A guide to an efficient way to build neural network architectures- Part II: Hyper-parameter selection and tuning for Convolutional Neural Networks using Hyperas on Fashion-MNIST

Shashank Ramesh Follow
May 7, 2018 · 14 min read

Intro

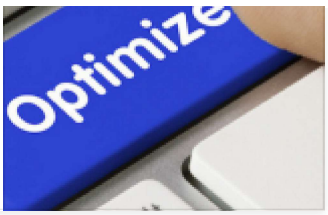
This article is a continuation to the article linked below which deals with the need for hyper-parameter optimization and how to do hyper-parameter selection and optimization using Hyperas for Dense Neural Networks (Multi-Layer Perceptrons)

A guide to an efficient way to build neural network architectures- Part I: Hyper-parameter

Part I: Hyper-parameter…

Intro

medium.com



In the current article we will continue from where we left off in part-I and would try to solve the same problem, the image classification task of the Fashion-MNIST data-set using Convolutional Neural Networks(CNN).

Why CNNs?

The CNNs have several different filters/kernels consisting of trainable parameters which can convolve on a given image spatially to detect features like edges and shapes. These high number of filters essentially learn to capture spatial features from the image based on the learned weights through back propagation and stacked layers of filters can be used to detect complex spatial shapes from the spatial features at every subsequent level. Hence they can successfully boil down a given image into a highly abstracted representation which is easy for predicting.

In Dense networks we try to find patterns in pixel values given as input for eg. if pixel number 25 and 26 are greater than a certain value it might belong to a certain class and a few complex variations of the same. This might easily fail if we can have objects anywhere in the image and not necessarily centered like in the MNIST or to a certain extent also in the Fashion-MNIST data.

RNNs on the other hand find sequences in data and an edge or a shape too can be thought of as a sequence of pixel values but the problem lies in the fact that they have only a single weight matrix which is used by all the recurrent units which does not help in finding many spatial features and shapes. Whereas a CNN can have multiple kernels/filters in a layer enabling them to find many features and build upon that to form shapes every subsequent layer. RNNs would require a lot of layers and hell lot of time to mimic the same as they can find only few sequences at a single layer.

So lets take our quest forward with convolutional networks and see how well could a deeper hyper-parameter optimized version of this do, but before that lets have a look at the additional hyper-parameters in a convolutional neural net.

Hyper-parameters: CNN

Here we will speak about the additional parameters present in CNNs, please refer part-I(link at the start) to learn about hyper-parameters in dense layers as they also are part of the CNN architecture.

1. Kernel/Filter Size: A filter is a matrix of weights with which we convolve on the input. The filter on convolution, provides a measure for how close a patch of input resembles a feature. A feature may be vertical edge or an arch,or any shape. The weights in the filter matrix are derived while training the data. Smaller filters collect as much local information as possible, bigger filters represent more global, high-level and representative information. If you think that a big amount of pixels are necessary for the network to recognize the object you will use large filters (as 11x11 or 9x9). If you think what differentiates objects are some small and local features you should use small filters (3x3 or 5x5). **Note in general we use filters with odd sizes.**

2. Padding: Padding is generally used to add columns and rows of zeroes to keep the spatial sizes constant after convolution, doing this might improve performance as it retains the information at the borders. Parameters for the padding function in Keras are ***Same***- output size is the same as input size by padding evenly left and right, but if the amount of columns to be added is odd, it will add the extra column to the right.***Valid***- Output size shrinks to **ceil((n+f-1)/s) where 'n' is input dimensions 'f' is filter size and 's' is stride length. ceil rounds off the decimal to the closet higher integer,** No padding occurs.

3. Stride: It is generally the number of pixels you wish to skip while traversing the input horizontally and vertically during convolution after each element-wise multiplication of the input weights with those in the filter. It is used to decrease the input image size considerably as after the convolution operation the size **shrinks to ceil((n+f-1)/s) where 'n' is input dimensions 'f' is filter size and 's' is stride length. ceil rounds off the decimal to the closet higher integer.**

4. Number of Channels: It is the equal to the number of color channels for the input but in later stages is equal to the number of filters we use for the convolution operation. The more the number of channels,more the number of filters used, more are the features learnt, and more is the chances to over-fit and vice-versa.

5. Pooling-layer Parameters: Pooling layers too have the same parameters as a convolution layer. Max-Pooling is generally used among all the pooling options. The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality by keeping the max value(activated features) in the sub-regions binned.

> ***Introducing Batch Normalization:-*** *Generally in deep neural network architectures the normalized input after passing through various adjustments in intermediate layers becomes too big or too small while it reaches far away layers which causes a problem of internal co-variate shift which impacts learning to solve this we add a batch normalization layer to standardize (mean centering and variance scaling) the input given to the later layers. This layer must generally be placed in the architecture after passing it through the layer containing activation function and before the Dropout layer(if any) . An exception is for the sigmoid activation function wherein you need to place the batch normalization layer before the activation to ensure that the values lie in linear region of sigmoid before the function is applied.*

Principles/Conventions to build a CNN architecture

> *The basic principle followed in building a convolutional neural network is to* **'keep the feature space wide and shallow in the initial stages of the network, and the make it narrower and deeper towards the end.'**

Keeping the above principle in mind we lay down a few conventions to be followed to guide you while building your CNN architecture

1.Always start by using smaller filters is to collect as much local information as possible, and then gradually increase the filter width to reduce the generated feature space width to represent more global, high-level and representative information

2. Following the principle, the number of channels should be low in the beginning such that it detects low-level features which are combined to form many complex shapes(by increasing the number of channels) which help distinguish between classes.

The number of filters is increased to increase the depth of the feature space thus helping in learning more levels of global abstract structures. One more utility of making the

feature space deeper and narrower is to shrink the feature space for input to the dense networks.

By convention the number of channels generally increase or stay the same while we progress through layers in our convolutional neural net architecture

3. General filter sizes used are 3x3, 5x5 and 7x7 for the convolutional layer for a moderate or small-sized images and for Max-Pooling parameters we use 2x2 or 3x3 filter sizes with a stride of 2. Larger filter sizes and strides may be used to shrink a large image to a moderate size and then go further with the convention stated.

4. Try using padding = same when you feel the border's of the image might be important or just to help elongate your network architecture as padding keeps the dimensions same even after the convolution operation and therefore you can perform more convolutions without shrinking size.

5. Keep adding layers until you over-fit. As once we achieved a considerable accuracy in our validation set we can use regularization components like l1/l2 regularization, dropout, batch norm, data augmentation etc. to reduce over-fitting

5. Always use classic networks like LeNet, AlexNet, VGG-16, VGG-19 etc. as an inspiration while building the architectures for your models. By inspiration i mean follow the trend used in the architectures for example trend in the layers Conv-Pool-Conv-Pool or Conv-Conv-Pool-Conv-Conv-Pool or trend in the Number of channels 32–64–128 or 32–32-64–64 or trend in filter sizes, Max-pooling parameters etc.

Building the CNN architecture

Hyper-parameter tuning of CNNs are a tad bit difficult than tuning of dense networks due to the above conventions. This is because the Hyperas uses random search for the best possible model which in-turn may lead to disobeying few conventions, to prevent this from happening ***we need to design CNN architectures and then fine-tune hyper-parameters in Hyperas to get our best model.***

The approach used with Hyperas while tuning dense networks,wherein we give a set a values for all the hyperparameters and let the module decide which is best won't work for tuning CNN. This is because at each layer the input dimensions decrease due to

operations like convolution and max-pooling hence if we give a range of values for hyperparameters like stride,filter-size etc. there always exists a chance that Hyperas chooses a module which ends up in a negative dimension exception and stops before completion.

So how to build the architecture you ask? lets begin. But before that i would like to recap that on using a Convolution or Pooling layer we reduce the dimensions of the input image of dimensions N to (N-f+1)/s where where 'f' is the filter size and 's' the stride length this will be very helpful to us during the process.

The first example on how to build a CNN architecture is shown below which takes inspiration from the LeNet-5 architecture

> *Step-1 is to make a sheet containing the dimensions ,activation shapes and sizes for the architecture just as shown below*

Please refer the video by AndrewNg for more reference- https://www.youtube.com/watch?v=w2sRcGha9nM&t=485s

Layer	Number of Filters	Padding	Activation Shape	Activation Size
Input Image	-	-	(28,28,1)	784
Conv2d(f=3,s=1)	8	Same	(28,28,8)	6,272
MaxPool(f=2,s=2)	-	Valid	(14,14,8)	1568
Conv2d(f=5,s=1)	16	Valid	(10,10,16)	1,600
MaxPool(f=2,s=2)	-	Valid	(5,5,16)	400
Flatten	-	-	(400,1)	400
Flatten	-	-	(120,0)	120
Dense	-	-	(64,1)	64
Softmax	-	-	(10,1)	10

Here we calculate the Activation shape after every convolution or pooling operation is **(ceil(N+f-1)/s,ceil(N+f-1)/s,Number of filters) wherever the padding value is 'valid'** and **dimensions are (N,N,Number of filters) where the padding used is 'same'** ,here 'N' is input dimensions 'f' is filter size and 's' is stride length. The Activation value is computed by multiplying all values in the dimensions of Activation Shape.

For second stage N=28,f=3,s=1,Number of filters=8 and padding is valid hence we get Activation shape as (28,28,8). Activation Value= 28x28x8=6272

For third stage Max-Pool Layer we get ceil((28–2+1)/2)=ceil(13.5)=14 therefore dimensions are (14,14,8). The last dimension does not change on using a pooling layer. Activation Value= 14x14x8=1568

For fourth stage N=14,f=5,s=1,Number of filters=8 and padding is valid hence we get Activation shape as ((14–5+1)/2,(14–5+1)/2,16)= (10,10,16)

Flatten makes the input into a one-dimensional list for input to the dense layers therefore Activation shape is (400,1). Activation Value= 400x1=400

Dense Activation shape (64,1) states 64 hidden units are used in the dense layer.Activation Value= 64x1=64

> *This computation tells us if we are choosing the right parameters while building our CNN architecture as the architecture* ***must not end up with negative dimensions*** *by going overboard with usage of high values of stride length and filter-sizes*
>
> *It is very important to ensure that you choose your values such that there is a* ***general decreasing trend*** *in the Activation values and there ain't any very abrupt changes in Activation values*

Next we convert the model architecture to keras code.

```
cnn1 = Sequential([
    Conv2D(8, kernel_size=(3, 3),
activation='relu',padding='same',input_shape=input_shape),
    MaxPooling2D(pool_size=(2, 2),strides=2),
    Conv2D(16, kernel_size=(5, 5), activation='relu'),
    MaxPooling2D(pool_size=(2, 2),strides=2),
    Flatten(),
    Dense(120, activation='relu'),
    Dense(84, activation='relu'),
    Dense(10, activation='softmax')
])
```

After converting to keras code to check if the conversion is correct and dimensions are the ones you desire you can use <model name>.summary the Output shape column gives you the output dimensions

```
1 cnn1.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_30 (Conv2D)           (None, 28, 28, 8)         80
_________________________________________________________________
max_pooling2d_26 (MaxPooling (None, 14, 14, 8)         0
_________________________________________________________________
conv2d_31 (Conv2D)           (None, 10, 10, 16)        3216
_________________________________________________________________
max_pooling2d_27 (MaxPooling (None, 5, 5, 16)          0
_________________________________________________________________
flatten_10 (Flatten)         (None, 400)               0
_________________________________________________________________
dense_20 (Dense)             (None, 120)               48120
_________________________________________________________________
dense_21 (Dense)             (None, 84)                10164
_________________________________________________________________
dense_22 (Dense)             (None, 10)                850
=================================================================
Total params: 62,430
Trainable params: 62,430
Non-trainable params: 0
_________________________________________________________________
```

Using Adam optimizer with learning rate of 0.001 generally does well for CNNs, so with we train the architecture using the same to get accuracy values as shown below

```
Train loss: 0.18204240553701917
Train accuracy: 0.932125
------------------------------
Validation loss: 0.24995902764300507
Validation accuracy: 0.9090833333333334
------------------------------
```

We repeat the same procedure for more CNN architectures

Next we use a Conv-Pool-Conv-Pool kind of architecture with doubling the number of filters every stage. The architecture is shown below

Layer	Number of Filters	Padding	Activation Shape	Activation Size
Input Image	-	-	(28,28,1)	784
Conv2d(f=3,s=1)	32	Valid	(26,26,32)	21,632
MaxPool(f=2,s=2)	-	Valid	(13,13,32)	5408
Conv2d(f=3,s=1)	64	Valid	(11,11,64)	7,744
MaxPool(f=2,s=2)	-	Valid	(5,5,64)	1600
Conv2d(f=3,s=1)	128	Valid	(3,3,128)	1,152
MaxPool(f=2,s=2)	-	Valid	(1,1,128)	128

Flatten	-	-	(128,1)	128
Dense	-	-	(64,1)	64
Softmax	-	-	(10,1)	10

Convert to Keras

```
cnn1 = Sequential([
    Conv2D(32, kernel_size=(3, 3),
activation='relu',input_shape=input_shape),
    MaxPooling2D(pool_size=(2, 2),strides=2),
    Conv2D(64, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2),strides=2),
    Conv2D(128, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2),strides=2),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

Using Adam optimizer with learning rate of 0.001 we train the architecture to get accuracy values as shown below

```
Train loss: 0.18507329044366877
Train accuracy: 0.9320625
------------------------------
Validation loss: 0.287726696540912
Validation accuracy: 0.8989166666666667
------------------------------
```

Finally we also train a vgg like model with the trend Conv-Conv-Pool-Conv-Conv-Pool

Layer	Number of Filters	Padding	Activation Shape	Activation Size
Input Image	-	-	(28,28,1)	784
Conv2d(f=3,s=1)	16	Same	(28,28,16)	12,544
Conv2d(f=3,s=1)	16	Same	(28,28,16)	12,544
MaxPool(f=2,s=2)	-	Valid	(14,14,16)	3136
Conv2d(f=3,s=1)	32	Valid	(12,12,32)	4,608
Conv2d(f=3,s=1)	32	Valid	(10,10,32)	3,200
MaxPool(f=2,s=2)	-	Valid	(5,5,32)	800
Flatten	-	-	(800,1)	800
Dense	-	-	(512,1)	512
Softmax	-	-	(10,1)	10

Convert to keras

```
cnn1 = Sequential([
    Conv2D(16, kernel_size=(3, 3),
activation='relu',padding='same',input_shape=input_shape),
    Conv2D(16, kernel_size=(3, 3), activation='relu',padding='same'),
    MaxPooling2D(pool_size=(2, 2),strides=2),
    Conv2D(32, kernel_size=(3, 3), activation='relu'),
    Conv2D(32, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2),strides=2),
    Flatten(),
    Dense(512, activation='relu'),
    Dense(10, activation='softmax')
])
```

Get the accuracy values

```
Train loss: 0.05137992699308476
Train accuracy: 0.9810625
------------------------------
Validation loss: 0.30437974256711703
Validation accuracy: 0.923
------------------------------
```

Step 2- Choose the architecture(s) for which you wish to do hyper-parameter optmization

Among all the models look at the one which has the highest validation set score use it as your base architecture

Among our models the vgg like model has the highest value of accuracy on the validation set hence we choose the base architecture as the one shown below

```
cnn1 = Sequential([
    Conv2D(16, kernel_size=(3, 3),
activation='relu',padding='same',input_shape=input_shape),
    Conv2D(16, kernel_size=(3, 3), activation='relu',padding='same'),
    MaxPooling2D(pool_size=(2, 2),strides=2),
    Conv2D(32, kernel_size=(3, 3), activation='relu'),
    Conv2D(32, kernel_size=(3, 3), activation='relu'),
```

```
    MaxPooling2D(pool_size=(2, 2),strides=2),
    Flatten(),
    Dense(512, activation='relu'),
    Dense(10, activation='softmax')
])
```

> *Step 3- Hyper-parameter optimization*

The train loss in the model is very low as compared to the validation set loss which tells us that the model has over-fit. Therefore we need to tune the hyper-parameters such that we get a low loss and yet don't over-fit for this we will use Hyperas.

Using Hyperas: CNN

We now optimize hyperparameters using Hyperas similiar to the way we have done in Part-I. Please refer to Part-I for more details

To optimize we need 3 code blocks

1. Data Function

The function which directly loads train and validation data from the source or if pre-processing is done it is recommended to store the data after pre-processing in a pickle/numpy/hdf5/csv file and write code in the data function to access data from that file.

```
def data():
    (X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
    X_train, X_val, y_train, y_val = train_test_split(X_train,
y_train, test_size=0.2, random_state=12345)
    X_train = X_train.astype('float32')
    X_val = X_val.astype('float32')
    X_train /= 255
    X_val /= 255
    nb_classes = 10
    Y_train = np_utils.to_categorical(y_train, nb_classes)
    Y_val = np_utils.to_categorical(y_val, nb_classes)
    return X_train, Y_train, X_val, Y_val
```

> *Debugging Tip:- If in case you get any error related to the data function just try to rerun the code block or add import statements of the functions or packages used in the data function again at the beginning of the function*

2. Model Function

```
def model(X_train, Y_train, X_val, Y_val):

    model = Sequential()
    model_choice = {{choice(['one', 'two'])}}
    if model_choice == 'one':
        model.add(Conv2D(16, kernel_size=3,
activation='relu',padding='same', input_shape=(1,28,28),
data_format='channels_first'))
        model.add(Conv2D(16, kernel_size=3,
activation='relu',padding='same'))
        model.add(MaxPooling2D(pool_size=2,strides=2))
        model.add(Dropout({{uniform(0, 1)}}))

        model.add(Conv2D(32, kernel_size=3, activation='relu'))
        model.add(Conv2D(32, kernel_size=3, activation='relu'))
        model.add(BatchNormalization())
        model.add(MaxPooling2D(pool_size=2,strides=2))
        model.add(Dropout({{uniform(0, 1)}}))
    elif model_choice == 'two':
        model.add(Conv2D(32, kernel_size=3,
activation='relu',padding='same', input_shape=(1,28,28),
data_format='channels_first'))
        model.add(Conv2D(32, kernel_size=3,
activation='relu',padding='same'))
        model.add(MaxPooling2D(pool_size=2,strides=2))
        model.add(Dropout({{uniform(0, 1)}}))

        model.add(Conv2D(64, kernel_size=3, activation='relu'))
        model.add(Conv2D(64, kernel_size=3, activation='relu'))
        model.add(BatchNormalization())
        model.add(MaxPooling2D(pool_size=2,strides=2))
        model.add(Dropout({{uniform(0, 1)}}))

    model.add(Flatten())
    model.add(Dense({{choice([256, 512,1024])}}, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout({{uniform(0, 1)}}))
    choiceval = {{choice(['one', 'two'])}}
    if choiceval == 'two':
        model.add(Dense({{choice([256, 512,1024])}},
activation='relu'))
        model.add(BatchNormalization())
        model.add(Dropout({{uniform(0, 1)}}))
```

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

    adam = keras.optimizers.Adam(lr=0.001)

    model.compile(loss='categorical_crossentropy', metrics=
['accuracy'],
                  optimizer=adam)

    model.fit(X_train, Y_train,
              batch_size=256,
              nb_epoch=15,
              verbose=2,
              validation_data=(X_val, Y_val))
    score, acc = model.evaluate(X_val, Y_val, verbose=0)
    print('Val accuracy:', acc)
    return {'loss': -acc, 'status': STATUS_OK, 'model': model}
```

> ***Note the positioning of the Dropout layer it is placed after the Max-Pool layer.*** *The placement is done in this way uphold the definition of Dropout which while learning removes high dependency on small set of features. If placed before the Max-Pool layer the values removed by Dropout may not affect the output of the Max-Pool layer as it picks the maximum from a set of values, therefore only when the maximum value is removed can it be thought of removing a feature dependency. Batch Normalization layer as stated is placed after the activation function is applied.*

In the model function we can choose the hyperparameters we need to optimize. In the above code block we are optimizing for the

1. Number of Hidden Units and Number of layers either one or two for the Dense Network

```
    model.add(Dense({{choice([256, 512,1024])}}, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout({{uniform(0, 1)}}))
    choiceval = {{choice(['one', 'two'])}}
    if choiceval == 'two':
        model.add(Dense({{choice([256, 512,1024])}},
activation='relu'))
        model.add(BatchNormalization())
        model.add(Dropout({{uniform(0, 1)}}))
```

The value of 'choiceval' decides we use a two layer dense network or a single layer

2. Dropout values

```
model.add(Dropout({{uniform(0, 1)}}))
```

3. Number of Channels in the architecture

```
if model_choice == 'one':
        model.add(Conv2D(16, kernel_size=3,
activation='relu',padding='same', input_shape=(1,28,28),
data_format='channels_first'))
        model.add(Conv2D(16, kernel_size=3,
activation='relu',padding='same'))
        model.add(MaxPooling2D(pool_size=2,strides=2))
        model.add(Dropout({{uniform(0, 1)}}))
        model.add(Conv2D(32, kernel_size=3, activation='relu'))
        model.add(Conv2D(32, kernel_size=3, activation='relu'))
        model.add(BatchNormalization())
        model.add(MaxPooling2D(pool_size=2,strides=2))
        model.add(Dropout({{uniform(0, 1)}}))

elif model_choice == 'two':
        model.add(Conv2D(32, kernel_size=3,
activation='relu',padding='same', input_shape=(1,28,28),
data_format='channels_first'))
        model.add(Conv2D(32, kernel_size=3,
activation='relu',padding='same'))
        model.add(MaxPooling2D(pool_size=2,strides=2))
        model.add(Dropout({{uniform(0, 1)}}))
        model.add(Conv2D(64, kernel_size=3, activation='relu'))
        model.add(Conv2D(64, kernel_size=3, activation='relu'))
        model.add(BatchNormalization())
        model.add(MaxPooling2D(pool_size=2,strides=2))
        model.add(Dropout({{uniform(0, 1)}}))
```

The value of '***model_choice***' decides whether we choose the architecture with initial layers as Conv(16)-Conv(16)-Pool-Conv(32)-Conv(32)-Pool or Conv(32)-Conv(32)-Pool-Conv(64)-Conv(64)-Pool. (the number in the bracket is representative of the number of filters in the layer.)

3. Excecute and Interpret

We begin the optimization using the above data and model functions

```
X_train, Y_train, X_val, Y_val = data()

best_run, best_model = optim.minimize(model=model,
                                      data=data,
                                      algo=tpe.suggest,
                                      max_evals=30,
                                      trials=Trials(),
                                      notebook_name='Fashion_MNIST')
```

On executing the above snippet we get the below framework in our output. We use this to match the values of the hyper-parameters tuned.

```
model = Sequential()
model_choice = space['model_choice']
if model_choice == 'one':
    model.add(Conv2D(16, kernel_size=3,
activation='relu',padding='same', input_shape=(1,28,28),
data_format='channels_first'))
    model.add(Conv2D(16, kernel_size=3,
activation='relu',padding='same'))
    model.add(MaxPooling2D(pool_size=2,strides=2))
    model.add(Dropout(space['Dropout']))
    model.add(Conv2D(32, kernel_size=3, activation='relu'))
    model.add(Conv2D(32, kernel_size=3, activation='relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=2,strides=2))
    model.add(Dropout(space['Dropout_1']))
elif model_choice == 'two':
    model.add(Conv2D(32, kernel_size=3,
activation='relu',padding='same', input_shape=(1,28,28),
data_format='channels_first'))
    model.add(Conv2D(32, kernel_size=3,
activation='relu',padding='same'))
    model.add(MaxPooling2D(pool_size=2,strides=2))
    model.add(Dropout(space['Dropout_2']))
    model.add(Conv2D(64, kernel_size=3, activation='relu'))
    model.add(Conv2D(64, kernel_size=3, activation='relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=2,strides=2))
    model.add(Dropout(space['Dropout_3']))

 model.add(Flatten())
 model.add(Dense(space['Dense'], activation='relu'))
 model.add(BatchNormalization())
 model.add(Dropout(space['Dropout_4']))
```

```
 choiceval = space['model_choice_1']
 if choiceval == 'two':
    model.add(Dense(space['Dense_1'], activation='relu'))
    model.add(BatchNormalization())
    model.add(Dropout(space['Dropout_5']))
    model.add(Dense(10, activation='softmax'))
adam = keras.optimizers.Adam(lr=0.001)
```

Now we need to get our values from the parameter indexes in the print(best_run) output

```
{'Dense': 1, 'Dense_1': 2, 'Dropout': 0.2799579955710103,
'Dropout_1': 0.8593089514091055, 'Dropout_2': 0.17434082481320767,
'Dropout_3': 0.2839296185815494, 'Dropout_4': 0.7087321230411557,
'Dropout_5': 0.3273210014856124, 'model_choice': 1, 'model_choice_1':
0}
```

On interpreting the above by matching the values with their position in the framework we get the the best architecture to be

```
cnn1 = Sequential([
    Conv2D(32, kernel_size=(3, 3),
activation='relu',padding='same',input_shape=input_shape),
    Conv2D(32, kernel_size=(3, 3), activation='relu',padding='same'),
    MaxPooling2D(pool_size=(2, 2),strides=2),
    Dropout(0.2),
    Conv2D(64, kernel_size=(3, 3), activation='relu'),
    Conv2D(64, kernel_size=(3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2),strides=2),
    Dropout(0.3),
    Flatten(),
    Dense(256, activation='relu'),
    BatchNormalization(),
    Dropout(0.7),
    Dense(10, activation='softmax')
])
```

The accuracy and loss values for the above architecture are

```
Train loss: 0.14321659112690638
Train accuracy: 0.9463333333333334
------------------------------
```

```
Validation loss: 0.17808779059847196
Validation accuracy: 0.93525
------------------------------
Test loss: 0.20868439328074456
Test accuracy: 0.9227
```

So finally we end up with 92.27% accuracy on our test set which is better than our optimized dense network architecture which gave us an accuracy of 88% on test set. But most importantly our model is not over-fitting and hence will generalize well to unseen points unlike other architecture which may achieve higher accuracy but over-fit to training set.

We can improve this accuracy further by using data augmentation techniques but that i'll leave it to you for experimentation.

Hope the article was of help and that you learned how to make an efficient CNN architecture through this article. Thank you for reading!

Artificial Intelligence | Convolutional Network | Hyperas | Hyperparameter Tuning | AI