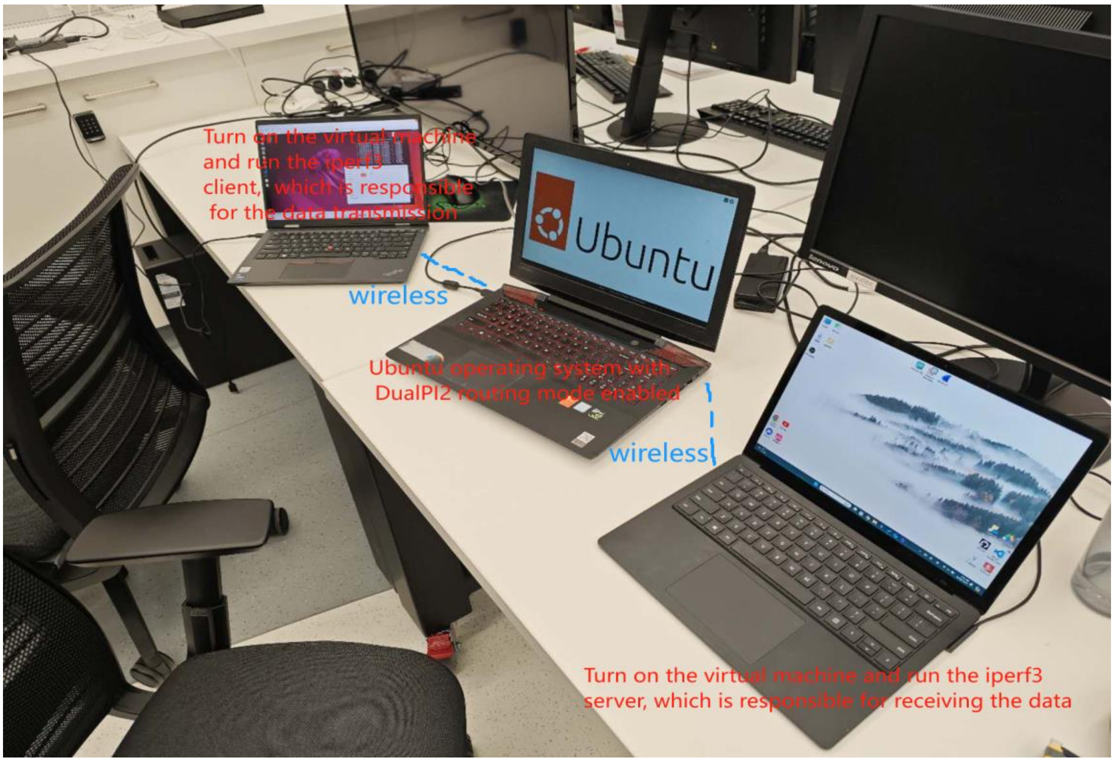
SIT724

LLM-CCA (Large Language Model for Congestion Control Algorithms) is a project to optimise TCP congestion control using Large Language Model (LLM). It dynamically adjusts the Congestion Window (CWND) by analysing network state data (e.g., RTT, CWND, and throughput) to improve network performance in complex network environments and high load conditions. The goal of LLM-CCA is to improve the responsiveness and adaptability of traditional TCP congestion control algorithms to reduce latency and packet loss, while maintaining efficient operation on hardware resource constrained devices.

Hardware and software required for the project:
Hardware: 2 Windows computers, 1 Ubuntu system computer.
Software: VMware Pro 16, iperf3, L4S, Runpod, NetLLM
A virtual machine needs to be installed on 2 windows computers!
L4S needs to be installed on all 3 computers

Testbed



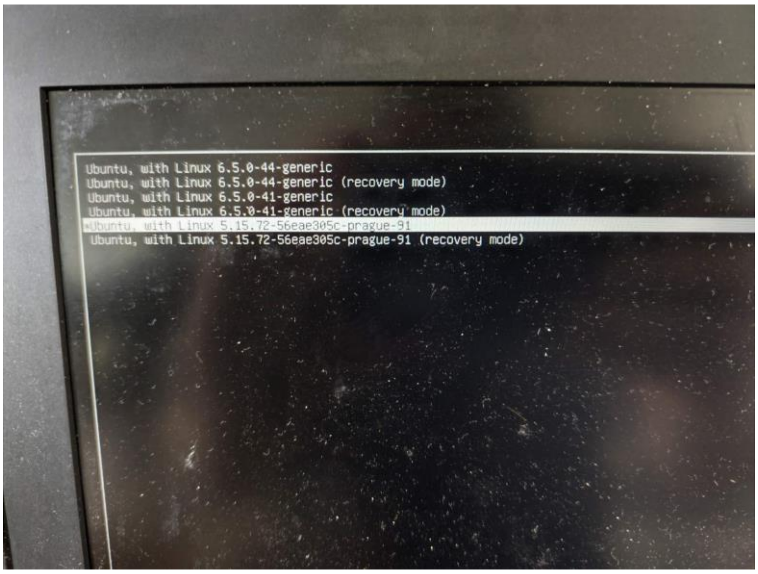
This experiment simulates the data transfer process and collects data by installing L4S for all 3 computers and using iperf3. Due to hardware constraints, we used Runpod to train and run our NetLLM.

How to reproduce experiment:

Step 0:
Client, Server, and Router within the tutorial refer to roles in the experiment structure, not items.
The SIT723 documentation has detailed how Client, Server and Router work with each other, and is currently replacing the L4S framework (replacing CCA and AQM) within the experiment structure of SIT723, which still has 2 Clients, 1 Server and 1 Router roles. tCP Prague is the CCA for L4S. DualPI2 is the AQM for L4S.

Step 1：
Installation of the L4S framework：https://github.com/L4STeam/linux
Please note that you need to install the L4S framework on the Client, Server, and Router.
The L4S framework involves modifying the kernel, so you must ensure that the kernel version of your Ubuntu virtual machine or Ubuntu system has a name similar to Ubuntu with Linux 5.15.72-56eae305c-prague-91.

After installing the L4S framework on the virtual machine, you will need to manually select the kernel boot option. During the virtual machine's startup, continuously press Esc until you are taken to a page where you can select the kernel. If you end up on the desktop, you'll need to reboot and try again. My laptop, which replaces the router, is a dual-boot system with Windows and Ubuntu, and it allows me to select the operating system at startup directly.



Checkpoint: At this point your Client, Server, and Router should have finished installing the L4S framework.

Please use sysctl net.ipv4.tcp_available_congestion_control on both Client and Server side to check if TCP Prague is enabled.

As the Router, you can either use a Linux-supported router or a Linux-based computer (but you will need to manually switch it to router mode).
At this point, use the command::
ip link show
to check how many network interfaces your device has. Linux laptops usually have two (corresponding to WiFi and Ethernet).

Next, use::
tc qdisc show dev [port name]
For example: tc qdisc show dev eth0
to check if AQM rules are enabled on your network interface.

You should see something like:
qdisc [AQM name] [number]: root refcnt 2 limit 1000p target 15ms ...
Here, **[AQM name]** should be **DualPI2**, but by default, it might be **fq**.

If you don't see **DualPI2**, check if **DualPI2** is enabled with:
lsmod | grep dualpi2
You should see the following output:
Sch_dualpi2 24576 3

Next, replace the AQM rule with:
sudo tc qdisc replace dev [port name] root dualpi2

Run the command again
tc qdisc show dev [port name]
You should now see **dualpi2** enabled on the network interface

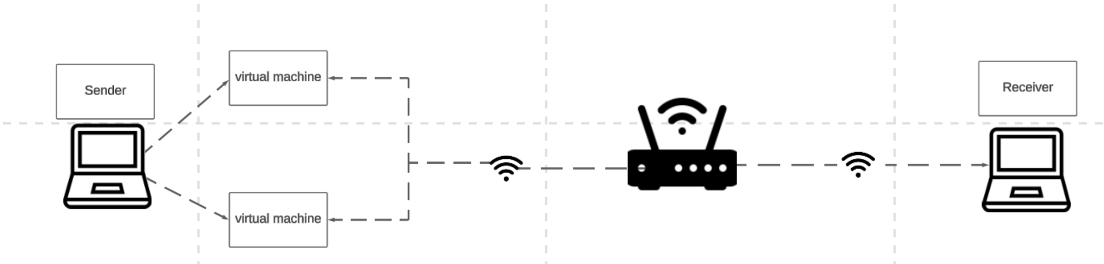
Checkpoint: At this point, your Client, Server, and Router have all installed the L4S framework. 2 VMs will be enabled on the Client side, 1 VM supporting the L4S architecture, and the other VM not supporting the L4S architecture (Reno, CUBIC, and BBR VMs from SIT723). The VM that supports L4S architecture needs to verify that the TCP Prague algorithm is available. Secondly, your Router has enabled DualPI2 rules into the NIC and ensured that this NIC will be used by the accessed device (e.g., DualPI2 rules are enabled for the Ethernet NIC, but your device is still using a Wi-Fi connection).

Step 2：
Begin verifying that the L4S architecture is working properly.
Re-establish the experimental architecture structure for SIT723.

Client X

<- - - - - - > Router <- - - - - -> Server

Client Y



Client X supports L4S architecture, Client Y does not support L4S architecture, Router supports L4S architecture. The reason Server also needs to install L4S architecture is that it needs to have ECN enabled and TCP Prague installed. Otherwise, Client X will show that Server does not support this congestion algorithm when it tries to connect.

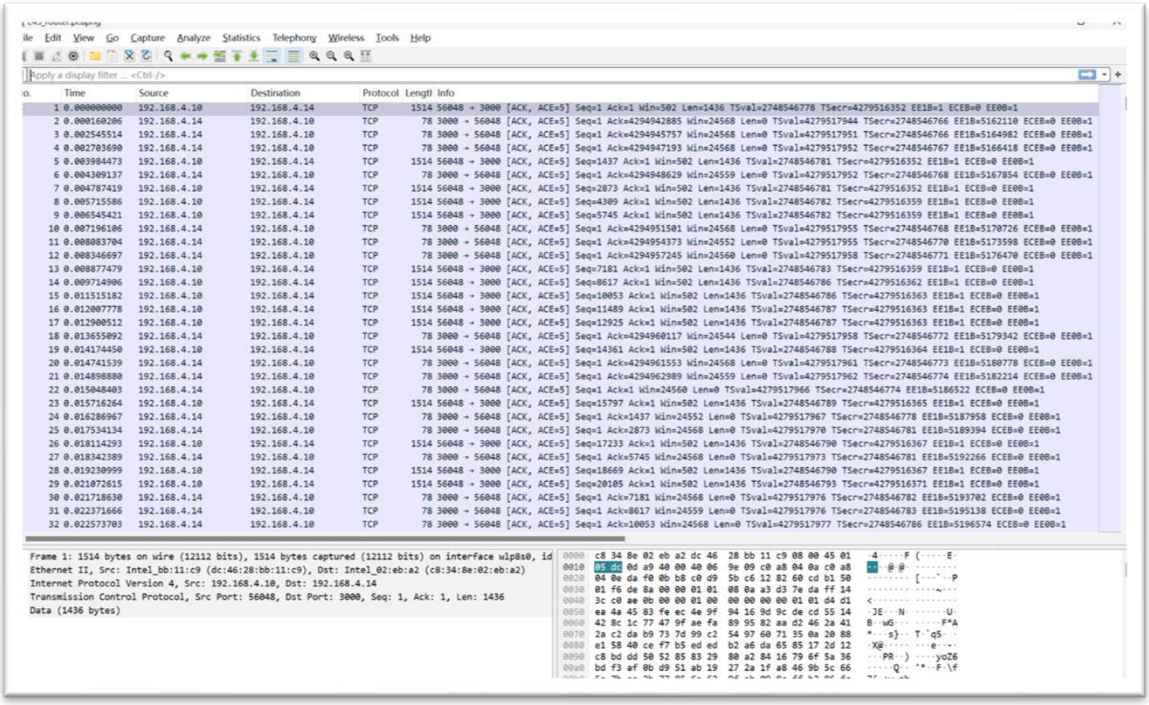
Server：
iperf3 -s -p 3000
iperf3 -s -p 3001

Client X：
iperf3 -c [ip address] -p 3000 -C prague -tinf

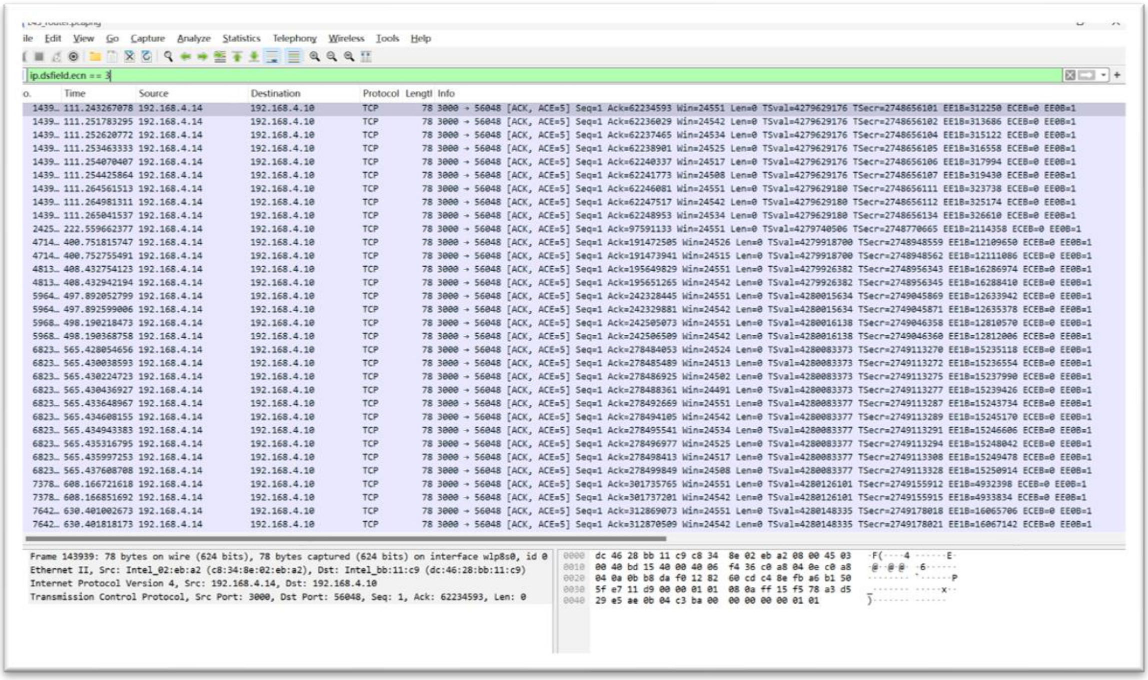
Client Y：
iperf3 -c [ip address] -p 3001 -C cubic -tinf

[ip address] is the IP address of your Server side.

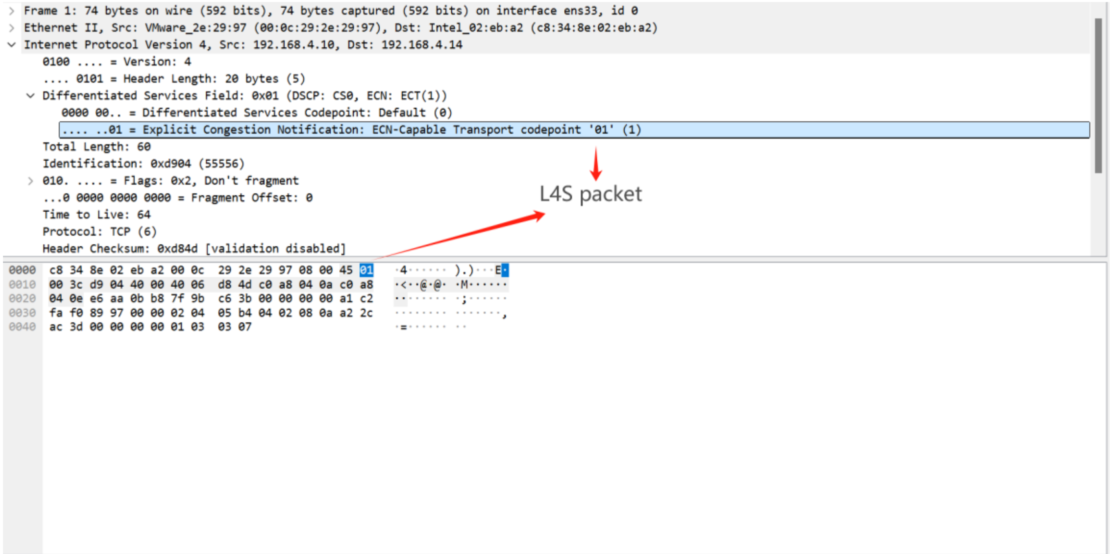
After iperf3 connects properly, please keep Clients connected to Server. Then start Wireshark on the Server side or Router side, after opening Wireshark enter ip.dsfield.ecn == 3

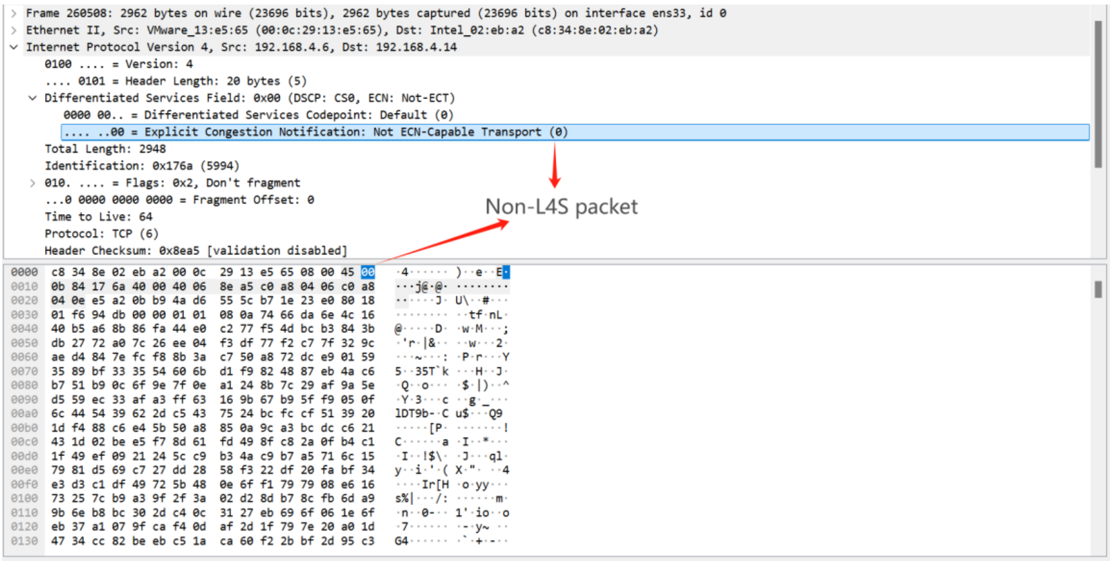


As long as you can see the result after you enter it, it means that ECN=3 marked packets are appearing, and it also means that DualPI2 is marking the congested packets with ECN. please let iperf3's connection last longer, in my previous tests congestion appeared at a ratio of about 1:6000 or 20,000th packets before the first congestion marking occurs.



You can check if there are L4S packets by typing ip.dsfield.ecn == 1 in the filter section, if there are, then L4S is enabled properly. Of course, you can also double-click the packet to see if it is an L4S packet instead of using the filter.





Step 2:

Collecting L4S Data

Since this experiment requires collecting data for **CWND**, **RTT**, and **throughput**, and we are using **iperf3** to simulate data transmission, we can directly write a script that includes the **iperf3** command. This script will save all the data during the iperf3 data transmission in a **JSON** file.

1. First, open the virtual machine and create a folder.
2. Open the terminal and create a script file using the command

Command: **nano file_name.sh**

3. Open the .sh file and enter:

Code:

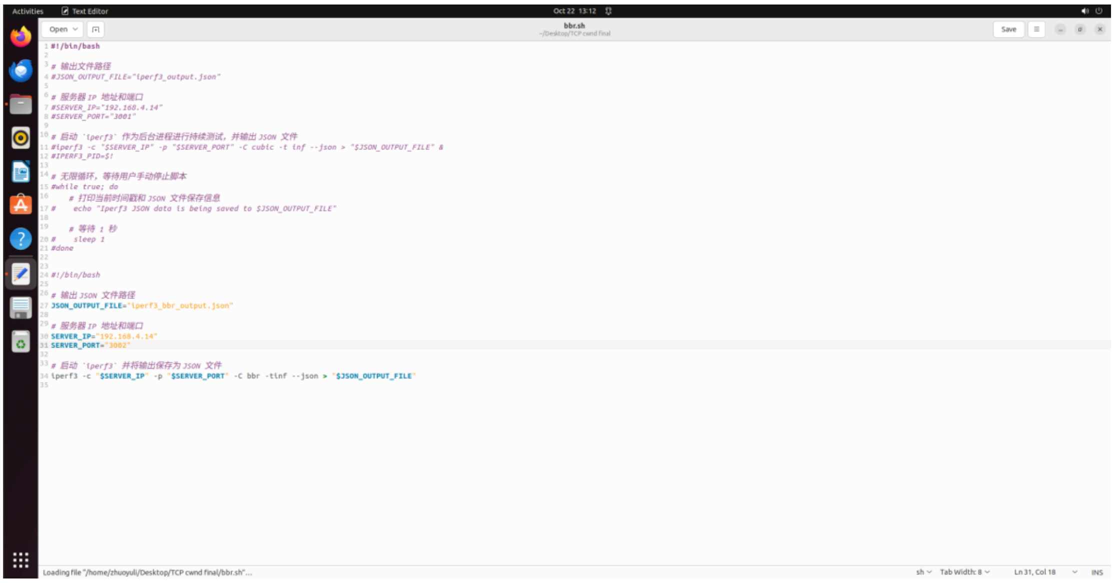
```
JSON_OUTPUT_FILE="iperf3_bbr_output.json"

SERVER_IP="192.168.4.14"
SERVER_PORT="3002"

iperf3 -c "$SERVER_IP" -p "$SERVER_PORT" -C bbr -tinf --json > "$JSON_OUTPUT_FILE"
```

Please replace **SERVER_IP** and **SERVER_PORT** with the server's IP and port. Currently, the script runs using the **BBR congestion control algorithm**. If you want to use a different algorithm, simply modify the **bbr** after the -C flag in the last line of the code to the desired congestion control algorithm, such as **Cubic**:

```
iperf3 -c "$SERVER_IP" -p "$SERVER_PORT" -C cubic -tinf --json > "$JSON_OUTPUT_FILE"
```



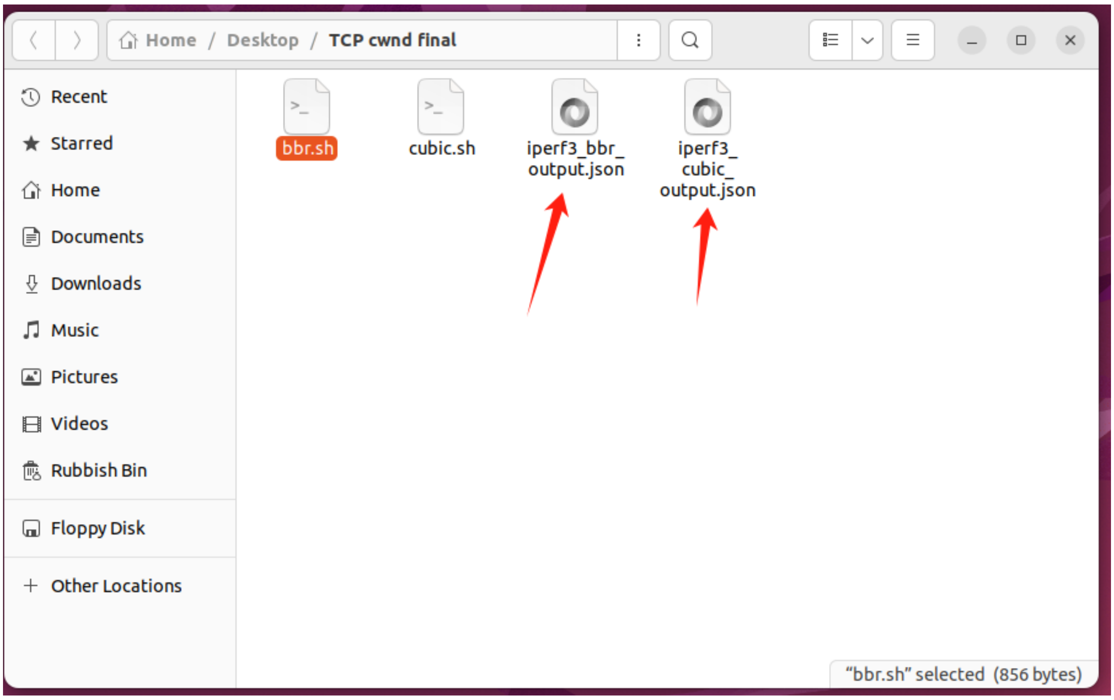
4. After completing the script, remember to save it and then enter the following command in the terminal to add execution permissions to the script:

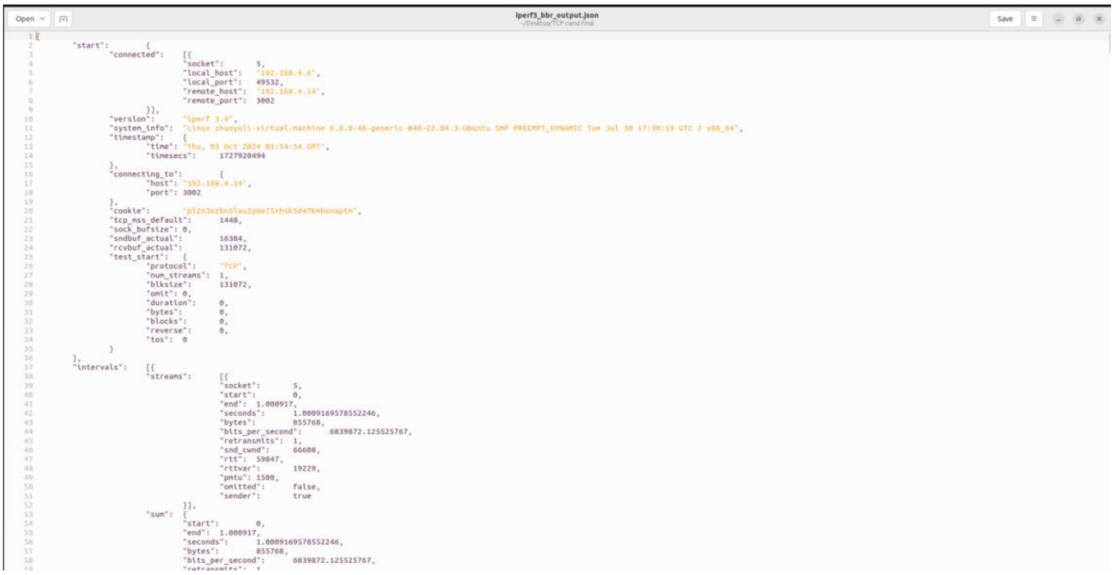
Command: **chmod +x file_name.sh**

5. Once everything is set up, you just need to ensure that **iperf3** is running on the server, and then you can run the script with the following command

Command: **./file_name.sh**

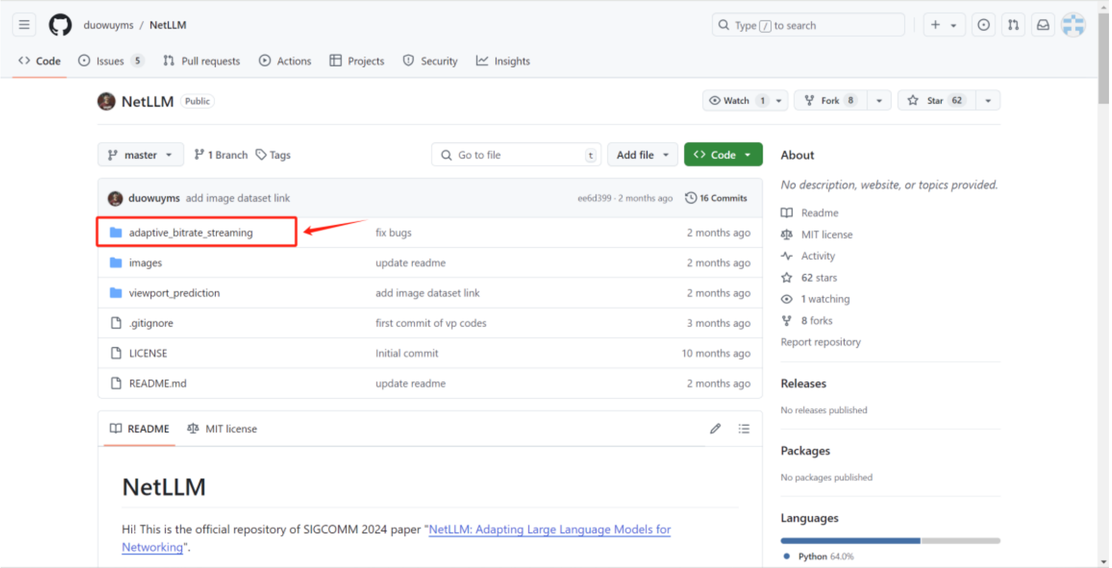
6. The JSON file containing all the information from the **iperf3** data transmission will be saved in the same folder where the script is located.





Step 3: **How to Download Llama2 and the Original NetLLM**

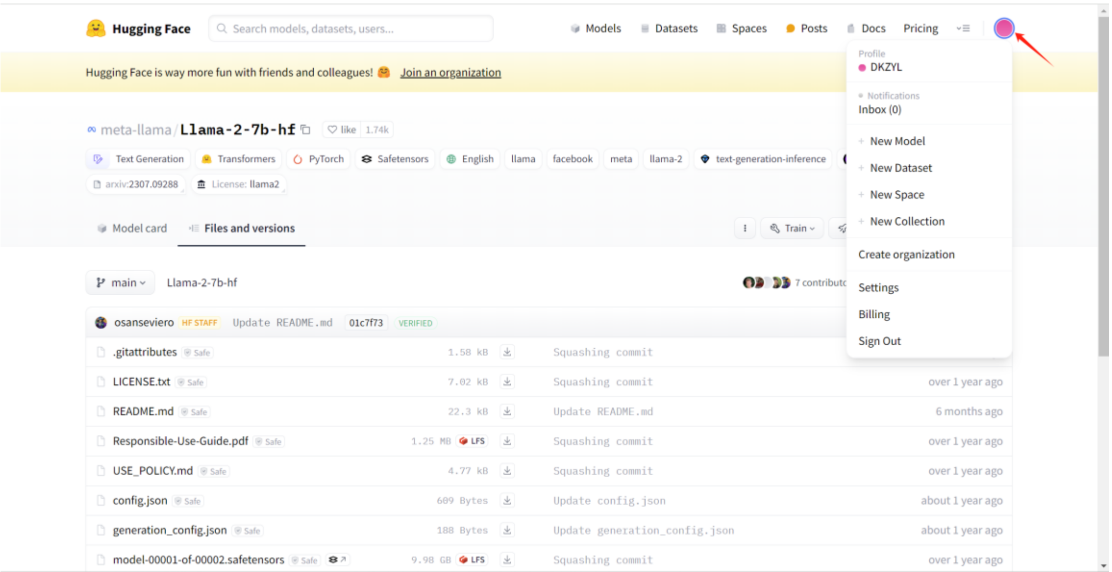
1. First, go to the official NetLLM GitHub and download the adaptive folder. Official GitHub link: https://github.com/duowuyms/NetLLM/tree/master



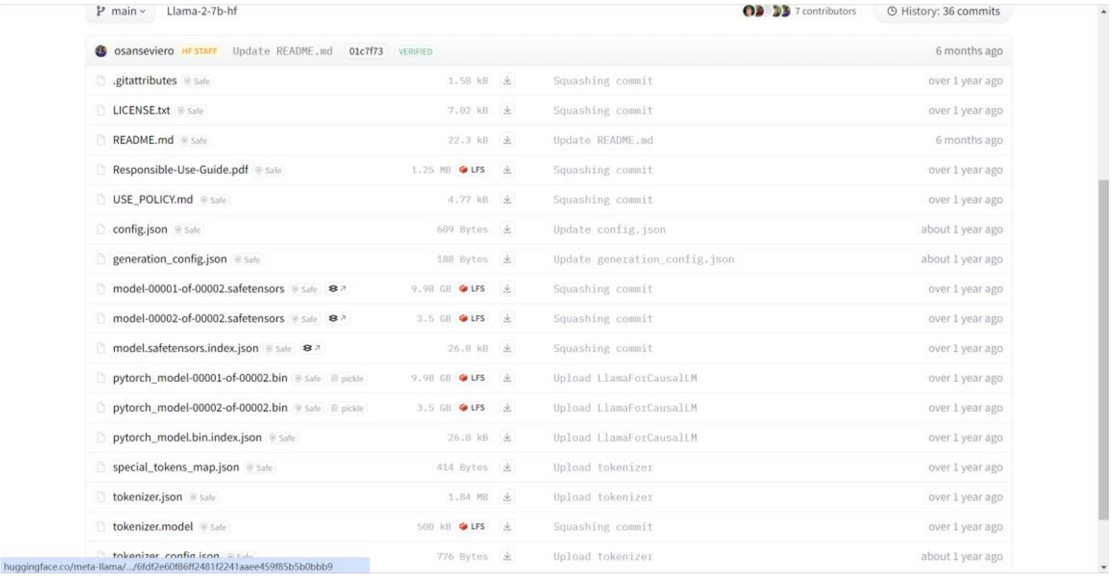
2. The LLM used in this experiment is **meta-llama/Llama-2-7b-hf**, so we will go to **Hugging Face** to download it.

Link: https://huggingface.co/meta-llama/Llama-2-7b-hf/tree/main

Due to Hugging Face's requirements, you need to log in or sign up before downloading Llama-2. If you encounter any issues during the download, make sure to sign Hugging Face's community guidelines and request permission to download Llama-2. All of this can be done directly on their website.

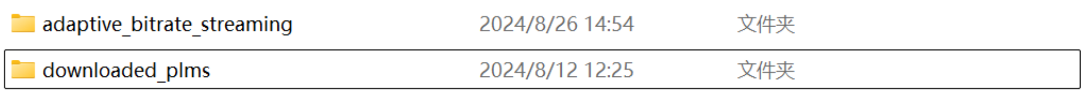


After obtaining permission, please download all the files from that page

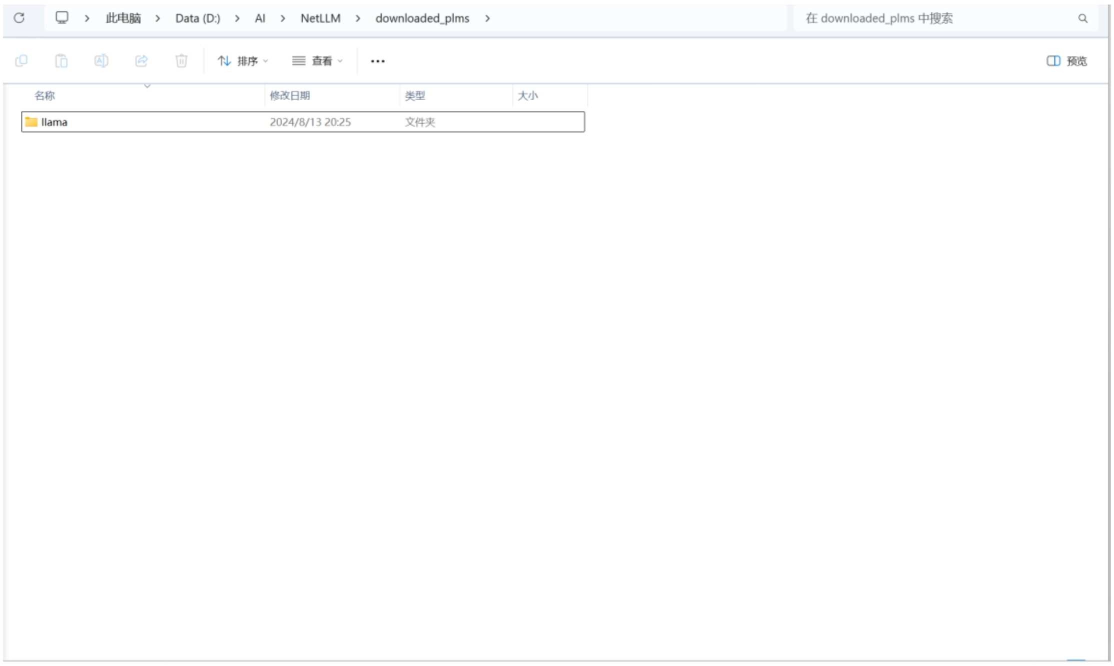


After downloading the **Llama-2** and **adaptive** folders, it's recommended to first create a larger folder to contain both of these folders. This will make it easier to manage and upload to Runpod later, as you'll only need to upload one large folder.

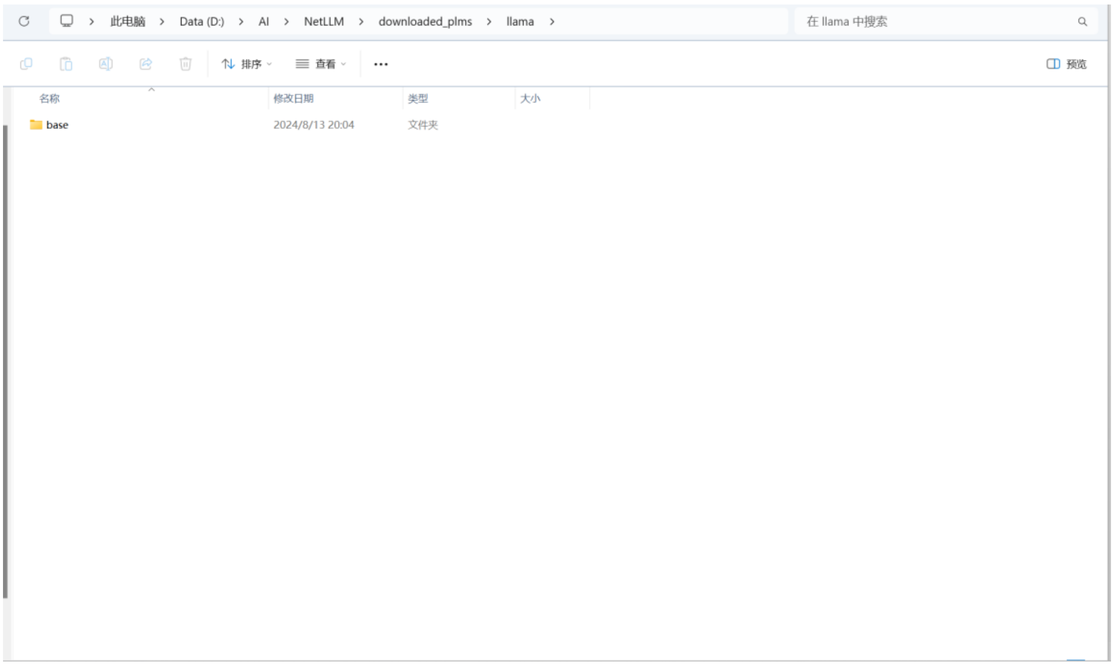
Next, place the **adaptive** folder inside this larger folder. Then, create a folder named **downloaded_plms** within it



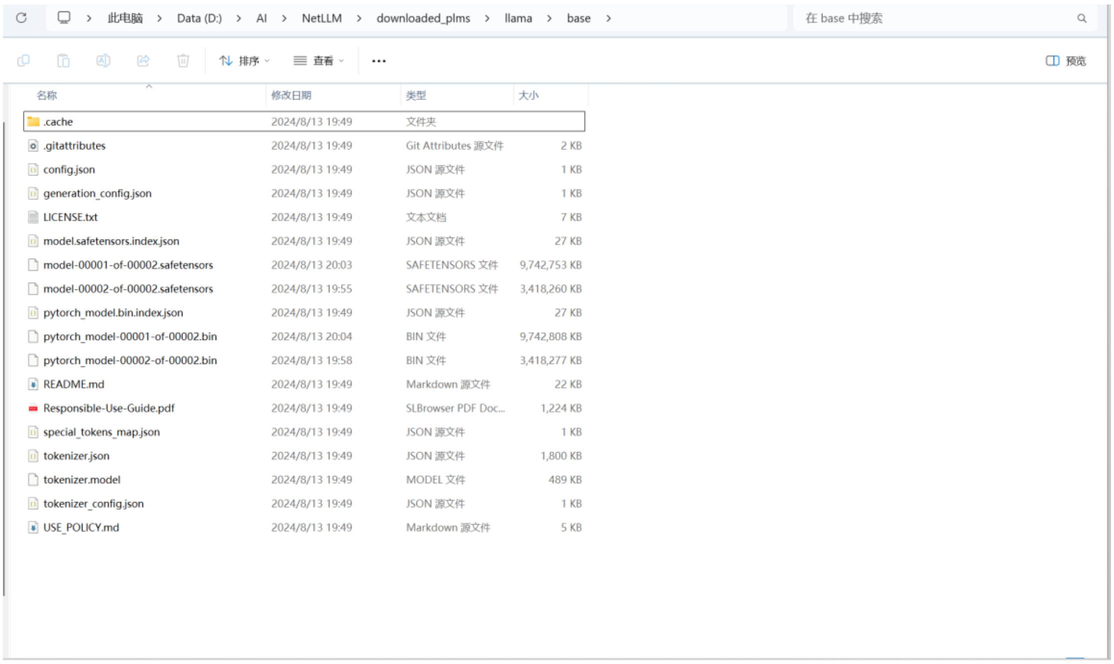
In this downloaded_plms folder we need to create a llama folder.



In this llama folder we need to create another base folder.



Then put all the Llama-2 files you just downloaded in the base folder.

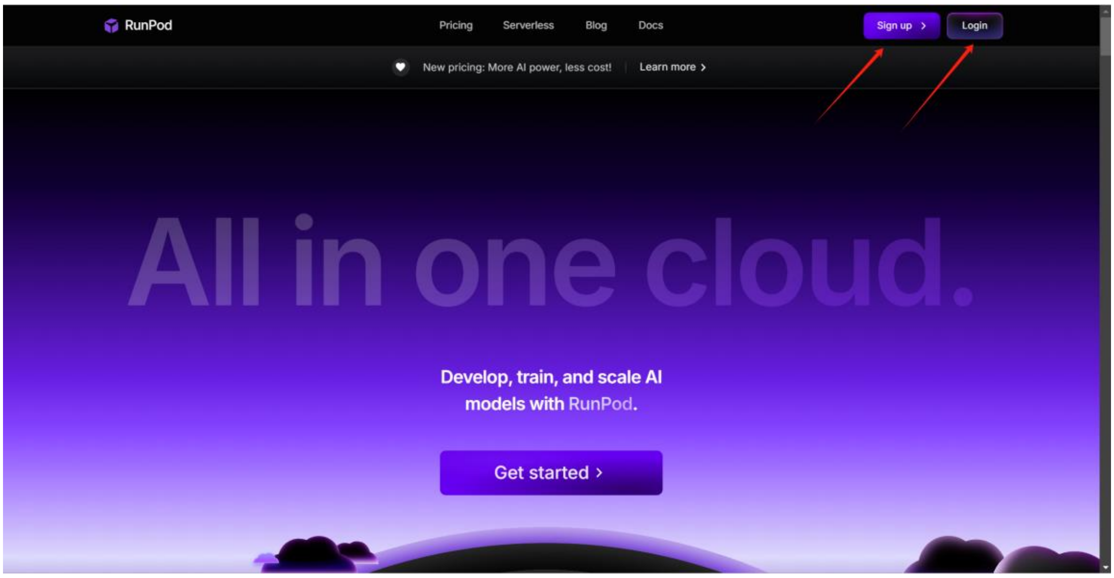


Step 4: **How to Use Runpod to Train NetLLM**

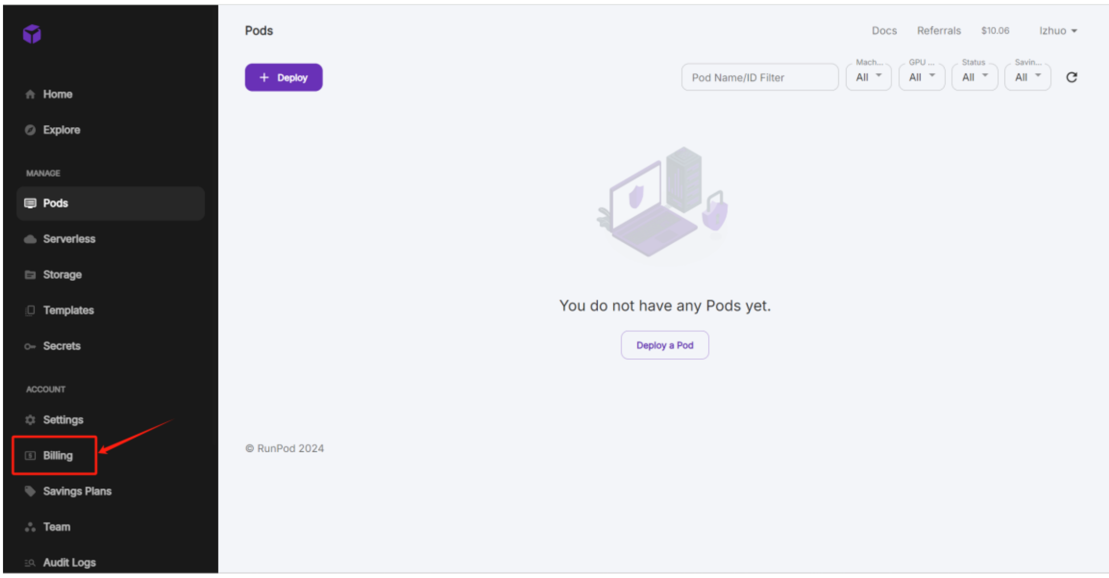
In this experiment, we use **Runpod** to run the **NetLLM** architecture.

Runpod link: https://www.runpod.io/

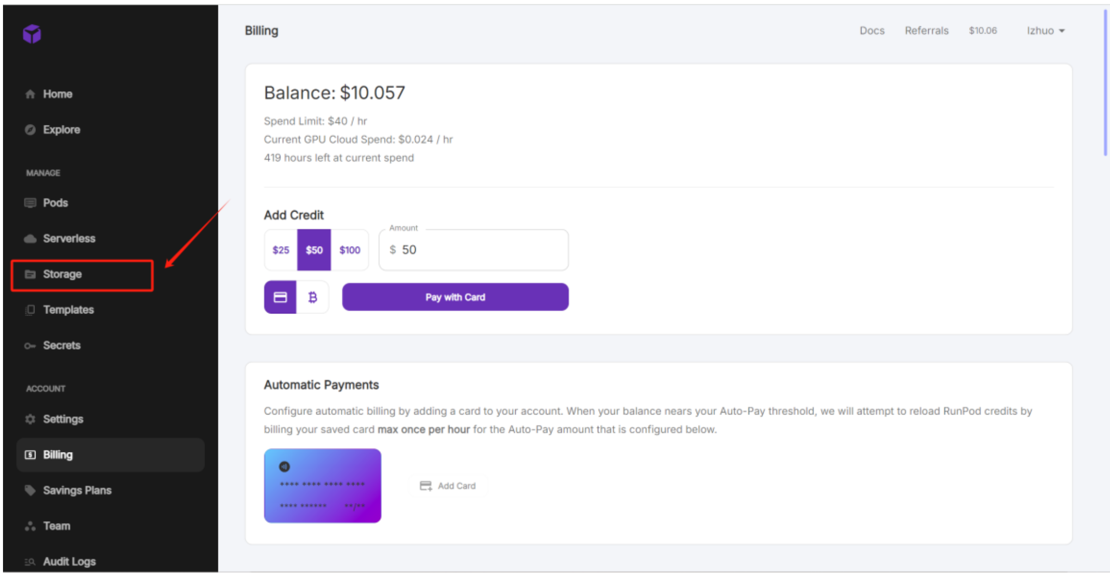
1. Click **Sign Up** or **Login** to register or log in



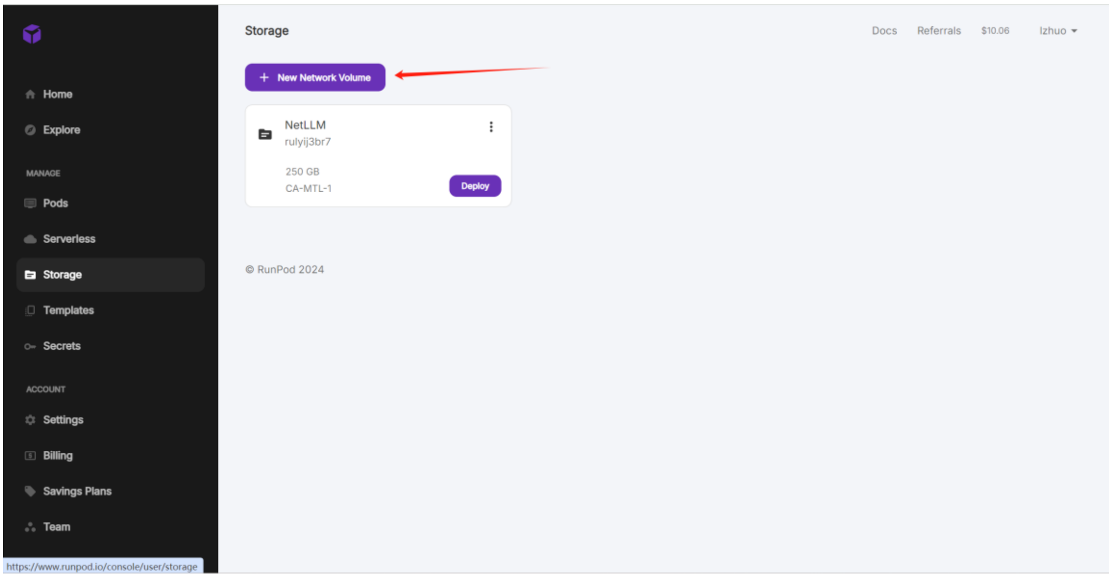
2. After logging in, first go to the **Billing** section to add funds, as Runpod cannot be used without a balance.



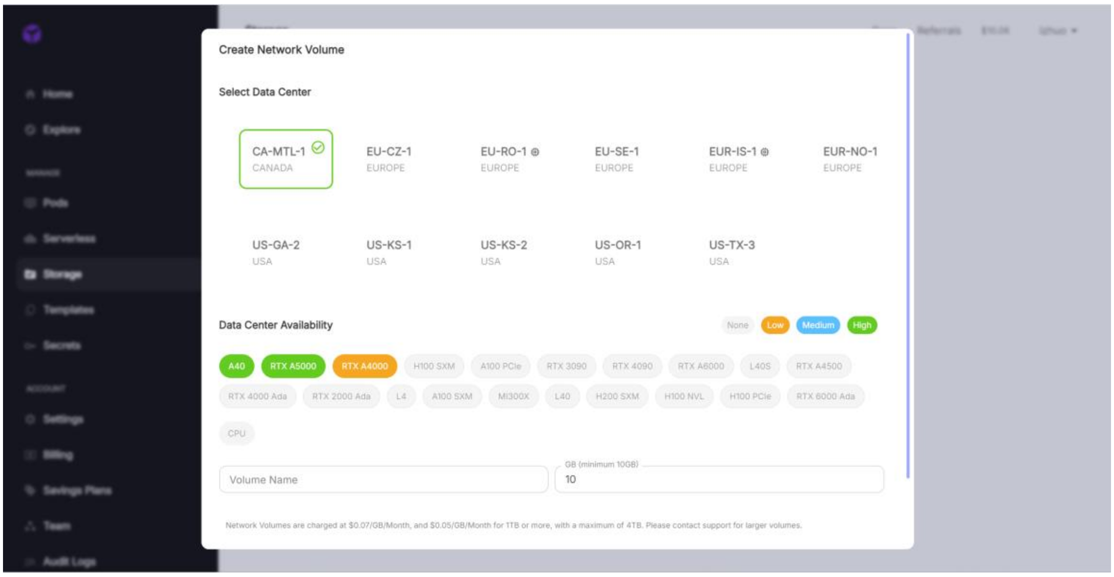
3. Click on the **Storage** section and create a storage. The storage is used to save the **LLM model** and **NetLLM architecture**. If you don't create storage and use only the container in Runpod, all data will be lost after the pod is shut down. By using storage, you only need to upload the **NetLLM** framework and **LLM** once, and you can reuse them multiple times afterward



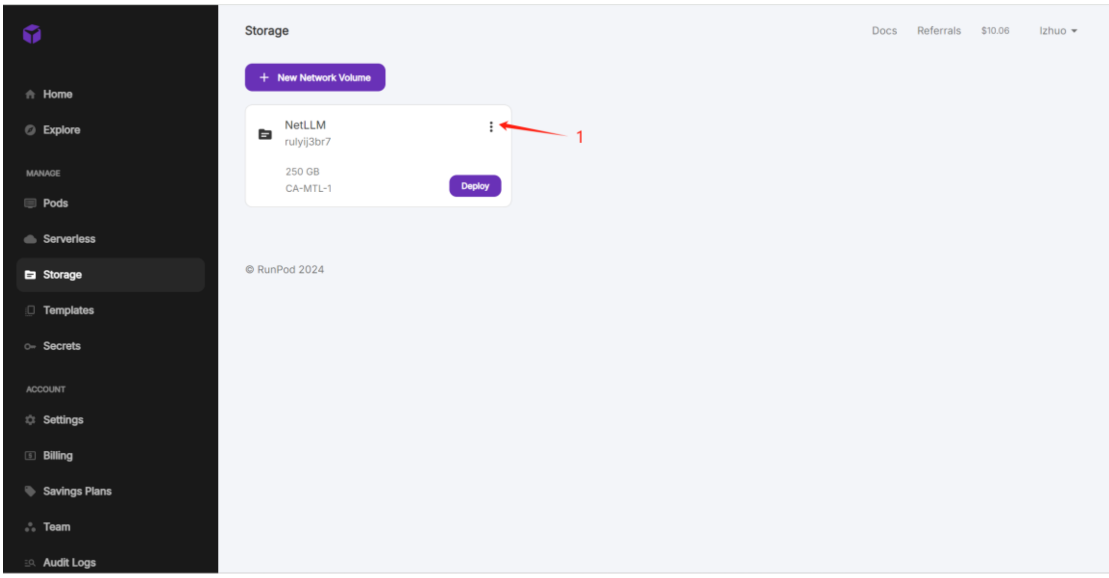
I've already created one, so you can ignore this step. Click on **New Network Volume**

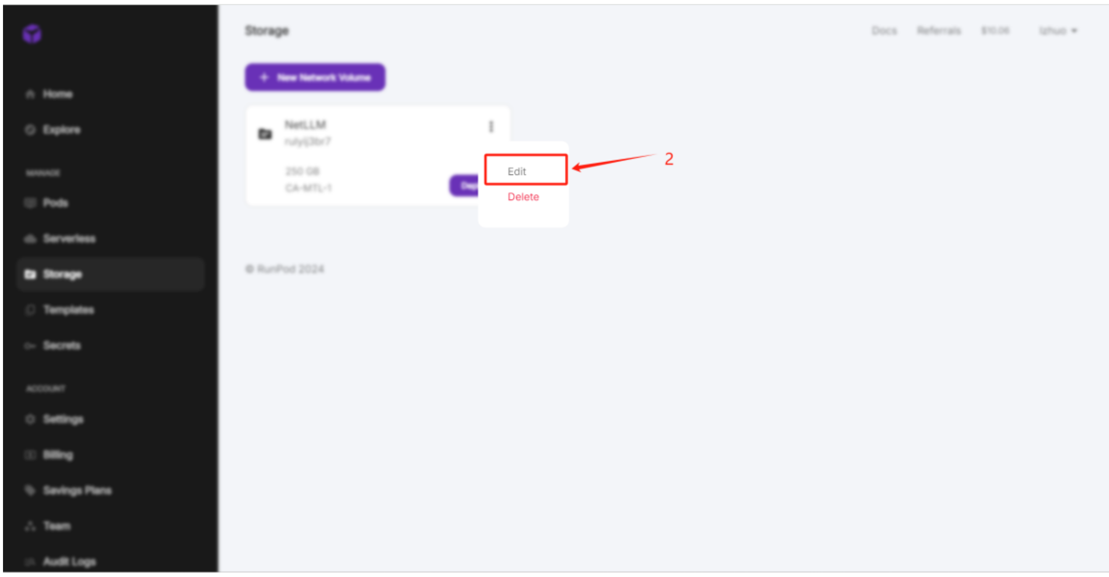


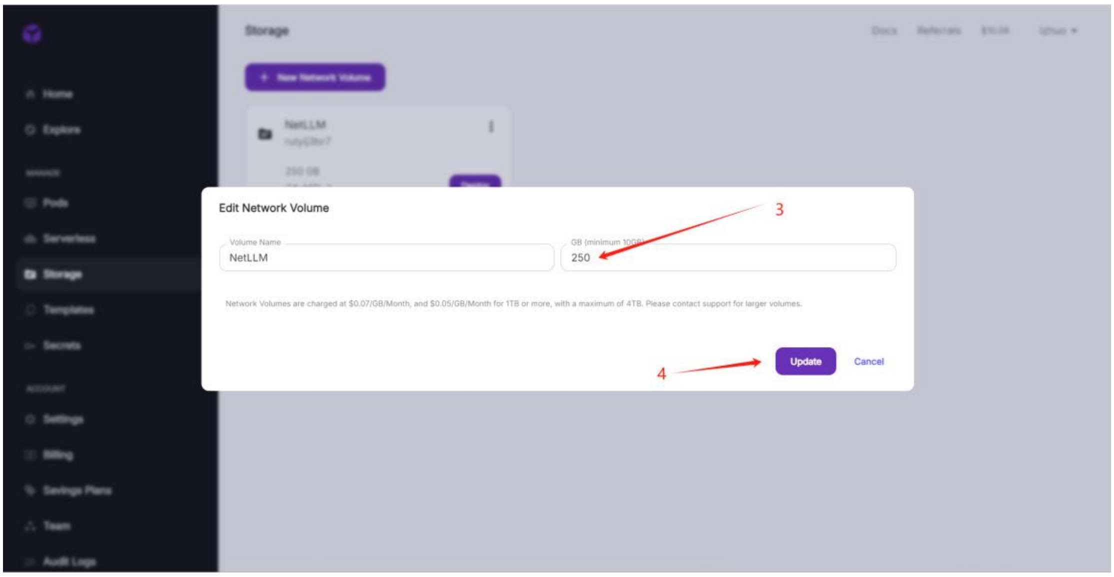
I recommend selecting **CA-MTL-1**, as this is the Runpod server closest to Australia. **Data Center Availability** shows the available GPU servers in the area. For our NetLLM project, we only need to rent 2 A40 GPUs (servers), making **CA-MTL-1** the best choice. Please remember to modify the storage name and adjust the storage size, with 50GB to 100GB being the recommended size. Once the modifications are done, click **Create** to create the storage



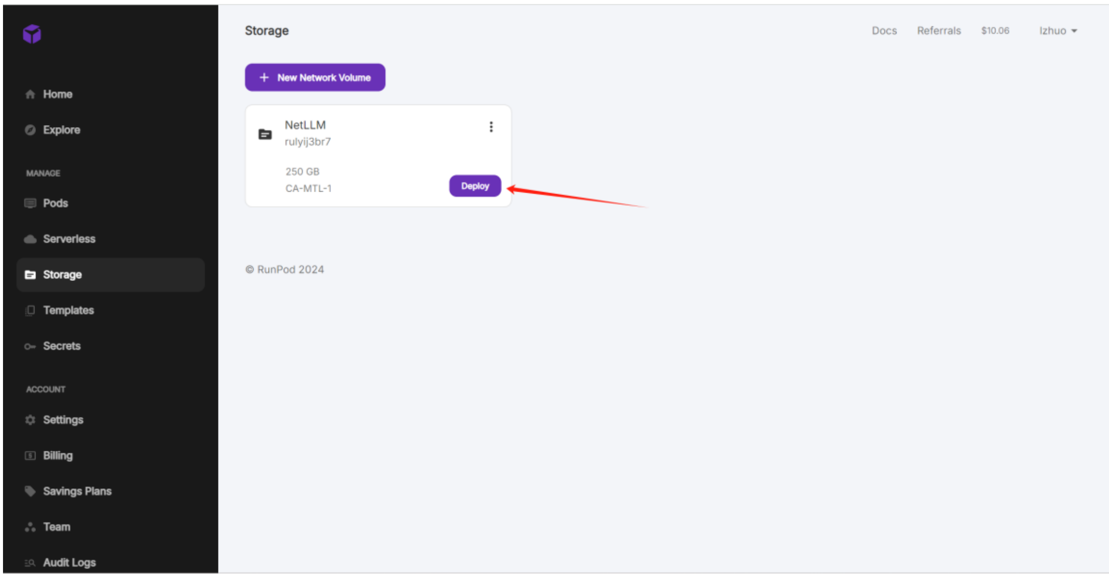
If you subsequently find that the storage space is insufficient, you can return to storage to expand the storage space



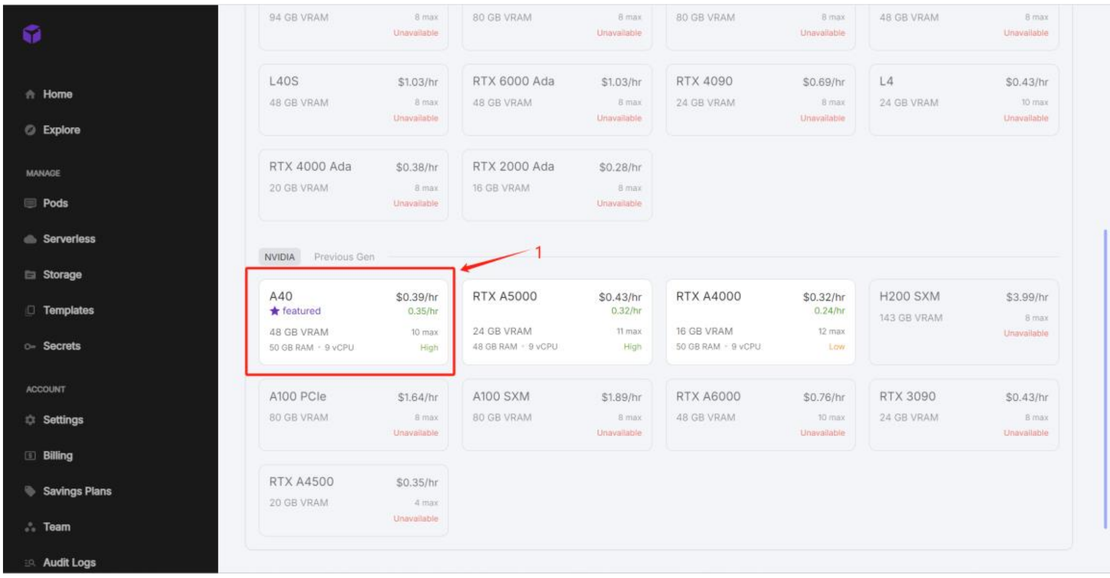

Edit
Delete
2


Edit Network Volume
Volume Name
NetLLM
250
3
Network Volumes are charged at $0.07/GB/Month, and $0.05/GB/Month for 1TB or more, with a maximum of 4TB. Please contact support for larger volumes.
4
Update
Cancel

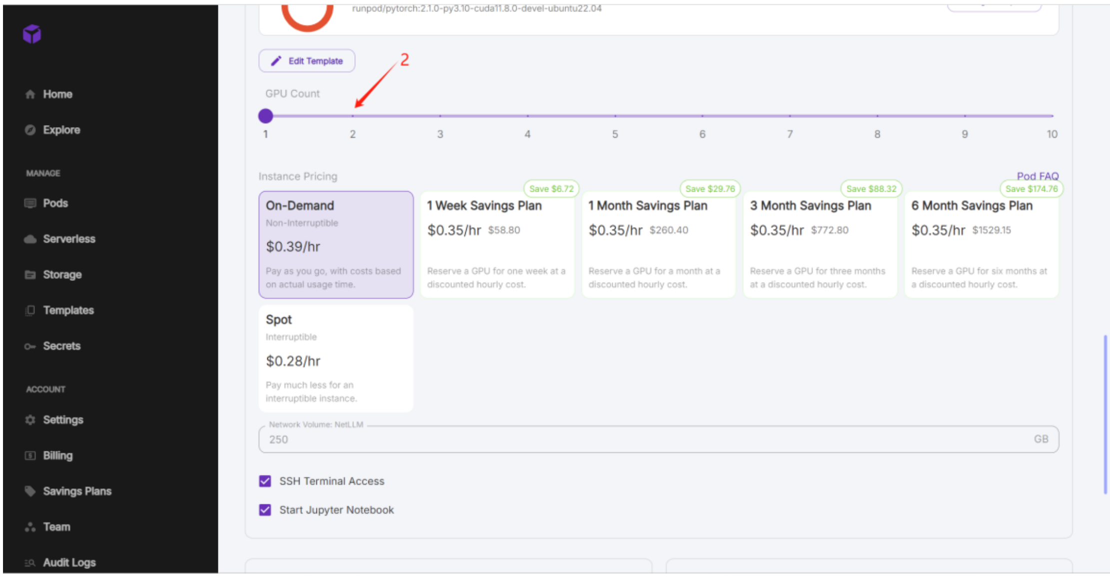
4. After setting up the storage, click on deploy.

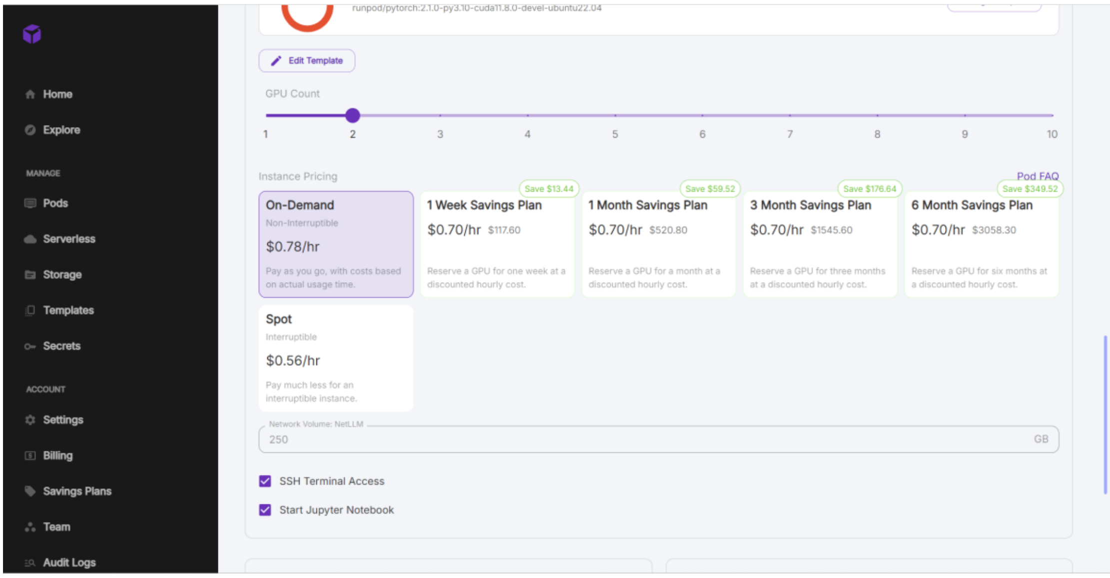


Find A40 and click on it.

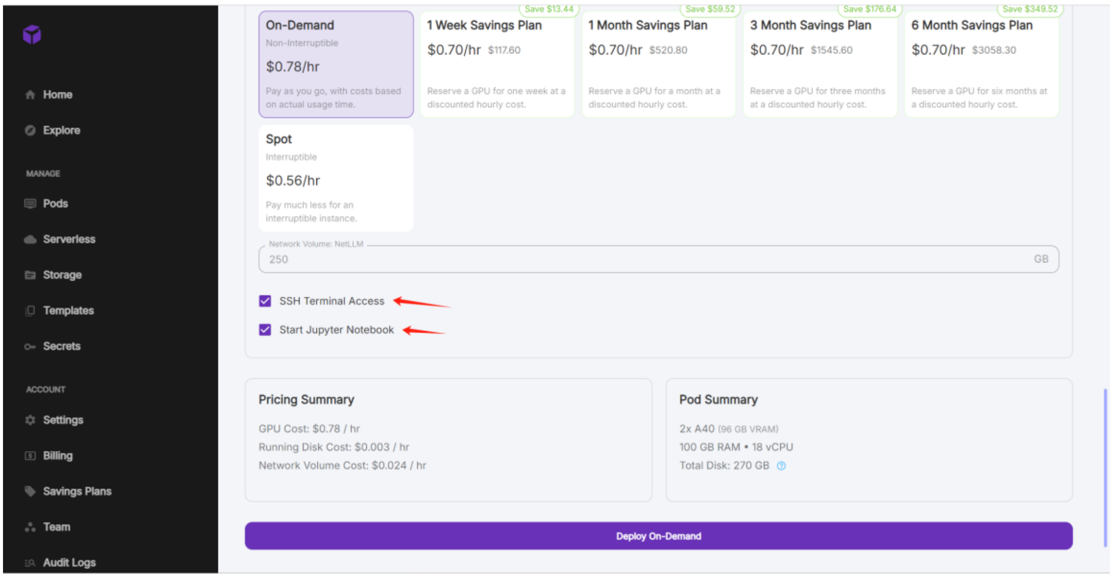


Increase GPU Count from 1 to 2.

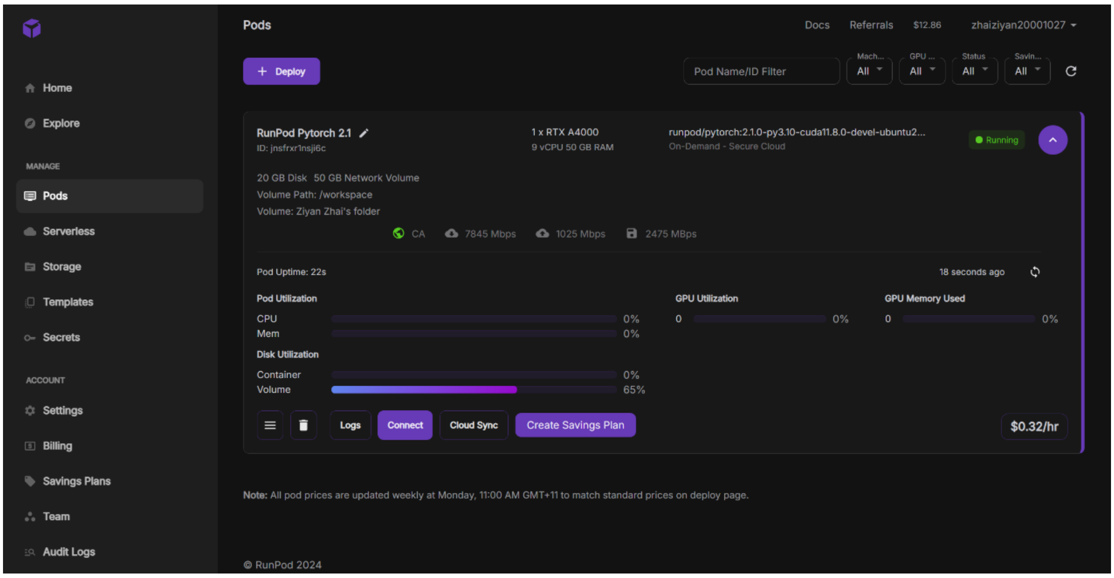




Remember to tick the following two boxes

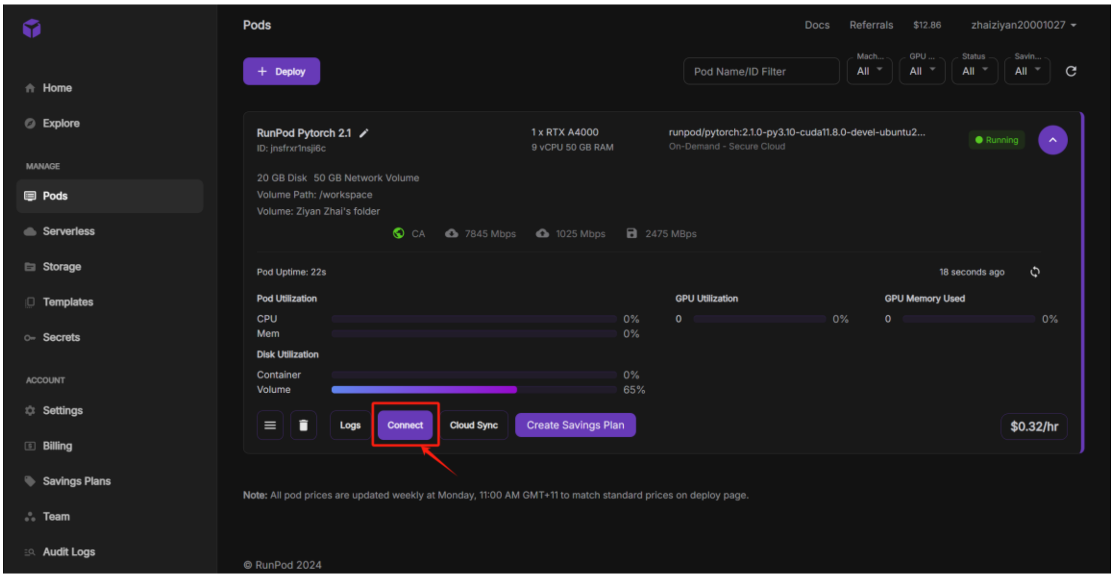


After the deployment is complete, you will see this page. My volume is at 65% because I have already uploaded the NetLLM-related files. If you are using newly created storage, the volume will be at 0.

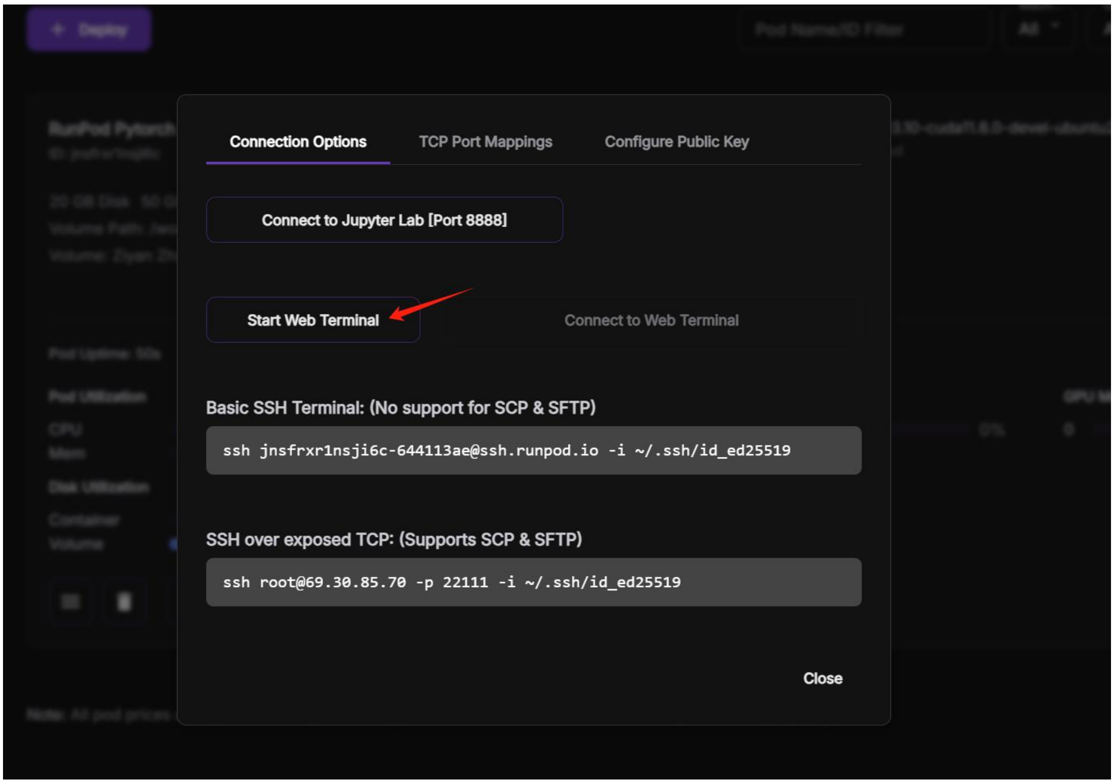


5. Using Runpod

After the deployment is complete, click **Connect**.



On the **Connect** page, click **Start Web Terminal**, then click **Connect to Web Terminal**.



After clicking **Connect to Web Terminal**, Runpod will launch a web terminal. If you see this web terminal, it means you have successfully accessed the Runpod cloud GPU terminal.

Next, use the command: **cd workspace/**

to enter the storage folder. Please ensure that all files are stored in the **workspace** folder. If you store them elsewhere, all files and data will be automatically deleted when you shut down the Pod.

6. Install the necessary components to run NetLLM (Note: Every time you start a new Pod, you will need to reinstall these components, as they won't be saved in the storage.)

You can either install them one by one:

Command:

python -m pip install --upgrade pip

pip install openprompt==1.0.1

pip install numpy==1.24.4

pip install peft==0.6.2

pip install transformers==4.34.1

pip install --upgrade huggingface_hub

pip install scikit-learn

pip install munch

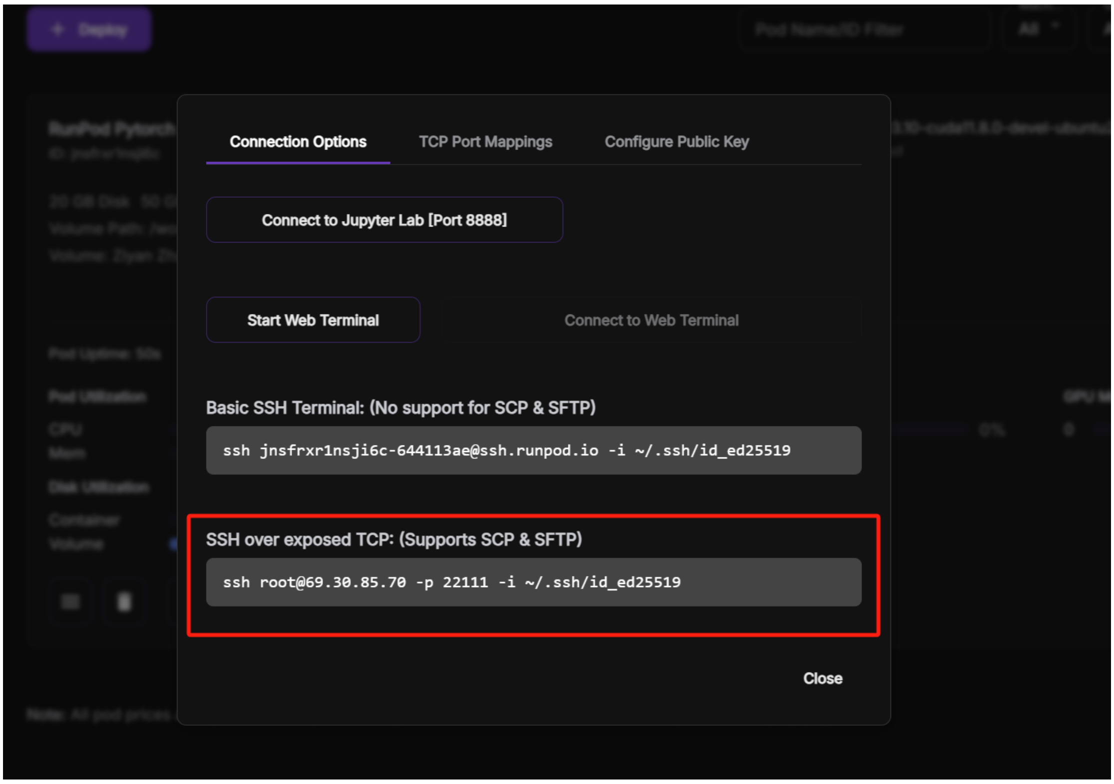
Or run a single command to install them all at once:

Command:

python -m pip install --upgrade pip && pip install openprompt==1.0.1 && pip install numpy==1.24.4 && pip install peft==0.6.2 && pip install transformers==4.34.1 && pip install --upgrade huggingface_hub && pip install scikit-learn && pip install munch

7. Upload the NetLLM architecture files to Runpod's storage

Let's return to the Runpod interface and click **Connect** again to go back to this page. We need to use the SSH provided by the Runpod server to upload the files. Each server will provide a different SSH connection.



Using this SSH as an example:

- **root**: Username for the SSH connection
- **69.30.85.70**: Runpod server's IP address
- **-p 22111**: SSH port number (22111)
- **-i ~/.ssh/id_ed25519**: Specifies the path to your SSH private key file

Command:

scp -i ~/.ssh/id_ed25519 -r -P port_number /path/to/local/folder root@ip_address:/path/to/remote/directory

Now that we have the SSH information, to upload the NetLLM files from your local machine, you just need to enter the command in your local terminal (cmd), replacing **port_number**, **/path/to/local/folder**, **ip_address**, and **/path/to/remote/directory** with your own details. For example, in this case, it should be modified to:

Command:

scp -P 22111 -i ~/.ssh/id_ed25519 -r "D:/NetLLM" root@69.30.85.70:/workspace/

8. How to Use NetLLM

After the upload is complete, first navigate to the **adaptive** folder within the NetLLM folder. The **adaptive** folder contains the main files for the NetLLM architecture.

Command:

cd NetLLM/

cd adaptive_bitrate_streaming

If the PKL file is included in the NetLLM folder, use the following command to start training NetLLM:

Command:

python run_plm.py --adapt --grad-accum-steps 32 --plm-type llama --plm-size base --rank 128 --device cuda:0 --device-out cuda:1 --lr 0.0001 --warmup-steps 2000 --num-epochs 20 --eval-per-epoch 2 --exp-pool-path your_exp_pool_path

Detailed Explanation of the Command Parameters:

python run_plm.py:

- Runs the run_plm.py script, which is the Python file responsible for training the NetLLM model. This file contains the core code for training the language model.

--adapt:

- Enables model adaptation or fine-tuning. This flag indicates that some adaptive training techniques are being used, or the language model is being fine-tuned for a specific task.

--grad-accum-steps 32:

- Specifies that gradient accumulation will occur over 32 steps. Gradient accumulation is often used when GPU memory is limited, meaning the model updates its parameters only after processing 32 mini-batches. This technique allows for larger batch sizes with less memory usage.

--plm-type llama:

- Specifies the type of Pretrained Language Model (PLM) as llama. In this case, we are using Llama2 as the LLM.

--plm-size base:

- Sets the size of the pretrained language model to base. This typically indicates a model with a standard number of parameters. Different sizes (e.g., small, base, large) represent different model complexities and capabilities.

--rank 128:

- Refers to the rank used in low-rank decomposition, which is set to 128. This parameter is used to reduce the computational complexity of model training, especially in techniques like matrix factorization.

--device cuda:0:

- Specifies that the training will run on GPU device cuda:0. This is the first GPU (indexed from 0) that will be used.

--device-out cuda:1:

- Specifies that output-related operations will take place on GPU device cuda:1. Since we are renting 2 A40 GPUs, both cuda:0 and cuda:1 will be used for training.

--lr 0.0001:

- Sets the learning rate to 0.0001. The learning rate controls how much the model's parameters are updated at each step, making it one of the most important hyperparameters for training.

--warmup-steps 2000:

- Specifies 2000 warmup steps at the beginning of the training. This means the learning rate will gradually increase from 0 over the first 2000 steps to avoid instability at the start of training.

--num-epochs 20:

- Sets the total number of epochs to 20. An epoch represents one complete pass through the entire training dataset. You can modify this as needed.

--eval-per-epoch 2:

- Specifies that the model will be evaluated twice per epoch, meaning performance will be assessed twice during each training cycle (e.g., on a validation set).

--exp-pool-path your_exp_pool_path:

- This parameter specifies the location of the PKL file.
-

Since we are currently using the original version of NetLLM, there is no need to provide our own experience pool file. You just need to run this command.

Command:

python run_plm.py --adapt --grad-accum-steps 32 --plm-type llama --plm-size base --rank 128 --device cuda:0 --lr 0.0001 --warmup-steps 2000 --num-epochs 80 --eval-per-epoch 2

If the command runs successfully, you will see information like this (the number of steps depends on the amount of data in your PKL file; the more data, the more steps):

```
Step 0 - mean train loss  1.985231
Step 100 - mean train loss  2.213923
Step 200 - mean train loss  2.189049
Step 300 - mean train loss  2.185949
Step 400 - mean train loss  2.221947
Step 500 - mean train loss  2.228495
Step 600 - mean train loss  2.236243
Step 700 - mean train loss  2.228206
Step 800 - mean train loss  2.220431
Step 900 - mean train loss  2.215590
==================== Training Iteration #0 ====================
>>>>>>>>>> Training Information:
{'time/training': 339.8703103065491,
 'training/train_loss_mean': 2.217822276684175,
 'training/train_loss_std': 0.3987985239587319}
Checkpoint saved at: data/ft_plms/llama_base/artifacts_exp_pools_ss_Non
early_stop_-1_checkpoint/0
```

Please note that completing **iteration 0** does not necessarily mean that there are no issues with your PKL file or the NetLLM program. Be patient and wait for the program to enter **iteration 1**. If **iteration 1** runs smoothly without errors, the rest of the training process is likely to proceed without issues. However, one important point to keep in mind: since we are using **Runpod** for training and the server is located in Canada rather than Australia, there may be **lost connection** issues during the training process. Because NetLLM, unlike other LLMs or deep learning models, cannot resume training from where it was interrupted, it is not recommended to run the NetLLM training program overnight. It's advisable to check the training progress every hour to ensure it hasn't been interrupted.

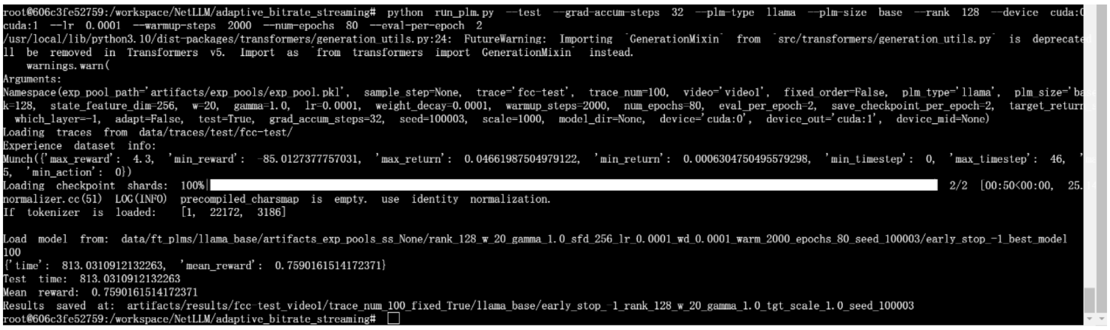
```
/usr/local/lib/python3.10/dist-packages/transformers/generation_utils.py:24: FutureWarning: Importing GenerationMixin from src/transformers/generation_utils.py is deprecate
ll be removed in Transformers v5. Import as `from transformers import GenerationMixin` instead.
  warnings.warn(
Arguments:
Namespace(exp_pool_path='./prague_exp_pool.pkl', sample_step=None, trace='fcc-test', trace_num=100, video='video1', fixed_order=False, plm_type='llama', plm_size='base', rank=
e_feature_dim=256, w=20, gamma=1.0, lr=0.0001, weight_decay=0.0001, warmup_steps=2000, num_epochs=5, eval_per_epoch=2, save_checkpoint_per_epoch=2, target_return_scale=1.0,
=-1, adapt=True, test=False, grad_accum_steps=32, seed=100003, scale=1000, model_dir=None, device='cuda:0', device_out='cuda:1', device_mid=None)
Loading traces from data/traces/test/fcc-test/
Experience dataset info:
Munch({'max_reward': 8.5759033476188, 'min_reward': 6.5074270790278925, 'max_return': 0.9107929413961323, 'min_return': 0.00021968027992012382, 'min_timestep': 0, 'max_times
 'max_action': 3, 'min_action': 1})
Loading checkpoint shards: 100%|████████████████████████████████████████| 2/2 [00:12<00:00,
normalizer.cc(51) LOG(INFO) precompiled_charsmap is empty. use identity normalization.
If tokenizer is loaded: [1, 22172, 3186]

Step 0 - mean train loss  2.940735
Step 100 - mean train loss  2.629836
==================== Training Iteration #0 ====================
>>>>>>>>>> Training Information:
{'time/training': 30.821568965911865,
 'training/train_loss_mean': 2.648814216666265,
 'training/train_loss_std': 0.42323751098136914}
Checkpoint saved at: data/ft_plms/llama_base/._ss_None/rank_128_w_20_gamma_1.0_sfd_256_lr_0.0001_wd_0.0001_warm_2000_epochs_5_seed_100003/early_stop_-1_checkpoint/0
>>>>>>>>>> Evaluation Information
{'best_return': 0.0,
 'episodes_len': 4700,
 'episodes_return': 0.0,
 'time/evaluation': 556.1744978427887}
Step 0 - mean train loss  2.094530
Step 100 - mean train loss  2.641876
==================== Training Iteration #1 ====================
>>>>>>>>>> Training Information:
{'time/training': 29.650806665420532,
 'training/train_loss_mean': 2.6438361023544172,
 'training/train_loss_std': 0.4226540673253825}
Step 0 - mean train loss  2.531451
Step 100 - mean train loss  2.649771
==================== Training Iteration #2 ====================
>>>>>>>>>> Training Information:
{'time/training': 29.55081295967102,
 'training/train_loss_mean': 2.6337102640659436,
 'training/train_loss_std': 0.4222205221205113}
Checkpoint saved at: data/ft_plms/llama_base/._ss_None/rank_128_w_20_gamma_1.0_sfd_256_lr_0.0001_wd_0.0001_warm_2000_epochs_5_seed_100003/early_stop_-1_checkpoint/2
```

After the training is complete, you can enter the following command to run a test.:

python run_plm.py --test --grad-accum-steps 32 --plm-type llama --plm-size base --rank 128 --device cuda:0 --lr 0.0001 --warmup-steps 2000 --num-epochs 80 --eval-per-epoch 2

The output should look something like this.



```
root@606c3fe52759:/workspace/NetLLM/adaptive_bitrate_streaming# python run_plm.py --test --grad-accum-steps 32 --plm-type llama --plm-size base --rank 128 --device cuda:0
cuda:1 --lr 0.0001 --warmup-steps 2000 --num-epochs 80 --eval-per-epoch 2
/usr/local/lib/python3.10/dist-packages/transformers/generation_utils.py:24: FutureWarning: Importing `GenerationMixin` from `src/transformers/generation_utils.py` is deprecated
ll be removed in Transformers v5. Import as `from transformers import GenerationMixin` instead.
  warnings.warn(
Arguments:
Namespace(exp_pool_path='artifacts/exp_pools/exp_pool.pkl', sample_step=None, trace='fcc-test', trace_num=100, video='video1', fixed_order=False, plm_type='llama', plm_size='ba
k=128, state_feature_dim=256, w=20, gamma=1.0, lr=0.0001, weight_decay=0.0001, warmup_steps=2000, num_epochs=80, eval_per_epoch=2, save_checkpoint_per_epoch=2, target_return
 which_layer=-1, adapt=False, test=True, grad_accum_steps=32, seed=100003, scale=1000, model_dir=None, device='cuda:0', device_out='cuda:1', device_mid=None)
Loading traces from data/traces/test/fcc-test/
Experience dataset info:
Munch({'max_reward': 4.3, 'min_reward': -85.0127377757031, 'max_return': 0.04661987504979122, 'min_return': 0.0006304750495579298, 'min_timestep': 0, 'max_timestep': 46, '
5, 'min_action': 0})
Loading checkpoint shards: 100%|██████████| 2/2 [00:50<00:00, 25.
normalizer.cc(51) LOG(INFO) precompiled_charsmap is empty. use identity normalization.
If tokenizer is loaded:  [1, 22172, 3186]

Load model from: data/ft_plms/llama_base/artifacts_exp_pools_ss_None/rank_128_w_20_gamma_1.0_sfd_256_lr_0.0001_wd_0.0001_warm_2000_epochs_80_seed_100003/early_stop_-1_best_model
100
{'time': 813.0310912132263, 'mean_reward': 0.7590161514172371}
Test time: 813.0310912132263
Mean reward: 0.7590161514172371
Results saved at: artifacts/results/fcc-test_video1/trace_num_100_fixed_True/llama_base/early_stop_-1_rank_128_w_20_gamma_1.0_tgt_scale_1.0_seed_100003
root@606c3fe52759:/workspace/NetLLM/adaptive_bitrate_streaming#
```

How to Use My NetLLM Program

If you want to use my NetLLM program, you can download my **adaptive** folder from GitHub and overwrite the original NetLLM folder on your local machine. Then, rerun the **scp** command to upload it to Runpod.

GitHub Link: https://github.com/MPTCP-FreeBSD/LLM-CCA/tree/main

Afterward, the rest of the process is similar to running the original NetLLM. We use the same command to start training, but now you can modify certain parts of the training command.

Suggested modifications:

num-epochs [number]

eval-per-epoch-[number]

--exp-pool-path your_exp_pool_path

For example,

Command:

python run_plm.py --adapt --grad-accum-steps 32 --plm-type llama --plm-size base --rank 128 --device cuda:0 --device-out cuda:1 --lr 0.0001 --warmup-steps 2000 --num-epochs 60 --eval-per-epoch 1 --exp-pool-path ./prague_exp_pool.pkl

Set the total number of training epochs to **60**.

Perform **1** evaluation per training epoch.

Use the **pragueexp_pool.pkl** file for the experience pool.

Step 5: Explanation of My Modifications to the Original NetLLM

1. Experience Pool Generation Script

In the official version of NetLLM, the experience pool files are generated using formulas and algorithms to simulate data. However, since we conducted real experiments and collected actual data, there is no need to generate the experience pool through simulation. To create our own experience pool file, I wrote a script. All you need to do is convert the data previously collected using **iperf3** from a **json** file into a **csv** file, ensuring it includes **RTT**, **throughput**, and **CWND** data.

In my experience pool file, I use **RTT**, **throughput**, and **CWND** as the **state**, and **CWND** as the **action** (since CWND controls the sender's bitrate, we want NetLLM to learn how to control CWND). The **reward** value is calculated as **throughput/RTT**, because when **throughput** is high and **RTT** is low, the reward increases; conversely, when **RTT** is high and **throughput** is low, the reward decreases. Then, we smooth the reward value using **log10**.

Additionally, we normalized the **CWND** values to ensure that NetLLM can correctly interpret the **action** values (without normalization, the **mean train loss** reported by NetLLM during training could be very high, which would hinder the model's learning).

```python
def run(args):
    df = pd.read_csv(args.csv_file)
    exp_pool = ExperiencePool()

    # 找到最大和最小的 cwnd 值
    max_cwnd_value = df['cwnd'].max()
    min_cwnd_value = df['cwnd'].min()

    for index, row in df.iterrows():
        rtt = float(row['rtt']) if pd.notna(row['rtt']) else 0
        cwnd = float(row['cwnd']) if pd.notna(row['cwnd']) else 0
        throughput = float(row['throughput']) if pd.notna(row['throughput']) else 0

        # 将 state 转换为 Tensor，只包含 [rtt, cwnd, throughput]
        state = torch.tensor([rtt, cwnd, throughput], dtype=torch.float32)

        # 将 cwnd 缩放到 [1, 2, 3] 范围
        # 先进行归一化，再映射到 [1, 2, 3] 范围
        action_normalized = (cwnd - min_cwnd_value) / (max_cwnd_value - min_cwnd_value)
        action = int(action_normalized * 2) + 1  # 映射到 1, 2, 3

        # 计算 reward
        reward = calculate_reward(rtt, throughput)
        done = index == len(df) - 1

        # 将 state, action, reward, done 加入经验池
        exp_pool.add(state, action, reward, done)

    # 保存 ExperiencePool 实例
    exp_pool_path = os.path.join(args.output_dir, 'prague_exp_pool.pkl')

    with open(exp_pool_path, 'wb') as f:
        pickle.dump(exp_pool, f)

    print(f"经验池已保存至: {exp_pool_path}")
```

2. run_plm.py

Originally, run_plm used **cross entropy** as the loss function, which is typically used for classification tasks. However, since our action is **CWND**, which is a continuous value, we needed to change the loss function to **MSELoss** (Mean Squared Error Loss).

```python
# 适应函数 adapt() 中的修改
def adapt(args, model, exp_dataset, exp_dataset_info, eval_env_settings, checkpoint_dir, best_model_dir, eval_process_reward_fn):
    optimizer = AdamW(
        model.parameters(),
        lr=args.lr,
        weight_decay=args.weight_decay,
    )
    lr_scheduler = LambdaLR(
        optimizer,
        lambda steps: min((steps + 1) / args.warmup_steps, 1)
    )
    loss_fn = MSELoss()  #MSELoss可以修改为Cross-Entropy Loss CE Loss | MSELoss can be replaced with Cross-Entropy Loss (CE Loss).
    trainer = Trainer(args, model=model, optimizer=optimizer, exp_dataset=exp_dataset, loss_fn=loss_fn, device=args.device, lr_schedu
                      grad_accum_steps=args.grad_accum_steps)

    target_return = exp_dataset_info.max_return * args.target_return_scale
    best_eval_return = 0.

    total_train_losses = []
    min_loss=np.inf
```

3. trainer.py

The trainer.py file was modified to calculate **MSE loss**. The **Trainer** class is responsible for managing the model's training process. It uses data from the experience pool to compute the loss via forward propagation, and then updates the model's parameters through backpropagation. Gradient accumulation is implemented to improve efficiency while maintaining small batch sizes.

```python
adaptive_bitrate_streaming > plm_special > trainer.py > Trainer > train_step
class Trainer:
    def train_epoch(self, report_loss_per_steps=100):
                # perform gradient accumulation update
                train_loss = train_loss / self.grad_accum_steps
                train_loss.backward()
                torch.nn.utils.clip_grad_norm_(self.model.parameters(), .25)
                if ((step + 1) % self.grad_accum_steps == 0) or (step + 1 == dataset_size):
                    self.optimizer.step()
                    self.optimizer.zero_grad(set_to_none=True)
                    if self.lr_scheduler is not None:
                        self.lr_scheduler.step()

                if step % 2 == 0:
                    mean_train_loss = np.mean(train_losses)
                    print(f'Step {step} - mean train loss {mean_train_loss:>9f}')

            logs['time/training'] = time.time() - train_start
            logs['training/train_loss_mean'] = np.mean(train_losses)
            logs['training/train_loss_std'] = np.std(train_losses)

            return logs, train_losses

    def train_step(self, batch):
        states, actions, returns, timesteps, labels = process_batch(batch, device=self.device)

        # 前向传播，获取预测的 actions_pred
        actions_pred = self.model(states, actions, returns, timesteps)

        # 调整 actions_pred 的形状，使其与 labels 形状一致
        actions_pred = actions_pred.squeeze(-1)  # 将形状从 (1, 20, 1) 调整为 (1, 20)

        labels = labels.float()  # 确保 labels 是 float 类型

        # 计算 MSE 损失
        loss = self.loss_fn(actions_pred, labels)

        return loss
```

4. state_encoder.py

The state_encoder was modified to encode numerical features such as **RTT**, **CWND**, and **throughput** into high-dimensional vectors. Each feature is mapped to the **embed_dim** dimension through a linear layer. These encoded vectors are then concatenated and passed through a fully connected layer to generate the final embedded vector output. This encoder can be part of a larger model, helping **NetLLM** or other models understand the relationships between these network performance indicators.

adaptive_bitrate_streaming > plm_special > models > state_encoder.py > EncoderNetwork > forward

```python
import torch.nn as nn
import torch

class EncoderNetwork(nn.Module):
    def __init__(self, embed_dim=128):
        super().__init__()
        self.embed_dim = embed_dim

        # Linear layers for numerical features
        self.rtt_fc = nn.Linear(1, embed_dim)  # For RTT
        self.cwnd_fc = nn.Linear(1, embed_dim)  # For CWND
        self.throughput_fc = nn.Linear(1, embed_dim)  # For Throughput

        # Final fully connected layer to combine all features
        self.fc_final = nn.Linear(embed_dim * 3, embed_dim)  # 3 inputs: RTT, CWND, Throughput

    def forward(self, state):
        # Extract the components from the state
        rtt = state[..., 0].unsqueeze(-1)  # RTT
        cwnd = state[..., 1].unsqueeze(-1)  # CWND
        throughput = state[..., 2].unsqueeze(-1)  # Throughput

        # Process numerical features
        rtt_encoded = torch.relu(self.rtt_fc(rtt))
        cwnd_encoded = torch.relu(self.cwnd_fc(cwnd))
        throughput_encoded = torch.relu(self.throughput_fc(throughput))

        # Concatenate all encoded features
        combined = torch.cat([rtt_encoded, cwnd_encoded, throughput_encoded], dim=-1)

        # Final output
        output = torch.relu(self.fc_final(combined))

        return output
```

5. rl_policy.py

The original rl_policy was modified, and a policy network class for **Offline Reinforcement Learning (Offline RL)** was defined, named **OfflineRLPolicy**. This class uses a **Multimodal Encoder** to process network states (such as **RTT**, **CWND**, and **throughput**) along with **PLM** (Pretrained Language Model) embeddings. Through this network, it is possible to learn how to predict **CWND** (Congestion Window) values in network control tasks based on state and historical information.

adaptive_bitrate_streaming > plm_special > models > rl_policy.py > ...

```python
INF = 1e5

class OfflineRLPolicy(nn.Module):
    def __init__(
            self,
            state_feature_dim,
            bitrate_levels,
            state_encoder,  # 用于处理 [RTT, CWND, Throughput]
            plm,
            plm_embed_size,
            max_length=None,
            max_ep_len=100,
            device='cuda' if torch.cuda.is_available() else 'cpu',
            device_out=None,
            residual=False,
            which_layer=-1,  # 早停
            **kwargs
    ):
        super().__init__()

        if device_out is None:
            device_out = device

        self.bitrate_levels = bitrate_levels
        self.max_length = max_length

        self.plm = plm
        self.plm_embed_size = plm_embed_size

        # =========== Multimodal Encoder (start) ===========
        self.state_encoder = state_encoder
        self.state_feature_dim = state_feature_dim
        self.embed_timestep = nn.Embedding(max_ep_len + 1, plm_embed_size).to(device)
        self.embed_return = nn.Linear(1, plm_embed_size).to(device)
        self.embed_action = nn.Linear(1, plm_embed_size).to(device)
        self.embed_state1 = nn.Linear(state_feature_dim, plm_embed_size).to(device)  # 状态编码器

        self.embed_ln = nn.LayerNorm(plm_embed_size).to(device)
        # =========== Multimodal Encoder (end) ===========
```