午睡二時四十分

2017-09-17

chainerのupdaterを自作して複雑なネットワークを訓練する

なぜupdaterの自作が必要か

chainerで様々なニューラルネットを試していると、どこかで複数のモデルが組み合つもの、 複数の出力を持つものなど、込み入ったネットワークを訓練したいことがあると思います。

よくあるmnistのサンプルなどでは

```
optimizer = chainer.optimizers.Adam()
optimizer.setup(model)
(略)
updater = training.StandardUpdater(train_iter, optimizer, device=args.gpu)
trainer = training.Trainer(updater, (args.epoch, 'epoch'), out=args.out)
```

という感じで単一のモデルを対象にして訓練ループを進めているのですが、このままでは 複数モデルや複数出力が絡むネットワークの訓練は扱えません。

自分でゴリゴリと更新やらトレーニングやら評価のプロセスを全て自分で書くことも可能は可能ですが、 せっかくならtrainerやoptimizerなどchainerの作った枠組みをどうせなら活かしたいです そこで必要になるのが、それぞれのネットワークの訓練に適した形のupdaterの自作です。

updaterの役割

updaterはあるネットワークと訓練データが与えられた時に、

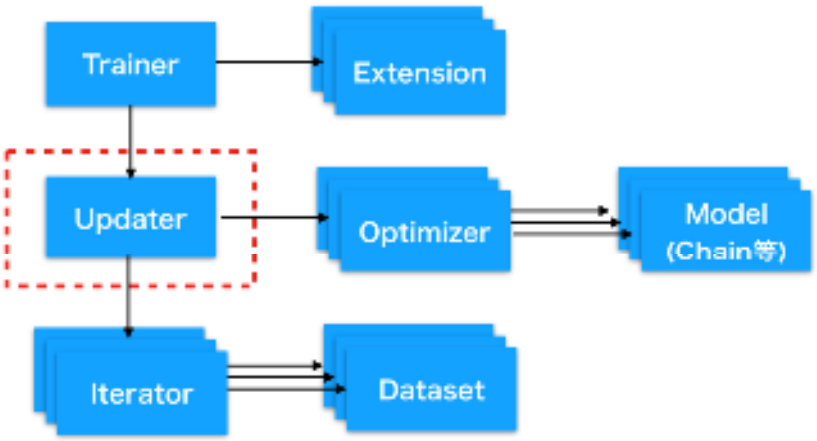
- 訓練データから切り出したミニバッチからの損失の計算方法
- 計算された損失からどのようにネットワークの重みを更新するのか

を定義するモジュールです。

updaterはtrainerに対して一つだけ定義され、一つのupdaterは

- 最適化対象となるモデルを含むoptimizer(1つ以上、複数の場合もあり)
- datasetを含み、epochやバッチサイズを定義したiterator

を受け取ります。(下図)



]

updaterの実装方法

updaterの作成方法は色々あるのかと思いますが、以下では最も基本的な StandardUpdaterを継承/オーバーライドする形でのupdater定義方法を記載します。

StandardUpdaterをオーバーライドして使う場合に、最低限変更が必要になるのが

- `__init__`
- `update_core`

です。

initの実装

`__init__` は言うまでもなくコンストラクタで、受け取る引数を定義します。 必ず受け取らないとエラーになるものはありませんが、その後の `update_core` で重み更新を行うために必要な 以下のような引数は設定したほうがよいと思います。

- 訓練データを含むiterator
- 損失関数の計算に必要なモデル(群)
- 重み変更対象になるモデルをsetupしたoptimizer(群)
- iteratorをミニバッチに変換するためのconverter
- 各種処理を行うためのデバイス指定(cpuなら-1、gpuなら0以上の整数)

また、 `__init__` 内で

- self._iterator (iteratorをまとめたした辞書)
- self._optimizers (optimizerをまとめた辞書)
- self.iteration=0

の3点はtrainerやupdater_core外のupdater処理で参照されているのでこの名前で 宣言しておくのが良いと思います（今回の手法では）。

update_coreの実装

update_coreでは、updaterの中心的な役割となる

1. iteratorから入力データ/ラベルの取り出し
2. 入力データ/ラベルからの損失計算
3. 損失からネットワークの更新

を定義します。

1. iteratorから入力データ/ラベルの取り出し: 後で損失計算できればなんでもよいですが、 chainerのDatasetをセットしたiteratorを渡している場合、next()メソッドを使うと [(入力データ, 教師ラベル), (入力データ, 教師ラベル)..] というlistが取れるため、convert.concat_examplesを使うと ([batch_size分の入力データ] , [batch_size分の教師ラベル]) というtupleに簡単に変換できます。

アクセスの多い記事

できるだけ丁寧にGANとDCGANを理解する

指定したファイルの更新があったらコマンドを自動実行するシェルスクリプト

chainerのupdaterを自作して複雑なネットワークを訓練する

生成モデルpix2pixを動かしてみる

seabornによる統計データ可視化（ポケモン種族値を例に）(1)

2. 入力データ/ラベルからの損失計算: これは目的に応じて。基本的に入力データと教師ラベルをモデルの**call**や然るべきメソッドに渡せば 算出されるように作っていると思います。

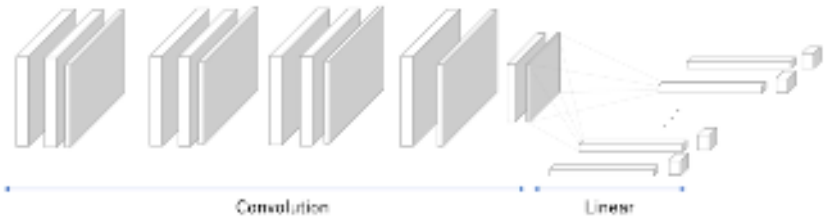
3. 損失からネットワークの更新: Chainerでネットワークの重み更新を行う場合、基本的には ① 各更新対象モデルのcleargrad()による勾配の初期化 ② lossからのbackward() による誤差逆伝搬 ③ optimizer.update()による各モデルの勾配更新 の流れとなります。 （このあたりはChainerのチュートリアルやプレイグラウンド（ここやここ）あたりに目を通すと良いと思います）

以下、update_coreを実装する上で幾つか気をつけるべき点です

- モデルの持つ重みを個別に `cleargrad()` しても良いですが、Chainクラスの場合 `cleargrads()` メソッドにより重みを一度に初期化できます
- `.barckward()` は次元を持たない（つまりスカラーな)Variableに対してしか実行できません
- `backward()` は指定したlossの計算過程をChainインスタンスを跨って伝搬します
- `optimizer.update()` はoptimizerに設定したモデルのみが重み更新対象になります。（逆に重み更新したくないモデルはupdate()しなければ良いです）

実装例

以下、参考になるかわかりませんが私の作ったUpdater例です。 このUpdaterは下図のような畳み込み層に対して複数のLinear層がそれぞれ個別に 入力を受け取り、別々に損失を計算するようなネットワークを訓練するために作ったものです。




```
import six
import numpy as np
import chainer
import chainer.functions as F
import chainer.links as L
from chainer import cuda, training, reporter
from chainer.datasets import get_mnist
from chainer.training import trainer, extensions
from chainer.dataset import convert
from chainer.dataset import iterator as iterator_module
from chainer.datasets import get_mnist
from chainer import optimizer as optimizer_module

class MyUpdater(training.StandardUpdater):
    def __init__(self, iterator, base_cnn, classifiers, base_cnn_optimizer, cl_optimizers, converter=con
        if isinstance(iterator, iterator_module.Iterator):
            iterator = {'main':iterator}
        self._iterators = iterator
        self.base_cnn = base_cnn
        self.classifiers = classifiers

        self._optimizers = {}
        self._optimizers['base_cnn_opt'] = base_cnn_optimizer
        for i in range(0, len(cl_optimizers)):
            self._optimizers[str(i)] = cl_optimizers[i]

        self.converter = convert.concat_examples
        self.device = device
        self.iteration = 0

    def update_core(self):
```

```
        in_arrays = self.converter(iterator, self.device)

        xp = np if int(self.device) == -1 else cuda.cupy
        x_batch = xp.array(in_arrays[0])
        t_batch = xp.array(in_arrays[1])
        y = self.base_cnn(x_batch)

        loss_dic = {}
        for i, classifier in enumerate(self.classifiers):
            loss = classifier(y, t_batch[:,i])
            loss_dic[str(i)] = loss

        for name, optimizer in six.iteritems(self._optimizers):
            optimizer.target.cleargrads()

        for name, loss in six.iteritems(loss_dic):
            loss.backward()

        for name, optimizer in six.iteritems(self._optimizers):
            optimizer.update()
```

少しわかりづらいですが、"base_cnn"が左側の畳み込み層のモデルを表し、 classifiersが右側のLinearモデルのリストになっています。（optimizerも同様）

そしてそれぞれの出力から誤差を逆伝搬し、畳み込み層は全ての出力からの誤差を蓄積した上で updateされるようになっています。

mizti 191日前

関連記事

2018-01-21
スタイル変換系論文サーベイ(1)
画像変換、特にStyle変換の論文を読んでまとめていく。随時追記…

2017-10-25
chainer: トレーニングモジュールの拡張方法まとめ
chainerのチュートリアルをこなしたあと、 mnistなどライブラリ…

2017-10-24
chainer: Evaluatorを自作してトレーニング中のモデルの評価を柔軟に行う
Evaluatorとは DNNの訓練を行う中でモデルの訓練が意図通り進ん…

2017-09-23

chainer: Extensionを自作してディープラーニングの訓練に独自処理を挟み込む

なぜExtensionを自作するのか Chainerのモデルのトレーニング中…

2017-01-13

chainer: 独自datasetを定義する方法

chainerで独自データセットクラスを作るための方法を明示的に示…

« chainer: Extensionを自作してディープラ… スレッドとキューとキューランナーとコー… »