Group 1: Jukka Pelto-aho, Teemu Tiainen, Jan Tilles, Mika Vilpo

# **Final Report: ML Model Training Overview**

## **Introduction**

This report provides an overview of the machine learning model training process implemented in the Jupyter Notebook assignment\_with\_pretrained\_model.ipynb. The goal of the training process was to generate a model capable of predicting whether a person in a webcam feed is wearing different types of headwear or not. The model was designed using transfer learning with a pretrained model as a feature extractor. The steps include model selection, dataset preparation, training, fine-tuning, and evaluation.

Additionally, a separate script, webcam-predictor.py, was developed to utilize the trained model for real-time classification from a webcam feed. The script captures video input, detects faces using MTCNN, extracts and preprocesses the head region, and classifies headwear using the trained model. The live video feed displays predictions with confidence scores and bounding boxes, allowing real-time headwear detection.

## **1. Model Selection**

A separate code was made for comparing different Keras Models for this assignment (comparing\_keras\_models.ipynb). Models were chosen from here: [Keras Applications](https://keras.io/api/applications). The idea was to leave a laptop running overnight and check the results in the morning. It took a couple of runs before achieving success.

Several pretrained models were considered and evaluated on test data. The following table presents the test loss and test accuracy for each model:

|  |  |  |
| --- | --- | --- |
| **Model Name** | **Test Loss** | **Test Accuracy** |
| ResNet50V2 | 0.2777522802352905 | 0.9287598729133606 |
| Xception | 0.18646970391273499 | 0.9472295641899109 |
| MobileNetV2 | 1.1013202667236328 | 0.831134557723999 |
| EfficientNetB0 | 1.2719537019729614 | 0.8205804824829102 |
| DenseNet201 | 0.31997838616371155 | 0.9076517224311829 |
| InceptionV3 | 0.34708961844444275 | 0.9102902412414551 |

The chosen model, **Xception**, was used as a base model, leveraging its pretrained ImageNet weights. The model’s top layers were removed, and new layers were added to adapt it to the specific classification task.

## **2. Data Preparation**

The dataset we used, was downloaded from here: <https://www.kaggle.com/datasets/mantasu/face-attributes-grouped/data> .

The dataset was structured into training, test, and validation sets.

* **Training data augmentation**: Techniques such as rotation, width/height shifting, zooming, and horizontal flipping were applied to improve generalization.
* **Validation set**: Only rescaled to normalize pixel values.

Data was loaded using the ImageDataGenerator API, which enabled batch-wise image loading and preprocessing.

## **3. Model Training Process**

### **Phase 1: Training the New Layers**

1. The **pretrained Xception model was frozen** (weights not updated).
2. New layers were added:
   1. **GlobalAveragePooling2D** for feature extraction.
   2. **Fully connected layers** with ReLU activation and dropout to improve robustness.
   3. **Softmax layer** for multi-class classification.
3. The model was compiled with the **Adam optimizer** and **categorical cross-entropy loss**.
4. Training was conducted using **early stopping** and **model checkpointing** to avoid overfitting.

### **Phase 2: Fine-Tuning the Model**

1. The **base model was unfrozen** to allow fine-tuning.
2. A **lower learning rate (0.0001)** was used to prevent drastic weight updates.
3. Training was resumed for additional epochs with the same early stopping and checkpointing strategies.

## **4. Model Evaluation and Performance Monitoring**

* **Early stopping** was used to monitor validation loss and stop training when it stopped improving.
* **Model checkpoints** saved the best performing weights.
* **ReduceLROnPlateau** was used to dynamically adjust the learning rate when training plateaued. Specifically, if the validation loss did not improve for a predefined number of epochs, the learning rate was reduced by a factor (e.g., 0.5). This helps the model converge to a better minimum by taking smaller steps when loss stagnation is detected, improving performance, and preventing unnecessary weight updates that might cause instability.

## **5. Real-Time Classification with Webcam**

The trained model was integrated into a real-time classification script, webcam-predictor.py, which performs the following steps:

1. **Video Capture**: Opens a webcam feed and processes frames in real time.
2. **Face Detection**: Uses MTCNN to detect faces and extract the head region.
3. **Preprocessing**: Resizes the head region to match the input size required by the model and normalizes pixel values.
4. **Prediction Smoothing**: Maintains a rolling buffer of predictions to reduce noise in classification results.
5. **Visualization**: Draws bounding boxes around detected faces and overlays classification labels with confidence scores.
6. **User Interaction**: The live feed is displayed in a window, and the program exits when the user presses 'q'.

Depending on the computer, the video detection time ranged from 80ms to 400ms.

## **6. Model Saving and Deployment**

The final trained model was saved as "bestmodel.keras", ensuring it can be loaded later for predictions without retraining.

## **7. Challenges**

The model training and evaluation process involved several challenges that required adjustments and testing. Finding the best model for the task was one of the main difficulties. We tested different pretrained models, and it took multiple runs to find the right balance between accuracy and efficiency before deciding on the final model.

While working on the multiclass model, we explored alternative approaches to improve performance. To simplify the problem, we attempted to train a YOLOv8n model with a different dataset, focusing on just detecting the presence of a hat. While the model was effective at identifying actual hats, it struggled with false positives by classifying various objects placed on the head as hats. For example, when having headphones on, the model would classify it as a hat. This was likely due to the dataset containing only images with hats and no negative examples.

## **Conclusion**

This approach leveraged **transfer learning** to train an image classification model efficiently. The use of a **pretrained model** reduced training time and improved accuracy, while **fine-tuning** further optimized performance. Data augmentation and early stopping helped prevent overfitting, leading to a robust final model.

The real-time classification implementation with webcam-predictor.py demonstrates the practical deployment of the model for live webcam-based inference, making it suitable for applications requiring real-time headwear detection.