[0.2] CNNs and ResNets

David Quarel

ARENA

June 9, 2026

What's today about?

- Goal: by the end of today you will have built **ResNet34** from raw PyTorch primitives, loaded pretrained weights, and used it for transfer learning.
- The road there:
  1. Layers as `nn.Modules` — the LEGO bricks.
  2. Assemble a multi-layer perceptron, train it on MNIST.
  3. Convolutions: a much better prior for images.
  4. Skip connections + batch norm $\Rightarrow$ ResNet.
  5. Feature extraction: stand on the shoulders of pretrained giants.

Why deep learning?

- Classical ML: hand-engineer features, learn a thin classifier on top.
- Deep learning: *learn the features too*, end-to-end, from raw data.
- **Universal approximation:** a sufficiently wide MLP with a nonlinearity can approximate any continuous function on a compact set.
  - Existence, not efficiency. Architecture priors (convolutions, attention, skip connections) are what make it tractable.
- Interpretability work later: knowing exactly what the model is made of is the prerequisite for taking it apart.

The `nn.Module` pattern

- Everything in a PyTorch model is a `Module`: layers, blocks, the whole network. A `Module` bundles *parameters* with a `forward` pass.

```
class Linear(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        # define parameters here
        self.W = nn.Parameter(torch.randn(d_out, d_in))
        self.b = nn.Parameter(torch.zeros(d_out))

    def forward(self, x):
        # defines what the module does
        y = einsum(x, self.W, "... d_in, d_out d_in -> ... d_out") + self.b
        # equiv. y = x @ self.W.T + self.b
        return y

>>> module = Linear(d_in=3, d_out=5)
>>> module(x) #equiv. to module.forward(x)
```

- `nn.Parameter` ⇒ tracked by `.parameters()`, updated by the optimizer.
- Modules can nest other modules; used often today

The training loop

- Same four steps every iteration: *forward*, *loss*, *backward*, *step*.

```
for x, y in dataloader:                       # sample data in batches
    y_hat = model(x)                          # forward pass through model
    loss = my_loss_function(y,y_hat)          # how good was the model?
    loss.backward()                           # compute gradient of loss w.r.t parameters
    optimizer.step()                          # update parameters to decrease loss
    optimizer.zero_grad()                     # reset gradients before next batch
```

- `loss.backward()` fills `.grad` on every parameter via autograd.
- `optimizer.step()` applies an update rule (e.g. Adam: momentum + per-parameter adaptive learning rate).
- `zero_grad()` before backward: gradients **accumulate** by default in PyTorch.

Cross-entropy loss

- Model outputs logits $z \in \mathbb{R}^K$; convert to probabilities with softmax:

$$\hat{p}_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

- True label y is one-hot. Cross-entropy:

Cross-Entropy

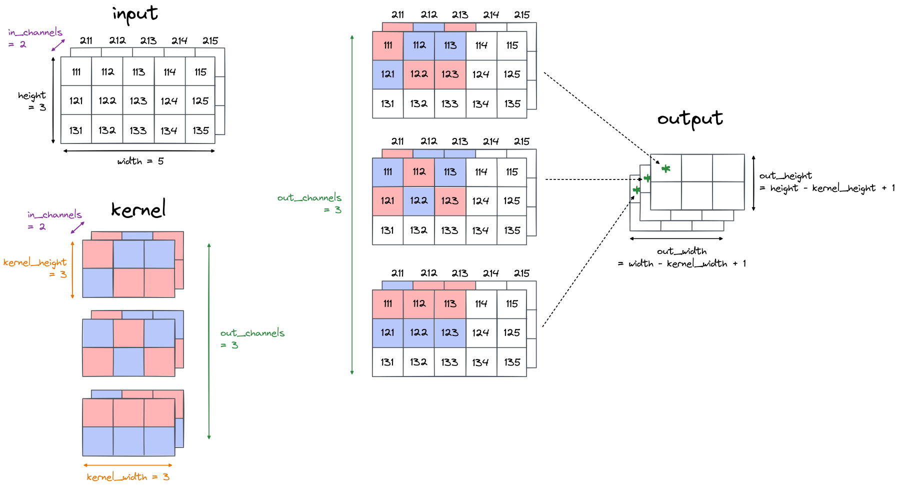
$$\mathcal{L}(z, y) = -\sum_k y_k \log \hat{p}_k = -\log \hat{p}_{y^*}$$

where y^* is the index of the true class.

- PyTorch's `F.cross_entropy` takes *logits directly* and fuses softmax + log internally for numerical stability.

Why convolutions?

- A 224×224 RGB image is a 150,528-dimensional vector. A fully-connected first layer with 1000 hidden units has $\sim 1.5 \times 10^8$ parameters just for one layer.
- Images have structure that an MLP completely ignores. Convolutions bake in three priors:
    1. **Locality:** nearby pixels are more related than distant ones.
    2. **Translation equivariance:** a cat in the corner is still a cat. Same filter, applied everywhere.
    3. **Weight sharing:** one kernel reused across all positions $\Rightarrow$ orders of magnitude fewer parameters.
- Same idea, different domain: this is to images what attention is to sequences.


input
in_channels = 2
height = 3
width = 5
211 212 213 214 215
111 112 113 114 115
121 122 123 124 125
131 132 133 134 135
kernel
in_channels = 2
kernel_height = 3
kernel_width = 3
out_channels = 3
output
out_height = height - kernel_height + 1
out_width = width - kernel_width + 1

MaxPool

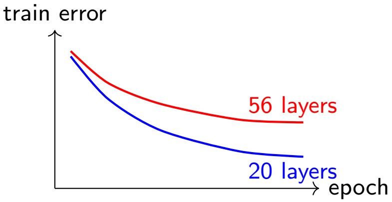
- **Pooling** downsamples a feature map; we replace each $k \times k$ window with its maximum:

$$\begin{array}{|c|c|c|c|} \hline 1 & 3 & 2 & 1 \\ \hline 4 & 2 & 1 & 0 \\ \hline 0 & 1 & 5 & 2 \\ \hline 2 & 1 & 3 & 0 \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|} \hline 4 & 2 \\ \hline 2 & 5 \\ \hline \end{array}$$

- No learnable parameters. Provides small-shift invariance and shrinks the spatial dimensions cheaply.
- Used between conv stages to gradually trade spatial resolution for channel depth.

Going deeper hurts (the degradation problem)

- Intuition: a 56-layer net should be *at least* as good as a 20-layer net: it could just learn identity in the extra layers.
- Empirically, plain deep nets do **worse**, even on training data.



- Gradients struggle to propagate cleanly through many layers; identity is hard to *learn* from scratch.

Skip connections (residual learning)

- **Idea:** make identity the *default*. Each block learns a residual $F(x)$ added to its input.

Residual block

$$y = F(x; \theta) + x$$

- Gradients flow back through the identity branch unimpeded.
- Now training 100+ layer networks is routine.

Pretrained models and feature extraction

- Training ImageNet-scale models from scratch is expensive. Almost no one does it twice.
- **Transfer learning:** take a pretrained backbone, replace the final classifier, train only the new head on your task.
  - Freeze backbone: `p.requires_grad = False` for every backbone parameter.
  - Swap in a fresh `Linear` sized for your new label set.
  - Optimizer only sees the unfrozen parameters.
- Today: take your ResNet34, freeze it, retrain the head on CIFAR-10.

Glossary

- **Module** (`nn.Module`): bundle of parameters + a `forward`. Composable.
- **Parameter** vs **Buffer**: both stored in `state_dict`; only Parameters get gradients.
- **ReLU**: $\max(0, x)$. The default nonlinearity.
- **Linear**: $y = Wx + b$. Kaiming-init for ReLU nets.
- **Cross-entropy**: $\mathcal{L} = -\log \hat{p}_{y^*}$.
- **Conv2d**: kernel + stride + padding sliding over spatial dims.
- **MaxPool**: take the max over a $k \times k$ window; no params.
- **BatchNorm**: normalize per channel; learned γ, β; running stats in eval mode.
- **Residual block**: $y = F(x) + x$. Identity is the default.
- **Training step:** forward → loss → `backward` → `step` → `zero_grad`.
- **Feature extraction:** freeze backbone, train new head on a new task.

The whole point of today: build each row above yourself, so by the end none of it is a black box.