Testear inicialización por defecto y He en el dataset del MNIST con activación Relu

Objetivos

Testear inicializaciones por defecto y He

Tabla de contenido

Preparación

In [1]:
```
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

In [2]:
```
# Import the libraries we need to use in this lab

# Using the following line code to install the torchvision library
# !conda install -y torchvision

import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
import torch.nn.functional as F
import matplotlib.pylab as plt
import numpy as np

torch.manual_seed(0)
```

Out[2]:
```
<torch._C.Generator at 0x1ce4286f270>
```

Módulo red neuronal y función de entrenamiento

Definimos el módulo red neuoronal con inicialización He

In [3]:
```
# Define the class for neural network model with He Initialization

class Net_He(nn.Module):

    # Constructor
    def __init__(self, Layers):
        super(Net_He, self).__init__()
        self.hidden = nn.ModuleList()

        for input_size, output_size in zip(Layers, Layers[1:]):
            linear = nn.Linear(input_size, output_size)
            torch.nn.init.kaiming_uniform_(linear.weight, nonlinearity='relu')
            self.hidden.append(linear)

    # Prediction
    def forward(self, x):
        L = len(self.hidden)
        for (l, linear_transform) in zip(range(L), self.hidden):
            if l < L - 1:
                x = F.relu(linear_transform(x))
            else:
                x = linear_transform(x)
        return x
```

Definimos la red neuronal con inicialización uniforme

In [4]:
```
# Define the class for neural network model with Uniform Initialization

class Net_Uniform(nn.Module):

    # Constructor
    def __init__(self, Layers):
        super(Net_Uniform, self).__init__()
        self.hidden = nn.ModuleList()

        for input_size, output_size in zip(Layers, Layers[1:]):
            linear = nn.Linear(input_size,output_size)
            linear.weight.data.uniform_(0, 1)
            self.hidden.append(linear)

    # Prediction
    def forward(self, x):
        L = len(self.hidden)
        for (l, linear_transform) in zip(range(L), self.hidden):
            if l < L - 1:
                x = F.relu(linear_transform(x))
            else:
                x = linear_transform(x)

        return x
```

Red neuronal con inicialización por defecto de PyTorch

In [5]:
```
# Define the class for neural network model with PyTorch Default Initialization

class Net(nn.Module):

    # Constructor
    def __init__(self, Layers):
        super(Net, self).__init__()
        self.hidden = nn.ModuleList()

        for input_size, output_size in zip(Layers, Layers[1:]):
            linear = nn.Linear(input_size, output_size)
            self.hidden.append(linear)

    def forward(self, x):
        L=len(self.hidden)
        for (l, linear_transform) in zip(range(L), self.hidden):
            if l < L - 1:
                x = F.relu(linear_transform(x))
            else:
                x = linear_transform(x)

        return x
```

Definimos una función para entrenar el modelo; retorna un diccionario para almacenar la pérdida de entrenamiento y precisión sobre los datos de validación.

In [6]:
```
# Define function to  train model

def train(model, criterion, train_loader, validation_loader, optimizer, epochs = 100)
    i = 0
    loss_accuracy = {'training_loss': [], 'validation_accuracy': []}

    #n_epochs
    for epoch in range(epochs):
        for i, (x, y) in enumerate(train_loader):
            optimizer.zero_grad()
            z = model(x.view(-1, 28 * 28))
            loss = criterion(z, y)
            loss.backward()
            optimizer.step()
            loss_accuracy['training_loss'].append(loss.data.item())

        correct = 0
        for x, y in validation_loader:
            yhat = model(x.view(-1, 28 * 28))
            _, label = torch.max(yhat, 1)
            correct += (label == y).sum().item()
        accuracy = 100 * (correct / len(validation_dataset))
        loss_accuracy['validation_accuracy'].append(accuracy)

    return loss_accuracy
```

Creamos algunos datos

Cargamos el dataset de entrenamiento:

In [7]:
```
# Create the training dataset

train_dataset = dsets.MNIST(root='/data/', train=True, download=True, transform=trans
```

Cargamos el dataset de validación:

In [8]:
```
# Create the validation dataset

validation_dataset = dsets.MNIST(root='/data/', train=False, download=True, transform=
```

Creamos los cargadores de datos:

In [9]:
```
# Create the data loader for training and validation

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=2000, shu
validation_loader = torch.utils.data.DataLoader(dataset=validation_dataset, batch_size
```

Definimos la red neuronal, costo, optimizador y entrenamos el modelo

Creamos la función de criterio:

In [10]:
```
# Create the criterion function

criterion = nn.CrossEntropyLoss()
```

Creamos una lista que contiene el tamaño de la capa:

In [11]:
```
# Create the parameters

input_dim = 28 * 28
output_dim = 10
layers = [input_dim, 100, 200, 100, output_dim]
```

Tesetamos los distintos tipos de inicialización

Entrenamos la red usando la inicialización por defecto de PyTorch:

In [12]:
```
# Train the model with the default initialization

model = Net(layers)
learning_rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
training_results = train(model, criterion, train_loader,validation_loader, optimizer,
```

Entrenamos la red usando la inicialización He:

In [13]:
```
# Train the model with the He initialization

model_He = Net_He(layers)
optimizer = torch.optim.SGD(model_He.parameters(), lr=learning_rate)
training_results_He = train(model_He, criterion, train_loader, validation_loader, opt
```

Entrenamos la red usando la inicialización uniforme:

In [14]:
```
# Train the model with the Uniform initialization

model_Uniform = Net_Uniform(layers)
optimizer = torch.optim.SGD(model_Uniform.parameters(), lr=learning_rate)
training_results_Uniform = train(model_Uniform, criterion, train_loader, validation_l
```

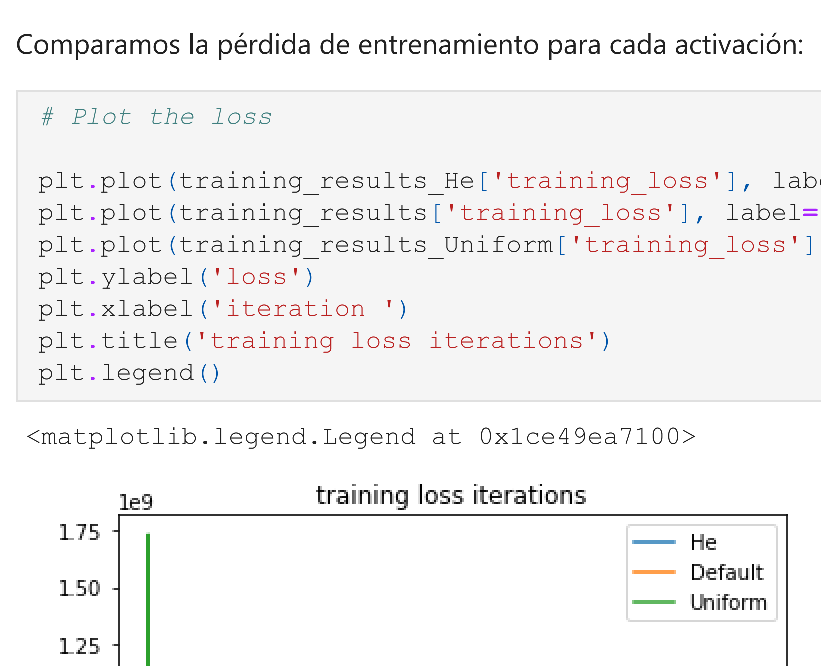
Análisis de resultados

Comparamos la pérdida de entrenamiento para cada activación:

In [18]:
```
# Plot the loss

plt.plot(training_results_He['training_loss'], label='He')
plt.plot(training_results['training_loss'], label='Default')
plt.plot(training_results_Uniform['training_loss'], label='Uniform')
plt.ylabel('loss')
plt.xlabel('iteration ')
plt.title('training loss iterations')
plt.legend()
```

Out[18]:
```
<matplotlib.legend.Legend at 0x1ce49ea7100>
```



Comparamos la precisión de validación para cada modelo:

In [17]:
```
# Plot the accuracy

plt.plot(training_results_He['validation_accuracy'], label='He')
plt.plot(training_results['validation_accuracy'], label='Default')
plt.plot(training_results_Uniform['validation_accuracy'], label='Uniform')
plt.ylabel('validation accuracy')
plt.xlabel('epochs ')
plt.legend()
plt.show()
```