

Backpropagation for CNNs

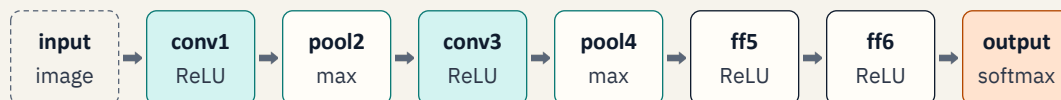
Forward: compute the loss.
Backward: assign the blame.

Michael Wollowski

1

THE NETWORK

Seven layers, one loss



Forward

Each layer turns its input into an activation. The softmax output is compared with the label to give the loss.

Backward

Each layer receives the gradient from the layer after it, finds its own weight gradients, and passes a gradient back.

2

FORWARD

Forward pass

```
public double[] forward(double[][] input) {  
    conv1Forward();  
    pool2Forward();  
    conv3Forward();  
    pool4Forward();  
    ff5Forward();  
    ff6Forward();  
    outputForward();  
    return outputLayer; // probabilities  
}
```

Keep for backprop

- The input to every layer
- Every ReLU output
- The winning cell of each max-pool window

BACKWARD

Backward pass

```
public void backward(int label) {  
    outputBackward(label);  
    ff6Backward();  
    ff5Backward();  
    pool4Backward();  
    conv3Backward();  
    pool2Backward();  
    conv1Backward();  
}
```

Mirror image

- The same seven calls, reversed
- Each call reads the gradient of the layer after it
- Each call hands a gradient to the layer before it

TRAINING

One training step

```
public void train(double[][] input, int label) {  
    clearGradients();  
    forward(input);  
    backward(label);  
    updateWeights();  
}
```

Why clear first?

Backward accumulates with +=, so old gradients would leak in.

Optional: Minibatches

Clear once, sum over the batch, average, then update once.

GRADIENTS

Every layer does three things

1 Receive the gradient from the next layer

```
double[] delta = nextLayerGradients;
```

2 Weight gradient: delta times this layer's input

```
weightGradients[n][i] += delta[n] * input[i];
```

3 Pass back: delta times this layer's weights

```
inputGradients[i] += weights[n][i] * delta[n];
```

ReLU and max-pool have no weights. They only mask or route the gradient.

GRADIENTS

Output layer: softmax and cross-entropy

```
outputGradients[i] = outputLayer[i] - (i == label ? 1.0 : 0.0);  
outputBiasGradients[i] = outputGradients[i];  
  
outputBiases[i] -= learningRate * outputBiasGradients[i]; // update step
```

	A	B
p	0.690	0.310
y	0	1
p - y	0.690	-0.690

Reading it

The desired label is B, so $y = (0, 1)$.

The update subtracts the gradient.
A positive gradient pushes the value down.

Class A is pushed down, class B is pulled up.

GRADIENTS

Fully connected layers: FF6 and FF5

```
for (int i = 0; i < ff6Size; i++) {  
    for (int n = 0; n < outputSize; n++)  
        ff6Gradients[i] +=  
            outputGradients[n] * outputWeights[n][i];  
    if (ff6Output[i] <= 0)  
        ff6Gradients[i] = 0.0;  
    ff6BiasGradients[i] = ff6Gradients[i];  
}
```

Order matters

- ReLU derivative goes after the inner loop
- FF5 is identical, one layer back

GRADIENTS

Pool4: gradients through a dense layer

```
for (int n = 0; n < ff5Size; n++)
  for (int f = 0; f < conv3FilterCount; f++)
    for (int i = 0; i < h; i++)
      for (int j = 0; j < w; j++)
        pool4Gradients[f][i][j] +=
          ff5Gradients[n] * ff5Weights[n][f][i][j];
```

Pooling has no ReLU and no weights, so it only passes the gradient on.

GRADIENTS

Max-pooling routes the gradient to the winner

Forward

2.0	0.0
1.5	0.5

Output 2.0, and the position (0, 0) is remembered.

Backward

0.276	0.0
0.0	0.0

Gradient 0.276 lands there; the other three get 0.

GRADIENTS

Conv3: route, mask, accumulate

```
// 1. route to the window's max
conv3Gradients[f][maxI][maxJ] +=
    pool4Gradients[f][i][j];

// 2. ReLU derivative
if (conv3Output[f][i][j] <= 0)
    conv3Gradients[f][i][j] = 0.0;

// 3. bias gradient
conv3BiasGradients[f] +=
    conv3Gradients[f][i][j];
```

Saved in forward

maxI and maxJ come from pool4Forward, which records each window's winner.

Order matters

Route first, mask with ReLU second, sum for the bias last.

GRADIENTS

Pool2: a full convolution

```
for (f, i, j, m, a, b) // six nested loops
    pool2Gradients[m][i+a][j+b] +=
        conv3Gradients[f][i][j] * conv3Filters[f][m][a][b];
```

Why not like Pool4?

Each pool2 cell sums over every filter window that covers it.

And conv1?

conv1Backward mirrors conv3Backward, but stops at the image.

UPDATING WEIGHTS

Dense layers: gradient times input

```
outputWeights[n][i] -= learningRate
    * outputGradients[n] * ff6Output[i];

ff6Weights[n][i] -= learningRate
    * ff6Gradients[n] * ff5Output[i];
```

Use what flowed in

The input is ff6Output for the output layer, ff5Output for FF6, and pool4Output for FF5. Biases subtract learningRate times their gradient.

UPDATING WEIGHTS

Conv weights: sum over all positions

```
for (f, i, j, m, a, b)
    conv3FilterGradients[f][m][a][b] +=
        conv3Gradients[f][i][j] * pool2Output[m][i+a][j+b];

conv3Filters[f][m][a][b] -=
    learningRate * conv3FilterGradients[f][m][a][b];
```

The filter is shared, so sum the contributions from every position, then update each weight once. Conv1 uses the same code with the image as input.

WORKSHEET

Now do it **by hand.**

A tiny CNN, one training step, every number on paper.