PROJECT

Big Transfer (BiT)

General Representation Learning Method

PROJECT CONTRIBUTOR

Fernando Rios

1. Transfer Learning

Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task.

It is a popular approach in deep learning where pre-trained models are used as the starting point on computer vision and natural language processing tasks given the vast compute and time resources required to develop neural network models on these problems and from the huge jumps in skill that they provide on related problems.

Nevertheless, transfer learning is popular in deep learning given the enormous resources required to train deep learning models or the large and challenging datasets on which deep learning models are trained.

Transfer learning only works in deep learning if the model features learned from the first task are general.

How to Use Transfer Learning?

We can use transfer learning on your own predictive modeling problems.

Two common approaches are as follows:

- Develop Model Approach
- Pre-trained Model Approach

Develop Model Approach

Select Source Task. You must select a related predictive modeling problem with an abundance of data where there is some relationship in the input data, output data, and/or concepts learned during the mapping from input to output data.

Develop Source Model. Next, you must develop a skillful model for this first task. The model must be better than a naive model to ensure that some feature learning has been performed.

Reuse Model. The model fit on the source task can then be used as the starting point for a model on the second task of interest. This may involve using all or parts of the model, depending on the modeling technique used.

Tune Model. Optionally, the model may need to be adapted or refined on the input-output pair data available for the task of interest.

Pre-trained Model Approach

Select Source Model. A pre-trained source model is chosen from available models. Many research institutions release models on large and challenging datasets that may be included in the pool of candidate models from which to choose from.

Reuse Model. The model pre-trained model can then be used as the starting point for a model on the second task of interest. This may involve using all or parts of the model, depending on the modeling technique used.

Tune Model. Optionally, the model may need to be adapted or refined on the input-output pair data available for the task of interest.

This second type of transfer learning is common in the field of deep learning.

When to Use Transfer Learning?

Transfer learning is an optimization, a shortcut to saving time or getting better performance.

In general, it is not obvious that there will be a benefit to using transfer learning in the domain until after the model has been developed and evaluated.

There are three possible benefits to look for when using transfer learning:

- Higher start. The initial skill (before refining the model) on the source model is higher than it otherwise would be.
- Higher slope. The rate of improvement of skill during training of the source model is steeper than it otherwise would be.
- Higher asymptote. The converged skill of the trained model is better than it otherwise would be.

Ideally, you would see all three benefits from a successful application of transfer learning.

It is an approach to try if you can identify a related task with abundant data and you have the resources to develop a model for that task and reuse it on your own problem, or there is a pre-trained model available that you can use as a starting point for your own model.

On some problems where you may not have very much data, transfer learning can enable you to develop skillful models that you simply could not develop in the absence of transfer learning.

The choice of source data or source model is an open problem and may require domain expertise and/or intuition developed via experience.

Popular pre-trained models

There are some pre-trained machine learning models out there that are quite popular. One of them is the Inception-v3 model, which was trained for the ImageNet “Large Visual Recognition Challenge." In this challenge, participants had to classify images into 1,000 classes like “zebra," “Dalmatian" and “dishwasher."

Microsoft also offers some pre-trained models, available for both R and Python development, through the MicrosoftML R package and the Microsoftml Python package.

2.- Big Transfer (BiT) architectural improvements

Big transfer is a receipt that uses the minimal number of tricks yet attains excellent performance on many tasks. Strong performance using deep learning usually requires a large amount of task-specific data and compute. These per-task requirements can make new tasks prohibitively expensive. Transfer learning offers a solution "Big Transfer". Task-specific data and compute are replaced with a pre-training phase. A network is trained once on a large, generic dataset, and its weights are then used to initialize subsequent tasks which can be solved with fewer data points, and less compute.

There are two (2) components that are necessary to build an effective network for transfer:

- Upstream components are those used during pre-training, and
- Downstream are those used during fine-tuning to a new task.

Upstream Pre-Training

- ***The first component is scale***. It is well-known in deep learning that larger networks perform better on their respective tasks. Further, it is recognized that larger datasets require larger architectures to realize benefits, and vice versa. Big transfer studies the effectiveness of scale (during pre-training) in the context of transfer learning, including transfer to tasks with very few data points. Big transfer investigates the interplay between computational budget (training time), architecture size, and dataset size. For this, were trained three BiT models on three large datasets:
    - ILSVRC-2012 which contains 1.3M images (BiT-S),
    - ImageNet-21k which contains 14M images (BiT-M), and
    - JFT which contains 300M images (BiT-L).

- ***The second component is Group Normalization (GN)*** **and Weight Standardization (WS)** . Batch Normalization (BN) is used in most state-of the-art vision models to stabilize training. However, we find that BN is detrimental to Big Transfer for two reasons.
    - First, when training large models with small per-device batches, BN performs poorly or incurs inter-device synchronization cost.
    - Second, due to the requirement to update running statistics, BN is detrimental for transfer. GN, when combined with WS, has been shown to improve performance on small-batch training for ImageNet and COCO

  Here, we show that the *combination of GN and WS is useful for training with large batch sizes*, and has a significant impact on transfer learning.

All of BiT models use a vanilla ResNet-v2 architecture, replacing all Batch Normalization layers with Group Normalization and use Weight Standardization in all convolutional layers. All the models upstream were trained using SGD with momentum.

Transfer to Downstream Tasks

BiT proposes a cheap fine-tuning protocol that applies to many diverse downstream tasks. Importantly, it avoids expensive hyperparameter search for every new task and dataset size; BiT tries only one hyperparameter per task. BiT uses a heuristic rule—which we call **BiT-HyperRule**—to select the most

important hyperparameters for tuning as a simple function of the task's intrinsic image resolution and number of datapoints.

BiT found it important to set the following hyperparameters per-task: **training schedule length, resolution, and whether to use MixUp regularization** . BiT used BiT-HyperRule for over 20 tasks, with training sets ranging from 1 example per class to over 1M total examples.

During fine-tuning, we use the following standard data pre-processing: we resize the image to a square, crop out a smaller random square, and randomly horizontally flip the image at training time. At test time, we only resize the image to a fixed size. In some tasks horizontal flipping or cropping destroys the label semantics, making the task impossible.

Recent work has shown that existing augmentation methods introduce inconsistency between training and test resolutions for CNNs. Therefore, it is common to scale up the resolution by a small factor at test time. As an alternative, one can add a step at which the trained model is fine-tuned to the test resolution.

The latter is well-suited for transfer learning; we include the resolution change during our fine-tuning step. We found that MixUp is not useful for pre-training BiT, likely due to the abundance of data. However, it is sometimes useful for transfer.

Interestingly, it is most useful for mid-sized datasets, and not for few-shot transfer. Surprisingly, we do not use any of the following forms of regularization during downstream tuning: weight decay to zero, weight decay to initial parameters, or dropout. Despite the fact that the network is very large—BiT has 928 million parameters—the performance is surprisingly good without these techniques and their respective hyperparameters, even when transferring to very small datasets. We find that setting an appropriate schedule length, i.e. training longer for larger datasets, provides sufficient regularization.

To attain a low per-task adaptation cost, BiT didn't perform any hyperparameter sweeps downstream, Instead, It used HyperRule, a heuristic to determine all hyperparameters for fine-tuning.

3. Differences between Group Normalization (GN), Weight Standardization (WS) and Batch Normalization (BN).

Batch Normalization (BN)

One of the main ideas behind BN is normalizing the layer inputs and preventing changes in their distribution during training, which enables faster and better convergence. BN transforms a mini-batch of features based on its computed statistics.

To understand the method let's assume that we have the net preactivation feature maps obtained after a convolutional layer in a four-dimensional tensor, Z, with the shape [m x h x w x c], where m is the number of examples in the batch size, h x w is the spatial dimension of the feature maps, and c is the number of channels. So, BN can be summarized in three steps, as follows:

1. Compute the mean and standard deviation of the net inputs for each mini-batch.
2. Standardize the net inputs for all examples in the batch.
3. Scale and shift the normalized net inputs using two learnable parameter vectors ɣ and β of size c (number of channels).

In the first step of BN, the mean μ and standard deviation σ of the mini batch are computed. Both μ and σ are vectors of size c (where c is the number of channels). Then, these statistics are used in step 2 to scale the examples in each mini-batch via z-score normalization (standardization), resulting in standardized net inputs, Z. As a consequence, these net inputs are mean-centered and have unit variance, which is generally a useful property for gradient descent-based optimization. On the other hand, always normalizing the net inputs such that they have the same properties across the different mini-batches, which can be diverse, can severely impact the representational capacity of Neural Networks (NNs). This can be understood by considering a feature, X ~ *N* (0,1), which, after sigmoid activation to σ(X), results in a linear region for values close to 0. Therefore in step 3, the learnable parameters, β and ɣ, which are vectors of size c (number of channels), allow BN to control the shift and spread of the normalized features.

During the training, the running averages, μ and running variance σ are computed, which are used along with the tuned parameters β and ɣ, to normalize the test example(s) at evaluation. [3]

Group Normalization (GN)

The main idea of GN is divides the channels into groups and computes within each group the mean and variance for normalization. GN's computation is independent of batch sizes, and its accuracy is stable in a wide range of batch sizes.

The channels of visual representations are not entirely independent. Classical features are group-wise representations by design, where each group of channels is constructed by some kind of histogram. These features are often processed by groupwise normalization over each histogram or each orientation. Higher-level features are also group-wise features where a group can be thought of as the sub-vector computed with respect to a cluster.

Analogously, it is not necessary to think of deep neural network features as unstructured vectors. For example,

for conv1 (the first convolutional layer) of a network, it is reasonable to expect a filter and its horizontal flipping to exhibit similar distributions of filter responses on natural images. If conv1 happens to approximately learn this pair of filters, or if the horizontal flipping (or other transformations) is made into the architectures by design, then the corresponding channels of these filters can be normalized together.

The higher-level layers are more abstract and their behaviors are not as intuitive. However, in addition to orientations, there are many factors that could lead to grouping, e.g.frequency, shapes, illumination, textures. Their coefficients can be interdependent. In fact, a well-accepted computational model in neuroscience is to normalize across the cell responses, "with various receptive-field centers (covering the visual field) and with various spatiotemporal frequency tunings"; this can happen not only in the primary visual cortex, but also "throughout the visual system" . Motivated by these works, the authors proposed new generic group-wise normalization for deep neural networks.

Formulation

Formally, a Group Norm layer computes μ and σ in a set S_i defined as:

$$\mathbf{Si = \{k \mid kN = i_N , [\, K_C \,/(C/G)\,] = [\, i_C /(C/G)\,]\}}$$

Here G is the number of groups, which is a predefined hyper-parameter (G = 32 by default). C/G is the number of channels per group. [ ] is the floor operation, and "$\mathbf{[\, K_C \,/(C/G)\,] = [\, i_C /(C/G)}$" means that the indexes i and k are in the same group of channels, assuming each group of channels are stored in a sequential order along the C axis. GN computes μ and σ along the (H, W) axes and along a group of C/G channels. The computation of GN is for example a simple case of 2 groups (G = 2) each having 3 channels.

Relation to Prior Work. Layer Norm (LN), Instance Norm (IN), and GN all perform independent computations along the batch axis. The two extreme cases of GN are equivalent to LN and IN.

Relation to Layer Normalization. GN becomes LN if we set the group number as G = 1. LN assumes all channels in a layer make "similar contributions". Unlike the case of fully-connected layers studied in [3], this assumption can be less valid with the presence of convolutions. GN is less restricted than LN, because each group of channels (instead of all of them) are assumed to subject to the shared mean and variance; the model

still has flexibility of learning a different distribution for each group. This leads to improved representational power of GN over LN.

Relation to Instance Normalization. GN becomes IN if we set the group number as G = C (i.e., one channel per group). But IN can only rely on the spatial dimension for computing the mean and variance and it misses the opportunity of exploiting the channel dependence.

Implementation

GN can be easily implemented by a few lines of code in PyTorch and TensorFlow where automatic differentiation is supported.

Python Code

```
def GroupNorm(x, gamma, beta, G, eps=1e−5):
    # x: input features with shape [N,C,H,W]
    # gamma, beta: scale and offset, with shape [1,C,1,1]
    # G: number of groups for GN
    N, C, H, W = x.shape
    x = tf.reshape(x, [N, G, C // G, H, W])
    mean, var = tf.nn.moments(x, [2, 3, 4], keep dims=True)
    x = (x − mean) / tf.sqrt(var + eps)
    x = tf.reshape(x, [N, C, H, W])
    return x ∗ gamma + beta
```

In fact, we only need to specify how the mean and variance (“moments”) are computed, along the appropriate axes as defined by the normalization method. [4]

Weight Standardization (WS)

WS is inspired by BN. It has been demonstrated that BN influences network training in a fundamental way: it makes the landscape of the optimization problem significantly smoother. We argue that we can also standardize the weights in the convolutional layers to further smooth the landscape. By doing so, we do not have to worry about transferring smoothing effects from activations to weights; moreover, the smoothing effects on activations and weights are also additive. Based on these motivations, the author proposed Weight Standardization (WS).

Consider a standard convolutional layer with its bias term set to 0:

$y = \hat{W} * x$

where $\hat{W} \in R^{O \times I}$ denotes the weights in the layer and $*$ denotes the convolution operation. For $\hat{W} \in R^{O \times I}$ O is the number of the output channels, I corresponds to the number of input channels within the kernel region of each output channel. Taking an example, where O = Cout and I = Cin × Kernel Size. In Weight Standardization, instead of directly optimizing the loss L on the original weights $\hat{W}$, we reparameterize the weights $\hat{W}$ as a function of W, i.e., $\hat{W} = WS(W)$, and optimize the loss L on W by SGD:

$$\hat{W} = \left[\hat{W}_{i,j} \mid \hat{W}_{i,j} = \frac{W_{i,j} - \mu_{W_{i,\cdot}}}{\sigma_{W_{i,\cdot}}}\right],$$

$$y = \hat{W} * x,$$

where

$$\mu_{W_{i,\cdot}} = \frac{1}{I}\sum_{j=1}^{I} W_{i,j}, \quad \sigma_{W_{i,\cdot}} = \sqrt{\frac{1}{I}\sum_{j=1}^{I} W_{i,j}^2 - \mu_{W_{i,\cdot}}^2 + \epsilon}.$$

Similar to BN, WS controls the first and second moments of the weights of each output channel individually in convolutional layers. Note that many initialization methods also initialize the weights in some similar ways.

Different from those methods, WS standardizes the weights in a differentiable way which aims to normalize gradients during back-propagation. Note that we do not have any affine transformation on $\hat{W}$. This is because we assume that normalization layers such as BN or GN will normalize this convolutional layer again, and having affine transformation will confuse and slow down training. [5]

4.- MixUp Regularization

Large deep neural networks are powerful, but exhibit undesirable behaviors such as memorization and sensitivity to adversarial examples. Mixup is a simple learning principle to alleviate these issues. In essence, mixup trains a neural network on convex combinations of pairs of examples and their labels. By doing so, mixup regularizes the neural network to favor simple linear behavior in-between training examples. Experiments on the ImageNet-2012, CIFAR-10, CIFAR-100, Google commands and UCI datasets show that mixup improves the generalization of state-of-the-art neural network architectures. Likewise, mixup reduces the memorization of corrupt labels, increases the robustness to adversarial examples, and stabilizes the training of generative adversarial networks.

The method of choice to train on similar but different examples to the training data is known as data augmentation (Simard et al., 1998), formalized by the Vicinal Risk Minimization (VRM) principle (Chapelle et al., 2000). In VRM, human knowledge is required to describe a vicinity or neighborhood around each example in the training data. Then, additional virtual examples can be drawn from the vicinity distribution of the training examples to enlarge the support of the training distribution. For instance, when performing image classification, it is common to define the vicinity of one image as the set of its horizontal reflections, slight rotations, and mild scalings. While data augmentation consistently leads to improved generalization (Simard et al., 1998), the procedure is dataset-dependent, and thus requires the use of expert knowledge. Furthermore, data augmentation assumes that the examples in the vicinity share the same class, and does not model the vicinity relation across examples of different classes.

These issues introduced a simple and data-agnostic data augmentation routine, termed mixup. In a nutshell, mixup constructs virtual training examples

$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j, \quad \text{where } x_i, x_j \text{ are raw input vectors}$$

$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j, \quad \text{where } y_i, y_j \text{ are one-hot label encodings}$$

(x_i, y_i) and (x_j, y_j) are two examples drawn at random from our training data, and $\lambda \in [0, 1]$. Therefore, mixup extends the training distribution by incorporating the prior knowledge that linear interpolations of feature vectors should lead to linear interpolations of the associated targets. mixup can be implemented in a few lines of code, and introduces minimal computation overhead.

Despite its simplicity, mixup allows a new state-of-the-art performance in the CIFAR-10, CIFAR100, and ImageNet-2012 image classification dataset. Furthermore, mixup increases the robustness of neural networks when learning from corrupt labels, or facing adversarial examples. Finally, mixup improves generalization on speech and tabular data, and can be used to stabilize the training of GANs.

The results of experiments suggest that mixup performs significantly better than related methods in previous work, and each of the design choices contributes to the final performance.

From empirical risk minimization (ERM) to Mixup

In supervised learning, we are interested in finding a function $f \in F$ that describes the relationship between a random feature vector X and a random target vector Y , which follow the joint distribution P(X, Y). To this end,

we first define a loss function that penalizes the differences between predictions f(x) and actual targets y, for examples (x, y) ~ P. Then, we minimize the average of the loss function over the data distribution P, also known as the expected risk:

$$R_\delta(f) = \int \ell(f(x), y) \mathrm{d}P_\delta(x, y) = \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i).$$

Learning the function f by minimizing is known as the Empirical Risk Minimization (ERM) principle (Vapnik, 1998). While efficient to compute, the empirical risk monitors the behaviour of f only at a finite set of n examples. When considering functions with a number parameters comparable to n (such as large neural networks), one trivial way to minimize is to memorize the training data (Zhang et al., 2017). Memorization, in turn, leads to the undesirable behaviour of f outside the training data (Szegedy et al., 2014).

However, the naive estimate P δ is one out of many possible choices to approximate the true distribution P. For instance, in the Vicinal Risk Minimization (VRM) principle (Chapelle et al., 2000), the distribution P is approximated by

$$P_\nu(\tilde{x}, \tilde{y}) = \frac{1}{n} \sum_{i=1}^{n} \nu(\tilde{x}, \tilde{y} | x_i, y_i),$$

where ν is a vicinity distribution that measures the probability of finding the virtual feature-target pair (˜x, y˜) in the vicinity of the training feature-target pair (xi , yi). In particular, Chapelle et al. (2000) considered Gaussian vicinities ν (˜x, y˜|xi , yi) = N (˜x − xi , σ 2) δ (˜y = yi), which is equivalent to augmenting the training data with additive Gaussian noise. To learn using VRM, we sample the vicinal distribution to construct a dataset D ν := {(˜xi , y˜i)} m i=1, and minimize the empirical vicinal risk:

$$R_\nu(f) = \frac{1}{m} \sum_{i=1}^{m} \ell(f(\tilde{x}_i), \tilde{y}_i).$$

The contribution of this paper is to propose a generic vicinal distribution, called mixup:

$$\mu(\tilde{x}, \tilde{y} | x_i, y_i) = \frac{1}{n} \sum_{j}^{n} \mathop{\mathbb{E}}_{\lambda} \left[\delta(\tilde{x} = \lambda \cdot x_i + (1 - \lambda) \cdot x_j, \tilde{y} = \lambda \cdot y_i + (1 - \lambda) \cdot y_j) \right],$$

where λ ~ Beta(α , α), for α $\in$ (0, ∞). In a nutshell, sampling from the mixup vicinal distribution produces virtual feature-target vectors

$$\tilde{x} = \lambda \, x_i + (1 - \lambda) x_j ,$$

$$\tilde{y} = \lambda \, y_i + (1 - \lambda) y_j ,$$

where (xi , yi) and (xj , yj) are two feature-target vectors drawn at random from the training data, and λ $\in$ [0, 1]. The mixup hyper-parameter α controls the strength of interpolation between feature-target pairs, recovering the ERM principle as α $\rightarrow$ 0. The implementation of mixup training is straightforward, and introduces a minimal computation overhead.

Mixup regularization may help combating memorization of corrupt labels, sensitivity to adversarial examples, and instability in adversarial training. In the experiments solved, the following trend is consistent: with

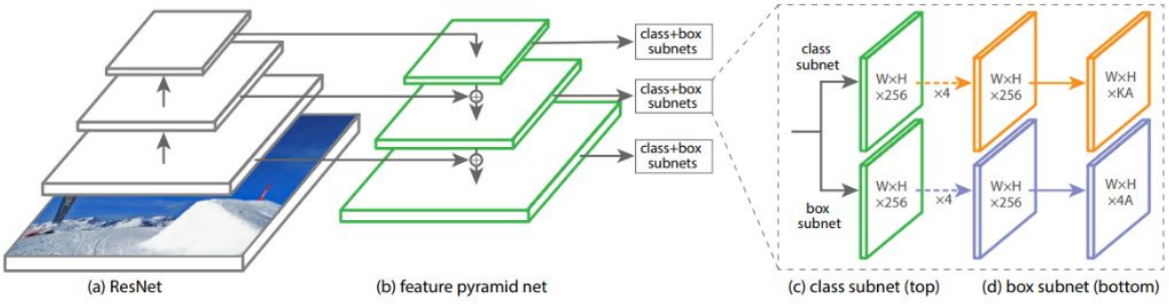
increasingly large α, the training error on real data increases, while the generalization gap decreases. This sustains our hypothesis that mixup implicitly controls model complexity. [6]

5. RetinaNet and ResNet

RetinaNet is one of the best one-stage object detection models that has proven to work well with dense and small scale objects. RetinaNet has been formed by making two improvements over existing single stage object detection models - Feature Pyramid Networks (FPN) [1] and Focal Loss [2]. Before diving into RetinaNet's architecture. FPN creates an architecture with rich semantics at all levels as it combines low-resolution semantically strong features with high-resolution semantically weak features. This is achieved by creating a top-down pathway with lateral connections to bottom-up convolutional layers.

RetinaNet architecture

There are four major components of a RetinaNet model architecture:
a) Bottom-up Pathway - The backbone network (e.g. ResNet) which calculates the feature maps at different scales, irrespective of the input image size or the backbone.
b) Top-down pathway and Lateral connections - The top down pathway upsamples the spatially coarser feature maps from higher pyramid levels, and the lateral connections merge the top-down layers and the bottom-up layers with the same spatial size.
c) Classification subnetwork - It predicts the probability of an object being present at each spatial location for each anchor box and object class.
d) Regression subnetwork - It regresses the offset for the bounding boxes from the anchor boxes for each ground-truth object.



Focal Loss

Focal Loss (FL) is an enhancement over Cross-Entropy Loss (CE) and is introduced to handle the class imbalance problem with single-stage object detection models. Single Stage models suffer from a extreme foreground-background class imbalance problem due to dense sampling of anchor boxes (possible object locations) [2]. In RetinaNet, at each pyramid layer there can be thousands of anchor boxes. Only a few will be assigned to a ground-truth object while the vast majority will be background class. These easy examples (detections with high probabilities) although resulting in small loss values can collectively overwhelm the

model. Focal Loss reduces the loss contribution from easy examples and increases the importance of correcting missclassified examples.

Restnet

ResNet is one of the most powerful deep neural networks which has achieved fantabulous performance results in the ILSVRC 2015 classification challenge. ResNet has achieved excellent generalization performance on other recognition tasks and won the first place on ImageNet detection, ImageNet localization, COCO detection and COCO segmentation in ILSVRC and COCO 2015 competitions.

There are many variants of ResNet architecture i.e. same concept but with a different number of layers. We have ResNet-18, ResNet-34, ResNet-50, ResNet-101, ResNet-110, ResNet-152, ResNet-164, ResNet-1202 etc. The name ResNet followed by a two or more digit number simply implies the ResNet architecture with a certain number of neural network layers.

Deeper networks can represent more complex features, therefore the model robustness and performance can be increased. However, stacking up more layers didn't work for the researchers. While training deeper networks, the problem of accuracy degradation was observed. In other words, adding more layers to the network either made the accuracy value to saturate or it abruptly started to decrease. The culprit for accuracy degradation was vanishing gradient effect which can only be observed in deeper networks.

What is Deep Residual Network?

Deep Residual Network is almost similar to the networks which have convolution, pooling, activation and fully-connected layers stacked one over the other. The only construction to the simple network to make it a residual network is the identity connection between the layers.

How does it work ?

The mapping we need to solve is: H(x)

Now we will convert this problem to solve the residual mapping function of the network, which is F(x), where F(x) = H(x)-x.

Residual: the difference between the observed value and the estimated value.

where H(x) is the observed value and x is the estimated value (that is, the feature map output by the previous layer of ResNet).

We generally call x the identity function, which is a jump connection; call F(x) ResNet Function.

Then the problem we want to solve becomes H(x) = F(x)+x.

Some may wonder, why do we have to solve H(x) after F(x)! Why is it so troublesome!

If it is a general convolutional neural network, we originally asked for the value of H(x) = F(x), right? Well, we now assume that when the network reaches a certain depth, our network has reached the optimal state, that is, when the error rate at this time is the lowest, further deepening the network will be There is a degradation problem (problem with an increased error rate). It will become very troublesome for us to update the weight

of the next layer of network now. The weight is to make the next layer of network also the optimal state. Right?

But using residual network can solve this problem well. Still assume that the depth of the current network can make the error rate the lowest. If we continue to increase our ResNet, in order to ensure that the network status of the next layer is still the optimal state, we only need to make F(x)=0! Because x is the optimal solution of the current output, in order to make it the optimal solution of the next layer, that is, if we want our output H(x)=x, is it enough to just let F(x)=0?

Of course, the above mentioned is just the ideal situation. In our real test, x is definitely difficult to achieve the optimal, but there will always be a time when it can be infinitely close to the optimal solution. If you use ResNet, you only need to update the weight value of the F(x) part! You don't have to do anything like a normal convolutional layer!

We use F(x)+x to represent H(x)!

Its formula is also quite simple (two-layer structure is given here)

$:a^{[l+2]}=Relu(W^{[l+2]}(Relu(W^{[l+1]}]a^{[l]}+b^{[l+1]})+b^{[l+2]}+a^{[l]})$

Note: If the dimension of the result of the residual mapping (F(x)) is different from the dimension of the jump connection (x), then we have no way to add the two of them. They can only be calculated when their dimensions are the same.

There are two ways to increase the dimension:

All 0 padding;

Use 1*1 convolution.

References

1.- Big Transfer (BiT) General Visual Representation Learning, Google Research, Brain Team Zurich Switzerland.

2.- A Gentle Introduction to Transfer Learning for Deep Learning, by Jason Brownlee.

https://machinelearningmastery.com/transfer-learning-for-deep-learning/

3.- Machine Learning with Python, Third edition. Sebastian Raschka & Vahid Mirjalili

4.- Group Normalization. Yuxin Wu & Kaiming He

5.- Micro-Batch training with batch-channel normalization and weight standardization. Siyuan Qiao, Huiyu Wang, Chenxi Liu, Wei Shen, and Alan Yuille.

6.- Mixup: Beyond empirical risk minimization. Hongyi Zhang (MIT) , Moustapha Cisse, Yann N. Dauphin, David Lopez-Paz

7.- How RetinaNet works?. https://developers.arcgis.com/python/guide/how-retinanet-works/

8.- Detailed Guide to Understand and Implement ResNets.

https://cv-tricks.com/keras/understand-implement-resnets/