project-source-code

September 3, 2020

1 Project Overview

- Traffic sign classification is an important task for self driving cars.
- In this project, we will train a Deep Convolutional Neural Network (DCNN) to classify traffic sign images.
- The dataset for experiments contains 51,839 color images classified into 43 different traffic signs.
- The following is the label number with the corresponding category name:
  - 0 = Speed limit (20km/h)
  - 1 = Speed limit (30km/h)
  - 2 = Speed limit (50km/h)
  - 3 = Speed limit (60km/h)
  - 4 = Speed limit (70km/h)
  - 5 = Speed limit (80km/h)
  - 6 = End of speed limit (80km/h)
  - 7 = Speed limit (100km/h)
  - 8 = Speed limit (120km/h)
  - 9 = No passing
  - 10 = No passing for vehicles over 3.5 metric tons
  - 11 = Right-of-way at the next intersection
  - 12 = Priority road
  - 13 = Yield
  - 14 = Stop
  - 15 = No vehicles
  - 16 = Vehicles over 3.5 metric tons prohibited
  - 17 = No entry
  - 18 = General caution
  - 19 = Dangerous curve to the left
  - 20 = Dangerous curve to the right
  - 21 = Double curve
  - 22 = Bumpy road
  - 23 = Slippery road
  - 24 = Road narrows on the right
  - 25 = Road work
  - 26 = Traffic signals
  - 27 = Pedestrians

- 28 = Children crossing
- 29 = Bicycles crossing
- 30 = Beware of ice/snow
- 31 = Wild animals crossing
- 32 = End of all speed and passing limits
- 33 = Turn right ahead
- 34 = Turn left ahead
- 35 = Ahead only
- 36 = Go straight or right
- 37 = Go straight or left
- 38 = Keep right
- 39 = Keep left
- 40 = Roundabout mandatory
- 41 = End of no passing
- 42 = End of no passing by vehicles over 3.5 metric tons

[1]:
```
import numpy as np
import pandas as pd
import pickle, random
from tensorflow import keras
from matplotlib import pyplot as plt
```

2 Data Extraction

[2]:
```
with open("./data/train.p", mode='rb') as tr_data:
    for_training = pickle.load(tr_data)
with open("./data/valid.p", mode='rb') as val_data:
    for_validation = pickle.load(val_data)
with open("./data/test.p", mode='rb') as test_data:
    for_testing = pickle.load(test_data)
```

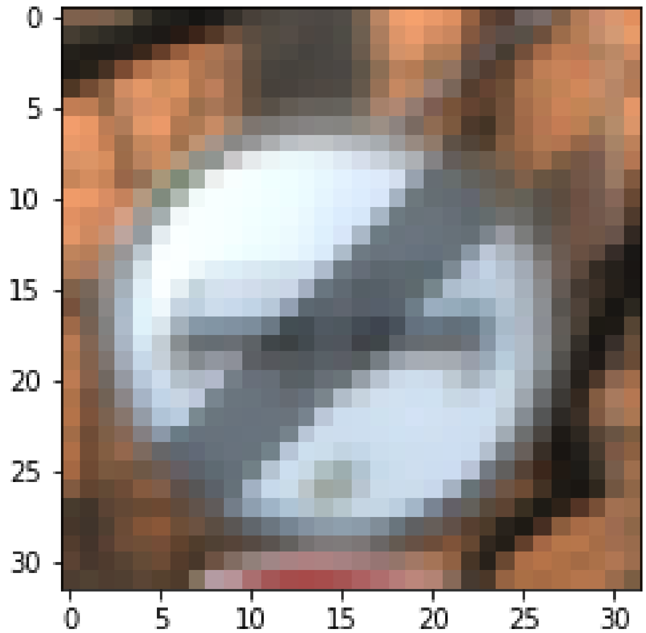
We want to separate the data into three parts that can be used training, validation, and test phases.

[3]:
```
X_train, y_train = for_training['features'], for_training['labels']
X_valid, y_valid = for_validation['features'], for_validation['labels']
X_test, y_test = for_testing['features'], for_testing['labels']
```

After extracting the data, we can plot a random image and print the corresponding label number to re-check the correctness of our actions.

[4]:
```
random_index = np.random.randint(0, 1000)
plt.imshow(X_train[random_index].squeeze())
print(f"The printed image is labelled as: {y_train[random_index]}")
```

```
The printed image is labelled as: 41
```



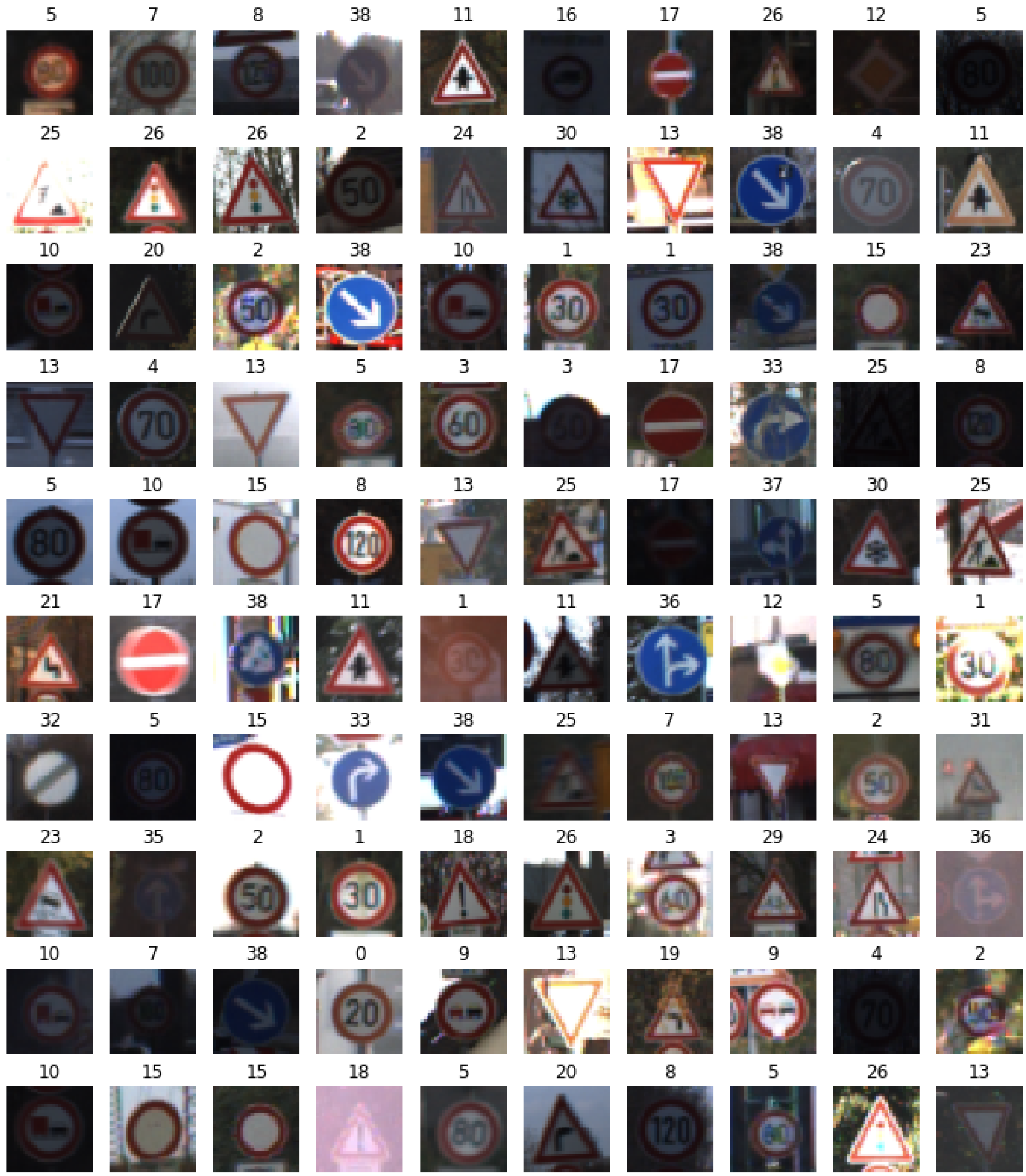
We can also plot grid of images and their corresponding labels.

[5]:
```
n_rows, n_cols = 10, 10

fig, axes = plt.subplots(nrows=n_rows, ncols=n_cols, figsize=(12,14))
axes = axes.ravel() # flatten the axes: before: 10x10; after: 100
n_train = len(X_train)

for index in range(n_rows * n_cols):

    random_index = np.random.randint(0, n_train)
    axes[index].imshow(X_train[random_index].squeeze())
    axes[index].set_title(y_train[random_index])
    axes[index].axis('off')
```



3 Data pre-processing

In this step, we would like to convert the color images into grayscale. Because the color is not the decisive factor in classification of trafic signs. Also, grayscale images contain three times fewer features, which makes the training process less time consuming and less computationally expensive.

[6]:
```
print(f"Dataset containing color images: {X_train.shape}")
X_train_gray = np.sum(X_train, axis=3, keepdims=True)
print(f"Dataset containing grayscale images: {X_train_gray.shape}")
```

```
Dataset containing color images: (34799, 32, 32, 3)
Dataset containing grayscale images: (34799, 32, 32, 1)
```

[7]:
```
print(f"Dataset containing color images: {X_valid.shape}")
X_valid_gray = np.sum(X_valid, axis=3, keepdims=True)
print(f"Dataset containing grayscale images: {X_valid_gray.shape}")
```

```
Dataset containing color images: (4410, 32, 32, 3)
Dataset containing grayscale images: (4410, 32, 32, 1)
```

[8]:
```
print(f"Dataset containing color images: {X_test.shape}")
X_test_gray = np.sum(X_test, axis=3, keepdims=True)
print(f"Dataset containing grayscale images: {X_test_gray.shape}")
```

```
Dataset containing color images: (12630, 32, 32, 3)
Dataset containing grayscale images: (12630, 32, 32, 1)
```

Next, we want to standardize the training, validation, and test data. Since, currently, the features have values ranging from 0 to 255 that makes the training and inference process slow and inefficient. Therefore, we transform the values into considerably smaller ones and make the features follow standard normal distribution. This can be achieved by subtracting mean of the training data and dividing by the standard deviation of the data from each sample in training, validation, and test data, respectively. * It is important to use mean and standard deviation of training data for the standardization. This allows to keep the data distribution the same for validation and test data, too. Otherwise, if training and validation or test data distributions are different, then the training will be useless.

[9]:
```
train_mean, train_std = np.mean(X_train_gray), np.std(X_train_gray)
def standardization(data, mean, std):
    return (data - mean) / std
def stats(data):
    print(f"Mean of the given data is: {np.mean(data)}\nStd of the given data␣
 ↪is: {np.std(data)}")
```

[10]:
```
stats(X_train_gray)
X_train_gray_standardized = standardization(X_train_gray, train_mean, train_std)
stats(X_train_gray_standardized)
```

```
Mean of the given data is: 248.03276711098917
Std of the given data is: 198.02938725647743
Mean of the given data is: 3.162312313596526e-17
Std of the given data is: 0.9999999999999993
```

[11]:
```
stats(X_valid_gray)
X_valid_gray_standardized = standardization(X_valid_gray, train_mean, train_std)
stats(X_valid_gray_standardized)
```

```
Mean of the given data is: 250.66928212691326
Std of the given data is: 203.96106434118835
Mean of the given data is: 0.013313756369449377
Std of the given data is: 1.0299535193583595
```

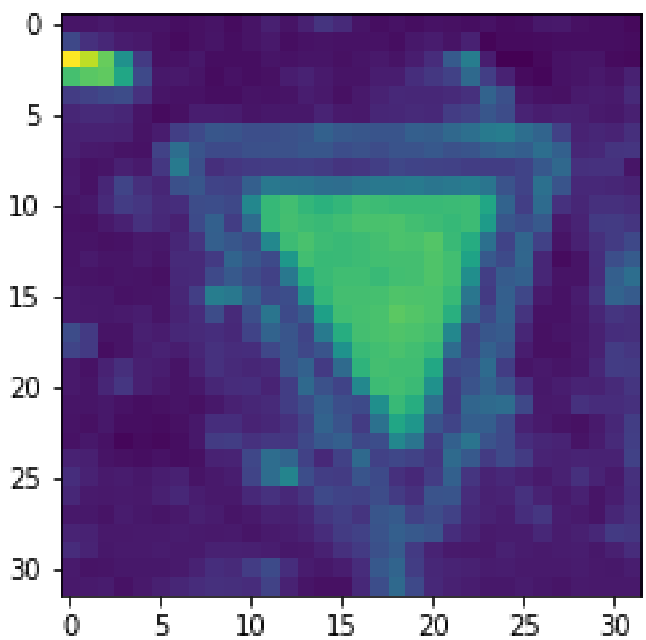
[12]:
```
stats(X_test_gray)
X_test_gray_standardized = standardization(X_test_gray, train_mean, train_std)
stats(X_test_gray_standardized)
```

```
Mean of the given data is: 246.4453810836055
Std of the given data is: 200.29273072777218
Mean of the given data is: -0.00801591142292316
Std of the given data is: 1.0114293312858829
```

Now, we can plot a random image from any of the datasets to see difference of the above transformations.

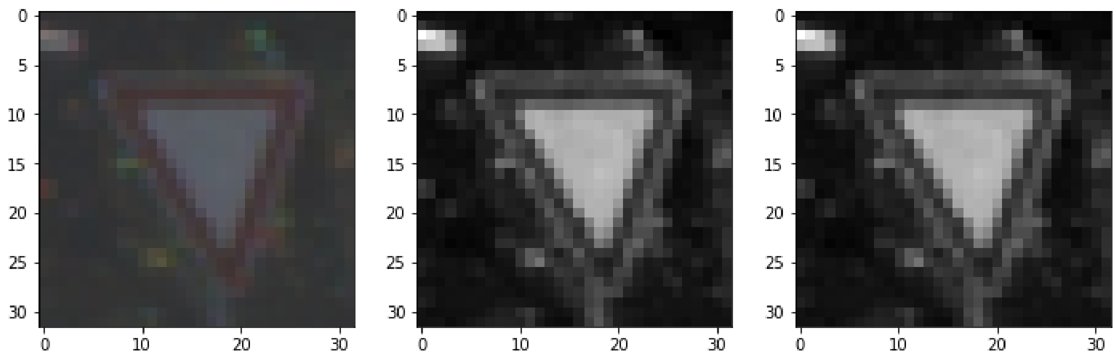
[13]:
```
plt.imshow(X_train[random_index].squeeze())
plt.imshow(X_train_gray[random_index].squeeze())
plt.imshow(X_train_gray_standardized[random_index].squeeze())
```

[13]: <matplotlib.image.AxesImage at 0x217c3c2d5f8>



[14]:
```python
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(12,5))
axes[0].imshow(X_train[random_index].squeeze())
axes[1].imshow(X_train_gray[random_index].squeeze(), cmap='gray')
axes[2].imshow(X_train_gray_standardized[random_index].squeeze(), cmap='gray')
```

[14]:
```
<matplotlib.image.AxesImage at 0x217c3ccacc0>
```



4 DCNN formulation

After the data is ready for being trained, we can formulate DCNN for training the classifier.

[15]:
```python
model = keras.models.Sequential([
    keras.layers.Conv2D(filters=16, kernel_size=(5, 5), activation='relu',␣
 ↪kernel_initializer='he_normal', input_shape=(32, 32, 1)),
    keras.layers.MaxPooling2D(pool_size=2, strides=2), keras.layers.
 ↪Dropout(rate=0.2),
    keras.layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu',␣
 ↪kernel_initializer='he_normal', input_shape=(32, 32, 1)),
    keras.layers.MaxPooling2D(pool_size=2, strides=2), keras.layers.
 ↪Dropout(rate=0.3),
    keras.layers.Flatten(), keras.layers.Dense(units=120, activation='relu'),␣
 ↪keras.layers.Dense(units=43, activation='softmax')
])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 16)        416
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 16)        0
_________________________________________________________________
```

```
dropout (Dropout)            (None, 14, 14, 16)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 12, 12, 32)       4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 32)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 6, 6, 32)         0
_________________________________________________________________
flatten (Flatten)            (None, 1152)             0
_________________________________________________________________
dense (Dense)                (None, 120)              138360
_________________________________________________________________
dense_1 (Dense)              (None, 43)               5203
=================================================================
Total params: 148,619
Trainable params: 148,619
Non-trainable params: 0
_________________________________________________________________
```

[16]:
```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
 ↪metrics=['accuracy'])
```

We can also use EarlyStopping Callback in order not to waste time waiting the end of the training phase. We can set the limit we want and the model training will be stopped automatically.

[17]:
```
class Callback(keras.callbacks.Callback):

    def on_epoch_end(self, epoch, logs={}):

        if (logs.get('loss') < 0.01):
            print("Training loss reached its limit! Cancelling training!")
            self.model.stop_training = True
        elif (logs.get('accuracy') > 0.995):
            print("Training accuracy reached its limit! Cancelling training!")
            self.model.stop_training = True

callback = Callback()
```

[18]:
```
results = model.fit(X_train_gray_standardized, y_train,␣
 ↪validation_data=(X_valid_gray_standardized, y_valid),
                    batch_size=256, epochs=50, verbose=1, callbacks=[callback])
```

```
Train on 34799 samples, validate on 4410 samples
Epoch 1/50
34799/34799 [==============================] - 4s 109us/sample - loss: 2.6885 -
accuracy: 0.3150 - val_loss: 1.5617 - val_accuracy: 0.5748
Epoch 2/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.9842 -
accuracy: 0.7188 - val_loss: 0.7015 - val_accuracy: 0.8190
```

```
Epoch 3/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.5216 -
accuracy: 0.8537 - val_loss: 0.4972 - val_accuracy: 0.8678
Epoch 4/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.3599 -
accuracy: 0.8990 - val_loss: 0.3883 - val_accuracy: 0.8966
Epoch 5/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.2796 -
accuracy: 0.9236 - val_loss: 0.3521 - val_accuracy: 0.9061
Epoch 6/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.2246 -
accuracy: 0.9382 - val_loss: 0.3288 - val_accuracy: 0.9098
Epoch 7/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.1886 -
accuracy: 0.9475 - val_loss: 0.3161 - val_accuracy: 0.9156
Epoch 8/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.1605 -
accuracy: 0.9560 - val_loss: 0.3168 - val_accuracy: 0.9186
Epoch 9/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.1455 -
accuracy: 0.9586 - val_loss: 0.2646 - val_accuracy: 0.9345
Epoch 10/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.1291 -
accuracy: 0.9643 - val_loss: 0.2738 - val_accuracy: 0.9299
Epoch 11/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.1144 -
accuracy: 0.9682 - val_loss: 0.2834 - val_accuracy: 0.9290
Epoch 12/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.1067 -
accuracy: 0.9706 - val_loss: 0.2390 - val_accuracy: 0.9365
Epoch 13/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0929 -
accuracy: 0.9738 - val_loss: 0.2436 - val_accuracy: 0.9363
Epoch 14/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0877 -
accuracy: 0.9755 - val_loss: 0.2389 - val_accuracy: 0.9429
Epoch 15/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0813 -
accuracy: 0.9763 - val_loss: 0.2939 - val_accuracy: 0.9329
Epoch 16/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0762 -
accuracy: 0.9785 - val_loss: 0.2741 - val_accuracy: 0.9395
Epoch 17/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0717 -
accuracy: 0.9797 - val_loss: 0.2327 - val_accuracy: 0.9438
Epoch 18/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0702 -
accuracy: 0.9790 - val_loss: 0.2377 - val_accuracy: 0.9444
```

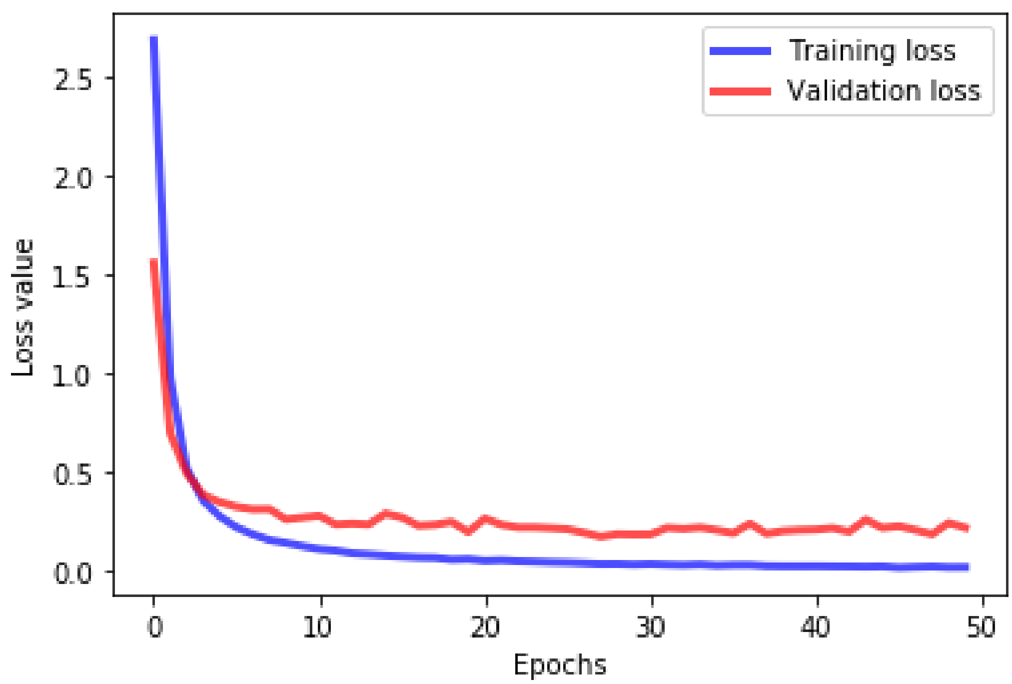
```
Epoch 19/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0612 -
accuracy: 0.9819 - val_loss: 0.2522 - val_accuracy: 0.9431
Epoch 20/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0637 -
accuracy: 0.9820 - val_loss: 0.2004 - val_accuracy: 0.9508
Epoch 21/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0546 -
accuracy: 0.9836 - val_loss: 0.2689 - val_accuracy: 0.9451
Epoch 22/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.0574 -
accuracy: 0.9828 - val_loss: 0.2377 - val_accuracy: 0.9410
Epoch 23/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.0528 -
accuracy: 0.9846 - val_loss: 0.2240 - val_accuracy: 0.9510
Epoch 24/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0505 -
accuracy: 0.9852 - val_loss: 0.2239 - val_accuracy: 0.9465
Epoch 25/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0477 -
accuracy: 0.9862 - val_loss: 0.2199 - val_accuracy: 0.9515
Epoch 26/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0460 -
accuracy: 0.9871 - val_loss: 0.2148 - val_accuracy: 0.9503
Epoch 27/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0427 -
accuracy: 0.9871 - val_loss: 0.1971 - val_accuracy: 0.9553
Epoch 28/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0389 -
accuracy: 0.9884 - val_loss: 0.1765 - val_accuracy: 0.9585
Epoch 29/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0394 -
accuracy: 0.9874 - val_loss: 0.1897 - val_accuracy: 0.9592
Epoch 30/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0368 -
accuracy: 0.9886 - val_loss: 0.1881 - val_accuracy: 0.9535
Epoch 31/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0392 -
accuracy: 0.9881 - val_loss: 0.1887 - val_accuracy: 0.9592
Epoch 32/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0363 -
accuracy: 0.9896 - val_loss: 0.2217 - val_accuracy: 0.9551
Epoch 33/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0341 -
accuracy: 0.9890 - val_loss: 0.2163 - val_accuracy: 0.9531
Epoch 34/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.0372 -
accuracy: 0.9890 - val_loss: 0.2231 - val_accuracy: 0.9476
```

```
Epoch 35/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.0314 -
accuracy: 0.9906 - val_loss: 0.2104 - val_accuracy: 0.9574
Epoch 36/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0340 -
accuracy: 0.9896 - val_loss: 0.1951 - val_accuracy: 0.9587
Epoch 37/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.0343 -
accuracy: 0.9892 - val_loss: 0.2431 - val_accuracy: 0.9467
Epoch 38/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.0285 -
accuracy: 0.9909 - val_loss: 0.1923 - val_accuracy: 0.9580
Epoch 39/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.0275 -
accuracy: 0.9919 - val_loss: 0.2061 - val_accuracy: 0.9544
Epoch 40/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0275 -
accuracy: 0.9920 - val_loss: 0.2108 - val_accuracy: 0.9512
Epoch 41/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0276 -
accuracy: 0.9922 - val_loss: 0.2128 - val_accuracy: 0.9546
Epoch 42/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0269 -
accuracy: 0.9918 - val_loss: 0.2202 - val_accuracy: 0.9540
Epoch 43/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0264 -
accuracy: 0.9922 - val_loss: 0.2023 - val_accuracy: 0.9560
Epoch 44/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0251 -
accuracy: 0.9926 - val_loss: 0.2631 - val_accuracy: 0.9494
Epoch 45/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.0267 -
accuracy: 0.9924 - val_loss: 0.2215 - val_accuracy: 0.9533
Epoch 46/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.0204 -
accuracy: 0.9938 - val_loss: 0.2299 - val_accuracy: 0.9580
Epoch 47/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0234 -
accuracy: 0.9930 - val_loss: 0.2106 - val_accuracy: 0.9590
Epoch 48/50
34799/34799 [==============================] - 1s 34us/sample - loss: 0.0257 -
accuracy: 0.9930 - val_loss: 0.1894 - val_accuracy: 0.9617
Epoch 49/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0221 -
accuracy: 0.9936 - val_loss: 0.2467 - val_accuracy: 0.9537
Epoch 50/50
34799/34799 [==============================] - 1s 35us/sample - loss: 0.0223 -
accuracy: 0.9932 - val_loss: 0.2224 - val_accuracy: 0.9565
```

[19]:
```
tr_loss = results.history['loss']
val_loss = results.history['val_loss']
tr_acc = results.history['accuracy']
val_acc = results.history['val_accuracy']
epochs = range(len(tr_loss))
```

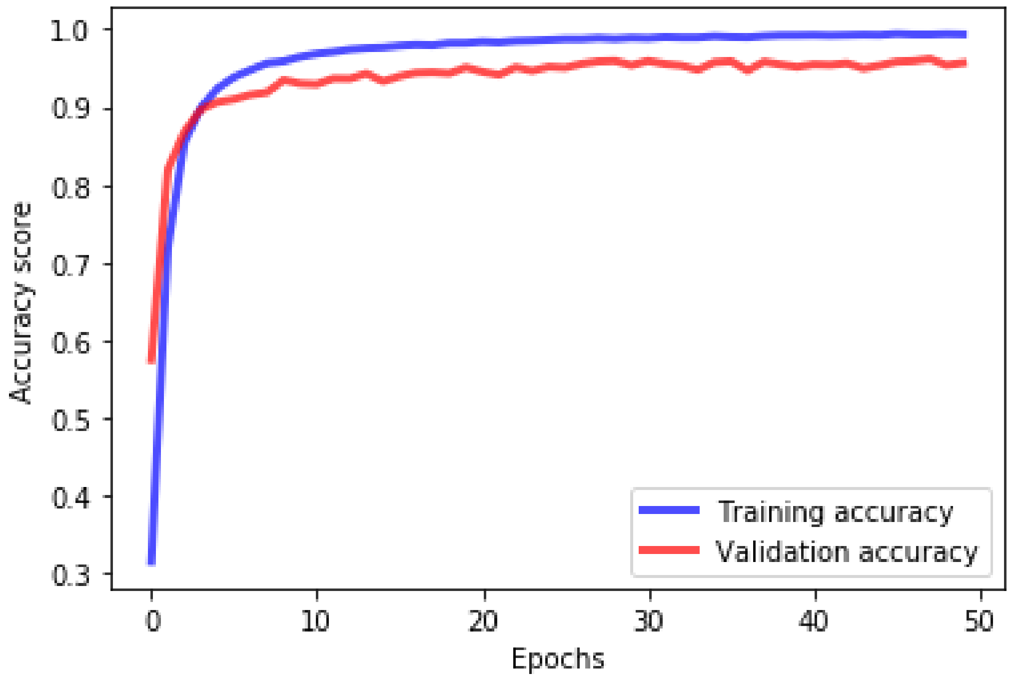
[20]:
```
plt.plot(epochs, tr_loss, linewidth=3, color='b', alpha=0.7, label='Training␣
 ↪loss')
plt.plot(epochs, val_loss, linewidth=3, color='r', alpha=0.7, label='Validation␣
 ↪loss')
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Loss value")
```

[20]:
```
Text(0, 0.5, 'Loss value')
```



[21]:
```
plt.plot(epochs, tr_acc, linewidth=3, color='b', alpha=0.7, label='Training␣
 ↪accuracy')
plt.plot(epochs, val_acc, linewidth=3, color='r', alpha=0.7, label='Validation␣
 ↪accuracy')
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Accuracy score")
```

[21]:
```
Text(0, 0.5, 'Accuracy score')
```



5 Model Evaluation

We can check the generalizability of the model by using unseen test data. Since we have the labels of the test data, we may be interested in the accuracy score of the model in test data, too. However, majority of the test datasets do not have label, so this process is not always possible.

[22]:
```
from sklearn.metrics import confusion_matrix, accuracy_score

test_preds = np.argmax(model.predict(X_test_gray_standardized), axis=1)
acc_score = accuracy_score(y_test, test_preds)
print(f"Accuracy score of the model on test data: {acc_score:.3f}")
```

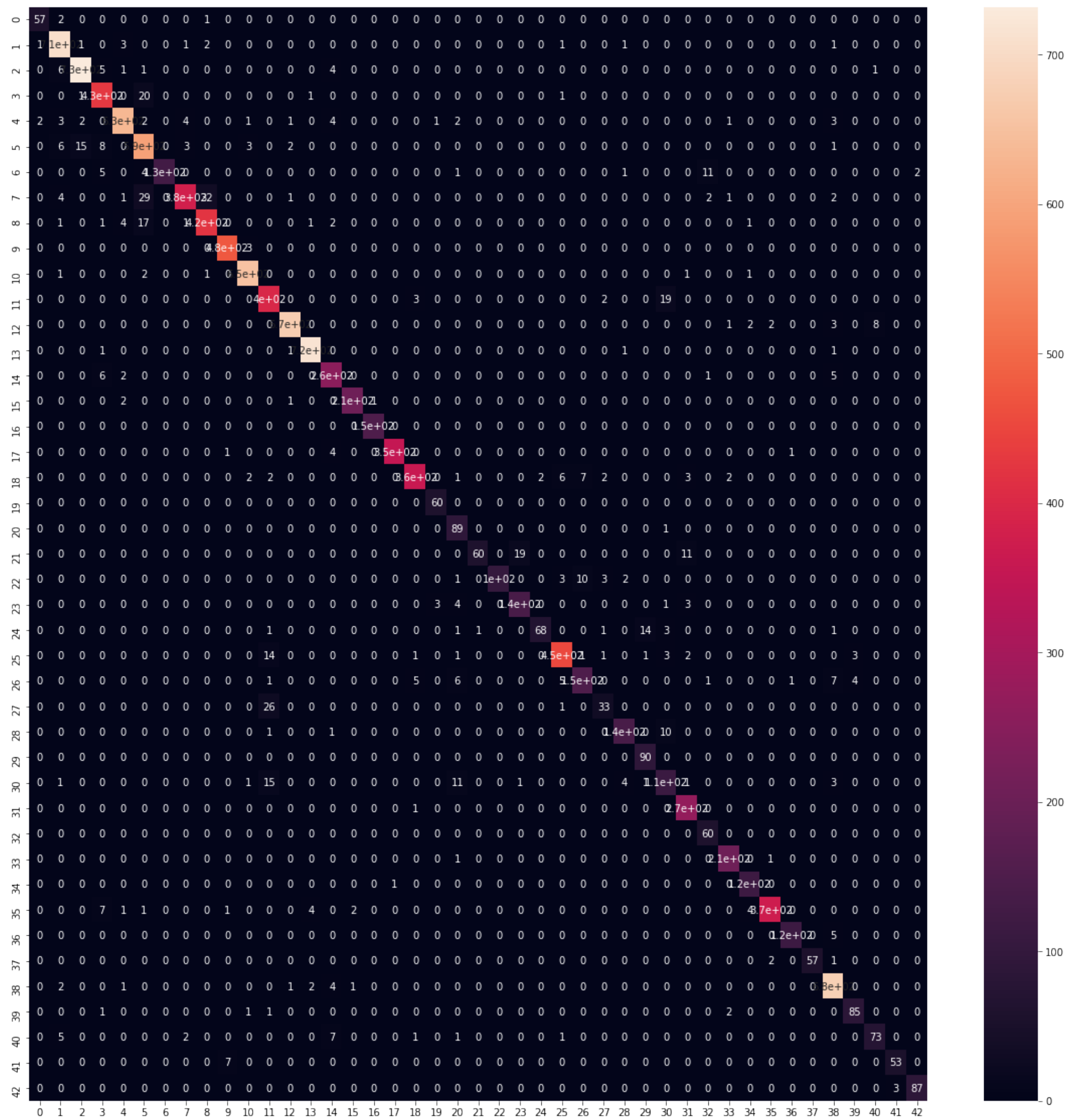
```
Accuracy score of the model on test data: 0.949
```

We may also want to plot confusion matrix to see in which particular categories the model made mistakes.

[23]:
```
import seaborn as sns

cm = confusion_matrix(y_test, test_preds)
plt.figure(figsize=(20, 20))
sns.heatmap(cm, annot=True)
```

[23]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x218b6538748>
```



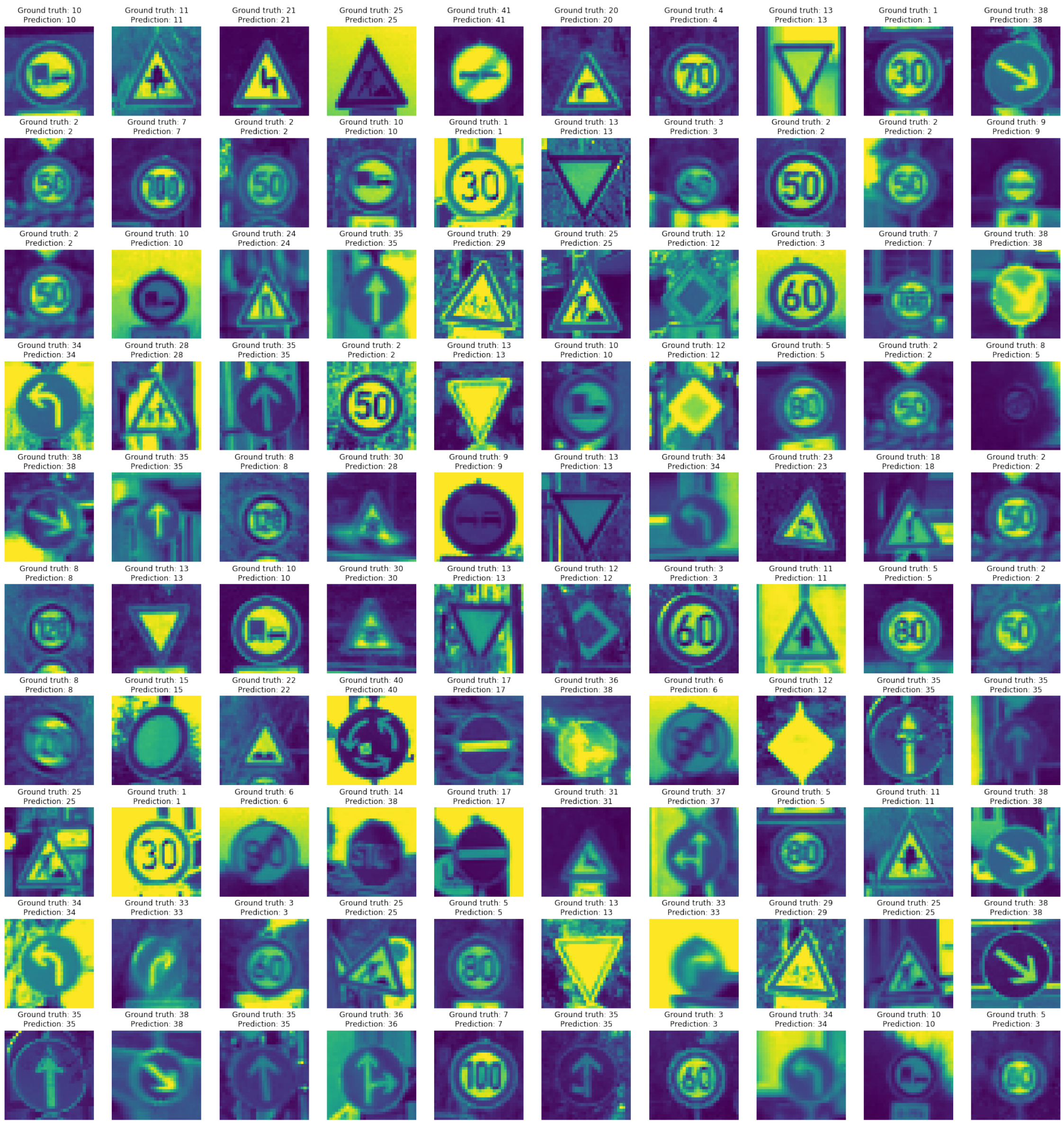
Finally, we can plot some random images from test data with their corresponding labels and observe the performance of the trained model.

[24]:
```
fig, axes = plt.subplots(n_rows, n_cols, figsize=(30, 32))
axes = axes.ravel()

for index in range(n_rows * n_cols):

    random_index = np.random.randint(0, len(X_test_gray_standardized))
    axes[index].imshow(X_test_gray_standardized[random_index].squeeze())
    axes[index].set_title(f"Ground truth: {y_test[random_index]}\nPrediction:␣
 ↪{test_preds[random_index]}")
```

```
    axes[index].axis('off')
```



As can be seen, the trained model performs well even on the test data!

6 That is it for this project! Thank you for following up!