In [1]:

```
import os
import numpy as np
import torch as T
import torch.nn as nn
import torch.optim as optim
from torch.distributions.normal import Normal
from torch.nn import functional as F
import matplotlib.pyplot as plt
import gym
```

```
D:\miniconda3\envs\cmpe260\lib\site-packages\tqdm\auto.py:22: TqdmWarning: IProgress not foun
d. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user
_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [2]:

```
import a3_gym_env
gym.make('Pendulum-v260')
```

Out[2]:

```
<OrderEnforcing<PassiveEnvChecker<CustomPendulumEnv<Pendulum-v260>>>>
```

In [3]:

```
class ReplayBuffer:
    def __init__(self):
        self.states = []
        self.probs = []
        self.vals = []
        self.actions = []
        self.rewards = []
        self.dones = []
        self.h_prevs = []

    def store_memory(self, state, action, probs, vals, reward, done, h_prevs):
        self.states.append(state)
        self.actions.append(action)
        self.probs.append(probs)
        self.vals.append(vals)
        self.rewards.append(reward)
        self.dones.append(done)
        self.h_prevs.append(h_prevs)
```

In [4]:

```
class ActorNetwork(nn.Module):
    def __init__(self, input_dims, output_dims, alpha,
                 hidden_layers = 2, hidden_dims=8, N=1):
        super().__init__()
        # actor network to estimate mean of the gaussian distribution
        self.hidden_dims = hidden_dims
        self.lin1 = nn.Linear(input_dims, hidden_dims)
        self.lstm = nn.LSTM(hidden_dims, hidden_dims)
        self.lin2 = nn.Linear(hidden_dims, output_dims)

        # use constant standard deviation
        log_std = -0.9 * np.ones(output_dims, dtype=np.float32)
        self.log_std = T.nn.Parameter(T.as_tensor(log_std))

        self.optimizer = optim.Adam(self.parameters(), lr=alpha/N, eps=1e-05)
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, state):
        obs, h_prev = state
        m = F.relu(self.lin1(obs))
        m = m.view(-1, 1, self.hidden_dims)
        m, h_next = self.lstm(m, h_prev)
        m = self.lin2(m)
#          m = outputs[-1]
        # return normal distribution
        dist = Normal(m, T.exp(self.log_std))
        return dist, h_next
```

In [5]:

```
class CriticNetwork(nn.Module):
    def __init__(self, input_dims, alpha, hidden_layers = 2, hidden_dims=8, N=1):
        super().__init__()
        # crtic network for estimating value
        self.hidden_dims = hidden_dims
        self.lin1 = nn.Linear(input_dims, hidden_dims)
        self.lstm = nn.LSTM(hidden_dims, hidden_dims)
        self.lin2 = nn.Linear(hidden_dims, 1)

        self.optimizer = optim.Adam(self.parameters(), lr=alpha/N, eps=1e-05)
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, state):
        # return value
        obs, h_prev = state
#          import pdb; pdb.set_trace()
        value = F.relu(self.lin1(obs))
        value = value.view(-1, 1, self.hidden_dims)
        value, _ = self.lstm(value, h_prev)
        value = self.lin2(value)
        return value
```

In [6]:

```
def choose_action_value(observation, h_prev, actor, critic):
        # convert state to tensor
        state = T.tensor([observation], dtype=T.float).to(actor.device)
        # get gaussian distribution for state
        dist, h_next = actor((state, h_prev))
        # get value for current state
        value = critic((state, h_prev))
        # sample action
        action = dist.sample()

        # get log prob for sampled action (to be used for ratio)
        probs = T.squeeze(dist.log_prob(action)).item()
        action = T.squeeze(action).item()
        value = T.squeeze(value).item()
        # return action, log prob for sampled action and value for current state
        return action, probs, value, h_next
```

In [7]:

```
def get_advantages(gamma, lmbda, values, dones, rewards,
                   next_value, next_done, use_gae):
    num_steps = len(rewards)
    if use_gae:
        advantages = np.zeros_like(rewards)
        last_gae_lam = 0
        for t in reversed(range(num_steps)):
            if t == num_steps - 1:
                next_non_terminal = 1.0 - next_done
                next_values = next_value
            else:
                next_non_terminal = 1.0 - dones[t + 1]
                next_values = values[t + 1]
            delta = rewards[t] + gamma * next_values * next_non_terminal - values[t]
            advantages[t] = last_gae_lam = delta + gamma * lmbda * next_non_terminal * last_ga

        returns = advantages + values
    else:
        returns = np.zeros_like(rewards)
        for t in reversed(range(num_steps)):
            if t == num_steps - 1:
                next_non_terminal = 1.0 - next_done
                next_return = next_value
            else:
                next_non_terminal = 1.0 - dones[t + 1]
                next_return = returns[t + 1]
            returns[t] = rewards[t] + gamma * next_non_terminal * next_return
        advantages = returns - values
    return returns, advantages
```

In [8]:

```python
def combine_trajectories(trajectories):
    c_states = []
    c_actions = []
    c_rewards = []
    c_logprobs = []
    c_values = []
    c_dones = []
    c_returns = []
    c_advantages = []
    c_h_prevs = []
    for k, v in trajectories.items():
        buf, ret, adv = v
        c_states.extend(buf.states)
        c_actions.extend(buf.actions)
        c_rewards.extend(buf.rewards)
        c_logprobs.extend(buf.probs)
        c_values.extend(buf.vals)
        c_dones.extend(buf.dones)
        c_returns.extend(ret)
        c_advantages.extend(adv)
        c_h_prevs.extend(buf.h_prevs)
    return {
        'states': c_states,
        'actions': c_actions,
        'rewards': c_rewards,
        'logprobs': c_logprobs,
        'values': c_values,
        'dones': c_dones,
        'returns': c_returns,
        'advantages': c_advantages,
        'h_prevs': c_h_prevs,
    }
```

In [9]:

```python
def ppo(N = 1, M = 50, max_trajectory_len = 200, batch_size = 25,
        alpha = 0.0001, hidden_layers = 2, hidden_dims = 8, gamma = 0.99,
        lmbda = 0.95, clip_value = 0.2, use_gae=True, use_clip=True):
    # N = 1 # number of times to collect new trajectories and update actor
    # M = 50 # num of trajectories
    # max_trajectory_len = 200 # trajectory length
    # batch_size = 25 # size for minibatch
    # n_epochs = 30 # number of epochs to optimize loss
    # alpha = 0.00005
    # hidden_layers = 3
    # hidden_dims = 10
    # gamma = 0.8
    # lmbda = 0.95
    # clip_value = 0.2
    # use_gae = True
    # use_clip = True

    env = gym.make('Pendulum-v260')
#     input_dim = env.observation_space.shape
    input_dim = 1
    actor = ActorNetwork(input_dims=input_dim, output_dims=1, alpha=alpha,
                        hidden_layers=hidden_layers, hidden_dims=hidden_dims)
    critic = CriticNetwork(input_dims=input_dim, alpha=alpha,
                        hidden_layers=hidden_layers, hidden_dims=hidden_dims)
    trajectories = {}
    total_loss_list = []
    loss_list = []
    reward_list = []
    observation_list = []
    for n in range(N):
        # collect M trajectories, their returns, and activations
        for i in range(M):
            curr_trajectory = ReplayBuffer()
            # hprev and cell state
            h_prev = (T.zeros([1, 1, hidden_dims], dtype=T.float).to(actor.device),
                      T.zeros([1, 1, hidden_dims], dtype=T.float).to(actor.device))
            observation = env.reset()
            # only use angular velocity
            observation = observation[-1]
            done = False
            steps = 0
            while steps < max_trajectory_len:
                action, prob, val, h_next = choose_action_value(
                    observation, h_prev, actor, critic)
                next_observation, reward, done, info = env.step([action])
                steps +=1
                curr_trajectory.store_memory(
                    observation, action, prob, val, reward, done, h_prev)
                observation = next_observation[-1]
                h_prev = h_next
                if done:
                    break
            # get GAE advantage
            # get value of next_observation
            next_value = critic((T.tensor([observation]).to(actor.device), h_prev))
            # calculate return and advantage
            returns, advantages = get_advantages(
                gamma, lmbda, curr_trajectory.vals, curr_trajectory.dones,
                curr_trajectory.rewards, next_value, done, use_gae)
            trajectories[i] = (curr_trajectory, returns, advantages)

        # combine_trajectories
        c_trajectories = combine_trajectories(trajectories)
        total_steps_taken = len(c_trajectories['states'])
        indicies = np.arange(total_steps_taken)
        np.random.shuffle(indicies)
        np.random.shuffle(indicies)

        # optimize loss
        for epoch in range(n_epochs):
            # get batches
            for start in range(0, total_steps_taken, batch_size):
                end = start + batch_size
                # batch indicies
                b_indicies = indicies[start:end]
                new_value = []
                distr = []
                for index in b_indicies:
                    state = T.tensor([c_trajectories['states'][index]]).to(actor.device)
                    hprev = c_trajectories['h_prevs'][index]
                    new_value.append(critic((state, hprev)))
                    distr.append(actor((state, hprev)))

                new_value = T.tensor(new_value).to(actor.device)
#                 states = T.tensor(
#                     np.array(c_trajectories['states'])[b_indicies],
#                     dtype=T.float).to(actor.device)
#                 import pdb; pdb.set_trace()
#                 h_prevs = c_trajectories['h_prevs'][b_indicies]
#                 new_value = critic((states, h_prevs))
#                 distr = actor((states, h_prevs))

                old_log_prob = T.tensor(
                    np.array(c_trajectories['logprobs'])[b_indicies]).to(actor.device)
                actions = T.tensor(
                    np.array(c_trajectories['actions'])[b_indicies],
                    dtype=T.float).to(actor.device)
                new_log_prob = T.tensor([d[0].log_prob(actions[i]) for i, d in enumerate(distr
                    dtype=T.float).to(actor.device)

                # Probability ratio
                log_ratio = new_log_prob - old_log_prob
                ratio = log_ratio.exp()

                b_returns = T.tensor(
                    np.array(c_trajectories['returns'])[b_indicies]).to(actor.device)
                b_advantages = T.tensor(
                    np.array(c_trajectories['advantages'])[b_indicies]).to(actor.device)
#                 b_advantages = (b_advantages-b_advantages.mean())/(b_advantages.std()+1e-8)
                # Clipping
                weighted_probs = b_advantages * ratio
                if use_clip:
                    weighted_clipped_probs = T.clamp(ratio, 1-clip_value,
                            1+clip_value)*b_advantages
                    actor_loss = -T.min(
                        weighted_probs, weighted_clipped_probs).mean()
                else:
                    weighted_probs = b_advantages * ratio
                    actor_loss = -(weighted_probs).mean()
                critic_loss = (b_returns-new_value)**2
                critic_loss = critic_loss.mean()

                total_loss = actor_loss + 0.5*critic_loss
                total_loss.requires_grad = True
                print("n: {}, epoch: {}, minibatch no.: {},".format(
                    n, epoch, start//batch_size))
                print("total_loss: {}, actor_loss: {}, critic_loss: {}".format(
                    total_loss, actor_loss, critic_loss))

                total_loss_list.append(total_loss)
                actor.optimizer.zero_grad()
                critic.optimizer.zero_grad()
                total_loss.backward()
                total_loss.backward()
                actor.optimizer.step()
                critic.optimizer.step()
#              actor.scheduler.step(actor_loss)
#              critic.scheduler.step(critic_loss)

        # get a single trajectory from actor for plotting
        loss_list.append(total_loss)
        reward_list.append(b_returns.mean())

        observation = env.reset()
        h_prev = (T.zeros([1, 1, hidden_dims], dtype=T.float).to(actor.device),
                  T.zeros([1, 1, hidden_dims], dtype=T.float).to(actor.device))
        obs = observation[-1]
        done = False
        steps = 0
        while steps < 200:
            action, prob, val, h_next = choose_action_value(
                obs, h_prev, actor, critic)
            next_observation, reward, done, info = env.step([action])
            observation_list.append(observation)
            observation = next_observation
            obs = observation[-1]
            steps += 1
            h_prev = h_next
            if done:
                break

    return total_loss_list, observation_list, loss_list, reward_list
```

Clipped and GAE

In [10]:

```
N = 100 # number of times to collect new trajectories and update actor
M = 1 # num of trajectories
max_trajectory_len = 200 # trajectory length
batch_size = 25 # size for minibatch
n_epochs = 30 # number of epochs to optimize loss
alpha = 0.0001
hidden_layers = 2
hidden_dims = 64
gamma = 0.7
lmbda = 0.9
clip_value = 0.1
use_gae = True
use_clip = True

total_losses, observations, loss_list, reward_list = ppo(
    N, M, max_trajectory_len, batch_size, alpha, hidden_layers, hidden_dims,
    gamma, lmbda, clip_value, use_gae, use_clip)
```

```
D:\miniconda3\envs\cmpe260\lib\site-packages\gym\utils\passive_env_checker.py:195: UserWarnin
g: WARN: The result returned by `env.reset()` was not a tuple of the form `(obs, info)`, wher
e `obs` is a observation and `info` is a dictionary containing additional information. Actual
type: `<class 'numpy.ndarray'>`
  logger.warn(
D:\miniconda3\envs\cmpe260\lib\site-packages\gym\utils\passive_env_checker.py:219: Deprecatio
nWarning: WARN: Core environment is written in old step API which returns one bool instead of
two. It is recommended to rewrite the environment with new step API.
  logger.deprecation(
```

In [11]:

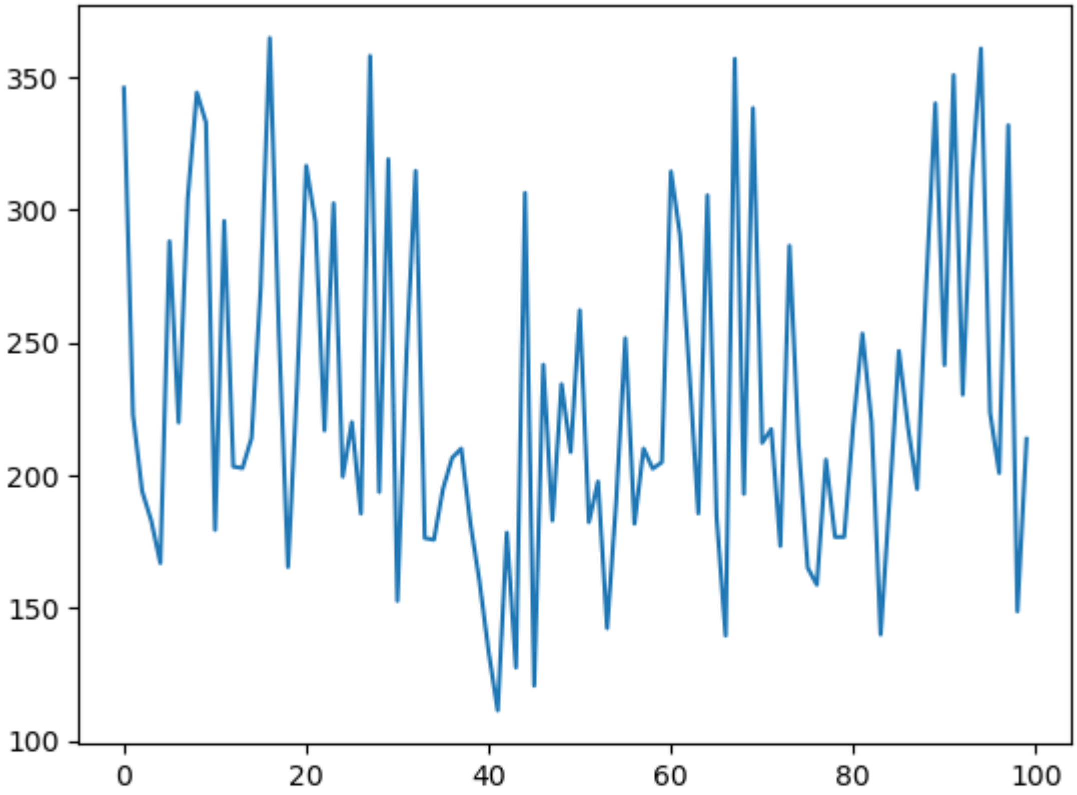
```
tmp = [x for x in range(100)]
```

In [12]:

```
plt.plot(tmp, np.array([l.cpu().detach() for l in loss_list]))
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x2508a053b80>]
```

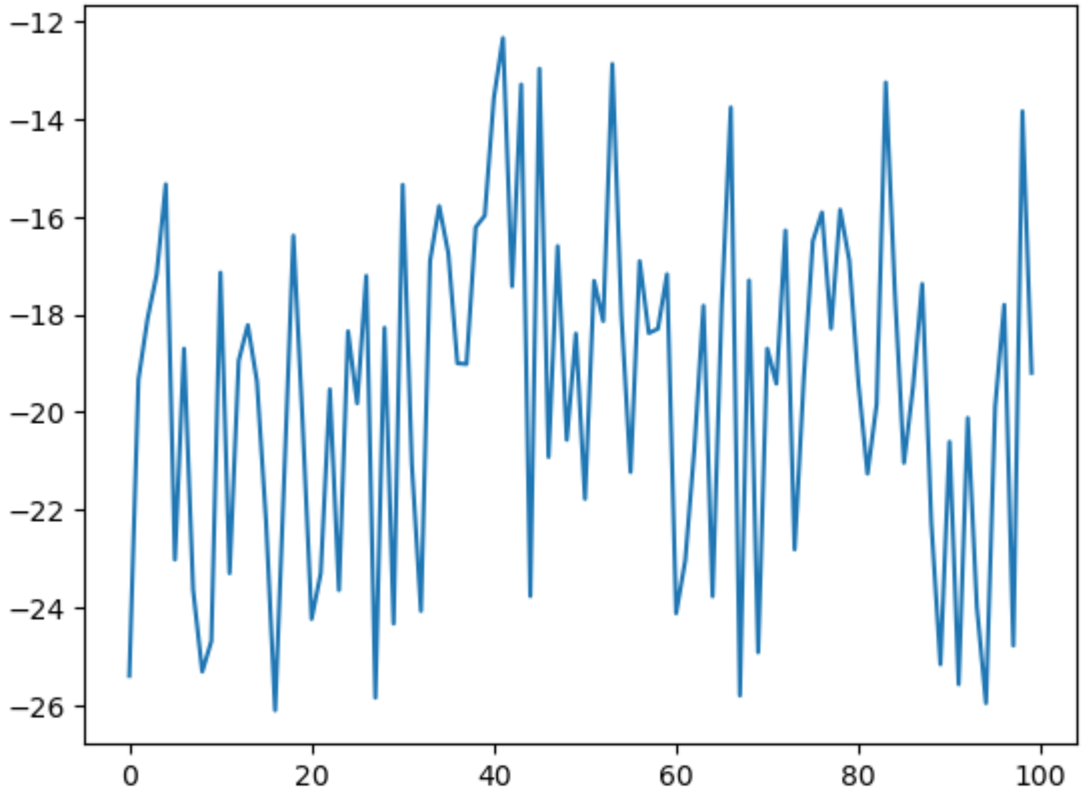


In [13]:

```python
plt.plot(tmp, np.array([l.cpu().detach() for l in reward_list]))
```

Out[13]:

```
[<matplotlib.lines.Line2D at 0x25105e1c4f0>]
```



In [14]:

```python
o = []
for i in range(N):
    o.append(observations[200*(i+1)-200:200*(i+1)])
```

In [15]:

```python
o_old = o.copy()
o = [j for i, j in enumerate(o) if i%10==0]
```

In [16]:

```python
o = o_old[-10:]
```

In [17]:

```python
x_ = [i+1 for i in range(200)]
```

In [18]:

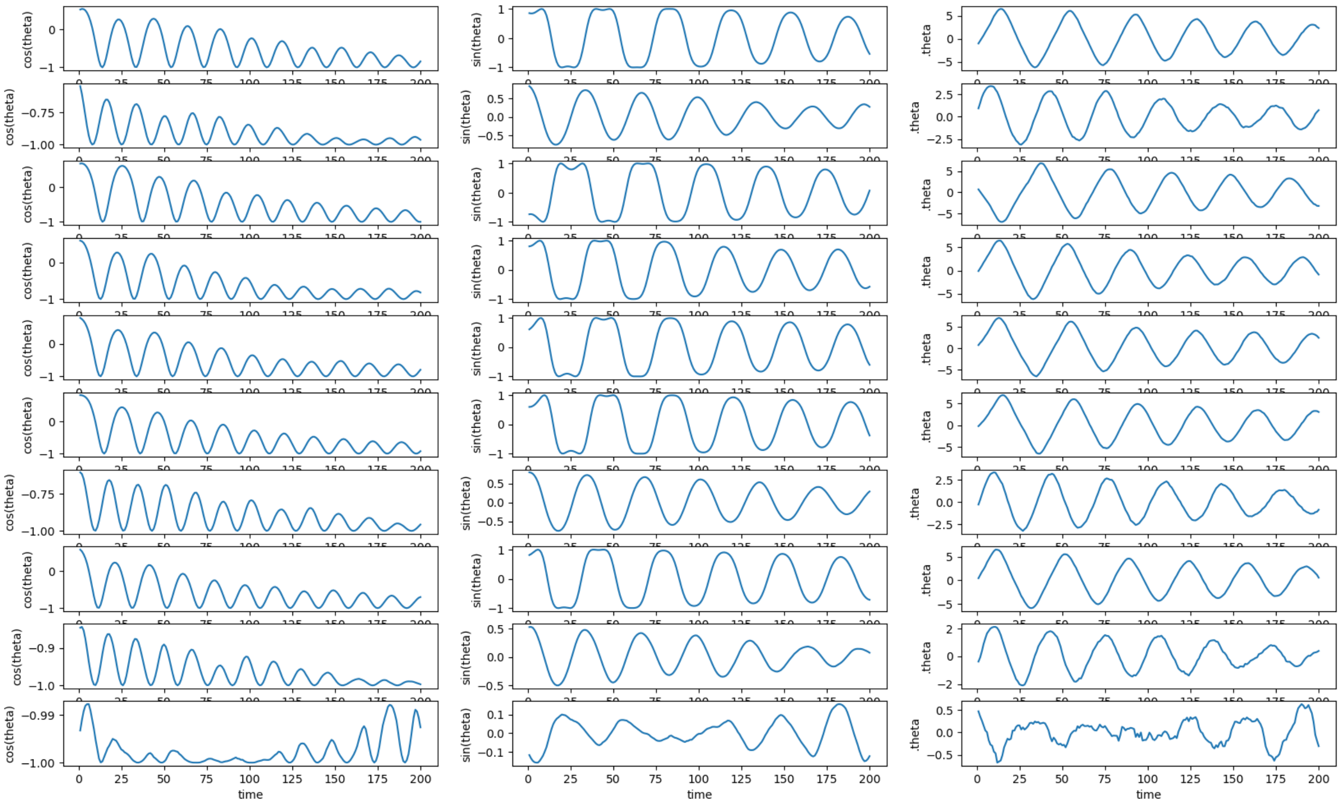
```python
np.array(o).shape
```

Out[18]:

```
(10, 200, 3)
```

In [19]:

```python
N = 10
plt.figure(1, figsize=(20, 12))
for i in range(N):
    obs = o[i]
    # print(obs)
    theta_ = []
    x_list = []
    y_list = []
    for x,y,v in obs:
        # print(x, y)
        theta_.append(v)
        x_list.append(x)
        y_list.append(y)
    plt.subplot(N, 3, 3*i+1)
    plt.plot(x_, x_list)
    plt.xlabel('time')
    plt.ylabel('cos(theta)')
    plt.subplot(N, 3, 3*i+2)
    plt.plot(x_, y_list)
    plt.xlabel('time')
    plt.ylabel('sin(theta)')
    plt.subplot(N, 3, 3*i+3)
    plt.plot(x_, theta_)
    plt.xlabel('time')
    plt.ylabel('.theta')
```



In [20]:

```python
len(total_losses)
```

Out[20]:

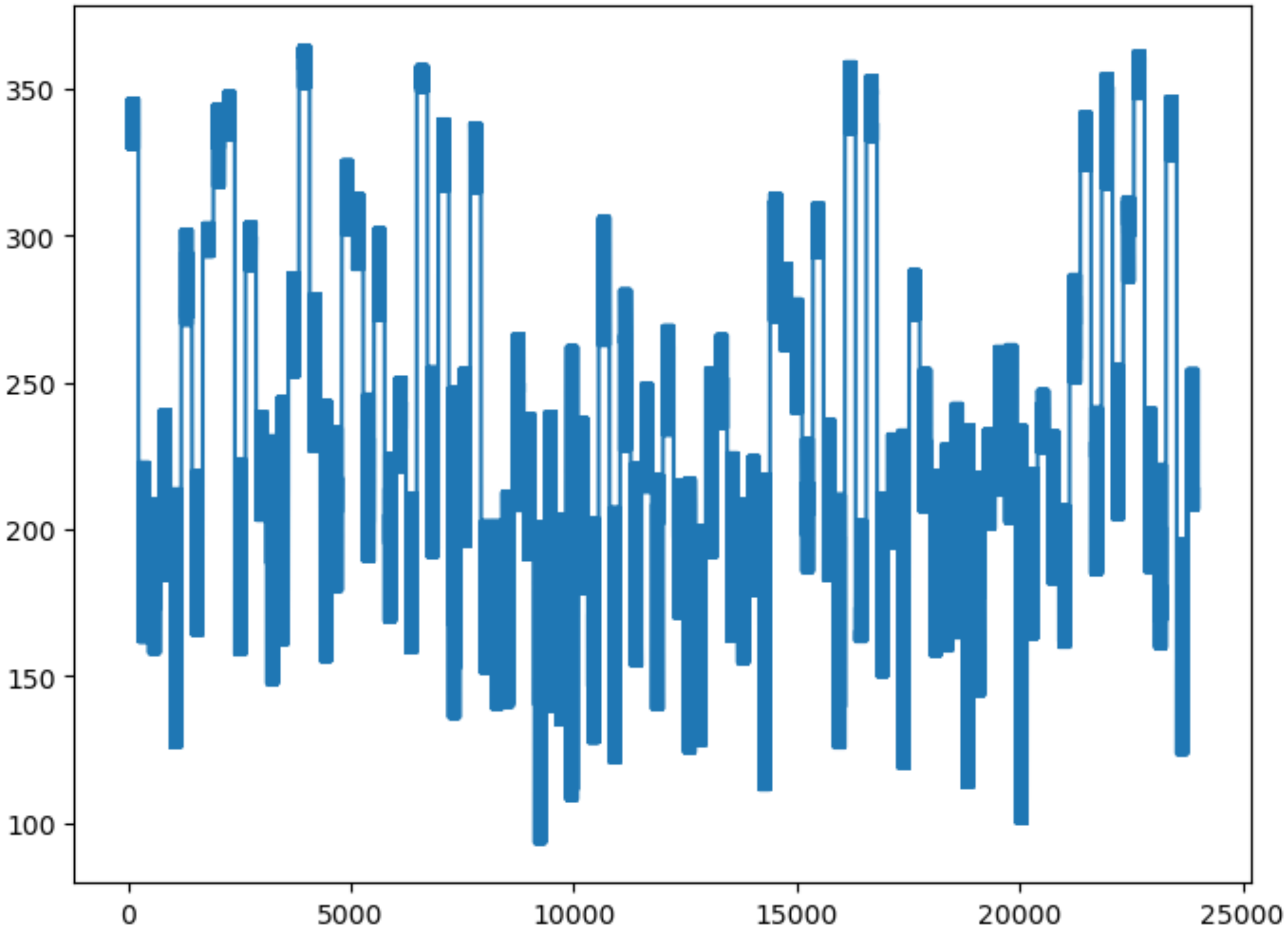
```
24000
```

In [21]:

```
x = np.arange(len(total_losses))
y = [float(x) for x in total_losses]
plt.figure(figsize=(8, 6))
plt.plot(x, y)
```

Out[21]:

```
[<matplotlib.lines.Line2D at 0x25139988790>]
```



Clipped and No GAE

In [22]:

```
N = 100 # number of times to collect new trajectories and update actor
M = 1 # num of trajectories
max_trajectory_len = 200 # trajectory length
batch_size = 25 # size for minibatch
n_epochs = 30 # number of epochs to optimize loss
alpha = 0.0001
hidden_layers = 2
hidden_dims = 64
gamma = 0.7
lmbda = 0.9
clip_value = 0.1
use_gae = False
use_clip = True

total_losses, observations, loss_list, reward_list = ppo(
    N, M, max_trajectory_len, batch_size, alpha, hidden_layers, hidden_dims,
    gamma, lmbda, clip_value, use_gae, use_clip)
```

In [23]:

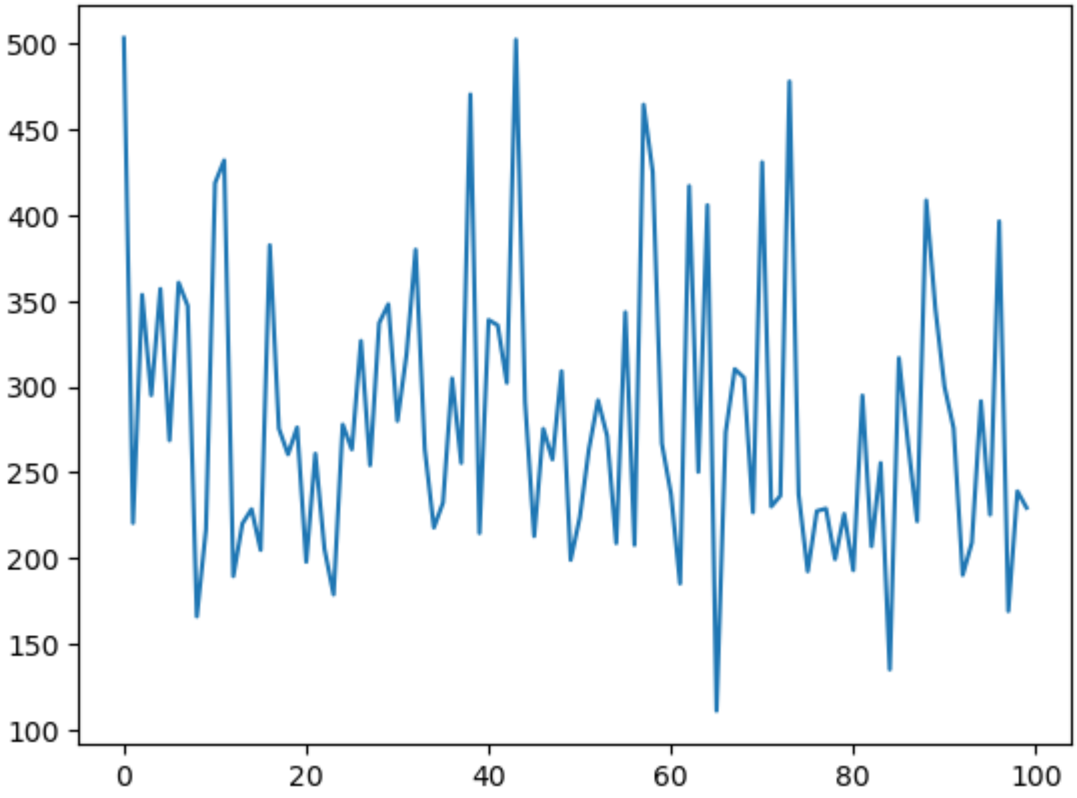
```
tmp = [x for x in range(100)]
```

In [24]:

```
plt.plot(tmp, np.array([l.cpu().detach() for l in loss_list]))
```

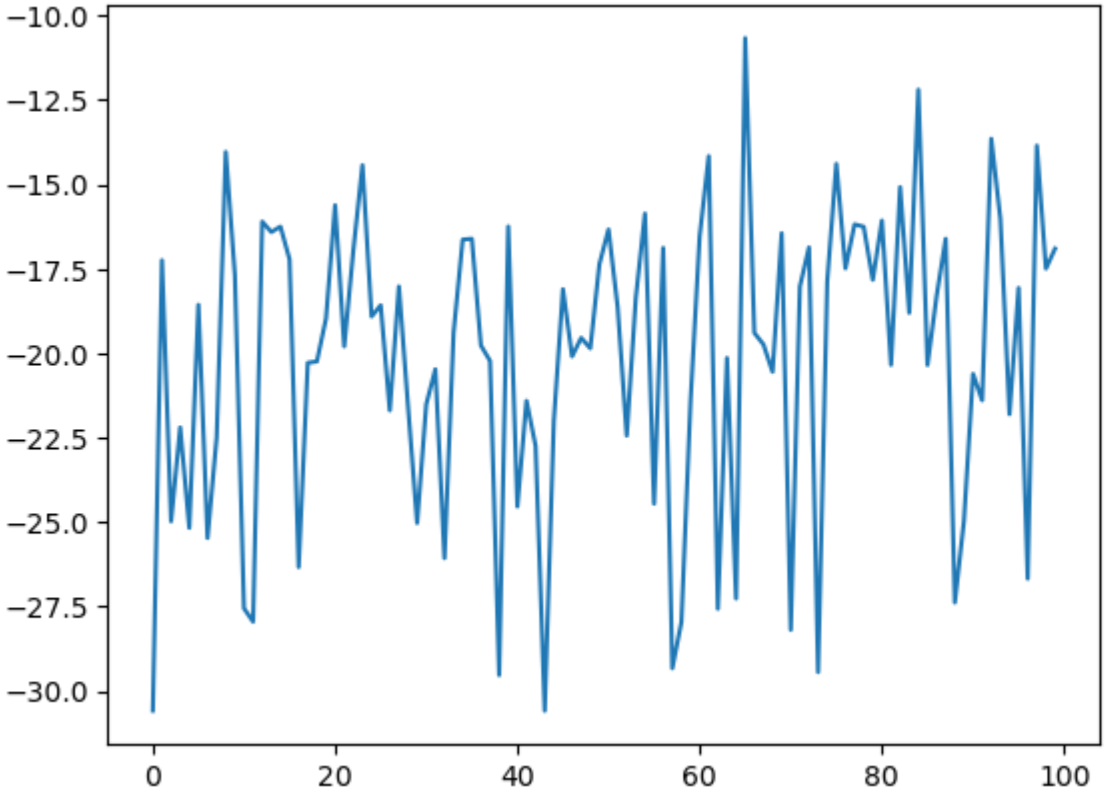
Out[24]:

```
[<matplotlib.lines.Line2D at 0x25139956fe0>]
```



```
In [25]: plt.plot(tmp, np.array([l.cpu().detach() for l in reward_list]))
```

```
Out[25]: [<matplotlib.lines.Line2D at 0x251399c6050>]
```



```
In [26]: o = []
         for i in range(N):
             o.append(observations[200*(i+1)-200:200*(i+1)])
```

```
In [27]: o_old = o.copy()
         o = [j for i, j in enumerate(o) if i%10==0]
```

```
In [28]: o = o_old[-10:]
```

```
In [29]: x_ = [i+1 for i in range(200)]
```

In [30]:

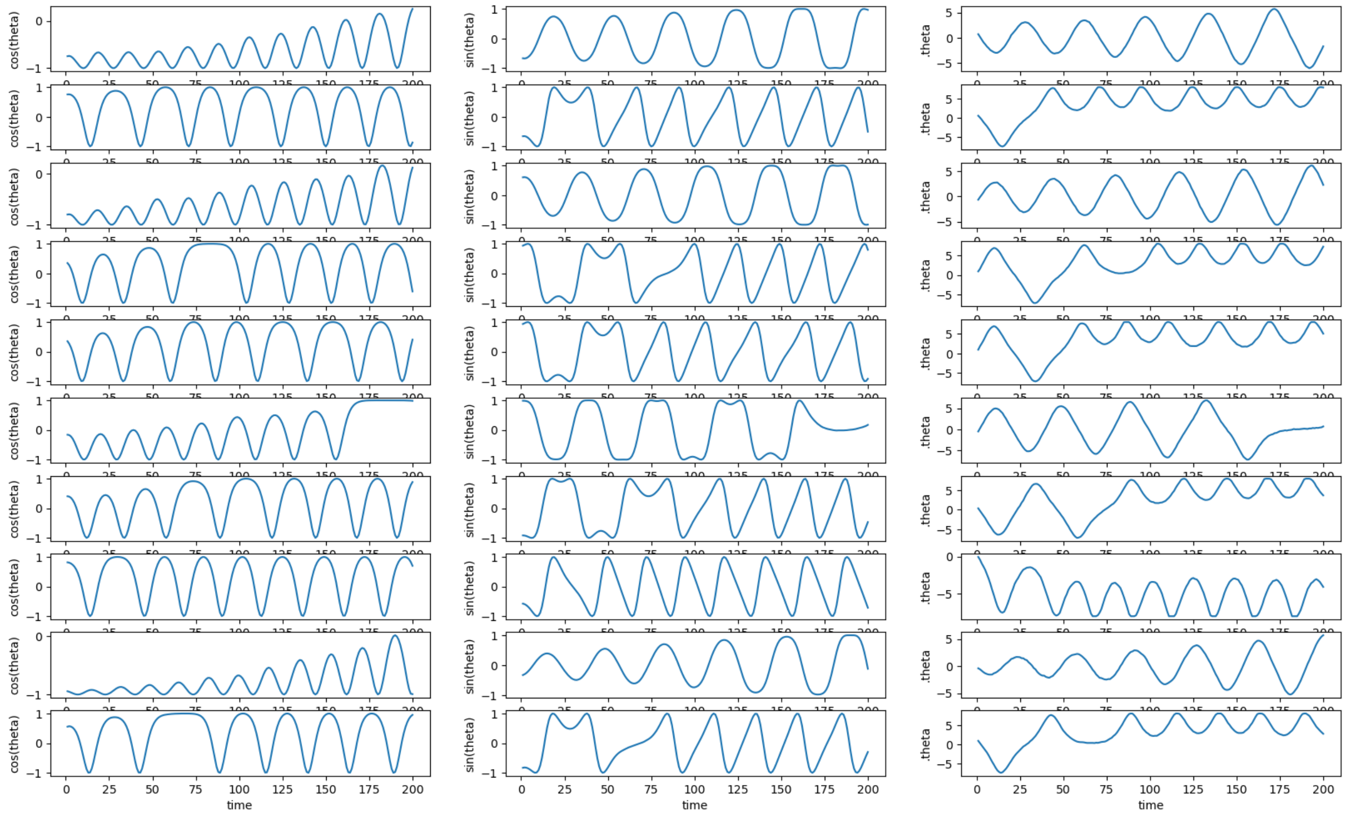
```
np.array(o).shape
```

Out[30]:

```
(10, 200, 3)
```

In [31]:

```
N = 10
plt.figure(1, figsize=(20, 12))
for i in range(N):
    obs = o[i]
    # print(obs)
    theta_ = []
    x_list = []
    y_list = []
    for x,y,v in obs:
        # print(x, y)
        theta_.append(v)
        x_list.append(x)
        y_list.append(y)
    plt.subplot(N, 3, 3*i+1)
    plt.plot(x_, x_list)
    plt.xlabel('time')
    plt.ylabel('cos(theta)')
    plt.subplot(N, 3, 3*i+2)
    plt.plot(x_, y_list)
    plt.xlabel('time')
    plt.ylabel('sin(theta)')
    plt.subplot(N, 3, 3*i+3)
    plt.plot(x_, theta_)
    plt.xlabel('time')
    plt.ylabel('.theta')
```



In [32]:

```
len(total_losses)
```

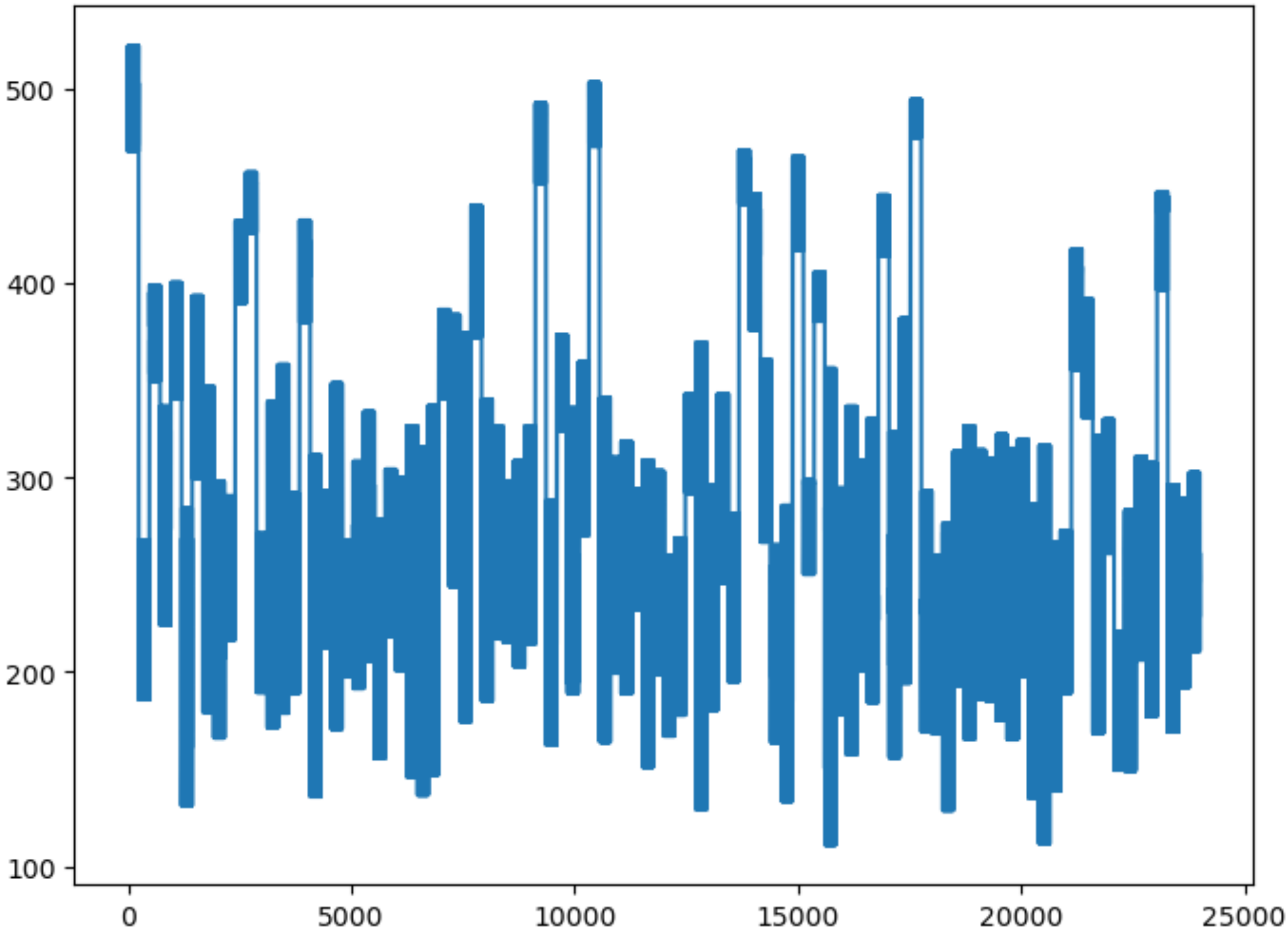
Out[32]:

```
24000
```

In [33]:

```
x = np.arange(len(total_losses))
y = [float(x) for x in total_losses]
plt.figure(figsize=(8, 6))
plt.plot(x, y)
```

Out[33]:
```
[<matplotlib.lines.Line2D at 0x251438cb8e0>]
```



No Clipping and GAE

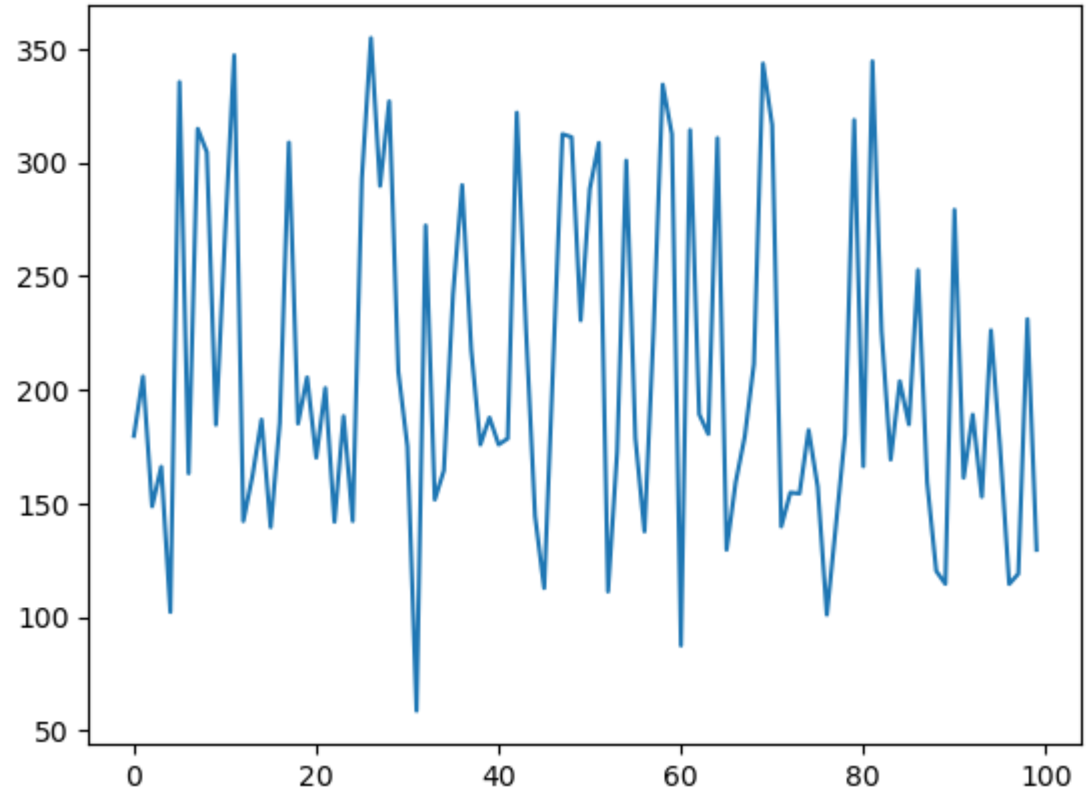
In [34]:
```python
N = 100 # number of times to collect new trajectories and update actor
M = 1 # num of trajectories
max_trajectory_len = 200 # trajectory length
batch_size = 25 # size for minibatch
n_epochs = 30 # number of epochs to optimize loss
alpha = 0.0001
hidden_layers = 2
hidden_dims = 64
gamma = 0.7
lmbda = 0.9
clip_value = 0.1
use_gae = True
use_clip = False

total_losses, observations, loss_list, reward_list = ppo(
    N, M, max_trajectory_len, batch_size, alpha, hidden_layers, hidden_dims,
    gamma, lmbda, clip_value, use_gae, use_clip)
```

In [35]:
```python
tmp = [x for x in range(100)]
```

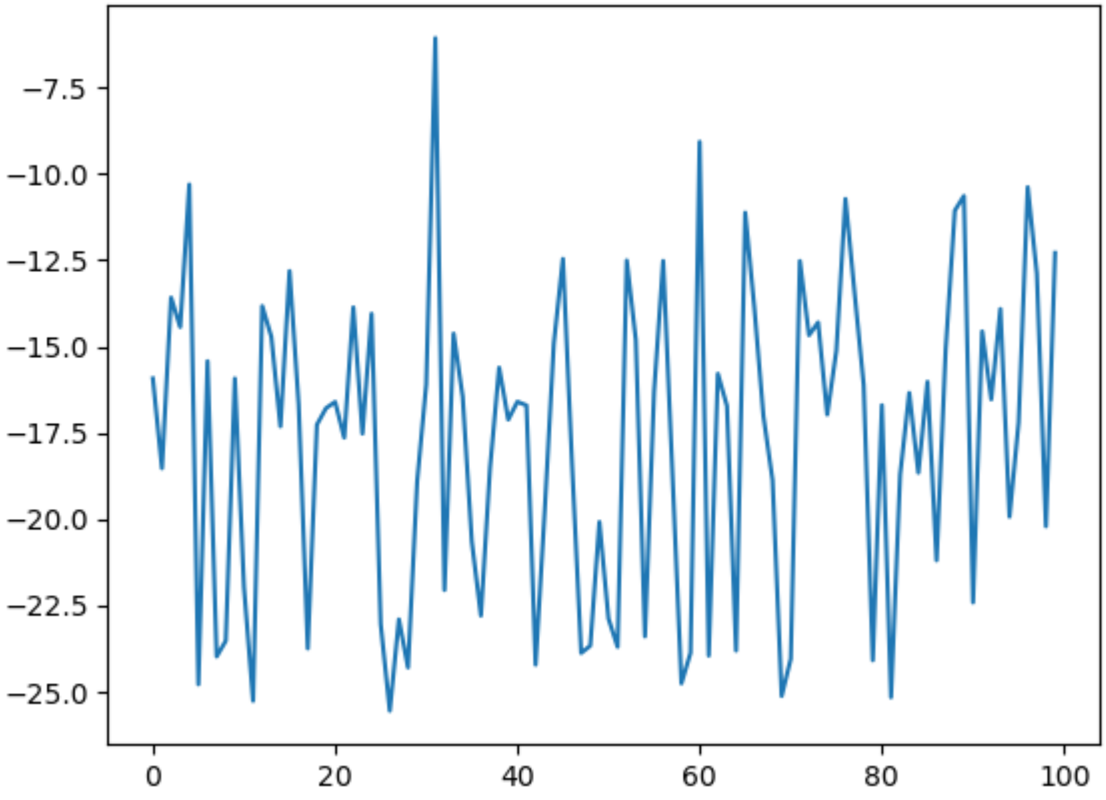
In [36]:
```python
plt.plot(tmp, np.array([l.cpu().detach() for l in loss_list]))
```

Out[36]:
```
[<matplotlib.lines.Line2D at 0x2510a6a2b30>]
```



```
In [37]: plt.plot(tmp, np.array([l.cpu().detach() for l in reward_list]))

Out[37]: [<matplotlib.lines.Line2D at 0x251107b5210>]
```



```
In [38]: o = []
         for i in range(N):
             o.append(observations[200*(i+1)-200:200*(i+1)])
```

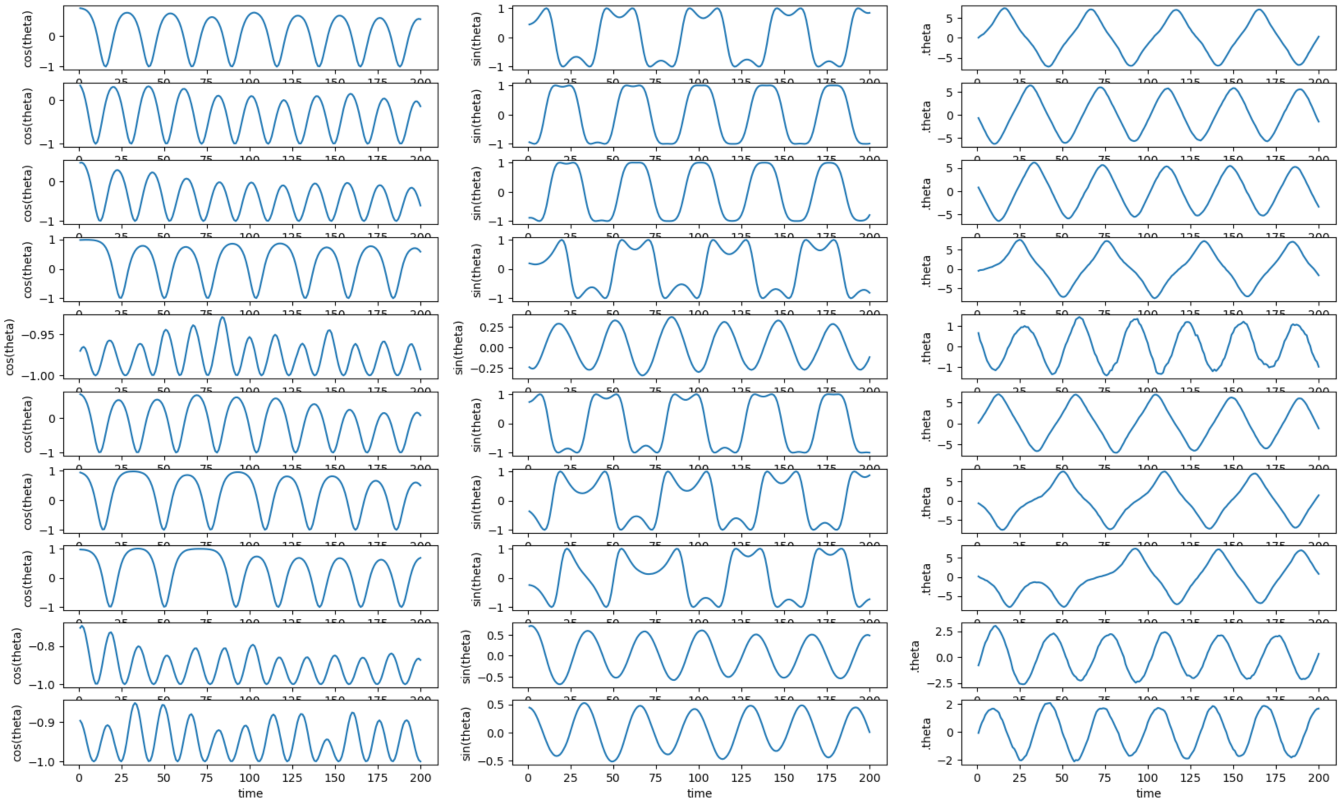
```
In [39]: o_old = o.copy()
         o = [j for i, j in enumerate(o) if i%10==0]
```

```
In [40]: o = o_old[-10:]
```

```
In [41]: x_ = [i+1 for i in range(200)]
In [42]: np.array(o).shape
```

```
Out[42]: (10, 200, 3)
```

```
In [43]: N = 10
         plt.figure(1, figsize=(20, 12))
         for i in range(N):
             obs = o[i]
             # print(obs)
             theta_ = []
             x_list = []
             y_list = []
             for x,y,v in obs:
                 # print(x, y)
                 theta_.append(v)
                 x_list.append(x)
                 y_list.append(y)
             plt.subplot(N, 3, 3*i+1)
             plt.plot(x_, x_list)
             plt.xlabel('time')
             plt.ylabel('cos(theta)')
             plt.subplot(N, 3, 3*i+2)
             plt.plot(x_, y_list)
             plt.xlabel('time')
             plt.ylabel('sin(theta)')
             plt.subplot(N, 3, 3*i+3)
             plt.plot(x_, theta_)
             plt.xlabel('time')
             plt.ylabel('.theta')
```

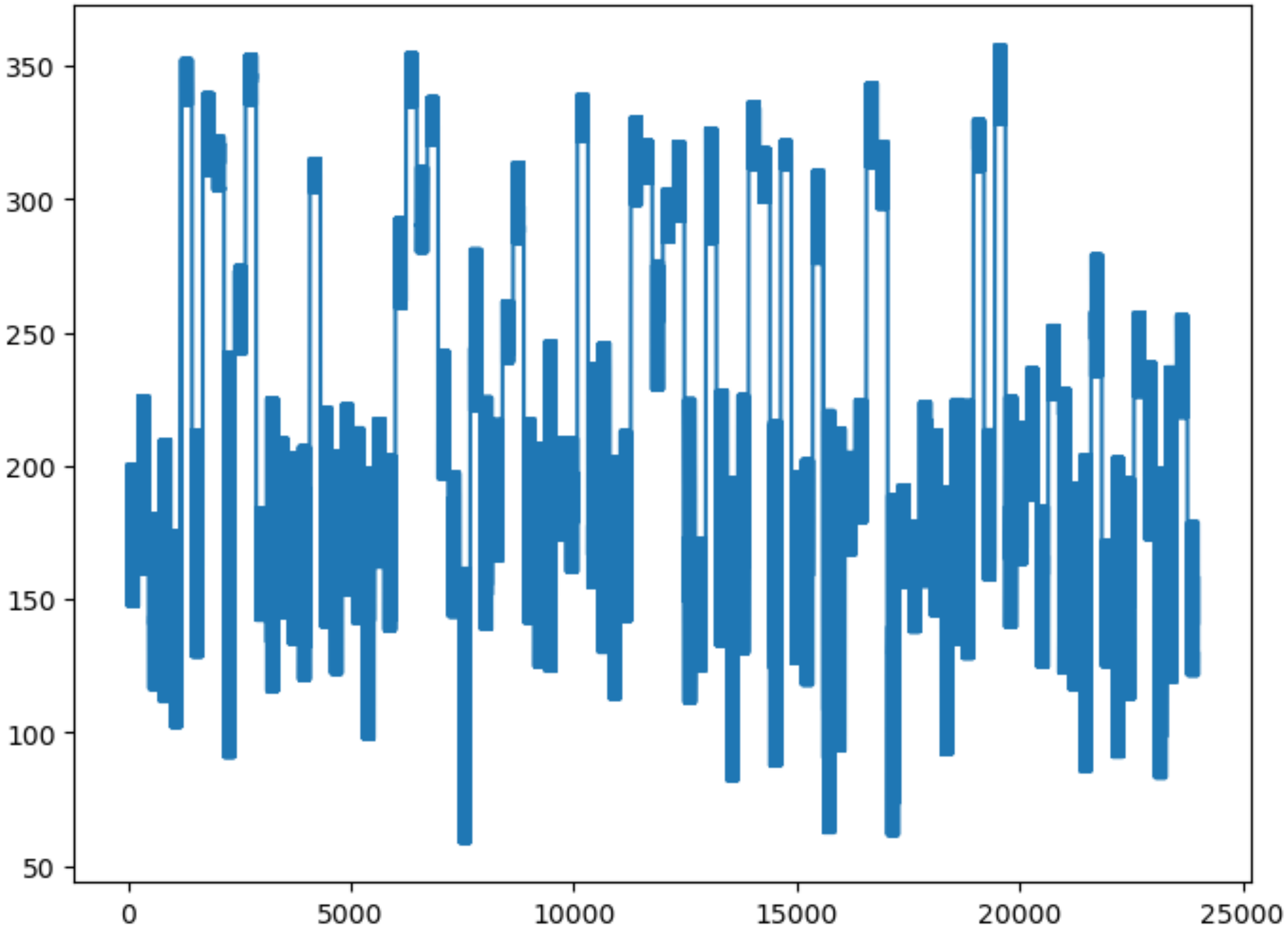


```
In [44]: len(total_losses)
```

```
Out[44]: 24000
```

```
In [45]: x = np.arange(len(total_losses))
         y = [float(x) for x in total_losses]
         plt.figure(figsize=(8, 6))
         plt.plot(x, y)
```

```
Out[45]: [<matplotlib.lines.Line2D at 0x25136191120>]
```



No Clipping and No GAE

In [46]:

```
N = 100 # number of times to collect new trajectories and update actor
M = 1 # num of trajectories
max_trajectory_len = 200 # trajectory length
batch_size = 25 # size for minibatch
n_epochs = 30 # number of epochs to optimize loss
alpha = 0.0001
hidden_layers = 2
hidden_dims = 64
gamma = 0.7
lmbda = 0.9
clip_value = 0.1
use_gae = False
use_clip = False

total_losses, observations, loss_list, reward_list = ppo(
    N, M, max_trajectory_len, batch_size, alpha, hidden_layers, hidden_dims,
    gamma, lmbda, clip_value, use_gae, use_clip)
```

In [47]:

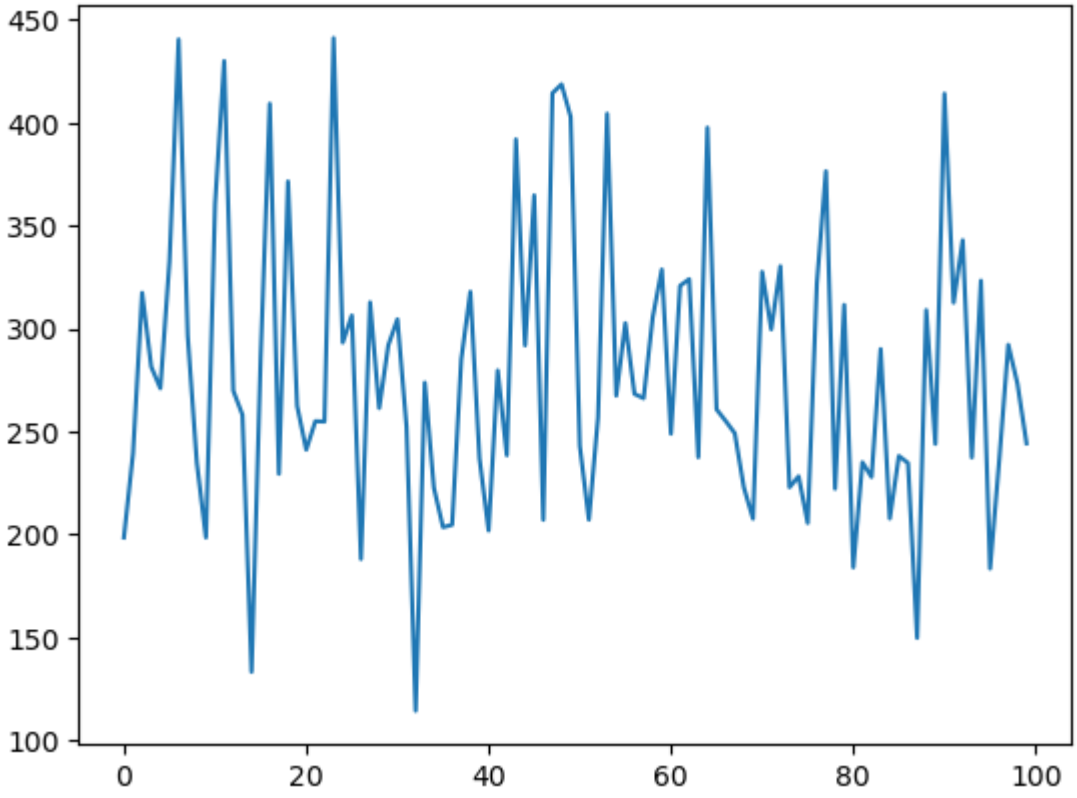
```
tmp = [x for x in range(100)]
```

In [48]:

```
plt.plot(tmp, np.array([l.cpu().detach() for l in loss_list]))
```

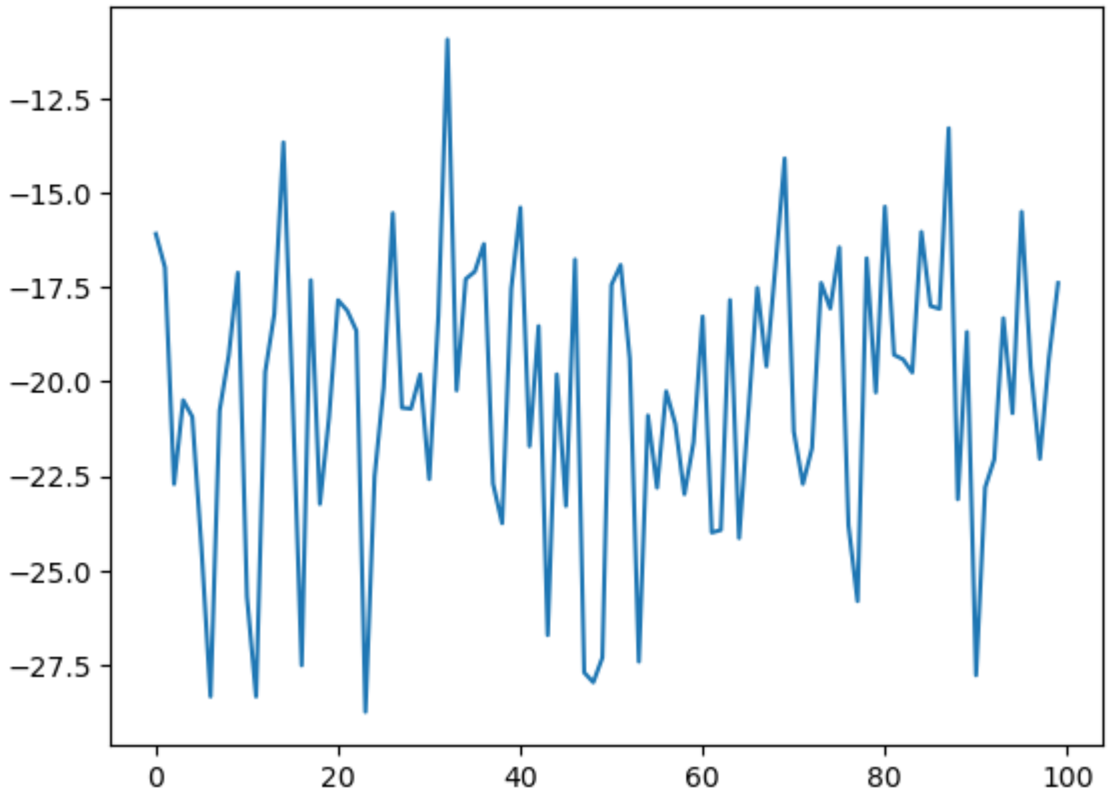
Out[48]:

```
[<matplotlib.lines.Line2D at 0x250d40ce7a0>]
```



```
In [49]: plt.plot(tmp, np.array([l.cpu().detach() for l in reward_list]))
```

```
Out[49]: [<matplotlib.lines.Line2D at 0x250d4129780>]
```



```
In [50]: o = []
         for i in range(N):
             o.append(observations[200*(i+1)-200:200*(i+1)])
```

```
In [51]: o_old = o.copy()
         o = [j for i, j in enumerate(o) if i%10==0]
```

```
In [52]: o = o_old[-10:]
```

```
In [53]: x_ = [i+1 for i in range(200)]
```

In [54]:

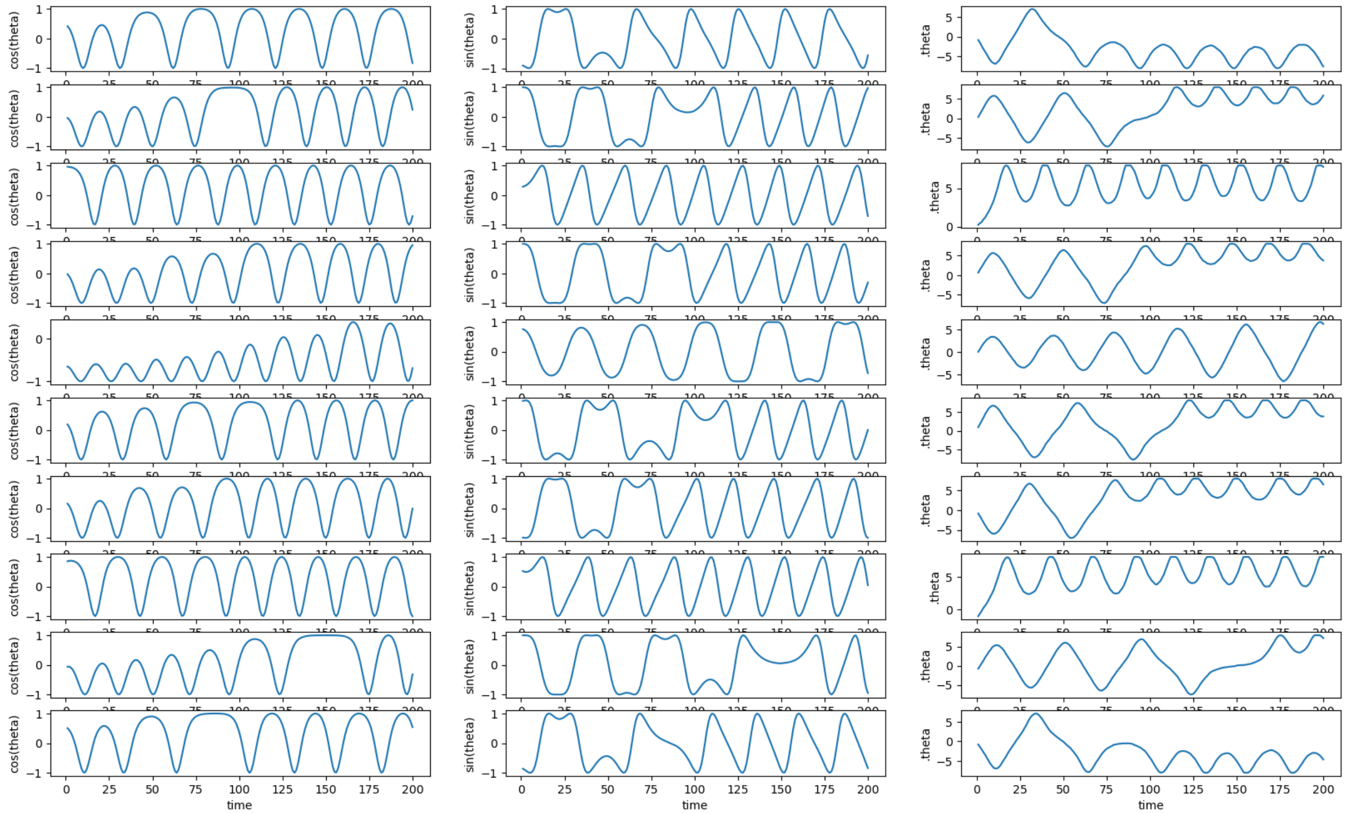
```
np.array(o).shape
```

Out[54]:

```
(10, 200, 3)
```

In [55]:

```
N = 10
plt.figure(1, figsize=(20, 12))
for i in range(N):
    obs = o[i]
    # print(obs)
    theta_ = []
    x_list = []
    y_list = []
    for x,y,v in obs:
        # print(x, y)
        theta_.append(v)
        x_list.append(x)
        y_list.append(y)
    plt.subplot(N, 3, 3*i+1)
    plt.plot(x_, x_list)
    plt.xlabel('time')
    plt.ylabel('cos(theta)')
    plt.subplot(N, 3, 3*i+2)
    plt.plot(x_, y_list)
    plt.xlabel('time')
    plt.ylabel('sin(theta)')
    plt.subplot(N, 3, 3*i+3)
    plt.plot(x_, theta_)
    plt.xlabel('time')
    plt.ylabel('.theta')
```



In [56]:

```
len(total_losses)
```

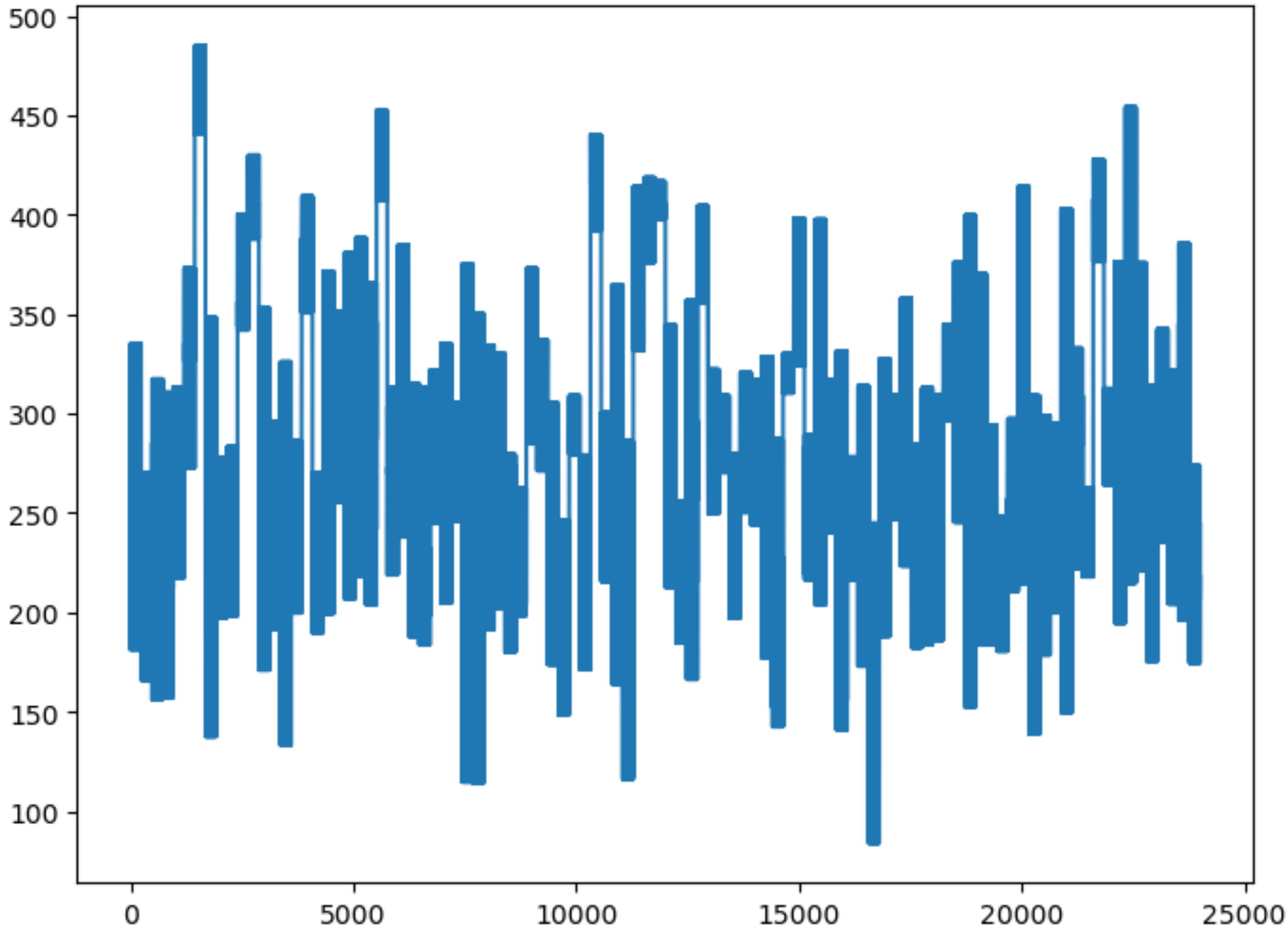
Out[56]:

```
24000
```

In [57]:

```
x = np.arange(len(total_losses))
y = [float(x) for x in total_losses]
plt.figure(figsize=(8, 6))
plt.plot(x, y)
```

```
Out[57]:  [<matplotlib.lines.Line2D at 0x2512b035c00>]
```


500
450
400
350
300
250
200
150
100
0
5000
10000
15000
20000
25000