Chainer to introduction 和訳

チェイナーの最初のセッション。ここのセッションでは、次のことについて学びます。

1. 既存のフレームワークの賛否と、なぜチェイナーを開発しているか。
2. forwardとbackwardの計算の簡単な例。
3. つながりの構築
4. パラメータの最適化
5. Linksとoptimizersのシリアライズ。

※シリアライズ(直列化)：オブジェクトを、バイト配列などに変換した上で、状態を維持することを指す。

ファイルとして保存することで、別の場所からでも呼び出せるようになる。

シリアライズ　参考URL：<https://www.sejuku.net/blog/31480>

このセクションを読んだ後、次のことができるようになる。

1. いくつかの算術で、勾配を計算できる。
2. チェイナーで、多層パーセプトロンをかける。

Core Concept

前のページでは、チェイナーは柔軟なニューラルネットワークであるといった。１つの重要なゴールは、柔軟性です。そのため、複雑な構造を「簡単」かつ「直感的」に書くことができなければならない。

既存のほとんどの深層学習は、‘Define-and-Run’の使用をベースにしている。‘Define-and-Run’は、まずネットワークが定義・固定される。そして、その時に、ユーザーがミニバッチ化された周期的に与えるというものである。いくつかのforwardとbackwardの計算の前に、そのネットワークは統計的に定義される。そのため、全てのロジックはデータとしてネットワーク構造に組み込まなければならない。結果、システムのようにネットワーク構築を定義することは、宣言型のアプローチに従う。命令型言語を使って静的なネットワーク定義できるフレームワークが未だにあることも書いておく。

Define-by-Run

チェイナーは、‘Define-and-Run’の枠組みに適応している。そのネットワークは進行中に、実際のforwardの計算を経由して定義される。チェイナーはプログラミングロジックの代わりに計算の履歴を保存する。この戦力は、python上のプログラミングロジックの力を最大限に引き出すことを可能にしました。例えば、チェイナーは状態や、ループをネットワークの定義に取り入れるためのマジックを全く必要としない。‘Define-and-Run’の枠組みはチェイナーの主要な概念である。このチュートリアルでどのように動的ネットワークが定義されていくかを見ていく。

この戦略は、ロジックはネットワーク操作と似ているため、マルチGPUの並列化を簡単に書くことができる。

このチュートリアルの最後の方で、仕組みの好ましさを検討する。

チェイナーは、１つのネットワークを１つの‘computational graph’の実行パスのように表します。１つの‘computational graph’は、アプリケーション関数の１つのシリーズです。そのため、複数の‘function’オブジェクトで述べられます。そのような関数は、一層のネットワークの時には、訓練を通してパラメータはアップデートされます。それゆえ、中で訓練できるパラメータをキープする必要があります。そのため、チェイナーは、クラスのオブジェクト内で学習できるパラメータを保持できる‘Link’クラスを持っています。‘Link’オブジェクト内で行われた関数のパラメータは‘Variable’オブジェクトとして表されます。‘Link’オブジェクトと‘function’オブジェクトの違いを簡単にいうと、訓練できるパラメータを含めるか含めないかです。自然なネットワークモデルは、一般的に‘Link’、‘function’シリーズのように述べられています。

‘chain’オブジェクトを定義するために、‘Link’や‘function’ の様々な種類の動的な’chaining’によって、‘computational graph’を構築できます。フレームワークの中では、ネットワークは、繋がれたグラフの実行によって、定義されます。

そのため、チェイナーと言います。

このチュートリアルのサンプルコードでは、単純化のために、次のsymbolがすでにimportされていると仮定します。

|  |
| --- |
| **import** numpy **as** np  **import** chainer  **from** chainer **import** cuda, Function, gradient\_check, report, training, utils, Variable  **from** chainer **import** datasets, iterators, optimizers, serializers  **from** chainer **import** Link, Chain, ChainList  **import** chainer.functions **as** F  **import** chainer.links **as** L  **from** chainer.training **import** extensions |

これらのimportは、chainerのコードや例に広く使われています。このチュートリアルでは、これらのimportを単純化のために省きます。

Forward/Backward Computation

上記で述べたように、chainerは’Difine-by-Run’の枠組みを使います。そのため、forwardの計算は、自分自身がネットワークを定義となります。forwardの計算を始めるために、インプット配列として、Variableオブジェクトを設定しなければいけません。そのために、要素が１つの簡単なn次元配列から始めます。

|  |
| --- |
| x\_data **=** np**.**array([5], dtype**=**np**.**float32)  ※chainerでは、ほとんどの計算いおいて、32ビットの浮動小数点のみをサポートする。  x **=** Variable(x\_data) |

Variableオブジェクトは、基本的な算術計算をします。 y=x2−2x+1を計算するには、次のように書きます。

|  |
| --- |
| y **=** x**\*\***2 **-** 2 **\*** x **+** 1 |

結果のyもVariableオブジェクトであり、‘data’属性にアクセスすることによって、yの値を得ることができる。

|  |
| --- |
| y**.**data  array([16.], dtype=float32) |

yは、答えだけを保持しているだけではありません。計算履歴も持っており、微分を計算することができます。

‘backward()’ メソッドを呼ぶことによって、微分を計算します。

|  |
| --- |
| y**.**backward() |

これは、誤差逆伝播法を走らせます。その時、その勾配は、計算され、変数xの‘grad属性’の保存されます。

|  |
| --- |
| x**.**grad  array([8.], dtype=float32) |

微分の中間変数も計算することができます。

※chainerでは、初期設定で、メモリー効率のため、勾配配列の中間変数を解放していることに注意してください。

勾配情報を保存するためには、‘backward’メソッドの‘reitain\_grad’引数をpassしてください。（trueを与えてください）

|  |
| --- |
| z **=** 2**\***x  y **=** x**\*\***2 **-** z **+** 1  y**.**backward(retain\_grad**=True**)  z**.**grad  array([-1.], dtype=float32) |

その他の方法は、次のように、z**.**gradをNoneにしてください。

|  |
| --- |
| y**.**backward() *# The default value of retain\_grad is False*  z**.**grad **is** **None** |

これらの計算は、複数要素の配列に一般化することができます。

もし、複数要素配列を持った変数から、‘backward()’の計算をしたいとしたら、初期エラー値を手動で設定しなければいけないことに、注意してください。

なぜなら、１つの変数‘size’が１の時、変数オブジェクトは、誤差変数として考えます。そのため、その変数の‘grad’属性は、自動的に１で満たされます。一方で、変数のsize’が１より大きい時、その変数の‘grad’属性はNoneのままである。‘backward()’を走らせる前に、明示的に初期エラーをセットする必要があります。

これは、出力変数の‘grad’属性を次のように設定することによって簡単にすることができます。

|  |
| --- |
| x **=** Variable(np**.**array([[1, 2, 3], [4, 5, 6]], dtype**=**np**.**float32))  y **=** x**\*\***2 **-** 2**\***x **+** 1  y**.**grad **=** np**.**ones((2, 3), dtype**=**np**.**float32)  y**.**backward()  x**.**grad  array([[ 0., 2., 4.],  [ 6., 8., 10.]], dtype=float32) |

注意

多くの関数は‘Valiable’オブジェクトを取るように、‘function’モジュールで定義されています。

複雑な関数を実現するために、これらを組み合わせて、自動‘backward’を計算することができます。

Links

ニューラルネットワークを書くために、パラメータ関数を結合し、パラメータを最適化しなければならない。

Linksを使うことこれができる。Linkはパラメータを保持したオブジェクトです。

最も基本的なlinksの１つは、通常の関数の振る舞いをすると同時に、パラメータによって、引数を入れ替えます。高次元Linksを紹介します。しかし、ここでは単にパラメータを持った関数として考えてください。

最も頻繁に使われるLinksは‘linear’ linkです。それは、関数を表現します。ここでは、行列とベクトルはパラメータです。３次元空間から、２次元空間への‘linear’ linkは次のように定義されます。

|  |
| --- |
| f **=** L**.**Linear(3, 2) |

注意

多くの関数とLinkはミニバッチ入力のみ受けつけます。入力配列の初めの次元は、バッチの次元（サイズ）として考えられます。上のケースでは、入力の形は、（N，3）でなければならない。Nはミニバッチサイズです。

Linkのパラメータは、属性で保持されます。それぞれのパラメータは‘Variable’のインスタンスです。‘linear’linkの場合、２つのパラメータ、とが保持されます。デフォルトによって、行列はランダムに初期化され、ベクトルはゼロで初期化されます。

|  |
| --- |
| >>>f**.**W**.**data  array([[ 1.0184761 , 0.23103087, 0.5650746 ],  [ 1.2937803 , 1.0782351 , -0.56423163]], dtype=float32)  >>> f**.**b**.**data  array([0., 0.], dtype=float32) |

‘linear’ linkのインスタンスは、通常の関数のように振る舞います。

|  |
| --- |
| >>> x **=** Variable(np**.**array([[1, 2, 3], [4, 5, 6]], dtype**=**np**.**float32))  >>> y **=** f(x)  >>> y**.**data  array([[3.1757617, 1.7575557],  [8.619507 , 7.1809077]], dtype=float32) |

注意

入力空間の次元の計算が時々、厄介です。‘linear’ linkや、いくつかの畳み込みlinkは、インスタンス化における入力次元を省くことができる。かつ、初めのミニバッチから入力次元を推論することができる。

例えば、次のような出力次元が２の‘linear’ linkである。

|  |
| --- |
| f **=** L**.**Linear(2) |

もし、ミニバッチの形が（N，M）で与えた場合、入力次元はMとして推論されます。つまり、f.Wは、2×M行列になる。※それらのパラメータは、初めのミニバッチにおいて怠惰なマナーで初期化されることに注意。

それゆえ、もし、何もデータがないものが、linkに置かれた場合、fはWの属性を持たない。

パラメータの勾配は、‘backward’メソッドによって計算されます。勾配は、上書きよりもむしろ、‘backward’メソッドによって蓄積されます。そのため、勾配の計算を更新するために、ゼロに初期化しなければなりません。これは、‘cleargrads()’メソッドを呼ぶことでできます。

|  |
| --- |
| >>> f**.**cleargrads() |

‘cleargrads()’メソッドは、v1.15では効率化のため、‘zerosgrads()’の代わりに紹介されています。

‘zerosgrads()’はbackwardの互換性のみを残します。

今、‘backward’メソッドを呼ぶことで、パラメータの勾配計算ができます。

|  |
| --- |
| >>> y**.**grad **=** np**.**ones((2, 2), dtype**=**np**.**float32)  >>> y**.**backward()  >>> f**.**W**.**grad  array([[5., 7., 9.],  [5., 7., 9.]], dtype=float32)  >>> f**.**b**.**grad  array([2., 2.], dtype=float32) |

Write a model as a chain

多くのニューラルネットワークの構造は、複雑なlinksを含んでいます。例えば、多層のパーセプトロンは、複雑な線形層を構成しています。次のように、複雑なlinksを結合することで、パラメータを学習させる複雑な手順を書くことができます。

|  |
| --- |
| >>> l1 **=** L**.**Linear(4, 3)  >>> l2 **=** L**.**Linear(3, 2)  >>> **def** **my\_forward**(x):  ... h **=** l1(x)  ... **return** l2(h) |

ここでは、Lは、linksモジュールを示します。この方法でパラメータを定義する手続きは、再利用が難しいです。より、pythonチックな方法は、linksを結合し、classの手続きをすることです。

|  |
| --- |
| >>> **class** **MyProc**(object):  ... **def** **\_\_init\_\_**(self):  ... self**.**l1 **=** L**.**Linear(4, 3)  ... self**.**l2 **=** L**.**Linear(3, 2)  ...  ... **def** **forward**(self, x):  ... h **=** self**.**l1(x)  ... **return** self**.**l2(h) |

よりお手頃に作るために、パラメータ管理と、CPU/GPUの移動、さらに、頑強かつ柔軟性のある保存/読み込みなどの機能を持ったサポートが欲しい。チェイナーでは、chainクラスによってこれらのすべての機能がサポートされています。その時にしなければならないことは、単に上記のクラスをchainのサブクラスと定義するだけです。

|  |
| --- |
| >>> **class** **MyChain**(Chain):  ... **def** **\_\_init\_\_**(self):  ... super(MyChain, self)**.**\_\_init\_\_()  ... **with** self**.**init\_scope():  ... self**.**l1 **=** L**.**Linear(4, 3)  ... self**.**l2 **=** L**.**Linear(3, 2)  ...  ... **def** **\_\_call\_\_**(self, x):  ... h **=** self**.**l1(x)  ... **return** self**.**l2(h) |

どのように、簡単なlinksによって、複雑な繋がりが構成されているかを見てきた。

l1、やl2のようなlinksは、MyChainの子linksと呼ばれるます。Chain自身は、Linkを受け継ぐことに注意。つまり、MyChainが子linksを持つように、より複雑な繋がりを定義することができます。

注意

よく、\_\_call\_\_オペレータを使うことによって、linkの簡単なforwardメソッド定義します。

そのようなlinksや繋がりは、呼び出すことができ、かつ、一般的なValiablesの関数のように振る舞います。

注意

Chainer v1では、学習可能層をchainの\_\_init\_\_()におく、あるいは、add\_link()を経由してそれらを登録することによって、学習可能層をモデルに登録することができます。しかし、Chainer v2ではこの方法は、非推奨です。上記で説明した方法をお勧めします。

他の繋がりを定義する方法は、ChainListクラスを使うことです。それは、linksのリストのように振る舞います。

|  |
| --- |
| >>> **class** **MyChain2**(ChainList):  ... **def** **\_\_init\_\_**(self):  ... super(MyChain2, self)**.**\_\_init\_\_(  ... L**.**Linear(4, 3),  ... L**.**Linear(3, 2),  ... )  ...  ... **def** **\_\_call\_\_**(self, x):  ... h **=** self[0](x)  ... **return** self[1](h) |

|  |
| --- |
|  |

ChainListは任意のlinksの数を使うのに便利です。しかしながら、上記のケースのようにlinksの数が固定されている場合、Chainを親クラスにすることをお勧めします。

Optimizer

良い、パラメータ値を得るために、‘Optimizer’クラスによって、それらのパラメータを最適化しなければいけません。‘Optimizer’は、‘link’に与えられた数値最適化アルゴリズムを走らせます。多くのアルゴリズムは、‘Optimizer’モジュールの中で実装されます。ここでは、‘Stochastic Gradient Descent’と呼ばれる、最もシンプルな‘Optimizer’を使います。

|  |
| --- |
| >>> model **=** MyChain()  >>> optimizer **=** optimizers**.**SGD()  >>> optimizer**.**setup(model) |

‘setup()’モジュールは、与えられた‘link’を最適化するための準備です。

Weightの減衰は(少しずつ減少していくこと)や、勾配を切り取るなどの多くのパラメータや勾配の操作は、‘Optimizer’に‘hook function’をセッティングすることによってできます。

‘hook function’は、‘after the gradient computation’や、‘right before the natural update of parameter’と呼ばれています。

例えば、次のコードを予め走らせることによって、Weight減衰の正則化を設定することができます。

|  |
| --- |
| >>> optimizer**.**add\_hook(chainer**.**optimizer**.**WeightDecay(0.0005)) |

もちろん、‘hook function’を自身で書くこともできます。その時には、‘Optimizer’を引数として受け取れるように、‘hook function’を関数とするまたは、オブジェクトを呼んでこなければいけません。

最適化する方法は、２つあります。１つ目は、‘Trainer’経由して‘Optimizer’を使うことです。これについては、次のセッションで見ます。もう一方の方法は、直接‘Optimizer’を使うことです。ここでは、後者を検討します。

もし、簡単に ‘Optimizer’を使える方法に興味がある場合は、このセッションを飛ばして、つぎのセンションに進んでください。

直接‘Optimizer’を使う方法は2つあります。1つ目は、手動で構内を計算し、その時に‘update()’メソッドを引数なしで呼び出す方法です。予め、勾配をクリアーにしておくことを忘れないようにしてください。

|  |
| --- |
| >>> x **=** np**.**random**.**uniform(**-**1, 1, (2, 4))**.**astype('f')  >>> model**.**cleargrads()  >>> *# compute gradient here...*  >>> loss **=** F**.**sum(model(chainer**.**Variable(x)))  >>> loss**.**backward()  >>> optimizer**.**update() |

もう１つは、誤差関数を‘update()’メソッドに受け渡す方法です。ここでの場合は、‘update()’メソッドによって‘cleargrads()’は自動的に呼ばれます。そのため、ユーザーは手動で‘cleargrads()’で呼ぶ必要はありません。

|  |
| --- |
| >>> **def** **lossfun**(arg1, arg2):  ... *# calculate loss*  ... loss **=** F**.**sum(model(arg1 **-** arg2))  ... **return** loss  >>> arg1 **=** np**.**random**.**uniform(**-**1, 1, (2, 4))**.**astype('f')  >>> arg2 **=** np**.**random**.**uniform(**-**1, 1, (2, 4))**.**astype('f')  >>> optimizer**.**update(lossfun, chainer**.**Variable(arg1), chainer**.**Variable(arg2)) |

完全な使用のために、‘Optimizer.update()’をみて下さい。

<http://docs.chainer.org/en/stable/reference/core/generated/chainer.Optimizer.html#chainer.Optimizer.update>

Trainer

ニューラルネットワークを訓練させたい時、パラメータをアップデートするために、数多くの訓練のループを走らせなければいけません。典型的な訓練のループは、次の手続きを含んでいます。

1. 訓練データセットのイテレーション。
2. 抽出されたミニバッチの前処理。
3. ニューラルネットワークのForward/backward計算。
4. パラメータのアップデート。
5. 分割データセットにおける現在のパラメータの評価。
6. 中間結果をログに残し、表示する。

Chainerは、訓練プロセスを簡単にかけるように、シンプルだが強力な方法を提供しています。その訓練ループは、抽象化すると、主に2つの要素で構成されています。

Dataset abstraction

これは、上記の1，2のリストの実行です。主な構成要素は、‘dataset’モジュールで定義されています。

‘dataset’モジュールと‘iterators’モジュールでは、それぞれ、データセットとイタレータに関する多くの実装があります。

Trainer

これは、上記のリストの3,4,5,6の実行です。全体の手続きは、‘Trainer’によって実行されます。

パラメータのアップデート（上記のリスト3,4）は、‘updater’によって定義され、自由にカスタマイズすることができます。リストの5,6は‘Extension’のインスタンスによって実行されます。これは、訓練ループに追加の手続きとして付加します。ユーザーは、拡張を付加することによって、訓練の手続きを自由にカスタマイズすることができます。ユーザーは、自身の拡張でこれらを実行できます。

下記のセッションの例において、‘Trainer’の使い方をみていきます。

参考URL：<http://docs.chainer.org/en/stable/tutorial/basic.html>

<http://www.iandprogram.net/entry/chainer_japanese>