# JAX =AutoDiff + XLA

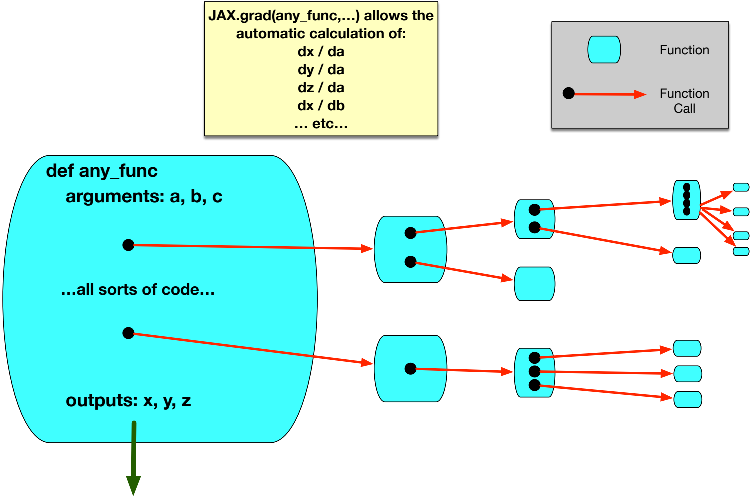
Keith L. Downing

November 29, 2023

# JAX in (a few) Words

- JAX enables "autodiff": the differentiation of arbitrary python and numpy functions.
- This yields gradients for updating key parameters.
- Accelerated Linear Algebra (XLA) can speed up matrix operations without source-code changes.
- Typical application = backpropagation for neural networks, but it is not limited to that.
- It can differentiate across conditionals, iterations, complex data structures, etc.
- This supports many types of optimization, such as finding proper parameters for a PID controller or proper weights for the component terms of a heuristic in A* or Minimax search.
- Although JAX works well, run-time errors are common. It helps to understand a bit of what is happening behind the scenes. Hence, this lecture.

# Main Idea: Differentiating any Python Code



# So What? Why differentiate code?

Answer: To solve optimization problems where a,b, and c are user-controlled parameters and x,y, and z are variables to be optimized (e.g. minimized or maximized).

- Let $\lambda$ = learning rate, and let $q \in \{-1, 1\}$
- IF goal = **maximize** output x, then q = 1. ELSE if goal = **minimize** x, then q = -1.
- After each run of **any_func**, update a, b and c as follows:
  - $a \leftarrow a + q\lambda \frac{\partial x}{\partial a}$
  - $b \leftarrow b + q\lambda \frac{\partial x}{\partial b}$
  - $c \leftarrow c + q\lambda \frac{\partial x}{\partial c}$
- Typical minimization problem: supervised learning with neural nets – minimize output error by changing network weights.
- Typical maximization problem: getting a simulated process to run for the longest amount of time without needing extra resources, or getting a process to produce the most product in a fixed number of timesteps.

# JAX in Action -Simple Examples

```
import numpy as np
import jax
import jax.numpy as jnp
```

- These are standard imports.
- They will not be shown on every slide.
- “jax” may appear as “JAX” for readability and emphasis.

```
def jaxf1(x,y):
    q = x**2 + 8
    z = q**3 + 5*x*y
    return z

def jaxf2(x,y):
    z = 1
    for i in range(int(y)):
        z *= (x+float(i))
    return z

def jaxf3(x,y):
    return x**y

df3a = jax.grad(jaxf3,argnums=0)
df3b = jax.grad(jaxf3,argnums=1)
df3c = jax.grad(jaxf3,argnums=[0,1])

def jaxf4(x,y):
    q = x**2 + 5
    r = q*y + x
    return q*r
```

Note use of iteration

Default: argnums = 0

Differentiate w.r.t. both 0th and 1st arguments

Create enhanced version of jaxf3 that, when called, will produce the derivative of x**y w.r.t. both x and y.

```
>>> df1 = jax.grad(jaxf1)
>>> df1
<function jaxf1 at 0x13f4215e0>
>>> df1(1.0,2.0)
Array(496., dtype=float32, weak_type=True)

>>> jax.grad(jaxf1)(3.0,4.0)  => [5222]
```

Notation: this will be written as [496] in these slides, to avoid clutter.

Create the JAX-traced version of jaxf1 and then call it with arguments 3.0, 4.0

```
>>> df2 = jax.grad(jaxf2)

>>> df2(2.0,3)  =>  [26.0]
```

y = 3 => jaxf2 computes z = x(x+1)(x+2) = x**3 + 3x**2 + 2x
Hence, dz / dx = 3x**2 + 6x + 2
=> dz / dx @(x=2) = 26

```
>>> jax.grad(jaxf3,argnums=[0,1])(3.0,2.0)
  => [6.0, 9.88751]
```

Create and execute

d (x**y) / dx = yx**(y-1)

d (x**y) / dy = ln(x)*x**y

# More Complex...But still Single-Scalar Outputs

```
def jumpinjax(x,n,switch,primes=[2,3,5,7,11]):

    if switch == 0:
        for i in range(int(n)):
            x = x**2
    elif switch == 1:
        for p in primes:
            x = x*p
    else:   return - x
    return x

djuja = jax.grad(jumpinjax) # Build gradient scaffolding
```

Only the function arguments for which JAX is computing the derivative "with respect to" need to be reals.

In this case, it is computing the derivative of jumpinjax "with respect to" X, so X has to be real; the others can be integers.

How do we know it's w.r.t. X?

JAX.grad(func) returns an enhanced version of jumpinjax that will, when called, compute the derivative of func's output value with respect to one or more of its input arguments.

JAX.grad assumes it is differentiating w.r.t. the FIRST argument to a function. To specify other arguments, or more than one argument, use the argnums keyword. For example: djuja_multi = jax.grad(jumpinjax,argnums=[0,1])

Nested function calls are no problem: jumpinjax2 and djuja2 give the same results as jumpinjax and djuja, respectively.

```
def jumpinjax2(x,n,switch,primes=[2,3,5,7,11]):
    n = int(n)   # JAX tracing requires reals, but
    switch = int(switch)
    if switch == 0:    return ranger(x,n)
    elif switch == 1:   return primer(x,primes)
    else:   return - x
    return x

def ranger(y,m):
    for _ in range(int(m)):
        y = y**2
    return y

def primer(x,primes):
    for p in primes:
        x *= p
    return x

djuja2 = jax.grad(jumpinjax2)
```

```
>>> jumpinjax(3,3,0)  =>  6561
>>> jumpinjax(3,3,1)  =>  6930
>>> jumpinjax(3,10,2)  =>  -3

>>> djuja(3.0,3,0) => [17496.0]  # 8*(3)**7 = 17496 = d(x**8)/dx @ x=3

>>> djuja(3.0,3,1) => [2310.0]   # 2*3*5*7*11 = 2310 = d(x*2*3*5*7*11) / dx @ x=3

>>> djuja(3.0,10,2) => [-1.0]  # d(-x / dx) = -1
```

# JAX in Action - Multiple Outputs

```
def jumpinjax3(x,n,switch):
    if switch == 0: return ranger(x,n)
    elif switch == 1:
        return jnp.array([x**i for i in range(n)])
    else:   return - x

djuja3 = jax.jacrev(jumpinjax3)
```

Output = an array of values. JAX needs to take the derivative of EACH value.

This also requires JNP.array instead of NUMPY.array.

Since we may get multiple output values (not just a single scalar), JAX needs to use JAX.jacrev() — Jacobian, reverse mode — instead of JAX.grad()

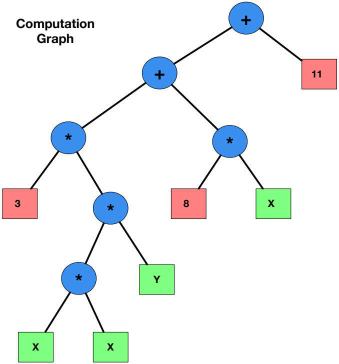
```
>>> jumpinjax3(3,3,0) => 6561

>>> jumpinjax3(3,3,1) => [1, 3, 9]

>>> jumpinjax3(3,3,2) => -3

>>> djuja3(3.0,3,0)
  => [17496.0]  # [d(x**8) / dx] @ x = 3.0

>>> djuja3(3.0,3,1)
  => [0., 1.0, 6.0]  # [d(x**0) / dx , dx/dx, d(x**2) / dx ] @ x = 3.0

>>> djuja3(3.0,3,2)
  => [-1.0]    # [d(-x) / dx ] @ x = 3.0
```

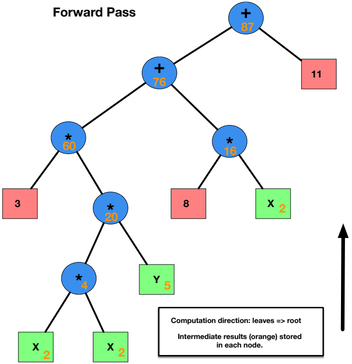
# Reverse-Mode Autodifferentiation

## AutoDiff = Fwd Pass then Bkwd Pass on a Computation Graph

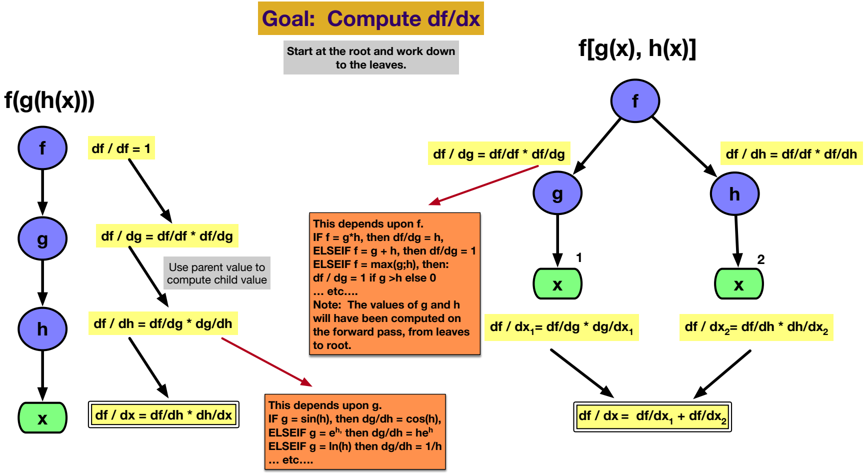
$f(x,y) = 3x^2y + 8x + 11$

$f(2,5) = 3(2)^2(5) + 8(2) + 11 = 87$

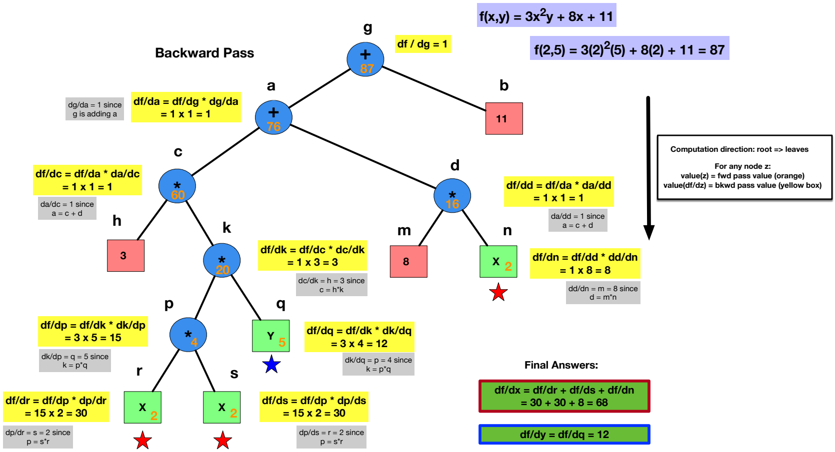


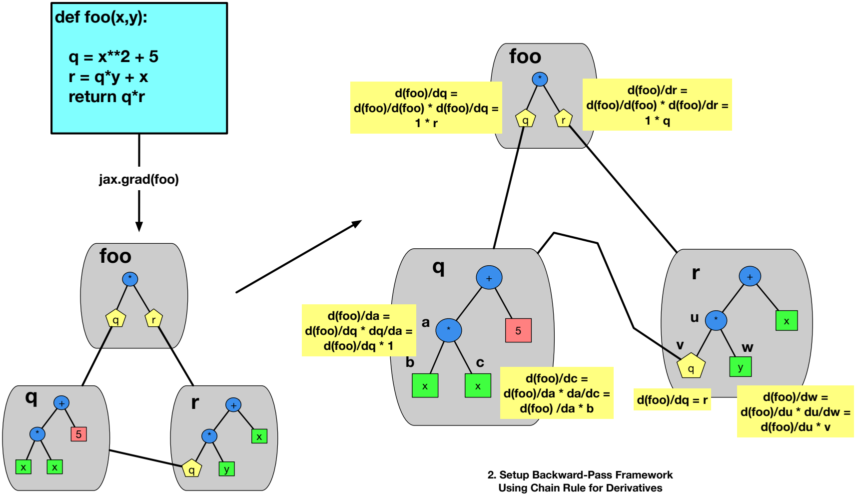
# Basic Calculus for the Backward Pass: Chain Rule



# Reverse-Mode Autodifferentiation: Backward Pass



# JAX.grad




1. Build Computation Graph

2. Setup Backward-Pass Framework
Using Chain Rule for Derivatives

* Use JAX.jacrev, not JAX.grad if the traced function (foo) has several outputs.

# JAX and Side-Effecting Functions

```
bad_news = 1.0

def gum(x,y):
   global bad_news
   bad_news += 10
   return bad_news*x*y**2

dgum = jax.grad(gum,argnums=[0,1])
dgum2 = jax.jit(dgum)  #compiled version
```

d_gum / d_x = bad_news*y**2

d_gum / d_y = 2*bad_news*x *y

```
> gum(2.0,3.0)
  198.0    # bad_news = 11
> gum(2.0,3.0)
  378.0   # bad_news = 21
```

Restart Python

* Uncompiled version handles updated global in gum.

```
> dgum(2.0,3.0)
  [[99.0], [132.0]]     # bad_news = 11
> dgum(2.0,3.0)
  [[189.0], [252.0]]     # bad_news = 21
```

Restart Python

Different behaviors ...not good...

```
> dgum2(2.0,3.0)
  [[99.0], [132.0]]     # bad_news = 1
> dgum2(2.0,3.0)
  [[99.0], [132.0]]     # bad_news = 1
```

* Compiled version ignores all side-effects to globals.

Bottom Line: Only use JAX.grad on "pure functions" = those without global side-effects.

# Wise JAX Convention: Pure Functions

```
def hum(x,y,good_news):
  good_news += 10
  return good_news*x*y**2 , good_news

dhum = jax.jacrev(hum,argnums=[0,1])
dhum2 = jax.jit(dhum)  #compiled version
```

d_hum / d_x = [good_news*y**2, 0.0]

d_hum / d_y = [2*good_news*x*y, 0.0]

Use JAX.jacrev instead of JAX.grad, since hum produces TWO outputs

```
> hum(2.0, 3.0, 1.0)
 ( 198.0 , 11.0 )
> hum(2.0, 3.0, 11.0)
 ( 378.0 , 22.0 )
> hum(2.0, 3.0, 1.0)
 ( 198.0 , 11.0 )
```

No need to restart Python

```
> dhum(2.0, 3.0, 1.0)
  [99.0 ,  132.0]  # d(hum)/ dx (dy)
  [0.0   ,    0.0  ]  # d(good_news) / dx (dy)
> dhum(2.0,3.0,1.0)
  [99.0 ,  132.0]
  [0.0   ,    0.0  ]
> dhum(2.0,3.0,11.0)
  [189.0 ,  252.0]
  [0.0   ,    0.0  ]
```

* Uncompiled version

Identical behavior ...as it should be...

```
> dhum2(2.0, 3.0, 1.0)
  [99.0 ,  132.0]
  [0.0   ,    0.0  ]
> dhum2(2.0,3.0,1.0)
  [99.0 ,  132.0]
  [0.0   ,    0.0  ]
> dhum2(2.0,3.0,11.0)
  [189.0 ,  252.0]
  [0.0   ,    0.0  ]
```

* Compiled version

Any “state” variable to be side-effected should either be a function argument or stored within an argument’s data structure, e.g. a pytree.

# Linear Regression with JAX

Params = $[w_0, w_1, \ldots w_{k-1}, \text{bias}]$ | $w_i$ = ith weight

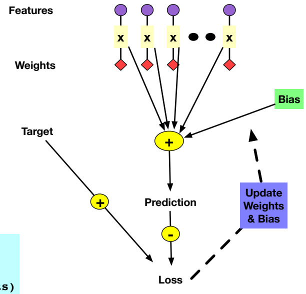
Case = $[f_0, f_1, \ldots f_{k-1}]$ | $f_i$ = ith feature

```
# Running a minibatch of n cases through the model, yielding n predictions

def model(params, cases):
  return jnp.array([ jnp.dot(params[0 : -1], case) + params[-1] for case in cases])
```

```
# The Loss function

def loss_func(params,feature_vectors, targets):
   predictions = model(params, feature_vectors)
   return jnp.mean((predictions - targets)**2)
```

```
# Apply gradient function to the cases and then update the parameters.

def update(loss_gradient, params feature_vectors, targets, learning_rate):
  return params - learning_rate * loss_gradient(params, feature_vectors, targets)
```

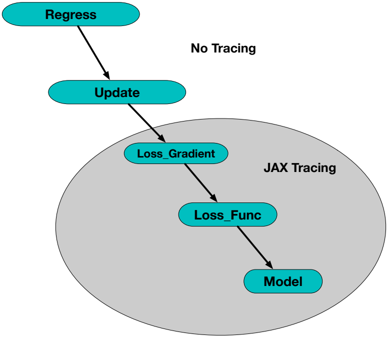
```
# ** Main **

def regress(steps, params, feature_vectors, targets, learning_rate):
    loss_gradient = jax.grad(loss_func)  # Creates gradient function
    for _ in range(steps):
        params = update(loss_gradient,params, feature_vectors, targets , learning_rate)
```




**Goal:** Minimize Loss

**Process:** Update params based on the derivative of loss w.r.t. params

# JAX Tracing



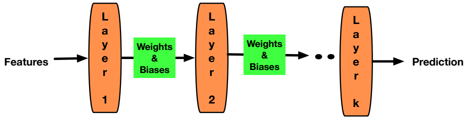

- Gradients can be calculated across many called functions (F).
- JAX builds scaffolding for computing derivatives in all $f \in F$.
- When you see dfoo = JAX.grad(foo) or dfoo = JAX.jacrev(foo) for any function (foo), then foo and all funcs in its call tree are traced.
- These traced functions may require JAX versions of typical numpy functions such as np.array (jnp.array) and np.dot (jnp.dot).
- Avoid side-effects in all these traced functions!

# Backpropagating Neural Net with JAX

```python
import numpy as np
import jax
import jax.numpy as jnp
```

```python
def gen_jaxnet_params(layers=[5,10,5]):
    sender = layers[0]; params = []
    for receiver in layers[1:]:
        weights = np.random.uniform(-.1,.1,(sender,receiver))
        biases = np.random.uniform(-.1,.1,(1,receiver))
        sender = receiver
        params.append([weights, biases])
    return params
```

One set of weights and biases for each non-input layer




Feed the features forward through all layers of the net.

```python
def predict(all_params, features):
    def sigmoid(x): return 1 / (1 + jnp.exp(-x))
    activations = features
    for weights, biases in all_params:
        activations = sigmoid(jnp.dot(activations,weights) + biases)
    return activations
```

Returned activations = the net's output

```python
# Make a batched version of the `predict` function.
# None => all params used on each call,
# 0 => take one row at a time of the cases.  vmap = vector map

batched_predict = jax.vmap(predict, in_axes=(None, 0))
```

```python
def jaxnet_loss(params, features,targets):
  predictions = batched_predict(params, features)
  return jnp.mean(jnp.square(targets - predictions))
```

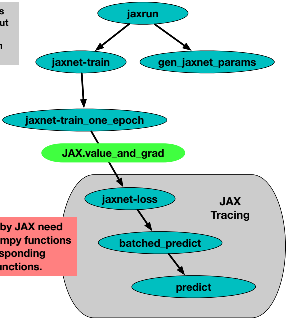
```python
def jaxnet_train_one_epoch(params,features,targets,lrate=0.1):
    mse, gradients = jax.value_and_grad(jaxnet_loss)(params, features,targets)
    return [(w - lrate * dw, b - lrate * db)
            for (w, b), (dw, db) in zip(params, gradients)], mse
```

JAX.value_and_grad is similar to JAX.grad, but it returns the value of jaxnet_loss along with the gradients.

```python
def jaxnet_train(params,features,targets,epochs,lrate=0.1):
    curr_params = params
    for _ in range(epochs):
        curr_params, mse = jaxnet_train_one_epoch(curr_params,features,targets,lrate)
    return current_params
```

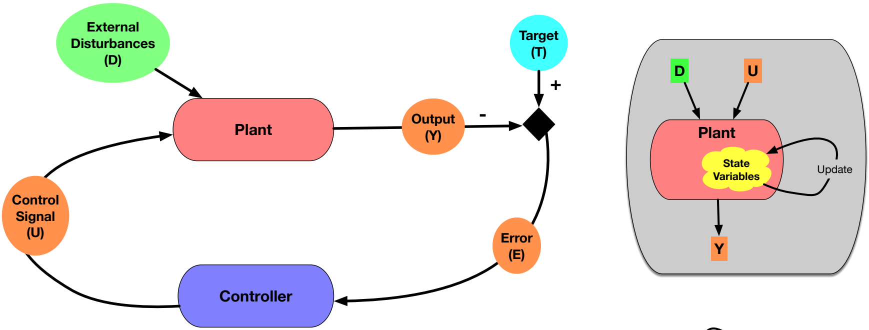
```python
def jaxrun(epochs,ncases,layer_sizes,lrate=0.03):
    features,targets = generate_data_cases(ncases)
    params = gen_jaxnet_params(layer_sizes)
    jaxnet_train(params,features,targets,epochs,lrate=lrate)
```

Functions traced by JAX need to replace some numpy functions with the corresponding JAX.numpy functions.



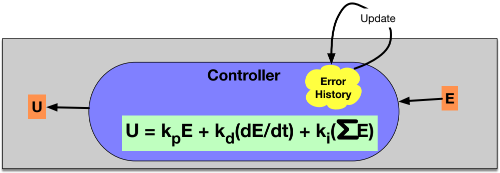
# PID Controller

p = proportional; i = Integral; d = Derivative




The controller needs error history to compute the error's derivative and integral (sum).

The plant can be a complex system (or mathematical model of one), and these typically have state variables that affect the mapping from inputs to outputs.



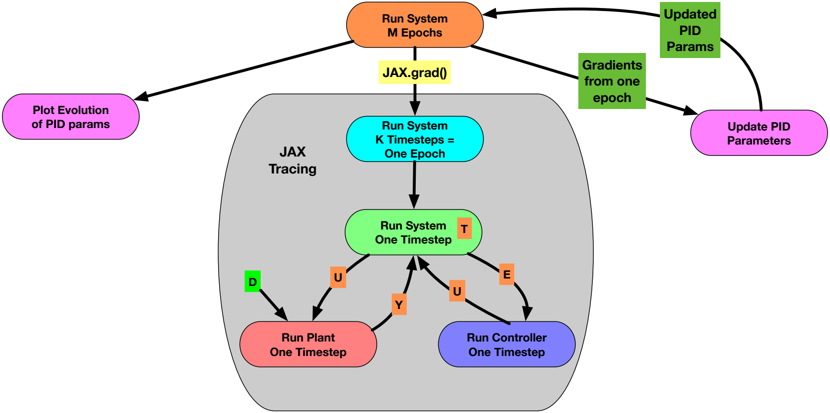
# PID Controller

- The values of the three PID parameters: $k_p$, $k_d$ and $k_i$, will vary with the control problem, i.e. the plant to be controlled.
- Tuning them for a particular plant can be time-consuming.
- Can we use gradient descent to do the job?
- JAX can be very helpful, since it permits tracing of the plant and controller across many timesteps (T) of operation.
- This allows us to compute $\frac{\partial(\sum E)}{\partial k_p}$, $\frac{\partial(\sum E)}{\partial k_d}$ and $\frac{\partial(\sum E)}{\partial k_i}$ or some other useful derivatives, such as $\frac{\partial E_T}{\partial k_p}$, $\frac{\partial E_T}{\partial k_d}$ and $\frac{\partial E_T}{\partial k_i}$.
- Use these derivatives to update parameters (where $\lambda$ = learning rate):

$$k_p = k_p - \lambda \frac{\partial(\sum E)}{\partial k_p}$$
$$k_d = k_d - \lambda \frac{\partial(\sum E)}{\partial k_d}$$
$$k_i = k_i - \lambda \frac{\partial(\sum E)}{\partial k_i}$$

- After updating k's, reset the plant 's state (and controller's error history) and run the coupled plant-controller for another T timesteps.
- Repeat for M epochs.

# JAX Tracing for Adaptive PID Control



# Some Code for Adaptive PID Control

```
def run_system(num_epochs):
    gradfunc = jax.value_and_grad(run_one_epoch, argnums=0)
        .. init params and state
    for _ in range(num_epochs):
        avg_error, gradients = gradfunc(params,state)
            .. execute run_one_epoch via gradfunc
        update_params(params, gradients)   # Use gradients to update controller params
```

Each gradient is the derivative of the average error (i.e. the output of run_one_epoch) w.r.t. one of the controller parameters ( $k_p$, $k_d$ or $k_i$ )

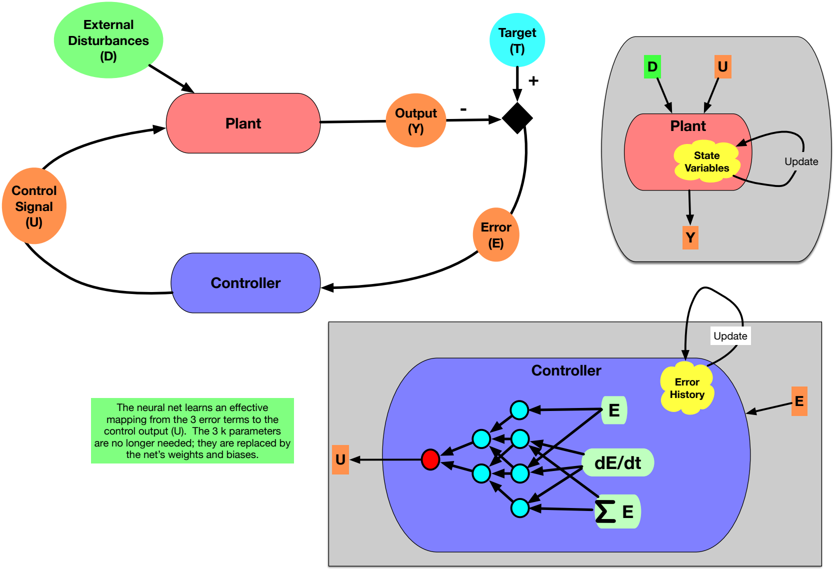
```
def run_one_epoch(params, state):
        .. state gets updated at each timestep
        :
    return avg_of_all_timestep_errors
```

gradfunc = jax.value_and_grad(run_one_epoch,argnums=0)

Create a traced version of run_one_epoch called gradfunc.

When called with the normal arguments to run_one_epoch, this will return both:
a: the normal result (R) of the call to run_one_epoch, and
b: the gradients of R with respect to all values in the 0th argument to run_one_epoch, i.e. params

# Control Using a Neural Network




The neural net learns an effective mapping from the 3 error terms to the control output (U). The 3 k parameters are no longer needed; they are replaced by the net's weights and biases.