# aayush-sharma-21bcon625-2

December 13, 2023

## 1 Aayush Sharma 21BCON625

## 2 DL Assignment 5

## 3 Section-G

[5]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import tensorflow as tf
import tensorflow.keras
from keras_tuner.tuners import RandomSearch
```

### 3.0.1 Write python code for necessary dependencies to import dataset.

[6]:
```
(xtrain,ytrain),(xtest,ytest)=tf.keras.datasets.mnist.load_data()
```

[7]:
```
xtrain.shape
```

[7]:
```
(60000, 28, 28)
```

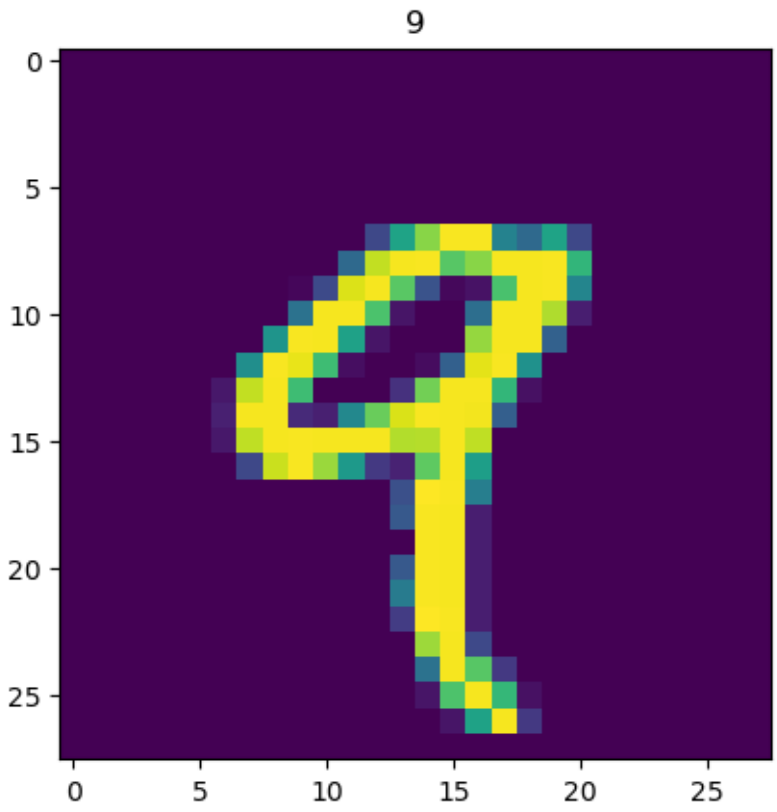
### 3.0.2 Write python code to check the datatype of all the columns present in the dataset.

[19]:
```
xtrain.dtype
```

[19]:
```
dtype('float64')
```

### 3.0.3 Write python code to visualize a sample of imagesfrom the dataset.

[9]:
```
plt.title(ytrain[4])
plt.imshow(xtrain[4])
plt.show()
```




### 3.0.4 Write python code to check the null value present in the in the dataset.

[10]:
```
pd.DataFrame(xtrain.reshape(xtrain.shape[0],28*28)).isna().sum().sum()
```

[10]: 0

[11]:
```
xtrain=xtrain/255
xtest=xtest/255
```

### 3.0.5 Define strides in CNN.

In machine learning, particularly in the context of convolutional neural networks (CNNs), the term "stride" refers to the number of pixels by which we move the filter across the input image.

### 3.0.6 Explain the concept of convolutional neural networks. How does it differ from ANN.

Convolutional Neural Networks (CNNs) are a type of deep learning model designed specifically for processing structured grid data, such as images. They are particularly effective in tasks related to computer vision, image recognition, and other tasks involving spatial relationships. CNNs are

an evolution of traditional Artificial Neural Networks (ANNs), and they differ in their architecture and the way they handle input data.

Here are the key concepts of Convolutional Neural Networks:

1. **Convolutional Layers:**
   - CNNs use convolutional layers as their basic building blocks. A convolutional layer applies convolutional operations to the input data. These operations involve sliding small filters (also known as kernels) over the input data to detect patterns, edges, and features.
2. **Pooling Layers:**
   - After convolutional operations, pooling layers are often used to down-sample the spatial dimensions of the input data. Max pooling, for example, retains the maximum value within each region of the input, effectively reducing the size of the feature maps.
3. **Local Connectivity:**
   - CNNs exploit the concept of local connectivity, meaning that neurons are connected to only a small region of the input volume. This allows the network to focus on local patterns and makes the model more efficient in processing spatial hierarchies.
4. **Weight Sharing:**
   - CNNs use weight sharing to reduce the number of parameters in the network. In convolutional layers, the same set of weights is applied to different parts of the input data. This sharing of weights enables the network to learn spatial hierarchies of features.
5. **Hierarchical Feature Learning:**
   - CNNs are designed to automatically learn hierarchical features from the input data. Lower layers tend to learn simple features like edges and textures, while higher layers combine these features to learn more complex patterns and object representations.

Now, let's compare CNNs with traditional ANNs:

1. **Spatial Hierarchies:**
   - ANNs treat input data as flat vectors, ignoring the spatial relationships in structured grid data like images. CNNs, on the other hand, preserve the spatial structure through convolutional and pooling layers, enabling them to capture local patterns and hierarchies.
2. **Parameter Sharing:**
   - CNNs use parameter sharing through convolutional operations, reducing the number of parameters and enhancing the ability to generalize to new data. ANNs typically have more parameters and may be more prone to overfitting.
3. **Translation Invariance:**
   - CNNs are inherently translation invariant due to weight sharing in convolutional layers. This means that the network can recognize patterns regardless of their position in the input space. ANNs may struggle with translation invariance, as they treat each input feature independently.
4. **Application to Grid Data:**
   - CNNs are specialized for grid-like data, such as images, where spatial relationships matter. ANNs are more general-purpose and may be used for various types of data, but they might not perform as well on tasks involving structured grid data.

In summary, CNNs are a specialized type of neural network architecture designed for processing grid-like data, with a focus on preserving spatial relationships and learning hierarchical features. Their architecture makes them particularly effective for computer vision tasks, where the spatial

arrangement of pixels is crucial for understanding the content of images.

### 3.0.7 What do you mean by pooling and flatten layers?

Pooling and Flatten layers are commonly used components in Convolutional Neural Networks (CNNs), which are a type of deep neural network architecture often employed in image recognition tasks.

**Pooling Layer**:

Pooling layers are used to downsample the spatial dimensions of the input volume, reducing the number of parameters and computational complexity in the network. Max pooling and average pooling are two common types of pooling operations. Max pooling retains the maximum value from a group of neighboring pixels, while average pooling takes the average. Pooling helps in capturing the most important features and making the representation more invariant to small translations and distortions.

**Flatten Layer**:

Flatten layers are used to convert the multi-dimensional output of the previous layer into a one-dimensional vector. It takes the output of the last convolutional or pooling layer and flattens it into a linear array. This flattened array serves as the input to the fully connected layers (dense layers) in the subsequent part of the neural network.

### 3.0.8 How do you use the Keras tuner to optimize the parameters in MNIST dataset?

Keras Tuner is a powerful tool that helps automate the hyperparameter tuning process for your deep learning models. Steps-

->Setup:Install Keras Tuner

->Import libraries

->Define the Model:

```
Define a simple deep learning model with Keras.

Wrap the model definition inside a function that takes hp (hyperparameters) as an argument.

Use Keras Tuner functions like hp.Int or hp.Choice to specify the range of hyperparameters to s
```

->Instantiate the Tuner and Perform Hyperparameter Tuning:

```
Choose a hyperparameter tuning algorithm.For example, use RandomSearch to randomly sample from

Pass the model building function and search space to the tuner.Start the hyperparameter tuning
```

->Train the Model: Extract the best model configuration from the tuner.Build and train the model with the best hyperparameters.

->Evaluate and Analyze.

**(Python code below along with model)**

[25]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from keras_tuner.tuners import RandomSearch

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

x_train = x_train.reshape((x_train.shape[0], -1)) / 255.0
x_test = x_test.reshape((x_test.shape[0], -1)) / 255.0

y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

x_train = x_train.reshape((x_train.shape[0], -1)) / 255.0
x_test = x_test.reshape((x_test.shape[0], -1)) / 255.0

y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

def build_model(hp):
    model = tf.keras.Sequential()

    model.add(layers.InputLayer(input_shape=(784,)))
    model.add(layers.Dense(units=hp.Int('units', min_value=32, max_value=512,␣
 ↪step=32), activation='relu'))
    model.add(layers.Dense(10, activation='softmax'))

    model.compile(optimizer=keras.optimizers.Adam(hp.Choice('learning_rate',␣
 ↪values=[1e-2, 1e-3, 1e-4])),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    return model

tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,
    directory='tuner_results',
    project_name='mnist_tuning'
)

tuner.search(x_train, y_train, epochs=5, validation_data=(x_test, y_test))
best_model = tuner.get_best_models(num_models=1)[0]
best_model.summary()
```

```
best_model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
```

```
Trial 5 Complete [00h 00m 54s]
val_accuracy: 0.963100016117096

Best val_accuracy So Far: 0.9634000062942505
Total elapsed time: 00h 07m 25s
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               200960

 dense_1 (Dense)             (None, 10)                2570

=================================================================
Total params: 203530 (795.04 KB)
Trainable params: 203530 (795.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 16s 8ms/step - loss: 0.1137 -
accuracy: 0.9723 - val_loss: 0.2154 - val_accuracy: 0.9629
Epoch 2/10
1875/1875 [==============================] - 15s 8ms/step - loss: 0.1135 -
accuracy: 0.9728 - val_loss: 0.2015 - val_accuracy: 0.9631
Epoch 3/10
1875/1875 [==============================] - 16s 8ms/step - loss: 0.1064 -
accuracy: 0.9748 - val_loss: 0.2325 - val_accuracy: 0.9585
Epoch 4/10
1875/1875 [==============================] - 17s 9ms/step - loss: 0.0996 -
accuracy: 0.9766 - val_loss: 0.2161 - val_accuracy: 0.9651
Epoch 5/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.1036 -
accuracy: 0.9769 - val_loss: 0.2787 - val_accuracy: 0.9615
Epoch 6/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.1024 -
accuracy: 0.9776 - val_loss: 0.2666 - val_accuracy: 0.9667
Epoch 7/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.0926 -
accuracy: 0.9800 - val_loss: 0.3166 - val_accuracy: 0.9617
Epoch 8/10
1875/1875 [==============================] - 16s 8ms/step - loss: 0.0893 -
accuracy: 0.9803 - val_loss: 0.3142 - val_accuracy: 0.9678
Epoch 9/10
1875/1875 [==============================] - 14s 8ms/step - loss: 0.0897 -
accuracy: 0.9815 - val_loss: 0.3308 - val_accuracy: 0.9628
```

```
Epoch 10/10
1875/1875 [==============================] - 15s 8ms/step - loss: 0.0951 -
accuracy: 0.9817 - val_loss: 0.3292 - val_accuracy: 0.9630
```

[25]: <keras.src.callbacks.History at 0x191686684d0>

#### 3.0.9 Explain the process of splitting a dataset into training and test sets. Elaborate on the role of the random state parameter in this operation.

Splitting a dataset into training and test sets is a common practice in machine learning to assess the performance of a model on unseen data. The process involves dividing the dataset into two disjoint subsets: one for training the model and the other for evaluating its performance. The training set is used to train the model, while the test set is used to assess how well the model generalizes to new, unseen data.

Here's a general overview of the process:

**Dataset Preparation**:

Start with a labeled dataset containing input features and corresponding target labels. Ensure that the dataset is representative of the overall distribution of data you expect the model to encounter. Random Splitting:

Randomly shuffle the dataset to ensure that the data points are not ordered in a way that might bias the training or test set. Split the shuffled dataset into two parts: the training set and the test set. Training Set:

Typically, a larger portion of the data (e.g., 70-80%) is allocated to the training set. The model learns patterns and relationships from this training data. Test Set:

The remaining portion of the data (e.g., 20-30%) is used as the test set. The model's performance is evaluated on this set to gauge how well it generalizes to new, unseen examples. The random_state parameter is used to ensure reproducibility when splitting the dataset. When you set a specific random seed (integer) using random_state, the random splitting process becomes deterministic. This means that if you use the same random seed, you will get the same train-test split every time you run the code.

#### 3.0.10 Discuss the significance of learning rate and optimizers in a deep learning model. Write python code to set these parameters in a CNN model for the MNIST dataset.

Learning rate and optimizers play crucial roles in training deep learning models, including Convolutional Neural Networks (CNNs). Let's discuss their significance:

Learning Rate:

The learning rate is a hyperparameter that determines the size of the step taken during the optimization process. If the learning rate is too high, the model might converge too quickly, potentially overshooting the optimal solution. If the learning rate is too low, the model may take a very long time to converge or may get stuck in a local minimum. Optimizers:

Optimizers are algorithms that adjust the weights of the neural network during training to minimize the loss function. Different optimizers use various strategies to update the weights, such as

Stochastic Gradient Descent (SGD), Adam, RMSprop, etc. The choice of optimizer can significantly impact the training speed and the quality of the learned model.

[20]:
```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam

# Load and preprocess the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.
 ↪mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255

train_labels = tf.keras.utils.to_categorical(train_labels)
test_labels = tf.keras.utils.to_categorical(test_labels)

# Define the CNN model
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

# Set the learning rate and optimizer
learning_rate = 0.001
optimizer = Adam(learning_rate=learning_rate)

# Compile the model
model.compile(optimizer=optimizer,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(train_images, train_labels, epochs=5, batch_size=64,␣
 ↪validation_data=(test_images, test_labels))
```

```
Epoch 1/5
938/938 [==============================] - 97s 96ms/step - loss: 0.1924 -
accuracy: 0.9408 - val_loss: 0.0507 - val_accuracy: 0.9849
Epoch 2/5
938/938 [==============================] - 81s 87ms/step - loss: 0.0527 -
accuracy: 0.9839 - val_loss: 0.0480 - val_accuracy: 0.9847
Epoch 3/5
938/938 [==============================] - 72s 76ms/step - loss: 0.0380 -
```

```
accuracy: 0.9881 - val_loss: 0.0394 - val_accuracy: 0.9869
Epoch 4/5
938/938 [==============================] - 80s 85ms/step - loss: 0.0301 -
accuracy: 0.9905 - val_loss: 0.0255 - val_accuracy: 0.9911
Epoch 5/5
938/938 [==============================] - 82s 87ms/step - loss: 0.0253 -
accuracy: 0.9922 - val_loss: 0.0315 - val_accuracy: 0.9890
```

[20]: `<keras.src.callbacks.History at 0x19132b02290>`

### 3.0.11 Define deep learning and its applications. Build a Convolutional Neural Network (CNN) for classifying digits in the MNIST dataset.

**Deep Learning Definition**:Deep learning is a subset of artificial intelligence (AI) that uses artificial neural networks (ANNs) with multiple hidden layers to learn from data. These ANNs learn complex representations by processing data through multiple layers, enabling them to solve tasks that are difficult or impossible for traditional algorithms.

Applications:Computer Vision,Natural Language Processing (NLP),Speech Recognition,Fraud Detection,etc.

[12]:
```
def model_builder(hp):
    c_optimizer=tf.keras.optimizers.Adam(learning_rate=0.01)
    model=tf.keras.models.Sequential()
    hp_units2=hp.Int('unit2',min_value=10,max_value=100,step=10)
    model.add(tf.keras.layers.
 ↪Conv2D(30,(3,3),activation="relu",input_shape=(28,28,1)))
    model.add(tf.keras.layers.MaxPooling2D((2,2)))
    model.add(tf.keras.layers.Flatten())
    hp_units1=hp.Int('unit1',min_value=100,max_value=500,step=50)
    model.add(tf.keras.layers.Dense(hp_units1,activation="relu"))
    model.add(tf.keras.layers.Dense(64,activation="relu"))
    model.add(tf.keras.layers.Dense(10,activation="softmax"))
    model.
 ↪compile(optimizer=c_optimizer,loss="sparse_categorical_crossentropy",metrics=["accuracy"])
    return model
```

[13]:
```
tuner=RandomSearch(model_builder,objective='val_accuracy',max_trials=10)
tuner.search(xtrain,ytrain,validation_split=0.2)
```

```
Trial 10 Complete [00h 01m 32s]
val_accuracy: 0.9725833535194397

Best val_accuracy So Far: 0.9770833253860474
Total elapsed time: 00h 14m 44s
```

[16]:
```
best_model=tuner.get_best_models(num_models=1)[0]
best_model.fit(xtrain,ytrain,epochs=10,batch_size=500)
```

```
Epoch 1/10
120/120 [==============================] - 34s 267ms/step - loss: 0.0494 -
accuracy: 0.9865
Epoch 2/10
120/120 [==============================] - 32s 270ms/step - loss: 0.0283 -
accuracy: 0.9918
Epoch 3/10
120/120 [==============================] - 34s 283ms/step - loss: 0.0192 -
accuracy: 0.9944
Epoch 4/10
120/120 [==============================] - 34s 281ms/step - loss: 0.0128 -
accuracy: 0.9966
Epoch 5/10
120/120 [==============================] - 31s 257ms/step - loss: 0.0087 -
accuracy: 0.9978
Epoch 6/10
120/120 [==============================] - 35s 295ms/step - loss: 0.0058 -
accuracy: 0.9988
Epoch 7/10
120/120 [==============================] - 38s 314ms/step - loss: 0.0041 -
accuracy: 0.9992
Epoch 8/10
120/120 [==============================] - 33s 277ms/step - loss: 0.0027 -
accuracy: 0.9994
Epoch 9/10
120/120 [==============================] - 34s 281ms/step - loss: 0.0019 -
accuracy: 0.9997
Epoch 10/10
120/120 [==============================] - 35s 290ms/step - loss: 0.0014 -
accuracy: 0.9997
```

[16]:
```
<keras.src.callbacks.History at 0x1913246e550>
```

[17]:
```
best_model.evaluate(xtest,ytest)
```

```
313/313 [==============================] - 4s 11ms/step - loss: 0.0711 -
accuracy: 0.9863
```

[17]:
```
[0.07112282514572144, 0.986299991607666]
```

[ ]:
```

```