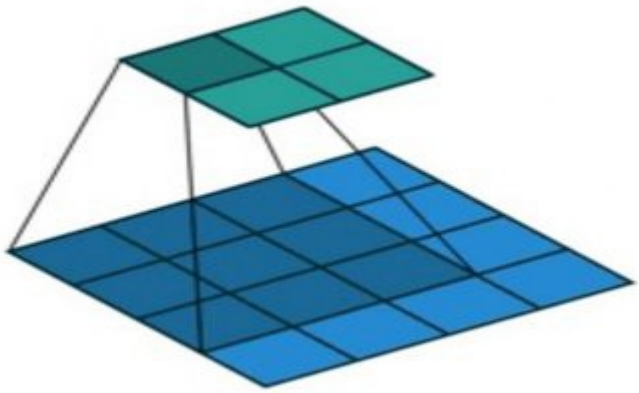
Building a Convolutional Neural Network (CNN) in Keras

Eijaz Allibhai Follow
Oct 17, 2018 · 7 min read

Deep Learning is becoming a very popular subset of machine learning due to its high level of performance across many types of data. A great way to use deep learning to classify images is to build a convolutional neural network (CNN). The Keras library in Python makes it pretty simple to build a CNN.

Computers see images using pixels. Pixels in images are usually related. For example, a certain group of pixels may signify an edge in an image or some other pattern. Convolutions use this to help identify images.

A convolution multiplies a matrix of pixels with a filter matrix or 'kernel' and sums up the multiplication values. Then the convolution slides over to the next pixel and repeats the same process until all the image pixels have been covered. This process is visualized below. (For an introduction to deep learning and neural networks, you can refer to my deep learning article here).



CNN (image credit)

In this tutorial, we will use the popular mnist dataset. This dataset consists of 70,000 images of handwritten digits from 0–9. We will attempt to identify them using a CNN.

Loading the dataset

The mnist dataset is conveniently provided to us as part of the Keras library, so we can easily load the dataset. Out of the 70,000 images provided in the dataset, 60,000 are given for training and 10,000 are given for testing.

When we load the dataset below, X_train and X_test will contain the images, and y_train and y_test will contain the digits that those images represent.

```
from keras.datasets import mnist

#download mnist data and split into train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

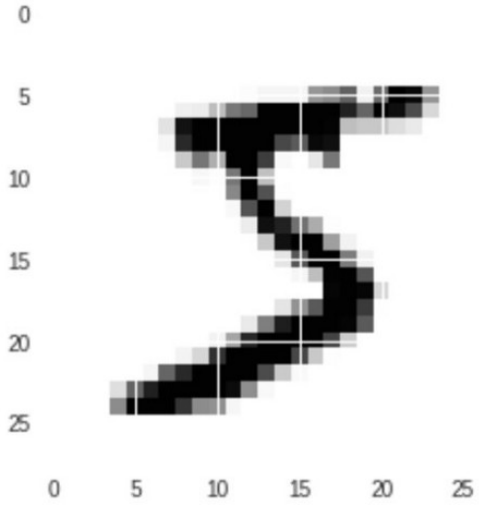
Exploratory data analysis

Now let's take a look at one of the images in our dataset to see what we are working with. We will plot the first image in our dataset and check its size using the 'shape' function.

```
import matplotlib.pyplot as plt

#plot the first image in the dataset
plt.imshow(X_train[0])
```

```
<matplotlib.image.AxesImage at 0x7f83bcfbdc88>
```



```
#check image shape
X_train[0].shape
```

```
(28, 28)
```

By default, the shape of every image in the mnist dataset is 28 x 28, so we will not need to check the shape of all the images. When using real-world datasets, you may not be so lucky. 28 x 28 is also a fairly small size, so the CNN will be able to run over each image pretty quickly.

Data pre-processing

Next, we need to reshape our dataset inputs (X_train and X_test) to the shape that our model expects when we train the model. The first number is the number of images (60,000 for X_train and 10,000 for X_test). Then comes the shape of each image (28x28). The last number is 1, which signifies that the images are greyscale.

```
#reshape data to fit model
X_train = X_train.reshape(60000,28,28,1)
X_test = X_test.reshape(10000,28,28,1)
```

We need to 'one-hot-encode' our target variable. This means that a column will be created for each output category and a binary variable is inputted for each category. For example, we saw that the first image in the dataset is a 5. This means that the sixth number in our array will have a 1 and the rest of the array will be filled with 0.

```
from keras.utils import to_categorical
```

```
#one-hot encode target column
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

```
y_train[0]
```

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

Building the model

Now we are ready to build our model. Here is the code:

```
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten
```

```
#create model
model = Sequential()
```

```
#add model layers
model.add(Conv2D(64, kernel_size=3, activation='relu',
input_shape=(28,28,1)))
model.add(Conv2D(32, kernel_size=3, activation='relu'))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

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

We use the 'add()' function to add layers to our model.

Our first 2 layers are Conv2D layers. These are convolution layers that will deal with our input images, which are seen as 2-dimensional matrices.

64 in the first layer and 32 in the second layer are the number of nodes in each layer. This number can be adjusted to be higher or lower, depending on the size of the dataset. In our case, 64 and 32 work well, so we will stick with this for now.

Kernel size is the size of the filter matrix for our convolution. So a kernel size of 3 means we will have a 3x3 filter matrix. Refer back to the introduction and the first image for a refresher on this.

Activation is the activation function for the layer. The activation function we will be using for our first 2 layers is the ReLU, or Rectified Linear Activation. This activation function has been proven to work well in neural networks.

Our first layer also takes in an input shape. This is the shape of each input image, 28,28,1 as seen earlier on, with the 1 signifying that the images are greyscale.

In between the Conv2D layers and the dense layer, there is a 'Flatten' layer. Flatten serves as a connection between the convolution and dense layers.

'Dense' is the layer type we will use in for our output layer. Dense is a standard layer type that is used in many cases for neural networks.

We will have 10 nodes in our output layer, one for each possible outcome (0–9).

The activation is 'softmax'. Softmax makes the output sum up to 1 so the output can be interpreted as probabilities. The model will then make its prediction based on which option has the highest probability.

Compiling the model

Next, we need to compile our model. Compiling the model takes three parameters: optimizer, loss and metrics.

The optimizer controls the learning rate. We will be using 'adam' as our optmizer. Adam is generally a good optimizer to use for many cases. The adam optimizer adjusts the learning rate throughout training.

The learning rate determines how fast the optimal weights for the model are calculated. A smaller learning rate may lead to more accurate weights (up to a certain point), but the time it takes to compute the weights will be longer.

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.

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.

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

Training the model

Now we will train our model. To train, we will use the 'fit()' function on our model with the following parameters: training data (train_X),

target data (train_y), 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.

```
#train the model
model.fit(X_train, y_train, validation_data=(X_test,
y_test), epochs=3)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/3
60000/60000 [==============================] - 22s 363us/step - loss: 1.3991 - acc: 0.8830 - val_loss: 0.0882 - val_acc: 0.9738
Epoch 2/3
60000/60000 [==============================] - 20s 334us/step - loss: 0.0712 - acc: 0.9790 - val_loss: 0.0874 - val_acc: 0.9729
Epoch 3/3
60000/60000 [==============================] - 20s 334us/step - loss: 0.0484 - acc: 0.9854 - val_loss: 0.0898 - val_acc: 0.9757
<keras.callbacks.History at 0x7fc38442e240>
```

After 3 epochs, we have gotten to 97.57% accuracy on our validation set. That's a very good start! Congrats, you have now built a CNN!

Using our model to make predictions

If you want to see the actual predictions that our model has made for the test data, we can use the predict function. 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.

Note: If we have new data, we can input our new data into the predict function to see the predictions our model makes on the new data. Since we don't have any new unseen data, we will show predictions using the test set for now.

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

```
array([[1.6117248e-09, 8.6684462e-16, 6.8095707e-10, 1.5486043e-08,
        6.2878847e-14, 1.2934288e-15, 1.1453808e-16, 9.9999928e-01,
        1.0626109e-08, 6.9729606e-07],
       [1.3555871e-07, 2.6465393e-06, 9.9999511e-01, 2.0351818e-08,
        1.9796262e-11, 1.6996018e-12, 2.1163373e-06, 1.2008194e-17,
        4.8792381e-10, 2.6086671e-13],
       [6.7238901e-08, 9.9785548e-01, 1.9031411e-04, 3.9194603e-08,
        1.2894072e-04, 1.5791730e-06, 1.2754040e-06, 4.1349044e-09,
        1.8221687e-03, 5.5910935e-08],
       [9.9999356e-01, 1.6909821e-12, 8.2496926e-10, 1.7359107e-11,
        1.7359230e-12, 1.8865266e-13, 6.4659162e-06, 2.3738855e-11,
        1.1319052e-08, 2.6948474e-08]], 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.

```
#actual results for first 4 images in test set
y_test[:4]
```

```
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!

Thanks for reading! The Github repository for this tutorial can be found here!