UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

33979 - Visión por Computador

Computer Vision Lab

Author:
Musæus, Lars Gjardar

7th June 2021

Contents

List of Figures

List of Tables

1 Introduction

This documents holds all the explainations for the different tasks/labs in Computer Vision Lab that I've done. It's splitted into different parts based on the task and all the tasks have a shared Github Repo that can be found **here**. I chose to do *Gender recognition*, **Car Model identification with bi-linear models** and **Image colorization**.

2 Gender Recognition

In this to part lab we are going to use a free to use dataset called *Labeled Faces In the Wild - LFW*[2] to implement a model that can accuratly tell the gender of the person in the photo. The code for this part can be found **here**.

2.1 Part 1

In the first part there is no limit in how many paramaters there are in the model, but the accuracy needs to be >97%. This is the first time I've made a model, but after some trail and error I got it working. The main source of inspiration when creating the convolutional neural network was this[1] paper.

1. Filters of all convolutional layers have a spatial dimension of 3x3 pixels and in all layers, rectified linear units (ReLU) are used as activation functions.
2. The input image resolution is 128x128 pixels because initial resolutions of face images in CASIA WebFace and LFW(the one that I use here) vary approximately from 60x60 to 120x120 pixels, and it does not make sense to significantly upsample input faces. Taking into account the smaller inputs, Starting CNN contains 8 in- stead of 10 convolutional layers.
3. Due to the fact that our problem is less complex than Imagenet classification (2 target classes instead of 1000 classes), we have reduced the number of filters in the convolutional layers and we have used only one fully-connected layer.

While trying different setup, I got some poor results and at first did not understand why. It turned out I was using Softmax as activation function, which is a bad choice as the classification problem is binary (Male or Female). By changing to the more suiting activation function Sigmoid which is indeed is used for binary classification. The final model, seen in Figure 1, ended up with 1,339,874 total parameters and a accuracy of 98.19%.

2.2 Part 2

After successfully being able to detect gender with >97% accuracy, it is time for part 2, namely reduce the parameters to <100,000 and still have a accuracy >92%. Using the model given in Figure 1 as a starting point I looked at the the amount of parameters given by each layer. It is fair to say that getting rid of the first dense layer which has 1,180,160 parameters would help, as long as the performance still is alright. The performance did not get much worse, and then by following more or less the setup they have in[1] I obtained the network shown in Figure 2. This network has 75,330 total parameters and a accuracy of 95.62%.

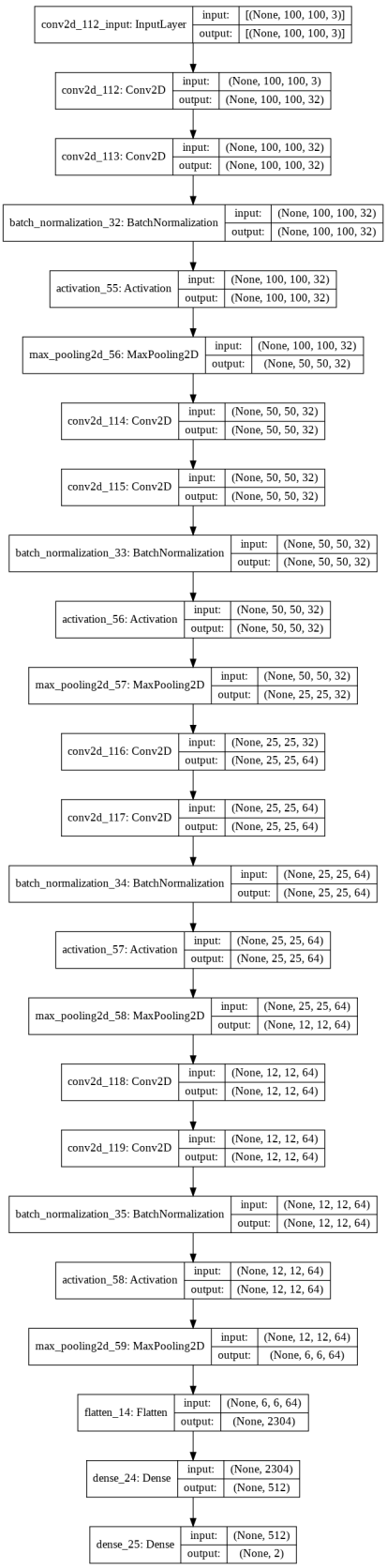

Figure 1: Model for Part 1 of Gender Recognition

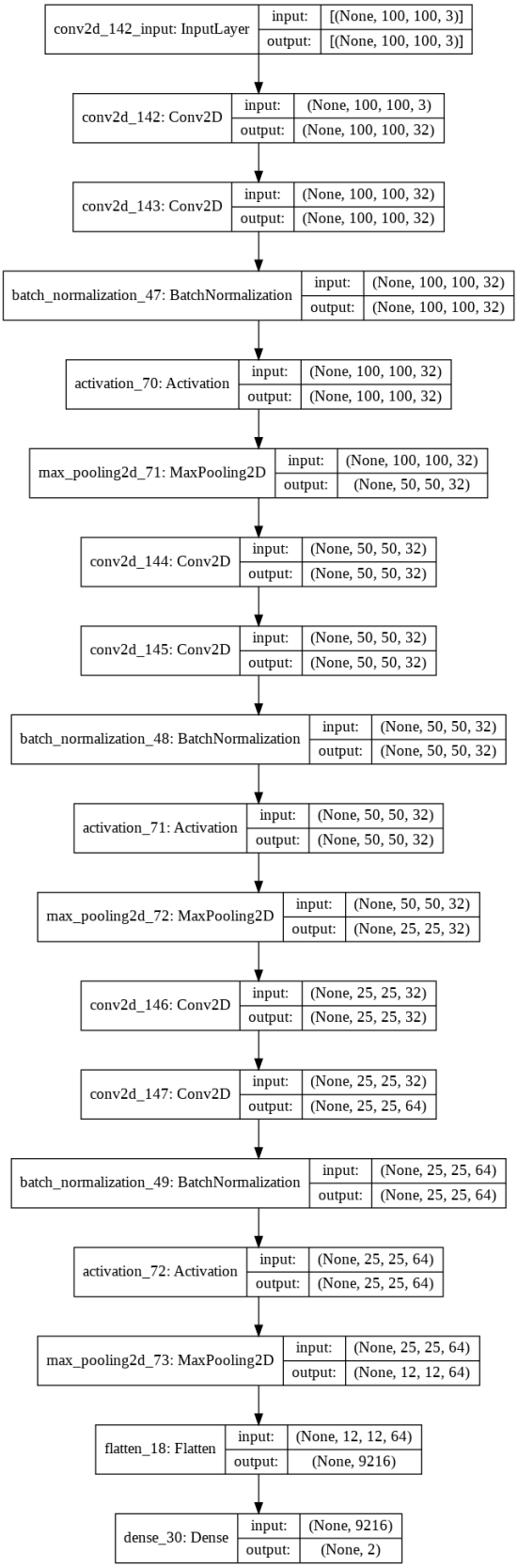

Figure 2: Model for Part 2 of Gender Recognition

3 Car Model Recognition

In this project the goal was to learn about and create a bi-linear model trained to recognize different car models. The reason for using a bi-linear convolutional network is because its used for fine grained classification, i.e. look for small differences in pictures like in this case different car models. So instead of classifying a car and a plane, we classify differences within the same domain. In my solution create the bi-linear model i followed the suggested approach for the task, recited here:

- Load a pre-trained VGG16
- Connect this pre-trained model and form a bi-linear
- Train freezing weights first, unfreeze after some epochs, very low learning rate
- Accuracy above 65% is expected

Thus, first I loaded two pretrained VGG16 networks as shown below:

```
vgg1 = VGG16(
    include_top=False,
    weights="imagenet",
    input_shape=x_train.shape[1:]
)
vgg2 = VGG16(
    include_top=False,
    weights="imagenet",
    input_shape=x_train.shape[1:]
)
```

Then I renamed the layers to distinguish them and then freeze the layers:

```
for layer in vgg1.layers:
  layer._name = layer._name + str('_1')
  layer.trainable = False;
for layer in vgg2.layers:
  layer._name = layer._name + str('_2')
  layer.trainable = False;
```

Then I extract the output of the final layer from each of the VGG16 models, construct a Lambda layer with the result of the outer product function, and then make the predictions:

```
vggOut1 = vgg1.get_layer('block5_pool_1').output

vggOut2 = vgg2.get_layer('block5_pool_2').output

x = Lambda(outer_product, name='outer_product')([vggOut1,vggOut2])

predictions=Dense(num_classes, activation='softmax', name='predictions')(x)
```

Then, finally, I put together the different components to complete the bi-linear network in one model.

```
model = Model(inputs=[vgg1.input, vgg2.input], outputs=predictions)
```

After this, I compile the model, train it with the frozen weights, unfreeze all weights in both of the VGG16 networks, re-compile it with a slower learning rate and train it again.

After training with frozen weights for 250 epochs I obtained an accuracy of 52.42%, and after unfreezing the weights like was suggested in the task description and then train for another 100 epochs I achieved a final accuracy of 80.23% which is a result I am pleased with. What I learned while doing this project is that how important it is with a slow enough learning rate for the optimizer. A "slow learning rate" was suggested, but being unexperienced I learned quick that 0.01 is not "slow", and had to try everything from 0.1 to 0.000001 before ending up with 0.00001 as the final learning rate. The training results can be seen in Figure 3 and Figure 4. The full code/notebook is found on my Github **here**.

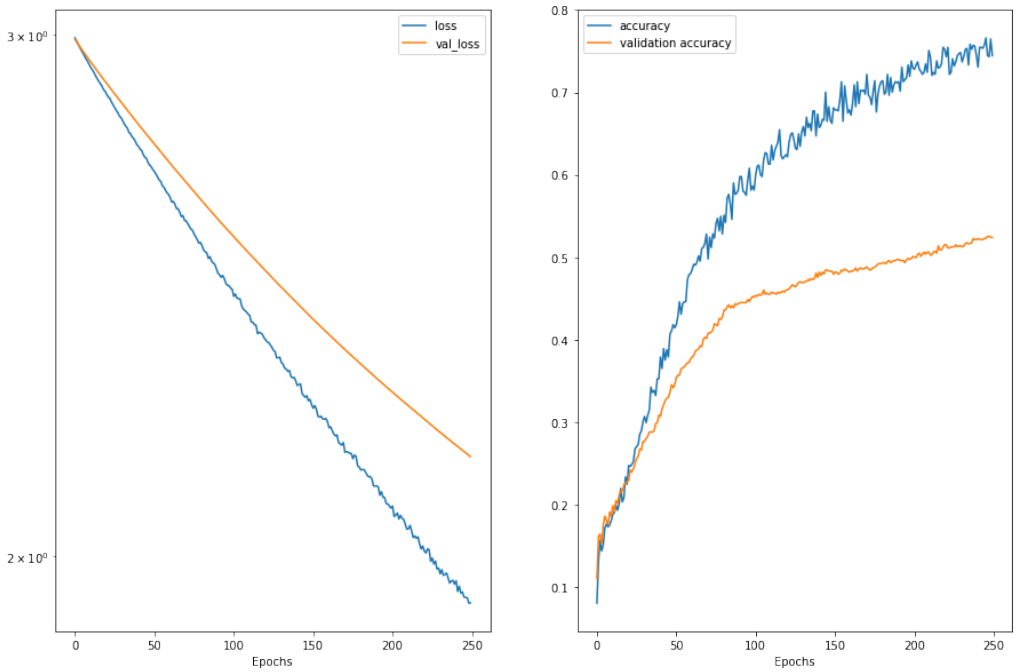

Figure 3: Training with frozen weights

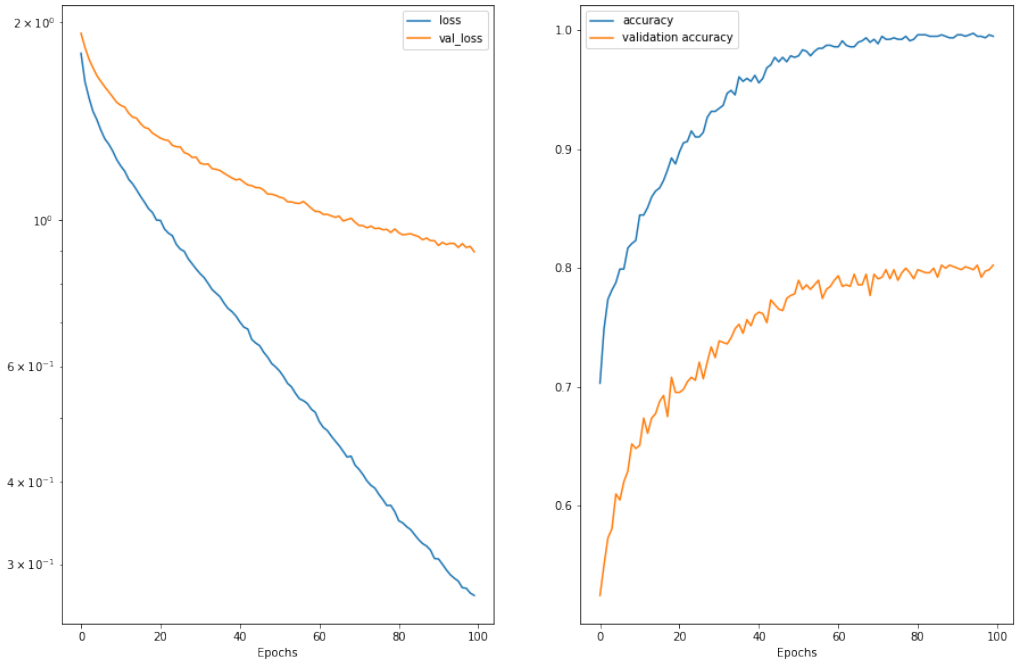

Figure 4: Training with unfrozen weights

4 Image Colorization

The key idea of this task was to be able to get output from the Inception network, and feed this into the fusion layer alongside the output of the encoder. This becomes a bit tricky when you want to use a generator, as we do, because we then want to get the output of the inception network inside the generator function, and later on yield this as input to the complete model for training.

This can be acchieved by calling a function (for simplicity) that takes the grayscaled rgb image as an input, and returns the predictions of the inception network, inside the generator.

Going through the code sequentially I will explain how this is acchieved, and how we in the end end up with a colorized image.

This part is just used to download the test and training images from **this** github repository. Once the images are downloaded the next face is to put the training data in the correct 24-bit RGB format:

```
!wget
↪  https://github.com/emilwallner/Coloring-greyscale-images/archive/master.tar.gz
!tar xvzf master.tar.gz
path = 'Coloring-greyscale-images-master/Full-version/Train/'
X = []
for filename in os.listdir(path):
    X.append(img_to_array(load_img(path+filename)))
X = np.array(X, dtype=float)
Xtrain = 1.0/255*X
```

Then we instantiate the Inception ResNet V2 model:

```
inception = InceptionResNetV2(weights='imagenet', include_top=True)
```

Then we construct the encoder:

```
encoder_input = Input(shape=(256, 256, 1,))
encoder_output = Conv2D(64, (3,3), activation='relu', padding='same',
↪  strides=2)(encoder_input)
encoder_output = Conv2D(128, (3,3), activation='relu',
↪  padding='same')(encoder_output)
encoder_output = Conv2D(128, (3,3), activation='relu', padding='same',
↪  strides=2)(encoder_output)
encoder_output = Conv2D(256, (3,3), activation='relu',
↪  padding='same')(encoder_output)
encoder_output = Conv2D(256, (3,3), activation='relu', padding='same',
↪  strides=2)(encoder_output)
encoder_output = Conv2D(512, (3,3), activation='relu',
↪  padding='same')(encoder_output)
encoder_output = Conv2D(512, (3,3), activation='relu',
↪  padding='same')(encoder_output)
encoder_output = Conv2D(256, (3,3), activation='relu',
↪  padding='same')(encoder_output)
```

Then we create an input that will, when running the model, represent the predictions from the inception network, that as explained above will be one of the two inputs to the complete model.

```
embed_input = Input(shape=(1000,))
```

Then we need to reshape our inception output so that it will fit with the output from the encoder when we are fusing them.

```
fusion_output = RepeatVector(32 * 32)(embed_input)
fusion_output = Reshape(([32, 32, 1000]))(fusion_output)
```

Then we are ready to fuse the two outputs, i.e. concatinate them to be more precise.

```
fusion_output = tf.keras.layers.Concatenate(axis=3)([encoder_output,
↪  fusion_output])
```

After that, we perform a 1*1 convolution to reshape the concatinated layer to the shape required by the decoder. After this part we now have our decoder input ready.

```
fusion_output = Conv2D(256, (1, 1), activation='relu',
↪  padding='same')(fusion_output)
```

Then we construct the decoder, which obviously uses the fusion_output as input.

```
decoder_output = Conv2D(128, (3,3), activation='relu',
↪  padding='same')(fusion_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(64, (3,3), activation='relu',
↪  padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(32, (3,3), activation='relu',
↪  padding='same')(decoder_output)
decoder_output = Conv2D(16, (3,3), activation='relu',
↪  padding='same')(decoder_output)
decoder_output = Conv2D(2, (3, 3), activation='tanh',
↪  padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
```

Then we are ready to create our complete model, which as mentioned before takes two inputs: A grayscaled image to run through the encoder, and the predictions of the inception layer. The output is of course the output of the decoder.

```
model = Model(inputs=[encoder_input, embed_input], outputs=decoder_output)
```

The following function will be called inside the generator and it will return the predictions from the inception network.

```
def create_inception_embedding(grayscaled_rgb):
    grayscaled_rgb_resized = []
    for i in grayscaled_rgb:
      i = resize(i, (299, 299, 3), mode='constant')
      grayscaled_rgb_resized.append(i)
    grayscaled_rgb_resized = np.array(grayscaled_rgb_resized)
    grayscaled_rgb_resized = preprocess_input(grayscaled_rgb_resized)
    embed = inception.predict(grayscaled_rgb_resized)
    return embed
```

Then we define a simple image data generator to slightly alter the training pictures, and define the batch size we will use.

```
datagen = ImageDataGenerator(
        shear_range=0.4,
        zoom_range=0.4,
        rotation_range=40,
        horizontal_flip=True)

batch_size = 20
```

This is the generator that keras will use to get the training set in each epoch. It mostly consists of obtaining the correct processed data, but notice the call to create_inception_embedding that we have mentioned before where we obtain the predictions of from the inception network.

```
def image_a_b_gen(batch_size):
    for batch in datagen.flow(Xtrain, batch_size=batch_size):
        grayscaled_rgb = gray2rgb(rgb2gray(batch))
        embed = create_inception_embedding(grayscaled_rgb)
        lab_batch = rgb2lab(batch)
        X_batch = lab_batch[:,:,:,0]
        X_batch = X_batch.reshape(X_batch.shape+(1,))
        Y_batch = lab_batch[:,:,:,1:] / 128
        yield ([X_batch, embed], Y_batch)
```

Then we compile and train the model, using the generator we defined above.

```
tensorboard = TensorBoard(log_dir="/output")
model.compile(optimizer='adam', loss='mse')
model.fit(image_a_b_gen(batch_size), callbacks=[tensorboard], epochs=50,
↪  steps_per_epoch=5)
```

Then we load our test data and pre-process it just as we did with the training data. Notice that we also here call the function create_inception_embedding

```
path = 'Coloring-greyscale-images-master/Full-version/Test/'
color_me = []
for filename in os.listdir(path):
    color_me.append(img_to_array(load_img(path+filename)))
color_me = np.array(color_me, dtype=float)
color_me = 1.0/255*color_me
color_me = gray2rgb(rgb2gray(color_me))
color_me_embed = create_inception_embedding(color_me)
color_me = rgb2lab(color_me)[:,:,:,0]
color_me = color_me.reshape(color_me.shape+(1,))
```

Then we test our model, obtaining the predictions before we scale it and convetert it to an rgb image that we store in our Result folder.

```
output = model.predict([color_me, color_me_embed])
output = output * 128

for i in range(len(output)):
    cur = np.zeros((256, 256, 3))
    cur[:,:,0] = color_me[i][:,:,0]
    cur[:,:,1:] = output[i]
    imsave("Result/img_"+str(i)+".png", lab2rgb(cur))
```

The code is based on [3] with some slight modifications to make it runnable.

I ran the model first with 50 epochs to see if the model worked, which it did, so then I ran the model with 1000 Epochs. With 50 epochs you definetly see colors compared to black and white, but with 1000 its a lot more colors. The resulting images after 50 epochs can be seen in Figure 5, and the results after 1000 epochs can be seen in Figure 6. The code for this project is also found on my Github **here**.

Figure 5: Image colorization after 50 epochs

Figure 6: Image colorization after 1000 epochs

Bibliography

[1] Grigory Antipov. *Minimalistic CNN-based ensemble model for gender prediction from face images*. URL: https://www.semanticscholar.org/paper/Minimalistic-CNN-based-ensemble-model-for-gender-Antipov-Berrani/d0eb3fd1b1750242f3bb39ce9ac27fc8cc7c5af0?p2df (visited on 29th Apr. 2021).

[2] The official website for LFW. URL: http://vis-www.cs.umass.edu/lfw/index.html (visited on 29th Apr. 2021).

[3] Emil Wallner. *Colorizing B&W Photos with Neural Networks*. URL: https://blog.floydhub.com/colorizing-b-w-photos-with-neural-networks/ (visited on 29th May 2021).