Tecnológico de Monterrey

TC 5033

Deep Learning

Fully Connected Deep Neural Networks

José Antonio Cantoral Ceballos, Ph.D.

Team Members:

- A01200230 - Armando Bringas Corpus

Activity 1b: Implementing a Fully Connected Network for Kaggle ASL Dataset

- Objective

The aim of this part of the activity is to apply your understanding of Fully Connected Networks by implementing a multilayer network for the Kaggle ASL (American Sign Language) dataset. While you have been provided with a complete solution for a Fully Connected Network using Numpy for the MNIST dataset, you are encouraged to try to come up with the solution.

- Instructions

  This activity requires submission in teams of 3 or 4 members. Submissions from smaller or larger teams will not be accepted unless prior approval has been granted (only due to exceptional circumstances). While teamwork is encouraged, each member is expected to contribute individually to the assignment. The final submission should feature the best arguments and solutions from each team member. Only one person per team needs to submit the completed work, but it is imperative that the names of all team members are listed in a Markdown cell at the very beginning of the notebook (either the first or second cell). Failure to include all team member names will result in the grade being awarded solely to the individual who submitted the assignment, with zero points given to other team members (no exceptions will be made to this rule).

Load and Preprocess Data: You are provided a starter code to load the data. Be sure to understand the code.

Review MNIST Notebook (Optional): Before diving into this activity, you have the option to revisit the MNIST example to refresh your understanding of how to build a Fully Connected Network using Numpy.

Start Fresh: Although you can refer to the MNIST solution at any point, try to implement the network for the ASL dataset on your own. This will reinforce your learning and understanding of the architecture and mathematics involved.

Implement Forward and Backward Pass: Write the code to perform the forward and backward passes, keeping in mind the specific challenges and characteristics of the ASL dataset.

Design the Network: Create the architecture of the Fully Connected Network tailored for the ASL dataset. Choose the number of hidden layers, neurons, and hyperparameters judiciously.

Train the Model: Execute the training loop, ensuring to track performance metrics such as loss and accuracy.

Analyze and Document: Use Markdown cells to document in detail the choices you made in terms of architecture and hyperparameters, you may use figures, equations, etc to aid in your explanations. Include any metrics that help justify these choices and discuss the model's performance.

- Evaluation Criteria
    - Code Readability and Comments
    - Appropriateness of chosen architecture and hyperparameters for the ASL dataset
    - Performance of the model on the ASL dataset (at least 70% acc)
    - Quality of Markdown documentation
- Submission

Submit this Jupyter Notebook in canvas with your complete solution, ensuring your code is well-commented and includes Markdown cells that explain your design choices, results, and any challenges you encountered.

Import Libraries

In [1]:
```
import numpy as np
import string
import pandas as pd
import matplotlib.pyplot as plt
import cv2 as cv
import os

%load_ext autoreload
%autoreload 2
#################################
%matplotlib inline
```

In [2]:
```
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'))
```

In [3]:
```
train_df.head()
```

Out[3]:

	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

Load Data

In [4]:
```
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)
```

Train and validation data

In [5]:
```
def split_val_test(x, y, pct=0.5, shuffle=True):
    '''
    Create a function that will allow you to split the previously loaded validation
    into validation and test.
    '''
    # verify that x_val and y_val have the same length
    assert len(x) == len(y)

    # total of data
```

```
    total_data = len(x)

    # shuffle data if it is required
    if shuffle:
        indices = np.arange(len(x))
        np.random.shuffle(indices)
        x, y = map(lambda data: np.array([data[i] for i in indices]), [x, y])

    # calculate split index
    split_idx = int(pct * len(x))

    return x[:split_idx], y[:split_idx].reshape(-1,1), x[split_idx:], y[split_idx:]
```

In [6]:
```
x_val, y_val, x_test, y_test = split_val_test(x_val, y_val)
```

Inspect shape of the splitted data

In [7]:
```
print(x_train.shape)
print(y_train.shape)

print(x_val.shape)
print(y_val.shape)

print(x_test.shape)
print(y_test.shape)
```

```
(27455, 784)
(27455,)
(3586, 784)
(3586, 1)
(3586, 784)
(3586, 1)
```

In [8]:
```
### The following

alphabet=list(string.ascii_lowercase)
alphabet.remove('j')
alphabet.remove('z')
print(len(alphabet))
```

```
24
```

Normalise

Lambda function that applies feature scaling to the dataset x_data using the mean x_mean and standard deviation x_std. Feature scaling is a method used to standardize the range of independent variables or features of data.

In [9]:
```
normalise = lambda x_mean, x_std, x_data: (x_data - x_mean) / x_std

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 [10]:
```
x_train.mean(), x_train.std()
```

Out[10]:
```
(3.6268384e-06, 0.99999946)
```

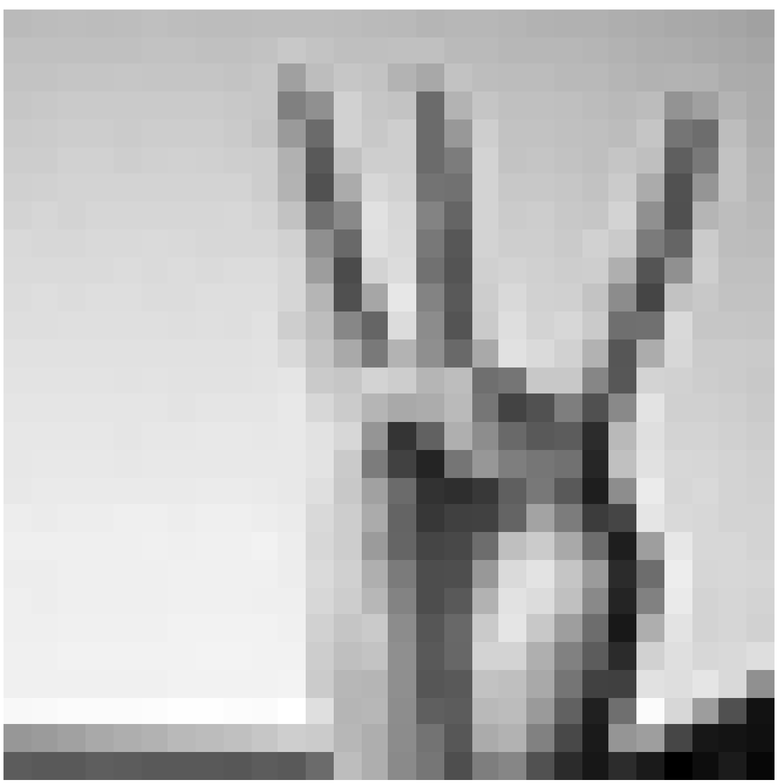
Plot Samples

In [11]:
```
def plot_number(image):
    plt.figure(figsize=(5,5))
    plt.imshow(image.squeeze(), cmap=plt.get_cmap('gray'))
    plt.axis('off')
    plt.show()
```

Plot a random sample

In [12]:
```
rnd_idx = np.random.randint(len(y_test))
image_data = x_test[rnd_idx].reshape(28, 28)
plot_number(image_data)

print(f'The sampled image represents a: {alphabet[y_test[rnd_idx][0]]}')
```



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

Equations for Arquitecture and Loss Function of a ReLU-Activated Neural Network Model

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

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

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

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

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

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

Additional Functions

Mini batches

In [13]:
```
#Function to create training minibatches
def create_minibatches(mb_size, x, y, shuffle = True):
    assert x.shape[0] == y.shape[0], 'Error en cantidad de muestras'
    total_data = x.shape[0]
    # Shuffle the dataset if the shuffle parameter is True
    if shuffle:
        idxs = np.arange(total_data)
        np.random.shuffle(idxs)
        # Shuffled indices to reorder the input features and labels
        x = x[idxs]
        y = y[idxs]
    # Generate minibatches
    return ((x[i:i+mb_size], y[i:i+mb_size]) for i in range(0, total_data, mb_size))
```

Linear, ReLU and Sequential classes

In [14]:
```
class np_tensor(np.ndarray): pass
```

Linear class

In [15]:
```
class Linear():
    def __init__(self, input_size, output_size):
        '''
        Init parameters utilizando Kaiming He
        '''
        # Initialize the weights using Kaiming He initialization for better perform
        self.W = (np.random.randn(output_size, input_size) / np.sqrt(input_size/2))
        # Initialize biases with zeros
        self.b = (np.zeros((output_size, 1))).view(np_tensor)
    def __call__(self, X):
        # Forward pass linear transformation
        Z = self.W @ X + self.b
        return Z
```

```
    def backward(self, X, Z):
        # Get the gradient with respect to the input
        X.grad = self.W.T @ Z.grad
        # Get the gradient with respect to the weights
        self.W.grad = Z.grad @ X.T
        # Get the gradient with respect to the biases
        self.b.grad = np.sum(Z.grad, axis = 1, keepdims=True)
```

ReLU class

In [16]:
```
class ReLU():
    def __call__(self, Z):
        # ReLU activation that replaces negative values in Z with 0
        return np.maximum(0, Z)
    def backward(self, Z, A):
        # Copy the gradient from the next layer
        Z.grad = A.grad.copy()
        # Zero the gradient where the function is not activated
        Z.grad[Z <= 0] = 0
        Z.grad[Z <= 0] = 0
```

Sequential class

In [17]:
```
class Sequential_layers():
    def __init__(self, layers):
        '''
        layers - lista que contiene objetos de tipo Linear, ReLU
        '''
        self.layers = layers
        self.x = None
        self.outputs = {}
    def __call__(self, X):
        self.x = X
        self.outputs['l0'] = self.x
        # Forward pass through each layer
        for i, layer in enumerate(self.layers, 1):
            self.x = layer(self.x)
            self.outputs['l'+str(i)]=self.x
        # Return the output for backpropagation
        return self.x
    def backward(self):
        for i in reversed(range(len(self.layers))):
            self.layers[i].backward(self.outputs['l'+str(i)], self.outputs['l'+str(
    def update(self, learning_rate = 1e-3):
        for layer in self.layers:
            if isinstance(layer, ReLU): continue
            # Update weights and biases with gradient descent
            layer.W = layer.W - learning_rate * layer.W.grad
            layer.b = layer.b - learning_rate * layer.b.grad
    def predict(self, X):
        # Forward pass and return the index of the max value
        return np.argmax(self.__call__(X))
```

Cost Function

In [18]:
```
def softmaxXEntropy(x, y):
    batch_size = x.shape[1]

    # Compute the exponential scores for numerical stability in softmax
    # Compute the probabilities for each class by normalizing the exponential score
    exp_scores = np.exp(x)
    probs = exp_scores / exp_scores.sum(axis = 0)
    preds = probs.copy()

    # Compute the cross-entropy cost
    y_hat = probs[y.squeeze(), np.arange(batch_size)]
    cost = np.sum(-np.log(y_hat)) / batch_size

    # Calculate gradients for backpropagation
    probs[y.squeeze(), np.arange(batch_size)] -= 1 #dl/dx
    x.grad = probs.copy()

    return preds, cost
```

Training Function

In [19]:
```
#Function for training the model
def train(model, epochs, mb_size=128, learning_rate = 1e-3):
    # Iterate over each epoch
    for epoch in range(epochs):
        for i, (x, y) in enumerate(create_minibatches(mb_size, x_train, y_train)):
            # Perform a forward pass and compute score
            scores = model(x.T.view(np_tensor))
            # Calculate the cost and gradients with respect score
            _, cost = softmaxXEntropy(scores, y)
            # Perform backward pass and then update learning rate
            model.backward()
            model.update(learning_rate)

        print(f'epochs: {epoch+1}\t cost: {cost:.4f} \t accuracy: {accuracy(x_val,
```

Accuracy Function

In [20]:
```
#Function to calculate the model accuracy
def accuracy(x, y, mb_size, model):
    correct = 0
    total = 0
    # Iterate over dataset in minibatches
    for i, (x, y) in enumerate(create_minibatches(mb_size, x, y)):
        # Perform forward pass to get predictions from the model
        pred = model(x.T.view(np_tensor))
        # Count how many predictions match with the true labels and get the total
        correct += np.sum(np.argmax(pred, axis=0) == y.squeeze())
        total += pred.shape[1]
```

```python
    # Handle the case where total is zero to avoid division by zero
    if total == 0:
        return 0  # or return an appropriate value or message indicating no data wa

    return correct/total
```

Create your model and train it

Model Hyperparameters

In [21]:
```python
# Constants
BATCH_SIZE = 512
INPUT_DIM = x_train.shape[1]
OUTPUT_DIM = len(alphabet)
NEURONS = [300, 500, 700]
LEARNING_RATES = [1e-3, 5e-4, 1e-4]
EPOCHS = 20
```

Neural Network Architecture:

Input -> Linear -> ReLU -> Linear -> ReLU -> Linear -> ReLU -> Linear -> Output

This type of architecture is common in feedforward neural networks, where the goal is to transform the input data through successive layers of computation to make a prediction. ReLU activation functions are used to help combat the vanishing gradient problem, which allows the network to learn faster and perform better on a variety of tasks.

The intention is to test with different combinations of model hyperparameters to determine which performs best.

In [22]:
```python
# Store models' accuracy for different configurations
models_accuracy = {}

# Initialize the best model and its accuracy
best_model = None
best_accuracy = 0

# Train different model configurations
for lr in LEARNING_RATES:
    for neuron_count in NEURONS:
        print('\n' + '-' * 60)
        print(f'Training with Neurons: {neuron_count}, Learning Rate: {lr}')
        print('-' * 60)

        # Create the model
        model = Sequential_layers([
            Linear(INPUT_DIM, neuron_count),
            ReLU(),
            Linear(neuron_count, neuron_count),
            ReLU(),
            Linear(neuron_count, neuron_count),
            ReLU(),
```

```python
            Linear(neuron_count, OUTPUT_DIM)
        ])

        # Train the model
        trained_model = train(model, EPOCHS, BATCH_SIZE, lr)

        # Calculate accuracy on the test set
        test_acc = accuracy(x_test, y_test, BATCH_SIZE, model)
        print(f'\nAccuracy: {test_acc:.4f}')

        models_accuracy[(lr, neuron_count)] = test_acc

        # Update the best model if the current model is better
        if test_acc > best_accuracy:
            best_accuracy = test_acc
            best_model = trained_model

# Print the best model's accuracy
print('\n' + '-' * 60)
print(f"\nBest Model Accuracy: {best_accuracy:.4f}")
```

```
------------------------------------------------------------
Training with Neurons: 300, Learning Rate: 0.001
------------------------------------------------------------
epochs: 1        cost: 0.2982    accuracy: 0.7214
epochs: 2        cost: 0.0145    accuracy: 0.7705
epochs: 3        cost: 0.0050    accuracy: 0.7889
epochs: 4        cost: 0.0041    accuracy: 0.7875
epochs: 5        cost: 0.0022    accuracy: 0.7922
epochs: 6        cost: 0.0015    accuracy: 0.7922
epochs: 7        cost: 0.0013    accuracy: 0.7895
epochs: 8        cost: 0.0013    accuracy: 0.7920
epochs: 9        cost: 0.0011    accuracy: 0.7945
epochs: 10       cost: 0.0012    accuracy: 0.7936
epochs: 11       cost: 0.0008    accuracy: 0.7942
epochs: 12       cost: 0.0006    accuracy: 0.7953
epochs: 13       cost: 0.0006    accuracy: 0.7956
epochs: 14       cost: 0.0006    accuracy: 0.7956
epochs: 15       cost: 0.0005    accuracy: 0.7953
epochs: 16       cost: 0.0005    accuracy: 0.7959
epochs: 17       cost: 0.0004    accuracy: 0.7959
epochs: 18       cost: 0.0005    accuracy: 0.7956
epochs: 19       cost: 0.0004    accuracy: 0.7945
epochs: 20       cost: 0.0003    accuracy: 0.7959

Accuracy: 0.8045

------------------------------------------------------------
Training with Neurons: 500, Learning Rate: 0.001
------------------------------------------------------------
epochs: 1        cost: 0.4810    accuracy: 0.6849
epochs: 2        cost: 0.0671    accuracy: 0.7741
epochs: 3        cost: 0.0092    accuracy: 0.7822
epochs: 4        cost: 0.0039    accuracy: 0.7847
epochs: 5        cost: 0.0029    accuracy: 0.7856
epochs: 6        cost: 0.0016    accuracy: 0.7847
epochs: 7        cost: 0.0013    accuracy: 0.7869
epochs: 8        cost: 0.0013    accuracy: 0.7867
epochs: 9        cost: 0.0010    accuracy: 0.7864
epochs: 10       cost: 0.0006    accuracy: 0.7858
epochs: 11       cost: 0.0006    accuracy: 0.7853
epochs: 12       cost: 0.0007    accuracy: 0.7853
epochs: 13       cost: 0.0006    accuracy: 0.7858
epochs: 14       cost: 0.0006    accuracy: 0.7867
epochs: 15       cost: 0.0005    accuracy: 0.7864
epochs: 16       cost: 0.0004    accuracy: 0.7875
epochs: 17       cost: 0.0004    accuracy: 0.7878
epochs: 18       cost: 0.0004    accuracy: 0.7867
epochs: 19       cost: 0.0004    accuracy: 0.7878
epochs: 20       cost: 0.0004    accuracy: 0.7878

Accuracy: 0.8067

------------------------------------------------------------
Training with Neurons: 700, Learning Rate: 0.001
------------------------------------------------------------
epochs: 1        cost: 0.7115    accuracy: 0.7094
```

```
epochs: 2        cost: 0.0135    accuracy: 0.7727
epochs: 3        cost: 0.0055    accuracy: 0.7727
epochs: 4        cost: 0.0031    accuracy: 0.7780
epochs: 5        cost: 0.0022    accuracy: 0.7780
epochs: 6        cost: 0.0014    accuracy: 0.7783
epochs: 7        cost: 0.0013    accuracy: 0.7794
epochs: 8        cost: 0.0009    accuracy: 0.7800
epochs: 9        cost: 0.0012    accuracy: 0.7803
epochs: 10       cost: 0.0009    accuracy: 0.7811
epochs: 11       cost: 0.0008    accuracy: 0.7814
epochs: 12       cost: 0.0006    accuracy: 0.7814
epochs: 13       cost: 0.0007    accuracy: 0.7814
epochs: 14       cost: 0.0006    accuracy: 0.7814
epochs: 15       cost: 0.0004    accuracy: 0.7817
epochs: 16       cost: 0.0005    accuracy: 0.7814
epochs: 17       cost: 0.0007    accuracy: 0.7817
epochs: 18       cost: 0.0004    accuracy: 0.7817
epochs: 19       cost: 0.0003    accuracy: 0.7819
epochs: 20       cost: 0.0003    accuracy: 0.7814

Accuracy: 0.7962

------------------------------------------------------------
Training with Neurons: 300, Learning Rate: 0.0005
------------------------------------------------------------
epochs: 1        cost: 0.2447    accuracy: 0.7724
epochs: 2        cost: 0.0252    accuracy: 0.7998
epochs: 3        cost: 0.0114    accuracy: 0.8026
epochs: 4        cost: 0.0073    accuracy: 0.8093
epochs: 5        cost: 0.0058    accuracy: 0.8109
epochs: 6        cost: 0.0048    accuracy: 0.8132
epochs: 7        cost: 0.0034    accuracy: 0.8146
epochs: 8        cost: 0.0026    accuracy: 0.8143
epochs: 9        cost: 0.0025    accuracy: 0.8143
epochs: 10       cost: 0.0019    accuracy: 0.8176
epochs: 11       cost: 0.0017    accuracy: 0.8171
epochs: 12       cost: 0.0016    accuracy: 0.8176
epochs: 13       cost: 0.0015    accuracy: 0.8171
epochs: 14       cost: 0.0017    accuracy: 0.8187
epochs: 15       cost: 0.0014    accuracy: 0.8193
epochs: 16       cost: 0.0011    accuracy: 0.8187
epochs: 17       cost: 0.0009    accuracy: 0.8185
epochs: 18       cost: 0.0012    accuracy: 0.8173
epochs: 19       cost: 0.0009    accuracy: 0.8196
epochs: 20       cost: 0.0008    accuracy: 0.8199

Accuracy: 0.8224

------------------------------------------------------------
Training with Neurons: 500, Learning Rate: 0.0005
------------------------------------------------------------
epochs: 1        cost: 0.1366    accuracy: 0.7504
epochs: 2        cost: 0.0171    accuracy: 0.7649
epochs: 3        cost: 0.0111    accuracy: 0.7638
epochs: 4        cost: 0.0065    accuracy: 0.7733
epochs: 5        cost: 0.0046    accuracy: 0.7683
```

```
epochs: 6        cost: 0.0044    accuracy: 0.7786
epochs: 7        cost: 0.0032    accuracy: 0.7775
epochs: 8        cost: 0.0030    accuracy: 0.7794
epochs: 9        cost: 0.0024    accuracy: 0.7811
epochs: 10       cost: 0.0021    accuracy: 0.7836
epochs: 11       cost: 0.0016    accuracy: 0.7805
epochs: 12       cost: 0.0013    accuracy: 0.7825
epochs: 13       cost: 0.0014    accuracy: 0.7858
epochs: 14       cost: 0.0011    accuracy: 0.7875
epochs: 15       cost: 0.0012    accuracy: 0.7872
epochs: 16       cost: 0.0012    accuracy: 0.7878
epochs: 17       cost: 0.0010    accuracy: 0.7883
epochs: 18       cost: 0.0011    accuracy: 0.7897
epochs: 19       cost: 0.0009    accuracy: 0.7878
epochs: 20       cost: 0.0009    accuracy: 0.7895

Accuracy: 0.7975

------------------------------------------------------------
Training with Neurons: 700, Learning Rate: 0.0005
------------------------------------------------------------
epochs: 1        cost: 0.1377    accuracy: 0.7323
epochs: 2        cost: 0.0222    accuracy: 0.7699
epochs: 3        cost: 0.0108    accuracy: 0.7833
epochs: 4        cost: 0.0074    accuracy: 0.7867
epochs: 5        cost: 0.0044    accuracy: 0.7917
epochs: 6        cost: 0.0039    accuracy: 0.7889
epochs: 7        cost: 0.0034    accuracy: 0.7903
epochs: 8        cost: 0.0024    accuracy: 0.7869
epochs: 9        cost: 0.0022    accuracy: 0.7934
epochs: 10       cost: 0.0016    accuracy: 0.7925
epochs: 11       cost: 0.0018    accuracy: 0.7925
epochs: 12       cost: 0.0015    accuracy: 0.7928
epochs: 13       cost: 0.0014    accuracy: 0.7934
epochs: 14       cost: 0.0013    accuracy: 0.7934
epochs: 15       cost: 0.0011    accuracy: 0.7962
epochs: 16       cost: 0.0011    accuracy: 0.7948
epochs: 17       cost: 0.0009    accuracy: 0.7942
epochs: 18       cost: 0.0010    accuracy: 0.7948
epochs: 19       cost: 0.0008    accuracy: 0.7934
epochs: 20       cost: 0.0007    accuracy: 0.7962

Accuracy: 0.7987

------------------------------------------------------------
Training with Neurons: 300, Learning Rate: 0.0001
------------------------------------------------------------
epochs: 1        cost: 0.8526    accuracy: 0.5945
epochs: 2        cost: 0.3229    accuracy: 0.6949
epochs: 3        cost: 0.1706    accuracy: 0.7170
epochs: 4        cost: 0.0836    accuracy: 0.7379
epochs: 5        cost: 0.0576    accuracy: 0.7443
epochs: 6        cost: 0.0407    accuracy: 0.7485
epochs: 7        cost: 0.0343    accuracy: 0.7532
epochs: 8        cost: 0.0250    accuracy: 0.7540
epochs: 9        cost: 0.0223    accuracy: 0.7538
```

```
epochs: 10       cost: 0.0195    accuracy: 0.7593
epochs: 11       cost: 0.0141    accuracy: 0.7607
epochs: 12       cost: 0.0127    accuracy: 0.7610
epochs: 13       cost: 0.0116    accuracy: 0.7635
epochs: 14       cost: 0.0097    accuracy: 0.7635
epochs: 15       cost: 0.0110    accuracy: 0.7644
epochs: 16       cost: 0.0082    accuracy: 0.7672
epochs: 17       cost: 0.0084    accuracy: 0.7674
epochs: 18       cost: 0.0078    accuracy: 0.7705
epochs: 19       cost: 0.0076    accuracy: 0.7674
epochs: 20       cost: 0.0065    accuracy: 0.7683

Accuracy: 0.7708

------------------------------------------------------------
Training with Neurons: 500, Learning Rate: 0.0001
------------------------------------------------------------
epochs: 1        cost: 0.7755    accuracy: 0.6316
epochs: 2        cost: 0.2999    accuracy: 0.7128
epochs: 3        cost: 0.1288    accuracy: 0.7451
epochs: 4        cost: 0.0819    accuracy: 0.7418
epochs: 5        cost: 0.0511    accuracy: 0.7518
epochs: 6        cost: 0.0348    accuracy: 0.7577
epochs: 7        cost: 0.0291    accuracy: 0.7560
epochs: 8        cost: 0.0199    accuracy: 0.7624
epochs: 9        cost: 0.0212    accuracy: 0.7613
epochs: 10       cost: 0.0150    accuracy: 0.7658
epochs: 11       cost: 0.0134    accuracy: 0.7638
epochs: 12       cost: 0.0110    accuracy: 0.7660
epochs: 13       cost: 0.0102    accuracy: 0.7694
epochs: 14       cost: 0.0108    accuracy: 0.7683
epochs: 15       cost: 0.0087    accuracy: 0.7677
epochs: 16       cost: 0.0071    accuracy: 0.7674
epochs: 17       cost: 0.0078    accuracy: 0.7727
epochs: 18       cost: 0.0068    accuracy: 0.7691
epochs: 19       cost: 0.0073    accuracy: 0.7691
epochs: 20       cost: 0.0070    accuracy: 0.7691

Accuracy: 0.7836

------------------------------------------------------------
Training with Neurons: 700, Learning Rate: 0.0001
------------------------------------------------------------
epochs: 1        cost: 0.6491    accuracy: 0.6729
epochs: 2        cost: 0.2533    accuracy: 0.7412
epochs: 3        cost: 0.1106    accuracy: 0.7571
epochs: 4        cost: 0.0618    accuracy: 0.7803
epochs: 5        cost: 0.0432    accuracy: 0.7800
epochs: 6        cost: 0.0364    accuracy: 0.7844
epochs: 7        cost: 0.0244    accuracy: 0.7797
epochs: 8        cost: 0.0226    accuracy: 0.7833
epochs: 9        cost: 0.0184    accuracy: 0.7856
epochs: 10       cost: 0.0151    accuracy: 0.7822
epochs: 11       cost: 0.0127    accuracy: 0.7833
epochs: 12       cost: 0.0095    accuracy: 0.7833
epochs: 13       cost: 0.0089    accuracy: 0.7828
```

```
epochs: 14      cost: 0.0087    accuracy: 0.7844
epochs: 15      cost: 0.0084    accuracy: 0.7853
epochs: 16      cost: 0.0081    accuracy: 0.7847
epochs: 17      cost: 0.0063    accuracy: 0.7853
epochs: 18      cost: 0.0061    accuracy: 0.7844
epochs: 19      cost: 0.0053    accuracy: 0.7864
epochs: 20      cost: 0.0058    accuracy: 0.7839

Accuracy: 0.7934

------------------------------------------------------------

Best Model Accuracy: 0.8224
```

Test your model on Random data from your test set

In [23]:

```
idx = np.random.randint(len(y_test))
plot_number(x_test[idx].reshape(28,28))
pred = model.predict(x_test[idx].reshape(-1, 1))

print(f'Predicted value is: {alphabet[pred]}, real value is:{alphabet[y_test[idx][0
```

```
Predicted value is: m, real value is:m
```

In [ ]: