TC 5033

Deep Learning

Fully Connected Deep Neural Networks using PyTorch

José Antonio Cantoral Ceballos, Ph.D.

Team Members:

- A01200230 - Armando Bringas Corpus

Activity 2a: Implementing a FC for ASL Dataset using PyTorch

- Objective

  The primary aim of this activity is to transition from using Numpy for network implementation to utilizing PyTorch, a powerful deep learning framework. You will be replicating the work you did for the ASL dataset in Activity 1b, but this time, you'll implement a your multi layer FC model using PyTorch.

- Instructions

  Review Previous Work: Begin by reviewing your Numpy-based Fully Connected Network for the ASL dataset from Activity 1b. Note the architecture, hyperparameters, and performance metrics for comparison.

Introduce PyTorch: If you're new to PyTorch, take some time to familiarize yourself with its basic operations and syntax. You can consult the official documentation or follow online tutorials.

Prepare the ASL Dataset: As before, download and preprocess the Kaggle ASL dataset.

Implement the Network: Design your network architecture tailored for the ASL dataset. Pay special attention to PyTorch modules like nn.Linear() and nn.ReLU().

Train the Model: Implement the training loop, making use of PyTorch's autograd to handle backpropagation. Monitor metrics like loss and accuracy as the model trains.

Analyze and Document: In Markdown cells, discuss the architecture choices, any differences in performance between the Numpy and PyTorch implementations, and insights gained from using a deep learning framework like PyTorch.

In [1]:
```
import numpy as np
import string
import pandas as pd
import matplotlib.pyplot as plt
import os
%matplotlib inline

#PyTorch stuff
import torch
import torch.nn as nn
import torch.nn.functional as F

# Solamente para usuarios de Jupyter Themes
from jupyterthemes import jtplot
jtplot.style(grid=False)
```

In [2]:
```
# Check torch version
torch.__version__
```

Out[2]:
```
'2.1.0'
```

In [3]:
```
torch.cuda.is_available()
```

Out[3]:
```
True
```

In [4]:
```
if torch.cuda.is_available():
    print(torch.cuda.get_device_name(0))
    print(torch.cuda.get_device_capability(0))
    print(torch.cuda.get_device_properties(0))
else:
    print("No GPU available")
```

```
NVIDIA GeForce GTX 1650
(7, 5)
_CudaDeviceProperties(name='NVIDIA GeForce GTX 1650', major=7, minor=5, total_memory
=4095MB, multi_processor_count=14)
```

In [5]:
```python
DATA_PATH = 'data/asl_data/'
train_df = pd.read_csv(os.path.join(DATA_PATH, 'sign_mnist_train.csv'))
valid_df = pd.read_csv(os.path.join(DATA_PATH, 'sign_mnist_valid.csv'))
```

Always a good idea to explore the data

In [6]:
```python
train_df.head()
```

Out[6]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775
0	3	107	118	127	134	139	143	146	150	153	...	207
1	6	155	157	156	156	156	157	156	158	158	...	69
2	2	187	188	188	187	187	186	187	188	187	...	202
3	2	211	211	212	212	211	210	211	210	210	...	235
4	12	164	167	170	172	176	179	180	184	185	...	92

5 rows × 785 columns

Get training label data

In [7]:
```python
y_train = np.array(train_df['label'])
y_val = np.array(valid_df['label'])
del train_df['label']
del valid_df['label']
x_train = train_df.values.astype(np.float32)
x_val = valid_df.values.astype(np.float32)
```

In [8]:
```python
print(x_train.shape)
print(y_train.shape)
```

```
(27455, 784)
(27455,)
```

In [9]:
```python
print(x_val.shape, y_val.shape)
```

```
(7172, 784) (7172,)
```

In [10]:
```python
def split_val_test(x, y, pct=0.5, shuffle=True):
    assert x.shape[0] == y.shape[0], 'Number of samples x!= number samples y'
    total_samples = x.shape[0]
    if shuffle:
        idxs = np.arange(x.shape[0])
        np.random.shuffle(idxs)
        x = x[idxs]
        y = y[idxs]
        #return x_val, y_val, x_test, y_test
#         return x[:total_samples//2, :], y[:total_samples//2], x[total_samples//2:
    return x[:int(total_samples*pct), :], y[:int(total_samples*pct)], x[int(total_s
```

In [11]:
```
x_val, y_val, x_test, y_test = split_val_test(x_val, y_val)
```

In [12]:
```
type(y_val)
```

Out[12]:
```
numpy.ndarray
```

In [13]:
```
print(x_val.shape, y_val.shape)
print(x_test.shape, y_test.shape)
```

```
(3586, 784) (3586,)
(3586, 784) (3586,)
```

In [14]:
```
alphabet=list(string.ascii_lowercase)
alphabet.remove('j')
alphabet.remove('z')
print(len(alphabet))
```

```
24
```

Normalise the data

In [15]:
```
def normalise(x_mean, x_std, x_data):
    return (x_data - x_mean) / x_std
```

In [16]:
```
x_mean = x_train.mean()
x_std = x_train.std()

x_train = normalise(x_mean, x_std, x_train)
x_val = normalise(x_mean, x_std, x_val)
x_test = normalise(x_mean, x_std, x_test)
```

In [17]:
```
x_train.mean(), x_train.std()
```

Out[17]:
```
(3.6268384e-06, 0.99999946)
```

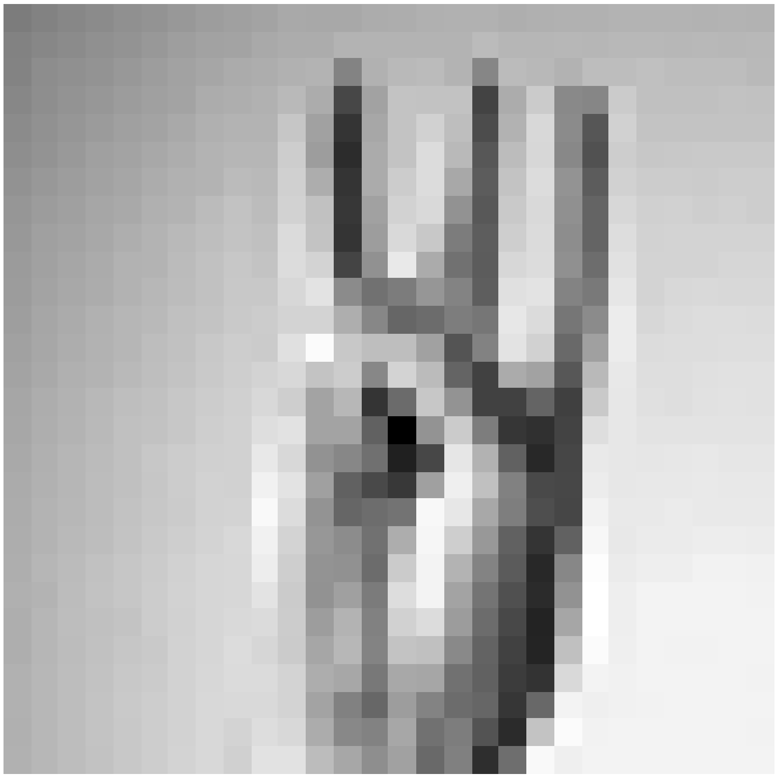
In [18]:
```
def plot_number(image):
    plt.figure(figsize=(5,5))
    plt.imshow(image.squeeze(), cmap=plt.get_cmap('gray'))
    plt.axis('off')
    plt.show()
```

In [19]:
```
type(x_val)
```

Out[19]:
```
numpy.ndarray
```

In [20]:
```
rnd_idx = np.random.randint(len(y_val))
# print(rnd_idx)
# print(y_val[rnd_idx])
print(f'The sampled image represents a: {alphabet[y_val[rnd_idx]]}')
plot_number(x_val[rnd_idx].reshape(28,28))
```

```
The sampled image represents a: w
```



The model

$$z^1 = W^1X + b^1$$

$$a^1 = ReLU(z^1)$$

$$z^2 = W^2a^1 + b^2$$

$$\hat{y} = \frac{e^{z^2 k}}{\sum_j e^{z_j}}$$

$$\mathcal{L}(\hat{y}^i, y^i) = -y^i \ln(\hat{y}^i) = -\ln(\hat{y}^i)$$

$$\mathcal{J}(w,b) = \frac{1}{num\_samples} \sum_{i=1}^{num\_samples} -\ln(\hat{y}^i)$$

Create minibatches

In [21]:
```
def create_minibatches(mb_size, x, y, shuffle = True):
    '''
    x  #muestras, 784
    y #muestras, 1
    '''
    assert x.shape[0] == y.shape[0], 'Error en cantidad de muestras'
    total_data = x.shape[0]
    if shuffle:
```

```
        idxs = np.arange(total_data)
        np.random.shuffle(idxs)
        x = x[idxs]
        y = y[idxs]

    return ((x[i:i+mb_size], y[i:i+mb_size]) for i in range(0, total_data, mb_size))
```

In [22]:

```
for i, (x, y) in enumerate(create_minibatches(128,x_train, y_train)):
    print(i)
```

Now the PyTorch part

In [23]:

```
x_train_tensor = torch.tensor(x_train.copy())
y_train_tensor = torch.tensor(y_train.copy())

x_val_tensor = torch.tensor(x_val.copy())
y_val_tensor = torch.tensor(y_val.copy())
```

```
x_test_tensor = torch.tensor(x_test.copy())
y_test_tensor = torch.tensor(y_test.copy())
```

In [24]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
```

cuda

Accuracy

In [25]:
```
def accuracy(model, x, y, mb_size):
    num_correct = 0
    num_total = 0
    cost = 0.
    model.eval()
    model = model.to(device=device)
    with torch.no_grad():
        for mb, (xi, yi) in enumerate(create_minibatches(mb_size, x, y),1):
            xi = xi.to(device=device, dtype = torch.float32)
            yi = yi.to(device=device, dtype = torch.long)
            scores = model(xi) # mb_size, 10
            cost += (F.cross_entropy(scores, yi)).item()
            _, pred = scores.max(dim=1) #pred shape (mb_size )
            num_correct += (pred == yi.squeeze()).sum() # pred shape (mb_size), yi
            num_total += pred.size(0)

        return cost/mb, float(num_correct)/num_total
```

Training Loop

In [26]:
```
def train(model, optimiser, mb_size, epochs=100):
    model = model.to(device=device)
    train_cost = 0.
    val_cost = 0.
    for epoch in range(epochs):
        train_correct_num  = 0.
        train_total = 0.
        train_cost_acum = 0
        for mb, (xi, yi) in enumerate(create_minibatches(mb_size, x_train_tensor, y
            model.train()
            xi = xi.to(device=device, dtype=torch.float32)
            yi = yi.to(device=device, dtype=torch.long)
            scores = model(xi)
            # funcion cost
            cost = F.cross_entropy(input= scores, target=yi.squeeze())
            optimiser.zero_grad()
            cost.backward()
            optimiser.step()

            train_correct_num += (torch.argmax(scores, dim=1) == yi.squeeze()).sum(
            train_total += scores.size(0)

            train_cost_acum += cost.item()
```

```
        val_cost, val_acc = accuracy(model, x_val_tensor, y_val_tensor, mb_size)
        train_acc = float(train_correct_num)/train_total
        train_cost = train_cost_acum/mb
        if epoch%20 == 0:
            print(f'Epoch:{epoch}, train cost: {train_cost:.6f}, val cost: {val_cos
                  f' train acc: {train_acc:.4f}, val acc: {val_acc:4f},'
                  f' lr: {optimiser.param_groups[0]["lr"]:.6f}')
```

Model using Sequential

Changing model hyperparameters to verify if accuracy can be improved

In [27]:
```
hidden = 300
lr = 1e-3
epochs = 100
mb_size = 256
```

In [28]:
```
model1 = nn.Sequential(nn.Linear(in_features=784, out_features=hidden),
                       nn.Dropout(),
                       nn.ReLU(),
#                        nn.Linear(in_features=hidden1, out_features=hidden), nn.Re
                       nn.Linear(in_features=hidden, out_features=24))
# optimiser = torch.optim.SGD(model1.parameters(), lr=lr, momentum=0.9, weight_deca
optimiser = torch.optim.Adam(model1.parameters(), lr=lr, weight_decay=1e-4)
scheduler = torch.optim.lr_scheduler.OneCycleLR(optimiser, 0.1, epochs=epochs, step

train(model1, optimiser, mb_size, epochs)
```

```
Epoch:0, train cost: 0.861163, val cost: 0.696276, train acc: 0.7290, val acc: 0.781
372, lr: 0.004000
Epoch:20, train cost: 0.110764, val cost: 1.814120, train acc: 0.9760, val acc: 0.80
2844, lr: 0.004000
Epoch:40, train cost: 0.103240, val cost: 2.020840, train acc: 0.9790, val acc: 0.80
1729, lr: 0.004000
Epoch:60, train cost: 0.090984, val cost: 2.568904, train acc: 0.9816, val acc: 0.79
5594, lr: 0.004000
Epoch:80, train cost: 0.163161, val cost: 2.652196, train acc: 0.9732, val acc: 0.80
3681, lr: 0.004000
```

In [29]:
```
accuracy(model1, x_test_tensor, y_test_tensor, mb_size)[1]
```

Out[29]:
```
0.8218070273284998
```

In [30]:
```
def predict(x, model):
    x = x.to(device=device, dtype = torch.float32)
    scores = model(x) # mb_size, 10
    _, pred = scores.max(dim=1) #pred shape (mb_size )
    return pred
```

In [31]:
```
rnd_idx = np.random.randint(len(y_test))
print(f'The sampled image represents a: {alphabet[y_test[rnd_idx]]}')
plot_number(x_test[rnd_idx].reshape(28,28))
```

```
pred=predict(x_test_tensor[rnd_idx].reshape(1, -1), model1)
print(f'The predicted value is: {alphabet[pred]}')
```

```
The sampled image represents a: h
```

```
The predicted value is: h
```

```
In [ ]:
```