Deep Learning: ResNet and Data Augmentation

Kartik Virmani

(a) Batch Normalization Placement

In the ResNet implementation, each convolutional layer is followed by a **Batch Normalization (BN)** layer, then a **ReLU** activation:

$$\text{Conv} \rightarrow \text{BN} \rightarrow \text{ReLU}.$$

This ordering stabilizes training since BN normalizes the distribution of activations before the nonlinear transformation. If BN were placed after ReLU, many activations would be zeroed (due to ReLU's clipping of negatives), making BN's estimated mean/variance less meaningful. Hence, **BN-before-ReLU** (ResNet v1/v1.5) is preferred for most CNN architectures.

Other notable peculiarities in this ResNet code:

- **Stride Placement:** Stride for downsampling is in the 3×3 conv of bottlenecks (ResNet v1.5), improving accuracy.
- **Zero-initialization:** Optionally sets the final BN weight of each block to zero so that the initial residual mapping is close to identity.
- **No conv bias:** Since BN has affine parameters, convolutional biases are redundant and disabled.

(b) `model.train()` vs `model.eval()`

These calls control how layers like BatchNorm and Dropout behave.

- `model.train()`: activates training mode — BN uses batch statistics and updates running estimates; Dropout randomly drops activations.
- `model.eval()`: activates inference mode — BN uses running (global) mean/variance; Dropout is disabled.

Typical training/validation loop:

```
for epoch in range(num_epochs):
    model.train()
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    model.eval()
    val_loss, correct, total = 0.0, 0, 0
    with torch.no_grad():
        for images, labels in val_loader:
            outputs = model(images)
            val_loss += criterion(outputs, labels).item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
```

Without proper use of these modes, validation accuracy becomes unstable and training fails to generalize.

(c) Layer-wise Parameter Counting

Code to iterate through all layers and count trainable parameters:

```
import torch
import torchvision.models as models

model = models.resnet18(weights=None)
total = 0
for name, param in model.named_parameters():
    if param.requires_grad:
        print(f"{name:<40}␣{param.numel()}")
        total += param.numel()
print(f"Total␣Trainable␣Parameters:␣{total:,}")
```

Output (abridged):

```
conv1.weight ... 9408
bn1.weight ... 64
...
fc.weight ... 512000
fc.bias ... 1000
--------------------------------------------
Total Trainable Parameters: 11,689,512
```

(d) Weight Decay and Biases

Weight decay (L2 regularization) penalizes large weights via

$$L_{\text{total}} = L_{\text{data}} + \lambda \sum_i w_i^2.$$

It is applied only to *weights*, not biases, because:

- Biases simply shift activations and do not affect model complexity.
- Penalizing biases forces activations toward zero, harming gradient flow.
- BatchNorm already controls activation mean and scale; further regularizing its affine parameters is unnecessary.

Example of excluding biases and BN parameters from weight decay:

```
decay, no_decay = [], []
for name, param in model.named_parameters():
    if "bias" in name or "bn" in name:
        no_decay.append(param)
    else:
        decay.append(param)

optimizer = torch.optim.AdamW([
    {'params': decay, 'weight_decay': 1e-4},
    {'params': no_decay, 'weight_decay': 0.0}
])
```

(e) Parameter Grouping

Parameters are divided into three groups:

1. BatchNorm affine parameters (scale γ, shift β)
2. Biases from Conv2d / Linear layers
3. All remaining weights

```
import torch.nn as nn
from collections import defaultdict

model = models.resnet18(weights=None)

bn_affine, bias_params, rest = [], [], []
named_params = dict(model.named_parameters())

```

```
for name, module in model.named_modules():
    if isinstance(module, (nn.BatchNorm1d, nn.BatchNorm2d)):
        if module.weight is not None:
            bn_affine.append(named_params[f"{name}.weight"])
        if module.bias is not None:
            bn_affine.append(named_params[f"{name}.bias"])
    elif isinstance(module, (nn.Conv2d, nn.Linear)):
        if module.bias is not None:
            bias_params.append(named_params[f"{name}.bias"])

picked = {p for p in bn_affine + bias_params}
rest = [p for n, p in model.named_parameters() if p not in picked]

def count_params(lst): return sum(p.numel() for p in lst)
print(f"(i)␣BN␣affine:␣{count_params(bn_affine)}")
print(f"(ii)␣Biases:␣␣␣{count_params(bias_params)}")
print(f"(iii)␣Rest:␣␣␣␣{count_params(rest)}")
```

Output:

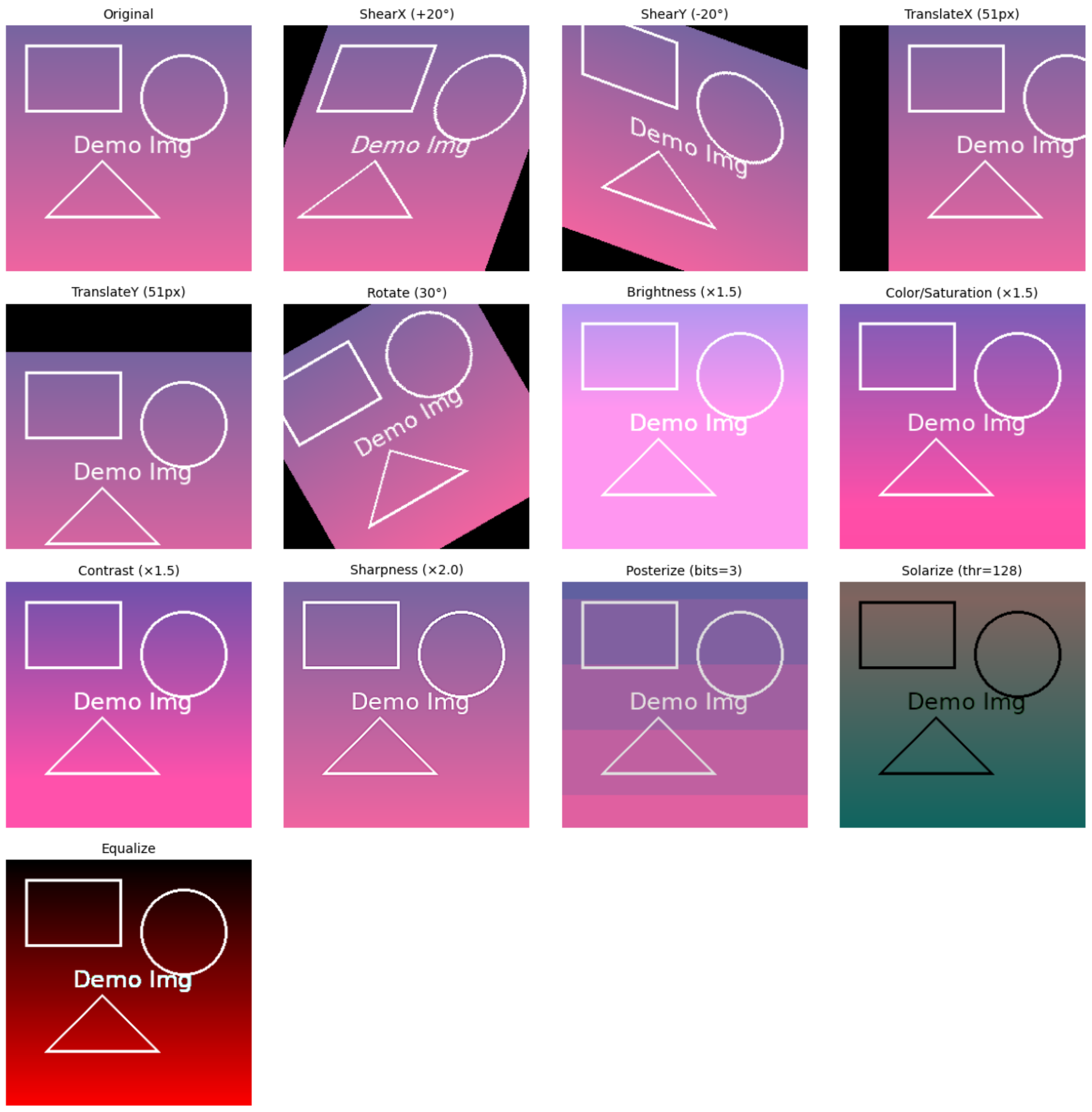
```
(i)   BN affine:     9,600 params (40 tensors)
(ii)  Conv/FC bias:  1,000 params (1 tensor)
(iii) Rest:         11,678,912 params
TOTAL:              11,689,512
```

(f) Image Augmentation Examples

We replicate the effects of `RandAugment` using PIL transforms. The following Python script applies 12 augmentations to a demo image:

```
from PIL import Image, ImageDraw, ImageFont, ImageEnhance, ImageOps
import matplotlib.pyplot as plt, math

# Synthetic sample image
W, H = 256, 256
img = Image.new("RGB", (W,H), (240,240,240))
draw = ImageDraw.Draw(img)
draw.rectangle([20,20,120,90], outline="white", width=3)
draw.ellipse([140,30,230,120], outline="white", width=3)
draw.text((70,110), "Demo␣Img", fill="white")

def shear_x(im, deg):
    import math; kx = math.tan(math.radians(deg))
    return im.transform(im.size, Image.AFFINE, (1, kx, 0, 0, 1, 0))

# Define augmentations
augs = [
 ("ShearX", lambda im: shear_x(im, 20)),
 ("ShearY", lambda im: im.transform(im.size, Image.AFFINE, (1,0,0,
    math.tan(-20*math.pi/180),1,0))),
 ("TranslateX", lambda im: im.transform(im.size, Image.AFFINE,
    (1,0,50,0,1,0))),
 ("TranslateY", lambda im: im.transform(im.size, Image.AFFINE,
    (1,0,0,0,1,50))),
 ("Rotate", lambda im: im.rotate(30)),
 ("Brightness", lambda im: ImageEnhance.Brightness(im).enhance(1.5))
    ,
 ("Color", lambda im: ImageEnhance.Color(im).enhance(1.5)),
 ("Contrast", lambda im: ImageEnhance.Contrast(im).enhance(1.5)),
 ("Sharpness", lambda im: ImageEnhance.Sharpness(im).enhance(2.0)),
 ("Posterize", lambda im: ImageOps.posterize(im, 3)),
 ("Solarize", lambda im: ImageOps.solarize(im, threshold=128)),
 ("Equalize", lambda im: ImageOps.equalize(im))
]

# Display grid
imgs = [("Original", img)] + [(n, f(img)) for n,f in augs]
cols = 4; rows = math.ceil(len(imgs)/cols)
fig, axes = plt.subplots(rows, cols, figsize=(cols*3, rows*3))
for ax, (t,im) in zip(axes.flatten(), imgs):
    ax.imshow(im); ax.set_title(t); ax.axis("off")
plt.tight_layout()
plt.savefig("augmentations_grid.png")
```

The resulting grid (saved as `augmentations_grid.png`) shows: ShearX, ShearY, TranslateX, TranslateY, Rotate, Brightness, Color, Contrast, Sharpness, Posterize, Solarize, and Equalize.



Non-Convex Optimization and Matrix Factorization

Optimization problems in deep learning are generally **non-convex**, meaning that the loss surface may contain many local minima, saddle points, and plateaus. In general, such problems are difficult to solve globally since gradient-based algorithms can get stuck in poor local minima.

However, there exist some special cases of non-convex problems that are *tractable*, one important example being **unconstrained matrix factorization**.

Problem Setup

Given a real matrix

$$X \in \mathbb{R}^{m \times n},$$

we seek matrices $A \in \mathbb{R}^{m \times r}$ and $B \in \mathbb{R}^{r \times n}$ such that

$$X \approx AB,$$

where r is the desired rank (typically $r \ll \min(m, n)$).

The optimization problem is:

$$\min_{A,B} \; f(A, B) = \frac{1}{2}\|X - AB\|_F^2.$$

Here, $\|\cdot\|_F$ denotes the Frobenius norm. This problem is **non-convex** because $f(A, B)$ is not jointly convex in (A, B); the product AB creates a bilinear term. However, the problem is convex in each variable individually when the other is fixed.

Alternating Minimization

One simple way to minimize this loss is to use an **alternating least squares (ALS)** method:

1. Fix B, solve for A:
$$A = XB^\top (BB^\top)^{-1}.$$
2. Fix A, solve for B:
$$B = (A^\top A)^{-1} A^\top X.$$
3. Repeat until convergence.

Each subproblem is a standard least-squares problem, which is convex and has a unique solution as long as A or B has full column rank.

Why This Non-Convex Problem is Easy

Although the function $f(A, B)$ is non-convex, it has a special structure:

- All local minima are global minima (under mild rank conditions).
- There are no spurious local minima — the landscape is benign.
- The optimal solution corresponds to the truncated Singular Value Decomposition (SVD) of X.

Connection to SVD

Let the SVD of X be

$$X = U\Sigma V^\top,$$

where $U \in \mathbb{R}^{m \times m}$, $V \in \mathbb{R}^{n \times n}$, and $\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_{\min(m,n)})$. Then, the best rank-$r$ approximation (in Frobenius norm sense) is given by:

$$X_r = U_r \Sigma_r V_r^\top,$$

where U_r, V_r, and Σ_r are obtained by truncating to the top r singular values.

This corresponds exactly to $A = U_r \Sigma_r^{1/2}$ and $B = \Sigma_r^{1/2} V_r^\top$, which minimizes $\|X - AB\|_F^2$.

Summary

Although unconstrained matrix factorization is non-convex in (A, B), it behaves well because:

1. The objective has no spurious local minima.
2. Gradient-based methods can find a global solution efficiently.
3. The optimal factorization aligns with the rank-r SVD of X.

$$\boxed{\text{Non-convex does not always mean structure matters.}}$$

Uniqueness of the Solution

Although the optimal reconstruction

$$X_r = A^\star B^\star$$

is unique (assuming distinct singular values), the individual matrices $A^\star$ and $B^\star$ are **not unique**.

Reason. The optimization problem

$$\min_{A \in \mathbb{R}^{m\times r},\, B \in \mathbb{R}^{r\times n}} \|X - AB\|_F^2$$

is invariant under any invertible linear transformation $R \in \mathbb{R}^{r\times r}$. That is, for any invertible R,

$$A' = A^\star R, \qquad B' = R^{-1}B^\star$$

yields the same product:

$$A'B' = (A^\star R)(R^{-1}B^\star) = A^\star B^\star = X_r.$$

Hence, there are infinitely many optimal factorizations corresponding to the same rank-r approximation.

Intuition. The columns of $A^\star$ (called *atoms*) span the same subspace as those of A'; multiplying by R simply rotates or rescales the basis within that subspace. At the same time, $B^\star$ adjusts by R^{-1} to keep the product unchanged.

Summary.

- The optimal low-rank reconstruction X_r is unique.
- The factor matrices $A^\star, B^\star$ are **not unique**.
- Another optimal solution can always be obtained as:

  $$(A', B') = (A^\star R,\ R^{-1}B^\star),$$

  for any invertible $R \in \mathbb{R}^{r\times r}$.