Ottoman Character Recognition via Deep Transfer Learning

BEDAI TEKNOLOJİ A.Ş.
Boğaziçi Teknopark – Kandilli Kampüsü
Kandilli Mah. Rasathane Cad. No:104/13-2, Üsküdar
İstanbul, Türkiye
info@bedai.com.tr

14 January 2026

Abstract

This project develops a reproducible deep-learning pipeline for Ottoman Turkish character recognition from image patches. The task is to classify isolated Ottoman characters under realistic degradation (noise, blur, small geometric distortions, and contrast variation), treating character classification as a practical subproblem of end-to-end Ottoman OCR. The work covers **Dataset Creation** (curation and splitting of a public Ottoman character dataset), **Dataset Annotation** (label-preserving augmentation to emulate manuscript/print artifacts), **Model Training or Fine-Tuning** (transfer learning with ImageNet-pretrained CNN foundation), and **Hyper-Parameter Tuning** (small sweeps and controlled ablations over augmentation, architecture, and scheduler choices). Training and experiment metadata are tracked with Weights & Biases (W&B) [14]. We evaluate using accuracy and class-balanced macro-F1, and we analyze errors via confusion matrices, per-class metrics, and qualitative examples. On the experimental split used in this repository snapshot (train/val/test = 2,905/513/465; 3,883 total) spanning 43 observed classes, the best configuration reaches test accuracy 0.8882 and macro-F1 0.8587. Finally, the accompanying notebook includes a simple demo that predicts Ottoman character labels and maps them to letter names and modern Turkish equivalents, providing an interpretable baseline toward full Ottoman OCR.

1 Introduction

Ottoman Turkish documents constitute a large historical record, yet automated recognition remains challenging due to limited labeled data, heavy domain shift from modern printed fonts, and severe degradation in scans (blur, ink bleed-through, low contrast, compression artifacts). From a computer-vision perspective, Ottoman OCR is difficult not only because of noise, but also because many graphemes share near-identical skeletons and differ primarily by small diacritics or stroke endpoints. These properties motivate a careful choice of evaluation metrics (class-balanced macro-F1, per-class analysis) and a robust preprocessing pipeline.

This report focuses on *character-level recognition* from pre-cropped image patches as a controlled, measurable subtask. While end-to-end OCR requires detection/segmentation and sequence modeling, high-quality character classification is a necessary building block for both classic OCR pipelines (segment-then-classify) and modern sequence models (as auxiliary supervision and error analysis tooling). Accordingly, our goals are:

- build a reproducible training/evaluation pipeline for Ottoman character classification,
- conduct systematic experimentation (controlled ablations and comparisons) to understand what improves performance and why,
- provide a demo pipeline that accepts images and returns interpretable predictions in Ottoman label space.

Table 1: Dataset splits used in experiments (as logged when building DataLoaders).

Split	# images	# classes
Train	2,905	43
Validation	513	43
Test	465	43

2 Methods and Results

2.1 Dataset Creation

Data source and classes. We use the Kaggle dataset `alpbintuuzun/ottoman-turkish-characters` [5]. The dataset contains 32×32 black/white character crops extracted from multiple writing/print styles (Talik, Rika, Matbu, plus a small mixed subset), and labels are encoded in filenames (the final two digits before `.png` correspond to the tag). The dataset description reports 3,894 character images overall.

In the accompanying notebook, images are indexed by filename across the dataset root and a precomputed stratified split is loaded when available (in the original Colab/Drive setup, this is `splits_fixed.json` under `PATHS["data"]`). To make the pipeline resilient to different directory layouts, missing stored paths are repaired by remapping via this filename index, and any still-unresolvable entries are filtered before training. In this repository snapshot, the split JSON itself is not included; however, the split sizes used throughout the experiments are logged when constructing the DataLoaders and match the exported evaluation artifacts: train/val/test = 2,905/513/465, totaling 3,883 images.

Curation and dataset preparation. Although we rely on public dataset sources, our "dataset creation" work is best understood as *curation for experimental validity and reproducibility*. Concretely, we (i) unify the dataset into a single, consistently indexed pool (by filename) to tolerate different folder layouts, (ii) enforce a well-defined label space and mapping from numeric tags to human-readable class names, and (iii) fix the evaluation protocol by using a stratified train/validation/test split that remains constant across all ablations and comparisons. This is particularly important for a small, imbalanced dataset: without a fixed split and stable preprocessing, differences across runs can be dominated by sampling noise rather than methodological choices.

Label space. The dataset provides 44 nominal tags (01–44) corresponding to Ottoman letters, digits, and the *lamelif* ligature. However, in our cleaned split we observe 43 classes: tag `41` is absent (no samples). Therefore we set the number of classes dynamically from the saved split metadata (`label2idx`/`idx2label`) rather than hard-coding K. This prevents silent shape mismatches and makes the pipeline robust to dataset subsets.

Label mapping. For interpretability, we map each numeric class tag to an Ottoman letter name and a modern Turkish equivalent in the format `(tag: ottoman_name: modern counterpart)`. This mapping is used consistently in the demo and in per-class metrics plots/tables.

Splits. We use a stratified train/validation/test split (created and saved by the notebook in the original Colab/Drive environment). Stratification is performed on the class label so that each split maintains similar class proportions. The split sizes used in this report (as logged when building the DataLoaders) are:

Preprocessing. Although the raw dataset images are 32×32 grayscale, our models use ImageNet-pretrained CNN backbones [2] that expect 3-channel inputs and benefit from larger spatial resolution. Therefore, each image is loaded with OpenCV [8], converted to RGB (by channel replication where needed), and resized to a larger working resolution (128×128). We normalize inputs using ImageNet mean/std so that fine-tuning starts from a compatible feature distribution.

Robust data loading and path repair. We observed that the notebook we used to implement this project often broke when file paths differed across machines (e.g., Google Drive vs local filesystem). In case a third party would like to replicate this project and to avoid training crashes, our dataset loader implements:

- filename-based indexing of all images under the dataset root (building a `PNG_INDEX` mapping filename → full path),
- dynamic path remapping when a stored path does not exist (try to resolve by filename),
- filtering of unresolved samples before training begins, so DataLoaders never hit missing files mid-epoch.

In practice, this engineering eliminates a common source of brittle failures and makes repeated ablations much more reliable.

2.2 Dataset Annotation (Augmentation)

Because historical documents exhibit noise and artifacts, we apply label-preserving transformations to simulate realistic conditions:

- **geometric**: small translations, scaling, and rotations (affine perturbations),
- **photometric**: blur (Gaussian) and additive noise (Gaussian noise),
- **contrast/quality**: brightness/contrast changes and optional JPEG compression artifacts,
- **polarity**: optional inversion (useful when foreground/background polarity varies across scans).

Augmentations are applied online per sample using Albumentations [1]. The transform definitions are written to remain compatible across Albumentations versions, and inputs are normalized using ImageNet mean/std to match pretrained backbones.

Our methodology of "dataset annotation" as a major contribution in this project. The downloaded Kaggle dataset is pre-labeled at the *class* level (each crop has a tag). For character classification, that class tag is the relevant annotation. Our contribution under the dataset-annotation component is therefore to *augment the labeled dataset with additional, label-consistent training instances*: each transformation produces a new image whose label is inherited from the source crop. This expands the effective training distribution and targets the dominant real-world failure modes in historical scans (blur, low contrast, noise, and small geometric distortions).

This differs from object-detection or segmentation settings (where annotation often means bounding boxes or masks), but it matches the structure of the present task: inputs are already isolated character crops, and the supervision signal is the class label. In that context, generating additional labeled instances through controlled, label-preserving transforms is a direct form of dataset annotation for robustness.

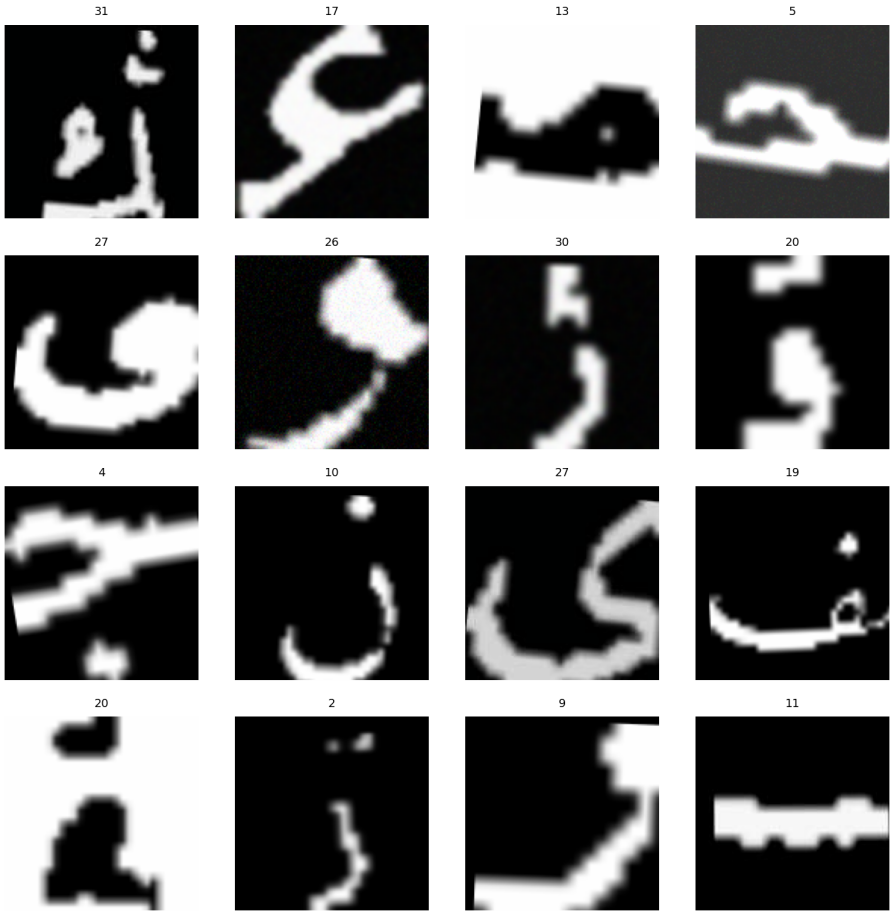

Figure 1: Augmented sample grid used to visually verify preprocessing and augmentation.

Evidence via controlled ablation. To avoid treating augmentation as an "opaque tricks," we evaluate it systematically via Task 1 (augmentation ablation), comparing none vs. geometric-only vs. photometric-only vs. combined settings under an otherwise fixed training protocol. Results are reported in Figure 2 and `outputs/augmentation\_ablation.json`. This ablation both justifies the annotation/augmentation work as a substantial component and helps identify transforms that help (or harm) generalization on this low-resolution glyph recognition task.

2.3 Model Training or Fine-Tuning

Architecture. We evaluate several ImageNet-pretrained backbones: ResNet-18/34 [3], EfficientNet-B0 [12], and MobileNetV3-Small [4]. For each architecture, we replace the final classification head to output $K = 43$ logits (based on the observed label set) and fine-tune the network on Ottoman character crops. Parameter counts (from the trained checkpoints) span from 1.56M (MobileNetV3-Small) to 21.31M (ResNet-34), enabling a direct accuracy/efficiency trade-off study.

Training details. All experiments share a consistent training protocol so comparisons are meaningful:

- optimizer: AdamW with weight decay [7],
- loss: cross-entropy with label smoothing [11] (to reduce overconfidence on small/imbalanced classes),

- training length: 8 epochs per experiment (unless explicitly stated otherwise),
- selection: save the best checkpoint by *validation macro-F1* (primary metric),
- default schedule: cosine annealing [6]; additionally evaluated OneCycleLR [10] and ReduceLROnPlateau.

Fine-tuning protocol (transfer learning). We initialize each backbone with ImageNet weights and replace the final classifier layer to match the observed label set ($K = 43$). We then fine-tune end-to-end using ImageNet-compatible preprocessing (RGB conversion and ImageNet normalization). Since the original crops are only 32×32, we resize inputs to 128×128 to better match the spatial scale expected by the pretrained feature extractors and to preserve small diacritic details during augmentation.

Checkpointing, evaluation, and efficiency. To make comparisons consistent across experiments, we select the best checkpoint by *validation macro-F1* and report test-set performance from that checkpoint on the fixed held-out split. Alongside accuracy and macro-F1, we export per-class metrics and confusion statistics for diagnostic analysis (Tasks 7 and the confusion analysis in Section 2.7). To contextualize model capacity and deployment trade-offs, we report parameter counts and a simple inference-time estimate in milliseconds per image, measured under a fixed evaluation loop.

Implementation and compute environment. All models are trained in PyTorch [9]. We use notebook-safe DataLoader defaults to avoid multiprocessing shutdown issues in Colab/Jupyter environments (typically `num_workers=0`, `pin_memory=False`), which keeps repeated ablations stable. Training runs on GPU when available (`device=cuda`); our main runs were executed on an NVIDIA L4 GPU (24 GB VRAM) in Google Colab, with artifacts saved to Google Drive.

Experiment tracking (W&B). All runs are tracked in Weights & Biases for side-by-side comparisons. We log train/val loss, accuracy, and macro-F1 (our primary, class-balanced metric) per epoch, along with the learning-rate schedule when relevant (e.g., cosine vs. OneCycle).

In addition to learning curves, we record final test-set performance (`test_loss`, `test_acc`, `test_macro_f1`), the selected best-validation checkpoint, and a simple inference-time estimate (`inference_ms_per_image`) to support accuracy–efficiency comparisons.

For transparency, we provide the complete W&B report for interactive inspection and a curated artifact directory containing the full set of outputs produced in the original training environment.

W&B interactive report:
`https://api.wandb.ai/links/nurcunal-bogaziciuniversitesi/w5zbphrx`

Project outputs (Google Drive artifact directory):
`https://drive.google.com/drive/folders/17OemY0j27mqwH7nLm_fgFIevS8jjFU9l?usp=share_link`

The Drive artifact directory archives all outputs needed to reproduce results in the original Colab/Drive environment, including model checkpoints, split JSON files, demo images, logs, and exported report figures. This repository snapshot includes a subset of those artifacts under `outputs/` and `demo_images/`.

2.4 Hyper-Parameter Tuning

We perform a small grid search over learning rate and batch size to select stable training defaults (`outputs/sweep_results.json`). Specifically, we test lr $\in \{10^{-3}, 3 \times 10^{-4}\}$ and batch_size $\in \{32, 64\}$ with short runs. The best validation macro-F1 is obtained at lr $= 10^{-3}$, batch size $= 64$ (val macro-F1 0.8829), which we use as a recommended stable configuration when GPU memory permits (our NVIDIA L4 GPU was equipped with 24 GB VRAM). For reference, the best validation accuracy is achieved at lr $= 3 \times 10^{-4}$, batch size $= 64$ (val acc 0.8967), but with slightly lower macro-F1 (0.8687).

Our tuning strategy proceeds in two stages. First, we establish stable optimizer settings (learning rate and batch size) so that subsequent comparisons are not confounded by an unstable baseline. Second, keeping this baseline fixed, we perform targeted ablations and comparisons over the major design axes that plausibly affect generalization in this setting: augmentation policy, backbone choice, scheduler choice, and strategies for handling class imbalance. Throughout, we select checkpoints by validation macro-F1 and report test-set results from the selected checkpoint to avoid overfitting comparisons to test performance.

Beyond the LR/BS sweep, we run controlled ablations and comparisons:

- **Task 1**: augmentation ablation (none vs geometric vs photometric vs all),
- **Task 2**: architecture comparison (4 backbones),
- **Task 3**: class imbalance mitigation (standard vs class-weighted loss),
- **Task 4**: embedding/representation analysis (t-SNE visualization of penultimate-layer features),
- **Task 5**: test-time augmentation (TTA) evaluation,
- **Task 6**: scheduler comparison (cosine vs onecycle vs plateau),
- **Task 7**: per-class metrics and error analysis.

Connection to final model choice. Taken together, these experiments form a structured hyper-parameter and design search over both *continuous* choices (learning rate, batch size) and *discrete* choices (augmentation policy, architecture, scheduler, and imbalance mitigation). Under the macro-F1 model-selection criterion, ResNet-18 provides the strongest overall class-balanced performance in our comparisons (Table 2), and we treat it as the primary model for subsequent analysis. We also report complementary outcomes where relevant: for example, ReduceLROnPlateau yields the highest test accuracy in the scheduler study, even though it does not maximize macro-F1.

2.5 Evaluation Metrics

We report multiple metrics because Ottoman character recognition is class-imbalanced and contains rare classes:

- **accuracy**: overall correctness, dominated by frequent classes,
- **macro-F1**: average F1 across classes (class-balanced and sensitive to rare-class failures),
- **per-class precision/recall/F1**: diagnostic breakdown to identify failure modes.

Macro-F1 is used as the primary model-selection metric and for most comparisons, because it penalizes "easy" gains from frequent classes and highlights whether improvements benefit the full label set.

Table 2: Performance on the held-out test set (43 classes).

Model / Setting	Params (M) ↓	Accuracy ↑	Macro-F1 ↑
ResNet-18 (cosine)	11.20	0.8882	0.8587
ResNet-34 (cosine)	21.31	0.8882	0.8321
EfficientNet-B0 (cosine)	4.06	0.8796	0.7738
MobileNetV3-Small (cosine)	1.56	0.8516	0.7575

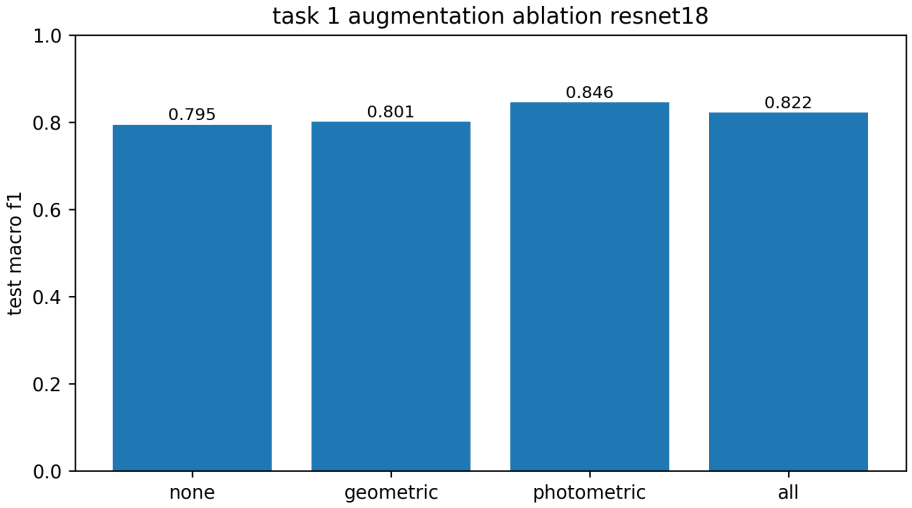

Figure 2: Test macro-F1 for augmentation variants (raw results in `outputs/augmentation_ablation.json`).

2.6 Results

Main metrics. Table 2 shows that ResNet-18 provides the strongest class-balanced performance on this dataset (test macro-F1 0.8587) and is also the fastest model in our inference-time measurement (0.109 ms/image). ResNet-34 matches ResNet-18 in accuracy but loses macro-F1 and incurs higher latency, making it a weaker trade-off under a macro-F1 selection criterion. EfficientNet-B0 and MobileNetV3-Small substantially reduce parameter count, but their macro-F1 drops sharply, suggesting that fine-grained diacritic/stroke distinctions benefit from higher-capacity feature extractors in this low-resolution setting.

Augmentation ablation (Task 1). We isolate augmentation effects by training identical ResNet-18 models with four augmentation configurations: none, geometric-only, photometric-only, and all combined. Results are saved as `outputs/augmentation_ablation.json` and visualized in `outputs/augmentation_ablation.png`. Notably, the configuration that maximizes validation macro-F1 does not necessarily maximize test macro-F1, highlighting the need for cautious selection on small validation sets.

Architecture comparison (Task 2). We compare four ImageNet-pretrained backbones under the same optimizer/loss/schedule and the selected augmentation setting. Figure 3 summarizes test accuracy, macro-F1, parameter count, and inference latency. ResNet-18 achieves the best macro-F1 (0.8587) while remaining the fastest model (0.109 ms/image). ResNet-34 matches ResNet-18 in accuracy (0.8882) but yields lower macro-F1 (0.8321) and higher latency (0.169 ms/image), which is consistent with reduced minority-class generalization at this dataset scale. EfficientNet-B0 and MobileNetV3-Small reduce parameters substantially (4.06M and 1.56M),

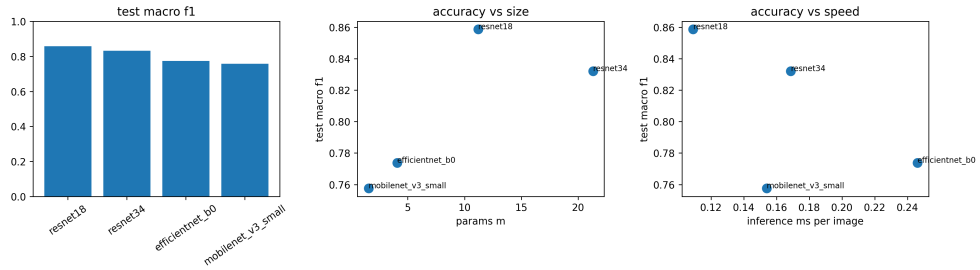

Figure 3: Accuracy–speed–size trade-offs across backbones (raw results in `outputs/architecture_comparison.json`).

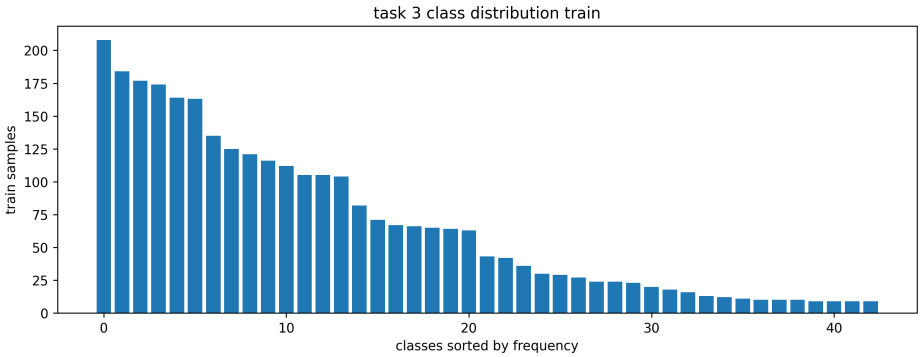

Figure 4: Training-set class frequency distribution.

but their macro-F1 drops to 0.7738 and 0.7575, respectively; notably, EfficientNet-B0 is also the slowest in this implementation, suggesting that parameter count alone is not a reliable proxy for latency.

Class imbalance analysis (Task 3). The dataset exhibits substantial class imbalance (some classes have only 1–3 samples in the test set; see `outputs/per_class_metrics.json`). We compute inverse-frequency class weights from the training split and compare standard vs class-weighted cross-entropy. In this setting, class weighting hurts performance: standard loss achieves test accuracy 0.8839 and macro-F1 0.8517, while weighted loss drops to 0.8516 accuracy and 0.8147 macro-F1. A plausible explanation is that strongly upweighting extremely rare classes can amplify label noise and produce unstable gradients when class support is very small. Figure 5 breaks down minority-class F1, showing that several rare classes degrade under weighting.

Test-time augmentation (Task 5). We evaluate standard inference vs test-time augmentation (TTA) on the test set (`outputs/tta_results.json`). With 5 stochastic views per image, TTA decreases macro-F1 by -0.0194 (from 0.8587 to 0.8393) and slightly decreases accuracy (from 0.8882 to 0.8860), while increasing inference cost. This negative result suggests that some TTA transforms are not perfectly label-preserving at 32×32 resolution (e.g., small rotations can shift diacritics), and that averaging over transformed views can introduce additional uncertainty for fine-grained glyph distinctions.

Scheduler comparison (Task 6). We compare CosineAnnealingLR, OneCycleLR, and ReduceLROnPlateau (`outputs/scheduler_comparison.json`). OneCycleLR achieves the best macro-F1 (0.8423) under this fixed 8-epoch budget, while ReduceLROnPlateau yields the best accuracy (0.8946) with slightly lower macro-F1 (0.8373). CosineAnnealingLR underperforms in

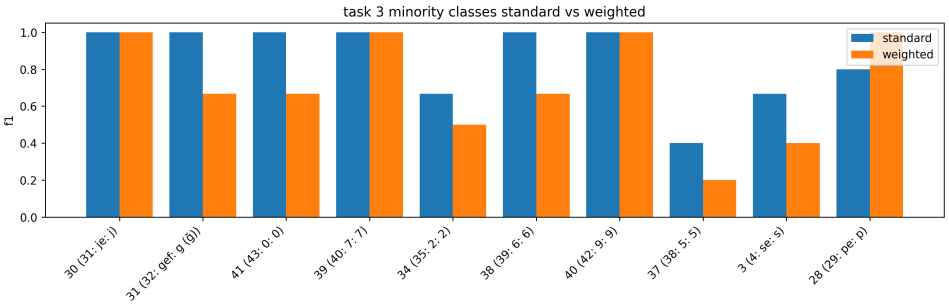

Figure 5: Minority-class F1 comparison: standard vs class-weighted loss (raw results in `outputs/class_imbalance_results.json`).

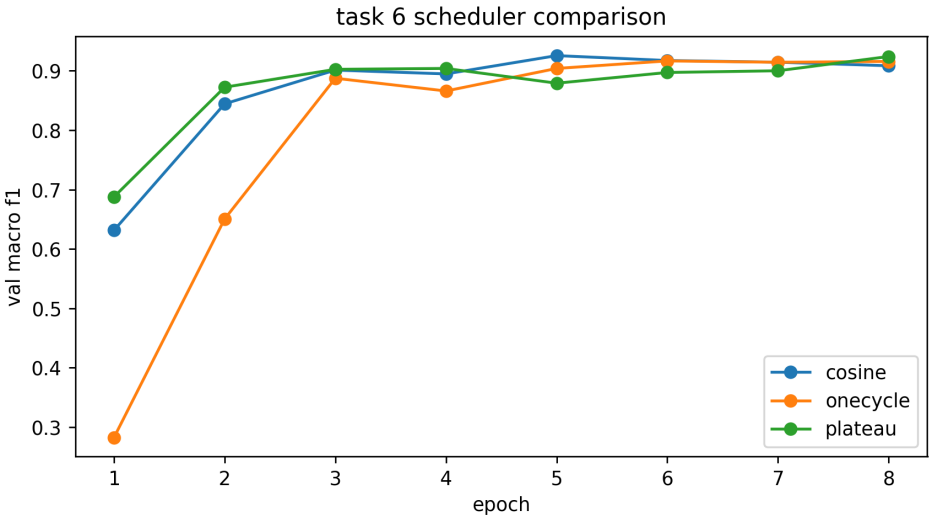

Figure 6: Validation macro-F1 curves across schedulers (raw results in `outputs/scheduler_comparison.json`).

macro-F1 (0.8067). Overall, these results suggest that more aggressive or adaptive learning-rate policies can improve class-balanced performance when training time is limited.

Per-class metrics (Task 7). Per-class precision/recall/F1 is exported in `outputs/per_class_metrics.jso` and visualized in `outputs/per_class_f1.png`. The mean per-class F1 equals the macro-F1 (0.8587) by definition, but the distribution is uneven: most classes exceed 0.8 F1, while a small number of low-support classes dominate the remaining errors. For instance, (38: 5: 5) attains F1 0.333 with support 1, and (9: zel: z) attains F1 0.333 with support 3. These results emphasize that additional labeled data for rare classes (or higher-resolution crops) is likely to yield the largest remaining gains.

Confusion analysis. We export the top confusion pairs to `outputs/top_confusions.json`. Table 3 lists the most frequent confusions, translated into human-readable Ottoman/modern equivalents. Many are plausibly explained by visual similarity or small diacritics that are difficult to resolve at low resolution.

Qualitative error cases.

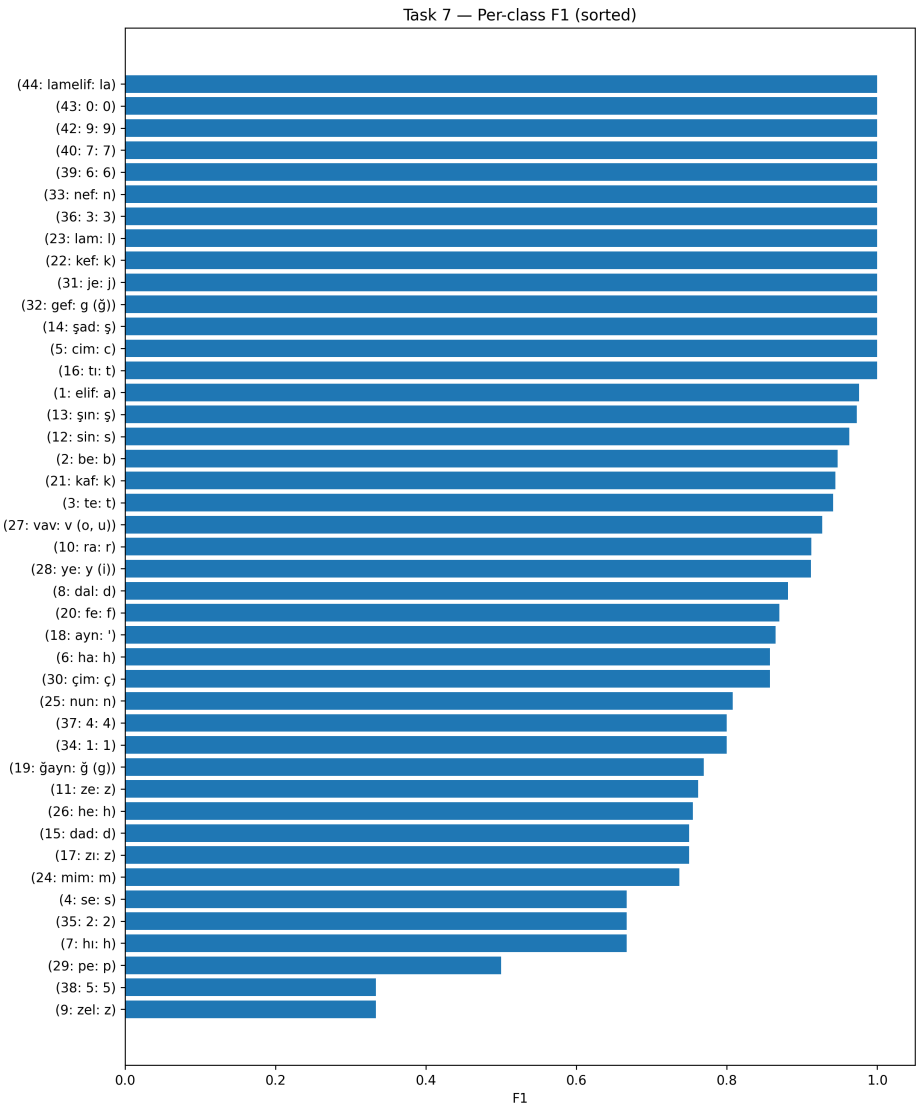

Figure 7: Per-class F1 scores, sorted (raw metrics in `outputs/per_class_metrics.json`).

Embedding visualization (Task 4). To probe whether the learned representation separates visually similar glyphs, we extract penultimate-layer embeddings from the selected model on the test set and project them to two dimensions using t-SNE [13]. While t-SNE is not a quantitative metric, it provides a useful diagnostic: distinct clusters suggest that classes are separable in feature space, whereas heavy overlap is consistent with the confusions observed in the confusion matrix. In this task, we use the visualization primarily to triangulate error analysis (Task 7) and to check whether systematic confusions correspond to overlapping neighborhoods in representation space.

2.7 Demo: from image input to predicted Ottoman character output

To demonstrate end-to-end usability, we provide an interactive demo pipeline that loads a trained checkpoint from the archived artifacts and accepts user images from multiple sources (local path, Drive folder, uploaded files, uploaded ZIP). The demo supports both (i) single-character crops and (ii) best-effort multi-character decoding.

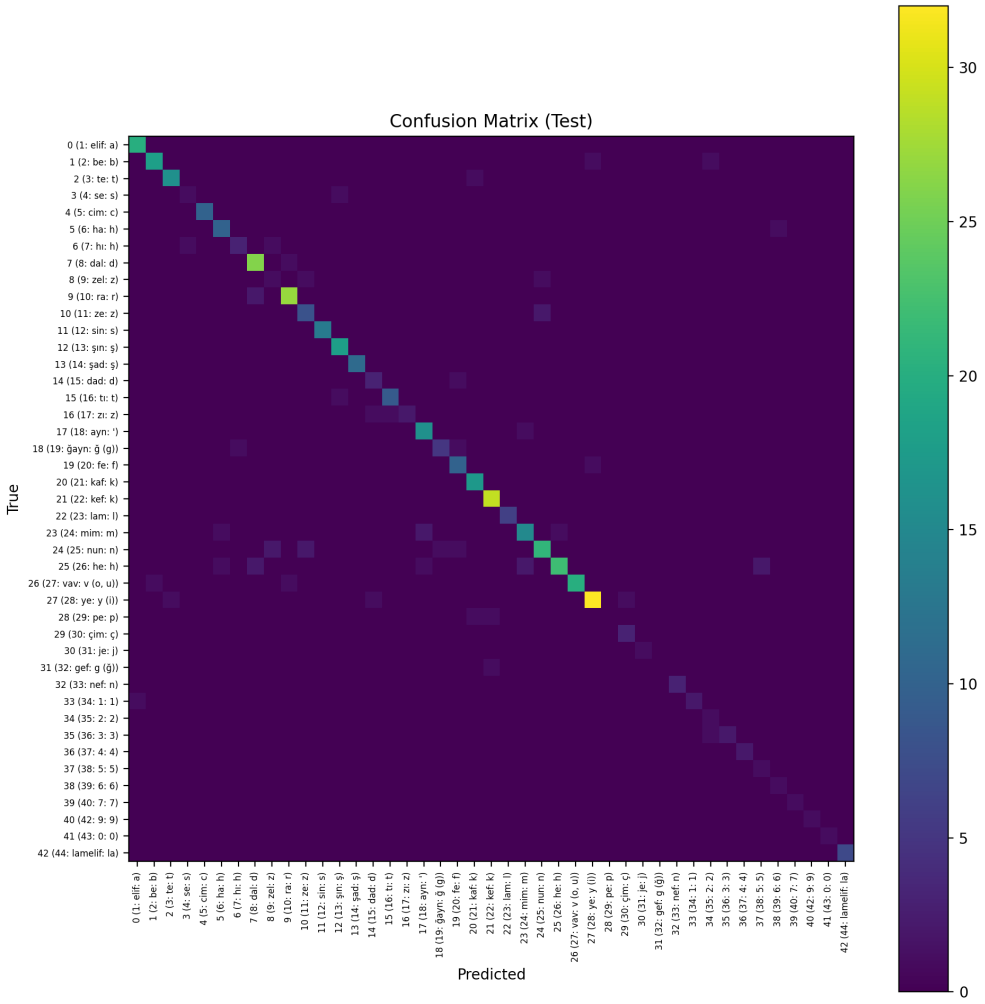

Figure 8: Confusion matrix highlighting common confusions between visually similar Ottoman graphemes.

Single-character mode. Given an input image, the demo preprocesses it with the same normalization used during training and outputs Top-K predictions with probabilities. Predictions are displayed in the interpretable format `(tag: ottoman_name: modern)`, rather than only numeric IDs. For example, on a held-out image tagged as `30` (*çim*), the model outputs:

`(30: çim: ç) (idx=29): p=0.943`

This presentation enables a non-technical user to understand outputs while preserving the exact tag required for evaluation and dataset alignment.

Demo assets. In this repository snapshot, demo assets are included under `demo_images/` (the folder `single_chars/` contains 18 representative images). In the original Colab/Drive environment, these assets were written under `outputs/demo_images/`.

The included demo images in this snapshot are available under `demo_images/single_chars/`.

3 Conclusion

This project demonstrates that transfer learning with modern CNN backbones can provide strong Ottoman character recognition performance on a small, low-resolution historical script

Table 3: Top confusion pairs on the test set (most frequent off-diagonal entries).

True class	Predicted class	Count
(26: he: h)	(8: dal: d)	3
(26: he: h)	(24: mim: m)	3
(26: he: h)	(38: 5: 5)	3
(6: ha: h)	(18: ayn: ')	2
(10: ra: r)	(8: dal: d)	2
(11: ze: z)	(25: nun: n)	2
(24: mim: m)	(18: ayn: ')	2
(24: mim: m)	(26: he: h)	2
(25: nun: n)	(9: zel: z)	2
(25: nun: n)	(11: ze: z)	2

Table 4: The four major components implemented in this project.

Requirement component	Evidence in this project
Dataset Creation	Kaggle dataset ingestion (3,894 images reported by download) and curation for reproducible experimentation: robust indexing/path repair, label mapping, and a fixed stratified split (train/val/test = 2,905/513/465; 3,883 total) created/loaded by the notebook (`splits.json`/`splits_fixed.json` in the original Colab/Drive setup; available via the Google Drive link).
Dataset Annotation	Online label-preserving augmentation + controlled ablation (Task 1; `outputs/augmentation_ablation.json`, `outputs/augmentation_ablation.png`).
Model Training or Fine-Tuning	Fine-tuning ImageNet-pretrained CNNs with checkpointing and evaluation (Task 2; `outputs/architecture_comparison.json`, `outputs/architecture_comparison.png`).
Hyper-Parameter Tuning	LR/BS sweep (`outputs/sweep_results.json`), controlled ablations (Tasks 1–7), and scheduler comparison (`outputs/scheduler_comparison.json`) with selection by validation macro-F1.

dataset. Using a robust preprocessing/data-loading layer and class-balanced evaluation, we reach test macro-F1 up to 0.8587 and accuracy up to 0.8946, and we provide detailed experimental evidence for key design choices (augmentation variants, architecture trade-offs, class weighting, schedulers, and TTA). Per-class metrics and confusion analysis identify where the classifier still fails: primarily rare classes and visually similar glyphs where diacritics or minor strokes are ambiguous at 32×32 scale.

Future work should extend this character classifier toward full OCR by adding detection/segmentation (line/word/character localization), sequence modeling (CTC or transformer decoders), and language-model-based disambiguation. From a data perspective, the largest gains are likely to come from (i) collecting more samples for rare classes, (ii) higher-resolution crops, and (iii) semi-supervised or auto-labeling strategies to scale beyond the small labeled set.

Task 4 — Top confusion pairs (examples)

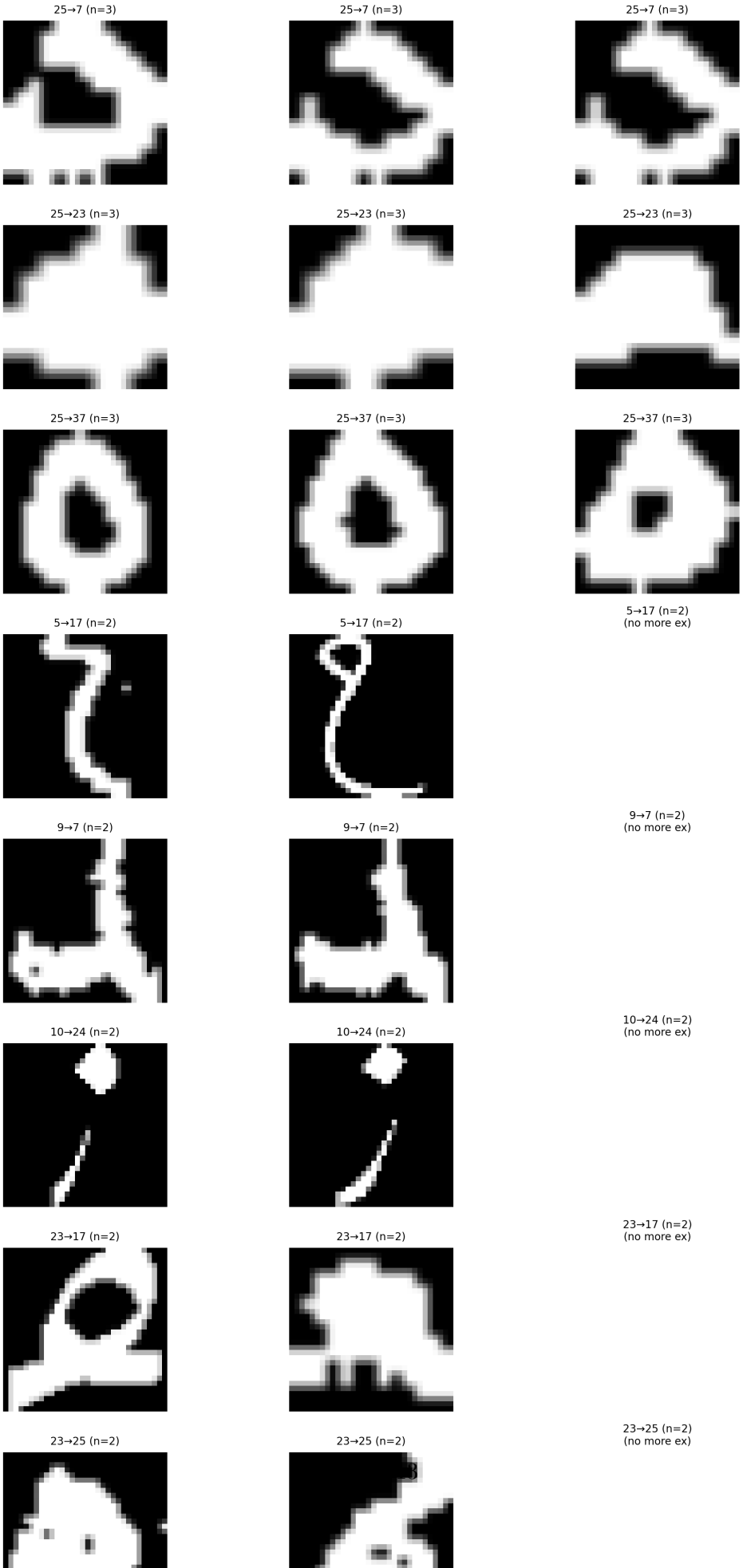

Figure 10: t-SNE [13] of penultimate-layer embeddings colored by true class.

References

[1] A. Buslaev, V. I. Iglovikov, E. Khvedchenya, A. Parinov, M. Druzhinin, and A. A. Kalinin, "Albumentations: Fast and Flexible Image Augmentations," *Information*, vol. 11, no. 2, p. 125, 2020. doi: 10.3390/info11020125.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *Proc. IEEE CVPR*, 2009, pp. 248–255. doi: 10.1109/CVPR.2009.5206848.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. IEEE CVPR*, 2016, pp. 770–778. doi: 10.1109/CVPR.2016.90.

[4] A. Howard *et al.*, "Searching for MobileNetV3," in *Proc. IEEE/CVF ICCV*, 2019, pp. 1314–1324. doi: 10.1109/ICCV.2019.00140.

[5] A. Bintuğ Uzun and A. Özer, *Ottoman Turkish Characters (Kaggle Dataset)*. `https://www.kaggle.com/datasets/alpbintuuzun/ottoman-turkish-characters`, accessed 2026-01-14.

[6] I. Loshchilov and F. Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts," in *International Conference on Learning Representations (ICLR)*, 2017. `https://openreview.net/forum?id=Skq89Scxx`.

[7] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," in *International Conference on Learning Representations (ICLR)*, 2019. `https://openreview.net/forum?id=Bkg6RiCqY7`.

[8] G. Bradski, *OpenCV: Open Source Computer Vision Library*. `https://opencv.org/`, accessed 2026-01-14.

[9] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. `https://papers.nips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html`.

[10] L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay," *arXiv preprint arXiv:1803.09820*, 2018. `https://arxiv.org/abs/1803.09820`.

[11] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," in *Proc. IEEE CVPR*, 2016, pp. 2818–2826. doi: 10.1109/CVPR.2016.308.

[12] M. Tan and Q. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in *Proc. ICML*, 2019, pp. 6105–6114. `https://proceedings.mlr.press/v97/tan19a.html`.

[13] L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008. `http://jmlr.org/papers/v9/vandermaaten08a.html`.

[14] Weights & Biases, Inc., *Weights & Biases*. `https://wandb.ai/`, accessed 2026-01-14.