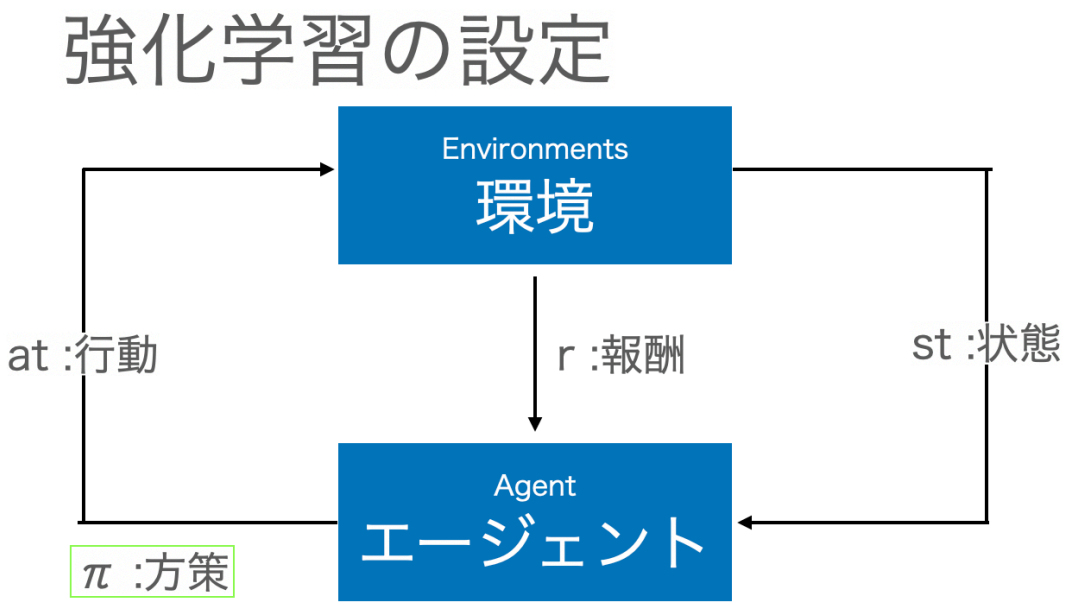
深層学習 day4

このレポートでは、深層学習 day1、day2、day3 と同じ用語と記法を用いる。

1　強化学習

強化学習は以下の図で表されるように、6 つの要素、**エージェント**、**環境**、**行動**、**状態**、**報酬**、**方策**からなる。



エージェントは例えば、ロボットや囲碁プログラムであり、これから行動を学習していく行動主体である。**環境**はエージェントが行動を加える対象であり、その行動に応じて、**状態**と**報酬**をエージェントに返す。次に、$t = 0, 1, 2, \ldots$ を時刻とする。時刻 t に環境の状態 $s(t)$ を観測したエージェントは、行動 $a(t)$ を選択して実行する。行動を受けた環境は状態を $s(t+1)$ へ変化させ、エージェントに報酬 $r(t+1)$ を返す。この一連流れが繰り返される。**方策** π はエージェントの行動を統制する。方策が決定的であるときは、$a(t) = \pi(s(t))$ であり、方策が確率的であるときは、$a(t) = \max_a \pi(a|s(t))$ である。次に、$\gamma \in (0, 1)$ とし、**利得** $R(t)$ を $R(t) = \sum_{k=0}^{\infty} \gamma^k r(t+k+1)$ で定義する（この値は必ず収束する）。**強化学習**とは、確率変数 $R(t)$ の期待値を最大にするような方策 π を見つけるためのアルゴリズムである。強化学習では、環境はマルコフ過程に従うと過程する。すなわち、時刻 t で状態 $s(t) = s$ であった環境が、エージェントが選択した行動 $a(t) = a$ に

よって次の状態 $s(t+1)=s'$ へ推移する確率は

$$P(s'|s,a)=P(s(t+1)=s'|s(t)=s,a(t)=a)$$

で与えられていると仮定する。次に、$R(s,a,s')=\sum_r rP(r|s',s,a)$ と定義し、

$$Q^\pi(s,a)=\sum_{s'}P(s'|s,a)R(s,a,s')+\gamma\sum_{s'}\sum_{a'}\pi(a'|s')P(s'|s,a)Q^\pi(s',a') \tag{1}$$

と定義する。(1) を**行動価値関数に対するベルマン方程式**という。強化学習の目的は、$Q^\pi(s,a)$ が全ての s,a に対して、最大となるような π を見つけることである。今、そのような π^* が見つかったとする。このとき、状態 s において、π^* が選択する行動 a は $a=\max_a Q^*(s,a)$ で与えられる。逆に考えれば、全ての s,a に対して、$Q(s,a)$ の値が最大となるような行動価値関数 $Q^*(s,a)$ を見つけることができれば、π^* が選択する行動 a は $a=\max_a Q^*(s,a)$ で与えられるということになる。**Sarsa 法**、Q **学習**は、行動価値関数 $Q(s,a)$ を更新していくことにより、そのような $Q^*(s,a)$ を見つけるアルゴリズムである。しかし、状況 s のとる値の個数が大きいときは、このような方法は現実的ではない。そのような場合には、行動を生成する方策を直接学習するという方法が用いられる。まず、方策 π をパラメーター θ による関数 $\pi(a|s)=\pi(a|s,\theta)$ としてモデル化する。時刻 0 での状態 s_0 から得られる長期間の報酬総和 $\rho(\pi)$ を

$$\rho(\pi)=E_{P,\pi}[\sum_{t=0}^{\infty}\gamma^t r(t+1)|s_0]$$

で定義する。この $\rho(\pi)$ を最大化するような $\pi(a|s,\theta^*)$ を見つけることを試みる。

$$\nabla_\theta\rho(\pi)\approx E_\pi[\nabla_\theta\log\pi(a|s,\theta)Q^\pi(s,a)]$$

が成り立つので、勾配上昇法

$$\theta^{(t+1)}=\theta^{(t)}+\eta E_\pi[\nabla_\theta\log\pi(a|s,\theta)Q^\pi(s,a)]$$

によって、$\pi(a|s,\theta^*)$ が求まる（η は学習率）。このような方法を**方策学習**という。

2 AlphaGo

AlphaGo は、深層学習に基づくコンピューター囲碁プログラムである。このプログラムでは、ニューラルネットによる教師あり学習と強化学習、そしてモンテカルロ木探索をうまく組み合わせることで、驚くほど高い能力を実現した。この章では、このプログラムについて紹介する。AlphaGo の学習過程は 3 つのステージで構成される。第 1 ステージでは、教師あり学習によって、囲碁の熟練者の手を σ をパラーメーターとするニューラルネットワーク $p_\sigma(a|s)$ に学習させる。$p_\sigma(a|s)$ は **SL 方策ネットワーク**と呼ばれる。このネットワークの入力 s は盤面の画像であり、出力 $p_\sigma(a|s)$ は差し手 a をとるべき確率である。第 2 ステージでは、第 1 ステージで最適化された $p_\sigma(a|s)$ のパラメーター σ を初期値にして、新たな ρ をパラメーターとするニューラルネット $p_\rho(a|s)$ を学習させる。$p_\rho(a|s)$ は **RL 方策ネットワーク**と呼ばれる。$p_\rho(a|s)$ の初期のパラメーターを σ とする。$p_\rho(a|s)$ の学習では、現在の方策とこれまでの方策からランダムに選択された方策どうしを戦わせる。パラメーターの更新には方策学習を用いる。第 3 ステージでは、盤面 s の画像に関する価値関数を θ をパラメーターとする価値ネットワーク $v_\theta(s)$ でモデル化する。パラメーターの更新には教師あり学習を用いる。学習後の実戦では、モンテカルロ木探索と最適化された $p_\rho(a|s),v_\theta(s)$ を用いて、与えらた盤面 s に対して、最適な差し手 a を選択する。

3　軽量化・高速化技術

この章では、深層学習モデルの軽量化と高速化について紹介する。

3.1　分散深層学習

深層学習モデルは学習の際、多くのデータを必要とし、調整するパラメーターの数も多いため、学習に時間がかかる。そこで、複数の計算機（ワーカー）を使用して、計算を並列化することによって、学習時間を短縮することを試みる。この手法を**分散学習**という。この節では、その方法を紹介する。

(1) データ並列化：親モデルを各ワーカーに子モデルとしてコピーし、データを分割して各ワーカーごとに計算させる。データ並列化は各モデルのパラメーターの合わせ方によって、同期型と非同期型に分かれる。同期型では、各ワーカーの計算が終わるのを待ち、全ワーカーの勾配が出たところでそれらの平均を計算し、親モデルのパラメーターを更新する。その後、その親モデルを各ワーカーに子モデルとしてコピーする。非同期型では、各ワーカーはそれぞれ別々に計算を行う。学習が終わった各子モデルはサーバーに置かれる。各ワーカーはサーバーに置かれた学習済みモデルを使って学習を行う。処理スピードは非同期型の方が早いが、非同期型は最新のモデルのパラメーターを利用できないので、学習が不安定になりやすい。現在は同期型の方が精度が良いことが多いので、主流となっている。
(2) モデル並列化：親モデルを各ワーカーに分割し、それぞれのモデルを学習させ、全てのデータで学習が終わった後で、一つのモデルに復元する。
(3) GPU による高速化：GPU は”Graphics Processing Unit”の略称であり、3D グラフィックスなどの画像描写を行う際に必要となる計算処理を行う半導体チップ (プロセッサ) のことである。GPU は並列計算に適しているため、GPU を使うことで、計算速度を高速化することができる。

3.2　モデル軽量化

深層学習モデルをメモリが少なく、演算処理の低いコンピューター（Iot、スマートフォン）で実行するには、モデルの精度を維持しつつ、パラメーターの数や演算回数を低減する手法が必要となる。これらの手法を**軽量化**という。この節では、いくつかの軽量化について紹介する。

(1) 量子化：大きなモデルは大量のパラメーターを持っているため、学習や推論に多くのメモリと演算処理が必要となる。そこで、パラメーターの 64bit 浮動小数点を 32bit などの下位の精度に落とすことで、メモリと演算処理の削減を行う。
(2) 蒸留：精度の高いモデルは大きなモデルであることが多く、大量のパラメーターを持っているため、学習や推論に多くのメモリと演算処理が必要となる。そこで、精度の高いモデルを用いて、軽量なモデルの学習を行う。具体的には、精度の高いモデルと軽量なモデルを結合し、精度の高いモデルのパラメーターを凍結した状態で、軽量なモデルのパラメーターを更新する。結果として、軽量なモデルは精度の高いモデルのパラメーターの情報を引き継ぐことができる。
(3) プルーニング：大きなモデルは大量のパラメーターを持っているが、必ずしもその全てのパラメーターが計算精度に貢献しているわけでない。そこで、計算精度に貢献しないパラメーターを削減すること

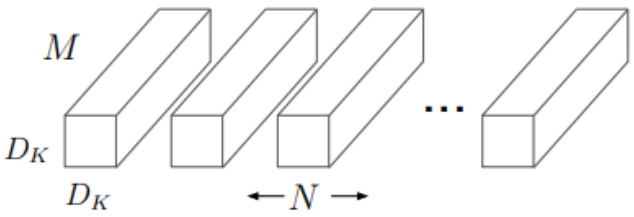
で、モデルを軽量化することができる。具体的には、ある閾値を設定し、値が閾値以下であるようなパラメーターを削減する。

4　応用モデル

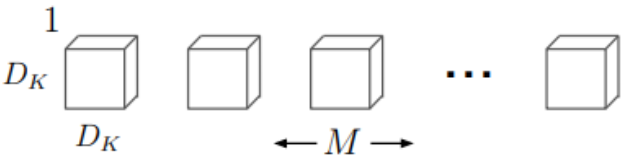
この章では、いくつかの応用モデルを紹介する。

4.1　MobileNet

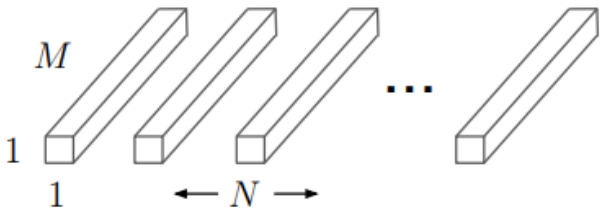
MobileNet は [5] で提案されたモデルである。MobileNet はモバイル機器における画像処理を行うための深層学習モデルであり、depthwise separable convolution を行うことによって、軽量化を図っている。depthwise separable convolution では、通常の convolution を depthwise convolution と、pointwise convolution と呼ばれる 1×1 convolution へ分解して行う。



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

ここでは、stride が 1 であり、パディングを行う convolution layer について考える。この convolution layer は、N 個の $D_K \times D_K \times M$（D_K は画像の縦横の幅、M はチャンネル数）のフィルターからなる kernel $\mathbf{K}$ を持ち、$D_F \times D_F \times M$ の画像データ $\mathbf{F}$（D_F は画像の縦横の幅、M はチャンネル数）に対して、$D_F \times D_F \times N$

の画像データ $\mathbf{G}$（N はチャンネル数）を出力する（図 (a) を参照)。このとき、

$$\mathbf{G}_{k,l,n} = \sum_{i,j,m} \mathbf{K}_{i,j,m,n} \cdot \mathbf{F}_{k+i-1,l+j-1,m} \tag{2}$$

が成り立つ。よって、convolution layer の計算コストは

$$D_K \times D_K \times M \times N \times D_F \times D_F \tag{3}$$

である。MobileNet は、depthwise separable convolution を行うことによって、この計算コストを削減する。depthwise separable convolution は depthwise convolution と pointwise convolution の 2 つの layer からなる。MobileNet では、それぞれの layer で batch normalizetion と ReLU 関数を用いている。depthwise convolution は 1 個の $D_K \times D_K \times M$（D_K は画像の縦横の幅、M はチャンネル数）のフィルターからなる kernel $\hat{\mathbf{K}}$ を持ち、画像データ

$$\hat{\mathbf{G}}_{k,l,m} = \sum_{i,j} \hat{\mathbf{K}}_{i,j,m} \cdot \mathbf{F}_{k+i-1,l+j-1,m} \tag{4}$$

を出力する（図 (b) を参照)。よって、depthwise convolution の計算コストは

$$D_K \times D_K \times M \times D_F \times D_F \tag{5}$$

である。一方、pointwise convolution は N 個の $1 \times 1 \times M$（M はチャンネル数）のフィルターからなる kernel を持ち、depthwise convolution の出力した画像データ $\hat{\mathbf{G}}$ に対して、$D_G \times D_G \times N$ の画像データ $\mathbf{G}$（D_G は画像の縦横の幅、N はチャンネル数）を出力する（図 (c) を参照)。よって、depthwise separable convolution の計算コストは

$$D_K \times D_K \times M \times D_F \times D_F + M \times N \times D_F \times D_F \tag{6}$$

である。従って、通常の convolution layer に比べて、depthwise separable convolution の計算コストははるかに小さい。この手法により、MobileNet は軽量化を実現している。

4.2 DenseNet

DenseNet は [6] で提案されたモデルである。CNN では、convolution layer を多く重ねると勾配消失が生じるという問題があった。DenseNet では、画像処理を行う深層学習モデルであり、dense block を用いることで、この問題を解決する。

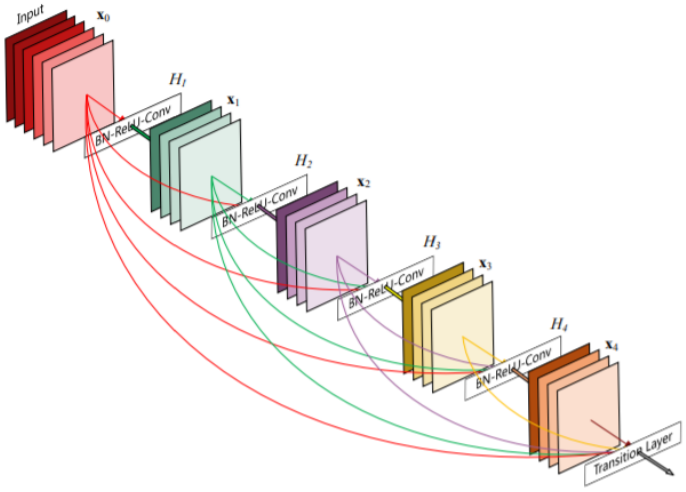
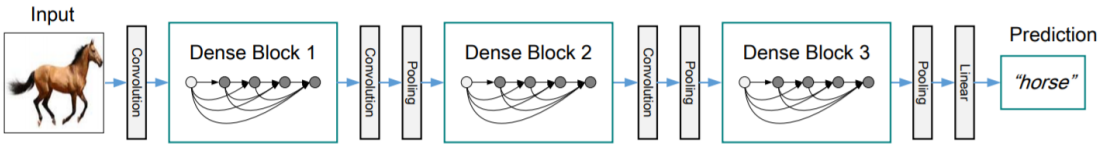


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

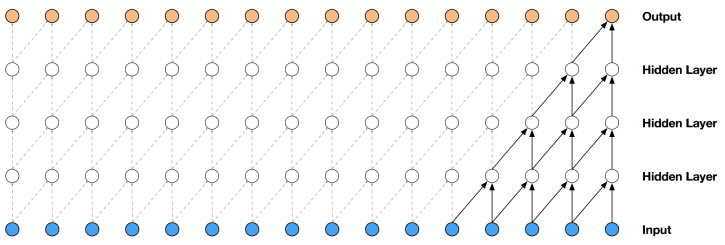
$\mathbf{x}_0$ を 1 枚の画像とし、この画像がある CNN に渡されるとする。この CNN が l 個の convolution layer $H_0(), \ldots, H_{l-1}()$ を持つと仮定する。ここで、$H_l()$ は、Batch Nomalization、ReLU、Pooling、Convolution などの合成であってもよい。l 番目の convolution layer の出力画像を x_l で表すことにする。dense block が導入された CNN では、$\mathbf{x}_l$ が $\mathbf{x}_l = H_l([\mathbf{x}_0, \ldots, \mathbf{x}_{l-1}])$（$[\mathbf{x}_0, \ldots, \mathbf{x}_{l-1}]$ は $\mathbf{x}_0, \ldots, \mathbf{x}_{l-1}$ のサイズを合わせて concatenate した画像）によって計算される（図を参照）。DenseNet の構造は



となっており、複数の dense block が convolution layer と pooling layer の間に挟まれている。

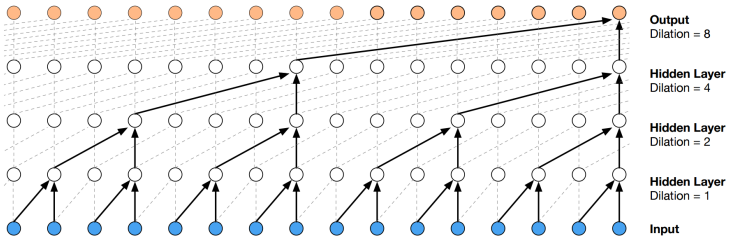
4.3 Wavenet

Wavenet は [9] で提案されたモデルである。Wavenet は生の音声波形を生成する深層学習モデルである。$\mathbf{x} = \{x_1, \ldots, x_T\}$ を波形とする。Wavenet の出力層は T 個の出力を持つ softmax 関数であり、



$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t | x_1, \ldots, x_{t-1}) \tag{7}$$

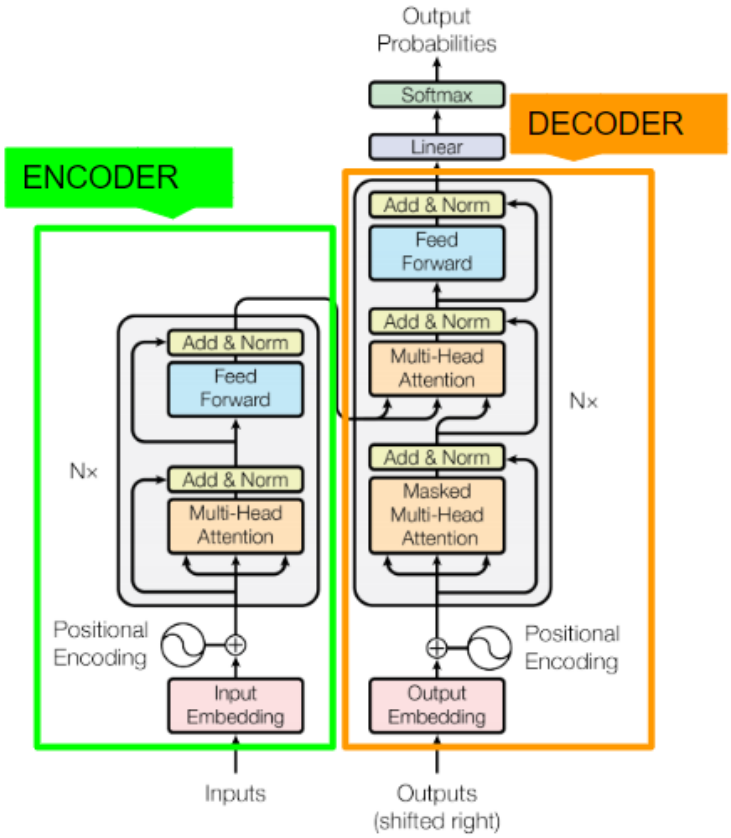
によって、波形 $x_1, \ldots, x_{T-1}$ から x_T を予測する（図を参照）。Wavenet の構造は PixelCNN の構造と類似しており、convolution layer を重ねることで、条件付き確率の確率分布をモデル化している。Wavenet の計算は通常の畳み込み計算（上図を参照）とは異なり、dilated convolution を用いる。dilated convolution では、ある間隔で入力を飛ばすことにより、フィルターが自身の長さよりも大きな領域に適用される（下図を参照）。



上図では、output は波形の後ろから 5 個文の受容野しか持たないが、下図では、output は波形の全体に対する受容野を持つ。これは、pooling や stride convolution と類似しているが、dilated convolution では、出力の大きさは変わらない。Wavenet では、dilated convolution を用いることにより、少ない層で出力層により広い受容野を持たせることができる。

5 Transformer

RNN は、前時刻に計算した結果を用いて、逐次的に計算を行う。そのため、RNN の計算を、時間方向で並列的に計算することはできない。従って、RNN は GPU を使った計算環境に適してない。このことから、RNN を取り除く研究が進められており、その中の 1 つのモデルが **Transformer** [14] である。



このモデルは、Encoder（左側）と Decoder（右側）からなる。Encoder は入力列 $(x_1, \ldots, x_n)$ をある表現 $\mathbf{z} = (z_1, \ldots, z_n)$ に変換する。Decoder は $\mathbf{z}$ を用いて、出力列 $(y_1, \ldots, y_m)$ を一つずつ時間ごとに出力する。Muti-Head Attention layer は複数の Scaled Dot-Product Attention layer からなっている。

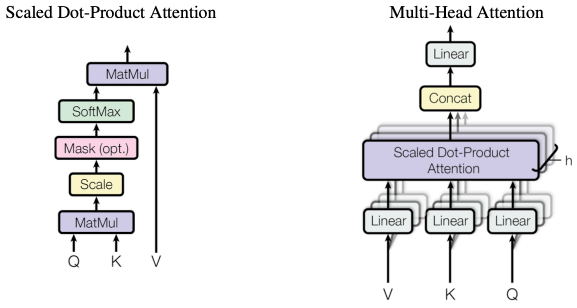


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Scaled Dot-Product Attention layer の入力は 2 つの d_k 次元ベクトル **query**、**key** と 1 つの d_v 次元ベクトル **value** である。実際には、複数の query、key、value を同時に処理したいので、それぞれをまとめて行列 Q, K, V をつくる。ここで、

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{8}$$

と定義する。次に、h を自然数、W_i^Q, W_i^K, W_i^V, W^O（$i = 1, \ldots, h$）を行列（パラメーター）とし、$\mathrm{head}_i = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$ とおき、

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h)W^O \tag{9}$$

と定義する。学習によって、W_i^Q, W_i^K, W_i^V, W^O の値は更新される。Muti-Head Attention layer は時系列データの中から、重要な部分だけを取り出すという操作を行なっており、学習によって、その精度が向上する。また、Transformer は RNN layer、CNN layer のいずれも持たないため、時系列の順番を利用するために、時系列のトークンに関連する位置情報を挿入しなければならない。これを行うのが、”positional encoding” である。Transformer では、2 つの positional encoding layer（左側と右側）が用いられている。ここでは左側の positional encoding layer について説明する。今、文章 $a_1, \ldots, a_T$ があるとする。Embedding layer によって、各 a_t は d_{model} 次元ベクトル x_t に変換されるとする。$x_t = (x_{t1}, \ldots, x_{td_{\mathrm{model}}})$ とおく。このとき、positional encoding layer は各 x_{ti} を $x_{ti} + \sin(t/10000^{2i/d_{\mathrm{model}}})$ に変える。右側の positional encoding layer では、左側と同様にして、x_{ti} を $x_{ti} + \cos(t/10000^{2i/d_{\mathrm{model}}})$ に変えている。この操作を行うことにより、元の文章における位置情報がベクトルに挿入される。

例 5.1. (コードは Transformer.ipynb) Transformer の最初の layer を実装する。

```
import numpy as np
import tensorflow as tf
import keras

class PositionalEncoding(keras.layers.Layer):
    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        if max_dims % 2 == 1: max_dims += 1 # max_dims must be even
        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))
        pos_emb = np.empty((1, max_steps, max_dims))
        pos_emb[0, :, ::2] = np.sin(p / 10000**(2 * i / max_dims)).T
        pos_emb[0, :, 1::2] = np.cos(p / 10000**(2 * i / max_dims)).T
        self.positional_embedding = tf.constant(pos_emb.astype(self.dtype))
    def call(self, inputs):
        shape = tf.shape(inputs)
        return inputs + self.positional_embedding[:, :shape[-2], :shape[-1]]

embed_size = 512; max_steps = 500; vocab_size = 10000
encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)
positional_encoding = PositionalEncoding(max_steps, max_dims=embed_size)
```

```
encoder_in = positional_encoding(encoder_embeddings)
decoder_in = positional_encoding(decoder_embeddings)
```

6　物体検知・セグメンテーション

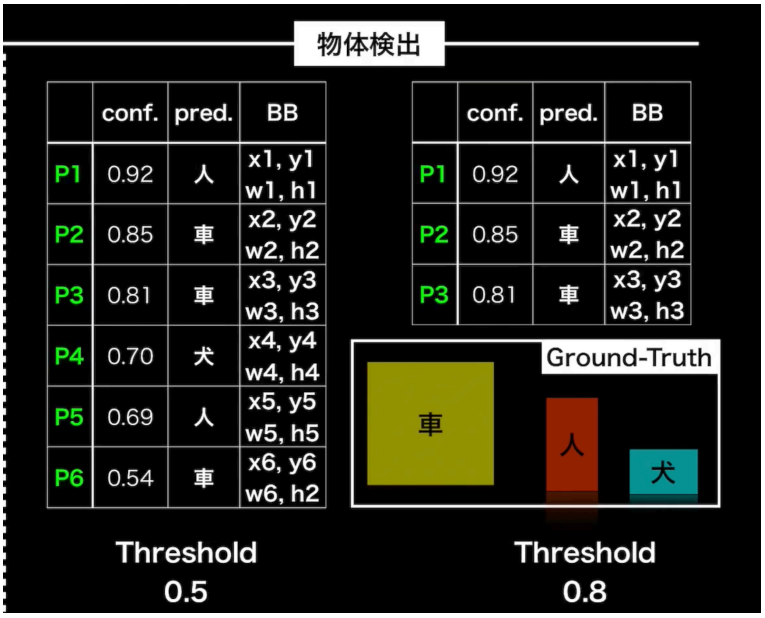
この章では、物体検知とセグメンテーションについて述べる。

6.1　物体検知

物体検知とは、ある画像中における物体の位置と、その物体に対応するクラスを予測するタスクである。物体検知における教師データには対象となる物体が含まれる短形領域を示すバウンディングボックスとそれに対応するクラスを用意する必要がある。このような教師データを用いて、物体検出モデルを作成したあと実際に予測をさせると、同一物体に対して複数のバウンディングボックスが検出されることがある。これを防ぐために、各バウンディングボックスに対応する信頼度スコアが最も高いものだけを採用する。これを非最大値抑制と呼ぶ。非最大値抑制によって決定されたバウンディングボックスと教師データのバウンディングボックスとの一致度を計る指標として、IoU（Intersection over Union）がある。真のバウンディングボックスで囲まれた領域を B_{true}、予測されたバウンディングボックスで囲まれた領域を B_{pred} としたとき、IoU は

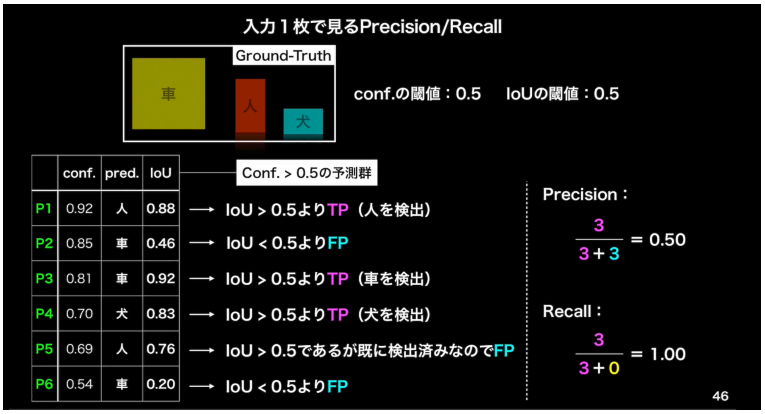
$$\text{IoU}(B_{\text{true}}, B_{\text{pred}}) = \frac{|B_{\text{true}} \cap B_{\text{pred}}|}{|B_{\text{true}} \cup B_{\text{pred}}|} \tag{10}$$

で定義される。学習の際はある閾値 t_1, t_2（threshold）を設定する。例えば、ある一枚の画像中にあるクラスに属する物体が 3 つ含まれているケースを考える。ここで、モデルは複数のバウンディングボックスとそれが対応するクラスを予測する。信頼度スコア（confidence）が t_1 以上であるようなバウンディングボックスとそれが対応するクラスを信頼度スコアの値が高い順に列挙する。

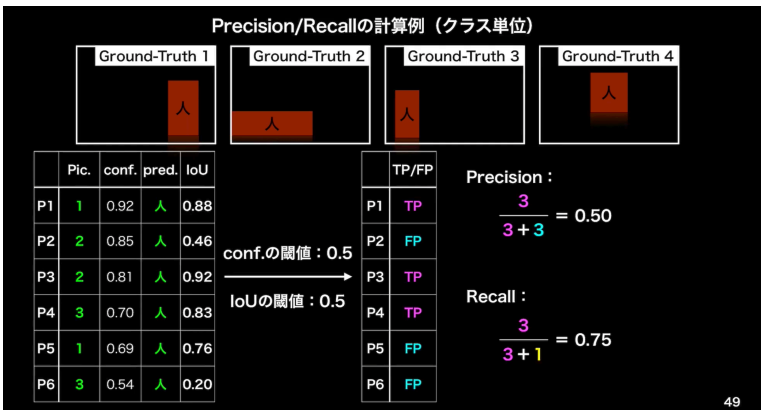


次に、IoU の値が t_2 以上であるようなバウンディングボックスとそれが対応するクラスを信頼度スコアの値

が高い順に列挙する。



その後、confidence と IoU を用いて、それぞれの予測データが TP であるか、FP であるかを判定し、Precision と Recall を計算する。次は同様なことを 1 つのクラス C について、全ての画像で行う。



上記の Precision と Recall は IoU の閾値 t_2 を 0.5 に固定し、confidence t_1 を変えれば、それに応じて変化する。従って、Precision と Recall は t_1 を変数とする関数である。すなわち、Precision $= P(t_1)$, Recall $= R(t_1)$ である。従って、P は R の関数であり、$P = f(R)$ と表せる。ここで、クラス C についての Average Precision（AP(C)）を

$$\mathrm{AP}(C) = \int_0^1 P(R)dR \tag{11}$$

で定義する。そして、クラスの総数が N であり、$C_1, \ldots, C_N$ を全てのクラスとするとき、meaned Average Precision（mAP）を

$$\mathrm{mAP} = \frac{1}{N}\sum_{i=1}^{N} \mathrm{AP}(C_i) \tag{12}$$

で定義する。この値を最大化するようにモデルのパラメーターは更新される。

6.2 セグメンテーション

セグメンテーションとは、画像の各ピクセルごとにクラス分類を行うことで、画像中に含まれる複数の物体に対し、それぞれのピクセルが存在する領域に対応するクラスを予測するタスクである。セグメンテーション

では、入力画像と出力画像のサイズが同じでなければならない。一方、通常の CNN では、層を経るごとに画像のサイズが小さくなるため、Deconvolution や Transposed convolution といった Up-sampling という手法が用いられる。**FCN** というモデルでは、プーリングを経てサイズが小さくなった特徴マップに Transposed convolution を適用することで、元の画像のサイズの大きさまで拡大する。その際、位置情報としてプーリングで得られた特徴マップを各画素ごとに足し合わせることで、局所的な情報と対局的な情報の両方を加味することができる。**SegNet** というモデルは、エンコーダーとデコーダーにより構成された、自己符号化器型の CNN である。SegNet では、エンコーダー側の最大値プーリングで取得した値の場所を記録しておき、その情報をデコーダー側の Up-sampling において利用する。**Unet** はというモデルは SegNet と同じく、自己符号化器型の CNN である。Unet では、エンコーダーで作成した特徴マップをデコーダーに伝達する。デコーダーはエンコーダーから渡された特徴マップと前層から渡されたチャンネル方向に結合し、Transposed convolution を適用することで、元の画像のサイズの大きさまで拡大する。

参考文献

[1] Andriy Burkov. (2019). The hundred-page machine learning book.

[2] Francois Chollet. (2018). Deep learning with python. Manning Publications Co.

[3] Marc Peter Deisenroth., A. Aldo Faisal., Cheng Soon Ong. (2020). Mathematics for machine learning. Cambridge University Press.

[4] Aurëlien Gëron. (2019). Hands-on machine learning with Scikit-Learn, Keras & TensorFlow. 2nd Edition. Oreilly.

[5] Andrew G. Howard., Menglong Zhu., Bo Chen., Dmitry Kalenichenko., Weijun Wang., Tobias Weyand., Marco Andreetto., Hartwig Adam. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv.

[6] Gao Huang., Zhuang Liu., Laurens van der Maaten., Kilian Q. Weinberger. (2017). Densely Connected Convolutional Networks. arXiv.

[7] 小縣信也., 斎藤翔汰., 溝口聡., 若杉一幸. (2021). ディープラーニング E 資格エンジニア問題集. インプレス.

[8] 岡谷貴之. (2015). 深層学習. 講談社.

[9] Aaron van den Oord., Sander Dieleman., Heiga Zen., Karen Simonyan., Oriol Vinyals., Alex Graves., Nal Kalchbrenner., Andrew Senior., Koray Kavukcuoglu. (2016). WaveNet: A Generative Model for Raw Audio. arXiv.

[10] Sebastian Raschka., Vahid Mirjalili. (2019). Python machine learning. Third Edition. Packt.

[11] 斎藤康毅. (2016). ゼロから作る deep learning. オライリージャパン.

[12] 斎藤康毅. (2018). ゼロから作る deep learning 2. オライリージャパン.

[13] 瀧雅人. (2017). これならわかる深層学習. 講談社.

[14] Ashish Vaswani., Noam Shazeer., Niki Parmar., Jakob Uszkoreit., Llion Jones., Aidan N. Gomez., Lukasz Kaiser., Illia Polosukhin. (2017). Attention Is All You Need. arXiv.