令和 3 年度茨城大学工学部情報工学科卒業研究論文

キーワード付与による画像キャプション生成

所属 情報工学科

著者 木村文飛 (18T4030R)

指導教員 新納浩幸教授

令和 4 年 2 月 1 日 (火)

令和 3 年度茨城大学工学部情報工学科卒業研究論文

キーワード付与による画像キャプション生成

著者

木村文飛 (18T4030R)

指導教員

新納浩幸教授

論文要旨

画像キャプション生成は入力された画像に対してその画像の説明文（キャプション）を生成するタスクである．画像へのキャプションを人手によって作成し学習する手法によって，近年目覚ましい発展を遂げている．しかし，キャプション生成の課題としてアノテーションに限りがあることがあげられ，従来の手法では人手でアノテーションされた画像を前提とする質のいいデータセットが必要となる．精度を高めるためにデータセットを増やすことはアノテーションの質と量，費用といったアノテーションコストの問題が生じる．英語のような教師データの多い言語と比較し，日本語のキャプション生成をする際にはより生じやすい問題となる．この問題は結果として，キャプションの精度を向上させるうえでのボトルネックとなる．

本論文では，入力として，対象の画像の他にその画像内にある物体名をキーワードとして学習に追加する手法を提案する，物体名を学習に追加することで，生成される日本語キャプションの精度を向上させることが目的である．

実験では，提案手法を適用したモデルとしなかったモデルを用意し，その 2 つのモデルを比較することで評価を行った．評価手法には CIDEr を採用し，数値の比較を行った．また，生成キャプション内のキーワード出現率などの調査を行うことで，提案手法の生成キャプションに対する効果を分析した．

結果として，提案手法は CIDEr のスコアを 0.7 改善でき，わずかではあるが効果があることが確認できた．また提案手法は物体名を学習に追加することで，物体検出が正確に行われた状況と酷似していると言えることから，画像キャプション生成の誤りが画像処理によるものか，言語処理によるものかを調査できるという長所をもつ．

提案手法がスコアを大きく改善できなかった要因として，与えられた複数の単語を用いて文を生成することは容易ではないことがあげられる．本論文の結果を踏まえて，今後は学習に追加するベクトルの形式を変化させて，実験を行う予定である．

目次

第 1 章 序論 5

第 2 章 関連研究 7

2.1 Show and Tell: A Neural Image Caption Generator 7

2.2 深層学習による日本語キャプション生成システムの開発 8

第 3 章 画像キャプション生成 9

3.1 Neural Network 10

3.2 Convolutional Neural Network 10

3.3 VGG16 11

3.4 Recurrent Neural Network 12

3.5 LSTM 13

3.6 MS-COCO 13

3.7 STAIR Captions 14

3.8 CIDEr 14

第 4 章 提案手法 15

第 5 章 実験 17

5.1 実験設定 17

5.2 モデル 19

5.3 評価手法 21

5.4 実験結果 22

5.5 出力例 24

第 6 章　考察 26

第 7 章　結論 28

参考文献 30

付録 32

A　プログラムリスト 32

第 1 章

序論

本論文では，画像からその画像の簡単な説明文（キャプション）を生成するタスクである画像キャプション生成について，キーワード付与の手法を提案する．

画像キャプション生成は計算機リソースの増加や，画像とキャプションのセットを人力で作成し機械学習する手法によって，画像からその画像の簡単な説明文（キャプション）を生成するタスクである．人力で作成された画像とキャプション文のセットを用いて学習する手法によって，近年目覚ましい発展を遂げている分野である．

画像キャプション生成における従来の手法の問題点として，十分なデータセットの用意が困難という点があげられる．画像とキャプション文のセットを人力で作成するアノテーションコストは非常に高く，量的な増加は時間と金銭的問題につながるためである．また，人力のアノテーションは一個人の感覚による面もあるため，教師データに揺れが生じ，学習の妨げとなりうる．データセットの問題に対する解決策として，別のデータセットを画像キャプション生成の学習に利用することを目的として，画像のキャプションを全く別のデータセットから用いて学習を教師なし学習 [1] [2] や，既存のデータセットを増やすことを目的として，マスク処理という，画像にノイズを加える処理を用いることで訓練データセットの種類を増やすデータ拡張 [3] などがある．本研究では，上記の研究のようにデータセットを増やすことで画像キャプションの精度を向上させるというアプローチではなく，学習に用いるベクトルに操作を加えることでキャプションの精度を向上させる．

また，画像キャプション生成の実装面の問題として，画像に対して不適当なキャプションが生成された際，原因の特定が難しい点があげられる．画像キャプション生成の実施には，画像畳み込みといった画像側の処理と，画像キャプション文の学習という言語側

の処理を一連の流れの中で行わなければならない．つまり，原因が言語側にあるか画像側にあるか不明確なのである．本論文ではこれらの問題点に着目し，訓練データのキャプションの中からキーワードを選定し，ベクトル化して学習情報として追加する手法を提案する．提案手法によりキーワードを選定することで，画像処理中の物体検出結果がキーワードの視点から検証しうることを実験にて示す．

実験では，訓練データの画像に MSCOCO のデータセット [4]，キャプションに STAIR Captions の日本語キャプション [5] を用いた．テストデータは MSCOCO のデータセットから一部を抜き出して使用した．提案手法により，CIDEr スコアが 0.7 改善した．また，提案手法を用いたことで物体検出の安定性が向上したことから，提案手法が，不適当なキャプションが生成された際の原因を言語側に絞ることができることを確認した．

第 2 章

関連研究

2.1 Show and Tell: A Neural Image Caption Generator

本研究では，Vinyals らが提唱した画像キャプション生成モデル，NIC の構造を参考にした．[6] このモデルは CNN（畳み込みニューラルネットワーク）をエンコーダ，LSTM（長短期記憶ニューラルネットワーク）をデコーダとして，それらを組み合わせることで画像キャプションを生成するモデルである．図 2.1 に NIC が行う画像キャプション生成の概略図を示す．

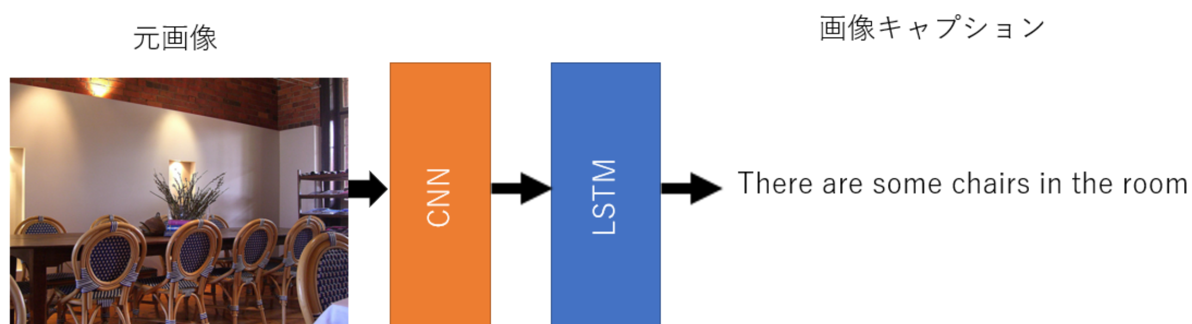

図 2.1: NIC の概略図

当時としては，NN（ニューラルネットワーク）ベースの手法は全く新しく，NIC は画像キャプション生成において，先行研究に比べ精度の面を大幅に向上させたと述べられている．この構造は機械翻訳に使われる encoder RNN と decoder RNN から着想を得ており，encoder RNN を CNN を置き換えたことで開発されたものである．NIC が開発される以前の研究の問題点として，個々の物体を訓練中に学習していても，見たことの

ない組み合わせについて適切なキャプションを生成出来ないとあげられている．それに対し，NIC には LSTM（長短期記憶ニューラルネットワーク）が用いられているため多様な表現が可能になったと述べられている．NIC が画像キャプションを生成する流れとしては，入力画像から事前学習させてある CNN（畳み込みニューラルネットワーク）によって特徴量を取り出したものと，画像の説明文を単語埋め込みによりベクトル化したものを，LSTM（長短期記憶ニューラルネットワーク）に入力することで文章を生成するものである．NIC は NN を用いて画像キャプション生成を行う先駆けとなったモデルであり，以降画像キャプション生成では NIC のモデル形式をベースとして，それらを発展させる形で研究されることが多い．

このモデルの特徴として，画像キャプション生成に利用可能なデータセットのサイズが大きくなればなるほど，出力するキャプションの精度が上がることは明らかであると述べられている．しかし，実際には利用可能なデータセットを増やすことは非常にコストがかかり容易ではない．

2.2　深層学習による日本語キャプション生成システムの開発

Vinyals らが提唱した NIC [6] を用いて，日本語のキャプションを生成する試みとして，小林らが作成した日本語キャプション生成システム，Deep Watcher [7] が存在する．

Deep Watcher に使用されたモデルは，NIC と同様に CNN と LSTM を用いてキャプションを生成する．データセットには画像 2000 枚に対し人力で日本語キャプションを付与したものを使用しており，データセットの言語が日本語になっても NIC と同様に学習させれば問題なく画像キャプション生成が行えることを示している．

Deep Watcher では，NIC と同様にモデルの作成に画像と画像の説明文がセットになったデータセットを用いて学習する必要があり，アノテーションコスト高く追加学習に高いコストがかかることが問題としてあげらている．また，データセットが不十分であったためモデルが過学習を起こしてしまい，画像キャプションモデルを一般化することが困難であることが述べられている．実際，実験結果では 58.4 %が内容と間違っているキャプションを生成したとあり，十分な結果が得られているとは言い難い結果となっている．

第 3 章

画像キャプション生成

画像キャプション生成とは，入力された画像からその画像を説明する簡単な文章（キャプション）を生成するタスクである．

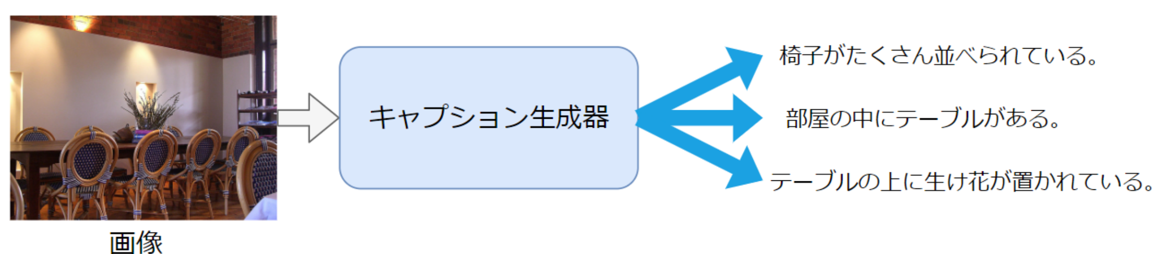

図 3.1: 画像キャプション生成の概略図

図 3.1 は画像キャプション生成について簡単に説明した図である．図のように，キャプション生成器へ画像を入力すると，キャプション生成器がもっともらしいキャプションを生成する．この時，画像内に存在するどの物体（テーブル，椅子，生け花）に注視するか，またどの程度の長さのキャプションを生成するかはキャプション生成器のモデルによって異なる．

近年，画像キャプション生成は Neural Network の普及とともに発展してきた分野であり，画像処理や物体検出といった画像分野だけではなく，自然言語処理も合わせて用いる必要がある分野である．画像キャプション生成の初期では，画像に映っているものについて出力を行う画像認識のタスクを応用することで，画像に映っている物や事柄に関する単語を複数用いて文章を生成する手法が一般的であった．しかし，単語が複数推

定されたとしても，それらの関係性がわからないため，単語をテンプレート文に当てはめることでキャプション生成を行う必要があった．当然，この手法ではテンプレート文に似たキャプションしか生成することが出来ず，不自然なキャプションが生成されてしまうことがあった．しかし，Recurrent Neural Network が開発され，従来の手法よりも生成することのできる文章がより自然なものとなると，これを画像キャプション生成に適応することで，従来の生成キャプションよりもスコアの高いキャプションが生成できるようになった．現在の画像キャプション生成の手法として，画像処理に Convolutional Neural Network，言語処理に LSTM を使う手法がベーシックであり，データセットとして MS-COCO，キャプション生成特有の評価手法として CIDEr が存在する．画像キャプション生成の派生研究として，キャプションから画像を生成するタスクや，画像を橋渡しにすることで異なった言語間の検索や翻訳を行う言語横断といったタスクが存在し，非常に将来性の高い分野といえる．

3.1　Neural Network

Neural Network（NN）は機械学習の手法の一つである．現在機械学習の言う分野において最も主流な機械学習手法であり，様々な研究や改良が行われているモデルでもある．Neural Network は入力層，隠れ層，出力層から構成され，また各層は複数のニューロンから形成される．ニューロンは重みと呼ばれる数値を持っており，これを更新していくことで学習を進めていく手法である．ReLU やシグモイド関数といった活性化関数と呼ばれる関数を何度も用いることで，モデルの非線形性を増加させ，表現力を高めるという特徴を持つ．

3.2　Convolutional Neural Network

Convolutional Neural Network（CNN）とは，主に機械学習分野における画像処理において活用されるニューラルネットワークの一種であり，日本語では畳み込みニューラルネットワークと呼ばれる．ニューラルネットワークとの違いは隠れ層に，畳み込み層とプーリング層という層が組み込まれているという点である．畳み込み層では，入力された画像に対してフィルタを適用し畳み込むことで画像の特徴マップを算出する．

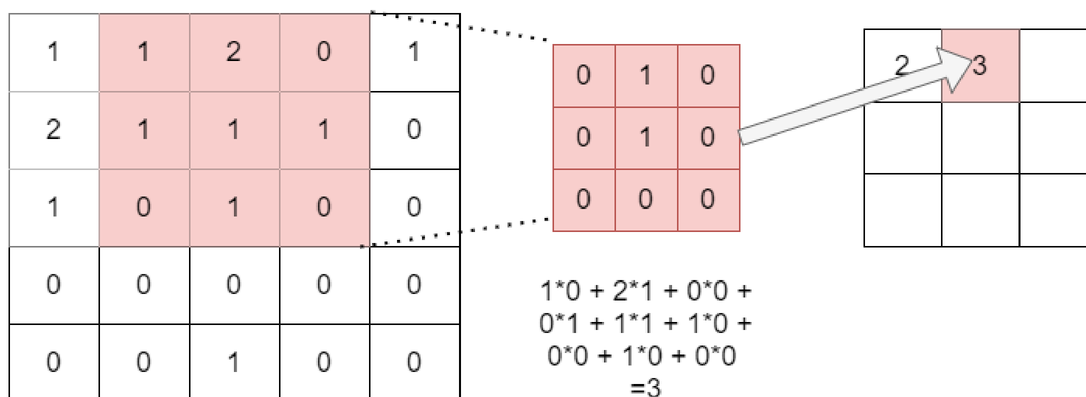

図 3.2: 畳み込み層で行われる処理の例

図 3.2 は畳み込み層で行われる計算を表している．図の例では，(5,5) の入力に対して，(3,3) のフィルタを適応した場合を表しており，入力に対して対応した場所にフィルターを枡掛けしていくと，その合計値が 3 となることがわかる．この処理をフィルタをずらしながら行うことで，畳み込み層では画像の特徴マップを算出している．プーリング層では，この特徴マップの局所空間における代表値を選出することで特徴マップの縮小を行う．プーリング層で行われるプーリングにはいくつか種類があるが，代表的なのは max プーリングであり，局所空間の最大値を代表値とする手法である．Convolutional Neural Network は画像の特徴を抽出することが主な役割であり，フィルタの数値を学習し更新することで出力の精度を向上させることが出来る．

3.3　VGG16

本研究では，エンコーダとなる CNN に VGG16 [8] を使用した．VGG シリーズには VGG16 や VGG19 等が存在し，いずれも 2014 年にオックスフォード大学のチーム Visual Geometry Group が開発した CNN であり，頭文字から VGG と名付けられた．VGG16 は広範囲のイメージに対する豊富な特徴表現を学習した，深さが 16 層の畳み込みニューラルネットワークである．VGG が開発された経緯として，小さいフィルタを何度も通す手法の方が，大きいフィルタを小数回通すよりも計算コストが低く，また活性化関数を通す回数が増えるためより非線形性を増すことができるという考えがあった．この考えの元開発されたのが，(3,3) という小さいサイズのフィルタを何度も通す構造である．図 3.3 に VGG16 の構造を表した図を示す

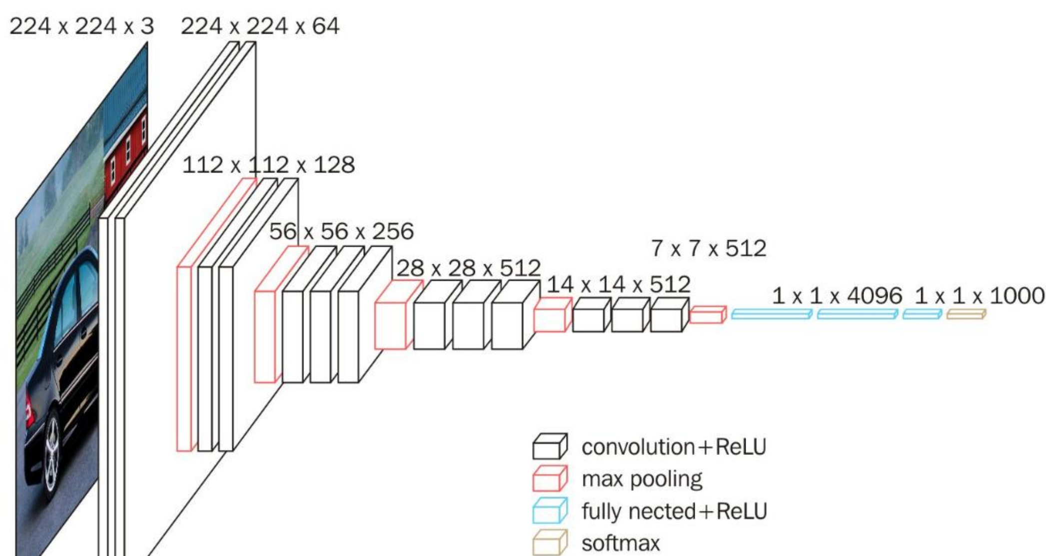

図 3.3: VGG16 の構造 [*1]

具体的な構造としては 3 × 3 のカーネルサイズをもつフィルタを 2 回通す．その後，max プーリングを 2 × 2 をストライド 2 で行うため，画像サイズが半分になる．これをもう一度繰り返したのち，フィルタを通す回数を 3 回に変更し，さらに 3 回繰り返す．最終的に 4096 次元のベクトルを得ることが出来る．畳み込みを行う構造としては非常にシンプルな畳み込みニューラルネットワークである．また，ImageNet(1000 種類のクラスを持つ 1400 枚を超える画像データセット) を使用して学習した重みを利用できるため，実装が非常に容易という特徴がある．実際に，VGG は 2014 年に開催された ILSVRC と呼ばれる ImageNet をデータセット用いて画像の分類精度を競うコンペティションにて，localisation 部門で 1 位を，classification 部門で 2 位を獲得している．本研究では，VGG16 の最終層を使わず，またドロップアウトも無効にすることで学習はせずに，画像を畳み込んだ結果である 4096 次元のベクトルをデコーダへの入力としている．

3.4　Recurrent Neural Network

Recurrent Neural Network（RNN）は，再帰型ニューラルネットワークと呼ばれ，時系列データという時間順序通りに取得されたデータを扱うために用いられる NN の構造である．通常，NN では入力に対しての出力が独立しているため，時系列データのように連続した情報を扱うことが出来ない．しかし，RNN では 1 つ前の入力に対する出力を，

[*1] https://www.freesion.com/article/64761155484/

次の入力として利用する．また，モデルが前の状態を引き継ぐため，前の入力の情報を次に引き継ぐことが出来るのである．

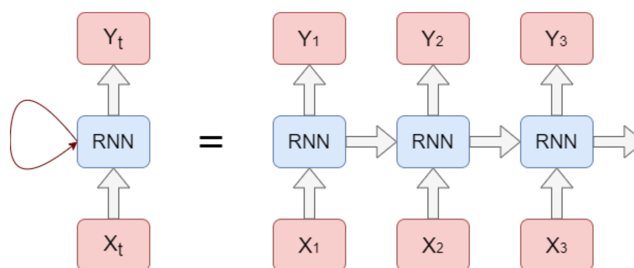

図 3.4: RNN の構造

図 3.4 は RNN の構造を表している．この図のように，前の情報を与えながら同じ処理を行う再帰的な構造が再帰型ニューラルネットワークの特徴である．RNN は上記のような特徴を持つため，自然言語処理においては機械翻訳や文章生成等の言葉の順序に意味があるデータを処理する際に使用される．

3.5　LSTM

LSTM は，Long Short Term Memory の略称であり，Recurrent Neural Network の一種である．Recurrent Neural Network は時系列データを扱うことが出来るニューラルネットワークだが，勾配消失問題と呼ばれる勾配が消失することで学習が進まなくなるという欠点を持っていた．LSTM は 3 つのゲートを導入することでより長期の情報を記憶することが可能であり，より安定して重みの学習が行えるように改良されている．また，より長期の情報を記憶することで文中で離れた単語間での関係を扱えるようになることから，キャプション生成の汎化性能の向上に大きく起因する機構である．本研究ではLSTM をデコーダとして使用しており，エンコーダの出力をそのまま入力とし，出力としてキャプションを生成する．

3.6　MS-COCO

MS-COCO は Microsoft Common Object in Context の略称であり，Microsoft が提供しているアノテーション付きの画像データセットである．[4] 本研究で行ったタスクであるキャプション生成だけではなく，物体検出や画像分類等様々なタスクにおける学習

や研究用に作成されたデータセットである．キャプション生成に利用することができるデータとして，訓練データやバリデーションデータ，テストデータを合わせて 15 万枚以上の画像を使用することが出来る．また，1 画像につき 5 つの英語キャプションが付与されている．上記の特徴から，研究のベンチマークとして使用されることが多いデータセットである．

3.7 STAIR Captions

STAIR Captions は日本語キャプションの生成を行うための, 日本語キャプション付きデータセットである．[5] MS-COCO のテスト画像を除くすべての画像に対して，人力で日本語キャプションを付与したデータセットであり，MS-COCO と同様に 1 画像につき 5 つの日本語キャプションが付与されている．また，既存のキャプション生成手法に単純に置き換えることで日本語キャプションが生成されることが確認できている．本論文では，MS-COCO データセットのキャプション部分を STAIR Captions に置き換えることで日本語キャプションの生成を可能にしている．

3.8 CIDEr

CIDEr は画像や動画のキャプションを評価する際に用いる評価手法である．[9] 既存の評価手法には BLEU や METEOR 等が存在するが，CIDEr はそれらと比べアノテーション数を考慮した計算方法であるため，キャプション生成というタスクにおいては適している評価手法である．CIDEr の計算式は，n-gram 形式を用いた TF-IDF による平均コサイン類似度を表すものであり，通常 1-gram から 4-gram にかけて算出することが多い．

第 4 章

提案手法

画像キャプション生成の問題点は，データセット拡張のコストが高いことと，不適切なキャプション生成の原因が言語側にあるか画像側にあるか不明確であることの 2 点である．これらを解決するため，学習に用いられるデータセットに，キャプションから取り出したキーワードをもとに作成したベクトルを付与する手法を提案する．

具体的な方法について記述する．まず，キーワード候補となる物体名を表す単語を，STAIR Captions データセット全体からすべて抜き出し，出現回数順に整列した．その後，物体検出で出力されないような名詞（色，数字等）は埋め込むキーワードとして不適切であるため，出現回数が多いものから手動で除外した．この作業を，出現回数が多い上位 150 単語が埋め込むキーワードとしてふさわしくなるまで続け，最終的に辞書として保存した．図 4.1 に，キーワードの辞書が作成される流れを示す．

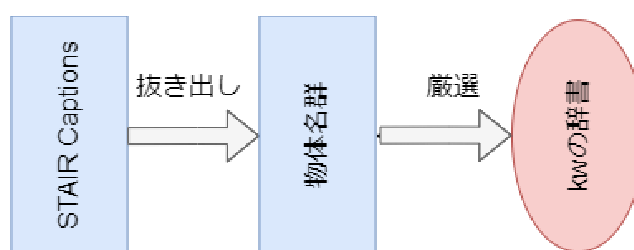

図 4.1: キーワード辞書作成の流れ

次に，実際に付与するベクトルについて説明する．本実験で付与するベクトルとは，キーワードとなる単語 150 個を用いて，学習に用いられるキャプションから作成したキーワードの bag of words である．これを各キャプションごとに作成し，キャプションに対応する画像を畳み込むことで得られるベクトルにアペンドした．図 4.2 に，ベクトルが付

与される流れを示す.

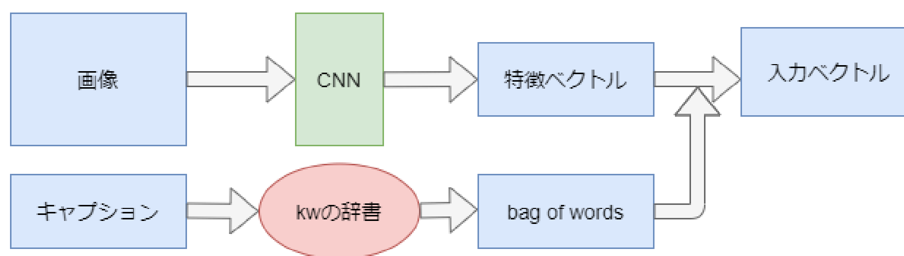

図 4.2: ベクトル付与の流れ

第 5 章

実験

5.1 実験設定

本実験では，MSCOCO の画像データセットと STAIR Captions の日本語キャプションを組み合わせて学習を行った．キーワード付与を行った場合におけるキャプションの品質の変化を調べるために，デフォルトのデータセットとキーワード付与を行ったデータセットの合わせて 2 パターンの訓練データを用意した．これらを用いて学習を行い，デフォルトのデータセットを用いて学習を行ったモデルと，提案手法を用いて作成されたデータセットを用いて生成されたモデルとを比べることによって品質が向上したかを調査する．

今回の実験では CNN は学習させずに，学習対象をエンコーダの最後に行う 1 層の全結合層と，デコーダとなる LSTM のみを学習させた．また，本実験では CNN に学習済みネットワークである VGG16 [8] を用いて画像畳み込みを行ったため，エンコーダで学習させたのは全結合層のみである．

学習に用いたデータセットは，MSCOCO の訓練データ画像と，STAIR Captions の 5 つのキャプションのセットを組み合わせた 82732 個である．図 5.1 に訓練データ例を示す．

stair caption
長い 机 と 椅子 が ， 並ん で 置い て ある
テーブル の 上 に ， 植物 が 置い て ある
背もたれ が 丸い 椅子 が たくさん 並べ られ て いるテーブル の 真ん中 に 大きな ガラス の 花瓶 に活け られ た 花 が 飾ら れ て いる
長い テーブル に お 揃い の イス が 置か れ て いる
10 人 掛け の テーブル の 上 に は 分厚い 本 や フラワー アレンジメント が 置か れ て いる

図 5.1: 訓練データ例

これに対してバッチサイズ 100，エポック数 10 として学習を行った．そして，学習する際の各エポック終了時にモデルを保存し，各モデルに対してテストデータとなる MSCOCO のデータセットから抜き出した画像 10000 枚からキャプションを生成させ評価を行った．

実験後，キーワードとして埋め込んだ単語がどれほどの出現率となるかを調査するために，提案手法を用いたモデルで最高精度だったモデルと提案手法を用いないモデルで最高精度だったモデルを用いて生成されたキャプション 10000 文に対して，ベクトルを

付与する際に用いたキーワードの bag of words を作成し分析した.

分析内容は以下の 3 通りである.

- キーワードとして付与した単語の出現率.
- bag of words の値で場合分けした，キーワードとして付与した単語の出現率.
- 不適切なキーワードの出現率.

これらについて調査を行い，分析を行った.

5.2 モデル

本実験で用いるモデルはエンコーダデコーダモデルである．エンコーダデコーダモデルとはエンコーダ層とデコーダ層からなるモデルであり，前半のエンコーダ層で入力された情報のエンコーディングをおこなう．次に，エンコーダ層で変形した情報を入力として，後半のデコーダ層で入力された情報のデコーディングをするというのが大まかな流れである.

エンコーダ層には，畳み込みニューラルネットワークの一種である VGG16 と 1 層の全結合層を用いた．入力された画像を VGG16 を用いて畳み込み，画像の特徴量である 4096 次元のベクトルを得る．次に，その出力を 1 層の全結合層に通す．ここまでがエンコーダ層の役割であり，画像をデコーダ層が学習しやすいベクトルへと変換する.

デコーダ層には，LSTM を用いた．LSTM の最初の入力はエンコーダ層を用いて算出したベクトルと文の始まりを表す単語である SOS の埋め込みベクトルである．LSTM の出力は 1 層の全結合層へと入力され，最終的にデコーダ層は単語の埋め込みベクトルを出力する．この出力された埋め込みベクトルを LSTM の入力として，文章の終わりを意味する EOS の埋め込みベクトルが出るまで単語の埋め込みベクトルを得ることで文章を生成することが出来る.

つまりキャプション生成の流れは，以下の順となる.

1. 画像（エンコーダへの入力）
2. VGG16
3. 全結合層 (エンコーダの出力)
4. LSTM (デコーダへの入力)

5. 全結合層 (デコーダの出力)
6. 文生成終了までデコーダをまわす

図 5.2 に，モデルが画像キャプションを生成する流れを示す．

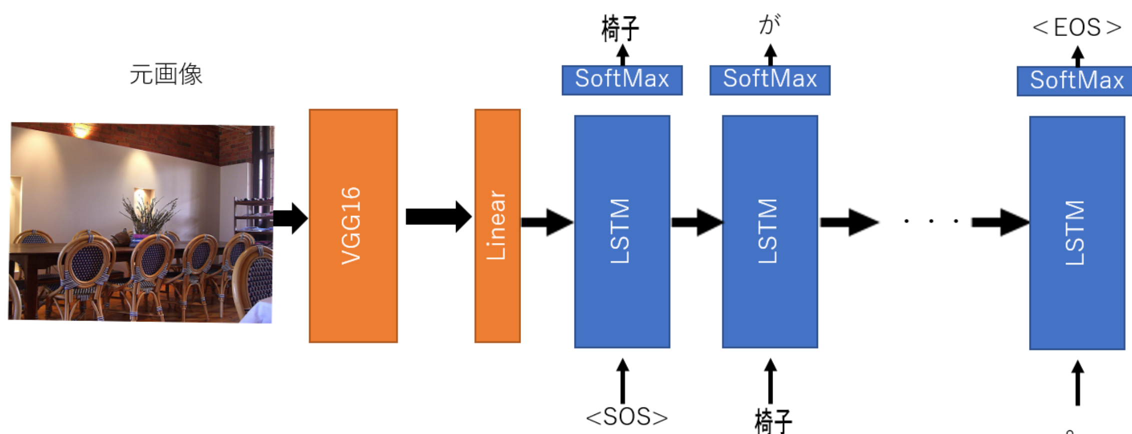

図 5.2: 画像キャプション生成の流れ

また，提案手法を適用する場所は VGG16 の出力である．画像の特徴量である 4096 次元のベクトルに，キーワードの bag of words である 150 次元のベクトルをアペンドする．ほかの変更点はないため，提案手法を適用したモデルのキャプション生成の流れは，以下の順となる．

1. 画像（エンコーダへの入力）
2. VGG16
3. キーワードの bag of words をアペンド
4. 全結合層 (エンコーダの出力)
5. LSTM (デコーダへの入力)
6. 全結合層 (デコーダの出力)
7. 文生成終了までデコーダをまわす

図 5.3 に，提案手法を適用したモデルが画像キャプションを生成する流れを示す．

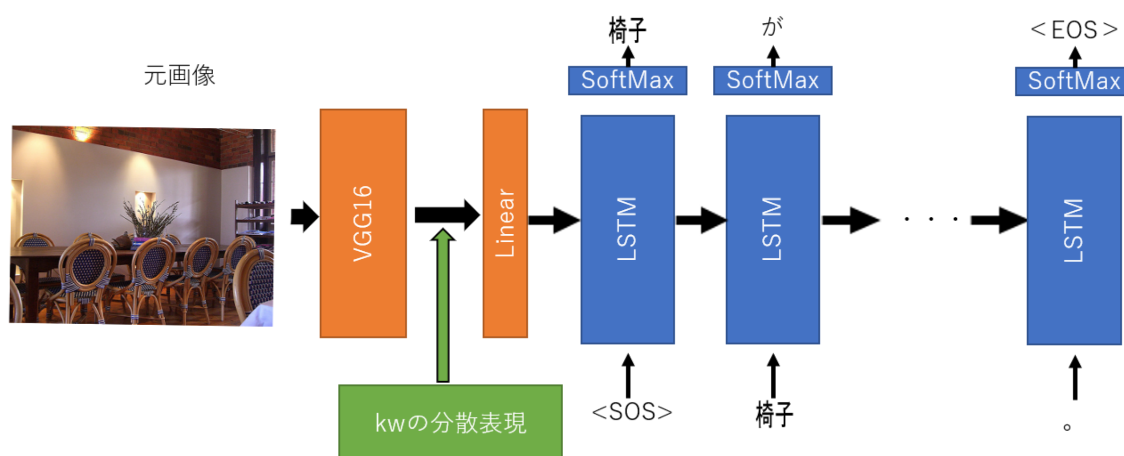

図 5.3: 提案手法を適用した，画像キャプション生成の流れ

5.3　評価手法

生成キャプションの評価には CIDEr [9] を用いた．

CIDEr では，s_{ij} を i 番目の画像に対する j 番目のアノテーション文としたときに，ω_k（各 n-gram）に対して付与する重みを算出する重み関数 $g_k(s_{ij})$ を以下のように定義している．

$$g_k(s_{ij}) = \frac{h_k(s_{ij})}{\sum_{\omega_l \in \Omega} h_l(s_{ij})} \log \left(\frac{|I|}{\sum_{I_p \in I} min(1, \sum_q h_k(s_{pq}))} \right) \tag{5.1}$$

ここで，$h_k(s_{ij})$ は s_{ij} に出てきた ω_k の個数であり，$\sum_{\omega_l \in \Omega} h_l(s_{ij})$ はその個数を同文中の全 n-gram に対して行っているため，つまり $\frac{h_k(s_{ij})}{\sum_{\omega_l \in \Omega} h_l(s_{ij})}$ は TF(Term frequency) である．また，$|I|$ はデータセットの画像の総数であり，$min(1, \sum_q h_k(s_{pq}))$ で，アノテーション文に ω_k が出てきた回数を画像ごとに計算し，1 との最小値をとっている．それを画像全体に対して行っているため，つまり $\log \left(\frac{|I|}{\sum_{I_p \in I} min(1, \sum_q h_k(s_{pq}))} \right)$ は画像全体に ω_k がどれほど出現するかを表しており，これは IDF（Inverse document frequency）である．つまり，重み関数 $g_k(s_{ij})$ は画像に対する複数キャプションに対応した TF-IDF である．

この重み関数を，i 番目の画像のアノテーション文すべてに用いて得られたベクトルの集合を $g^n(S_i)$，i 番目の画像の生成文に用いて得られたベクトルを $g^n(c_i)$ として，両ベクトル間の平均コサイン類似度を算出する関数 $CIDEr_n(c_i, S_i)$ を以下のように定義する．

$$CIDEr_n(c_i, S_i) = \frac{1}{m} \sum_j \frac{g^n(c_i) \cdot g^n(s_{ij})}{\|g^n(c_i)\| \|g^n(s_{ij})\|} \tag{5.2}$$

ここで，m はアノテーション文の数であり，アノテーション文の数を考慮する形となっている．

そして，定義した $CIDEr_n$ を用いて，i 番目の画像における最終的な CIDEr のスコアを以下のように定義する．

$$CIDEr(c_i, S_i) = \sum_{n=1}^{N} w_n CIDEr_n(c_i, S_i) \tag{5.3}$$

この式はつまり，$CIDEr_n(c_i, S_i)$ について，n-gram の n を 1 から N まで変えてその合計を算出し，定数 w_n との積をとったものである．原著論文では w_n は $\frac{1}{N}$ が，N は 4 がそれぞれ最適とされていたため，本研究ではその通りに実装した．

以上のことから，CIDEr は画像に対するアノテーション内で多く使われる n-gram に対しては重みを高くし，画像全体のアノテーションに頻繁に登場する n-gram に対しては重みを低くするという特徴があることがわかる．

CIDEr の値は，つまり平均コサイン類似度であるので，値が高ければ高いほど生成キャプションが高品質である．

5.4　実験結果

実験結果として，キーワード付与前と付与後のモデルで行ったテスト結果をまとめたものを図 5.4 に示す．

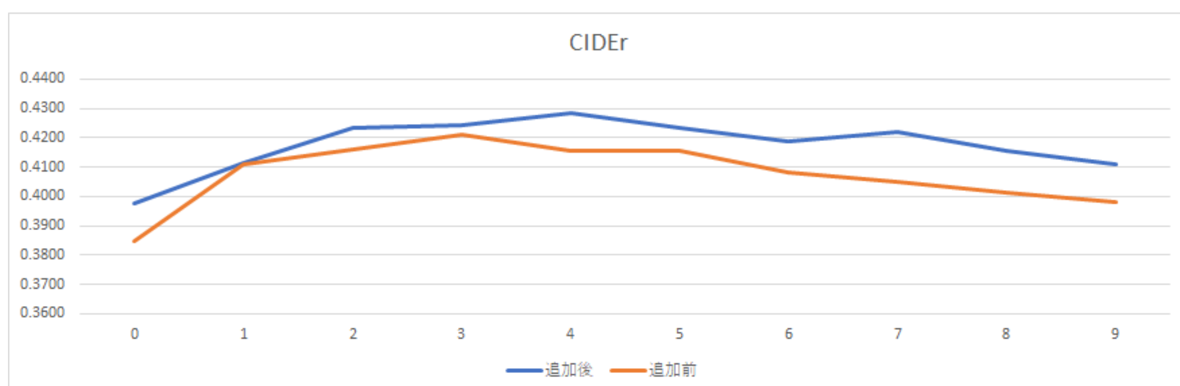

図 5.4: CIDEr の評価

キーワードの付与前のモデルで最高精度となったのはエポック 3 のものであり，値は

42.1 であった．キーワードの付与後のモデルで最高精度となったのはエポック 4 のものであり，値は 42.8 であった．この結果により，CIDEr という評価手法では提案手法がベースラインを上回る最高精度を記録し，提案手法が精度向上に起因することが確認できた．

キーワードとして付与した単語の出現率についての分析結果をまとめたものを表 5.1 に示す．

表 5.1: 埋め込んだキーワード数（平均）に対する，出力されたキーワード数（平均）

kw 数	出力できた kw 数（追加前）	出力できた kw 数（追加後）
3.62	1.04	1.06

この結果から，キーワードとして付与した単語の出現率は，上昇したが大きな変化がなかったことが確認できた．

キーワードを付与する際に埋め込まれるベクトルは，正解キャプション 5 つから作成したキーワードの bag of words であるので，各キーワードごとに出現回数によって重みが違うという問題がある．表 5.1 はその問題点を考慮していないため，画像ごとではなく単語単位で，正解キャプション 5 つにおけるキーワードの出現回数によって場合分けし，分析した．その結果を表 5.2 に示す．

表 5.2: 正解文内の各キーワードの出現回数（bag of words の数値）をもとにした，キーワードの出力率

正解文の各 kw 出現回数	追加前	追加後	単語総数
1 回	12.01 %	11.31 %	15564
2 回	24.68 %	24.35 %	6941
3 回	37.21 %	39.18 %	4934
4 回	49.51 %	52.20 %	4272
5 回	61.58 %	64.88 %	3964
6 回	69.30 %	73.24 %	355
7 回	75.00 %	77.00 %	100
8 回	80.00 %	86.67 %	30
9 回	73.33 %	66.67 %	15
10 回	100.00 %	75.00 %	4

間違ったキーワードが出現した確率も同時に調査した．具体的には，追加したベクトルに含まれていないキーワードが出現する確率を計算した．計算結果は追加前が 52.75 %，追加後が 47.74 %であった．

5.5　出力例

図 5.5 にモデル別のキャプション出力例を示す．

「パソコン」，「サーフ」はキーワードであり，ベクトルに埋め込まれているため，提案手法を用いることで生成されるキャプションがより正確になっている場合があることが確認できる．

	生成キャプション
前	男性 が ベッド の 上 で 本 を 読ん で いる
後	男性 が パソコン を 見 て いる

	生成キャプション
前	砂浜 で 男性 が フリスビー を し て いる
後	砂浜 で サーフ ボード を 持っ て いる 人 が いる

図 5.5: テスト画像と生成キャプション例

第 6 章

考察

実験結果より，本実験の提案手法は CIDEr の評価値こそ微増させたものの，埋め込んだキーワードをキャプションに出現させるという観点においてはあまり効果がないということが分かった．特に，正解キャプション内で出現回数が少ないキーワードに関してはほとんど改善が見られなかった．埋め込んだキーワードをよりキャプションに出現させることができなかった要因の一つに，画像に様々な物体が写っているという要因が考えられる．提案手法を追加する前から，生成キャプション 1 つにつき平均 1 つのキーワードを出現させることが出来ている．よってこの提案手法によってキャプションの精度を向上させるには，すでに生成キャプションに含まれている物体に加えて他の物体も表現しなければならないのである．

埋め込んだキーワード数の平均から，多くの画像には 3 つ以上の物体が写っていることが確認でき，これらは提案手法によって，学習に用いるベクトルへと確実に付与されている．故に，正確に物体検出出来ている状況に近い状態にあると言える．しかし，それでもあまり改善が見られないことから，本実験によって，単純に画像の物体すべてを物体検出することが出来ても，それらを複数用いて正しく画像キャプション生成をすることは困難であるということが考えられる．

また，提案手法は表記ゆれを吸収できないという欠点がある．例えば，「男性」，「男」，「青年」などといった意味上は似通った単語については，本研究では特に操作せずに別の単語として扱っている．加えて，「男」や「青年」等の単語は埋め込むキーワードにバリエーションを持たせるために除去した．そのため，意味上は似通った単語であっても追加するベクトルには反映されない．これが影響し，埋め込んだキーワードをキャプションに出現させることが出来なかった可能性が考えられる．

しかし，提案手法が全く機能していなかったというわけではない．表 5.2 から，正解キャプション内で出現回数が 5 回の単語に関しては，出現確率が約 3.31 %，4 回の単語に関しては，出現確率が 2.69 %ほど上昇している．この結果から，出現回数が多い，つまり重要度の高いキーワードの出現確率は向上していることが確認出来る．逆に，出現回数が 1 回や 2 回だけの，出現回数が少ないキーワードに関しては出現確率が下降している．これらは単純に bag of words の数値に起因するものであると考察できる．bag of words が高いキーワードのみが優先されてしまい，低いキーワードの生成に至らなかったと考えられる．

また，誤ったキーワードを含めたキャプションが 52.75 %から 47.74 %へと減少したことから，図 5.5 の例のように，当初の目的である物体検出の安定，キャプションの正確性を増すという点では提案手法は効果的であるということが確認できた．

第 7 章

結論

本論文では，キーワード付与を行うことで画像キャプション生成における品質の向上を行った．具体的には，画像を畳み込んだベクトルに，厳選したキーワードを用いた正解キャプションの bag of words をアペンドした．提案手法を用いて生成したキャプションは評価値については微増することが確認できた．しかし，複数のキーワードの生成という点においてあまり大きな効果がないことが実験結果の分析からわかった．

本実験を行っていくうえで Chive のような日本語単語分散表現を追加するベクトルに使ってみるというアイデアは出ていた．本実験ではよりシンプルになる用に bag of words を採用したが，これらのベクトルを利用して実験を行うことも検討する．また，本提案手法の弱点として述べた表記ゆれについての対抗策も検討する予定である．

キーワードを付与することで物体検出は出来ていると考えられることから，問題点は言語処理側にあるといえるはずである．今後は言語処理側に注視し，追加するベクトルについて改良に関する研究を重ねてく．

謝辞

本研究を進めるにあたり，指導教員である新納浩幸教授には，研究に関する適切な助言や指導をいただきました．心から感謝申し上げます．また新納研究室の皆様には，研究のみならず本論文の作成に関してまでもご教授いただきました．お礼申し上げます．ありがとうございました．

参考文献

[1] Yang Feng, Lin Ma, Wei Liu, and Jiebo Luo. Unsupervised image captioning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[2] Ukyo Honda, Yoshitaka Ushiku, Atsushi Hashimoto, Taro Watanabe, and Yuji Matsumoto. Removing word-level spurious alignment between images and pseudo-captions in unsupervised image captioning. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pp. 3692–3702, Online, April 2021. Association for Computational Linguistics.

[3] 岩村紀与彦, ルイ笠原純ユネス, モロアレッサンドロ, 山下淳, 淺間一. アテンション機構を用いたクロップとマスクによるキャプション生成のためのデータ拡張. 精密工学会誌, Vol. 86, No. 11, pp. 904–910, 2020.

[4] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pp. 740–755, Cham, 2014. Springer International Publishing.

[5] Stair captions: Constructing a large-scale japanese image caption dataset. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 417–421, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[6] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE Conference*

on Computer Vision and Pattern Recognition (CVPR), June 2015.

[7] 小林豊, 鈴木諒, 谷津元樹, 原田実. 深層学習による日本語キャプション生成システムの開発. 人工知能学会第二種研究会資料, Vol. 2017, No. AM-17, p. 04, 2017.

[8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[9] Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

付録

A　プログラムリスト

モデルの構築及び学習を行うソースコードを A.1 に示す．尚，本来はモデルを構築する際に提案手法を適用した場合としていない場合の二種類のソースコードが存在するが，変更点はわずかであるため，提案手法を適用していないソースコードに関しては省略する．

ソースコード A.1: キャプション生成モデル構築および学習

```
1  import torch.nn.functional as F
2  import torch
3  import random
4  import matplotlib.pyplot as plt
5  import numpy as np
6  import matplotlib
7  import json
8  from torchvision import models, transforms
9  from torch.autograd import Variable
10 from torch import nn
11 from PIL import Image
12 from nltk.translate.bleu_score import sentence_bleu
13 from nltk import word_tokenize
14 from IPython import display
15 from gensim.models import Word2Vec
16 from collections import Counter, defaultdict
17 import nltk
18 from torch._C import dtype
19 nltk.download('punkt')
20
21
22 # 使う cuda
23 use_cuda = True
```

```
24
25
26  # データをロード
27
28
29  vocabularySize = 10000
30
31
32
33  # トレーニング画像のアノテーションファイルを読み込む.
34  mscoco_train = json.load(open('stair_captions_v1.2_train_tokenized.
        json'))
35  train_ids = [entry['id']
36               for entry in mscoco_train['images']][:100000]
37  train_id_to_file = {entry['id']: 'train2014/' +
38                      entry['file_name'] for entry in mscoco_train['
                            images']}
39
40
41  # トレーニング画像のキャプションを抽出する
42  train_id_set = set(train_ids)
43  train_id_to_captions = defaultdict(list)
44  for entry in mscoco_train['annotations']:
45      if entry['image_id'] in train_id_set:
46          train_id_to_captions[entry['image_id']].append(
47              entry['tokenized_caption'])
48
49
50  # 検証画像のアノテーションファイルを読み込む
51  mscoco_val = json.load(open('stair_captions_v1.2_val_tokenized.json')
        )
52  val_ids = [entry['id'] for entry in mscoco_val['images'
        ]][20000:30000]
53  val_id_to_file = {entry['id']: 'val2014/' + entry['file_name']
54                    for entry in mscoco_val['images']}
55
56
57  # 検証画像のキャプションを抽出する
58  val_id_set = set(val_ids)
59  val_id_to_captions = defaultdict(list)
60  for entry in mscoco_val['annotations']:
```

```
61          if entry['image_id'] in val_id_set:
62              val_id_to_captions[entry['image_id']].append(
63                  entry['tokenized_caption'])
64
65
66  # テスト画像用のアノテーションファイルの読み込み
67  mscoco_test = json.load(open('stair_captions_v1.2_val_tokenized.json'
        ))
68  test_ids = [entry['id'] for entry in mscoco_test['images']][:10000]
69  test_id_to_file = {entry['id']: 'val2014/' + entry['file_name']
70                     for entry in mscoco_test['images']}
71
72
73
74
75  def preprocess_numberize(sentence):
76      """文字列形式の文が与えられると，この関数はそれを数字のリスト（語彙
            のインデックス）に前処理する
77
78
79
80      """
81      tokenized = word_tokenize(sentence.lower())
82
83
84      tokenized = ["<SOS>"] + tokenized + ["<EOS>"]
85      numberized = [word2index.get(word, 0) for word in tokenized]
86
87      return numberized
88
89
90  def preprocess_one_hot(sentence):
91      """文字列形式の文が与えられると，この関数はワンホットベクトルの配列
            に前処理．
92
93      numpy
94      """
95      numberized = preprocess_numberize(sentence)
96
97      # 各単語をワンホットで表現する
98      one_hot_embedded = one_hot_embeddings[numberized]
99
```

```
100         return one_hot_embedded
101 
102 
103 
104     # 画像を適切にスケーリング
105     img_size = 224
106     loader = transforms.Compose([
107         transforms.Resize(img_size),
108         transforms.CenterCrop(img_size),
109         transforms.ToTensor(),
110     ])
111 
112 
113     def load_image(filename, volatile=False):
114         """画像を読み込んで前処理を行う
115 
116 
117         1. 画像を開く.
118         2. 画像を拡大・縮小し，float に変換. tensor
119         3. 変数に変換する (のモデルへの入力はすべて変数 PyTorch).
120         4. テンソルの先頭に別の次元を追加.
121         5. 5. 変数をに移動させる. GPU
122         """
123         image = Image.open(filename).convert('RGB')
124         image_tensor = loader(image).float()
125         image_var = Variable(image_tensor, volatile=volatile).unsqueeze
                (0)
126         return image_var.cuda()
127 
128 
129     display.display(Image.open(train_id_to_file[train_ids[0]]))
130 
131     # 最初のキャプションを確認
132     for caption in train_id_to_captions[train_ids[0]]:
133         print(caption)
134 
135 
136     # Image Encoder エンコードして画像を数値化する
137 
138     vgg_model = models.vgg16(pretrained=True).cuda()#の使い方
            vgghttps://note.nkmk.me/python-pytorch-pretrained-models-image-
```

```
        classification/
139
140   vgg_model.eval()
141
142   # 分類器の最終層を削除し，モデルがトレーニングではなく推論に使われてい
        ることをに示す．PyTorch
143   # トレーニングじゃない（ドロップアウトを無効にする）．
144   modified_classifier = nn.Sequential(
145       *list(vgg_model.classifier.children())[:-1])
146   modified_classifier.eval()
147
148   # 修正した分類器をモデルに再割り当て VGG
149   vgg_model.classifier = modified_classifier
150
151
152   # モデルをプリントアウトして，現在の状態を確認する
153   # print(vgg_model)
154   print('モデル構築終了')
155
156   """データの処理次に，それぞれの画像をベクトル表現に変換．
157
158
159
160   """
161
162   training_vectors = []
163
164   for i,image_id in enumerate(train_ids):
165       # 画像の読み込み前処理/
166       img = load_image(train_id_to_file[image_id])
167
168       # 畳み込み層を実行して，出力をリサイズ
169       output = vgg_model(img)
170
171       training_vectors.append(np.array(list(output.data.squeeze()),
            dtype = 'float32'))
172
173
174   # の配列に変換して，結果をファイルに保存．numpy
175   training_vectors = np.stack(training_vectors, axis=0)
176   np.save(open('outputs/training_vectors', 'wb+'), training_vectors)
```

```
177 print('保存完了 training_vectors')
178
179 #次に，すべてのをベクトル化し，その結果をファイルに書き込みます．
        validation
180 print('val_vecotor␣開始')
181
182 validation_vectors = []
183
184 for i,image_id in enumerate(val_ids):
185     # 画像の読み込み前処理を行います．/
186     img = load_image(val_id_to_file[image_id])
187
188     # 畳み込み層を実行して，出力をリサイズ．
189     output = vgg_model(img)
190
191     validation_vectors.append(np.array(list(output.data.squeeze()),
            dtype = 'float32'))
192 print('for␣fin')
193
194 # これをの配列に変換して，結果をファイルに保存．numpy
195 validation_vectors = np.stack(validation_vectors, axis=0)
196 np.save(open('outputs/validation_vectors', 'wb+'), validation_vectors
        )
197 print('保存完了 training_vectors')
198
199
200 test_vectors = []
201
202 for image_id in test_ids:
203
204     img = load_image(test_id_to_file[image_id])
205     features_output = vgg_model.features(img)
206     classifier_input = features_output.view(1, -1)
207     output = modified_classifier(classifier_input)
208     test_vectors.append(list(output.data.squeeze()))
209
210 test_vectors = np.array(test_vectors,dtype = 'float32')
211 np.save(open('outputs/test_vectors', 'wb+'), test_vectors)
212
213
214 # ベクトルを読み込む
```

```
215   training_vectors = np.load(open('append_vec/
          mk2_append_ed_vector_onehot_men_gensen_150', 'rb'))
216
217   #val train  各データのどれを使うかしっかりと確
          認！！！！！！！！！！！！！！！！ test
218
219   #validation_vectors = np.load(open('outputs/validation_vectors', 'rb
          '))
220   #training_vectors = np.load(open('outputs/training_vectors92783', 'rb
          '))
221   test_vectors = np.load(open('outputs/test_vectors', 'rb'))
222
223
224   print(training_vectors.shape)
225   #print(validation_vectors.shape)
226   print(test_vectors.shape)
227   print('vector␣is␣ok')
228
229
230   """ここからボキャブラリーの辞書の定義  辞書に登録する単語がいくつかに気
          を付ける！！！！！
231
232
233   """
234
235   sentences = [sentence for caption_set in train_id_to_captions.values
          ()
236                for sentence in caption_set]
237
238   # 文章を小文字にし，それらをトークン化し，<SOSと><EOSのトークンを追加す
          る>
239   sentences = [
240       ["<SOS>"] + word_tokenize(sentence.lower()) + ["<EOS>"] for
              sentence in sentences]
241
242   # ボキャブラリーを作成します．ボキャブラリーにない単語を表すために，
          <UNKというトークンを追加していることに注意>!
243
244   word_counts = Counter([word for sentence in sentences for word in
          sentence])
245   vocabulary = ["<UNK>"] + [e[0]
246                             for e in word_counts.most_common(
```

```
                                  vocabularySize-1)]
247 word2index = {word: index for index, word in enumerate(vocabulary)}
248 one_hot_embeddings = np.eye(vocabularySize)
249
250 # 学習データの中で最も長い文章を最大シーケンス長と定義する.
251 maxSequenceLength = max([len(sentence) for sentence in sentences])
252
253 print('dic␣is␣ok')
254
255 train_id_to_vector = {} # からエンコードされた画像の数値を呼び出せるよ
        うにするよ id
256 # ここのの中の値は絶対に間違えないように train_ids
257 for i, train_id in enumerate(train_ids[:100000]):
258     train_id_to_vector[train_id] = training_vectors[i]
259
260 print('train_id_to_vector␣is␣ok')
261
262 class ImageEncoder(nn.Module):
263     def __init__(self, input_size, hidden_size):
264         super(ImageEncoder, self).__init__()
265         self.out = nn.Linear(input_size, hidden_size)
266
267     def forward(self, inputs):
268         return self.out(inputs)
269 # にはを使うよ decoderLSTM
270
271
272 class DecoderLSTM(nn.Module):
273     def __init__(self, input_size, hidden_size, output_size):
274         super(DecoderLSTM, self).__init__()
275         self.hidden_size = hidden_size
276
277         self.lstm = nn.LSTM(input_size, hidden_size)
278         self.out = nn.Linear(hidden_size, output_size)
279
280     def forward(self, input, hidden):
281         output = F.relu(input)
282         output, hidden = self.lstm(output, hidden)
283         output = self.out(output)
284         output = F.log_softmax(output.squeeze())
285         return output.unsqueeze(0), hidden
```

```
286
287
288 encoder = ImageEncoder(input_size=4246, hidden_size=500).cuda()
289 decoder = DecoderLSTM(input_size=len(vocabulary),
290                       hidden_size=500, output_size=len(vocabulary)).
                              cuda()
291 print('encoder', encoder)
292 print('decoder', decoder)
293
294
295
296 """モデルの学習損失関数の定義
297
298
299
300 # https://gist.github.com/jihunchoi/f1434a77df9db1bb337417854b398df1
301 """
302
303
304 def _sequence_mask(sequence_length, max_len=None):
305     if max_len is None:
306         max_len = sequence_length.data.max()
307     batch_size = sequence_length.size(0)
308     seq_range = torch.arange(0, max_len).long()
309     seq_range_expand = seq_range.unsqueeze(0).expand(batch_size,
            max_len)
310     seq_range_expand = Variable(seq_range_expand)
311     if sequence_length.is_cuda:
312         seq_range_expand = seq_range_expand.cuda()
313     seq_length_expand = (sequence_length.unsqueeze(1)
314                          .expand_as(seq_range_expand))
315     return seq_range_expand < seq_length_expand
316
317
318 def compute_loss(logits, target, length):
319     """
320
321
322     logits: 各クラスの正規化されていない確率を含む，サイズ
            (batch, max_len, num_classes) のを含む変数．FloatTensor
323     target: size (batch, max_len) のLongTensor を含む変数で，対応する
```

```
            各ステップの真のクラスのインデックスをふくむ．
324     length: バッチ内の各データの長さを含む，サイズ（バッチ,）のを含む変
            数．LongTensor戻り値損失．長さによってマスクされた平均損失値．
325
326
327
328
329
330     """
331     # logits_flat: (batch * max_len, num_classes)
332     logits_flat = logits.view(-1, logits.size(-1))
333     # log_probs_flat: (batch * max_len, num_classes)
334     log_probs_flat = logits_flat
335     # target_flat: (batch * max_len, 1)
336     target_flat = target.view(-1, 1)
337     # losses_flat: (batch * max_len, 1)
338     losses_flat = -torch.gather(log_probs_flat, dim=1, index=
            target_flat)
339     # losses: (batch, max_len)
340     losses = losses_flat.view(*target.size())
341     # mask: (batch, max_len)
342     mask = _sequence_mask(sequence_length=length, max_len=target.
            size(1))
343     losses = losses * mask.float()
344     loss = losses.sum() / length.float().sum()
345     return loss
346
347
348 """学習ゾーン
349
350
351 """
352
353
354 def train(input_variables,
355           embed_caption,
356           target_caption,
357           input_lens,
358           encoder,
359           decoder,
360           encoder_optimizer,
```

```
361             decoder_optimizer,
362             criterion,
363             embeddings=one_hot_embeddings):
364         encoder_optimizer.zero_grad()
365         decoder_optimizer.zero_grad()
366 
367         # エンコーダに画像を通す
368         encoder_output = encoder(input_variables).unsqueeze(0)
369 
370         # デコーダの入力を構築する(初期状態ではバッチごとに<SOSとなってい
                る>
371         decoder_input = Variable(torch.FloatTensor([[embeddings[
                word2index["<SOS>"]]
372                                                   for i in range(
                                                          embed_caption.
                                                          size(1))]]))
373         decoder_input = decoder_input.cuda() if use_cuda else
                decoder_input
374 
375         # デコーダの初期隠れ状態をエンコーダの出力に設定する
376         decoder_hidden = (encoder_output, encoder_output)
377 
378         # 結果テンソルの準備
379         all_decoder_outputs = Variable(torch.zeros(*embed_caption.size()
                ))
380         if use_cuda:
381             all_decoder_outputs = all_decoder_outputs.cuda()
382 
383         all_decoder_outputs[0] = decoder_input
384 
385         # 最初のインデックスの後のインデックスを反復.
386         for t in range(1, embed_caption.size(0)):
387             decoder_output, decoder_hidden = decoder(decoder_input,
                    decoder_hidden)
388 
389             if random.random() <= 0.8:
390                 decoder_input = embed_caption[t].unsqueeze(0)
391             else:
392                 topv, topi = decoder_output.data.topk(1)
393 
394                 # インプットの準備
```

```
395                 decoder_input = torch.stack([Variable(torch.FloatTensor(
                        embeddings[ni])).cuda()
396                                                 for ni in topi.squeeze()]).
                                                        unsqueeze(0)
397
398             # デコーダ出力の保存
399             all_decoder_outputs[t] = decoder_output
400
401         loss = compute_loss(all_decoder_outputs.transpose(0, 1).
                contiguous(),
402                             target_caption.transpose(0, 1).contiguous(),
403                             Variable(torch.LongTensor(input_lens)).cuda()
                                    )
404
405         loss.backward()
406
407         torch.nn.utils.clip_grad_norm(encoder.parameters(), 10.0)
408         torch.nn.utils.clip_grad_norm(decoder.parameters(), 10.0)
409
410         encoder_optimizer.step()
411         decoder_optimizer.step()
412
413         return loss.item()
414
415
416     """モデルのトレーニング
417
418
419     """
420
421
422     def pad_seq(arr, length, pad_token):
423         """トークンを使って配列をある長さまでパッド
424
425
426         """
427         if len(arr) == length:
428             return np.array(arr)
429
430         return np.concatenate((arr, [pad_token]*(length - len(arr))))
431
```

```
432
433  """個々の関数　　がに当たる関数
434  caption_imagetest
435  """
436
437
438  def caption_image(image_vector, embeddings=one_hot_embeddings,
         max_length=20):
439      """画像のベクトルが与えられたら，その画像にキャプションを付けてる．
440
441      """
442
443      input_variable = Variable(torch.FloatTensor([image_vector])).
             cuda()
444      encoder_output = encoder(input_variable).unsqueeze(0)
445
446      decoder_input = Variable(torch.FloatTensor(
447          [[embeddings[word2index["<SOS>"]]]])).cuda()
448
449      decoder_hidden = (encoder_output, encoder_output)
450
451      decoder_outputs = []
452      for t in range(1, max_length):
453          decoder_output, decoder_hidden = decoder(decoder_input,
                 decoder_hidden)
454
455          topv, topi = decoder_output.data.topk(1)
456          ni = topi[0][0]
457          decoder_outputs.append(ni)
458
459          if vocabulary[ni] == "<EOS>":
460              break
461
462          decoder_input = Variable(torch.FloatTensor([[embeddings[ni
                 ]]])).cuda()
463          decoder_input = decoder_input.cuda() if use_cuda else
                 decoder_input
464
465      return '␣'.join(vocabulary[i] for i in decoder_outputs)
466
467
```

```
468  # 損失関数と最適化手法の定義
469  encoder_optimizer = torch.optim.Adam(decoder.parameters(), lr
         =0.001)
470  decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr
         =0.001)
471  criterion = nn.CrossEntropyLoss()
472
473  # トレーニングデータの作成
474  train_data = [(train_id, caption) for train_id, captions in
         train_id_to_captions.items()
475                for caption in captions if len(caption) > 0]
476
477  # ここで順番をぐちゃぐちゃに
478  random.shuffle(train_data)
479
480  num_epochs = 10
481  batch_size = 100
482  print(len(train_data)//batch_size)
483  for _ in range(num_epochs):
484
485      for i in range(len(train_data)//batch_size):
486          # バッチのデータを取得
487          batch = train_data[i*batch_size:(i+1)*batch_size]
488
489          # すべての画像のベクトルを取得
490          input_variable = np.stack(
491              [train_id_to_vector[train_id] for train_id, _ in batch])
492          input_variable = Variable(torch.FloatTensor(input_variable))
                 .cuda()
493
494          # 文章の取得
495          sentences = [sentence for _, sentence in batch]
496
497          # 文章の長さの取得
498          sentence_lens = [len(preprocess_numberize(sentence))
499                           for sentence in sentences]
500
501          # すべてのパッドの長さを決定する統一できるように
502          max_len = max(sentence_lens)
503
504          # 各バッチのすべての文を前処理する
```

```
505             one_hot_embedded_list = [preprocess_one_hot(
506                 sentence) for sentence in sentences]
507             one_hot_embedded_list_padded = [pad_seq(embed, max_len, np.
                    zeros(len(vocabulary)))
508                                                for embed in
                                                       one_hot_embedded_list]
509
510             numberized_list = [preprocess_numberize(
511                 sentence) for sentence in sentences]
512             numberized_list_padded = [pad_seq(numb, max_len, 0).astype(
513                 torch.LongTensor) for numb in numberized_list]
514
515             # 変数への変換
516             embed_caption = Variable(torch.FloatTensor(
517                 one_hot_embedded_list_padded)).cuda()
518             target_caption = Variable(
519                 torch.LongTensor(numberized_list_padded)).cuda()
520
521             # batch_size x max_seq_len x vocab_size から
                    max_seq_len x batch_size x vocab_size への転置.
522             embed_caption = embed_caption.transpose(0, 1)
523             target_caption = target_caption.transpose(0, 1)
524
525             loss = train(input_variable,
526                          embed_caption,
527                          target_caption,
528                          sentence_lens,
529                          encoder,
530                          decoder,
531                          encoder_optimizer,
532                          decoder_optimizer,
533                          criterion)
534             # 十回ごとに損失関数の値を表示
535             if i % 10 == 0:
536                 print(_ , i, loss)
537             #キャプションを生成してみる
538             if i % 100 == 0:
539                 print(train_id_to_captions[train_ids[0]],
540                       '\n', caption_image(training_vectors[0]))
541                 print(train_id_to_captions[train_ids[1]],
542                       '\n', caption_image(training_vectors[1]))
```

```
543                 print(train_id_to_captions[train_ids[2]],
544                       '\n', caption_image(training_vectors[2]))
545         outfile = "mk2_gensen_append_onehot"+str(len(train_ids))+"
                decoder_" + str(_)+"_"+str(vocabularySize) + ".model"
546         torch.save(decoder.state_dict(),outfile)
547         outfile2 ="mk2_gensen_append_onehot"+str(len(train_ids))+"
                encoder_" + str(_) + "_"+str(vocabularySize) +".model"
548         torch.save(encoder.state_dict(),outfile2)
```