PLT: Inception (cuz there are so many layers)

By: Andrew Aday, (aza2112) Amol Kapoor (ajk2227), Jonathan Zhang (jz2814)

Proposal

Abstract

In this document we propose *Inception*, a new language developed to make the design and implementation of deep learning models easier and more intuitive. Deep learning is an extremely popular field that is quickly becoming a mainstay of major tech companies like Google and Amazon. Deep learning models consistently beat state of the art algorithms in many fields. Deep learning models are mostly created with libraries built on top of Python, e.g. Tensorflow. Though powerful, these libraries lack visual clarity and are in general cluttered by other unnecessary features coming from a general purpose language like Python. *Inception* is a language built from the ground up specifically for deep learning models that emphasizes visual clarity and minimalism. We believe *Inception* can help deep learning experts program deep learning models the way they think about deep learning models, leading to fewer errors and less programming time.

Overview of domain

In the past few years deep learning has seen an explosion of interest leading to a corresponding explosion in tooling. Most popular deep learning model implementations come from Python based libraries - specifically, Tensorflow, Theano, and PyTorch (there are some C and C++ libraries, but for now these three big ones will suffice for the domain discussion).

Each of these libraries operates by creating a graph structure in sequential run through to represent the model, and then running the generated graph with specific inputs in a separate function call. In other words, these libraries separate graph construction with graph running. These libraries also expose individual variable parameters, allowing users significant flexibility in manipulating the graph. We call these libraries neuron-based libraries, as they focus on neuron level instantiations and manipulations (though they may have helper functions that define full layers/models).

Certain operations and functions are called repeatedly in the deep learning world. In the libraries mentioned above, writing large models is fairly tedious and there is no good way to rapidly duplicate commonly used components. Two wrapper libraries, Slim and Keras, attempt to decrease verbosity and emphasize readability by exposing a large common library of commonly used model layers and handling all neuron interaction in the background. Slim and Keras make writing large models and duplicating functionality significantly easier by optimizing for model creation at a macro level. We therefore call Slim and Keras layer based libraries, instead of neuron based libraries.

Purpose

Inception aims to be a layers-based library like Slim and Keras, optimized for complex deep learning models. *Inception* optimizes for deep learning through *Inception*'s unique syntax: almost everything in the language is defined as a layer (processing) or as an array (data). The minimalist nature of the language makes it extremely easy to quickly define layers that stack into full, complex deep learning models. The language also allows users to quickly identify how a data flows through a model (i.e. what components of a piece of code are processing and what components are data).

Language Outline

Types

Inception will have five types: arrays, layers/functions, floats, booleans, and strings. As in python, types will be inferred by the variables when stored and will throw a runtime (compiler?) error when incorrect types are passed between layers.

arrays: arrays are the fundamental building block of our language and most inputs to and outputs from layers will be a array. We borrow the notion of arrays from Matlab and python's numpy library to refer to a collection of numbers with any number of dimensions rather than the one-dimensional type common in many languages. Unlike numpy and to a lesser extent Matlab, our arrays can be off dimensional (i.e. `[[1, 2, 3], [1, 2], [1]]`) and it is up to the layers/functions to interpret these correctly or throw an error.

layers/functions: In Inception, layers and functions are essentially and functionally the same entity. They both receive input via our arrow notation, operate on input, and produce output that can be assigned to a variable or passed to a different layer. Both complex (e.g. neural-network convolutions, backpropagation, etc.) and simple (arithmetic, modulo, etc.) operations are implemented and can be used via layers. (See syntax section)

Floats: basic 32-bit floating point numbers used to do calculations. We decided against including integer types as the majority of neural network applications required floating point numbers

Booleans: True or False

Strings: Strings will mostly be used either as inputs (in the form of an array) to a network layer and for clean display of data.

Operators

=> By far the most common operator used in Inception will be the arrow symbol (=>). This operator is similar to a pipe in standard unix systems with some added functionality. Generally, the right arrow takes the output generated by the previous layer or variable, combines it with additional inputs if provided, and either sends the aggregated data to the next layer or saves it in a variable.
An example as below

```
layer1() => layer1_data, auxiliary_data => layer2() => Print()
```

would be equivalent to

```
layer1_data = layer1()
Print(layer2(layer1_data, auxiliary_data))
```

in common imperative languages. Our motivation behind using this arrow notation is the inherent feed forward nature of artificial neural network where layers are stacked atop one another and the outputs from one layer is passed as the inputs to the next. The goal is for our language to help the programmer better visualize and implement this structure.
Additionally, the arrow notation will extend into and from control sequences. For example the code

```
input =>
for (i = 0; i < 5; i++):
      layer1() =>
```

would pass `input` through a sequence of five `layer1`'s.

+, -, /, * : For now, arithmetic operators will only work with our float types and we will support parenthesis with them. For arrays, we will use special arithmetic layers to indicate pairwise operations within the array. However, we may consider implementing syntactic sugar to do array operations with the standard operators for floats. These will not work on strings or booleans.

Anonymous Functions?

Syntax

Layer Calls: Layers are differentiated from common variables by the addition of a set of parenthesis (a function "call"). We allow parameter values to be input within the parentheses but "true" inputs should be passed via the right arrow notations.

def: Like in python, this keyword will be used to define a function/layer. def must be followed by a name identifier and a set of parentheses which can be empty or include the names of input arguments and parameters to be used by the program. All of the instruction within the scope of a given function should be indented four spaces (as in python).

if/else: We borrow the if/else statements precisely from python with the same colon syntax and indentation rules. One question to consider is if outputs can be right-arrow passed directly into if statements (i.e. `layer1() => if ==True: ...`) to synergize with the continuity of passing inputs directly via =>. This would make the if statement essentially a layer

for (i = <int>; i < N; i++): We support C and Java style looping.

Example Script

As the goal of our language is an efficient and clear method to describe and use Neural Networks, our canonical example is a minimalistic two layer network that can be used for simple learning and classification tasks such as the popular MNIST Handwriting recognition.

```
def MNIST(images, labels):
      '''Defines a simple two layer network that can be
      trained to recognize the MNIST dataset. There are
      two fully connected layers which feed into a Softmax
      Cross-Entropy loss function. Finally, depending on
      external specification, if the network is training
      we backprop all gradients
      '''
      images =>
      FC() =>
      FC() =>
      fc_results, labels =>
      SoftMaxCrossEntropyLoss() => loss =>
      Print("loss: ")

      loss =>
      if Train:
            SGD() =>
      output
      return output

def runMNIST(images, labels):
      '''Runner method that runs training and test
      images through network
      '''
      80 => epochs
      MNIST => trainedMNIST
      for (i = epochs; i > 0; i--):
            trainedMNIST, images[:1000], labels[:1000] =>
            nn.Train() => trainedMNIST
```

```
      for (i = 5; i > 0; i--)
            trainedMNIST, images[1000:], labels[1000:] =>
            nn.Run()
```

The MNIST example demonstrates the ease of passing the outputs from one layer to the next layer. Since the fundamental building blocks of the Inception language are layers and arrays (matrices), each arrow (=>) indicates passing the array output from the previous layer to the next layer.

The runMNIST method demonstrates how training and running each network will work. The weights are learned via the nn.Train() layer which takes as input a network definition and the training data (images and their corresponding labels). The trained network is then stored in a user-defined trainedMNIST variable. When this variable is passed into nn.Run() along with images and labels, the network will return results from test images.

Some common layers:
FC() : Fully-Connected layer
This layer takes an n-dimensional array of inputs and returns a k-dimensional output array calculated by the learned weights in the FC layer.

Conv(): Convolutional layer
This layer takes an n-by-k dimensional array of inputs and returns an m-by-j dimensional output dependant on the kernel size and stride.

```
def GCD(x, y):
      '''Example GCD program using the Euclidean algorithm
      to recursively find the GCD between two integers
      '''
      x, y =>
      Mod() => x
      if x == 0:
            return y
      y, x =>
      GCD()

[4, 6, 7], [8, 10 14] =>
GCD()

>>> [4, 2, 7]
```

This example GCD script demonstrates how to use layers outside of the context of neural networks. The layer/function `GCD()` takes in two inputs `x` and `y` which are both passed into the

`Mod()` layer. This layer takes a pairwise modulo of the elements and stores the resulting array into `x`, clobbering the old value. The algorithm checks if `x == 0` in which case we return the GCD. Otherwise, the inputs `y` and the new `x` are then passed again into the `GCD()` recursively. We hardcode the inputs to demonstrate how to call the GCD function/layer.

END OF PROJECT PROPOSAL

List of Layers

Operators

- Add
- Sub
- Mult
- Div
- Mod

Control

- Loop
- Conditional

Models

Layers

- FC
- Conv
- LSTM

Losses

- SoftmaxCrossEnt
- SigmoidCrossEnt

Optimizers

- Adam
- SGD

Misc

- StopGradients