A CNN for MNIST Handwritten Digit Classification

MNIST Handwritten Digit Classification Dataset: The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset. It is a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9. (60,000 training examples and 10,000 testing examples)

TASK:

Classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9, inclusively.

TOOLS:

keras: TensorFlow's high-level API. we will be using tensorflow 2.0

Pre-processing

1- Initialization:

We know that each grayscale image has dimensions 28x28, there are 784 pixels per image. Therefore, each input image corresponds to a tensor of 784 normalized floating point values between 0.0 and 1.0. We also know that there are 10 classes and that classes are represented as unique integers. In terms of our code, we have img_rows = 28, img_cols = 28 and num_classes = 10.

Another important element to set up is the random seed as we want to keep the start point when a computer generates a random number sequence.

---> We start by importing the MNIST dataset.

In [14]:
```
import tensorflow as tf # tensorflow 2.0
from tensorflow.keras.datasets import mnist
import numpy as np
import keras
seed=0
np.random.seed(seed) # fix random seed
tf.random.set_seed(seed)
# input image dimensions
num_classes = 10 # 10 digits so we have 10 classes
img_rows, img_cols = 28, 28 # number of pixels
# the data is shuffled and split between train and test sets
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
```

2- Reshaping and rescaling:

As mentioned , our inputs have shapes (number_examples, img_rows, img_cols).But in order to use the data with our convolutional neural network, we need to get it into NHWC format. NHWC format has a shape with four dimensions:

```
-Number of image data samples (batch size)
-Height of each image
-Width of each image
-Channels per image
```

The height and width of each image from the dataset are (img_rows, img_cols), Number of channels= 1 (since the images are grayscale).

A good starting point is to normalize the pixel values of grayscale images, e.g. rescale them to the range [0,1] where 0.0 corresponds to a grayscale pixel value of 255 (pure white), while 1.0 corresponds to a grayscale pixel value of 0 (pure black). This involves first converting the data type from unsigned integers to floats, then dividing the pixel values by the maximum value.

In [15]:
```
X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)
# cast floats to single precision
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# rescale data in interval [0,1]
X_train /= 255
X_test /= 255
```

3- Casting label vectors Y

We need to transform our classes into vectors. We do this by tapping the following line:

In [16]:
```
Y_train = keras.utils.to_categorical(Y_train, num_classes)
Y_test = keras.utils.to_categorical(Y_test, num_classes)
```

Now that we have process our data, we can start to build out model.

Convolution Neural Network model

1-Filters:

We use filters to transform inputs and extract features that allow our model to recognize certain images. We train our neural network (via the kernel matrix weights) to produce filters that are able to extract the most useful hidden features.

2-Convolution:

The convolution represents how we apply our filter weights to the input data. The main operation used by a convolution is the matrix dot product, i.e. a summation over the element-wise product of two matrices. The number of matrix dot products in a convolution depends on the dimensions of the input data and kernel matrix, as well as the stride size. The stride size is the vertical/horizontal offset of the kernel matrix as it moves along the input data.

3-Padding:

Sometimes, when we do the dot product operation we don't use a row or a column. To avoid this phenomenon we can use padding. We can pad the input data matrix with 0's. This means we add rows/columns made entirely of 0's to the edges of the input data matrix.

4-Convolution layer:

A convolution layer in a CNN applies multiple filters to the input tensor. While each filter has a separate kernel matrix for each of the input channels, the overall result of a filter's convolution is the sum of the convolutions across all the input channels. Adding more filters to a convolution layer allows the layer to better extract hidden features. However, this comes at the cost of additional training time and computational complexity, since filters add extra weights to the model.

5-Pooling:

While the convolution layer extracts important hidden features, the number of features can still be pretty large. We can use pooling to reduce the size of the data in the height and width dimensions. This allows the model to perform fewer computations and ultimately train faster. The type of pooling that is usually used in CNNs is referred to as max pooling.

Multiple layers

1- Adding extra layers:

Like all neural networks, CNNs can benefit from additional layers beacause they allow a CNN to essentially stack multiple filters together for use on the image data. Risks: Similar to building any neural network, we need to be careful of how many additional layers we add. If we add too many layers to a model, we run the risk of having it overfit to the training data and therefore generalizing very poorly. Furthermore, each additional layer adds computational complexity and increases training time for our model.

2- Increase filters:

We usually increase the number of filters in a convolution layer the deeper it is in our model. In this case, our second convolution layer has 64 filters, compared to the 32 filters of the first convolution layer. The deeper the convolution layer, the more detailed the extracted features become. For example, the first convolution layer may have filters that extract features such as lines, edges, and curves. When we get to the second level, the filters of the convolution layer could now extract more distinguishing features, such as the sharp angle of a 77 or the intersecting curves of an 88.

Fully-connected layer

1-Fully-connected layer:

We apply a fully-connected layer of size 1024 (i.e. the number of neurons in the layer) to the output data of the second pooling layer. The number of units is somewhat arbitrary. Enough to be powerful, but not so much as to be too resource intensive. The purpose of the fully-connected layer is to aggregate the data features before we convert them to classes. This allows the model to make better predictions than if we had just converted the pooling output directly to classes.

2-Flattening:

The data we have been using in our model is of the NHWC format. However, in order to use a fully-connected layer, we need the data to be a matrix, where the number of rows represents the batch size and the columns represent the data features. This time we need to reshape in the opposite direction and converting from NHWC to a 2-D matrix.

Dropout

1- Co-adaptation

Co-adaptation refers to when multiple neurons in a layer extract the same, or very similar, hidden features from the input data. This can happen when the connection weights for two different neurons are nearly identical.

When a fully-connected layer has a large number of neurons, co-adaptation is more likely to occur. This can be a problem for two reasons. First, it is a waste of computation when we have redundant neurons computing the same output. Second, if many neurons are extracting the same features, it adds more significance to those features for our model. This leads to overfitting if the duplicate extracted features are specific to only the training set.

Solution:

2-Dropout

The way we minimize co-adaptation for fully-connected layers with many neurons is by applying dropout during training. In dropout, we randomly shut down some fraction of a layer's neurons at each training step by zeroing out the neuron values.

Soft-max Layer

Since there are 10 possible digits an MNIST image can be, we use a 10 neuron fully-connected layer to obtain the classes for each digit class. The Softmax function is applied to the classes to convert them into per class probabilities.

In [17]:

```
from tensorflow.keras.models import Sequential #The model type that we will be
from tensorflow.keras.layers import Dense, Conv2D, Flatten
from tensorflow.keras.layers import MaxPooling2D, Dropout
model = Sequential()#add model layers
model.add(Conv2D(32, kernel_size=(5, 5),
                 activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
# add second convolutional layer with 20 filters
model.add(Conv2D(64, (5, 5), activation='relu'))

# add 2D pooling layer
model.add(MaxPooling2D(pool_size=(2, 2)))

# flatten data
model.add(Flatten())

# add a dense all-to-all relu layer
model.add(Dense(1024, activation='relu'))

# apply dropout with rate 0.5
model.add(Dropout(0.5))

# soft-max layer
model.add(Dense(num_classes, activation='softmax'))#compile model using accura
```

Sequential:

The model type that we will be using is Sequential which is the easiest way to build a model in Keras. It allows you to build a model layer by layer.

add() :

we used the add() method to attach layers to our model. it suffices to focus on Dense layers for simplicity.

Dense(number of neurons, The type of activation function, name of the activation function)

Every Dense() layer accepts these arguments. Examples include relu, tanh, elu, sigmoid, softmax.

In this neural network, we have 2 convolution layers followed each time by a pooling layer. Then we flatten the data to add a dense layer on which we apply dropout with a rate of 0.5. Finally, we add a dense layer to allocate each image with the correct class.

Compiling the model

Compiling the model takes three parameters: optimizer, loss and metrics.

optimizer:

controls the learning rate. Adam is generally a good optimizer to use for many cases. The adam optimizer adjusts the learning rate throughout training.

loss:

We will use 'categorical_crossentropy' for our loss function. This is the most common choice for classification. A lower score indicates that the model is performing better.

metrics:

To make things even easier to interpret, we will use the 'accuracy' metric to see the accuracy score on the validation set when we train the model.

In [18]:
```
#compile model using accuracy to measure model performance
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['ac
```

Training the model

To train, we will use the 'fit()' function on our model with the following parameters: training data (X_train), target data (Y_train), validation data, and the number of epochs.

For our validation data, we will use the test set provided to us in our dataset, which we have split into X_test and Y_test.

The number of epochs is the number of times the model will cycle through the data. The more epochs we run, the more the model will improve, up to a certain point. After that point, the model will stop improving during each epoch. For our model, we will set the number of epochs to 3.

In [19]:
```
#train the model

model.fit(X_train, Y_train, validation_data=(X_test, Y_test), epochs=3)
```

```
Epoch 1/3
1875/1875 [==============================] - 156s 83ms/step - loss: 0.1192 - a
ccuracy: 0.9640 - val_loss: 0.0330 - val_accuracy: 0.9895
Epoch 2/3
1875/1875 [==============================] - 156s 83ms/step - loss: 0.0418 - a
ccuracy: 0.9869 - val_loss: 0.0258 - val_accuracy: 0.9918
Epoch 3/3
1875/1875 [==============================] - 154s 82ms/step - loss: 0.0314 - a
ccuracy: 0.9902 - val_loss: 0.0329 - val_accuracy: 0.9898
```

Out[19]:
```
<tensorflow.python.keras.callbacks.History at 0x24660a2b50>
```

Evaluate the model

Now we have trained our model we can evaluate its performance:

In [20]:
```
# evaluate the model
score = model.evaluate(X_test, Y_test, verbose=1)
# print performance
print()
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
313/313 [==============================] - 8s 25ms/step - loss: 0.0329 - accur
acy: 0.9898

Test loss: 0.032879382371902466
Test accuracy: 0.989799976348877
```

we have an accuracy of 99,3% and a lost of 0.025 on the test set which is very good. We can still improve the model by increasing the number of epoch and by introducing a batch size.

Make predictions

If you want to see the actual predictions that our model has made for the test data, we can use the predict_classes function. We can also to this by using the predict function will give an array with 10 numbers. These numbers are the probabilities that the input image represents each digit (0–9). The array index with the highest number represents the model prediction. The sum of each array equals 1 (since each number is a probability).

To show this, we will show the predictions for the first 4 images in the test set.

In [25]:
```
#predict first 4 images in the test set
model.predict_classes(X_test[:4])
```

Out[25]:
```
array([7, 2, 1, 0], dtype=int64)
```

In [26]:
```
#predict first 4 images in the test set
model.predict(X_test[:4])
```

Out[26]:
```
array([[3.01651704e-09, 1.57010387e-08, 1.78218613e-07, 4.86884176e-07,
        1.67755865e-09, 1.36745826e-09, 1.06966421e-12, 9.99994874e-01,
        1.21540147e-08, 4.43903900e-06],
       [2.11310095e-07, 8.54790798e-08, 9.99999762e-01, 2.30387667e-10,
        1.31116548e-10, 7.42021394e-14, 3.85762933e-09, 2.62286096e-11,
        1.73387160e-09, 1.20441103e-12],
       [1.22947119e-08, 9.99999523e-01, 1.28827892e-07, 1.95814795e-10,
        1.34194944e-09, 2.61334510e-09, 1.40856293e-09, 3.75415908e-07,
        3.44688793e-08, 3.57300056e-09],
       [9.99986649e-01, 9.23611054e-09, 3.25701990e-06, 1.03837845e-08,
        1.37231027e-08, 4.13643919e-09, 1.42833164e-06, 4.89105616e-07,
        2.73866448e-07, 8.00661928e-06]], dtype=float32)
```

We can see that our model predicted 7, 2, 1 and 0 for the first four images.

Let's compare this with the actual results.

In [28]:
```
#actual results for first 4 images in test set
Y_test[:4]
```

Out[28]:
```
array([[0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

The actual results show that the first four images are also 7, 2,1 and 0. Our model predicted correctly!

In [ ]: