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

Recall the computation that takes $\mathcal{L}'_{(l)}$ as input and produces $\mathcal{L}'_{(l-1)}$ as output

$$\begin{aligned}
\mathcal{L}'_{(l-1)} &= \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l-1)}} \\
&= \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} \\
&= \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}
\end{aligned}$$

And to compute the derivative of the Loss with respect to the layer's weights

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

the Chain Rules gives us

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

But note: there are **no** weights in a Multiplication layer !

So we only need to compute $\mathcal{L}'_{(l-1)}$ in this case.

Were the operation to have weights, the code logic would be very similar to this case.

In [3]:

```
  def backward(self, dL_dz):
        """
        This layer computes:
           z = x * y
        on the forward pass.

        The backward pass:
        - Computes dL_dzz:
          - the derivative of the loss wrt the output zz of the previous layer
 (which are this layer's inputs)
          - where zz = [ x, y ]
        - Computes dL_dW
          - Where W are the weights of this layer (not applicable for this laye
r)

        Parameters
        ----------
        dL_dz: scalar.  "loss gradient": dL/dz :
        - The derivative of the loss wrt the output (z) of this layer

        Returns
        --------
        [ dL_dW, dL_dzz ] where
        - dL_dW is derivative of Loss wrt weights (not applicable for multiplica
tion)
        - dL_dzz =  [dL_dx, dL_dy] is derivative of Loss wrt to prior layer's ou
tputs zz = [ x, y ]
        """
```

""" Since this layer's operation is multiplication, z = x*y dz/dx = y, dz/dy = x """ dz_dx = self.y, dz_dy = self.x # Chain rule dL_dx = dL_dz * dz_dx dL_dy = dL_dz * dz_dy dL_dzz = [dL_dx, dL_dy] # No weights W for this layer dL_dW = null return [dL_dW, dL_dzz]

The `backward` method takes the loss gradient $\mathcal{L}'_{(l)} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}} = \frac{\partial \mathcal{L}}{\partial z}$

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

$$
\begin{aligned}
\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} &= \left[\frac{\partial \mathbf{y}_{(l)}}{\partial x}, \frac{\partial \mathbf{y}_{(l)}}{\partial y}\right] && \text{Since } \mathbf{y}_{(l-1)} = [x, y] \\
&= \left[\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}\right] && \text{Since } z = y_{(l)} \\
&= \left[\frac{\partial (x*y)}{\partial x}, \frac{\partial (x*y)}{\partial y}\right] && \text{Since } z = x * y \\
&= [y, x] && \text{Since } z = x * y
\end{aligned}
$$

- Multiplies the local gradients by the loss gradient $\mathcal{L}'_{(l)}$ to get $\mathcal{L}'_{(l-1)}$

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 [4]:
```
print("Done")
```

```
Done
```