Convolutional Neural Network from Scratch

Bhanu Prasanna Koppolu

Table of contents

Source Code

Full implementation available at DL_from_scratch/cnn.py

Convolutional Neural Network

Core Concept

The key concept in CNN is calculating what is sent to the next layer through convolution operations.

Pooling vs Batch Normalization

Pooling:

- Downsamples spatial dimensions
- Reduces computational cost
- Loses some spatial information

Batch Normalization:

- Normalizes feature distribution using mean and variance
- Preserves spatial dimensions
- Stabilizes training

Output Size Calculation

For convolution and pooling operations:

$$\text{output\_size} = \left\lfloor \frac{\text{input} + 2 \times \text{padding} - \text{kernel}}{\text{stride}} \right\rfloor + 1$$

Example with Pooling:

- Input: (batch=1, channels=32, height=28, width=28)
- MaxPool2d(kernel=2, stride=2)
- Output: (1, 32, 14, 14) $\rightarrow$ Half the spatial size

$$h_{out} = \frac{28 + 0 - 2}{2} + 1 = 14$$

Standard CNN Pattern

```
Conv -> BatchNorm -> ReLU -> MaxPool (repeat 2-3 times) -> Flatten -> Dense
```

Implementation

2D Convolution Layer

```python
class CNN2D:
    """2D Convolutional Layer with stride and padding support.

    Performs correlation (not true convolution) for forward pass.
    """

    def __init__(self,
                 in_channels=1,    # Number of input channels
                 out_channels=32,  # Number of filters
                 kernel=3,         # Kernel size
                 stride=1,         # Stride for sliding
                 padding=0):       # Zero padding

        # Xavier initialization
        self.weights = np.random.randn(out_channels, in_channels, kernel, kernel) \
                       * np.sqrt(2.0 / (in_channels * kernel * kernel))
        self.bias = np.zeros(out_channels)

    def _pad_input(self, X):
        """Apply zero padding to input."""
        ...

    def _correlate2d(self, input_slice, kernel):
        """2D correlation (sliding kernel over input)."""
        ...

    def forward(self, X):
        """Forward pass: apply convolution filters.

        For each output channel:
            Sum correlation of input channels with corresponding kernels
            Add bias
```

```
        """
        ...

    def backward(self, out_gradient, lr=0.001):
        """Backward pass with gradient computation.

        1. If stride > 1, upsample gradient
        2. Compute kernel gradients via correlation
        3. Compute input gradients via full convolution
        4. Update weights and bias
        """
        ...
```

Activation and Pooling Layers

```
class ReLU:
    """ReLU activation: max(0, x)"""

    def forward(self, X):
        self.input_data = X
        return np.maximum(0, X)

    def backward(self, out_gradient):
        return out_gradient * (self.input_data > 0)


class MaxPool2D:
    """Max Pooling layer for downsampling.

    Keeps track of max indices for backward pass.
    """

    def __init__(self, pool_size=2, stride=2):
        self.pool_size = pool_size
        self.stride = stride

    def forward(self, X):
        """Select maximum value in each pooling window.
        Store indices for backward pass."""
        ...
```

```
    def backward(self, out_gradient):
        """Route gradients only to max positions."""
        ...
```

Flatten and Dense Layers

```
class Flatten:
    """Flatten spatial dimensions for dense layers."""

    def forward(self, X):
        self.input_shape = X.shape
        return X.reshape(1, -1)

    def backward(self, out_gradient):
        return out_gradient.reshape(self.input_shape)


class Dense:
    """Fully connected layer."""

    def __init__(self, input_size, output_size):
        # He initialization
        self.weights = np.random.randn(input_size, output_size) \
                       * np.sqrt(2.0 / input_size)
        self.bias = np.zeros(output_size)

    def forward(self, X):
        """Linear transformation: Y = XW + b"""
        ...

    def backward(self, out_gradient, lr=0.001):
        """Compute and apply gradients."""
        ...
```

Softmax and Loss

```
class Softmax:
    """Softmax activation for classification."""
```

```
    def forward(self, X):
        exp_X = np.exp(X - np.max(X, axis=1, keepdims=True))
        self.output = exp_X / np.sum(exp_X, axis=1, keepdims=True)
        return self.output

    def backward(self, y_true):
        # Combined softmax + cross-entropy gradient
        return self.output - y_true


def cross_entropy_loss(y_pred, y_true):
    """Cross entropy loss for one-hot encoded labels."""
    return -np.sum(y_true * np.log(y_pred + 1e-8)) / y_pred.shape[0]
```

Model Architecture

```
# Build CNN model
conv1 = CNN2D(in_channels=1, out_channels=8, kernel=3, stride=1, padding=1)
relu1 = ReLU()
pool1 = MaxPool2D(pool_size=2, stride=2)

conv2 = CNN2D(in_channels=8, out_channels=16, kernel=3, stride=1, padding=1)
relu2 = ReLU()
pool2 = MaxPool2D(pool_size=2, stride=2)

flatten = Flatten()
dense1 = Dense(16 * 7 * 7, 128)  # After two 2x2 poolings: 28->14->7
relu3 = ReLU()
dense2 = Dense(128, 10)
softmax = Softmax()
```

Training Loop

The training loop processes one sample at a time:

1. **Forward pass** through all layers sequentially
2. **Compute loss** using cross-entropy
3. **Backward pass** propagating gradients through all layers in reverse
4. **Weights update** happens inside each layer's backward method