ДОДАТОК А

Вихідний код програми для імітаційного моделювання

```
class CanNet(nn.Module):

    def __init__(self, signals_per_id, h_scale):
        """
        Model architecture constructor
        """
        super().__init__()

        self.h_scale = h_scale  # h_scale = 5
        self.signals_per_id = signals_per_id  # signals_per_id = [2,
3, 2, 1, 2, 2, 2, 1, 1, 4]

        self.hidden_states = self.init_hidden()  # init with zeros
        self.cell_states = self.init_cells()     #

# ===================================layers
generation=============================================================
======

        lstm_heads = [nn.LSTM(signal, signal * self.h_scale) for
signal in self.signals_per_id]

        fully_connected = [nn.Linear(sum(self.signals_per_id) *
self.h_scale,  # Fully Connected 1
                                     sum(self.signals_per_id) *
self.h_scale // 2),

                           nn.Linear(sum(self.signals_per_id) *
self.h_scale // 2,  # Fully Connected 2
                                     sum(self.signals_per_id) - 1)]

        reconstructions = [nn.Linear(sum(self.signals_per_id) - 1,
signal) for signal in self.signals_per_id]  # generate 10
reconstructions

        self.layers =
nn.Sequential(*(lstm_heads+fully_connected+reconstructions))  #
stack it up

    def forward(self, x, i):
        """
        @brief: performs corresponding model branch forward pass
        @param x: input. torch tensor with shape of (1, 1,
payload_size)
        @param i: model id
        @return x: result of model forward pass. Reconstruction of
the input signal
        """
        x, states = self.layers[i](x, (self.hidden_states[i],
self.cell_states[i]))  # forward on corresponding LSTM
        self.hidden_states[i] = x  # update corresponding LSTM`s
hidden state
```

```
        self.cell_states[i] = states[1]
        x = torch.cat(self.hidden_states, dim=2)  # joint latent
vector
        x = F.elu(self.layers[10](x))  # forward on Fully Connected
1
        x = F.elu(self.layers[11](x))  # forward on Fully Connected
2
        x = F.elu(self.layers[12 + i](x))  # forward on
corresponding reconstruction

        return x

    def init_hidden(self):
        """
        @brief: Initialize each LSTM-head hidden states with zeros
        @return: list of zero-like tensors corresponding to each
LSTM
        """
        return [torch.zeros(1, 1, i * self.h_scale,
dtype=torch.float32) for i in self.signals_per_id]

    def init_cells(self):
        """
        @brief: Initialize each LSTM-head cell states with zeros
        @return: list of zero-like tensors corresponding to each
LSTM
        """
        return [torch.zeros(1, 1, i * self.h_scale,
dtype=torch.float32) for i in self.signals_per_id]

class TrainingModule:

    def __init__(self, signals_per_id, h_scale, epochs, batch_size,
update_interval, segment_length, dataset_inv_freqs, save_interval,
save_weights_path):
        """
        Training module constructor
        """
        self.signals_per_id = signals_per_id
        self.h_scale = h_scale
        self.epochs = epochs
        self.batch_size = batch_size
        self.update_interval = update_interval
        self.segment_length = segment_length

        # calculated as number of each signal occurrences divided by
total number of records and substracted from 1
        # so it`s linearly smaller
        self._dataset_inv_freqs = dataset_inv_freqs
        self.save_weights_path = save_weights_path

        self.main_model = None
        self.loss = None
        self.optimizer = None
        self._save_interval = save_interval
        self.generator = DataGenerator(batch_size=self.batch_size,
segment_length=self.segment_length,
```

```
max_payload_length=max(self.signals_per_id))  # module for randomly
starting batch generation

        self.saved_hidden = self.init_saved_hidden()
        self.saved_cells = self.init_saved_cells()

    def init_saved_hidden(self):
        """
        @brief: Initialize each LSTM-head hidden states with zeros
for each batch
        @return: list of zero-like tensors corresponding to each
LSTM for each batch
        """
        return [[torch.zeros(1, 1, i * self.h_scale,
dtype=torch.float32) for i in self.signals_per_id]] * 25

    def init_saved_cells(self):
        """
        @brief: Initialize each LSTM-head cell states with zeros for
each batch
        @return: list of zero-like tensors corresponding to each
LSTM for each batch
        """
        return [[torch.zeros(1, 1, i * self.h_scale,
dtype=torch.float32) for i in self.signals_per_id]] * 25

    def setup(self):
        """
        @brief: setup training configuration
        @return: None
        """
        self.main_model = CanNet(self.signals_per_id, self.h_scale)
        self.loss = nn.MSELoss(reduction='sum')
        self.optimizer =
torch.optim.Adam(self.main_model.parameters(), lr=0.01)
        # Not sure if max_decay_steps=200, end_learning_rate=0.0001
are the "best" parameters
        #self.lr_scheduler = PolynomialLRDecay(self.optimizer,
max_decay_steps=400, end_learning_rate=0.00001, power=2.0)
        self.lr_scheduler =
torch.optim.lr_scheduler.ExponentialLR(self.optimizer, gamma=0.99)

#==================only calculate gradients of corresponding LSTM
and reconstruction layers============================
    def freeze_layers(self):
        """
        @brief: disables all model parameters while calculating
gradient
        @return: None
        """
        for param in self.main_model.parameters():
            param.requires_grad = False

    def unfreeze_layers(self, signal_id):
        """
        @brief: enables parameters of a certain LSTM and
corresponding layers
```

```
        @return: None
        """
        for param in self.main_model.layers[signal_id].parameters():
# choose corresponding LSTM
            param.requires_grad = True
        for param in self.main_model.layers[10].parameters():  # 1st
fully connected of size N*h_scale/2
            param.requires_grad = True
        for param in self.main_model.layers[11].parameters():  # 2nd
fully connected of size N-1
            param.requires_grad = True
        for param in self.main_model.layers[12 +
signal_id].parameters():  # choose corresponding reconstruction
            param.requires_grad = True
#
================================================================
================================================
    def training_procedure(self):
        """
        @brief: method that runs training procedure
        @return: None
        """
        for epoch in range(self.epochs):
            print('Epoch: ', epoch+1)

            self.main_model.hidden_states =
self.main_model.init_hidden()  # zero states every new epoch
            self.main_model.cell_states =
self.main_model.init_cells()     #

            self.main_model.zero_grad()

            batch = self.generator.get_sample_batch()

            epoch_avg_loss = 0

            for iteration in
range(self.segment_length//self.update_interval):  # sequence
length//number of iterations

                iteration_loss = 0
                for i, batch_el in enumerate(batch):

                    self.main_model.hidden_states =
self.saved_hidden[i]
                    self.main_model.cell_states =
self.saved_cells[i]

                    for j, input_signal in
enumerate(batch_el[iteration * self.update_interval:
                    iteration * self.update_interval +
self.update_interval]):

                        self.freeze_layers()  # freeze every layer

                        signal_id = int(input_signal[0] - 1)  # get
arrived signal_id
```

```
                        self.unfreeze_layers(signal_id)  # freeze
everything not related to current signal_id
                        payload_size =
self.signals_per_id[signal_id]  # how much elements in current
signal_id

                        input_s =
torch.as_tensor(np.array(input_signal[1:1+payload_size],
dtype=np.float32).
                                                        reshape(1, 1,
payload_size)).float()

                        reconstruction = self.main_model(input_s,
signal_id)  # forward

                        loss_value = self.loss(reconstruction,
input_s) \
                                               *
self._dataset_inv_freqs[signal_id]  # multiply by signals inverse
freq.

                        loss_value.backward()  # accumulate grads
                        iteration_loss += loss_value.item()
                    # exclude states from the computation graph

                        self.main_model.hidden_states[signal_id] =
self.main_model.hidden_states[signal_id].detach_()
                        self.main_model.cell_states[signal_id] =
self.main_model.cell_states[signal_id].detach_()

                    self.saved_hidden[i] =
self.main_model.hidden_states
                    self.saved_cells[i] =
self.main_model.cell_states

                print(f'Iteration {iteration} average loss: ',
iteration_loss/(self.batch_size*self.update_interval))

                for param in self.main_model.parameters():
                    param.grad /= self.batch_size
                self.optimizer.step()
                self.main_model.zero_grad()

                epoch_avg_loss +=
iteration_loss/(self.batch_size*self.update_interval)
            self.lr_scheduler.step()

            print(f'EPOCH_{epoch+1}_AVG_LOSS: ',
epoch_avg_loss/(self.segment_length//self.update_interval))

            with open('loss_history_minibatch_exp', 'a') as
opened_file:

opened_file.write(f'{epoch_avg_loss/(self.segment_length//self.updat
e_interval)}\n')  # 20 = 5000/250

            if epoch % self._save_interval == 0:
```

```
                torch.save(self.main_model.state_dict(),
self.save_weights_path+'canet_epoch-{}.pt'.format(epoch))
```

ДОДАТОК Б

Відомість атестаційної роботи магістра

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	77 с.
	Інші документи	
2.	Презентаційні матеріали	20 плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Остапенко М.О			Модуль глибинного навчання для виявлення аномалій у сигналах з CAN шини автомобіля	Шифр групи	Код напр./спец.
Перевірив		Узлов Д.Ю.				СШІм-19-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					