Homework 2

Machine Learning for Image Classification

Lorenzi Flavio 1662963

9 Dec 2019

A Y 2019-2020

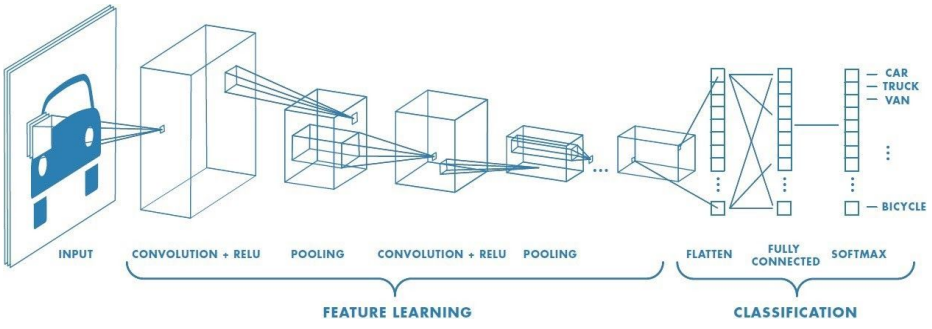
Contents

1. Image Classification

The image classification task it's the process that can classify an image according to its visual content. As required I solved this problem in two ways: a) through the training and testing of a CNN ; b) through the transfer learning methods.

a) Convolutional Neural Network

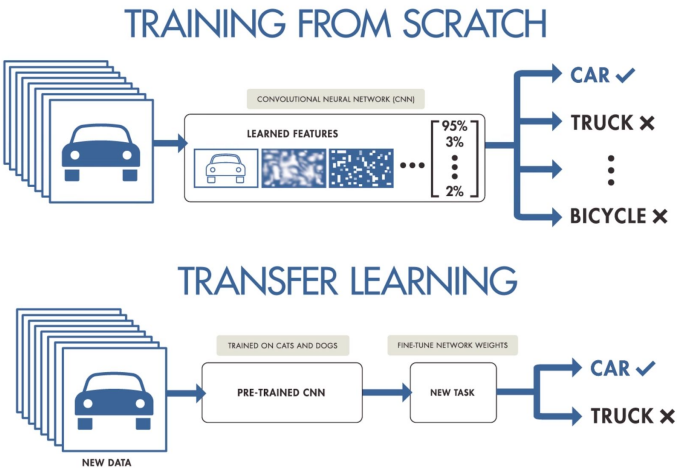
A neural network of this type is called "convolutional" because its layers convolve the input and pass its result to the next layer each time: in fact it employs a mathematical operation called convolution used in place of general matrix multiplication in at least one of their layers.
The input is a tensor with shape (image width) x (image height) x (image depth).
First to convolve into the next, each layer is also characterized by Pooling operation (very useful to reduce image dimension and complexity → "down sampling") and Activation functions (convert and assign some properties to the current output given the current input, preparing it for the next layer).
In the end we have the fully connected layer where each previous layer is connected.
NB: In a fully connected layer, each neuron receives input from *every* element of the previous layers. In a convolutional layer, neurons receive input from only a restricted subarea of the previous layer, called local receptive field". This is a main concept in CNN, because each neuron analyzes only a portion of the current image.



The figure shows a possible example of a CNN structure where we have 2 convolutional layers with ReLU activations and pooling operations in the first part (feature learning) and the "dense" operation for the fully connection with the relative activation function Softmax.
NB: there are various types of architecture for design a CNN (for example VGGNet or AlexNet). For my implementation I used the LeNet model (similar to the figure).

b)Transfer learning method

It is a useful approach in deep learning where pre-trained models are used as the starting point on computer vision and nlp tasks given the vast compute and time resources required to develop neural network models on these problems and from the huge jumps in skill that they provide on related problems.
So we can “transfer” knowledge from pre-trained models to our models in a very short time to implement each related problems we want.



The figure shows as this approach is very important because often it is able to lead us to better results than those of the generic DL approach.

2. Dataset and imported libraries

The dataset define our image classification problem as a multi-label classification: in fact it is characterized in four labels of images (Haze, Sunny, Rainy and Snowy) that describes the Weather for each photo, downloaded by Google Drive and put into training_set folder for the feature learning task. I took 400 images for each class. The test_set instead is characterized of unlabelled mixed images where, however, there is no Haze class. I took from test folder only 100 images to speed up the result.

During the execution of homework requests I used a lot of libraries imported in Python:
-Keras, a high-level neural networks API, used to do operations as "*train_test_split*" for splitting the dataset into train and test; "*to_categorical*" to convert the labels from integers to vectors; "*adam"* optimizer to optimize the model; each layer-operation used in CNN (*Conv2D,MaxPooling2D,Activation,Flatten* …); And so on … .
-**CV2** from computer vision API OpenCV, that is very useful for load/read images "*imread(imagePath)*" and for "*resize"* operations.
-**matplotlib** to plot graphs about accuracy and loss functions.
-**Tensorflow==1.14**, the free and open-source software library for dataflow and differentiable programming across lots of tasks.

3. Description of the models

a) CNN

The implemented model is a **LeNet** Convolutional Neural Network:

```
class LeNet:
    @staticmethod
    def build(width, height, depth, classes):
        # initialize the model
        model = Sequential()
        inputShape = (height, width, depth)

        # if we are using "channels first", update the input shape
        if K.image_data_format() == "channels_first":
            inputShape = (depth, height, width)

        # first set of CONV => RELU => POOL layers
        model.add(Conv2D(20, (5, 5), padding="same", input_shape=inputShape,name='Conv_1'))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

        # second set of CONV => RELU => POOL layers
        model.add(Conv2D(50, (5, 5), padding="same",name='Conv_2'))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

        # first (and only) set of FC => RELU layers
        model.add(Flatten())
        model.add(Dense(500,name='Dense_1'))
        model.add(Activation("relu"))

        # softmax classifier
        model.add(Dense(classes,name='Dense_2'))
        model.add(Activation("softmax"))

        # return the constructed network architecture
        return model
```

We can observe the two convolutional layers where: respectively 20 and 50 are the dimensionality of the output space; (5, 5) is the kernel size; with the Padding operation we apply a frame of zeros that gives more info and more accuracy (SAME stands for same size as the previous one). Then we have the ReLU Activations for each layer, while for the final output we use the Softmax.
In the end we have Flatten (to flat dimension of tensors) and Dense operation to guarantee the fully connection.

So we call the cnn in this way:
model = LeNet.build(width=28, height=28, depth=3, classes=num_class)

And the training is done with:

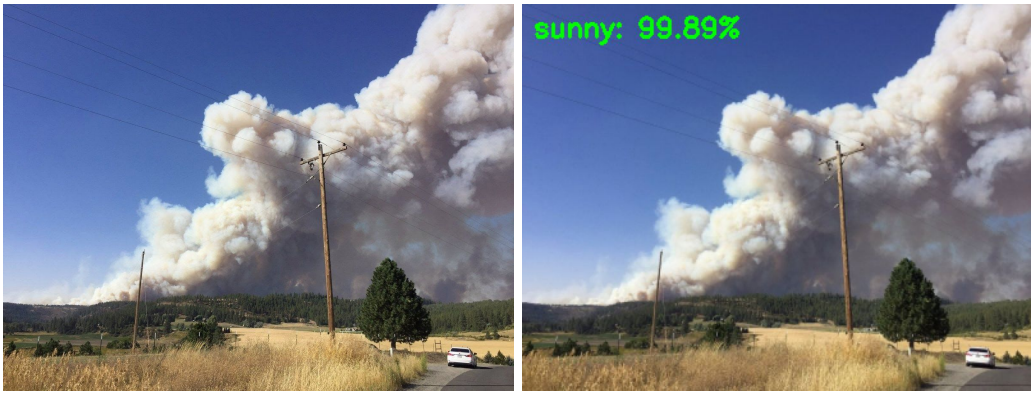
```
# train the network
print("[INFO] Training Network...")
H = model.fit_generator(aug.flow(trainX, trainY, batch_size=BS),
    validation_data=(testX, testY), steps_per_epoch=len(trainX) // BS,
    epochs=EPOCHS, verbose=1 )

# save the model to disk
print("[INFO] Saving Model...")
model_base=args["model"]+'.h5'
model.save(model_base)
```

Where Y are labels of X images during the train/test.

In the end the prediction part given the test_set is characterized by these steps:
-load the image, the model (trained) and the labels
-pre-process the image for classification (resize and dimension operations)
-load the trained cnn model and draw the label on the images

The command lines to start training and testing are:
python train_network.py --dataset training_set
python predict.py --dataset training_set --model trained_model --image test_set/ ‘’



Example of prediction

b)Transfer learning model

As we said the basic premise of TF is simple: take a model trained on a large dataset and *transfer* its knowledge to a smaller dataset to solve a different problem.

So the pre-trained imported model is the **Inception-v3** model that is a widely-used image recognition model that has been shown to attain greater than 78.1% accuracy on the ImageNet dataset. The model itself is made up of symmetric and asymmetric building blocks, including convolutions, average pooling, max pooling, concats, dropouts, and fully connected layers. Batchnorm is applied to activation inputs while Loss is computed via Softmax.
With the script classify.py I imported the model from:
http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz .
NB: it was immediately possible to test it with an image and see its output.
Then with **python retrain.py --output_graph=training_set/retrained_graph.pb --output_labels=training_set/retrained_labels.txt --image_dir=training_set**
I retrained the network with the new (Weather) train_dataset to catch the new labels and tf.graph trends through Tensorflow and thanks to bottleneck layers (reduced dimensionality for input) to transfer its knowledge to the Weather Classification.

```
def get_bottleneck_path(image_lists, label_name, index, bottleneck_dir,
                        category, architecture):
  """Returns a path to a bottleneck file for a label at the given index.

  Args:
    image_lists: Dictionary of training images for each label.
    label_name: Label string we want to get an image for.
    index: Integer offset of the image we want. This will be moduloed by the
    available number of images for the label, so it can be arbitrarily large.
    bottleneck_dir: Folder string holding cached files of bottleneck values.
    category: Name string of set to pull images from - training, testing, or
    validation.
    architecture: The name of the model architecture.

  Returns:
    File system path string to an image that meets the requested parameters.
  """
  return get_image_path(image_lists, label_name, index, bottleneck_dir,
                        category) + '_' + architecture + '.txt'
```

```
def create_bottleneck_file(bottleneck_path, image_lists, label_name, index,
                           image_dir, category, sess, jpeg_data_tensor,
                           decoded_image_tensor, resized_input_tensor,
                           bottleneck_tensor):
  """Create a single bottleneck file."""
  tf.logging.info('Creating bottleneck at ' + bottleneck_path)
  image_path = get_image_path(image_lists, label_name, index,
                              image_dir, category)
  if not gfile.Exists(image_path):
    tf.logging.fatal('File does not exist %s', image_path)
  image_data = gfile.FastGFile(image_path, 'rb').read()
  try:
    bottleneck_values = run_bottleneck_on_image(
        sess, image_data, jpeg_data_tensor, decoded_image_tensor,
        resized_input_tensor, bottleneck_tensor)
  except Exception as e:
    raise RuntimeError('Error during processing file %s (%s)' % (image_path,
                                                                 str(e)))
  bottleneck_string = ','.join(str(x) for x in bottleneck_values)
  with open(bottleneck_path, 'w') as bottleneck_file:
    bottleneck_file.write(bottleneck_string)
```

bottleneck for each img file

So with the method **create_image_lists(image_dir, testing_percentage, validation_percentage)** it builds a list of training images from the file system.
So it will create a new trained model, saved into logs.
Then with **python label.py --image test_set/ ‘’** we can finally observe the results of each image prediction and compare with the ground truth.

```
def load_labels(label_file):
  label = []
  proto_as_ascii_lines = tf.gfile.GFile(label_file).readlines()
  for l in proto_as_ascii_lines:
    label.append(l.rstrip())
  return label

if __name__ == "__main__":
  file_name = ""
  model_file = "training_set/retrained_graph.pb"
  label_file = "training_set/retrained_labels.txt"
  input_height = 299
  input_width = 299
  input_mean = 0
  input_std = 255
  input_layer = "Mul"
  output_layer = "final_result"
```

```
def read_tensor_from_image_file(file_name, input_height=299, input_width=299,
        input_mean=0, input_std=255):
  input_name = "file_reader"
  output_name = "normalized"
  file_reader = tf.read_file(file_name, input_name)
  if file_name.endswith(".png"):
    image_reader = tf.image.decode_png(file_reader, channels = 3,
                                       name='png_reader')
  elif file_name.endswith(".gif"):
    image_reader = tf.squeeze(tf.image.decode_gif(file_reader,
                                                  name='gif_reader'))
  elif file_name.endswith(".bmp"):
    image_reader = tf.image.decode_bmp(file_reader, name='bmp_reader')
  else:
    image_reader = tf.image.decode_jpeg(file_reader, channels = 3,
                                        name='jpeg_reader')
  float_caster = tf.cast(image_reader, tf.float32)
  dims_expander = tf.expand_dims(float_caster, 0);
  resized = tf.image.resize_bilinear(dims_expander, [input_height, input_width])
  normalized = tf.divide(tf.subtract(resized, [input_mean]), [input_std])
  sess = tf.Session()
  result = sess.run(normalized)

  return result
```

Thanks to these two methods the algorithm loads the learned labels and assigns them in according to the input tensors for each image, finally obtaining the probability output (prediction).
NB: the network trained is a cnn where each layer has a softmax activation.
NB: the network follows the gradient descent optimization during labels evaluation.

4. Performance, testing and comparisons

With a dataset of 400 images for each label (tot 1600 images) and training my cnn with a learning rate of 1e^-3, a batch size of 32 (with 64 similar results found) and for 25 epochs (going beyond with this small network there is a risk of overfitting),
I achieved:
-Training phase: accuracy 0.927 ; loss 0.17 .
-Test phase: accuracy 0.9 ; loss 0.21 .
Results reached through cross-validation method supplied by Keras library:
from sklearn.model_selection import train_test_split

pyplot graph.

```
Epoch 14/25
37/37 [==============================] - 1s 32ms/step - loss: 0.2001 - accuracy: 0.9101 - val_loss: 0.2633 - val_accuracy: 0.8860
Epoch 15/25
37/37 [==============================] - 1s 33ms/step - loss: 0.2004 - accuracy: 0.9101 - val_loss: 0.2147 - val_accuracy: 0.9045
Epoch 16/25
37/37 [==============================] - 1s 33ms/step - loss: 0.1855 - accuracy: 0.9200 - val_loss: 0.2076 - val_accuracy: 0.8990
Epoch 17/25
37/37 [==============================] - 1s 33ms/step - loss: 0.1851 - accuracy: 0.9203 - val_loss: 0.2903 - val_accuracy: 0.8755
Epoch 18/25
37/37 [==============================] - 1s 35ms/step - loss: 0.1940 - accuracy: 0.9159 - val_loss: 0.2048 - val_accuracy: 0.9090
Epoch 19/25
37/37 [==============================] - 1s 31ms/step - loss: 0.1982 - accuracy: 0.9102 - val_loss: 0.2059 - val_accuracy: 0.9090
Epoch 20/25
37/37 [==============================] - 1s 32ms/step - loss: 0.1812 - accuracy: 0.9209 - val_loss: 0.2459 - val_accuracy: 0.8950
Epoch 21/25
37/37 [==============================] - 1s 32ms/step - loss: 0.1760 - accuracy: 0.9220 - val_loss: 0.1972 - val_accuracy: 0.9135
Epoch 22/25
37/37 [==============================] - 1s 32ms/step - loss: 0.1758 - accuracy: 0.9252 - val_loss: 0.2518 - val_accuracy: 0.8980
Epoch 23/25
37/37 [==============================] - 1s 35ms/step - loss: 0.1774 - accuracy: 0.9225 - val_loss: 0.2179 - val_accuracy: 0.8970
Epoch 24/25
37/37 [==============================] - 1s 32ms/step - loss: 0.1673 - accuracy: 0.9296 - val_loss: 0.2403 - val_accuracy: 0.8870
Epoch 25/25
37/37 [==============================] - 1s 31ms/step - loss: 0.1706 - accuracy: 0.9271 - val_loss: 0.2118 - val_accuracy: 0.9020
[INFO] Saving Model...
[INFO] Completed...
```

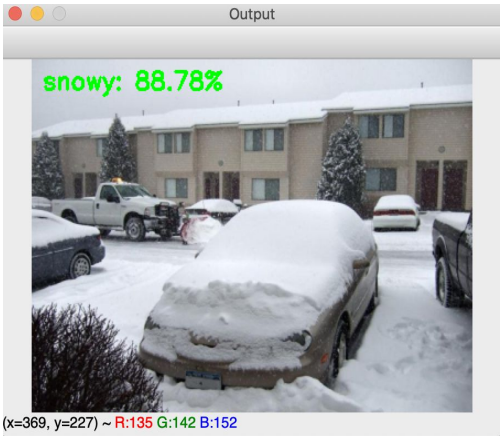
Training progress.

NB: After a few attempts with smaller dataset I achieved a good result with the current one (1600 images for training).
Thanks to the script **prediction.py** we can see for each image included into the test_set folder the relative prediction of my network (with associated probability).
Some results:

ok

ok

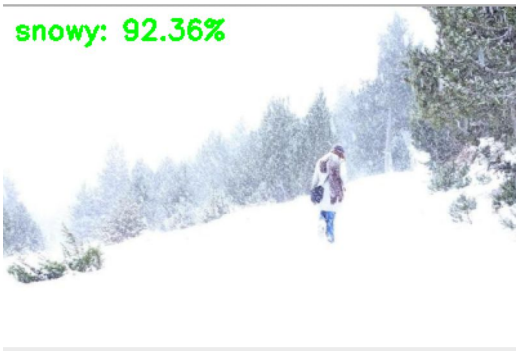
ok

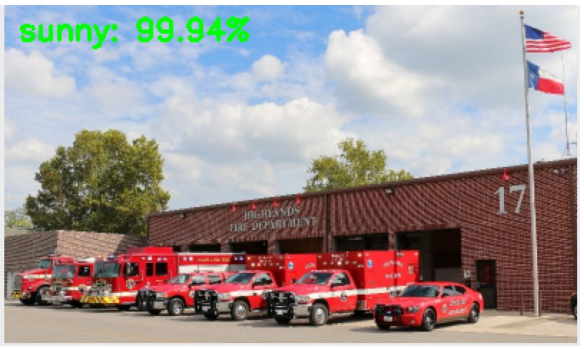


ok

ok

ok



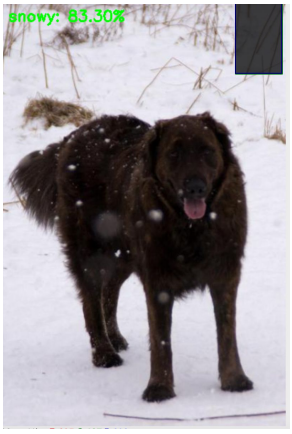
ok


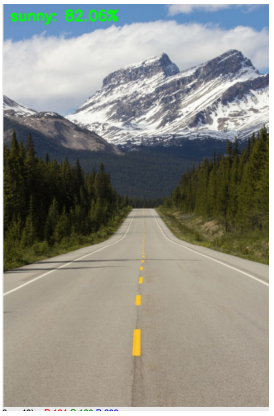
ok

E


E

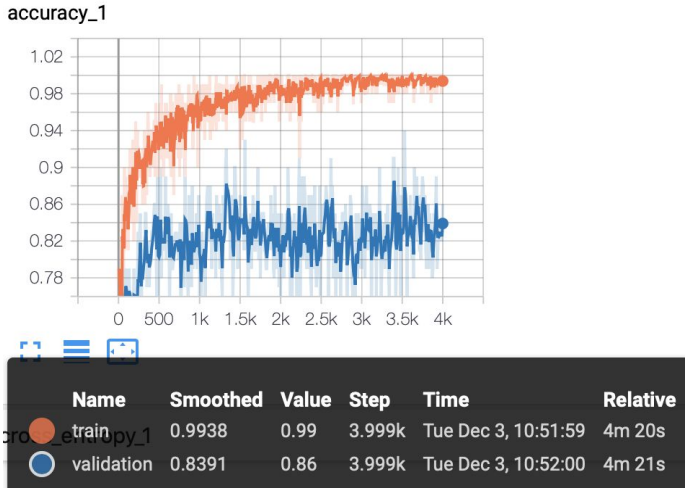
As it is shown in two last image predictions there is a little probability of error and this could due to many things:
-not complete feature extraction during the training: for example in the last photo we have a snowy weather but it predicts rainy because there are no white objects (due to yellow light) and the falling snow is exchanged with rain;
after some experiments I notice that sunny fails when we have sunset or too hot colors, because main extracted features could be a blue sky; same thing with rainy features oriented on wet green grass for example or blurry photos.
-not clear input photos (noise) can be easily misunderstood.
Other examples of the high success of the network:





Despite the noise problem my cnn has however a high success, but in general the image classification task performed only with Classical Deep Learning approach can lead us to much worse results, for example when the train dataset dimension is very larger, with lots of labels but small number of examples: so we prefer the transfer learning approach.

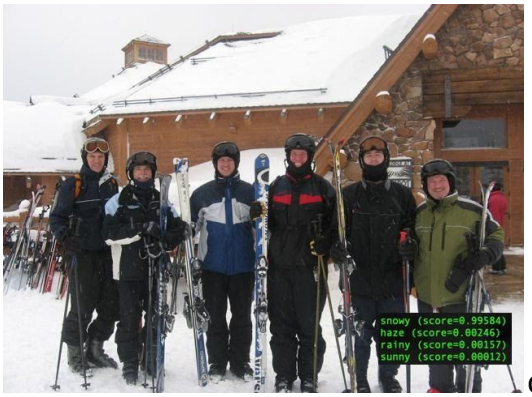
Retraining the Inception-v3 model, after four experiments I tuned hyperparameters with { batch size = 100 and learning rate = 0.01 for a number of epochs = 4000} and through the train_dataset (same used for request A) to achieve the following results in terms of accuracy, that are really satisfying:



tensorboard graph

NB:tb called with **tensorboard --logdir /tmp/retrain_logs**
NB: given the training_dataset the test/validation percentage (what percentage of images to use as a test/validation set) is equal to 10%.

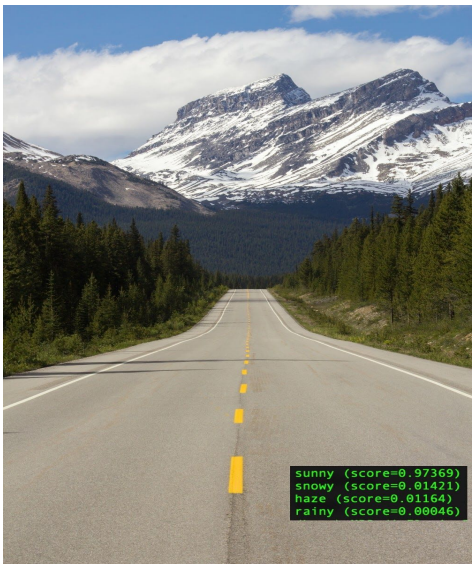
I tested the prediction over the test_dataset (same used for request A) with a margin of error close to zero. Some example are shown below:



ok

ok

ok



ok

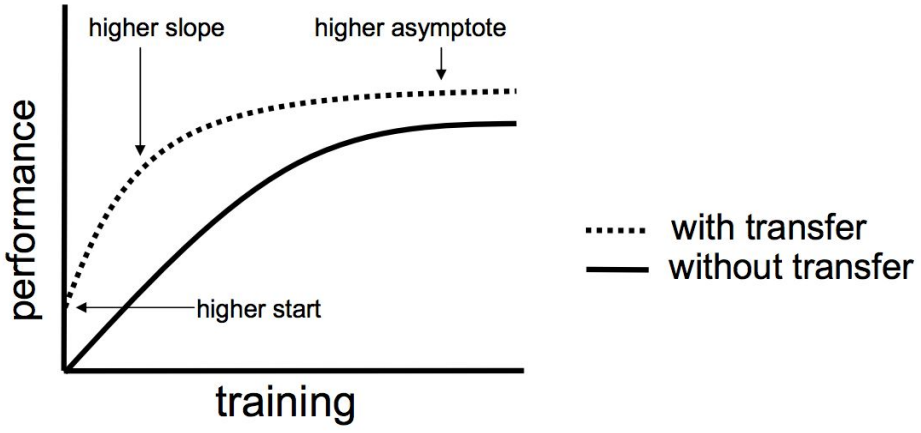
ok SOLVED

E better than prev

small error with snowy=0,42 and sunny=0.41

As it is shown the results are better than classical approach ones; the important thing is that a lot of mistakes are solved (for example the problem of the hot light for snowy recognition in classical cnn). The very few errors I found are all about the blur (as we said in the previous part) of the images.

So through these experiments I checked that the Transfer learning methods can achieve higher performance in terms of accuracy and prediction (and precision).



NB: Transfer Learning works only if the functions learned from the first task are generic, which means that they can also be useful in other related tasks.

5. Conclusion

Thanks to this homework, I was able to experience two learning methods studied in class related to neural networks and transfer learning approach for image classification, so I managed to learn a little better about how the Deep Learning world works.
I saw the construction and the training of the ConvNet (LeNet model) and the mechanisms of TF approach, with the use of a pre-trained model (Inception-v3).

In the end I can conclude that Transfer Learning give us many advantages: for example you can save training time, your neural network works better in most cases and you do not need a lot of data. You can build a solid machine learning model with relatively small training data because the model is already pre-trained.
It is usually used when we do not have sufficiently marked training data to train our network from the beginning or there is already a network that is previously trained in a similar task that is usually trained on huge amounts of data, saving us a great deal of time.

6.References

-https://github.com/vishalprabha/Image-Classification-Transfer-Learning-with-Inception-v3
-https://github.com/Ayushprasad28/Multi-Class-Flower-Classification
-https://classroom.google.com/u/0/c/MjM4NzY1MDgxMDJa
-https://machinelearningmastery.com/transfer-learning-for-deep-learning/
-Machine Learning, Tom M. Mithcell, McGraw-Hill Interational Edition
-https://cloud.google.com/tpu/docs/inception-v3-advanced