



# Module 0.2 - Models and Modules



# Module 0.2

## Models and Modules



# Class Note

- You need to link your GitHub account
- Still several students with unlinked accounts



# Review





# Function Type

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

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

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



# Functions as Arguments

```
from typing import Callable

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



# Functional Python

## Functions as Returns

```
def combine3(  
    fn: Callable[[float, float], float]  
) -> Callable[[float, float, float], float]:  
    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))
```



# Quiz





# Outline

- Modules
- Visualization
- Datasets



# Modules



# Model

- Models: parameterized functions.
  - $m(x; \theta)$
  - $x$  - input
  - $m$  - model
- Initial Focus:
  - $\theta$  - parameters



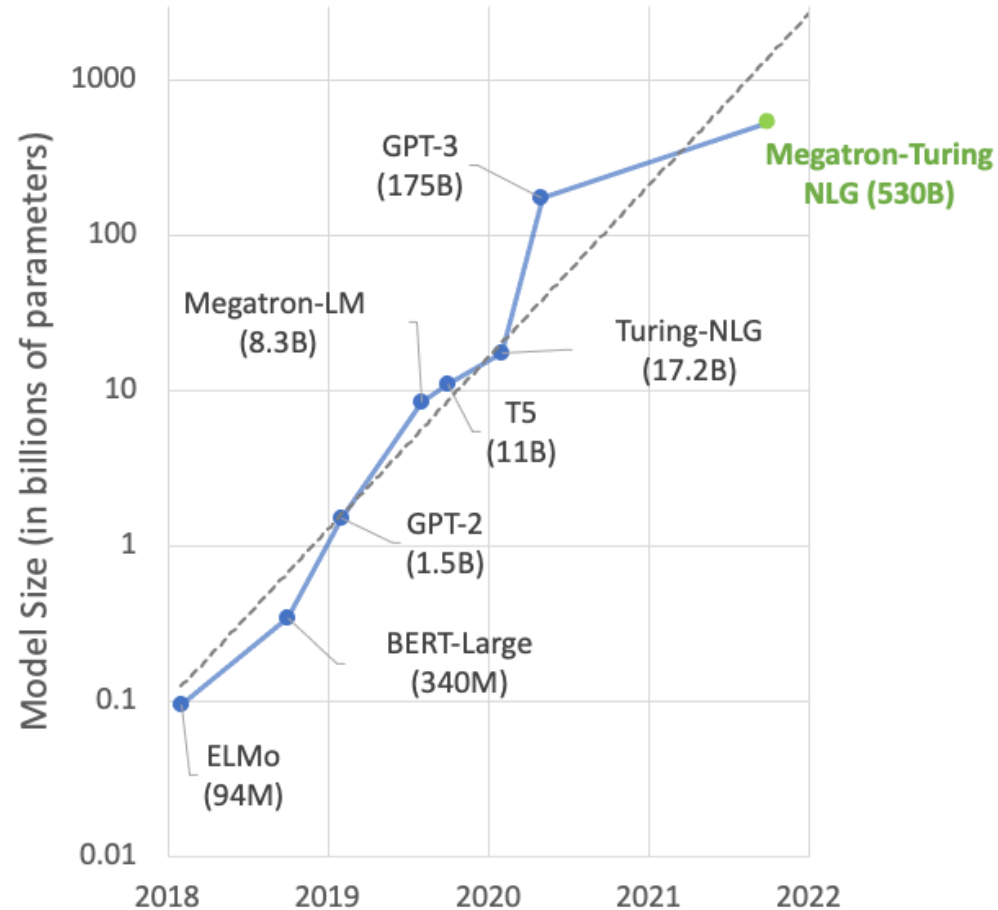
# Parameters

- Anything learned is in the parameters.
- Modern parameters sets are both:
  - Large
  - Complex





# Growth in Parameter Size





# Complexity

Inception - Table of precise sizes



# Specifying Parameters

- Datastructures to specify parameters
- Requirements
  - Independent of implementation
  - Compositional



# Module Trees

- Each Module owns a set of parameters
- Each Module owns a set of submodules





# Module Trees

## Benefits

- Can extract all parameters without knowing about Modules
- Can use mix and match Modules for different tasks

## Downsides

- Verbose, repeats some functionality of declaration and use.



# Module Storage

Stores three things:

- Parameters
- Submodules
- Generic Python attributes



# Module Example

```
from minitorch import Module, Parameter

class OtherModule(Module):
    def __init__(self):
        # Must initialize the super class!
        super().__init__()
        self.uncool_parameter = Parameter(60)

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

        # Type 1, a parameter.
        self.parameter1 = Parameter(15)
        self.cool_parameter = Parameter(50)

        # Type 2, user data
        self.data = 25

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



# Parameters

- Everything that is learned in the model
- Controlled and changed outside the class





# Submodules

- Other modules that are called
- Store their own parameters and submodules
- Together forms a tree



# Everything Else

- Modules act mostly like standard python objects
- You can have additional information stored



# Module Example

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

```
[('parameter1', 15),  
 ('cool_parameter', 50),  
 ('sub_module_a.uncool_parameter', 60),  
 ('sub_module_b.uncool_parameter', 60)]
```



# Extended Example

```
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)
```





# Extended Example

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

```
Module1().named_parameters()
```

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



# How does this work?

- Internally `Module` spies to find `Parameter` and `Module` objects
- A list is stored internally.
- Implemented through Python magic methods



# Detail: Magic Methods

- Any method that starts and ends with `__`
- Used to override default behavior of the language.
- We will use for many things, including operator overloading



# Interception Code

## Module construction

```
class MyModule(Module):  
    def __setattr__(self, key, val):  
        if isinstance(val, Parameter):  
            self.__dict__["_parameters"][key] = val  
        elif isinstance(val, Module):  
            self.__dict__["_modules"][key] = val  
        else:  
            super().__setattr__(key, val)
```



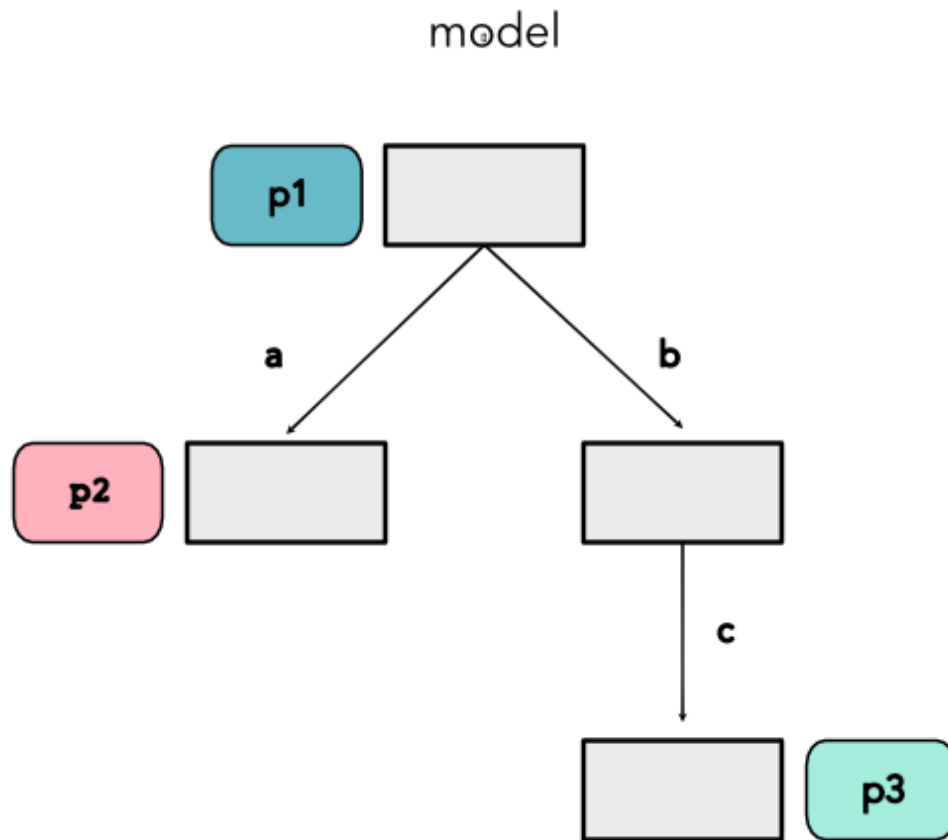


# Parameter Naming

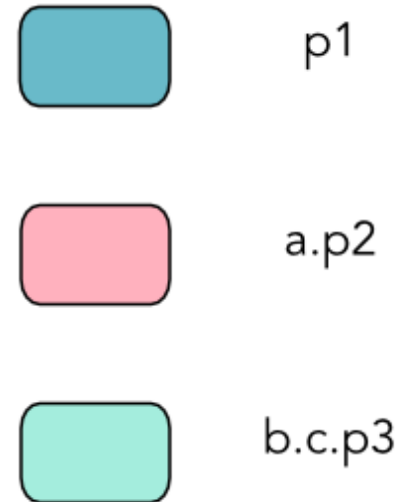
- Every parameter in a model has a unique name.
- Naming is determined by walking the tree.
- Names are prefixed by the path from the root.



# Module Naming



named\_parameters





# Other Module Metadata

- Other information can be communicated through the tree.
- Common example: Is the model in train or test mode?



# Homework Note

- Must be recursive implementation
- Have to walk the full tree
- (Companies love this as an interview question!)





# Real World Examples

## Code from GPT-2

```
from torch import nn

class Block(nn.Module):
    def __init__(self, n_ctx, config, scale=False):
        super().__init__()
        hidden_size = config.n_embd
        inner_dim = config.n_inner if config.n_inner is not None else 4 * hidden_size
        self.ln_1 = nn.LayerNorm(hidden_size, eps=config.layer_norm_epsilon)
        self.attn = Attention(hidden_size, n_ctx, config, scale)
        self.ln_2 = nn.LayerNorm(hidden_size, eps=config.layer_norm_epsilon)
        if config.add_cross_attention:
            self.crossattention = Attention(
                hidden_size, n_ctx, config, scale, is_cross_attention=True
            )
        self.ln_cross_attn = nn.LayerNorm(
            hidden_size, eps=config.layer_norm_epsilon
        )
        self.mlp = MLP(inner_dim, config)
```



# Real World Examples

## Block from Inception

```
class Inception3(nn.Module):
    def __init__(
        self,
        num_classes=1000,
        aux_logits=True,
        transform_input=False,
        inception_blocks=None,
        init_weights=None,
    ):
        super(Inception3, self).__init__()
        ...
        self.aux_logits = aux_logits
        self.transform_input = transform_input
        self.Conv2d_1a_3x3 = conv_block(3, 32, kernel_size=3, stride=2)
        self.Conv2d_2a_3x3 = conv_block(32, 32, kernel_size=3)
        self.Conv2d_2b_3x3 = conv_block(32, 64, kernel_size=3, padding=1)
        self.maxpool1 = nn.MaxPool2d(kernel_size=3, stride=2)
        self.Conv2d_3b_1x1 = conv_block(64, 80, kernel_size=1)
        self.Conv2d_4a_3x3 = conv_block(80, 192, kernel_size=3)
        self.maxpool2 = nn.MaxPool2d(kernel_size=3, stride=2)
```



# Visualization



# Main Idea

- Show properties of your model as you code
- See real time graphs as you train models
- Make convincing figures of your full system





# Library: Streamlit

## Easy to use Python GUI

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



# Code Snippet

## Streamlit windows

```
import streamlit as st
st.write("## Sandbox for Model Training")
...
st.plotly_chart(fig)
```



# Gotchas

- Changes to the visualization code will autoupdate
- Changes to the library will not autoupdate



# Other Options

Many other ML tailored options

- Tensorboard
- Hosted services: Weights and Biases, Comet





# Datasets



# Sneak Preview

- Task 0.5: Intro to our first ML problem
- Basic separation of points on a graph
- Manual classifier



# Datasets

- Simple
- Diag
- Split
- Xor



# Parameter Knobs

- $W_1$
- $W_2$
- Bias





# Sneak Preview

Playground



# Q&A

