Source: https://github.com/explosion/thinc/blob/master/examples/01_intro_model_definition_methods.ipynb

Intro to Thinc's `Model` class, model definition, and methods

Thinc uses a functional-programming approach to model definition, effective for: * complicated network architectures and, * use cases where different data types need to be passed through the network to reach specific subcomponents.

This notebook shows how to compose Thinc models and use the *`Model`* class and its methods.

Principles: * Thinc provides layers (functions to create *`Model`* instances) * Thinc tries to avoid inheritance, preferring function composition.

```
import numpy
from thinc.api import Linear, zero_init

nI: int = 16
nO: int = 10
NUM_HIDDEN: int = 128


nIn = numpy.zeros((NUM_HIDDEN, nI), dtype="f")
nOut = numpy.zeros((NUM_HIDDEN, nO), dtype="f")

nIn.shape
nOut.shape
```

```
(128, 10)
```

```
model = Linear(nI = nI, nO = nO, init_W = zero_init)
model

model.get_dim("nI")
model.get_dim("nO")

print(f"Initialized model with input dimension nI={nI} and output dimension nO={nO}.")
```

```
Initialized model with input dimension nI=16 and output dimension nO=10.
```

Key Point: Models support dimension inference from data. You can defer some or all of the dimensions.

```
modelDeferDims = Linear(init_W = zero_init)
modelDeferDims
print(f"Initialized model with no input/ouput dimensions.")
```

```
Initialized model with no input/ouput dimensions.
```

```
X = numpy.zeros((NUM_HIDDEN, nI), dtype="f")
Y = numpy.zeros((NUM_HIDDEN, nO), dtype="f")

# Here is where the dimension inference happens: during initialization of the model
modelDeferDims.initialize(X = X, Y = Y)
modelDeferDims

# We can see that dimension inference has occurred:
modelDeferDims.get_dim("nI")
modelDeferDims.get_dim("nO")

print(f"Initialized model with input dimension nI={nI} and output dimension nO={nO}.")
```

```
Initialized model with input dimension nI=16 and output dimension nO=10.
```

Combinators

There are functions like *chain* and *concatenate* which are called *combinators*. *Combinators* take one or more models as arguments, and return another model instance, without introducing any new weight parameters.

Combinators are layers that express higher-order functions: they take one or more layers as arguments and express some relationship or perform some additional logic around the child layers.

chain()

Purpose of chain: The *chain* function wires two models together with a feed-forward relationship. Composes two models *f* and *g* such that they become layers of a single feed-forward model that computes *g(f(x))*.

Also supports chaining more than 2 layers. * NOTE: dimension inference is useful here.

```
from thinc.api import chain, glorot_uniform_init

NUM_HIDDEN: int = 128
X = numpy.zeros((NUM_HIDDEN, nI), dtype="f")
Y = numpy.zeros((NUM_HIDDEN, nO), dtype="f")

# Linear layer multiplies inputs by a weights matrix and adds a bias vector
# layer 1: Linear layer wih only the output dimension provided
# layer 2: Linear layer with all dimensions deferred
modelChained = chain(layer1 = Linear(nO = NUM_HIDDEN, init_W = glorot_uniform_init),
                     layer2 = Linear(init_W = zero_init), )
modelChained
```

```
<thinc.model.Model at 0x7efefad64730>
```

```
# Initializing model
modelChained.initialize(X = X, Y = Y)
modelChained
```

```
modelChained.layers
```

```
[<thinc.model.Model at 0x7effb03f98c8>, <thinc.model.Model at 0x7effb03f96a8>]
```

```
nI: int = modelChained.get_dim("nI")
nI
nO: int = modelChained.get_dim("nO")
nO

nO_hidden = modelChained.layers[0].get_dim("nO")
nO_hidden
```

```
print(f"Initialized model with input dimension nI={nI} and output dimension nO={nO}.")
print(f"The size of the hidden layer is {nO_hidden}.")
```

```
Initialized model with input dimension nI=16 and output dimension nO=10.
The size of the hidden layer is 128.
```

concatenate()

Purpose of concatenate(): the *concatenate* combinator function produces a layer that *runs the child layer separately* and then *concatenates their outputs together*. Useful for combining features from different sources. (Thinc uses this to build spacy's embedding layers). Composes two or more models *f*, *g*, etc, such that their outputs are concatenated, i.e. *concatenate(f, g)(x)* computes *hstack(f(x), g(x))*. * NOTE: functional approach here

```
from thinc.api import concatenate

modelConcat = concatenate(Linear(nO = NUM_HIDDEN), Linear(nO = NUM_HIDDEN))
```

```
modelConcat
modelConcat.layers

# Initializing model, and this is where dimension inference occurs (for nI)
modelConcat.initialize(X = X)

# Can see that dimension nI was inferred
nI: int = modelConcat.get_dim("nI")
nI

# Can see that dimension nO is now twice the NUM_HIDDEN which we passed in: 256 = 128 + 128 since concatenation o
nO: int = modelConcat.get_dim("nO")
nO
print(f"Initialized model with input dimension nI={nI} and output dimension nO={nO}.")
```

```
Initialized model with input dimension nI=16 and output dimension nO=256.
```

clone()

Some combinators work on a layer and a numeric argument. The *clone* combinator creates a number of copies of a layer and chains them together into a deep feed-forward network.

Purpose of clone: Construct *n* copies of a layer with distinct weights. For example, *clone(f, 3)(x)* computes *f(f'(f''(x)))*

- NOTE: shape inference is useful here: we want the first and last layers to have different shapes so we can avoid giving any dimensions into the layer we clone. Then we just have to specify the first layer's output size and let the res of the dimensions be inferred from the data.

```
from thinc.api import clone

modelClone = clone(orig = Linear(), n = 5)
modelClone
modelClone.layers
```

```
[<thinc.model.Model at 0x7efefad64ea0>,
 <thinc.model.Model at 0x7efefad64f28>,
 <thinc.model.Model at 0x7efefad647b8>,
 <thinc.model.Model at 0x7efefad64598>,
 <thinc.model.Model at 0x7efefad64510>]
```

```
modelClone.layers[0].set_dim("nO", NUM_HIDDEN)
modelClone.layers[0].get_dim("nO")

# Initializing the model here
modelClone.initialize(X = X, Y = Y)

nI: int = model.get_dim("nI")
nI
nO: int = model.get_dim("nO")
nO

# num hidden is still 128
modelClone.layers[0].get_dim("nO")

print(f"Initialized model with input dimension nI={nI} and output dimension nO={nO}.")
```

```
Initialized model with input dimension nI=16 and output dimension nO=10.
```

Can apply *clone* to model instances that have child layers, making it easier to define complex architectures. For instance: usually we want to attach an activation and dropout to a linear layer and then repeat that substructure a number of times.

```
from thinc.api import Relu, Dropout

def hiddenLayer(dropout: float = 0.2):
    return chain(Linear(), Relu(),  Dropout(dropout))

modelCloneHidden = clone(hiddenLayer(), 5)
modelCloneHidden
```

```
<thinc.model.Model at 0x7efefad0a9d8>
```

`with_array()`

Some combinators are unary functions (they take only one model). These are usually **input and output transformations*. For instance: **Purpose of `with_array`:** produce a model that flattens lists of arrays into a single array and then calls the child layer to get the flattened output. Then, it reverses the transformation on the output. (In other words: Transforms sequence of data into a continguous two-dim array on the way into and out of a model.)

```
from thinc.api import with_array, Model

modelWithArray: Model = with_array(layer = Linear(nO = 4, nI = 2))
modelWithArray

Xs = [modelWithArray.ops.alloc2f(d0 = 10, d1 = 2, dtype = "f")]
Xs
Xs[0].shape

modelWithArray.initialize(X = Xs)
modelWithArray

# predict(X: InT) -> OutT: call the model's `forward` function with `is_train = False` and return the output inst
Ys = modelWithArray.predict(X = Xs)
Ys

print(f"Prediction shape: {Ys[0].shape}.")
```

```
Prediction shape: (10, 4).
```

Example of Concise Model Definition with Combinators

Combinators allow you to wire complex models very concisely.

Can take advantage of Thinc's **operator overloading** which lets you use infox notation. Must be careful to use **in a contextmananger** to avoid unexpected results.

Example network:

1. Below, the network expects a list of arrays as input, where each array has two columns with different numeric identifier features.
2. The two arrays are embedded using separate embedding tables
3. The two resulting vectors are added
4. Then passed through the *`Maxout`* layer with layer normalization and dropout.
5. The vectors pass through two pooling functions (*`reduce_max`* and *`reduce_mean`*) and the results are concatenated.
6. The concatenated results are passed through two *`Relu`* layers with dropout and residual connections.
7. The vectors are passed through an output layer, which has a *`Softmax`* activation.

```
from thinc.api import add, chain, concatenate, clone
from thinc.api import with_array, reduce_max, reduce_mean, residual
from thinc.api import Model, Embed, Maxout, Softmax

nH: int = 5 # num hidden layers
```

```
with Model.define_operators({">>": chain, "|":concatenate, "+":add, "**":clone}):
    modelOp: Model = (
        with_array(layer =
                   # Embed: map integers to vectors using fixed-size lookup table.
                   (Embed(nO = 128, column = 0) + Embed(nO = 64, column=1))
        >> Maxout(nO = nH, normalize = True, dropout = 0.2)
    )
    >> (reduce_max() | reduce_mean())
    >> residual(layer = Relu() >> Dropout(rate = 0.2)) ** 2
    >> Softmax()
)

modelOp
modelOp.layers
modelOp.attrs
modelOp.param_names
modelOp.grad_names
modelOp.dim_names
modelOp.ref_names
modelOp.define_operators
modelOp.walk
modelOp.to_dict
```

```
<bound method Model.to_dict of <thinc.model.Model object at 0x7efefad20b70>>
```

Using A Model

Defining the model:

```
from thinc.api import Linear, Adam
import numpy

nI, nO, nH = 10, 10, 128
nI, nO, nH

X = numpy.zeros((nH, nI), dtype="f")
dY = numpy.zeros((nH, nO), dtype="f")

modelBackpropExample: Model = Linear(nO = nO, nI = nI)
```

Initialize the model with a sample of the data:

```
modelBackpropExample.initialize(X=X, Y=dY)
```

```
<thinc.model.Model at 0x7efefad20bf8>
```

Run some data through the model:

```
Y = modelBackpropExample.predict(X = X)
Y
```

```
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.]], dtype=float32)
```

Get a callback to backpropagate:

```
# begin_update(X: InT) -> Tuple[OutT, Callable[[InT], OutT]]
# Purpose: Run the model over a batch of data, returning the output and a callback to complete the backward pass.
```

```
# Return: tuple (Y, finish_update), where Y = batch of output data, and finish_update = callback that takes the g
Y, backprop = modelBackpropExample.begin_update(X = X)
Y, backprop
```

```
(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.]], dtype=float32),
 <function thinc.layers.linear.forward.<locals>.backprop(dY:thinc.types.Floats2d) -> thinc.types.Floats2d>)
```

Run the callback to calculate the gradient with respect to the inputs.

backprop(): * is a callback to calculate gradient with respect to inputs. * only increments the parameter gradients, doesn't actually change the weights. To increment the weights, call *model.finish_update()* and pass an optimizer * If the model has trainable parameters, gradients for the parameters are accumulated internally, as a side-effect.

```
dX = backprop(dY)
dX
```

```
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.]], dtype=float32)
```

Incrementing the weights now by calling *model.finish_update()* and by passing an optimizer.

finish_update(optimizer: Optimizer) -> None * update parameters with current gradients * the optimizer is called with each parameter and gradient of the model

```
adamOptimizer = Adam()

modelBackpropExample.finish_update(optimizer = adamOptimizer)
modelBackpropExample
```

```
<thinc.model.Model at 0x7efefad20bf8>
```

Get and set dimensions, parameters, attributes, by name:

```
modelBackpropExample.get_dim("nO")
# weights matrix
W = modelBackpropExample.get_param("W")
W

modelBackpropExample.attrs["something"] = "here"

modelBackpropExample.attrs.get("foo", "bar")
```

```
'bar'
```

Get parameter gradients and increment them explicitly:

```
dW = modelBackpropExample.get_grad("W")
dW

modelBackpropExample.inc_grad(name = "W", value = 1 + 0.1)
modelBackpropExample.get_grad("W")
```

```
array([[1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1],
       [1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1],
```

```
        [1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1],
        [1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1],
        [1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1],
        [1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1],
        [1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1],
        [1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1],
        [1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1],
        [1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1]], dtype=float32)
```

Can serialize model to bytes and to dist and load them back with `from_bytes` and `from_disk`

```
modelBytes = modelBackpropExample.to_bytes()
modelBytes
```


```
b'\x84\xa5nodes\x91\x84\xa5index\x00\xa4name\xa6linear\xa4dims\x82\xa2nO\n\xa2nI\n\xa4refs\x80\xa5attrs\x91\x81\xa
```