

How a Neural Network toolkit works

TensorFlow is the toolkit of primitives that underlies Keras.

It is what powers training and computation in Neural Networks.

Although it might seem mysterious, it (and similar toolkits) is based on a very simple concept.

Here is pseudo-code for the *training loop*

- The part of the Keras framework that implements `fit`
- It solves for the optimal weights \mathbf{W}^* that minimize the Loss function
- Pre-Keras, the user coded this loop for each problem

It is nothing more than Gradient Descent.

```
initialize(W) # Training loop to implement mini-batch SGD for epoch in range(n_epochs):` for X_batch,  
y_batch in next_batch(X_train, y_train, batch_size, shuffle=True): # Forward pass y = NN(X_batch) # Loss  
calculation loss = loss_fn(y, y_batch) # Backward pass grads = gradient(loss, W) # Update W = W - grads *  
learning_rate
```

- We process all the training examples once per epoch
- The epoch is divided into *mini-batches*: disjoint subsets of training examples
- The estimate of the weights is updated in each epoch
- We do this for many epochs, until the Loss function no longer decreases

Each epoch consists of two phases

- A Forward Pass in which inputs are mapped into predictions, for each example in the mini batch
 - An Average Loss is computed over all examples in the mini batch
- A Backward Pass in which gradients of the Average Loss are computed
 - And used to update the weights

The Forward and Backward API

There is a clever "trick" that facilitates

- Computation of predictions (Forward Pass)
- Computation of analytical derivatives (Backward Pass)

Each atomic operation is implemented by an Object-Oriented Class

The class implements methods

- forward for the Forward Pass
- backward for the Backward Pass

This trick is repeated many times, for each atomic operation.

That's all there is to it: Consistent application of a simple trick !

Let's illustrate using the Multiplication operation.

Inside the Forward Pass

The essential part of the Forward Pass is computing layer l 's output $\mathbf{y}_{(l)}$ from the layer's input $\mathbf{y}_{(l-1)}$ and the layer's weights $\mathbf{W}_{(l)}$.

$$\mathbf{y}_{(l)} = a_{(l)}(f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)}))$$

For simplicity of presentation, we will temporarily assume that the activation $a_{(l)}$ is the identity function.

(Without loss of generality, we can implement the activation as a separate layer that also obeys the per layer logic we are about to present).

Consider the atomic operation of multiplication $x * y$

We define a class `MultiplyLayer`

- derived from parent class `Layer`, which requires the forward and backward methods

Here is the code for the Forward Pass

```
class MultiplyLayer(Layer): """ A layer that multiplies its two inputs (x,y) """
    def forward(self, x, y): """ The forward pass: compute the product of x, y
    Parameters ----- x, y: ndarrays Returns ----- z: ndarray
    that is the product of x and y """
```

Compute the product $z = x * y$ # Remember the two inputs: we will need to take derivatives with respect to each self.x, self.y = x, y return z

Not surprisingly

- The key statement is the one that multiplies the two inputs
- And returns the product

Just as you would expect.

But also notice that we are saving the two multiplicands (x and y).

We will need them for the Backward Pass.

Inside the Backward Pass

The job of the Backward Pass is

- To take the Loss gradient $\mathcal{L}'_{(l)}$ for the layer
- Compute the Loss gradient $\mathcal{L}'_{(l-1)}$ to "flow backwards" to the previous layer
- Compute the Local gradients
- Obtain the derivative with respect to $\mathbf{W}_{(l)}$, the layer's weights, using the Loss and Local gradients

That is, the derivative of the Loss with respect to the layer's weights

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{(l)}}$$

is computed via the Chain Rules as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{W}_{(l)}} = \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{W}_{(l)}}$$

Here is the code


```
def backward(self, dz): """ The backward pass: - update the derivative of the loss function (to the
derivative wrt the output of the prior layer) - compute the derivatives of the loss function with respect to
each input Parameters ----- dz: scalar. "loss gradient":  $dL/dz$ : - The derivative of the loss wrt the
output (z) of this layer Returns ----- [dx, dy]: "local gradients" wrt inputs [x,y]
```

```
# Compute loss gradient of this layer, given that of successor layer # Loss gradient this layer = Upstream
Loss gradient * local gradient  $dL/dx = dL/dz * dz/dx$   $dL/dy = dL/dz * dz/dt$  Since this layer's operation is
multiplication,  $z = x*y$  - "local gradients" are  $dz/dx = y$ ,  $dz/dy = x$   $dz$  is given as input "" local_grad_x =
self.y local_grad_y = self.x  $dx = local\_grad\_x * dz$   $dy = local\_grad\_y * dz$  return [dx, dy]
```

The backward method

- Takes the loss gradient $\mathcal{L}'_{(l)} = \frac{\partial \mathcal{L}}{\partial y_{(l)}}$ as input via formal parameter dz
 - Variable dz denotes $\frac{\partial \mathcal{L}}{\partial z}$. the derivative of the loss with respect to z
 - Which is the loss gradient $\frac{\partial \mathcal{L}}{\partial y_{(l)}}$
 - Since variable z is the name for $\mathbf{y}_{(l)}$

- Computes the local gradients $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$

$$\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} = \left[\frac{\partial \mathbf{y}_{(l)}}{\partial x}, \frac{\partial \mathbf{y}_{(l)}}{\partial y} \right] \quad \text{Since } \mathbf{y}_{(l-1)} = [x, y]$$

$$= \left[\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y} \right] \quad \text{Since } z = y_{(l)}$$

$$= \left[\frac{\partial (x*y)}{\partial x}, \frac{\partial (x*y)}{\partial y} \right] \quad \text{Since } z = x * y$$

$$= [y, x] \quad \text{Since } z = x * y$$

- `local_grad_x`, `local_grad_y` are the variables that store the local gradients

- Multiplies the loss gradient $\mathcal{L}'_{(l)}$ (stored in variable dz)
- By the local gradients $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$ (stored in variables `local_grad_x`, `local_grad_y`)
- To compute the product which is $\mathcal{L}'_{(l-1)}$
- Returned as [dx, dy]

Thus, the backward method flows the loss gradient "backwards" one layer

- And facilitates the computation of

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{W}_{(l)}} = \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{W}_{(l)}}$$

- In the multiply layer, there are no weights $\mathbf{W}_{(l)}$

Now you can see why the `forward` method stored the multiplicands x , y

- They were needed as $[y, x] = [\frac{\partial(x*y)}{\partial x}, \frac{\partial(x*y)}{\partial y}]$

Conclusion

The whole basis of toolkits for Neural Networks is this simple Module API consisting of methods

- forward
- backward

Knowing this: you can implement *your own* operations if you ever find that necessary.

That is how more complex layers are implemented (e.g., Convolution).

Hopefully this demystified the notion that Neural Network toolkits are complicated.

In [5]: `print("Done")`

Done