Batch Normalization con el dataset del MNIST

Objetivos

Definir varias redes neuronales, función de criterio y optimizador

Entrenar la red neuronal con y sin normalización por lotes

Tabla de contenido

Preparación

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

In [2]:
```python
# These are the libraries will be used for 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 0x202d66ad270>
```

Módulo red neuronal y función de entrenamiento

Módulo red neuronal con 2 capas ocultas usando normalización por lotes

In [3]:
```python
# Define the Neural Network Model using Batch Normalization

class NetBatchNorm(nn.Module):

    # Constructor
    def __init__(self, in_size, n_hidden1, n_hidden2, out_size):
        super(NetBatchNorm, self).__init__()
        self.linear1 = nn.Linear(in_size, n_hidden1)
        self.linear2 = nn.Linear(n_hidden1, n_hidden2)
        self.linear3 = nn.Linear(n_hidden2, out_size)
        self.bn1 = nn.BatchNorm1d(n_hidden1)
        self.bn2 = nn.BatchNorm1d(n_hidden2)

    # Prediction
    def forward(self, x):
        x = self.bn1(torch.sigmoid(self.linear1(x)))
        x = self.bn2(torch.sigmoid(self.linear2(x)))
        x = self.linear3(x)
        return x

    # Activations, to analyze results
    def activation(self, x):
        out = []
        z1 = self.bn1(self.linear1(x))
        out.append(z1.detach().numpy().reshape(-1))
        a1 = torch.sigmoid(z1)
        out.append(a1.detach().numpy().reshape(-1).reshape(-1))
        z2 = self.bn2(self.linear2(a1))
        out.append(z2.detach().numpy().reshape(-1))
        a2 = torch.sigmoid(z2)
        out.append(a2.detach().numpy().reshape(-1))
        return out
```

Módulo red neuronal con 2 capas ocultas sin normalización por lotes

In [4]:
```python
# Class Net for Neural Network Model

class Net(nn.Module):

    # Constructor
    def __init__(self, in_size, n_hidden1, n_hidden2, out_size):

        super(Net, self).__init__()
        self.linear1 = nn.Linear(in_size, n_hidden1)
        self.linear2 = nn.Linear(n_hidden1, n_hidden2)
        self.linear3 = nn.Linear(n_hidden2, out_size)

    # Prediction
    def forward(self, x):
        x = torch.sigmoid(self.linear1(x))
        x = torch.sigmoid(self.linear2(x))
        x = self.linear3(x)
        return x

    # Activations, to analyze results
    def activation(self, x):
        out = []
        z1 = self.linear1(x)
        out.append(z1.detach().numpy().reshape(-1))
        a1 = torch.sigmoid(z1)
        out.append(a1.detach().numpy().reshape(-1).reshape(-1))
        z2 = self.linear2(a1)
        out.append(z2.detach().numpy().reshape(-1))
        a2 = torch.sigmoid(z2)
        out.append(a2.detach().numpy().reshape(-1))
        return out
```

Función para entrenar el modelo. Devuelve un diccionario Python para almacenar la pérdida de entrenamiento y precisión sobre los datos de validación.

In [5]:
```python
# Define the function to train model

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

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

        correct = 0
        for x, y in validation_loader:
            model.eval()
            yhat = model(x.view(-1, 28 * 28))
            _, label = torch.max(yhat, 1)
            correct += (label == y).sum().item()

        accuracy = 100 * (correct / len(validation_dataset))
        useful_stuff['validation_accuracy'].append(accuracy)

    return useful_stuff
```

Crear algunos datos

Cargamos el dataset de entrenamiento:

In [6]:
```python
# load the train dataset

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

Cargamos el dataset de validación:

In [8]:
```python
# load the train dataset

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

Creamos los cargadores de datos

In [9]:
```python
# Create Data Loader for both train and validating

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, función de criterio y optimizador

Creamos la función de criterio:

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

criterion = nn.CrossEntropyLoss()
```

Variables para la red neuronal:

In [11]:
```python
# Set the parameters

input_dim = 28 * 28
hidden_dim = 100
output_dim = 10
```

Entrenamos la red neuronal con y sin normalización por lotes

Usando normalización por lotes:

In [12]:
```python
# Create model, optimizer and train the model

model_norm  = NetBatchNorm(input_dim, hidden_dim, hidden_dim, output_dim)
optimizer = torch.optim.Adam(model_norm.parameters(), lr = 0.1)
training_results_Norm=train(model_norm , criterion, train_loader, validation_loader, o
```

Sin normalización por lotes:

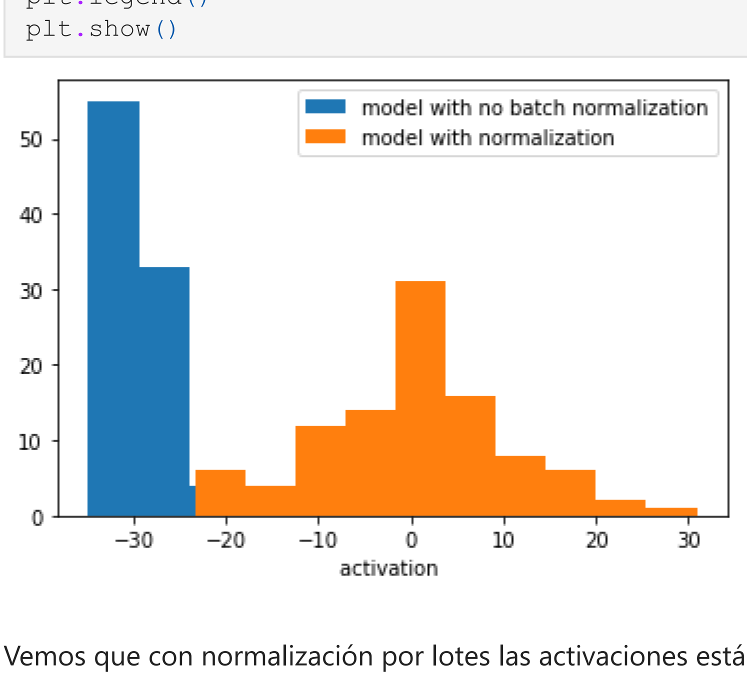
In [13]:
```python
# Create model without Batch Normalization, optimizer and train the model

model = Net(input_dim, hidden_dim, hidden_dim, output_dim)
optimizer = torch.optim.Adam(model.parameters(), lr = 0.1)
training_results = train(model, criterion, train_loader, validation_loader, optimizer,
```

Análisis de resultados

Comparamos los histogramas de la activación para la primer capa de la primer muestra para ambos modelos.

In [14]:
```python
model.eval()
model_norm.eval()
out=model.activation(validation_dataset[0][0].reshape(-1,28*28))
plt.hist(out[2],label='model with no batch normalization' )
out_norm=model_norm.activation(validation_dataset[0][0].reshape(-1,28*28))
plt.hist(out_norm[2],label='model with normalization')
plt.xlabel("activation ")
plt.legend()
plt.show()
```

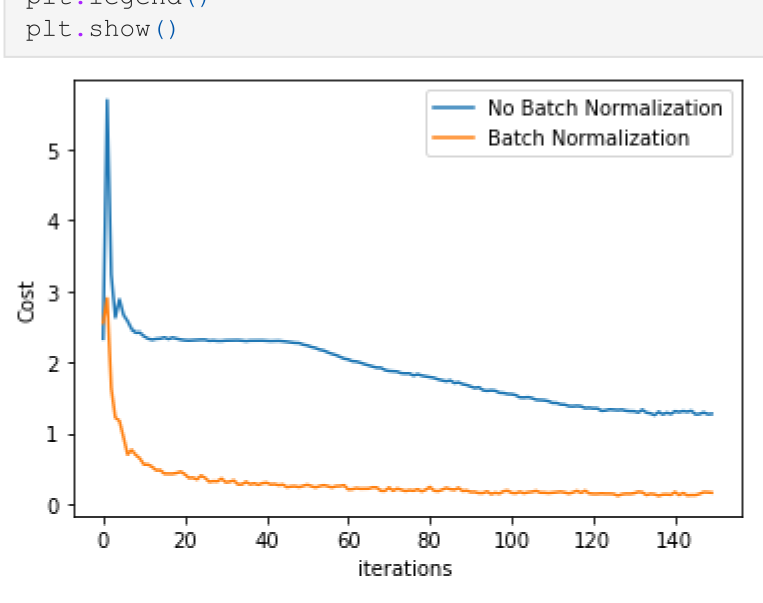


Vemos que con normalización por lotes las activaciones están centradas en 0 y tienen una menor varianza.

Comparamos la pérdida de entrenamiento para cada iteración:

In [15]:
```python
# Plot the diagram to show the loss

plt.plot(training_results['training_loss'], label='No Batch Normalization')
plt.plot(training_results_Norm['training_loss'], label='Batch Normalization')
plt.ylabel('Cost')
plt.xlabel('iterations ')
plt.legend()
plt.show()
```



Comparamos la precisión de validación para cada iteración:

In [16]:
```python
# Plot the diagram to show the accuracy

plt.plot(training_results['validation_accuracy'],label='No Batch Normalization')
plt.plot(training_results_Norm['validation_accuracy'],label='Batch Normalization')
plt.ylabel('validation accuracy')
plt.xlabel('epochs ')
plt.legend()
plt.show()
```