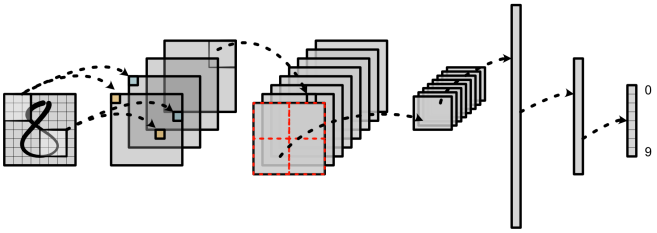
MiniTorch

MiniTorch is a *diy teaching library* for machine learning engineers who wish to learn about the internal concepts underlying deep learning systems. It is a pure Python re-implementation of the Torch API designed to be simple, easy-to-read, tested, and incremental. The final library can run Torch code.



The main repo for the course is available on GitHub. To complete the assignments you fill in TODO statements and pass the included unit test suite. There are also additional visualization tools to follow your progress.

```
class ReLU(ScalarFunction):

    @staticmethod
    def forward(ctx, a):
        # TODO: Implement for Task 1.2.
        raise NotImplementedError('Need to implement for Task 1.2')

    @staticmethod
    def backward(ctx, d_output):
        # TODO: Implement for Task 1.4.
        raise NotImplementedError('Need to implement for Task 1.4')
```

Individual assignments cover:

- ML Programming Foundations
- Autodifferentiation
- Tensors
- GPUs and Parallel Programming
- Foundational Deep Learning

The project was developed for the course *Machine Learning Engineering* at Cornell Tech and based on my experiences working at Hugging Face. Reach out if you are interested in the teacher's version of the repository.

Enjoy!

Sasha Rush (@srush_nlp) with Ge Gao, Anton Abilov, and Aaron Gokaslan.

Setup

MiniTorch requires Python 3.11 or higher. To check your version of Python, run either:

```
>>> python --version
>>> python3 --version
```

We recommend creating a global MiniTorch workspace directory that you will use for all modules.

```
>>> mkdir workspace; cd workspace
```

We also highly recommand setting up a *virtual environment*. The virtual environment lets you install packages that are only used for your assignments and do not impact the rest of the system. We suggest venv or anaconda.

For example, if you choose venv, run the following command:

```
>>> python -m venv .venv
>>> source .venv/bin/activate
```

The first line should be run only once, whereas the second needs to be run whenever you open a new terminal to get started for the class. You can tell if the second line works by checking if your terminal starts with `(venv)`. See https://docs.python.org/3/library/venv.html for further instructions on how this works.

Each assignment is distributed through a Git repo. We assume the knowledge of git throughout the course. See https://guides.github.com for a tutorial about using git and GitHub.

You should fork the template of the assignment and then edit yours in your forked repo. Once you have forked the template code, you can clone your own version by running the following command:

```
>>> git clone {{ASSIGNMENT}}
>>> cd {{ASSIGNMENT}}
```

The last step is to install packages. There are several packages used throughout these assignments, and you can install them in your virtual enviroment by running:

```
>>> python -m pip install -r requirements.txt
>>> python -m pip install -r requirements.extra.txt
```

```
>>> python -m pip install -Ue .
```

For anaconda users, you need to run an extra command to install llvmlite:

```
>>> conda install llvmlite
```

Make sure that everything is installed by running `python` and then checking:

```
>>> import minitorch
```

Fundamentals

This introductory module is focused on introducing several core software engineering methods for testing and debugging, and also includes some basic mathematical foundations.

Before starting this assignment, make sure to set up your workspace following the setup guide, to understand how the code should be organized.

Guides

Each module has a set of guides to help with the background material. We recommend working through the assignment and utilizing the guides suggested for each task.

- Contributing
- Functional Python
- Property Testing
- Modules
- Visualization

In addition to completing each of the tasks below, you need to ensure that all the unit tests and style checks pass in the module. This may require writing docstrings or adding types to functions.

Task 0.1: Operators

This task is designed to help you get comfortable with style checking and testing. We ask you to implement a series of basic mathematical functions. These functions are simple, but they form the basis of MiniTorch. Make sure that you understand each of them as some terminologies might be new.

Todo

Complete the following functions in `minitorch/operators.py` and pass tests marked as `task0_1`.

- `mul` - Multiplies two numbers
- `id` - Returns the input unchanged
- `add` - Adds two numbers
- `neg` - Negates a number
- `lt` - Checks if one number is less than another
- `eq` - Checks if two numbers are equal
- `max` - Returns the larger of two numbers
- `is_close` - Checks if two numbers are close in value
- `sigmoid` - Calculates the sigmoid function
- `relu` - Applies the ReLU activation function
- `log` - Calculates the natural logarithm
- `exp` - Calculates the exponential function
- `inv` - Calculates the reciprocal
- `log_back` - Computes the derivative of log times a second arg
- `inv_back` - Computes the derivative of reciprocal times a second arg
- `relu_back` - Computes the derivative of ReLU times a second arg

Task 0.2: Testing and Debugging

We ask you to implement property tests for your operators from Task 0.1. These tests should ensure that your functions not only work but also obey high-level mathematical properties for any input. Note that you need to change arguments for those test functions.

Todo

Complete the test functions in `tests/test_operators.py` marked as `task0_2`.

Task 0.3: Functional Python

To practice the use of higher-order functions in Python, implement three basic functional concepts. Use them in combination with operators described in Task 0.1 to build up more complex mathematical operations that work on lists instead of single values.

Todo

Complete the following functions in `minitorch/operators.py` and pass tests marked as `tasks0_3`.

- `map` - Higher-order function that applies a given function to each element of an iterable
- `zipWith` - Higher-order function that combines elements from two iterables using a given function
- `reduce` - Higher-order function that reduces an iterable to a single value using a given function

Using the above functions, implement:

- `negList` - Negate all elements in a list using `map`
- `addLists` - Add corresponding elements from two lists using `zipWith`
- `sum` - Sum all elements in a list using `reduce`
- `prod` - Calculate the product of all elements in a list using `reduce`

Task 0.4: Modules

This task is to implement the core structure of the :class:`minitorch.Module` class. We ask you to implement a tree data structure that stores named :class:`minitorch.Parameter` on each node. Such a data structure makes it easy for users to create trees that can be `walked` to find all of the parameters of interest.

To experiment with the system use the `Module Sandbox`:

```
>>> streamlit run app.py -- 0
```

Todo

Complete the functions in `minitorch/module.py` and pass tests marked as `tasks0_4`.

`minitorch.Module.train() -> None`

Set the mode of this module and all descendent modules to `train`.

`minitorch.Module.eval() -> None`

Set the mode of this module and all descendent modules to `eval`.

`minitorch.Module.named_parameters() -> Sequence[Tuple[str, Parameter]]`

Collect all the parameters of this module and its descendents.

Returns

```
The name and `Parameter` of each ancestor parameter.
```

`minitorch.Module.parameters() -> Sequence[Parameter]`

Enumerate over all the parameters of this module and its descendents.

Task 0.5: Visualization

For the first few assignments, we use a set of datasets implemented in `minitorch/datasets.py`, which are 2D point classification datasets. (See TensorFlow Playground for similar examples.) Each of these dataset can be added to the visualization.

To experiment with the system use:

```
streamlit run project/app.py -- 0
```

Read through the code in `project/run_torch.py` to get a sneak peek of an implementation of a model for these datasets using Torch.

You can also provide a model that attempts to perform the classification by manipulating the parameters.

Parameters

Parameter: linear.weight_0_0

-10.00 10.00

Parameter: linear.weight_1_0

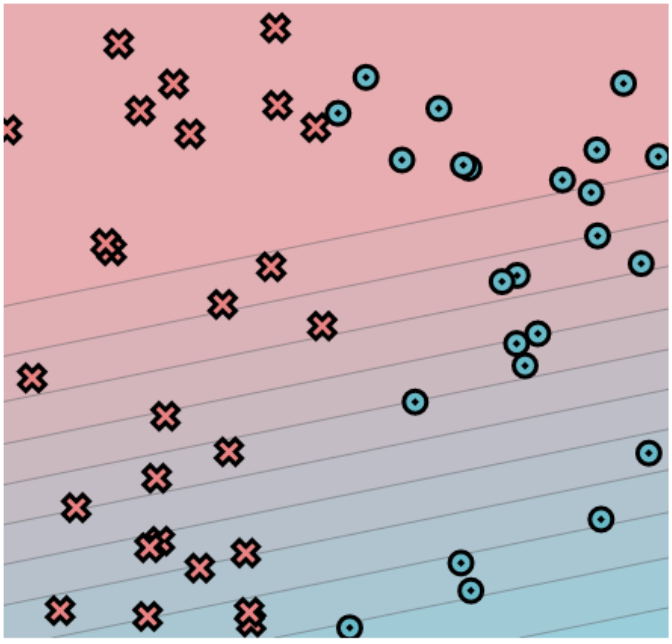
-10.00 10.00

Parameter: linear.bias_0

-10.00 10.00

☐ Show X-Axis Only (For Simple)

Initial setting



Todo

Add docstrings for all the different datasets required for this part.

Start a streamlit server and print an image of the dataset. Hand-create classifiers that split the linear dataset into the correct colors.

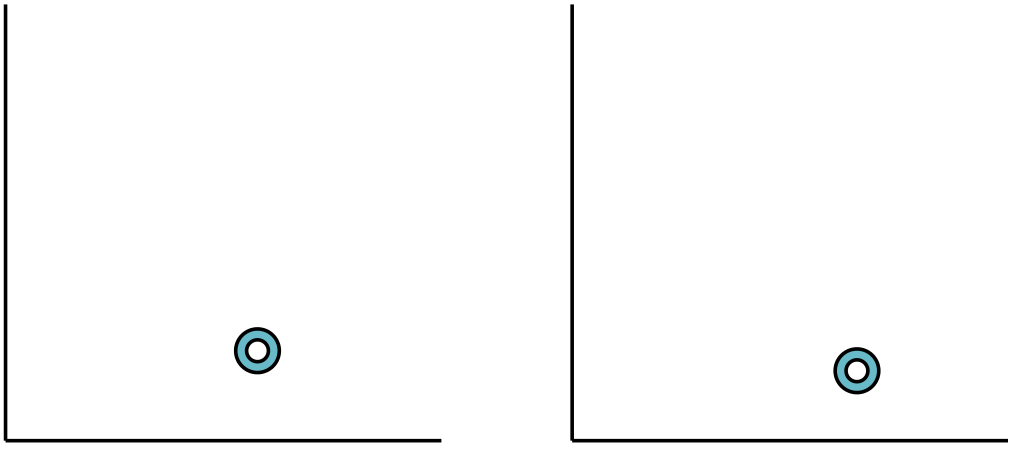
Add the image in the README file in your repo along with the parameters that your used.

ML Primer

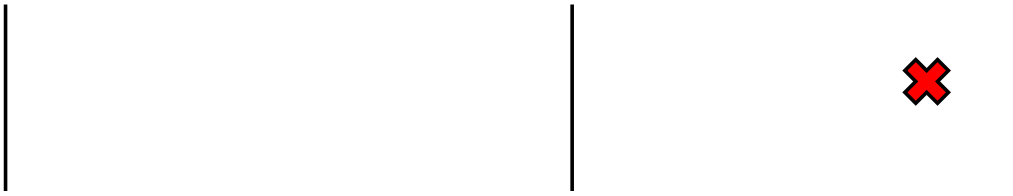
This guide is a primer on the very basics of machine learning that are necessary to complete the assignments and motivate the final system. Machine learning is a rich and well-developed field with many different models, goals, and learning settings. There are many great texts that cover all the aspects of the area in detail. This guide is not that. Our goal is to explain the minimal details of *one* dataset with *one* class of model. Specifically, this is an introduction to supervised binary classification with neural networks. The goal of this section is to learn how a basic neural network works to classify simple points.

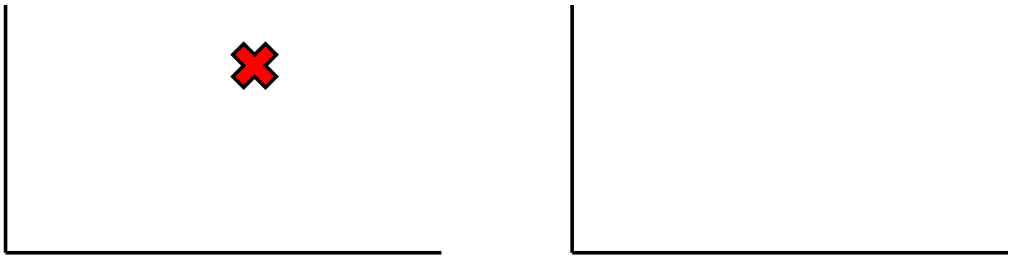
Dataset

Supervised learning problems begin with a labeled `training` dataset. We assume that we are given a set of labeled points. Each point has two coordinates x_1 and x_2, and has a label y corresponding to an O or X. For instance, here is one O labeled point:

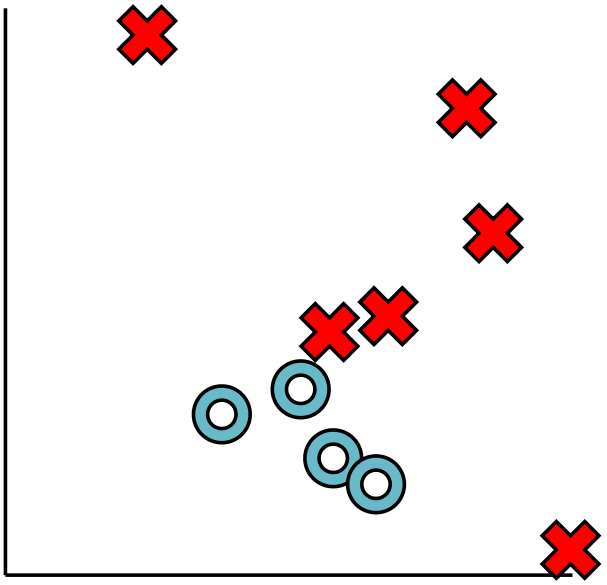


And here is an X labeled point.

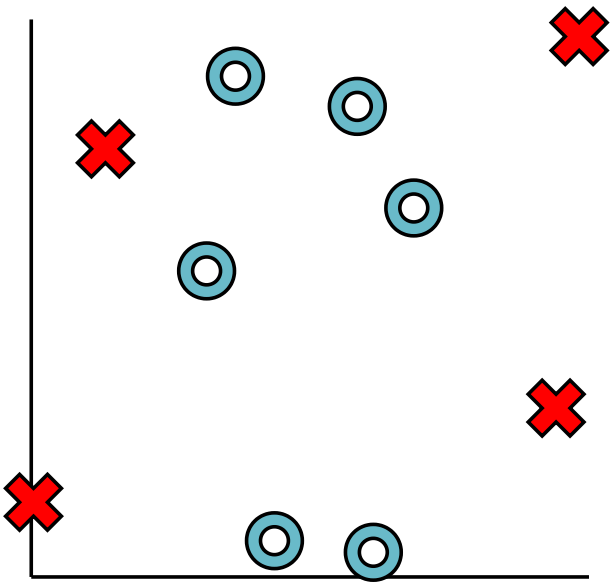




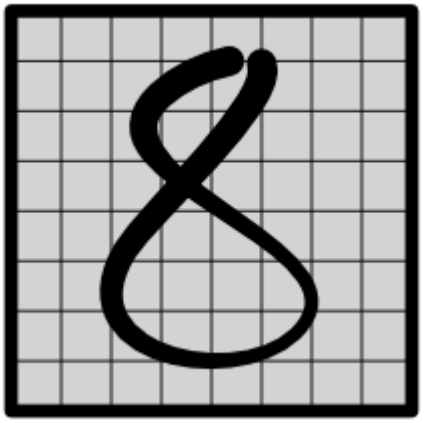
It is often convenient to plot all of the points together on one set of axes.



Here we can see that all the X points are in the top-right and all the O points are on the bottom-left. Not all datasets is this simple, and here is another dataset where points are split up a bit more.



Later in the class, we will consider datasets of different forms, e.g. a dataset of handwritten numbers, where some are 8's and others are 2's:



Here is an example of what this dataset looks like.

Model

Our ML system needs to specify a model that we want to the data. A model is a function that assigns labels to data points. We can specify a model in Python through its parameters and function.

```python
@dataclass
class Linear:
    # Parameters
    w1: float
    w2: float
    b: float

    def forward(self, x1: float, x2: float) -> float:
        return self.w1 * x1 + self.w2 * x2 + self.b
```

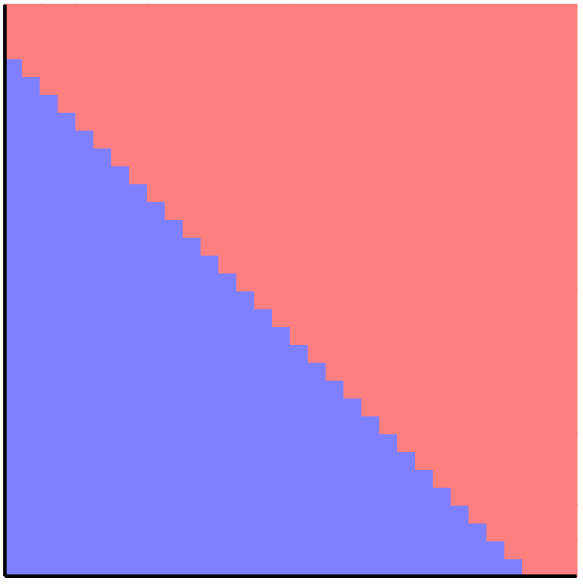
This model can be written mathematically as,

$$m(x_1, x_2; w_1, w_2, b) = x_1 \times w_1 + x_2 \times w_2 + b$$

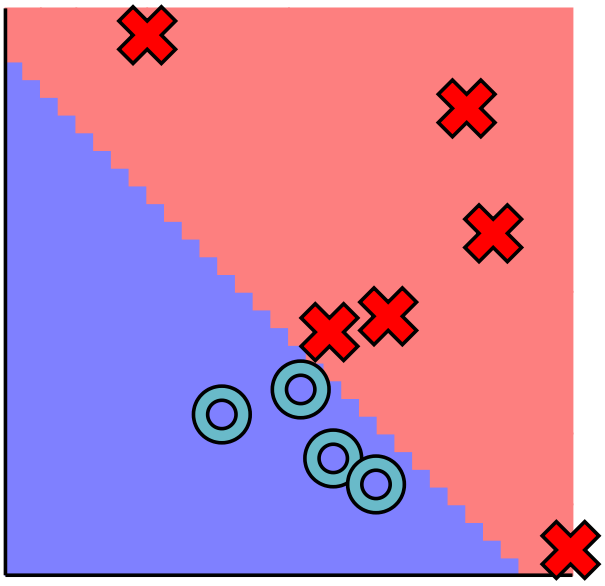
.

We call it a linear model because it divides the data points up based on a line. We can visualize this be computing the "decision boundary", i.e. the areas where this function returns a positive and negative boundary.

```
model = Linear(1, 1, -0.9)
```



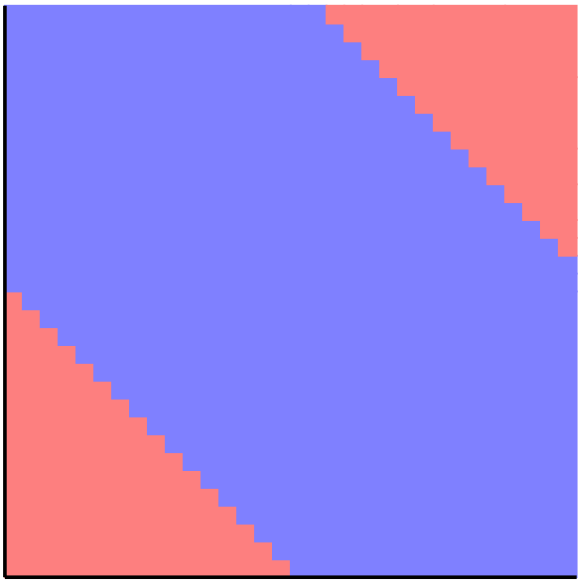
We can overlay the simple dataset described earlier over this model. This tells us roughly how well the model fits this dataset.



Models can take many different forms, Here is another model which has a compound form. We will discuss these types of models more below. It splits its decision into three regions (Model B).

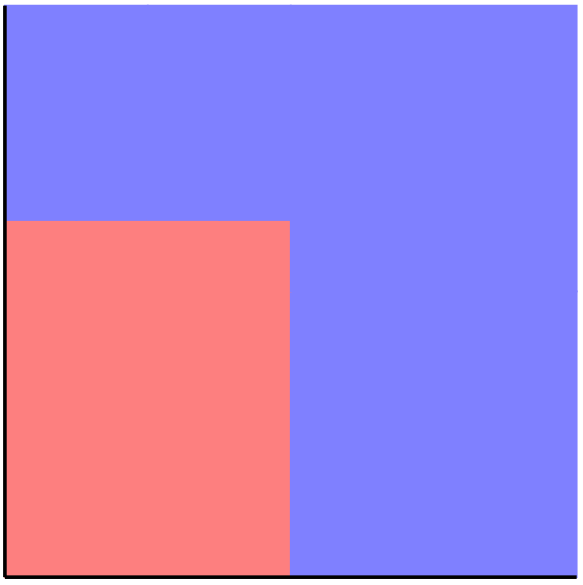
```
@dataclass
class Split:
    m1: Linear
    m2: Linear

    def forward(self, x1, x2):
        return self.m1.forward(x1, x2) * self.m2.forward(x1,
x2)
```

```
model_b = Split(Linear(1, 1, -1.5), Linear(1, 1, -0.5))
```



Models may also have strange shapes and even disconnected regions. Any blue/red split will do, for instance (Model C):

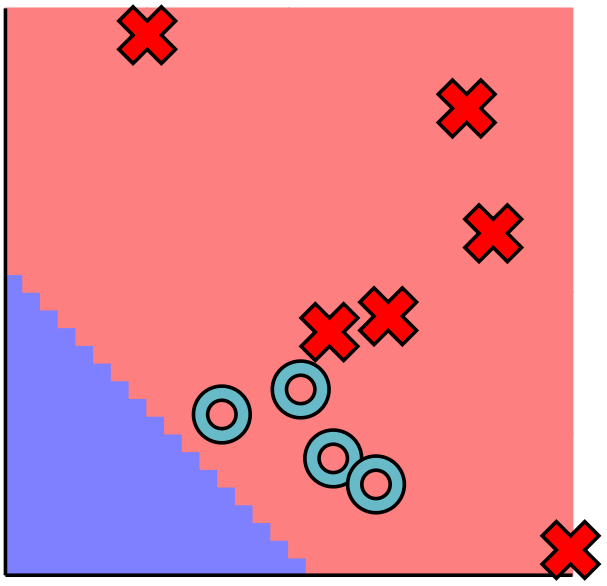
```
@dataclass
class Part:
    def forward(self, x1, x2):
        return 1 if (0.0 <= x1 < 0.5 and 0.0 <= x2 < 0.6) else
0
```



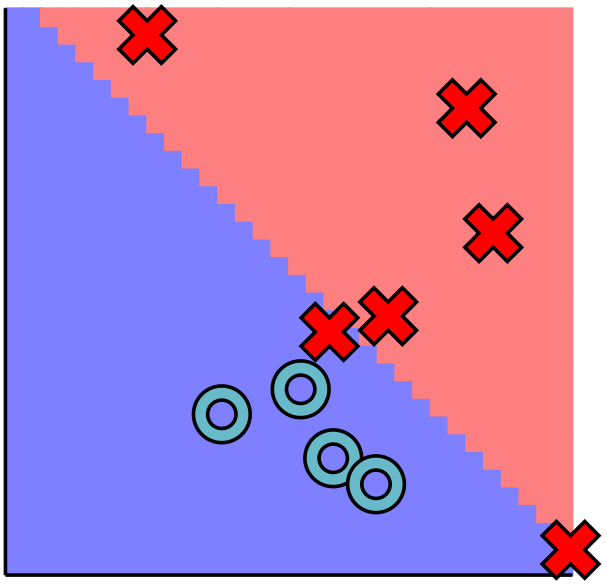
Parameters

Once we have decided on the shape that we are using, we need a way to move between models in that class. Ideally, we would have internal knobs that alter the properties of the model.

```
show(Linear(1, 1, -0.5))
```

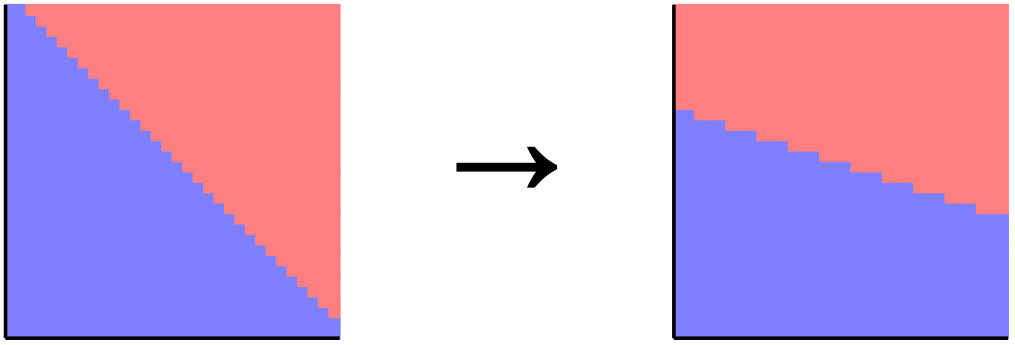


```
show(Linear(1, 1, -1))
```

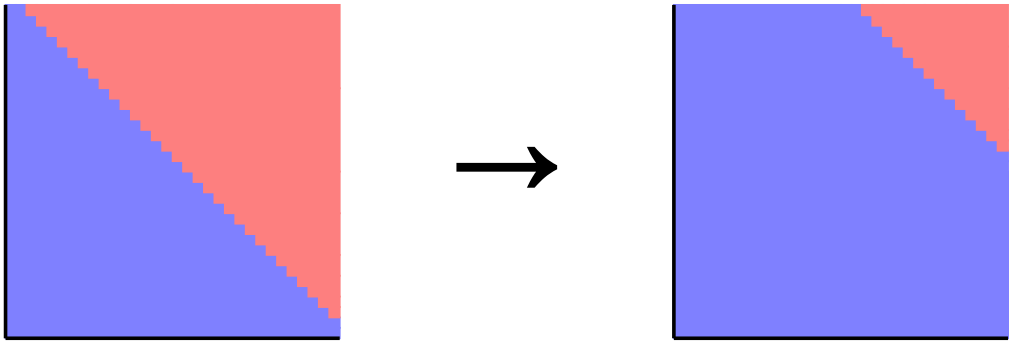


In the case of the linear models, there are two knobs,

a. rotating the separator



b. changing the separator cutoff



Parameters are the set of numerical values that fully define a model's decisions. Parameters are critical for storing how a model acts, and necessary for producing its

decision on a given data point.

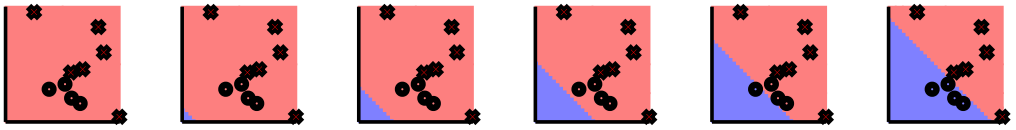
Recall the functional form of the model is,

$$m(x_1, x_2; w_1, w_2, b) = x_1 \times w_1 + x_2 \times w_2 + b$$

Here w_1, w_2, b are parameters, x_1, x_2 are the input point. The semi-colon notation indicates which arguments are for parameters and which are for data.

Our goal in this class will be to move these knobs to find the best data fit.

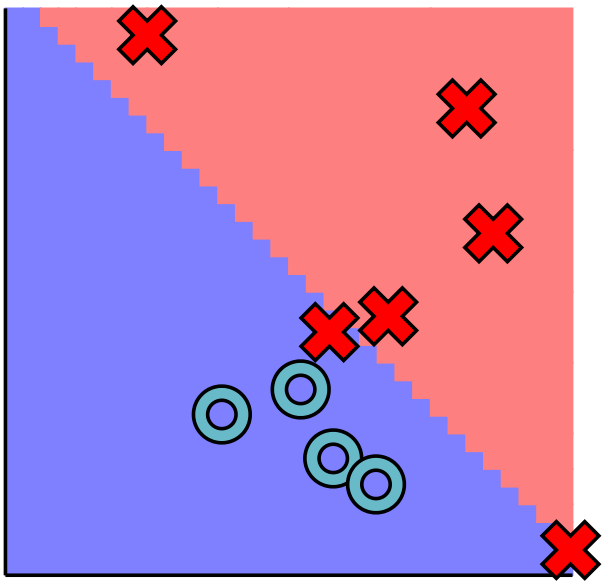
```
biases = [(i / 25.0) - 0.1 for i in range(0, 26, 5)]
```



Loss

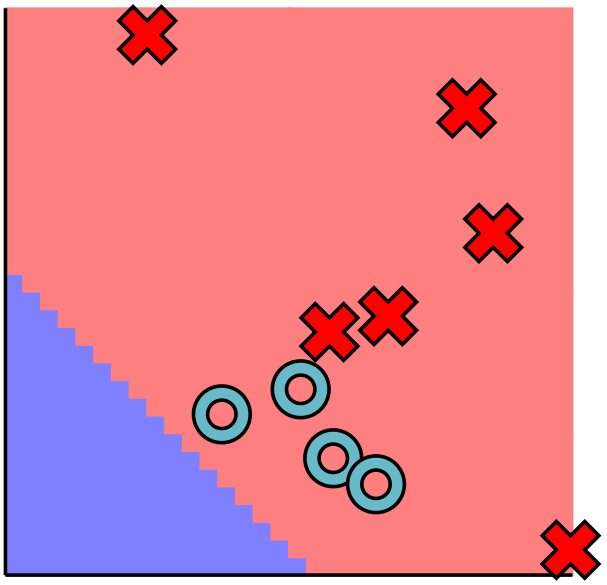
Observing the data, we can see that some parameters lead to good models with few classification errors,

```
show(Linear(1, 1, -1.0))
```



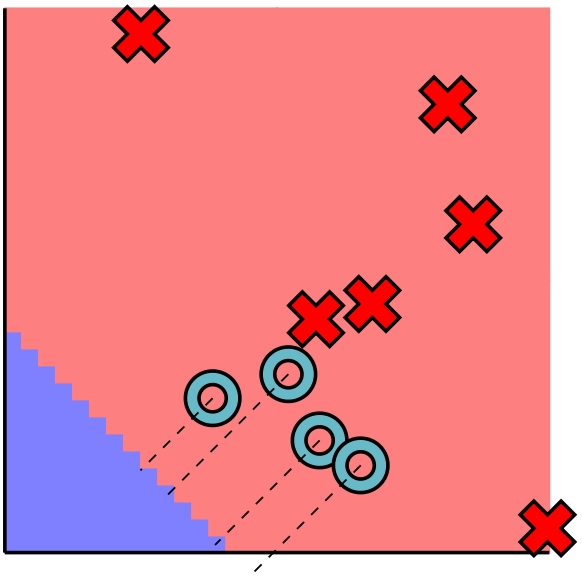
And some are bad and make multiple errors,

```
show(Linear(1, 1, -0.5))
```



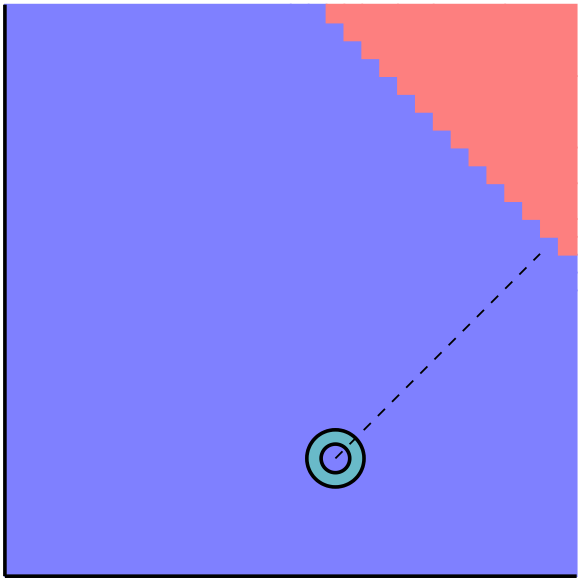
In order to find a good model, we need to first define what *good* means. We do this through a `loss` function that quantifies how badly we are currently doing. A good model has small loss.

Our loss function will be based on the distance and direction of the line from each point to the decision boundary.

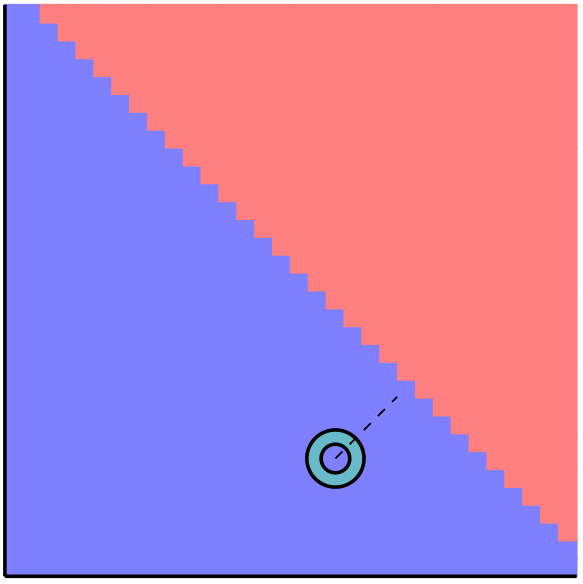


Consider a single point with different models.

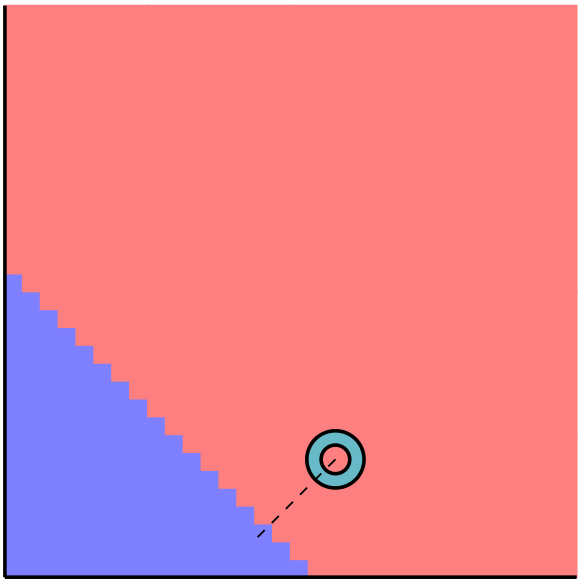
This point might be classified the correct side and very far from this line (Point A, "great"):



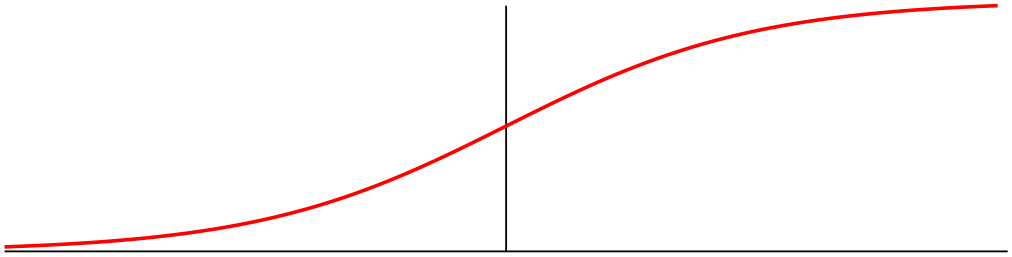
Or it might be on the correct side of the line, but close to the line (Point B, "worrisome"):



Or this point might be classified on the wrong side of the line (Point C, "bad"):

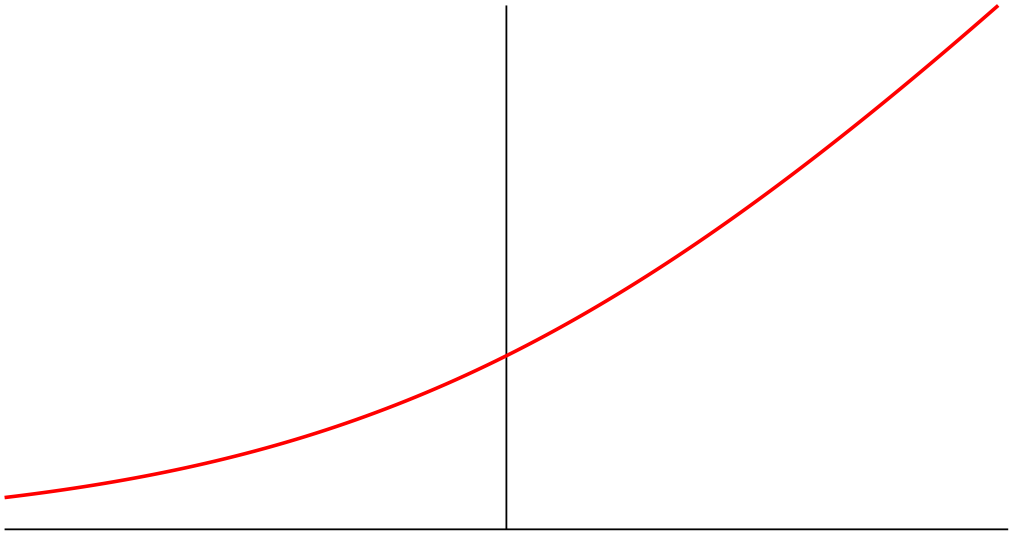


The loss is determined based on a function of this distance. The most commonly used function (and the one we will focus on) is the *sigmoid* function. For strong negative inputs, it goes to zero, and for strong positive, it goes to 1. In between, it forms a smooth S-curve.

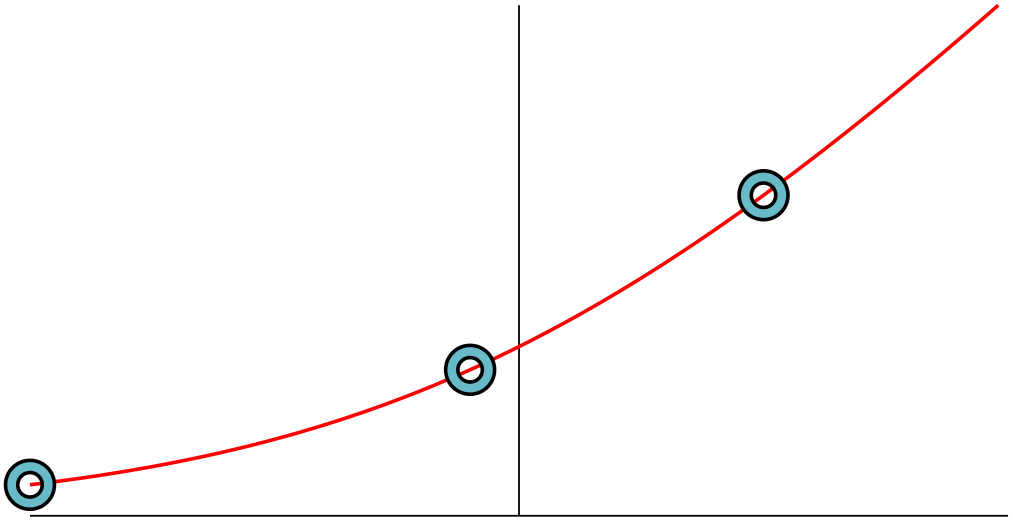


For computational reasons, in practice we work with the log of this function. This yields a loss function that gets much worse as we move further from the decision boundary.

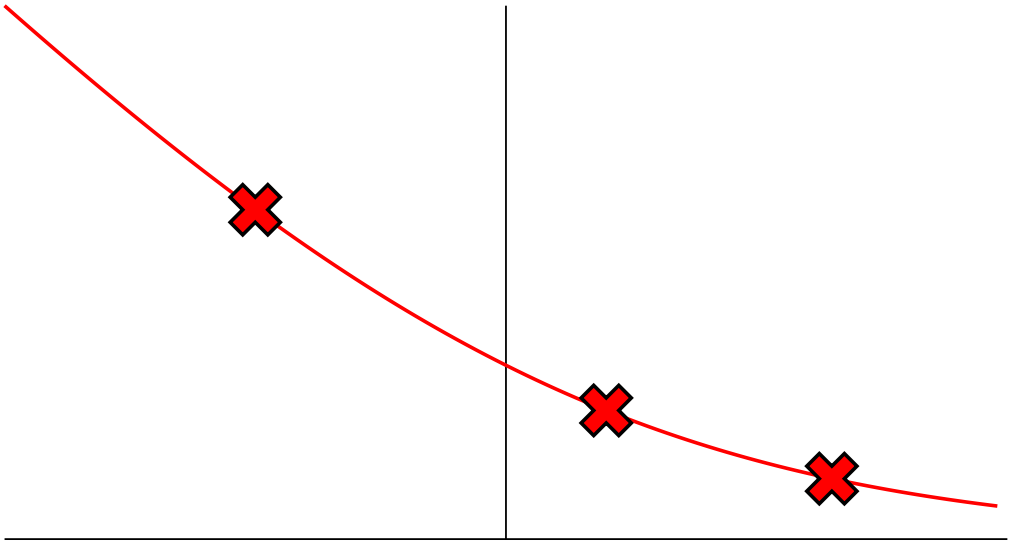
```
def point_loss(x):
    return -math.log(minitorch.operators.sigmoid(-x))
```



The losses of three X points land on the following positions for the sigmoid curve. Almost zero for Point A, middle value for Point B, and nearly one for Point C.



Loss is given for the red points as well, but they are penalized in the opposite direction,



The total loss function L for a model is the sum of each of the individual losses. Specifically,

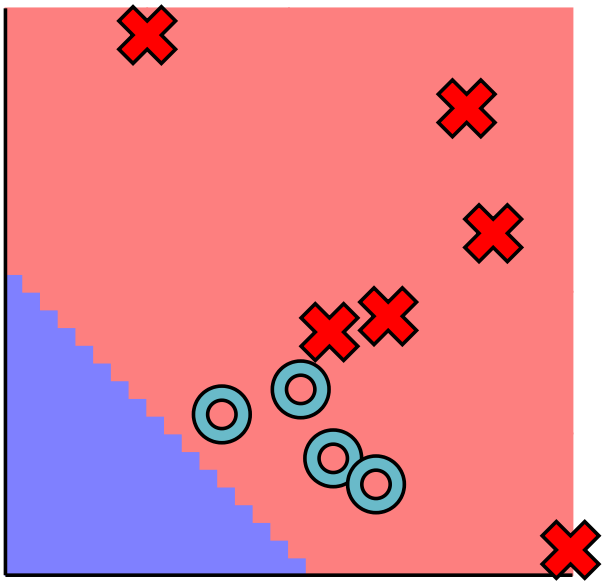
$$L(w_1, w_2, b) = -\sum_j \log \sigma(y^j \times m(x_1^j, x_2^j; w_1, w_2, b))$$

Where (x^j, y^j) are the datapoints, σ is the sigmoid function, and multiplying by y reverses the function based on the true class of the point. Here is what this looks like in code.

```
def full_loss(m):
    l = 0
    for x, y in zip(s.X, s.y):
        l += point_loss(-y * m.forward(*x))
    return -l


# -

# Fitting Parameters
# --------------------

# To review, the model class tells us what shapes we can
consider, the parameters
# tell us the decision boundary, and the loss tells us how well
the current model is doing.
#
# The last step is to produce a method for finding a good model
# given a loss function, referred to as *parameter fitting*.

# Exact parameter fitting is difficult. For all but the
```

```
# simplest models, it is a challenging task.
# This example has just 3 parameters, but some large models may
have billions of parameters that need to be fit.

# We will focus on parameter fitting with *gradient
# descent*. Gradient descent works in the following manner.

# 1. Compute the loss function, $L$, for the data with the
current parameters.
# 2. See how small changes to each of the parameters would
change the loss.
# 3. Update the parameters with a small change in the direction
that locally
#    most reduces the loss.

# Let's return to the incorrect model above.
```

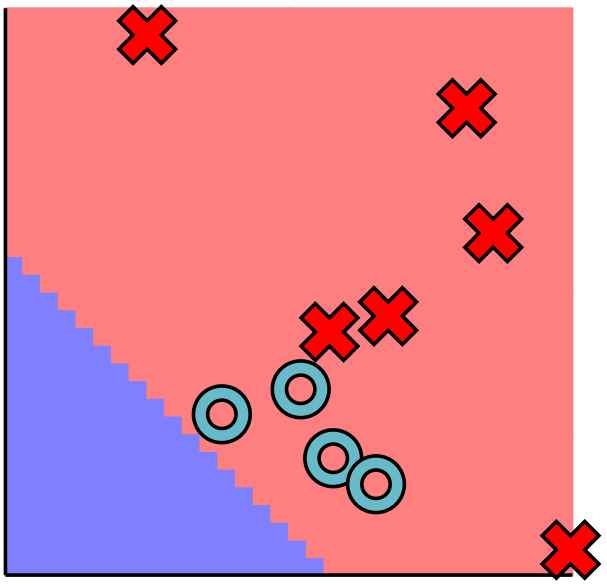
```
m = Linear(1, 1, -0.5)
```

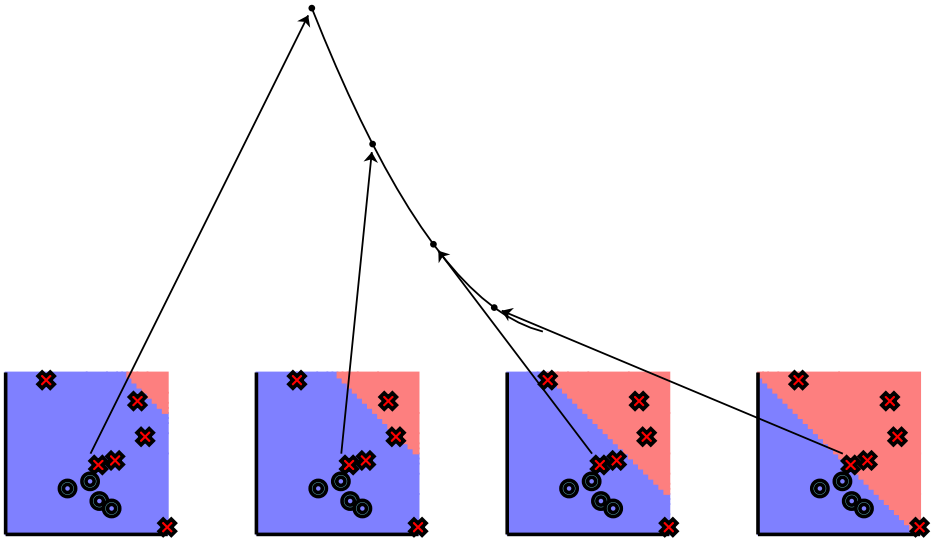


As we noted, this model has a high loss, and we want to consider ways to "turn the knobs" of the parameters to find a better model. Let us focus on the parameter controlling the intercept.

We can consider how the loss changes with respect to just varying this parameter. It seems like the loss will go down if we move the intercept a bit.

```
m = Linear(1, 1, -0.55)
```





Doing this leads to a better model.

```
chalk.set_svg_height(200)
```

We can repeat this process for the intercept as well as for all the other parameters in the model.

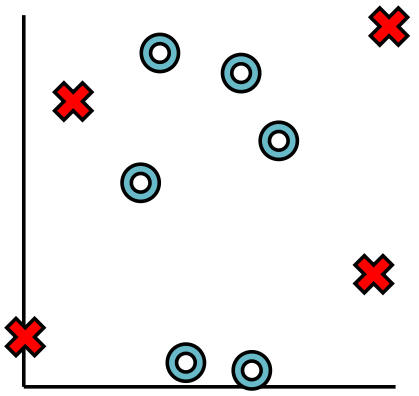
But how did we know how the loss function will change? For a small problem, we can move and see. But remember that machine learning models are large.

In the first module of Minitorch, we will see how to compute the direction efficiently for small problems, and then scale it up to much large models.

Neural Networks

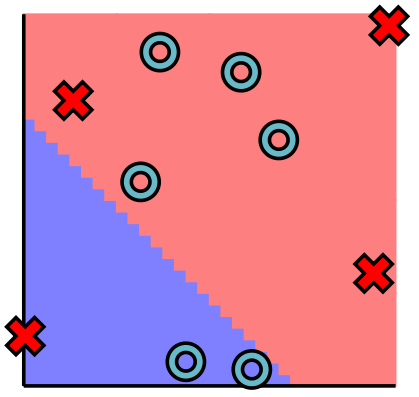
The linear model class can be used to find good fits to the data we have considered so far, but it fails for data that splits up into multiple segments. These datasets are not *linearly separable*. Let us consider a very simple dataset with this property.

```
split_graph(s1_hard, s2_hard, show_origin=True)
```



Let's look at our dataset:

```
model = Linear(1, 1, -0.7)
```

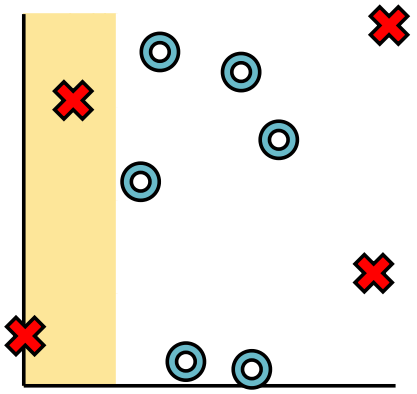


An alternative model class for this data is a neural network. Neural networks can be used to specify a much wider range of separators.

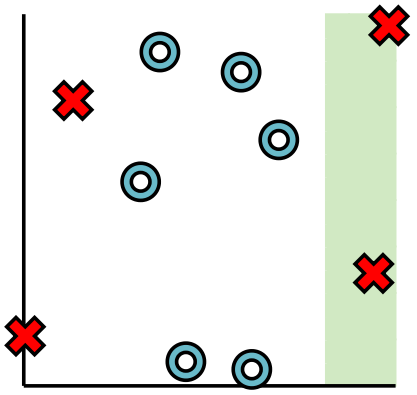
Neural networks are compound model classes that divide classification into two or more stages.

Each stage uses a linear model to seperate the data. And then an *activation* function to reshape it.

To see how this works consider how we might split up the datasets above. Instead of splitting all the points directly, we might first split off the left points,



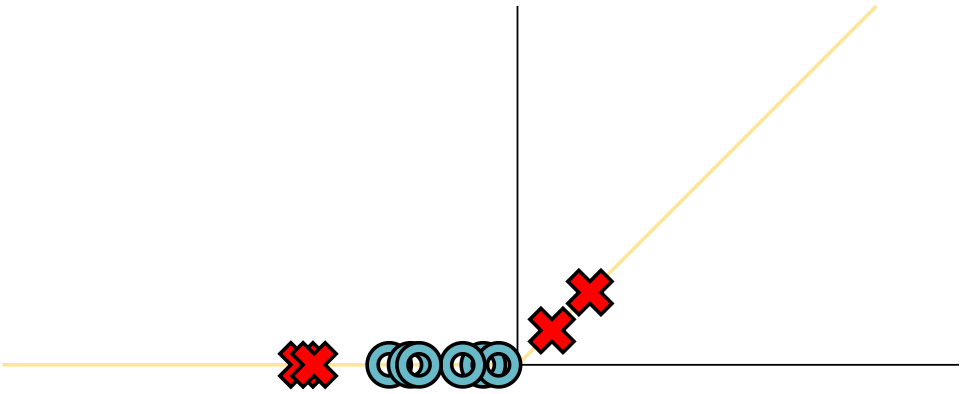
And then produce another separator (green) to pull apart the red points,



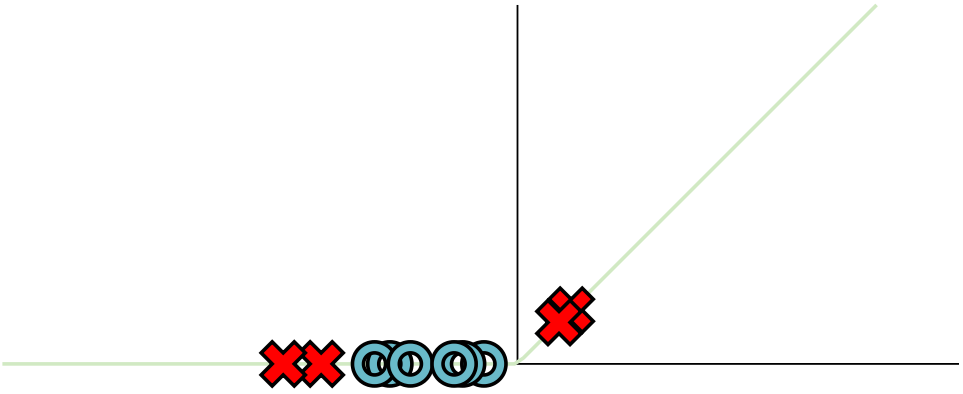
We would like only points in the green or yellow sections to be classified as X's.

To do this, we employ an activation function that filters out only these points. This function is known as a ReLU function, which is a fancy way of saying "threshold".

$$\mathrm{ReLU}(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases}$$

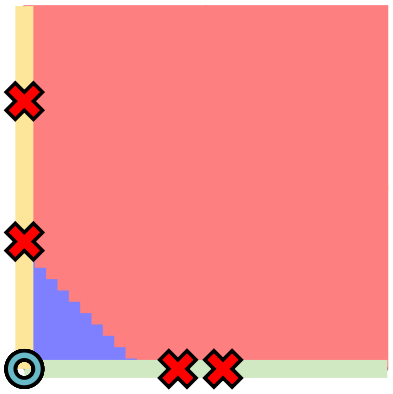


For the yellow separator, the ReLU yields the following values:



Basically the right X's are thresholed to positive values and the other O's and X's are 0.

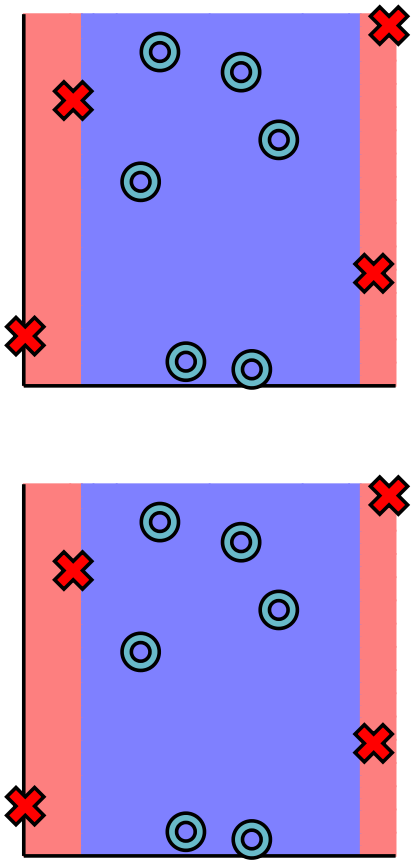
Finally yellow and green become our new x_1, x_2. Since all the O's are now at the origin it is very easy to separate out the space.



Looking back at the original model, this process appears like it has produced two lines to pull apart the data.

```
@dataclass
class MLP:
    lin1: Linear
    lin2: Linear
    final: Linear

    def forward(self, x1, x2):
        x1_1 = minitorch.operators.relu(self.lin1.forward(x1,
x2))
        x2_1 = minitorch.operators.relu(self.lin2.forward(x1,
x2))
        return self.final.forward(x1_1, x2_1)
```

```
mlp = MLP(green, yellow, Linear(3, 3, -0.3))
draw_with_hard_points(mlp)
```



Mathematically we can think of the transformed data as values h_1, h_2 which we get from applying separators with different parameters to the original data. The final prediction then applies a separator to h_1, h_2.

$$
\begin{aligned}
h_1 &= \mathrm{ReLU}(x_1 \times w_1^0 + x_2 \times w_2^0 + b^0) \\
h_2 &= \mathrm{ReLU}(x_1 \times w_1^1 + x_2 \times w_2^1 + b^1) \\
m(x_1, x_2) &= h_1 \times w_1 + h_2 \times w_2 + b
\end{aligned}
$$

Here $w_1, w_2, w_1^0, w_2^0, w_1^1, w_2^1, b, b^0, b^1$ are all parameters. We have gained more flexible models, at the cost of now needing to fit many more parameters to the data.

This neural network will be the main focus for the first couple models. It appears quite simple, but fitting it effectively will require building up systems infrastructure. Once we have this infrastructure, though, we will be able to easily support most modern neural network models.

Assignment

This module shows how to build the first version of MiniTorch using only simple values and functions. This covers key aspects of auto-differentiation: the key technique in the system. Then you will use your code to train a preliminary model.

Guides

- Derivatives
- Scalar
- Autodifferentiation
- Backpropagation

Task 1.1: Numerical Derivatives

Implement scalar numerical derivative calculation. This function will not be used in the main library but will be critical for testing the whole module.

> **Todo**
>
> Complete the following function in `minitorch/autodiff.py` and pass tests marked as `task1_1`.

`minitorch.autodiff.central_difference(f: Any, *vals: Any, arg: int = 0, epsilon: float = 1e-06) -> Any`

Computes an approximation to the derivative of `f` with respect to one arg.

See :doc:`derivative` or https://en.wikipedia.org/wiki/Finite_difference for more details.

```
f : arbitrary function from n-scalar args to one value
*vals : n-float values $x_0 \ldots x_{n-1}$
arg : the number $i$ of the arg to compute the derivative
epsilon : a small constant
```

```
An approximation of $f'_i(x_0, \ldots, x_{n-1})$
```

Task 1.2: Scalars

Implement the overridden mathematical functions required for the `minitorch.Scalar` class. Each of these requires wiring the internal Python operator to the correct `minitorch.Function.forward` call.

Read the example ScalarFunctions that we have implemented for guidelines. You may find it useful to reuse the operators from Module 0.

We have built a debugging tool for you to observe the workings of your expressions to see how the graph is built. You can run it in the *Autodiff Sandbox*. You can alter the expression at the top of the file and then run the code to create a graph in Streamlit:

```
streamlit run project/app.py -- 1
```

Todo

Add `ScalarFunction` classes with the following `forward` methods in `minitorch/scalar_functions.py` for the following math functions.

- mul
- inv
- neg
- sigmoid
- relu
- exp
- lt
- eq

Todo

Complete the following function in `minitorch/scalar.py`, and pass tests marked as `task1_2`. See Python numerical overrides for the interface of these methods. All of these functions should return `minitorch.Scalar` arguments.

- less than
- greater than
- subtract
- negation
- +
- log
- exp
- sigmoid
- relu

Task 1.3: Chain Rule

Implement the chain_rule function in Scalar for functions of arbitrary arguments. This function should be able to backward process a function by passing it in a context and d and then collecting the local derivatives. It should then pair these with the right variables and return them. This function is also where we filter out constants that were used on the forward pass, but do not need derivatives.

Todo

Complete the following function in `minitorch/scalar.py`, and pass tests marked as `task1_3`.

```
minitorch.Scalar.chain_rule(d_output: Any) -> Iterable[Tuple[Variable, Any]]
```

Task 1.4: Backpropagation

Implement backpropagation. Each of these requires wiring the internal Python operator to the correct `minitorch.Function.backward` call.

Read the example ScalarFunctions that we have implemented for guidelines. Feel free to also consult differentiation rules if you forget how these identities work.

Todo

Complete the following functions in `minitorch/autodiff.py` and pass tests marked as `task1_4`.

`minitorch.topological_sort(variable: Variable) -> Iterable[Variable]`

Computes the topological order of the computation graph.

```
variable: The right-most variable
```

```
Non-constant Variables in topological order starting from the right.
```

`minitorch.backpropagate(variable: Variable, deriv: Any) -> None`

Runs backpropagation on the computation graph in order to compute derivatives for the leave nodes.

```
variable: The right-most variable
deriv  : Its derivative that we want to propagate backward to the
leaves.
```

No return. Should write to its results to the derivative values of each leaf through `accumulate_derivative`.

And add backward methods to each of your `ScalarFunction`s.

Task 1.5: Training

If your code works, you should now be able to run the training script. Study the code in `project/run_scalar.py` carefully to understand what the neural network is doing.

You will also need Module code to implement the parameters `Network` and for `Linear`. You can modify the dataset and the module with the parameters at the bottom of the file. Start with this simple config:

```
PTS = 50
DATASET = minitorch.datasets["Simple"](PTS)
HIDDEN = 2
RATE = 0.5
```

You can then move up to something more complex, for instance:

```
PTS = 50
DATASET = minitorch.datasets["Xor"](PTS)

HIDDEN = 10
RATE = 0.5
```

If your code is successful, you should be able to run the full visualization:

```
streamlit run project/app.py -- 1
```

Todo

Train a scalar model for each of the 4 main datasets.

Add the output training logs and final images to your README file.

Tensors

We now have a fully developed autodifferentiation system built around scalars. This system is correct, but you saw during training that it is inefficient. Every scalar number requires building an object, and each operation requires storing a graph of all the values that we have previously created. Training requires repeating the above operations, and running models, such as a linear model, requires a `for` loop over each of the terms in the network.

This module introduces and implements a *tensor* object that will solve these problems. Tensors group together many repeated operations to save Python overhead and to pass off grouped operations to faster implementations.

Guides

- Tensors
- Broadcasting
- Tensor Operations
- Autograd

For this module we have implemented the skeleton `tensor.py` file for you. This is similar in spirit to `scalar.py` from the last assignment. Before starting, it is worth reading through this file to have a sense of what a Tensor does. Each of the following tasks asks you to implement the methods this file relies on:

- `tensor_data.py` : Indexing, strides, and storage
- `tensor_ops.py` : Higher-order tensor operations
- `tensor_functions.py` : Autodifferentiation-ready functions

Tasks 2.1: Tensor Data - Indexing

The MiniTorch library implements the core tensor backend as `minitorch.TensorData`. This class handles indexing, storage, transposition, and low-level details such as strides. You will first implement these core functions before turning to the user-facing class `minitorch.Tensor`.

Todo

Complete the following functions in `minitorch/tensor_data.py`, and pass tests marked as `task2_1`.

`minitorch.index_to_position(index: Index, strides: Strides) -> int`

Converts a multidimensional tensor `index` into a single-dimensional position in storage based on strides.

```
index : index tuple of ints
strides : tensor strides
```

```
Position in storage
```

`minitorch.to_index(ordinal: int, shape: Shape, out_index: OutIndex) -> None`

Convert an `ordinal` to an index in the `shape`. Should ensure that enumerating position 0 ... size of a tensor produces every index exactly once. It may not be the inverse of `index_to_position`.

```
ordinal: ordinal position to convert.
shape : tensor shape.
out_index : return index corresponding to position.
```

`minitorch.TensorData.permute(*order: int) -> TensorData`

Permute the dimensions of the tensor.

```
*order: a permutation of the dimensions
```

```
New `TensorData` with the same storage and a new dimension order.
```

Tasks 2.2: Tensor Broadcasting

Todo

Complete following functions in `minitorch/tensor_data.py` and pass tests marked as `task2_2`.

`minitorch.shape_broadcast(shape1: UserShape, shape2: UserShape) -> UserShape`

Broadcast two shapes to create a new union shape.

```
shape1 : first shape
shape2 : second shape
```

```
broadcasted shape
```

```
IndexingError : if cannot broadcast
```

`minitorch.broadcast_index(big_index: Index, big_shape: Shape, shape: Shape, out_index: OutIndex) -> None`

Convert a `big_index` into `big_shape` to a smaller `out_index` into `shape` following broadcasting rules. In this case it may be larger or with more dimensions than the `shape` given. Additional dimensions may need to be mapped to 0 or removed.

```
big_index : multidimensional index of bigger tensor
big_shape : tensor shape of bigger tensor
shape : tensor shape of smaller tensor
out_index : multidimensional index of smaller tensor
```

```
None
```

Tasks 2.3: Tensor Operations

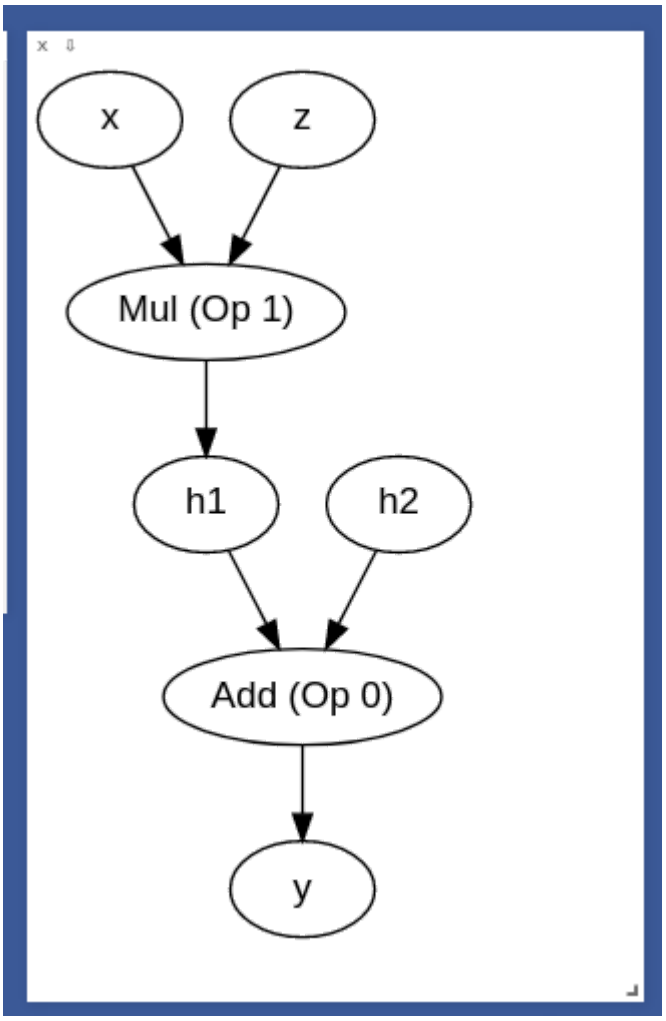
Tensor operations apply high-level, higher-order operations to all elements in a tensor simultaneously. In particular, you can map, zip, and reduce tensor data objects together. On top of this foundation, we can build up a `Function` class for Tensor, similar to what we did

for the ScalarFunction. In this task, you will first implement generic tensor operations and then use them to implement `forward` for specific operations.

We have built a debugging tool for you to observe the workings of your expressions to see how the graph is built. You can alter the expression at in `Streamlit` to view the graph

```
y = x * z + 10.0
```

```
>>> python project/show_expression.py
```



Todo

Add functions in `minitorch/tensor_ops.py` and `minitorch/tensor_functions.py` for each of the following, and pass tests marked as `task2_3`.

```
minitorch.tensor_ops.tensor_map(fn: Callable[[float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides], None]
```

Low-level implementation of tensor map between tensors with *possibly different strides*.

Simple version:

- Fill in the `out` array by applying `fn` to each value of `in_storage` assuming `out_shape` and `in_shape` are the same size.

Broadcasted version:

- Fill in the `out` array by applying `fn` to each value of `in_storage` assuming `out_shape` and `in_shape` broadcast. (`in_shape` must be smaller than `out_shape`).

```
fn: function from float-to-float to apply
```

```
Tensor map function.
```

`minitorch.tensor_ops.tensor_zip(fn: Callable[[float, float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides, Storage, Shape, Strides], None]`

Low-level implementation of tensor zip between tensors with *possibly different strides*.

Simple version:

- Fill in the `out` array by applying `fn` to each value of `a_storage` and `b_storage` assuming `out_shape` and `a_shape` are the same size.

Broadcasted version:

- Fill in the `out` array by applying `fn` to each value of `a_storage` and `b_storage` assuming `a_shape` and `b_shape` broadcast to `out_shape`.

```
fn: function mapping two floats to float to apply
```

```
Tensor zip function.
```

`minitorch.tensor_ops.tensor_reduce(fn: Callable[[float, float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides, int], None]`

Low-level implementation of tensor reduce.

- `out_shape` will be the same as `a_shape` except with `reduce_dim` turned to size `1`

```
fn: reduction function mapping two floats to float
```

```
Tensor reduce function.
```

`minitorch.tensor_functions.Mul.forward(ctx: Context, a: Tensor, b: Tensor) -> Tensor` `staticmethod`

`minitorch.tensor_functions.Sigmoid.forward(ctx: Context, t1: Tensor) -> Tensor` `staticmethod`

`minitorch.tensor_functions.ReLU.forward(ctx: Context, t1: Tensor) -> Tensor` `staticmethod`

`minitorch.tensor_functions.Log.forward(ctx: Context, t1: Tensor) -> Tensor` `staticmethod`

`minitorch.tensor_functions.Exp.forward(ctx: Context, t1: Tensor) -> Tensor` `staticmethod`

`minitorch.tensor_functions.LT.forward(ctx: Context, a: Tensor, b: Tensor) -> Tensor` `staticmethod`

`minitorch.tensor_functions.EQ.forward(ctx: Context, a: Tensor, b: Tensor) -> Tensor` `staticmethod`

`minitorch.tensor_functions.Permute.forward(ctx: Context, a: Tensor, order: Tensor) -> Tensor` `staticmethod`

`minitorch.tensor_functions.IsClose.forward(ctx: Context, a: Tensor, b: Tensor) -> Tensor` `staticmethod`

Tasks 2.4: Gradients and Autograd

Similar to `minitorch.Scalar`, `minitorch.Tensor` is a Variable that supports autodifferentiation. In this task, you will implement `backward` functions for tensor operations.

Todo

Complete following functions in `minitorch/tensor_functions.py`, and pass tests marked as `task2_4`.

`minitorch.tensor_functions.Mul.backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, Tensor]` staticmethod

`minitorch.tensor_functions.Sigmoid.backward(ctx: Context, grad_output: Tensor) -> Tensor` staticmethod

`minitorch.tensor_functions.ReLU.backward(ctx: Context, grad_output: Tensor) -> Tensor` staticmethod

`minitorch.tensor_functions.Log.backward(ctx: Context, grad_output: Tensor) -> Tensor` staticmethod

`minitorch.tensor_functions.Exp.backward(ctx: Context, grad_output: Tensor) -> Tensor` staticmethod

`minitorch.tensor_functions.LT.backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, Tensor]` staticmethod

`minitorch.tensor_functions.EQ.backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, Tensor]` staticmethod

`minitorch.tensor_functions.Permute.backward(ctx: Context, grad_output: Tensor) -> Tuple[Tensor, float]` staticmethod

Task 2.5: Training

If your code works you should now be able to move on to the tensor training script in `project/run_tensor.py`. This code runs the same basic training setup as in `module1`, but now utilize your tensor code.

Todo

Implement the missing `forward` functions in `project/run_tensor.py`. They should do exactly the same thing as the corresponding functions in `project/run_scalar.py`, but now use the tensor code base.

- Train a tensor model and add your results for all datasets to the README.
- Record the time per epoch reported by the trainer. (It is okay if it is slow).

Efficiency

In addition to helping simplify code, tensors provide a basis for speeding up computation. In fact, they are really the only way to efficiently write deep learning code in a slow language like Python. However, nothing we have done so far really makes anything faster than `module0`. This module is focused on taking advantage of tensors to write fast code, first on standard CPUs and then using GPUs.

You need the files from previous assignments, so make sure to pull them over to your new repo.

Guides

- Parallelism
- Matrix Multiply
- CUDA

For this assignment, you will need access to a GPU. We recommend running commands in a Google Colab environment. Follow these instructions for Colab setup.

This assignment does not require you to change the main tensor object. Instead you will only change the core higher-order operations code.

- `fast_ops.py` : Low-level CPU operations
- `cuda_ops.py` : Low-level GPU operations

Task 3.1: Parallelization

> **Note**
>
> This task requires basic familiarity with Numba `prange`.
>
> Be sure to very carefully read the section on parallelism, Numba.

The main backend for our codebase are the three functions `map`, `zip`, and `reduce`. If we can speed up these three, everything we built so far will get better. This exercise asks you to utilize Numba and the `njit` function to speed up these functions. In particular if you can utilize

parallelization through `prange` you can get some big wins. Be careful though! Parallelization can lead to funny bugs.

In order to help debug this code, we have created a parallel analytics script for you

```
python project/parallel_check.py
```

Running this script will run NUMBA diagnostics on your functions.

> **Todo**
>
> Complete the following in `minitorch/fast_ops.py` and pass tests marked as `task3_1`.
>
> - Include the diagnostics output from the above script in your README.
> - Be sure that the code implements the optimizations specified in the docstrings. We will check for this explicitly.

`minitorch.fast_ops.tensor_map(fn: Callable[[float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides], None]`

NUMBA low_level tensor_map function. See `tensor_ops.py` for description.

Optimizations:

- Main loop in parallel
- All indices use numpy buffers
- When `out` and `in` are stride-aligned, avoid indexing

```
fn: function mappings floats-to-floats to apply.
```

```
Tensor map function.
```

`minitorch.fast_ops.tensor_zip(fn: Callable[[float, float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides, Storage, Shape, Strides], None]`

NUMBA higher-order tensor zip function. See `tensor_ops.py` for description.

Optimizations:

- Main loop in parallel

- All indices use numpy buffers
- When `out`, `a`, `b` are stride-aligned, avoid indexing

```
fn: function maps two floats to float to apply.
```

```
Tensor zip function.
```

`minitorch.fast_ops.tensor_reduce(fn: Callable[[float, float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides, int], None]`

NUMBA higher-order tensor reduce function. See `tensor_ops.py` for description.

Optimizations:

- Main loop in parallel
- All indices use numpy buffers
- Inner-loop should not call any functions or write non-local variables

---

```
fn: reduction function mapping two floats to float.
```

```
Tensor reduce function
```

Task 3.2: Matrix Multiplication

Matrix multiplication is key to all the models that we have trained so far. In the last module, we computed matrix multiplication using broadcasting. In this task, we ask you to implement it directly as a function. Do your best to make the function efficient, but for now all that matters is that you correctly produce a multiply function that passes our tests and has some parallelism.

In order to use this function, you will also need to add a new `MatMul` Function to `tensor_functions.py`. We have added a version in the starter code you can copy. You might also find it useful to add a slow broadcasted `matrix_multiply` to `tensor_ops.py` for debugging.

In order to help debug this code, you can use the parallel analytics script.

After you finish this task, you may want to skip to 3.5 and experiment with training on the real task under speed conditions.

Todo

Complete the following function in `minitorch/fast_ops.py`. Pass tests marked as `task3_2`.

- Include the diagnostics output from the above script in your README.
- Be sure that the code implements the optimizations specified in the docstrings. We will check for this explicitly.

`minitorch.fast_ops._tensor_matrix_multiply(out: Storage, out_shape: Shape, out_strides: Strides, a_storage: Storage, a_shape: Shape, a_strides: Strides, b_storage: Storage, b_shape: Shape, b_strides: Strides) -> None`

NUMBA tensor matrix multiply function.

Should work for any tensor shapes that broadcast as long as

```
assert a_shape[-1] == b_shape[-2]
```

Optimizations:

- Outer loop in parallel
- No index buffers or function calls
- Inner loop should have no global writes, 1 multiply.

```
out (Storage): storage for `out` tensor
out_shape (Shape): shape for `out` tensor
out_strides (Strides): strides for `out` tensor
a_storage (Storage): storage for `a` tensor
a_shape (Shape): shape for `a` tensor
a_strides (Strides): strides for `a` tensor
b_storage (Storage): storage for `b` tensor
b_shape (Shape): shape for `b` tensor
b_strides (Strides): strides for `b` tensor
```

```
None : Fills in `out`
```

Task 3.3: CUDA Operations

We can do even better than parallelization if we have access to specialized hardware. This task asks you to build a GPU implementation of the backend operations. It will be hard to equal what PyTorch does, but if you are clever you can make these computations really fast (aim for 2x of task 3.1).

Reduce is a particularly challenging function. We provide guides and a simple practice function to help you get started.

Todo

Complete the following functions in `minitorch/cuda_ops.py`, and pass the tests marked as `task3_3`.

`minitorch.cuda_ops.tensor_map(fn: Callable[[float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides], None]`

CUDA higher-order tensor map function. ::

fn_map = tensor_map(fn) fn_map(out, ...)

```
fn: function mappings floats-to-floats to apply.
```

```
Tensor map function.
```

`minitorch.cuda_ops.tensor_zip(fn: Callable[[float, float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides, Storage, Shape, Strides], None]`

CUDA higher-order tensor zipWith (or map2) function ::

fn_zip = tensor_zip(fn) fn_zip(out, ...)

```
fn: function mappings two floats to float to apply.
```

```
Tensor zip function.
```

`minitorch.cuda_ops._sum_practice(out: Storage, a: Storage, size: int) -> None`

This is a practice sum kernel to prepare for reduce.

Given an array of length n and out of size $n//extblockDIM$ it should sum up each blockDim values into an out cell.

$[a_1, a_2, \ldots, a_{100}]$

|

$[a_1 + \ldots + a_{31}, a_{32} + \ldots + a_{64}, \ldots,]$

Note: Each block must do the sum using shared memory!

```
out (Storage): storage for `out` tensor.
a (Storage): storage for `a` tensor.
size (int):  length of a.
```

minitorch.cuda_ops.tensor_reduce(fn: Callable[[float, float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides, int], None]

CUDA higher-order tensor reduce function.

```
fn: reduction function maps two floats to float.
```

```
Tensor reduce function.
```

Task 3.4: CUDA Matrix Multiplication

Finally we can combine both these approaches and implement CUDA `matmul`. This operation is probably the most important in all of deep learning and is central to making models fast. Again, we first strive for accuracy, but, the faster you can make it, the better.

Implementing matrix multiplication and reduction efficiently is hugely important for many deep learning tasks. Follow the guides provided in class for implementing these functions.

You should document your code to show us that you understand each line. Prove to us that these lead to speed-ups on large matrix operations by making a graph comparing them to naive operations.

Todo

Implement `minitorch/cuda_ops.py` with CUDA, and pass tests marked as `task3_4`. Follow the requirements specified in the docs.

`minitorch.cuda_ops._mm_practice(out: Storage, a: Storage, b: Storage, size: int) -> None`

This is a practice square MM kernel to prepare for matmul.

Given a storage `out` and two storage `a` and `b`. Where we know both are shape [size, size] with strides [size, 1].

Size is always < 32.

Requirements:

- All data must be first moved to shared memory.
- Only read each cell in `a` and `b` once.
- Only write to global memory once per kernel.

Compute

```
for i:
    for j:
         for k:
             out[i, j] += a[i, k] * b[k, j]
```

```
out (Storage): storage for `out` tensor.
a (Storage): storage for `a` tensor.
b (Storage): storage for `b` tensor.
size (int): size of the square
```

`minitorch.cuda_ops._tensor_matrix_multiply(out: Storage, out_shape: Shape, out_strides: Strides, out_size: int, a_storage: Storage, a_shape: Shape, a_strides: Strides, b_storage: Storage, b_shape: Shape, b_strides: Strides) -> None`

CUDA tensor matrix multiply function.

Requirements:

- All data must be first moved to shared memory.
- Only read each cell in `a` and `b` once.
- Only write to global memory once per kernel.

Should work for any tensor shapes that broadcast as long as ::

```
assert a_shape[-1] == b_shape[-2]
```

Returns: None : Fills in `out`

Task 3.5: Training

If your code works, you should now be able to move on to the tensor training script in `project/run_fast_tensor.py`. This code is the same basic training setup as `module2`, but now utilizes your fast tensor code. We have left the `matmul` layer blank for you to implement with your tensor code.

Here is the command ::

```
python run_fast_tensor.py --BACKEND gpu --HIDDEN 100 --DATASET split --RATE
0.05
python run_fast_tensor.py --BACKEND cpu --HIDDEN 100 --DATASET split --RATE
0.05
```

Todo

- Implement the missing functions in `project/run_fast_tensor.py`. These should do exactly the same thing as the corresponding functions in `project/run_tensor.py`, but now use the faster backend
- Train a tensor model and add your results for all dataset to the README.
- Run a bigger model and record the time per epoch reported by the trainer.

Train a tensor model and add your results for all three dataset to the README. Also record the time per epoch reported by the trainer. (As a reference, our parallel implementation gave a 10x speedup). On a standard Colab GPU setup, aim for you CPU to get below 2 seconds per epoch and GPU to be below 1 second per epoch. (With some cleverness you can do much better.)

Networks

We now have a fully working deep learning library with most of the features of a real industrial system like Torch. To take advantage of this hard work, this module is entirely based on using the software framework. In particular, we are going to build an image recognition system. We will do this by build the infrastructure for a version of LeNet on MNIST: a classic convolutional neural network (CNN) for digit recognition, and for a 1D conv for NLP sentiment classification.

You need the files from previous assignments, so maker sure to pull them over to your new repo. We recommend you to get familiar with tensor.py, since you might find some of those functions useful for implementing this Module.

Guides

- Convolution
- Pooling
- Softmax

Task 4.1: 1D Convolution

You will implement the 1D convolution in Numba. This function gets used by the `forward` and `backward` pass of conv1d.

Todo

Complete the following function in `minitorch/fast_conv.py`, and pass tests marked as `task4_1`.

- _tensor_conv1d

Task 4.2: 2D Convolution

You will implement the 2D convolution in Numba. This function gets used by the `forward` and `backward` pass of conv2d.

Todo

Complete the following function in `minitorch/fast_conv.py`, and pass tests marked as `task4_2`.

- _tensor_conv2d

Task 4.3: Pooling

You will implement 2D pooling on tensors with an average operation.

Todo

Complete the following functions in `minitorch/nn.py`, and pass tests marked as `task4_3`.

`minitorch.tile(input: Tensor, kernel: Tuple[int, int]) -> Tuple[Tensor, int, int]`

Reshape an image tensor for 2D pooling

```
input: batch x channel x height x width
kernel: height x width of pooling
```

```
Tensor of size batch x channel x new_height x new_width x (kernel_height *
kernel_width) as well as the new_height and new_width value.
```

`minitorch.avgpool2d(input: Tensor, kernel: Tuple[int, int]) -> Tensor`

Tiled average pooling 2D

```
input : batch x channel x height x width
kernel : height x width of pooling
```

```
Pooled tensor
```

Task 4.4: Softmax and Dropout

You will implement max, softmax, and log softmax on tensors as well as the dropout and max-pooling operations.

Todo

- Complete the following functions in `minitorch/nn.py`, and pass tests marked as `task4_4`.
- Add a property tests for the function in `test/test_nn.py` and ensure that you understand its gradient computation.

`minitorch.max(input: Tensor, dim: int) -> Tensor`

`minitorch.softmax(input: Tensor, dim: int) -> Tensor`

Compute the softmax as a tensor.

$z_i = \frac{e^{x_i}}{\sum_i e^{x_i}}$

```
input : input tensor
dim : dimension to apply softmax
```

```
softmax tensor
```

`minitorch.logsoftmax(input: Tensor, dim: int) -> Tensor`

Compute the log of the softmax as a tensor.

$z_i = x_i - \log \sum_i e^{x_i}$

See https://en.wikipedia.org/wiki/LogSumExp#log-sum-exp_trick_for_log-domain_calculations

---

```
input : input tensor
dim : dimension to apply log-softmax
```

```
 log of softmax tensor
```

`minitorch.maxpool2d(input: Tensor, kernel: Tuple[int, int]) -> Tensor`

Tiled max pooling 2D

```
input: batch x channel x height x width
kernel: height x width of pooling
```

```
Tensor : pooled tensor
```

```
minitorch.dropout(input: Tensor, rate: float, ignore: bool = False) -> Tensor
```

Dropout positions based on random noise.

```
input : input tensor
rate : probability [0, 1) of dropping out each position
ignore : skip dropout, i.e. do nothing at all
```

```
tensor with random positions dropped out
```

Task 4.4b: Extra Credit

Implementing convolution and pooling efficiently is critical for large-scale image recognition. However, both are a bit harder than some of the basic CUDA functions we have written so far. For this task, add an extra file `cuda_conv.py` that implements `conv1d` and `conv2d` on CUDA. Show the output on colab.

Task 4.5: Training an Image Classifier

If your code works, you should now be able to move on to the NLP and CV training scripts in `project/run_sentiment.py` and `project/run_mnist_multiclass.py`. This script has the same basic training setup as :doc:`module3`, but now adapted to sentiment and image classification. You need to implement `Conv1D`, `Conv2D`, and `Network` for both files.

We recommend running on the command line when testing. But you can also use the Streamlit visualization to view hidden states of your model, like the following:

Todo

- Train a model on Sentiment (SST2), and add your training printout logs as a text file `sentiment.txt` to the repo. It should show train loss, train accuracy and validation accuracy. (The model should get to >70% best validation accuracy.)
- Train a model on Digit classification (MNIST) logs as a text file `mnist.txt` to the repo. It should show train loss and validation accuracy out of 16

Contributing

The Minitorch codebase is structured to mimic the experience of contributing to a real open-source project. It is not sufficient to implement functions correctly; the code itself needs to meet the specific contributor requirements. These are checked by the system automatically before acceptance.

Style

It is required to keep your code organized and clean to make it easier to debug, optimize, and document. To help with this process, we utilize required formatting on all assignments.

Fixing style bugs can be an annoying process. However, there are now tools to fix most formatting issues automatically. We use `ruff` to automatically reformat all of your code to fit most of the requirements (see the ruff website for more details).

Ruff will fix many of your issues, but cannot check for aspects like using unknown variables.

We recommend setting up your editor or IDE to highlight other style issues. Many developers utilize VSCode with plugins to check for these issues as they code.

Testing

Each assignment has a series of tests that require your code to pass. These tests are in the `tests/` directory and are in the `pytest` format (https://docs.pytest.org/en/stable/). Any function in that directory starting with `test` is run as part of the test suite.

```
>>> pytest
```

Each assignment has 4 task groups that you will need to pass. To run individual task groups you can use the `-m` option.

```
>>> pytest -m task0_0
```

In addition to running a full task which runs all of the tests, you can run tests in a single file with:

```
>>> pytest tests/test_operators.py
```

Or even a particular test with:

```
>>> pytest tests/test_operators.py -k test_sum
```

Note: PyTest will hide all print statements unless a test fails. If you want to see output for a given tests you may need to cause an assertion failure.

Type Checking

Modern versions of Python allow for static type checking to ensure that functions take and returns objects of the correct type. The Minitorch code base is fully annoted with typed on each function. Users will not have to provide types, but they provide documentation as to what functions expect as arguments and as for their return values. For example, if we are writing a function that does multiplication it would have the following signature.

```
def mul(x: float, y: float) -> float:
    ...
```

As we get to more complex topics the type signatures will get more complex as well. For example, `Iterable` is use to represent a type that can be iterated over.

```
def negList(ls: Iterable[float]) -> Iterable[float]:
    ...
```

In order to check that the code matches the types, the project requires that you use the `pyright` library. This is run as part of the style check.

Documentation

Throughout the codebase, we require you to document all functions in a standardized style. Documentation is critical for our Python codebase, and we use it to convey requirements on many functions. Functions should have docstrings in the following form (known as Google docstyle):

```
  def index(ls: List[Any], i: int) -> Any:
      """
      List indexing.

      Args:
          ls: A list of any type.
          i: An index into the list

      Returns:
          Value at ls[i].
      """
      ...
```

For short function you can also just include a single line docstring. A full description of this docstyle is listed as the Google format.

The project also requires that you keep documentation up-to-standard throughout. Lint errors will be thrown if your documentation is in the incorrect format.

Pre-Commit

These elements can be checked automatically through a tool known as pre-commit. Using the precommit tool is optional but will likely save you time in your coding process and ensure that you are not making too many unnecessary failed pull requests. You can install the tool by running,

```
pip install pre-commit
```

You can run all the checks (black, flake8, mypy, etc) and corrections on your code with the following command:

```
pre-commit run --all
```

You can also 'install' pre-commits which will run on every commit automatically, and prevent you from committing bad code.

```
pre-commit install
```

Continuous Integration (CI)

In addition to local testing, the project is set up such that on each code push, tests are automatically run and checked on the server. You are able to see how well you are doing on the assignment by committing your code, pushing to the server, and then logging in to GitHub. This process takes several minutes, but it is an easy way to keep track of your progress as you go.

Specifically, you can run:

```
>>> git commit -am "First commit"
>>> git push origin master
```

Then go to your GitHub and click on "Pull requests". Clicking on the request itself gives a link to show the current progress of your work.

Property Testing

```
from hypothesis import example, given
from hypothesis.strategies import integers
```

Testing and debugging are critical for software engineering in general, and particularly necessary for framework code that will be used in unexpected ways. Unfortunately, machine learning code is notoriously hard to test and debug. Many practitioners seem to just run models and wait until they are trained before starting the debugging process.

But how do you effectively test mathematically oriented code? Unlike many software projects where unit tests can cover most of the input cases, mathematical functions make this impossible.

For example, let's say you have a function that is meant to add two numbers (this sounds really silly, but we will see it is not)

```
def my_add(a: int, b: int) -> int:
    """A customized integer addition function."""
    out = a
    for _ in range(-b):
        out -= 1
    for _ in range(b):
        out += 1
    return out
```

A (somewhat naive) unit test might look like this::

```
def test_add_basic():
    # Check same as slow system add
    assert my_add(10, 7) == 10 + 7
    # Check that order doesn't matter
    assert my_add(10, 7) == my_add(7, 10)
```

```
test_add_basic()
```

This is fine, and certainly can help catch easy bugs, but it is not very reassuring. It is particularly devastating when your code has been running for 20 hours, and then

encounters some cases where your add function fails.

An alternative idea is to test properties instead of specific cases. That is, check if key aspects of the expected behavior always hold. For instance, you might imagine directly checking if these properties hold for every pair of integers:

```
def test_add_naive():
    for a in range(-100, 100):
        for b in range(-100, 100):
            assert my_add(a, b) == a + b
            assert my_add(a, b) == my_add(b, a)
```

This provides better coverage, but is also naive and clearly hopelessly inefficient. Unit tests are supposed to be quick easy snippets of code that can be run quickly while developing.

A clever middle ground is to use `randomized` property checking. This method was popularized by a library called QuickCheck (http://wikipedia.org/wiki/quickcheck). This approach randomly selects interesting inputs in order to test your codebase's correctness. It gives you the speed of the first approach and some of the breadth of the second. Another nice benefit of randomized property checking is that it actually makes tests shorter and easier to write since it generates cases for you.

In MiniTorch, we will use a property checking library in Python called `Hypothesis` (https://hypothesis.readthedocs.io/). Hypothesis predefines a whole set of building block *strategies* that the user can pick from when writing tests. (You can also write your strategies, which you will do in the next assignment.) You can generate integers, floats, lists, strings, etc. Each test can be decorated with values that it operates on::

```
@given(integers(), integers())
def test_add(a, b):
    # Check same as slow system add
    assert my_add(a, b) == a + b
    # Check that order doesn't matter
    assert my_add(a, b) == my_add(b, a)
```

The function `integers` is a strategy function that tells us what type of values to test on. It is a function because we may want to constrain the values in various ways. When debugging we can force it to give us some values.

```
integers(min_value=0, max_value=10).example()
```

```
10
```

It is also easy to combine randomized testing with example based testing. This can be useful if you want to create easy to debug test cases.

```
@given(integers(), integers())
@example(5, 7)
def test_add2(a, b):
    # Check same as slow system add
    assert my_add(a, b) == a + b
    # Check that order doesn't matter
    assert my_add(a, b) == my_add(b, a)
```

Modules

Researchers disagree on exactly what the term *deep* learning means, but one aspect that everyone agrees on is that deep models are big and complex. Models can include hundreds of millions of learned `parameters`. In order to work with such complex systems, it is important to have data structures which abstract away the complexity so that it is easier to access and manipulate specific components, and group together shared regions. These structures are not rigorous mathematical objects, but a convenient way of managing complex systems.

On the programming side, `Modules` have become a popular paradigm to group parameters together to make them easy to manage, access, and address. There is nothing specific to machine learning about this setup (and everything in MiniTorch could be done without modules), but they make life easier and code more organized. For now, do not worry about what parameters are for, just that we would like to group and name them in a convenient way.

In Torch, modules are a hierarchical data structure. Each module stores three things: 1) parameters, 2) user data, 3) other modules. Internally, the user interacts with each of these on `self`, but under the hood the module sorts everything into three types.

Let's work through an example of module.

```
class OtherModule(Module):
    pass


class MyModule(Module):
    def __init__(self):
        # Must initialize the super class!
        super().__init__()

        # Type 1, a parameter.
        self.parameter1 = Parameter(15)

        # Type 2, user data
        self.data = 25

        # Type 3. another Module
        self.sub_module = OtherModule()
```

Internally Module partitions these elements. Parameters (type 1) are stored in a parameters dictionary, user data (type 2) is stored on `self`, modules (type 3) are stored in a modules dictionary. This is a bit tricky. Python is a very dynamic language and allows us to override simple things like assignment.

Be careful. All subclasses must begin their initialization by calling `super().__init__()`. This line allows the module to capture any members of type `Module` or `Parameter`.

The benefit of this behavior is that it allows us to easily extract all parameters and subparameters from modules. Specifically we can get out all of a modules parameters using the `named_parameters` function. This function returns a dictionary of all of the parameters in the module and in all descendent sub-modules.

```
MyModule().named_parameters()
```

```
[('parameter1', 15)]
```

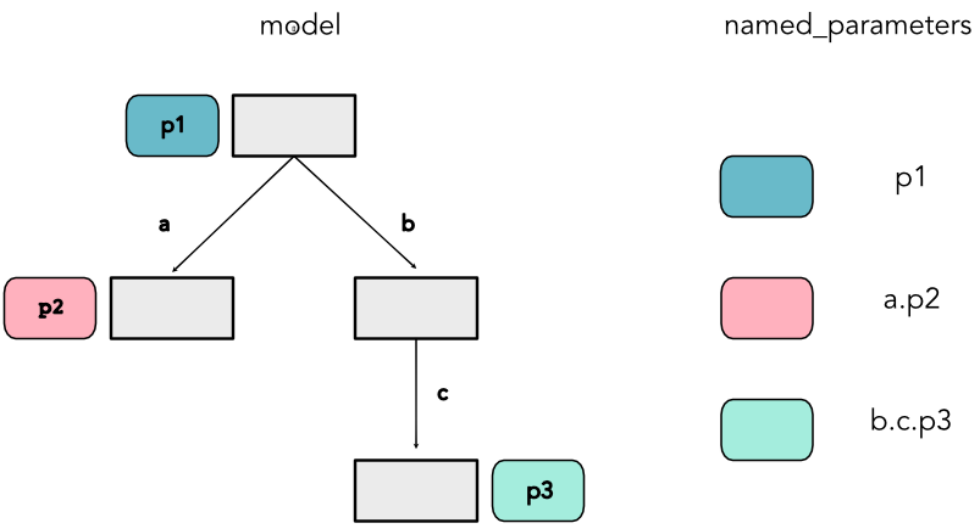
The names here refer to the keys in the dictionary which give the path to each parameter in the tree (similar to python dot notation). Critically this function does not just return the current module's parameters, but recursively collects parameters from all the modules below as well.

Here is an example of how you can create a tree of modules and then extract the flattened parameters

```
class Module1(Module):
    def __init__(self):
        super().__init__()
        self.p1 = Parameter(5)
        self.a = Module2()
        self.b = Module3()


class Module2(Module):
    def __init__(self):
        super().__init__()
        self.p2 = Parameter(10)


class Module3(Module):
    def __init__(self):
        super().__init__()
        self.c = Module4()
```

```
class Module4(Module):
    def __init__(self):
        super().__init__()
        self.p3 = Parameter(15)


Module1().named_parameters()
```

```
[('p1', 5), ('a.p2', 10), ('b.c.p3', 15)]
```



Modules can also be used to find all submodules.

```
Module1().modules()
```

```
[Module2(),
 Module3(
   (c): Module4()
 )]
```

Additionally, a module can have a `mode` indicating how it is currently operated. The mode should propagate to all of its child modules. For simplicity, we only consider the train and eval mode.

```
module1 = Module1()
module1.train()
module1.training
```

```
True
```

```
module1.a.training
```

```
True
```

```
module1.eval()
module1.training
```

```
False
```

Functional

Externally, MiniTorch supports the standard Torch API, which allows Python users to develop in a standard Python coding style. Internally, the library uses a functional-style. This approach is preferred for two reasons: first, it makes it easy to test, and secondly it makes it easy to optimize. While this style requires a bit of extra thought to understand, it has some benefits.

Primarily we will use the functional style to define higher-order, functions. These are functions that take functions as arguments and return functions as results. Python defines a special type for these: `Callable`.

```
from typing import Callable, Iterable
```

Any function can be turned into a variable of type callable. (Although this in itself is not very interesting).

```
def add(a: float, b: float) -> float:
    return a + b


v: Callable[[float, float], float] = add


def mul(a: float, b: float) -> float:
    return a * b


v: Callable[[float, float], float] = mul
```

It is interesting though, to pass functions as arguments to other functions. For example, we can pass a callable to a function that uses it.

```
def combine3(
    fn: Callable[[float, float], float], a: float, b: float, c:
float
) -> float:
    return fn(fn(a, b), c)
```

```
print(combine3(add, 1, 3, 5))
print(combine3(mul, 1, 3, 5))
```

```
9
15
```

We can also use this approach to create new functions as reutrn arguments.

```
def combine3(fn):
    def new_fn(a: float, b: float, c: float) -> float:
        return fn(fn(a, b), c)

    return new_fn
```

```
add3: Callable[[float, float, float], float] = combine3(add)
mul3: Callable[[float, float, float], float] = combine3(mul)

print(add3(1, 3, 5))
```

```
9
```

As an extended example, let's create a higher-order version of the filter function. Filter should take a list and return only the values that are true under a function.

```
def filter(fn: Callable[[float], bool]) ->
Callable[[Iterable[float]], Iterable[float]]:
    def apply(ls: Iterable[float]):
        ret = []
        for x in ls:
            if fn(x):
                ret.append(x)
        return ret

    return apply
```

We then use this to create a new function.

```
def more_than_4(x: float) -> bool:
    return x > 4


filter_for_more_than_4 = filter(more_than_4)
filter_for_more_than_4([1, 10, 3, 5])
```

```
[10, 5]
```

Functional programming can be elegant, but also hard. When in doubt remember that you can always write things out in a simpler form first and then check that you get the same sensible result.

Visualization

While testing is nice for maintaining correctness, exploratory analysis is also critical for gaining intuition. When you are stuck, often the best thing to do is to just look at your data and outputs. Visualizing our system can't prove that it is correct, but it can often directly help us to figure out what goes wrong. Throughout our development, we will use visualization to observe intermediate states, training progress, outputs, and even final models.

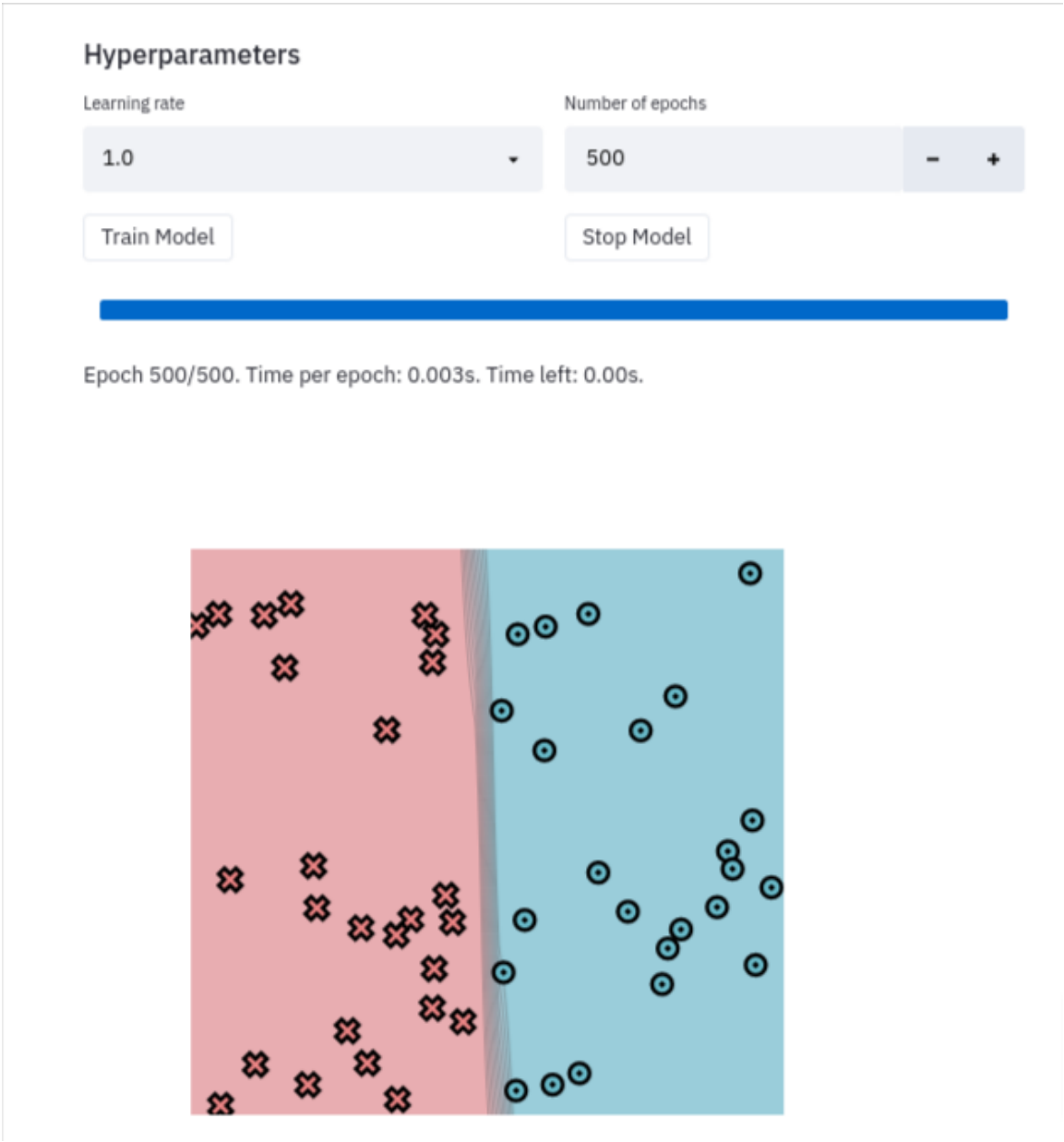
The main library we will use is called Streamlit (streamlit/streamlit).

You can think of it as sending images and graphs from your code to a centralized, organized place. Nothing that magical, we could just output them to a directory, but we will see this has some nice benefits.

To start streamlit, you need to run the following command in your virtual env.

```
>>> streamlit run app.py -- {{module number}}
```

Next, open up a browser window and, go to http://localhost:8501 (or whichever port it starts on).



Each unit will include a set of different sandbox elements that you can use and visualize to explore your underlying problem. You can use these to help you debug as well as plot functions that go directly to this board.

There is a lot more you can do with Streamlit. Check out [streamlit](https://docs.streamlit.io/en/stable/) for a list of goodies.

Derivatives

We begin by discussing derivatives in the context of programming. We assume no prior knowledge of derivatives, so we start with basic concepts and develop notation gradually.

Symbolic Derivatives

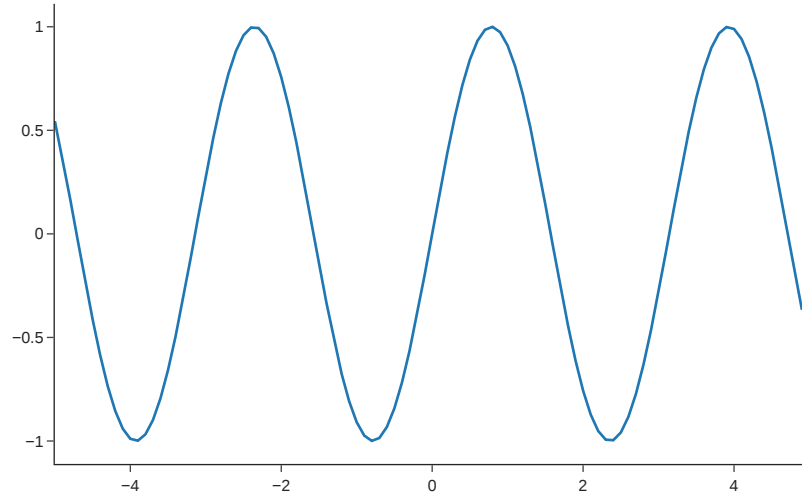
```
import math
```

Assume we are given a function, $f(x) = \sin(2 \times x)$. We can compute a function for its derivative by applying rules from univariate calculus. We use *Lagrange* notation where the derivative of a one-argument function f is denoted f'. To compute f' we can apply standard univariate derivative rules.

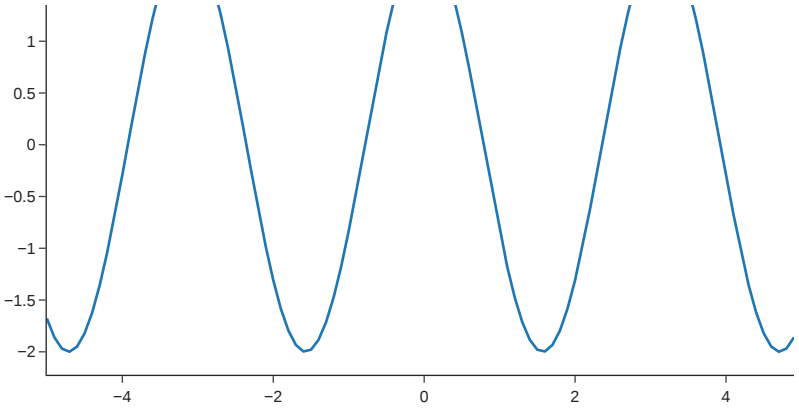
$$f'(x) = 2 \times \cos(2x)$$

```
def f(x):
    return math.sin(2 * x)


def d_f(x):
    return 2 * math.cos(2 * x)


plot_function("f(x) = sin(2x)", f)
plot_function("f'(x) = cos(2x)", d_f)
```

f(x) = sin(2x)



f'(x) = cos(2x)

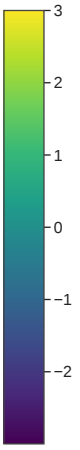


We also work with two-argument functions.

$$f(x, y) = \sin(x) + 2\cos(y)$$

```
def f(x, y):
    return math.sin(x) + 2 * math.cos(y)


plot_function3D("f(x, y) = sin(x) + 2 * cos(y)", f)
```

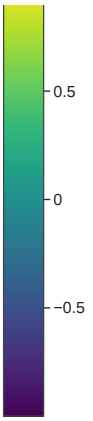
f(x, y) = sin(x) + 2 * cos(y)



We use a subscript notation to indicate which argument we are taking a derivative with respect to.

```
def d_f_x(x, y):
    return math.cos(x)


plot_function3D("f'_x(x, y) = cos(x)", d_f_x)
```

f'_x(x, y) = cos(x)



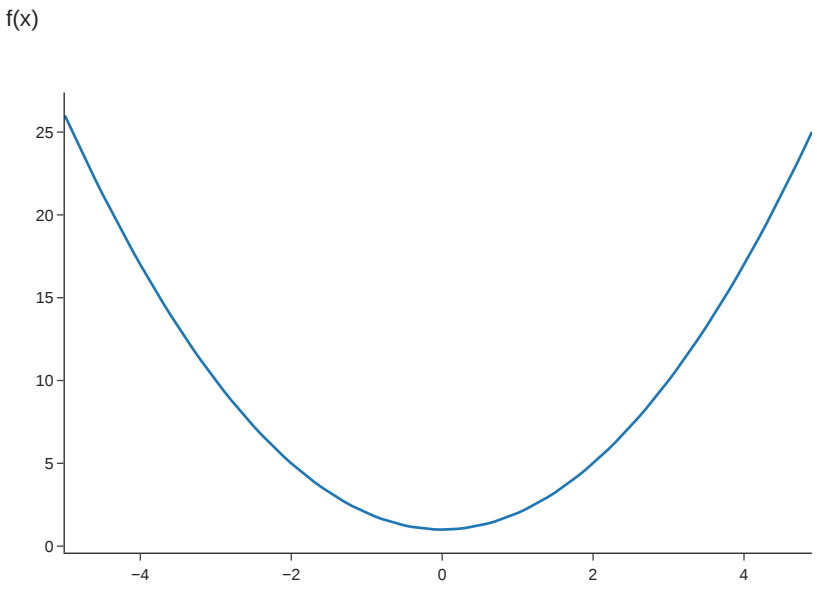
In general, we will refer to this process of mathematical transformation as the *symbolic* derivative of the function. When available symbolic derivatives are ideal, they tell us everything we need to know about the derivative of the function.

Numerical Derivatives

Visually, derivative functions correspond to slopes of tangent lines in 2D. Let's start with this simple function:

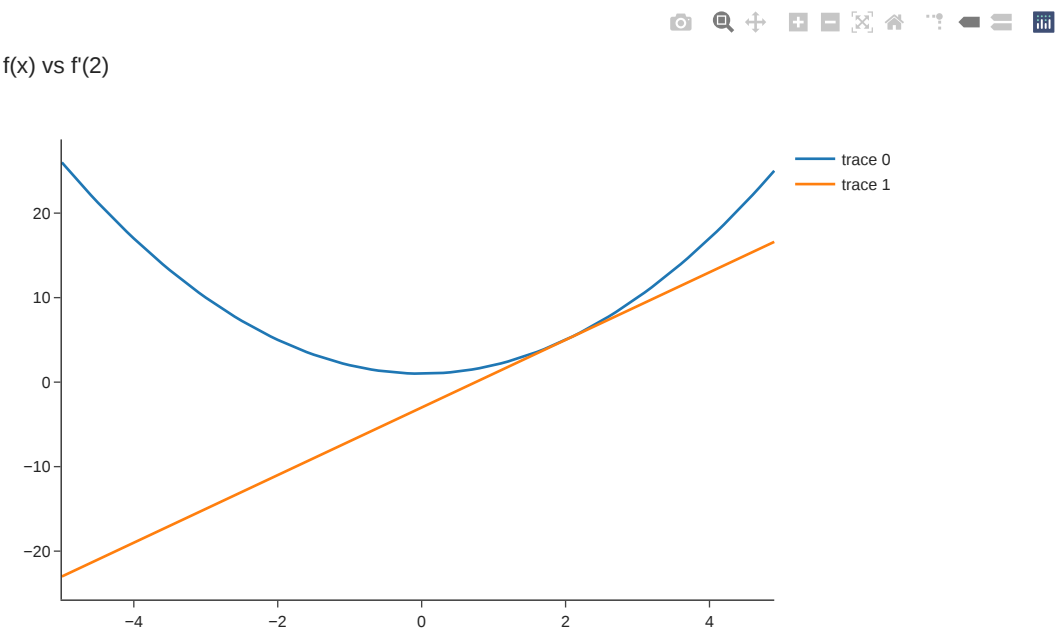
$$f(x) = x^2 + 1$$

```
def f(x):
    return x * x + 1.0


plot_function("f(x)", f)
```



Its derivative at an arbitrary input is the slope of the line tangent to that input.

```
def d_f(x):
    return 2 * x


def tangent_line(slope, x, y):
    def line(x_):
        return slope * (x_ - x) + y

    return line


plot_function("f(x) vs f'(2)", f, fn2=tangent_line(d_f(2), 2, f(2)))
```



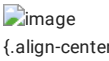
The above visual representation motivates an alternative approach to estimate a [numerical]{.title-ref} derivative. The underlying assumption is that we assume we do not know the symbolic form of the function, and instead want to estimate it by querying specific values.

Recall one definition of the derivative function is this slope as we approach to a tangent line:

If we set $epsilon$ to be very small, we get an approximation of the derivative function:

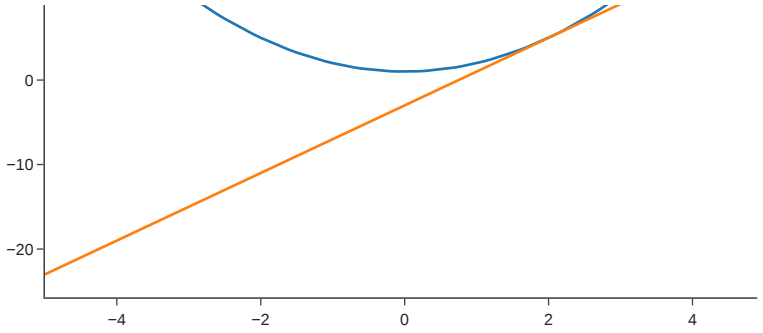
Alternatively, you could imagine approaching x from the other side, which would yield a different derivative function:

You can show that doing both simultaneously yields a better approximation (you probably proved this in high school!):



{.align-center}

```
eps = 1e-5
slope = (f(2 + eps) - f(2 - eps)) / (2 * eps)
plot_function("f(x) vs f'(2)", f, fn2=tangent_line(slope, 2, f(2)))
```



This formula is known as the central difference, and is a specific case of finite differences.

When working with functions of multiple arguments. The each derivative corresponds to the slope of one dimension of the tangent plane. The central difference approach can only tell us one of these slope at a time.

The more variables we have, the more function calls we need to make.

Implementing Numerical Approximations

The key benefit of the [numerical]{.title-ref} approach is that we do not need to know everything about the function: all we need is to be able to compute its value under a given input. From a programming sense, this means we can approximate the derivative for any black-box function. Note, we did not need to actually know the specifics of the function to compute this derivative.

In implementation, it means we can write a [higher-order function]{.title-ref} of the following form:

```
def central_difference(f: Callable[[float], float], x: float) -> float: ...
```

Assume we are just given an arbitrary python function:

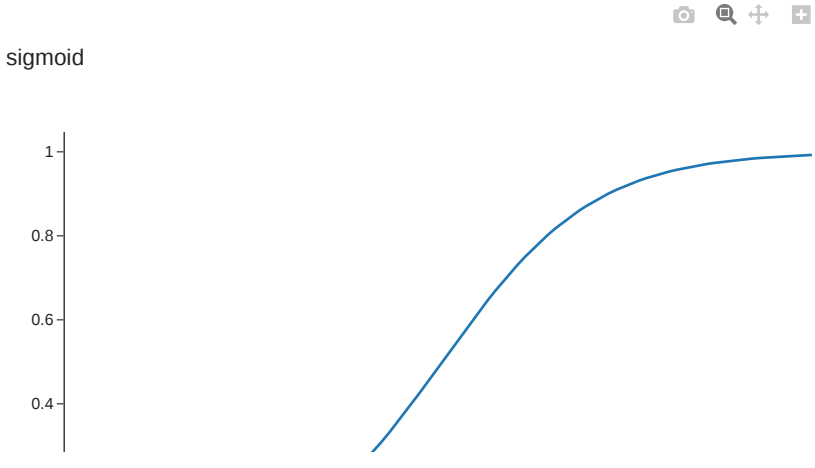
```
def f(x: float) -> float:
    """Compute some unknown function of x."""
    ...
```

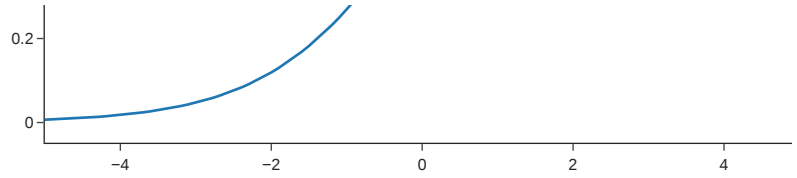
we can call `central_difference(f, x)` to immediately approximate the derivative of this function f on input x.

We will see that this approach is not a great way to train machine learning models, but it provides a generic alternative approach to check if your derivative functions are correct, e.g. free property testing.

Here are some examples of Module-0 functions with central difference applied. It is important to know what derivatives of these important functions look like.

```
plot_function("sigmoid", minitorch.operators.sigmoid)
```

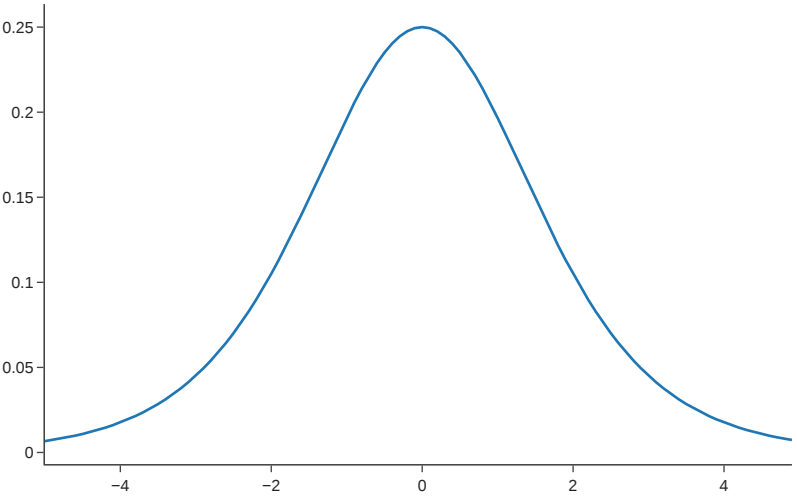





```
def d_sigmoid(x):
    return minitorch.central_difference(minitorch.operators.sigmoid, x)
```

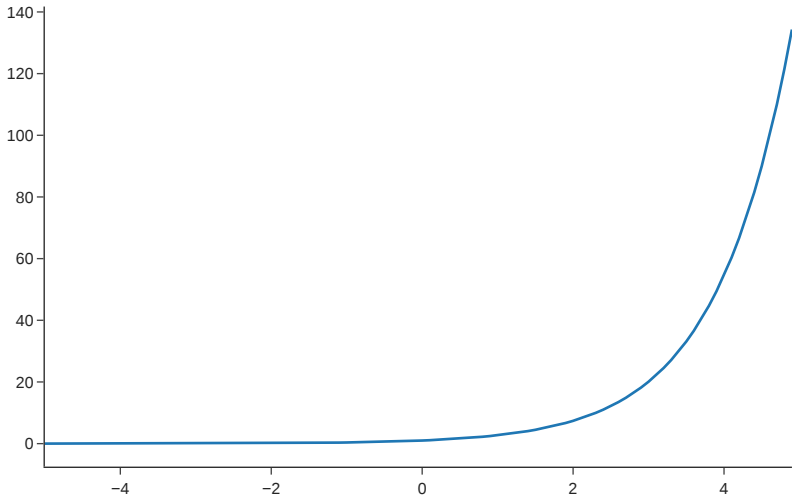
```
plot_function("Derivative of sigmoid", d_sigmoid)
```

Derivative of sigmoid




```
plot_function("exp", minitorch.operators.exp)
```

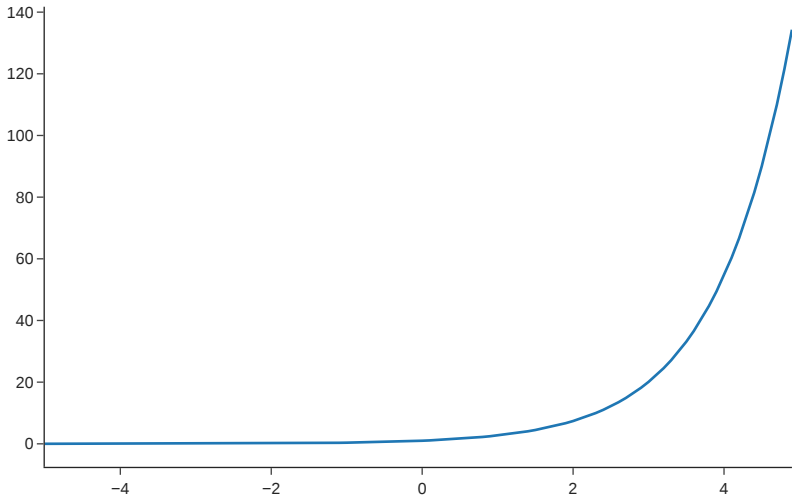
exp



```
def d_exp(x):
    return minitorch.central_difference(minitorch.operators.exp, x)
```

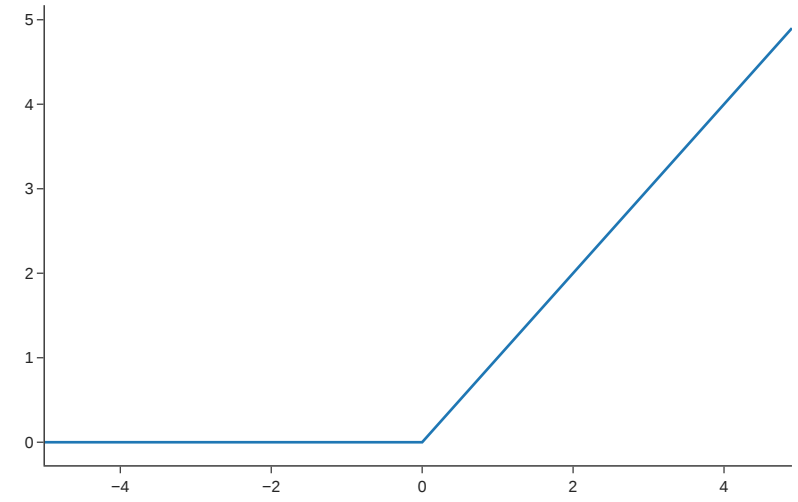
```
plot_function("Derivative of exp", d_exp)
```

Derivative of exp




```
plot_function("ReLU", minitorch.operators.relu)
```

ReLU

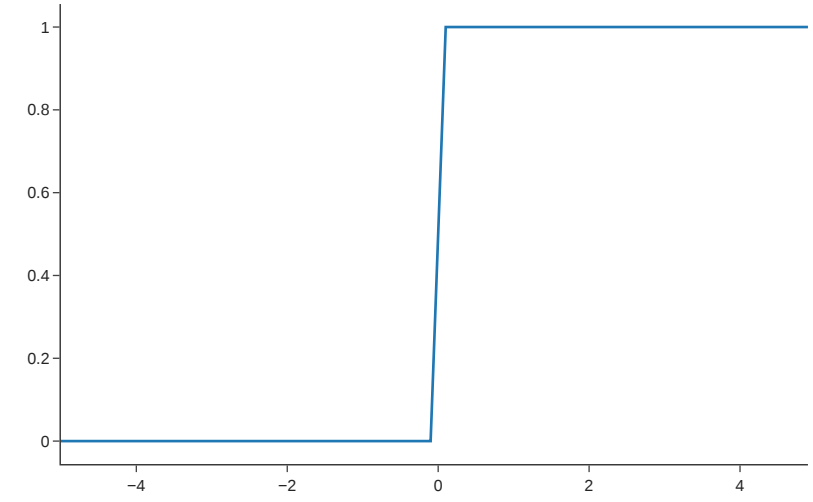



```
def d_relu(x):
    return minitorch.central_difference(minitorch.operators.relu, x)
```

```
plot_function("Derivative of ReLU", d_relu)
```

Derivative of ReLU

Derivative of ReLU



Scalar

In last section, we discussed two ways to compute derivatives. *Symbolic derivatives* require access to the full symbolic function, whereas *numerical derivatives* require only a black-box function. The first is precise but rigid, whereas the second is imprecise but more flexible. This module introduces a third approach known as **autodifferentiation** which is a tradeoff between symbolic and numerical methods.

Autodifferentiation works by collecting information about the computation path used within the function, and then transforming this information into a procedure for computing derivatives. Unlike the black-box method, autodifferentiation will allow us to use this information to compute each step more precisely.

However, in order to collect the information about the computation path, we need to track the internal computation of the function. This can be hard to do since Python does not expose how its inputs are used in the function directly: all we get is the output only. This doc describes one method for tracking computation.

Overriding Numbers

Since we do not have access to the underlying language interpreter, we are going to build a system to track the mathematical operations applied to each number.

1. Replace all numbers with proxy a class, which we will call `Scalar`.
2. Replace all mathematical functions with proxy operators.
3. Remember what operators were applied to each Scalar.

Consider the following code which shows the result of this approach.

```
x1 = minitorch.Scalar(10)
x2 = minitorch.Scalar(30)
y = x1 + x2
y.history
```

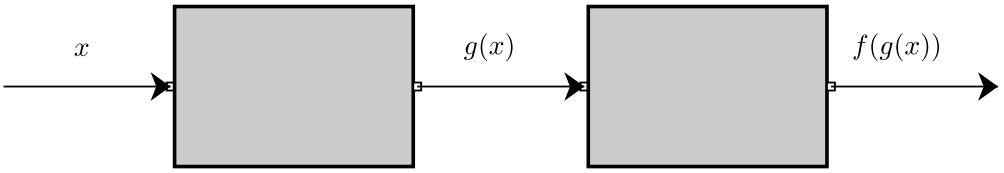
```
ScalarHistory(last_fn=<class 'minitorch.scalar_functions.Add'>, ctx=Context(no_grad=False, saved_values=()), inputs=[Scalar(10.000000), Scalar(30.000000)])
```

Scalar should behave exactly like numbers. The goal is that the user cannot tell the difference. But we will utilize the extra information to implement the operations we need.

Functions

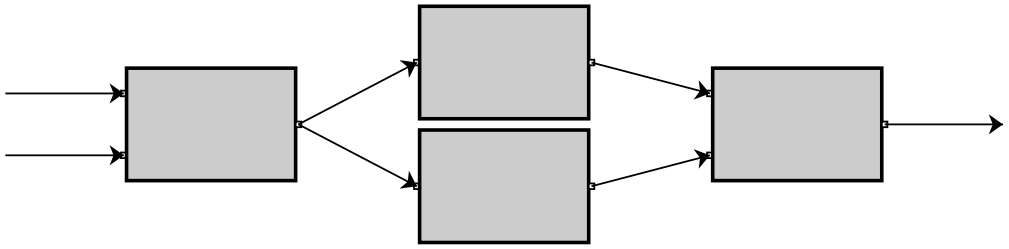
When working with these new number we restrict ourselves to use a small set of mathematical functions f of one or two arguments. Graphically, we will think of functions as little boxes. For example, a one-argument function would look like this,

Internally, the box *unwraps* the content of x, manipulates it, and returns a new value with the saved history. We can chain together two of these functions to produce more complex functions.



Similarly, a two-argument function unwraps the content of both inputs x and y, manipulates them, and returns a new wrapped version:

Finally we can create more complex functions that chain these together in various ways.



Implementation

We will implement tracking using the `Scalar` class. It wraps a single number (which is stored as an attribute) and its history.

```
x = minitorch.Scalar(10)
```

To implement functions there is a corresponding class `ScalarFunction`. We will need to reimplement each mathematical function that we would like to use by inheriting from this class.

For example, say our function is `Neg`, $g(x) = -x$

```
class Neg(ScalarFunction):
    @staticmethod
    def forward(ctx, x):
        return -x
```

Or, say the function is `Mul`, $f(x, y) = x \times y$ that multiplies x by y

```
class Mul(ScalarFunction):
    @staticmethod
    def forward(ctx: Context, x: float, y: float) -> float:
        return x * y
```

Within the forward function, x and y are always unwrapped numbers. Forward function processes and returns unwrapped values.

If we have scalars x, y, we can apply the above function by

```
z = Neg.apply(x)
out = Neg.apply(z)
```

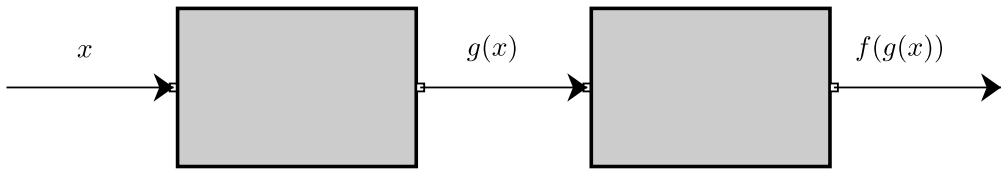
```
# or
out2 = Mul.apply(x, z)
```

Note, that we do not call forward directly but instead apply. Internally 'apply' converts the inputs to standard numbers to call forward, and then wraps the output float with the history it needs.

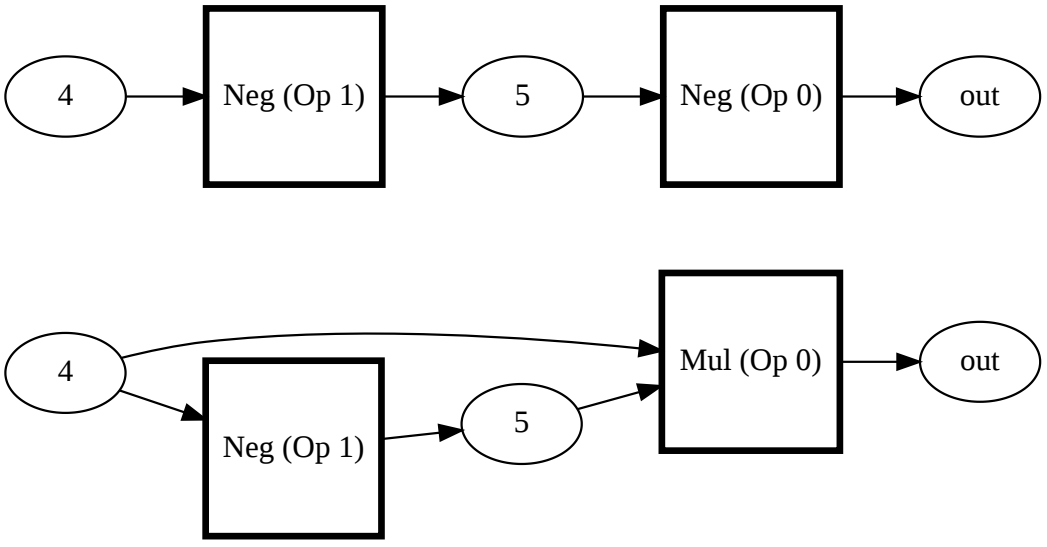
```
print(out.history)
```

```
ScalarHistory(last_fn=<class '__main__.Neg'>, ctx=Context(no_grad=False, saved_values=()), inputs=[Scalar(-10.000000)])
```

Here `out` has remembered the graph that led to its creation.



Minitorch includes a library to allow you to draw these box diagrams for arbitrarily complex functions.



Operators

There is still one minor issue. This is what our code looks like to use [Mul]{.title-ref},

```
out2 = Mul.apply(x, y)
```

It is a bit annoying to write code this way. Also, we promised that we would have functions that look just like the Python operators we are used to writing.

To get around this issue, we need to augment the `Scalar` class so that it can behave normally under standard mathematical operations. Instead of calling regular multiplication, Python will call our `mul`. Once this is achieved, we will have the ability to record and track how x is used in the Function, while still being able to write

```
out2 = x * y
```

To achieve this, the class needs to provide syntax that makes it appear like a number when in use. You can read emulating numeric types to learn how this could be done.

Autodifferentiation

We are going to utilize the computation graph as a way to automatically compute derivatives of arbitrary python functions. The trick behind this *autodifferentiation* is to implement the derivative of each invidual function, and then utilize the *chain rule* to compute a derivative for any scale value.

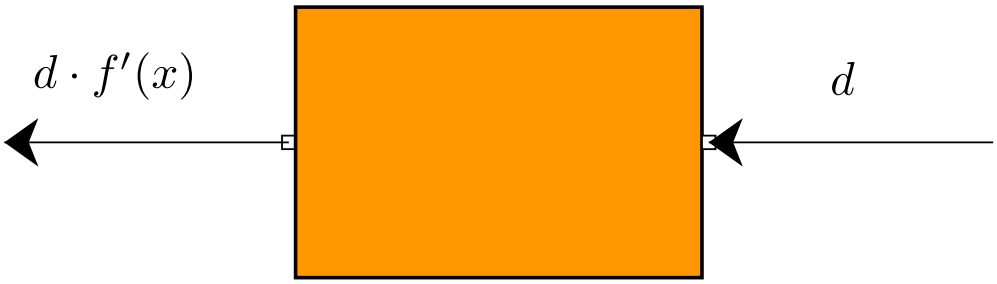
Backward

Our computation graph was made up of individual atomic functions $f(x)$. For each of these functions we are now going to implement a backward method to provide this local derivative information.

The API for backward is to compute $d \cdot f'(x)$ where $f'(x)$ is the derivative of the function and d is a value passed to backward (discussed more below).

For a simple function $f(x) = -x$, we can consult our derivative rules and get $f'(x) = -1$. Therefore the backward is,

```
class Neg(ScalarFunction):
    @staticmethod
    def forward(ctx, x):
        return -x

    @staticmethod
    def backward(ctx, d):
        f_prime = -1
        return f_prime * d
```



Note that backward works a bit different than the mathematical notation. Sometimes the function for the derivative $f'(x)$ depends directly on x; however, backward does not take x as an argument. This is where the context arguments `ctx` comes in.

Consider a function *Sin*, $f(x) = \sin(x)$ which has derivative $f'(x) = \cos(x)$. We need to write it in code as,

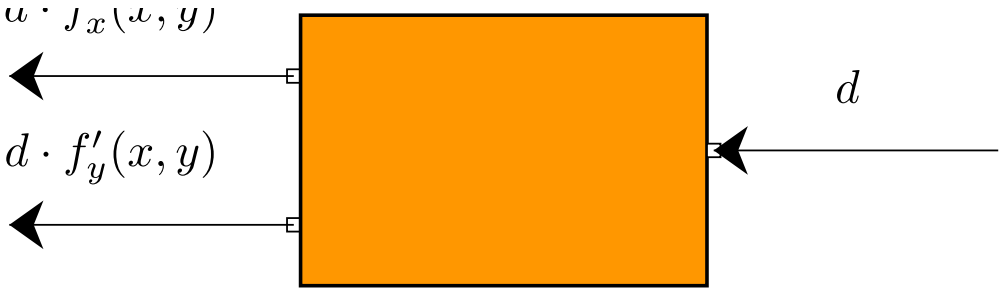
```
class Sin(ScalarFunction):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return math.sin(x)

    @staticmethod
    def backward(ctx, d):
        (x,) = ctx.saved_values
        f_prime = math.cos(x)
        return f_prime * d
```

```
def d_call(x):
    ctx = minitorch.Context()
    Sin.forward(ctx, x)
    return Sin.backward(ctx, 1)


plot_function("f(x) = sin(x)", lambda x: Sin.apply(x).data)
plot_function("1 * f'(x) = cos(x)", d_call)
```

For functions that take multiple arguments, backward returns derivatives with respect to each input argument. For example, if the function computes $f(x, y)$, we need to return $f'_x(x, y)$ and $f'_y(x, y)$

```
class Mul(ScalarFunction):
    @staticmethod
    def forward(ctx, x, y):
        ctx.save_for_backward(x, y)
```

```
        return x * y

    @staticmethod
    def backward(ctx, d):
        # Compute f'_x(x, y) * d, f'_y(x, y) * d
        x, y = ctx.saved_values
        f_x_prime = y
        f_y_prime = x
        return f_x_prime * d, f_y_prime * d
```



Chain Rule

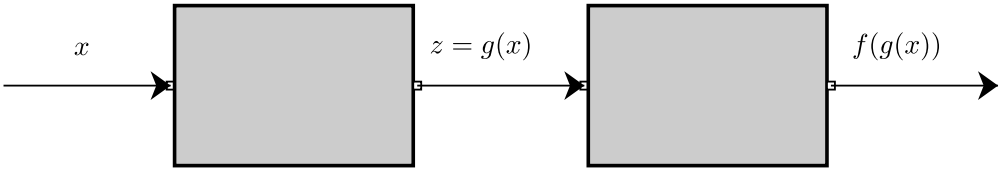
This section discusses implementation of the chain rule for univariate differentiation. Before reading, review the mathematical definition of Chain Rule.

Computing backward gives a way to compute the derivative for simple functions, but what if we have more complex functions? Let's go through each of the different cases to compute the derivatives.

- One argument
- Two argument
- Same argument

One argument

Let us say that we have a complex function $h(x) = f(g(x))$. We want to compute $h'(x)$. For simplicity we use $z = g(x)$, and draw h as two boxes left to right.

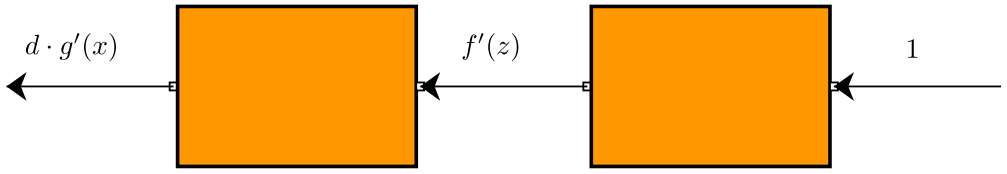


The chain rule tell us how to compute this term. Specifically it gives the following formula.

$$
\begin{aligned}
d &= 1 \cdot f'(z) \\
h'_x(x) &= d \cdot g'(x)
\end{aligned}
$$

The above derivative function tells us to compute the derivative of the right-most function (f), and then multiply it by the derivative of the left function (g).

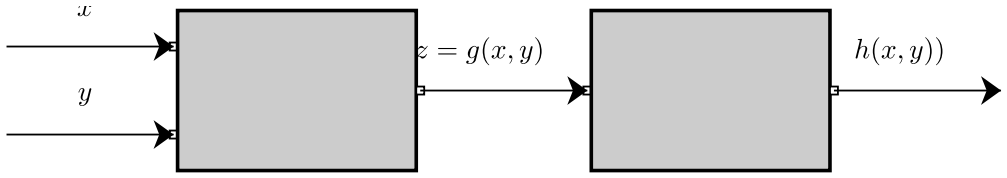
Here is where the perspective of thinking of functions as boxes pays off. We simply reverse the order.



The d multiplier passed to backward of the first box (left) should be the value returned by [backward]{.title-ref} of the second box. The 1 at the end is to start off the chain rule process with a value for d_{out}.

Two arguments

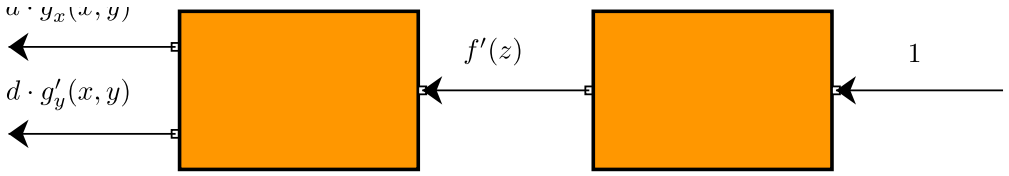
Next is the case of a two argument function. We will write this as $h(x, y) = f(g(x, y))$ where $z = g(x, y)$.



Applying the chain rule we get the following equations.

$$
\begin{aligned}
d &= 1 \cdot f'(z) \\
h'_x(x, y) &= d \cdot g'_x(x, y) \\
h'_y(x, y) &= d \cdot g'_y(x, y)
\end{aligned}
$$

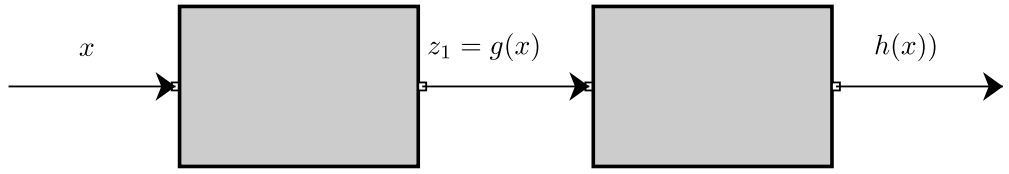
Drawing this again with boxes.



Note that this shows that the second box (f) does not care how many arguments the first box (g) has, as long as it passes back d which is enough for the chain rule to work.

Multiple Uses

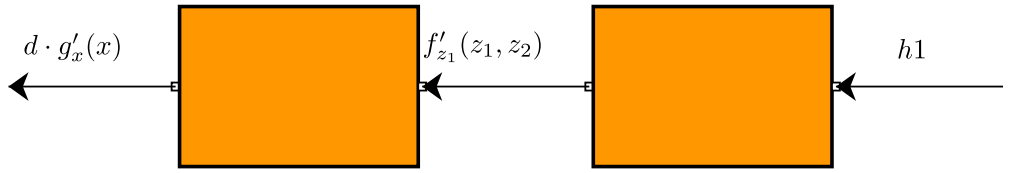
Finally, what happens when 1 value is used by two future boxes? Next is the case of a two argument function. We will write this as $h(x) = f(z_1, z_2)$ where $z_1 = z_2 = g(x)$.



Derivatives are linear, so the d term that comes from the second box is just the sum of the two individual derivatives.

$$d = 1 \cdot f'_{z_1}(z_1, z_2) + 1 \cdot f'_{z_2}(z_1, z_2)$$
$$h'_x(x) = d \cdot g'_x(x)$$

Specifically in terms of boxes, this means that if an output is used multiple times, we should sum together the derivative terms. This rule is important, as it means that we cannot call backward until we have aggregated together all the values that we need to calculate d.



Backpropagate

The `backward` function tells us how to compute the derivative of one operation. The chain rule tells us how to compute the derivative of two sequential operations. In this section, we show how to use these to compute the derivative for an arbitrary series of operations.

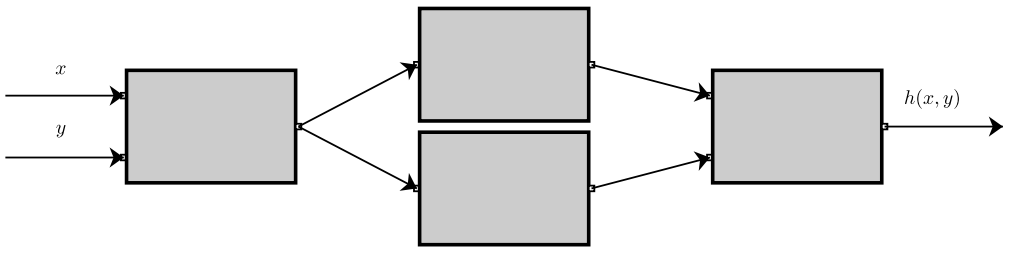
Practically this looks like re-running our graph in reverse order from right-to-left. However, we need to ensure that we do this in the correct order. The key implementation challenge of backpropagation is to make sure that we process each node in the correct order, i.e. we have first processed every node that uses a Variable before that varible itself.

Running Example

Assume we have Variables x, y and a Function $h(x, y)$. We want to compute the derivatives $h'_x(x, y)$ and $h'_y(x, y)$

$$h(x, y) = \log(z) + \exp(z)$$
$$\text{where} \quad z = x \times y$$

We assume x, +, log, and exp are all implemented as simple functions. This means that the final output Variable has constructed a graph of its history that looks like this:



Here, starting from the left, the first arrows represent inputs x, y, the left node outputs z, the top node $\log(z)$, the bottom node $\exp(z)$ and the final right node $h(x, y)$. Forward computation proceeds left-to-right.

The chain rule tells us methods for propagating the derivatives. We can use the rules from the previous section right-to-left until we reach the initial Variables x, y, i.e. the `leaf` Variables.

Topological Sort

We could just apply these rules randomly and process each nodes as they come aggregating the resulted values. However this can be quite inefficient. It is better to wait to call `backward` until we have accumulated all the values we will need.

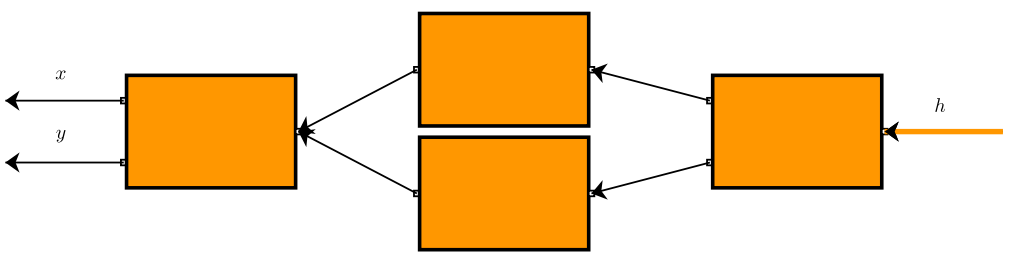
To handle this issue, we will process the nodes in `topological order`. We first note that our graph is directed and that acyclic. Directionality comes from the `backward` function, and the lack of cycles is a consequence of the choice that every Function must create a new variable.

The topological ordering of a directed acyclic graph is an ordering that ensures no node is processed after its ancestor, e.g. in our example that the left node cannot be processed before the top or bottom node. The ordering may not be unique, and it does not tell us whether to process the top or bottom node first.

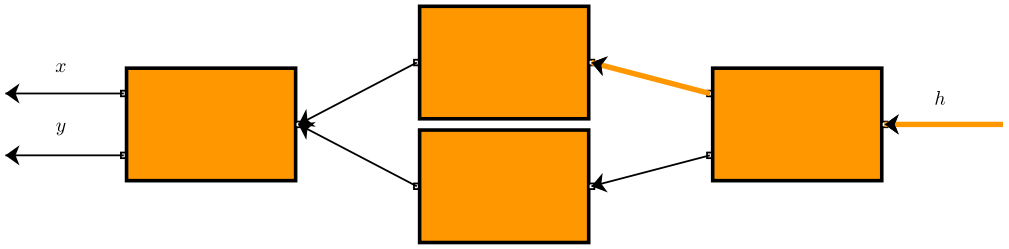
There are several easy-to-implement algorithms for topological sorting. As graph algorithms are beyond the scope of this document, we recommend using the depth-first search algorithm described in pseudocode section of Topological Sorting.

Backprop

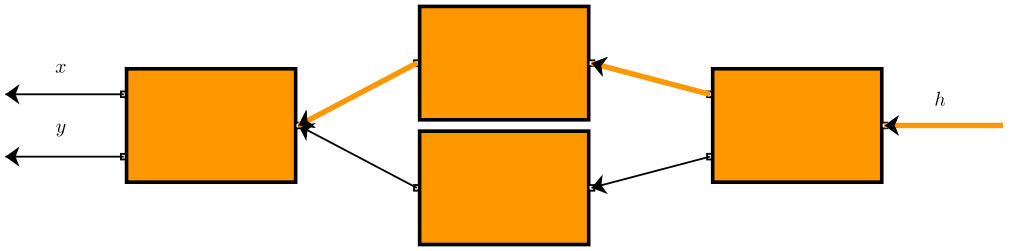
Once we have the order defined, we process each node one at a time in order. We start the rightmost node ($h(x, y)$) with red arrow in the graph below. The starting derivative is an argument given to us.



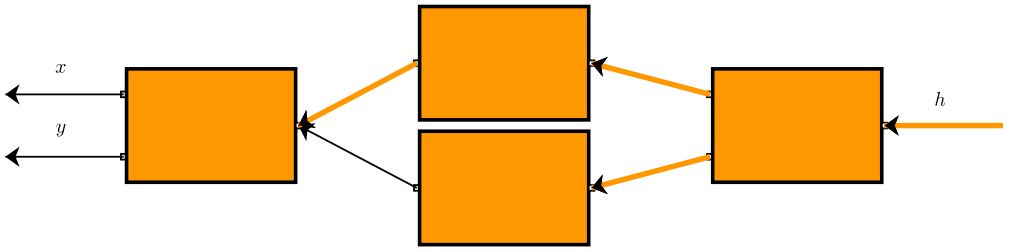
We then process the Function with the chain rule. This calls `backward` of +, and gives the derivative for the two red Variables (which correspond to $\log(z)$, $\exp(z)$ from the `forward` pass). You need to track these intermediate red derivative values in a dictionary.



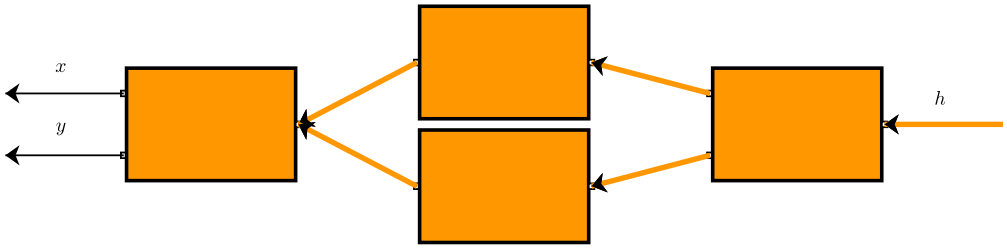
Let us assume the next Variable in the order is the top node. We have just computed and stored the necessary derivative d_{out}, so we can apply the chain rule. This produces a new derivative (corresponding to z: left red arrow below) for us to store.



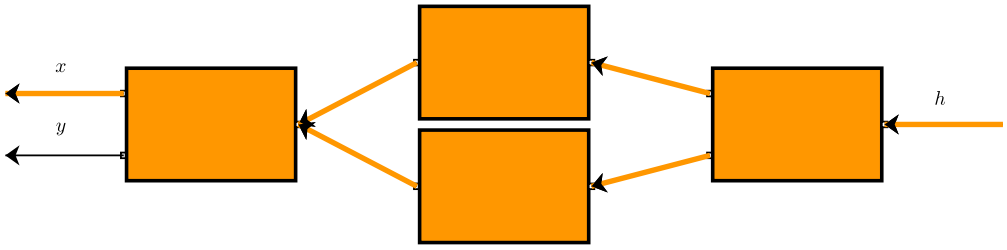
The next Variable in the order is the bottom node. Here we have an interesting result. We have a new arrow, but it corresponds to the same Variable (z) that we just computed. It is is a useful exercise to show that as a consequence of the two argument chain rule that the derivative for this Variable is the sum of each of these derivatives. Practically this means just adding it to your dictionary.



After working on this Variable, at this point, all that is left in the is our input leaf Variables.



When we reach the leaf variables in our order, for example x, we store the derivative with that variable. Since each step of this process is an application of the chain rule, we can show that this final value is $h'_x(x, y)$. The next and last step is to compute $h'_y(x, y)$.



By convention, the variables x, y have their derivatives stored as::

```
x.derivative, y.derivative
```

Algorithm

As illustrated in the graph for the above example, each of the red arrows represents a constructed derivative which eventually passed to d_{out} in the chain rule. Starting from the rightmost arrow, which is passed in as an argument, backpropagate should run the following algorithm:

0. Call topological sort to get an ordered queue
1. Create a dictionary of Scalars and current derivatives
2. For each node in backward order, pull a completed Scalar and derivative from the queue: a. if the Scalar is a leaf, add its final derivative (`accumulate_derivative`) and loop to (1) b. if the Scalar is not a leaf,
    a. call `.chain_rule` on the last function with d_{out}
    b. loop through all the Scalars+derivative produced by the chain rule

c. accumulate derivatives for the Scalar in a dictionary

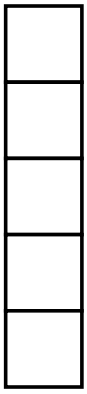
Final note: only leaf Scalars should ever have non-None `.derivative` value. All intermediate Scalars should only keep their current derivative values in the dictionary. This is a bit annoying, but it follows the behavior of PyTorch.

Tensors

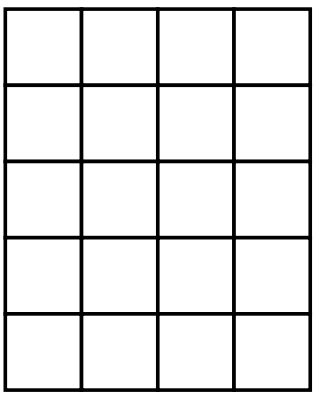
Tensor is a fancy name for a simple concept. A tensor is a multi-dimensional array of arbitrary dimensions. It is a convenient and efficient way to hold data, which becomes much more powerful when paired with fast operators and autodifferentiation.

Tensor Shapes

So far we have focused on scalars, which correspond to 0-dimensional tensors. Next, we consider a 1-dimensional tensor (vector):

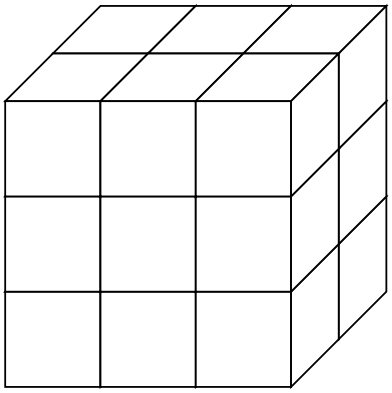


Then a 2-dimensional tensor (matrix):

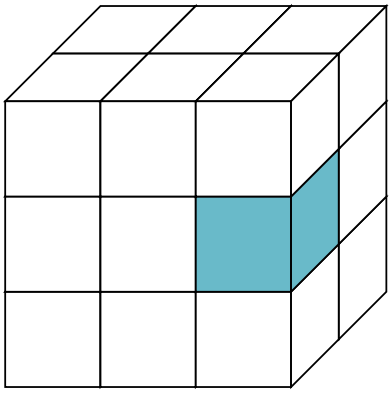


In addition to its dimension (`dims`), other critical aspects of a tensor are its `shape` and `size` . The shape of the above vector is (5,)) and its size (i.e. number of squares in the graph) is 5. The shape of the above matrix is (4,4) and its size is 20.

A three-dimensional tensor with shape (2, 3, 3) and size 18 looks like this:

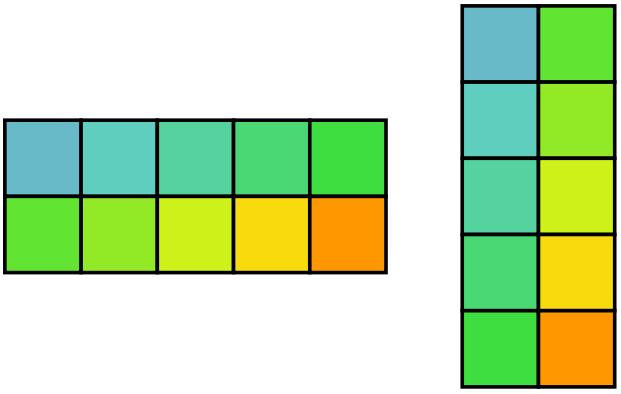


We access an element of the tensor by tensor index notation: `tensor[i]` for 1-dimension, `tensor[i, j]` for 2-dimension, `tensor[i, j, k]` for 3-dimension, and so forth. For example, `tensor[0, 1, 2]` would give this blue cube:

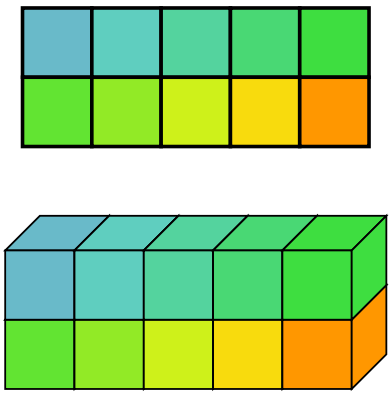


Typically, we access tensors just like multi-dimensional arrays, but there are some special geometric properties that make tensors different.

First, tensors make it easy to change the order of the dimensions. For example, we can `transpose` the dimensions of a matrix. For a general tensor, we refer to this operation as `permute` . Calling `permute` arbitrarily reorders the dimensions of the input tensor. For example, as shown below, calling `permute(1,0)` on a matrix of shape (2, 5) gives a matrix of shape (5, 2). For indexing into the permuted matrix, we access elements using `tensor[j, i]` instead of `tensor[i, j]` .



Second, tensors make it really easy to add or remove additional dimensions. Note that a matrix of shape (5, 2) can store the same amount of data as a matrix of shape (1, 5, 2), so they have the same size as shown below:



We would like to easily increase or decrease the dimension of a tensor without changing the data. We will do this with a `view` function: use `view(1, 5, 2)` for the above example. Element `tensor[i, j]` in the (5,2) matrix is now `tensor[0, i, j]` in the 3-dimensional tensor.

Critically, neither of these operations changes anything about the input tensor itself. Both `view` and `permute` are `tensor tricks`, i.e. operations that only modify how we look at the tensor, but not any of its data. Another way to say this is that they do not move or copy the data in any way, but only the external tensor wrapper.

Tensor Strides

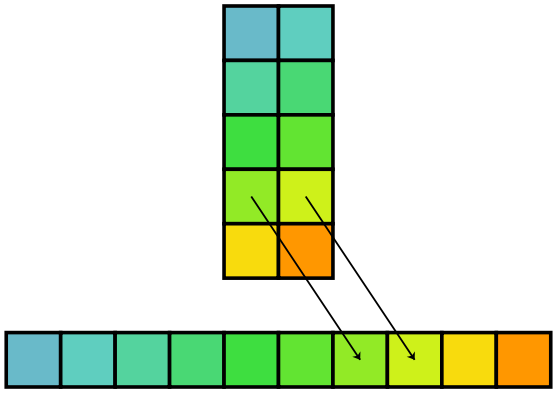
Users of a Tensor library only have to be aware of the `shape` and `size` of a tensor. However, there are important implementation details that we need to keep track of. To make our code a bit cleaner, we need to separate out the internal `tensor data` from

the user-facing tensor. In addition to the `shape`, :class:`minitorch.TensorData` manages tensor `storage` and `strides`:

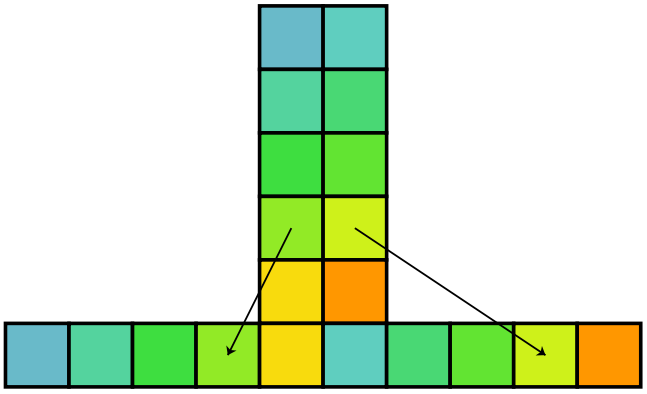
- **Storage** is where the core data of the tensor is kept. It is always a 1-D array of numbers of length `size`, no matter the dimensionality or `shape` of the tensor. Keeping a 1-D storage allows us to have tensors with different shapes point to the same type of underlying data.

- **Strides** is a tuple that provides the mapping from user indexing to the position in the 1-D `storage`.

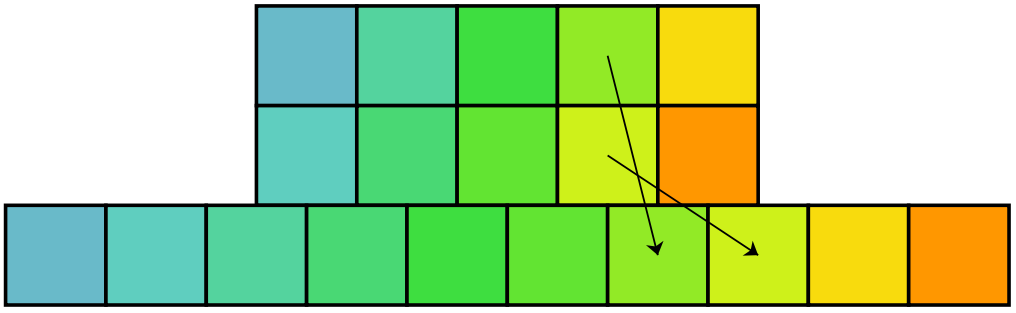
`Strides` can get a bit confusing to think about, so let's go over an example. Consider a matrix of shape (5, 2). The standard mapping is to walk left-to-right, top-to-bottom to order this matrix to the 1-D `storage`:



We call it `contiguous` mapping, since it is in the natural counting order (bigger strides left). Here the strides are $(2, 1)$. We read this as each column moves 1 step in storage and each row moves 2 steps. We can have different strides for the same shape. For instance, if we were walking top-to-bottom, left-to-right, we would have the following stride map:

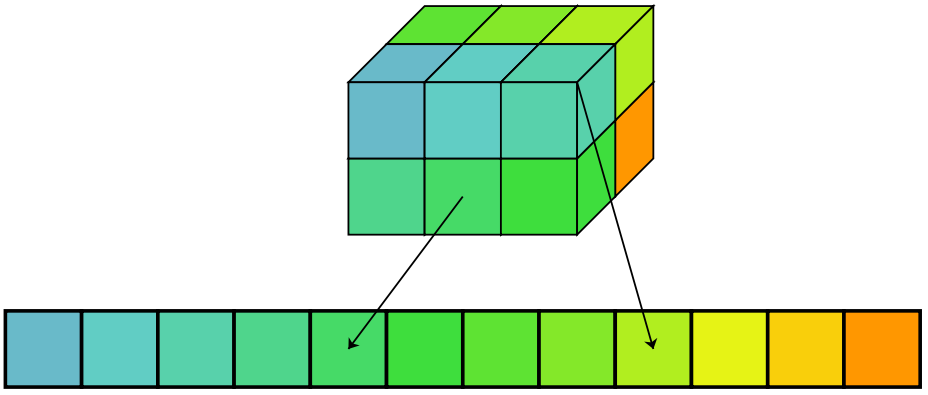


Contiguous strides are generally preferred, but non-contiguous strides can be quite useful as well. Consider transposing the above matrix and using strides (1,2):



It has new strides (1,2) and new shape (2,5), in contrast to the previous (2,1) stride map on the (5,2) matrix. But notably no change in the `storage`. This is one of the super powers of tensors mentioned above: we can easily manipulate how we view the same underlying `storage`.

Strides naturally extend to higher-dimensional tensors.



Finally, strides can be used to implement indexing into the tensor. Assuming strides are (s_1, s_2) and we want to look up `tensor[i, j]`, we can directly use strides to find its postion in the `storage`::

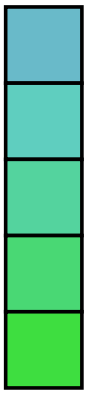
```
storage[s1 * i + s2 * j]
```

Or in general::

```
storage[s1 * index1 + s2 * index2 + s3 * index3 ... ]
```

Broadcasting

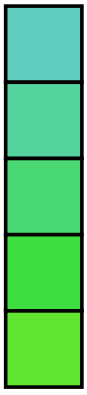
```
import chalk
```



Broadcasting makes tensors convenient and efficient to use, which comes in handy particularly for `zip` operations. So far all of our `zip` operations assume two input tensors of **exactly** the same size and shape. However there are many interesting cases to `zip` two tensors of different size.

Perhaps the simplest case is we have a vector of size 3 and want to add a scalar constant to every position::

```
# In math notation, vector1 + 1
vector1 + tensor([1])
```

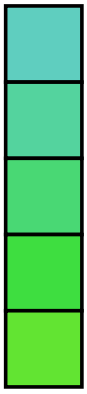


Intuitively, we would like to interpret this expression as the standard vector+scalar: adding 10 to each position. However, the above operation will fail because of shape

mistach: we are adding a tensor of shape(1,) to `vector1` which has shape (3,).

We could ask users to create a tensor of the same shape instead, but it is both annoying and, more importantly, inefficient::

```
vector1 + tensor([1, 1, 1, 1, 1]).view(5, 1)
```



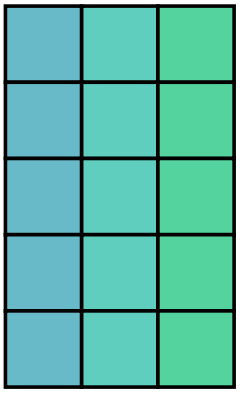
`Broadcasting` is a protocol that allows us to automatically interpret the frist expression as implying the second one. Inside `zip`, we pretend that 10 is a vector of shape (3,) when zipping it with a vector of shape (3,). Again, this is just an interpretation: we never actually create this vector.

This gives us the first rule of broadcasting:

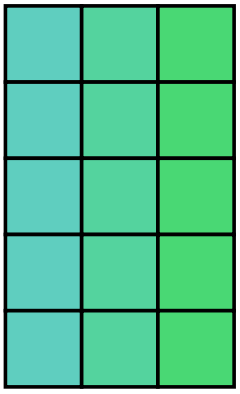
Rule 1: Any dimension of size 1 can be zipped with dimensions of size n > 1 by assuming the dimension is copied n times.

Now let's apply this approach to a matrix of shape (5, 3)::

```
matrix1
```



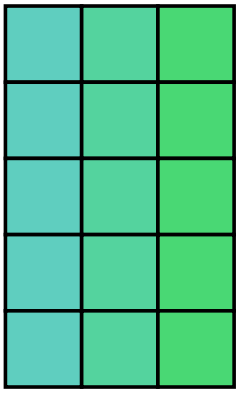
```
matrix1 + tensor([1])
```



Here we are trying to zip a matrix (2-D) of shape (5, 3) with a vector (1-D) of shape (1,). Here we are not just off on the shape, but also on the number of dimensions.

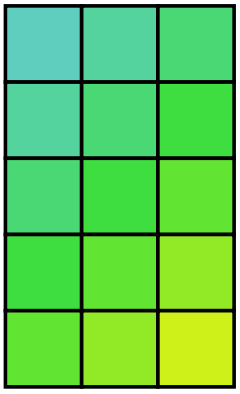
However, recall that adding an extra dimension of shape-1 doesn't change the size of the tensor. Therefore we can allow our protocol to add these in. Here if we add an empty dimension and then apply rule 1 twice, we can interpret the above expression as an efficient version of::

```
matrix1 + tensor([[1 for _ in range(3)] for _ in range(5)])
```



Rule 2: Extra dimensions of shape 1 can be added to a tensor to ensure the same number of dimensions with another tensor.

```
vector3 = tensor([1, 2, 3, 4, 5]).view(5, 1)
matrix1 + vector3
```

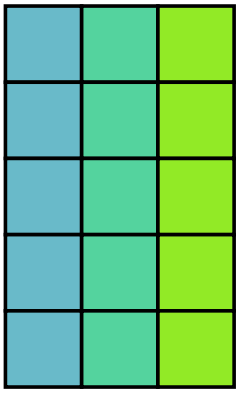


Finally, there is a question of where to add the empty dimension. This is not an issue in the above example but could become an issue in more complicated cases. Thus we introduce another rule:

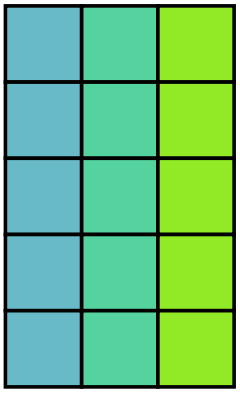
Rule 3: Any extra dimension of size 1 can only be implicitly added on the left side of the shape.

This rule has the impact of making the process easy to follow and replicate. You always know what the shape of the final output will be. For example

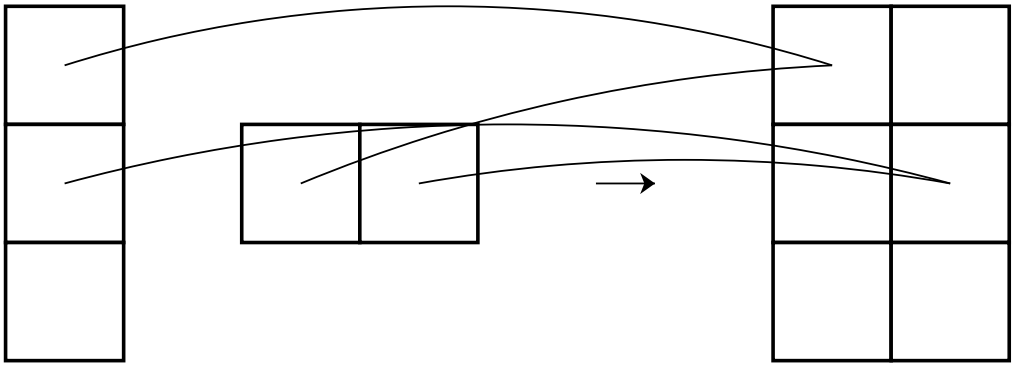
```
# This will fail: mismatch of (5, 3) and (3,)
vector2 = tensor([2, 2, 3])
matrix1 * vector2
```



```
# These two expression are equivalent
matrix1 * vector2
matrix1 * vector2.view(1, 3)
```



We can apply broadcasting as many times as we want::



Here is a more complicated example::

```
out = minitorch.zeros((2, 3, 1)) + minitorch.zeros((7, 2, 1, 5))
out.shape
```

```
(7, 2, 3, 5)
```

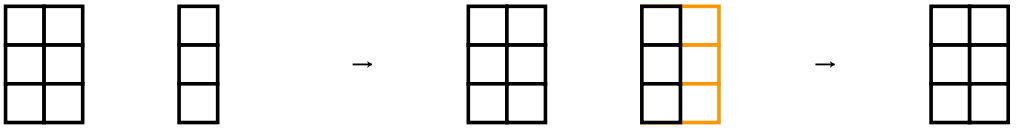
We end this guide with two important notes:

1. Broadcasting is **only** about shapes. It does not take strides into account in any way. It is purely a high-level protocol.
2. Broadcasting has some impact on the `backward` pass. We will discuss some in the code base, but it is not required for any of the tasks.

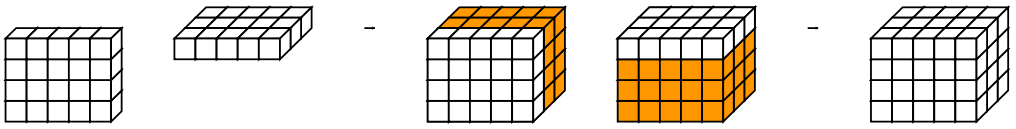
Examples

- Tensor-Scalar operations can be easily written using broadcasting for tensors of any dimension.

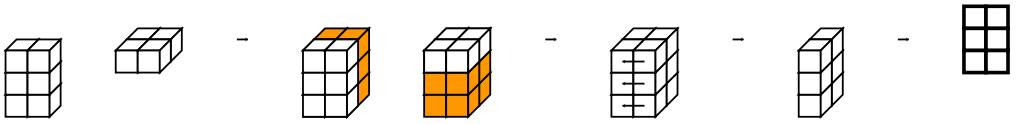
- Matrix-vector operations can be written using broadcasting, but you need to be careful to make sure that the vector is shaped such the the dimensions align. This can be done with `view` calls.



- Matrix-matrix operations can be written using broadcasting even when the dimensions don't align. Here is an example of that process.



- Matrix multiplication can be written in this style, here is (BxA^T) where A is 3 x 2 and B is 2 x 2. (And you will need to use this for the assignment). However, note this is a memory inefficient way to do matrix multiplication, as it needs to create an intermediate tensor in the process.

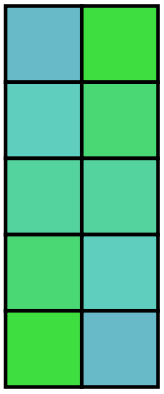


Operations

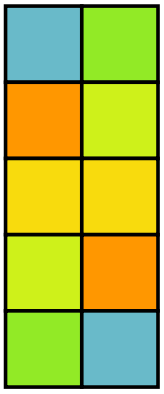
Now we would like to reimplement all our mathematical operations on tensors. The goal is to make this feel simple and intuitive to users of the library. We can break these operations down as unary transformations

a) Return a new tensor with the same shape as `tensor_a` where each position is the log/exp/negative of the position in `tensor_a`

```
tensor_a
```



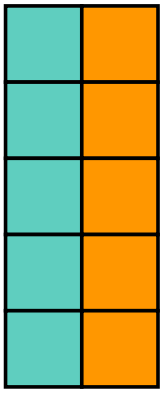
```
-tensor_a
```



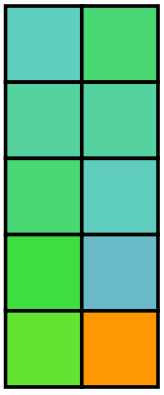
Binary transformations::

b) Return a new tensor where each position is the sum/mul/sub of the position in `tensor_a` and `tensor_b`

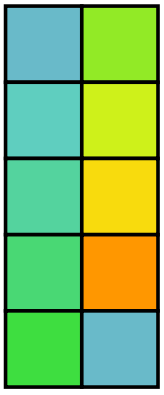
```
tensor_b = minitorch.tensor([[1, 1, 1, 1, 1], [-1, -1, -1, -1,
-1]]).permute(1, 0)
tensor_b
```



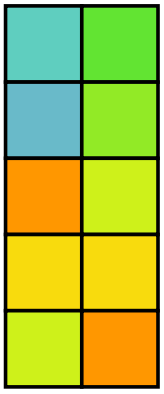
```
tensor_a + tensor_b
```



```
tensor_a * tensor_b
```



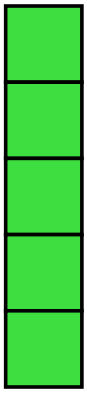
```
tensor_b - tensor_a
```



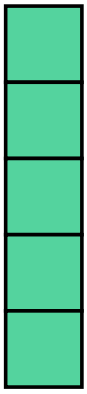
And reductions::

c) Return a new tensor where dim-1 is size 1 and represents the sum/mean over dim-1 in `tensor_a`

```
tensor_a.sum(1)
```



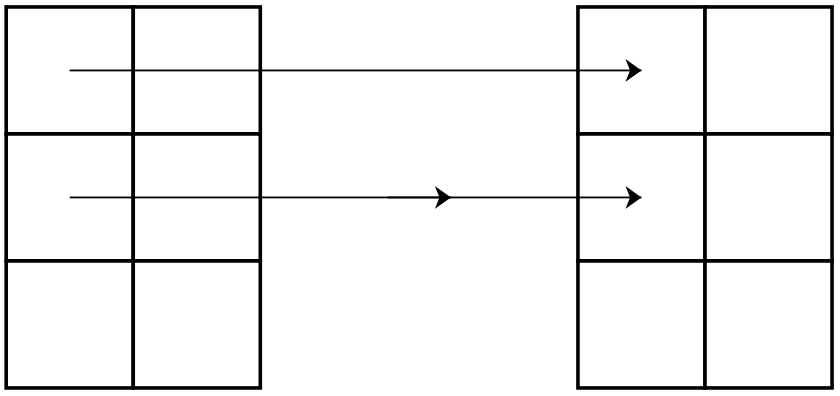
```
tensor_a.mean(1)
```



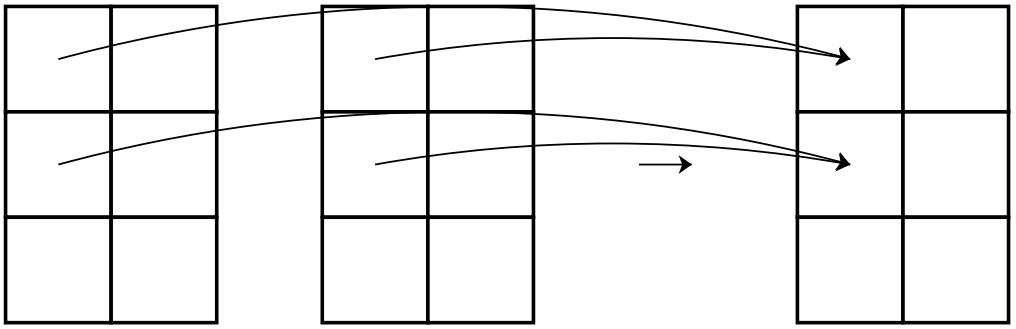
Core Operations

We could implement each of these operations individually, but we can also be a bit lazy and note the structural similarities. If we squint, these operations look very much like the higher-order functions that we implemented in module0:

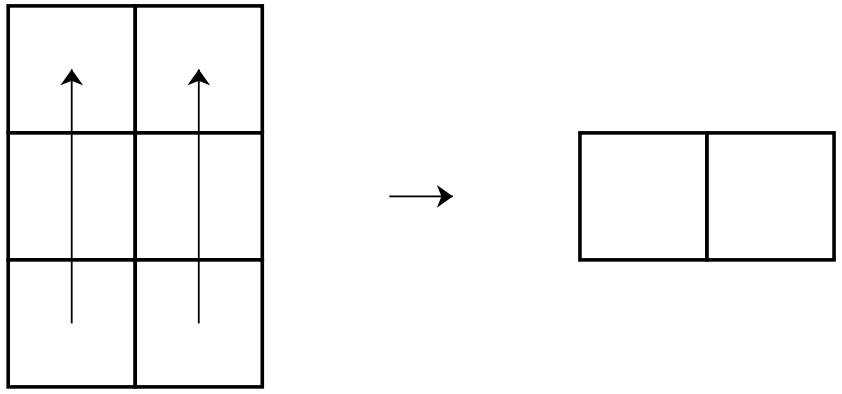
- Operation a / map: These operations just touch each of the positions in the tensor individually. They don't need to deal with other positions or know anything about the shape or size of the tensor. We can view these operations as applying the following transformation:



- Operation b / zip: These operations only need to pair operations between input tensors. If we assume the tensors have the same size and shape, this type of operation simply aligns these two tensors and applies an operator to each pair of elements:



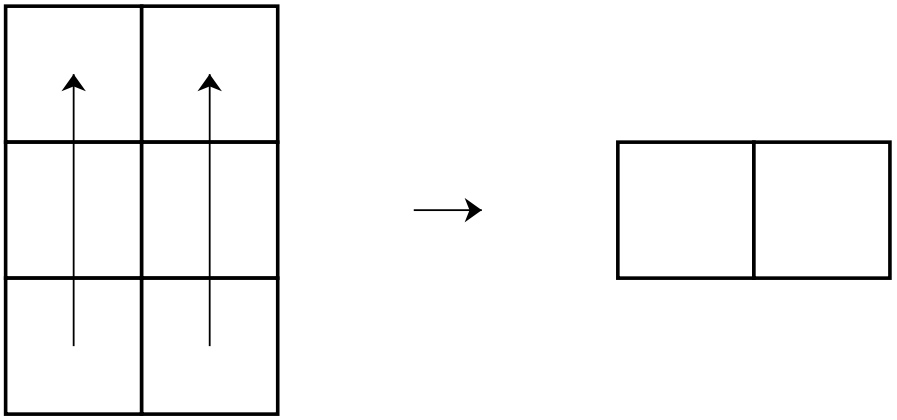
- Operation c / reduce: These operations need to group together cells within a single tensor. We can think of there being an implied `reduce` shape that is eliminated in the process of the output construction. For instance, in the example below, we start with an input of shape (3, 2) and create an output of shape (1, 2). Implicitly, we reduce over a tensor of shape (3, 1) for each element in the output.

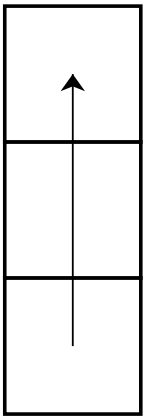


Reductions

Reduction is a bit more complex than the others, so let's discuss how it is implemented.

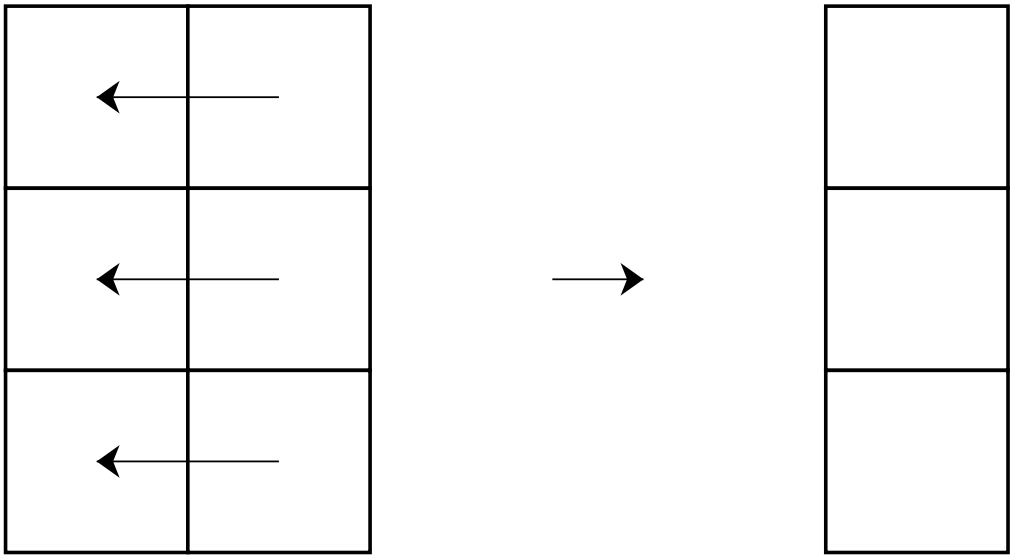
Reductions can specify a dimension (or axis) that tells us which elements to reduce. The reduction is then applied along that dimension. For example if we `reduce` dimension 0 we get the following reduction.

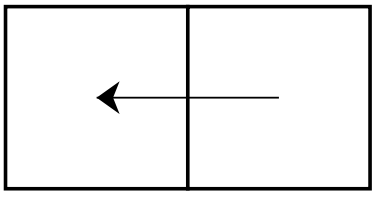




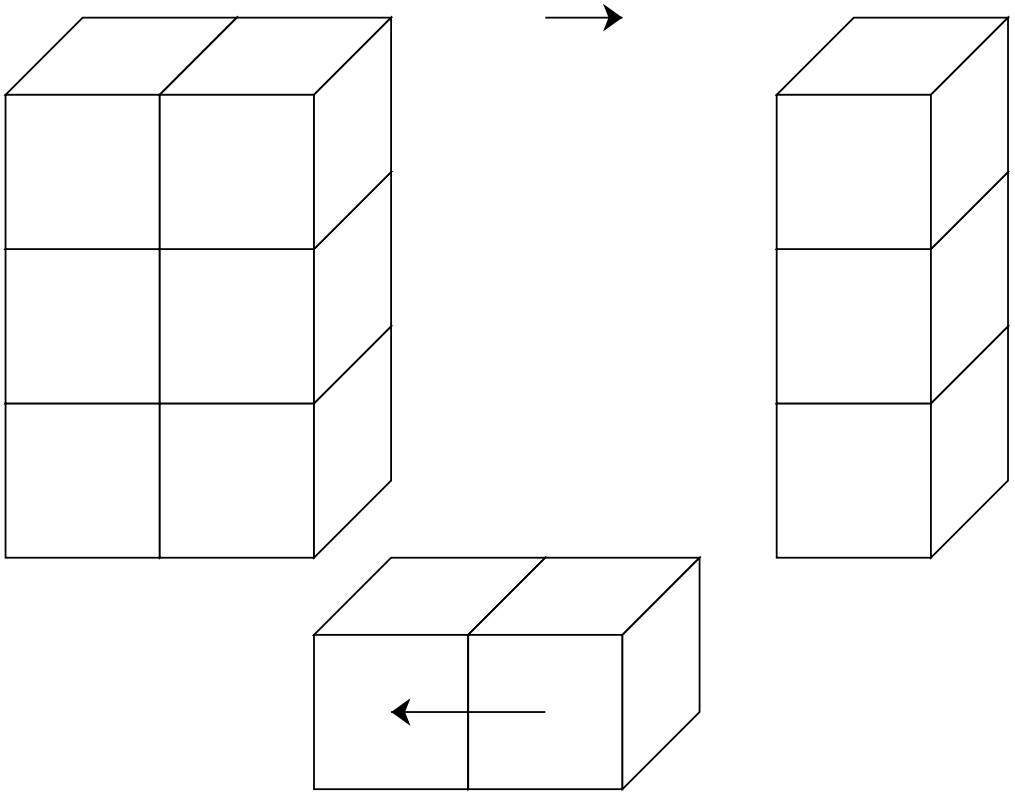
This reduction changes our shape from (3, 2) to (1, 2), i.e. `reducing` the size of 0-dim. Another way you can view this is as 2 parallel operations, both of which apply a Module-0 style reduce along the 0-dim. We can look at the reduction shape of this procedure (3, 1).

Applying a reduction along dim 1 creates a different reduction shape which is applied 3 times. Here is what that looks like,

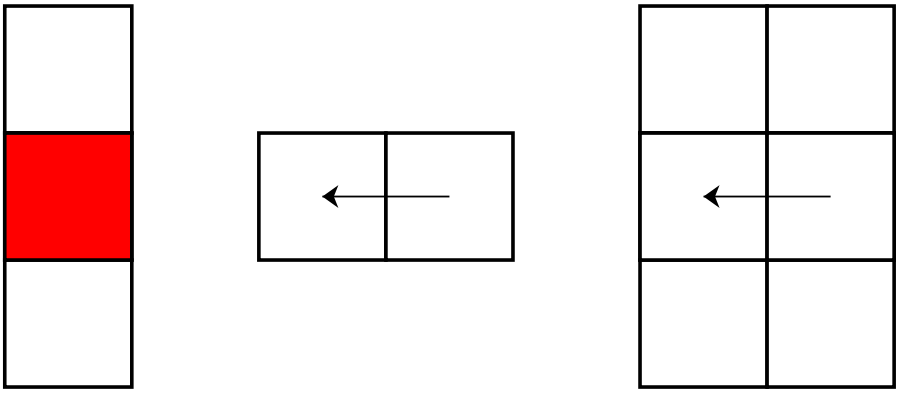




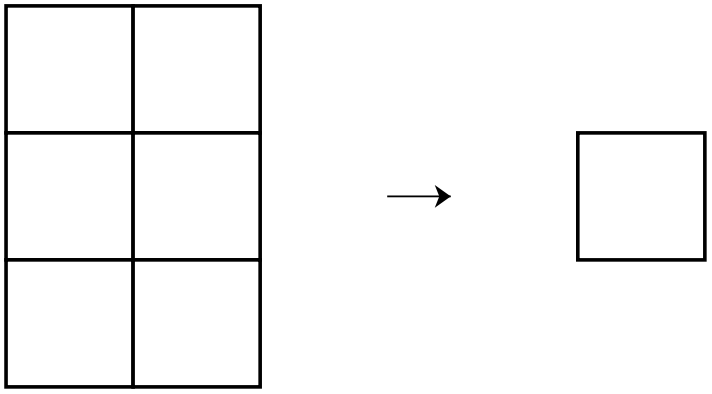
The same approach can be applied in higher dimensions. If we want to sum over one of the dimensions we just create the reduction shape and apply it,



When you implement, think about enumerating over positions in the final tensor, and then applying the reduction shape to get the indices over the original tensor,



Finally, there is a special case reduction where we reduce over the entire tensor. You can think of this as creating a reduction shape over the full tensor and then viewing the result as a scalar.



In the next module, we will discuss efficient implementation of the above operations. For now, they can be implemented by slow but correct loops over all elements in the input tensors. This approach can be used to implement key tensor operations, without doing more than implementing the above higher-order functions.

Auto-Grad

Next, we consider autodifferentiation in the tensor framework. We have now moved from scalars and derivatives to vectors, matrices, and tensors. This means *multivariate* calculus can bring into play some more terminology. However, most of what we actually do does not require complicated terminology or much technical math. In fact, except some name changes, we have already built almost everything we need in Module 1.

The key idea is, just as we had `Scalar` and `ScalarFunction`, we need to construct `Tensor` and `TensorFunction` (which we just call `Function`). These new objects behave very similar to their counterparts:

a) Tensors cannot be operated on directly, but need to be transformed through a function. b) Functions must implement both `forward` and `backward`. c) These transformations are tracked, which allow backpropagation through the chain rule.

All of this machinery should work out of the box.

The main new terminology to know is *gradient*. Just as a tensor is a multidimensional array of scalars, a gradient is a multidimensional array of derivatives for these scalars. Consider the following code::

Scalar auto-derivative notation

```
def f(a, b, c):
    return a + b + c
```

```
a, b, c = Scalar(1), Scalar(2), Scalar(3)
out = f(a, b, c)
out.backward()
a.derivative, b.derivative, c.derivative
```

```
(1.0, 1.0, 1.0)
```

Tensor auto-gradient notation

```
tensor1 = tensor([1, 2, 3])
out = tensor1.sum()
out.backward()
# shape (3,)
tensor1.grad
```

```
[1.00 1.00 1.00]
```

```
tensor1.grad.shape
```

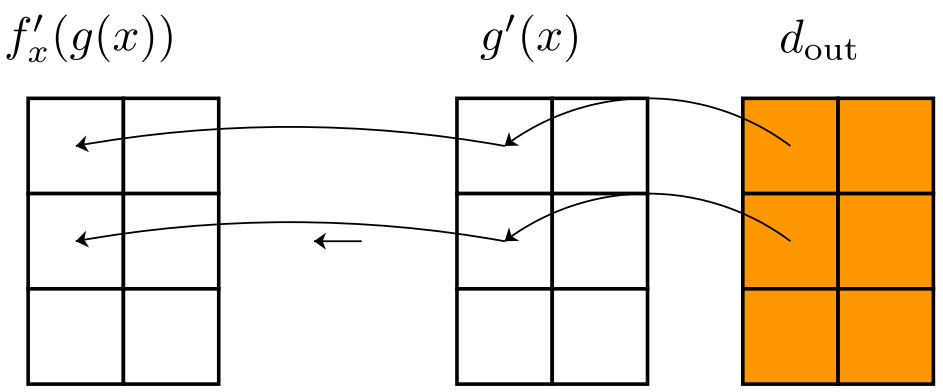
```
(3,)
```

The gradient of `tensor1` is a tensor that holds the derivatives of each of its elements. Another place that gradients come into play is that `backward` no longer takes d_{out} as an argument, but now takes $grad_{out}$ which is just a tensor consisting of all the d_{out}.

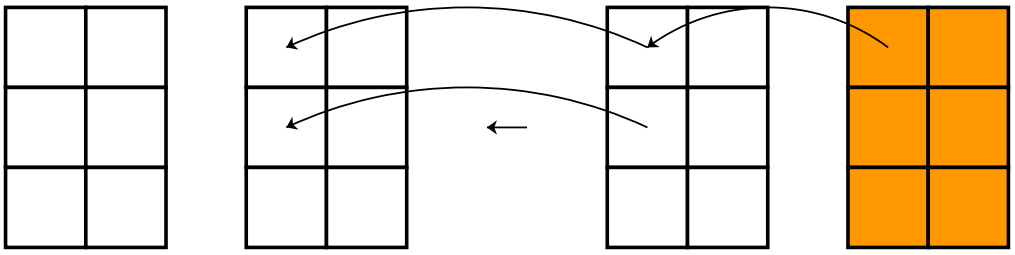
Note: You will find lots of different notation for gradients and multivariate terminology. For this Module, you are supposed to ignore it and stick to everything you know about derivatives. It turns out that you can do most of machine learning without ever thinking in higher dimensions.

If you think about gradient and $grad_{out}$ in this way (i.e. tensors of derivatives and d_{out}), then you can see how we can easily compute the gradient for tensor operations using univariate rules.

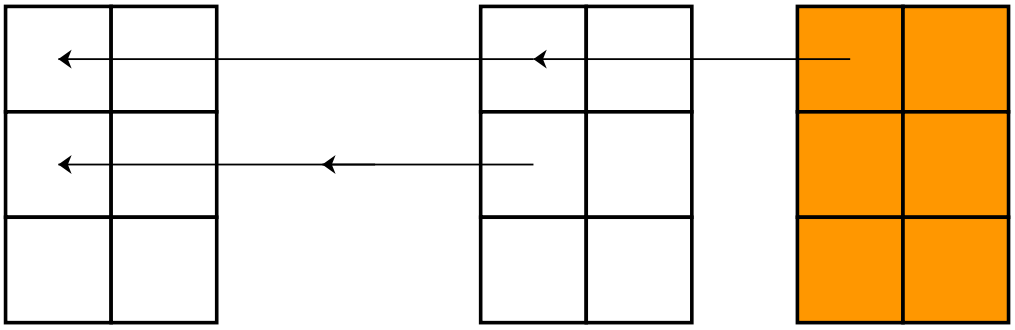
1. **map**. Given a tensor, `map` applies a univariate operation to each scalar position individually. For a scalar x, consider computing $g(x)$. From Module 1, we know that the derivative of $f(g(x))$ is equal to $g'(x) \times d_{out}$. To compute the gradient in `backward`, we only need to compute the derivative for each scalar position and then apply a `mul` map.



2. **zip**. Given two tensors, `zip` applies a binary operation to each pair of scalars. For two scalars x and y, consider computing $g(x, y)$. From Module 1, we know that the derivative of $f(g(x, y))$ is equal to $g'_x(x, y) \times d_{out}$ and $g'_y(x, y) \times d_{out}$. Thus to compute the gradient, we only need to compute the derivative for each scalar position and apply a `mul` map.



3. **reduce**. Given a tensor, `reduce` applies an aggregation operation to one dimension. For simplicity, let's consider sum-based reductions. For scalars x_1 to x_n , consider computing $x_1 + x_2 + \ldots + x_n$. For any x_i value, the derivative is 1. Therefore, the derivative for any position computed in `backward` is simply d_{out}. This means to compute the gradient, we only need to send d_{out} to each position. (For other reduce operations such as `product` , you get different expansions, which can be calculated just by taking derivatives).



Parallel Computation

```
from numba import njit, prange
```

The major technique we will use to speed up computation is parallelization. General parallel code can be difficult to write and get correct. However, we can again take advantage of the fact that we have structured our infrastructure around a set of core, general operations that are used throughout the codebase.

Let's start by going back to `module0` and our `map` operation:

There are many ways to implement this function. Let's consider a version where we pass it an explict output list to fill in. Here's a simple implementation

```
def simple_map(fn):
    def _map(out, input):
        for i in range(len(out)):
            out[i] = fn(input[i])

    return _map
```

This function produces the correct result, but it's not very efficient. Ideally, we could do something fast to take advantage of the fact that all the function calls are identical. Also, we shouldn't have to loop over the list one value at a time. By definition, `map` (and `zipWith`) can be fast and parallelized, since none of the individual computations interact with each other.

Python itself doesn't have great fuctions for fast math and parallelism built-in, but luckily it has good libraries to help speed up numerical computation. We will utilize one of these libraries known as Numba.

Numba JIT

Numba is a numerical JIT compiler for Python. When a function is first created, it converts raw Python code to faster numerical operations under the hood. We have seen an example of this library earlier when developing our mathematical operators.

```
def neg(x):
    return -x
```

This JIT function alone does not make the code much faster, but it allows us to use this code within other code. For instance, if we want to improve our `map` implementation, we can change it to look like this:

```
def map(fn):
    # Change 1: Move function from Python to JIT version.
    fn = njit()(fn)

    def _map(out, input):
        for i in range(len(out)):
            out[i] = fn(input[i])

    # Change 2: Internal _map must be JIT version as well.
    return njit()(_map)
```

Note that all JIT happens when outer map is first called.

```
neg_map = map(neg)
```

When the above function is called, instead of running slow Python code, it will run fast low-level code that takes advantage of the structure. This approach requires a bit of overhead on startup, but can make things much faster.

Furthermore, if we know that the loop can be done in parallel, we can speed it up further with one small change.

```
def map(fn):
    fn = njit()(fn)

    def _map(out, input):
        # Change 3: Run the loop in parallel (prange)
        for i in prange(len(out)):
            out[i] = fn(input[i])

    return njit(parallel=True)(_map)
```

What's neat about this is that the above code is basically the same as the Python code without parallelization. You can switch back and forth between the two without much change.

Warning: You have to be a bit careful to ensure that the loop actually can be parallelized. In short, this means that steps cannot depend on each other and each iteration cannot write to the same output value. For instance, when implementing `reduce`, you have to be careful to mix parallel and non-parallel loops.

For full details on how Numba works, read this tutorial.

Fusing Operations

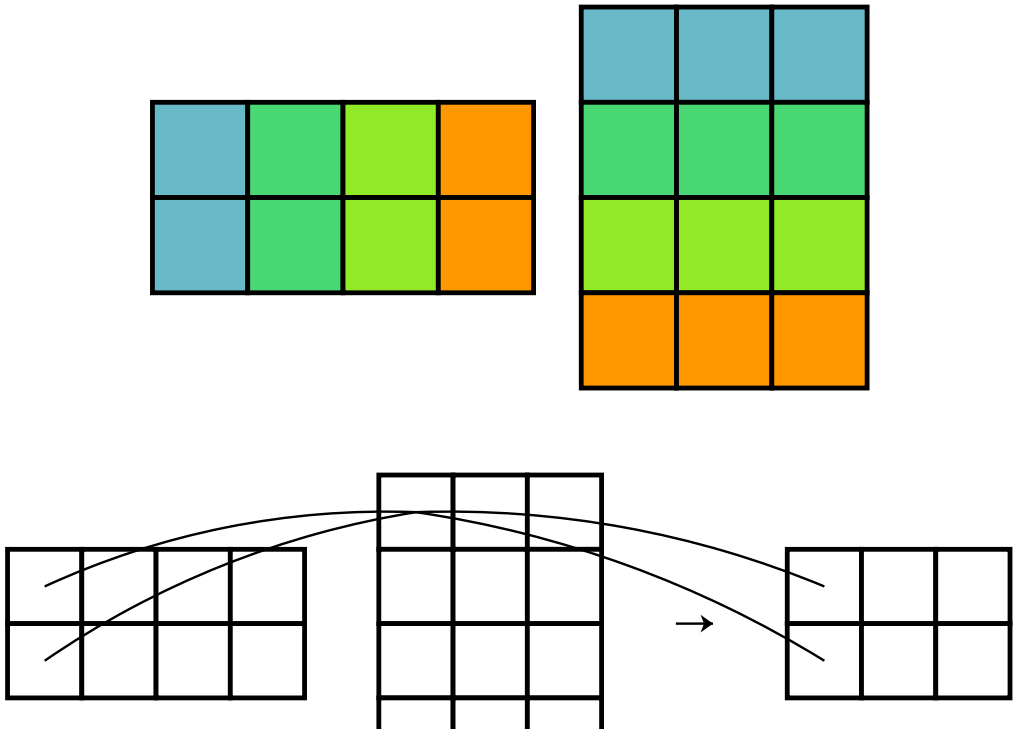
Another approach we can use to help improve tensor computations is to specialize commonly used combinations of operators. Combining operations can eliminate unnecessary intermediate tensors. This is particularly helpful for saving memory.

There is a lot of ongoing work on how to do this operator fusion automatically. However, for very common operators, it is worth just writing these operators directly and speeding them up.

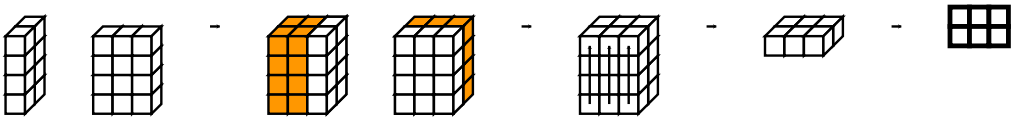
In minitorch we will do this fusion by customizing special operations. If we find it useful, we will add these operations in Numba

Example: Matrix Multiplication

Let's consider a matrix multiplication example. Recall that the rows of the first matrix interact with the columns of the second.

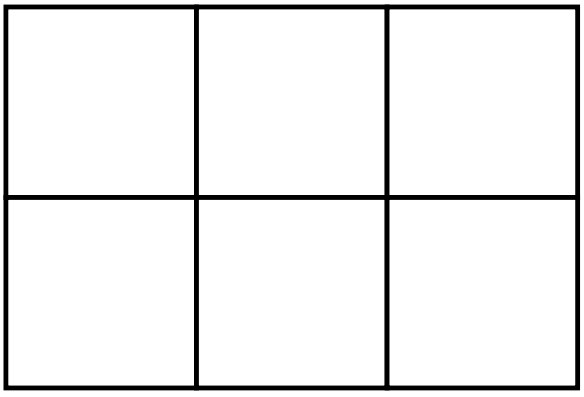


In past modules we have done matrix multiplication by applying a broadcasted `zip` and then a `reduce`. For example, consider a tensor of size (2, 4) and a tensor of size (4, 3). We first zip these together with broadcasting to produce a tensor of size (2, 4, 3):



And then reduce it to a tensor of size (2, 1, 3), which we can view as (2, 3):

```
matrix(2, 3)
```



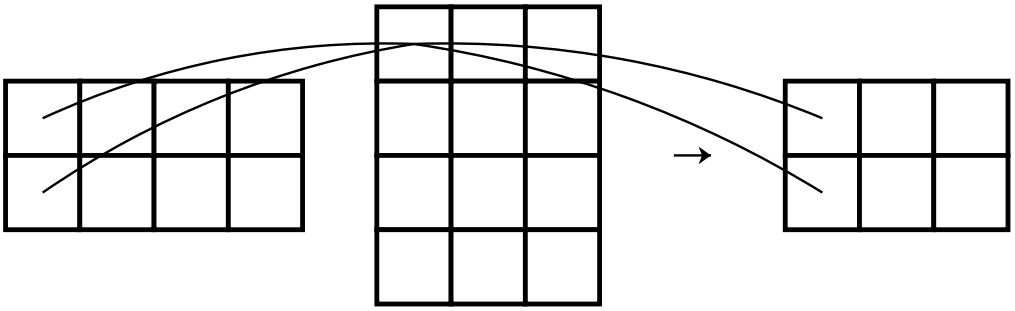
This computation is correct, but it has two issues. First, it needs to create a tensor of size (2,4,3) in the intermediate step. These tensor can be an order of magnitude larger than either of the tensors that were passed in. Another more subtle problem is that the `reduce` operator may need to call `save_for_backwards` on this tensor, which means it stays in memory throughout all of our forward computations.

An alternative option is to fuse together these two operations. We will create a single matmul operations using python `@` operator. We can then skip computing the intermediate value and directly compute the output. We can do this by writing a specialized tensor `Function` with a `forward` function that directly produces the output and a `backward` that produces the required gradient.

This allows us to specialize our implementation of matrix multiplication. We do this by 1) walking over the output indices, 2) seeing which indices are reduced, and 3) then seeing which were part of the zip.

Given how important this operator is, it is worth spending the time to think about how to make each of these steps fast. Once we know the output index we can very efficiently walk through the input indices and accumulate their sums.

A fast matrix multiplication operator can be used to compute the `forward`. To compute backward we need to reverse the operations that we computed. In particular this requires zipping grad out with the alternative input matrix. We can see the `backward` step by tracing the forward arrows:



It would be a bit annoying to optimize this code as well, but again luckily, to compute the `backward` step , we can reuse our forward optimization. In fact we can simply use the following identity from matrix calculus that tells us backward can be computed as a transpose and another matrix multiply.

$$
\begin{aligned}
f(M, N) &= MN \\
g'_M(f(M, N)) &= dN^T \\
g'_N(f(M, N)) &= M^T d
\end{aligned}
$$

GPU Programming

CPU parallelization and operator fusion is important, but when you really need efficiency and scale, specialized hardware is critical. It is really hard to exaggerate how important GPU computation is to deep learning: it makes it possible to run many models that would have been intractable even just several years ago.

Writing code of GPUs requires a bit more work than the CPU parallelization examples. GPUs have a slightly different programming model than CPUs, which can take some time to fully understand. Luckily though, there is a nice Numba library extension that allows us to code for GPUs directly in Python.

Getting Started

For Module 3, you will need to either work in an environment with a GPU available or utilize Google Colab. Google Colab provides free GPUs in a Python notebook setting. You can change the environment in the menu to request a GPU server.

We recommend working in your local setup and then cloning your environment on to a notebook:

```
>>> git clone {GITHUB_PATH}
>>> pip install -r requirements.txt
>>> pip install -e .
```

You can run your tests with the following command:

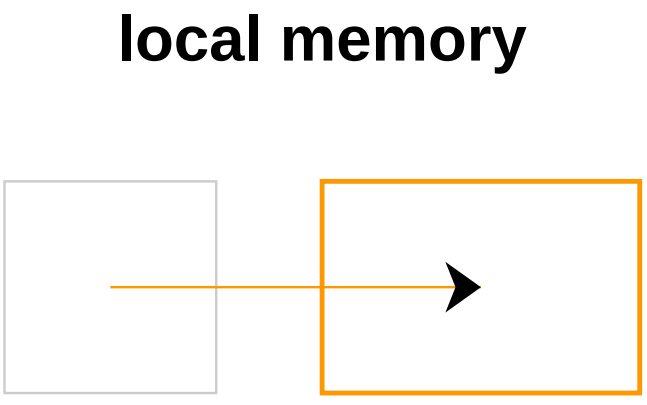
```
>>> python -m pytest -m task3_3
```

CUDA

The most commonly used programming model for GPUs in deep learning is known as CUDA. CUDA is a proprietary extension to C++ for Nvidia devices. Once you fully understand the terminology, CUDA is a relatively straightforward extension to the mathematical code that we have been writing.

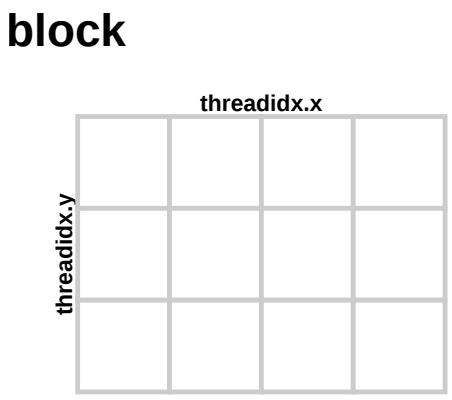
The main mechanism is `thread`. A thread can run code and store a small amount of states. We represent a thread as a little robot:

Each thread has a tiny amount of fixed local memory it can manipulate, which has to be *constant* size:

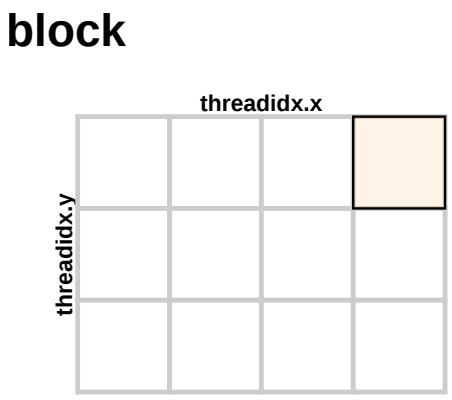


Threads hang out together in `blocks`. Think of these like a little neighborhood. You can determine the size of the blocks, but there are a lot of restrictions. We assume there are less than 32 threads in a block:

You can also have square or even cubic blocks. Here is a square block where the length and width of the neighborhood are the block size:

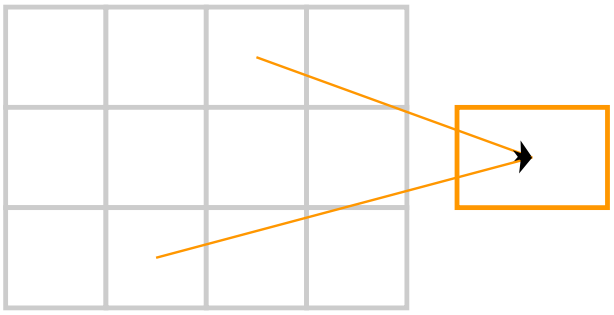


Each thread knows exactly where it is in the block. It gets this information in local variables telling it the `thread index`.

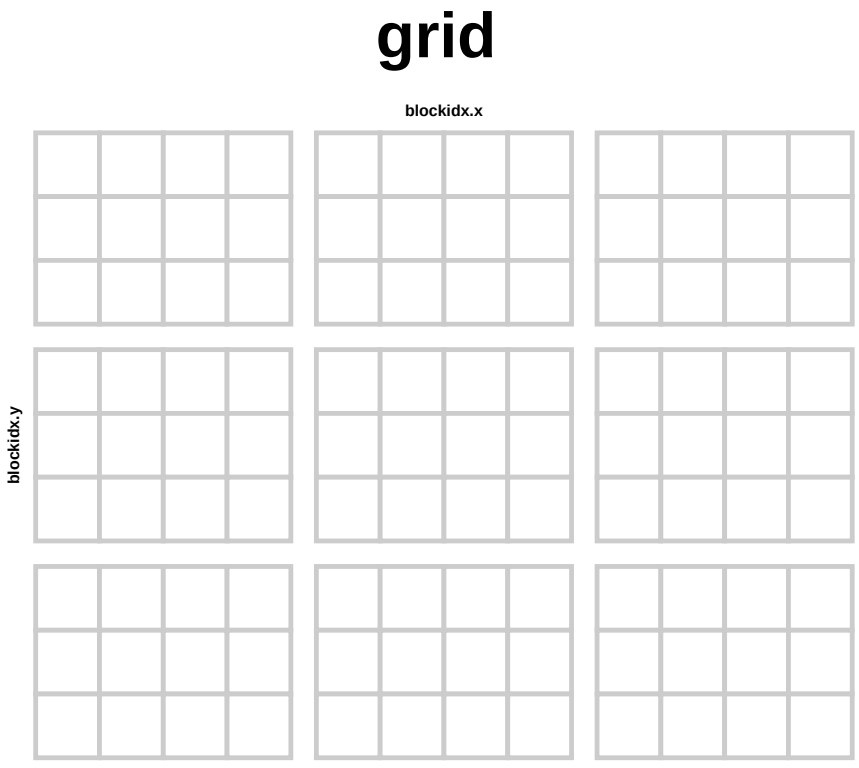


Threads in the same block can also talk to each other through `shared memory`. This is another constant chunk of memory that is associated with the block and can be accessed and written to by all of these threads:

block shared memory



Blocks come together to form a `grid`. Each of the blocks has exactly the same size and shape, and all have their own shared memory. Each thread also knows its position in the global grid:

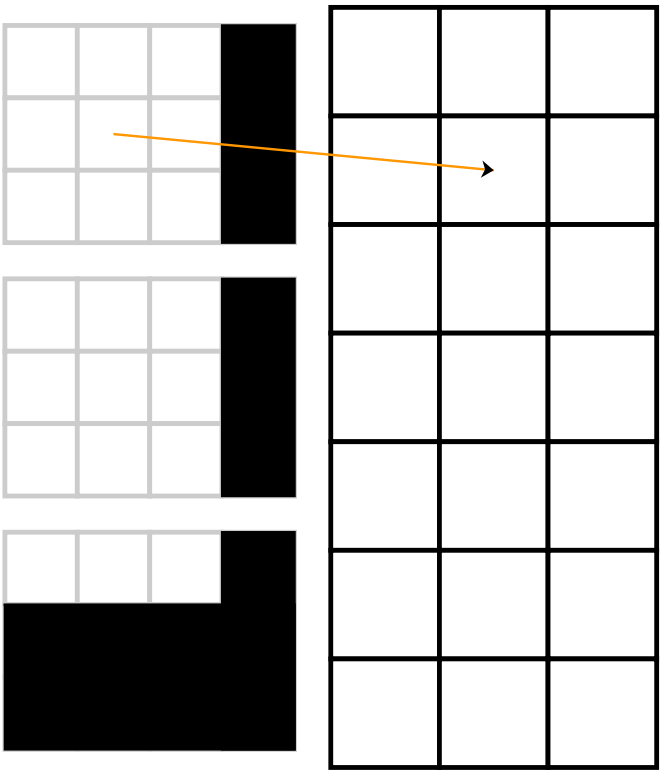


For instance, we can compute the global position `x, y` for a thread as::

```
def kernel():
    x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y
    print(x, y)
```

Now here comes the interesting part. When you write code for CUDA, you have to code all of the threads with the same code at the same time. Each thread behaves in

lockstep running the same function:



In Numba, you can write the thread instructions as a single function::

```
# Helper function to call in CUDA
# @cuda.jit(device=True)
def times(a, b):
    return a * b
```

Main cuda launcher

```
# @cuda.jit()
def my_func(input, out):
    # Create some local memory
    local = cuda.local.array(5)

    # Find my position.
    x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y

    # Compute some information
    local[1] = 10
```

```
    # Compute some global value
    out[x, y] = times(input[x, y], local[1])
```

Note that we cannot call the above function directly: we need to `launch` it with instructions for how to set up the blocks and grid. Here is how you do this with Numba::

```
threadsperblock = (4, 3)
blockspergrid = (1, 3)
my_func[blockspergrid, threadsperblock](in, out)
```

This sets up a block and grid structure similar to the `map` function mentioned earlier. The code in `my_func` is run simultaneously for all the threads in the structure. However, you have to be a bit careful as some threads might compute values that are outside the memory of your structure

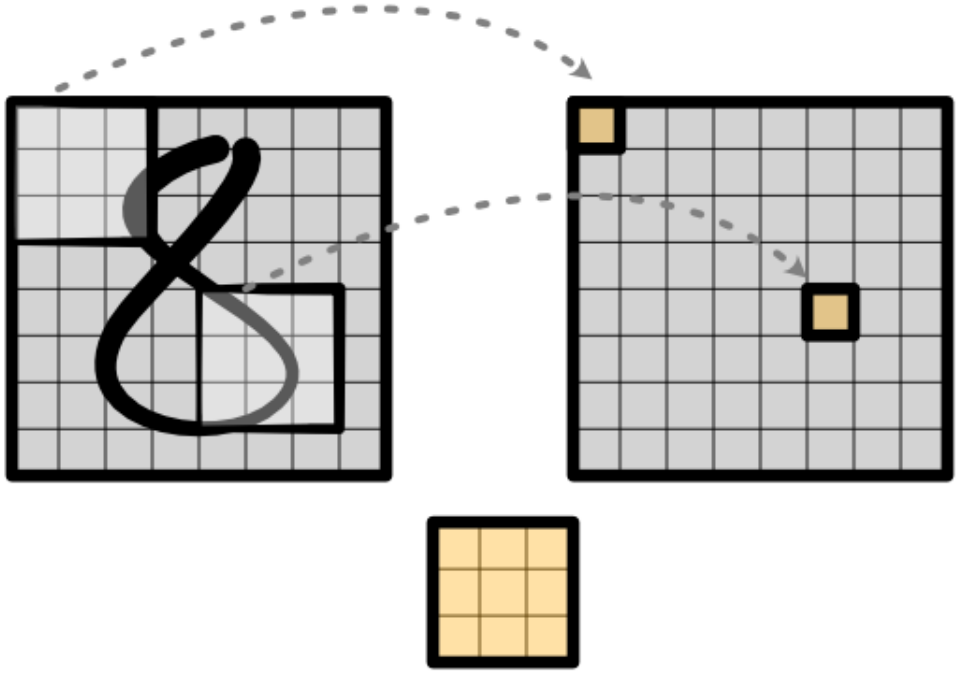
Main cuda launcher

```
# @cuda.jit()
def my_func2(input, out):
    # Create some local memory
    local = cuda.local.array(5)

    # Find my position.
    x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y

    # Compute some information
    local[1] = 10

    # Guard some of the threads.
    if x < out.shape[0] and y < out.shape[1]:
        # Compute some global value
        out[x, y] = times(input[x, y], local[1])
```

Convolution

So far, our main approach to classification problems is to first feed the input to a `Linear` layer which applies many different linear seperators to the input, and then apply the ReLU function in order to transform it into a new hidden representation.

One major problem with the above approach is that it is based on the `absolute` position of the original input features, which prevents us from learning generalizable features of the input. Instead, we could have a sliding window (i.e. convolution) that uses the same set of learned parameters on different local regions of the input, as shown below:



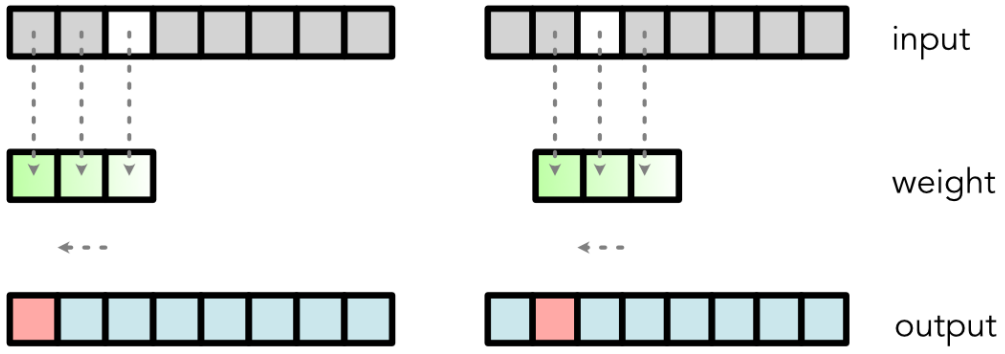
Instead of directly transforming the entire input image, the same convolution is applied at each part of the image to produce a new representation. It is `sliding` in the sense that, conceptually, the window over what region to process slides across the entire image to produce the output.

We will primarily use the convolution on images to understand how to learn these locally applied parameters, but to get an intuitivie sense of how convolution works, we will begin with input sequences in 1D.

Note: Convolutions are extensively covered by many excellent tutorials. In addition to this Guide, we recommend you to check other tutorials on convolution for more details.

1D Convolution

Our simple 1D convolution takes an input vector of length T and a weight (or kernel) vector of length K to produce an output vector of length T. It computes the output by sliding the weight along the input, zipping the weight with part of the input, reducing the zipped result to one value, and then saving this value in the output:



If the sliding window goes over the edge of the input, to make things simpler, we just assume the out-of-edge input values are 0.

An alternative way to think about a convolution is `unrolling` the input. Now imagine we have a function named `unroll` which could take an input tensor and produce a new output tensor:

```
def unroll(input, T, K):
    out = [[input[i + k] if i + k < T else 0 for k in range(K)]
for i in range(T)]
    return tensor(out)


input = tensor([1, 2, 3, 4, 5, 6])
K = 3
T = input.shape[0]
unrolled_input = unroll(input, T, K)
print(unrolled_input)
```

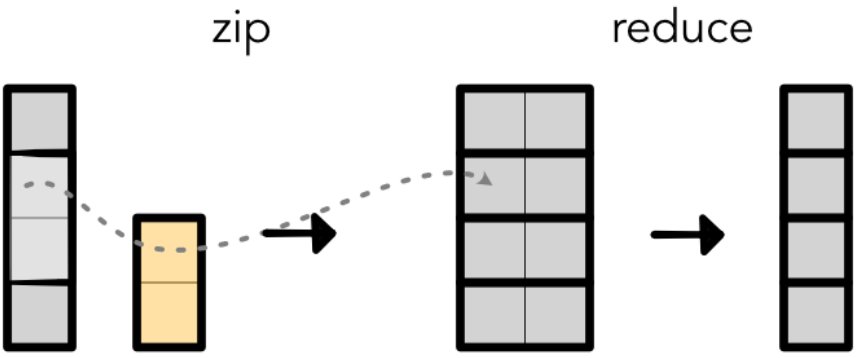
```
[
        [1.00 2.00 3.00]
        [2.00 3.00 4.00]
        [3.00 4.00 5.00]
        [4.00 5.00 6.00]
        [5.00 6.00 0.00]
        [6.00 0.00 0.00]]
```

We then apply matrix multiplication to take the dot-product: ::

```
weight = tensor([5, 2, 3])
output = (unrolled_input @ weight.view(K, 1)).view(T)
print(output)
```

```
[18.00 28.00 38.00 48.00 37.00 30.00]
```

We can treat convolution as a fusion of the following two separate operations (note that we do not implement it this way in practice!):



Given an input and weight, it efficiently unrolls the input and takes matrix multiplication with weight. This technique is very useful because it allows you to `hover` over a local segment of the input sentence. You can think of the weight as capturing a pattern (or many different patterns) in the original input.

Same as every other operation in the model, we need to be able to compute the `backward` operation of this 1D convolution. Note that we can reason through the flow of each cell with matrix multiplication. Going back to the previous example, the third cell in the original input (i.e. input[2]) is only utilized to compute three different cells in the ouput: output[0], output[1] and output[2].

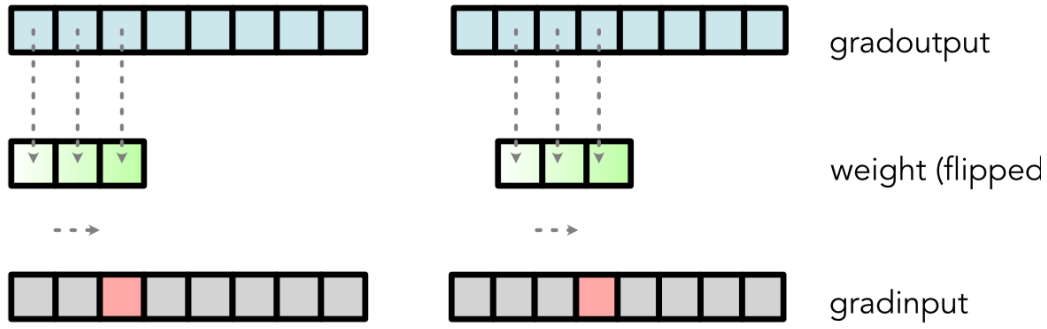
```
output[0] = weight[0] * input[0] + weight[1] * input[1] +
weight[2] * input[2]
output[1] = weight[0] * input[1] + weight[1] * input[2] +
```

```
weight[2] * input[3]
output[2] = weight[0] * input[2] + weight[1] * input[3] +
weight[2] * input[4]
```

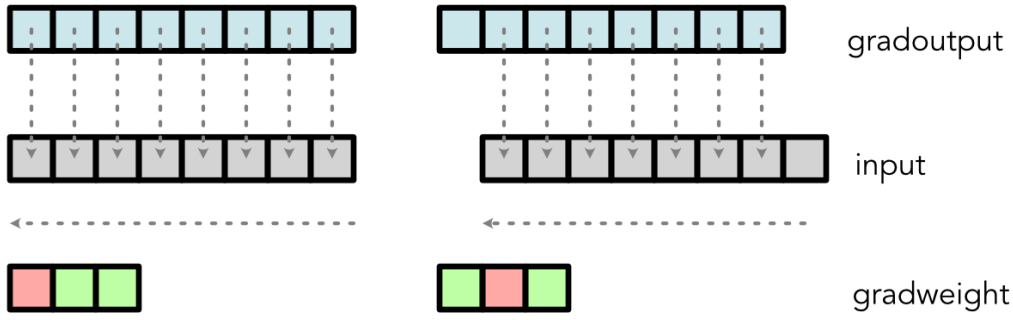
Therefore, the gradient calculation is,

```
class Conv:
   @staticmethod
   def backward(ctx, d):
       ...
       grad_input[2] = weight[0] * d[2] + weight[1] * d[1] +
weight[2] * d[0]
       ...
```

Visually, it implies that the `backward` of convolution is a convolution anchored in the opposite side with the reversed weights:



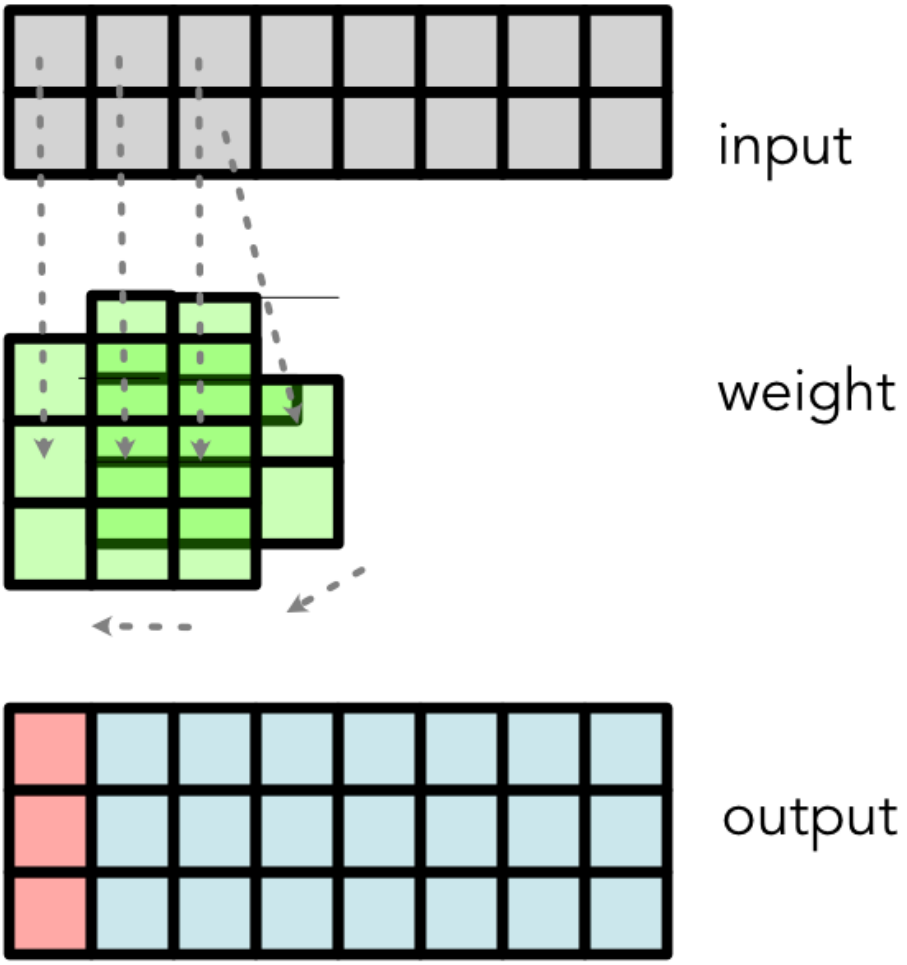
Similar gradient calculation can be derived for the weight:



The above implies that, same as matrix multiplication, implementing a fast convolution can be used for both `forward` and `backward`.

Finally, the above approach can be scaled up to handle multiple input features and multiple weights simultaneously through `channels`. Analogously to matrix

multiplication, we take an (in_channels, T) input and an (out_channels, in_channels, K) weight to produce an (out_channels, T). Below is an example with in_channels = 2, out_channels = 3, K = 3 and T = 8.



Codewise (might be a bit harder to read so we recommend just sticking with the above diagram):

```
def unroll_chan(input, T, C, K):
    out = [
        [input[i + k, c] if i + k < T else 0 for k in range(K)
for c in range(C)]
        for i in range(T)
    ]

    return tensor(out)


in_channels = 2
```

```
input = rand(T, in_channels)
unrolled_input = unroll_chan(input, T, in_channels, K)
print(unrolled_input.shape)  # Shape: T x (in_channels * K)
```

```
(6, 6)
```

We can now do this as a matrix multiplication

```
out_channels = 3
weight = rand(in_channels * K, out_channels)
output = unrolled_input @ weight
print(output.shape)
```

```
(6, 3)
```

1D convolution has all sorts of neat applications: it can be applied in NLP as a way of applying a model to multiple neighboring words; it can be used in speech recognition as a way of recognizing important sounds; it can be used in anomoly detection to find patterns that trigger an alertl; anywhere you can imagine that a mini-neural network applying to subset of a sequence can be useful.

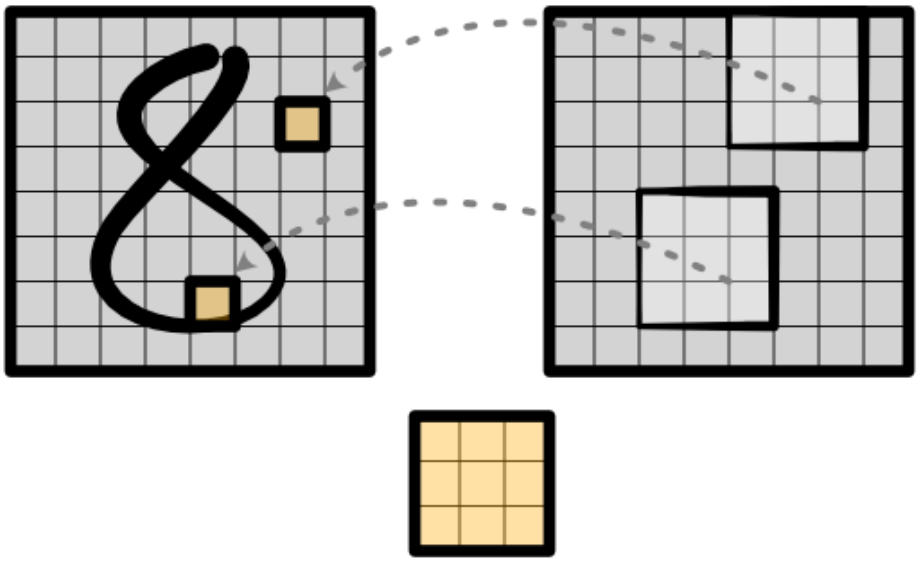
2D Convolution

While 1D convolutions detect patterns along a sequence, 2D convolutions detect patterns within a grid. The underlying math for the simple 2D is very similar to the 1D case. Our simple 2D convolution takes in an (H, W) input (i.e. height and width) and a (KH, KW) weight to produce an (H, W) output. The operation is nearly identical: we walk through each possible unrolled rectangle in the input matrix, and multiply with the weight. Assuming we had an analogous unroll function for matrices, this would be equivalent to computing:

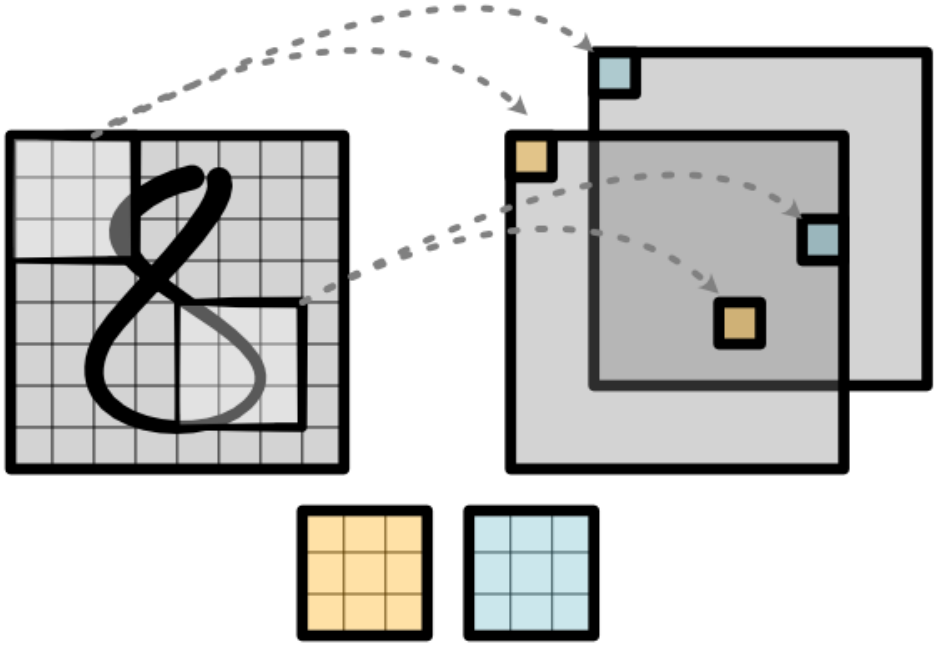
```
output = (unrolled_input.view(H, W, KH * KW) @ weight.view(KH * KW, 1)).view(H, W)
```

Another way to think about it is just applying weight as a `Linear` layer to each one of the rectangles in the input image.

Critically, just as the 1D convolution is anchored at the left, the 2D convolution is anchored at the top left. To compute its `backward`, we compute a bottom-right reverse convolution:



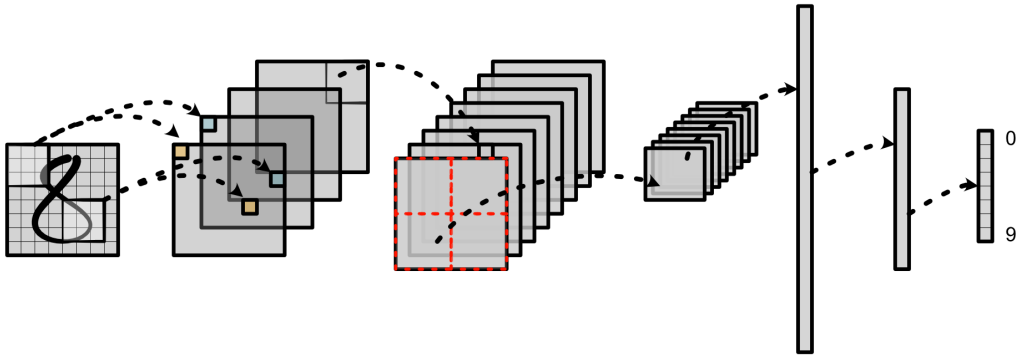
Finally, we can again complicate things by applying many weights to many input features simultaneously, which gives us the standard 2D convolution used in Torch. We take an (in_channels, H, W) input, and an (out_channels, in_channels, KW, KH) weight matrix to produce an (out_channels, H, W) output:



Very roughly, the output takes this form:

```
output = unrolled_input.view(H, W, in_channels * KH * KW) \
    @ weight.view(in_channels * KH * KW, out_channels)
```

2D convolution is the main operator for image recognition systems. It allows us to process images into local feature representations. It is also the key step in the convolutional neural network pipeline. It transforms the input image into hidden features which get propagated through each stage of the network:

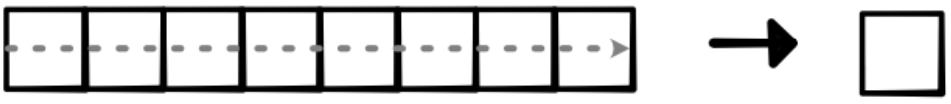


Pooling

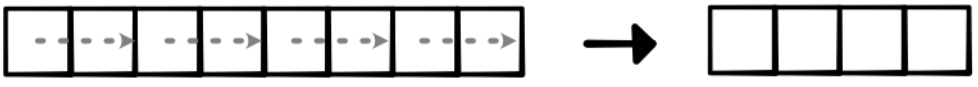
In previous modules, we have found it useful to reduce over certain dimensions to reduce the shape of tensors. For example, in the NLP example, we sum over the length of the sentence in order to classify based on the summed word representations. Critically, this operation does not remove the importance of words (i.e. they still receive gradient information), but it does allow us to make a single classification decision based on a fixed-sized representation.

This style of network goes by the informal name `pooling` in the neural network literature. When we reduce part of the input in order to work with a smaller size, we say we have pooled together the input representations.

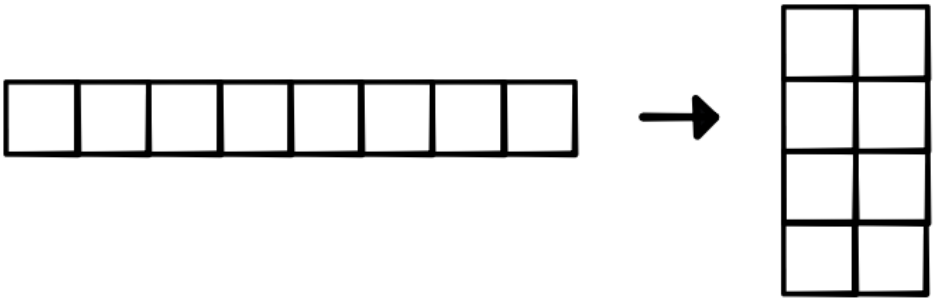
Applying reduction over length or other dimensions is one form of pooling. For sequential cases, we might call it X-over-time pooling (where X might be a sum, mean, max etc.):



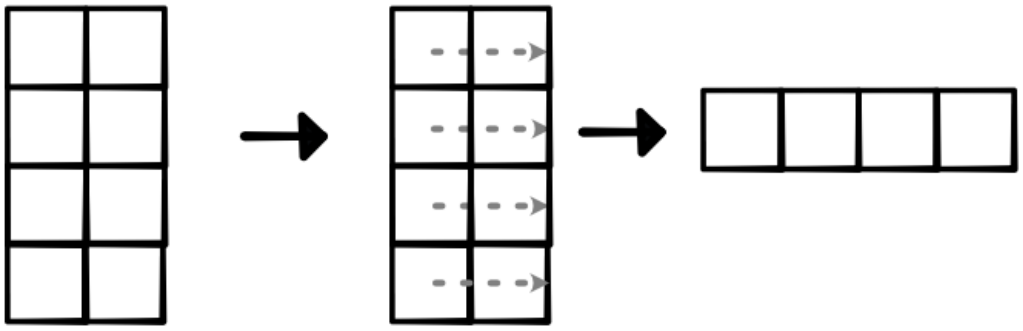
Another common form of pooling is to only pool locally within a dimension. For instance, we might pool together neighboring elements to reduce thel length for that dimension, as visualized below. This is common in domains like speech recognition where the input sequences are very long.



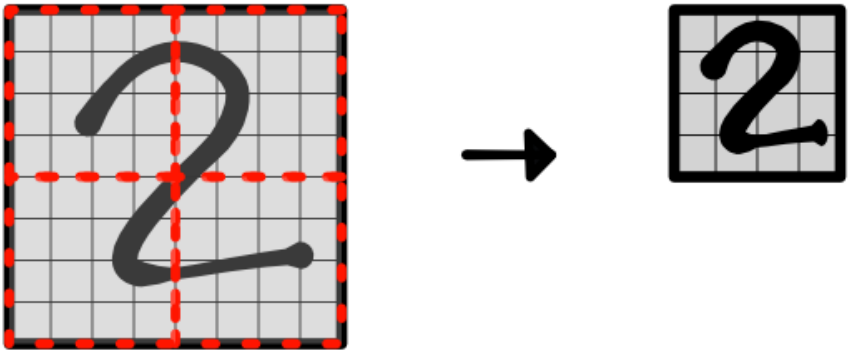
In practice, to avoid implementing the pooling operation as a loop, we can manipulate the shape and strides of the input tensor in order to pool it directly. Assuming the input tensor is contiguous, we can simply add an extra dimension by applying `view`. For instance, we can `view` a (8,) input as (4, 2):



Once we have the input in this form, we can reduce over the second dimension to get a (4,1) tensor, which can be viewed as a (4,). As long as the dimensions for pooling are divisible by the pooling constant (i.e. the input size for each individual pooling operation), this procedure will produce the correct pooled result, as visualized below. If not divisible, we can pad our input tensors, or add padding along the way.



You will implement a version of this type of pooling in two dimensions for images. You need to generalize the above idea in 1D pooling to create a shape with two extra dimesions to reduce over:

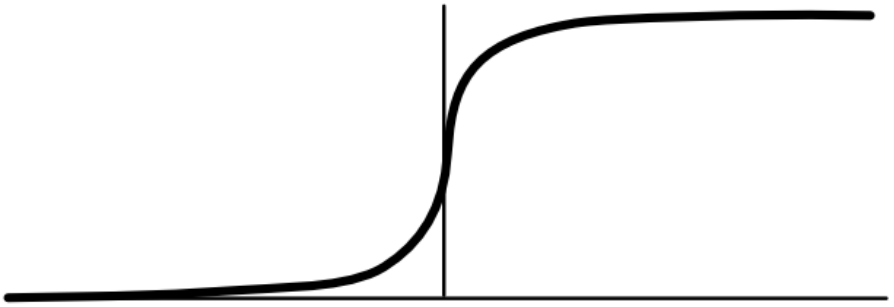


The benefit of pooling is that applying a small convolution over the pooled results covers a larger region in the original input image. These later layers ideally can learn higher-level properties of the input image.

Multiclass Classification

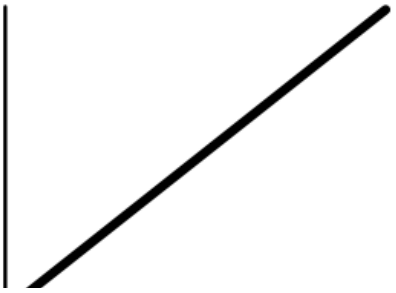
Sigmoid

So far, the key function that we have relied on for calculating loss is the sigmoid function. As visualized below, it goes to zero for large negative inputs, and goes to 1 for large positive inputs. In between, it forms a smooth S-curve.



As we saw in Module 1, sigmoid function makes it easier to apply auto-differentiation when training our models. It can be thought of as a smooth version of the step function $x > 0$, which signals whether x is greater than zero by returning a binary value. Another way to write this step function is $step(x) = argmax\{0, x\}$, i.e. returns which argument is bigger, 0 or 1. Whereas step function returns a binary choice, sigmoid function gives a "softer" differentiable choice.

We can connect sigmoid function to another function that we have used in previous MiniTorch Modules: the ReLU function that we use for activations. Recall that this function is defined as $ReLU(x) = max\{0, x\}$.



This has the following simple derivative function:

$$\text{ReLU'}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{ow} \end{cases}$$

Multiclass

The sigmoid function works great for binary classification problems. However, for many problems, we may want to do `multiclass` classification, where we have K possible output classes to select from. For these problems, we can assume that the model should output a K-dimensional vector which gives a score for each of the K possible classes:

Naturally, we pick the output class that has the highest score. Given a vector, argmax function returns a one-hot vector with 1 at the position of the highest-scored element and 0 for all other elements:

While argmax function seems a bit different at the first glance, we can view it as a generalization of the $x > 0$ function: each position is either 0 or 1. We can also see that its derivative will be zero almost everywhere: a small perturbation to the input will not change the output value.

In order to fix this issue, we need a soft version of the argmax function, just like sigmoid function smooths over the input changes. The generalization of sigmoid function is appropriately known as the `softmax` function, which is computed as:

$$\text{softmax}(\mathbf{x}) = \frac{\exp \mathbf{x}}{\sum_i \exp x_i}$$

Like the sigmoid function, every value of softmax function is between 0 and 1, and a small change to any of the input scores will result in a change to all of the output values.

As the softmax function requires exponentiating the input scores, it can be numerically unstable in practice. Therefore it is common to use a numerical trick to compute the log of the softmax function instead:

$$\text{logsoftmax}(\mathbf{x}) = \mathbf{x} - \log \sum_i \exp x_i = \mathbf{x} - \log(\sum_i \exp(x_i - m)) - m$$

where m is the max element of $\mathbf{x}$. This trick is common enough that there is a nice derivation on wikipedia. (This is a pratical trick for sigmoid function as well, which we ignored in earlier modules.)

Speaking of max, we can add a max operator to our code base. We can compute the max of a vector (or tensor in general) as a reduction, which returns the single highest-scored element in the input. Intuitively, we can think about how small changes to the input impact this returned value. Ignoring ties, only the element that has the highest score will have a non-zero derivative, and its derivative will be 1. Thereforce the gradient of the max reduction is a one-hot vector with 1 for the highest-scored element, i.e. the argmax function.