dPASP: Possíveis Aplicações

Soma de 2 dígitos

- Entrada: 2 imagens de números escritos à mão
- Saída: soma desses números
- Dataset utilizado para treinar o modelo: MNIST
- Precisão: ~93%

```
#python
import torch
import torchvision

# Digit classification network definition.
class Net(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.encoder = torch.nn.Sequential(
      torch.nn.Conv2d(1, 6, 5),
      torch.nn.MaxPool2d(2, 2),
      torch.nn.ReLU(True),
      torch.nn.Conv2d(6, 16, 5),
      torch.nn.MaxPool2d(2, 2),
      torch.nn.ReLU(True)
    )
    self.classifier = torch.nn.Sequential(
      torch.nn.Linear(16 * 4 * 4, 120),
      torch.nn.ReLU(),
      torch.nn.Linear(120, 84),
      torch.nn.ReLU(),
      torch.nn.Linear(84, 10),
      torch.nn.Softmax(1)
    )

  def forward(self, x):
    x = self.encoder(x)
    x = x.view(-1, 16 * 4 * 4)
    x = self.classifier(x)
    return x

# Return an instance of Net.
def digit_net(): return Net()

# Retrieve the MNIST data.
def mnist_data():
  train = torchvision.datasets.MNIST(root = "/tmp", train = True, download = True)
  test  = torchvision.datasets.MNIST(root = "/tmp", train = False, download = True)
  return train.data.float().reshape(len(train), 1, 28, 28)/255., train.targets, \
         test.data.float().reshape(len(test), 1, 28, 28)/255., test.targets
```

Soma de 2 dígitos

- Entrada: 2 imagens de números escritos à mão
- Saída: soma desses números
- Dataset utilizado para treinar o modelo: MNIST
- Precisão: ~93%

```
# Normalization function to center pixel values around mu with standard deviation sigma.
def normalize(X_R, Y_R, X_T, Y_T, mu, sigma):
  return (X_R-mu)/sigma, Y_R, (X_T-mu)/sigma, Y_T

train_X, train_Y, test_X, test_Y = normalize(*mnist_data(), 0.1307, 0.3081)
# Whether to pick the first or second half of the dataset.
def pick_slice(data, which):
  h = len(data)//2
  return slice(h, len(data)) if which else slice(0, h)
# MNIST images for the train set.
def mnist_images_train(which): return train_X[pick_slice(train_X, which)]
# MNIST images for the test set.
def mnist_images_test(which): return test_X[pick_slice(test_X, which)]
# Observed atoms for training.
def mnist_labels_train():
  # We join the two halves (top and bottom) of MNIST and join them together to get
  # two digits side by side. The labels are atoms encoding the sum of the two digits.
  labels = torch.concatenate((train_Y[:(h := len(train_Y)//2)].reshape(-1, 1),
                              train_Y[h:].reshape(-1, 1)), axis=1)
  return [[f"sum({x.item() + y.item()})"] for x, y in labels]
#end.

% Data of the first digit.
input(0) ~ test(@mnist_images_test(0)), train(@mnist_images_train(0)).
% Data of the second digit.
input(1) ~ test(@mnist_images_test(1)), train(@mnist_images_train(1)).

% Neural annotated disjunction over each digit from 0 to 9; use Adam as optimizer
% and a learning rate of 0.001.
?::digit(X, {0..9}) as @digit_net with optim = "Adam", lr = 0.001 :- input(X).
% The sum.
sum(Z) :- digit(0, X), digit(1, Y), Z = X+Y.

% Learn the parameters of the program from the "sum(X)" atoms.
#learn @mnist_labels_train, lr = 1., niters = 5, alg = "lagrange", batch = 500.
#semantics maxent.
% Ask for the probability of all groundings of sum(X).
#query sum(X).
```

Soma de 4 dígitos

- Entrada: 2 imagens de números escritos à mão
- Saída: soma desses números
- Dataset utilizado para treinar o modelo: MNIST
- Precisão: ~95%

```python
#python
import torch
import torchvision
import matplotlib.pyplot as plt

# Digit classification network definition.
class Net(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.encoder = torch.nn.Sequential(
      torch.nn.Conv2d(1, 32, 3, padding=1),
      torch.nn.MaxPool2d(2),
      torch.nn.ReLU(True),
      torch.nn.Conv2d(32, 64, 3, padding=1),
      torch.nn.MaxPool2d(2),
      torch.nn.ReLU(True)
    )
    self.classifier = torch.nn.Sequential(
      torch.nn.Linear(64 * 7 * 7, 1024),
      torch.nn.ReLU(),
      torch.nn.Linear(1024, 10),
      torch.nn.Softmax(1)
    )

  def forward(self, x):
    x = self.encoder(x)
    x = x.view(-1, 64 * 7 * 7)
    x = self.classifier(x)
    return x

# Return an instance of Net.
def digit_net(): return Net()

# Retrieve the MNIST data.
def mnist_data():
  train = torchvision.datasets.MNIST(root = "/tmp", train = True, download = True)
  test  = torchvision.datasets.MNIST(root = "/tmp", train = False, download = True)
  return train.data.float().reshape(len(train), 1, 28, 28)/255., train.targets, \
         test.data.float().reshape(len(test), 1, 28, 28)/255., test.targets

```

Soma de 4 dígitos

- Entrada: 2 imagens de números escritos à mão
- Saída: soma desses números
- Dataset utilizado para treinar o modelo: MNIST
- Precisão: ~95%

```
# Normalization function to center pixel values around mu with standard deviation sigma.
def normalize(X_R, Y_R, X_T, Y_T, mu, sigma): return (X_R-mu)/sigma, Y_R, (X_T-mu)/sigma, Y_T

train_X, train_Y, test_X, test_Y = normalize(*mnist_data(), 0.1307, 0.3081)

def pick_slice(data, which):
  h = len(data)//4
  if which == 0: return slice(0, h)
  elif which == 1: return slice(h, 2*h)
  elif which == 2: return slice(2*h, 3*h)
  return slice(3*h, 4*h)

# MNIST images for the train set.
def mnist_images_train(which): return train_X[pick_slice(train_X, which)]

# MNIST images for the test set.
def mnist_images_test(which): return test_X[pick_slice(test_X, which)]

# Observed atoms for training.
def mnist_labels_train():
  h = len(train_Y)//4
  labels = torch.concatenate((train_Y[:h].reshape(-1, 1),
                              train_Y[h:2*h].reshape(-1, 1),
                              train_Y[2*h:3*h].reshape(-1, 1),
                              train_Y[3*h:4*h].reshape(-1, 1)), axis=1)
  T = []
  for i in range(h): T.append(labels[i][0] + labels[i][1] + labels[i][2] + labels[i][3])
  T = torch.tensor(T)

  D = [[f"sum({a.item()+b.item() + c.item() + d.item()})"] for a, b, c, d in labels]
  return D
#end.

% Data of the first number.
input(0) ~ test(@mnist_images_test(0)), train(@mnist_images_train(0)).
input(1) ~ test(@mnist_images_test(1)), train(@mnist_images_train(1)).
% Data of the second number.
input(2) ~ test(@mnist_images_test(2)), train(@mnist_images_train(2)).
input(3) ~ test(@mnist_images_test(3)), train(@mnist_images_train(3)).
```

Soma de 4 dígitos

- Entrada: 2 imagens de números escritos à mão
- Saída: soma desses números
- Dataset utilizado para treinar o modelo: MNIST
- Precisão: ~95%

```
81  % Neural annotated disjunction over each number from 0 to 9; use Adam as optimizer
82  % and a learning rate of 0.001.
83  ?::digit(X, {0..9}) as @digit_net with optim = "Adam", lr = 0.001 :- input(X).
84  % The sum.
85  sum(Z) :- digit(0, A), digit(1, B), digit(2, C), digit(3, D), Z = A+B+C+D.
86
87  % Learn the parameters of the program from the "sum(X)" atoms.
88  #learn @mnist_labels_train, lr = 0.001, niters = 10, alg = "lagrange", batch = 64.
89  #semantics maxent.
90  #query sum(X).
91  % Ask for the probability of all groundings of sum(X).
```

Soma de números de 2 dígitos

- Entrada: 2 listas de 2 imagens de números escritos à mão
- Saída: soma desses números
- Dataset utilizado para treinar o modelo: MNIST
- Precisão: ~49%

```python
#python
import torch
import torchvision
import matplotlib.pyplot as plt

# Digit classification network definition.
class Net(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.encoder = torch.nn.Sequential(
      torch.nn.Conv2d(1, 32, 3, padding=1),
      torch.nn.MaxPool2d(2),
      torch.nn.ReLU(True),
      torch.nn.Conv2d(32, 64, 3, padding=1),
      torch.nn.MaxPool2d(2),
      torch.nn.ReLU(True)
    )
    self.classifier = torch.nn.Sequential(
      torch.nn.Linear(64 * 7 * 7, 1024),
      torch.nn.ReLU(),
      torch.nn.Linear(1024, 10),
      torch.nn.Softmax(1)
    )

  def forward(self, x):
    x = self.encoder(x)
    x = x.view(-1, 64 * 7 * 7)
    x = self.classifier(x)
    return x

# Return an instance of Net.
def digit_net(): return Net()

# Retrieve the MNIST data.
def mnist_data():
  train = torchvision.datasets.MNIST(root = "/tmp", train = True, download = True)
  test  = torchvision.datasets.MNIST(root = "/tmp", train = False, download = True)
  return train.data.float().reshape(len(train), 1, 28, 28)/255., train.targets, \
         test.data.float().reshape(len(test), 1, 28, 28)/255., test.targets

```

Soma de números de 2 dígitos

- Entrada: 2 listas de 2 imagens de números escritos à mão
- Saída: soma desses números
- Dataset utilizado para treinar o modelo: MNIST
- Precisão: ~49%

```
41  # Normalization function to center pixel values around mu with standard deviation sigma.
42  def normalize(X_R, Y_R, X_T, Y_T, mu, sigma):
43    return (X_R-mu)/sigma, Y_R, (X_T-mu)/sigma, Y_T
44
45  train_X, train_Y, test_X, test_Y = normalize(*mnist_data(), 0.1307, 0.3081)
46
47  # which slice of the dataset.
48  def pick_slice(data, which):
49    h = len(data)//4
50    if which == 0: return slice(0, h)
51    elif which == 1: return slice(h, 2*h)
52    elif which == 2: return slice(2*h, 3*h)
53    return slice(3*h, 4*h)
54
55  # MNIST images for the train set.
56  def mnist_images_train(which):
57      return train_X[pick_slice(train_X, which)]
58
59  # MNIST images for the test set.
60  def mnist_images_test(which):
61      return test_X[pick_slice(test_X, which)]
62
63
64  def printMnist(labels, index):…
85
86  # Observed atoms for training.
87  def mnist_labels_train():
88    h = len(train_Y)//4
89    labels = torch.concatenate((train_Y[:h].reshape(-1, 1),
90                                train_Y[h:2*h].reshape(-1, 1),
91                                train_Y[2*h:3*h].reshape(-1, 1),
92                                train_Y[3*h:4*h].reshape(-1, 1)), axis=1)
93
94    D = [[f"sum({10*a.item() + b.item() + 10*c.item() + d.item()})"] for a, b, c, d in labels]
95    return D
96  #end.
```

Soma de números de 2 dígitos

- Entrada: 2 listas de 2 imagens de números escritos à mão
- Saída: soma desses números
- Dataset utilizado para treinar o modelo: MNIST
- Precisão: ~49%

```
% Data of the first number.
input(0) ~ test(@mnist_images_test(0)), train(@mnist_images_train(0)).
input(1) ~ test(@mnist_images_test(1)), train(@mnist_images_train(1)).
% Data of the second number.
input(2) ~ test(@mnist_images_test(2)), train(@mnist_images_train(2)).
input(3) ~ test(@mnist_images_test(3)), train(@mnist_images_train(3)).

% Neural annotated disjunction over each number from 0 to 9; use Adam as optimizer
% and a learning rate of 0.001.
?::digit(X, {0..9}) as @digit_net with optim = "Adam", lr = 0.001 :- input(X).
% The sum.
sum(Z) :- digit(0, A), digit(1, B), digit(2, C), digit(3, D), Z = 10*A+B+10*C+D.

% Learn the parameters of the program from the "sum(X)" atoms.
#learn @mnist_labels_train, lr = 0.001, niters = 10, alg = "lagrange", batch = 64.
#semantics maxent.
#query sum(X).
% Ask for the probability of all groundings of sum(X).
```

Tradução e solução de fórmulas escritas à mão

- Entrada: sequência de imagens que representam um dígito ou um símbolo atirmético e um inteiro representando o tamanho da fórmula;
- O tamanho da fórmula varia entre 1 e 7 símbolos;
- Saída: um número racional.

Bubble Sort

- Abordagem similar à ∂4 (differentiable forth);
- O programa não especifica quando ocorre a troca dos elementos;
- DeepProbLog atingiu 100% de precisão em listas nas quais ∂4 não chegou a uma resposta.

Accuracy on the sorting (**T6**) problem (results for ∂4 reported by Bošnjak et al. [8]).

	Test length	Training length				
		2	3	4	5	6
∂4 [8]	8	100.0	100.0	49.22	–	–
	64	100.0	100.0	20.65	–	–
DeepProbLog	8	100.0	100.0	100.0	100.0	100.0
	64	100.0	100.0	100.0	100.0	100.0

neural predicate

```
hole(X,Y,X,Y):-
   swap(X,Y,0).

hole(X,Y,Y,X):-
   swap(X,Y,1).
```

bubble sort

(a)

```
bubble([X],[],X).
bubble([H1,H2|T],[X1|T1],X):-
   hole(H1,H2,X1,X2),
   bubble([X2|T],T1,X).

bubblesort([],L,L).

bubblesort(L,L3,Sorted) :-
   bubble(L,L2,X),
   bubblesort(L2,[X|L3],Sorted).

sort(L,L2) :- bubblesort(L,[],L2).
```

Solução de WAPs

- Dados a descrição de um problema matemático simples e números extraídos dessa descrição, o programa não especifica quando usar as operações necessárias para chegar na solução (permutação, troca ou outra operação);
- DeepProbLog atingiu precisão de 96%.

Classificação de moedas e comparação

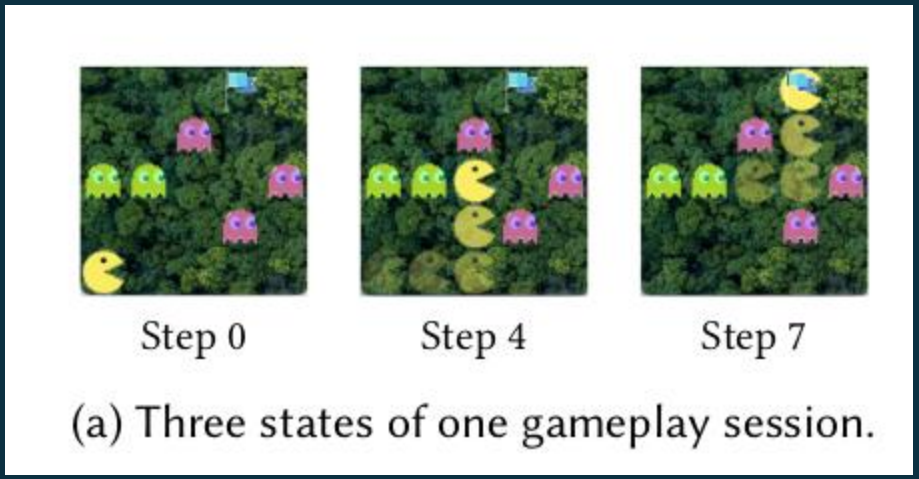
- Dada uma imagem de duas moedas, deve-se dizer se ambas são cara ou coroa ou se são diferentes;
- No experimento do deepProbLog, após 10 exemplos o modelo atingiu uma precisão acima de 90%.

Poker simplificado

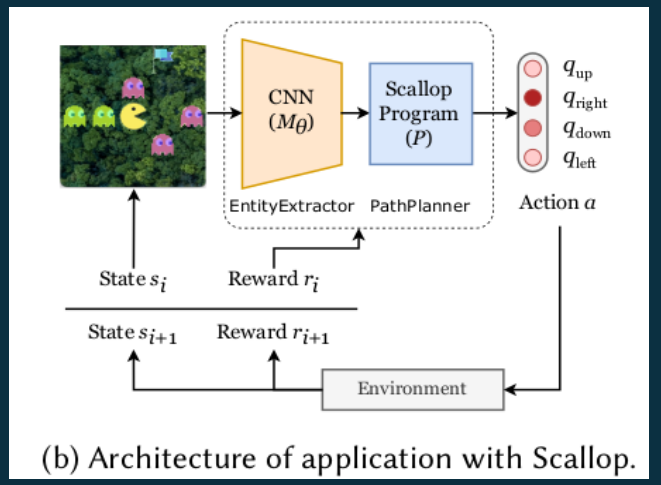
- Dois jogadores recebem 2 cartas e devem fazer uma combinação de poker com essas cartas e uma carta "compartilhada" na mesa;
- O programa deve determinar a probabilidade do jogador 1 vencer, no entanto a carta compartilhada não é conhecida;
- A entrada é um conjunto de quatro cartas (entre J, Q, K e A);
- Todo rótulo contém a probabilidade do jogo terminar em vitória, derrota ou empate;

PacMan

- Dado um labirinto, o programa deve traçar uma rota para o pacman chegar na bandeira sem encontrar inimigos no caminho;
- Avaliação de taxa de sucesso baseia-se na quantidade de vezes que o pacman chegou na bandeira dentro de um certo tempo;
- Treinamento é realizado com conjuntos de estados em um labirinto que levam à vitória.
- Q-Learning leva 50 mil episódios para atingir 84,9% de sucesso;
- Abordagem da Scallop necessita de apenas 50 episódios para obter 99,4% de taxa de sucesso.



(a) Three states of one gameplay session.



(b) Architecture of application with Scallop.

Referências

- Neural probabilistic logic programming in DeepProbLog;
- Scallop: A Language for Neurosymbolic Programming;
- Programming with a Differentiable Forth Interpreter.