こしあん
@koshian2 著

モザイク

除去から学ぶ

最先端のディープラーニング

ICCV, CVPR, ECCVなどの国際学会の最新論文と実装
データの作成からマニアックなGANまで
Google Colaboratory+TPU(TF2.0)による演習問題
これがモザイク除去のState of The Artだ

目次

計 195 ページ

序章
DeepCreamPyで遊ぼう

DeepCreamPyをご存知だろうか？　AIを使ったモザイク除去装置としてに話題になったアプリだ。DeepCreamPyは2018年に最初に作られたが、近年バージョンアップされてさらに強力になっている。画像は配布ページより引用した。

DeepCreamPyは、このように緑色のペンで落書き（マスク）を加えても、画像を元通りに修復することができる。ここでは「復元」の処理をAIにやらせている。落書きの応用として、**モザイクのかかった画像をマスクとして認識させてあげれば、モザイク除去ができる**わけだ。なんととても夢もあるアプリだ。

すごそうなことをやっているように見えるが、この本を最後まで読めば、**DeepCreamPyのバックグラウンドの技術が理解できるようになり、あなたもDeepCreamPyのようなアプリケーションを作れるようになる。**それどころか、**モザイク除去（Inpainting）の最先端の研究を身につける**ことができる。その前に、DeepCreamPyとはどんなものか、遊んでみることにしよう。

※本書の執筆時点（2020年2月）では、DeepCreamPyはVer2.2.0となっている。

インストール

配布ページはこちらにある。

https://github.com/deeppomf/DeepCreamPy

Windowsの場合はバイナリが配布されており、以下のページから「DeepCreamPy_{version}_win64.zip」のファイルをダウンロードできる。ソースコードから実行する場合は、ローカルにPython環境が必要になる。Windowsのバイナリを例を解説する。

https://github.com/deeppomf/DeepCreamPy/releases

Windowsのバイナリを使う場合は、「Visual C++ Redistributable for Visual Studio 2015」というランタイムが必要である。以下のURLからインストールしよう。

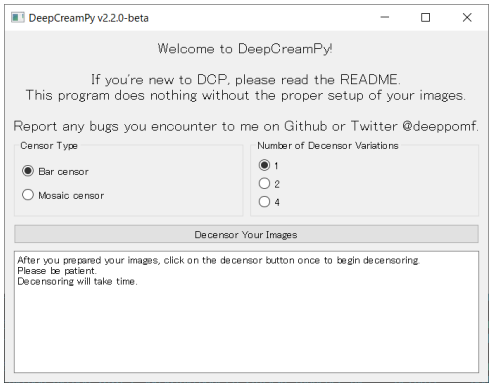
https://www.microsoft.com/en-us/download/details.aspx?id=53587

「違うバージョンのランタイムがインストールされているため、インストールに失敗しました」のようなメッセージが出た場合は、必要なライブラリが揃っているので特に問題ない。

線状のマスクを消してみよう

DeepCreamPyのマスクは緑（0, 255, 0）で描く。本当はうふふなイラストで試すと面白いのだが、大人の事情によりこの本では載せられないのがとても残念だ。DeepCreamPyにはテスト用に画像が入っている。バイナリを解凍すると、「decensor_input」というフォルダがあり、「put_me_in_decensor_input_to_test.png」というファイルがある。

最初に線の形のマスクを消してみよう。マスクは左図のように緑色の線で描く。マスクの形状は特に指定はなく、四角形、三角形、円、ランダムな形状であっても良い。とにかく緑(0, 255, 0)で指定したピクセルを全てマスクとみなす。



main.exeを実行してみよう。

ここで気にする必要があるのは**Censor Type**だ。2種類ありマスクに対する扱い方が違う。

- Bar censor: 「decensor_input」にある各画像（PNG形式である必要がある）について、緑で塗りつぶされた領域を復元する
- Mosaic censor: 「decensor_input_original」ではモザイクがかかっている画像を置き、「decensor_input」ではモザイクの領域を緑色で指定する。「decensor_input」にある各画像について、緑で塗りつぶされた領域にモザイクがかけられているとみなす。「decensor_input」の各画像と同名ファイルを「decensor_input_original」を配置し、こちらは緑に塗りつぶさない。

手動でマスクを追加したいときは、ペイントのようなツールから行えばよいが、**色は正確に緑：(0, 255, 0)を指定すること**。ペイントで編集する場合は、デフォルトのパレットにある緑色はこの色ではないので、「色の編集」などでカラーコードを指定すること。

マスクの編集が終わったらmain.exeの「Decensor Your Images」をクリックしよう。1〜2分ほどで「decensor_output」というフォルダに結果が出力される。結果は左図のようになるはずだ。

結果は左のようになる。わお、**落書きが綺麗さっぱり消えている**。これがDeepCreamPyの実力なのだ。

モザイク消しをやってみよう

さて、DeepCreamPyを使ったモザイク消しをやってみよう。本来DeepCreamPyはスケベな画像（カラーかつ、二次元のイラスト限定）を対象としている。

DeepCreamPyのモザイク消しは対象が限定されているので、すけべなゲームの、すけべなシーンがまさに打ってつけだ。もし**あなたが18歳以上**なら、サンプル画像やキャプチャ画像で試してみるのをおすすめする（Getchuにサンプル画像があるのでダウンロードしてみよう）。**あなたのやっているゲームが、機械学習の勉強素材になり、かつ無修正になる**からだ。大事なことだが、これができるのは18歳以上限定だ。

※本書ではGoogle Colaboratoryを演習問題として用いており、その関係でGoogle Driveにファイルを退避させることがあるが。もしすけべな画像をDeepCreamPyでテストしたい場合は、**Google DriveやOne Driveなどのオンラインストレージにはアップロードしないほうが良い**だろう。すけべ画像を上げると（仮に二次元でも）、Google Drive側のAIがポルノと判定し、ストレージが凍結されてしまうことがある。再現性は定かではないが、度々話を聞くので注意しよう。ローカルファイルに保存すればそういった心配は全くない。

この本は小学生でも読める健全な内容を目指しているので、18歳未満にも配慮しなくてはいけない。そう、健全な画像で試す必要がある。いらすとやを見ていたらとても健全な画像があった。

そう、とても美味しそうな、踊り焼きで食べたくなるようなアワビである。特に深い意味はない、美味しい海産物である。本書でも使うPythonという素敵な道具を使い、これにモザイクをつけてみた。

この画像を「awabi_mosaic.png」として、「decensor_input_original」のフォルダに保存する。なお、右の画像はGitHubにアップロードしてあるので、以下のURLからダウンロードできる。

https://github.com/koshian2/InpaintingUtils/blob/master/dcp_test_img/awabi_mosaic.png

DeepCreamPyでモザイク消しをするには、モザイクの領域を明示してあげないといけないので、もう1枚画像が必要である。モザイクを除去したい場合は、落書きと同じ要領で**緑色**で塗りつぶす。左図のようにする。

モザイク画像の同じファイル名「awabi_mosaic.png」とし、「decensor_input」のフォルダに保存する。モザイク消しの場合は、**モザイク画像と、緑色に塗りつぶした画像の2枚が必要**である。この画像もGitHubにアップロードしてあるので、以下のURLからダウンロードできる。

https://github.com/koshian2/InpaintingUtils/blob/master/dcp_test_img/awabi_mosaic_mask.png

さて、いよいよモザイク消しだ。DeepCreamPyは美味しそうなアワビのモザイクを消せるだろうか？　main.exeの「Censor Type」を**Mosaic censor**に切り替え、実行しよう。

結果は次のようになった。2段目のプログレスバーから数分待っても動かないケースは、ファイル名やマスクが間違っているのでもう一度確認しよう。「本物、モザイク、復元」の順にプロットしてみる。

うーん、これは健全！　モザイクを除去後のほうが、つやが出て舐めたくなるぐらい美味そうなアワビに見える。AIはここまで綺麗にモザイク除去ができる。

18歳以上の方は、Getchuなどから画像をダウンロードし、ぜひ本来の使い方（意味深）をしてみてほしい。
DeepCreamPyの本当の実力が見れるはずだ。

『DeepCreamPyで学ぶモザイク除去』との違い

さて本書と似た内容の本を、以前お買い上げいただいた方もいるかもしれない。前回と今回の両方買っていただいた方には深くお礼を申し上げたい。

『DeepCreamPyで学ぶモザイク除去』
https://note.com/koshian2/n/naa60d5c9ebba

こちらの本は筆者が2019年4月に書いた本である。本書はその約1年後の2020年2月に書いた本だ。位置づけとしては続編や増強版にあたる。前作から9割以上書き直ししており、ほぼ別の本になっている。

前回の本とは大きく次のような違いがある。

- 本書のほうが最新の研究が多く載っている。ICCV, CVPR, ECCVなどのコンピュータービジョンの世界最高レベルの学会の、最新（2019年）の研究の理論解説＋実装を行っている
- 本書の約半分をGAN（敵対的生成ネットワーク）の研究に費やしている。前回は「GANができたらいいね」程度の軽い位置づけで終わっていたが、本書はかなりGANのコアの部分に切り込んでいる。GANの安定性についても触れる予定だ。
- 前回は割と有名な話（U-Netやスタイル変換）が多かったのに対して、今回は**とにかく濃い内容**を集めた。その分難しい内容になっているので、初心者にとっては前回のほうが読みやすいかもしれない。しかし、本書の導入は機械学習がわからない人向けからはじめているので、根気のある方はついてこられることを期待している。
- 逆に前回の内容で「物足りないな」と思った方は今回は期待してOKだ。参考文献の論文だけで相当ボリュームがある。前回の本が1ヶ月程度でサクッと書いたのに対して、今回は3～4ヶ月かけてじっくりと本気出して書いた。
- 前回もGoogle Colaboratoryで実行することを想定していたが、今回はColabのNotebookを公開し、自習用の演習問題を作った。演習問題は各モデルの「エッセンス」を含んでいる。読者の理解がより深まることを期待したい。
- 前回がTensorFlow1.Xだったのに対し、今回はTensorFlow2.X。TPU周りのコードが若干簡単になっている。

前回の本と合わせて楽しんでいただければ作者冥利に尽きる。

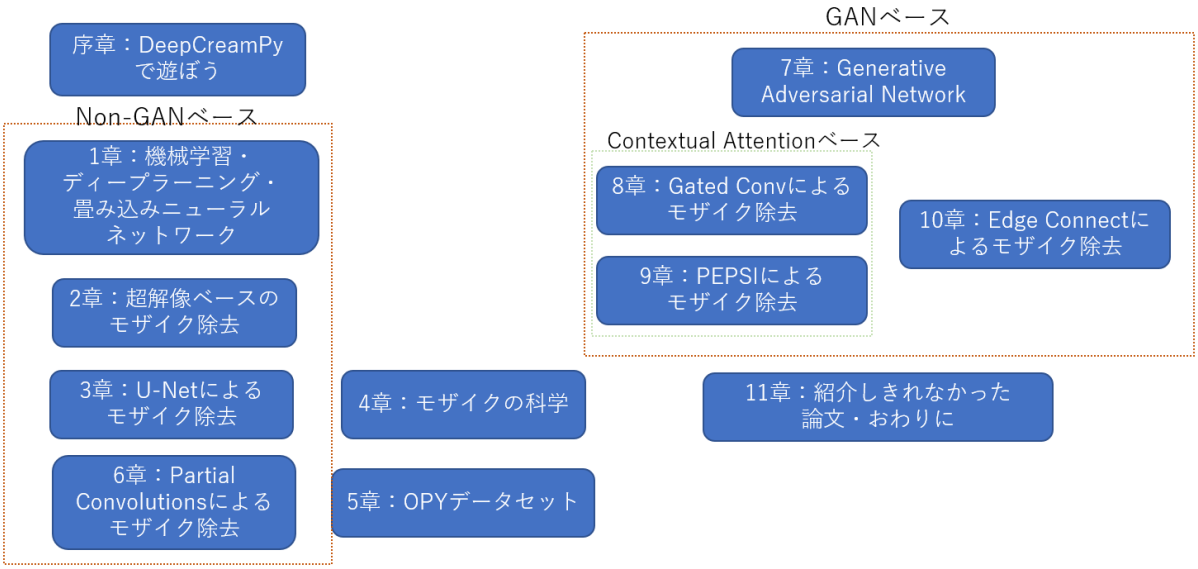
なお、**本書の電子版はAdobe Acrobat ReaderまたはGoogle Chromeで閲覧することを想定**している。Microsoft Edgeではフォントが想定した表示にならないので注意してほしい。

モザイク除去のSoTAへの道

さて、この本では「モザイク除去のSoTA（State of The Art）」にたどり着くことを期待している。DeepCreamPyのようなモザイク除去プログラムを自分で作れることを最終目標とする。DeepCreamPyは背景技術に2019年の論文を用いているので、ディープラーニングの最先端に踏み込んでいかなければならない。

でも大丈夫だ。この本では背景技術も含めしっかり誘導する。難しい内容もあるかもしれないが、ディープラーニングが初めての人も読めるように書いている。「モザイク消しなんて動機が不純だ」と思うかもしれないが、**不純だからこそ続く**のである。繰り返すが、モザイクは人類の夢なので、ディープラーニングを使ってシリコンバレーに起業するのがアメリカン・ドリームならば、ディープラーニングでギリギリなモザイクを消したり、すけべなイラストを書くのがジャパニーズ・ドリームといっても過言ではない。

本書の構成を示そう。これからわれわれが通っていく道である。



今いるのが序章「DeepCreamPyで遊ぼう」である。今どこにいるかわからなくなったときに見返すとよいだろう。これから11個の章について見ていく。中には難しいものもいくつかあるが（特に8章が大変である）、つらいなと思ったら適宜やめてもらっても構わない。楽しいからこそ速く上達するのであって、嫌いになってしまっては元も子もない。

本書を読み終わったときに、あなたはモザイク除去の最先端にいることに気づくだろう。事実ここで紹介している論文は、どれも最先端の論文である。さらに本章は背景技術として、ディープラーニング、特に畳み込みニューラルネットワークの著名な論文を多く集めており、巷に出回っている本や、動かしてみた程度ではなかなかここまでカバーできない。前回よりも更に踏み込んだ内容となっている。

あとはただ楽しんでほしい。機械学習やディープラーニングの勉強は二の次ぐらいで読むのがちょうどよいだろう。モザイク王に俺はなる！！

1章 機械学習・ディープラーニング・畳み込みニューラルネットワーク

ようこそ機械学習へ！　これから長い道のりをかけてDeepCreamPyのような、モザイク除去を実践できるようにしていく。モザイクの「State of the Art」への道は長いが、ぜひ気軽につきあっていただきたい。まずは、機械学習の世界を一から見ていこう。

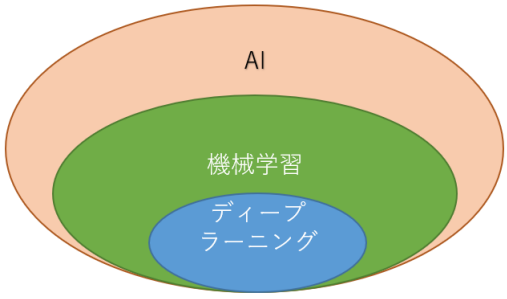
機械学習の具体例

今回、モザイク除去（DeepCreamPy）を見ていくが、これは**機械学習**の応用である。機械学習というと馴染みがないかもしれないが、身近にも機械学習のアプリケーションは多くある。以下は機械学習のアプリケーションの例だ。

- オンライン広告、ネットショッピングやWebサイトでのリコメンデーション（この商品を買った人はこんな商品も買っています）、将来の売上の予測（時系列解析）、信用リスクの評価、クレジットカードの不正利用の検出（異常検知）
- 顔認証、自動運転、物体検出、画像分類、マスク画像の補完、画像生成
- スマートスピーカー（Amazon Echo、Google Home）、AIアシスタント（Hey Siri、OK Google）、自動作曲
- 機械翻訳、チャットbot、文章から感情やキーワードを読み取る（Yahooリアルタイム検索）、自動要約

これらはほんの一例にすぎない。これらの例は俗に「AI（人工知能）」として語れることも多いだろう。AIと機械学習の違いとは何だろうか？

機械学習とAI



人工知能学会のホームページ[*1]によれば、人工知能（AI）について次のように書かれている。

> 知的な機械，特に，知的なコンピュータプログラムを作る科学と技術です．人の知能を理解するためにコンピュータを使うことと関係がありますが，自然界の生物が行っている知的手段だけに研究対象を限定することはありません．

ポイントは「知的なプログラムを作る」ということだ。例えば「Hello, world」のような、人間が全ての命令を与えて、その通りに実行させるようなプログラムは人工知能とは言わない。人間が全ての命令を与えなくても、コンピューターが考える（知的な機械）ようにさせるのが人工知能である。

2020年現在、いわゆるAIと言われるような、SF映画に出てくるような全自動な機械（例えばスターウォーズのC-3POやR2-D2）を作るというのは技術的に厳しい。R2-D2は非常に知的な機械なので、もちろん人工知能の領分であるが、そのかわりに、2020年現在では主に、**機械学習のアプリケーション＝AI**と呼んでいるのが実情である。R2-D2はできなくても、「機械学習のアプリケーションとして人工知能を実践する」というアプローチなら技術的に十分可能だ。AIより機械学習のほうが狭い概念であるが、現在AIと呼んでいるものは、手法的には機械学習であるということを覚えておこう。

機械学習の定義

では機械学習とは何だろうか。まさに冒頭に上げた例は全て機械学習のアプリケーションである。よりフォーマルな定義では、トム・ミッチェルの定義[*2]が有名である。

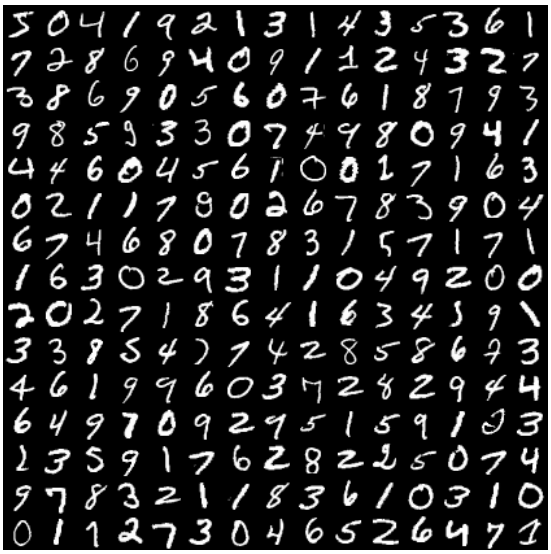
> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

トム・ミッチェルの定義では、タスクT、パフォーマンスの指標P、経験Eという3つの要素が登場する。これらは何を示すだろうか？

項目	手書き数字の認識
タスク T	手書き数字の画像から数字を認識すること
パフォーマンスの指標 P	どのくらい正しく認識できたかという精度
経験 E	手書き数字のデータ

手書き数字の認識（右の図参照。演習問題で見ていく）とは機械学習の「Hello, world」ポジションだが、**データという経験から、数字の認識というタスクに対する精度を向上させる**というものであり、機械学習の定義を満たす。冒頭で示した

機械学習の例はかなりがこちらに属するし、現在機械学習やAIと言われているアプリケーションは、手書き数字の認識のような比率が圧倒的に多い。これらは正解のデータを持つため、**教師あり学習**という。



よく「データサイエンス」という言葉を聞いたことがないのだろうか。そしてデータサイエンスには機械学習が密接に関わっている。なぜそうなのかというと、**機械学習のタスクの指標を向上するにおいてデータという「経験」が有用に活きることが多いから**である。

現在の人工知能は、ほとんどが機械学習のアプリケーションとして実践されている。そして人工知能とは、Hello worldとは異なる、機械自ら考えるような知的なプログラムである。では、そのプログラム自身の思考は、何によって規定されているのかというと**データ**である。データという経験Eにおいて、人工知能の性能が向上する。「人工知能はデータが命」ということがわかるだろう。

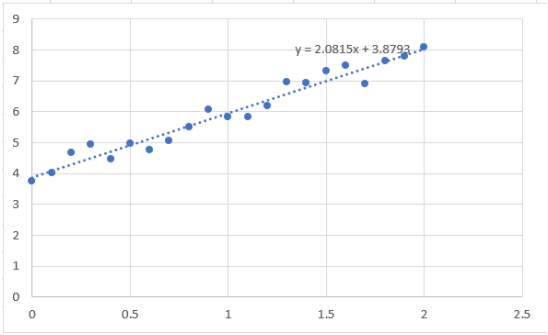
ポイント：機械学習には、タスク、パフォーマンスの指標、経験の3要素がある。現在の機械学習のアプリケーションの多くは、データを経験として活かしてパフォーマンスを向上している。

機械学習とディープラーニング：最小二乗法をこえて

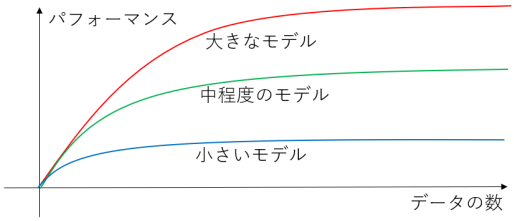
ここで、ディープラーニングという言葉について触れておきたい。**ディープラーニングとは、機械学習のジャンルの一つで、より手法的な狭い領域**である。モザイク除去アプリケーションである、DeepCreamPyは機械学習のアプリケーションでもあり、ディープラーニングの実践でもある。Deep～とつくのはディープラーニングが関係していることが多い。

ディープラーニングと機械学習の大きな違いはなんだろうか。確かにディープラーニングは機械学習の一部であるが、あたかも独立した1つの分野のように語られるのはなぜだろうか。その理由とは、ディープラーニングには、「**スケーラビリティ**」と「**表現の多様性**」という2つの大きな便利な特徴を持っているからだと思われる。

スケーラビリティ



例えば、従来の統計で出てくる最も基本的な手法で**最小二乗法**というのがある。これは統計の手法と同時に機械学習の手法でもある（機械学習の定義を満たす）。最小二乗法はとても便利で、Excelで簡単にできる。



20個ぐらいなら最小二乗法でよかった。しかし、ビッグデータの時代には最小二乗法では物足りない。例えばデータが100万個だったらどうだろうか？　これはかなり誤解されがちなのだが、$y = ax + b$という回帰モデル（最小二乗法）において、モデルが同一なら、**1000個のデータを100万個に増やしたところでほとんど精度が上がらない**。モデルの大きさとデータの数、パフォーマンスには右図のような関係がある。

最小二乗法をこの図で表すのなら、「小さいモデル」にあたるだろう。**データ数に見合ったモデルの大きさがなければ、データの多さを活かすことができない**ということに気づくだろうか？　これがディープラーニングがもてはやされる大きな理由である。ディープラーニングでは**隠れ層**（後述）の追加や削除で、モデルの大きさをかなり自在にコントロールできる。例えばこの本で使うTensorFlow/Kerasというディープラーニングのフレームワークでは、

```
import tensorflow.keras.layers as layers

input = layers.Input((784,))
x = layers.Dense(128, activation="relu")(input)
x = layers.Dense(128, activation="relu")(x) # この行をコメントアウトする
x = layers.Dense(10, activation="softmax")(x)
```

演習問題で詳しく見ていくので、いまはわからなくても良い。「この行をコメントアウトする」というところをコメントアウトするだけで、モデルを小さくすることができる。別にKerasでなく、PyTorchやchainerといった他のディープラーニングのフレームワークでも、容易にスケールを変更できる。

モデルのスケールを簡単に変更することができるのが、ディープラーニングの大きな特徴なのである。これは最小二乗法にはないし、他の機械学習のアルゴリズムでも比較的珍しいものである。

ディープラーニングの**スケーラビリティ**、なんとなくわかっただろうか。

表現の多様性

2つ目のディープラーニングの強みとして表現の多様性がある。特にこれは**損失関数やモデルの多様性**といえるだろう。損失関数が何なのかは、演習問題で詳しくみていくので、今は簡単な説明にとどめる。

損失関数とは、最適化したい目標である。例えば、最小二乗法なら「真の値と予測値の平均二乗誤差」を最適化している。式で書けば、$\min(y_{true} - y_{pred})^2$という最適化なのである。ここで、$y_{true}$は真の値、$y_{pred}$は予測値である。

最小二乗法ではこの目標が固定なのだが、この**目標を自由に設定できるのがディープラーニングの強み**なのである。損失関数は微分できる関数なら何でも良いので、予測値との3乗や4乗に設定しても構わないし、指数変換しても構わない。もっと言えば、損失関数がニューラルネットワーク（ディープラーニングで用いるネットワークモデルのこと）であっても構わないし（Style lossやPerceptual lossなど）、損失関数自体を静的な関数ではなく、動的に学習させても構わない（GAN）。

最小二乗法時代には、解を解析的に求めようとすることや、求められない場合数値計算で求めることが中心だった。ニューラルネットワークも数値計算であることには変わりないのだが、連鎖的に微分計算するという**Back propagation**という大きなブレイクスルーを導入している。イメージとしては合成関数の微分に近い。合成関数の微分では、

$$\frac{dy}{dx} = \frac{dy}{da}\frac{da}{db}\cdots\frac{dw}{dx}$$

このような連鎖律が成立した。「どんな複雑な関数でも、1個1個が微分可能なら、その合成は微分できるよね」というのが大きな主張である。逆に考えれば、「画像や音といった複雑な関数を、微分可能な単純な関数の合成で近似すれば、本物との差は微分できるよね？」ということができる。この発想がディープラーニングの根幹にある。

そもそも微分可能な関数を積み重ねればよいのだから、「本物との差」を見る損失関数も、「本物を近似するための」モデルもこれらの関数の積み重ねなのである。つまり、モデルや近似目標によって、最適化アルゴリズムを変える必要がない。微分計算できれば何でもいいという単純明快な発想と、微分可能な関数の組み合わせ的な積み重ねが、表現の多様性を許容しているということなのである。

ディープラーニングの位置づけ

「スケーラビリティ」と「表現の多様性」という2点についてディープラーニングの強みを見てきた。あくまで、ディープラーニングの機械学習の1ジャンルということに注意しよう。しかし、これらの強みによって、機械学習の中でもあたかも独立したかのような存在感を示し、冒頭で説明したような機械学習の例も、かなりがディープラーニングの実践であるぐらいに頭角を現した。

領域の広さとしては、**人工知能＞機械学習＞ディープラーニング**である。左にいくほど概念的で、右ほど手法的だ。本書における手法とは、かなりの比率がディープラーニング、特に画像処理についてのものである。「広い目線で見るとこういう位置づけなんだよ」というのを、頭の片隅に止めておいていただければ幸いである。

ポイント: ディープラーニングの強みとは、データサイズに応じたモデルのスケーラビリティや、表現の多様性などが挙げられる。

Google Colaboratory

さて、ここからしばらくPythonと行列計算のチュートリアルになる。**既にご存知なら飛ばしても構わない。**

この章では、プログラミングの演習問題を並行して解くことで、理解を進めていく形式としている。以下のURLを開いてほしい。これは**Google Colaboratory**（**Colab**）といって、ブラウザ上で実行可能な機械学習の仮想環境である。面倒な環境構築は自動的にやってくれるので、ポチポチ感覚で遊ぶことができる。**無料**で利用できる。Google Chromeで動かすのを推奨。

完全には確認していないが、AndroidスマホのChromeからも動かすことができる。タイピングがつらいので、キーボードはあったほうが良いだろう。以降はPC＋Chromeで動かすことを想定して書いていく。

Colaboratory・Pythonチュートリアル
https://colab.research.google.com/drive/1wenkuZYMdOnTo-cis0qmvN4MgfTvTxm6

このような画面が出てくると思うので、「**Google Colaboratoryで開く**」をクリックする。場合によってはリンクに飛んだ時点でColabの画面に遷移していることもある。Google Colaboratoryで開くすら表示されなかった場合は、以下のリンクからColabのトップページに飛んでから、再度上のノートブックへのリンクをクリックする。

https://colab.research.google.com/notebooks/welcome.ipynb?hl=ja

もしそれでもダメなら、Colabトップ画面から「ファイル→Python3の新しいノートブック」で一回新規ファイルを作り、ドライブと紐付けしておく（ファイルを作るだけで良い）。

ここからが重要なのだが、演習問題のノートブックを開いたら、最初に必ず「**ファイル→ドライブにコピーを保存**」をクリックすること。なぜこの操作が必要なのかというと、元のNotebookは閲覧専用なので編集ができない。PlayGroundモードでは編集ができるが、**自分が書き込んだ演習問題の答えが保存されない**。入力した答えを保存するために、ドライブにコピーしておくことが必須となる。コピーは各演習問題にアクセスした最初の1回だけでよく、2回目以降はコピーしたものを操作することになる。

ポイント： Google Colabは無料で使える。演習問題をやる前に必ずNotebookをドライブにコピーしよう。

5分で理解するPython

機械学習はPythonというプログラミング言語で行うのがメジャーである。「Colaboratory・Pythonチュートリアル」のノートブックを触りながら読んでほしい。詳しい解説はノートブックに書いてあるので、本側では一部割愛する。

まずは「Hello, world」。`print`関数を使う。機械学習はコンソール上での操作が多いので、printの出番はかなり多い。

```
print("Hello, world")
```

次に繰り返し。他の言語同様`for`ループがある。

```
for i in range(5):
    print(i)
```

これは他の言語での`for(var i=0; i<5; i++)`に相当する。

次に関数。コードを何度もコピペするぐらいなら関数を活用しよう。関数の入力を**引数**、出力を**返り値**という。例えば、これは引数を2倍して返り値を出力する関数である。

```
def function(inputs):
    return 2 * inputs
```

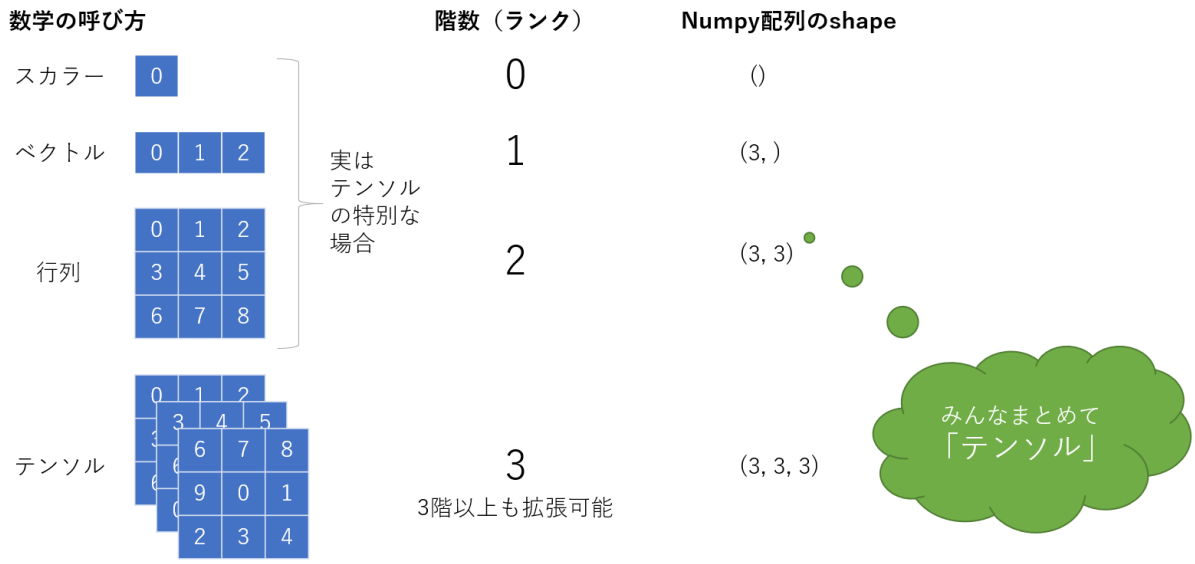
これでチュートリアルのQ1〜Q3ができるので解いてみよう。

ベクトル、行列、テンソル

ディープラーニングで避けて通れない概念として、**テンソル**がある。数学のテンソルとは複雑な概念だが、TensorFlowやPyTorchのオブジェクトとしてのテンソルは**配列変数**としての側面のほうが強い。TensorFlowやPyTorchとはディープラーニングのフレームワークだ。この本では、数学的なテンソル（前者）よりも、オブジェクトとしてのテンソル（後者）をテンソルと呼ぶことにする。

行列計算は大学の線形代数の知識を必要とするが、プログラミング的には配列変数、2次元配列である。2次元配列だろうが3次元配列だろうが、プログラミングでは初歩的な内容なので、配列変数と考えることでテンソルや行列に対する恐怖心を取り除ける。数学部分の理解に躓いて先に進めないのだったら、配列変数と捉えて実装できたほうが何百倍も幸せになれる。

スカラー、ベクトル、行列、全てテンソルの特別な場合である。以下のような関係がある。



ベクトルと行列、ベクトルとスカラーの差とは何だというと、**テンソルの階数**（rank）である。次元とは別の概念なので注意したい。例えば、

$$\begin{bmatrix}0 & 1 & 2 & 3 & 4\end{bmatrix}$$

は5次元のベクトルであり、1階のテンソルである。また、

$$\begin{bmatrix}0 & 1 & 2\\ 3 & 4 & 5\end{bmatrix}$$

は2次元×3次元の行列であり、2階のテンソルである。**次元と階数は独立した要素**で、次元がどれだけ大きくなっても1階テンソルであり、行列は常に2階テンソルである。

次元と階数の関係性は、NumpyやTensorFlow配列の`shape`プロパティを見ることで、より明瞭に理解できる。変数`X`に対し、`X.shape`を見ればよい。上のベクトル、行列に対するshapeは、

```
(5, ) # 5次元ベクトル
(2, 3) # 2x3行列
```

となる。つまり、**`shape`の具体的な値が次元で、`len(shape)`が階数**である。`len(shape)>3`なら、ベクトル・行列と区別してテンソルというが、そもそもベクトルや行列も（あまつさえスカラーも）テンソルの特別な場合なので、テンソルの次元と階数を見ればよい。これらはshapeのプロパティで見ることができる。

ポイント： ディープラーニングのテンソルは、とりあえず配列変数として考えよう。

行列積

さて、いくらTensorFlowのテンソルが多次元配列だといっても、線形代数の行列演算を使うことがしばしばある。最も多いのが行列積であろう。Aを2x3行列, Bを3x2行列とし、

$$A = \begin{bmatrix}a_{11} & a_{12} & a_{13}\\ a_{21} & a_{22} & a_{23}\end{bmatrix}$$

$$B = \begin{bmatrix}b_{11} & b_{12}\\ b_{21} & b_{22}\\ b_{31} & b_{32}\end{bmatrix}$$

とすると、行列積$C = AB$は、

$$C = \begin{bmatrix}a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32}\\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32}\end{bmatrix}$$

で表される。**(p, q)と(q, r)の行列同士の積を取ると(p, r)となる**（この公式はよく使うので覚えておくと良い）。行列積がなぜ必要になるのかというと、連立方程式を考えるのがわかりやすい。例えば、

$$\begin{cases} x + y = 3 \\ x + 2y = 5 \end{cases}$$

を行列表現すれば、

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}$$

と表せる。この行列積を展開すると連立方程式になる。行列積のメリットと一つに、**連立方程式のような複数の関数を1回の計算で同時に扱うことができるから**という点がある。

ブロードキャスト

これはPython特有の仕様であり数学的には正しくないが、**テンソルのブロードキャスト**という重要な概念がある。

例えば、次のような3次元ベクトルと、2次元ベクトルの和は、Pythonでも数学でも定義できない。

$$\begin{bmatrix} 0 & 1 & 2 \end{bmatrix} + \begin{bmatrix} 3 & 4 \end{bmatrix} = \text{undifined}$$

スカラーと3次元ベクトルの和は、Pythonでも数学でも定義できる。

$$3 + \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 4 & 5 \end{bmatrix}$$

では、1次元ベクトルと3次元ベクトルの和はどうだろうか？　これは**数学では定義できなくても、Pythonでは定義できる**（ように取り決めた）。

$$\begin{bmatrix} 3 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 4 & 5 \end{bmatrix}$$

これを**ブロードキャスト**という。Pythonでは明示することなく自動的に行われる。ブロードキャストの内部的な処理は、ここでは1次元ベクトルを3次元までコピーだ、つまり、

$$\begin{bmatrix} 3 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 3 & 3 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 4 & 5 \end{bmatrix}$$

スカラーの和と何ら変わらない。**ブロードキャストは1次元の軸のみ適用**できる。例えば、「6次元ベクトルと2次元ベクトルの和を取りたいから、2次元ベクトルを自動的に3回繰り返して」というのはできない。なぜなら、2次元の「1, 2」を「111222」と繰り返すのか「121212」と繰り返すのか判別不可能だからだ。もしこのような計算をしたい場合は明示的に値をコピーし、次元をあわせる必要がある。

ブロードキャストを使う際は、変数のshapeに注意を払う必要がある。例えば、shapeが`(1, 3) + (3, 3)`の和と、`(3, 1) + (3, 3)` の和は明確に異なる。

これは成分のコピーを考えればわかる。`(1, 3) + (3, 3)`では

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 4 & 6 & 8 \\ 7 & 9 & 11 \end{bmatrix}$$

と縦方向（axis=0の方向）にコピーされる。一方で`(3, 1) + (3, 3)` では、

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \\ 9 & 10 & 11 \end{bmatrix}$$

と横方向（axis=1の方向）にコピーされる。ここを間違えると余計なバグを生んでしまうので、常にshapeには気をつける必要がある。

また、これは筆者が心がけていることだが、**スカラーを除き階数が異なるテンソル同士の、要素間の演算をおこないようにする**という縛りをおくとよい。例えば、`(3,)`というshapeのベクトルと、`(3, 3)`というshapeの行列の和を取るなら、ベクトル側を`(1, 3)`か`(3, 1)`に変形してから和を取る（どの軸でブロードキャストされるかわからなくなるから）。これはNumpyなら、`X.reshape(1, 3)`とすればよい。

ここまでが「Colaboratory・Pythonチュートリアル」のQ4〜最後までの内容なので、演習問題を解くとさらに理解が深まるはずだ。

ポイント： ブロードキャストをうまく使いこなそう。スカラー以外で階数が異なるテンソルは、計算の前にreshapeするように心がけよう。

多層パーセプトロン

ここから3つ（うち1つは強い人向けオプション）の演習問題を解いていく。1つ目はこちら。

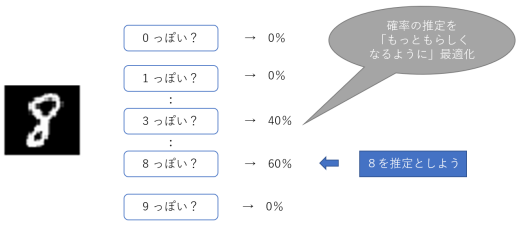
1章・演習問題（１）～ディープラーニング入門～
https://colab.research.google.com/drive/1lFEAvFrXUuRY_yeXC7AK2LlC3pPZ2tYv

こちらのノートブックも「ファイル→ドライブにコピー」してから使おう。この項では、ディープラーニングの入門として、**多層パーセプトロン**（Multi Layer Perceptron：**MLP**）の回帰問題・分類問題を行っていく。具体的な解説はノートブックにまかせてこちらでは理論的な説明をしていく。以下の内容は演習問題をやりながら読んでほしい。

まず、回帰問題・分類問題とは何だろうか？　これは予測するものが異なる。

- **回帰問題**：価格、アクセス数、売上高、大きさといった「数値」そのものを予測。予測する変数の範囲は、実数全体でも正の数でもOK。
- **分類問題**：どのクラス（犬、猫といった与えられたクラスの中のどれに）属するか、スパムメールかどうか、カメラの顔画像が本人かどうか、といった「確率」の予測。予測する変数の範囲は、確率の定義を満たすために「0以上1以下」となる。

回帰問題はまだしも、分類問題はピンとこないかもしれない。例えば手書き数字（演習問題で行っている）はこちらにあたる。



例えば、「8」という数字の画像があったとしよう。われわれは8と認識できるが、コンピューターにとってはただのピクセル値の羅列だ。ピクセル値の集合から、「0, 1, 2, ..., 9」というクラスを予測するのが分類問題である。予測するのは何というと、クラスごとの**確率**である。

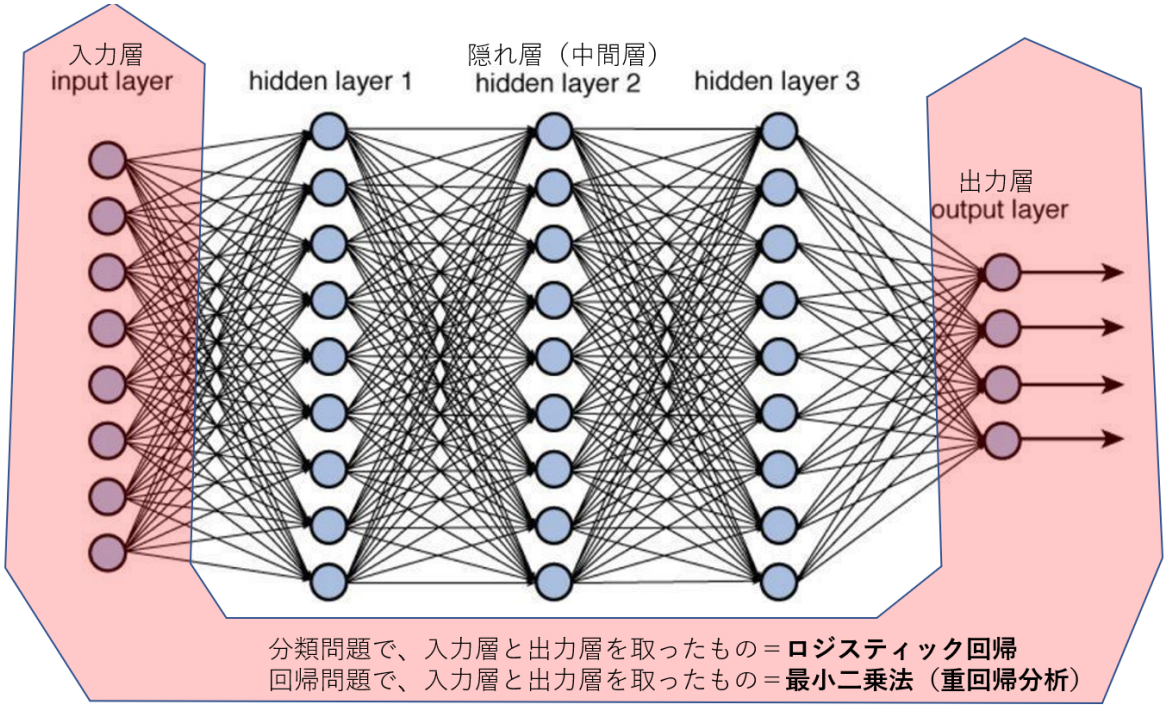
「確率の予測とはどうするのか」というのは活性化関数のところで説明するとして、まずは予測した確率からどう分類するのかを説明しよう。例えば、8という数字の画像に対して、「3である確率が40%」、「8である確率が60%」と計算されたとしよう。最も予測確率が高いクラスを予測とするのが分類問題である。クラス数をM、クラスあたりの予測確率をp_iとする。

$$\arg\max(p_1, p_2, \cdots, p_M)$$

フォーマルには「argmax」という関数で表される。コードを書く際も使うことがある（`np.argmax`）。例えば、$a_1 = 3, a_2 = 1, a_3 = 4$としよう。$\max a_i = 4$であるが、$\arg\max a_i = 3$である。

ところで、回帰問題も分類問題も統計学で古くからある予測手法がある。**前者は最小二乗法（単回帰分析、重回帰分析）、後者はロジスティック回帰**という。しかし、これらは単一モデルのスケーラビリティがなく、大きなデータに対しては性能が頭打ちになってしまう。

多層パーセプトロンはこれらの統計手法の（特にロジスティック回帰）の延長線上にある。ロジスティック回帰を1つのモデルの中で何個も積み重ねて（これを**中間層**という）、モデルのスケーラビリティをもたせたのが多層パーセプトロンだ。



図は[*3]より引用、加工した。このようにたくさんの点（ニューロンという）をつなぎ合わせて、網目状に構成したのが多層パーセプトロンだ。縦のニューロンの塊を**層（Layer）といい、最初の層を入力層**、最後の層を**出力層**という。入力層と出力層の間を**中間層**という。中間層の間で呼び方の差は特にない。

多層パーセプトロンがなぜ従来の統計手法の延長線上かというと、中間層をすべて消して入力層と出力層だけのモデルを作れば、ロジスティクス回帰や最小二乗法になるからだ。ディープラーニングでもロジスティクス回帰や最小二乗法は行うことができる。

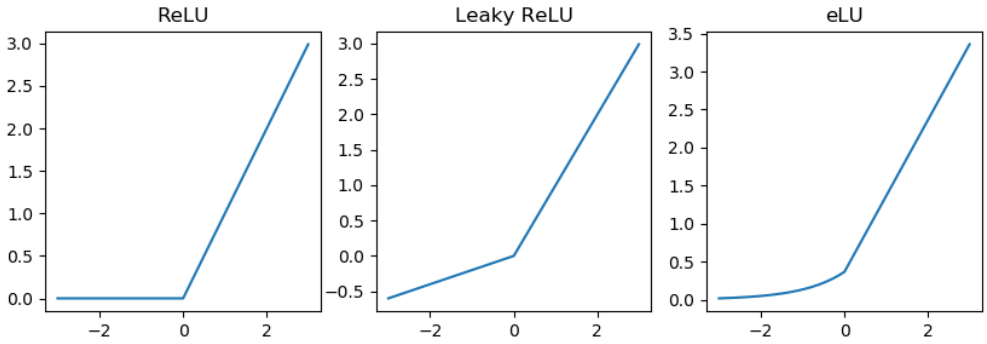
ポイント： 多層パーセプトロンとは、ロジスティック回帰や最小二乗法のすごい版。回帰問題も分類問題も解ける。

活性化関数

中間層の活性化関数

ニューラルネットワークにおいて、入力層と中間層、中間層と次の中間層、中間層と出力層の間には**活性化関数**という非線形関数を挟む。これにより、モデルのスケールが意味のあるものとなる。

逆に言えば、**非線形の活性化関数がないと、中間層をいくら積み上げてもスケーラビリティの効果はない**。なぜなら、線形の世界で$a_2(a_1x+b_1)+b_2$という計算をした場合、$a'x+b'$と表せるa', b'が必ず存在するからだ。これは中間層を100層入れても1層しかないのと同じになる。



中間層の活性化関数は、ほぼ（95%ぐらいのケースで）ReLU（Rectified Linear Units）を使う。一番左の図だ。ReLUは以下の式で表される。

$$ReLU(x) = \max(x, 0) = \begin{cases} x & (x \geq 0) \\ 0 & (x < 0) \end{cases}$$

ReLUはスイッチの役割をしている。負の数をカットしているからである。本書ではReLUの亜種としてLeakyReLU[*4]（中央の図）と、eLU[*5]（右の図）を使うことがある。一般的な出現頻度としては、LeakyReLUが5%ぐらい（GANで出てくることがある）、eLUはごく稀に使うという印象である。それぞれ以下の式である。

$$LeakyReLU(x) = \begin{cases} x & (x \geq 0) \\ \alpha x & (x < 0) \end{cases}$$

$$elu(x) = \begin{cases} x & (x \geq 0) \\ \alpha(e^x - 1) & (x < 0) \end{cases}$$

Leaky ReLUのαは0.2、eLUのαは1を使うことが多い。

出力層の活性化関数

出力層の活性化関数は特別な意味を持つ。出力層の活性化関数は回帰問題か分類問題かを特徴づけている。

出力層の代表的な活性化関数は以下の4つがある。

- **Linear（線形活性化関数・活性化関数なし）**: 回帰問題で使う関数。活性化関数を適用せず、出力の値域は$[-\infty, \infty]$。分類問題でもロジット（シグモイド関数適用前のx）で予測したいときは、活性化関数を適用しないことがある。
- **Sigmoid（シグモイド）**: 分類問題で使う関数。出力の値域は$[0, 1]$となる。値域が$[0, 1]$になるのは確率の予測に都合がいい。元ネタはロジスティック回帰。後の章で出てくる画像を出力するケースでも使うことがある（画像出力でも上限と下限があるのは都合がいい）。
- **Softmax（ソフトマックス）**: 多クラス分類問題で使う関数。シグモイド関数が2クラスの分類しかできないのに対し、ソフトマックス関数はクラス数が複数になってもクラス間の合計確率が1となることが保証される。
- **Tanh**：出力の値域は$[-1, 1]$。画像を出力とするケースや、基準化したパラメーター（例えば、おいしさ、うれしさのようにポジティブ・ネガティブあるが、物理的な定義が難しい特徴量）を回帰問題で解きたいときも使う。シグモイド関数より勾配の伝達効率が良いので、シグモイドの代替として使うこともある。
- **ReLU**: 答えが正の数となる回帰問題を解きたいときに使う。出力の値域は$[0, \infty]$。統計学の打ち切り回帰（truncated regression）の代替としても使える。

出力層の活性化関数は、出力の値域に応じて問題ごとに選ぶと良い。各活性化関数の数式を記しておく。

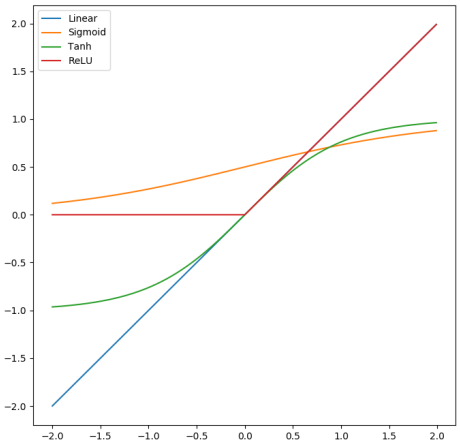
$$Linear(x) = x$$

$$Sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$Softmax(x_i) = \frac{e^{x_i}}{e^{x_1} + \cdots + e^{x_m}}$$

$$Tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$ReLU(x) = \max(x, 0)$$



シグモイド関数やソフトマックス関数がなぜ分類問題・確率の予測に使えるかは、この値域の変換にポイントがある。実数全体の入力を$[0, 1]$スケールに変換しているからである。

ポイント: 中間層の活性化関数は非線形性によるモデルのスケール、出力層の活性化関数は問題設定の決定という役割がある。出力層の活性化関数は特に重要。

損失関数・評価関数

損失関数

損失関数とは、**予測値が真の値からどの程度離れているかを示す指標で、ニューラルネットワークの最適化に非常に大きな影響**を与える。この損失関数を適切に設定することが議論の対象となることも多い。

損失関数の種類は果てしない（微分できて評価指標がよくなれば何でも損失関数として成立できる）のだが、ここでは回帰問題・分類問題用に3つほど損失関数を紹介しておく。まずは回帰問題用の損失関数。

- Mean Squared Error（平均二乗誤差、L2 loss）：$\frac{1}{n}\sum_{i=1}^{n}(y_{true}^{i} - y_{pred}^{i})^2$。これは最小二乗法と同じ。
- Mean Absolute Error（L1 loss）：$\frac{1}{n}\sum_{i=1}^{n}|y_{true}^{i} - y_{pred}^{i}|$。平均二乗誤差の絶対値版。画像を出力とするケースでよく用いられる。

二乗が良いか絶対値が良いかはケースバイケースだが、どちらも似たような傾向を示す。次は分類問題の損失関数。

- Categorical Cross Entropy（交差エントロピー）: 多クラス分類で用いられる損失関数。以下の式で表される。

$$CCE(y_{true}, y_{pred}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{true}^{i,j} \log y_{pred}^{i,j}$$

交差エントロピーの最小化は、L1やL2の最小化に比べて何をやっているのか直感的に理解しづらいが、予測分布と真の分布の差を最小化しているものと考えて良い。Kerasで使う場合は`model.compile(optimizer, "categorical_crossentropy")`とするだけで、厳密に理解しなくても使うことができる。

また、ケースによっては損失関数の引数を`sparse_categorical_crossentorpy`とすることがある。これは内部的にはCatrgorical Cross Entropyだが、真の値の入力形式が違う。真の値をクラスラベルのインデックス（$0, 1, 2, 3, \cdots$）で与えるのが`sparse_categorical_crossentropy`、真の値を確率（one-hotベクトル）で与えるのが`categorical_crossentropy`である。one-hotベクトルとは、

$$y_i = \begin{bmatrix}1 & 0 & 0\end{bmatrix}, \begin{bmatrix}0 & 1 & 0\end{bmatrix}, \begin{bmatrix}0 & 0 & 1\end{bmatrix}$$

のような表現である。`sparse_categorical_crossentropy`を使う場合は、one-hotベクトルへの変換を省略できるというメリットがある。

評価関数

評価関数は最適化（微分計算）に関係しない単なる指標である。**精度**（Accuracy）がわかりやすい。クラスインデックス`y_true, y_pred`が与えられたときに、

```
import numpy as np
accuracy = np.mean(y_true == y_pred)
```

で計算されるのが精度だ。評価関数はただ計測しているだけなのでいくつあっても良い。

ポイント: 最適化のターゲットとなるのが損失関数、ターゲットとならないただの指標が評価関数

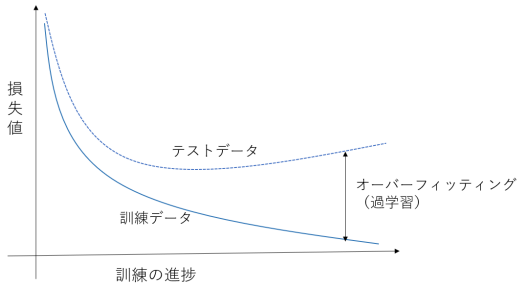
交差検証

ニューラルネットワークにとって、学習で使ったデータに対する損失を最小化することは難しくはない。しかし、あるデータに最小化されたからといって、未知のデータに対して同じ精度が出るかというのは話が別だ。**交差検証**（Cross Validation）が重要になる。

直感的にはテスト対策を考えればよい。ニューラルネットワークはとても優秀なので、10回分のテスト問題を全部暗記して答えを正確に求めることができる。しかし、10回分のテストで満点を取ることと、本番のテスト問題で満点を取ることは話が別だ。ときどき暗記していない問題に対して、どの程度点数が取れるのか確認する必要がある。これが検証/テスト（Validation/Test）である。

交差検証では、訓練の前にデータ全体をある一定比率で分割する。訓練に使う側を**訓練データ**、検証に使う側を**テストデータ、Validation（Val）データ**と言う。比率は伝統的には訓練7、テスト3が良いと言われるが、全体の母数や不均衡度によってまちまちなので、必ずしもこれが絶対というわけではない。

もう少し厳密なことを言うと、100個データがあったとして、1〜33をVal残りをTrain、33〜66をVal残りをTrain、66〜100をVal残りをTrainと3つのホールド（K-fold）に分割するのが交差検証である。先程述べたのはホールドアウト検証とよばれるもので、厳密には交差していないので交差検証とはいえないが、厳密性を要求しなければ日常的にはホールアウトも含めてCV（Cross Validation）ということもある。



一般的には、訓練データに対する損失値（**訓練損失**）、テストデータに対する損失値（**テスト損失**）に対して左図のような関係が成り立つことが多い。

これを「学習曲線（Training Curve）」という。あるところまでは訓練損失とテスト損失は同じように下がっていくが、訓練が進むほど訓練損失は下がっているのに、テスト損失が下がらなくなる（場合によっては上がっていく）という状態になる。このような状態のことを**オーバーフィッティング（過学習）**という。

過学習対策について深くは述べないが、機械学習においては過学習対策が非常に重要なポジションを占めるといっても過言ではない。ひとまず「過学習はいつでも起きるのでテストはしておこう」という発想が大事である。

ちなみに、交差検証の分割について、「テストデータで検証するな。Train：Validation：Testの3分割をしてValidationで検証をしろ」という意見もあるだろう。これはもっともであるが、3分割するか2分割でいいかは割とケースバイケースである。なぜなら、3分割におけるテストデータの意義とは、「Validationでもバイアスされていないデータに対して性能を測りたい」という理由があるからで、3分割したところでテストデータで何度も測っていてはValidationと何ら変わらなくなってしまう。画像分類の論文（例えばImageNet）での精度の報告も、Validationデータに対する精度を競うことがほとんどである。もちろん、製品の精度を自慢する際は、一切バイアスされていないテストデータに対する指標を報告することは意義ある。しかし、テストとValidationの差は曖昧で、Validationの意味でテストデータを使っても問題ないケースも多い。論文でのValidationデータに対するオーバーフィッティングの懸念は、データセットをいろいろ変えて、多くのデータセットで同様の効果が示せるかで解消しているように思える。

もう少し突っ込んで見ると、3分割ではなく4分割をするというやり方もある。これは、「訓練（Train）：TrainVal：Val：Test」という4分割である。どういうケースで必要になるのかというと、訓練データとValやTestの間に分布差がある場合。例えば、「訓練データは全世界の熊の画像データだが、やりたいタスクは日本の熊の分類タスク」という場合。日本の熊の分布と（野生ではヒグマとツキノワグマしかいない）、全世界の熊の種類の間には当然差がある。このようなケースでは、全世界の熊画像を集めるのは容易いが、日本の熊画像は少数であることが多い。そこで、ValとTestは日本の熊画像を使い、TrainとTrainValは全世界の熊画像＋日本の熊画像の残りを使う。データの分布差によるズレはTrainValとValの間で見る。分割数に限らず、**ValとTestは同一の分布にするべきである。**

この本ではTrain Testの2分割という最も簡単なケースで考える。そのため、テストをValidationの意味で使っていることがあるがご容赦いただきたい。

ここまでで「演習問題（１）〜ディープラーニング入門」の内容は全てカバーした。こちらのNotebookでは回帰問題、分類問題の両方をやっているのでぜひ試してみてほしい。

ポイント: 未知のデータに対してどれぐらい精度を出すかを求めるのが交差検証。過学習とは、訓練データに対して精度が高いが、未知のデータに対しては精度が低い状態。

画像処理の畳み込み

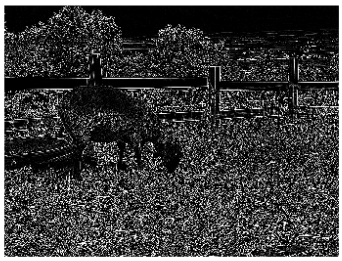
次は、こちらのNotebookの内容になる。

1章・演習問題（２）〜画像処理の畳み込みから、畳み込みニューラルネットワークへ〜
https://colab.research.google.com/drive/1vq5fvJvQlRq7ehsi8UJV8RKpYE4TiTf3

このNotebookは次の構成になっている。(1)画像の畳み込みについて見て、(2)どのようにして畳み込みニューラルネットワーク（CNN）につながっていくかを見ていく。本書ではここの理解に比較的重きをおいた。なぜなら、ここの橋渡しをきちんと解説している本は自分が記憶している限りだとないからである。「CNNはエッジ検出みたいなことやっている」といった漠然とした説明で、実際に画像分類できて「AIは画像を理解できるすごい」という解説は比較的よく見るが、これだとCNNが何をやっているのかまではわからないからである。CNNの本質的な理解につながれば幸いである。

画像処理の畳み込みとは、ディープラーニング以前からあるフィルター処理の一種である。例えば、エッジ検出とは、畳み込み処理（フィルター）の1つだ。

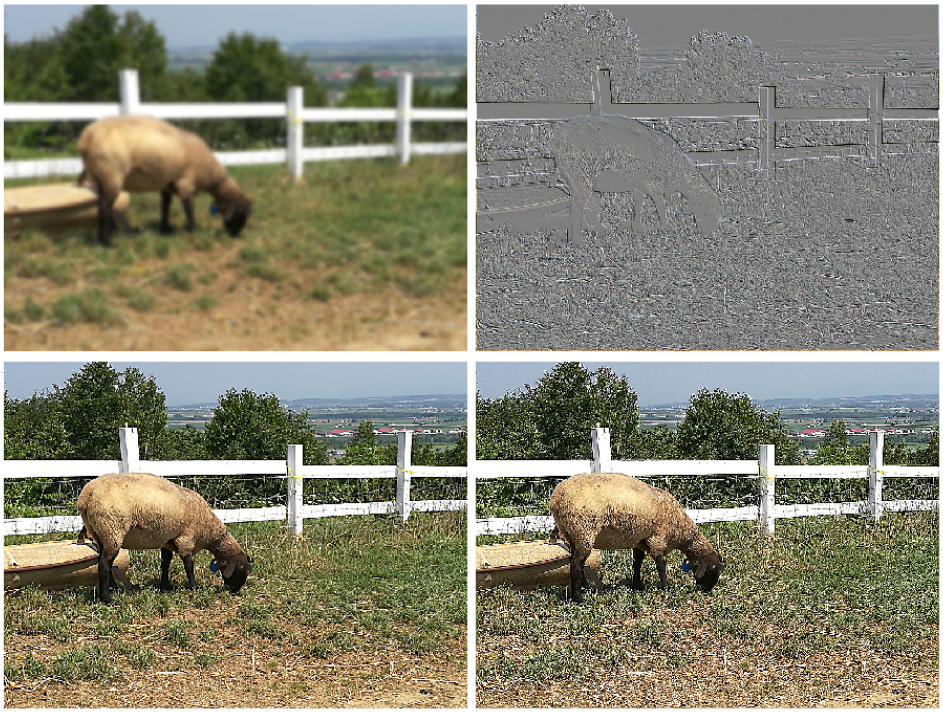




左からカラー画像、グレースケール変換した画像、エッジ画像だ。グレースケール→エッジの変換に畳み込み処理をしている。これはPILライブラリを使えば、以下のように簡単にできる。

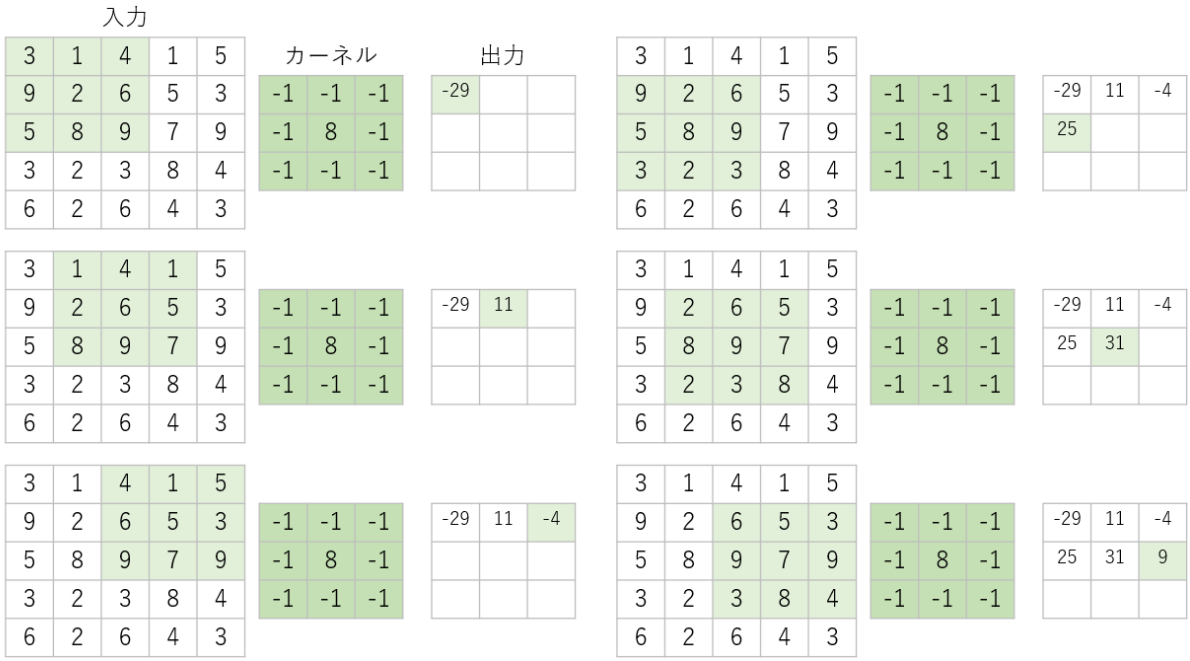
```
from PIL import Image, ImageFilter

with Image.open("image.jpg") as img:
    gray = img.convert("L") # グレースケール変換
    edge = gray.filter(ImageFilter.FIND_EDGES) # エッジ検出
```

畳み込みフィルターはエッジ検出だけではない。例えば、以下はすべて畳み込みフィルターの一例である。



左上から、ガウシアンぼかし、エンボスフィルター、アンシャープマスク、エッジ強調である。これはすべてエッジ検出と同様にImageFilterでできる。

これらはすべて「畳み込み」という計算を使っているのが、これほど出力画像が異なるのはなぜだろうか。畳み込みフィルターにおいて効果の違いを生み出しているのは、**カーネル**の数値の違いである。これがどういうことかを知るには、画像における畳み込み計算がどのようなものかを説明しよう。



上図では、話を簡略化するために入力もカーネルも行列としている。入力が5x5行列、カーネルが3x3行列としよう。カーネルとは係数だ。

畳み込みの大きなポイントとして、入力をカーネルと同じサイズのウィンドウ（**パッチ**という）に切り出して計算するという点がある。例えば、一番左上は「3, 1, 4, 9, 2, 6, 5, 8, 9」という3x3のパッチを切り出し、カーネルと要素間の積を取り、合計を求めたものをスカラーとして返す。そして図のようにパッチをスライドさせて計算を進めていく。畳み込みフィルターごとのカーネルの違いは以下の通りだ。

フィルター	ImageFilter	カーネル	オフセット
エッジ検出	FIND_EDGES	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	0
エンボスフィルター	EMBOSS	$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	0.5
エッジ強調	EDGE_ENHANCE	$0.5 \begin{bmatrix} -1 & -1 & -1 \\ -1 & 10 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	0
等高線フィルター	CONTOUR	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	1

ガウシアンぼかしは、ぼかしの半径のパラメーターがあるためカーネルは定数は表せない。アンシャープマスクも同様にパラメーターがあり、もう少し複雑な後処理をする（しかし畳み込み計算を使っている）。等高線フィルターは演習問題でやっているので確認してほしい。

畳み込み計算の説明では画像は行列としたが、実際の**入力は3階または4階のテンソル**となる。なぜなら、画像は縦解像度×横解像度×カラーチャンネルの3つの軸を持っているからである。それにバッチ（サンプル数）の軸を入れて4階テンソルとする。これは、ディープラーニングのフレームワークでは複数データを同時に扱えると都合が良いからである。

画像処理の畳み込みと、ディープラーニングの畳み込みの異なる点は、カーネルの階数である。画像処理の畳み込みのカーネルは行列（2階テンソル）だが、ディープラーニングの畳み込み（後述）では4階テンソルとなる。**ディープラーニングの畳み込みは、画像処理の畳み込みの延長線上にある。**

例えば、画像処理の畳み込み（エンボスフィルターはこの一例であるが）のカーネルは以下のような行列になる。これをKとおく。

$$K = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

カラー画像に対してどう計算するのかというと、**チャンネル単位で同一のカーネルで畳み込み計算する**。画像`image`を`(H, W, C)`の3階テンソルとすると、

```
for i in range(C):
    output[:,:,i] = convolution(image[:,:,i], K)
```

というイメージである。ここでconvolutionとは、上の図で説明したような行列×行列の畳み込みを行う関数である。

画像処理の畳み込みフィルターは、同一の計算（畳み込み）にもかかわらず、**カーネルの値を変えることで様々な効果を変えることができる**という強みがある。ディープラーニングの畳み込みニューラルネットワーク（CNN）はこの性質を活用している。CNNが学習しているのは、畳み込みフィルターのカーネルの値である。

言い方を変えれば、ディープラーニングの畳み込みとは（CNN）は、**畳み込みフィルターの選択**であり、学習を通じてタスクに応じた最適なフィルターを選んでいるということになる。ただし、画像処理の畳み込みとディープラーニングの畳み込みは少しカーネル周りの設定が違うので補足していく。

ポイント: 画像処理の畳み込みでは、同一計算でカーネルの値だけ変えることで、多様なフィルター効果を実現できる。

ディープラーニングの畳み込み

画像処理の畳み込みではカーネルは行列（2階テンソル）であったが、**ディープラーニングでの畳み込みではカーネルは4階テンソル**となる。しかし、畳み込み計算自体は変わらない。チャンネル間の演算が変わるだけである。

画像処理の畳み込みカーネルをKと表そう。Kは行列だが、あたかも定数のように考えることができる。画像処理の畳み込みをディープラーニングのカーネル（4階テンソル）は次のように表現できる。

$$\begin{bmatrix} K \\ K \\ K \end{bmatrix} \ or \ \begin{bmatrix} K & 0 & 0 \\ 0 & K & 0 \\ 0 & 0 & K \end{bmatrix}$$

左側は**Depthwise Conv**での表現、右側は**Conv2D**での表現となる。これは行列での表現（Depthwiseは列が1個の行列とみなす）だが、Kは行列なのでディープラーニングのカーネルの実体は4階テンソルである。「ある軸の行列を定数のように固定してみると行列となる」という発想を持つと、テンソルを理解しやすくなる。

Depthwise ConvもConv2Dもディープラーニングでのレイヤー（層）である。本来Depthwise ConvもConv2Dも、カーネルの行列表現の各成分がすべて同一$= K$である必要はない。つまり、**画像処理の畳み込みはConv2DやDepthwise Convに対して特別な制約をおいた場合**といえる。

Depthwise ConvとConv2Dの違いは何かというと、パッチとカーネルの積を取ったあと、チャンネル単位で加算をするかどうかの違いである。入力画像のチャンネル数C_{in}、出力画像のチャンネル数（特にConv2Dでは出力チャンネルと入力チャンネルが同一である必要性はない）をC_{out}とする。入力画像の$1 \leq i \leq C_{in}$チャンネルでの、切り出したパッチをO_i、出力画像の$1 \leq j \leq C_{out}$チャンネルにおける対応するピクセルをI_jとする。このとき、

$$\begin{bmatrix} O_1 \\ \vdots \\ O_j \\ \vdots \\ O_{C_{out}} \end{bmatrix} = \begin{bmatrix} K_{1,1}P_1 + \cdots + K_{i,1}P_i + \cdots + K_{C_{in},1}P_{C_{in}} \\ \vdots \\ K_{1,j}P_1 + \cdots + K_{i,j}P_i + \cdots + K_{C_{in},j}P_{C_{in}} \\ \vdots \\ K_{1,C_{out}}P_1 + \cdots + K_{i,C_{out}}P_i + \cdots + K_{C_{in},C_{out}}P_{C_{in}} \end{bmatrix}$$

とするのがConv2D、$C_{in} = C_{out} = C$という縛りをおいて（`tf.nn.depthwise_conv2d`ではchannel_multiplierというパラメーターで$C_{in} \neq C_{out}$とできるが、ここではchannel_multiplier=1とし、$C_{in} = C_{out}$として説明する）、

$$\begin{bmatrix} O_1 \\ \vdots \\ O_j \\ \vdots \\ O_C \end{bmatrix} = \begin{bmatrix} K_{1,1}P_1 \\ \vdots \\ K_{i,1}P_i \\ \vdots \\ K_{C,1}P_C \end{bmatrix}$$

とするのがDepthwise Convである。KPという表記は要素間の積＋和を取っているものとする。つまり、**チャンネル間の和を撮らないのがDepthwise Conv**である。$K_{C,1}$の2つ目の軸はchannel_multiplierであり、1と仮定している。

画像処理の畳み込みはDepthwise Convに対して、$K_{1,1} = \cdots = K_{i,1} = \cdots = K_{C,1} = K$という定数の縛りを置く。つまり、

$$\begin{bmatrix} O_1 \\ \vdots \\ O_j \\ \vdots \\ O_C \end{bmatrix} = \begin{bmatrix} KP_1 \\ \vdots \\ KP_i \\ \vdots \\ KP_C \end{bmatrix}$$

と簡略化したのが画像処理の畳み込みである。また、Depthwise Convのカーネルを Conv2Dのカーネルとして表現する場合は、次のように書ける。

$$\begin{bmatrix} K_{1,1} & K_{1,2} & K_{1,3} & \cdots \\ K_{2,1} & K_{2,2} & K_{2,3} & \cdots \\ K_{3,1} & K_{3,2} & K_{3,3} & \cdots \\ & \vdots & & \ddots \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & \cdots \\ 0 & 1 & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ & \vdots & & \ddots \end{bmatrix}$$

Depthwise Convのカーネルは、Conv2Dのカーネルのチャンネルの2軸に対し単位行列をかけたものと解釈することができる。ここで$A \otimes B$は要素間の積を表す。

まとめると次のような関係がある。

／	カーネルの値	チャンネル間の演算
画像処理の畳み込み	チャンネル間で同一	しない
Depthwise Conv	同一でなくても良い	しない
Conv2D	同一でなくても良い	する

Depthwise ConvはConv2Dの特別な場合であり、さらに**画像処理の畳み込みはDepthwise ConvやConv2Dの特別な場合**ということがわかっただろうか？　ここは演習問題をやっていくと理解が深まるはずなので、ぜひ楽しんでほしい。

なお、TensorFlowで画像処理の畳み込み（PILのImageFilter）に相当する処理は次のように書ける。まずはDepthwise Convの場合。`image`の変数（(N, H, W, C)の4階テンソル）に画像が格納されているものとする。

```
# Depthwise Convのケース
kernel = np.array([-1, -1, -1, -1, 8, -1, -1, -1, -1], np.float32).reshape(3, 3, 1, 1)
kernel = np.broadcast_to(kernel, (3, 3, 3, 1))
out = tf.nn.depthwise_conv2d(image, kernel, [1, 1, 1, 1], padding="SAME")
```

次はConv2Dの場合。Channel単位で計算してConcatするという方法と、単位行列のマスクを作ってかけるという2つの方法がある。

```
# Conv2D：チャンネル単位→Concat
kernel = np.array([-1, -1, -1, -1, 8, -1, -1, -1, -1], np.float32).reshape(3, 3, 1, 1)
out = [tf.nn.conv2d(image[:,:,:, i:i+1], kernel, 1, padding="SAME") for i in range(3)]
out = tf.concat(out, axis=-1)

# Conv2D : 単位行列のマスク
kernel = np.array([-1, -1, -1, -1, 8, -1, -1, -1, -1], np.float32).reshape(3, 3, 1, 1)
kernel = np.broadcast_to(kernel, (3, 3, 3, 3))
mask = np.eye(3).reshape(1, 1, 3, 3)
kernel = kernel * mask
out = tf.nn.conv2d(image, kernel, 1, padding="SAME")
```

ここで`padding="SAME"`とは、畳み込み計算における出力解像度の減少を起こさないように、畳み込み前に周囲のピクセルを埋めて調整しておく処理である。詳しくは演習問題で見ていこう。

ここまでが「1章・演習問題（２）～画像処理の畳み込みから、畳み込みニューラルネットワークへ」の内容になる。次は畳み込みニューラルネットワークを見ていく。

ポイント： Conv2Dの特別な場合がDepthwise Convであり、Depthwise Convの特別な場合が画像処理の畳み込み。Conv2Dは画像処理の畳み込みの拡張。

畳み込みニューラルネットワーク

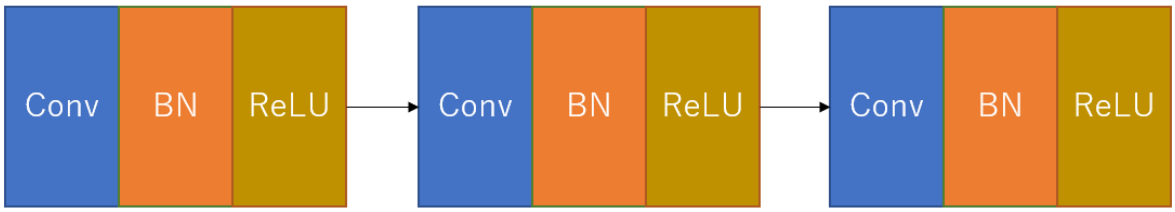
画像処理からディープラーニングの畳み込みへの橋渡しができたので、ここからは畳み込みニューラルネットワーク（Convolutional Neural Network）を見ていく。3つ目の演習問題の中の3番目にあたる。次のNotebookを開こう。

1章・演習問題（３）～畳み込みニューラルネットワーク～
https://colab.research.google.com/drive/1QImzsStP7Ba3Yfb6GAYZaEhkmByXEg06

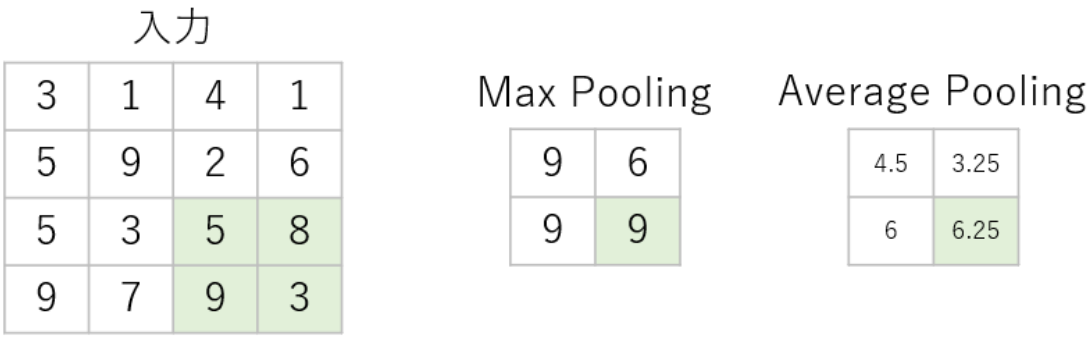
このNotebookでは、CIFAR-10のCNNによる分類を行っている。GPUで訓練している。

畳み込みニューラルネットワークでは**Conv2D**というレイヤーを中間層に使う。ニューラルネットワークには活性化関数が必要なので、多層パーセプトロンと同様に活性化関数にReLUを入れる。ただし、ConvとReLUの間に訓練を加速させるために、BatchNormalization[*6]（BatchNorm、BN）を追加する。「Conv→BatchNorm→ReLU」を1セットとすることが多い。細かいことを言えば、BatchNormはGANにおいて訓練を不安定にさせることが確認されている[*7]ので、別のレイヤーで置き換えることがある。これは後の章で見ていく。

典型的なCNNは次のような構成だ。



「Conv→BatchNorm→ReLU」のブロックを何度も繰り返していく。ただし、これでは解像度が変わらないので、ダウンサンプリングのレイヤーをはさむ。これを**Pooling層**という。

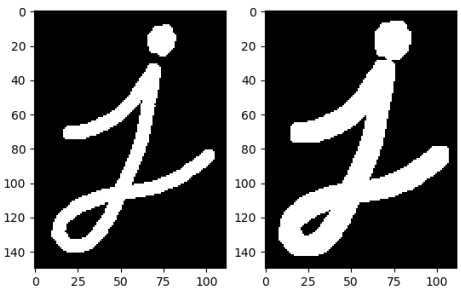


Poolingは畳み込み層と似ているが、パッチのスライドをカーネルサイズと同一にする。つまり、1マスずつずらさないで、カーネルサイズが2x2なら2マスずつずらす。ずらす幅を**stride**という。stride≠kernel_sizeなPoolingもあるが[*8]、ここでは等しいものだけを考える。stride=2の場合、出力解像度は縦横ともに入力の半分になる。

Poolingには2種類あり、**Max Pooling**と**Average Pooling**がある。Max Poolingはパッチ内の最大値を取る処理、Average Poolingはパッチ内の平均を出力する。Max Poolingはモルフォロジー変換（右図）の拡張、Average Poolingは畳み込みの特別な場合として考えられる。Average Poolingは畳み込みカーネルを、

$$\begin{bmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \end{bmatrix}$$

とし、Depthwise Convを取ったものと同じである。strideはあたかもPooling専用のパラメーターのように書いてしまったが、実はstrideはConv2DやDepthwise Convでも使うことができる。論文によっては、stride>1のConv2DをPoolingの代用としているのもある[*9]。



畳み込みレイヤーもPoolingレイヤーも、いくらダウンサンプリングを挟んでも出力は4階テンソルである。出力層では、MNIST同様クラスの分類や回帰など、2階のテンソルであったほうが都合がいいことが多い。そういったケースでは、**Global Average Pooling**といい、その時点の解像度をカーネルサイズとしたAverage Poolingを取る。例えば、その時点の解像度が16x16ならカーネルサイズは16、解像度が8x8ならカーネルサイズは8とする。これにより、出力解像度は1x1となるので、画像の縦と横の軸を潰して2階テンソルとすることができる。Numpyのreshapeの発想でPoolingを取らずに2階テンソルに変換することもある（これは**Flatten**レイヤーという）。本書ではGlobal Average Poolingを使う。Global Average Poolingについては演習問題で見ていこう。

ここまでが演習問題1-3の内容である。演習ではCIFAR-10の分類をやっているのでチャレンジしてほしい。

ポイント: CNNは畳み込みカーネルを学習するネットワークであり、畳み込みフィルターの選択である。畳み込み層の他にPooling層を使う。

入力

3	1	4	1
5	9	2	6
5	3	5	8
9	7	9	3

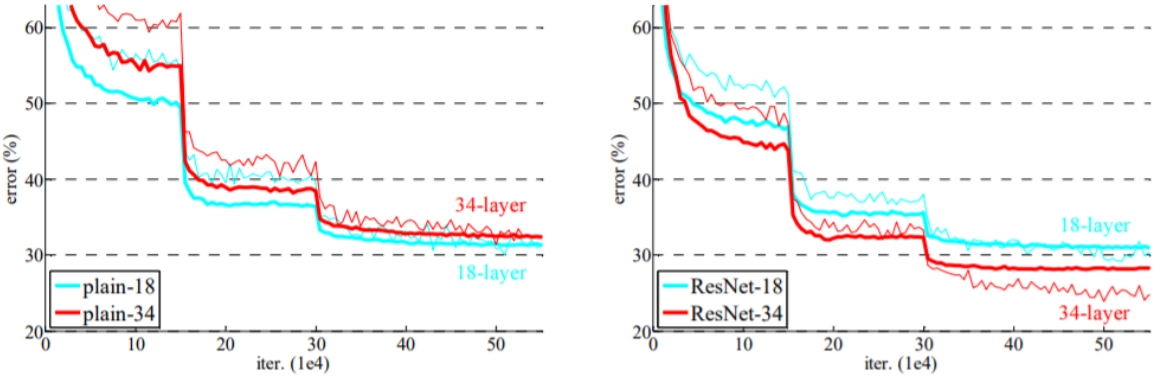
Global Average Pooling

5

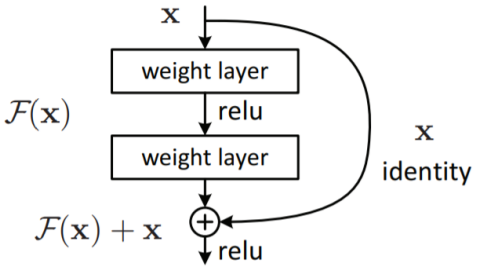
ResNet

どのようなCNNを構築をすれば分類精度が上がるのか、というのはディープラーニングにおける主要な議論対象である。個々のネットワークを見ていくとそれだけで本ができてしまうので、CNNの構造に大きな影響を与えたアイディアである**ResNet**を見ていく。この項の画像は特に断りがなければ、ResNetの論文[*9]からの引用である。

ResNetのバックの思想にあるのが、「CNNの層をどんどん増やし深くしていけば表現能力が増えて、精度が上がるでしょ」というものである。しかし、ただConv→BN→ReLUと増やしていくのでは、勾配の爆発や消失問題が起きてしまう。これがなぜ問題になるのかというと、どんどん深くしていったときにあるところから、精度とネットワークの深さが連動しなくなるからである。



縦軸がエラーレート（1-精度）、横軸が訓練の進行を表す学習曲線である。左が特になにも工夫しない場合だ。34層の畳み込み層を持つモデルより、18層の畳み込み層を持つモデルのほうが一貫して精度が高くなっている。これは困った問題で、34層のほうがモデルとしては高価なのだから、18層よりも精度が高くなってほしい。なぜ、34層のモデルは精度が出にくいのかというと、勾配の爆発や消失という別の問題が起きているからである。

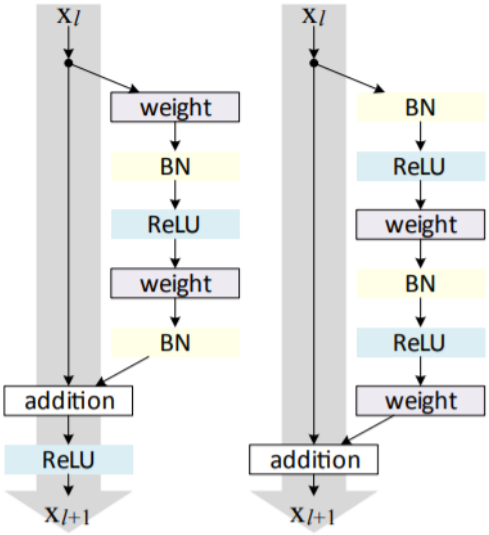


これを解消したのがResNetにおける、**Residual Block**で、学習曲線は右側のグラフにあたる。Residual Blockを導入することで、18層モデルよりも34層モデルのほうが精度が高くなり、レイヤー数と精度が連動するようになる。ではResidual Blockとはどういったものかというと、左の図のような構造だ。「Weight Layer」というのはConv2Dのような畳み込みレイヤーを指す。このように一定間隔で前の（この説明では2レイヤー前の）畳み込み出力と足すということを行う、いわばバイパスのような構造を適宜入れているのが特徴である。このバイパスの結合を、**Skip Connection**という。Residual Blockを積み重ねていったのがResNetである。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

「3x3 64, 3x3 64」x2というのは、カーネルサイズが3x3、64チャンネルのConv2Dレイヤー2個からなるResidual Blockを2個つけるという意味である。Residual Blockの数を増やすことで層を多くする（スケールする）ということを行っている。

また、50層以上のモデルではResidual Block内のConvレイヤーの数が2個から3個に増えている。これはただ単に層を増やすだけはなく、計算量の削減も兼ねている。「1x1 3x3 1x1」のように、「大きいサイズのカーネルを持つConv2Dを、小さいサイズのカーネルのConv2Dでサンドイッチする」という構造はBottleneckとよばれ、単に3x3 Convを2つ並べるよりも計算量が減る。なぜなら、畳み込み層の計算量は$K_w K_h W H C_{in} C_{out}$に比例するからである。ここで、$K_w, k_h$は縦横のカーネルサイズ、$C_{in}, C_{out}$は畳み込み層の入力チャンネル、出力チャンネル、$W, H$は入力画像の解像度である。



細かい話ではあるが、Residual Blockの構造には2種類あることに留意しておく必要がある。「Conv-BN-ReLU-Conv-BN-Add-ReLU」のようにConvが最初にくる**オリジナル**[*9]の方法（便宜上これを**post-act**と呼ぶことにする）。「BN-ReLU-Conv-BN-ReLU-Conv」のようにBNやReLUが最初にくる**pre-act**の方法。左図はResNetの後続の論文からの引用[*10]だが、左側がpost-actで、右側がpre-actである。画像の分類の場合、pre-actだろうがpost-actだろうが目に見えた差にならないことがしばしばあるが（元の論文でもCIFAR-10での改善は0.24%だった）、この**pre-act, post-actの差はGANにおいて非常に重要**になることがある。以前SNGAN[*7]を実装したときに、post-actで実装したら学習がうまくいかなかったことがあり、論文通りにpre-actで実装したらうまくいった。そのため、この本では特に断りがなければ、Residual Blockといったときはpre-actを指すことにする。

ResNetの実装は、次のTPUのNotebookで簡単にやっているので、余裕があれば確認してほしい。

ポイント： 勾配消失・爆発問題に対応するResidual Blockを導入し、ネットワークをどんどん深くすることに成功したのがResNet。Residual Blockの構成は論文によって異なるので要注意。

Colab TPUでのCNNの訓練（Keras API）

前置きの章であるにも関わらず非常に長い章となってしまったが、Colab TPUを使ったCNNの実装についても触れておきたい。TPUとはGoogleが作った機械学習用の専用デバイスで、CNNにおいては生半可なGPUよりも速いことが確認されている（ただしTPUはTensorFlow2.0では試験対応中で、場合によってはGPUのほうが速いこともある。まだ不安定な面や対応している演算が限られていたり、GPUとどっちが良いかはケースバイケースの側面が強い）。

ポイントは**Colab TPUは2020年現在、無料で利用できる**ということである。ColabのGPUは使いすぎると低性能なGPUしか割り当てられなくなったり、場合によってはGPUそのものが割り当てられなくなったりと制約が大きい。また、GPUでのディープラーニングはGPUメモリがボトルネックとなることが多いが、TPUは1個あたり64GBの大容量メモリを確保している（TPUv2）。大きめのモデルを訓練する際は、ColabでもTPUの利用を前提に考えたほうが良いケースがある。TPUというと、TF1.X時代では同時に訓練可能なモデルが1個だったり（これは2つのモデルを並行して訓練する前提のGANが大きく制約される）と、必ずしも使い勝手がよくはなかった。しかし、TF2.0によってこの制約が外れたため、TPUでもモデルの自由度が上がったという大きなメリットがある。安定性に関してはまだまだGPUに及ばないことも多いが、非常に将来性のあるデバイスといえよう。

※Colab Proの導入によって、TPUランタイムでも長時間訓練しているとランタイムが割り当てられなくなることが多くなった。特にハイメモリインスタンスの場合。2020年2月現在、Colab Proに課金（月額9.99ドル）してしまえば規制の頻度を減らすことができる。

以下のNotebookは演習問題としているが、若干解説を端折った難しめの問題なので、難しいなと思ったり、めんどくさいなと感じたらやらなくても良い。TPUでCNNを分類する内容である。

1章・演習問題（オプション）～Colab TPUによるCNNの訓練～
https://colab.research.google.com/drive/1D9jqqY-zEm7Wzns_7OAr73UATnsVK-T4

TPUの利用のためのおまじない

これはTF2.0時点のコードである。TF2.0ではTPUは試験対応であり、今後のバージョンアップではおまじないが変わる可能性がある。

まずは、TPUの初期化のためのおまじないがある。これはどのケースでも一緒なのでコピペでよい。

```
import tensorflow as tf
import os
tpu_grpc_url = "grpc://" + os.environ["COLAB_TPU_ADDR"]
tpu_cluster_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_grpc_url)
tf.config.experimental_connect_to_cluster(tpu_cluster_resolver)
tf.tpu.experimental.initialize_tpu_system(tpu_cluster_resolver)
strategy = tf.distribute.experimental.TPUStrategy(tpu_cluster_resolver)
```

最後の`strategy`という変数が重要である。このおまじないはランタイム再起動後1回実行すればよく、複数回実行するとエラーになってしまうので、おまじない専用のセルを用意しておくのが良いだろう。

Kerasによる訓練

Keras APIを使う場合はGPUとさほど差がない。敷居が低くておすすめである。

```
with strategy.scope():
    model = create_model() # モデルを作るための関数
    model.compile(...) # オプティマイザー、損失関数などを指定してコンパイル
    model.fit(...) # 訓練
```

ポイント: Keras APIでのTPU利用はGPUやCPUのケースとほとんど変わらない

Colab TPUでのCNNの訓練（カスタム訓練ループ）

モデルが1個で表されるような簡単なモデルの場合はKeras APIで十分だが、GANなど複雑なモデルを訓練する際は、Keras訓練ループを自分で書いたほうがわかりやすいケースがある。カスタム訓練ループでの、CIFAR-10での訓練コードを一気に書くと次のようになる。かなり長いので要注意。

```
import tensorflow as tf
import tensorflow.keras.layers as layers
import numpy as np
from enum import Enum
import os

# 初期化のおまじない
tpu_grpc_url = "grpc://" + os.environ["COLAB_TPU_ADDR"]
tpu_cluster_resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu_grpc_url)
tf.config.experimental_connect_to_cluster(tpu_cluster_resolver)
tf.tpu.experimental.initialize_tpu_system(tpu_cluster_resolver)
strategy = tf.distribute.experimental.TPUStrategy(tpu_cluster_resolver)

# TensorFlowのデータセットとして読み込み
def create_dataset(batch_size):
    (X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
    X_train, X_test = X_train.astype(np.float32) / 255.0, X_test.astype(np.float32) / 255.0
    y_train, y_test = y_train.astype(np.float32), y_test.astype(np.float32)
    trainset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
    trainset =
trainset.shuffle(2048).batch(batch_size).prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
    testset = tf.data.Dataset.from_tensor_slices((X_test, y_test))
    testset = testset.batch(batch_size).prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
    return trainset, testset

# モデルの作成
def create_model():
    inputs = layers.Input((32, 32, 3))
    x = inputs
    for ch in [64, 128, 256]:
        for i in range(3):
            x = layers.Conv2D(ch, 3, padding="same")(x)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
        if ch < 256:
            x = layers.AveragePooling2D(2)(x)
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dense(10, activation="softmax")(x)
    return tf.keras.models.Model(inputs, x)

# Per-replicaのReduceを楽にするためのデコレーター
class Reduction(Enum):
    NONE = 0
```

```
    SUM = 1
    MEAN = 2
    CONCAT = 3

def distributed(*reduction_flags):
    def _decorator(fun):
        def per_replica_reduction(z, flag):
            if flag == Reduction.NONE:
                return z
            elif flag == Reduction.SUM:
                return strategy.reduce(tf.distribute.ReduceOp.SUM, z, axis=None)
            elif flag == Reduction.MEAN:
                return strategy.reduce(tf.distribute.ReduceOp.MEAN, z, axis=None)
            elif flag == Reduction.CONCAT:
                z_list = strategy.experimental_local_results(z)
                return tf.concat(z_list, axis=0)
            else:
                raise NotImplementedError()

        @tf.function
        def _decorated_fun(*args, **kwargs):
            fun_result = strategy.experimental_run_v2(fun, args=args, kwargs=kwargs)
            if len(reduction_flags) == 0:
                assert fun_result is None
                return
            elif len(reduction_flags) == 1:
                assert type(fun_result) is not tuple and fun_result is not None
                return per_replica_reduction(fun_result, *reduction_flags)
            else:
                assert type(fun_result) is tuple
                return tuple((per_replica_reduction(fr, rf) for fr, rf in zip(fun_result,
reduction_flags)))
        return _decorated_fun
    return _decorator

# メイン部分
def main():
    batch_size = 512
    trainset, testset = create_dataset(batch_size)

    with strategy.scope():
        model = create_model()
        optim = tf.keras.optimizers.SGD(0.4, momentum=0.9)
        acc = tf.keras.metrics.SparseCategoricalAccuracy()
        loss_function = tf.keras.losses.SparseCategoricalCrossentropy(
            reduction=tf.keras.losses.Reduction.NONE)

        # Distributed Training用のデータセット
        trainset = strategy.experimental_distribute_dataset(trainset)
        testset = strategy.experimental_distribute_dataset(testset)

        # バッチ単位の訓練
        @distributed(Reduction.SUM)
        def train_on_batch(X, y_true):
            with tf.GradientTape() as tape:
                y_pred = model(X, training=True)
                loss = loss_function(y_true, y_pred)
                loss = tf.reduce_sum(loss, keepdims=True) * 1.0 / batch_size
            grads = tape.gradient(loss, model.trainable_weights)
            optim.apply_gradients(zip(grads, model.trainable_weights))
            acc.update_state(y_true, y_pred)
            return loss

        # バッチ単位の検証
        @distributed(Reduction.SUM)
        def validation_on_batch(X, y_true):
```

```
            y_pred = model(X, training=False)
            loss = loss_function(y_true, y_pred)
            loss = tf.reduce_sum(loss, keepdims=True) * 1.0 / batch_size
            acc.update_state(y_true, y_pred)
            return loss

        # 訓練ループ
        for epoch in range(10): # training epoch
            acc.reset_states()
            train_loss = []

            for X, y in trainset:
                train_loss.append(train_on_batch(X, y).numpy())
            train_loss = np.mean(np.array(train_loss))
            train_acc = acc.result().numpy()

            acc.reset_states()
            val_loss = []
            for X, y in testset:
                val_loss.append(validation_on_batch(X, y).numpy())
            val_loss = np.mean(np.asarray(val_loss))
            val_acc = acc.result().numpy()

            print("Epoch :", epoch+1, "Train loss :", train_loss, "Train acc :", train_acc,
                "Val loss :", val_loss, "Val acc :", val_acc)

if __name__ == "__main__":
    main()
```

ただし、Colabのデフォルトが TF1.x系である場合は、以下のようにTF2.xに切り替える必要がある。

```
%tensorflow_version 2.x
```

ただし、このコードは再起動してしまうと設定が外れてしまうので（執筆当時の時点なのでもしかしたら改善しているかもしれない）、

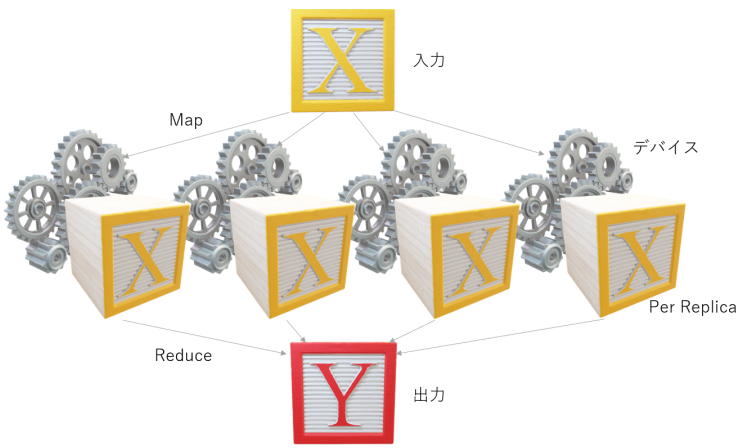
```
!pip install tensorflow==2.0.0
```

のように、TensorFlow自体を2.x系で上書きしてしまうのも良い（バージョンは適宜変更すること）。上記のコードはTensorFlow2.0での動作を確認している。個々の要素を見ていこう。

データセットの作成

Keras APIの場合、`fit(...)`にNumpy配列を渡せばよかったが、訓練ループを自分で書く場合はTensorFlowのデータセットとして定義する必要がある。これはDistributed Training対応のデータセットに変換する都合上である。

Distributed Trainingとはなにかというと、複数GPUで訓練するイメージを考えれば良い。ColabのTPUはデバイスが8個あるため、あたかもGPUが8枚あるように考えられる。GPUが2枚以上あるときどう訓練しているのかというと、**デバイス単位でデータを分割**して計算している。一見、モデル全体の計算が100あるとして、1〜25をデバイスAが、26〜50をBが……のように考えがちだが、実はデバイスAもBも1〜100までの計算はしている。分割しているのはデータであり、例えばGPUが1枚なら128個（これをグローバルバッチサイズという）のデータを全部計算するところを、GPUが4枚なら



デバイス単位で32個ずつのデータを計算する。この方式をData Parallelという。モデルが複数個ある場合は、モデル1をデバイスAが、モデル2をデバイスBのように計算するという方式もあり、これはModel Parallelという（大事なのは、モデルの中身をばらしてデバイス単位で分割計算はせずに、モデル単位で分割するか、サンプルを分割して計算するかということである）。ここではモデルが1個しかないので、Data Parallelだけを考える。

Data Parallelでは、入力$\boldsymbol{X}$をデバイス単位で分割して個々に計算する（この呼び方は公式のものではないが、分散コンピューティングと非常に似ているため、そのアナロジーからMapと呼ぶことにする）。デバイス単位にMapされたデータ$\boldsymbol{X}$の塊をReplica、もしくはPer Replicaと呼ぶ。デバイスごとにReplicaをモデルに食わせ、Replicaに対する出力$\boldsymbol{Y}$を計算する。ここで、全てのデバイス、つまりReplica単位での出力$\boldsymbol{Y}$をまとめるという作業が必要になり、これはReduce（これは公式の呼び方である）と呼ばれる。デバイスが1個のときと比べて、MapとReduceという余計な行程が必要になるため、必ずしも計算速度はデバイス数に比例しないが、ほぼそれに近い形でスケールできる。Distributed Trainingにおいて、GPUとTPUの計算フローの差をユーザー側は特に意識する必要はなく（strategyのクラスは変わるが）、どちらもMapとReduceという形で実践することになる。

TensorFlow2.0でのDistributed Trainingにおいて、Mapはこの項にでてくるデータセットの変換で、ユーザー側が特に意識することなく実装することができる。ユーザーが意識する必要が出てくるのは、Reduce側である。Reduce側は後で見ていくとしよう。

前置きが長くなってしまったが、Mapをするためにはデータセットは TensorFlowのデータセット（tf.data）である必要がある。Keras APIの場合は、全てこの変換をTensorFlow側でやってくれたので特に意識する必要はない。しかし、訓練ループを自分で書く場合は明示的に実装する必要がある。

TensorFlowのデータセットは前処理をパイプラインで書くわかりやすい仕組みになっている。以下のようにメソッドチェーンで実装するのが一般的だ。

```
trainset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
trainset =
trainset.shuffle(2048).batch(batch_size).prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
```

上から順に見ていこう

- `tf.data.Dataset.from_tensor_slices`: これはNumpy配列もしくはTensorFlowのテンソルの大きな塊を、バッチ単位で適宜切り出すようなデータセットを作りましょうということである。ただしテンソルのサイズが2GB近く/以上になる場合は、シリアライズでエラーとなるケースがある。切り出しの部分は後ほど指定する。
- `.shuffle(N)`: サンプルの中からN個を取り出し、その中でシャッフルするという処理である。もしNがデータ数なら完全なシャッフルとなる。メモリの都合上でNをデータ数未満にしてなんちゃってシャッフルというのも可能である。これはRAMサイズやCPUパワーと相談して決める。
- `.batch(N)`: バッチサイズNで切り出しなさいということ。このデータセットはDistributed Training関係なく、TPUのようにDistributed Trainingする場合でもNにグローバルバッチサイズを指定すればよい。
- `.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)` : バッチデータを事前にバッファーしておきましょうということ。データの入力がボトルネックとなるケースが多いので、訓練速度を上げるためには有効である。AUTOTUNEはメモリ（RAM）サイズから自動的にチューニングされる。

次に、Distributed Training対応のデータセットに変換しよう。これはstrategyのスコープで行う必要がある。

```
with strategy.scope():
    trainset = strategy.experimental_distribute_dataset(trainset)
```

これでDistributed Training対応のデータセットとなった。このデータセットをforループで回すと、Per-Replicaのバッチが出てくる。したがってMap-ReduceのMap部分はこれでOKで、あとは`train_on_batch`（バッチ単位での訓練関数）に渡せば良い。

ReplicaのReduce

実はReplicaのReduceというのは、現状のTensorFlowだとあまりイケていないので、以下のようなデコレーターを作るのが良い。

```
class Reduction(Enum):
    NONE = 0
    SUM = 1
    MEAN = 2
    CONCAT = 3

def distributed(*reduction_flags):
    def _decorator(fun):
```

```
        def per_replica_reduction(z, flag):
            if flag == Reduction.NONE:
                return z
            elif flag == Reduction.SUM:
                return strategy.reduce(tf.distribute.ReduceOp.SUM, z, axis=None)
            elif flag == Reduction.MEAN:
                return strategy.reduce(tf.distribute.ReduceOp.MEAN, z, axis=None)
            elif flag == Reduction.CONCAT:
                z_list = strategy.experimental_local_results(z)
                return tf.concat(z_list, axis=0)
            else:
                raise NotImplementedError()

        @tf.function
        def _decorated_fun(*args, **kwargs):
            fun_result = strategy.experimental_run_v2(fun, args=args, kwargs=kwargs)
            if len(reduction_flags) == 0:
                assert fun_result is None
                return
            elif len(reduction_flags) == 1:
                assert type(fun_result) is not tuple and fun_result is not None
                return per_replica_reduction(fun_result, *reduction_flags)
            else:
                assert type(fun_result) is tuple
                return tuple((per_replica_reduction(fr, rf) for fr, rf in zip(fun_result,
reduction_flags)))
        return _decorated_fun
    return _decorator
```

これは何を指すのかと言うと、`@tf.function`のラップ＋train_on_batchの返り値を適宜Reduceするためのデコレーターである（自分が作った）。どう使うのかと言うと、

```
@distributed(Reduction.SUM, Reduction.CONCAT)
def train_on_batch(X, y):
    # 何らかの処理
    return a, b # a, bはper replica
```

のように使う。SUMやMEAN、CONCATはReduceにおける集約方法を示し、上の例では、aという返り値に対してはSUM、bという返り値に対してはCONCATするということである。SUMとMEAN、CONCATの使い所とはどうかというと、

- `SUM/MEAN`: SUMは損失値の集約で使う。MEANは損失関数では使わないほうが良い。評価関数では細かいこと気にしなければMEANでも良いと思う。
- `CONCAT`: 画像を出力するようなケースで使う。axis=0でPer-ReplicaをConcatして出力する。

Distributed Trainingにおける損失関数

TensorFlow組み込みの損失関数を使う際は注意が必要である。デフォルトの損失関数は、全ての軸で平均を取ってしまうので出力がスカラーになってしまう。

```
tf.keras.losses.SparseCategoricalCrossentropy(reduction=tf.keras.losses.Reduction.NONE)
```

のように、バッチの軸を残すように指定する必要がある（したがって、損失値はベクトルとなるべきである）。自分で損失関数を組み場合は、

```
def loss_func(y_true, y_pred):
    # L1 loss, y_true/y_predが行列の場合
    return tf.reduce_mean(tf.abs(y_true-y_pred), axis=-1)
```

のように出力がベクトルとなるように定義すればOKである。

バッチ単位の訓練

X, yそれぞれがReplicaの集合体であるバッチを受け取り、訓練し、ロスを返すような関数を定義する。返り値のロスは別になくてもよい。

```
@distributed(Reduction.SUM)
def train_on_batch(X, y_true):
    with tf.GradientTape() as tape:
        y_pred = model(X, training=True)
        loss = loss_function(y_true, y_pred)
        loss = tf.reduce_sum(loss, keepdims=True) * 1.0 / batch_size
    grads = tape.gradient(loss, model.trainable_weights)
    optim.apply_gradients(zip(grads, model.trainable_weights))
    acc.update_state(y_true, y_pred)
    return loss
```

まずは`with tf.GradientTape() as tape`の部分。GradientTapeとはなにかというと、このスコープ以下は微分が計算できるように計算が記録される。具体的な微分計算は`tape.gradient(...)`の部分で行われる。`model(X, training=True)`は予測値（fore-prop）を計算している。予測値と真の値の差を損失関数で`loss_function(y_true, y_pred)`で計算している。TensorFlowでは、損失関数も評価関数も(y_true, y_pred)の順で統一されているので、頭の片隅に入れておくとよいだろう。ただし、この損失値はサンプル単位のベクトルなので、一回Per replicaの和を取る。その後グローバルバッチサイズで割り、レプリカ単位のロスとする。「最初から平均を取れば良いではないか？」と思うかもしれないが、例えば、サンプル数が3000、バッチサイズが128の場合、「3000 % 128 = 56」なので最後に56個分の端数が出る。もし平均の場合、56個からなるバッチに対して128個のスケールの勾配を適用してしまうため、56個の端数の寄与が高くなってしまう。もし和を取ってバッチサイズで割ればこのようなことは起きない。そのため、このような回りくどいことをしている。

次は微分計算である。`tape.gradient(loss, model.trainable_weights)`はまさに偏微分を計算している。$\partial y / \partial x$を計算したい場合、`tape.gradient(y, x)`とすればよい。

`optim.apply_gradients`はオプティマイザーに対して勾配を適用するということである。具体的に係数が更新されるのはここである。最後に、評価関数のアップデートをし損失値を返しているが、ここはなくても良い。返り値はPer-Replicaではなく、1つのテンソルとして返される。

バッチ単位でのValidation

訓練と同じである。異なるのは微分計算したり係数を更新したりしない点である。

```
@distributed(Reduction.SUM)
def validation_on_batch(X, y_true):
    y_pred = model(X, training=False)
    loss = loss_function(y_true, y_pred)
    loss = tf.reduce_sum(loss, keepdims=True) * 1.0 / batch_size
    acc.update_state(y_true, y_pred)
    return loss
```

Validationにおいて、損失値を求める必要は必ずしもなく（特にGANで画像を出力するようなケース）、別の評価関数でValidationしたい場合は損失計算は省いて良い。

訓練ループ

全てのパーツが揃ったので訓練ループ全体を示す。わかりやすいように細かいコードを端折って再掲する。

```
for epoch in range(10): # training epoch
    acc.reset_states()

    for X, y in trainset:
        train_on_batch(X, y)
    train_acc = acc.result().numpy()

    acc.reset_states()
```

```
    for X, y in testset:
        validation_on_batch(X, y)
    val_acc = acc.result().numpy()
```

非常にシンプルなコードになった。単にtrainsetとtrainsetに対して、それぞれtrain_on_batchとvalidation_on_batchを適用しているだけである。評価関数accはtrainとvalidationで同じオブジェクトを参照しているため、trainとvalの間で一旦ステータスをresetしている。

演習問題へ

ここまでTPUのカスタム訓練ループの方法を解説したので、あとは演習問題で実践してみよう。繰り返しになるが、初心者にとっては難しめの内容なので、つらいなと思ったらスルーしても構わない。

ポイント: TPU＋訓練ループは意識する項目が非常に多い

まとめ・演習問題

この章では、ディープラーニングでモザイク除去をするために以下の下準備をした。

- 機械学習、ディープラーニングのざっくりとした振り返り
- 多層パーセプトロン
- 画像の畳み込みからディープラーニングの畳み込みへ
- 畳み込みニューラルネットワーク
- Colab TPUを使ったCNNの訓練

演習問題は以下のとおりだ。まだやっていないという場合はぜひチャレンジしてほしい。

- Colaboratory・Pythonチュートリアル（初心者向け）
  https://colab.research.google.com/drive/1wenkuZYMdOnTo-cis0qmvN4MgfTvTxm6
- 演習問題１：ディープラーニング入門
  https://colab.research.google.com/drive/1lFEAvFrXUuRY_yeXC7AK2LIC3pPZ2tYv
- 演習問題２：画像処理の畳み込みから畳み込みニューラルネットワークへ
  https://colab.research.google.com/drive/1vq5fvJvQlRq7ehsi8UJV8RKpYE4TiTf3
- 演習問題３：畳み込みニューラルネットワーク
  https://colab.research.google.com/drive/1QlmzsStP7Ba3Yfb6GAYZaEhkmByXEg06
- 演習問題（オプション）：Colab TPUによるCNNの訓練 https://colab.research.google.com/drive/1D9jqqY-zEm7Wzns_7OAr73UATnsVK-T4

参考文献

1. 人工知能のFAQ | 人工知能学会　http://www.ai-gakkai.or.jp/whatsai/AIfaq.html
2. Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science(1997). p2. http://profsite.um.ac.ir/~monsefi/machine-learning/pdf/Machine-Learning-Tom-Mitchell.pdf
3. Training Deep Neural Networks | Towards Data Science https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964
4. A. L. Maas, A. Y. Hannun, A. Y. Ng. *Rectifier Nonlinearities Improve Neural Network Acoustic Models*. International Conference on Machine Learning (ICML) (2013) .
   https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf
5. D. Clevert, T. Unterthiner, S. Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. ICLR 2015. https://arxiv.org/abs/1511.07289v1
6. S. Ioffe, C. Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. ICML 2015. https://arxiv.org/abs/1502.03167
7. T. Miyato, T. Kataoka, M. Koyama, Y. Yoshida. *Spectral Normalization for Generative Adversarial Networks*. ICLR 2018. https://openreview.net/pdf?id=B1QRgziT-
8. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich. *Going Deeper with Convolutions*. CVPR 2015.
   https://static.googleusercontent.com/media/research.google.com/ja//pubs/archive/43022.pdf
9. K. He, X. Zhang, S. Ren, J. Sun. *Deep Residual Learning for Image Recognition*. CVPR 2016.
   https://arxiv.org/abs/1512.03385
10. K. He, X. Zhang, S. Ren, J. Sun. *Identity Mappings in Deep Residual Networks*. ECCV 2016.
    https://arxiv.org/abs/1603.05027

演習問題解答

Colaboratory・Pythonチュートリアル

ColaboratoryでのPython

```
```

```
## Q1
print("Hello, world")

## Q2
for i in range(5):
    print(i)

## Q3
def triple(inputs):
    return inputs**3
for i in range(5):
    print("i =", i, "/ i^3 =", triple(i))
```

Numpyチュートリアル

```
## Q4
print(np.ones(5))
print(np.zeros(5))

## Q5
print(np.arange(5))

## Q6
a, b = np.arange(5), np.arange(5)
print(a*b)

## Q7
print(np.arange(9).reshape(3,3))

## Q8
x = np.arange(9).reshape(3,3)
a = np.ones((3,3))
b = np.eye(3)
print(np.dot(x, a))
print(np.dot(x, b))

## Q9
a = np.arange(6).reshape(2,3)
b = np.arange(12).reshape(3,4)
c = np.arange(8).reshape(4,2)
print(np.dot(a, b)) # 1の答え
print(np.dot(np.dot(a, b), c)) # 2の答え
print(np.dot(b, c)) # 3の答え
print(np.dot(a, np.dot(b, c))) # 4の答え

## Q10（回答部分のみ）
a = np.arange(1, 4).reshape(3, 1)
b = np.arange(1, 4).reshape(1, 3)
```

ディープラーニング入門

最小二乗法

```
## Q1
X, y = data["data"], data["target"]
print(X.shape, y.shape)

## Q2
inputs = layers.Input((13,))
```

```
## Q3
outputs = layers.Dense(1)(inputs)

## Q4
model = tf.keras.models.Model(inputs, outputs)

## Q5
model.compile("adam", "mean_squared_error")

## Q6
model.fit(X, y, epochs=10)

## Q7
y_pred = model.predict(X)

## Q8（オプション）
inputs = layers.Input((13,))
x = layers.BatchNormalization()(inputs)
x = layers.Dense(1)(x)
model = tf.keras.models.Model(inputs, x)
model.compile("adam", "mean_squared_error")
model.fit(X, y, epochs=10)

## Q9
model.fit(X_train, y_train, validation_data=(X_train, y_train), epochs=10)
```

手書き数字の分類

```
## Q10
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()

## Q11
print(X_train[0])

## Q12（回答部分のみ）
x = layers.Dense(128, activation="relu")(x)
x = layers.Dense(10, activation="softmax")(x)
model = tf.keras.models.Model(inputs, x)

## Q13
model.compile("adam", "sparse_categorical_crossentropy", ["sparse_categorical_accuracy"])
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10)

## Q14
y_pred_prob = model.predict(X_test)
```

画像処理の畳み込みから畳み込みニューラルネットワークへ

全て回答部分のみ示す。PILでの畳み込み

```
## Q1
with Image.open("flower.jpg") as img:

## Q2
with Image.open("flower.jpg") as img:
    img = img.filter(ImageFilter.GaussianBlur())

## Q3
with Image.open("flower.jpg") as img:
    img1, img2 = img.filter(ImageFilter.EMBOSS), img.filter(ImageFilter.CONTOUR)
    img3, img4 = img.filter(ImageFilter.EDGE_ENHANCE), img.filter(ImageFilter.BLUR)
```

```
## Q4
outputs = np.zeros((row-kr+1, column-kc+1), inputs.dtype)
for i in range(outputs.shape[0]):
    for j in range(outputs.shape[1]):
        patch = inputs[i:i+kr, j:j+kc]
        prod = patch * kernel
        sum = np.sum(prod)
        outputs[i,j] = sum

## Q5
for i in [0, 1, 3, 4]:
    kernel = (np.arange(9).reshape(3,3) == i).astype(np.float32)
    print("K"+str(i+1))
    print(conv(X, kernel))

## Q6
img = img.filter(ImageFilter.Kernel((3,3), (-1,-1,-1,-1,8,-1,-1,-1,-1), 1, 255))
```

TensorFlowによる画像処理の畳み込み

```
## Q7
img = tf.io.decode_jpeg(tf.io.read_file("flower.jpg"))

## Q8
img = tf.expand_dims(img, axis=0)

## Q9
kernel = np.array([-1,-1,-1,-1,10,-1,-1,-1,-1]).reshape(3,3,1,1) / 2.0

## Q10
for i in range(3):
    conv_result = tf.nn.conv2d(float_img[:,:,:,i:i+1], kernel, 1, padding="SAME")

## Q11
emboss_kernel = np.array([-1,0,0,0,1,0,0,0,0]).reshape(3,3,1,1)
emboss_kernel = np.broadcast_to(emboss_kernel, (3,3,3,1))

# Q12
outputs = tf.nn.depthwise_conv2d(float_img, emboss_kernel, [1,1,1,1], padding="SAME") + 0.5

# Q13
contour_kernel = np.array([-1,-1,-1,-1,8,-1,-1,-1,-1]).reshape(3,3,1,1).astype(np.float32)
contour_kernel = np.broadcast_to(contour_kernel, (3,3,3,3))
mask = np.eye(3).reshape(1,1,3,3).astype(np.float32)
contour_kernel = contour_kernel * mask
outputs = tf.nn.conv2d(float_img, contour_kernel, 1, padding="SAME")outputs += 1.0
```

畳み込みニューラルネットワーク

```
## Q1
inputs = layers.Input((32,32,3))
x = layers.Conv2D(64, 3, padding="same")(inputs)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
model = tf.keras.models.Model(inputs, x)

## Q2
inputs = layers.Input((32,32,3))
x = layers.AveragePooling2D(2)(inputs)
model = tf.keras.models.Model(inputs, x)
```

```
## Q3
inputs = layers.Input((32,32,3))
x = inputs
for ch in [64, 128, 256]:
    x = layers.Conv2D(ch, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    if ch != 256:
        x = layers.AveragePooling2D(2)(x)
model = tf.keras.models.Model(inputs, x)

## Q4
x = layers.GlobalAveragePooling2D()(inputs)
x = layers.Dense(10, activation="softmax")(x)
model = tf.keras.models.Model(inputs, x)

## Q5
for ch in [64, 128, 256]:
    for i in range(3):
        x = layers.Conv2D(ch, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)
        x = layers.ReLU()(x)
    if ch != 256:
        x = layers.AveragePooling2D()(x)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(10, activation="softmax")(x)

model = tf.keras.models.Model(inputs, x)

## Q6
model.compile("adam", "sparse_categorical_crossentropy", ["sparse_categorical_accuracy"])

## Q7
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=50, batch_size=128)
```

Colab TPUによるCNNの訓練

TPUによる訓練（Keras API）。回答部分のみ示す。

```
## Q1
# Shortcut path部分のConvは1x1カーネルでもOK
x = layers.BatchNormalization()(inputs)
x = layers.ReLU()(x)
x = layers.Conv2D(ch, 3, strides=strides, padding="same",
                  kernel_regularizer=tf.keras.regularizers.l2(1e-4))(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
x = layers.Conv2D(ch, 3, padding="same", kernel_regularizer=tf.keras.regularizers.l2(1e-4))
(x)
# shortcut path
if inputs.shape[-1] != ch or strides > 1:
    s = layers.Conv2D(ch, 3, strides=strides, padding="same",
                      kernel_regularizer=tf.keras.regularizers.l2(1e-4))(inputs)

## Q2
def create_resnet():
    inputs = layers.Input((32, 32, 3))
    x = layers.Conv2D(16, 3, padding="same")(inputs)
    for ch in [16, 32, 64]:
        for i in range(7):
            strides = 2 if i == 0 else 1
            if ch == 16:
```

```
                strides = 1
            x = residual_block(x, ch, strides)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dense(10, activation="softmax")(x)

    return tf.keras.models.Model(inputs, x)

## Q3
def lr_scheduler(epoch):
    if epoch<60: return 0.1
    elif epoch<85: return 0.01
    else: return 0.001

## Q4
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()
X_train = X_train.astype(np.float32) / 255.0
X_test = X_test.astype(np.float32) / 255.0
y_train, y_test = y_train.astype(np.float32), y_test.astype(np.float32)

## Q5
with strategy.scope():
    model = create_resnet()
    model.compile(tf.keras.optimizers.SGD(0.1, momentum=0.9),
                  "sparse_categorical_crossentropy", ["sparse_categorical_accuracy"])

    scb = tf.keras.callbacks.LearningRateScheduler(lr_scheduler)
    model.fit(X_train, y_train, validation_data=(X_test, y_test), batch_size=128,
              steps_per_epoch=50000//128, epochs=100, callbacks=[scb])

## Q6
y_pred = np.argmax(model.predict(X_test), axis=-1)
```

TPUによる訓練（カスタム訓練ループ）

```
## Q7
@distributed(Reduction.SUM)
def train_on_batch(X, y):
    with tf.GradientTape() as tape:
        y_pred = model(X, training=True)
        loss = loss_function(y, y_pred)
        loss = tf.reduce_sum(loss, keepdims=True) * (1.0 / 128.0) # グローバルバッチサイズで割
る
    grads = tape.gradient(loss, model.trainable_weights)
    optim.apply_gradients(zip(grads, model.trainable_weights))
    acc.update_state(y, y_pred)
    return loss

## Q8
@distributed(Reduction.SUM)
def validation_on_batch(X, y):
    y_pred = model(X, training=False)
    loss = loss_function(y, y_pred)
    loss = tf.reduce_sum(loss, keepdims=True) * (1.0 / 128.0)
    acc.update_state(y, y_pred)
    return loss

## Q9
for X, y in trainset:
    train_loss.append(train_on_batch(X, y).numpy())
train_loss = np.mean(np.array(train_loss))
train_acc = acc.result().numpy()
```

```
acc.reset_states()
val_loss = []
for X, y in testset:
    val_loss.append(validation_on_batch(X, y).numpy())
val_loss = np.mean(np.array(val_loss))
val_acc = acc.result().numpy()

optim.lr = lr_scheduler(epoch)
```

正式版へのリンク

↓続きはこちらから↓

https://koshian2.booth.pm/items/1835219

みんな買ってね！！