Diploma research on the subject of

Illumination Estimation Using ML

Author:	Roman Dubovyi
Study:	Master's degree in Computer Science in Polish-Japanese Academy of Information Technology, Warsaw, Poland
Specialization:	Project Management (IM)
Student id:	s13709
E-mail:	r_dubovoy@yahoo.com
First Supervisor:	
Second Supervisor:	

Abstract

This research proposes another way of determining light source position using Machine Learning. In the century that promises us rise and rapid growth of Artificial Intelligence (AI) we can expect that computational capabilities of our machines will enable us to embed such AI-driven solutions in almost any software and/or hardware solution.

Another promising trend in technologies is Augmented Reality (AR). Today we already can see that our mobile devices with AR on board can deliver significant amount of user-driven content. The good examples that generate big revenues and have millions of daily users are AR filters that allow users to place a virtual object on top of the face. Big players in this niche are Snapchat and Instagram, I think that everyone smiled at least once when using one of their filters.

Following the trends, I took the liberty of exploring the possibility of illuminating virtual object depending on user's face illumination. The idea is to let robust and compact Neural Network decide about how to cast light on virtual component while being efficient enough to be deployed on mobile device. Having limited resources this thesis explores only the tip of the great potential of such technology.

Abstract (polish)

Contents

List of Graphs and Images

Abbreviations and Keywords

Introduction

The notion of Artificial Intelligence was as old as the first computer processor. Many great people speculated and wondered about how much of a change would Artificial Intelligence introduce to our world.

Although the theory began to convert into reality decades ago, the "Artificial Intelligence Winter" took place very soon in 1970's. Many promises couldn't have been kept due to lack of funding and high computational costs. While AI still couldn't solve any of the real problems during that period in time, some of the very important concepts of Machine Learning were developed, like Convolutional Neural Networks and backpropagation.

But as the time passed and technology advanced, we got to the point when we had so much computational power on our hands that AI finally began to find its purpose in business applications sparking the new wave of interest. Today AI is embedded in some of the applications so tightly that it basically defines the purpose of such application.

With the ever-increasing power of our devices some people raise the question whether at some point replacing classical approach to the problem solving with Machine Learning methods may be possible. For my thesis I decided to perform a case study to determine if it will be viable to use Machine Learning approach in the problem of determining light source position. The premise of the problem is very easy, a user wants to make a snapshot of himself with one of the new AR filters, for example he wants to place a hat on his head. The problem is to make hat of relatively the same illumination as the users face to make things a bit more realistic. The other part of a problem is to design a lightweight model that will execute fast enough on mobile device.

I'd like to explore one way of achieving the solution for such problem. In this thesis I describe the steps and reasoning behind them on the way of getting the product to alpha stage. This stage usually serves as the milestone for a product, when the potential is clearly visible and further improvements and strategies are proposed. This project incorporates usage of classical programming, usage of Deep Learning API, usage of pre-trained models, pipeline composition, data generation, feature extraction, 3d scene composing and rendering. All of that I will gladly describe in this paper.

Exploring the Problem

To begin with I'd like to talk more about the nature of the problem. Understanding is very intuitive. I think it's better to begin with an actual example right away. For that I used my Instagram profile and found an Instagram filter that places a hat on top of your head.

Let's take a look:

Image 1 'Me Instagram'

The hat looks a lot out of this world, isn't it? And I am not talking about the color. Our eyes are used to objects being illuminated or shaded. As we can see half of my face is in the shadows, because the light source is on the left from my perspective. What makes virtual object pop so hard from the rest of the image is misalignment of its illumination comparing to the rest of the image. On the photo it looks like the source of the light for this hat comes from the top or even ambient.

To help myself make a point I prepared a little sketch. On it I depict boundaries of the objects and also shadows as diagonal lines. So that's what we see on this example in the matter of shadows. That's what we see currently:

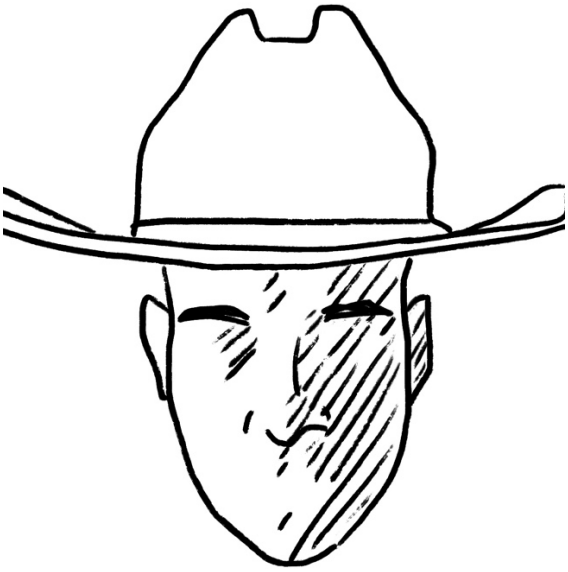

Image 2 'Shadows Sketch 1/2'

And that's what we would like to see:

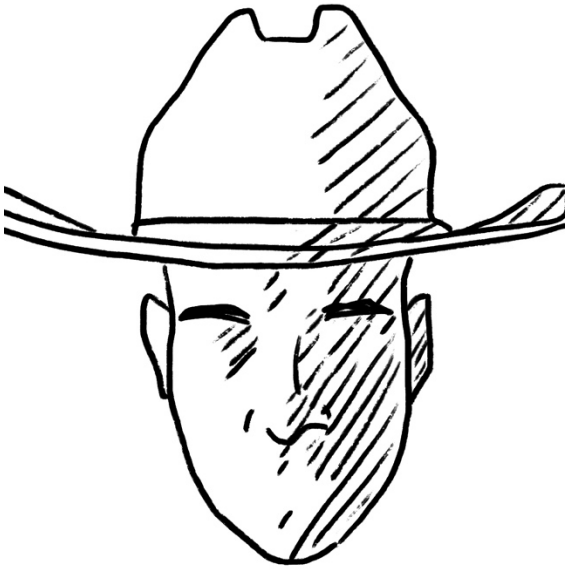

Image 3 'Shadows sketch 2/2'

Illuminating the virtual object with the light source located in the same direction would drastically improve the effect of "blending into real world". What else can be done? Well aside from the position of the light source it would be best also to add parameters that define intensity and color of the light source.

Predicting the illumination is not an easy task if you want to use mathematics and classical programming. To be sincere, I don't even imagine how to do that. Good thing we have a Machine Learning nowadays. So why is using Machine Learning for this problem will be a good example of when Machine Learning should be used? Well, as I said, because it will be very hard to come up with the solution in other way!

Machine Learning driven solution is a completely different approach to problem solving. We can describe classical problem-solving as coming up with a program that converts input into desired output. Machine Learning on other hand provides us a "program" itself if we have enough input and desired output. I think this illustration sums it up pretty good:

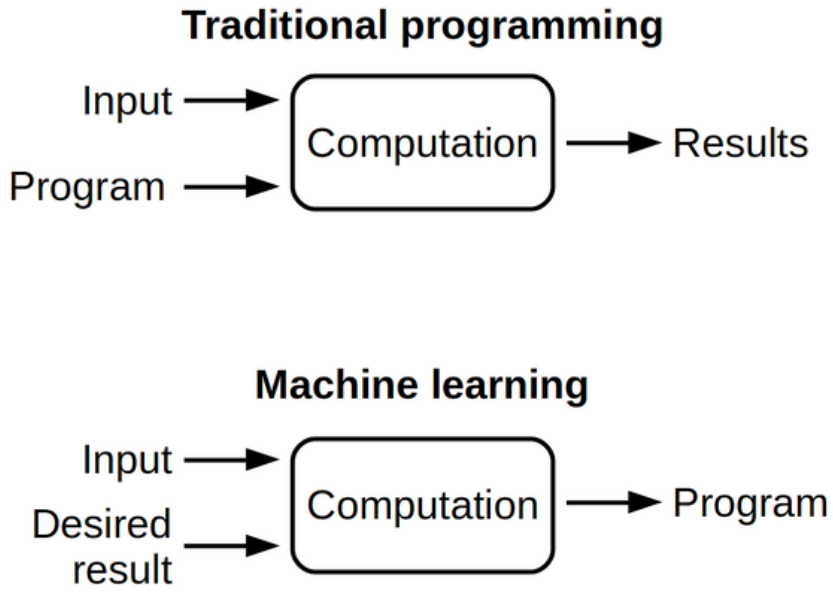

Image 3 'Traditional Programming vs ML'
Source

Of course, Machine Learning presents other problems. Like getting significant amount of data, feature extraction, deciding on model architecture and experimentation. But with a bit of creativity everything is double nowadays.

General Approach and Software

I'd like to give an insight on the general approach and software that I will be using in this project.

Firstly – getting the data. While it's desirable to get the real data, sometimes real data that you can acquire instantly doesn't really do the trick and needs some tweaking. But even more frequently the problem may be so specialized, that the data must be generated all by yourself. If the big company requires data, they may even come up with a temporal department that will spend a month or so on simply creating and gathering realistic data. Of course, realistic data has big advantages. It has diversity and generally more practical.

Another way round is to generate data and it's what I am doing. Do generate data I will setup a scene in Blender. I chose Blender as the main rendering software because it has Python scripting and is easy to learn.

Image 4 "Blender Logo"
Source

It's important to pass into the NN only carefully selected information. I propose to pass data based on the facial landmarks because we deal with the faces this time. Because I use simplified version of a problem, where the actor always faces camera in perpendicular manner, I go with simple facial mapping. Although in realistic scenario I would dedicate more time to come up with data which contains polar coordinates of the face relative to the camera or something similar.

I will be also using OpenCV library while dealing with images. Numpy library for data manipulation and Tensorboard for training process visualization.

Image 5 "OpenCV & Dlib"
Source

The other thing is to decide about Machine Learning API and NN Model itself. While I'd like to discuss NN Model in more detail later now I want to talk about API of my choice. I chose Keras for this task. While a year ago many could argue that PyTorch is better, because it's more intuitive and it executes in robust mode, today Keras built on top of Tensorflow 2.0 provides basically identical experience. Tensorflow 2 now executes in "Eager mode" by default. That means that you can define the variable as Pythonic variable and immediately use it, instead of building the model in "Graph mode". Keras also provides Sequential model, that allows straightforward appendage of layers, making job easy, while Functional API allows you to build more custom models. On top of that Keras is supported by handy "save/load" functionality and also Tensorboard, which allows easy logging and visualization.

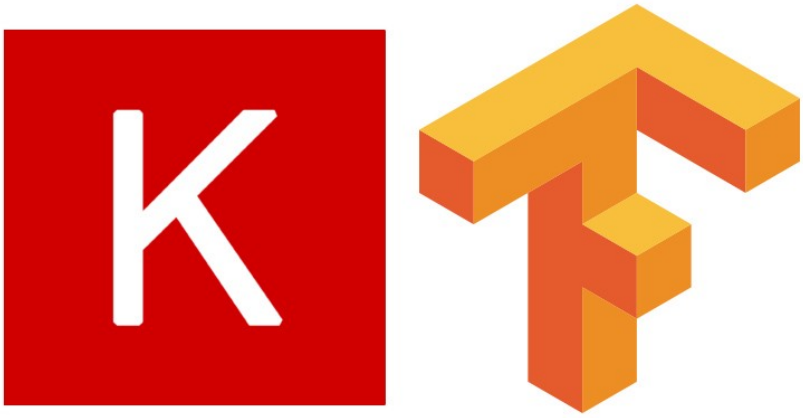

Image 6 "Tensorflow & Keras"
Source

As for the process of development I will be sticking to the default life cycle of Machine Learning Project:

Data Gathering → Data Preparation → Hypothesis & Modeling → Model Training → Model Testing → Corrections.

Scene Setup and Data Generation

As I stated previously, I will be generating data for this project. I am going to set up a simple 3D scene where I can cast light on the subject from different angles.

As for the actor model, turns out even with such big amount of Open-Source software, it's still not easy to find a high-quality model of human head. Turns out such model would cost between $40 and $150. Luckily one kind human being decided to scan his head and put it on the website for free. Big thanks to this person. Meet Ruslan:

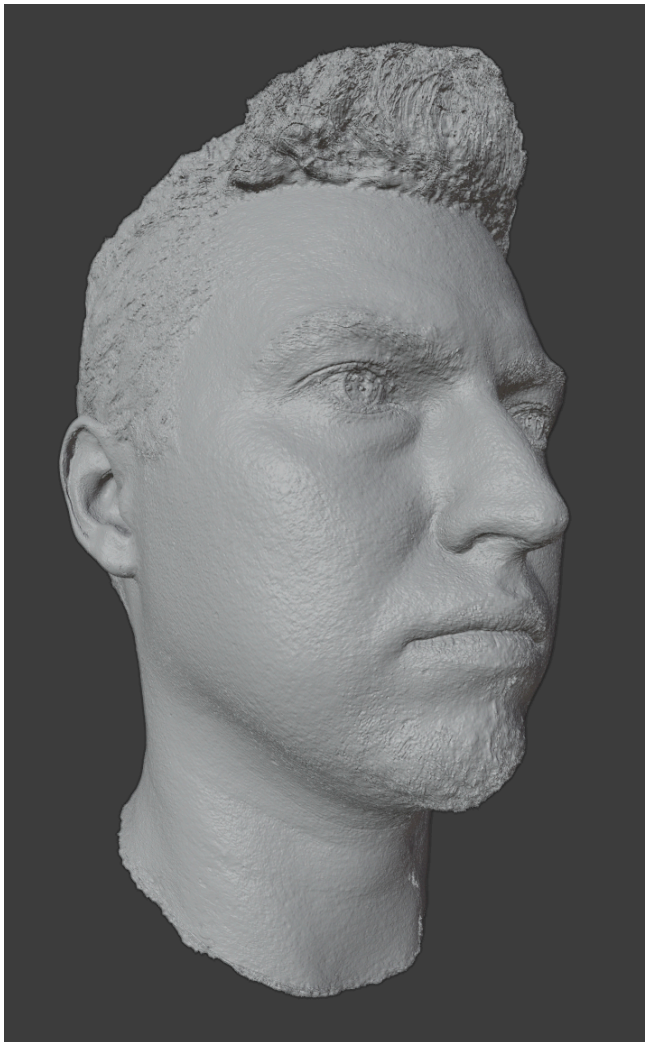

Image 7 "Ruslan Raw"

The model itself is very detailed and realistic. All the facial traits that define shadows and enlightened parts of the face are well defined (like nose, brows, cheeks, lips, forehead bone). Which is good.

With a little bit of Googling I was able to set up a simple scene consisting of Ruslan's head, point light source of 300W and camera looking directly at Ruslan.
This is how scene looks like:

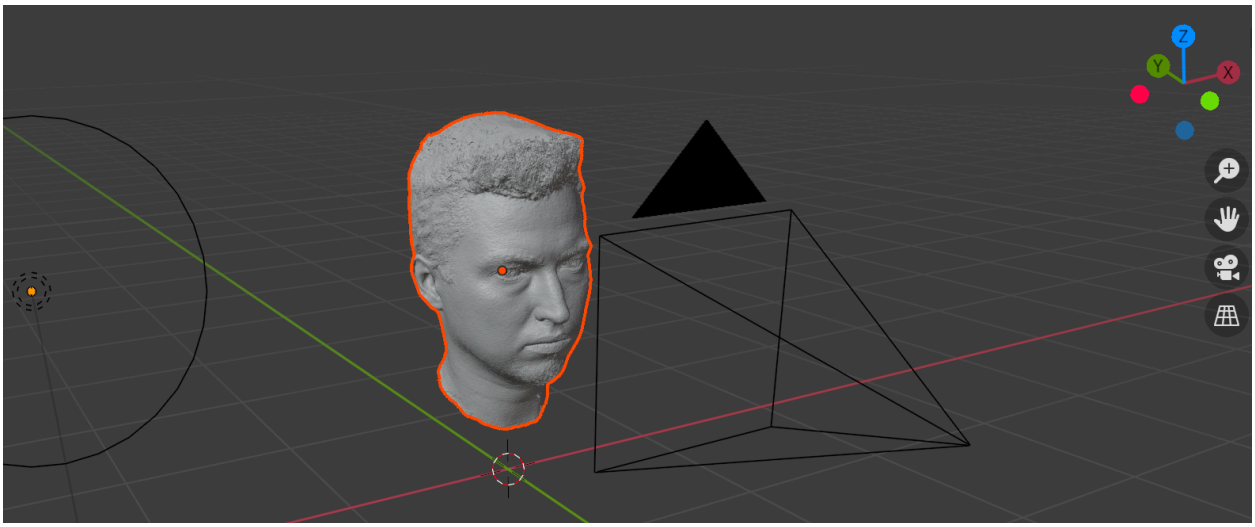

Image 8 "Ruslan Scene"

And that's how render looks like:

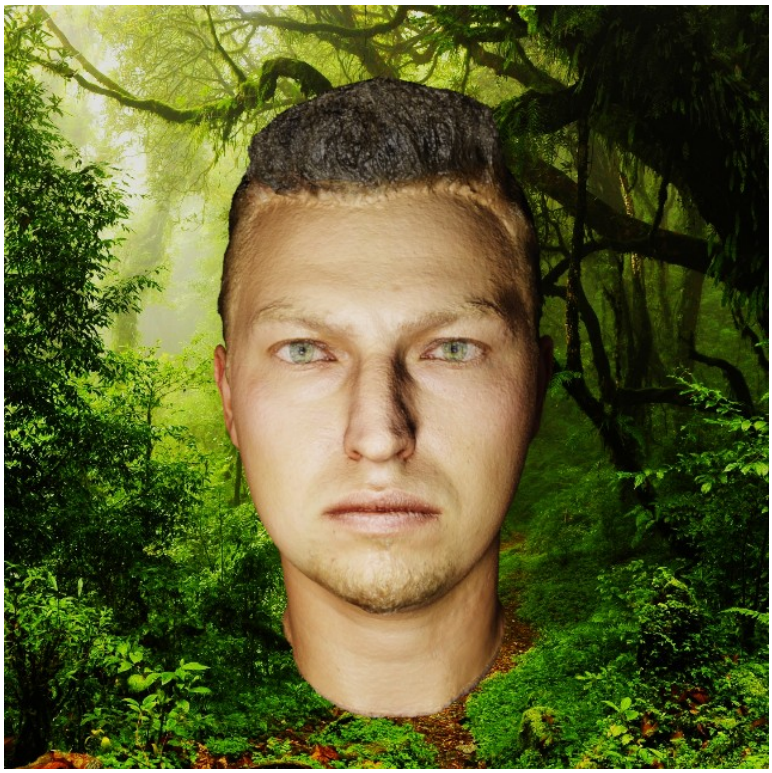

Image 9 "Ruslan Render"

As I already said, one of the main reasons for me to choose Blender as the rendering software is having Blender library for Python. It allowed me to generate a lot of images with different light positions. Here is the script that allows me to generate images while moving light source for defined step distance on all 3 axes. This way I am able to generate images with uniform distribution of light source positions.

```
DATA_BATCH_NAME = "ruslan 300 W uniform 320x320"
RESOLUTION_X = 320
RESOLUTION_Y = 320

scene = bpy.context.scene

scene.render.resolution_x = RESOLUTION_X
scene.render.resolution_y = RESOLUTION_Y
scene.render.resolution_percentage = 100

if not 'ML_LIGHT_DIR' in os.environ:
    os.system("say %s" % ("Create root environment variable"))
    raise Exception("Create root environment variable")

root = os.environ['ML_LIGHT_DIR']

light = bpy.context.scene.objects.get('Light01')

step_x = 0.3
step_y = 0.3
step_z = -0.3

initial_x = -3
initial_y = -3.6
initial_z = 3.6
id = 0

x = initial_x
while(x <= 3):
    y = initial_y
    while(y <= -0.3):
        z = initial_z
        while(z >= 0.3):

            light.location.x = x
            light.location.y = y
            light.location.z = z

            timestamp = time.time()
            bpy.context.scene.render.filepath = root +
            + 'training/blender/render/{}/{} {} {} {}'.format(DATA_BATCH_NAME, x, y, z, timestamp)
            bpy.ops.render.render(write_still=True)

            z += step_z
        y += step_y
    x += step_x
```

Besides I wrote have a script that puts light in a random position each time.
With all that I generated 3024 of uniformly distributed light positions and a 400 of random ones.
Total count of data samples = 3424.

Feature Extraction

There are two approaches to the feature extraction. First – to let Neural Network decide about the features on its own. Using Convolutional Neural Network would do the trick. However, Convolutional Neural Network has a huge giveaway. It's much more computationally expensive to support this type on network.

My view of this product is that algorithm will be fast enough to execute to be feasible for it to deploy on mobile device. That's why I propose another way to extract information about face illumination. For that I will be using Dlib library that contains implementation of the One Millisecond Face Alignment with an Ensemble of Regression Trees (Source) paper by Kazemi and Sullivan (2014).

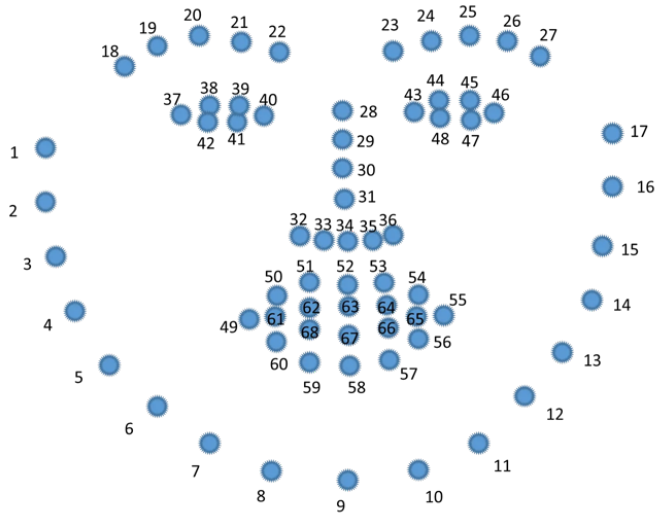

Image 11 "Dlib Landmarks"
Source

The process of extracting landmarks consists of two steps:

1. Finding faces in the image.
2. Landmark position estimation.

There are several variants for landmark estimations out there. You can even train your own. I am using standard 68 landmark predictor:

```
9
10  # detector to detect faces
11  _detector = dlib.get_frontal_face_detector()
12  # predictor to predict landmarks
13  _predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")
14
```

```
42  def place_landmarks_single(detector, predictor, img):
43      img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
44      faces = detector(img_gray, 1)
45      if(len(faces) != 0):
46          landmarks = predictor(img_gray, faces[0])
47          landmarks = face_utils.shape_to_np(landmarks)
48          return landmarks, True
49      return None, False
```

If there is a face present on the image – I landmarks will be returned. Otherwise, function returns False. This is important in order to not get any NaN data.

If we take a look at landmarks – it becomes obvious that this information will not be enough. They are all situated on the edges of the face.

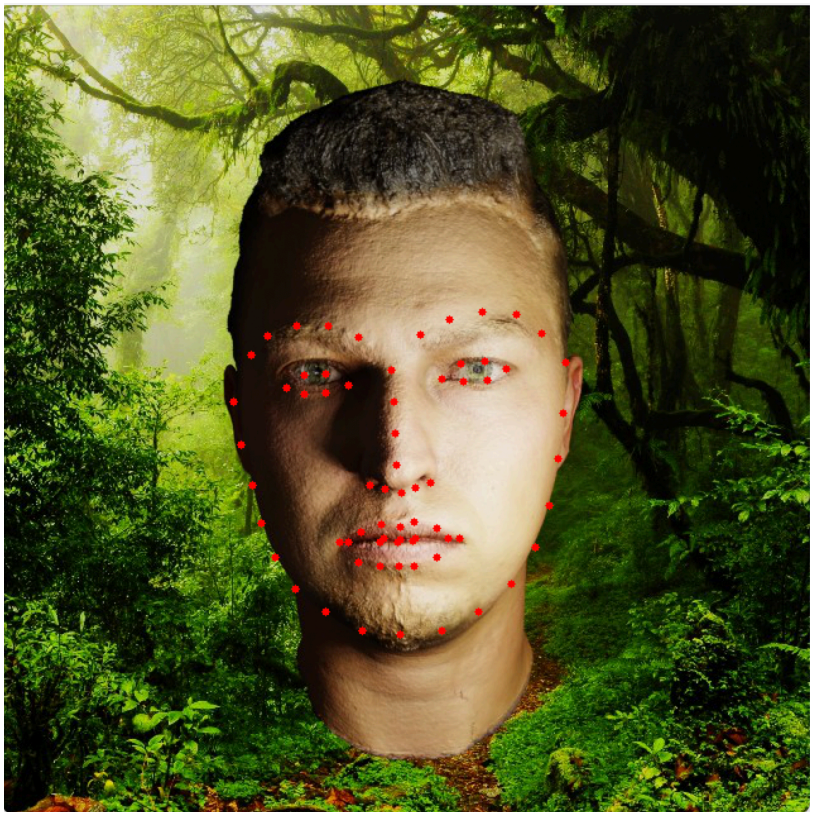

Image 12 “Ruslan Dlib Landmarks”

All the information is actually located in the area of the face, not on the perimeter. For that I suggest deriving additional landmarks based on the present ones. All of the additional landmarks are hardcoded by hand.

This is the part of the *calc_additional_points()* function in which I calculate points around the mouth:

```
    # Mouth parts
    right_beard0 = dot_median(landmarks[7], landmarks[58])
    right_beard1 = dot_median(landmarks[7], landmarks[58])
    right_beard2 = dot_median(landmarks[6], landmarks[59])
    right_beard3 = dot_median(landmarks[5], landmarks[59])
    left_beard0 = dot_median(landmarks[9], landmarks[56])
    left_beard1 = dot_median(landmarks[9], landmarks[56])
    left_beard2 = dot_median(landmarks[10], landmarks[55])
    left_beard3 = dot_median(landmarks[11], landmarks[55])
    right_lip1 = dot_median(landmarks[31], landmarks[49])
    right_lip0 = dot_median(right_cheek3, right_lip1)
    right_lip2 = dot_median(landmarks[32], landmarks[50])
    mid_lip = dot_median(landmarks[33], landmarks[51])
    left_lip1 = dot_median(landmarks[35], landmarks[53])
    left_lip0 = dot_median(left_cheek3, left_lip1)
    left_lip2 = dot_median(landmarks[34], landmarks[52])
    mid_beard = dot_median(landmarks[8], landmarks[57])
    face_mouth = np.array(
        [mid_beard, right_beard0, right_beard1, right_beard2,
         right_beard3, left_beard0, left_beard1, left_beard2,
         left_beard3, right_lip0, right_lip1, right_lip2,
         mid_lip, left_lip0, left_lip1, left_lip2])
    points = np.append(points, face_mouth, axis=0)
```

Having calculated additional points, they are now situated on the key points of the face *(Image 13)*. The skin at these points will be either lighter or darker depending on the position of the light source thanks to the topology of the face.

The next step of the feature extraction is to sample the color of the skin in the area of so-called light-points. There is no need to store RGB value of the skincolor, because the skin color is the same for the whole face. Thus I first convert the image to Grayscale. To sample the color in the area of the point I am using OpenCV library. In order to calulate the the average color around the point you have to:

1. Create the mask of the same shape as the image with the circle in the area that is needed to be sampled. The mask itself represents the black image of RGB(0, 0, 0) with the white circle in the area RGB(255, 255, 255).
2. Apply *mean()* function and pass the mask mask.

```
def extract_colors(lightpoints, img, radius):
    colors = np.empty((0, 3), dtype=int)
    for lightpoint in lightpoints:
        circle_img = np.zeros((img.shape[0], img.shape[1]), np.uint8)
        cv2.circle(circle_img, (lightpoint[0], lightpoint[1]), radius, (255, 255, 255), -1)
        color = cv2.mean(img, mask=circle_img)[::-1]
        color = np.array([[color[1], color[2], color[3]]])
        colors = np.append(colors, color, axis=0)
    return colors
```

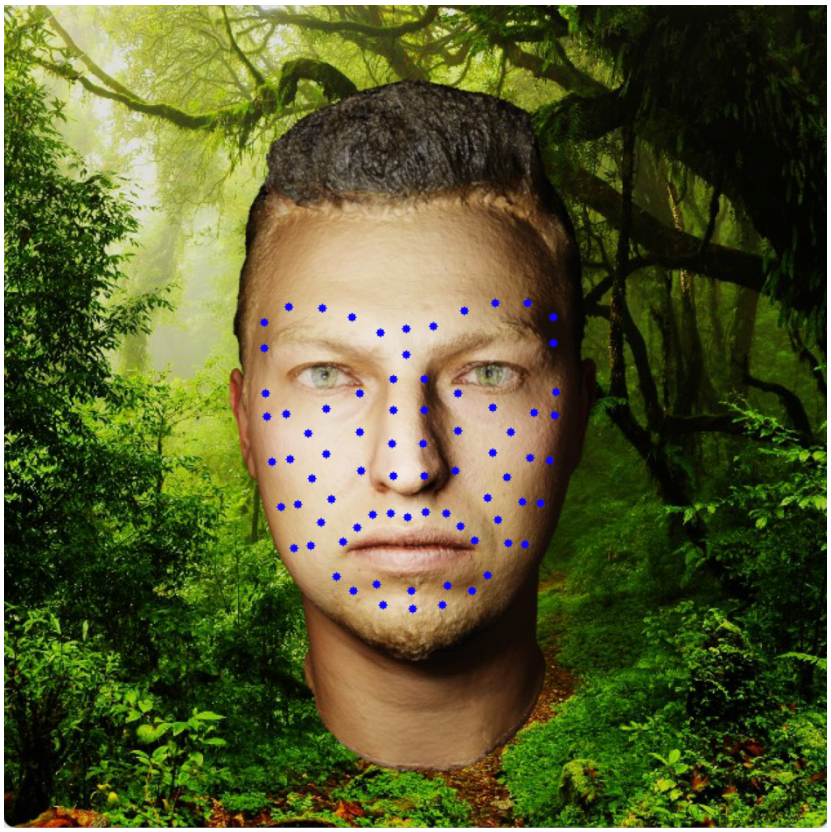

Image 12 “Derived Light-points”

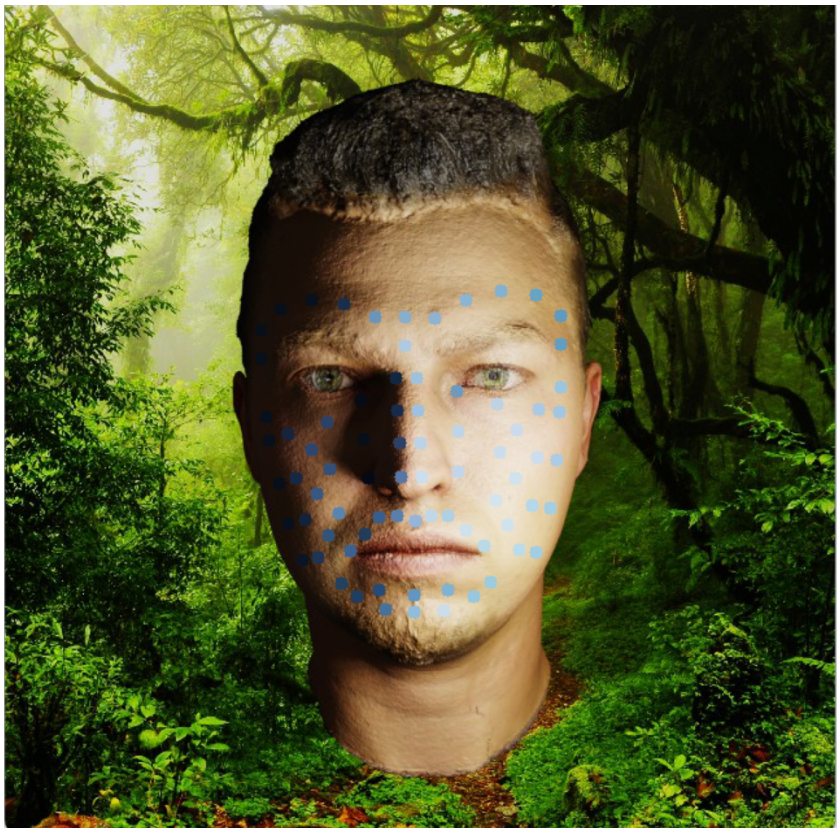

Image 13 “Color Areas in Light-points”

On *Image 13* I visualized the average colors in the areas of the light-points. I decided to paint them in blue color so that they will be more apparent. As you can see, in the very light areas circles are almost blending with Ruslan's skin. In the dark areas on other hand circles are almost invisible. With that I would agree that extracted data represents the whole picture pretty well.

To conclude, these are the steps for feature extraction for every image in my project:

1. Find face.
2. Estimate landmark position for a face.
3. Calculate light-points basing on the landmarks
4. Convert image to RGB
5. For each light-point calculate the average color in the area of a radius of 3 pixels.

All data related functionalities are located in files *landmark_utils.py* and *data_manip.py*.

Model Selection, Training and Testing

When working with visual data as an input usually there are two choices. Either you use Convolutional Neural Network that will "find" features on itself, or you have to select extract features in another way and pass them into Neural Network.

For the sake of computational efficiency, the decision was made to go with the second option. When it comes to decision making during model selection, you have to choose between different parameters, hyperparameters and model architecture.

I already referred to one of my favorite descriptions on Machine Learning, when it replaces the part of program composition by having significant amount of input and desired output. It's only logical that the reasoning behind Model should start by looking and the form of input and output. To recap, the single input entity for this Model will consist of 79 values that represent the average color in the area of light-points. As for the output my Model should estimate the position of the light source basing its decision on these color probes.

With the input layer it's pretty straightforward, it should be the layer consisting of 79 weights. As for the output, it should represent values of 3D space (x, y , z) which will be passed to the render engine to obtain a virtual object with (hopefully) same illumination. As for the hidden layers there can be a few options and it's open to experimentation. But usually for such problem you want to have 1 or 2 hidden layers.

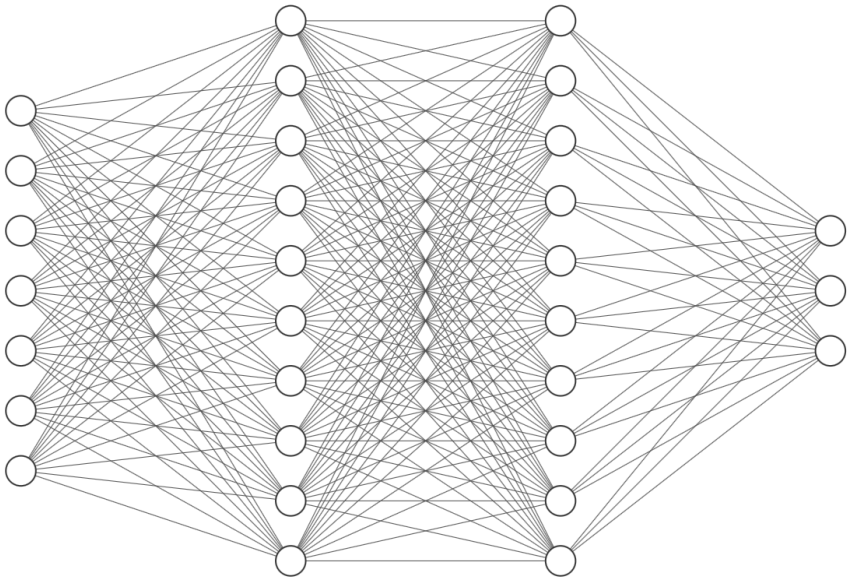

Image 14 "Fully Connected"

For the purpose of flexibility and being able to perform more experiment faster I constructed a dedicated Class. This allows to instantiate a Model with hyperparameters of your choice. I suggest going through Model class.

```
6   class FNN:
7       def __init__(self, input_shape=0, layer_units=[0], output_layer=0, load_path = '', learning_rate = 0.01):
8           print(_bcolors.BOLD + "input shape of a model is: {}".format(input_shape) + _bcolors.ENDC)
9           if load_path != '':
10              self.model = keras.models.load_model(load_path)
11              print(_bcolors.OKGREEN + 'loaded model: {}'.format(load_path) + _bcolors.ENDC)
12          else:
13              self.model = keras.models.Sequential()
14              self.model.add(keras.layers.Input(shape=(input_shape)))
15              for unit in layer_units:
16                  ...
19                  self.model.add(keras.layers.Dense(unit, activation = 'relu'))
20              ...
24              self.model.add(keras.layers.Dense(output_layer))
25              self.model.compile(optimizer = 'adam',
26                                 loss = 'mean_squared_error', metrics = ['accuracy'], learning_rate=learning_rate)
```

When creating Fully Connected Neural Network with my class you have an option of choosing input shape, number of hidden layers and variables amount, output layer and learning rate. If you specify *load_path* parameter – then previously model will be loaded. I'd also like to point out that I use Sequential API in Keras. With it you just "append" layers one to another having Fully Connected Neural Network in the end.

```
31      def info(self):
32          self.model.summary()
33
34      def train(self, X_train, X_test, y_train, y_test, epochs):
35          self.model.fit(X_train, y_train, epochs=epochs, batch_size=8, validation_data=(X_test, y_test))
36
37      def eval(self, test_data, test_labels):
38          self.model.evaluate(test_data, test_labels)
39
40      def save(self, filename):
41          self.model.save(filename)
42
43      def predict_single(self, data, label):
44          predictions = self.model.predict(data)
45          prediction = predictions[0]
46          prediction = np.round(prediction, 3)
47          label = label.astype(float)
48          # label = np.round(label, 3)
49          print(_bcolors.BOLD + "%12s" % "RES: " + _bcolors.ENDC + "X: {:.3f} Y: {:.3f} Z: {:.3f}".format(predictio
50          print(_bcolors.BOLD + "%12s" %  "EXPECTED: " + _bcolors.ENDC + "X: {:.3f} Y: {:.3f} Z: {:.3f}".format(lab
51          print(_bcolors.BOLD + "%12s" % "DIFF: " + _bcolors.ENDC + "X: {:.3f} Y: {:.3f} Z: {:.3f}".format(label[0]
```

Other class methods provide information, training, evaluation and saving of a model. There is also *predict_single()* method allows to test a single image.

As for the Model parameters, they are hardcoded. For a optimization algorithm I choose Adam optimizer. Adam is used for gradient descent in Deep Learning and combines AdaGrad and RMSProp algorithms. The advantage of Adam is that it handles sparse and noisy data well.

For the activation function in hidden layers I use ReLU activation. Recently ReLU became much more popular than Sigmoid activation. It's all due to the fact that ReLU doesn't have a problem of a vanishing ingredients which occurs with Deep Learning Networks due to the

depth of the Model. In my problem I don't have a very deep Network, but ReLU shown better results.

$$\sigma(x) = \begin{cases} max(0, x) & , x >= 0 \\ 0 & , x < 0 \end{cases}$$

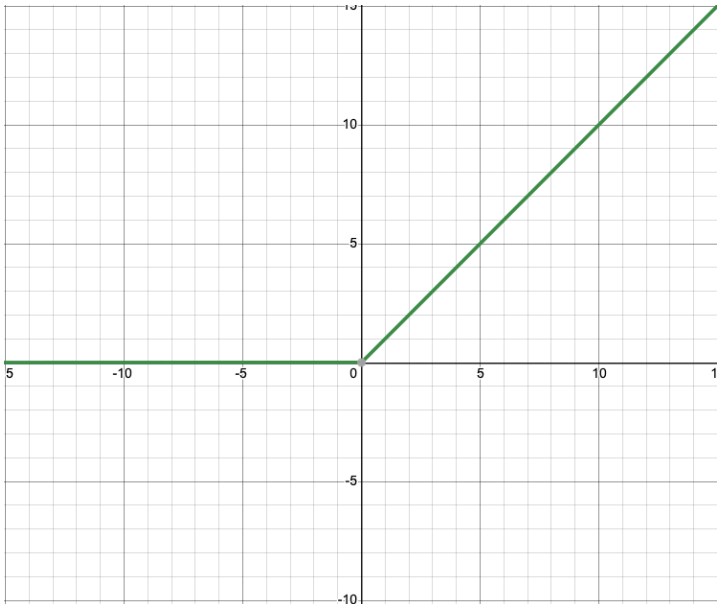

Image 15 "ReLU"
Source

Now because we have a problem where Model should output values that directly represent dimensions that means that it's a regression problem. Because of that I don't have an activation function in the last layer, it just outputs 3 numbers. Error is calculated in somewhat "classic" way, simply Means Squared Error.

$$\text{MSE} = \underbrace{\frac{1}{n}\sum_{i=1}^{n}}_{\text{test set}} \underbrace{(y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i)^2}_{\text{actual value}}$$

Before Training and Testing I would like to say a word on how Data is handled in my project. After using *import_raw_data()* that takes one argument and runs each image through feature extraction process you end up having a pretty large batch of data. The whole process of extracting features may take different amounts of time depending on number of images. For me it took about ~25-30 seconds to process 3000 images that way. For that I use a very handy Numpy functions *Numpy.save()* and *Numpy.load()*.

```
batch_name = "uniform320_GRAY_Feb18"
colors, labels, lightpoints = data_manip.load_np_data("{}_data.npy".format(batch_name),
                                                      "{}_labels.npy".format(batch_name),
                                                      "{}_lightpoints.npy".format(batch_name))

```

After you load data, you usually want to shuffle it and split in Train and Test sets. It's easily done with Scikit module:

```
X_train, X_test, y_train, y_test = train_test_split(colors, labels,
                                                    test_size=0.3, shuffle=True)
```

To instantiate and train and save model:

```
model = model_FNN_regression.FNN(input_shape=colors.shape[1],
                                 layer_units=np.array([128, 64]),
                                 output_layer=3, learning_rate=0.001)
model.info()
model.train(X_train, X_test, y_train, y_test, 20)
model.save("model_name")
```

For the given model 20 epoch training results are:

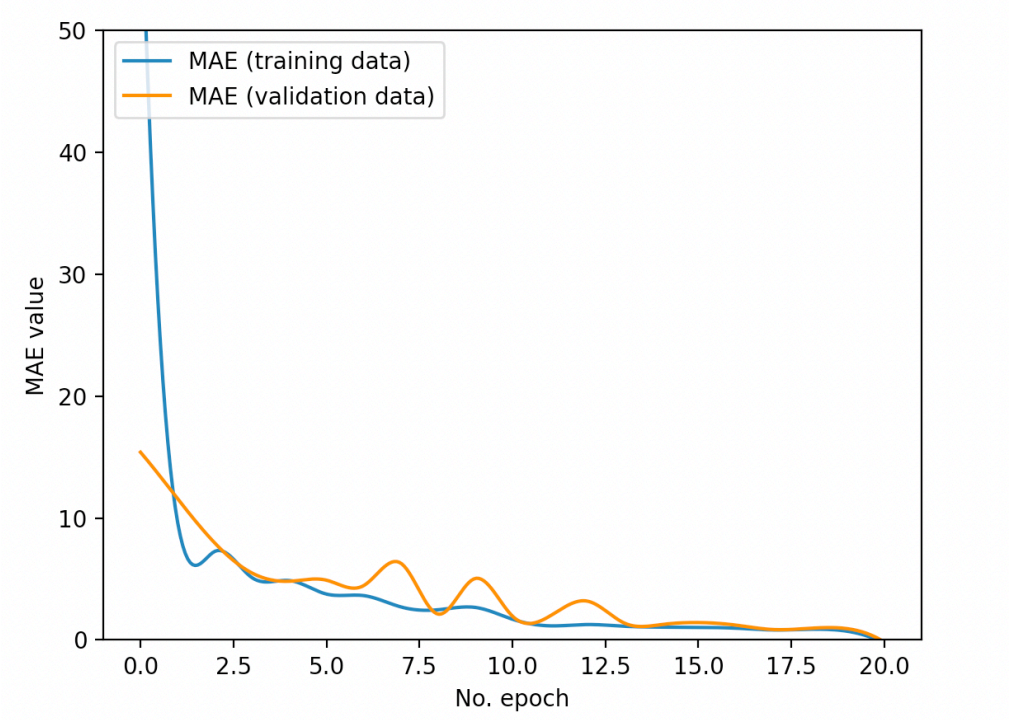

Image 16 "FNN model training"

Training process looks legit, and the curve is more or less consistent. What's important is that loss decreases, where *training_loss = 1.0138* and *validation_loss = 0.9179.*

However, when we take a “hands on” approach to testing our Model, we find out that truthfulness of the prediction is completely off. To test the model I use *pedict_single()* function. In it I use *Keras.model.predict()* that outputs predictions for passed data. Passing a single image outputs the values of 3D space in which light source is supposedly located. Then function displays expected result, actual result and difference between them.

In the *interfaces.py* you can find two functions that provide ways to help test the model manually. First function *test_one_by_one()* allows you to provide file paths of images one by one that will be passed to *model.predict_single()* until user decides to quit the interface.

```
19  def test_one_by_one(model):
20      print("Provide path to make a prediction. Input q to quit.")
21      while True:
22          image_path = input("Path: ")
23          if(image_path == 'q' or image_path == 'quit'):
24              break
25          else:
26              if(os.path.exists(image_path)):
27                  print(image_path)
28                  colors, label, _ = data_manip.prep_single(image_path)
29                  model.predict_single_compare(colors, label)
30              else:
31                  print(_bcolors.FAIL +
32                        "File doesn't exist. Provide correct path."
33                        + _bcolors.ENDC)
```

Here are the results for a few randomly chosen images:

```
/Users/romandubovyi/ml_jedis/ml_light/training/blender/render/ruslan point 300 W
031541026 2.4602249361241513 1601468698.512081.jpg
       RES: X: 2.913 Y: -0.631 Z: 1.984
  EXPECTED: X: 0.749 Y: -1.875 Z: 2.460
      DIFF: X: -2.164 Y: -1.244 Z: 0.476

/Users/romandubovyi/ml_jedis/ml_light/training/blender/render/ruslan point 300 W
830365932 2.0517425159807368 1601468791.157402.jpg
       RES: X: 1.116 Y: -2.699 Z: 1.816
  EXPECTED: X: 0.971 Y: -1.743 Z: 2.052
      DIFF: X: -0.145 Y: 0.956 Z: 0.236

/Users/romandubovyi/ml_jedis/ml_light/training/blender/render/ruslan point 300 W
265182838 2.796550724375484 1601468869.8730931.jpg
       RES: X: 1.627 Y: -3.991 Z: -1.683
  EXPECTED: X: -0.708 Y: -1.320 Z: 2.797
      DIFF: X: -2.335 Y: 2.671 Z: 4.480
```

I should mention that ranges for coordinates of light sources can vary in such way:
For X coordinate values are [-3; 3].
For Y coordinate values are [-3.6; -0.3]
For Z coordinate values are [3.6; 0.3]

Looking at the results we can see that model has a pretty big mismatch with the expectations. For example, in second test sample Y value differs by 0.9 which is significant for a value that ranges in [-3.6; -0.3]. In fact, it's almost 30% of a mismatch. For other two samples error is very huge.

The question is why the actual error and training loss are so different? When it came to calculating loss with Mean Squared Error the value for validation set was ~0.9. While if we calculate mean squared error for these three samples it will actually be a whopping 11,4. Definitely something is wrong in the model architecture.

With a bit of reasoning the problem becomes pretty obvious actually. What's wrong is that for prediction of 3 values we calculate a single value of loss and a single gradient backpropagation. And what's "happening inside" is that, for example, while making correction for a prediction of X value we actually interfere with the weights responsible for prediction of Y and Z values. While in reality all 3 dimensions must be independent.

The correct logic for such a problem would be if basing on 79 light-points each optimization would fit what's best for every dimension separately. This means that each output value should have its own hidden layers. You can say that we will have three subnets running in parallel having one common input layer. Visualization of this Neural Network would look like this:

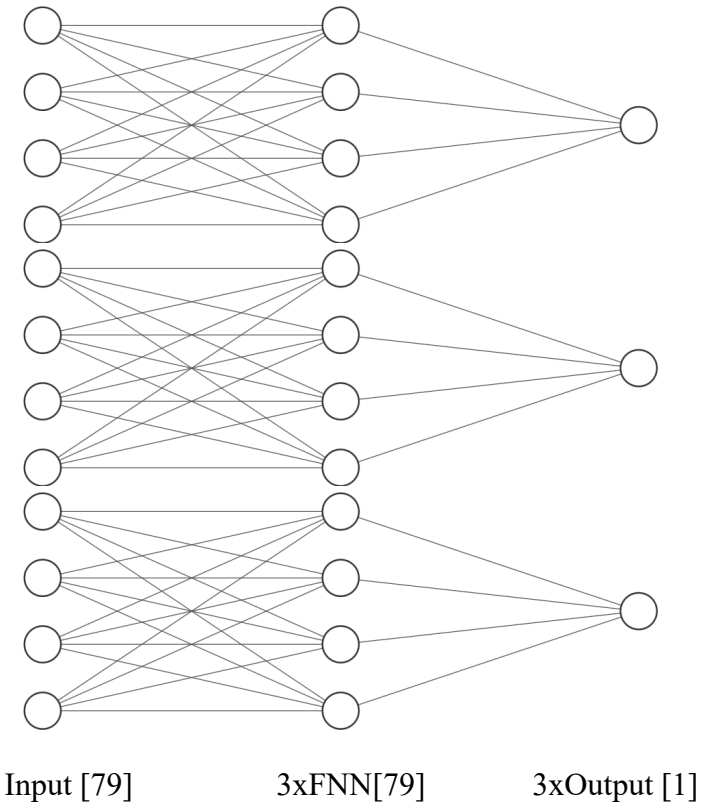

Image 17 "Parallel regression"

In order to achieve this effect, I used Keras Functional API. Previously I was using Sequential modeling which allows straightforward appendage of layers. Now in Functional API

you should explicitly specify the input layer when instantiating another. To do so, you set the input layer in parenthesis after layer initialization.

```
class Parallel:
    def __init__(self, input_shape=0, hidden_units=0, load_path = '',
                 learning_rate = 0.01):
        print(_bcolors.BOLD + "input shape of a model is: "
                              "{}".format(input_shape) + _bcolors.ENDC)
        if load_path != '':
            self.model = keras.models.load_model(load_path)
            print(_bcolors.OKGREEN + 'loaded model: {}'.format(load_path)
                  + _bcolors.ENDC)
        else:
            input_layer = keras.layers.Input(shape=(input_shape))

            hidden_layer_x = keras.layers.Dense(hidden_units,
                                                activation = 'relu')(input_layer)
            out_layer_x = keras.layers.Dense(1)(hidden_layer_x)

            hidden_layer_y = keras.layers.Dense(hidden_units,
                                                activation='relu')(input_layer)
            out_layer_y = keras.layers.Dense(1)(hidden_layer_y)

            hidden_layer_z = keras.layers.Dense(hidden_units,
                                                activation='relu')(input_layer)
            out_layer_z = keras.layers.Dense(1)(hidden_layer_z)

            merge_layer = keras.layers.concatenate([out_layer_x,
                                                    out_layer_y, out_layer_z])
            out_layer_final = keras.layers.Dense(3)(merge_layer)

            self.model = keras.Model(inputs=[input_layer],
                                     outputs=[out_layer_final])
            self.model.compile(loss='mae', optimizer='adam')
```

Training and evaluation for this one looks the same. The results are:

```
/Users/romandubovyi/ml_jedis/ml_light/training/blender/render/ruslan point 300 W
389314925 1.891465524521254 1601468910.551051.jpg
       RES: X: 1.850 Y: -2.390 Z: 2.136
  EXPECTED: X: 1.174 Y: -1.894 Z: 1.891
      DIFF: X: -0.676 Y: 0.496 Z: -0.245

/Users/romandubovyi/ml_jedis/ml_light/training/blender/render/ruslan point 80 W
4730848391 0.8519784335585006 1600097805.900328.jpg
       RES: X: 0.818 Y: -0.119 Z: 1.204
  EXPECTED: X: -0.049 Y: -0.634 Z: 0.852
      DIFF: X: -0.867 Y: -0.515 Z: -0.352
```

```
/Users/romandubovyi/ml_jedis/ml_light/training/blender/render/ruslan 300 W unifc
000000000000008 2.100000000000001 1613593909.872468.jpg
       RES: X: 0.601 Y: -0.432 Z: 1.910
  EXPECTED: X: 0.300 Y: -1.200 Z: 2.100
      DIFF: X: -0.301 Y: -0.768 Z: 0.190
```

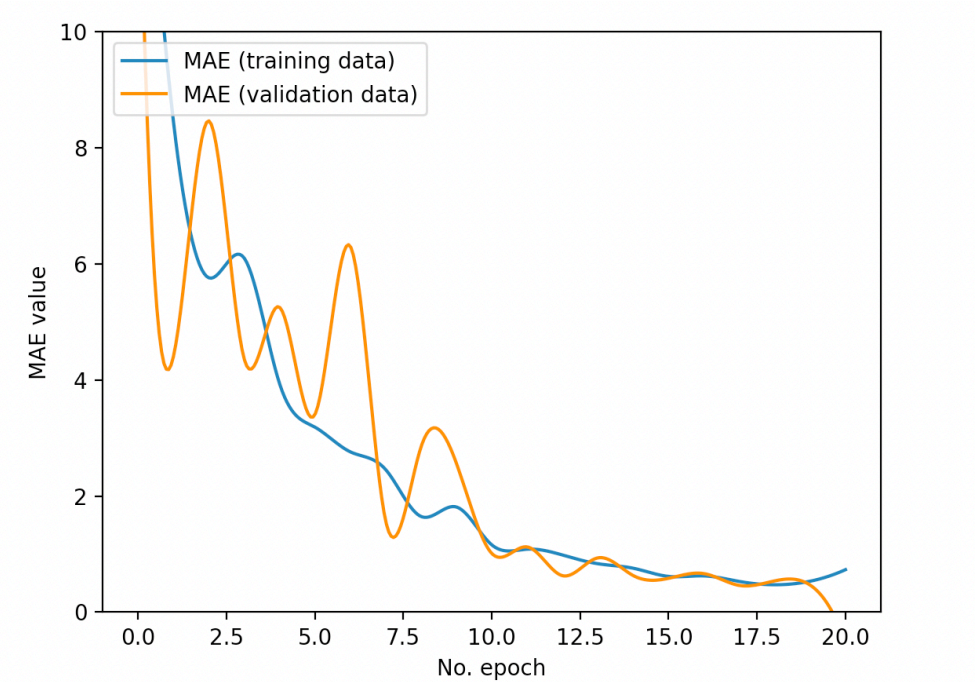

Image 18 “Parallel regression training”

Training curves for loss show better results than the previous Model. The lowest value for *validation_loss = 0.45* and *training_loss = 0.5.* As for the model testing with one by one we can see that the difference between desired and actual results is much smaller. Indeed, training each “dimension” separately produces much better predictions.

And the last but not least important measure in this Model – is execution time of prediction. I am measuring elapsed time simply by using Python’s *time.time()*.

```
64      def predict_single(self, data):
65          start = time.time()
66          prediction = self.model.predict(data)[0]
67          prediction = np.round(prediction, 5)
68          end = time.time()
69          print("Prediction time elapsed: " + _bcolors.BOLD + str(end-start) + _bcolors.ENDC)
70          return prediction
```

For 3 samples the measurements of time elapsed during Model prediction are:

```
dPrediction time elapsed: 0.14964795112609863
Blender 2.83.0 (hash 211b6c29f771 built 2020-06-03 15:05:55)
```

```
Prediction time elapsed: 0.1431736946105957
Blender 2.83.0 (hash 211b6c29f771 built 2020-06-03 15:05:55)

Prediction time elapsed: 0.15618896484375
Blender 2.83.0 (hash 211b6c29f771 built 2020-06-03 15:05:55)
```

Which is always less than quarter of a second. By the way Tensor operations are performed on the Inter processor, not on Nvidia architecture. This means, that on mobile device such model would perform a bit slower, but definitely in reasonable execution times. My guess is that this model would make a prediction in 0.3 of a second on last version of Apple mobile processor.

Showing Results

Now it's time to test the model in more intuitive and comprehensive way. What I propose for testing is basically to:

1. Render a new Ruslan image with random light source location.
2. Predict light source position
3. Render hat on Ruslan illuminated with separate light based on prediction.

Now there was a problem. In order to achieve intuitive showcase, I needed to basically illuminate Ruslan with one light source and Ruslan's hat with another one. They also have not to overlap. While using blender had its advantages, lack of so called "light linking". Because of that everything had to be ordered using render layers.

In order to achieve the effect, hat and face had to be separated into layers and then composed into one image using Alpha Over:

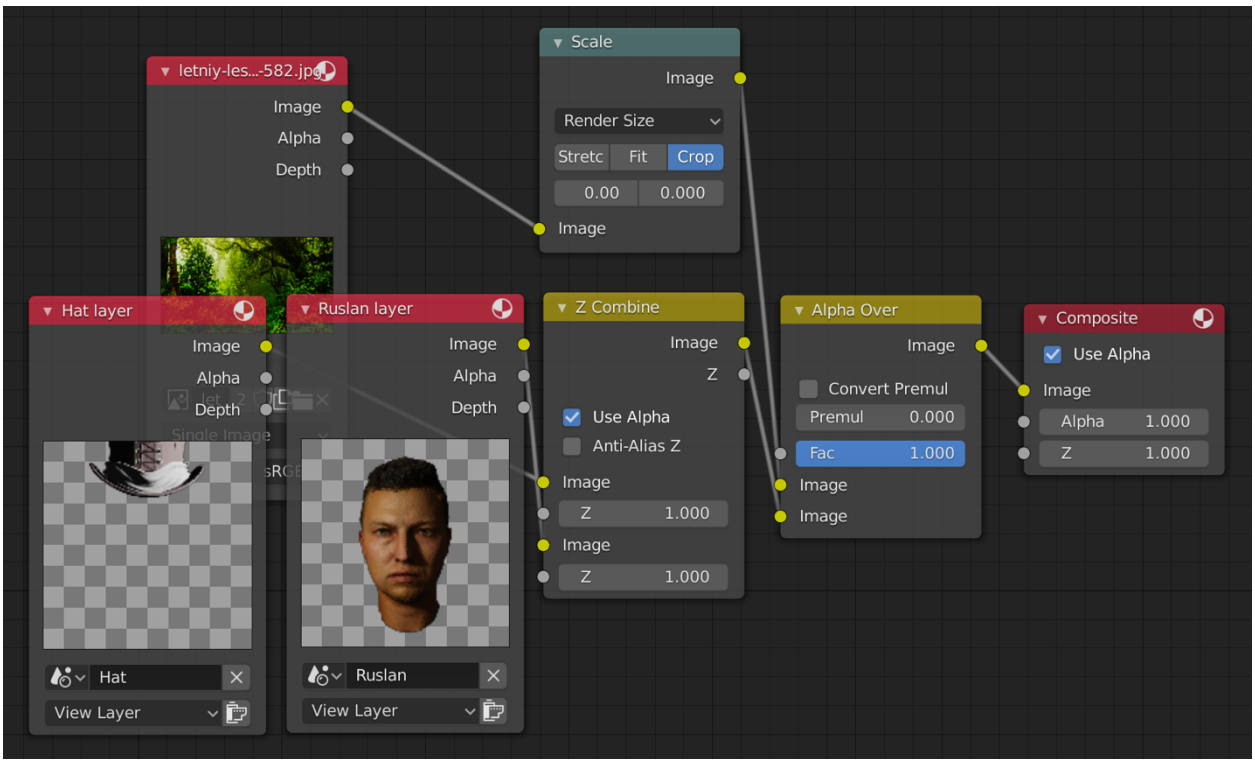

Image 19 "Scene Composition"

As you can see, Ruslan and his hat have different illumination on them. For the purpose of visualization, I will be rendering image with random light positions using 0 light power for the hat. This way hat model will just appear black and won't be distracting and misleading.

Blender supports Python scripting, but unfortunately, the same modules that are required for support of BPY (blender python) package can't support Tensorflow due to the version conflicts.

For this purpose, I implemented interface that executes Python scripts using Blender externally. The code is pretty straightforward, execute script that generates random image, open that image, predict and generate output with previously generated random Ruslan's light position and predicted Hat's light position. All parameters that will be used inside Blender are passed after "--" and are parsed with another script. I also use "--background" flag that disables Blender UI speeding up the process.

```
"""Render-Predict-Render"""
def rpr_random(model):
    # Render and show test subject
    file_name = 'before'
    render_path = root + 'training/blender/render_single/layers_upd/'
    SCENE = root + 'training/blender/scenes/ruslan-with-hat-layers.blend'
    SCRIPT = root + 'training/python/scripts/layers_before_random.py'
    exec = os.system(
        'blender --background {} '.format(SCENE) +
        '--python {} -- {} '.format(SCRIPT, file_name))

    newest = get_newest(render_path)
    img = cv2.imread(newest)
    cv2.imshow('before', img)
    cv2.waitKey(0)

    colors, label, _ = data_manip.prep_single(newest)
    prediction = model.predict_single(colors)

    # Render and show prediction

    file_name = 'after'
    SCENE = root + 'training/blender/scenes/ruslan-with-hat-layers.blend'
    SCRIPT = root + 'training/python/scripts/layers_after.py'
    exec = os.system(
        'blender --background {} '.format(SCENE) +
        '--python {} -- {} {} {} {} {} {} {} '
        .format(SCRIPT, file_name, label[0], label[1], label[2], prediction[0],
                                                    prediction[1], prediction[2]))
    newest = get_newest(render_path)
    img = cv2.imread(newest)
    cv2.imshow('after', img)
    cv2.waitKey(0)
```

Now I'd like to showcase a few samples. Image on the left is rendered with random parameters for light position, illumination on Hat is disabled on that one. Next to it will be the image where Ruslan is illuminated with random light and Hat is illuminated with predicted light.

Image 20 “Showcase 1”

Image 21 “Showcase 2”

Looking at those two examples it almost seems like if both objects were illuminated by the same source of light. Even though error may seem pretty significant in numbers, in the actual render the difference is so subtle that it can’t be comprehended. At least by my eyes. Something on these images looks odd, and that’s the absence of the shadow from the hat on Ruslan’s face. That’s because being in different layers these objects “don’t know anything of each other”.

These were pretty standard examples. It is always recommended to test your solution on extreme examples as well. They usually instantly show if something is wrong with the model. I will be showing only image with predictions from now on.

Image 22 “Special case 1”

On this example it’s clearly visible that face is illuminated from above. Model actually performed well. You can’t see hat casting shadow on face because of previously discussed reasons. But you can see that uneven cloth part of the hat casts shadow on itself as if the source of the light is on top.

Image 23 “Special case 2”

Another extreme example here. As we can see the source of light is located very much on far-right side (from out perspective). Light source is also very close to the face. Yes, power of illumination of the object in my case is determined by distance to the light source. The power of light stays the same all the time. And in this case prediction was extremely accurate. I am sure that if I’d illuminate the hat from the same light source that illuminates Ruslan and then compare that render to prediction – I could’ve seen difference. But speaking from the practical point of view, there is almost no difference.

Image 24 “Special case 3”

This is actually one of the few examples where you can see inconsistency in prediction. If we take a look at the shadows on the face and compare them to shadows on hat – we can see that they are located on different sides. I highlighted them, but on clean image they are not very apparent, because the overall luminosity is pretty much similar.

In overall I am very satisfied with the result. However, there is much more to be done for the technology to be usable in real world applications. I’d like to discuss that in the next section.

Thoughts on Improvements

In the final chapter I'd like to discuss potential improvements of my Model and how it can be applied to real world solutions.

One can say that this solution was trained exclusively on artificially generated data and therefore it can't be applied to real world solutions. A way to generate real data would require time and a light source with 3D awareness. It's not so hard to do having AR tools and LiDAR implemented into iPhone, for example. But it will also require actors and sufficient time.

Image 25 "liDAR"
Source

However, basing your solution on generated data it is a great idea to add several virtual actors to the pool. While having so-called light-points as the input to the Model I actually don't need to set up new scenes. What I could do is just add the color to the skin texture so that it could represent the specter of different human races (Caucasian, African, Asian, Hispanic, etc.).

Image 25 "Skin Tones"
Source

To bring back the solution closer to the real-world application is to change the way I represent the light position. A smarter way to represent it would be to calculate polar coordinates. That way we would know the position of the light source independent from the degree of the face towards the camera. Right now Ruslan always faces camera, while in the real world situation you don't position yourself in from of the phone that way.

Theoretically we could represent by the triplet (α,β,d).

Where:

α - the angle in radians between the z-axis of candide3 model (model facing direction).

β - the angle between model y-axis (model up direction)

d - distance to the face.

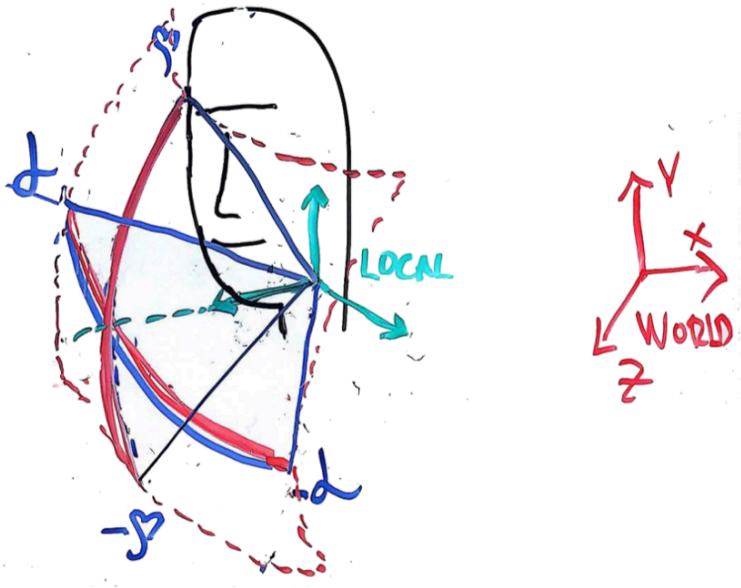

Image 26 "Polar coordinates"

Another way to achieve such result would be to train a Convolutional Neural Network. You could use raw images as the input in such case. However, in order for CNN to perform fast enough for mobile application some optimization techniques must be applied. Example of such technique would be Quantization. Quantization for deep learning is the process of approximating a neural network that uses floating-point numbers by a neural network of low bit width numbers. This dramatically reduces both the memory requirement and computational cost of using neural networks.

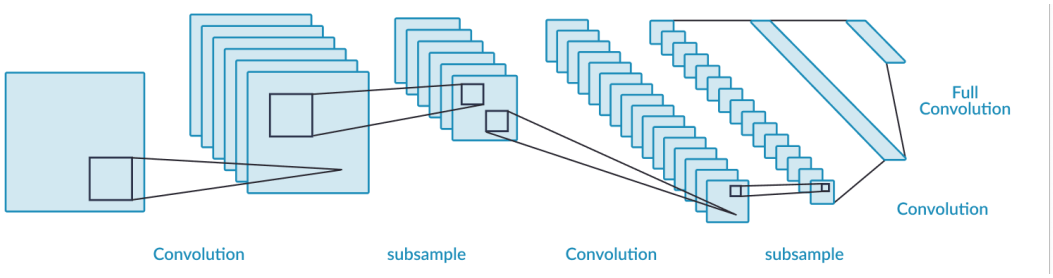

Image 27 "CNN"

But again, another problem is presented with such approach, Convolutional Neural Networks as any other Deep Learning Network requires much more data. This data must also be diverse, so that model can generalize.

The other dimension in which the model can be improved is to "teach" it determine type of illumination. This can include determining the number of light sources, position for each light source, their luminosity and light color. In real world it's rare to have only one light source. A great idea would be to start determining point-like source (as we already did), but also determine ambient light, because it's usually the case in the real world, when you have directional light and also ambience.

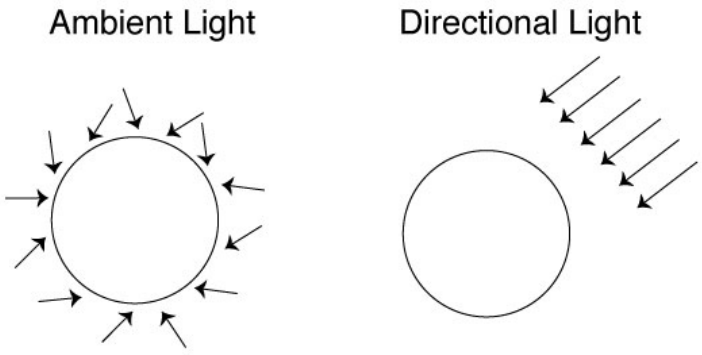

Image 27 "Ambient light"
Source

If it is decided to stay with feature selection that I proposed, the next step into improving it would actually be by using face Candide-3 model, or a similar one.

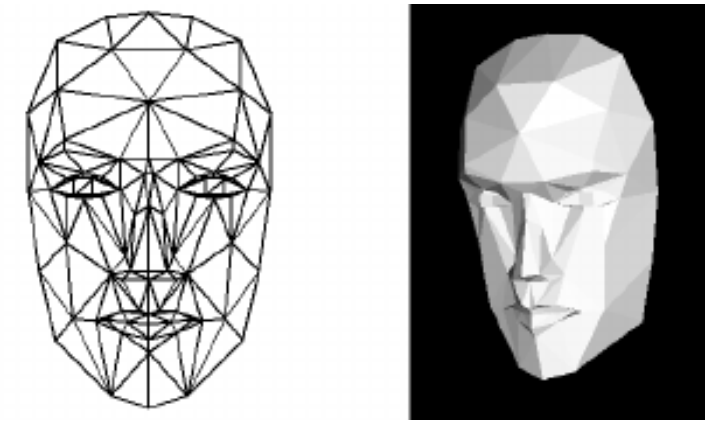

Image 28 "Candide-3"
Source

Using Candide-3 you get a low-polygon version of a face. It lets you to determine the average color inside the polygon, which can be later used to predict light source position, much similarly to the light-point mapping that I propose.

With all that I would like to conclude, that this subject has much potential when it comes to matching augmented reality with real environment. I think that in the future the most optimal way to estimate color atmosphere will be with the help of Deep Learning. Tasks that not so long ago were considered unprecedently complex due to the amount of engineering and brilliancy that must be dedicated to solving them are now completely doable.

Today, having so much computational capabilities, we start to see some truly amazing solutions that make their way into masses. I can't wait to see other wonderful things. As they say, "the future is in the pocket".

List of References