if epoch % 300 == 0:
            PlotStuff(X, Y, model, epoch, leg=True)
            plt.show()
            model(X)
            plt.scatter(model.a1.detach().numpy()[:, 0], model.a1.detach().numpy()[:,
            plt.title('activations')
            plt.show()
    return cost
```

Creamos algunos datos

In [16]:
```
# Make some data

X = torch.arange(-20, 20, 1).view(-1, 1).type(torch.FloatTensor)
Y = torch.zeros(X.shape[0])
Y[(X[:, 0] > -4) & (X[:, 0] < 4)] = 1.0
```

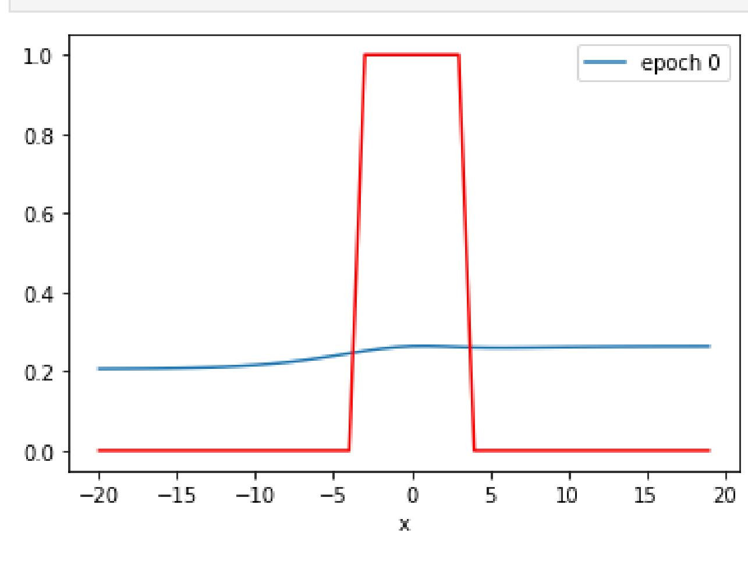
Definimos la red neuronal, función de criterio, optimizador y entrenamos el modelo

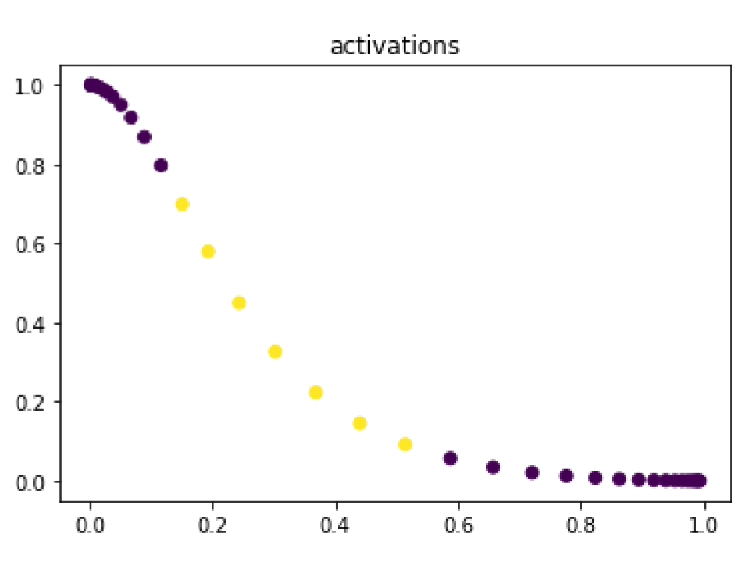
Creamos la función de costo Cross-Entropy:

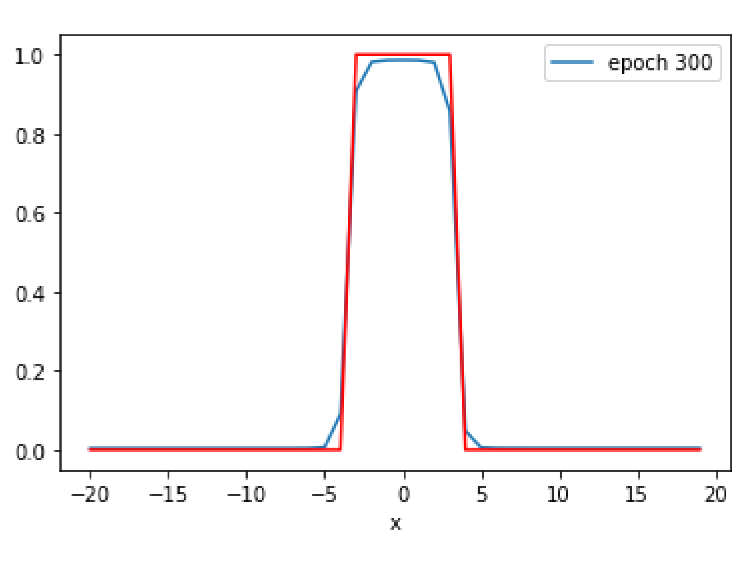
In [17]:
```
# The loss function

def criterion_cross(outputs, labels):
    out = -1 * torch.mean(labels * torch.log(outputs) + (1 - labels) * torch.log(1 -
    return out
```

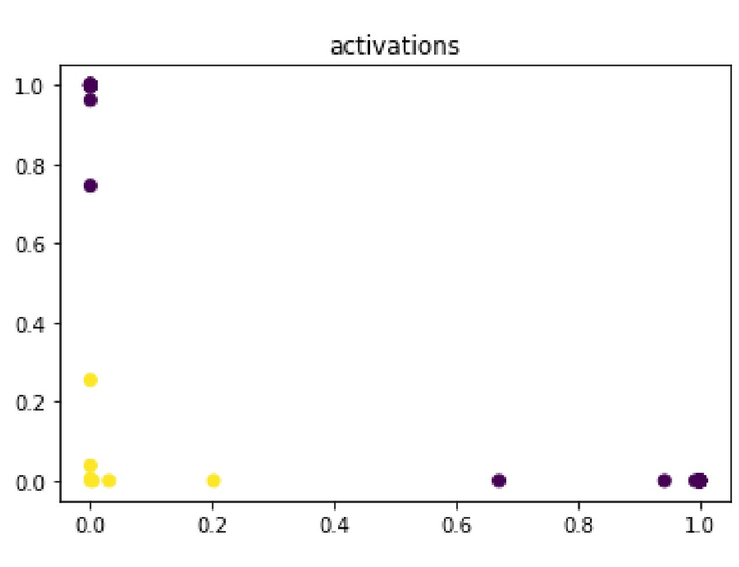
Definimos la red neuronal, optimizador y entrenamos el modelo:

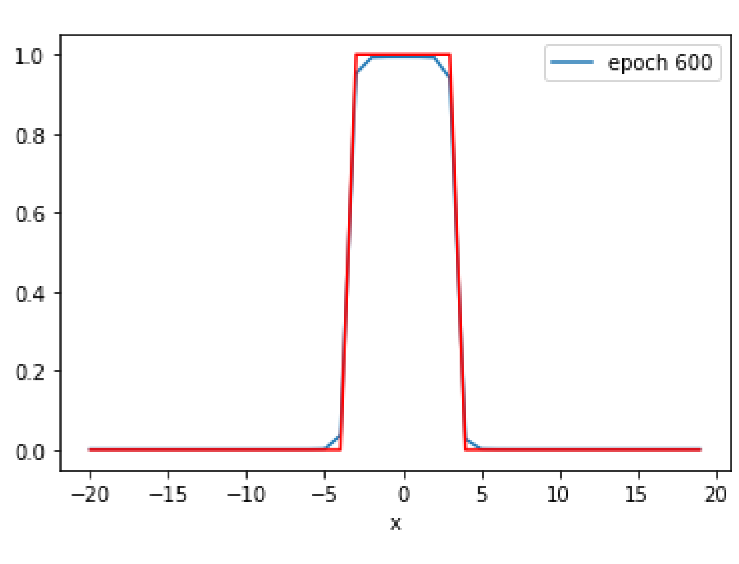
In [18]:
```
# Train the model
# size of input
D_in = 1
# size of hidden layer
H = 2
# number of outputs
D_out = 1
# learning rate
learning_rate = 0.1
# create the model
model = Net(D_in, H, D_out)
#optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
#train the model usein
cost_cross = train(Y, X, model, optimizer, criterion_cross, epochs=1000)
#plot the loss
plt.plot(cost_cross)
plt.xlabel('epoch')
plt.title('cross entropy loss')
```

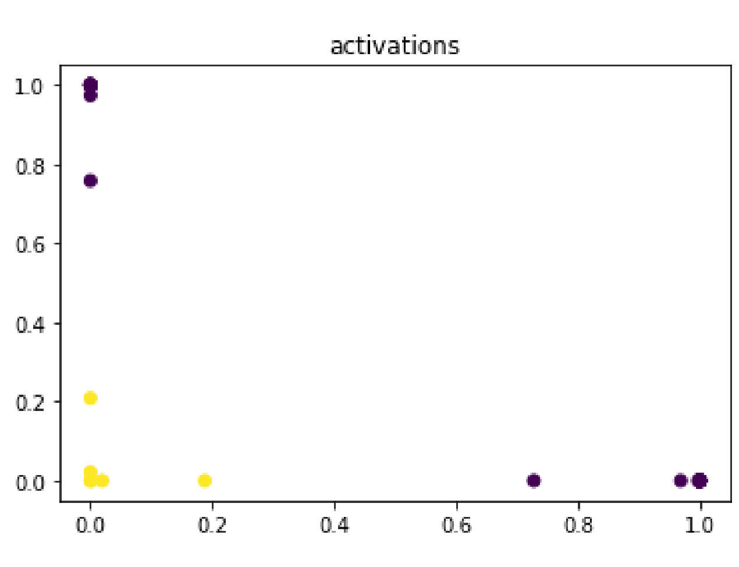


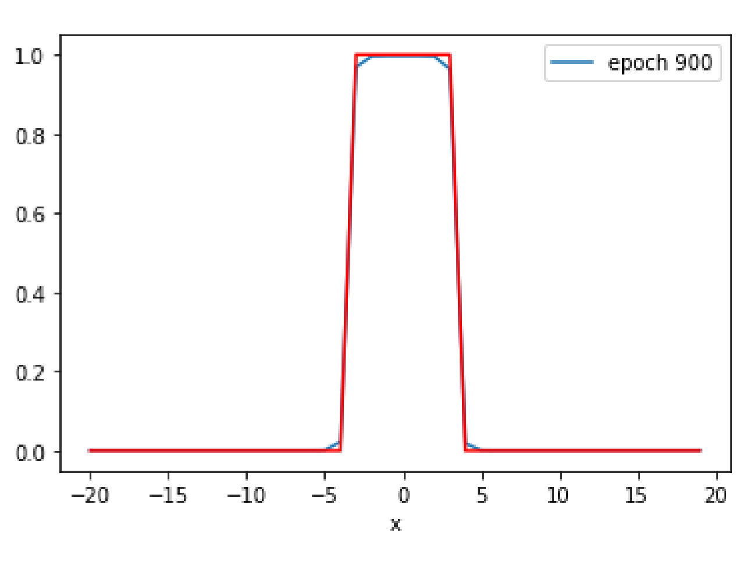


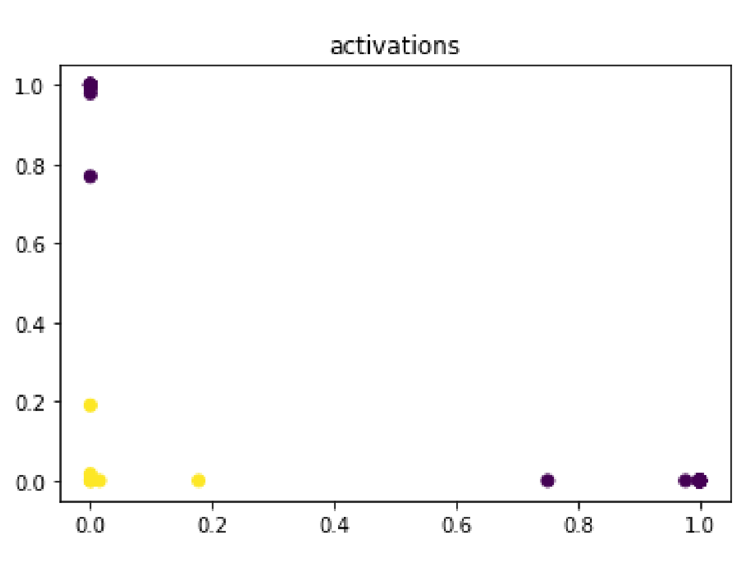




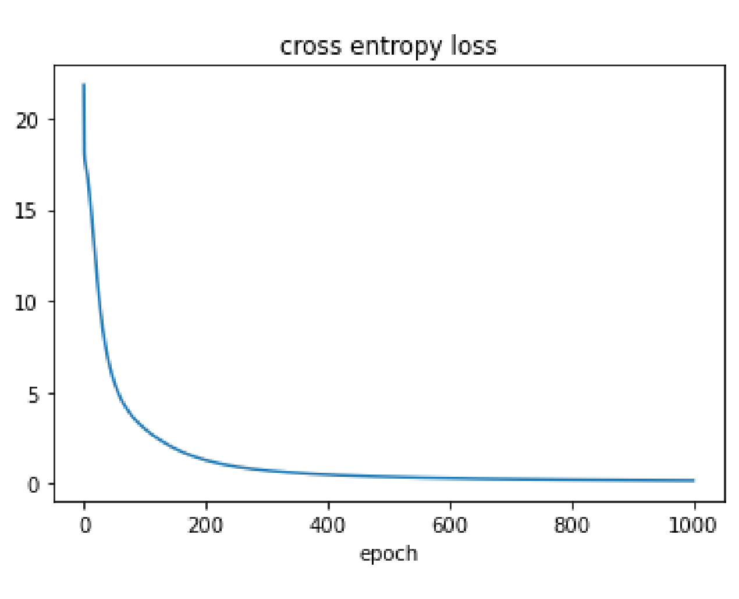








Out[18]:
```
Text(0.5, 1.0, 'cross entropy loss')
```



Examinando la salida de la activación, vemos que para el epoch 600 los datos han sido mapeados a un espacio linealmente separable:

podemos realizar una predicción para tensores arbitrarios

In [19]:
```
x=torch.tensor([0.0])
yhat=model(x)
yhat
```

Out[19]:
```
tensor([0.9969], grad_fn=<SigmoidBackward>)
```

In [20]:
```
X_=torch.tensor([[0.0],[2.0],[3.0]])
Yhat=model(X_)
Yhat
```

Out[20]:
```
tensor([[0.9969],
        [0.9963],
        [0.9680]], grad_fn=<SigmoidBackward>)
```

podemos aplicar un umbral:

In [21]:
```
Yhat=Yhat>0.5
Yhat
```

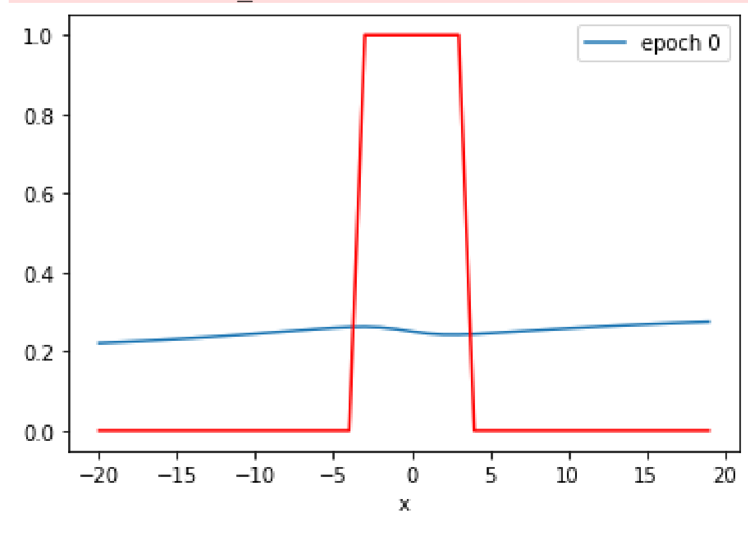
Out[21]:
```
tensor([[True],
        [True],
        [True]])
```

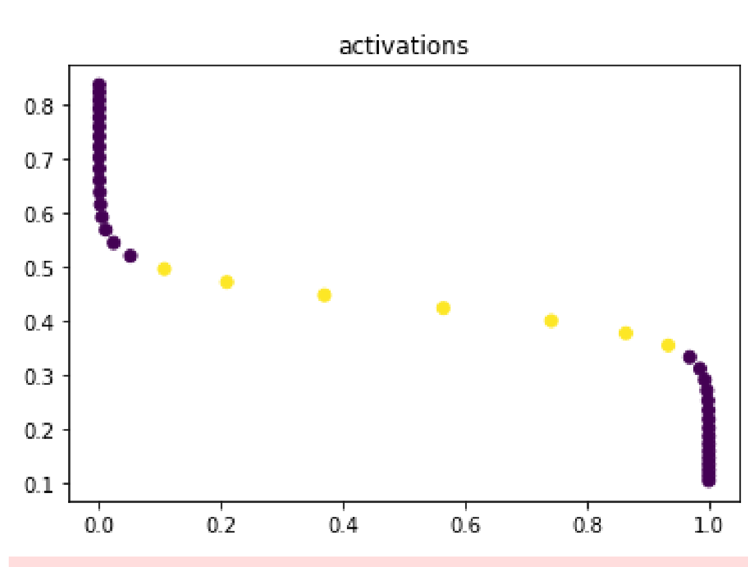
Práctica

Repita los pasos anteriores pero utilizando el costo MSE:

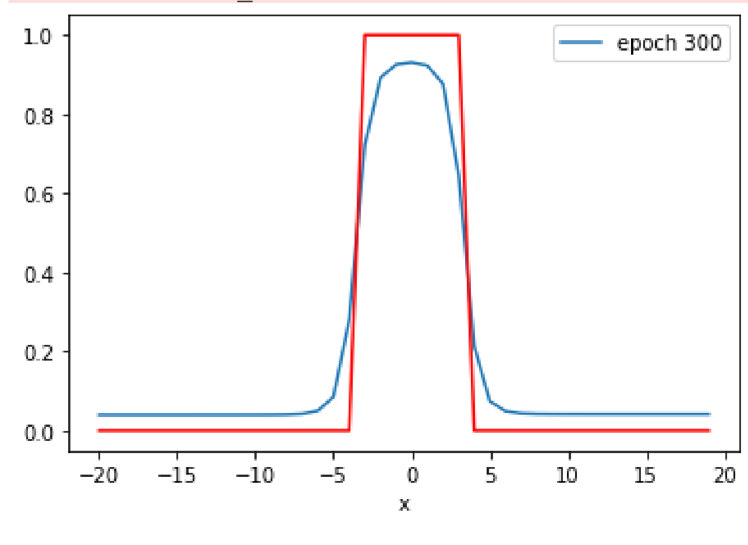
In [22]:
```
# Practice: Train the model with MSE Loss Function

learning_rate = 0.1
criterion_mse=nn.MSELoss()
model=Net(D_in,H,D_out)
optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate)
cost_mse=train(Y,X,model,optimizer,criterion_mse,epochs=1000)
plt.plot(cost_mse)
plt.xlabel('epoch')
plt.title('MSE loss ')
```

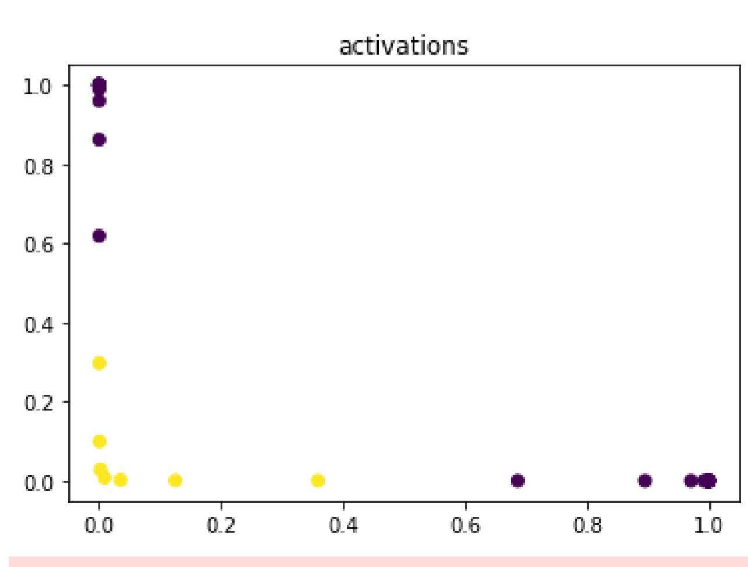
```
C:\Users\marco\Anaconda3\lib\site-packages\torch\nn\modules\loss.py:528: UserWarning:
Using a target size (torch.Size([])) that is different to the input size (torch.Size
([1])). This will likely lead to incorrect results due to broadcasting. Please ensure
they have the same size.
  return F.mse_loss(input, target, reduction=self.reduction)
```



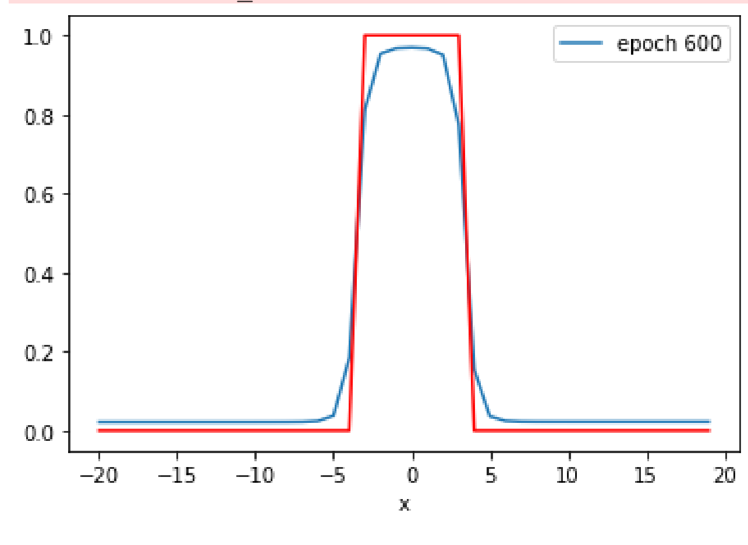


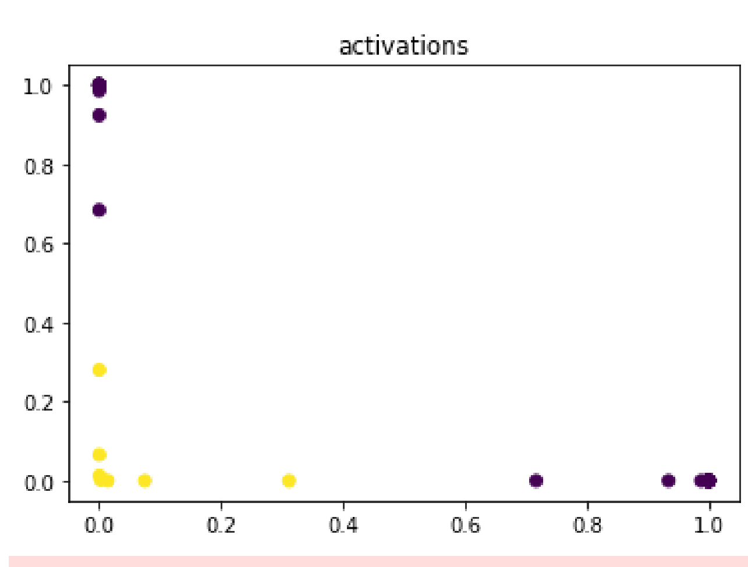
```
C:\Users\marco\Anaconda3\lib\site-packages\torch\nn\modules\loss.py:528: UserWarning:
Using a target size (torch.Size([])) that is different to the input size (torch.Size
([1])). This will likely lead to incorrect results due to broadcasting. Please ensure
they have the same size.
  return F.mse_loss(input, target, reduction=self.reduction)
```



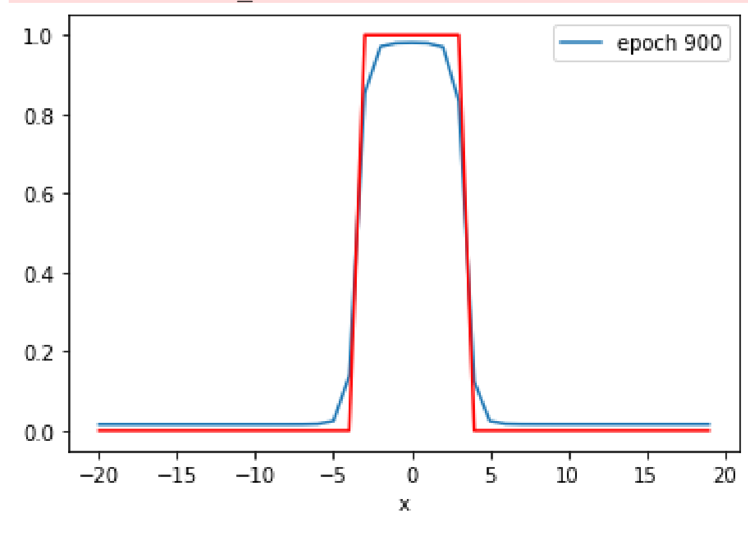


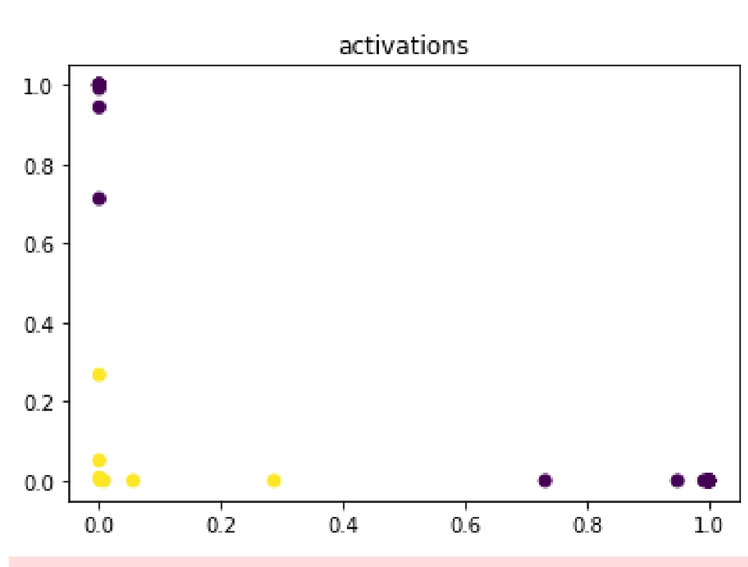
```
C:\Users\marco\Anaconda3\lib\site-packages\torch\nn\modules\loss.py:528: UserWarning:
Using a target size (torch.Size([])) that is different to the input size (torch.Size
([1])). This will likely lead to incorrect results due to broadcasting. Please ensure
they have the same size.
  return F.mse_loss(input, target, reduction=self.reduction)
```





```
C:\Users\marco\Anaconda3\lib\site-packages\torch\nn\modules\loss.py:528: UserWarning:
Using a target size (torch.Size([])) that is different to the input size (torch.Size
([1])). This will likely lead to incorrect results due to broadcasting. Please ensure
they have the same size.
  return F.mse_loss(input, target, reduction=self.reduction)
```





```
C:\Users\marco\Anaconda3\lib\site-packages\torch\nn\modules\loss.py:528: UserWarning:
Using a target size (torch.Size([])) that is different to the input size (torch.Size
([1])). This will likely lead to incorrect results due to broadcasting. Please ensure
they have the same size.
  return F.mse_loss(input, target, reduction=self.reduction)
```

Out[22]:
```
Text(0.5, 1.0, 'MSE loss ')
```