CHAIN RULE:

This is one of the core principle of calculus and it is used whenever we need to use a derivative even reasonably complex. The chain rule states that the derivative of

$$f(g(x)) = f'(g(x)) \cdot g'(x)$$

In other words, it helps us differentiate "composite functions". For example, $sin(x^2)$ is a composite function because it can be constructed as

$$f(g(x)) \text{ for } f(x) = sin(x) \text{ and } g(x) = x^2$$

Using the chain rule and the derivatives of $sin(x)$ and x^2, we can find the derivative of $sin(x^2)$.

Let's take this example and calculate the chain rule:

$$h(x) = (sin\ x)^2$$

$$\frac{\delta h(x)}{\delta x} = \frac{\delta h(x)}{\delta (sin\ x)} \frac{\delta (sin\ x)}{\delta x}$$

$$= \frac{\delta [(sin\ x)^2]}{\delta (sin\ x)} \frac{\delta (sin\ x)}{\delta x}$$

$$= 2(sin\ x)\ cosx$$

As subsequently, as we have studied in our calculus class, given the value of $sin\ x$ and $cos\ x$ we can find the value.

AUTOMATIC DIFFERENTIATION:

Automatic Differentiation is a technical term refers to a specific family of techniques that a specific family of techniques that compute derivatives through accumulation of values during code execution to generate numerical derivative evaluations rather than derivative expressions. This aids in accurate evaluation of derivatives at machine precision with only a small constant factor of overhead and ideal asymptotic efficiency. Automatic Differentiation can be applied to our existing code without having to make any large change which further enables branching, loops and recursion in comparison to the effort involved effort involved in arranging code as closed-form expressions under the syntactic and semantic constraints of symbolic differentiation. It is due to

this generality, Automatic Differentiation has been widely accepted in the computing industry and academia and have found applications in optimization, fluid dynamics, atmospheric science etc.

Automatic Differentiation performs a non-standard interpretation of a given computer program by replacing the domain of the variables to incorporate derivative values and redefining the semantics of the operators to propagate derivatives per the chain rule of differential calculus. Despite its widespread use in other fields, general-purpose Automatic Differentiation has been not been used much by the machine learning community until now. Given the success of deep learning as the state of the art finding its usage in various applications that used machine learning and the present day workflow that is based on rapid prototyping and code reuse in frameworks such as Theano, Torch and Tensorflow; the scenario is slowly changing where projects like Autograd, Chainer and PyTorch are pioneers in bringing general purpose Automatic Differentiation to the mainstream.

In machine learning, a specialized counterpart of Automatic differentiation known as backpropagation algorithm has been widely accepted to train neural networks. Backpropagation uses gradient descent to model learnings related to neural network weights, looking out for the minimum value of an objective function. This required value of gradient is obtained by the backward propagation of the sensitivity of the objective value at the output by using the chain rule to compute partial derivatives of the objective with respect to each weight. The algorithm that results out of this equivalent to transforming the network evaluation function composed with the objective function under reverse mode Automatic Differentiation which actually generalizes the backpropagation idea. Putting it in other words, Backpropagation is a specialized case of Automatic Differentiation.

How Automatic Differentiation is different from Symbolic and Numerical Differentiation?

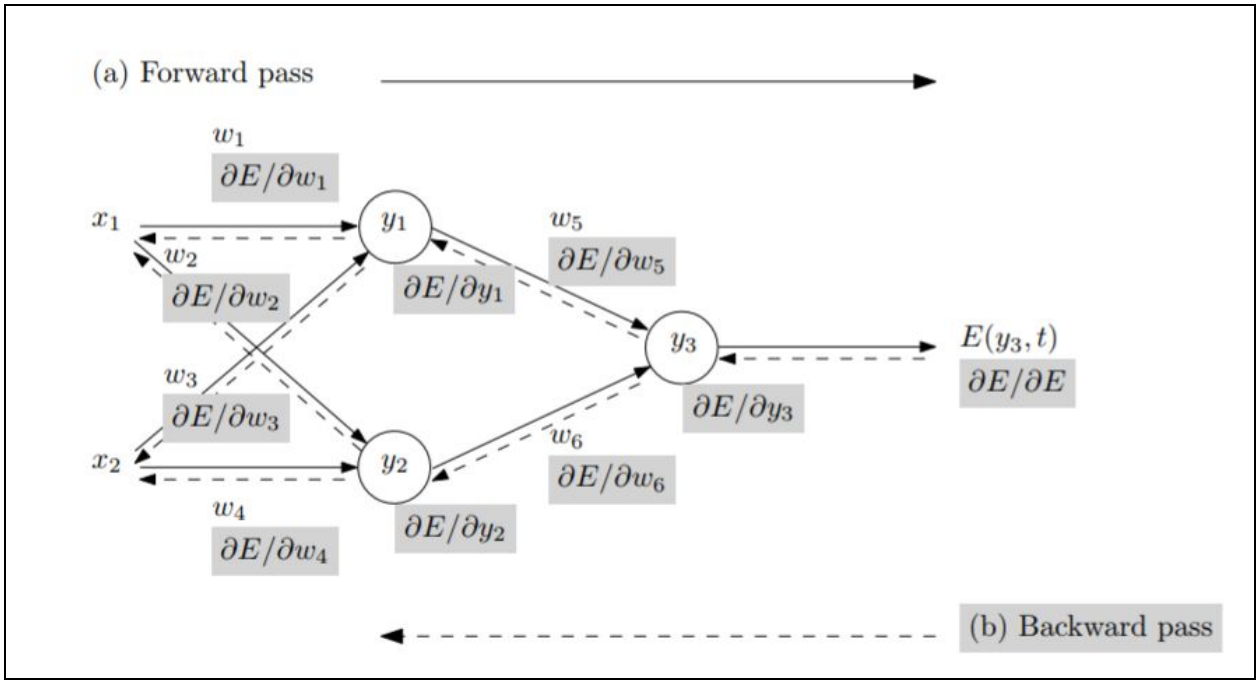


Figure: Overview of Backpropagation

Briefly talking about what is happening inside the above figure, we have 2 phases - Forward Pass (a) and the Backward Pass (b). What is happening in the forward pass is - the training inputs x_i (x_1 and x_2) are fed forward generating the corresponding activations y_i (y_1 and y_2) respectively. An error E is computed between the actual output y_3 and target out t. In the backward pass - the adjoining error is propagated backward, generating the gradients with respect to the weights $\Delta w_i E = \left(\frac{\delta E}{\delta w_1}, \ldots, \frac{\delta E}{\delta w_6}\right)$ which is subsequently used in gradient descent procedure. The gradient with respect to the inputs $\Delta x_i E$ can be calculated in the same backward pass.

Automatic Differentiation is different from Numeric Differentiation:

Numerical differentiation is the finite difference approximation of derivatives using values of the original function evaluated at some sample points. It is based on the limit definition of a derivative.

Numerical Differentiation relies on the definition of the derivative -

$$\frac{\delta f(x)}{\delta x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In this equation you can easily calculate the left hand side, by evaluating the right hand side with a very minimal non zero value of h. Even though this has the advantage of easy to code, but brings in

the disadvantage of costing *O(n)* evaluation of *f* for gradients in *n* dimension (where n can be as large as millions and billions in case of deep learning models).

Numerical approximations of derivatives are inherently ill-conditioned and unstable, with the exception of complex variable methods that are applicable to a limited set of holomorphic functions due to the introduction of truncation and round-off error imposed by the limited precision of computations and the chosen value of the step size *h*. Truncation error tends to reduce as $h \rightarrow 0$ (h decreases towards zero). However, as h is decreased, round-off error increases and becomes prevalent. Various techniques have been developed to mitigate these errors but they simply increase programming complexity and consumes a lot of computation power.

Automatic Differentiation is different from Symbolic Differentiation:

Symbolic differentiation is the automatic manipulation of expressions for obtaining derivative expressions, carried out by applying transformations representing rules of differentiation. This is studied in calculus 101. The equation is -

$$\frac{\delta}{\delta x}(f(x) + g(x)) = \frac{\delta}{\delta x}f(x) + \frac{\delta}{\delta x}g(x)$$

$$\frac{\delta}{\delta x}(f(x)\,g(x)) = \left(\frac{\delta}{\delta x}f(x)\right)g(x) + f(x)\left(\frac{\delta}{\delta x}g(x)\right)$$

When formulae are represented as data structures, symbolically differentiating an expression tree is a perfectly a mechanical process, which are subjected to mechanical automation. It is implemented in computer algebra systems like Maxima, Mathematica and machine learning framework such a Theano. In optimization, symbolic derivatives can give valuable insight into the structure of the problem domain and, in some cases, produce analytical solutions that can eliminate the need for derivative calculation altogether. Whereas, as they get exponentially larger than the expression whose derivative they represent, they no longer remain efficient to runtime.

Considering the rule of multiplication and a function *h(x) = f(x)g(x)* from the above equation, if h is a product, *h(x)* and $\frac{\delta}{\delta x}h(x)$ have some common components, namely *f(x)* and *g(x)*. Also, on the right hand side of the above equation, *f(x)* and $\frac{\delta}{\delta x}f(x)$ appears separately. If we just could symbolically differentiate *f(x)* and used its derivative into the appropriate place, we would have duplicated any computations that is common in between *f(x)* and $\frac{\delta}{\delta x}f(x)$ and nested duplications are not good as they produce exponentially large symbolic expressions which take correspondingly long to evaluate. This problem is known as expression swell.

When we are concerned with the accurate numerical evaluation of derivatives and not so much with their actual symbolic form, it is in principle possible to significantly simplify computations by

storing only the values of intermediate sub-expressions in memory. Moreover, for further efficiency, we can interleave as much as possible the differentiation and simplification steps. This interleaving idea forms the basis of Automatic Differentiation and provides an account of its simplest form: apply symbolic differentiation at the elementary operation level and keep intermediate numerical results in lockstep with the evaluation of the main function. This is Automatic Differentiation in the forward accumulation mode, which we will discuss next.

Automatic Differentiation Techniques: Forward Accumulation Mode and Reverse Accumulation Mode

Forward mode computes directional derivatives:

One of the benefits with using Automatic Differentiation is that it comes in two basic flavors (and therefore, advanced usage permits many combinations and variations). The easier basic flavor to describe and understand (and implement and use) is called forward mode, which computes directional derivatives.

To do forward mode Automatic Differentiation on a program, we need do a nonstandard interpretation of the program, as follows -
Define "dual numbers" as formal truncated Taylor series of the form $x+\varepsilon x'$.
Define arithmetic on dual numbers by $\varepsilon 2=0$, and by interpreting any non-dual number y as $y+\varepsilon 0$.
Therefore:

$$(x+\varepsilon x')+(y+\varepsilon y') = (x+y)+\varepsilon (x'+y')$$

$$(x+\varepsilon x')(y+\varepsilon y') = (xy)+\varepsilon (xy'+x'y)$$

The coefficients of ε is just the corresponding symbolic derivative rules.

$$f(x+\varepsilon x')=f(x)+\varepsilon f'(x)x' \qquad (1)$$

These dual numbers are just data structures for carrying the derivative around together with the undifferentiated answer (called the primal). The chain rule applied to (1) gives two applications

$$f(g(x+\varepsilon x')) = f(g(x)+\varepsilon g'(x)x')$$

$$f(g(x+\varepsilon x')) = f(g(x))+\varepsilon f'(g(x))g'(x)x'$$

where the coefficient of ε on the right hand side is exactly the derivative of the composition of f and g. That means that since we implement the primitive operations to respect the invariant (1), all

compositions of them will too. This, in turn, means that we can extract the derivative of a function of interest by evaluating it in this nonstandard way on an initial input with a 1 coefficient for ε :

$$\frac{df(x)}{dx}|_x \quad = \text{epsilon-coefficient(dual-version}(f)(x+\varepsilon 1))$$

This also generalizes naturally to taking directional derivatives of

$$f: R^n \rightarrow R^m$$

That's all there is to forward mode, conceptually. There are of course some subtleties, but first let's look at what we have: We have a completely mechanical way to evaluate derivatives at a point. This mechanism works for any function that can be composed out of the primitive operations that we have extended to dual numbers. It turns every arithmetic operation into a fixed number of new arithmetic operations, so the arithmetic cost goes up by a small constant factor. We also do not introduce any gross numerical sins, so we can reasonably expect the accuracy of the derivative to be no worse than that of the primal computation.

This technique also naturally extends to arbitrary program constructs—dual numbers are just data, so things like data structures can just contain them. As long as no arithmetic is done on the dual number, it will just sit around and happily be a dual number; and if it is ever taken out of the data structure and operated on again, then the differentiation will continue. The semantic difficulties around what the derivative of a data structure is only arise at the boundary of Automatic Differentiation, if the function of interest *f* produces a data structure rather than a number, but do not embarrass the internal invariants. If you look at it from the point of view of the dual number, Automatic Differentiation amounts to partially evaluating *f* with respect to that input to derive a symbolic expression, symbolically differentiating that expression, and collapsing the result (in a particular way) to make sure it isn't too big; all interleaved to prevent intermediate expression swelling.

Another way to look at forward mode of a function $f: R^n \rightarrow R^m$ is to consider the Jacobian (matrix of partial derivatives) $J_x f$ at some point *x*. The function *f* perforce consists of a sequence of primitive operations applied to various data in its internal storage. If we call this sequence $f_k \bullet \ldots\ldots\ldots \bullet f_2 \bullet f_1$ (for *k* primitive operations, with data flowing from right to left), then the full Jacobian of decomposes as

$$Jf = Jf_k \bullet \ldots\ldots \bullet Jf_2 \bullet Jf_1$$

where the points at which each intermediate Jacobian is taken are determined by the computation of *f* thus far. The best thing about this decomposition is that even though the full Jacobian *Jf* may be a huge dense matrix, each of the pieces Jf_i very sparse, because each primitive f_i only operates on a tiny part of the data (in fact, for f_i unary or binary, $Jf_i - I$ will have just one or two non-zero entries, respectively). So we can look at the result of forward-mode automatic differentiation as an

implicit representation of the point-indexed family of matrices $J_x f$, in the form of a program that computes matrix-vector products $J_x f \bullet v$ given *x* and *v*.

Reverse mode computes directional gradients:

As mentioned in the previous section, forward mode Automatic Differentiation applied to a function *f* amounts to an implicit representation of the Jacobian *Jf* that takes advantage of the factorization of *Jf* into extremely sparse factors Jf_i -

$$Jf = Jf_k \bullet ..... \bullet Jf_2 \bullet Jf_1$$

Forward mode uses this decomposition to compute directional derivatives, which are vector products $J_x f \bullet v$. In modern practice, however, gradients of functions $f : R^n \to R$ are generally much more valuable as they enable multidimensional optimization methods, multiparameter sensitivity analyses, simulations of systems following potentials, etc. The concept for a gradient of $f : R^n \to R$ generalizes trivially to directional gradients of $g : R^n \to R^m$; for x in R^n and u in R^m, the directional gradient of g at *x* with respect to *u* is just the ordinary gradient at x of $g(x) \bullet u$.

The directional gradient is a symmetric concept to directional derivative: where the directional derivative in the direction v is $J_x f \bullet v$, the directional gradient is $u^T \bullet J_x f$. One might therefore hope that the decomposition of

$$Jf = Jf_k \bullet ..... \bullet Jf_2 \bullet Jf_1$$

would make a good implicit representation of *Jf* for directional gradients as well. Indeed it does, but with a twist. The twist is that for directional derivatives, the information from *x* about the point at which to take each Jacobian flows through this product in the same direction as the information from *v* about what to multiply by. For directional gradients, however, they flow in opposite directions, necessitating computing and storing some explicit representation of the decomposition, based on the point *x*, first, before being able to multiply through by u^T to compute the desired gradient. This can be done—the method is called reverse mode—but it introduces both code complexity and runtime cost in the form of managing this storage (traditionally called the "tape"). In particular, the space requirements of raw reverse mode are proportional to the runtime of *f*. There are various techniques for mitigating this problem, but it remains without an ideal solution.

Role of derivatives in Machine Learning and relevance of Automatic Differentiation:

Gradient - Based Optimization: Gradient-based optimization is one of the pillars of machine learning. Gradient based methods make use of the fact that the objective function decreases steeply if one goes in the direction of negative gradient. The convergence rate of gradient-based methods is usually improved by adaptive step-size techniques that adjust the step size η on every iteration. As we have seen earlier, for large *n*, reverse mode Automatic Differentiation provides a highly efficient method for computing gradients. Second-order methods based on Newton's method make use of both the gradient Δf and the Hessian H_f, automatic differentiation also provides a way of automatically computing the exact Hessian, enabling succinct and convenient general-purpose implementations.

Neural Networks, Deep Learning, Differentiable Programming: Training of a neural network is an optimization problem with respect to its set of weights, which can in principle be addressed by using any method ranging from evolutionary algorithms to gradient-based methods. As we have seen earlier, the backpropagation algorithm is only a special case of automatic differentiation: by applying reverse mode AD to an objective function evaluating a network's error as a function of its weights, we can readily compute the partial derivatives needed for performing weight updates.

Computer Vision: Computer Vision is another field that significantly uses deep learning especially after achieving significant results in the works by the pioneers in the field. Apart from deep learning, automatic differentiation can be applied to inverse graphics or analysis by synthesis. Gradient-based optimization in inverse graphics requires propagating derivatives through whole image synthesis pipelines including the renderer. Barrett and Siskind (2013) present a use case of general-purpose Automatic Differentiation for the task of video event detection using hidden Markov models (HMMs) and Dalal and Triggs (2005) presents another use case of object detectors, performing training on a corpus of pre-tracked video using an adaptive step size gradient descent with reverse mode Automatic Differentiation.

Natural Language Processing: A number of NLP applications such as machine translation, language modeling, dependency parsing uses deep learning techniques and have achieved significant improvement in results. Besides deep learning approaches, statistical models in NLP are commonly trained using general purpose or specialized gradient-based methods and mostly remain expensive to train. Training time can be reduced by utilizing distributed training methods.

Automatic Differentiation Implementation considerations:

- First and foremost considerations of Automatic Differentiation implementation is to carefully monitor performance overhead and bookkeeping introduced by the automatic differentiation arithmetic. If the arithmetic are not handled carefully for example - unknowingly allocating data structures for storing dual numbers would involve significant memory access and allocation for every arithmetic operations which would further increase the cost. Similarly operator overloading may introduce method dispatches with attendant cost which will significantly reduce the speed of computation.

- Another considerations that needs to be taken care of is not to encounter "perturbation confusion". This is a situation where two ongoing differentiation targets the same piece of code, the epsilons introduced by both needs to be kept distinct. Such situations may also arise when Automatic Differentiation is nested.

- Automatic Differentiation is not accustomed to floating point arithmetic which would sometimes result in inducing numeric issues which were not initially present in the original calculation.

- If a procedure is approximating an ideal function, automatic differentiation always gives the derivative of the procedure that was actually programmed, which might not be a good approximation of the derivative of the ideal function that the procedure was approximating.

COMPUTATION GRAPHS:

We know that the computation of a neural network are configured by a forward pass (forward propagation) in which we calculate the output, followed by a backward pass (backward propagation) in which we calculate derivatives and gradients. The computation graph has the answer to why does neural network work this way.

Let's look at an example of a computation graph, say, we have an equation $f = [(a + b) * (c - d)]/e$. To calculate the computation graph, let's break this equations down into small parts:

Part 1 is $(a + b)$, consider this as m

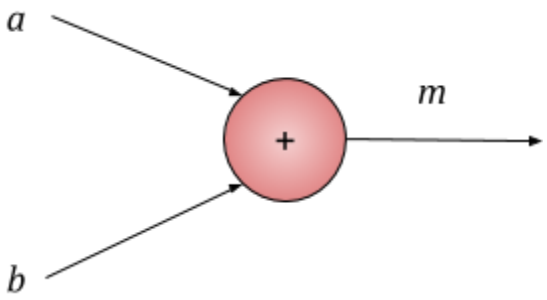
Part 2 is $(c - d)$, consider this as n

Part 3 is $[m * n]$, consider this as s

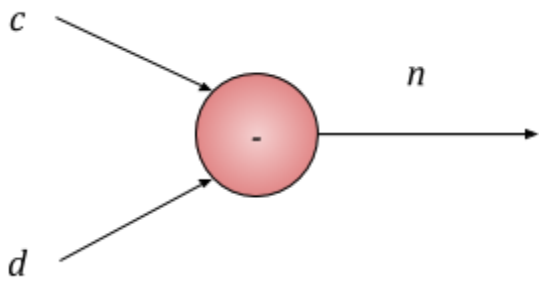
Part 4 is s/e

Now, let's see how the computation graph would look like:

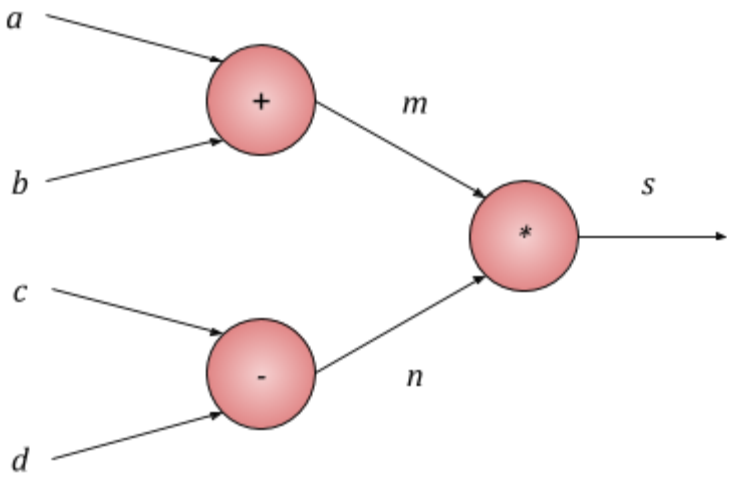
Part 1:



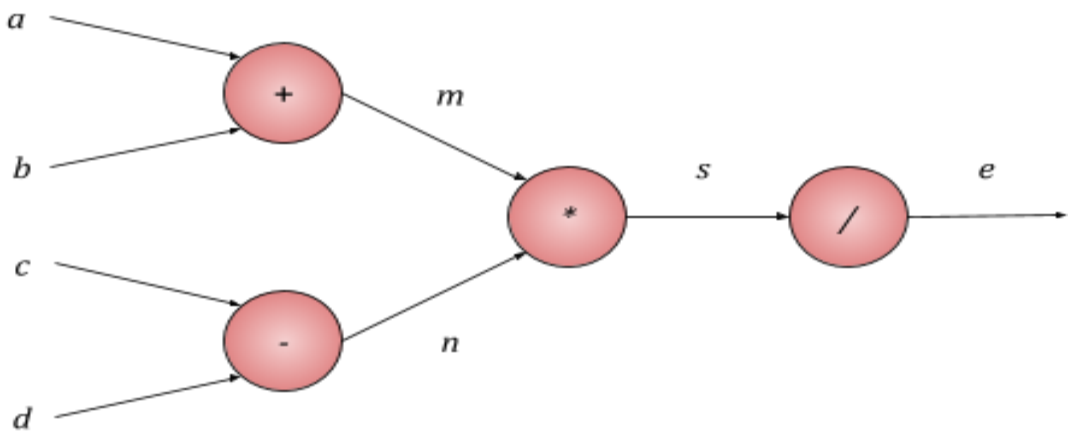
Part 2:



Part 3:



Part 4:



Consider, this kind of computation graphs in case of much larger neural network computation.

Static Computation Graphs vs Dynamic Computation Graphs:
In case of static computation graphs, the complete graph has to be built from scratch before a model can be executed. This follows a define and run approach. Also, separate sessions needs to be specified to make any communication to the outer world. In order to make any change, the whole graph needs to be built from scratch.

In case of dynamic computation graph, things are more flexible and can be defined on the run, Changes in the node can be made while on execution. This really makes debugging your neural network easier.

IMPERATIVE PROGRAMMING:

Imperative Programming is a programming paradigm in which the program describes a sequence of steps to reach a particular goal. On the other hand, declarative programming describes the solution, provide the necessary details required to achieve the objective. It doesn't provide the necessary steps required to achieve a solution. Basically, declarative programming describes "what" a program should accomplish whereas imperative programming explicitly tells the computer "how" to achieve a goal.

Characteristics of Imperative Programming:

- Statements are commands
    - Exact command order is extremely crucial for correct execution.
    - Programmers get to have all the control - algorithm specification, variable declaration, memory management and a lot more.
- Imperative programming work by modifying a program state.

- The statements given reflects machine language instructions.

W.R.T PyTorch -
PyTorch has an imperative / eager computation toolkit. It follows the linear approach of coding which means that you can code top down, interactively write in jupyter notebook and so on.

There are other approaches in which you code an entire neural network from scratch in one environment and then execute in another environment. If you encounter an error in such case, then you would need to get back to the development environment and compare the error code and then debug. This is a kind of non-linear approach of coding and debugging.

PyTorch in the other hand follows a linear approach where you can debug while you code and this is fairly easy as well. But approach is not unique to PyTorch only. Other libraries like Chainer, Dyner, MXNet-gluon, Tensorflow-imperative, Tensorflow-eager uses this approach.

To build this automatic differentiation engine that is imperative and interactive, the fundamental bottleneck that existed among other libraries before PyTorch was launched is that they had huge node creation and bookkeeping overhead. While building PyTorch a lot of care was put into micro optimization and automatic differentiation engine so that the node creation and bookkeeping overhead is not more than 20-30 microseconds which is still large in case of small computation workloads, but suffice in case of bigger workloads. The other options that are available has several milliseconds/seconds of node creation overhead especially if the coding style is interactive.