Convolutional Neural Networks for Multiclass Image Classification — A Beginners Guide to Understand CNN

Following are the arguments of the Conv2D function-

1. **filters** — Number of different filters (feature detectors) that will be applied on the original image to create feature maps. Different types of filters are Edge Detection Filter, Blur Filter etc.
2. **kernel_size** — Dimension of the convolution filter (n x n) matrix.
3. **Activation** — The activation function for the neurons. A general thumb of rule is to use a **Rectifier Linear Unit (Relu)** function as an activation function for every layer besides the output layer. The Relu function also adds non linearity to our network which is highly required to eliminate any linear relationships that does exist in the feature maps.
4. **Input Layer** — Takes the shape of the Input Images and number of channels (3 for color and 1 for B/W image).

Following are the arguments of the MaxPool2D function-

1. **pool_size** — Dimension of pooling matrix (m x m)
2. **strides** — The number of rows and columns traversed per slide.

Classification

The Classification Net consists of two layers — The Flatten Layer and The Fully Connected Layers. The Flatten layer is used to convert the 2D output array from Pooling Layer or Convolutional layer to 1D array (Flattening the input) before feeding it to the fully connected layers.

The fully connected layers are a network of serially connected dense layers that would be used for classification. In a fully connected network every neuron from layer1 is connected to every neuron in layer2. Usually for Computer Vision Application, the dense layers have large numbers of neurons (256, 128 etc.) for achieving higher accuracy in our predictions.

The following code can be used to add a Flatten and a network of fully connected layers to our CNN.

```python
# adding a flatten layer to CNN
cnn.add(tf.keras.layers.Flatten())

# adding fully connected layers
cnn.add(tf.keras.layers.Dense(128, activation='relu'))
cnn.add(tf.keras.layers.Dense(64, activation='relu'))

# output layer -> 6 Neurons for 6 different classes
# activation function used for multiclass classification is softmax, for binary use sigmoid as act
cnn.add(tf.keras.layers.Dense(6, activation='softmax'))
```

Adding_Flatten_Fully_Connected_Layer.py hosted with ❤ by GitHub view raw

Adding Flatten & Fully Connected Layer to CNN

The final layer in our CNN is the output layer. The number of Neurons in this layer is 1 (for regression and binary classification) or equal to number of distinct classes in a multiclass classification task.

After adding the output layer to the network, the next step is to compile the model and then train it on a training dataset.

For compiling the model, we have to specify the following parameters —

1. **optimizer** → Algorithm used for updating the weights of our CNN. "Adam" (Gradient Descent) is one of the popular optimizer used for updating weights.
2. **loss** → Cost function used for calculating the error between the predicted & actual value. In our case we will be using "categorical_crossentropy" since we are dealing with multiclass classification. In case of binary classification we have to use "binary_crossentropy" as loss function.
3. **metrics** → Evaluation metric for checking performance of our model.

For fitting the model, we have to specify the following parameters —

1. **batch_size** → Number of images that will be used by to train our CNN model before updating the weights using back propagation.
2. **epochs** → An **epoch** is a measure of the number of times all of the training images are used once to update the weights.

Till now we have learnt about the different components of a Convolutional Neural Network. Let us now develop our own CNN for image classification.

The problem at our hand is image data of Nature Scenes around the world. The Data contains around 25k images of size 150x150 distributed under 6 categories.

> *{'buildings' -> 0, 'forest' -> 1, 'glacier' -> 2, 'mountain' -> 3, 'sea' -> 4, 'street' -> 5 }*

Our goal is to develop a CNN that can accurately classify the new images into one of these category.

Developing a CNN for Multiclass Image Classification

Before training our CNN on training set images, we first have to apply image augmentation. The reason for performing image augmentation is to eliminate any chance of overfitting our model (training accuracy >> testing accuracy).

Image Augmentation can be applied using **ImageDataGenerator** function of **keras.preprocessing.image** class. Along with augmentation, we also need to rescale our images (every pixel value becomes a value between 0 & 1).

The following code can be used to develop a CNN for image classification.

```python
# import statements
import tensorflow as tf
from keras.preprocessing.image import ImageDataGenerator
import numpy as np

# loading training data
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)
train_generator = train_datagen.flow_from_directory(
        '/kaggle/input/intel-image-classification/seg_train/seg_train',
        target_size=(64, 64),
        batch_size=32,
        class_mode='categorical')

# loading testing data
test_datagen = ImageDataGenerator(rescale=1./255)
test_generator = train_datagen.flow_from_directory(
        '/kaggle/input/intel-image-classification/seg_test/seg_test',
        target_size=(64, 64),
        batch_size=32,
        class_mode='categorical')

# initialising sequential model and adding layers to it
cnn = tf.keras.models.Sequential()
cnn.add(tf.keras.layers.Conv2D(filters=48, kernel_size=3, activation='relu', input_shape=[64, 64,
cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
cnn.add(tf.keras.layers.Conv2D(filters=48, kernel_size=3, activation='relu'))
cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
cnn.add(tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation='relu'))
cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
cnn.add(tf.keras.layers.Flatten())
cnn.add(tf.keras.layers.Dense(128, activation='relu'))
cnn.add(tf.keras.layers.Dense(64, activation='relu'))
cnn.add(tf.keras.layers.Dense(6, activation='softmax'))

# finally compile and train the cnn
cnn.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"])
cnn.fit(x=train_generator, validation_data=test_generator, epochs=30)
```

Image_Classifier.py hosted with ❤ by GitHub view raw

Code for developing a CNN

The following is the output of the model while training.

```
Epoch 1/30
439/439 [==============================] - 52s 119ms/step - loss:
1.1012 - accuracy: 0.5655 - val_loss: 0.9319 - val_accuracy: 0.6333
Epoch 2/30
439/439 [==============================] - 53s 121ms/step - loss:
0.8520 - accuracy: 0.6773 - val_loss: 0.7698 - val_accuracy: 0.7140
Epoch 3/30
439/439 [==============================] - 52s 119ms/step - loss:
0.7336 - accuracy: 0.7292 - val_loss: 0.7235 - val_accuracy: 0.7243
Epoch 4/30
439/439 [==============================] - 53s 120ms/step - loss:
0.6590 - accuracy: 0.7579 - val_loss: 0.6601 - val_accuracy: 0.7593
Epoch 5/30
439/439 [==============================] - 52s 119ms/step - loss:
0.6014 - accuracy: 0.7832 - val_loss: 0.6266 - val_accuracy: 0.7777
Epoch 6/30
439/439 [==============================] - 52s 118ms/step - loss:
0.5648 - accuracy: 0.7964 - val_loss: 0.6243 - val_accuracy: 0.7740
Epoch 7/30
439/439 [==============================] - 52s 119ms/step - loss:
0.5266 - accuracy: 0.8109 - val_loss: 0.5804 - val_accuracy: 0.7880
Epoch 8/30
439/439 [==============================] - 53s 120ms/step - loss:
0.4994 - accuracy: 0.8195 - val_loss: 0.5344 - val_accuracy: 0.8097
Epoch 9/30
439/439 [==============================] - 52s 120ms/step - loss:
0.4811 - accuracy: 0.8253 - val_loss: 0.5347 - val_accuracy: 0.8140
Epoch 10/30
439/439 [==============================] - 53s 120ms/step - loss:
0.4729 - accuracy: 0.8292 - val_loss: 0.5352 - val_accuracy: 0.8113
Epoch 11/30
439/439 [==============================] - 52s 119ms/step - loss:
0.4439 - accuracy: 0.8407 - val_loss: 0.5560 - val_accuracy: 0.8010
Epoch 12/30
439/439 [==============================] - 52s 119ms/step - loss:
0.4406 - accuracy: 0.8409 - val_loss: 0.5026 - val_accuracy: 0.8273
Epoch 13/30
439/439 [==============================] - 54s 124ms/step - loss:
0.4236 - accuracy: 0.8459 - val_loss: 0.5692 - val_accuracy: 0.8107
Epoch 14/30
439/439 [==============================] - 52s 118ms/step - loss:
0.4188 - accuracy: 0.8499 - val_loss: 0.4980 - val_accuracy: 0.8190
Epoch 15/30
439/439 [==============================] - 52s 119ms/step - loss:
0.3954 - accuracy: 0.8590 - val_loss: 0.5697 - val_accuracy: 0.7963
Epoch 16/30
439/439 [==============================] - 54s 123ms/step - loss:
0.3771 - accuracy: 0.8643 - val_loss: 0.5335 - val_accuracy: 0.8127
Epoch 17/30
439/439 [==============================] - 53s 120ms/step - loss:
0.3766 - accuracy: 0.8616 - val_loss: 0.5396 - val_accuracy: 0.8120
Epoch 18/30
439/439 [==============================] - 52s 118ms/step - loss:
0.3639 - accuracy: 0.8666 - val_loss: 0.5345 - val_accuracy: 0.8163
Epoch 19/30
439/439 [==============================] - 52s 119ms/step - loss:
0.3561 - accuracy: 0.8705 - val_loss: 0.4960 - val_accuracy: 0.8313
Epoch 20/30
439/439 [==============================] - 52s 119ms/step - loss:
0.3425 - accuracy: 0.8764 - val_loss: 0.5402 - val_accuracy: 0.8240
Epoch 21/30
439/439 [==============================] - 52s 119ms/step - loss:
0.3405 - accuracy: 0.8746 - val_loss: 0.5499 - val_accuracy: 0.8310
Epoch 22/30
439/439 [==============================] - 52s 119ms/step - loss:
0.3257 - accuracy: 0.8825 - val_loss: 0.5232 - val_accuracy: 0.8253
Epoch 23/30
439/439 [==============================] - 51s 117ms/step - loss:
0.3154 - accuracy: 0.8835 - val_loss: 0.5071 - val_accuracy: 0.8340
Epoch 24/30
439/439 [==============================] - 51s 117ms/step - loss:
0.3121 - accuracy: 0.8858 - val_loss: 0.5404 - val_accuracy: 0.8143
Epoch 25/30
439/439 [==============================] - 52s 118ms/step - loss:
0.3082 - accuracy: 0.8847 - val_loss: 0.5662 - val_accuracy: 0.8217
Epoch 26/30
439/439 [==============================] - 52s 119ms/step - loss:
0.2939 - accuracy: 0.8950 - val_loss: 0.5530 - val_accuracy: 0.8230
Epoch 27/30
439/439 [==============================] - 52s 117ms/step - loss:
0.2821 - accuracy: 0.8970 - val_loss: 0.5394 - val_accuracy: 0.8253
Epoch 28/30
439/439 [==============================] - 52s 118ms/step - loss:
0.2794 - accuracy: 0.8966 - val_loss: 0.5438 - val_accuracy: 0.8360
Epoch 29/30
439/439 [==============================] - 51s 117ms/step - loss:
0.2709 - accuracy: 0.9017 - val_loss: 0.5954 - val_accuracy: 0.8170
Epoch 30/30
439/439 [==============================] - 51s 116ms/step - loss:
0.2683 - accuracy: 0.9023 - val_loss: 0.5720 - val_accuracy: 0.8240
```

As you can see, our model has an accuracy of ~83 % on validation data which is not bad. The accuracy can be further increased by playing around with different parameters such as

1. Increasing number of Neurons
2. Increasing number of hidden layers
3. Increasing epochs
4. Playing around with convolutional layer parameters

So this was all about how you can build your own Convolutional Neural Network for Image Classification.

If you want to learn more about CNNs, you can refer to the below links

1. *Yann LeCun et al., 1998, Gradient-Based Learning Applied to Document Recognition*
2. *Adit Deshpande, 2016, The 9 Deep Learning Papers You Need To Know About (Understanding CNNs Part 3)*
3. *C.-C. Jay Kuo, 2016, Understanding Convolutional Neural Networks with A Mathematical Model*