AI自動作曲プログラミング

第１１回

ディープラーニング
プログラミング

バックプロパゲーション理解のための5つの要素

1．訓練データとテストデータ

2．損失関数

3．勾配降下法

4．最適化アルゴリズム

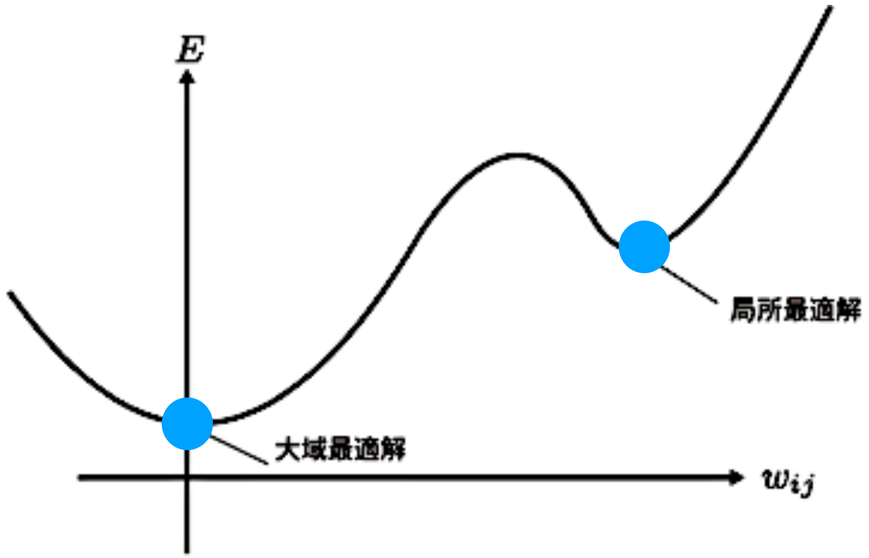
5．バッチサイズ

最適化アルゴリズム

最適化アルゴリズム

最適化とは

プログラムなどがあらかじめ定められた判定基準に照して，最も望ましい値をとるようにすること。
機械学習・ディープラーニングで言えば勾配が最小である状態。
一般に直接決定あるいは操作することのできる変量を変えることによって，最適化を達成する。



最適化アルゴリズムとは

機械学習・ディープラーニングで勾配が最小、つまり大域最適解に早く確実にたどり着くためのアルゴリズム

最適化アルゴリズムの種類

最適化アルゴリズムの種類

- 確率的勾配降下法（SGD)
- Momentum
- AdaGrad
- RMSProp
- Adam

- 確率的勾配降下法（SGD）
- Momentum
- AdaGrad
- RMSProp
- Adam

確率的勾配降下法（SGD）

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

$$b \leftarrow b - \eta \frac{\partial E}{\partial b}$$

確率的勾配降下法（stochastic gradient descent, SGD）
連続最適化問題に対する勾配法の乱択アルゴリズム。
更新毎に訓練データからランダムにサンプルを選び出す。

メリット

・局所最適解に囚われにくい
・更新量の決め方がシンプル（学習率x勾配）
・シンプルなのでコードが簡単、実装が楽

デメリット

・更新量の更新が柔軟には調整できない

- 確率的勾配降下法（SGD）
- Momentum
- AdaGrad
- RMSProp
- Adam

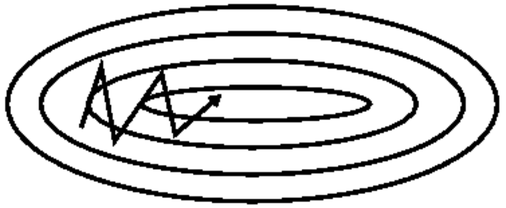
Momentum

$$w \leftarrow w - \eta \frac{\partial E}{\partial w} + \alpha \Delta w$$

$$b \leftarrow b - \eta \frac{\partial E}{\partial b} + \alpha \Delta w$$

Momentumなし

Momentumあり



確率的勾配降下法に慣性を加えたアルゴリズム。
図のように、関連性のある方向へSGDを加速させる方法です。
現在の更新ベクトルに、過去のタイムステップの更新ベクトルを加えることにより実現します。
αは慣性の強さ　△wは前回の（重みの）更新量

メリット

・更新がなめらか（より勾配に従った更新）

デメリット

・定数が増えて調整が難しくなる

- 確率的勾配降下法（SGD）
- Momentum
- AdaGrad
- RMSProp
- Adam

AdaGrad

$$h \leftarrow h + (\frac{\partial E}{\partial w})^2$$

$$w \leftarrow w - \eta \frac{1}{\sqrt{h}} \frac{\partial E}{\partial w}$$

$$h \leftarrow h + (\frac{\partial E}{\partial b})^2$$

$$b \leftarrow b - \eta \frac{1}{\sqrt{h}} \frac{\partial E}{\partial b}$$

更新量が自動で調整され、学習が進むと学習率が次第に小さくなるアルゴリズム
数式の通りhは勾配が大きければ増え、小さければ減る

メリット

・更新の効率が良い

デメリット

・更新量が誤って 0 になると学習が止まる

- 確率的勾配降下法（SGD）
- Momentum
- AdaGrad
- RMSProp
- Adam

RMSProp

$$h \leftarrow \rho h + (1-\rho)(\frac{\partial E}{\partial w})^2$$

$$w \leftarrow w - \eta \frac{1}{\sqrt{h}} \frac{\partial E}{\partial w}$$

AdaGradの更新が 0 で止まってしまう問題を克服したアルゴリズム
Geoffrey HintonがCouseraの講義内で提案しました。
学習率の初期値が0.001の場合、pに0.9を代入することを推奨しています。

メリット

・AdaGradの良さを引き継ぎ更新の効率が良い

デメリット

・実装がやや複雑

ジェフリー・ヒントン

（英: Geoffrey Everest Hinton、1947年12月6日 - ）は、イギリス生まれのコンピュータ科学および認知心理学の研究者。
ニューラルネットワークの研究で有名。
現在は、トロント大学とGoogleで働いている。
ニューラルネットワークのバックプロパゲーション、ボルツマンマシン、オートエンコーダ、ディープ・ビリーフ・ネットワークの開発者の1人であり、オートエンコーダやディープ・ビリーフ・ネットワークは深層学習になった。

特に２００６年のオートエンコーダーはその後の機械学習・ディープラーニングの進化に多大な貢献を果たした。

- 確率的勾配降下法（SGD）
- Momentum
- AdaGrad
- RMSProp
- Adam

Adam

$$m_0 = v_0 = 0$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial E}{\partial w}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\frac{\partial E}{\partial w})^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$w \leftarrow w - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Adam（Adaptive Moment Estimation）
それぞれのパラメータに対し学習率を計算し適応させていきます。
RMSpropでは、過去の勾配の二乗の指数関数的減衰平均を蓄積していました。
Adamではこれに加え、同じように過去の勾配の指数関数的減数平均を保持します。Momentumと似た方法です。

メリット

・高性能で良い結果をもたらすことが多い

デメリット

・複雑で実装が難しい

バックプロパゲーション理解のための5つの要素

1．訓練データとテストデータ

2．損失関数

3．勾配降下法

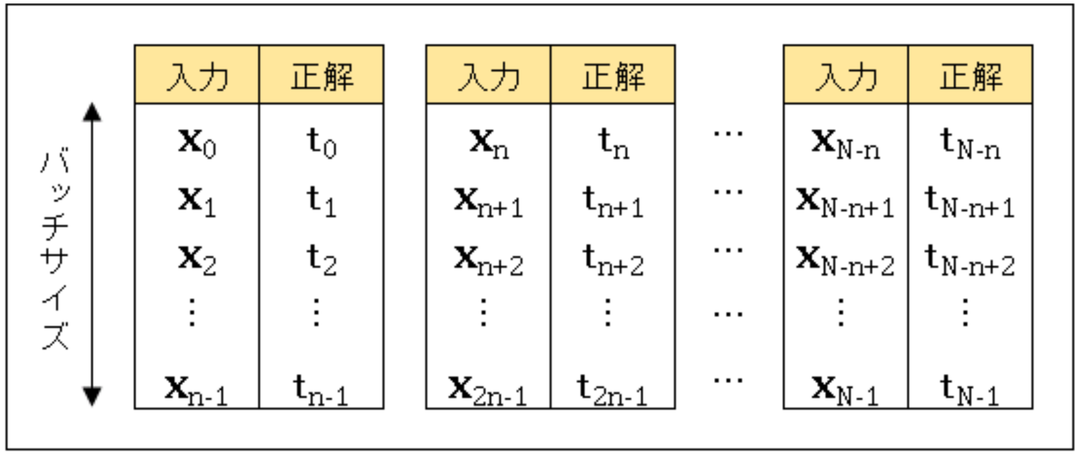
4．最適化アルゴリズム

5．バッチサイズ

バッチサイズ

バッチサイズ

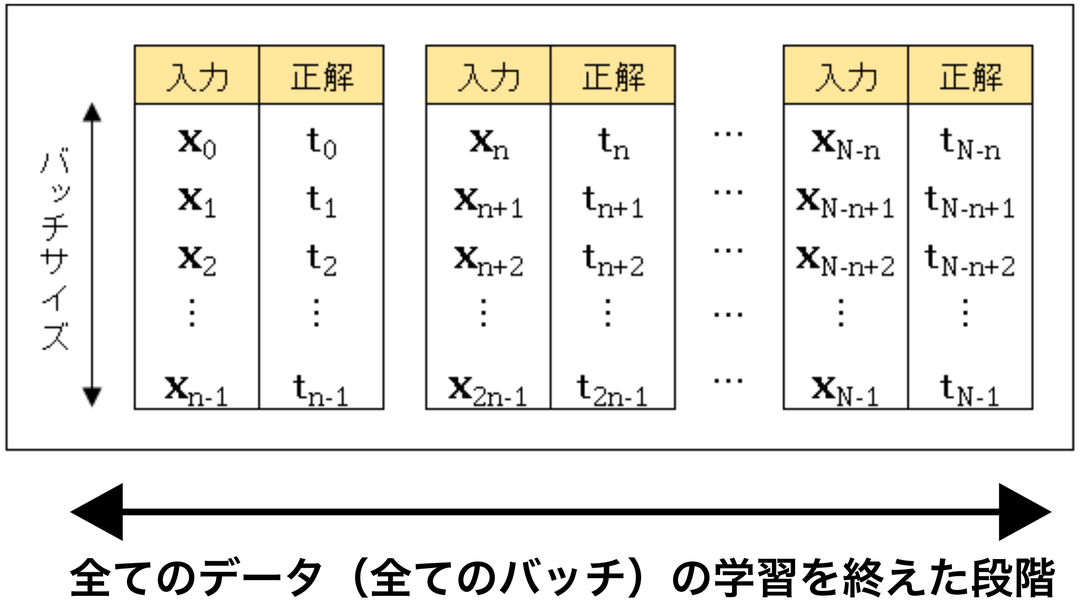
機械学習・ディープラーニングにおいては、通常手持ちのデータをまとめて全部学習に使うようなことはしません。
全データを訓練データとテストデータに分け、さらに訓練データもいくつかのバッチ（かたまり）に分割して、バッチごとに重みの最適化を実行します。
言い換えれば、バッチとは重みの更新間隔です。
1 バッチに含まれるデータ数を バッチサイズ とよびます。
一般的にはどのバッチもサイズが等しくなるように訓練データを均等分割します。



Epoch

Epoch

すべての訓練データについて学習し終えた段階を 1 エポック (epoch) といいます。たとえば、5000 個の訓練データをサイズ 100 のバッチに分割した場合、バッチ数は 50 個となるので、50 バッチの学習を終えたときに 1 エポックとなります。



バッチ学習

バッチ学習

1 バッチの平均誤差

$$E = \frac{1}{N}\sum_{i=1}^{N} E_i$$

誤差関数の重みに対する勾配

$$\frac{\partial E}{\partial w} = \sum_{i=1}^{N} \frac{\partial E_i}{\partial w}$$

バッチサイズがデータの総数と等しい場合、の学習をバッチ学習とよびます。
つまり、全訓練データの数とバッチサイズに等しくなるので、重みの更新間隔は 1 エポックとなります。
手持ちのデータをすべて使うので、個々のデータが学習に与える影響は相対的に小さくなり、極端に間違った方向に学習を進めることがありません。
ただし局所最適解に陥る可能性があります。

新しいデータが追加されると、すべてのデータを使って学習をやり直さなくてはならないので、古いデータも保管しておく必要があり、十分なメモリを確保できる環境が要求されます。

オンライン学習

オンライン学習

バッチサイズを 1 にした場合はオンライン学習とよばれます。
オンライン学習では 1 回の学習で 1 個のデータのみを使用します。
学習に使ったデータを捨てることができるので、
「ビッグデータを使って学習させたいけれど、すべてのデータをメモリに保管できない」
というような状況で有効な学習法です。
また、学習中にも逐次追加データが入ってくるような状況（天気予報やスパム分類など）にも、オンライン学習が適しています。

ミニバッチ学習

ミニバッチ学習

１ バッチの平均誤差

$$E = \frac{1}{N}\sum_{i=1}^{N} E_i$$

誤差関数の重みに対する勾配

$$\frac{\partial E}{\partial w} = \sum_{i=1}^{N} \frac{\partial E_i}{\partial w}$$

バッチ学習とオンライン学習の中間です。
全データを訓練データとテストデータに分け、さらに訓練データもいくつかのバッチに分割して、バッチごとに重みとバイアスの最適化を実行します。
バッチとは重みとバイアスの更新間隔です。
例えばデータ総数１０００
バッチサイズ５０とすれば
１Epoch辺り２０回の更新が行われます。

ミニバッチ学習は
バッチ学習に比べ局所最適解に囚われにくく
オンライン学習の様におかしな方向に学習が進む事がありません。

Magenta

Performance RNN

Performance RNN

Performance RNN はポリフォニック演奏を可能にした音楽生成モデルです。
Dynamics（ベロシティによる抑揚）やタイミングの微妙な変化までを再現した音楽生成ができます。

他のモデルと違い以下の様な特徴を持ちます

- ノートオンイベント(*pitch*): start a note at *pitch*
- ノートオフイベント(*pitch*): stop a note at *pitch*
- タイムシフト(*amount*): advance time by *amount*
- ベロシティ: change current velocity to *value*

Performance RNN

- ノートオンイベント(*pitch*): start a note at *pitch*
- ノートオフイベント(*pitch*): stop a note at *pitch*

通常は発音イベントが発音時間とともに生成されますが、ノートオンとノートオフのイベントが別々に生成されます。
その際、ノートオンのないノートオフイベント、またノートオフのないノートオンイベントは無視されます、

- タイムシフト(*amount*): advance time by *amount*
- ベロシティ: change current velocity to *value*

表現力豊かなタイミングをサポートするために、最大1秒まで10ミリ秒単位で進むタイムシフトイベントでクロックを制御します。（BPMに応じて拍が決まるモデルではありません）
ベロシティは、１～１２７までのMIDI準拠に変換されます

Performance RNN

学習済みモデル
モデルに使用している学習データはミネソタ大学で開催されたPiano-e-Competitionの参加者の演奏をMIDI化したものを使用しています。

GithubのPerformance RNNのページからダウンロードできます。
https://github.com/tensorflow/magenta/tree/master/magenta/models/performance_rnn

- performance
- performance_with_dynamics
- performance_with_dynamics_and_modulo_encoding
- density_conditioned_performance_with_dynamics
- pitch_conditioned_performance_with_dynamics
- multiconditioned_performance_with_dynamics

Performance RNN

各モデルにはそれぞれ少しづつ違いがあり特徴つけられています。

- performance
  ベロシティの変化を含みまないモデルです。

- performance_with_dynamics
  ベロシティの変化を含むモデルです。

- performance_with_dynamics_and_modulo_encoding
  ベロシティの変化を含むモデルです。modulo encodingというエンコード方法が用いられています。

- density_conditioned_performance_with_dynamics
  ベロシティの変化を含むモデルです。note densityの制御ができます。

- pitch_conditioned_performance_with_dynamics
  ベロシティの変化を含むモデルです。pitch conditionの制御ができます。

- multiconditioned_performance_with_dynamics
  ベロシティの変化を含むモデルです。note densityとpitch conditionの制御ができます。

モデルを定義する
Performance_model.py

モデルを定義する Performance_model.py

```
default_configs = {
    'performance': PerformanceRnnConfig(
        magenta.protobuf.generator_pb2.GeneratorDetails(
            id='performance',
            description='Performance RNN'),
        magenta.music.OneHotEventSequenceEncoderDecoder(
            magenta.music.PerformanceOneHotEncoding()),
        tf.contrib.training.HParams(
            batch_size=64,
            rnn_layer_sizes=[512, 512, 512],
            dropout_keep_prob=1.0,
            clip_norm=3,
            learning_rate=0.001)),
```

モデル名称

protobufを使用してidと
説明文を定義

tensorflowの
tf.contrib.training.HParams
メソッドを使用して
ハイパーパラメータの指定

モデルを定義する Performance_model.py

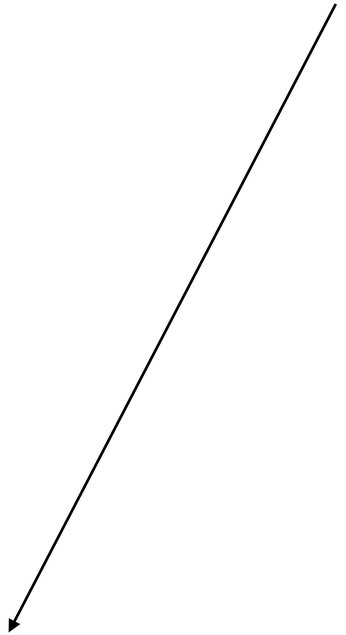
```
'performance_with_dynamics': PerformanceRnnConfig(
        magenta.protobuf.generator_pb2.GeneratorDetails(
            id='performance_with_dynamics',
            description='Performance RNN with dynamics'),
        magenta.music.OneHotEventSequenceEncoderDecoder(
            magenta.music.PerformanceOneHotEncoding(
                num_velocity_bins=32)),
        tf.contrib.training.HParams(
            batch_size=64,
            rnn_layer_sizes=[512, 512, 512],
            dropout_keep_prob=1.0,
            clip_norm=3,
            learning_rate=0.001),
        num_velocity_bins=32),
```

Performance with dynamicsでは
ベロシティのパラメータが追加

モデルを定義する Performance_model.py

```
'density_conditioned_performance_with_dynamics': PerformanceRnnConfig(
        magenta.protobuf.generator_pb2.GeneratorDetails(
            id='density_conditioned_performance_with_dynamics',
            description='Note-density-conditioned Performance RNN + dynamics'),
        magenta.music.OneHotEventSequenceEncoderDecoder(
            magenta.music.PerformanceOneHotEncoding(
                num_velocity_bins=32)),
        tf.contrib.training.HParams(
            batch_size=64,
            rnn_layer_sizes=[512, 512, 512],
            dropout_keep_prob=1.0,
            clip_norm=3,
            learning_rate=0.001),
        num_velocity_bins=32,
        control_signals=[
            magenta.music.NoteDensityPerformanceControlSignal(
                window_size_seconds=3.0,
                density_bin_ranges=[1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0])
        ]),
```

Density Conditioned performance with dynamicsでは Densityのパラメータが追加
Densityは時間あたりの音の密度



モデルを定義する Performance_model.py

```
'pitch_conditioned_performance_with_dynamics': PerformanceRnnConfig(
        magenta.protobuf.generator_pb2.GeneratorDetails(
            id='pitch_conditioned_performance_with_dynamics',
            description='Pitch-histogram-conditioned Performance RNN'),
        magenta.music.OneHotEventSequenceEncoderDecoder(
            magenta.music.PerformanceOneHotEncoding(
                num_velocity_bins=32)),
        tf.contrib.training.HParams(
            batch_size=64,
            rnn_layer_sizes=[512, 512, 512],
            dropout_keep_prob=1.0,
            clip_norm=3,
            learning_rate=0.001),
        num_velocity_bins=32,
        control_signals=[
            magenta.music.PitchHistogramPerformanceControlSignal(
                window_size_seconds=5.0)
        ]),
```

Pitch Conditioned performance with dynamicsでは PithHistogramが追加

PithHistogramは１２音それぞれに重みを任意で指定できる

Performance RNN
で音楽生成

Performance RNNで音楽生成

```
BUNDLE_PATH=<absolute path of .mag file>
CONFIG=<one of 'performance', 'performance_with_dynamics', etc., matching the bundle>

performance_rnn_generate \  #python generate.pyでも可能です
--config=${CONFIG} \
--bundle_file=${BUNDLE_PATH} \
--output_dir=/tmp/performance_rnn/generated \
--num_outputs=10 \  #生成曲数
--num_steps=3000 \  #ステップ、拍や秒数ではなく音数
--primer_melody="[60,62,64,65,67,69,71,72]"
```

生成オプションコマンド

primer_pitches: 生成元となる音を和音としてベクトルで指定　例："[60, 64, 67]”

primer_melody: 生成元となるメロディーをベクトルで指定 0~127 ノートナンバー -1ノートオフ -2ノーイベント

例: "[60, -2, 60, -2, 67, -2, 67, -2]”

primer_midi：生成元となる音をMIDIファイルで指定できる

notes_per_second：1秒内の音数

その他はMelody RNNなどと同様なので参照（qpmなどはありません）

Performance RNNで音楽生成

```
python magenta/models/performance_rnn/performance_rnn_generate.py \
--config=pitch_conditioned_performance_with_dynamics \
--bundle_file=/ご自身の環境に合わせパス指定/pitch_conditioned_performance_with_dynamics.mag \
--output_dir=/ご自身の環境に合わせパス指定/ \
--num_outputs=1 \
--num_steps=3000 \
--notes_per_second=1 \
--pitch_class_histogram="[2, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1]"
```

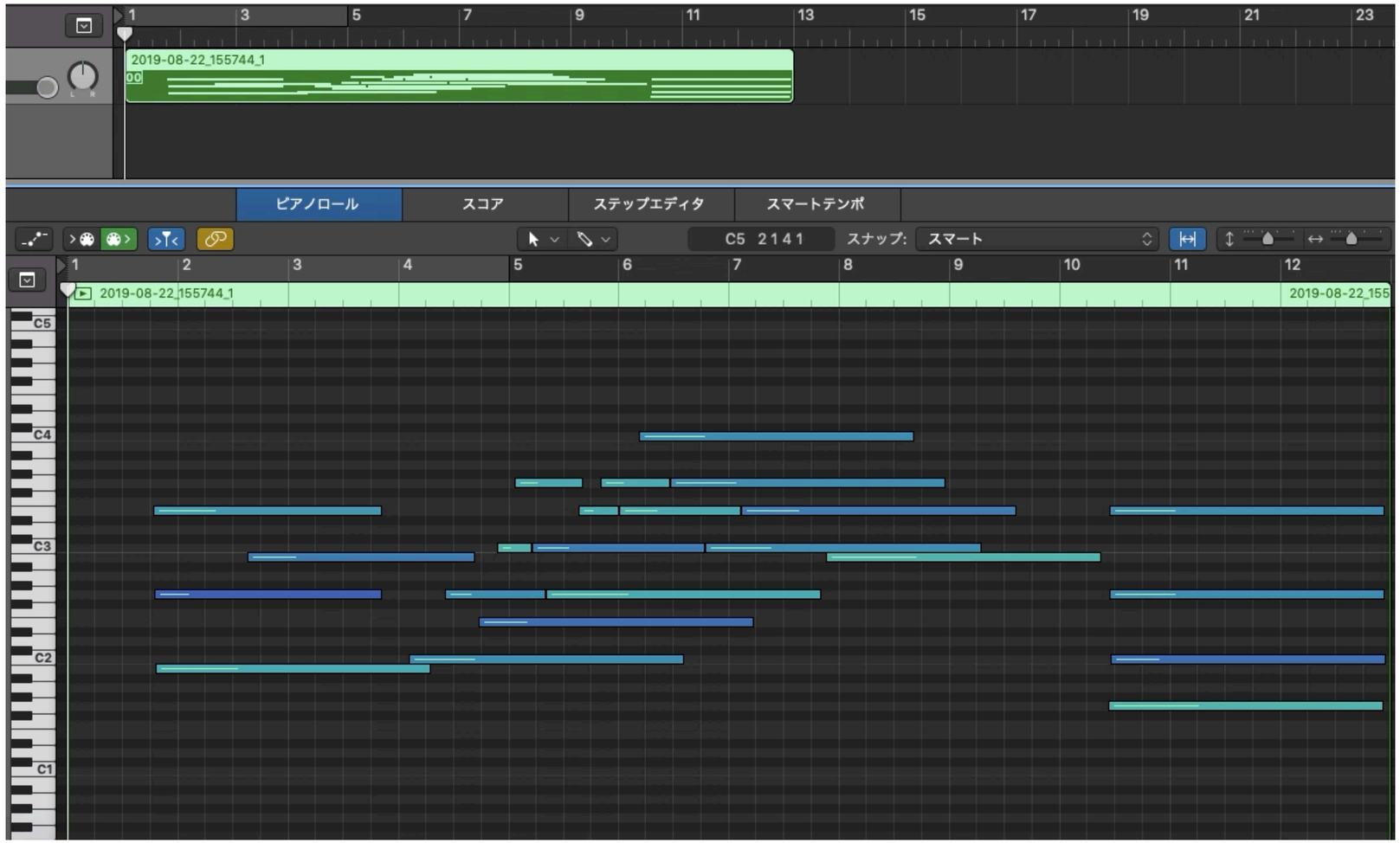
pitch_class_histogram
は使用する音を指定する事ができるコマンド

12音それぞれ個別に重みを数値で指定。

上記例はCM7（c, e, g, b）でCの重みのみ２。

notes_per_secondを１に指定。

Performance RNNで音楽生成



※資料では動画は再生できません

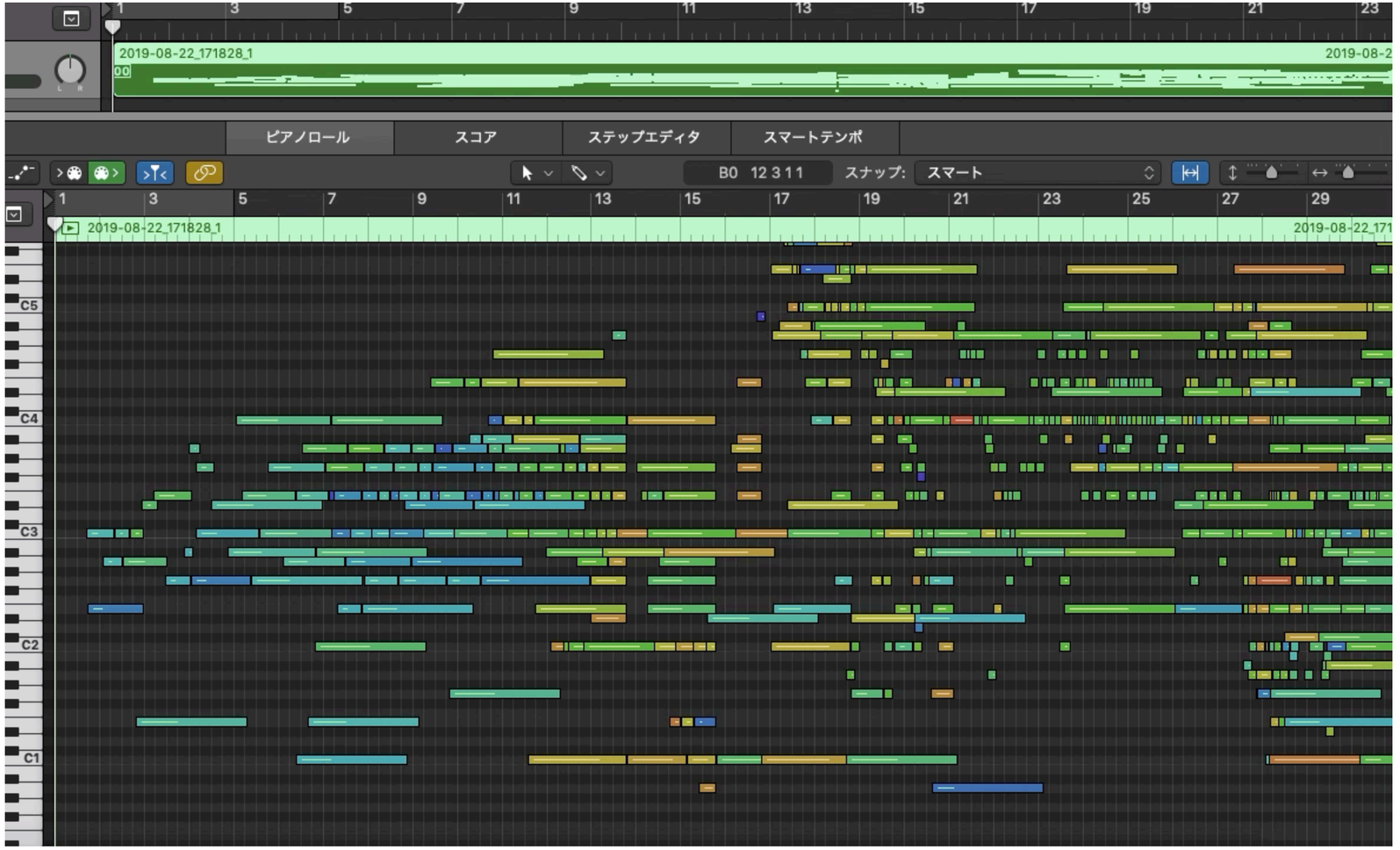
CM7（c, e, g, b）でnotes_per_secondを 1 に

Performance RNNで音楽生成

```
python magenta/models/performance_rnn/performance_rnn_generate.py \
--config=pitch_conditioned_performance_with_dynamics \
--bundle_file=/ご自身の環境に合わせパス指定/pitch_conditioned_performance_with_dynamics.mag \
--output_dir=/ご自身の環境に合わせパス指定/ \
--num_outputs=1 \
--num_steps=9000 \
--notes_per_second=64 \
--pitch_class_histogram=“[5, 0, 1, 1, 4, 0, 1, 3, 0, 2, 1, 1]”
```

ブルーノート的スケールでnotes_per_secondを６４に

Performance RNNで音楽生成



※資料では動画は再生できません　　ブルーノート的スケールで生成

Performance RNN
で独自モデル作成

独自の学習モデルの作成

Notesequenceの作成

学習させるMIDIファイル（またはMusicXMLファイル）があるディレクトリーの指定

```
INPUT_DIRECTORY=<folder containing MIDI and/or MusicXML files. can have child folders.>

SEQUENCES_TFRECORD=/tmp/notesequences.tfrecord

convert_dir_to_note_sequences \
  --input_dir=$INPUT_DIRECTORY \
  --output_file=$SEQUENCES_TFRECORD \
  --recursive
```

指定のディレクトリーにtfrecordファイルを作成（規定はtmpフォルダー）

Notesequence作成の実行

独自の学習モデルの作成

Sequence Exampleの作成

configを一つ指定

```
CONFIG=<one of 'performance', 'performance_with_dynamics', etc.>

performance_rnn_create_dataset \
--config=${CONFIG} \
--input=/tmp/notesequences.tfrecord \
--output_dir=/tmp/performance_rnn/sequence_examples \
--eval_ratio=0.10
```

notesequenceの指定

保存先ディレクトリーの指定

トレーニングデータと評価データの割合

２つのtfrecordファイルが作成されます。
トレーニングデータと評価データです。
eval_ratioは評価データの割合です。
例：0.10＝トレーニングデータ９０%、評価データ１０%

赤字がMelodyRNNなどとの違い

独自の学習モデルの作成

モデルの学習と評価

```
performance_rnn_train \
--config=${config} \
--run_dir=/tmp/melody_rnn/logdir/run1 \
--sequence_example_file=/tmp/melody_rnn/sequence_examples/training_performances.tfrecord \
--hparams="batch_size=64,rnn_layer_sizes=[64,64]" \
--num_training_steps=2000
```

configを１つ指定

実行ディレクトリーの指定

トレーニングデータの指定

ハイパーパラメーターの指定
バッチサイズはtfrecordのファイル数よりも少なく指定する

学習回数の指定

独自の学習モデルの作成

モデルの学習と評価

```
performance_rnn_train \
--config=${config} \
--run_dir=/tmp/melody_rnn/logdir/run1 \
--sequence_example_file=/tmp/melody_rnn/sequence_examples/training_performances.tfrecord \
--hparams="batch_size=64,rnn_layer_sizes=[64,64]" \
--num_training_steps=2000 \
--eval
```

configを１つ指定

実行ディレクトリーの指定

トレーニングデータの指定

ハイパーパラメーターの指定
バッチサイズはtfrecordのファイル数よりも少なく指定する

学習回数の指定

評価用コマンド

WindowsにNVIDIAのGPU環境作成

PCごとに作成方法が異なるためあくまで参考講義としてください

WindowsにNVIDIAのGPU環境作成

必要要件

ハードウェア
NVIDIA GPU グラフィックボード

ソフトウェア

以下の NVIDIAソフトウェアをシステムにインストールする必要があります。

- NVIDIA GPU ドライバ - CUDA 10.0 では 410.x 以降が必要です
- CUDA ツールキット - TensorFlow は CUDA 10.0 に対応しています（TensorFlow は 1.13.0 以降）
- CUPTI （CUDA Profiling Toolkit Interface）は CUDA ツールキットに付属します
- cuDNN SDK（7.4.1 以降）

WindowsにNVIDIAのGPU環境作成

- **NVIDIA GPU ドライバ**
  CUDA 10.0 では 410.x 以降が必要です

ドライバーはGPUの種類に合わせる必要があります。

- **CUDA**

（Compute Unified Device Architecture：クーダ）とは、NVIDIAが開発・提供している、GPU向けの汎用並列コンピューティングプラットフォーム（並列コンピューティングアーキテクチャ）およびプログラミングモデル。
専用のC/C++コンパイラ (nvcc) やライブラリ (API) などが提供されています。
なおNVIDIA製GPUにおいては、OpenCL/DirectComputeなどの類似APIコールは、すべて共通のGPGPUプラットフォームであるCUDAを経由することになります。

GPUドライバー
のダウンロードとインストール

NIVIDIAのドライバー検索ページで該当のドライバーバージョンを検索

https://www.nvidia.co.jp/Download/index.aspx?lang=jp

NIVIDIAのドライバー検索ページで該当のドライバーバージョンを検索

https://www.nvidia.co.jp/Download/index.aspx?lang=jp

例：RTX2080tiの場合バージョン430です

LINUX X64 (AMD64/EM64T) DISPLAY DRIVER

バージョン:	430.26
リリース日:	2019.6.10
オペレーティングシステム:	Linux 64-bit
言語:	Japanese
ファイルサイズ:	105.48 MB

ダウンロード

リリースハイライト | 製品サポートリスト | 追加情報

- Added support for the following GPUs:

  Quadro P520
  Quadro RTX 3000
  Quadro T1000
  Quadro T2000

- Fixed a bug, introduced in 415.13, that caused audio over DisplayPort to not work in some configurations.

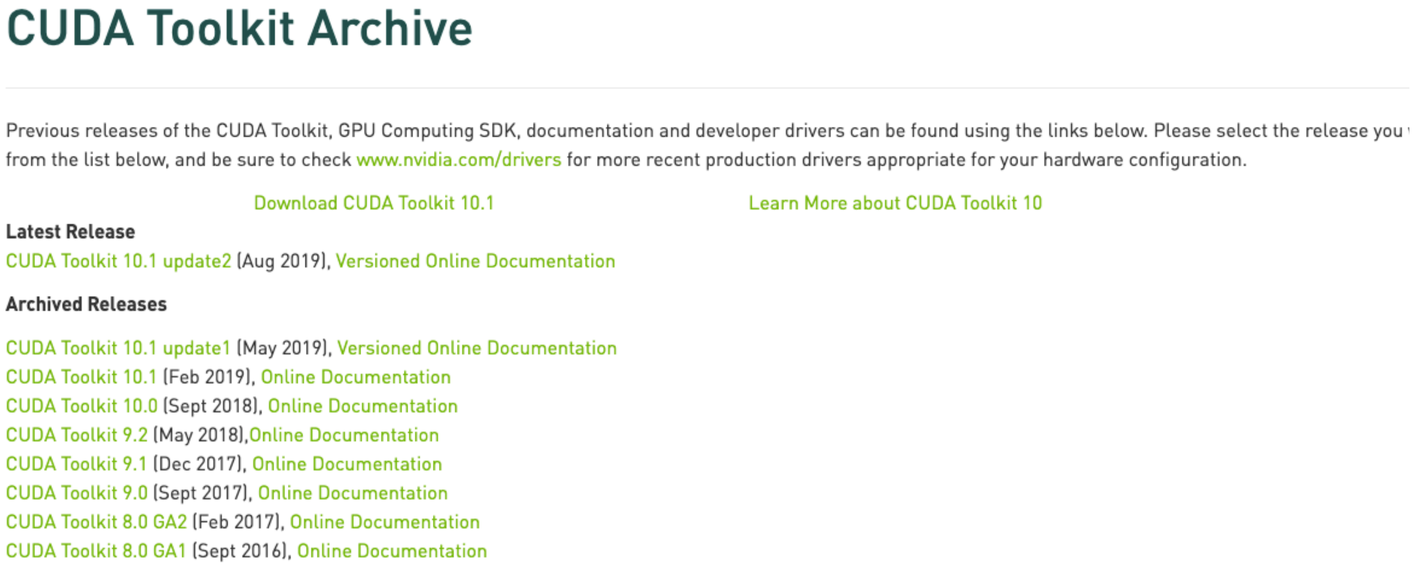
CUDA
のダウンロードとインストール

WindowsにNVIDIAのGPU環境作成

- **CUDA ツールキット**
  TensorFlow は CUDA 10.0 に対応しています（TensorFlow は 1.13.0 以降）CUPTI （CUDA Profiling Toolkit Interface）は CUDA ツールキットに付属します

下記よりダウンロードしインストール

https://developer.nvidia.com/cuda-toolkit-archive



CUDA Toolkit Archive

Previous releases of the CUDA Toolkit, GPU Computing SDK, documentation and developer drivers can be found using the links below. Please select the release you
from the list below, and be sure to check www.nvidia.com/drivers for more recent production drivers appropriate for your hardware configuration.

Download CUDA Toolkit 10.1 Learn More about CUDA Toolkit 10

Latest Release

CUDA Toolkit 10.1 update2 (Aug 2019), Versioned Online Documentation

Archived Releases

CUDA Toolkit 10.1 update1 (May 2019), Versioned Online Documentation
CUDA Toolkit 10.1 (Feb 2019), Online Documentation
CUDA Toolkit 10.0 (Sept 2018), Online Documentation
CUDA Toolkit 9.2 (May 2018),Online Documentation
CUDA Toolkit 9.1 (Dec 2017), Online Documentation
CUDA Toolkit 9.0 (Sept 2017), Online Documentation
CUDA Toolkit 8.0 GA2 (Feb 2017), Online Documentation
CUDA Toolkit 8.0 GA1 (Sept 2016), Online Documentation

cuDNN
のダウンロードとインストール

WindowsにNVIDIAのGPU環境作成

- **cuDNN**
  NVIDIAが提供している機械学習・Deep Learning用のライブラリです。

ダウンロードには登録が必要です。

下記よりダウンロードしインストール

https://developer.nvidia.com/rdp/cudnn-download

インストールした NVIDIA ソフトウェア パッケージのバージョンが一致していることをご確認ください。
cuDNN64_7.dll ファイルがないと、TensorFlow は読み込まれません。

CUDA、CUPTI、cuDNN の各インストール先ディレクトリを %PATH% 環境変数に追加します。
たとえば、CUDA ツールキットを C:¥Program Files¥NVIDIA GPU Computing Toolkit¥CUDA¥v10.0 にインストールし、cuDNN を C:¥tools¥cuda にインストールする場合、%PATH% を次のように更新します。

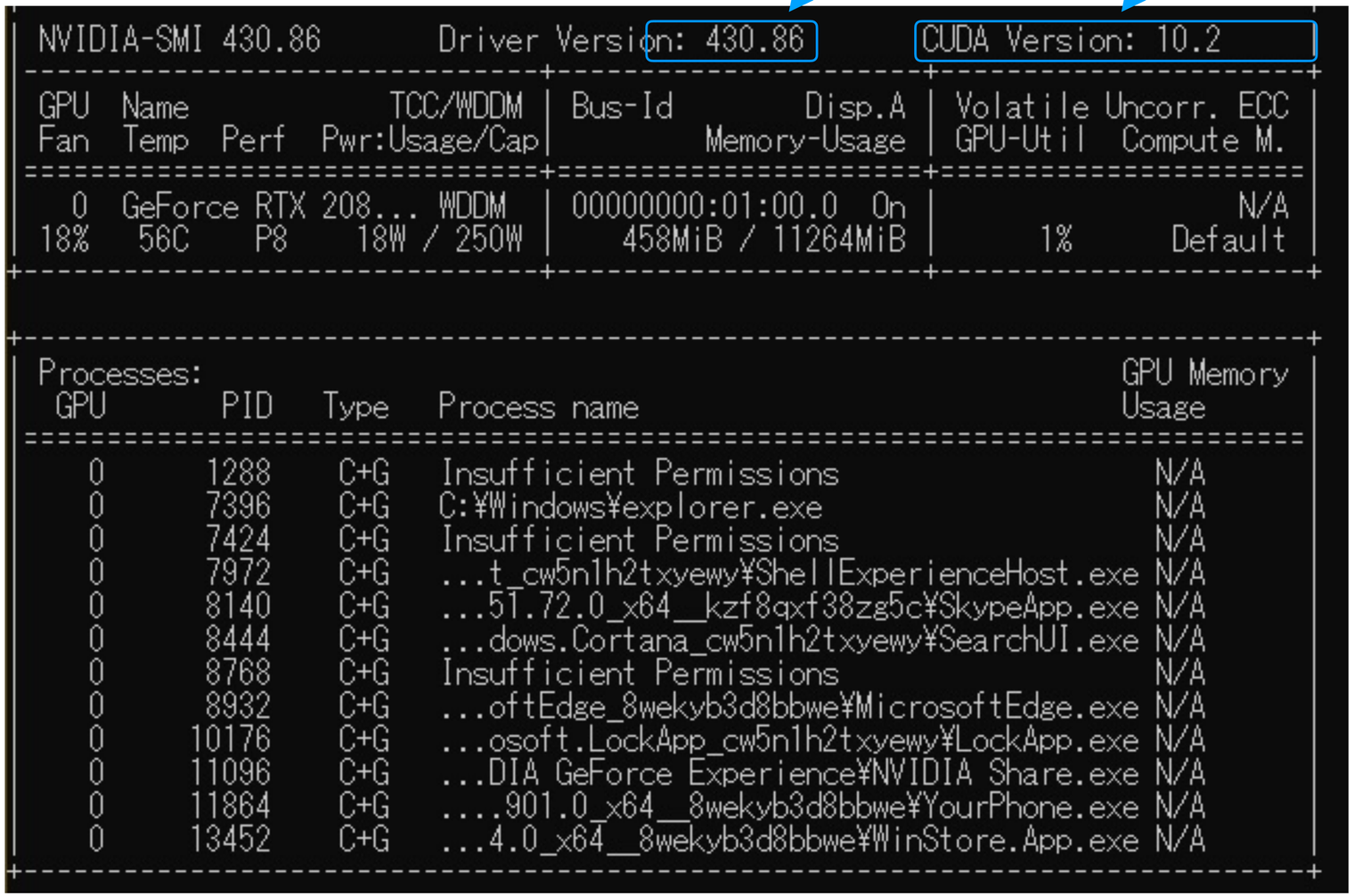
```
SET PATH=C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\bin;%PATH%
SET PATH=C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\extras\CUPTI\libx64;%PATH%
SET PATH=C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\include;%PATH%
SET PATH=C:\tools\cuda\bin;%PATH%
```

WindowsにNVIDIAのGPU環境作成

ドライバーバージョン

CUDAバージョン

ターミナルで nvidia-smi を実行すると表示されます



tensorflow-gpu
magenta-gpu
のインストール

tensorflow-gpuとmagenta-gpuのインストール

pipでインストールできます。

```
pip install tensorflow-gpu
pip install magenta-gpu
```

インストールできたかは　`pip list` でモジュール一覧表示して確認

※CPU版と両方インストールで大丈夫です。

プログラムで実行の際GPU環境であれば優先して使用されます。

tensorflow-gpu
の動作確認

tensorflow-gpuの動作確認

```
import tensorflow as tf

with tf.device('/gpu:0'):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
    c = tf.matmul(a, b)
with tf.Session() as sess:
    print (sess.run(c))
```

デフォルトGPU (device:0)を指名で行列計算を行う

1.0~6.0の６つの要素を持つ
２x３と３x２の行列の作成

行列積の実行

実行結果

```
[[22. 28.]
 [49. 64.]]
```

実行されればGPU使用されています

tensorflow-gpuの動作確認

```
from tensorflow.python.client import device_lib
device_lib.list_local_devices()
```

このコードを実行

実行結果

```
[name: "/device:CPU:0"
 device_type: "CPU"
 memory_limit: 268435456
 locality {
 }
 incarnation: 4078575508350901766, name: "/device:GPU:0"
 device_type: "GPU"
 memory_limit: 9116851241
 locality {
   bus_id: 1
   links {
   }
 }
 incarnation: 10344218195456938314
 physical_device_desc: "device: 0, name: GeForce RTX 2080 Ti, pci bus
id: 0000:01:00.0, compute capability: 7.5"]
```

GPUが表示されればOK

mnistプログラムでgpuの動作確認

mnistプログラムでgpuの動作確認

お配りしているmnist_cnn.pyでGPUでの学習を実行していただけます。

実行例
ターミナルから
python ¥ご自身の保存ディレクトリー¥mnist_cnn.py

またはjupyter notebookかGoogle Colabにコードをコピペで実行

次ページからGPUとCPUで学習速度の違いを検証します。

mnistプログラムでgpuの動作確認

mnistをGPUで

```
sk:0/device:GPU:0 with 8694 MB memory) -> physical GPU (device: 0, name: GeForce RTX 2080 Ti, pci bus id: 0000:01:00.0, compute
 7.5)
60000/60000 [==============================] - 7s 108us/step - loss: 0.2631 - acc: 0.9196 - val_loss: 0.0569 - val_acc: 0.9817
Epoch 2/12
60000/60000 [==============================] - 3s 56us/step - loss: 0.0877 - acc: 0.9737 - val_loss: 0.0439 - val_acc: 0.9847
Epoch 3/12
60000/60000 [==============================] - 3s 55us/step - loss: 0.0635 - acc: 0.9806 - val_loss: 0.0361 - val_acc: 0.9880
Epoch 4/12
60000/60000 [==============================] - 3s 55us/step - loss: 0.0548 - acc: 0.9838 - val_loss: 0.0332 - val_acc: 0.9885
Epoch 5/12
60000/60000 [==============================] - 3s 55us/step - loss: 0.0466 - acc: 0.9856 - val_loss: 0.0288 - val_acc: 0.9906
Epoch 6/12
60000/60000 [==============================] - 3s 56us/step - loss: 0.0411 - acc: 0.9877 - val_loss: 0.0305 - val_acc: 0.9895
Epoch 7/12
60000/60000 [==============================] - 3s 55us/step - loss: 0.0349 - acc: 0.9887 - val_loss: 0.0269 - val_acc: 0.9906
Epoch 8/12
60000/60000 [==============================] - 3s 56us/step - loss: 0.0321 - acc: 0.9905 - val_loss: 0.0277 - val_acc: 0.9910
Epoch 9/12
60000/60000 [==============================] - 3s 55us/step - loss: 0.0308 - acc: 0.9905 - val_loss: 0.0270 - val_acc: 0.9917
Epoch 10/12
60000/60000 [==============================] - 3s 56us/step - loss: 0.0286 - acc: 0.9911 - val_loss: 0.0253 - val_acc: 0.9923
Epoch 11/12
60000/60000 [==============================] - 3s 56us/step - loss: 0.0264 - acc: 0.9919 - val_loss: 0.0284 - val_acc: 0.9916
Epoch 12/12
60000/60000 [==============================] - 3s 56us/step - loss: 0.0254 - acc: 0.9921 - val_loss: 0.0261 - val_acc: 0.9926
Test loss: 0.026135571048522616
Test accuracy: 0.9926
```

1 epoch３．５秒ほど。

12 epochでも４５．５秒です。

約１０倍という速度で学習できています。

mnistプログラムでgpuの動作確認

mnistをCPUで

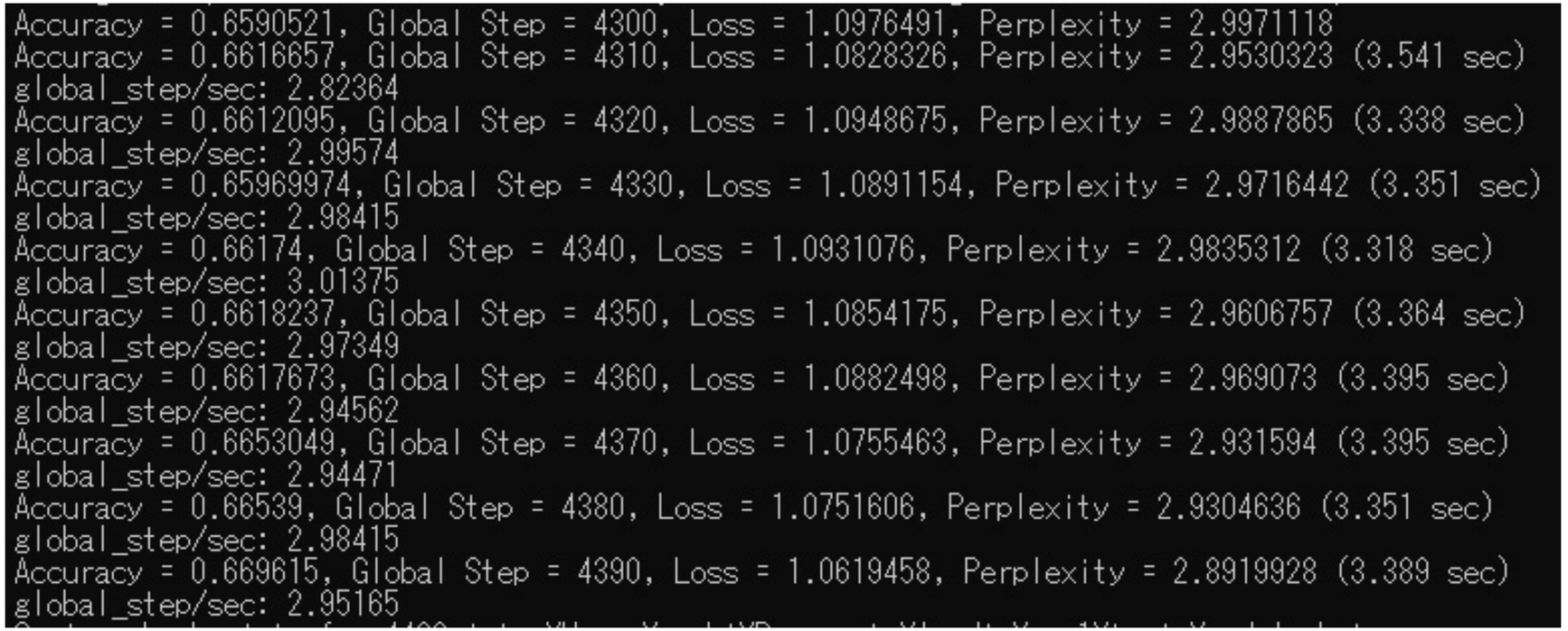
```
2019-07-26 23:14:54.372823: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this Tensor
low binary was not compiled to use: AVX2
60000/60000 [==============================] - 37s 614us/step - loss: 0.2736 - acc: 0.9166 - val_loss: 0.0604 - val_acc: 0.9796
Epoch 2/12
60000/60000 [==============================] - 37s 610us/step - loss: 0.0891 - acc: 0.9745 - val_loss: 0.0414 - val_acc: 0.9855
Epoch 3/12
60000/60000 [==============================] - 37s 610us/step - loss: 0.0651 - acc: 0.9810 - val_loss: 0.0484 - val_acc: 0.9843
Epoch 4/12
60000/60000 [==============================] - 37s 609us/step - loss: 0.0549 - acc: 0.9836 - val_loss: 0.0332 - val_acc: 0.9887
Epoch 5/12
60000/60000 [==============================] - 36s 608us/step - loss: 0.0474 - acc: 0.9859 - val_loss: 0.0290 - val_acc: 0.9906
Epoch 6/12
60000/60000 [==============================] - 36s 608us/step - loss: 0.0427 - acc: 0.9870 - val_loss: 0.0294 - val_acc: 0.9910
Epoch 7/12
60000/60000 [==============================] - 37s 611us/step - loss: 0.0375 - acc: 0.9879 - val_loss: 0.0308 - val_acc: 0.9900
Epoch 8/12
60000/60000 [==============================] - 36s 607us/step - loss: 0.0332 - acc: 0.9897 - val_loss: 0.0291 - val_acc: 0.9903
Epoch 9/12
60000/60000 [==============================] - 36s 608us/step - loss: 0.0312 - acc: 0.9910 - val_loss: 0.0277 - val_acc: 0.9908
Epoch 10/12
60000/60000 [==============================] - 39s 647us/step - loss: 0.0285 - acc: 0.9913 - val_loss: 0.0269 - val_acc: 0.9918
Epoch 11/12
60000/60000 [==============================] - 36s 608us/step - loss: 0.0258 - acc: 0.9918 - val_loss: 0.0284 - val_acc: 0.9907
Epoch 12/12
60000/60000 [==============================] - 37s 610us/step - loss: 0.0247 - acc: 0.9919 - val_loss: 0.0282 - val_acc: 0.9919
Test loss: 0.02818426782928136
Test accuracy: 0.9919
```

１epoch３７秒ほどです。

12 epoch学習させていますので７分２５秒ほどです。

mnistプログラムでgpuの動作確認

GPUで音楽データを５００曲学習



１０回の学習x１０で１００回
平均３．４秒x１０で３４秒ほどです。

mnistプログラムでgpuの動作確認

CPUで音楽データを５００曲学習

```
ccuracy = 0.6474323, Global Step = 4201, Loss = 1.1721498, Perplexity = 3.2289267
ccuracy = 0.65508264, Global Step = 4211, Loss = 1.1282436, Perplexity = 3.090224 (9.738 sec)
lobal_step/sec: 1.02702
ccuracy = 0.65816337, Global Step = 4221, Loss = 1.1185216, Perplexity = 3.0603263 (11.169 sec)
lobal_step/sec: 0.895325
ccuracy = 0.65741104, Global Step = 4231, Loss = 1.1203393, Perplexity = 3.0658941 (12.695 sec)
lobal_step/sec: 0.787709
ccuracy = 0.6611363, Global Step = 4241, Loss = 1.1015493, Perplexity = 3.0088239 (13.597 sec)
lobal_step/sec: 0.735476
ccuracy = 0.65728456, Global Step = 4251, Loss = 1.1088501, Perplexity = 3.0308712 (14.153 sec)
lobal_step/sec: 0.706547
ccuracy = 0.6561398, Global Step = 4261, Loss = 1.109446, Perplexity = 3.032678 (15.018 sec)
lobal_step/sec: 0.665875
ccuracy = 0.6616916, Global Step = 4271, Loss = 1.0974346, Perplexity = 2.996469 (16.178 sec)
lobal_step/sec: 0.618134
ccuracy = 0.6591863, Global Step = 4281, Loss = 1.0925922, Perplexity = 2.9819942 (16.738 sec)
lobal_step/sec: 0.597399
ccuracy = 0.66220206, Global Step = 4291, Loss = 1.0916753, Perplexity = 2.979261 (17.533 sec)
lobal_step/sec: 0.570402
```

１０回の学習x１０で１００回

平均１４秒x１０で２分２０秒ほどです。