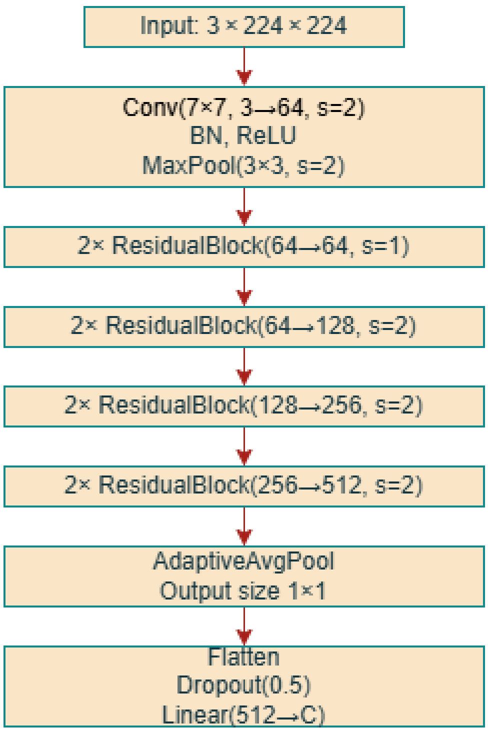
Multi-Label Image Classification using CNN

Duc Tam Nguyen

1 Introduction

Multi-label image classification involves assigning multiple labels to each image to capture the presence of multiple objects or attributes simultaneously. This richer semantic representation is critical in applications such as scene understanding, medical imaging, and multimedia annotation. State-of-the-art solutions leverage deep convolutional neural networks, using transfer learning and loss functions like binary cross-entropy to address non-exclusive targets.

In this project, instead of fine-tuning a standard pretrained backbone, I built and trained a custom `AdvancedCNN` composed of residual blocks. This architecture scales depth and channel width flexibly while preserving gradient flow. I address class imbalance, threshold calibration, and overfitting through stratified data splits, per-class threshold tuning, and data augmentation.



2 Methods/Case Study

2.1 Data Preprocessing

I implemented a custom `MultiLabelDataset` class together with torchvision transforms to load and preprocess images. Each image is:

- Resized to 224×224 pixels.
- Augmented with `RandomHorizontalFlip()` during training to improve generalization.
- Converted to a tensor via `ToTensor()`.
- Normalized using mean and standard deviation of 0.5 in each channel.

```
train_tfms = T.Compose([
    T.Resize((224,224)),
    T.RandomHorizontalFlip(),
    T.ToTensor(),
    T.Normalize([0.5,0.5,0.5],
                [0.5,0.5,0.5]),
])

train_dataset = MultiLabelDataset(train_df,
                             image_directory,
                             transform=train_tfms)
val_dataset   = MultiLabelDataset(val_df,
                             image_directory,
                             transform=val_tfms)
```

2.2 Model Architecture

I built a custom `AdvancedCNN` by stacking residual blocks to allow very deep feature extraction while preserving gradient flow. The key components are:

1. **Stem:** An initial convolutional layer
   - `Conv2d(3,64, kernel_size=7, stride=2, padding=3)`
   - `BatchNorm2d(64)`, `ReLU(inplace=True)`
   - `MaxPool2d(kernel_size=3, stride=2, padding=1)`

   This reduces the input ($3\times224\times224$) to a $64\times56\times56$ feature map.
2. **Residual Stages:** Four stages, each created by my helper `_make_layer()`
   - *Stage1:* two `ResidualBlock(64→64, stride=1)` blocks
   - *Stage2:* two `ResidualBlock(64→128, stride=2)` blocks (downsampling)

- *Stage3:* two `ResidualBlock(128→256, stride=2)` blocks
- *Stage4:* two `ResidualBlock(256→512, stride=2)` blocks

Each `ResidualBlock` contains:

```
conv3×3 → BN → ReLU → conv3×3 → BN
add (identity or downsample) → ReLU
```

3. **Pooling & Classifier:**

- `AdaptiveAvgPool2d((1,1))` reduces to a 512-dim vector.
- `Flatten()` → `Dropout(0.5)` → `Linear(512→C)`
- I chose `BCEWithLogitsLoss` and Adam (lr=1e-4) for multi-label learning.

An instantiation snippet in my training script looks like:

```
model     = AdvancedCNN(num_classes).to(device)
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
```

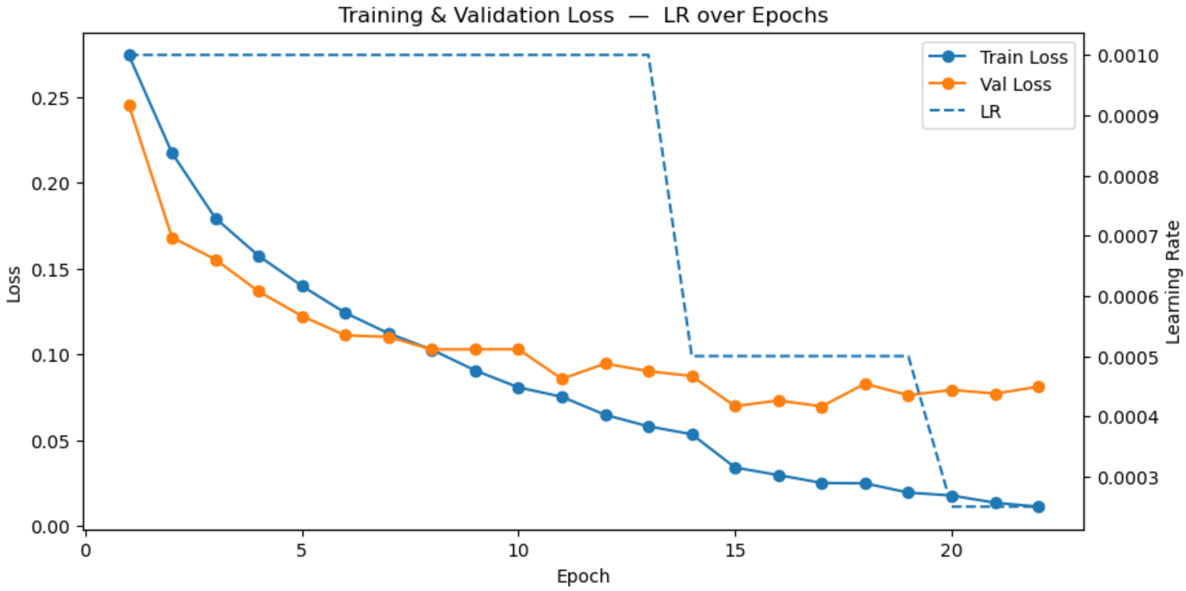
2.3 Training and Hyperparameter Tuning

I trained the `AdvancedCNN` for up to 100 epochs using the Adam optimizer (initial lr=1e-3) and a `ReduceLROnPlateau` scheduler (factor=0.5, patience=2). I implemented early stopping after 5 epochs without validation loss improvement. During each epoch I recorded train/val losses and the learning rate, checkpointing the best model:

```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
scheduler = ReduceLROnPlateau(optimizer,
                              mode='min',
                              factor=0.5,
                              patience=2,
                              verbose=True)

best_val_loss = float('inf')
no_improve    = 0
patience_es   = 5
for epoch in range(1, num_epochs+1):
    train_loss = train_one_epoch(train_loader)
    val_loss   = eval_one_epoch(val_loader)
    scheduler.step(val_loss)
    lr_history.append(optimizer.param_groups[0]['lr'])
```

```
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        no_improve    = 0
        torch.save(model.state_dict(), "best_model.pth")
    else:
        no_improve += 1
        if no_improve >= patience_es:
            break
```



After training converged, I tuned per-class decision thresholds on the validation set by maximizing F1 scores using precision–recall curves:

```
from sklearn.metrics import precision_recall_curve

best_thresh = {}
for i, lbl in enumerate(label_cols):
    p, r, t = precision_recall_curve(y_val_true[:,i],
                                     y_val_prob[:,i])
    f1 = 2*(p*r)/(p+r+1e-8)
    best_thresh[lbl] = t[f1.argmax()]
```

Finally, I evaluated on the test set by applying these thresholds to the predicted probabilities.

2.4 Results and Discussion

After training, I tuned per-class thresholds on the validation set by maximizing F1 via precision–recall curves. Table 1 summarizes the optimal thresholds and corresponding validation F1 scores. Table 2 reports the final test mAP and per-class F1 using these thresholds.

Table 1: Validation thresholds and F1 scores

Class	**Best Threshold**	**Val F1**
motorcycle	0.12	0.902
truck	0.36	0.867
boat	0.58	0.889
bus	0.40	0.889
cycle	0.57	0.916
sitar	0.78	0.906
ektara	0.42	0.913
flutes	0.34	0.951
tabla	0.90	0.984
harmonium	0.27	0.950

Table 2: Test set performance with tuned thresholds

Metric	**Value**
mAP (macro)	0.9604
Class	**Test F1**
motorcycle	0.877
truck	0.850
boat	0.885
bus	0.837
cycle	0.907
sitar	0.914
ektara	0.958
flutes	0.982
tabla	0.982
harmonium	0.960

3 Conclusion

I successfully designed and trained a custom `AdvancedCNN` featuring four stages of residual blocks, achieving a test mAP of **0.9604**. After per-class threshold tuning, most labels exceed F1 of 0.90—however, *bus* (0.837) and *truck* (0.850) remain the weakest, indicating that small object size and background clutter still pose challenges. Key takeaways include:

- Residual connections enabled deep feature extraction with stable training.
- Learning-rate scheduling and early stopping prevented overfitting.
- Per-class threshold calibration significantly boosted F1 scores.

References

- Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)* (pp. 448–456). PMLR.
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 770–778).
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 32.