```
import torch, sys, os
from core.utils import tasks, networks, learners, criterions
from core.logger import Logger
from backpack import backpack, extend
sys.path.insert(1, os.getcwd())
from HesScale.hesscale import HesScale
from core.network.gate import GateLayer, GateLayerGrad
import signal
import traceback
import time
from functools import partial

def signal_handler(msg, signal, frame):
    print('Exit signal: ', signal)
    cmd, learner = msg
    with open(f'timeout_{learner}.txt', 'a') as f:
        f.write(f"{cmd} \n")
    exit(0)

class RunStats:
    name = 'run_stats'
    def __init__(self, n_samples=10000, task=None, learner=None, save_path="logs", seed=0,
 network=None, **kwargs):
        self.n_samples = int(n_samples)
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.task = tasks[task]()
        self.task_name = task
        self.learner = learners[learner](networks[network], kwargs)
        self.logger = Logger(save_path)
        self.seed = int(seed)

    def start(self):
        torch.manual_seed(self.seed)
        losses_per_task = []
        plasticity_per_task = []
        n_dead_units_per_task = []
        weight_rank_per_task = []
        weight_l2_per_task = []
        weight_l1_per_task = []
        grad_l2_per_task = []
        grad_l1_per_task = []
        grad_l0_per_task = []

        if self.task.criterion == 'cross_entropy':
            accuracy_per_task = []
        self.learner.set_task(self.task)
        if self.learner.extend:
            extension = HesScale()
            extension.set_module_extension(GateLayer, GateLayerGrad())
        criterion = extend(criterions[self.task.criterion]()) if self.learner.extend else
criterions[self.task.criterion]()
        optimizer = self.learner.optimizer(
            self.learner.parameters, **self.learner.optim_kwargs
        )

        losses_per_step = []
        plasticity_per_step = []
        n_dead_units_per_step = []
        weight_rank_per_step = []
        weight_l2_per_step = []
        weight_l1_per_step = []
```

```
        grad_l2_per_step = []
        grad_l1_per_step = []
        grad_l0_per_step = []

        if self.task.criterion == 'cross_entropy':
            accuracy_per_step = []

        for i in range(self.n_samples):
            input, target = next(self.task)
            input, target = input.to(self.device), target.to(self.device)
            optimizer.zero_grad()
            output = self.learner.predict(input)
            loss = criterion(output, target)
            if self.learner.extend:
                with backpack(extension):
                    loss.backward()
            else:
                loss.backward()
            optimizer.step()
            losses_per_step.append(loss.item())
            if self.task.criterion == 'cross_entropy':
                accuracy_per_step.append((output.argmax(dim=1) == target).float().mean().i
tem())

            # compute some statistics after each task change
            with torch.no_grad():
                output_new = self.learner.predict(input)
                loss_after = criterion(output_new, target)
                loss_before = torch.clamp(loss, min=1e-8)
                plasticity_per_step.append(torch.clamp((1-loss_after/loss_before), min=0.0
, max=1.0).item())
            n_dead_units = 0
            for _, value in self.learner.network.activations.items():
                n_dead_units += value
            n_dead_units_per_step.append(n_dead_units / self.learner.network.n_units)

            sample_weight_rank = 0.0
            sample_max_rank = 0.0
            sample_weight_l2 = 0.0
            sample_grad_l2 = 0.0
            sample_weight_l1 = 0.0
            sample_grad_l1 = 0.0
            sample_grad_l0 = 0.0
            sample_n_weights = 0.0

            for name, param in self.learner.network.named_parameters():
                if 'weight' in name:
                    if 'conv' in name:
                        sample_weight_rank += torch.torch.linalg.matrix_rank(param.data).f
loat().mean()
                        sample_max_rank += torch.min(torch.tensor(param.data.shape)[-2:])
                    else:
                        sample_weight_rank += torch.linalg.matrix_rank(param.data)
                        sample_max_rank += torch.min(torch.tensor(param.data.shape))
                    sample_weight_l2 += torch.norm(param.data, p=2) ** 2
                    sample_weight_l1 += torch.norm(param.data, p=1)

                    sample_grad_l2 += torch.norm(param.grad.data, p=2) ** 2
                    sample_grad_l1 += torch.norm(param.grad.data, p=1)

                    sample_grad_l0 += torch.norm(param.grad.data, p=0)
```

```
                    sample_n_weights += torch.numel(param.data)

            weight_l2_per_step.append(sample_weight_l2.sqrt().item())
            weight_l1_per_step.append(sample_weight_l1.item())
            grad_l2_per_step.append(sample_grad_l2.sqrt().item())
            grad_l1_per_step.append(sample_grad_l1.item())
            grad_l0_per_step.append(sample_grad_l0.item()/sample_n_weights)
            weight_rank_per_step.append(sample_weight_rank.item() / sample_max_rank.item()
)


            if i % self.task.change_freq == 0:
                losses_per_task.append(sum(losses_per_step) / len(losses_per_step))
                if self.task.criterion == 'cross_entropy':
                    accuracy_per_task.append(sum(accuracy_per_step) / len(accuracy_per_ste
p))
                plasticity_per_task.append(sum(plasticity_per_step) / len(plasticity_per_s
tep))
                n_dead_units_per_task.append(sum(n_dead_units_per_step) / len(n_dead_units
_per_step))
                weight_rank_per_task.append(sum(weight_rank_per_step) / len(weight_rank_pe
r_step))
                weight_l2_per_task.append(sum(weight_l2_per_step) / len(weight_l2_per_step
))
                weight_l1_per_task.append(sum(weight_l1_per_step) / len(weight_l1_per_step
))
                grad_l2_per_task.append(sum(grad_l2_per_step) / len(grad_l2_per_step))
                grad_l1_per_task.append(sum(grad_l1_per_step) / len(grad_l1_per_step))
                grad_l0_per_task.append(sum(grad_l0_per_step) / len(grad_l0_per_step))

                losses_per_step = []
                if self.task.criterion == 'cross_entropy':
                    accuracy_per_step = []
                plasticity_per_step = []
                n_dead_units_per_step = []
                weight_rank_per_step = []
                weight_l2_per_step = []
                weight_l1_per_step = []
                grad_l2_per_step = []
                grad_l1_per_step = []
                grad_l0_per_step = []


        if self.task.criterion == 'cross_entropy':
            self.logger.log(losses=losses_per_task,
                            accuracies=accuracy_per_task,
                            plasticity_per_task=plasticity_per_task,
                            task=self.task_name,
                            learner=self.learner.name,
                            network=self.learner.network.name,
                            optimizer_hps=self.learner.optim_kwargs,
                            n_samples=self.n_samples,
                            seed=self.seed,
                            n_dead_units_per_task=n_dead_units_per_task,
                            weight_rank_per_task=weight_rank_per_task,
                            weight_l2_per_task=weight_l2_per_task,
                            weight_l1_per_task=weight_l1_per_task,
                            grad_l2_per_task=grad_l2_per_task,
                            grad_l0_per_task=grad_l0_per_task,
                            grad_l1_per_task=grad_l1_per_task,
            )
```

```
        else:
            self.logger.log(losses=losses_per_task,
                            plasticity=plasticity_per_task,
                            task=self.task_name,
                            learner=self.learner.name,
                            network=self.learner.network.name,
                            optimizer_hps=self.learner.optim_kwargs,
                            n_samples=self.n_samples,
                            seed=self.seed,
                            n_dead_units_per_task=n_dead_units_per_task,
                            weight_rank_per_task=weight_rank_per_task,
                            weight_l2_per_task=weight_l2_per_task,
                            weight_l1_per_task=weight_l1_per_task,
                            grad_l2_per_task=grad_l2_per_task,
                            grad_l0_per_task=grad_l0_per_task,
                            grad_l1_per_task=grad_l1_per_task,
            )


if __name__ == "__main__":
    # start the run using the command line arguments
    ll = sys.argv[1:]
    args = {k[2:]:v for k,v in zip(ll[::2], ll[1::2])}
    run = RunStats(**args)
    cmd = f"python3 {' '.join(sys.argv)}"
    signal.signal(signal.SIGUSR1, partial(signal_handler, (cmd, args['learner'])))
    current_time = time.time()
    try:
        run.start()
        with open(f"finished_{args['learner']}.txt", "a") as f:
            f.write(f"{cmd} time_elapsed: {time.time()-current_time} \n")
    except Exception as e:
        with open(f"failed_{args['learner']}.txt", "a") as f:
            f.write(f"{cmd} \n")
        with open(f"failed_{args['learner']}_msgs.txt", "a") as f:
            f.write(f"{cmd} \n")
            f.write(f"{traceback.format_exc()} \n\n")
```