Testear inicializaciones por defecto y de Xavier sobre el dataset del MNIST para la activación tanh

Objetivos

Testear diferentes métodos de inicialización

Tabla de contenido

Preparación

In [19]:
```python
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

In [2]:
```python
# 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 matplotlib.pylab as plt
import numpy as np

torch.manual_seed(0)
```

Out[2]:
```
<torch._C.Generator at 0x1bb5a4b9270>
```

Módulo red neuronal y función de entrenamiento

Definimos el módulo red neuronal con inicialización de Xavier:

In [3]:
```python
# Define the neural network with Xavier initialization

class Net_Xavier(nn.Module):

    # Constructor
    def __init__(self, Layers):
        super(Net_Xavier, 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.xavier_uniform_(linear.weight)
            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 = torch.tanh(linear_transform(x))
            else:
                x = linear_transform(x)
        return x
```

Definimos el módulo red neuronal con inicialización uniforme:

In [4]:
```python
# Define the neural network 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 = torch.tanh(linear_transform(x))
            else:
                x = linear_transform(x)
        return x
```

Definimos el módulo red neuronal con la inicialización por defecto de PyTorch:

In [5]:
```python
# Define the neural network with 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)

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

Definimos la función para entrenar el modelo, en este caso retorna un diccionario para almacenar la pérdida de entrenamiento y la precisión sobre los datos de validación.

In [6]:
```python
# function to Train the model

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

    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
```

Crear algunos datos

Cargamos el dataset de entrenamiento:

In [7]:
```python
# Create the train dataset

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

Cargamos el dataset de validación:

In [9]:
```python
# Create the validation dataset

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

Creamos los cargadores de datos:

In [10]:
```python
# Create Dataloader for both train dataset and validation dataset

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

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

Costo:

In [11]:
```python
# Define criterion function

criterion = nn.CrossEntropyLoss()
```

Creamos el modelo con 100 capas ocultas:

In [12]:
```python
# Set the parameters

input_dim = 28 * 28
output_dim = 10
layers = [input_dim, 100, 10, 100, 10, 100, output_dim]
epochs = 15
```

Testear los distintos tipos de inicialización

Entrenamos la red con la inicialización por defecto de PyTorch:

In [13]:
```python
# Train the model with 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 el modelo usando inicialización de Xavier:

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

model_Xavier = Net_Xavier(layers)
optimizer = torch.optim.SGD(model_Xavier.parameters(), lr=learning_rate)
training_results_Xavier = train(model_Xavier, criterion, train_loader, validation_loa
```

Entrenamos el modelo usando inicialización uniforme:

In [15]:
```python
# Train the model with 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
```

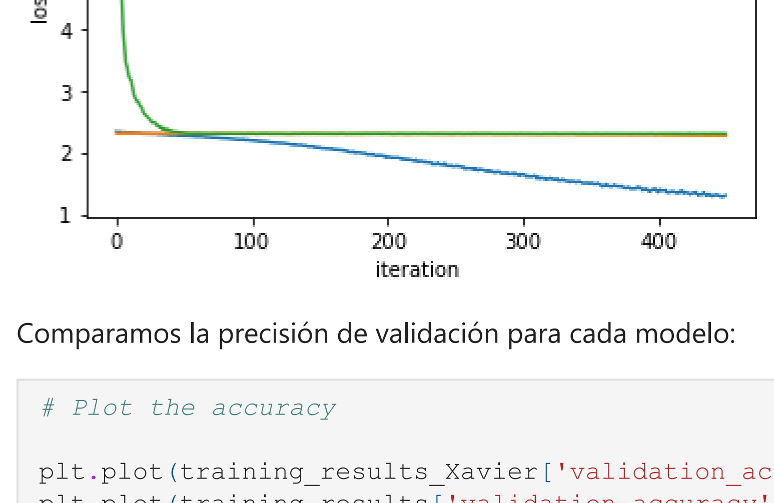
Análisis de resultados

Comparamos la pérdida de entrenamiento para cada inicialización:

In [16]:
```python
# Plot the loss

plt.plot(training_results_Xavier['training_loss'], label='Xavier')
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[16]:
```
<matplotlib.legend.Legend at 0x1bb600d5280>
```



Comparamos la precisión de validación para cada modelo:

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

plt.plot(training_results_Xavier['validation_accuracy'], label='Xavier')
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()
```

Out[17]:
```
<matplotlib.legend.Legend at 0x1bb60b4e340>
```