Building Ultra-Lightweight Models for Image Tasks
Exploring the Lower Bounds of Visual Intelligence

Saptak Bhoumik
saptakbhoumik.acad@gmail.com

Abstract

Most image-based ML models today are bulky, data-hungry, and overkill for many practical tasks. This project takes the opposite approach, how small can we go while still getting the job done?
We explore a range of techniques to aggressively shrink image models, from replacing learned filters with handcrafted ones to simplifying architecture designs, and other clever tricks.

1. Introduction

This paper is version 1 of an ongoing project exploring how small we can make image-based machine learning models while still keeping them functional. The goal is to test the limits by stripping things down to the bare minimum and seeing what survives.

Right now, we are not trying to explain *why* certain tricks work. That will be saved for a future version. This version is focused on *what* works best given hard constraints on parameter count and compute. Many different ideas have been tested, but only the best-performing setup is described here.

For simplicity, we limit this version to a single task: basic binary image classification. Specifically, we work on distinguishing cats from dogs using the well-known Microsoft Dogs vs Cats dataset (`https://www.kaggle.com/c/dogs-vs-cats`). It is simple, popular, and just complex enough to make things interesting.

All code and experiments are available in the GitHub repository: `https://github.com/SaptakBhoumik/HCV`

Future versions of this paper will explore other tasks, include comparisons between methods, and dig into the underlying reasons behind performance. For now, this is just a snapshot of the best setup we have found so far.

2. Cat vs Dog Classification

We test our approach on a basic binary image classification task: distinguishing cats from dogs using the Microsoft Dogs vs Cats dataset (`https://www.kaggle.com/c/dogs-vs-cats`). It is widely used, relatively clean, and visually varied enough to test whether lightweight models can capture meaningful differences.

Dataset and Augmentation

We used the Microsoft Dogs vs. Cats dataset available on Kaggle, which contains a total of 25,000 labeled images. A simple 50% train-validation split was used, giving us 12,500 samples in each set.

To boost the model's ability to generalize, we applied left-right flipping as the only augmentation technique. This flip was applied to both training and validation images. We didn't just use the flipped images on the fly — the original and flipped versions were both included explicitly, doubling the dataset size for both sets.

Preprocessing

We use a handcrafted preprocessing pipeline that transforms grayscale input images into a **30-channel stacked representation**, capturing different aspects of visual structure through classical image processing techniques. The final output is standardized and stacked for model input.

Here is a breakdown of the transformations and the meaning of their parameters:

- **Multi-pass Enhancement**
  - Applies upscaling, denoising, CLAHE, contrast stretching, and sharpening.
  - Channels included: enhanced image and its inversion.
- **Otsu Thresholding**
  - Applies Otsu's binarization to the enhanced image.
  - Channels included: thresholded binary image and its inversion.
- **Canny Edge Detection**
  - Parameters: `(t1_ratio, t2_ratio)` — thresholds as a ratio of the Otsu value.
  - Ratios used: (0.3, 0.7), (0.5, 1.0), (0.8, 1.2)
  - For each pair, we include the Canny edge map and its inversion.
- **Scharr Gradient Magnitude**
  - Parameters: `(scale, delta)` — scale adjusts gradient strength, delta adds a bias.
  - Combinations used: (1, 0), (0.5, 0), (2.0, 0), (1, 20), (1, 50)
  - For each, we include the gradient magnitude and its inversion.
- **Local Binary Patterns (LBP)**
  - Parameters: `(P, R)` — number of sampling points and radius.
  - Configurations used: (8, 1), (16, 2), (24, 3)
  - For each, we include the normalized LBP image and its inversion.
- **Gabor Filters**
  - Parameter: `theta` — orientation angle in radians.
  - Orientations used: 0 (horizontal), $\frac{\pi}{2}$ (vertical)
  - For each, we include the Gabor filter response and its inversion.

All outputs are resized to a fixed resolution of 128x128 and stacked to form the complete input to the model. This approach avoids any learned preprocessing and instead builds on interpretable, handcrafted features designed to capture structure, texture, and edges from multiple perspectives.

Model Design

We designed a compact model that takes the 30-channel stacked image input and processes it through a series of lightweight convolutional layers. The model is built specifically to preserve information from handcrafted features while reducing parameter count and compute requirements. The architecture is modular, with separate blocks for each feature group (Otsu, Canny, Scharr, LBP, Gabor), and is optimized for small models without compromising on visual reasoning ability.

The total compute and size:

- MACs (Multiply-Accumulate Operations): **13.17 million**
- Parameters: **4.64 thousand**

The architecture is implemented as follows:

```
class MishLike(nn.Module):
    def __init__(self, count, is_2d=True, init=1):
        super().__init__()
        shape = (count, 1, 1) if is_2d else (count,)
        self.scale0 = nn.Parameter(torch.ones(*shape) * init)
        self.scale1 = nn.Parameter(torch.ones(*shape) * init)
        self.func = nn.Sequential(nn.Softplus(), nn.Tanh())

    def forward(self, x):
        x0 = self.func(x)
        x1 = torch.where(x > 0, x * self.scale0, x * self.scale1)
        return x0 * x1

def conv_block(in_ch, out_ch, init):
    return nn.Sequential(
        nn.Conv2d(in_ch, in_ch, kernel_size=3, padding=1, stride=2),
        MishLike(in_ch, init=init),
        nn.BatchNorm2d(in_ch),
        nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1, stride=2),
        MishLike(out_ch, init=init),
        nn.BatchNorm2d(out_ch),
    )

class InitialModel17(nn.Module):
    def __init__(self, init=1):
        super().__init__()

        self.conv0 = nn.Sequential(
            nn.Conv2d(30, 30, kernel_size=7, padding=3, stride=2, groups=30),
            MishLike(30, init=init),
            nn.BatchNorm2d(30),
        )

        self.conv1 = nn.Sequential(
            nn.Conv2d(30, 15, kernel_size=7, padding=3, groups=15),
            MishLike(15, init=init),
            nn.BatchNorm2d(15),
        )

        self.otsu_block   = conv_block(2, 1, init)
        self.canny_block  = conv_block(3, 1, init)
        self.scharr_block = conv_block(5, 1, init)
        self.lbp_block    = conv_block(3, 1, init)
        self.gabor_block  = conv_block(2, 1, init)

        self.conv2 = nn.Sequential(
            nn.Conv2d(5, 5, kernel_size=3, padding=1, stride=2),
            MishLike(5, init=init),
            nn.BatchNorm2d(5),
            nn.Conv2d(5, 5, kernel_size=2, stride=2),
            MishLike(5, init=init),
            nn.BatchNorm2d(5),
```

```
            nn.Conv2d(5, 5, kernel_size=2, stride=2),
            MishLike(5, init=init),
            nn.BatchNorm2d(5),
        )

        self.head = nn.Sequential(
            nn.Flatten(),
            nn.Linear(20, 10),
            nn.PReLU(),
            nn.Linear(10, 5),
            nn.PReLU(),
            nn.Linear(5, 2),
        )

    def forward(self, x):
        x = self.conv0(x)
        x = self.conv1(x)

        x_otsu   = self.otsu_block(x[:,  0: 2])
        x_canny  = self.canny_block(x[:, 2: 5])
        x_scharr = self.scharr_block(x[:, 5:10])
        x_lbp    = self.lbp_block(x[:,10:13])
        x_gabor  = self.gabor_block(x[:,13:15])

        x = torch.cat([x_otsu, x_canny, x_scharr, x_lbp, x_gabor], dim=1)
        x = self.conv2(x)
        return self.head(x)
```

Optimization Details

We used the `AdamW` optimizer from PyTorch, which helps prevent overfitting by decoupling weight decay from the gradient update. The loss function was `CrossEntropyLoss`, which is standard for binary classification when working with logits.

Result

The final model was trained on the processed dataset using the architecture described above. The performance metrics on the training and validation (test) sets are as follows:

- **Training Loss**: 0.4121
- **Training Accuracy**: 80.90%
- **Validation Loss**: 0.5107
- **Validation Accuracy**: 75.73%

These results reflect a strong performance for a lightweight model under 5k parameters.

Conclusion

The results so far are promising, especially given the tiny parameter count and minimal compute. A lot of experiments are still ongoing, and future versions of this paper will include more comparisons, ablations, and benchmarks on other tasks.

We haven't gone into the deeper motivations behind this approach yet — that part will be explored in the next version of the paper.

For updates, more experiments, or to argue about filters: saptakbhoumik.acad@gmail.com