SURT AI

ML Engineering Internship | Progress Report Two

Tanya Budhiraja | January 5th 2026

Goals

Weeks 5–8: Computer Vision Focus
Explore CNNs and image embeddings. Implement a simple image classifier in Rust that recognizes face presence using pretrained weights. Begin Coursera CNN and "Improving Deep Neural Networks." Deliverable: model accuracy comparison + code walkthrough.

Data Description

Face images were sourced from the CelebA dataset:
https://www.kaggle.com/datasets/jessicali9530/celeba-dataset

All images from this dataset were placed into a single face/ directory.

Non-face images were sourced from the Natural Images dataset:
https://www.kaggle.com/datasets/prasunroy/natural-images

This dataset contains the following categories:

- airplane (727 images)
- car (968 images)
- cat (885 images)
- dog (702 images)
- flower (843 images)
- fruit (1000 images)
- motorbike (788 images)
- person (986 images)

Since the goal is to exclude human faces from the non-face class, the **person** category was manually removed**.** All of the remaining categories were merged into a single non_face/ directory.

After collecting the raw images, the following preprocessing steps were applied using prepare_dataset.py:

1. File name cleaning
2. Label assignment

    - Assigned labels:
        - 1 = face
        - 0 = non face
3. Image validation
4. Image resizing
    - Resized all images to 224 × 224, which is the expected input size for ResNet-18.
5. Class balancing
    - Addressed dataset imbalance (CelebA contains 200k+ face images, while Natural Images contains ~5k non face images) by subsampling to ensure balanced classes.
6. Train / validation / test split
    - 70% training
    - 15% validation
    - 15% test
7. CSV manifest creation
8. Convert all files to JPEG

Dataset Structure

The dataset is organized into two main components: image data and splits with labels.

All images are stored in two class based folders:

- faces contains all face images (label = 1)
- no_faces contains all non face images (label = 0)

Train, validation, and test splits are defined using CSV files stored in a separate splits/ directory. Each CSV file contains three columns (filename, label, class) and specifies the image path and corresponding class label for each sample.

This structure decouples the physical image storage from the data splits, enabling flexible, reproducible training and evaluation.

What is a CNN

Convolutional Neural Networks are a class of deep learning models designed for image data, as they learn from groups of nearby pixels rather than treating each pixel independently. CNNs use filters to detect patterns such as edges, textures, and shapes, then reduce the image size while preserving key features. As depth increases, the network learns higher level representations, allowing it to capture complex visual concepts such as facial structure. These learned features are typically passed to fully

connected layers for final classification. CNNs are very good for face detection tasks because they are translation invariant, robust to background noise, and capable of learning hierarchical features directly from raw images without feature engineering. Using pretrained CNN architectures, such as ResNet-18, further improves performance by using image embeddings learned from large scale datasets. CNNs are an effective choice for distinguishing faces from non faces in this project.

ResNet

Residual Networks (ResNets) are a family of CNNs designed to enable very deep architectures by introducing residual (skip) connections. Skip connections allow layers to learn residual mappings instead of full transformations which significantly reduces vanishing gradient issues when training. ResNet-18 and ResNet-34 are commonly used lightweight variants consisting of 18 and 34 layers. They are a strong balance between representational power and computational efficiency. Deeper ResNet models, like ResNet-50 or ResNet-101, include bottleneck layers that further increase depth and expressiveness but at the cost of higher memory and compute requirements. In this project, ResNet-18 and ResNet-34 were used due to their strong performance on image classification tasks, fast inference, and availability of pretrained weights, making them practical for experimentation and deployment. ResNet-50 was also evaluated to assess whether increased depth and better feature representations would improve classification performance despite its higher computational cost. These architectures are effective for image data because they learn hierarchical visual features while residual connections preserve important information across layers. Additionally, ResNet models are well supported in Rust through the tch (PyTorch) library. This enables efficient model loading, inference, and integration into a performant, memory safe Rust codebase.

Procedure

The project began by loading pretrained ResNet weights and integrating them into a Rust codebase using the tch (PyTorch) library. These pretrained models act as strong image feature extractors, allowing the network to reuse visual representations learned from previous training on big datasets. Initial training was attempted locally on a Mac machine, but this quickly ran into memory constraints. Despite having 18 GB of RAM, the system could not handle batch sizes larger than 1 across multiple epochs. Training under these conditions resulted in an accuracy of roughly 50%, which is equivalent to random guessing.

After trial and error with memory usage and doing research on common deep learning workflows, the training pipeline was moved to Google Colab. Colab provided access to free GPU resources and enough memory to train the model efficiently in Python. The network was trained in Python, and once training was complete, only the weights of the final fully connected (FC) layer were saved. These trained FC weights were then transferred back into the Rust implementation and loaded on top of the pretrained ResNet backbone for inference. Only the FC layer was saved because the rest of the network was already pretrained and did not need to be retrained. By training and saving only the FC layer, the project could take advantage of the strong, pretrained ResNet features while keeping the Rust implementation lightweight and memory efficient.

To make the model usable I made a simple command line interface (CLI) tool in Rust. This tool loads the pretrained backbone and the trained FC layer, processes an input image, and outputs a face vs. non face prediction, enabling lightweight and efficient inference directly from the command line.

During the prediction, the CLI passes the input image through the model to get a score for each class. These scores are converted into probabilities using a softmax function. The class with the highest score is chosen as the prediction, and the confidence is calculated as the probability of that predicted class, reported as a percentage. The CLI returns this confidence score along with the prediction.

ResNet18

The ResNet-18 model was trained in a Google Colab notebook using a GPU due to memory and compute constraints on my local hardware. The training code is contained in the Python notebook
training_scripts_for_colab/face_classifier_training_resnet18.ipynb, which trains a pretrained ResNet-18 model and exports the trained FC layer in a TorchScript-compatible format for use in Rust.

After training, the learned weights of the final FC layer were downloaded and added to the project under: models/resnet18/fc_layer_resnet18_cpu.pt. This layer was then loaded on top of the pretrained ResNet-18 backbone in Rust to make predictions. A CLI was developed to allow users to classify images as face or non face directly from the terminal using the trained model. The CLI is located in src/bin/infer_resnet18.rs and can be called with:

bash scripts/run_resnet18.sh clean_data/data/faces/face_000000.jpg {replace with path of your image}.

When run, the CLI processes the input image and outputs both the predicted class and a confidence score, for example:

FACE DETECTED

Confidence: 74.80%

This setup enables efficient, lightweight inference in Rust while leveraging a model trained using GPU resources in Python.

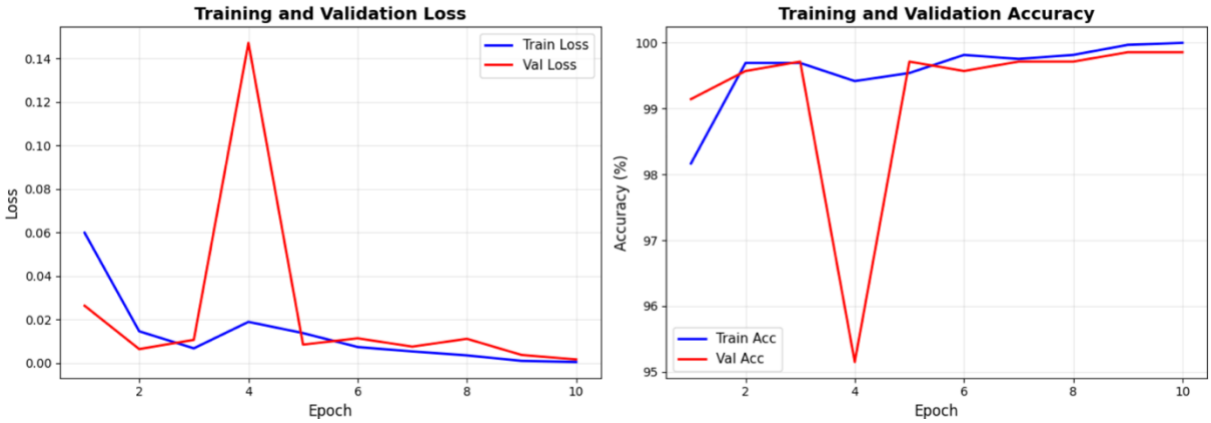

Fig 1. Training curve for Resnet-18, best accuracy: 99.87%.

The training and validation curves for ResNet-18, as seen in Figure 1, demonstrate fast convergence and strong overall performance. Training loss decreases across epochs, while validation loss remains low and closely follows training loss. This indicates effective learning without overfitting. There is a brief spike in validation loss around epoch 4, which is likely due to batch level variance or class sampling noise rather than instability in the actual model, as the loss quickly recovers in subsequent epochs.

Training and validation accuracy increase within the first few epochs and stabilize near 99 -100% as seen in Figure 1. This shows that the model can separate face and non face images with very high reliability. At epoch 4 there is a dip in validation accuracy which mirrors the spike in validation loss, this is expected. Importantly, both accuracy and loss curves realign after epoch 4 which shows that the model generalizes well to unseen data.

The close alignment between training and validation metrics indicates that ResNet-18 is able to learn visual features for this task without significant overfitting. The results confirm that ResNet-18 provides an effective balance between performance and computational efficiency for face detection in this project.

ResNet34

The same training and deployment process used for ResNet-18 was repeated for ResNet-34. The model was trained in a Google Colab notebook using GPU acceleration, and the training code is contained in training_scripts_for_colab/face_classifier_training_resnet34.ipynb.

As with ResNet-18, a pretrained ResNet-34 backbone was used, and only the FC layer was trained for face vs. non face classification. After training, the FC layer weights were exported in a TorchScript-compatible format and downloaded to the project under models/resnet34/fc_layer_resnet34_cpu.pt.

These trained FC weights were then loaded on top of the pretrained ResNet-34 backbone in the Rust implementation for inference. A CLI was implemented to classify user provided images directly from the terminal. The ResNet-34 CLI is located in

src/bin/infer_resnet34.rs

and can be executed with: bash scripts/run_resnet34.sh clean_data/data/faces/face_000000.jpg

When run, the CLI processes the input image and outputs both the predicted class and the associated confidence score, enabling efficient and lightweight inference in Rust using the ResNet-34 architecture.

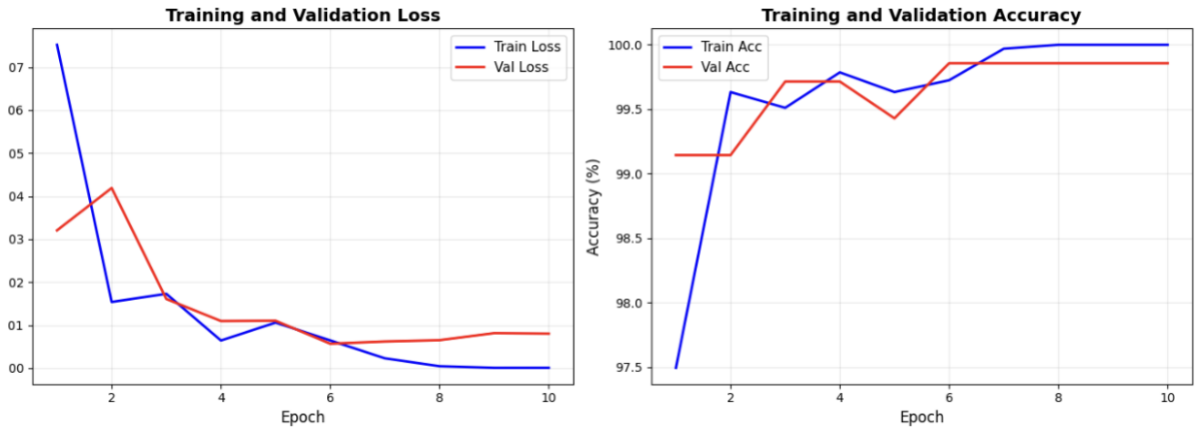

Fig 2. Training curve for Resnet-34, best accuracy: 99.86%.

The training and validation curves for ResNet-34, as seen in figure 2, show fast and stable convergence. Training loss decreases sharply in the first few epochs and continues to decline toward zero, while validation loss follows a similar trend and

remains low. This indicates effective learning without significant overfitting. Training and validation accuracy increase and stabilize around 99–100%, demonstrating strong classification performance. Minor fluctuations in validation metrics during early epochs are expected due to stochastic batch sampling and quickly resolve, suggesting that the model generalizes well to unseen data. Overall, ResNet-34 achieves high accuracy with stable training behavior, confirming its suitability for the face vs. non-face classification task.

ResNet50

The same training and deployment process used for ResNet-18 and ResNet-34 were repeated for ResNet-50. The model was trained in a Google Colab notebook using GPU acceleration, and the training code is contained in training_scripts_for_colab/face_classifier_training_resnet50.ipynb.

As with ResNet-18, a pretrained ResNet-50 backbone was used, and only the FC layer was trained for face vs. non face classification. After training, the FC layer weights were exported in a TorchScript-compatible format and downloaded to the project under models/resnet34/fc_layer_resnet50_cpu.pt.

These trained FC weights were then loaded on top of the pretrained ResNet-50 backbone in the Rust implementation for inference. A CLI was implemented to classify user provided images directly from the terminal. The ResNet-50 CLI is located in

src/bin/infer_resnet50.rs

and can be executed with: bash scripts/run_resnet50.sh clean_data/data/faces/face_000000.jpg

When run, the CLI processes the input image and outputs both the predicted class and the associated confidence score, enabling efficient and lightweight inference in Rust using the ResNet-50 architecture.

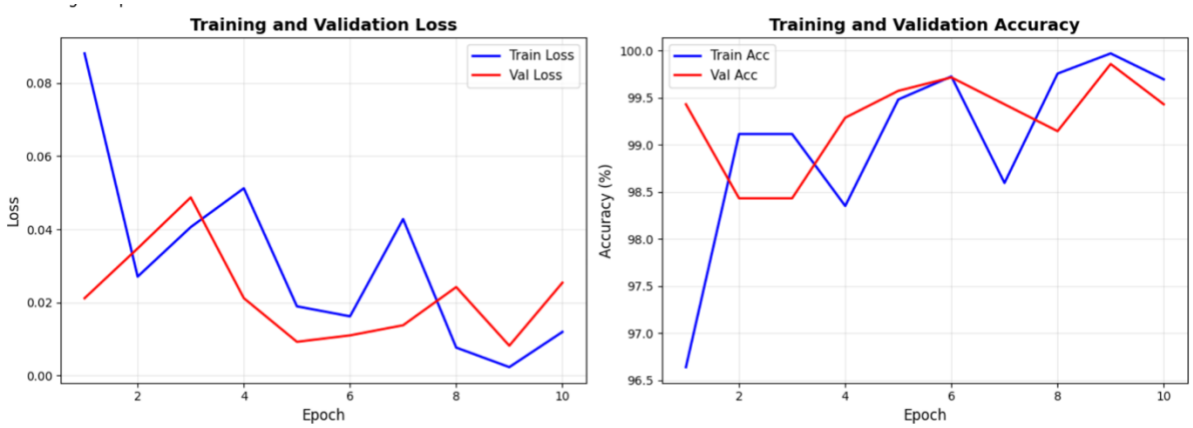

Fig 3. Training curve for Resnet-50, best accuracy: 99.86%.

The training and validation curves for ResNet-50, as seen in figure 3, show strong overall performance but with more variability as when compared to the shallower models. Training loss's overall trend decreased across epochs, while validation loss fluctuated, showing less stable convergence. Training and validation accuracy remained high, reaching approximately 99 - 100%. However, there were large oscillations across epochs. These fluctuations suggest that the increased model depth may introduce sensitivity to batch - level variation and potential overfitting on this relatively simple binary classification task. Overall, while ResNet-50 achieves high accuracy, its added complexity does not provide a clear advantage over ResNet-18 or ResNet-34 and comes at a higher computational cost.

Discussion

ResNet-18, ResNet-34, and ResNet-50 all achieved very similar performance, with peak validation accuracy near 99 - 100% and no meaningful separation between models. This suggests that face vs. non face detection is likely too simple of a task to fully differentiate model capacity, as even the shallowest architecture was sufficient to learn the required features. Confidence scores produced by the CLI were also comparable across all models, typically falling between 65–85%, further indicating that increased depth did not significantly improve prediction certainty. While deeper models such as ResNet-50 offer greater representational power, this did not translate into measurable performance gains and introduced more variability during training. A more complex task such as facial recognition or identity classification would likely better show the differences between architectures by requiring more feature learning.

If this project were repeated, improvements could include using a more challenging dataset, having more edge cases in the dataset, and performing deeper error analysis on misclassified samples. Future work could extend this pipeline to more complex vision

tasks or integrate it into a larger system. Overall, these results suggest that ResNet-18 offers the best balance between performance, stability, and computational efficiency for this project.

Next steps

Weeks 9–12: Structuring ML Projects

Study error analysis and dataset strategy via Coursera’s Structuring ML Projects. Begin planning for the liveness pipeline. Define pipeline components (capture module, inference core, feedback loop). Deliverable: design doc for the full pipeline