Source

Basic CNN Part-of-Speech Tagger with Thinc

We implement a basic CNN for pos-tagging (without external dependencies) in Thinc, and train the model on the Universal Dependencies AnCora corpus.

This tutorial shows three different workflows:

1. Composing the model in **code**
2. Composing the model using **only config file**
3. Composing the model in **code and configuring it via config** (recommended approach)

```
from thinc.api import prefer_gpu
from thinc.config import Config

prefer_gpu()
```

```
False
```

Define the helper functions for loading data, and training and evaluating a given model. * NOTE: need to call `model.initialize` with a batch of input and output data to initialize model and infer missing shape dimensions.

```
import ml_datasets
from tqdm.notebook import tqdm
from thinc.api import fix_random_seed, Model
from thinc.optimizers import Optimizer

from thinc.types import Array2d
from typing import Optional, List


fix_random_seed(0)

def trainModel(model: Model, optimizer: Optimizer, numIters: int, batchSize: int):

    (trainX, trainY), (devX, devY) = ml_datasets.ud_ancora_pos_tags()
    # Need to do shape inference:
    model.initialize(X = trainX[:5], Y = trainY[:5])

    for epoch in range(numIters):
        loss: float = 0.0
        # todo: type??
        batches = model.ops.multibatch(batchSize, trainX, trainY, shuffle=True)

        for X, Y in tqdm(batches, leave = False):
            Yh, backprop = model.begin_update(X = X)
            # todo type ??
            dLoss = []

            for i in range(len(Yh)):
                dLoss.append(Yh[i] - Y[i])
                loss += ((Yh[i] - Y[i]) ** 2).sum()

            backprop(dLoss)
            model.finish_update(optimizer = optimizer)

        # todo type?
        score = evaluate(model = model, devX = devX, devY = devY, batchSize = batchSize)
        #print(f"{i}\t{loss:.2f}\t{score:.3f}")
```

```
        print("Epoch: {} | Loss: {} | Score: {}".format(epoch, loss, score))
```

```
# todo types??
def evaluate(model: Model, devX, devY, batchSize: int) -> float:

    numCorrect: float = 0.0
    total: float = 0.0

    for X, Y in model.ops.multibatch(batchSize, devX, devY):
        # todo type of ypred??
        Yh = model.predict(X = X)

        for yh, y in zip(Yh, Y):
            numCorrect += (y.argmax(axis = 1) == yh.argmax(axis=1)).sum()

            # todo: what is the name of the dimension shape[0]?
            total += y.shape[0]

    return float(numCorrect / total)
```

1. Composing the Model in Code

Here's the model definition, using … * `>>` operator for the `chain` combinator. * `strings2arrays` to transform a sequence of strings to a list of arrays * `with_array` transforms sequences (the passed sequences of arrays) into a contiguous two-dimensional array on the way into and out of the model it wraps.

Final model signature: `Model[Sequence[str], Sequence[Array2d]]`

```
from thinc.api import Model, chain, strings2arrays, with_array, HashEmbed, expand_window, Relu, Softmax, Adam, wa
```

```
width: int = 32
vectorWidth: int = 16
numClasses: int = 17
learnRate: float = 0.001
numIters: int = 10
batchSize: int = 128
```

```
with Model.define_operators(operators = {">>": chain}):

    modelFromCode = strings2arrays() >> with_array(

        layer = HashEmbed(nO = width, nV = vectorWidth, column=0)
        >> expand_window(window_size=1)
        >> Relu(nO = width, nI = width * 3)
        >> Relu(nO = width, nI = width)
        >> Softmax(nO = numClasses, nI = width)
    )
```

```
optimizer = Adam(learn_rate = learnRate)
```

```
modelFromCode
```

```
<thinc.model.Model at 0x7fedb7d69598>
```

Training the model now:

```
trainModel(model = modelFromCode,
           optimizer = optimizer,
```

```
            numIters = numIters,
            batchSize = batchSize)
#
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 0 | Loss: 387245.6607032418 | Score: 0.43985546589781516
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 1 | Loss: 291325.42196020484 | Score: 0.540711062849868
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 2 | Loss: 259087.54757650197 | Score: 0.5839776833355176
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 3 | Loss: 230784.68576764315 | Score: 0.6207103139685095
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 4 | Loss: 212752.10526858037 | Score: 0.6424091513302005
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 5 | Loss: 203007.67740350612 | Score: 0.6580794937562017
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 6 | Loss: 196406.51488909405 | Score: 0.668376612435175
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 7 | Loss: 191051.0628584926 | Score: 0.6745923277104825
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 8 | Loss: 186566.58660080412 | Score: 0.681781588751802
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 9 | Loss: 182936.37714191142 | Score: 0.6885402430120008
```

2. Composing the Model via a Config File

Thinc's config system lets describe **arbitrary trees of objects**:

1. The config can include values like hyperparameters or training settings, or references to functions and the values of their arguments.
2. Thinc then creates the config **bottom-up** so you can define one function with its arguments, then pass the return value into another function.

To rebuild the model in the above config file we need to break down its structure:

- `chain` (takes any number of positional arguments)
- `strings2array` (with no arguments)
- `with_array` (one argument **layer**)
  - **layer:** `chain` (any number of positional arguments)
  - `HashEmbed`
  - `Relu`
  - `Relu`
  - `Softmax`

`chain` takes arbitrarily many positional arguments (layers to compose). In the config, positional arguments can be expressed using `*` in the dot notation (For example, `model.layer` could describe a function passed to `model` as the argument `layer`, while `model.*.relu` defines a positional argument passed to `model`. In this case, the argument name `relu` doesn't matter, it just needs to be unique.)

- NOTE: not recommended to "program via config files" because it doesn't solve any problem and makes the model definition just as complicated.
- NOTE: recommend instead the hybrid approach: wrap the model definition in a registered function and configure it via the config.
- NOTE: need to keep function names so can't start using camelcase at the naming ".v1" part because otherwise we get this error when calling `registry.make_from_config(CONFIG)`: Cant't find `'withArray.v1'` in registry `thinc -> layers`. Available names: `CauchySimilarity.v1, Dropout.v1, Embed.v1, FeatureExtractor.v1, HashEmbed.v1`
  - NOTE: can also get `ConfigValidationError` if the names like `learn_rate` are not spelled correctly (to match later function arguments).

```
CONFIG_STR: str = """
[hyper_params]
width = 32
vector_width = 16
learn_rate = 0.001

[training]
n_iter = 10
batch_size = 128

[model]
@layers = "chain.v1"
```

```
[model.*.strings2arrays]
@layers = "strings2arrays.v1"

[model.*.with_array]
@layers = "with_array.v1"

[model.*.with_array.layer]
@layers = "chain.v1"

[model.*.with_array.layer.*.hashembed]
@layers = "HashEmbed.v1"
nO = ${hyper_params:width}
nV = ${hyper_params:vector_width}
column = 0

[model.*.with_array.layer.*.expand_window]
@layers = "expand_window.v1"
window_size = 1

[model.*.with_array.layer.*.relu1]
@layers = "Relu.v1"
nO = ${hyper_params:width}
nI = 96

[model.*.with_array.layer.*.relu2]
@layers = "Relu.v1"
nO = ${hyper_params:width}
nI = ${hyper_params:width}

[model.*.with_array.layer.*.softmax]
@layers = "Softmax.v1"
nO = 17
nI = ${hyper_params:width}

[optimizer]
@optimizers = "Adam.v1"
learn_rate = ${hyper_params:learn_rate}
"""
```

When the config is loaded it is parsed as a dictionary and all references to values from other sections (like `${hyperParams:width}`) are replaced by their defined values. The result is a nested dictionary describing the objects defined in the config.

```
from thinc.api import registry, Config

config: Config = Config().from_str(CONFIG_STR)
config
```

```
{'hyper_params': {'width': 32, 'vector_width': 16, 'learn_rate': 0.001},
 'training': {'n_iter': 10, 'batch_size': 128},
 'model': {'@layers': 'chain.v1',
  '*': {'strings2arrays': {'@layers': 'strings2arrays.v1'},
   'with_array': {'@layers': 'with_array.v1',
    'layer': {'@layers': 'chain.v1',
     '*': {'hashembed': {'@layers': 'HashEmbed.v1',
       'nO': 32,
       'nV': 16,
       'column': 0},
      'expand_window': {'@layers': 'expand_window.v1', 'window_size': 1},
```

```
      'relu1': {'@layers': 'Relu.v1', 'nO': 32, 'nI': 96},
      'relu2': {'@layers': 'Relu.v1', 'nO': 32, 'nI': 32},
      'softmax': {'@layers': 'Softmax.v1', 'nO': 17, 'nI': 32}}}}}},
 'optimizer': {'@optimizers': 'Adam.v1', 'learn_rate': 0.001}}
```

Next, use *registry.make_from_config* to create the objects and call the functions **bottom-up**.

```
CONFIG: Config = registry.make_from_config(config)
CONFIG
```

```
{'hyper_params': {'width': 32, 'vector_width': 16, 'learn_rate': 0.001},
 'training': {'n_iter': 10, 'batch_size': 128},
 'model': <thinc.model.Model at 0x7fedb7d698c8>,
 'optimizer': <thinc.optimizers.Optimizer at 0x7fedb7cbd198>}
```

Training the model, since we have declared the model, optimizer, and training settings:

```
modelFromConfig: Model = CONFIG["model"]
optimizer: Optimizer = CONFIG["optimizer"]
numIters: int = CONFIG["training"]["n_iter"]
batchSize: int = CONFIG["training"]["batch_size"]
```

```
modelFromConfig
```

```
<thinc.model.Model at 0x7fedb7d698c8>
```

```
trainModel(model = modelFromConfig,
           optimizer = optimizer,
           numIters = numIters,
           batchSize = batchSize)
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 0 | Loss: 393883.2075020075 | Score: 0.41403778106453487
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 1 | Loss: 290904.9920806326 | Score: 0.5343642933368281
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 2 | Loss: 262409.2779584527 | Score: 0.5655739239510981
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 3 | Loss: 250073.04503394663 | Score: 0.5863179375807388
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 4 | Loss: 239642.0183173269 | Score: 0.604665530863273
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 5 | Loss: 226512.68565293401 | Score: 0.625390822458952
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 6 | Loss: 213239.2184684947 | Score: 0.6420159886170034
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 7 | Loss: 203582.5307474602 | Score: 0.6564506768015277
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 8 | Loss: 196849.01451857202 | Score: 0.6644262632692416
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 9 | Loss: 191774.66215213854 | Score: 0.6734128395708909
```

3. Composing the Model with Code and Config

Creating the Code:

Can register your own layers and model definitions using the `@thinc.registry` decorator. These can later be referenced in config files → gives flexibility while keeping config and model definitions concise.

- NOTE: The function you register will be filled in by the config – e.g. the value of `width` defined in the config block will be passed in as the argument `width`. If arguments are missing, you'll see a validation error. If you're using **type hints** in the function, the values will be parsed to ensure they always have the right type. If the types are invalid – e.g. if you're passing in a `list` instead of `int` as the value of `width` – you'll see an error. This makes it easier to prevent bugs caused by incorrect values lower down in the network.

```
import thinc
from thinc.api import Model, chain, strings2arrays, with_array, HashEmbed, expand_window, Relu, Softmax, Adam, wa

@thinc.registry.layers("CnnTagger.v1")
def createCnnTagger(width: int, vectorWidth: int, numClasses: int = 17):

    with Model.define_operators({">>": chain}):
        model: Model = strings2arrays() \
                       >> with_array(layer =
```

```
                                HashEmbed(nO = width, nV = vectorWidth, column = 0)
                                >> expand_window(window_size=1)
                                >> Relu(nO = width, nI = width * 3)
                                >> Relu(nO = width, nI = width)
                                >> Softmax(nO = numClasses, nI = width)
                                )

        return model
```

Creating the Config:

The config now must only define one model block with @layers = "CnnTagger.v1" and the function arguments. Can optionally move function arguments to a section like *[hyper_param]* or could hard-code them into the block.

Advantage of separate section: values are **preserved in the parsed config object** (so not just passed into the function) so can always print and view them.

```
CONFIG_STR: str = """
[hyper_params]
width = 32
vector_width = 16
learn_rate = 0.001

[training]
n_iter = 10
batch_size = 128

[model]
@layers = "CnnTagger.v1"
width = ${hyper_params:width}
vectorWidth = ${hyper_params:vector_width}
numClasses = 17

[optimizer]
@optimizers = "Adam.v1"
learn_rate = ${hyper_params:learn_rate}
"""

CONFIG: Config = registry.make_from_config(Config().from_str(CONFIG_STR))
CONFIG
```

```
{'hyper_params': {'width': 32, 'vector_width': 16, 'learn_rate': 0.001},
 'training': {'n_iter': 10, 'batch_size': 128},
 'model': <thinc.model.Model at 0x7fedb3b760d0>,
 'optimizer': <thinc.optimizers.Optimizer at 0x7fedb3c406d8>}
```

Training the model now:

```
modelFromCodeAndConfig: Model = CONFIG["model"]
optimizer: Optimizer = CONFIG["optimizer"]
numIters = CONFIG["training"]["n_iter"]
batchSize = CONFIG["training"]["batch_size"]

modelFromCodeAndConfig
```

```
<thinc.model.Model at 0x7fedb3b760d0>
```

```
trainModel(model = modelFromCodeAndConfig,
           optimizer = optimizer,
           numIters = numIters,
           batchSize = batchSize)
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 0 | Loss: 398764.6167009473 | Score: 0.3471252316851703
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 1 | Loss: 316407.5635571573 | Score: 0.4933817609945144
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 2 | Loss: 278551.48323122226 | Score: 0.5623724561436354
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 3 | Loss: 242246.51710079028 | Score: 0.6112556868178158
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 4 | Loss: 218306.53896981804 | Score: 0.6410798869189148
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 5 | Loss: 206447.67564318783 | Score: 0.6521446089903207
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 6 | Loss: 199336.3151329594 | Score: 0.6641641547937768
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 7 | Loss: 193730.89788747078 | Score: 0.6706607005785109
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 8 | Loss: 189242.50445418147 | Score: 0.6779248497556775
```

```
HBox(children=(FloatProgress(value=0.0, max=112.0), HTML(value='')))
```

```
Epoch: 9 | Loss: 185335.72993982863 | Score: 0.6827364124838522
```