WHITE PAPER

ディープラーニングは、時系列予測でも最強なのか？

～RNNと従来手法との対比から見える使いどころ～

SAS THE POWER TO KNOW.

目次

1　概要 1
2　RNNはどのようなモデルでしょうか？ 1
2.1　ARIMAモデルと状態空間モデル 1
2.2　ディープラーニング 2
2.3　RNN（Recurrent Neural Network） 3
3　どのような場合にRNNを使うべきなのでしょうか? 5
3.1　RNNの長所 6
3.2　RNNの短所 6
3.3　RNNの使いどころ 7
4　SAS® Viya®におけるRNNを用いた時系列予測の例 8
4.1　太陽光発電：前処理 9
4.2　太陽光発電：モデル構築 11
4.3　太陽光発電：後処理 15
5　まとめ 17
Appendix 18
A.　CAS Serverについて 18
B.　CAS ActionとPython向けAPI（SWATとDLPy） 19
C.　参考情報 20

1　概要

本記事では、SAS® Viya® の機能を用いたディープラーニングによる時系列予測について紹介します。

近年、アナリティクスにおける有効な手法としてディープラーニングが注目を浴びています。ディープラーニングは予測モデルの一つで、同じく予測モデルの一つであるニューラルネットワークにおいて隠れ層を多数重ねたものです。ディープラーニングの特徴は豊かな表現力にあります。様々な応用に適した形のディープラーニングが提案され、画像認識、音声認識、言語翻訳、時系列予測等の広い範囲に適用されています。

以下では、時系列データに適したディープラーニングであるRNN（Recurrent Neural Network）について紹介します。まず、ARIMAモデルや状態空間モデル等の時系列モデルと比較しながら、RNNが具体的にどのような構造を持つモデルなのかを紹介します。次に、RNNがどのような長所や短所を持つモデルなのか、どのような場面で用いるのに適しているのかについて説明します。最後に、SAS ViyaによるRNNを用いた時系列予測の例を紹介します。

2　RNNはどのようなモデルでしょうか？

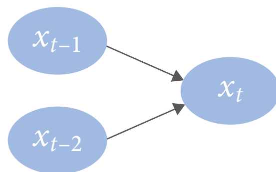
2.1　ARIMAモデルと状態空間モデル

時系列予測のためのモデルであるRNNについて説明する前に、時系列予測のためによく用いられるモデルであるARIMAモデルと状態空間モデルについて説明します。

ARモデル（Autoregressive model）は、予測値を過去の観測値の線形結合で表す時系列予測モデルです。その予測式は、例えば以下の様に書かれます。

$$x_t = a_1 x_{t-1} + a_2 x_{t-2}$$

これは、1時刻前の観測値と2時刻前の観測値を用いて、時刻tの値を予測する式になっています。このモデルをグラフで表すと以下の様になります。

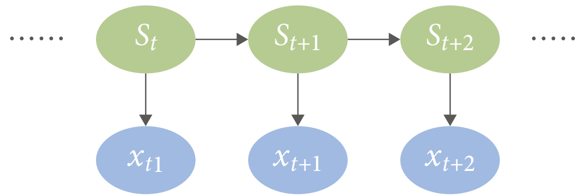


このグラフはx_{t-1}とx_{t-2}を入力としてx_tの値が決まるということを図示しています。

このグラフはARモデルの予測式（上の式の右辺）に周期性を表す項とトレンド（一定の割合での増加または減少）を表す項を追加したモデルがARIMAモデルです。①周期性がある、②トレンドがある、③過去の値から現在の値が決まる、という時系列の主要な性質を考慮できるため、ARIMAモデルは時系列予測の分野に広い範囲で適用されてきました。

ARモデルやARIMAモデルは、予測式の詳細こそ異なるものの、過去の観測値（のみ）を用いて予測値を表すという点は共通しています。これに対して、「状態」（もしくは潜在変数）という概念を導入したモデルが状態空間モデルです。

状態空間モデルでは、直接観測することのできない「状態」というものを考えます。そのうえで、状態が時間的に変化し、各時刻の状態に従って各時刻の観測値が現れると考えます。状態空間モデルをグラフで表現すると、例えば以下の様になります。



「状態」は、潜在変数とも呼ばれます。ここでいう潜在変数とは、時刻毎に推定すべきパラメータのことを指します。ARモデルのパラメータ (a_1, a_2) と異なるのは、データ全体に対して一定数のパラメータがあるのではなく、時刻毎に（つまり時間長に比例した数だけ）パラメータがある※1という点です。

状態空間モデルを用いることの主な利点は、状態という概念を導入したモデル（予測式）を考えることでARIMAモデルと比較してより一般的な表現が可能になる※2ところにあります。

2.2　ディープラーニング

時系列予測のためのディープラーニングモデルであるRNNについて説明する前に、ディープラーニングの概要を説明します。

ディープラーニングは、ニューラルネットワークという予測モデルの中の一種類です。ニューラルネットワークとは、複数の関数を並列もしくは直列につなげて予測式を導く予測モデル※3です。複数の関数を並列につなげた予測式とは例えば以下の様になります。

$$y = \omega_1 f_1(x) + \omega_2 f_2(x) + \ldots$$

ここでは、xが入力変数、yがターゲット変数を表します。また複数の関数を直列につなげた予測式とは例えば以下の様になります。

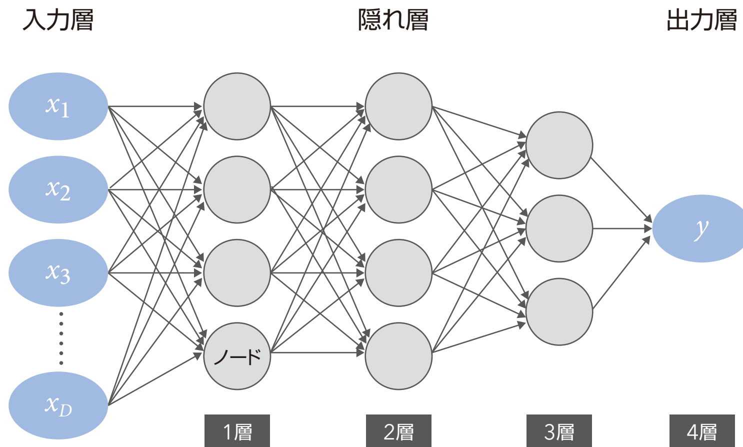
$$y = f_1(f_2(f_3(x)))$$

ニューラルネットワークはこれらを組み合わせて表され、グラフで表現すると以下の様になります。

※1　状態空間モデルのパラメータには、状態を表す潜在変数の値のみではなく、状態遷移を表すモデルのパラメータと、状態から観測値を生成するモデルのパラメータも含まれます。

※2　「より一般的な表現が可能になる」とは、「より広い範囲のデータに適用可能である」ということとほぼ同じです。

※3　ここでは、一旦時系列予測モデル（時間依存性）については考えないことにします。

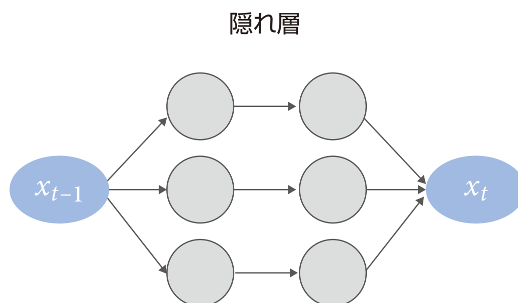


ニューラルネットワークにおける一つの関数（上図の○）をノードと呼び、並列したノード群（上図における縦方向の○の並び）を層と呼びます。層は、入力層、隠れ層、出力層の三つに分類されます。入力層は単にデータ（正確には教師あり学習における入力変数）を入力するだけの層です。ただし、入力層に標準化等のデータ加工の機能が含まれる場合もあります。出力層は、直前の層の値を用いて（何らかの関数に噛ませて）最終的な出力値（教師あり学習におけるターゲット変数の予測値）を出力します。入力·出力層に挟まれた全ての層を隠れ層もしくは中間層と呼びます。従って、ニューラルネットワークは、層を何層にするか[※4]、各層の中にいくつのノードを配置するか、各ノードに紐づく関数がどのような形か、によって特徴づけられます。

ニューラルネットワークにおいて層を多層に重ねたものをディープラーニングと呼びます。ディープラーニングの強みは、その豊かな表現力にあります。ニューラルネットワークで表現できるものの複雑さは、ノード数に対して多項式的に増加し、層の数に対して指数関数的に増加することが知られています[※5]。これは、層を重ねることで豊かな表現力を獲得できるということ、つまりディープラーニングによってより一般的な表現が可能になるということを示しています。実際に、画像認識、音声認識、翻訳、時系列予測等の広い範囲にディープラーニングが適用されていて、大きな成功を収めています。

2.3 RNN（Recurrent Neural Network）

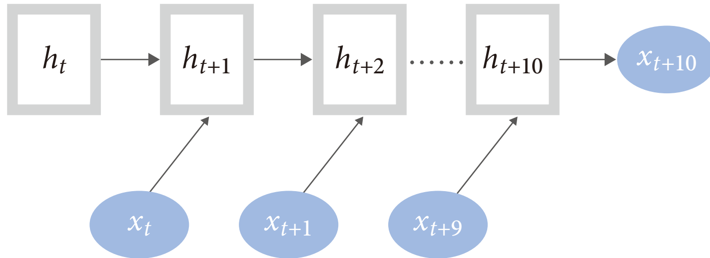
RNNは、ディープラーニングに時系列データを扱う構造を取り込んだものです。ここでは、まずRNNについて説明する前に、ディープラーニングを用いて時系列予測をするにはどのようにすれば良いのかを考えます。素朴に考え付くのは、以下のような構造のネットワークを考えることです。



※4 ニューラルネットワークにおいて、層の数は入力層以外の層の数（隠れ層＋出力層の数）として定義されます。上図の例では、隠れ層3層＋出力層1層で、4層のニューラルネットワークを構成していることになります。

※5 Montufar, Guido F., et al. "On the number of linear regions of deep neural networks." *Advances in neural information processing systems*. 2014

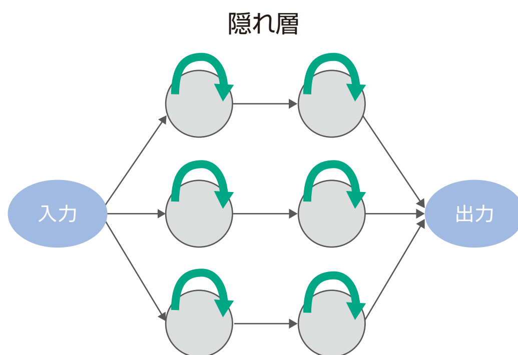
ネットワークは複雑な構造を持ちますが、このモデルで x_{t-1} から x_t を予測するというだけのモデルです。そのため、周期性やトレンド等を上手く表現するのは困難です。これでは、より一般的な表現が可能になるというディープラーニングの利点が活かされません。そこで、次に考えられるのは、以下のように複数の時刻に対応した複数の層を用いてネットワークを構成することです。四角形は隠れ層を表しています。



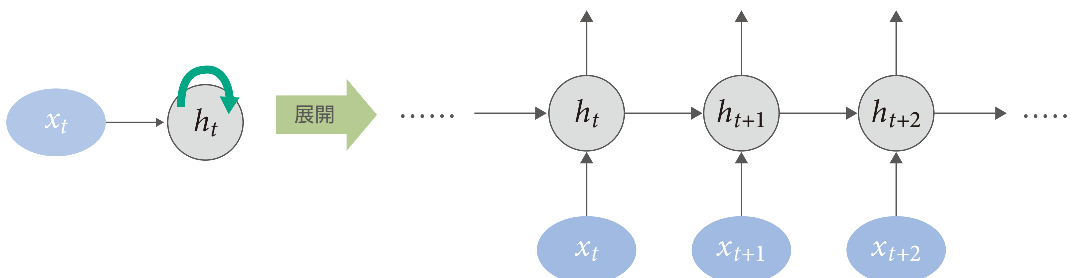
このグラフは一見良さそうですが、問題があります。それは、長周期の時系列を表すのが困難であることです。例えば、100時刻前の情報を考慮したいとなった場合に、この構造のディープラーニングでは100個の層が必要となります。これに伴って、モデルのパラメータ数が非常に大きくなり、モデルがロバストではなくなります。

このような困難を回避する一つの方法は、ネットワークの構造を工夫する代わりに、ノードに時系列予測に適した構造を持たせることです。そのようなディープラーニングの一つとしてRNN（Recurrent Neural Network）が挙げられます。

RNNでは、各ノードが前時刻の値を入力とする自己ループを持ちます。



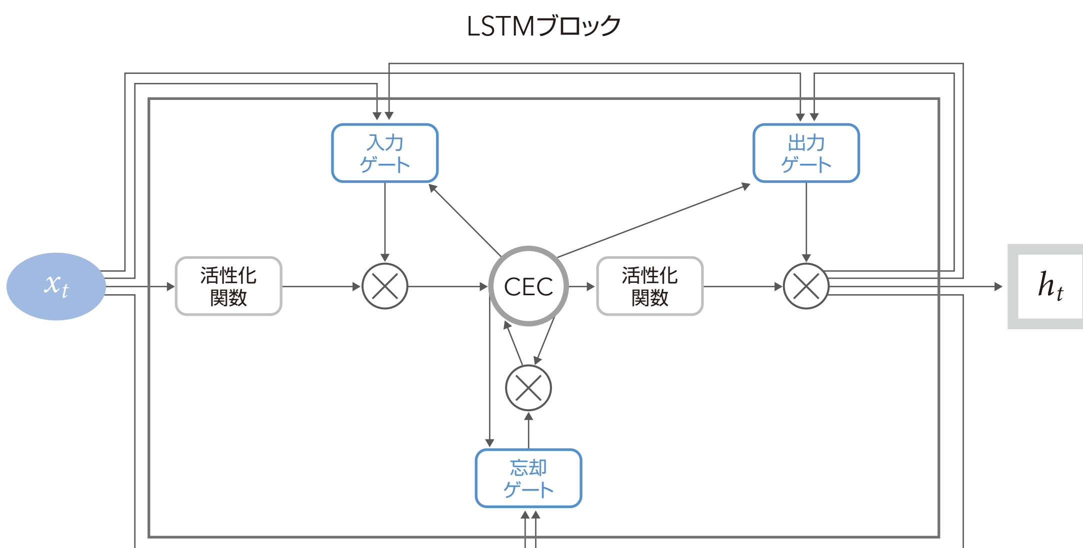
このグラフの一つのノードを別の描き方で表すと以下の様になります。



このグラフの構造は状態空間モデルのものと似ていますが、一点大きな違いがあります。それは、入力からノードの状態への矢印があることです。これはつまり、各時刻における状態が、前時刻の入力の情報を持つ（「記憶を持つ」）ということです。そのため、この構造によって、大量の層を重ねることなしに長周期の時系列を表現することができるようになります。

ここで述べたように、RNNは、状態空間モデルのより複雑な形（潜在変数＝ノードの状態として、潜在変数間の関係をより複雑なネットワークで表したもの）とみなせます。従って、RNNを用いることで、状態空間モデルよりも更に一般的な表現が可能となります。

時系列予測に対して、RNNの中でも特によく使われるモデルとしてLSTM（Long Short-Term Memory）が挙げられます。LSTMは、RNNにおいてノードにある特定の構造を持たせたものになります。LSTMでは、RNNにおける自己ループ付きのノードが以下のような「LSTMブロック」と呼ばれるもので置き換えられています。



このブロックを導入することで、LSTMでは時系列が持つ長期依存性を効率よく学習することが可能になります。LSTMブロックは以下で挙げる二つの特徴的な構造を持っています。一つ目は入力ゲート・出力ゲートと呼ばれるもので、対象となる時系列の持っている特徴的なパターンから外れる値を学習対象から外すことでパターンの効率的な学習を助けます。二つ目は忘却ゲートと呼ばれるもので、それまでに記憶した状態をある程度忘れさせることで時系列の状態遷移に対してモデルが追随することを助けます。

3　どのような場合にRNNを使うべきなのでしょうか?

時系列予測を扱う場合に限らず、万能なモデルや手法というものは存在しません。データの性質と照らし合わせて長所や短所を考慮したうえで、用いるモデルを選択する必要が常にあります。これは、高性能なモデルとして知られているRNN（ディープラーニング）においても例外ではありません。長所と短所を認識し、長所を活かせる（短所に邪魔されない）場合にRNNを使うべきです。そこで、ここではRNNの長所と短所について纏めます。

3.1　RNNの長所

RNNの長所は主に以下の四つです。

一つ目は、RNNによってARIMAモデルや状態空間モデルよりも一般的な表現が可能となる点です。言い換えると、層を重ねることによって得られる豊かな表現力によって、より広い範囲に適用可能なモデルができるということです。時系列予測に限らず、予測モデルを構築する際には、複数のモデル(例えば、ARIMAモデル、状態空間モデル、RNN)を構築してその中で最も精度が高いものを選ぶということがされます。これは、最適なモデルをデータの性質から決めるということです。この一つ目の長所から、多くの場合にRNNが選ばれると期待されます。

二つ目は、長期依存性を持つ時系列に対するモデル化が可能である点です。上述の通り、過去の時系列の状態を記憶することによって、RNNは時系列の長期依存性をモデルで再現します。長期依存性とは、100ステップ以上前のような多ステップ前[※6]の情報がその時点での時系列の値に影響を与える性質のことです。長期依存性を持つ時系列の値は、短期的に変動をしますが、その一方で全体的な値の大小が長期的に変動（値が徐々に上がったり徐々に下がったり）します。このような性質を持つデータの例として、株価や原油価格のような金融分野の時系列や、太陽の黒点数や川の水位のような自然科学分野の時系列等が挙げられます。

三つ目は、モデル構築の際に、時系列の性質に対する事前知識が不要である点です。RNNにおいては、周期の長さやトレンドの大きさ等はデータから自動的に学習されます。このため、何らかの事前知識を基にしたモデリングは不要となり、事前知識を獲得するコストやそれらに対応した設定について試行錯誤するコストが節約されます。

四つ目は、時間順序のみが意味を持つような時系列データ、例えば自然言語等、に対しても適用できる点です。自然言語処理では、各単語がどのように並んでいるかに意味があり、各単語がどの時刻に発生したかには意味がありません。このように、時間的に等間隔に並んでいるわけではなく、その順序のみが意味を持つようなデータに対してもRNNは適用できます。特に自然言語処理の分野において、RNNは高い成果をあげていて、機械翻訳や文章生成等の用途で広く用いられています。

3.2　RNNの短所

一方で、RNNには以下の三つの短所があります。

一つ目は、RNNが複雑なモデルであるという点です。RNNは多数のパラメータを持つ複雑なモデルであるため、対象とする時系列の学習オブザベーション数が少ない場合にはロバストなモデルを構成できません。例えば、月次データから在庫予測をしたいがデータは過去3年分（36ヶ月分）しかない、というような場合にはRNNは上手く機能しないと思われます。この例のように月次や年次のデータでは記録されているオブザベーション数が小さい場合が多いためにRNNが上手く機能しないことが多くなります。一方で、一秒毎や一時間毎に記録されているデータではオブザベーション数が大きい場合が多く、RNNが上手く機能することが多くなります。

二つ目は、RNNが複雑なモデルであるため、モデル推定に計算コストが掛かるという点です（あくまでARIMAモデルや状態空間モデルと比較してという意味で）。このため、大量の時系列を扱うような場合には計算コストが許容範囲に収まらない可能性があります。例えば、数万種類の商品に対して在庫予測をしたい、というような場面では、全てにRNNだけで対応するのは難しいという可能性が現れます。

※6　ここでいう「長期」とは、物理的な時間が長いことではなく、タイムステップ数が多いことを表します。例えば、月次データ（1タイムステップ=1ヶ月）を扱う場合には1年（=12ステップ）は長期ではありませんが、日次データ（1タイムステップ=1日）を扱う場合には1年（=365ステップ）は長期とみなされます。

三つ目は、RNNは可読性や解釈性が低いという点です。RNNを含むニューラルネットワークは複雑な構造のモデルであり、得られたモデルに対してどのような変数や成分が予測値に効いているか等といった定性的な解釈をすることは困難です。そのためにこれらのモデルの運用はブラックボックス化し易くなります。モデルがブラックボックス化することの弊害は主に以下の二つです。第一の弊害は、モデル構築後のアクションの設計が難しくなることです。このため、予測精度の他にビジネス上のアクションの設計及びそれに対する責任の取り方が重要視される場合には、RNNの適用が難しくなります。第二の弊害は、人の直感を働かせ難いため、モデルのチューニングの難易度が上がることです。上記第三の長所の裏返しで、RNNを用いる場合には事前知識を反映したモデリングが不要である一方で、ネットワークの構造という直感を働かせ難いものをチューニングする必要があります。

3.3　RNNの使いどころ

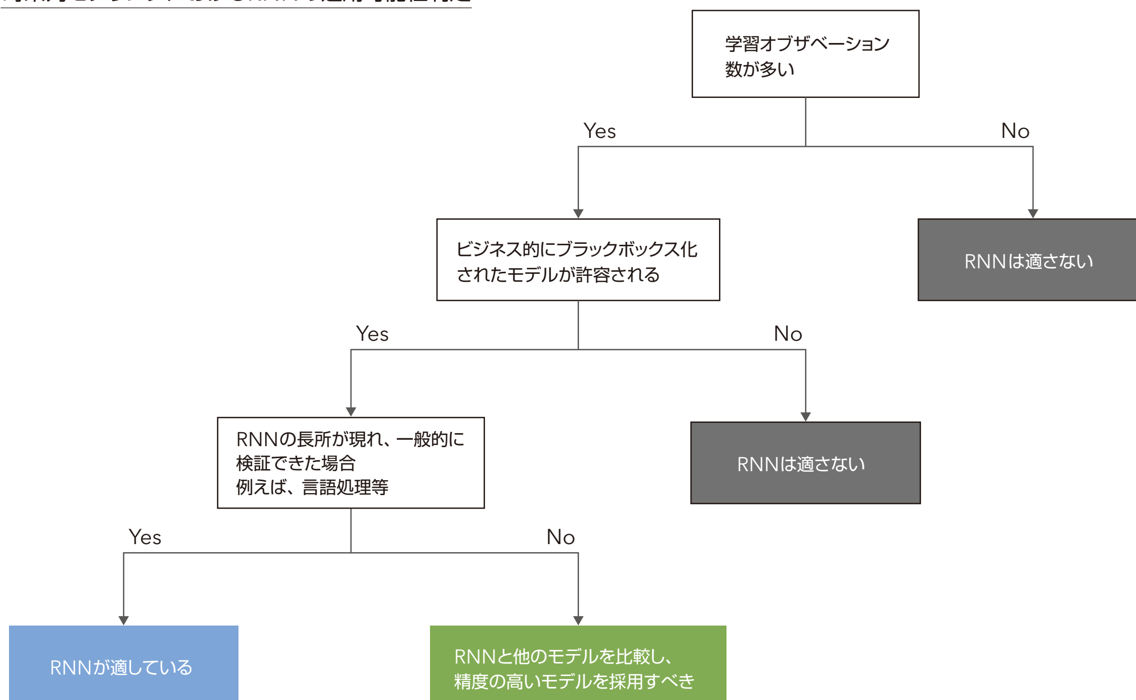
では、結局どのような場合にRNNを用いるべきなのでしょうか?　基本的な考え方は以下の三つにまとめられます。

第一に、RNNの短所が大きく現れる場合にはRNNを使わないようにします。例えば、学習オブザベーション数が少ない時系列が入力である場合や、モデルの解釈やそれを用いたアクションが重要視される場合にはRNN以外のモデルを考えることにします。

第二に、第一の条件がクリアされた場合に、RNNの長所が大きく現れる場合にはRNNを使うようにします。例えば、実行したい分析が自然言語処理である場合にはRNNを用いることにします。

第三に、第一の条件がクリアされたうえでRNNが適しているか分からない場合には、RNNを含む複数のモデルを構築してその中で最も精度が高いものを選ぶようにします。

これらの例を図にしたものが以下の図です。



こちらの図を見るとRNNの適用範囲が狭いように見えるかもしれませんが、そのようなことはありません。図の緑色のノードに多くの時系列データが当てはまります。そのうえで、データに依存することではありますが、他のモデルとの比較でRNNが選択されることが多いです。従って、広い範囲の時系列に適用可能であるという上述したRNNの長所とこちらの図は矛盾しません。

4 SAS® Viya®における RNNを用いた時系列予測の例

SAS Viyaはクライアント･サーバー型のシステム構成となっており、利用する際にはクライアント端末からサーバーへデータや命令を送って実行します。この時のサーバーをCAS（Cloud Analytic Services）サーバーと呼びます。CASサーバーへの通信はREST APIをベースとして提供されているため、クライアント端末からはPythonやRをはじめとした様々な言語プラットフォームを利用可能です。ここでは、クライアント端末としてJupyter Notebookをインストールした Python開発環境を使用した例を紹介します。

SAS ViyaのPythonによる開発時の特徴として、SWAT、DLPyというライブラリの存在があります。SWATとDLPyはそれぞれ、SAS Viyaの利用とディープラーニングモデルの構築を行う際に有用なPythonモジュールであり、SASにより開発･提供がされています。このモジュールを利用することで、ユーザはREST APIの詳細を意識することなく、慣れ親しんだPythonプログラミングの感覚でSAS Viyaを使いこなすことができます。

SAS Viyaのクライアント･サーバー間の通信構成とSWAT、DLPyの関係については、Appendixにまとめてありますのでご参照ください。

SASは環境に優しい企業として認知されており、9つの太陽光発電施設から年間380万キロワット時（kWh）を超えるクリーンで再生可能なエネルギーを生み出しています。太陽光エネルギーを配電網に組み込むことは、持続可能なビジネス慣行における重要なステップであり、そのためには太陽光発電の正確な予測がしばしば必要とされます。一方で、気象条件は非常に不安定であることが多いため、正確な予測が困難であることが多いです。

以下では、太陽光発電量の予測を例として、SAS ViyaにおけるRNNを用いて時系列予測を実行する方法を紹介します。分析対象とするのは、一時間毎に記録された太陽光発電量のデータです。このデータを基に、一時間毎の太陽光発電量を予測することが、解くべき問題となります。分析に用いるデータセットは以下の変数からなり、予測対象である太陽光発電量の他に、4つの外部変数（独立変数）を含みます。

<従属変数>

- Solar_Power：太陽光発電施設における1時間当りの太陽光発電出力です。この変数の値を予測するのが今解くべき問題になります。

<独立変数>

- Global_Horizontal_Irradiance：太陽光発電施設に対する日射強度です。
- Hour：一日のうちの時間です。24時間の中の何時間目かを表します。
- Solar_Elevation_Angle：地平線と太陽の中心との間の角度です。
- Day_Time_Indicator：日中の場合は1、そうでなければ0となる二値フラグです。

分析対象データは1時間毎のデータですが、数日周期の変動が含まれると思われます。これは、100タイムステップ程度の長周期性が含まれるだろうということです。そのため、以下では、長周期の時系列予測に適しているRNNを用いて予測を実行します。

4.1　太陽光発電：前処理

分析のための前処理として、分析で必要な環境の準備と、RNNを学習させるのに必要なデータの加工を行います。

まずは必要なライブラリをインポートし、CASへアクセスするためのセッションを作成します。

1.ライブラリのインポート

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import swat
import dlpy
from dlpy import Sequential
from dlpy.layers import *
from dlpy.model import Optimizer, AdamSolver, Sequence
%matplotlib inline
```

2.CASセッションの作成

```
mysession = swat.CAS('localhost',5570, '[illegible]', '[illegible]', caslib="casuser")

#セッション情報の表示
print(mysession)
```

次に、csvをPandasで読込み、データの欠損状況を確認します。今回使用するデータは'Solar_Power'で26件の欠損が存在し、RNNでは欠損値を扱えないため、今回は前のレコードの値を代入する方法で補完します。

3.欠損値の確認とテーブルの作成

```
#データのロード
data=pd.read_csv("./RNN_solar_demo.csv")
```

```
#欠損のある項目と数の確認
data.isnull().sum()
```

```
datetime                      0
date                          0
Solar_Power                  26
Global_Horizontal_Irradiance  0
Hour                          0
Solar_Elevation_Angle         0
Day_Time_Indicator            0
dtype: int64
```

```
#前のレコードので穴埋め
data=data.fillna(method='ffill')
```

SAS Viyaを使用するには、データをCAS Tableに格納する必要があります。今回はfrom_pandas()を用いて、先ほど欠損値補完をしたPandasのデータを基に作成します。

```
#欠損値補完済みのDataFrameを基にCAS Tableを作成
solarDemo = dlpy.TimeseriesTable.from_pandas(conn=mysession,pandas_df=data,casout=dict(name='solarDemo',replace=True))
```

SAS Viyaでは直接ローカルよりCAS Tableへのデータのロードも可能です。Pythonではメジャーである Pandasから直接データを読み込めるため、他のPythonのライブラリをSAS Viyaで有効に活用できると言えます。

SAS Viyaにおいて RNNを学習させる際には、1オブザベーションが1つの部分時系列に対応するデータセット[※7]を入力データとして用意する必要があります。このようなデータの作成のために、全ての変数に対してラグ変数を作成します。ここでは部分時系列の長さを5とすることにします。適切な長さはデータの性質に依存しますが、ここでは5に固定します。DLPyのprepare_subsequences()を使用することで、ラグ変数を容易に作成できます（これ以降、DLPyに搭載の関数を使用）。

実行後に、以下のようなラグ変数が作成されているのを確認できます。

4.ラグデータの作成

```
#時系列データの設定とフォーマット化
solarDemo.timeseries_formatting(timeid='datetime',timeseries='Solar_Power',timeid_informat="datetime19.")
```

```
NOTE: Timeseries formatting is completed.
```

```
#ラグの作成
solarDemo.prepare_subsequences(seq_len=5, target='Solar_Power',missing_handling=None)
solarDemo.prepare_subsequences(seq_len=5, target='Global_Horizontal_Irradiance',missing_handling=None)
solarDemo.prepare_subsequences(seq_len=5, target='Hour',missing_handling=None)
solarDemo.prepare_subsequences(seq_len=5, target='Solar_Elevation_Angle',missing_handling=None)
solarDemo.prepare_subsequences(seq_len=5, target='Day_Time_Indicator',missing_handling=None)
```

or	datetime	Solar_Power_lag1	Solar_Power_lag2	Solar_Power_lag3	...	Solar_Elevation_Angle_lag1	Solar_Elevation_Angle_lag2	Solar_Elevation_Angle_lag3
.0	2008-12-19 00:00:00	NaN	NaN	NaN	...	NaN	NaN	NaN
.0	2008-12-19 01:00:00	0.0	NaN	NaN	...	-77.329325	NaN	NaN
.0	2008-12-19 02:00:00	0.0	0.0	NaN	...	-73.855250	-77.329325	NaN
.0	2008-12-19 03:00:00	0.0	0.0	0.0	...	-63.624366	-73.855250	-77.329325

ラグ変数を作成する際に、欠損になってしまった項目があるため、それらの除去を行います。

```
#ラグの作成により欠損となってしまった行の削除
solarDemo.dropna(how='any',inplace=True)
```

timeseries_partion()を用いてデータを学習、検証、テストデータに分割します。時系列データを使っているため、分割の際にはデータセットの時系列の順序を維持する必要があります[※8]。

5.データ分割

```
import datetime
training_start=datetime.datetime(2008,12,19,hour=0,minute=0,second=0)
validation_start = datetime.datetime(2015,2,18,hour=0,minute=0,second=0)
testing_start = datetime.datetime(2015,3,18,hour=0,minute=0,second=0)
end_time=datetime.datetime(2015,3,27,hour=23,minute=59,second=59)

train_tbl,valid_tbl,test_tbl = solarDemo.timeseries_partition(training_start=training_start,validation_start=validation_start, testing_start
```

```
NOTE: Training set has 53982 observations
NOTE: Validation set has 672 observations
NOTE: Testing set has 240 observations
```

※7　部分時系列とは、時系列$\{x_1, x_2, x_3, ...\}$の中から、$\{x_{11}, x_{12}, x_{13}, x_{14}\}$のように時間的に連続する部分を切り出したものを指します。今の例では、$\{x_{11}, x_{12}, x_{13}, x_{14}\}$が1オブザベーションとなるようなデータセットを構築することになります。

※8　時間が早い方から順に、学習、検証、テストデータとなるようにします。

また、以上の前処理はSAS data stepを用いても実行が可能です。以下は欠損値の削除をSAS data stepを用いて実行した例になります。

```
removalOfMissing =
'''
data solarDemo missingData;
set solarDemo;
if cmiss(of _all_) then output missingData;
else output solarDemo;
run;
'''
mysession.dataStep.runCode(removalOfMissing)
```

4.2　太陽光発電：モデル構築

次に、時系列予測のためのRNNモデルを構築します。モデルの構築は、①モデルの構造定義と、②モデルの学習の2つのパートからなります。

モデルの構造定義では、RNNのネットワーク構造を指定します。具体的には、入力層の性質、出力層の性質、隠れ層の性質、隠れ層のノード数及びノードの性質（LSTMブロックを用いるかどうか等）等を指定します。SAS Viyaでは、RNNの各層を入力層から順に定義して追加して行く形でネットワークの形を指定します。モデルの隠れ層の数および各層におけるノードの数は、データの複雑さやサイズに依存して決まります。今回の予測では隠れ層を2層、各隠れ層のノード数を15としてRNNを構築します。

Sequential()でモデルを作成し、モデルの識別名をsolar_forecastとします。add()を使用し、先ほど作成したモデルに層を追加していきます。入力層、隠れ層、出力層ではそれぞれ設定する内容が異なるため、それぞれの層を個別に定義し、重ねることで一つのモデルを完成させます。

入力層の設定では、"std="で入力層の規格化方法をSTDとします。STDは標準化を表し、平均が0、標準偏差が1となるように入力データを規格化することを意味します。

隠れ層では、LSTMブロックを用いたノード数が15の隠れ層を2つ作成します。隠れ層1では、"output_type="で層の出力タイプをsamelengthとし、これは入力と同じ長さの出力を生成するという意味です。"init="重みパラメータの初期値化方法をXAVIER[※9]としています。隠れ層2では層の出力のタイプをencodingとし、これは固定長のベクトルを生成するという意味です。

"reversed="で時間の逆順の伝播を許すかを指定します。逆順の伝播は言語翻訳のように未来の値を予測するわけではない場合に用いられる方法です。今の例では、未来の値を予測することが目的なので、隠れ層1、2ともに"reversed=False"とし、逆順の伝播を許さないと指定します。

出力層では、"act="で使用する活性化関数をIDENTITY（恒等関数）と指定しています[※10]。また、"error="で損失関数をNORMALとしています。これは最尤推定の際の誤差分布として正規分布を採用するという意味です。

※9　以下の論文で提示された重みパラメータの初期化方法です：
Glorot, X., and Bengio, Y. "Understanding the difficulty of training deep feedforward neural networks." In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249-256, 2010.

※10　今回の例では出力層でのみ活性化関数を指定しましたが、隠れ層においても指定すること（デフォルト値以外を指定すること）ができます。

6.モデルの定義

```
#モデルの作成
model1 = Sequential(mysession, model_table='solar_forecast')
#入力層
model1.add(InputLayer(std="STD"))
#隠れ層1
model1.add(Recurrent(rnn_type='LSTM', output_type= 'samelength',n=15,reversed_=False,init='XAVIER'))
#隠れ層2
model1.add(Recurrent(rnn_type='LSTM', output_type= 'encoding',n=15, reversed_=False))
#出力層
model1.add(OutputLayer(act='IDENTITY',error='NORMAL'))

#モデルのサマリを表示
model1.print_summary()
```

```
NOTE: Input layer added.
NOTE: Recurrent layer added.
NOTE: Recurrent layer added.
NOTE: Output layer added.
NOTE: Model compiled successfully.
```

今回は上記のように詳細にパラメータを設定しましたが、本来は最小限の設定でモデル作成が可能です。

モデルの定義（RNNのネットワーク構造の定義）が完了したので、次にモデルの学習（RNNのパラメータ推定）を実行します。

学習の前に、学習の際に必要な設定を行います。主に、モデル（RNN）の入力と出力、パラメータ推定の際の最適化手法、部分時系列、について設定します。

まずは、RNNの入力と出力が何かを指定します。今回の例では、過去の従属変数(太陽光発電量)と独立変数（4つの外部変数）が入力となり、予測したい時点の従属変数が出力となります。ここでは、事前に変数の文字列を格納するリストを作成します。

7.学習

```
#モデルの入出力の定義
#入力
inputs = ['Solar_Power_lag1', 'Global_Horizontal_Irradiance',
          'Hour', 'Solar_Elevation_Angle', 'Day_Time_Indicator']
for i in range(2,6):
    inputs += ['Solar_Power_lag{}'.format(i), 'Global_Horizontal_Irradiance_lag{}'.format(i-1),
             'Hour_lag{}'.format(i-1), 'Solar_Elevation_Angle_lag{}'.format(i-1),
             'Day_Time_Indicator_lag{}'.format(i-1)]

inputs.reverse()

#出力
outputs = 'Solar_Power'
```

その結果、以下のような入力変数のリストが作成されました。

```
In [2]: inputs

Out[2]: ['Day_Time_Indicator_lag4',
         'Solar_Elevation_Angle_lag4',
         'Hour_lag4',
         'Global_Horizontal_Irradiance_lag4',
         'Solar_Power_lag5',
         'Day_Time_Indicator_lag3',
         'Solar_Elevation_Angle_lag3',
         'Hour_lag3',
         'Global_Horizontal_Irradiance_lag3',
         'Solar_Power_lag4',
         'Day_Time_Indicator_lag2',
         'Solar_Elevation_Angle_lag2',
         'Hour_lag2',
         'Global_Horizontal_Irradiance_lag2',
         'Solar_Power_lag3',
         'Day_Time_Indicator_lag1',
         'Solar_Elevation_Angle_lag1',
         'Hour_lag1',
         'Global_Horizontal_Irradiance_lag1',
         'Solar_Power_lag2',
         'Day_Time_Indicator',
         'Solar_Elevation_Angle',
         'Hour',
         'Global_Horizontal_Irradiance',
         'Solar_Power_lag1']
```

次に、学習に用いられる最適化手法について設定します。RNNの学習では、予測値と観測値の二乗誤差のような「損失」を考え、損失を最小化するようにしてパラメータの値を求めます。ディープラーニングにおいて損失を大域的に最小化することは難しいため、逐次的に損失の微分を求め、それを用いてパラメータを更新する方法がとられます。微分を求める際に入力データセットの全オブザベーションを用いると時間が掛かるため、一部分（少数のオブザベーション）を用いて微分を求めます。このような部分データセットを「ミニバッチ」と呼びます。全ミニバッチに対して入力データセットに対して一周分だけパラメータを更新することを「エポック」と呼びます。また、微分の何倍分だけ（損失を小さくする方向にどれだけの「勢い」で）パラメータを変化させるかを決める係数を「学習率」と呼びます。更に、パラメータ推定の際には、微分の値を用いてどのようにパラメータを更新するかという「最適化手法」を指定する必要があります。

以下、具体的に学習のための設定を行います。mini_batch_sizeでは、一つのミニバッチ（時間的に連続する部分時系列の集合）にいくつの部分時系列が含まれるようにするか（RNNの学習用データセットの何オブザベーションをミニバッチとするか）を指定しますが、今回は4とします。

パラメータ更新がいつまでも終わらないことを防ぐために、エポック数の上限を指定します。max_epochsでエポック数の上限を指定し、今回は150とします。これは、パラメータ推定が収束しなくても150エポック分のパラメータ更新がなされたならば学習を終了する、ということです。

```
#最適化手法の設定
optimizer = Optimizer(algorithm=dlpy.model.AdamSolver(learning_rate=0.01, learning_rate_policy='step', gamma=0.2,
                                                      step_size=30, clip_grad_max=10000, clip_grad_min=-10000),
                      mini_batch_size=4,max_epochs=150)
```

今回は最適化手法としてADAMを使用することにし、AdamSolver()を使います。ADAMは、確率的勾配降下法に要素毎（パラメータ毎）の学習率の更新を加えたもので、それまでの更新量が大きい要素に対して学習率を小さくとることで速い収束を実現します。ここでは、初期の学習率（learning_rate）を0.01、学習率の減衰方法（learning_rate_policy）をstep、学習率にかける係数gammaを0.2、学習率の減衰タイミング（step_size）を30と指定します。これは、学習率の初期値を0.01ととること、及び学習率×gammaに従い、ミニバッチ30個毎に学習率を減衰させることを意味します。数値計算上の不安定さを回避するために、crip_gradによって微分値の最大と最小を指定します[※11]。今回は最大値を10000、最小値を－10000とします。

最後に部分時系列の設定を行います[※12]。部分時系列の設定においては、input_length（部分時系列の長さ。ここでは5）とtarget_length（従属変数の個数）、そしてtoken_sizeを指定します。token_sizeとは1タイムステップに含まれる変数の数で、ここでは従属変数1 ＋ 独立変数4 = token_size 5 となります。今回はsequence_optに格納されている情報を使用し、これら3つの値を指定します。これはprepare_subsequences()を実行した際に、指定された情報をもとに自動的に作成されたものです。ただし、自動的に作成された情報のtoken_sizeは1である（一つの変数に対してラグ変数の生成をしたため、変数の数は1と認識される）ため、今回は明示的に5と指定しています。

```
#部分時系列の設定
seq_spec  = Sequence(**train_tbl.sequence_opt)
seq_spec['token_size']=5
```

学習に関する全ての設定が完了したので、train()を使用し、先ほど設定した内容を基に学習データを用いてモデルの学習を行います。"seed="で乱数の種を指定し、今回は1234とします。

また、SAS ViyaではGPUでの演算をサポートしています。GPUはCPUと違い、単純な計算をたくさん高速に実行することが可能なアーキテクチャをしています。ディープラーニングのような、膨大な数の掛け算と足し算で構成されている演算であれば、GPUのほうがCPUよりも断然はやく計算することが可能です。今回はデータ量も多くないため、使用していませんが、train()の引数で"gpu=1"と指定することでサーバーが持っている全てのGPU使用が可能になります。

```
#モデルの学習
r1=model1.train(train_tbl,model='solar_forecast',
                optimizer=optimizer,
                sequence=seq_spec,
                valid_table=valid_tbl,
                inputs= inputs,target= outputs,seed=1234,
               model_weights=mysession.CASTable('rnn_weights',replace=True))
```

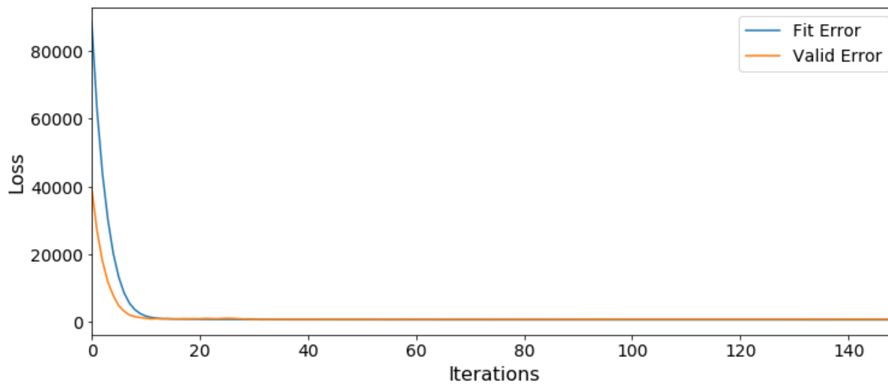
学習データ及び検証データに対して、エポック毎の損失をプロットします。

```
#学習結果のプロット
r1.OptIterHistory[['FitError', 'ValidError']][0:].plot(figsize=(12,5))
ax = plt.gca()
ax.get_xaxis().set_tick_params(labelsize=14)
ax.get_yaxis().set_tick_params(labelsize=14)
ax.legend(['Fit Error', 'Valid Error'], loc='upper right', prop={'size': 14})
ax.set_xlabel('Iterations',fontsize=16)
ax.set_ylabel('Loss',fontsize=16)
```

プロット結果から、損失は収束しており学習が上手くいっていることと、学習データと検証データで損失の推移が大きく変わらないこと（つまり大幅な過学習は起こっていないこと）が見てとれます。

※11 微分の値が最大値以上であれば最大値で置き換え、最小値以下であれば最小値で置き換えることで、数値計算を不安定にするような桁数の大きな値が現れるのを防ぎます。

※12 ここで部分時系列の設定をするのではなく、事前に作成した部分時系列データの情報（4つの独立変数を持つ長さ5の時系列を1オブザベーションとしてRNNを学習するためのデータセットを作ったこと）をSAS Viyaに教えるというのが正確な表現になります。



4.3　太陽光発電：後処理

最後に、テストデータを用いて学習済モデルを評価します。スコアリングの結果から予測誤差を算出するとともに、予測結果と実際の値をプロットします。

ここでは、ある時刻までの観測値を学習済みのモデルに入力して1時間後の値を予測し、観測値と予測値を比較することで予測精度を評価します。つまり、テストデータ内でモデルの更新はしませんが、時刻の予測値を求める際にはテストデータ内の時刻までの観測値を使用します。

score()を用いてスコアリングを実行します。"init_weights="ではスコアリングにおけるモデルパラメータの初期値が指定でき、今回は学習済みモデルのパラメータを指定します。"copy_vars="では入力データセットからスコアリング結果のデータセットにコピーする変数を指定でき、今回は'Solar_Power'と'datetime'の二つをコピーします。スコアリング結果をsolar_forecast_score_outというテーブルで出力します。

8.評価

```
#モデルの評価
#train同様にscoreのオプションの'gpu=1'の指定により使用が可能
r2= model1.score(test_tbl,model='solar_forecast',
                 init_weights='rnn_weights',
                 copy_vars=['Solar_Power','datetime'],
                 casout = dict(name='solar_forecast_score_out',replace=True))

tbl=mysession.CASTable(name='solar_forecast_score_out')
```

また、score()でも引数で"gpu=1"と指定することでサーバーが持っている全てのGPUの使用が可能になります。

RNNによる予測値には0未満の値が存在しますが、太陽光発電量には0未満の値は存在しません。このため、ここで0未満の予測値を0で置換します。新しく_DL_Pred_2という項目を作成し、そこに置換後の値を格納します。

9.プロット

```
#予測値が0以下の値を0に置換
tbl['_DL_Pred_2']=(tbl['_DL_Pred_'].clip_lower(0))
```

次に、予測結果を評価します。ここでは評価指標としてWAPE※13とRMSE※14を用いることにします。結果を見ると、WAPE=6.17%という良い結果が得られています。

```
# 結果のプロット
from matplotlib.dates import (DateFormatter, drange)

fig, ax = plt.subplots(1,1,figsize=(12,8))
data = localtest
x_data = pd.to_datetime(data['datetime'])
y_data = data['Solar_Power']
ax.plot(x_data, y_data,  linewidth=2, color='blue')

y2_data = data['_DL_Pred_2']

ax.plot(x_data, y2_data, linestyle='--', color='red', linewidth=2)
ax.get_xaxis().set_tick_params(direction='out', labelsize=14)
ax.get_yaxis().set_tick_params(labelsize=14)

xfmt = DateFormatter('%d%b%Y:%H')
ax.xaxis.set_major_formatter(formatter=xfmt)
fig.autofmt_xdate()
ax.set_xlabel('Datetime',fontsize=16)
ax.set_ylabel('Power Output (kWh)',fontsize=16)
ax.legend(['Actual Power', 'Forecasted Power'], loc='upper right', prop={'size': 14})
```

```
#予測誤差の表示

import numpy as np
#testscore_tbl = mysession.CASTable('solar_forecast_score_out')
localtest = tbl.fetch(to=240, orderby='datetime')['Fetch']
one_hour_rmse = np.sqrt(np.mean(np.square(localtest.Solar_Power - localtest._DL_Pred_2)))
one_hour_wape = np.sum(np.fabs(localtest.Solar_Power - localtest._DL_Pred_2))/np.sum(localtest.Solar_Power)
print("1-hour ahead Weighted Absolute Percentage Error(WAPE): {:.2f}%".format(one_hour_wape*100))
print("1-hour ahead Root Mean Squared Error(RMSE): {:.4f} kWh".format(one_hour_rmse))
```

```
1-hour ahead Weighted Absolute Percentage Error(WAPE): 6.17%
1-hour ahead Root Mean Squared Error(RMSE): 19.2038 kWh
```

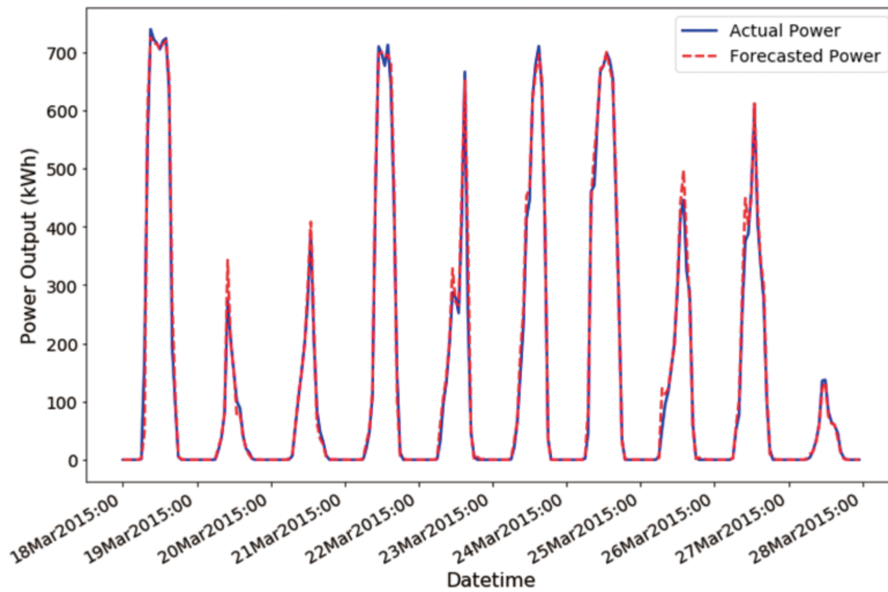
最後に予測結果と実際の値をプロットします。WAPEのような評価指標のみならず、時系列の形という意味でも、以下のように正確に予測がなされていることが分かります。

※13 WAPE: Weighted Absolute Percentage Errorの略で、次の式で定義されます。

$$\frac{\sum_{オブザベーション} |予測値 - 実測値|}{\sum_{オブザベーション} 実測値}$$

今の例では、太陽光発電量の予測誤差の絶対値の和が総発電量の何％に当たるかを表しています。分母の形から、実測値が常に0以上である場合以外ではこの指標を用いることはできません。

※14 RMSE: Root Mean Squared Errorの略で、$\sqrt{(予測値 - 実測値)^2 の平均値}$ で定義されます。今の例では、1タイムステップ（1時間）当りの太陽光発電量の誤差（比率ではなく絶対値）を表します。



5　まとめ

本記事では、SAS Viyaの機能を用いたディープラーニングによる時系列予測、特に、時系列データに適したディープラーニングであるRNN（Recurrent Neural Network）を用いた時系列予測について紹介しました。まず、ARIMAモデルや状態空間モデル等の時系列モデルと比較しながら、RNNが具体的にどのような構造を持つモデルなのかを説明しました。次に、RNNがどのような長所や短所を持つモデルなのか、どのような場面で用いるのに適しているのかについて説明しました。最後に、SAS ViyaによるRNNを用いた時系列予測の例を紹介しました。

最後の例からも分かるように、SAS Viyaを使用することで、簡単かつ高精度にRNNによる時系列予測が可能になります。RNNをはじめとするディープラーニングを実装するには多数のハイパーパラメータを調整する必要がありますが、DLPyを用いると明示的な指定を極力省いてモデルを作成できるため、初めてディープラーニングに触れられる方にも使いやすいと言えます。一方、詳細な設定も可能であり、DLPyを用いることで特定の応用に特化した精密なモデリングも可能ですので、ディープラーニングに習熟されている方にとっても使いやすいものとなっています。

Appendix

A.　CAS Serverについて

SAS Viyaの機能を用いた分析処理はインメモリ分析エンジンであるSAS Cloud Analytic Services（CAS）サーバー上で実行されます。CASサーバーは大量データに対する複雑な分析処理を高速に実施するためのサーバー構成（SMP/MPP）が可能となっており、これによりSAS Viyaの利用者は簡単にマルチスレッド、かつインメモリの並列分散処理やGPU処理を実施することが可能となっています。

SAS Viyaで分析処理を行う際には、まず初めに「セッション」をサーバーと確立し、その「セッション」が使用するCASサーバー上のメモリに対して分析対象となるデータをロードする必要があります（但し、分析の都度同じデータをメモリにロードし直す必要はありません。一度ロードしたデータはメモリ内で保持され、特定ユーザー、あるいは複数のユーザー間で共有し、いつでも利用可能です）。

例えば、以下のコードを実行することで、ローカルにあるディレクトリ内の画像ファイル群を読み込んで得られた画像データセットをCASテーブル（インメモリのテーブル）としてロードします。

```
# コード1
import swat
import dlpy
sess = swat.CAS( host, port, user, password)
image_table = dlpy.ImageTable.load_files(
    sess,
    path='./imgs/',
    casout={'name':'my_images'})
```

※（コード中のswat、dlpyについては次節で解説します）

分析処理によって生成されたモデルに関しても、CASテーブルとしてメモリにロードすることができます。

例えば、以下のコードを実行することで、ローカルにある重みファイルを用いて画像認識モデルをCASテーブルとしてロードします。

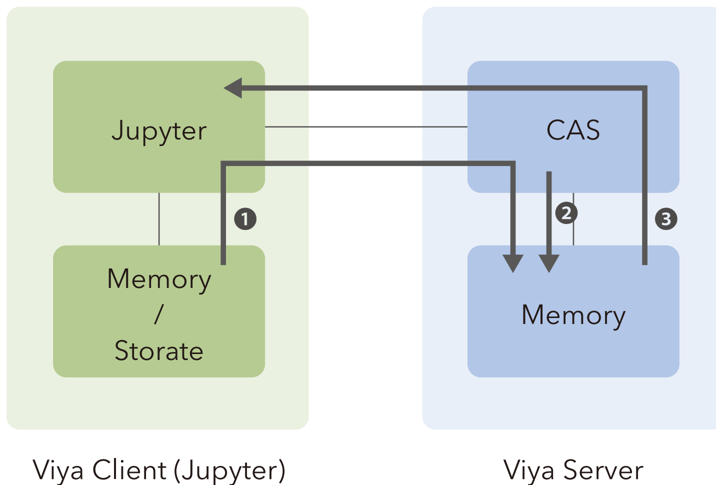
```
# コード2
from dlpy import applications
vgg16model = applications.VGG16(
    sess,
    pre_trained_weights=True,
    pre_trained_weights_file="./VGG_ILSVRC_16_layers.caffemodel.h5",
    include_top=True)
```

このようにして、CASサーバー上に用意したデータセットとモデルを利用して分析を実施し、結果を高速に取得します。

例えば、以下のコードを実行することで、画像認識（スコアリング）の結果をresultというクライアント側の変数メモリに格納します。

```
# コード3
result = vgg16model.predict(image_table)
```

以上を図で表すと、下記のようになります。



❶ ローカルにあるファイルやデータを読み込んで、CASサーバーに格納する(# コード1)
❷ CASサーバー上にモデルを構築する(# コード2)
❸ 分析を実行し、結果を取得する(# コード3)

B.　CAS ActionとPython向けAPI(SWATとDLPy)

CASへの命令は、CASサーバーが用意しているWebAPIを呼び出すことで実行します。

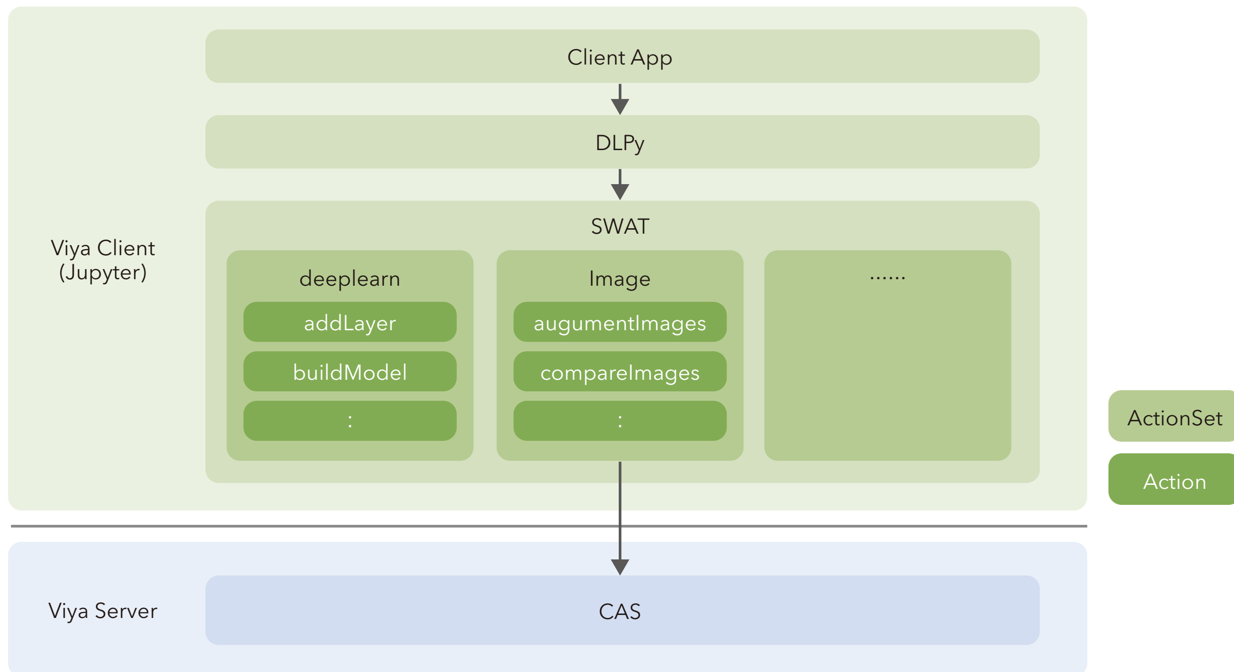
WebAPIは一般的なREST形式で提供しているため、プログラミング言語に依存せずに呼び出しを実行することが可能です。特にPython言語に対しては、このWebAPI呼び出し部分をモジュール化したライブラリとして「SWAT」(Scripting Wrapper for Analytics Transfer)を提供しており、このSWATを用いることでWebAPIを通常のPythonの関数呼び出しの要領で利用することが可能です(さらに、一部のAPIについては、バイナリ形式のAPIを利用することで通常のREST形式のAPI呼び出しよりも高速に実行されます)。

CASでは、各種の分析機能を「Action」というモジュール形式で提供しており、それぞれのActionをカテゴリー別にグループ化したものが「ActionSet」です。例えば画像処理についてはImageActionSetに、ディープラーニングについてはDeeplearnActionSetに関連する機能(Action)が含まれています。

SWATでは、必要に応じてこのActionSetをロードすることで、それぞれのActionsetが提供する機能(Action)をSWATの関数として拡張して利用することができます。

さらに、ディープラーニング用途に対しては、このSWATに対してより抽象化を行った「DLPy」というライブラリも提供しています。DLPyでは、SWATとDeeplearnActionSetにより実施するディープラーニングモデルの作成や推論を、KerasやPyTorchに近いPythonicな記法で行うことができます。また、DNNモデルのネットワーク構造のグラフ表示や各層の活性化状況の可視化、ONNX形式モデルデータのインポートなどといった、SWATが持たない機能についてもヘルパ関数としてサポートしています。

CAS、Action、ActionSet、SWAT、DLPyの関係を図示すると下記のようになります。



C. 参考情報

以下に、SAS Viyaを用いたディープラーニングモデルの開発時に参考になるサイトを紹介します。

- Getting Started with SAS Viya 3.4 for Python：
  https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=caspg3&docsetTarget=titlepage.htm&locale=en
- SWATのAPIドキュメント（執筆時バージョンは1.5.0）：
  https://sassoftware.github.io/python-swat/api.html
- SWATのソースコードリポジトリ：
  https://github.com/sassoftware/python-swat
- DLPyのAPIドキュメント（執筆時バージョンは1.0.1）：
  https://sassoftware.github.io/python-dlpy/index.html
- DLPyのソースコードリポジトリ：
  https://github.com/sassoftware/python-dlpy
- DLPyによるディープラーニング実装のサンプルノートブック：
  https://github.com/sassoftware/python-dlpy/tree/master/examples
- An Introduction to SAS Viya 3.4 Programming：
  https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=pgmdiff&docsetTarget=titlepage.htm&locale=en

- SAS Cloud Analytic Services 3.4: Fundamentals：
  http://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=casfun&docsetTarget=titlepage.htm&locale=ja

- Deep Learning Action Set のAPIドキュメント：
  http://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=casdlpg&docsetTarget=cas-deeplearn-TblOfActions.htm&locale=ja

SAS Institute Japan 株式会社　www.sas.com/jp　jpnsasinfo@sas.com

本社　〒106-6111 東京都港区六本木6-10-1 六本木ヒルズ森タワー 11F　Tel: 03 6434 3000　Fax: 03 6434 3001
大阪支店　〒530-0004 大阪市北区堂島浜1-4-16 アクア堂島西館12F　Tel: 06 6345 5700　Fax: 06 6345 5655

このカタログに記載された内容は、改良のため予告なく仕様・性能を変更する場合があります。あらかじめご了承ください。
SAS、SAS ロゴ、その他の SAS Institute Inc. の製品名・サービス名は、米国およびその他の国における SAS Institute Inc. の登録商標または商標です。
その他記載のブランド名および製品名は、それぞれの会社の商標です。Copyright © 2019, SAS Institute Inc. All rights reserved.

JP2019WP_Viya-RNN_FK