UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS num. 2322

Enhancing transfer learning for head pose estimation

Josip Matak

Zagreb, July 2020.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 13 March 2020

MASTER THESIS ASSIGNMENT No. 2322

Student:	**Josip Matak (0036491106)**
Study:	Computing
Profile:	Computer Science
Mentor:	prof. Domagoj Jakobović

Title: **Enhancing transfer learning for head pose estimation**

Description:

Describe the problem of head pose estimation from RGB images. Conduct an in-depth analysis of problem applications as well as other methods of solving the problem. Explore the possibility of taking advantage of previously trained models and labels generated with state of the art methods using a domain adaptation and transfer learning. Use model evolution for finding the most suitable model with as few parameters as possible. Define an appropriate fitness function for the process of evolving architectures, with detailed importance analysis for each of the fitness function components. Compare the efficiency with the state of the art models on relevant datasets. Include original program codes, obtained results with usage instructions and used literature.

Submission date: 30 June 2020

U ovaj rad kao kruna mog akademskog obrazovanja satkana je dobra volja iznimno bliskih i dragih osoba.

Posebnu, ujedno i najveću zahvalu dugujem svojim RODITELJIMA čija su bezuvjetna potpora, pažnja i razumijevanje bili najveći motivator preko ugodnih, do onih najtežih trenutaka.

Zahvaljujem se svim prijateljima, kolegama i dragim osobama koje su oblikovale put koji je doveo do ovog rada. Zahvaljujem se mentoru, prof. dr. sc. Domagoju Jakobovića na savjetima i pomoći kroz cjelokupnu zajedničku suradnju. Također, zahvaljujem se tvrtci RealNetworks, posebno mentoru dr.sc. Pavlu Prentašiću, koji su mi omogućili temu, savjete, kao i sredstva za istraživanje na ovom radu.

CONTENTS

1. Introduction

The hot topic in today's technology and software development is face modeling. Techniques such are facial recognition, detection, or landmark extraction play a key role in widely used software. The need for described tasks has developed with the availability of devices for digital image analysis. Nowadays, a key role in designing software that can handle these tasks is put on increasing precision and reducing computational complexity.

One approach for reducing computational cost is transfer learning. After building a model for a certain task, its knowledge can be used and distilled for another, closely related task. Head pose estimation is an example of such a method that is going to be analyzed in this thesis. The common solution that can generate a state of the art results is deep learning. In this thesis, facial recognition is going to be used as the baseline task, which knowledge will be transferred to estimate head pose.

In order to have a more precise model, a specific and compound neural network architecture must be found. Changing the depth (number of layers), width (number of channels), used operations, and other hyperparameters of the neural network results in accuracy change. Using evolutionary computing, the goal of this work is to manipulate a learning process in such a way to generate more precise and less complex models. This will benefit the final precision and latency.

1.1. Contribution

Among some of the previously used techniques, there are several contributions of this master thesis. Although work on this thesis is focused on estimating head pose, multiple areas of heuristic methods and deep learning are used. In the first chapter, the head pose estimation problem is explained in detail, as well as associated work in that area. The main contribution, a solving method, is also revealed and clarified in the first chapter. Moving towards the next chapter, we demonstrate deep learning methods and building blocks that are part of the study. Another type of analysis in the area of heuristic and optimization methods is conducted in the third chapter. Following, real application and implementation of the optimization pipeline is explained. The last chapter gives insight into the final results.

2. Pose estimation problem

When given a photograph with a human face on it, even the most skillful people won't be able to give the answer to what is the angle that describes the orientation of the head. This challenging problem usually seeks more resources in terms of depth cameras or 3D sensors which will generate precise information on the presented task. Unfortunately, for usage in real-world practice, these techniques would request spending a large amount of money given the price of those products. However, nowadays developed software that depends only on single-image is able to provide precise information about an angle.

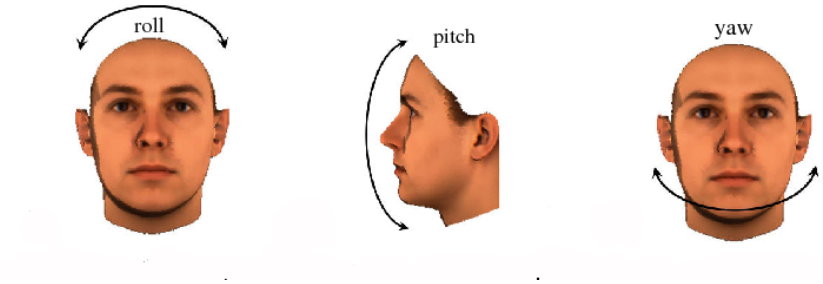


Figure 2.1: Head pose angles showing roll, pitch and yaw, respectively [Vatahska et al. (2007)]

Head pose is defined with a 3D vector that describes yaw, pitch, and roll, meaning angles defining rotation around Z, Y, and X-axis in a common coordinate system, respectively. Better insight into previous statement can be found in the figure 2.1. It is the relative orientation of the head with respect to the local head coordinate system. Such a problem can be described in another way, it is the problem of mapping 2D data to a 3D space.

Information about head pose can be useful where there is a need for modeling information about the human head. Industries such are healthcare, car industry, or security dependable places like airports, offices, and educational facilities all use tasks where head pose information is used. Use cases can vary across different practice areas. Such can include enhancing gaze direction estimation, which can be used in driving for detection of the loss in focus. Similarly, pose estimation is the backbone for any gesture detection software, vastly being applied in the healthcare industry. Finally, using the task of estimating the head's rotation, more useful information can be introduced to other computer vision techniques such may be facial identification.

2.1. Problem formulation

In order to fully reconstruct head position and orientation in 3D space, 2 kinds of motion should be presented.

1. **Translation:** moving an object from origin point to a new 3D location. Translation vector that moves an object from point (x, y, z) to point (x', y', z') can be written like: $\mathbf{t} = (x - x', y - y', z - z')$.

2. **Rotation:** operation of rotation depends on three different directions around x, y and z-axis. It can be written as:

   (a) 3x3 rotation matrix

   (b) Euler angles representing yaw, pitch and roll

   (c) Direction vector and angle

In this work, only rotational transformation is considered. Given the RGB image of a face, the goal is obtaining all three Euler angles which fully describes the head's pose.

2.2. Related work and solving techniques

In past years, researchers have yielded many different approaches in solving head pose estimation problem. However, two approaches stand out, being divided into landmark-based and landmark-free approaches.

The landmark-based approach tends to fit a 3D head model using facial landmarks [Abate et al. (2019), Vatahska et al. (2007)]. Facial landmarks are a representation of salient regions on the face, including eyes, mouth, ears, jaw, etc. They are widely used in mapping the structure of the face, therefore they are a valuable tool in many computer vision tasks, such is pose estimation. To determine Euler angles from facial landmarks, one has to establish the correspondence between landmarks and the 3D head model and calculate alignment parameters. Since regressing the pose from a face image isn't its main task, obtained results may not be sufficiently accurate. Another downside to this approach is a dependence on the quality of the 3D model, which varies by the number of represented features. Also, high computational complexity can be a time bottleneck that may make this approach unuseful for practical implementation.

On the other side, landmark free methods directly regress pose angles from the input image. Most of the approaches rely on the deep learning and automated feature extraction, while historically several tried to handcraft features which they found to be important for this task. Using mostly convolutional networks, authors proposed several different neural

network architectures to deal with the problem. For example, [Wang et al. (2019)], as well as [Ruiz et al. (2018)] proposed commonly used convolutional techniques, while modeling different loss functions. By contrast to such approach, [Yang et al. (2019)] deliver FSA-Net, a novel hybrid architecture which can obtain a state of the art results on standardized tests. In the next chapter, a transfer learning method will be explained.

2.3. Proposed method

As mentioned above, different techniques and different approaches all led to the same conclusion. That is the motivation for proposing a new method that will be able to yield similar results. Since head pose estimation is closely related to other facial modeling tasks, the suggested method includes taking a different level of features from the model of the already solved related task.

Such a task can vary and it can be facial recognition, facial detection, or another familiar task. Using extracted features and deep learning model on top of them, the assumption is that it will be sufficient enough to generate close to a state of the art results and will be useful for fine practical application. This is the example of transfer learning which is going to be explained later in the work. The advantages of this technique rely on the fact that developed models can be used for another problem. Furthermore, if features are well selected, with small data overhead it is possible to obtain accurate results (Figure 2.2).

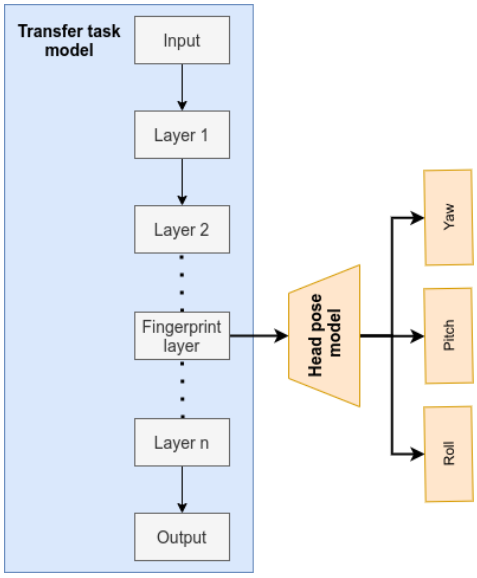


Figure 2.2: Transfer learning visualization, taking the exraction layer from related face modeling task

3. Deep learning

Formally explained, deep learning is a subclass of machine learning algorithms that tends to derive fine-grained features using matrix operations from raw input. The idea for deep learning firstly came from the idea of the human brain, a fully neurologically connected unit capable of solving complex problems. When it first came to the computer science scene with trained perceptron [Rosenblatt (1958)], this method didn't have the capacity to become what it is today. Publications in this area have put the deep learning among top ranked research areas [Goodfellow et al. (2016)].

In a few years, deep learning has become standard in various tasks, such as include image processing, computer vision, natural language processing, bioinformatics, etc. None of this would be possible if there was no investment in solving principal problems in the study of deep learning algorithms. Such a scenario also became real with development in hardware which was able to handle big amounts of matrix processing. The next section will explain some of the basic concepts, then moving towards more advanced topics in the following sections.

3.1. Basic concepts

3.1.1. Multi layer perceptron

As it is stated before, a deep learning concept is based on neural activity. Basic building block for every neural network is perceptron, in biological terms it is a neuron that fires on some kind of previous activity. Mathematically speaking, it is usually a non-linear function that transforms input with respect to weights factor and activation. This simple transformation can be written as:

$$a(\sum_{i=1}^{N} w_i x_i + b) \equiv a(\mathbf{w} \cdot \mathbf{x} + b) \tag{3.1}$$

In the equation, a denotes some kind of activation function, regardless of it being linear or non-linear. Summation of input (denoted as x_i) with weights (w_i) and addition of bias b_i (can be seen as activation threshold) simply mimics human neuron. Stacking those elements

is used for building a complex model that tries to replicate the neural system. This model is called a multi-layer perceptron, where layers are denoted as one-way transformation. "Training" such a model is considered in adapting weights factors to properly transform input to wanted output.

3.1.2. Convolutional layer

To address problem of image processing, previously explained transformations have no effect. Effective fix to this problem has emerged from empirical tests. Another type of transformation is used to reveal salient image features. These transformations are made from operation of convolution. Because of the discrete domain and matrix operations, convolution is mathematical operation of function w usually called kernel, and function $f(i, j)$, original image with elements on position (i, j). It is applied on image tensor with dimensions of [width $\times$ height $\times$ depth]. Operation can be written as:

$$G(i,j) = w * f(x,y) = \sum_{u=-k}^{k} \sum_{v=-k}^{k} w(u,v) f(i-u, j-v) \tag{3.2}$$

Where $G(i, j)$ denotes filtered image and u, v are offset in kernel. In the general case (described in figure 3.1) in large dimensional image data, a filter is the collection of kernels across input dimension depth called the number of input channels. Each filter in the convolutional network produces one output channel.

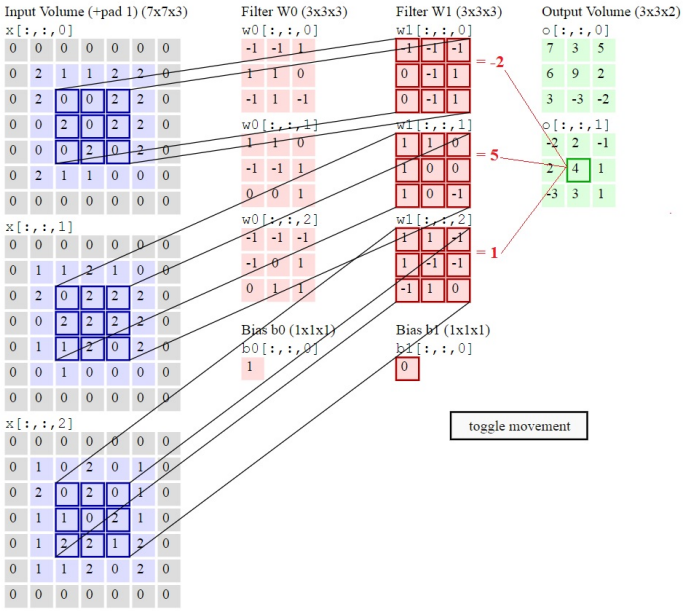


Figure 3.1: Visualization of 2D convolutional operation. Number of blue matrices stand for input channels, while green matrices are produced by 2 different red filters and their quantity stand for number of output channels. Illustration by [StanfordVisualLab]

3.1.3. Pooling layer

Not all features and information from convolution are useful for the final task. This is why pooling layers are useful. Input tensor is split into regions of size [pool_size $\times$ pool_size], each of these regions produces only one input, which is regularly a maximum or average element, which is used in maximum pooling and average pooling elements respectively.

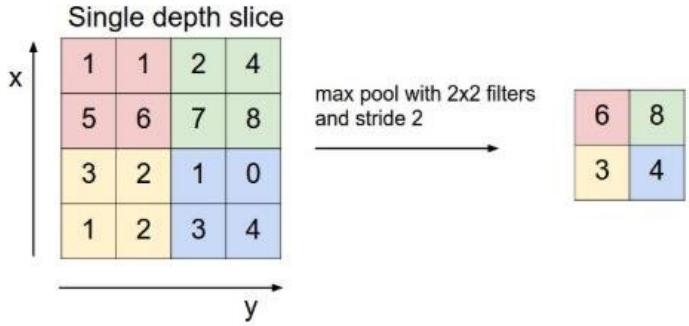


Figure 3.2: Maximum pooling layer [Illustration by Asad et al. (2018)]

3.1.4. Activation function

Mathematical functions that describe behavior in nature are usually complex and can not be addressed with simple regression to some simple formula. Because of that, the deep model uses a non-linear transformation to deal with this problem. The most popular and commonly used is ReLU transformation, which stands for "rectified linear unit". It is defined as a positive part of input:

$$f(x) = max(0, x) \tag{3.3}$$

Furthermore, experience has shown that it is possible to improve accuracy of models by introducing learnable leakage parameter a. This leads to the new activation function called PReLU, or "parametric rectified linear unit". Finally, formula looks like this:

$$f(x) = \begin{cases} x, \text{if } x > 0 \\ a \cdot x, \text{otherwise} \end{cases} \tag{3.4}$$

The benefit of such activation function is gradient propagation of only important factors. Moreover, it has been shown that the model can be trained faster once when ReLU activation is applied, because of the non-saturating gradients (stuck in the same value for a large number of epochs).

On the other side, problems might occur when defining a derivative in zero point, because it is mathematically undefined. Finally, another problem is dead units. In terms of ReLU activation, this happens once when the activation is constantly negative in a certain neuron,

therefore it outputs the same value. Parametric ReLU prevents such behavior by adding parametric leakage factor.

3.1.5. Batch normalization

A lot of improvement in the learning phase can be gained by normalizing input, shifting inputs to zero mean and unit variance. Exactly that does batch normalization layer [Ioffe and Szegedy (2015)]. The author's suggestion is performing BN transformation immediately before activation because constraining its first and second moments would not eliminate the distribution covariance shift. Layer normalizes input in the training phase and subsequently updates global mean and variance variables which will be used in inference.

Ideally, whole training data would need to be processed for estimating mean and variance, but since the stochastic system is used, during training phase normalization is fixed to the size of a learning mini-batch. For the inference step (equation 3.5), mean and variance are calculated as expectancy and variance estimation for these parameters using the information from training batches. Input and output are denoted as x and y, expectancy and variance are calculated from training variables, while γ and β are learnable variables.

$$y = \frac{\gamma}{\sqrt{\mathrm{Var}[x] + \epsilon}} x + \left(\beta - \frac{\gamma E[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} \right) \tag{3.5}$$

The benefits of this method include faster learning because layers work better with normalized input. Furthermore, the cost of data normalization offers the benefit of higher learning rates, easier weights initialization, and diverse activation of neurons.

3.1.6. Training

The most important element when using deep learning model is probably its learning phase. Like it is mentioned before, training means adapting weights to properly transform input to wanted output. Few things are important for the training phase: initialization of parameters, loss function, and learning algorithm.

Parameter initialization. Since gradient vanishing and exploding can be a massive problem in neural network learning, proper initialization can play a key role in preventing this scenario. To maintain the level variance and mean of gradients through the whole network, [Glorot and Bengio (2010)] proposed a new way of initialization:

$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}} \tag{3.6}$$

Where n_i is number of incoming, while n_{i+1} is number of outgoing connections.

Loss function. Quantization of how good the network performs is called a loss function. It can be measured as properly classified examples, or distance between output and wanted result. This function is optimized for better network performance.

Learning algorithm There are many ways to train a deep model, most popular, and the way used in this work is backpropagation. First, we introduce examples of x and wanted labels y. There is a requirement for loss function to be derivable. Loss function $L = f(y, y')$, where y' is network's output $y' = g(x)$, shows how dependable are input and output in loss function. By calculating an error gradient and therefore moving weights in the opposite way of a gradient, the model will decrease a loss function value, therefore lower the error. It uses a chain rule to calculate gradients layer by layer and optimize the network from it's output to the input layer. This is done by calculating partial gradients $\frac{\partial L}{\partial w}$, where L is a loss function and w is network parameter.

Although it is a derivative method, because of stochastic behavior and the complex nature of the loss function, there is no insurance that method will converge to a global minimum. Also, when using backpropagation in the training process, complex operations can not be used since every operation needs to be derivable. Such a constraint can be avoided by using other, derivative-free algorithms.

3.2. Transfer learning

The same tool can be used multiple times in numerous closely related tasks in nature. In such a way function machine learning algorithms. Using the basic knowledge obtained from one task, it is a common assumption that sharing that knowledge can improve work on multiple associated problems. If data comes from the same distribution, it is possible to acquire basic knowledge about data, and use it in improving performance on the target task if such target is a subcategory of provided task.

In the literature [Ruder (2019)], most of the scientists split transfer learning techniques into a few categories. Those categories are:

- **Inductive transfer learning:** having both the same domains for source and target tasks, inductive transfer learning uses basic domain knowledge and address it on specific tasks. It can be done in two distinct ways, by multi-task or sequential learning, depending on do we introduce multiple tasks at once or one-by-one.
- **Transductive transfer learning:** there are lots of labeled data in the source domain, but none in the target domain. Such a method is also called "domain adaptation". By learning shapes and other important features, the model can learn to distinguish objects in 2 different domains without explicitly showing the difference.

– **Unsupervised transfer learning:** this is closely related to inductive transfer learning where the target task is different but related to the source task. What differs these techniques from the previous two is focus on the unsupervised task. The goal is using the basic knowledge to solve clustering, reduction of dimensionality, etc.

This work will utilize sequential transfer learning, a subcategory of the inductive transfer learning technique. A goal is to use a fully trained network that was used on one task, to be able to operate with a new task, only by adding a small amount of newly trainable parameters.

3.2.1. Popular networks

Deep learning is a hot topic in modern computer science research. There has been a vast majority of papers published in the last five years. Therefore, there has been a lot of different approaches in particular regions of computer vision that majorly exploit the benefits of deep learning. With that in mind, it is important to mention widely and commonly used networks for face modeling tasks. Once when the most similar task to estimating head pose is addressed, the best working model is used for transferring knowledge.

As it was previously emphasized, to get the best out of image data, convolutional neural networks play a key role. Their architecture has been growing since the first commonly known architectures, as well as the number of parameters and floating operations. With the growth of the network's complexity, accuracy has also increased. The main reason behind that statement is an expansion of capacity, with larger width (number of convolutional channels) or more depth (number of layers) network performs better because it is able to find and extract important hidden features related to the input image.

When it comes to face modeling tasks, two of them play a major role and arise as clear options for transferring knowledge. These are face recognition and face detection task. While face recognition is nowadays strongly related to deep learning, face detection can use simpler ways to deal with the problem. However, both use common architectures that can extract important information about the image.

Historically speaking, with the introduction of AlexNet [Krizhevsky et al. (2012)], a hype for solving computer vision tasks with deep learning has raised. That paper presented the usage of the enormous dataset and GPU learning technique. The most popular architecture in the face detection area is probably "Multi-Task Cascaded Convolutional Neural Network" [Zhang et al. (2016)] or shortly, MTCNN.

It is a hybrid deep learning method. The name comes from multiple scales image is resized on the start. Following the scaling, three parts of the network process the image (Figure 3.3). First is called "Proposal network" (P-Net) which generates candidate facial regions. The next step is the "Refine network" (R-Net) which refines those regions to single bounding boxes. The final stage is called "Output network" or O-Net, which extracts features

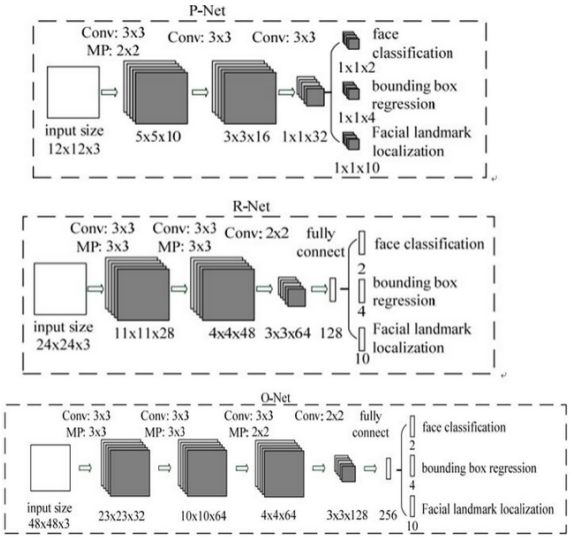


Figure 3.3: MTCNN architecture visualization [Zhang et al. (2016)]

and obtains facial landmarks. Although it is most popular, MTCNN is not a perfect face detector. Nowadays, architecture like RetinaFace [Deng et al. (2019)] or AlnnoFace [Zhang et al. (2019)] are used for the state of the art performance. Once again they feature multi cascade pyramid network which is suitable for such task.

Moving to the next task, face recognition has used more networks over the years. The first leap forward has come with DeepFace [Taigman et al. (2014)], a CNN in which the input is a face image preprocessed with 3D-alignment. DeepID [Sun et al. (2014)] systems were among the first ones that achieve better than human precision on facial recognition datasets. Subsequent work has come up with techniques of FaceNet [Schroff et al. (2015)] and VGGNet [Simonyan and Zisserman (2014)], which are used as the backbone network for many of today's face recognition architectures.

It is important to mention Inception-ResNet [Szegedy et al. (2016)] architecture. With its deep layers and convolutional nature, it is the ideal fit for face modeling tasks. By introducing techniques such as dimensionality reduction blocks, batch normalization, and residual skip connections, this network has achieved the state of the art performance on many problems. It is built from various building modules which were introduced in version v4, where those modules became uniformly designed over the network. Adding residual blocks didn't contribute to increasing accuracy, but it allowed the network to train much faster, which is the reason why those layers are still used in this architecture. Model is built with three main modules of different names, module A, module B, and module C, which are used to change the width and height of the grid. The first module outputs the grid size of 35x35, following by 17x17, and 8x8 respectively from names. Model's top-down architecture can be seen on figure [3.4]. Inception ResNet is extremely fast, taking into consideration its complex and deep architecture.

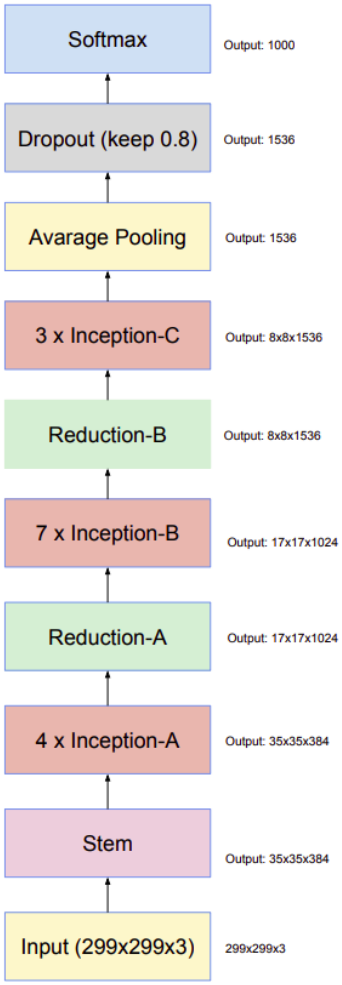


Figure 3.4: Inception-ResNet-v2 (Illustration, [Szegedy et al. (2016)]), a backbone architecture for FaceNet, with slight change in output layers and convolutional grid size.

3.3. Neural network architecture

The important detail in building the neural network is its structure or how layers are connected. Since deep learning models can't be interpreted as other common machine learning algorithms, designing them brings intuition and specific domain knowledge. The model strongly depends on its architecture and a lot of work has been performed in finding the most suitable architecture for a particular task. Tuning such search requires lots of human interventions, and there is no silver bullet on how to deal with that problem.

Today, deep learning models differ from a number of trainable parameters to variance in used operations. Research in this area has been growing gradually, from simpler models to more complicated ones. These complex models were often inspired by simpler ones.

The initial task is to determine which type of network should be employed for this problem. As it was already mentioned, certain network type is strongly correlated with the task

it can solve. Recurrent network solves time dependable problems, radial basis networks are connected with clustering and autoencoder networks sample from the distribution. Perfect choice for image processing is a convolutional neural network (CNN).

Moreover, after the proper type of model has been picked, numerous types of operations can be stacked together. Operations can be stacked into blocks, or be sparsely connected, as can be seen in [3.7]. In that equation, $\mathcal{N}$ is a network, k denotes the number of layers, and $\mathcal{F}$ is processing operation. Having in mind all the variations of different layers that can be included in a deep model, search space for finding the most suitable one is enormous.

$$\mathcal{N} = \mathcal{F}_k \odot ... \odot \mathcal{F}_2 \odot \mathcal{F}_1(\mathcal{X}) \tag{3.7}$$

Computational resource constraints may also affect the model's architecture. By introducing more layers or a greater number of channels, an amount of parameters increases. Processing those huge amounts of parameters on a graphical unit can be time-consuming. Neural network speed is correlated with a number of operations needed for a forward pass. That number is called FLOP, or floating point operations. That number, although it is heavily theoretical since it does not depend on the number of memory accesses or branching, can give a helpful insight into speed. The number of parameters may also be a good indicator for speeding up performance. Reducing the number of fully connected layers and adding convolutional ones can exploit the power of the graphical processing unit, and therefore create the lightweight and fast model.

3.4. Network scaling

When designing a convolutional neural network, general practice is to come up with a baseline model and scale it for better performance. The premise is that with a larger number of parameters and more floating operations, the network should be able to learn and therefore, generalize better. Balancing network width (number of convolutional channels), depth (number of layers), and resolution (input height $\times$ width dimension) will lead to better performance [Tan and Le (2019)]. Research has shown that scaling multiple dimensions at once (also called compound scaling) can produce substantially better results than scaling one dimension at the time.

Today's models rely on an increasing number of parameters to gain accuracy performance. Fortunately, that is not always the case. Depending on the task, equivalent performance can be obtained using ResNet-18 as well as using ResNet-200, two networks with different number of layers expressed in name. Although there is four times more capacity in ResNet-200, the network does not generalize better. The gain in efficiency and accuracy is clear on that example.

Following the cited research, the premise is that scaling up to 2^N more computational resources (FLOPs) requires increasing the network's depth by a factor α^N, width by β^N, and resolution by γ^N with respect to equation [3.8]. Such scaling is intuitive, by enlarging the input resolution factor γ, fewer resources should be spent on broadening depth or width to maintain a certain scale. The same applies to changing the other two factors.

Since dimensions of scaling are not independent, the empirical analysis came up with a formula that describes behavior in scaling multiple dimensions at once. Given the target on memory and FLOPs resources, the way that compound scale works can be written as [Tan and Le (2019)] proposed:

$$\begin{aligned} \text{depth: } d &= \alpha^N \\ \text{width: } w &= \beta^N \\ \text{resolution: } r &= \gamma^N \\ \text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 &\approx 2 \\ \alpha \geq 1; \beta \geq 1; \gamma &\geq 1 \end{aligned} \tag{3.8}$$

In this scenario by the formula [3.8], N is a factor that determines how much more resources can be added into the network, meaning, having this factor equal to 3, our network will be scaled 2^3 times. In this thesis, 2^N is called "FLOPs multiplier" and will be used as a scaling factor. Also, α, β and γ describe how to assign those resources to the model. Empirical formula says that if we want to double the resources we can linearly enlarge the depth factor , or quadratically change the width or resolution scaling factors (β and γ). Moreover, since dealing with transfer learning there is no need to talk about image resolution. Input is a 3D tensor of the extracted layer and therefore it is less scalable. To come up with this, the resolution is considered constant, width and depth are the only scalable factors.

Our goal is to generate a model with FLOPs resource restriction by scaling 2 dimensions. To address the contribution of this work, the obtained model will only be used for a baseline, on which neural architecture search method will be applied.

3.5. Building blocks

Neural networks can be built by stacking different blocks. These blocks may contain different operations and therefore be optimized for a certain task. With multiple available convolution types, each type of layer will be explained in the next sections.

Since neural network architecture searches should not have a large search space, to prevent that, only a few different types of blocks are introduced in this paper. These were firstly

proposed in [Zhu and Jin (2020)] and reused in this paper with a slight change in their architecture to benefit the need for pose estimation. They are convolutional [3.5], residual [3.6], inverted residual [3.7], and depthwise separable block [3.8], each one of them containing unique operations. Every block has two different variants, first is normal, and the second is an upscale variant. With the first one, the number of channels is preserved, while the second upscales, more specifically, doubles the channel size. To model a fully functional unit, those blocks are connected to the so-called master module, which will be explained later in this thesis.

3.5.1. Convolutional block

First block is simple convolutional (figure 3.5). Although research [Zhu and Jin (2020)] proposes working with an identity block, which acts as an empty block, when having a small-scale master module it is better to exploit the operation of convolution. Its normal variation is simple convolutional block dependable on kernel size and same padding, while the upscale alternative has two paths. Each of the two paths is convolution on its own. Finally, these parts are concatenated and adapted with batch normalization and parametric ReLU. Output has two times broader channel size than input does.

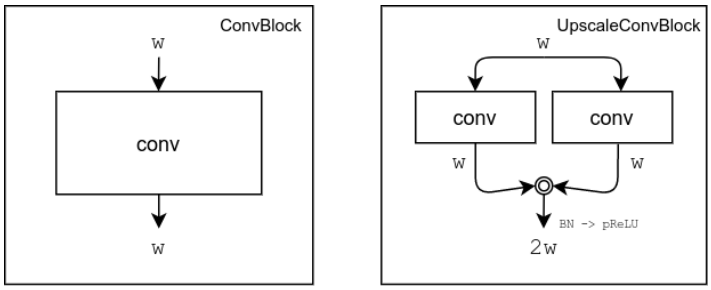


Figure 3.5: Image showing normal convolutional block (left), ad its upscale variant (right)

3.5.2. Residual block

The next described block is residual. It is a widely and commonly used block in neural network design. Residual block in its normal version features two subsequently connected convolutional blocks. Such a design can be seen in a typical ResNet structure [He et al. (2015)]. Also, the major difference is a skip connection, which can help in simplifying the network and a more stable and faster training process. This way information from the first layer is passed and mixed with processed information after two convolutions and batch normalizations.

On the other hand, an upscale block removes skip connections. It is based on doubling channel width while still keeping residual block architecture.

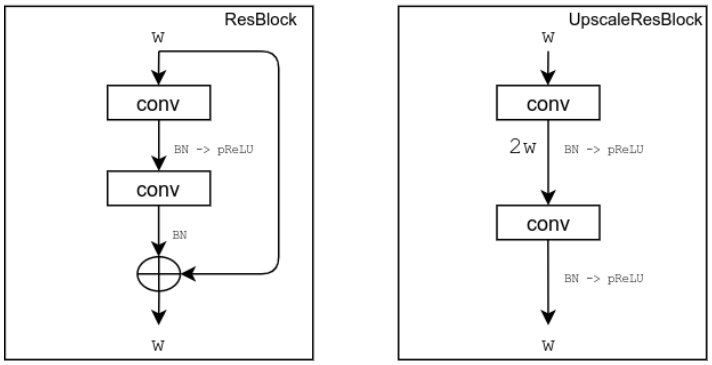


Figure 3.6: Image showing normal residual block (left), and its upscale variant (right)

3.5.3. Inverted residual block

An inverted residual block is a specific one. It features an "inverse bottleneck" structure. The reason why is called inverse bottleneck is that it features expanding dimension in the first 1x1 convolutional layer, moving towards depthwise convolution which is able to catch multiple channel features and in the last part, another 1x1 convolution is there to reduce dimensionality and mix output with the residual connection.

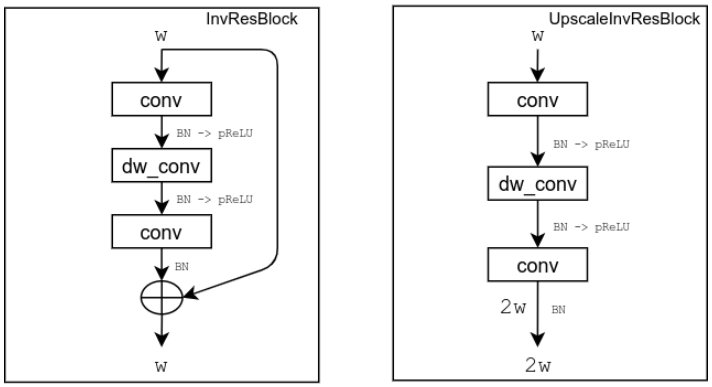


Figure 3.7: Image showing normal inverted residual block (left), and its upscale variant (right)

Again, as it was the case with a simple residual block, this block in its upscale variant features the same architecture, but without skip connection.

3.5.4. Depthwise separable block

This block, as the name says, features more depthwise convolutions. Although it looks that it may be computationally challenging, these depthwise convolutions consume less computational power than common ones. This block has almost the same architecture as the residual one, only with the addition of depthwise convolution layers. On the other hand, an upscale depthwise separable convolutional block features the same connections with the introduction of parametric ReLU on output and missing skip connection.

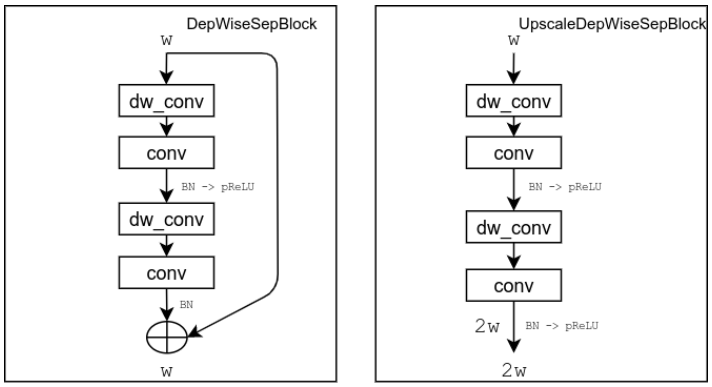


Figure 3.8: Image showing normal depthwise separable convolutional block (left), and its upscale variant (right)

4. Heuristic methods for architecture search

Manually designing deep neural models can give headaches for the creators. With the growth in computational power and a huge amount of accessible computational units, attention for the automatic search for best model architecture has grown. Nowadays, many heuristic methods are used in terms of neural architecture search. Maybe the most common one in "reinforcement learning" [Zoph and Le (2016)], a machine learning method where agent evolves with trial and error technique, each followed by a simple reward strategy. Another approach, which may be seen in this thesis, is using an evolutionary strategy in terms of architecture search. For detailed explanation of neural architecture search, three different issues need to be addressed and these are: **search space**, **performance estimation strategy** and **search strategy**.

Research has been conducted to divide two different types of architecture search. First is searching in macro space, where network scaling is important and there is a need to optimize width, depth, or input resolution [Tan and Le (2019)]. Roughly said, macro search tends to find the most suitable parameters for previously described "master module". On the other side, a micro search space contains different operations that one module may include. Optimizing hyper-parameters for these operations is also a task for the micro search algorithm.

An important step in designing the algorithm for finding the best model is constructing a fitness function. It is an underlying objective function which is usually too complex to solve with common methods of optimizations. These functions can be non-linear, non-smooth, discontinuous, etc. and their dimensionality can be high. Furthermore, the evaluation of this complex function may be time-consuming, and it is important to have as few evaluations as possible. Because of that, a proper optimization algorithm must be applied to find a global optimum.

Such functions are a perfect fit for metaheuristic methods. When dealing with complex functions, heuristic methods are a smart way to bypass heavy function evaluation. To address the ruggedness of objective function, the population methods are used, mostly in terms of

the evolutionary algorithm (Algorithm [1]). These algorithms are biologically inspired. With the biological process of recombination, mutation, and generation, the goal is to find the best possible individual, i.e individual that matches global optimum. An individual is, the usually numerical representation of solution.

Algorithm 1 Population based algorithm

```
procedure SEARCH
    pop_size ← number_of_individuals
    pop ← initialize_population(pop_size)
    for max_iter do
        pop ← evaluate(pop)
        new_pop ← update_heuristic(pop)
        pop ← new_pop
    end for
    return pop[best]
end procedure
```

Given the algorithm above, constructing one solution representation (individual) may be the hardest part of the algorithm. Traditional mapping usually includes binary or float type representation. In this work, a vector of floating-point numbers will be used as a representation of the solution. This choice makes it easier to facilitate common mutation operations, as well as implementing other search techniques.

4.1. Architecture representation

Previously mentioned macro and micro search spaces both have their representations. Macro search space includes basic knowledge about a model, its width, depth, and extraction layer from the transfer network. On the other side, the micro search space is designed as a block-structured neural network. To distinguish these two types of representations, total search space includes:

1. Transfer extraction layer (*macro search*)
2. Scaling factors (*macro search*)
3. Number of layers (*macro search*)
4. Operational blocks set (*micro search*)
5. Operational parameters - kernel size, stride, padding... (*micro search*)

To make it easier to work with, a few predefined sets of hyperparameters are prepared in macro search space. As it is previously explained, equation [3.8] defines the behavior of these parameters. Another task is to define the best performing extraction layer from the transfer network, i.e. the fingerprint extraction layer that is used as a network input. With these sets of variables, since there are only a few of them, a simple grid search (checking each combination of parameters) is performed, and best is picked for baseline model for micro search. Macro search yields set of **width scaling**, **depth scaling** and **extraction layer** variables.

Micro search on the other side has a more complicated structure. Architecture is made as a set of chained blocks. We distinguish two different types of blocks. First are "normal" types of the block that process the information and output the same number of input features. On the other hand, upscale blocks upsample the information with the introduction of larger channel size. To reduce the search space we use a set of previously used and designed blocks. As can be seen in figure [4.1], the design of blocks is pre-defined, while each block is then populated with its parameters.

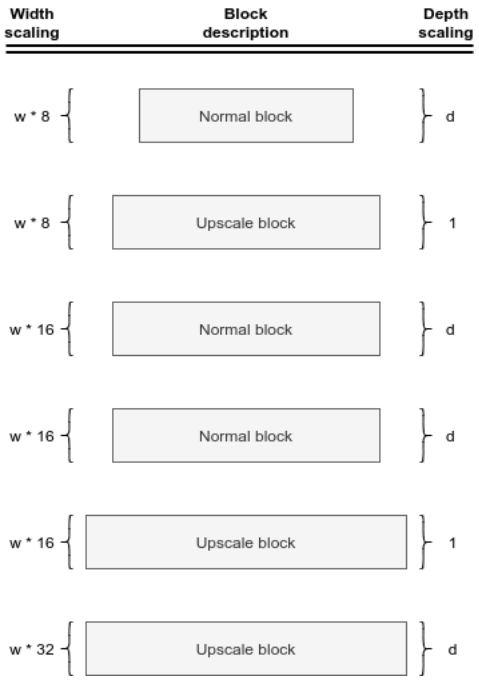


Figure 4.1: "Master module" architecture. With "w" is described model's width scale while "d" depicts the model's depth scale

A solution instance in this work is designed as the array of doubles. Formally speaking, the representation $\mathcal{R}$ of the neural network can be written as:

$$\mathcal{R} = \{b_1, b_2 ... b_n\} \cup \{p_1, p_2 ... p_n\} \cup \{s\} \tag{4.1}$$

Symbols of b_i stand for different used blocks, as it can be noticed in the chained block

structure above. The symbol p_i depicts parameters used in a block b_i, usually only kernel size. Last one to mention is s, a 2^N scaling factor from equation [3.8]. To map from this kind of genotype to the double array, a trivial mapping technique is used. Each of the variables is used as a floating-point in ther interval $0 - 1$, where the interval is split according to the size of the search space, where a position in the interval determines the used variable. For example, in the search space of 4 different kernel sizes, interval $0 - 0.25$ is intended for the first kernel, interval $0.25 - 0.5$ to the second, etc. Illustration of such individual can be seen in figure [4.2]. This allows easy usage in the overall heuristic algorithm. Although, on the other hand, this might lead to the problem where jumping from the first block to the last one requires moving through the various others. That problem may lead to the objective function being rugged and may slow the learning process.

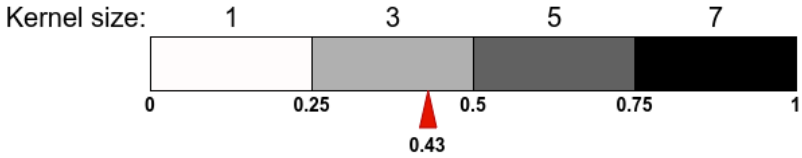


Figure 4.2: Illustration of the kernel size choice. In this example, a kernel size 3 is picked

4.2. Problem formulation

The problem is formulated as the multi-objective search, but its exploration strategy does not include multi-objective algorithms. Another approach is to derive a complex goal function that connects each of the various objectives. In this case, we want to model a function that will be able to maximize precision and reduce the number of floating operations needed. Let LOSS(m) denote surrogate error function of the model m. Also, let FLOPs(m) address the number of floating operations for a given model. Floating operations are just an approximation of the real model's latency on the device. To describe the model's precision, a proper loss function is introduced in the next chapter.

The goal is to find criterion optimal solutions. In the usual situation, this means finding Pareto-optimal solutions, where optimality refers to finding solutions where no criterion can be better off without making another criterion worse off. Some algorithms specialize in these tasks, however, without having a theoretical insight into optimization criteria, in this work weighted criterion function [Tan and Le (2019)] is used to obtain solutions in which both terms are optimized at once:

$$\text{minimize}(m)\text{: } LOSS(m) \times (\frac{FLOPs(m)}{T})^w \tag{4.2}$$

In this equation, T denotes the pre-defined desired number of FLOPS, while w is a constant that determines how important is the complexity factor. Since it is problem specific constant, w is derived empirically to match the target number of FLOPs. It is set to a constant value of 0.06. Having such an objective function, we hope to minimize both model loss and the number of floating operations to derive the best conceivable solution, or to come up with solutions that satisfy both equations.

4.3. Heuristic algorithms

4.3.1. Naive approach - random search

To address the issue of search strategy few different algorithms are defined. The first approach is a naive one. An obvious algorithm for searching through state space is a random algorithm. Just by random sampling points in the distribution, we hope to find the best solution. This algorithm is tremendously simple, but it is not commonly used since search space can be wide. Each individual is randomly created without any domain information, and therefore there is no previous insight into how well will next individual behaves.

With this in mind, as search space dimensions grow, the random search algorithm performs worse. The reason behind that is random sampling through large dimensionality space without proper direction policy. In small dimensional space, a random search might succeed in finding a global optimum. When underlying optimization function is too complex, without any regularity on its behavior, in the some cases, the random search can perform similar to other heuristic techniques. Furthermore, a random search can have its benefits in avoiding local minima. Nevertheless, the convergence rate of such an algorithm is disastrous because there is no way to direct the algorithm.

That being said, in this work, a naive approach is implemented to work like it is explained above. In each iteration, a random population is sampled and the best performing individual is preserved in solution. This way, we don't lose the best solution in the next iterations.

4.3.2. Covariance Matrix Adaptation Evolution Strategy

Covariance matrix adaptation evolution strategy [Hansen and Ostermeier (2001)] or CMA-ES is evolutionary strategy for non-linear black box function optimization, meaning it is used for difficult optimization where underlying function is not known. It is a derivate free method for numerical optimization. A method depends on sampling from multivariate normal distribution:

$$\mathbf{x_i} \sim \mathbf{m} + \sigma \cdot \mathcal{N}_i(\mathbf{0}, \mathbf{C}) \tag{4.3}$$

In terms of evolutionary strategy, recombination amounts to selecting a new mean ($\mathbf{m}$) value for distribution, where the mean vector represents the favorite solution. Sigmac (σ) stands for step size in each iteration. Lastly, $\mathbf{C}$ stands for covariance matrix which determines the shape of the distribution and it is hard to update from iteration to iteration [Hansen (2016)].

The goal is to properly adapt distribution parameters during the evolution process to converge the population mean into global optimum. In the first step, population of size λ is generated by sampling from normal distribution. In the next step, mean is a weighted average from μ selected individuals from previous step. These individuals are selected according to their fitness value in previous step, where ($x_{i:\lambda}$ stands for *i*-th ranked individual:

$$\mathbf{m}^{(g+1)} = \mathbf{m}^{(g)} + c_m \sum_{i=1}^{\mu} w_i (x_{i:\lambda} - \mathbf{m}^{(g)}) \tag{4.4}$$

In the equation [4.4], weights w_i adds up to a sum of 1. Another unknown constant is c_m, a learning rate, usually set to 1. To update the other two algorithm's variables, we need to introduce parameters p_σ and p_c which both stand for "evolutionary paths", a direction in which distribution is updated.

Step size (σ) is updated subsequently. Firstly, its corresponding path is updated. How the author came to formulas stated upfront is explained in the previously cited tutorial. In this work, we aim to explain the aspect of each equation.

$$\mathbf{p}_\sigma{}^{(g+1)} = (1 - c_\sigma)\, \mathbf{p}_\sigma{}^{(g)} + \sqrt{c_\sigma(2 - c_\sigma)\mu_{eff}}\, \mathbf{C}^{(g)\,-1/2}\, \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \tag{4.5}$$

In this equation, c_σ denotes the decay factor which enables us to store some amount of information from previous steps. Notation under root is a normalization constant. $\mathbf{C}^{(g)\,-1/2}$ is the eigendecomposition of the covariance matrix in a single step. It is used to map the mean difference to the new generation space. An important factor is a difference between mean in consecutive generations, which is divided by the current step size.

$$\sigma^{(g+1)} = \sigma^{(g)} \times \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|\mathbf{p}_\sigma^{(g+1)}\|}{E\|\mathcal{N}(\mathbf{0}, \mathbf{I})\|} - 1\right)\right) \tag{4.6}$$

Step size is updated with the formula above. It depends on the updated length of evolution path, divided by the expected value for standard normal distribution, depending on the number of samples. If the resulting path length is larger than expected, a step will increase. Constants c_σ and d_σ are used for exponential decay.

The major role in CMA-ES takes the update of a covariance matrix. Updating this matrix is done via two different updates: rank-μ update and rank-one update. The first one is just a maximum likelihood estimation of the covariance matrix in sorted best performing individuals. The other is a rank-one update, which is constructed from the covariance matrix evolution path, i.e history aware distribution updates.

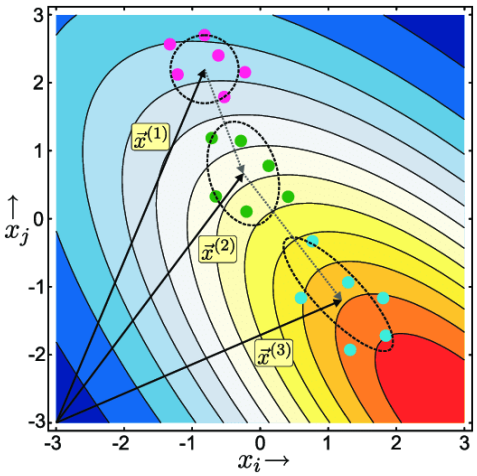


Figure 4.3: Illustration of three iterations of CMA-ES algorithm. Image shows progression in covariance, mean and variance adaptation in three steps. [Shir et al. (2011)]

$$\mathbf{p}^{(g+1)} = (1 - c_c)\,\mathbf{p}_c^{(g)} + \sqrt{c_c(2 - c_c)\mu_{eff}}\,\frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \tag{4.7}$$

The final update can be seen below, with the first part of the summation being decay factor from the previous generation, second being rank-one update, and last being rank-μ update. This enables the covariance matrix to fit the exploration in a certain evolution step.

$$\mathbf{C}^{(g+1)} = (1 - c_1 - c_\mu \sum w_j)\,\mathbf{C}^{(g)} + c_1 \mathbf{p}_c^{(g+1)} \mathbf{p}_c^{(g+1)^T} + c_\mu \sum_{i=1}^{\lambda} w_i \mathbf{y}_{i:\lambda}^{(g+1)} \left(\mathbf{y}_{i:\lambda}^{(g+1)}\right)^T \tag{4.8}$$

This method behaves well in conditions of black-box optimization, where the objective function is not known. It majorly benefits from global exploration in the first steps and local exploitation in final steps.

4.3.3. Genetic algorithm

A typical evolutionary algorithm is a "genetic algorithm". It is a metaheuristic approach inspired by the process of natural selection. A goal is evolving generations (population in each iteration of algorithm) to progress with good quality of genes. We start by initializing a random population that contains different possible solutions to the problem.

In standard genetic algorithm with methods of natural selection (mutation and crossover), an old set of worst-performing individuals is replaced with better performing newly created individuals. This is called elitism, where the genetic code of well-performing individuals is preserved in the next epoch. In this work, to breed a new generation, only the process of mutation is used, because there is a hypothesis that crossover wouldn't benefit in terms of

the creation of new individuals. This comes upon a fact that mutation makes larger steps in the search steps.

Firstly, the population is sorted by the lowest objective function values. Half of the population is preserved in each epoch. The other half of the population is generated with the process of selection and mutation. For the selection process, simple tournament selection is used, while mutation is done via Gaussian mutation.

– **Tournament selection:** this type of selection exploits genes of best individuals in a structure called "tournament". The algorithm repeats itself several times firstly by selecting k individuals from the population. These k individuals are sorted and the best of them is taken for mutation. This is the way of preserving gene material. The stochastic element of this type of selection is a random choice of an individual from the population. Selection pressure can be easily adjusted by variation of the number of chosen individuals (k). With more individuals in the selection process, there is a higher chance of selecting globally best.

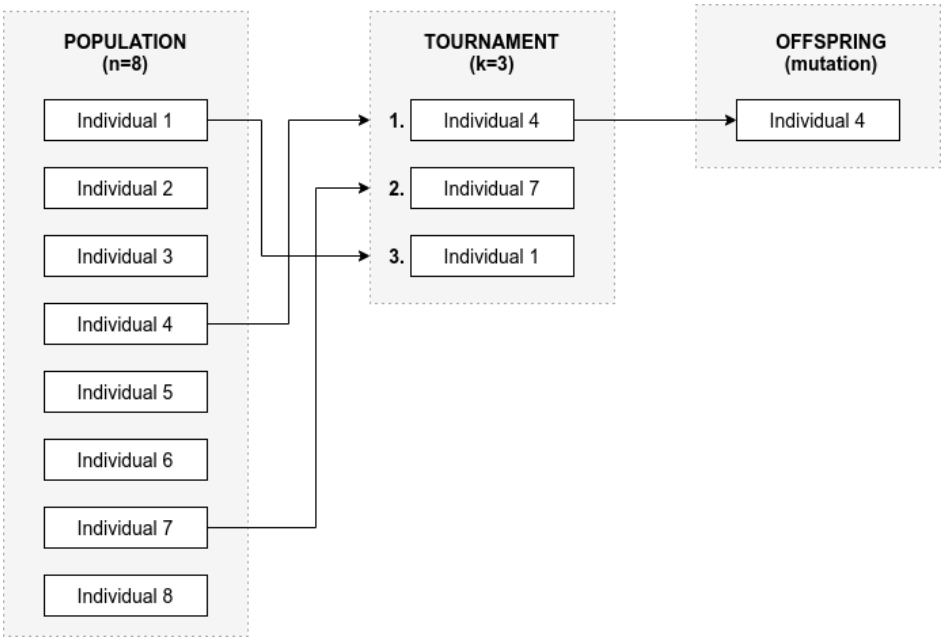


Figure 4.4: Tournament selection process

– **Gaussian mutation:** this type of mutation implements the addition of random noise from a gaussian distribution. Parameters of this distribution are predefined and defined empirically (standard deviation and mean). To keep the information from the previous epoch, each gene (usually one floating point number) is mutated with mutation probability ($p = 0.3$).

In each epoch, the best individuals are preserved, therefore, this directs the algorithm to the convergence.

4.3.4. Particle swarm optimization

Another population-based technique of heuristic optimization is particle swarm optimization [Kennedy and Eberhart (1995)] (PSO). As can be seen from its name, its heritage comes from natural inspiration. To determine nomenclature, having a population (here called "swarm") means having a set of individuals (presented as "particles"). The major contribution of this work is the introduction of particle dependency. Each is updated depending on local and global particle performance.

With a few simple rules of moving through the search space, PSO exploits great results. The first rule is keeping the best known found solution for each of the particle histories. Next is the storage of the best known global solution. The final rule is determining the direction where each particle should be moved to. Keeping both local and global solution can help in locating local and global objective function's optimums.

Algorithm 2 Particle swarm optimization

procedure PSO
    $\mathbf{p} \leftarrow$ random_initialize_population(**pop_size**)
    for max_iter **do**
        for p_i in $\mathbf{p}$ **do**
            if $f(p_i) \leq f(pb_i)$ **then**
                $pb_i \leftarrow p_i$
            end if
            if $f(p_i) \leq f(gb)$ **then**
                $gb \leftarrow p_i$
            end if
        end for
        for p_i in $\mathbf{p}$ **do**
            $v_i^{t+1} \leftarrow \omega \cdot v_i^t + c_1 r_1 (p_i - pb_i) + c_2 r_2 (p_i - gb)$
            $p_i^{t+1} \leftarrow p_i^t + v_i^{t+1}$
        end for
    end for
    return gb
end procedure

Having formally written the algorithm above [2], let's discuss its behavior. We start by initializing a random population of particles (p_i), setting the locally best solution to the initial one. Velocities (v_i) are set to be random directions where particles are heading to.

In each iteration, we update globally and locally best solutions found by particles and update its position in search space. Point is updated concerning direction it is heading in, random movement in direction of local search (best-known location of the particle), and

random movement into direction of the globally best location. Random movements are determined by uniform random variables r_1 and r_2 which are bounded 0-1. Constants c_i decide the impact of different search strategies, local exploration, or global exploitation, respectively. To adjust the velocity factor in the main formula, an acceleration (ω) is added to limit the impact of the velocity.

This simple technique yields great results in black-box optimization because of the great sampling of search space, and therefore, here is used as a neural architecture search method.

4.3.5. Powell's derivative free optimization (BOBYQA)

Not only population base algorithms are popular among the derivative-free optimizations, but also single-state methods. One of these techniques is designed by Michael J.D. Powell by the name of *"Bound Optimization BY Quadratic Approximation"* or simply, BOBYQA [Powell (2009)]. To fit the need of this work, an algorithm is adapted to work with the other population-based methods.

Running the single-state algorithm as a population-based heuristic method can be structured as multiple single-state runs. This has the effect of running the population of random and independent individuals. The downside of this technique is dependence on the random initialization of starting points.

BOBYQA is a derivative-free method, which makes it a solver for the constrained problem without the usage of the derivatives of the objective function. It starts with the construction of the objective function approximate via a quadratic approximation using up to $2 \cdot n + 1$ points, where n denotes the dimension of the search space. Using the quadratic model and generated trust regions, a regional subproblem is solved by a careful change in variables and cautiously reducing the trust region. This is not a perfect match for the method in which a minor change of genotype (individual) does not affect phenotype (final architecture). Further implementation details are not part of this work, where the goal is to present the effect of the single-state method.

When combining the population-based approach and BOBYQA, where the absolute goal is to reduce the number of objective function evaluations, the effect of constructing the trust-region might take up the unreasonable number of iterations, which might make this algorithm inferior in the low amount of epochs need to construct the best solution.

5. Optimization pipeline

With all the technical explanations above, a mix of these methods will generate a good ground for the mentioned task. To enhance a transfer learning on the problem of head pose estimation, above mentioned heuristic search methods are applied. Furthermore, standard procedure on training head pose problems is used, with three standard datasets which will be introduced later.

The backbone model used for extracting features is Facenet [Schroff et al. (2015)] with the Inception-ResNet [Szegedy et al. (2016)] architecture trained for the task of face recognition. The assumption is that head pose estimation and facial recognition should use the same features. This way, with little parameters overhead it is possible to obtain pose angles from structured fine-grained features.

That overhead is another model, built on top of the first one. Its architecture is yet to be defined. The first set of experiments is dedicated to determining the scale of the model, set of depth and width attributes that would fit the problem. For that task, the baseline model is used in pair with the grid searching method, which will be explained in detail. Another task is to adapt parameters to determine the best possible micro-architecture. Our goal function is to minimize output and angle difference using the defined loss function. Another goal is to determine architecture with as few parameters as possible, which would be able to generate the same results as larger models. Illustration of such approach can be seen in figure [5.1].

On top of that, various heuristic methods include generative search, covariance matrix adaptation, particle swarm optimization and single-state method are used to find the optimal architecture. Once when architecture is generated, the model is adapted for real-world usage.

5.1. Technical details

A most common way to address machine learning problems is using **Python** programming language. His modularity and an enormous variety of easy to use libraries offer simple usage. Because of its simplicity, numerous different experiments were done.

Timely consuming neural architecture search can be accelerated with the proper hardware. For the task of learning deep models, a graphical processing unit with 8 GB of graphi-

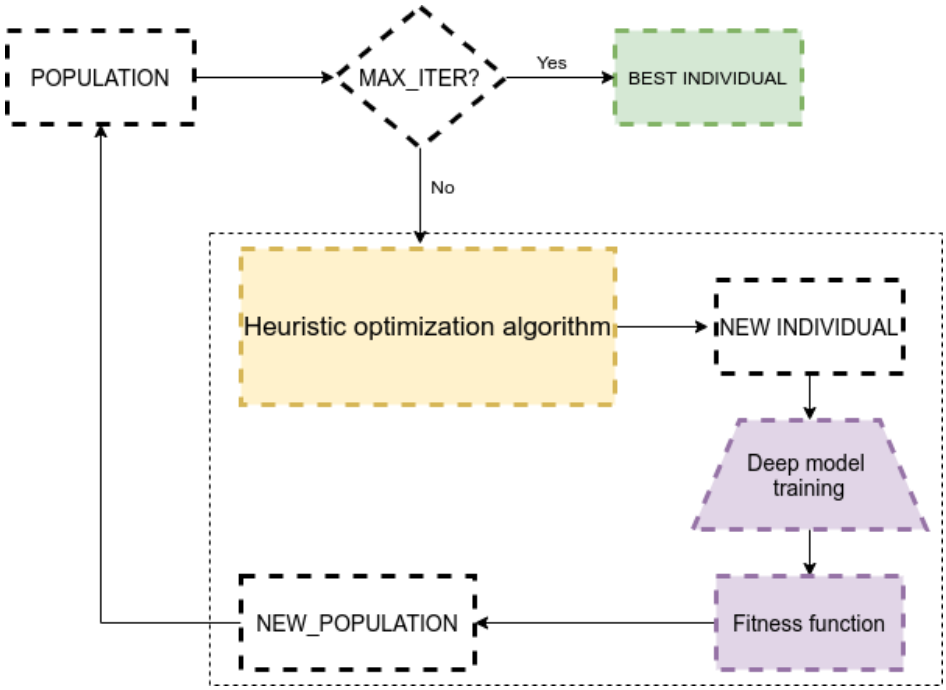


Figure 5.1: Top down illustration of the optimization pipeline

cal memory is used. All training process was done on a single **NVIDIA GeForce RTX 2070 SUPER** graphical processing unit. In each algorithm, the models were trained subsequently.

For implementing deep neural network models, **Tensorflow** was used (version 1.14). It is an open-source library made for machine learning tasks. To deal with images, it is important to mention the image processing library. We used **OpenCV**, the most popular open-source library for computer vision. It is optimized to work with various machines and does not depend on programming language being used on top of it. Another major important library is **NumPy**, a package for scientific calculations in Python. It offers simple API for working with linear algebra operations, as well as dealing with array structures easily.

Heuristic algorithms are partly written from scratch and partly used as include libraries. Such algorithms offer a clean API for optimizing objective function with constraints.This offers easy to use optimization where there is only a need for parameter fixation.

5.2. Datasets

For the task of head pose estimation, the standard procedure includes training on a dataset called 300W_LP and testing its performance on AFLW2000 and BIWI datasets. Illustration of how those datasets visually look like can be seen in figure [5.2]. The goal is to train the model to work on real data. Each one of the training and testing datasets will be explained in subsequent paragraphs.

– **300W_LP** [Zhu et al. (2015)] - derived from the 300W dataset, it contains syntheti-

Figure 5.2: Visual representation of three datasets being used

cally made faces for larger poses. Dataset is a union of several datasets with labeled 68 facial keypoints. A pose is then derived from keypoint transformation into the front-looking face model. It contains 3D generated 61,225 images, that are furthermore expanded to 122,450 samples simply by flipping images.

- **AFLW2000** [Martin Koestinger and Bischof (2011)] - contains 2000 first identities in the AFLW dataset, which stands for faces in the wild with large poses. Examples included in this subset are a non-trivial benchmark for any face modeling method. This dataset includes several conditions of lighting, brightness, and poses which makes it difficult to test on.
- **BIWI** [Fanelli et al. (2013)] - gathered in laboratory settings using the Kinect device. It contains 24 videos of 20 subjects that count up to 15,000 different frames to test on. Dataset offers fine-grained annotations, which were precisely captured with the depth sensors. Since it has better-made pose labels, it is widely used as a testing dataset.

Distributions of these 3 datasets vary (figure 5.3). This may lead to different performance depending on the dataset being used to validate. The major difference is offset in pitch angle distribution mean. This being said, when training on 300W_LP, a model might learn that malicious offset and therefore make mistakes on real-world data. Although the training dataset is gathered synthetically, it contains enough information for generalization in the real system.

5.3. Optimization loop

Evaluation protocol can be easily described from start. The whole pipeline comes up from previously described parts. This chapter concatenates earlier sections and interprets the optimization loop to gather best results on neural architecture optimization.

Optimization starts with the choice of optimization algorithm (random search, generation

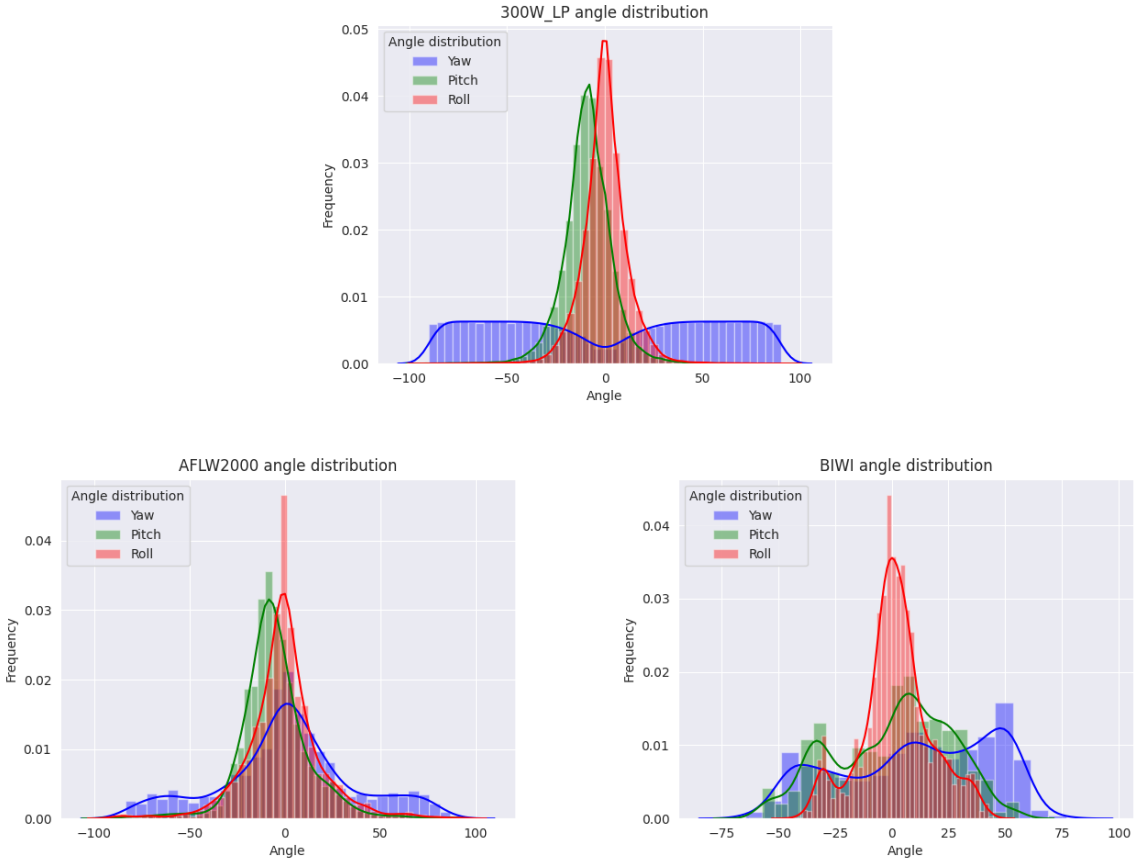


Figure 5.3: Yaw, pitch and roll distribution on 300W_LP, AFLW2000 and BIWI respectively

tournament search, CMA-ES search, particle swarm search, and Powell's search). Each one of the optimization algorithms codes the information of neural networks in the array of doubles bounded from 0 to 1. One example of a single individual can be seen in the image below.

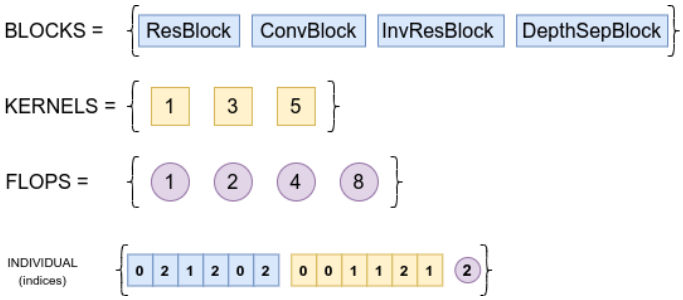


Figure 5.4: One random individual, with possible architecture choices above

For example individual can be represented as the following array:

$$\text{ind} = (0.07, 0.65, 0.33, 0.71, 0.10, 0.67, 0.19, 0.17, 0.51, 0.47, 0.98, 0.41, 0.61) \tag{5.1}$$

These numbers are mapped to indices representation as it is seen in the figure [5.4] (numbers do match with visual representation). Each index is representation in phenotype state,

meaning that index 0 of blocks means ResBlock is going to be applied. The first step is generating a population of such individuals. This population is evaluated with the evaluation algorithm.

This algorithm takes the individual, creates a unique neural network, and starts the process of training. To adapt the weights of a deep model, a pre-defined number of epochs is specified. Learning rates are also constant values since each one of deep models should have the same training hyperparameters. The evaluation depends on the multi-criterion function as was mentioned before. Both the number of floating operations and loss function are optimized. Depending on the choice, certain different regression loss functions are used, advantages and problems are described below:

1. **Mean absolute error:** calculates the absolute difference between true and predicted value. Low magnitude can help the deep model to adjust weights without exploding gradients and there is smaller weight emphasis on outlier examples. On the other side, absolute error ensures that there is no direction of error, which might be valuable information. Because outliers are adjusted with small weight factors, the model might have a huge error on such angles in this case.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - x_i| \tag{5.2}$$

2. **Mean squared error:** the most common loss function calculates the squared error from true labels and predictions. Since huge error is generated with outlier predictions and labels, these do not occur in generalization because they are penalized in the training process. These huge penalties may result in enormous gradients that can deform the process of decreasing error.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2 \tag{5.3}$$

3. **Huber loss:** something in between the previous two losses is Huber loss. It uses the best of the two worlds. If an error is large enough, the average error is used, treating the model with small amplitude samples, while on the other side, if the error is beneath threshold δ, a squared error is used.

$$Huber\ Loss = \begin{cases} \frac{1}{2}(y-x)^2 & \text{if } |y-x| \leq \delta \\ \delta|y-x| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} \tag{5.4}$$

When loss is obtained after several epochs, the fitness function is calculated from the equation [4.2], depending on the targeted number of FLOPs and validation loss. While

training loss can vary from previously mentioned, for validation, absolute error is calculated, because it gives a nice insight into the real difference in between predictions and labels.

After each model is evaluated, using the optimization algorithm, new individuals are created, and depending on the algorithm, inserted into the new generation. In each of the different algorithms, the overall best solution is preserved and it represents the output from the optimization algorithm.

In the next chapter, results generated using this optimization loop will be presented and described between various methods of optimization as well as between currently known mentioned state of the art methods.

6. Results

After a comprehensive explanation of the training process, several different experiments were conducted on the obtained models. In this chapter, both case study on different algorithmic optimization methods and comparison to state of the art are conducted.

In the first section, a concrete set of parameters and operations used in research are written down as an experiment setting. Next on, in the second section, a suitable extraction layer is found. In the third section of this chapter, a search is employed using the network scaling method. A grid search technique is applied to find the best performing scaling parameters. Furthermore, the fourth section brings an analysis of various heuristic algorithm performance. Finally, the last section gains insight into comparison to the state of the art methods, as well as real-world performance illustration.

6.1. Experiment settings

First, we need to define the settings for an experiment. The algorithm of choice first defines the whole population set with a random number of architectures. Different algorithms use a different set of parameters, each chosen set of parameters is defined in the table below.

CMA-ES	
Parameter	**Value**
Population size	10
Max iterations	20
Initial step size (σ_0)	0.3
Genetic algorithm	
Parameter	**Value**
Population size	10
Max iteration	38
Mutation chance (p_m)	0.3
Mutation scale (σ_m)	0.3
Tournament size (k)	3

Particle swarm optimization	
Parameter	**Value**
Population size	10
Max iteration	20
Velocity factor (ω)	0.5
"Particle best" factor (ϕ_{pb})	0.4
"Global best" factor (ϕ_g)	0.6
BOBYQA	
Parameter	**Value**
Population size (no. runs)	4
Max iter	50
Trust region radius (ρ_{beg})	0.3

Table 6.1: Parameters used in different heuristic methods for neural architecture search

Each individual (i.e architecture) is then trained with hyperparameters that were experimentally selected. Parameters for deep learning model can be seen in table below (table 6.2).

Neural network hyperparameters	
Hyperparameter	**Value**
No. epochs	6
Batch size	32
Regularization	L2 (0.001)
Learning rate	$[5 \times 10^{-4}, 2 \times 10^{-4}, 9 \times 10^{-5}, 4 \times 10^{-5}, 1 \times 10^{-5}, 1 \times 10^{-5}]$
Actvation function	PReLU

Table 6.2: Parameters used in deep model training process

Since pre-trained Inception-ResNet modeled for the task of face recognition was a network of choice in the feature extraction, its selected layers are predefined inputs into our deep model. Furthermore, the extraction network requires an input image of size 160x160, which is considered as the high resolution in the subtask of face modeling. To create as many different examples as possible, training images are saved as RGB images with the size of 190x190 which are randomly cropped to adapt input size. Moreover, during the training process, random color transformations are subsequently applied to images, such are change, in contrast, hue, and saturation. The last step in image processing is prewhitening of an input image, subtracting the mean and normalizing the range of the pixel values for input images,

which makes the training process easier.

6.2. Extraction layer search

As was previously explained, determining extraction layers is done by simply acquiring different layer outputs from the feature extraction model and running several training processes on a simple baseline model. The extraction layer defines the input size and features being extracted. The premise of this work is to find the layer that encapsulates head pose information into small resolution size which is an input into the head pose estimation model. To have a visually pleasing representation of features in each of the major parts of Inception ResNet, further heatmap images are obtained (figure 6.1).

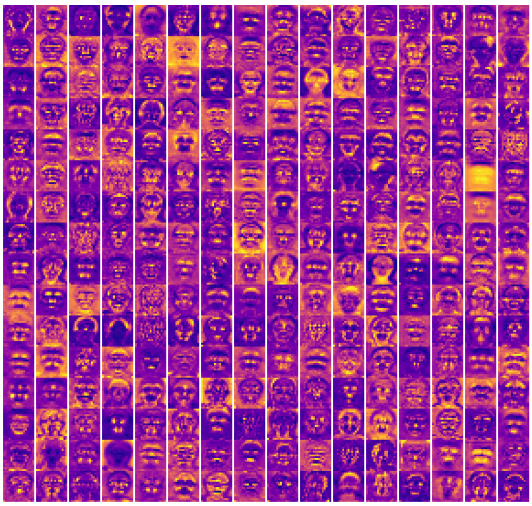
In the features that are lower in a hierarchy, the shape of the face is visible with some part being overly exposed. Moving towards middle layers, only major important head parts are being visible, while high representation in higher-level features tends to lose visually pleasing results and become non-interpretable.

Furthermore, in the table [6.3], different extraction layers are expressed alongside with mean absolute error on the validation set. Used layer names are similar to the ones described in figure [3.4]. Although, because of the different use-case, the grid sizes are not equivalent to the ones in the original paper. Modules A, B, and C have a grid sizes of 17x17, 8x8, and 3x3 respectively.

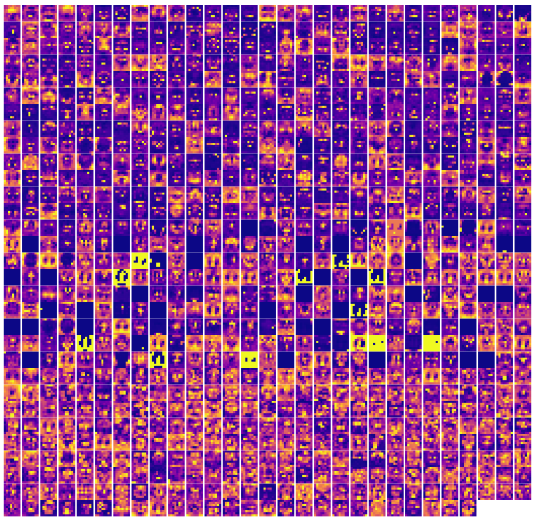
Extraction layer name	**Loss value (MAE)**
Module A - layer 3	4.153 ± 0.049
Module A - layer 5	3.998 ± 0.036
Module B - layer 1	4.018 ± 0.060
Module B - layer 3	3.967 ± 0.031
Module B - layer 5	3.920 ± 0.047
Module B - layer 7	3.889 ± 0.097
Module B - layer 10	$\mathbf{3.793 \pm 0.024}$
Module C - layer 1	4.031 ± 0.028
Module C - layer 3	4.105 ± 0.023

Table 6.3: Resulting loss values for different extraction pinpoints

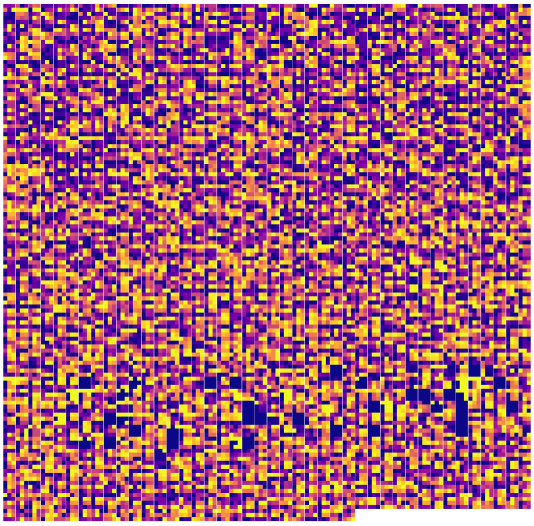
The best performing layer is the 10th layer of middle block B, which might seem like a surprise because lower level features hide activations from which humans can determine pose angle. An important thing to acknowledge, when using such a transfer learning procedure, importing features from low-level layers means having faster inference.



(a) Activations in layer from module A



(b) Activations in layer from module B



(c) Activations in layer from module C

Figure 6.1: Output activations from layers in different position in Inception-ResNet face recognition model

6.3. Macro scaling search

To conduct an efficient scaling search, an experimental setup with a variety of different scales is applied. After determining an extraction layer, the baseline model can be scaled to different sizes to resolve what type of scaling can affect the learning process and accuracy.

Experiment on different types of scaling incorporated usage of grid search technique. Defined in equation [3.8], for various different scaling factors (1,2,4 and 8), grid search is performed on different scales of width and depth. Since the scaling model depends on the linear depth and squared width, only round factors of depth are taken into consideration, while the width is calculated as a square root of scaling factor multiplier and depth. This fact allows us to perform a grid search on a small number of static parameters and include best

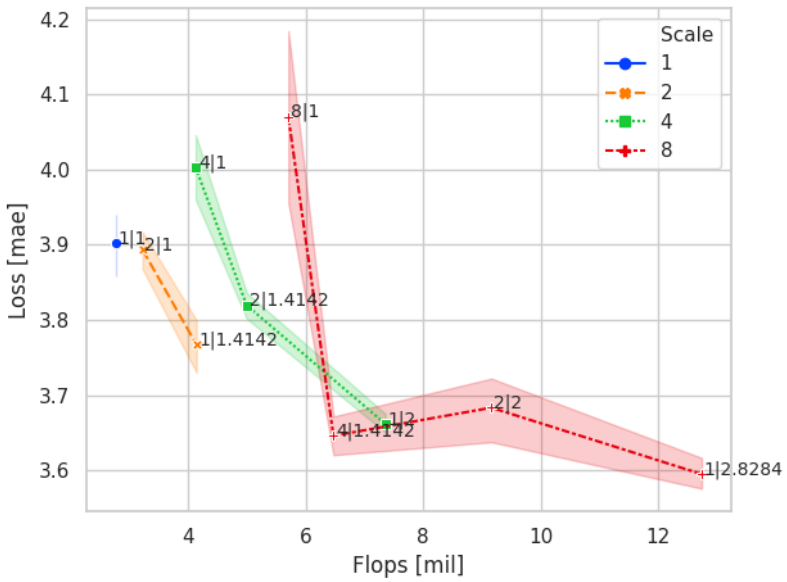


Figure 6.2: Macro architecture search on scaling parameters, number on left side denotes depth scaling, while the right is width scaling

performing into a micro-architecture search.

The results of such search were clear, scaling on depth didn't improve the model's performance, while keeping the depth with a wider model allowed the reconstruction of features from the extraction model. This research differed from EfficientNet [Tan and Le (2019)] results, where compound scaling generated better performing models. In this case, using the resources to broaden the model's width results in better precision.

The reasoning behind such a result might lay in the fact that input in such a network isn't an original image, but extracted features. Those features are low resolution and large channeled data, which is the main reason why investing in the model's depth doesn't produce intuitive results.

6.4. Comparison of heuristic methods

After gaining insight into the extraction layer being used as well as information about best performing scaling factors, heuristic algorithms can be used for architecture search. Each algorithm is set to be run in the predefined number of objective function evaluations, which in this case is defined as the model's accuracy and latency after training.

A multiple-criterion objective function was being used as a minimization goal function for the algorithm. Each algorithm produced 200 new individuals split into a different number of epochs. Figure [6.3] brings a breakdown of the performance in terms of convergence.

All algorithms are tested in the same conditions. The first one was the CMA-ES. In

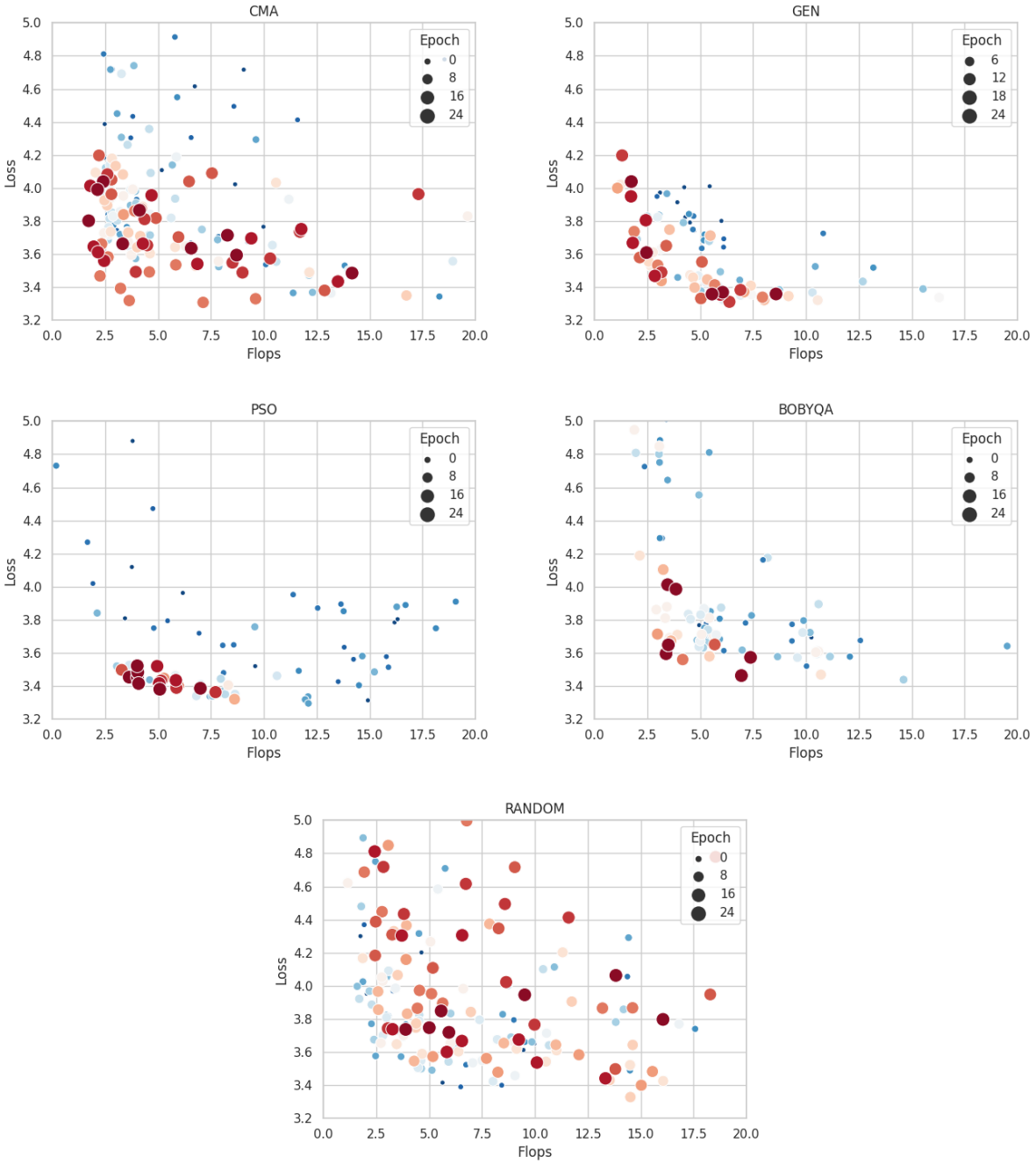


Figure 6.3: Behavior of heuristic neural architecture search algorithms. Color and scale define epoch in which individual is obtained. Earlier epochs have colder colors, while later epochs have warmer ones.

terms of convergence, this algorithm certainly pulls individuals on both criteria, preventing error, and a large number of floating operations. We can spot that large variance diversifies individuals in later epochs, but keeps them on track for both criteria. This can be seen in graphs that describe the behavior of PSO and Genetic algorithm as well. In their case, this phenomenon is even more clear, the convergence rate is satisfying because the low amount of epochs brings large pressure on both criteria.

Previously described behavior can not be applied to the BOBYQA. This algorithm stucks in local optimum and because of local exploration in the first epochs it does not have enough information about space around him. Larger changes in the direction of exploration should make this algorithm better-performing. This can be done via enlarging the starting trust-region radius.

Last to mention is the random algorithm. No information about the epoch is important for this one because of its random nature. On the other hand, a comparable best performing individuals can be found among the ones generated by other algorithms. This result can mislead to the conclusion that each run would be as satisfying as this one. Reality is a bad convergence rate and no guarantee that more iterations would generate a better solution.

We can conclude that, when it comes to neural architecture search, fast and stable results can be gained using the PSO or genetic search.

6.5. State of the art comparison

When best performing individuals are gathered with a neural architecture search, they are compared with other state-of-the-art methods. Talking about the state of the art methods, multiple are mentioned across most-cited papers in this area. The first one is **Dlib** [Kazemi and Sullivan (2014)] which is a face modeling library that contains landmarks detections that are used to predict pose. Another 2D face landmark extractor is **FAN** [Bulat and Tzimiropoulos (2017)]. Authors of the 300W_LP dataset have come up with a convolutional network 3D fit method called **3DDFA** [Zhu et al. (2015)], which aligns 3D landmarks to estimate pose. Next are landmark-free methods, first is **Hopenet** [Ruiz et al. (2018)], a ResNet architecture trained with the crossover of regression and classification loss applied on angles. Last, the best performing is **FSA-Net** [Yang et al. (2019)], a fine-grained features extractor with implementation of Capsule module.

First, we will analyze one of the best performing individuals in terms of generated architecture, after fine-tuning in a larger amount of epochs, for obtaining better results.

Architecture starts with a PReLU layer, a beneficial pre-processing for face image features. Dimensionality reduction is done via the convolution layer with 8 channels. The baseline model is scaled 4x, meaning channels are 2x broader than baseline. Moving next,

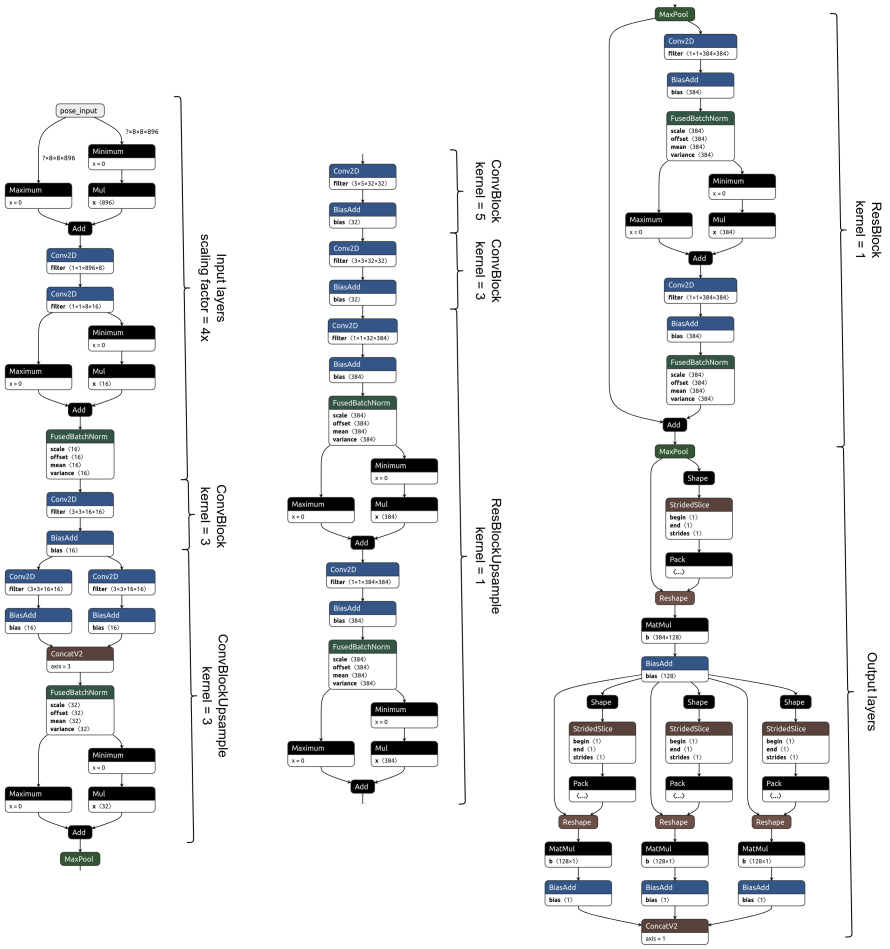


Figure 6.4: Presentation of best performing model, stacked from left to right

4 simple convolutional and 2 residual blocks are concatenated sequentially. Blocks are concatenated with maximum pooling layers. This architecture was best performing architecture in a search performed via a genetic algorithm.

It is important to mention that such a simple structure generates results that are close to state of the art. Other close-performing individuals have diversified architectures, with differences from the scaling method to used blocks. The interesting and common thing in networks generated by automated search is using convolutional and residual blocks in normal layers, rare usage of inversive residual blocks, and usage of depthwise separable convolution in later blocks of master module.

After analysis of best performing individuals, best among them are trained across 30 epochs with decaying learning rate. Afterward, they are compared with the state of the art methods. When comparing size, only the transfer module is calculated, while the initial face recognition network is not taken into calculation. This model is compared on both testing datasets, AFLW2000, and BIWI, results are commented in the tables below.

Method	MB	Yaw	Pitch	Roll	MAE
Dlib		23.1	13.6	10.5	15.8
3DDFA		5.40	8.53	8.25	7.39
FAN	183	6.36	12.3	8.71	9.12
SSR-Net-MD	**1.1**	5.14	7.09	5.89	6.01
Hopenet	95.9	6.47	6.56	5.44	6.16
FSA-Net	5.1	**4.50**	**6.08**	**4.64**	**5.07**
1111001121002_GEN	2.3	5.56	6.55	5.29	5.80
3000111212102_CMA	1.7	6.35	6.35	5.32	6.01
0100120200113_BOBYQA	2.8	5.51	6.45	5.28	5.74
2003102200002_PSO	1.8	6.01	6.36	5.46	5.94

Table 6.4: Comparison with the state-of-the-art methods on AFLW2000 dataset

First testing dataset is AFLW2000. With low model complexity, there is a large error between state of the art yaw angle, while pitch and roll are included into satisfying margin of one degree. This result is compared with SSR-Net-MD [Yang et al. (2018)], another transfer learning method, but fine-tuned to specific task, which is not the same technique.

Method	MB	Yaw	Pitch	Roll	MAE
Dlib		16.8	13.8	6.19	12.2
3DDFA		36.2	12.3	8.78	19.1
FAN	183	8.53	7.48	7.63	7.89
SSR-Net-MD	**1.1**	4.49	6.31	3.61	4.65
Hopenet	95.9	5.17	6.98	3.39	5.18
FSA-Net	5.1	**4.27**	**4.96**	**2.76**	**4.00**
1111001121002_GEN	2.3	4.39	6.97	3.43	4.93
3000111212102_CMA	1.7	3.61	8.22	3.18	5.01
0100120200113_BOBYQA	2.8	3.81	8.17	2.89	4.96
2003102200002_PSO	1.8	4.10	8.10	3.28	5.16

Table 6.5: Comparison with the state-of-the-art methods on BIWI dataset

The next dataset is BIWI. This dataset is interesting because labels are extracted in laboratory conditions. That specific attribute makes it more precise than other datasets, but different distribution training datasets makes it harder for inference. High pitch error is explained with a discrepancy between pose distributions across datasets as well as dependency on extracted features.

To visually present how well the model generated with the genetic algorithm performs, labels are generated across some of the most iconic television moments, shown below on figure [6.5].

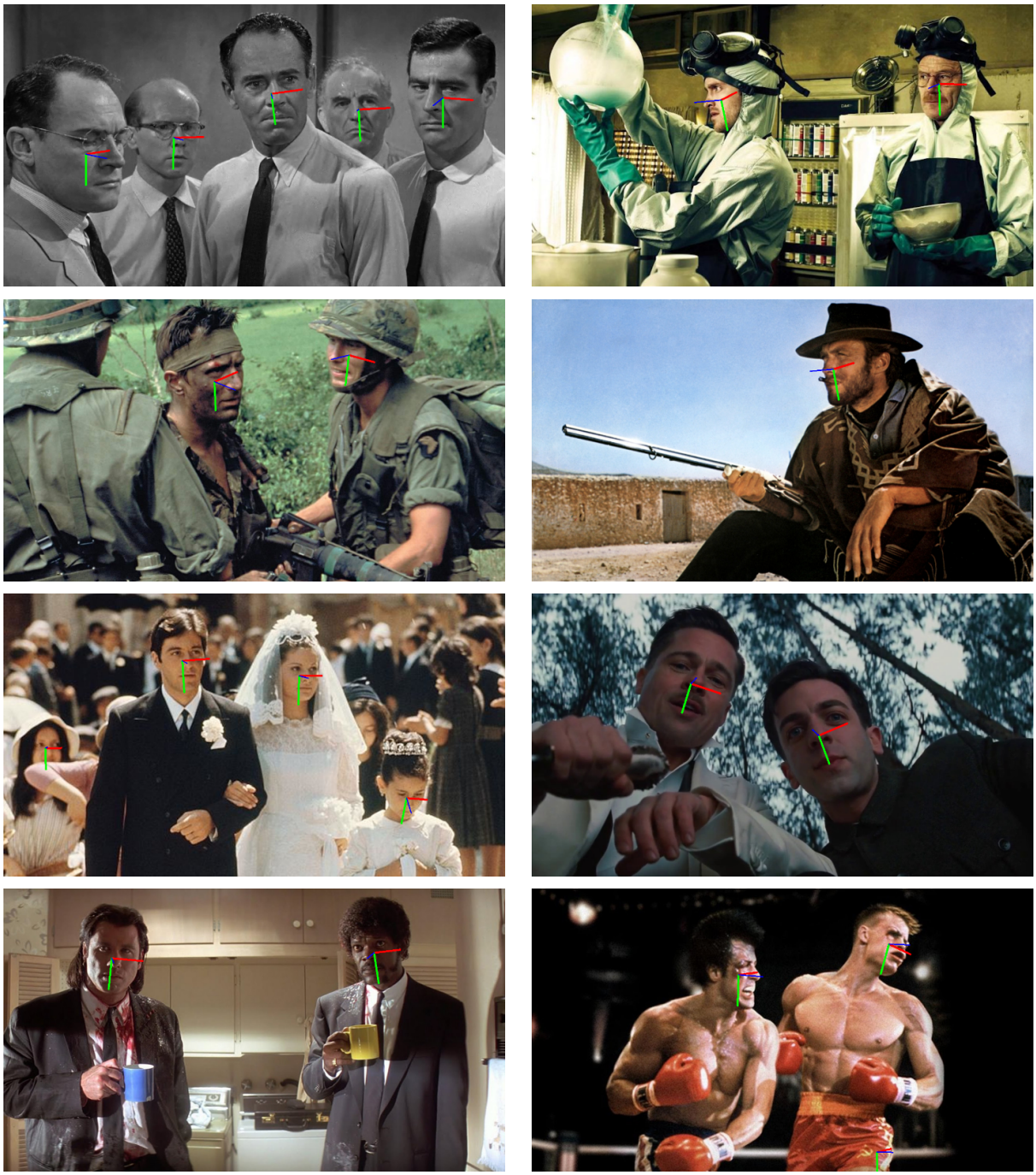


Figure 6.5: Example of pose estimation on images contaning collection of "in the wild" faces, detected with MTCNN.

7. Conclusion

This work employs various strategies for finding optimal deep model architecture for the task of estimating head pose. To obtain both precise and small models, optimization which combines multiple criteria was conducted. Having fine coding structure for the architectures, heuristic methods have shown that their use can help automatize the neural architecture search task. Also, this work has shown that using the related task to head pose estimation, features can be applied from another model as a transfer of knowledge. This enables us to use the same network for various tasks, with minor overhead in the number of parameters. The benefits of that approach lie in the inference speed and memory consumption.

On the other hand, neural architecture search can be time and resource exhausting task. This is the main reason for putting an effort into more advanced algorithms which increase convergence speed and allow generation of models with better precision. Such improvement would enable us to reduce the time spent in finding suitable architecture and therefore apply that resource into different research areas.

Future work in this area would include several different approaches that were not addressed in this work. A study should be done on different feature extrapolation techniques, which might be additionally trained to fit the transferred task. On the predictor side, disentangling the correlation between features and used blocks would help in understanding the underlying cause that helps the model to successfully predict angles. Finally, some state of the art automated architecture search techniques may gain additional benefit in terms of reducing speed or increase in precision.

BIBLIOGRAPHY

A. F. Abate, P. Barra, C. Bisogni, M. Nappi, and S. Ricciardi. Near real-time three axis head pose estimation without training. *IEEE Access*, 7:64256–64265, 2019.

M. N. Asad, İ. Cantürk, F. Z. Genç, and L. Özyılmaz. Investigation of bone age assessment with convolutional neural network by using dog filtering and à trous wavelet as preprocessing techniques. In *2018 6th International Conference on Control Engineering Information Technology (CEIT)*, pages 1–7, 2018.

Adrian Bulat and Georgios Tzimiropoulos. How far are we from solving the 2d & 3d face alignment problem? (and a dataset of 230, 000 3d facial landmarks). *CoRR*, abs/1703.07332, 2017. URL `http://arxiv.org/abs/1703.07332`.

Jiankang Deng, Jia Guo, Yuxiang Zhou, Jinke Yu, Irene Kotsia, and Stefanos Zafeiriou. Retinaface: Single-stage dense face localisation in the wild. *CoRR*, abs/1905.00641, 2019. URL `http://arxiv.org/abs/1905.00641`.

Gabriele Fanelli, Matthias Dantone, Juergen Gall, Andrea Fossati, and Luc Gool. Random forests for real time 3d face analysis. *Int. J. Comput. Vision*, 101(3):437–458, February 2013.

Xavier Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Nikolaus Hansen. The cma evolution strategy: A tutorial. 2016.

Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9:159–195, 06 2001. doi: 10.1162/106365601750190398.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL `http://arxiv.org/abs/1512.03385`.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

Vahid Kazemi and Josephine Sullivan. One millisecond face alignment with an ensemble of regression trees. 06 2014. doi: 10.13140/2.1.1212.2243.

J. Kennedy and R. Eberhart. Particle swarm optimization. 4:1942–1948 vol.4, 1995.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. page 1097–1105, 2012.

Peter M. Roth Martin Koestinger, Paul Wohlhart and Horst Bischof. Annotated Facial Landmarks in the Wild: A Large-scale, Real-world Database for Facial Landmark Localization. In *Proc. First IEEE International Workshop on Benchmarking Facial Image Analysis Technologies*, 2011.

M. Powell. The bobyqa algorithm for bound constrained optimization without derivatives. *Technical Report, Department of Applied Mathematics and Theoretical Physics*, 01 2009.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. ISSN 0033-295X. doi: 10.1037/h0042519. URL http://dx.doi.org/10.1037/h0042519.

Sebastian Ruder. Neural transfer learning for natural language processing. 2019. URL http://ruder.io/thesis/neural_transfer_learning_for_nlp.pdf.

Nataniel Ruiz, Eunji Chong, and James M. Rehg. Fine-grained head pose estimation without keypoints. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Jun 2018. doi: 10.1109/cvprw.2018.00281. URL http://dx.doi.org/10.1109/CVPRW.2018.00281.

Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. *CoRR*, abs/1503.03832, 2015. URL http://arxiv.org/abs/1503.03832.

Ofer Shir, Jonathan Roslund, Darrell Whitley, and Herschel Rabitz. Evolutionary hessian learning: Forced optimal covariance adaptive learning (focal). 12 2011.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

StanfordVisualLab. Cs231n convolutional neural networks for visual recognition. URL https://cs231n.github.io/convolutional-networks/#add.

Y. Sun, X. Wang, and X. Tang. Deep learning face representation from predicting 10,000 classes. pages 1891–1898, 2014.

Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016. URL http://arxiv.org/abs/1602.07261.

Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. pages 1701–1708, 2014.

Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. 2019. URL http://arxiv.org/abs/1905.11946. cite arxiv:1905.11946Comment: Published in ICML 2019.

T. Vatahska, Maren Bennewitz, and Sven Behnke. Feature-based head pose estimation from images. *2007 7th IEEE-RAS International Conference on Humanoid Robots*, pages 330–335, 2007.

Haofan Wang, Zhenghua Chen, and Yi Zhou. Hybrid coarse-fine classification for head pose estimation. 2019.

T. Yang, Y. Chen, Y. Lin, and Y. Chuang. Fsa-net: Learning fine-grained structure aggregation for head pose estimation from a single image. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1087–1096, 2019.

Tsun-Yi Yang, Yi-Hsuan Huang, Yen-Yu Lin, Pi-Cheng Hsiu, and Yung-Yu Chuang. Ssr-net: A compact soft stagewise regression network for age estimation. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1078–1084. International Joint Conferences on Artificial Intelligence Organization, 7 2018. doi: 10.24963/ijcai.2018/150. URL https://doi.org/10.24963/ijcai.2018/150.

Faen Zhang, Xinyu Fan, Guo Ai, Jianfei Song, Yongqiang Qin, and Jiahong Wu. Accurate face detection for high performance. *CoRR*, abs/1905.01585, 2019. URL http://arxiv.org/abs/1905.01585.

Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li, and Yu Qiao. Joint face detection and alignment using multi-task cascaded convolutional networks. *CoRR*, abs/1604.02878, 2016. URL http://arxiv.org/abs/1604.02878.

Hangyu Zhu and Yaochu Jin. Real-time federated evolutionary neural architecture search, 2020.

Xiangyu Zhu, Zhen Lei, Xiaoming Liu, Hailin Shi, and Stan Z. Li. Face alignment across large poses: A 3d solution. *CoRR*, abs/1511.07212, 2015. URL http://arxiv.org/abs/1511.07212.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016. URL http://arxiv.org/abs/1611.01578.

Enhancing transfer learning for head pose estimation

Abstract

Estimating head pose is a useful practice in many industries, especially those of a medical, technical, or automotive nature. Precision and speed of execution are the key things that make these methods useful in real-world practice.

The solution to this task is offered by the current hot topic in deep learning, a machine learning subset. The main contribution of this paper is the transfer learning technique and several different search methods for the best neural network architecture, built from predefined blocks. The results presented in the paper suggest that features extracted from the face recognition method are sufficient for estimating head pose. Also, deep model architecture optimization methods have shown improvements in execution speed and precision by optimizing the multi-criterion objective function. Although this is a task that requires time and memory resources, automated search allows users to change the main focus of research into different problems in the same domain.

Keywords: transfer learning, deep learning, heads pose estimation, evolutionary computing, neural architecture search, heuristic methods

Prijenosno učenje za određivanje smjera gledanja

Sažetak

Određivanje smjera gledanja korisna je praksa u mnogim industrijama, naročito onim koje su medicinske, tehničke ili automobilske prirode. Preciznost i brzina izvođenja ključne su stvari koje ove metode čine prilagođenima korištenju u praksi.

Rješenje za takvu zadaću nudi sve češće korištena grana strojnog učenja, duboko učenje. Glavni doprinos ovog rada je tehnika prijenosnog učenja, te nekoliko metoda pretrage najbolje arhitekture neuronske mreže, sagrađene od prije definiranih blokova. Rezultati koji su izloženi u radu sugestiraju da su značajke izvučene iz metode prepoznavanja lica dovoljne za ekstrakciju smjera gledanja. Također, metode optimiranja arhitekture dubokog modela pokazale su unaprijeđenje brzine izvođenja i preciznosti optimiranjem više-kriterijske funkcije cilja. Iako je zadatak koji zahtjeva vremenske i memorijske resurse, automatizirana pretraga omogućuje usmjeravanje ka drugim smjerovima istraživanja istog problema.

Ključne riječi: prijenosno učenje,duboko učenje, estimacija smjera gledanja, evolucijsko računarstvo, pretraga arhitekture dubokog modela, heurističke metode