Machine Learning
Sign Language Prediction

Group 13
Mahir Umut Aldemir – 2083294
Aura Orval – 2077579
Joël van Run – 2057225
Yingchen Ye – 2075659

Task 1 - Recognize Sign Language

Models used for predicting sign language

The models that have been chosen to make prediction for task 1 are the following: Logistic Regression (LR), and Convolutional Neural Networks (CNN).

Logistic Regression (LR)

LR has been chosen as the model transfers image features to the possibilities of each class in the form of logit(p) and applies stochastic gradient descent (SGD) to find the model parameters when the loss function (cross-entropy) is lowest.

Convolutional Neural Networks (CNN)

The second model this project focused on, is the convolutional neural network (*CNN). The rationale for applying this model, is that a convolutional neural network is conventionally and inherently a model that works excellent on image or visual data – and therefore, it seemed logical to implement such a model for the classification of sign language letters. Thus, to explain the model, it will be simplified in three main blocks – the convolution layers, dense & output layers and the chosen optimizer, loss function & metric.

1. <u>Convolution Layers:</u>

For the CNN model, it was decided to build the model with the "Keras Conv2D class" on three convolution layers. The reason for adding three convolution layers, is simply convention and to make the model more complex. Furthermore, the activation function "ReLU" is applied for the simple reason that the model is a non-linear model – and ReLU was chosen in a conventional manner. Additionally, each convolution layer is followed by a "MaxPool" pooling layer to reduce the spatial size. MaxPool has been chosen due to the rational that conventionally, MaxPool captures features better than other pooling layers.

2. <u>Dense & Output Layers:</u>

It was also decided to add a dense layer with 512 neurons/units. This number has been chosen in an ascending manner with respects to the filters in the convolution layers. Again, the ReLU activation function was chosen. Moreover, an output layer with 24 neurons/units was added for each of the 24 sign language letter labels/classes. Additionally, the "softmax" activation function was implemented in the output layer to convert the data into probabilities.

3. <u>Optimizer, Loss function & Metric:</u>

The model was compiled using the gradient descent-based optimizer "Adam". It is believed conventionally that Adam results in the best classification. Furthermore, due to the fact that the sign language classification task is a multi-class classification problem – it was appropriate to apply the "categorical crossentropy" loss function to the model. Lastly, the performance metric of "accuracy" was implemented to observe the model.

Input for Classifier

–Logistic Regression

For LR, feature transformation is used. The data comprise images and their respective labels. Each image is reshaped from one dimension of size 784 to a 3-dimensional array with the shape of (1, 28, 28) because each image has the size of 28*28 pixels. "Random split" from the torch library is used to randomly select 20.000 examples used in the training set. The remaining examples were used as validation. Feature engineering has not been applied because LR does not make any assumptions about the distribution of independent variables.

Input for Classifier

-CNN

The provided training dataset has been split into a training dataset, which consists of 75 percent of the initial provided training dataset – and into a validation dataset, which consists of 25 percent of the initial provided training dataset. This split has been conducted with the "train_test_split()" function. Furthermore, regarding feature engineering, feature transformation has been applied. All *x*-feature values of all sets have been reshaped with the "numpy.reshape()" function and all *Y*-label values of all sets have been binarized using the "LabelBinarizer()" function. This has been done to put the data in the appropriate shape for the model to work properly.

Hyperparameter tuning
-Logistic Regression
In the LR model, different batch sizes, learning rates, and epoch sizes were used. These were all chosen manually and are defined in Table 1.
Table 1 – Logistic Regression Hyperparameter Tuning

	Batch Size	Learning Rate	Epoch Size
Model 1	32	0.001	30
Model 2	64	0.001	30
Model 3	32	0.0001	30

For the same batch size of 32, the learning rate is higher (lr = 0.001) and reaches a higher accuracy faster than model 3 (lr = 0.0001). However, the reduction of loss is less stable than Model 3. With a controlled learning rate (lr = 0.001), Model 1 (batch size = 32) has a lower batch size and reaches high accuracy faster than Model 2 (batch size = 64), and is more stable on both the performance and accuracy during loss training as seen in Figure 1.

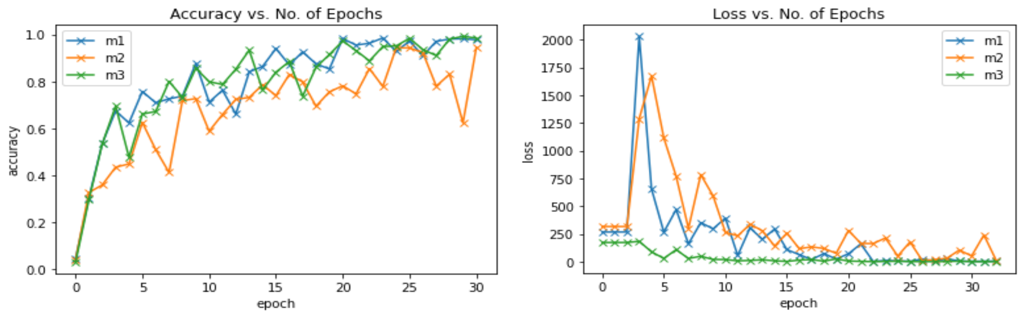


Figure 1 – Loss & Accuracy in Validation Set vs. Epoch for the LR model with different hyperparameters.

Hyperparameter tuning
-*CNN*
The provided training dataset has for the CNN model been split into a training dataset which consists of 75% of the initial provided dataset. The validation set includes 25% of the provided dataset. The split is conducted with the "train_test_split()" function. Feature transformation is applied. All x-feature values of all sets are binarized using the "LabelBinarizer()" function to put the data in an appropriate shape. Rather than choosing hyperparameters manually, the "RandomizedSearchCV() function was used instead of conducting a grid search to make is computationally more efficient. The pre-tuned model resulted in a 99% accuracy on the training set with the randomly chosen hyperparameters of batch size 64, with 35 epochs. The candidate values for batch size were between 10 and 100, with a step size of 10. The parameter distributions for the number of epochs were between 10 and 50 with a step size of 5. Consequently, after twelve hours of computing, the algorithm returned an optimal batch size of 25 and 36 numbers of epochs across all given parameter distributions. Yet again, this resulted in a 99% accuracy on the training-, and validation set.

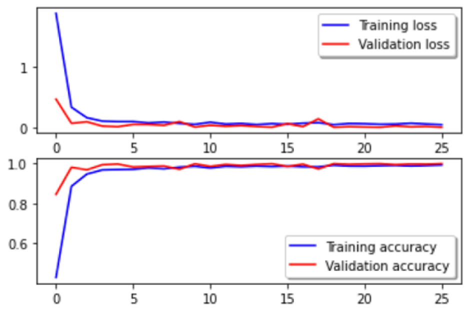


Figure 2 – CNN accuracy and loss

Model Chosen
-*LR*
Method or system built for classification
The training method for LR is SGD as the optimizer, and cross-entropy as the loss function. SGD is used due to its low computational cost and the efficiency which focusses on finding the global minimum. The loss function is required by SGD, and cross-entropy is the best loss function for multiclass logistic regression due to its simplicity and the ability to always find the global minimum.

Model Chosen
-*CNN*
The same model architecture described in "*Models used for predicting sign language"* is used, including the same number of layers, numbers of neurons, the activation functions per layer, et cetera. However, the tuned hyperparameters in the aforementioned section were implemented to train the model.

Results Report
-*LR*
The overall accuracy of the LR model and the loss in the validation set is 0.1967, and 0.9992 respectively. In the testing set it is 30.8357 and 0.6686 respectively, which indicated that the model is overfitting. The sensitivity per class in the testing set (with an 0-25 index is label 0-25, 9 and 25 are represented by NaN) as seen in Figure 3. According to the sensitivity per class, predicting in testing set, A(0) is the easiest one to predict while T(19) is the hardest.

A	1	I	0.649	Q	0.75	Y	0.502
B	0.859	J	nan	R	0.625	Z	nan
C	0.906	K	0.338	S	0.415		
D	0.837	L	0.9	T	0.254		
E	0.731	M	0.718	U	0.466		
F	0.903	N	0.361	V	0.364		
G	0.747	O	0.61	W	0.636		
H	0.711	P	0.965	X	0.502		

Table 2 – Sensitivity per class.

Results Report
-*CNN*

In the CNN model, the overall accuracy on the test set was 91,29%. This indicated that the model is quite accurate in predicting hand sign language letters. The classes are represented by labels [0:23]. According to the confusion matrix (Figure 4), label 19 which denoted the letter "U", had an accuracy of 62% and thus performed the worst. Labels 0, 2, 4, 5, 14, 15, and 21 which indicate the letters "A", "C", "E", "F", "P', "Q", and "W" performed best with 100% accuracy. The classification report in appendix B illustrates that the macro-average across precision, recall, and the F1-score was also 91% which was also the case for the weighted average. However, the weighted average for precision was 92%.

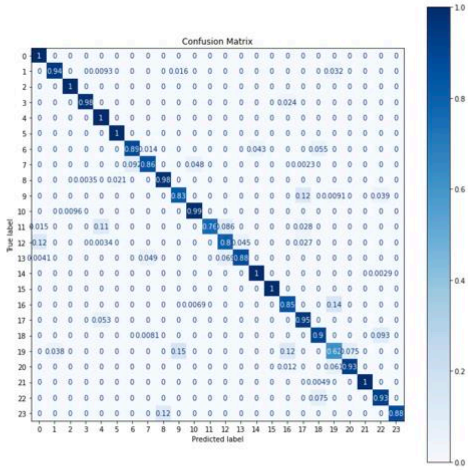


Figure 3 – CNN Confusion Matrix

Comparison
-*LR and CNN*
The overall accuracy when predicting in the test set for LR is ~67%. The training time for the third LR model is 30.9s for 30 epochs. The CNN produced an accuracy of 91% and it took around 1453 seconds for 26 epochs. It indicates that the CNN model performed significantly better than the LR model but the LR model was faster in training than the CNN model. By comparing Figure 3 and 4, The "U" scored better on the LR model than the CNN model. Besides this, the CNN model achieved 100% accuracy for seven letters, while the LR model only achieved 100% sensitivity for one letter, indicating CNN performs better according to the performance per class.

Task 2 - Identifying sign language

Strategy to detect and crop images
The model we used to detect and crop images were both models from Task 1 since our CNN model performed with an accuracy of 91%. Thus, we did not feel the need to use different models. We did focus more on the CNN model though, as its performance was higher than the performance of our LR model. The strategy we used for detecting and/or cropping images was to calculate the mode of each image with a pixel size of 28*200. Within that column we found the mode for which 200 is the background (noise). With "for" loops we deleted these columns for each image, and we split the rest into separate images with a column number of 28 and made predictions each split image. If it were the case that the column number was not a multiplication of 28, "if-else" statements were used to add columns for full "255" (white), or to delete the columns that are equal to the remainder. For the top 5 accuracy, we included the .csv file which is also uploaded with the submission of the report and .py file.

REFERENCES

References used for coding the models:

G. (2021, September 13). *CNN_Model*. Kaggle. Retrieved June 10, 2022, from https://www.kaggle.com/code/gaurav3435/cnn-model/notebook

Kurdekar, S. (2022, April 26). *RandomizedSearchCV to find Optimal Parameters in Python* -. ProjectPro. Retrieved June 10, 2022, from https://www.projectpro.io/recipes/find-optimal-parameters-using-randomizedsearchcv-for-regression

S. (2020, June 30). *sign-prediction(using pytorch)*. Kaggle. Retrieved June 10, 2022, from https://www.kaggle.com/code/sachinsom/sign-prediction-using-pytorch#Logistic-regression

S. (2020a, April 24). *Sign-Language Classification CNN (99.40% Accuracy)*. Kaggle. Retrieved June 10, 2022, from https://www.kaggle.com/code/sayakdasgupta/sign-language-classification-cnn-99-40-accuracy/notebook

APPENDIX A: Logistic Regression Model

In [ ]:
```
# resource:
## logistic regression: https://www.kaggle.com/code/sachinsom/sign-prediction-using-pytorch#Logistic-regression
```

In [206…
```
#! pip install torch
#! pip install torchvision
#! pip install seaborn
```

In [205…
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import metrics
```

In [209…
```
import torch
from torch.utils.data import TensorDataset, DataLoader, random_split

from PIL import Image
import pandas as pd

from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt

import torch.nn as nn
import torch.nn.functional as F

from torchvision.utils import make_grid

import jovian
```

In [210…
```
#load data, data is image represented by numpy array and labels is their relative labels
with np.load('train_data_label.npz') as data:
    train_data = data['train_data']
    train_labels = data['train_label']

with np.load('test_data_label.npz') as data:
    test_data = data['test_data']
    test_labels = data['test_label']
```

In [211…
```
#check data shape
print(train_data.shape, train_labels.shape, test_data.shape, test_labels.shape)
train_data = train_data.astype('float32')
test_data  = test_data.astype('float32')
```

```
(27455, 784) (27455,) (7172, 784) (7172,)
```

In [212…
```
#reshape the data, thus an image is represented by a 28 x 28 array
train_images_shaped = train_data.reshape(train_data.shape[0],1,28,28)
test_images_shaped = test_data.reshape(test_data.shape[0],1,28,28)
```

In [213…
```
#check shape
print(train_images_shaped.shape, train_labels.shape, test_images_shaped.shape, test_labels.shape)
```

```
(27455, 1, 28, 28) (27455,) (7172, 1, 28, 28) (7172,)
```

In [214…
```
#turn numpy array into pytorch tensors
train_images_tensors = torch.from_numpy(train_images_shaped)
train_labels_tensors = torch.from_numpy(train_labels)

test_images_tensors = torch.from_numpy(test_images_shaped)
test_labels_tensors = torch.from_numpy(test_labels)
```

In [115…
```
# pytorch dataset
#this dataset will further devided into validation dataset and training dataset
train_ds_full = TensorDataset(train_images_tensors, train_labels_tensors)
#for prediction
test_ds = TensorDataset(test_images_tensors, test_labels_tensors)
```

In [215…
```
# each image is converted to a (1, 28, 28)
# The first dimension is for the number of channels.
# The second and third dimensions are for the size of the image, in this case, 28px by 28px.

img, label = train_ds_full[0]
print(img.shape, label)
img.type()
```

```
torch.Size([1, 28, 28]) tensor(3)
```

Out [215…

```
'torch.FloatTensor'
```

In [216…

```
# Hyperparameters
batch_size = 64
learning_rate = 0.001

# Other constants
in_channels = 1
input_size = in_channels * 28 * 28
num_classes = 26
```

In [158…

```
# split dataset to training, validation, testing sets
random_seed = 11
torch.manual_seed(random_seed);
```

In [217…

```
val_size = 7455
train_size = len(train_ds_full) - val_size

train_ds, val_ds = random_split(train_ds_full, [train_size, val_size,])
len(train_ds), len(val_ds), len(test_ds)
```

Out [217…

```
(20000, 7455, 7172)
```

In [218…

```
train_dl = DataLoader(train_ds, batch_size, shuffle=True, num_workers=4, pin_memory=True)
val_dl = DataLoader(val_ds, batch_size*2, num_workers=4, pin_memory=True)
test_dl = DataLoader(test_ds, batch_size*2, num_workers=4, pin_memory=True)
```

In [219…

```
for img, label in train_dl:
    print(img.size())
    break
```

```
torch.Size([64, 1, 28, 28])
```

In [220…

```
# Logistic Regression Model
class ASLModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, xb):
        xb = xb.reshape(-1, in_channels*28*28)
        out = self.linear(xb)
        return out

    def training_step(self, batch):
        images, labels = batch
        out = self(images)                  # Generate predictions
        loss = F.cross_entropy(out, labels) # Calculate loss
        return loss

    def validation_step(self, batch):
        images, labels = batch
        out = self(images)                    # Generate predictions
        loss = F.cross_entropy(out, labels)   # Calculate loss
        acc = accuracy(out, labels)           # Calculate accuracy
        return {'val_loss': loss.detach(), 'val_acc': acc.detach()}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()   # Combine losses
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()      # Combine accuracies
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], val_loss: {:.4f}, val_acc: {:.4f}".format(epoch, result['val_loss'], result['val_acc']))

model = ASLModel()
```

In [221…

```
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, 1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))
def matrics(outputs, labels):      # confusion matrices/not used
    _, preds = torch.max(outputs, 1)
    return metrics.confusion_matrix(labels, preds)
def evaluate(model, val_loader):
```

```
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)
```

In [ ]:

```
#attempt 1 (can't calculate accuracy)
for images, labels in test_dl:
    outputs = model(images)
    print(labels)
    print(matrics(outputs, labels))

    break

plt.figure(figsize=(50,50))
sns.heatmap(matrics(outputs, labels), annot=True, fmt=".3f", linewidths=.5, square = True, cmap = 'Blues_r');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(accuracy(outputs, labels))
plt.title(all_sample_title, size = 5);
```

In [222…

```
#attempt 2, the accuracy function seems not right

def accuracy_per_class(outputs, labels):
    confusion_matrix = torch.zeros(num_classes, num_classes)
    with torch.no_grad():
        for images, labels in test_dl:
            outputs = model(images)
            _, preds = torch.max(outputs, 1)
            for t, p in zip(labels.view(-1), preds.view(-1)):
                    confusion_matrix[t.long(), p.long()] += 1
    return (confusion_matrix.diag()/confusion_matrix.sum(1))
accuracy_per_class(outputs, labels)
```

Out[222…

```
tensor([0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           nan, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.6579, 0.0000, 0.6068, 0.0000, 0.0000,    nan])
```

In [223…

```
for images, labels in test_dl:
    outputs = model(images)
    print(labels)
    print(accuracy(outputs, labels))


    break

print('outputs.shape : ', outputs.shape)
print('Sample outputs :\n', outputs[:2].data)
```

```
tensor([ 6,  5, 10,  0,  3, 21, 10, 14,  3,  7,  8,  8, 21, 12,  7,  4, 22,  0,
         7,  7,  2,  0, 21,  4, 10, 15,  2, 15,  7,  1,  7,  8, 13, 19,  3, 21,
        13,  3, 18, 14, 15, 23,  8, 15, 14,  5, 17,  4, 19, 13, 20, 22, 20,  5,
        16, 16, 21,  4,  7, 22, 10, 13, 11, 22,  2, 10,  1,  4, 18,  4, 20,  6,
        15,  4,  3, 20, 15, 11,  2,  2, 17,  2,  7, 21, 23,  7, 12, 17, 24, 14,
         2,  1,  7, 23,  8,  5,  0,  0, 19, 21,  8,  4,  2, 20, 16,  1, 15, 14,
         2,  6, 12,  5,  0, 24,  2, 19, 14, 24, 16, 10,  4,  8,  8, 12, 12,  8,
         6, 21])
tensor(0.0469)
outputs.shape :  torch.Size([128, 26])
Sample outputs :
 tensor([[  19.9887,  -12.8110,  -72.3961,   37.6670,  -52.9996,  -93.5604,
          -54.8120, -102.7464,  -86.5440,    5.8669,   22.6612, -238.6040,
           48.8837,  -80.4912,  -49.6546,  -53.8310,   14.1044,  -74.3636,
           87.4094,    3.9108,  105.2022,   -7.5323,  120.2522,   25.9618,
          -57.5934,   19.2525],
        [  25.8676,   -3.5260,  -58.0672,   23.1839,  -17.0213, -116.5799,
          -20.3643, -141.2641,  -51.8904,  -10.2210,   30.1070, -273.9760,
           19.8409,  -74.3655,  -12.8379,  -81.8716,  -33.8949,  -38.5402,
           91.5073,  -43.0517,  181.9059,  -16.3331,  160.5794,   62.5223,
          -58.4056,  -18.2539]])
```

In [224…

```
def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim.SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        # Training Phase
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation phase
        result = evaluate(model, val_loader)
        model.epoch_end(epoch, result)
        history.append(result)
    return history
```

In [225…
```
result0 = evaluate(model, val_dl)
result0
```

Out[225…
```
{'val_loss': 214.5165252685547, 'val_acc': 0.050351966617245674}
```

In [226…
```
# The initial accuracy is around 4%,
# Let's train for 10 epochs for 4 times (total 40 epochs) and look at the results.
history1 = fit(10, 0.001, model, train_dl, val_dl)
```

```
Epoch [0], val_loss: 2887.5164, val_acc: 0.1782
Epoch [1], val_loss: 1217.7384, val_acc: 0.3234
Epoch [2], val_loss: 1511.0090, val_acc: 0.3689
Epoch [3], val_loss: 863.5461, val_acc: 0.4502
Epoch [4], val_loss: 674.6116, val_acc: 0.5418
Epoch [5], val_loss: 571.4136, val_acc: 0.6226
Epoch [6], val_loss: 305.9597, val_acc: 0.5474
Epoch [7], val_loss: 802.7087, val_acc: 0.5576
Epoch [8], val_loss: 74.9344, val_acc: 0.8125
Epoch [9], val_loss: 530.8260, val_acc: 0.5896
```

In [227…
```
history2 = fit(10, 0.0001, model, train_dl, val_dl)
```

```
Epoch [0], val_loss: 27.7897, val_acc: 0.9102
Epoch [1], val_loss: 24.4208, val_acc: 0.9122
Epoch [2], val_loss: 22.9626, val_acc: 0.9173
Epoch [3], val_loss: 20.8648, val_acc: 0.9212
Epoch [4], val_loss: 19.2588, val_acc: 0.9241
Epoch [5], val_loss: 18.7075, val_acc: 0.9216
Epoch [6], val_loss: 17.5894, val_acc: 0.9233
Epoch [7], val_loss: 16.0324, val_acc: 0.9267
Epoch [8], val_loss: 16.8926, val_acc: 0.9144
Epoch [9], val_loss: 15.2275, val_acc: 0.9275
```

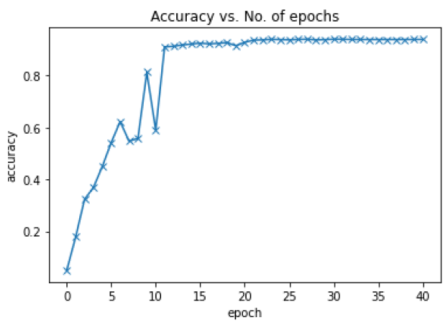
In [228…
```
history3 = fit(10, 0.00001, model, train_dl, val_dl)
```

```
Epoch [0], val_loss: 12.6428, val_acc: 0.9358
Epoch [1], val_loss: 12.6041, val_acc: 0.9369
Epoch [2], val_loss: 12.5687, val_acc: 0.9390
Epoch [3], val_loss: 12.3868, val_acc: 0.9381
Epoch [4], val_loss: 12.4568, val_acc: 0.9370
Epoch [5], val_loss: 12.1960, val_acc: 0.9386
Epoch [6], val_loss: 12.1439, val_acc: 0.9396
Epoch [7], val_loss: 12.1691, val_acc: 0.9369
Epoch [8], val_loss: 12.1079, val_acc: 0.9378
Epoch [9], val_loss: 12.0134, val_acc: 0.9396
```

In [229…
```
history4 = fit(10, 0.000001, model, train_dl, val_dl) # seems the accuracy reach convergence
```

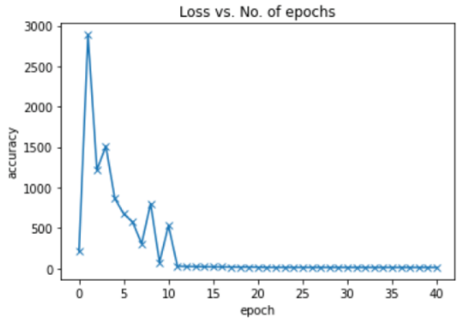
```
Epoch [0], val_loss: 11.9393, val_acc: 0.9393
Epoch [1], val_loss: 11.9174, val_acc: 0.9385
Epoch [2], val_loss: 11.8952, val_acc: 0.9385
Epoch [3], val_loss: 11.8812, val_acc: 0.9382
Epoch [4], val_loss: 11.8682, val_acc: 0.9382
Epoch [5], val_loss: 11.8517, val_acc: 0.9384
Epoch [6], val_loss: 11.8347, val_acc: 0.9384
Epoch [7], val_loss: 11.8251, val_acc: 0.9384
Epoch [8], val_loss: 11.8240, val_acc: 0.9385
Epoch [9], val_loss: 11.8076, val_acc: 0.9388
```

In [230…
```
history = [result0] + history1 + history2 + history3 + history4
accuracies = [result['val_acc'] for result in history]
plt.plot(accuracies, '-x')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('Accuracy vs. No. of epochs');
```



In [231…

```
#loss function is cross_entropy
history = [result0] + history1 + history2 + history3 + history4
accuracies = [result['val_loss'] for result in history]
plt.plot(accuracies, '-x')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('Loss vs. No. of epochs');
```



In [232…

```
# evaluate on test dataset
result = evaluate(model, test_dl)
result
```

Out[232…

```
{'val_loss': 166.9779510498047, 'val_acc': 0.6805098652839661}
```

In [233…

```
accuracy_per_class(outputs, labels)
```

Out[233…

```
tensor([1.0000, 0.7755, 0.8677, 0.8857, 0.8554, 0.9150, 0.8017, 0.7156, 0.7014,
           nan, 0.4441, 0.8995, 0.5254, 0.5086, 0.5976, 0.8674, 0.7805, 0.4236,
        0.3252, 0.5282, 0.3985, 0.4249, 0.6650, 0.5918, 0.5693,    nan])
```

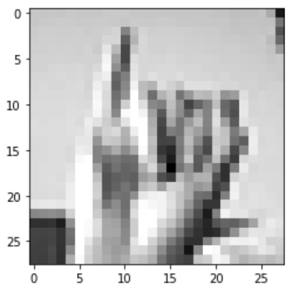
In [234…

```
# Prediction
def predict_image(img, model):
    xb = img.unsqueeze(0)
    yb = model(xb)
    _, preds  = torch.max(yb, dim=1)
    return preds[0].item()
```

In [235…

```
img, label = test_ds[10]
plt.imshow(img.view(28,28), cmap='gray')
print('Label:', label.item(), ', Predicted:', predict_image(img, model))
```

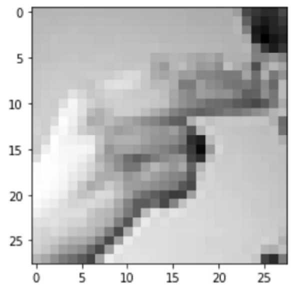
```
Label: 8 , Predicted: 8
```



In [236…

```
img, label = test_ds[200]
plt.imshow(img.view(28,28), cmap='gray')
print('Label:', label.item(), ', Predicted:', predict_image(img, model))
```

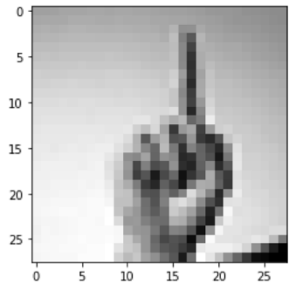
```
Label: 7 , Predicted: 7
```



In [237…

```
img, label = test_ds[1000]
plt.imshow(img.view(28,28), cmap='gray')
print('Label:', label.item(), ', Predicted:', predict_image(img, model))
```

```
Label: 3 , Predicted: 3
```



APPENDIX B: CNN Model

CNN MODEL CODE RETRIEVED FROM: https://www.kaggle.com/code/sayakdasgupta/sign-language-classification-cnn-99-40-accuracy/notebook

In [1]:
```
#1. IMPORTING INITIAL NECESSARY LIBRARIES
import numpy as np
import pandas as pd
import keras
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPool2D, Dropout
import matplotlib.pyplot as plt
```

In [2]:
```
#2. DATA PREPERATION / PREPROCESSING

#2.1 Importing the dataset:
with np.load('train_data_label.npz') as data:
    X_train = data['train_data']
    Y_train = data['train_label']

with np.load('test_data_label.npz') as data:
    X_test = data['test_data']
    Y_test = data['test_label']
```

In [3]:
```
#2.2 Checking the shape:
print(X_train.shape, Y_train.shape, X_test.shape, Y_test.shape)
```

```
(27455, 784) (27455,) (7172, 784) (7172,)
```

In [4]:
```
#2.3 Splitting the training data set into a training and validation set:
from sklearn.model_selection import train_test_split

X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size = 0.25, random_state = 999)
print(X_train.shape, Y_train.shape, X_val.shape, Y_val.shape, X_test.shape, Y_test.shape)
```

```
(20591, 784) (20591,) (6864, 784) (6864,) (7172, 784) (7172,)
```

In [5]:
```
#2.4 Reshaping the data:
X_train = X_train.reshape(-1, 28,28, 1)
print(X_train.shape)

X_val = X_val.reshape(-1, 28, 28, 1)
print(X_val.shape)

X_test = X_test.reshape(-1, 28, 28, 1)
print(X_test.shape)
```

```
(20591, 28, 28, 1)
(6864, 28, 28, 1)
(7172, 28, 28, 1)
```

In [6]:
```
#2.5 Converting the labels to binary form:
from sklearn.preprocessing import LabelBinarizer
lb = LabelBinarizer()

Y_train = lb.fit_transform(Y_train)
Y_val = lb.fit_transform(Y_val)
Y_test = lb.fit_transform(Y_test)
```

In [7]:
```
#2.6 Checking the data after reshaping and binarizing:
print(X_train.shape, Y_train.shape, X_val.shape, Y_val.shape, X_test.shape, Y_test.shape)
```

```
(20591, 28, 28, 1) (20591, 24) (6864, 28, 28, 1) (6864, 24) (7172, 28, 28, 1) (7172, 24)
```

The CNN model consist of:

1. Three convolution layers - each followed by MaxPooling for better feature capture.
2. Dense layer of 512 units.
3. Output layer with 24 units for 24 different classes.

In [8]:
```
#3. BUILDING THE CNN MODEL

#3.1 Convolution layers:
model = Sequential()
model.add(Conv2D(128, kernel_size = (5, 5),
                 strides = 1, padding = 'same', activation = 'relu', input_shape = (28, 28, 1)))
model.add(MaxPool2D(pool_size = (3, 3), strides = 2, padding = 'same'))
model.add(Conv2D(64, kernel_size = (2, 2),
                strides = 1, activation = 'relu', padding = 'same'))
model.add(MaxPool2D((2, 2), 2, padding = 'same'))
```

```
model.add(Conv2D(32, kernel_size = (2, 2),
                strides = 1,activation = 'relu', padding = 'same'))
model.add(MaxPool2D((2, 2), 2, padding = 'same'))

model.add(Flatten())
```

```
2022-06-04 20:44:37.550680: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized wit
h oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:
AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

In [9]:

```
#3.2 Dense and output layers:
model.add(Dense(units = 512, activation = 'relu'))
model.add(Dropout(rate = 0.25))

model.add(Dense(units = 24, activation = 'softmax'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 128)       3328

 max_pooling2d (MaxPooling2D  (None, 14, 14, 128)      0
 )

 conv2d_1 (Conv2D)           (None, 14, 14, 64)        32832

 max_pooling2d_1 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 7, 7, 32)          8224

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 32)         0
 2D)

 flatten (Flatten)           (None, 512)               0

 dense (Dense)               (None, 512)               262656

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 24)                12312

=================================================================
Total params: 319,352
Trainable params: 319,352
Non-trainable params: 0
_________________________________________________________________
```

In [10]:

```
#3.3 Adding the optimizer, loss and metric:
model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
```

In [11]:

```
#4. TRAINING THE CNN MODEL
history = model.fit(X_train, Y_train, batch_size = 64, epochs = 35, verbose = 2)
```

```
Epoch 1/35
322/322 - 66s - loss: 1.7683 - accuracy: 0.5160 - 66s/epoch - 205ms/step
Epoch 2/35
322/322 - 71s - loss: 0.2093 - accuracy: 0.9301 - 71s/epoch - 219ms/step
Epoch 3/35
322/322 - 69s - loss: 0.0658 - accuracy: 0.9779 - 69s/epoch - 215ms/step
Epoch 4/35
322/322 - 77s - loss: 0.0348 - accuracy: 0.9883 - 77s/epoch - 238ms/step
Epoch 5/35
322/322 - 76s - loss: 0.0468 - accuracy: 0.9852 - 76s/epoch - 237ms/step
Epoch 6/35
322/322 - 72s - loss: 0.0165 - accuracy: 0.9948 - 72s/epoch - 224ms/step
Epoch 7/35
322/322 - 77s - loss: 0.0672 - accuracy: 0.9795 - 77s/epoch - 239ms/step
Epoch 8/35
322/322 - 81s - loss: 0.0505 - accuracy: 0.9850 - 81s/epoch - 253ms/step
Epoch 9/35
322/322 - 61s - loss: 0.0422 - accuracy: 0.9870 - 61s/epoch - 191ms/step
Epoch 10/35
322/322 - 70s - loss: 0.0364 - accuracy: 0.9891 - 70s/epoch - 216ms/step
Epoch 11/35
322/322 - 75s - loss: 0.0083 - accuracy: 0.9973 - 75s/epoch - 232ms/step
Epoch 12/35
322/322 - 85s - loss: 0.0032 - accuracy: 0.9989 - 85s/epoch - 265ms/step
Epoch 13/35
322/322 - 75s - loss: 0.0872 - accuracy: 0.9764 - 75s/epoch - 233ms/step
Epoch 14/35
322/322 - 75s - loss: 0.0078 - accuracy: 0.9973 - 75s/epoch - 233ms/step
```

```
Epoch 15/35
322/322 - 76s - loss: 0.0402 - accuracy: 0.9883 - 76s/epoch - 237ms/step
Epoch 16/35
322/322 - 79s - loss: 0.0415 - accuracy: 0.9876 - 79s/epoch - 246ms/step
Epoch 17/35
322/322 - 86s - loss: 0.0269 - accuracy: 0.9924 - 86s/epoch - 266ms/step
Epoch 18/35
322/322 - 71s - loss: 0.0443 - accuracy: 0.9874 - 71s/epoch - 219ms/step
Epoch 19/35
322/322 - 68s - loss: 0.0328 - accuracy: 0.9903 - 68s/epoch - 210ms/step
Epoch 20/35
322/322 - 68s - loss: 0.0287 - accuracy: 0.9916 - 68s/epoch - 210ms/step
Epoch 21/35
322/322 - 80s - loss: 0.0168 - accuracy: 0.9960 - 80s/epoch - 249ms/step
Epoch 22/35
322/322 - 88s - loss: 0.0230 - accuracy: 0.9938 - 88s/epoch - 274ms/step
Epoch 23/35
322/322 - 93s - loss: 0.0222 - accuracy: 0.9943 - 93s/epoch - 289ms/step
Epoch 24/35
322/322 - 68s - loss: 0.0406 - accuracy: 0.9907 - 68s/epoch - 210ms/step
Epoch 25/35
322/322 - 86s - loss: 0.0450 - accuracy: 0.9913 - 86s/epoch - 266ms/step
Epoch 26/35
322/322 - 85s - loss: 0.0201 - accuracy: 0.9953 - 85s/epoch - 265ms/step
Epoch 27/35
322/322 - 70s - loss: 0.0152 - accuracy: 0.9966 - 70s/epoch - 217ms/step
Epoch 28/35
322/322 - 61s - loss: 0.0035 - accuracy: 0.9987 - 61s/epoch - 189ms/step
Epoch 29/35
322/322 - 53s - loss: 0.0430 - accuracy: 0.9913 - 53s/epoch - 163ms/step
Epoch 30/35
322/322 - 52s - loss: 0.0740 - accuracy: 0.9863 - 52s/epoch - 163ms/step
Epoch 31/35
322/322 - 52s - loss: 0.0189 - accuracy: 0.9957 - 52s/epoch - 160ms/step
Epoch 32/35
322/322 - 68s - loss: 0.0088 - accuracy: 0.9982 - 68s/epoch - 212ms/step
Epoch 33/35
322/322 - 60s - loss: 0.0228 - accuracy: 0.9953 - 60s/epoch - 187ms/step
Epoch 34/35
322/322 - 52s - loss: 0.0278 - accuracy: 0.9936 - 52s/epoch - 161ms/step
Epoch 35/35
322/322 - 58s - loss: 0.0243 - accuracy: 0.9957 - 58s/epoch - 182ms/step
```

HYPERPARAMETER TUNING CODE RETRIEVED FROM: https://www.projectpro.io/recipes/find-optimal-parameters-using-randomizedsearchcv-for-regression

In [12]:

```
#5. HYPERPARAMETER TUNING: GRID SEARCH ON BATCH SIZE AND EPOCHS
from sklearn.model_selection import RandomizedSearchCV
from keras.wrappers.scikit_learn import KerasClassifier
from scipy.stats import randint

#5.1 Create function:
def create_model():
    tuned_model = Sequential()
    tuned_model.add(Conv2D(128, kernel_size = (5, 5),
                    strides = 1, padding = 'same', activation = 'relu', input_shape = (28, 28, 1)))
    tuned_model.add(MaxPool2D(pool_size = (3, 3), strides = 2, padding = 'same'))
    tuned_model.add(Conv2D(64, kernel_size = (2, 2),
                   strides = 1, activation = 'relu', padding = 'same'))
    tuned_model.add(MaxPool2D((2, 2), 2, padding = 'same'))
    tuned_model.add(Conv2D(32, kernel_size = (2, 2),
                   strides = 1,activation = 'relu', padding = 'same'))
    tuned_model.add(MaxPool2D((2, 2), 2, padding = 'same'))
    tuned_model.add(Flatten())
    tuned_model.add(Dense(units = 512, activation = 'relu'))
    tuned_model.add(Dropout(rate = 0.25))
    tuned_model.add(Dense(units = 24, activation = 'softmax'))
    tuned_model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
    return tuned_model
```

In [13]:

```
#5.2 Random seed:
seed = 10
np.random.seed(seed)
```

In [14]:

```
#5.3 Create model for hyperparameter tuning:
tuned_model = KerasClassifier(build_fn = create_model, verbose = 0)
```

```
/var/folders/98/kttfxt7d7f1fv3shp9t9wr3c0000gn/T/ipykernel_56904/3328085265.py:2: DeprecationWarning: KerasClassifier is deprecated, use Sci-Keras (https://github.com/adriangb/scikeras) instead. See https://www.adriangb.com/scikeras/stable/m
igration.html for help migrating.
  tuned_model = KerasClassifier(build_fn = create_model, verbose = 0)
```

In [15]:

```
#5.4 Defining the randomized search hyperparameters & executing the randomized search:
```

```python
param_distributions = {'batch_size': randint.rvs(10, 100, size = 10),
                       'epochs': randint.rvs(10, 50, size = 5)
                      }

rs = RandomizedSearchCV(estimator = tuned_model,
                        param_distributions = param_distributions,
                        n_iter = 10,
                        n_jobs = -1,
                        cv = 3
                       )

rs_result = rs.fit(X_train, Y_train)
```

```
2022-06-04 21:27:15.643950: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized wit
h oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:
AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2022-06-04 21:27:15.889298: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized wit
h oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:
AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2022-06-04 21:27:15.891056: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized wit
h oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:
AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2022-06-04 21:27:15.928964: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized wit
h oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:
AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

In [17]:

```python
#5.5 Results randomized search:
print("\n The best score across ALL searched params:\n", rs_result.best_score_)
print("\n The best parameters across ALL searched params:\n", rs_result.best_params_)
```

```
 The best score across ALL searched params:
 0.9988344113032023

 The best parameters across ALL searched params:
 {'epochs': 26, 'batch_size': 25}
```

In [18]:

```python
#5.6 Fitting the tuned model and testing it on the validation set:
tuned_history = tuned_model.fit(X_train, Y_train, batch_size = 25,
                                epochs = 26,
                                verbose = 2,
                                validation_data = (X_val, Y_val)
                               )
```

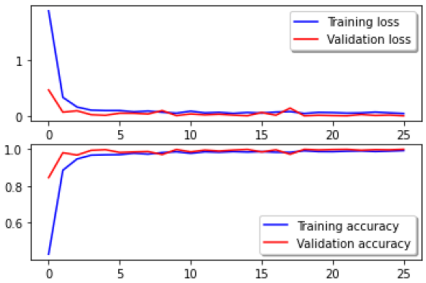
```
Epoch 1/26
824/824 - 60s - loss: 1.8929 - accuracy: 0.4298 - val_loss: 0.4666 - val_accuracy: 0.8460 - 60s/epoch - 73ms/step
Epoch 2/26
824/824 - 63s - loss: 0.3331 - accuracy: 0.8862 - val_loss: 0.0658 - val_accuracy: 0.9814 - 63s/epoch - 77ms/step
Epoch 3/26
824/824 - 65s - loss: 0.1581 - accuracy: 0.9475 - val_loss: 0.0903 - val_accuracy: 0.9688 - 65s/epoch - 79ms/step
Epoch 4/26
824/824 - 67s - loss: 0.1016 - accuracy: 0.9682 - val_loss: 0.0195 - val_accuracy: 0.9943 - 67s/epoch - 81ms/step
Epoch 5/26
824/824 - 59s - loss: 0.0938 - accuracy: 0.9703 - val_loss: 0.0091 - val_accuracy: 0.9978 - 59s/epoch - 72ms/step
Epoch 6/26
824/824 - 61s - loss: 0.0943 - accuracy: 0.9712 - val_loss: 0.0474 - val_accuracy: 0.9830 - 61s/epoch - 74ms/step
Epoch 7/26
824/824 - 57s - loss: 0.0728 - accuracy: 0.9784 - val_loss: 0.0465 - val_accuracy: 0.9856 - 57s/epoch - 69ms/step
Epoch 8/26
824/824 - 57s - loss: 0.0861 - accuracy: 0.9740 - val_loss: 0.0337 - val_accuracy: 0.9876 - 57s/epoch - 69ms/step
Epoch 9/26
824/824 - 57s - loss: 0.0639 - accuracy: 0.9823 - val_loss: 0.0937 - val_accuracy: 0.9717 - 57s/epoch - 69ms/step
Epoch 10/26
824/824 - 57s - loss: 0.0478 - accuracy: 0.9865 - val_loss: 0.0036 - val_accuracy: 0.9990 - 57s/epoch - 69ms/step
Epoch 11/26
824/824 - 58s - loss: 0.0842 - accuracy: 0.9782 - val_loss: 0.0335 - val_accuracy: 0.9859 - 58s/epoch - 70ms/step
Epoch 12/26
824/824 - 58s - loss: 0.0525 - accuracy: 0.9865 - val_loss: 0.0168 - val_accuracy: 0.9958 - 58s/epoch - 70ms/step
Epoch 13/26
824/824 - 71s - loss: 0.0619 - accuracy: 0.9841 - val_loss: 0.0281 - val_accuracy: 0.9905 - 71s/epoch - 86ms/step
Epoch 14/26
824/824 - 62s - loss: 0.0442 - accuracy: 0.9874 - val_loss: 0.0125 - val_accuracy: 0.9958 - 62s/epoch - 75ms/step
Epoch 15/26
824/824 - 60s - loss: 0.0596 - accuracy: 0.9852 - val_loss: 0.0015 - val_accuracy: 0.9993 - 60s/epoch - 73ms/step
Epoch 16/26
824/824 - 59s - loss: 0.0483 - accuracy: 0.9873 - val_loss: 0.0598 - val_accuracy: 0.9857 - 59s/epoch - 72ms/step
Epoch 17/26
824/824 - 60s - loss: 0.0674 - accuracy: 0.9842 - val_loss: 0.0105 - val_accuracy: 0.9972 - 60s/epoch - 73ms/step
Epoch 18/26
824/824 - 59s - loss: 0.0764 - accuracy: 0.9835 - val_loss: 0.1389 - val_accuracy: 0.9730 - 59s/epoch - 71ms/step
Epoch 19/26
824/824 - 63s - loss: 0.0425 - accuracy: 0.9918 - val_loss: 5.1834e-04 - val_accuracy: 0.9997 - 63s/epoch - 76ms/step
Epoch 20/26
```

```
824/824 - 72s - loss: 0.0620 - accuracy: 0.9875 - val_loss: 0.0103 - val_accuracy: 0.9965 - 72s/epoch - 87ms/step
Epoch 21/26
824/824 - 63s - loss: 0.0579 - accuracy: 0.9871 - val_loss: 0.0047 - val_accuracy: 0.9985 - 63s/epoch - 77ms/step
Epoch 22/26
824/824 - 65s - loss: 0.0488 - accuracy: 0.9895 - val_loss: 7.8748e-04 - val_accuracy: 0.9996 - 65s/epoch - 79ms/step
Epoch 23/26
824/824 - 73s - loss: 0.0524 - accuracy: 0.9910 - val_loss: 0.0229 - val_accuracy: 0.9949 - 73s/epoch - 89ms/step
Epoch 24/26
824/824 - 65s - loss: 0.0669 - accuracy: 0.9879 - val_loss: 0.0070 - val_accuracy: 0.9980 - 65s/epoch - 79ms/step
Epoch 25/26
824/824 - 68s - loss: 0.0535 - accuracy: 0.9902 - val_loss: 0.0131 - val_accuracy: 0.9971 - 68s/epoch - 83ms/step
Epoch 26/26
824/824 - 73s - loss: 0.0424 - accuracy: 0.9930 - val_loss: 4.2883e-04 - val_accuracy: 0.9999 - 73s/epoch - 89ms/step
```

EVALUATION CODE RETRIEVED FROM: https://www.kaggle.com/code/gaurav3435/cnn-model/notebook

In [19]:

```
#6. EVALUATING THE CNN MODEL

#6.1 Plotting the accuracy / loss:
fig, ax = plt.subplots(2,1)

ax[0].plot(tuned_history.history['loss'],
           color='b',
           label = "Training loss"
          )
ax[0].plot(tuned_history.history['val_loss'],
           color='r',
           label = "Validation loss",
           axes = ax[0]
          )
legend = ax[0].legend(loc = 'best', shadow = True)

ax[1].plot(tuned_history.history['accuracy'],
           color = 'b',
           label = "Training accuracy"
          )
ax[1].plot(tuned_history.history['val_accuracy'],
           color = 'r',
           label = "Validation accuracy"
          )
legend = ax[1].legend(loc = 'best', shadow = True)
```



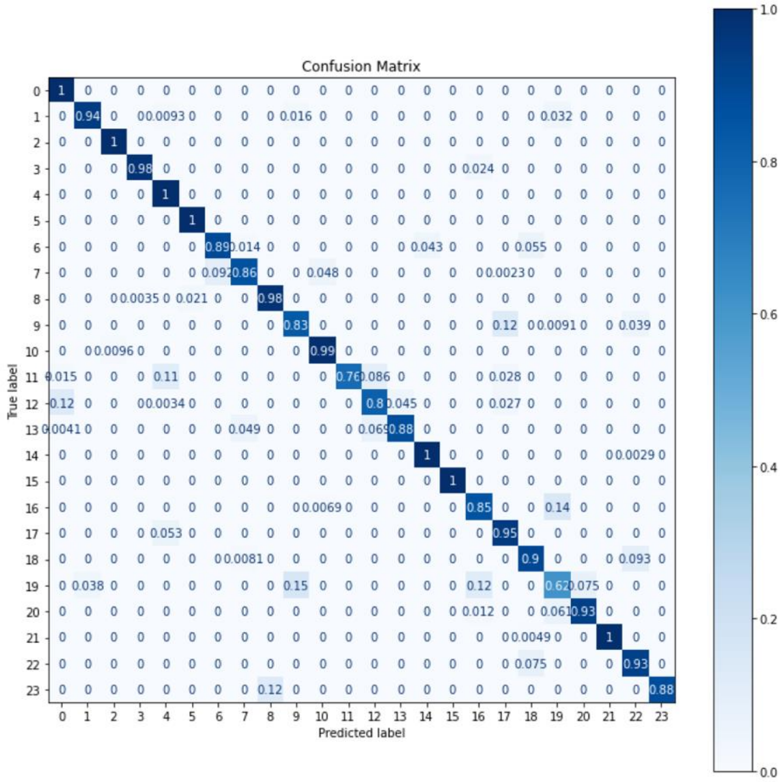
In [56]:

```
#6.2 Predictions on the test set:
Y_pred = model.predict(X_test, batch_size = 25)

Y_pred_class = np.argmax(Y_pred, axis = 1)
Y_test_class = np.argmax(Y_test, axis = 1)

test_accuracy = np.mean(Y_pred_class == Y_test_class)

print("Test accuracy: ", test_accuracy, "\n")
```

```
287/287 [==============================] - 5s 19ms/step
Test accuracy:  0.9128555493586168
```

In [57]:

```
#6.3 Plotting the confusion matrix:
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import confusion_matrix

fig, ax = plt.subplots(figsize = (12, 12))
cm = confusion_matrix(Y_test_class, Y_pred_class, normalize = 'true')
disp = ConfusionMatrixDisplay(confusion_matrix = cm)
disp = disp.plot(ax = ax,cmap = plt.cm.Blues)
ax.set_title("Confusion Matrix")
plt.show()
```



In [58]:

```
#6.4 Classification report:
from sklearn.metrics import classification_report

print(classification_report(Y_test_class, Y_pred_class))
```

```
              precision    recall  f1-score   support

           0       0.89      1.00      0.94       331
           1       0.98      0.94      0.96       432
           2       0.99      1.00      1.00       310
           3       1.00      0.98      0.99       245
           4       0.89      1.00      0.94       498
           5       0.98      1.00      0.99       247
           6       0.89      0.89      0.89       348
           7       0.95      0.86      0.90       436
           8       0.88      0.98      0.92       288
           9       0.85      0.83      0.84       331
          10       0.90      0.99      0.95       209
          11       1.00      0.76      0.87       394
          12       0.82      0.80      0.81       291
          13       0.94      0.88      0.91       246
          14       0.96      1.00      0.98       347
          15       1.00      1.00      1.00       164
          16       0.75      0.85      0.80       144
          17       0.79      0.95      0.86       246
          18       0.85      0.90      0.87       248
          19       0.74      0.62      0.67       266
          20       0.94      0.93      0.93       346
          21       1.00      1.00      1.00       206
          22       0.87      0.93      0.90       267
          23       1.00      0.88      0.94       332

    accuracy                           0.91      7172
   macro avg       0.91      0.91      0.91      7172
weighted avg       0.92      0.91      0.91      7172
```