Neural Networks and Image Recognition with Convolutional Neural Networks

1. Import and Process Data

   Import Pixels and Values as X and Y respectively.

```
In [5]: # Load the data
        X = pd.read_csv("pixels.csv")
        Y = pd.read_csv("values.csv")
```

```
In [7]: X_train = X.iloc[:,1:677]
        X_train
```

```
In [8]: Y_train = Y.iloc[:,1]
        Y_train
```

   Normalize data

   We perform a grayscale normalization to reduce the effect of illumination's differences.

```
In [12]: # Normalize the data
         X_train = X_train / 255.0
```

   Reshape all data to 28x28x1 3D matrices.

```
In [13]: # Reshape image in 3 dimensions (height = 26px, width = 26px , canal = 1)
         X_train = X_train.values.reshape(-1,26,26,1)
```

   Labels are 10 digits numbers from 0 to 9. e.g. 2 as [0,0,1,0,0,0,0,0,0,0]).

```
In [14]: # Encode labels to one hot vectors (ex : 2 -> [0,0,1,0,0,0,0,0,0,0])
         Y_train = to_categorical(Y_train, num_classes = 10)
```

   Split data set into training (90%) and testing sample (10%).

```
In [15]: # Set the random seed
         random_seed = 2
         # Split the train and the validation set for the fitting
         X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size = 0.1, random_state=random_seed)
```

2. CNN model

```
model = Sequential()
```

Firstly, start the convolutional neural network algorithm with Keras Sequential API which is Tensorflow backend, then we can add layers from the input.

```
model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same',
                 activation ='relu', input_shape = (26,26,1)))
model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same',
                 activation ='relu'))
```

```
model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',
                 activation ='relu'))
model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',
                 activation ='relu'))
```

Then we built four layers. The first is the convolutional (Conv2D) layer and we choose to set 32 filters for the first two conv2D layers and 64 filters for the two last ones. Each filter transforms a part of the image, which is defined by the kernel size, using the kernel filter.

The kernel filter matrix is applied on the whole image. Filters can be seen as a transformation of the image. And 'relu' is the rectifier, also called activation function. The rectifier activation function is used to add non linearity to the network.

```
model.add(MaxPool2D(pool_size=(2,2)))
```

The second important layer is the MaxPool2D layer. This pooling layer simply acts as a down sampling filter. It looks at the 2 neighboring pixels and picks the maximal value. It would reduce both computational cost and reduce overfitting. We have to choose the area size of pooled in each time as 2 pixels multiply 2 pixels.

```
model.add(Dropout(0.25))
```

Dropout is the regularization method, where a proportion of nodes in the layer are randomly ignored which is realized by setting their weights to zero for each training sample. This drops randomly a proportion of the network and forces the network to learn features in a distributed way. It would also improve generalization and reduces the overfitting. In this case, our proportion is one fourth.

```
model.add(Flatten())
```

The Flatten layer is used to convert the final feature maps into a one single 1D vector. This flattening step makes use of fully connected layers after some convolutional divided by max-pool layers. It combines all the found local features of the previous convolutional layers.

```
model.add(Dense(256, activation = "relu"))
```

```
model.add(Dense(10, activation = "softmax"))
```

In the end we use the features in two fully-connected layers which is artificial neural networks classifier. In the last layer, the net outputs the distribution of probability of each class.

```
# Define the optimizer
optimizer = RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)
```

In order to minimize the loss, the function above will iteratively improve parameters, including filters kernel values, weights and bias of neurons. It would update the adaptive gradient method by reducing its aggressive, monotonically decreasing learning rate.

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

We define the loss function to measure how poorly our model performs on images with known labels. It is the error rate between the observed labels and the predicted ones. We use the form for categorical classifications called the "categorical_crossentropy". The metric function "accuracy" is used is to evaluate the performance our model. This metric function is similar to the loss function, but only used in evaluation.

```
# Set a learning rate annealer
learning_rate_reduction = ReduceLROnPlateau(monitor='val_acc',
                                            patience=3,
                                            verbose=1,
                                            factor=0.5,
                                            min_lr=0.00001)
```

In order to make the optimizer converge faster to the global minimum of the loss function, we used an annealing method of the learning rate. The LR is the step by which the optimizer walks through the 'loss landscape'. The higher LR, the bigger are the steps and the quicker is the convergence. With the “ReduceLROnPlateau” function, we choose to reduce the LR by half if the accuracy is not improved after 3 epochs.

Then we run the fit. If the epoch equals 1 and set batch size as 86, we get the accuracy of 88.04%. Then we turn on the epochs to 4 and batch size to 100, and we get the accuracy of 99.28%. But we are not sure about the algorithm, for that there may be overfitting problem.

```
datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=10,  # randomly rotate images in the range (degrees, 0 to 180)
        zoom_range = 0.1, # Randomly zoom image
        width_shift_range=0.1,  # randomly shift images horizontally (fraction of total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of total height)
        horizontal_flip=False,  # randomly flip images
        vertical_flip=False)  # randomly flip images


datagen.fit(X_train)
```

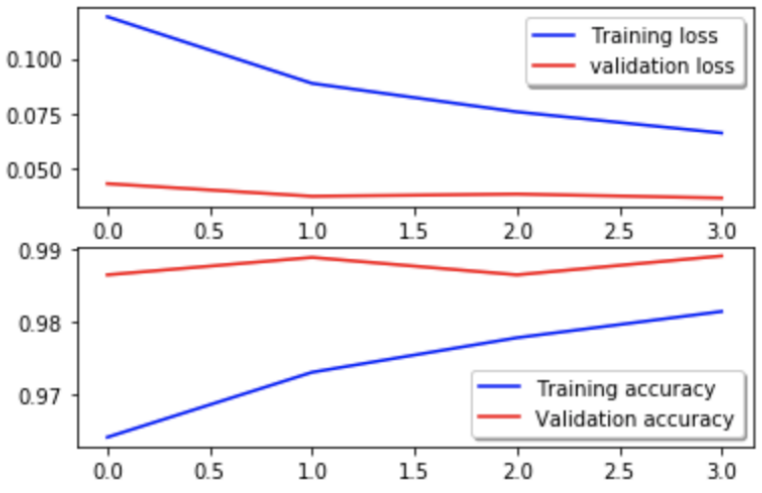
In order to avoid overfitting problem, we need to expand artificially our dataset by altering the training data with small transformations to reproduce the variations occurring when writing a digit. Approaches that alter the training data in ways that change the array representation while keeping the label the same. The detailed explanation is showed in codes which are randomly rotate some training images by 10 degrees, Zoom by 10% some training images, shift images horizontally by 10% of the width and vertically by 10% of the height.

```
# Fit the model
history = model.fit_generator(datagen.flow(X_train,Y_train, batch_size=batch_size),
                              epochs = epochs, validation_data = (X_val,Y_val),
                              verbose = 2, steps_per_epoch=X_train.shape[0] // batch_size
                              , callbacks=[learning_rate_reduction])
```

```
Epoch 1/4
 - 186s - loss: 0.1189 - accuracy: 0.9640 - val_loss: 0.0431 - val_accuracy: 0.9864
Epoch 2/4
 - 186s - loss: 0.0886 - accuracy: 0.9730 - val_loss: 0.0374 - val_accuracy: 0.9888
Epoch 3/4
 - 186s - loss: 0.0757 - accuracy: 0.9777 - val_loss: 0.0384 - val_accuracy: 0.9864
Epoch 4/4
 - 185s - loss: 0.0661 - accuracy: 0.9814 - val_loss: 0.0366 - val_accuracy: 0.9890
```

Finally, we get the modified CNN model, with enlarged training sample, batch size equals 100, epochs equal 4. The outcome of the fit is satisfied for that training score is similar with testing score so there's no overfitting problem anymore.

3. Model Evaluation



We could evaluate the model performance by plotting the accuracy and loss for both training and validation sets. The y-axis is the numerical value of loss and accuracy and the x-axis is the iteration times. As before, the model fits well without overfitting issues.