Transfer Learning in High-Dimensional Regression

January 2025

Alexandra Zamfirescu alexandra.zamfirescu1@s.unibuc.ro
Adrian-George Leventiu adrian-george.leventiu@s.unibuc.ro
Elena-Denisa Nazare elena-denisa.nazare@s.unibuc.ro

Abstract

This project aims to investigate the key differences between fine-tuning and transfer learning in the context of high-dimensional regression. We took as an example the **MTCNN**[1] model to compare the performance, speed, and ease of use between pre-trained and fine-tuned versions. Our goal is to understand the strengths and limitations of each approach by evaluating the precision and adaptability of the model. See the full code on our Github.

1 Introduction

Using MTCNN as a case study, we seek to understand the strengths and limitations of each approach, Transfer Learning and Fine-Tuning of the model.

Transfer learning is a machine learning technique where a model trained on one task (Task A) is reused or fine-tuned for a different, but related, task (Task B). The key idea is that the knowledge gained from Task A can help in solving Task B, as the two tasks share some underlying similarities. While using a pretrained model can save time and resources, the performance on Task B may not match that of a model trained specifically for it, especially if the tasks are only loosely related.

On the other hand, **Fine-tuning** is an approach that involves modifying the pre-trained model's weights by continuing training on the target task. Fine-tuning allows the model to adapt to task-specific nuances, but it risks overfitting, especially when the target dataset is small or poorly aligned with the source domain.

[1]Multi-task Cascaded Convolutional Networks is a deep learning architecture designed for face detection.

Alexandra and Elena focused on gathering relevant data to thoroughly understand the project's objectives and challenges, particularly regarding the comparison between fine-tuning and transfer learning in high-dimensional regression, while also preparing an approach for tackling the experimental part.

Adrian was responsible for coding and executing the experimental implementations, including the adjustments necessary for both the pre-trained and fine-tuned versions of the MTCNN. Adrian also made sure to explain the logic behind the code to the other members of the team, ensuring a shared understanding of the implementation details and enabling everyone to contribute effectively to discussions and decision-making throughout the project.

Alexandra and Elena concentrated as well on synthesizing all gathered information, results, and insights into a cohesive and comprehensive report document. This included analyzing the outputs from the experiments, structuring the report to reflect the project's findings, and ensuring clarity and precision in presenting conclusions and limitations.

We conducted several experiments (some of them unsuccessful) to address this project, such as the following:

- augmenting the dataset, adding non-faces bounding boxes by randomly sampling patches in the picture that do not overlap with a face in order to help the model generalize better and improve the performance;

- getting more pictures of anything but faces using **Hard Negative Mining**[2] in order to add some more layers of filtering regarding faces (unsuccessful);

- we examined the training dataset to identify features that might indicate a face, such as contours (unsuccessful);

2 Approach

2.1 Brainstorming and Development

We found a dataset of images from Dexter's Laboratory to train a facial recognition model. To process and analyze the faces, we searched for a neural network specialized in facial recognition that can also be fine-tuned and discovered the MTCNN model, which proved well-suited for the task.

MTCNN is a neural network originally trained on real-life human faces (Task A) for tasks like face detection and alignment. We applied it to facial recognition on cartoon characters from Dexter's Laboratory (Task B), which is somehow similar but at the same time it differs significantly from real-life images.

We reviewed some base codes from **Github**[3] to grasp the initial approach.

Following the initial evaluation, we wrote the code for the initial execution of the pre-trained model. Later, we tried to modify the thresholds in order to maximize the precision of the model.

The next part involved a comprehensive analysis of the model architecture, identifying the specific components that required training, and determining the appropriate training methodologies in order to start the second approach, the fine-tuned version of the model.

Next, we developed the code for the fine-tuned version of the MTCNN, ensuring that it was adapted to the specific dataset we were using.

As part of the evaluation process, we analyzed the output data from the entire data set of input pictures given. This included examining the plots that represented the average precision.

2.2 Prerequisites

Python was the programming language used with the following libraries:

numpy	**matplotlib**
opencv-python	**torch**
pickle	**facenet**
PIL	**ntpath**

Table 1: Libraries Used

2.3 Work Flow

This section outlines the workflow for the face detection pipeline, from training data preprocessing to predicting the output. The pipeline consists of several key stages:

2.3.1 Training Data Preprocessing

2.3.1.1 Data Source The dataset consists of images of faces (*good*) and non-faces (*no_good*) stored in separate folders. Positive examples (faces) are extracted from annotated bounding boxes in the training images, while negative examples (non-faces) are sampled from regions without faces.

2.3.1.2 Positive Sample Generation For each training image, bounding boxes for faces are extracted using `preprocess_images()`. The images are resized to a fixed patch size (*12x12 pixels*). The patch size was selected because MTCNN's PNet layer operates optimally with 12x12 pixel patches. After conducting thorough research, we discovered that this specific patch size is crucial for the layer to effectively detect facial features at an early stage. By standardizing the input size, we ensure consistent performance and accuracy in the subsequent stages of the network.

2.3.1.3 Negative Sample Generation Non-overlapping patches of size *12x12 pixels* are randomly sampled from regions that do not overlap with annotated face bounding boxes (**IoU**[4] < 0.3).

[2] A technique used in digital image processing for getting non-face images that are easily confused with faces in order to reinforce the model.

[3] A platform for version control and collaboration, enabling developers to manage and share code.

[4] Known as Intersection over Union, it measures the overlap between a predicted bounding box and the ground truth, defined as the ratio of their intersection area to their union area.

Each patch is saved as a training example, labeled appropriately (1 for face, 0 for no face).

2.3.2 Neural Network

2.3.2.1 MTCNN We use the MTCNN model to differentiate between patches containing faces and those without faces. The thresholds parameters were tested for multiple values; *[0.1, 0.2, 0.2]* is used for simplicity.

2.3.2.2 Custom PNet Layer Creation We created a custom PNet layer for MTCNN that performs initial face detection by scanning the input image using convolutional layers to extract features, applying activation functions for non-linearity, pooling for dimensionality reduction, and finally predicting face classification and bounding box regression.

2.3.2.3 Layer Training Our function trains the PNet layer of the MTCNN model by iterating through the dataset, calculating classification and bounding box regression losses, and updating the model's parameters using backpropagation and optimization to improve detection accuracy.

2.3.3 Project Run

This section is shared between both model versions. The key difference is that the fine-tuned version loads the pre-trained PNet layer from saved files.

We process all images in the testing dataset, allowing our model to make predictions. The output is then saved for further analysis.

2.4 Results

2.4.1 Aggregation

The final output includes bounding boxes of detected faces and their confidence scores. These results are aggregated and saved for all test images.

2.4.2 Evaluation

We use the code provided by one of our Computer Vision instructors to evaluate the predictions, and we acknowledge that the credit for its creation belongs to them.

This code computes the IoU, which measures the overlap between two bounding boxes by dividing the area of their intersection by the area of their union. This metric assesses the degree of overlap between two bounding boxes. If the maximum IoU exceeds a threshold (0.3 in this case) and the ground truth box has not been matched to another detection, it is considered a true positive. Otherwise, it is a false positive. The code also calculates cumulative true positives and false positives, and finally, it determines recall and precision.

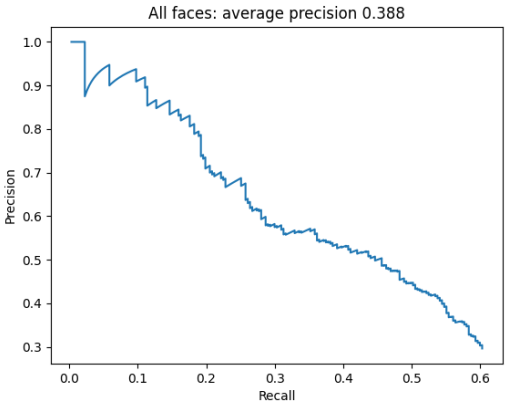

Figure 1: Final - Transfer Learning

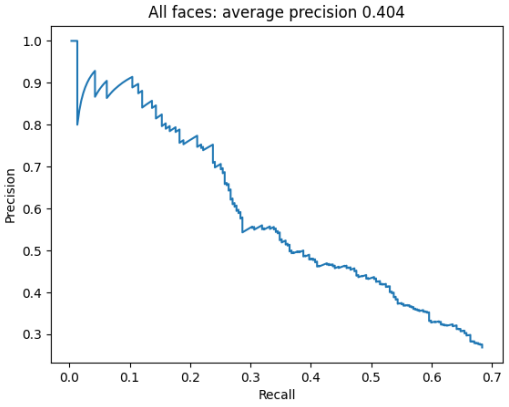

Figure 2: Final - Fine-tuning

2.5 Execution Time Estimation

For a 200 testing sample dataset with the pre-trained model, it takes about 20 seconds to process and create all needed files. If the model needs to be trained, it would take about 2 minutes for training and running (depending on the system that the project is being run on).

3 Limitations

The limitations of this project are mainly due to resource constraints. However, these are the following limitations that we encountered:

- the training dataset used for fine-tuning the model was relatively small, which likely limited the model's ability to generalize effectively
- the hardware used for training posed significant challenges; insufficient CPU cores and GPU resources made the process slow and inefficient, often pushing our laptops to its limits and causing frequent interruptions
- a lack of in-depth expertise in training neural networks of this complexity further constrained the project's progress, highlighting the need for future investigation and learning to optimize both the model and the training pipeline

4 Conclusions

4.1 Project

The pre-trained MTCNN model demonstrated superior speed, making it more efficient for quick deployments.
However, the fine-tuned model showed a slight performance improvement of 4.1% from the pre-trained version, suggesting better adaptation to the specific dataset. We think that with more data for the task, we could have achieved a better result. The challenges in fine-tuning highlight the need for more data, robust debugging and potential adjustments in the training process.
Overall, the choice between pre-trained and fine-tuned models will depend on the specific requirements of speed versus performance for the task at hand.

4.2 Group

From our perspective, we could have improved our organization and communication to achieve better results. Greater involvement from all sides would have also been beneficial given that the time frame wasn't ideal for us as we were juggling numerous other projects and exams simultaneously. Despite this, we believe that we did a pretty decent job.

5 Resource Catalog

- **Medium**
  https://towardsdatascience.com/how-do-you-train-a-face-detection-model-a60330f15f45
- **Github**
  https://github.com/reinaw1012/pnet-training/blob/master/PNet_Training.ipynb
  https://github.com/baomingwang/MTCNN-Tensorflow
- **MTCNN**
  https://mtcnn.readthedocs.io/en/stable/training/
- **Feature Representation Analysis of Deep Convolutional Neural Network using Two-stage Feature Transfer—An Application for Diffuse Lung Disease Classification**
  https://core.ac.uk/outputs/250629672/?source=2
- **Chat GPT**
  https://chatgpt.com/
- **Copilot**
  https://copilot.microsoft.com/