Big Transfer

John Morrison
NJIT 2020

Part I. Transfer Learning.

Overview

Transfer learning is a methodology utilized in the training aspect of neural networks. The general idea is that you leverage a model that has already been trained on a robust set of data and incorporate it in your own model. This could be extremely useful if you happen to lack tons of training data. One example might be you want to create a model that classifies types of plants and your dataset is relatively small, less than 1000 labels. The challenge here is that image classification requires massive amounts of data. Luckily for you people have already done this heavy lifting, one common model people use is ResNet. Being that this model has been trained on millions of diverse types of images it knows the basics of analyzing pixels, detecting edges, curves, objects, etc. - this knowledge could then be leveraged by your model which has a specific use case of classifying plants.

Another example is from human biology. Babies come equipped with the ability to recognize faces. This is incredible because with that problem out of the way the baby could focus on more complex problems like understanding emotions or language. The analogy can be made that the face model was already embedded in this new model, lets call it, emotion classifier. The model that is trained on the robust set of data is called the pretrained model and the specialized model is the one constructed to perform the users desired task. These models are combined and in that combination the knowledge is transferred to the new model.

Mechanics

Okay, so how do you do it? First, you take the pretrained model and lop off the last layer. This makes sense because the last layer is responsible for spitting out a prediction, usually its some one hot encoding. This then could be optimized for the new classes by generating losses and conducting back-prop. To illustrate, you might replace the last layer of the pretrained model, which was originally be responsible for predicting a pair of scissors or a cat, and then insert a new layer that is built to predict a class of plant. Once the new model is constructed, to combine the pretrained and new final layer(s), you might want to think about if you should freeze the pretrained weights otherwise they would be adjusted in the backpropagation process. Sometimes freezing is ideal especially if you do not have a lot of training data. The only thing that would be adjusted is the new plant specific layer(s). The next step is to feed the new data into the model, in our case it is images of plant with various class labels. Again, this step is referred to as fine-tuning. At this stage, the model is adapting to your specific use case.

Advantages

Just to give a general recap, transfer learning starts with a very large pretrained model that is great for developing important feature representations and then in the fine-tuning process these are mapped to the new data. Transfer learning is great because you could leverage other people's work. Also, transfer learning helps speed up your training performance quite a bit. The lower-level features are already learned, so you can get to a correct prediction faster. In this explanation of transfer learning, I have only mentioned image classification, but this methodology could be extended to different domains like voice detection, natural language processing, etc.

Part II. ResNet Architectural Improvements.

BiT Overview

BiT stands for big transfer – the research was conducted by Google's brain team. The main goals of the research were to create various pretrained models mainly for the purpose of transfer learning. The models were trained on three different data sets: a "small" set containing 1.28 million images and 1000 classes, a "medium" set containing 14.2 million images 21k classes (multi-label), and a "large" consisting of 300 million images with 1.26 labels per image and a hierarchy of classes.

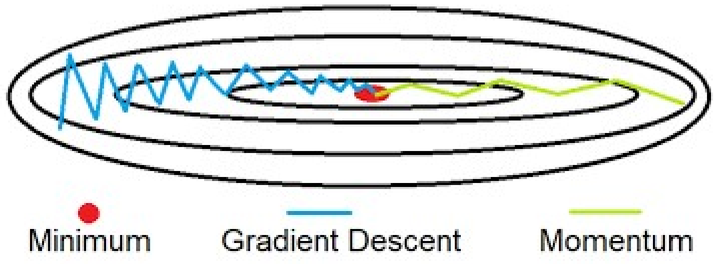
General Format

The paper's analysis is bifurcated into two forms of commentary: upstream and downstream. The upstream sections refer to the pretrained models/architecture and the downstream sections refer to the fine-tuning tasks and methodologies. The upstream models contain various architectural tweaks to ResNet-v1, which will be discussed in the following sections. On the fine-tuning side, the researchers employ a heuristic to set the different hyperparameters. Many of the hyperparameters are fixed, however some are adjusted based on training set sizes and image resolution. The heuristic helps lower the per task adaptation cost (hyperparameter search can be computationally expensive). However, it should be noted that the researchers abandoned to the heuristic to achieve better results on certain benchmarks. The heuristic is referred to as the BiT Hyper Rule. The data is preprocessed with common image techniques: square resizing, center cropping, and random horizontal flipping. The researchers use augmentation methods to improve consistency. They scale up the resolution by a small factor at test time.

ResNet Adjustments

The Resnet model was adjusted mainly to make it more robust and scalable. The first change worth noting is the swapping of batch normalization (BN) for group normalization (GN). This adjustment was important because the training was distributed to various machines and processing units which in many cases cause the batches to become very small. This made the batch dimension somewhat unreliable in terms of normalization because the statistics were skewed due to the small number of samples per batch. Sometimes there was only one sample in the batch at a time – and you cannot normalize if you have nothing to compare to. More to come on this topic in section III. Additionally, weight standardization was used in all convolution layers. Another adjustment that was made was they increased the size of the hidden layers by a factor of four (the largest model "default" ResNet 152x4).

The pretrained models were trained using SGD with momentum. The momentum improves training performance significantly. The momentum promotes steps that go downward toward the minimum and apply "friction" to steps that do not. This helps limit the oscillation that might occur during gradient descent. The phenomenon of the oscillation is denoted by the blue line bouncing back and forth and the green line shows the improved learning with momentum. The initial learning rate was set to 0.03 with a momentum of 0.9 which is most commonly used. Also, the images are cropped, randomly flipped, and resized to 224x224. They schedule the learning rate in a stepwise fashion, reducing by a factor of 10. The learning rate would drop at epochs 30, 60, and 80. The largest model requires less epochs, or there are less due to size. For this model, the learning rates change at epochs 10, 23, 30, and 37.



Like momentum, the scheduled learning rate improves the performance with respect to optimization – the steps get smaller as the model moves through the epochs. An epoch is a full pass through all the training data. Lastly, the researchers also utilize warm up and weight decay.

Part III. GN, WS and BN.

Normalization Overview

First, I want to discuss why normalizing input data is important and then later express the mechanics of doing these processes within a neural network. There are benefits to normalizing your data – training speed is the most apparent. The main reason we normalize the dataset is because the different features might have wildly different scales. For example, one feature might be denoted in dollars and another denoted in minutes. These two data points might be far apart in a linear space. This slight mismatch in scale could make your cost function elongated which is a non-optimal shape when conducting gradient descent. Alternatively, when you normalize your data you make the scales all the same and they could range from either -1 to 1 or even 0 to 1, all of which is better than the initial representation. By transforming your different features into this scale, your cost function will subsequently shift from being long and complex into of more of a bowl shape, not exactly because you are most likely dealing with high-dimensional space, but it is a helpful visualization. This subsequently speeds up the training process and reduces unnecessary oscillations in the learning steps. To do the transformation you subtract the mean of the input data from the data point and then divide by the standard deviation. This should be done for both the training and test set.

Everything described above relates to input data. These steps occur prior to the model receiving the data. Afterwards, an important thing to consider is the fact that while the data is going through the various layers the normalized inputs get shifted and transformed by the parameters. This means that more normalization will need to be done within the model to keep the training performance.

Batch Norm

To piggyback on the problem posed at the end of the previous section, batch norm aims to address the normalization within the network. I think it is easy to intuit that -> if it makes sense to do this transformation at the input stage -> it makes sense to do in between the hidden layers. Rather than transforming all the data points again, which is not feasible during training, the normalization is conducted at the mini-batch level. Not only is it conducted at the batch level, but it also occurs after any layer where it is implemented (it may be implemented multiple times). This makes the various layers more independent in some regard. The intermediary transformation that occurs when batch norm is not implemented is a called internal covariate shift. Actually, the shift could still occur when batch norm is implemented, but in the unnormalized scenario there is less control. When you do not implement batch norm you could potentially be in a situation where your data is altered, and then you lose all the benefits of normalizing your input data.

Additional trainable parameters are introduced in batch norm as well. Sometimes within these layers you want the data to shift if that means it will produce a better prediction. The data is being transformed by the weights for a reason, so it makes sense that you would want some adjustment to occur. The new trainable parameters are denoted as gamma and beta - these subsequently would be adjusted in back-prop – the gamma is a scalar, and the beta is a bias. Since BN has a bias it usually makes sense to eliminate the other bias that might directly precede the batch norm layer.

One last interesting fact about Batch Norm is since the statistics used to normalize are at the minibatch level they introduce additional noise due to the variance. This added noise could create some drop-out like effects. Even though this is the case it is still very common to use dropout and regularization techniques in addition to batch norm.

Group Norm and Weight Standardization

In the BiT paper they explain the main issue with using batch norm for their implementation is that many times their batch only contains one data point, or image. This fact makes it impossible to generate statistics and normalize because there is nothing to compare it to. This problem occurs because they were distributing the training across multiple GPU's (or TPU's). With that said Batch Norm was abandoned, and in place they used a strategy called Group Norm (GN). Group Norm does not share this same issue because the normalization operations are done locally within a data point rather than across the batch dimension.

The inspiration for GN came from different image processing frameworks like HOG and SIFT. To give some background HOG stands for Histogram Oriented Gradients which is a machine learning approach that utilizes support vector machines for classification tasks, a non-deep learning approach. In this approach the gradients detect edges in an image and they are represented by histograms that get concatenated and normalized. There were other precursors to GN as well such as Layer Norm and Instance Norm. Layer Norm aimed to normalize across multiple channels but in a single datapoint. The main downside of this approach is the fact that if you lump all the channels together certain features might not be generated due the fact that you are convolving them all together. Instance Norm goes to the extreme in the other direction where the normalization is done only in one channel in one data point – this is essentially batch norm with a batch size of 1. Group Norm kind of meets in the middle and normalizes groups of channels. Essentially you take the number of channels and divide by the number groups that you want and then normalize the data within the groups.

Weight Standardization is the final technique or methodology to be discussed. This differs from BN and GN in the fact that rather than transforming the data itself you are normalizing weights. In a CNN you are standardizing the kernel specifically. The is kernel is responsible for sliding across the various pixels of an image and doing some transformation. These kernel weights are trainable as well. As a side note, the term kernel gets thrown around in machine learning and deep learning, but it does not always mean the same thing. The term, from my understanding, broadly speaking just represents some function utilized within a model. Anyway, the kernel's weights first are initialized, but then once the training occurs, they can potentially get larger or smaller. This could create instability in the optimization of the model. The solution to this is to do the same normalization techniques, but this time on the weights. You would take the weight subtract the mean of the weights and divide by the standard deviation. After it is normalized you would multiply by the data point as you generally would. This process can easily be processed via backprop as well.

The researchers at google showed this is a very successful methodology and reduces the inter-device synchronization costs. It can also be stated that it holds up in terms of performance on large batch sizes as well, so it could be safe to say that Group Norm matched with Weight Standardization can be a very effective and scalable approach.

Part IV. MixUp Regularization.

Overfitting

Regularization in general is an approach aimed to lessen the effects of overfitting. Deep neural networks are equipped to minimize the average error over a training set when making predictions – this is principle is sometimes referred to as Empirical Risk Minimization. If you look at your training set as some sample of the real population that exists in the world, this ERM phrasing makes sense. You can never know the true global answer you can merely minimize the risk of being wrong in an empirical sense. In the neural network framework this is achieved by calculating the loss and adjusting accordingly and going fully through the data multiple times. Sometimes, especially in large models, this training framework works too well and that is when the phenomenon of overfitting occurs. There are a number of strategies to deal with this problem – L2 regularization is one of them. This is where you try to minimize the effect of certain parameters, so that they do not overfit to your data. In many cases the model will become overly complicated and fit on the random noise of the training set. The l2 approach makes certain weights less damaging by imposing a penalty. The other side of the overfitting coin is the term generalization. This means not only does the model do well on the training set, but it also works on new data that comes in via the test set. With that said you sometimes you will have to sacrifice some performance on the training set to generalize better.

Augmentation and MixUp

In image classification it is common to perform data augmentation to achieve generalization. Data augmentation is an approach where you potentially flip, rotate, alter images, etc., so that you keep the same spirit of the important features but alter the data slightly. Many times, the model will just memorize the data at a certain point and this augmentation helps deal with that. However, these augmentation steps sometimes require domain expertise in terms of choosing what augmentation strategies to use, or it can be very data dependent. This caveat is what led to the development of MixUp, which is data agnostic method.

The approach is the following, you take two training examples and mix-up the features to create a new training example. By creating this virtual data point, a new image, it prevents the model from memorizing the picture exactly because it has never seen it in its original form. These two images are interpolated by sampling from a beta distribution. The beta distribution helps you sample proportions, essentially values between 0 and 1, and you can adjust how skewed you want them to be by tuning alpha. This sampling gives you the lambda for your MixUp. Side note, lambda is a reserved keyword in Python so avoid spelling it out completely if you are setting it as a variable name. After you have lambda you can then multiply it by the first training example to give you some portion of the image. Afterwards, you multiply the compliment of that lambda (1-lambda) by the other sampled data point. Those two images would combine to make your new x value. The same would be done for your labels, also known as the y values. In regards to the labels it is important that they are one-hot-encodings to start … afterwards the new y should be represented by some percentage of the new classes. For example, if you combined a picture of 75% a cat and a picture of 25% a tree -> the y distribution should reflect that rather than a one-hot-encoding. This is important in calculating the loss.

There are other types of augmentation-like techniques that might be relevant like regional dropout where random parts of an image are removed. Another method is cut and paste which is like MixUp but slightly harsher in terms of its' borders. SmoothMix is another technique that incorporates a blending mask.

In the BiT paper they really only used MixUp in the downstream tasks. The researchers explain that it was especially useful in the pretrained models mainly because of the vast amount of training data. I would imagine that the model is not quite large enough, from a parameter standpoint, to overfit at that scale. In the downstream tasks they noted that MixUp was not especially necessary for the few shot learning tasks. However, they did use it for the large and medium tasks and they set the alpha to 0.1.

V. Models.

ResNet

For the various performance results they ran analysis on the different ResNet architectures: ResNet-50x1, ResNet-50x3, ResNet-101x1, ResNet-101x3, and ResNet-152x4. The first number refers to the number of layers in the original architecture and the second figure, the multiplier, represents the increase the BiT team made to the layers. The ResNet framework in general is a deep convolutional network, the “Res” stands for Residual. The ResNet project was born out of Microsoft and their main innovation in this architecture is the skip connection technique. This allowed for the networks to be deeper and more complex. The skip mechanism passes information down from further upstream bypassing the main path – this helps the model retain information.

RetinaNet

For object detection tasks the RetinaNet model was used. Object detection is slightly more complicated compared to your standard classification problem. There is a challenge of understanding features that lie in the foreground and background. In object detection you typically see some bounding boxes and labels that are associated. This particular model aimed to improve on the process used in R-CNN models which use a two-stage approach. The first stage typically generates a sparse matrix of candidates while the second pass does some region proposals. The RetinaNet achieves object detection in one stage. The main improvement was with Feature Pyramid Networks (FPN) and an improved loss function. The loss function is called Focal Loss. They attach a subnet to the FPN level predict object presence at various spatial positions. The state of the art in object detection is ever changing.

In closing, this project helped me understand the entire ML/AI community is standing on the shoulders of giants. There is constant adaption of old techniques to more modern ones. Transfer learning truly exemplifies this sentiment.

References

1. Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, Neil Houlsby: "Big Transfer (BiT): General Visual Representation Learning", 2019; arXiv:1912.11370.

2. Yuxin Wu, Kaiming He: "Group Normalization", 2018; arXiv:1803.08494.

3. Siyuan Qiao, Huiyu Wang, Chenxi Liu, Wei Shen, Alan Yuille: "Micro-Batch Training with Batch-Channel Normalization and Weight Standardization", 2019; arXiv:1903.10520.

4. Sergey Ioffe, Christian Szegedy: "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", 2015; arXiv:1502.03167.

5. Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, David Lopez-Paz: "mixup: Beyond Empirical Risk Minimization", 2017; arXiv:1710.09412.

6. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun: "Deep Residual Learning for Image Recognition", 2015; arXiv:1512.03385.

7. Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollár: "Focal Loss for Dense Object Detection", 2017; arXiv:1708.02002.