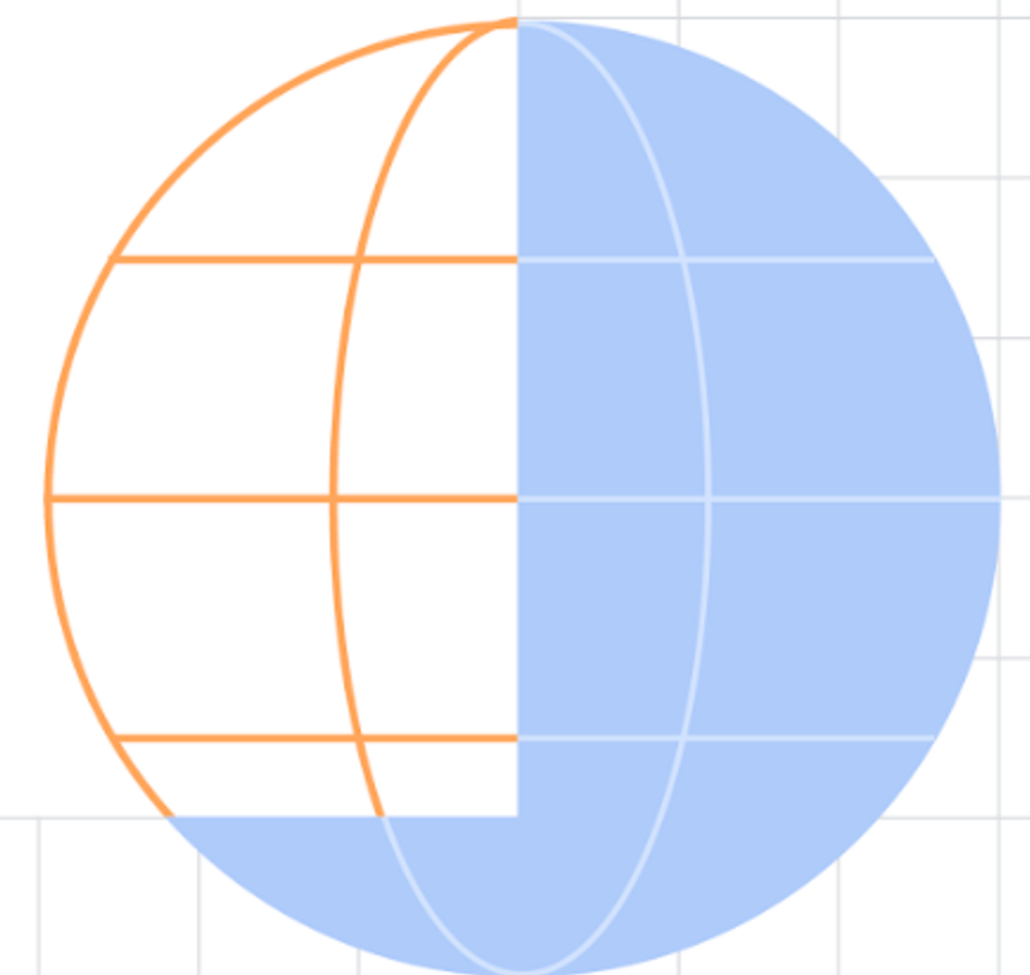


# Introduction to JAX with Flax



Phillip Lippe  
GDE Amsterdam  
[@phillip\\_lippe](https://twitter.com/phillip_lippe)  
[phlippe.github.io](https://phlippe.github.io)

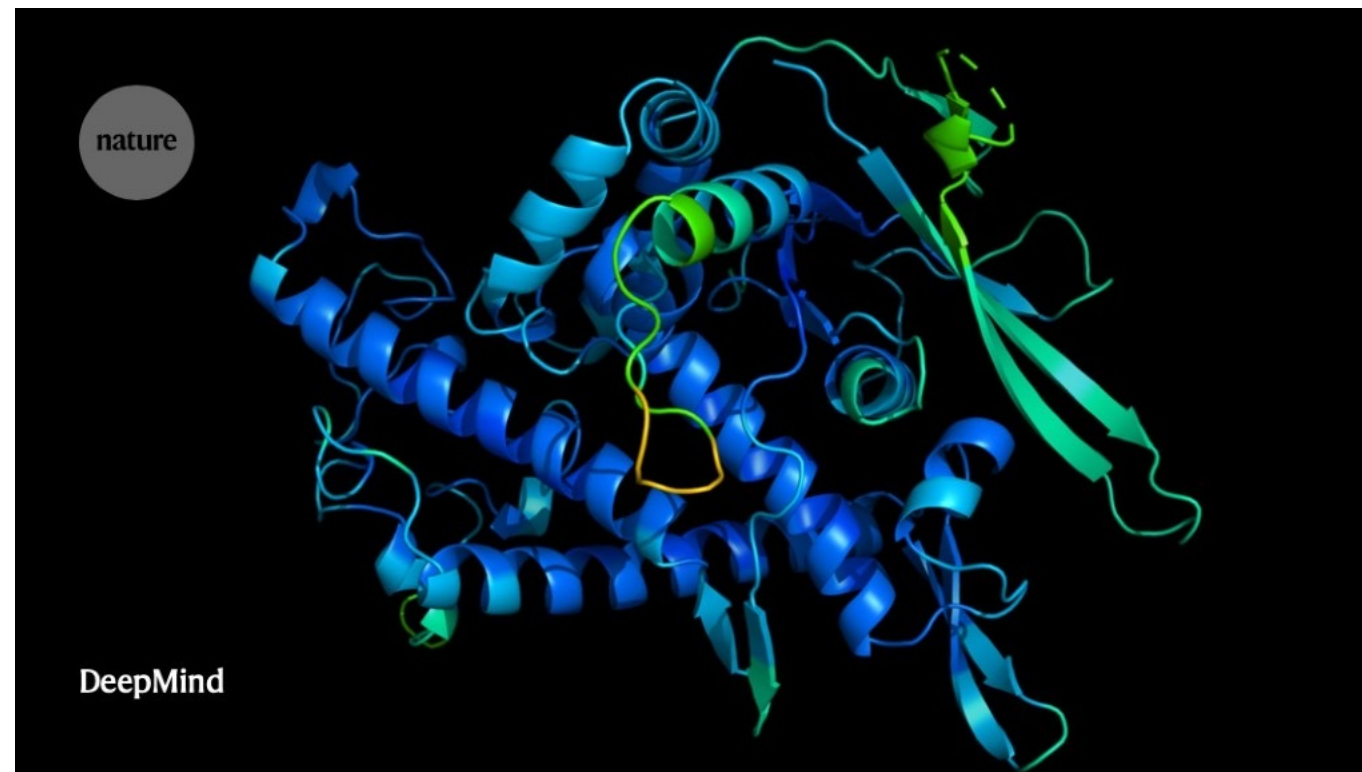


# Goals of this talk

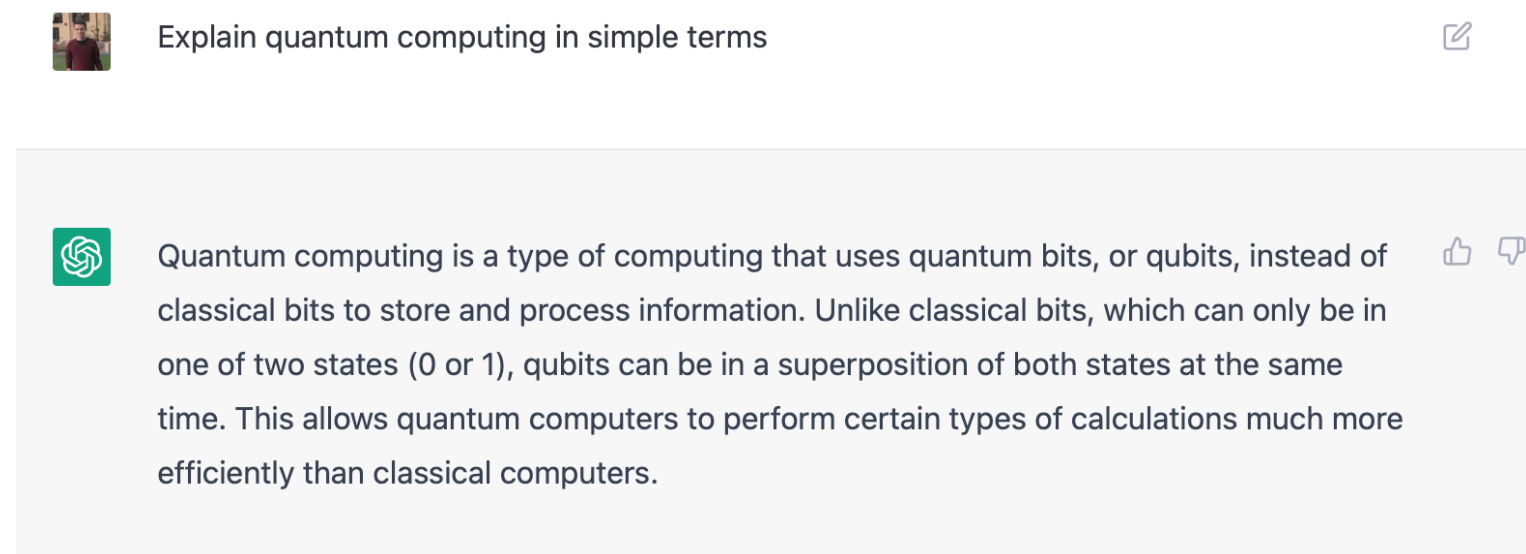
- 1) What features are we looking for in an ML/DL framework?
- 2) What is JAX?
- 3) What sets JAX apart from other frameworks?
- 4) How can train Neural Networks in JAX with Flax?
- 5) Where can I continue my learning journey into JAX with Flax?



# Successes of Machine and Deep Learning



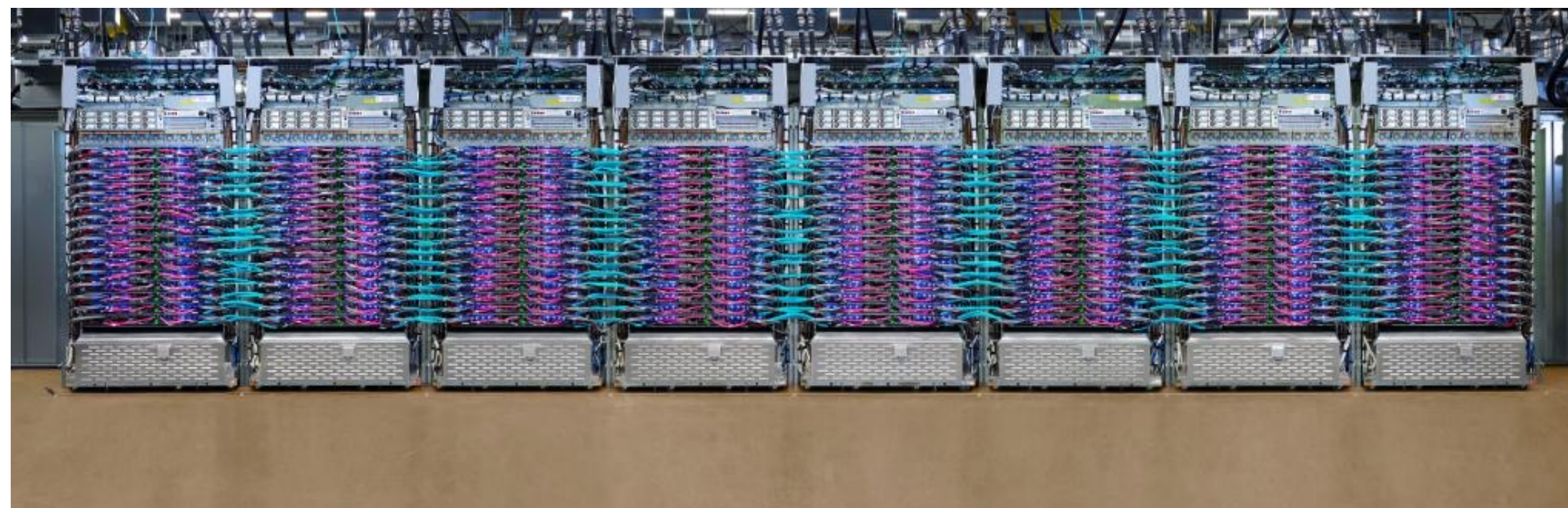
AlphaFold



ChatGPT



Imagen



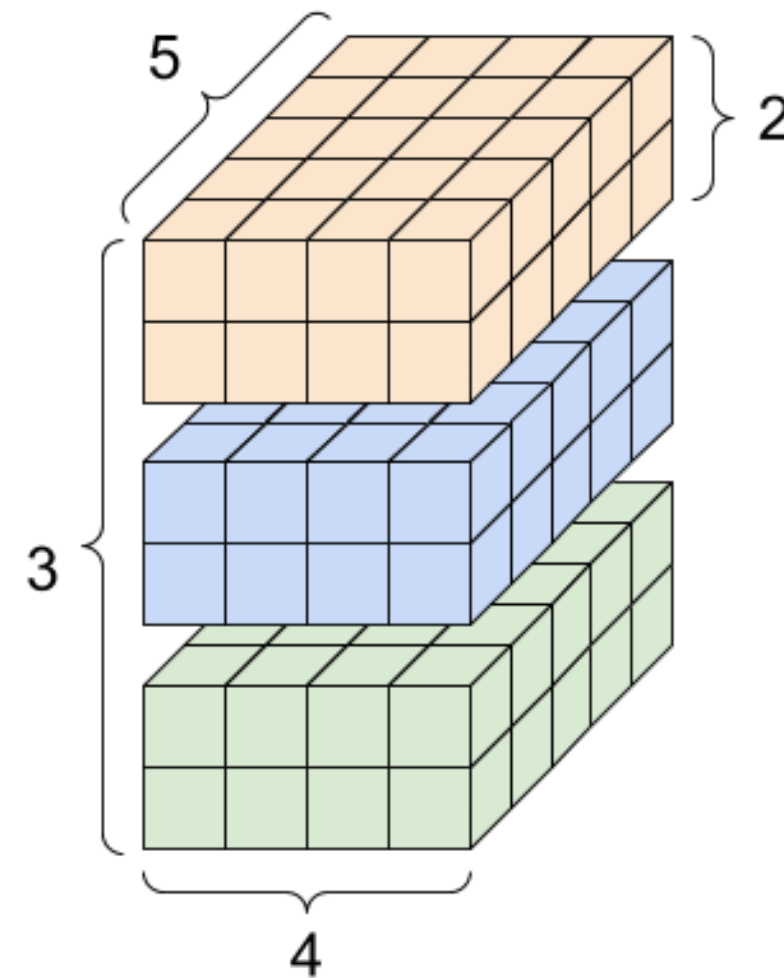
TPU Data Center

Secret sauce:  
Enormous compute power

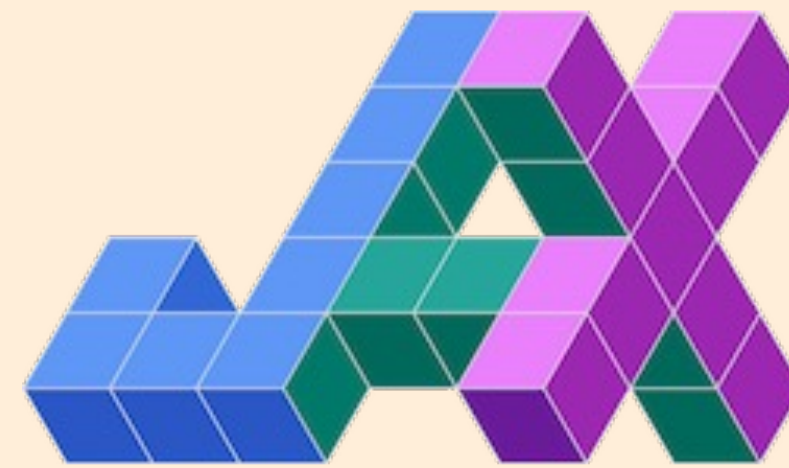


# Basic Requirements on a Deep Learning Framework

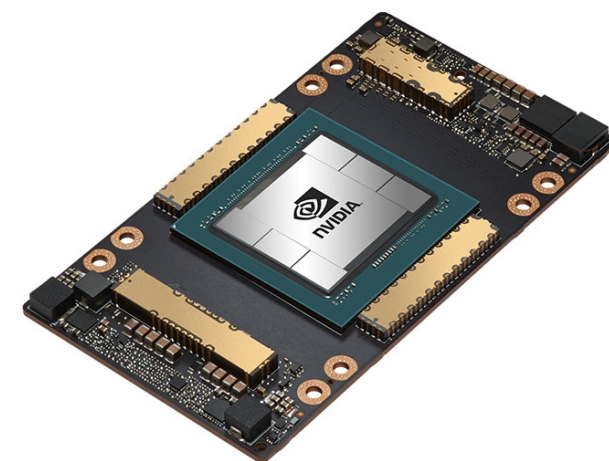
Matrix operations



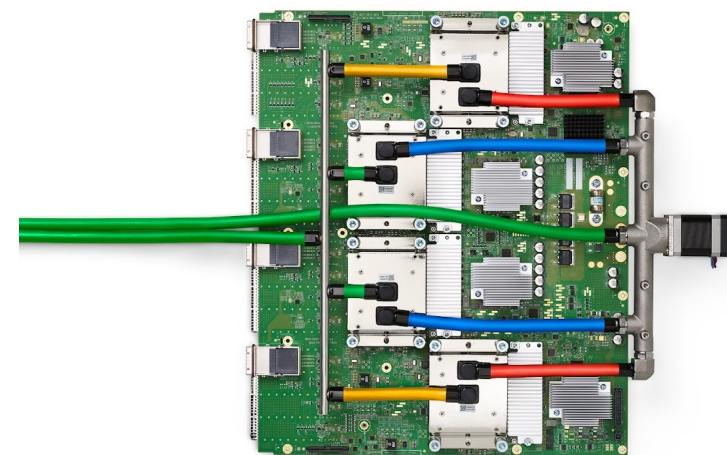
**NumPy** 



Accelerator support

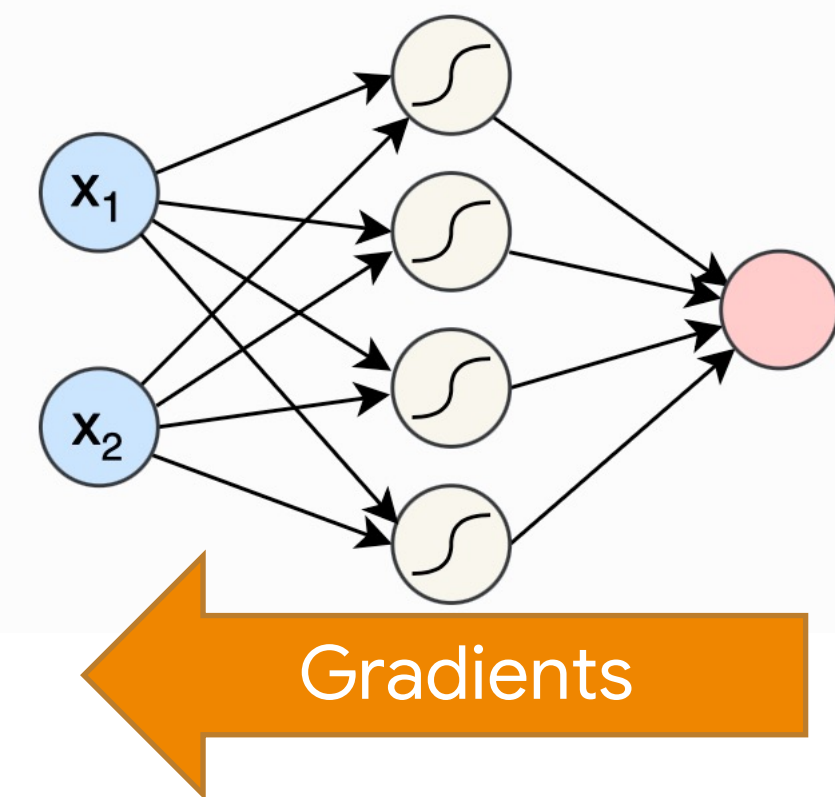


GPUs ([NVIDIA A100](#))



[TPU v4](#)

Automatic  
Differentiation





# JAX: Accelerated NumPy with Autograd

- **Matrix Operations:** JAX shares the same basic API with NumPy

```
import jax
import jax.numpy as jnp
print("Using jax", jax.__version__)
```

Using jax 0.3.25

```
a = jnp.zeros((2, 5), dtype=jnp.float32)
print(a)
```

```
[[0.  0.  0.  0.  0.]
 [0.  0.  0.  0.  0.]]
```

# JAX: Accelerated NumPy with Autograd

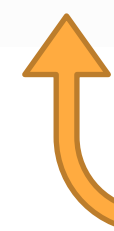
- **Accelerator Support:** JAX supports operations on various backends, e.g. GPUs

```
b = jnp.arange(6)  
print(b)
```

```
[0 1 2 3 4 5]
```

```
b.__class__
```

```
jaxlib.xla_extension.DeviceArray
```



Tensor by default on accelerator (here GPU)  
JAX arrays on CPU are NumPy arrays



# JAX: Accelerated NumPy with Autograd

- **Automatic Differentiation:** JAX allows to *transform* functions, such as taking the gradient of a function via `jax.grad`

```
def mse_loss(preds, labels):  
    return ((preds - labels) ** 2).mean()  
  
print('Loss', mse_loss(jnp.array([1.0, 2.0]),  
                        jnp.array([0.0, 1.5])))
```

Loss 0.625

```
mse_grad_fn = jax.grad(mse_loss)  
print('Loss gradient', mse_grad_fn(jnp.array([1.0, 2.0]),  
                                   jnp.array([0.0, 1.5])))
```

Loss gradient [1. 0.5]

Analytical gradient:  
 $(\text{preds} - \text{labels})$

# JAX: Function Transformations

- JAX is based on **transformations of pure functions**
  - Pure functions = functions without any side effects, limited to input and outputs
- For this, JAX lifts functions into an intermediate language called `jaxpr`

```
exmp_preds = jnp.array([1.0, 2.0])  
exmp_labels = jnp.array([0.0, 1.5])
```

```
jax.make_jaxpr(mse_loss)(exmp_preds, exmp_labels)
```

```
{ lambda ; a:f32[2] b:f32[2]. let  
  c:f32[2] = sub a b  
  d:f32[2] = integer_pow[y=2] c  
  e:f32[] = reduce_sum[axes=(0,)] d  
  f:f32[] = div e 2.0  
in (f,) }
```

Inputs with evaluated shapes

Basic function operations

Output with evaluated shape



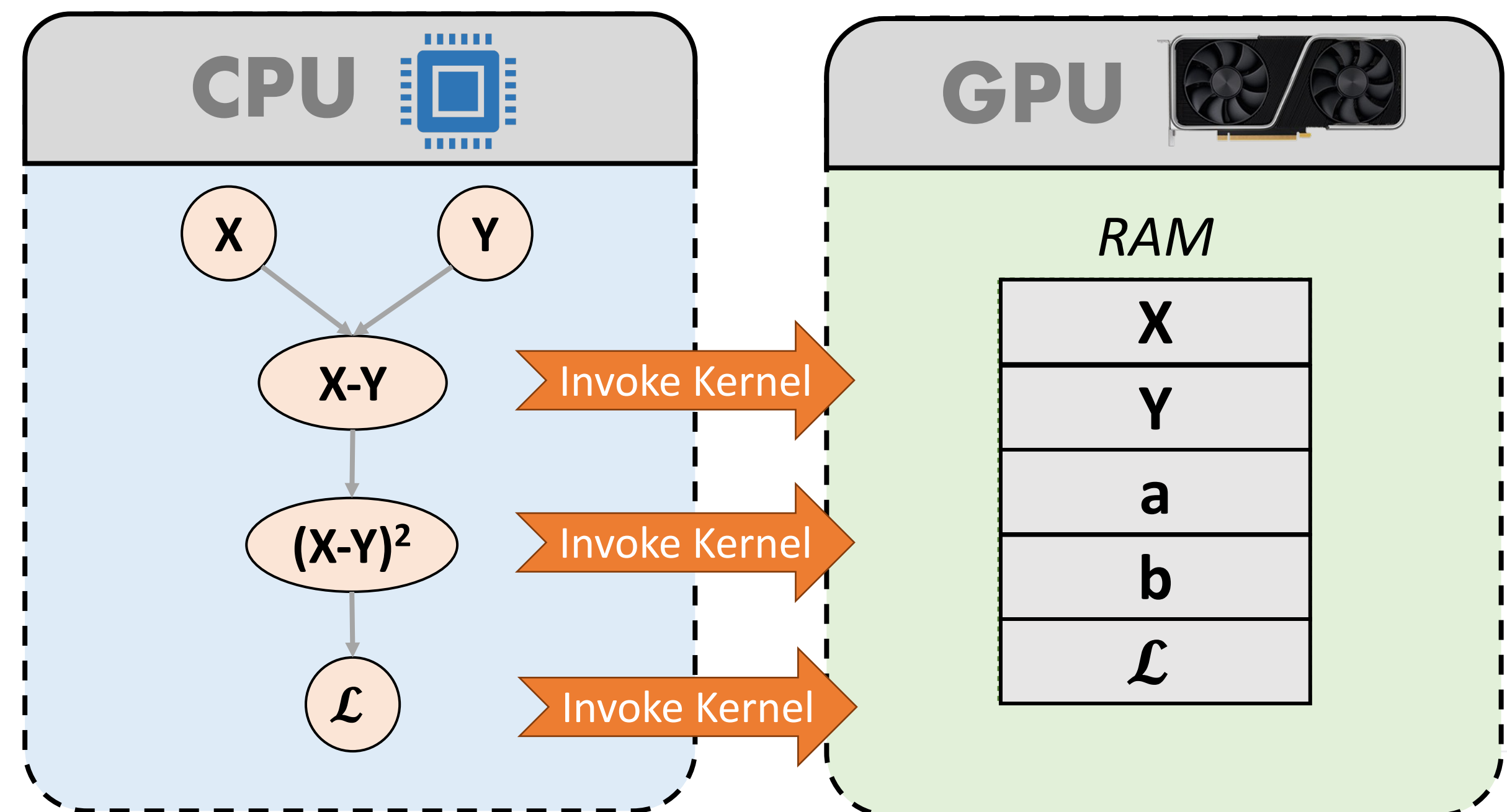
# JAX: Function Transformations

- JAX offers several function transformations, most importantly:
  - `jax.grad` – Gradient of a function
  - `jax.jit` – Just-In-Time Compilation
  - `jax.vmap` – Vectorize function
  - `jax.pmap` – Parallelize function on multiple devices

# JAX: Just-In-Time Compilation

- Naïve execution of operations can lead to considerable overhead in GPU  $\leftrightarrow$  CPU communication
- Can we do better?
- Yes, with Just-In-Time Compilation!

```
def mse_loss(x, y):  
    return ((x - y) ** 2).mean()
```





# JAX: Just-In-Time Compilation

- Just-In-Time (JIT) compilation allows for very efficient code with little effort
- Transforming a function via `jax.jit` uses XLA (Accelerated Linear Algebra) to compile multiple operations together to improve speed and memory usage

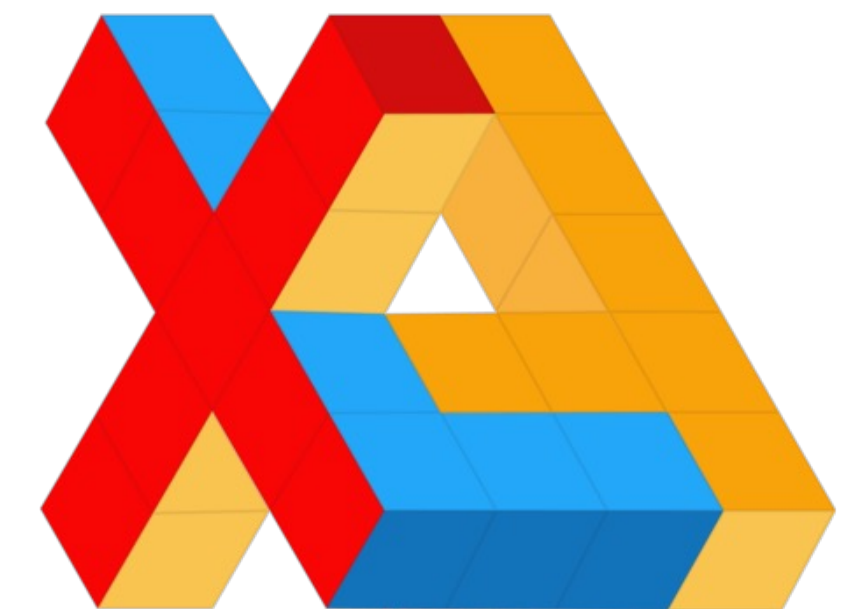
```
mse_jit = jax.jit(mse_loss)
```

```
%timeit mse_loss(x_batch, y_batch).block_until_ready()
```

```
434 µs ± 13.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
%timeit mse_jit(x_batch, y_batch).block_until_ready()
```

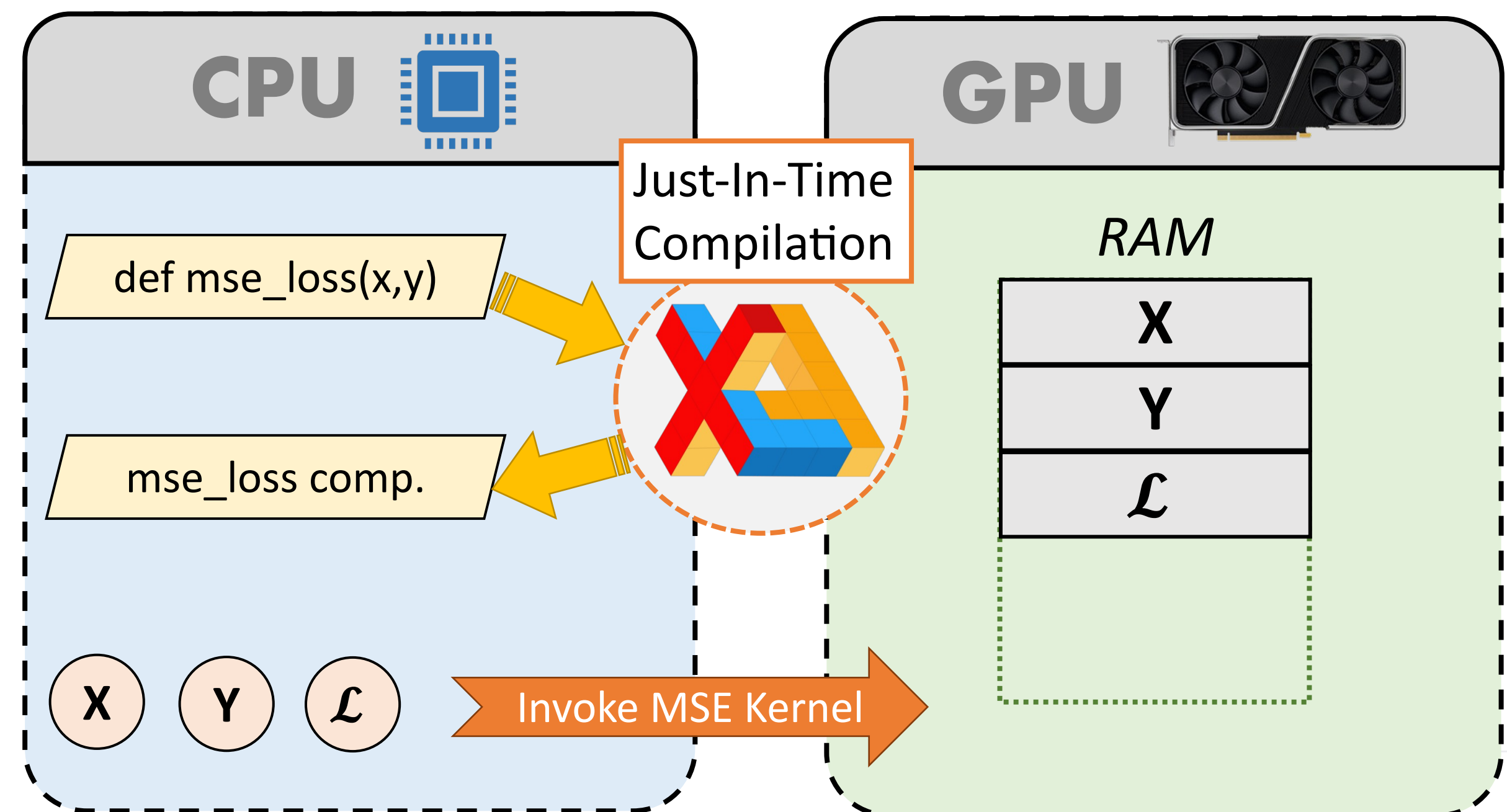
```
58.7 µs ± 1.42 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```



# JAX: Just-In-Time Compilation

- Improves **execution speed** (operation fusion, specializing for shapes)
- Improves **memory usage** (fewer intermediate variables)
- Improves **portability** (any backend supported in XLA)

```
@jax.jit
def mse_loss(x, y):
    return ((x - y) ** 2).mean()
```



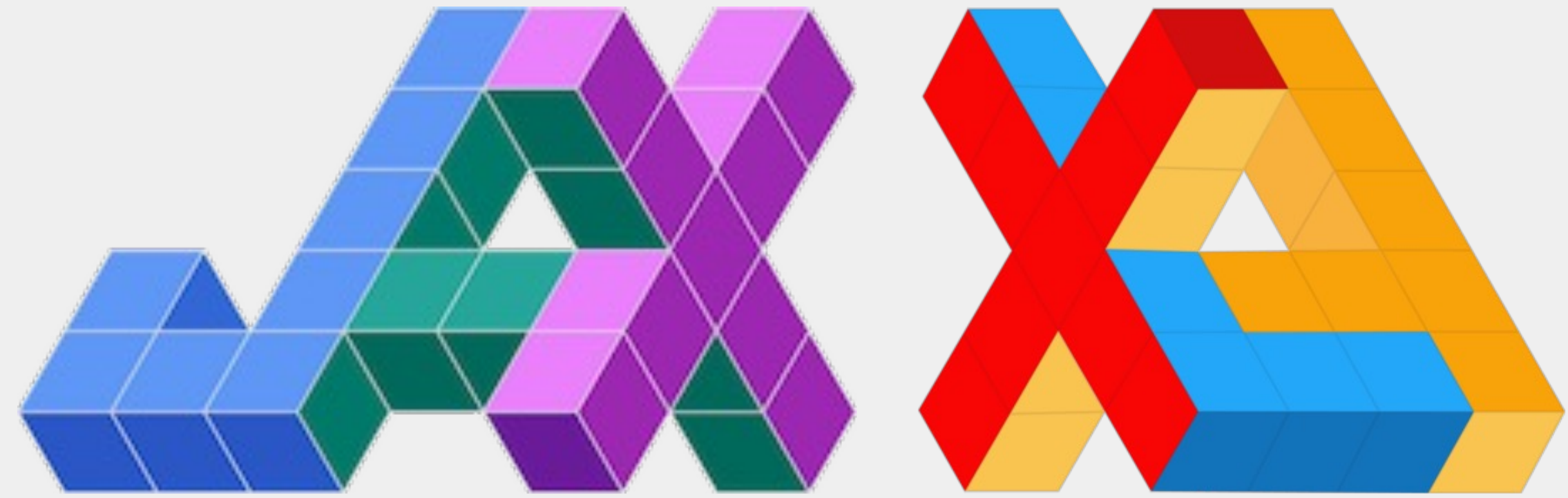


# JAX: The Sharp Bits

- JAX relies on pure functions, which requires function-centric programming
  - **Immutable Tensors:** in-place operations could have side-effects (JIT may still use in-place ops)
  - **Pseudo Random Numbers:** seed needs to be passed explicit to functions, no global seed variable
- JIT-Compilation is shape specific, need more care with dynamic shapes (e.g., graphs or natural language – use padding)
- Debugging in JIT-compiled functions more difficult since compiled functions can't throw an error → potentially undesired side-effects

# Summary

- JAX is Accelerated NumPy with Autograd
- Key feature: transformations of pure functions
- Just-In-Time compilation for taking full advantage of accelerators



```
exmp_preds = jnp.array([1.0, 2.0])  
exmp_labels = jnp.array([0.0, 1.5])
```

```
jax.make_jaxpr(mse_loss)(exmp_preds, exmp_labels)
```

```
{ lambda ; a:f32[2] b:f32[2]. let  
  c:f32[2] = sub a b  
  d:f32[2] = integer_pow[y=2] c  
  e:f32[] = reduce_sum[axes=(0,)] d  
  f:f32[] = div e 2.0  
in (f,) }
```

# Neural Networks with JAX

- How can we implement an NN in JAX?
- Several libraries available, such as:
  - [Flax](#) – Google Brain, focuses on flexibility and clarity
  - [Haiku](#) – DeepMind, focuses on simplicity and compositionality
  - [Trax](#) – Google Brain, solutions for common training tasks
  - [Equinox](#) – Patrick Kidger and Cristian Garcia, NNs as callable Pytrees



# Flax: Neural Networks with JAX



- Main aspects of a Neural Network library in JAX:

- How can we implement Neural Network layers as functions?

Linen API

- How do we handle parameters (weights, biases, etc.)?

Pytrees

- How do we optimize the model's parameters?

Optax

- How do we put everything together with JIT support?

TrainState

# Flax: Linen API



- Flax defines layers as **Modules**, acting as an *immutable dataclass*

```
from flax import linen as nn
```

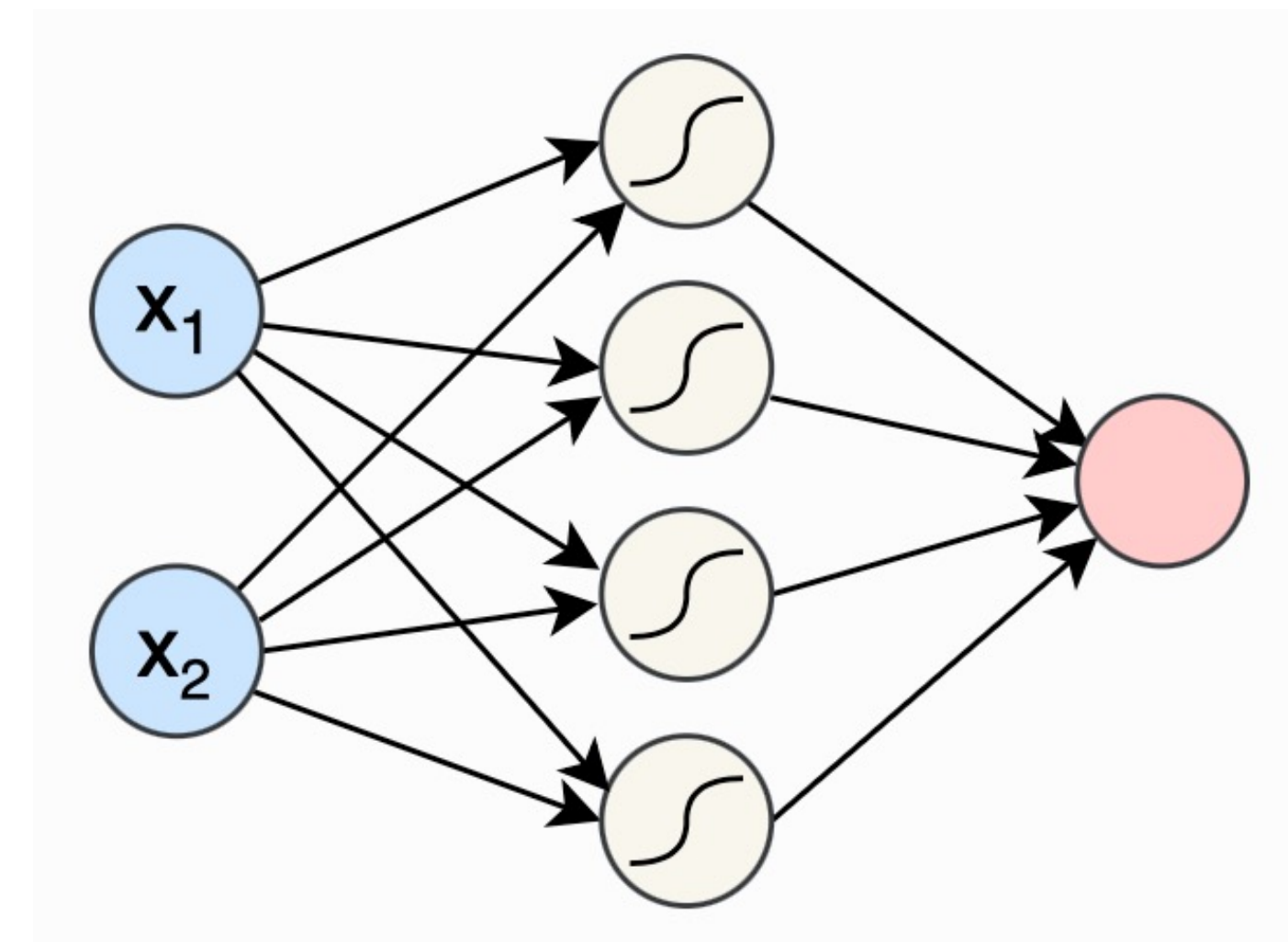
```
class MyModule(nn.Module):  
    # Some dataclass attributes, like hidden dimension  
    # varname : vartype  
  
    def setup(self):  
        # Flax uses "lazy" initialization.  
        # In here, define your submodules etc.  
        pass  
  
    def __call__(self, x):  
        # Function for forward pass  
        pass
```

# Flax: Linen API



- Flax defines layers as **Modules**, acting as an *immutable dataclass*

```
class SimpleClassifier(nn.Module):  
    num_hidden : int    # Number of hidden neurons  
    num_outputs : int   # Number of output neurons  
  
    def setup(self):  
        # Create the modules we need to build the network  
        # nn.Dense is a linear layer  
        self.linear1 = nn.Dense(features=self.num_hidden)  
        self.linear2 = nn.Dense(features=self.num_outputs)  
  
    def __call__(self, x):  
        # Forward pass  
        x = self.linear1(x)  
        x = nn.tanh(x)  
        x = self.linear2(x)  
        return x
```



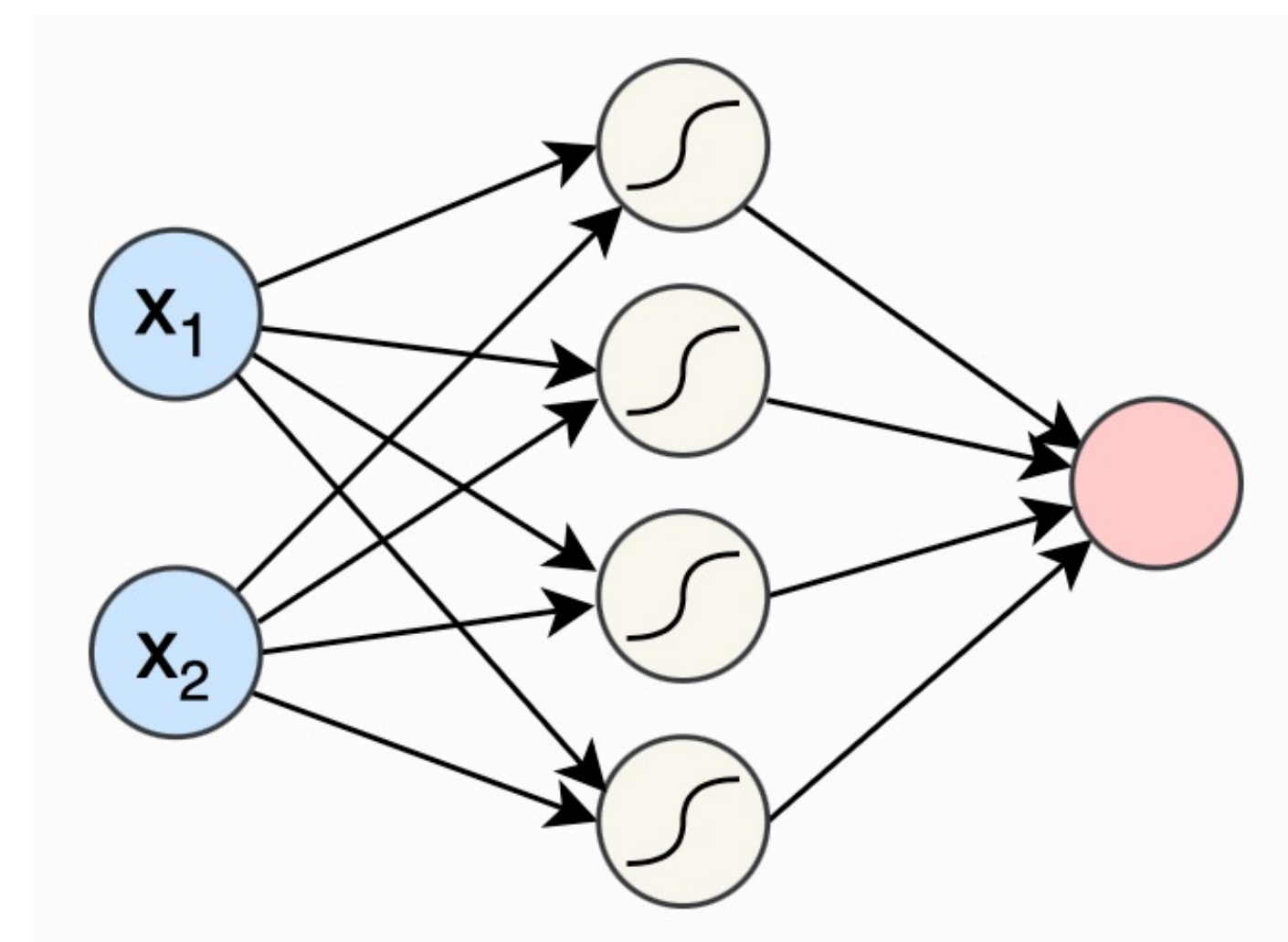


# Flax: Linen API



- We can combine sub-module definition and their call via `nn.compact`:

```
class SimpleClassifierCompact(nn.Module):  
    num_hidden : int    # Number of hidden neurons  
    num_outputs : int   # Number of output neurons  
  
    @nn.compact  
    def __call__(self, x):  
        # Forward pass while defining necessary layers  
        x = nn.Dense(features=self.num_hidden)(x)  
        x = nn.tanh(x)  
        x = nn.Dense(features=self.num_outputs)(x)  
        return x
```



# Flax: Linen API

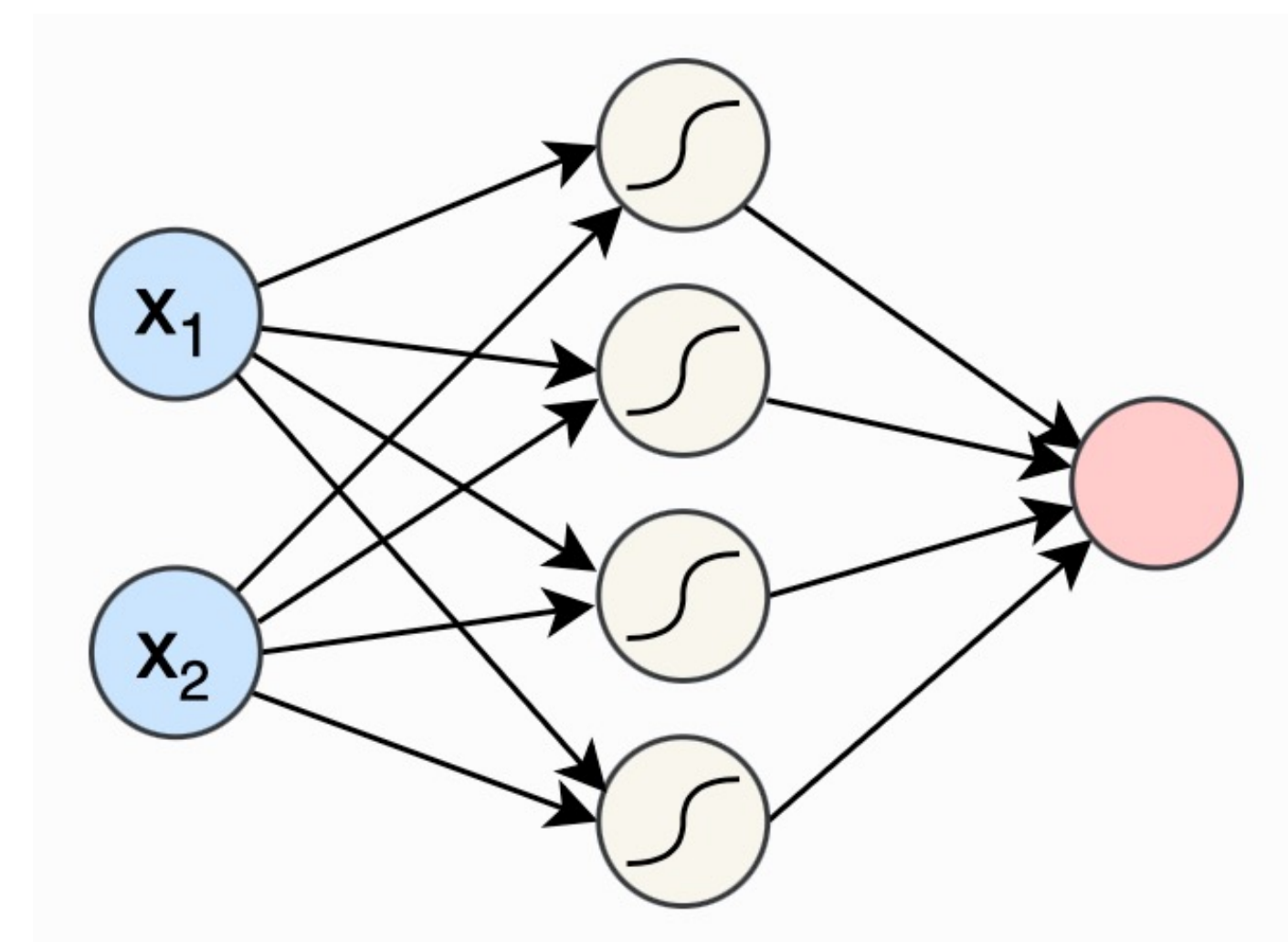


- Parameters cannot be part of the Module since they must be *mutable*
- Solution: parameters act as an additional input to the module
- Create parameters via `module.init` function:

```
example_input = jnp.zeros((16, 2))
```

```
model = SimpleClassifier(num_hidden=4, num_outputs=1)
```

```
params = model.init(jax.random.PRNGKey(42),  
                    example_input)
```



# Flax: Linen API



- Parameters are stored as **Immutable Pytrees**, a tree-structured container
  - In Flax, Pytrees are mostly nested dictionaries, with leafs being the parameter values
  - Pytrees allow functions to access collections like parameters, and potentially give one as output
- JAX defines several operations on Pytrees, example: print the shapes of the parameters

```
jax.tree_map(lambda p: p.shape, params)
```

```
FrozenDict({  
  params: {  
    linear1: {  
      bias: (4,),  
      kernel: (2, 4),  
    },  
    linear2: {  
      bias: (1,),  
      kernel: (4, 1),  
    },  
  },  
})
```

Sub-module names

Parameters within sub-module



# Flax: Linen API



- Run a module with parameters via `module.apply`:

```
example_input = jax.random.normal(jax.random.PRNGKey(0),  
                                   (4, 2))
```

```
model.apply(params, example_input)
```

```
DeviceArray([[ -0.07297172],  
             [ 0.22177655],  
             [-0.18521681],  
             [-0.165456  ]], dtype=float32)
```

# Flax: Linen API



- Several common network layers have been predefined in the Linen API, such as:
  - Linear Layers (`nn.Dense`, `nn.DenseGeneral`)
  - Convolutions (`nn.Conv`, `nn.ConvTranspose`, etc.)
  - Normalizations (`nn.BatchNorm`, `nn.LayerNorm`, etc.)
  - Attention mechanisms (`nn.SelfAttention`, etc.)
  - Recurrent Neural Networks (`nn.LSTMCell`, `nn.GRUCell`, etc.)

# Flax: Neural Networks with JAX



- Main aspects of a Neural Network library in JAX:

- How can we implement Neural Network layers as functions?

Linen API



- How do we handle parameters (weights, biases, etc.)?

Pytrees



- How do we optimize the model's parameters?

Optax

- How do we put everything together with JIT support?

TrainState



# Optax: NN Optimization in JAX

- Optax is a gradient processing and optimization library for JAX based on Pytrees
- Provides building blocks and common optimizers (SGD, Adam, etc.) in a similar function-oriented fashion as Flax

```
optimizer = optax.adam(learning_rate=1e-3)
```

Immutable dataclass

```
opt_state = optimizer.init(params)
```

In Adam, momentum and adaptive learning rate parameters

```
updates, opt_state = optimizer.update(grads,  
                                     opt_state,  
                                     params)
```

Update momentum etc., and get parameter updates/changes

```
params = optax.apply_updates(params, update)
```

Creates a new Pytree with updated parameters

# Flax: Training API



- How can we combine the model execution, gradient calculation, and optimizer step, while allowing for Just-In-Time compilation?
- Flax offers a solution with the flax.training sub-library, in particular: TrainState
  - Immutable dataclass with model forward function, parameters, and optimizer (can be extended)

```
from flax.training.train_state import TrainState  
  
model_state = TrainState.create(apply_fn=model.apply,  
                                params=params,  
                                tx=optimizer)
```

# Flax: Training API



- A TrainState object can be used as input argument to a function, on which we may apply function transformations (`jax.grad`, `jax.jit`, etc.)
- Example: binary classification

```
def calculate_loss(state, params, batch):  
    data_input, labels = batch  
  
    logits = state.apply_fn(params, data_input)  
    logits = logits.squeeze(axis=-1)  
  
    loss = optax.sigmoid_binary_cross_entropy(logits,  
                                              labels)  
    loss = loss.mean()  
    return loss
```

Obtain model predictions

Calculate loss (error of model)

# Flax: Training API



- Combine everything into a function that executes a whole training step:

```
grad_fn = jax.value_and_grad(calculate_loss,  
                             argnums=1)
```

Returns both output value (loss) and  
gradients for second input argument  
(parameters)

```
@jax.jit  
def train_step(state, batch):  
    loss, grads = grad_fn(state, state.params, batch)  
    state = state.apply_gradients(grads=grads)  
    return state, loss
```

Creates new TrainState with updated  
optimizer state and parameters



# Flax: Training API



- To train the model, we can now just write a training loop that calls the training function several times for different input batches

```
def train_model(state, data_loader, num_epochs=100):  
    for epoch in range(num_epochs):  
        for batch in data_loader:  
            state, loss = train_step(state, batch)  
    return state
```

```
trained_model_state = train_model(model_state,  
                                  train_data_loader,  
                                  num_epochs=100)
```

# Flax: Neural Networks with JAX



- Main aspects of a Neural Network library in JAX:

- How can we implement Neural Network layers as functions?

Linen API



- How do we handle parameters (weights, biases, etc.)?

Pytrees



- How do we optimize the model's parameters?

Optax



- How do we put everything together with JIT support?

TrainState



# Flax: Neural Networks with JAX



- What we haven't discussed yet:
  - Logging can be done with external libraries (e.g., TensorBoard)
  - Flax supports data loading from any other library (e.g., TensorFlow, PyTorch, etc.)
  - Binding parameters to a specific module for easier evaluation
  - Automatically vectorizing and/or parallelizing via `jax.vmap` and `jax.pmap`
  - Writing a research code framework for minimal code overhead
  - And much, much more...

# Summary

- Flax is a library for NN tools in JAX
- Using immutable dataclasses for more object-oriented programming “feeling”
- Can be combined with several external libraries for optimization, logging, data loading, etc.



```
def calculate_loss(state, params, batch):  
    data_input, labels = batch  
  
    logits = state.apply_fn(params, data_input)  
    logits = logits.squeeze(axis=-1)  
  
    loss = optax.sigmoid_binary_cross_entropy(  
        logits, labels  
    )  
  
    loss = loss.mean()  
    return loss
```

```
grad_fn = jax.value_and_grad(calculate_loss,  
                             argnums=1)
```

```
@jax.jit  
def train_step(state, batch):  
    loss, grads = grad_fn(state, state.params,  
                           state)  
    state = state.apply_gradients(grads=grads)  
    return state, loss
```



# When and why to use JAX with Flax?

## Benefits

- JAX is extremely fast with Just-In-Time compilation
- Function transformations are powerful tools to easily parallelize and vectorize your code
- Function-oriented programming is helpful in areas like meta-learning, where one needs explicit gradients

## Drawbacks

- The code overhead is usually larger than in other frameworks
- Not as user-friendly / fail-safe as other frameworks
- Handling dynamic shapes can be annoying
- Community still considerably smaller than e.g. TensorFlow or PyTorch

Recommendation: if you are doing research and want to get maximum performance out of your code, give JAX a try!

# Goals of this talk

- 1) What features are we looking for in an ML/DL framework?
- 2) What is JAX?
- 3) What sets JAX apart from other frameworks?
- 4) How can train Neural Networks in JAX with Flax?
- 5) Where can I continue my learning journey into JAX with Flax?

# Further resources on JAX with Flax

- The [JAX](#) and [Flax](#) documentations have great introduction tutorials
- If you are interested in seeing JAX with Flax used in practice, and learn new methods in Deep Learning, check out our [UvA Deep Learning tutorials!](#)

DEEP LEARNING 1 (JAX+FLAX)

Tutorial 2 (JAX): Introduction to JAX+Flax

JAX as NumPy on accelerators

Function transformations with Jaxpr

Implementing a Neural Network with Flax

🌟 The Fancy Bits 🌟

🔪 The Sharp Bits 🔪

Tutorial 3 (JAX): Activation Functions

Tutorial 4 (JAX): Optimization and Initialization

Tutorial 5 (JAX): Inception, ResNet and DenseNet

Tutorial 6 (JAX): Transformers and Multi-Head Attention

Tutorial 7 (JAX): Graph Neural Networks

## Tutorial 2 (JAX): Introduction to JAX+Flax

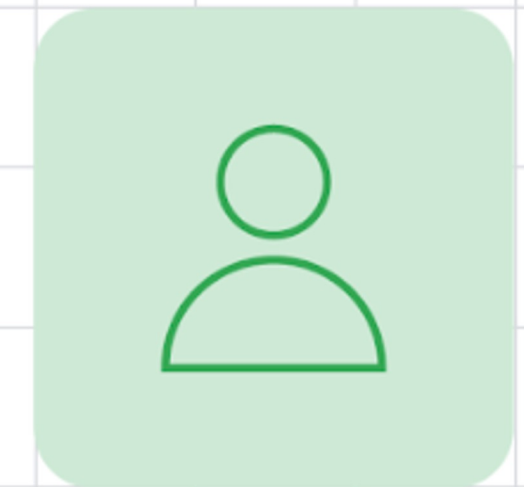
Status **Finished**

Filled notebook: [Repo](#) [View On Github](#) [Open in Colab](#)

Author: Phillip Lippe

Welcome to our JAX tutorial for the Deep Learning course at the University of Amsterdam! The following notebook is meant to give a short introduction to JAX, including writing and training your own neural networks with **Flax**. But why should you learn JAX, if there are already so many other deep learning frameworks like **PyTorch** and **TensorFlow**? The short answer: because it can be extremely fast. For instance, a small GoogleNet on CIFAR10, which we discuss in detail in **Tutorial 5**, can be trained in JAX 3x faster than in PyTorch with a similar setup. Note that for larger models, larger batch sizes, or smaller GPUs, a considerably smaller speedup is expected, and the code has not been designed for benchmarking. Nonetheless, JAX enables this speedup by compiling functions and numerical programs for accelerators (GPU/TPU) *just in time*, finding the optimal utilization of the hardware. Frameworks with dynamic computation graphs like PyTorch cannot achieve the same efficiency, since they cannot anticipate the next operations before the user calls them. For example, in an Inception block of GoogleNet, we apply multiple convolutional layers in parallel on the same input. JAX can optimize the execution of this layer by compiling the whole forward pass for the available accelerator and fusing operations where possible, reducing memory access and speeding up execution. In contrast, when calling the first convolutional layer in PyTorch, the framework does not know that multiple convolutions on the same feature map will follow. It sends each

Google Developers



Thank You!



Phillip Lippe  
GDE Amsterdam  
[@phillip\\_lippe](https://twitter.com/phillip_lippe)  
[phlippe.github.io](https://phlippe.github.io)

Slides  
(personal website)

