Nirbhay sharma (B19CSE114)

Regularizing Neural Networks via Adversarial Model Perturbations

---

How to Run

Installing requirements

```
pip install -r requirements.txt
```

Run for CIFAR10 dataset

```
CUDA_VISIBLE_DEVICES=<gpu_id> python main.py --dataset cifar10 --model
<model_name>
```

Run for gtsrb dataset

```
CUDA_VISIBLE_DEVICES=<gpu_id> python main.py --dataset gtsrb --model
<model_name>
```

Execution flow of the code

The algorithm is outlined below.

Algorithm 1 Adversarial Model Perturbation Training

Require: Training set $\mathcal{D} = \{(\boldsymbol{x}, \boldsymbol{y})\}$, Batch size m, Loss function ℓ, Initial model parameter $\boldsymbol{\theta}_0$, Outer learning rate η, Inner learning rate ζ, Inner iteration number N, L_2 norm ball radius ϵ

while $\boldsymbol{\theta}_k$ not converged **do**
    Update iteration: $k \leftarrow k + 1$
    Sample $\mathcal{B} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^m$ from training set $\mathcal{D}$
    Initialize perturbation: $\Delta_\mathcal{B} \leftarrow \mathbf{0}$
    for $n \leftarrow 1$ to N **do**
        Compute gradient:
            $\nabla \mathcal{J}_{\text{AMP},\mathcal{B}} \leftarrow \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{x}_i, \boldsymbol{y}_i; \boldsymbol{\theta}_k + \Delta_\mathcal{B})/m$
        Update perturbation: $\Delta_\mathcal{B} \leftarrow \Delta_\mathcal{B} + \zeta \nabla \mathcal{J}_{\text{AMP},\mathcal{B}}$
        if $\|\Delta_\mathcal{B}\|_2 > \epsilon$ **then**
            Normalize perturbation: $\Delta_\mathcal{B} \leftarrow \epsilon \Delta_\mathcal{B} / \|\Delta_\mathcal{B}\|_2$
        end if
    end for
    Compute gradient:
        $\nabla \mathcal{J}_{\text{AMP},\mathcal{B}} \leftarrow \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{x}_i, \boldsymbol{y}_i; \boldsymbol{\theta}_k + \Delta_\mathcal{B})/m$
    Update parameter: $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \eta \nabla \mathcal{J}_{\text{AMP},\mathcal{B}}$
end while

The whole problem can be seen as a minimization problem of the $max_{\delta \in B(\mu,\epsilon)}(L_{EMP}(x, y, \theta + \delta))$ so the first task is find a suitable δ which lies in L_2 ball with radius ϵ and then simply minimizing it over model parameters θ would do the work.

so the entire algorithm is as follows:

- first initialize the parameters (θ, δ)
- run for specific iterations (N) to obtain δ using gradient ascent
  - calculate gradient of loss as $\sum \nabla_\theta l(x, y; \theta + \delta)$
  - update δ as $\delta = \delta + \eta \sum \nabla_\theta l(x, y; \theta + \delta)$
- once δ is obtained again compute the gradient using the same formula and update the model parameters as follows
  - $\theta = \theta - \eta \sum \nabla_\theta l(x, y; \theta + \delta)$
- The entire workflow first finds a suitable delta in L_2 ball and then update the model parameters accordingly.

The authors have implemented this as an optimization algorithm known as AMP in the code as follows.

```
@torch.no_grad()
def step(self, closure=None):
    if closure is None:
        raise ValueError('Adversarial model perturbation requires closure, but it was not provided')
    closure = torch.enable_grad()(closure)
    outputs, loss = map(lambda x: x.detach(), closure())
    for i in range(self.defaults['inner_iter']):
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is not None:
                    if i == 0:
                        self.state[p]['dev'] = torch.zeros_like(p.grad)
                    dev = self.state[p]['dev'] + group['inner_lr'] * p.grad
                    clip_coef = group['epsilon'] / (dev.norm() + 1e-12)
                    dev = clip_coef * dev if clip_coef < 1 else dev
                    p.sub_(self.state[p]['dev']).add_(dev) # update "theta" with "theta+delta"
                    self.state[p]['dev'] = dev
        closure()
    for group in self.param_groups:
        for p in group['params']:
            if p.grad is not None:
                p.sub_(self.state[p]['dev']) # restore "theta" from "theta+delta"
    self.base_optimizer.step()
    return outputs, loss
```

Here authors have done the same thing as described above, first they find out the suitable delta to use as a regularizer and then they use standard optimization algorithm (SGD) to finally update the model parameters.