Source: https://thinc.ai/docs/concept#annotations:rRYs7HvtEeqjKcMi29YdQw

Concept and Design

Model Composition Problem

The central problem for a neural network implementation is this: during the forward pass, you compute results that will later be useful during the backward pass. How do you keep track of this arbitrary state, while making sure that layers can be cleanly composed?

Example: Uncomposable Model:

The most obvious idea is that we have some thing called a model, and this thing holds some parameters ("weights") and has a method to predict from some inputs to some outputs using the current weights. So far so good. But we also need a way to update the weights. The most obvious API for this is to add an update method, which will take a batch of inputs and a batch of correct labels, and compute the weight update.

```
class UncomposableModel:
    def __init__(self, W):
        self.W = W

    def predict(self, inputs):
        return inputs @ self.W.T

    def update(self, inputs, targets, learningRate=0.001):
        guesses = self.predict(inputs)
        dGuesses = (guesses - targets) / targets.shape[0]  # gradient of loss w.r.t. output

        # The @ is newish Python syntax for matrix multiplication
        dInputs = dGuesses @ self.W

        dW = dGuesses.T @ inputs  # gradient of parameters
        self.W -= learningRate * dW  # update weights

        return dInputs
```

Problem: Cannot Backprop Through Multiple Layers

The `update()` method only works as the outer-level API. You wouldn't be able to put another layer with the same API after this one and backpropagate through both of them. Let's look at the steps for backpropagating through two matrix multiplications:

```
def backpropTwoLayers(W1, W2, inputs, targets):
    hiddens = inputs @ W1.T
    guesses = hiddens @ W2.T

    dGuesses = (guesses-targets) / targets.shape[0]  # gradient of loss w.r.t. output
    dW2 = dGuesses @ hiddens.T
    dHiddens = dGuesses @ W2
    dW1 = dHiddens @ inputs.T
    dInputs = dHiddens @ W1

    return dW1, dW2, dInputs
```

To update the first layer, we must know the **gradient with respect to its output**, but that is only revealed after the full forward pass, gradient of loss, and backpropagation through the second layer. Hence, the `UncomposableModel` is uncomposable: the `update` method expects the input and target to both be available. This only works for the outermost API, but not for intermediate layers. We would need another API for intermediate layers.

Solution: Reverse-Model Auto-Differentiation

Solution is to base the API around the *predict* method, which doesn't have the same composition problem, since there is no problem with writing *model3.predict(model2.predict(model1.predict(X)))*, or *model3.predict(model2.predict(X) + model1.predic*

Key Idea of Thinc: To fix the API problem directly to enable model composition, both forwards and backwards.

Key Design (1): No (explicit) Computational Graph - Just Higher Order Functions

```
from thinc.types import *
from typing import *

def reduceSumLayer(X: Floats3d) -> Tuple[Floats2d, Callable[[Floats2d], Floats3d]]:
    Y: Floats2d = X.sum(axis = 1)

    # Backward pass runs from gradient-of-output (dY) to gradient-of-input (dX)
    # This means we will always have two matching pairs:
    # ---> (inputToForward, outputOfBackprop) == (X, dX), and
    # ---> (outputOfForward, inputOfBackprop) == (Y, dY) TODO ??
    def backpropReduceSum(dY: Floats2d) -> Floats3d:
        (dyFirstDim, dySecDim) = dY.shape
        dX: Floats3d = np.zeros(X.shape) # TODO thinc uses just `zeros` function -- from where??
        dX += dY.reshape((dyFirstDim, 1, dySecDim))

        return dX # d_inputs

    # outputs, backpropFunc
    return Y, backpropReduceSum


def reluLayer(inputs: Floats2d) -> Tuple[Floats2d, Callable[[Floats2d], Floats2d]]:
    mask: Floats2d = inputs >= 0
    outputs: Floats2d = inputs * mask

    def backpropRelu(dOutputs: Floats2d) -> Floats2d:
        return dOutputs * mask # == dInputs

    return outputs, backpropRelu
```

Example: Chain Combinator (using callbacks)

The most basic we we will want to combine layers is in a feed-forward relationship. Calling this combinator *chain()*, after the calculus chain rule:

```
def chain(firstLayer, secondLayer):
    def forwardChain(X):
        Y, getdX = firstLayer(X)
        Z, getdY = secondLayer(Y)

        def backpropChain(dZ):
            dY = getdY(dZ)
            dX = getdX(dY)

            return dX

        return Z, backpropChain

    return forwardChain
```

We can use the *chain()* combinator to build a function that runs our *reduceSUmLayer* and *reluLayer* layers in succession:

```
# from thinc.api import glorot_uniform_init
import numpy as np

chainedForward = chain(firstLayer = reduceSumLayer, secondLayer = reluLayer)

B, S, W = 2, 10, 6 # (batch size, sequence length, width)

# TODO don't know which method thinc uses here: 'uniform' ???? Looked everywhere in thinc.api and thinc.backends
X = np.random.uniform(low = 0, high = 1, size = (B, S, W))
dZ = np.random.uniform(low = 0, high = 1, size = (B, W))

# Returns Z, backpropChain
Z, getdX = chainedForward(X = X)
# The backprop chain in action:
dX = getdX(dZ = dZ)

assert dX.shape == X.shape
```

Example: Chain Combinator (No Callbacks)

Our *chain* combinator works because our **layers return callbacks**, ensuring no distinction in API between the outermost layer and a layer that is part of a larger network. Imagine the alternative, where the function expects the gradient with respect to the output along its input:

```
def reduceSum_noCallback(X: Floats3d, dY: Floats2d) -> Tuple[Floats2d, Floats3d]:
    Y: Floats2d = X.sum(axis = 1)

    # This was in the backprop method of reduceSumLayer():
    (dyFirstDim, dySecDim) = dY.shape

    dX: Floats3d = np.zeros(X.shape) # TODO thinc uses just `zeros` function -- from where??
    dX += dY.reshape((dyFirstDim, 1, dySecDim))

    return Y, dX


def relu_noCallback(inputs: Floats2d, dOutputs: Floats2d) -> Tuple[Floats2d, Floats2d]:
    mask: Floats2d = inputs >= 0
    outputs: Floats2d = inputs * mask

    # NOTE: this was in the backprop of the relu() method
    dInputs: Floats2d = dOutputs * mask
    #def backpropRelu(dOutputs: Floats2d) -> Floats2d:
    #    return dOutputs * mask
    #return inputs * mask, backpropRelu
    return outputs, dInputs


# How do we call `firstLayer`?
# We can't, because its signature expects dY  as part of its input — but we don't know dY yet!
# We can only  compute dY once we have Y. That's why layers must return callbacks.
def chain_noCallback(firstLayer, secondLayer):

    def forwardChain_noCallback(X, dZ):

        # NO CALLBACK:
        # Y, dX = firstLayer(X = X, dY = ???) # this is the stumbling block
```

```
        # WITH CALLBACK:  the callback way doesn't require firstLayer to take dY as its argument:
        # Y, getdX = firstLayer(X)

        raise NotImplementedError()
```

Key Design (2): Encapsulation, Modularity

The problem with no callbacks is more than just functional: the extra parameters passed in the functions in the above No Callback case are not just another kind of input variable to the network. The parameters are not part of the neural network design. We can't just say that parameters (like dY in the *reduceSum_noCallback*) are part of the network because that is not how we want to use the network. We want the parameters of a layer to be an internal detail - **we don't want to have to pass in the parameters on each input**.

Parameters must be handled differently from input variables (of a network) because we want to specify them at different times. We'd like to specify the parameters once *when we create the function* and then have them be an internal detail that doesn't affect the function's signature.

The most direct approach is to introduce another layer of closures, and make the parameters and their gradients arguments to the outer layer. The gradients can then be incremented during the backward pass:

```
# TODO ERROR this code piece has errors: says dW, db are referenced before assignment!!!


def Linear(W, b, dW, db):

    def forwardLinear(X):  # X = inputs

        Y = X @ W.T + b  # Y = outputs

        def backwardLinear(dY):  # dY = d_outputs
            dW = np.zeros(shape = W.shape)
            db = np.zeros(shape = b.shape)

            dW += dY.T @ X
            db += dY.sum(axis = 0)
            #print(dW, db)
            dX = dY @ W

            return dX    # dX = d_inputs

        return Y, backwardLinear

    return forwardLinear


(numBatches, nIn, nOut) = 128, 16, 32

# Initializing the inputs to neural network
W = np.random.uniform(low = 0, high = 1, size = (nOut, nIn)) # matrix
b = np.random.uniform(low = 0, high = 1, size = (nOut, )) # vector

# Initializing the derivatives
# TODO ERROR: initializing the class, just to compile because otherwise RunTime error "dW, db are referenced befo
dW = np.zeros(shape = W.shape)
db = np.zeros(shape = b.shape)

X = np.random.uniform(low = 0, high = 1, size = (numBatches, nIn))
YTrue = np.random.uniform(low = 0, high = 1, size = (numBatches, nOut))
```

```
linear = Linear(W = W, b = b, dW = dW, db = db)
YOut, getdX = linear(X = X) # forward linear

dY = (YOut - YTrue) / numBatches # YTrue.shape[0]

dX = getdX(dY = dY) # backward linear

# Now we can do an optimization step like:
W -= 0.001 * dW
b -= 0.001 * db
#dW.fill(0.0)
#db.fill(0.0)
dW
```

```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

Use a `Model` class to **keep track of parameters, gradients, dimensions** since handling parameters and their gradients explicitly quickly gets unwieldy.

Two possible approaches:

1. **Inheritance Approach:** introduce one class per layer type, with the forward pass implemented as a method on the class (like PyTorch)
2. **Composition Approach:**
    - Each layer constructs a `Model` instance, and passes its `forward` function to this `Model` instance upon construction (example is in the `thinc.layers.linear`). The `Model` object lets you pass in an `init` function to support **shape inference**.
    - In the `forward` method, the `Model` instance is passed in as a parameter, giving you access to the dimensions, parameters, gradients, attributes, and layers. THe second argument of `forward` is the input data and the third argument is a boolean

that lets layers run differently during training and prediction (customary feature).

Want to be able to define complex neural networks passing **only genuine configuration** - shouldn't have to pass in a lot of variables whose values are dictated by the rest of the network.

In the `Linear` example, there are many ways for the inputs to `Linear` to be invalid: the `W` and `dW` variables could be different shapes, size of `b` could fail to match first dimension of `W`, the second dimension of `W` could fail to match the second dimension of the input, etc. With separate inputs like these there is no way we can expect functions to validate their inputs reliably, leading to upredictable logic errors that making debugging hard.

In a network with two `Linear` layers, only one dimension is an actual hyperparameter. The input size to the first layer and output size of the second layer are both **determined by the shape of the data.** Thus the only free variable is number of hidden units (this determines output size of the first layer and input size of second layer). Goal to have missing dimensions **inferred layer** based on input and output data.

Example: Initialization logic:

To make this work, we need to specify the **initialization logic** for each layer in the network. For example, the initialization logic for the `Linear` and `chain` layers is:

```
from typing import Optional

from thinc.api import Model, glorot_uniform_init
from thinc.types import Floats2d
from thinc.util import get_width


def initLogic(model: Model,
              X: Optional[Floats2d] = None,
              Y: Optional[Floats2d] = None) -> None:

    if X is not None:
        model.set_dim(name = "nI", value = get_width(X = X))

    if Y is not None:
        model.set_dim(name = "nO", value = get_width(Y))


    W: Floats2d = model.ops.alloc2f(d0 = model.get_dim(name = "nO"),
                                    d1 = model.get_dim(name = "nI"))

    b: Floats1d = model.ops.alloc1f(d0 = model.get_dim(name = "nO"))

    glorot_uniform_init(ops = model.ops, shape = W.shape)

    model.set_param(name = "W", value = W)
    model.set_param(name = "b", value = b)
```