

Source: https://github.com/explosion/thinc/blob/master/examples/00_intro_to_thinc.ipynb

Intro to Thinc: Defining Model and Config and Wrapping PyTorch, TensorFlow and MXNet

Defining Model in Thinc

```
from thinc.api import prefer_gpu
prefer_gpu() # returns boolean indicating if GPU was activated
```

False

Declaring data below for the whole file: Using ml-datasets package in Thinc for some common datasets including MNIST:

```
import ml_datasets

# note: these are numpy arrays
(trainX, trainY), (devX, devY) = ml_datasets.mnist()
print(f"Training size={len(trainX)}, dev size={len(devX)}")
```

Training size=54000, dev size=10000

Step 1: Define the Model

Defining a model with two *Relu-activated hidden layers*, followed by a *softmax-activated output layer*. Also add *dropout* after the two hidden layers to help model generalize better.

The `chain` combinator: acts like `Sequential` in PyTorch or Keras since it combines a list of layers together with a feed-forward relationship.

```
from thinc.api import chain, Relu, Softmax, Model

numHidden = 32
dropout = 0.2

model: Model = chain(Relu(n0=numHidden, dropout=dropout),
                     Relu(n0=numHidden, dropout=dropout), Softmax())

model

<thinc.model.Model at 0x7fdd3038e048>
```

Step 2: Initialize the Model

Call `Model.initialize` after creating the model and pass in a small batch of input data X and small batch of output data Y. Lets Thinc *infer the missing dimensions* (when we defined the model we didn't tell it the input size `nI` or the output size `nO`)

When passing in the data, call `model.ops.asarray` to make sure the data is on the right device (transforms the arrays to `cupy` when running on GPU)

```
from thinc.backends.ops import ArrayXd

# Making sure the data is on the right device
trainX: ArrayXd = model.ops.asarray(trainX)
trainY: ArrayXd = model.ops.asarray(trainY)
devX: ArrayXd = model.ops.asarray(devX)
devY: ArrayXd = model.ops.asarray(devY)

# Initializing model
model.initialize(X=trainX[:5], Y=trainY[:5])

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

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

Initialized model with input dimension `nI` = 784 and output dimension `nO` = 10

Step 3: Train the Model

Create optimizer and make several passes over the data, randomly selecting paired batches of the inputs and labels each time.

**** Key difference between Thinc and other ML libraries:**** other libraries provide a single `.fit()` method to train a model all at once, but Thinc lets you *shuffle and batch your data*.

```
from tqdm.notebook import tqdm

def trainModel(data, model, optimizer, numIter: int, batchSize: int):
    (trainX, trainY), (devX, devY) = data
    # todo why need indices?
```

```

# indices = model.ops.xp.arange(trainX.shape[0], dtype="i")

for i in range(numIter):
    # multibatch(): minimatch one or more sequences of data and yield lists with one batch
    batches = model.ops.multibatch(batchSize, trainX, trainY, shuffle=True)

    for X, Y in tqdm(batches, leave=False):
        # begin_update(self, X: InT) -> Tuple[OutT, Callable[[InT], OutT]]:
        # Purpose: run the model over a batch of data, returning the output and a callable
        # pass.
        # Returned: tuple (Y, finishedUpdated), where Y = batch of output data, and finishedUpdated
        Yh, backprop = model.begin_update(X=X)

        backprop(Yh - Y)

        # finish_update(): update parameters with current gradients. The optimizer is called
        model.finish_update(optimizer=optimizer)

    # Evaluate and print progress
    numCorrect: int = 0
    totalCount: int = 0

    for X, Y in model.ops.multibatch(batchSize, devX, devY):
        # predict(X: InT) -> OutT: calls the model's forward function with is_train=False
        Yh = model.predict(X=X)
        numCorrect += (Yh.argmax(axis=1) == Y.argmax(axis=1)).sum()
        # todo?
        totalCount += Yh.shape[0]

    score = numCorrect / totalCount

    print(f" {i}: {float(score):.3f}")

from thinc.api import Adam, fix_random_seed

fix_random_seed(0)
adamOptimizer = Adam(0.001)
BATCH_SIZE: int = 128
NUM_ITERATIONS: int = 10
print("Measuring performance across iterations: ")

trainModel(data=((trainX, trainY), (devX, devY)),
            model=model,
            optimizer=adamOptimizer,
            numIter=NUM_ITERATIONS,
            batchSize=BATCH_SIZE)

```

Measuring performance across iterations:

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

0: 0.844

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

1: 0.882

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

2: 0.891

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

3: 0.904

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

4: 0.909

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

5: 0.914

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

6: 0.916

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

7: 0.923

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

8: 0.923

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

9: 0.926

Another Way to Define Model: Operator Overloading

- Thinc lets you *overload operators* and bind arbitrary functions to operators like `+`, `*`, and `>>` or `@`.
- The `Model.define_operators` contextmanager takes a dictionary of operators mapped to functions (typically combinators like `chain`)
- Operators in the dict are only valid for the `with` block

```
# Example of using the operators:
from thinc.api import Model, chain, Relu, Softmax

numHidden: int = 32
dropout: float = 0.2

with Model.define_operators({">>": chain}):
    modelByMyOp = Relu(n0=numHidden, dropout=dropout) >> Relu(
        n0=numHidden, dropout=dropout) >> Softmax()
```

NOTE: bunch of things here in source tutorial about config files ...

Wrapping TensorFlow, PyTorch, and MXNet models

Can wrap the underlying model using Thinc interface to get type hints and use config system.

1. Wrapping TensorFlow Models

Tensorflow's `Sequential` layer is equivalent to Thinc's `chain`. Defining here model with two Relu and dropout and softmax output.

```
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.models import Sequential
from thinc.api import TensorFlowWrapper, Adam

width: int = 32
n0: int = 10
nI: int = 784
dropout: float = 0.2

tfModel: Sequential = Sequential()
tfModel.add(Dense(width, activation="relu", input_shape=(nI, )))
tfModel.add(Dropout(dropout))
tfModel.add(Dense(width, activation="relu", input_shape=(nI, )))
tfModel.add(Dropout(dropout))
tfModel.add(Dense(n0, activation="softmax"))
tfModel

<tensorflow.python.keras.engine.sequential.Sequential at 0x7fdd29e2eeb8>
```

The wrapped tensorflow model:

```
wrappedTFModel: Model = TensorFlowWrapper(tensorflow_model=tfModel)
wrappedTFModel
```

```
<thinc.model.Model at 0x7fdd29e44840>
```

Training the wrapped tensorflow model:

```
data = ml_datasets.mnist()
#data

from thinc.optimizers import Optimizer

adamOptimizer: Optimizer = Adam(learn_rate=0.001)
adamOptimizer
```

```
<thinc.optimizers.Optimizer at 0x7fdd2a0536d8>
```

```
# Providing batch of input data and batch of output data to do shape inference.  
wrappedTFModel.initialize(X=trainX[:5], Y=trainY[:5])
```

```
<thinc.model.Model at 0x7fdd29e44840>
```

```
# Training the model  
NUM_ITERATIONS = 10  
BATCH_SIZE = 128
```

```
trainModel(data=data,  
            model=wrappedTFModel,  
            optimizer=adamOptimizer,  
            numIter=NUM_ITERATIONS,  
            batchSize=BATCH_SIZE)
```

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

0: 0.915

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

1: 0.927

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

2: 0.933

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

3: 0.939

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

4: 0.945

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

5: 0.946

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

6: 0.947

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

7: 0.949

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

8: 0.950

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

9: 0.951

2. Wrapping PyTorch Models

Thinc's `PyTorchWrapper` wraps the model and turns it into a regular Thinc `Model`.


```

import torch
import torch.nn

# Renaming imports for reading clarity:
#import torch.nn.modules.dropout.Dropout2d as Dropout2d
#import torch.nn.Linear as Linear
import torch.tensor as Tensor
import torch.nn.functional as F

# Thinc imports
from thinc.api import PyTorchWrapper, Adam

width: int = 32
n0: int = 10
nI: int = 784
dropout: float = 0.2

class PyTorchModel(torch.nn.Module):
    def __init__(self, width: int, n0: int, nI: int, dropout: float):
        super(PyTorchModel, self).__init__()

        self.firstDropout: torch.nn.Dropout2d = torch.nn.Dropout2d(dropout)
        self.secondDropout: torch.nn.Dropout2d = torch.nn.Dropout2d(dropout)

        self.firstLinearLayer: torch.nn.Linear = torch.nn.Linear(in_features=nI,
                                                                    out_features=width)

        self.secondLinearLayer: torch.nn.Linear = torch.nn.Linear(in_features=width,
                                                                    out_features=n0)

    def forward(self, x: Tensor) -> Tensor:
        x: Tensor = F.relu(x)
        x: Tensor = self.firstDropout(x)
        x: Tensor = self.firstLinearLayer(x)
        x: Tensor = F.relu(x)
        x: Tensor = self.secondDropout(x)
        x: Tensor = self.secondLinearLayer(x)

        output: Tensor = F.log_softmax(input = x, dim = 1)

        return output

wrappedPyTorchModel: Model = PyTorchWrapper(pytorch_model=
                                             PyTorchModel(width = width,

```

```
n0 = n0,
nI = nI,
dropout=dropout))
```

```
wrappedPyTorchModel
```

```
<thinc.model.Model at 0x7fdd2a6e38c8>
```

Training the wrapped pytorch model:

```
data = ml_datasets.mnist()
adamOptimizer: Optimizer = Adam(learn_rate = 0.001)

wrappedPyTorchModel.initialize(X = trainX[:5], Y = trainY[:5])
wrappedPyTorchModel
```

```
<thinc.model.Model at 0x7fdd2a6e38c8>
```

```
NUM_ITERATIONS = 10
BATCH_SIZE = 128
```

```
trainModel(data=data,
            model=wrappedPyTorchModel,
            optimizer=adamOptimizer,
            numIter=NUM_ITERATIONS,
            batchSize=BATCH_SIZE)
```

```
#
```

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

```
0: 0.913
```

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

```
1: 0.920
```

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

2: 0.925

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

3: 0.925

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

4: 0.931

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

5: 0.931

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

6: 0.933

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

7: 0.936

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

8: 0.938

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

9: 0.939

3. Wrapping MXNet Models

Thinc's `MXNetWrapper` wraps the model and turns it into a regular Thinc `Model`.

MXNet uses a `.softmax()` method instead of a `Softmax` layer so to integrate it with the rest of the components, we must combine it with a `Softmax()` Thinc layer using the `chain` combinator. * NOTE: make sure to `initialize()` the MXNet model and the Thinc model (both).

```
from mxnet.gluon.nn import Dense, Sequential, Dropout
from thinc.api import MXNetWrapper, chain, Softmax

width: int = 32
n0: int = 10
n1: int = 784
dropout: float = 0.2

mxnetModel = Sequential()
mxnetModel.add(Dense(units = width, activation = "relu"))
mxnetModel.add(Dropout(rate = dropout))
mxnetModel.add(Dense(units = width, activation = "relu"))
mxnetModel.add(Dropout(rate = dropout))
mxnetModel.add(Dense(units = n0))

mxnetModel

Sequential(
  (0): Dense(None -> 32, Activation(relu))
  (1): Dropout(p = 0.2, axes=())
  (2): Dense(None -> 32, Activation(relu))
  (3): Dropout(p = 0.2, axes=())
  (4): Dense(None -> 10, linear)
)

mxnetModel.initialize()
mxnetModel
```

```

Sequential(
  (0): Dense(None -> 32, Activation(relu))
  (1): Dropout(p = 0.2, axes=())
  (2): Dense(None -> 32, Activation(relu))
  (3): Dropout(p = 0.2, axes=())
  (4): Dense(None -> 10, linear)
)

wrappedMxnetModel: Model = chain(layer1 = MXNetWrapper(mxnet_model = mxnetModel),
                                layer2 = Softmax())
wrappedMxnetModel

<thinc.model.Model at 0x7fdd2a6ef6a8>

Training the wrapped mxnet model

data = ml_datasets.mnist()
adamOptimizer: Optimizer = Adam(learn_rate = 0.001)

wrappedMxnetModel.initialize(X = trainX[:5], Y = trainY[:5])
wrappedMxnetModel

<thinc.model.Model at 0x7fdd2a6ef6a8>

NUM_ITERATIONS = 10
BATCH_SIZE = 128

trainModel(data=data,
            model=wrappedMxnetModel,
            optimizer=adamOptimizer,
            numIter=NUM_ITERATIONS,
            batchSize=BATCH_SIZE)

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

0: 0.744

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

```

1: 0.877

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

2: 0.909

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

3: 0.925

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

4: 0.932

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

5: 0.937

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

6: 0.941

HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))

7: 0.944

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

8: 0.945

```
HBox(children=(FloatProgress(value=0.0, max=422.0), HTML(value='')))
```

9: 0.950