Deep Learning

Exercise 8: Open-Set Learning

Room: **BIN-1-B.01**
Instructor: Manuel Günther
Email: guenther@ifi.uzh.ch
Office: AND 2.54

Friday, April 29, 2022

Outline

Outline

1. PyTorch
   - Network Implementation
   - Gradient Definition

Network Implementation

Modules in PyTorch

- A module can be
  - → A separate layer (e.g. Linear, ReLU, ...)
  - → A block of layers (e.g. ResNet Block)
  - → A complete network (e.g. LeNet, ResNet)

⇒ A network is a module

- Implemented as `torch.nn.Module`

Defining a Module

- Implemented as class (e.g. `Network`)
- Derive from `torch.nn.Module`:

  `class Network(torch.nn.Module):`
- Two functions to implement
  - → Constructor: `def __init__(self, ...)`
  - → Forward function: `def forward(self, x)`

Network Implementation

The Constructor `def __init__(self, ...)`

- Call base class constructor:
  - → `super(Network, self).__init__(...)`
- Instantiate all submodules:
  - → `self.conv1 = torch.nn.Conv2d(...)`
  - → `self.pooling = torch.nn.MaxPool2d(...)`
  - → `self.activation = torch.nn.ReLU(...)`
  - → `self.fc1 = torch.nn.Linear(...)`
- If needed: define variables as `Parameter`, similar to `torch.Tensor`
  - → `self.my_param = torch.nn.Parameter(...)`
- All these are automatically extracted by `self.parameters()`

Network Implementation

The Forward Function `def forward(self, x)`

- Implement all processing steps of your network
  - → `x = self.conv1(x)`
  - → `x = self.pooling(x)`
  - → `x = self.activation(x)`
- Can be grouped into (logical) blocks
  - → `layer1 = self.activation(self.pooling(self.conv1(x)))`
- Can also use other non-parametric functions
  - → `flattened = torch.flatten(layer1)`
- Can have multiple outputs (be aware of inplace functions)
  - → `logits = self.fc1(flattened)`
  - → `return logits, flattened`

Gradient Definition

Processing of Derivatives in PyTorch

- Usually handled by automatic differentiation
- Defined for each operation in PyTorch
  - $\rightarrow$ Enabled when using PyTorch functionality throughout
- Might not be optimal
  - $\rightarrow$ For example $\mathcal{J}^{\mathrm{CCE}}$ on SoftMax has simple gradient

Implement your own Derivatives (Jacobian)

- Implement a `torch.autograd.Function`
  - $\rightarrow$ https://pytorch.org/docs/master/notes/extending.html#extending-torch-autograd
- Define `forward` function with several inputs
- Provide a Jacobian for each of the inputs in `backward`

Gradient Definition

The `forward` Function

- Is defined as static method via `@staticmethod` decorator
  → Belongs to the class, not to the object
- Takes all parameters that your operation requires

```
@staticmethod
def forward(ctx, param1, param2, ...):
```

- Provides context information via `ctx`
  → Can store required variables to the `backward` function

```
ctx.save_for_backward(param1, param2)
```

- Returns the result of your operation

```
result = operation(param1, param2)
return result
```

Gradient Definition

The `backward` Function

- Also defined as static method via `@staticmethod` decorator
- Has two parameters: context `ctx`, result of `forward`

```
@staticmethod
def backward(ctx, result):
```

- Extract stored information from context

```
param1, param2 = ctx.saved_tensors
```

- Return Jacobian for **each input of** `forward`
  - → Need to be of same shape as input parameters; not exactly the Jacobian
  - → Can be `None` if derivative for one parameter makes no sense

```
derivative_for_param1 = ...
return derivative_for_param1, None
```

Outline

2. Open-Set Training
   - Dataset
   - Loss Function and Confidence
   - Network and Training
   - Evaluation

Dataset

MNIST Dataset

- MNIST total 10 classes
  - → 10 different digits
- Split into 3 categories:
  - → 4 known classes, e.g.: `4, 5, 8, 9`
  - → 4 known unknown classes, e.g.: `0, 2, 3, 7`
  - → 2 unknown unknown classes, e.g.: `1, 6`
- Split into three subsets
  - → Training partition: training sets of known classes and known unknown
  - → Validation partition: test sets of known classes and known unknown
  - → Test partition: test sets of known classes and unknown unknown

Dataset

Task 1: Target Vectors

- Provide target vectors $\vec{t}$ for target class t
- One-hot targets for knowns:
  - $\rightarrow$ 4 $\Rightarrow$ (1,0,0,0)
  - $\rightarrow$ 5 $\Rightarrow$ (0,1,0,0)
  - $\rightarrow$ 8 $\Rightarrow$ (0,0,1,0)
  - $\rightarrow$ 9 $\Rightarrow$ (0,0,0,1)
- Equal targets for unknowns:
  - $\rightarrow$ 0, 2, 3, 7, 1, 6 $\Rightarrow$ (.25,.25,.25,.25)

Test 1: Check Target Vectors

- Implement test case to check that the target vectors are correct

Dataset

Task 2: Dataset Construction

- Extend `torchvision.datasets.MNIST` dataset class
- Implement constructor for our class `__init__(self, purpose)`
- Call base class constructor with parameters based on `purpose`
  - → This populates `self.data` and `self.targets` for all 10 classes
- Select samples of the desired classes only (for our `purpose`)

Task 3: Dataset Item Selection

- Implement the `__getitem__(self, index)` function
- Return image at `index` in desired format
- Return target vector at `index` as required (Task 1)

Dataset

Test 2: Data Sets

- Instantiate data loader for training split with $B = 64$
- Assert that inputs and targets are in the desired format

Task 4: Utility Function

- Write a function that takes a batch and the targets
- Split the batch into known and unknown
- Return `batch[known]`, `targets[known]`, `batch[unknown]`

  $\rightarrow$ This function will be used several times later

Loss Function and Confidence

Task 5: Loss Function

- Implement forward pass
  - $\rightarrow$ Choose one of the two methods
- Store required variables
- Implement Jacobian in backward pass

Task 5a: Alternative Loss

- Implement as loss function
  - $\rightarrow$ Choose one of the two methods
  - $\rightarrow$ Use `torch.log_softmax` or `torch.logsumexp`

Possible Implementation

$$\mathcal{J}^{\text{CCE}} = -\sum_{o=1}^{O} t_o \ln y_o$$

Other Implementation

$$\mathcal{J}^{\text{CCE}} = -\sum_{o=1}^{O} t_o z_o + \frac{1}{O} \ln \sum_{o=1}^{O} e^{z_o}$$

Derivative (Jacobian)

$$\frac{\partial \mathcal{J}}{\partial z_o} = y_o - t_o$$

Loss Function and Confidence

Task 6: Confidence Evaluation

- Implement `confidence(logits, targets)` function
- Compute the SoftMax confidence scores $\vec{y}$ based on the `logits`
- Split data into known and unknown
- For known, take confidence of correct class: $\mathrm{conf} = y_t$
- For unknown, use maximum over all classes: $\mathrm{conf} = 1 - \max\limits_{o} y_o + \frac{1}{O}$

Test 3: Check Confidence Implementation

- Generate optimal logit values for known and unknown classes
- Get appropriate target vectors (see Task 1)
- Check that the computed confidence is close to 1

Network and Training

Task 7: Network Definition

- Implement small convolutional network
  - → Two convolutional layers with max pooling and ReLU
  - → Two fully-connected layers
- Return output of both FC layers: deep features and logits

Task 8: Training and Valiation Loop

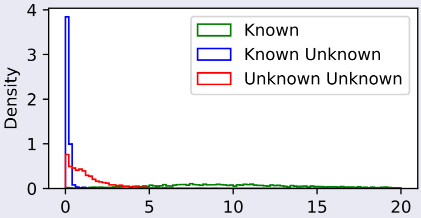
- Use SGD optimizer with reasonable learning rate
- Use self-defined loss function
- Compute confidence values for training and validation set
- Train for 10 (or 100) epochs

Evaluation

Task 9: Featue Magnitude Histograms

- Extract deep features for validation and test set
- Compute deep feature magnitude for all features
  - → Split into known, known unknown and unknown unknown samples
- Plot histograms for all three types of samples

Example Histogram Plot



Evaluation

Task 10: Classification

- Extract softmax confidence score for test set samples
  - $\rightarrow$ Split into known and unknown
- Select confidence threshold τ
- Compute CCR for known
- Compute FPR for unknown

$\Rightarrow$ Good values are CCR $> 90\%$ for FPR $< 10\%$

  - $\rightarrow$ Maybe change threshold τ

False Positive Rate (FPR)

$$\frac{\left|\left\{x^{[n]} \mid t^{[n]} = 0 \wedge \max\limits_{1 \leq o \leq O} y_o^{[n]} \geq \tau\right\}\right|}{\left|\{x^{[n]} \mid t^{[n]} = 0\}\right|}$$

Correct Classification Rate (CCR)

$$\frac{\left|\left\{x^{[n]} \mid t^{[n]} > 0 \wedge \operatorname*{arg\,max}\limits_{1 \leq o \leq O} y_o^{[n]} = t^{[n]} \wedge y_{t^{[n]}}^{[n]} \geq \tau\right\}\right|}{\left|\{x^{[n]} \mid t^{[n]} > 0\}\right|}$$