Scallop and Neuro-Symbolic Programming

Lecture 3: Scallop, with Neural Networks

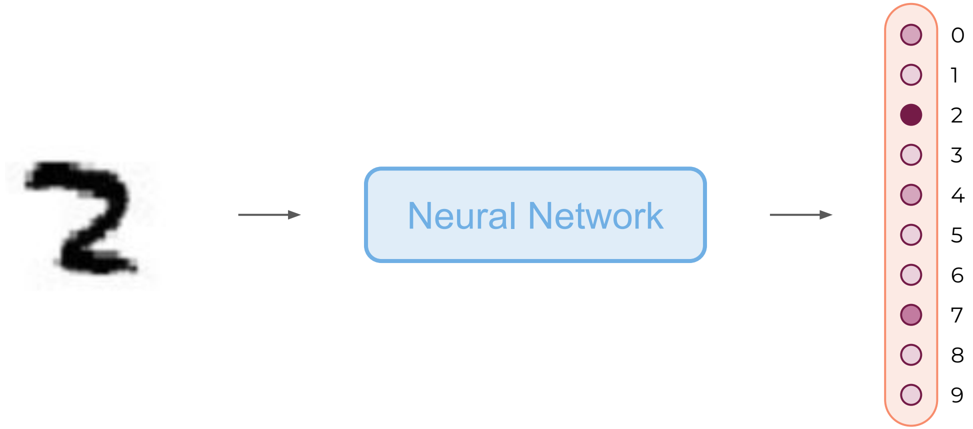
Agenda

1. **Machine Learning Crash Course**
   Training Loop, Back-Propagation, Intro to PyTorch
2. **Scallop and Differentiable Reasoning**
   Enhanced Training Loop, More Provenance Semiring
3. **Scallop with PyTorch**
   Working with MNIST Sum 2 Example

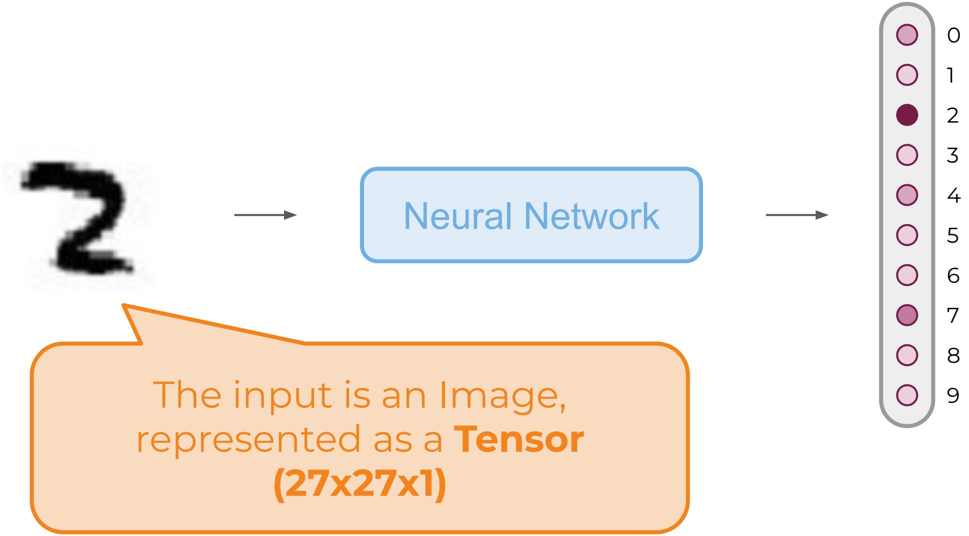
Machine Learning Crash Course

2022 Scallop Language Group, #3

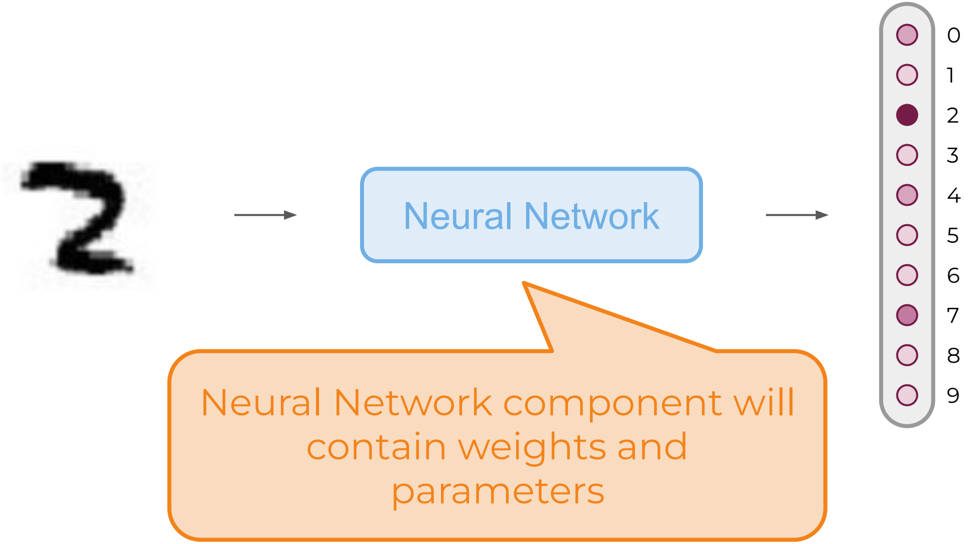
Classification Task (MNIST)



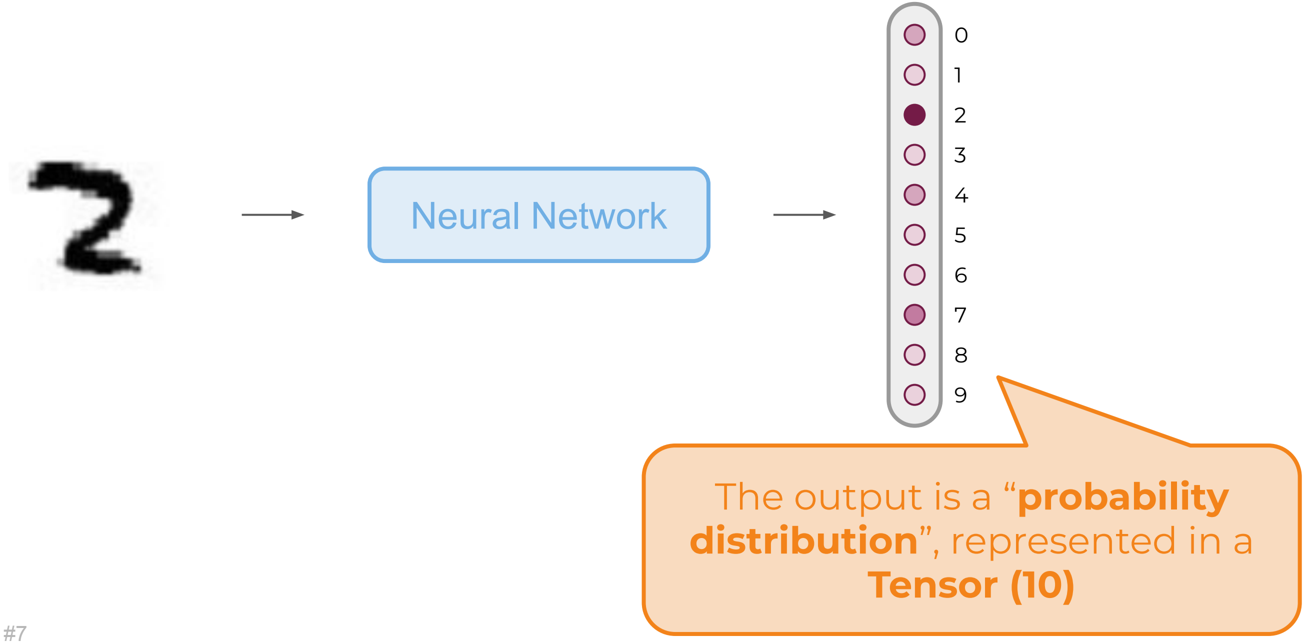
Classification Task (MNIST)



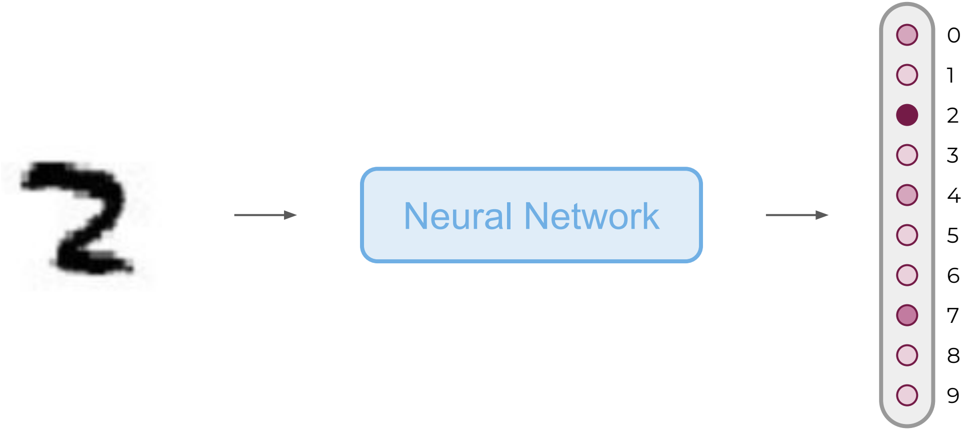
Classification Task (MNIST)



Classification Task (MNIST)



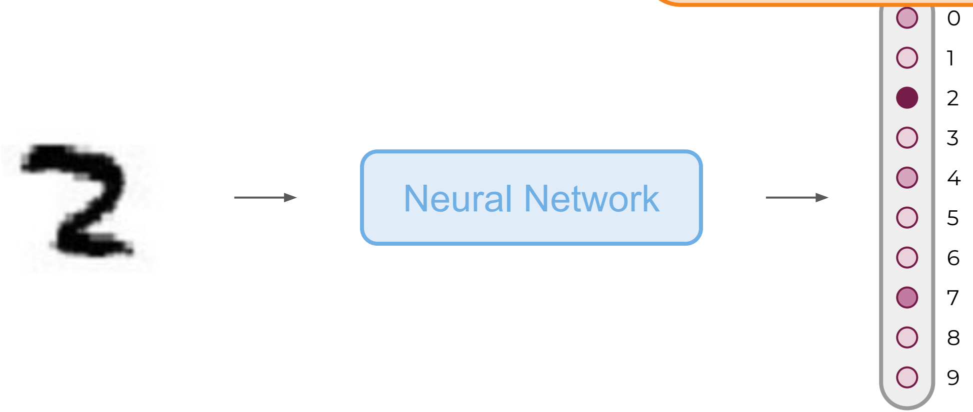
Classification Task (MNIST)



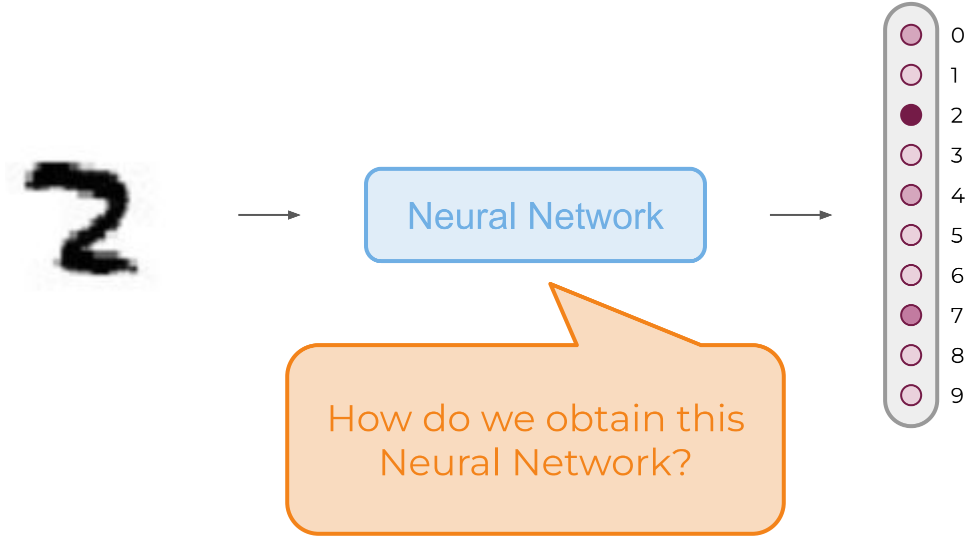
This is a **10-way** classification task, since there are only 10 digits (0, 1, 2, ..., 9)

Classification Task (MNIST)

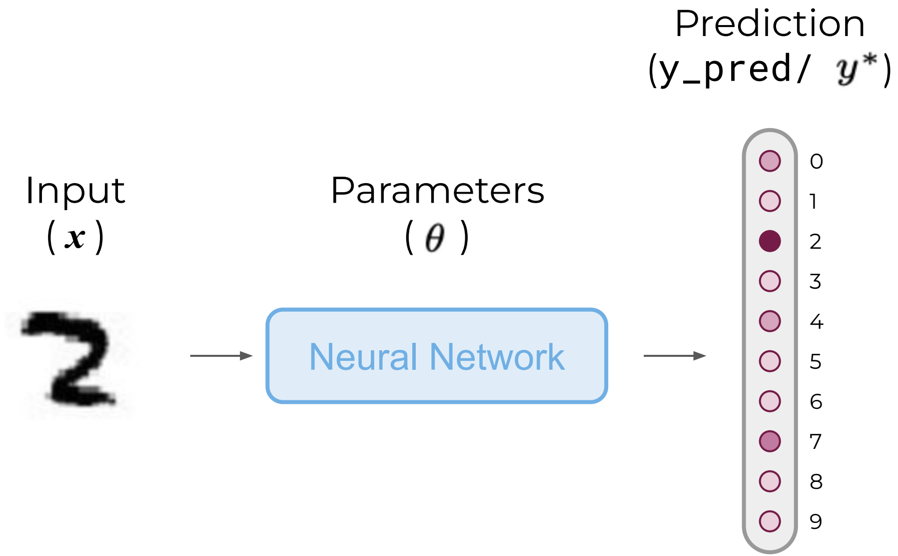
The probability for “**2**” is ranked the highest in the distribution, suggesting that this is a correct prediction



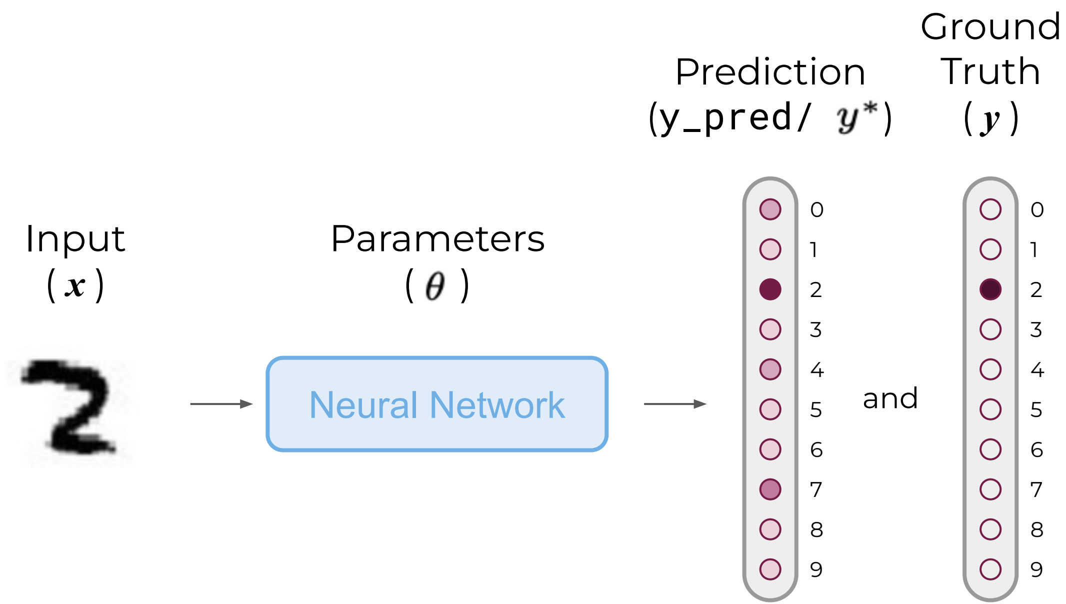
Classification Task (MNIST)



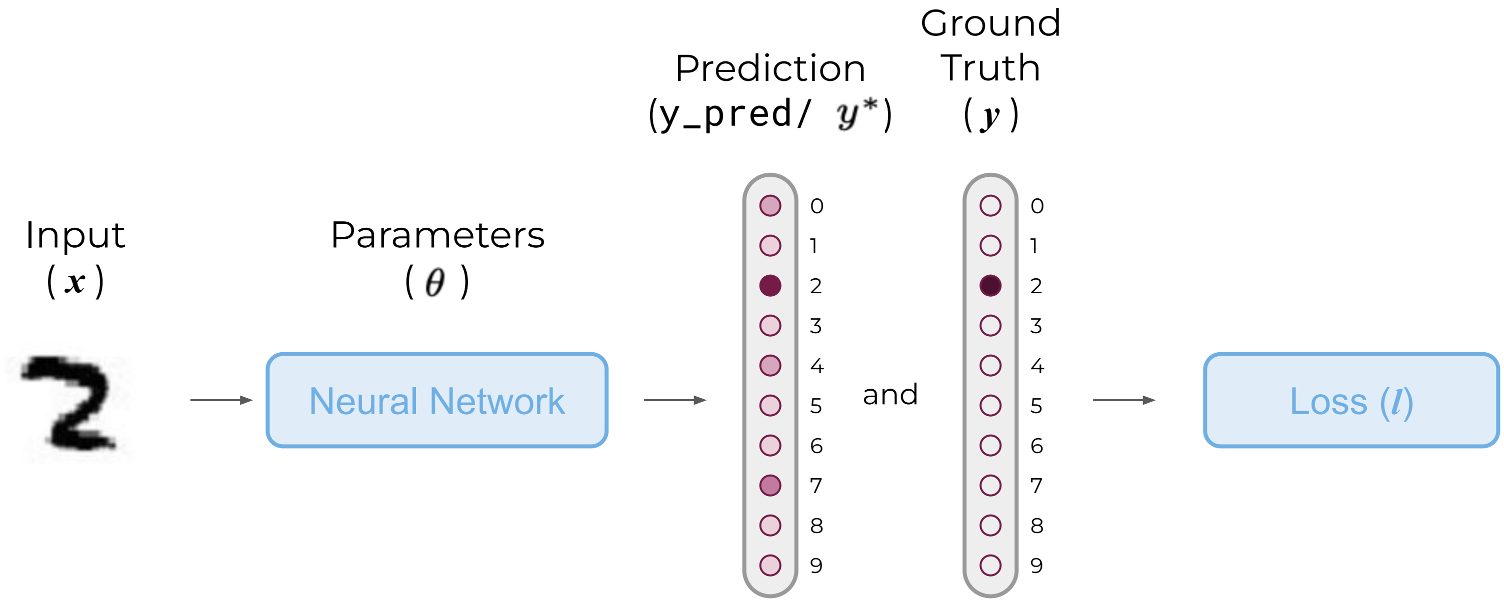
Training Loop



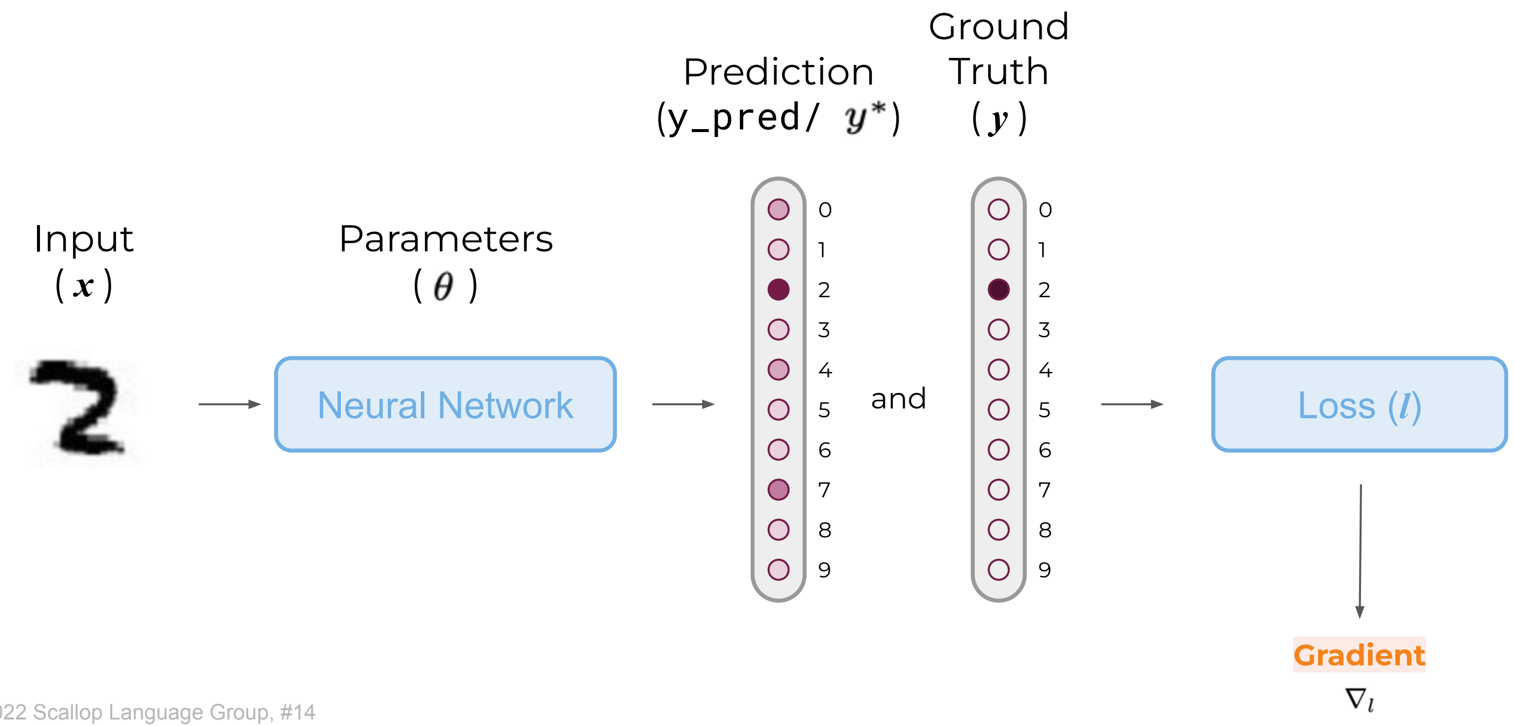
Training Loop



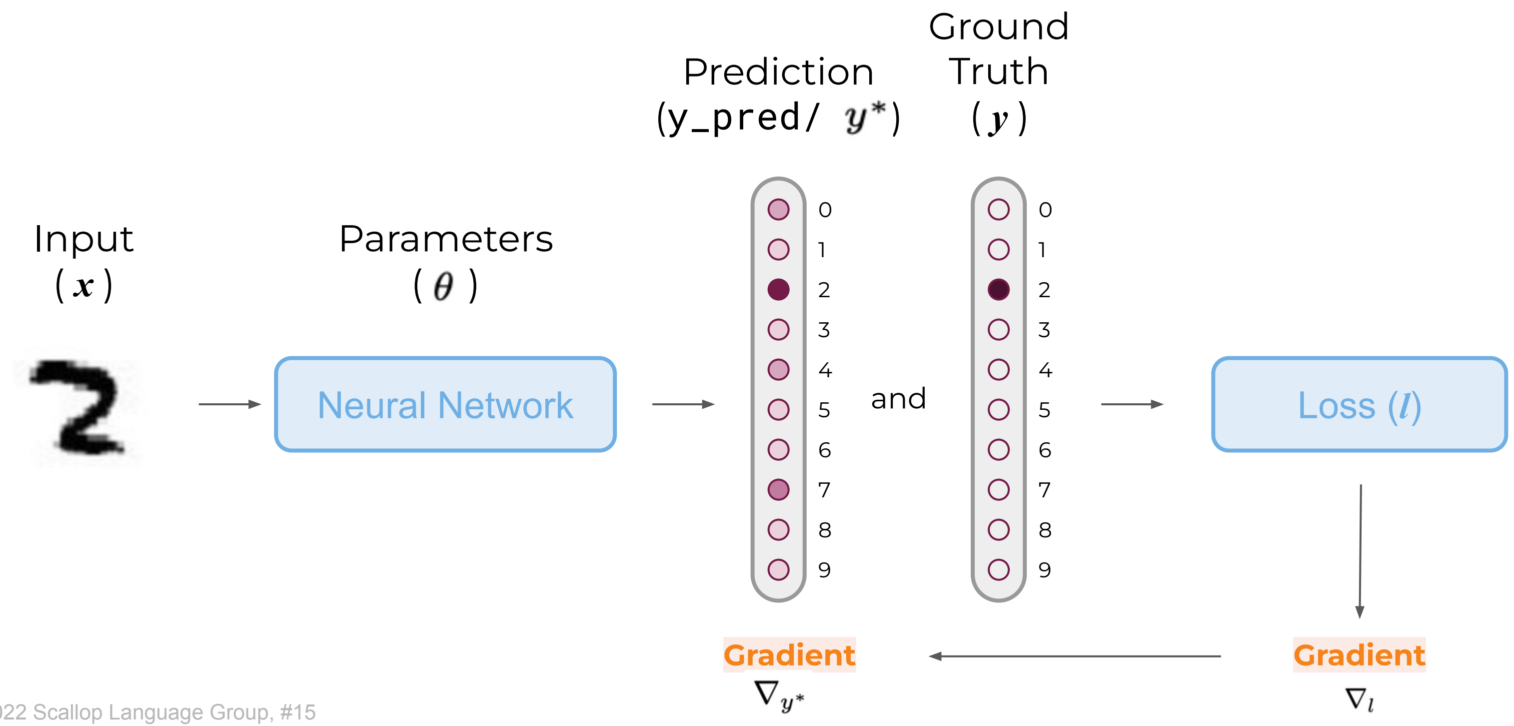
Training Loop



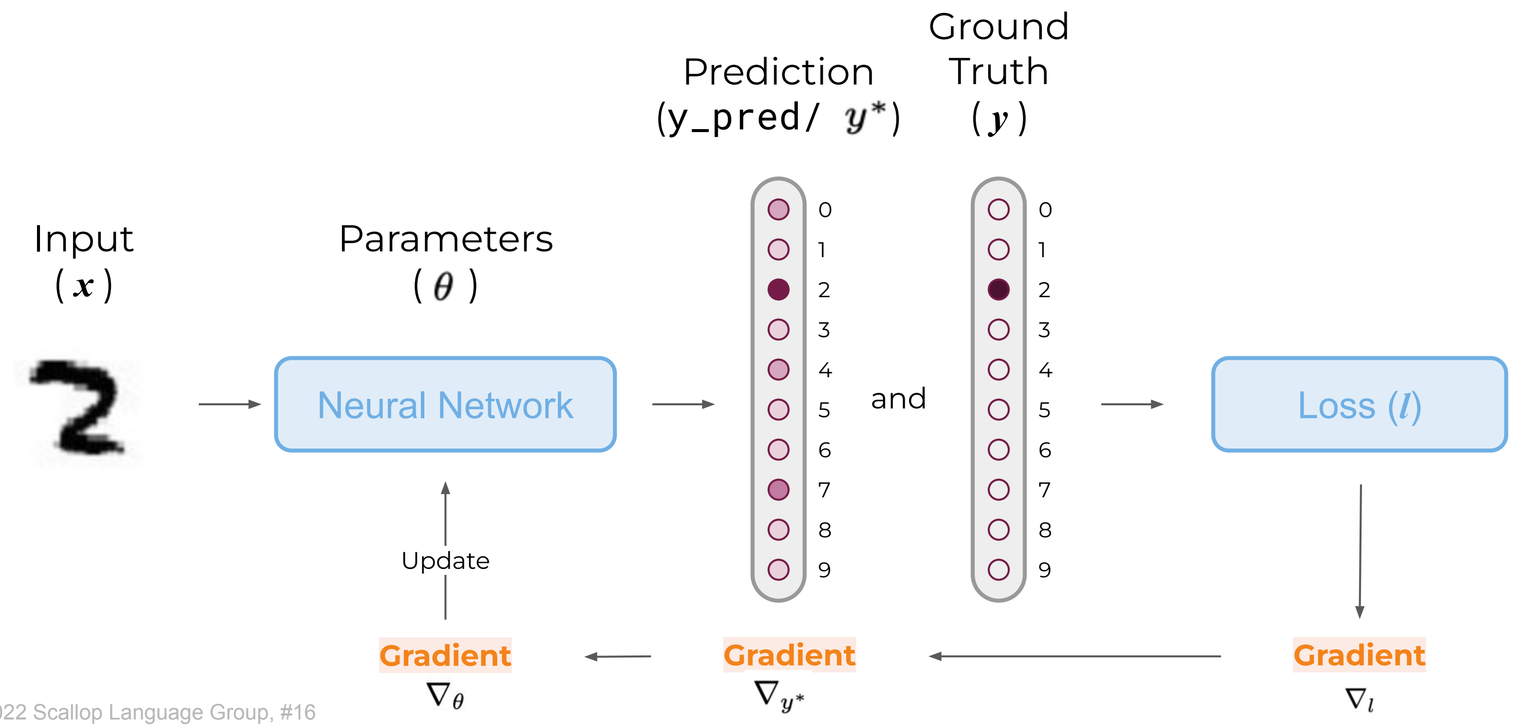
Training Loop



Training Loop



Training Loop



Training Loop for a Classification Task

- Problem Definition:
  - Input: a dataset of (x, y) pairs
  - Output: a neural network (with parameters θ)
- Step-by-step:
  - Pass input (x) into a randomly initialized neural network and get prediction ($y*$)
  - Pass prediction ($y*$) and ground truth (y) into a loss function and get loss (l)
  - Try to minimize the loss by back-propagating gradients into neural network (θ)
  - Repeat the process for the whole dataset for multiple epochs

Training Loop for a Classification Task

- Step-by-step:
  - Pass input (x) into a randomly initialized neural network and get prediction (y^*)
  - Pass prediction (y^*) and ground truth (y) into a loss function and get loss (l)
  - Try to minimize the loss by back-propagating gradients into neural network (θ)
  - Repeat the process for the whole dataset for multiple epochs

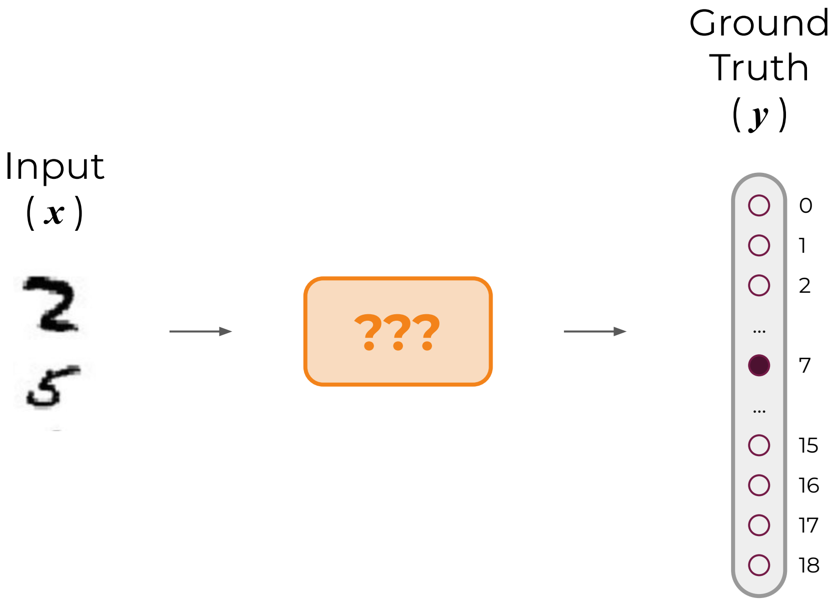
```python
for i in range(num_epochs):
  for (x, y) in dataset:
    self.optimizer.zero_grad()
    y_pred = self.model(x)
    l = self.loss_function(y_pred, y)
    l.backward()
    self.optimizer.step()
```

Why do we need differentiability?

- When doing the “back-propagation” step, we need to know how to update neural network parameters (θ) in order to minimize the loss
- This is done through calculating the gradients of the current layer of parameters w.r.t the previous layer of parameters

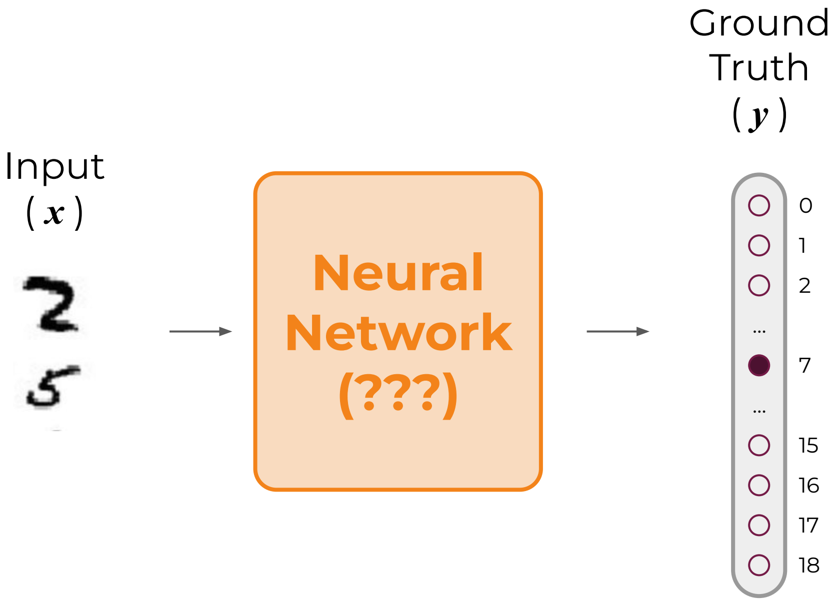
Scallop and Differentiable Reasoning

Training Loop for MNIST Sum-2 Task

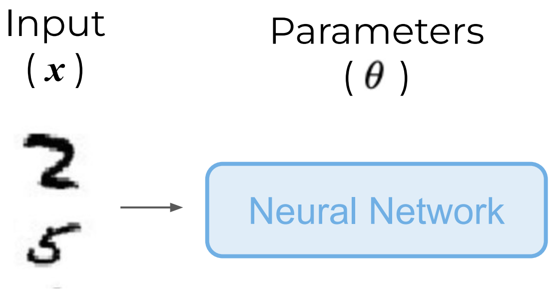


Task Initially Proposed by DeepProbLog (R Manhaeve et. al, 2018)

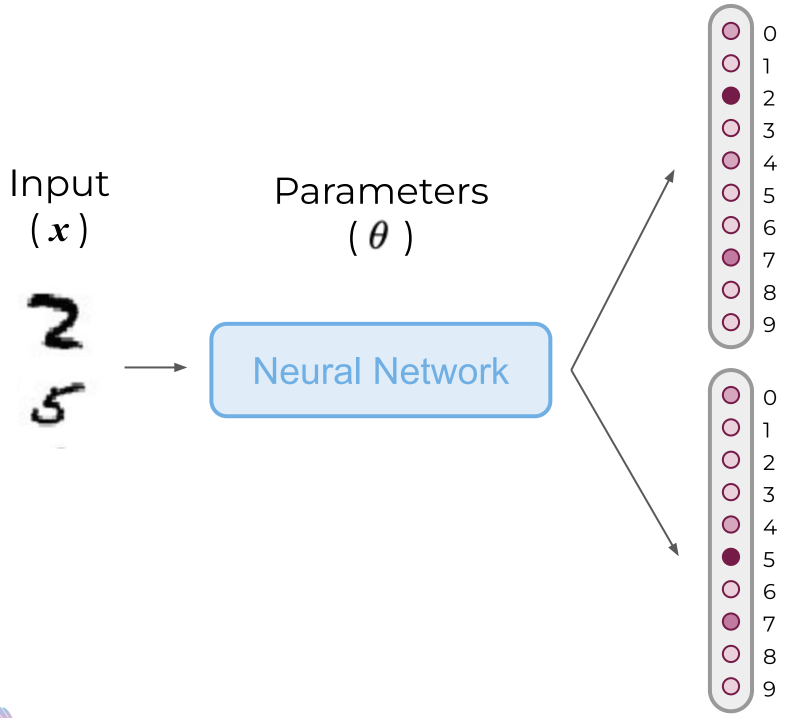
Training Loop for MNIST Sum-2 Task



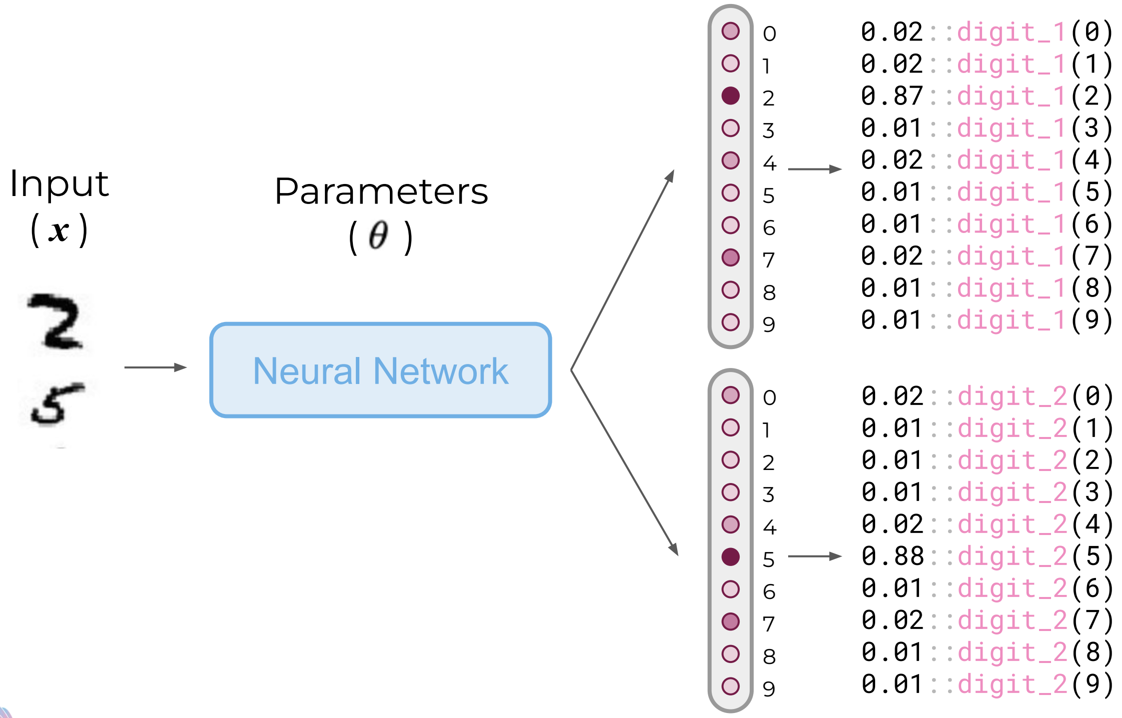
Training Loop for MNIST Sum-2 Task



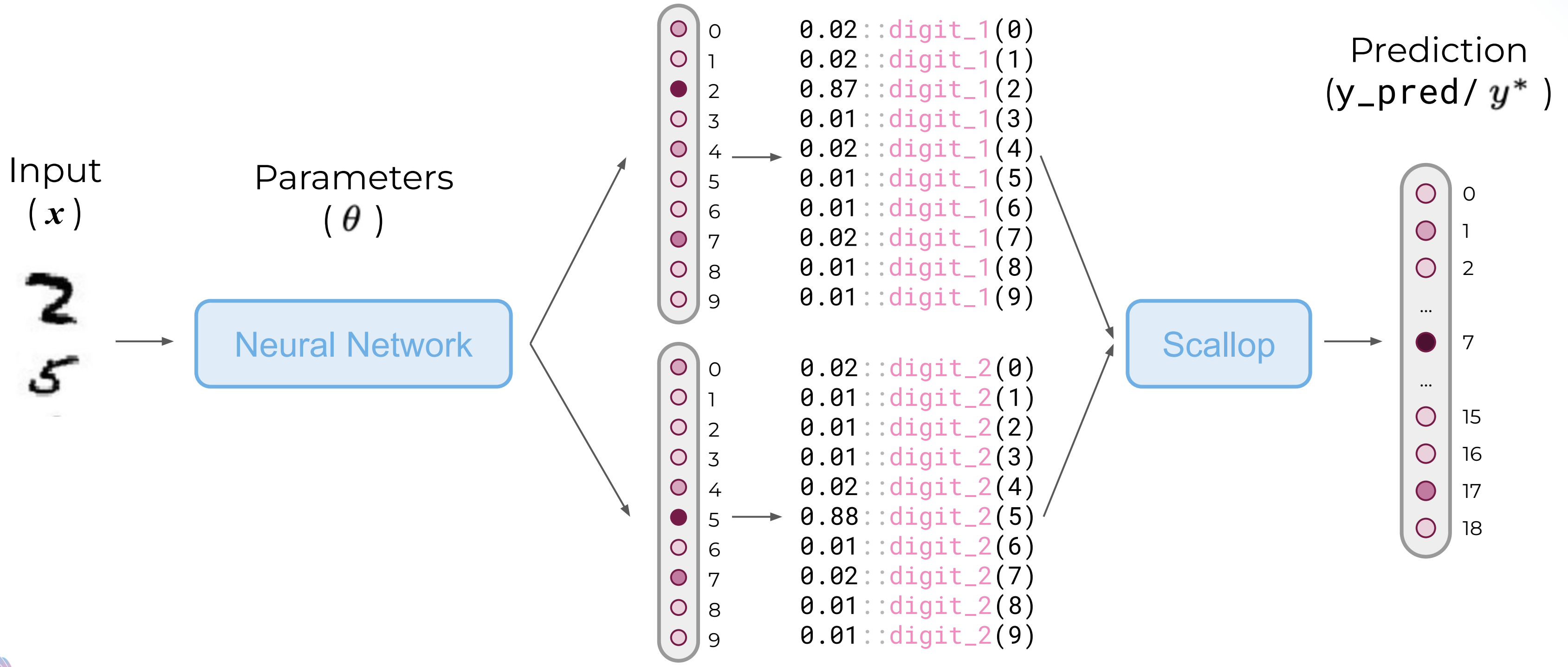
Training Loop for MNIST Sum-2 Task



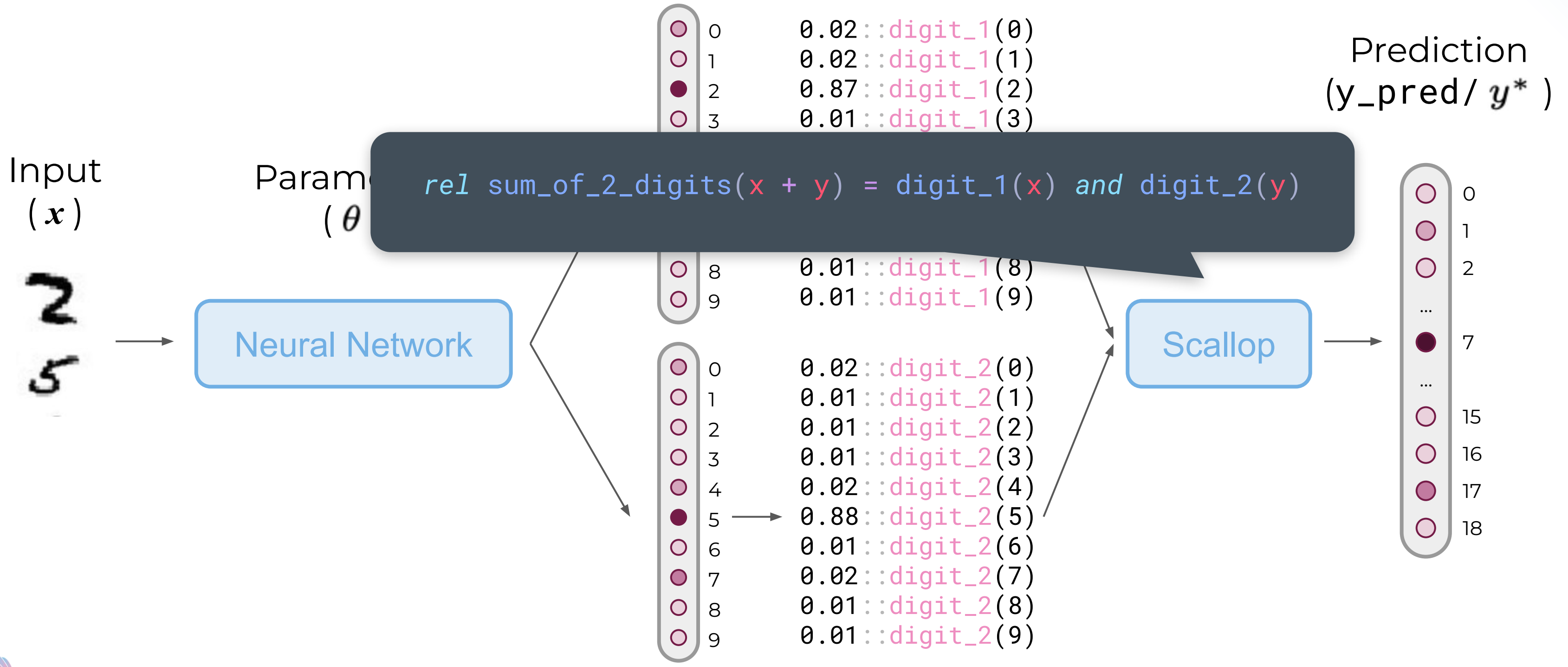
Training Loop for MNIST Sum-2 Task



Training Loop for MNIST Sum-2 Task



Training Loop for MNIST Sum-2 Task



Training Loop involving Perception → Reasoning

- We have some input data (x) that is noisy
- We use neural networks to process the noisy input data into some structured symbolic form (i.e. differentiable & probabilistic facts)
- People write a program in Scallop to reason about these probabilistic facts, and produce some output with probabilities ($y*$)
- Prediction ($y*$) and ground truth (y) will be passed to a loss function to produce the loss (l), which will be back-propagated
- The back-propagation can go through a Scallop's differentiable provenance module – we can obtain the gradients of Scallop output w.r.t the probabilities of the input facts
- ...The rest of the pipeline stays the same as before...

Differentiation & Provenance

- Differentiation:
  - You want to know how a variation of the input value would affect the output
  - i.e. We can obtain gradient of the output w.r.t the input
- Provenance:
  - You assign a tag to each input tuple, and tag for each derived fact encodes “how such a fact is derived”
  - Since you know how facts are derived, you know how to change the input tags in order to alter the output tags towards where we want (i.e. minimizing loss)
  - i.e. We can obtain gradient of the output tags w.r.t the input tags

Supported Differentiable Provenance

- If the following provenances are employed, the output tags will be associated with gradients w.r.t input tags
- The `diff`erentiable counterpart of the probabilistic provenances
    - `diffminmaxprob`
    - `diffaddmultprob`
    - `diffnandmultprob`
    - `difftopkproofs`
    - `diffsamplekproofs`
    - `difftopbottomkclauses`

Scallop and PyTorch

scallopy

- scallopy is the Python binding for Scallop
- It allows you to use Scallop in a programmable environment
    - Dynamically add relations, rules, and facts in Python
    - Branch-off execution
- It allows you to integrate Scallop with PyTorch
    - It accepts torch.tensor and automatically convert them to facts and tags
    - Automatically back-propagate gradients
    - Allow creating a “forward” function following PyTorch standards
    - Can save/load the model

MNIST Sum 2: Dataset

```
class MNISTSum2Dataset(torch.utils.data.Dataset):
  …
  def __getitem__(self, idx):
    (a_img, a_digit) = self.mnist_dataset[self.index_map[idx * 2]]
    (b_img, b_digit) = self.mnist_dataset[self.index_map[idx * 2 + 1]]
    return (a_img, b_img, a_digit + b_digit)
```

We can setup the Dataset inheriting a **PyTorch** Dataset class

7

Each Datapoint contains two images of number and their sum

MNIST Sum 2: Single Digit Recognition

```
class MNISTNet(nn.Module):
  …
  def forward(self, x):
    x = F.max_pool2d(self.conv1(x), 2)
    x = F.max_pool2d(self.conv2(x), 2)
    x = x.view(-1, 1024)
    x = F.relu(self.fc1(x))
    x = F.dropout(x, p = 0.5, training=self.training)
    x = self.fc2(x)
    return F.softmax(x, dim=1)
```

MNISTNet is a single digit recognition neural network

MNIST Sum 2: Sum Two Digits

```
class MNISTSum2Net(nn.Module):
  def __init__(self):
    super(MNISTSum2Net, self).__init__()
    self.mnist_net = MNISTNet()
    self.scl_ctx = scallopy.ScallopContext(provenance="difftopkproofs", k=3)
    …
```

ScallopyContext is the handle for us to execute Scallop in Python.

MNIST Sum 2: Sum Two Digits

```python
class MNISTSum2Net(nn.Module):
  def __init__(self, provenance, k):
    super(MNISTSum2Net, self).__init__()
    self.mnist_net = MNISTNet()
    self.scl_ctx = scallopy.ScallopContext(provenance="difftopkproof
    self.scl_ctx.add_relation("digit_1", int, input_mapping=list(range(10)))
    self.scl_ctx.add_relation("digit_2", int, input_mapping=list(range(10)))
    self.scl_ctx.add_rule("sum_2(a + b) :- digit_1(a), digit_2(b)")
    …
```

We can add **relations** and **rules** to the Scallopy context

MNIST Sum 2: Sum Two Digits

```
class MNISTSum2Net(nn.Module):
  def __init__(self, provenance, k):
    super(MNISTSum2Net, self).__init__()
    self.mnist_net = MNISTNet()
    self.scl_ctx = scallopy.ScallopContext(provenance="difftopkproofs", k=3)
    self.scl_ctx.add_relation("digit_1", int, input_mapping=list(range(10)))
    self.scl_ctx.add_relation("digit_2", int, input_mapping=list(range(10)))
    self.scl_ctx.add_rule("sum_2(a + b) :- digit_1(a), digit_2(b)")
    self.sum_2 = self.scl_ctx.forward_function("sum_2", output_mapping=[(i,) for i in
range(19)])
```

Defining a scallopy **forward function** with the mapping of the query output into tensor

MNIST Sum 2: Sum Two Digits

```
class MNISTSum2Net(nn.Module):
  …
  def forward(self, x: Tuple[torch.Tensor, torch.Tensor]):
    (a_imgs, b_imgs) = x

    a_distrs = self.mnist_net(a_imgs) # Tensor 64 x 10
    b_distrs = self.mnist_net(b_imgs) # Tensor 64 x 10

    return self.sum_2(digit_1=a_distrs, digit_2=b_distrs) # Tensor 64 x 19
```

Applying the **forward function** provides the query result tagged with differentiable probabilities

MNIST Sum 2: Sum Two Digits

```
class Trainer():
  …
  def train_epoch(self, epoch):
    self.network.train()
    iter = tqdm(self.train_loader, total=len(self.train_loader))
    for (data, target) in iter:
      self.optimizer.zero_grad()
      output = self.network(data)
      loss = self.loss(output, target)
      loss.backward()
      self.optimizer.step()
      iter.set_description(f"[Train Epoch {epoch}] Loss: {loss.item():.4f}")
```

Voilà! You can just write the training pipeline as usual.