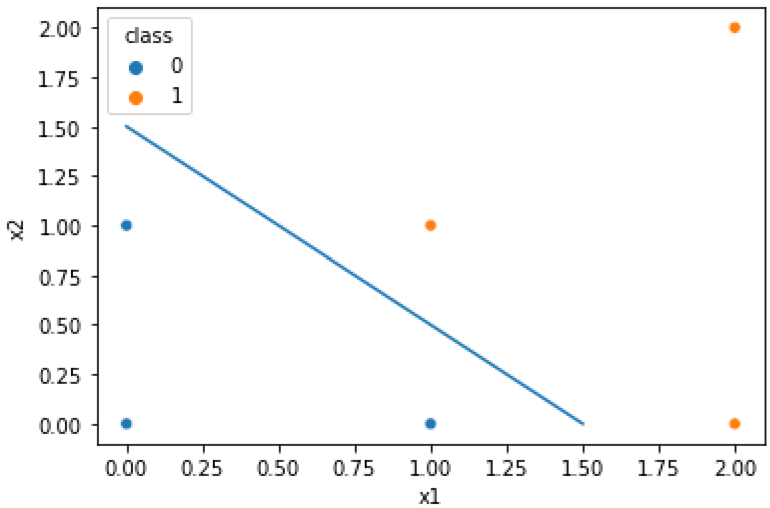
Sumit Kumar, 2020249



1. Yes, the points are linearly separable.

2. We know that the equation of the line is y-y1 = [(y1-y2)/(x1-x2)] * (x-x1) where (x1,y1) & (x2,y2) are the midpoints of any two closest points of two different classes. Suppose we consider (x1,y1)=(1,0.5)[mid-point of (0,1) and (1,1)] and (x2,y2)=(1,0)[the midpoint of (1,0) & (2,0)], after simplification, the equation of the optimal hyperplane turns out to be x + y - 1.5 = 0. In this case, (0,1), (1,1),(1,0), and (2,0) become the support vectors as these vectors are the closest to the line from different classes. From the equation of the above line, we can see that the weight vector comes out to be (1, 1) as the coefficients of x and y in the line.

3. As we can see, there are four support vectors. Each support vector is 0.5 units away from the hyperplane, and the margin becomes 1 unit as the distance between the nearest points is one unit. Suppose we remove (1,1) or (1,0) from the particular dataset. In that case, the margin will increase, while on the other hand, if we remove the other two, the optimal decision boundary will be as it is. There will be no change in the optimal hyperplane so that the margin will remain unchanged. We can verify the same by calculating the value of the distance from all the support vectors by the formula of the distance of a point from a line, i.e., d =[|Ax1 + By1 + C|]/ √(A2 + B2).

4. We get an optimal value at least as good as the previous one when we remove some constraints from a constrained maximization problem. This is because the set of candidates satisfying the initial (more robust) set of constraints is a subset of the candidates fulfilling the current (weaker) set of constraints. The old optimal solution is still available, and there may be better alternatives. In SVM, we maximize the margin while keeping the constraints imposed by training points in mind. Depending on the dataset, dropping any constraints can cause the margin to increase or remain the same. Generally, in the case of realistic datasets, it is expected that the margin will increase when we drop support vectors. But in this data, we saw that removing or keeping the support vectors can increase or preserve the margin unchanged, as we saw above

In [12]:
```
import tensorflow as tf
import matplotlib.pyplot as plt
import math
import random
import numpy as np
import pandas as pd
from copy import deepcopy
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import pickle
(x,y) ,(x_, y_)=tf.keras.datasets.mnist.load_data()
```

In [13]:
```
x=x.reshape(60000,784)
x_=x_.reshape(10000,784)
```

In [14]:
```
class Mnn():
    acti_fns = ['relu', 'sigmoid', 'linear', 'tanh', 'softmax','leaky_relu']
    weight_inits = ['zero', 'random', 'normal']

    def __init__(self, n_layers = 3, layer_sizes = [768,1,10], activation = "tanh", lea
        self.min_loss = 100000000
        self.weights = []
        self.biases = []
        self.n_layers = n_layers
        self.layer_sizes = layer_sizes
        self.convergence = convergence

        if activation not in self.acti_fns:
            raise Exception('Incorrect Activation Function')
        else:
            self.activation = activation

        self.learning_rate =learning_rate


        if weight_init not in self.weight_inits:
            raise Exception('Incorrect Weight Initialization Function')
        else:
            self.weight_init = weight_init


        self.batch_size = batch_size
        self.num_epochs = num_epochs

        if(weight_init=="zero"):
            for i in range(self.n_layers-1):
                weight = self.zero_init(shape =(self.layer_sizes[i],self.layer_sizes[i+
                self.weights.append(weight)
        elif(weight_init=="random"):
            for i in range(self.n_layers-1):
                weight = self.random_init((self.layer_sizes[i],self.layer_sizes[i+1]))
                self.weights.append(weight)
        elif(weight_init=="normal"):
            for i in range(self.n_layers-1):
```

```
                weight = self.normal_init((self.layer_sizes[i],self.layer_sizes[i+1]))
                self.weights.append(weight)
        else:
            raise Exception("Error in setting weights")

        for i in range(self.n_layers-1):
            bias = self.zero_init((1,self.layer_sizes[i+1]))
            self.biases.append(bias)

    def relu(self, X):
        return X*(X>0)

    def relu_grad(self, X):
        return np.array(X>0,dtype=int)

    def sigmoid(self, X):
        return 1/(1+np.exp(-X))

    def sigmoid_grad(self, X):
        return self.sigmoid(X)*(1-self.sigmoid(X))

    def linear(self, X):
        return X

    def linear_grad(self, X):
        return np.ones(X.shape)

    def tanh(self, X):
        return np.tanh(X)

    def tanh_grad(self, X):
        return 1 - np.tanh(X)**2

    def softmax(self, X):
        new_arr = []
        # print(type(X[0]))
        for i in X:
            # print(type(i))
            exponential = np.exp(i)
            total = exponential.sum()
            new_arr.append(exponential/total)
        return np.array(new_arr)

    def softmax_grad(self, X):
        return X*(1-X)

    def leaky_relu(self,z):
        return np.maximum(0.01 * z, z)
    def leaky_relu_gradient(self,z):
        grad = np.ones_like(z)
        grad[z < 0] = 0.01
        return grad

    def zero_init(self, shape):
        return np.zeros(shape)

    def random_init(self, shape):
        return np.random.rand(shape[0],shape[1])*0.01

    def normal_init(self, shape):
```

```
        return np.random.normal(size = shape)*0.01

    def activate(self, X):
        if(self.activation == "relu"):
            return self.relu(X)
        elif(self.activation == "sigmoid"):
            return self.sigmoid(X)
        elif(self.activation == "linear"):
            return self.linear(X)
        elif(self.activation == "tanh"):
            return self.tanh(X)
        elif(self.activation == "softmax"):
            return self.softmax(X)
        elif(self.activation=='leaky_relu'):
            return self.leaky_relu(X)
        else:
            print("error in activate fucntion")

    def activate_grad(self, X):
        if(self.activation == "relu"):
            return self.relu_grad(X)
        elif(self.activation == "sigmoid"):
            return self.sigmoid_grad(X)
        elif(self.activation == "linear"):
            return self.linear_grad(X)
        elif(self.activation == "tanh"):
            return self.tanh_grad(X)
        elif(self.activation == "softmax"):
            return self.softmax_grad(X)
        elif(self.activation=='leaky_relu'):
            return self.leaky_relu_gradient(X)
        else:
            print("error in activate fucntion grad")

    def cross_entropy(self, y_pred, y_true):
        ce =  -1*np.log(y_pred[np.arange(len(y_true)), y_true.argmax(axis=1)])
        return np.sum(ce)

    def forward(self, X):
        before_activation = []
        after_activation = []
        x = deepcopy(X)
        for i in range(self.n_layers-2):
            op = x.dot(self.weights[i]) + self.biases[i]
            before_activation.append(op)
            op = self.activate(op)
            after_activation.append(op)
            x = op
        op = x.dot(self.weights[-1]) + self.biases[-1]
        before_activation.append(op)
        op = self.softmax(op)
        after_activation.append(op)
        return before_activation, after_activation

    def backward(self, y, before_activation, after_activation):
        grads = []
        final_pred = after_activation[-1]
        loss = final_pred - y
        grads.append(loss)
        for layer in range(self.n_layers - 3, -1, -1):
```

```
            curr_error = loss.dot(self.weights[layer+1].T)
            grad = self.activate_grad(before_activation[layer])
            loss = curr_error*grad
            grads.append(loss)
        grads.reverse()
        return grads

    def fit(self, X, y, X_test=None, y_test=None):
        loss = []
        val_loss = []
        for epoch in range(self.num_epochs):
            for batch in range(0,len(X),self.batch_size):
                currX = X[batch:batch+self.batch_size,:]
                currY = y[batch:batch+self.batch_size,:]
                bef,aft = self.forward(currX)
                grads = self.backward(currY, bef, aft)
                zumm = currX
                for i in range(self.n_layers-1):
                    grad = zumm.T.dot(grads[i])/len(currX)
                    zumm = aft[i]
                    self.weights[i] = self.weights[i] - self.learning_rate*grad
                    self.biases[i] = self.biases[i] - self.learning_rate*np.sum(grads[i
            #cross entropy
            b,a = self.forward(X)
            loss.append(self.cross_entropy(a[-1],y)/len(y))
            if(loss[-1]<self.min_loss):
                self.min_loss = loss[-1]
            b,a = self.forward(X_test)
            val_loss.append(self.cross_entropy(a[-1],y_test)/len(y_test))
            print("epoch",epoch,", loss:",loss[-1])
            if(self.convergence != None):
                if((loss[-1] - self.min_loss > 0.1)):
                    print("Stopping iteration due to convergence (minima lost)")
                    break
                if(len(loss)>2 and epoch > self.num_epochs//5):
                    if(abs(loss[-2] - loss[-1]) < self.convergence):
                        print("Stopping iteration due to convergence")
                        break
        self.loss = loss
        self.val_loss = val_loss
        return self

    def predict_proba(self, X):
        return self.forward(X)[1][-1]

    def predict(self, X):
        return self.forward(X)[1][-1].argmax(axis=1)

    def score(self, X, y):
        y_pred = self.predict(X)
        c = 0
        for i in range(len(y_pred)):
            if(y[i][y_pred[i]]==1):
                    c+=1
        return c/len(y_pred)
```

In [4]:

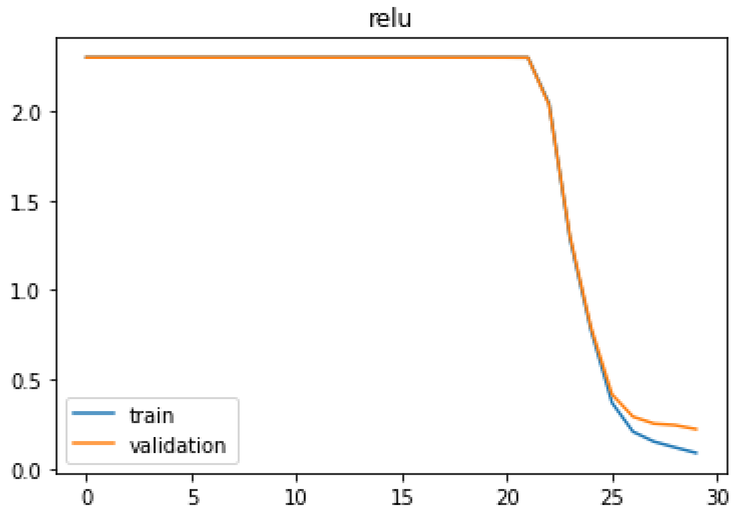
```
scaler = StandardScaler()
temp = np.zeros((y.size, int(y.max())+1))
```

```
temp[np.arange(y.size), y.astype(int)] = 1
y = temp
X_train, X_testval, y_train, y_testval = train_test_split(x, y, test_size=0.2)
X_train = scaler.fit_transform(X_train)
X_testval = scaler.transform(X_testval)
X_test, X_val, y_test, y_val = train_test_split(X_testval, y_testval, test_size=0.5)
```

In [5]:

```
nn_relu = Mnn(n_layers=6, layer_sizes=[784,256,128,64,32,10],activation="relu", weight_
nn_relu.fit(X_train,y_train,X_val,y_val)
plt.plot(nn_relu.loss,label="train")
plt.plot(nn_relu.val_loss, label="validation")
plt.title("relu")
plt.legend()
plt.show()
print(nn_relu.score(X_test,y_test))
pickle.dump(nn_relu,open("relu.pkl","wb"))
```

```
epoch 0 , loss: 2.301485468167068
epoch 1 , loss: 2.3014910598291354
epoch 2 , loss: 2.301490905197303
epoch 3 , loss: 2.301490450987469
epoch 4 , loss: 2.3014900140127703
epoch 5 , loss: 2.3014895386766074
epoch 6 , loss: 2.301488969463436
epoch 7 , loss: 2.301488342209451
epoch 8 , loss: 2.3014875580341907
epoch 9 , loss: 2.301486668487556
epoch 10 , loss: 2.301485601322965
epoch 11 , loss: 2.3014842597344063
epoch 12 , loss: 2.301482576527112
epoch 13 , loss: 2.301480403974597
epoch 14 , loss: 2.3014774934993234
epoch 15 , loss: 2.301473432540421
epoch 16 , loss: 2.30146746064785
epoch 17 , loss: 2.3014580087059384
epoch 18 , loss: 2.3014413849464583
epoch 19 , loss: 2.3014071511900944
epoch 20 , loss: 2.301314373930616
epoch 21 , loss: 2.3008293854275412
epoch 22 , loss: 2.044115591321575
epoch 23 , loss: 1.2855667701479911
epoch 24 , loss: 0.7743509769904707
epoch 25 , loss: 0.3681201011971479
epoch 26 , loss: 0.20577041826471873
epoch 27 , loss: 0.15008088636094283
epoch 28 , loss: 0.1177848184554482
epoch 29 , loss: 0.0869424084183926
```



0.9438333333333333

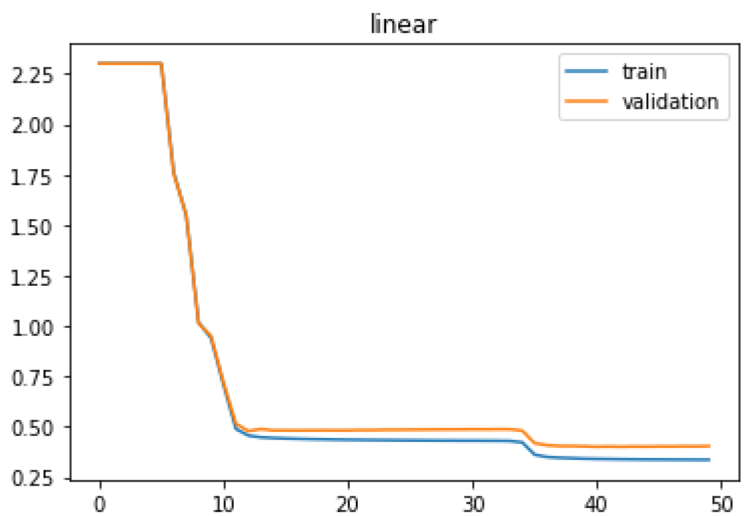
In [5]:

In [6]:

```
nn_linear = Mnn(n_layers=6, layer_sizes=[784,256,128,64,32,10],activation="linear", wei
nn_linear.fit(X_train,y_train,X_val,y_val)
plt.plot(nn_linear.loss,label="train")
plt.plot(nn_linear.val_loss, label="validation")
plt.title("linear")
plt.legend()
plt.show()
print(nn_linear.score(X_test,y_test))
pickle.dump(nn_relu,open("linear.pkl","wb"))
```

```
epoch 0 , loss: 2.3014743770020303
epoch 1 , loss: 2.3014683207667472
epoch 2 , loss: 2.301451211312131
epoch 3 , loss: 2.301421673684004
epoch 4 , loss: 2.3013559318304613
epoch 5 , loss: 2.3010836102793117
epoch 6 , loss: 1.76087097679784
epoch 7 , loss: 1.547689029074755
epoch 8 , loss: 1.0171809592654708
epoch 9 , loss: 0.94044575014983
epoch 10 , loss: 0.7089441259570743
epoch 11 , loss: 0.48935614177893505
epoch 12 , loss: 0.45623236355557084
epoch 13 , loss: 0.4473737377615199
epoch 14 , loss: 0.4439204681693736
epoch 15 , loss: 0.4415254511367745
epoch 16 , loss: 0.43976175889094876
epoch 17 , loss: 0.43834087212246525
epoch 18 , loss: 0.43726792756454713
epoch 19 , loss: 0.4361098891298429
epoch 20 , loss: 0.43524991365471405
epoch 21 , loss: 0.43461778449927996
epoch 22 , loss: 0.4338275307393148
epoch 23 , loss: 0.43335757808790853
epoch 24 , loss: 0.43287624009029135
epoch 25 , loss: 0.43237676228539074
epoch 26 , loss: 0.43202273546012576
```

```
epoch 27 , loss: 0.4315555104427561
epoch 28 , loss: 0.4313254701176669
epoch 29 , loss: 0.43089446189823777
epoch 30 , loss: 0.43080105251331063
epoch 31 , loss: 0.43028903675216473
epoch 32 , loss: 0.430153699727276
epoch 33 , loss: 0.42932451594457993
epoch 34 , loss: 0.42225066205371475
epoch 35 , loss: 0.3608284355574697
epoch 36 , loss: 0.34948795061907556
epoch 37 , loss: 0.3457421402115054
epoch 38 , loss: 0.34354065020714764
epoch 39 , loss: 0.34128235191470013
epoch 40 , loss: 0.34005928672207075
epoch 41 , loss: 0.3393754208509637
epoch 42 , loss: 0.33843920231706887
epoch 43 , loss: 0.3378294420173866
epoch 44 , loss: 0.33725845371058083
epoch 45 , loss: 0.33696987559632574
epoch 46 , loss: 0.33659577640481425
epoch 47 , loss: 0.3362716421839427
epoch 48 , loss: 0.335922726706864
epoch 49 , loss: 0.33580556144378315
```



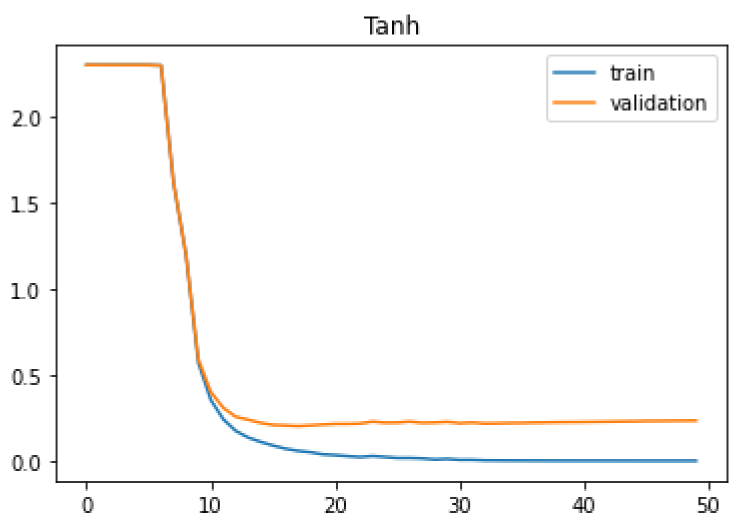

```
0.8953333333333333
```

In [7]:

```
nn_tanh = Mnn(n_layers=6, layer_sizes=[784,256,128,64,32,10],activation="tanh", weight_
nn_tanh.fit(X_train,y_train,X_val,y_val)
plt.plot(nn_tanh.loss,label="train")
plt.plot(nn_tanh.val_loss, label="validation")
plt.title("Tanh")
plt.legend()
plt.show()
print(nn_tanh.score(X_test,y_test))
pickle.dump(nn_relu,open("tanh.pkl","wb"))
```

```
epoch 0 , loss: 2.3014791324174944
epoch 1 , loss: 2.301474560249197
epoch 2 , loss: 2.3014602297534545
epoch 3 , loss: 2.301436737236324
epoch 4 , loss: 2.301388973372883
epoch 5 , loss: 2.30123979718874
epoch 6 , loss: 2.2987188258083617
epoch 7 , loss: 1.6158673274699
```

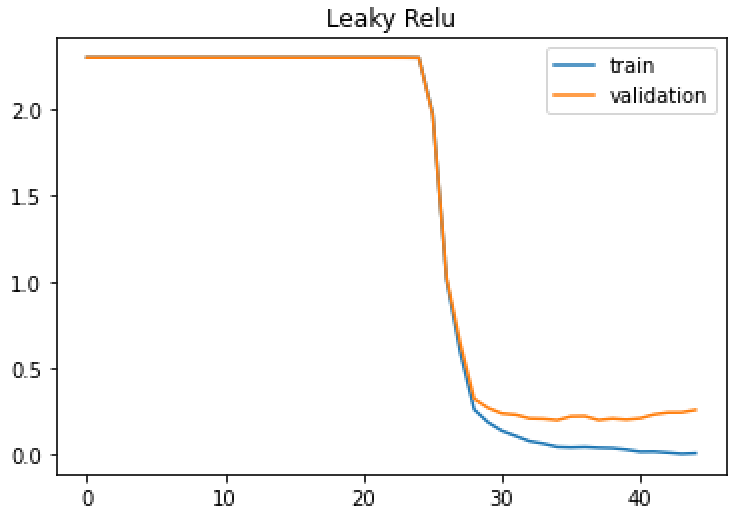
```
epoch 8 , loss: 1.1978121136680604
epoch 9 , loss: 0.5700305645876753
epoch 10 , loss: 0.3544210888180196
epoch 11 , loss: 0.2429075123586408
epoch 12 , loss: 0.17508900027785804
epoch 13 , loss: 0.13633924359281088
epoch 14 , loss: 0.11123053466653711
epoch 15 , loss: 0.08970456484319915
epoch 16 , loss: 0.07122716170432657
epoch 17 , loss: 0.05870631246229349
epoch 18 , loss: 0.05083698943907232
epoch 19 , loss: 0.038395486718069825
epoch 20 , loss: 0.033978345148395706
epoch 21 , loss: 0.027785233403186435
epoch 22 , loss: 0.022723888010002583
epoch 23 , loss: 0.028718273052217788
epoch 24 , loss: 0.023103260284369247
epoch 25 , loss: 0.01717384274230169
epoch 26 , loss: 0.017327664005036197
epoch 27 , loss: 0.014543090326300452
epoch 28 , loss: 0.008738866024971302
epoch 29 , loss: 0.011782979208425729
epoch 30 , loss: 0.006450093815754906
epoch 31 , loss: 0.00655764796803535
epoch 32 , loss: 0.0034119305448375528
epoch 33 , loss: 0.0028790288531064523
epoch 34 , loss: 0.0022787159746259587
epoch 35 , loss: 0.00207119969312837 6
epoch 36 , loss: 0.0018342195697517948
epoch 37 , loss: 0.001603746013866665
epoch 38 , loss: 0.0014294281916500417
epoch 39 , loss: 0.0013003424373838068
epoch 40 , loss: 0.0012054763505405393
epoch 41 , loss: 0.0011324749274237058
epoch 42 , loss: 0.001064982015578669
epoch 43 , loss: 0.0009747702556465967
epoch 44 , loss: 0.0009303291832054153
epoch 45 , loss: 0.0008425437217308922
epoch 46 , loss: 0.000813271351400861
epoch 47 , loss: 0.0007543321545629841
epoch 48 , loss: 0.0007232426202362238
epoch 49 , loss: 0.0007084263782989027
```




```
0.9573333333333334
```

In [8]:
```
nn_leaky_relu = Mnn(n_layers=6, layer_sizes=[784,256,128,64,32,10],activation="leaky_re
nn_leaky_relu.fit(X_train,y_train,X_val,y_val)
plt.plot(nn_leaky_relu.loss,label="train")
plt.plot(nn_leaky_relu.val_loss, label="validation")
plt.title("Leaky Relu")
plt.legend()
plt.show()
print(nn_leaky_relu.score(X_test,y_test))
pickle.dump(nn_leaky_relu,open("leaky_relu.pkl","wb"))
```

```
epoch 0 , loss: 2.301485763625222
epoch 1 , loss: 2.3014913605199743
epoch 2 , loss: 2.3014910537375197
epoch 3 , loss: 2.3014905082950907
epoch 4 , loss: 2.301489959975759
epoch 5 , loss: 2.3014893978496342
epoch 6 , loss: 2.3014888115636363
epoch 7 , loss: 2.3014881569671366
epoch 8 , loss: 2.3014874134275574
epoch 9 , loss: 2.301486569693447
epoch 10 , loss: 2.301485592124682
epoch 11 , loss: 2.3014844453194727
epoch 12 , loss: 2.3014830903819843
epoch 13 , loss: 2.30148145908657
epoch 14 , loss: 2.3014794678868937
epoch 15 , loss: 2.3014769807159823
epoch 16 , loss: 2.301473801883498
epoch 17 , loss: 2.3014695794920543
epoch 18 , loss: 2.3014637741941266
epoch 19 , loss: 2.3014552989625985
epoch 20 , loss: 2.30144197566639
epoch 21 , loss: 2.3014193392352937
epoch 22 , loss: 2.3013744116146975
epoch 23 , loss: 2.301255954120485
epoch 24 , loss: 2.3006454248305706
epoch 25 , loss: 1.9748497870101351
epoch 26 , loss: 1.0157410231502773
epoch 27 , loss: 0.5901380969914486
epoch 28 , loss: 0.262787590305588
epoch 29 , loss: 0.1885067648239623
epoch 30 , loss: 0.1392088226267297
epoch 31 , loss: 0.10938360683124872
epoch 32 , loss: 0.07854349054807441
epoch 33 , loss: 0.06432532463421434
epoch 34 , loss: 0.04587053571935216
epoch 35 , loss: 0.04279109867758852
epoch 36 , loss: 0.04542849800150416
epoch 37 , loss: 0.04076900478733739
epoch 38 , loss: 0.03870134013645452
epoch 39 , loss: 0.030181463818222336
epoch 40 , loss: 0.01752438283394341
epoch 41 , loss: 0.018299243009564967
epoch 42 , loss: 0.012630474467286093
epoch 43 , loss: 0.005339894070057351
epoch 44 , loss: 0.009057035335855425
```




```
0.9615
```

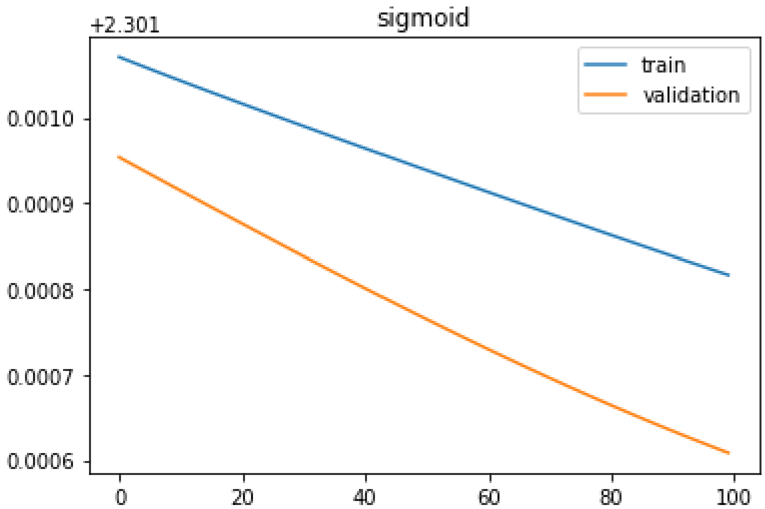
In [15]:

```
for i in [256,572,784]:
  print('Batch size{}',i)
  nn_sigmoid = Mnn(n_layers=6, layer_sizes=[784,256,128,64,32,10],activation="sigmoid",
  nn_sigmoid.fit(X_train,y_train,X_val,y_val)
  plt.plot(nn_sigmoid.loss,label="train")
  plt.plot(nn_sigmoid.val_loss, label="validation")
  plt.title("sigmoid")
  plt.legend()
  plt.show()
  print(nn_sigmoid.score(X_test,y_test))
  pickle.dump(nn_relu,open(str(i)+"sigmoid.pkl","wb"))
```

```
Batch size{} 256
epoch 0 , loss: 2.302071183759006
epoch 1 , loss: 2.3020683457017705
epoch 2 , loss: 2.302065522034392
epoch 3 , loss: 2.302062712114132
epoch 4 , loss: 2.302059915305505
epoch 5 , loss: 2.302057130996239
epoch 6 , loss: 2.3020543585968203
epoch 7 , loss: 2.302051597540037
epoch 8 , loss: 2.302048847280527
epoch 9 , loss: 2.3020461072943217
epoch 10 , loss: 2.302043377078394
epoch 11 , loss: 2.3020406561502087
epoch 12 , loss: 2.302037944047272
epoch 13 , loss: 2.3020352403266826
epoch 14 , loss: 2.3020325445646934
epoch 15 , loss: 2.3020298563562624
epoch 16 , loss: 2.3020271753146178
epoch 17 , loss: 2.302024501070823
epoch 18 , loss: 2.3020218332733426
epoch 19 , loss: 2.3020191715876126
epoch 20 , loss: 2.3020165156956205
epoch 21 , loss: 2.3020138652954807
epoch 22 , loss: 2.30201122010102
epoch 23 , loss: 2.302008579841363
epoch 24 , loss: 2.3020059442605314
epoch 25 , loss: 2.3020033131170337
epoch 26 , loss: 2.3020006861834705
epoch 27 , loss: 2.3019980632461436
```

```
epoch 28 , loss: 2.3019954441046644
epoch 29 , loss: 2.301992828571573
epoch 30 , loss: 2.3019902164719634
epoch 31 , loss: 2.301987607643106
epoch 32 , loss: 2.301985001934085
epoch 33 , loss: 2.301982399205437
epoch 34 , loss: 2.3019797993287945
epoch 35 , loss: 2.3019772021865346
epoch 36 , loss: 2.30197460767144
epoch 37 , loss: 2.301972015686353
epoch 38 , loss: 2.3019694261438497
epoch 39 , loss: 2.30196683896591
epoch 40 , loss: 2.3019642540835967
epoch 41 , loss: 2.3019616714367412
epoch 42 , loss: 2.3019590909736327
epoch 43 , loss: 2.3019565126507184
epoch 44 , loss: 2.3019539364323025
epoch 45 , loss: 2.301951362290256
epoch 46 , loss: 2.301948790203731
epoch 47 , loss: 2.301946220158883
epoch 48 , loss: 2.301943652148594
epoch 49 , loss: 2.301941086172207
epoch 50 , loss: 2.301938522235263
epoch 51 , loss: 2.3019359603492444
epoch 52 , loss: 2.3019334005313286
epoch 53 , loss: 2.3019308428041354
epoch 54 , loss: 2.3019282871954974
epoch 55 , loss: 2.3019257337382206
epoch 56 , loss: 2.3019231824698587
epoch 57 , loss: 2.301920633432492
epoch 58 , loss: 2.3019180866725124
epoch 59 , loss: 2.3019155422404065
epoch 60 , loss: 2.3019130001905603
epoch 61 , loss: 2.3019104605810488
epoch 62 , loss: 2.3019079234734483
epoch 63 , loss: 2.3019053889326466
epoch 64 , loss: 2.301902857026654
epoch 65 , loss: 2.3019003278264307
epoch 66 , loss: 2.301897801405706
epoch 67 , loss: 2.301895277840818
epoch 68 , loss: 2.3018927572105397
epoch 69 , loss: 2.301890239595927
epoch 70 , loss: 2.301887725080161
epoch 71 , loss: 2.3018852137484003
epoch 72 , loss: 2.3018827056876314
epoch 73 , loss: 2.301880200986534
epoch 74 , loss: 2.3018776997353387
epoch 75 , loss: 2.3018752020256987
epoch 76 , loss: 2.301872707950562
epoch 77 , loss: 2.301870217604045
epoch 78 , loss: 2.3018677310813156
epoch 79 , loss: 2.3018652484784763
epoch 80 , loss: 2.3018627698924523
epoch 81 , loss: 2.301860295420885
epoch 82 , loss: 2.301857825162027
epoch 83 , loss: 2.3018553592146405
epoch 84 , loss: 2.301852897677901
epoch 85 , loss: 2.3018504406513056
epoch 86 , loss: 2.3018479882345795
epoch 87 , loss: 2.301845540527592
```

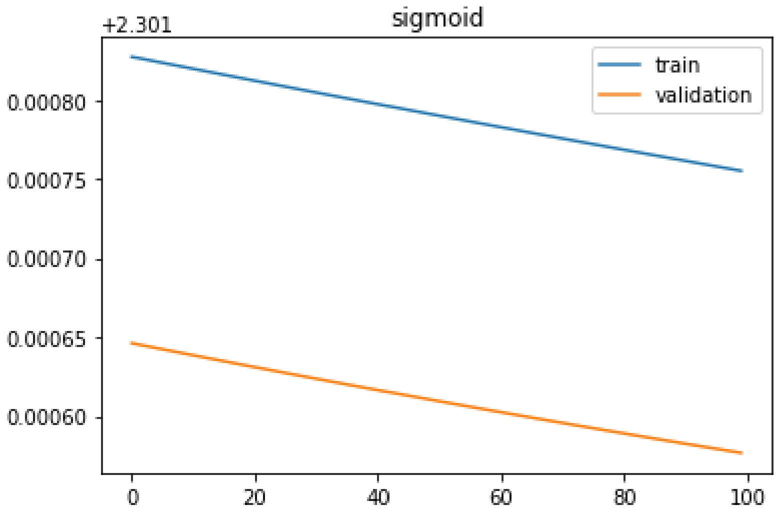
```
epoch 88 , loss: 2.3018430976302704
epoch 89 , loss: 2.301840659642521
epoch 90 , loss: 2.3018382266641515
epoch 91 , loss: 2.3018357987947966
epoch 92 , loss: 2.301833376133844
epoch 93 , loss: 2.3018309587803674
epoch 94 , loss: 2.301828546833062
epoch 95 , loss: 2.3018261403901756
epoch 96 , loss: 2.3018237395494543
epoch 97 , loss: 2.3018213444080793
epoch 98 , loss: 2.3018189550626107
epoch 99 , loss: 2.3018165716089363
```




```
0.09816666666666667
Batch size{} 572
epoch 0 , loss: 2.3018272099950163
epoch 1 , loss: 2.3018265831749227
epoch 2 , loss: 2.301825818585395
epoch 3 , loss: 2.301825054660981
epoch 4 , loss: 2.3018242914442513
epoch 5 , loss: 2.301823528936047
epoch 6 , loss: 2.301822767137208
epoch 7 , loss: 2.3018220060485652
epoch 8 , loss: 2.3018212456709484
epoch 9 , loss: 2.301820486005182
epoch 10 , loss: 2.301819727052087
epoch 11 , loss: 2.3018189688124804
epoch 12 , loss: 2.301818211287174
epoch 13 , loss: 2.301817454476977
epoch 14 , loss: 2.301816698382694
epoch 15 , loss: 2.301815943005126
epoch 16 , loss: 2.301815188345069
epoch 17 , loss: 2.301814434403316
epoch 18 , loss: 2.301813681180656
epoch 19 , loss: 2.3018129286778732
epoch 20 , loss: 2.301812176895749
epoch 21 , loss: 2.30181142583506
epoch 22 , loss: 2.3018106754965797
epoch 23 , loss: 2.3018099258810776
epoch 24 , loss: 2.301809176989317
epoch 25 , loss: 2.3018084288220617
epoch 26 , loss: 2.3018076813800676
epoch 27 , loss: 2.301806934664089
epoch 28 , loss: 2.3018061886748757
```

```
epoch 29 , loss: 2.301805443413174
epoch 30 , loss: 2.301804698879726
epoch 31 , loss: 2.3018039550752696
epoch 32 , loss: 2.30180321200054
epoch 33 , loss: 2.3018024696562676
epoch 34 , loss: 2.3018017280431797
epoch 35 , loss: 2.3018009871619993
epoch 36 , loss: 2.301800247013446
epoch 37 , loss: 2.3017995075982367
epoch 38 , loss: 2.301798768917082
epoch 39 , loss: 2.3017980309706907
epoch 40 , loss: 2.301797293759767
epoch 41 , loss: 2.3017965572850128
epoch 42 , loss: 2.301795821547124
epoch 43 , loss: 2.3017950865467953
epoch 44 , loss: 2.3017943522847157
epoch 45 , loss: 2.3017936187615717
epoch 46 , loss: 2.3017928859780463
epoch 47 , loss: 2.3017921539348176
epoch 48 , loss: 2.3017914226325615
epoch 49 , loss: 2.3017906920719486
epoch 50 , loss: 2.3017899622536473
epoch 51 , loss: 2.301789233178322
epoch 52 , loss: 2.301788504846633
epoch 53 , loss: 2.301787777259238
epoch 54 , loss: 2.30178705041679
epoch 55 , loss: 2.301786324319939
epoch 56 , loss: 2.301785598969331
epoch 57 , loss: 2.3017848743656097
epoch 58 , loss: 2.3017841505094134
epoch 59 , loss: 2.3017834274013773
epoch 60 , loss: 2.3017827050421342
epoch 61 , loss: 2.3017819834323126
epoch 62 , loss: 2.3017812625725362
epoch 63 , loss: 2.3017805424634283
epoch 64 , loss: 2.301779823105605
epoch 65 , loss: 2.3017791044996816
epoch 66 , loss: 2.301778386646269
epoch 67 , loss: 2.301777669545973
epoch 68 , loss: 2.301776953199399
epoch 69 , loss: 2.3017762376071462
epoch 70 , loss: 2.3017755227698125
epoch 71 , loss: 2.3017748086879894
epoch 72 , loss: 2.3017740953622683
epoch 73 , loss: 2.3017733827932347
epoch 74 , loss: 2.3017726709814714
epoch 75 , loss: 2.301771959927558
epoch 76 , loss: 2.3017712496320692
epoch 77 , loss: 2.3017705400955797
epoch 78 , loss: 2.3017698313186554
epoch 79 , loss: 2.3017691233018645
epoch 80 , loss: 2.3017684160457668
epoch 81 , loss: 2.301767709550923
epoch 82 , loss: 2.301767003817886
epoch 83 , loss: 2.301766298847208
epoch 84 , loss: 2.301765594639438
epoch 85 , loss: 2.3017648911951207
epoch 86 , loss: 2.301764188514797
epoch 87 , loss: 2.3017634865990044
epoch 88 , loss: 2.301762785448278
```

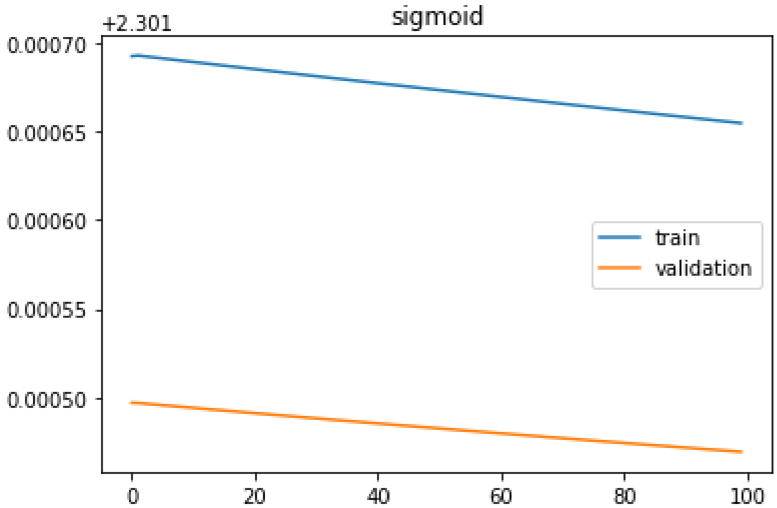
```
epoch 89 , loss: 2.301762085063148
epoch 90 , loss: 2.301761385444143
epoch 91 , loss: 2.301760686591787
epoch 92 , loss: 2.3017599885066
epoch 93 , loss: 2.3017592911891005
epoch 94 , loss: 2.3017585946398014
epoch 95 , loss: 2.3017578988592136
epoch 96 , loss: 2.3017572038478455
epoch 97 , loss: 2.3017565096061983
epoch 98 , loss: 2.3017558161347744
epoch 99 , loss: 2.301755123434069
```




```
0.09816666666666667
Batch size{} 784
epoch 0 , loss: 2.3016924090773765
epoch 1 , loss: 2.301692768913152
epoch 2 , loss: 2.301692364856191
epoch 3 , loss: 2.3016919570993806
epoch 4 , loss: 2.301691549724843
epoch 5 , loss: 2.301691142749785
epoch 6 , loss: 2.301690736174009
epoch 7 , loss: 2.3016903299972435
epoch 8 , loss: 2.301689924219217
epoch 9 , loss: 2.3016895188396584
epoch 10 , loss: 2.3016891138582993
epoch 11 , loss: 2.301688709274869
epoch 12 , loss: 2.301688305089099
epoch 13 , loss: 2.3016879013007183
epoch 14 , loss: 2.3016874979094597
epoch 15 , loss: 2.3016870949150543
epoch 16 , loss: 2.3016866923172348
epoch 17 , loss: 2.3016862901157324
epoch 18 , loss: 2.301685888310281
epoch 19 , loss: 2.3016854869006123
epoch 20 , loss: 2.301685085886461
epoch 21 , loss: 2.3016846852675608
epoch 22 , loss: 2.3016842850436445
epoch 23 , loss: 2.3016838852144472
epoch 24 , loss: 2.301683485779704
epoch 25 , loss: 2.3016830867391502
epoch 26 , loss: 2.3016826880925203
epoch 27 , loss: 2.301682289839551
epoch 28 , loss: 2.301681891979978
epoch 29 , loss: 2.301681494513538
```

```
epoch 30 , loss: 2.301681097439967
epoch 31 , loss: 2.3016807007590034
epoch 32 , loss: 2.3016803044703833
epoch 33 , loss: 2.301679908573846
epoch 34 , loss: 2.301679513069128
epoch 35 , loss: 2.301679117955969
epoch 36 , loss: 2.3016787232341067
epoch 37 , loss: 2.301678328903281
epoch 38 , loss: 2.301677934963231
epoch 39 , loss: 2.3016775414136963
epoch 40 , loss: 2.301677148254417
epoch 41 , loss: 2.301676755485133
epoch 42 , loss: 2.301676363105586
epoch 43 , loss: 2.3016759711155164
epoch 44 , loss: 2.301675579514665
epoch 45 , loss: 2.301675188302774
epoch 46 , loss: 2.301674797479585
epoch 47 , loss: 2.30167440704484
epoch 48 , loss: 2.3016740169982826
epoch 49 , loss: 2.301673627339654
epoch 50 , loss: 2.3016732380686986
epoch 51 , loss: 2.301672849185159
epoch 52 , loss: 2.3016724606887795
epoch 53 , loss: 2.301672072579304
epoch 54 , loss: 2.301671684856476
epoch 55 , loss: 2.3016712975200413
epoch 56 , loss: 2.3016709105697437
epoch 57 , loss: 2.3016705240053286
epoch 58 , loss: 2.3016701378265423
epoch 59 , loss: 2.3016697520331295
epoch 60 , loss: 2.3016693666248367
epoch 61 , loss: 2.30166898160141
epoch 62 , loss: 2.3016685969625965
epoch 63 , loss: 2.301668212708143
epoch 64 , loss: 2.301667828837796
epoch 65 , loss: 2.3016674453513035
epoch 66 , loss: 2.3016670622484128
epoch 67 , loss: 2.3016666795288723
epoch 68 , loss: 2.30166629719243
epoch 69 , loss: 2.301665915238835
epoch 70 , loss: 2.301665533667834
epoch 71 , loss: 2.30166515247918
epoch 72 , loss: 2.301664771672619
epoch 73 , loss: 2.3016643912479005
epoch 74 , loss: 2.301664011204777
epoch 75 , loss: 2.3016636315429966
epoch 76 , loss: 2.30166325226231
epoch 77 , loss: 2.301662873362468
epoch 78 , loss: 2.301662494843222
epoch 79 , loss: 2.301662116704322
epoch 80 , loss: 2.301661738945521
epoch 81 , loss: 2.3016613615665693
epoch 82 , loss: 2.30166098456722
epoch 83 , loss: 2.301660607947224
epoch 84 , loss: 2.301660231706334
epoch 85 , loss: 2.3016598558443033
epoch 86 , loss: 2.3016594803608843
epoch 87 , loss: 2.3016591052558306
epoch 88 , loss: 2.3016587305288954
epoch 89 , loss: 2.3016583561798325
```

```
epoch 90 , loss: 2.301657982208395
epoch 91 , loss: 2.3016576086143385
epoch 92 , loss: 2.301657235397416
epoch 93 , loss: 2.3016568625573814
epoch 94 , loss: 2.3016564900939915
epoch 95 , loss: 2.301656118007001
epoch 96 , loss: 2.301655746296165
epoch 97 , loss: 2.3016553749612374
epoch 98 , loss: 2.301655004001976
epoch 99 , loss: 2.3016546334181363
```




```
0.124
```

In [ ]:

In [1]:
```
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
import numpy as np
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import log_loss
from sklearn import metrics
```

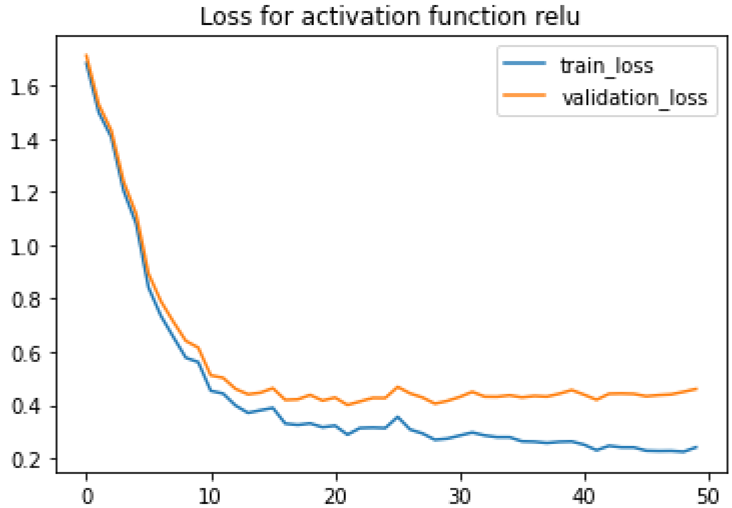
In [2]:
```
import warnings
warnings.filterwarnings('ignore')
# warnings.filterwarnings(action='once')
```

In [3]:
```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
```

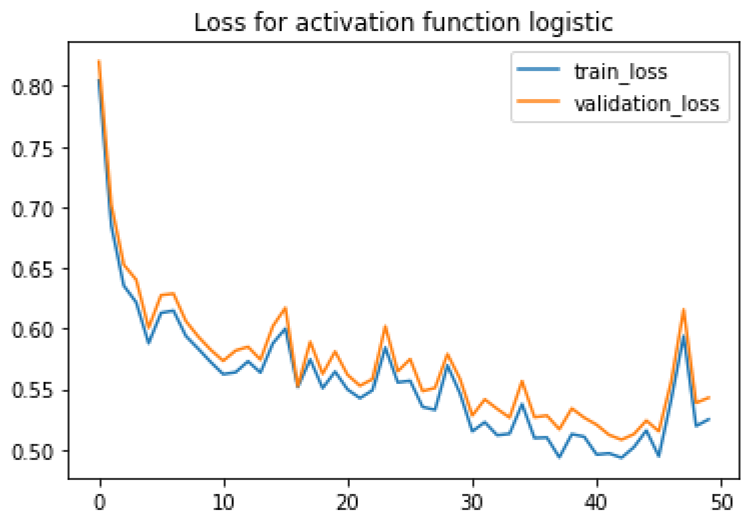
In [4]:
```
X_, x_val_, Y_, y_val_ = train_test_split(x_train, y_train, test_size=0.15, random_stat
```

In [12]:
```
def mlp_activation(X,Y,X_val,Y_val,epochs,x_test,y_test):
    functions=['relu','logistic','tanh','identity']
    for i in functions:
        loss=[]
        v_loss=[]
        mlp = MLPClassifier(hidden_layer_sizes=(256,32), activation=i, solver='adam', r
        for e in range(epochs):
            mlp.partial_fit(X.reshape(51000,784), Y, classes=np.unique(Y))

            loss.append(log_loss(Y,mlp.predict_proba(X.reshape(51000,784))))

            v_loss.append(log_loss(Y_val,mlp.predict_proba(X_val.reshape(9000,784))))
        plt.plot(range(epochs),loss,label='train_loss')
        plt.plot(range(epochs),v_loss,label='validation_loss')
        plt.legend()
        plt.title('Loss for activation function '+i)
        plt.show()
        print("Tranning Accuracy Score",metrics.accuracy_score(Y, mlp.predict(X.reshape
        print("Validation Accuracy Score",metrics.accuracy_score(Y_val, mlp.predict(X_v
        print("testing Accuracy Score",metrics.accuracy_score(y_test,mlp.predict(x_test
```

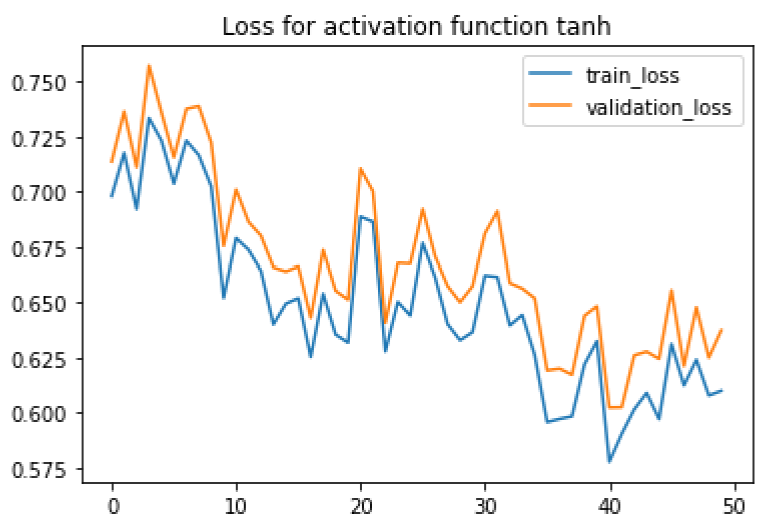
In [13]:
```
mlp_activation(X_,Y_,x_val_,y_val_,50,x_test,y_test)
```



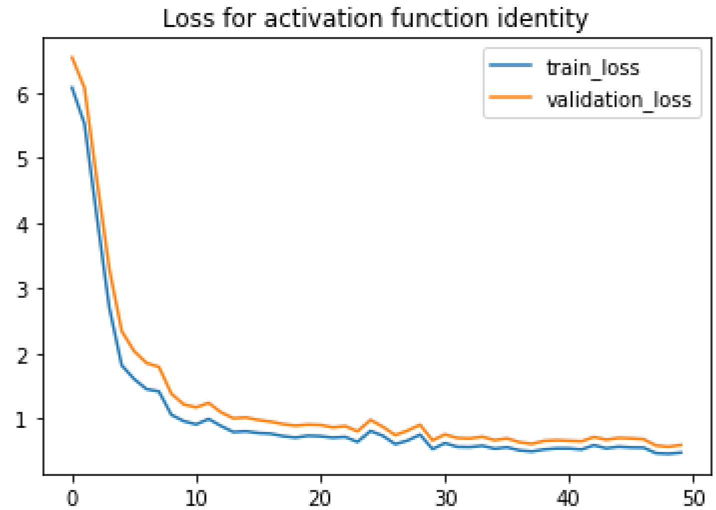

```
Tranning Accuracy Score 0.9117254901960784
Validation Accuracy Score 0.8643333333333333
testing Accuracy Score 0.8611
```




```
Tranning Accuracy Score 0.805156862745098
Validation Accuracy Score 0.798
testing Accuracy Score 0.7877
```




```
Tranning Accuracy Score 0.7764313725490196
Validation Accuracy Score 0.7652222222222222
testing Accuracy Score 0.7597
```




```
Tranning Accuracy Score 0.8441372549019608
Validation Accuracy Score 0.8241111111111111
testing Accuracy Score 0.8202
```

The best activation function is **Relu** as you can see from the above plots and Accuracies for Tranning,Validation and Testing sets all best when relu is used. The Reason for the same is because Relu does not activate all the neurons. When the linear transformation output is negative it does not activate the neuron.

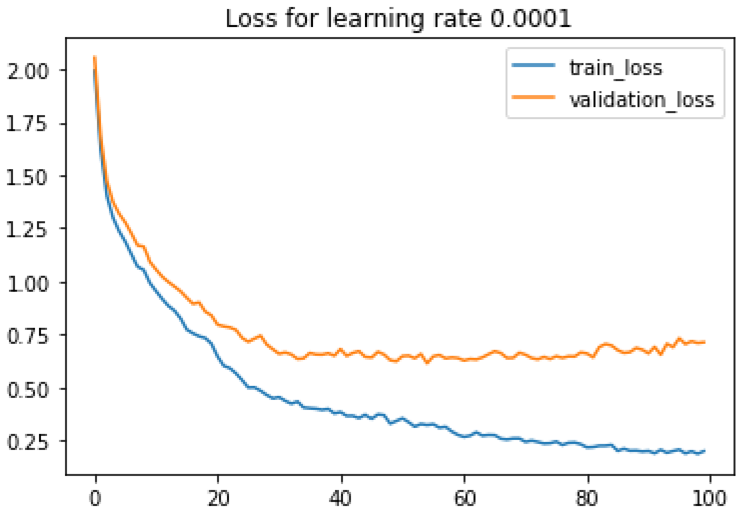
In [ ]:

In [16]:

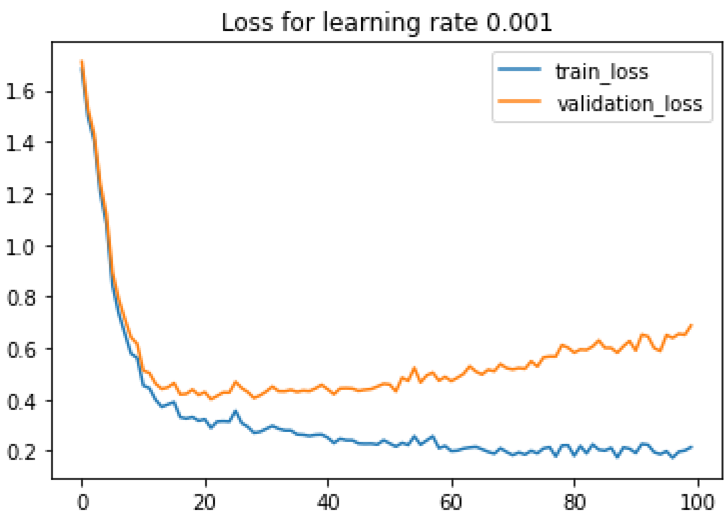
```
def mlp_lrate(X,Y,X_val,Y_val,epochs,x_test,y_test):
    l_rte=[0.0001,0.001,0.01]
    for i in l_rte:
        loss=[]
        v_loss=[]
        mlp = MLPClassifier(hidden_layer_sizes=(256,32), activation='relu', solver='ada
        for e in range(epochs):
            mlp.partial_fit(X.reshape(51000,784), Y, classes=np.unique(Y))

            loss.append(log_loss(Y,mlp.predict_proba(X.reshape(51000,784))))

            v_loss.append(log_loss(Y_val,mlp.predict_proba(X_val.reshape(9000,784))))
        plt.plot(range(epochs),loss,label='train_loss')
        plt.plot(range(epochs),v_loss,label='validation_loss')
        plt.legend()
        plt.title('Loss for learning rate '+str(i))
        plt.show()
        print("Tranning Accuracy Score",metrics.accuracy_score(Y, mlp.predict(X.reshape
        print("Validation Accuracy Score",metrics.accuracy_score(Y_val, mlp.predict(X_v
        print("testing Accuracy Score",metrics.accuracy_score(y_test,mlp.predict(x_test
```

In [17]:

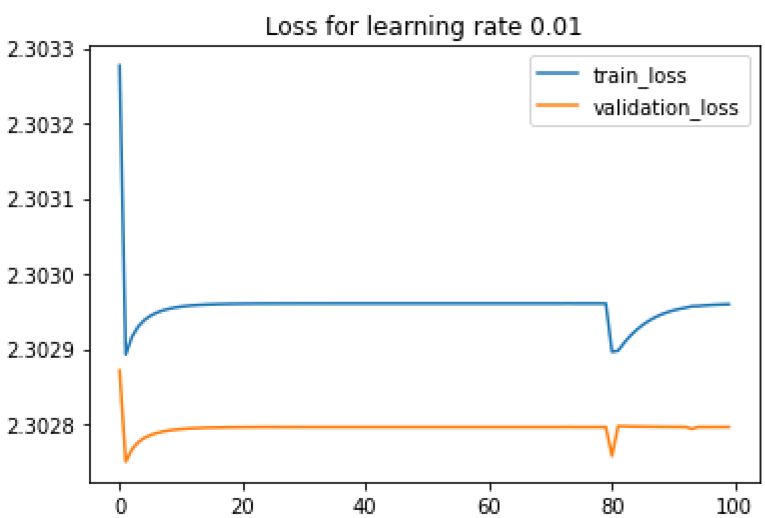
```
mlp_lrate(X_,Y_,x_val_,y_val_,100,x_test,y_test)
```




```
Tranning Accuracy Score 0.9218627450980392
Validation Accuracy Score 0.8585555555555555
testing Accuracy Score 0.8552
```




```
Tranning Accuracy Score 0.936
Validation Accuracy Score 0.8714444444444445
testing Accuracy Score 0.8718
```




```
Tranning Accuracy Score 0.09998039215686275
Validation Accuracy Score 0.10011111111111111
testing Accuracy Score 0.1
```

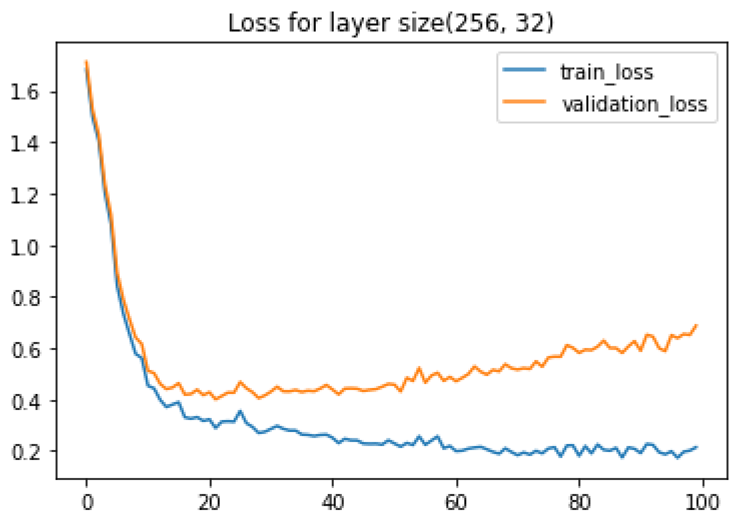
Learning rate determines the step size at each iteration while moving toward a minimum of a loss function. The best learning rate is **0.001** Because if we go slower than 0.001 we do not reach the minima in 100 iterations if we use a bigger step size it will make the model converge too quickly to a suboptimal solution. The same ca be verified with the above graphs when we use 0.0001 the accuracy is 0.92 but when we use 0.001 the accuracy increased.When we used 0.01 the accuracy decresed to 0.09

In [18]:

```
def mlp_layer(X,Y,X_val,Y_val,epochs,x_test,y_test):
    layers=[(256,32),(128,16),(64,8)]

    for i in layers:
        loss=[]
        v_loss=[]
        mlp = MLPClassifier(hidden_layer_sizes=i, activation='relu', solver='adam', ran
        for e in range(epochs):
            mlp.partial_fit(X.reshape(51000,784), Y, classes=np.unique(Y))

            loss.append(log_loss(Y,mlp.predict_proba(X.reshape(51000,784))))

            v_loss.append(log_loss(Y_val,mlp.predict_proba(X_val.reshape(9000,784))))
        plt.plot(range(epochs),loss,label='train_loss')
        plt.plot(range(epochs),v_loss,label='validation_loss')
        plt.legend()
        plt.title('Loss for layer size'+str(i))
        plt.show()
        print("Tranning Accuracy Score",metrics.accuracy_score(Y, mlp.predict(X.reshape
        print("Validation Accuracy Score",metrics.accuracy_score(Y_val, mlp.predict(X_v
        print("testing Accuracy Score",metrics.accuracy_score(y_test,mlp.predict(x_test
```

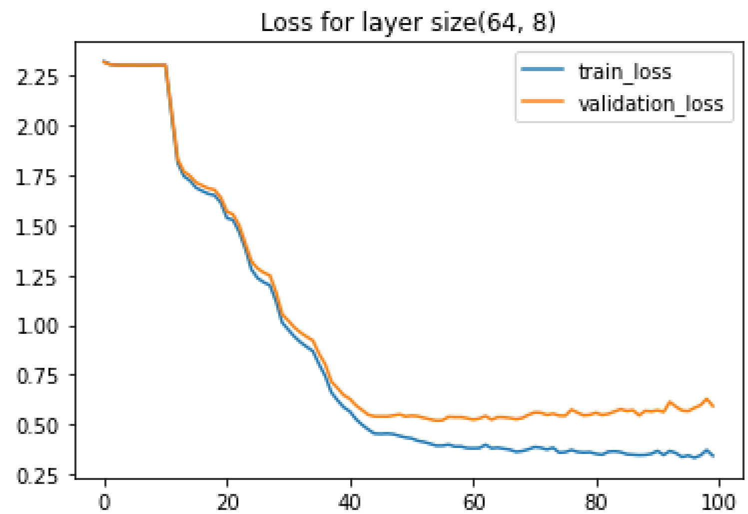
In [19]:

```
mlp_layer(X_,Y_,x_val_,y_val_,100,x_test,y_test)
```




```
Tranning Accuracy Score 0.936
Validation Accuracy Score 0.8714444444444445
testing Accuracy Score 0.8718
```


```
Tranning Accuracy Score 0.9113921568627451
Validation Accuracy Score 0.8598888888888889
testing Accuracy Score 0.8556
```




```
Tranning Accuracy Score 0.8836666666666667
Validation Accuracy Score 0.8464444444444444
testing Accuracy Score 0.8376
```

Using too few neurons in the hidden layers will result in underfitting. Underfitting occurs when there are too few neurons in the hidden layers to adequately detect the signals in a complicated data set. Out dataset have images (each of 28X28)which is complex dataset (with 784 feature values for each entry) and reducing the hidden layer size results in underfitting because there is a loss of data. The best accuracy is given by **(256,32)**

In [ ]:

```python
X, x_val, Y, y_val = train_test_split(x_train, y_train, test_size=0.8, random_state=42)
X.shape
```

Out[ ]:

```
(12000, 28, 28)
```

In [ ]:

```python
from sklearn.model_selection import GridSearchCV


grid = {
        'max_iter':[30,40,50],
        'activation': ['relu','tanh'],
```

```
        'hidden_layer_sizes':[(256,32),(64,8)],
        'alpha':[0.001,0.01]

        }
mlp=MLPClassifier()
# print(mlp.get_params().keys())
clf_cv = GridSearchCV(mlp, grid, n_jobs=1, cv=5)

clf_cv.fit(X.reshape(12000, 784),Y)
```

Out[ ]:

```
GridSearchCV(cv=5, estimator=MLPClassifier(), n_jobs=1,
             param_grid={'activation': ['relu', 'tanh'], 'alpha': [0.001, 0.01],
                         'hidden_layer_sizes': [(256, 32), (64, 8)],
                         'max_iter': [30, 40, 50]})
```

In [ ]:

```
print("GridSearch():\n")
combinations = 1
for x in grid.values():
    combinations *= len(x)
print('number of combinations',combinations)
print("Configuration ",clf_cv.best_params_)
print("Accuracy CV:",clf_cv.best_score_)
ppn_cv = clf_cv.best_estimator_
print(ppn_cv)
```

```
GridSearch():

number of combinations 24
Configuration  {'activation': 'relu', 'alpha': 0.001, 'hidden_layer_sizes': (256, 32),
'max_iter': 30}
Accuracy CV: 0.7436666666666667
MLPClassifier(alpha=0.001, hidden_layer_sizes=(256, 32), max_iter=30)
```

The best MLP classifier is the one having all the best parameters together from the 3 steps above where we separately found out the best of each parameter. The MLP with **Relu** as the activation function,**0.001** as the step size and **(256,32)** as the hidden layer size comes out to be the best after grid search.

In [ ]: