Neural Networks Continued

CS 6375.002: Machine Learning

Fall 2023

Backpropagation in General

- We saw how to do backpropagation for a specific architecture:
  - Fully-connected layers
  - Uniform activation function across all neurons (sigmoid)

- What if we have a different network?
  - Recursive property no longer holds!

- What if we also have over 1 billion parameters?
  - Don’t have enough paper to write out the math!

- Is there a general algorithm for performing backpropagation?

Computation Graphs

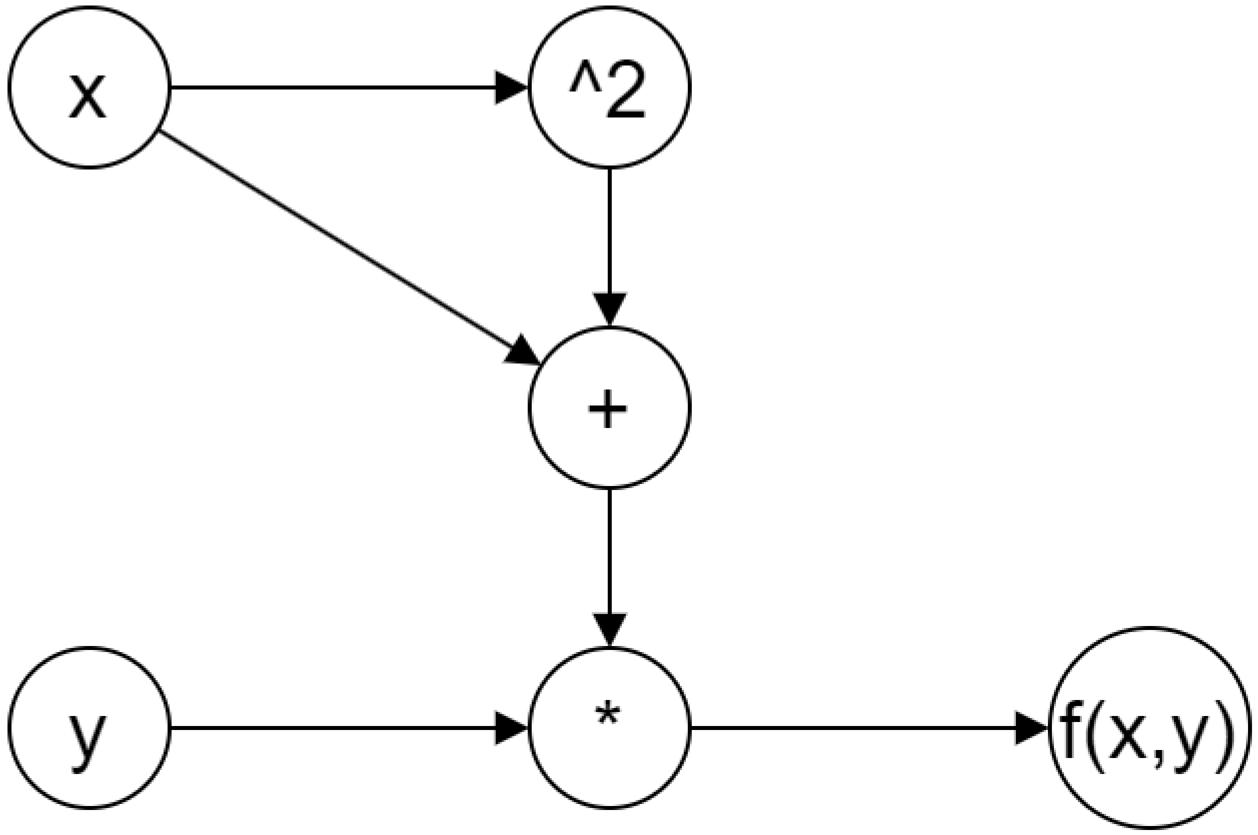
- Idea: Record sequence of operations as a graph
  - We can represent all the operations that a neural network does with a computation graph!

- A computation graph is a directed acyclic graphical representation of a function where:
  - All the nodes with in-degree 0 are the inputs to the function
  - There is a single node with out-degree 0 denoting the output of the function
    - For now, assume that it is a scalar value such as *a loss value*
  - Every other node represents an elementary operation on its inputs
    - All outgoing edges carry the same forward value!
  - Note: NOT the same as a neural network graph!

Computation Graph Example

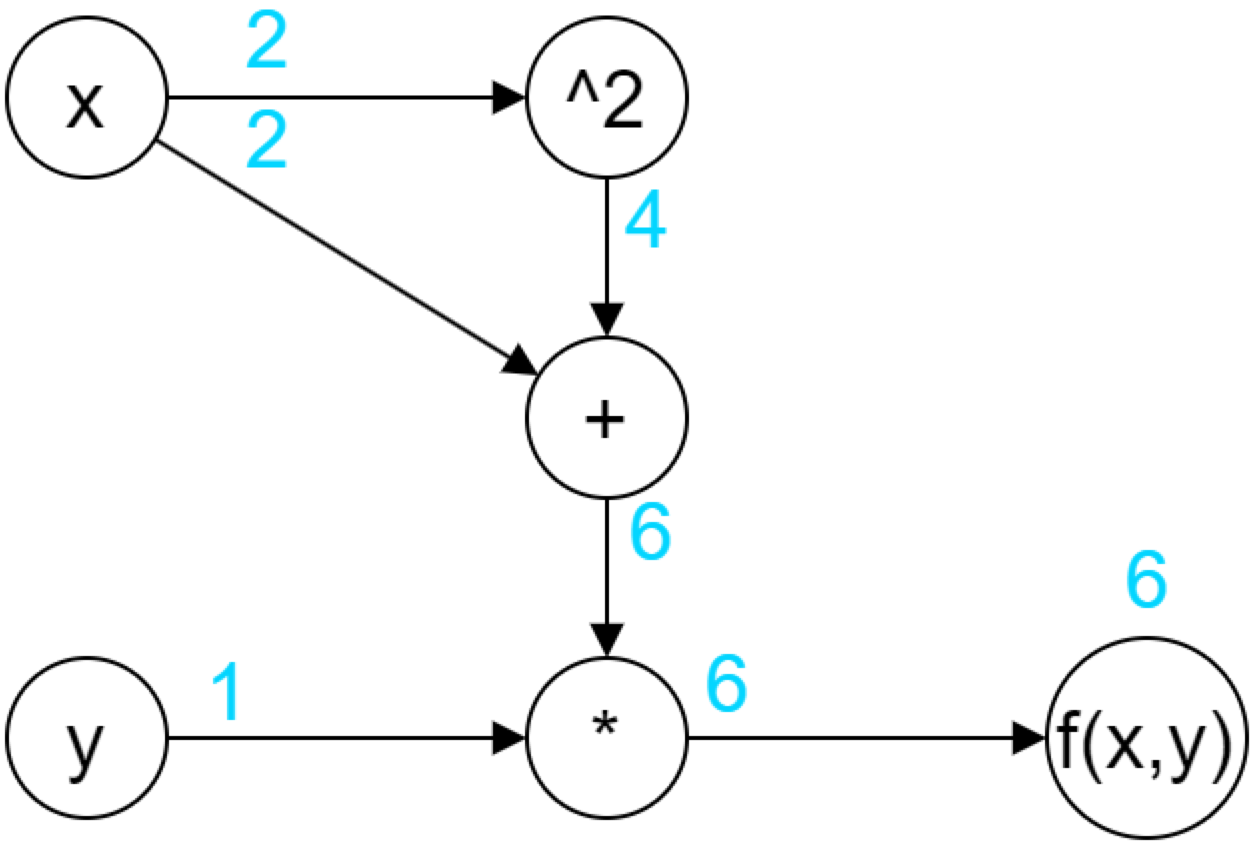
- $f(x, y) = y(x^2 + x)$

- We can set values for the inputs and derive the output based on the operations



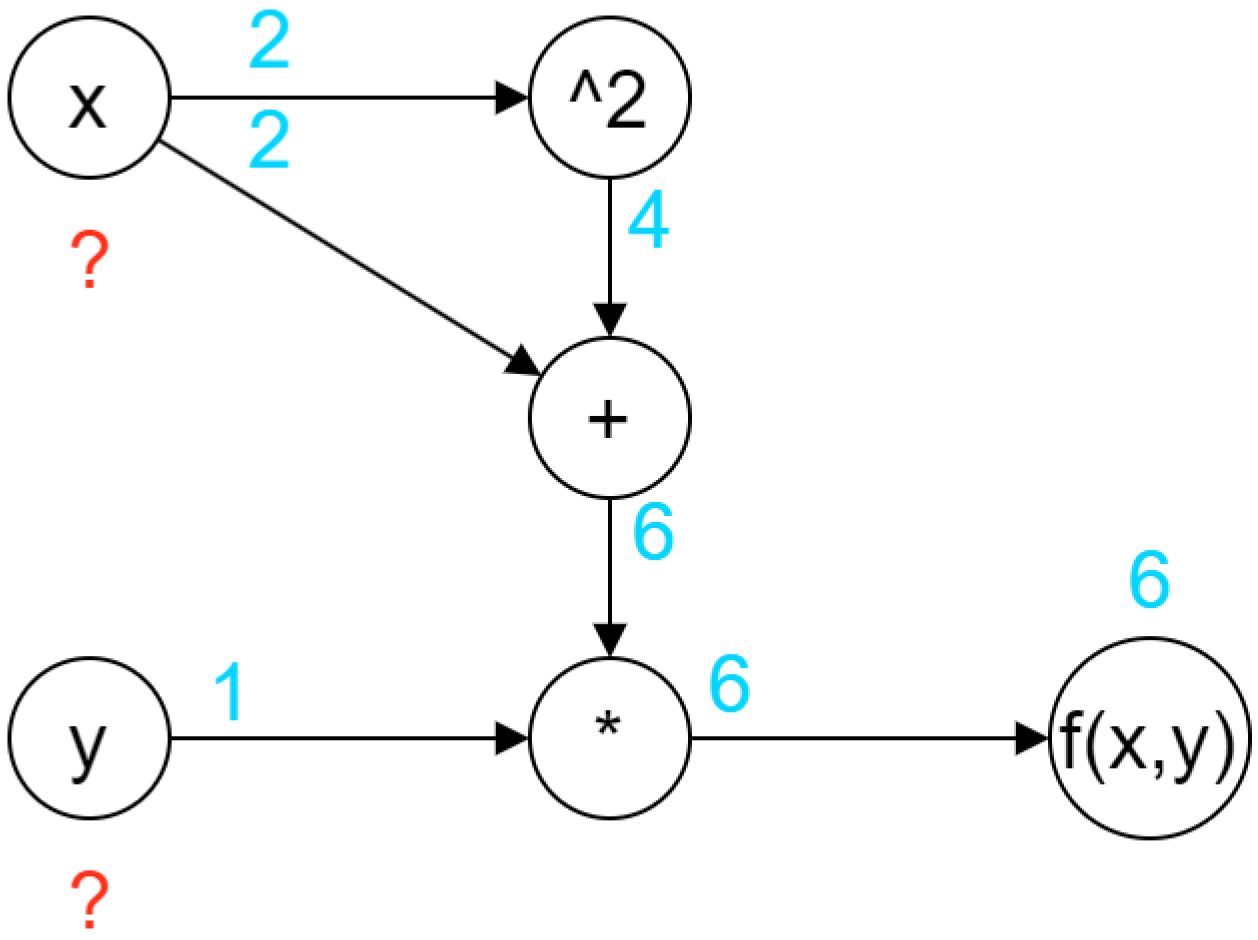
Computation Graph Example

- $f(x,y) = y(x^2 + x)$
- We can set values for the inputs and derive the output based on the operations
  - Let us try x=2, y=1
- Why is this useful?



Computation Graph Example

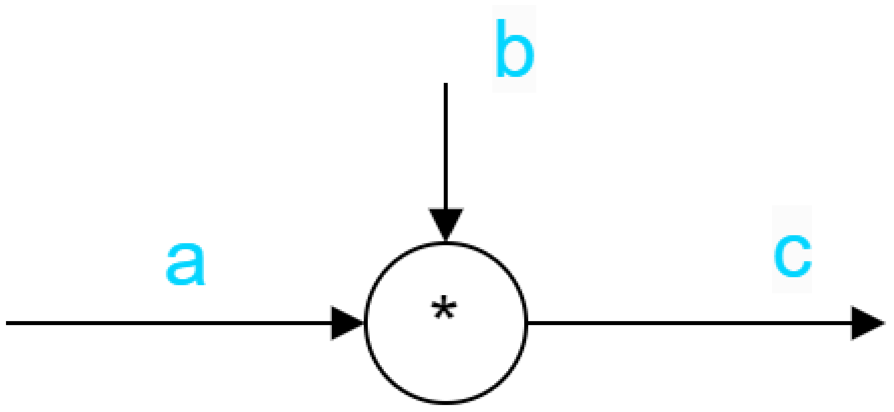
- $f(x, y) = y(x^2 + x)$
- We can set values for the inputs and derive the output based on the operations
  - Let us try x=2, y=1
- Why is this useful?
  - We can utilize a backwards process to get gradients!



Local Gradients

- Each elementary operation admits a simple gradient
  - *Local* refers to the single elementary operation

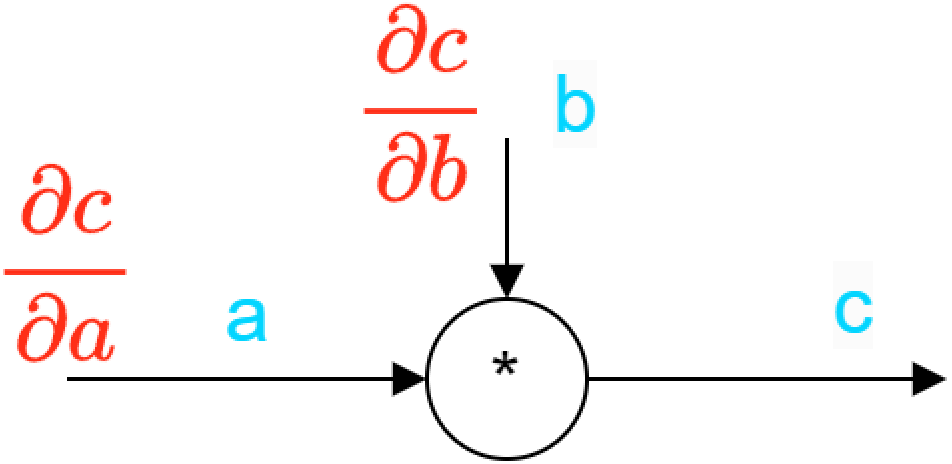
- What gradients do we need to calculate for this example?



Local Gradients

- Each elementary operation admits a simple gradient
  - *Local* refers to the single elementary operation

- What gradients do we need to calculate for this example?
  - $\frac{\partial c}{\partial a}, \frac{\partial c}{\partial b}$

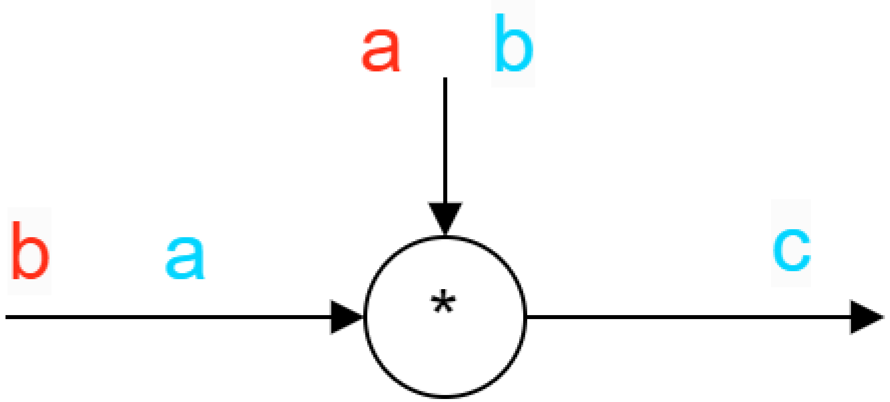


Local Gradients

- Each elementary operation admits a simple gradient
  - *Local* refers to the single elementary operation

- What gradients do we need to calculate for this example?
  - $\frac{\partial c}{\partial a}, \frac{\partial c}{\partial b}$

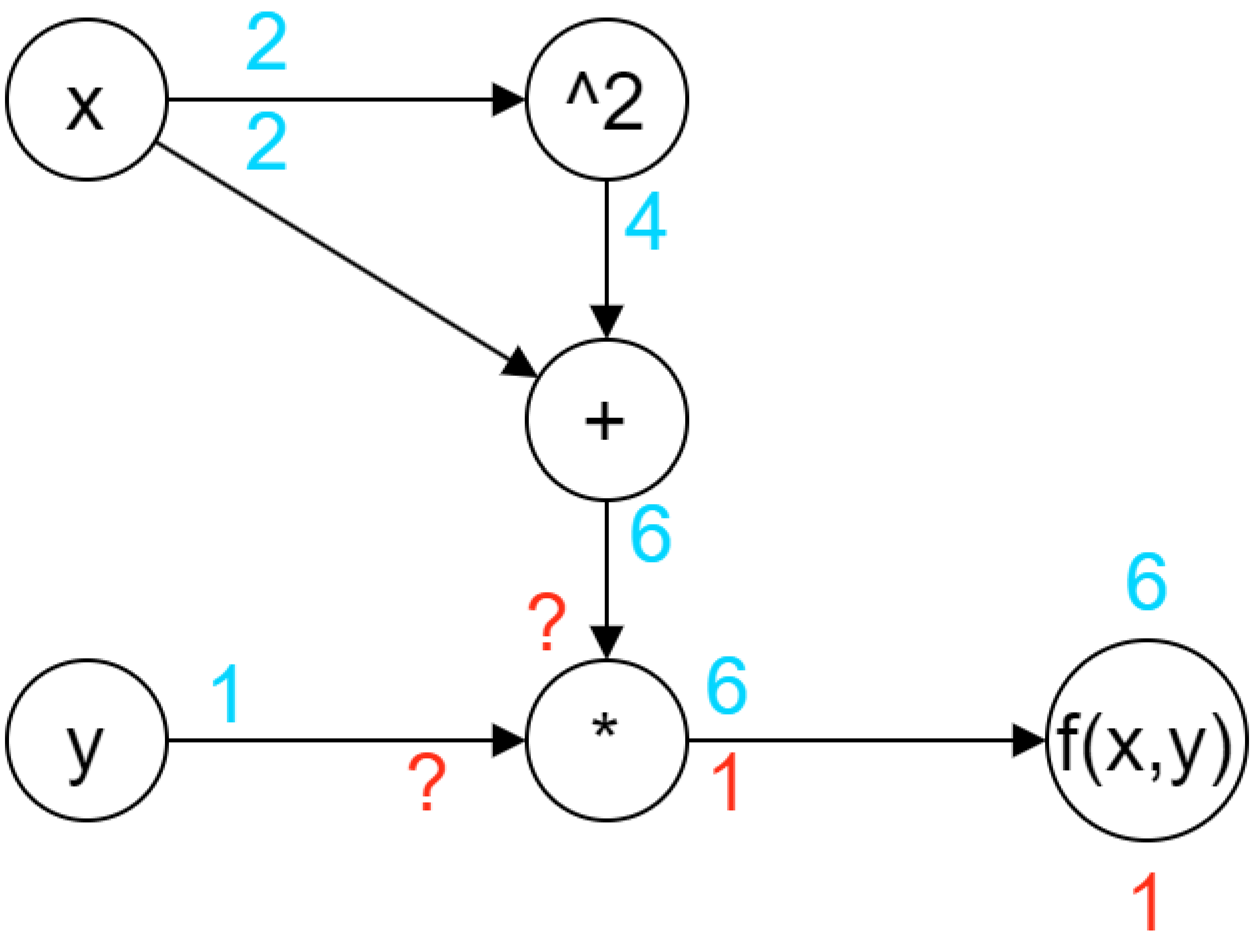
- What are their values?
  - b, a



Propagating Local Gradients

- We can start with $\frac{\partial f}{\partial f} = 1$ and begin propagating!
  - Multiply the local gradients with the backpropagated gradient

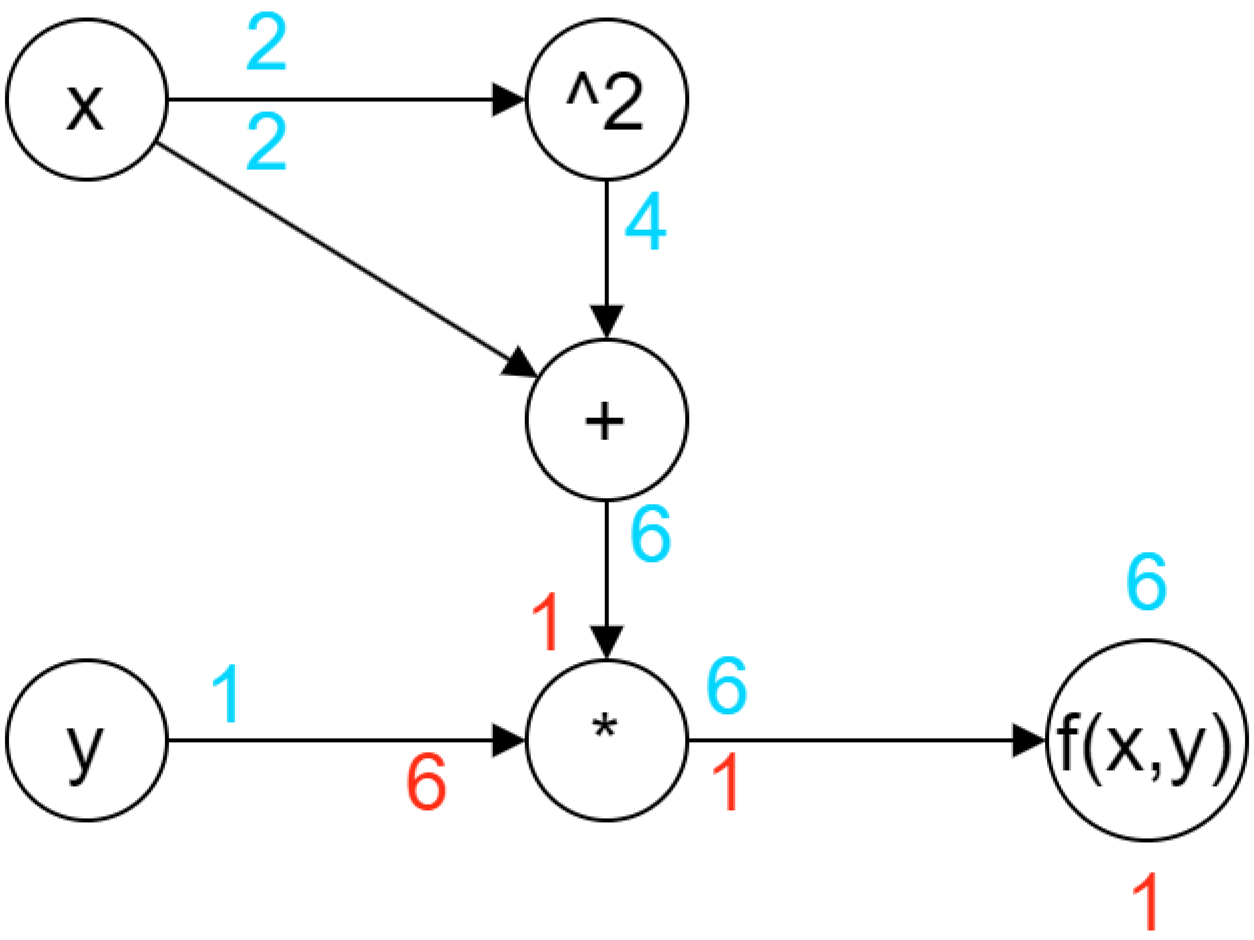
- What are the gradients marked with ?



Propagating Local Gradients

- We can start with $\frac{\partial f}{\partial f} = 1$ and begin propagating!
  - Multiply the local gradients with the backpropagated gradient

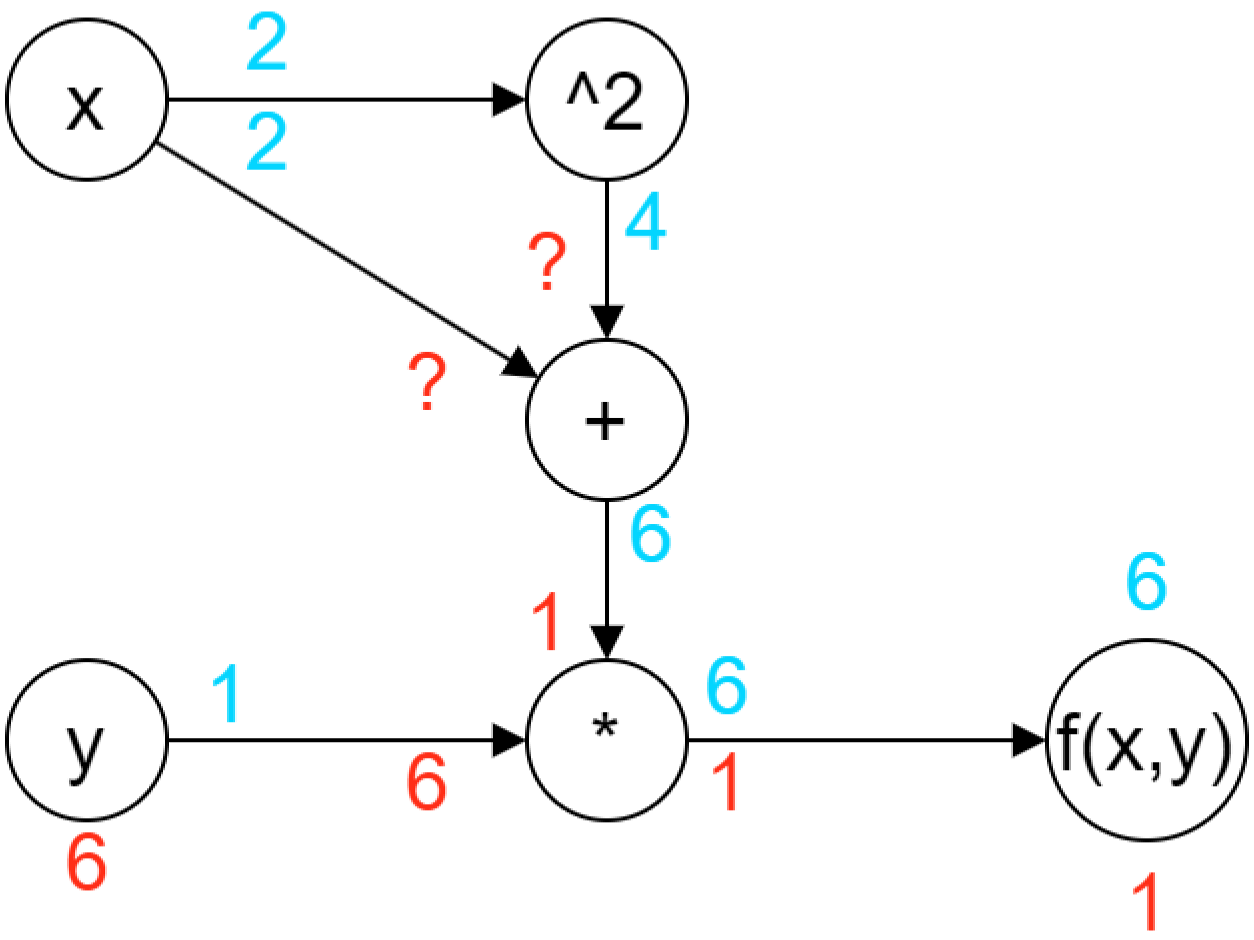
- What are the gradients marked with ?
  - 6: 1 * 6
  - 1: 1 * 1



Propagating Local Gradients

- We can start with $\frac{\partial f}{\partial f} = 1$ and begin propagating!
  - Multiply the local gradients with the backpropagated gradient

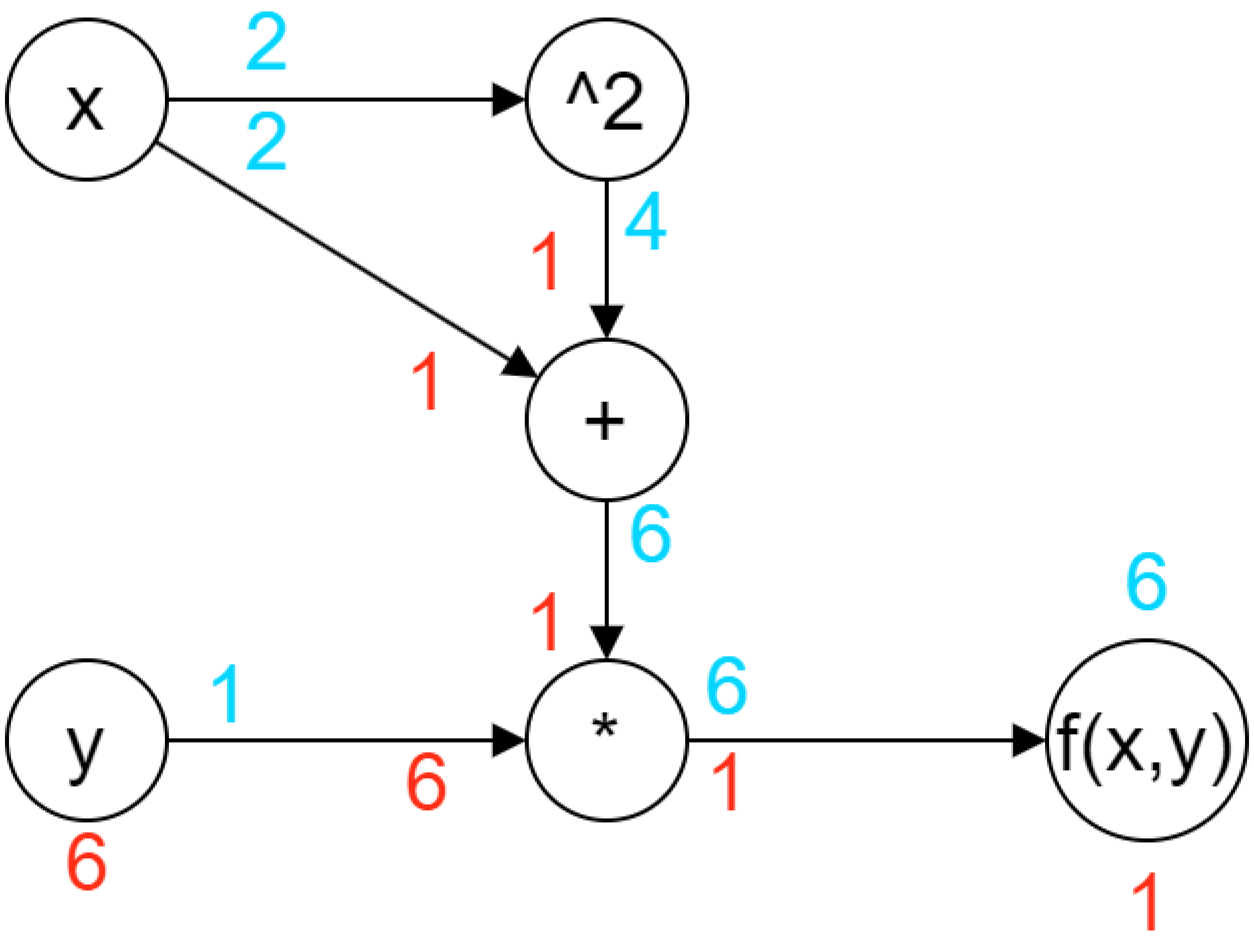
- What are the gradients marked with ?



Propagating Local Gradients

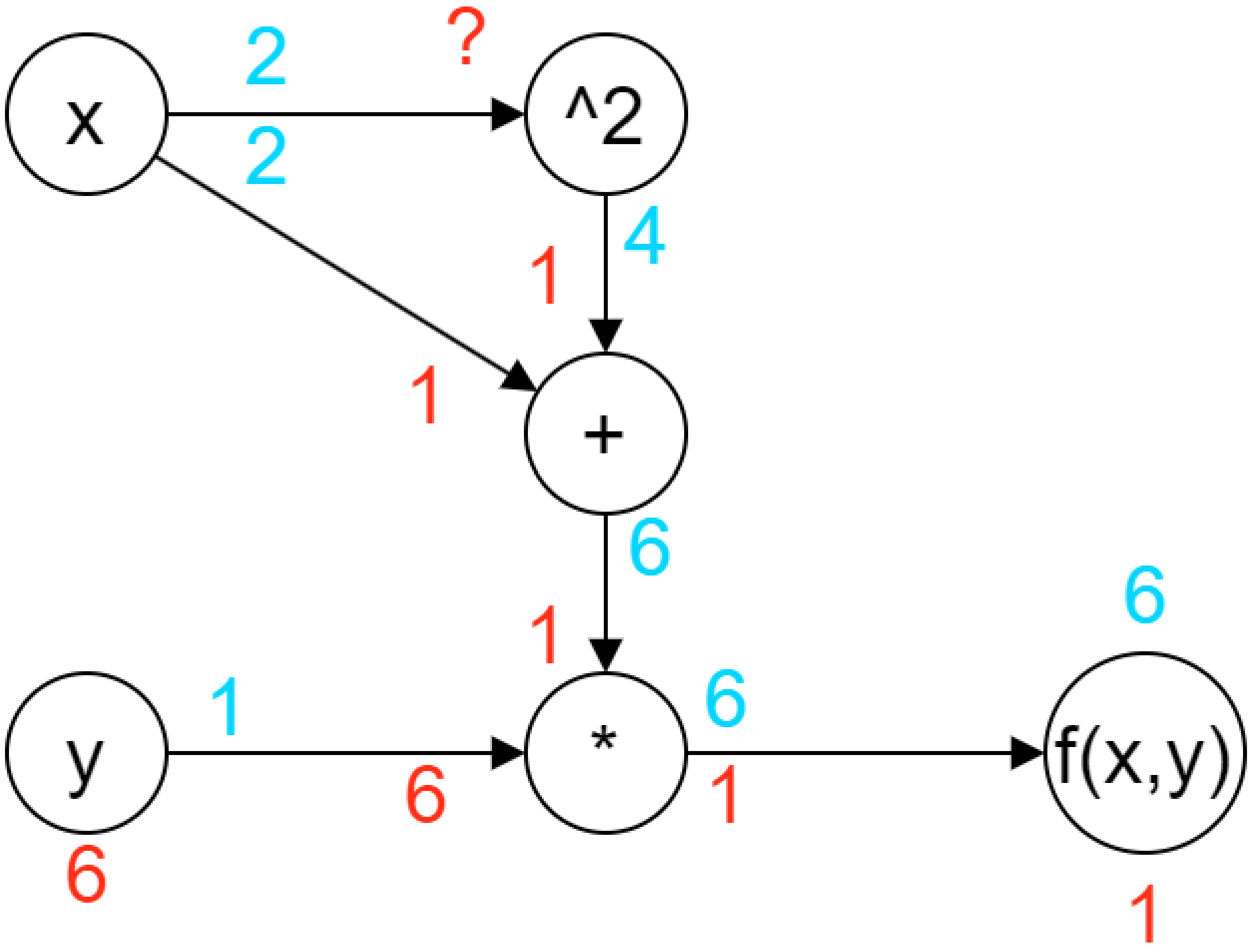
- We can start with $\frac{\partial f}{\partial f} = 1$ and begin propagating!
  - Multiply the local gradients with the backpropagated gradient

- What are the gradients marked with ?
  - Both are 1! Why?
  - $\frac{\partial}{\partial a}(a+b) = 1$
  - $\frac{\partial}{\partial b}(a+b) = 1$
  - $1 * 1 = 1!$



Propagating Local Gradients

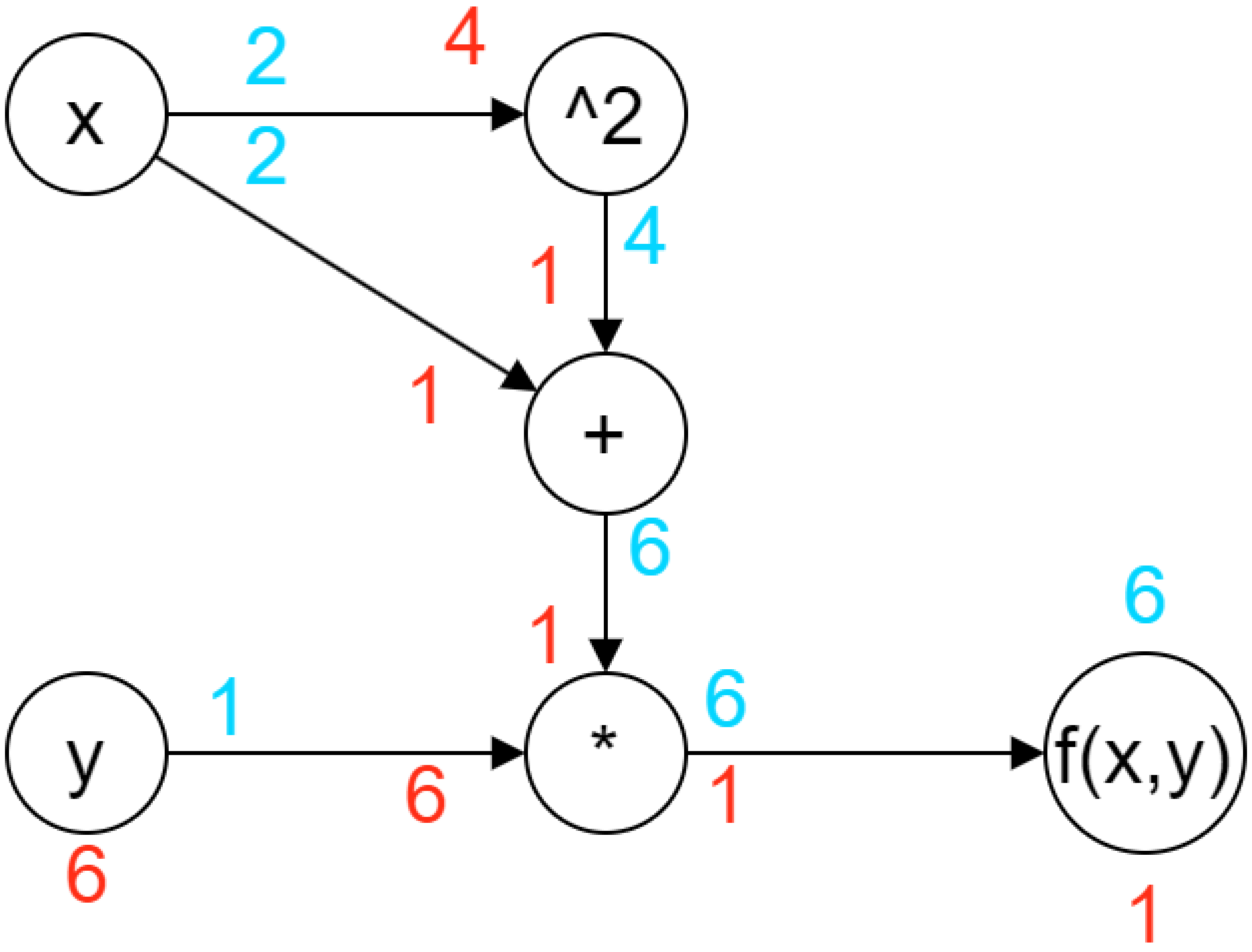
- We can start with $\frac{\partial f}{\partial f} = 1$ and begin propagating!
  - Multiply the local gradients with the backpropagated gradient
- What is the gradient marked with ?



Propagating Local Gradients

- We can start with $\frac{\partial f}{\partial f} = 1$ and begin propagating!
  - Multiply the local gradients with the backpropagated gradient

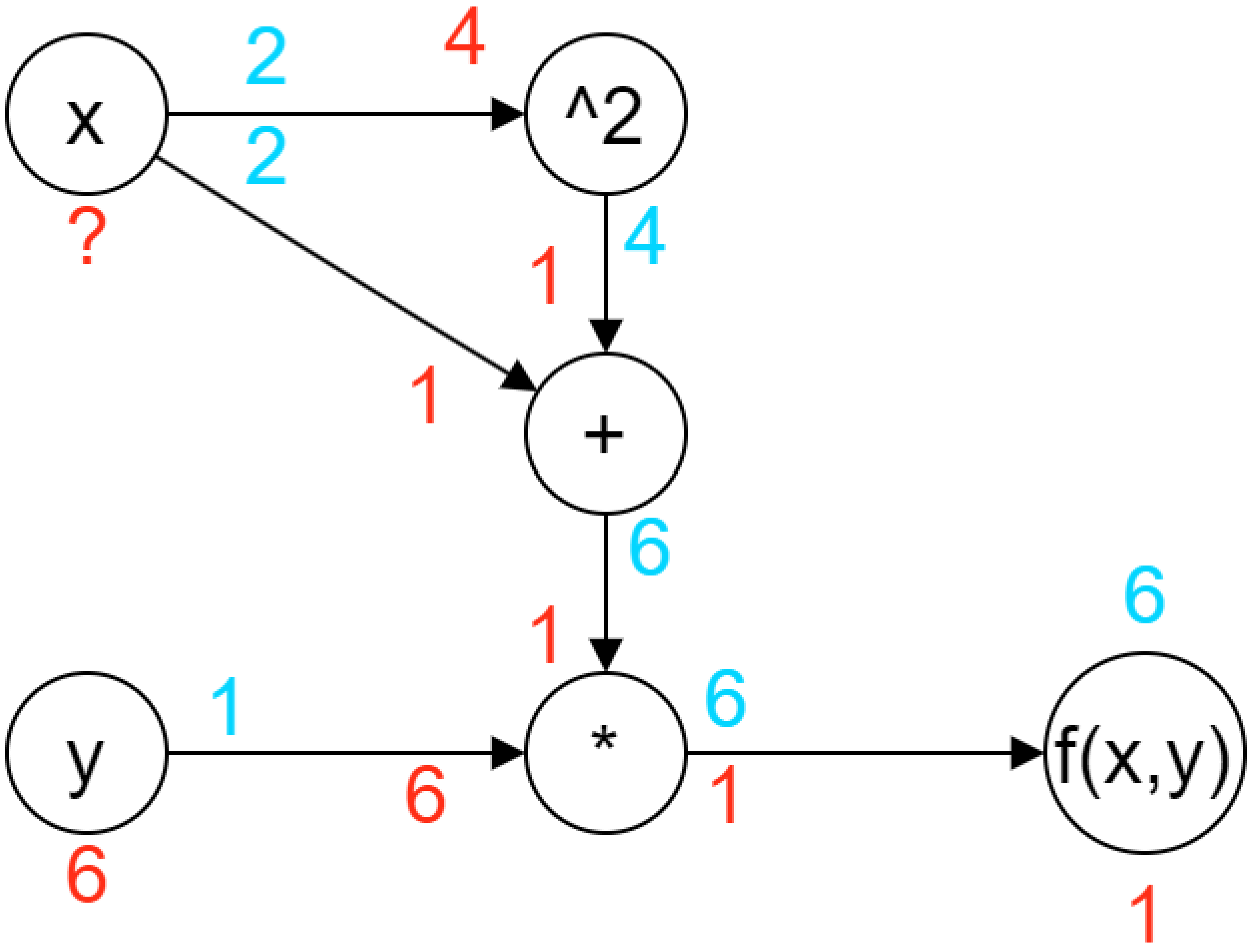
- What is the gradient marked with ?
  - 4. Why?
  - $\frac{\partial}{\partial a}(a^2) = 2a$
  - $2 * 2 * 1 = 4$



Propagating Local Gradients

- We can start with $\frac{\partial f}{\partial f} = 1$ and begin propagating!
  - Multiply the local gradients with the backpropagated gradient

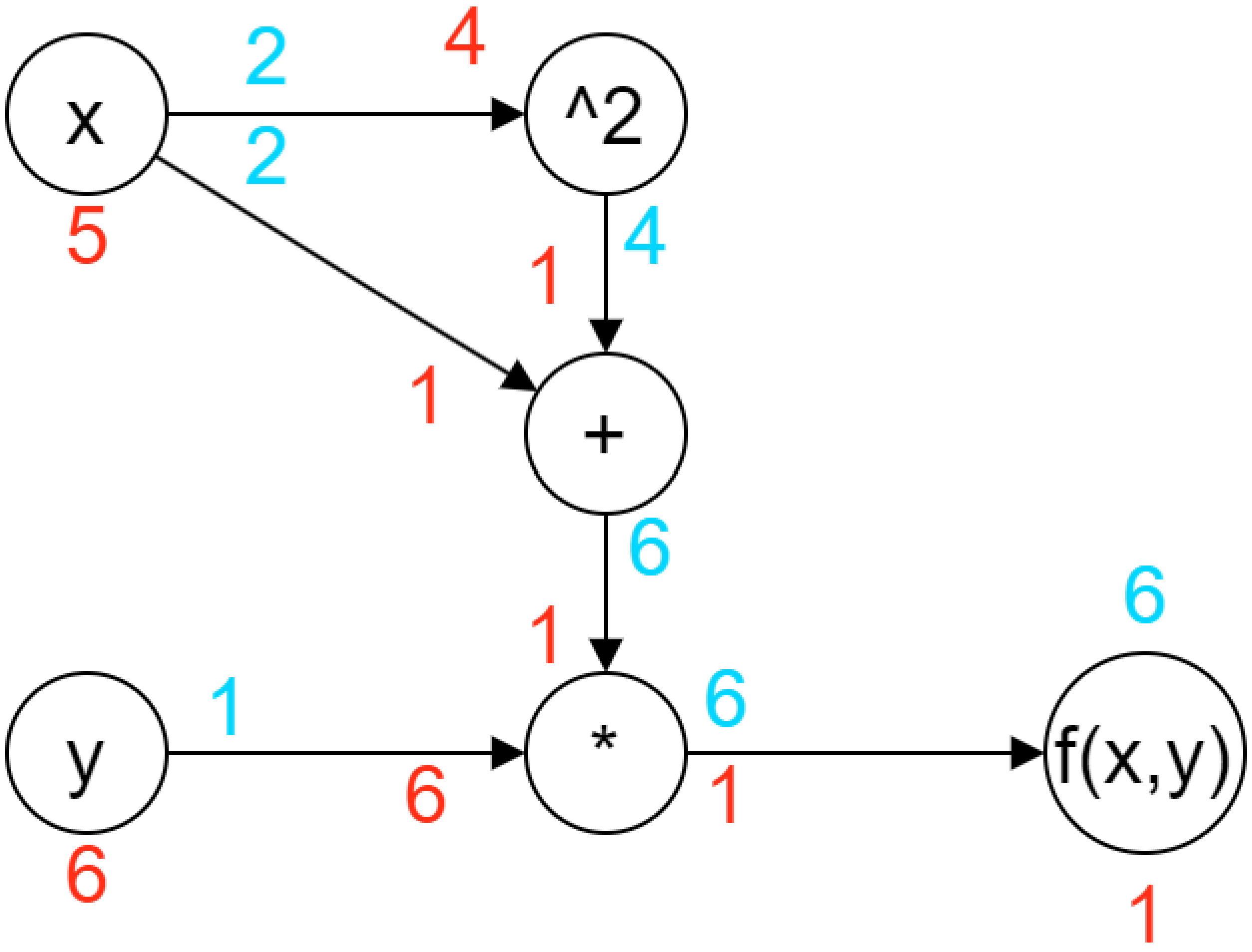
- What should our final gradient for x be?



Propagating Local Gradients

- We can start with $\frac{\partial f}{\partial f} = 1$ and begin propagating!
  - Multiply the local gradients with the backpropagated gradient

- What should our final gradient for x be?
  - 5. Why?
  - We add gradients incident on a node. But… why?



Why does this work?

The Chain Rule. Again.

- We can backpropagate gradients in this fashion and accumulate gradients additively in nodes because of the chain rule!

- Recall the multi-variate chain rule: If $f(a) = g\big(h(a)\big)$, then

$$J_f(a) = J_g\big(h(a)\big)J_h(a)$$

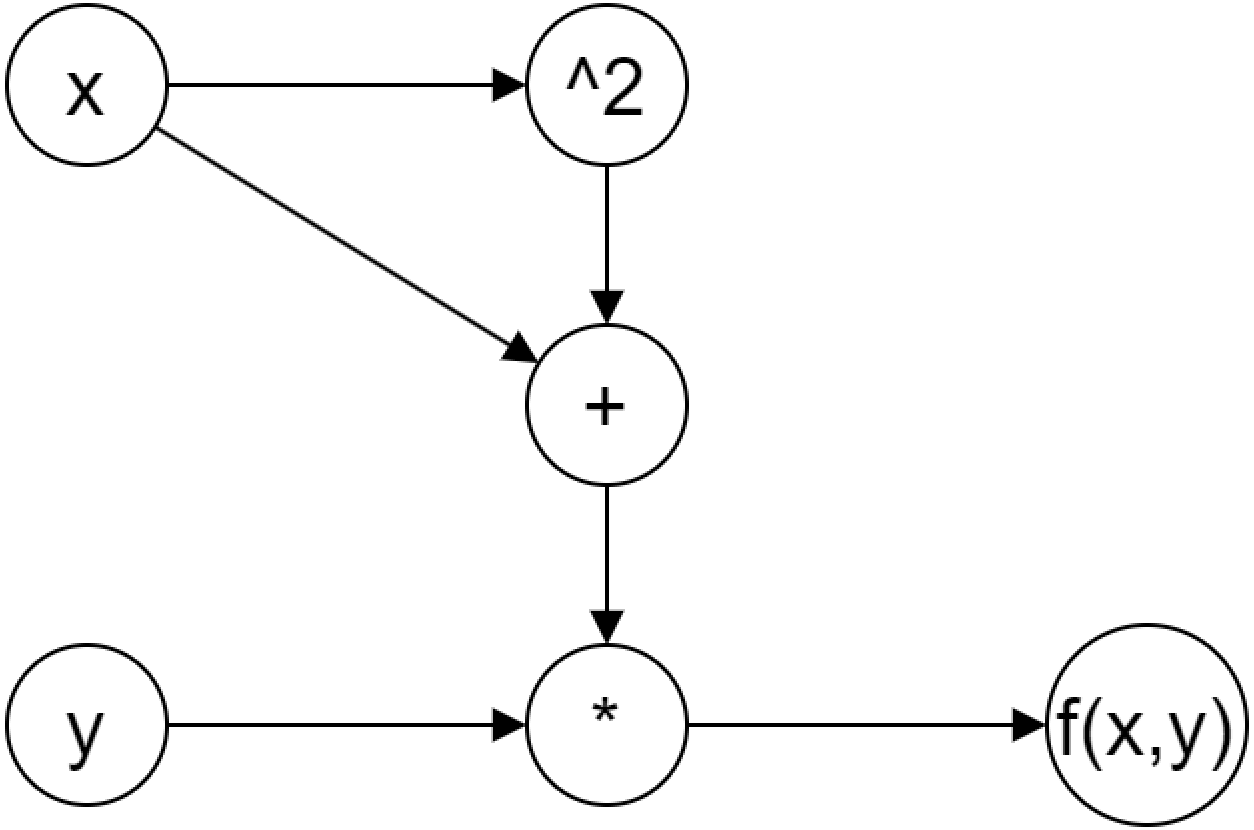
- Also recall our assumption that f produces a single scalar value.
  - The Jacobian matrix $J_f(a)$ is a $1 \times \dim(a)$ matrix!

The Chain Rule. Again.

- These assumptions allow us to express the gradient as

$$J_f(a) = \sum_k \left[\frac{\partial g}{\partial h_k}\right]_{h=h(a)} \left[\frac{\partial h_k}{\partial c}\right]_{c=a}$$

- How does this map back to our computation graph?

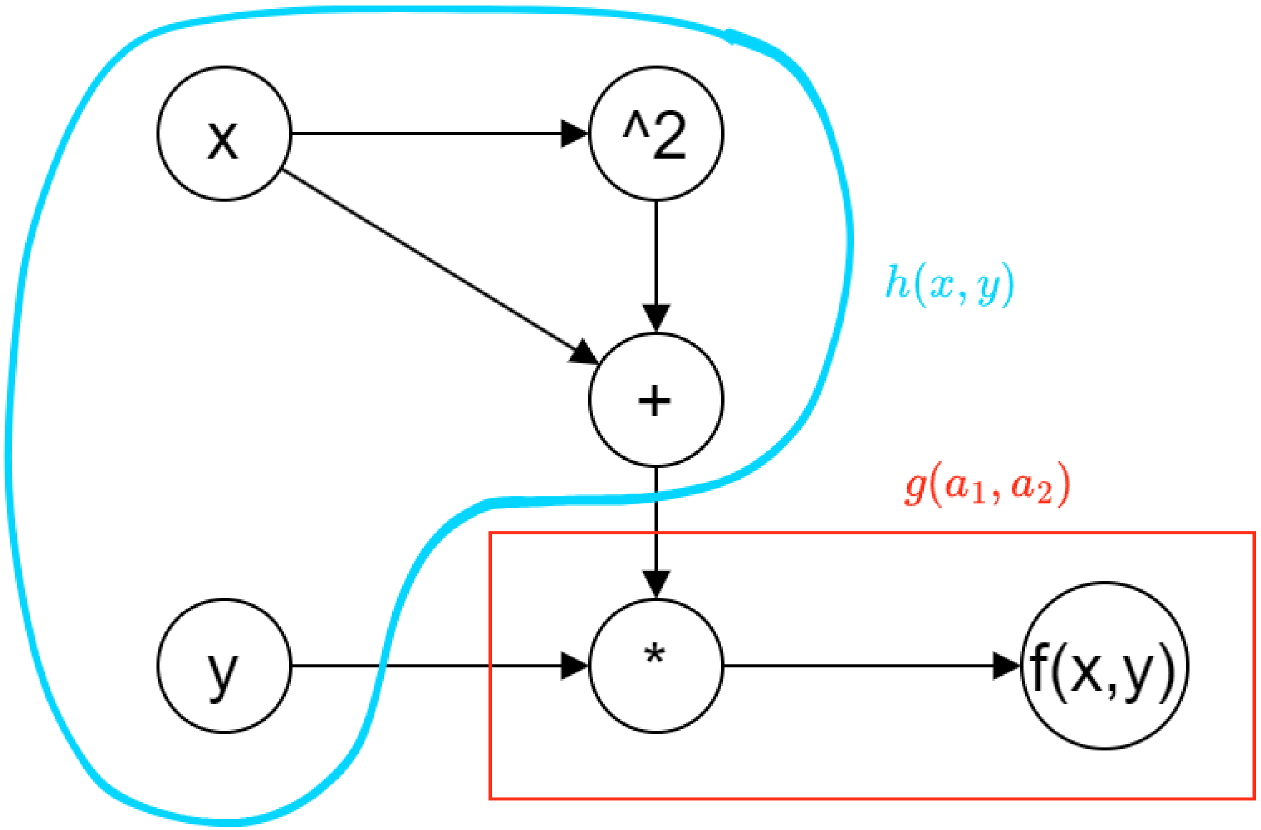


The Chain Rule. Again.

- WLOG, let us look at the gradient wrpt x. According to chain rule,

$$\frac{\partial f}{\partial x} = \sum_k \left[\frac{\partial g}{\partial h_k}\right]_{h=h(a)} \left[\frac{\partial h_k}{\partial x}\right]_{c=a}$$

- We can separate the graph into g and h functions!

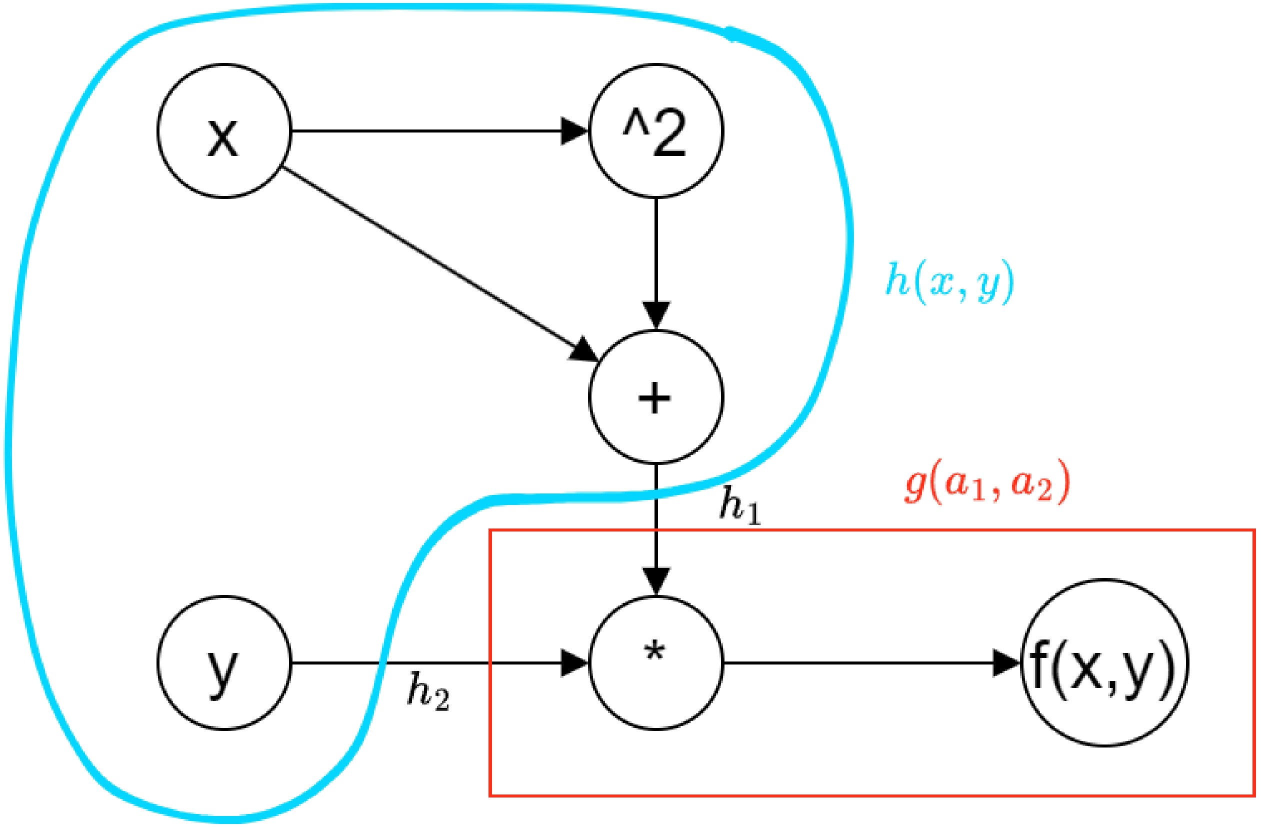


The Chain Rule. Again.

- We can see that $k = 2$, so

$$\frac{\partial f}{\partial x} = \left[\frac{\partial g}{\partial h_1}\right]_{h=h(a)} \left[\frac{\partial h_1}{\partial x}\right]_{c=a} + \left[\frac{\partial g}{\partial h_2}\right]_{h=h(a)} \left[\frac{\partial h_2}{\partial x}\right]_{c=a}$$

- What does this tell us?
  - If h_i is not on a path from a specific node x to the output f(x,y), then $\frac{\partial h_i}{\partial x} = 0$
  - If h_i IS on a path from a specific node x to the output f(x,y), then we need to multiply $\frac{\partial h_i}{\partial x}$ with the running gradient to h_i's input node, which is $\frac{\partial g}{\partial h_i}$!

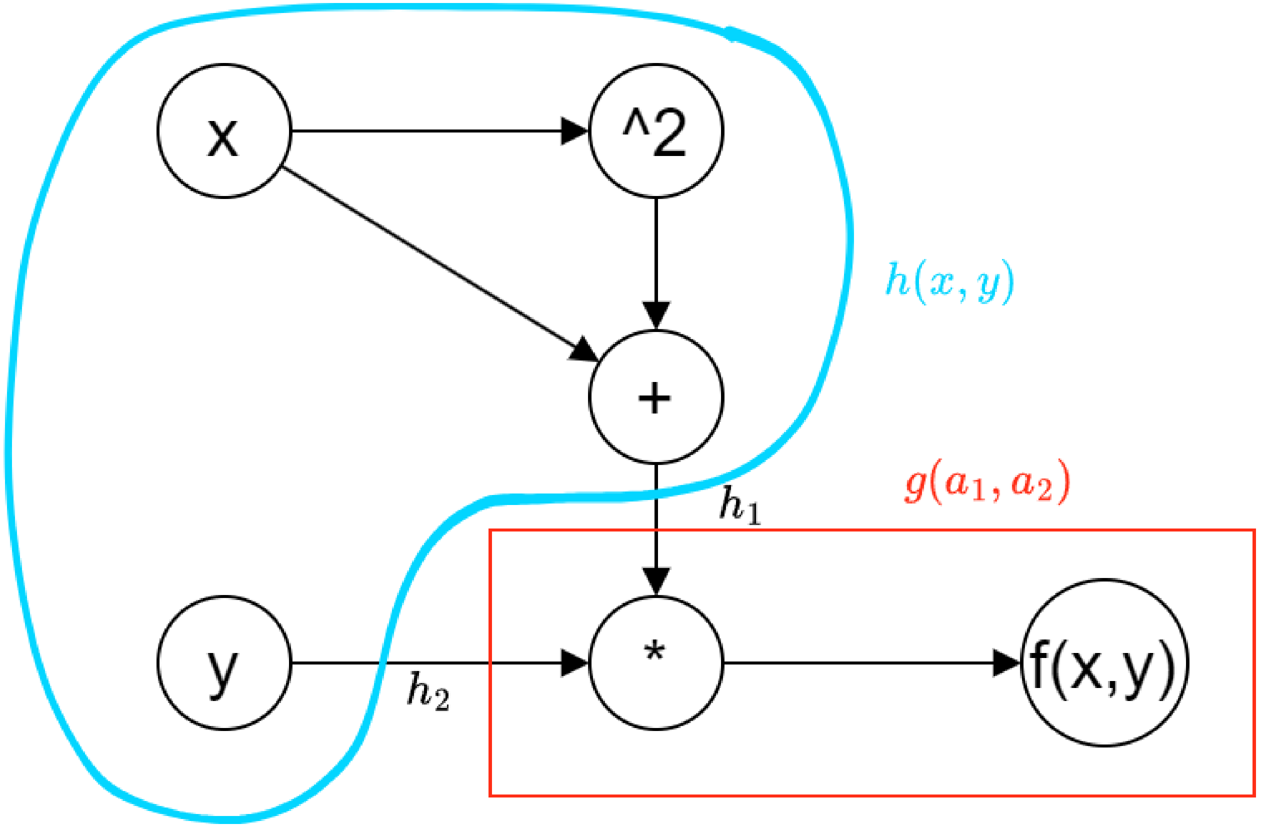


The Chain Rule. Again.

- We can see that $k = 2$, so

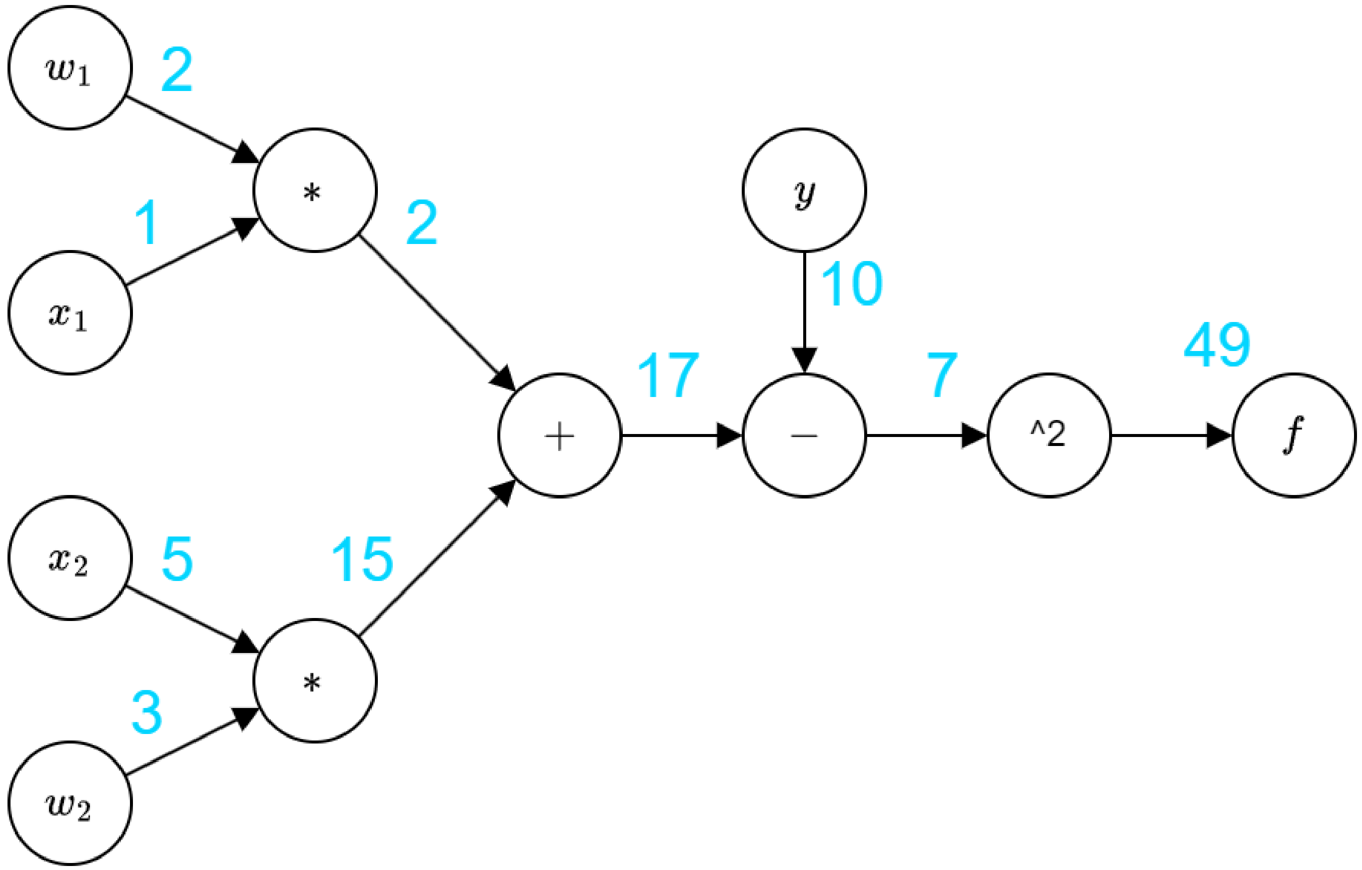
$$\frac{\partial f}{\partial x} = \left[\frac{\partial g}{\partial h_1}\right]_{h=h(a)} \left[\frac{\partial h_1}{\partial x}\right]_{c=x} + \left[\frac{\partial g}{\partial h_2}\right]_{h=h(a)} \left[\frac{\partial h_2}{\partial x}\right]_{c=x}$$

- In summary?
  - Both observations allow us to backpropagate gradients in our prescribed fashion!
  - The addition above allows us to accumulate gradients into nodes, which is why we added before!



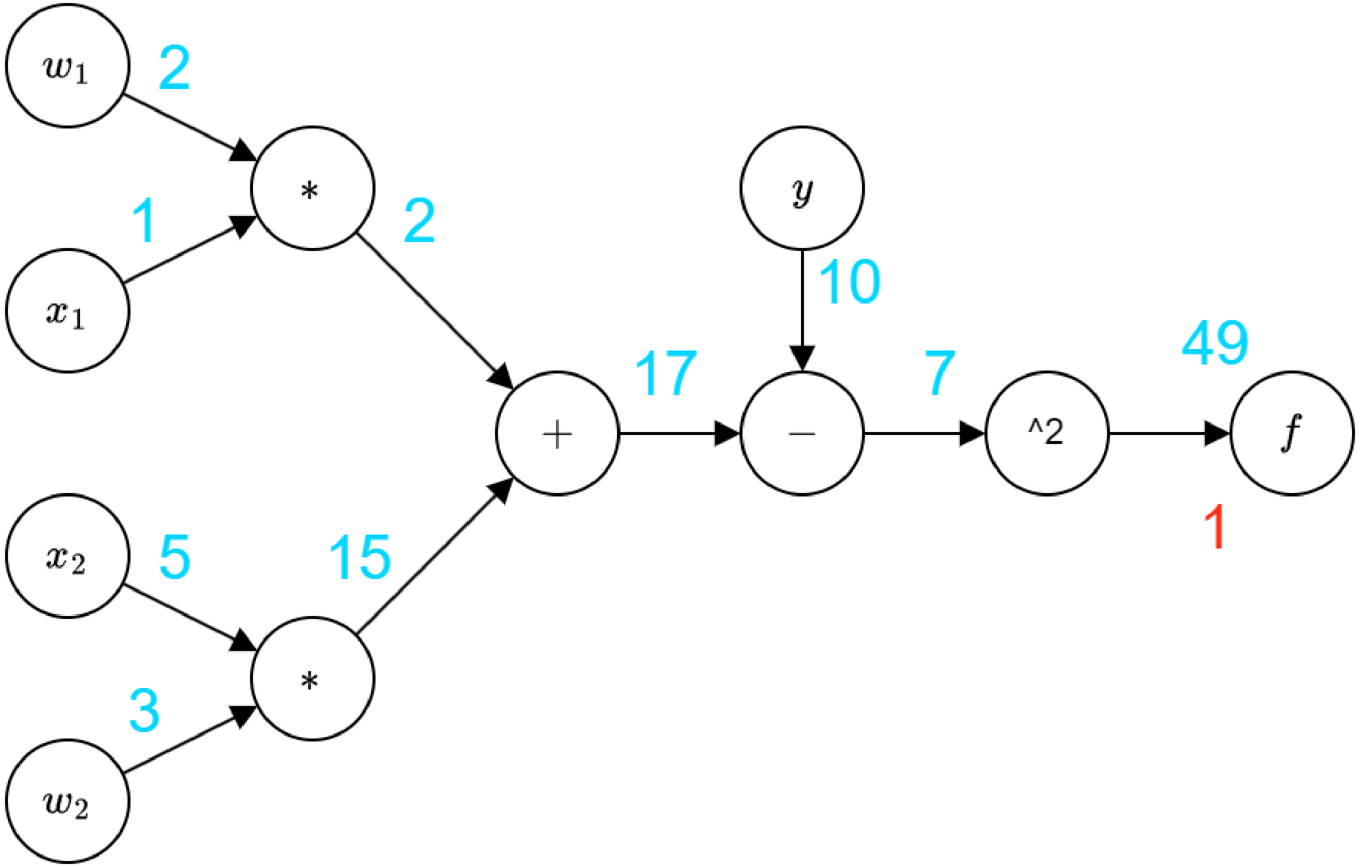
One More Backpropagation Example

- $L(x, y, w) = (w^T x - y)^2$ for 2D input (our favorite regression loss!)



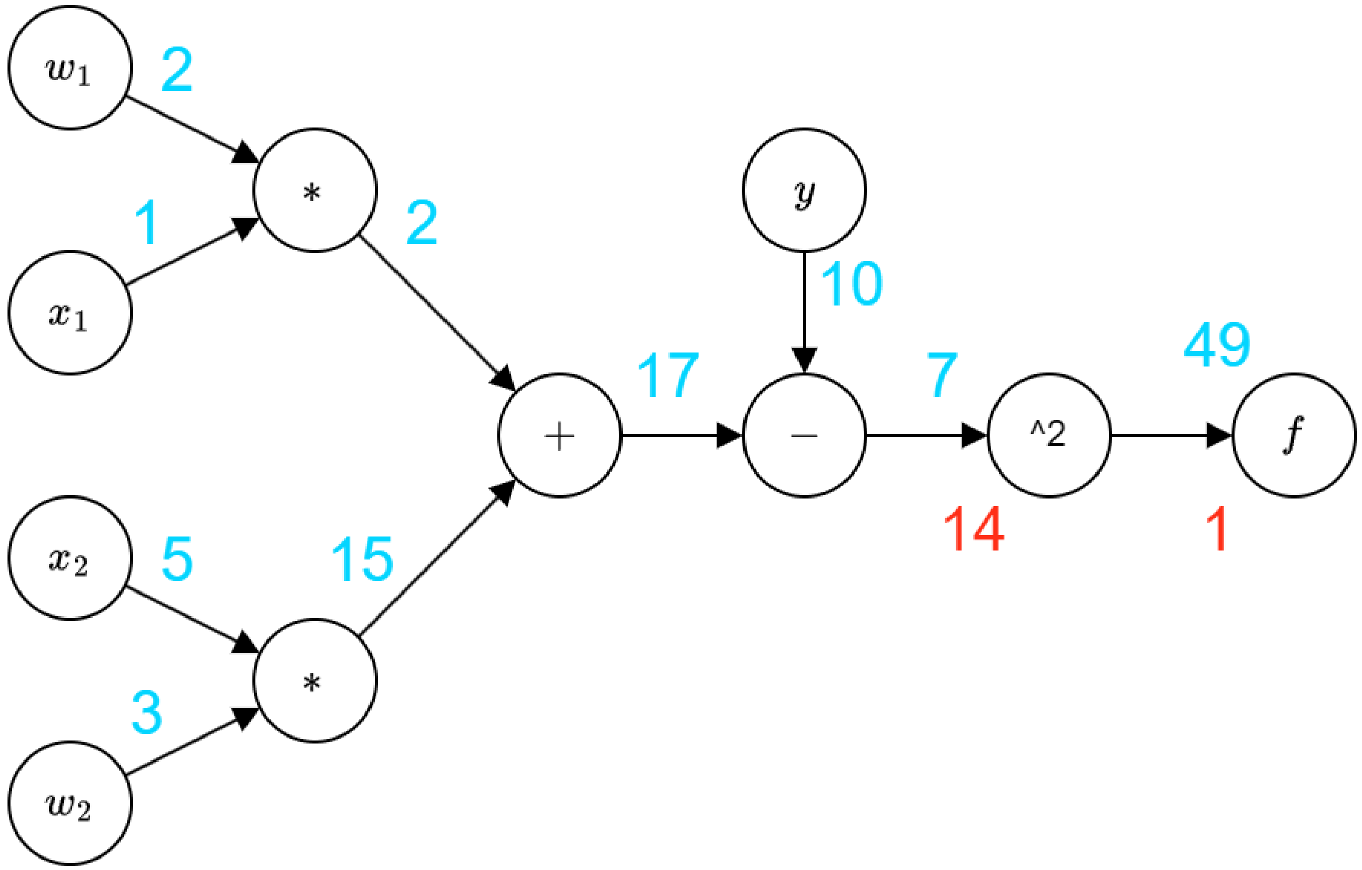
One More Backpropagation Example

- $L(x, y, w) = (w^T x - y)^2$ for 2D input (our favorite regression loss!)



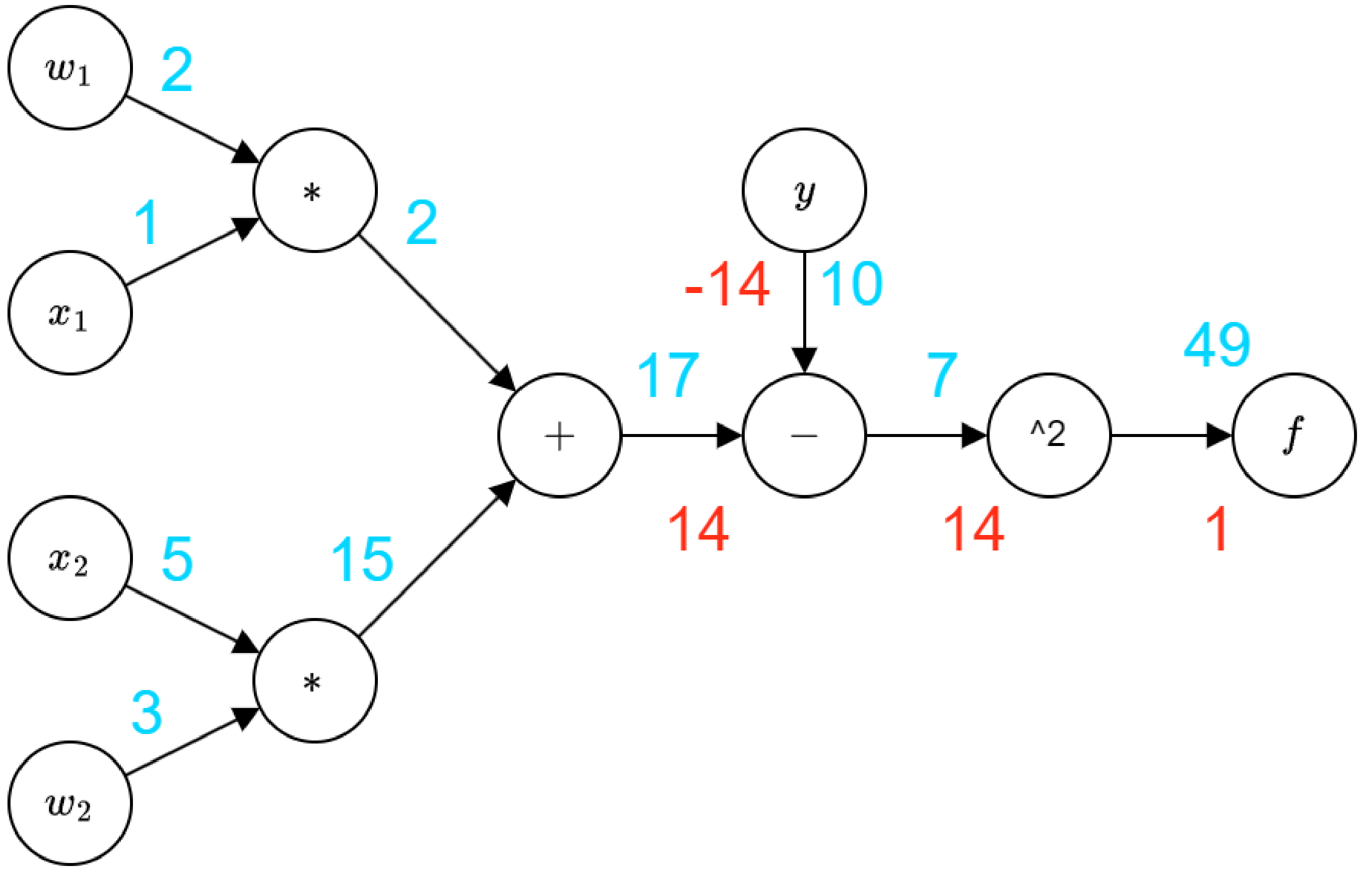
One More Backpropagation Example

- $L(x, y, w) = (w^T x - y)^2$ for 2D input (our favorite regression loss!)



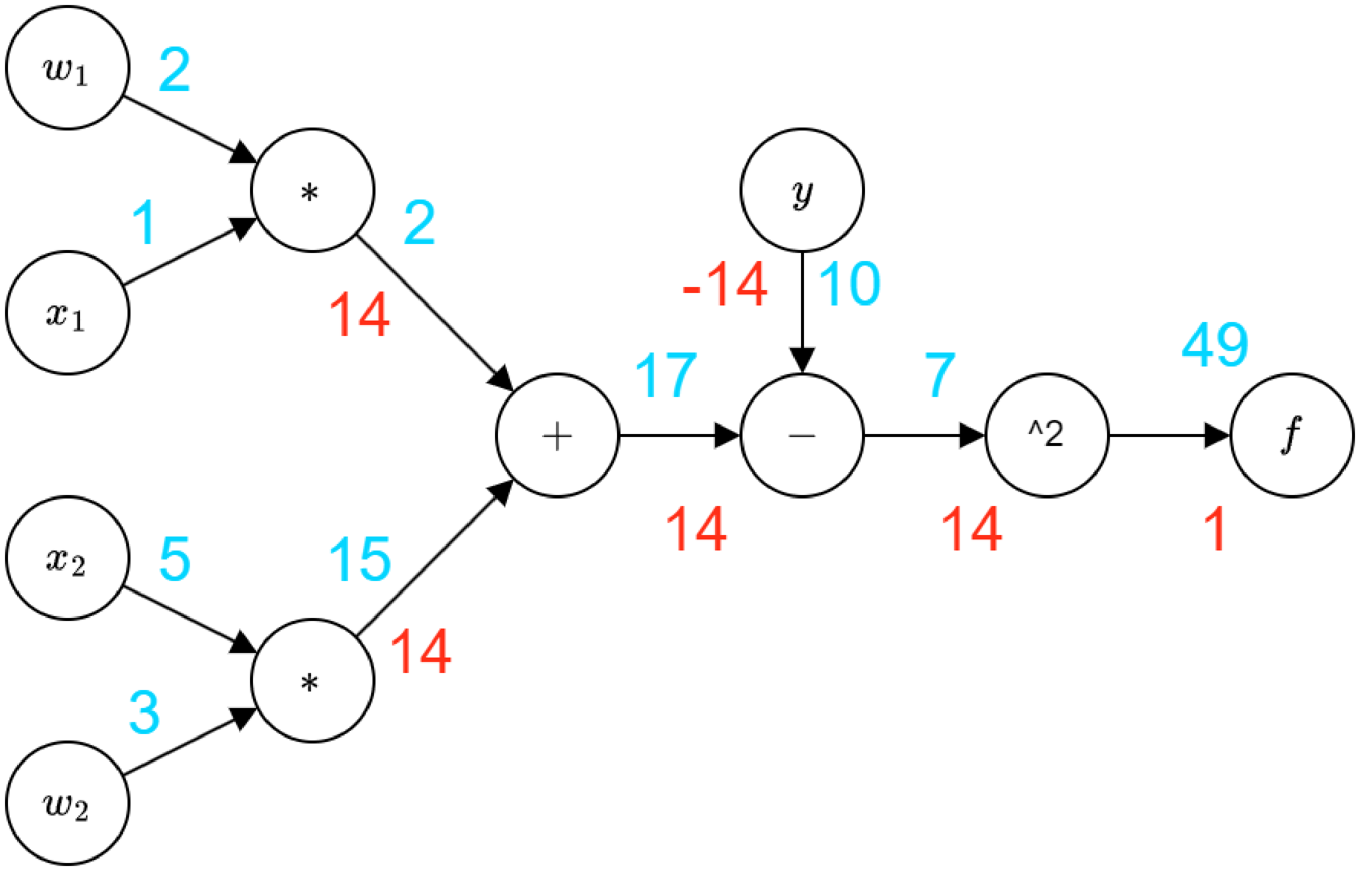
One More Backpropagation Example

- $L(x, y, w) = (w^T x - y)^2$ for 2D input (our favorite regression loss!)



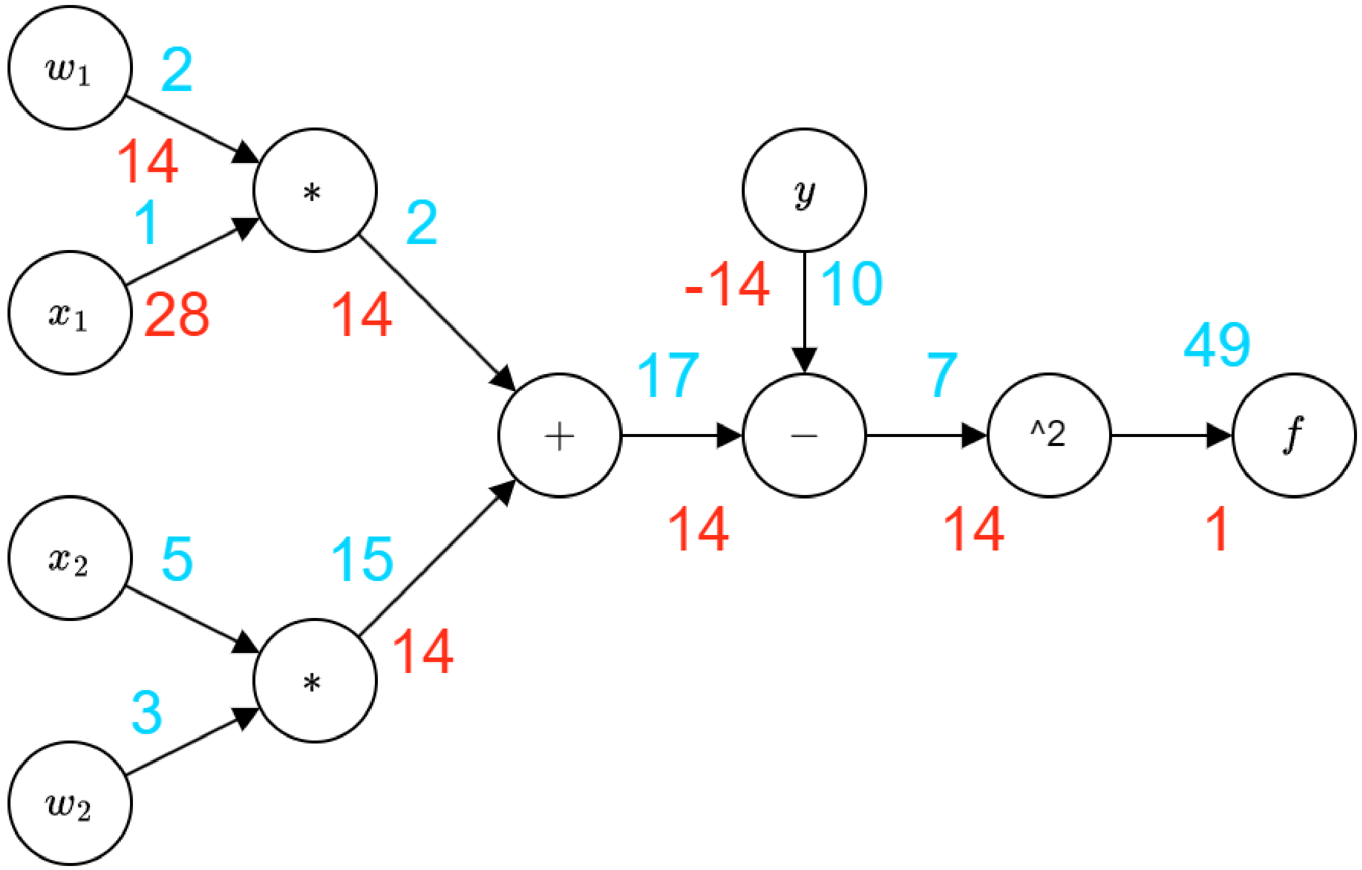
One More Backpropagation Example

- $L(x, y, w) = (w^T x - y)^2$ for 2D input (our favorite regression loss!)



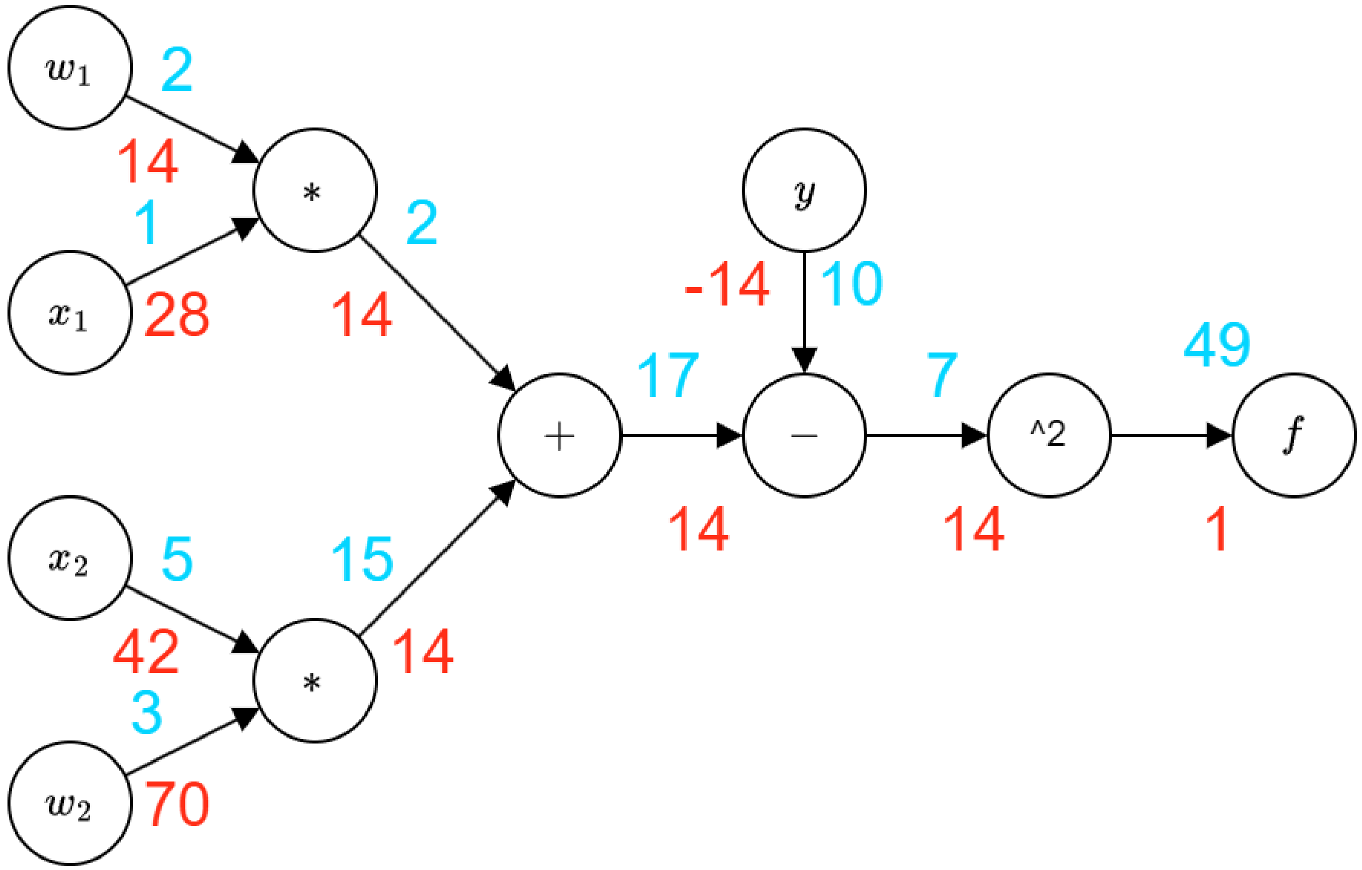
One More Backpropagation Example

- $L(x, y, w) = (w^T x - y)^2$ for 2D input (our favorite regression loss!)



One More Backpropagation Example

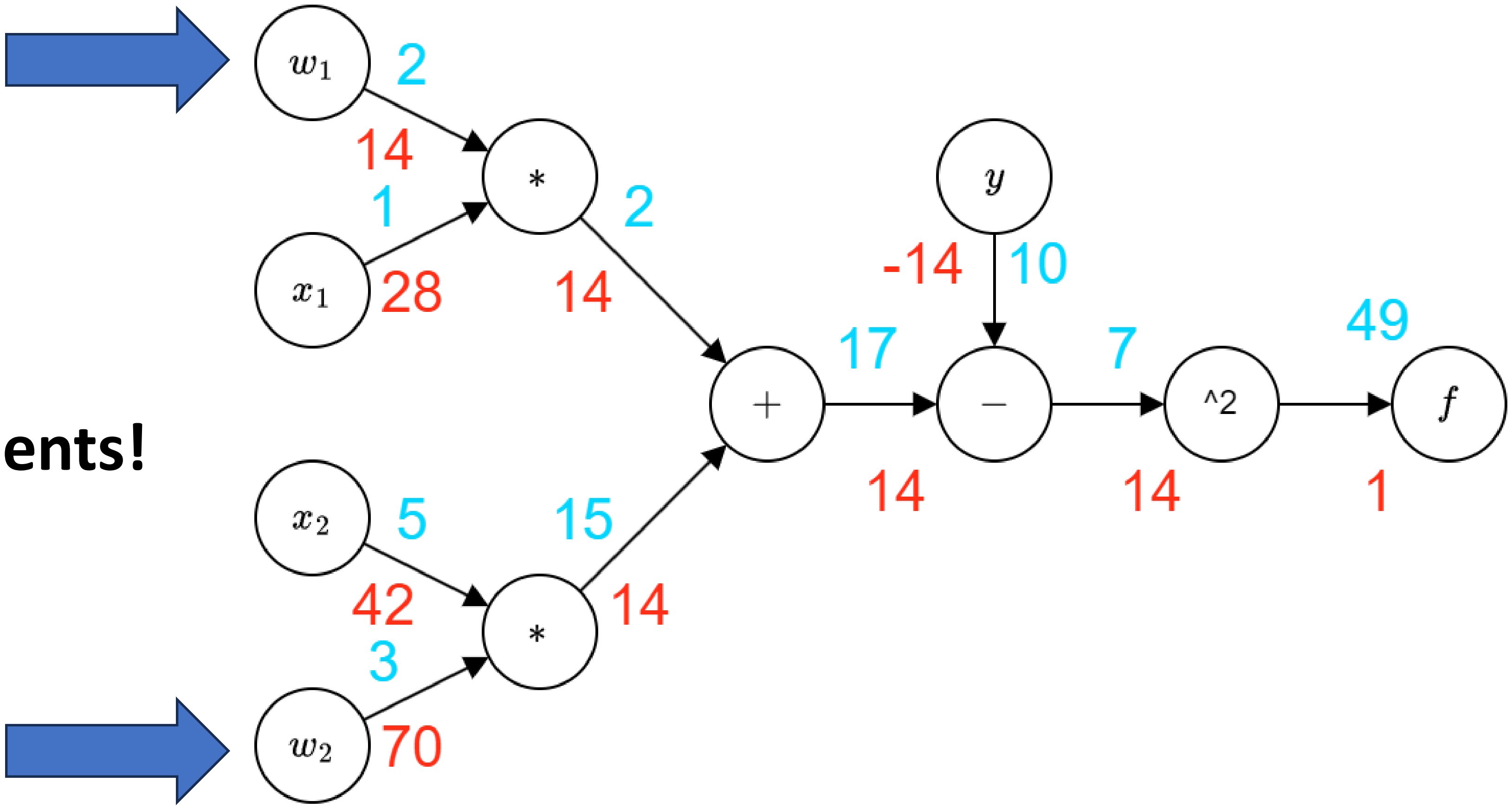
- $L(x, y, w) = (w^T x - y)^2$ for 2D input (our favorite regression loss!)



One More Backpropagation Example

- $L(x, y, w) = (w^T x - y)^2$ for 2D input (our favorite regression loss!)

Loss Gradients!



PyTorch: Training Your Own Neural Network!

- PyTorch is the premiere deep learning library for building and training neural networks
  - Idea: Link *modules* together to form the network
  - Modules specify forward and backward functions
  - Good news: In 99.99% of cases, you do not need to specify your own backward pass!
  - Better news: It takes a single line of code to do the entirety of backpropagation!

- PyTorch is Pythonic and utilizes CUDA to *greatly* accelerate training

- What does training a neural network look like in PyTorch?

PyTorch: Defining a Simple MLP

- We create a new Module class that specifies component Modules
  - 3 linear layers
  - 1 activation function that can be applied multiple times

- We specify how to compute the forward pass for an input
  - Boon: PyTorch constructs the computation graph for us!

```python
import torch
import torch.nn as nn

class SimpleMLP(nn.Module):

    def __init__(self):
        super().__init__() # Required by PyTorch to be the first line!

        self.linear_1 = nn.Linear(8,10) # The first fully-connected layer
        self.linear_2 = nn.Linear(10,7) # The second fully-connected layer
        self.linear_3 = nn.Linear(7,3)  # The third fully-connected layer
        self.activ    = nn.Sigmoid()    # Our choice of activation function


    def forward(self, x):

        x = self.linear_1(x)
        x = self.activ(x)
        x = self.linear_2(x)
        x = self.activ(x)
        x = self.linear_3(x)
        return x
```

PyTorch: Backpropagating a Loss

- We compute a forward pass by calling the network on an input
  - This automatically constructs a computation graph under the hood
- Next, a loss function computes the loss value
- The entire backpropagation is achieved by calling backward()!

```
# Instantiate a model, define # training points and dimensionality
mlp            = SimpleMLP()
num_points     = 10
input_dim      = 8
target_dim     = 3

# Generate some dummy data
random_batched_input  = torch.rand(num_points, input_dim)
random_batched_label  = torch.rand(num_points, target_dim)

# Predict an output with the model and use a loss function
loss_function = nn.MSELoss()
model_output  = mlp(random_batched_input)
loss_value    = loss_function(model_output, random_batched_label)

# The power of backpropagation... in the palm of one method call...
loss_value.backward()
```

PyTorch: Optimizing a Model

- We create an optimizer object that takes the model's parameters as input

- The optimizer applies its own update rule (SGD) based on the backpropagated gradients!

- If we repeatedly do this for several epochs, we can learn the model's parameters!

```python
import torch.optim as optim

sgd_optimizer = optim.SGD(mlp.parameters(), lr=0.01)

epoch_count = 100
for i in range(epoch_count):
    sgd_optimizer.zero_grad()
    model_output  = mlp(random_batched_input)
    loss_value    = loss_function(model_output, random_batched_label)
    loss_value.backward()
    sgd_optimizer.step()
    print(loss_value)
```

Neural Network Architectures

- Now that we can backpropagate for any architecture, what common architectural patterns are being used for NNs?
  - Convolutional Neural Networks
    - Discussed at length in the Computer Vision course
  - Recurrent Neural Networks
    - Discussed at length in the Natural Language Processing course
  - Transformers
    - Discussed everywhere
    - Drives some of the most powerful models today like GPT
      - The “T” stands for transformer!
  - And others…

Convolutional Neural Networks

- Fully connected layers require quite a few weights
  - $d_{i+1} \times d_i$ parameters!

- Do we really need to connect every neuron of one layer with every neuron of the next?
  - We may be able to *tie* some parameters together by instead orienting our connections more locally!
  - Works well for data like images where information is locally related

- We can borrow the convolution (cross correlation) operation in signal processing!

Convolution and Cross Correlation: 1D

- Utilizing a filter g of size $2K+1$, scan over an input sequence f by applying convolution (or cross correlation)

- Convolution

$$(f * g)[n] = \sum_{k=1}^{2K+1} f(n-k)g(k)$$

- Cross Correlation

$$(f \star g)[n] = \sum_{k=1}^{2K+1} f(n+k)g(k)$$

Convolution and Cross Correlation: 2D

- Utilizing a filter g of size $(2K+1) \times (2K+1)$, scan over an input sequence f by applying convolution (or cross correlation)

- 2D Convolution

$$(f * g)[m,n] = \sum_{k_1=1}^{2K+1} \sum_{k_2=1}^{2K+1} f(m-k_1, n-k_2) g(k_1, k_2)$$

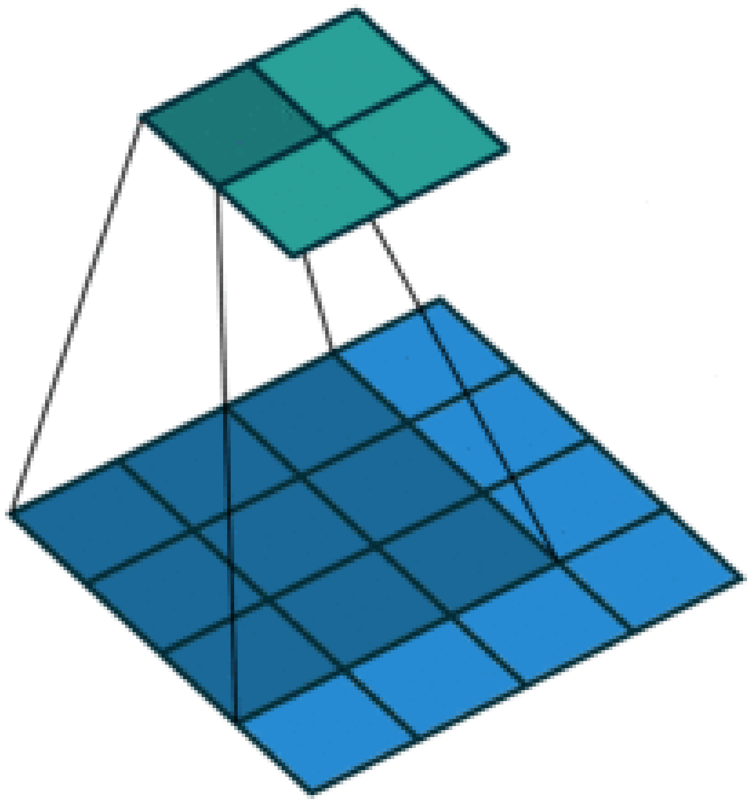
- 2D Cross Correlation

$$(f \star g)[m,n] = \sum_{k_1=1}^{2K+1} \sum_{k_2=1}^{2K+1} f(m+k_1, n+k_2) g(k_1, k_2)$$

A 2D Example

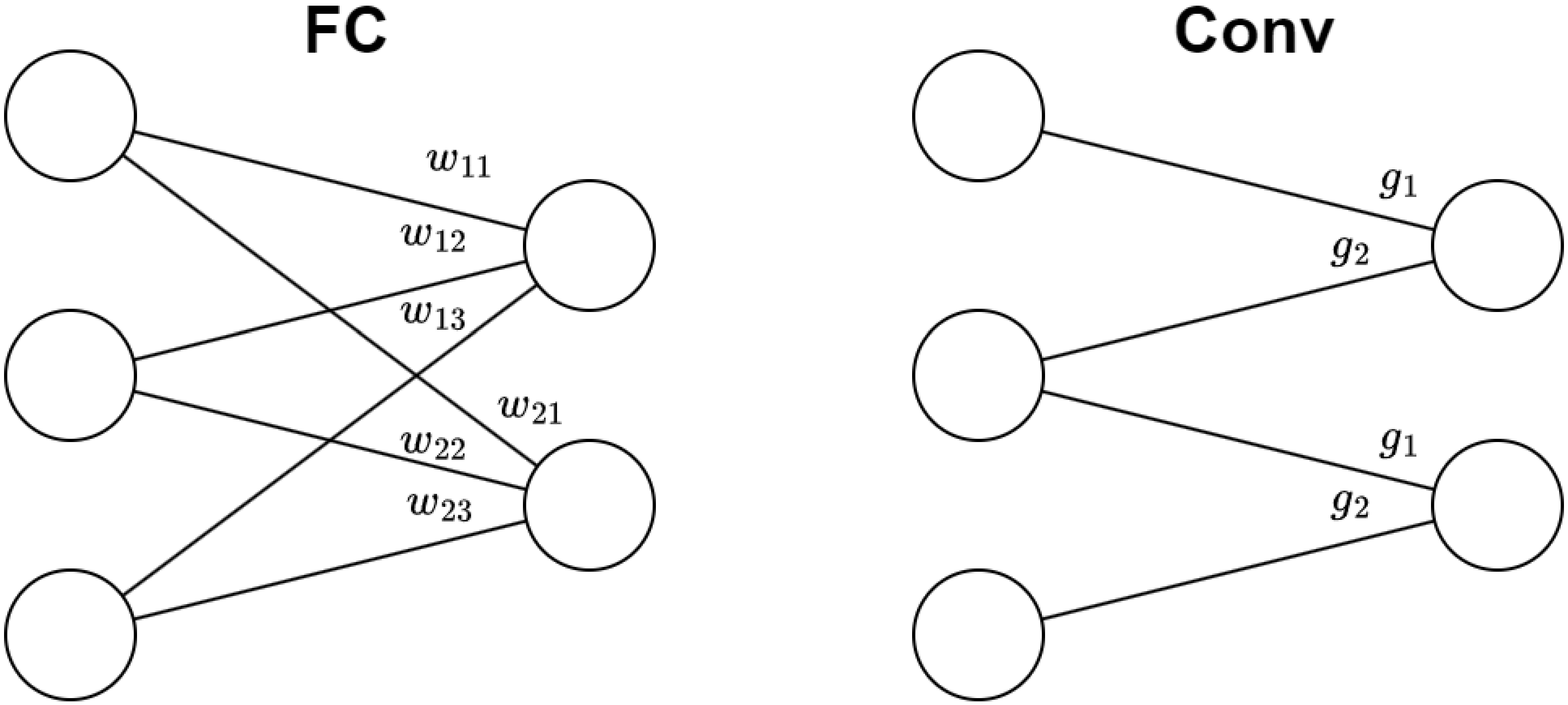
- Input: Blue
- Output: Turquoise

- Each position in the output is obtained by taking a dot product with the filter and its overlapping area on the input map



https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md

Fully-Connected Comparison: 1D



- 1D convolution with filter size 2
  - Convolutional operation ties weights and uses fewer connections!

Closing Notes on Convolution

- Most networks run the convolution operation for *multiple filters* across multiple *channels*

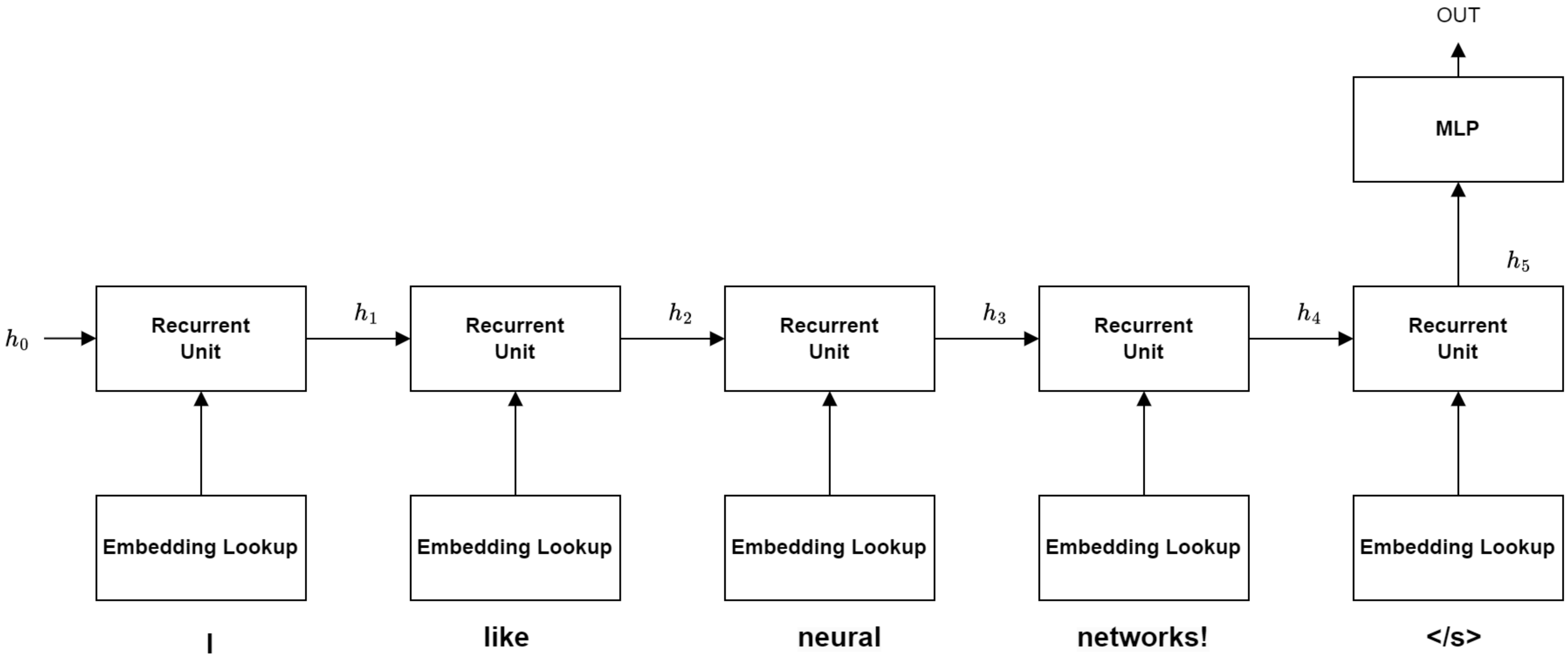
$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

- See PyTorch's docs for more details!
  - https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

Recurrent Neural Networks

- Q: How do we handle sequences as inputs?
  - Prime example: Sentences!
- We need an architecture that can process sequences of variable length
- Idea: Convert elements of sequence into feature embeddings
  - Sequence classification: Condense embeddings into one vector and use MLP
  - Per-element classification: Process embeddings and use MLP on each
- We ALSO need to effectively encode properties of different elements of the sequence and their relative positions
  - Another idea: Process sequence in order by keeping track of a *hidden state*

Recurrent Neural Networks



Recurrent Neural Networks

- What are some common recurrent units?
  - Long Short-Term Memory (LSTM)
  - Gated Recurrent Unit (GRU)
  - …

- RNNs greatly improve performance on sequence tasks over simpler models
  - But they can suffer from long-range dependency issues
  - Ex: “I don’t like this movie, but the scene where the lead managed to outrun all those bandits was interesting.”
  - If we are classifying sentiment, the model should predict negative sentiment; however, RNNs focus more on the last part and predict positive sentiment!

Transformers and Attention

- Observation: People typically pay attention to specific parts of the sequence when making decisions
  - “I don’t like this movie, but the scene where the lead managed to outrun all those bandits was interesting”

- How can we model attention?
  - Based on some query that we have, we attempt to find a matching key within the sequence and get its corresponding value
  - Ex: We know to look for negative phrases (query), find “I don’t like this movie” (key), and retrieve it (also the value).

Transformers and Attention

- We can model the query-key lookup by measuring similarity between a query vector and a key vector
  - Common approach: Dot-product
    - Maximized when q and k align
    - 0 when q and k are orthogonal
    - Minimized q and k are opposite directions

$$q \cdot k$$

- If we have a single query and multiple key-value pairs, we can derive a weighted average of the values based on the query-key similarities:

$$\sum_i^n (q^T k_i) v_i$$

Transformers and Attention

- If we have a single query and multiple key-value pairs, we can derive a weighted average of the values based on the query-key similarities:

$$\sum_i^n (q^T k_i) v_i$$

- We can extend this to a vectorized form for multiple queries:

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

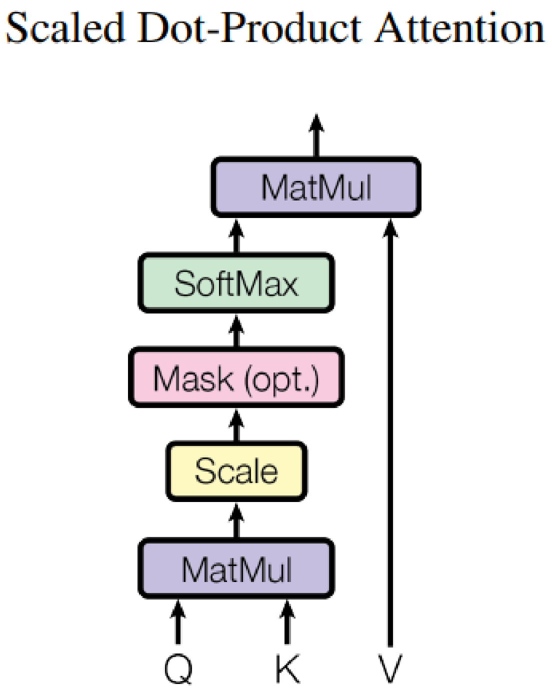
- Q, K, V are $m \times d_k$, $n \times d_k$, $n \times d_v$ matrices, and the weighting factors are scaled by softmax and the dimension of the keys

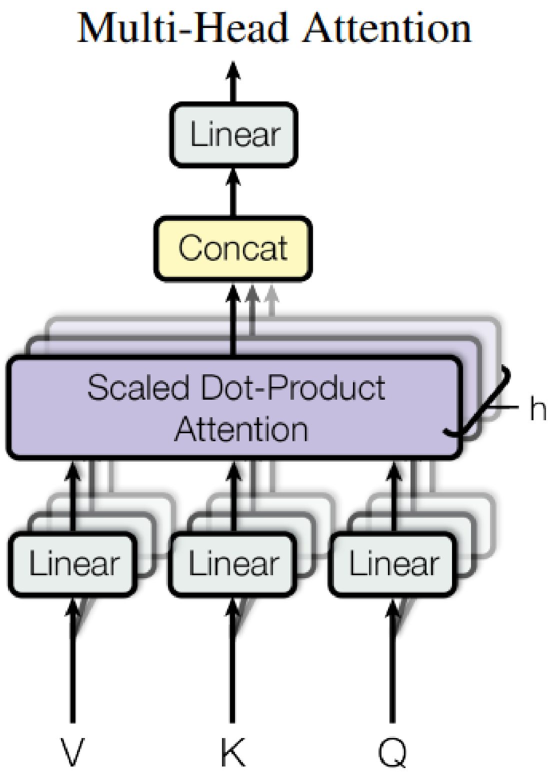
Attention Is All You Need. Vaswani et al., 2017.

Multi-Head Attention

- Project inputs to different feature spaces to pay attention to different kinds of information

- This corresponds to a bunch of attentions per query, key, and value combination





[*Attention Is All You Need.* Vaswani et al., 2017.]()

The Transformer (Seq2Seq)

- Sequence-to-sequence: Convert from one sequence to another
  - Common architecture: Encoder-Decoder

- Stack a bunch of attention + FC layers together to encode the input sequence and decode an output sequence!
  - Functionality depends on what we assign to be the queries, keys, and values.
  - Self-Attention: All queries, keys, and values are the same.
  - Encoder-Decoder Attention: Queries come from output of decoder layers; keys and values come from the output of the encoder

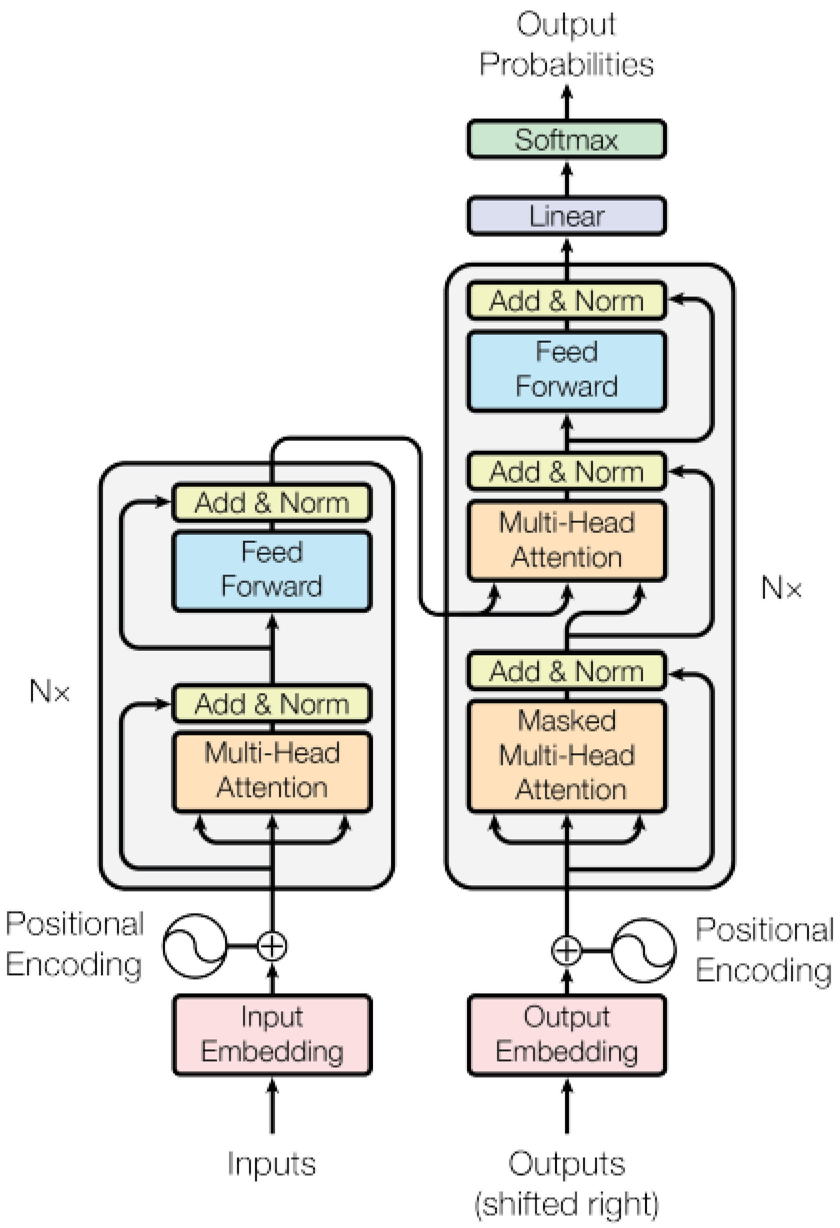


Figure 1: The Transformer - model architecture.

[*Attention Is All You Need.* Vaswani et al., 2017.]