101. 深層学習の適用方法 (画像認識)

秋葉洋哉

2024 年 7 月 15 日

1 ResNet(転移学習)

1.1 概要

学習用のデータが少ない場合に、ImageNet と呼ばれるデータセットを用いて作成された汎用的な画像識別モデルをベースとして、そのモデルに特定のドメインに特化したデータを与えて、学習を行わせることで、簡潔に精度の高い画像識別モデルを作成することができる。この手法を転移学習と呼ぶ。

画像の識別モデルの事前学習として用いられる ImageNet は 1400 万枚以上の画像が収録されており、1000 クラス以上のカテゴリに分類されている。ImageNet には、犬や猫、車、飛行機などの画像が収録されており、汎用的な画像識別モデルでは、それらの画像を識別することができる。ImageNet で学習したモデルでは、ResNet50,101,152 といったモデルが存在する。

ドメインシフト

訓練データとテストデータの分布が異なってしまい、学習したモデルがうまく予測できない現象をドメインシフトと呼ぶ。ドメインシフトの原因は大きく分けて 4 つに分類できる。

1. 時間的変動 (夏にソフトクリームが良く売れるなど時間によって予測と異なる)
2. 地域的変動 (大阪でたこ焼きが多く売れるなど地域差によって予測と異なる)
3. 機器的変動 (搭載機器の性能によって予測と異なる)
4. 操作的変動 (人それぞれの操作方法や習慣によって予測と異なる)

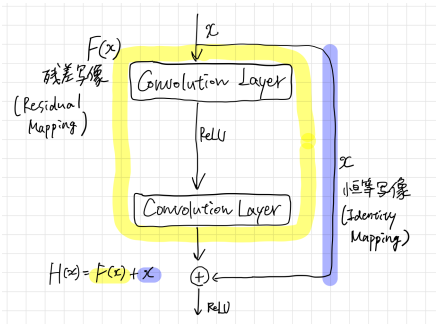
これらのドメインシフトを解消するためには

1. データの再収集
2. データの拡張 (画像を回転、反転、色調変更)
3. 転移学習 (大量のデータで訓練された学習したモデルを新しいタスクに転移)
4. ドメイン適応 (訓練データとテストデータのドメインの違いを数値にしてモデルを学習)
5. 不変表現学習 (異なるドメインのデータでも共通の特徴表現を学習)

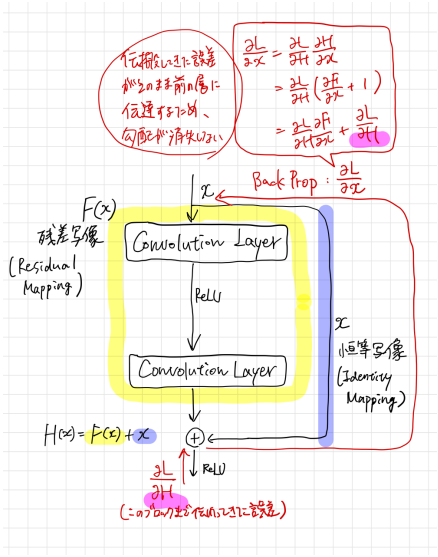
といった対策が考えられる。ドメインシフトは実際のビジネスでの機械学習の適用において重要な問題であるため、適切な対策を講じることが求められる。

1.2 ResNet

ResNet は、2015 年にかつての Microsoft Research に所属していた Kaiming He が考案したモデルであり、ImageNet での画像識別コンペティションで優勝したモデルである (図 1a) 。ResNet は、畳み込み層を 152 層も積み重ねたモデルであり、SkipConnection という仕組みを導入し、「元の入力」と「Residual Block の出力」の二つを「Identity Mapping」と呼ばれる組み合わせ手法で結合することで、層が深くなるにつれて発生する勾配消失問題を回避している (図 1b)。



(a) 仕組み: 学習部分は $F(x)$ の残差部分、残差部分のブロックを ResidualBlock と呼ぶ。



(b) 勾配消失問題が回避できる理由： $F(x)$ の勾配が消失しても、x の勾配が残るため、勾配消失問題を回避できる。

図 1: Skip Connection について

しかし、ResNet では、いくつかのブロックだけが有用な表現を学習し、他のブロックは学習が進まないという問題があった。これは、Identity Mapping には、誤差逆伝播時に勾配が Residual Block を強制的に経由させる仕組みが無いため、何も学習せずスキップしてしまうということが起こるためである。

この問題に対して、層を増やしてパラメータを少なくするために、できるだけ 1 層あたりの内部構造を薄くしようとした結果、BottleNeck アーキテクチャという仕組みが導入された。BottleNeck アーキテクチャ (図 2) は、畳み込み層の前後に 1x1 の畳み込み層を挟むことで、計算量を削減している。

また、無意味な層を減らすという目的のため、Residual Block 内の畳み込み層のチャネル方向を広げる、という方針で、WideResNet というモデルが考案された。

1.3 WideResNet

ResNet を工夫した、WideResNet は、ResNet の畳み込み層の層数を減らし、フィルタ数を k 倍に増やして GPU の特性に合わせることで、高速かつ高精度の学習を可能にしたモデルである。また、残差ブロック内

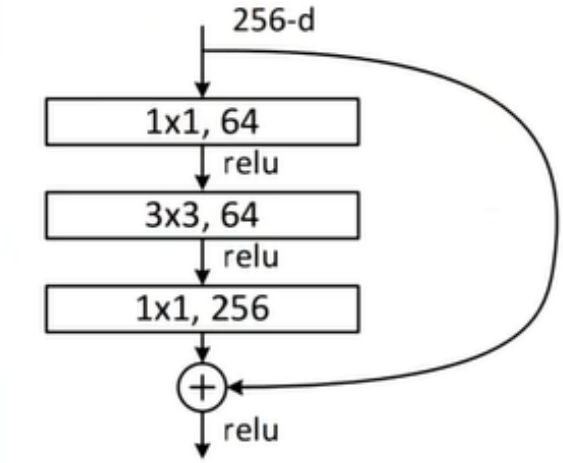


図 2: BottleNeck アーキテクチャの仕組み

l	CIFAR-10
1	6.69
2	5.43
3	5.65
4	5.93

Table 3: Test error (%, median over 5 runs) on CIFAR-10 of WRN-40-2 (2.2M) with various l.

図 3: WideResNet 内の Residual Block の層の数に対するエラー率の遷移: Residual Block 内では、l=1 では表現力が足りず、l>=3 の時は残差接続が減少して最適化が難しくなるため、l=2 が最適という研究結果になっている。

に Dropout を導入することで、過学習を抑制している。

まず、Residual Block の表現力を向上させる方法は、以下の 3 つが考えられる。

1. ブロックにより多くの畳み込み層を加える
2. より多くの特徴平面を加えることで畳み込み層を広げる
3. 畳み込み層のフィルタサイズを大きくする

このうち、1, 2 を実現しようとしたのが、WideResNet である。WideResNet の表記「WRN-n-k」は、n 層の畳み込みを持ち、幅 (チャネル数) を k 倍することを意味する。パラメータ数と計算量は k の二乗になる。$k = 1$ のときは元の ResNet と同一で、$k > 1$ の時に WideResNet となる。ResNet-1001(パラメータ数:10.2×10^6) よりも WRN-40-4(パラメータ数:8.9×10^6) の方がパラメータ数が大幅に削減され、8 倍高速になった。

1.4 ファインチューニング

ファインチューニングとは、転移学習の一種であり、事前学習モデルを用意し、バッチ正規化部分のパラメータだけを固定して、追加したタスクに加えて、事前学習モデル内も含めて、すべてのパラメータを再学習させる手法である。Tensorflow Hub の ResNet は、trainable を True に変更するだけで、ファインチューニングを行い、事前学習モデルの特徴を引き継ぎつつ、新しいタスクに適応した高精度モデルを簡単に作成することができる。

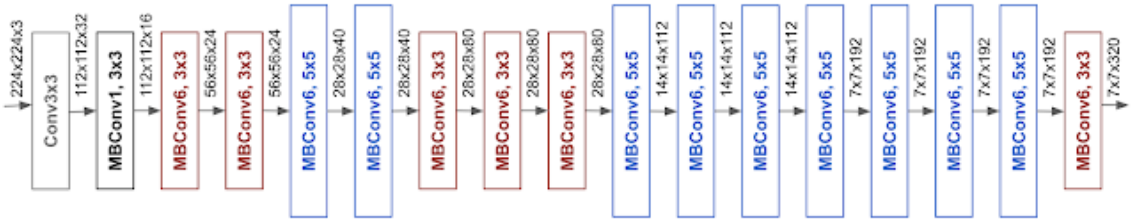


図 4: EfficientNet の構造

2 EfficientNet

2.1 概要

AlexNet 以降は、CNN モデルを大規模にスケールアップすることで精度を改善するアプローチがトレンドだった (ResNet-18 から ResNet-200 まである)。スケールアップの変数は、幅 (1 層あたりのノード数)、深さ (層数)、解像度 (入力画像の解像度) の 3 つである。

1. 深さ (d) : 表現力が高くなり、複雑な特徴表現を獲得できるようになる。
2. 幅 (w) : 細かい特徴表現を獲得し、学習を高速化できるようになる。
3. 解像度 (r) : 画像の特徴をより詳細に捉えることができるようになる。

これらの変数のスケールアップにより、精度の向上が期待できたが、ある程度まで増やすと精度の向上は横ばいとなってしまう上に、演算量 (FLOPS) は、$d \cdot w^2 \cdot r^2$ で増加するため、モデルのサイズが大きくなると、計算リソースが不足する問題があった。

2019 年に Google が提案した EfficientNet は、効率的なスケールアップの規則を採用することで、計算量を抑えつつ、精度を向上させることに成功したモデルである。

その規則とは、Compound Coefficent(複合係数) と呼ばれる 1 つの係数 ϕ で説明され、その係数を用いて一様にスケーリング可能になる。EfficientNet では d, w, r が以下で定義される。

$$\text{depth} : d = \alpha^\phi \tag{1}$$
$$\text{width} : w = \beta^\phi \tag{2}$$
$$\text{resolution} : r = \gamma^\phi \tag{3}$$
$$\text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \tag{4}$$
$$\alpha \geq 1, \beta \geq 1, \gamma \geq 1 \tag{5}$$

α, β, γ はハイパーパラメータであり、それぞれは、(4) の制約条件を満たすようにグリッドサーチで設定されるため、FLOPS は $\sim (\alpha \cdot \beta^2 \cdot \gamma^2)^\phi \approx 2^\phi$ と表され、計算リソースに対するモデルのスケールの見積もりがしやすくなる。例えば、$\phi = 1$ のときの最適値、$\alpha = 1.2, \beta = 1.1, \gamma = 1.15$ は、EfficientNet-B0 と呼ばれるモデルであり、FLOPS$\sim$ 0.35G、パラメータ数 $\sim$ 5M である。EfficientNet-B0 のアーキテクチャは、図 4 のように表されれ、EfficientNet は、EfficientNet-B0 から EfficientNet-B7 までの 8 つのモデルが提案されている。

2.2 Compound Scaling Method

EfficientNet の Compound Scaling Method は、モデルのスケールアップを効率的に行うための方法であり、以下の数式で表される

$$\max_{\alpha,\beta,\gamma} \text{Accuracy}(\mathcal{N}(d,w,r)) \tag{6}$$

$$\text{s.t. } \mathcal{N}(d,w,r) = \bigodot_{i=1\cdots s} \hat{\mathcal{F}}_i^{d\cdot\hat{L}_i}(X_{\langle r\cdot\hat{H}_i, r\cdot\hat{W}_i, w\cdot\hat{C}_i\rangle}) \tag{7}$$

$$\text{Memory}(\mathcal{N}(d,w,r)) \leq \text{target memory} \tag{8}$$

$$\text{FLOPS}(\mathcal{N}(d,w,r)) \leq \text{target FLOPS} \tag{9}$$

ここで、$\mathcal{N}(d,w,r)$ は、深さ d, 幅 d, 解像度 r のネットワークを表し、$\hat{\mathcal{F}}^{L_i}(X_{\langle \hat{H}_i,\hat{W}_i,\hat{C}_i\rangle})$ は L_i 回繰り返す畳み込み層を表し、入力時の高さ $\hat{H}_i$、幅 $\hat{W}_i$、チャンネル数 $\hat{C}_i$ を引数で取る。

まず、式 (6) は、モデルの精度を最大化するための目的関数である。

式 (7) は、モデルのアーキテクチャを表す関数を表し、$\odot_{i=1\cdots s}$ は、モデル $\mathcal{N}$ が s 個のブロックから構成されていることを示す。F は畳み込み層を表し、L はブロックの中で何度 F を繰り返すかを表す。

式 (8) と、式 (9) は、モデルのメモリと FLOPS が目標値以下であることを示す制約条件である。

これらの式を解くことで、最適な α, β, γ を求めることができる。

3 Vision Transformer(ViT)

3.1 概要

Vision Transformer(ViT) は、2020 年に Google が提案した、画像認識のための Transformer ベースのモデルである。ViT は、画像をパッチと呼ばれる小さな領域に分割し、それぞれのパッチを系列 (Sequence) データとして Transformer に入力することで、画像認識を行う。ViT は、CNN ベースのモデルと比較して、計算効率が高く、畳み込み層を用いないため、モデルの拡張が容易であるという特徴がある。ViT のアーキテクチャを図 5 に示す。

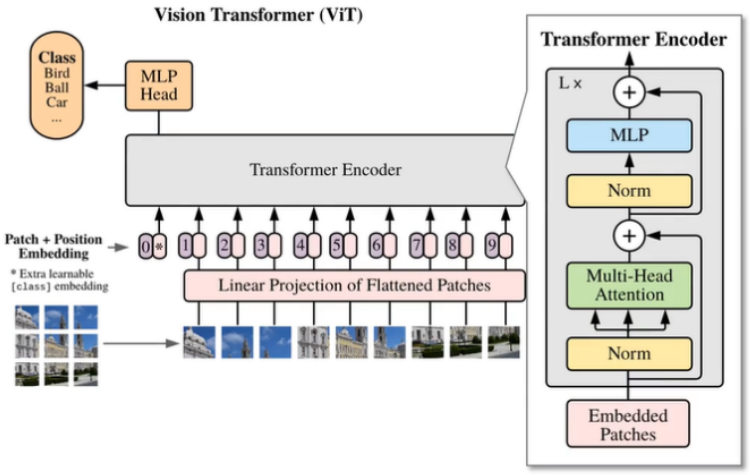


図 5: Vision Transformer のアーキテクチャ

ViT の処理は、以下の手順で行われる。

1. 画像をパッチに分割する
2. パッチ事に Flatten 処理を行い、トークン系列を得る。
3. Embedding 表現 (埋め込み表現) に変換する。
4. CLS トークンを系列データの最初に負荷する。
5. Positional Encoding を付加する。
6. Transformer に入力する。
7. 出力を取り出し、分類を行う。

ここで、Flatten 処理は、画像のパッチを 1 次元のベクトルに変換する処理である。また、Embedding 表現は、トークンをベクトル表現に変換する処理であり、Inductive bias と Hybrid Architecture という 2 つの表現方法がある。CLS トークンは、系列データの最初に配置され、系列データ全体の特徴を表すトークンである。

ViT の計算過程は、数式を用いて以下のように表される。

$$\text{Input} : x \in \mathbb{R}^{H \times W \times C} \tag{10}$$

$$\text{Patch Embedding} : \mathbb{X}_p \in \mathbb{R}^{N \times (C \times P^2)} \tag{11}$$

$$\left(\text{s.t. } N = \frac{H \times W}{P^2}\right) \tag{12}$$

$$\text{Class Token} : \mathbb{X}_{cls} \in \mathbb{R}^{1 \times (C \times P^2)} \tag{13}$$

$$\text{Encoder Input} : \mathbb{Z}_0 = \left[X_{\text{cls}}; X_p^1\mathbb{E}; X_p^2\mathbb{E}; \cdots ; X_p^N\mathbb{E}\right] + \mathbb{E}_{pos} \tag{14}$$

$$\left(\text{s.t. } \mathbb{E} \in \mathbb{R}^{(C \times P^2) \times D}, \mathbb{E}_{pos} \in \mathbb{R}^{(N+1) \times D}\right) \tag{15}$$

$$\text{Transformer(l layer)} : \mathbb{Z'}_l = \text{MSA}(\text{LN}(\mathbb{Z}_{l-1})) + \mathbb{Z}_{l-1} \tag{16}$$

$$\text{Transformer(Output)} : \mathbb{Z}_l = \text{MLP}(\text{LN}(\mathbb{Z'}_l)) + \mathbb{Z'}_l \tag{17}$$

$$\text{Encoder Output} : \mathbb{Y} = \text{LN}(\mathbb{Z}_l^0) \tag{18}$$

MSA は、Multi-Head Self-Attention を表し、MLP は、Multi-Layer Perceptron を表す。また、LN は、Layer Norm を表す。Encoder Output は D 次元の特徴量出力、MLP Head に入力され、最終的な出力が得られる。MLP Head では D 次元を K クラスに変換する。ファインチューニング時には、MLP Head のみを学習させることで、新しいタスクに適応したモデルを作成することができる。また、Position Embeding を変更することで、事前学習よりも高解像度の画像に対応したモデルを作成することができる。

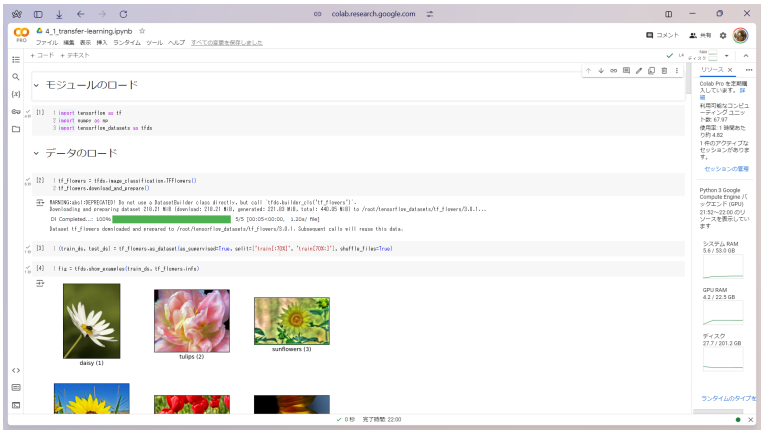
ViT は、計算量が増えるほど精度が向上するという特徴がある。また、ViT は、CNN ベースのモデルと比較して、計算効率が高く、畳み込み層を用いないため、モデルの拡張が容易であるという特徴がある。

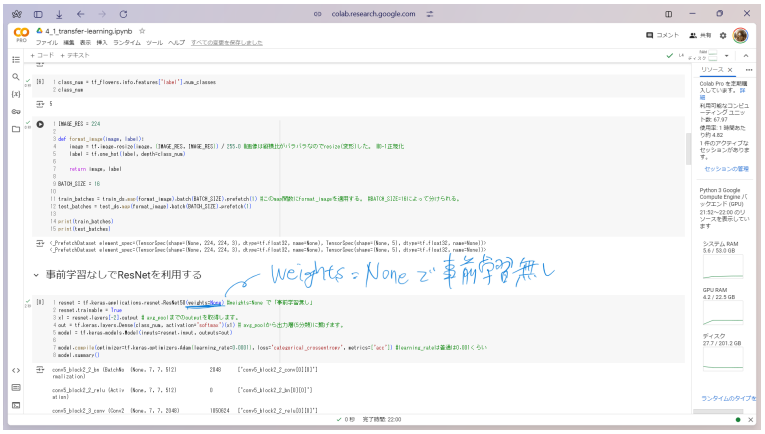
■参考文献

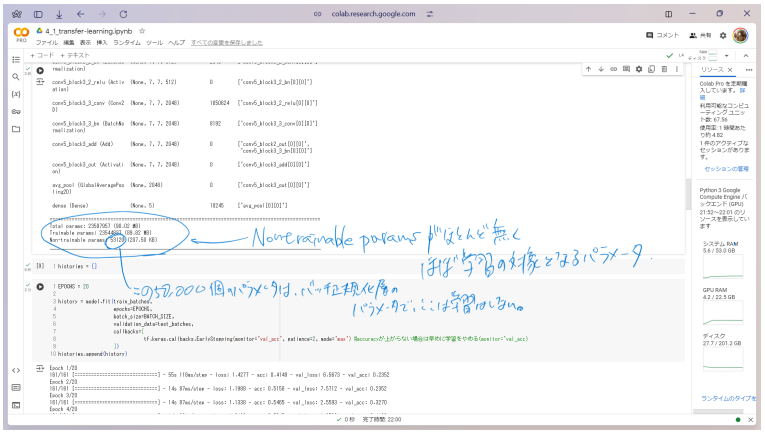
1. スキップ接続 (Skip connection) CVML エキスパートガイド https://cvml-expertguide.net/terms/dl/inter-layer-connection/skip-connection/
2. 【初心者にも分かりやすく】画像認識モデルの ResNet の要点を解説 https://dx-consultant-fast-evolving.com/resnet/
3. 【E 資格まとめ】ResNet と WideResNet https://tt-tsukumochi.com/archives/8991
4. ResNet の改良モデル WideResNet を詳細解説! https://deepsquare.jp/2021/08/wideresnet/
5. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks , Tan and Le 2019 https://arxiv.org/abs/1905.11946
6. EfficientNet: Improving Accuracy and Efficiency through AutoML and Model Scaling https://research.google/blog/efficientnet-improving-accuracy-and-efficiency-through-automl-and-model-s

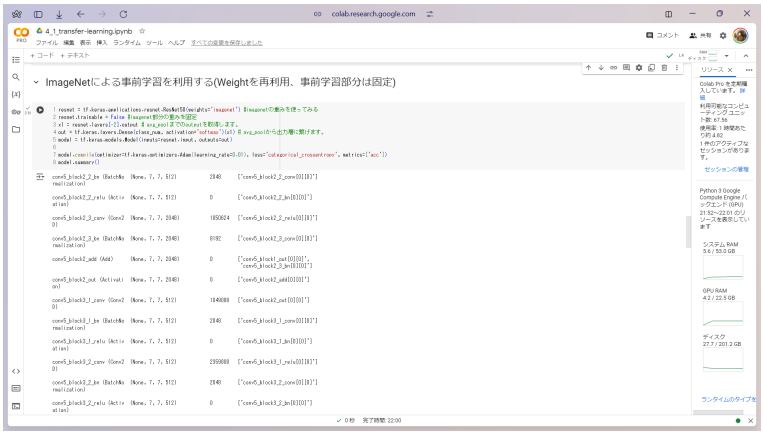
4　実装演習キャプチャ

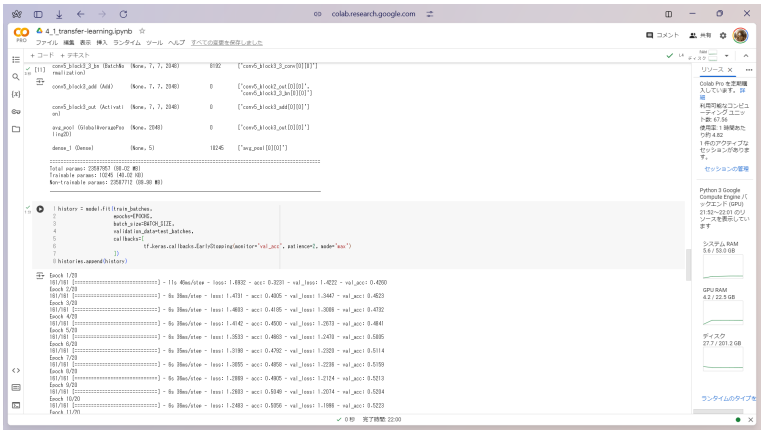
4.1　ResNet

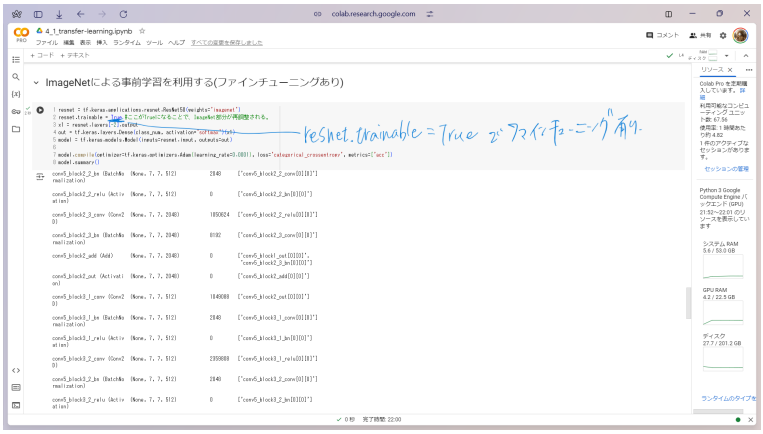


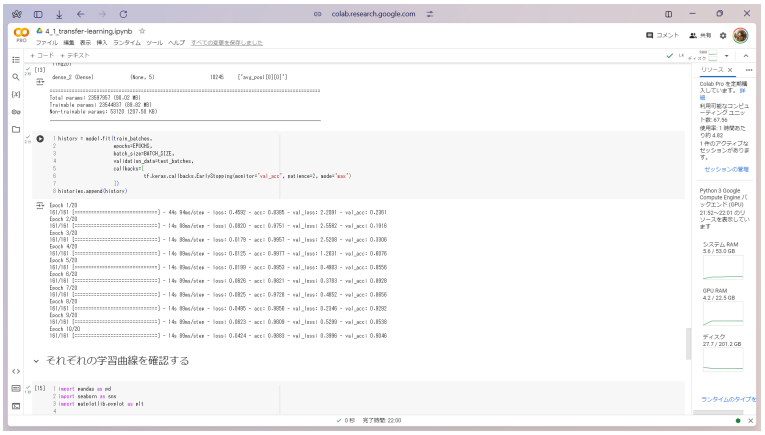


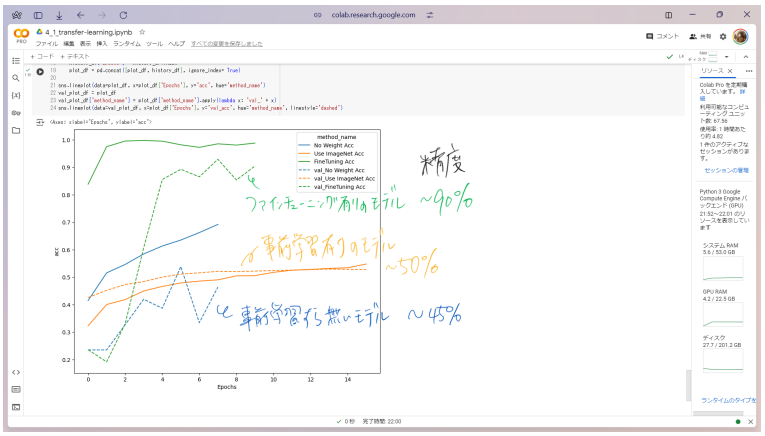



4_1_transfer-learning.ipynb
ImageNetによる事前学習を利用する(Weightを再利用、事前学習部分は固定)

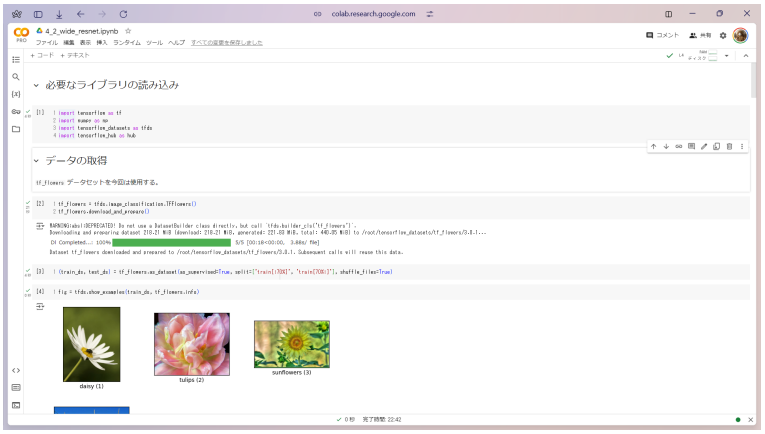

4_1_transfer-learning.ipynb

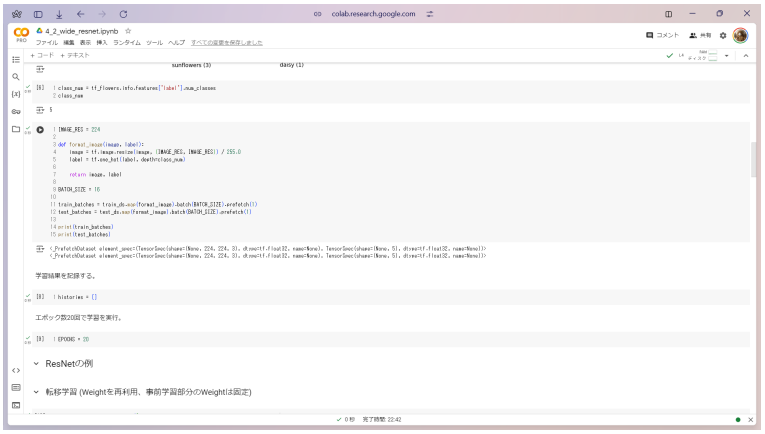

4_1_transfer-learning.ipynb
ImageNetによる事前学習を利用する(ファインチューニングあり)
resnet.trainable = True でファインチューニング有り

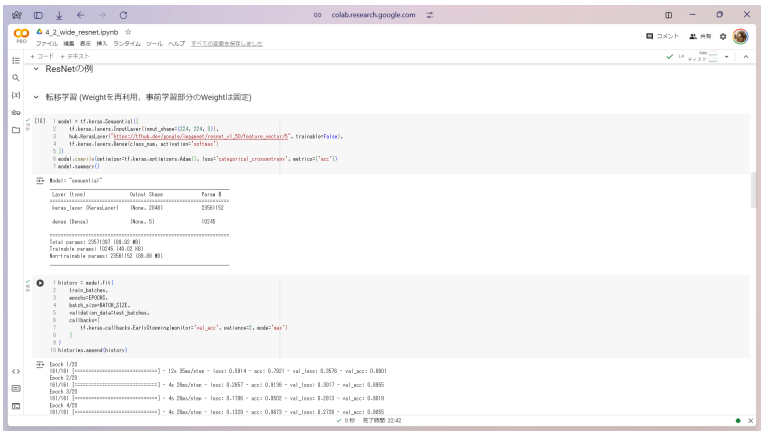

4_1_transfer-learning.ipynb
それぞれの学習曲線を確認する

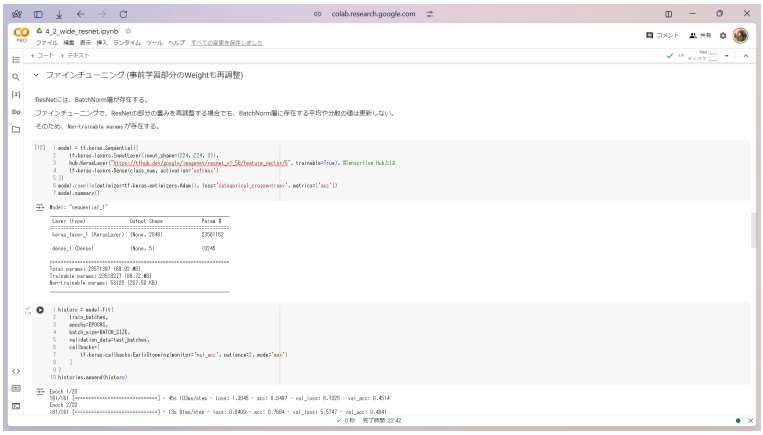

4_1_transfer-learning.ipynb
method_name
No Weight Acc
Use ImageNet Acc
FineTuning Acc
val_No Weight Acc
val_Use ImageNet Acc
val_FineTuning Acc
epochs
acc
精度
ファインチューニングが有りのモデル ~90%
転移学習有りのモデル ~50%
転移学習なし無いモデル ~45%

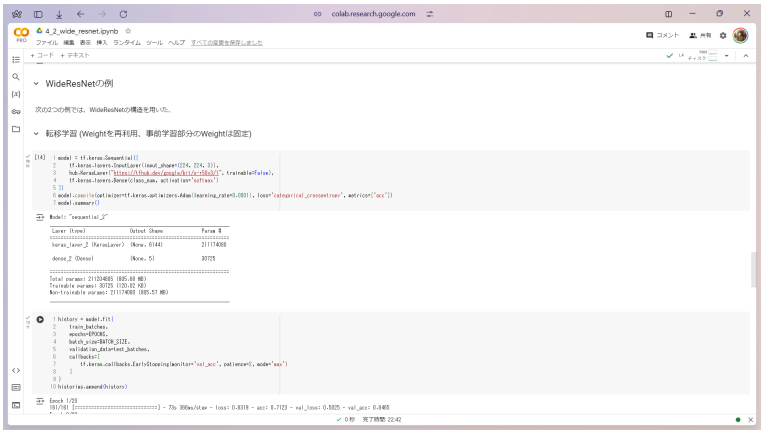
4.2 WideResNet

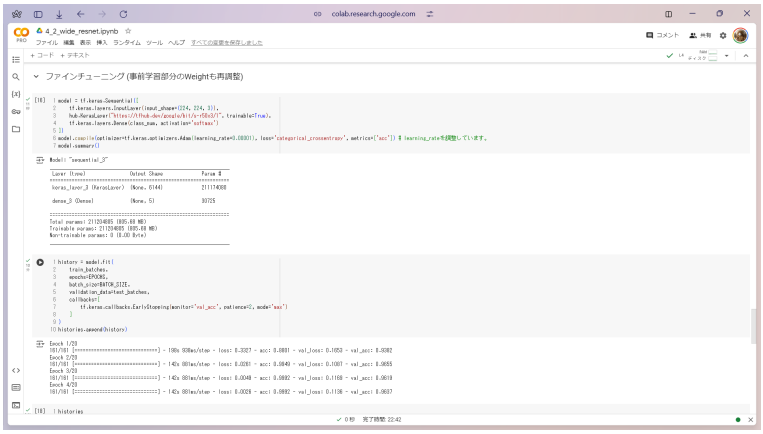


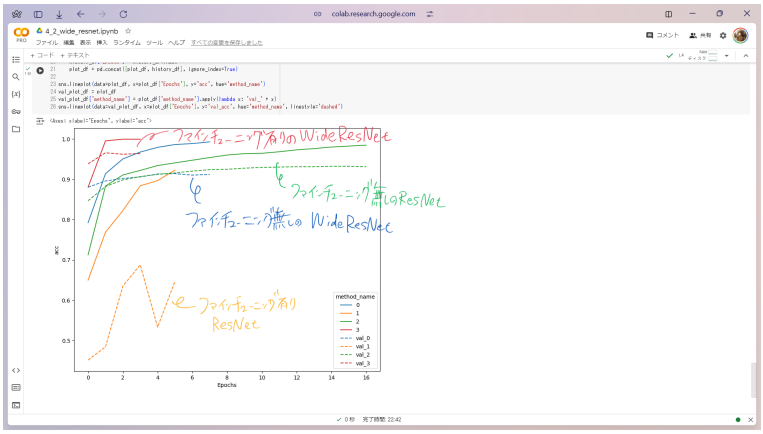





colab.research.google.com
4_2_wide_resnet.ipynb
ファイル 編集 表示 挿入 ランタイム ツール ヘルプ すべての変更を保存しました
コメント 共有
+ コード + テキスト
ファインチューニング (事前学習部分のWeightも再調整)
ResNetには、BatchNorm層が存在する。
ファインチューニングで、ResNetの部分の重みを再調整する場合でも、BatchNorm層に存在する平均や分散の値は更新しない。
そのため、Non-trainable varsが存在する。
[12] ! model = tf.keras.Sequential([
2 tf.keras.layers.InputLayer(input_shape=[224, 224, 3]),
3 hub.KerasLayer("https://tfhub.dev/google/imagenet/resnet_v1_50/feature_vector/5", trainable=True), # Tensorflow Hubから
4 tf.keras.layers.Dense(class_num, activation='softmax')
5])
6 model.compile(optimizer=tf.keras.optimizers.Adam(), loss='categorical_crossentropy', metrics=['acc'])
7 model.summary()
Model: "sequential_1"
Layer (type) Output Shape Param #
keras_layer_1 (KerasLayer) (None, 2048) 23561152
dense_1 (Dense) (None, 5) 10245
Total params: 23571397 (89.92 MB)
Trainable params: 23517277 (89.72 MB)
Non-trainable params: 54120 (211.41 KB)
! history = model.fit(
train_batches,
epochs=EPOCHS,
batch_size=BATCH_SIZE,
validation_data=test_batches,
callbacks=[
tf.keras.callbacks.EarlyStopping(monitor='val_acc', patience=2, mode='max')
]
)
histories.append(history)
Epoch 1/20
101/101 [==============================] - 45s 112ms/step - loss: 1.3045 - acc: 0.6467 - val_loss: 6.7325 - val_acc: 0.4514
Epoch 2/20
101/101 [==============================] - 13s 81ms/step - loss: 0.4459 - acc: 0.7694 - val_loss: 5.5747 - val_acc: 0.4641
0秒 完了時間: 22:42


colab.research.google.com
4_2_wide_resnet.ipynb
ファイル 編集 表示 挿入 ランタイム ツール ヘルプ すべての変更を保存しました
コメント 共有
+ コード + テキスト
WideResNetの例
次の2つの例では、WideResNetの構造を用いた。
転移学習 (Weightを再利用、事前学習部分のWeightは固定)
[14] ! model = tf.keras.Sequential([
2 tf.keras.layers.InputLayer(input_shape=[224, 224, 3]),
3 hub.KerasLayer("https://tfhub.dev/google/wit/r50/3", trainable=False),
4 tf.keras.layers.Dense(class_num, activation='softmax')
5])
6 model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss='categorical_crossentropy', metrics=['acc'])
7 model.summary()
Model: "sequential_2"
Layer (type) Output Shape Param #
keras_layer_2 (KerasLayer) (None, 6144) 211174008
dense_2 (Dense) (None, 5) 30725
Total params: 211204485 (805.68 MB)
Trainable params: 30725 (120.02 KB)
Non-trainable params: 211174008 (805.57 MB)
! history = model.fit(
train_batches,
epochs=EPOCHS,
batch_size=BATCH_SIZE,
validation_data=test_batches,
callbacks=[
tf.keras.callbacks.EarlyStopping(monitor='val_acc', patience=2, mode='max')
]
)
histories.append(history)
Epoch 1/20
101/101 [==============================] - 73s 396ms/step - loss: 0.8318 - acc: 0.7123 - val_loss: 0.5825 - val_acc: 0.8405
0秒 完了時間: 22:42


colab.research.google.com
4_2_wide_resnet.ipynb
ファイル 編集 表示 挿入 ランタイム ツール ヘルプ すべての変更を保存しました
コメント 共有
+ コード + テキスト
ファインチューニング (事前学習部分のWeightも再調整)
[18] ! model = tf.keras.Sequential([
2 tf.keras.layers.InputLayer(input_shape=[224, 224, 3]),
3 hub.KerasLayer("https://tfhub.dev/google/wit/r50/3", trainable=True),
4 tf.keras.layers.Dense(class_num, activation='softmax')
5])
6 model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00001), loss='categorical_crossentropy', metrics=['acc']) # learning_rateを調整しています。
7 model.summary()
Model: "sequential_3"
Layer (type) Output Shape Param #
keras_layer_3 (KerasLayer) (None, 6144) 211174008
dense_3 (Dense) (None, 5) 30725
Total params: 211204485 (805.68 MB)
Trainable params: 211204485 (805.68 MB)
Non-trainable params: 0 (0.00 Byte)
! history = model.fit(
train_batches,
epochs=EPOCHS,
batch_size=BATCH_SIZE,
validation_data=test_batches,
callbacks=[
tf.keras.callbacks.EarlyStopping(monitor='val_acc', patience=2, mode='max')
]
)
histories.append(history)
Epoch 1/20
101/101 [==============================] - 198s 938ms/step - loss: 0.3327 - acc: 0.8881 - val_loss: 0.1052 - val_acc: 0.9382
Epoch 2/20
101/101 [==============================] - 142s 881ms/step - loss: 0.0281 - acc: 0.9849 - val_loss: 0.1007 - val_acc: 0.9655
Epoch 3/20
101/101 [==============================] - 142s 881ms/step - loss: 0.0048 - acc: 0.9982 - val_loss: 0.1168 - val_acc: 0.9610
Epoch 4/20
101/101 [==============================] - 142s 881ms/step - loss: 0.0026 - acc: 0.9992 - val_loss: 0.1136 - val_acc: 0.9637
[19] ! histories
0秒 完了時間: 22:42


colab.research.google.com
4_2_wide_resnet.ipynb
ファイル 編集 表示 挿入 ランタイム ツール ヘルプ
+ コード + テキスト
plot_df = pd.concat([plot_df, history_df], ignore_index=True)
sns.lineplot(data=plot_df, x=plot_df['Epochs'], y='acc', hue='method_name')
val_plot_df = plot_df
val_plot_df['method_name'] = plot_df['method_name'].apply(lambda x: 'val_' + x)
sns.lineplot(data=val_plot_df, x=plot_df['Epochs'], y='val_acc', hue='method_name', linestyle='dashed')
<Axes: xlabel='Epochs', ylabel='acc'>
ファインチューニング有りのWideResNet
ファインチューニング無しのResNet
ファインチューニング無しのWideResNet
ファインチューニング有りResNet
acc
Epochs
method_name
0
1
2
3
val_0
val_1
val_2
val_3
0 秒 完了時間: 22:42