Automatic Differentiation

Outline

- Recap neural networks and backpropagation
- Simple minimization examples
- Example with partial derivatives and a compute graph
- Partial derivatives and compute graphs for a one-layer network
- Partial derivatives and compute graphs for a two-layer network
- Automatic differentiation in code

Recap: Neural Networks and Backpropagation

- Take five minutes to draw
  - Whatever will help you remember (no correct or incorrect drawings)
  - You’ll keep a running drawing log the rest of the semester

Automatic Differentiation (AD)

- Distinct from
  - Symbolic differentiation (exact, slow, must have symbolic representation)
  - Numerical differentiation (approximation/imprecise, fast, flexible)

to a degree

Py Match

- Forms of AD
  - Forward mode AD (exact, fast, flexible)
  - Reverse mode AD (exact, faster, flexible) (used by PyTorch as the default[1])

- Some terms
  - Differentiation: Process of finding a derivative
  - Derivative: The rate of change of a function (sometimes at a specific point)
  - Partial derivative: Derivative of a function with respect to one of several variables
  - Gradient: All partial derivatives

$$f(a) = 4(a-2)^2 - 3$$

Plot

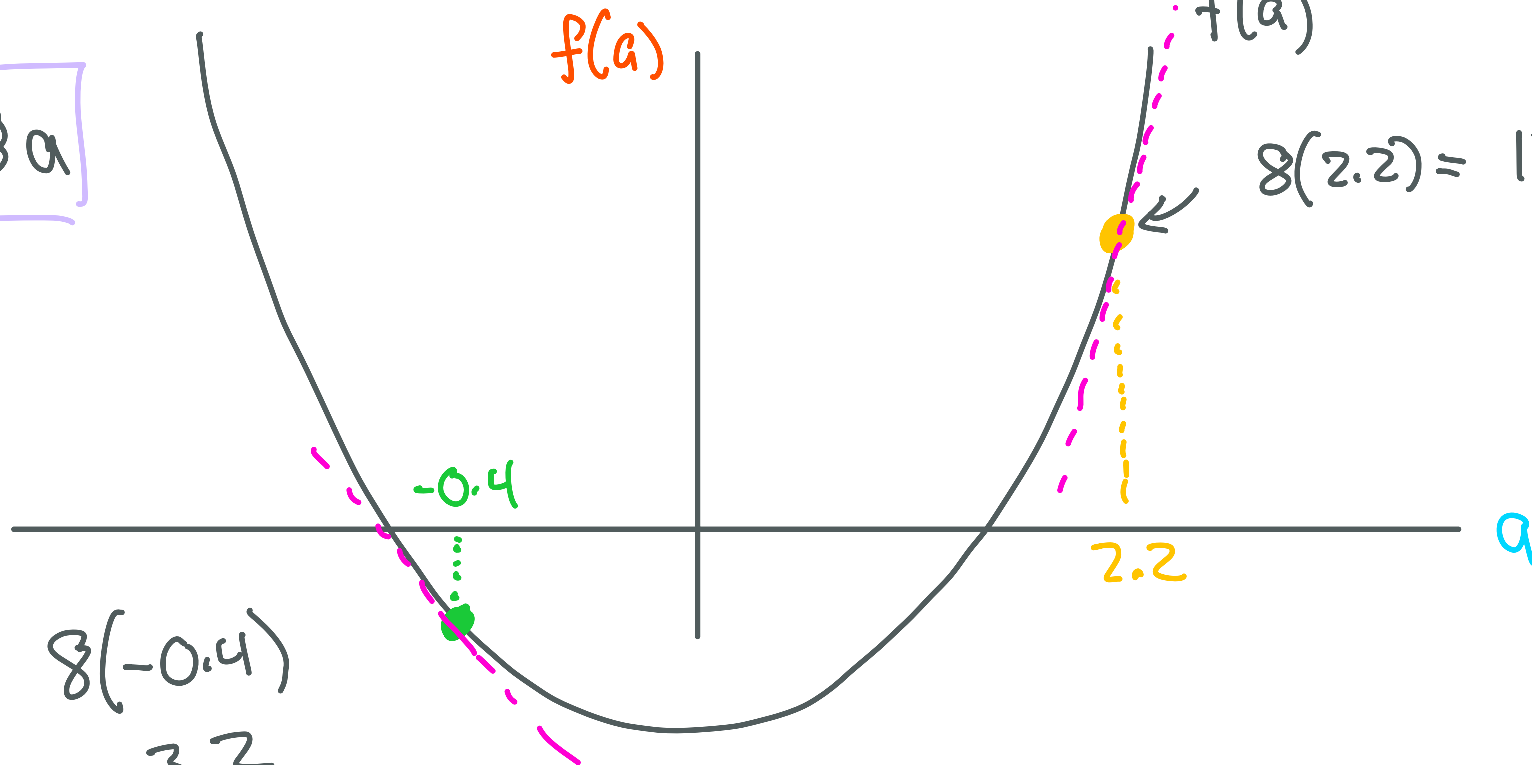
Closed-form solution

ttps://www.desmos.com/calculator/utlqlqutld

$$f(a) = 4a^2 - 3, \text{ where } a = \cancel{1.8} \text{ and } a = 2.2$$

-0.4

Plot and Gradient Descent



$$\frac{d}{da} f = \boxed{8a}$$

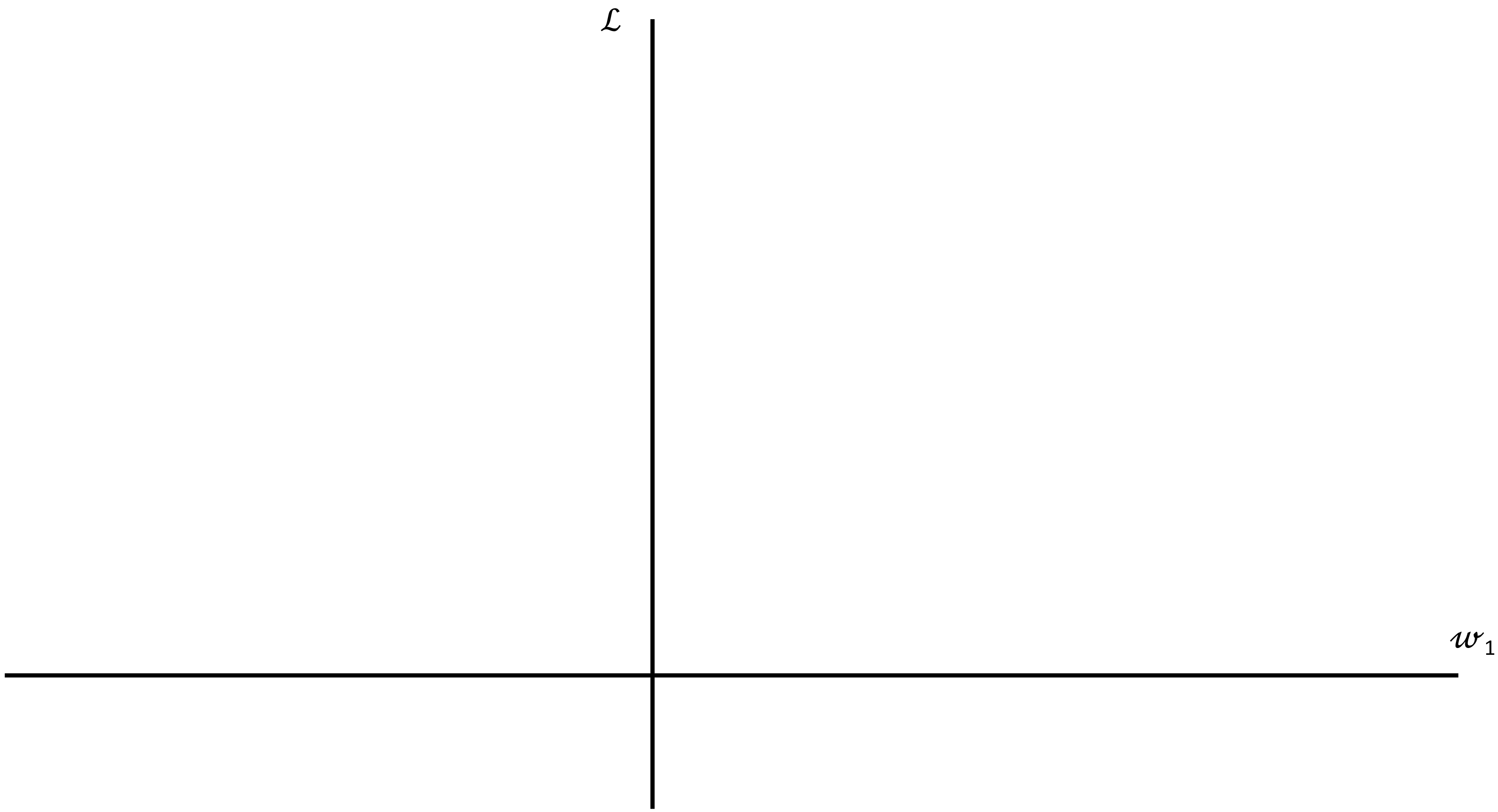
$8(2.2) = 17.6$

$8(-0.4)$

-3.2

$$z = AW + b$$

$\oplus$



$\mathcal{L}$
w_1

$$f(a, b, c, d) = b(a - c)^2 - d$$

$$\cancel{\frac{d}{da} f} \qquad \frac{\partial}{\partial a} f$$

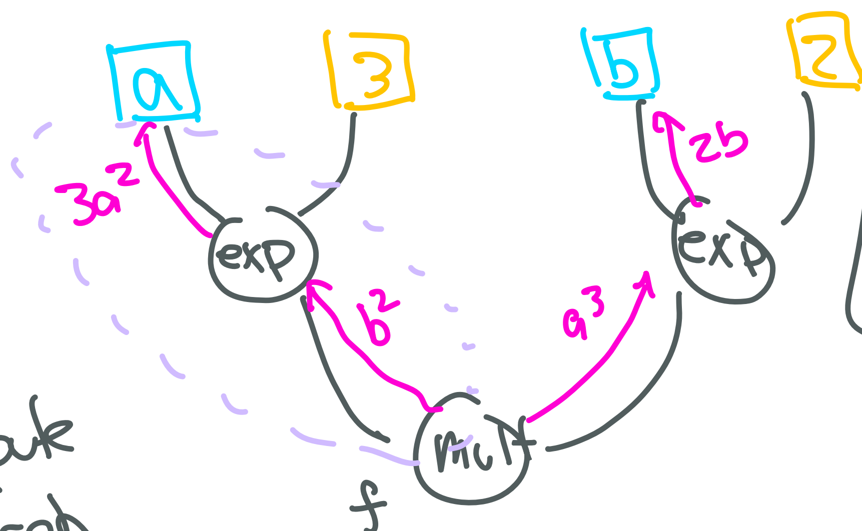
$$\nabla f$$

$$f(a,b) = a^3 b^2$$

Minimize with respect to a

f = a**3 * b**2

Tensor Tensor



Compute Graph

$$\frac{\partial f}{\partial a} = \frac{\partial}{\partial a} a^3 \cdot b^2$$

$$= 3a^2 b^2$$

[1, 2, 3] (3,)

[[1, 2, 3], [4, 5, 6]] (2,3)

[[1, 2, 3] [1, 2]] X Jagged :(

$$\boxed{\frac{\partial f}{\partial a} = 3a^2 b^2}$$

f.backwords()

One-Layer Network (MSE and Sigmoid) Compute Graph

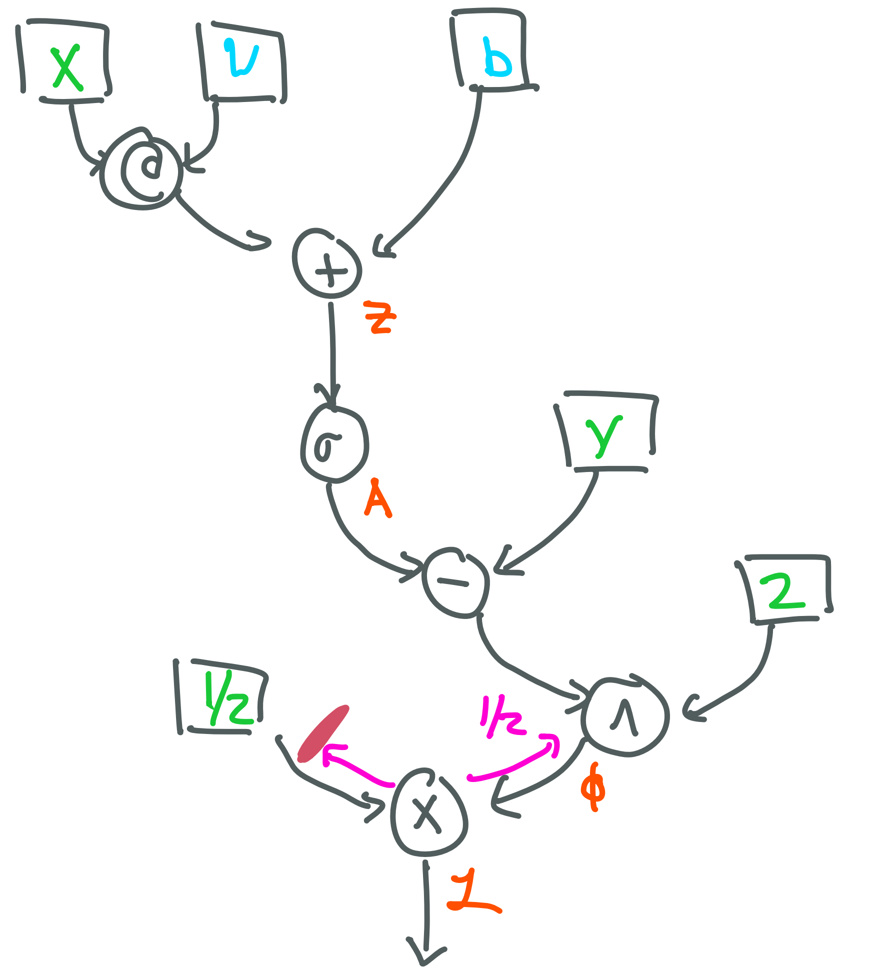
- $Z = XW^T + b$
- $A = \sigma(Z) = \sigma(XW^T + b)$
- $\mathcal{L} = \frac{1}{2}(A - Y)^2$

$\boxed{\mathcal{L}.\text{backward}()}$

$$\mathcal{L} = \underbrace{\tfrac{1}{2}}_{\alpha}\underbrace{\left(\sigma(XW^T + b) - Y\right)^2}_{\phi}$$

$$\mathcal{L} = \alpha \cdot \phi$$

$$\frac{\partial \mathcal{L}}{\partial \alpha} = \phi \quad , \quad \frac{\partial \mathcal{L}}{\partial \phi} = \alpha$$



One-Layer Network (MSE and Sigmoid) Compute Graph

- $Z = XW^T + b$
- $A = \sigma(Z) = \sigma(XW^T + b)$
- $\mathcal{L} = \frac{1}{2}(A - Y)^2$

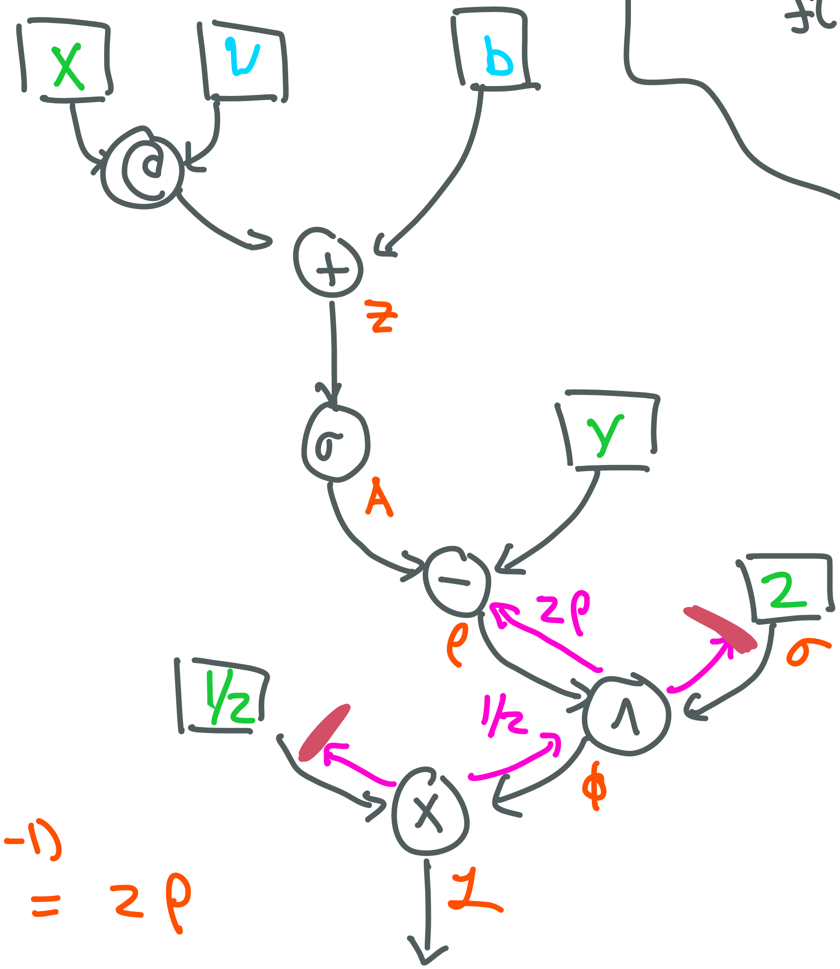
$\boxed{\mathcal{L}.\text{backward}()}$

$$\mathcal{L} = \frac{1}{2}\left(\sigma(XW^T + b) - Y\right)^2$$

A

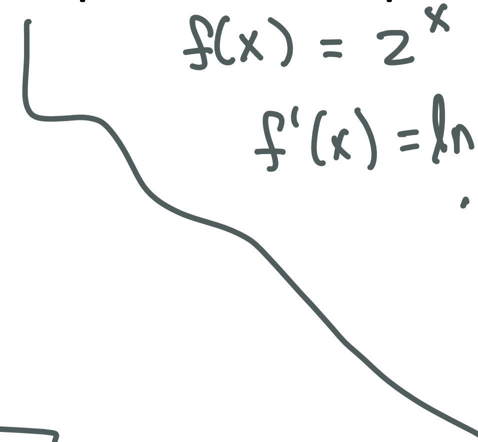
$$\phi = \rho^\sigma$$

$$\frac{\partial \phi}{\partial \rho} = \sigma \rho^{(\sigma - 1)} = 2\rho^{(2-1)} = 2\rho$$



$f(x) = 2^x$

$f'(x) = \ln(2) \cdot 2^x$

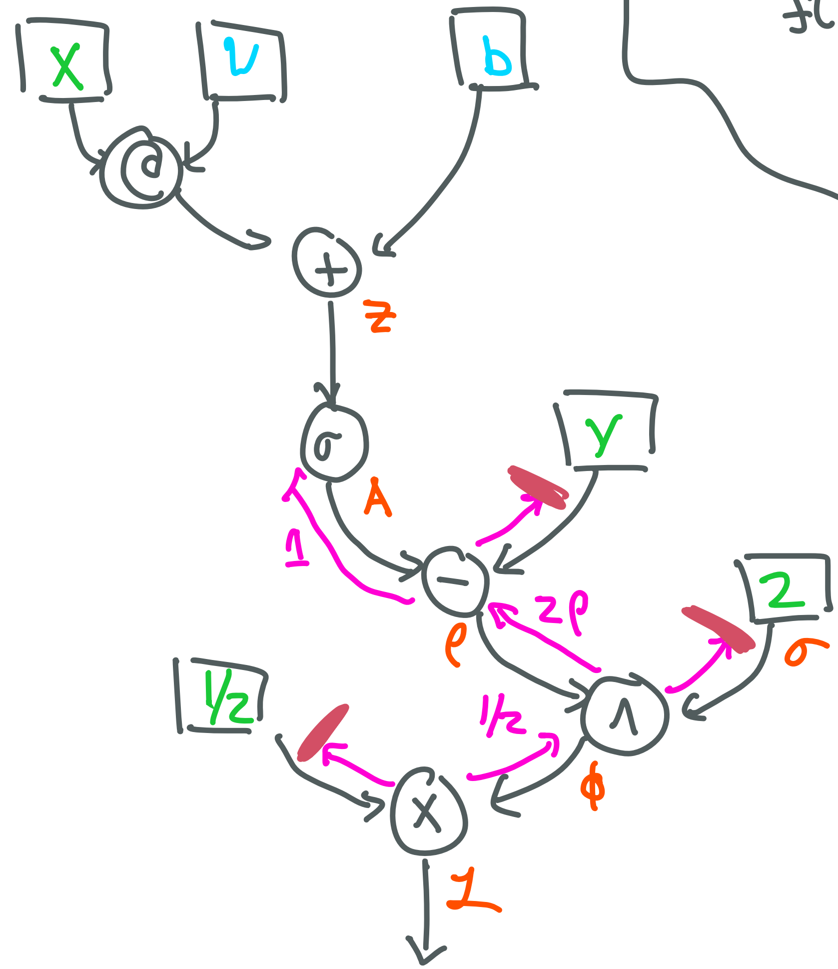


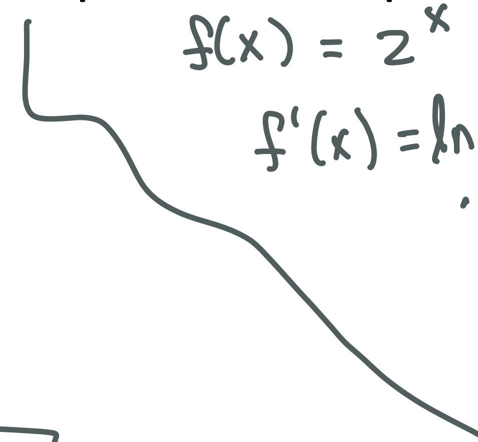
One-Layer Network (MSE and Sigmoid) Compute Graph

- $Z = XW^T + b$
- $A = \sigma(Z) = \sigma(XW^T + b)$
- $\mathcal{L} = \frac{1}{2}(A - Y)^2$

$\boxed{\mathcal{L}.\text{backward}()}$

$\mathcal{L} = \frac{1}{2}\left(\sigma(XW^T + b) - Y\right)^2$





$f(x) = 2^x$

$f'(x) = \ln(2$

$\cdot 2^x$

$\rho = A - Y$

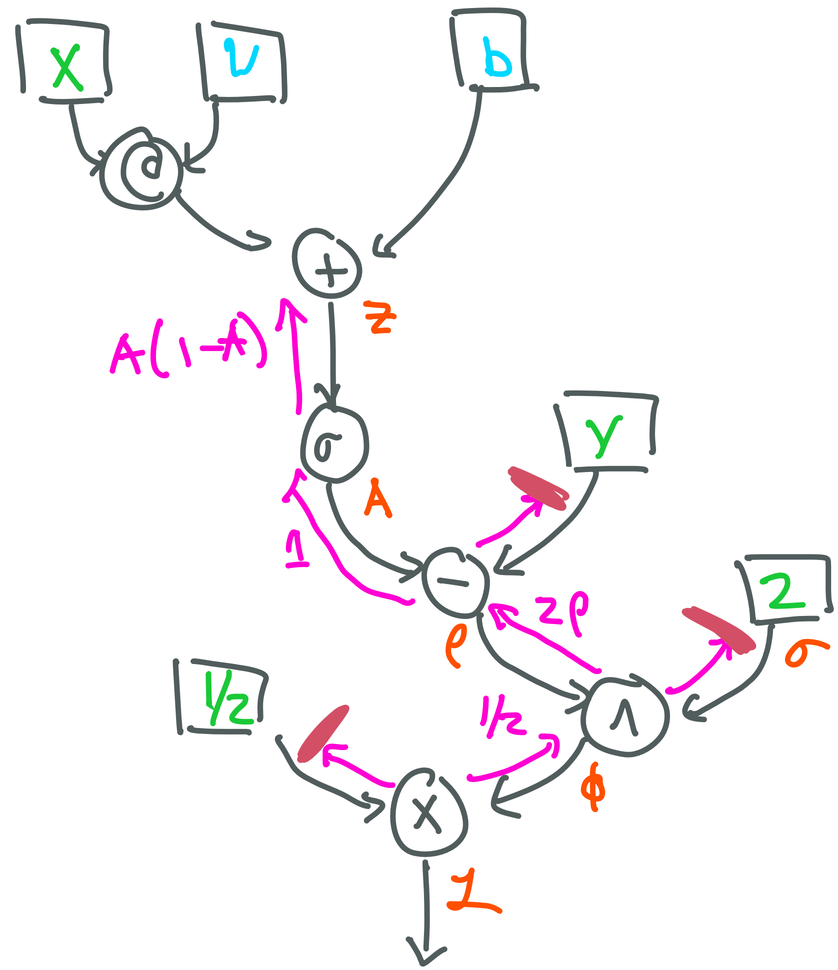
$\frac{\partial \rho}{\partial A} = 1, \quad \frac{\partial \rho}{\partial Y} = -1$

One-Layer Network (MSE and Sigmoid) Compute Graph

- $Z = XW^T + b$
- $A = \sigma(Z) = \sigma(XW^T + b)$
- $\mathcal{L} = \frac{1}{2}(A - Y)^2$

$\boxed{\mathcal{L}.\text{backward}()}$

$\mathcal{L} = \frac{1}{2}\left(\sigma(XW^T + b) - Y\right)^2$



$A = \sigma(z)$

$\frac{\partial A}{\partial z} = \sigma(z)(1 - \sigma(z))$
$= A(1 - A)$

One-Layer Network (MSE and Sigmoid) Compute Graph

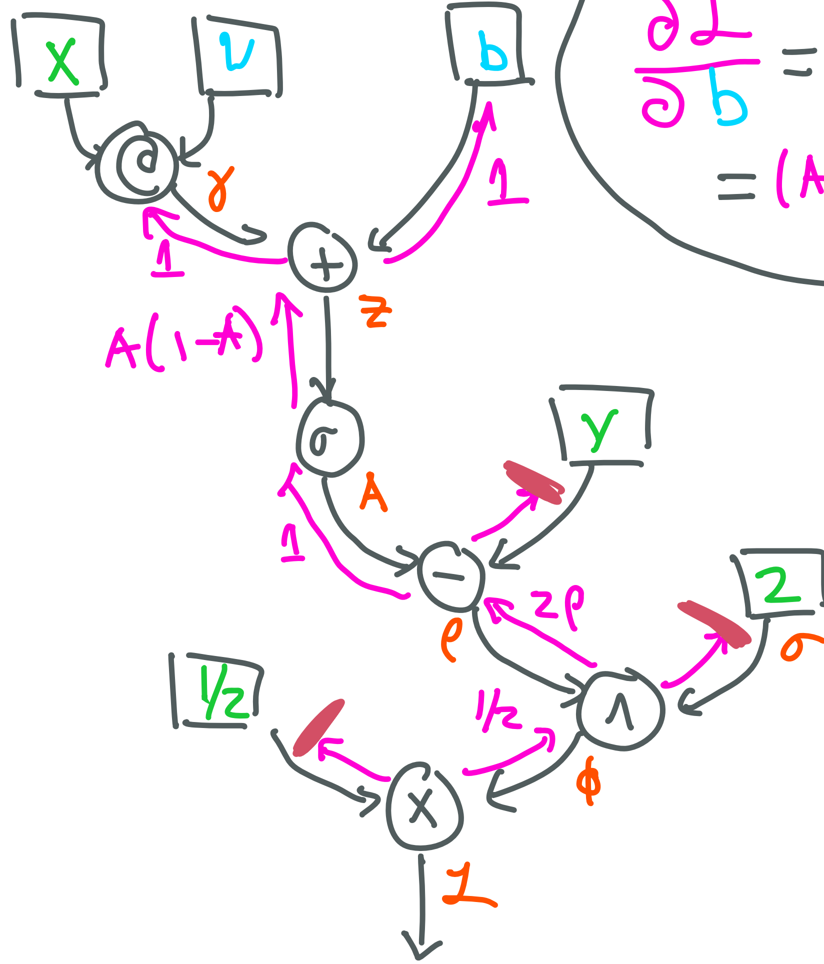
- $Z = \underbrace{XW^T}_{\gamma} + b$
- $A = \sigma(Z) = \sigma(XW^T + b)$
- $\mathcal{L} = \frac{1}{2}(A - Y)^2$

$\boxed{\mathcal{L}.\text{backward}()}$

$$\mathcal{L} = \frac{1}{2}\left(\sigma(XW^T + b) - Y\right)^2$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{2} 2 \rho A(1-A)$$

$$= (A-Y)A(1-A)$$



$$z = \gamma + b$$

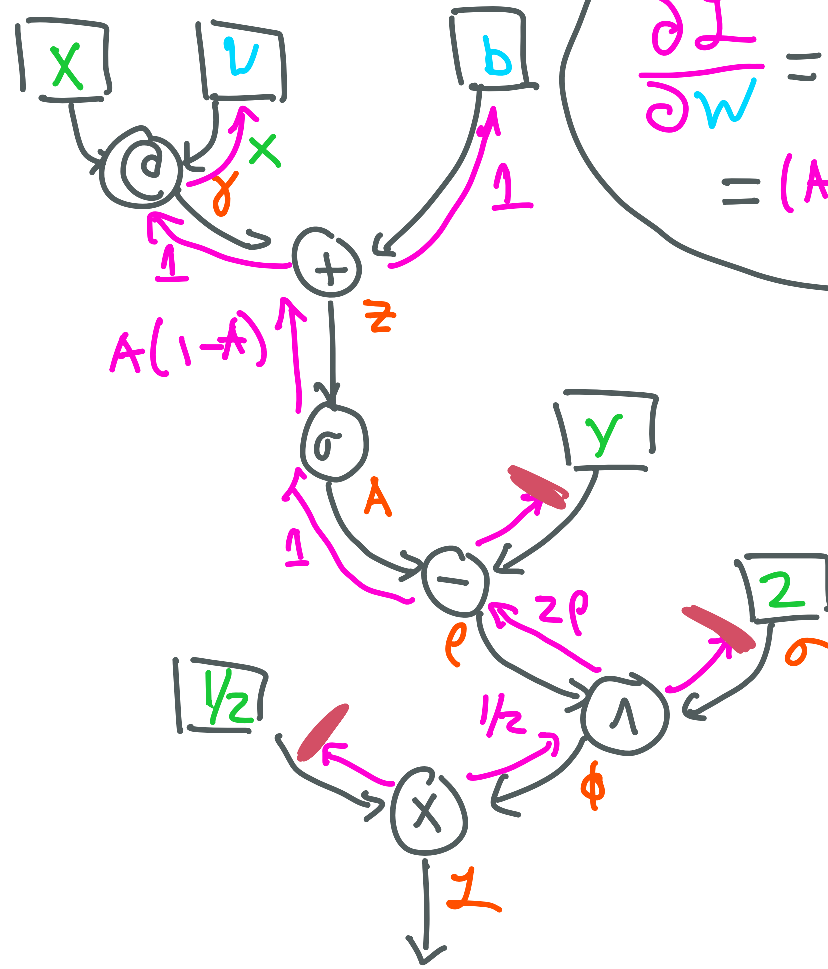
$$\frac{\partial z}{\partial \gamma} = 1, \quad \frac{\partial z}{\partial b} = 1$$

One-Layer Network (MSE and Sigmoid) Compute Graph

- $Z = XW^T + b$
- $A = \sigma(Z) = \sigma(XW^T + b)$
- $\mathcal{L} = \frac{1}{2}(A - Y)^2$

$\boxed{\mathcal{L}.\text{backward}()}$

$$\mathcal{L} = \frac{1}{2}\left(\sigma(XW^T + b) - Y\right)^2$$



$$\frac{\partial \mathcal{L}}{\partial W} = \frac{1}{2} 2 \rho A(1-A)X$$
$$= (A-Y)A(1-A)$$

$$\gamma = XW^T$$

$$\frac{\partial \gamma}{\partial X} = \cancel{\;\;}, \quad \frac{\partial \gamma}{\partial W} = X$$

Two-Layer Network (MSE and Sigmoid)

- $Z^{[1]} = A^{[0]}W^{[1]T} + b^{[1]T}$
- $A^{[1]} = \sigma\left(Z^{[1]}\right)$
- $Z^{[2]} = A^{[1]}W^{[2]T} + b^{[2]T}$
- $A^{[2]} = \sigma(Z^{[2]})$
- $\mathcal{L} = \frac{1}{2}\left(A^{[2]} - Y\right)^2$

Exercise for the reader

Manual Differentiation

- How does the derivation change if we add hidden layers?

More derivation

- How does the derivation change if we change the activation function?

$$\frac{\partial A}{\partial z} = \sigma(z)(1 - \sigma(z))$$

- How does the derivation change if we change the loss function?

$$\mathcal{L} = \frac{1}{2}(A - Y)^2 \qquad \mathcal{L} = -[[\, Y \log(A) + (1-Y)\log(1-A)]]$$

Automatic Differentiation (Autodiff, Autograd)

```
# Forward propagation A0 = X, Yhat = A2
Z1 = linear(A0, W1, b1)
A1 = sigmoid(Z1)
Z2 = linear(A1, W2, b2)
A2 = sigmoid(Z2)

# Compute loss as the mean-square-error
bce_loss = torch.mean(Y * torch.log(A2) + (1 - Y) * torch.log(1 - Yhat))

learning_rate = 0.01 # aka alpha or α
```

Automatic Differentiation (Autodiff, Autograd)

```
# Forward propagation A0 = X, Yhat = A2
Z1 = linear(A0, W1, b1)
A1 = sigmoid(Z1)
Z2 = linear(A1, W2, b2)
A2 = sigmoid(Z2)

# Compute loss as the mean-square-error
bce_loss = torch.mean(Y * torch.log(A2) + (1 - Y) * torch.log(1 - Yhat))

learning_rate = 0.01 # aka alpha or α
```

```
# Compute gradients for W^[2] and b^[2]
dL_dY = (Y / Yhat - (1 - Y) / (1 - Yhat)) / 2
dY_dZ2 = Yhat * (1 - Yhat)
dZ2 = dL_dY * dY_dZ2
dW2 = (1 / N) * dZ2.T @ A1
db2 = dZ2.mean(dim=0)

# Compute gradients for W^[1] and b^[1]
dZ1 = dZ2 @ W2 * ((A1 * (1 - A1)))
dW1 = (1 / N) * dZ1.T @ X
db1 = dZ1.mean(dim=0)

W1 -= learning_rate * dW1
b1 -= learning_rate * db2
W2 -= learning_rate * dW2
b2 -= learning_rate * db2
```

Automatic Differentiation (Autodiff, Autograd)

```
# Forward propagation A0 = X, Yhat = A2
Z1 = linear(A0, W1, b1)
A1 = sigmoid(Z1)
Z2 = linear(A1, W2, b2)
A2 = sigmoid(Z2)

# Compute loss as the mean-square-error
bce_loss = torch.mean(Y * torch.log(A2) + (1 - Y) * torch.log(1 - Yhat))

learning_rate = 0.01 # aka alpha or α
```

```
# Compute gradients for W^[2] and b^[2]
dL_dY = (Y / Yhat - (1 - Y) / (1 - Yhat)) / 2
dY_dZ2 = Yhat * (1 - Yhat)
dZ2 = dL_dY * dY_dZ2
dW2 = (1 / N) * dZ2.T @ A1
db2 = dZ2.mean(dim=0)

# Compute gradients for W^[1] and b^[1]
dZ1 = dZ2 @ W2 * ((A1 * (1 - A1)))
dW1 = (1 / N) * dZ1.T @ X
db1 = dZ1.mean(dim=0)

W1 -= learning_rate * dW1
b1 -= learning_rate * db2
W2 -= learning_rate * dW2
b2 -= learning_rate * db2
```

```
bce_loss.backward()

W1 -= learning_rate * W1.grad
b1 -= learning_rate * b1.grad
W2 -= learning_rate * W2.grad
b2 -= learning_rate * b2.grad
```

Creating the Compute Graph

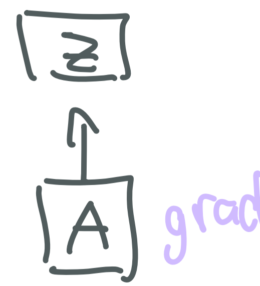
```python
# Sigmoid compute node from matrix.py
def sigmoid(self) -> Matrix:
    """Element-wise sigmoid."""
    result = Matrix(self.data.sigmoid(), children=(self,))

    def _gradient() -> None:
        self.grad += result.data * (1 - result.data) * result.grad

    result._gradient = _gradient
    return result
```

matrix.py

$\sigma(z)$ $(1 - \sigma(z))$ Back propagated

$Z = \ldots$

$A = Z.\text{sigmoid}($




```python
# Sigmoid math from list2d.py
def sigmoid(self) -> List2D:
    vals = [
        [sigmoid(self.vals[i][j]) for j in range(self.ncol)]
        for i in range(self.nrow)
    ]
    return List2D(*self.shape, vals)
```

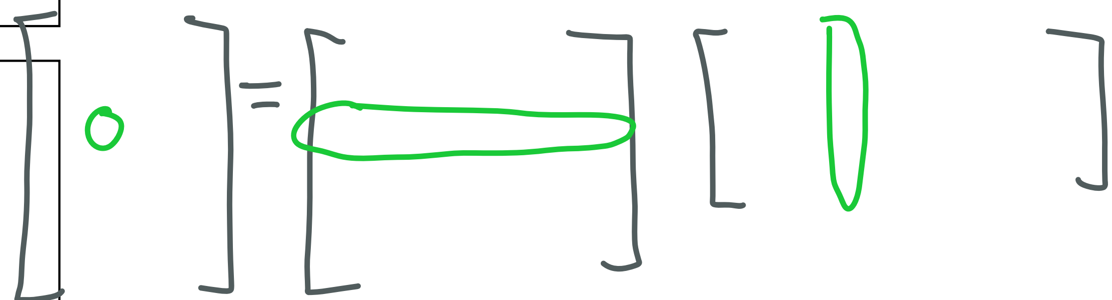
list2d.py

Creating the Compute Graph

```python
def sigmoid(self) -> Matrix:
    """Element-wise sigmoid."""
    result = Matrix(self.data.sigmoid(), children=(self,))

    def _gradient() -> None:
        self.grad += result.data * (1 - result.data) * result.grad

    result._gradient = _gradient
    return result
```

$A(1-A)$

$T = X v^T$



```python
def __matmul__(self, rhs: Matrix) -> Matrix:
    """Matrix multiplication: self @ rhs."""
    result = Matrix(self.data @ rhs.data, children=(self, rhs))

    def _gradient() -> None:
        self.grad += result.grad @ rhs.data.T
        rhs.grad += self.data.T @ result.grad

    result._gradient = _gradient
    return result
```

```python
def __add__(self, rhs: float | int | Matrix) -> Matrix:
    """Element-wise addition."""
    rhs_vals = rhs.data if isinstance(rhs, Matrix) else rhs
    children = (self, rhs) if isinstance(rhs, Matrix) else (self,)
    result = Matrix(self.data + rhs_vals, children=children)

    def _gradient() -> None:
        self.grad += result.grad.unbroadcast(*self.shape)
        if isinstance(rhs, Matrix):
            rhs.grad += result.grad.unbroadcast(*rhs.shape)

    result._gradient = _gradient
    return result
```

$*1$

$*1$

Backpropagation (.backward)

```
def backward(self) -> None:
    """Compute all gradients using backpropagation."""

    sorted_nodes: list[Matrix] = []
    visited: set[Matrix] = set()

    # Sort all elements in the compute graph using a topological ordering (DFS)
    # (Creating a closure here for convenience; capturing sorted_nodes and visited)
    def topological_sort(node: Matrix) -> None:
        if node not in visited:
            visited.add(node)
            for child in node._children:
                topological_sort(child)
            sorted_nodes.append(node)

    # Perform the topological sort
    topological_sort(self)

    # Initialize all gradients with ones
    self.grad.ones_()

    # Update gradients from output to input (backwards)
    for node in reversed(sorted_nodes):
        node._gradient()
```

$\mathcal{L} = \frac{1}{2}(A - Y)^2$

$\mathcal{L}$.back()