

Intro

第 1 章

1.1 Deep Neural Networks の基礎

1.1.1 なぜいま AI ブームなのか

現在、第 3 期 AI ブームだといわれる。第 1 期は 1956 年に開催されたダートマス会議を契機として 1970 年前半まで続いたもので、パーセプトロンの原理が発明され、数学定理の証明や言語意味理解などが試みられた。第 2 期は 1980 年台に起こり、Multi Layer Perceptron (MLP) を効率よく学習させる Back Propagation (BP) 手法が確立されたほか、ある専門領域の知識を何らかの形で抽出し質問に答えたり問題を解いたりするエキスパートシステム研究が盛んに行われた。国内でも第五世代コンピュータの研究開発プロジェクトに莫大な資金が投入された。しかし、いずれのブームも大きな失望感とともに急速に廃れ、とある国際会議では Artificial Intelligence というキーワードが入っているだけで論文が落とされるというジョークもまことしやかにささやかれるほどであった。

では、いまなぜ改めて第 3 期 AI ブームなのだろうか？現在の AI ブームを牽引しているのは Deep Neural Networks (DNN, 深層学習) であることは間違いない。しかし、現在の DNN の源流となるネオコグニトロンは 1980 年前後にすでに (当時 NHK 放送技術研究所の) 福島ら [16, 8, 9] によって提案されている。原因は諸説あるが、下記のような複数の技術的要因がうまく噛み合ったためだと思う。

- SNS の流行、IoT 技術の普及などにより様々なドメインのデータがビッグデータ化した。
- GPU やクラウドコンピュータなど DNN を現実的な時間内で計算終了できるだけだけの計算資源が得られるようになった。
- クラウドワーキングによって正解データを作成してくれる人の確保が格段に容易になった。

特に DNN をうまく動作させようと思うと、学習のための膨大なデータが必要である。Weakly-supervised Learning (弱教師学習) や Semi-supervised Learning (半教師付き学習) など全てのトレーニングデータに正解を付与しなくともある程度学習ができるようになるアルゴリズムは存在するが、それは正解が得られない場合の回避策であり、全てのトレーニングデータに対して正解が付与されている状態が最大の性能を発揮する。半導体の講義で学習したであろうムーアの法則に則って考えると、計算機は 3 年で 4 倍の性能を得ることができるので 30 年前と比べてざっと 100 万倍の計算能力を手に入れた計算になる。これは、現在 1 日かけて行う計算は 30 年前の計算機では 100 万日かかっていたことを意味するが、30 年前から今日まで 1 万日ちょっとしか経っていない。

さらには、社会的な情勢もそれを後押ししたと指摘する研究者もいる。

- Google, Facebook, Microsoft など巨大 IT 企業が DNN を重要視し、それぞれ TensorFlow, PyTorch, The Microsoft Cognitive Toolkit (CNTK) などのディープラーニングフレームワーク (ソフトウェアライブラリ) を作成・公開した。また、国内でも PFN 社が Chainer を公開。
- オープンソース化が主流となり、GitHub など多くのプラットフォームを利用して団体や個人が作成したソフトウェアが公開されるようになった。これにより、自分で実装をする必要がなくなり、すぐに試せる、もしくは公開されている機能から先のみを実装すればよくなった。
- 論文もオープンソース化の流れにのり、多くの出版社でオープンソースとして公開去れることが増えた。また、ArXiv のように速報性を重視するサービスも登場し、その流れを一層加速させた。
- 物体認識コンテストのように、多くのコンテストが開催され、同じデータで多くの研究者が競い、性能の差異が議論しやすくなった。また、Kaggle のようにコンテスト・ランキングをビジネスにするサービスも現れた。

学生の諸君は信じられないであろうが、ほんの 10 - 20 年前は画像を読み込むにも自前で実装するところから始めなくてはならなかった。そのため、ソフトウェアの開発効率は今と比べて極めて悪かったのである。いまは ArXiv にいち早く一流国際会議に投稿された会議が掲載され (当落の保証はない)、ソースコードとデータセットが同時に公開され、それらを DL すれば一瞬で再現でき、しかも前述のフレームワークの登場で可読性が極めて高く改変がやりやすい。という時代である。Twitter やブログで一気に話題化するのも要因の一つであろう。

1.1.2 人工知能, 機械学習, 深層学習

人工知能, 機械学習, 深層学習が同列に語られることも多いが、厳密に言うと機械学習は人工知能技術の 1 つであり、深層学習は機械学習技術の 1 手法という位置づけである。特にマスコミの記事などでは受けの良さから「人工知能による処理」と謳うものが多くなりがちであるが、必ずしもそれは学習ベースの処理出ないこともあるので注意が必要である。

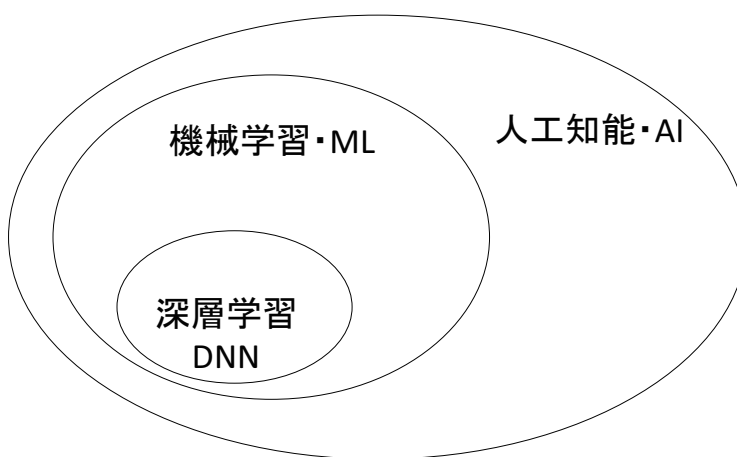


図 1.1 人工知能、機械学習、深層学習の関係。

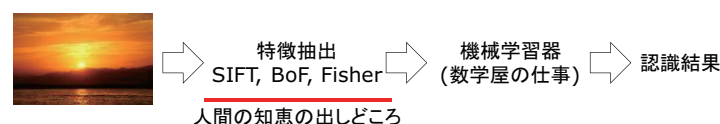
1.1.3 DNN は何が破壊的だったか？

画像認識を例に、果物の画像を分類するシステムを開発すると仮定しよう。りんごとみかんとバナナ・・・を認識しようとしたとき、自分でプログラムを書くとしたら何を手がかりとしてそれらの画像を分類するだろうか？赤ければりんご、橙色であればみかん、黄色であればバナナ・・・としてみればどうだろう。しかし、これは例えば青りんごや未完熟の緑色の果実はうまく分類することができない。それでは、大きな丸い形はりんご、小さい楕円形はみかん、長細ければバナナ・・・としてみてはどうだろう。と、このように複数のクラスをうまく切り分けることのできる画像特徴を考えるのが一般的である。DNN 以前の画像認識はまさにこの例に述べた通り画像認識に有効な新しい特徴を人間が考えるというのが画像認識の研究だった。特徴ベクトルを一旦抽出することができればあとは Support Vector Machine (SVM) や Random Forest (RF) など成熟した既存の機械学習器を用いて実際の認識・分類を行う。

DNN を用いた画像認識では、DNN に与えるのは画像そのものと正解ラベルのみである。画像を分類するための特徴は DNN が自動で考えるのである。言葉で記述するとそれだけなのであるが、人間が特徴を考えるよりも効率的な特徴設計ができる。画像分類では ILSVRC という 1000 クラスの画像を分類する国際コンテストが行われている。DNN 以前は何らかの工夫により毎年 1~2% ずつ誤差率が下がっていくというのが一般的であったが、2012 年に DNN が初めて用いられたときには人間が考えた特徴量による最高性能が誤差率 26% であったのに対し、DNN の誤差率は 15% と、実に 10% 以上もの差をつけて大勝した。その後、DNN は音声やタンパク質の結合予測コンテストでも軒並みこれまでの「人間が考える特徴量」に大差をつけて優勝し、DNN 大流行のきっかけとなった。

もちろん、DNN の登場によってやることはなくなったかということそうではない。DNN は深層学習の一般的な呼称であり、どのようなネットワークを構築するかは利用者の手に委ねられている。また、学習に関わる様々なハイパーパラメータなども人間が決定しなければならない。

◆ DNN以前



◆ DNN以後

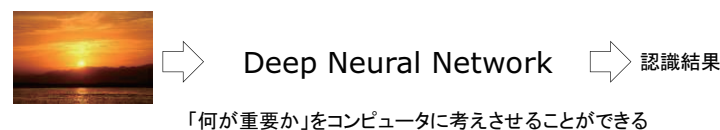


図 1.2 DNN 以前と DNN 後の画像認識手法の違い。

1.1.4 DNN の特徴

上記に述べた通り、特徴の設計も自動的にできるという利点以外に、下記にあげるような利点・特徴がある。

バッチ型ではなく逐次型の学習である

Support Vector Machine (SVM), Random Forest (RF) を始め、主要な機械学習技術の殆どがバッチ型学習、すなわち学習データはすべてそろえたあとすべてを一気に使って学習させるという手法をとる。そのほうが学習データの分布を見渡した上で最適な学習が行えるためである。SVM や RF を逐次学習、すなわち少しずつ学習データが増えていった場合に対応させる試みもなされてきたがバッチ学習と比べると精度は遠く及ばないものであった。

それに対して、NN は学習データを少量ずつ見ては内部のパラメータを更新していくという手法をとる。もちろん、初期のパーセプトロンでは学習データ 1 つごとにパラメータ更新を行っていたが、それでは BP を行った際ノイズに弱くなるので現在の DNN では複数の学習データに対する誤差を平均化して BP する手法 (ミニバッチ) など工夫はなされている。逐次学習は裏を返せば学習を途中で止めて、その後ネットワーク構造や学習データを変化させて学習を続けても良いということを示しており、この特徴が下記に続く様々な特徴へと続いていく。

Transfer learning (転移学習) が容易である

対象とするクラスの教師画像が多数無い場合が存在する。最初大量にトレーニング画像を準備できるクラスで pre-train したあと対象クラス画像で fine-tune すると高性能が出ることが知られている。例えて言えば、果物の分類器と花の分類器があったとする。果物の画像は集めるのが容易だが花の画像を集めるのは困難であるとしよう。これまでは、果物の分類器と花の分類器は独立して学習しなければならず、果物の分類器を花の分類に転用するのは困難な問題であった。そのため、花の分類性能を向上させるためには花のトレーニングデータの数を増やすしかなかった。

しかし、DNN では

- ある程度画像認識一般に有用な汎用的な特徴表現を内部表現として獲得している
- 逐次学習であるため、途中からトレーニングデータを変えても学習を進められる

などの特徴があるためである。この特徴のお陰で様々な分野へ画像分類の応用が可能となった。ただし、闇雲にデータを集めても転移学習がうまくいくわけではなく、画像の種類が大きく異なるような場合は転移学習の効果が得られにくくスクラッチ学習 (他のデータを使わないで 0 から学習) したほうが性能が高い場合もあるので注意が必要である。また、そのような問題に対して研究する Cross-Domain Learning (ドメインを跨ぐ学習) という研究領域も存在する。

ネットワークの分岐が容易である

Neural Network は文字通りネットワークであるため、途中で分岐させることが容易である。これが、DNN の性能を向上させる要因の一つとなっている。

ネットワークが途中で複数に分離する場合

途中までのネットワーク構造を共通にして、途中から複数に分離し、複数のタスクを解くネットワークを学習させることができる。これを Multi-task Learning と呼ぶ。前段の共通ネットワークは、複数のタスクからの誤差率が逆伝搬されパラメータがアップデートされる。すなわち、前段ネットワークは複数タスクを同時に解くのに効率的なネットワークになっていることが期待される。

ネットワークが途中で 1 つに集約する場合

複数のネットワークに異なるモーダルデータのデータを入力し、途中で 1 つのネットワークに統合してタスクを解くネットワークを学習させることができる。これを Multi-feature Fusion と呼ぶ。これまでの機械学習では、画像、音声、テキストなどモーダルの異なるデータを融合して総合的に学習を行うことが困難であった。しかし、このネットワーク構造では複数種類のデータをどの

ように結合すればよいかについても DNN が自動的に最適化するので無理のない Multi-feature Fusion が可能となる。

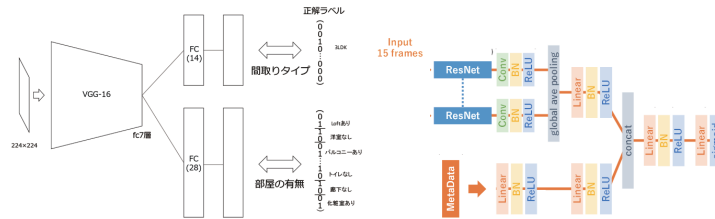


図 1.3 ネットワークの分岐。(左) Multi-task Learning、(右) Multi-feature Fusion。

データのモーダル間の垣根が少なくなった

本実験ではたった 10 日間の日程で画像・音声・言語を対象としている。一昔まえであればそれぞれ 10 日間のプログラムにしてもおかしくないほど異なる専門知識が必要とされたきた。しかし、DNN によって画像・音声・言語といったモーダルの違いは以前に比べて意識しなくても良くなってきている。それぞれのモーダルに応じてちょっとした前処理に違いは生じるものの、一旦 DNN の中に入力してしまえば、もしくは DNN 特徴量として表現してしまえば、それらのデータの扱いは全く同じになる。それを証拠に、現在 image2text, text2image といった画像と言語の融合や、「猫」という画像表現と「猫」の鳴き声の特徴空間では互いに近くなるよう学習する手法などが登場している。

1.1.5 Neural Network 基礎

山崎の講義を参照すること。

Natural Language Processing

第 2 章

本課題では、ニューラルネットワークを利用した自然言語処理の中核的な技術である、sequence-to-sequence モデル（以下、seq2seq モデル）の基本的な原理と動作を学ぶ。Seq2seq モデルは、単語列や文字列といった「系列」を入力し、新たな系列を出力することのできるニューラルネットワークモデルであり、機械翻訳や質問応答、対話といった様々な言語処理タスクへの応用が急速に進んでいる。なかでも機械翻訳に関しては、近年の翻訳精度の向上に大きく貢献し、従来の統計的機械翻訳よりも格段に流暢な翻訳を可能にしている [7]。

2.1 リカレントニューラルネットワーク

自然言語処理では、文字列や単語列といった「系列」を扱う処理が多い。しかし、通常のフィードフォワードニューラルネットワークでは、入力と出力の次元数は固定されており、任意の長さを持つ系列を扱う処理には適さない。それに対して、リカレントニューラルネットワーク (recurrent neural network, RNN) では、ネットワークに「状態」を持たせることによって、任意の長さの系列を扱うことを可能にしている。

図 2.1 にリカレントニューラルネットワークの構造を示す。図 2.1(a) に示すように、リカレントニューラルネットワークでは、ネットワーク自身が状態 $\mathbf{h}_t \in \mathbb{R}^H$ を持つことによって、その時点での入力 $\mathbf{x}_t \in \mathbb{R}^D$ だけではなく、過去の入力 $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{t-1}$ にも依存するような値 $\mathbf{y}_t \in \mathbb{R}^V$ を出力することができる。単純なリカレントニューラルネットワークでは、状態 \mathbf{h}_t は、

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h \mathbf{h}_{t-1} + \mathbf{b}_h) \quad (2.1)$$

と計算される。ここで、 $\mathbf{W}_h \in \mathbb{R}^{H \times D}$ 、 $\mathbf{U}_h \in \mathbb{R}^{H \times H}$ 、 $\mathbf{b}_h \in \mathbb{R}^D$ は、このリカレントニューラルネットワークの動作を決めるパラメータである。

リカレントニューラルネットワークは、一見普通のフィードフォワードニューラルネットワー

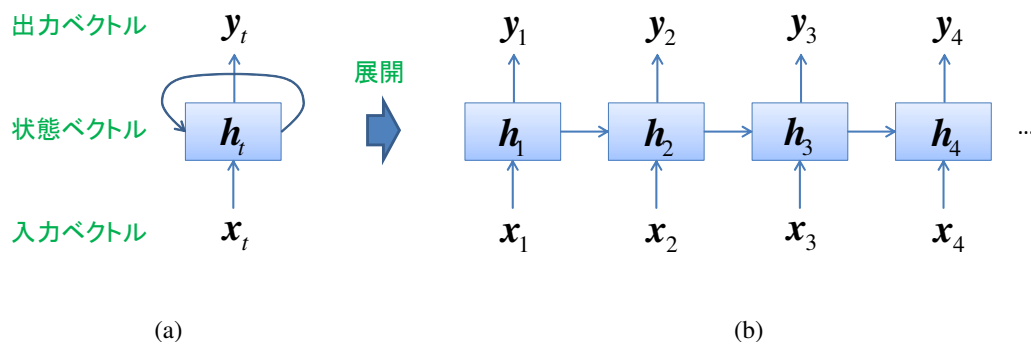


図 2.1 リカレントニューラルネットワーク

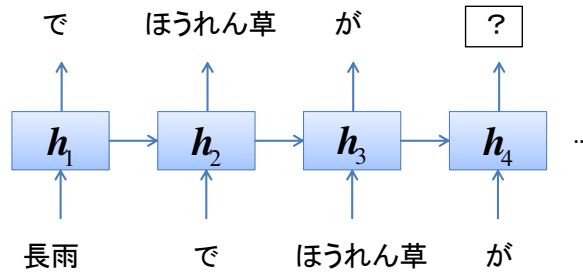


図 2.2 言語モデル

クと大きく異なる仕組みのように見えるが、実は本質的には通常のフィードフォワードニューラルネットワークと同じものである。図 2.1(a) のリカレントニューラルネットワークを時間方向に展開して考えると、図 2.1(b) に示すように、パラメータ \mathbf{W}_h , \mathbf{U}_h , \mathbf{b}_h を共有した多数のフィードフォワードネットワークを連結したものと等価であることがわかる。したがって、パラメータの学習も通常のフィードフォワードニューラルネットワーク同様、誤差逆伝播法 (backpropagation) によって効率的に行うことができる。

2.2 言語モデル

リカレントニューラルネットワークを利用すると言語モデル (language model) を容易に実現することができる [12]。言語モデルでは、単語列 w_1, w_2, \dots, w_n の生成を以下のような確率モデルで表現する。

$$P(w_1, w_2, \dots, w_n) = \prod_{t=1}^n P(w_t | w_1, \dots, w_{t-1}) \quad (2.2)$$

ここで図 2.2 のように、リカレントニューラルネットワークの入力として、単語列 w_1, w_2, \dots, w_{t-1} を考え、出力として、単語列 w_2, w_3, \dots, w_t を考えれば、文脈 w_1, \dots, w_{t-1} から、次に出現するであろう単語 w_t を予測するモデル $P(w_t | w_1, \dots, w_{t-1})$ が実現できることになる。 $P(w_t | w_1, \dots, w_{t-1})$ は、可能性のあるすべての単語に関して和をとると 1 になる、すなわち単語の語彙に関する確率分布となっている必要あるため、出力 \mathbf{y}_t は、ソフトマックス関数

$$\text{softmax}(\mathbf{a}) : \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_V \end{bmatrix} \rightarrow \frac{1}{\sum_{i=1}^V \exp(a_i)} \begin{bmatrix} \exp(a_1) \\ \exp(a_2) \\ \vdots \\ \exp(a_V) \end{bmatrix} \quad (2.3)$$

を用いて

$$\mathbf{y}_t = \text{softmax}(\mathbf{U}_y \mathbf{h}_t + \mathbf{b}_y) \quad (2.4)$$

と計算すればよい。ここで、 $\mathbf{U}_y \in \mathbb{R}^{V \times H}$ と $\mathbf{b}_y \in \mathbb{R}^V$ は、このリカレントニューラルネットワークの出力動作を規定するパラメータである。

パラメータの学習は、学習データ (コーパス) でのモデルの対数尤度

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N \log P_{\boldsymbol{\theta}}(w_i | w_1, \dots, w_{i-1}) \quad (2.5)$$

を最大化するように行う。ただし、 N は学習コーパスの単語数、 $\boldsymbol{\theta}$ はリカレントニューラルネットワークのすべてのパラメータを表す。

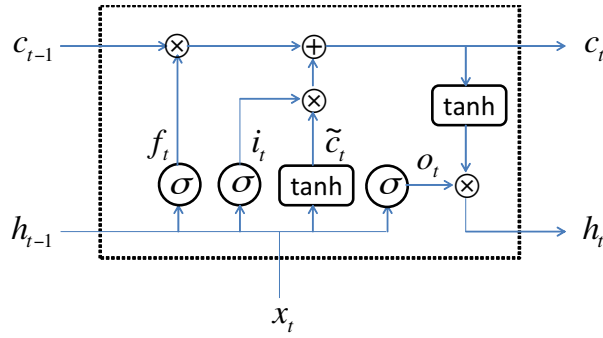


図 2.3 Long Short-Term Memory

2.3 Long Short-Term Memory

実は、2.1 節で説明したような単純なリカレントニューラルネットワークでは、高性能な言語モデルや翻訳モデルを実現することは難しい。言語現象を高い精度で再現するためには、遠く離れた単語間の依存関係もとらえる必要があるが、単純なリカレントニューラルネットワークでは、誤差逆伝播を行う際の「勾配消失」と呼ばれる問題によって、長距離依存関係をうまく学習することができないことが知られている。

自然言語処理では、上述の問題点を解消し、高精度なリカレントニューラルネットワークを実現する手法として LSTM (Long Short-Term Memory) [10] や GRU (Gated Recurrent Unit) [2] と呼ばれる構造がよく用いられる。図 2.3 に LSTM のユニットの構造を示す。LSTM では、リカレントニューラルネットワークの通常の状態ベクトルである h_t に加えて、メモリーセルと呼ばれる状態ベクトル c_t を持ち、どのような情報をどのような場合に記憶（あるいは忘却）するかを細かく制御する「ゲート」と呼ばれる機構を備えている。具体的には、LSTM の状態は、

$$i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1} + b^{(i)}) \quad (2.6)$$

$$f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1} + b^{(f)}) \quad (2.7)$$

$$o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1} + b^{(o)}) \quad (2.8)$$

$$\tilde{c}_t = \sigma(W^{(\tilde{c})}x_t + U^{(\tilde{c})}h_{t-1} + b^{(\tilde{c})}) \quad (2.9)$$

$$c_t = i_t \odot \tilde{c}_t + f_t \odot c_{t-1} \quad (2.10)$$

$$h_t = o_t \odot \tanh(c_t) \quad (2.11)$$

という式によって更新される。ただし、 $\sigma(\cdot)$ はシグモイド関数、 \odot はベクトルの要素積を表す。 i_t , f_t , o_t はそれぞれ、入力ゲート、忘却ゲート、出力ゲートと呼ばれるベクトルであり、状態ベクトルの更新処理を制御する。

LSTM は、ニューラルネットワークを用いた自然言語処理で広く用いられており、いまでは多くの深層学習ライブラリで、高性能なリカレントニューラルネットワークを実現する標準的な「部品」として提供されている。本実験課題でも、LSTM を実験参加者が直接実装する必要はない。

2.4 エンコーダ・デコーダモデル

現在のニューラル機械翻訳技術のベースとなっているエンコーダ・デコーダモデル (encoder-decoder model) [6] では、2 つのリカレントニューラルネットワークを使用する (図 2.4)。ひとつはエンコーダと呼ばれ、翻訳元の文の単語を順次読み込み、文全体の内容を表す実数値ベクトルを生成する役割を担う。もうひとつのリカレントニューラルネットワークはデコーダと呼ばれ、エンコーダが出力した実数値ベクトルを初期状態とし、上述の言語モデルと同様の仕組みで

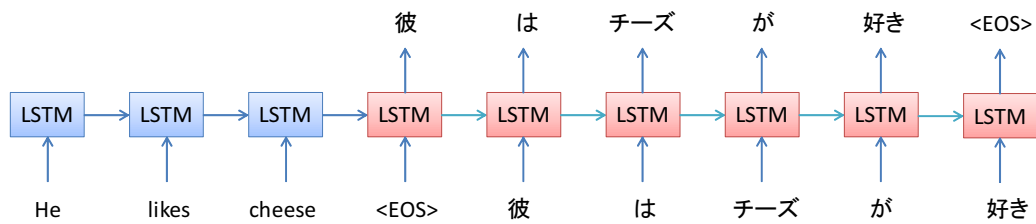


図 2.4 エンコーダ・デコーダモデル

翻訳先の単語列を出力する。文全体の内容をひとつの実数値ベクトルで表現してしまうというのはいかにも乱暴だが、このような単純な仕組みで機械翻訳が実現できてしまうというのは驚きである。

エンコーダ・デコーダモデルの学習は end-to-end で行う。つまり、エンコーダとデコーダをそれぞれ独立に学習されるのではなく、ひとつの大きなニューラルネットワークとして、翻訳元の文から生成される翻訳先の単語列に関するモデルの対数尤度を最大化するようにパラメータを最適化する。

このような学習に必要なのは、「パラレルコーパス」と呼ばれる学習データである。パラレルコーパスとは、文同士の翻訳関係の対応がついている 2 言語のコーパスである。たとえば、機械翻訳モデルの学習・評価用データとしてよく用いられる WMT データセットでは、英仏では約 3600 万文ペア、英独では約 500 万文ペアが提供されている。学習データの量が多ければ多いほど翻訳精度が向上するというのは従来の統計的機械翻訳と同様だが、ニューラル機械翻訳の場合は特にその傾向が強いと言われている。

2.5 アテンション

ニューラル機械翻訳のレベルを大きく引き上げたのは、アテンション (attention) と呼ばれる仕組みである [1]。これは、デコーダが各単語を出力する際の情報として、エンコーダにおける各単語の隠れ状態の重み付き平均を入力として用いるという方法で、翻訳元の文の文脈情報をより詳細にとらえることができるようになる。アテンションの機構が導入されたことによってニューラル機械翻訳の精度は大きく向上し、従来の統計的機械翻訳モデルの性能を凌駕することとなった。

2.6 その他の応用

Seq2seq モデルは、機械翻訳だけでなく様々な自然言語処理タスクに利用されている。代表的な応用に、対話 [15]、文書要約 [14]、質問応答 [3] などがある。

2.7 自然言語処理の実践

本節では、ニューラルネットワーク用フレームワークの Chainer を用いて自然言語を扱う方法を学ぶ。

まず最初に、リカレントニューラルネットワークを用いた言語モデルの学習を行い、動作を確認する。その後、エンコーダー・デコーダモデルを用いた英日翻訳機を作成し、学習及び動作の確認を行う。

2.7.1 言語モデルの作成

まずはじめに、RNN を用いて日本語の言語モデルを学習する。「言語モデル」の詳しい説明は理論編を参照すること。

なにはともあれまずはサンプルを実行してみよう。実験で使用するソースコードはサーバーのホームディレクトリに `dl_exp_nlp.tar.gz` という名前で配置されている。

```
$tar xzvf dl_exp_nlp.tar.gz
$cd dl_exp_nlp
$python language_model_rnn_train.py
```

学習には 10 分前後かかるため、その間にプログラムの解説を行う。学習データは、dataset ディレクトリ内の `data_1000.txt` を使用している。一行毎に、英文とそれに対応する日本語文がタブ文字で区切られて記入されている。また、それぞれの文は半角スペースで単語に区切られている。

- 学習データは、「田中コーパス」 (http://www.edrdg.org/wiki/index.php/Tanaka_Corpus) を加工したものである。
- 「コーパス」とは、主に研究目的のため、自然言語のデータを体系的に収集し、構造化したデータのことを指す。
- `data_full.txt` は田中コーパスの全てのデータであり、`data_100.txt`、`data_1000.txt` はコーパスの中からそれぞれ 100 文、1000 文を抜き出したものになっている。

学習データのパスを行っている `sentence_data.py` を見てみよう。自然言語処理の分野ではしばしば単語を「ID」で管理する。今回のプログラムでも、コーパス中の各単語に ID を割り振っている。日本語については `SentenceData` クラスの `japanese_word_id()` メソッドで「単語文字列 → 単語 ID」、`japanese_word()` メソッドで「単語 ID → 単語文字列」の変換を行えるようになっている。また、`japanese_sentence_data()` メソッドはインデックスを与えると対応する文章データを返すが、これは「単語 ID のリスト」という形式であることに注意する。なお、文の終端を表す EOS (End of Sentence) という概念が存在する。今回は EOS の ID を 0 と定義した。

次に `language_model_rnn.py` を読む。 `LanguageModel` というクラスが定義されており、`chainer.Chain` クラスを継承している。 `chainer.Chain` はニューラルネットワークのモデルを表現するためのクラスであり、ネットワークのパラメータを `chainer.Chain` の `__init__` に渡している。今回用いるのは図 2.5 に示すネットワークである。 x は単語 ID を表現するベクトルで、語彙数に等しい次元を持ち、単語 ID に対応する位置の要素のみ 1 で、他の要素は 0 になっている。このようなベクトルを one hot vector と呼ぶ。本来ならばニューラルネットワークの入力としてはこの one hot vector を用いなければならないが、Chainer には one hot vector を扱うための `EmbedID` というクラスが存在する。このクラスは、入力に one hot vector の代わりに単一の整数を取ることができ、指定した整数の位置が 1 となる one hot vector として扱って

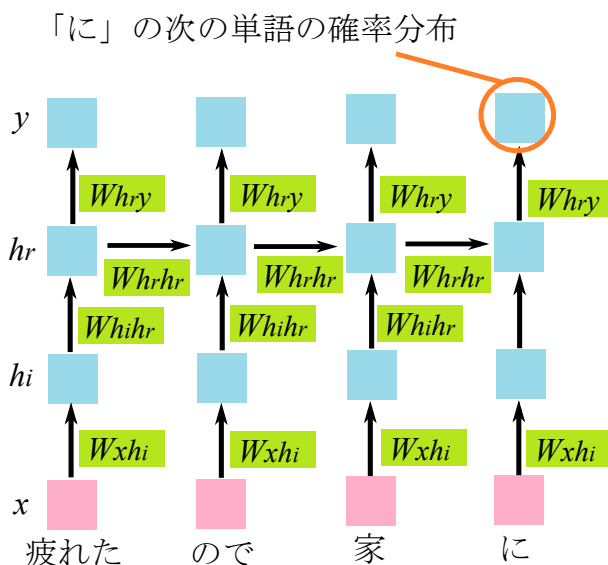


図 2.5 RNN を用いた言語モデル

くれる。今回は W_{xh_i} が EmbedID になる。 W_{xh_i} は、単語 ID に対応する h_i の組が横に並んだ行列と見ることもできる。 Linear クラスは単なる行列である。 EmbedID や Linear には初期化時に入力の次元と出力の次元を渡す必要がある。

また、今回はリカレントニューラルネットワークとして、隠れ層 h_r の値を保存する必要があるため、メンバとして h_r の値を持っており、 `reset_state()` メソッドで初期化できるようにしている。 `__call__` により、単語 ID を入力として受け取りネットワークの出力を返す。出力は、語彙数の次元を持ち、各単語 ID に対応する位置の値が、その単語の出現確率を示している。

softmax

より正確には、出力ベクトルに softmax と呼ばれる処理を行うことで、ベクトルの要素を確率に変換する。

`language_model_rnn_train.py` で実際に学習を行っている。学習データ全体を一周すると「1 epoch」となり、今回はデフォルトでは 10 epochs の学習を行う。学習結果は 1 epoch 毎に、 `trained_model` ディレクトリ以下に保存される。

さて、そろそろ学習が終わったのだろうか。学習が終了したら、 `language_model_rnn_test.py` を実行することで、学習した言語モデルをテストすることができる。

```
$python language_model_rnn_test.py
```

として実行し、最初の単語、例えば「私」と入力すると、2 単語目に来る確率が最も高い単語が出力される。さらに、「入力した 1 単語目、推定された 2 単語目」という単語列から、最も確率の高い 3 単語目が続けて出力され、文の終端である EOS が出力されるまで、再帰的に推定を行う。入力には半角スペースで区切った複数単語を入れることもでき、「私 は ここ」と入力した場合、最も確率の高い 4 単語目から推定が行われる。

演習 2.7.1 適当な単語または単語列を入力し、入力によって出力が変化すること、ある程度「日本語らしい」文章になることを確認せよ。

2.7.2 Long Short-Term Memory (LSTM)

理論編で学んだように、単純な RNN で精度の良い学習を行うのは難しく、LSTM や GRU 等の改良された構造が用いられることが多い。ここでは LSTM を使用して先程と同様に言語モデルの学習を行う。

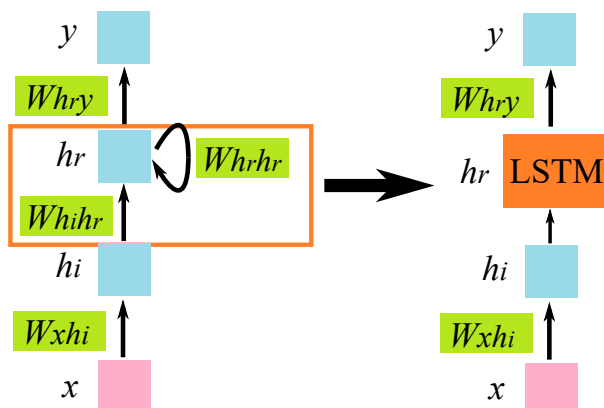


図 2.6 Chainer の LSTM クラスによる RNN の置き換え

Chainer には既に LSTM のモデルが用意されている。図 2.6 に示すように、RNN 部分をまるごと置き換えるようになっており、隠れ層のデータも含まれている。language_model_lstm.py に、LSTM を用いて言語モデルの学習を行う例を示した。RNN を自分で実装したときとは異なり、reset_state() メソッドでは、chainer.links.LSTM の reset_state() メソッドを呼び出す様になっている。これは、隠れ層のデータは chainer.links.LSTM の内部に含まれているからである。さて、今回は LanguageModel クラスの __call__ の実装をまだ行っていない。language_model_rnn.py も参考にしつつ、各自で実装してみよ。

演習 2.7.2 language_model_lstm.py における LanguageModel クラスの __call__ を実装し、実際に学習を行ってみよ。学習は language_model_lstm_train.py を実行することで行うことができる。学習が終了したら、language_model_lstm_test.py を実行し、動作を確認せよ。

演習 2.7.3 language_model_lstm_train.py 及び language_model_lstm_test.py を書き換え、英語の言語モデルを学習し、動作を確認せよ。学習には時間がかかるため、学習中に次の章を読み進めておくとい。

2.7.3 エンコーダー・デコーダモデル

次に、エンコーダー・デコーダモデルを用いた英日翻訳機の学習を行う。エンコーダー・デコーダモデルについては理論編を参照すること。LSTM を用いた英日翻訳の例を図 2.7 に示す。translator_model.py を読むと、このモデルが定義されている。注意点として、Chainer の LSTM クラスは隠れ層のデータを内部に保持している。今回は隠れ層のデータをエンコーダ用 LSTM からデコーダ用 LSTM に引き継ぐ必要があるため、LSTM クラスの代わりに StatelessLSTM クラスを用いた。StatelessLSTM は、隠れ層のデータを内部に持たず、外部に変数として保存して計算時に与えるようになっている。また理論編で学んだように、LSTM は隠れ層の他にメモリーセルと呼ばれる状態ベクトルも持つため、これも隠れ層と同様に変数として保存する必要がある。

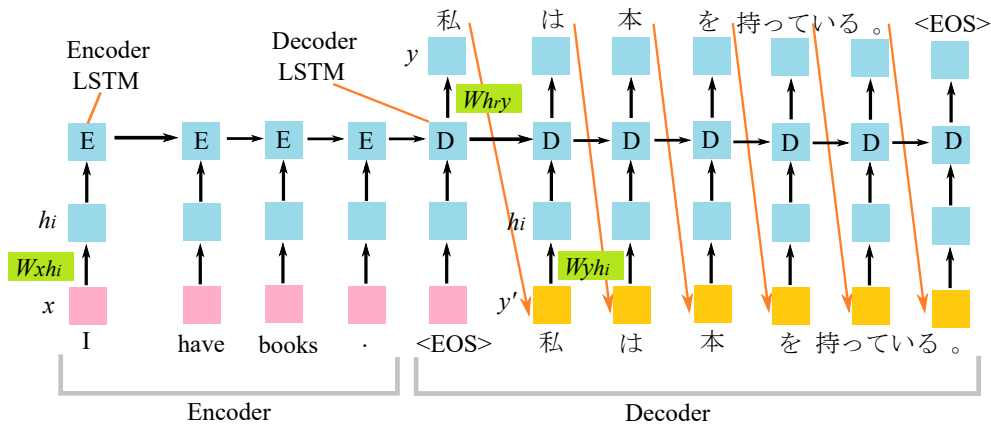


図 2.7 エンコーダー・デコーダモデルによる英日翻訳

演習 2.7.4 `translator_model_train.py` を実行し、翻訳モデルの学習を行ってみよ。学習している間に、`translator_model.py` 及び `translator_model_train.py` を読んで、ネットワークを定義している部分、エンコードやデコードを行う部分などを探してみよ。

演習 2.7.5 学習が終了したら、`translator_model_test.py` を実行し、結果を確かめてみよ。入力の際は「I have a book .」のように、ピリオドも単語として区切るため、スペースを挟む必要があることに注意する。

大抵の場合、一見日本語のような文章ではあるが、翻訳結果としては間違った文章が出力されたと思われる。これは主に学習データの数がないためである。今回は 1000 文の学習データを用いて 10 epochs の学習を行ったが、この数では正確な翻訳は不可能である。`traind_model/translator_full.model` は、`data_full.txt` を用いて約 15 万文で 10 epochs の学習を行ったデータである。`translator_model_test_full.py` を実行することで、このデータを用いて翻訳を行うことができる。

演習 2.7.6 `translator_model_test_full.py` を実行し、動作を確認せよ。

2.7.4 未知語対応

今までのプログラムで作成したモデルでは、コーパスに含まれない単語が入力された場合に対応できなかった。そこで、未知の単語が入力された場合でもある程度動作する仕組みを考える。簡単な例としては、未知の単語を共通の「<UNKNOWN>」という単語に置き換えてしまうという手法が考えられる。この手法を動作させるには、学習の際に「<UNKNOWN>」を含むデータセットが必要になる。そのため、データセットを読み込む際に変換を行う。変換方法の例としては

- 出現回数が一定回数以下の単語を全て「<UNKNOWN>」に置き換える。
- 各単語が初めて出現したときに「<UNKNOWN>」に置き換える。

等が考えられる。「<UNKNOWN>」を扱うための準備として、`sentence_data.py` に

```
UNKNOWN_WORD_ID = 1
```

を追記し、また単語の辞書作成部分は

```
self.en_word_to_id = {"<EOS>": EOS_ID, "<UNKNOWN>": UNKNOWN_WORD_ID}
self.en_word_list = ["<EOS>", "<UNKNOWN>"]
self.jp_word_to_id = {"<EOS>": EOS_ID, "<UNKNOWN>": UNKNOWN_WORD_ID}
```



```
self.jp_word_list = ["<EOS>", "<UNKNOWN>"]
```

とする。

演習 2.7.7 `sentence_data.py` の `__init__` で学習データを読み込んでいる部分を書き換え、データセットに「<UNKNOWN>」が含まれるようにせよ。

演習 2.7.8 `language_model_lstm_test.py` や `translator_model_test.py` を書き換え、未知の単語が入力された場合に「<UNKNOWN>」に置き換えて、出力が得られるようにせよ。これをテストする場合は、前の課題で書き換えた `sentence_data.py` を用いて学習をやり直す必要があることに注意せよ。

2.7.5 発展課題

余力のある人は、以下の発展課題に取り組んでみよ。

- いくつかの学習データをまとめて学習することで、主に GPU を使用する場合の学習効率を上げることができる。この手法を mini batch と呼ぶ。Chainer では入力データの次元を挙げることで簡単に mini batch 化が可能であるが、系列データはデータによってデータ長が異なるため、少々面倒である。Chainer には可変長データをバッチ化しやすくするために `NStepLSTM` というクラスがある。`NStepLSTM` の使い方を調べ、実験で用いたプログラムを mini batch 化せよ。
- 今回は単語 ID から特長ベクトル h_i を生成するための行列 W_{xh_i} も同時に学習したが、各単語を表す特徴ベクトルを事前に学習することも可能である。学習済みのデータを公開している人も多いので、それらのデータを用いるように今回のプログラムを変更してみよ。Word embedding や、Word2Vec 等のキーワードで検索すると良い。
- 配布したデータセットは、前述のように田中コーパス (http://www.edrdg.org/wiki/index.php/Tanaka_Corpus) を加工したものであるが、加工前のコーパスデータには、単語の原型などの追加データが存在する。それらのデータを上手く使うことで、学習の効率や精度を上げることができないか検討せよ。なお、今回英単語の切り分けには Stanford CoreNLP (<https://stanfordnlp.github.io/CoreNLP/>) を用いた。
- ネットワークの構造を変更することで、より精度良く学習が行なえないか検討せよ。例えば、理論編で学んだ attention を導入してみよ。

2.7.6 自由課題のヒント

- エンコーダー・デコーダモデルの用途は翻訳に限らず、系列データの入力から系列データを出力する一般のタスクに応用できる。例えば、長い文章を入力すると要点をまとめて短くしてくれるような要約タスクなどが考えられる。
- RNN の入力や出力は単語（列）に限らない。例えば画像分野と組み合わせて、連続画像を入力としたり、画像から文章を生成する等の応用も考えられる。

Reinforcement Learning

第 3 章

本実験課題では、近年の人工知能研究において様々な場面で用いられている、強化学習 (reinforcement learning) についてその基本的な原理と動作を学ぶ。

強化学習が有用な例として、与えられた環境の中で「賢く」行動するロボットの AI (以下、エージェントと呼ぶ) を作ることを考えてみよう。そのようなエージェントを実現するもっとも素朴な方法として、「X という状態のときには Y する」といったようなルールをひたすら書き連ねたプログラムを書く、というアプローチが考えられる。しかし、このような「ルールベース」の方法は、少し環境が複雑になるとすぐに破綻する。そこで、強化学習では、人間が明示的にエージェントの行動のためのルールを教えるのではなく、エージェント自身に多数の試行錯誤をさせ、その経験を通して適切な行動基準を見つけさせるというアプローチをとる。人間は、エージェントが適切な行動をした場合には、プラスの「報酬」を与え、不適切な行動をした場合にはマイナスの報酬を与える。報酬は必ずしも各行動ごとに与える必要はなく、どのように行動すれば (長い目で見て) 報酬を最大化できるのかについてはエージェント自身が考える、というのが強化学習の特徴である。

近年、強化学習は様々な分野で応用が進んでいる。ブロック崩しやインベーダーゲームといった簡単なビデオゲームをプレイすることのできる AI [4] や、人間の棋譜を一切使わずにきわめて高い棋力を達成した囲碁プログラム [5] などのニュースは記憶に新しい。

3.1 マルコフ決定過程

強化学習を考える上での基本的な枠組みであるマルコフ決定過程 (Markov decision process, MDP) について簡単に述べる。マルコフ決定過程とは、

- 状態 (state) の有限集合 S
- 行動 (action) の有限集合 A
- 状態遷移関数 $P(s'|s, a)$: 状態 $s \in S$ において行動 $a \in A$ をとった場合に状態 $s' \in S$ に遷移する確率
- 報酬関数 $R(s, a, s')$: 状態 s から行動 a によって状態 s' に遷移したときに得られる報酬

によって構成され、エージェントの環境 (environment) を規定する。

いま、図 3.1 に示すように、この環境の中で、時刻 t において状態 s_t にあるエージェントが、行動 a_t を選択したとする。その結果、状態は s_{t+1} に変化し、同時にエージェントは報酬 r_{t+1} を受け取る。強化学習におけるエージェントの目的は、現在から未来にわたる累積報酬

$$g_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad (3.1)$$

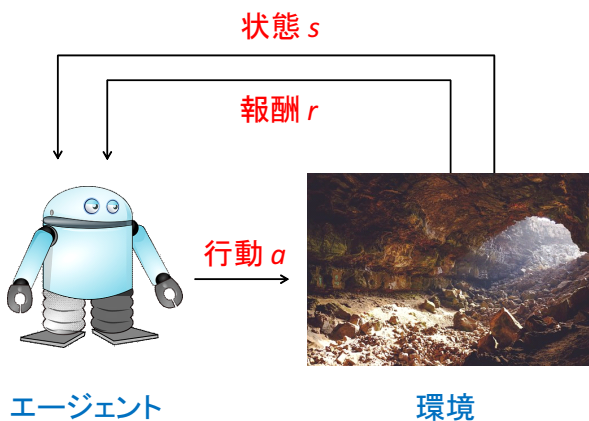


図 3.1 強化学習

を最大化する行動を選択することだと定式化される． γ は割引率 (discount factor) と呼ばれる値であり，現時点ですぐにもらえる報酬を未来の報酬よりもどれだけ重視するかを決めるパラメータである．

3.2 Q 学習

強化学習のためのアルゴリズムとして最もよく知られているもののひとつに，Q 学習 (Q-learning) と呼ばれるアルゴリズムがある．いま，状態 s のときに行動 a を取り，その後，最善の行動を取り続けた場合に得られる累積報酬の期待値を $Q^*(s, a)$ と書くものとする． $Q^*(s, a)$ は，Bellman 方程式と呼ばれる以下の再帰的な関係式を満たす．

$$Q^*(s, a) = \sum_{s'} P(s'|s, a)(R(s, a, s') + \gamma \max_{a'} Q^*(s', a')) \quad (3.2)$$

もし，任意の s と a の組み合わせに対して $Q^*(s, a)$ がわかっているのであれば，状態 s_t において，エージェントは， $Q^*(s_t, a_t)$ が最大の行動 a_t を選択すれば累積報酬を最大化することができる．つまり，エージェントの行動選択の問題が解けたことになる．

もちろん通常は $Q^*(s, a)$ は未知なので，何らかの方法で推定する必要がある．Q 学習では，エージェントが環境のなかで行動するたびに，以下の更新式にしたがって $Q^*(s, a)$ の推定値である $Q(s, a)$ を更新する．

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (3.3)$$

すなわち， $Q(s_t, a_t)$ を

$$r_{t+1} + \gamma \max_a Q(s_{t+1}, a) \quad (3.4)$$

に近づけるように更新する．ただし， $\alpha \in [0, 1]$ は学習率と呼ばれるパラメータである．Q 学習では，エージェントが全ての状態をゼロ以上の確率で訪れるように行動する場合，学習率を徐々に下げていくことにより， $Q(s, a)$ が $Q^*(s, a)$ に収束することが知られている．

3.2.1 テーブルを用いた Q 学習の例

Q 学習の様子を例を用いて説明しよう．この例では，学習率 α は常に 1.0，割引率 γ は 0.9 であるとする．

図 3.2 に学習の初期状態を示す．エージェントは状態 1 から出発するものとする．また，エージェントは各状態において Up, Down, Left, Right の 4 つの行動をとることができる．ただし，

	Up	Down	Left	Right	End
1	0	0	0	0	
2	0	0	0	0	
3	0	0	0	0	
4	0	0	0	0	
5	0	0	0	0	
6	0	0	0	0	
7					0
8	0	0	0	0	
9	0	0	0	0	
10					0
11	0	0	0	0	
12	0	0	0	0	

図 3.2 Q 学習（初期状態）

	Up	Down	Left	Right	End
1	0	0	0	0	
2	0	0	0	0	
3	0	0	0	0	
4	0	0	0	0	
5	0	0	0	0	
6	0	0	0	0	
7					100
8	0	0	0	0	
9	0	0	0	0	
10					-100
11	0	0	0	0	
12	0	0	0	0	

図 3.3 Q 学習（状態 7 と状態 10 を経験した後）

状態 7 と状態 10 では、End という行動しかとることができず、それぞれ 100 もしくは -100 の報酬を受け取ったのち、エピソードが終了する。図の右側に示すテーブルは、 $Q(s, a)$ を表し、すべてゼロで初期化されている。

この状態からエージェントがランダムに行動した結果、状態 7 に到達したとする。すると、次のステップでは行動として End をとり、報酬 100 を受け取り、エピソードが終了するため、更新式

$$Q(7, \text{End}) \leftarrow Q(7, \text{End}) + 1.0 \times [100 + 0.9 \times 0 - 0] \quad (3.5)$$

によって、 $Q(7, \text{End})$ の値が 100 に更新される。次のエピソードで、状態 10 に到達した場合、報酬として -100 を受け取るため、同様の計算により $Q(10, \text{End})$ の値は -100 となる。この時点での $Q(s, a)$ の値を格納するテーブルの状態を図 3.3 に示す。

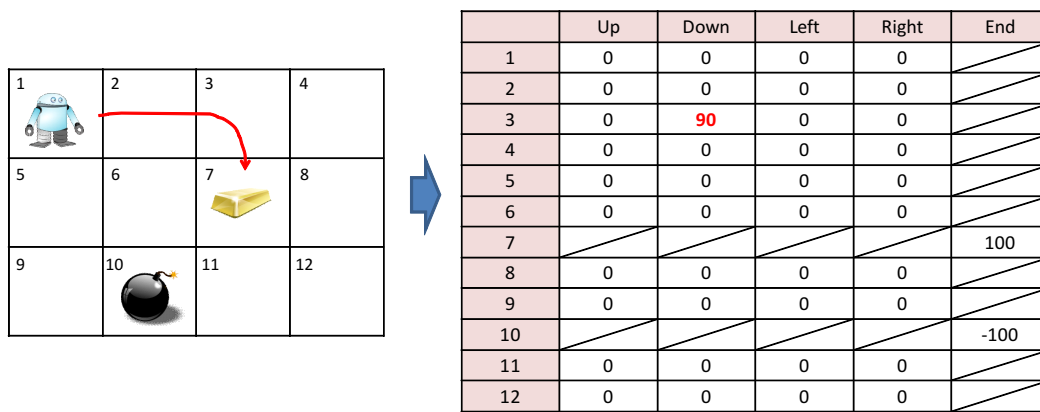
その後、エージェントは図 3.4 に示すように、状態 3 を経由して状態 7 に到達したとする。この場合、更新式は、

$$Q(3, \text{Down}) \leftarrow Q(3, \text{Down}) + 1.0 \times [0 + 0.9 \times 100 - 0] \quad (3.6)$$

となるため、 $Q(3, \text{Down})$ の値が 90 に更新されることになる。

3.3 関数近似による Q 学習

前節で説明した Q 学習では、 $Q(s, a)$ の値を格納するためにテーブル（配列）を利用した。状態の数が少ない場合にはこのような方法で学習を行うことができるが、現実の問題に強化学習を



	Up	Down	Left	Right	End
1	0	0	0	0	
2	0	0	0	0	
3	0	90	0	0	
4	0	0	0	0	
5	0	0	0	0	
6	0	0	0	0	
7					100
8	0	0	0	0	
9	0	0	0	0	
10					-100
11	0	0	0	0	
12	0	0	0	0	

図 3.4 Q 学習 (状態 3 から状態 7 に遷移した後)

適用しようとする場合、テーブルを利用した Q 学習は不可能な場合が多い。例えば、ビデオゲームの AI を強化学習で作ることを考えてみよう。入力として、ビデオゲームの画面、すなわち多数のピクセルから構成される画像の情報を考えた場合、それによって決まる「状態」は無数に存在することになる。状態の数が非常に大きい場合、テーブルに必要なメモリが莫大になることもさることながら、そもそもエージェントが同じ状態を経験すること自体がまれになるため、学習が実質的にほとんど進まない。

関数近似 (function approximation) によるアプローチでは、 $Q(s, a)$ をテーブルで表現するのではなく、ニューラルネットワーク等、パラメータ θ によって規定される関数 $Q(s, a; \theta)$ で表現し、 $Q^*(s, a)$ を精度よく近似することを目指す。

いま、ある時点 i でのパラメータを θ_i としたとき、それをより適切な値にするにはどのように更新すればよいだろうか？ テーブルを用いた Q 学習では、 $Q(s_t, a_t)$ を $r_{t+1} + \gamma \max_a Q(s_{t+1}, a)$ に近づけるように更新を行ったのであった。関数近似でも、それと同様の考え方により、 $Q(s, a; \theta)$ を近づける目標として、パラメータ θ_{i-1} から計算される

$$y_i = r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) \quad (3.7)$$

を考え、それに対する誤差が小さくするように更新を行うという方法が考えられる。誤差の基準として 2 乗誤差を用いた場合は、

$$L(\theta_i) = \mathbb{E}[(y_i - Q(s, a; \theta_i))^2] \quad (3.8)$$

を最小化するようにパラメータを最適化する問題だとして解くことができる。

3.4 強化学習の実践

本節では、強化学習を用いて問題を解く AI を作成する方法を学ぶ。

まず最初に、与えられた単純な迷路に対して報酬を最大化する AI を、単純な Q 学習を用いて作ることで、強化学習の基礎を理解する。その後に、ChainerRL を用いて実践的な AI を作り、複雑な問題に対して強化学習を適用する方法を学ぶ。

3.4.1 準備

最初のタスクは、単純な迷路である。図 3.5 に示す 2 次元の離散的な空間（グリッドワールド）において、スタートからゴールを目指すことが目標となる。図 3.5 において、エージェントは S と書かれた位置 $((y, x) = (0, 0))$ から行動を始め、G と書かれた位置 $((y, x) = (2, 3))$ にたどり着くことを目指す。エージェントは 4 方向に動くことができるが、灰色で表されるマスには侵入できない（移動できず 1step が経過する）。エージェントは迷路全体を観測することはできず、自分のいる x, y 座標を観測することができる。

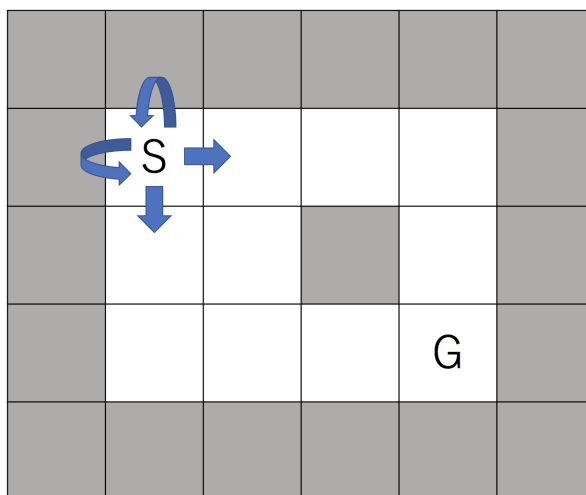


図 3.5 単純な迷路

`dl_exp_rl.tar.gz` を展開してソースコードを入手しよう。ソースコードの中にある `main.py` を実行することで、ランダムに動くエージェントが迷路に取り組む様子が観察できる。

```
$tar xzvf dl_exp_rl.tar.gz
$cd dl_exp_rl
$python main.py
```

演習 3.4.1 ソースコードをダウンロードし、`main.py` を実行せよ。また、`README.md` を読み、`random_agent.py` と `easymaze_env.py` の場所を確認せよ。

迷路の様子が標準出力に一気に流れ込んでくるので、少し眺めたら `Ctrl+c` で止めてしまっていて構わない。また、ゆっくり眺めたければ

```
$python main.py | less
```

のようにすればよい。

3.4.2 相互作用の理解

このタスクに限らず、強化学習は環境とエージェントが相互作用することで学習を行うものである。

今回用いるソースコードは、オープンソースライブラリである OpenAI Gym (<https://github.com/openai/gym>)、および ChainerRL (<https://github.com/chainer/chainerrl>) の書式に則って書かれており、環境とエージェントが完全に分かれている。その二者の相互作用を担うのが `main.py` である。

`main.py` を読み、適宜編集・実行しつつ、エージェントと環境がどのように相互作用しているのかを理解しよう。

演習 3.4.2 `main.py` の中で、`env` および `agent` を作成している行を答えよ。また、これらに対して呼び出されているインスタンス変数・メソッドをすべて挙げよ。自由課題などで `env` や `agent` を自分で作成する場合、最低限これらの変数・メソッドをすべて実装する必要がある（インタフェース）。

演習の解答は `answer_sheet.md` に書き込むとよい。

なお、`main.py` 内の `prints_detail` 変数を `False` にすることで、毎 `step` の迷路の出力を止めることができる。

演習 3.4.3 `train`、`test` のそれぞれで 100 episodes の実験を行って、ランダムエージェントがゴールするまでの episode ごとの平均 `step` 数を出力せよ。`main.py` の編集が必要となる。300 steps 掛かってもゴールできない場合、300 steps でゴールしたものとして平均を計算せよ。

episode ごとの平均 `reward` は既に出力されるようになっているので、その実装を参考にするとうよい。

3.4.3 エージェントの理解

`random_agent.py` を読み、ランダムエージェントがどのように動作しているかを理解しよう。

ランダムエージェントは、`__init__()` で行動集合のサイズを取得し、`act()` が呼び出されると行動の中から一つをランダムに選ぶようになっている。

ランダムエージェントはこの小さな迷路でもゴールするのにかなり時間がかかってしまうが、迷路の形状は毎回同じなので、人間であればすぐに最短経路を求めることができるはずである。

例えば、「壁にぶつかるまで下に行き、ぶつかったら右に行く」などはその一例である。

演習 3.4.4 コメントを参考に、穴あきファイル `rulebase_agent.py` の穴を埋めることで、この形状の迷路を最短 step 数で解くルールベースエージェントを作成せよ。「壁にぶつかるまで下に行き、ぶつかったら右に行く」ように実装すればよい。エージェントが受け取る観測を、`print()` 関数を用いたり `easymaze_env.py` を読んだりすることで確認することが必要となることに注意せよ。

`main.py` の

```
agent = agents.RandomAgent(env, gpu_id)
```

を

```
agent = agents.RulebaseAgent(env, gpu_id)
```

に変更するだけで、ランダムエージェントの代わりにルールベースエージェントを試すことができる。

また、`rulebase_agent.py` 内の

```
raise NotImplementedError()
```

は、「このセクションはまだ実装されていないので、ここを通るときは必ず実行時エラーを出してください」という合図である。穴を埋めた後はこの行を削除すればよい。

3.4.4 環境の理解

演習 3.4.4 で作成したように、人間が迷路を観察し、あらかじめ移動ルールを定めておくことで、単純な迷路を解く AI は作成できる。

しかしながら、この方法では異なる環境に対応できない。`easymaze_env.py` を編集し、これを示そう。

演習 3.4.5 `easymaze_env.py` を編集し、迷路に最小限の変更を行うことで、演習 3.4.4 で作成したエージェントがゴールできないような環境にせよ。ただし、別のルートでゴールできるようにしておくこと。

この章ではあまり詳しく触れないが、自由課題で自作の環境に対して強化学習を適用したい場合、このソースコードを基に各メソッドを定義するとよい。特に `_` から始まっているメソッドは必ず実装しなければならないものとなっている。

演習 3.4.6 `easymaze_env.py` において、`_` から始まるメソッドをすべて探し出し、それぞれの役割を `easymaze_env.py` に 1 行のコメントで書き込め。

3.4.5 Table Q 学習の実装

Q 学習は「ある状態からある行動を取ったとき、最大でいくら累積報酬がもらえるか」を学習する手法である。

Q 値のハッシュテーブルを用いて学習を行うエージェントを作成し、この迷路に適用してみよう。

演習 3.4.7 コメントを参考に、穴あきファイル `table_q_agent.py` の穴を埋めることで、table-Q エージェントを実装せよ。ルールベースエージェントの代わりに table-Q エージェントを用いるために、`main.py` も変更する必要があることに注意せよ。

演習 3.4.8 演習 3.4.3 と同様に、100 episodes の実験を行って、table-Q エージェントがゴールするまでの平均 step 数を出力せよ。また、train のうち、最初の 10 episodes、最後の 10 episodes の平均 step 数を出力せよ。

演習 3.4.9 table-Q エージェントには、迷路の各地点における Q 値を文字列形式で出力する `q_table_to_str()` メソッドが用意されている。このメソッドを適宜呼び出し `print()` することで、Q 値が学習されていく過程を観察せよ。時間があれば、割引率や学習率、迷路の報酬などを変えて、学習の過程がどのように変わるかを観察してみよ。

3.4.6 Train と Test, On-Policy と Off-Policy

教師あり学習において train データと test データが区別されるように、強化学習においても train と test という概念がある。

強化学習では、エージェントは方策 (policy) π を持っており、環境から得られた観測 s に対して行動集合上の確率分布 $\pi(s)$ に従って行動 $a \sim \pi(s)$ を選択する。train 時の行動選択に用いる方策 π について、その方策 π をより良い方策にしていく学習を方策オン型 (on-policy) という。これに対し、 π とは別の test 時用の方策 π' を用意し、この方策 π' を最適方策に近づけていく学習を方策オフ型 (off-policy) という [13]。

方策オフ型学習の利点の一つとして、強化学習において大きな問題の一つである探索と活用の問題を避けることができる点が挙げられる。強化学習においては、次の行動を選択する際に、「これまでの学習によって、既に良いとわかっている行動」を取るべきか、「これまでの学習で、まだ良いか悪いか判断できない行動」を取るべきか、という問題に直面することになる。前者だけを取ってはいは局所解に陥ってしまい、後者だけを取ってはいは報酬を最大化することができない。前者を取ることを活用 (exploitation)、後者を取ることを探索 (exploration) という。方策オフ型学習では、少なくとも test 時にはこの問題を回避することができる。test 時には探索は不要だからである。

Q 学習における ϵ -greedy は、この探索と活用の問題を解決するための手法の一つである。train 時において、Q 学習エージェントはできるだけ Q 値の高い行動を選択することで、現時点で「良い」と思われる行動についてより多く学習したい (活用)。しかしながら、活用だけでは「まだ見つかっていない」より良い行動を学習することができない (探索)。そこで、一定の確率でランダムに行動し (ϵ)、それ以外るとき Q 値が最大の行動を選択することを考える (greedy)。このようにすることで、探索と活用を両立することができる。

Q 学習は方策オフ型の学習である。なぜなら、train 時にのみ ϵ -greedy を用いて学習し、test 時は常に Q 値が最大の行動を選択するからである。この意味で、前項で実装した table-Q エージェントは Q 学習の実装としては十分はでない。これを修正しよう。

演習 3.4.10 Table-Q エージェントを、train 時に ϵ -greedy で行動し、test 時に greedy で行動するようにせよ。その後、演習 3.4.3 と同様に、100 episodes の実験を行って、エージェントがゴールするまでの平均 step 数を出力し、train 時と test 時の結果を比較せよ。

実装の仕方がわからない場合は、まず train 時と test 時で行動を選択するときのどのメソッドが呼び出されるかに着目してみよ。 ϵ -greedy 自体は既に `table_q_agent.py` に実装されているはずである。train 時にはこれを行い、test 時には greedy に行動するようにすればよい。

ただし、`main.py` を書き換えて呼び出すメソッドを変えてはいけない。エージェントのインタフェースは共通のものを用いているからである。

正しく実装できていれば、test 時には第 3.4.3 項で実装したルールベースエージェントと同じ

最短距離でゴールすることができるはずである。

3.4.7 Deep Q-Network

ここまでで我々は、ハッシュテーブルを用いて実装された Q 学習が、単純な環境に対して最適方策を学習できることを見てきた。しかしながら、この方法は環境が複雑になると適用できなくなってしまう。

例えば、環境の状態 s が離散的な値ではなく連続量になることを考えてみよう。今回の実装では $Q(s, a)$ は s をキーとする dictionary (hash) で実装されており、 s が連続量になると同じ状態が起こる確率がほぼ 0 になってしまうため、学習ができなくなってしまう。また、 s が離散値であっても、空間がメモリに乗り切らないほど大きければ、やはり今回の実装では学習ができなくなってしまう。

このような環境に対処するために、 $Q(s, a)$ をハッシュテーブルではなくニューラルネットワークを用いて表現することを考える。すなわち、Q 関数を連続量を取る関数で近似することで、環境の状態が連続量であっても対処できるようにする。ニューラルネットワークは weight の次元数を（学習対象と無関係に）自由に定めることができるので、環境の状態数が非常に大きかったとしても、（正しく学習できる保証はないが）メモリに乗り切る程度の weight で Q 関数を表現することができる。

Q 学習をニューラルネットワークを用いて行う方法を考えよう。通常の Q 学習では、状態 s に対して行動 a を取り、報酬 r と次状態 s' を得た場合の更新式は次のようになる。

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right) \quad (3.9)$$

Q 学習が十分収束したとき、この更新式を用いても Q は更新されないはずである。すなわち、

$$r + \gamma \max_{a'} Q(s', a') - Q(s, a) \simeq 0 \quad (3.10)$$

となっているはずである。実際、この式が成立しているということは、Q 値が「現状態における報酬と、次状態以降の累積報酬の最大値の和」を表しているということであるから、Q 値を正しく学習できているということになる。

すなわち、Q 学習とは、大量の学習データ (s, a, r, s') に対して、

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a') \quad (3.11)$$

が成り立つような関数 Q を求める学習であるといえる。これは教師あり学習と同様であるから、二乗誤差などを用いて左辺と右辺の差を最小化するようにニューラルネットワークを学習させればよい。

Mnih らが 2015 年に発表した deep Q-network (DQN) では、この基本アイデア以外に様々な工夫が為されている。例えば以下のようなものである：

- Experience replay. 学習データを replay buffer と呼ばれるバッファに溜め込み、そこからランダムにサンプリングして学習を行う。これにより、学習データの時間的相関を減らすことができ、学習が安定する。
- Fixed target Q-network. 学習対象となる式 (3.11) の左辺に用いられる Q 関数を固定し、一定時間ごとに最新の Q 関数に置き換える。これにより、学習対象があまり変化しなくなり、学習が安定する。

ChainerRL を用いると、DQN を始めとする様々な深層強化学習の手法を手軽に実装することができる。これを確かめよう。

演習 3.4.11 `main.py` を編集し, DQN エージェント (`agents/dqn_agent.py`) にこの迷路を解かせてみよ. その後, `dqn_agent.py`, `agents/models/dqn_models.py` を読んでみよ. 重要な実装はすべて `chainerrl` が担当していて, 最小限の記述で DQN が実装できていることがわかるはずである. 時間があれば, DQN のモデルを色々変えて (線形回帰にしてみる, 活性化関数を変えてみるなど), 学習がどのように変わるかを観察せよ.

3.4.8 状態空間が連続な環境における Q 学習

前項で述べたとおり, ハッシュテーブルを用いて実装した Q 学習では, 環境の状態が連続値になったときに対応できない. これに対して, DQN では, Q 関数をニューラルネットワークを用いて表現しているため, 状態空間が連続になっても対応できる. これを確かめよう.

CartPole

状態空間が連続なゲームとして, ここでは OpenAI Gym に実装されている CartPole というゲームを扱う.

CartPole は次のようなゲームである. まず, 図 3.6 のように黒い台車の上に茶色の棒が乗った状況を考える. 棒の下端は台車に取り付けられており, 放っておくと棒は下端を中心に回転してしまう. エージェントは台車を右または左のどちらかに押すことができる. この状況で, できるだけ長い間棒を倒さないようにすることが目標となる. 観測として台車の位置, 速度, 棒の角度, 角速度の 4 つの実数が与えられ, ゲームが終了していなければエージェントは各 step で 1 単位の報酬を得る. エージェントが行える行動は右に押すか, 左に押すかの 2 択のみである. 棒が一定以上傾くか, 台車が一定範囲外に出るか, 200steps が経過した時点でゲームは終了する. すなわち, 報酬の最大値は 200 である.

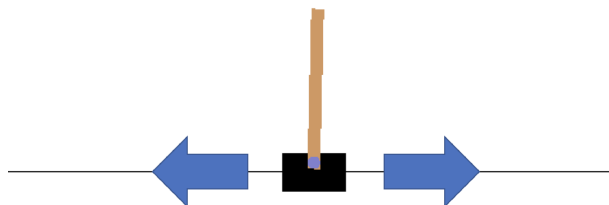


図 3.6 Cart Pole の様子

`main.py` の

```
env = gym.make('Easymaze-v0')
```

を

```
env = gym.make('CartPole-v0')
```

に変更するだけで, CartPole を試すことができる.

演習 3.4.12 CartPole に対して DQN を学習させ, 様子を観察せよ. もし余裕があれば, ローカル環境 (学科 PC など) でも試してみるとよい. ローカル環境では, `main.py` の `prints_detail` 変数を `True` にすることで, エージェントが CartPole をプレイする様子を動画で見ることができる. ゲームは棒が 15 度傾いた時点で終わってしまうので, 各 episode が一瞬で終わってし

まって動画がよくわからないかもしれない。その場合は、`done` が `True` になってもループを抜くずに `env.render()` を続けるようにするとよい。

先述の通り、報酬の最大値は 200 点である。おそらく、現在の DQN は数十点程度しか取れなかったことだろう。

演習 3.4.13 CartPole において、10episodes 連続で 200 点を取ることができるエージェントを作成せよ。現在の DQN のどこに問題があるのかを考え・調べ、様々な方法を試してみよ。

例えば、以下のような項目を調べるとよい。

- 隠れ層の大きさや枚数は必要十分か。
- 学習率や最適化アルゴリズムは適切か。
- 学習の episode 数は十分か。
- 過学習を起こしていないか。
- 探索の方式は適切か。
- Replay buffer は小さすぎないか。
- DQN より良いアルゴリズムはないか。

周りの学生や TA と相談し、できるだけ高得点を取ることのできるエージェントを作ろう。

3.4.9 発展課題・アイデア

ここでは、実験後半の自由課題で強化学習を使いたい学生のために、いくつか課題の種を挙げる。これらの課題のうちいずれかを選択してもよいし、これらを基に自由に課題を設定しても構わない。

- OpenAI Gym には、様々なゲームが用意されている。CartPole のように比較的単純なゲームだけでなく、ビデオゲーム (Atari) や対戦ゲーム (囲碁) などもある。これらのゲームで高得点を得られるエージェントを作ってみよう。その際、1 つのゲームに特化するのか、どんなゲームにも適用できるようにするのは意識すべきである。例えば、今年の 6 月には Microsoft のチームが Atari の “Ms. Pac-Man” で最高得点である 999,990 点を取ったと発表した [11] が、彼らはこのゲームに特化したエージェントを作っており、全く事前知識を用いていないというわけではない。その一方で、DeepMind が発表した DQN は、Atari の 49 のゲーム全てに全く同じパラメータを用いて学習させ、多くのゲームで人間を超えるスコアを出しているが、すべてのゲームで最高得点を取ったというわけではない。
- 強化学習は非常に一般的なモデルであり、現実の様々な問題に適用できるはずである。解決すべき身近な問題を見つけ、強化学習の仕組みに落とし込むことで解決してみよう。制御・最適化などの分野や、自律ロボットの行動などが応用先として有名である。
- この実験の他のトピックと組み合わせてみよう。画像分類で強化学習をするとはどういうことだろうか？ あるいは、翻訳や要約と強化学習を組み合わせるとどうなるだろうか？
- 最新の強化学習の手法を調べ、実装してみよう。まだソースコードが公開されていないなら、実装して世界に公開することで、誰かの役に立てるかもしれない。
- ソーシャルゲームや有名なボードゲームなど、身近なゲームを `gym.Env` として実装し、強化学習を適用してみよう。どのようなエージェントが得られるだろうか。
- 強化学習の手法を、他の手法と比較してみよう。例えば、強化学習の環境として仮定されている Markov Decision Process では、状態遷移規則 $T(s, a) \in S$ がわかっている線形

計画法などで解くことができる。また、遺伝的アルゴリズムなどを用いてもエージェントを作ることができる。これらの手法と強化学習の手法との違いはなんだろうか。

- 強化学習のエージェントが取れる点数をゲームの「難しさ」だと考えることで、ゲームの難易度調整をしてみよう。例えば、プレイしているうちにだんだんと難易度がユーザにとって最適なものになっていくゲームが作れないだろうか？
- 2017 年 10 月 19 日に Google DeepMind が強化学習で（人間の棋譜を用いずに）囲碁の強いエージェントを作ること成功したと発表した (<https://deepmind.com/blog/alphago-zero-learning-scratch/>)。この論文を読み、AlphaGo Zero がどのようにして学習を行っているのかを理解し、まとめてみよう。OpenAI Gym の枠組みで AlphaGo Zero は実装できるだろうか？ もしできないとしたら、それはなぜか？

Image Recognition

第 4 章

4.1 CNN とは

主に全結合層，畳み込み層，プーリング層から構成される Deep Neural Networks (DNN) のことである．各層はニューロンによって構成される．

4.1.1 全結合層

全結合層 (FC 層) に含まれる各ニューロンは前の層の全てのニューロンと連結している．前の層の出力を h^{i-1} とすると重み W の FC 層の出力 h^i は

$$h^i = Wh^{i-1} + b.$$

b は定数である．下の図 4.1 と 4.2 はそれぞれ FC 層の構造とどのようにニューロンの値を計算するかを示したものである．定数 b も学習対象ではあるが省略している．

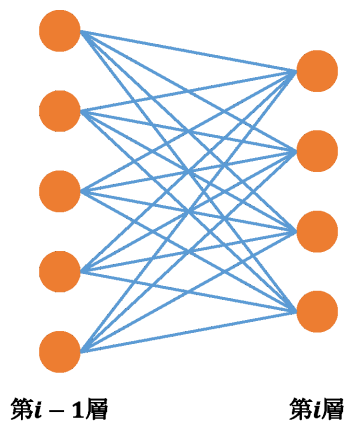


図 4.1 FC 層の構造

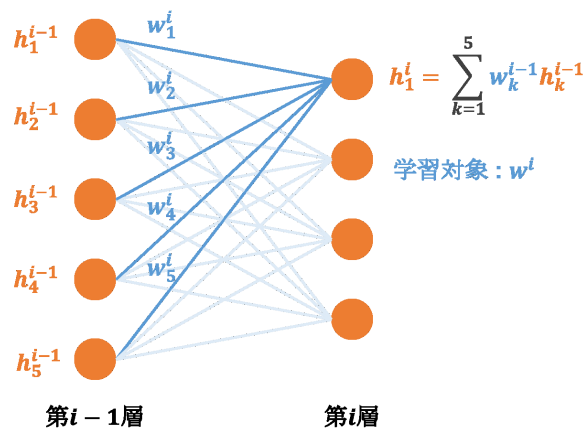


図 4.2 FC 層の計算と学習対象

4.1.2 畳み込み層

畳み込み層はカーネルと呼ばれる小さな受容野をもとに結合していくため前層のニューロンに対して疎結合となる．これは局所特徴に注目することに相当する．言葉で説明すると分かりづらいが図 4.3 と図 4.4 を見てほしい．いくつかの kernel を畳み込むことで新しい層を作っている．第 i 層のサイズは第 $i-1$ 層のサイズと padding, kernel のサイズと stride 幅によって決まる．padding というのは第 $i-1$ 層の周囲に適当な数字をくっつけてサイズを大きくするためのもので

ある. kernel のサイズは縦と横の大きさのことであり, チャンネル (厚み) は第 $i-1$ 層と同じである. stride 幅は畳み込む時に kernel を移動させる幅のことを意味し, 大きければ大きいほど第 i 層のサイズは小さくなる.

また第 i 層の厚み, すなわちチャンネル数は kernel の数と等しい. これは図 4.3 と 4.4 をじっくり見比べてもらえばわかると思う.

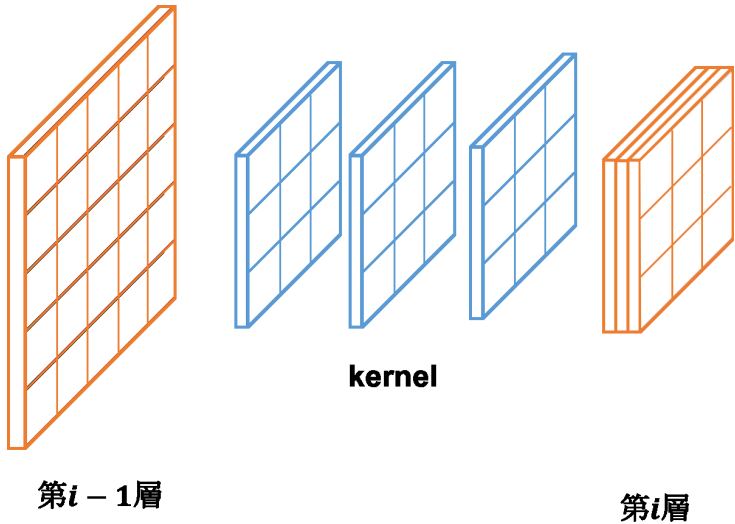


図 4.3 畳み込み層の構造

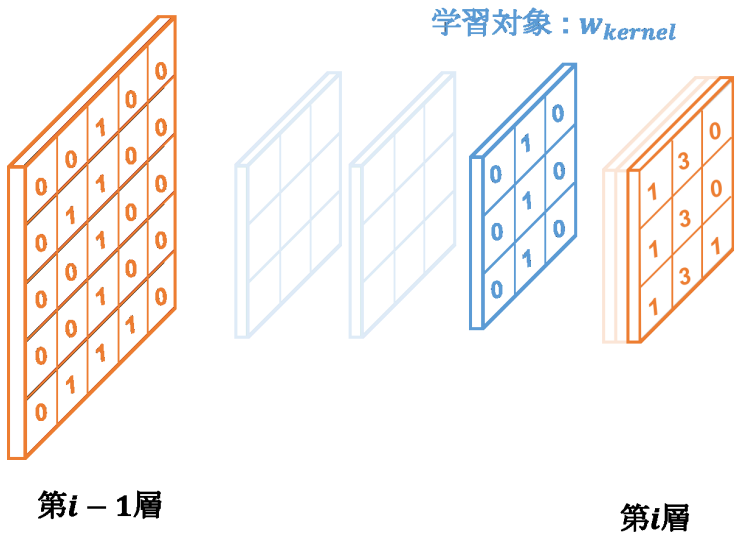


図 4.4 畳み込み層の計算と学習対象

4.1.3 プーリング層

畳み込み層と似ているがパラメータは固定されており, 学習対象は存在しない. 図 4.5 は kernel が見ている範囲の平均値を出力するものである (average pooling と呼ばれる).

プーリング層の目的は画像内の特徴に関して, 位置情報を曖昧にするためである. プーリングを行なうことで位置の変化に対して頑健な特徴表現を学習できる. 平均値を求めるもの以外にも注目領域内の最大値を出力する max pooling などが存在する.

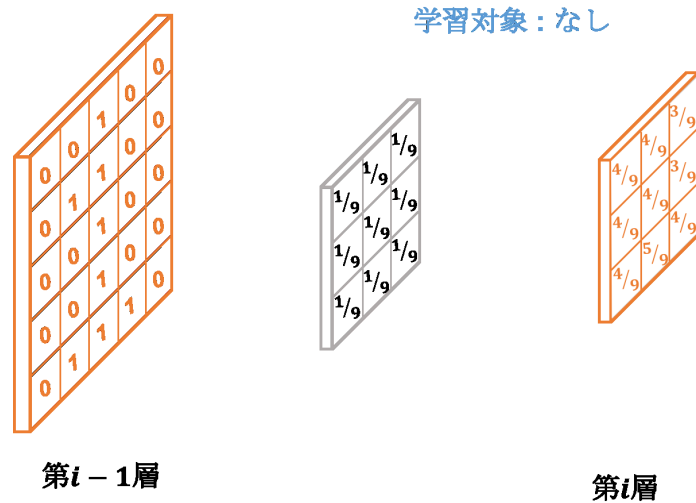


図 4.5 プーリング層の計算

4.1.4 活性化関数

FC 層や畳み込み層は結局のところ線形変換であり，このままでは豊かな変換能力を習得できない（線形変換を重ね合わせたところで表現能力は増さない）ため，殆どの場合，各ニューロンに対して活性化関数を適用することで非線形変換を学習させる．図 4.6 は FC 層に対して活性化関数を適用している図である．学習対象は何もないがこれにより非線形変換を表現できる． ϕ には sigmoid 関数や ReLU 関数がよく用いられ，特に画像分野では ReLU 関数が注目を集めている．

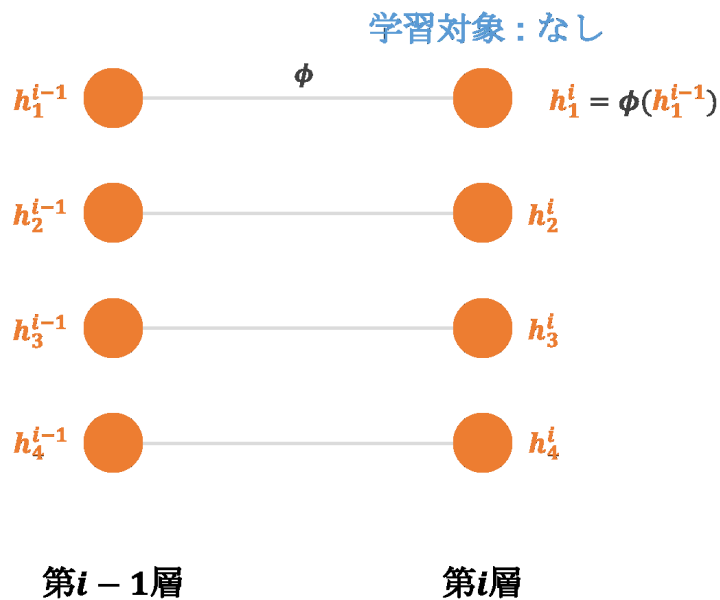


図 4.6 活性化関数

4.1.5 畳み込み層と全結合層

畳み込み層は 1 次元的なイメージなのに対し畳み込み層は 3 次元であるためその結合がイメージしづらいと思う。そこで図 4.7 を見てもらえればまだなんとなくイメージがつかめるのではないだろうか。図は channel 数 3, 高さ 2, 幅 2 の 3 次元の第 $i-1$ 層からサイズ 5 の FC 層への連結方法を示している。図が煩雑になるので第 $i-1$ 層の 1channel 目からしか線を伸ばさなかったが要は全ニューロン同士が連結していることを掴んでもらえば問題ない。

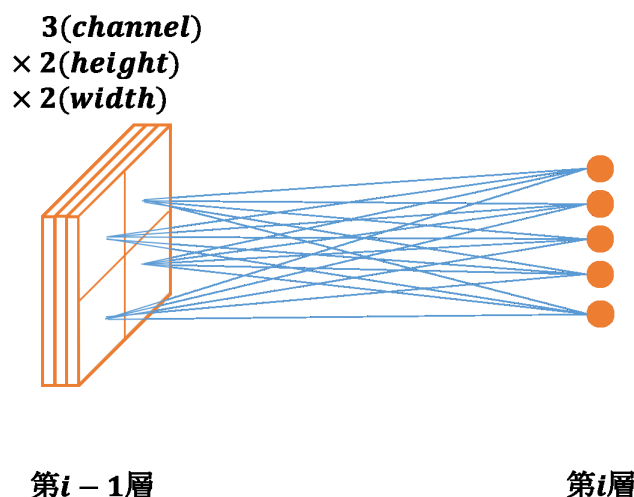


図 4.7 畳み込み層から全結合層へ

以上のことは <https://qiita.com/icoxfog417/items/5fd55fad152231d706c2> や https://deepage.net/deep_learning/2016/11/07/convolutional_neural_network.html により、詳細に説明されている。上記以外にも DropOut や Batch Normalization 等様々な手法が提案されている。調べてみたり TA に聞いてみたりしよう。

また有名な CNN である VGG16 や ResNet 等の構造を調べてみよう。

4.2 chainer の基礎

本章では、deeplearning のフレームワークの一つである chainer について基本的なコードを用いて説明する。

4.2.1 開発環境について

ssh について

配布した user 名、パスワードで以下のコマンドで開発環境にログインできる。

```
ssh Username@IPAddress
```

yes/no を聞かれるので、yes と入力した後、配られたパスワードを入力することでログインできる。

この工程を簡略化したい場合は、<https://qiita.com/ir-yk/items/af8550fea92b5c5f7fca>, <https://qiita.com/passol78/items/2ad123e39efeb1a5286b> などを参考にすること。

scp について

コードを編集する際、リモートではなく、ローカルで編集しその結果をリモートに反映させたいという必要があると思う。その際は、scp コマンドを用いる。

```
scp LocalFilePath Username@IP:RemotePath
```

と打てば、ローカル→リモートにファイルの転送ができる。その他の使い方については、<https://qiita.com/katsukii/items/225cd3de6d3d06a9abcb>などを参考にすること。

また、エディタによってはこういった機能をサポートしてくれるものもある。”””使っているエディタ sftp”””などで検索すると良い。

tmux について

開発環境でコードを実行していてもそのままだと、パソコンを閉じる、開発環境と接続を切るなどすると実行が終了してしまう。それを防ぐのが tmux である。tmux 上で実行すればこういった問題を解決できる。詳細については、<https://qiita.com/vintersnow/items/be4b29652ff665c45198>などを参考にすること。

4.2.2 サンプルコードの実行

今回扱うサンプルコードは mnist と呼ばれる手書き数字の分類を行うものである。

演習 4.2.1 train_mnist_mlp.py を GPU で実行せよ。

次のコマンドを打つと学習を開始できる。

```
python train_mnist_mlp.py -g 0
```

数分待つと、学習が終了し、result フォルダに結果が出力されている。

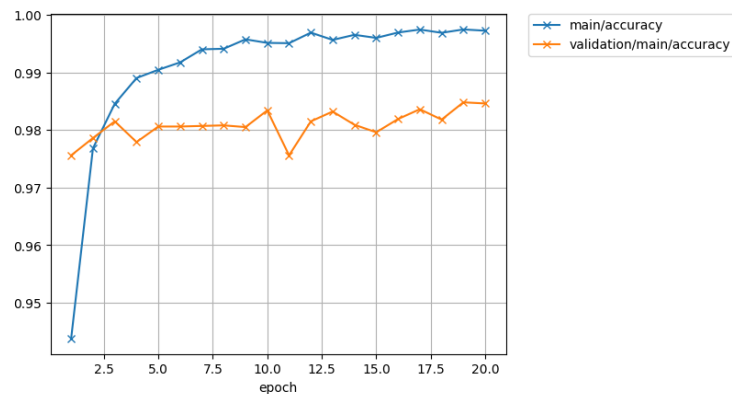


図 4.8 学習結果の例

次に、学習がうまくできたか確認する。

演習 4.2.2 白黒の手書き数字画像を作成せよ。この時黒地に白文字で数字を書くことに注意する。(学習させた手書き数字データが黒地に白なので)

演習 4.2.3 自分が書いた手書き文字を先程学習したモデルに通して出力を表示し、正しく数字が認識されたか確かめよ。次のコマンドでテストコードを実行できる。

```
python test_mnist_mlp.py -g 0 -i (path to the input image) -m result/model_20
```



図 4.9 手書き数字の例

4.2.3 サンプルコードの解説

先の章で実行したサンプルコードについて簡単に解説を行う。chainer の公式ドキュメント (<https://docs.chainer.org/en/stable/>) を適宜参考にすること。

コマンドライン引数

```
parser = argparse.ArgumentParser(description= 'Chainer example: MNIST' )
parser.add_argument( '--batchsize' , '-b' , type=int, default=100,
                    help= 'Number of images in each mini-batch' )
parser.add_argument( '--epoch' , '-e' , type=int, default=20,
                    help= 'Number of sweeps over the dataset to train' )
parser.add_argument( '--frequency' , '-f' , type=int, default=-1,
                    help= 'Frequency of taking a snapshot' )
parser.add_argument( '--gpu' , '-g' , type=int, default=-1,
                    help= 'GPU ID (negative value indicates CPU)' )
parser.add_argument( '--out' , '-o' , default= 'result' ,
                    help= 'Directory to output the result' )
parser.add_argument( '--resume' , '-r' , default= '' ,
                    help= 'Resume the training from snapshot' )
parser.add_argument( '--unit' , '-u' , type=int, default=1000,
                    help= 'Number of units' )
args = parser.parse_args()
```

このプログラムで使うことのできるコマンドライン引数を定義している。各変数に対する説明は省略する。

モデル

```
model = MLP(args.unit, 10)
if args.gpu >= 0:
    # Make a specified GPU current
    chainer.cuda.get_device_from_id(args.gpu).use()
    model.to_gpu() # Copy the model to the GPU
```

この行で今回の学習に用いるモデルを定義している。また、GPU を使用していたらモデルを GPU に変更している。args.gpu は使用する GPU の番号を示している。MLP 自体は以下で定義している。

```
class MLP(chainer.Chain):
    def __init__(self, n_units, n_out):
        super(MLP, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(None, n_units) # n_in -> n_units
            self.l2 = L.Linear(None, n_units) # n_units -> n_units
            self.l3 = L.Linear(None, n_out) # n_units -> n_out

    def __call__(self, x, t=None):
        h = F.relu(self.l1(x))
        h = F.relu(self.l2(h))
```

```

        h = self.l3(h)

        loss = F.softmax_cross_entropy(h, t)
        accuracy = F.accuracy(h, t)
        chainer.report({'loss': loss}, self)
        chainer.report({'accuracy': accuracy}, self)

        if chainer.config.train:
            return loss
        else:
            return h

    def predict(self, x):
        h = F.relu(self.l1(x))
        h = F.relu(self.l2(h))
        h = self.l3(h)
        return h

```

init 関数で今回使うネットワークの構造を定義しており、今回は入力層 (l1), 中間層 (l2), 出力層 (l3) の全 3 層である。今回用いた層は全て全結合層である。x がネットワークの input, h がネットワークの output, t が input の正解ラベルである。train の時は loss を返し学習をさせており、validation の時は学習をさせたくないで h を返している。(validation の時も学習させてしまうと validation の accuracy が正当な評価にならない。)

optimizer

```

# Setup an optimizer
optimizer = chainer.optimizers.Adam()
optimizer.setup(model)

```

この部分では、loss を最適化するための手法を決めておりこのプログラムでは Adam を用いている。他の最適化手法を用いたい場合は <https://docs.chainer.org/en/stable/reference/optimizers.html> などを参考にすること。

データセット

```

# Load the MNIST dataset
train, test = chainer.datasets.get_mnist()

```

この部分では、今回の学習に用いるデータセットを定義している。今回用いた mnist のデータセットは chainer が用意してくれているのでこのように 1 行書くだけで呼び出せる。自分で用意したデータセットを用いたい場合については後の演習に譲る。

Trainer の準備

```

train_iter = chainer.iterators.SerialIterator(train, args.batchsize)
test_iter = chainer.iterators.SerialIterator(test, args.batchsize,
                                              repeat=False, shuffle=False)

# Set up a trainer
updater = training.StandardUpdater(train_iter, optimizer, device=args.gpu)
trainer = training.Trainer(updater, (args.epoch, 'epoch'), out=args.out)

```

これまでに定義したものを用いて学習のための Trainer を作成している。

model の評価と結果の可視化

```

# Evaluate the model with the test dataset for each epoch
trainer.extend(extensions.Evaluator(test_iter, model, device=args.gpu))

```

```

trainer.extend(extensions.dump_graph( 'main/loss' ))

frequency = args.epoch if args.frequency == -1 else max(1, args.frequency)
trainer.extend(extensions.snapshot(), trigger=(frequency, 'epoch'))
trainer.extend(extensions.snapshot_object(model, 'model_{.updater.epoch}'),
                trigger=(frequency, 'epoch'))

trainer.extend(extensions.LogReport())
if extensions.PlotReport.available():
    trainer.extend(
        extensions.PlotReport([ 'main/loss' , 'validation/main/loss' ],
                               'epoch' , file_name= 'loss.png' ))

    trainer.extend(
        extensions.PlotReport(
            [ 'main/accuracy' , 'validation/main/accuracy' ],
            'epoch' , file_name= 'accuracy.png' ))

trainer.extend(extensions.PrintReport(
    [ 'epoch' , 'main/loss' , 'validation/main/loss' ,
      'main/accuracy' , 'validation/main/accuracy' , 'elapsed_time' ]))

trainer.extend(extensions.ProgressBar())

```

ここでは、学習結果の出力と可視化を主に行なっている。extensions.snapshot_object 関数は、学習したモデルを出力する関数で、trigger はその出力を何 epoch 毎に行うかを決定している。他の関数については説明を省略する。

ここまでが、train_mnist_mlp.py の大まかな解説である。次に test_mnist_mlp.py の解説を行う。

訓練モデルの読み込み

```

model = MLP(args.unit,10)
serializers.load_npz(args.model, model)

```

先ほど学習したモデルを読み込んでいる。先に学習したモデルと同じ形のモデルを作っておかないと読み込めないことに注意する。

画像の読み込み

```

try:
    img = Image.open(args.image).convert( "L" ).resize((28,28))
except :
    print( "invalid input" )
    return

```

画像を読み込み、mnist と同じサイズ (白黒, 28*28) に変換している。

結果の出力

```

img_array = np.asarray(img,dtype=np.float32).reshape(1,784)
result = model.predict(img_array)
print( "predict: ", np.argmax(result.data))

```

今回使用したネットワークは画像を2次元の形で入力するのではなく、1次元に潰して入力するので 28*28 を 1*784 に reshape を行っている。モデルの出力は、[入力数字が0である確信度, 入力数字が1である確信度, 入力数字が2である確信度,...] となっているのでその配列の argmax を取ることで最も確信度が高い数字つまり予測数字を出力している。model.predict の返り値が Variable 型と呼ばれるものになっているので、.data を使うことで中身の配列にアクセスすることができる。

演習 4.2.4 `test_mnist_mlp.py` を書き換えることで予測数字を確信度が高い方から上位 3 つま
で出すように変更せよ。

4.2.4 ネットワークの変更

先の章で用いたネットワークは、中間層のノード数を変化させることができ、それによりモデルの精度、実行時間などが変化する。

演習 4.2.5 MLP の中間ノード数を

1. 3
2. 10000

にしてそれぞれ 20epoch 学習し、最終的な精度と学習にかかった時間を比較せよ。

MLP はネットワークに画像を入力する際、2 次元の画像を 1 次元に潰して入力してしまっている。それにより、隣接するピクセルの関係をういた学習が難しくなっている。そのため本来全結合のみのネットワークは画像の学習に適さない。そこで全結合層ではなく、畳み込み層を用いたネットワークで学習させることを考える。

演習 4.2.6 `net.py` を書き換えることで、下の図 4.10 のネットワークを `MNIST_CNN` クラスに実装せよ。ただし、カーネルサイズはどちらの畳み込み層についても (5×5) とする。`MNIST_CNN` クラスを実装したら、`train_mnist_cnn.py` を同様に実行することで精度を確かめよ。興味がある人は、`test_mnist_mlp.py` を参考にテスト用のコードを書き実行せよ。

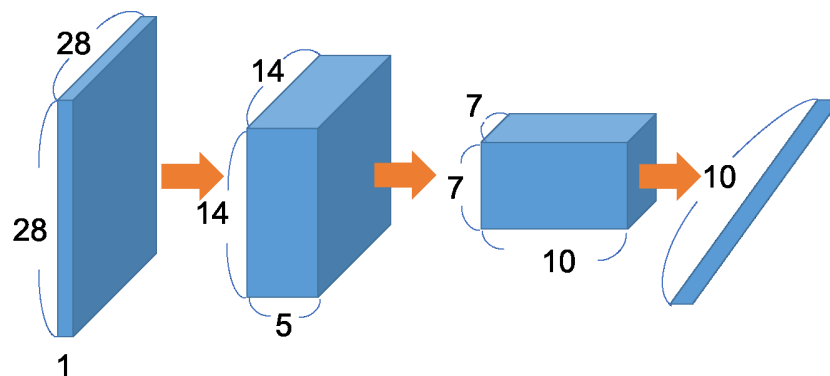


図 4.10 作成する CNN

ヒント：chainer のドキュメントで `Convolution2D` を検索せよ。

4.2.5 他のデータセットでの学習

先章では、mnist を題材として学習を行なったが、本章ではもう少し複雑なデータセットである CIFAR-10 を扱う。CIFAR-10 は 10 クラス、 32×32 のカラー画像のデータセットである。

演習 4.2.7 `dataset.py` を埋めることで、`cifar10` の画像を読み込むコードを完成させよ。CIFAR-10 の画像は `mini_cifar/train/` にある。

`dataset.py` を正しく書くことで以下のコマンドで CIFAR-10 が実行できるようになる。

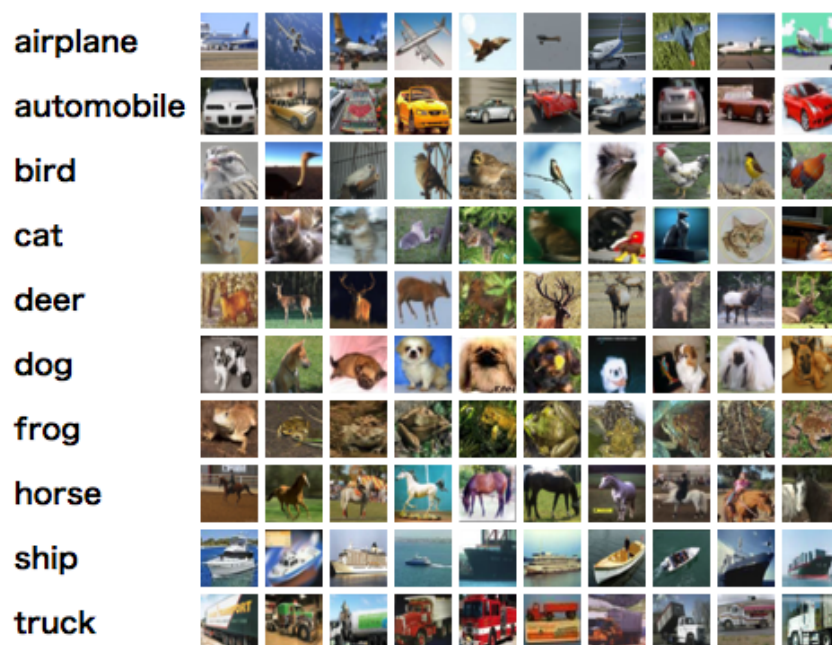


図 4.11 CIFAR-10 データセット

```
python train_cifar10_cnn.py -g 0
```

このプログラムは、mnist のプログラムとは異なり、1epoch ごとに model が出力させる。

演習 4.2.8 出力された model から最も精度が良いモデルを選び、その model を用いて、test_cifar10_cnn.py を実行せよ。以下のコマンドで test_cifar10_cnn.py を実行できる。

```
python test_cifar10_cnn.py -g 0 -m (path to your model)
```

演習 4.2.9 (発展課題) 本ネットワークの精度は約 30 ~ 40 パーセントとあまり良くない。精度を上げてみよ。例えば、ネットワーク構造や、最適化手法などを変化させて見ると良い。

4.3 CNN 応用分野

これより先は CNN の応用分野であり、必須課題ではない。画像分類以外にどのような方法で CNN が利用されているかをいくつか述べるものである。

4.3.1 深層特徴

第二章では最終層に softmax 関数をかけてクラスの確率分布を出力していた。CNN は最終層以外の中間層の出力も有用である。例えば以下の VGG と呼ばれる CNN では FC6 層、FC7 層の値を利用することが頻繁にある。

CNN は特徴抽出と分類/回帰を同時に行うモデルである。分類/回帰の結果は最終層に現れるが、特徴表現は隠れ層に出てくるとされている。何らかの課題を学習した CNN は、データのより良い表現 (特徴) の表現を隠れ層で学習している。そのため、データの良い特徴を得るという目的で「学習済みの CNN」にデータを通しその際の隠れ層の出力を取得する、ということがよく行われる。

CNN の特徴抽出

より正確には、CNN は最終的なタスク (分類/回帰) を行なうのに有用な特徴表現を学習する。例えばある画像を入力に「人物が写っているか/写っていないか」の 2 値分類を CNN で解く場合、恐らく CNN はそこに鳥がいるか空が写っているかなどの情報は無視して、人間の存在を分けるような特徴の表現を学習するだろう。

深層特徴の利用方法

単純に分類問題や回帰問題を解く場合は、end-to-end なネットワークを構築して学習した方が精度が高い。しかし、ネットワーク上に問題を落とし込めない時、深層特徴だけを利用するケースがある。例えば、画像検索などがその良い例である。ある画像と似た画像を探したい。単純に考えると、他の画像とのピクセルレベルでの距離を計算し、近いものから取得してくるという方法が挙げられる。しかしこれには 2 つの問題がついてくる。それは

- ピクセルレベルの距離が近いものが欲しい画像とは限らない
- 検索に時間がかかる

例えば左上に赤いリングが移った図のような写真を入力にして、他のリング画像を得ようとする。しかし、図とのピクセルレベルでの距離は同じリングが写っているにも関わらずピクセルレベルの値の距離は大きい。これはリングが写っている位置が異なるために生じてしまう。また、この画像が仮に幅 640px、高さ 480px だとすると $640 \times 480 = 307,200$ の距離計算を行なう必要がある。

そこで、深層特徴を用いることで画像をより小さいサイズでより高次元の情報を保持した特徴に変換することで検索性能を高めることができる。先程述べた通り深層特徴は中間層の出力であり、よく用いられるネットワーク (VGG, AlexNet, ResNet など) ではおよそ 1,024-4,096 次元程度のコンパクトなベクトルを得ることが出来る。また深層特徴は (ネットワークを訓練したタスクによるが) 単純な RGB ではない高度な情報を保持する。この画像にどのようなものが写っているか、のような情報である。これにより色が似ているといった単純な情報以外を利用した検索が可能になる。

深層特徴の抽出方法

単純にネットワークを組んでデータを入れて中間層の値を得たとしてもそれは有用な情報ではない。ネットワークのパラメータがデタラメなため何の意味もない値が出てくるだけである。有用な深層特徴を得ようと思ったら、CNN を訓練しておく必要がある。

訓練の仕方は様々である。一般的には「深層特徴を利用するケースとなるべく近い形で DNN を訓練する」というのが基本である。例えば先程の「リング画像を検索したい」という例では「リングが写っているか否か」の DNN を訓練しその中間層を利用するという手がある。しかし普通はもう少し幅の広い課題を解くことが多く、また DNN の訓練に必要な大規模データも手に入ることは少ない。そこで ImageNet というデータセットを利用して、ImageNet データセットの分類問題を学習した DNN の中間層出力を深層特徴として利用することが多い。ImageNet は大量の画像を「人」「車」「鳥」のようなそこに写っているもので分類したデータセットのことであり、このデータセットで訓練した DNN モデルは他の領域にも応用しやすい汎用的な特徴表現能力を得ることが知られている。

そのため、「取り敢えず深層特徴を使いたい!」という場合は有名なネットワークを ImageNet で訓練してその中間層出力を利用するケースがほとんどである。

次の章で実際に深層特徴を用いて画像検索を行なう例を見てみる。

深層特徴抽出の実装

長々書いてきたが要は深層特徴とは「データから抽出したい感じの特徴」であり、非常に有用なものである。そのため Chainer では深層特徴を非常に簡単に得る事ができるように設計されている。

実装課題

search.py と create_db.py を使用します

演習 4.3.1 search.py の argparse 部分をよく読んで cifar10/test/以下から適当に画像を選んで search.py を CPU 上で実行せよ。

演習 4.3.2 search.py の中で深層特徴 (src_df) を計算する部分を探し、深層特徴の shape を表示せよ。

演習 4.3.3 create_db.py の中で深層特徴として使用する隠れ層を指定している部分を探し、VGG16 の fc6 層を利用するよう書き換えよ。その後もう一度 search.py を実行せよ。

演習 4.3.4 MNIST で MLP を訓練し、その隠れ層出力を深層特徴として利用した場合の検索結果を表示せよかなり難易度は高いので TA を利用してください。

4.3.2 GAN

詳細は触れないが CNN は分類/回帰以外にも利用可能である。一例として画像生成技術である Generative Adversarial Network, 通称 GAN を挙げる。GAN は generator と discriminator という 2 つの CNN を用いる。generator は何らかの適当な入力から 1 枚の画像を生成する CNN である。discriminator は入力された画像が generator が作ったものか実際の画像かを判定する CNN である。訓練の最初の方では generator はわけの分からない画像を出力する。しかし discriminator もあまり賢くないので generator の画像と実際の画像を見分けることが出来ない。その後、訓練が進むと discriminator も賢くなり、generator から出てきた画像を見抜けるようになる。そこで、generator の損失関数をうまく設定することで、generator は discriminator を騙せるような画像を生成させるように訓練する。

generator は discriminator を騙せるような画像を生成しようとし、discriminator は generator から出てきた画像を見抜こうとする。お互いがお互いを刺激しあって学習することで generator はより本物っぽい画像を生成することが出来る様になる。

Deep Convolutional GAN (通称 DCGAN) の実装は chainer/examples/dcgan の中にある。generator と discriminator の定義を見つけてみよう。また、このコードでは updater を自作している。updater はバッチを受け取ってモデルに通し、損失を計算してモデルの更新を行なうよう支持する場所である。updater は StandardUpdater を継承して update_core 関数を上書きして作るのだが、もし特殊な学習を行いたければここを参考にすると良い。

4.3.3 Super Resolution

画像生成とは微妙に異なるが、CNN を利用して低画質の画像を高画質に変換しようとするものもある。低画質の画像にはない情報を推測するのももちろん正確な高画質画像を作ることは不可能だが、GAN と組み合わせて利用するといかにも高画質画像に見える画像を生成することが出来る。

Audio and Speech Processing

第 5 章

音声情報処理においても、深い構造のニューラルネットワークを用いた深層学習技術は、様々なタスクで用いられており、大きな成果をあげている。音声情報処理では音声に内在する信号処理的な側面と言語処理的な側面の双方を扱う必要があり、それぞれにおいて画像処理や自然言語処理において用いられている深層学習の知見が活用できる。本章では、数ある音声情報処理技術の中から、主に音声から音声への変換である声質変換という技術について取り上げ、その中での深層学習の寄与について説明する。

5.1 声質変換

声質変換は、入出力の対応関係を記述する変換モデルに基づいて、任意の文に対して入力音声の声質を所望の声質へ変換する技術である。特に入力発声の話者性を制御する話者変換はテキスト音声合成への応用や福祉応用を目的として数多く研究されている。簡潔に述べるとすれば、Aさんが喋った発話音声を、その文章内容を変化させずにBさんが喋ったような発話音声へと変化させるのが声質変換である^{*1}。

声を変化させるといった場合に、最初に思いつくのは再生速度を変化させることであるが、その場合、音の“高さ”と“太さ”を独立にコントロールするといったことは難しい。これらを取り扱う際には、音の“高さ”に相当する成分と音の“太さ”に相当する成分とに分解したのち、これらの特徴量を取り扱う方法が一般的である^{*2}。人間の音声生成過程に着目すると、声帯振動によって駆動された信号源が口腔内の空洞（声道と呼ばれる）を通ることでその空洞形状の影響が畳み込まれ、口から放射されるという過程を経て人の音声が生まれる。これをソースフィルタモデルという^{*3}。加えて声帯振動の信号はパルス列で近似され、このパルス列の幅が基本周期に対応して音の高さとして知覚されることになる。一方で声道特性の違いが言語的な内容の違い（音韻の違い）や話者の違いを生む。声質変換では、音声を入力した話者のこれらの特性を出力した話者の対応する特性に変化させることが必要となる。

5.2 回帰問題

上述の生成過程の背景を前提とすると、声質変換の問題は入力 \mathbf{x} が与えられたときに出力 \mathbf{y} を求める、回帰と呼ばれる機械学習の問題と考えることができる。回帰問題は教師データとして (\mathbf{x}, \mathbf{y}) のペアが与えられたときにその関係をモデル化する教師あり学習の一種である。

^{*1} 有名なアニメで出てくるリボン型のネクタイをご想像いただければよい。

^{*2} 近年では波形そのものを深層学習で取り扱うアプローチも登場している。興味があれば調べてみるとよい。

^{*3} ソースフィルタモデルの詳細は言語・音声情報処理の講義や後期実験テキストの音声対話システムを参照するとよい。

5.2.1 線形回帰

まずはいきなり深層学習を用いる前に、 \mathbf{x} と \mathbf{y} の間に線形関係が存在し、データ観測時にノイズが加わったケースを考えてみよう。すなわち以下のような生成過程で \mathbf{y} が観測されると仮定する。

$$\mathbf{y} \sim \mathbf{A}\mathbf{x} + \mathbf{b} + \epsilon \quad (5.1)$$

ただし、 ϵ は平均 0 分散 $\sigma\mathbf{I}$ の正規分布に従ってランダムに付与される。このような生成過程で N ペアの学習データ $(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_N, \mathbf{y}_N)$ が与えられた時に \mathbf{A} および \mathbf{b} の値を推定する。この場合は、最小二乗誤差基準を用いて推定することで反復なしで求めることができ、

$$\mathbf{A} = \left(\sum_{i=1}^N \mathbf{y}_i \mathbf{x}_i^\top \right) \left(\sum_{i=1}^N \mathbf{x}_i \mathbf{x}_i^\top \right)^{-1} \quad (5.2)$$

$$\mathbf{b} = \left(\frac{1}{N} \sum_{i=1}^N \mathbf{y}_i \right) - \mathbf{A} \left(\frac{1}{N} \sum_{i=1}^N \mathbf{x}_i \right) \quad (5.3)$$

となる *4。

5.2.2 ニューラルネットワークを用いた回帰

線形回帰では、例えば 1 次元の場合を考えると直線的な対応づけしか求めることができない。一方でニューラルネットワークは、中間素子に含まれる活性化関数の非線形性によって、理論的には任意の関数を近似することが可能である。この万能関数近似としてのニューラルネットワークを用いて \mathbf{x} と \mathbf{y} の関係をニューラルネットワークとして直接表現してしまうのがニューラルネットワークによる回帰である。

5.3 回帰アプローチを声質変換に適用する場合

回帰アプローチを声質変換に適用する場合にはいくつか注意を要する。

5.3.1 適用する特徴量

声道特性は音声の短時間フーリエ解析によって得られたスペクトルのうち、声道特性に対応するスペクトル包絡部分がモデル化される。スペクトル包絡そのものを取り扱うアプローチの他、聴覚特性を考慮した低次元表現であるメルケプストラムが特徴量として用いられる。回帰の学習では、入力話者と出力話者のメルケプストラムの対応関係を学習する。

5.3.2 声の高さに対する適用

声の高さは、前述の通り、声帯振動の音源をパルスで近似した際の、パルスの周期に対応する。音声ではこの逆数をとったものを基本周波数 F_0 という。一方声帯を振動させずに発声された音声（例えば、無声摩擦音である /s/ など）では、基本周波数は定義されない。そのため、有声音として基本周波数が定義された時間区間のみについて変換を適用する。基本周波数についてはその対数をとった上で、一次元の値について単に非常に単純な線形回帰を適用することも多い。

*4 自身の手で一度導出してみるとよい。

5.3.3 発話長の違いの取り扱い

声質変換においては入力と出力について、同一内容の発話を別の話者が発声したデータ（翻訳と同様これを「パラレルデータ」と呼ぶ）を用いることが多い。この際、話速の違いなどが存在するため、 $(\mathbf{x}_t, \mathbf{y}_t)$ の対応するデータを直接作することは難しい。機械翻訳と同様にエンコーダ・デコーダモデルを使用するアプローチも存在するが、単純にフィードフォワードネットワークを採用する際には、動的時間伸縮（Dynamic Time Warping）と呼ばれるアプローチによって入出力間の時間的な対応づけをあらかじめとっておく。

5.4 演習

5.5 その他の音声・音響分野におけるタスクやアプローチ

参考文献

- [1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, 2015.
- [2] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.
- [3] Kumar et al. Ask me anything: Dynamic memory networks for natural language processing. In *Proceedings of ICML*, volume 48, pages 1378–1387, 20–22 Jun 2016.
- [4] Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.
- [5] Silver et al. Mastering the game of Go without human knowledge. *Nature*, 550:354–359, October 2017.
- [6] Sutskever et al. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.
- [7] Wu et al. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv:1609.08144*, 2016.
- [8] K. Fukushima. A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):93–202, 1980.
- [9] K. Fukushima, S. Miyake, and T. Ito. Neocognitron: a neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(3):826–834, 1983.
- [10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [11] Allison Linn. <https://blogs.microsoft.com/ai/2017/06/14/divide-conquer-microsoft-researchers-used-ai-master-ms-pac-man>.
- [12] Tomas Mikolov, Martin Karafiat, Lukas Burget, Jan Cernocky, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, pages 1045–1048, 2010.
- [13] David L. Poole and Alan K. Mackworth. *Artificial Intelligence: Foundations of Computational Agents, 2nd Edition*. Cambridge University Press, 2017.
- [14] Alexander M Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. In *Proceedings of EMNLP*, 2015.
- [15] Oriol Vinyals and Quoc Le. A neural conversational model. *arXiv preprint arXiv:1506.05869*, 2015.
- [16] 福島邦彦. 位置ずれに影響されないパターン認識機構の神経回路モデルーネオコグニトロンー. 電子情報通信学会論文誌 A, J62-A(10):658–665, 1979.

謝辞

本実験は Microsoft の支援のもと，Microsoft Azure を使用して行われた．Azure サーバの構築等は付録を参照のこと．