03_asl_cnn

August 4, 2024

1 Python Program to Implement a CNN with TensorFlow and Keras

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

# Assuming X_train, y_train, X_valid, y_valid are already defined

# Define the CNN model
model = models.Sequential()

# First Convolutional Layer
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 1)))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.25))

# Second Convolutional Layer
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.25))

# Flatten Layer
model.add(layers.Flatten())

# Dense Layer
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dropout(0.5))

# Output Dense Layer
model.add(layers.Dense(10, activation='softmax'))

# Compile the model
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
```

```
    metrics=['accuracy']
)

# Fit the model
history = model.fit(
    X_train, y_train,
    epochs=10,
    batch_size=32,
    validation_data=(X_valid, y_valid)
)

# Print the model summary
print(model.summary())
```

2 Convolutional Neural Networks

In the previous lecture, we built and trained a simple model to classify ASL images. The model was able to learn how to correctly classify the training dataset with very high accuracy, but, it did not perform nearly as well on validation dataset. This behavior of not generalizing well to non-training data is called overfitting, and in this section, we will introduce a popular kind of model called a convolutional neural network that is especially good for reading images and classifying them.

2.1 Loading and Preparing the Data

The below cell contains the data preprocessing techniques we learned in the previous lectures. Execute it.

[ ]:
```
import tensorflow.keras as keras
import pandas as pd

train_df = pd.read_csv("sign_mnist_train.csv")
valid_df = pd.read_csv("sign_mnist_valid.csv")

# Separate out our target values
y_train = train_df['label']
y_valid = valid_df['label']
del train_df['label']
del valid_df['label']

# Separate out our image vectors
x_train = train_df.values
x_valid = valid_df.values

# Turn our scalar targets into binary categories
num_classes = 24
y_train = keras.utils.to_categorical(y_train, num_classes)
y_valid = keras.utils.to_categorical(y_valid, num_classes)
```

```
# Normalize our image data
x_train = x_train / 255
x_valid = x_valid / 255
```

2.2 Reshaping Images for a CNN

The individual pictures in our dataset are in the format of long lists of 784 pixels:

[ ]:
```
x_train.shape, x_valid.shape
```

[ ]:
```
((27455, 784), (7172, 784))
```

In this format, we don't have all the information about which pixels are near each other. Because of this, we can't apply convolutions that will detect features. Let's reshape our dataset so that they are in a 28x28 pixel format. This will allow our convolutions to associate groups of pixels and detect important features.

Note that for the first convolutional layer of our model, we need to have not only the height and width of the image, but also the number of color channels. Our images are grayscale, so we'll just have 1 channel.

That means that we need to convert the current shape `(27455, 784)` to `(27455, 28, 28, 1)`. As a convenience, we can pass the reshape method a `-1` for any dimension we wish to remain the same, therefore:

[ ]:
```
x_train = x_train.reshape(-1,28,28,1)
x_valid = x_valid.reshape(-1,28,28,1)
```

[ ]:
```
x_train.shape
```

[ ]:
```
(27455, 28, 28, 1)
```

[ ]:
```
x_valid.shape
```

[ ]:
```
(7172, 28, 28, 1)
```

[ ]:
```
x_train.shape, x_valid.shape
```

[ ]:
```
((27455, 28, 28, 1), (7172, 28, 28, 1))
```

2.3 Creating a Convolutional Model

These days, many data scientists start their projects by borrowing model properties from a similar project. Assuming the problem is not totally unique, there's a great chance that people have created models that will perform well which are posted in online repositories like TensorFlow Hub and the NGC Catalog.

[ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
```

```
    Dense,
    Conv2D,
    MaxPool2D,
    Flatten,
    Dropout,
    BatchNormalization,
)

model = Sequential()
model.add(Conv2D(75, (3, 3), strides=1, padding="same", activation="relu",
                 input_shape=(28, 28, 1)))
model.add(BatchNormalization())
model.add(MaxPool2D((2, 2), strides=2, padding="same"))
model.add(Conv2D(50, (3, 3), strides=1, padding="same", activation="relu"))
model.add(Dropout(0.2))
model.add(BatchNormalization())
model.add(MaxPool2D((2, 2), strides=2, padding="same"))
model.add(Conv2D(25, (3, 3), strides=1, padding="same", activation="relu"))
model.add(BatchNormalization())
model.add(MaxPool2D((2, 2), strides=2, padding="same"))
model.add(Flatten())
model.add(Dense(units=512, activation="relu"))
model.add(Dropout(0.3))
model.add(Dense(units=num_classes, activation="softmax"))
```

2.3.1 Conv2D

These are our 2D convolutional layers. Small kernels will go over the input image and detect features that are important for classification. Earlier convolutions in the model will detect simple features such as lines. Later convolutions will detect more complex features. Let's look at our first Conv2D layer:

```
model.add(Conv2D(75 , (3,3) , strides = 1 , padding = 'same'...)
```

75 refers to the number of filters that will be learned. (3,3) refers to the size of those filters. Strides refer to the step size that the filter will take as it passes over the image. Padding refers to whether the output image that's created from the filter will match the size of the input image.

2.3.2 BatchNormalization

Like normalizing our inputs, batch normalization scales the values in the hidden layers to improve training. Read more about it in detail here.

2.3.3 MaxPool2D

Max pooling takes an image and essentially shrinks it to a lower resolution. It does this to help the model be robust to translation (objects moving side to side), and also makes our model faster.

2.3.4 Dropout

Dropout is a technique for preventing overfitting. Dropout randomly selects a subset of neurons and turns them off, so that they do not participate in forward or backward propagation in that particular pass. This helps to make sure that the network is robust and redundant, and does not rely on any one area to come up with answers.

2.3.5 Flatten

Flatten takes the output of one layer which is multidimensional, and flattens it into a one-dimensional array. The output is called a feature vector and will be connected to the final classification layer.

2.3.6 Dense

We have seen dense layers before in our earlier models. Our first dense layer (512 units) takes the feature vector as input and learns which features will contribute to a particular classification. The second dense layer (24 units) is the final classification layer that outputs our prediction.

2.4 Summarizing the Model

This may feel like a lot of information, but don't worry. It's not critical that to understand everything right now in order to effectively train convolutional models. Most importantly we know that they can help with extracting useful information from images, and can be used in classification tasks.

Here, we summarize the model we just created. Notice how it has fewer trainable parameters than the model in the previous notebook:

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

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 75)        750
_________________________________________________________________
batch_normalization (BatchNo (None, 28, 28, 75)        300
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 75)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 50)        33800
_________________________________________________________________
dropout (Dropout)            (None, 14, 14, 50)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 14, 14, 50)        200
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 50)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 25)          11275
```

```
_________________________________________________________________
batch_normalization_2 (Batch (None, 7, 7, 25)          100
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 25)          0
_________________________________________________________________
flatten (Flatten)            (None, 400)               0
_________________________________________________________________
dense (Dense)                (None, 512)               205312
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 24)                12312
=================================================================
Total params: 264,049
Trainable params: 263,749
Non-trainable params: 300
_________________________________________________________________
```

2.5 Compiling the Model

We'll compile the model just like before:

[ ]:
```
model.compile(loss="categorical_crossentropy", metrics=["accuracy"])
```

2.6 Training the Model

Despite the very different model architecture, the training looks exactly the same. Run the cell below to train for 20 epochs and let's see if the accuracy improves:

[ ]:
```
model.fit(x_train, y_train, epochs=10, verbose=1, validation_data=(x_valid,␣
↪y_valid))
```

```
Epoch 1/10
858/858 [==============================] - 5s 5ms/step - loss: 0.3072 -
accuracy: 0.9066 - val_loss: 0.2882 - val_accuracy: 0.9127
Epoch 2/10
858/858 [==============================] - 4s 5ms/step - loss: 0.0195 -
accuracy: 0.9937 - val_loss: 0.3995 - val_accuracy: 0.9095
Epoch 3/10
858/858 [==============================] - 4s 5ms/step - loss: 0.0102 -
accuracy: 0.9965 - val_loss: 0.3287 - val_accuracy: 0.9223
Epoch 4/10
858/858 [==============================] - 4s 5ms/step - loss: 0.0077 -
accuracy: 0.9974 - val_loss: 0.2268 - val_accuracy: 0.9516
Epoch 5/10
858/858 [==============================] - 4s 5ms/step - loss: 0.0076 -
accuracy: 0.9982 - val_loss: 0.3375 - val_accuracy: 0.9165
Epoch 6/10
858/858 [==============================] - 4s 5ms/step - loss: 0.0069 -
```

```
accuracy: 0.9983 - val_loss: 0.2774 - val_accuracy: 0.9495
Epoch 7/10
858/858 [==============================] - 4s 5ms/step - loss: 0.0028 -
accuracy: 0.9992 - val_loss: 0.3765 - val_accuracy: 0.9413
Epoch 8/10
858/858 [==============================] - 4s 5ms/step - loss: 0.0037 -
accuracy: 0.9993 - val_loss: 0.1556 - val_accuracy: 0.9787
Epoch 9/10
858/858 [==============================] - 4s 5ms/step - loss: 0.0029 -
accuracy: 0.9993 - val_loss: 0.3367 - val_accuracy: 0.9561
Epoch 10/10
858/858 [==============================] - 4s 5ms/step - loss: 0.0025 -
accuracy: 0.9994 - val_loss: 0.2632 - val_accuracy: 0.9501
```

[ ]: `<tensorflow.python.keras.callbacks.History at 0x7fb3c326ab38>`

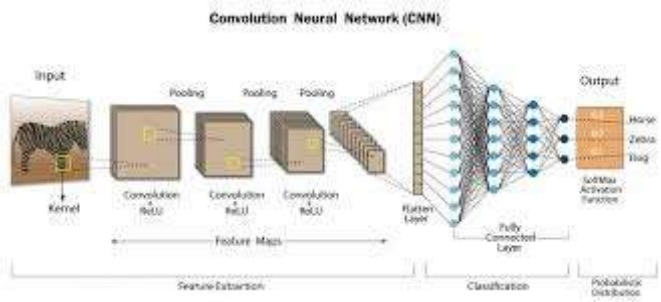
2.7 Discussion of Results

It looks like this model is significantly improved! The training accuracy is very high, and the validation accuracy has improved as well. This is a great result, as all we had to do was swap in a new model.

In [ ]:
```
1
```

cnn

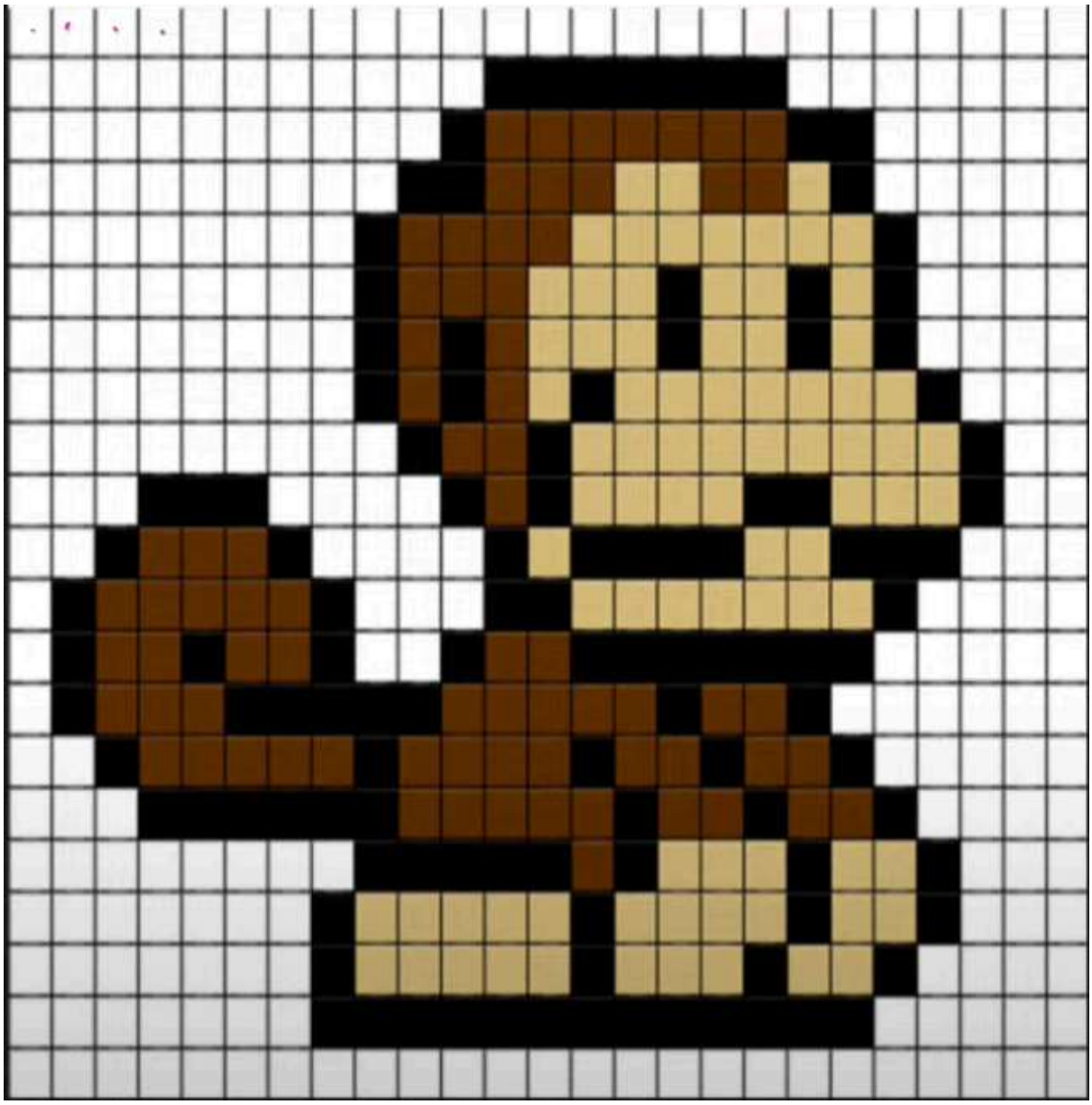
Convolution neural network(convenet) are a special kind of neural network for processing data that has a known grid-like topology like time series data (1D) or images (2d)



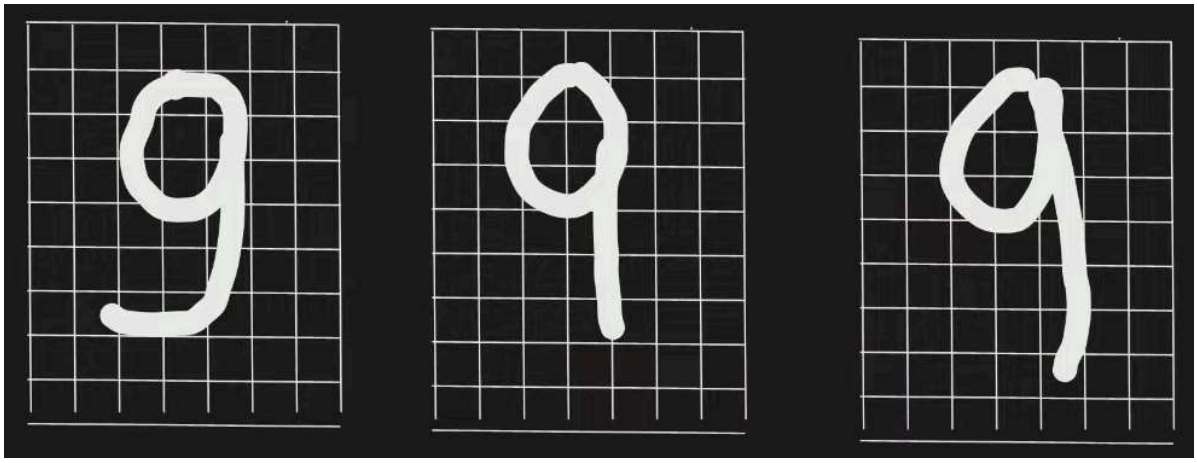
Feature	Artificial Neural Networks (ANNs)	Convolutional Neural Networks (CNNs)
Architecture	Fully connected layers	Convolutional layers followed by pooling layers, then fully connected layers
Connection Pattern	Each neuron connected to every neuron in subsequent layer	Local connections using filters/kernels
Handling of Input Data	Designed for 1D structured data (e.g., tabular data)	Designed for 2D/3D spatial data (e.g., images, videos)
Parameter Sharing	No weight sharing, each connection has its own weight	Weights are shared within the filters
Feature Extraction	Does not inherently extract spatial features	Extracts spatial hierarchies (edges, textures, objects
Dimensionality Reduction	Not inherently designed for dimensionality reduction	Uses pooling layers to reduce spatial dimensions
Computational Efficiency	Less efficient with high-dimensional data	More efficient due to weight sharing and reduced parameters
Training Complexity	Requires more parameters for high-dimensional data	Fewer parameters due to shared weights and pooling
Applications	Classification, regression, time series prediction	Image and video recognition, object detection, image segmentation
Examples of Use Cases	Predicting stock prices, customer churn prediction	Face recognition, medical image analysis, autonomous driving
Sensitivity to Input	Sensitive to the scale of input features	Less sensitive due to normalization layers (e.g., batch normalization)
Overfitting	Higher risk of overfitting with large datasets	Lower risk due to spatial hierarchies and pooling layers

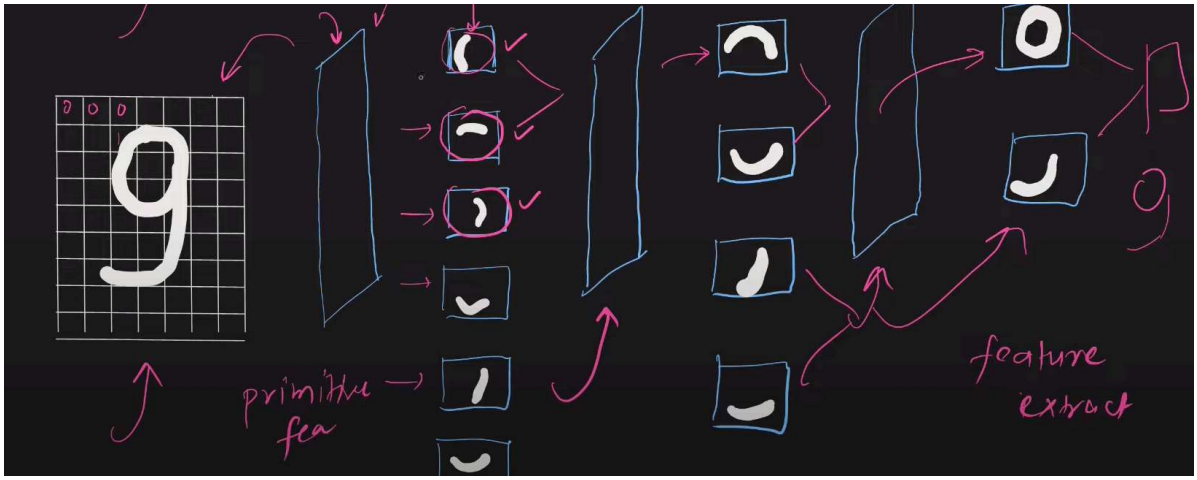
why not ANN?

1. high computation cost
2. overfitting
3. loss of imp info like spatial arrangement of pixels

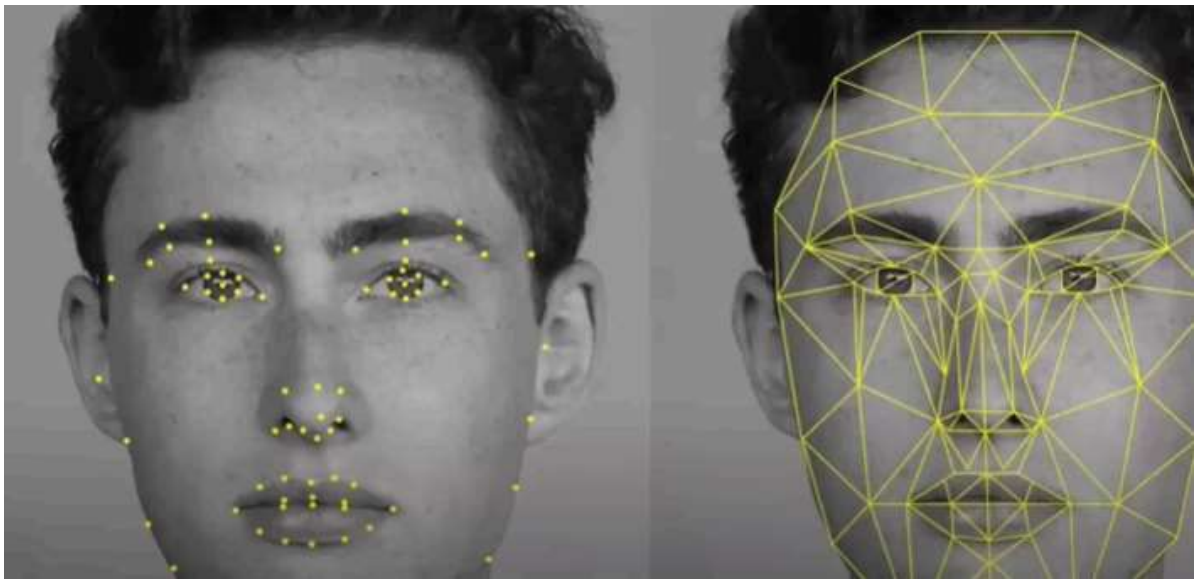


CNN INTUITION

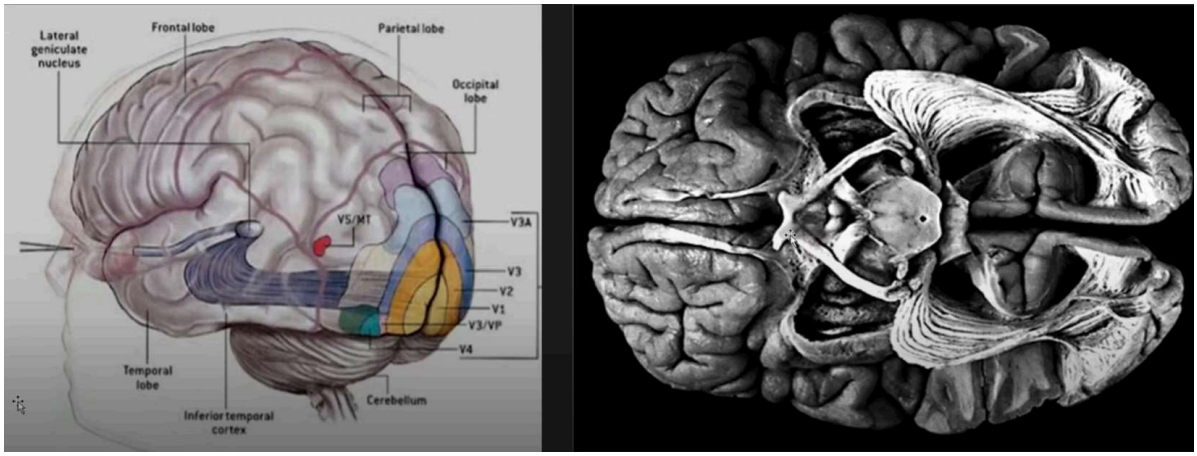




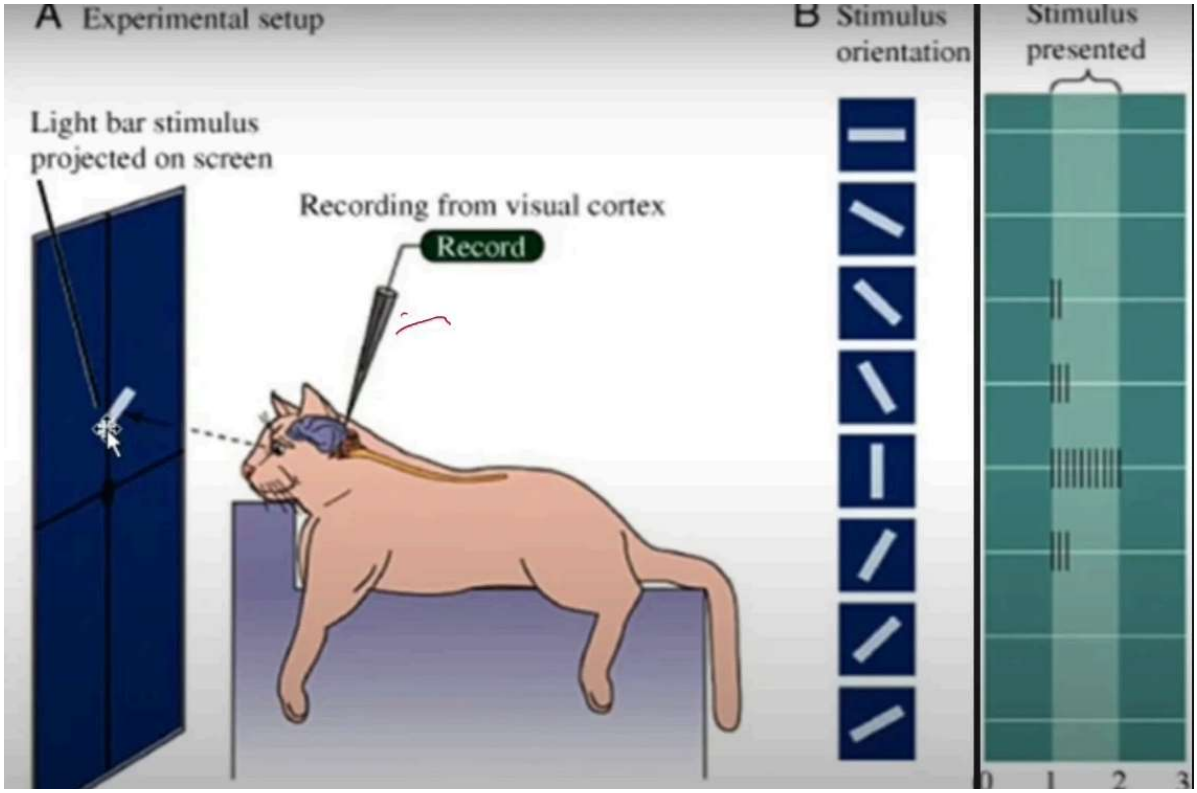
APPLICATION WISE



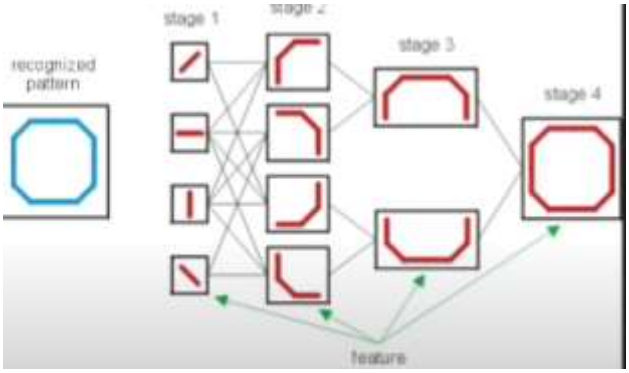
HUMAN VISUAL CORTEX



EXPRIMENT SETUP

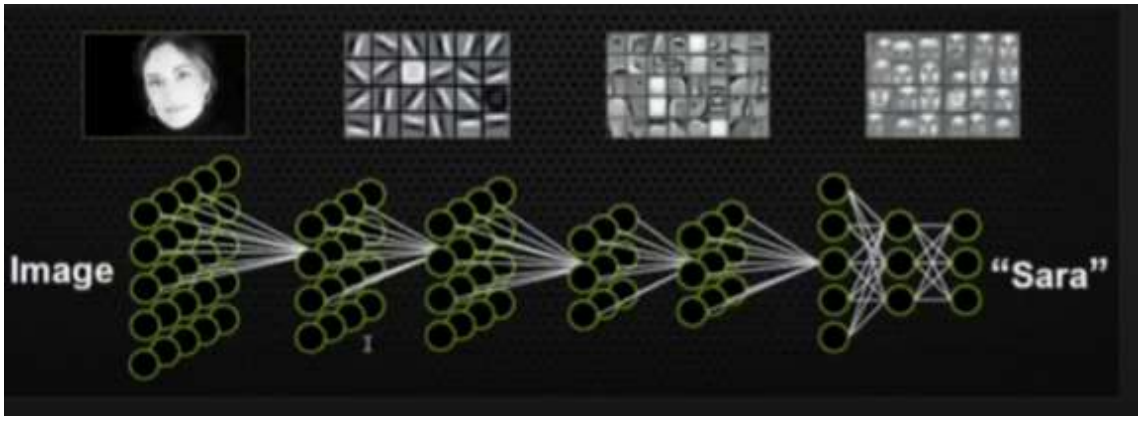


- EDGE DETECTION (PRIMARY FEATURE CELL)
- COMPLEX DETECTION (COMPLEX FEATURE CELL)



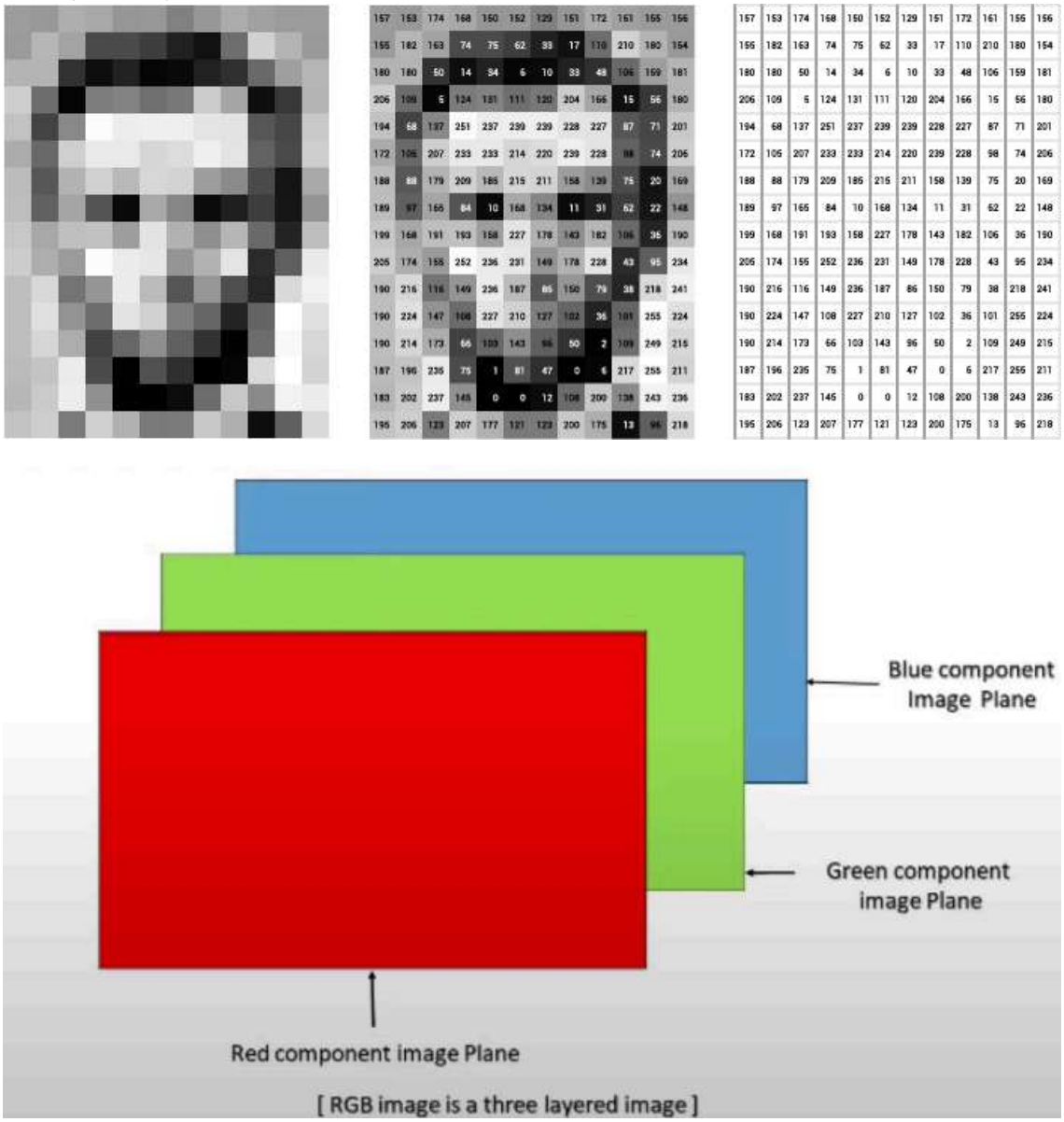
CONVOLUTION OPERATION

- CONV-POOLING-FC

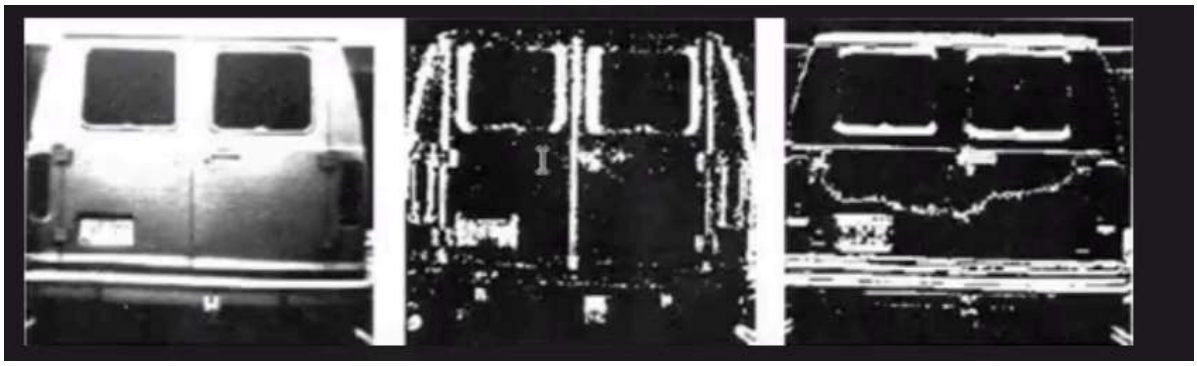


**BASIC OF IMAGES

- GRAYSCALE(B/W) (ONE)
- RGB (THREE)



EDGE DETECTION IN CONVOLUTION



https://deeplizard.com/resource/pavq7noze2 (https://deeplizard.com/resource/pavq7noze2)

```
# SHAPE AFTER CONVOLUTION
######  IMAGE * FILTER =  FEATURE MAP
###### (N*N) *(M*M) =(N-M+1)*(N-M+1)
```

```
- (28*28) * (3*3) =(26*26)
```

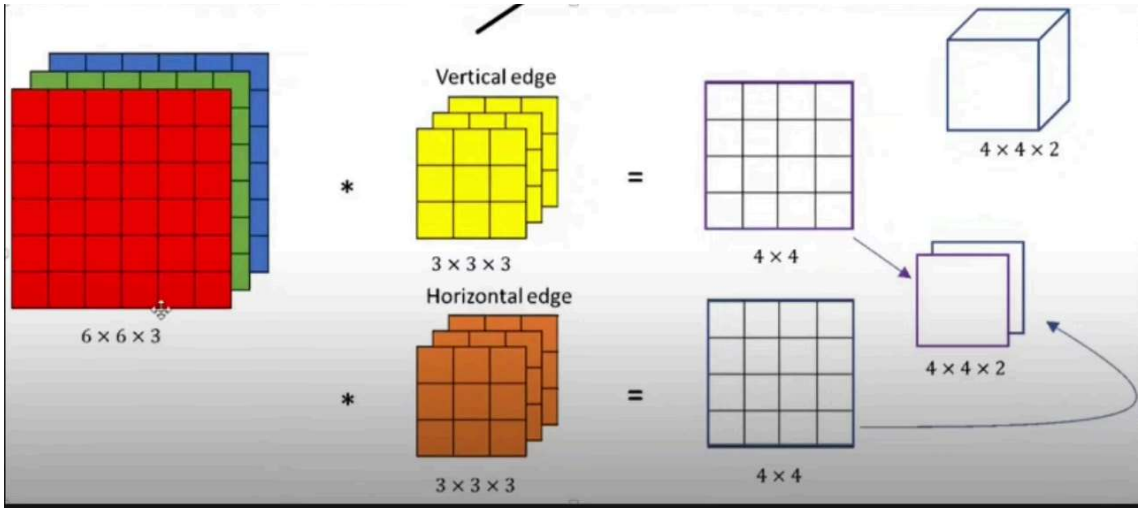
RGB IMAGE * FILTER = FEATURE MAP (N*N*C)* (M*M*C) =(N-M+1)*(N-M+1) (SINGLE CHANNEL (2828) * (33) =(2626)*

RGB

IMAGE * FILTER = FEATURE MAP

(NNC) (MMC) =(N-M+1)(N-M+1) (SINGLE CHANNEL

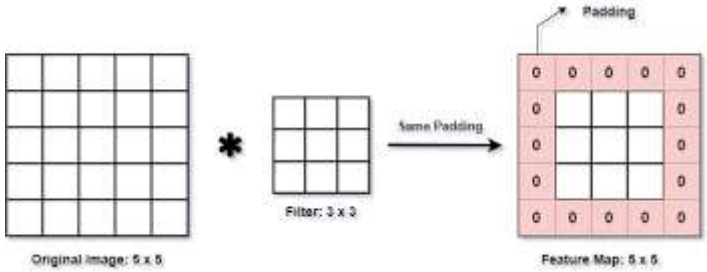
- (2828) * (33) =(26*26)




```
# IN OUTPUT LAYER 4*4*2 MEANS 2 FILTER APPLY
```

PADDING

- padding is a technique used to preserve the spatial dimensions of the input image after convolution operations on a feature map. Padding involves adding extra pixels around the border of the input feature map before convolution.
- For a grayscale (n x n) image and (f x f) filter/kernel, the dimensions of the image resulting from a convolution operation is (n – f + 1) x (n – f + 1).



- p = number of layers of zeros added to the border of the image,
- then (n x n) image —> (n + 2p) x (n + 2p) image after padding.
- (n + 2p) x (n + 2p) * (f x f) ——> outputs (n + 2p – f + 1) x (n + 2p – f + 1) images

In [27]:
```
import tensorflow
from tensorflow import keras
from keras.layers import Dense,Conv2D,Flatten
from keras import Sequential
from keras.datasets import mnist
```

In [28]:
```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

In [29]:
```
model = Sequential()

model.add(Conv2D(32,kernel_size=(3,3),padding='valid', activation='relu',
model.add(Conv2D(32,kernel_size=(3,3),padding='valid', activation='relu'))
model.add(Conv2D(32,kernel_size=(3,3),padding='valid', activation='relu'))

model.add(Flatten())

model.add(Dense(128,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

In [30]:
```
model.summary()
```

Model: "sequential_8"

Layer (type)	Output Shape	
conv2d_22 (Conv2D)	(None, 26, 26, 32)	
conv2d_23 (Conv2D)	(None, 24, 24, 32)	
conv2d_24 (Conv2D)	(None, 22, 22, 32)	
flatten_8 (Flatten)	(None, 15488)	
dense_16 (Dense)	(None, 128)	
dense_17 (Dense)	(None, 10)	

Total params: 2,002,698 (7.64 MB)

Trainable params: 2,002,698 (7.64 MB)

Non-trainable params: 0 (0.00 B)

In [33]:
```
model = Sequential()

model.add(Conv2D(32,kernel_size=(3,3),padding='same', activation='relu', i
model.add(Conv2D(32,kernel_size=(3,3),padding='same', activation='relu'))
model.add(Conv2D(32,kernel_size=(3,3),padding='same', activation='relu'))

model.add(Flatten())

model.add(Dense(128,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

```
C:\Users\VISHAL\anaconda3\Lib\site-packages\keras\src\layers\convolutional\ba
se_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argumen
t to a layer. When using Sequential models, prefer using an `Input(shape)` ob
ject as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [34]:
```
model.summary()
```

Model: "sequential_10"

Layer (type)	Output Shape	
conv2d_28 (Conv2D)	(None, 28, 28, 32)	
conv2d_29 (Conv2D)	(None, 28, 28, 32)	
conv2d_30 (Conv2D)	(None, 28, 28, 32)	
flatten_10 (Flatten)	(None, 25088)	
dense_20 (Dense)	(None, 128)	
dense_21 (Dense)	(None, 10)	

Total params: 3,231,498 (12.33 MB)

Trainable params: 3,231,498 (12.33 MB)

Non-trainable params: 0 (0.00 B)

In [31]:
```
model = Sequential()

model.add(Conv2D(32,kernel_size=(3,3),padding='same',strides=(2,2), activa
model.add(Conv2D(32,kernel_size=(3,3),padding='same',strides=(2,2), activa
model.add(Conv2D(32,kernel_size=(3,3),padding='same',strides=(2,2), activa

model.add(Flatten())

model.add(Dense(128,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

In [32]:
```
model.summary()
```

Model: "sequential_9"

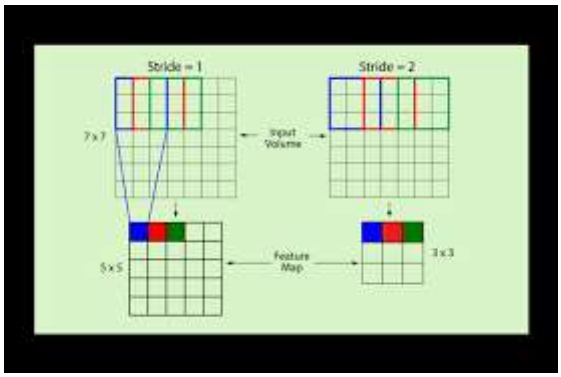
Layer (type)	Output Shape	
conv2d_25 (Conv2D)	(None, 14, 14, 32)	
conv2d_26 (Conv2D)	(None, 7, 7, 32)	
conv2d_27 (Conv2D)	(None, 4, 4, 32)	
flatten_9 (Flatten)	(None, 512)	
dense_18 (Dense)	(None, 128)	
dense_19 (Dense)	(None, 10)	

Total params: 85,770 (335.04 KB)

Trainable params: 85,770 (335.04 KB)

Non-trainable params: 0 (0.00 B)

sTRIDE: Stride determines how many squares or pixels our filters skip when they move across the image, from left to right and from top to bottom.




```
The input dimensions of the image – > I (ixi)
The size of filter/kernel – > F (fxf)
Strides – > S (integer)
Padding – > P (integer)
Depth/Number of feature maps/activation maps – > D (integer)
Convolution Output dimension = [(I - F +2 *P) / S] +1 x D > Formula1
```

In [21]:

```
model = Sequential()

model.add(Conv2D(32,kernel_size=(3,3),padding='same',strides=(2,2), activa
model.add(Conv2D(32,kernel_size=(3,3),padding='same',strides=(2,2), activa
model.add(Conv2D(32,kernel_size=(3,3),padding='same',strides=(2,2), activa

model.add(Flatten())

model.add(Dense(128,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

In [22]:
```
1 model.summary()
```

Model: "sequential_6"

Layer (type)	Output Shape	
conv2d_17 (Conv2D)	(None, 14, 14, 32)	
conv2d_18 (Conv2D)	(None, 7, 7, 32)	
conv2d_19 (Conv2D)	(None, 4, 4, 32)	
flatten_6 (Flatten)	(None, 512)	
dense_12 (Dense)	(None, 128)	
dense_13 (Dense)	(None, 10)	

Total params: 85,770 (335.04 KB)

Trainable params: 85,770 (335.04 KB)

Non-trainable params: 0 (0.00 B)

https://deeplizard.com/resource/pavq7noze3 (https://deeplizard.com/resource/pavq7noze3)

In [23]:
```
1 import tensorflow
2 from tensorflow import keras
3 from keras.layers import Dense,Conv2D,Flatten,MaxPooling2D
4 from keras import Sequential
5 from keras.datasets import mnist
```

In [24]:
```
1 (X_train, y_train), (X_test, y_test) = mnist.load_data()
```

In [25]:
```
model = Sequential()

model.add(Conv2D(32,kernel_size=(3,3),padding='valid', activation='relu',
model.add(MaxPooling2D(pool_size=(2, 2), strides=2, padding='valid'))
model.add(Conv2D(32,kernel_size=(3,3),padding='valid', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=2, padding='valid'))

model.add(Flatten())

model.add(Dense(128,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

In [26]:
```
model.summary()
```

Model: "sequential_7"

Layer (type)	Output Shape	
conv2d_20 (Conv2D)	(None, 26, 26, 32)	
max_pooling2d_2 (MaxPooling2D)	(None, 13, 13, 32)	
conv2d_21 (Conv2D)	(None, 11, 11, 32)	
max_pooling2d_3 (MaxPooling2D)	(None, 5, 5, 32)	
flatten_7 (Flatten)	(None, 800)	
dense_14 (Dense)	(None, 128)	
dense_15 (Dense)	(None, 10)	

Total params: 113,386 (442.91 KB)

Trainable params: 113,386 (442.91 KB)

Non-trainable params: 0 (0.00 B)