# Zad 08 - `Zadanie liniowe RNN`

**Temat:** Liniowe sieci rekurencyjne (RNN) i algorytm BPTT

## Treść zadania

Utworzenie sieci RNN według wariantu zadania.

**Wariant zadania: 14**

Dane wejściowe składają się z 30 sekwencji po 20 kroków czasowych każda. Każda sekwencja wejściowa jest generowana z jednolitego rozkładu losowego, który jest zaokrąglany do 0.33, 0.66 lub 1. Cele wyjściowe `t` to liczba wystąpień „0.33” w sekwencji.

### Kod Python

In [1]:
```python
import numpy as np

# 1. Generowanie danych
np.random.seed(1)
nb_of_samples = 30
sequence_len = 20

X = np.zeros((nb_of_samples, sequence_len))
for row_idx in range(nb_of_samples):
    # Losowe wartości z [0,1) i zaokrąglenie do 0.33, 0.66, 1
    rand_vals = np.random.rand(sequence_len)
    rounded = np.zeros_like(rand_vals)
    for i, val in enumerate(rand_vals):
        if val < 0.33:
            rounded[i] = 0.33
        elif val < 0.66:
            rounded[i] = 0.66
        else:
            rounded[i] = 1.0
    X[row_idx, :] = rounded

# Cele: liczba wystąpień 0.33 w każdej sekwencji
t = np.sum(X == 0.33, axis=1).astype(float)
```

In [2]:
```python
# 2. Definicje funkcji modelu i gradientów
def update_state(xk, sk, wx, wRec):
    return xk * wx + sk * wRec

def forward_states(X, wx, wRec):
    S = np.zeros((X.shape[0], X.shape[1] + 1))
    for k in range(X.shape[1]):
        S[:, k + 1] = update_state(X[:, k], S[:, k], wx, wRec)
    return S

def loss(y, t):
```

```
    return np.mean((t - y) ** 2)

def output_gradient(y, t):
    return 2.0 * (y - t)

def backward_gradient(X, S, grad_out, wRec):
    grad_over_time = np.zeros((X.shape[0], X.shape[1] + 1))
    grad_over_time[:, -1] = grad_out
    wx_grad = 0
    wRec_grad = 0
    for k in range(X.shape[1], 0, -1):
        wx_grad += np.sum(grad_over_time[:, k] * X[:, k-1])
        wRec_grad += np.sum(grad_over_time[:, k] * S[:, k-1])
        grad_over_time[:, k-1] = grad_over_time[:, k] * wRec
    return (wx_grad, wRec_grad), grad_over_time
```

In [3]:
```
# 3. Funkcja aktualizacji RProp
def update_rprop(X, t, W, W_prev_sign, W_delta, eta_p, eta_n):
    S = forward_states(X, W[0], W[1])
    grad_out = output_gradient(S[:, -1], t)
    W_grads, _ = backward_gradient(X, S, grad_out, W[1])
    W_sign = np.sign(W_grads)
    for i in range(len(W)):
        if W_sign[i] == W_prev_sign[i]:
            W_delta[i] *= eta_p
        else:
            W_delta[i] *= eta_n
    return W_delta, W_sign

# 4. Inicjalizacja wag i hiperparametrów RProp
eta_p = 1.2
eta_n = 0.5
W = [-1.5, 2.0]  # [wx, wRec]
W_delta = [0.001, 0.001]
W_sign = [0, 0]
ls_of_ws = [tuple(W)]

# 5. Pętla treningowa (RProp)
for iteration in range(500):
    W_delta, W_sign = update_rprop(X, t, W, W_sign, W_delta, eta_p, eta_n)
    for i in range(len(W)):
        W[i] -= W_sign[i] * W_delta[i]
    ls_of_ws.append(tuple(W))

print(f'Final weights: wx = {W[0]:.4f}, wRec = {W[1]:.4f}')
```

```
Final weights: wx = 1.7264, wRec = 0.8206
```

In [4]:
```
# 6. Test na przykładowej sekwencji
test_seq = np.asmatrix([[0.33, 0.66, 0.33, 1.0, 0.33, 0.66, 0.66, 1.0, 0.33, 0.3
                         0.66, 1.0, 0.33, 0.66, 0.33, 1.0, 0.66, 0.33, 0.66, 0.3
test_output = forward_states(test_seq, W[0], W[1])[:, -1]
target = np.sum(test_seq == 0.33)
print(f'Test sequence: {test_seq.tolist()[0]}')
print(f'Target count of 0.33: {target}')
print(f'Model output: {test_output[0]:.2f}')
```

```
Test sequence: [0.33, 0.66, 0.33, 1.0, 0.33, 0.66, 0.66, 1.0, 0.33, 0.33, 0.66, 1
.0, 0.33, 0.66, 0.33, 1.0, 0.66, 0.33, 0.66, 0.33]
Target count of 0.33: 9
Model output: 5.18
```