NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CX4042: Neural Networks & Deep Learning

Gender Classification (Project D) Project Report

Names: Calvin Hong Wen Kit, Do Anh Tu, Wong Shi Heng
Matriculation Number: U1820310J, U1721947F, U1820100F

Contents

1. Introduction

Neural networks have been gaining popularity in recent years as one of the most powerful machine learning toolsets to solve real life, practical problems. One type of neural network that has become the go-to choice of many machine learning practitioners for image-related problems is Convolutional Neural Network (CNNs).

Given an image recognition problem, one may build their CNNs from the ground up and fine tune the parameters such that it can successfully achieve the required functionality to solve that problem. However, this requires a great amount of empirical observations and therefore takes up large costs in time and resources for training models. Another alternative to this would be to leverage the generalization of award winning CNNs trained on huge multi-class datasets and conduct transfer learning to specialize those models to the domain problem that require solutions.

Our project will be investigating these two approaches, their pros and cons in solving the gender classification problem of in-the-wild human images. This has many use cases such as surveillance and image postprocessing to name a few.

2. Problem Statement

Predicting the gender of a human subject from an image is a binary classification problem where the label is either 'male' or 'female'. Many models have been developed to solve this problem, but most are trained on highly controlled and standardized datasets, which might not represent well the potential domain use case. These datasets consist of strictly frontal aligned images, taken in controlled lighting, with no occlusions or blurriness. On the other hand, images of humans in the wild such as on the internet and social media platforms are hardly that cleaned and controlled. This is where the use case of gender classification probably lies, and therefore a dataset with better representations of human's images in the wild should be considered.

In this project, we will be developing CNN models and testing them on the *Adience dataset* [1]. It has more than 26000 images of more than 2000 subjects collected from the media hosting service Flickr, which consists of real-life images of people taken without much control on environments and posing of subjects.

The main goals of this project are:

1. Build a robust CNN model from the ground up,
2. Experiment and fine tune the hyperparameters of this model and
3. Carry out transfer learning based on a generic, pre-trained CNN architecture to solve gender classification in the wild.

3. Related Works

3.1. Simple CNNs for Gender Classification

Levi and Hassner, 2015 [2] did a quick survey of the various techniques employed by researchers before them in solving this classification problem. Some of these techniques are general machine learning techniques such as SVM, AdaBoost or computer vision approaches. However, a common trend amongst these works are that the training and validation are carried out on datasets with high uniformity, low variances between images. The typical characteristics of images in those datasets are frontal aligned, no occlusions and well-lit of face features being learned to predict the gender. However, such ideal image characteristics are modelling the high variance nature of people's image in the wild. For example, on social media platforms, people's images are from a variety of angle, lighting conditions and image quality (occlusion, blurriness, etc.).

In [2], the authors from *The Open University of Israel* proposed the *OUI-Adience dataset* [1] as a new benchmark dataset for gender and/or age classification tasks. Since the images are collected from Flickr media sharing service, they are more familiar to how people's images in-the-wild look like. The authors also published a lean convolution neural network architecture to accompany their introduced dataset. Their lean network was able to perform relatively well to the challenging dataset and robust to overfitting, a common problem for machine learning on small datasets.

Our work in this project leverage on the benchmark dataset and baseline model published in [2] by experimenting around and fine tuning the hyperparameters with the aim to increase its performance.

3.2. EfficientNets

First introduced in *Tan and Le, 2019* [3], EfficientNets is a family of CNN models that boast state-of-the-art accuracy on common corpora with up to 10 times better efficiency compared to existing implementations before it. Unlike some of the most popular pre-trained models that follow the conventional model scaling practice of arbitrarily increasing the CNN dimensions to improve performance, EfficientNets achieves its impressive showings in both efficiency and accuracy departments through a heuristic model scaling approach: by utilizing a fixed set of compound coefficients that scales each dimension uniformly. This results in models that are not only smaller (less parameters to train as performance is not bounded above by model dimensions) but faster in comparison to traditional methods. Within the family of models, there are currently 8 variations (B0 – B7) that are in order of increasing size (trainable parameters) and performance. Selection of the optimal model to use as a base will depend on the size and nature of the training dataset as well as the compute performance of the system that the training process is being done on.

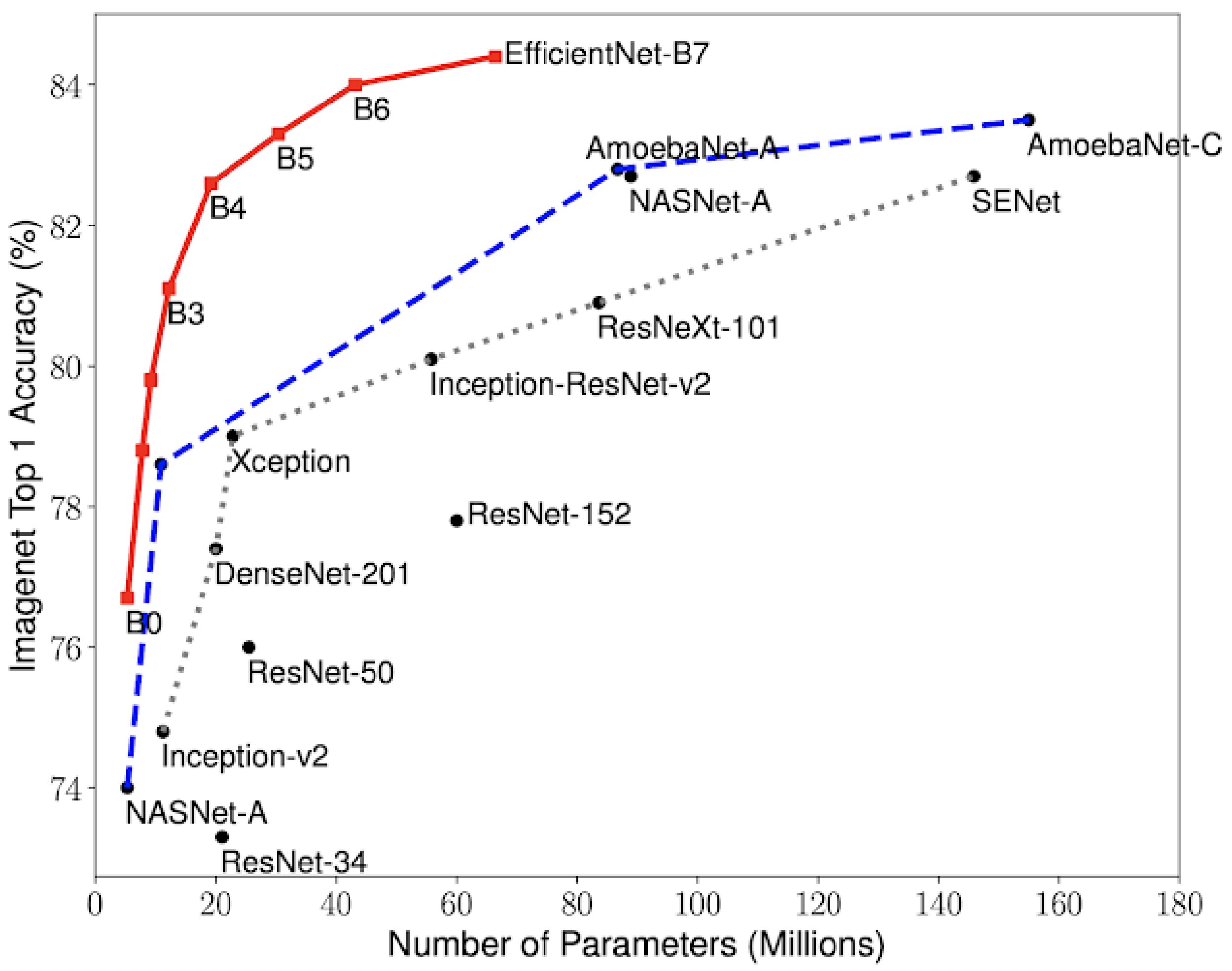


Figure 1. ImageNet CNN size vs. performance comparison

The figure above showcases the performance advantage of EfficientNets compared to some of the most popular CNNs on ImageNet. EfficientNet-B0 is the baseline network developed by AutoML MNAS, while B1-B7 are scaled up variants of the baseline network. It is not hard to see that EfficientNets not only tops the performance chart by some margin, with B7 even achieving never before seen state-of-the-art results, but was able to do so with much smaller number of parameters. Compared to models such as AmoebaNet-A and AmoebaNet-C that have some of the best results previously, an EfficientNet-B5 easily matches their performances while being ~3.2x and ~6.2x smaller respectively, making it less demanding on the system that runs the model training.

4. Methodologies and Implementations

4.1. Data Preprocessing

(Code for this part is found in gender_preprocessing.ipynb)

4.1.1. Dataset's Quality

To determine the necessary steps in preprocessing the images and other data for training, a study of the Adience dataset provided from their home website is conducted.

Although the author insisted that the dataset has more than 26000 images, in reality the dataset presented on the website **only has about 18000 images** with sufficient labels and metadata that we can use in training. We shall assume that the dataset used in Levi and Hassner's work only involves these subsets.

For the format of the dataset provided, it consists of two types of data: **1) Text files which contain the labels** for the experiment (gender and age) and other metadata such as name and folder of the corresponding image to the record, and **2) The images, split into folders of human's subjects.**

A quick look at the text files shows that the records have been pre-split into 5 folds and these are exactly the 5 folds that the authors of [2] used in their experiments for cross validation to compare their model's performances with other previous works. This is why **Adience is prematurely a benchmark dataset for in-the-wild gender and age classification as it is pre-split for ease of replication of experiments.** The text files have a small amount of records with NaN and empty values of label that require our attention in our cleaning effort.

On the other hand, one can notice that the images accompanying these text files have unstandardized sizes (not of the same uniform size). This suggests that resizing needs to be done before any training can be conducted.

4.1.2. Data Cleaning and Preprocessing Steps

a. Data Cleaning of Each Fold's Text File

For each of the fold's text file, we import it into our iPython notebook and drop those records with empty gender labels and only select those with not null value for gender label.

Once this is done, the cleaned records are randomly split into 80-20 partitions where the 80% portion of the records is used to form the final train dataset and the 20% portion is used to form the final test set. These 5 80% and 5 20% partitions of the folds are saved as *cv_fold_number_train.csv* and *cv_fold_number_test.csv* accordingly.

b. Generate Final Train and Test Set

We combine the 5 80% train partitions of the 5 folds together to form the final train dataset. Similarly, the final test dataset is done by combining the 5 20% test partitions generated in *step a*. The final train and test set have 14048 and 3515 data records, corresponding to 80% and 20% of the total number of records = 17563 records.

The rationale behind sampling 20% of each fold to form a test set is because besides benchmarking our models, we wish to evaluate the performance of optimal models in real situations too. By leaving out 20% from each fold, the optimal model's training will only be conducted on the remaining 80% which they have seen during cross validation. This is to ensure that the test set and train set (train subset + validation set) are independent from each other and prevent data leaking between them, which may give us a wrong idea about the performance of the optimal model.

c. Generate Train Subset and Validation Set for Cross Validation (CV)

From *step a*, we have 5 80% partitions from each fold and from *step b* we have a final train set by combining these 80% partitions together. To perform 5-fold cross validation, we choose 1 of the 5 80% partitions of the 5 folds as the validation set and the remaining 4 partitions as the train subset. The choice of validation set will rotate around until all 5 80% partitions from each fold has had the opportunity to be the validation set.

We then export these train subsets and validation sets as *cv_fold_number_train_subset.csv* and *cv_fold_number_val.csv* accordingly. In total we should have 5 validation set and 5 train subsets now.

d. Resizing of Images

All images are resized to 256x256 square images as mentioned in the preprocessing steps of [2].

e. Serialize Data for Storage

All the train subset, validation set, final train and test set (csv or txt files) generated so far have attributes such as face_id that link to the particular image associated with their records. In training, one can read the data record in and then based on this ID to load the correct image into memory, conduct resizing and preprocessing before feeding it to the models. This approach can potentially take a long time as the bottleneck can happen at every batch of data due to the mismatch in speed between memory hierarchies of the system (loading image from disk, preprocess in memory then bring to GPUs or CPUs for processing).

In contrast, we attempt to load all images at once into our data preprocessing notebook (and memory), carrying out the resizing of images and then serialize them together with the label (from csv files above) into a dictionary. These serialized objects can be deserialized during runtime once before training and will not require time cost in loading from disk, preprocess at each batch of data. Despite a larger memory is required to fully load the whole serialized dataset, this can, in theory, speed up our training process thanks to the exchange of time complexity with space complexity. Given Colab Pro has a higher RAM quota, this approach is feasible for the project.

4.2. Data Augmentation

The work of *Levi and Hassner* presents two approaches to data augmentation to reduce the overfitting of our model. We shall select one of them as our main data augmentation approach, which includes 2 steps: **1) Randomly centered cropped** the 256x256 images (from data preprocessing) to 227x227 and **2) Applying random horizontal flipping of images**.

Our main goal will be building our CNNs which are trained on the dataset with these augmentations and compare the results to the ones produced with these augmentations in the *Levi and Hassner* paper [2]. We have no intention to conduct more data augmentation techniques for our experiments, as the main interest is on how different architecture of CNNs and combinations of hyperparameters can perform better than their published work and not an emphasis on how data augmentation can further lift the performance.

4.3. Model Architecture

4.3.1. Lean CNN by Levi and Hassner

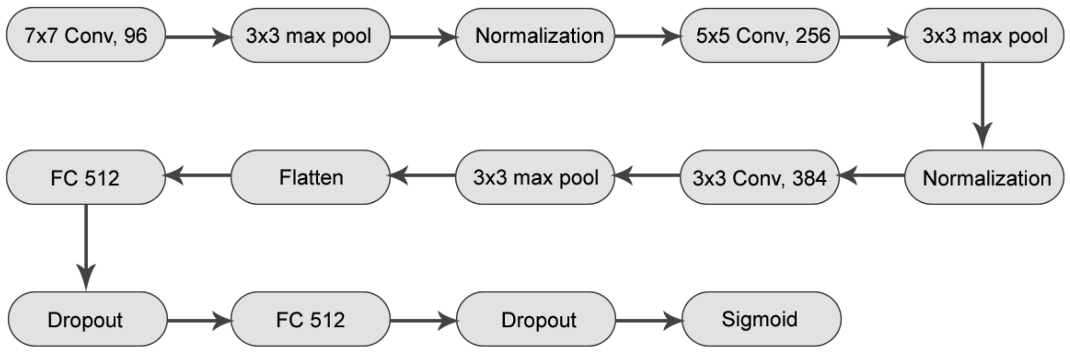
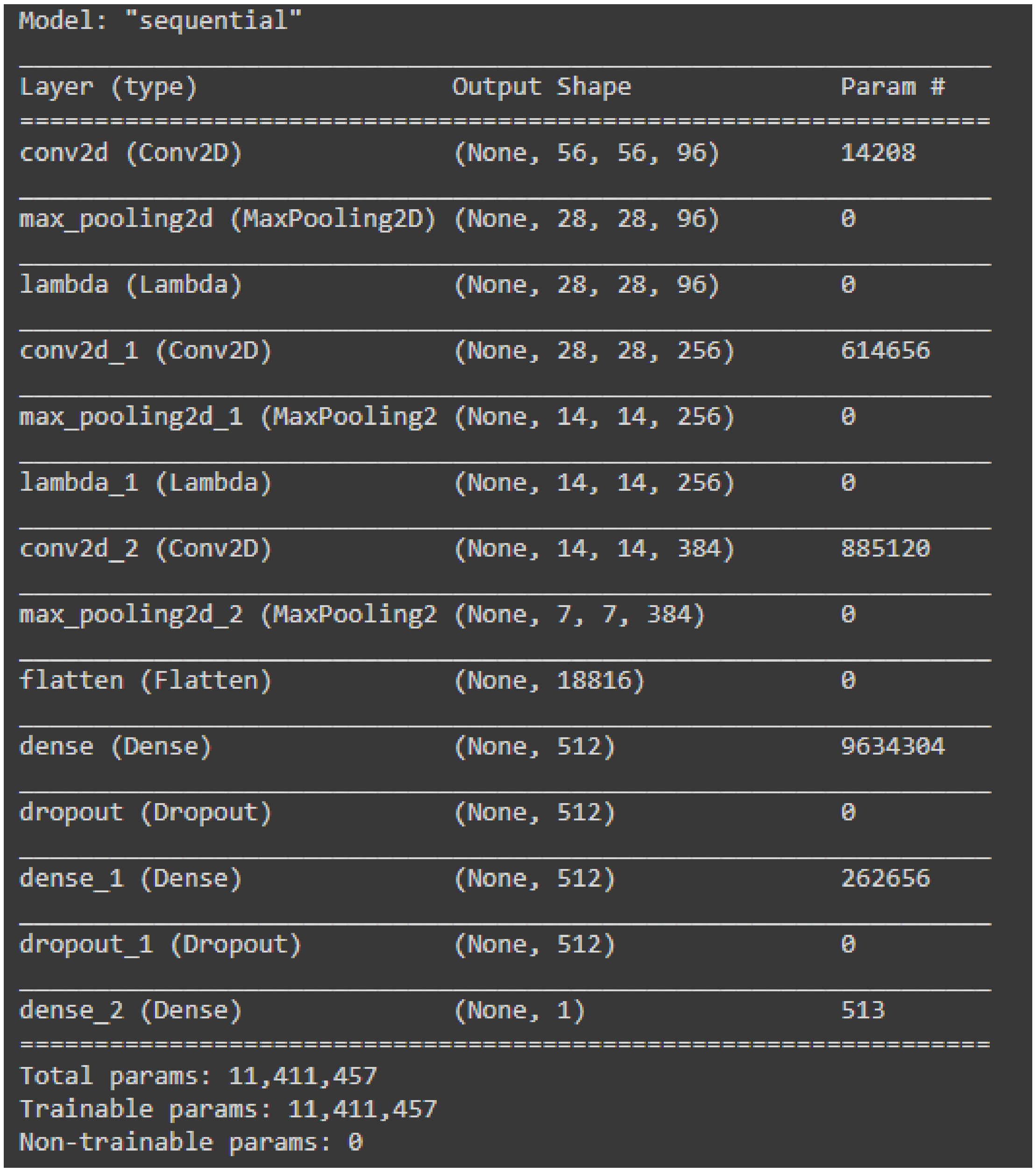


Figure 2. Lean, baseline CNN architecture

Our first approach to building a simple CNN and fine tune the hyperparameters from the ground up is inspired from the published work of Gil Levi and Tal Hassner [2]. The network architecture proposed by them is shown in figure 2. Our summary of implementation of this network architecture is shown in figure 3 below.

Their lean CNN architecture consists of 3 convolution layers, followed by 3 max pooling layers each, and normalization layers when necessary. The output of these layers is then flattened and fed to the 3 fully connected layers (with dropout in between) to produce a sigmoidal output for binary classification.

The first convolution layer has 96 filters of size 7x7x3 pixels, with ReLu activation, valid padding and are applied to the 227x227 input. The resulting 56x56x96 feature maps are passed through a max pooling layer with a 3x3 window and stride of 2. As the window size is odd and stride is even, same padding is applied to produce the expected 28x28x96 blobs. This is then passed into the *Lambda* function implementing local response normalization.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 56, 56, 96)        14208
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 28, 28, 96)        0
_________________________________________________________________
lambda (Lambda)              (None, 28, 28, 96)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 256)       614656
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 256)       0
_________________________________________________________________
lambda_1 (Lambda)            (None, 14, 14, 256)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 384)       885120
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 384)         0
_________________________________________________________________
flatten (Flatten)            (None, 18816)             0
_________________________________________________________________
dense (Dense)                (None, 512)               9634304
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=================================================================
Total params: 11,411,457
Trainable params: 11,411,457
Non-trainable params: 0
```

Figure 3. Model implementation summary for Lean CNN proposed by [2]

The 28x28x96 blob is then passed through the second convolution layer with 256 filters of size 5x5x96. This layer has ReLU activation and same padding. The 28x28x256 output is passed through a max pool layer with parameters same as above and another similar local response normalization. The output of these 3 layers is a 14x14x256 blob.

This blob is then passed through the final convolution layer with 384 filters of size 3x3x256. It has same padding and ReLU activation, which produced an output of shape 14x14x384. This is then passed through a max pool layer with similar parameters like the other two, producing a 7x7x384 feature map. This is then flattened before feeding to the fully connected layers.

The first and second fully connected layers are of 512 neurons, ReLU activated while the last one only has 1 neuron, sigmoid activated (since it is a binary classification problem). In between the first and second fully connected layer and between the second and last fully connected layer, a dropout of 50% is applied.

4.3.2. Transfer learning with Google's EfficientNet

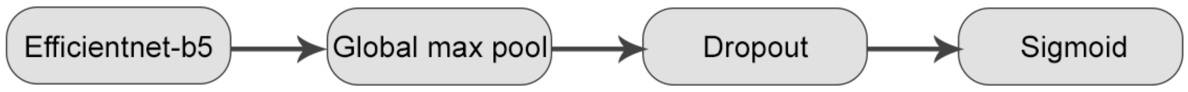
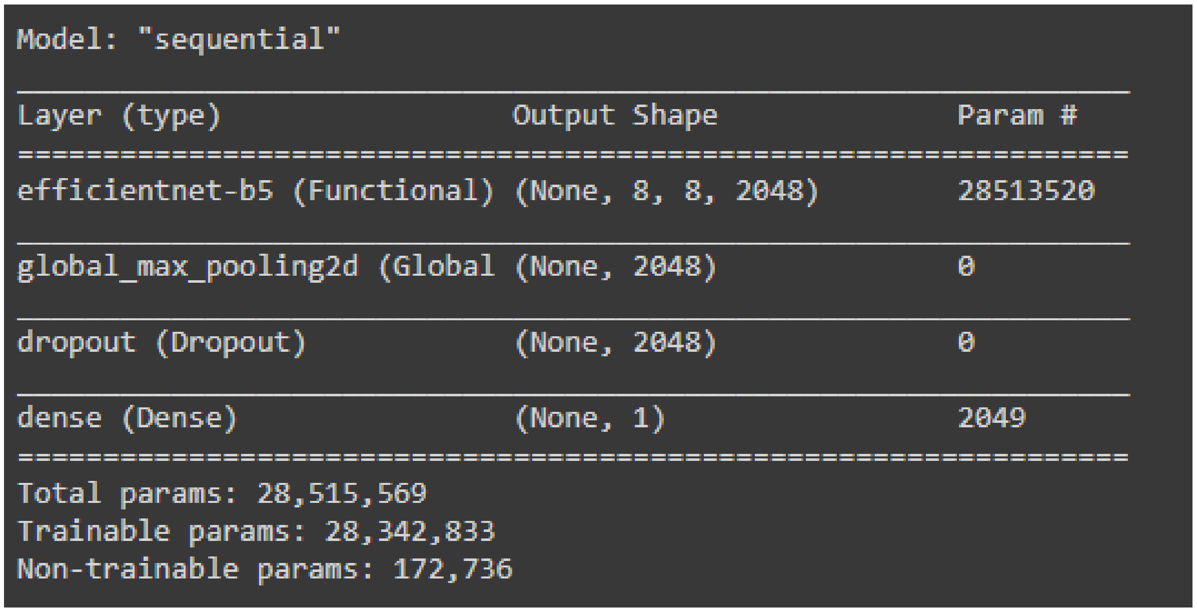


Figure 4. EffjcientNet-based transfer learning CNN architecture

On to transfer learning, we implemented our model by utilizing EfficientNet-B5 as the backbone and added an additional pooling layer and a dropout layer. The output is fed to a sigmoidal layer that performs binary classification onto the results as shown in figure 4.

The EfficientNet base is instantiated in Keras by importing the *efficientnet.keras* module and calling the *EfficientNetB5* function. The weights of the base model are set to the ImageNet pre-trained weights such that we wish to observe the vanilla performance of the model.

The implementation summary of this transfer learning model and the number of parameters that needs to be trained (for the unfrozen part) is shown in figure 5.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
efficientnet-b5 (Functional) (None, 8, 8, 2048)        28513520
_________________________________________________________________
global_max_pooling2d (Global (None, 2048)              0
_________________________________________________________________
dropout (Dropout)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 1)                 2049
=================================================================
Total params: 28,515,569
Trainable params: 28,342,833
Non-trainable params: 172,736
```

Figure 5. Model implementation Summary for EffecientNet transfer learning

4.4. Model Selection

4.4.1. Cross Validation

As the benchmark work of *Levi and Hassner* [2] was conducted with 5-fold cross validation, it only makes sense if we apply the same 5-fold cross validation approach to evaluate the effect of various modifications on our model performances. 5-Fold cross validation is done for both approaches of building and tuning our own CNNs based on *Levi and Hassner's* published architecture [2] as well as the transfer learning CNN based on Google's EfficientNet [3].

The 5 folds used for training and validate testing are generated in the data preprocessing steps above (section 4.1.2). In 5-fold cross validation we will be dividing the dataset into 5 portions and select 1 of them as validation set, while using the other 4 as train set. This is repeated until all 5 portions have had the opportunity to be the validation set. Cross validation allows us to make use of all the data available in the dataset and gives us an idea of how well the models generated by our experiments can perform with unseen data.

4.4.2. Performance Metrics and Criteria for Selection

The metric used to compare performances between models is the test (or validation) accuracy (depending if we are comparing optimal models or cross-validated models correspondingly). This will give us an idea of how well our model can perform based on unseen data.

4.5. Experiments and Results Collection Methodology

4.5.1. Determining the Optimal Model

The optimal model with optimal hyperparameters in each of our experiments are determined from cross validation results. We will be carrying out model training on the train subsets and then obtain the best validation accuracy (max) results for each fold subsets. The average of the 5 folds' best validation accuracy is calculated and used as the score in cross-validation and selection of the best hyperparameters.

The optimal model with optimal hyperparameters in each experiment is then trained on the final train set and test on the test set.

4.5.2. Considerations on GPU training and Bagging of Folds

Despite setting random seed values in our experiments, it is unavoidable that the results from our training models (cross validation or even optimal model training) vary from iterations to iterations. This may be because the GPU libraries such as CUDA used in training have random seed implementations that is out of our control.

To minimize the randomness that can affect the cross-validation results when running the cross-validation process at different times, we resolved to bagging of folds. This means that we repeat the training of the model being cross validated for 3 times and select the best validation accuracy in each of the iterations. We then average these best validation accuracies and use this as the best validation accuracy for the particular fold. Then, this value is used to compute the average validation accuracy of the whole model like it was mentioned in 4.4.1 and used in comparing performances between different hyperparameters of the model.

This is **an average of average approach** which can result in a closer true average performance as the randomness of GPU can be accounted with the averaging of 3 bags and the averaging of 5 folds.

4.5.3. Early Stopping and Definition of best validation accuracy in training cross validation model.

We shall stop training when the validation loss of the model fully converges and reach its minimum value**. The validation loss will be diverging afterwards so the search space for best validation accuracy of each fold should be from the epoch that validation loss is minimal all the way back to epoch 1**. This is to ensure that the higher validation accuracy we obtained is not due to overfitting of the model**. More explanation on this is provided in part b of section 6.1.**

The best validation accuracy for the model instance trained on the train subset of this fold is the highest validation accuracy in the search space between epoch 1 to the epoch when validation loss is minimal.

To ensure the true validation loss has been achieved, we used a stopping tolerance of 30 epochs. Meaning if model is stopped at 100 epochs, the search space for best validation accuracy shall be from epoch 1 to epoch 70.

4.5.4. Training of Optimal Models

Once the cross-validation results are evaluated and models with best performance from optimal hyperparameters combination are determined, these models are trained on the full train set and evaluate on the full test set (80-20% of the whole 18000 images of Adience respectively). This suggests how our optimal model can perform in reality.

For optimal models that are built on top of the work of Levi and Hassner, 2015 [2] using hyperparameters tuning, we will be training to 200 epochs as the computing resources allow for it. On the other hand, we will only be training the transfer learning model based on EffecientNet for 10 epochs only. This is for interest of time.

4.6. Hyperparameters Tuning (for Lean CNN)

4.6.1. Experiments with Depth

Levi and Hassner [2] published work claim that the optimal depth of the model consists of 3 convolution – max pool layer set and 3 fully connected layers to produce the probability of gender at output. We shall evaluate their claim on the optimal depth of this model.

Previous work of [4] have shown that large amount of dense, fully connected layers at the end of CNNs may not contribute meaningfully to the performance of the model. Instead, they suggest that adding more convolution and max pooling layers may allow for more efficient features to be extracted from images and give rise to better performance. This also inspire many modern, generic deep neural network, one of which is Google's EffecientNet used in our transfer learning approach.

Based on these findings, we attempt a few modifications to the lean network produced by [2]:

- Remove 1, 2 fully connected layers from the lean CNN architecture.
- Add in 1 extra convolution layer and 1 max pool layer at the end of the third convolution – max pool layer set. Specification of conv layer is 512 filters, 3x3 kernel size, padding is valid, ReLU activation and weight initialized like the other conv layers. The max pool layer is 2x2 windows of stride 1 (to extract the strongest features rather than down-sampling).
- A combination of both changes: 4 conv layers, 1 fully connected; 4 conv layers, 2 fully connected.

The optimal depth from this experiment shall be used in the next experiments. Our search space for this optimal depth is 2fc, 1fc, 4conv, 2fc4conv, 1fc4conv and the baseline architecture. Detail for each codeword are below:

- 2fc: Remove 1 fully connected layer (hidden layer NOT output layer) and its following dropout layer.
- 1fc: Remove 2 fully connected layers (1st and 2nd after flattening) and their following dropout layers.

- 4conv: Add in 1 convolution layer before flattening and feed to fully connected layers.
- 2fc4conv: Add in 1 convolution layer like 4conv but remove 1 fully connected layer like 2fc.
- 1fc4conv: Add in 1 convolution layer like 4conv but remove 2 fully connected layers like 1fc.

4.6.2. Experiments with Width

Another dimension that we can modify for our lean CNN is the width (number of neurons or filters in each layer. For conv layers, the number of filters decide the number of features the model learn from the images and for dense fully connected, the number of neurons can affect the capacity of the prediction (space of representable outputs enlarged).
We modify the width of the following layers with a limited search space shown below:

- Conv1: 48, 96 or 192 filters
- Conv2: 128, 256 or 512 filters
- Conv3: 192, 384 or 768 filters
- Fully Connected layers: 256, 512 or 1024 neurons.

Our grid search approach is to exhaust the all the possible combination with one width parameter different from the baseline model (meaning only one difference to baseline model at a time in term of number of filters or number of neurons). Then we apply a heuristic randomized search for 2+ differences as if we need to exhaust everything, that will need 3x3x3x3x(3x5) = 1215 models to be trained (1 model for 1 fold and 1 bag).
The model with optimal hyperparameter in this experiment will be used in the next experiment.

4.6.3. Experiments with Dropouts

Through our coursework, we learnt that dropout at fully connected layers can greatly lift the performance of the model in classification problem. Levi and Hassner has applied 2 dropout layers (1 after the first fully connected layer and 1 after the second one) with 50% rate each. In this experiment, we attempt to randomized grid search amongst [0, 20%, 50% and 80%] dropout rate and add on 1 additional dropout layer between the flatten of last max pooling output and the input of the first fully connected layer. Total combinations to exhaust 3 dropouts out layer search with the above search space is 3x4x(3x5) = 180 models (1 per 1-fold and 1-bag).

The model with optimal dropout configuration will be use for the next round of experiment.

4.6.4. Experiments with Optimizers

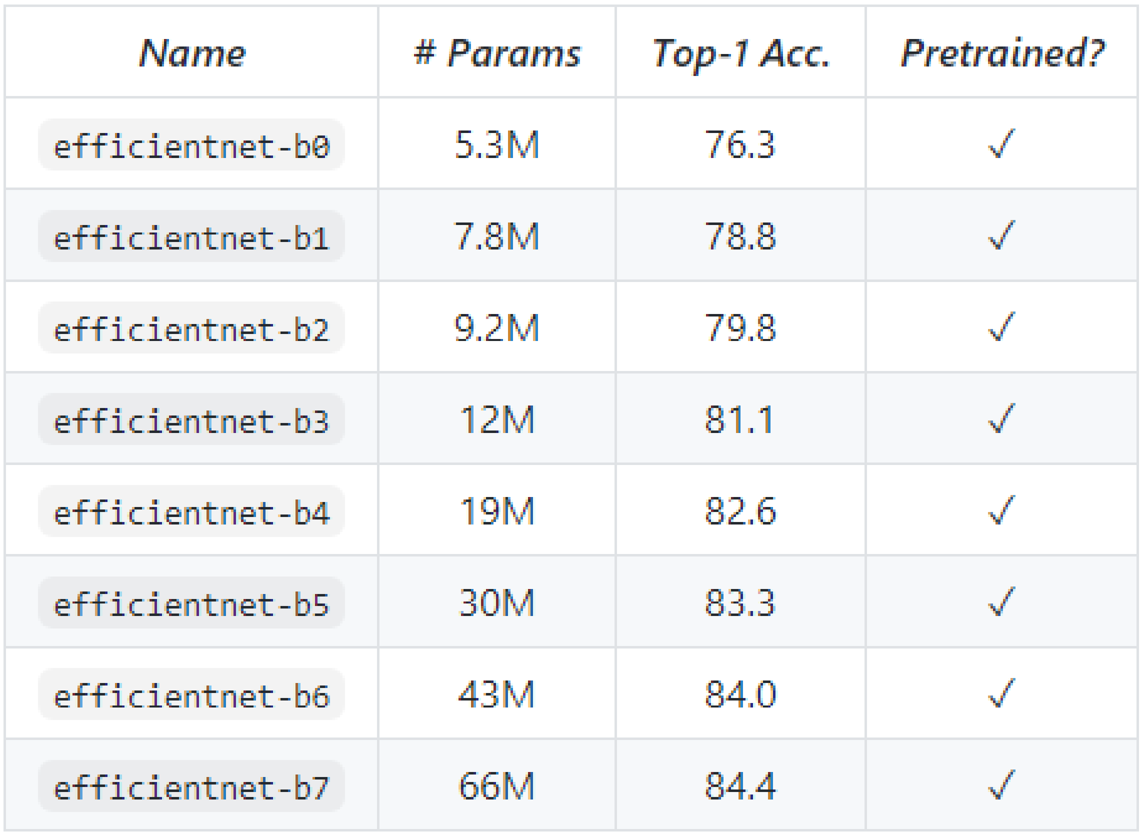
Mini-batch stochastic gradient descent learning yields the best result for training neural network in most of the cases but might require very long time for the model to converge. In these experiments, we explore a few variations on the mini-batch SGD we have been using so far as well as evaluating other optimizers.

Firstly, we will be exploring the difference between SGD, SGD with momentum (0.3 and 0.8), RMSProp and Adam optimizer (all at learning rate 1e-3). Then we will be trying out RMSProp and Adam at lower learning rate of 1e-5. These are mainly to see if there are any other optimizer that can achieve a better result that SGD (since computing resources is not our concern).

4.7. Transfer Learning

Apart from building a CNN model from the ground up, we also approached the gender classification problem from another aspect: transfer learning. With transfer learning, we are able to leverage the knowledge that off-the-shelf pre-trained models specially designed for computer vision have acquired for one task to solve other related ones. As mentioned in 3.2, we chose Google's EfficientNet [3] as our model backbone for its state-of-the-art performance and as a benchmark for the Levi and Hassner's approach [2] to compare to.

Among all the implementations of EfficientNets, it would be easy and straightforward to choose EfficientNet-B7 for optimal performance output. However, this is not the case due to hardware limitations and time concerns. Below in figure 6 is a comparison table detailing the performance and discrepancies of the 8 available implementations:



Name	*# Params*	*Top-1 Acc.*	*Pretrained?*
`efficientnet-b0`	5.3M	76.3	✓
`efficientnet-b1`	7.8M	78.8	✓
`efficientnet-b2`	9.2M	79.8	✓
`efficientnet-b3`	12M	81.1	✓
`efficientnet-b4`	19M	82.6	✓
`efficientnet-b5`	30M	83.3	✓
`efficientnet-b6`	43M	84.0	✓
`efficientnet-b7`	66M	84.4	✓

Figure 6. EfficientNet models and their characteristics

Considering the model size (together with RAM and VRAM considerations) and their corresponding performance, we chose EfficientNet-B5 as the backbone to our transfer learning model.

To create our own classification layer stack on top of the EfficientNet convolutional base mode, we adapted *GlobalMaxPooling2D* to convert 4D input tensors of shape (batch_size, row, cols, channels) into 2D tensors with shape (batch_size, channels) to reduce the number of parameters to train compared to simply utilizing the *Flatten* layer. In addition, we added a *Dropout* layer with a dropout ratio of 0.2 just before the output layer as a regularization method to reduce overfitting during model training.

The training process is split into two parts: the frozen part and the unfrozen part. For the frozen part, the convolutional base's weights are frozen and left untouched during the model training process. This is to prevent representations previously learned by the model from the ImageNet dataset from being destroyed and generalized to our own dataset. Thus, only the final few layers that we have added to the base model is being trained in this part. For the 2nd part, the base weights are unfrozen, and the performance of our transfer learning model can be observed from this process.

4.8. Implementation of models and Resources

Models are coded in Python and run as Jupyter Notebook. Our framework for implementation of model is Tensorflow, specifically the in-built Keras API which aid in the rapid prototyping of our model.

For running of notebook, the preprocessing and generation of serialized image was done on local machine since the resource of a typical computer is sufficient to do so. For training of model, we resolve to Google Colab Pro ($15 SGD per month, which gives us access to 25GB of RAM and powerful GPUs such as Nvidia Tesla V100 and Tesla P100. We can run 4 instances of notebook at the same time, so at any point in time, 4 instances of model training can be conducted. Because of this reason, the training of model on each fold is split to a notebook on its own to capitalize this parallelism.

As for training time of the model, this varies depending on if the powerful Tesla V100 is assigned to us or the slower GPUs. Results are stored in Google Sheet to facilitate collaboration between teammates.

5. Experiments and Results

5.1. Building and Tuning CNN from the ground up

5.1.1. Baseline model

a. Benchmark Results

Firstly, we replicate the architecture and experimental approach proposed by Levi and Hassner in [2] as closest as possible to produce our own baseline model for this approach of tuning the lean CNN. It is important to reiterate that we are following their first approach in data augmentation which is random horizontal flipping of images and centered cropping of 256x256 images to 227x227 images before feeding into our network.

The results for validation accuracy of the individual bags/iterations for each of the 5 folds are shown below in table 1.

Table 1. Validation Accuracies per bag, per fold and overall, of baseline model

Fold	**Bags**			**Average (fold) Accuracy**
	0	1	2	
0	0.87207	0.86927	0.86833	0.86989
1	0.85185	0.86016	0.85324	0.85508
2	0.85720	0.86861	0.86428	0.86336
3	0.88721	0.87926	0.88569	0.88405
4	0.83755	0.83466	0.83682	0.83634
			Overall Accuracy	86.17±1.77%

From table 1, it can be seen that our baseline model has an **overall average validation accuracy of 86.17% and standard deviation of 1.77%**. This is obtained by firstly repeating each fold's model training for 3 times, taking their best validation accuracy in a manner described in 4.5 and then compute the average best validation accuracy of each fold (the right-most column). Then, the overall validation accuracy of the baseline model is obtained by averaging the average best validation accuracy of each folds. The standard deviation of these 5 folds reading is also obtained.

Our baseline model **has very close performance to that reported in the Levi and Hassner paper**. They reported a validation accuracy of **85.9%±1.4%,** while our result is **86.17±1.77%.** The slight difference of about 0.1% in mean validation accuracy and 0.3% in standard deviation of the 5 folds could be explained to our resampling of given 5 fold dataset to form the final train and test set (meaning we are only training and validating on 80% of what they did). This may slightly decrease the performance due to less training images for the model to learn on (in each fold). The total trainable parameters for this baseline model are 11,411,457.

(Code for above part is in from_scratch/baseline_gender_fold0.ipynb)

b. Optimal Model Training

We also train the baseline model on the final train set and test it on the test set (split in 80%-20% ratio as mentioned in part b of 4.1.2. The train, test accuracies and losses plot versus epochs of training are shown below in figure 7 and 8 below.

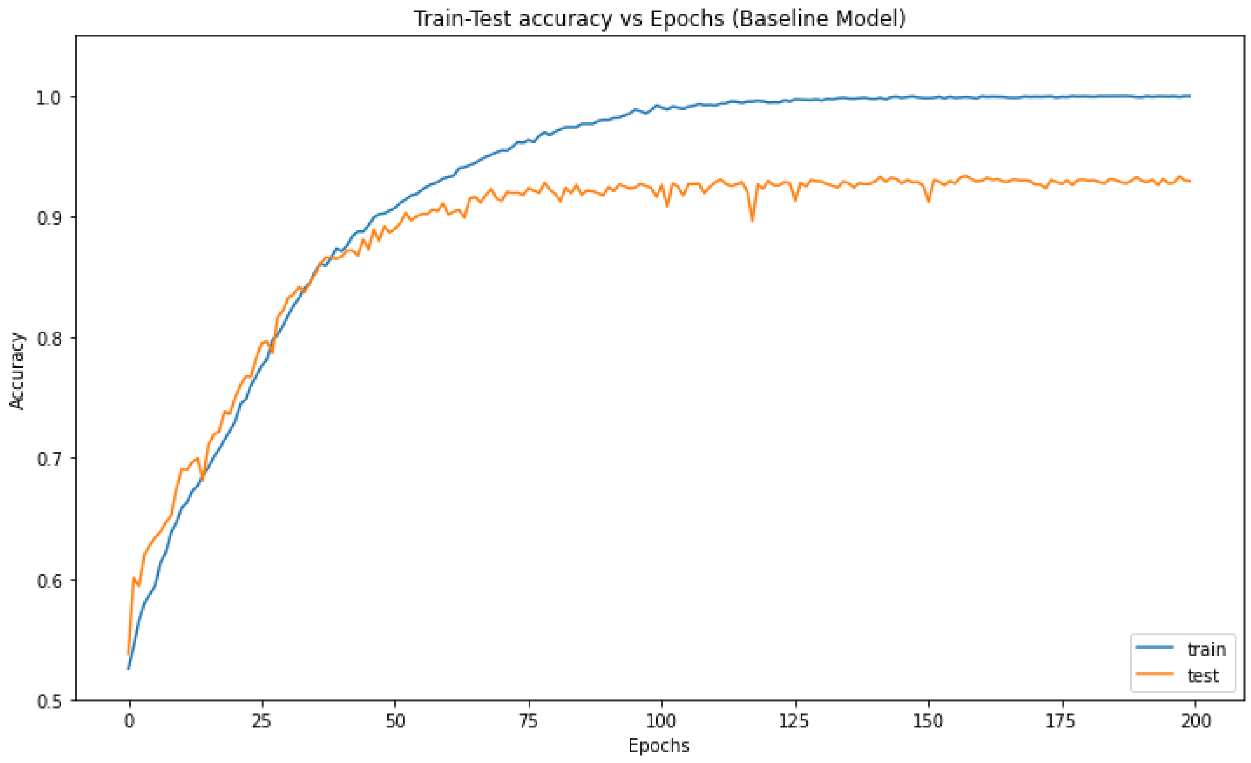


Figure 7. Train-Test accuracies vs epochs of baseline model

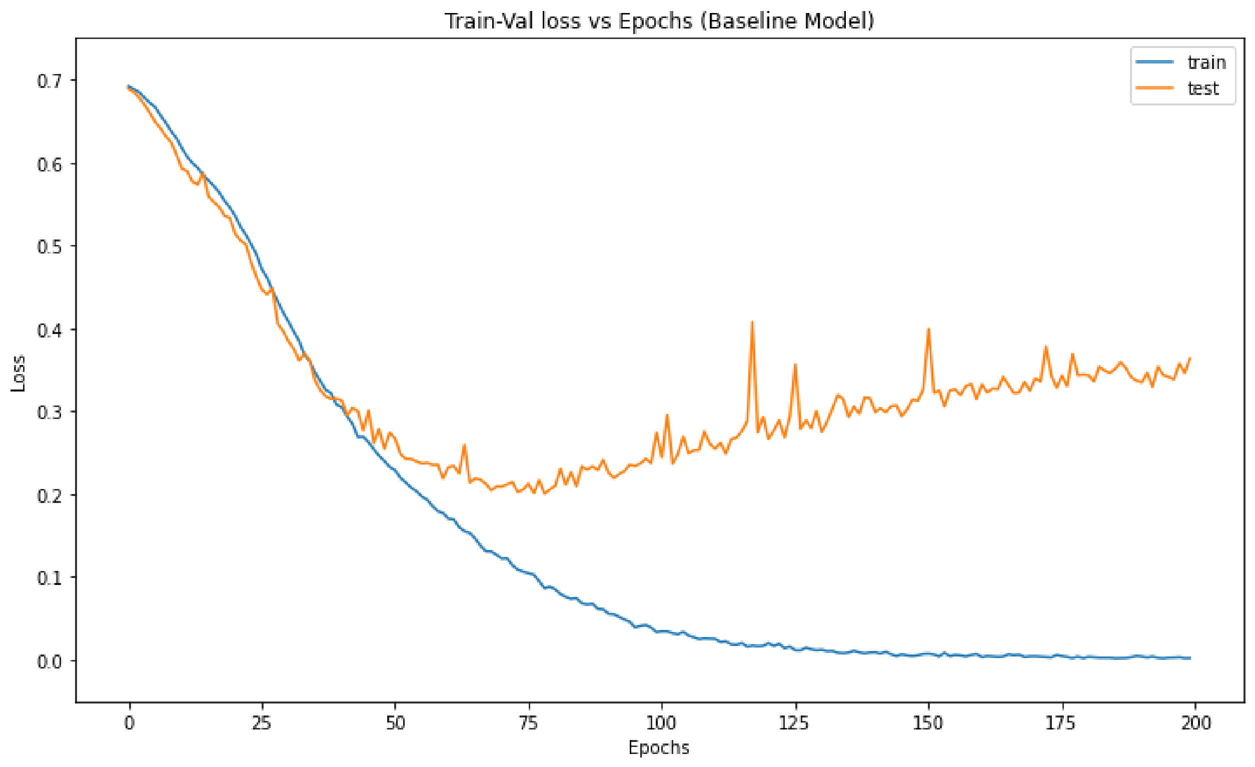


Figure 8. Train-test losses vs epochs for baseline model

From figure 7 and 8, we can observe that overfitting of this model starts to happen at about 35 epochs into training (for both accuracy and loss). The test accuracy reaches convergence at about 80 epochs while test loss reaches its minimum point at about the same epoch, but continuously increasing afterwards. **The max value for test accuracy and min value for test loss amongst 200 epochs is 0.93371 and 0.2003, respectively**.

(Code for above part is in from_scratch/optimal_baseline_gender.ipynb)

5.1.2. Experiments on Depth

a. Benchmark Results

Next, we try to remove fully connected layers and add in more convolution and max pool layers as discussed in 4.6.1. For presentation purpose, only the mean accuracy and standard deviation across 5 folds are shown in table 2 below for each of the mode. The details result per bag and per fold of each model (similar to what is presented in table 1) is included in annex A, part a.

Table 2. Overall Test Accuracy across 5 folds of different depth architecture CNNs

Codeword	Model	Overall Accuracy	Number of Parameters
	Baseline (3 convolutions + 3 fully connected layers)	86.17±1.77%	11,411,457
4conv	4 convolutions + 3 fully connected layers	85.77±1.60%	7,741,953
2fc	3 convolutions + 2 fully connected layers	86.11±1.78%	11,148,801
1fc	3 convolutions + 1 fully connected layer	85.40±1.70%	1,532,801
2fc4conv	4 convolutions + 2 fully connected layers	85.51±2.08%	7,479,297
1fc4conv	4 convolutions + 1 fully connected layer	85.90±1.61%	3,292,161

From table 2, we can observe that the number of parameters in baseline model decreases significantly from the addition of the extra convolution and max pool layer (from 11,411,457 in baseline to 7,741,953 in 4conv). The reduction is logical as the newly added convolution and max pooling layers can further reduce the size of flattened output fed into the fully connected layers (from 18816 to 8192), which results in a drop of parameters in the first fully connected layers by half, in exchange at only 1,700,000 parameters increased in the last convolution and max pooling layer. This is an evidence that convolution layer with its parameter sharing capabilities, can constrain its neurons to use the same set of weight and bias, therefore give rise to smaller increase in number of trainable parameters comparing to that of fully connected layers.

Similarly, the number of parameters of the model also decreases with more fully connected layers removed (from 11,411,457 to 11,148,801 in 2fc and 1,532,801 in 1fc).
In addition, from table 2, our baseline model still has the highest overall average accuracy. In the best case of this experiment, the removal of 1 fully connected layer in model 2fc yields overall validation accuracy of 86.11±1.78%, which is the closest to our baseline model in terms of average validation accuracy and standard deviation between the 5 folds validation accuracy.

This suggests that adding more convolution layer does not enhance the performance of the baseline model. The similar trend can be observed in removing the fully connected layers from the model architecture. A combination of addition of conv layers and removing FC layers do not improve the performance either. Such observation confirms the claim of Levi and Hassner that their published architecture in [2] has the best depth for learning gender classification.

(Code for above part is in from_scratch/depth_experiments_fold0.ipynb)

b. Optimal Model

The optimal model in this experiment is the baseline model which is the same as that in 5.1.1.b. Please refer to 5.1.1.b. for the corresponding train-test graphs and comment.

5.1.3. Experiments on Width

a. Benchmark Results

In these experiments, we vary the width (number of filters or number of neurons) of the layers in search for a better performance of our baseline model and optimistically a lower number of parameters needed. This setup is described in 4.6.2. The overall average validation results (across 5 folds) for these modified architectures are shown in the table 3 below. For details on the individuals bag validation accuracies and fold average accuracies, refer to annex A, part b.

Table 3. Overall Test Accuracy across 5 folds of different width architecture CNNs

No.	Model (conv1, conv2, conv3, fc)	Overall Accuracy	Number of Parameters	Note
1	(96,256,384,512)	86.17±1.77%	11,411,457	Baseline
2	(48,256,384,512)	86.45±1.51%	11,097,153	Single width changes
3	(192,256,384,512)	85.71±1.93%	12,040,065	
4	(96,128,384,512)	86.18±1.59%	10,661,791	
5	(96,512,384,512)	85.70±1.64%	12,910,849	
6	(96,256,192,512)	86.10±1.78%	6,152,001	
7	(96,256,768,512)	86.30±1.66%	21,930,369	
8	(96,256,384,256)	85.94±1.52%	6,397,185	
9	(96,256,384,1024)	86.02±1.72%	21,833,217	
10	(48,128,768,512)	86.37±1.75%	20,577,601	More than 1 width changes
11	(96,128,768,512)	86.37±1.64%	20,738,305	
12	(48,256,768,512)	86.31±1.47%	21,616,065	
13	(48,128,384,512)	86.35±1.75%	10,501,057	

From table 3, we can observe that changes to the width of the lower convolution layers makes very little change in the number of trainable parameters of the model. For example, reducing the number of filters by half in the first convolution layer (model 2) only decreases the number of parameters by about 300,000 parameters or increasing it by twice the number of filters (model 3) only increase the number of parameters by about 600,000 parameters.

On the other hand, changes make in the number of filters for higher convolution layers and the number of neurons in the fully connected layers give rise to bigger changes in number of trainable parameters. For instances, reducing the number of filters for the third conv layer (model 6) or the fully connected layers by half (model 8) can reduce the number of trainable parameters by half of that of the baseline. Similarly doubling them (model 7 and model 9) can leads to about double the number of parameters (from the baseline model).

For single change in width combination (model 2 to 9, with respect to the width combination of baseline model 1), we notice that reducing the 1st conv filters, 2nd conv filters by half and increasing the 3rd conv filters by 2 times lead to an increment in validation accuracy. We use this and carry out a randomized grid search for more than 1 width changes as shown in the models 10 - 13 in table 3. It is clear that with a combination of the above 3 changes, the model performances will be lifted. However, these improvements are not as big as the improvement from simply reducing the number of filters in the 1st conv layer by half**. Therefore, the model 2 with width specifications (48,256,384,512) has the highest accuracy. This is the optimal model we want to focus on (highlighted in green).**

(Code for above part is in from_scratch/width_experiment_gender_fold0.ipynb)

b. Optimal Model

We train our optimal model in this experiment, which is the model with width of conv1, conv2, conv3 and fcs to be (48,256,384,512) on the final train set and test it on the test set. The train-test accuracies and losses vs training epochs are shown below in figure 9 and figure 10.

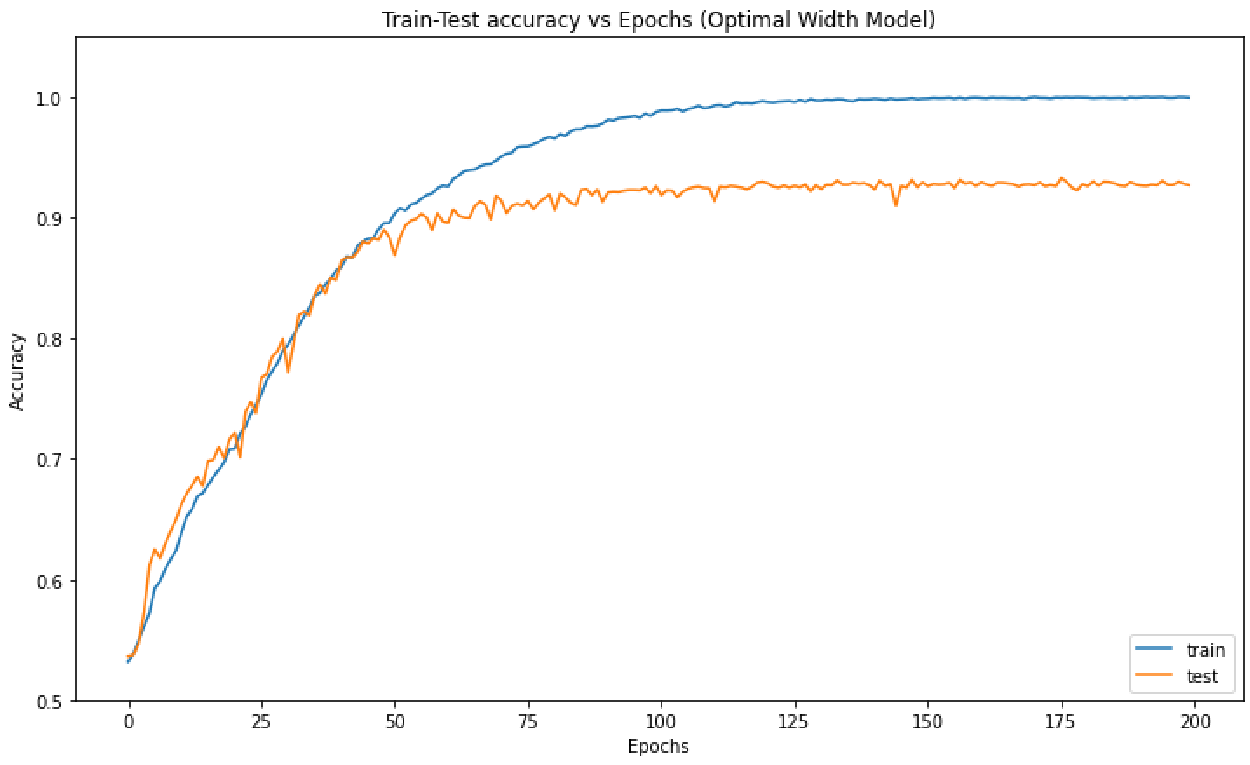


Figure 9. Train-Test accuracies vs epochs of optimal width model

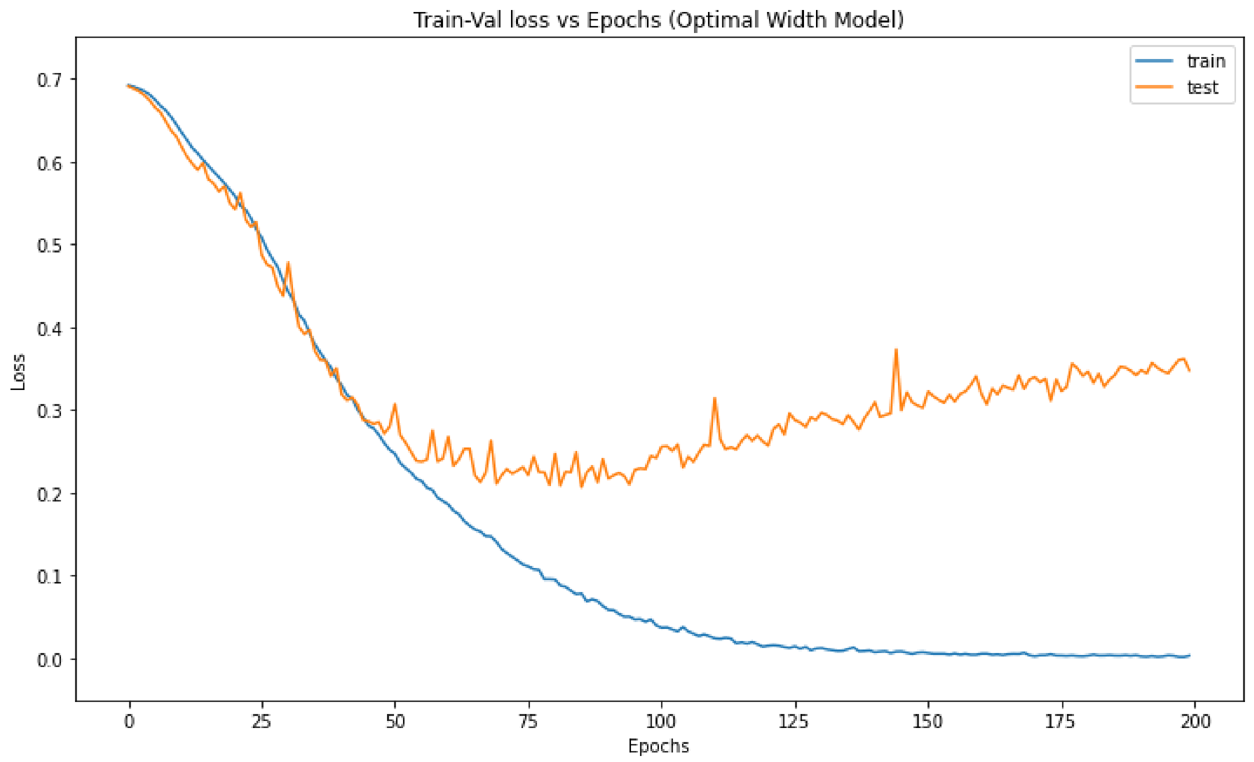


Figure 10. Train-test losses vs epochs for optimal width model

From figure 9 and 10, we can observe that overfitting of this model starts to happen at about 40 epochs into training (for both accuracy and loss). The test accuracy reaches convergence at about 85 epochs while test loss reaches its minimum point at about the same epoch, but continuously increasing afterwards. **The max value for test accuracy and min value for test loss amongst 200 epochs is 0.93286 and 0.2068, respectively**.

(Code for this part is found in from_scratch/optimal_width_experiment_gender.ipynb)

5.1.4. Experiments on Dropouts

a. Benchmark Results

Using the optimal model from the previous experiment (width) in 5.1.3, we experiment with different combinations of dropout values and also adding in more dropout layers.
The results for these experiments are shown in the table 4 below.

Table 4. Overall Test Accuracy across 5 folds of different dropout architecture CNNs

No.	Model (Dropout1, Dropout2, Dropout3)	Accuracy	Note
1	(0, 50, 50)	86.44±1.51%	Optimal model from 5.1.3
2	(0, 0, 0)	85.83±1.57%	2 Dropout Layers (similar to Levi and Hassner)
3	(0, 20, 20)	86.04±1.52%	
4	(0, 80, 80)	86.35±1.80%	
5	(0, 50, 80)	86.52±1.35%	
6	(0, 50, 20)	86.31±1.50%	
7	(0, 20, 50)	85.90±1.88%	
8	(0, 20, 80)	86.04±1.66%	
9	(0, 80, 20)	86.57±1.57%	
10	(0, 80, 50)	86.66±1.33%	
11	(20, 20, 20)	85.98±1.70%	1 Additional Dropout before first fully connected layer
12	(50, 50, 50)	86.65±1.98%	
13	(20, 50, 80)	86.77±1.86%	
14	(50, 20, 80)	86.69±1.91%	
15	(20, 80, 80)	86.66±1.76%	
16	(80, 50, 20)	87.24±2.21%	

From the table, it can be seen that the model with worse overall cross-validation accuracy model 2 with no dropout (0,0,0) before any fully connected layers (85.83±1.57%). **The best performing model with 2 dropout layers is model 10 (0,80,50) with cross-validation accuracy of (86.66±1.33%).** When adding in 1 additional dropout layer (at input of the first fully connected layer), the optimal model 16 (80,50,20) is found, with cross-validation accuracy of 87.24±2.21%.

A general trend we can observe that adding more dropout layers into our model architecture can lift up the performance of our cross-validation model as seen from the improvement in performance of 3 layers dropout (models 11 – 16) to 2 layers dropout (models 1, 3-10) and to no dropouts at all (model 2). On the other hand, by applying different dropout rate at different fully connected layers input, the performance of the model can also be lifted.

Given our main criteria for selecting model from cross-validation results to be the best average validation accuracy amongst 5 folds, the above optimal model choice is valid since model 16 has the max average validation accuracy comparing to the rest. However, it is worth noting that it also has the highest standard deviation (calculated from the 5 folds' validation accuracy). We also notice a trend that more layers of dropout will lead to higher standard deviation between the 5-fold validation accuracies of that model (models 11-16). Again, for readability purpose, the detailed results for each bag and each fold validation accuracies are shown in annex A, part c instead of our discussion here.

The optimal model with dropout settings to be (0.80,0.50,0.20) will be trained with the final train and test on test set as in the next part.

(Code for this part is found in from_scratch/dropout_gender_fold0.ipynb)

b. Optimal Model

The optimal model for this experiment is the one with added dropout layer before the first fully connected and having the following dropout rate combination (80,50,20) for the three dropout layers. We will be training this optimal model with the final train set and test on the final test set. The train-test accuracies and losses of this optimal model vs epochs are plotted during training and shown in the figures 11 and 12 below.

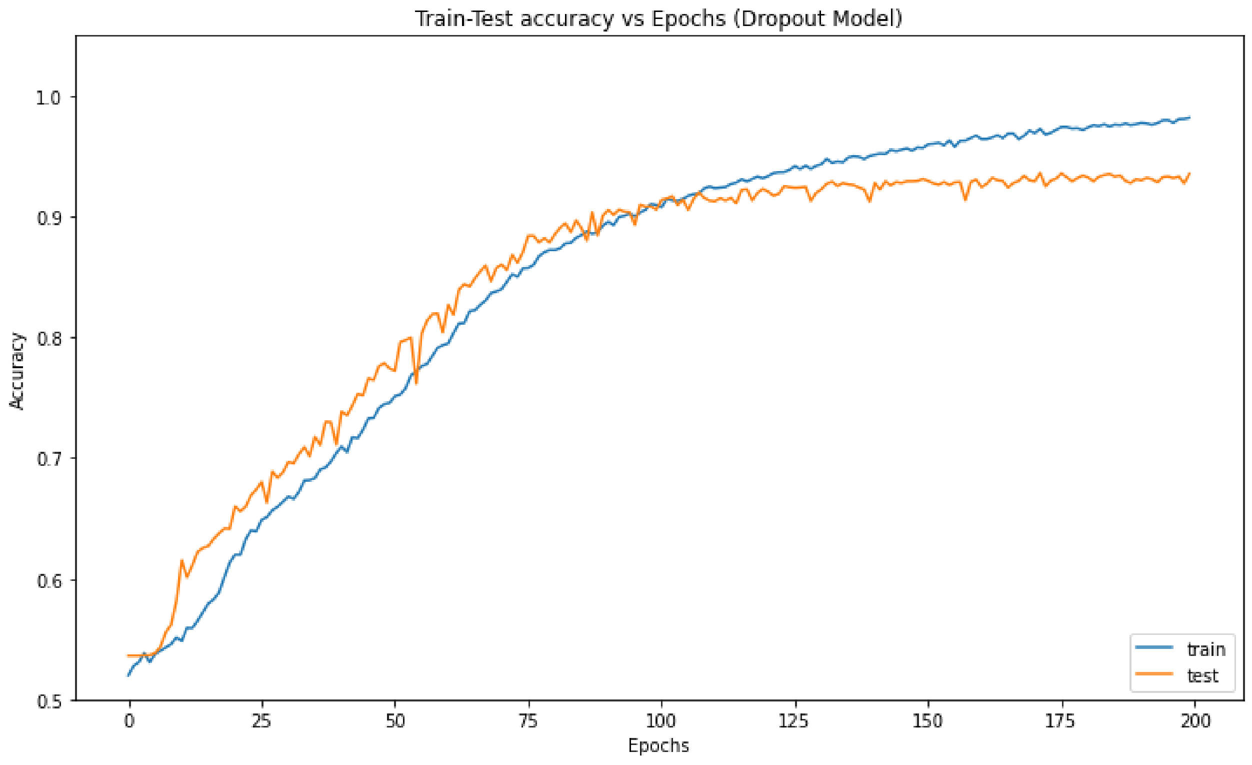


Figure 11. Train-Test accuracies vs epochs of optimal dropout model

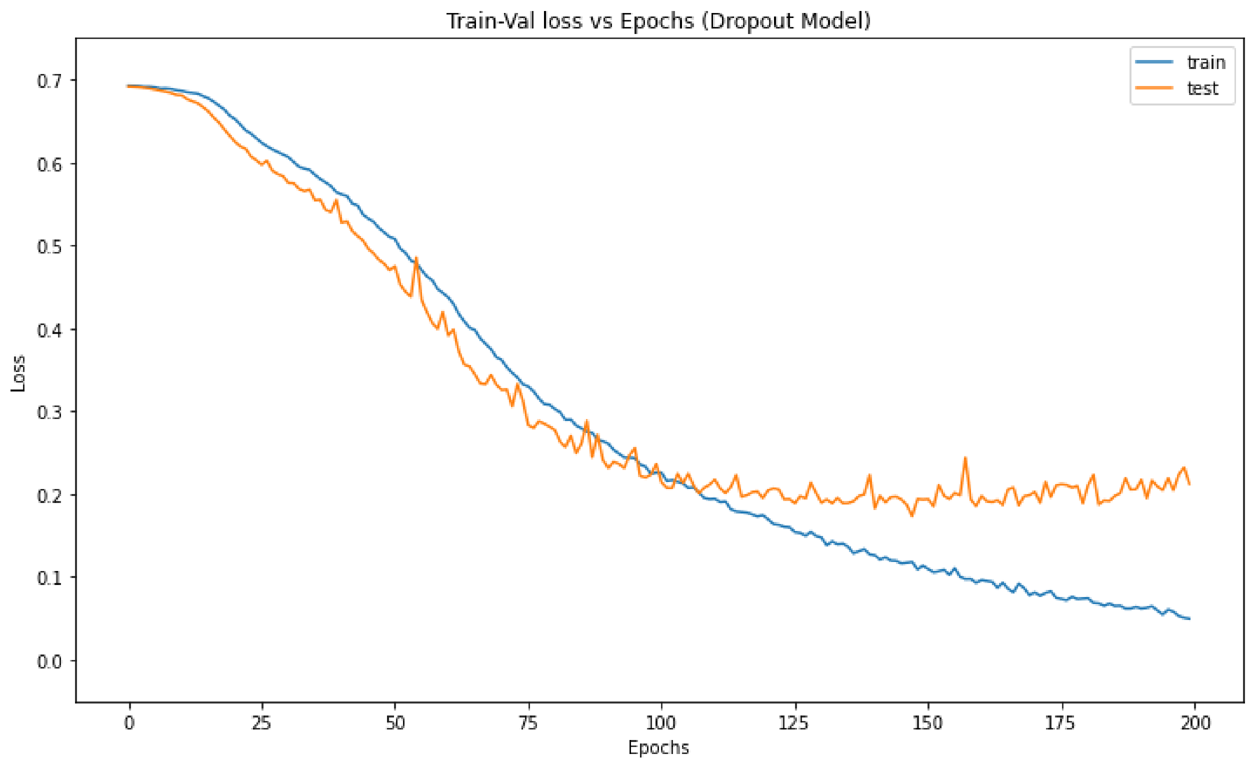


Figure 12. Train-test losses vs epochs for optimal dropout model

From figures 11 and 12, we can observe that overfitting of this model starts to happen at about 100 epochs into training (for both accuracy and loss). The test accuracy reaches convergence at about 150 epochs while test loss reaches its minimum point at about the same epoch, but continuously increasing afterwards. **The max value for test accuracy and min value for test loss amongst 200 epochs is 0.93599 and 0.1733, respectively**.

(Code for this part is found in from_scratch/optimal_dropout_gender.ipynb)

5.1.5. Experiments on Optimizers

a. Benchmark Results

Using the model with the optimal combinations of dropout rate and dropout layers configuration, we experiment with different optimizers in learning. Namely, we try out SGD with added momentum and RMSProp and Adam optimizer (at 2 different learning rates). The results are shown in the table 5 below. For more details on how the optimizers perform across each folds, refer to annex A, part d

Table 5. Overall Test Accuracy across 5 folds of different optimizers

Optimizer	Overall Accuracy
SGD (optimal model from 5.1.4)	87.24±2.21%
SGD, Momentum = 0.3	87.16±2.18%
SGD, Momentum = 0.8	86.94±2.40%
RMSProp (lr = 1e-3)	81.20±2.57%
Adam (lr = 1e-3)	60.00±4.76%
RMSProp (lr = 1e-5)	85.83±2.14%
Adam (lr = 1e-5)	85.67±2.24%

From the table 5, it can be deduced that there are no improvements as a result of using other type of optimizer than SGD. Hence the optimal model remains the same as the optimal model from the previous part.

Amongst these optimizers, Adam optimizer at learning rate = 1e-3 has the worst performance 60.00±4.76%. Its mean accuracy is the lowest amongst all optimizers being experimented while its standard deviation is extremely high comparing to the rest. Following Adam optimizer at learning rate = 1e-3 is RMSProp at 1e-3 learning rate with mean validation accuracy and std. of 81.20±2.57%.

We suspect that 1e-3 is a very large learning rate for these two optimizers hence our choice of experimenting with another learning rate of 1e-5 for both these optimizers. The results are on par with the baseline model performance referenced in [2].

The SGD optimizer with momentum did not give rise to a lift in mean validation accuracy, hence they are not selected for optimal model either.

The observations in this experiment confirm our understanding that SGD optimized model will have the best performance overall. However, one may select Adam optimizer or RMSProp if the time of training is important for them and only an acceptable performance is required. In our case, despite having a faster training time (in number of epochs) at the same learning rate as SGD, RMSProp and Adam perform very badly, especially Adam optimizer. We can try to offset this by reducing the learning rate, but this offset the main benefits of Adam and RMSProp as the models will now have to train for longer number of epochs due to slower learning rate.

(Code for this part is found in from_scratch/optimizer_gender_fold0.ipynb)

b. Optimal Model

There were no new optimal hyperparameter combinations found in this experiment, thus the optimal model for this experiment is the optimal model in 5.1.4 b. Please refer to the figures and discussion in that part.

5.2. Transfer learning with Google's EfficientNet

a. Benchmark Results

Similar to the approach taken for replicating *Levi and Hassner*'s work and using the same processed data of Adience benchmark dataset, the results for validation accuracy of the individual bags/iterations of each of the 5 folds are acquired and shown below in Table 6.

Table 6. Validation Accuracies per bag, per fold and overall, of the transfer learning model

Fold	**Bags**			**Average (fold) Accuracy**
	0	1	2	
0	0.92200	0.91074	0.91763	0.91679
1	0.91450	0.90412	0.88993	0.90285
2	0.89536	0.90598	0.90126	0.90087
3	0.93263	0.91711	0.93679	0.92884
4	0.87798	0.87004	0.87581	0.87461
			Overall Accuracy	90.48±1.94%

From Table 6, it can be seen that our EfficientNet-B5 based transfer learning model achieved an overall validation accuracy of **90.48% and a standard deviation of 1.94%**. This result is not only better than what was reported in the *Levi and Hassner* paper (85.9% ± 1.4%), but also the baseline CNN model that we have built from scratch (86.17 ± 1.77%).

Furthermore, it is to be highlighted that for every bag of every fold of the test, the optimal accuracies were mostly observed at epochs ranging from 5 to 10, which concludes a higher rate of convergence from the model compared to the Lean CNN.

(Code for this part is found in transfer_learning/effecientnet_implementation.ipynb)

b. Optimal Model Training

Again, we also trained the transfer learning model on the final train set and test it on the test set as mentioned in part b of 4.1.2. **The model achieved a final test accuracy of 96.24% and a min test loss of 0.1176**. Figure 13 and 14 shows the variations of train-test accuracy and loss vs training epochs respectively.

(Code for this part is found in transfer_learning/final_effecientnet_implementation.ipynb)

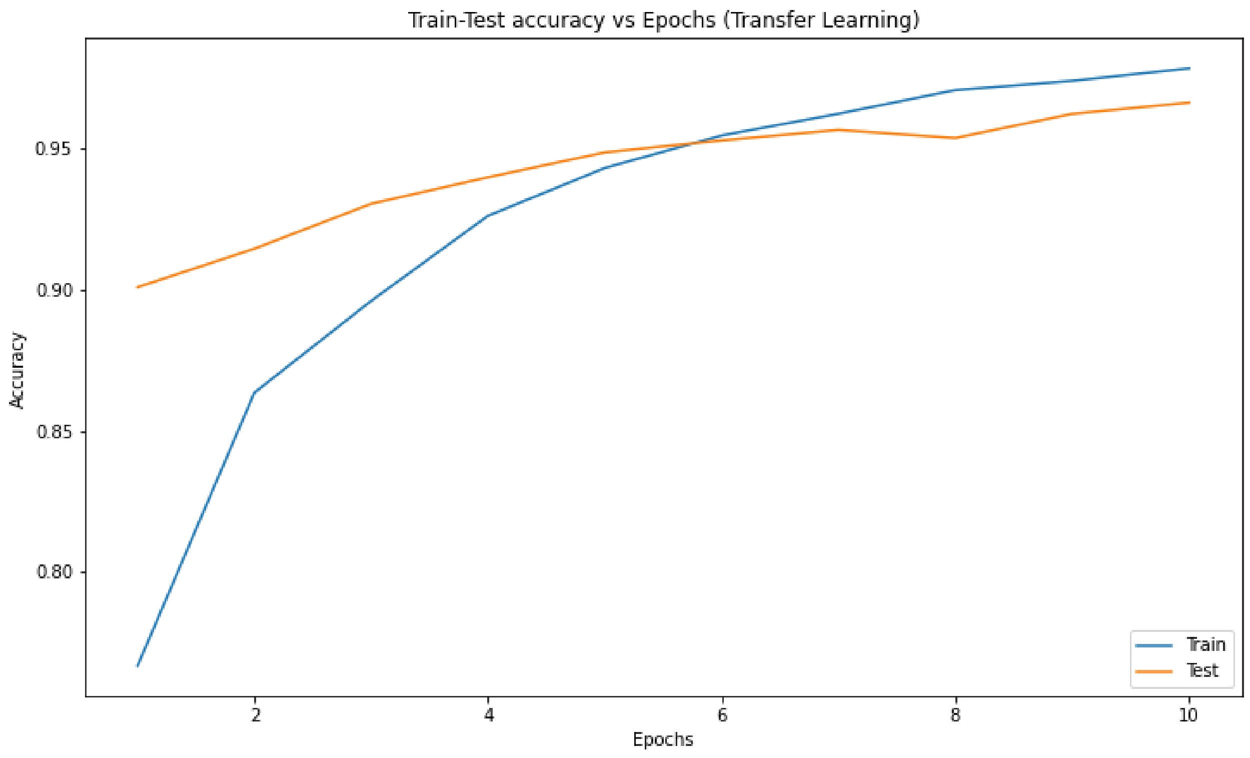


Figure 13. Train-Test accuracies vs epochs of transfer learning model

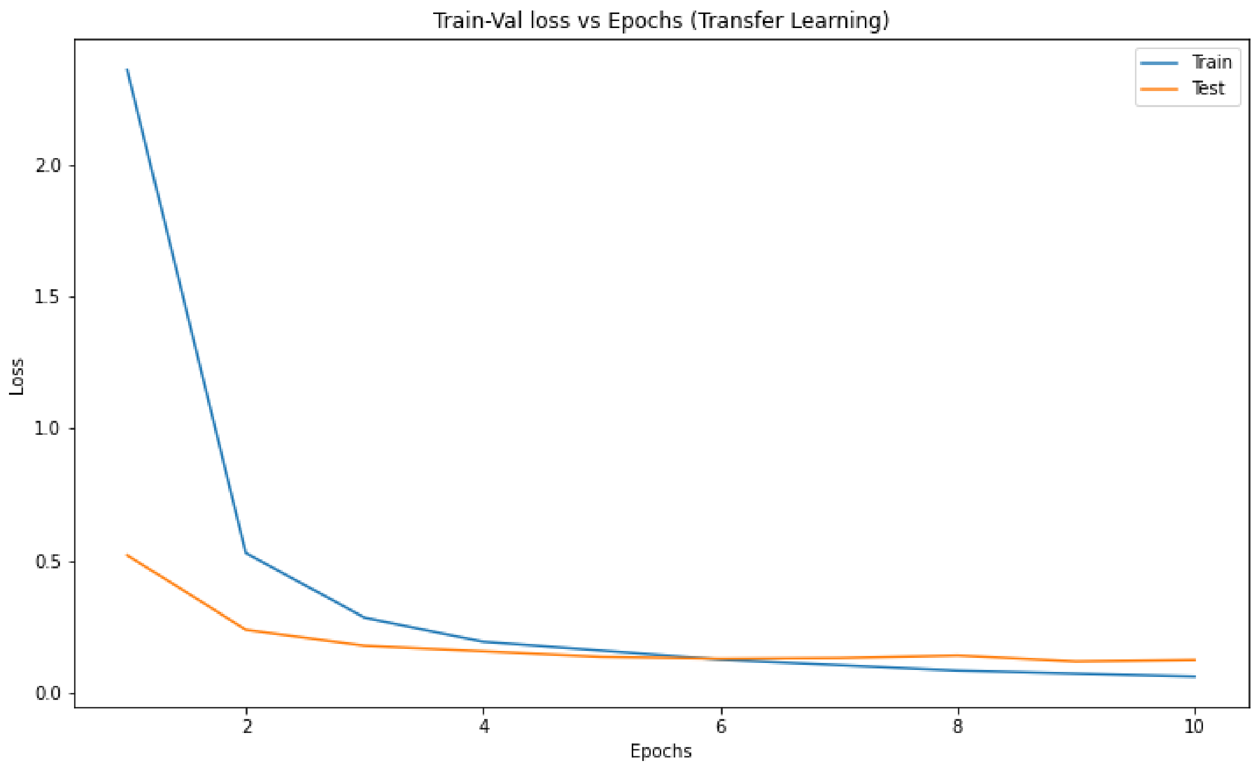


Figure 14. Train-test losses vs epochs for transfer learning model

6. Discussion

6.1. Discussion on Hyperparameters Tuning Experiments

a. General trends in training and test accuracies of optimal models

From Figure 7, 9 and 11, it can be seen that the optimal models' accuracies follow similar general trends. That is, they start with their test accuracies being higher than their train accuracies until the crossover point happens (at 40, 45 and 100 epochs respectively). From this point onwards, the test accuracy is smaller than the training accuracy and the gap between these two metrics will be expanding with more epochs of training.

Such trend of better testing accuracies recorded in the first part of the training process can be explained by the usage of dropouts in all three models, which will give rise to a different behavior in training and testing. During training of the baseline and optimal models, a portion of the features are discarded while in testing, all the dropouts or other regularization units are turned off, giving rise to a better testing performance. This effect is amplified with more aggressive dropout or regularization strategies, as shown in Figure 11, where we used 3 dropout layers instead of 2, which made the underfitting larger compared to the other 2 models.

At some point in training, overfitting starts to happen as the models start to memorize the noise and intrinsic features of the train dataset too well, which reduces its generalization capabilities and performance on the test set (test accuracies are smaller than train accuracies from this point onwards).

By adding more dropout regularization layers (in dropout optimal model), we managed to delay the starting point of the overfitting from 40 epochs to 100 epochs and extensively reduce the degree of overfitting. This is shown by the distance between train and test accuracies in Figure 7 and 9, which are larger than that in Figure 11, where the optimal dropout model with 3 dropout layers is used. Dropout works by temporary dropping different set of neurons during each training iterations, which means we are training different neural networks in every iteration. Since different networks overfit in different ways, when ensembled together, the overall overfitting is reduced. With more dropout layers, this means a larger portion of neurons in more fully connected layers are dropped, which can result in a statistically greater variety of networks being trained, therefore when combined together, their overall overfitting should be smaller than that of model with less dropout layers.

b. General trends in training and test losses of optimal models:

From Figure 8, 10, and 12, it can be seen that the optimal models' losses follow similar general trends. They start with their test losses being lower than their train losses until the crossover point happens (at 40, 45 and 100 epochs for the baseline, optimal width and optimal dropout models respectively). From this point onwards, the test loss is larger than the training loss, indicating overfitting. In additional, the train loss will converge while test loss will diverge after it hits the minimum value possible.

For the first half of the training process when test loss is smaller than training loss, similar explanation to part a is available. This is mainly due to the usage of dropout in our models.

For increasing test loss while test accuracy remains converged during overfitting, it can be explained that in this phase, as the model is 'memorizing' the train set to a greater extent, its prediction of the 'bad' images in the test set is getting worse too (as generalization capability decreases). As a result, during the overfitting phase, the test loss will 'blow up'. The difference between test accuracy (which converges in this phase shown in Figure 7, 9 and 11) and test loss (Figure 8,10,12) indicates that despite the model can predict the gender of images in test set with high accuracy, the confidence of its prediction is decreasing. This is why during cross validation training, we early stop the training on each fold when minimum validation loss is achieved and obtain the best validation accuracy from that point all the way back to the first epoch, since only within this search space, the validation accuracy of the model has the highest confidence. This is also to save time by not overtraining models.

Similar to optimal models' accuracies trend, by adding an additional dropout layer at the first FC input, we managed to reduce the effect of overfitting both in time and amplitude. This is shown in Figure 12, in comparison to Figure 8 and 10, where the test loss of the optimal dropout model increases at a smaller rate and at later epochs compared to the baseline and optimal width model. The optimal dropout model that uses 3 layers of dropout also starts overfitting at later epochs (100 epochs compared to about 40 epochs for the other 2 models).

c. Comparing performances of the optimal models

The performances of the three optimal models are judged based on their test accuracies similar to the cross-validation process. During their training processes, only the weights and bias from the best performing epochs are saved. The best test accuracies of the 3 optimal models after 200 epochs are shown below:

Table 7. Best Test accuracies of each optimal model

Model	**Best Test Accuracy**
Baseline	0.93371
Optimal Width	0.93286
Optimal Dropout	0.93599

As seen from Table 7, the test accuracy of the optimal dropout model is the highest, confirming our experiment result that it is indeed the best performing CNN model we can produce from the hyperparameter approach to predict gender from images of people in the wild.

Looking at its (optimal dropout model) train-test accuracies and losses curve from figure 11 and figure 12, we can also see it has the smallest overfitting in the three optimal above at 200 epochs. More explanation on why this is the case can be found in part a and part b of this section. Its test loss is also the lowest, suggesting the high confidence in its prediction comparing to the two optimal model at 200 epochs. Besides having the highest test accuracy, the dropout optimal model also has the highest generalization capabilities in predicting unseen image, thus it is our best model from our hyperparameter tuning approach.

6.2. Comparing the performances of optimal from hyperparameter tuning and transfer learning model.

a. Adience Cross-validation benchmark

The optimal model for building and tuning our lean CNN architecture is the model with dropout configuration (0.80, 0.50, 0.20) and number of filters in the 1st conv layer to be 48. Its 3 bags, 5 folds best validation accuracy readings are shown in annex A, part c. For the transfer learning model, its 3 bags, 5 folds best validation accuracy readings are shown in part a of section 5.2. For readability, the mean validation accuracy across 5 folds of these two models are shown in table 8 below.

Table 8. Mean Validation Accuracy for each model

Model	**Mean Validation Accuracy**
Optimal Hyperparameters (from 5.1)	87.24%
Transfer Learning (from 5.2)	90.48%

Clearly, the transfer learning model outperforms the optimal hyperparameter model in Adience cross-validation benchmark by roughly 3.24% in mean validation accuracy. Since we are evaluating models solely based on their average best validation accuracy across the 5 folds, it can be said that using transfer learning with the state-of-the-art EfficientNet model gives rise to a better model than the optimal hyperparameter lean CNN model that we spent more time on training and experimenting on.

We shall confirm their performances when trained on the full train and tested on the full test set next.

b. Optimal Model Performances

The best test accuracy and test loss as reported in part b of section 5.1.4 and 5.2 are consolidated in the table below.

Table 9. Details of optimal models from each experiment

Model	Best Test Accuracy	Best Test Loss	Number of Epochs
Optimal Hyperparameters (from 5.1)	0.9360	0.1733	200
Transfer Learning (from 5.2)	0.9624	0.1176	10

From the table, we can deduce that the transfer learning model also outperforms our model with optimal hyperparameters in testing on the test set too. The EfficientNet model is able to achieve a lower test loss and higher test accuracy with small number of epochs required in training.

In the later part (unfreeze training) of EfficientNet model, it can be observed that the overfitting of accuracy and loss in training this model is much smaller than the overfitting in training our best lean CNN with optimal hyperparameters too (figure 13 vs 11 and figure 14 vs 12).

These observations again emphasize the superior performance of transfer learning EfficientNet model to the best performing Lean CNN proposed by *Levi and Hassner* [2] in 2015, suggesting that most image recognition problem can be solved efficiently with transfer learning and better generic pretrained network like EfficientNet.

c. Number of Parameters and Training Time

Our optimal model from hyperparameter tuning of the architecture proposed by [2] has 11,097,153 parameters. On the other hand, the EfficientNet we are using has about 28,342,833 parameters to be trained per epoch. The larger EfficientNet proves to occupy more memory resource than our lean CNN implementation.

Also, while the training time for our lean CNN (per bag of 1 fold) is about 20 minutes, it took us up to 2 hours to train 1 bag of the transfer learning EfficientNet (despite shorter number of epochs needed to be trained). Therefore, on average, the training time that we spent for cross-validation benchmark of our EfficientNet model and the tuning of our lean CNN is almost comparable.

6.3. Pros and Cons of each approach

In this project, we have attempted two methods in building neural networks to solve the task of gender classification in the wild with the Adience benchmark dataset. Both methods have their own pros and cons.

For our first methodology of modifying the hyperparameters of the lean CNN published in [2], the main advantage is that the CNNs is very lean, allowing us to easily load them in for training. The training time for this method is also shorter due to a smaller number of parameters needed to be processed during training than the transfer learning model. Moreover, modifying the hyperparameters of this lean model gives us more control on our experiment as we can investigate every aspect that contributes to the performance of the model. However, this methodology also has a fair share of downsides too. For starters, the number of hyperparameters that we can tune is endless for a CNN that is built particularly for this gender classification problem. In addition, the search space for values to be experimented with those hyperparameters is also endless. In our experiments, we only selected a few hyperparameters and search spaces to look for, but in actuality, we are not even near exhausting any of these dimensions. Despite shorter training times per model, this approach can too be very time consuming as we need to experiment to find out the optimal combination of hyperparameters. This is an iterative and empirical process that takes a lot of patience to do. It can be guaranteed that the model's performance will improve, but the main problem is finding the optimal parameters to do so.

Regarding our second approach of this project, which is applying transfer learning on a pre-trained, award-winning deep CNN architecture, the main advantage that we are looking for is faster time to deliver. We only have to retrain our own classifier at the top level of the published architecture in [3] for a functional model. The performance from transfer learning is also very good compared to our first approach since the model that we have chosen was trained on a massive image dataset (such as ImageNet), which gave rise to its ability to extract features and generalization very well compared to our lean CNN, which is trained on the limited Adience dataset. However, the downside of this approach is larger memory usage for a very deep architecture and long training time (which is the case for newer pretrained model today due to the advancement of computing resources in researching). Regardless, this is still a fool-proof approach for software engineer to solve image recognition problems fast and efficiently.

7. Conclusion

In conclusion, our project managed to investigate two methodologies in solving a gender classification problem. The first method was building a lean CNN and tuning its hyperparameters from the ground up, specifically for the dataset in benchmark, while the second one is pretraining our model with a state-of-the-art generic, pretrained CNN such as EfficientNet. Both methods present better performance than what is reported in [2], while the transfer learning model is superior to the lean, optimal CNN due to its better generalization capabilities. It also takes less effort to develop as we only need to redesign the last classifier layers while for the former, a long amount of time is expected for experimenting around with the hyperparameters to find the best performing models.

Given the hectic workload of this semester, we think that there are many limitations in the project that we can improve in the future. For starters, we will try to experiment with a wider range of hyperparameter types and values so as to increase our performance in the first methodology. Next, we shall repeat the experiments with even more iterations that currently (5-10 iterations) so as to obtain more reliable results. Also, we shall conduct higher number of k-fold validation, potentially 10-fold so as to make our results more reliable too. Lastly, we can attempt to combine the networks develop in this project to another software development project, helping to bring our hard work to good use in reality.

8. References

[1] “Adience Dataset,” Face Image Project - Data. [Online]. Available: https://talhassner.github.io/home/projects/Adience/Adience-data.html. [Accessed: 22-Nov-2020].

[2] G. Levi and T. Hassner. Age and gender classification using convolutional neural networks. In IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) workshops, June 2015.

[3] M. Tan and Q. Le, "EfficientNet: Improving Accuracy and Efficiency through AutoML and Model Scaling", Google AI Blog, 2020.

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. CoRR, abs/1409.4842, 2014

Appendix A: Detailed Bags – Folds Results for Hyperparameter Tuning Experiments.

a. Experiment with Depth

4conv training results

Fold	Bags			Average
	0	1	2	
0	0.87020	0.87020	0.86178	0.86739
1	0.85012	0.86224	0.84112	0.85116
2	0.85051	0.86861	0.85838	0.85903
3	0.87585	0.87850	0.87472	0.87636
4	0.83069	0.83574	0.83718	0.83454
			Overall Average Accuracy	85.76±1.60%

2fc training results

Fold	Bags			Average
	0	1	2	
0	0.86521	0.87239	0.86396	0.86719
1	0.85462	0.84770	0.85358	0.85012
2	0.86113	0.86703	0.86467	0.86428
3	0.88380	0.88607	0.88494	0.88494
4	0.83682	0.83718	0.83755	0.83755
			Overall Average Accuracy	86.11±1.78%

1fc training results

Fold	Bags 0	Bags 1	Bags 2	Average
0	0.85835	0.86989	0.85179	0.86001
1	0.85185	0.85047	0.85012	0.85508
2	0.84461	0.84343	0.86497	0.84500
3	0.87926	0.88039	0.87888	0.87951
4	0.83791	0.82563	0.84007	0.83453
			Overall Average Accuracy	85.40±1.70%

2fc4conv training results

Fold	Bags 0	Bags 1	Bags 2	Average
0	0.86053	0.85928	0.86615	0.86199
1	0.84631	0.84770	0.85185	0.84832
2	0.84343	0.86900	0.84815	0.85353
3	0.88077	0.88683	0.88494	0.88418
4	0.83032	0.81877	0.83213	0.82707
			Overall Average Accuracy	85.51±2.08%

1fc4conv training results

Fold	Bags 0	Bags 1	Bags 2	Average
0	0.85616	0.86209	0.86771	0.86200
1	0.84735	0.86258	0.84804	0.85266
2	0.86231	0.86192	0.85877	0.86100
3	0.87926	0.88607	0.88039	0.88191
4	0.83755	0.83116	0.83357	0.83743
			Overall Average Accuracy	85.90±1.61%

b. Experiment with Width

(48,256,384,512) training results

Fold	Bags			
	0	1	2	Average
0	0.87051	0.86989	0.86396	0.86812
1	0.85981	0.85289	0.86258	0.85842
2	0.87175	0.86467	0.87333	0.86991
3	0.88342	0.88721	0.87926	0.88329
4	0.85415	0.83502	0.83863	0.8426
			Overall Average Accuracy	86.44±1.51%

(192,256,384,512) training results

Fold	Bags			
	0	1	2	Average
0	0.87083	0.86708	0.86552	0.86781
1	0.8612	0.85254	0.86189	0.85854333
2	0.84304	0.84382	0.84736	0.84474
3	0.87926	0.88494	0.88153	0.88191
4	0.83466	0.82744	0.83574	0.83261333
			Overall Average Accuracy	85.71±1.93%

(96,128,384,512) training results

Fold	Bags			
	0	1	2	Average
0	0.86084	0.86147	0.87176	0.86469
1	0.85774	0.86189	0.86362	0.86108333
2	0.86585	0.8572	0.85051	0.85785333
3	0.88683	0.88569	0.88229	0.88493667
4	0.83682	0.84043	0.84477	0.84067333
			Overall Average Accuracy	86.18±1.59%

(96,512,384,512) training results

Fold	Bags			
	0	1	2	Average
0	0.86427	0.86615	0.86864	0.86635333
1	0.8522	0.84804	0.84701	0.84908333
2	0.84776	0.84815	0.84815	0.84802
3	0.87585	0.88229	0.88494	0.88102667
4	0.84043	0.83682	0.84477	0.84067333
			Overall Average Accuracy	85.70±1.64%

(96,256,192,512) training results

Fold	Bags			
	0	1	2	Average
0	0.86677	0.86989	0.8702	0.86895333
1	0.86777	0.86224	0.85635	0.86212
2	0.8572	0.85995	0.85051	0.85588667
3	0.8838	0.88077	0.88456	0.88304333
4	0.82852	0.8361	0.83971	0.83477667
			Overall Average Accuracy	86.10±1.78%

(96,256,768,512) training results

Fold	Bags			
	0	1	2	Average
0	0.87738	0.8727	0.86927	0.87311667
1	0.8522	0.86639	0.86397	0.86085333
2	0.86074	0.86703	0.86428	0.86401667
3	0.87585	0.88039	0.88494	0.88039333
4	0.83357	0.8361	0.84007	0.83658
			Overall Average Accuracy	86.30±1.66%

(96,256,384,256) training results

Fold	Bags			
	0	1	2	Average
0	0.86583	0.86802	0.87207	0.86864
1	0.85497	0.85116	0.86224	0.85612333
2	0.84815	0.85602	0.86153	0.85523333
3	0.87434	0.88115	0.88039	0.87862667
4	0.84007	0.84296	0.83249	0.83850667
			Overall Average Accuracy	85.94±1.52%

(96,256,384,1024) training results

Fold	Bags			
	0	1	2	Average
0	0.87051	0.86178	0.87332	0.86853667
1	0.85808	0.84943	0.85843	0.85531333
2	0.8572	0.86231	0.85563	0.85838
3	0.88948	0.87926	0.87926	0.88266667
4	0.84224	0.82671	0.83899	0.83598
			Overall Average Accuracy	86.02±1.72%

(48,128,768,512) training results

Fold	Bags			
	0	1	2	Average
0	0.86895	0.88456	0.87613	0.87654667
1	0.86051	0.86501	0.84943	0.85831667
2	0.85602	0.85799	0.85445	0.85615333
3	0.88569	0.88456	0.88683	0.88569333
4	0.84621	0.84332	0.83538	0.84163667
			Overall Average Accuracy	86.37±1.75%

(96,128,768,512) training results

Fold	Bags			
	0	1	2	Average
0	0.87239	0.86958	0.8727	0.87155667
1	0.85808	0.86397	0.84874	0.85693
2	0.86585	0.86074	0.86389	0.86349333
3	0.8891	0.88607	0.88039	0.88518667
4	0.84188	0.83394	0.84801	0.84127667
			Overall Average Accuracy	86.37±1.64%

(48,256,768,512) training results

Fold	Bags			
	0	1	2	Average
0	0.87363	0.8702	0.87738	0.87373667
1	0.86258	0.86743	0.85358	0.86119667
2	0.86546	0.85366	0.85405	0.85772333
3	0.87964	0.87774	0.88342	0.88026667
4	0.83755	0.84513	0.84477	0.84248333
			Overall Average Accuracy	86.31±1.47%

(48,128,384,512) training results

Fold	Bags			
	0	1	2	Average
0	0.86708	0.87676	0.86677	0.87020333
1	0.85774	0.85531	0.85462	0.85589
2	0.86349	0.86703	0.86861	0.86637667
3	0.88456	0.88721	0.88607	0.88594667
4	0.83899	0.84007	0.83755	0.83887
			Overall Average Accuracy	86.35±1.75%

c. Experiment with Dropout

(0,50,50) training results

Fold	Bags			
	0	1	2	Average
0	0.87051	0.86989	0.86396	0.86812
1	0.85981	0.85289	0.86258	0.85842667
2	0.87175	0.86467	0.87333	0.86991667
3	0.88342	0.88721	0.87926	0.88329667
4	0.85415	0.83502	0.83863	0.8426
			Overall Average Accuracy	86.45±1.51%

(0,0,0) training results

Fold	Bags			
	0	1	2	Average
0	0.85959	0.85991	0.86427	0.86125667
1	0.84978	0.84666	0.84424	0.84689333
2	0.85208	0.85838	0.85563	0.85536333
3	0.88002	0.88002	0.88229	0.88077667
4	0.82274	0.83971	0.84513	0.83586
			Overall Average Accuracy	85.83±1.57%

(0,20,20) training results

Fold	Bags			
	0	1	2	Average
0	0.86396	0.86677	0.86271	0.86448
1	0.85635	0.84874	0.86847	0.85785333
2	0.85917	0.86192	0.86349	0.86152667
3	0.87926	0.87926	0.88266	0.88039333
4	0.84513	0.82563	0.84332	0.83802667
			Overall Average Accuracy	86.05±1.52%

(0,80,80) training results

Fold	Bags			
	0	1	2	Average
0	0.85803	0.87051	0.87395	0.86749667
1	0.86051	0.86951	0.86708	0.8657
2	0.86546	0.869	0.84894	0.86113333
3	0.88683	0.88304	0.89023	0.8867
4	0.83646	0.83574	0.83682	0.83634
			Overall Average Accuracy	86.35±1.80%

(0,50,80) training results

Fold	Bags			
	0	1	2	Average
0	0.86895	0.87239	0.86771	0.86968333
1	0.85324	0.87227	0.86535	0.86362
2	0.86271	0.85681	0.87018	0.86323333
3	0.88342	0.88266	0.8838	0.88329333
4	0.84079	0.85054	0.84657	0.84596667
			Overall Average Accuracy	86.52±1.35%

(0,50,20) training results

Fold	Bags			
	0	1	2	Average
0	0.85741	0.87426	0.86178	0.86448333
1	0.85497	0.85635	0.87504	0.86212
2	0.85759	0.86782	0.86743	0.86428
3	0.88304	0.8838	0.8838	0.88354667
4	0.84657	0.83646	0.84043	0.84115333
			Overall Average Accuracy	86.31±1.50%

(0,20,50) training results

Fold	Bags			
	0	1	2	Average
0	0.86895	0.86646	0.86303	0.86614667
1	0.85739	0.85047	0.84874	0.8522
2	0.85956	0.86546	0.86349	0.86283667
3	0.88191	0.88266	0.88266	0.88241
4	0.8296	0.82888	0.83646	0.83164667
			Overall Average Accuracy	85.90±1.88%

(0,20,80) training results

Fold	Bags			
	0	1	2	Average
0	0.86459	0.86396	0.86739	0.86531333
1	0.85635	0.85601	0.86154	0.85796667
2	0.85877	0.86271	0.86231	0.86126333
3	0.8785	0.88304	0.8838	0.88178
4	0.83935	0.83899	0.82816	0.8355
			Overall Average Accuracy	86.04±1.66%

(0,80,20) training results

Fold	Bags			
	0	1	2	Average
0	0.87363	0.86864	0.87207	0.87144667
1	0.87366	0.85531	0.86258	0.86385
2	0.87333	0.86349	0.86743	0.86808333
3	0.88304	0.88834	0.88039	0.88392333
4	0.84332	0.84116	0.83863	0.84103667
			Overall Average Accuracy	86.57±1.57%

(0,80,50) training results

Fold	Bags			
	0	1	2	Average
0	0.86521	0.87301	0.86927	0.86916333
1	0.86812	0.85462	0.86639	0.86304333
2	0.86389	0.86507	0.86113	0.86336333
3	0.88759	0.88077	0.89251	0.88695667
4	0.85126	0.85126	0.84874	0.85042
			Overall Average Accuracy	86.66±1.33%

(20,20,20) training results

Fold	Bags			
	0	1	2	Average
0	0.85959	0.86396	0.86427	0.86260667
1	0.85012	0.85704	0.85635	0.85450333
2	0.87451	0.85445	0.85641	0.86179
3	0.88456	0.88456	0.88191	0.88367667
4	0.8361	0.8496	0.82996	0.83634
			Overall Average Accuracy	85.98±1.70%

(50,50,50) training results

Fold	Bags			
	0	1	2	Average
0	0.86927	0.87332	0.85928	0.86729
1	0.8612	0.86466	0.86293	0.86293
2	0.86861	0.86507	0.88002	0.87123333
3	0.89251	0.90008	0.88683	0.89314
4	0.84043	0.83249	0.84043	0.83778333
			Overall Average Accuracy	86.65±1.98%

(20,50,80) training results

Fold	Bags			
	0	1	2	Average
0	0.87832	0.86552	0.87988	0.87457333
1	0.8702	0.86743	0.86535	0.86766
2	0.86467	0.86664	0.8631	0.86480333
3	0.89288	0.88796	0.89326	0.89136667
4	0.83574	0.83971	0.84477	0.84007333
			Overall Average Accuracy	86.77±1.86%

(50,20,80) training results

Fold	Bags			
	0	1	2	Average
0	0.87239	0.8702	0.8727	0.87176333
1	0.86743	0.86535	0.86604	0.86627333
2	0.86703	0.86153	0.87805	0.86887
3	0.89213	0.88683	0.89213	0.89036333
4	0.83971	0.84152	0.83069	0.83730667
			Overall Average Accuracy	86.69±1.91%

(20,80,80) training results

Fold	Bags			
	0	1	2	Average
0	0.86365	0.87207	0.87644	0.87072
1	0.86604	0.86847	0.85635	0.86362
2	0.87175	0.8631	0.86743	0.86742667
3	0.89175	0.8891	0.88796	0.88960333
4	0.84585	0.84116	0.8343	0.84043667
			Overall Average Accuracy	86.64±1.76%

(80,50,20) training results

Fold	Bags			
	0	1	2	Average
0	0.88456	0.87551	0.87582	0.87863
1	0.86881	0.87366	0.86777	0.87008
2	0.87175	0.8753	0.87136	0.87280333
3	0.89743	0.90613	0.90008	0.90121333
4	0.83755	0.84007	0.84079	0.83947
			Overall Average Accuracy	87.24±2.21%

d. Experiments with Optimizer

SGD, momentum = 0.3 training results

Fold	Bags			
	0	1	2	Average
0	0.88674	0.87488	0.88362	0.88174667
1	0.86812	0.8657	0.86985	0.86789
2	0.87097	0.86743	0.87687	0.87175667
3	0.89818	0.90121	0.8944	0.89793
4	0.83863	0.8361	0.84116	0.83863
			Overall Average Accuracy	87.16±2.18%

SGD, momentum = 0.8 training results

Fold	Bags			
	0	1	2	Average
0	0.87426	0.88424	0.89298	0.88382667
1	0.86535	0.86985	0.86362	0.86627333
2	0.87293	0.85484	0.87766	0.86847667
3	0.89705	0.90008	0.89099	0.89604
4	0.85054	0.82491	0.82166	0.83237
			Overall Average Accuracy	86.94±2.40%

RMSProp (lr = 1e-3) training results

Fold	Bags			
	0	1	2	Average
0	0.84337	0.82559	0.82995	0.83297
1	0.8072	0.79543	0.79993	0.80085333
2	0.80173	0.82022	0.79111	0.80435333
3	0.84936	0.83876	0.84027	0.84279667
4	0.78592	0.76823	0.78339	0.77918
			Overall Average Accuracy	81.20±2.57%

Adam (lr = 1e-3) training results

Fold	Bags			
	0	1	2	Average
0	0.66833	0.48986	0.5541	0.57076333
1	0.5739	0.54413	0.73486	0.61763
2	0.62038	0.63454	0.74705	0.66732333
3	0.54164	0.54315	0.54164	0.54214333
4	0.52888	0.74693	0.52996	0.60192333
			Overall Average Accuracy	60.00±4.76%

Adam (lr = 1e-5) training results

Fold	Bags			
	0	1	2	Average
0	0.86178	0.86708	0.86459	0.86448333
1	0.84978	0.86258	0.8477	0.85335333
2	0.86153	0.85917	0.869	0.86323333
3	0.8838	0.88266	0.87774	0.8814
4	0.81516	0.82527	0.82202	0.82081667
			Overall Average Accuracy	85.67±2.24%

RMSProp (lr = 1e-5) training results

Fold	Bags			
	0	1	2	Average
0	0.86521	0.86209	0.8571	0.86146667
1	0.85358	0.8567	0.85739	0.85589
2	0.86625	0.86192	0.86349	0.86388667
3	0.88531	0.88304	0.88607	0.88480667
4	0.81661	0.82744	0.83213	0.82539333
			Overall Average Accuracy	85.83±2.14%