DEEP LEARNING PROJECT FASHION MNIST CLASSIFICATION

INDEX:-

1.ABSTARCT :-

In this project, i have built a fashion apparel recognition using the Convolutional Neural Network (CNN) model. To train the CNN model, we have used the Fashion MNIST dataset. After successful training, the CNN model can predict the name of the class given apparel item belongs to. This is a multiclass classification problem in which there are 10 apparel classes the items will be classified.

The fashion training set consists of 70,000 images divided into 60,000 training and 10,000 testing samples. Dataset sample consists of 28x28 grayscale images, associated with a label from 10 classes.

So the end goal is to train and test the model using Convolution neural network

2.OBJECTIVE:-

This work is part of my experiments with Fashion-MNIST dataset using various Machine Learning algorithms/models. The objective is to identify (predict) different fashion products from the given images using various best possible Machine Learning Models (Algorithms) and compare their results (performance measures/scores) to arrive at the best ML model. I have also experimented with 'dimensionality reduction' technique for this problem.

Artificial Intelligence is among the next big things in the software engineering field that empowers numerous applications in health care, finance, logistic, industries for good measure. Several scientists have embarked on utilising the advancement of Artificial Intelligence and high-grade dataset to change our lives profoundly. One notable field in Artificial Intelligence is Computer Vision, which rapidly emerged over the last decade thank to an enormous amount of visual data and significant development in GPU processing power. Neural networks can now have millions of trainable parameters which makes technologies like diseases self-diagnoses or self-driving car possible.

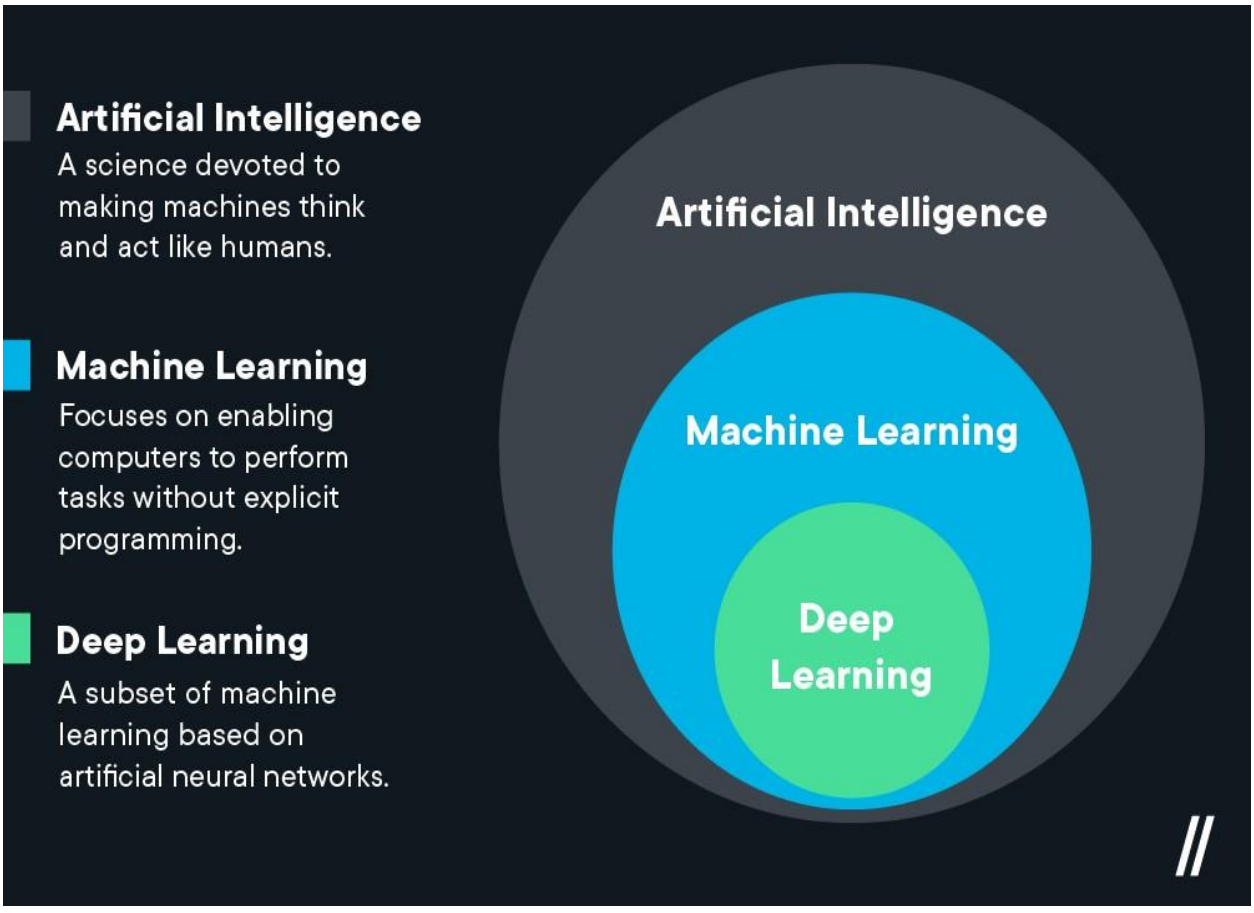
In this work, we implemented a multi-layer neural network to classify images of fashion products using the Zalando's Fashion MNIST Dataset using softmax outputs for classification in order to identify strategies that optimize performance in neural networks. The multi-layer neural network included inputs, hidden layers, and outputs. The goal was to classify 28x28 greyscale images into one of 10 classes: top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot. The Fashion MNIST dataset is a more complicated problem to solve compared to the classical MNIST dataset.

First, a neural network was implemented containing 2 hidden layers, each with 50 units, and tanh activation for the hidden layers. Cross-entropy loss was used as the objective function. We used mini-batch stochastic gradient descent for training throughout. Here we compared different learning rates and minibatch sizes to find the best model. Next, we performed various experiments on the multi-layer neural network with hopes of optimizing the performance of the network.

We experimented with regularization where we added a weight decay, L2 regularization, in the update scheme. Additionally, we tested various activation functions for the hidden units to compare to the initial tanh function that was used. Lastly, we varied the neural network topology. In each scenario, we analyzed network performance in order to optimize classification and minimize loss. Figure 1 summarizes the results from our network optimization experiments. The rest of this work goes into the findings and the resulting intuition gleaned in detail.

An overview of Deep Learning:-

Artificial Intelligence is a vast field which concerns several areas like Statistic, Optimization, Machine Learning and Deep Learning. This section introduces DL as the state-of-the-art solution for AI problems with complex datasets like visual data, sound or natural languages and how it's related other fields of AI. Besides, this section explains the concepts of Convolutional Neural Network, optimization process of Neural Network training and then dives deeper to Mathematics theory behind their implementation.



3.INTRODUCTION:-

A tremendous interest in deep learning has emerged in recent years . The most established algorithm among various deep learning models is convolutional neural network (CNN), a class of artificial neural networks that has been a dominant method in computer vision tasks since the astonishing results were shared on the object recognition competition known as the ImageNet Large Scale Visual Recognition Competition (ILSVRC) in 2012. Medical research is no exception, as CNN has achieved expert-level performances in various fields. Gulshan et al, Esteva et al. , and Ehteshami Bejnordi et al demonstrated the potential of deep learning for diabetic retinopathy screening, skin lesion classification, and lymph node metastasis detection, respectively. Needless to say, there has been a surge of interest in the potential of CNN among radiology researchers, and several studies have already been published in areas such as lesion detection , classification , segmentation , image reconstruction , and natural language processing . Familiarity with this state-of-the-art methodology would help not only researchers who apply CNN to their tasks in radiology and medical imaging, but also clinical radiologists, as deep learning may influence their practice in the near future. This article focuses on the basic concepts of CNN and their application to various radiology tasks, and discusses its challenges and future directions.

What is CNN ?

CNN is a type of deep learning model for processing data that has a grid pattern, such as images, which is inspired by the organization of animal visual cortex and designed to automatically and adaptively learn spatial hierarchies of features, from low- to high-level patterns. CNN is a mathematical construct that is typically composed of three types of layers (or building blocks): convolution, pooling, and fully connected layers. The first two, convolution and pooling layers, perform feature extraction, whereas the third, a fully connected layer, maps the extracted features into final output, such as classification. A convolution layer plays a key role in CNN, which is composed of a stack of mathematical operations, such as convolution, a specialized type of linear operation. In digital images, pixel values are stored in a two-dimensional (2D) grid, i.e., an array of numbers , and a small grid of parameters called kernel, an optimizable feature extractor, is applied at each image position, which makes CNNs highly efficient for image processing, since a feature may occur anywhere in the image. As one layer feeds its output into the next layer, extracted features can hierarchically and progressively become more complex. The process of optimizing parameters such as kernels is called training, which is performed so as to minimize the difference between outputs and ground truth labels through an optimization algorithm called backpropagation and gradient descent, among others.

An overview of a convolutional neural network (CNN) architecture and the training process. A CNN is composed of a stacking of several building blocks:-

convolution layers,

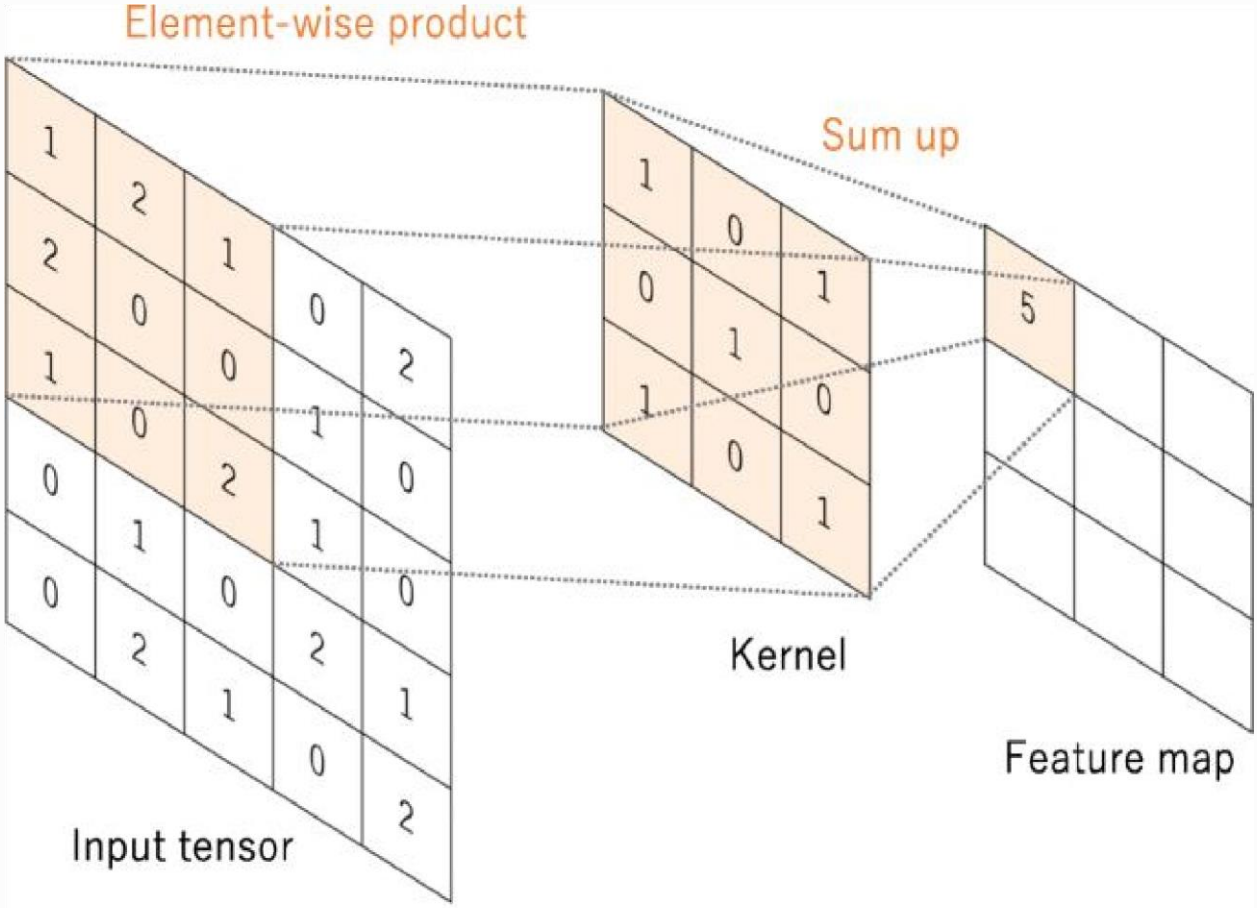
pooling layers (e.g., max pooling),

and fully connected (FC) layers.

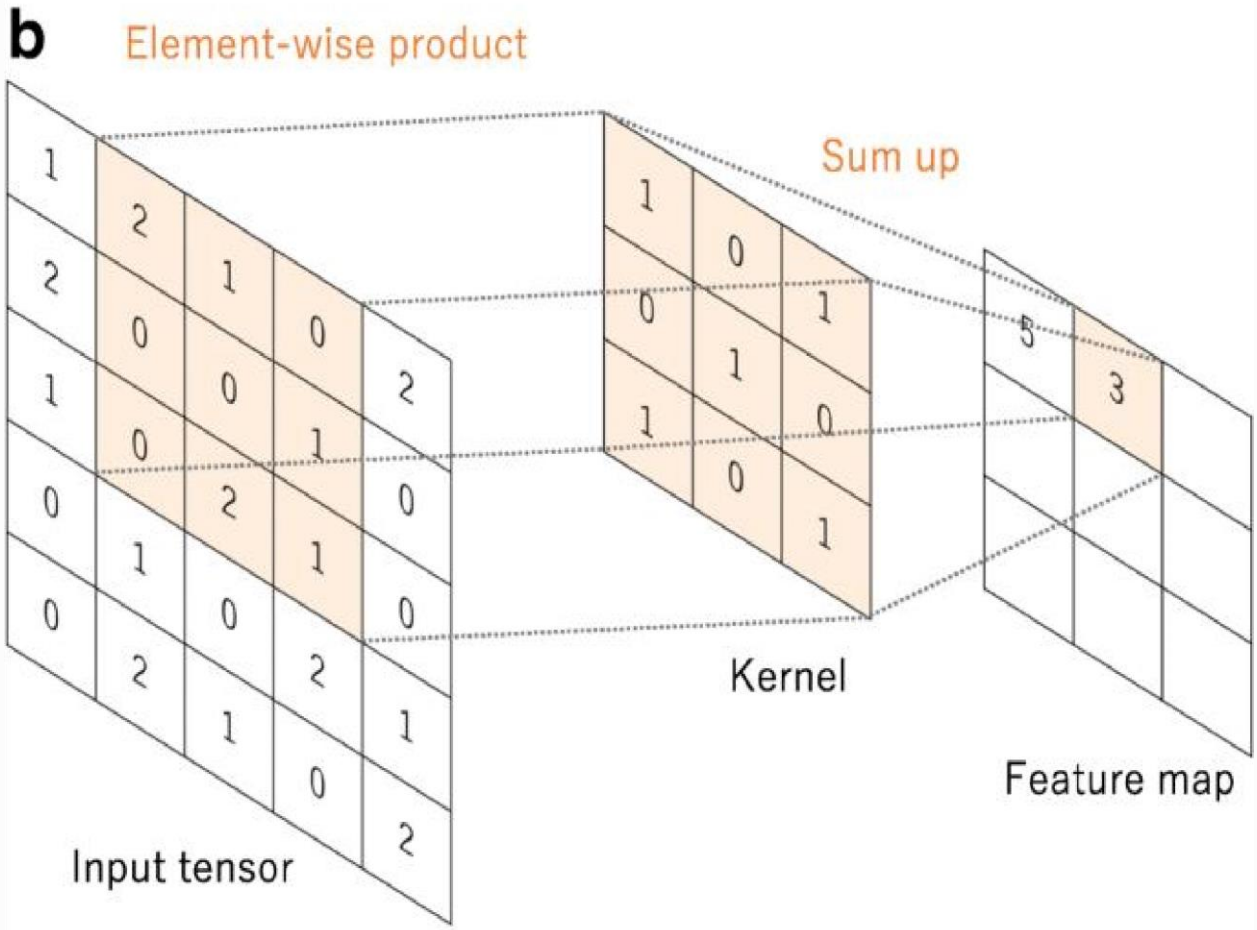
A model's performance under particular kernels and weights is calculated with a loss function through forward propagation on a training dataset, and learnable parameters, i.e., kernels and weights, are updated according to the loss value through backpropagation with gradient descent optimization algorithm. ReLU, rectified linear unit

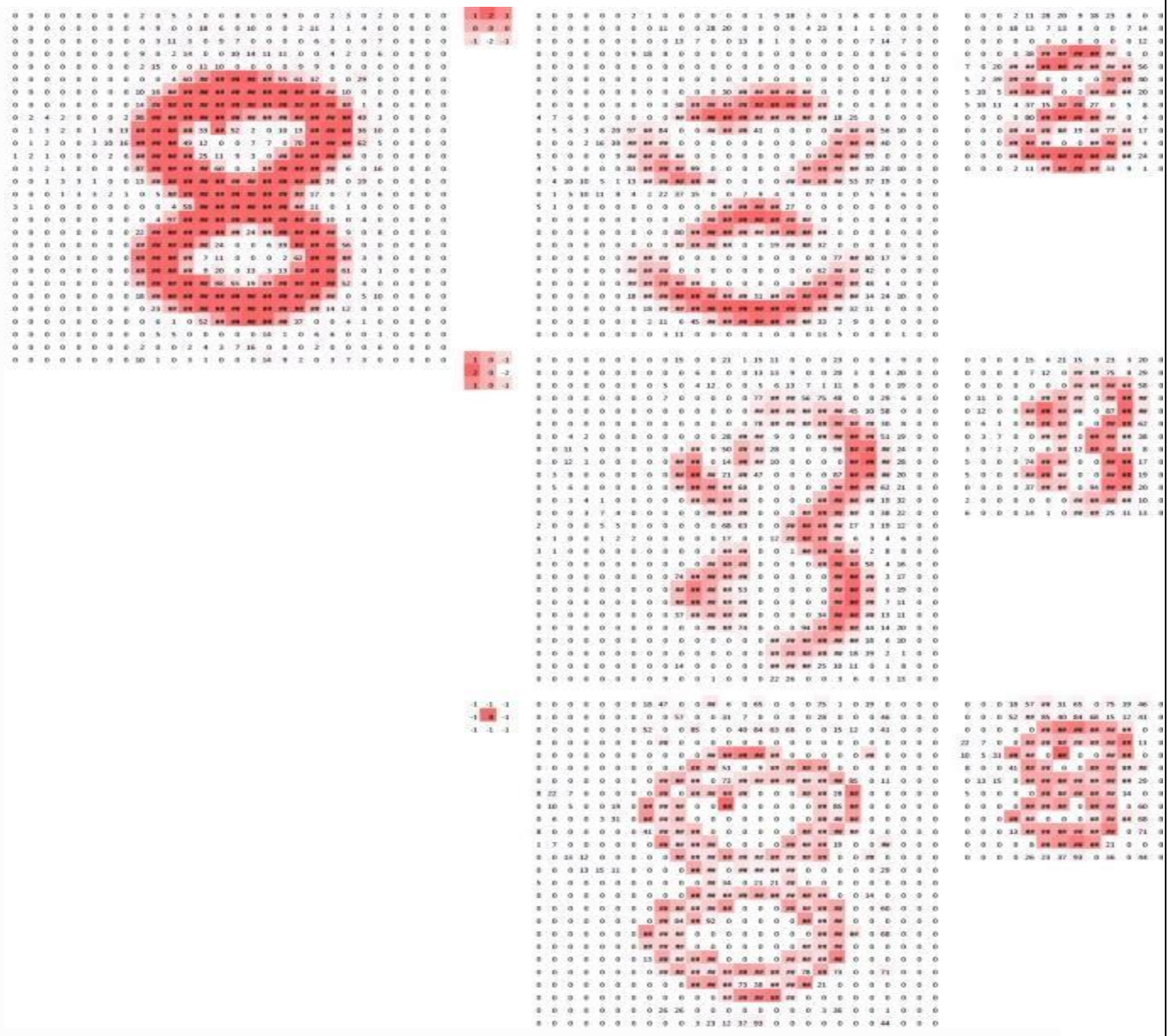
Convolution:-

Convolution is a specialized type of linear operation used for feature extraction, where a small array of numbers, called a kernel, is applied across the input, which is an array of numbers, called a tensor. An element-wise product between each element of the kernel and the input tensor is calculated at each location of the tensor and summed to obtain the output value in the corresponding position of the output tensor, called a feature map (Fig. **3**a–c). This procedure is repeated applying multiple kernels to form an arbitrary number of feature maps, which represent different characteristics of the input tensors; different kernels can, thus, be considered as different feature extractors (Fig. **3**d). Two key hyperparameters that define the convolution operation are size and number of kernels. The former is typically 3×3, but sometimes 5×5 or 7×7. The latter is arbitrary, and determines the depth of output feature maps.


Element-wise product
Sum up
Input tensor
Kernel
Feature map

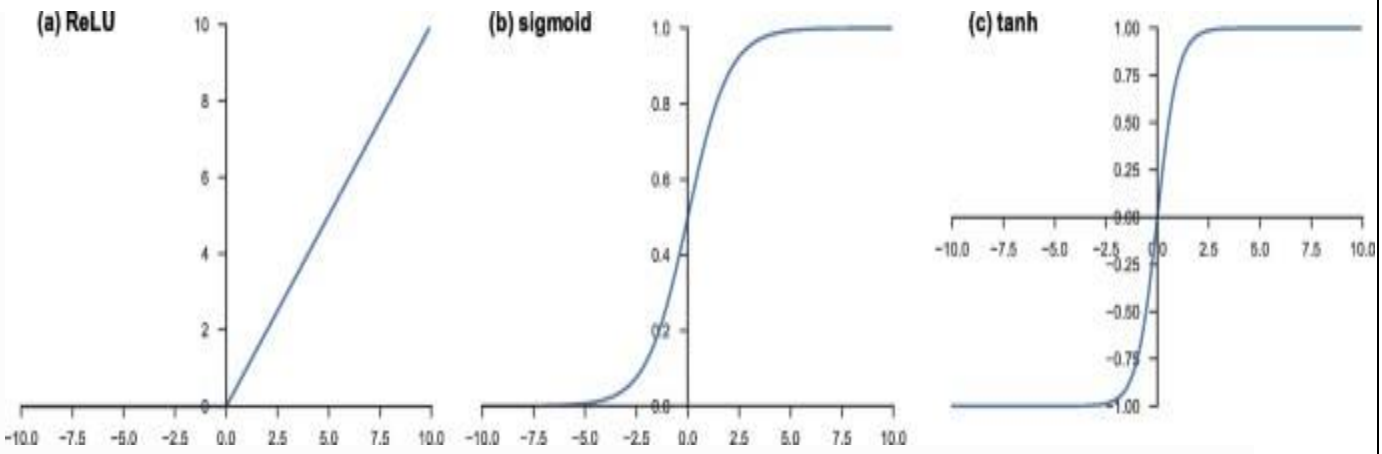
b


Element-wise product
Sum up
Input tensor
Kernel
Feature map



Nonlinear activation function:-

The outputs of a linear operation such as convolution are then passed through a nonlinear activation function. Although smooth nonlinear functions, such as sigmoid or hyperbolic tangent (tanh) function, were used previously because they are mathematical representations of a biological neuron behavior, the most common nonlinear activation function used presently is the rectified linear unit (ReLU), which simply computes the function: $f(x) = \max(0, x)$.



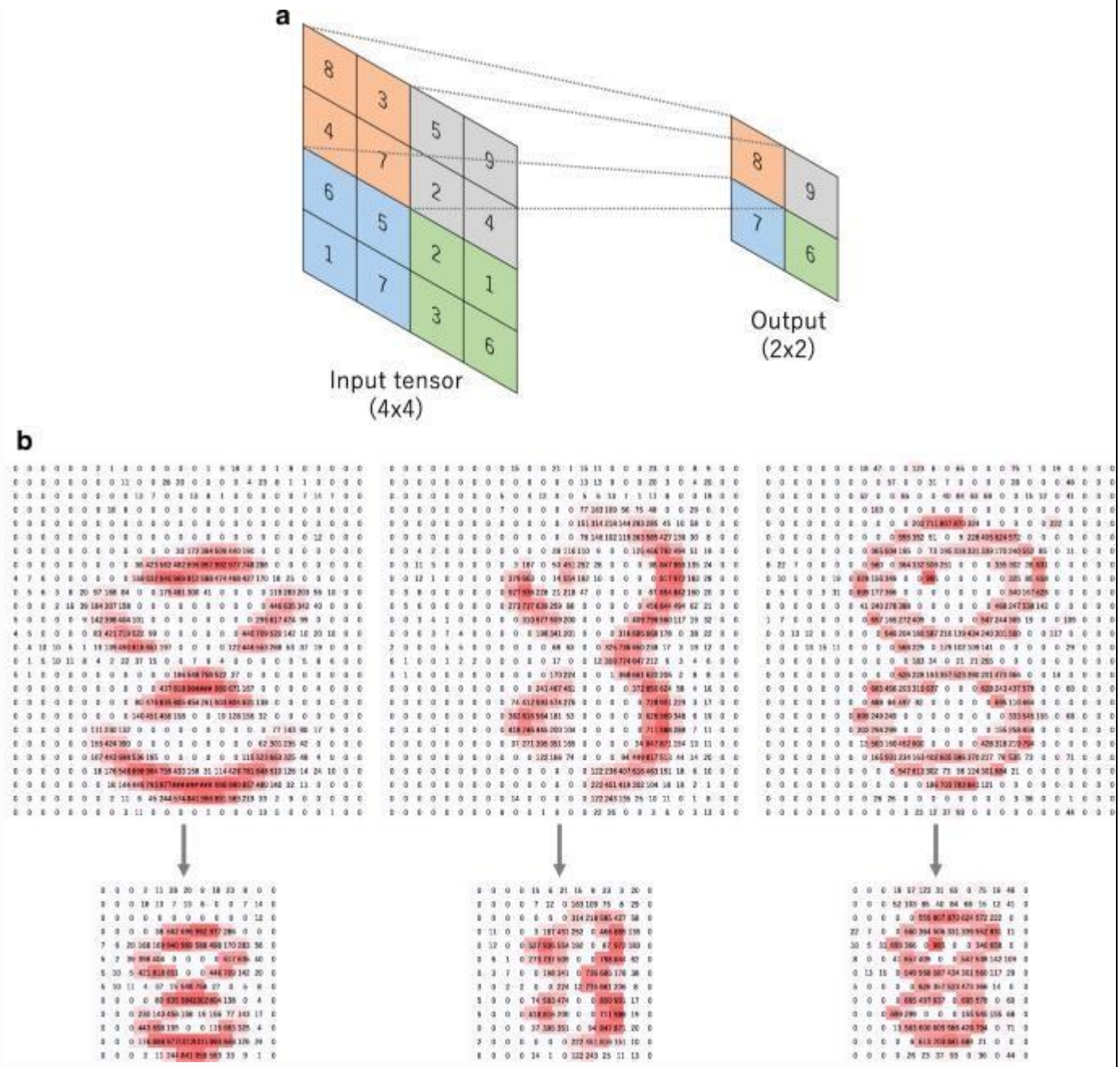
Activation functions commonly applied to neural networks: **a** rectified linear unit (ReLU), **b** sigmoid, and **c** hyperbolic tangent (tanh)

Pooling layer:-

A pooling layer provides a typical downsampling operation which reduces the in-plane dimensionality of the feature maps in order to introduce a translation invariance to small shifts and distortions, and decrease the number of subsequent learnable parameters. It is of note that there is no learnable parameter in any of the pooling layers, whereas filter size, stride, and padding are hyperparameters in pooling operations, similar to convolution operations.

Max pooling:-

The most popular form of pooling operation is max pooling, which extracts patches from the input feature maps, outputs the maximum value in each patch, and discards all the other values . A max pooling with a filter of size 2×2 with a stride of 2 is commonly used in practice. This downsamples the in-plane dimension of feature maps by a factor of 2. Unlike height and width, the depth dimension of feature maps remains unchanged.



a An example of max pooling operation with a filter size of 2 × 2, no padding, and a stride of 2, which extracts 2 × 2 patches from the input tensors, outputs the maximum value in each patch, and discards all the other values, resulting in downsampling the in-plane dimension of an input tensor by a factor of 2. **b** Examples of the max pooling operation on the same images in . Note that images in the upper row are downsampled by a factor of 2, from 26 × 26 to 13 × 13

Global average pooling:-

Another pooling operation worth noting is a global average pooling [**20**]. A global average pooling performs an extreme type of downsampling, where a feature map with size of height × width is downsampled into a 1 × 1 array by simply taking the average of all the elements in each feature map, whereas the depth of feature maps is retained. This operation is typically applied only once before the fully connected layers. The advantages of applying global average pooling are as follows: (1) reduces the number of learnable parameters and (2) enables the CNN to accept inputs of variable size.

Fully connected layer:-

The output feature maps of the final convolution or pooling layer is typically flattened, i.e., transformed into a one-dimensional (1D) array of numbers (or vector), and connected to one or more fully connected layers, also known as dense layers, in which every input is connected to every output by a learnable weight. Once the features extracted by the convolution layers and downsampled by the pooling layers are created, they are mapped by a subset of fully connected layers to the final outputs of the network, such as the probabilities for each class in classification tasks. The final fully connected layer typically has the same number of output nodes as the number of classes. Each fully connected layer is followed by a nonlinear function, such as ReLU, as described above.

Last layer activation function:-

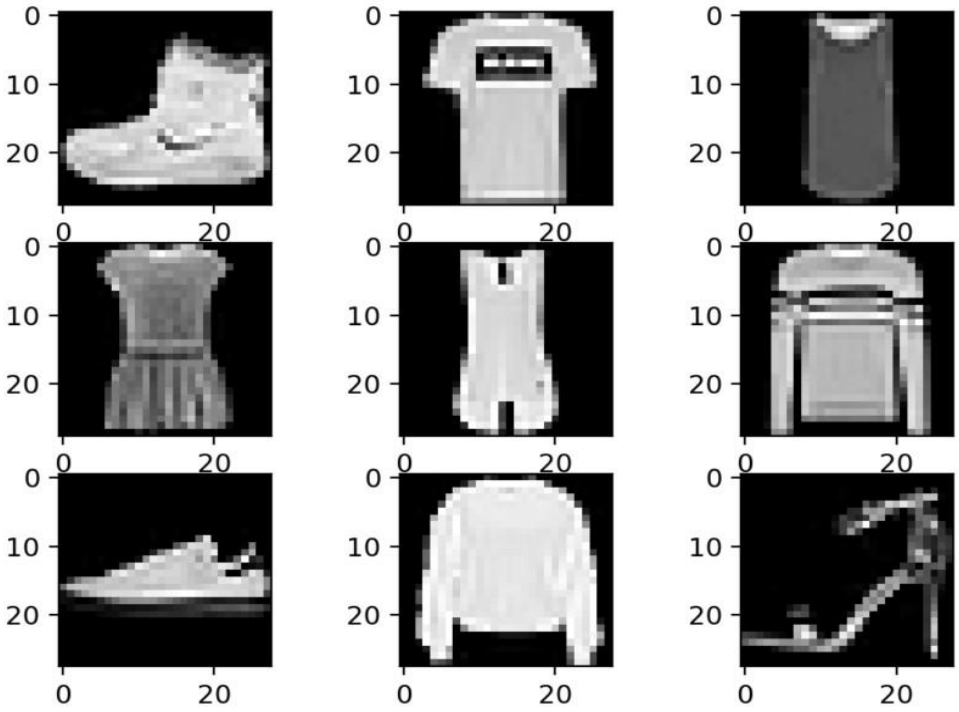
The activation function applied to the last fully connected layer is usually different from the others. An appropriate activation function needs to be selected according to each task. An activation function applied to the multiclass classification task is a softmax function which normalizes output real values from the last fully connected layer to target class probabilities, where each value ranges between 0 and 1 and all values sum to 1. Typical choices of the last layer activation function for various types of tasks are summarized

Fashion MNIST Clothing Classification:-

The Fashion-MNIST dataset is proposed as a more challenging replacement dataset for the MNIST dataset.

It is a dataset comprised of 60,000 small square 28×28 pixel grayscale images of items of 10 types of clothing, such as shoes, t-shirts, dresses, and more. The mapping of all 0-9 integers to class labels is listed below.

- 0: T-shirt/top
- 1: Trouser
- 2: Pullover
- 3: Dress
- 4: Coat
- 5: Sandal
- 6: Shirt
- 7: Sneaker
- 8: Bag
- 9: Ankle boot



4.METHODOLOGY:-

Images were loaded and preprocessed from the provided Fashion MNIST dataset. Each image, originally 28x28 pixels, was flattened to a 1D vector of pixels of length 784 and normalized. The images for each category were then stacked into a data matrix with dimensions of the number of examples by the flattened number of pixels. Output labels were transformed using one hot encoding. The training data was loaded from the provided file and split 80/20 into a training and validation set. The testing data was used as provided. The neural network parameters were loaded from a config file and implemented as detailed below.

4.1 Multi-Layer Neural Network Implementation:-

The multi-layer neural network was originally built with the specifications provided in the assignment. We used an input layer, two hidden layers each with 50 units and tanh activation, and a softmax output layer. The neural network comprised of three classes: Activation, Layer, and Neuralnetwork. The Activation class defined all the activation functions and their gradients as well as forward and backward pass functions. The activation functions were treated as an additional layer on top of a linear layer that computed the weight sum of inputs to that unit. The Layer class was for standard linear layers with both forward and backward passes. The forward function converted the input vector ”x” to an output variable ”a” to then be passed to the activation function. The backward function used the weighted sum of deltas and computed the weight and bias gradients. For each layer, weights and bias were initialized according to normal distributions with zero mean and unit standard deviation. They were updated by gradient descent with momentum and L2 regularization. Parameters (inputs, outputs, weights and gradients) for every layer were stored here. Lastly, the Neuralnetwork class created a list of Layers. Forward and backward functions defined in this class recursively call the forward and backward functions in Activation and Layer classes. Loss function and initial delta were normalized by number of classification classes (10 for Fashion MNIST) and number of examples in each mini-batch. This class was initialized with specifications given in the configuration file. These three classes were used for later tasks: numerical approximation, training, testing and experimentation. For training the neural network, we used mini-batch stochastic gradient descent as well as momentum in our update rule weighted by a term γ. We also implemented early stopping to use the weights 2 with the best performance on the validation set during the training period; however, for the purposes of generating graphs for this report, we allowed the model to run for the full number of epochs and stored the weights that best performed on the validation set. We used an initial learning rate of $1.0e-3$, mini-batch size of 128, number of epochs of 100, and momentum gamma of 0.9. After trying other values for these features in order to optimize the model, we found that changing the learning rate of $1.2e-2$ while leaving the other parameters the same gave the best performance.

4.2 Numerical Approximation:-

In order to confirm that our code was working, we computed the gradients using numerical approximation: $\frac{d}{d\omega} E(\omega) \approx E(w+) - E(w-) \; 2$ (1) with $= 10{-}2$ and compared the results with answers obtained from backpropagation. We used 10 examples, one in each category, and added 10 numbers in the end for comparison. Each final answer is computed with only one weight changed in the network for numerical approximation.

4.3 Network Optimization Strategies:-

4.3.1 Regularization:-

In order to further optimize our network, we experimented with a variety of aspects of the model. We initially implemented L2 (ridge) regularization, which works to penalize large weights by adding a sum of the weights squared to the loss term. This influences the weight update rule by adding a 2W term weighted by a regularization factor, 2λ. We varied the regularization term to optimize the model performance.

4.3.2 Activation Functions:-

Next, using the best performing regularization rate, we implemented various different hidden unit activation functions to compare to the loss and accuracy obtained by the tanh function. These included sigmoid, ReLU, and leakyReLU activation functions. In order to obtain good performance and avoid overflow errors with these activation functions, we decreased the learning rate by 2 orders of magnitude (1.2e-4) and decreased the batch size to one, leading to slower training

4.3.3 Network Topology:-

Lastly, we experimented by changing the neural network topology. This involved halving and doubling the number of hidden units in each layer to analyze the effect on performance. In addition to 50×50, we used the combinations of 50×100, 100×50, 25×50, and 50×25. We also changed the number of hidden layer to 1 and 3 instead of 2 while maintaining approximately the same number of parameters. This means we used 53 nodes in the hidden layer for the network with one hidden layer and 47 nodes in each layer in the 3 layer network.

5.CODE:-

[ ]:

```
1    #Fashion MNIST Data Classification
```

[ ]:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
import keras
```

2 load the data

```
[ ]: (X_train,Y_train),(X_test,Y_test)=tf.keras.datasets.fashion_mnist.load_data()
```

Downloading data from https://storage.googleapis.com/tensorflow/tf-kerasdatasets/train-labels-idx1-ubyte.gz

29515/29515 [==============================] - 0s 0us/step

Downloading data from https://storage.googleapis.com/tensorflow/tf-kerasdatasets/train-images-idx3-ubyte.gz

26421880/26421880 [==============================] - 0s 0us/step Downloading data from https://storage.googleapis.com/tensorflow/tf-kerasdatasets/t10k-labels-idx1-ubyte.gz

5148/5148 [==============================] - 0s 0us/step

Downloading data from https://storage.googleapis.com/tensorflow/tf-kerasdatasets/t10k-images-idx3-ubyte.gz

4422102/4422102 [==============================] - 0s 0us/step

3 print shape

```
[ ]: X_train.shape,Y_train.shape,"************",X_test.shape,Y_test.shape
```

```
[ ]: ((60000, 28, 28), (60000,), '************', (10000, 28, 28), (10000,))
```

```
[ ]: X_train[    0]
```

```
[ ]: array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,
         0,  0, 13, 73,  0,  0,  1,  4,  0,  0,  0,  0,  1,
         1,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  3,
         0, 36, 136, 127, 62, 54,  0,  0,  0,  1,  3,  4,  0,
         0,  3],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  6,
```

0, 102, 204, 176, 134, 144, 123, 23, 10, 0], 0, 0, 0, 0, 12,

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 155, 236, 207, 178, 107, 156, 161, 109, 64, 23, 77, 130, 72, 15],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,

69, 207, 223, 218, 216, 216, 163, 127, 121, 122, 146, 141, 88, 172, 66],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,

200, 232, 232, 233, 229, 223, 223, 215, 213, 164, 127, 123, 196, 229, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

183, 225, 216, 223, 228, 235, 227, 224, 222, 224, 221, 223, 245, 173, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

193, 228, 218, 213, 198, 180, 212, 210, 211, 213, 223, 220, 243, 202, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 3, 0, 12,

219, 220, 212, 218, 192, 169, 227, 208, 218, 224, 212, 226, 197, 209, 52],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 99,

244, 222, 220, 218, 203, 198, 221, 215, 213, 222, 220, 245, 119, 167, 56],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 55,

236, 228, 230, 228, 240, 232, 213, 218, 223, 234, 217, 217, 209, 92, 0],

[0, 0, 1, 4, 6, 7, 2, 0, 0, 0, 0, 0, 237,

226, 217, 223, 222, 219, 222, 221, 216, 223, 229, 215, 218, 255, 77, 0],

[0, 3, 0, 0, 0, 0, 0, 0, 0, 62, 145, 204, 228,

207, 213, 221, 218, 208, 211, 218, 224, 223, 219, 215, 224, 244, 159, 0],

[0, 0, 0, 0, 18, 44, 82, 107, 189, 228, 220, 222, 217,

226, 200, 205, 211, 230, 224, 234, 176, 188, 250, 248, 233, 238, 215, 0],

[0, 57, 187, 208, 224, 221, 224, 208, 204, 214, 208, 209, 200,

159, 245, 193, 206, 223, 255, 255, 221, 234, 221, 211, 220, 232, 246, 0],

[3, 202, 228, 224, 221, 211, 211, 214, 205, 205, 205, 220, 240,

80, 150, 255, 229, 221, 188, 154, 191, 210, 204, 209, 222, 228, 225, 0],

[98, 233, 198, 210, 222, 229, 229, 234, 249, 220, 194, 215, 217,

241, 65, 73, 106, 117, 168, 219, 221, 215, 217, 223, 223, 224, 229, 29],

[75, 204, 212, 204, 193, 205, 211, 225, 216, 185, 197, 206, 198,

```
  213, 240, 195, 227, 245, 239, 223, 218, 212, 209, 222, 220, 221, 230, 67],
 [ 48, 203, 183, 194, 213, 197, 185, 190, 194, 192, 202, 214, 219,
  221, 220, 236, 225, 216, 199, 206, 186, 181, 177, 172, 181, 205, 206, 115],
 [ 0, 122, 219, 193, 179, 171, 183, 196, 204, 210, 213, 207, 211,
  210, 200, 196, 194, 191, 195, 191, 198, 192, 176, 156, 167, 177, 210, 92],
  [ 0,   0, 74, 189, 212, 191, 175, 172, 175, 181, 185, 188, 189,
  188, 193, 198, 204, 209, 210, 210, 211, 188, 188, 194, 192, 216, 170,   0],
  [ 2,   0,   0,   0, 66, 200, 222, 237, 239, 242, 246, 243, 244,
  221, 220, 193, 191, 179, 182, 182, 181, 176, 166, 168, 99, 58, 0, 0],
 [ 0,   0,   0,   0,   0,   0,   0, 40, 61, 44, 72, 41, 35,
      0, 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
      0, 0],
 [ 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
      0, 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
      0, 0],
 [ 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
      0, 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
        0, 0]], dtype=uint8)
```

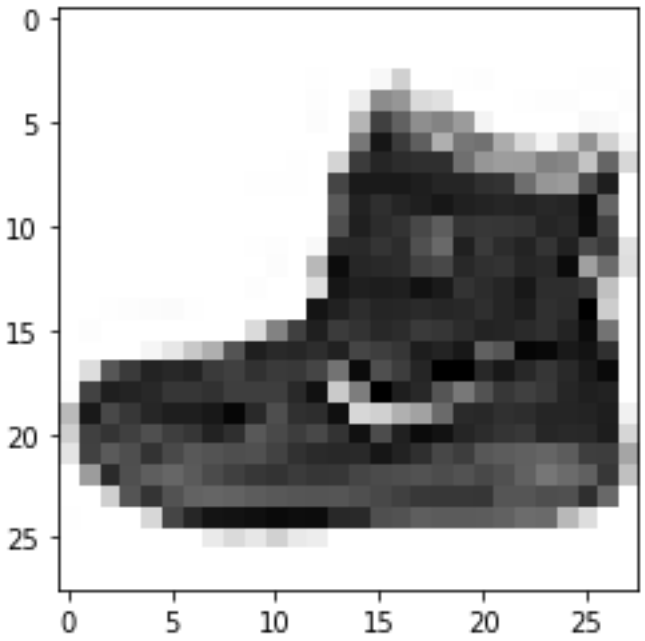
[ ]:
```
Y_train[   0]
```

[ ]: 9

[ ]:
```
Class_labels=("T-shirt/
,→top","trouser","pullover","dress","coat","sandal","shirt","sneaker","bag","Ankle␣ ,→boot")
```

[ ]:
```
plt . imshow(X_train[   0] ,Cmap =' Greys  ' )
```

[ ]: <matplotlib.image.AxesImage at 0x7f3db8d87f90>



[ ]:
```
plt . figure(figsize =( 16 , 16 ))
j =1
for i in np. random . randint( 0, 1000 , 25 ):
    plt . subplot( 5, 5,j);j +=1
    plt . imshow(X_train[i],cmap =' Greys ' )
    plt . axis( ' off ' )
    plt . title( ' {} / {}' . format(Class_labels[Y_train[i]],Y_train[i]))
```

[ ]:
```
X_train . ndim
```

[ ]: 3

[ ]:
```
X_train =np. expand_dims(X_train, - 1)
```

[ ]:
```
X_test =np. expand_dims(X_test, - 1)
```

4 features Scaling

[ ]:
```
X_train =X_train / 255
X_test =X_test / 255
```

5 split dataset

[ ]:
```
from sklearn.model_selection import train_test_split
X_train,x_validation,Y_train,y_validation=train_test_split(X_train,Y_train,test_size=0.2,random_state=2020)
```

6 building a *CNN* model

Model: "sequential"

Layer (type)	Output Shape	Param #

[ ]:
```
model =keras . models . Sequential([
    keras . layers .
Conv2D(filters =32 ,kernel_size =3,strides =( 1, 1) ,padding =' valid ' ,activation =' relu ' ,input_shape
    keras . layers . MaxPooling2D(pool_size =( 2, 2)) ,
    keras . layers . Flatten(),
    keras . layers . Dense(units =128 ,activation =' relu ' ) ,
    keras . layers . Dense(units =10 ,activation =' softmax ' )
])
```

[ ]:
```
model . summary()
```

conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D	(None, 13, 13, 32))	0
flatten (Flatten)	(None, 5408)	0
dense (Dense)	(None, 128)	692352
dense_1 (Dense)	(None, 10)	1290

Total params: 693,962

Trainable params: 693,962

Non-trainable params: 0

[ ]: model.

```
,→compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

[ ]:
```
model.
,→fit(X_train,Y_train,batch_size=512,epochs=10,verbose=1,validation_data=(x_validation,y_valid
```

```
Epoch 1/10
94/94 [==============================] - 43s 457ms/step - loss: 0.6312 -
accuracy: 0.7884 - val_loss: 0.4296 - val_accuracy: 0.8492
Epoch 2/10
94/94 [==============================] - 20s 210ms/step - loss: 0.3783 -
accuracy: 0.8670 - val_loss: 0.3811 - val_accuracy: 0.8627
Epoch 3/10
94/94 [==============================] - 20s 211ms/step - loss: 0.3270 -
accuracy: 0.8858 - val_loss: 0.3361 - val_accuracy: 0.8842
Epoch 4/10
94/94 [==============================] - 26s 278ms/step - loss: 0.3037 -
accuracy: 0.8924 - val_loss: 0.3116 - val_accuracy: 0.8928
Epoch 5/10
94/94 [==============================] - 20s 210ms/step - loss: 0.2808 -
accuracy: 0.9007 - val_loss: 0.2971 - val_accuracy: 0.8961
Epoch 6/10
94/94 [==============================] - 22s 230ms/step - loss: 0.2584 -
accuracy: 0.9086 - val_loss: 0.2906 - val_accuracy: 0.8991
Epoch 7/10
94/94 [==============================] - 20s 209ms/step - loss: 0.2456 -
accuracy: 0.9119 - val_loss: 0.2889 - val_accuracy: 0.8974
Epoch 8/10
94/94 [==============================] - 20s 209ms/step - loss: 0.2325 -
accuracy: 0.9175 - val_loss: 0.2917 - val_accuracy: 0.9000
Epoch 9/10
94/94 [==============================] - 21s 222ms/step - loss: 0.2248 accuracy: 0.9190 -
val_loss: 0.2745 - val_accuracy: 0.9067
```

Epoch 10/10

94/94 [==============================] - 22s 231ms/step - loss: 0.2119 accuracy: 0.9239 - val_loss: 0.2904 - val_accuracy: 0.8972

[ ]: <keras.callbacks.History at 0x7f3db80d3390>

[ ]:

[ ]:
```
y_pred =model . predict(X_test)
y_pred . round( 2)
```

313/313 [==============================] - 2s 7ms/step

[ ]: array([[0. , 0. , 0. , ..., 0.01, 0. , 0.99], [0. , 0. , 1. , ..., 0. , 0. , 0.
],

[0. , 1. , 0. , ..., 0. , 0. , 0.],

...,

[0. , 0. , 0. , ..., 0. , 0.99, 0.],

[0. , 1. , 0. , ..., 0. , 0. , 0.],

[0. , 0. , 0. , ..., 0.22, 0.04, 0.01]], dtype=float32)

[ ]:
```
Y_test
```

[ ]: array([9, 2, 1, ..., 8, 1, 5], dtype=uint8)

[ ]:
```
model . evaluate(X_test,Y_test)
```

313/313 [==============================] - 2s 7ms/step - loss: 0.2946 accuracy: 0.8902

[ ]: [0.2946320176124573, 0.8902000188827515]

[ ]:
```
plt.figure(figsize=(16,16)) j=1 for i in
np.random.randint(0,1000,25):
    plt.subplot(5,5,j);j+=1
    plt.imshow(X_test[i].reshape(28,28),cmap='Greys') plt.title('Actual={}/{}
    \npredicted={}/{}'.
 ,→format(Class_labels[Y_test[i]],Y_test[i],Class_labels[np.
 ,→argmax(y_pred[i])],np.argmax(y_pred[i]))) plt.axis('off')
```

[ ]:
```
plt.figure(figsize=(16,30))  j=1  for  i  in
 np.random.randint(0,1000,25):
    plt.subplot(10,6,j);j+=1

    plt.imshow(X_test[i].reshape(28,28),cmap='Greys') plt.title('Actual={}/{}
    \npredicted={}/{}'.

                ,→format(Class_labels[Y_test[i]],Y_test[i],Class_labels[np.
  ,→argmax(y_pred[i])],np.argmax(y_pred[i]))) plt.axis('off')
```

7 confusion matrix

[ ]:
```
from sklearn.metrics import confusion_matrix
plt.figure(figsize=(16,9)) y_pred_labels=[np.argmax(label) for
label in y_pred] cm=confusion_matrix(Y_test,y_pred_labels)
```

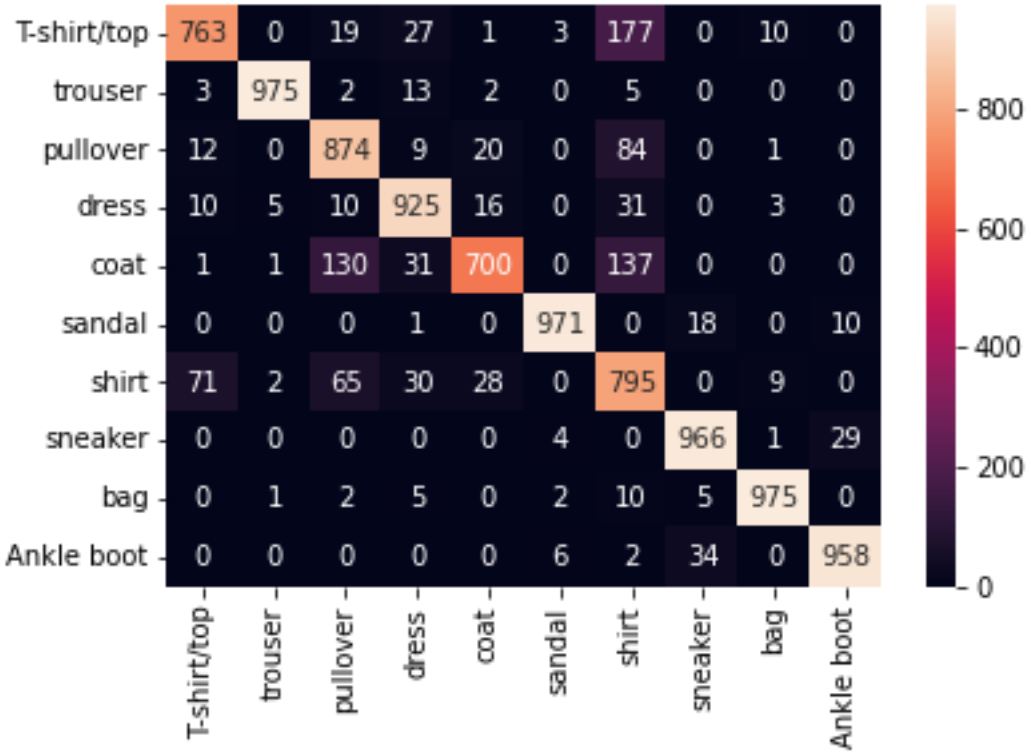
```
<Figure size 1152x648 with 0 Axes >
```

[ ]:
```
sns .
heatmap(cm,annot =True,fmt =' d' ,xticklabels =Class_labels,yticklabels =Class_labels)

from sklearn.metrics import classification_report
cr=classification_report(Y_test,y_pred_labels,target_names=Class_labels) print(cr)
```

```
              precision    recall  f1-score   support
```

T-shirt/top	0.89	0.76	0.82	1000
trouser	0.99	0.97	0.98	1000
pullover	0.79	0.87	0.83	1000
dress	0.89	0.93	0.91	1000
coat	0.91	0.70	0.79	1000
sandal	0.98	0.97	0.98	1000
shirt	0.64	0.80	0.71	1000
sneaker	0.94	0.97	0.96	1000
bag	0.98	0.97	0.98	1000
Ankle boot	0.96	0.96	0.96	1000
accuracy			0.89	10000
macro avg	0.90	0.89	0.89	10000
weighted avg	0.90	0.89	0.89	10000



8 saving the model

[]:
```
from google.colab import drive
drive . mount( ' /content/drive ' )
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).

[]:
```
model . save(  ' fashion_mnist_cnn_model.h5          ' )
```

9 Building 2 complex CNN

[ ]:
```
#Building CNN model

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
import keras

cnn_model2=keras.models.Sequential([
    keras.layers.
 ↪Conv2D(filters=32,kernel_size=3,strides=(1,1),padding='valid',activation='relu',input_shape
    keras.layers.MaxPooling2D(pool_size=(2,2)),
    keras.layers.
 ↪Conv2D(filters=64,kernel_size=3,strides=(1,1),padding='same',activation='relu'),
    keras.layers.MaxPooling2D(pool_size=(2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(units=128,activation='relu'),
    keras.layers.Dropout(0.25),
    keras.layers.Dense(units=256,activation='relu'),
    keras.layers.Dropout(0.25),
    keras.layers.Dense(units=128,activation='relu'),
    keras.layers.Dense(units=10,activation='softmax')
])

#compile the model
cnn_model2.
 ↪compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

[ ]:
```

```

10 Train the model

[ ]:
```
cnn_model2.
    fit(X_train,Y_train,batch_size=512,epochs=20,verbose=1,validation_data=(x_validation,y_valid
```

```
Epoch 1/20
94/94 [==============================] - 45s 473ms/step - loss: 0.8822 -
accuracy: 0.6667 - val_loss: 0.5113 - val_accuracy: 0.8043
Epoch 2/20
94/94 [==============================] - 44s 465ms/step - loss: 0.4875 -
accuracy: 0.8175 - val_loss: 0.4094 - val_accuracy: 0.8487
Epoch 3/20
94/94 [==============================] - 43s 462ms/step - loss: 0.3996 -
accuracy: 0.8556 - val_loss: 0.3474 - val_accuracy: 0.8742
Epoch 4/20
94/94 [==============================] - 45s 482ms/step - loss: 0.3530 -
accuracy: 0.8709 - val_loss: 0.3174 - val_accuracy: 0.8868
Epoch 5/20
94/94 [==============================] - 43s 461ms/step - loss: 0.3150 -
accuracy: 0.8849 - val_loss: 0.2947 - val_accuracy: 0.8940
Epoch 6/20
94/94 [==============================] - 43s 462ms/step - loss: 0.2972 -
accuracy: 0.8916 - val_loss: 0.2831 - val_accuracy: 0.8978
Epoch 7/20
94/94 [==============================] - 43s 461ms/step - loss: 0.2731 -
accuracy: 0.8988 - val_loss: 0.2764 - val_accuracy: 0.8995
Epoch 8/20
94/94 [==============================] - 44s 464ms/step - loss: 0.2591 -
accuracy: 0.9052 - val_loss: 0.2710 - val_accuracy: 0.8999
Epoch 9/20
94/94 [==============================] - 43s 460ms/step - loss: 0.2457 -
```

accuracy: 0.9106 - val_loss: 0.2591 - val_accuracy: 0.9079

Epoch 10/20

94/94 [==============================] - 46s 495ms/step - loss: 0.2293 accuracy: 0.9154 - val_loss: 0.2429 - val_accuracy: 0.9128

Epoch 11/20

94/94 [==============================] - 45s 480ms/step - loss: 0.2160 accuracy: 0.9211 - val_loss: 0.2432 - val_accuracy: 0.9163

Epoch 12/20

94/94 [==============================] - 44s 469ms/step - loss: 0.2072 accuracy: 0.9239 - val_loss: 0.2445 - val_accuracy: 0.9122

Epoch 13/20

94/94 [==============================] - 44s 465ms/step - loss: 0.1978 accuracy: 0.9269 - val_loss: 0.2351 - val_accuracy: 0.9170

Epoch 14/20

94/94 [==============================] - 44s 464ms/step - loss: 0.1849 accuracy: 0.9320 - val_loss: 0.2430 - val_accuracy: 0.9166

Epoch 15/20

94/94 [==============================] - 44s 465ms/step - loss: 0.1785 -

accuracy: 0.9333 - val_loss: 0.2327 - val_accuracy: 0.9197

Epoch 16/20

94/94 [==============================] - 46s 486ms/step - loss: 0.1664 -

accuracy: 0.9379 - val_loss: 0.2532 - val_accuracy: 0.9167

Epoch 17/20

94/94 [==============================] - 44s 464ms/step - loss: 0.1641 -

accuracy: 0.9398 - val_loss: 0.2304 - val_accuracy: 0.9230

Epoch 18/20

94/94 [==============================] - 43s 462ms/step - loss: 0.1542 -

accuracy: 0.9427 - val_loss: 0.2425 - val_accuracy: 0.9164

Epoch 19/20

94/94 [==============================] - 43s 462ms/step - loss: 0.1430 -

accuracy: 0.9474 - val_loss: 0.2387 - val_accuracy: 0.9224

Epoch 20/20

94/94 [==============================] - 43s 462ms/step - loss: 0.1359 accuracy: 0.9497 - val_loss: 0.2540 - val_accuracy: 0.9180

[ ]: <keras.callbacks.History at 0x7f3dada0d890>

[ ]:
```
cnn_model2.save('fashion_mnist_cnn_model2.h5')
```

11 Building CNN model 3

[ ]:
```
#Building CNN model3

cnn_model3=keras.models.Sequential([ keras.layers.
    Conv2D(filters=64,kernel_size=3,strides=(1,1),padding='valid',activation='relu',input_shape
    keras.layers.MaxPooling2D(pool_size=(2,2)), keras.layers.
    Conv2D(filters=128,kernel_size=3,strides=(1,1),padding='same',activation='relu'),
    keras.layers.MaxPooling2D(pool_size=(2,2)), keras.layers.
    Conv2D(filters=64,kernel_size=3,strides=(1,1),padding='same',activation='relu'), keras.layers.Flatten(),

    keras.layers.Dense(units=128,activation='relu'), keras.layers.Dropout(0.25),

    keras.layers.Dense(units=256,activation='relu'), keras.layers.Dropout(0.5),

    keras.layers.Dense(units=256,activation='relu'), keras.layers.Dropout(0.25),

    keras.layers.Dense(units=128,activation='relu'), keras.layers.Dropout(0.10),
```

Epoch 1/50

94/94 [==============================] - 135s 1s/step - loss: 1.0337 - accuracy: 0.6014 - val_loss: 0.5371 - val_accuracy: 0.7875

Epoch 2/50

94/94 [==============================] - 133s 1s/step - loss: 0.5253 - accuracy: 0.8038 - val_loss: 0.4208 - val_accuracy: 0.8400

Epoch 3/50

94/94 [==============================] - 135s 1s/step - loss: 0.4270 - accuracy: 0.8475 - val_loss: 0.3514 - val_accuracy: 0.8698

Epoch 4/50

94/94 [==============================] - 132s 1s/step - loss: 0.3560 - accuracy: 0.8744 - val_loss: 0.3043 - val_accuracy: 0.8902

Epoch 5/50

94/94 [==============================] - 134s 1s/step - loss: 0.3139 - accuracy:

```
        keras . layers . Dense(units =10 ,activation =' softmax ' )
])

#compile the model
cnn_model3 .
 ,→ compile(optimizer =' adam' ,loss =' sparse_categorical_crossentropy ' ,metrics =[ ' accuracy ' ])

#Train the model
cnn_model3 .
 ,→ fit(X_train,Y_train,batch_size =512 ,epochs =50 ,verbose =1,validation_data =( x_validation,y_valid

#saving the model
cnn_model3 . save( ' fashion_mnist_cnn_model3.h5 ' )

cnn_model3 . evaluate(X_test,Y_test)
```

0.8898 - val_loss: 0.2911 - val_accuracy: 0.8933

Epoch 6/50

94/94 [==============================] - 132s 1s/step - loss: 0.2880 - accuracy:

0.8972 - val_loss: 0.2776 - val_accuracy: 0.8996

Epoch 7/50

94/94 [==============================] - 134s 1s/step - loss: 0.2675 - accuracy:

0.9055 - val_loss: 0.2602 - val_accuracy: 0.9068

Epoch 8/50

94/94 [==============================] - 132s 1s/step - loss: 0.2451 - accuracy:

0.9134 - val_loss: 0.2607 - val_accuracy: 0.9074

Epoch 9/50

94/94 [==============================] - 133s 1s/step - loss: 0.2298 - accuracy:

0.9193 - val_loss: 0.2541 - val_accuracy: 0.9112

Epoch 10/50

94/94 [==============================] - 131s 1s/step - loss: 0.2135 - accuracy:

0.9251 - val_loss: 0.2369 - val_accuracy: 0.9166 Epoch 11/50

94/94 [==============================] - 134s 1s/step - loss: 0.1975 - accuracy:

0.9295 - val_loss: 0.2432 - val_accuracy: 0.9170

Epoch 12/50

94/94 [==============================] - 132s 1s/step - loss: 0.1848 - accuracy:

0.9338 - val_loss: 0.2400 - val_accuracy: 0.9190

Epoch 13/50

94/94 [==============================] - 133s 1s/step - loss: 0.1748 - accuracy:

0.9380 - val_loss: 0.2561 - val_accuracy: 0.9152

Epoch 14/50

94/94 [==============================] - 131s 1s/step - loss: 0.1656 - accuracy:

0.9408 - val_loss: 0.2398 - val_accuracy: 0.9227

Epoch 15/50

94/94 [==============================] - 133s 1s/step - loss: 0.1554 - accuracy:

0.9445 - val_loss: 0.2395 - val_accuracy: 0.9218

Epoch 16/50

94/94 [==============================] - 131s 1s/step - loss: 0.1480 - accuracy:

0.9477 - val_loss: 0.2461 - val_accuracy: 0.9202

Epoch 17/50

94/94 [==============================] - 133s 1s/step - loss: 0.1338 - accuracy:

0.9519 - val_loss: 0.2469 - val_accuracy: 0.9212

Epoch 18/50

94/94 [==============================] - 128s 1s/step - loss: 0.1280 - accuracy:

0.9548 - val_loss: 0.2488 - val_accuracy: 0.9238

Epoch 19/50

94/94 [==============================] - 130s 1s/step - loss: 0.1199 - accuracy:

0.9573 - val_loss: 0.2622 - val_accuracy: 0.9185

Epoch 20/50

94/94 [==============================] - 130s 1s/step - loss: 0.1121 - accuracy:

0.9599 - val_loss: 0.2553 - val_accuracy: 0.9239

Epoch 21/50

94/94 [==============================] - 132s 1s/step - loss: 0.1014 - accuracy:

0.9642 - val_loss: 0.2800 - val_accuracy: 0.9216

Epoch 22/50

94/94 [==============================] - 130s 1s/step - loss: 0.1006 - accuracy:

0.9630 - val_loss: 0.2761 - val_accuracy: 0.9183

Epoch 23/50

94/94 [==============================] - 131s 1s/step - loss: 0.0925 - accuracy:

0.9666 - val_loss: 0.2839 - val_accuracy: 0.9224

Epoch 24/50

94/94 [==============================] - 129s 1s/step - loss: 0.0816 - accuracy:

0.9702 - val_loss: 0.3004 - val_accuracy: 0.9232

Epoch 25/50

94/94 [==============================] - 131s 1s/step - loss: 0.0854 - accuracy:

0.9686 - val_loss: 0.2957 - val_accuracy: 0.9216

Epoch 26/50

94/94 [==============================] - 129s 1s/step - loss: 0.0825 - accuracy:

0.9700 - val_loss: 0.3139 - val_accuracy: 0.9239 Epoch 27/50

94/94 [==============================] - 131s 1s/step - loss: 0.0700 - accuracy:

0.9745 - val_loss: 0.3200 - val_accuracy: 0.9246

Epoch 28/50

94/94 [==============================] - 129s 1s/step - loss: 0.0695 - accuracy:

0.9751 - val_loss: 0.3336 - val_accuracy: 0.9218

Epoch 29/50

94/94 [==============================] - 131s 1s/step - loss: 0.0682 - accuracy:

0.9746 - val_loss: 0.3110 - val_accuracy: 0.9236

Epoch 30/50

94/94 [==============================] - 130s 1s/step - loss: 0.0652 - accuracy:

0.9771 - val_loss: 0.3439 - val_accuracy: 0.9247

Epoch 31/50

94/94 [==============================] - 132s 1s/step - loss: 0.0603 - accuracy:

0.9786 - val_loss: 0.3235 - val_accuracy: 0.9250

Epoch 32/50

94/94 [==============================] - 130s 1s/step - loss: 0.0592 - accuracy:

0.9793 - val_loss: 0.3352 - val_accuracy: 0.9243

Epoch 33/50

94/94 [==============================] - 131s 1s/step - loss: 0.0545 - accuracy:

0.9806 - val_loss: 0.3391 - val_accuracy: 0.9243

Epoch 34/50

94/94 [==============================] - 130s 1s/step - loss: 0.0510 - accuracy:

0.9820 - val_loss: 0.3915 - val_accuracy: 0.9224

Epoch 35/50

94/94 [==============================] - 132s 1s/step - loss: 0.0532 - accuracy:

0.9811 - val_loss: 0.3574 - val_accuracy: 0.9219

Epoch 36/50

94/94 [==============================] - 130s 1s/step - loss: 0.0512 - accuracy:

0.9823 - val_loss: 0.3516 - val_accuracy: 0.9232

Epoch 37/50

94/94 [==============================] - 131s 1s/step - loss: 0.0445 - accuracy:

0.9845 - val_loss: 0.3858 - val_accuracy: 0.9229

Epoch 38/50

94/94 [==============================] - 129s 1s/step - loss: 0.0418 - accuracy:

0.9848 - val_loss: 0.3953 - val_accuracy: 0.9213

Epoch 39/50

94/94 [==============================] - 131s 1s/step - loss: 0.0423 - accuracy:

0.9850 - val_loss: 0.3715 - val_accuracy: 0.9213

Epoch 40/50

94/94 [==============================] - 129s 1s/step - loss: 0.0430 - accuracy:

0.9846 - val_loss: 0.4037 - val_accuracy: 0.9227

Epoch 41/50

94/94 [==============================] - 132s 1s/step - loss: 0.0410 - accuracy:

0.9856 - val_loss: 0.3695 - val_accuracy: 0.9223

Epoch 42/50

94/94 [==============================] - 129s 1s/step - loss: 0.0352 - accuracy:

0.9876 - val_loss: 0.4111 - val_accuracy: 0.9190 Epoch 43/50

94/94 [==============================] - 131s 1s/step - loss: 0.0387 - accuracy:

0.9867 - val_loss: 0.3964 - val_accuracy: 0.9232

Epoch 44/50

94/94 [==============================] - 130s 1s/step - loss: 0.0364 - accuracy:

0.9873 - val_loss: 0.3840 - val_accuracy: 0.9222

Epoch 45/50

94/94 [==============================] - 132s 1s/step - loss: 0.0365 - accuracy:

0.9875 - val_loss: 0.4024 - val_accuracy: 0.9249

Epoch 46/50

94/94 [==============================] - 130s 1s/step - loss: 0.0305 - accuracy:

0.9896 - val_loss: 0.3784 - val_accuracy: 0.9246

Epoch 47/50

94/94 [==============================] - 131s 1s/step - loss: 0.0330 - accuracy:

0.9889 - val_loss: 0.3946 - val_accuracy: 0.9220

Epoch 48/50

94/94 [==============================] - 130s 1s/step - loss: 0.0308 - accuracy:

0.9892 - val_loss: 0.4339 - val_accuracy: 0.9217

Epoch 49/50

94/94 [==============================] - 131s 1s/step - loss: 0.0308 - accuracy:

0.9891 - val_loss: 0.4207 - val_accuracy: 0.9247

Epoch 50/50

94/94 [==============================] - 129s 1s/step - loss: 0.0271 - accuracy:

0.9906 - val_loss: 0.4714 - val_accuracy: 0.9209

313/313 [==============================] - 9s 27ms/step - loss: 0.4871 accuracy: 0.9178

[ ]: [0.4870823621749878, 0.9178000092506409]

6.CONCLUSION:-

In this project, i applied various classification methods on an image classification problem. i have explained why the CNNs are the best method we can employ out of considered ones, and why do the other methods fail. Some of the reasons why CNNs are the most practical and usually the most accurate method are:

- They can transfer learning through layers, saving inferences, and making new ones on subsequent layers.
- No need for feature extraction before using the algorithm, it is done during training.
- It recognizes important features.

However, they also have their caveats. They are known to fail on images that are rotated and scaled differently, which is not the case here, as the data was pre-processed. And, although the other methods fail to give that good results on this dataset, they are still used for other tasks related to image processing (sharpening, smoothing etc.).