

無線通信のための深層学習の基礎

名古屋工業大学

和田山 正

目次

第 1 章	はじめに	5
第 2 章	深層学習に関する基礎的事項	7
2.1	2 値推定関数の学習	7
2.2	深層ネットワークモデル	11
2.3	確率的勾配法（ミニバッチ学習法）	14
2.4	誤差逆伝播法	16
2.5	フレームワーク	19
第 3 章	PyTorch 入門	23
3.1	PyTorch の実行環境	23
3.2	Google Colaboratory	24
3.3	PyTorch のチュートリアルを実行してみる	25
3.4	2 入力 AND 関数の学習	27
第 4 章	深層学習を利用した無線通信技術	31
4.1	ブラックボックスモデル	31
4.2	深層展開	40
4.3	最適化模擬	46
4.4	分散学習	51
第 5 章	深層展開	55
5.1	勾配法への深層展開の適用	55
5.2	射影勾配法への深層展開の適用	58
5.3	近接勾配法とISTA	61
5.4	ISTA の登場	64
5.5	TISTA について	65

5.6	複素学習型ISTA	66
5.7	近接法と深層展開	68
第6章	むすび	71
第7章	付録	73
7.1	文献ガイド	73
7.2	LDPC 符号関連のサンプルコード	75
7.3	実験の再現可能性を高めるために	75
参考文献		79

第1章

はじめに

近年、深層学習技術 [9]、ならびにその応用に関する研究が様々な分野において、爆発的な勢いで進展中です。その理由のひとつとして、人間の得意とするパターン認識分野において、深層学習技術が高い潜在能力を持ち、従前のパターン認識技術に対して圧倒的な優位性を示すことが明らかになってきた点が挙げられます。画像認識 [2]・音声認識 [12]などの分野においては、人間の認識能力に匹敵する、あるいは凌駕する認識性能を与える認識アルゴリズムが報告されています。また自然言語処理（テキスト分類・機械翻訳など）の分野でも、深層学習技術は広く活用され始めています。

深層学習技術がもたらしたブレークスルーは、従前では機械学習技術と関連が薄いと思われていた分野にも急速に波及しつつあります。無線通信分野においても新しい研究の方向性として、深層学習技術の積極的な適用の機運が急速に高まってきています。多くの通信・信号処理関連の国際会議において、機械学習と関連のある研究発表やワークショップ・チュートリアルが急増中で多くの参加者の興味を惹きつけています。

無線通信分野において、求められている技術はますます高度化の一途をたどっています。スタートが間近である 5G やその先に来るであろう 6G で利用が見込まれるミリ波（さらにその先のテラヘルツ波）を利用するセルラ通信では、高度なビーム形成・ユーザ位置推定・電力等リソース割り当て・スペクトルセンシングなどの実現が求められています。これらの実現のためには、リアルタイムに大規模な凸最適化問題、または非凸最適化問題を解くことが求められます。また、無線通信品質を左右する信号検出アルゴリズム、例えば、ミリ波 Massive MIMO 信号検出や大量のパイロット信号が必要とされないブラインド型通信路推定、より高度化した誤り訂正符号の復号などに対する技術的要件も将来的により厳しくなることが予想されます。

無線分野と機械学習の結びつきの強まりのもうひとつの理由は、社会全体の AI・機械学習技術への需要の増加です。自動運転・デバイス間通信など自律的システムの台頭により、AI 技術と無線通信技術は、近い将来より密接な連携を深めることが予想されます。そ

のような時代においては、大規模な分散 AI 技術を支えることが無線通信の非常に重要なタスクになるとともに、逆に分散 AI 技術に支えられた高度機能を有する無線ネットワークが登場するでしょう。それほど遠くない未来において、機械学習技術と無線技術は、いわば表裏一体・渾然一体の形で社会の根幹に根付いていくのではないでしょうか。

本稿は、深層学習技術を利用した無線通信技術（特に物理層）の研究に興味を持つ若手研究者・初学者・学生^{*1}を主な対象として想定しています。特にこの分野に興味があるものの、どこから手をつけるのが良いのか分からぬという方にとって役立つ

「無線通信 × 深層学習」 研究スタートガイド

を目指しています。最近、PyTorch, Tensorflow/Keras などのフレームワークの発展は著しく、機械学習を専門としない研究者が深層学習技術を利用するための敷居が非常に下がっています。その意味では、先進のフレームワークの力を借りれば機械学習を専門としない研究者にとっても、深層学習技術を利用した研究を立ち上げるのは容易な状況になってきています。

本稿では、PyTorch を使った具体的なプログラムを提示しながら、無線通信への深層学習の適用例を説明していきます。深層学習関連の研究では実験の重要性がとても高く、まずは「適切に実験ができるようになる」ことが最重要であると言ってもよいと思います。その意味から、本稿ではこの形式を選択しました。

本稿の内容に関連する PyTorch を使った参考コード類 (Jupyter Notebook 形式) を Github のレポジトリ

<https://github.com/wadayama/MIKA2019>

に置いています^{*2}。本稿を読みながらプログラムを実行したり、プログラムを修正して自分の興味のあるアルゴリズムで試してみたりすれば、関連研究のスタート地点に最短でたどり着くことができるのではないか、と考えています。参考コードは、Google Colaboratory すぐに実行できますので、PyTorch のインストールなど準備・環境整備も必要ありません。お気軽に試してみてください。

本稿がこの分野に興味を持たれる方のご参考になれば幸いです。

和田山 正

^{*1}もちろん、分野横断的研究を模索されているベテラン研究者の方にも有益な情報も含まれます。

^{*2} 実はこの参考コード類が主役で逆に本稿が「付録」的位置づけかもしれません。

第 2 章

深層学習に関する基礎的事項

本章では、本稿の議論で必要とされる深層学習に関する基礎的事項を説明します。必要最低限のこと^{*1}しか書かれていませんので、さらに深層学習技術の詳細に興味のある読者は、7.1 章で紹介されている教科書等を参照されることをお勧めします。

2.1 2 値推定関数の学習

本節では、簡単な例を挙げて説明を行います。以下では、統計的性質が全く未知の通信路 $P(\mathbf{y}|x)$ が与えられているものと仮定します(図 2.1 参照)。

関数 $P(\mathbf{y}|x)$ は条件付確率密度関数を表し、 $\mathbf{y} \in \mathbb{R}^n$, $x \in \{+1, -1\}$ です。すなわち、送信器は $x = +1$ もしくは、 $x = -1$ の信号を通信路に送ります。そして受信器は、条件付確率密度関数 $P(\mathbf{y}|x)$ に従う受信ベクトル \mathbf{y} を受信します。また、受信側では送信シンボルを可能な限り正しく推定したいものとします。さらに、受信器はデータセット

$$D := \{(x_1, \mathbf{y}_1), (x_2, \mathbf{y}_2), \dots, (x_K, \mathbf{y}_K)\}$$

を保持していると仮定します。ここで、 \mathbf{y}_i は x_i に対応する受信語であり、 $\mathbf{y}_i \sim P(\mathbf{y}|x_i)$ です。

もし、 $P(\mathbf{y}|x)$ が完全に既知の状況ならば、最尤推定則

$$\hat{x} := \arg \max_{x \in \{+1, -1\}} P(\mathbf{y}|x) \quad (2.1)$$

に従い、尤度の最大化により誤り率最小の推定が可能です。しかし、上でおいた仮定では $P(\mathbf{y}|x)$ が未知であるので、最尤推定則を直接利用することはできません。ここでは、最尤推定則の代わりに、推定関数 $\hat{x} = f(\mathbf{y})$ をデータセットから学習するというアプローチ

^{*1} 歴史的コンテキストにもまったく言及していません。一度は、「本当の深層学習の教科書」をご覧いただくのがよいかと思います。

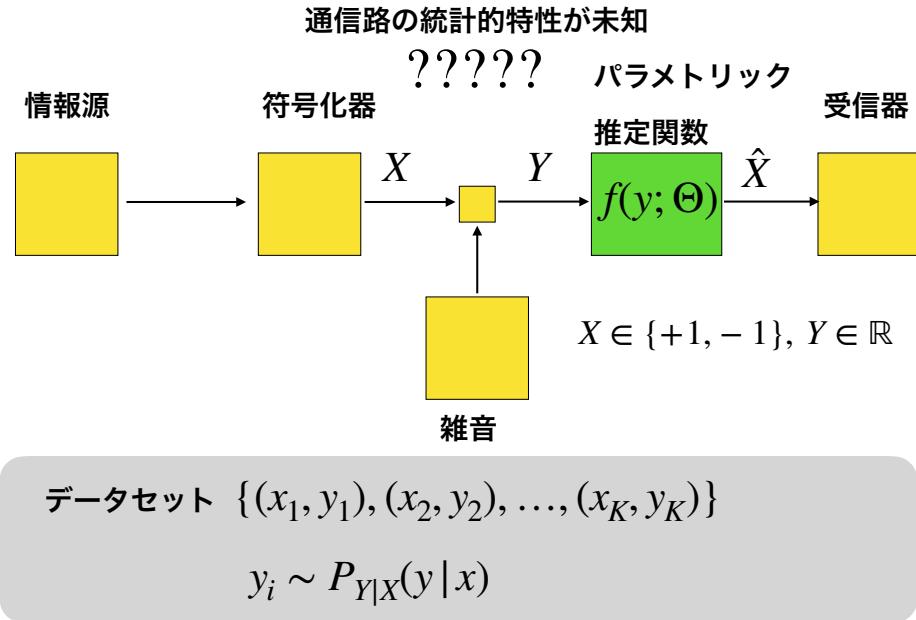


図 2.1 統計的性質が未知な通信路モデル: データセットは与えられており、それに基づきパラメトリック推定関数 $f(\mathbf{y}; \Theta)$ を学習します。

を考えます。この問題は、機械学習の文脈においては、**2 クラス判別問題**と呼ばれる問題設定にちょうど対応します。

以下では、推定関数 f が複数の学習可能パラメータの集合 Θ によりその形状が制御されるものとします。具体的な関数の内部構造としては、深層ネットワークモデルと呼ばれる深層ニューラルネットワークを利用しますが、その詳細は 2.2 節で説明します。深層ネットワークモデルに含まれる学習可能パラメータの値を変更することで関数 f の形状が変化します。以下では、学習可能パラメータへの依存を明示するため、 $\hat{x} = f(\mathbf{y}; \Theta)$ と表記することにしましょう。

学習プロセスとは、与えられたデータセット D に基づき、損失関数を最小化するようにパラメータ Θ を調整する過程を指します。言い換えると Θ が f の形状をコントロールする調節つまみのようなもので、損失関数を小さくするようにそれらを調整することができます。

深層学習においては一般的にミニバッチ学習法と呼ばれる手法が学習(訓練)プロセスにおいて利用されます。ミニバッチ学習法では、データセット D の部分集合 $D' \subset D$ (ミニバッチと呼びます)をランダムに選び出し、ミニバッチに基づき 2 乗誤差関数に基づく

損失関数を

$$E(\Theta; D') := \frac{1}{|D'|} \sum_{(x, y) \in D'} |x - f(y; \Theta)|^2 \quad (2.2)$$

と定義します^{*2}。この $E(\Theta; D')$ に基づき Θ を更新する処理を確率的勾配法と呼びます。確率的勾配法では、損失関数の勾配ベクトル $\nabla E(\Theta; D')$ に基づくパラメータ更新式

$$\Theta := \Theta - \alpha \nabla E(\Theta; D') \quad (2.3)$$

によりパラメータ Θ を更新します^{*3}。

ここで α は学習率（学習係数）と呼ばれる正実数です。学習・訓練プロセスが有効であるためには、適切に学習率を選ぶ必要があります^{*4}。以上の手続きを反復して実行した後に得られたパラメータを Θ^* とするとき、受信側では、推定則 $\hat{x} = f(y; \Theta^*)$ を利用して、推定信号を得ることができます。

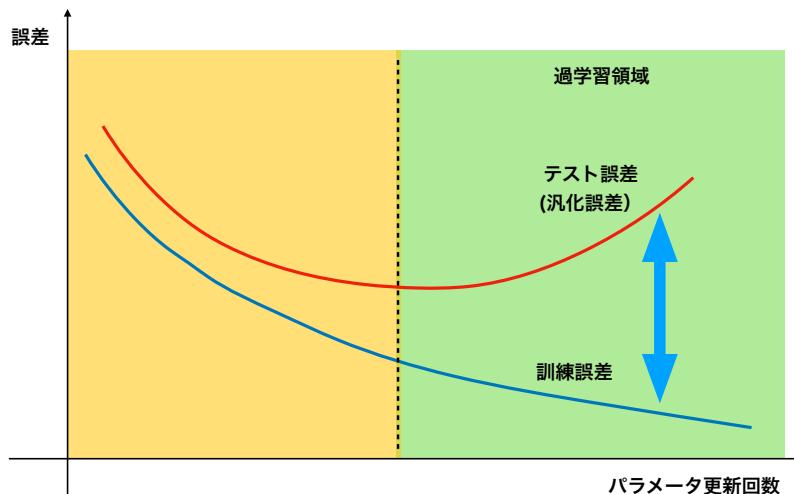


図 2.2 訓練誤差とテスト誤差: 一般に確率勾配法によるパラメータ更新が進むと訓練誤差は単調に減少します。一方、データセット以外のデータに基づき定義されるテスト誤差はある点で増加に転じる場合があります。この現象は過学習と呼ばれます。

ただし、学習プロセスの最終的な目標は損失関数の最小化ではなく、訓練中に見ることがなかった未知の入力に対して質の良い推定値を出力する推定器を学習することです。そ

^{*2} 実際には、判別問題では損失関数としてクロスエントロピーが利用されることが多いです。

^{*3} Θ を集合として定義しているので、左辺の記述は数学的に厳密ではありません。計算手続きを表す一種の疑似コードと考えてください。

^{*4} 深層学習の学習プロセスでは、反復中に学習率を変更することがしばしば行われます。

の意味では推定器のテスト誤差が最終的な推定性能を決めます。テスト誤差の測定の際には、学習プロセスで利用したデータセットを利用することはできません。テスト誤差測定用のテスト用データセットを利用する必要があります^{*5}。

図2.2に、一般的な訓練誤差とテスト誤差の関係を示します。確率的勾配法のパラメータ更新回数を横軸に、誤差を縦軸としています。損失関数値は、データセットに関する誤差値に相当しますので同図の訓練誤差に相当します。訓練誤差は更新回数が増えるに従い、単調に減少していきます。一方、未知のデータに関する誤差であるテスト誤差(汎化誤差)は、反復数が少ないところでは減少していきますが、あるところから増加に転じる場合があります。この現象は過学習と呼ばれます^{*6}。

過学習領域でさらにテスト誤差を減少させるためには、適切な正則化を導入する必要があります。深層学習を含めて多くの機械学習の問題において、適切な正則化を行うことが重要であることが知られています。このことは、機械学習の多くの問題が単なる凸・非凸最適化問題と等価ではないこと示しています。

以上の手続きは、最小2乗法に基づく曲線あてはめに良く似ています。例えば、与えられたデータ点を最小2乗誤差の意味で良く近似する曲線を求める問題を考えましょう。3次多項式モデル

$$f(x; \Theta) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 \quad (2.4)$$

で、データ点(データセットを D とする)を近似する場合、損失関数

$$E(\Theta) := \frac{1}{|D|} \sum_{(x,y) \in D} |x - f(y; \Theta)|^2 \quad (2.5)$$

の最小化により、適切な3次曲線あてはめを与える $\Theta^* = \{\theta_0^*, \theta_1^*, \theta_2^*, \theta_3^*\}$ が得られます(図2.3参照)。深層学習も曲線あてはめ(一般には回帰問題と呼ばれます)も同一の考え方に基づいていると言っても過言ではありません。このことを意識しておくと深層学習技術に少し取り付きやすくなるかもしれません。

単純な最小2乗法と深層学習との差異について少し補足しておきます。上で述べられた回帰問題はパラメータの調整のために2次関数の最小化問題を解けばよいので、単に目的関数の勾配ベクトル(gradient)をゼロと置けば最適パラメータが容易に求められます。しかし、次節で紹介される深層ネットワークモデルにおいては、通常、複数の非線形関数が活性化関数として利用されています。これらの非線形要素の影響で損失関数は一般には非

^{*5} 3種類のデータセットが必要となります。ひとつは、学習用のデータセットであり、もうひとつはハイパーパラメータの調整などに利用するバリデーション用データセットです。最後のひとつがテスト誤差測定用のデータセットとなります。

^{*6} 深層学習モデルでは過学習が起こりにくいことが知られていますが、状況によっては生じる場合もあります

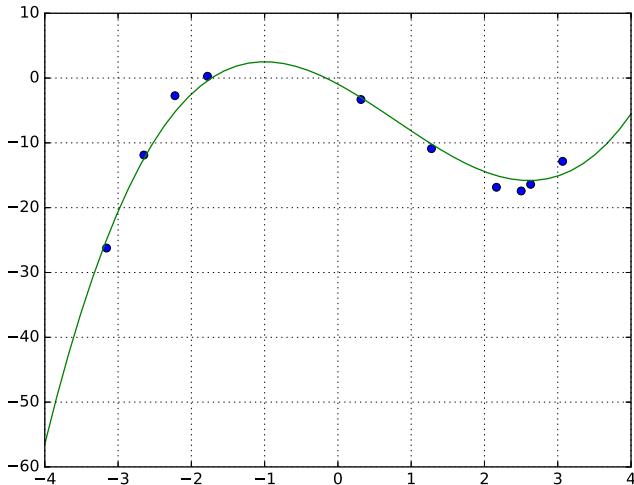


図 2.3 3 次関数モデルに基づく曲線あてはめ: 用意された 3 次関数モデルに対して、データ点における残差の合計を最小化するようにモデルの係数を定めます。

凸関数となります。よく知られるように非凸関数の大域的最小値を見出すことは計算量的に非常に困難な問題です。

2.2 深層ネットワークモデル

前節では、2 値推定問題を例として深層学習の学習プロセスを簡単に説明しました。本節では、 $f(x; \Theta)$ の中身について説明します。図 2.4 に $f(y; \Theta)$ の構造である深層ネットワークモデルを示します⁷。

図 2.4 を左側から見ていきましょう。深層ネットワークへの入力ベクトルを $\mathbf{h}_1 \in \mathbb{R}^{n_1}$ としています。入力信号 \mathbf{h}_1 は線形層 (linear layer)⁸ と活性化関数 (activation function) 層を通り、中間出力

$$\mathbf{h}_2 := g_1 (\mathbf{W}_1 \mathbf{h}_1 + \mathbf{b}_1)$$

に変換されます。

⁷ 本稿では、層数が少ない場合でも深層ネットワーク、または深層ネットワークモデルとこのモデルを称します。

⁸ 実際にはアフィン変換ですが慣習的に線形層と呼ばれます。

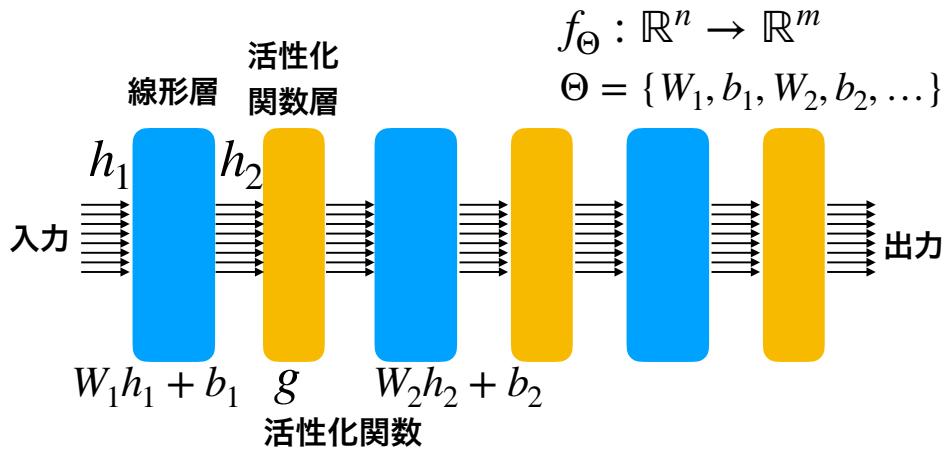


図 2.4 深層ネットワークモデル $f(\mathbf{y}; \Theta)$ の構造: 線形層 (通常はアフィン変換) と非線形関数からなる活性化関数層が交互に連なります。

ここで、 $\mathbf{h}_2 \in \mathbb{R}^{n_2}$ であり、重み行列 $\mathbf{W}_1 \in \mathbb{R}^{n_2 \times n_1}$ 、バイアスベクトル \mathbf{b}_1 は学習可能パラメータです。本稿で学習可能パラメータと呼ぶパラメータは Θ の要素であり、学習プロセスにおいて更新が行われます。

関数 $g_1 : \mathbb{R} \rightarrow \mathbb{R}$ は活性化関数と呼ばれる関数であり、深層ネットワークに非線形性を与える本質的に重要な要素です。機械学習系の論文においてよく利用される記法ですが、関数 $g : \mathbb{R} \rightarrow \mathbb{R}$ と $\mathbf{h} = (h_1, h_2, \dots, h_n)^T \in \mathbb{R}^n$ に対して、

$$g(\mathbf{h}) := (g(h_1), g(h_2), \dots, g(h_n))^T$$

と定義します。すなわち、要素ごとの関数適用を上記の形で表記します。

深層学習でよく利用される活性化関数として、ReLU 関数 (ランプ関数)・シグモイド関数・双曲線正接関数・恒等関数・ソフトマックス関数 [9] などが挙げられます。例えば、近年、よく用いられる **ReLU** 関数 (図 2.5 参照) は

$$\text{ReLU}(x) := \max\{0, x\}$$

と定義される非線形関数です。ReLU 関数は勾配消失問題を引き起こしにくいことが知られています。

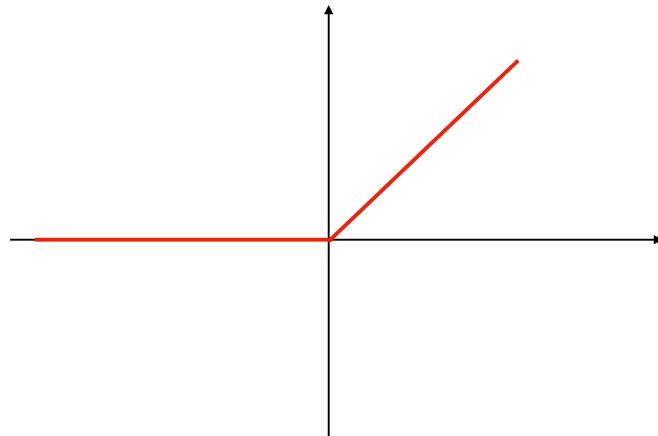


図 2.5 ReLU 関数の概形: 導関数が入力が正の領域で定数 1 となります。この性質が勾配消失を防ぎます。

深層ネットワークモデルでは、同様の部分構造

$$\mathbf{h}_{i+1} := g_i(\mathbf{W}_i \mathbf{h}_i + \mathbf{b}_i)$$

が反復的に繰り返されて最終的に \mathbf{h}_T が右端から出力されます。

深層ネットワークモデル内のすべての学習可能パラメータをまとめると

$$\Theta := \{\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \dots, \mathbf{W}_{T-1}, \mathbf{b}_{T-1}\}$$

となります。画像認識によく利用される畳み込み型深層ニューラルネットワークでは、行列 \mathbf{W}_i や活性化関数層に対して特殊な構造(畳み込み層やプーリング層)が利用されます。

確率的勾配法を利用するためには、これらパラメータ $\mathbf{W}_i, \mathbf{b}_i$ に含まれる要素の偏微分値を効率よく計算できる必要があります。これらの学習可能パラメータの勾配ベクトルを効率よく計算するために誤差逆伝播法 [21] が利用されます。この誤差逆伝播法の実行時間が深層学習の学習プロセスの計算時間において支配的となります。

2.3 確率的勾配法（ミニバッチ学習法）

近年の深層学習技術における革新の要因のひとつは、確率的勾配法 (SGD; Stochastic gradient descent) [9] ^{*9}に基づくミニバッチ学習法と大量の訓練データの組み合わせにあります。

深層ネットワークモデル、または信号流グラフにより表現される多変数入力・ベクトル値関数(深層展開の場合、後述)を

$$\mathbf{y} = f(\mathbf{x}; \Theta)$$

と書きます。ここで、 $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$ である。また、 $\Theta \in \mathbb{R}^N$ は調整可能な学習可能パラメータの組とします。

関数 f は一般に非凸関数であり、 f の内部に含まれる非線形関数(深層ネットワークの場合は活性化関数)の影響から多峰性の関数であるとともに、一般に多くの停留点を持ちます。一般には、大規模問題において、このような関数の大域的最小解を見出すことは計算量的に非常に困難な問題です。そこで、確率的勾配法を利用して大域的最小値に近い値を与える停留点を効率良く見出しが現実的な目標となります。さらに深層ネットワークの学習においては、鞍点を含む停留点でいかに探索の停滞を防ぐかも重要になります。

以下の説明では、訓練データ

$$D := \{(x_1, y_1), (x_2, y_2), \dots, (x_T, y_T)\}$$

が与えられているとします。データ x_i が入力値に相当する値で、 y_i が対応する出力値です。

確率勾配法 (SGD) は、勾配法にランダム性を導入して得られる最適化手法です。まず訓練データから、一様ランダムにミニバッチと呼ばれる部分訓練データ集合

$$B := \{(x_{b_1}, y_{b_1}), (x_{b_2}, y_{b_2}), \dots, (x_{b_K}, y_{b_K})\}$$

を取り出します。ここでサイズ K をミニバッチサイズと呼びます。。このミニバッチ B に対して、損失関数 $E(\Theta; B)$ を

$$E(\Theta; B) := \frac{1}{K} \sum_{k=1}^K \|y_{b_k} - f(x_{b_k}; \Theta)\|_2^2 \quad (2.6)$$

と定義します^{*10}。確率的勾配法(ミニバッチ学習法)では、次の手順により、学習可能パラメータ Θ の値を更新します(図 2.6 参照)。

^{*9} 確率的勾配法は長い歴史を持ち、80 年代からニューラルネットワーク学習に用いられてきました。

^{*10} 損失関数には様々な種類のものがあります。ここでは一例として平均二乗誤差に基づく損失関数を例として挙げています。

確率的勾配法 (SGD)

Step 1 (初期点設定) $\Theta := \Theta_0$

Step 2 (ミニバッチ取得) B をランダムに生成

Step 3 (勾配ベクトルの計算) $g := \nabla E(\Theta; B)$

Step 4 (探索点更新) $\Theta := \Theta - \alpha g$

Step 5 (反復) Step 2 に戻る

ミニバッチ B は確率変数であるため、確率的勾配法は、損失関数 $E(\Theta; B)$ の期待値を最小化している手順と見なすことができます。ニューラルネットワーク学習などの学習問題に確率的勾配法を適用することにより汎化性能の高いパラメータが見出されることが経験的に知られています [9]。

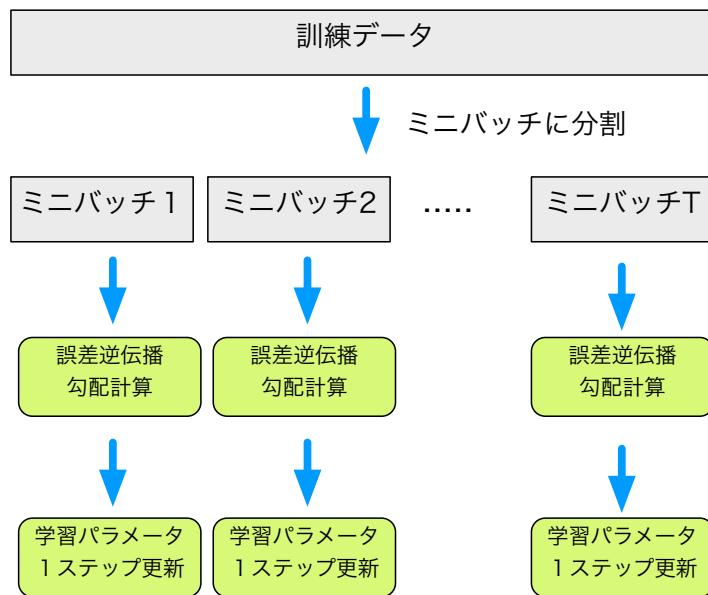


図 2.6 ミニバッチ学習法: ミニバッチは実際には全訓練データをランダムにミニバッチに分割し、そのミニバッチを順次利用することが多いようです。すべての訓練データを使い果たすと(1エポックと呼ばれます)、改めて再度ランダムにミニバッチに分割し手続きを繰り返します。

上で述べた確率的勾配法は最も単純なものであり (**SGD** と呼ばれます)、多くのバリエーションや関連した収束加速手法が知られています [9]。例えば、**RMSprop**, **Adam**などの手法は、学習率 α を適応的に定めるメカニズムを持っているため鞍点や極小点における学習の減速を抑制することが可能であり、広く利用されています。また、単純な SGD、または慣性項を取り入れた慣性 SGD においても、学習率 α のスケジューリングを

適切に行うことで、多くの問題で比較的速い収束が期待できることが知られています。

適切な確率的勾配法の選択、学習率の設定とそのスケジューリング、ミニバッチサイズなどは、ミニバッチ学習において最も重要なハイパーパラメータ群であり、学習結果の良否はこれらの設定の良し悪しに強く影響されます。良いハイパーパラメータ設定を見つけるためには、多少の（場合によってはかなりの）試行錯誤的プロセスが必要となることもあります。

ハイパーパラメータを決めるためには、グリッドサーチやランダムサーチ [9] を利用することが一般的でしたが、最近、ベイズ最適化を利用したハイパーパラメータ最適化手法の研究が活発に行われています。例えば、ベイズ最適化に基づくハイパーパラメータ最適化ツールである Optuna

<https://optuna.org>

などを利用することで、比較的手軽にベイズ最適化に基づくハイパーパラメータ調整が可能となりつつあります^{*11}。

2.4 誤差逆伝播法

確率的勾配法の実行のためには、 Θ に含まれる学習可能パラメータの偏微分値（勾配ベクトル）を計算する必要があります。誤差逆伝播法は、微分の連鎖則を利用することにより層構造（または疎な計算グラフ構造）を持つ関数に対して効率の良い導関数値の計算を実現します。

ここでは簡単な例に基づき誤差逆伝播法の考え方を説明します。まず合成関数に関する微分の連鎖律を復習しておきます。実数値関数 $f : \mathbb{R} \rightarrow \mathbb{R}, g : \mathbb{R} \rightarrow \mathbb{R}$ に基づく合成関数 $y = f(g(x))$ について、合成関数の微分公式（微分の連鎖律）は

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x} \quad (2.7)$$

で与えられます。ここで、 $u = g(x), y = f(u)$ です。

以下では、入れ子となった合成関数

$$y = a_3 f(a_2 f(a_1 f(x))) \quad (2.8)$$

を考えます。言わば、これが非常に単純化した深層ネットワークモデルです。ここで $a_1, a_2, a_3 \in \mathbb{R}$ であり、これらが学習可能パラメータであると考えます。パラメータ更新の際には、 a_1, a_2, a_3 に関する偏微分値が必要となります。

^{*11} 多数のハイパーパラメータの調整のためには計算時間はかかりますが、研究者の時間的負担は少なくなります。

以下では、 a_1 について偏微分値の計算の手順をみていきます。まず中間変数 u_1, u_2, u_3 を導入し

$$u_1 = a_1 f(x) \quad (2.9)$$

$$u_2 = a_2 f(u_1) \quad (2.10)$$

$$u_3 = a_3 f(u_2) \quad (2.11)$$

元の式を書き換えます。微分の連鎖律を使うと

$$\frac{\partial u_3}{\partial a_1} = \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial a_1} \quad (2.12)$$

が得られます。ここで、

$$\frac{\partial u_1}{\partial a_1} = f(x), \quad \frac{\partial u_2}{\partial u_1} = a_2 f'(u_1), \quad \frac{\partial u_3}{\partial u_2} = a_3 f'(u_2)$$

より、

$$\frac{\partial u_3}{\partial a_1} = a_3 f'(u_2) a_2 f'(u_1) f(x) \quad (2.13)$$

となります。

パラメータ a_1 の偏微分値の評価を行う x を \tilde{x} とするとき、まず次の計算を順次行います。

$$u_1 = a_1 f(\tilde{x}) \quad (2.14)$$

$$u_2 = a_2 f(u_1) \quad (2.15)$$

$$u_3 = a_3 f(u_2) \quad (2.16)$$

これらの計算を前向き計算と呼びます。前向き計算の後、 a_1 に関する偏微分値は

$$\frac{\partial u_3}{\partial a_1} = a_3 f'(u_2) a_2 f'(u_1) f(\tilde{x}) \quad (2.17)$$

と計算することができます。この計算は後ろ向き計算と呼ばれます。前向き計算と後ろ向き計算は、図 2.7 のように信号流グラフに関して、左から右に前向き計算を行い、その後に右から左に後ろ向き計算を行うと見ることができます。

誤差逆伝播法は、80 年代に開発された学習パラメータの勾配ベクトルの高速算法であり、前述の例のように合成関数の微分に関する連鎖律を利用する算法となっています。誤差逆伝播法は、前向き計算フェーズと後ろ向き計算フェーズの 2 つ計算フェーズを持ちます。深層ネットワークモデルにおける計算の手順を図 2.8 に示します。

前向き計算フェーズでは、深層ニューラルネットワークにおいては入力層から順に出力層に向かって信号ベクトル値の評価が行われます。信号流グラフにおいても同様であり、各層での処理を多変数入力・ベクトル値関数 ϕ_1, ϕ_2, \dots で表現すると

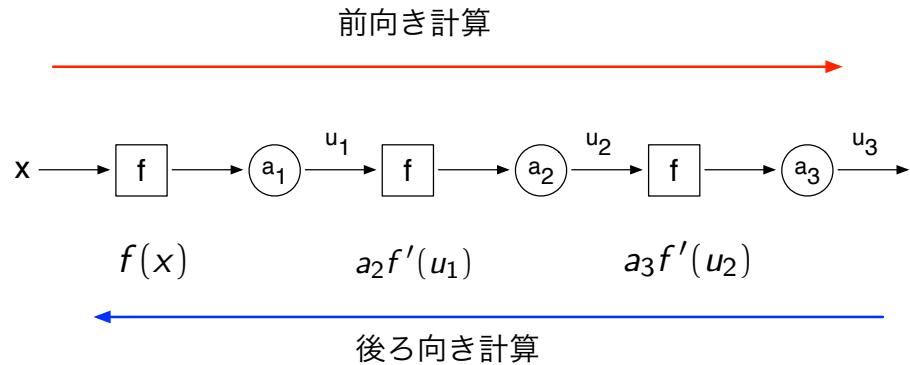


図 2.7 前向き・後ろ向き計算: 前向き計算をまず実行します。途中の計算結果を捨てずに入力することがポイントです。後ろ向き計算では、前向き計算の結果と合わせて偏微分値が定まります。通信分野でよく利用される BCJR アルゴリズム・ビリーフプロパゲーションと良く似た手続きです。

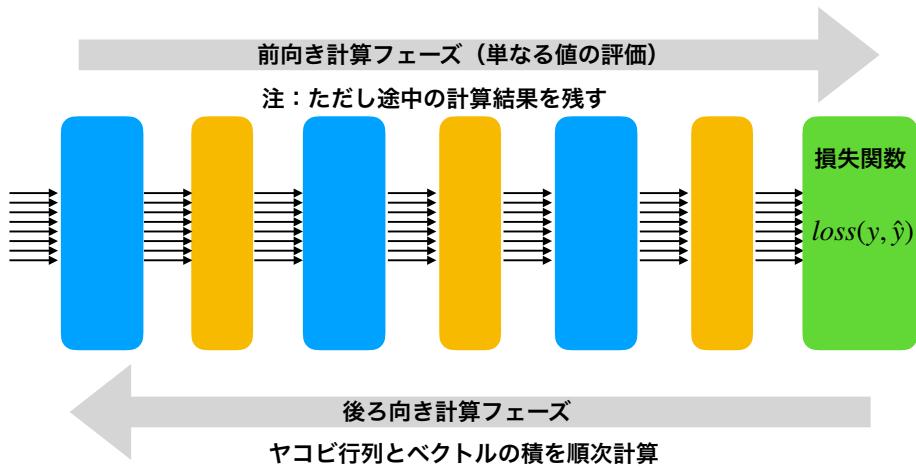


図 2.8 誤差逆伝播法の処理の流れ: 画像データなどを取り扱うときには、前向き計算・後ろ向き計算において大規模な行列・ベクトル積 (より正確には、テンソル積計算) が必要となります。この部分の高速計算のために GPU による並列計算が非常に有効です。

$y = \phi_k(\cdots \phi_2(\phi_1(x)))$ として関数値を順に計算していきます（推論計算と呼ばれることもあります）。なお、後ろ向き計算フェーズのために計算の途中結果は捨ててしまわずに残しておく必要があります。

後ろ向き計算フェーズでは、まず、損失関数値を初期値として、出力端から入力端に向かって、微分の連鎖律に従い計算が進行します。具体的な計算としては、深層ネットワークの場合には、重み行列に対応するヤコビ行列と逆伝播ベクトルの積の計算が順次行われます。

最近のフレームワーク（TensorFlow, PyTorch など）では、前向き計算のみプログラムに記述をすれば、逆伝播計算は自動で行ってくれます。その意味では、エンドユーザは逆誤差伝播に関するプログラミングについて頭を悩まることは、現状ではほとんどないと言えますが、勾配消失（図 2.9 参照）や勾配爆発などの現象を理解したり、活性化関数による学習プロセスの振る舞いの違いを理解するためには、誤差逆伝播法に関して基本的な理解を得ておくことは大変有益です。

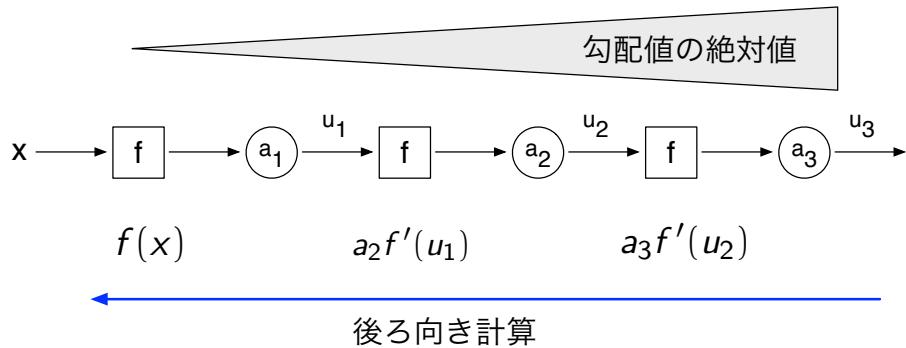


図 2.9 勾配消失問題: 活性化関数の種類によっては、後ろ向き計算のときに後ろ向きベクトルのノルムがどんどん小さくなり、学習が停滞する現象が生じることがあります。例えば、活性化関数が $f(x) := \tanh(x)$ の場合、その導関数 $f'(x)$ は $x = 0$ 付近を除き、 x の全領域で非常に小さい値となるため、勾配消失を引き起こしやすいことが知られています。極端な場合として $f(x) := \text{sign}(x)$ は、ほぼ至るところで微分値がゼロとなるため、その活性化関数の左側に勾配情報が全く流れなくなります。

2.5 フレームワーク

フレームワークは、深層学習の要素技術の主たるものを作成した形でまとめたもので、深層学習技術を利用した研究を進める上で欠かせないプログラム開発のプラットフォームです。代表的なフレームワークとして、TensorFlow/Keras, PyTorch, Chainer などがあり、ライブラリ仕様の拡張や機能拡充などを含めて、現在も活発に開発が行われています。

ています^{*12}。

フレームワークを利用する最も大きな利点は、それらが自動微分機能を持っている点です。フレームワーク利用者（プログラム開発者）は誤差逆伝播法の後ろ向き計算を自分で書く必要はありません。すでに述べたように後ろ向き計算フェーズにおいては、微分の連鎖律に従い、行列・ベクトル積の計算と非線形関数の適用を反復して計算を行うことになりますが、前向き計算と異なり後ろ向き計算は記述をミスしやすく、その計算自体を陽にプログラムとして記述することは、プログラム開発者の著しい負担となります。フレームワーク利用者は、自動微分機能の恩恵を受けることができ、前向き計算（推論計算）のみに注力すればよいことになります。

また、フレームワークは各種の確率的勾配法などの最適化アルゴリズム類やデータ前処理用のアルゴリズムも含んでいます。深層学習に必要な様々な定形処理や学習プロセスに必要なアルゴリズムについては信頼性の高いフレームワークに含まれる関数を利用することで、それらの実装に時間を費やす必要がなくなります。

フレームワークは深層学習に必要とされるテンソル（多次元配列）計算をサポートしています。信号処理・通信系アルゴリズムの多くは、その計算処理をテンソル（行列・ベクトル積）計算に帰着させることができます。TensorFlow や PyTorch などのフレームワークは、テンソル計算の記述が平易になるように開発がされています。そのため、これらにより信号処理・通信系アルゴリズムを見通しよく記述でき、十分に複雑な信号処理アルゴリズムを実装することもできます^{*13}。

すでに述べたように深層学習において、最も計算時間がかかるのが誤差逆伝播法における行列・ベクトル積（正確には、ひとつのミニバッチが一気に計算されるためにテンソル積計算となります）の計算に必要とされる計算時間です。特に画像認識のように扱うべきテンソルのサイズが大きい場合には、GPU に基づく並列計算の利用が必須となります。すべてのフレームワークは GPU 計算に対応しており、GPU プログラミングに精通していない開発者でも GPU をフル活用した計算を容易に行うことができる点もフレームワークの優れた点として数えることができます。

深層学習関連の研究を開始する際には、まずどのフレームワークを利用するかを決める必要があります。アルゴリズム設計・研究開発においては、現時点では、TensorFlow（TensorFlow 上で動作する Keras も含めて）と PyTorch の二択の状況にあります。フレームワークの利用者数の点では、世界的に TensorFlow が優位に立っており、それに伴い膨大な関連の情報をネット上で見つけることができます。また、最新の深層学習の論文に掲載されているアルゴリズムの実装を Github において公開することが広く行われて

*12 本稿執筆時（2019年8月）のPyTorchの最新バージョンは1.2です。大幅な進化が予定されているTensorFlowのver2.0（安定版）がリリースされたところです。

*13 信号処理において欠かせない複素数の取り扱いについては、5.8節を参照してください。

おり、TensorFlow による実装が公開されているケースも多いようです。まもなく安定版が登場する TensorFlow ver.2.0 では、Eager モード (define by run) がデフォルトとなり、以前より書きやすく、デバッグしやすくなるものと思われます。

一方、PyTorch は、TensorFlow(ver.1) と異なる計算グラフ構成の方針 (define by run) を取っており、ネットワーク記述部分を素直な python のコードとして書けばよいようになっています。そのためデータ依存のネットワークアーキテクチャを容易に書くことができ、また、明示的に計算グラフを構成する TensorFlow(ver.1) と比べてプログラミング・デバッグがしやすいという利点があり、その柔軟性が研究者に好まれているようです。TensorFlow と同様に最新の実装プログラムが PyTorch で書かれるケースも増えており、多くのコードをネット上で見つけることができます。

深層学習の研究を進めるためには、これらフレームワークをある程度使いことなすことが必要になります。フレームワークに関する知識を得る最も有効な方法の一つが他の人の書いたコードを読むことですので、その意味では PyTorch, TensorFlow/Keras をいずれを選んでも甲乙つけられない状況です^{*14}。

著者のグループでは当初 (2016 年)TensorFlow で研究を開始しましたが、書きやすさ・デバッグのしやすさから 2018 年から PyTorch に切り替えました。そのため本稿で紹介するコードは PyTorch で書かれています。しかし、フレームワークに依存する部分はほとんどないので、TensorFlow (ver.2) で書き直すことは簡単なはずです。

^{*14} TensorFlow ver.2 で Eager モードが取り入れられたので、書きやすさも同等になると予想されます。

第3章

PyTorch 入門

深層学習関連の研究では、プログラムを実装し動かしてみることが非常に重要です。その準備として本章では、PyTorch に慣れるために基本的なコードを動かしてみます。

3.1 PyTorch の実行環境

深層学習に関する研究をスタートする（あるいは試してみる）ためには、それらの計算を行うための環境が必要となります。各人の状況（予算や研究の計画）に従って環境を選ぶことになりますが、どのような選択肢があるのかを知っておくのは悪いことではないと思われます。図 3.1 に代表的な手段のコスト・メリット・デメリットを列挙しています。

深層学習関連の研究を本格的に進めるかどうか分からぬ段階では、次節で紹介する **Google Colab**（以下、Google Colab と表記する）の利用が最もお勧めです。Google Colab は、Google の提供するサーバーサービスであり、Python の実行環境を無料で利用することができます。Google Colab の大きなメリットとして

- (1) 環境構築が不要（TensorFlow, PyTorch を含めて主要なライブラリ（numpy, matplotlib, pandas,...）がすでにインストールされています）
- (2) GPU 計算も無料で実行可能
- (3) Jupyter Notebook ベースのインターフェース

が挙げられます。無料・環境構築不要ですので、まずは Google Colab で研究をスタートしてみて、必要が生じれば手元に GPU マシンを購入^{*1}するなり、クラウド（AWS, GCP, Azure など）で GPU 付きマシンをレンタルするという方針が合理的かと思います。

なお、深層学習をせずに信号処理だけで使うとしても、GPU 計算を無料でできる

^{*1} 本稿執筆時では、NVIDIA RTX2080 Ti がコストパフォーマンスがよいようです（半精度計算はかなりのスピードが出ます）。予算が豊富にあるかたは NVIDIA V100 を買いましょう。

	コスト	メリット	デメリット
Google Colaboratory	無料	環境構築不要	12時間以上の連続計算ができない
AWS, GCPなどクラウドを借りる	有料	環境構築ボタンひとつ・メンテ不要	GPU付きインスタンスを借りるとそこそこのお値段
GPUつきマシンを買う	初期投資が必要	手元に実行環境があるのは色々快適	GPUの稼働率はそれほど高くない。。。
CPU	手元のCPUを利用すればタダ	気軽に試せる	遅い(相対的に)

図 3.1 深層学習の実験を行うための計算環境: まずは Google Colab でスタートするのが予算も準備もいらないのでお勧めです。部局に NVIDIA DGX-2 などがあり、湯水のごとく GPU が利用できます、という方はそちらをどしどしつかいましょう。

Google Colab のメリットは大きいですので、まだお使いになられていない方は一度、試してみられてはいかがでしょうか。

3.2 Google Colaboratory

Google Colab は、Google アカウントがあればブラウザベースで誰でも利用可能です。ブラウザで

<https://colab.research.google.com/>

にアクセスすればすぐに利用ができます(図 3.2 参照)。Google colab の具体的な利用方法については、ネット上に数多くの情報がありますので、そちらをご参照ください。

Google Colab は **Jupyter Notebook** をベースに開発されています。Jupyter Notebook(または、Jupyter Lab)は、セルベースの Python 実行環境で、コードだけではな

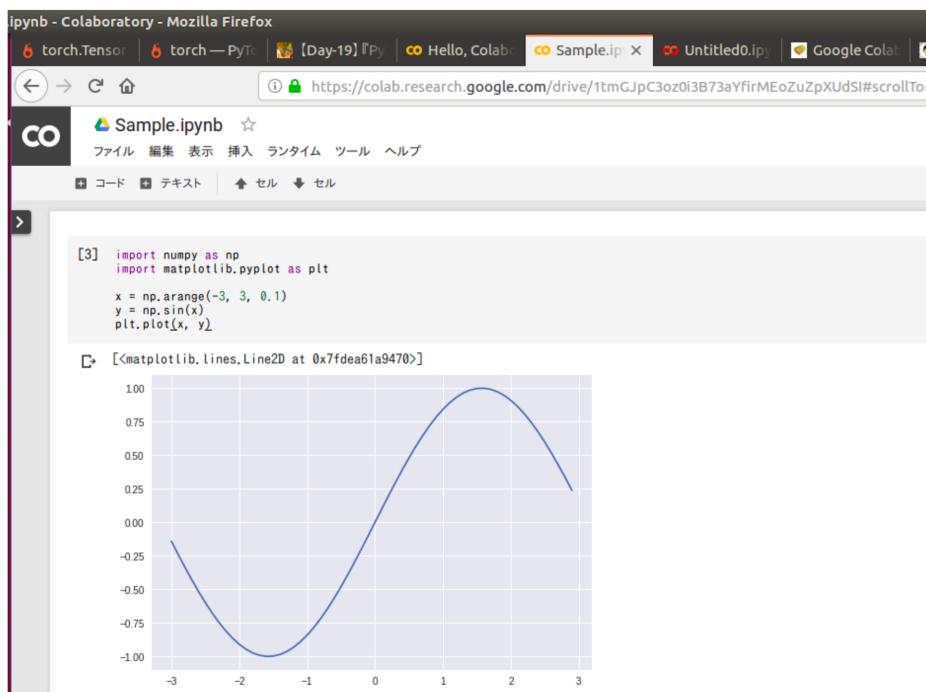


図 3.2 Google Colaboratory: 機械学習関連で利用しそうな Python のライブラリはすでにインストール済みで環境構築が不要な点もうれしい点です。グラフ描画ライブラリである matplotlib をインポートすればグラフもすぐに描けます。

く、テキスト（マークダウン形式）・数式・図・グラフなどをノートブックに含めることができます。研究者間、または研究室内での情報交換や研究ノートとして、便利に活用できますので、こちらもお勧めができます*2。

3.3 PyTorch のチュートリアルを実行してみる

ここから先は PyTorch を利用していきます。手元のマシンに環境を構築する方は PyTorch の本家サイト

<https://pytorch.org/>

にある Get Started の記述に従い、手元のマシンにインストールを行ってください。Widows/Mac/Linux(Ubuntu) が利用可能です。Google Colab を使って試してみるという方は手元のマシンに PyTorch 実行環境を構築する必要はありません。

*2 Jupyter Lab は Jupyter Notebook の上位互換です。新しく使われる方には Jupyter Lab をお勧めします。複数のタブを並べて表示ができるなど便利な機能が追加されています。

最初は、そもそも PyTorch で何ができるのか見当がつかないと思います。Matlab をご存知でしたら、少し書き方の違う **Matlab** みたいな言語と思っていただけだとよいかもしません。

スタート時には、本家サイトのチュートリアルである、

Tutorial → Getting Started → Deep learning with PyTorch: A 60 Minutes Blitz

を順番に実行していくことをお勧めします。チュートリアルのすべてのページの上側に **Run in Google Colab** と書かれたボタンがあります。それを押すと Google Colab によりそのページの内容がオープンされます。セルの先頭の三角(再生マーク)を押すか、そのセルにおいて Shift + Return (Enter) を押すとそのセルの内容が実行されます。セルの内容は自由に自分で書き換えて実行ができますので、単に実行するだけでなく、書き換えながら進んでいくとより理解が深まります。

A 60 Minutes Blitz ですが、チュートリアルの割にどんどん先に進みますので、細かい点がわからなくても気にする必要はありません。最初は

- What is Pytorch
- Autograd: Automatic Differentiation
- Neural networks

の 3 つで十分です。A 60 Minutes Blitz のこの 3 つが終われば、そのページにある Leaning PyTorch with Examples を試してみてもよいかもしれません^{*3}。少しややこしい部分も含んでいますので、このチュートリアルは、もう少し先に回しても大丈夫です。

MNIST と呼ばれる手書き数字データセットに基づく数字認識プログラムを書くことは、深層学習フレームワークを学ぶ上で “Hello world” のプログラムを書くことに相当するそうです。下記のサイト

ライトニング pytorch 入門

<https://qiita.com/sh-tatsuno/items/42fccff90c98103dfffc9>

などを参考にして、MNIST 数字認識を一応やっておきましょう^{*4}。本レポジトリにも簡単な MNIST 数字認識プログラム `mnist.ipynb` を置いていますので、そちらをベースにして改造してみてよいと思います。文字認識が簡単にできるのを目の当たり

^{*3} 色々面白そうなチュートリアル (DCGAN Tutorial など) がありますが、あまり初期から動かしてみても混乱してしまうかもしれませんので、将来の楽しみに残しておきましょう。

^{*4} このページに限らず、さまざまな PyTorch による MNIST 実装がネットにあります。例えば、Github で “PyTorch MNIST” で検索してみましょう。

にすると勉強するモチベーションが上がります。

このぐらいの時点で、様々な種類のテンソル計算を自分で色々実行してみて、テンソル計算の感覚を掴んでおくのがよいでしょう。本家サイトの Doc に PyTorch のドキュメンテーションがあります。これを読みながら、順番に実行していくべき（膨大な数の関数がありますので、もちろん気になる関数だけです）、PyTorch での計算のコツが掴めてくると思います。テンソルのインデキシングなどは Matlab によく似ていますので Matlab に精通されている方ならば、上達は速いのではないかと思います。

画像認識や画像処理の場合は、訓練データとして画像を利用することになりますので、画像の取り扱い（前処理）やデータローダの使い方が重要になりますが、本稿で扱うトッピクス（1 次元信号処理）では必要ありません。MNIST データの取り扱い以上の画像信号の取り扱いについては、余裕ができてからの勉強で十分です^{*5}。

3.4 2 入力 AND 関数の学習

PyTorch によるニューラルネットワーク学習の手順に慣れるためにここでは、論理関数である 2 入力 AND 関数の学習のコードを例にとって説明を行います。

ここで説明を行うコードは Jupyter ノートブック形式で本稿の Github レポジトリ

<https://github.com/wadayama/MIKA2019/>

の下に `ANDfunction.ipynb` という名前で置いています。

手元に PyTorch 環境を構築している方は、レポジトリを手元にクローンするか、レポジトリ全体を zip ファイルでダウンロードしてください。あとは手元の Jupyter Notebook、または Jupyter Lab 環境でノートブックファイルを見てください。また、Google Colab を利用する方は、ファイルの上部にある **Open in Colab** のボタン（図 3.3）を押すと Google Colab でこのファイルをオープンし、すぐに実行することができます^{*6}。

では、`ANDfunction.ipynb` の中身を見ていきましょう。図 3.3 に示される `ANDfunction.ipynb` の冒頭部分でまず重要なのがニューラルネットワークの定義の部分です。ここでは、2 つの線形層（`fc1`, `fc2`）を利用しておらず（以下では `self` は文脈より明らかなので省略します）、活性化関数としてはシグモイド関数を使っています。`fc1` の入力は 2 次元で出力も 2 次元です。一方、`fc2` の入力は 2 次元で出力は 1 次元となって

^{*5}もちろん、もし、画像信号処理の研究を進める方向ならばこの部分もしっかり理解する必要があります。本家サイトのチュートリアルのコードが大変参考になります。

^{*6}なぜかブラウザによっては、ボタンの右側をクリックする必要があるようです。また、ブラウザがポップアップブロックをしていると正しく表示できない場合もあります。その場合は、ブラウザのポップアップブロックの設定を変更してください

AND関数の学習

[Open in Colab](#)

本ノートブックでは、ニューラルネットワークにより、AND関数 AND(a,b)の学習を行う。

必要なパッケージのインポート

```
[1]: 1 import torch # テンソル計算など
2 import torch.nn as nn # ネットワーク構築用
3 import torch.optim as optim # 最適化関数
```

グローバル定数の設定

```
[2]: 1 mbs = 5 # ミニバッチサイズ
```

ネットワークの定義

```
[3]: 1 class Net(nn.Module): # nn.Module を継承
2     def __init__(self): # コンストラクタ
3         super(Net, self).__init__()
4         self.fc1 = nn.Linear(2, 2) # W_1, b_1
5         self.fc2 = nn.Linear(2, 1) # W_2, b_2
6     def forward(self, x): # 推論計算をforwardに書く
7         x = torch.sigmoid(self.fc1(x)) # 活性化関数としてシグモイド関数を利用
8         x = torch.sigmoid(self.fc2(x))
9         return x
```

図 3.3 ANDfunction.ipynb コード (1): ニューラルネットワークの定義は nn.Module を継承するクラスの形で定義する。そのクラスには、コンストラクタと forward メソッドを定義しておく必要がある。

います。forward 部分を見ると入力された 2 次元の信号の流れは、

入力信号 → fc1 → シグモイド関数 → fc2 → シグモイド関数 → 出力信号

となっています。学習可能パラメータは fc1 が持つ W1, b1、そして fc2 が持つ W2, b2 となります。これらの学習可能パラメータを学習プロセスにより調整し、AND 関数の動作をこのニューラルネットワークで模擬することがここでの目標です。

図 3.4 に示されるコードには、学習の際に利用されるミニバッチを生成する関数が与えられています。ランダムに生成された入力テンソルとそれに対応する AND 関数の出力が返り値として計算されます。この AND 関数の出力が教師信号 (result) として利用されます。

画像認識などの PyTorch プログラムの場合は、ミニバッチの生成のためには通常は PyTorch に備わるデータローダと呼ばれるメカニズムを利用するのですが、本稿で示される全てのプログラムではデータセットはランダムに生成しますので、本稿ではデータローダは利用しません。そのため、プログラムがシンプルで理解しやすいものになっています。

ミニバッチ生成関数

```
[4]: 1 def gen_minibatch():
2     inputs = torch.bernoulli(0.5 * torch.ones(mbs, 2)) # 確率0.5で1となるベルヌーイ分布に従う乱数テンソル
3     result = torch.Tensor(mbs, 1)
4     for j in range(mbs):
5         if (inputs[j, 0] == 1.0) and (inputs[j, 1] == 1.0): # AND関数
6             result[j] = 1.0
7         else:
8             result[j] = 0.0
9     return inputs, result
```

```
[5]: 1 inputs, result = gen_minibatch() # 実行例
2 print('inputs = ', inputs)
3 print('result = ', result)

inputs = tensor([[1., 1.],
               [0., 1.],
               [1., 0.],
               [1., 1.],
               [1., 1.]])
result = tensor([[1.],
                [0.],
                [0.],
                [1.],
                [1.]])
```

訓練ループ

```
[6]: 1 model = Net() # ネットワークインスタンス生成
2 loss_func = nn.MSELoss() # Loss関数の指定(二乗誤差関数)
3 optimizer = optim.Adam(model.parameters(), lr=0.1) # Optimizerの指定(Adamを利用)
4 for i in range(1000):
5     inputs, result = gen_minibatch() # minibatchの生成
6     optimizer.zero_grad() # optimizerのgrad初期化
7     outputs = model(inputs) # 推論計算
8     loss = loss_func(outputs, result) # 損失値の計算
9     loss.backward() # 誤差逆伝播法(後ろ向き計算の実行)
10    optimizer.step() # 学習可能パラメータの更新
11    if i % 100 == 0:
12        print('i =', i, 'loss =', loss.item())
```

図 3.4 `ANDfunction.ipynb` コード (2): 動かしてみた後はいろいろパラメータを変えて動かしてみてください。OR 関数・XOR 関数は学習できるでしょうか。

図 3.4 の訓練ループの部分は、このプログラムの核となる部分であり、PyTorch のコードを書くときの典型的なパターンを示しています。まず最初の 3 行で、ニューラルネットワーク、損失関数、オプティマイザのインスタンスを生成しています。次の `for` 文が学習のためのループを構成しています。ループの内部では

- (1) ミニバッチの生成 (`gen_minibatch`)
- (2) オプティマイザ内の勾配情報の初期化 (`optimizer.zero_grad`)

- (3) 推論計算 (`model(inputs)`)
- (4) 損失関数値の計算 (`loss_func(outputs, result)`)
- (5) 誤差逆伝播法 (後ろ向き計算) (`loss.backward`)
- (6) 学習可能パラメータの更新 (`optimizer.step`)

という処理が順次実行されます。この手順は、PyTorchにおいて学習プロセスの典型的な例となっており、本稿に登場する全てのプログラムはこの学習手順に従っています。このパターンに慣れておけば PyTorch のコードを書く難易度がぐっと下がります。

結果の表示部分については、`ANDfunction.ipynb` を参照してください。また、学習プロセスにおける反復回数や学習率、ミニバッチサイズなどを変えて実行してみて、どのように結果が変わるかを観察するのも有意義です。

第4章

深層学習を利用した無線通信技術

深層学習技術に基づく無線通信技術に関する論文・国際会議発表はすでにかなりの数となっており、その内容も多岐に渡っています。本稿ではそれらを網羅することを目指すのではなく、下記の代表的なアプローチ

- (1) ブラックボックスモデル [19]
- (2) 深層展開 (deep unfolding) [10] [23]
- (3) 最適化模擬 (learning to optimize) [24]
- (4) 分散学習 (distributed learning)[25]

について、その基本となる考え方を参考コードとともに紹介していきます。特に著者が興味を持っており、研究の将来性があると考えている深層展開については本章では概要のみを紹介し、次章で詳細について議論を行います。

4.1 ブラックボックスモデル

現実の通信路において取得された実データ、または通信路モデルに従ってランダムに生成された人工データに基いて、深層ネットワークモデルで表現された通信システム（符号化器・変調器・通信路・復調器・復号器）全体や信号推定器における推定則を学習することができます。

そのような通信路モデルや推定器では、通信路の統計的モデルに関する事前知識を全く利用せず、言わばブラックボックスとして深層ネットワークを扱うことから、本稿ではこの種のモデルをブラックボックスモデルと呼びます。本節では、ブラックボックスモデルについて解説します。

4.1.1 ブラックボックスモデルの特徴

ブラックボックスモデルにより、推定器をニューラルネットワークとして構成したり、送信信号点配置の設計を行うことが可能となります。本稿の2.1節にて議論を行った推定器もブラックボックスモデルに分類されます。

ブラックボックスモデルに基づき構成された深層ネットワークの学習プロセスにおいては、訓練データ以外には何も必要とされておらず、その意味では、通信系に関する事前知識が不要である点はブラックボックスモデルの大きな利点です。

画像認識分野では、従来、性能向上に最も影響があったのが画像の特徴量の抽出をいかに行うか、という点でした。そのような特徴量は特徴量設計を研究する研究者により設計がなされてきました。しかし、近年の登場した畳み込み深層ニューラルネットワークでは、畳み込み層における畳込みカネールの学習という形で特徴量抽出の自動化に成功しています。すなわち、画像ドメイン固有の専門知識を利用せずに、柔軟性を持つブラックボックスモデルを利用することで^{*1}非常に高い画像認識性能が達成できることが示されています。

画像分野の成功を踏まえると無線通信系分野においても未知の通信路に対して最大限の柔軟性を持つ通信システムを構成するために、ブラックボックスモデルが有効な手法であることが予想されます。また、通信中にオンラインで学習する仕組みを付与できるならば、通信路の統計的性質の時間的变化に対して高い追従性を持つ通信方式を実現できる可能性があります^{*2}。

ブラックボックスモデルのひとつの欠点は、学習の結果得られたパラメータを見ても、その深層ネットワークの動作機序を読み取ることがほぼ不可能である点が挙げられます。この種の問題は、解釈可能性問題として知られています。学習の結果得られた推定器に何らかの不備があった場合でも、その不備の原因を探りだすことは困難であり、その解決策を見出すことも容易ではない可能性があります。

また、ブラックボックスモデルでは学習可能パラメータ数が大きくなる傾向があり、それに伴い学習に必要となるデータセットサイズが大きくなります。同時に学習プロセスにかかる計算量が大きくなります。学習プロセス自体は、通信システムにおいて一種のオーバーヘッドとなる可能性があるため、ブラックボックスモデルのスケーラビリティについては慎重な検討が求められます。

^{*1} 畳み込み構造の2次元フィルタが特徴量の抽出に有効であるという知見は生かされています。

^{*2} 学習プロセスは推論計算(前向き計算)に比べて計算量的に重い処理となるため、それが実際にリアルタイム処理できるかどうかは今後の検討が必要です。

4.1.2 自己符号化器による通信系全体の模擬

O’Shea と Hoydis [19] は、通信系（送信側・通信路・受信側）全体を自己符号化器としてモデル化するという興味深いアイデアを提案しています。

自己符号化器では、深層ネットワークモデルの入出力関係を $\hat{x} = f(x; \Theta)$ とするとき、その出力と入力が近くなるように、すなわち $\hat{x} \simeq x$ となるように学習時に Θ を調整します（図 4.1 参照）。

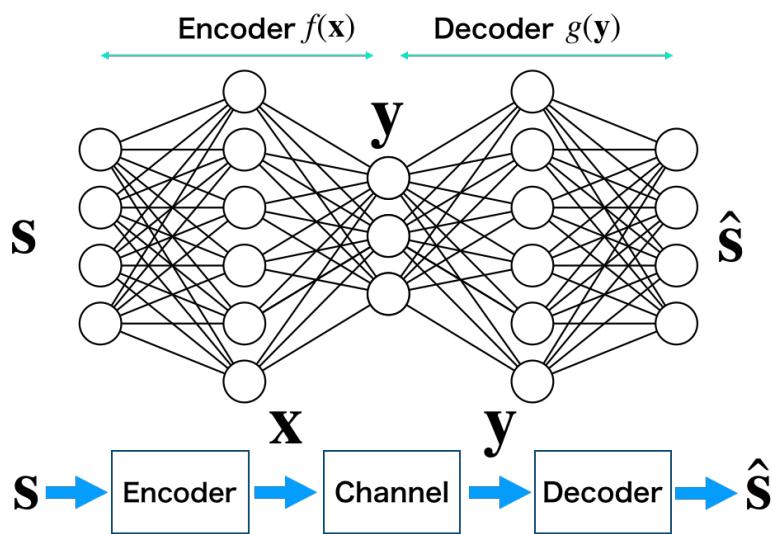


図 4.1 自己符号化器による通信路の模擬：自己符号化器と通信路モデルを対応させて考えます。自己符号化器の中央部分が通信路に対応します。

図 4.1 では、自己符号化器の下に通信路モデルが書かれています。O’Shea らのアイデアは、通信路への入出力に関するデータセットに基づいて通信路モデルに自己符号化器を対応させて学習を行うことにより、その通信路に適した信号表現（変調）を与える符号化器とそれに対応する復号器が自動的に生成されることをねらっています。

例えば、加法的白色ガウス雑音通信路（AWGN 通信路）などに適した信号表現などは良く知られており、例えば平均エネルギー制約やピーク電力制約のもとに PSK, PAM, QAM などが利用されてきています。通信路が厳しい非線形効果を持ち、信号依存性の雑音が生じるようなケースにおいては、与えられた制約条件（電力制約など）のもとに優れた信号表現（信号点配置）を求める問題はそれほど簡単な問題ではありません。また、非線形性を含む通信路特性が時間的に変化する場合には、最適信号点配置を高速に求めることが望ましい場合もあります。

O’Shea らは、自己符号化器への入力 s とネットワークから得られる推定出力 \hat{s} の間の誤差が小さくなるようにモデルを訓練することにより、送信側では信号点配置の自動設計

が、受信側では対応する信号推定器の自動設計が可能になることを実験的に確認しました。最近、O’Shea らの仕事に触発された形で干渉通信路、多重アクセス通信路における信号点配置設計への応用が検討されています。彼らの手法は、複雑な通信路モデルに対する信号点配置設計のための新しいアプローチを提供しています。

4.1.3 自己符号化器アプローチの PyTorch 実装

自己符号化器アプローチの実装は容易です。ここでは、レポジトリに置いてある `autoencoder.ipynb` について説明します。

One-hotベクトル(長さ M)をランダムに生成する (ミニバッチ生成関数)

```
In [25]: one_hot_generator = torch.distributions.OneHotCategorical((1.0/M)*torch.ones(mbs, M))
def gen_minibatch():
    return one_hot_generator.sample()
```

ネットワークの定義

```
In [26]: class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.enc1 = nn.Linear(M, num_hidden_units) # 符号化器用レイヤー
        self.enc2 = nn.Linear(num_hidden_units, 2) # 符号化器用レイヤー
        self.dec1 = nn.Linear(2, num_hidden_units) # 復号器用レイヤー
        self.dec2 = nn.Linear(num_hidden_units, M) # 復号器用レイヤー
    def normalize(self, x): # 送信信号の正規化
        # 平均エネルギー制約
        norm = torch.sqrt((x.norm(dim=1)**2).sum() / mbs)
        x = x / norm
        return x
    def forward(self, m):
        s = torch.relu(self.enc1(m))
        s = self.enc2(s)
        x = self.normalize(s) # normalization
        y = x + torch.normal(torch.zeros(mbs, 2), std=sigma) # AWGN通信路
        s = torch.relu(self.dec1(y))
        m_hat = torch.softmax(self.dec2(s), dim=1)
        return m_hat, y, x
```

図 4.2 自己符号化器のネットワーク定義部 (`autoencoder.ipynb`): 通信路は AWGN 通信路、平均エネルギー制約を仮定しています。

このノートブックでは、 M 個の信号点を 2 次元空間に配置する問題を考えています。制約条件は平均エネルギー制約を課します。また、通信路は AWGN 通信路を仮定しています。図 4.2 にミニバッチ生成関数とネットワーク定義を示しています。

ここで利用するネットワークでは、2 つの線形層を符号化器として利用し、さらに 2 つの線形層を復号器として利用しています。最終層以外は、活性化関数として ReLU 関数

を、最終層ではソフトマックス関数を使っています。

`forward` 関数部分をみると途中で AWGN 通信路がネットワーク定義内に含まれていることが分かります。この部分では、推論計算(前向き計算)ごとに新たにガウス乱数(疑似乱数)が生成されますので、このネットワークは確定的なネットワークではなく、確率的に出力が変わり得る確率的ネットワークになっています。また、AWGN 通信路の部分の次元が 2 次元になっていることに注意してください。これは、問題設定から今回は 2 次元信号空間上の信号点配置を求めることが今回の目標になっているためです。より多次元の信号点配置を検討する場合は、この部分を変更するとよいでしょう。

`forward` 関数に含まれる `normalize` 関数は通過するミニバッチのノルムを正規化し、平均エネルギー制約を満足するようにしています。この正規化層がないと、学習の際に信号点のノルムが発散します。正規化層を変更すると(例えばピーク電力制約や等電力制約にすると)出てくる結果が変わるので、試してみると面白いと思います^{*3}。

この自己符号化器ネットワークへの入力は、長さ M の one-hot ベクトル^{*4}となります。ミニバッチ生成関数 `gen_minibatch` では、ランダムに one-hot ベクトルが mbs 個並ぶミニバッチを生成します。

学習(訓練)ループ部分は、先に見た AND 関数の学習のときに利用した訓練コードとほとんど同じです。今回は教師信号として入力値をふたたび利用しますので、損失値の計算部分が `loss = loss_func(m_hat, m)` という形になっています。

学習の結果として得られた得られた信号点配置を図 4.3 に示します。比較的リーズナブルな形状の配置が得られています。このコードをベースにすれば、例えば、通信路が非線形歪・非線形干渉を持ち、信号依存性雑音が生じるようなケースでの実験も可能です。信号点設計問題に興味をお持ちの方はお試しください。

4.1.4 深層ネットワークに基づく信号推定アルゴリズム

ブラックボックスモデルの特徴を最も直接的に利用しているのが、受信器(信号推定関数)のみを深層ネットワークで表現する試みでちょうど図 2.1 で示される状況です。受信側で正確に通信路モデルがわかっていない場合にも、柔軟に対応ができる信号推定アルゴリズムの構成が可能となります。

信号検出問題の代表的な例として、MIMO 検出問題を考えてみます。よく知られるように MIMO 最尤信号検出問題は、計算量的に困難なクラス(NP 困難問題)に属する問題であり、変調信号点数・送受信アンテナ数が増加すると最尤推定法の計算量が指数的に増大してしまいます。ZF, MMSE 復調といった簡便な推定法による推定精度は、最尤推定

^{*3} 等電力制約はコード内にコメントアウトしてありますので、すぐに試せます。

^{*4} One-hot ベクトルとは、要素の一つのみが 1 でその他の要素が 0 であるベクトルを意味します。

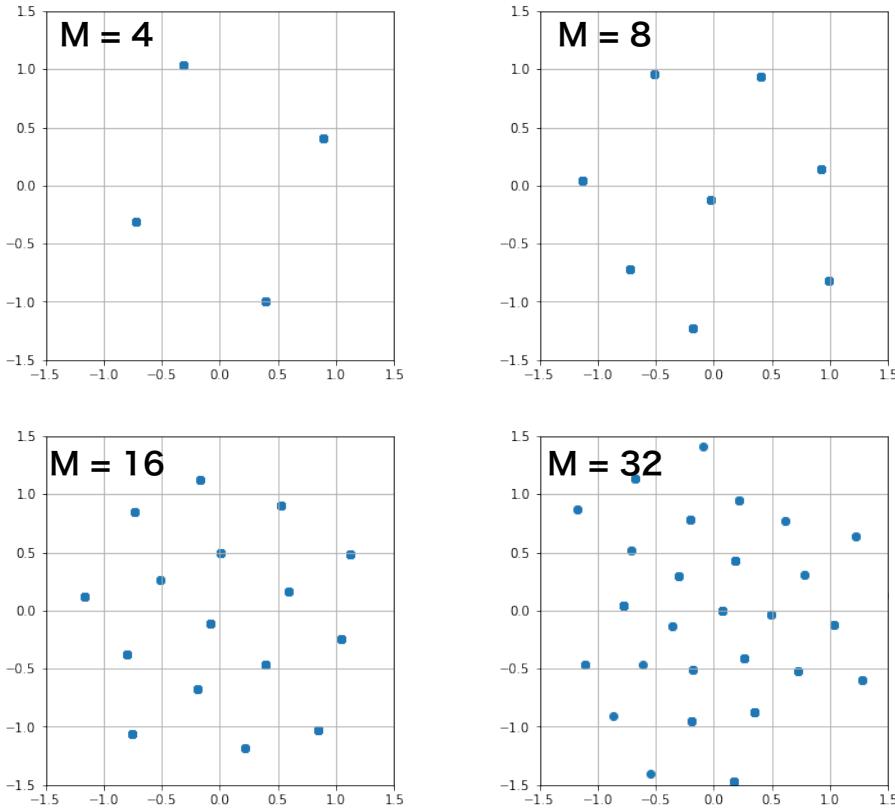


図 4.3 自己符号化器の学習の結果得られた信号点配置 ($M = 4, 8, 16, 32$): AWGN 通信路で平均エネルギー制約を仮定しています。

に比べて著しく劣るため、最尤推定法と MMSE 復調との間に位置する近似的推定法の研究開発が重要になります。深層ネットワークによるアプローチにより、推定精度と計算量のバランスのよい方式をもし実現することができれば工学的に意味のある結果となるでしょう。

近年、この分野に属する方式が多く提案・検討されています。Samuel ら [27] は彼らの論文で 2 種類のニューラルネットワークに基づく MIMO 検出器を提案していますそのうちの 1 つ目 (FullyCon) が深層ネットワークモデルに基づく MIMO 検出器です。Farsad [8] らは、RNN(Recursive Neural Network) を利用したポアソン通信路に対する信号検出アルゴリズムを提案しています。

4.1.5 ニューラル推定器の PyTorch 実装

ここでは、送受信アンテナ数が 4 本の場合の MIMO 通信路に対するニューラル推定器を PyTorch で実装してみます。コードはレポジトリにある `MIMO.ipynb` です。

この問題設定では、複素信号を扱う必要がありますが、ここではコードを簡単にするため、実数領域の検出問題を考えます。なお、変調信号点配置が実数信号点配置の直積として表現できる場合 (QPSK, QAM) には複素領域の MIMO 検出問題を実数領域の検出問題に帰着できます [17]。問題設定は次のとおりです。

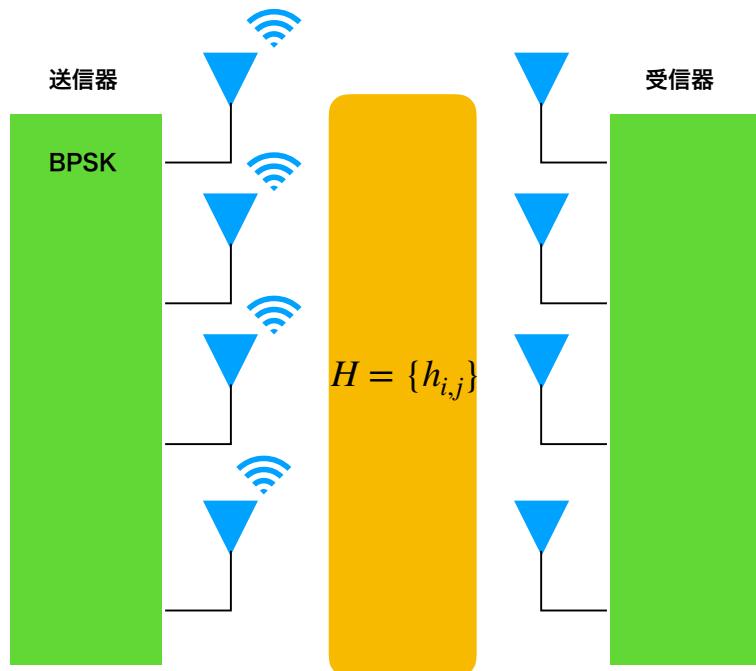


図 4.4 4×4 アンテナ実数領域 MIMO 通信路モデル: 送信信号は BPSK、受信信号は $\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{w}$ と表されます。

送信側は $\mathbf{x} \in \{+1, -1\}^4$ (列ベクトル) を送信したいメッセージに従い、一様ランダムに選択します (BPSK 変調と見なすことができます)。受信側は、受信ベクトル

$$\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{w} \quad (4.1)$$

を受信します。ここで、 $\mathbf{H} \in \mathbb{R}^{4 \times 4}$ は実干渉行列です。干渉行列 \mathbf{H} の各要素は、平均 0、分散 1 のガウス分布に従いランダムに定めます。ここで考える MIMO 通信路モデルを図 4.4 に示します。

なお、もし、複素モデルを考えたい場合は、5.8節に登場する複素ライブラリを利用すれば、ほとんど手間をかけずに複素モデルに拡張することもできます。ベクトル w は AWGN ベクトルで、その各要素は平均 0、分散 σ^2 のガウス分布に従います。推定アルゴリズムの目標は、もちろん受信ベクトルから可能な限り正確に送信ベクトル x を推定することです。

ネットワークの定義

```
[218]: 1 class Net(nn.Module): # nn.Module を継承
2     def __init__(self): # コンストラクタ
3         super(Net, self).__init__()
4         self.detector = nn.Sequential(
5             nn.Linear(n, h), # W_1, b_1,
6             nn.ReLU(), # 活性化関数としてReLUを利用
7             nn.Linear(h, h), # W_2, b_2
8             nn.ReLU(),
9             nn.Linear(h, n) # W_3, b_3
10        )
11    def forward(self, x): # 推論計算をforwardに書く
12        x = self.detector(x)
13        x = torch.tanh(x) # x \in {+1,-1}^4 なので、最終層はtanhを利用
14        return x
```

ミニバッチ生成関数

```
[219]: 1 def gen_minibatch():
2     x = 1.0 - 2.0 * torch.randint(0, 2, (mbs, n)) # 送信ベクトル x をランダムに生成
3     x = x.float()
4     w = torch.normal(mean=torch.zeros(mbs, n), std = noise_std) # 加法的白色ガウス雑音の生成
5     y = torch.mm(x, H) + w
6     return x, y
```

図 4.5 ニューラル信号推定器のネットワーク定義部 (MIMO.ipynb): 深層ネットワークはシンプルに線形層と ReLU 関数を重ねたものとなっています。最後の活性化関数としては、tanh が利用されています。

図 4.5 にネットワークの定義とミニバッチ生成関数を示します。ひとつ大きな注意点があります。ミニバッチ生成関数 `gen_minibatch` を見ると受信ベクトルの生成部は `y = torch.mm(x, H) + w` となっています (ここでコード内の `x` は行ベクトルに対応します)。すなわち、PyTorch のコード内では入力ベクトル x が行ベクトルとして扱われていることになります。これは、PyTorch のコードを書くときにミニバッチテンソルの第ゼロ次元 (先頭次元) をミニバッチ用の次元とするという慣用に従っているためです⁵。そ

⁵ ミニバッチ次元を最後に持っていくことも可能です。どうしても列ベクトル形式で書きたい方はそのよう

のため、本稿内での数式とプログラム内の式がちょうど列と行が入れ替わった形になっています。少し混乱しやすい部分ですので、コードの数式と本稿の数式が違うな、と思った際にはこの点を思い出すようにしてください。

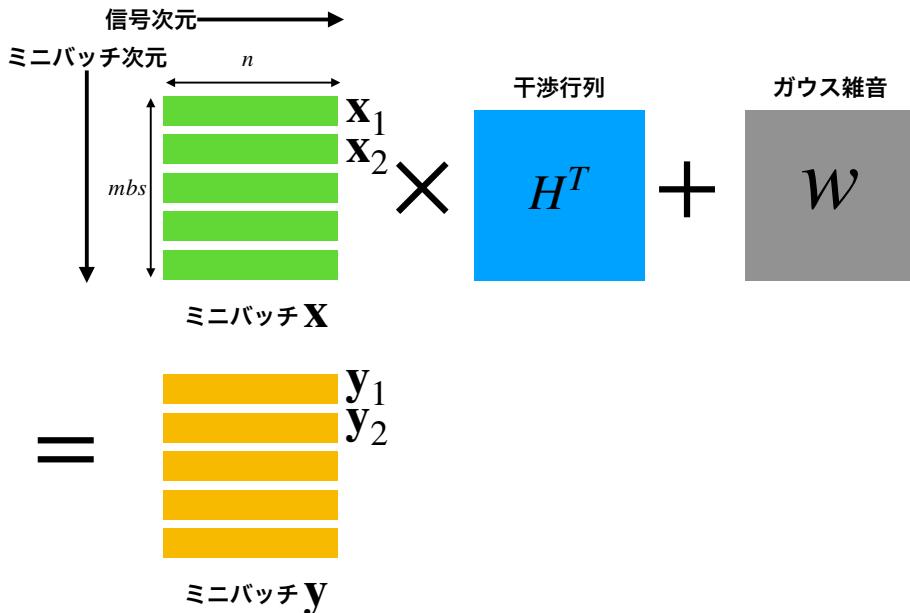


図 4.6 ミニバッチテンソル: 複数の $\mathbf{x}_i, \mathbf{y}_i$ が `mbs` 個集まることでミニバッチが構成されています。

また、PyTorch でモデルを記述する際には、処理途中のテンソルの形状 (何次元のテンソルかなど) を意識してコードを書く必要があります。図 4.6 にミニバッチ生成関数で生成されるテンソルの形状を示します。PyTorch に慣れないうちは、第 0 次元目をミニバッチ用の次元として利用していることにしきりにいかないかもしれません、いくつか PyTorch のプログラムを書くうちに次第に慣れていきます。

さて、ネットワーク定義部に戻りましょう。3 層の線形層と ReLU 関数・tanh 関数に基づく活性化関数層に基づく典型的な深層ネットワークの構造となっています。層の数が多くなってくる場合には、コンテナ関数 `Sequential` を利用して、このように連続する層を列挙する形に書くことで、すっきりと書くことができます。ミニバッチ生成関数については先に述べた通信路モデルにしたがってミニバッチを生成しています。

訓練 (学習) ループについては、ほぼいつも通りの構造になっています。訓練ループを

実行すると損失関数値が表示されるようになっていますが、どんどん小さくなってくることが見て取れます。確率的勾配法を利用しているので、損失関数値は単調には減少せず、上がったり下がったりします。そのため、移動平均を表示するほうが損失関数値のトレンドを見るためには好都合です。ネット上で見つけたプログラムなどを参考にこの部分を改良してみてもよいでしょう。また、損失関数値をグラフとしてプロットするのも、確率的勾配法の振る舞いを知るために有益です。

学習結果の確認の部分では、学習が終了したニューラル推定器の動作を確認できます。例えば、

```
x = tensor([[ 1.,  1., -1., -1.]])
y = tensor([-1.4855,  2.5058, -1.4543,  0.7748])
x_hat = tensor([ 0.9987,  0.9993, -0.9987, -0.9998])
```

といった結果が表示されます(乱数シードの初期化をしていないので結果は実行ごとに変わります)。この例の場合、 y からニューラル推定器により推定された x_{hat} は送信ベクトル x に非常に近いことが見て分かります。ニューラル推定器は所望の推定動作をしているようです。

Zero Forcing (ZF) 推定では、受信ベクトルに \mathbf{H} の逆行列を乗じることで推定値を得ます:

$$\hat{x} := \mathbf{H}^{-1}y. \quad (4.2)$$

このプログラムの最後の部分では、ニューラル推定器と ZF 推定器のシンボル誤り率を算出しています。ニューラル推定器が ZF 推定器よりも小さい誤り率を与えていることが分かります。

4.2 深層展開

本節では、深層展開についてその概略を説明します。深層展開の考え方は非常にシンプルです

多くの既存の信号処理・通信系アルゴリズムは反復構造を持ちます。例えば、LDPC 符号の復号に利用されるビリーフプロパゲーションなどはその一例です。反復処理内で実行される部分処理を仮に A, B, C としましょう。それらは一般に多変数を入力とする非線形ベクトル値関数です。反復処理過程を時間方向に展開して得られる信号流グラフを展開グラフと呼びます。信号流グラフと深層順伝播型(フィードフォワード型)ニューラルネットワークとの間には著しい類似性があります。信号流グラフが含む部分処理 A, B, C の非線形関数がほぼ至るところで微分可能であり、かつ、非ゼロの導関数値を持つ多変数関数であるならば、標準的な深層学習技術(誤差逆伝播法・確率的勾配法)をこの信号流

グラフに適用することができます。

4.2.1 深層展開の特徴

反復的構造を持つ信号処理・通信系アルゴリズムの信号の流れを信号流グラフとして図 4.7(a) に示します。図中で矢印が信号で、サブプロセス A, B, C は多変数を入力とする線形・非線形ベクトル値関数です。信号の流れに注目して、図 4.7(a) の反復処理過程を時間方向に展開して得られる展開グラフ (unfolded graph) を図 4.7(b) に示します。

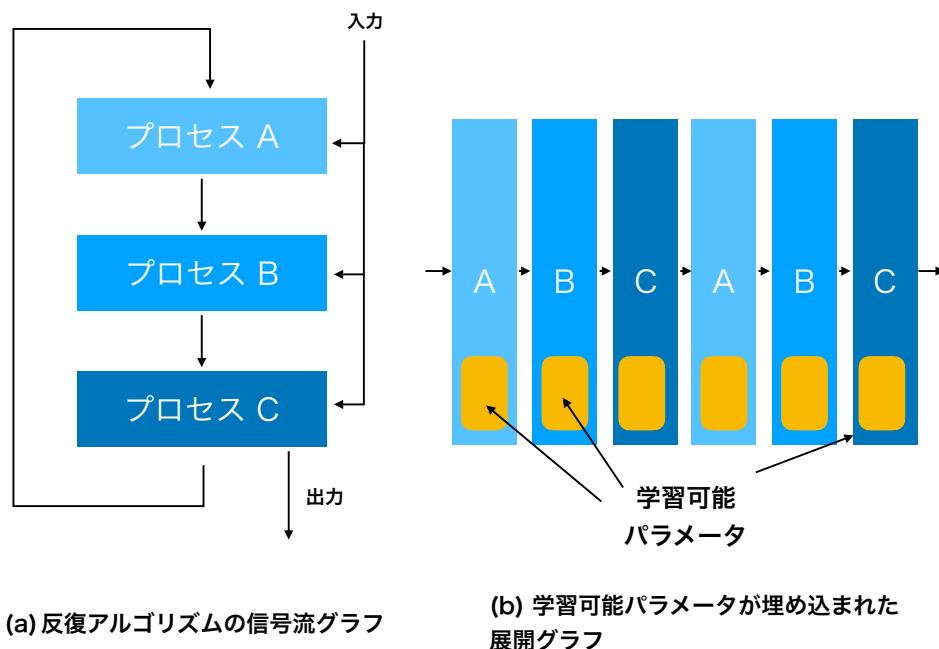


図 4.7 (a) 反復アルゴリズムの信号流グラフ, (b) 時間方向に展開された展開グラフ (unfolded graphs): 入力信号は (b) の信号流グラフの左側から入り、右方向に流れます。

この展開グラフは、先に紹介した深層ネットワーク、あるいはフィードフォワード型ニューラルネットワークに非常に良く似た構造をしています。サブプロセス A, B, C に含まれる非線形関数が微分可能であり、かつ、非ゼロの導関数値を持つ多変数関数であるならば、標準的な深層学習技術（誤差逆伝播法・確率的勾配法）をこの信号流グラフに適用することができます。すなわち、このグラフが学習可能パラメータを含んでいれば、それらを調整することが可能であることを意味します。

このように深層学習技術は、必ずしもその対象が深層ニューラルネットワークに限られず、入出力を伴う微分可能な反復型アルゴリズムに対して適用することができます。従前

から知られている優れた反復型アルゴリズムをベースとして、その中に学習可能パラメータや学習可能な部分構造(ニューラルネットワークに基づく可塑的な非線形関数^{*6})を埋め込むことにより、データに基づく学習可能性を持つ柔軟な派生アルゴリズムを構成できます。

このアプローチの可能性を最初に示したのは、スパース信号推定のための反復アルゴリズムである ISTA[3, 5] (Iterative Soft Thresholding Algorithm) の改良を行った Gregor と LeCun [10] です。最近では、このアプローチを深層展開(deep unfolding)と呼ぶことが多くなってきています。深層展開の分野は現在、急速に研究が進んでおり、多くの関連研究が発表されています。サーベイ文献 [23] は、深層展開の最近の研究動向をよくまとめています。

4.2.2 BP 復号アルゴリズムの改善

Nachmani ら [18] は、LDPC 符号 (Low-Density Parity-Check codes) の復号アルゴリズムであるビリーフプロパゲーション(BP)復号法の性能改善のために深層学習技術を初めて応用しました。BP 復号法では、変数ノード処理・チェックノード処理により算出されるメッセージの交換を反復することにより復号プロセスが進行します。Nachmani らはメッセージに乘じる形の学習可能パラメータを導入し、ミニバッチ学習によりそれらの学習パラメータの最適化を行っています。実験の結果として、BCH 符号のように比較的密度が高い検査行列を持つ符号に対して、BP 復号特性の顕著な特性改善が確認されています。これは、復号に悪影響を及ぼすタナーグラフに含まれる短いループの影響を軽減するように乗数が学習されたものと考えられています。

4.2.3 スパース信号再現アルゴリズムの改善

圧縮センシング(スパース信号再現問題)は、劣決定系である線形観測システムにおいて、線形観測ベクトルから元信号である疎ベクトルを推定する問題です。

無線通信の文脈では、圧縮センシングは、通信路推定(マルチパスフェーディング通信路)、到来波角度推定、スペクトルセンシング、MIMO 検出技術、NOMA 検出技術などと深く関係しており、ますます無線分野でその重要性が高まりつつあります。圧縮センシング技術の無線通信への応用については、文献 [28] に様々な例が紹介されていますので、ご興味がある方はそちらを参照してください。

圧縮センシングの問題設定は次のとおりです。原信号 $\mathbf{x} \in \mathbb{R}^N$ は非ゼロ要素数が N に

^{*6} 適切な条件を満たす多段ニューラルネットワークは万能関数近似器として利用することができます。

対して十分に小さい疎ベクトルであると仮定します。観測ベクトル $\mathbf{y} \in \mathbb{R}^M (M < N)$ は

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{w} \quad (4.3)$$

と与えられます。ここで、行列 $\mathbf{A} \in \mathbb{R}^{M \times N} (N > M)$ は観測行列と呼ばれる実行列です。また、ベクトル $\mathbf{w} \in \mathbb{R}^M$ は、ガウス雑音ベクトルです。

圧縮センシング問題(スパース信号再現問題)の目標は、与えられた観測ベクトル \mathbf{y} から原信号 \mathbf{x} を可能な限り高い精度で再現することです。注意すべき点は $M < N$ が仮定されていることから、この問題は劣決定性問題(未知変数の数が独立した方程式数よりも多い)となっていることです。通常であれば解は不定となるところですが、元信号のスパース性が解の唯一性を与える場合があり、そのときにスパース信号再現問題が意味を持ちます。

疎ベクトル再現アルゴリズムは従来から様々な手法が知られていますが、ここでは深層学習技術を利用するアルゴリズムのみを紹介します。Gregor と LeCun[10] は、反復型のスパース信号再現アルゴリズムである ISTA の中に登場する線形推定式の推定行列を学習可能パラメータとして学習する手法である LISTA(Learned ISTA) を提案しました。この論文は、反復型信号処理アルゴリズムに対して深層学習技術を適用した最初の論文です。学習の結果として、オリジナルの ISTA に対して大幅に収束速度の向上が得られることを彼らは示しており、深層学習技術の信号処理アルゴリズム設計における可能性を初めて示した論文として高く評価できます。ISTA の詳細については改めて次章で述べます。

Borgerding と Schniter [7] は、スパース信号再現アルゴリズムとして近年活発に関連研究が成されている AMP に学習可能パラメータを導入し、LAMP と呼ばれるアルゴリズムを構成しました。

伊藤・高邊・和田山ら [13, 14] は、ISITA に対して深層展開を適用し、スパース信号再現アルゴリズム **TISTA** を提案しました。TISTA の最も大きな特徴は、ISTA 内の勾配法ステップにおいて現れるステップサイズパラメータを学習プロセスにより調節することにあります。実験的評価の結果、ほとんどの場合、ISTA, LISTA, AMP, OAMP に比べて、TISTA の出力は解への速い収束を示すことが確認されています。また、広いクラスの観測行列(2 値行列、悪条件の行列など)において、TISTA は優れた収束性能を示しており、ロバストな性質を持つことが明らかになっています。

4.2.4 スパース信号再現アルゴリズムの PyTorch 実装

ここでは、スパース信号再現アルゴリズムの PyTorch 実装である **ISTA.ipynb** について解説します。このプログラムで利用されるアルゴリズムは、TISTA[13, 14] をベースとする学習型 **ISTA** です。アルゴリズムの詳細については次章で説明しますので、本節で

は、実装面からみた深層展開の方法の解説に重点を置きます。

ここで紹介するアルゴリズムはシンプル（アルゴリズム本体は5行程度）ですが、既存のアルゴリズムと比較しても十分競争力のあるスパース再現能力を持っていていますので、スパース信号再現に関する研究の起点としてご利用いただけます。本プログラムでは実数体上の圧縮センシング問題を扱っていますが、次章で紹介される複素ライブラリを利用すれば複素数体上の圧縮センシング問題への拡張も容易です。

```

1 class ISTA(nn.Module):
2     def __init__(self, max_itr):
3         super(ISTA, self).__init__()
4         self.beta = nn.Parameter(0.001*torch.ones(max_itr)) # 学習可能ステップサイズパラメータ
5         self.lam = nn.Parameter(0.1*torch.ones(max_itr)) # 学習可能縮小パラメータ
6     def shrinkage(self, x, lam): # 縮小関数（ソフトしきい値関数）
7         return (x-lam)*(x-lam > 0).float() + (x + lam)*(x+lam < 0).float()
8     def forward(self, num_itr):
9         s = torch.zeros(mbs, n) # 初期探索点
10        for i in range(num_itr):
11            r = s + self.beta[i] * torch.mm(y - torch.mm(s, A.t()), A)
12            s = self.shrinkage(r, self.lam[i])
13        return s

```

訓練ループ(インクリメンタルトレーニング)

```

1 model = ISTA(max_itr)
2 opt = optim.Adam(model.parameters(), lr=adam_lr)
3 loss_func = nn.MSELoss()
4 for param in model.named_parameters():
5     print(param)
6 for gen in range(max_itr):
7     for i in range(50):
8         x = gen_minibatch() # 元信号の生成
9         w = torch.normal(torch.zeros(mbs, m), sigma)
10        y = torch.mm(x, A.t()) + w # 観測信号の生成
11        opt.zero_grad()
12        x_hat = model(gen + 1)
13        loss = loss_func(x_hat, x)
14        loss.backward()
15        opt.step()
16        print(gen, loss.item())

```

図 4.8 学習可能 ISTA のネットワーク定義部と訓練ループ (ISTA.ipynb)

図 4.8 は、ISTA.ipynb のアルゴリズム本体であるネットワーク定義部と訓練ループ（学習プロセス）のコードを示しています。

ネットワーク定義部は、学習型 ISTA のアルゴリズムを定義しています。11行目が目的関数の勾配ベクトルに基づく勾配ステップと呼ばれる処理で12行目がソフトしきい値関数に基づく縮小ステップとなっています。学習型 ISTA の信号再現プロセスでは、この2つのステップが交互に反復されることになります。これらのステップの詳細については、次章で説明を行います。ネットワーク定義部の構造では、学習可能パラメータである self.beta[], self.lam[] が定義されています。

パラメータ `self.beta[]` は、勾配ステップにおけるステップサイズに、パラメータ `self.lam[]` は縮小関数の縮小の度合い（正則化パラメータに対応）にそれぞれ対応します。これらのパラメータが定義される箇所で使われている関数 `nn.Parameter` は、引数を初期値とする学習可能パラメータを生成し、オプティマイザに渡す更新パラメータリストにそれらを登録します。

訓練ループを実行すると

```
('beta', Parameter containing:  
tensor([0.0010, 0.0010, 0.0010, 0.0010, ...  
       0.0010, 0.0010], requires_grad=True))  
('lam', Parameter containing:  
tensor([0.1000, 0.1000, 0.1000, ...  
       0.1000, 0.1000], requires_grad=True))
```

という表示がされます。これは、誤差逆伝播法による偏微分値計算とパラメータ更新の対象となっている変数を

```
for param in model.named_parameters():  
    print(param)
```

として表示した結果です。上の表示結果で `requires_grad=True` となっているのは、これらのテンソル変数が偏微分値計算の対象となっていることを意味します。

これらの学習可能パラメータ（ステップサイズ、正則化パラメータ）は、信号再現プロセスの振る舞いに非常に強い影響を与えます。訓練プロセスでは、これらのパラメータの調整をインクリメンタル学習と呼ばれる手法に基づいて行います。インクリメンタル学習では、反復1回のISTAの訓練（学習）をして、次に反復2回の訓練をして、というように順次反復数を増やしながら学習を進める手法です。この学習手法は、勾配消失問題の対策として有効な場合が多いことが経験的に知られています^{*7}。学習プロセスの核となる部分は、今までに出てきたコードとほとんど同じです。

図4.9にISTA.ipynbによるスパース信号再現実験の結果の例を示します。実験パラメータの詳細は、図のキャプションに記述されています^{*8}。このパラメータ設定においては、比較的良好にスパース信号再現ができていることが確認できます。

ISTA.ipynbでは、ステップサイズ、正則化に関するパラメータは学習可能パラメータになっていました。この部分の学習を行わないようにコードを書き換えてみるとパラメー

^{*7} 問題設定によっては、インクリメンタル学習の必要がない場合もあります。

^{*8} ちなみにこのパラメータ設定では、よく知られるAMPは動きません。観測行列 \mathbf{A} の分散がAMPの想定する分散とは異なるためです。

タ学習の効果が明確になります。

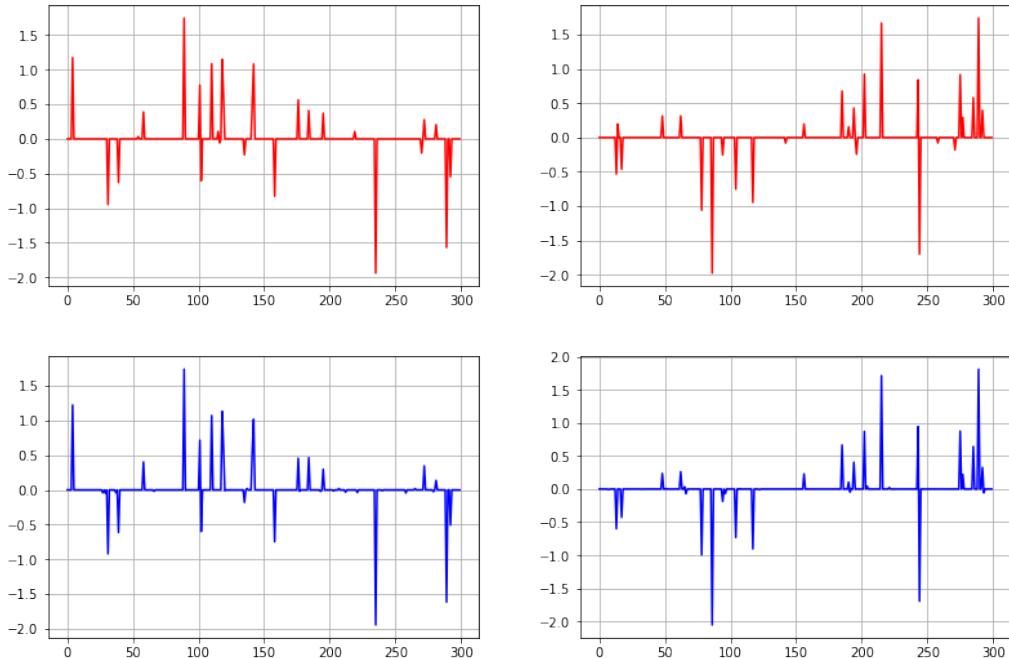


図 4.9 スパース信号再現の実験結果の例 (ISTA.ipynb): 上段が原信号で下段が学習型 ISTA に基づく再現信号。実験パラメータは次の通り: $n = 300, m = 150$ 、非ゼロ要素の生起確率 $p = 0.1$ 、観測行列 A の各要素は平均 0、分散 1 のガウス分布に従う。加法的雑音 w の各要素は平均 0、標準偏差 0.05 のガウス分布に従う。

4.3 最適化模擬

多ユーザが参加する無線通信環境においては、他ユーザからの干渉の影響の軽減のため、送信電力制御を適切に実行することは通信路の利用効率の向上のために非常に重要です。多ユーザ環境での最適ビーム形成問題やプリコーダ設計問題も、同様の種類の問題と見ることもできます。これらの問題は数理最適化問題として定式化されますが、リアルタイムでの電力制御・ビームフォーミングを考える場合には、この最適化問題をいかに高速に解くかという視点が重要になっていきます。ここで紹介する最適化模擬では、深層ネットワークを利用して数理最適化アルゴリズム自身を模擬(エミュレート)するという考え方に基づいて、高速な最適化処理を目指すものです。

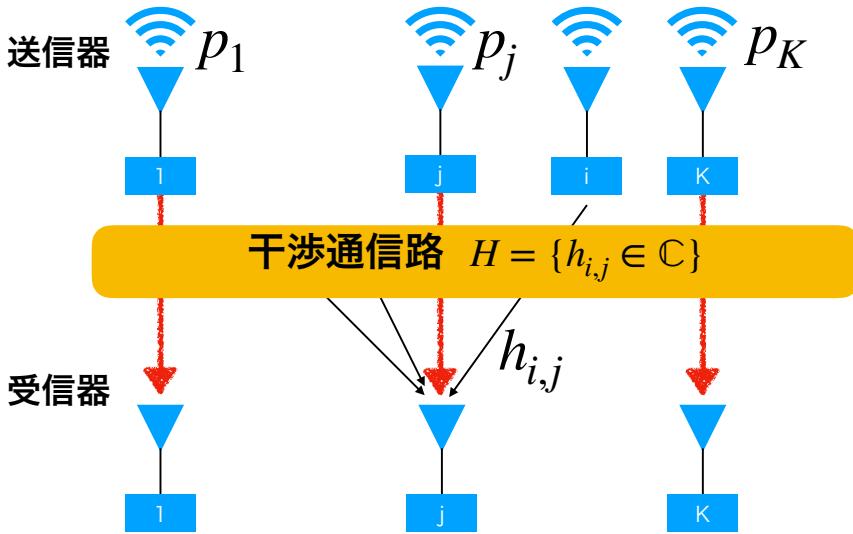


図 4.10 干渉型通信路: K 個の送信器は同じ番号を持つ受信器に変調信号を送りたいものとします。ただし、他の送信器の送信信号は干渉雑音として受信信号に含まれます。

4.3.1 最適化模擬の考え方

文献 [24] に沿って、合計レート最大化問題を例にとって、以下では説明を行います (図 4.10 参照)。合計レート問題は多くのユーザが干渉を受けながら通信を行う際に自然に現れる問題のひとつで、複数の送信器の送信電力を適切に設定することにより、良好な通信状況を達成を目指します。

通信路としては、 K 個の送信器 (1 から K で番号付けられています) が対応する K 個の受信器に変調信号を届けたいものとしましょう。通信路は、干渉行列 $H = \{h_{i,j}\} \in \mathbb{C}^{K \times K}$ で定義される干渉通信路とします。ここで、 $r_k \in \mathbb{C}$ を受信器 k ($k = 1, 2, \dots, K$) が受信する信号とすると

$$r_k = \sum_{j=1}^K h_{k,j} x_j + n_k \quad (4.4)$$

となります。ここで、 $x_j \in \mathbb{C}$ ($j = 1, 2, \dots, K$) は送信器 j から送出された変調信号 (送信電力を p_j) です。また、 n_k は平均 0, 分散 σ_k^2 のガウス分布に従う加法的雑音です。受信

器 k における受信信号電力対干渉および雑音電力比 $SINR_k$ は

$$SINR_k := \frac{|h_{k,k}|^2 p_k}{\sum_{j \neq k} |h_{k,j}|^2 p_j + \delta_k^2} \quad (4.5)$$

と与えられます。

合計レート最大化問題は、以上の問題設定において受信器側の合計レートを最大化する送信電力の設定を見出す問題であり、

$$\max_{p_1, p_2, \dots, p_K} \sum_{k=1}^K \alpha_k \log(1 + SINR_k) \quad (4.6)$$

$$\text{subject to } 0 \leq p_k \leq P_{max} \quad \text{for all } k = 1, 2, \dots, K \quad (4.7)$$

と定式化することができます。ここで、 P_{max} は各送信器の満たすべき最大電力値です。 α_k は各受信器に対応する重み付け定数です。レート部分に対応する $\log(1 + SINR_k)$ はシャノンの通信路容量に対応します。

この合計レート最大化問題は、残念ながら凸最適化問題ではなく、正確に解を求めることが計算量的に困難な問題である NP 困難問題であることが知られています。そのため、効率よく解くためには近似アルゴリズムの利用が一般的です。

特に **WMMSE** アルゴリズムと呼ばれる近似アルゴリズムは、合計レート最大化問題の近似解法としてよく知られています。WMMSE アルゴリズムでは、 H の各要素の絶対値 $\{|h_{j,k}|\}$ を入力として近似パワー設定 $\tilde{p}_1, \tilde{p}_2, \dots, \tilde{p}_K$ を出力するアルゴリズムと考えることができます。WMMSE アルゴリズムの詳細については、文献 [24] などを参照してください。WMMSE は反復型アルゴリズムであり、システムのサイズ K が大きいときは、それなりに重い計算が必要になります。

Sun ら [24] は WMMSE アルゴリズムの入出力関係のみに着目し、WMMSE を模擬する深層ネットワークを構築するというアプローチを提案しました (図 4.11 参照)。すなわち、WMMSE アルゴリズムの出力を教師信号として、二乗誤差損失関数値が小さくなるように深層ネットワークの訓練を行います。同論文では、この手法により訓練されたネットワークが質のよい解を出力すること、深層ネットワークの利用により大きな計算量削減が得られることが主張されています。

このように最適化模擬では、計算量の多い最適化処理をニューラルネットワークで模擬することで、最適化に関する計算量再現を目指す技術であると考えることができます。既存の近似アルゴリズムや凸最適化アルゴリズムを教師信号生成器として利用し訓練プロセスを実行するため元のアルゴリズムの解の品質を凌駕することは困難ですが、一種の高速な近似最適化技法として実用的に活用できる可能性があります。

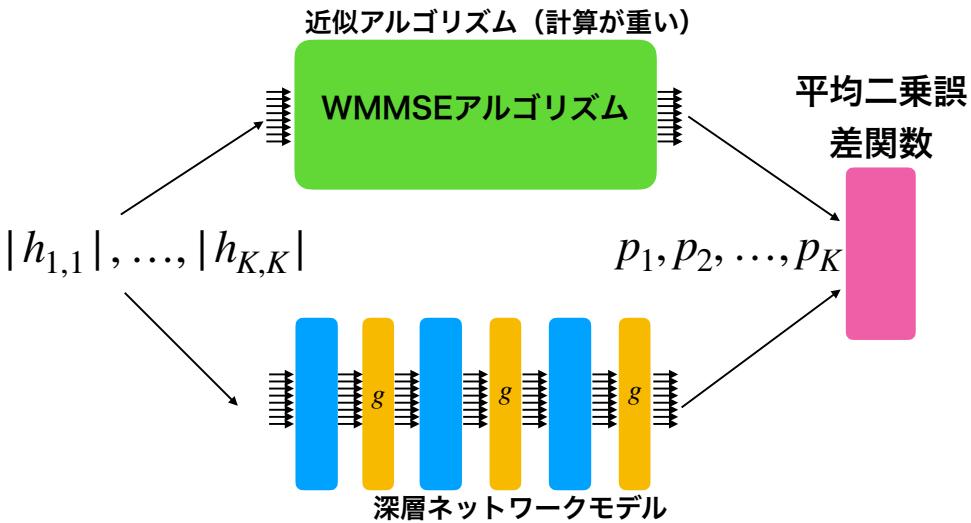


図 4.11 合計レート最大化問題に対する最適化模擬アプローチ: 計算量の多い WMMSE の入出力関係をよく再現するように深層ネットワークモデルが訓練されます。

4.3.2 最適化模擬の PyTorch 実装

ここでは、最適化模擬の概念実証 (Proof of Concept) の意味の PyTorch 実装をご紹介します。ここでは Python 上で利用できる凸最適問題ソルバである **CVXPY** を利用します。CVXPY は凸最適化問題を自然な式の形で与えることができるため利用が簡単であり、半正定値計画問題も含め広い範囲の凸最適化問題を解くことができます。CVXPY の詳細は

<https://www.cvxpy.org>

を参照してください。

ここでは、CVXPY の Example に掲載されている幾何計画問題 (geometric programming problem) の例

https://www.cvxpy.org/examples/dgp/power_control.html

を利用しています。この例は、前節で述べた合計レート問題とは少し違うタイプの送信電

```
In [19]: def gen_minibatch():
    sigma = np.random.rand(n)
    p = cp.Variable(shape=(n,), pos=True)
    objective = cp.Minimize(cp.sum(p))
    S_p = []
    for i in range(n):
        S_p.append(cp.sum(cp.hstack(G[i, k]*p for k in range(n) if i != k)))
    S = sigma + cp.hstack(S_p)
    signal_power = cp.multiply(cp.diag(G), p)
    inverse_sinr = S / signal_power
    constraints = [
        p >= p_min,
        p <= p_max,
        inverse_sinr <= (1/sinr_min),
    ]
    problem = cp.Problem(objective, constraints)
    problem.solve(gp=True)
    return torch.tensor(sigma).float(), torch.tensor(p.value).float()
```

ネットワークモデル

```
In [20]: class Net(nn.Module): # nn.Module を継承
    def __init__(self): # コンストラクタ
        super(Net, self).__init__()
        self.comp = nn.Sequential(
            nn.Linear(n, h),
            nn.ReLU(), # 活性化関数としてReLUを利用
            nn.Linear(h, h),
            nn.ReLU(),
            nn.Linear(h, n),
        )
    def forward(self, x): # 推論計算をforwardに書く
        x = self.comp(x)
        return x
```

図 4.12 ミニバッチ生成部と深層ネットワーク定義部: ミニバッチ生成関数の中に CVXPY を利用した最適化プロセスが含まれています。

力制御問題となっています。問題の定義と詳細は、この URL のページの記述を参照してください。

レポジトリにある `powercontrol.ipynb` は、この送信電力制御問題に対する最適化模擬に関するコードです。深層ネットワークへの入力は、各受信器ごとの雑音電力であり、出力は制約を満たす各送信器の送信電力となります。図 4.12 にこのコードにおけるミニバッチ生成部と深層ネットワーク定義部を示します。

ミニバッチ生成部ではランダムに入力ベクトルを生成し、その入力に対して定義される幾何計画最適化問題を CVXPY に基づいて解いています。そして、その結果と入力をミニバッチ（サイズ 1）として返しています。一方、ネットワークモデルは、線形層と ReLU 関数層からなる典型的な深層ネットワークモデルとなっていることが分かります。学習を行った後に確認すると比較的よく CVXPY ソルバの出力を模擬するネットワークが学習できることができます。

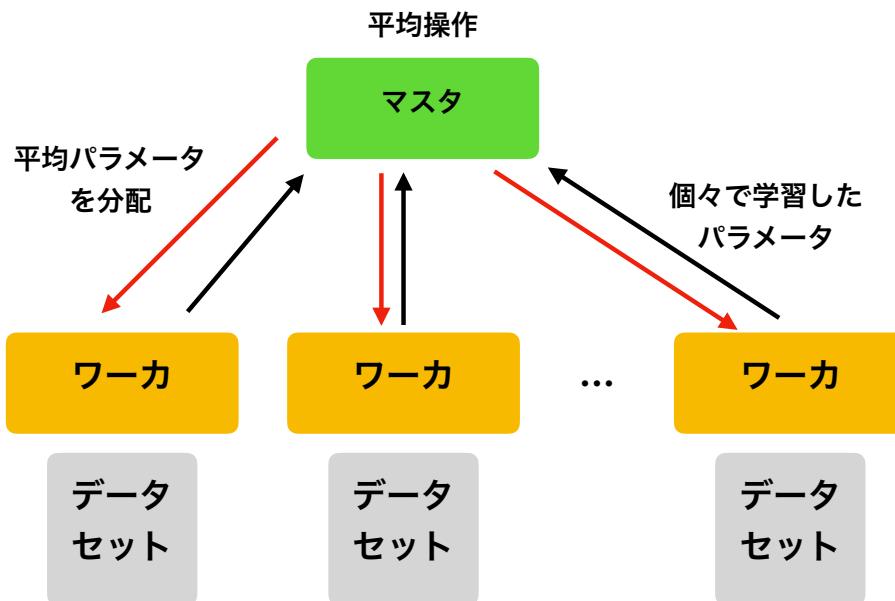


図 4.13 分散学習のシステム構成: 共通のデータセットを持つ設定、共通でないデータセットを持つ設定があります。マスタ・ワーカ間の通信路の伝送容量に制約がある場合には、パラメータを圧縮することが望まれます。

4.4 分散学習

ひとつのマシンにデータセットを集めて、モデルの学習を行う方法を集中型学習と呼びます。ここまで説明してきたモデルの学習プロセスは暗黙のうちにこの集中学習モデルを仮定してきました。もうひとつの異なるシナリオとして、共通したモデルを持つ複数のマシンが協調して学習を進める分散学習も重要な技術です。また、無線通信分野においても、分散学習はホットなテーマになりつつあります [26]。

4.4.1 分散学習の概要

分散学習の典型的な設定では、1台のマスタサーバと複数のワーカサーバがシステムに参加しており、ワーカサーバでは、相異なるデータセット^{*9}を持つ分散学習のシステム構成を図 4.13 に示します。

ワーカは手元のデータセットに基づき、独立・並列的に学習プロセスを進めます。そし

^{*9} 同じデータセットを持つ設定もあります。

て一定のタイミングで、ワーカ達は手元の学習可能パラメータをマスタサーバに送信します。マスタでは、それらの学習可能パラメータを統合（例えば平均を取る）し、新しい学習可能パラメータを作成し、それらをワーカ達に分配します。以上のプロセスを複数回反復することにより、協調的に学習プロセスを進めていくのが分散学習の代表的な学習方法です^{*10}。

分散学習を行う利点の代表的なものは次の3点です。

- (1) **計算負荷の軽減:** 膨大な数の画像データを扱うケースでは、学習プロセス自体が非常に負荷の高い計算プロセスとなります。学習プロセスを並列的に進めることで、一台あたりの計算負荷の軽減が可能となります。最近では大規模な学習プロセスの場合、複数のGPUを並列して利用することが必須となっています。
- (2) **ネットワーク負荷の軽減:** センサネットワークなどでセンシングされたデータなど興味あるデータが空間的に局在する場合、そのデータそのものをマスタサーバに送信するのは、ネットワーク負荷の点で現実的ではない場合があります。データそのものは、各ワーカ（センサ）が手元に持ち（学習後には捨ててもよい）、圧縮したパラメータのみをワーカ・マスタ間でやり取りすることでネットワーク負荷の軽減が実現できます。
- (3) **データのプライバシ保護:** 各ワーカが例えばスマートフォンである状況を考えましょう。ユーザの動作、何らかのセンシングなどでワーカが取得したデータをそのままクラウド上のマスタに送信するのはユーザのプライバシ保護の観点から問題が生じる場合があります。各ワーカで学習プロセスを実行し、学習後のパラメータのみを送信することで上記のプライバシ問題の軽減が図れます^{*11}。

近年登場した連合学習 (federated learning) は、特にネットワーク負荷の削減に重きを置いた技術であり、その無線通信環境への適用なども議論が始まっています [26]。5Gから6Gの流れの中で、低遅延・高速伝送の能力が生かされるタスクとして無線通信環境をベースとした分散学習環境が登場する可能性は高いと予想されます。

4.4.2 分散学習の PyTorch 実装

ここでは、分散学習に関する理解を深めるために、分散学習をエミュレートしたコードである `distributed.ipynb` をご紹介します。このコードは、分散計算の手順をシング

^{*10} 連合学習 (Federated learning) などの先端的な分散学習アルゴリズムでは、ここで述べた手順よりも複雑な手順を実行します。

^{*11} 学習後のパラメータは個人についても個人のプライバシだという論点もあり得るとは思いますので、パラメータ化するだけでプライバシ保護になっているかどうかは議論の余地がありそうです。

マスタにおいてワーカのパラメータの平均とる

```
[18]: 1 def aggregate_and_average():
2     global d_index
3     for i in range(num_workers):
4         if i == 0:
5             sum_vec = torch.nn.utils.parameters_to_vector(worker[0].parameters())
6         else:
7             sum_vec += torch.nn.utils.parameters_to_vector(worker[i].parameters())
8     nn.utils.vector_to_parameters(sum_vec / num_workers, master.parameters())
```

マスタのパラメータをワーカに分配する

```
: 1 def distribute_parameters():
2     master_param_vec = nn.utils.parameters_to_vector(master.parameters())
3     for i in range(num_workers):
4         nn.utils.vector_to_parameters(master_param_vec, worker[i].parameters())
```

分散トレーニング

```
: 1 num_itr = 10
2
3 ##### num_workers = 1
4 num_workers = 1
5
6 d_index_list = []
7 d_loss_list = []
8 d_index = 0
9
10 # ワーカとマスタのインスタンスを作成
11 worker = [0] * num_workers
12 for i in range(num_workers):
13     worker[i] = Net()
14 master = Net()
15
16 #分散トレーニングループ
17 for loop in range(1000//num_itr):
18     for i in range(num_workers):
19         train(i) # 各ワーカのトレーニングを行う
20         aggregate_and_average() # マスタにパラメータを集約・平均
21         distribute_parameters() # 各ワーカにパラメータを分配
22
```

図 4.14 分散学習の PyTorch 実装例 (`distributed.ipynb`): 各ワーカの学習可能パラメータの平均を取る部分と平均を各ワーカに分配する部分では学習可能パラメータの一次元ベクトル表現を利用しています。

ルスレッドのプログラムでエミュレートするプログラムですので、本当には分散学習（並列計算・分散計算）を行っていないことに注意してください。分散学習における学習可能パラメータのマスタ・ワーカ間の受け渡しとそれぞれの処理を可能な限りシンプルに示すことを目指しています。

学習の対象とするモデルは、ブラックボックスモデルのときに扱った MIMO.ipynb に登場したモデルと同一です。図 4.14 に `distributed.ipynb` の重要な部分を示しています。

特に重要な部分が図 4.14 冒頭のマスタ側の平均プロセスです。`for` 文では、まず各ワーカの学習可能パラメータリストである `worker[i].parameters()` を `torch.utils.parameters_to_vector` で一次元ベクトル化します。各ワーカごとの一次元パラメータベクトルを加算し計算した平均一次元ベクトルを `nn.utils.vector_to_parameters` を利用して今度は逆にマスタの学習可能パラメータリスト `master.parameters()` に戻します。もし、分散学習において、パラメータの圧縮などの実験を試してみたい場合には、この部分の 1 次元ベクトルを圧縮・解凍する

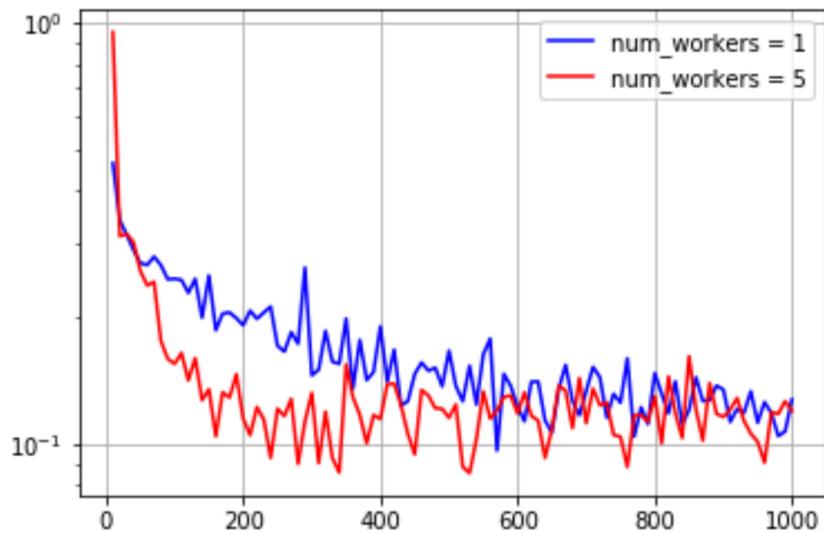


図 4.15 分散学習における損失関数値: ワーカ数が 1 の場合と 5 の場合の比較

コードを付加するだけです。

分散トレーニングのループでは、(1) 各ワーカの訓練、(2) マスタにおける平均計算、(3) 更新されたパラメータを各ワーカに分配、という 3 種類の処理が反復されていることが分かります。

図 4.15 にワーカ数 1 のときと 5 のときの学習プロセスにおける損失関数値を示します。ワーカは並列に動作しているものと仮定した上で反復回数を横軸としています。ですから、ワーカ数が異なっていても公平な比較となっています。この図を見るとワーカ数 5 の学習プロセスがワーカ数 1 の場合に比べて、学習が速く進んでいることが確認できます。

第 5 章

深層展開

前章では、深層展開の考え方をみました。本章のひとつの目標は、勾配法や射影勾配法などの単純な反復型の最小化アルゴリズムへの深層展開の適用を議論することで、深層展開の有効性を明確化することです。

深層展開は、反復的な数理計画法である勾配法、射影勾配法、近接勾配法、ペナルティ関数法などに対して特に有効な技術です。これらの手法において、

- (1) 勾配法、勾配ステップにおけるステップサイズ
- (2) 近接写像 (proximal map)・縮小ステップにおける正則化係数
- (3) 縮小写像としての MMSE 推定関数における誤差分散パラメータ [13, 14]
- (4) ペナルティ関数法におけるペナルティ係数 [15]

を学習可能パラメータとして深層展開を適用することで、多くの場合において収束速度の向上が得られることが経験的に知られています。

本章のもう一つの目標は、近接勾配法への深層展開の適用の詳細をご紹介することです。射影勾配法・ISTA などへの深層展開の実例をみることで、深層展開への理解が深まります。

5.1 勾配法への深層展開の適用

本節では、単純な勾配法を例に取って議論を進めます。

目的関数を $f : \mathbb{R}^n \rightarrow \mathbb{R}$ とするとき、ステップサイズを固定した勾配法の勾配ステップは

$$\mathbf{s}_{t+1} := \mathbf{s}_t - \gamma \nabla f(\mathbf{s}_t). \quad (5.1)$$

と与えられます ($t = 0, 1, \dots, T - 1$)。ベクトル \mathbf{s}_t は探索点を表します。ここで、パラメータ γ は、ステップサイズパラメータで、この値が探索プロセスの振る舞い (収束する

かしないか、収束する場合には収束速度) に強く影響を与えます。以下では、式 (5.1) の勾配ステップを持つ勾配法を GD 方式と呼ぶことにします

勾配法に深層展開を適用した場合、このステップサイズパラメータを学習可能パラメータとするのが自然です。そこで、勾配ステップを

$$\mathbf{s}_{t+1} := \mathbf{s}_t - \beta_t \nabla f(\mathbf{s}_t). \quad (5.2)$$

とする方式 (TGD 方式) を考えましょう。ここで、 $\beta_t (t = 0, 1, \dots, T-1)$ を学習可能パラメータとします。

以下では、2 変数の 2 次関数

$$f(x_1, x_2) = x_1^2 + 8x_2^2, \quad (5.3)$$

を最小化問題を考えます。勾配法 (GD 法, PGD 法) の最小化プロセスにおいては、実行ごとに初期探索点 \mathbf{s}_0 の各要素を実区間 $[-10, 10]$ から一様分布に従い、ランダムに定めることにします。最適解はもちろん $\mathbf{s}^* = (0, 0)$ であり、最小値は 0 です。

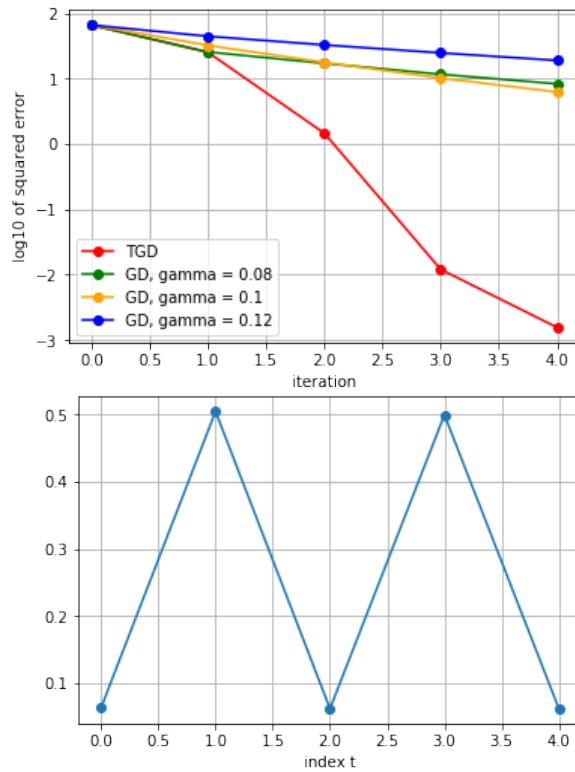


図 5.1 (上段) 勾配降下プロセスにおける誤差の比較 (GD 法・TGD 法) (下段) 学習されたパラメータ β_t (`quadratic.ipynb`): $f(x_1, x_2) = x_1^2 + 8x_2^2$.

レポジトリにある `quadratic.ipynb` では、 $q = 8$ の場合の GD 法, TGD 法の実行が可能です。TGD 法の学習プロセスでは、インクリメンタル学習が利用されています。

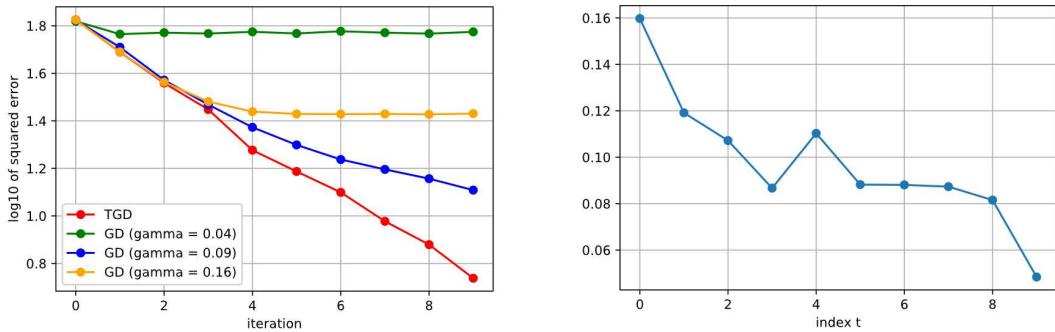


図 5.2 卵入れ関数における TGD 法と GD 法の平均誤差 (左) 学習後の β_t (右).

図 5.1 に、勾配降下プロセスにおける誤差値と学習後の学習可能パラメータ β_t の値を示します。誤差値は 10000 回の試行における誤差 $\|\mathbf{s}_t - \mathbf{s}^*\|_2^2$ の平均値であり、図中では反復回数 (横軸) の関数として表示されています。この図から、TGD 法は、GD 法と比較して非常に速い収束速度を示していることが見て取れます。このときに学習されたパラメータ β_t は図 5.1(下段) に示されるようにジグザグのパターンを示しています。

この例から、単純な凸関数においてもステップサイズを学習により適切に定めれば、ステップサイズ固定の勾配法と比べて、収束速度の点で TGD 法に大きなメリットがあることが見てとれます。

次に極小値が複数ある非凸関数について検討しています。卵入れ関数 (egg crate function) は

$$f(x_1, x_2) = x_1^2 + x_2^2 + 25(\sin^2(x_1) + \sin^2(x_2)) \quad (5.4)$$

と定義される 2 変数の関数です。この関数は数多くの極小値を持ち、大域的最小値は原点 $(x_1, x_2) = (0, 0)$ となります。なお、この関数の等高線表示は図 5.3 に含まれています。勾配法では探索ベクトルが極小点に付近で停滞する可能性が高いため、最小値を見出すことが困難なタイプの関数です。

レポジトリにある Egg.ipynb が本実験に対応するノートブックになります。さきほどの 2 次関数の場合と同様に、ランダム初期値からスタートする GD 法と TGD 法の比較を行います。図 5.2 (左) は、GD 法と TGD 法の誤差曲線を示しています。さきほどの 2 次関数の場合と同様に TGD 法が最小の平均誤差を与えており、GD 法では、 $\gamma = 0.09$ のケースがもっと小さい誤差を与えています。図 5.2(右) は学習後のステップサイズパラメータとなります。2 次関数の場合 (ジグザグ型) と大きく異なる学習結果となっています。

図 5.3 に GD 法により得られた探索軌跡の例 (5 パターン) を示します。この図には、卵入れ関数の等高線表示が含まれています。ステップサイズが小さいとき ($\gamma = 0.04$, 左) には、探索点が極小点にトラップされていることが見てとれます。一方、ステップサイズが

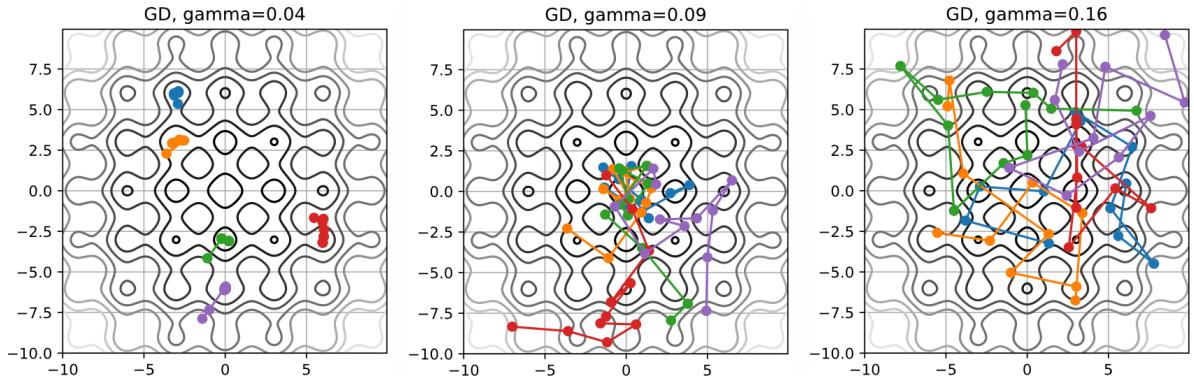


図 5.3 GD 法の探索軌跡の例: $\gamma = 0.04$ (左), $\gamma = 0.09$ (中央), $\gamma = 0.16$ (右).

大きいとき ($\gamma = 0.16$, 右) には、探索軌跡は、原点を中心としたランダムウォーク状の動きをしています。極小点にトラップされることはないですが、かといって原点(大域最小解)に漸近していくことも無さそうに見えます。中程のステップサイズ ($\gamma = 0.09$, 中央)の場合、ランダムウォーク的な動きと極小点への収束の両者の適切なバランスがとれているようです。実際、 $\gamma = 0.09$ がこの三者でもっとも小さい平均誤差を与えています(図 5.3(左) 参照)。

図 5.2 (右) に示される TGD 法における学習後のステップサイズ β_t の形状については次のように解釈することができます。最初の数回では、探索点はランダムウォーク状の動き ($\beta_t \simeq 0.16$) して、目的関数値が小さくなる領域を探します。その後、探索プロセスは、図 5.3 (中央、 $\beta_t \simeq 0.09$) に示される中間的な動作モードを経て、最終的には収束モード ($\beta_t \simeq 0.04$) に移行します。このように深層展開により学習されたステップサイズは、与えられた非凸関数に適した探索戦略を自動的に見出しているようです。

5.2 射影勾配法への深層展開の適用

前節では、2次元の問題について学習型勾配法の特性を見てきました。本節では、無線通信に登場するタイプの最適化問題、すなわち

- (1) 高次元の非凸最小化問題である
- (2) 最適化のインスタンスが確率的に決定される
- (3) 変数に関して事前条件(離散値条件)が課されている

に関して、深層展開の可能性を議論します。これらの条件に対応するためにここでは、最適化アルゴリズムとして射影勾配法を利用します。

ここで考える最適化問題は、BPSK の MIMO 検出問題や線形ベクトル通信路における

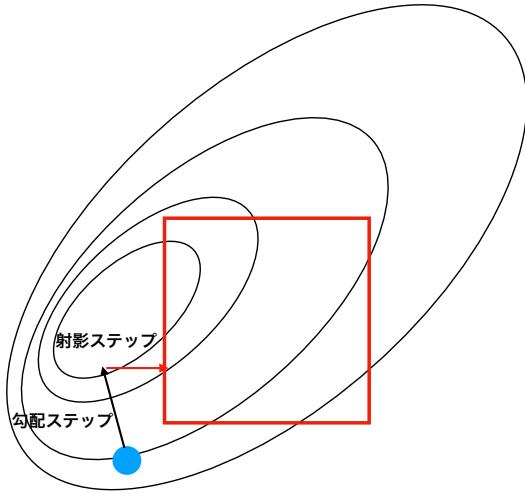


図 5.4 射影勾配法の概念図: 勾配ステップと射影ステップが交互に実行されます。

信号検出問題と本質的に同等な

$$\underset{\mathbf{x}}{\text{minimize}} \frac{1}{2} \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2 \text{ subject to } \mathbf{x} \in \{-1, 1\}^n, \quad (5.5)$$

です。ここで、 $\mathbf{A} \in \mathbb{R}^{n \times n}$ は、各要素が平均 0、分散 1 のガウス分布に従うランダム行列です。ベクトル $\mathbf{y} \in \mathbb{R}^n$ は、

$$\mathbf{y} := \mathbf{A}\mathbf{x} + \mathbf{w}$$

として生成されるものとします。ここで、 $\mathbf{x} \in \{-1, +1\}^n$ は一様ランダムに選択された 2 値ランダムベクトルであり、 $\mathbf{w} \in \mathbb{R}^n$ は、各要素が平均 0、分散 σ^2 のガウス分布に従うランダムベクトルです。

一般の制約付き最小化問題

$$\underset{\mathbf{x}}{\text{minimize}} f(\mathbf{x}) \text{ subject to } \mathbf{x} \in \mathcal{F} \quad (5.6)$$

に対する射影勾配法の基本構成は

$$\mathbf{r}_t := \mathbf{s}_t - \gamma \nabla f(\mathbf{s}_t) \quad (5.7)$$

$$\mathbf{s}_{t+1} := \varphi_{\mathcal{F}}(\mathbf{r}_t), \quad (5.8)$$

となります。ここで、 $\varphi_{\mathcal{F}}$ は実行可能領域 \mathcal{F} への射影写像です。式 (5.7) は勾配ステップ、式 (5.8) は射影ステップと呼ばれます。射影勾配法の概念図を図 5.4 に示します。

目的関数を

$$f(\mathbf{x}) := \frac{1}{2} \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2 \quad (5.9)$$

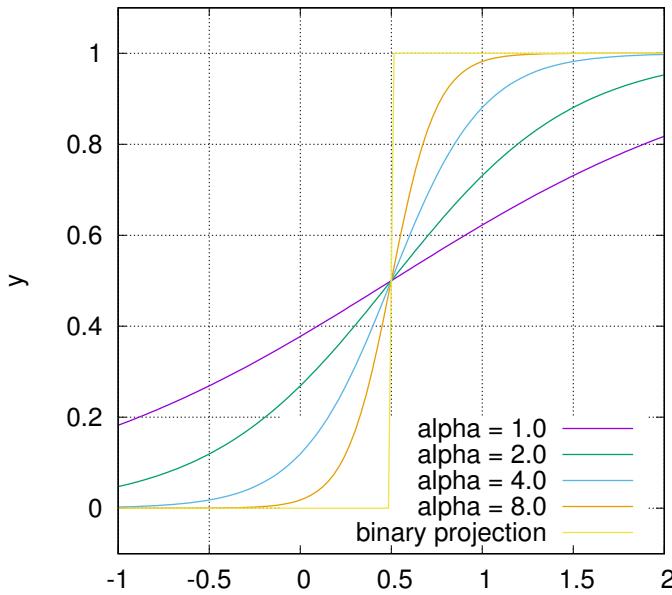


図 5.5 2 値ベクトルへのソフト射影関数 $\varphi_{\mathcal{F}}(\mathbf{x}) \simeq \tanh(\alpha \mathbf{x}) : \alpha$ の値が大きくなるに従い、立ち上がりが急峻になり sign 関数に近づいていきます。

とするとき、この目的関数に対する勾配ベクトルは

$$\nabla f(\mathbf{x}) = -\mathbf{A}^T(\mathbf{y} - \mathbf{Ax}) \quad (5.10)$$

であることに注意すると最適化問題 (5.5) に対応する射影勾配法として

$$\mathbf{r}_t := \mathbf{s}_t + \gamma \mathbf{A}^T(\mathbf{y} - \mathbf{As}_t) \quad (5.11)$$

$$\mathbf{s}_{t+1} := \tanh(\alpha \mathbf{r}_t) \quad (5.12)$$

という反復式が得られます。ここでは、射影ステップで $\{-1, +1\}^n$ への真の射影写像の代わりに、それをソフト化した関数

$$\varphi_{\mathcal{F}}(\mathbf{x}) \simeq \tanh(\alpha \mathbf{x})$$

を利用しています。正実数 α はソフト化の度合いを制御するパラメータと考えることができます (図 5.5 参照)。

ステップサイズパラメータを学習可能パラメータとすることで、勾配法の場合と同様に学習型射影勾配法 (以下では TPG 法と呼びます) を

$$\mathbf{r}_t := \mathbf{s}_t + \beta_t \mathbf{A}^T(\mathbf{y} - \mathbf{As}_t) \quad (5.13)$$

$$\mathbf{s}_{t+1} := \tanh(\alpha \mathbf{r}_t), \quad (5.14)$$

と構成できます。以下では、ステップサイズパラメータを固定した通常の射影勾配法を PG 法と呼ぶことにします。

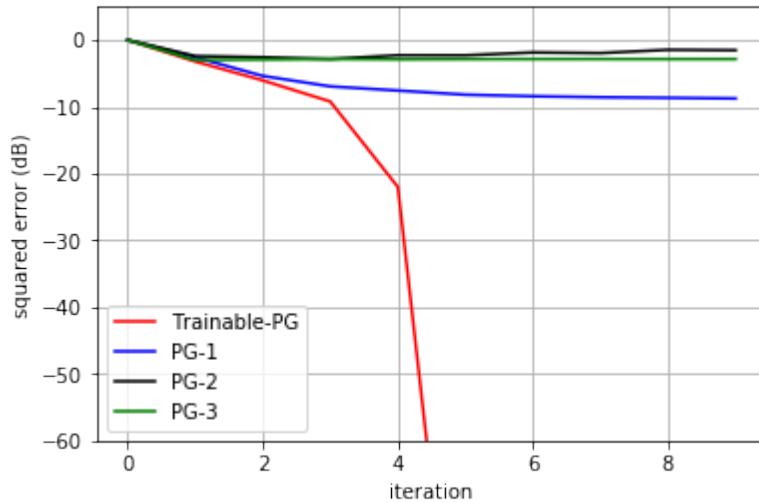


図 5.6 TPG 法と PG 法の平均誤差: TPG 法の場合、反復数が大きくなるに従って急速に誤差値が小さくなることがわかります。

レポジトリにある `ProjectedGradient.ipynb` は学習型射影勾配法に関するノートブックです。このノートブックでは、信号長 $n = 500$, $\sigma = 0.5$ のケースにおいて、TPG 法と PG 法の比較を行っています。学習後の平均誤差を図 5.6 に示します。PG 法に比べると TPG 法は急峻な平均誤差の減少を示していることが分かります。この実験と関連する実験が文献 [17] にありますので、詳細についてご興味がある方はそちらをご参照ください。また、同論文で提案されている過負荷 MIMO 通信系に適した学習型射影勾配法に基づく検出手法 [17] の参考コード

https://github.com/wadayama/overloaded_MIMO

に置いています。

5.3 近接勾配法と ISTA

近接勾配法 (proximal gradient method) は、微分できない関数を含む凸最適化問題を効率よく解くための手法です。近接勾配法では、射影写像を一般化した近接写像 (proximal map) と呼ばれる写像を最適化プロセス内で利用します。関数 $h : \mathbb{R}^n \rightarrow \mathbb{R}^n$ に対応する近接写像は

$$\text{prox}_h(\mathbf{x}) = \operatorname{argmin}_{\mathbf{u} \in \mathbb{R}^n} \left(h(\mathbf{u}) + \frac{1}{2} \|\mathbf{u} - \mathbf{x}\|_2^2 \right). \quad (5.15)$$

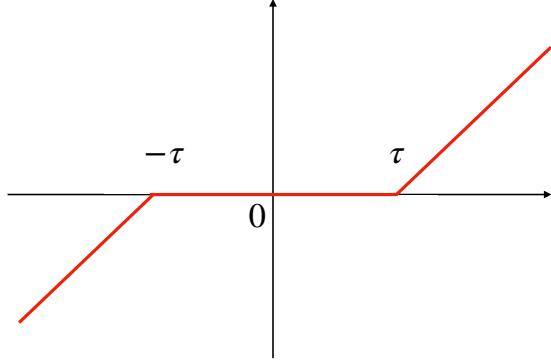


図 5.7 ソフトしきい値関数 $\text{sign}(x) \max\{|x| - \tau, 0\}$

と定義されます。

例えば、L1 正則化関数 $f(x) = |x| (f : \mathbb{R} \rightarrow \mathbb{R})$ に対応する近接写像は

$$\eta_\tau(x) := \text{prox}_{\tau f}(x) = \text{sign}(x) \max\{|x| - \tau, 0\} \quad (5.16)$$

と与えられ、ソフトしきい値関数と呼ばれます（図 5.7 参照）。

いま、目的関数 $f(\mathbf{x})$ が $f(\mathbf{x}) = g(\mathbf{x}) + h(\mathbf{x})$ という構造をしているものとします。ここで、 $g(\mathbf{x})$ と $h(\mathbf{x})$ はともに凸関数であると仮定します。無制約最小化問題

$$\text{minimize } g(\mathbf{x}) + h(\mathbf{x})$$

を解くための近接勾配法の反復式は

$$\mathbf{x}_{t+1} := \text{prox}_{\alpha h} (\mathbf{x}_t - \alpha \nabla g(\mathbf{x}_t)), \quad (5.17)$$

と与えられます。ここで、 $\alpha \in \mathbb{R}$ はスカラーパラメータです。いくつかの条件が満足されるならば、近接勾配法において探索点は目的関数の最小点に収束することが知られています。

より一般に近接写像を利用する最適化技法を近接法 (proximal method, proximal algorithm) と呼ぶことがあります。近接法については、文献 [29] にわかりやすい説明があります。最近、画像処理、信号処理でしばしば利用される **ADMM** も近接法の一種です。本稿の以下の部分では、近接勾配法に議論を絞ります。

圧縮センシング (4.2.3 節) における信号再現を行うために、信号再現問題を凸最適化問題として定式化する手法として、**LASSO** 定式化がよく知られています。LASSO 定式化では、スパース信号再現問題を

$$\hat{\mathbf{x}} := \operatorname{argmax}_{\mathbf{x} \in \mathbb{R}^n} \left(\frac{1}{2} \|\mathbf{y} - \mathbf{Ax}\|_2^2 + \lambda \|\mathbf{x}\|_1 \right), \quad (5.18)$$

と定式化します。ここで、左辺の $\hat{\mathbf{x}}$ は推定結果です。また、 $\lambda (> 0)$ は正則化定数であり、L1 正規化項の効きの強さを調整します。L1 正規化項はスパース解を選好するため、LASSO を解くことによりスパース信号再現が可能となります。

LASSO を解くためにはどのような凸最適化技法を利用しても問題はありませんが、ここでは、近接勾配法を利用して解くことを考えてみましょう。いま、 $h(\mathbf{x})$ を誤差項 (受信ベクトルと再現候補との間の誤差) として

$$h(\mathbf{x}) = \frac{1}{2} \|\mathbf{y} - \mathbf{Ax}\|_2^2 \quad (5.19)$$

と置き、そして $g(\mathbf{x})$ として L1 正則化項

$$g(\mathbf{x}) = \|\mathbf{x}\|_1 \quad (5.20)$$

を取ります。

ここで、式 (5.17) で与えられる近接勾配法を適用します。まず 2 次の項の勾配ベクトルは

$$\nabla h(\mathbf{x}) = -\mathbf{A}^T(\mathbf{Ax} - \mathbf{y}). \quad (5.21)$$

と与えられます。すでに述べたように L1 正規化 $\lambda \|\mathbf{x}\|_1$ の近接写像は、ソフトしきい値関数となりますので近接勾配法のプロセスは、次の反復式

$$\mathbf{r}_t = \mathbf{s}_t + \beta \mathbf{A}^T(\mathbf{y} - \mathbf{As}_t) \quad (5.22)$$

$$\mathbf{s}_{t+1} = \eta_\tau(\mathbf{r}_t), \quad (5.23)$$

でまとめることができます。ここで、 β と τ は外部から与えられるパラメータです。このアルゴリズムは **ISTA (Iterative Soft Threshold Algorithm)** と呼ばれるアルゴリズムです [5]。

ISITA の反復式の中に現れる定数 β と τ の設定は ISTA によるスパース再現プロセスの振る舞いに大きく影響を与えます。適切なパラメータ設定 [5] をすることにより、ISTA により漸近的には LASSO 解を再現できることが知られています。ただし、 τ と β の選択は非常にシビアで、AMP など最近よく利用されるアルゴリズムと比較すると収束はそれほど速くはありません。

4.2.4 節で紹介した `ISITA.ipynb` に含まれるスパース信号再現アルゴリズムは、ここで述べた ISTA を深層展開の考え方に基づき変更し、学習可能型 ISTA としたものです

(TISTA[13, 14] の簡易版とみることができます。)。変更の方針として、決定の難しいパラメータである上記のパラメータ β と τ を学習可能パラメータ (ISTA.ipynb のプログラム内では、`beta`, `lambda` になっている点には注意してください) 変更します。また、反復ごとに違う値をとり得るようにしています。これだけの簡単な変更の後に学習を行うと学習型 ISTA の収束速度は元の ISITA のそれに比べて著しく向上します。

5.4 LISTA の登場

本節では、深層展開の歴史的な経緯について、少し説明します。具体的な研究として、深層展開を初めて利用したのは、Gregor と LeCun [10] による **LSITA**(Learned ISTA) の研究です。LISTA も ISTA [5] のバリエーションのひとつであり、学習可能パラメータが反復処理内に含まれています。もとの ISTA と比べて非常に高い信号再現能力を LISTA が持つことを彼らは実験結果として示しています。

LISTA の詳細に入るまえに ISTA の勾配ステップを復習しておきましょう。ISTA の勾配ステップは

$$\mathbf{r}_t = \mathbf{s}_t + \beta \mathbf{A}^T (\mathbf{y} - \mathbf{A}\mathbf{s}_t) \quad (5.24)$$

として \mathbf{r}_t の値を更新します。ここで、この更新式の右辺に注目すると状態ベクトル \mathbf{s}_t と観測ベクトル \mathbf{y} の線形変換の和になっていることが分かります。Gregor と LeCun [10] は、ISTA の勾配ステップに学習可能パラメータとして 2 種類の行列 $\mathbf{B}_t \in \mathbb{R}^{n \times n}, \mathbf{S}_t \in \mathbb{R}^{n \times m} (t = 1, \dots, T)$ を導入し、 \mathbf{s}_t と \mathbf{y} の線形変換の和を構成しました。彼らの提案アルゴリズムである LISTA の反復式は

$$\mathbf{r}_t = \mathbf{B}_t \mathbf{s}_t + \mathbf{S}_t \mathbf{y} \quad (5.25)$$

$$\mathbf{s}_{t+1} = \eta_{\tau_t}(\mathbf{r}_t) \quad (5.26)$$

となります。これらの行列は、学習プロセスにおいて損失関数値が小さくなるように調整されます。LSITA により、ISTA と比較して著しい収束スピードの改善が得られることが実験的に示されています [10]。この結果は、深層展開によるアルゴリズム改善の可能性を強く示唆しています。LISTA に触発された形で、例えば AMP を学習可能にした LAMP[7] など、後続の関連研究が数多く生まれています。

LeCun は近年、深層展開を含む概念として微分可能プログラミングを提唱しています。彼は、自身のブログに

“... people are now building a new kind of software by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization...”.

と書いています。深層ネットワークだけではなく、微分可能なパラメータを含む関数からなるプログラム全体が標準的な深層学習技術により訓練可能であること、また、データ駆動的にアルゴリズムを構成できる可能性が示唆されています。

5.5 TISTAについて

TISTA[13, 14]は、ISTAに各層ごとにひとつの学習可能パラメータを導入するという形で構成された非常にシンプルなISTAの派生アルゴリズムのひとつです。また、OAMPにも強く影響を受けています。TISTAの反復式は次の通り与えられます：

$$\mathbf{r}_t = \mathbf{s}_t + \gamma_t \mathbf{W}(\mathbf{y} - \mathbf{A}\mathbf{s}_t), \quad (5.27)$$

$$\mathbf{s}_{t+1} = \eta_{MMSE}(\mathbf{r}_t; \tau_t^2), \quad (5.28)$$

$$v_t^2 = \max \left\{ \frac{\|\mathbf{y} - \mathbf{A}\mathbf{s}_t\|_2^2 - m\sigma^2}{\text{tr}(\mathbf{A}^T \mathbf{A})}, \epsilon \right\}, \quad (5.29)$$

$$\tau_t^2 = \frac{v_t^2}{n}(n + (\gamma_t^2 - 2\gamma_t)m) + \frac{\gamma_t^2 \sigma^2}{n} \text{tr}(\mathbf{W}\mathbf{W}^T). \quad (5.30)$$

ここで、 $\gamma_t \in \mathbb{R}$ ($t = 1, 2, \dots, T$) が学習可能パラメータとなります。式(5.27)(5.28)がちょうど更新式に対応しています。行列 \mathbf{W} は、観測行列 \mathbf{A} のムーア・ペンローズ疑似逆行列です。関数 η_{MMSE} は MMSE 推定関数です。詳細については、文献[13, 14]を参照してください。また、TISTAのPyTorch実装コードが公開されています：

<https://github.com/wadayama/TISTA>

実装に興味をお持ちの方はこちらをご参照ください。

TISTAは、現時点では知られている強力なスパース信号再現アルゴリズム(AMP, LISTA, LAMP, OAMP)よりも高速な収束特性を示すことが実験的に示されています。LISTAとTISTAは、どちらもISTAをベースとしていますが、学習可能パラメータの入れ方が大きく異なっています。LISTAは、先に見たように各反復において、2つの行列型の学習可能パラメータが導入されています。一方、TISTAは各反復においてスカラーの学習可能パラメータがただ一つ入れられています。すなわち、反復あたりの学習可能パラメータ数(スカラーを単位に数える)は $O(N^2)$ (LISTA)と $O(1)$ (TISTA)となり、学習可能パラメータ数が大きく違います。学習可能パラメータ数が小さいことは、(1) 学習プロセス時間、(2) 学習の安定性(再現性)の点で大きな優位点となります。実際にTISTAは、LISTAに比べて必要となる学習時間は非常に短いことが確認されています。

5.6 複素学習型ISTA

無線通信(物理層)信号処理では、ベースバンド信号が複素数であることから複素数計算が重要となります。ここでは、PyTorchで複素数を扱う方法について解説します。PyTorch自体は複素数を直接扱うことはできません^{*1}がここでは、複素テンソル計算をサポートするPyTorchライブラリ関数を導入します。

先に紹介してきたISTAは、スパース信号再現の文脈で議論されることがほとんどなので、一見するとスパース再現専用のアルゴリズムに見えるのですが、縮小関数の部分を適切に取り替えるとMIMOやOFDMにおける離散変調信号の信号推定にも利用が可能です。これらのシステムでは、複素受信ベクトルは

$$\mathbf{y} = \mathbf{Ax} + \mathbf{w} \quad (5.31)$$

とモデル化できます。無線通信の場合は、行列 \mathbf{A} が複素行列($\mathbf{A} \in \mathbb{C}^{m \times n}$)となる点と \mathbf{x} の各要素が複素信号点(例えば、QPSK, PSK, QAMの信号点配置に含まれる複素点)となる点が本稿でこれまでに議論してきた圧縮センシングとの差異となります。

このような差異はあるものの、最適化問題としてみたときに無線通信に関する信号検出問題と実数体上のスパース信号再現問題の間には本質的な違いはありません。これが本稿で近接勾配法の解説にスペースを割いている理由です。深層展開を考える・考えないに関わらず、信号検出問題を考える上で近接勾配法(より一般に近接法)の知見は重要です。したがって、式(5.31)で与えられるモデルにおける信号再現問題に対して、ISTAを含めて近接法の適用を考えるのは極めて自然な流れです。この用途においては、従来のISTAはパラメータ設定が困難であり、他の競合アルゴリズムに対して収束速度が相対的に非常に遅いことから、それほど興味が持たれてきたわけではないようです。

しかし、TISTAや学習型ISTAのように、深層展開によりパラメータ設定の問題と収束速度の問題が解決できるようになってきた現在は、複素体上での近接勾配法(もしくはより広く近接法)を詳細に再検討するよいタイミングです。

レポリトリ上の`fftista.ipynb`(このコードはGoogle Colab上での実行はできません)は、複素体上で動作する学習型ISTAの参考コードです。このコードは複素テンソル計算ライブラリ`complexlib.py`を利用しています。図5.8は、`complexlib.py`内で定義されている複素テンソル計算に関する関数群の一例です。仕組みは単純で、実部を表すテンソルと虚部を表すテンソルの組(tuple)を複素テンソルとしています。関数の利用法

^{*1} 複素数をネーティブに取り扱うことができるPyTorchの開発が一部で進められているようですが、しばらくは公式版に取り入れられることは無さそうな状況です。

```

3
4 #=====
5 # Elementary functions
6 #=====
7
8 # tensor addition
9 def add(X, Y):
10     return (X[0] + Y[0], X[1] + Y[1])
11
12 # tensor subtraction (X - Y)
13 def sub(X, Y):
14     return (X[0] - Y[0], X[1] - Y[1])
15
16 # Hermitian transpose
17 def ht(X):
18     return (X[0].t(), -X[1].t())
19
20 # tensor multiplication
21 def mm(X, Y):
22     Z_re = torch.mm(X[0], Y[0]) - torch.mm(X[1], Y[1])
23     Z_im = torch.mm(X[0], Y[1]) + torch.mm(X[1], Y[0])
24     return (Z_re, Z_im)
25
26 # scalar multiplication
27 def scalar_mul(a, X):
28     return (a * X[0], a * X[1])
29
30 # matrix inverse
31 def inverse(X):
32     X_re = X[0]
33     X_im = X[1]
34     X_re_inv = torch.inverse(X_re)
35     tmp = torch.mm(X_im, X_re_inv)
36     tmp = torch.mm(tmp, X_im)
37     Z_re = torch.inverse(X_re + tmp)
38     tmp = -torch.mm(X_re_inv, X_im)
39     Z_im = torch.mm(tmp, Z_re)
40     return (Z_re, Z_im)

```

図 5.8 複素テンソル計算ライブラリ `complexlib.py`: 複素テンソルに対する基本演算に関する関数が含まれています。

については、関数のコードを見ればすぐに分かることと思いますので、ここでの説明は割愛します。

`fftista.ipynb` では通信路としては複素 AWGN 通信路を仮定しています。送信信号は、信号点配置は 8PSK から一様ランダムに選択されたのち、IFFT をかけて後に通信路へと送出されます^{*2}。信号長(ベクトル長)は n としています。

図 5.9 に複素学習型 ISTA のモデル定義部を示しています。ISTA に勾配ステップにててくる A と A^H (複素の場合は、エルミート転置になります) とベクトルの積の計算には FFT と IFFT を利用しています。このように複素体上の ISTAにおいて、観測行列が DFT(または IDFT) 行列の場合には、FFT を勾配ステップで利用することにより、効率的な計算が可能となります。離散信号に対応する複素縮小関数 (`self.c_shrinkage`) の詳細については、関連論文 [30] 内の説明をご参照ください。このモデルの定義をみると `ISTA.ipynb` のコアの部分はほぼ同じ形になっていることがわかると思います。コア部分はシンプルですが、信号再現能力は強力です。

複素学習型 ISTA による信号推定の結果(信号散布図)を図 5.10 に示します。比較対象

^{*2} サイクルプレフィックなど何もついていない、原始的 OFDM 方式と見ることもできます。

```

1  class CISTA(nn.Module):
2      def __init__(self, max_itr):
3          super(CISTA, self).__init__()
4          self.beta = nn.Parameter(0.1*torch.ones(max_itr)) # 学習可能ステップサイズ
5          self.lam = nn.Parameter(0.1*torch.ones(max_itr)) # 収縮関数制御パラメータ
6      def c_shrinkage(self, x, var_mat):
7          eps = 1e-10
8          num_re = torch.zeros(mbs, n).to(device)
9          num_im = torch.zeros(mbs, n).to(device)
10         deno = torch.zeros(mbs, n).to(device) + eps
11         for i in range(M):
12             r = (x[0] - point[i][0])**2 + (x[1] - point[i][1])**2
13             f = torch.exp(-r/var_mat)
14             num_re += point[i][0] * f
15             num_im += point[i][1] * f
16             deno += f
17         return (num_re/deno, num_im/deno)
18     def forward(self, y, num_itr):
19         s = c.zeros(mbs, n) # 初期探索点
20         for i in range(num_itr):
21             tmp = c.sub(y, c.ifft(s))
22             tmp2 = c.scalar_mul(self.beta[i], c.fft(tmp))
23             r = c.add(s, tmp2)
24             s = self.c_shrinkage(r, self.lam[i])
25         return s

```

図 5.9 複素学習型 ISTA のモデル定義部: コア部分は通常の ISTA の形ですが、すべての行列・ベクトルは複素行列・複素ベクトルになっています。

として、同条件において ZF 推定 (受信ベクトルに FFT を適用) を行った結果得られた推定値も同図に示します。ZF 推定と比較して複素 ISTA の推定精度の良さが見て取れます。

論文 [30] では、 $y = f(Ax) + w$ という形でモデル化できる系における複素信号再現問題を論じています。ここで、 f は要素毎に作用する複素非線形関数です。ウィルティンガ微分の連鎖則を利用することで、適切な勾配ステップを構成することができます。関連の参考コードは

<https://github.com/wadayama/C-TISTA>

に置いてあります。量子化・信号クリッピングなど幅広い問題に適用が可能です。

5.7 近接法と深層展開

本章では、特に近接勾配法 (射影勾配法も近接勾配法の一種です) に焦点を当てて、深層展開の適用について議論をしてきました。近接勾配法は、それ自体が強力な最適化原理であり、広い応用範囲を持つ数理最適化技法です。近接勾配法に対して、深層展開を適用

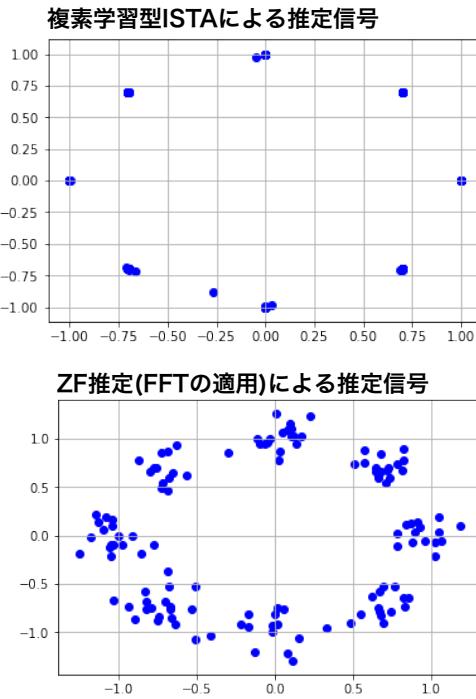


図 5.10 複素学習型 ISTA による信号推定: IFFT に基づく ZF 推定の結果に比べて複素学習型 ISTA は優れた信号再現を可能にしています。

することで、より強力なアルゴリズムを構成することができます。例えば、勾配ステップにおけるステップサイズや縮小ステップにおける正則化係数などをデータに基づき学習することにより、他の手法では困難であった内部パラメータの最適化が可能となります。

図 5.11 に深層展開に基づくアルゴリズムデザインの三階層を示します。一番下の層は、問題の数理最適化問題としての定式化、問題に適した近接法の選択と反復式の導出となります。例えば、スパース信号再現問題における LASSO 定式化と ISTA の導出などがこの階層での仕事とみることができます。

第二層は、学習可能パラメータの埋め込みです。学習可能パラメータの選択には、豊富な自由度があるため、この部分が最も設計者の考え方反映される部分となります。例えば、LISTA の場合は行列の学習可能パラメータを導入されていましたが、TISTA の場合はスカラーのステップサイズパラメータが学習可能パラメータとなっていました。この部分はアルゴリズム設計プロセスにおいて、最終的なアルゴリズムの限界性能と学習プロセスの計算量を決める重要な部分となります。

第三層は、学習プロセス (訓練) により学習可能パラメータを調整するフェーズです。本稿で暗黙のうちに仮定されていたオフライン学習の設定では、ランダムに生成されたデータセットに基づき、確率的勾配法を利用してパラメータの学習を行います。または、

実環境から入手したデータに基づいて学習を行うことも可能です。この部分は、理想的にはデータセットさえあれば言わば全自動で進む部分ですが、実際にはある程度のハイパラメータ（学習率、ミニバッチサイズなど）の人手による調整が必要となります^{*3}

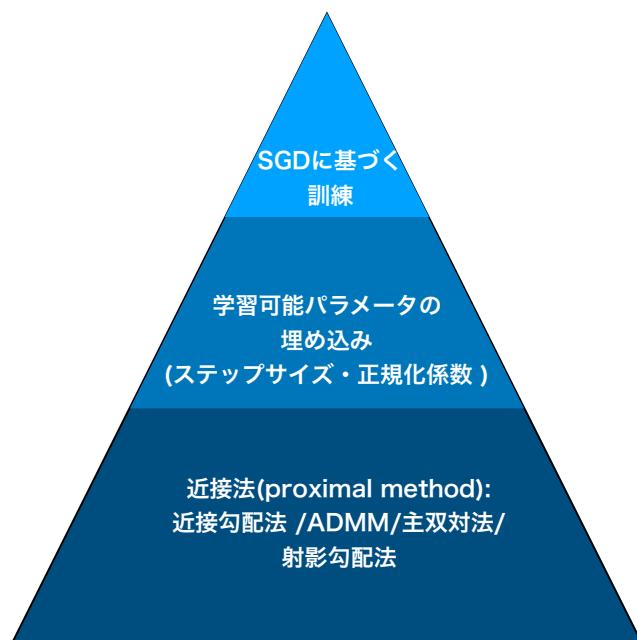


図 5.11 近接法と深層展開に基づくアルゴリズムデザインの三階層: 第一階層は数理的・演繹的です。第二階層の部分は現状ではアート(技芸)に属すると思われます。第三階層はデータ駆動的です。

^{*3} ハイパーパラメータ調整については、近年ベイズ最適化に基づく技術の進展が顕著です。近い将来、この部分も全自動化される可能性は高いと思われます。

第6章

むすび

本稿は、最初に書いたように「無線通信 × 深層学習」研究スタートアップガイドとしてまとめました。かなり説明が省かれている部分も実はあります、コードを Github のレポジトリで公開していますので、その部分はコードを読み解いていただけたと理解が深まるかと思います。

深層学習関連の研究を進めるために心得ておいたほうがよいと思われることをいくつ述べて、むすびとしたいと思います。深層学習関連の研究を進めるためには私たちの慣れている研究の方法論（原理→アルゴリズム：演繹的）とは異なる方法論（データ→アルゴリズム：帰納的）が必要であり、実験科学的アプローチが求められます。具体的に研究を進める上でのポイントとしては、

- (1) 説得力のあるベースラインを用意する
- (2) ハイパーパラメータ（学習率など）の調整に時間をかけすぎない
- (3) Github などで公開されているコードを読む・利用させていただく
- (4) 学習して得られた結果を解釈する努力をする
- (5) 実験の再現性を確保する

などが挙げられます。特に(1)のベースラインは重要です。深層学習を利用したアルゴリズムの性能を評価する場合に、既存方式の最も性能のよいものと比較して、提案方式の優位性を明確に示す必要があります。ベースラインとの比較が曖昧だと学習により得られたアルゴリズムの良し悪しの客観評価ができなくなり、単に「学習ができました」というだけの主張になってしまいます^{*1}。ハイパーパラメータ調整(2)も拘りだすときりがなく、労力に対して得られるものが少ないので、考察や他の実験の実施などに時間を割くほうが

^{*1} この部分はドメインの専門知識が要求される部分で、その分野の専門知識がないと意味あるベースラインが準備できません。

有益です^{*2}。(3)ですが、深層学習関連のネット上のリソースは膨大です。機械学習分野の慣習で ArXiv に論文を投稿したのと同時に Github 上でコードが公開されることも珍しくありません。勉強のソースとなるもの、使えるものはどしどし使わせていただきましょう^{*3}。論文を読んだだけでは分からぬ詳細が理解できる場合も多いです。(4)ですが、学習結果を読み解く努力をしてみるのは無益ではありません。学習結果について解釈がまったくできない場合も多いですが、場合によっては理論的成果につながる糸口を学習結果から見いだせる可能性もあります。(5)は一番最後に述べていますが、一番重要かもしれません。再現性を考慮して実験を計画し、実験記録を残し、再現実験できる形で論文に実験条件の記載を行う、また可能ならば実験コードを公開する、などの実験の再現性を確保する営みが研究コミュニティにとって有益な貢献につながると思います。

冒頭で述べた 5G/6G 向けた複雑な無線環境の到来、AI・機械学習技術への需要増大に応えるために、高度な無線通信技術への社会的ニーズは高まりつつあります。深層学習技術を利用する無線通信技術に関する研究発表が活性化してきたのは、ここ 2,3 年のこととで、まだまだ価値あるアイデアが発見されず埋もれている可能性は高いのではないかでしょうか。分野横断的な研究姿勢を持つ若手研究者の本分野への参入を期待したいと思います。

謝辞

本チュートリアル講演の機会を与えてくださった MIKA2019 の実行委員会の皆様に感謝申し上げます。大阪市立大学の林和則氏には、様々な機会に無線通信技術についてご教示いただいています。名古屋工業大学の高邊賢史氏との日頃の議論は本稿の内容に対して大いに影響を与えています。また、同氏との共同研究 (TISTA, C-TISTA) の内容が本書に深く反映されています。両氏に厚く感謝の意を表します。本稿の研究の一部は、科研費基盤 (A)17H01280、科研費基盤 (B)19H02138 に基づきます。

^{*2} ベイズ最適化に基づくパラメータ調整ツール (Oputuna など) の利用を考えてみてもよいでしょう。

^{*3} ただし、実験に Github 上のコードを利用した場合には利用したコードに対する適切なクレジット (引用) を論文に明記するべきかと思います。

第7章

付録

7.1 文献ガイド

この分野の研究をスタートを目指す初学者の方にお勧めできる文書、文献、書籍についてまとめています。

- 「信号・データ処理のための行列とベクトル」田中 聰久、コロナ社
研究をスタートしようとする初学者の方(特に学生さん)にとっては、無線通信、機械学習のどちらの分野にも線形代数・確率に関する知識が必要となります。無線通信分野では、複素数の取り扱いについても馴染んでおく必要があります。最近出版されたこの本ではこれらの事項が明快に説明されていて素晴らしい教科書です。まずはこれを読みましょう。
- 「初学者のための無線通信信号処理入門」林 和則、
http://www.ip.info.eng.osaka-cu.ac.jp/~kazunori/paper/rcs201707_handout.pdf
無線通信の初学者の方へはこの資料をお勧めできます。他ではあまり記述されていない事項も(ウィルティンガー微分など)も平易に分かりやすく書かれています。
- “A User’s Guide to Compressed Sensing for Communications Systems,” K.Hayashi, M.Nagahara, and T.Tanaka, IEICE Trans.Commun., vol.E960B, No.3 March, pp. 685–712, 2013.

NOMA や過負荷 MIMO, 1-bit 量子化 MIMO などの過負荷系・劣決定系の通信技術の重要性が今後高くなるのではないかでしょうか。劣決定系の代表例である圧縮センシングに関する理解はそのような通信技術を理解するための助けになります。このサーベイ論文では、圧縮センシングの考え方から始まり、再現アルゴリズムの紹介、無線技術と圧縮センシングの関わりについてなどが分かりやすく述べられています。

ます。

- “Convex Optimization,” S.Boyd, L.Vandenberghe, Cambridge University Press

無線通信・機械学習のどちらにしても最適化に関する理解を得ておくのが、(急がば回れということで) よろしいかと思います。学生さんには少々ヘビーかもしれませんがこの本を挙げておきたいと思います。

- “Information Theory, Inference and Learning Algorithms,” David J.C.MacKay, Cambridge University Press

情報理論・機械学習・ニューラルネットワークなどをつなぐ道筋が著者独自の語り口で興味深く語られます。ベイズ的立場からのニューラルネットワークが扱われているなど、今読んでも全く古びない内容です。ちなみに著者の David MacKay は LDPC 符号の再評価に大きな影響を与えました。

- 「Python と Keras によるディープラーニング」,Francois Chollet, マイナビ出版

深層学習の入門本は山程でていますので、まずは自分に合いそうなものを選べばよいと思いますが、内容の薄いものもありますので注意が必要かもしれません。この本は、Keras の設計者の F.Chollet の書いた本です。機械学習・深層学習の勉強を始めた初心者が押さえておくべきことがしっかり書かれており、また筆者の機械学習の深い経験がじみ出る記述もあり大変勉強になります。

- ゼロから作る Deep Learning —Python で学ぶディープラーニングの理論と実装, 斎藤 康毅, オライリー・ジャパン

巷で人気の高い本です。フレームワークやライブラリに頼らずに一から説明するという立場で書かれています。私も自分で C++ のバックプロパゲーションのプログラムを書いたことがあります、本書の記述には何度も助けられました。フレームワークは便利ですが、一度はこの本を見ながらフルスクラッチ実装に挑戦してみるのもよい経験になります。

- “Deep Learning,” I. Goodfellow, Y. Bengio, Aaron Courville, MIT Press. GAN の作者の Goodfellow、深層学習の立役者で大御所の Bengio らの世界的定番の教科書です。いろいろ参考になることが書かれていて勉強になる素晴らしい内容なのですが、ある程度、ニューラルネットワークの基礎を押さえてから(2 冊目として) 読んでいくほうがよいと思います。

- 「つくりながら学ぶ! PyTorch による発展ディープラーニング」, 小川雄太郎, マイナビ出版

PyTorch の中級者向けの本で、実践的なコードが沢山載っています。こう書けばいいのか、と思わされることが多いです。内容は、ちょっと通信分野とは離れます

が GAN の実装や高度な画像認識手法も学べます。また、データローダの作成法など詳しく書かれていますので、この本と PyTorch 本家サイトのチュートリアルと合わせてコードを読んで（写経して）いけば、かなり PyTorch に強くなれる（加えて先端の深層学習系技術にふれることができる）のではないでしょうか。

- “PyTorch Documentation, <https://pytorch.org/docs/stable/index.html>

本家 PyTorch サイトに置いてあるレファレンスマニュアルです。ある意味で一番役に立ちます。本稿で述べられているタイプの研究を進めていく上で、フレームワークの詳細（何が技術的に可能で、何が不可能なのか）について習熟しておくことが研究の効率を大きく左右します。リファレンスマニュアルを時折眺めて、使える関数を増やしていくとよいでしょう。

7.2 LDPC 符号関連のサンプルコード

レポジトリの `ldpclib.py` には、LDPC 符号関連のコードを書くときに便利な関数がいくつか含まれています。

- spmat 形式の検査行列ファイルの読み込み
- 検査行列の生成
- 生成行列の生成

などのための関数が含まれており、LDPC 符号の符号化・パリティ検査などが可能となります。この `ldpclib.py` の利用例として、**Noisy GDBF** 復号法の PyTorch 実装を `NoisyGDBF.ipynb` として、レポジトリに置いていますので、ライブラリ関数の具体的な使い方などはそちらを参照してください。

7.3 実験の再現可能性を高めるために

むすびでも書きましたが、PyTorch を利用した実験では、可能な限り再現可能な形で実験データを残すことが望ましいと考えられます。再現可能性に関する記述が PyTorch 本家サイトにあります。

<https://pytorch.org/docs/stable/notes/randomness.html>

特に GPU を利用している場合は完全な実験の再現は難しいようですが、例えば PyTorch 内で利用される乱数の種を

```
import torch
torch.manual_seed(0)
```

として固定しておくのはひとつの方策です。Numpy や Python 組み込みの乱数関数を利用する場合は、そちらの乱数の種の初期化も行うとよいでしょう。

実験結果を論文・レポートにまとめる際には可能な限りハイパーパラメータを記載することが望ましいと考えます。ハイパーパラメータの代表的なものは次のとおりです：

- ミニバッチサイズ
- オプティマイザの選択・学習率などのパラメータ
- 学習率のスケジューリングをする場合はそのルールとパラメータ詳細
- 学習可能パラメータの初期値
- 学習（訓練）ループ回数・エポック数
- モデルの構造・層数・活性化関数の種類・ユニット数
- 正則化・ドロップアウトを利用する場合には、その関連パラメータ

自分でコードを書く場合は、Jupyter Notebook の場合は、これらのハイパーパラメータを一箇所にまとめて記述するのがよいと思います。通常の Python のコードとする場合は（論文データとして利用する系統的な実験や複数のマシンにジョブとして投入するにはこちらが便利です）コマンドライン引数として、これらのハイパーパラメータを与える形にしておきます。実験用のシェルスクリプト、例えば

```
python hoge --batchsize=128 --admlr=0.001 ...
```

を含むシェルスクリプトを用意しておけば、それ自身がその実験に関するハイパーパラメータの記録となります。

また、忘れやすいですが学習可能パラメータの初期値も重要なハイパーパラメータですので、自分で明示的に初期化を行い、その値を記録しておくほうがよいでしょう。本稿で示した参考コードでは、コードを単純にするために線形レイヤの重み行列・バイアスの初期値の初期化はライブラリのデフォルト動作におまかせでサボっていました。例えば次のようにして、インスタンス化の瞬間に初期化を行っておくとよいでしょう (MIMO.ipynb の例)。

```
class Net(nn.Module): # nn.Module を継承
    def __init__(self): # コンストラクタ
        super(Net, self).__init__()
        self.detector = nn.Sequential(
            nn.Linear(n, h), # W_1, b_1,
```

```
nn.ReLU(), # 活性化関数として ReLU を利用
nn.Linear(h, h), # W_2, b_2
nn.ReLU(),
nn.Linear(h, n) # W_3, b_3
)
self.reset_parameters()

def reset_parameters(self):
    for name, param in self.named_parameters():
        if 'weight' in name:
            print(name)
            param.data.uniform_(-0.01, 0.01)
        if 'bias' in name:
            print(name)
            param.data.uniform_(-0.02, 0.02)
```


参考文献

- [1] G. E. Hinton, R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” *Science*, vol. 313, no. 5786, pp. 504-507, Jun. 2006:
- [2] A. Krizhevsky, I. Sutskever, G. E. Hinton, “Imagenet classification with deep convolutional neural networks.” *Advances in Neural Inf. Proc. Sys.* 2012, pp. 1097-1105, 2012.
- [3] A. Chambolle, R. A. DeVore, N. Lee, and B. J. Lucier, “Nonlinear wavelet image processing: variational problems, compression, and noise removal through wavelet shrinkage,” *IEEE Trans. Image Process.*, vol. 7, no. 3, pp. 319–335, Mar, 1998.
- [4] G. E. Dahl, D. Yu, L. Deng and A. Acero, ”Context-Dependent Pre-Trained Deep Neural Networks for Large-Vocabulary Speech Recognition,” in *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 1, pp. 30-42, Jan. 2012.
- [5] I. Daubechies, M. Defrise, and C. De Mol, “An iterative thresholding algorithm for linear inverse problems with a sparsity constraint,” *Comm. Pure and Appl. Math.*, col. 57, no. 11, pp. 1413-1457, Nov. 2004.
- [6] D. L. Donoho, A. Maleki, and A. Montanari, “Message-passing algorithms for compressed sensing,” *Proceedings of the National Academy of Sciences*, vol. 106, no. 45, pp. 18914–18919, Nov. 2009.
- [7] M. Borgerding and P. Schniter, “Onsager-corrected deep learning for sparse linear inverse problems,” *2016 IEEE Global Conf. Signal and Inf. Proc. (GlobalSIP)*, Washington, DC, Dec. pp. 227-231, 2016.
- [8] N. Farsad, and A. Goldsmith, “Neural Network Detection of Data Sequences in Communication Systems,” *IEEE Transactions on Signal Processing*, vol. 66, Issue: 21, pp. 5663-5678, Nov., 2018.
- [9] I. Goodfellow, Y. Bengio, A. Courville “Deep learning,” *The MIT Press*, 2016.

- [10] K. Gregor, and Y. LeCun, “Learning fast approximations of sparse coding,” *Proc. 27th Int. Conf. Machine Learning*, pp. 399–406, 2010.
- [11] T. Gruber, S. Cammerer, J. Hoydis, and S. ten Brink, “On deep learning-based channel decoding,” arXiv:1701.07738, 2017.
- [12] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, “Deep neural networks for acoustic modeling in speech recognition: The Shared Views of Four Research Groups,” IEEE Signal Processing Magazine, vol. 29, no. 6, pp. 82-97, Nov. 2012.
- [13] D. Ito, S. Takabe, and T. Wadayama, “Trainable ISTA for sparse signal recovery,” IEEE International Conference on Communications (ICC2019), Workshop on Promises and Challenges of Machine Learning in Communication Networks, Kansas city, May, 2018.
- [14] D. Ito, S. Takabe, and T. Wadayama, “Trainable ISTA for sparse signal recovery,” IEEE Trans. Signal Processing, vol. 67, no. 12, pp. 3113-3125, Jun., 2019.
- [15] T. Wadayama and S. Takabe, “Deep learning-aided trainable projected gradient decoding for LDPC Codes,” IEEE Int. Symposium on Information Theory (ISIT); arXiv:1901.04630, 2019.
- [16] S. Takabe, M. Imanishi, T. Wadayama, and K. Hayashi, “Deep learning-aided projected gradient detector for massive overloaded MIMO channels,” IEEE International Conference on Communications (ICC2019), ; <https://arxiv.org/abs/1806.10827>, 2019.
- [17] S.Takabe, M.Imanishi, T.Wadayama, R.Hayakawa, K.Hayashi, “Trainable Projected Gradient Detector for Massive Overloaded MIMO Channels: Data-driven Tuning Approach”, IEEE Access, DOI: 10.1109/ACCESS.2019.2927997, July 2019.
- [18] E. Nachmani, Y. Beéry and D. Burshtein, “Learning to decode linear codes using deep learning,” *2016 54th Annual Allerton Conf. Comm., Control, and Computing*, pp. 341-346, 2016.
- [19] T. O’Shea and J. Hoydis, “An introduction to deep learning for the physical layer,” IEEE Transactions on Cognitive Communications and Networking, vol.3, issue 4, pp. 563-575, 2017.
- [20] J. G. Proakis, “Digital communications,” McGraw-Hill Book, 1989.
- [21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” Nature, vol. 323, no. 6088, pp. 533-536, 1986.
- [22] T. Wadayama and S. Takabe, “Joint quantizer optimization based on neu-

- ral quantizer for sum-product decoder, ” IEEE Globecom, Abu Dhabi, ; arXiv:1804.06002, 2018.
- [23] A. Balatsoukas-Stimming and C. Studer “Deep unfolding for communications systems: a survey and some new directions,” arXiv:1906.05774, 2019.
 - [24] H. Sun, X. Chen, Q. Shi, M. Hong, Xiao Fu, and N. D. Sidiropoulos, “Learning to optimize: training deep neural networks for interference management,” IEEE Trans. on Signal Processing, vol.56, no.20, pp.5438-5453, Oct., 2018.
 - [25] J. Konečný, H. B. McMahan, D. Ramage, P. Richtárik, “Federated optimization: distributed machine learning for on-device intelligence,” arXiv:1610.02527, 2018.
 - [26] D. Guñduž, P. de Kerret, N. D. Sidiropoulos, D. Gesbert, C. Murthy, M. van der Schaar, “Machine learning in the air,” arXiv:1610.02527, 2018.
 - [27] N. Samuel, T. Diskin, and A. Wiesel, “Learning to detect,” IEEE Trans. Signal Process., vol. 67, no. 10, pp. 2554-2564, May. 2019.
 - [28] K.Hayashi, M.Nagahara, and T.Tanaka, “A user’s guide to compressed sensing for communications systems,” IEICE Trans.Commun., vol.E960B, No.3 March, pp. 685–712, 2013.
 - [29] N.Parikh and S.Boyd, “Proximal Algorithms,” Foundations and Trends in Optimization, Vol. 1, No. 3, Now Publishers, 2013.
 - [30] S.Takabe and T.Wadayama, “Complex Field-Trainable ISTA for Linear and Non-linear Inverse Problems,” <https://arxiv.org/abs/1904.07409>, 2019.