平成29年度茨城大学工学部情報工学科
卒業研究論文

ニューラルネット翻訳における未知語処理

平成30年2月5日 提出

茨城大学　工学部情報工学科
13T4054L 根岸 睦

指導教員：新納 浩幸 教授

平成 29 年度茨城大学工学部情報工学科卒業研究

ニューラルネット翻訳における未知語処理

氏名：13T4054L 根岸 睦
指導教員：新納 浩幸 教授

論文要旨

本論文では英語から日本語への翻訳に対するニューラルネット翻訳（Neural Machine Translation：NMT）の翻訳精度向上のために翻訳処理時に発生する未知語の数を減らすことを試みる。NMT はニューラルネットを用いた機械翻訳の手法であり、Google 翻訳はこの NMT を発展させたシステムを採用している。その結果翻訳精度が向上したため現在 NMT の注目度は高い。また、未知語とは翻訳モデルを作成する際に使用した語彙データに存在しない単語のことで、特別な記号 $< UNK >$ として翻訳結果の文に出力される。未知語 $< UNK >$ の存在はそれだけで文に曖昧性をもたらし、翻訳が完全に完了したとは言えない。通常、未知語 $< UNK >$ は語彙データに存在する意味的に近いもので置き換えなどが成されるが翻訳の精度は落ちる。本論文では出力された未知語に対する置換ではなくそもそもの出力される未知語の数を減らすことを目的としている。

具体的に本論文ではモデルを作成する際に入力する目的言語の文を細かく分ける。通常、翻訳モデルの学習では原言語、目的言語ともに単語に分割するが、本論文では目的言語に対して単語以外にも文字や、サブワードと呼ばれる単語と文字の中間的な単位に分割してから入力を行う。サブワードに分割する手法として本論文では Byte Pair Encoding（BPE）を用いる。BPE の分割方法は、まず全ての文字を語彙に登録して、その中で最も頻度の高い 2 文字の連続を新たな語彙として登録するということを設定された最大語彙サイズまで繰り返す。これによって得られた語彙データにより文の分割を行うというものである。各分割法の例を挙げると「私はチーズが好きです。」という文を「私　は　チーズ　が　好き　です　。」のように分割するのが単語分割、「私　は　チ　ー　ズ　が　好　き　で　す　。」のように分割するのが文字分割、もし「私」と「は」、「で」と「す」が高頻度の文字であれば「私は　チ　ー　ズ　が　好　き　です　。」のように分割するのが BPE によるサブワード分割である。

実験では、田中コーパス (Tanaka Corpus) を用いる。実際に実験で使われた文数は翻訳モデルの学習に日英ともに 148,839 文、翻訳処理に英語 1,000 文、翻訳結果の評価に日本語 1,000 文である。また、実験の評価方法は未知語 $< UNK >$ の数に対しての翻訳精度の数値で測るものとした。翻訳精度の評価には BLEU（BiLingual Evaluation Understudy）と呼ばれる手法で出力された値を用いている。出力される BLEU 値は 0 ～1 で、数値が大きい方が翻訳精度が高い。実験の結果としては未知語の数は 1,000 文中、単語で分割した場合：623 個、文字で分割した場合：7 個、BPE によるサブワードで分割した場合：14 個となり、BLEU 値は単語で分割した場合：0.205、文字で分割した場合：0.201、BPE によるサブワードで分割した場合：0.189 となり、文字で分割するとき未知語の数に対しての評価値が高いことが分かった。

目次

第1章

はじめに

1.1 概要

本論文では英語から日本語へのニューラルネット翻訳（Neural Machine Translation：NMT）の翻訳精度向上を目的とし、そのために翻訳処理の際に発生する未知語の削減を行う。NMTはGooleが自社の翻訳システムに採用されたことで有名となり、翻訳精度も向上したことからNMTを改良したシステムの研究は活発になっている。しかしNMTには問題点がいくつかあり、それらの完全な解決策はいまだ発見されていない。

NMTの抱える問題のうち、今回は未知語に関する問題について扱う。未知語とは語彙データに存在しない語彙であり、翻訳結果に未知語$< UNK >$としてそのまま出現してしまうのでそのままでは文に曖昧をもたらし、翻訳が完全に完了した状態にはできない。未知語$< UNK >$は通常、語彙データに存在する最も意味的に近い語彙で置換されるのだがその方法では翻訳精度が下がる。そこで本論文では出力された未知語に対する処理ではなく、そもそもの出力される未知語を減らすことを考える。今回は、翻訳モデルを作成するに入力する目的言語（日本語）に対して、入力を行う前に文を細かく分けることで未知語の削減を目指す。通常、翻訳モデルの学習では入力文を単語に分割するが、本論文ではそれ以外にも文字や、サブワードにより分割してから入力を行い未知語の出現数と出力結果による翻訳精度の比較を行う。

実験では、田中コーパス (Tanaka Corpus)を用いる。実際に実験で使われた文数は翻訳モデルの学習に日英ともに148,839文、翻訳処理に英語1,000文、翻訳結果の評価に日本語1,000文である。また、実験の評価方法は未知語$< UNK >$の数に対しての翻訳精度の数値で測るものとした。翻訳精度の評価にはBLEU（BiLingual Evaluation Understudy）と呼ばれる手法で出力された値を用いている。出力されるBLEU値は0〜1で、数値が大きい方が翻訳精度が高い。実験の結果としては未知語の数は1,000文中、単語で分割した場合：623個、文字で分割した場合：7個、BPEによるサブワードで分割した場合：14個となり、BLEU値は単語で分割した場合：0.205、文字で分割した場合：0.201、BPEによるサブワードで分割した場合：0.189となり、文字で分割するとき未知語の数に対しての評価値が高いことが分かった。

1.2　構成

本論文では以下のような構成となっている。

2章では、本論文で扱うシステムの理論と概要について記述する。

3章では、実験の概要とその結果ついて記述する。

4章では、実験の結果を基に考察を記述する。

5章では、全体を通しての結論とまとめを記述する。

第2章

ニューラルネット翻訳

2.1 機械翻訳とは

機械翻訳とは、自然言語つまり人間が日常的に用いる言語を別の自然言語に変換するという作業をコンピュータで自動的に行うというものであり、1954年のジョージタウン大学などの研究グループにより発表されたことを皮切りに研究が進められてきた。研究初期の機械翻訳は人力で設定した対訳ルールに基づいて翻訳するルールベース機械翻訳と呼ばれるものであったが、1980年代後半にはIBMの研究グループにより統計的機械翻訳(Statistical Machine Translation：SMT)の研究が開始された。これは単語の翻訳確率や並べ替えの確率などの翻訳に必要な知識を対訳コーパスから統計的な情報として学習するものであり、これを拡張して2003年に提案されたものが、現在でもスタンダードな機械翻訳手法とされている句に基づく翻訳（Phrase-Based SMT：PBSMT）である。しかし2010年以降は国際会議に投稿されるSMTの論文の数も減り、研究も頭打ちの状態となっっていった。そんな中新しい手法として2014年に登場したのが本研究でも取り上げているニューラルネットを用いた機械翻訳、ニューラルネット翻訳(Neural Machine Translation：NMT)である。ニューラルネットとは人間の脳内にある神経回路網を人工ニューロンという数式的なモデルで表現したもので、近年の画像、医療、自動車事業、生産管理、そして機械翻訳を含む自然言語処理等の分野で高いパフォーマンスを発揮している。2016年11月にはGoogle翻訳がNMTを採用し、その翻訳精度を向上させたことで大きな話題となった。

2.2 ニューラルネット

2.2.1 モデル

ニューラルネット（以下NN）は、m 次元のベクトル x を n 次元のベクトル y に写す関数 f を推定する学習方法である。

$$y = f(x) \qquad s.t \quad x \in R^m,\ y \in R^n \tag{2.1}$$

NNでは f のモデルを図2.1に示すようなネットワークで表現できる。モデルは入力層、中間層、出力層

の 3 層となっており入力層のユニットは $m+1$ 個、中間層のユニットは $h+1$ 個、そして出力層のユニットは n 個存在する。ユニット間にはエッジというものが存在し、それぞれ重みと呼ばれる実数値が付与されている。また、入力層と中間層にある $b^{(1)}$ と $b^{(2)}$ はバイアスと呼ばれ、バイアスのユニットからユニット i の間には $b_i^{(1)}$ や $b_i^{(2)}$ という重みが付与されている。入力層のユニット i から中間層のユニット k には $w_{ki}^{(1)}$ という重み、中間層のユニット kから出力層のユニット j には $w_{jk}^{(2)}$という重みが付与されている。

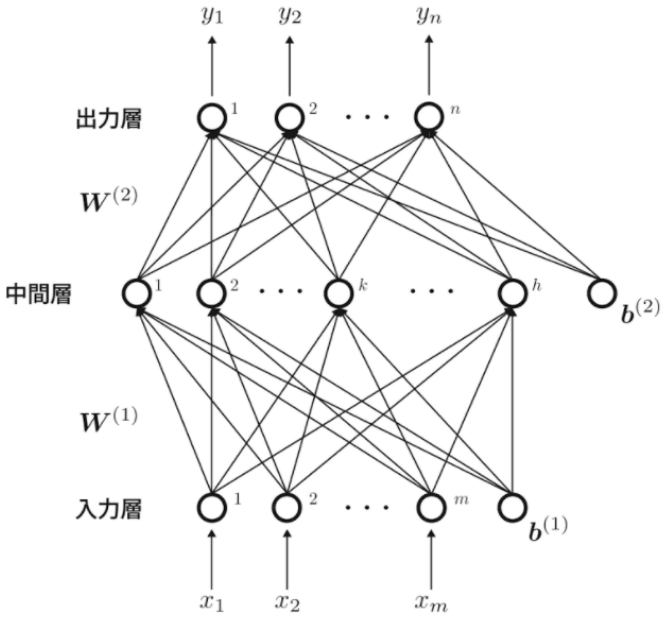

図2.1　NNにおける関数のモデル

入力層のユニット i には、入力であるベクトル xの第 i 次元目の値である x_i が入力される。そして x_i に重み $w_{ki}^{(1)}$ が乗じられて、中間層のユニット k に入力される。中間層のユニット k には、入力層の各ユニットから上記のように重みが乗じられた値が入力される。中間層のユニット k では、これらの入力値の和を取り、さらにその和をある活性化関数 σ に与えた結果の値を出力する。つまり、中間層のユニット k の出力 o_k は以下のように表される。

$$o_k = \sigma\left(\sum_{i=1}^{m} w_{ki}^{(1)} x_i + b_k^{(1)}\right) \tag{2.2}$$

標準的には、活性化関数 σ として以下のシグモイド関数が用いられる。

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.3}$$

シグモイド関数は頻繁に出現する重要な関数であり、グラフは図2.2のような形をしている。定義域は実数値全体で、値域は0から1を取る。このため、確率との相性が良い。また、シグモイド関数は負の部分では0、正の部分では1を取る階段関数を連続関数で近似したものとも見なせる。

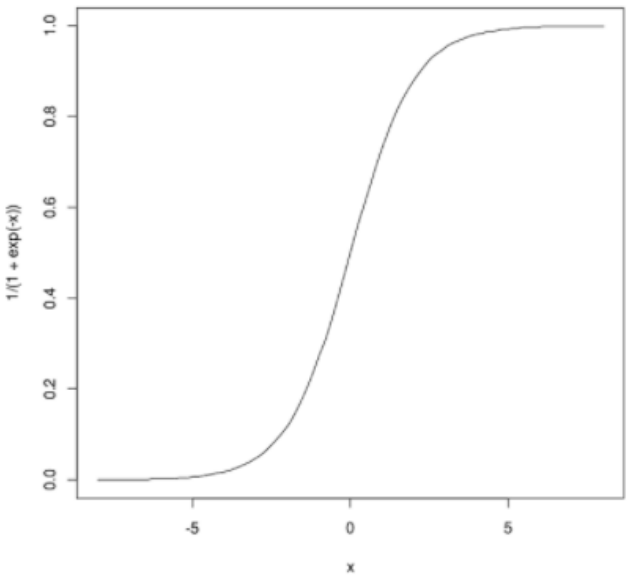

図2.2　シグモイド関数

h 行 m 列の大きさの行列でその k 行 i 列の要素が $w_{ki}^{(1)}$ となっている行列を $W^{(1)}$ と置く。そして、中間層の出力を並べたものをベクトル o と置く。

$$o = (o_1, o_2, \cdots, o_h) \tag{2.4}$$

すると、以下の関係式が成立する。

$$o = \sigma_1\left(W^{(1)}x + b^{(1)}\right) \tag{2.5}$$

中間層から出力層へも同様に考えれば、出力層の出力は以下のようになる。

$$\sigma_2\left(W^{(2)}o + b^{(2)}\right) = \sigma_2\left(W^{(2)}\sigma_1\left(W^{(1)}x + b^{(1)}\right) + b^{(2)}\right) \tag{2.6}$$

出力層のユニット j の出力を出力のベクトル y の第 j 次元目の値と見なせば、NNにおける関数 f のモデルは以下のようになる。

$$y = f(x) = \sigma_2\left(W^{(2)}\sigma_1\left(W^{(1)}x + b^{(1)}\right) + b^{(2)}\right) \tag{2.7}$$

2.2.2　確率的勾配降下法と誤差逆伝播法

前項で示した関数 f のモデルのパラメータは、バイアス $b^{(1)}$ と $b^{(2)}$ 及び重みの行列 $W^{(1)}$ と $W^{(2)}$ である。パラメータの個数は $b^{(1)}$ で h 個、$b^{(2)}$ で n 個、$W^{(1)}$ で mh 個、そして $W^{(2)}$ で hn 個あるので、合計 $V = h + n + mh + hn$ 個存在する。これら V 個のパラメータを θ で表すとする。結局、θ を訓練

データから推定することになる。訓練データは入力値 x とその出力値 y のペアのデータの集合であるので、このペアのデータが N 個あったとすると、訓練データ D は以下のような集合となる。

$$D = \{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)\} \tag{2.8}$$

NNでは適当なパラメータの初期値 $\theta^{(0)}$ から初めて $\theta^{(i)}$ を $\theta^{(i+1)}$ に更新していくことで θ を求める。更新の方法として現在標準的に用いられているものは、確率的勾配降下法（Stochastic Gradient Descent, 以下 SGD）と呼ばれる手法である。SGDでは訓練データの k 番目のデータ (x_k, y_k) の 2 乗誤差 E_k を減少させるように $\theta^{(i)}$ を $\theta^{(i+1)}$ に更新する。

$$E_k = \frac{1}{2}|f(x_k; \theta^{(i)}) - y_k|^2 = \frac{1}{2}\sum_{j=1}^{n}(f_j - y_j)^2 \tag{2.9}$$

ここで f_j と y_j はそれぞれ $(x_k; \theta^{(i)})$ と y_k の j 次元目の値である。上記の各データに対する更新を全データに対して何度か行うことで θ を求める。各データに対する更新については、具体的には以下の計算式でその時点の $\theta^{(i)}$ を $\theta^{(i+1)}$ に更新する。

$$\theta^{(i+1)} = \theta^{(i)} - \alpha \nabla E_k \tag{2.10}$$

ここで α は学習率と呼ばれるパラメータで、この値が大きいほど更新量が大きくなる。そして ∇E_k は E_k を各パラメータ θ_k で偏微分した式の θ の部分に $\theta^{(i)}$ を代入したものである。

$$\nabla E_k = \left(\left.\frac{\partial E_k}{\partial \theta_1}\right|_{\theta=\theta^{(i)}}, \left.\frac{\partial E_k}{\partial \theta_2}\right|_{\theta=\theta^{(i)}}, \cdots, \left.\frac{\partial E_k}{\partial \theta_V}\right|_{\theta=\theta^{(i)}} \right) \tag{2.11}$$

結局、SGDでは各 $\frac{\partial E_k}{\partial \theta_i}$ が求まればよいことがわかる。

これを求めるために、入力層を第 1 層、次の層を第 2 層と出力層に向かって順に数えることにして、一般的な第 l 層のユニット j を考える（図2.3参照）。

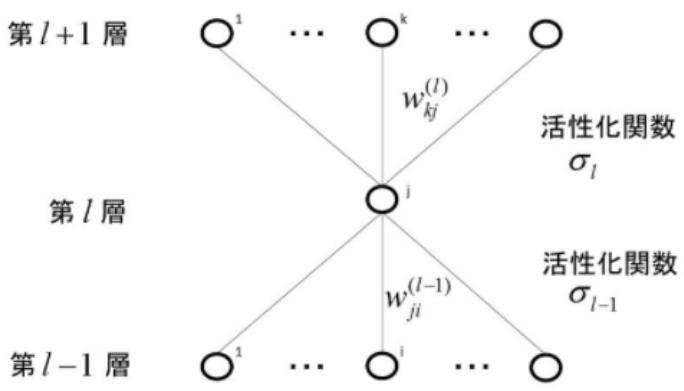

図2.3　第 l 層のユニット j

第 l 層のユニット j の出力を作る際に、活性化関数に与える入力を $\alpha_j^{(l)}$とする。つまり、第 l 層のユニット j の出力は $\sigma_1(a_j^{(l)})$であり、

$$a_j^{(l)} = \sum_i w_{ji}^{(l-1)} \sigma_{l-1}(a_i^{(l-1)}) + b_j^{(l-1)} \tag{2.12}$$

の関係があるため、合成関数の微分を用いると以下が成立する。

$$\frac{\partial E_k}{\partial w_{ji}^{(l-1)}} = \frac{\partial E_k}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{ji}^{(l-1)}} = \frac{\partial E_k}{\partial a_j^{(l)}} \sigma_{l-1}(a_i^{(l-1)}) \tag{2.13}$$

$$\frac{\partial E_k}{\partial b_j^{(l-1)}} = \frac{\partial E_k}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial b_j^{(l-1)}} = \frac{\partial E_k}{\partial a_j^{(l)}} \tag{2.14}$$

つまり、第 $l-1$ 層と第 l 層の間に存在するパラメータは $\frac{\partial E_k}{\partial a_j^{(l)}}$ を計算することで求められる。
さらに、他変数関数の合成関数の微分を用いると以下が成立する。

$$\frac{\partial E_k}{\partial a_j^{(l)}} = \sum_h \frac{\partial E_k}{\partial a_h^{(l+1)}} \frac{\partial a_h^{(l+1)}}{\partial a_j^{(l)}} \tag{2.15}$$

ここで

$$a_h^{(l+1)} = \sum_j w_{hj} \sigma_l(a_j^{(l)}) + b_k^{(l)} \tag{2.16}$$

なので

$$\frac{\partial a_h^{(l+1)}}{\partial a_j^{(l)}} = w_{hj} \sigma_l'(a_j^{(l)}) \tag{2.17}$$

が成立し、最終的に

$$\frac{\partial E_k}{\partial a_j^{(l)}} = \sum_h \frac{\partial E_k}{\partial a_h^{(l+1)}} w_{hj} \sigma_l'(a_j^{(l)}) \tag{2.18}$$

となっている。つまり $\frac{\partial E_k}{\partial a_j^{(l)}}$ を計算するには、 1 つ上の層の $\frac{\partial E_k}{\partial a_j^{(l+1)}}$ を計算できればよいことがわかる。
$\frac{\partial E_k}{\partial a_j^{(l)}}$ は第 l 層のユニットj の誤差を表している。よって出力層の誤差から入力層に向かって、逆向きに誤差を伝播させていくことでパラメータを求める形になっているため、この手法は誤差逆伝播法と呼ばれている。

なお、最も上位の層となる出力層におけるユニット j の出力が $\sigma_2(a_j^{(3)})$ であり、しかもそれは f_jであったので

$$\frac{\partial E_k}{\partial a_j^{(3)}} = (f_j - y_j) \frac{\partial f_j}{\partial a_j^{(3)}} = (f_j - y_j) \sigma_2'(a_j^{(3)}) \tag{2.19}$$

となり、さらに σ_2 が恒等関数であれば $\sigma_2'(x) = 1$ なので $\frac{\partial E_k}{\partial a_j^{(3)}} = f_j - y_j$ という差分 = 誤差という分かりやすい形になる。

また、活性化関数の微分の計算が必要となるが、シグモイド関数の場合、以下の関係式が利用できる。

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{2.20}$$

2.3　RNN

2.3.1　基本概念

Recurrent Neural Networks（以下RNN）とは時系列データに対応したNNである。具体的には文章などの連続的な情報を利用することができる。次の言葉を予測したい場合などは、その前の言葉を覚えておく必要があるが、RNNは以前に計算された情報を記憶することが可能なのである。

RNNのネットワークを概念図にすると図2.4、それを時間展開すると図2.5となる。入力は時刻 $t = 1$から $t = T$ までの時系列データ $x_1, x_2, \cdots, x_T$ であり、出力は各時刻 t に対する出力 y_t の列、$y_1, y_2, \cdots, y_T$ である。また、$W^{(1)}$ 、$W^{(2)}$ および H は線形作用素となっている。

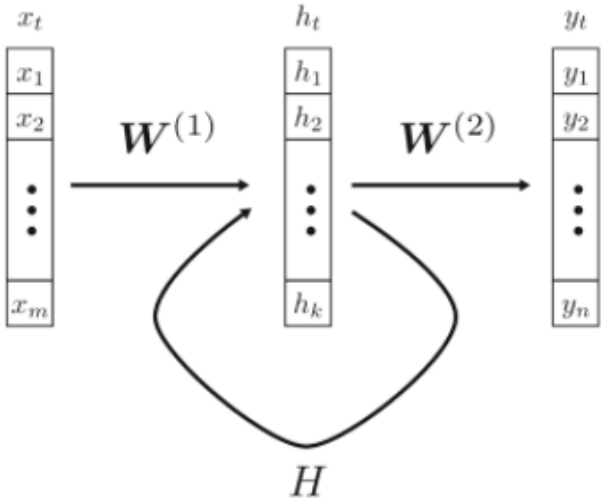

図2.4　RNNの概念図

時刻 t におけるネットワークへの入力は、入力データ x_t と時刻 t における中間層への入力 h_{t-1}であり、出力は通常の y_t と時刻 $t+1$ における中間層への入力となる h_tである。これらには以下の関係がある。s は適当な活性化関数とする。

$$h_t = tanh(W^{(1)}x_t + Hh_{t-1} + b_h)s \tag{2.21}$$

$$y_t = s(W^{(2)}h_t + b_w) \tag{2.22}$$

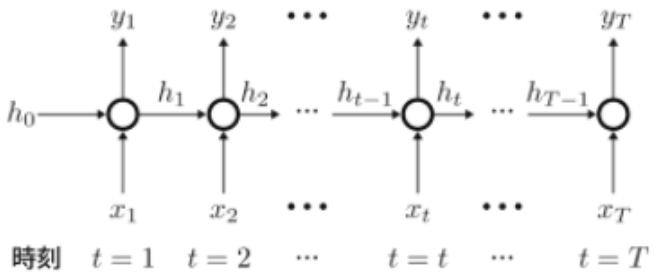

図2.5　時間展開したRNNのネットワーク

ここで b_h は線形作用素 H のバイアス、b_w は線形作用素 $W^{(2)}$ のバイアスである。$x_1, x_2, \cdots, x_T$ が与えられた時、まず h_0 を適当に設定し、 x_1 に対して上記の式から y_1 と h_1 を得る。次に x_2 に対して上記の式から y_2 と h_2 を得る。これを繰り返していくことで、最終的に $y_1, y_2, \cdots, y_T$ が得られる。

そしてRNNの特徴である以前の情報を記憶するという役割を担っているのが H である。RNNは H を利用して、過去の時系列のデータの情報を圧縮した形で次の時刻に引き継いでいる。また、学習では誤差の累積から誤差逆伝播法を用いる。各 x_i に対する教師信号を t_i としておくと、x_1 に対して前述した手順で y_1 が求まるので、 t_1 と y_1 から誤差を求める。次に、x_2 に対しても前述した手順で y_2 が求めるので、t_2 と y_2 から誤差を求める。これを繰り返していき、最後に x_T に対して求まった y_T から t_T と y_T との誤差を求め、ここまでの誤差の累積に加え、最終的な誤差の累積を求める。ここから誤差逆伝播法を用いれば、$W^{(1)}$ 、$W^{(2)}$ および H を求めることができる。

2.3.2　言語モデル

自然言語処理の分野では、RNNの代表的な応用として、言語モデルの構築がある。RNNで作られた言語モデルは、RNNLM（RNN Language Model）と呼ばれる。

言語モデルとは、文 s が現れる確率 $P(s)$ を与える確率モデルである。文 s が $w_1 w_2 \cdots w_N$ という N 個の単語の列である場合、$P(s)$ は以下のように分解できる。

$$\begin{aligned} P(s) &= P(w_1, w_2, \cdots, w_N) \\ &= P(w_1)P(w_2|w_1)P(w_3|w_1w_2)\cdots P(w_N|w_1w_2\cdots w_{N-1}) \\ &= P(w_1)\prod_{t=2}^{N} P(w_t|w_1w_2\cdots w_{t-1}) \end{aligned} \tag{2.23}$$

言語モデルは $P(w_t|w_1w_2\cdots w_{t-1})$ の部分に注目すると「ある単語列が与えられた時に次に現れる単語を予測する」モデルとなっている。また、言語モデルが得られることの利点は自然言語処理で利用できる場面が多々あるからと言える。例えば

$s =$ 彼女はいがいにも武道の心得があった。

という文の”いがい”の部分を漢字に変換する場合、”意外”と”以外”、もしくは”遺骸”の3通りの可能性がある。どちらが正しいかを判定する問題は様々な手法が存在するが、どの手法も本質的には3通りの確率を比較して高いほうを選ぶというやり方になっているはずである。

また、直接的に言語モデルは文生成に利用できる。翻訳や要約など最終的に文を出力しなければならない自然言語のアプリケーションでは、何らかの言語モデルに相当するものが利用されている。

RNNLMでの時刻 t におけるネットワークは図2.6の形になっている。

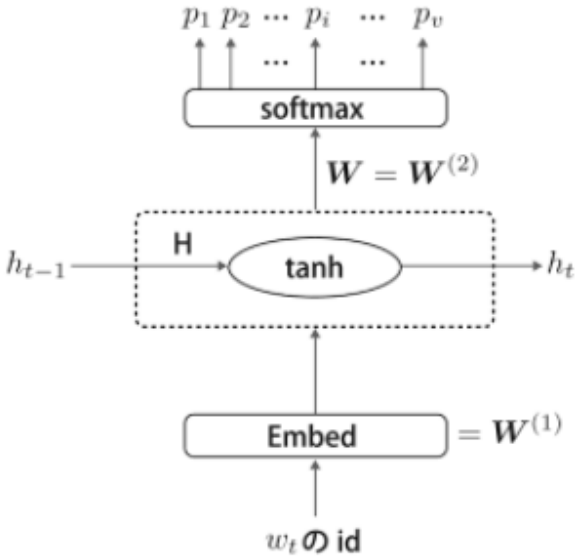

図2.6　時刻 t におけるRNNLM

第 1 層目の線形作用素 $W^{(1)}$ にEmbedを使い高次元の実数ベクトル化を行っている。出力は各単語が現れる確率となるので、単語の種類数を V とすると、出力 y_t は以下の V 次元のベクトルである。

$$y_t = (y_1, y_2, \cdots, y_v) \tag{2.24}$$

そして y_t は単語id i の単語が出現する確率になる。このため出力層からの出力には softmax関数を被せることになる。

softmax関数とは第 i 次元の値が x_i であるとき、x_i を確率値に変換する関数であり、以下の式で表される。

$$softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)} \tag{2.25}$$

2.3.3　LSTM

通常のRNNではネットワーク階層が深くなりすぎると、誤差逆伝播のアルゴリズムでは勾配が消失したり発散したりする問題が生じる。その結果、長期依存をうまく扱えないという状況に陥る。長期依存というのは、文の最初の法に出てきた単語が、かなりあとの単語の出現に影響を与える現象のことである。例えば「今日、イタリアンレストランにディナーに行ったのですが、一番美味しかった料理は○○です。」という文があった時に、「○○」の直前まで読んで、「○○」の単語を推定する時、「○○」よりもかなり前方に出現した「イタリアンレストラン」という単語が影響を与えている。これが長期依存である。この場合、素のRNNの誤差逆伝播では「○○」からの勾配（誤差）を「イタリアンレストラン」まで伝播させられず、結果として長期依存をうまく扱えなくなっている。

この点を改良したのがLSTM（Long Short Term Memory）である。LSTMのネットワーク図を図2.7に示す。LSTMの場合、中間層はLSTMブロックと呼ばれる。図2.7のようにLSTMブロックの中身を隠蔽してしまうと、LSTMは素のRNNと同じ形になる。

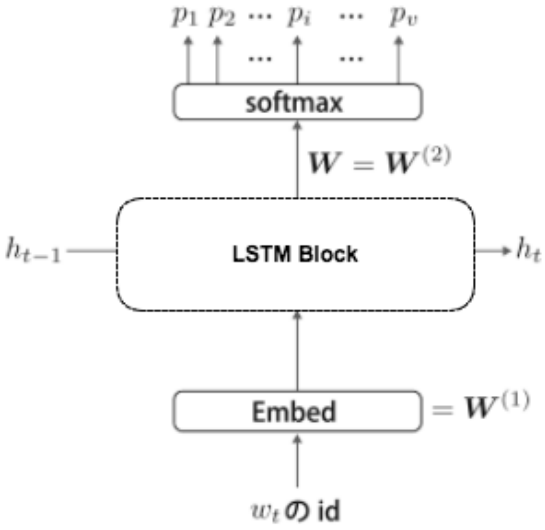

図2.7　LSTMのブロック

そしてそのLSTMブロックの中身を更生する要素は、記憶セル、入力ゲート、出力ゲートおよび忘却ゲートの4つである。ゲートとは実質的には関数である。つまりLSTMブロックの中身は上記のゲートの関数を合成したものであり、LSTMブロック自身が 1 つの関数と見なせる。

LSTMブロックの中身の図を描くと図2.7のようになっている。なお図2.7では、説明の簡略化のためにLSTMブロックへの第1層からの入力を x_t 、LSTMブロックの出力を y_t としておく。また図2.7では、ベクトルに重みを付与する線形作用素は記述していない。基本的にベクトルには、線形作用素により重みが

付与されて次にユニットに渡る。

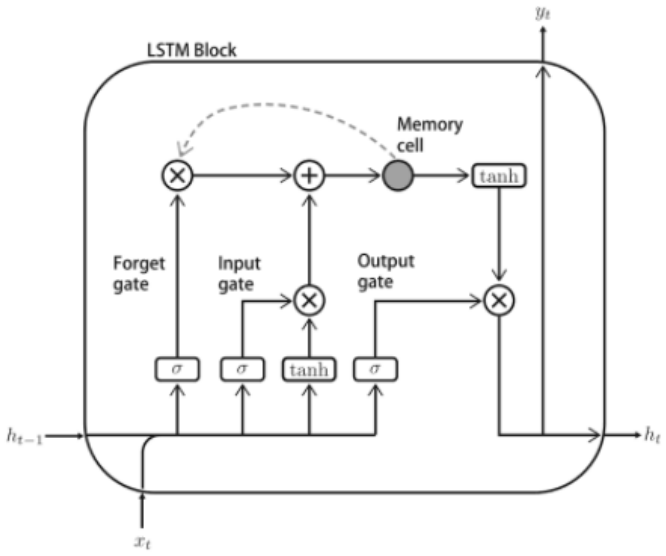

図2.8　LSTMのブロックの中身

時刻 t におけるLSTMブロックへの入力は1つ下の層の x_t と、時刻 $t-1$ におけるLSTMブロックの出力 h_{t-1} である。x_t と h_{t-1} は、素のRNNと同じく、以下のように変換される。

$$\bar{z_t} = W_z x_t + R_z h_{t-1} + b_z \tag{2.26}$$

$$z_t = tanh(\bar{z_t}) \tag{2.27}$$

入力ゲートにおける変換は以下である。

$$\bar{i_t} = W_i x_t + R_i h_{t-1} + b_i \tag{2.28}$$

$$i_t = \sigma(\bar{i_t}) \tag{2.29}$$

忘却ゲートにおける変換は以下である。

$$\bar{f_t} = W_f x_t + R_f h_{t-1} + b_f \tag{2.30}$$

$$f_t = \sigma(\bar{f_t}) \tag{2.31}$$

そして次の記憶セルでの変換が以下である。

$$c_t = i_t \otimes z_t + f_t \otimes c_{t-1} \tag{2.32}$$

式2.32では c_{t-1} が出てきているが、これは記憶セルの時刻 $t-1$ 、つまり1つ前の時刻での記憶セルの出力である。結局記憶セルでは、現在の出力を次の時刻の処理で使うために、一時的に記憶している。また、演算記号 $\otimes$ は要素同士の積を表している。

次に出力ゲートにおける変換は以下である。

$$\bar{o_t} = W_o x_t + R_o h_{t-1} + b_o \tag{2.33}$$

$$o_t = \sigma(\bar{o_t}) \tag{2.34}$$

最後にLSTMブロックの出力を作る部分が以下である。

$$y_t = h_t = o_t \otimes tanh(c_t) \tag{2.35}$$

LSTMブロックの実体は式2.26から式2.35により定義される合成関数である。

また、学習の対象はLSTMブロックへの入力である x_t の次元を m 、出力である y_t の次元を n 、h_t の次元も n とすると、$m \times n$ の行列として、W_z 、W_i、 W_f、 W_o、 $n \times n$ の行列として、R_z 、R_i、 R_f、R_o、n 次元ベクトルであるバイアスとして b_z、b_i、 b_f、 b_o のパラメータである。

さて、上記のような構造を持つLSTMが、なぜ素のRNNで問題となった勾配消失問題を回避できるのか。それを示すのは非常に困難であるが、大雑把に説明すると勾配消失問題は勾配に関してある条件が成り立つ時に生じるが、記憶セルが行っている式2.32が成立するように出力の y_t を作成すると、その条件が成り立たなくなる。ということになる。ただし注意として勾配消失問題から長期依存の問題が生じているが、勾配消失問題を回避できたとしても完全な長期依存の問題の解決にはならない。LSTMを使うと長期依存の問題が少し緩和される程度だということを留意しておきたい。

2.4　翻訳モデル

自然言語処理においてDeep Learning、すなわち十分なデータ量をもとに人間の力なしに機会が自動的にデータから特徴を抽出する学習の応用として注目されているものといえば翻訳が挙げられる。翻訳というと英語から日本語への翻訳といった言語間翻訳をイメージするであろうが、ここでの翻訳は、もう少し一般的に記号列から記号列への変換のイメージである。自然後処理であるので入出力のどちらか一方の記号列は言語になっているが、もう一方の記号列は言語の他、様々なメディアであっても良い。例えば画像からそのキャプションの生成や、文からその内容の絵の描画、あるいは音声からのその文生成（音声認識）など様々な応用が、一般的に翻訳と呼べるのである。Deep Learningでは、入力層と出力層を結ぶ中間層が入力と出力との間にある共通の意味を表しているモデルを考え、あとは膨大な入出力の組から中間層を学習させれば翻訳ができる、という単純な発想である。

本節では、NMTを構成する要素であるEncoder-Decoder翻訳モデル、Attentionの解説をする。

2.4.1　Encoder-Decoder翻訳モデル

Encoder-Decoder翻訳モデルとは、原言語のRNNのLSTMブロック列と目的言語のRNNのLSTMブロック列とを連結したモデルである。ネットワークの略図を図2.9に示す。

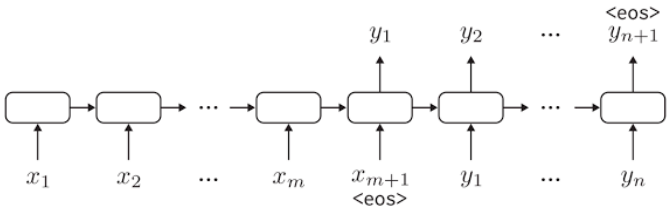

図2.9　Encoder-Decoder翻訳モデルの略図

原言語の文 $x_1x_2\cdots x_mx_{m+1}$ を目的言語に翻訳することを考える。ここでの各 x_i は単語であり、x_{m+1} は文末記号$< eos >$である。

まず、原言語の各単語 x_t が時系列的に入力される。RNNと同じように、中間層であるLSTMブロックでは、その時点までの文脈情報 h_t を次の中間層（LSTMブロック）へ渡す。通常のRNNではLSTMブロックから出力層へ渡す出力もあるが、原言語側ではそれはない。原言語側では最後に単語$< eos >$、つまり文末記号を読み込む。

ここから目的言語側のRNNに処理が移る。目的言語側のRNNでは通常のRNNと同じように、中間層は次の中間層へ h_{m+1} を出力する。同時に目的言語側の単語 y_1を出力する。そして、次の入力が単語 y_1 になる。

以降は通常のRNNと同じ処理が繰り返され、目的言語側の単語列 $y_1y_2\cdots y_ny_{n+1}$ が生成される。文末記号$< eos >$を生成したら終了とする。つまり y_{n+1} は文末記号$< eos >$である。Encoder-Decoder翻訳モデルは、こうして作られた目的言語の文 $y_1y_2\cdots y_n$ を原言語の文 $x_1x_2\cdots x_m$ の翻訳と考える。

学習には対訳ペアが必要となる。 $x_1x_2\cdots x_m$ の翻訳が $t_1t_2\cdots t_n$ だったとする。上記で述べた x_{m+1} つまり入力文の文末記号$< eos >$をよみこんだ際に、出力される y_1 と t_1 との誤差が損失になる。次の入力は実際の翻訳処理では y_1 となるが、学習の段階では t_1 である。そして出力される y_2 と t_2 との誤差が損失となり、その損失を累積する。これを繰り返して y_n と t_{n+1} 、つまり文末記号$< eos >$との誤差である損失を計算し、この損失を累積して、最終的に損失の累積を得る。この損失の累計から誤差逆伝播を行って、パラメータの学習を行う。実際の翻訳は学習したモデルを用いて行うが処理は学習とほぼ同じで、翻訳の際に出力した単語を次の入力として使っているかの違いしかない。翻訳処理も$< eos >$を出力したら処理は終了するが、学習がうまくいっていないと永遠に$< eos >$を出力しない場合もあるので、プログラムを作成する際はある程度の単語数が出力されたら強制的に終わらせるように処理したほうが良い。

2.4.2　Attention

Encoder-Decoder翻訳モデルでは、原言語の入力文の情報が、$< eos >$が入力された後の中間層のベクトル h だけに押し込まれている形になっている。これではLSTMが勾配消失問題を解決してくれているといっても、精度の良い翻訳を出すのは難しい。そこでDecoderがエンコードすべき箇所を制御する手法がAttentionと呼ばれる手法である。

大まかに言えば、Encoder側で入力単語に対する中間層の情報を大域的にとっておいて、Decoder側でそれを利用するという形である。本質的にやっていることは両言語のフレーズの対応、すなわちアライメントに類する情報を学習している形である。

Encoder-Decoder翻訳モデルにAttentionを導入したモデルのネットワークの概略図は、図2.10となる。

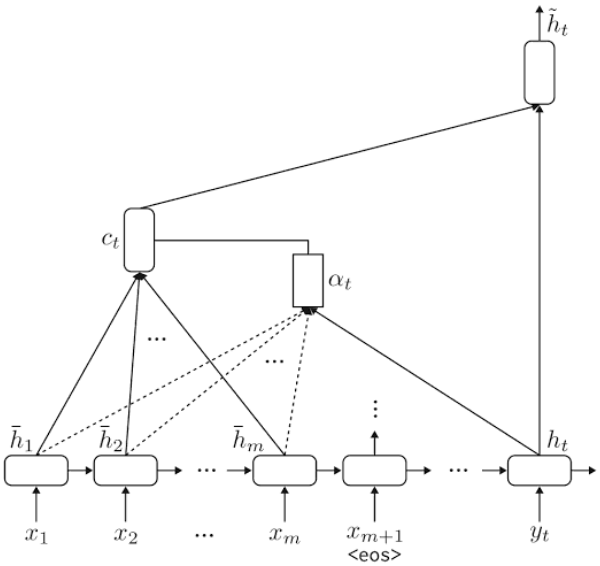

図2.10　Attentionを導入したモデル

実際の翻訳処理は、Encoder側の入力 $x_1, x_2, \cdots, x_m$ は通常のEncoder-Decoder翻訳モデルと同様である。ただ1つ異なるのは、各 x_i に対して、中間層の出力である $\overline{h}_i$ を大域的に保持している点である。

次に、Decoder側も基本的にはEncoder-Decoder翻訳モデルと同じ処理である。y_t の入力から $\overline{y}_t$ を出力し、この $\overline{y}_t$ を次の入力 y_{t+1} とする。違いは $\overline{y}_t$ の作り方にある。

まず y_t に対する中間層の出力を h_t とする。Encoder側で保持しておいた $\overline{h}_i$ を使って、以下の値 $\alpha_t(i)$ を計算する。

$$\alpha_t(i) = \frac{exp((\overline{h}_i, h_t))}{\sum_{j=1}^{m} exp((\overline{h}_i, h_t))} \tag{2.36}$$

$(\overline{h}_i, h_t)$ は $\overline{h}_i$ と h_t の内積を表す。結局、y_t と x_i との類似度を正規化したものが $\alpha_t(i)$ である。この

$\alpha_t(i)$ と $\overline{h}_i$ を使って、以下のcontext vector c_t を作成する。

$$c_t = \sum_{i=1}^{m} \alpha_t(i)\overline{h}_t \tag{2.37}$$

次に c_t と h_t を連結させたベクトル $[c_t;h_t]$ を作り、これを線形作用素 W_c で重みを付けて、活性化関数 tanh を被せることによって、y_t に対する中間層の出力 $\widetilde{h}_t$ を作成する。

$$\widetilde{h}_t = tanh(W_c[c_t;h_t]) \tag{2.38}$$

あとは素のEncoder-Decoder翻訳モデルと同じように、 $\widetilde{h}_t$ に対して線形作用素 W で重みを付けて、softmax関数を通して $\overline{y}_t$ を作成する。

次に学習の処理においては、パラメータとして W_c が増えているだけである。ただ、注意すべき点は、Encoder側で保持している $\overline{h}_i$ にはパラメータが含まれていないことある。 x_i に対して中間層の出力である $\overline{h}_i$ が作成された時に、大域的に保持する $\overline{h}_i$ はその時点の値がコピーして作られ、実体の $\overline{h}_i$ は次の中間層へ渡される。つまり誤差逆伝播の際に、大域的の保持している $\overline{h}_i$ には影響が及ばないということである。

2.4.3　NMT

NMTは前述の通り、基本的にはEncoder-Decoder翻訳モデルにAttentionを取り入れたものとなっており、処理の流れはEncoderで入力文を実数値の集合であるベクトル表現に符号化（embedding）を行い、Attentionにより符号化された入力文のどこに注目すべきか決定して、それらの情報を基にDecoderがsoftmax関数により算出した確率値を基に出力する単語を決定するというものになっている。

NMTの特徴をまとめると、まずモデルは単一で完結するend-to-endのモデルとなっており、従来までの機械翻訳で用いた単語アライメントやフレーズテーブルの構築が不要である。また、最初の入力文と最後の出力文以外の部分は全てベクトルのような数値で表現されており、翻訳の過程は数値の足し算や掛け算で実現される。モデル自体は非常にシンプルになるが、翻訳の過程の解釈は困難である。利点としては単語をベクトルで表現するembeddingのおかげで、同じ意味だが別の単語であっても、同じような翻訳を出力することができる点である。

また問題点も現状いくつか存在する。まず、翻訳文の生成において符号化された入力文とAttentionの情報からDEcoderが次に出力する単語を1つずつ決定し、$< eos >$が出力された時点で翻訳が完了するのだがここで問題なのは翻訳の終了をコントロールすることができないことである。Decoderが $< eos >$を出力するまで翻訳は継続し、いつeosが出力されるかは今の所知る術がない。このため入力文の全てが過不足なく訳出されるという保証がないのである。Attention機構が次に訳出するべき箇所を決定するのだが、過去に訳出した箇所の情報を持っていないため、同じところを複数回訳出したり（重複訳：over translation）、

まだ訳出していない部分があるにもかかわらず、$< eos >$にAttentionして翻訳を終了させてしまったり（訳抜け：under translation）ということが起こる。また、未知語の問題も存在する。未知語とは翻訳モデルを作成する際に使用した語彙データに存在しない単語のことで、特別な記号$< UNK >$として翻訳結果の文に出力される。未知語$< UNK >$の存在はそれだけで文に曖昧性をもたらし、翻訳が完全に完了したとは言えない。通常、未知語$< UNK >$は語彙データに存在する意味的に近いもので置き換えなどが成されるが翻訳の精度は落ちる。これらの問題点に関して様々な研究者が可決に向けて研究しているところだが、今のところ決定的な方法は考案されていない。

また、本研究で用いたNMTは以下のページよりcloneしたものである。

https://github.com/mlpnlp/mlpnlp-nmt.git

2.5　Python

Pythonとは、1991年に登場したプログラミング言語である。特徴としてはフリーソフトかつオープンソース、クロスプラットフォーム、インタラクティブシェルであり、学習の面でも文法がシンプル、インデント（字下げ）の強制、演算処理や機械学習で利用可能なライブラリが複数用意されているなど非常に理解しやすいものとなっており、主に米国で広く普及している言語となっている。

本研究で用いられるPythonのバージョンは2.7.0と3.6.0である。

2.6　Chainer

ChainerはPFI/PFN（株式会社 Preferred Infrastructure / 株式会社 Preferred Networks）が開発するDeep Learningのフレームワークであり、2015年6月にMITライセンスに基づくフリーソフトウェアとして公開されている。

基本、Linux上で動かすことを想定しており、推奨されているプラットホームはUbuntu 14.04/16.04 LTS 64bit および CentOS 7 64bitである。いずれかの環境であれば、インストール可能である。Chainerの特徴としては開発者のブログ記事に以下のようにまとめられている。

- Pythonのライブラリとして提供（要Python2.7+）
- あらゆるニューラルネットの構造に柔軟に対応
- 動的な計算グラフ構築による直感的なコード
- GPUをサポートし、複数のGPUを使った学習も直感的に記述可能

本研究では主にNMTの処理に用いられている。利用しているChainerのバージョンは、2.1.0である。

2.7　Mecab

MeCabは 京都大学情報学研究科－日本電信電話株式会社コミュニケーション科学基礎研究所 共同研究ユニットプロジェクトを通じて開発されたオープンソース 形態素解析エンジンである。形態素解析とは、ある文章・フレーズを意味を持つ最小限の単位（単語）に分解し、文章やフレーズの内容を判断するために用いられる。本論文では、日英の対訳データを単語で分かち書きする際に使用した。

今回、文の単語分割をする際に用いた実行コードは以下の通りである。

```
$ mecab –O wakati 単語分割を施すファイル > 出力ファイル
```

2.8　BPE

Byte Pair Encoding（BPE）は文章をサブワードと呼ばれる単語と文字の中間に位置する単位に分割するための手法の一つである。BPEの処理の大まかな手順は以下の通りである。

1. 全ての文字を語彙に登録する。
2. データの中で最も頻度の高い2文字の連続を新たな語彙として登録する。
3. 設定された最大語彙サイズまで 2. を繰り返す。

本論文ではBPEにより日英の対訳データをサブワードで分かち書きする際に使用した。なお、使用したBPEは以下のページよりcloneしたものである。

https://github.com/rsennrich/subword-nmt.git

また、今回BPEによる文のサブワード分割に用いた実行コードは以下の通りである。閾値は3、最大語彙サイズは10000とした。

vocabファイル作成コード

```
cat origin/origin.de origin/origin.en | ./learn_bpe.py -s \
10000 -o bpe.model
./apply_bpe.py -c bpe.model < origin/origin.de | \
./get_vocab.py > vocab.de
```

BPE実行コード

```
./apply_bpe.py -c bpe.model --vocabulary vocab.de \
--vocabulary-threshold 3 < origin/origin.de > origin_bpe.de
```

2.9　BLEU

BLEU（BiLingual Evaluation Understudy）は機械翻訳システムの翻訳精度を自動的に評価する手法の1つである。特にBLEUは、n-gramマッチ率に基づく手法を用いている。以下に式を示す。

$$BLEU = BPexp\left(\sum_{n=1}^{N} w_n log p_n\right) \tag{2.39}$$

$$w_n = 1/N \tag{2.40}$$

$$p_n = \frac{\sum_i \text{出力中}\, i \,\text{と正解文中}\, i \,\text{で一致した}\, n-gram \,\text{数}}{\sum_i \text{出力文}\, i \,\text{中の全}\, n-gram \,\text{数}} \tag{2.41}$$

ここで、式2.39のBPは、機械翻訳の出力文が、正解文よりも短い場合のペナルティである。また、BLEUは文単位ではなくドキュメント単位での使用を前提としている。そのため、本研究では、文単位で使用するために変更が行なわれている。つまり、 $P_n = 1$ の場合にBLEU値が0になってしまうため、 $P_n \not\to 0$ であるような最大のnをNとして使用する。BLEUは0から1の値を出力し、スコアが大きいほど評価が良い。 なお、本論文で用いたBLEU評価関数は以下のページよりcloneしたものを使用している。

https://github.com/lingng/BLEU-score.git

また、今回BLEU値算出に用いた実行コードは以下の通りである。

```
$ python calculatebleu.py /path/to/candidate /path/to/reference
```

第3章

実験

3.1 概要

本実験では、まず英語から日本語へのNMTの翻訳モデルを作成する。翻訳モデルを作成する際に原言語と目的言語の対訳データを用いるのだがNMTでは空白で区切られた単位を語彙と見なし学習を行う。このため何らかの分割を文に施さなければならない。今回は以下の3パターンで翻訳モデルを作成する。

1. 英語（単語分割）＋　日本語（単語分割）
2. 英語（単語分割）＋　日本語（文字分割）
3. 英語（単語分割）＋　日本語（サブワード分割）

分割処理を行う際、単語分割にはMecab、文字分割には自作のプログラム（プログラムコードの詳細は付録Aを参照のこと）、サブワード分割にはBPEを用いた。また、翻訳モデルを作成する際のバッチサイズは32で13epoch行った。

次に作成されたモデルにより翻訳処理を行う。入力する原言語の文はMecabにより単語分割されたものを用いる。また、翻訳処理は13epoch行った。（翻訳モデル作成および翻訳処理に用いた実行コードは付録Bを参照のこと）

NMTの翻訳モデル作成および翻訳処理における実行環境は表4.1の通りである。

表3.1　NMT実行環境

OS	Ubuntu 16.04 LTS
GPU	GeForce GTX 1050 Ti
Python	2.7.0
Chainer	2.1.0

出力された翻訳結果の文に対してパターン 2 とパターン 3 については一度空白を削除してからMecabで単語分割を行った。理由としてはBLEU評価値を出すためには原言語側と目的言語側の文の分割方法を合わせる必要があったからである。また空白を削除するプログラムは自作のものを用いた。（プログラムコー

ドの詳細は付録Cを参照のこと）

3.2　測定方法

本実験では未知語をどれだけ削減できたかということを測定するために、各パターンにおいて得られた翻訳結果の文に対して未知語$< UNK >$が翻訳文に出現している個数に対してのBLEU値による翻訳精度の値で測るものとした。また、未知語の個数をカウントするプログラムは自作のものを用いた。（プログラムコードの詳細は付録Dを参照のこと）

測定環境は表3.2の通りである。

表3.2　測定環境

OS	MacBook Air OS X EL Caption version:10.116
IDE	Eclipse verion: Neon.2 Release (4.6.2)
Python	3.6.0

3.3　使用データ

英日の対訳データは田中コーパス (Tanaka Corpus)を用いた。実験で使われた文数は翻訳モデルの学習に日英ともに148,839文、翻訳処理に英語1,000文、翻訳結果の評価に日本語1,000文である。

3.4　結果

各パターンの測定結果は表3.3の通りである。

表3.3　測定結果

	未知語の個数	BLEU値
パターン1	623	0.205
パターン2	7	0.201
パターン3	14	0.189

これよりパターン２の「翻訳モデル作成時の目的言語側の文に文字分割を適応した場合」が本実験中最も未知語の個数に対するBLEU値が高いことがわかった。

第4章

考察

NMTにおいて未知語処理を考える場合、翻訳結果がより優れたものになったかどうか判断するのは容易ではない。なぜなら機械翻訳システムの評価法としてのスタンダードであるBLEUでの評価には文字の並び、単語数を主に用いているからである。これは文中に未知語$< UNK >$が数多く存在したとしても他単語の並び順や総単語数などが正解文と一致していれば高い評価値を出してしまう。実際本実験でも最も未知語の多かった単語による分割を行ったパターンではBLEU値も最も高かった。しかし、単語の中身が以下に出鱈目であろうと文字の並びと単語数が近ければ正解文に近い物であるというのもその通りなのである。ここに機械翻訳の評価が困難である理由があると考える。

次に、本実験におけるサブワード分割にも注目していきたい。今回サブワード作成に用いたBPEはデータ圧縮の手法であり、それを機械翻訳に適応させたものである。今回の実験ではBPEの設定は閾値3、最大語彙数10,000としたが、これらの値を変更した際にどれほど未知語やBLEU値の値に変動がみられるのか実験を行いたかった。しかし、翻訳システムの作成には約1日かかるので何度も実験を行えなかったのは残念なところである。また、サブワードと一口に言ってもその作成法は多岐に渡り、NMT用にカスタマイズされたサブワード分割法も存在するので、それらを用いた場合の実験も行う必要がある。

第5章

結論

本論文では、NMTにおける未知語の出現数を減らすことを目標とした。結果として文字分割を翻訳モデル作成時の目的言語側に施すことで高い評価を得ることができた。しかし、サブワード分割を用いた場合のBLEU値が最も低いという点が気になった。今回の実験で用いたサブワード分割は他の手法も多数存在する。今後は本実験とは違う分割手法、実行環境設定などで実験を行いたい。また、単語分割以外にも未知語に対するアプローチは存在するのでそれらの手法を試していきたい。

謝辞

本論文を作成するにあたり、ご指導を頂いた情報工学科の新納教授に心より感謝致します。また、日常の議論を通じて多くの知識や示唆を頂いた新納研究室の皆様に深く感謝致します。

参考文献

[1] 新納浩幸, Chainer v2による実践深層学習, 2017/9/15, p20-26, p94-113, p129-143

[2] 中澤敏明, 機械翻訳の新しいパラダイム：ニューラル機械翻訳の原理, J-STAGE 60巻(2017)5号 p.299-306

[3] Papineni Kishore, Salim Roukos, Todd Ward, Wei-Jing Zhu: “BLEU: a method for automatic evaluation of machine translation”, 40th Annual meeting of the Association for Computational Linguistics pp. 311-318, 2002.

[4] Rico Sennrich, Barry Haddow and Alexandra Birch (2016): Neural Machine Translation of Rare Words with Subword Units Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL 2016). Berlin, Germany.

付録A

日本語文の文字分割

```
1  # -*- coding: utf-8 -*-
2  import codecs
3  import sys, io
4  sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding='utf-8')
5
6  #文字分割を施すファイルおよび出力ファイル
7  # f = codecs.open('tanaka_test.de', 'r', 'utf-8')
8  # g = codecs.open('tanaka_test2.de', 'w', 'utf-8')
9  # f = codecs.open('tanaka_dev.de', 'r', 'utf-8')
10 # g = codecs.open('tanaka_dev2.de', 'w', 'utf-8')
11 f = codecs.open('tanaka_train.de', 'r', 'utf-8')
12 g = codecs.open('tanaka_train2.de', 'w', 'utf-8')
13 for row in f:
14     chars = list(row)
15     jplist = []
16     for ch in chars:
17         if(ch is '\n'):
18             del jplist[-1]
19             jplist.append(ch)
20             continue
21         if(ch is not '␣'):
22             jplist.append(ch)
23             jplist.append('␣')
24     for w in jplist:
25         g.write(w)
26 f.flush()
27 f.close()
28 g.flush()
29 g.close()
```

付録B

NMT翻訳モデル作成、翻訳処理

```
# 語彙ファイルの準備例
for f in 英日ファイルtrain ;do \
    echo ${f} ; \
    cat ${f} | sed '/^$/d' | perl -pe 's/^\s+//; s/\s+\n$/\n/; s/ +/\n/g' | \
    LC_ALL=C sort | LC_ALL=C uniq -c | LC_ALL=C sort -r -g -k1 | \
    perl -pe 's/^\s+//; ($a1,$a2)=split;
        if( $a1 >= 3 )\
        { $_="$a2\t$a1\n" }\
        else\
        { $_="" } ' > $ 英日語彙出力先ファイル  ;\
done
# 学習（翻訳モデル作成）
SLAN=en; TLAN=de; GPU=-1;  EP=13 ;  \
MODEL= 翻訳モデル出力先ファイル  ; \
python -u ./LSTMEncDecAttn.py -V2 \
   -T                          train \
   --gpu-enc                   ${GPU} \
   --gpu-dec                   ${GPU} \
   --enc-vocab-file 英語語彙ファイル          \
   --dec-vocab-file 日本語語彙ファイル          \
   --enc-data-file 英語訓練ファイル          \
   --dec-data-file 日本語訓練ファイル          \
   --enc-devel-data-file 英語ディベロップファイル    \
   --dec-devel-data-file 日本語ディベロップファイル    \
   -D                              512 \
   -H                              512 \
   -N                              2 \
   --optimizer                     SGD \
   --lrate                         1.0 \
   --batch-size                    32 \
```

```
    --out-each                      0 \
    --epoch                         ${EP} \
    --eval-accuracy                 0 \
    --dropout-rate                  0.3 \
    --attention-mode                1 \
    --gradient-clipping             5 \
    --initializer-scale             0.1 \
    --initializer-type              uniform \
    --merge-encoder-fwbw            0 \
    --use-encoder-bos-eos           0 \
    --use-decoder-inputfeed         1 \
    -O                              ${MODEL} \
# 翻訳処理
SLAN=de; GPU=-1;  EP=13 ; BEAM=5 ;  \
MODEL= 翻訳モデル    ; \
python -u ./LSTMEncDecAttn.py \
    -T                      test \
    --gpu-enc               ${GPU} \
    --gpu-dec               ${GPU} \
    --enc-data-file 英語テストファイル      \
    --init-model            ${MODEL}.epoch${EP} \
    --setting               ${MODEL}.setting    \
    --beam-size             ${BEAM} \
    --max-length            150 \
    > 翻訳結果出力先ファイル
```

付録C

日本語文の空白削除

```
1  # -*- coding: utf-8 -*-
2  import codecs
3  import sys, io
4  sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding='utf-8')
5
6  # 空白処理を施すファイルおよび出力ファイル
7  # f = codecs.open('honyaku1.txt', 'r', 'utf-8')
8  # g = codecs.open('end1.txt', 'w', 'utf-8')
9  # f = codecs.open('honyaku2.txt', 'r', 'utf-8')
10 # g = codecs.open('end2.txt', 'w', 'utf-8')
11 f = codecs.open('honyaku3.txt', 'r', 'utf-8')
12 g = codecs.open('end3.txt', 'w', 'utf-8')
13 for row in f:
14     chars = list(row)
15     for ch in chars:
16         if ch is not '␣':
17             g.write(ch)
18 f.flush()
19 f.close()
20 g.flush()
21 g.close()
```

付録D

未知語の個数カウント

```
1  # -*- coding: utf-8 -*-
2  import codecs
3  import sys, io
4
5  sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding='utf-8')
6  #  未知語カウント処理を施すファイル
7  f = codecs.open('mecab1.txt', 'r', 'utf-8')
8  g = codecs.open('mecab2.txt', 'r', 'utf-8')
9  h = codecs.open('mecab3.txt', 'r', 'utf-8')
10
11 count = 0
12 num = 1
13 Flist = [f,g,h]
14 for n in Flist:
15     for l in n:
16         lsp = l.split()
17         for word in lsp:
18             if not word.find("UNK"):
19                 count += 1
20
21     print("パターン",num,":",count,"個")
22     count = 0
23     num += 1
24
25 f.flush()
26 f.close()
27 g.flush()
28 g.close()
29 h.flush()
30 h.close()
```