Back Propagation PAM

May 15, 2023

1 Imports

[1]:
```
%matplotlib inline
import numpy as np
from matplotlib import pyplot as plt
from tqdm import tqdm
```

2 Data Generation & Shaping

Below functions are used to generate the data. Since the data should be a sampled PAM-signal, there are functions for generating 4-PAM symbols (-3, -1, 1, 3), converting these to a signal using a pulse (as in 34210 assignment 2), adding noise from a normal distribution, adding a bias dimension to the input (as explained in the Back Propagation exercises), and finally for splitting the data.

[2]:
```
def get_data(L):
    symbols = [-3, -1, 1, 3]
    one_hot_symbols = {
        -3: [1, 0, 0, 0],
        -1: [0, 1, 0, 0],
        1: [0, 0, 1, 0],
        3: [0, 0, 0, 1]
    }
    input_sequence = np.random.choice(symbols, size=L)
    output_sequence = np.array([one_hot_symbols[i] for i in input_sequence])
    return input_sequence, output_sequence

def get_signal(input_data):
    a0 = np.zeros(input_data.size * (m), input_data.dtype)
    a0[::m] = input_data
    v = np.convolve(g_T, a0)
    return v

def add_noise(input_data, sigma):
    N = input_data.shape[0]
    noise = np.random.normal(loc=0, scale=sigma, size=N)
    return input_data + noise
```

```
def add_bias_dimension(input_data):
    N = input_data.shape[0]
    return np.column_stack((input_data, np.ones(N)))

def split_data(input_data, fraction):
    N = input_data.shape[0]
    part = int(fraction*N)
    return input_data[:part], input_data[part:]
```

3 Network Parameters

[22]:
```
ETA = 0.1
EPOCH = 1000
SIGMA = 1
INPUT_SIZE = 1
HIDDEN_SIZE = 7
OUTPUT_SIZE = 4

T = 1
m = 10
Ts = T/m
ti = np.arange(-4*T, 4*T, Ts)
g_T = (np.cos(2*np.pi * ti/T)) / (1- (4* ti/T)**2)
```

We generate 20000 symbols, convert them to a PAM-signal, add noise, sample the signal and add a bias dimension. This will be split in training and testing parts and used as-is. A cut-out of the sent- and received signals are shown together to illustrate how noisy the data is.

[4]:
```
L = 20000
fraction = 3/4
noise_sigma = 3

input_sequence, output_sequence = get_data(L)
v = get_signal(input_sequence)
v_noisy = add_noise(v, noise_sigma)
start = np.argmax(np.convolve(g_T,g_T))
s = np.convolve(g_T, v_noisy)
a_hat = s[start:-start:m]
input_sequence = add_bias_dimension(a_hat)
training_input, testing_input = split_data(input_sequence, fraction)
training_output, testing_output = split_data(output_sequence, fraction)
```

[5]:
```
plt.plot(v[150:450], label='Sent signal')
plt.plot(v_noisy[150:450], label='Received signal' ,alpha=0.4)
plt.legend()
plt.show()
```

4 Network Definition

Below are all the functions required to forward- and backpropagate. Just for experimenting, a combination of sigmoid, tanh, relu and softmax have been tried to see which are most effective. Only softmax has been used on the output, however, for reasons explained in the back propagation exercises. The others will be used later.

[23]:
```
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def sigmoid_derivative(x):
    y = sigmoid(x)
    return y * (1.0 - y)

def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    return 1-tanh(x)**2

def relu(x):
```

```
    return np.maximum(x, 0)

def relu_derivative(x):
    return np.sign(relu(x))

def softmax(x):
    out = np.exp(x)
    total = np.sum(out, axis=1)
    return out / total[:,None]

def forward_propagate(input_data, weights):
    node_1 = input_data @ weights[0]
    activation_1 = relu(node_1)
    node_2 = activation_1 @ weights[1]
    activation_2 = softmax(node_2)
    return activation_1, activation_2

def sum_of_squares_error(y_pred, y_true):
    return sum( (sum( (y_pred - y_true)** 2 )) / 2 )

def same_answer(pred, true):
    return 1 if np.argmax(pred) != np.argmax(true) else 0

def count_errors(pred, true):
    return sum(map(same_answer, pred, true))
```

This time, we choose random initial weights from a uniform distribution.

[24]:
```
#np.random.seed(1)
W1 = np.random.uniform(low=-1, high=1, size=(INPUT_SIZE+1, HIDDEN_SIZE+1))
W2 = np.random.uniform(low=-1, high=1, size=(HIDDEN_SIZE+1, OUTPUT_SIZE))
weights = [W1, W2]
errors = [0 for _ in range(EPOCH)]
error_counts = [0 for _ in range(EPOCH)]
```

[25]:
```
N = training_input.shape[0]
for i in tqdm(range(EPOCH)):
    hidden_layer, output_activation = forward_propagate(training_input, weights)

    errors[i] = sum_of_squares_error(output_activation, training_output)
    error_counts[i] = count_errors(output_activation, training_output)

    output_error = output_activation - training_output
    W2_change = (hidden_layer.T @ output_error) / N
    W1_change = (((output_error @ weights[1].T) \
                  * relu_derivative(hidden_layer)).T \
                  @ training_input) \
                  / N
```

```
    weights[1] = weights[1] - ETA*W2_change
    weights[0] = weights[0] - ETA*W1_change.T
```

```
100%|                | 1000/1000 [00:49<00:00, 20.12it/s]
```

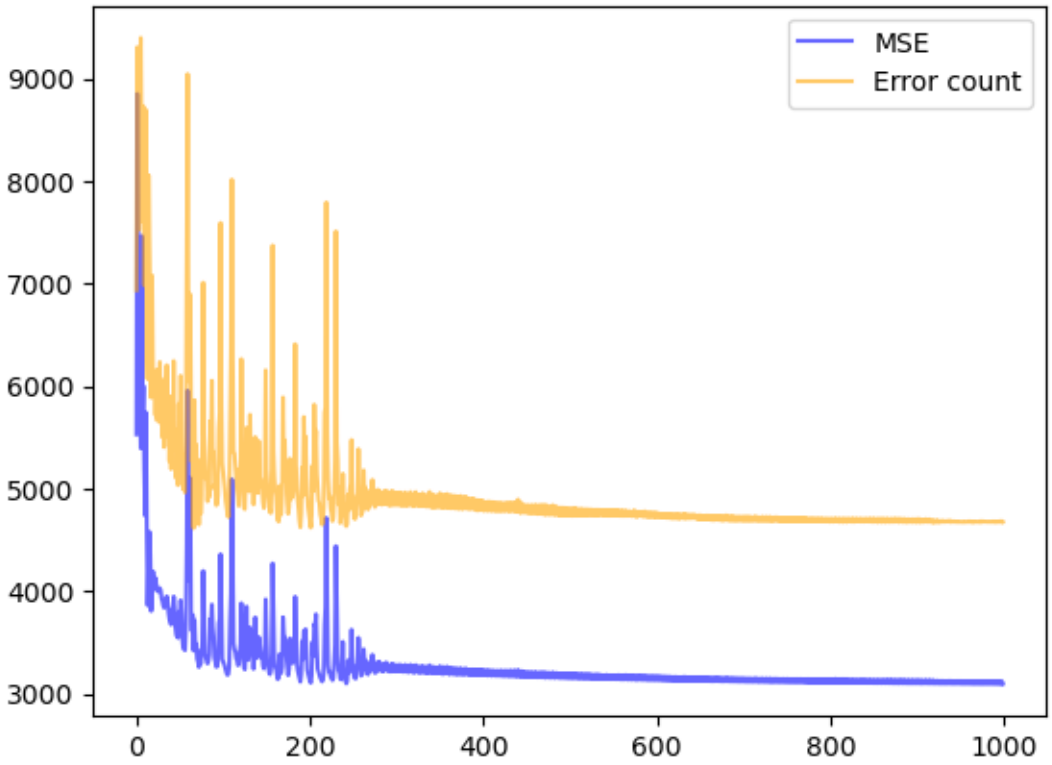
5 Testing

[26]:
```
def same_answer(pred, true):
    return 1 if np.argmax(pred) != np.argmax(true) else 0

def test_total_errors(input_data, output_data, weights):
    _, output = forward_propagate(input_data,weights)
    return sum(map(same_answer, output, output_data))
```

[27]:
```
total_errors = test_total_errors(input_data=testing_input,
                                 output_data=testing_output,
                                 weights=weights)
print(f'Total errors: {total_errors}, which is {total_errors/testing_input.
 ↪shape[0]*100:.3}%')
```

```
Total errors: 1579, which is 31.6%
```

[28]:
```
plt.plot(errors, c='b', label='MSE', alpha=0.6)
plt.plot(error_counts, c='orange', label='Error count', alpha=0.6)
plt.legend()
plt.show()
```



It is notable how the network takes a long time to stabilise. The network is almost unusable until about 250 iterations, where it then stabilises.

6 Activation functions

On the hidden layer, one could choose between a number of different functions; tanh, relu, and sigmoid just to name a few. These will now be compared to see which are most effective.

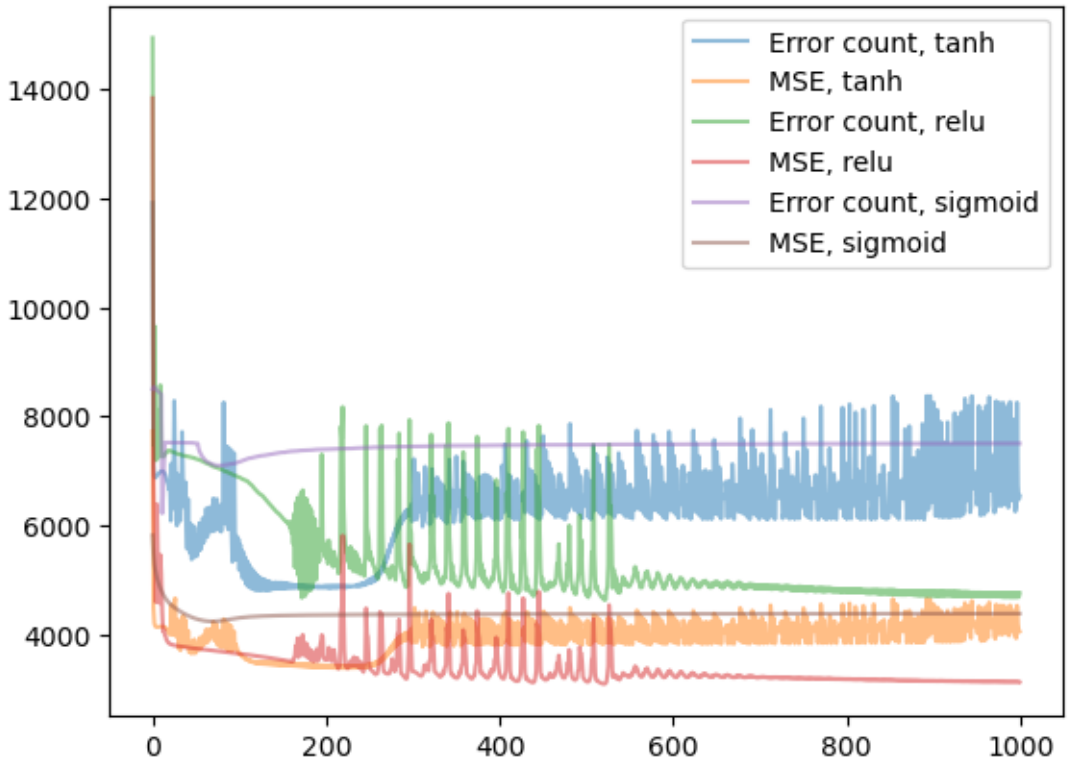
[14]:
```
functions = ((tanh,tanh_derivative), (relu,relu_derivative),␣
 ↪(sigmoid,sigmoid_derivative))
```

[15]:
```
def forward_propagate(input_data, weights, activation_function):
    node_1 = input_data @ weights[0]
    activation_1 = activation_function(node_1)
    node_2 = activation_1 @ weights[1]
    activation_2 = softmax(node_2)
    return activation_1, activation_2

function_error_counts = []
function_mses = []
function_weights = []
```

```
functions = ((tanh,tanh_derivative), (relu,relu_derivative),␣
 ↪(sigmoid,sigmoid_derivative))
for activation_function,function_derivative in functions:
    W1 = np.random.uniform(low=-1, high=1, size=(INPUT_SIZE+1, HIDDEN_SIZE+1))
    W2 = np.random.uniform(low=-1, high=1, size=(HIDDEN_SIZE+1, OUTPUT_SIZE))
    weights = [W1, W2]
    errors = [0 for _ in range(EPOCH)]
    error_counts = [0 for _ in range(EPOCH)]

    N = training_input.shape[0]
    for i in tqdm(range(EPOCH)):
        hidden_layer, output_activation = forward_propagate(training_input,␣
 ↪weights, activation_function)

        errors[i] = sum_of_squares_error(output_activation, training_output)
        error_counts[i] = count_errors(output_activation, training_output)

        output_error = output_activation - training_output
        W2_change = (hidden_layer.T @ output_error) / N
        W1_change = (((output_error @ weights[1].T) \
                      * function_derivative(hidden_layer)).T \
                      @ training_input) \
                      / N
        weights[1] = weights[1] - ETA*W2_change
        weights[0] = weights[0] - ETA*W1_change.T
    function_error_counts.append(error_counts)
    function_mses.append(errors)
    function_weights.append(weights)
```

```
100%|                | 1000/1000 [00:52<00:00, 19.03it/s]
100%|                | 1000/1000 [00:51<00:00, 19.44it/s]
 33%|                           | 334/1000 [00:17<00:35,
18.99it/s]/tmp/ipykernel_539914/2240171596.py:2: RuntimeWarning: overflow
encountered in exp
  return 1.0 / (1.0 + np.exp(-x))
100%|                | 1000/1000 [00:52<00:00, 19.12it/s]
```

[16]:
```
for e,mse,name in zip(function_error_counts, function_mses,␣
 ↪['tanh','relu','sigmoid']):
    plt.plot(e, label='Error count, '+name, alpha=0.5)
    plt.plot(mse, label='MSE, '+name, alpha=0.5)
plt.legend()
plt.show()
```



Interestingly, tanh never stabilises and is quite poor at actually guessing the correct symbol. The MSE, however, is quite stable and even lower than the sigmoid MSE. ReLu is "just better" than both of these; the lowest error count as well as MSE. As it did above, it takes some time for it to stabilise, and it never fully stops learning. As it can be seen below, it also performs better on the test set, so it is not an egregious case of overfitting, either.

[21]:
```
def test_total_errors(input_data, output_data, weights, function):
    _, output = forward_propagate(input_data,weights,function)
    return sum(map(same_answer, output, output_data))
for fs,w in zip(functions,function_weights):
    activation_function,function_derivative = fs
    total_errors = test_total_errors(input_data=testing_input,
                                     output_data=testing_output,
                                     weights=w,
                                     function=activation_function)
    print(f'{activation_function.__name__}: Total errors: {total_errors}, \
    which is {total_errors/testing_input.shape[0]*100:.3}%')
```

```
tanh: Total errors: 2534,     which is 50.7%
relu: Total errors: 1579,     which is 31.6%
sigmoid: Total errors: 2471,     which is 49.4%
```

```
/tmp/ipykernel_539914/2240171596.py:2: RuntimeWarning: overflow encountered in
```

```
    exp
      return 1.0 / (1.0 + np.exp(-x))
```

[ ]: