Midterm and automatic differentiation (aka autograd)

Lecture 06 — CS 577 Deep Learning

Instructor: Yutong Wang

Computer Science
Illinois Institute of Technology

September 25, 2024

Topics

- The midterm
- Attention models
- Tricks for doing homework 3.
  - `np.matmul` with multidim arrays
  - Batched matrix multiplication
  - Batched diagonalization
  - Batched outer product
  - `np.moveaxis`

midterm ↑

exam ↓

- Automatic differentiation

More on homework 3: Next-word-prediction

φ word embedding $\varphi(\text{over})$ $\varphi(\text{the}) \in \mathbb{R}^d$

The quick brown fox jumps over the ____.

$$\mathbf{X}^{(i)} = \begin{bmatrix} \mathbf{x}^{(i,1)} & \mathbf{x}^{(i,2)} & \cdots & \mathbf{x}^{(i,C-1)} & \mathbf{x}^{(i,C)} \end{bmatrix} \in \mathbb{R}^{d \times C} \quad \text{is a } d \times C \text{ matrix}$$

a single data point (also called a prompt)

the

$$y^{(i)} \in \underbrace{\{1, \ldots, K\}}_{\text{vocabulary}} \quad \text{is set of all candidate words}$$

More on homework 3: Next-word-prediction

The quick brown fox jumps over the _____.

empty

the

$$\mathbf{X}^{(i)} = \left[\mathbf{x}^{(i,1)} \quad \mathbf{x}^{(i,2)} \quad \cdots \quad \mathbf{x}^{(i,C-1)} \quad \mathbf{x}^{(i,C)}\right] \in \mathbb{R}^{d\times C} \quad \text{is a } d \times C \text{ matrix}$$

should have the most important

$$y^{(i)} \in \{1, \ldots, K\} \quad \text{is set of all candidate words}$$

More on homework 3: Next-word-prediction

The quick brown fox jumps over the _____.

$$\mathbf{X}^{(i)} = \begin{bmatrix} \mathbf{x}^{(i,1)} & \mathbf{x}^{(i,2)} & \cdots & \mathbf{x}^{(i,C-1)} & \mathbf{x}^{(i,C)} \end{bmatrix} \in \mathbb{R}^{d \times C} \quad \text{is a } d \times C \text{ matrix}$$

- unrealistic
- good for learning + writing papers

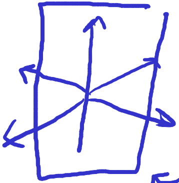
$y^{(i)} \in \{\pm 1\}$ we are keeping it simple

Notations

- C as `n_context` dim of word embed 700-ish
- d as `n_features`
- n as `n_samples` training
- q as `n_reduced`, where $q < d$ projection onto small space of dim q

Let

$$\theta = [W^{(1)}, W^{(2)}, w^{(3)}]$$



$\mathbb{R}^3$ $\downarrow$ $\mathbb{R}^2$

where

$$W^{(1)} \text{ and } W^{(2)} \in \mathbb{R}^{q \times d}$$

and

$$w^{(3)} \in \mathbb{R}^d$$

and

$$f(X^{(i)}; \theta) = w^{(3)\top} X^{(i)} \underbrace{\text{softmax}\left(X^{(i)\top} W^{(2)\top} W^{(1)} x^{(i,C)}\right)}_{\text{attention}}.$$

subspace of some concept.

("Single-layer" and "single-headed") Attention

$$f(X^{(i)};\theta) = w^{(3)\top} X^{(i)} \operatorname{softmax}\left(X^{(i)\top} W^{(2)\top} W^{(1)} x^{(i,C)}\right).$$

$[x^{(i,1)} \cdots\cdots x^{(i,C)}]$

$\propto \exp(x^{(i,j)\top} W^{(2)\top} W^{(1)} x^{(i,C)})$

propto

Similarity between j-th word emb & C-th word emb

j

$\mathbb{R}^C \ni$

prob. vector

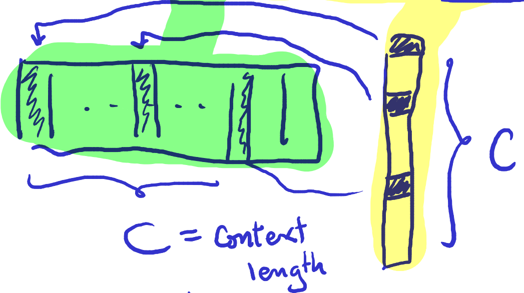
Σ to 1 and positive

① Note softmax playing totally different role compared to CE.

(“Single-layer” and “single-headed”) Attention

$$f(X^{(i)};\theta) = w^{(3)\top} X^{(i)} \operatorname{softmax}\left(X^{(i)\top} W^{(2)\top} W^{(1)} x^{(i,C)}\right).$$

Learning to pick positions in the prompt



■ = big

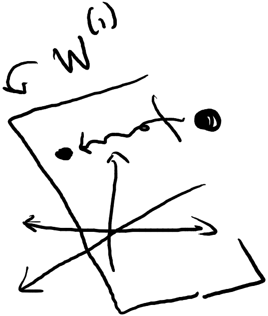
□ = small

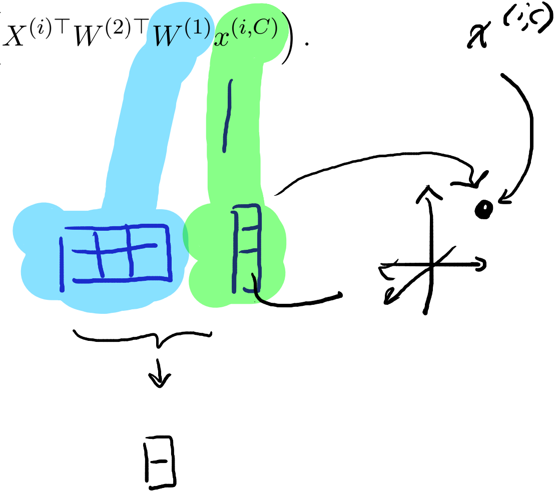
$w^{(3)T}$ ← [ ] weighted avg

picking out columns to pay attention

(“Single-layer” and “single-headed”) Attention

$$f(X^{(i)};\theta) = \underbrace{w^{(3)\top} X^{(i)}}_{\text{full context}} \operatorname{softmax}\left(X^{(i)\top} W^{(2)\top} W^{(1)} x^{(i,C)}\right).$$



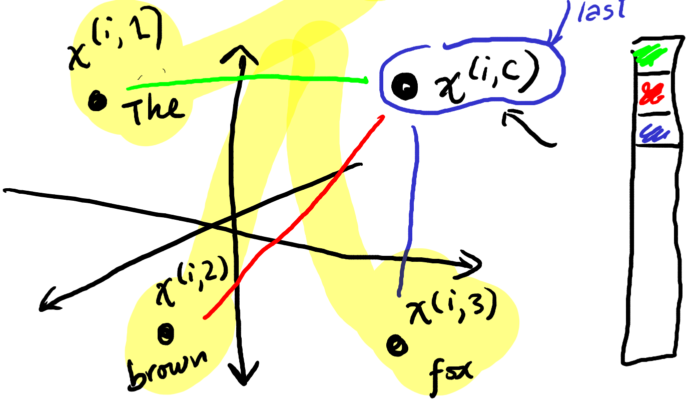


(“Single-layer” and “single-headed”) Attention

$$f(X^{(i)};\theta) = w^{(3)\top} X^{(i)}\,\mathrm{softmax}\left(X^{(i)\top} W^{(2)\top} \boxed{W^{(1)} x^{(i,C)}}\right).$$

last

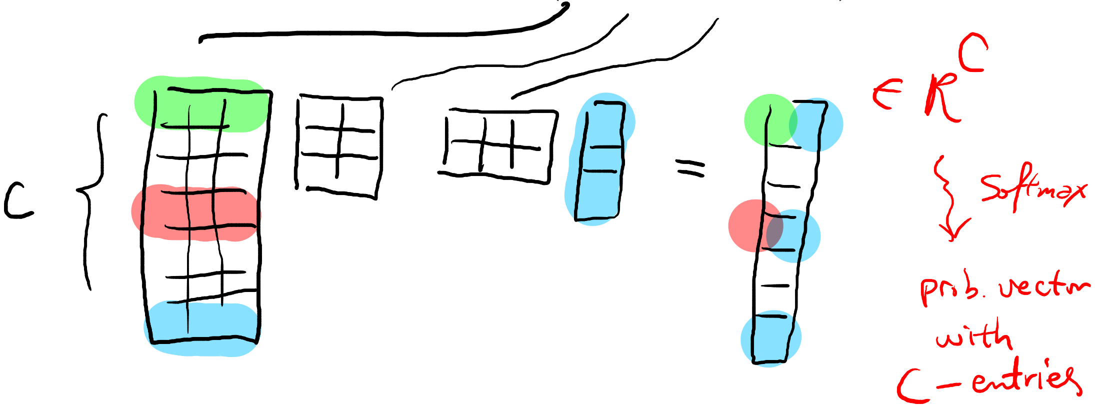
against rest



("Single-layer" and "single-headed") Attention

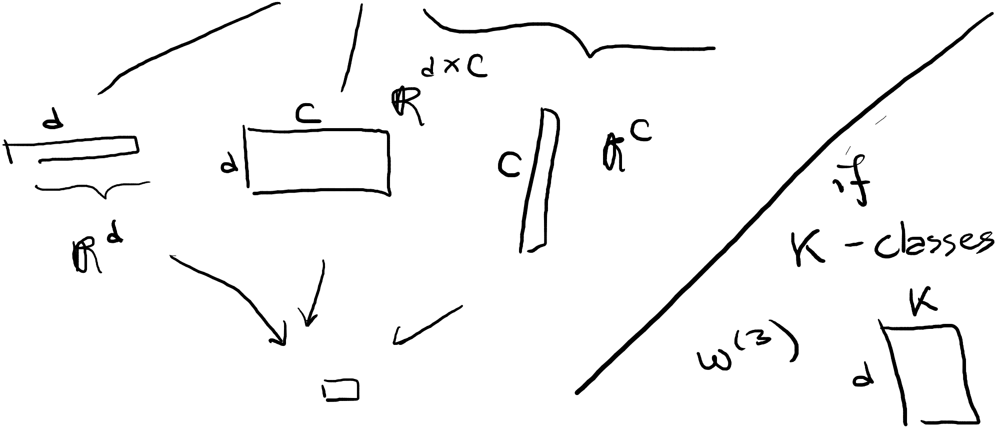
just a number

$$f(X^{(i)};\theta) = w^{(3)\top} X^{(i)} \,\mathrm{softmax}\left(X^{(i)\top} W^{(2)\top} W^{(1)} x^{(i,C)}\right).$$



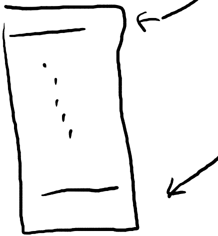
("Single-layer" and "single-headed") Attention

$$f(X^{(i)};\theta) = w^{(3)\top} X^{(i)}\,\mathrm{softmax}\left(X^{(i)\top} W^{(2)\top} W^{(1)} x^{(i,C)}\right).$$



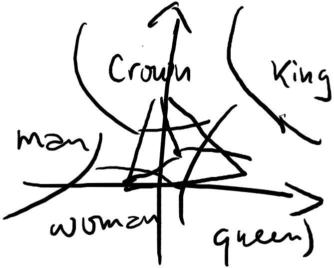
(“Single-layer” and “single-headed”) Attention

$$f(X^{(i)};\theta) = w^{(3)\top} \underbrace{X^{(i)}\,\mathrm{softmax}\left(\underbrace{X^{(i)\top}}_{\text{Keys}}\underbrace{W^{(2)\top}}_{\text{key matrix}}\underbrace{W^{(1)}}_{\text{query matrix}}\underbrace{x^{(i,C)}}_{\text{query}}\right)}_{\text{weighted comb of the Keys}}.$$



("Single-layer" and "single-headed") Attention

$$f(X^{(i)};\theta) = w^{(3)\top} X^{(i)} \,\mathrm{softmax}\left(X^{(i)\top} W^{(2)\top} W^{(1)} x^{(i,C)}\right).$$



key space

("Single-layer" and "single-headed") Attention

$$f(X^{(i)};\theta) = \underline{w}^{(3)\top} X^{(i)} \operatorname{softmax}\left(X^{(i)\top} W^{(2)\top} W^{(1)} x^{(i,C)} \right).$$

" $W^{(1)}$ " 1 symbol

U

Reason for "(1), (2)" is to reuse

W for weights

("Single-layer" and "single-headed") Attention

$$f(X^{(i)}; \theta) = w^{(3)\top} X^{(i)} \operatorname{softmax}\left(X^{(i)\top} W^{(2)\top} W^{(1)} x^{(i,C)} \right).$$ ← typo

bold $\mathbf{W}$ capital stands multi-dim arr
including matrix & tensor

bold $\mathbf{w}$ lower case = vector

non-bold w l.c. = scalar

In this class

Goodfellow convention

("Single-layer" and "single-headed") Attention

$$f(X^{(i)};\theta) = w^{(3)\top} X^{(i)}\, \text{softmax}\left(X^{(i)\top} W^{(2)\top} W^{(1)} x^{(i,C)}\right).$$

sign

C - dot products

$\{\pm 1\}$ $+1$ = foo

-1 = bar

$\{1, \ldots, K\}$ argmax

1 = the

2 = an

3 = and

("Single-layer" and "single-headed") Attention

$$f(X^{(i)};\theta) = w^{(3)\top} X^{(i)} \,\mathrm{softmax}\left(X^{(i)\top} W^{(2)\top} W^{(1)} x^{(i,C)}\right).$$

NWP NTP

token

GPT-models

BERT

Masked-seq training

Tricks for doing homework 3.

- `np.matmul` with multidim arrays
- Batched matrix multiplication
- Batched diagonalization
- Batched outer product
- `np.moveaxis`

np.matmul with multidim arrays

X

rank 3 tensor

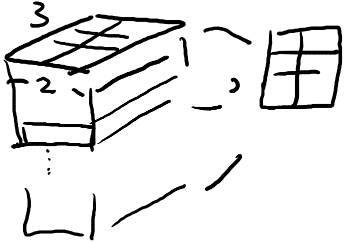
$W^{(2)}$

rank 2 tensor

aka matrix

```
# IN (n_samples, n_context, n_features) @ (n_features, n_reduced)
Keys = np.matmul(X, W2)
# OUT  (n_samples, n_context, n_reduced)
```

np.matmul (



)

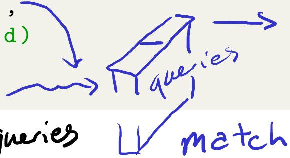
a stack of matmul

Batched ~~matrix~~ matrix-vector multiplication

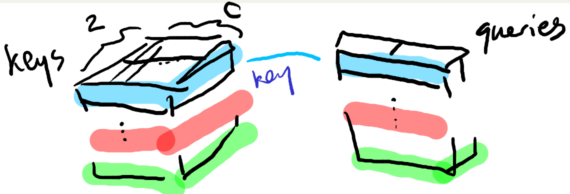
Keys

queries.shape

```
queries.shape == (n_samples, n_reduced)

# (n_samples, n_context, n_reduced)
np.sum(
    # (n_samples, n_context, n_reduced) * (n_samples, 1, n_reduced)
    Keys * np.expand_dims(queries,axis=1),
    # --> (n_samples, n_context, n_reduced)
    axis=2)
# --> (n_samples, n_context)
```

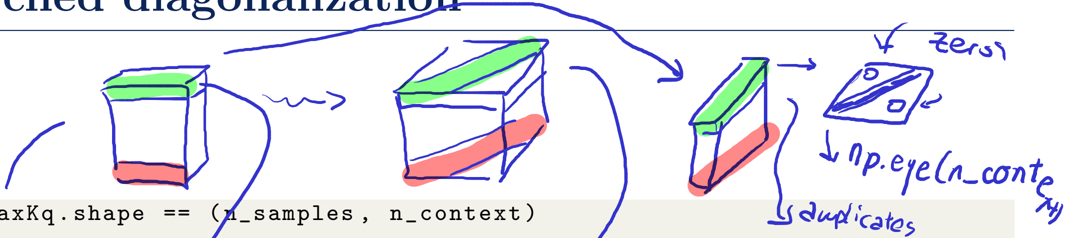


sum(axis=2)



match the 0th dim and do stackwise matmul

Batched diagonalization




```
softmaxKq.shape == (n_samples, n_context)

# (n_samples, 1, n_context) * (n_context, n_context)
softmaxKq_diag = # YOUR CODE HERE
# --> (n_context, n_context, n_context)
```

$$\begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_C \end{bmatrix} \qquad \begin{bmatrix} P_1 & & 0 \\ & P_2 & \\ 0 & & \ddots \quad P_C \end{bmatrix}$$

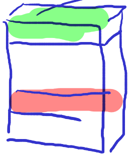
Batched diagonalization

_ * _

_.shape _.shape

```
softmaxKq.shape == (n_samples, n_context)

# (n_samples, 1, n_context) * (n_context, n_context)
softmaxKq_diag = # YOUR CODE HERE
# --> (n_context, n_context, n_context)
```

Batched outer product



softmaxKq

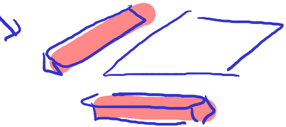
(n samples, ncontext)

j-th entry

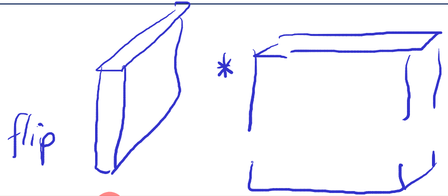
$p_i p_j$


```
# (n_samples, 1, n_context) * (n_samples, n_context, 1)
softmaxKq_outer = # YOUR CODE HERE
# --> (n_samples, n_context, n_context)
```

i-th entry



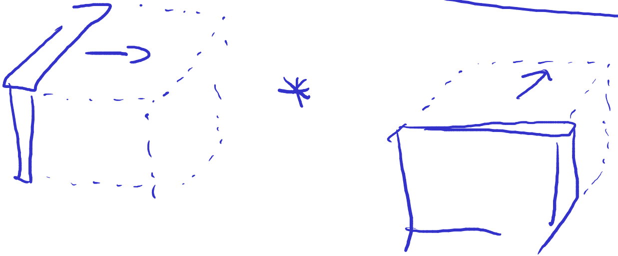
Batched outer product



expand_dims(axis = 1)

```
# (n_samples, 1, n_context) * (n_samples, n_context, 1)
softmaxKq_outer = # YOUR CODE HERE
# --> (n_samples, n_context, n_context)
```

n_context



n_context

np.moveaxis

$X^{(i)}$ stacked

not touching D

```
X.shape == (n_samples, n_context, n_features)

# (n_samples, n_features, n_context) @ (n_samples, n_context, n_context)
A = np.moveaxis(X,1,2) @ D
# --> (n_samples, n_features, n_context)
```

want batchwise matmul

transpose each stack

Welcome to the second half the course

- Understand how automatic differentiation works under the hood
- What is overfitting, when it happens, and what can you do about it
- Transfer learning
- Convolution layers, attention
- How to compute things in deep learning really really fast.

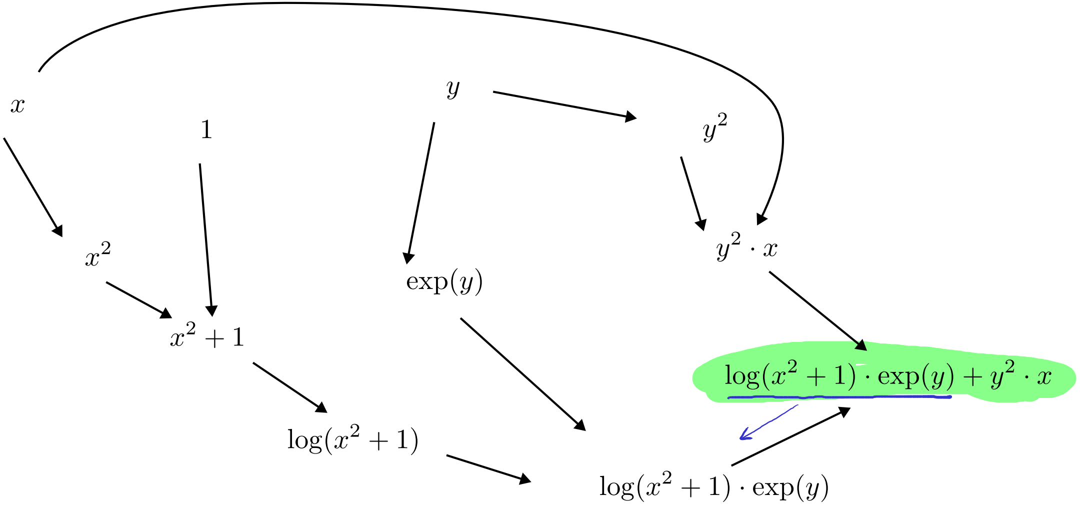
Automatic differentiation

Let's compute

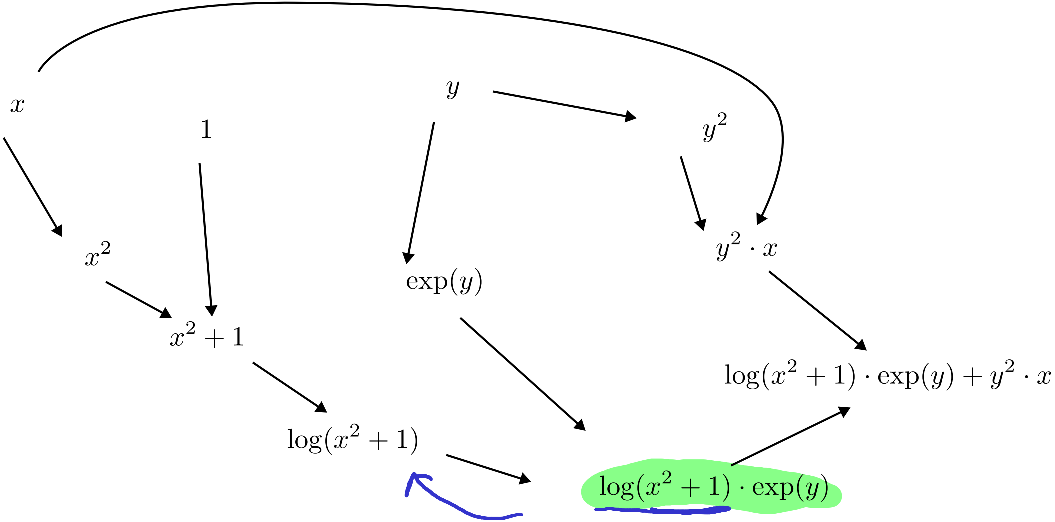
$$f(x, y) = \log(x^2 + 1)\exp(y) + y^2 x \qquad x, y \in \mathbb{R}$$

using automatic differentiation.

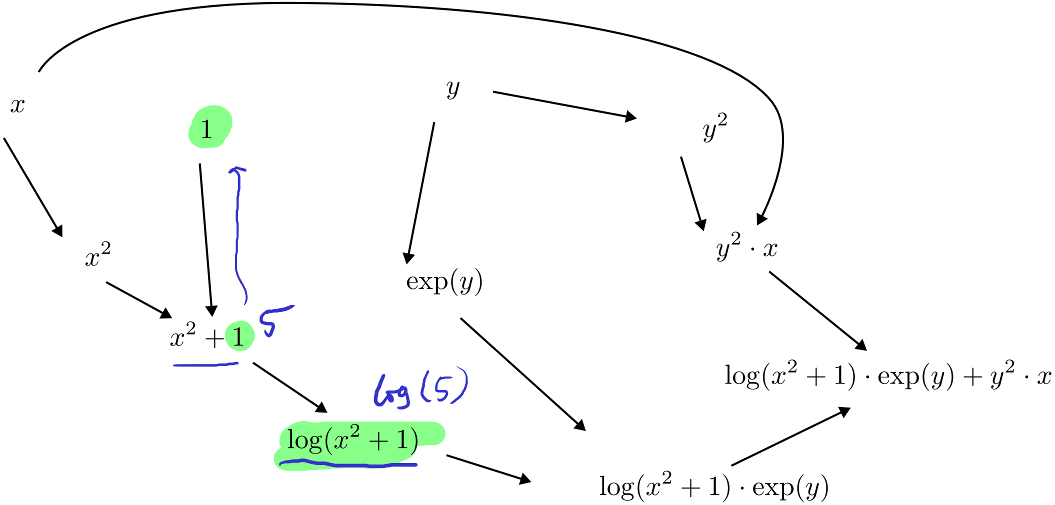
Eval $f(x, y) = \log(x^2+1)\exp(y) + y^2 x$ at $(x, y) = (2, 3)$



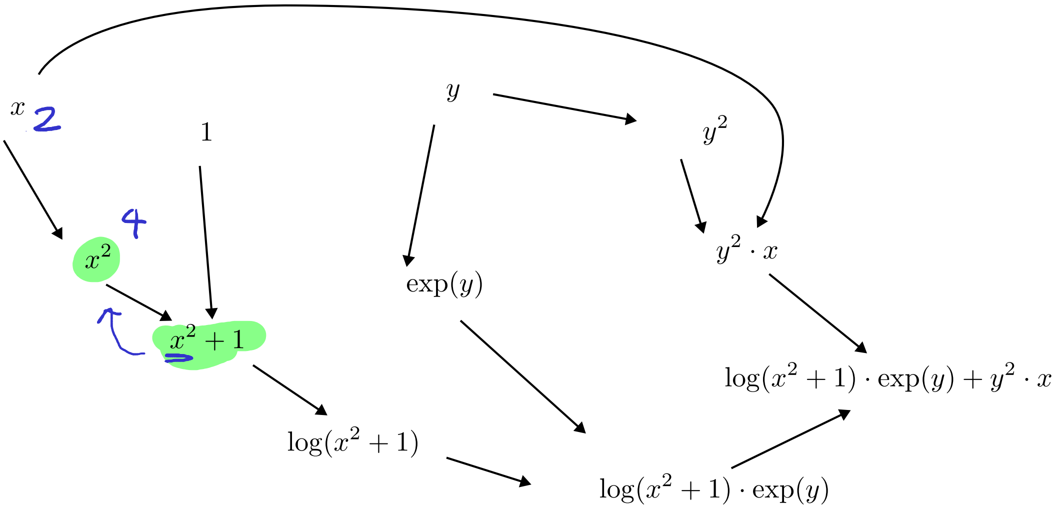
Eval $f(x, y) = \log(x^2 + 1)\exp(y) + y^2 x$ **at** $(x, y) = (2, 3)$



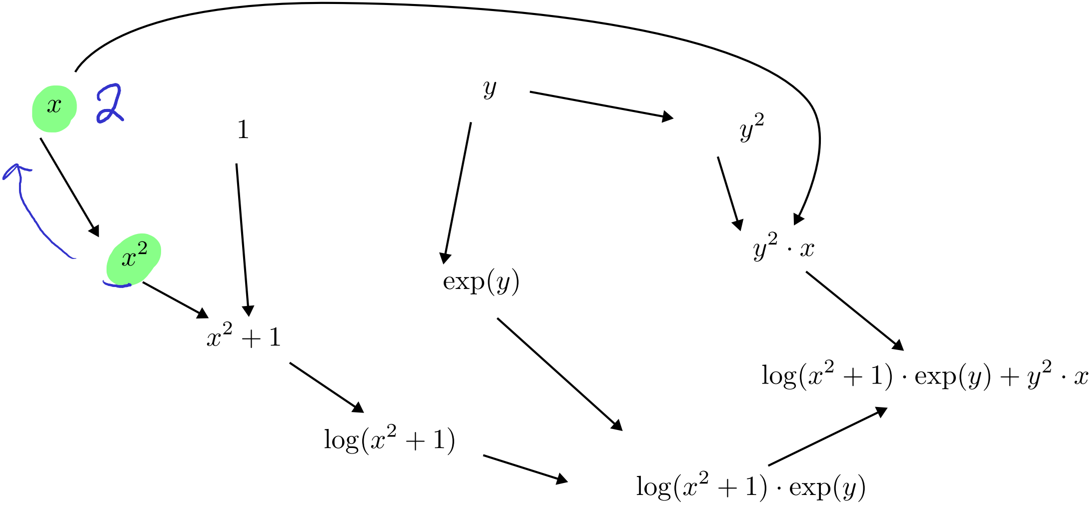
Eval $f(x, y) = \log(x^2 + 1)\exp(y) + y^2 x$ at $(x, y) = (2, 3)$



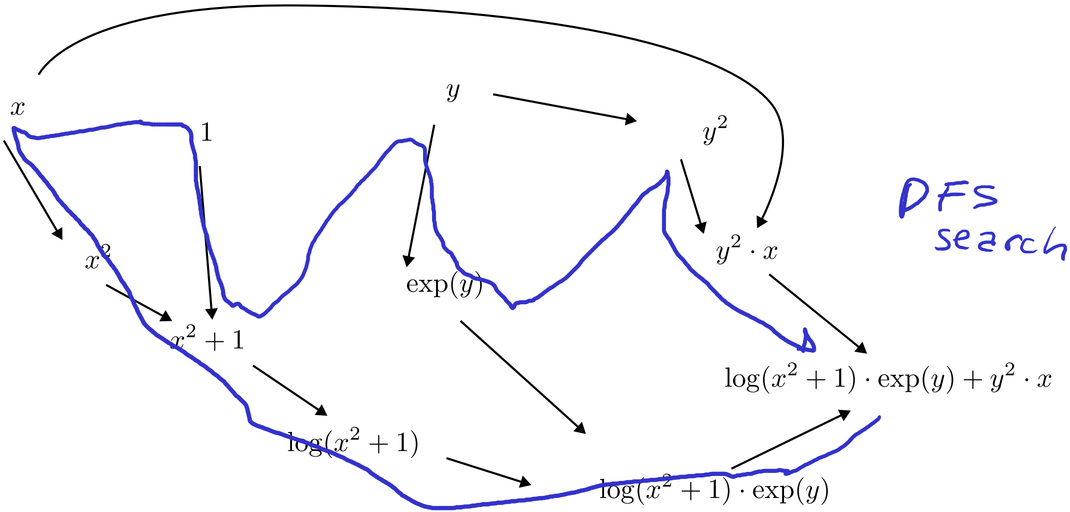
Eval $f(x,y) = \log(x^2+1)\exp(y) + y^2x$ **at** $(x,y) = (2,3)$



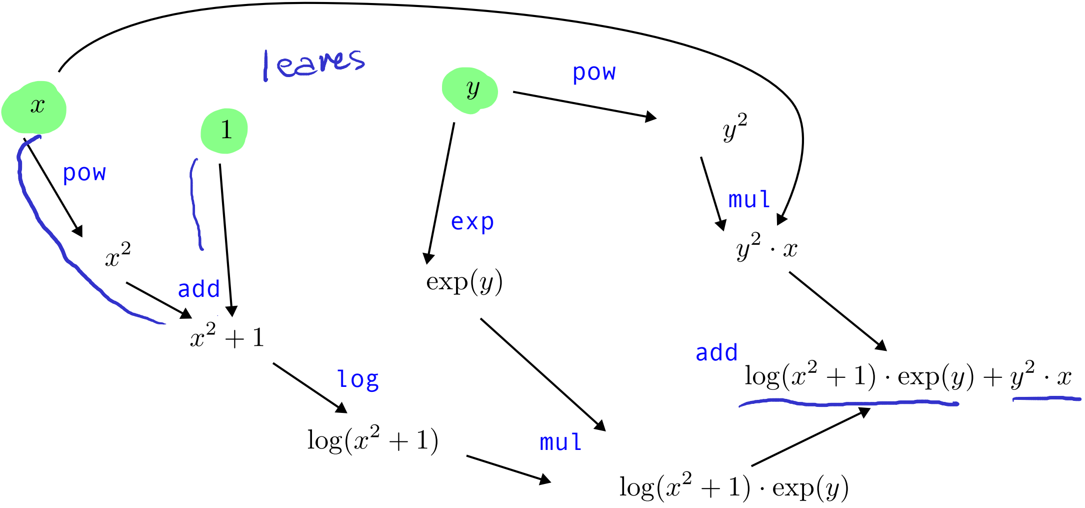
Eval $f(x,y) = \log(x^2+1)\exp(y) + y^2x$ **at** $(x,y) = (2,3)$



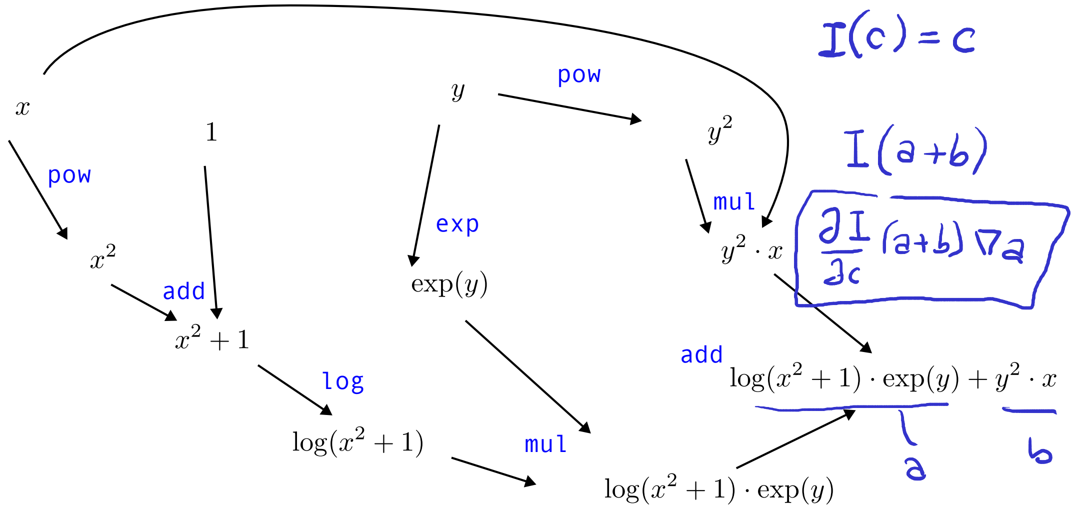
Eval $f(x, y) = \log(x^2 + 1)\exp(y) + y^2 x$ at $(x, y) = (2, 3)$



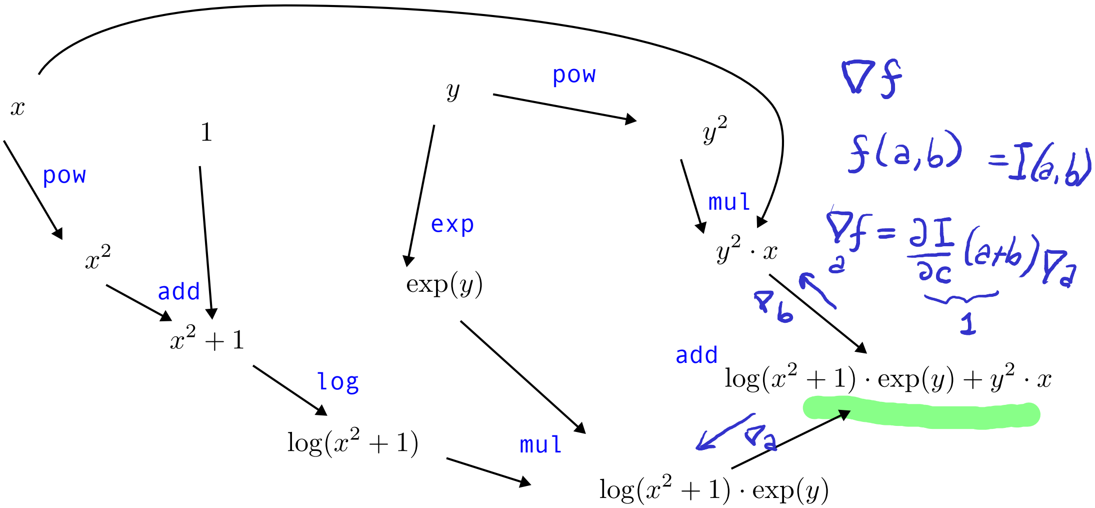
Eval $\nabla f(x, y) = \log(x^2 + 1)\exp(y) + y^2 x$ at $(2, 3)$



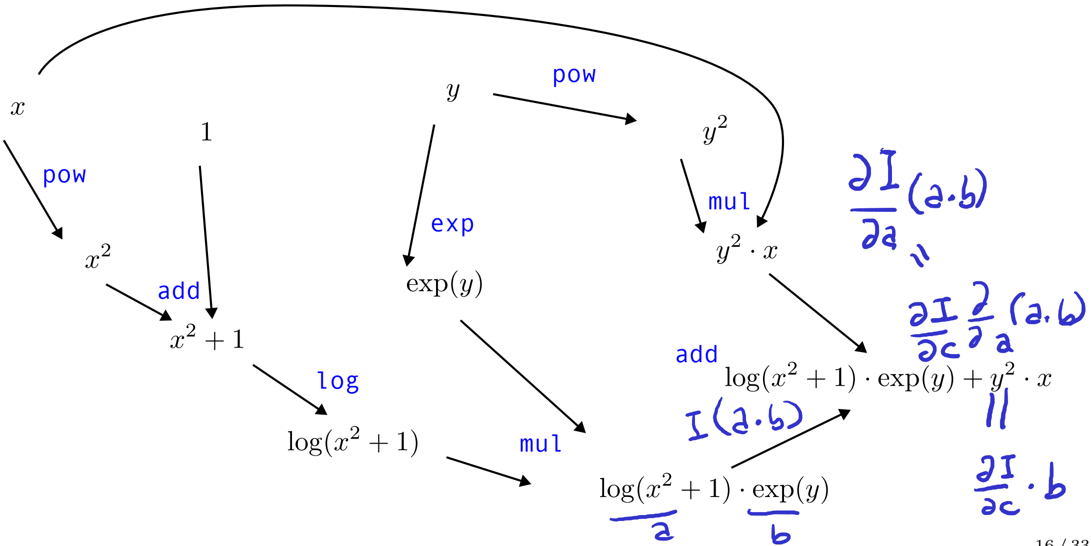
Eval $\nabla f(x, y) = \log(x^2 + 1)\exp(y) + y^2 x$ at $(2, 3)$



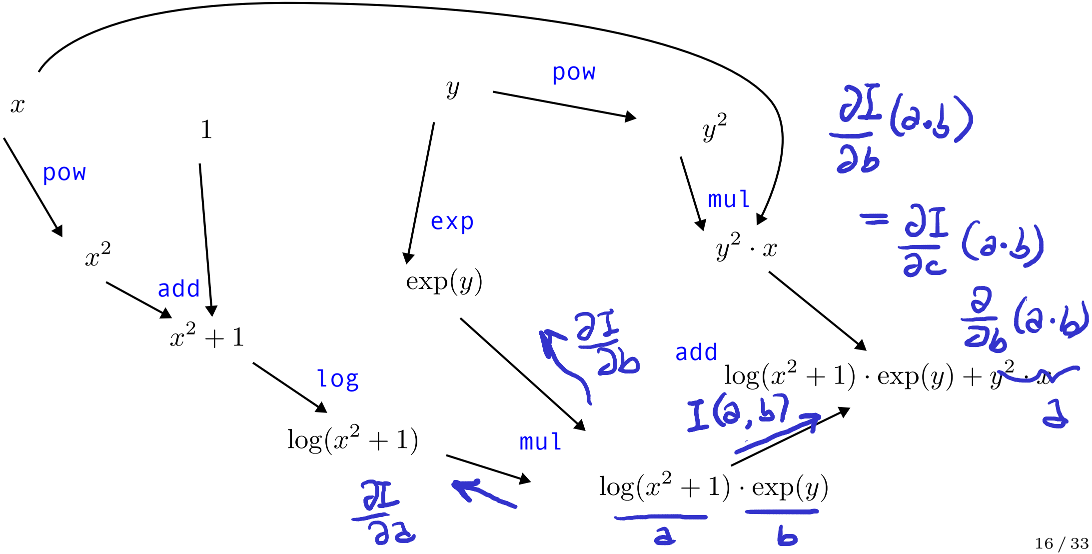
Eval $\nabla f(x,y) = \log(x^2+1)\exp(y) + y^2 x$ at $(2,3)$



Eval $\nabla f(x, y) = \log(x^2 + 1)\exp(y) + y^2 x$ at $(2, 3)$



Eval $\nabla f(x, y) = \log(x^2 + 1)\exp(y) + y^2 x$ at $(2, 3)$



An autograd “module”

```
class ag: # AutoGrad
    class Scalar: # Scalars with grads
        def __init__(self, value, op="", _backward= lambda : None, inputs=[], label=""):

            self.value = float(value)
            self.grad = 0.0

            # ... lines skipped
```

An autograd “module”

```
class ag: # AutoGrad
    class Scalar: # Scalars with grads
        def __init__(self, value, op="", _backward= lambda : None, inputs=[], label=""):

            self.value = float(value)
            self.grad = 0.0

            # ... lines skipped

        def __add__(self, other): # ...        +
        def __mul__(self, other): # ...        *
        def __pow__(self, exponent): # ...     **

    def exp(input): # ...
    def log(input): # ...
```

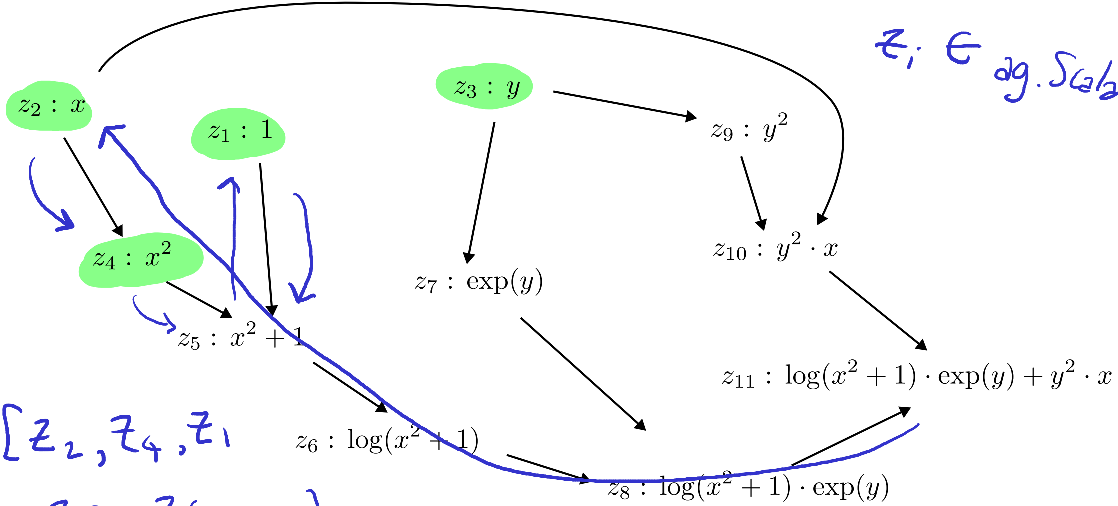
Implement f in our framework

```
x = ag.Scalar(2, label="z2:x")   # wrap x
y = ag.Scalar(3, label="z3:y")   # wrap y

# implement log(x^2+1)*exp(y) + y^2*x
def f(x,y):
    z1 = ag.Scalar(1,label= "z1:1")   # constant 1
    z2 = x
    z3 = y

    z4 = z2**2
    z4.label = "z4:x^2"

    # ... lines skipped
```

along left first

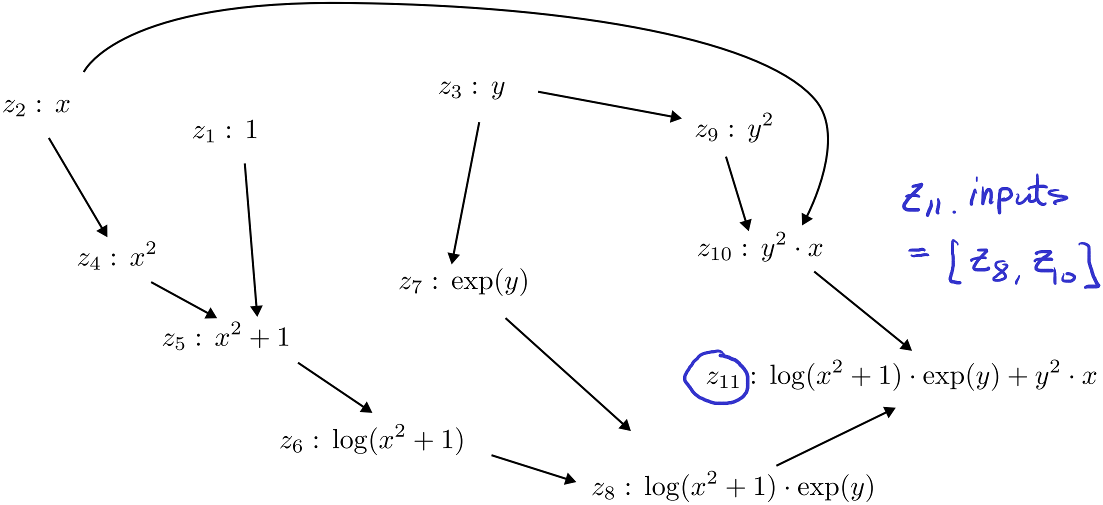
Forward pass — DFS — computation graph

$z_i \in$ eg. Scalar



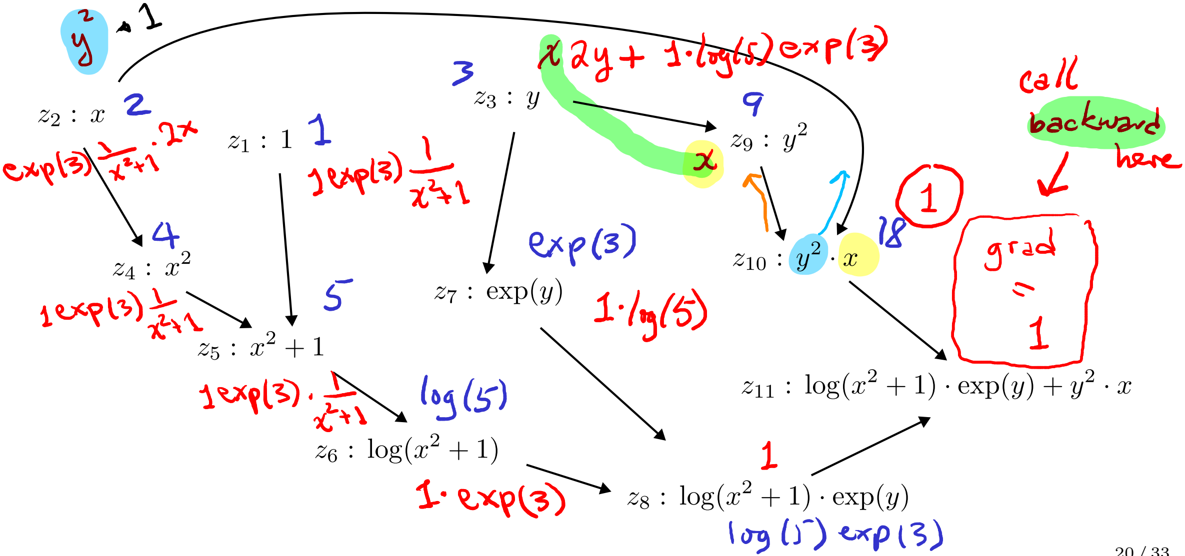
[z_2, z_4, z_1, z_5, z_6, ...]

Forward pass — DFS — computation graph



z_{11}. inputs $= [z_8, z_{10}]$

Forward pass — DFS — computation graph



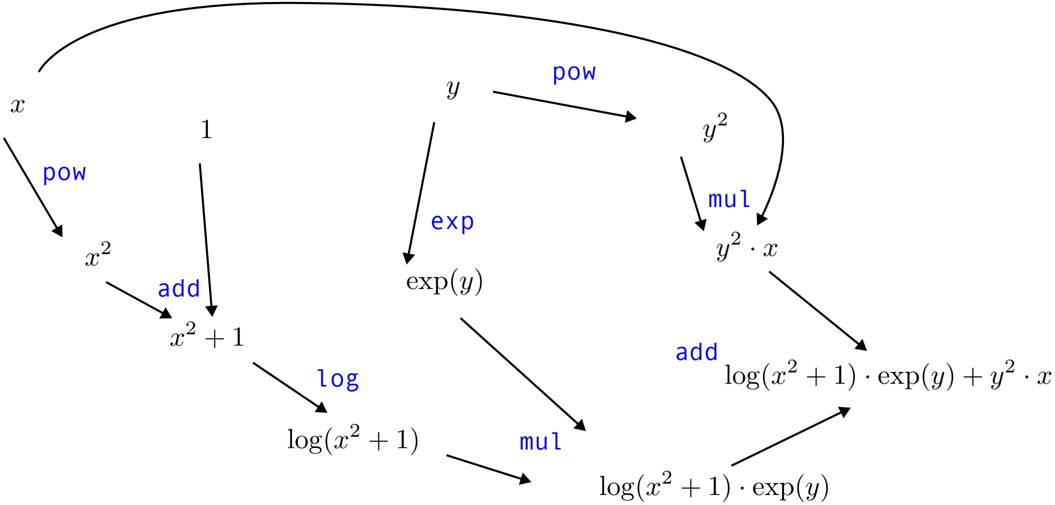
Exercise 2.A

Implement `ag.Scalar.topological_sort`

```
        def topological_sort(self):
            ## EXERCISE 2.A
            ## YOUR CODE HERE
```

```
Check your answer
[z2:x,                    leaves first
 z4:x^2    pow(2),
 z1:1,
 z5:x^2 + 1    add,
 z6:log(x^2 + 1)    log,
 z3:y,
 z7:exp(y)    exp,
 z8:log(x^2 + 1) * exp(y)    mul,
 z9:y^2    pow(2),
 z10:y^2 * x    mul,
 z11:log(x^2 + 1) * exp(y) + y^2 * x    add]        root last
```

Eval $\nabla f(x, y) = \log(x^2 + 1)\exp(y) + y^2 x$ at $(2, 3)$



Backward and add

```python
def backward(self):
    self.grad = 1.0
    topo_order = self.topological_sort()
    for node in reversed(topo_order):
        node._backward()

def __add__(self, other):
    assert isinstance(other, ag.Scalar)
    output = ag.Scalar(self.value + other.value,
                       inputs=[self, other], op="add")
    def _backward():
        # pass
        self.grad += output.grad      1.0
        other.grad += output.grad     1.0

    output._backward = _backward
    return output
```

Exercise 2.B

Implement `ag.Scalar.topological_sort`

```
        def __mul__(self, other):
            # ...
            def _backward():
                ## EXERCISE 2.B
                ## YOUR CODE HERE
                return None
# ... and also the backward function in
        def __pow__(self, exponent): # exponent is just a python float
# ... and in
    def exp(input):
```

self * other

x ** exponent

```
Check your answer
f(x,y) 50.3264246157732
x.grad 25.068429538550134
y.grad 44.3264246157732
```

Exercise 2.C

Implement `ag.Scalar.relu`

```python
    def relu(input):
        output = ag.Scalar(max(0, input.value), inputs=[input], op="relu")
        def _backward():
            ## EXERCISE 2.C
            ## YOUR CODE HERE
# ...
```

```python
# Check your answer with the plot
xs_raw = [i - 5 for i in range(10)]
xs = [ag.Scalar(i) for i in xs_raw]
ys = [ag.relu(x)**3 for x in xs]
[y.backward() for y in ys]
grads = [x.grad for x in xs]
plt.plot(xs_raw, grads, label='autograd')
plt.plot(xs_raw, [3*max(0,x_raw)**2 for x_raw in xs_raw], label='manual grad')
```

What can we do with `ag.Scalar`?

For the remaining of this class, we will build a "knock-off" PyTorch consisting of:

- A `Model` class that encapsulates the parameters and the forward pass
- A `Loss` class for calculating Mean Squared Error (MSE)
- An `Optimizer` class for performing gradient descent
- A training loop

We will fit a 1-hidden layer neural network

$$f(x; \mathbf{w}_1, \mathbf{b}_1, \mathbf{w}_2, b_2) = \mathbf{w}_2^\top \mathrm{relu}(\mathbf{w}_1 x + \mathbf{b}_1) + b_2$$

where $\mathbf{w}_1$ and $\mathbf{w}_2$ are both vectors of `n_hidden` dimensions.

The `Model` class

```python
class Model:
    def __init__(self, n_hidden, rng_seed=42):
        np.random.seed(rng_seed)

        w1np = np.random.randn(n_hidden)
        # ...
        b2np = np.random.randn(1)

        self.w1 = [ag.Scalar(val) for val in w1np]
        # ...
        self.b2 = [ag.Scalar(val) for val in b2np]

        self.parameters = self.w1 + self.b1 + self.w2 + self.b2

    def forward(self, x): # ...
```

The `forward` function

```python
    def forward(self, x):
        # "upgrade" x into ag.Scalars
        x_scalar = [ag.Scalar(val) for val in x]
        n_samples = len(x_scalar)

        # calculate the forward

        ## YOUR CODE HERE
        return [ag.Scalar(0.0) for i in range(n_samples)]
```

The Loss class

```
class Loss:
    def mse(self, predictions, targets):
        # mean squared error
        assert len(predictions) == len(targets)
        n_samples = len(predictions)
        loss = ag.Scalar(0.0)

        # YOUR CODE HERE

        return loss
```

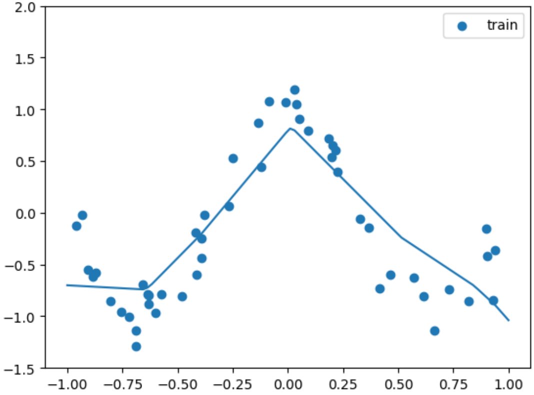
The `Optimizer` class

```python
class Optimizer:
    def __init__(self, parameters, lr=0.01):
        self.parameters = parameters
        self.lr = lr

    def zero_grad(self):
        # YOUR CODE HERE
        pass

    def step(self):
        # YOUR CODE HERE
        pass
```

Training Loop

```
model = Model(n_hidden=20)
loss_fn = Loss()
optimizer = Optimizer(model.parameters, lr=0.1)

for epoch in range(100):
    optimizer.zero_grad()
    output = model.forward(xnp)
    loss = loss_fn.mse(output, ynp)
    loss.backward()
    optimizer.step()
    if epoch % 10 == 0:
        print(f"Iteration {epoch}, Loss: {loss.value}")
```

Exercise 3



References I