# Application of NAS to classification problem

(\* The data used in the experiment are specially selected for computational convenience and understanding of the experiment).

Below we will present the process of searching for architecture for a model that classifies fashion products into 10 categories. For the purpose of the problem (training and evaluation) the Fashion-MNIST database will be used which contains 70000 images (28x28 pixels, greyscaled) with corresponding tags from the 10 categories.

Below is a pipeline training scenario, which will then be optimized with the help of the NAS:

Image containing text, screenshot, diagram, drawing

Auto-generated description

**1.**

**Here is the code (with clues and notes) that builds and trains this model**.

From tensorflow import keras

import numpy as np

from keras.layers import Conv2D,MaxPooling2D, Flatten, Dense

from keras.models import Sequential

from keras.datasets import fashion\_mnist

from keras.utils import to\_categorical

NUM\_CLASSES = 10

def load\_data():

# We divide the images into those that will be used for training and those that will be used to evaluate the model

(x\_train,y\_train),(x\_test,y\_test)=fashion\_mnist.load\_data()

# The images are greyscaled, so the pixels take values between 0 and 255. We normalize between 0 and 1 so that our model is trained more efficiently

x\_train = x\_train.astype(“float32”)/255.0

x\_test = x\_test.astype(“float32”)/255.0

# For the operation of convolution, a third dimension of the image is needed that marks its kind (black and white/colored)

x\_train = np.expand\_dims(x\_train,-1).astype(“float32”)

x\_test = np.expand\_dims(x\_test,-1).astype(“float32”)

# We assign the labels of each image to the corresponding of the 10 categories we defined as a vector 1x10, where the vector will contain zeros and 1 in the position (index) of the corresponding category

y\_train = to\_categorical(y\_train, NUM\_CLASSES)

y\_test = to\_categorical(y\_test, NUM\_CLASSES)

return (x\_train,x\_test),(y\_train,y\_test)

def build\_model():

# We define the serial training scenario of the model (pipeline)

model = Sequential([

Conv2D(32,3,activation='replay'),

Conv2D(64,3,activation='replay'),

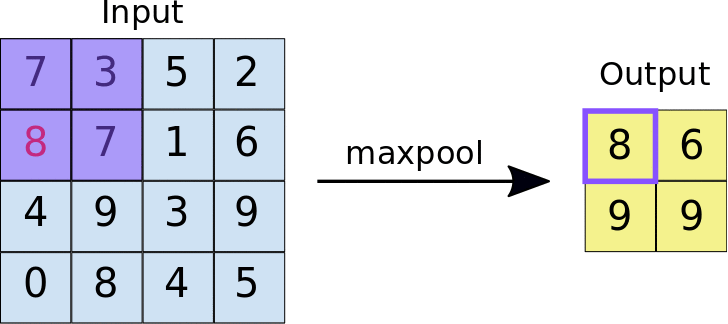
*The convolution function in an image (i.e. an array of pixels) is done to extract features such as edges, angles, boundaries, etc. in order for these features to help draw conclusions. (For example, if the edges at two points in the image are curved, then the image is probably going to depict a blouse.) This function is also shown in the following animation:*

Image containing rectangle, square, drawing, cube

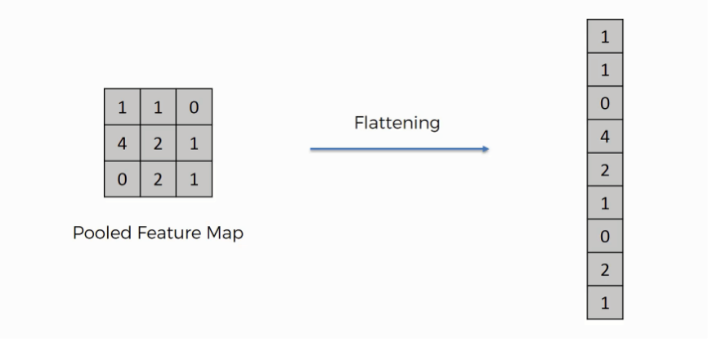
Auto-generated description

MaxPooling2D(),

*In turn, the MaxPooling function makes even more obvious the characteristics extracted from the previous convolution:*

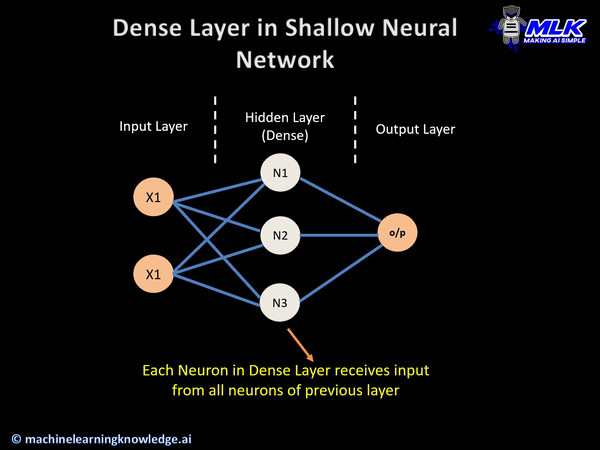


Flatten(),



Dense(100,activation = 'relu'),

*A dense (density) level on a network returns the probabilities (using the trigger function) that an input will carry a particular property. In our case this property is the category to which the input belongs :*



10,000 = 'soft')

])

# We add an optimizer for architecture network weights based on a learning rate at each step

optimizer = keras.optimizers.Adam (learning\_rate=0.01)

# We define the cost function based on which we estimate the network’s performance of the architecture

#In this case, we choose categorical crossentropy which returns a measure (cost) of the difference between the estimated and actual value of each input (in the prediction of the category to which the input belongs). this cost is small if a value close to 0 is returned and large if a value close to 1 is returned

model.compile(optimizer=optimizer, loss=’categorical\_crossentropy’, metrics=[‘accuracy’])

return model

#TRAINING OF THE MODEL

#1. Load the data by calling the corresponding function

(x\_train,x\_test),(y\_train,y\_test) = load\_data()

#2. Constructing the model by calling the corresponding function

model = build\_model()

#3.Education

#At this stage we need to define (the training data, how many images the model will process at the same time, for how many iterations (epochs) to train the model as well as the performance estimation data of the model) as shown in the corresponding arguments

model.fit(x\_train,y\_train,batch\_size=64,epochs=10, validation\_data=(x\_test,y\_test))

# Loss is a metric that evaluates how well the model predicts test data relative to its actual values.

# Accuracy is the percentage of predictions that are correct relative to actual values.

loss, acc = model.evaluate(x\_test,y\_test)

print(acc)

**2.**

Image containing text, screenshot, software, media software

Auto-generated description

With the above scenario we see that our model is evaluated with an accuracy of 0.8980000019073486, i.e. close to 90%. This means that for a particular input (image) the model will place it in the correct category with a probability of about 0.9.

But how can we be sure that the architecture used is the most appropriate for this process and how can we study the above question in a fast and efficient way?

The answer comes to give the NAS as will be shown below:

**3.1 Define search space**

In order to look for better architecture, we need to create a space that will contain all possible architectures. This can happen as shown below, where for the network operations (of convolution, probability formation of belonging and the rate at which the model is trained) in the training scenario we quoted earlier, alternatives are given. In this way we have at our disposal 3\*3\*2\*2\*3 = 108 architectures in the global search space, each of which is tested at different learning rates (learning rates are randomly selected in the continuous interval (0.001-0.01)) :

search\_space = {

"filter\_size\_c1" : {'\_type' : 'choice', '\_value': [32,64,128]},

"filter\_size\_c2" : {'\_type' : 'choice', '\_value': [32,64,128]},

Image containing diagram, text, drawing, technical drawing

Auto-generated description

"kernel\_size\_c1" : {'\_type' : 'choice', '\_value': [3,5]},

"kernel\_size\_c2" : {'\_type' : 'choice', '\_value': [3,5]},

# The number of units (nb units) is related to the Dense operation described above and is the number of nodes that the process will have at its hidden layer.

"nb\_units" : {'\_type' : 'choice', '\_value': [80, 100, 120]},

"learning\_rate" : {'\_type' : 'uniform', '\_value': [0.001,0.01]}

}

**3.2**

Now that the space has been defined, we are reconfiguring our code so that it now trains the model based on the functions provided to it during the experiment.

def build\_model(params):  
 # we define the pipeline   
model = Sequential**([ Conv2D(params['filter\_size\_c1'], params['kernel\_size\_c1'], activation='relu'), Conv2D(params['filter\_size\_c2'], params['kernel\_size\_c2'], activation='relu'), MaxPooling2D(), Flatten**(), Dense(params['nb\_units**'], activation='relu'), Dense(10, activation='softmax') ])**  
  
 # We add an optimizer for our network weights based on a learning rate at each step   
optimizer = keras.optimizers.Adam**(learning\_rate=params['learning\_rate'])#**  
  
 # we define the cost function based on which we will estimate the performance of our network   
 #In this case we choose categorical crossentropy which returns us a measure (cost) of the difference between the estimated and actual value of each input (in the provision of the category to which the entry belongs). This cost is small if a value close to 0 is returned, and large if a value close to 1 is returned

model.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics=['accuracy']) return model  
  
# class responsible for reporting during the experiment the accuracy of the model after the end of each era (epoch) training

class SendMetrics(Callback):

def on\_epoch\_end(self, epoch, logs=None):

nni.report\_intermediate\_result(logs['val\_accuracy'])  
  
#TRAINING OF THE MODEL  
 def run(params):

#1. Load the data using the corresponding function   
(x\_train,x\_test),(y\_train,y\_test) = load\_data()   
#2. Constructing the model by calling the corresponding function   
model = build\_model(params)  
 #3.Training   
#In this stage we need to define (the training data, how many images the model will process at once, for how many iterations (epochs) to train the model, the model performance estimation data, )

model.fit(x\_train,y\_train,batch\_size=200,epochs=10,validation\_data=(x\_test,y\_test),callbacks=[SendMetrics()])

# Loss (or loss) is a metric that evaluates how well the model predicts test data relative to its actual values.

# Accuracy is the percentage of predictions that are correct relative to actual values.   
loss, acc = model.evaluate(x\_test,y\_test)

# After the training is completed, the final accuracy is evaluated and reported as the final result   
nni.report\_final\_result(acc)   
print(acc)

if \_\_name\_\_ == '\_\_main\_\_':

try:

# after training and evaluating each architecture, with the next choice strategy, we select the next architecture from the search space

params = nni.get\_next\_parameters()   
run(params)

except Exception:

raise

**3.3**

Having defined the search space and strategy, we are now able to shape the NAS experiment, with the help of the Neural Network Intelligence (NNI) tool:

from nni.experiment import Experiment

#procedural statements about conducting the experiment

experiment = Experiment('local')

experiment.config.trial\_command = 'python fashionmnistmodel.py'

experiment.config.trial\_code\_directory = '.'

experiment.config.search\_space = search\_space

experiment.config.tuner.name = 'TPE'

experiment.config.trial\_concurrency = 1

#number of architectures to be tested (small for computing convenience)

experiment.config.max\_trial\_number = 5

**4.**

Finally, we perform the experiment:

experiment.run(web server listening portal)

Below we see the results on accuracies of the five most successful architectures:

**Image containing text, screenshot, software

Auto-generated description**

Image containing text, screenshot, font, drawing

Auto-generated descriptionWe see that the architecture with the above characteristics, gives the best accuracy of 0.914 which is better than the one presented earlier ( 0.898 ).

# Bibliography

<https://www.youtube.com/watch?v=td820ts6gUU&t=3687s>