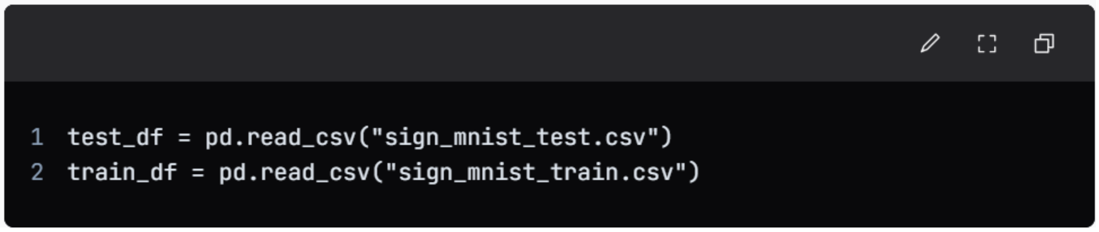
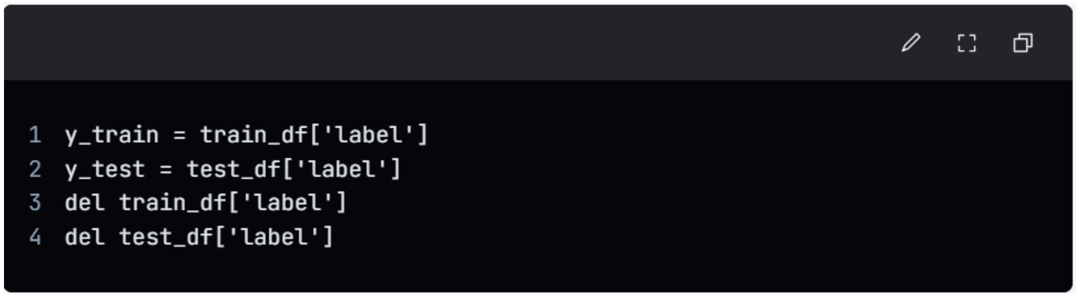
TRAINING THE MODEL



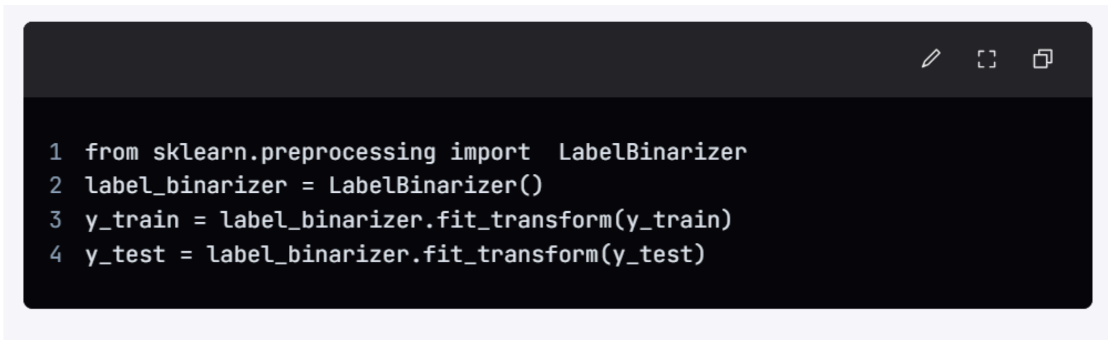
```python
test_df = pd.read_csv("sign_mnist_test.csv")
train_df = pd.read_csv("sign_mnist_train.csv")
```

The provided code reads the test and trains CSV files into Pandas DataFrames. The `pd.read_csv()` function is used to read the CSV files and store the contents in the `test_df` and `train_df` variables. The CSV files contain pixel values and labels for the sign language digits. The pixel values are used as input features for the CNN model, while the labels are used as the target variable for classification.



```python
y_train = train_df['label']
y_test = test_df['label']
del train_df['label']
del test_df['label']
```

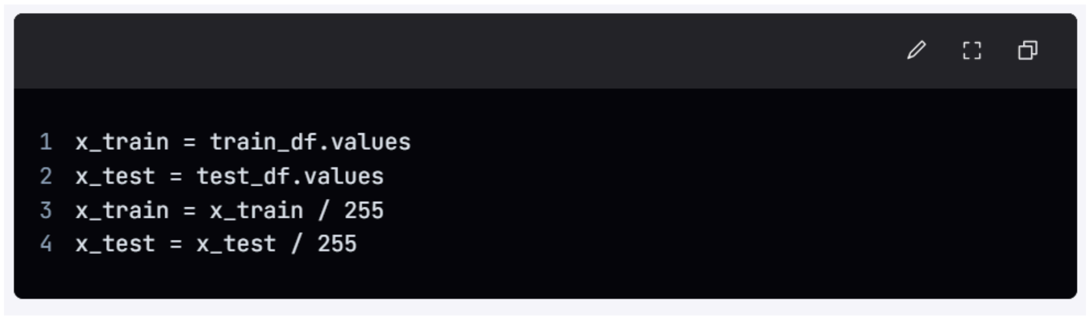
The provided code extracts the label columns from the `train_df` and `test_df` DataFrames and stores them in the `y_train` and `y_test` variables. The label columns are then removed from the DataFrames using the `del` keyword.



```python
from sklearn.preprocessing import  LabelBinarizer
label_binarizer = LabelBinarizer()
y_train = label_binarizer.fit_transform(y_train)
y_test = label_binarizer.fit_transform(y_test)
```

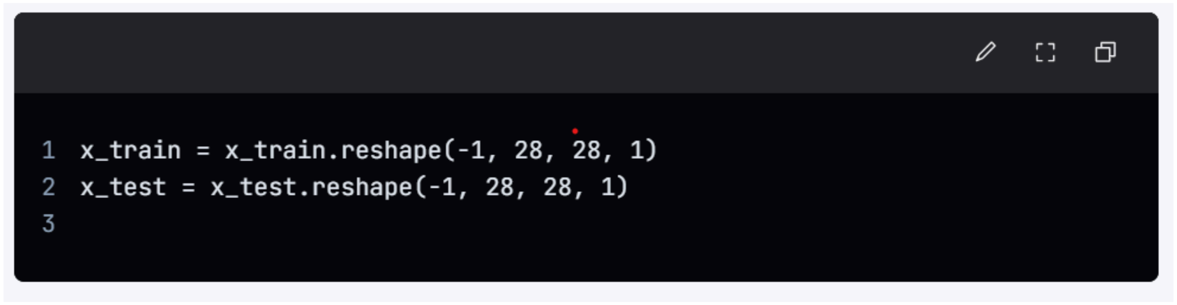
The provided code converts the labels to one-hot encoded format using the `LabelBinarizer` class from scikit-learn. The `LabelBinarizer` class is a useful tool for

converting categorical variables to one-hot encoded format, which is often required for machine learning models that use categorical variables as input features.
The `LabelBinarizer` class is initialized using the `LabelBinarizer()` constructor, and the `fit_transform()` method is used to fit the class to the `y_train` and `y_test` variables and convert them to one-hot encoded format. The `fit_transform()` method returns a new array with the same shape as the input array, but with the categorical variables converted to one-hot encoded format.



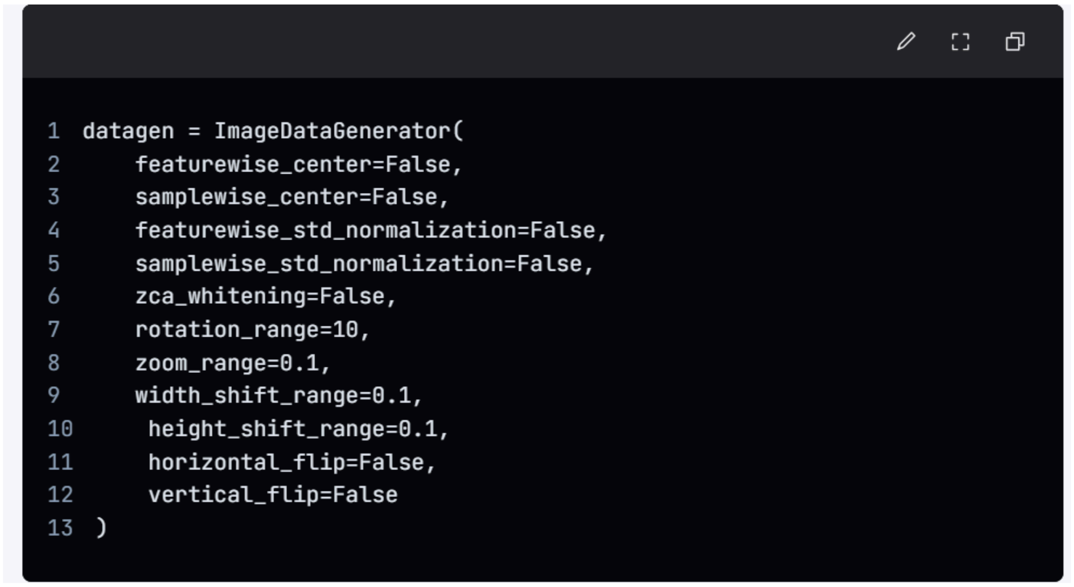
```
x_train = train_df.values
x_test = test_df.values
x_train = x_train / 255
x_test = x_test / 255
```

The provided code extracts the pixel data from the `train_df` and `test_df` DataFrames and normalizes them to a range between 0 and 1. The `values` attribute is used to extract the pixel data as NumPy arrays, and the `/ 255` operation is used to normalize the pixel values to a range between 0 and 1.



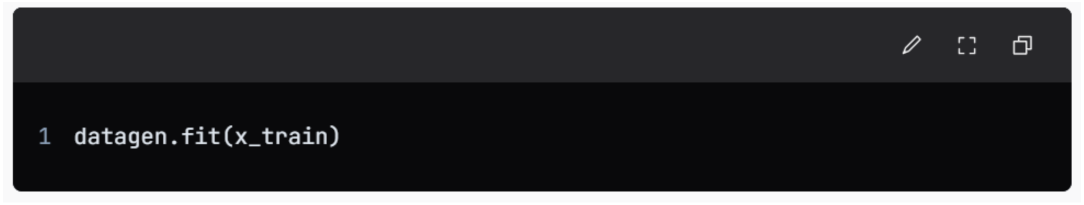
```
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

```

The provided code reshapes the pixel data to a 4-dimensional format suitable for CNN input. The `reshape()` method is used to change the shape of the `x_train` and `x_test` arrays to a shape of `(-1, 28, 28, 1)`, where the first dimension is the batch size, the second and third dimensions are the height and width of the image, and the fourth dimension is the number of color channels.



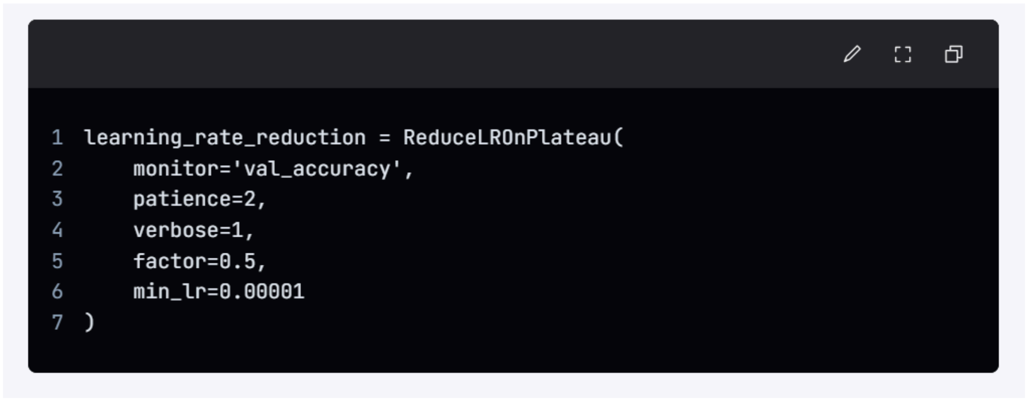
```python
datagen = ImageDataGenerator(
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    rotation_range=10,
    zoom_range=0.1,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=False,
    vertical_flip=False
)
```

The provided code creates an image data generator for data augmentation using the `ImageDataGenerator` class from Keras. The `ImageDataGenerator` class is a useful tool for generating augmented versions of the input images, which can help improve the performance and generalization of the model.
Overall, the use of the `ImageDataGenerator` class is a best practice for data augmentation in Python. The arguments used in this case are well-chosen and should help improve the performance and generalization of the model.



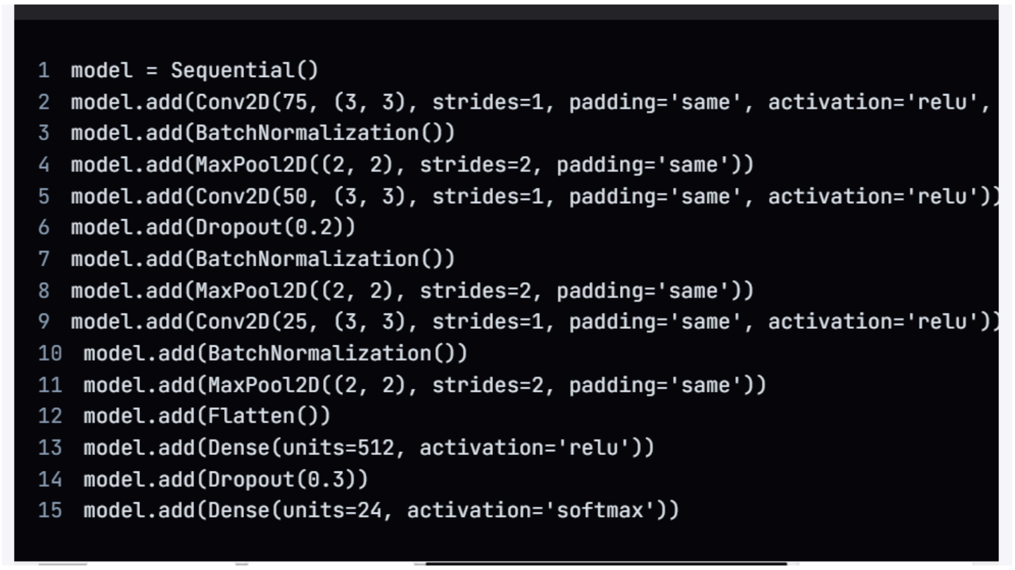
```python
datagen.fit(x_train)
```

The provided code computes necessary statistics for data augmentation using the `fit()` method of the `ImageDataGenerator` class. The `fit()` method is used to compute the mean and standard deviation of the pixel values in the `x_train` array, which are used for data augmentation.



```
learning_rate_reduction = ReduceLROnPlateau(
    monitor='val_accuracy',
    patience=2,
    verbose=1,
    factor=0.5,
    min_lr=0.00001
)
```

The provided code creates a learning rate reduction callback using the `ReduceLROnPlateau` class from Keras. The `ReduceLROnPlateau` class is a useful tool for reducing the learning rate when the validation accuracy stops improving.
Overall, the use of the `ReduceLROnPlateau` class is a best practice for learning rate scheduling in Python. The arguments used in this case are well-chosen and should help improve the convergence and performance of the model.



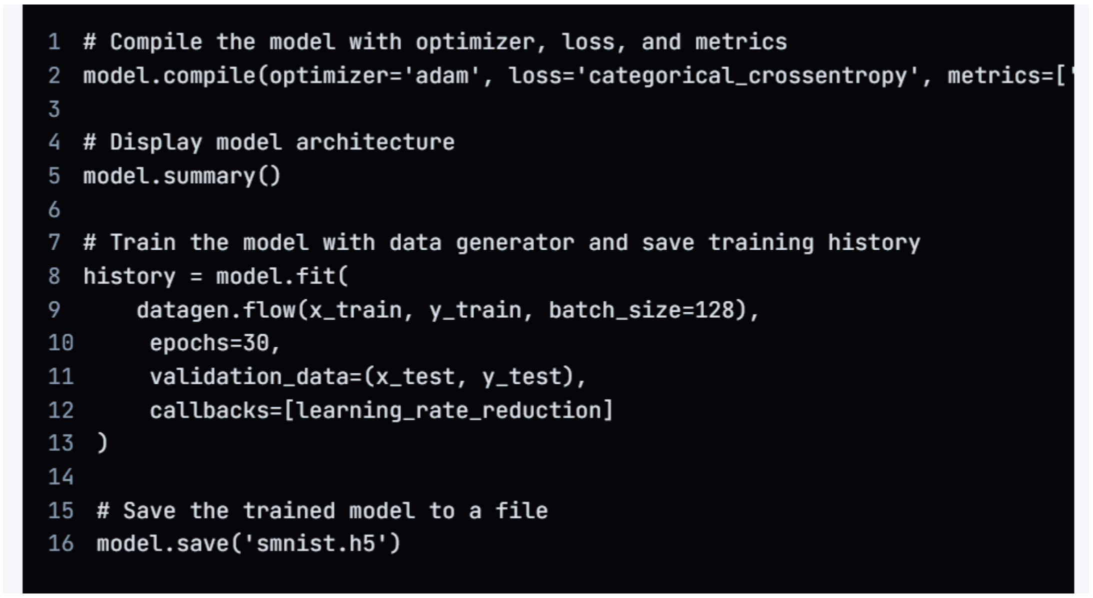
```
model = Sequential()
model.add(Conv2D(75, (3, 3), strides=1, padding='same', activation='relu',
model.add(BatchNormalization())
model.add(MaxPool2D((2, 2), strides=2, padding='same'))
model.add(Conv2D(50, (3, 3), strides=1, padding='same', activation='relu'))
model.add(Dropout(0.2))
model.add(BatchNormalization())
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=24, activation='softmax'))
```

The provided code creates a sequential model for image classification using the `Sequential` class from Keras. The `Sequential` class is a useful tool for creating deep learning models with a linear stack of layers.The `Sequential` model is created by adding layers to the model using the `add()` method. In this case, the following layers are added:

- `Conv2D(75, (3, 3), strides=1, padding='same', activation='relu', input_shape=(28, 28, 1))`: This layer is a 2D convolutional layer with 75 filters, a kernel size of (3, 3), a stride of 1, padding of 'same', and a ReLU activation function. The `input_shape` argument is used to specify the shape of the input data.

- `BatchNormalization()`: This layer is a batch normalization layer, which normalizes the activations of the previous layer to have zero mean and unit variance. This can help improve the convergence and performance of the model.
- `MaxPool2D((2, 2), strides=2, padding='same')`: This layer is a 2D max pooling layer with a pool size of (2, 2) and a stride of 2. This layer reduces the spatial dimensions of the input by taking the maximum value over non-overlapping patches of the input.
- `Conv2D(50, (3, 3), strides=1, padding='same', activation='relu')`: This layer is a 2D convolutional layer with 50 filters, a kernel size of (3, 3), a stride of 1, padding of 'same', and a ReLU activation function.
- `Dropout(0.2)`: This layer is a dropout layer with a dropout rate of 0.2. This layer randomly sets a fraction of the input units to zero during training, which can help prevent overfitting.
- `BatchNormalization()`: This layer is a batch normalization layer, which normalizes the activations of the previous layer to have zero mean and unit variance.
- `MaxPool2D((2, 2), strides=2, padding='same')`: This layer is a 2D max pooling layer with a pool size of (2, 2) and a stride of 2.
- `Conv2D(25, (3, 3), strides=1, padding='same', activation='relu')`: This layer is a 2D convolutional layer with 25 filters, a kernel size of (3, 3), a stride of 1, padding of 'same', and a ReLU activation function.
- `BatchNormalization()`: This layer is a batch normalization layer, which normalizes the activations of the previous layer to have zero mean and unit variance.
- `MaxPool2D((2, 2), strides=2, padding='same')`: This layer is a 2D max pooling layer with a pool size of (2, 2) and a stride of 2.
- `Flatten()`: This layer is a flattening layer, which flattens the spatial dimensions of the input into a 1D array.
- `Dense(units=512, activation='relu')`: This layer is a fully connected layer with 512 units and a ReLU activation function.
- `Dropout(0.3)`: This layer is a dropout layer with a dropout rate of 0.3.
- `Dense(units=24, activation='softmax')`: This layer is a fully connected layer with 24 units and a softmax activation function. This layer outputs the probabilities for each of the 24 classes.

Overall, the use of the `Sequential` class is a best practice for creating deep learning models in Python. The layers added to the model are well-chosen and should help extract useful features from the input images. The use of batch normalization and dropout layers should help improve the convergence and performance of the model. The final softmax layer is appropriate for multi-class classification problems.



```
# Compile the model with optimizer, loss, and metrics
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['

# Display model architecture
model.summary()

# Train the model with data generator and save training history
history = model.fit(
    datagen.flow(x_train, y_train, batch_size=128),
    epochs=30,
    validation_data=(x_test, y_test),
    callbacks=[learning_rate_reduction]
)

# Save the trained model to a file
model.save('smnist.h5')
```

The provided code compiles the model with an optimizer, loss function, and metrics using the `compile()` method of the `Sequential` class. The `compile()` method is used to specify the optimization algorithm, loss function, and evaluation metrics for the model. The `model.summary()` method is then used to display the architecture of the model. This provides a summary of the layers in the model, including the number of parameters in each layer and the output shape of each layer.
The `fit()` method is then used to train the model with data augmentation using the `ImageDataGenerator` class. The `fit()` method is used to specify the training data, batch size, number of epochs, validation data, and callbacks for the model.
Finally, the `save()` method is used to save the trained model to a file named `'smnist.h5'`. This allows the trained model to be loaded and used later for making predictions on new data.