EE 514 Machine Learning Project Report

Instructor: Dr Zubair Khalid

Amina Batool	2025-10-0253
Rayaan Sohaib	2025-10-0280
Lina Nouman	2026-10-0297

Department of Electrical Engineering
Syed Babar Ali School of Science and Engineering
Lahore University of Management Sciences
Pakistan

Abstract

In this project, our aim was to develop a machine learning model capable of performing age and gender inference from facial images and deploy it on edge devices. We used ResNet-50 pretrained on ImageNet and fine-tuned on the UTKFace dataset. After encountering serialization challenges during deployment with TensorFlow.js due to version incompatibilities, we retrained the model using TensorFlow 2.11 and Keras 2.11. This allowed us to successfully convert it into TensorFlow.js format and run it in-browser via a React.js frontend. The final model demonstrates accurate real-time inference on webcam input, making it suitable for resource-constrained devices with privacy requirements.

Contents

Age and Gender Prediction (GenLens)

1.1 Introduction

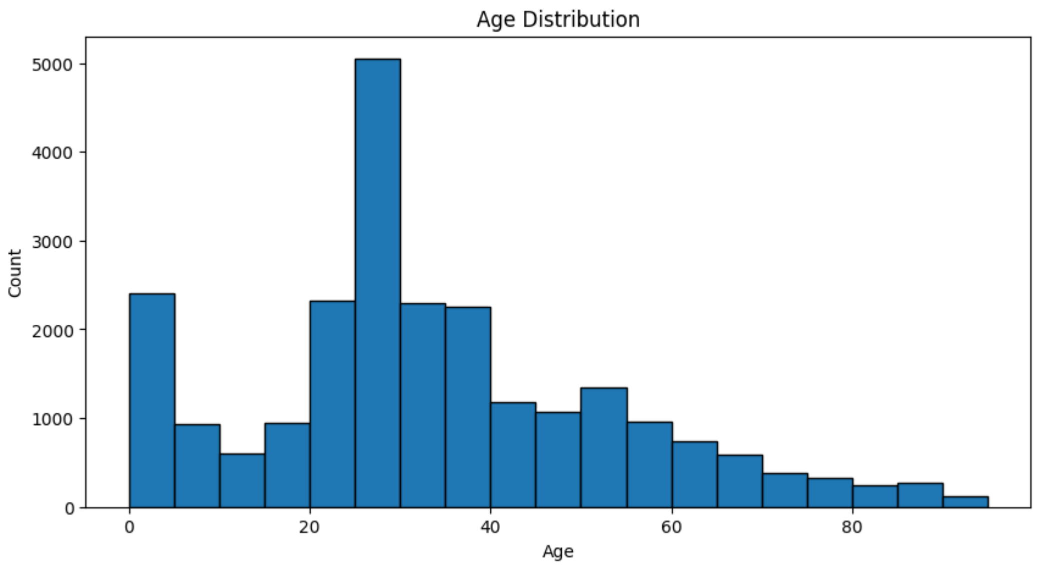
In recent years, machine learning has become increasingly central to real-world applications, especially in fields like computer vision. Our project focuses on building an age and gender prediction model that can run efficiently on edge devices, such as mobile phones or browsers. The goal was to create a system that not only gives accurate predictions from a single facial image but also works in real time without relying on cloud-based computation. Initially, we trained a ResNet-50 model using the UTKFace dataset, treating age prediction as a regression task and gender classification as a binary classification task. Later on, we trained RestNet-152 with a Bayesian approach. Beyond just training a high-performing model, we explored deployment strategies using TensorFlow.js to make the model lightweight and compatible with in-browser inference. This project allowed us to gain practical experience with deep learning, model optimization, and real-world deployment challenges on constrained devices.

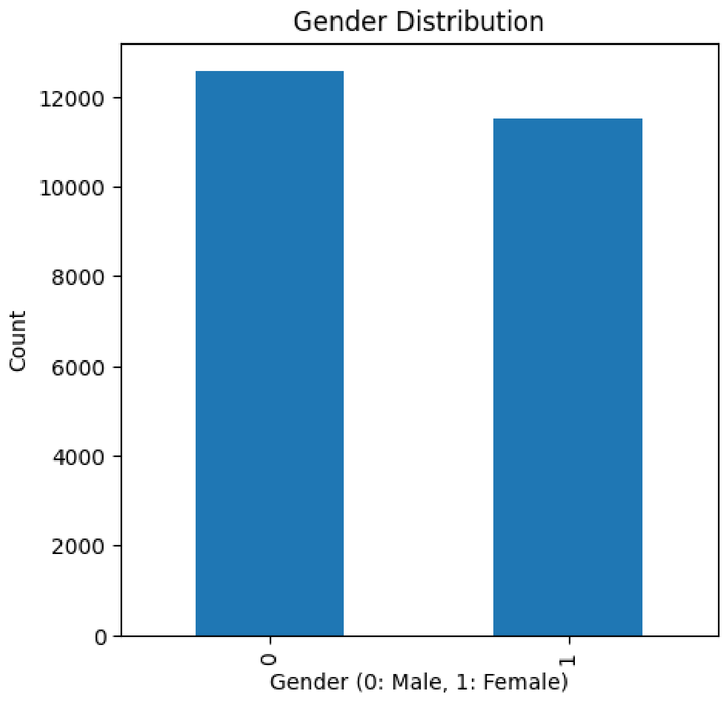
1.2 Literature Review

To build a strong foundation for our project, we looked into existing work on facial attribute prediction. One of the most relevant datasets we came across was FairFace by Kärkkäinen and Joo (2019), which focuses on creating a more balanced dataset across race, gender, and age. Although we used the UTKFace dataset in our project, FairFace helped us understand the importance of demographic balance and the potential biases that can exist in facial data. It made us more aware of how model performance can vary across different groups, for example if your training data has more images of a certain race than the other. We also referred to the work of Levi and Hassner (2015), who used a relatively simple CNN architecture for joint age and gender classification and showed that even basic models could perform surprisingly well on this task. Their results gave us a performance benchmark and helped justify using separate output heads for age and gender, which we also applied in our model. Together, these papers guided both the ethical considerations and the structural choices we made throughout our implementation.

1.3 Dataset and Preprocessing

The UTKFace dataset consists of 24000 labeled images with a variety of ages and ethnicities. Since ethnicities were not a focus of our project they were pre processed but unused at the time of training.





The dataset had a fairly even spread when it came to gender but in age very young (<20) and older ages (>40) were lacking. Knowing this we knew that the model might perform better on the age range of 20-30 than outside it, which was consistent with our testing. The data was loaded, resized (224,224) , converted to tensors, normalised and packed into 24 chunks of pkl files for training later.

1.4 Model Architecture

For all of our models the CNN was built on a standard ResNet backbone. We remove the original classification layer and treat the 2048/512-dimensional feature map as a shared representation. Early blocks remain frozen only the later, higher-level blocks are fine-tuned on our dataset.

From the shared backbone, we apply global average pooling:

- Age head (regression): a pair of fully connected layers (with batch-normalization, a leakyrelu, and dropout in between) that maps from the pooled features to a single continuous output.

- Gender head (classification): (again with normalization, leaky_relu activation, and dropout) that produces a single logit, followed by a sigmoid to yield a male/female probability.

This basic design was used throughout with simple changes / tuning of hyper-parameters made as we transitioned from ResNet 34 to 152 and then to bayesian implementations (for bayesian the loss function was adjusted to accommodate the kl divergence term).

1.5 Deployment and Frontend Integration

The deployment phase of our project involved several iterations of experimentation, troubleshooting, and optimization to enable our trained machine learning model to run directly in web browsers and on edge devices. This was one of the most challenging yet rewarding parts of our workflow, as it required bridging the gap between deep learning and real-time frontend implementation, from conversion to browser integration and performance logging. Below is a breakdown of the deployment journey:

1.5.1 Model Export and Format Conversion

Our model was initially trained in Keras (with TensorFlow backend), using the .h5 format. However, we quickly encountered compatibility issues when attempting to convert the model to a format usable by TensorFlow.js. Specifically, TensorFlow.js requires a .json architecture file and associated .bin weight files, but Keras 3.x introduced breaking changes that made direct conversion unstable or impossible. There were endless compatibility errors, where one version of the .h5 file would be of a higher keras version, and the converter would ask for a lower version, however, the lower version would have even more compatibility issues.

To fix this, we downgraded our training environment to TensorFlow 2.19, which restored compatibility with the tensorflowjs_converter tool, but on the condition that we convert the file into an .onnx file first. After retraining the model in this environment, we successfully converted the .onnx file into TensorFlow.js Layers format (model.json and binary weight files). This step was critical, as it enabled the model to be loaded directly in the browser without a server or external API.

1.5.2 Frontend Setup using React.js

We chose React.js for our frontend framework due to its modularity, responsiveness, and ease of integration with JavaScript-based ML libraries like TensorFlow.js. The frontend consists of the following key components:

- **Webcam Access and Frame Capture**
  We used getUserMedia() from the WebRTC API to capture video directly from the user's webcam. The captured frames are rendered onto an off-screen canvas element, which allows us to extract the image data and feed it into the model for inference.

- **Image Preprocessing**
  The raw webcam frame is resized and normalized using TensorFlow.js's tf.browser.fromPixels() and tf.image.resizeBilinear() functions. The image tensor is reshaped to match the model's input format (224x224x3), converted to float32, and normalized to the [0, 1] range, replicating the preprocessing pipeline used during training.

1.5.3 Model Loading and Real-Time Inference

Once the model was converted into TensorFlow.js format (model.json and .bin weights), we loaded it asynchronously in the browser using tf.loadGraphModel('/tfjs_model/model.json'). This allowed us to keep the inference entirely client-side, preserving user privacy and reducing latency. Because of its size, we observed that load time varied by device—averaging 3–5 seconds on most laptops and Android phones, and up to 30+ seconds on iOS due to Safari's memory restrictions.

The React app triggers inference when the user clicks the “Capture & Predict” button. Here's what happens under the hood:

1) The current frame from the webcam is drawn on an invisible canvas (<canvas> element).

2) Using tf.browser.fromPixels(), the canvas is converted into a tensor.

3) The image tensor is resized to (224, 224), normalized to the [0, 1] range, and reshaped to match the model’s expected input: [1, 224, 224, 3].

4) The model then returns two outputs:

    a) A floating-point number for age (regression output).

    b) A probability value (0 to 1) for gender, which is thresholded at 0.5 to determine “Male” or “Female.”

The predicted age is now rounded using Math.round() to present a more natural and intuitive output to the user, as decimal age values (like 24.7) are not commonly used. This also improves UI clarity. The gender prediction is interpreted based on whether the probability is greater than 0.5.

1.5.4 Displaying Results in the UI

We designed the interface to show real-time predictions directly over the webcam feed. Once a prediction is made:

- The predicted age and gender are displayed in a styled card below the webcam.

- The gender text color changes dynamically — pink for female, blue for male — to improve visual distinction.

- A small caption below shows the time taken for the last prediction in milliseconds, giving the user insight into how fast the model is operating.
- For iOS, we added a warning message about camera load delays and ensured proper behavior to prevent full screen video playback issues.

This part of the implementation helped us understand the importance of responsive design and visual feedback when building real-time ML systems.

1.5.5 Logging Inference Metrics to Firebase

To track usage and test behavior across different devices, we also implemented a real-time logging system using Firebase Realtime Database.

Each time a prediction is made, the following metadata is logged:

- Timestamp (ISO string)

- Device information (via navigator.userAgent)

- Predicted age (rounded integer)

- Predicted gender

- Prediction time in milliseconds

These records are pushed to a “metrics” collection in the Firebase database. This helped us analyze latency patterns and device-specific performance, and provided a basic audit log of inference sessions. We could then compare latency across devices, average prediction times vs model load times, and device-specific behavior and error patterns.

1.5.6 Testing on Edge Devices and Browsers

To validate the robustness of the deployment, we tested the React and TensorFlow.js app on multiple devices and environments:

- Laptop: Chrome on Windows 11 (Intel Core i7, 16 GB RAM)
- Android Phone: Chrome on Android 12 (Snapdragon 732G, 6 GB RAM)
- iPhone: Safari on iOS 16

Key Observations, covered in more detail in results and analysis:

- The model loaded within 3–5 seconds on all devices after initial load, which usually took around 30 seconds to 3 minutes.
- iOS required special handling using the playsInline attribute and specific canvas configuration to avoid rendering delays.
- Webcam permissions and privacy prompts varied between browsers but were generally handled gracefully by our app.

1.5.7 Deployment Challenges and Workarounds

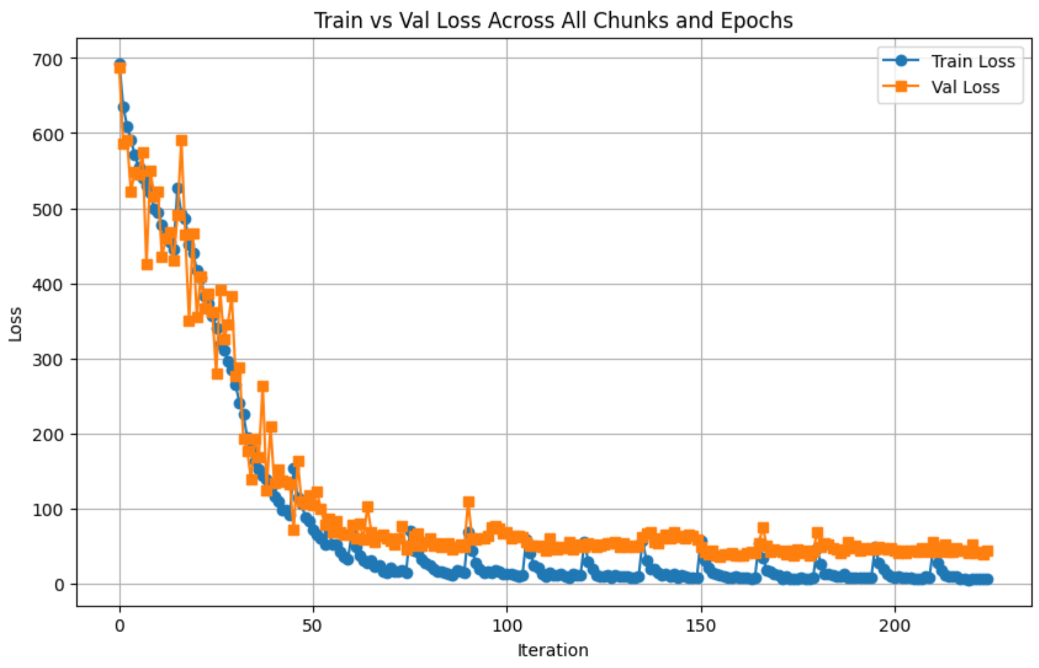
The biggest hurdle was the incompatibility between Keras 3.x and TensorFlow.js converter. This required us to retrain our entire model in an older, stable environment (TensorFlow 2.19) to enable conversion. We fixed our pipeline by converting it to an .onnx file first, then to a tensorflow lite file 32 bits, which was converted to .json and .bin format. Upon deployment, we encountered device-specific quirks, especially on Safari and iOS, where webcam behavior was inconsistent without inline play or fallback UI. Moreover we noted that the model performed better on Safari than chrome for iOS users. Moreover, our predictions would sometimes get stuck and not update as fluidly, so managing memory in the browser using tf.engine().startScope() and endScope() was critical to prevent memory leaks from repeated inference calls. We implemented scope cleanup and tensor disposal for every prediction to maintain smooth browser performance. Web deployment was debugged through using console and printing out raw predictions and tensor shapes to confirm any discrepancy between our model output and inference.

1.6 Performance Evaluation and Results

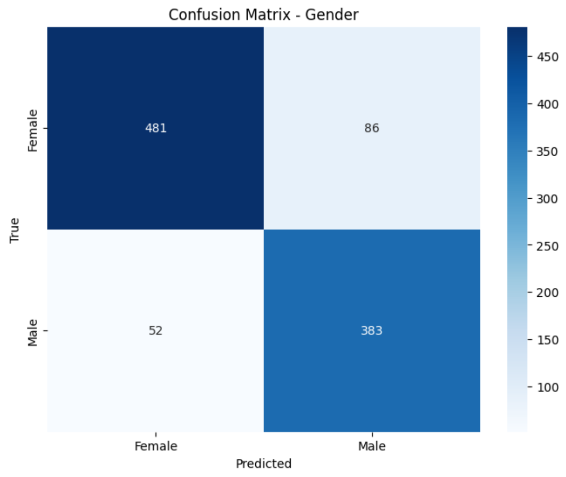
1.6.1 Training Results

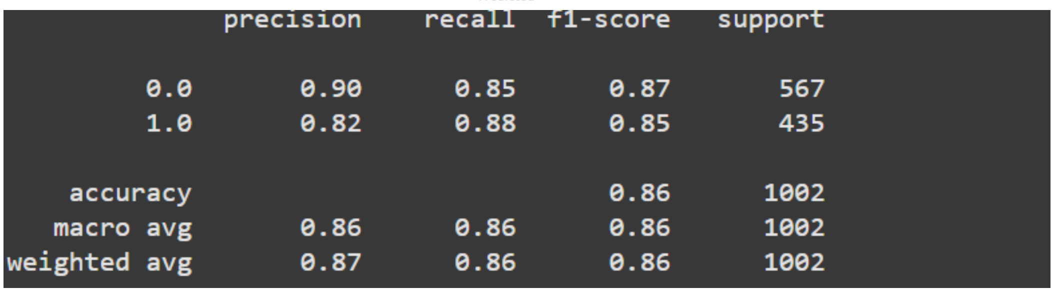
All 4 implementations of the ResNet model showed interesting variation performance both during training and after evaluating on completely unseen data. All models were trained using a mini-batch method , the bayesian models were trained for fewer epochs as their loss curve fell off much more quickly than the vanilla models. Below are the training vs validation loss curves for each model we trained. Along with the models performance on 1002 unseen images from the dataset. Confusion matrix, F1 score report, age scatter plot, r^2 score and MAE have been attached along side the loss curves

ResNet 34 implementation

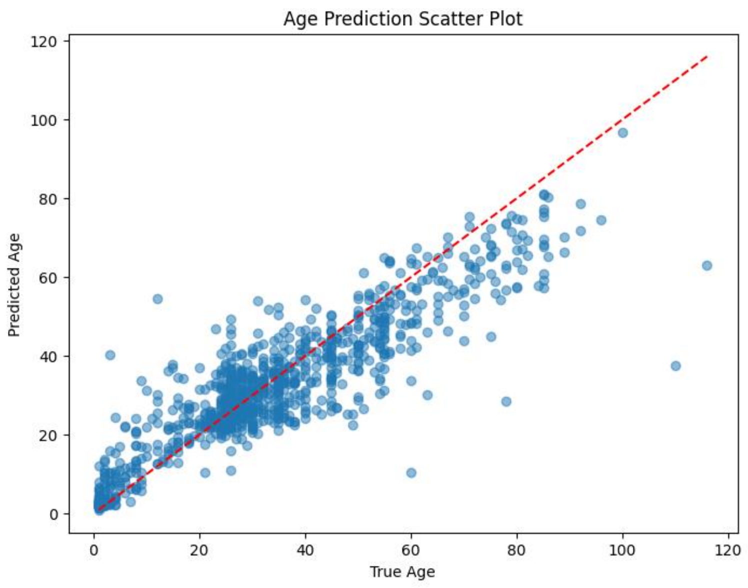


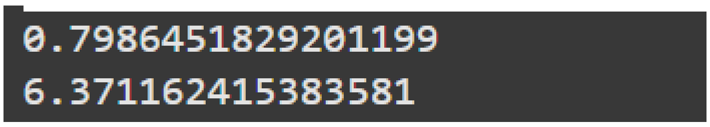
Testing on 1002 unseen images:





```
              precision    recall  f1-score   support

         0.0       0.90      0.85      0.87       567
         1.0       0.82      0.88      0.85       435

    accuracy                           0.86      1002
   macro avg       0.86      0.86      0.86      1002
weighted avg       0.87      0.86      0.86      1002
```

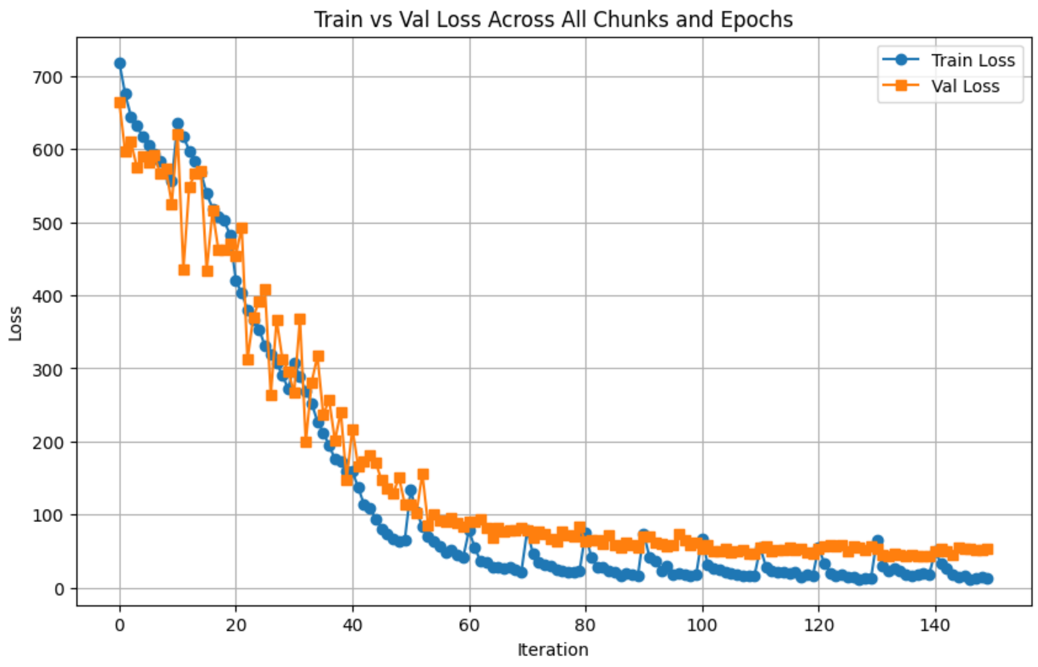


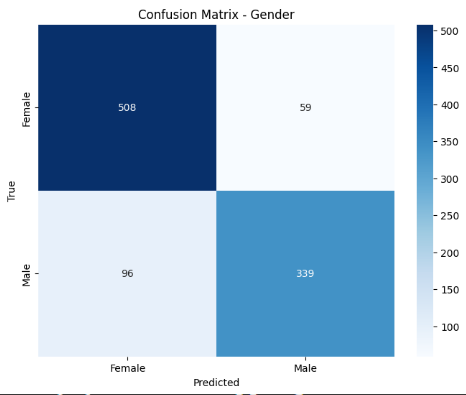


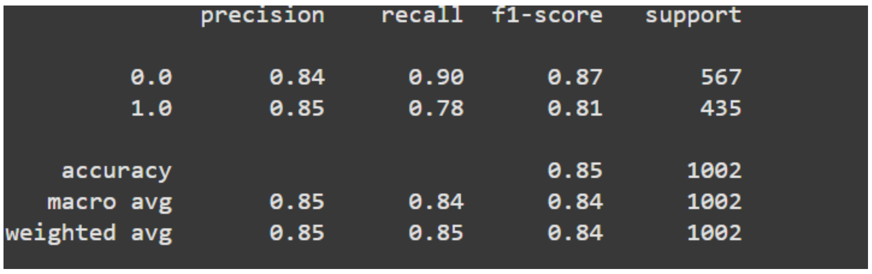
```
0.7986451829201199
6.371162415383581
```

The model performed fairly well given its simplicity.

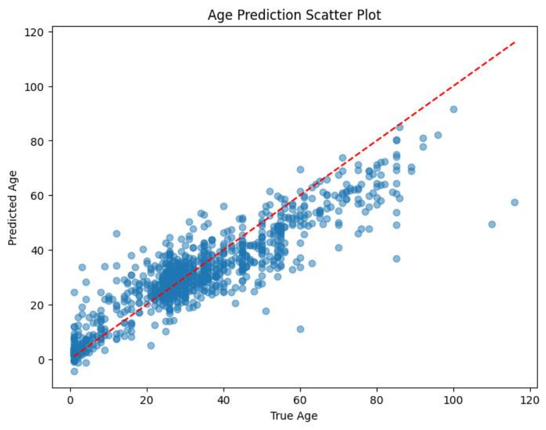
ResNet 152 - Vanilla variant

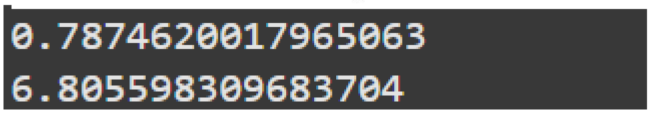






```
              precision    recall  f1-score   support

         0.0       0.84      0.90      0.87       567
         1.0       0.85      0.78      0.81       435

    accuracy                           0.85      1002
   macro avg       0.85      0.84      0.84      1002
weighted avg       0.85      0.85      0.84      1002
```

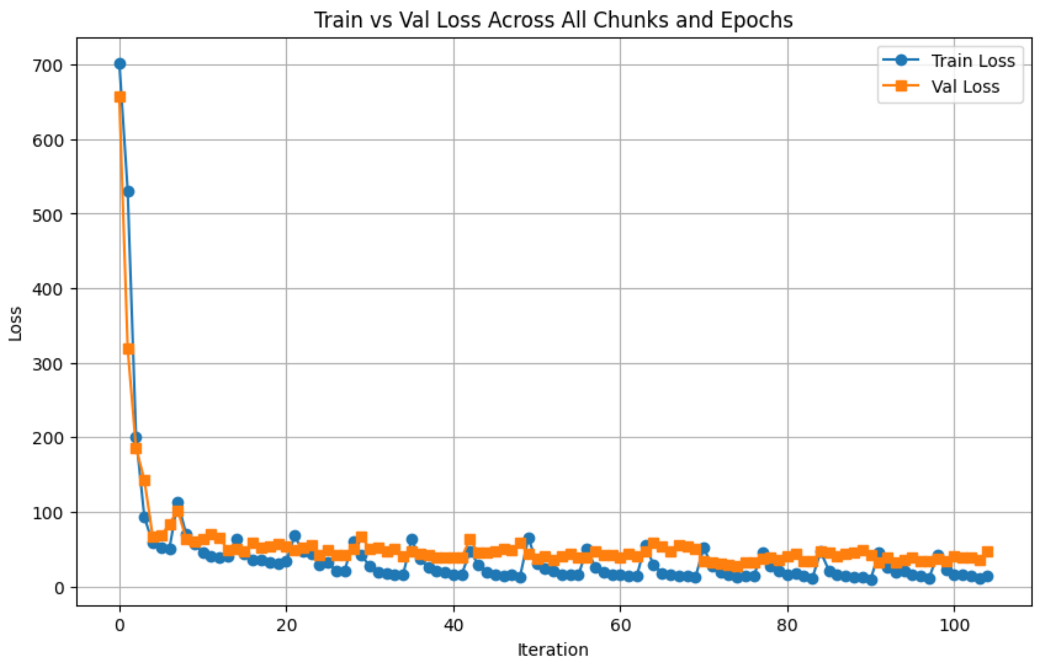


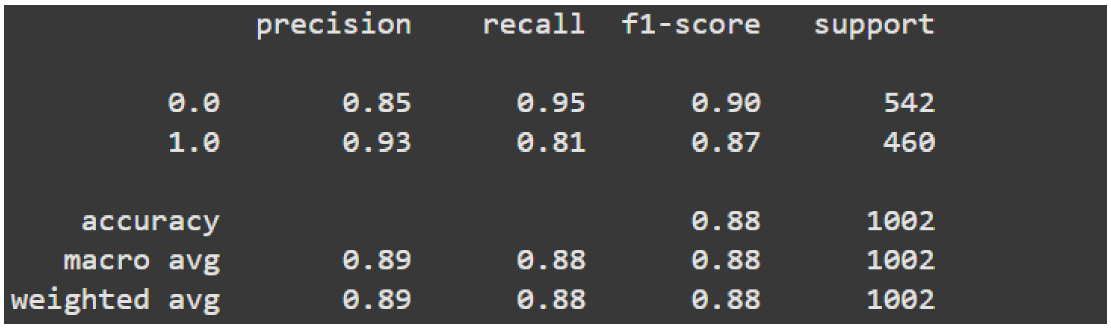


```
0.7874620017965063
6.805598309683704
```

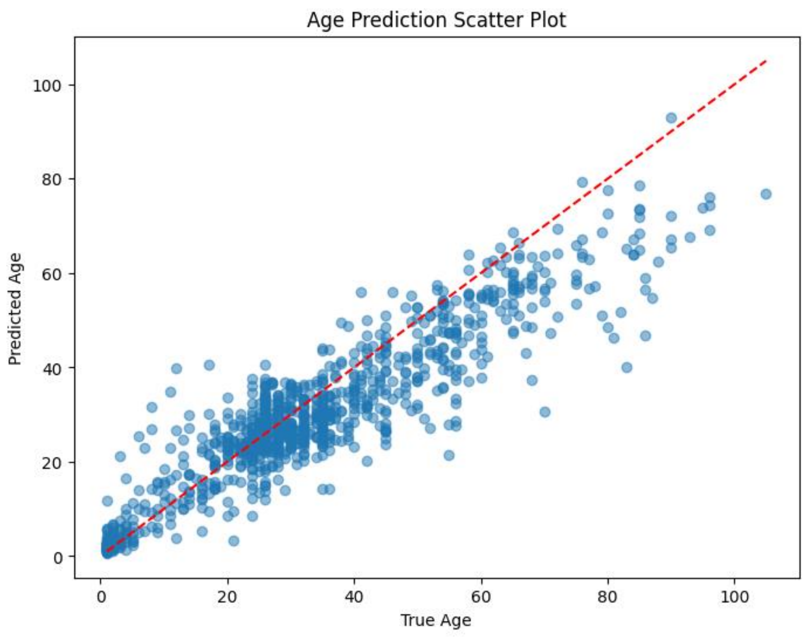
The 152 variant performed below our expectation matching around the same as the 34 model with an almost unnoticeable difference in metrics. After this we moved onto different bayesian implementations to improve the performance.

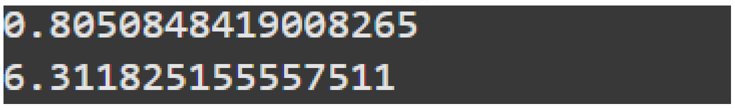
Half Bayesian Approach (Gender head - Bayesian / Age head- Non-Bayesian)



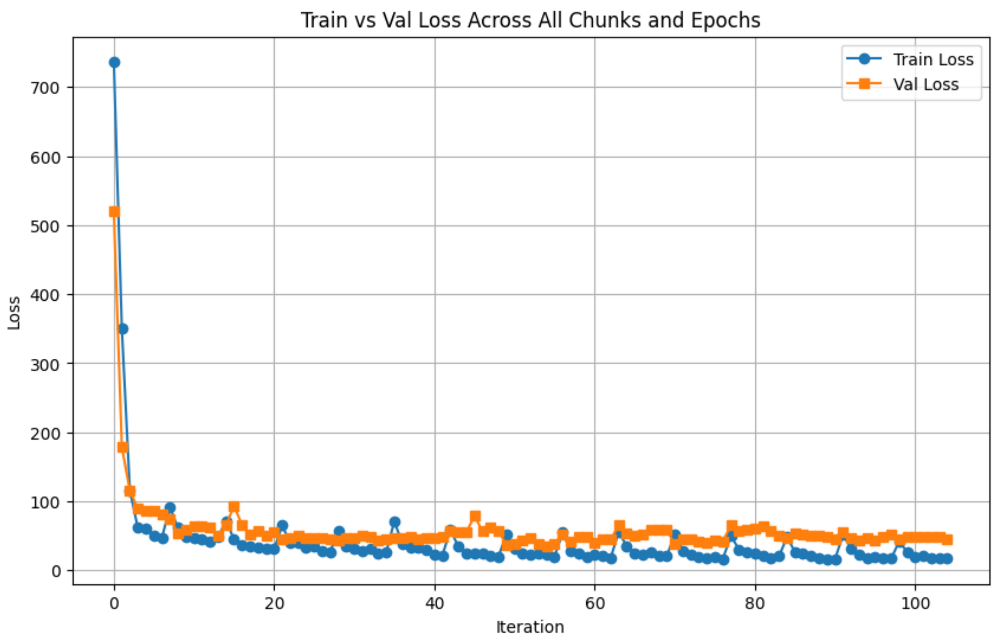


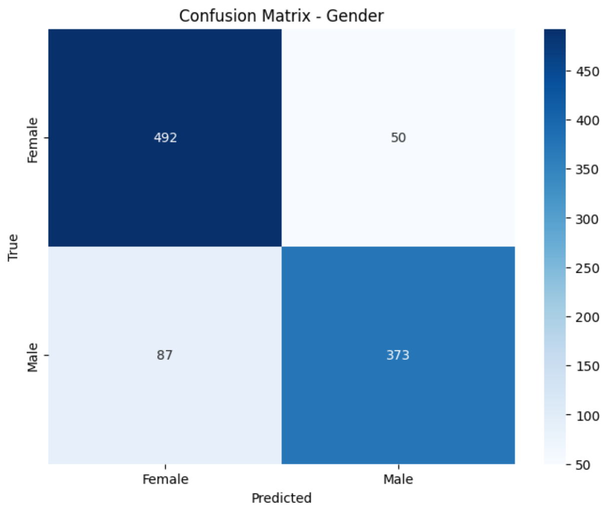
```
              precision    recall  f1-score   support

         0.0       0.85      0.95      0.90       542
         1.0       0.93      0.81      0.87       460

    accuracy                           0.88      1002
   macro avg       0.89      0.88      0.88      1002
weighted avg       0.89      0.88      0.88      1002
```

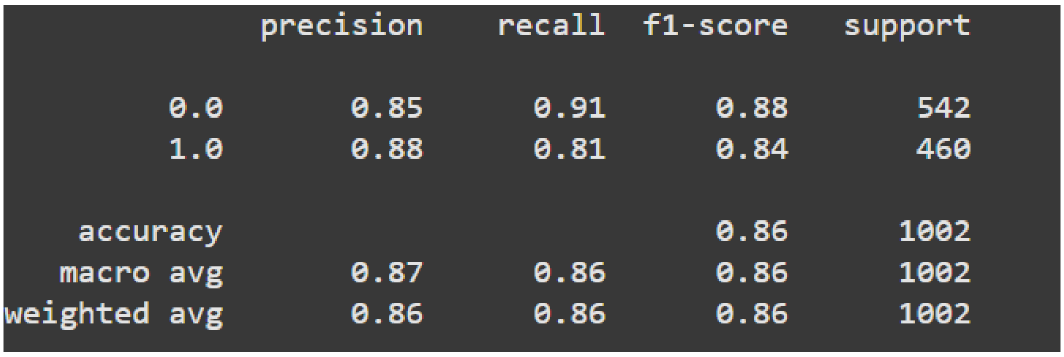




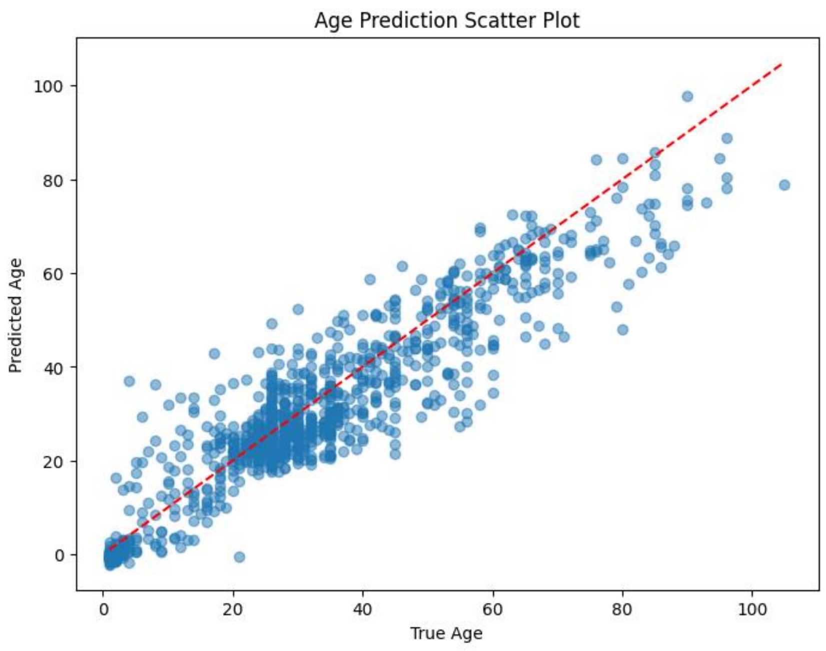
```
0.8050848419008265
6.311825155557511
```

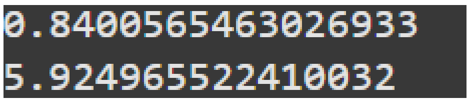






```
              precision    recall  f1-score   support

         0.0       0.85      0.91      0.88       542
         1.0       0.88      0.81      0.84       460

    accuracy                           0.86      1002
   macro avg       0.87      0.86      0.86      1002
weighted avg       0.86      0.86      0.86      1002
```



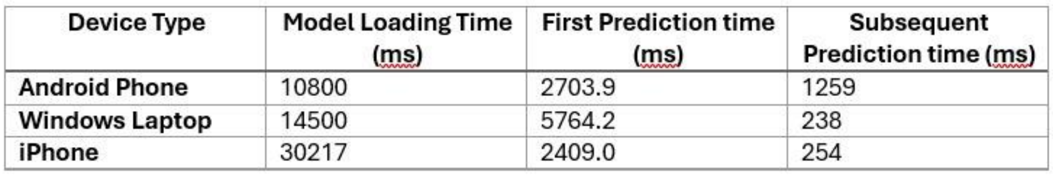


```
0.8400565463026933
5.924965522410032
```

The full bayesian and half bayesian versions both handily outperform the vanilla variants. The bayesian versions have a much faster initial drop off in loss and plateau at a higher loss compared to their vanilla counterparts. In the end our final deployment was of the full bayesian.

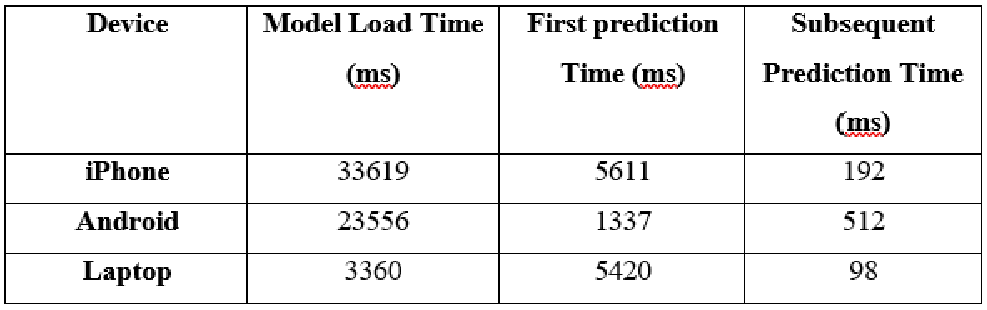
1.6.2 Deployment Results

During the deployment phase, our goal was not only to ensure that the model could perform age and gender prediction reliably in the browser, but also to evaluate how well it performed across different edge devices. We tested both the ResNet-152 and ResNet-34 model variants on three primary platforms: a Windows laptop, an Android phone, and an iPhone. We recorded three critical latency metrics: model load time, first prediction time, and average subsequent prediction times.



Device Type	Model Loading Time (ms)	First Prediction time (ms)	Subsequent Prediction time (ms)
Android Phone	10800	2703.9	1259
Windows Laptop	14500	5764.2	238
iPhone	30217	2409.0	254

Table 1. ResNet-152 Latency on Edge-Based Devices

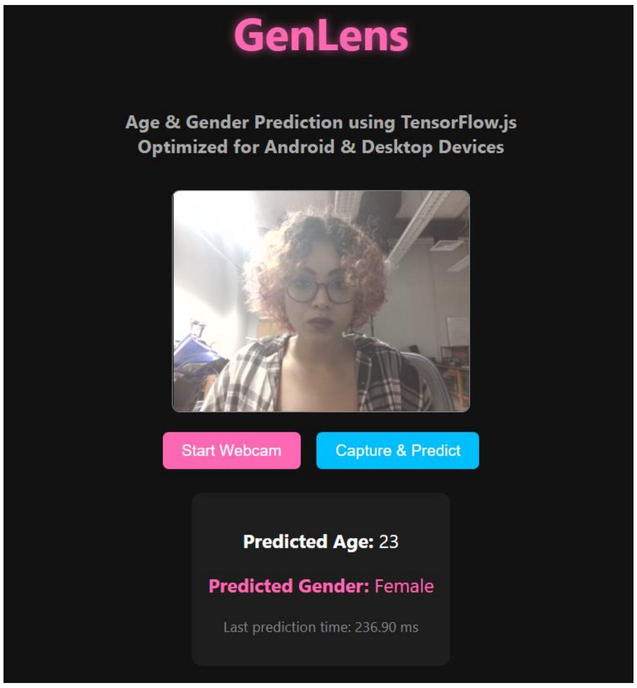


Device	Model Load Time (ms)	First prediction Time (ms)	Subsequent Prediction Time (ms)
iPhone	33619	5611	192
Android	23556	1337	512
Laptop	3360	5420	98

Table 2. ResNet-34 Latency on Edge-Based Devices

In the ResNet-152 deployment, the iPhone device achieved the fastest average first prediction time (~2400ms) but showed higher subsequent latency due to limited mobile processing power. The Windows laptop had a moderate model load time (14.5s) but delivered consistently low inference latency on subsequent predictions (~238ms). The iPhone, while taking over 30s to load the model due to Safari's rendering and memory restrictions, performed surprisingly well with low subsequent latency (~254ms), indicating that once loaded, the model remained stable and responsive.

In comparison, the ResNet-34 variant shown in table 2 had significantly reduced model complexity and file size, which led to much faster model load times across all devices. Notably, the model loaded in just 3.3 seconds on the laptop and under 24 seconds on Android. Despite this, prediction performance still varied: the laptop again demonstrated the lowest latency (~98ms subsequent), while Android's subsequent latency increased (~512ms). The iPhone maintained a consistent ~192ms latency once the model was cached. This validates the use of a lightweight model like ResNet-34 for highly constrained devices, especially in contexts where model startup time is critical.


GenLens

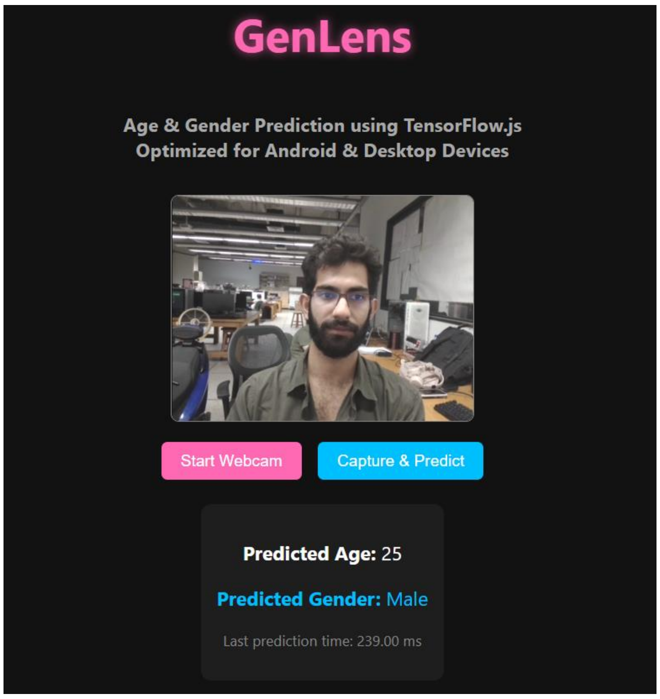
Age & Gender Prediction using TensorFlow.js
Optimized for Android & Desktop Devices

Predicted Age: 23

Predicted Gender: Female

Last prediction time: 236.90 ms

Fig A. ResNet-152 Female Prediction



GenLens

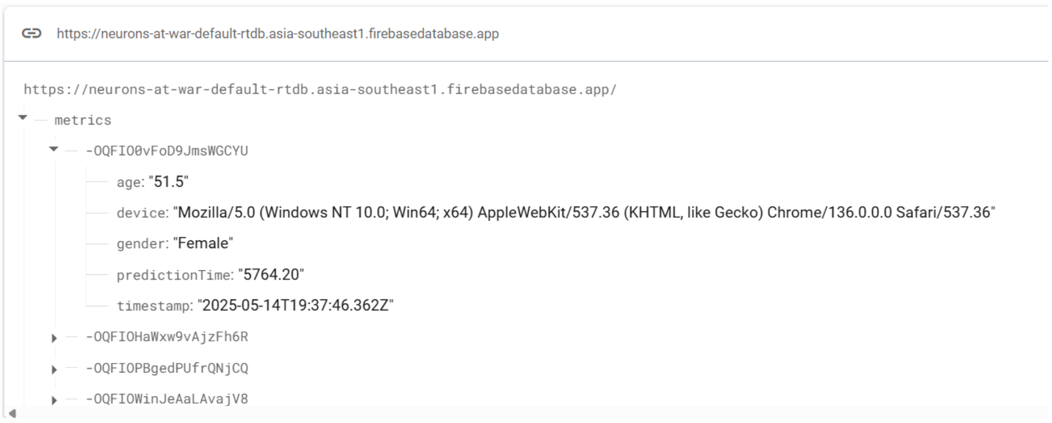
Age & Gender Prediction using TensorFlow.js
Optimized for Android & Desktop Devices

Predicted Age: 25

Predicted Gender: Male

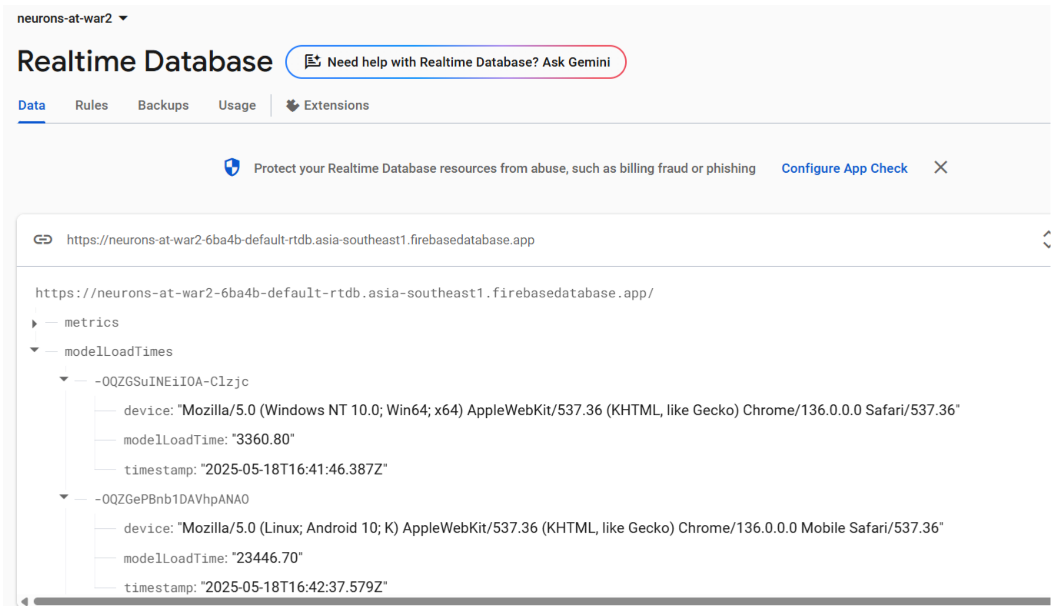
Last prediction time: 239.00 ms

Fig B. ResNet-152 Male Prediction



```
https://neurons-at-war-default-rtdb.asia-southeast1.firebasedatabase.app

https://neurons-at-war-default-rtdb.asia-southeast1.firebasedatabase.app/
▾ — metrics
    ▾ — -OQFIO0vFoD9JmsWGCYU
          — age: "51.5"
          — device: "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0 Safari/537.36"
          — gender: "Female"
          — predictionTime: "5764.20"
          — timestamp: "2025-05-14T19:37:46.362Z"
    ▸ — -OQFIOHaWxw9vAjzFh6R
    ▸ — -OQFIOPBgedPUfrQNjCQ
    ▸ — -OQFIOWinJeAaLAvajV8
```

Fig C. ResNet-152 Firebase Database Console


neurons-at-war2
Realtime Database
Need help with Realtime Database? Ask Gemini
Data
Rules
Backups
Usage
Extensions
Protect your Realtime Database resources from abuse, such as billing fraud or phishing
Configure App Check

```
https://neurons-at-war2-6ba4b-default-rtdb.asia-southeast1.firebasedatabase.app/
▸ metrics
▾ modelLoadTimes
    ▾ -OQZGSuINEiIOA-Clzjc
        device: "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0 Safari/537.36"
        modelLoadTime: "3360.80"
        timestamp: "2025-05-18T16:41:46.387Z"
    ▾ -OQZGePBnb1DAVhpANAO
        device: "Mozilla/5.0 (Linux; Android 10; K) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0 Mobile Safari/537.36"
        modelLoadTime: "23446.70"
        timestamp: "2025-05-18T16:42:37.579Z"
```

Fig D. ResNet-34 Firebase Database Console

We also visualized correct gender predictions (Figures A and B) showing both female and male cases, with real-time webcam inference running entirely in-browser. To support our latency data, we include screenshots of the Firebase Realtime Database logs (Figures C and D) for both ResNet variants. These logs contain timestamps, device information, predicted gender and age, and exact prediction time in milliseconds. This logging mechanism allowed us to benchmark performance across hardware while maintaining user privacy, as all inference and logging occurred locally and anonymously.

These deployment metrics demonstrate that while deeper models like ResNet-152 offer better performance, ResNet-34 provides a much more efficient trade-off between speed and accuracy, especially for mobile-first or offline applications. Our frontend handled both models without crashing, thanks to memory management using TensorFlow.js scopes and careful tensor disposal after each prediction.

1.7 Uncertainty Quantification

We imported the bayesian learning package and added kl divergence to our loss function so our model doesn't just spit out labels, but also to tell us how sure it is. So instead of treating the last layer’s weights as fixed numbers, we treat them like probabilities.During training, we try to push that probability to stay close to a simple prior by adding a KL-divergence term.

1.8 Challenges Faced

1. TensorFlow and ONNX Model Incompatibility

Converting a PyTorch model to TensorFlow.js is a multi-step process that introduced several compatibility issues. The intermediate ONNX format lacked support for certain layers like custom activations or dropout configurations. Additionally, discrepancies in input/output tensor shapes and naming conventions caused functional mismatches. This required manual intervention to adjust layer definitions, outputs, and input formatting, which consumed significant development time and introduced potential for silent model failure if not carefully validated.

2. High Bias in Dataset (Ethnicity Imbalance)

The UTKFace dataset has a significant skew toward white faces, which led the model to favor the most represented class during training. This bias impacted both age prediction accuracy and gender classification reliability, especially for underrepresented ethnic groups. To mitigate this, techniques such as targeted data augmentation, weighted loss functions, and ethnicity-aware validation splits were applied. However, completely neutralizing this bias remains a complex challenge in real-world demographic modeling.

3. Uneven Distribution of Gender and Age Labels

Gender and age labels in UTKFace are highly imbalanced, with a larger presence of young males compared to older females. Age, being a continuous variable, also suffers from labeling noise, leading to frequent prediction errors in underrepresented ranges. The imbalance caused the model to overfit to dominant age clusters like children and young adults, while struggling to generalize on rare cases. Although label smoothing and age normalization were used, edge-case accuracy remained a challenge due to inherent dataset limitations.

4. Large Dataset Size (Training Time: 2–3 Hours)

The dataset contains over 20,000 images with a resolution of 224x224x3, making it computationally intensive to process all at once. Running the full training loop on the entire dataset in memory caused out-of-memory errors, particularly in environments like Google Colab. To address this, the data was split into smaller .pkl chunks that were trained sequentially. This allowed the model to be trained without exceeding memory limits while also enabling progress tracking through intermediate loss metrics and model checkpoints.

5. Latency on Edge Devices

Inference latency in the browser varied significantly across devices and browsers, particularly between desktops and smartphones. The original ResNet-152 model is deep and computationally expensive, making real-time use difficult on lower-end devices. To optimize performance, early layers of the model were frozen, FP16 precision was used where possible, and unnecessary graph nodes were pruned during conversion to TensorFlow.js. These steps helped reduce inference time, but performance still varied depending on hardware and WebGL support.

6. Live Camera Testing with Variable Lighting & Backgrounds
Testing in real-world conditions exposed inconsistencies in webcam input. Lighting conditions had a strong impact—bright lighting would blow out features, while low light would degrade confidence. Additionally, complex backgrounds distracted the model, making it harder to focus on faces. Robust preprocessing, such as adaptive brightness normalization and automatic face cropping, was necessary. Real-time feedback loops were also considered to guide users toward optimal positioning and lighting during use.

7. Cross-Framework Consistency
One of the biggest technical hurdles was ensuring preprocessing consistency between PyTorch (used for training) and TensorFlow.js (used for deployment). PyTorch expects images in (C, H, W) format, while TensorFlow.js requires (H, W, C). Even minor mismatches in resizing, normalization, or data scaling led to severe accuracy drops or unnoticeable silent failures where the model output was meaningless. A detailed log of preprocessing steps and identical normalization parameters were maintained to ensure consistent behavior across frameworks.

8. Deployment & Hosting Issues
Deploying the model via Firebase Hosting posed specific limitations. TensorFlow.js model files, especially model.json and the binary shard files, are

large and can result in long loading times, especially on slower connections. Additionally, Firebase requires all static files to be served from the public/ directory, which had to be properly structured, and the files were sent to build. Optimizations such as loading screen spinner were introduced to manage user experience and avoid delays during model initialization in the browser.

1.9 Optimization and Future Work

1. Model Optimization

Model optimization can be achieved through pruning, quantization, and backbone replacement. Pruning involves removing redundant neurons or channels from ResNet-152 to reduce computational load and model size. Quantization further compresses the model by converting weights to lower precision formats such as INT8 or FP16, significantly decreasing memory usage with minimal impact on accuracy—especially when using TensorFlow Lite or TF.js tooling. Additionally, replacing the ResNet-152 backbone with a lightweight architecture like MobileNetV3, EfficientNet-lite, or SqueezeNet can drastically improve inference speed, making the model more suitable for real-time deployment on edge devices.

2. Preprocessing & Pipeline Improvements

Preprocessing can be enhanced to improve model accuracy and robustness in real-time settings. Integrating face detection and cropping using lightweight detectors like BlazeFace or MediaPipe ensures that only relevant facial regions are analyzed, reducing background noise. Additionally, applying automatic brightness and contrast normalization can standardize input across varying lighting conditions, improving consistency in predictions. Implementing dynamic input resizing allows the system to adapt to device performance or network speed, lowering the input resolution when needed and upscaling it intelligently, which helps maintain responsiveness without significantly affecting prediction quality.

3. Dataset-Level Enhancements

Improving dataset quality is essential for fair and accurate predictions. Balanced subsampling or oversampling ensures that all age groups, genders, and ethnicities are equally represented during training, reducing model bias. Synthetic data generation using techniques like GANs or heavy augmentations can help increase the diversity of underrepresented classes, strengthening the model's generalization. Additionally, applying noise reduction and label cleaning—such as outlier detection to correct or remove mislabeled ages, which are common in UTKFace—can significantly improve training stability and overall prediction reliability.

4. Real-World Robustness & Deployment

To enhance deployment and user experience, the application can be packaged as a Progressive Web App (PWA), allowing users to install it on their devices and access it offline, increasing accessibility and convenience. Incorporating inference time monitoring across different devices and browsers helps identify performance bottlenecks and optimize responsiveness. Additionally, adding prediction confidence intervals—especially when using Bayesian models—can provide users with a sense of how certain the model is about its output, improving transparency and trust in the system's predictions.

5. Extensions & Research-Oriented Work

For future extensions and research-oriented improvements, fairness evaluation should be conducted to assess model performance across gender, ethnicity, and age groups, helping identify and address biases using techniques like reweighting or adversarial debiasing. Expanding the model through multitask learning can enable additional outputs such as emotion recognition, head pose estimation, or face authenticity, increasing its utility. Incorporating uncertainty estimation using Bayesian ResNet methods allows the model to output not only predictions but also confidence intervals, making it more reliable in real-world applications. Lastly, fine-tuning the model on real-world, user-provided webcam inputs can further adapt it to real deployment scenarios, enhancing accuracy and robustness.

2.0 References

K. Kärkkäinen and J. Joo, “FairFace: Face attribute dataset for balanced race, gender, and age,” arXiv preprint arXiv:1908.04913, Aug. 2019.

G. Levi and T. Hassncer, "Age and gender classification using convolutional neural networks," 2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Boston, MA, USA, 2015, pp. 34-42, doi: 10.1109/CVPRW.2015.7301352.

2.1 Google Drive Links (TFJS files & Frontend)

Resnet-152 final implementation:
https://drive.google.com/drive/folders/1JAdCiX32SFT330F-t6-aeBp6xffH99MG?usp=drive_link

Resnet-34 implementation:
https://drive.google.com/drive/folders/1kiYq3BxNxG0VKPPeto1S3Hn7devI6XpP?usp=drive_link

React implementation on VScode integrated with firebase:

https://drive.google.com/file/d/1jAYZlLWaZidSiN8J3waqXjci4oyJaKZu/view?usp=sharing