1 Clasificación con MNIST

```
# Mariana Hernandez
# 150845

import torch.nn as nn

# Regresion logistica multinomial

class RegresionMultinomial(nn.Module):
        """docstring for RegresionMultinomial"""
        def __init__(self, input_size, num_classes):
                super(RegresionMultinomial, self).__init__()
                self.linear = nn.Linear(input_size, num_classes)

        # define como se propaga la informacion
        # equivale al algoritmo 2
        def forward(self, x):
                out = self.linear(x)
                return out

# Red neuronal de C capas escondidas con activaciones sigmoid

class RedNeuronal(nn.Module):
        """docstring for RedNeuronal"""
        def __init__(self, input_size, num_classes):

                # HEREDA LA FUNCION CONSTRUCTORA DE LA CLASE PADRE
                super(RedNeuronal, self).__init__()

                # CANTIDAD DE UNIDADES PARA CADA CAPA ESCONDIDA
                hidden_size1 =  500

                # CREA LA ESTRUCTURA DE LA RED CON LAS FUNCIONES
                # DE ACTIVACION CORRESPONDIENTE
                self.fc1 = nn.Linear(input_size,hidden_size1)
                self.sigmoid = nn.Sigmoid()
                self.fc2 = nn.Linear(hidden_size1,num_classes)

        def forward(self, x):
                out = self.fc1(x)
                out = self.sigmoid(out)
                out = self.fc2(out)
                return out

class RedNeuronal3(nn.Module):
```

```
        """docstring for RedNeuronal3"""
        def __init__(self, input_size, num_classes):

                # HEREDA LA FUNCION CONSTRUCTORA DE LA CLASE PADRE
                super(RedNeuronal3, self).__init__()

                # CANTIDAD DE UNIDADES PARA CADA CAPA ESCONDIDA
                hidden_size1 =  500
                hidden_size2 = 100
                hidden_size3 = 30

                # CREA LA ESTRUCTURA DE LA RED CON LAS FUNCIONES
                # DE ACTIVACION CORRESPONDIENTE
                self.fc1 = nn.Linear(input_size,hidden_size1)
                self.sigmoid = nn.Sigmoid()
                self.fc2 = nn.Linear(hidden_size1,hidden_size2)
                self.sigmoid = nn.Sigmoid()
                self.fc3 = nn.Linear(hidden_size2, hidden_size3)
                self.sigmoid = nn.Sigmoid()
                self.fc4 = nn.Linear(hidden_size3,num_classes)

        def forward(self, x):
                out = self.fc1(x)
                out = self.sigmoid(out)
                out = self.fc2(out)
                out = self.sigmoid(out)
                out = self.fc3(out)
                out = self.sigmoid(out)
                out = self.fc4(out)
                return out

class RedNeuronal5(nn.Module):
        """docstring for RedNeuronal5"""
        def __init__(self, input_size, num_classes):

                # HEREDA LA FUNCION CONSTRUCTORA DE LA CLASE PADRE
                super(RedNeuronal5, self).__init__()

                # CANTIDAD DE UNIDADES PARA CADA CAPA ESCONDIDA
                hidden_size1 =  500
                hidden_size2 = 300
                hidden_size3 = 100
                hidden_size4 = 50
                hidden_size5 = 30

                # CREA LA ESTRUCTURA DE LA RED CON LAS FUNCIONES
```

```
                # DE ACTIVACION CORRESPONDIENTE
                self.fc1 = nn.Linear(input_size,hidden_size1)
                self.sigmoid = nn.Sigmoid()
                self.fc2 = nn.Linear(hidden_size1,hidden_size2)
                self.sigmoid = nn.Sigmoid()
                self.fc3 = nn.Linear(hidden_size2, hidden_size3)
                self.sigmoid = nn.Sigmoid()
                self.fc4 = nn.Linear(hidden_size3,hidden_size4)
                self.sigmoid = nn.Sigmoid()
                self.fc5 = nn.Linear(hidden_size4,hidden_size5)
                self.sigmoid = nn.Sigmoid()
                self.fc6 = nn.Linear(hidden_size5,num_classes)

        def forward(self, x):
                out = self.fc1(x)
                out = self.sigmoid(out)
                out = self.fc2(out)
                out = self.sigmoid(out)
                out = self.fc3(out)
                out = self.sigmoid(out)
                out = self.fc4(out)
                out = self.sigmoid(out)
                out = self.fc5(out)
                out = self.sigmoid(out)
                out = self.fc6(out)
                return out
```

Código para probar modelos

```
# Mariana Hernandez
# 150845

import torch
import torch.nn as nn
import torchvision
from torchvision.datasets import MNIST
from torch.utils.data import DataLoader
from torchvision.transforms import ToTensor
from modelos import RegresionMultinomial
from modelos import RedNeuronal
from modelos import RedNeuronal3
from modelos import RedNeuronal5
from torch.utils.tensorboard import SummaryWriter
import matplotlib.pyplot as plt
```

```
# SI LA COMPUTADORA TIENE CUDA, DAR ACCESO A CUDA
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hiper-parametros
input_size = 784 # dim de datos de entrada 28x28
num_classes = 10 # MNIST tiene 10 clases(numeros del 1 al 10)
num_epochs = 5 # numero de epocas para entrenar
bs = 100 # tamanio de lote
lr = 0.001 # tasa de aprendizaje

# DIVIDIMOS EL CONJUNTO DE DATOS EN DATOS DE ENTRENAMIENTO Y DATOS DE PRUEBA
# ACCEDE A LOS DATOS DE LA CARPETA './datos' Y LOS METE A UN TENSOR
root = './datos' # path
train_data = MNIST(root, train = True, transform=ToTensor(), download=True)
test_data = MNIST(root, train = False, transform=ToTensor())

# GENERA CONJUNTOS ITERABLES A PARTIR DE LOS DATOS
train_loader = DataLoader(train_data, batch_size=bs, shuffle=True)
test_loader = DataLoader(test_data, batch_size=bs, shuffle=False)

# definimos modelo

# INSTANCIAMOS EL MODELO Y LA FUNCION DE PERDIDA
model = RedNeuronal5(input_size, num_classes).to(device)
loss_function = nn.CrossEntropyLoss()

# optimizacion

# IMPLEMENTA DESCENSO DEL GRADIENTE ESTOCASTICO PARA OPTIMIZAR
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

# entrenamiento del modelo
for epoch in range(num_epochs):
        for i, (xi, yi) in enumerate(train_loader):

                # PREPARA LOS DATOS PARA LA CLASIFICACIONE EN EL
                # DISPOSITIVO QUE LE CORRESPONDA (CUDA O CPU)
                xi = xi.reshape(-1, 28*28).to(device) # imagenes
                yi = yi.to(device) # etiquetas

                # propagacion para adelante
                output = model(xi)
                loss = loss_function(output, yi)

                # propagacion para atras y paso de optimizacion
                optimizer.zero_grad()
```

```
                loss.backward()
                optimizer.step()

                if (i+1)%100==0:
                        print ('Epoca: {}/{}, Paso: {}/{},
                        Perdida: {:.5f}'.format(epoch+1,num_epochs, i+1,
                        len(train_loader), loss.item()))

# Prueba del modelo
# Al probar, usamos torch.no_grad() porque SOLO NECESITAMOS EL GRADIENTE
# RESPECTO A LOS PARAMETROS DE LAS CAPAS Y NO RESPECTO A LOS PARAMETROS
# UTILIZADOS PARA INICIALIZARLAS

writer = SummaryWriter()

with torch.no_grad():
        correct = 0
        total = 0
        for xi, yi in test_loader:

                #  INDICA CON QUE DISPOSITIVO SE VAN A CLASIFICAR LOS DATOS
                xi = xi.reshape(-1, 28*28).to(device)
                yi = yi.to(device)

                # PREDICE LA CLASIFICACION DE LOS DATOS
                output = model(xi)
                _, predicted = torch.max(output.data,1)

                total += yi.size(0)
                correct += (predicted==yi).sum().item()

        print(f'Precision del modelo en {total} imagenes: {100*correct/total}')

# GUARDAMOS EL MODELO
save_model = False
if save_model is True:
         torch.save(model.state_dict(), 'modelo.ckpt')
```