DALHOUSIE UNIVERSITY

Faculty of Computer Science

CSCI 6515 – Machine Learning for Big Data

Name: Arka Ghosh
Banner ID: B00911033
Assignment: 03

Task 1

(i) The dataset [1] that has been used in this assignment is Sign Language MNIST which is a collection of handwritten digits. The dataset is divided into two metadata namely "sign_mnist_train.csv" and "sign_mnist_test.csv" respectively. The training metadata contains 27,455 training samples and the testing metadata contains 7,172 test samples. Each of these examples consists of a grayscale 28x28 pixel picture with values ranging from 0 to 255 and a number label for the image from 0 to 25, with each value corresponding to a letter of the English alphabet, such as 0 to A, 1 to B, and so on. However, there is no cases for the J (9) and Z (25) due to the motion gesture motions.

For this assignment, I have worked with both the metadata file namely ""sign_mnist_train.csv" and "sign_mnist_test.csv" as both the files include a header row with the labels pixel1, pixel2,……, pixel784, each of which stands for a single 28x28 pixel picture with grayscale values ranging from 0-255.

(ii) The raw data of the Sign Language MNIST dataset has been provided as class-wise distributed csv files with pixel to pixel intensities ranging from 0-255. For the goal of recognizing gestures and signals related to sign language, several proposals have been put forth by researchers in the form of patents and research papers using the dataset. Mannan et al. [2] proposed a DeepCNN model for sign language recognition that consists of three consecutive CNN layers with kernel size of 3x3 and ReLU activation function followed by a flatten layer and three dense layers among which the final layer being the output layer. Additionally, 2x2 maxpooling has been used after each CNN layers to gradually shrink the spatial dimension to match the model's fewer parameters and simpler processing. The model is trained using six different learning rate and yields a training accuracy of 98.69% and validation accuracy of 98.93% with 0.00050 learning rate. The model has also been assessed on unforeseen data and yielded a testing accuracy of 99.67%. Ma et al. [3] proposed a Two-Stream Mixed (TSM) approach that combines feature extraction with fusion to enhance the correlation of feature expression between two time-consecutive pictures of hand gestures for dynamic movements. The TSM-CNN system consists of preprocessing, TSM block and CNN classifier where scaling, transformation, and augmentation are performed on the two successive pictures in the dynamic gesture that are utilized as streams' inputs in the pre-processing stage. Four different CNN models have been implemented with TSM namely TSM-LeNet, TSM-AlexNet, TSM-ResNet18, and TSM-ResNet50. Among these models, TSM-ResNet50 yielded best result for both MNIST and ASL datasets with an accuracy of 97.57%. So [4] proposed a VGG16 inspired CNN model for sign language classification which consists of three 2D convolutional layers with 64, 128 and 256 filters respectively and a kernel size of 3x3. The final two layers are a fully connected layer with 512 units connected to a dense layer of the number of the classes. A dropout of 0.2 has been used as well to randomly set the weights of the neurons to zero to avoid overfitting. In addition to that, several data augmentation technique has also been used by rotating the images up to 30 degrees and shifting and zooming in 10 with horizontal flips to increase the size of the training dataset. This VGG16 inspired CNN model has been able to achieve a training accuracy of 0.9970, validation accuracy of 0.9999 and testing accuracy of 0.9118 on the original dataset. With the augmented data, this architecture has been able to achieve a training accuracy of 0.8715, validation accuracy of 0.9591 and testing accuracy of 0.04785 and 0.95411 on two different runs that reduces the overfitting.

(iii) The dataset has two metadata, one with training samples and other with testing samples. Both of the dataset has no null or NA values which is shown in the following screenshot:

```
In [8]: print("Total Null values in the Training Dataset:", train_data.isnull().values.sum())
        print("Total NA values in the Training Dataset:", train_data.isna().values.sum())

        Total Null values in the Training Dataset: 0
        Total NA values in the Training Dataset: 0

In [9]: print("Total Null values in the Testing Dataset:", test_data.isnull().values.sum())
        print("Total NA values in the Testing Dataset:", test_data.isna().values.sum())

        Total Null values in the Testing Dataset: 0
        Total NA values in the Testing Dataset: 0
```

Figure 1: Number of NA and Null values in Training and Testing Data

The dataset provides pixel values ranging from 0 to 255. Although these pixel values can be fed directly in their raw form to neural network models, this might create difficulties during modelling, such as slower than anticipated model training [5]. Small weight values are processed by neural networks, while high integer values might slow down learning. This is the reason why I have scaled the pixel values between 0 and 1 which is achieved by diving all the pixel values by the highest pixel value 255. Also, CNN takes only 4D array as an input where the first dimension denotes the batch size of the image and the other three dimension represents the height, width and depth respectively [6]. As the images are 28x28 and the channel is 1 since grayscale images are being used, so training and testing images need to be reshaped in the following way before feeding them as inputs into the neural network:

```
In [17]: X_train = X_train.values.reshape(X_train.shape[0],28,28,1)

In [18]: X_test = X_test.values.reshape(X_test.shape[0],28,28,1)

In [19]: print(X_train.shape)
         print(X_test.shape)

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

Figure 2: Reshaping of the Training and Testing Samples

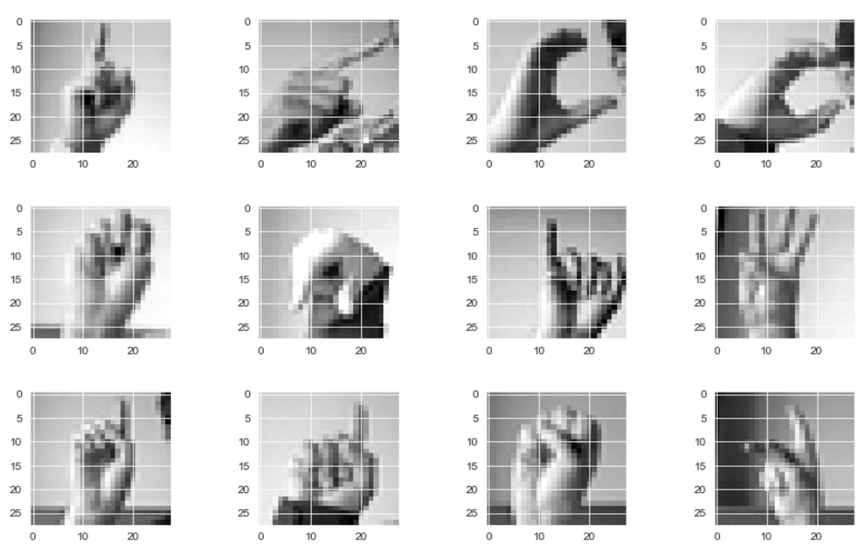


Figure 3: First 12 Images from the Training Dataset

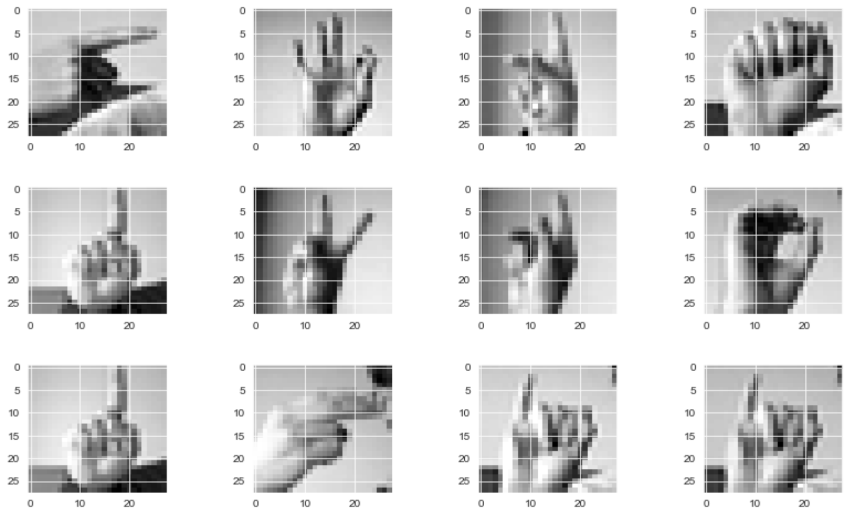


Figure 4: First 12 images from the Testing Dataset

(iv) The dataset is already split into training and testing samples so it is not needed to further split the dataset into training and testing samples. Instead, the training dataset is divided into training and validation set with a proportion of 80-20 where 80% data has been used for training and 20% data has been used for validation.

Task 2

(i) Convolution refers to a mathematical procedure of combining two sets of information which is used to produce a third function [7]. In terms of Convolutional Neural Network, convolution is used to filter the input data and create a feature map. Convolution happens in the convolutional layer which is the core building of CNN and it requires some components i.e. input data, a filter and feature map. Let's assume we are trying to feed a grayscale image to the CNN. So, the input will have 4 dimensions – batch size, height, width and depth – which is the channel of the image. We do have a feature detector, known as kernel that is two dimensional array of weights. These kernels can be different in size such as 2x2, 3x3 matrix. The feature map revolves around a certain field of the image to calculate a dot product between the input pixels and filter which is later fed into an output array. This process is known as convolution [8]. The kernel then moves forward by a stride and repeats the operation until the kernel has covered the entire picture. The final output array is known as feature map of convolved map which is output from the series of dot products produced by the filter and the input.

(ii) I have used Accuracy as an evaluation metric in this assignment which is computed by dividing by the total number of correct predictions by the total number of predictions made for a dataset. In the Sign Language dataset, the target variable in both the training and testing metadata consist of numerical values ranging from 0 to 25 where each number correspond to an alphabet. The distribution of the target variable in training dataset has been shown in the following screenshot:

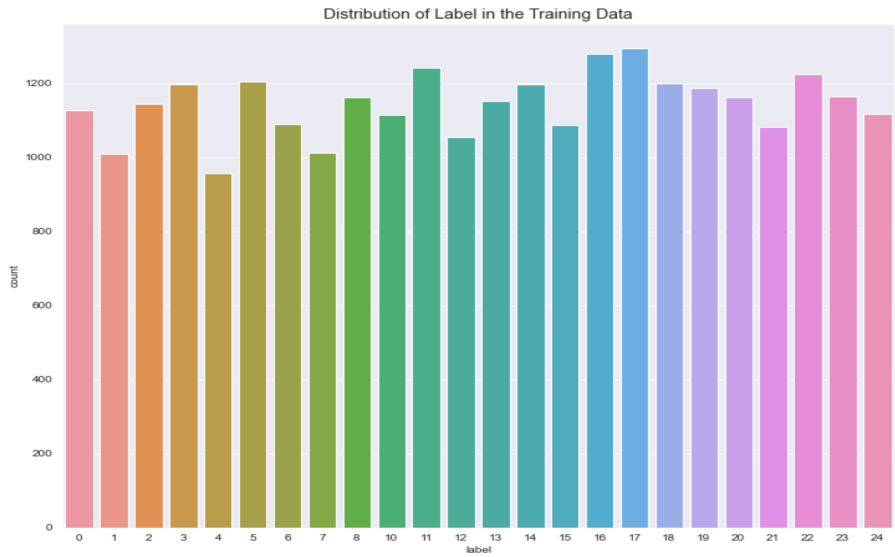


Figure 5: Distribution of Target Variable in the Training Dataset

From the above screenshot it can be noticed that the label in the training images are fairly equally distributed across each classes, with an average of around 1000 samples in each class. As a matter of fact, it can be said that the dataset is fairly balanced. Accuracy is selected as an evaluation metric for this task as we know, accuracy is well suited when the target class is well balanced [9].

(iii) I have used three different sequential CNN models with different numbers and types of layers which has been described below:

Model 1:

The first CNN model that I have used has three CNN layers followed by two dense layer among which the last layer being the output layer. I have also used batch normalization with each batch's mean and standard deviation to normalize the values of the units and dropout value of 0.2 to randomly select 20% of the neurons and set their weights to zero to avoid overfitting [10]. The input that has been fed into this model has a shape of (28, 28, 1) where the first two dimensions denotes the height and width of the input and the third dimension denotes the channel of the input which is 1 in this case as we are using greyscale images as input. The filters used in this model are 32, 64, 128 for the CNN layers and 64, 26 for the dense layers which refers to the dimensionality of the output space. Rectified Linear Unit (ReLU) has been used as an activation function in the CNN layers and the first dense layer as it helps in limiting the exponential increase of the computation needed to train the neural network because of its nature of not activating all the neurons at the same time. Softmax is used in the last layer as it is the output layer that turns raw outputs into a vector of probabilities over the input classes [11]. Also, it is a multiclass classification task and Softmax provides probabilities for each class label. The architecture diagram of the model is provided below:

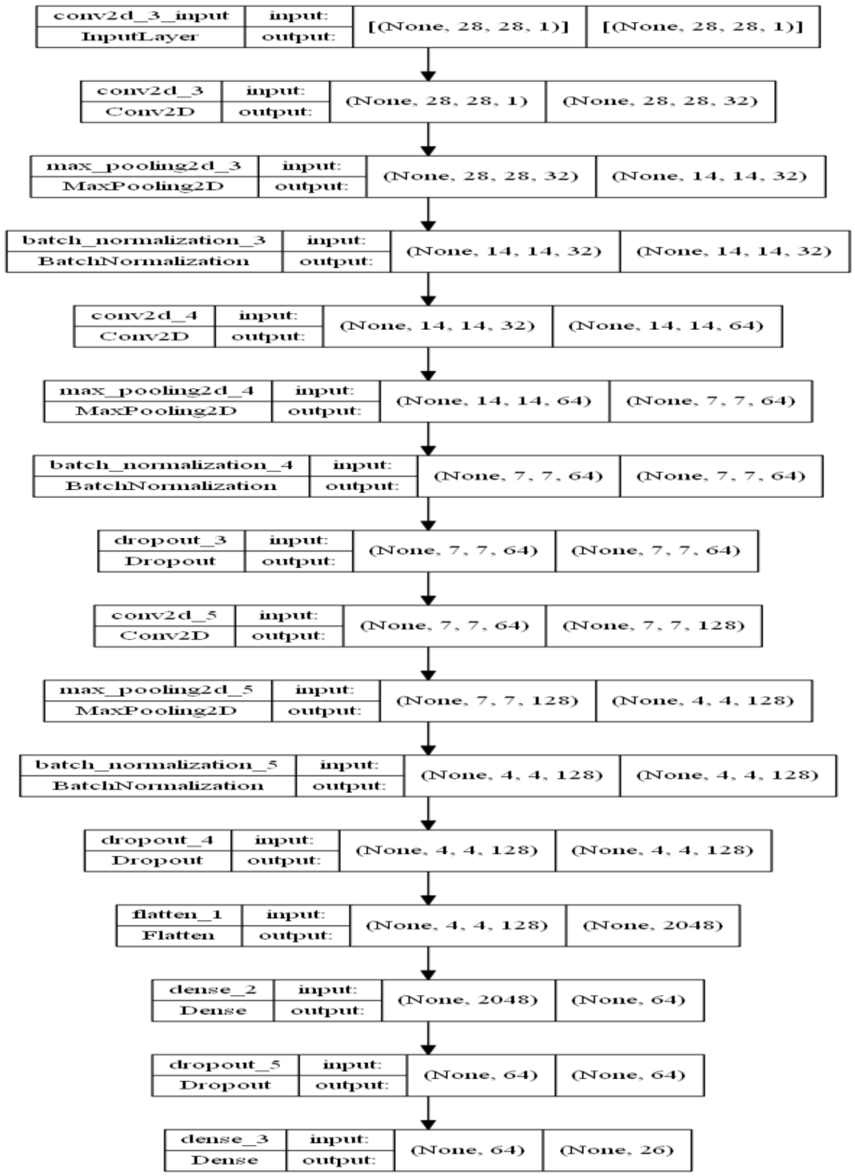


Figure 6: Model 1 – Three CNN Layers with Batch Normalization and Dropout followed by two Dense Layers

The obtained training, validation and testing accuracy of this model are:

Training Accuracy: 100.00%

Testing Accuracy: 97.59%

Model 2:

The second CNN model that I have used has similar three CNN layers followed by two dense layer among which the last layer being the output layer as the first model but I have removed the batch normalization and dropout in this model. All other components are kept similar as the first model. The diagram architecture of this model has been provided below:

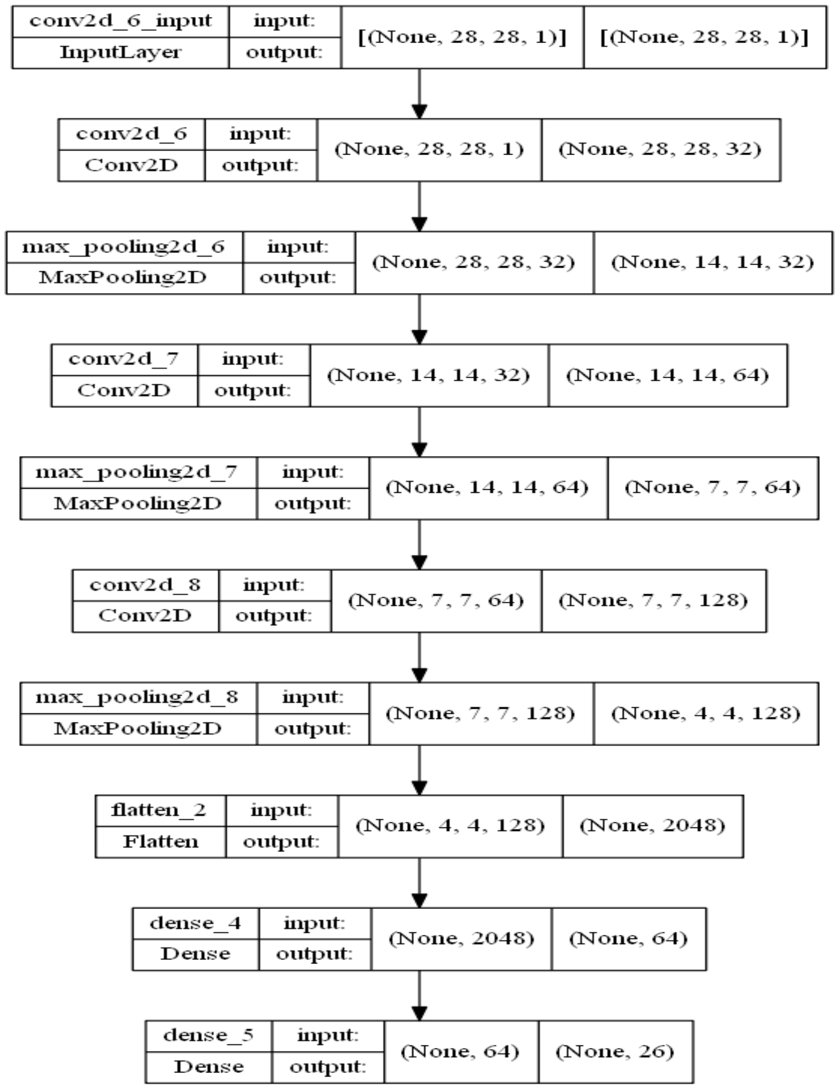


Figure 7: Figure: Model 2 - Three CNN Layers without Batch Normalization and Dropout followed by two Dense Layers

The obtained training, validation and testing accuracy of this model are:

Training Accuracy: 100.00%

Testing Accuracy: 91.09%

Model 3:

The third CNN model that I have used consists of two CNN layers followed by two dense layer among which the last layer being the output layer. Batch normalization and dropout of 0.2 has been used in this model as well. The activation function used in this first three layers (two CNN layers and first dense layer) is ReLU and the activation function used in the output layer is softmax. The filters used in the CNN layers are 64, 128 and in the dense layers are 64, 26 respectively. The architecture diagram of this model is provided below:

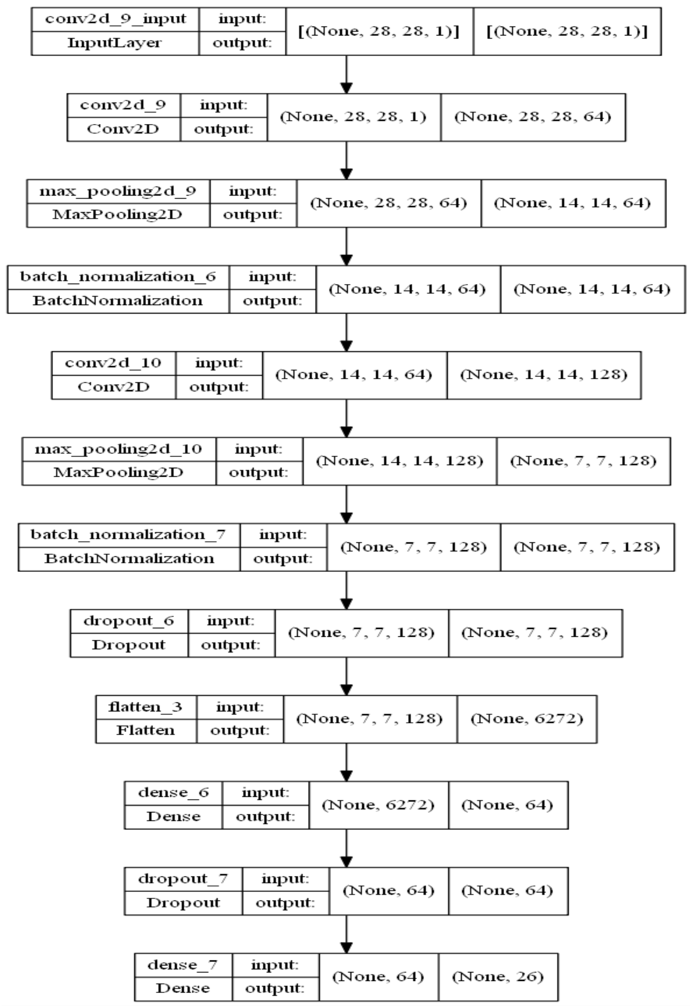


Figure 8: Figure: Model 3 - Two CNN Layers with Batch Normalization and Dropout followed by two Dense Layers

The obtained training, validation and testing accuracy of this model are:

Training Accuracy: 97.59%

Testing Accuracy: 87.05%

(iv) All of the three models have been compiled with the spare_categorical_crossentropy activation function, accuracy evaluation metric and adam optimizer with its default parameter values. Analysis on the results using the learning curve has been discussed below:

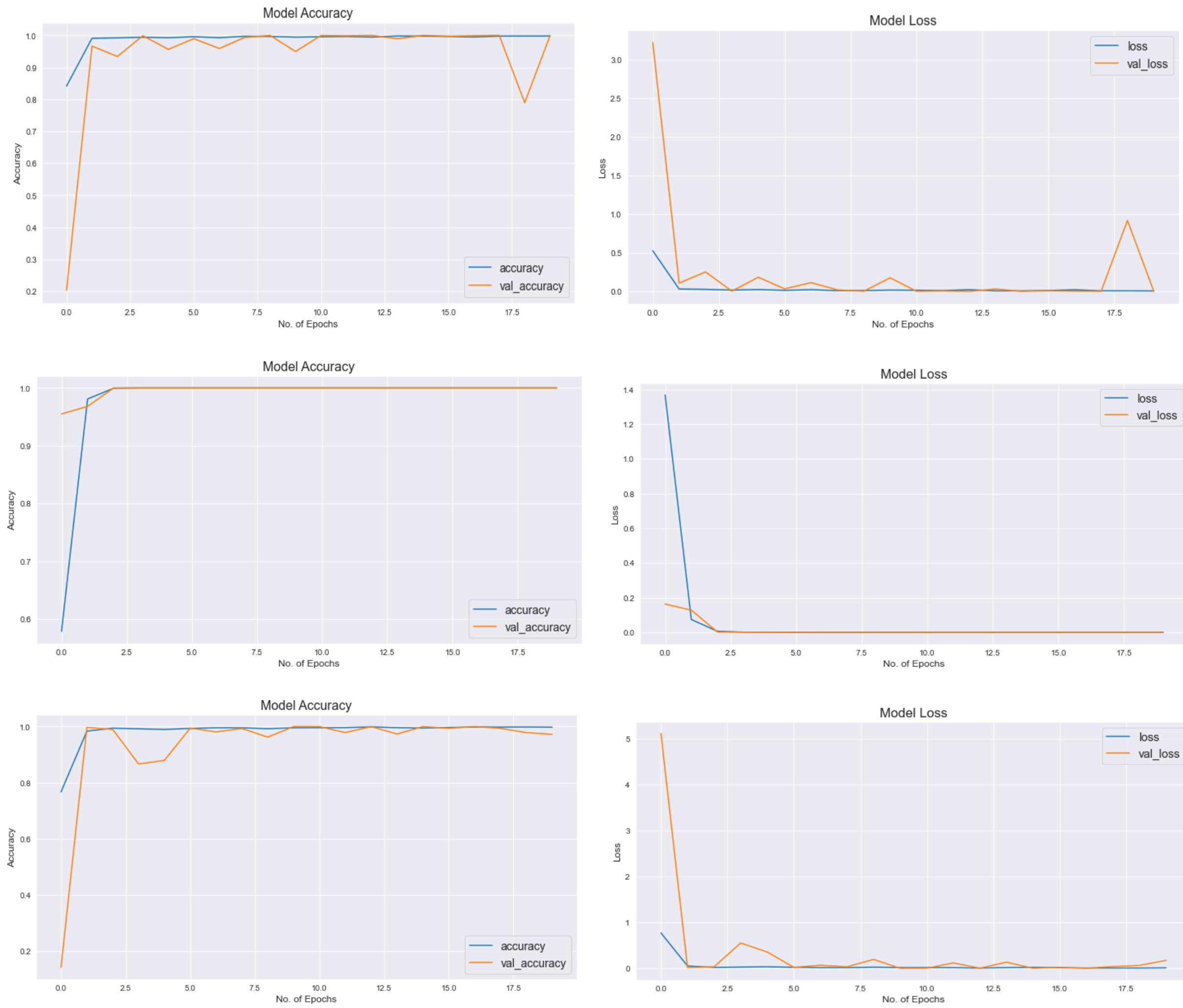


Figure 11: Learning Curve of Four Model (1st Row – Model 1, 2nd Row – Model 2, 3rd Row – Model 3)

From Figure 10, it can be noticed that Model 1has high validation loss at the beginning which gradually decreases with some spikes upon adding more training samples. The validation loss somewhat gets flatten around 10th epoch but had a high spike around 18th epoch which indicates that the model could keep improving to improve the performance on training dataset. This typically means the model is overfitting and unable to generalize to new data. One of the reasons behind it might be the complexity of this model as this model has got the highest number of layers among the four models with batch normalization and dropout. Though the model is overfitted, the model has been able to achieve the highest testing accuracy resulting to 97.59% on the test data.

From the loss curve of Model 2 and 3, it can be seen that both the models have high validation loss at the beginning which gradually decreases upon adding more training samples after 1st epoch. In case of Model 2, both the training and validation loss gets flatten and overlap from the 1st epoch which means the model is getting overfitted as the training is continuing. Due to overfitting, the model is unable to distinguish between irrelevant noise and relevant data that constitutes a pattern, which prevents it from making accurate predictions about new data. As a result, the model has only been able to achieve an accuracy of 91.09% on test data. Same is the case with model 3 as the model's validation loss gradually decreases with an increase at the last epoch and doesn't flatten which is an indication of an overfit model. Model 3 has achieved the lowest accuracy among the three models which is only 87.05% for the test data.

(v) The result of the three models have been shown in the following table:

Models	Accuracy
Model 1 – Three CNN Layers with Batch Normalization and Dropout followed by two Dense Layers	Training Accuracy: 100.00% Testing Accuracy: 97.59%
Model 2 - Three CNN Layers without Batch Normalization and Dropout followed by two Dense Layers	Training Accuracy: 100.00% Testing Accuracy: 91.09
Model 3 - Two CNN Layers with Batch Normalization and Dropout followed by two Dense Layers	Training Accuracy: 97.59% Testing Accuracy: 87.05%

All of the models used in (iii) consists of sequential 2D convolutional layers with different number of kernels and filters followed by flatten and dense layer. Among the three models I have used, the first one with three CNN layers followed by two dense layers with batch normalization and dropout has worked the best in case of generalizing the data points even though the model was overfitted. Both model 2 and 3 demonstrate slightly high variance as the accuracy of training data is quite higher than the accuracy of the testing data. This refers that models are gathering all of the training data's information, including extraneous noise, leading to high training data model accuracy but failure for new testing data.

As the model demonstrates better accuracy on both training and testing data, and slight overfitting, I will consider it as the benchmark model and work with this model in the next parts of the assignment.

Task 3

(i) For this task I have worked with two different optimizers, one is Stochastic Gradient Descent (SGD) and another one is Root Mean Squared Propagation (RMSProp). The obtained results using both of these optimizers have been discussed below:

SGD: I have worked with changing values two parameters of SGD optimizer which are learning_rate and momentum. The default learning rate of the SGD is 0.01 and momentum is 0.0. I have changed the learning_rate to 0.001 and momentum to 0.9. Using this parameters, it can be seen from Figure 12 that the validation loss goes lower than the training loss after the 1st epoch which indicates the overfitting to the training data and the model is struggling to generalize the validation data. But after the 1st epoch, both the training loss and validation loss flattens and overlaps. Using these parameter, I have got a training accuracy of 100.00% and 95.59% on the testing accuracy.

```
In [74]: score = model5.evaluate(X_train, Y_train, verbose=0)
         print("Training Accuracy: ", f'{score[1]*100:.2f}%')

         score = model5.evaluate(X_val, Y_val, verbose = 0)
         print("Validation Accuracy:", f'{score[1]*100:.2f}%')

         score = model5.evaluate(X_test, Y_test, verbose=0)
         print("Testing Accuracy:" , f'{score[1]*100:.2f}%')

Training Accuracy:  100.00%
Validation Accuracy: 100.00%
Testing Accuracy: 95.59%
```

Figure 12: Training, Testing and Validation Accuracy using SGD

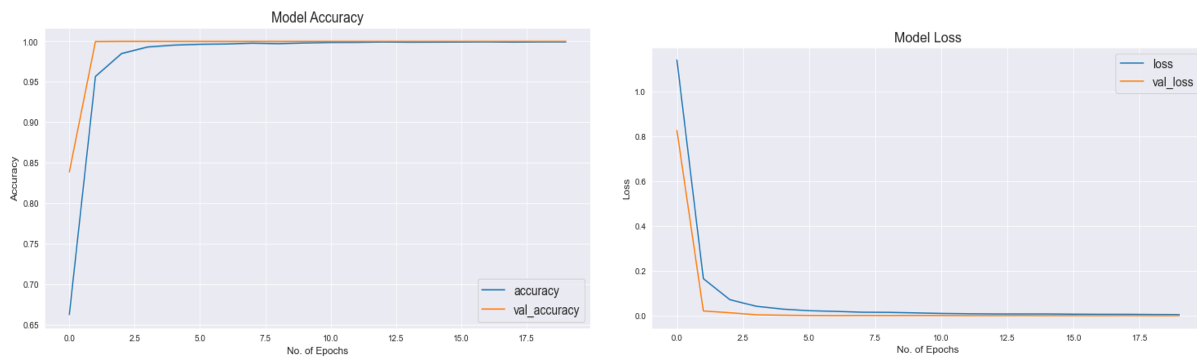


Figure 13: Learning Curve using SGD

RMSProp: I have worked with changing values two parameters of RMSProp optimizer which are learning_rate, rho and momentum. The default learning rate of the RMSProp is 0.001, rho = 0.9 and momentum is 0.0. I have changed the learning_rate to 0.01, rho = 0.95 and momentum to 0.9. From the learning curve in Figure 14, it can be noticed that the validation loss begin oscillation with divergence throughout the entire training which indicates that the learning rate is set too high. As a result, it is putting the model to go over the minima and prevent convergence which is known as exploding gradient. The model using RMSProp with these parameter's values yield a very training and testing accuracy which is only 4.29% and 4.02% respectively.

```
In [86]: score = model5.evaluate(X_train, Y_train, verbose=0)
         print("Training Accuracy: ", f'{score[1]*100:.2f}%')

         score = model5.evaluate(X_val, Y_val, verbose = 0)
         print("Validation Accuracy:", f'{score[1]*100:.2f}%')

         score = model5.evaluate(X_test, Y_test, verbose=0)
         print("Testing Accuracy:" , f'{score[1]*100:.2f}%')

         Training Accuracy:  4.29%
         Validation Accuracy: 3.99%
         Testing Accuracy: 4.02%
```

Figure 14: Training, Testing and Validation Accuracy using SGD

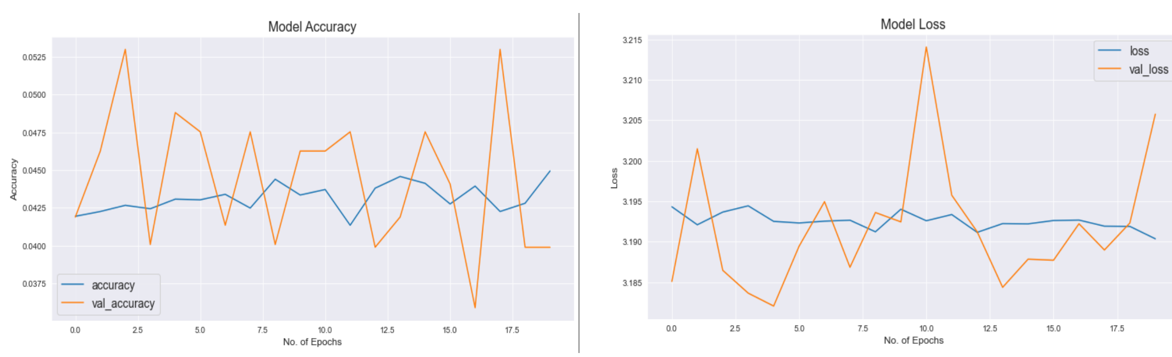


Figure 15: Learning Curve using RMSProp

(ii) The first optimizer that I used in this part is SGD which is iterative technique for maximizing an objective function with acceptable smoothness qualities. SGD is slower but generalizes well in the testing data which can be seen from the testing accuracy that we got using SGD. As the learning rate is set for 0.001 for SGD, it tries to cover a lot of data points which make the training slower but aid in making the model more generalized so that it can work well on the testing dataset. Also, the momentum value of 0.9 has aided in accelerating speed since it reduces the size of the steps, resulting in a higher effective learning rate in the directions of low curvature. On the other hand, RMSProp, with learning rate of 0.01, puts the model to go over the minima and prevent convergence resulting into exploding gradient. This is why the model showed very poor performance both on the training and testing dataset.

Task 4

(i) I have worked with the Model 1 from part 2 for this task. I have used Image Augmentation technique to improve the model both in case of overfitting and accuracy. Data augmentation is the straightforward process of increasing the amount of data to increase the number of images present in the training dataset for neural networks. Image data can be augmented using various techniques such as shifting, zooming in/out, rotating, flipping etc. These minor adjustments to the original image just offer a different angle for catching the object in real life; they do not alter the target class [12]. These image augmentation methods not only increase the size of your dataset but also add a degree of variation, which helps your model generalize more effectively to unobserved data which aid in combat overfitting as well.

(ii) For augmenting the original dataset, I have used Keras' built-in ImageDataGenerator class which provides a host of different augmentation techniques.

```
In [98]: data_gen=ImageDataGenerator(rotation_range = 0.2, zoom_range=0.2, width_shift_range=0.1, height_shift_range=0.1,
                                     horizontal_flip=True)
         data_gen.fit(X_train)
```

From the above screenshot, it is noticeable that I have used five parameters of the ImageDataGenerator class to augment the data which are:

- rotation_range: Rotates the images by 0.2 degree angle.
- zoom_range: As the zoom_range is set to 0.2 which is smaller than 1, the class will zoom in on the image.
- width_shift_range: This argument shifts the image horizontally. As I have used 0.1 for this argument, that would shift 1% of the width of each copy.
- height_shift_range: This argument shifts the image vertically. As I have used 0.1 for this argument, that would shift 1% of the height of each copy.
- horizontal_flip: Takes a Boolean argument for this parameter. As its set to True, the training datasets will be flipped along the horizontal axis.

Using the ImageDataGenerator, I have been able to achieve a training accuracy of 99.97% and validation accuracy of 99.91% and an accuracy of 99.74% when evaluated the model on the testing data. This is an improvement from the model that I had built in part 2 as after doing augmentation the model is more generalized and can classify the unseen data more precisely. Also, as the training and testing accuracy are a lot identical, this indicates that the model has been able to combat overfitting to a great extent. The training, validation and testing accuracy has been shown in the following screenshot:

```
In [135]: score = model6.evaluate(X_train, Y_train, verbose=0)
          print("Training Accuracy: ", f'{score[1]*100:.2f}%')

          score = model6.evaluate(X_val, Y_val, verbose = 0)
          print("Validation Accuracy:", f'{score[1]*100:.2f}%')

          score = model6.evaluate(X_test, Y_test, verbose=0)
          print("Testing Accuracy:" , f'{score[1]*100:.2f}%')

          Training Accuracy:  99.97%
          Validation Accuracy: 99.91%
          Testing Accuracy: 99.74%
```

(iii) After doing data augmentation, I have been able to get good accuracy on both training and testing data which refers to a low bias low variance situation. As image data augmentation is performed to increase the training dataset size and the model's effectiveness and generalizability, this has helped the model to perform more precisely for both training and testing data with respectable accuracy since it did not capture the noise present in the training data.

Summary of Results and Conclusions

Task 1

- The Sign Language MNIST dataset used for this is clean as there is no Null or NA values in the dataset.
- The pixel values are scaled between 0 and 1 which is achieved by diving all the pixel values by the highest pixel value 255. This turn the images into Gray Scale Images to combat difficulties during training, i.e. slower than anticipated model training.
- Also, I have to reshape the images into (28, 28, 1) as CNN takes input in this format where the first two dimension denotes the height and width of the images and the last one denotes the depth. As gray scale images are being used, the depth is set to 1.
- The dataset is already split into training and testing dataset for which there is no need to split the dataset into training and testing dataset. Instead, I have divide the training dataset into training and validation dataset with 80-20 split.

Task 2

- Accuracy has been chosen as an evaluation metric as the target variable in the training dataset is fairly balanced, totaling an average of around 1000 per class.
- Three different models have been used in this part with different numbers of CNN and fully connected layers with pooling, batch normalization and dropout. The result summary of the three models have been shown in the following table:

Models	**Accuracy**
<u>Model 1</u> – Three CNN Layers with Batch Normalization and Dropout followed by two Dense Layers	Training Accuracy: 100.00% Testing Accuracy: 97.59%
<u>Model 2</u> - Three CNN Layers without Batch Normalization and Dropout followed by two Dense Layers	Training Accuracy: 100.00% Testing Accuracy: 91.09
<u>Model 3</u> - Two CNN Layers with Batch Normalization and Dropout followed by two Dense Layers	Training Accuracy: 97.59% Testing Accuracy: 87.05%

- Among the three models, the first model demonstrates highest accuracy on the test dataset which means the model is more generalized and predict the unforeseen data more precisely. The other two model is highly overfitted for which I have more forward with the model 1 for Task 3 and 4.

Task 3

- Two different optimizers have been used for this task which are SGD and RMSProp.
- SGD, with the learning rate of 0.001 and momentum value of 0.9, yielded better accuracy on both training and testing which are 100% and 95.59% respectively but the model goes overfitted.
- RMSProp, learning_rate to 0.01, rho = 0.95 and momentum to 0.9, the model oscillates with divergence throughout the entire training which indicates that the learning rate is set too high. Also, the training and accuracy using RMSProp is so poor which is only 4.29% and 4.02% respectively.

Task 4

- I have used data augmentation using the ImageDataGenerator class of Keras to increase the training dataset so that the model can learn from more training data which can help in generalizing in unseen data.
- I have worked with five parameters form the ImageDataGenerator which are rotation_range, zoom_range, width_shift_range, height_shift_range and horizontal_shift to rotate, zoom in, shift and flip the training data horizontally respectively.
- Data augmentation has showed a better result on the testing data, yielding an accuracy of 99.74%.
- Additionally, the training and testing accuracy is almost identical which is an indication of combatting overfitting.

Reference:

[1] "Sign language mnist," *Kaggle*, 20-Oct-2017. [Online]. Available: https://www.kaggle.com/datasets/datamunge/sign-language-mnist?resource=download. [Accessed: 09-Nov-2022].

[2] A. Mannan, A. Abbasi, A. R. Javed, A. Ahsan, T. R. Gadekallu, and Q. Xin, "Hypertuned deep convolutional neural network for Sign language recognition," *Computational Intelligence and Neuroscience*, vol. 2022, pp. 1–10, 2022.

[3] Y. Ma, T. Xu, and K. Kim, "Two-stream mixed convolutional neural network for American Sign Language recognition," *Sensors*, vol. 22, no. 16, p. 5959, 2022.

[4] H. So, "Convolutional neural networks for sign language Classification." [Online]. Available: https://haleyso.github.io/projects/ASL.pdf. [Accessed: 09-Nov-2022].

[5] J. Brownlee, "How to manually scale image pixel data for deep learning," *Machine Learning Mastery*, 05-Jul-2019. [Online]. Available: https://machinelearningmastery.com/how-to-manually-scale-image-pixel-data-for-deep-learning/. [Accessed: 10-Nov-2022].

[6] S. Verma, "Understanding input output shapes in convolution neural network | Keras ...," 31-Aug-2019. [Online]. Available: https://towardsdatascience.com/understanding-input-and-output-shapes-in-convolution-network-keras-f143923d56ca. [Accessed: 10-Nov-2022].

[7] S. W. Smith, *The scientist and engineer's Guide to Digital Signal Processing*. San Diego, CA: California Technical Pub., 1997.

[8] IBM Cloud Education, “What are convolutional neural networks?,” *IBM*, 20-Oct-2020. [Online]. Available: https://www.ibm.com/cloud/learn/convolutional-neural-networks. [Accessed: 10-Nov-2022].

[9] J. Ramzai, “Top 10 model evaluation metrics for Classification ML Models.” [Online]. Available: https://towardsdatascience.com/top-10-model-evaluation-metrics-for-classification-ml-models-a0a0f1d51b9. [Accessed: 11-Nov-2022].

[10] I. Kim, “Demystifying batch normalization vs drop out,” *Medium*, 11-Oct-2021. [Online]. Available: https://medium.com/mlearning-ai/demystifying-batch-normalization-vs-drop-out-1c8310d9b516#:~:text=BN%20normalizes%20values%20of%20the,neural%20network%20to%20prevent%20overfitting. [Accessed: 12-Nov-2022].

[11] A. Rosebrock, “Softmax classifiers explained,” *PyImageSearch*, 17-Apr-2021. [Online]. Available: https://pyimagesearch.com/2016/09/12/softmax-classifiers-explained/. [Accessed: 12-Nov-2022].

[12] A. Bhandari, “Image augmentation keras: Keras imagedatagenerator,” *Analytics Vidhya*, 16-Aug-2020. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/08/image-augmentation-on-the-fly-using-keras-imagedatagenerator/. [Accessed: 13-Nov-2022].

[13] K. Team, “Keras Documentation: SGD,” *Keras*. [Online]. Available: https://keras.io/api/optimizers/sgd/. [Accessed: 14-Nov-2022].

[14] K. Team, “Keras Documentation: RMSprop,” *Keras*. [Online]. Available: https://keras.io/api/optimizers/rmsprop/. [Accessed: 14-Nov-2022].

[15] “Tf.keras.preprocessing.image.imagedatagenerator : tensorflow V2.10.0,” *TensorFlow*. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator. [Accessed: 14-Nov-2022].

CSCI 6515 - Assignment 3

In [1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import csv
```

Pre-processing the dataset

In [2]:
```python
train_data = pd.read_csv("sign_mnist_train.csv")
train_data.head()
```

Out[2]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780
0	3	107	118	127	134	139	143	146	150	153	...	207	207	207	207	206	206
1	6	155	157	156	156	156	157	156	158	158	...	69	149	128	87	94	163
2	2	187	188	188	187	187	186	187	188	187	...	202	201	200	199	198	199
3	2	211	211	212	212	211	210	211	210	210	...	235	234	233	231	230	226
4	13	164	167	170	172	176	179	180	184	185	...	92	105	105	108	133	163

5 rows × 785 columns

In [3]:
```python
train_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 27455 entries, 0 to 27454
Columns: 785 entries, label to pixel784
dtypes: int64(785)
memory usage: 164.4 MB
```

In [4]:
```python
train_data.describe()
```

Out[4]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pi
count	27455.000000	27455.000000	27455.000000	27455.000000	27455.000000	27455.000000	27455.000000	27455.000000	27455.000000	27455.00
mean	12.318813	145.419377	148.500273	151.247714	153.546531	156.210891	158.411255	160.472154	162.339683	163.95
std	7.287552	41.358555	39.942152	39.056286	38.595247	37.111165	36.125579	35.016392	33.661998	32.65
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
25%	6.000000	121.000000	126.000000	130.000000	133.000000	137.000000	140.000000	142.000000	144.000000	146.00
50%	13.000000	150.000000	153.000000	156.000000	158.000000	160.000000	162.000000	164.000000	165.000000	166.00
75%	19.000000	174.000000	176.000000	178.000000	179.000000	181.000000	182.000000	183.000000	184.000000	185.00
max	24.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.00

8 rows × 785 columns

In [5]:
```python
test_data= pd.read_csv("sign_mnist_test.csv")
test_data.head()
```

Out[5]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780
0	6	149	149	150	150	150	151	151	150	151	...	138	148	127	89	82	96
1	5	126	128	131	132	133	134	135	135	136	...	47	104	194	183	186	184
2	10	85	88	92	96	105	123	135	143	147	...	68	166	242	227	230	227
3	0	203	205	207	206	207	209	210	209	210	...	154	248	247	248	253	236
4	3	188	191	193	195	199	201	202	203	203	...	26	40	64	48	29	46

5 rows × 785 columns

In [6]:
```python
test_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7172 entries, 0 to 7171
Columns: 785 entries, label to pixel784
dtypes: int64(785)
memory usage: 43.0 MB
```

In [7]:
```python
test_data.describe()
```

Out[7]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...
count	7172.000000	7172.000000	7172.000000	7172.000000	7172.000000	7172.000000	7172.000000	7172.000000	7172.000000	7172.000000	...
mean	11.247351	147.532627	150.445761	153.324317	155.663413	158.169688	160.790853	162.282766	163.649191	165.589515	...
std	7.446712	43.593144	41.867838	40.442728	39.354776	37.749637	36.090916	36.212636	35.885378	33.721876	...
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	10.000000	0.000000	0.000000	0.000000	...
25%	4.000000	122.000000	126.000000	130.000000	134.000000	137.000000	141.000000	144.000000	145.000000	147.000000	...
50%	11.000000	154.000000	157.000000	159.000000	161.000000	163.000000	165.000000	166.000000	168.000000	169.000000	...
75%	18.000000	178.000000	179.000000	181.000000	182.000000	184.000000	185.000000	186.000000	187.000000	187.000000	...
max	24.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	255.000000	...

8 rows × 785 columns

In [8]:
```python
print("Total Null values in the Training Dataset:", train_data.isnull().values.sum())
print("Total NA values in the Training Dataset:", train_data.isna().values.sum())
```

```
Total Null values in the Training Dataset: 0
Total NA values in the Training Dataset: 0
```

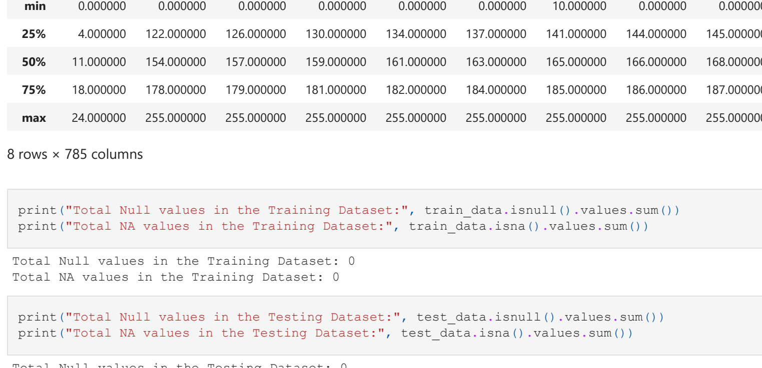
In [9]:
```python
print("Total Null values in the Testing Dataset:", test_data.isnull().values.sum())
print("Total NA values in the Testing Dataset:", test_data.isna().values.sum())
```

```
Total Null values in the Testing Dataset: 0
Total NA values in the Testing Dataset: 0
```

In [10]:
```python
plt.figure(figsize=(12,10))
sns.set_style("darkgrid")
ax = sns.countplot(train_data["label"])
ax.set_title("Distribution of Label in the Training Data", fontsize = 14)
```

```
C:\Users\User\anaconda3\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass the following variable as a keyword arg: x. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```

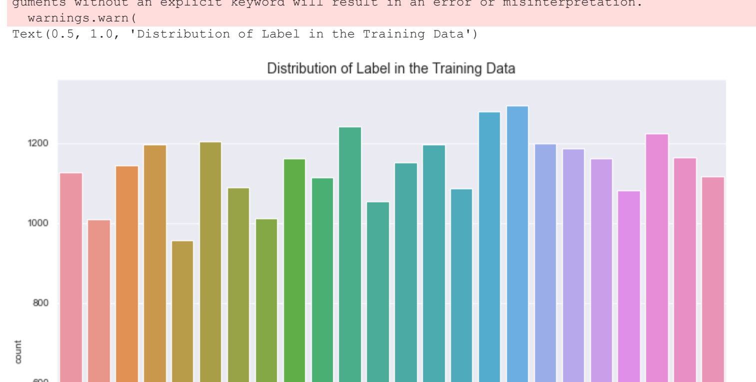
Out[10]:
```
Text(0.5, 1.0, 'Distribution of Label in the Training Data')
```



In [11]:
```python
plt.figure(figsize=(12,10))
sns.set_style("darkgrid")
ax = sns.countplot(test_data["label"])
ax.set_title("Distribution of Label in the Testing Data", fontsize = 14)
```

```
C:\Users\User\anaconda3\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass the following variable as a keyword arg: x. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```

Out[11]:
```
Text(0.5, 1.0, 'Distribution of Label in the Testing Data')
```



In [12]:
```python
#Seperating Feature variable and label from the training Dataset
Y_train = train_data["label"]
X_train = train_data.drop(["label"], axis = 1)
```

In [13]:
```python
#Seperating Feature variable and label from the testing Dataset
Y_test = test_data["label"]
X_test = test_data.drop(["label"], axis = 1)
```

In [14]:
```python
#Scaling the X values from Training and Testing Dataset
X_train /= 255
X_test /= 255
```

In [15]:
```python
print(X_train)
```

```
         pixel1    pixel2    pixel3    pixel4    pixel5    pixel6    pixel7  \
0      0.419608  0.462745  0.498039  0.525490  0.545098  0.560784  0.572549
1      0.607843  0.615686  0.611765  0.611765  0.611765  0.615686  0.611765
2      0.733333  0.737255  0.737255  0.733333  0.733333  0.729412  0.733333
3      0.827451  0.827451  0.831373  0.831373  0.827451  0.823529  0.827451
4      0.643137  0.654902  0.666667  0.674510  0.690196  0.701961  0.705882
...         ...       ...       ...       ...       ...       ...       ...
27450  0.741176  0.741176  0.745098  0.745098  0.752941  0.756863  0.756863
27451  0.592157  0.603922  0.615686  0.619608  0.627451  0.631373  0.639216
27452  0.682353  0.682353  0.682353  0.682353  0.682353  0.686275  0.686275
27453  0.694118  0.709804  0.721569  0.725490  0.733333  0.741176  0.745098
27454  0.701961  0.705882  0.705882  0.705882  0.713725  0.709804  0.713725

         pixel8    pixel9   pixel10  ...  pixel775  pixel776  pixel777  \
0      0.588235  0.600000  0.611765  ...  0.811765  0.811765  0.811765
1      0.619608  0.619608  0.615686  ...  0.270588  0.584314  0.501961
2      0.737255  0.733333  0.729412  ...  0.792157  0.788235  0.784314
3      0.823529  0.823529  0.827451  ...  0.921569  0.917647  0.913725
4      0.721569  0.725490  0.729412  ...  0.360784  0.411765  0.411765
...         ...       ...       ...  ...       ...       ...       ...
27450  0.756863  0.756863  0.760784  ...  0.517647  0.647059  0.388235
27451  0.643137  0.650980  0.654902  ...  0.776471  0.776471  0.776471
27452  0.682353  0.678431  0.678431  ...  0.474510  0.768627  0.819608
27453  0.749020  0.749020  0.745098  ...  0.466667  0.219608  0.105882
27454  0.717647  0.713725  0.713725  ...  0.423529  0.517647  0.666667

       pixel778  pixel779  pixel780  pixel781  pixel782  pixel783  pixel784
0      0.811765  0.807843  0.807843  0.807843  0.800000  0.796078  0.792157
1      0.341176  0.368627  0.639216  0.686275  0.403922  0.529412  0.584314
2      0.780392  0.776471  0.780392  0.776471  0.764706  0.760784  0.764706
3      0.905882  0.901961  0.886275  0.882353  0.870588  0.898039  0.639216
4      0.423529  0.521569  0.639216  0.615686  0.639216  0.643137  0.701961
...         ...       ...       ...       ...       ...       ...       ...
27450  0.301961  0.203922  0.784314  0.917647  0.784314  0.870588  0.882353
27451  0.776471  0.776471  0.768627  0.764706  0.764706  0.764706  0.760784
27452  0.815686  0.807843  0.800000  0.796078  0.792157  0.784314  0.784314
27453  0.227451  0.400000  0.309804  0.184314  0.250980  0.341176  0.364706
27454  0.760784  0.839216  0.796078  0.772549  0.803922  0.819608  0.843137

[27455 rows x 784 columns]
```

In [16]:
```python
print(X_test)
```

```
        pixel1    pixel2    pixel3    pixel4    pixel5    pixel6    pixel7  \
0     0.584314  0.584314  0.588235  0.588235  0.588235  0.592157  0.592157
1     0.494118  0.501961  0.513725  0.517647  0.521569  0.525490  0.529412
2     0.333333  0.345098  0.360784  0.376471  0.411765  0.482353  0.529412
3     0.796078  0.803922  0.811765  0.807843  0.811765  0.819608  0.823529
4     0.737255  0.749020  0.756863  0.764706  0.780392  0.788235  0.792157
...        ...       ...       ...       ...       ...       ...       ...
7167  0.529412  0.466667  0.423529  0.400000  0.411765  0.388235  0.239216
7168  0.615686  0.623529  0.631373  0.643137  0.650980  0.650980  0.670588
7169  0.745098  0.749020  0.745098  0.749020  0.745098  0.745098  0.752941
7170  0.788235  0.803922  0.815686  0.819608  0.839216  0.847059  0.854902
7171  0.678431  0.682353  0.678431  0.682353  0.678431  0.678431  0.686275

        pixel8    pixel9   pixel10  ...  pixel775  pixel776  pixel777  \
0     0.588235  0.592157  0.596078  ...  0.541176  0.580392  0.498039
1     0.529412  0.533333  0.541176  ...  0.184314  0.407843  0.760784
2     0.560784  0.576471  0.596078  ...  0.266667  0.650980  0.949020
3     0.819608  0.823529  0.819608  ...  0.603922  0.972549  0.968627
4     0.796078  0.796078  0.796078  ...  0.101961  0.156863  0.250980
...        ...       ...       ...  ...       ...       ...       ...
7167  0.403922  0.474510  0.521569  ...  0.423529  0.439216  0.454902
7168  0.682353  0.686275  0.690196  ...  0.835294  0.835294  0.835294
7169  0.752941  0.749020  0.752941  ...  0.847059  0.843137  0.835294
7170  0.874510  0.886275  0.898039  ...  0.439216  0.662745  1.000000
7171  0.686275  0.682353  0.686275  ...  0.788235  0.784314  0.772549

      pixel778  pixel779  pixel780  pixel781  pixel782  pixel783  pixel784
0     0.349020  0.321569  0.376471  0.415686  0.439216  0.470588  0.419608
1     0.717647  0.729412  0.721569  0.721569  0.721569  0.713725  0.705882
2     0.890196  0.901961  0.890196  0.886275  0.882353  0.878431  0.870588
3     0.972549  0.992157  0.925490  0.901961  0.941176  0.992157  1.000000
4     0.188235  0.113725  0.180392  0.192157  0.180392  0.180392  0.207843
...        ...       ...       ...       ...       ...       ...       ...
7167  0.447059  0.462745  0.705882  0.721569  0.690196  0.654902  0.639216
7168  0.839216  0.835294  0.827451  0.823529  0.823529  0.819608  0.815686
7169  0.839216  0.839216  0.835294  0.823529  0.827451  0.819608  0.815686
7170  1.000000  0.929412  0.443137  0.356863  0.262745  0.274510  0.247059
7171  0.776471  0.776471  0.772549  0.764706  0.764706  0.756863  0.752941

[7172 rows x 784 columns]
```

In [17]:
```python
X_train = X_train.values.reshape(X_train.shape[0],28,28,1)
```

In [18]:
```python
X_test = X_test.values.reshape(X_test.shape[0],28,28,1)
```

In [19]:
```python
print(X_train.shape)
print(X_test.shape)
```

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

In [20]:
```python
fig, axes = plt.subplots(figsize=(14, 12))

for i in range(12):
    plt.subplot(4,4,i+1)
    plt.imshow(X_train[i], cmap="gray")
    plt.subplots_adjust(hspace = 0.4)

plt.show()
```

In [21]:
```python
fig, axes = plt.subplots(figsize=(14, 12))

for i in range(12):
    plt.subplot(4,4,i+1)
    plt.imshow(X_test[i], cmap="gray")
    plt.subplots_adjust(hspace = 0.4)

plt.show()
```

In [22]:
```python
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score
from sklearn.metrics import classification_report
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.layers import BatchNormalization
from keras.layers import BatchNormalization
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, Conv2D, MaxPooling2D, GlobalAveragePooling2D
from timeit import Timer
```

Model 1: Three CNN layers with BatchNormalization and Dropout followed by Two Dense Layers

In [23]:
```python
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size = 0.2, random_state = 42)
```

In [24]:
```python
print(X_train.shape)
print(Y_train.shape)
print(X_val.shape)
print(Y_val.shape)
print(X_test.shape)
print(Y_test.shape)
```

```
(21964, 28, 28, 1)
(21964,)
(5491, 28, 28, 1)
(5491,)
(7172, 28, 28, 1)
(7172,)
```

In [25]:
```python
model1 = Sequential()

model1.add(Conv2D(32, kernel_size = 5, strides = 1, padding = "same", activation = "relu", input_shape = (28, 2
model1.add(MaxPooling2D(padding = "same"))
model1.add(BatchNormalization())

model1.add(Conv2D(64, kernel_size = 5, strides = 1, padding = "same", activation = "relu"))
model1.add(MaxPooling2D(padding = "same"))
model1.add(BatchNormalization())
model1.add(Dropout(0.2))

model1.add(Conv2D(128, kernel_size = 5, strides = 1, padding = "same", activation = "relu"))
model1.add(MaxPooling2D(padding = "same"))
model1.add(BatchNormalization())
model1.add(Dropout(0.2))

model1.add(Flatten())

model1.add(Dense(64, activation = "relu"))
model1.add(Dropout(0.2))

model1.add(Dense(26, activation = "softmax"))
```

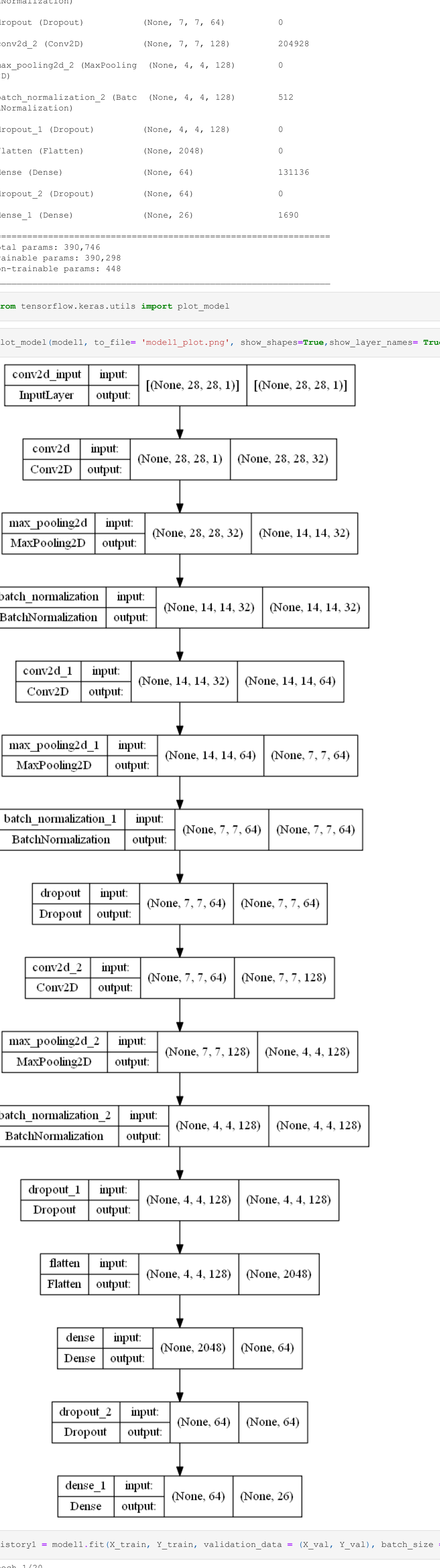
In [26]:
```python
model1.compile(loss="sparse_categorical_crossentropy", metrics = "accuracy", optimizer = "adam")
model1.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 32)        832

 max_pooling2d (MaxPooling2D  (None, 14, 14, 32)       0
 )

 batch_normalization (BatchN  (None, 14, 14, 32)       128
 ormalization)

 conv2d_1 (Conv2D)           (None, 14, 14, 64)        51264

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

 batch_normalization_1 (Batc  (None, 7, 7, 64)         256
 hNormalization)

 dropout (Dropout)           (None, 7, 7, 64)          0

 conv2d_2 (Conv2D)           (None, 7, 7, 128)         204928

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 batch_normalization_2 (Batc  (None, 4, 4, 128)        512
 hNormalization)

 dropout_1 (Dropout)         (None, 4, 4, 128)         0

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 64)                131136

 dropout_2 (Dropout)         (None, 64)                0

 dense_1 (Dense)             (None, 26)                1690

=================================================================
Total params: 390,746
Trainable params: 390,298
Non-trainable params: 448
_________________________________________________________________
```

In [27]:
```python
from tensorflow.keras.utils import plot_model
```

In [28]:
```python
plot_model(model1, to_file= 'model1_plot.png', show_shapes=True,show_layer_names= True)
```

Out[28]:



In [29]:
```python
history1 = model1.fit(X_train, Y_train, validation_data = (X_val, Y_val), batch_size = 50, epochs = 20)
```

```
Epoch 1/20
440/440 [==============================] - 232s 520ms/step - loss: 0.5255 - accuracy: 0.8413 - val_loss: 3.2208
- val_accuracy: 0.2029
Epoch 2/20
440/440 [==============================] - 227s 516ms/step - loss: 0.0299 - accuracy: 0.9909 - val_loss: 0.1071
- val_accuracy: 0.9665
Epoch 3/20
440/440 [==============================] - 231s 525ms/step - loss: 0.0260 - accuracy: 0.9922 - val_loss: 0.2510
- val_accuracy: 0.9341
Epoch 4/20
440/440 [==============================] - 222s 505ms/step - loss: 0.0164 - accuracy: 0.9943 - val_loss: 0.0030
- val_accuracy: 0.9991
Epoch 5/20
440/440 [==============================] - 231s 524ms/step - loss: 0.0237 - accuracy: 0.9926 - val_loss: 0.1827
- val_accuracy: 0.9563
Epoch 6/20
440/440 [==============================] - 228s 517ms/step - loss: 0.0132 - accuracy: 0.9959 - val_loss: 0.0321
- val_accuracy: 0.9898
Epoch 7/20
440/440 [==============================] - 225s 511ms/step - loss: 0.0221 - accuracy: 0.9928 - val_loss: 0.1132
- val_accuracy: 0.9590
Epoch 8/20
440/440 [==============================] - 226s 514ms/step - loss: 0.0090 - accuracy: 0.9973 - val_loss: 0.0192
- val_accuracy: 0.9936
Epoch 9/20
440/440 [==============================] - 229s 520ms/step - loss: 0.0100 - accuracy: 0.9968 - val_loss: 8.3662
e-04 - val_accuracy: 0.9998
Epoch 10/20
440/440 [==============================] - 227s 515ms/step - loss: 0.0167 - accuracy: 0.9944 - val_loss: 0.1754
- val_accuracy: 0.9497
Epoch 11/20
440/440 [==============================] - 226s 515ms/step - loss: 0.0142 - accuracy: 0.9958 - val_loss: 9.4319
e-04 - val_accuracy: 0.9996
Epoch 12/20
440/440 [==============================] - 232s 528ms/step - loss: 0.0098 - accuracy: 0.9966 - val_loss: 0.0045
- val_accuracy: 0.9987
Epoch 13/20
440/440 [==============================] - 231s 525ms/step - loss: 0.0215 - accuracy: 0.9947 - val_loss: 9.1220
e-06 - val_accuracy: 1.0000
Epoch 14/20
440/440 [==============================] - 232s 528ms/step - loss: 0.0058 - accuracy: 0.9981 - val_loss: 0.0302
- val_accuracy: 0.9894
Epoch 15/20
440/440 [==============================] - 232s 526ms/step - loss: 0.0065 - accuracy: 0.9977 - val_loss: 1.8072
e-05 - val_accuracy: 1.0000
Epoch 16/20
440/440 [==============================] - 224s 510ms/step - loss: 0.0110 - accuracy: 0.9966 - val_loss: 0.0067
- val_accuracy: 0.9975
Epoch 17/20
440/440 [==============================] - 229s 520ms/step - loss: 0.0205 - accuracy: 0.9950 - val_loss: 0.0026
- val_accuracy: 0.9991
Epoch 18/20
440/440 [==============================] - 227s 515ms/step - loss: 0.0069 - accuracy: 0.9978 - val_loss: 1.7715
e-05 - val_accuracy: 1.0000
Epoch 19/20
440/440 [==============================] - 238s 540ms/step - loss: 0.0065 - accuracy: 0.9980 - val_loss: 0.9173
- val_accuracy: 0.7895
Epoch 20/20
440/440 [==============================] - 224s 509ms/step - loss: 0.0053 - accuracy: 0.9980 - val_loss: 1.8502
e-06 - val_accuracy: 1.0000
```

In [30]:
```python
train_pred = model1.predict(X_test)
train_pred = np.argmax(train_pred,axis = 1)
```

In [31]:
```python
print(classification_report(Y_test, train_pred))
```

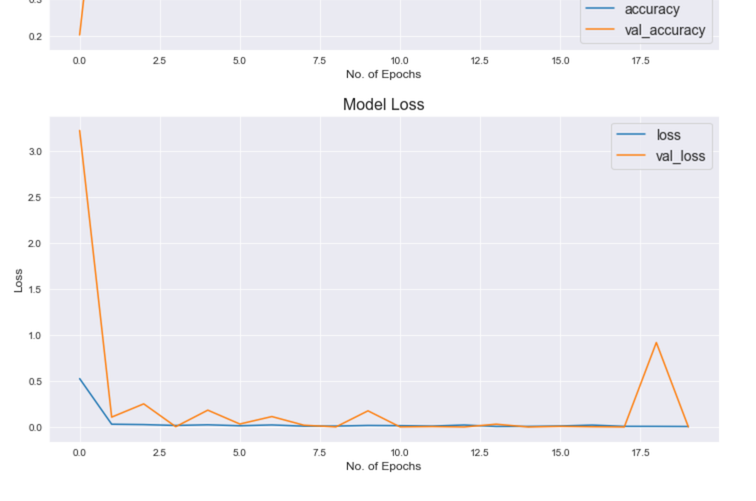
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       331
           1       1.00      1.00      1.00       432
           2       1.00      1.00      1.00       310
           3       1.00      1.00      1.00       245
           4       1.00      1.00      1.00       498
           5       1.00      1.00      1.00       247
           6       0.88      0.98      0.93       348
           7       0.96      0.94      0.95       436
           8       1.00      1.00      1.00       288
          10       0.94      0.95      0.95       331
          11       0.96      1.00      0.98       209
          12       0.96      1.00      0.98       394
          13       1.00      0.93      0.96       291
          14       1.00      0.99      1.00       246
          15       0.97      1.00      0.99       347
          16       1.00      0.99      1.00       164
          17       0.89      0.78      0.83       144
          18       1.00      1.00      1.00       246
          19       0.99      0.80      0.89       248
          20       0.93      1.00      0.96       266
          21       0.96      1.00      0.98       346
          22       1.00      1.00      1.00       206
          23       1.00      1.00      1.00       267
          24       1.00      0.94      0.97       332

    accuracy                           0.98      7172
   macro avg       0.98      0.97      0.97      7172
weighted avg       0.98      0.98      0.98      7172
```

In [32]:
```python
score = model1.evaluate(X_train, Y_train, verbose=0)
print("Training Accuracy: ", f'{score[1]*100:.2f}%')

score = model1.evaluate(X_val, Y_val, verbose = 0)
print("Validation Accuracy:", f'{score[1]*100:.2f}%')

score = model1.evaluate(X_test, Y_test, verbose=0)
print("Testing Accuracy:" , f'{score[1]*100:.2f}%')
```

```
Training Accuracy:  100.00%
Validation Accuracy: 100.00%
Testing Accuracy: 97.59%
```

In [33]:
```python
def plot_accuracy():
    plt.plot(history1.history['accuracy'])
    plt.plot(history1.history['val_accuracy'])
    plt.title('Model Accuracy', fontsize = 16)
    plt.ylabel('Accuracy', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['accuracy', 'val_accuracy'], fontsize = 14)
    plt.show()

def plot_loss():
    plt.plot(history1.history['loss'])
    plt.plot(history1.history['val_loss'])
    plt.title('Model Loss', fontsize = 16)
    plt.ylabel('Loss', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['loss', 'val_loss'], fontsize = 14)
    plt.show()
```

In [34]:
```python
plt.figure(figsize=(12,6))
plot_accuracy()
plt.figure(figsize=(12,6))
plot_loss()
```



Model 2 - Three CNN layers without Batch Normalization and Dropout followed by Two Dense Layers

In [35]:
```python
model2 = Sequential()

model2.add(Conv2D(32, kernel_size = 5, strides = 1, padding = "same", activation = "relu", input_shape = (28, 2
model2.add(MaxPooling2D(padding = "same"))

model2.add(Conv2D(64, kernel_size = 5, strides = 1, padding = "same", activation = "relu"))
model2.add(MaxPooling2D(padding = "same"))

model2.add(Conv2D(128, kernel_size = 5, strides = 1, padding = "same", activation = "relu"))
model2.add(MaxPooling2D(padding = "same"))

model2.add(Flatten())

model2.add(Dense(64, activation = "relu"))

model2.add(Dense(26, activation = "softmax"))
```

In [36]:
```python
model2.compile(loss="sparse_categorical_crossentropy", metrics = "accuracy", optimizer = "adam")
model2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 28, 28, 32)        832

 max_pooling2d_3 (MaxPooling  (None, 14, 14, 32)       0
 2D)

 conv2d_4 (Conv2D)           (None, 14, 14, 64)        51264

 max_pooling2d_4 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 conv2d_5 (Conv2D)           (None, 7, 7, 128)         204928

 max_pooling2d_5 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 flatten_1 (Flatten)         (None, 2048)              0

 dense_2 (Dense)             (None, 64)                131136

 dense_3 (Dense)             (None, 26)                1690

=================================================================
Total params: 389,850
Trainable params: 389,850
Non-trainable params: 0
_________________________________________________________________
```

In [37]:

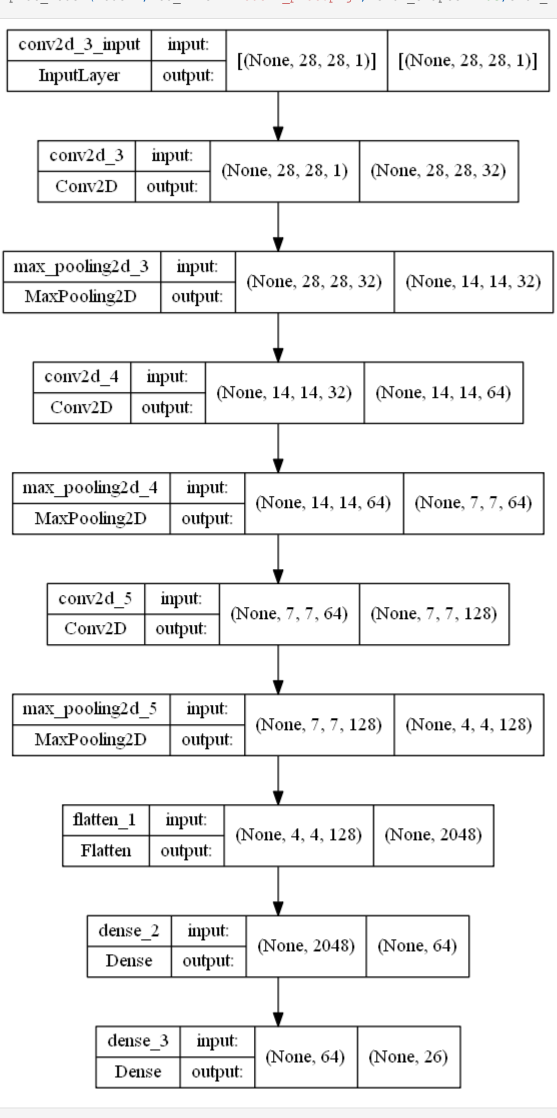
```
history2 = model2.fit(X_train, Y_train, validation_data = (X_val, Y_val), batch_size = 50, epochs = 20)
```

```
Epoch 1/20
440/440 [==============================] - 209s 470ms/step - loss: 1.3681 - accuracy: 0.5792 - val_loss: 0.1630
- val_accuracy: 0.9550
Epoch 2/20
440/440 [==============================] - 208s 472ms/step - loss: 0.0737 - accuracy: 0.9809 - val_loss: 0.1276
- val_accuracy: 0.9681
Epoch 3/20
440/440 [==============================] - 209s 475ms/step - loss: 0.0060 - accuracy: 0.9995 - val_loss: 0.0016
- val_accuracy: 1.0000
Epoch 4/20
440/440 [==============================] - 212s 483ms/step - loss: 8.5921e-04 - accuracy: 1.0000 - val_loss: 7.
7005e-04 - val_accuracy: 1.0000
Epoch 5/20
440/440 [==============================] - 208s 472ms/step - loss: 5.2120e-04 - accuracy: 1.0000 - val_loss: 4.
1853e-04 - val_accuracy: 1.0000
Epoch 6/20
440/440 [==============================] - 206s 469ms/step - loss: 2.7601e-04 - accuracy: 1.0000 - val_loss: 2.
7049e-04 - val_accuracy: 1.0000
Epoch 7/20
440/440 [==============================] - 207s 470ms/step - loss: 1.7686e-04 - accuracy: 1.0000 - val_loss: 1.
8877e-04 - val_accuracy: 1.0000
Epoch 8/20
440/440 [==============================] - 212s 482ms/step - loss: 1.2822e-04 - accuracy: 1.0000 - val_loss: 1.
3185e-04 - val_accuracy: 1.0000
Epoch 9/20
440/440 [==============================] - 210s 477ms/step - loss: 8.5963e-05 - accuracy: 1.0000 - val_loss: 9.
6989e-05 - val_accuracy: 1.0000
Epoch 10/20
440/440 [==============================] - 214s 486ms/step - loss: 6.1300e-05 - accuracy: 1.0000 - val_loss: 7.
5731e-05 - val_accuracy: 1.0000
Epoch 11/20
440/440 [==============================] - 211s 479ms/step - loss: 4.4810e-05 - accuracy: 1.0000 - val_loss: 5.
1805e-05 - val_accuracy: 1.0000
Epoch 12/20
440/440 [==============================] - 211s 481ms/step - loss: 3.2772e-05 - accuracy: 1.0000 - val_loss: 4.
2298e-05 - val_accuracy: 1.0000
Epoch 13/20
440/440 [==============================] - 210s 477ms/step - loss: 2.4967e-05 - accuracy: 1.0000 - val_loss: 3.
0753e-05 - val_accuracy: 1.0000
Epoch 14/20
440/440 [==============================] - 221s 503ms/step - loss: 1.9431e-05 - accuracy: 1.0000 - val_loss: 2.
3327e-05 - val_accuracy: 1.0000
Epoch 15/20
440/440 [==============================] - 215s 490ms/step - loss: 1.3756e-05 - accuracy: 1.0000 - val_loss: 1.
6832e-05 - val_accuracy: 1.0000
Epoch 16/20
440/440 [==============================] - 211s 479ms/step - loss: 1.0087e-05 - accuracy: 1.0000 - val_loss: 1.
3721e-05 - val_accuracy: 1.0000
Epoch 17/20
440/440 [==============================] - 213s 484ms/step - loss: 7.7515e-06 - accuracy: 1.0000 - val_loss: 9.
7006e-06 - val_accuracy: 1.0000
Epoch 18/20
440/440 [==============================] - 217s 494ms/step - loss: 5.9322e-06 - accuracy: 1.0000 - val_loss: 7.
8417e-06 - val_accuracy: 1.0000
Epoch 19/20
440/440 [==============================] - 220s 500ms/step - loss: 4.5405e-06 - accuracy: 1.0000 - val_loss: 6.
8866e-06 - val_accuracy: 1.0000
Epoch 20/20
440/440 [==============================] - 221s 503ms/step - loss: 3.4688e-06 - accuracy: 1.0000 - val_loss: 5.
0015e-06 - val_accuracy: 1.0000
```

In [38]:

```
plot_model(model2, to_file= 'model2_plot.png', show_shapes=True,show_layer_names= True)
```

Out[38]:



In [39]:

```
score = model2.evaluate(X_train, Y_train, verbose=0)
print("Training Accuracy: ", f'{score[1]*100:.2f}%')

#score = model1.evaluate(X_val, Y_val, verbose = 0)
#print("Validation Accuracy:", f'{score[1]*100:.2f}%')

score = model2.evaluate(X_test, Y_test, verbose=0)
print("Testing Accuracy:" , f'{score[1]*100:.2f}%')
```

```
Training Accuracy:  100.00%
Testing Accuracy: 91.09%
```

In [40]:

```
train_pred = model2.predict(X_test)
train_pred = np.argmax(train_pred,axis = 1)
```

In [41]:

```
print(classification_report(Y_test, train_pred))
```

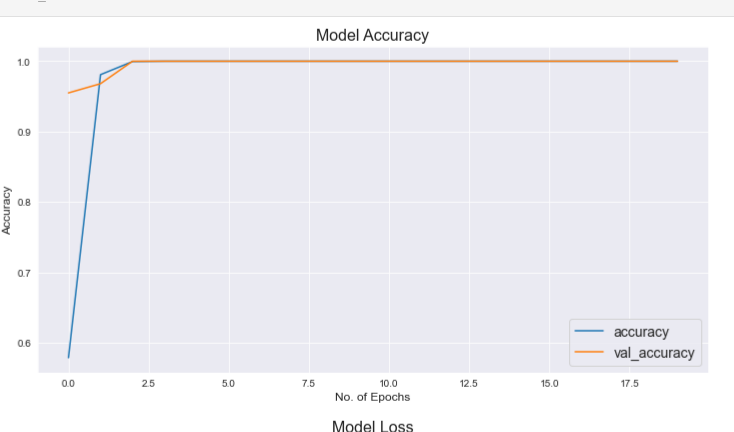
```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       331
           1       1.00      0.94      0.97       432
           2       1.00      0.94      0.97       310
           3       0.87      0.90      0.89       245
           4       0.91      0.99      0.95       498
           5       0.93      1.00      0.96       247
           6       0.84      0.89      0.86       348
           7       0.96      0.95      0.96       436
           8       1.00      0.99      0.99       288
          10       0.83      0.81      0.82       331
          11       0.87      1.00      0.93       209
          12       0.91      0.89      0.90       394
          13       0.86      0.76      0.81       291
          14       0.99      0.83      0.90       246
          15       0.95      1.00      0.97       347
          16       1.00      1.00      1.00       164
          17       0.69      0.84      0.76       144
          18       0.77      0.77      0.77       246
          19       0.88      0.71      0.79       248
          20       1.00      0.82      0.90       266
          21       0.90      0.83      0.86       346
          22       0.74      0.91      0.82       206
          23       0.86      0.97      0.92       267
          24       0.99      0.99      0.99       332

    accuracy                           0.91      7172
   macro avg       0.91      0.91      0.90      7172
weighted avg       0.91      0.91      0.91      7172
```

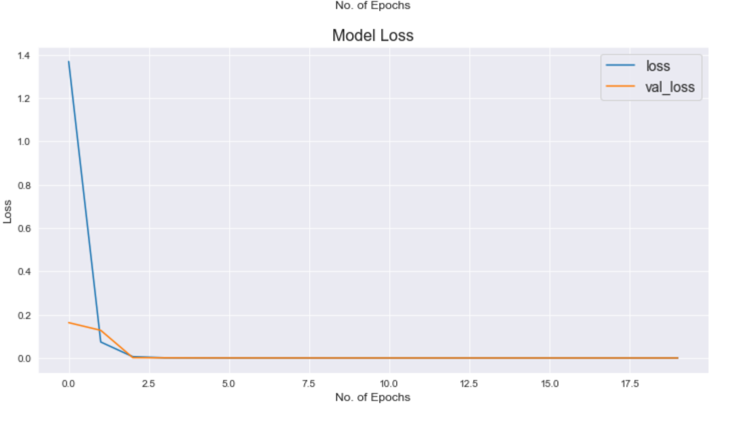
In [42]:

```
def plot_accuracy():
    plt.plot(history2.history['accuracy'])
    plt.plot(history2.history['val_accuracy'])
    plt.title('Model Accuracy', fontsize = 16)
    plt.ylabel('Accuracy', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['accuracy', 'val_accuracy'], fontsize = 14)
    plt.show()

def plot_loss():
    plt.plot(history2.history['loss'])
    plt.plot(history2.history['val_loss'])
    plt.title('Model Loss', fontsize = 16)
    plt.ylabel('Loss', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['loss', 'val_loss'], fontsize = 14)
    plt.show()
```

In [43]:

```
plt.figure(figsize=(12,6))
plot_accuracy()
plt.figure(figsize=(12,6))
plot_loss()
```





Model 3 - Two CNN Layers with Batch Normalization and Dropout followed by Two Dense Layers

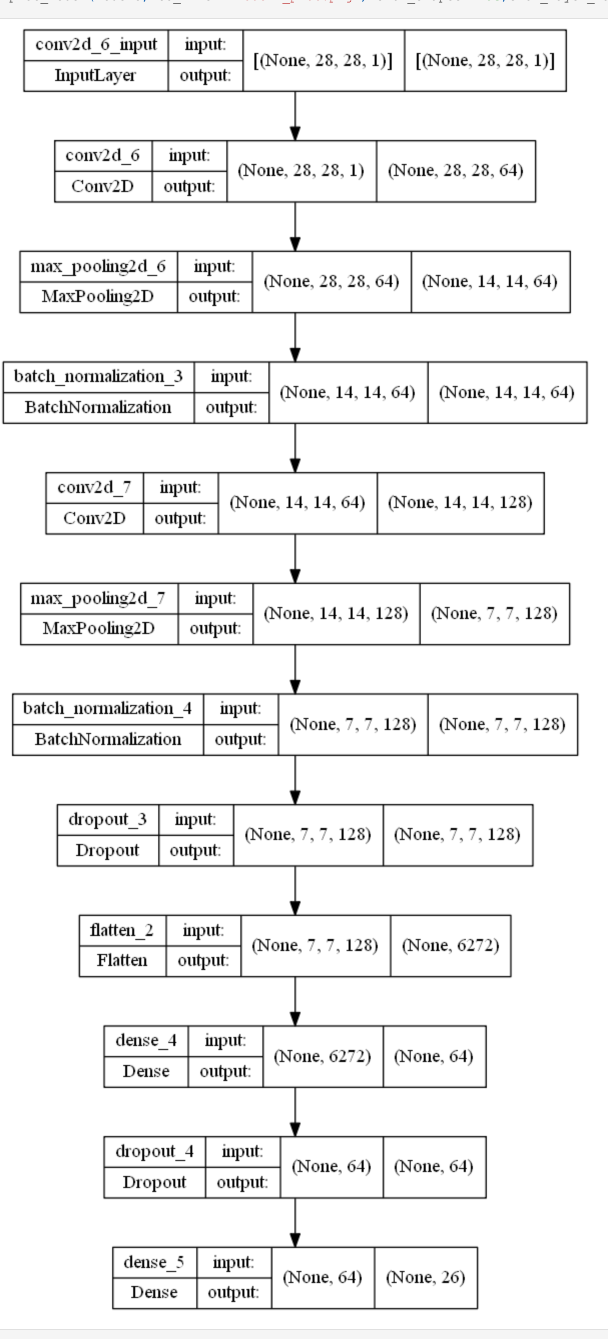
In [44]:

```
model3 = Sequential()

model3.add(Conv2D(64, kernel_size = 5, strides = 1, padding = "same", activation = "relu", input_shape = (28, 2
model3.add(MaxPooling2D(padding = "same"))
model3.add(BatchNormalization())

model3.add(Conv2D(128, kernel_size = 5, strides = 1, padding = "same", activation = "relu"))
model3.add(MaxPooling2D(padding = "same"))
model3.add(BatchNormalization())
model3.add(Dropout(0.2))

model3.add(Flatten())

model3.add(Dense(64, activation = "relu"))
model3.add(Dropout(0.2))

model3.add(Dense(26, activation = "softmax"))
```

In [45]:

```
plot_model(model3, to_file= 'model2_plot.png', show_shapes=True,show_layer_names= True)
```

Out[45]:



In [46]:

```
model3.compile(loss="sparse_categorical_crossentropy", metrics = "accuracy", optimizer = "adam")
model3.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 28, 28, 64)        1664

 max_pooling2d_6 (MaxPooling  (None, 14, 14, 64)       0
 2D)

 batch_normalization_3 (Batc  (None, 14, 14, 64)       256
 hNormalization)

 conv2d_7 (Conv2D)           (None, 14, 14, 128)       204928

 max_pooling2d_7 (MaxPooling  (None, 7, 7, 128)        0
 2D)

 batch_normalization_4 (Batc  (None, 7, 7, 128)        512
 hNormalization)

 dropout_3 (Dropout)         (None, 7, 7, 128)         0

 flatten_2 (Flatten)         (None, 6272)              0

 dense_4 (Dense)             (None, 64)                401472

 dropout_4 (Dropout)         (None, 64)                0

 dense_5 (Dense)             (None, 26)                1690

=================================================================
Total params: 610,522
Trainable params: 610,138
Non-trainable params: 384
_________________________________________________________________
```

In [47]:

```
history3 = model3.fit(X_train, Y_train, validation_data = (X_val, Y_val), batch_size = 50, epochs = 20)
```

```
Epoch 1/20
440/440 [==============================] - 387s 866ms/step - loss: 0.7705 - accuracy: 0.7674 - val_loss: 5.1173
- val_accuracy: 0.1431
Epoch 2/20
440/440 [==============================] - 386s 877ms/step - loss: 0.0510 - accuracy: 0.9837 - val_loss: 0.0147
- val_accuracy: 0.9969
Epoch 3/20
440/440 [==============================] - 371s 843ms/step - loss: 0.0188 - accuracy: 0.9942 - val_loss: 0.0333
- val_accuracy: 0.9896
Epoch 4/20
440/440 [==============================] - 366s 831ms/step - loss: 0.0270 - accuracy: 0.9919 - val_loss: 0.5513
- val_accuracy: 0.8667
Epoch 5/20
440/440 [==============================] - 363s 826ms/step - loss: 0.0333 - accuracy: 0.9897 - val_loss: 0.3548
- val_accuracy: 0.8798
Epoch 6/20
440/440 [==============================] - 363s 825ms/step - loss: 0.0225 - accuracy: 0.9932 - val_loss: 0.0201
- val_accuracy: 0.9945
Epoch 7/20
440/440 [==============================] - 365s 829ms/step - loss: 0.0132 - accuracy: 0.9955 - val_loss: 0.0668
- val_accuracy: 0.9809
Epoch 8/20
440/440 [==============================] - 366s 833ms/step - loss: 0.0134 - accuracy: 0.9954 - val_loss: 0.0323
- val_accuracy: 0.9931
Epoch 9/20
440/440 [==============================] - 369s 838ms/step - loss: 0.0250 - accuracy: 0.9922 - val_loss: 0.1939
- val_accuracy: 0.9623
Epoch 10/20
440/440 [==============================] - 374s 851ms/step - loss: 0.0156 - accuracy: 0.9955 - val_loss: 1.9309
e-04 - val_accuracy: 1.0000
Epoch 11/20
440/440 [==============================] - 368s 836ms/step - loss: 0.0159 - accuracy: 0.9958 - val_loss: 0.0017
- val_accuracy: 0.9996
Epoch 12/20
440/440 [==============================] - 363s 826ms/step - loss: 0.0139 - accuracy: 0.9961 - val_loss: 0.1175
- val_accuracy: 0.9785
Epoch 13/20
440/440 [==============================] - 358s 814ms/step - loss: 0.0051 - accuracy: 0.9986 - val_loss: 0.0010
- val_accuracy: 0.9998
Epoch 14/20
440/440 [==============================] - 365s 829ms/step - loss: 0.0156 - accuracy: 0.9960 - val_loss: 0.1342
- val_accuracy: 0.9734
Epoch 15/20
440/440 [==============================] - 369s 839ms/step - loss: 0.0205 - accuracy: 0.9945 - val_loss: 3.8335
e-04 - val_accuracy: 0.9998
Epoch 16/20
440/440 [==============================] - 379s 862ms/step - loss: 0.0118 - accuracy: 0.9967 - val_loss: 0.0204
- val_accuracy: 0.9938
Epoch 17/20
440/440 [==============================] - 369s 839ms/step - loss: 0.0060 - accuracy: 0.9982 - val_loss: 0.0011
- val_accuracy: 0.9998
Epoch 18/20
440/440 [==============================] - 385s 875ms/step - loss: 0.0064 - accuracy: 0.9980 - val_loss: 0.0381
- val_accuracy: 0.9936
Epoch 19/20
440/440 [==============================] - 392s 891ms/step - loss: 0.0061 - accuracy: 0.9983 - val_loss: 0.0620
- val_accuracy: 0.9783
Epoch 20/20
440/440 [==============================] - 372s 845ms/step - loss: 0.0101 - accuracy: 0.9976 - val_loss: 0.1728
- val_accuracy: 0.9723
```

In [48]:

```
score = model3.evaluate(X_train, Y_train, verbose=0)
print("Training Accuracy: ", f'{score[1]*100:.2f}%')

#score = model1.evaluate(X_val, Y_val, verbose = 0)
#print("Validation Accuracy:", f'{score[1]*100:.2f}%')

score = model3.evaluate(X_test, Y_test, verbose=0)
print("Testing Accuracy:" , f'{score[1]*100:.2f}%')
```

```
Training Accuracy:  97.59%
Testing Accuracy: 87.05%
```

In [49]:

```
train_pred = model3.predict(X_test)
train_pred = np.argmax(train_pred,axis = 1)
```

In [50]:

```
print(classification_report(Y_test, train_pred))
```

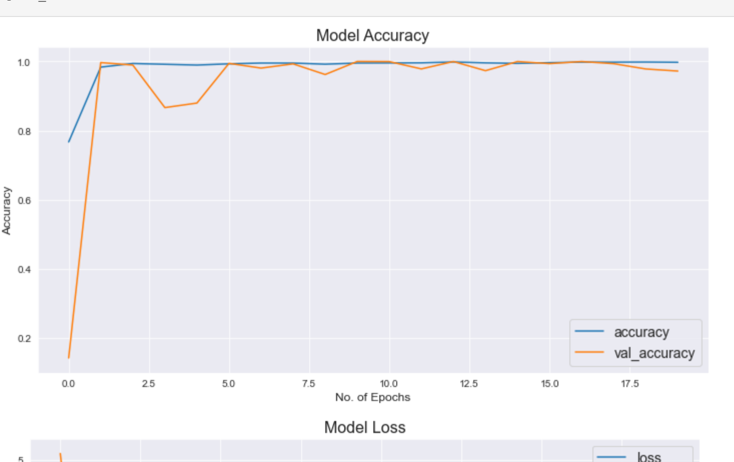
```
              precision    recall  f1-score   support

           0       0.94      1.00      0.97       331
           1       1.00      0.79      0.89       432
           2       0.98      1.00      0.99       310
           3       1.00      0.96      0.98       245
           4       1.00      0.86      0.92       498
           5       0.96      1.00      0.98       247
           6       0.88      0.84      0.86       348
           7       0.94      0.87      0.90       436
           8       0.97      0.77      0.86       288
          10       0.74      0.93      0.83       331
          11       0.99      1.00      1.00       209
          12       0.91      0.88      0.90       394
          13       0.86      0.86      0.86       291
          14       1.00      0.90      0.95       246
          15       0.69      1.00      0.81       347
          16       0.98      0.99      0.98       164
          17       0.59      0.38      0.46       144
          18       0.58      1.00      0.73       246
          19       0.77      0.56      0.65       248
          20       0.85      0.63      0.72       266
          21       1.00      0.70      0.82       346
          22       0.99      0.96      0.98       206
          23       0.82      0.93      0.87       267
          24       0.71      0.95      0.81       332

    accuracy                           0.87      7172
   macro avg       0.88      0.86      0.86      7172
weighted avg       0.89      0.87      0.87      7172
```

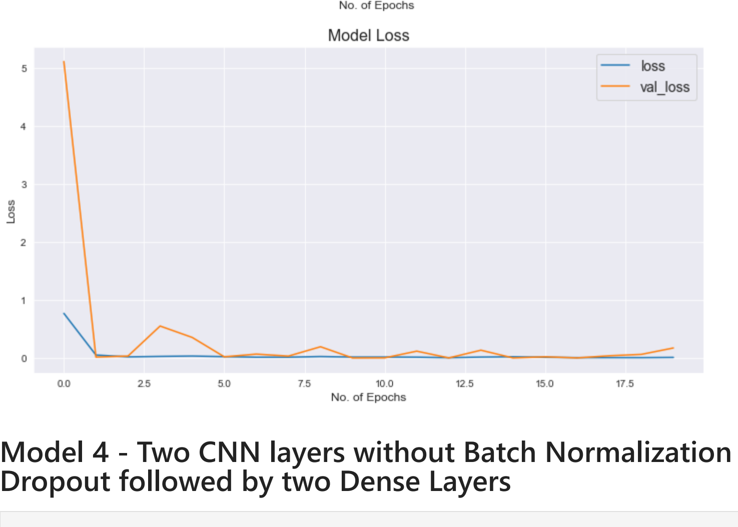
In [51]:

```
def plot_accuracy():
    plt.plot(history3.history['accuracy'])
    plt.plot(history3.history['val_accuracy'])
    plt.title('Model Accuracy', fontsize = 16)
    plt.ylabel('Accuracy', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['accuracy', 'val_accuracy'], fontsize = 14)
    plt.show()

def plot_loss():
    plt.plot(history3.history['loss'])
    plt.plot(history3.history['val_loss'])
    plt.title('Model Loss', fontsize = 16)
    plt.ylabel('Loss', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['loss', 'val_loss'], fontsize = 14)
    plt.show()
```

In [52]:

```
plt.figure(figsize=(12,6))
plot_accuracy()
plt.figure(figsize=(12,6))
plot_loss()
```





Model 4 - Two CNN layers without Batch Normalization and Dropout followed by two Dense Layers

In [53]:

```
model4 = Sequential()

model4.add(Conv2D(64, kernel_size = 5, strides = 1, padding = "same", activation = "relu", input_shape = (28, 2
model4.add(MaxPooling2D(padding = "same"))

model4.add(Conv2D(128, kernel_size = 5, strides = 1, padding = "same", activation = "relu"))
model4.add(MaxPooling2D(padding = "same"))

model4.add(Flatten())

model4.add(Dense(64, activation = "relu"))

model4.add(Dense(26, activation = "softmax"))
```

In [54]:

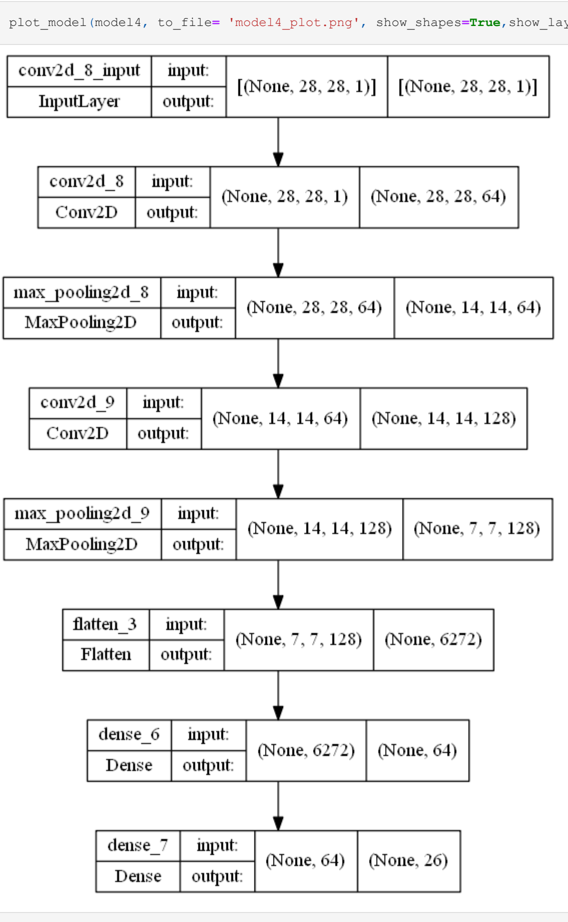
```
model4.compile(loss="sparse_categorical_crossentropy", metrics = "accuracy", optimizer = "adam")
model4.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 28, 28, 64)        1664

 max_pooling2d_8 (MaxPooling  (None, 14, 14, 64)       0
 2D)

 conv2d_9 (Conv2D)           (None, 14, 14, 128)       204928

 max_pooling2d_9 (MaxPooling  (None, 7, 7, 128)        0
 2D)

 flatten_3 (Flatten)         (None, 6272)              0

 dense_6 (Dense)             (None, 64)                401472

 dense_7 (Dense)             (None, 26)                1690

=================================================================
Total params: 609,754
Trainable params: 609,754
Non-trainable params: 0
_________________________________________________________________
```

In [55]:

```
plot_model(model4, to_file= 'model4_plot.png', show_shapes=True,show_layer_names= True)
```

Out[55]:



In [56]:

```
history4 = model4.fit(X_train, Y_train, validation_data = (X_val, Y_val), batch_size = 50, epochs = 20)
```

```
Epoch 1/20
440/440 [==============================] - 335s 759ms/step - loss: 1.4374 - accuracy: 0.5570 - val_loss: 0.3773
- val_accuracy: 0.8944
Epoch 2/20
440/440 [==============================] - 338s 768ms/step - loss: 0.1571 - accuracy: 0.9617 - val_loss: 0.0623
- val_accuracy: 0.9834
Epoch 3/20
440/440 [==============================] - 337s 766ms/step - loss: 0.0259 - accuracy: 0.9959 - val_loss: 0.0070
- val_accuracy: 1.0000
Epoch 4/20
440/440 [==============================] - 337s 766ms/step - loss: 0.0043 - accuracy: 1.0000 - val_loss: 0.0029
- val_accuracy: 1.0000
Epoch 5/20
440/440 [==============================] - 336s 763ms/step - loss: 0.0019 - accuracy: 1.0000 - val_loss: 0.0017
- val_accuracy: 1.0000
Epoch 6/20
440/440 [==============================] - 336s 764ms/step - loss: 0.0011 - accuracy: 1.0000 - val_loss: 0.0010
- val_accuracy: 1.0000
Epoch 7/20
440/440 [==============================] - 341s 775ms/step - loss: 6.6073e-04 - accuracy: 1.0000 - val_loss: 6.
2519e-04 - val_accuracy: 1.0000
Epoch 8/20
440/440 [==============================] - 340s 774ms/step - loss: 4.2226e-04 - accuracy: 1.0000 - val_loss: 4.
5880e-04 - val_accuracy: 1.0000
Epoch 9/20
440/440 [==============================] - 352s 800ms/step - loss: 2.8766e-04 - accuracy: 1.0000 - val_loss: 3.
2593e-04 - val_accuracy: 1.0000
Epoch 10/20
440/440 [==============================] - 359s 815ms/step - loss: 2.1338e-04 - accuracy: 1.0000 - val_loss: 2.
3786e-04 - val_accuracy: 1.0000
Epoch 11/20
440/440 [==============================] - 348s 790ms/step - loss: 1.5118e-04 - accuracy: 1.0000 - val_loss: 1.
8559e-04 - val_accuracy: 1.0000
Epoch 12/20
440/440 [==============================] - 392s 890ms/step - loss: 1.1063e-04 - accuracy: 1.0000 - val_loss: 1.
3273e-04 - val_accuracy: 1.0000
Epoch 13/20
440/440 [==============================] - 379s 862ms/step - loss: 8.1560e-05 - accuracy: 1.0000 - val_loss: 1.
0017e-04 - val_accuracy: 1.0000
Epoch 14/20
440/440 [==============================] - 390s 887ms/step - loss: 6.2256e-05 - accuracy: 1.0000 - val_loss: 7.
4005e-05 - val_accuracy: 1.0000
Epoch 15/20
440/440 [==============================] - 386s 877ms/step - loss: 4.7440e-05 - accuracy: 1.0000 - val_loss: 6.
3518e-05 - val_accuracy: 1.0000
Epoch 16/20
440/440 [==============================] - 390s 886ms/step - loss: 3.5383e-05 - accuracy: 1.0000 - val_loss: 4.
4919e-05 - val_accuracy: 1.0000
Epoch 17/20
440/440 [==============================] - 375s 851ms/step - loss: 2.6630e-05 - accuracy: 1.0000 - val_loss: 3.
4960e-05 - val_accuracy: 1.0000
Epoch 18/20
440/440 [==============================] - 388s 882ms/step - loss: 2.0751e-05 - accuracy: 1.0000 - val_loss: 2.
5635e-05 - val_accuracy: 1.0000
Epoch 19/20
440/440 [==============================] - 379s 862ms/step - loss: 1.6578e-05 - accuracy: 1.0000 - val_loss: 2.
1541e-05 - val_accuracy: 1.0000
Epoch 20/20
440/440 [==============================] - 411s 933ms/step - loss: 1.2512e-05 - accuracy: 1.0000 - val_loss: 1.
6388e-05 - val_accuracy: 1.0000
```

In [57]:

```
score = model4.evaluate(X_train, Y_train, verbose=0)
print("Training Accuracy: ", f'{score[1]*100:.2f}%')

#score = model1.evaluate(X_val, Y_val, verbose = 0)
#print("Validation Accuracy:", f'{score[1]*100:.2f}%')

score = model4.evaluate(X_test, Y_test, verbose=0)
print("Testing Accuracy:" , f'{score[1]*100:.2f}%')
```

```
Training Accuracy:  100.00%
Testing Accuracy: 89.89%
```

In [58]:

```
train_pred = model4.predict(X_test)
train_pred = np.argmax(train_pred,axis = 1)
```

In [59]:

```
print(classification_report(Y_test, train_pred))
```

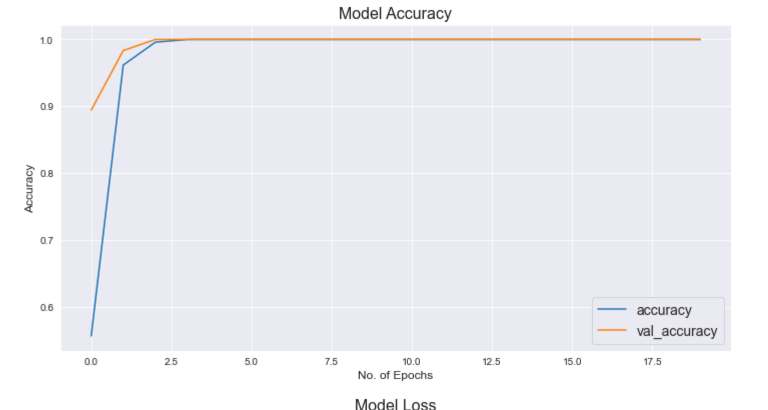
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       331
           1       1.00      0.95      0.98       432
           2       1.00      1.00      1.00       310
           3       0.74      0.90      0.81       245
           4       0.96      0.96      0.96       498
           5       0.91      1.00      0.95       247
           6       0.95      0.89      0.92       348
           7       0.96      0.96      0.96       436
           8       0.99      0.85      0.92       288
          10       0.82      0.88      0.85       331
          11       0.91      1.00      0.95       209
          12       0.99      0.84      0.91       394
          13       0.91      0.79      0.84       291
          14       1.00      0.87      0.93       246
          15       0.92      1.00      0.96       347
          16       0.69      1.00      0.82       164
          17       0.77      0.97      0.86       144
          18       0.68      0.82      0.74       246
          19       0.89      0.67      0.76       248
          20       0.93      0.79      0.85       266
          21       0.87      0.82      0.84       346
          22       0.86      0.98      0.91       206
          23       0.74      0.88      0.80       267
          24       0.92      0.76      0.83       332

    accuracy                           0.90      7172
   macro avg       0.89      0.90      0.89      7172
weighted avg       0.91      0.90      0.90      7172
```

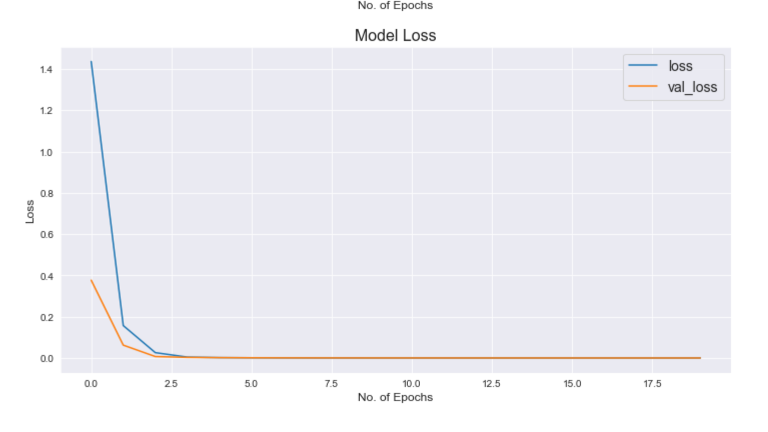
In [62]:

```
def plot_accuracy():
    plt.plot(history4.history['accuracy'])
    plt.plot(history4.history['val_accuracy'])
    plt.title('Model Accuracy', fontsize = 16)
    plt.ylabel('Accuracy', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['accuracy', 'val_accuracy'], fontsize = 14)
    plt.show()

def plot_loss():
    plt.plot(history4.history['loss'])
    plt.plot(history4.history['val_loss'])
    plt.title('Model Loss', fontsize = 16)
    plt.ylabel('Loss', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['loss', 'val_loss'], fontsize = 14)
    plt.show()
```

In [63]:

```
plt.figure(figsize=(12,6))
plot_accuracy()
plt.figure(figsize=(12,6))
plot_loss()
```





Changing the Optimizer and its parameter with the Best Baseline Model

In [69]:
```
model5 = Sequential()

model5.add(Conv2D(32, kernel_size = 5, strides = 1, padding = "same", activation = "relu", input_shape = (28, 2
model5.add(MaxPooling2D(padding = "same"))
model5.add(BatchNormalization())

model5.add(Conv2D(64, kernel_size = 5, strides = 1, padding = "same", activation = "relu"))
model5.add(MaxPooling2D(padding = "same"))
model5.add(BatchNormalization())
model5.add(Dropout(0.2))

model5.add(Conv2D(128, kernel_size = 5, strides = 1, padding = "same", activation = "relu"))
model5.add(MaxPooling2D(padding = "same"))
model5.add(BatchNormalization())
model5.add(Dropout(0.2))

model5.add(Flatten())

model5.add(Dense(64, activation = "relu"))
model5.add(Dropout(0.2))

model5.add(Dense(26, activation = "softmax"))
```

In [70]:
```
opt1 = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
```

In [72]:
```
model5.compile(loss="sparse_categorical_crossentropy", metrics = "accuracy", optimizer = opt1)
model5.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_13 (Conv2D)          (None, 28, 28, 32)        832

 max_pooling2d_13 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 batch_normalization_8 (Batc  (None, 14, 14, 32)       128
 hNormalization)

 conv2d_14 (Conv2D)          (None, 14, 14, 64)        51264

 max_pooling2d_14 (MaxPoolin  (None, 7, 7, 64)         0
 g2D)

 batch_normalization_9 (Batc  (None, 7, 7, 64)         256
 hNormalization)

 dropout_8 (Dropout)         (None, 7, 7, 64)          0

 conv2d_15 (Conv2D)          (None, 7, 7, 128)         204928

 max_pooling2d_15 (MaxPoolin  (None, 4, 4, 128)        0
 g2D)

 batch_normalization_10 (Bat  (None, 4, 4, 128)        512
 chNormalization)

 dropout_9 (Dropout)         (None, 4, 4, 128)         0

 flatten_5 (Flatten)         (None, 2048)              0

 dense_10 (Dense)            (None, 64)                131136

 dropout_10 (Dropout)        (None, 64)                0

 dense_11 (Dense)            (None, 26)                1690

=================================================================
Total params: 390,746
Trainable params: 390,298
Non-trainable params: 448
_________________________________________________________________
```

In [73]:
```
history5 = model5.fit(X_train, Y_train, validation_data = (X_val, Y_val), batch_size = 50, epochs = 20)
```

```
Epoch 1/20
440/440 [==============================] - 199s 451ms/step - loss: 1.1414 - accuracy: 0.6621 - val_loss: 0.8267
- val_accuracy: 0.8385
Epoch 2/20
440/440 [==============================] - 198s 451ms/step - loss: 0.1652 - accuracy: 0.9566 - val_loss: 0.0207
- val_accuracy: 0.9998
Epoch 3/20
440/440 [==============================] - 195s 443ms/step - loss: 0.0714 - accuracy: 0.9849 - val_loss: 0.0129
- val_accuracy: 1.0000
Epoch 4/20
440/440 [==============================] - 192s 437ms/step - loss: 0.0419 - accuracy: 0.9930 - val_loss: 0.0040
- val_accuracy: 1.0000
Epoch 5/20
440/440 [==============================] - 194s 440ms/step - loss: 0.0294 - accuracy: 0.9954 - val_loss: 0.0020
- val_accuracy: 1.0000
Epoch 6/20
440/440 [==============================] - 194s 440ms/step - loss: 0.0221 - accuracy: 0.9964 - val_loss: 0.0010
- val_accuracy: 1.0000
Epoch 7/20
440/440 [==============================] - 214s 487ms/step - loss: 0.0190 - accuracy: 0.9968 - val_loss: 4.9454
e-04 - val_accuracy: 1.0000
Epoch 8/20
440/440 [==============================] - 295s 671ms/step - loss: 0.0152 - accuracy: 0.9978 - val_loss: 9.5553
e-04 - val_accuracy: 1.0000
Epoch 9/20
440/440 [==============================] - 274s 623ms/step - loss: 0.0147 - accuracy: 0.9972 - val_loss: 6.9712
e-04 - val_accuracy: 1.0000
Epoch 10/20
440/440 [==============================] - 272s 619ms/step - loss: 0.0123 - accuracy: 0.9981 - val_loss: 8.5228
e-04 - val_accuracy: 1.0000
Epoch 11/20
440/440 [==============================] - 238s 541ms/step - loss: 0.0098 - accuracy: 0.9987 - val_loss: 7.6044
e-04 - val_accuracy: 1.0000
Epoch 12/20
440/440 [==============================] - 223s 507ms/step - loss: 0.0085 - accuracy: 0.9988 - val_loss: 2.8860
e-04 - val_accuracy: 1.0000
Epoch 13/20
440/440 [==============================] - 221s 501ms/step - loss: 0.0077 - accuracy: 0.9992 - val_loss: 2.7764
e-04 - val_accuracy: 1.0000
Epoch 14/20
440/440 [==============================] - 230s 523ms/step - loss: 0.0076 - accuracy: 0.9990 - val_loss: 5.0932
e-04 - val_accuracy: 1.0000
Epoch 15/20
440/440 [==============================] - 224s 510ms/step - loss: 0.0077 - accuracy: 0.9990 - val_loss: 2.9038
e-04 - val_accuracy: 1.0000
Epoch 16/20
440/440 [==============================] - 224s 509ms/step - loss: 0.0066 - accuracy: 0.9992 - val_loss: 2.8827
e-04 - val_accuracy: 1.0000
Epoch 17/20
440/440 [==============================] - 253s 575ms/step - loss: 0.0060 - accuracy: 0.9994 - val_loss: 1.3522
e-04 - val_accuracy: 1.0000
Epoch 18/20
440/440 [==============================] - 210s 476ms/step - loss: 0.0060 - accuracy: 0.9990 - val_loss: 5.4719
e-04 - val_accuracy: 1.0000
Epoch 19/20
440/440 [==============================] - 222s 504ms/step - loss: 0.0052 - accuracy: 0.9993 - val_loss: 6.7861
e-05 - val_accuracy: 1.0000
Epoch 20/20
440/440 [==============================] - 219s 499ms/step - loss: 0.0046 - accuracy: 0.9994 - val_loss: 1.9454
e-04 - val_accuracy: 1.0000
```

In [74]:
```
score = model5.evaluate(X_train, Y_train, verbose=0)
print("Training Accuracy: ", f'{score[1]*100:.2f}%')

score = model5.evaluate(X_val, Y_val, verbose = 0)
print("Validation Accuracy:", f'{score[1]*100:.2f}%')

score = model5.evaluate(X_test, Y_test, verbose=0)
print("Testing Accuracy:" , f'{score[1]*100:.2f}%')
```

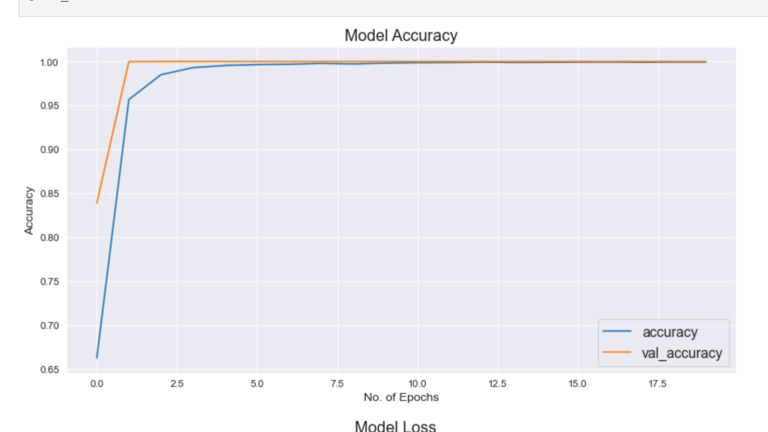
```
Training Accuracy:  100.00%
Validation Accuracy: 100.00%
Testing Accuracy: 95.59%
```

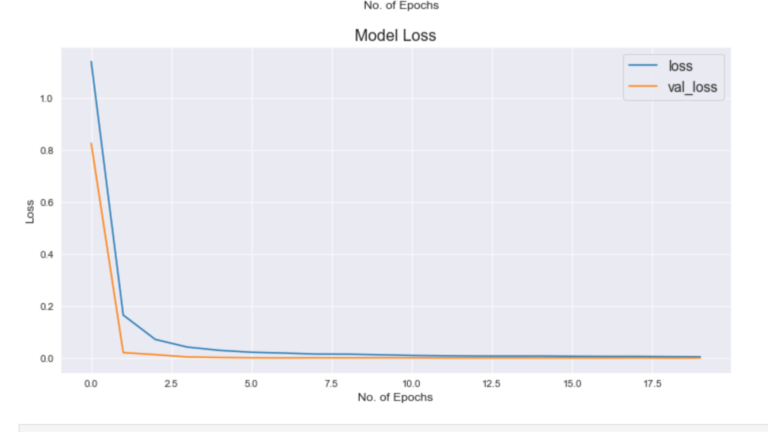
In [76]:
```
train_pred = model5.predict(X_test)
train_pred = np.argmax(train_pred,axis = 1)
```

In [77]:
```
print(classification_report(Y_test, train_pred))
```

```
              precision    recall  f1-score   support

           0       0.97      1.00      0.98       331
           1       1.00      1.00      1.00       432
           2       1.00      1.00      1.00       310
           3       1.00      1.00      1.00       245
           4       0.92      1.00      0.96       498
           5       1.00      1.00      1.00       247
           6       0.94      0.95      0.94       348
           7       1.00      0.95      0.98       436
           8       0.95      1.00      0.97       288
          10       1.00      0.94      0.97       331
          11       0.92      1.00      0.96       209
          12       0.90      0.95      0.92       394
          13       1.00      0.82      0.90       291
          14       1.00      1.00      1.00       246
          15       1.00      1.00      1.00       347
          16       1.00      1.00      1.00       164
          17       0.82      0.85      0.84       144
          18       0.91      0.91      0.91       246
          19       0.89      0.68      0.77       248
          20       1.00      0.89      0.94       266
          21       0.90      1.00      0.95       346
          22       1.00      1.00      1.00       206
          23       0.81      1.00      0.89       267
          24       1.00      0.90      0.95       332

    accuracy                           0.96      7172
   macro avg       0.96      0.95      0.95      7172
weighted avg       0.96      0.96      0.96      7172
```

In [78]:
```
def plot_accuracy():
    plt.plot(history5.history['accuracy'])
    plt.plot(history5.history['val_accuracy'])
    plt.title('Model Accuracy', fontsize = 16)
    plt.ylabel('Accuracy', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['accuracy', 'val_accuracy'], fontsize = 14)
    plt.show()

def plot_loss():
    plt.plot(history5.history['loss'])
    plt.plot(history5.history['val_loss'])
    plt.title('Model Loss', fontsize = 16)
    plt.ylabel('Loss', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['loss', 'val_loss'], fontsize = 14)
    plt.show()

plt.figure(figsize=(12,6))
plot_accuracy()
plt.figure(figsize=(12,6))
plot_loss()
```





In [83]:
```
opt2 = tf.keras.optimizers.RMSprop(learning_rate=0.01, rho=0.95, momentum=0.9)
```

In [84]:
```
model5.compile(loss="sparse_categorical_crossentropy", metrics = "accuracy", optimizer = opt2)
model5.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_13 (Conv2D)          (None, 28, 28, 32)        832

 max_pooling2d_13 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 batch_normalization_8 (Batc  (None, 14, 14, 32)       128
 hNormalization)

 conv2d_14 (Conv2D)          (None, 14, 14, 64)        51264

 max_pooling2d_14 (MaxPoolin  (None, 7, 7, 64)         0
 g2D)

 batch_normalization_9 (Batc  (None, 7, 7, 64)         256
 hNormalization)

 dropout_8 (Dropout)         (None, 7, 7, 64)          0

 conv2d_15 (Conv2D)          (None, 7, 7, 128)         204928

 max_pooling2d_15 (MaxPoolin  (None, 4, 4, 128)        0
 g2D)

 batch_normalization_10 (Bat  (None, 4, 4, 128)        512
 chNormalization)

 dropout_9 (Dropout)         (None, 4, 4, 128)         0

 flatten_5 (Flatten)         (None, 2048)              0

 dense_10 (Dense)            (None, 64)                131136

 dropout_10 (Dropout)        (None, 64)                0

 dense_11 (Dense)            (None, 26)                1690

=================================================================
Total params: 390,746
Trainable params: 390,298
Non-trainable params: 448
_________________________________________________________________
```

In [85]:
```
history5 = model5.fit(X_train, Y_train, validation_data = (X_val, Y_val), batch_size = 50, epochs = 20)
```

```
Epoch 1/20
440/440 [==============================] - 201s 454ms/step - loss: 3.1943 - accuracy: 0.0419 - val_loss: 3.1851
- val_accuracy: 0.0419
Epoch 2/20
440/440 [==============================] - 244s 554ms/step - loss: 3.1921 - accuracy: 0.0423 - val_loss: 3.2015
- val_accuracy: 0.0463
Epoch 3/20
440/440 [==============================] - 197s 448ms/step - loss: 3.1937 - accuracy: 0.0427 - val_loss: 3.1865
- val_accuracy: 0.0530
Epoch 4/20
440/440 [==============================] - 191s 435ms/step - loss: 3.1944 - accuracy: 0.0424 - val_loss: 3.1836
- val_accuracy: 0.0401
Epoch 5/20
440/440 [==============================] - 203s 462ms/step - loss: 3.1925 - accuracy: 0.0431 - val_loss: 3.1820
- val_accuracy: 0.0488
Epoch 6/20
440/440 [==============================] - 290s 659ms/step - loss: 3.1923 - accuracy: 0.0430 - val_loss: 3.1894
- val_accuracy: 0.0475
Epoch 7/20
440/440 [==============================] - 305s 692ms/step - loss: 3.1925 - accuracy: 0.0434 - val_loss: 3.1949
- val_accuracy: 0.0413
Epoch 8/20
440/440 [==============================] - 306s 696ms/step - loss: 3.1926 - accuracy: 0.0425 - val_loss: 3.1868
- val_accuracy: 0.0475
Epoch 9/20
440/440 [==============================] - 334s 760ms/step - loss: 3.1912 - accuracy: 0.0444 - val_loss: 3.1936
- val_accuracy: 0.0401
Epoch 10/20
440/440 [==============================] - 309s 703ms/step - loss: 3.1940 - accuracy: 0.0433 - val_loss: 3.1924
- val_accuracy: 0.0463
Epoch 11/20
440/440 [==============================] - 291s 661ms/step - loss: 3.1926 - accuracy: 0.0437 - val_loss: 3.2141
- val_accuracy: 0.0463
Epoch 12/20
440/440 [==============================] - 246s 560ms/step - loss: 3.1933 - accuracy: 0.0413 - val_loss: 3.1957
- val_accuracy: 0.0475
Epoch 13/20
440/440 [==============================] - 237s 538ms/step - loss: 3.1911 - accuracy: 0.0438 - val_loss: 3.1912
- val_accuracy: 0.0399
Epoch 14/20
440/440 [==============================] - 233s 530ms/step - loss: 3.1922 - accuracy: 0.0446 - val_loss: 3.1844
- val_accuracy: 0.0419
Epoch 15/20
440/440 [==============================] - 225s 512ms/step - loss: 3.1922 - accuracy: 0.0441 - val_loss: 3.1878
- val_accuracy: 0.0475
Epoch 16/20
440/440 [==============================] - 217s 494ms/step - loss: 3.1926 - accuracy: 0.0428 - val_loss: 3.1877
- val_accuracy: 0.0441
Epoch 17/20
440/440 [==============================] - 239s 544ms/step - loss: 3.1927 - accuracy: 0.0439 - val_loss: 3.1922
- val_accuracy: 0.0359
Epoch 18/20
440/440 [==============================] - 217s 492ms/step - loss: 3.1919 - accuracy: 0.0423 - val_loss: 3.1890
- val_accuracy: 0.0530
Epoch 19/20
440/440 [==============================] - 212s 483ms/step - loss: 3.1919 - accuracy: 0.0428 - val_loss: 3.1923
- val_accuracy: 0.0399
Epoch 20/20
440/440 [==============================] - 201s 458ms/step - loss: 3.1904 - accuracy: 0.0449 - val_loss: 3.2058
- val_accuracy: 0.0399
```

In [86]:
```
score = model5.evaluate(X_train, Y_train, verbose=0)
print("Training Accuracy: ", f'{score[1]*100:.2f}%')

score = model5.evaluate(X_val, Y_val, verbose = 0)
print("Validation Accuracy:", f'{score[1]*100:.2f}%')

score = model5.evaluate(X_test, Y_test, verbose=0)
print("Testing Accuracy:" , f'{score[1]*100:.2f}%')
```

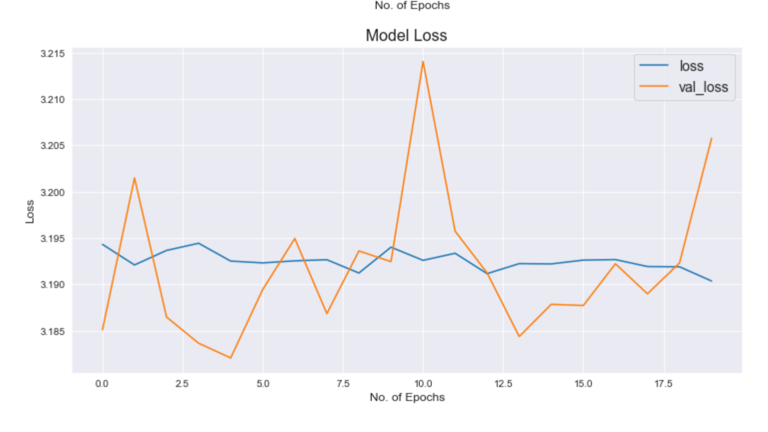
```
Training Accuracy:  4.29%
Validation Accuracy: 3.99%
Testing Accuracy: 4.02%
```

In [87]:
```
train_pred = model5.predict(X_test)
train_pred = np.argmax(train_pred,axis = 1)
print(classification_report(Y_test, train_pred))
```

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00       331
           1       0.00      0.00      0.00       432
           2       0.00      0.00      0.00       310
           3       0.00      0.00      0.00       245
           4       0.00      0.00      0.00       498
           5       0.00      0.00      0.00       247
           6       0.00      0.00      0.00       348
           7       0.00      0.00      0.00       436
           8       0.04      1.00      0.08       288
          10       0.00      0.00      0.00       331
          11       0.00      0.00      0.00       209
          12       0.00      0.00      0.00       394
          13       0.00      0.00      0.00       291
          14       0.00      0.00      0.00       246
          15       0.00      0.00      0.00       347
          16       0.00      0.00      0.00       164
          17       0.00      0.00      0.00       144
          18       0.00      0.00      0.00       246
          19       0.00      0.00      0.00       248
          20       0.00      0.00      0.00       266
          21       0.00      0.00      0.00       346
          22       0.00      0.00      0.00       206
          23       0.00      0.00      0.00       267
          24       0.00      0.00      0.00       332

    accuracy                           0.04      7172
   macro avg       0.00      0.04      0.00      7172
weighted avg       0.00      0.04      0.00      7172
```

```
C:\Users\User\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1248: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division`
parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\User\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1248: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division`
parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
C:\Users\User\anaconda3\lib\site-packages\sklearn\metrics\_classification.py:1248: UndefinedMetricWarning: Prec
ision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division`
parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

In [88]:
```
def plot_accuracy():
    plt.plot(history5.history['accuracy'])
    plt.plot(history5.history['val_accuracy'])
    plt.title('Model Accuracy', fontsize = 16)
    plt.ylabel('Accuracy', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['accuracy', 'val_accuracy'], fontsize = 14)
    plt.show()

def plot_loss():
    plt.plot(history5.history['loss'])
    plt.plot(history5.history['val_loss'])
    plt.title('Model Loss', fontsize = 16)
    plt.ylabel('Loss', fontsize = 12)
    plt.xlabel('No. of Epochs', fontsize = 12)
    plt.legend(['loss', 'val_loss'], fontsize = 14)
    plt.show()

plt.figure(figsize=(12,6))
plot_accuracy()
plt.figure(figsize=(12,6))
plot_loss()
```



Training the Model with Data Augmentation

In [89]:
```
from keras.preprocessing.image import ImageDataGenerator
from keras.regularizers import l2
```

In [98]:
```
data_gen=ImageDataGenerator(rotation_range = 0.2, zoom_range=0.2, width_shift_range=0.1, height_shift_range=0.1
                            horizontal_flip=True)
data_gen.fit(X_train)
```

In [105…
```
history6 = model6.fit(data_gen.flow(X_train,Y_train,batch_size=50),epochs=20,validation_data = (X_val, Y_val))
```

```
Epoch 1/20
440/440 [==============================] - 213s 482ms/step - loss: 0.2411 - accuracy: 0.9798 - val_loss: 0.2142
- val_accuracy: 0.9924
Epoch 2/20
440/440 [==============================] - 218s 496ms/step - loss: 0.2477 - accuracy: 0.9798 - val_loss: 0.2183
- val_accuracy: 0.9903
Epoch 3/20
440/440 [==============================] - 215s 488ms/step - loss: 0.2528 - accuracy: 0.9796 - val_loss: 0.2355
- val_accuracy: 0.9902
Epoch 4/20
440/440 [==============================] - 211s 480ms/step - loss: 0.2444 - accuracy: 0.9830 - val_loss: 0.1705
- val_accuracy: 0.9987
Epoch 5/20
440/440 [==============================] - 221s 502ms/step - loss: 0.2261 - accuracy: 0.9835 - val_loss: 0.1669
- val_accuracy: 1.0000
Epoch 6/20
440/440 [==============================] - 244s 555ms/step - loss: 0.2326 - accuracy: 0.9825 - val_loss: 0.1957
- val_accuracy: 0.9945
Epoch 7/20
440/440 [==============================] - 259s 589ms/step - loss: 0.2412 - accuracy: 0.9822 - val_loss: 0.2781
- val_accuracy: 0.9654
Epoch 8/20
440/440 [==============================] - 257s 584ms/step - loss: 0.2272 - accuracy: 0.9829 - val_loss: 0.1910
- val_accuracy: 0.9903
Epoch 9/20
440/440 [==============================] - 303s 688ms/step - loss: 0.2211 - accuracy: 0.9833 - val_loss: 0.1740
- val_accuracy: 0.9998
Epoch 10/20
440/440 [==============================] - 247s 560ms/step - loss: 0.2194 - accuracy: 0.9853 - val_loss: 0.2053
- val_accuracy: 0.9924
Epoch 11/20
440/440 [==============================] - 277s 628ms/step - loss: 0.2129 - accuracy: 0.9859 - val_loss: 0.1742
- val_accuracy: 0.9956
Epoch 12/20
440/440 [==============================] - 262s 595ms/step - loss: 0.2509 - accuracy: 0.9812 - val_loss: 0.1892
- val_accuracy: 0.9989
Epoch 13/20
440/440 [==============================] - 281s 639ms/step - loss: 0.2415 - accuracy: 0.9816 - val_loss: 0.2416
- val_accuracy: 0.9831
Epoch 14/20
440/440 [==============================] - 224s 509ms/step - loss: 0.2304 - accuracy: 0.9843 - val_loss: 0.1542
- val_accuracy: 0.9984
Epoch 15/20
440/440 [==============================] - 228s 519ms/step - loss: 0.1959 - accuracy: 0.9868 - val_loss: 0.1556
- val_accuracy: 0.9967
Epoch 16/20
440/440 [==============================] - 221s 503ms/step - loss: 0.2020 - accuracy: 0.9856 - val_loss: 0.1868
- val_accuracy: 0.9962
Epoch 17/20
440/440 [==============================] - 227s 515ms/step - loss: 0.2405 - accuracy: 0.9829 - val_loss: 0.2576
- val_accuracy: 0.9831
Epoch 18/20
440/440 [==============================] - 278s 631ms/step - loss: 0.2030 - accuracy: 0.9872 - val_loss: 0.1769
- val_accuracy: 0.9960
Epoch 19/20
440/440 [==============================] - 257s 583ms/step - loss: 0.2312 - accuracy: 0.9823 - val_loss: 0.1856
- val_accuracy: 1.0000
Epoch 20/20
440/440 [==============================] - 248s 564ms/step - loss: 0.2361 - accuracy: 0.9835 - val_loss: 0.1758
- val_accuracy: 0.9991
```

In [106…
```
score = model6.evaluate(X_train, Y_train, verbose=0)
print("Training Accuracy: ", f'{score[1]*100:.2f}%')

score = model6.evaluate(X_val, Y_val, verbose = 0)
print("Validation Accuracy:", f'{score[1]*100:.2f}%')

score = model6.evaluate(X_test, Y_test, verbose=0)
print("Testing Accuracy:" , f'{score[1]*100:.2f}%')
```

```
Training Accuracy:  99.97%
Validation Accuracy: 99.91%
Testing Accuracy: 99.74%
```

In [135…
```
score = model6.evaluate(X_train, Y_train, verbose=0)
print("Training Accuracy: ", f'{score[1]*100:.2f}%')

score = model6.evaluate(X_val, Y_val, verbose = 0)
print("Validation Accuracy:", f'{score[1]*100:.2f}%')

score = model6.evaluate(X_test, Y_test, verbose=0)
print("Testing Accuracy:" , f'{score[1]*100:.2f}%')
```

```
Training Accuracy:  99.97%
Validation Accuracy: 99.91%
Testing Accuracy: 99.74%
```

In [ ]:

In [ ]: