

CNN – Lab 3

Backpropagation and first tests

CSSE 313 Artificial Intelligence · Lab 3 (80 points)

Name: _____ Date: _____

Overview

In Lab 2 you implemented the forward pass of LeNet-5. In this lab you implement the backward pass: the gradients of every layer and the weight update. You then convince yourself, with a handful of simple tests, that your gradients are right and that the network learns. Lab 4 then puts the trained network through more demanding tests.

Please work on it by yourself. As before, you may use AI to write and debug your code and to explain concepts to you. The hand computations in Part A and the answers in Part D must be your own.

Part A — Before you code: a backward pass by hand

15 points

Do this on paper first. Your numbers become the expected results for Test T1.

A1. Softmax and cross-entropy (4 points)

A three-class output layer receives the scores $z = [2, 1, 0]$, and the correct class is $\text{label} = 0$. The forward and backward computations are:

```
// forward: probabilities and loss
sum = 0
for k = 0 to 2:  sum = sum + exp(z[k])
for k = 0 to 2:  p[k] = exp(z[k]) / sum
loss = -ln(p[label])

// backward: gradient of the loss for each score
for k = 0 to 2:
  if k == label:  dz[k] = p[k] - 1
  else:          dz[k] = p[k]
```

1. Compute p and loss (four decimals).
2. Compute dz . What do its three entries add up to, and why must that be so?

A2. Convolution, ReLU and pooling, backwards (11 points)

Return to the example from Lab 1, Part 1B ($X, K, b = -2$, and your Z, A and P). Suppose the layers above report the gradient of the loss for each entry of the pooled output:

```
dP =  1  -1
      2   1
```

Run the following steps by hand. dA , dZ and dK are arrays of the same size as A , Z and K .

```
// 1. max pooling, backwards: each gradient goes to the window's winner
dA = all zeros (4x4)
for each pooling window (wy, wx), wy and wx in 0..1:
    (r, c) = position in A of the winner    // first maximum, row by row
    dA[r][c] = dP[wy][wx]

// 2. ReLU, backwards: the gradient passes only where Z was positive
for y = 0 to 3, x = 0 to 3:
    if Z[y][x] > 0: dZ[y][x] = dA[y][x]
    else:           dZ[y][x] = 0

// 3. convolution, backwards: kernel and bias gradients
for r = 0 to 2, c = 0 to 2:
    dK[r][c] = 0
    for y = 0 to 3, x = 0 to 3:
        dK[r][c] = dK[r][c] + dZ[y][x] * X[y + r][x + c]
db = sum of all dZ[y][x]

// 4. one gradient-descent step, learning rate 0.1
for r = 0 to 2, c = 0 to 2: K[r][c] = K[r][c] - 0.1 * dK[r][c]
b = b - 0.1 * db
```

1. (3) Step 1: write down dA (4×4).
2. (2) Step 2: write down dZ . Why does the gradient of the bottom-right window contribute nothing?
3. (4) Step 3: compute dK (3×3) and db .
4. (2) Step 4: give the new K and b .

Part B — Implement backpropagation

graded through Part C

1. Continue with your `LeNet5.java` and `MNISTCNN.java` from Lab 2.
2. Download the [CNN Labs3-4 Provided Code.zip](#)
3. Add a constructor `LeNet5(long seed)` that draws the initial weights from `java.util.Random(seed)` instead of `Math.random()`. The same seed must give the same network, so runs can be repeated and compared. The provided code calls this constructor.
4. Change the initial weights to the range $[-0.1, 0.1)$ and use learning rate 0.01, both as named constants. (Part D asks why not $[-0.05, 0.05)$.)
5. During the forward pass, keep what the backward pass will need: each layer's output (or its input to ReLU), and for every pooling window the position of the winning entry.
6. Implement one backward procedure per kind of layer, mirroring your forward procedures. Each one receives the gradient for its outputs ($dOut$) and produces the gradients for its weights, its biases and its inputs (dIn), which become $dOut$ for the layer below. The pseudocode below covers every layer type; Part A already used the pooling, ReLU and convolution steps.

```
// output layer: softmax + cross-entropy (as in A1)
```

```

dz[k] = p[k] - (k == label ? 1 : 0)

// fully connected (F6, output): out[j] = b[j] + sum of W[j][i] * in[i]
for each output j:
    dB[j] = dOut[j]
    for each input i:
        dW[j][i] = dOut[j] * in[i]
        dIn[i] = dIn[i] + W[j][i] * dOut[j]    // dIn starts at zero

// ReLU: pass the gradient only where the input was positive (as in A2, step 2)
dIn[n] = (input[n] > 0) ? dOut[n] : 0

// max pooling (S4, S2): send each gradient to the winning entry (as in A2, step 1)

// convolution (C5, C3, C1):
// out[o][y][x] = b[o] + sum over i, r, c of W[o][i][r][c] * in[i][y+r][x+c]
for each output map o, each position (y, x):
    g = dOut[o][y][x]
    dB[o] = dB[o] + g    // all gradients start at zero
    for each input map i, each r, c in 0..4:
        dW[o][i][r][c] = dW[o][i][r][c] + g * in[i][y+r][x+c]
        dIn[i][y+r][x+c] = dIn[i][y+r][x+c] + g * W[o][i][r][c] // skip in C1

```

C5 can be treated as a convolution whose outputs are 1x1, or as a fully connected layer over 400 inputs.

6. Implement `train(double[][] image, int label, boolean verbose)`: forward pass, backward pass, then update every weight and bias. One example at a time. After `train` returns, `getPrediction()` and `computeLoss(label)` must still describe the forward pass that `train` just ran, because `MNISTTrainer` relies on this.

```

for every weight or bias w with gradient dw:
    w = w - LEARNING_RATE * dw    // LEARNING_RATE = 0.01

```

7. Keep the methods the provided code uses (some come from Lab 2):

```

public LeNet5(long seed)
public double[] forward(double[][] image32, boolean verbose) // 10 probs
public void train(double[][] image32, int label, boolean verbose)
public int getPrediction() // index of the largest output probability
public double computeLoss(int label) {
    return -Math.log(outputLayer[label] + 1e-10);
}

// getters used by NetworkVisualizer and MNISTTrainer
public double[][] getInputLayer()    public double[][][] getConv1Output()
public double[][][] getPool1Output() public double[][][] getConv2Output()
public double[][][] getPool2Output() public double[] getFc1Output()
public double[] getFc2Output()       public double[] getOutputLayer()

```

Part C — First tests**50 points**

Paste the output of each test into your write-up. If a test fails, record what you saw and how you found and fixed the problem.

T1. The hand trace in code (9 points)

Call your convolution, ReLU and pooling procedures, forward and then backward, on the Part A example (a few lines in a main). Compare dA , dZ , dK and db with your hand computation. If they differ, find out which one is wrong.

T2. One step downhill (8 points)

Take the first MNIST training image. Print the loss, call `train` once on that image, and print the loss on the same image again. The loss must go down. If it goes up, a sign is wrong somewhere.

T3. Finite-difference gradient check (14 points)

Write a small method (AI may write it) that checks the gradient of one parameter w on one training image:

```
h = 0.00001

// the gradient your backward pass computes
run forward and backward on (image, label) WITHOUT updating the weights
analytic = the gradient your code stored for w

// the gradient measured by nudging w
original = w
w = original + h;   lossPlus = loss after a forward pass on (image, label)
w = original - h;   lossMinus = loss after a forward pass on (image, label)
w = original
numeric = (lossPlus - lossMinus) / (2 * h)

relativeError = |analytic - numeric| / max(|analytic|, |numeric|)
```

For "without updating", a method that does everything `train` does except the final update works well.

Check one weight and one bias in each of C1, C3, C5, F6 and the output layer (10 parameters) and report them in a table. Choose parameters whose gradient is not zero. A correct implementation gives relative errors well below 0.00001.

Layer	Parameter (indices)	Analytic	Numeric	Relative error
C1 weight				
C1 bias				
C3 weight				
C3 bias				
C5 weight				

Layer	Parameter (indices)	Analytic	Numeric	Relative error
C5 bias				
F6 weight				
F6 bias				
Output weight				
Output bias				

T4. Learn three synthetic digits (9 points)

Run the provided `TrainingExample.java`. It trains for 100 epochs on three tiny hand-drawn digits (0, 1, 2). With a correct backward pass the average loss falls close to zero and the accuracy reaches 100%.

T5. One epoch of MNIST (10 points)

With the four MNIST files in your working directory, run `java MNISTTrainer 1 1` (one epoch, seed 1). It trains on the first 50,000 training images, reports accuracy on the last 10,000 training images (the *validation* set) every 5,000 examples, and then reports the test accuracy once. Paste the output.

Validation accuracy should be well above 90% by the end of the epoch.

Part D — What do the tests tell you?**15 points**

- (4) Suppose T3 passes for the output layer and F6 but fails for C5, C3 and C1. Where is the bug most likely to be, and why? (Think about the order in which the backward pass computes the gradients.)
- (3) In T3, some parameters have an analytic and a numeric gradient that are both exactly 0. What does the check tell you about such a parameter? Give two reasons why a gradient in this network can be exactly 0 for one image.
- (4) T4 passes. Name two kinds of bugs that T4 would *not* reveal, and explain why memorizing three images is a weak test of a network with about 61,000 parameters.
- (4) Run `TrainingExample` once with initial weights in $[-0.05, 0.05]$ (as in Lab 2) and once with $[-0.1, 0.1]$. Report both outputs. Explain in words why smaller initial weights make a five-layer network start learning so much more slowly. (Hint: think about how large the signal is that reaches the output, and how large the error signal is that gets back to C1.)

Submission

Submit `LeNet5.java`, `MNISTCNN.java` and any other code you wrote, together with a write-up containing Part A, the output of T1–T5, and your answers to Part D. Put your name at the top of the write-up.