Straight Through Estimator

Motivation

The key to training a Neural Network

- is Gradient Descent
- implemented through Back Propagation

But there are conditions in which Back Propagation breaks down.

Let's illustrate the issue.

For notational consistency

- we will explain Back Propagation by equating operators and layers
- the operator for layer $\ll$ maps input $\backslash\mathbf{y}_{(\ll-1)}$ to output $\backslash\mathbf{y}_{\backslash\mathrm{llp}}$
- this works for operators organized in non-layered architecture as well

Our eventual goal is to compute the gradient of the Loss with respect to the weights $\backslash\mathbf{W}_{\backslash\mathrm{llp}}$ of each layer $\ll$

$$\frac{\partial \backslash\mathrm{loss}}{\partial \backslash\mathbf{W}_{\backslash\mathrm{llp}}}$$

Back Propagation achieves this

- through repeated use of the Chain Rule
- starting from the deepest layer (head) and proceeding to the shallowest layer (input)

We define the Loss Gradient for layer i as

$$\backslash\mathrm{loss}'_{\backslash\mathrm{llp}} = \frac{\partial \backslash\mathrm{loss}}{\partial \backslash\mathrm{y}_{\backslash\mathrm{llp}}}$$

It is the derivative of $\backslash\mathrm{loss}$ with respect to the output of layer $\ll$, i.e., $\backslash\mathrm{y}_{\backslash\mathrm{llp}}$.

The desired gradient to update the weights follows by the chain rule using

- the Loss Gradient $\backslash\mathrm{loss}'_{\backslash\mathrm{llp}}$
- the local gradient $\frac{\partial \backslash\mathrm{y}_{\backslash\mathrm{llp}}}{\partial \backslash\mathrm{W}_{\backslash\mathrm{llp}}}$

$$\frac{\partial \backslash\mathrm{loss}}{\partial \backslash\mathrm{W}_{\backslash\mathrm{llp}}} = \backslash\mathrm{loss}'_{\backslash\mathrm{llp}} * \frac{\partial \backslash\mathrm{y}_{\backslash\mathrm{llp}}}{\partial \backslash\mathrm{W}_{\backslash\mathrm{llp}}}$$

Back propagation inductively updates the Loss Gradient from the output of layer $\ll$ to its inputs (e.g., prior layer's output $\backslash\mathbf{y}_{(\ll-1)}$)

- Given $\backslash\text{loss}'_{\backslash\text{llp}}$
- Compute $\backslash\text{loss}'_{(\ll-1)}$
- Using the chain rule

$$
\begin{aligned}
\backslash\text{loss}'_{(\ll-1)} &= \frac{\partial\backslash\text{loss}}{\partial\backslash\mathbf{y}_{(\ll-1)}} \\
&= \frac{\partial\backslash\text{loss}}{\partial\backslash\mathbf{y}_{\backslash\text{llp}}}\frac{\partial\backslash\mathbf{y}_{\backslash\text{llp}}}{\partial\backslash\mathbf{y}_{(\ll-1)}} \\
&= \backslash\text{loss}'_{\backslash\text{llp}}\frac{\partial\backslash\mathbf{y}_{\backslash\text{llp}}}{\partial\backslash\mathbf{y}_{(\ll-1)}}
\end{aligned}
$$

The loss gradient "flows backward", from $\backslash\mathbf{y}_{(L+1)}$ to $\backslash\mathbf{y}_{(1)}$.

This is referred to as the *backward pass*.

That is:

- the upstream Loss Gradient $\backslash\mathbf{loss}'_{\backslash\mathbf{llp}}$
- is modulated by the local gradient $\frac{\partial \backslash\mathbf{y}_{\backslash\mathbf{llp}}}{\partial \backslash\mathbf{y}_{(\ll -1)}}$
- where the "layer" is the operation transforming input $\backslash\mathbf{y}_{(\ll -1)}$ to output $\backslash\mathbf{y}_{\backslash\mathbf{llp}}$

The problematic issue for Back-Propagation is the local gradient term

$$\frac{\partial \backslash \mathbf{y}_{\backslash \mathrm{llp}}}{\partial \backslash \mathbf{y}_{(\ll -1)}}$$

that relates the output of an operation (layer) to its input.

What happens when the operation

- is non-differentiable
- or has zero derivative almost everywhere
- is non-deterministic (e.g., `tf.argmin` when two inputs are identical)

Either

- the backward gradient flow is killed (zero derivative)
- or can't be computed (non-differentiable or non-deterministic) **analytically**

Solution: Straight Through Estimator (STE)

Suppose *for the purpose of Back Propagation only* we replace the problematic

$$\frac{\partial \backslash\mathbf{y}_{\backslash\text{llp}}}{\partial \backslash\mathbf{y}_{(\ll -1)}}$$

with the gradient of a *proxy function* $\tilde{\backslash\mathbf{y}}$

$$\frac{\partial \tilde{\backslash\mathbf{y}}_{\backslash\text{llp}}}{\partial \backslash\mathbf{y}_{(\ll -1)}}$$

That is, for the purpose of computing gradients

- we treat the operator as mapping $\backslash\mathbf{y}_{(\ll -1)}$ to $\tilde{\backslash\mathbf{y}}_{\backslash\text{llp}}$ rather than $\backslash\mathbf{y}_{\backslash\text{llp}}$

A common choice for the proxy function is the *identity function*

- operator implements $\tilde{\backslash\mathbf{y}}_{\backslash\mathrm{llp}} = \backslash\mathbf{y}_{(\ll -1)}$
- hence the formerly problematic gradient

$$\frac{\partial \backslash\mathbf{y}_{\backslash\mathrm{llp}}}{\partial \backslash\mathbf{y}_{(\ll -1)}}$$

is replaced by

$$\begin{aligned} \frac{\partial \tilde{\backslash\mathbf{y}}_{\backslash\mathrm{llp}}}{\partial \backslash\mathbf{y}_{(\ll -1)}} &= \frac{\partial \backslash\mathbf{y}_{(\ll -1)}}{\partial \backslash\mathbf{y}_{(\ll -1)}} \quad \text{since proxy implements } \tilde{\backslash\mathbf{y}}_{\backslash\mathrm{llp}} = \backslash\mathbf{y}_{(\ll -1)} \\ &= 1 \end{aligned}$$

The Loss Gradient flows through the operator unchanged !

Allowing the Gradient to flow backwards unchanged allows us to construct something called a *Straight Through Estimator*

- treat problematic layer $\ll$ as a pass-through of input $\backslash\mathbf{y}_{(\ll-1)}$ to $\backslash\mathbf{y}_{\backslash\text{llp}}$

A consequence of using a Straight Through Estimator is that

- Back propagation does not compute the true gradient
$$\frac{\partial\backslash\text{loss}}{\partial\backslash\mathbf{W}_{\backslash\text{llp}}}$$
- but results in a value g_i (referred to as the *coarse gradient*)
    - not clear whether g_i is the gradient of any true function

This seems disturbing at first.

But recall that the purpose of computing the true gradient

- is to update weights $\backslash\mathbf{W}_{\backslash\mathrm{llp}}$

$$\backslash\mathbf{W}_{\ll} = \backslash\mathbf{W}_{\backslash\mathrm{llp}} - \alpha * \frac{\partial \backslash\mathrm{loss}}{\partial \backslash\mathbf{W}_{\backslash\mathrm{llp}}}$$

- **As long as the direction $\boldsymbol{g_i}$ has a non-zero correlation with $\frac{\partial \backslash\mathbf{loss}}{\partial \backslash\mathbf{W}_{\backslash\mathbf{llp}}}$**
    - the weight update step will move in the direction of reducing the Loss

Hence, the Straight Through Estimator can be used for the purpose of Gradient Descent.

Proxy functions

Under certain assumptions (https://arxiv.org/pdf/1903.05662.pdf#page=5)

- a non-zero correlation can be established between the true and coarse gradients
- for some specific proxy functions

Interestingly enough: the Identity function is **not** [one of those functions (https://arxiv.org/pdf/1903.05662.pdf#page=8)](https://arxiv.org/pdf/1903.05662.pdf#page=8) !

- ReLU and Clipped ReLU *are* such functions
    - these are more commonly used as Activation Functions
    - but here they just map an input to an output

Nonetheless: the Identity function is commonly used as the proxy

- it may initially lead to decreased Loss
- but will may stop decreasing the loss as it approaches a local minimum

Implementing a Straigh Through Estimator in TensorFlow

Stop Gradient operator

The *Stop Gradient* operator `sg` in TensorFlow

- acts as the identity operator on the Forward Pass
$$\text{sg}(\backslash\mathbf{x}) = \backslash\mathbf{x}$$
- But on the Backward Pass of Backpropagation: *it stops the gradient* from flowing backwards
$$\frac{\partial\,\text{sg}(\backslash\mathbf{x})}{\partial\backslash\mathbf{y}} = 0 \text{ for all } \backslash\mathbf{y}$$

We can use the Stop Gradient operator

- to prevent the computation of a gradient for a problematic operation/layer
- but we must take an extra step to allow the Loss Gradient to flow backwards through the problematic operation/layer

Implementing Straight Through Estimation using Stop Gradient

Consider the implementation of an operator `ProblemOp`

- taking input `in`
- and producing output `out`
- defined by

```
class ProblemOp(layers.Layer):
    def call(self, in):
        # Computation of out:
        #   Problem: RHS of definition of result is NOT differentiable
        result = ...

        # Straight-through estimator.
        out = in + tf.stop_gradient(result - in)

        return out
```

n.b., the `call` method is what implements the Forward Pass in TensorFlow.

The line of code

```
out = in + tf.stop_gradient(result - in)
```

is the implementation of the *Straight Through Estimator**.

It seems odd: mathematically, it just copies `result` to `out`

The line of code

- connects the output of the operator (the LHS of the assignment)
$$\backslash\mathbf{y}_{\backslash\text{llp}} = \text{out}$$
- to the input of the operator (appearing on the RHS of the assignment)
$$\backslash\mathbf{y}_{(\ll -1)} = \text{in}$$

- causing the Loss Gradient to flow from `out` to `in` during Back Propagation

    - unchanged

    - because the `sg` operator creates a zero gradient during Back Propagation

$$\frac{\partial \text{ tf.stop\_gradient}(\text{result - in})}{\partial \ldots} = 0$$

Thus, a Straight Through Estimator using an Identity proxy function has been created by the odd-looking statement.

Note too that the line of code

- **also** connects the output (LHS of assignment) to `result` (appearing on RHS of assignment)
- but, because of Stop Gradient
    - no gradient flows from `out` to `result`

Putting the problematic operation (calculation of `result`) within a Stop Gradient

- solves the potential issue of computing a problematic gradient

More formally:

Let's interpret this line of code as a layer $\ll$

$$
\begin{aligned}
\textcolor{red}{\backslash \mathbf{y}_{\backslash \text{llp}}} &= \text{in} \quad + \quad \text{tf.stopgradient(result -in)} && \text{definition of out} \\
\textcolor{red}{\backslash \mathbf{y}}_{(\ll -1)} &= [\text{in}, \text{result}] && \text{inputs to layer}
\end{aligned}
$$

Then

$$
\begin{aligned}
\frac{\partial \backslash\mathrm{y}_{\backslash\mathrm{llp}}}{\partial \backslash\mathrm{y}_{(\ll -1)}} &= \frac{\partial \text{ out}}{\partial \text{ [in,result]}} && \text{definitio} \\
&= \frac{\partial \text{ in } + \text{ tf.stopgradient(result -in)}}{\partial \text{ [in,result]}} && \text{definitio} \\
&= \frac{\partial \text{ in}}{\partial \text{ [in,result]}} \quad + \quad \frac{\partial \text{ tf.stopgradient(result -in)}}{\partial \text{ [in,result]}} && \text{derivati} \\
&= [1, 0] \quad + \quad [0, 0] && \frac{\partial \text{ in}}{\partial \text{ in}} = 1 \\
& && \frac{\partial \text{ in}}{\partial \text{ result}} = \\
& && \frac{\partial \text{sg(..)}}{\partial ..} = \\
& \quad [1, 0]
\end{aligned}
$$

Thus

$$\frac{\partial \text{ out}}{\partial \text{ [in,result]}} = [1, 0]$$

This above shows

- the derivative flows backwards to `in` undiminished (gradient of `out` w.r.t `in` is 1)
- NO derivative flows backwards to `result` (gradient of `out` w.r.t `result` is 0)

In [2]:

```
print("Done")
```

Done