

## Performance-Evaluation der potenziellen Engpässe der neuronalen Netzimplementierung in SetlX

(network.stlx durch sigmoid\_timing.stlx oder zip\_timing.stlx ersetzen zum Testen)

1. Aufruf der Funktion zip(l1, l2). Bildung jeweils einer Liste für l1 und l2 (durch toList(v)) bedeutet zusätzlichen Rechenaufwand.

Codeabschnitte mit Zeitmessungspunkten:

```
sgd := procedure(training_data, epochs, mini_batch_size, eta, test_data) {
  if(test_data != null) {
    n_test := #test_data;
  }
  n := #training_data;

  for(j in {0..epochs}) {
    s1 := now();
    training_data := shuffle(training_data);
    // Get mini-batches from the training data to train the network
    mini_batches := [ training_data[k..k+mini_batch_size-1] : k in [1,mini_batch_size..n] ];

    for(mini_batch in mini_batches) {
      update_mini_batch(mini_batch, eta);
    }
    epoche_time := now() - s1;
    // Visual output
    if(test_data != null) {
      ev := evaluate(test_data);
      //print("Epoch $j$: $ev$ / $n_test$");
      print("Zipping-time:\t" + zip_time);
      print("Epoche-time:\t" + epoche_time);
      print("--> " + 100.0 * zip_time/epoche_time + "%");
      this.zip_time := 0;
    }
    else {
      print("Epoch $j$ complete");
    }
  }
};
```

```
/* Computes the Cartesian product of two matrices or vectors */
zip := procedure(l1, l2) {
  s1 := now();
  res := toList(l1) >< toList(l2);
  this.zip_time += (now() - s1);

  return res;
};
/* casts vector to list */
toList := procedure(v) {
  return [v[i] : i in [1..#v]];
};
```

Anzahl Datensätze: 10.000 Testsätze, 10.000 Trainingssätze

Rechnerdaten: Intel Core i7-4720HQ, 16GB RAM, NVIDIA GeForce GTX 960M

#### Ergebnisse 1.:

```
Start SGD
Zipping-time: 7154
Epoche-time: 23136
--> 30.921507607192254%
Zipping-time: 6451
Epoche-time: 19665
--> 32.8044749555047%
Zipping-time: 6006
Epoche-time: 20937
--> 28.68605817452357%
Zipping-time: 6371
Epoche-time: 19349
--> 32.926766241149416%
Zipping-time: 6550
Epoche-time: 20508
--> 31.938755607567778%
Zipping-time: 6229
Epoche-time: 18742
--> 33.23551381922954%
Zipping-time: 6248
Epoche-time: 18913
--> 33.03547824247872%
```

## 2. sigmoid\_prime(z) und sigmoid\_vector(z)

Codeabschnitte mit Zeitmessungspunkten:

```
sgd := procedure(training_data, epochs, mini_batch_size, eta, test_data) {
  if(test_data != null) {
    n_test := #test_data;
  }
  n := #training_data;

  for(j in {0..epochs}) {
    s1 := now();
    training_data := shuffle(training_data);
    // Get mini-batches from the training data to train the network
    mini_batches := [ training_data[k..k+mini_batch_size-1] : k in [1,mini_batch_size..n] ];

    for(mini_batch in mini_batches) {
      update_mini_batch(mini_batch, eta);
    }
    epoche_time := now() - s1;
    // Visual output
    if(test_data != null) {
      ev := evaluate(test_data);
      //print("Epoch $j$: $ev$ / $n_test$");
      print("Sigmoid-time:\t" + sigmoid_time);
      print("Epoche-time:\t" + epoche_time);
      print("--> " + 100.0 * sigmoid_time/epoche_time + "%");
      this.sigmoid_time := 0;
    }
    else {
      print("Epoch $j$ complete");
    }
  }
};
```

```
// Sigmoid function for vectors
// 1.0/(1.0+np.exp(-z))
sigmoid_vector := procedure(z) {
  // z is a vector, so the function has to be used on every part of it
  s1 := now();
  res := la_vector([ 1.0/(1.0 + exp(- z[i] )) : i in [1..#z] ]);
  this.sigmoid_time += (now() - s1);
  return res;
};

// Derivative of the sigmoid function, when z is a vector
// sigmoid(z)*(1-sigmoid(z))
sigmoid_prime := procedure(z) {
  s := sigmoid_vector(z);
  s1 := now();
  res := la_matrix([ [ s[i] * (1 - s[i]) ] : i in [1..#s] ]);
  this.sigmoid_time += (now() - s1);
  return res;
};
```

Anzahl Datensätze: 10.000 Testsätze, 10.000 Trainingssätze

Rechnerdaten: Intel Core i7-4720HQ, 16GB RAM, NVIDIA GeForce GTX 960M

## Ergebnisse 2.:

```
Start SGD
Sigmoid-time: 1423
Epoche-time: 23065
--> 6.169520919141556%
Sigmoid-time: 1391
Epoche-time: 20955
--> 6.63803388212837%
Sigmoid-time: 1328
Epoche-time: 24388
--> 5.445300967689027%
Sigmoid-time: 1220
Epoche-time: 21434
--> 5.691891387515163%
Sigmoid-time: 1583
Epoche-time: 22733
--> 6.963445211806625%
Sigmoid-time: 1337
Epoche-time: 20903
--> 6.396211070181313%
```

### 3. Zeitmessung vor und nach der zip(11,12) – Anpassung

Messungspunkte:

```
sgd := procedure(training_data, epochs, mini_batch_size, eta, test_data) {
  if(test_data != null) {
    n_test := #test_data;
  }
  n := #training_data;

  for(j in {0..epochs}) {
    s1 := now();
    training_data := shuffle(training_data);
    // Get mini-batches from the training data to train the network
    mini_batches := [ training_data[k..k+mini_batch_size-1] : k in [1,mini_batch_size..n] ];

    for(mini_batch in mini_batches) {
      update_mini_batch(mini_batch, eta);
    }

    // Visual output
    if(test_data != null) {
      ev := evaluate(test_data);
      s2 := now() - s1;
      print("Epoch $j$: $ev$ / $n_test$");
      print("Time: " + s2 + "ms");
    }
    else {
      print("Epoch $j$ complete");
    }
  }
};
```

Anzahl Datensätze: 10.000 Testsätze, 10.000 Trainingssätze

Rechnerdaten: Intel Core i7-4720HQ, 16GB RAM, NVIDIA GeForce GTX 960M

### Ergebnisse 3. – Mit Zipping:

```
Start SGD
Epoch 0: 9069 / 10000
Time: 29336ms
Epoch 1: 9223 / 10000
Time: 27678ms
Epoch 2: 9288 / 10000
Time: 25479ms
Epoch 3: 9299 / 10000
Time: 25782ms
Epoch 4: 9300 / 10000
Time: 25992ms
Epoch 5: 9348 / 10000
Time: 23974ms
Epoch 6: 9328 / 10000
Time: 23415ms
Epoch 7: 9325 / 10000
Time: 23649ms
Epoch 8: 9350 / 10000
Time: 24359ms
Epoch 9: 9334 / 10000
Time: 25321ms
Epoch 10: 9328 / 10000
Time: 24775ms
```

### Ergebnisse 3. – Ohne Zipping, mit Faktorisierung:

```
Start SGD
Epoch 0: 9069 / 10000
Time: 17632ms
Epoch 1: 9223 / 10000
Time: 15750ms
Epoch 2: 9288 / 10000
Time: 15539ms
Epoch 3: 9299 / 10000
Time: 18112ms
Epoch 4: 9300 / 10000
Time: 15605ms
Epoch 5: 9348 / 10000
Time: 15181ms
Epoch 6: 9328 / 10000
Time: 17120ms
Epoch 7: 9325 / 10000
Time: 17762ms
Epoch 8: 9350 / 10000
Time: 15264ms
Epoch 9: 9334 / 10000
Time: 15402ms
Epoch 10: 9328 / 10000
Time: 16151ms
```

#### 4. Zeitliche Betrachtung diverser Funktionen.

Anzahl Datensätze: 10.000 Testsätze, 10.000 Trainingssätze

Rechnerdaten: Intel Core i7-4720HQ, 16GB RAM, NVIDIA GeForce GTX 960M

##### Ergebnisse 4. – Backprop-Funktion:

```
Start SGD
Epoch-Time: 20356ms
Backprop-Time: 15363ms
--> 75.47160542346236%
Epoch-Time: 15799ms
Backprop-Time: 11748ms
--> 74.35913665421862%
Epoch-Time: 17571ms
Backprop-Time: 11309ms
--> 64.36173239997723%
Epoch-Time: 14984ms
Backprop-Time: 11113ms
--> 74.1657768286172%
Epoch-Time: 15325ms
Backprop-Time: 11215ms
--> 73.18107667210441%
Epoch-Time: 16283ms
Backprop-Time: 11981ms
--> 73.57980716084259%
```

##### Ergebnisse 4. – getNabla b and w - Funktion:

```
Start SGD
Epoch-Time: 17427ms
Nabla-Time: 6963ms
--> 39.95524186606989%
Epoch-Time: 15131ms
Nabla -Time: 6837ms
--> 45.185381005881965%
Epoch-Time: 15853ms
Nabla -Time: 7680ms
--> 48.445089257553775%
Epoch-Time: 17673ms
Nabla -Time: 7021ms
--> 39.72726758331919%
Epoch-Time: 15527ms
Nabla -Time: 6934ms
--> 44.65769305081471%
Epoch-Time: 15058ms
Nabla -Time: 6852ms
--> 45.50405100278922%
```

5. Vergleich der la-Funktionen in SetlX mit Numpy in Python.

Rechnerdaten: Intel Core i7-4720HQ, 16GB RAM, NVIDIA GeForce GTX 960M

Dateien: la\_timing.stlx, py\_timing.py



### Versuchsaufbau:

```
1  m := la_matrix( [ [random() : j in {1..1000}] : i in {1..1000} ] );
2
3  print("Hadamard:");
4  sum := 0;
5  for(i in [1..10]) {
6      startTime := now();
7      la_hadamard(m,m);
8      endTime := now() - startTime;
9      print("$i$. Runde:\t" + endTime + "ms");
10     sum += endTime;
11 }
12 print("Durchschnitt:\t" + sum/10 * 1.0 + "ms");
13
14 print("\nMatrizen:");
15 sum := 0;
16 for(i in [1..10]) {
17     startTime := now();
18     la_matrix( m );
19     endTime := now() - startTime;
20     print("$i$. Runde:\t" + endTime + "ms");
21     sum += endTime;
22 }
23 print("Durchschnitt:\t" + sum/10 * 1.0 + "ms");
24
25 print("\nMatrizen-Multiplikation:");
26 sum := 0;
27 for(i in [1..10]) {
28     startTime := now();
29     res := m*m;
30     endTime := now() - startTime;
31     print("$i$. Runde:\t" + endTime + "ms");
32     sum += endTime;
33 }
34 print("Durchschnitt:\t" + sum/10 * 1.0 + "ms");
```

Setlx

```

1  import numpy as np
2  from time import time
3
4  m = np.random.random((1000, 1000))
5
6  print "Hadamard:"
7  mSum = 0
8  for i in range(10):
9      startTime = time()
10     res = m*m
11     endTime = time() - startTime
12     print str(i+1) + ". Runde:\t" + str(endTime * 1000) + "ms"
13     mSum += endTime * 1000
14 print "Durchschnitt:\t" + str(mSum/10) + "ms"
15
16 print "\nMatrizen:"
17 mSum = 0
18 for i in range(10):
19     startTime = time()
20     np.random.random((1000, 1000))
21     endTime = time() - startTime
22     print str(i+1) + ". Runde:\t" + str(endTime * 1000) + "ms"
23     mSum += endTime * 1000
24 print "Durchschnitt:\t" + str(mSum/10) + "ms"
25
26 print "\nMatrizen-Multiplikation:"
27 mSum = 0
28 for i in range(10):
29     startTime = time()
30     np.dot(m,m)
31     endTime = time() - startTime
32     print str(i+1) + ". Runde:\t" + str(endTime * 1000) + "ms"
33     mSum += endTime * 1000
34 print "Durchschnitt:\t" + str(mSum/10) + "ms"

```

Python

#### Ergebnisse 5. – la\_hadamard in SetlX:

1. Runde:	43ms
2. Runde:	13ms
3. Runde:	10ms
4. Runde:	9ms
5. Runde:	9ms
6. Runde:	8ms
7. Runde:	8ms
8. Runde:	8ms
9. Runde:	8ms
10. Runde:	8ms
Durchschnitt:	12.4ms

#### Ergebnisse 5. – Hadamard in Python:

1. Runde:	0.0ms
2. Runde:	0.0ms
3. Runde:	16.0000324249ms
4. Runde:	0.0ms
5. Runde:	0.0ms
6. Runde:	16.0000324249ms
7. Runde:	0.0ms
8. Runde:	15.0001049042ms
9. Runde:	16.0000324249ms
10. Runde:	0.0ms
Durchschnitt:	6.3000202179ms

Python performanter  
(ca. 2 Mal schneller)

#### Ergebnisse 5. – la\_matrix in SetlX:

1. Runde:	168ms
2. Runde:	138ms
3. Runde:	47ms
4. Runde:	38ms
5. Runde:	62ms
6. Runde:	116ms
7. Runde:	38ms
8. Runde:	16ms
9. Runde:	31ms
10. Runde:	37ms
Durchschnitt:	69.1ms

#### Ergebnisse 5. – np.zeros in Python:

1. Runde:	14.9998664856ms
2. Runde:	0.0ms
3. Runde:	51.9998073578ms
4. Runde:	27.0001888275ms
5. Runde:	20.9999084473ms
6. Runde:	16.0000324249ms
7. Runde:	16.0000324249ms
8. Runde:	3.99994850159ms
9. Runde:	63.9998912811ms
10. Runde:	68.0000782013ms
Durchschnitt:	28.2999753952ms

Python performanter  
(ca. 2,4 Mal schneller)

#### Ergebnisse 5. – Matrizen-Multiplikation in SetlX:

1. Runde:	2396ms
2. Runde:	2622ms
3. Runde:	2283ms
4. Runde:	2454ms
5. Runde:	2240ms
6. Runde:	2408ms
7. Runde:	2181ms
8. Runde:	3266ms
9. Runde:	2203ms
10. Runde:	1999ms
Durchschnitt:	2405.2ms

#### Ergebnisse 5. – Matrizen-Multiplikation in Python:

1. Runde:	411.000013351ms
2. Runde:	299.999952316ms
3. Runde:	252.00009346ms
4. Runde:	345.00002861ms
5. Runde:	401.999950409ms
6. Runde:	447.999954224ms
7. Runde:	654.99997139ms
8. Runde:	248.000144958ms
9. Runde:	283.999919891ms
10. Runde:	275.000095367ms
Durchschnitt:	362.000012398ms

Python wesentlich performanter  
(ca. 6,5 Mal schneller)

## 6. Matrizen-Multiplikation in Java mittels Jama.

Rechnerdaten: Intel Core i7-4720HQ, 16GB RAM, NVIDIA GeForce GTX 960M

Ausführung:    `javac Jama_Timing.java -classpath .`  
                  `java -cp . Jama_Timing`

### Versuchsaufbau:

```
1  import java.util.Random;
2  import Jama.*;
3
4  public class Jama_Timing
5  {
6      public static void main(String [] args)
7      {
8          double[][] array = new double[1000][1000];
9          for(int i=0; i<1000; i++) {
10             for(int j=0; j<1000; j++) {
11                 array[i][j] = new Random().nextInt();
12             }
13         }
14         Matrix M = new Matrix(array);
15
16         long sum = 0;
17         System.out.println("Matrizen-Multiplikation:");
18         for(int i=0; i<10;i++) {
19             long startTime = System.currentTimeMillis();
20             Matrix Res = M.times(M);
21             long endTime = System.currentTimeMillis() - startTime;
22             System.out.println((i+1) + ".Runde:\t" + endTime + "ms");
23             sum += endTime;
24         }
25
26         System.out.println("Durchschnitt:\t" + sum/10 + "ms");
27     }
28 }
```

Java

### Ergebnisse 6. – Matrizen-Multiplikation in Jama:

1. Runde:	1019ms
2. Runde:	1009ms
3. Runde:	1050ms
4. Runde:	1107ms
5. Runde:	1102ms
6. Runde:	1090ms
7. Runde:	1099ms
8. Runde:	1098ms
9. Runde:	1090ms
10. Runde:	1105ms
Durchschnitt:	1076ms