Differentiable Programming

Dr. Svetlin Penkov

Logistic Regression

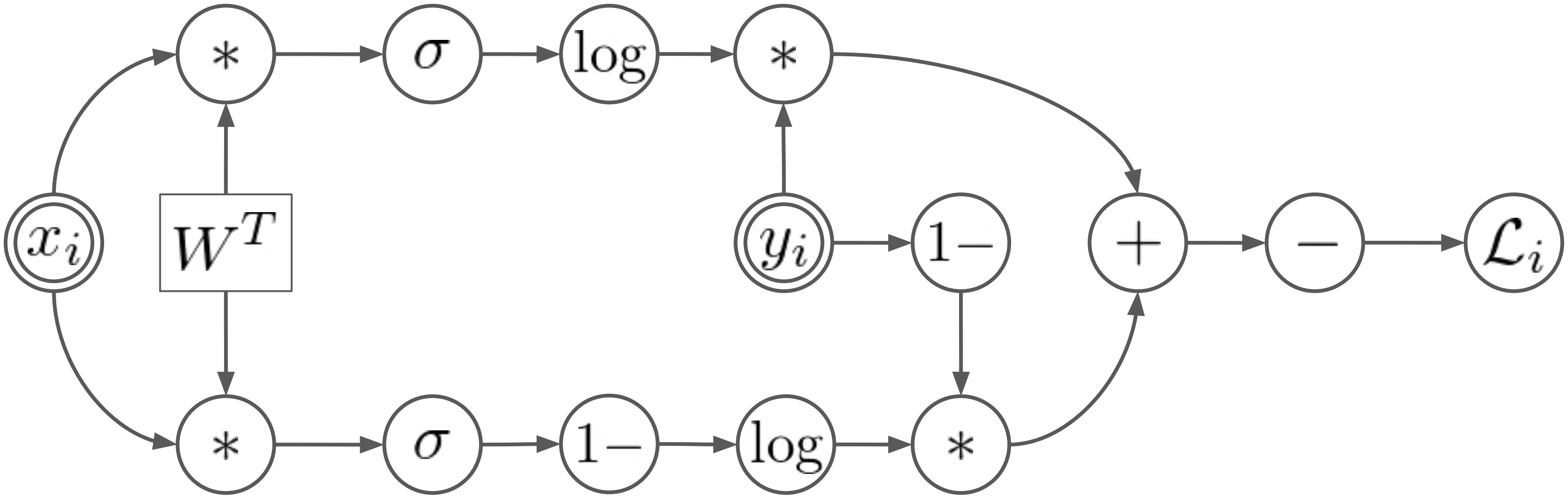
$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log \sigma(x_i W^T) + (1 - y_i) \log (1 - \sigma(x_i W^T))$$

where

$$p(y_i = 1 | x_i, W) = \sigma(x_i W^T)$$

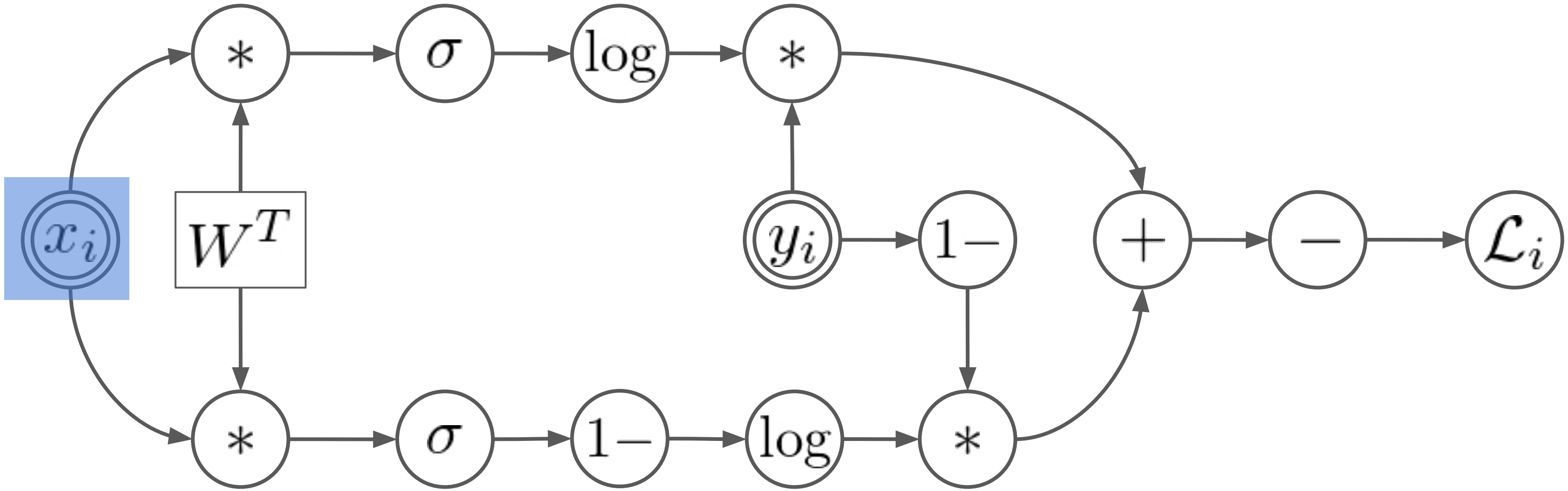
Computational Graph

$$\mathcal{L}_i = -\left[y_i \log \sigma(x_i W^T) + (1 - y_i) \log\left(1 - \sigma(x_i W^T)\right)\right]$$



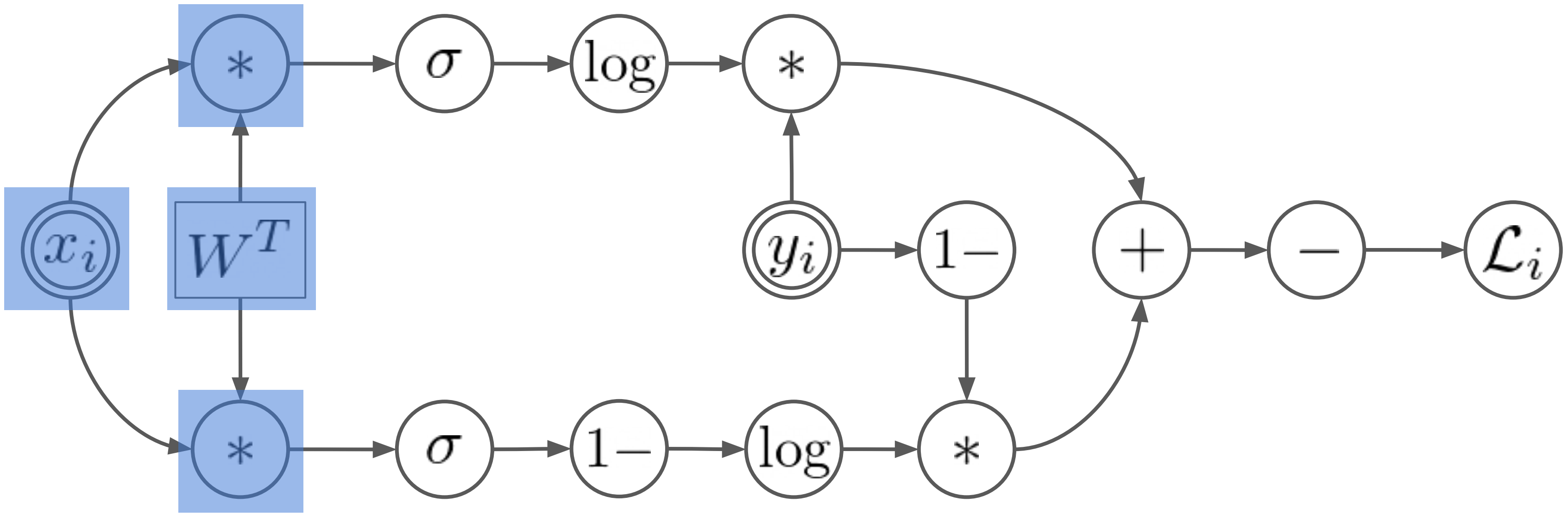
Computational Graph

$$\mathcal{L}_i = -\left[y_i \log \sigma(x_i W^T) + (1 - y_i) \log\left(1 - \sigma(x_i W^T)\right)\right]$$



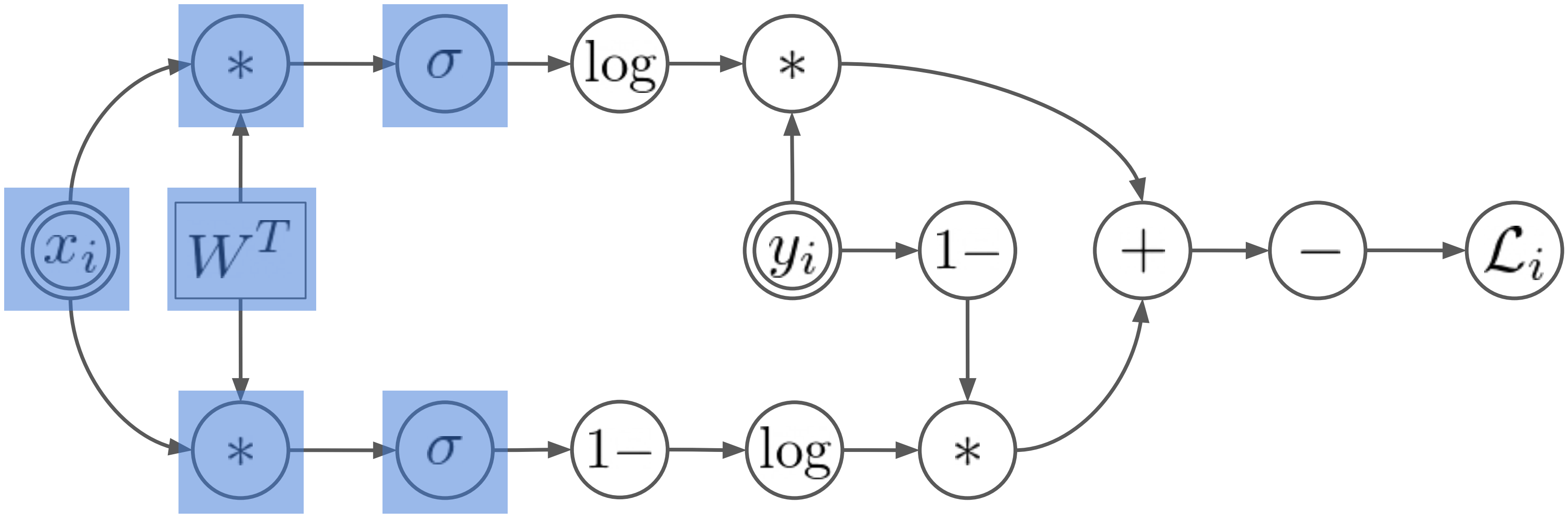
Computational Graph

$$\mathcal{L}_i = -\left[y_i \log \sigma(x_i W^T) + (1 - y_i) \log (1 - \sigma(x_i W^T))\right]$$



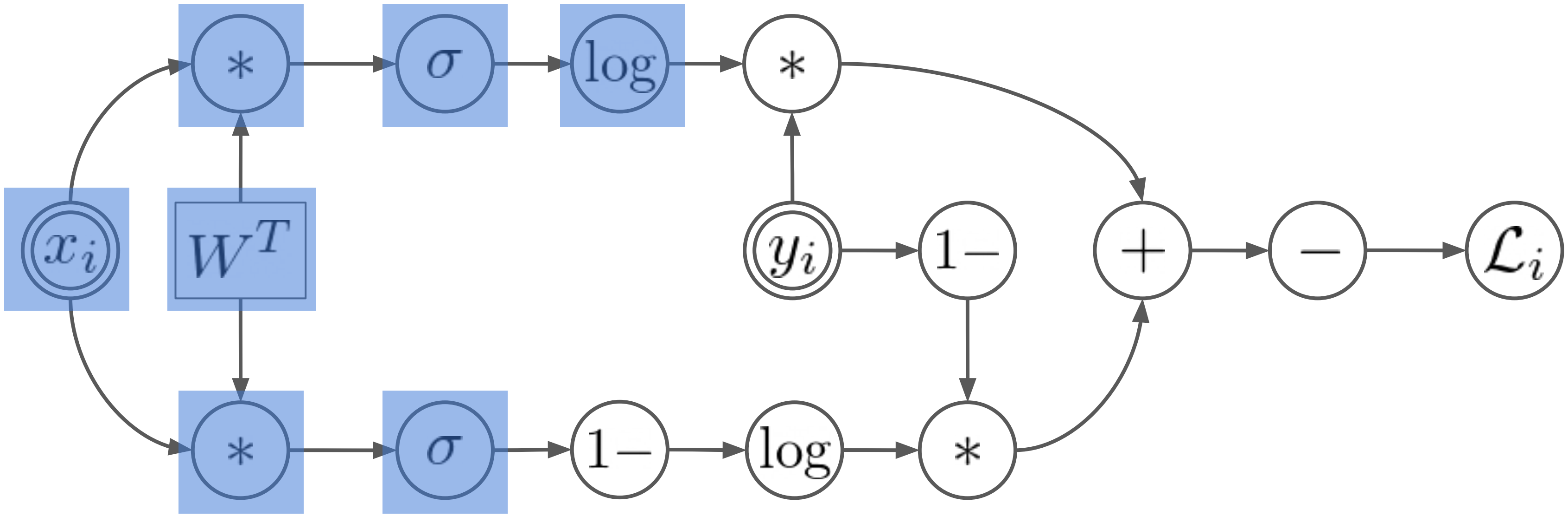
Computational Graph

$$\mathcal{L}_i = -\left[y_i \log \sigma(x_i W^T) + (1 - y_i) \log\left(1 - \sigma(x_i W^T)\right)\right]$$



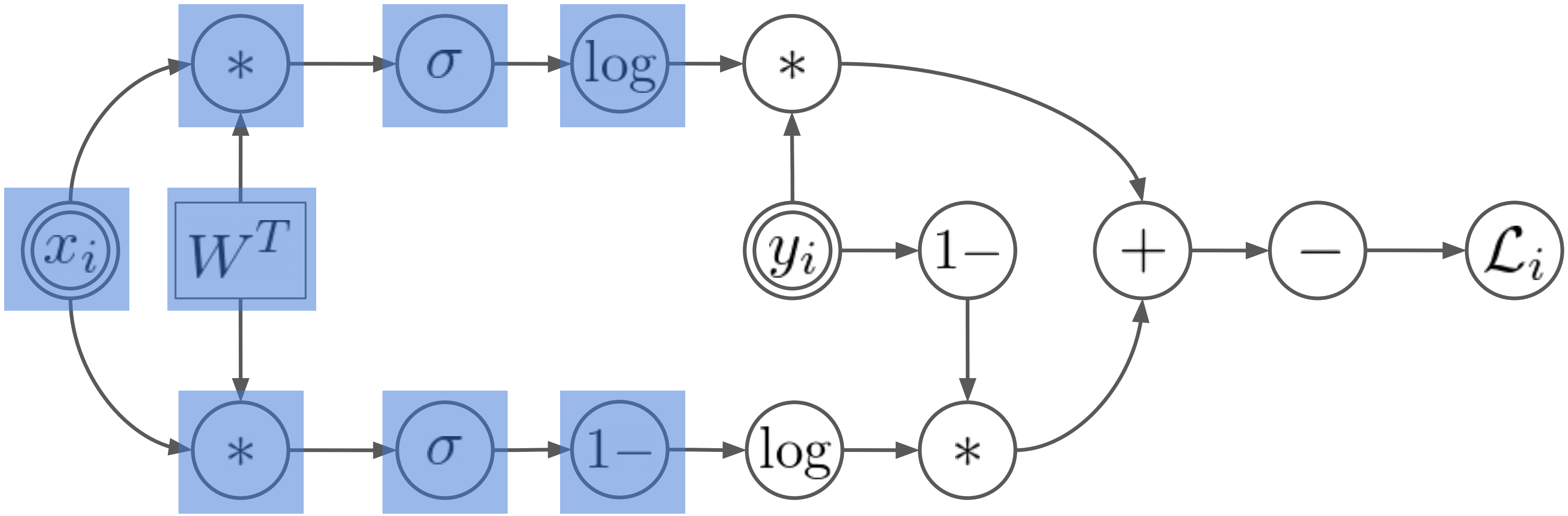
Computational Graph

$$\mathcal{L}_i = -\left[y_i \log \sigma(x_i W^T) + (1 - y_i) \log (1 - \sigma(x_i W^T))\right]$$



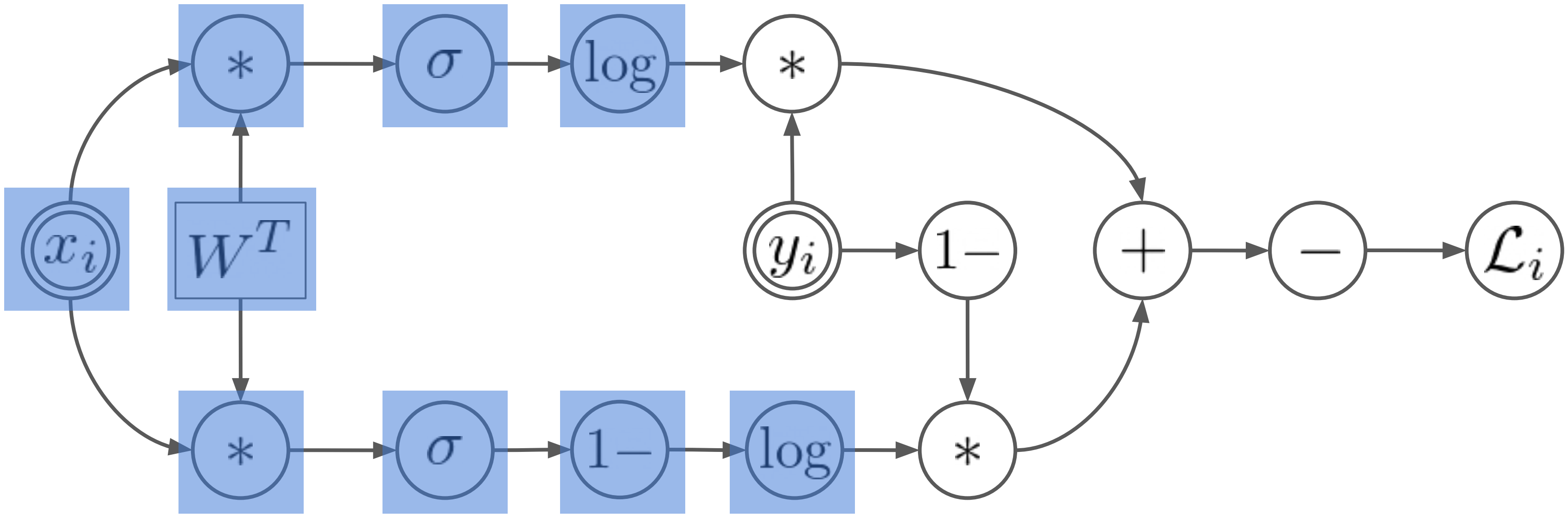
Computational Graph

$$\mathcal{L}_i = -\left[y_i \log \sigma(x_i W^T) + (1 - y_i) \log\left(1 - \sigma(x_i W^T)\right)\right]$$



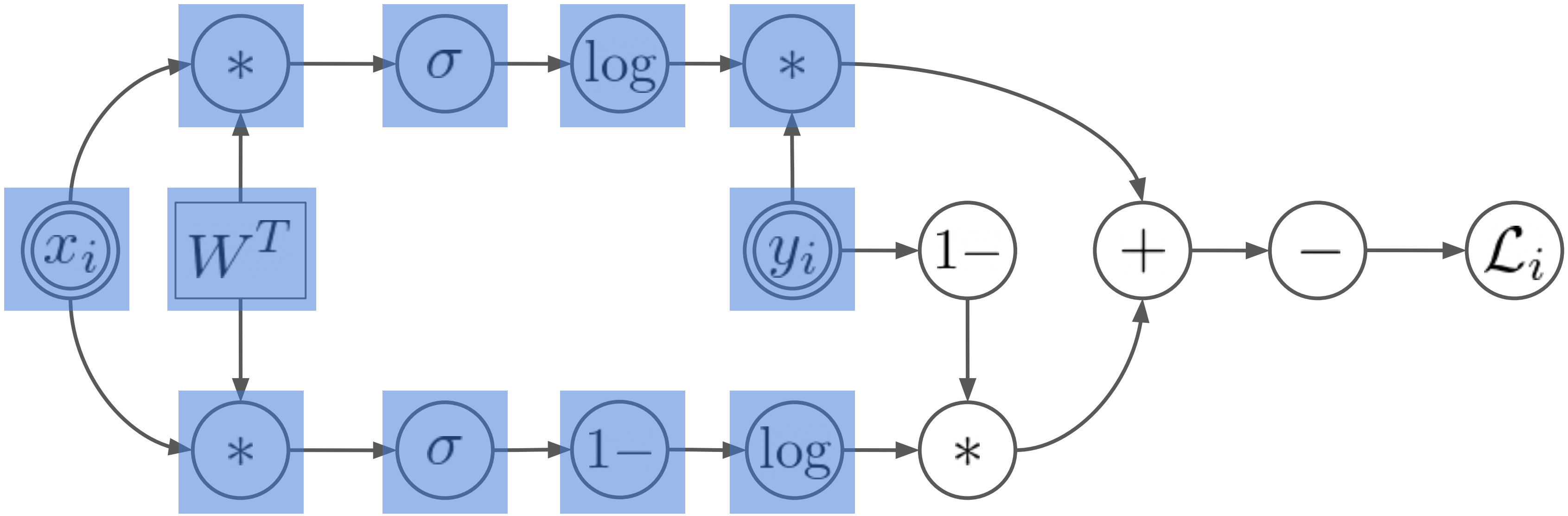
Computational Graph

$$\mathcal{L}_i = -\left[y_i \log \sigma(x_i W^T) + (1 - y_i) \log\left(1 - \sigma(x_i W^T)\right)\right]$$



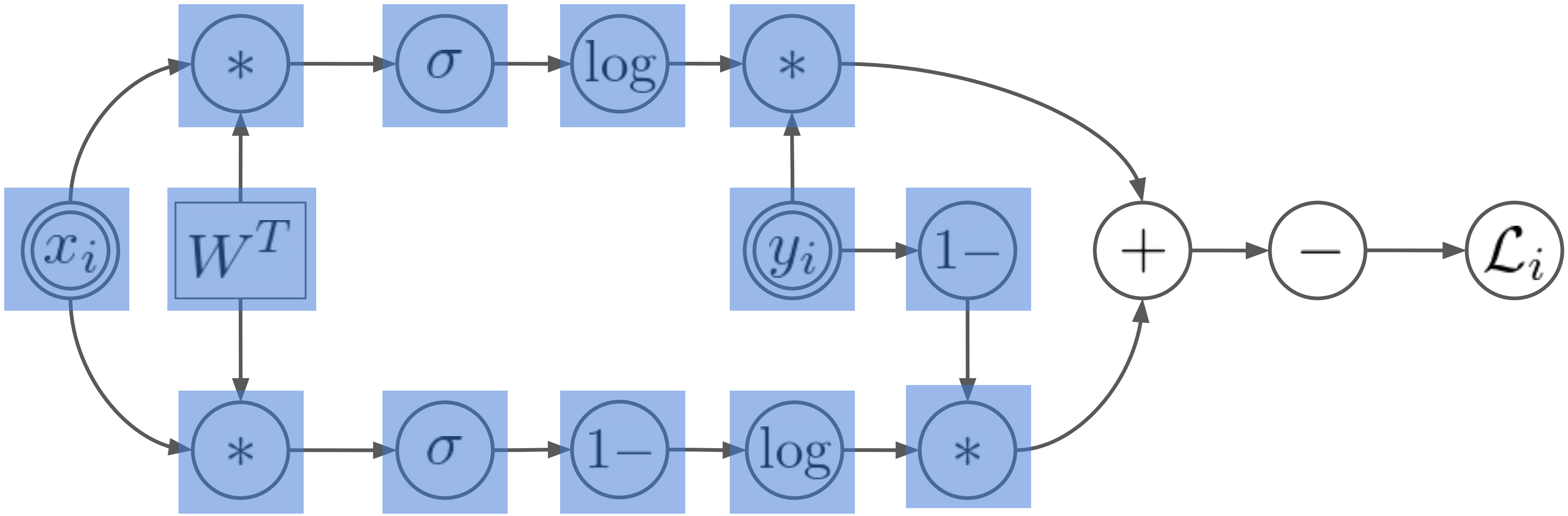
Computational Graph

$$\mathcal{L}_i = -\left[y_i \log \sigma(x_i W^T) + (1 - y_i) \log\left(1 - \sigma(x_i W^T)\right)\right]$$



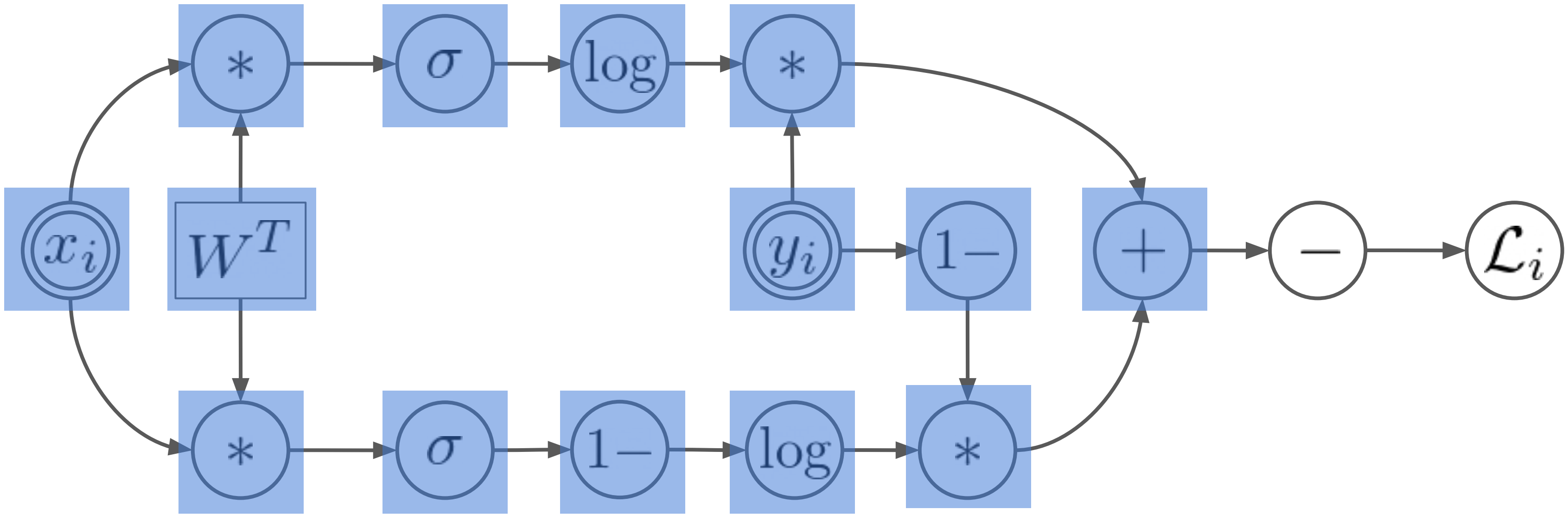
Computational Graph

$$\mathcal{L}_i = -\left[y_i \log \sigma(x_i W^T) + (1 - y_i) \log\left(1 - \sigma(x_i W^T)\right)\right]$$



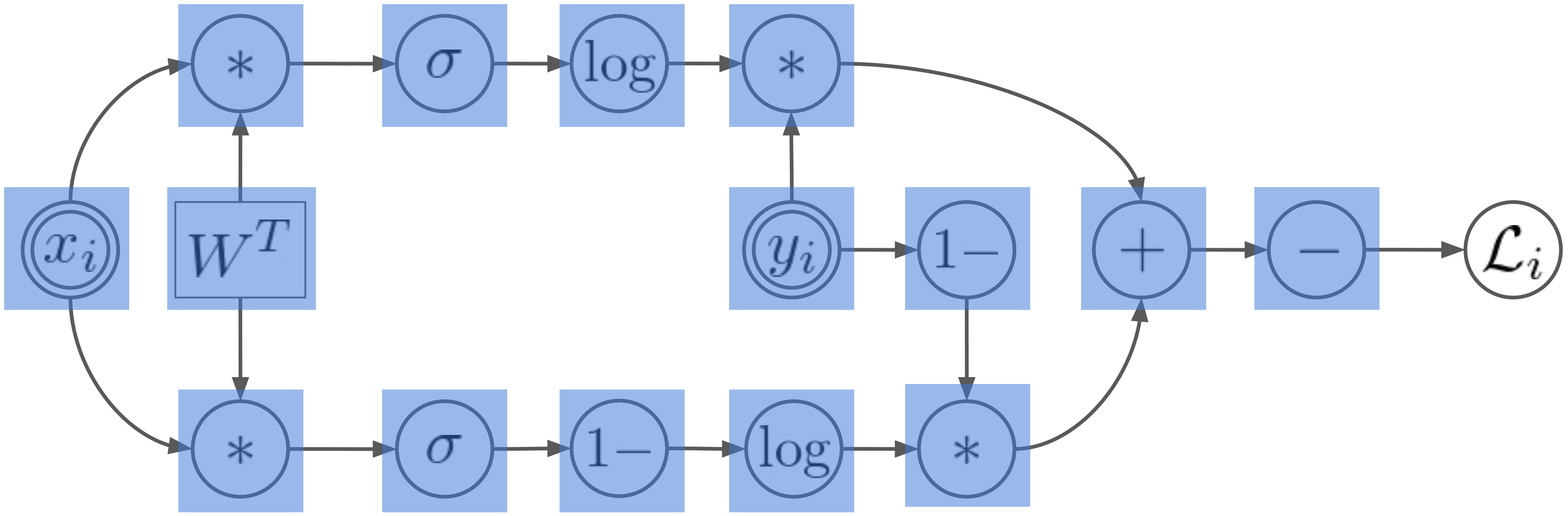
Computational Graph

$$\mathcal{L}_i = -\left[y_i \log \sigma(x_i W^T) + (1 - y_i) \log \left(1 - \sigma(x_i W^T)\right)\right]$$



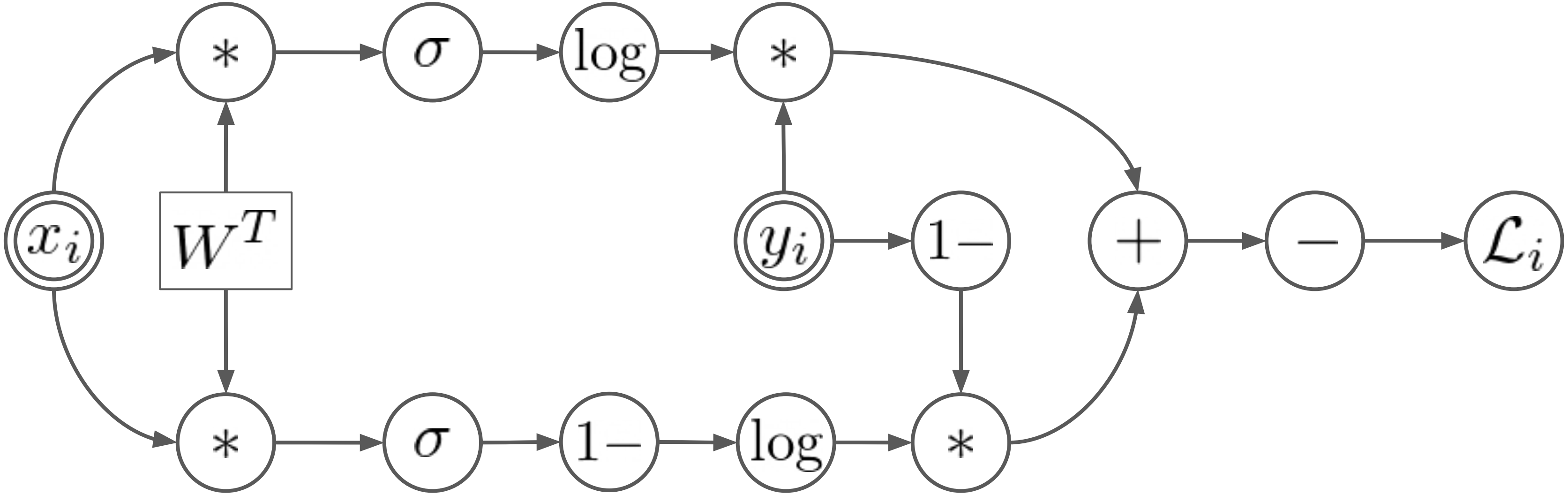
Computational Graph

$$\mathcal{L}_i = -\left[y_i \log \sigma(x_i W^T) + (1 - y_i) \log\left(1 - \sigma(x_i W^T)\right)\right]$$



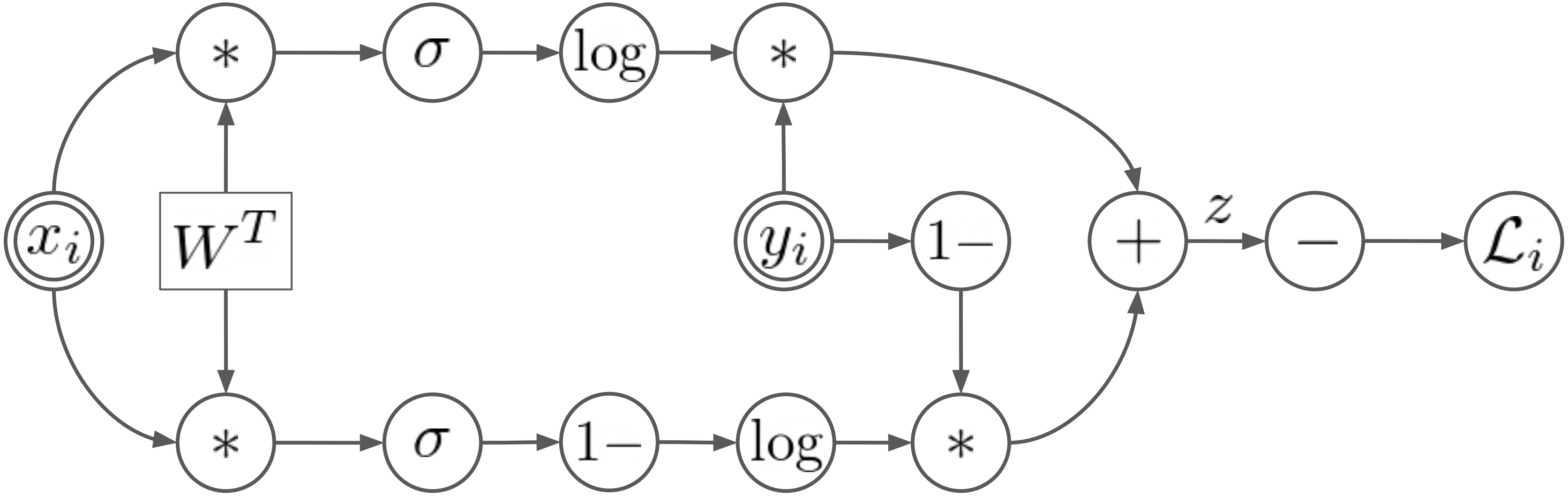
Computational Graph

$$\frac{\partial \mathcal{L}_i}{\partial W^T} = ?$$



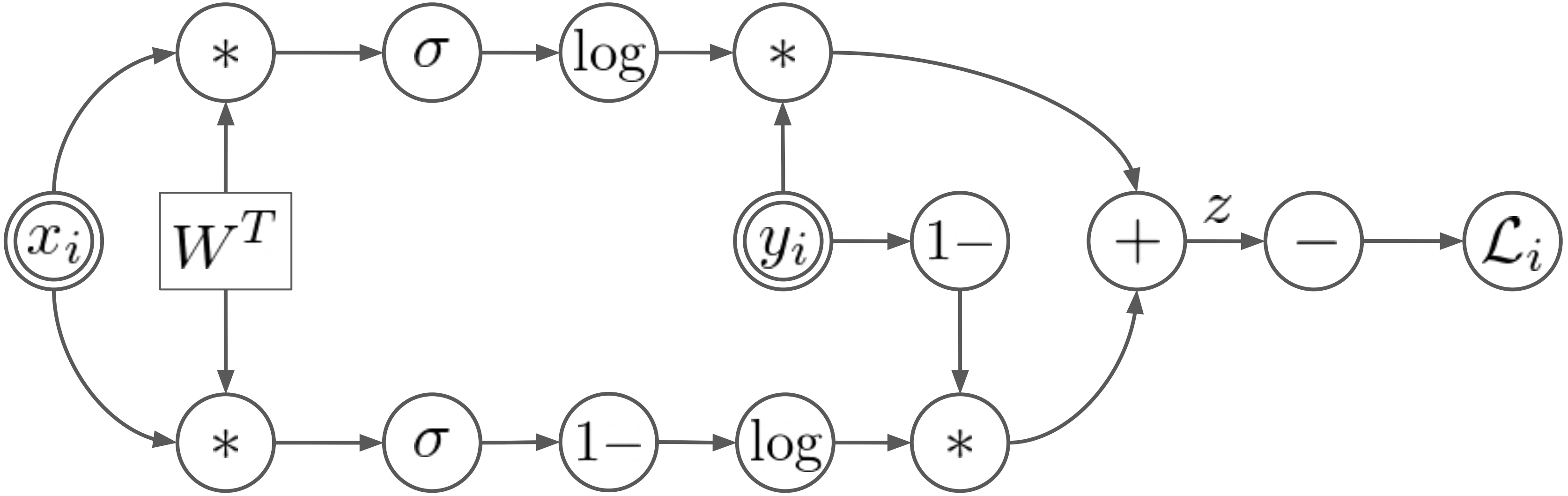
Computational Graph

$$\frac{\partial \mathcal{L}_i}{\partial W^T} = ?$$



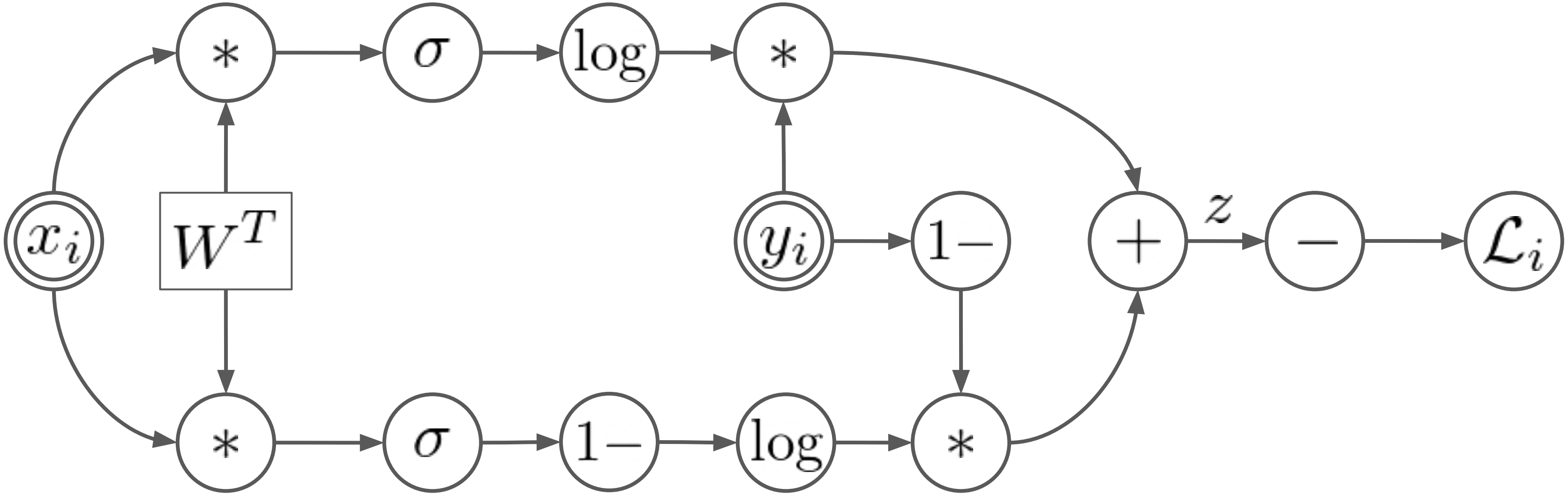
Computational Graph

$$\frac{\partial \mathcal{L}_i}{\partial W^T} = \frac{\partial \mathcal{L}_i}{\partial z} \frac{\partial z}{\partial W^T}$$



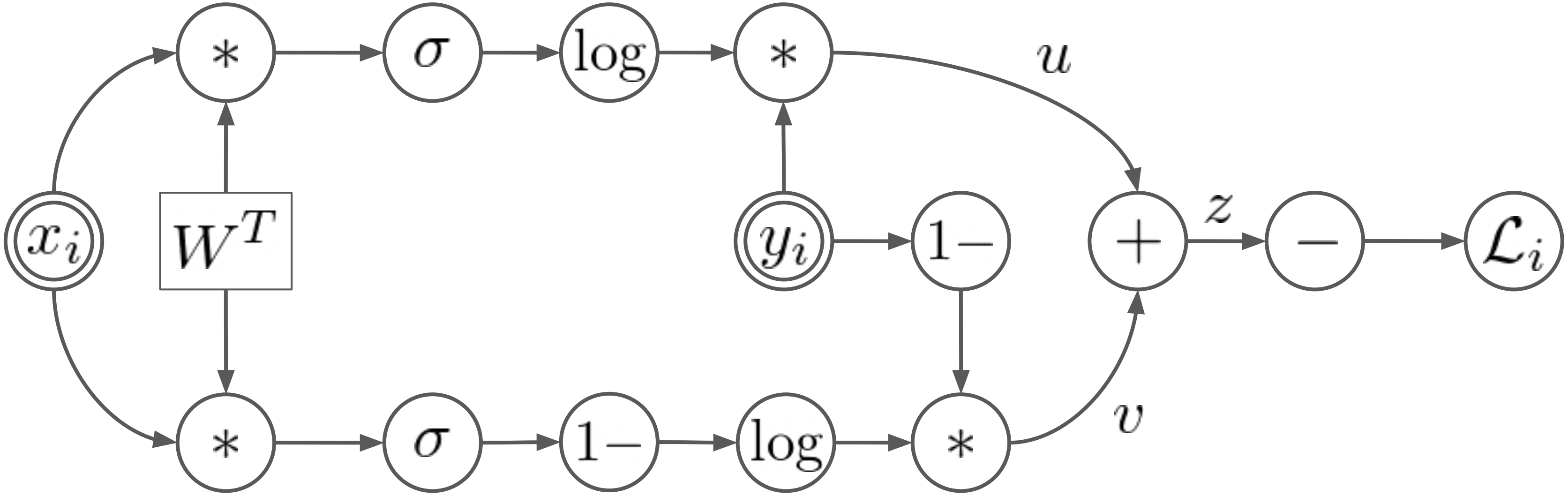
Computational Graph

$$\frac{\partial \mathcal{L}_i}{\partial W^T} = \overbrace{\frac{\partial \mathcal{L}_i}{\partial z}}^{\text{known}} \frac{\partial z}{\partial W^T}$$



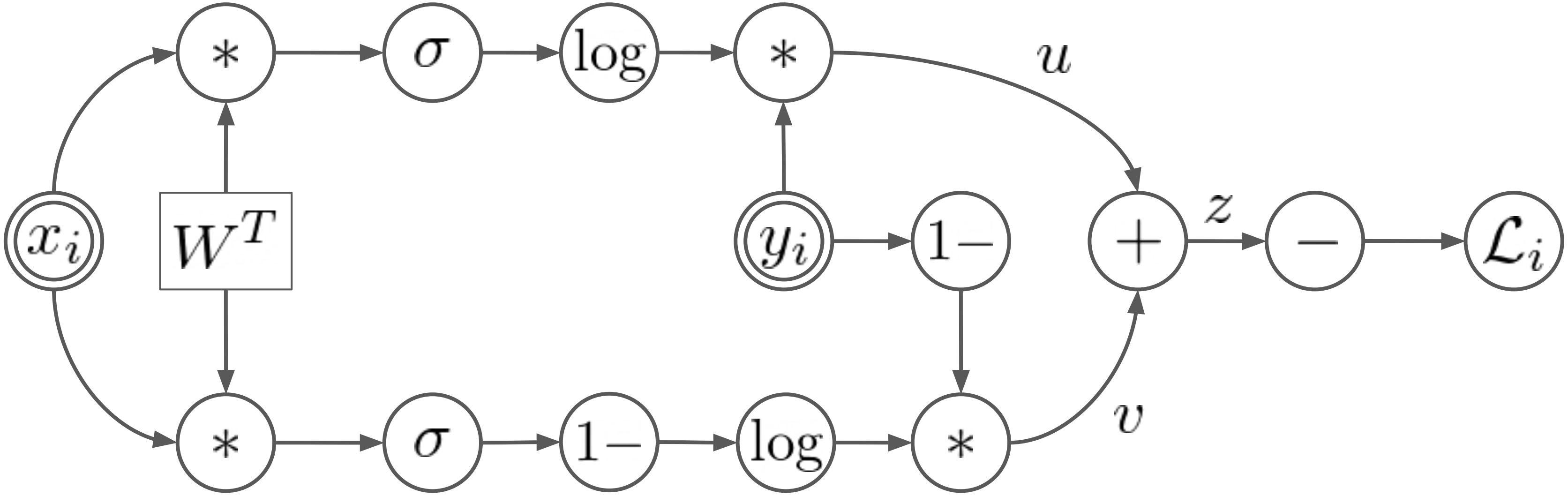
Computational Graph

$$\frac{\partial \mathcal{L}_i}{\partial W^T} = \overbrace{\frac{\partial \mathcal{L}_i}{\partial z}}^{\text{known}} \overbrace{\frac{\partial z}{\partial W^T}}^{\text{repeat}}$$

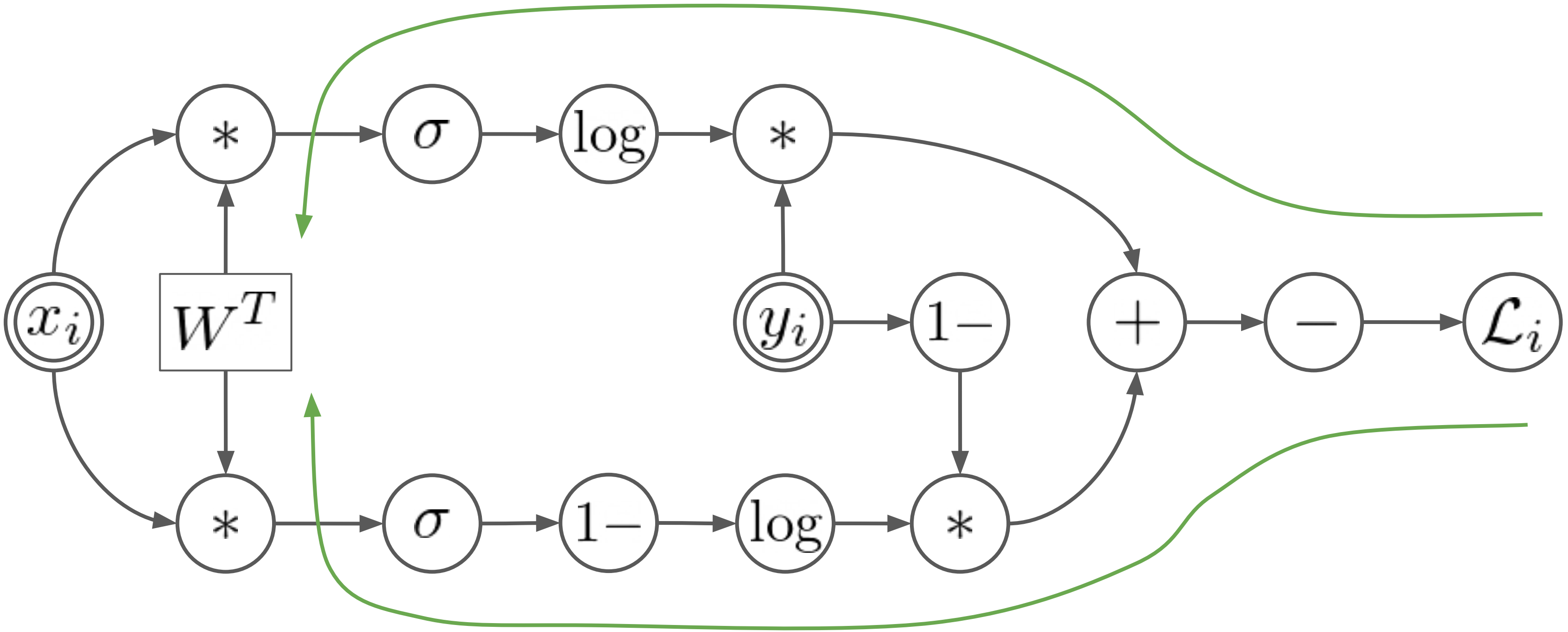


Computational Graph

$$\frac{\partial \mathcal{L}_i}{\partial W^T} = \frac{\partial \mathcal{L}_i}{\partial z}\left(\frac{\partial z}{\partial u}\frac{\partial u}{\partial W^T} + \frac{\partial z}{\partial v}\frac{\partial v}{\partial W^T}\right)$$

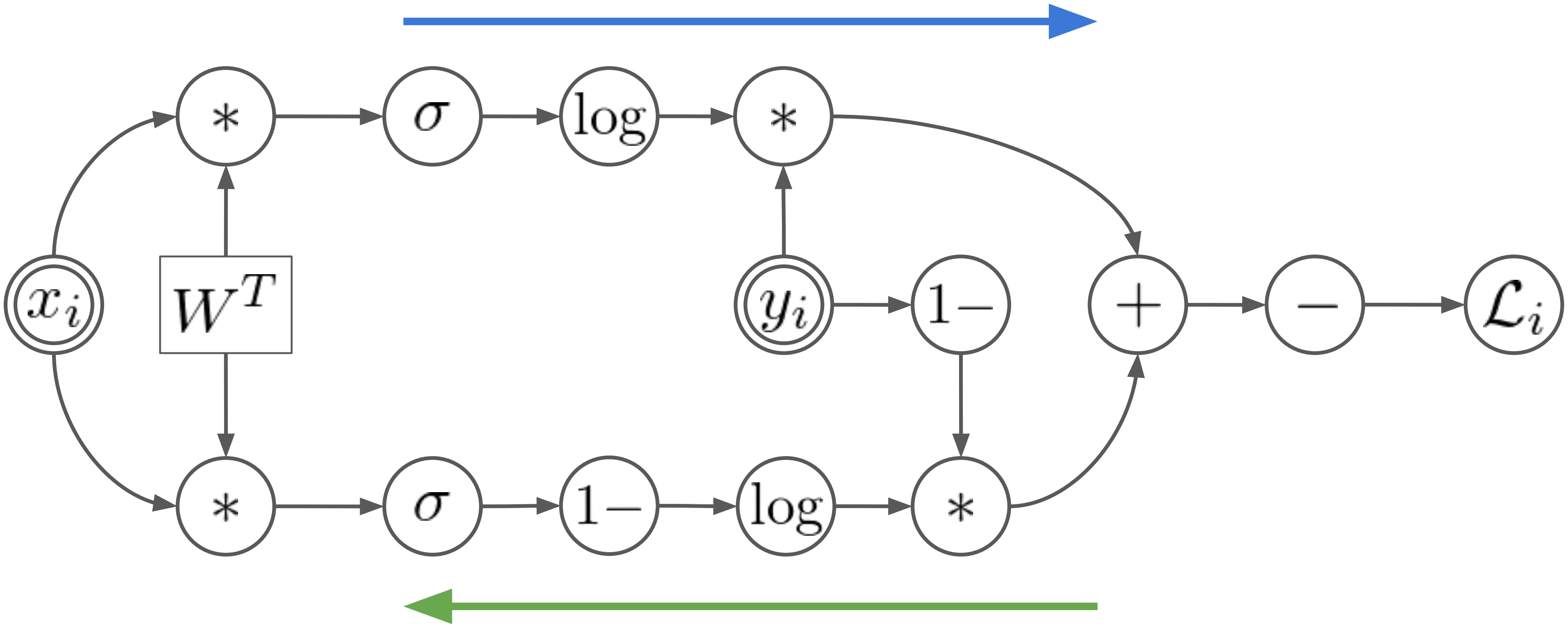


Computational Graph: Backprop



Computational Graph

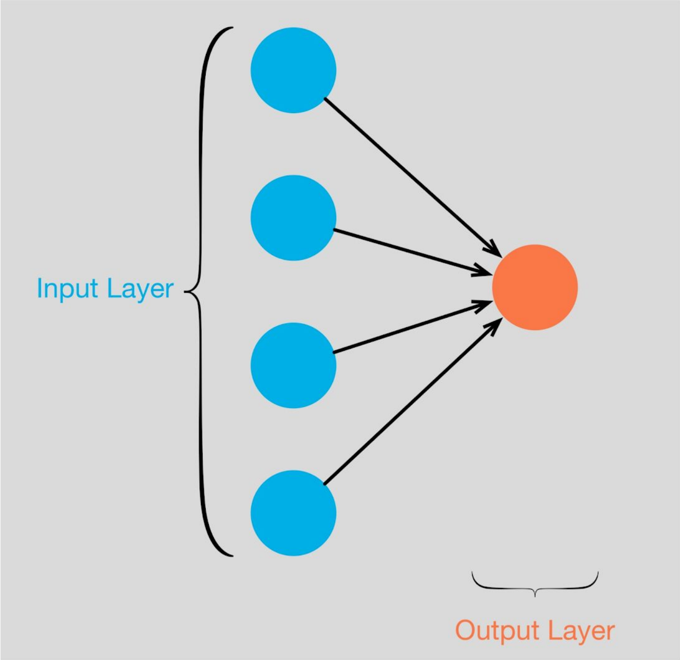
Forward Pass: Compute output



Backward Pass: Compute gradients

Logistic Regression

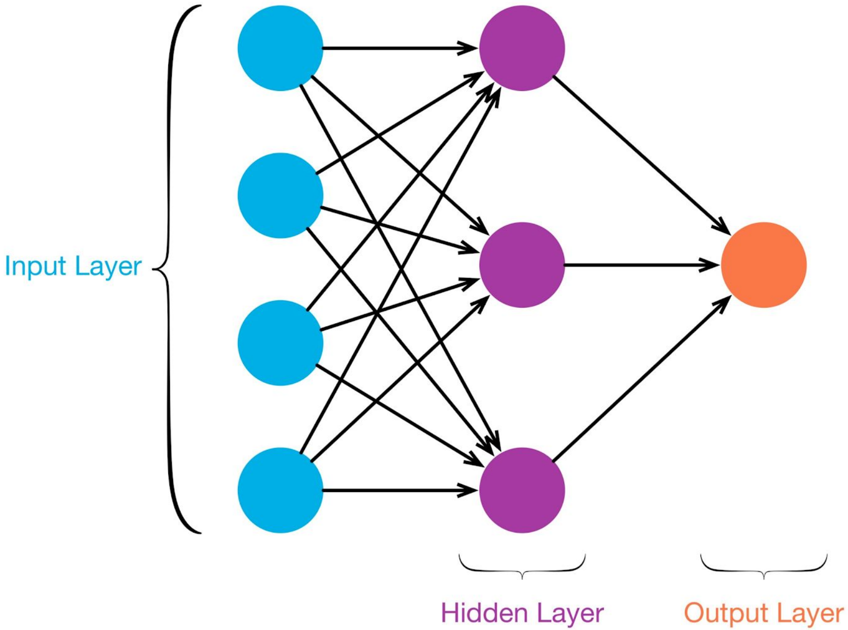
$$p(y_i = 1 | x_i, W) = \sigma(x_i W^T)$$



https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f

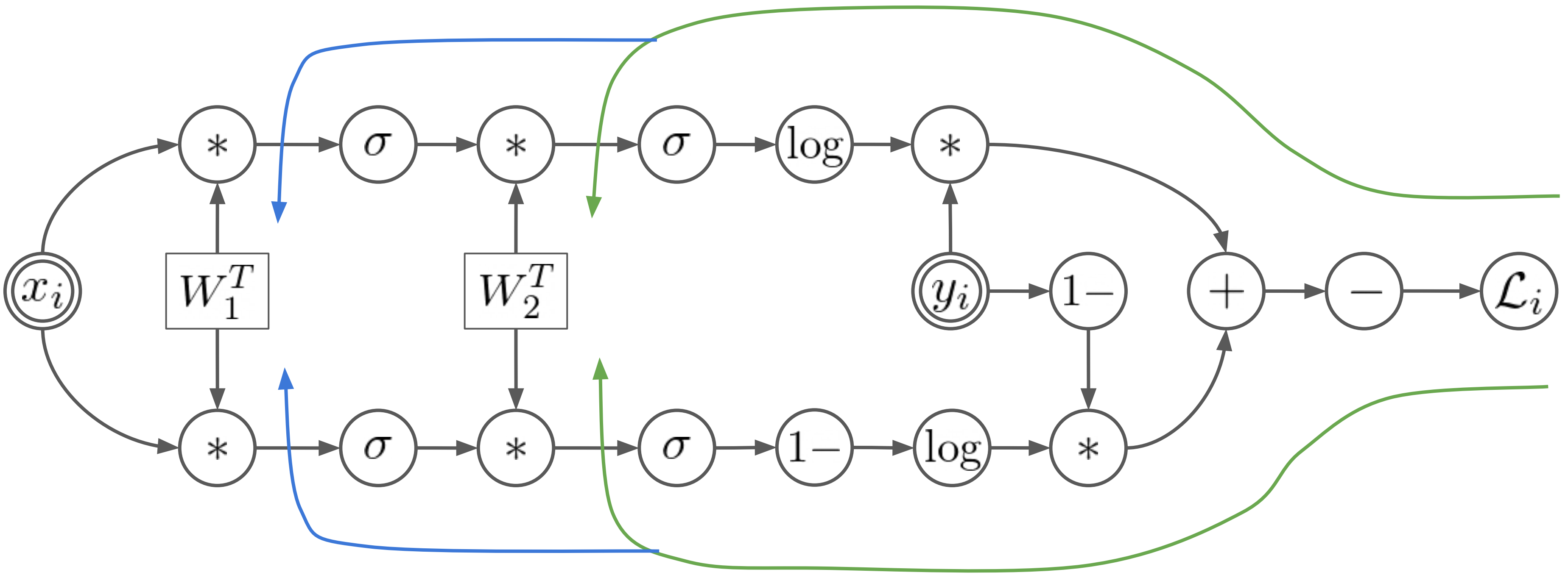
2-layered Logistic Regression

$$p(y_i = 1 | x_i, W_1, W_2) = \sigma(\sigma(x_i W_1^T) W_2^T)$$



https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f

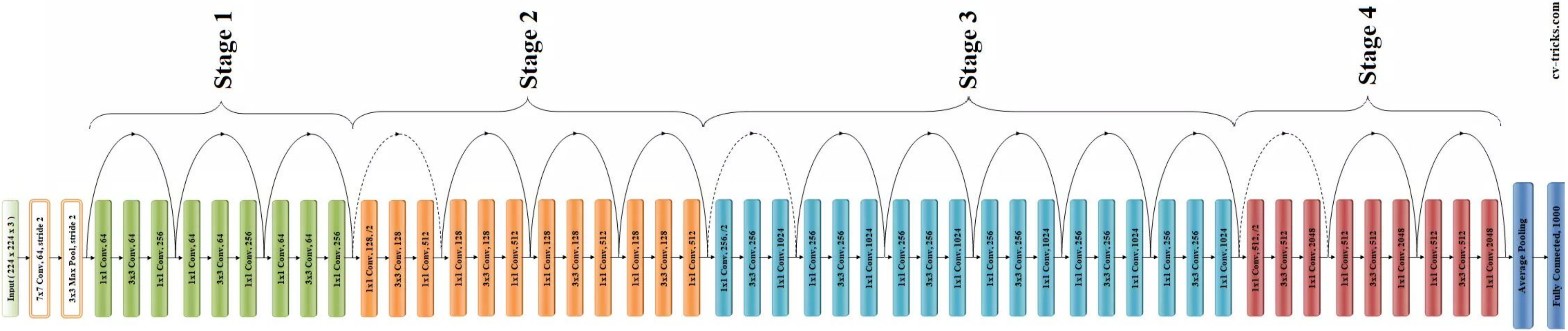
2-layered Logistic Regression



1-layered vs 2-layered Logistic Regression

Demo...

Deep Neural Networks: ResNet50



https://cv-tricks.com/keras/understand-implement-resnets/

Differentiable Programming

Automatic differentiation:

1. Write a function: `def f(...)`
2. Automatically generate: `grad_f(...)`

Parameter learning:

- lots of algorithms based on stochastic gradient descent

PYTORCH *

Three main components:

- Tensor processing library both for GPU & CPU
- Autograd library
- Neural networks library

* Many deep learning and autodiff frameworks exist, PyTorch is the one I currently use the most.

2-layer Logistic Regression with Numpy

```
W1 = np.random.randn(H, D + 1)
W2 = np.random.randn(1, H)

for iter in range(1000):
    nu = sigmoid(X @ W1.T)
    mu = sigmoid(nu @ W2.T)

    nll = - y.T @ np.log(mu) - (1 - y).T @ np.log(1 - mu)

    gW1 = X.T @ (((mu - y) @ W2) * nu * (1 - nu))
    W1 -= lr * gW1.T

    gW2 = nu.T @ (mu - y)
    W2 -= lr * gW2.T
```

2-layer Logistic Regression with Torch

```
W1 = torch.randn(H, D + 1, requires_grad=True)
W2 = torch.randn(1, H, requires_grad=True)

for iter in range(1000):
    nu = torch.sigmoid(X @ W1.T)
    mu = torch.sigmoid(nu @ W2.T)
    nll = - y.T @ np.log(mu) - (1 - y).T @ np.log(1 - mu)
    nll.squeeze().backward()

    with torch.no_grad():
        W1 -= lr * W1.grad
        W2 -= lr * W1.grad
```

2-layer Logistic Regression with Torch.nn

```
class LogisticRegression(nn.Module):
    def __init__(self, input_dims, hidden_dims):
        super(LogisticRegression, self).__init__()
        self.lin1 = nn.Linear(input_dims, hidden_dims)
        self.lin2 = nn.Linear(hidden_dims, 1)

    def forward(self, x):
        h = F.sigmoid(self.lin1(x))
        y = F.sigmoid(self.lin2(h))
        return y
```

2-layer Logistic Regression with Torch.nn

```
model = LogisticRegression(D, H)
optimizer = torch.optim.Adadelta(model.parameters(), lr=lr)

for iter in range(1000):
    nll = F.nll(model(X))
    nll.backward()
    optimizer.step()
    optimizer.zero_grad()
```

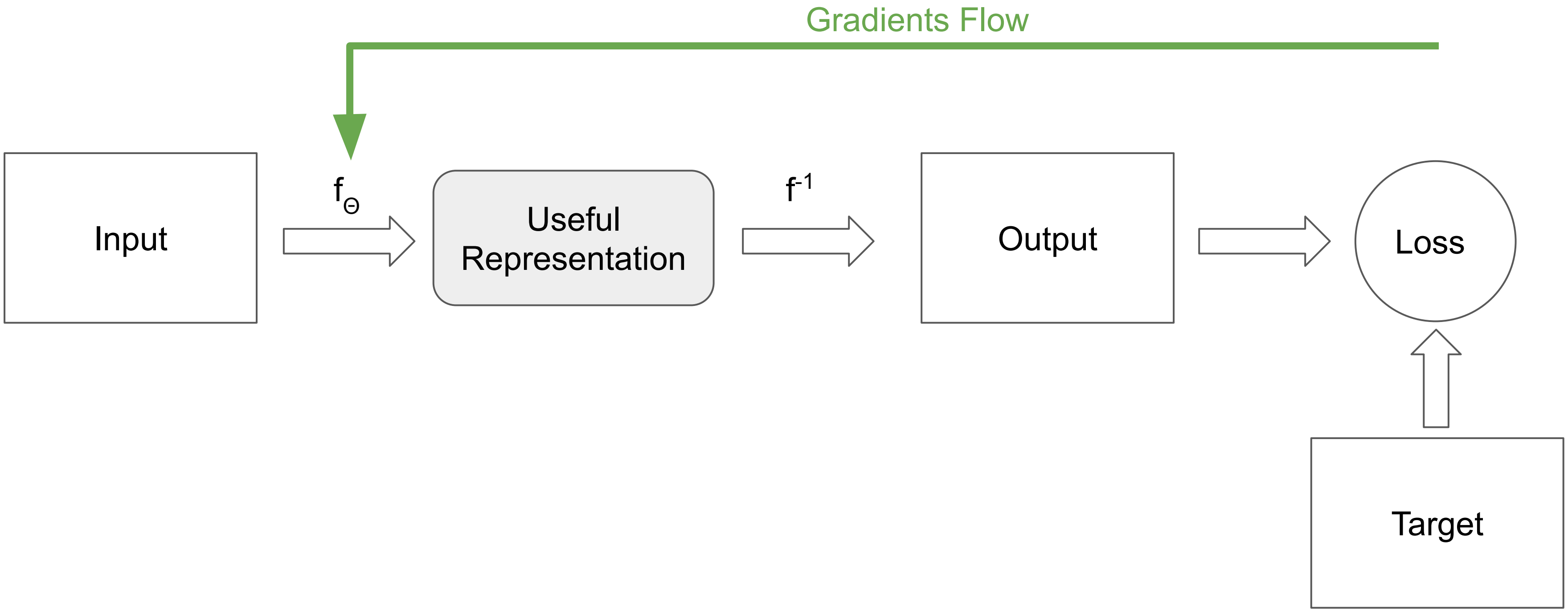
3-layer Logistic Regression with Torch.nn

```
class LogisticRegression(nn.Module):
    def __init__(self, input_dims, hidden_dims):
        super(LogisticRegression, self).__init__()
        self.lin1 = nn.Linear(input_dims, hidden_dims)
        self.lin2 = nn.Linear(hidden_dims, hidden_dims)
        self.lin3 = nn.Linear(hidden_dims, 1)

    def forward(self, x):
        h = F.sigmoid(self.lin1(x))
        h = F.sigmoid(self.lin2(h))
        y = F.sigmoid(self.lin3(h))
        return y
```

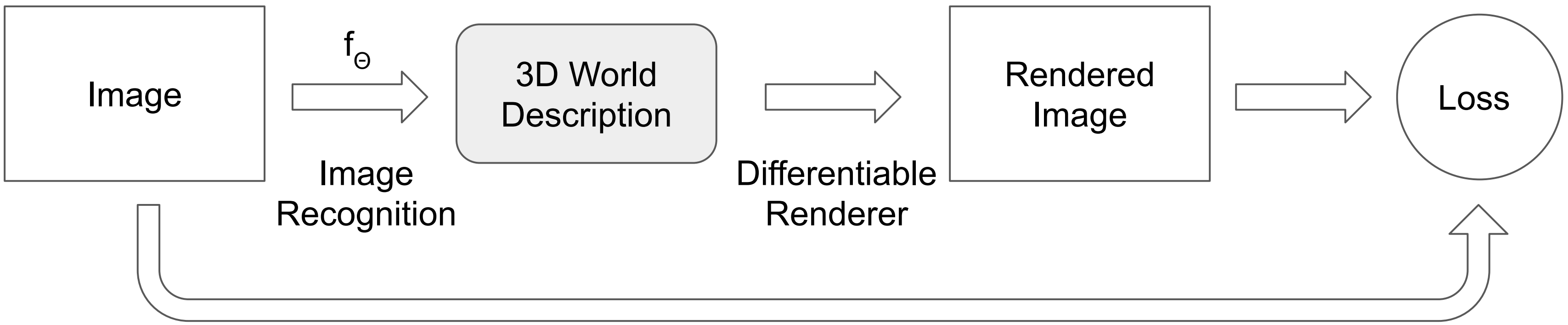
N-layer Logistic Regression with Torch.nn

```
class LogisticRegression(nn.Module):
    def __init__(self, dims: List[int]):
        super(LogisticRegression, self).__init__()
        self.lins = [
            nn.Linear(in_dims, out_dims)
            for in_dims, out_dims in zip(dims[:-1], dims[1:])
        ]

    def forward(self, x):
        res = x
        for lin in self.lins:
            res = lin(res)
        return res
```

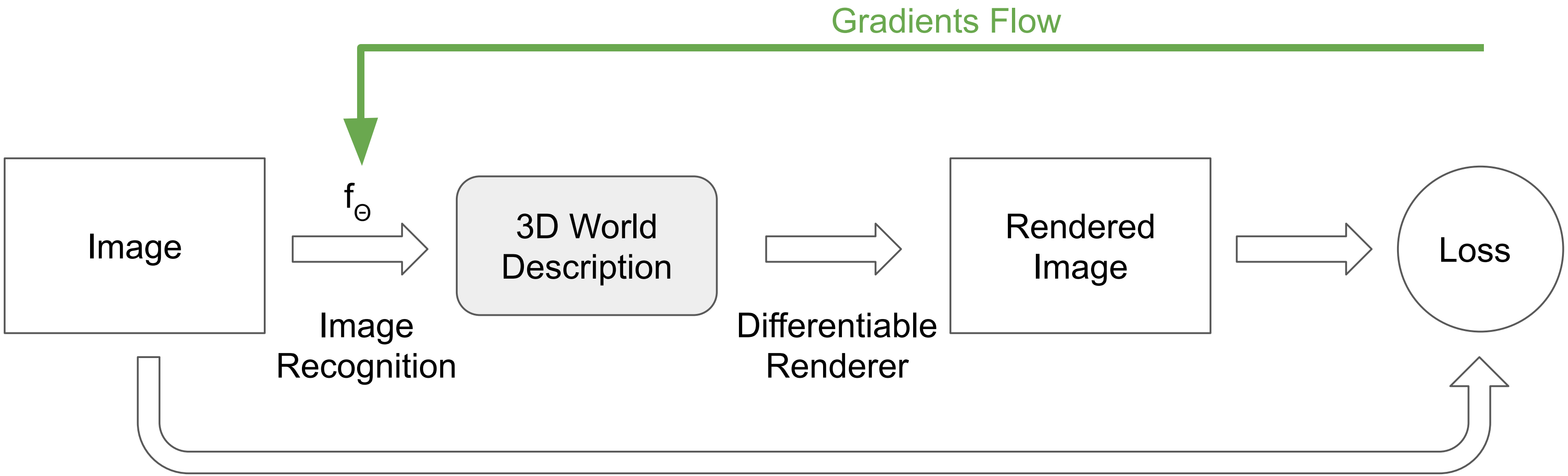
Learning Inverse Functions



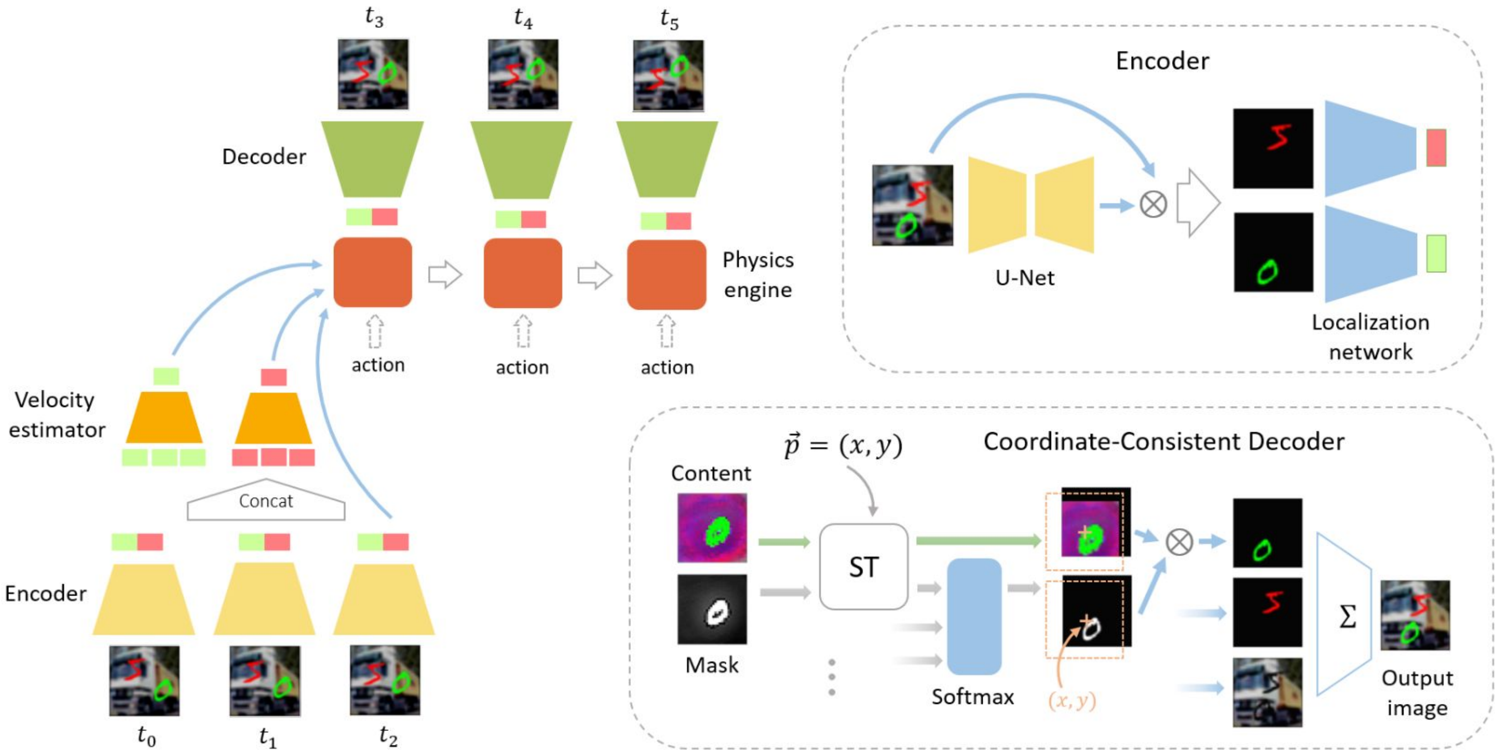
Differentiable Renderer (Inverse Graphics)



Differentiable Renderer (Inverse Graphics)

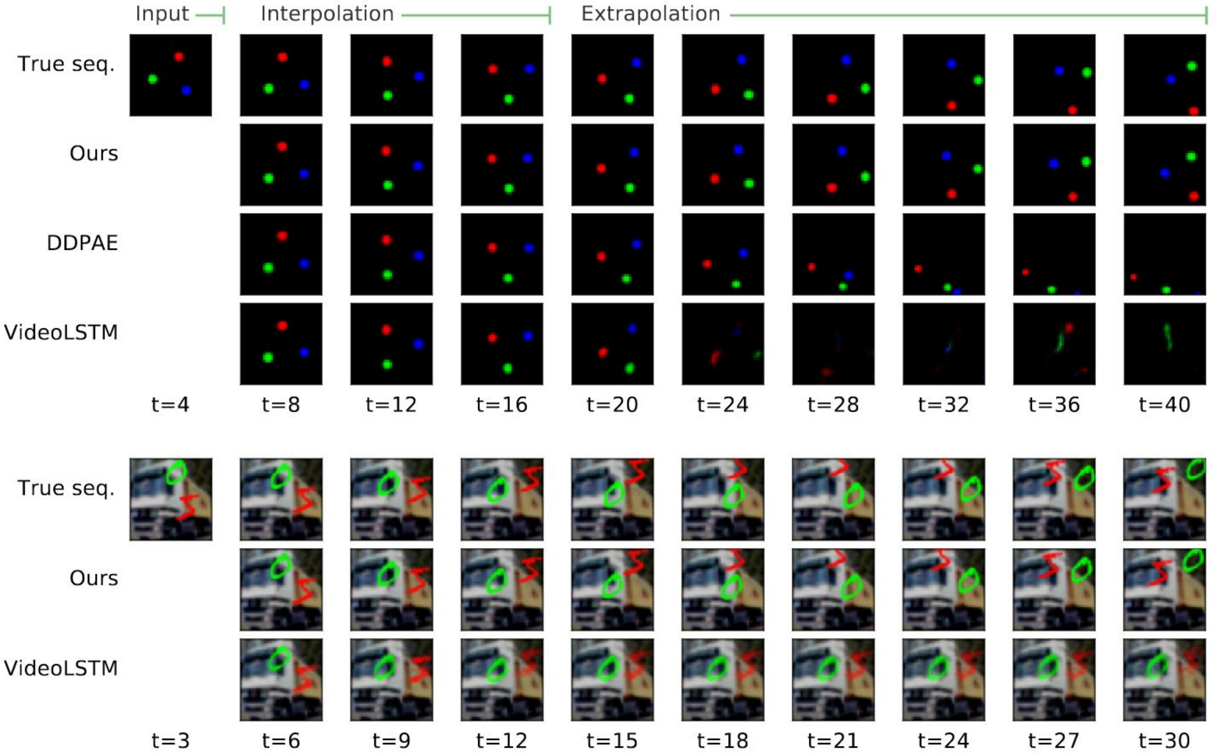


Physics-as-Inverse-Graphics



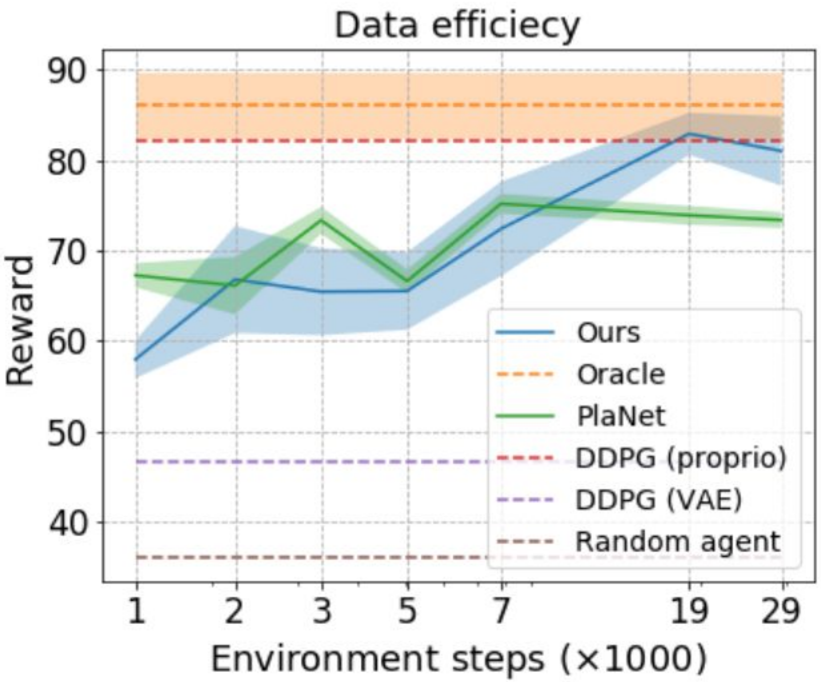
Jaques et al, ICLR 2020, https://arxiv.org/pdf/1905.11169.pdf

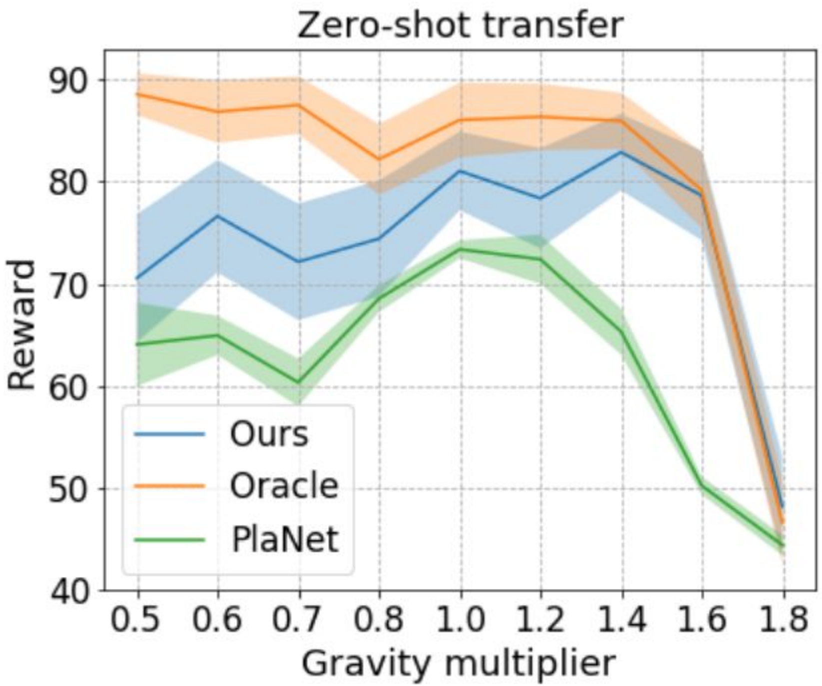
Learn Physical Parameters



Jaques et al, ICLR 2020, https://arxiv.org/pdf/1905.11169.pdf

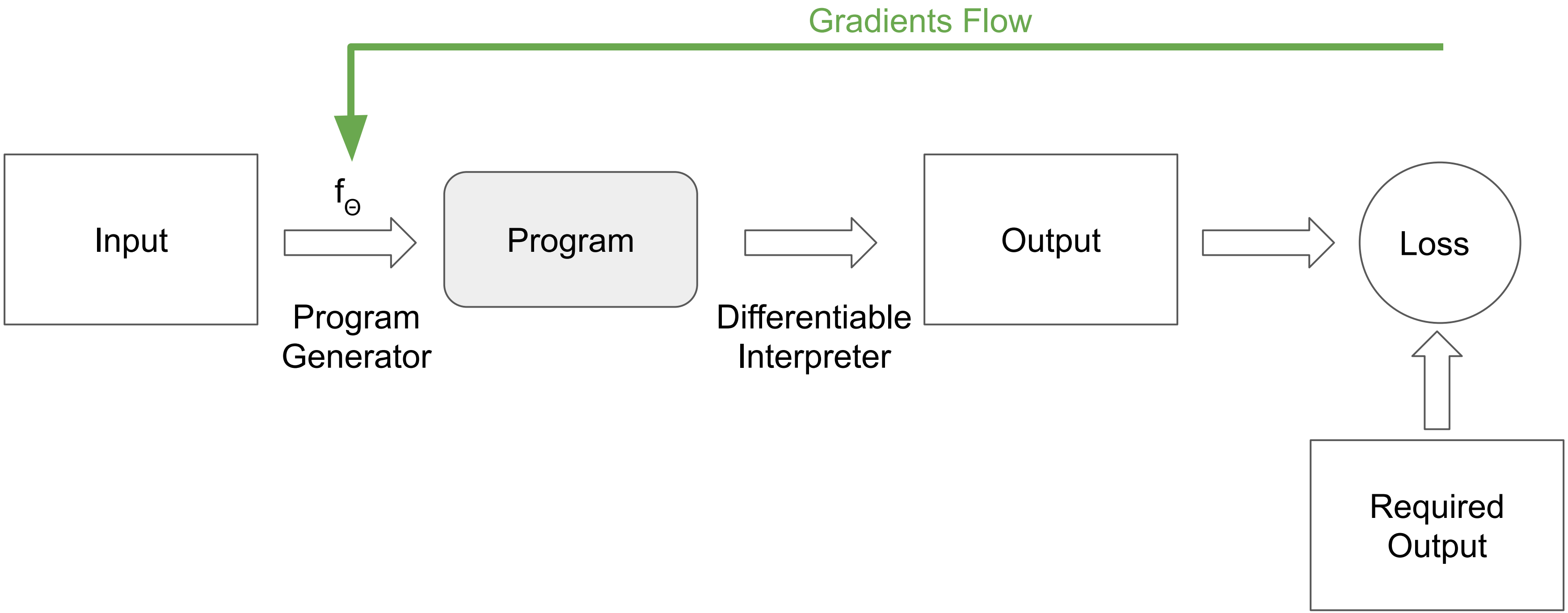
Vision Based Model Predictive Control





Jaques et al, ICLR 2020, https://arxiv.org/pdf/1905.11169.pdf

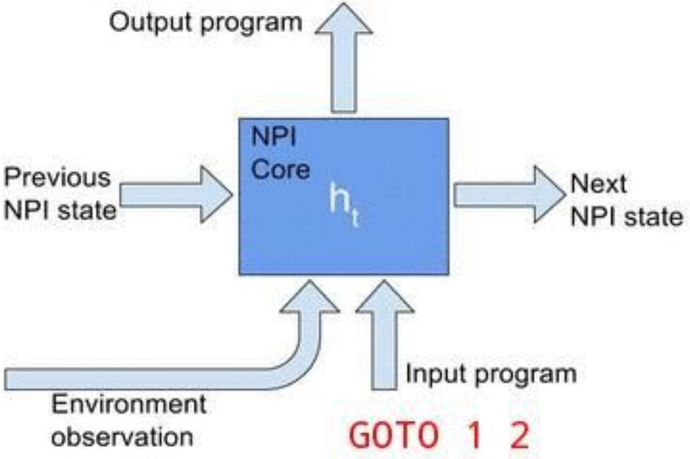
Differentiable Program Interpreter



Neural Program Interpreter

Car rendering

NPI inference



Generated commands

GOTO 1 2

Reed & de Freitas, ICLR 2015, https://arxiv.org/abs/1511.06279

Differentiable Program with Neural Libraries

Declaration & initialization

```
# constants
max_int = 15; n_instr = 3; T = 45
W = 5; H = 3; w = 50; h = 50

# variables
img_grid = InputTensor(w, h)[W, H]
init_X = Input(W)
init_Y = Input(H)
final_X = Output(W)
final_Y = Output(Y)
path_len = Output(max_int)

instr = Param(4)[n_instr]
goto  = Param(n_instr)[n_instr]

X = Var(W)[T]
Y = Var(H)[T]
dir = Var(5)[T]
reg = Var(max_int)[T]
instr_ptr = Var(n_instr)[T]

X[0].set_to(init_X)
Y[0].set_to(init_Y)
dir[0].set_to(1)
reg[0].set_to(0)
instr_ptr[0].set_to(0)
```

Instruction Set

```
# Discrete operations
@Runtime([max_int], max_int)
def INC(a):
  return (a + 1) % max_int

@Runtime([max_int], max_int)
def DEC(a):
  return (a - 1) % max_int

@Runtime([W, 5], W)
def MOVE_X(x, dir):
  if   dir == 1: return (x + 1) % W # →
  elif dir == 3: return (x - 1) % W # ←
  else: return x

@Runtime([H, 5], H)
def MOVE_Y(y, dir):
  if   dir == 2: return (y - 1) % H # ↑
  elif dir == 4: return (y + 1) % H # ↓
  else: return y

# Learned operations
@Learn([Tensor(w, h)], 5,
       hid_sizes=[256,256])
def LOOK(img):
  pass
```

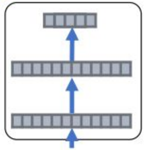
Execution model

```
for t in range(T - 1):
  if dir[t] == 0:  # halted
    dir[t + 1].set_to(dir[t])
    X[t + 1].set_to(X[t])
    Y[t + 1].set_to(Y[t])
    reg[t + 1].set_to(reg[t])
  else:
    with instr_ptr[t] as i:
      if instr[i] == 0:   # INC
        reg[t + 1].set_to(INC(reg[t + 1]))
      if instr[i] == 1:   # DEC
        reg[t + 1].set_to(DEC(reg[t + 1]))
      else:
        reg[t + 1].set_to(reg[t])

      if instr[i] == 2:   # MOVE
        X[t + 1].set_to(MOVE_X(X[t], dir[t]))
        Y[t + 1].set_to(MOVE_Y(Y[t], dir[t]))
      else:
        X[t + 1].set_to(X[t])
        X[t + 1].set_to(Y[t])

      if instr[i] == 3:   # LOOK
        with pos[t] as p:
          dir[t + 1].set_to(LOOK(img_grid[p]))
      else:
        dir[t + 1].set_to(dir[t])

  instr_ptr[t + 1].set_to(goto[i])

final_X.set_to(X[T - 1])
final_Y.set_to(X[T - 1])
path_len.set_to(reg)
```

Solution

```
instr = [3,2,0]
 goto = [1,2,0]

L0
if not halted:
  dir = LOOK
  halt if dir == 0
goto L1

L1
if not halted:
  MOVE(dir)
goto L2

L2
if not halted:
  reg = INC(reg)
goto L0
```

LOOK :=



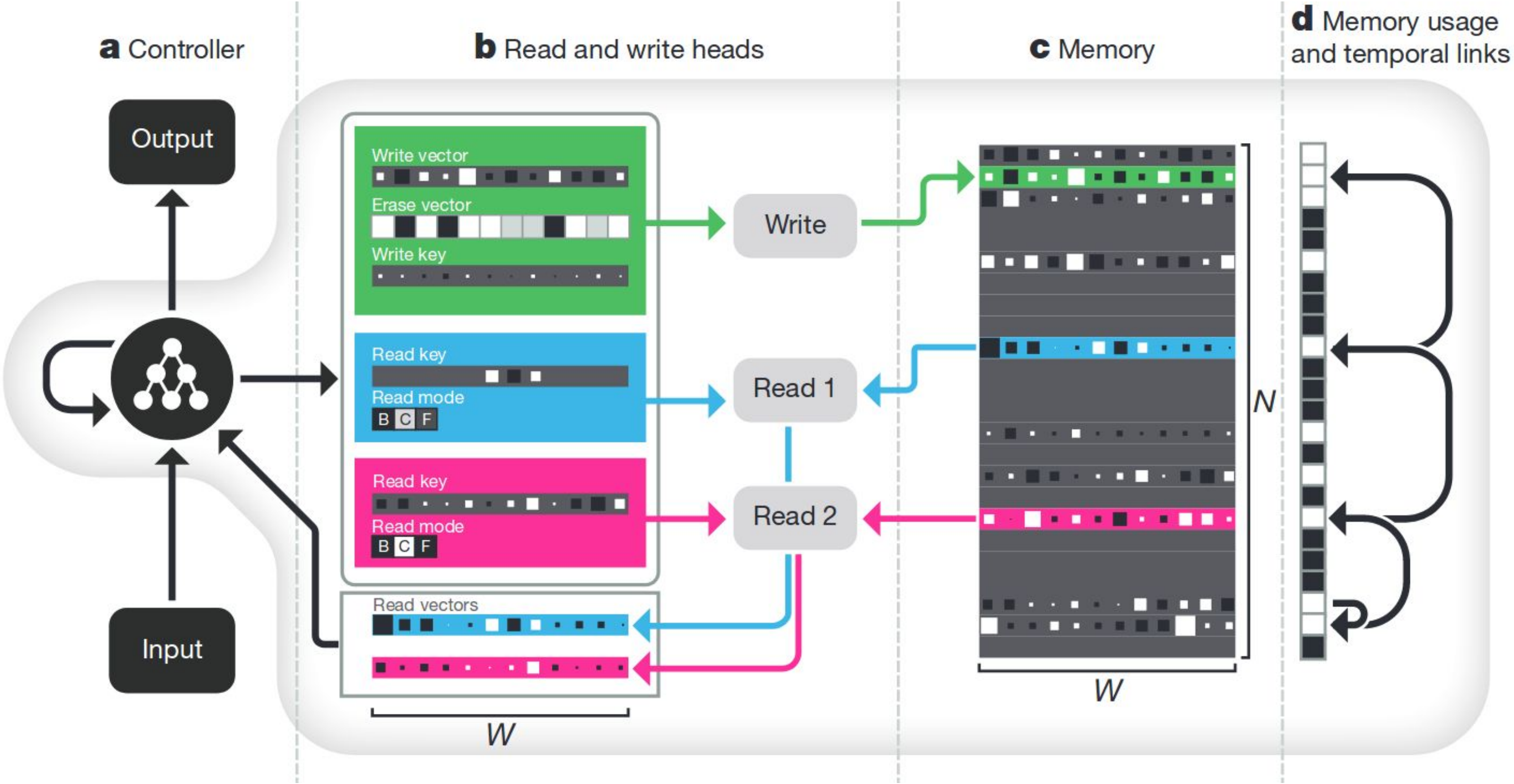
Input-output data set

img_grid =


```
init_X = 0
init_Y = 1

final_X  = 4
final_Y  = 2
path_len = 7
```

Gaunt et al, ICML 2017, https://arxiv.org/pdf/1611.02109.pdf

Differentiable Neural Computer



Graves et al., Nature 2016, https://www.nature.com/articles/nature20101