Deep Learning from First Principles

Lesson 8 - A LeNet-style CNN

Andrea Giardina
contact@andreagiardina.com
https://www.linkedin.com/in/agiardina

Today's Goal: Build a LeNet-style CNN from Scratch

In the previous lessons we implemented all the core components of backpropagation for "Torcino", our mini deep–learning framework.

Today's lesson has two objectives:

1. **Complete the MaxPool backward pass** This is the only new mathematical piece: we learn how max pooling routes gradients only through the maximal element of each window.
2. **Reorganize all the code into PyTorch-style layers:**
   - `Conv2d`, `Linear`, `ReLU`, `MaxPool2d`, `Flatten`
   - `Sequential`, `SoftmaxCrossEntropyLoss`, `SGD`

   Then we combine them into a coherent framework for building a CNN.

Final target: Construct and train, from scratch, a complete **LeNet-style convolutional network** for digit classification using only our NumPy implementation.

MaxPool Backprop: Intuition

Forward

MaxPool selects the maximum value inside each spatial window (kernel size kH x kW). Each window produces one output value.

Key Idea

Only the element that achieved the maximum in the forward pass receives the gradient in the backward pass. All other elements in the same window receive zero.

Therefore MaxPool backprop is essentially a masking operation.

MaxPool Backprop: Step by Step

Given

Input X of shape (C, H, W)
Output Y of shape (C, H_out, W_out)
Upstream gradient dY with same shape as Y

1. For each channel independently
2. For each pooling window:
   - Find the index (i, j) of the max value selected in the forward pass
   - Create a zero window
   - Insert dY at position (i, j)
3. Place this gradient window back into dX in the correct location

Result: dX has same shape as X.

MaxPool Backprop: Formula

Let W be the set of indices inside one pooling window, and let argmax(W) be the index of the forward maximum.

Gradient rule

For each window:

$$dX[i,j] = dY \quad \text{if } (i,j) = argmax(W)$$

$$dX[i,j] = 0 \quad \text{otherwise}$$

Important

MaxPool has no learnable parameters.
Backprop only distributes gradients to input positions.
Works independently for each channel.

Lab Time: Update MaxPool2d Forward and Implement Backward

Goal

Refactor MaxPool2d so that the forward pass saves all the information needed for the backward pass using a cache (internal state).

- Modify `forward(self, x)` so that it stores:
  - the input `x` (or the padded version if padding >0)
  - the pooling parameters (kH, kW, stride)
  - any intermediate representation needed (e.g. X_col or argmax mask)
- Implement `backward(self, dY)`:
  - Retrieve cached values saved during forward
  - Reconstruct pooling windows using im2col
  - Create masks that identify max positions
  - Multiply masks by dY and use col2im to obtain dX
- Only `dY` should be passed into backward.

Reminder

The next layer does NOT pass `x`. Your MaxPool2d must store forward inputs internally using a cache, just like PyTorch and other autodiff frameworks.

My trivial implementation

```
class MaxPool2d:
    def __init__(self, kernel_size, stride=None, padding=0):
...
        self.cache = None  # NEW: cache to store values for backward

    def forward(self, x):
...
        self.cache = {
            "x": x,                             # padded input
            "input_shape": (N, C, H, W), # original (before padding)
        }

        return out
```

My trivial implementation

```python
def backward(self, dY):
    # dY: (N, C, H_out, W_out) return: dX with shape (N, C, H, W)
    x = self.cache["x"]
    N, C, H_pad, W_pad = x.shape
    N2, C2, H_out, W_out = dY.shape
    kH, kW = self.kernel_size
    p = self.padding
    s = self.stride
    dX_pad = np.zeros_like(x)

    for n in range(N):
        for c in range(C):
            X_col = im2col(x[n, c:c+1], kH, kW, stride=s)
            max_vals = X_col.max(axis=0, keepdims=True)
            max_mask = (X_col == max_vals)
            dY_flat = dY[n, c].reshape(1, -1)
            dX_col = max_mask * dY_flat
            dX_patch = col2im(dX_col, 1, H_pad, W_pad, kH, kW, stride=s)
            dX_pad[n, c] = dX_patch[0]
    if p > 0:
        dX = dX_pad[:, :, p:-p, p:-p]
    else:
        dX = dX_pad

    return dX
```

Lab time: Implementing a Linear Layer

Goal: Create a `Linear` layer similar to `nn.Linear` in PyTorch.

What we already know (from previous lessons):

- Forward and backward of a fully connected layer.
- Use of He initialization (Normal with $\sigma = \sqrt{2/\text{fan\_in}}$).
- How layers store parameters and accumulate gradients.
- Broadcasting rules (useful for handling the bias term).

Your task: write a class `Linear` with:

- Weight matrix: (out_features, in_features)
- Optional bias vector: (out_features)
- He initialization for the weights

Class `Linear`: Function Signatures

Implement the following class:

```
class Linear:
    def __init__(self, in_features, out_features, bias=True):
        ...

    def forward(self, x):
        """
        x: shape (N, in_features)
        return: (N, out_features)
        """
        ...

    def backward(self, dY):
        """
        dY: gradient from next layer, shape (N, out_features)
        return: dX (N, in_features)
        """
        ...

    def zero_grad(self):
        """Reset gradient buffers."""
        ...
```

My trivial implementation

```
class Linear:
    def __init__(self, in_features, out_features, bias=True):
        self.in_features = in_features
        self.out_features = out_features

        # PyTorch: weight shape = (out_features, in_features)
        self.weight = he_normal((out_features, in_features), fan_in=in_features)
        self.bias = np.zeros(out_features, dtype=np.float32) if bias else None


        self.grad_weight = np.zeros_like(self.weight)
        self.grad_bias = np.zeros_like(self.bias) if bias else None

        self.x = None
```

My trivial implementation

```
def forward(self, x):
    """
    x: shape (N, in_features)
    return: shape (N, out_features)
    """
    self.x = x # Cached for the backward

    y = x @ self.weight.T
    if self.bias is not None:
        y = y + self.bias
    return y

def backward(self, dY):
    """
    dY: gradient from the next layer, shape (N, out_features)
    return: dX, shape (N, in_features)
    """

    # dL/dX = dY @ W
    dX = dY @ self.weight

    # dL/dW = dY^T @ X
    # shape: (out_features, in_features)
    self.grad_weight += dY.T @ self.x
    if self.bias is not None:
        self.grad_bias += dY.sum(axis=0)

    return dX

def zero_grad(self):
    self.grad_weight.fill(0.0)
    if self.grad_bias is not None:
        self.grad_bias.fill(0.0)
```

Lab time: Implement ReLU as a Layer

We already implemented ReLU as a simple function. Now we want to turn it into a **proper layer**, just like `Conv2d`, `MaxPool2d`, or `Linear`.

Specifications:

- Create a class `ReLU` with methods:

```
class ReLU:
    def forward(self, x):
        ...
    def backward(self, dY):
        ...
```

Hints:

- ReLU has no parameters (no weight, no bias).
- In the forward pass, store a mask indicating where $x > 0$.
- In the backward pass, use this mask to zero out gradients where the unit was inactive.

My trivial implementation

```python
class ReLU:
    def forward(self, x):
        self.mask = (x > 0)
        return x * self.mask

    def backward(self, dY):
        return dY * self.mask
```

Exercise: Implementing a `Flatten` Layer

We need a layer that converts convolutional feature maps into a vector, so they can be passed to fully connected layers.

Goal: Implement a `Flatten` layer with the following behavior:

```
class Flatten:
    def forward(self, x):
        ...
    def backward(self, dY):
        ...
```

Specifications:

- Input shape: `(N, C, H, W)`
- Output shape: `(N, C*H*W)`
- Store the original shape during the forward pass.
- In the backward pass, reshape the gradient back to `(N, C, H, W)`.
- No learnable parameters.

My trivial implementation

```
class Flatten:
    def __init__(self):
        self.original_shape = None

    def forward(self, x):
        # x has shape (N, C, H, W)
        self.original_shape = x.shape
        N = x.shape[0]
        return x.reshape(N, -1)

    def backward(self, dY):
        # dY has shape (N, C*H*W)
        return dY.reshape(self.original_shape)
```

Let's glue things together: a `Sequential` Model

In our framework we now have many layers:

- `Conv2d`, `MaxPool2d`
- `ReLU`, `Flatten`
- `Linear`

To build a real neural network, we need a way to:

- chain these layers in the correct order,
- run a full forward pass with a single call,
- run a full backward pass automatically,
- collect all parameters (weights and biases) for the optimizer.

This is exactly the role of `Sequential`, just like in PyTorch:

$$\text{output} = \text{model}(x)$$

It acts as a container that executes each layer in sequence.

Lab time: Implement a Minimal `Sequential` Container

Write a `Sequential` class that takes any number of layers and behaves like a simple PyTorch-style model.

Your class must implement:

```
class Sequential:
    def __init__(self, *layers):
        ...
    def __call__(self, x):
        ...
    def forward(self, x):
        ...
    def backward(self, dY):
        ...
    def parameters(self):
        ...
```

Hints:

- `__call__` should simply call `forward`.
- `forward`: pass the output of each layer to the next.
- `backward`: iterate over layers in reverse order.
- `parameters`: by convention, collect `(weight, grad_weight)` and optionally `(bias, grad_bias)` from layers that have them.

My trivial implementation

```python
class Sequential:
    def __init__(self, *layers):
        self.layers = list(layers)

    def __call__(self, x):
        return self.forward(x)

    def forward(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def backward(self, dY):
        for layer in reversed(self.layers):
            dY = layer.backward(dY)
        return dY

    def parameters(self):
        params = []
        for layer in self.layers:
            if hasattr(layer, "weight"):
                params.append((layer.weight, layer.grad_weight))
            if hasattr(layer, "bias") and layer.bias is not None:
                params.append((layer.bias, layer.grad_bias))
        return params
```

Why Do We Need a Loss Function?

To train a neural network we need a scalar loss that measures how far our predictions are from the correct labels.

For classification tasks the standard loss is:

$$\textbf{Softmax} + \textbf{CrossEntropy}$$

- Softmax converts logits into probabilities.
- CrossEntropy penalises incorrect predictions.
- The loss produces the gradient that starts backpropagation.

We've already implemented the 'SoftmaxCrossEntropyLoss' in lesson number 4, but now we want to imlpement a **PyTorch-style** API:

$$\text{loss} = \text{loss\_fn}(y_{\text{pred}}, y_{\text{true}})$$

$$dY = \text{loss.backward}()$$

Implementation: Forward Pass (NumPy)

From Logits to Loss

- `logits`: Logits from the last layer. Shape `(N, C)`.
- `targets`: One-hot labels. Shape `(N, C)`.
- `N` = batch size, `C` = number of classes.

Step 1: Softmax

```
shifted = logits - np.max(logits, axis=1, keepdims=True)
exp = np.exp(shifted)
probs = exp / exp.sum(axis=1, keepdims=True)
```

Step 2: Cross-Entropy Loss (old version)

```
log_probs = np.log(probs)
loss_samples = -np.sum(targets * log_probs, axis=1)
total_loss = np.mean(loss_samples)
```

Step 2: Cross-Entropy Loss (alternative version, more efficient)

```
N = probs.shape[0]
total_loss = -np.log(probs[np.arange(N), targets]).mean()
```

Implementation: Backward Pass (NumPy)

The Gradient for Backpropagation

- **Loss:** $L = \frac{1}{N} \sum L_{\text{sample}}$
- **Gradient:** $\frac{\partial L}{\partial z_i} = \frac{1}{N}(a_i - y_i)$

Gradient w.r.t. logits dZ(old version)

```
N = targets.shape[0]
dLogits = (probs - targets) / N
```

Gradient w.r.t. logits dZ (alternative version, more efficient

```
dLogits = probs.copy()
dLogits[np.arange(N), targets] -= 1
dLogits /= N
```

Lab time: Implement `SoftmaxCrossEntropyLoss`

Write a loss class with the following structure:

```
class SoftmaxCrossEntropyLoss:
    def __call__(self, logits, targets):
        ...
    def forward(self, logits, targets):
        ...
    def backward(self):
        ...
```

Specifications:

- `logits`: shape `(N, C)` (output of last Linear)
- `targets`: shape `(N,)` with class indices
- Save what you need during the forward pass
- Compute probabilities with a numerically stable softmax
- Implement the batch mean cross-entropy loss
- Backward must return the gradient w.r.t. logits

My trivial implementation

```python
class SoftmaxCrossEntropyLoss:
    def __call__(self, logits, targets):
        return self.forward(logits, targets)

    def forward(self, logits, targets):
        shifted = logits - logits.max(axis=1, keepdims=True)
        exp = np.exp(shifted)
        self.probs = exp / exp.sum(axis=1, keepdims=True)
        self.targets = targets
        return -np.log(self.probs[np.arange(logits.shape[0]), targets]).mean()

    def backward(self):
        N = self.probs.shape[0]
        dLogits = self.probs.copy()
        dLogits[np.arange(N), self.targets] -= 1
        dLogits /= N
        return dLogits
```

Lab time: Implement a Minimal SGD Optimizer

To update the model's parameters after backpropagation, we need an optimizer. The simplest one is **Stochastic Gradient Descent (SGD)**.

Implement the following class:

```
class SGD:
    def __init__(self, model, lr=0.01):
        ...
    def zero_grad(self):
        ...
    def step(self):
        ...
```

Specifications:

- The optimizer receives the `model` and can retrieve its parameters via `model.parameters()`.
- `zero_grad()` must reset all gradients to zero by iterating over `model.parameters()`.
- `step()` must update each parameter:

$$\theta \leftarrow \theta - \mathrm{lr} \cdot \nabla_{\theta}$$

- No additional features: no momentum, no weight decay, etc.

My trivial implementation

```
class SGD:
    def __init__(self, model, lr=0.01):
        self.model = model
        self.lr = lr

    def zero_grad(self):
        for param, grad in self.model.parameters():
            grad[...] = 0

    def step(self):
        for param, grad in self.model.parameters():
            param[...] -= self.lr * grad
```

Typical Training Pipeline (Big Picture)

- **Goal:** learn model parameters that minimize a loss on the training data.
- Typical steps:
  1. Load and split the dataset (train / test).
  2. Preprocess inputs (scaling, normalization, reshape).
  3. Define the model (layers and parameters).
  4. Choose a loss function and an optimizer.
  5. Training loop over epochs and mini-batches:
     - forward pass
     - compute loss
     - backward pass (gradients)
     - optimizer step (update parameters)
  6. Evaluate on a separate test set.

Step 1: Data Loading and Preprocessing

- We start from NumPy arrays on disk:
  - `Xtr, ytr`: training images and labels
  - `Xte, yte`: test images and labels
- Preprocessing pipeline:
  1. **Type cast** to the right dtypes (e.g. `float32`, `int64`).
  2. **Rescale** pixel values to a standard range (e.g. divide by 255).
  3. **Normalize** using train mean and std:
     - fit on `Xtr` $\Rightarrow$ get `mean`, `std`
     - apply the same transform to `Xtr` and `Xte`
  4. **Reshape** to the format expected by Conv2d:

$$(N, H, W) \rightarrow (N, C, H, W)$$

(here: $C = 1$, $H = W = 28$)

- This step is independent from the specific model architecture.

Step 2: Defining the Model

- We build a **computation graph** using layers:

$$\text{model} = \text{Sequential}(\text{layers...})$$

- Example architecture (LeNet-style):
  - `Conv2d(1, 4, 5)`: 1 input channel $\rightarrow$ 4 feature maps, kernel 5×5.
  - `ReLU()`: non-linear activation.
  - `MaxPool2d(2, 2)`: downsampling.
  - `Flatten()`: from (N, C, H, W) to $(N, C \cdot H \cdot W)$.
  - `Linear(4 * 12 * 12, 32)`.
  - `ReLU()`.
  - `Linear(32, 10)`: 10 classes (digits 0–9).
- Model parameters (weights, biases) are initialized randomly and will be updated during training.

Step 3: Loss Function and Optimizer

- **Loss function** measures how bad the predictions are.
  - Here: `SoftmaxCrossEntropyLoss()` for multi-class classification.
  - Input: logits from the model and true labels `y`.
  - Output: a scalar loss (average over the mini-batch).
- **Optimizer** updates the model parameters using the gradients.
  - Here: `SGD(model, lr=0.01)` (Stochastic Gradient Descent).
  - Keeps a reference to all parameters in the model.
  - The learning rate controls the step size in parameter space.
- Once these are defined, we can run the training loop.

Step 4: Training Loop Structure

- Training happens in **epochs**: one epoch = one full pass over the training set.
- Inside each epoch, we use **mini-batches**: split the N training samples into chunks of size `batch_size`, each mini-batch produces one gradient update.

For each epoch:

- Shuffle the training data.

For each mini-batch we perform:

1. Forward pass
2. Compute loss
3. Backward pass (gradients)
4. Optimizer step (update parameters)

- This pattern is shared by most deep learning frameworks (PyTorch, TensorFlow, ...).

Inside One Mini-batch Step

- Given a mini-batch `(xb, yb)`:
  1. **Forward pass**
     - `logits = model(xb)`
     - compute predictions for all samples in the batch
  2. **Compute loss**
     - `loss = loss_fn(logits, yb)`
     - scalar value = average loss over the batch
  3. **Zero existing gradients**
     - `optimizer.zero_grad()`
     - avoid accumulating from previous steps
  4. **Backward pass**
     - `dY = loss_fn.backward()`
     - `model.backward(dY)`
     - compute gradients for all parameters via backpropagation
  5. **Optimizer step**
     - `optimizer.step()`
     - update each parameter using its gradient (SGD rule)
- This is one *learning step* in parameter space.

Step 5: Evaluation on Test Set

- After training, we evaluate the model on unseen data.
- Only **forward pass**, no gradients, no updates:
  - `logits_te = model(Xte)`
  - `y_pred = argmax(logits_te, axis=1)`
- Compute metrics:
  - **Accuracy**:

$$\text{acc} = \frac{\#\{\text{correct predictions}\}}{\#\{\text{test samples}\}}$$

  - other possible metrics: loss on test set, precision, recall, ...
- Important: test data are never used to update parameters.

Lab time: Build a LeNet-style CNN (1/2)

We want to train a small convolutional network on MNIST using the layers we implemented so far.

1. **Load the preprocessed arrays**
   - `train_images.npy` with shape $(60000, 784)$
   - `train_labels.npy` with shape $(60000,)$
   - `test_images.npy`, `test_labels.npy`
2. **Preprocessing steps**
   1. Scale pixel values to $[0, 1]$ by dividing by 255.
   2. Normalize using `normalize_fit` and `normalize_apply`.
   3. Reshape to **NCHW**: $(N, 1, 28, 28)$.
3. **Define a small LeNet-style model**
   - `Conv2d(1, 4, 5)`
   - `ReLU()`
   - `MaxPool2d(2,2)`
   - `Flatten()`
   - `Linear(4*12*12, 32)`
   - `ReLU()`
   - `Linear(32, 10)`

Use our `Sequential` container to combine the layers.

Exercise: Build a LeNet-style CNN (2/2)

Once the model is defined, follow the standard training loop:

1. **Create the loss and optimizer**
   - `SoftmaxCrossEntropyLoss()`
   - `SGD(model, lr=0.01)`

2. **For each epoch:**
   1. Shuffle the training data.
   2. Split into mini-batches (e.g. size 64).
   3. For each batch:
      - Forward pass: `logits = model(xb)`
      - Compute loss
      - Backward pass: `optimizer.zero_grad()`, `dY = loss.backward()`, `model.backward(dY)`
      - Update parameters: `optimizer.step()`

3. **Evaluate:** run the model on the test set and compute accuracy.

My trivial implementation

```
Xtr = np.load("train_images.npy").astype(np.float32)[:10000]
ytr = np.load("train_labels.npy").astype(np.int64)[:10000]
Xte = np.load("test_images.npy").astype(np.float32)
yte = np.load("test_labels.npy").astype(np.int64)

Xtr /= 255.0
Xte /= 255.0
mean, std = normalize_fit(Xtr)
Xtr = normalize_apply(Xtr, mean, std)
Xte = normalize_apply(Xte, mean, std)

Xtr = Xtr.reshape(-1, 1, 28, 28)
Xte = Xte.reshape(-1, 1, 28, 28)

model = Sequential(
    Conv2d(1, 4, 5),
    ReLU(),
    MaxPool2d(2,2),
    Flatten(),
    Linear(4 * 12 * 12, 32),
    ReLU(),
    Linear(32, 10)
)
```

My trivial implementation

```
loss_fn = SoftmaxCrossEntropyLoss()
optimizer = SGD(model, lr=0.01)

batch_size = 64
epochs = 1
N = Xtr.shape[0]

for epoch in range(epochs):
    idx = np.random.permutation(N)
    Xtr_shuf = Xtr[idx]
    ytr_shuf = ytr[idx]
    for i in range(0, N, batch_size):
        #print(f"Batch N. {i}")
        xb = Xtr_shuf[i:i + batch_size]
        yb = ytr_shuf[i:i + batch_size]
        logits = model(xb)
        loss = loss_fn(logits, yb)
        optimizer.zero_grad()
        dY = loss_fn.backward()
        model.backward(dY)
        optimizer.step()
```

My trivial implementation

```
logits_te = model(Xte)
y_pred = np.argmax(logits_te, axis=1)
acc = (y_pred == yte).mean()
print("Test accuracy:", acc)
```

Summary and Relation to Real Frameworks

- A typical training loop always follows the same pattern:
  1. data loading and preprocessing
  2. model definition
  3. loss and optimizer
  4. epoch / mini-batch training loop
  5. evaluation on validation / test sets
- Our NumPy framework makes all steps explicit:
  - `model.backward(dY)`, `optimizer.step()`, etc.
- In libraries like PyTorch or TensorFlow:
  - the same logic is used, but often with higher-level helpers (e.g. `DataLoader`, `loss.backward()`, `optimizer.step()`).
- Understanding this high-level structure is key to reading and writing training code in any deep learning toolbox.

Thanks!