**Project 3: Big Transfer (‘BiT’) Learning**

**CS 677 Deep Learning**

**Abhishek Singh**

# Summary of ‘Big Transfer’ Learning

Transfer learning[1](#_Reference) focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. For example, knowledge gained while learning to recognize cars could apply when trying to recognize trucks. From the practical standpoint, reusing or transferring information from previously learned tasks for the learning of new tasks has the potential to significantly improve the sample efficiency of a reinforcement learning agent.

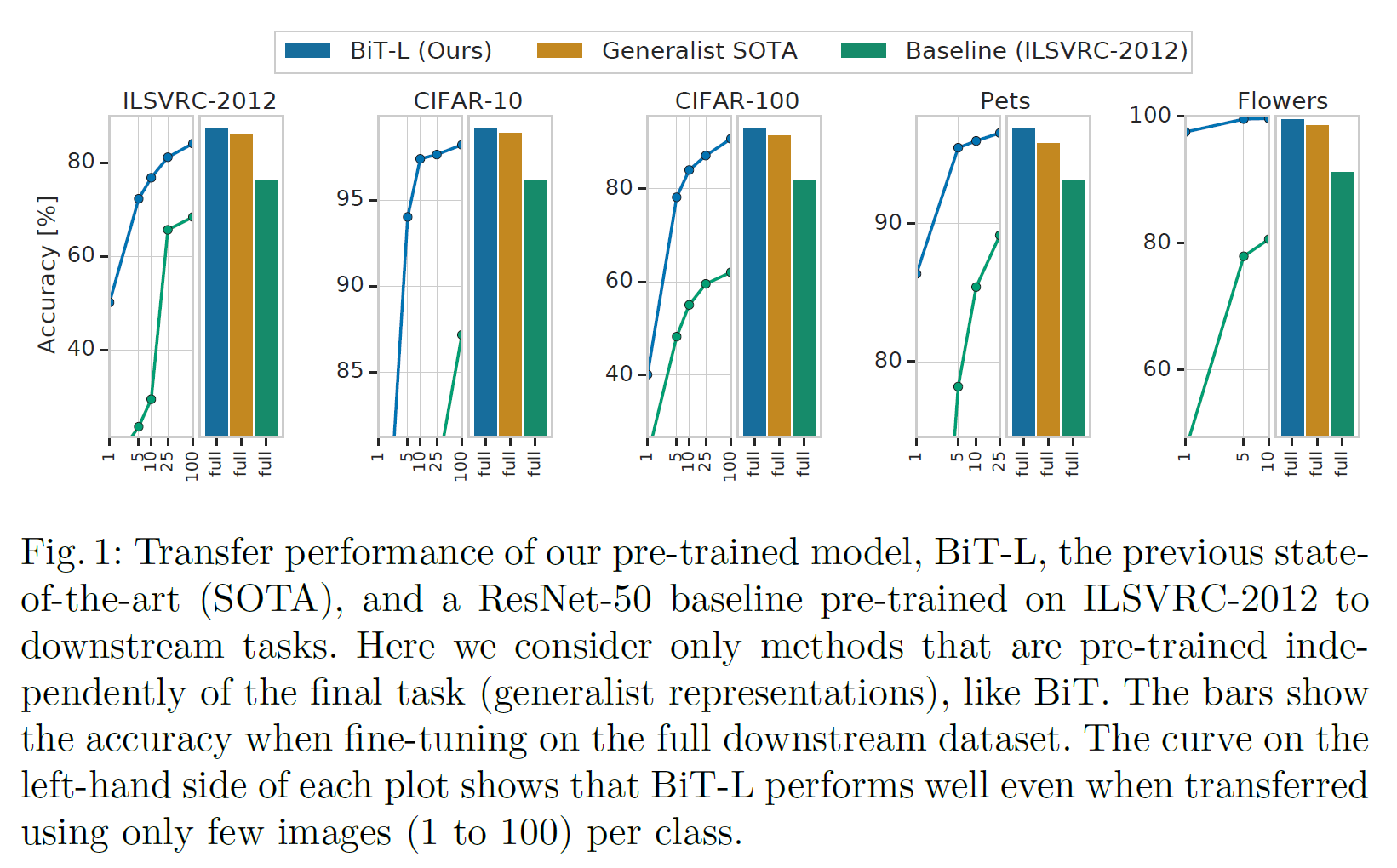
Deep learning model performance depends on huge amount of data and compute capacity. However, this also makes it expensive and thus prohibitive. To overcome this challenge with deep learning, transfer learning could be applied, especially if task-specific data and compute could be replaced with a pre-training 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.

Deep learning models have made great strides in computer vision, particularly in recognizing objects in images. One key to this success has been the availability of large-scale labelled datasets: collections of images with corresponding text descriptions of the objects they contain. These datasets must be created manually, with human workers applying a label to each of thousands of images: the popular ImageNet dataset, for example, contains over 14 million labeled images containing 21k different object classes. However, the images are usually generic, showing commonplace objects such as people, pets, or household items. Creating a dataset of similar scale for a specialized task, say for an industrial robot, might be prohibitively expensive or time-consuming.

In this situation, data engineers often apply transfer learning, a strategy that has become popular with large-scale natural-language processing (NLP) models. A neural network is first pre-trained on a large generic dataset until it achieves a certain level of performance on a test dataset. Then the model is fine-tuned with a smaller task-specific dataset, sometimes with as few as a single example of the task-specific objects. Large NLP models routinely set new state-of-the-art performance levels using transfer learning.

In Big transfer (‘BiT’) technique, a set of pre-trained image models can be transferred to obtain excellent performance on new datasets, even with only a few examples per class. Importantly, BiT only needs to be pre-trained once and subsequent finetuning to downstream tasks is cheap. By contrast, other state-of-the-art methods require extensive training on support data conditioned on the task at hand. Not only does BiT require a short fine-tuning protocol for each new task, but BiT also does not require an extensive hyperparameter tuning on new tasks. Instead, use heuristic method for setting the hyperparameters for transfer, which works well on our diverse evaluation suite.

In the seminal paper on BiT learning[2](#_Reference), the Google Brain team lists two components of BiT 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. The Google Brain team investigated large-scale pre-training and the effects of model size, dataset size, training duration, normalization strategy, and hyperparameter choice. As a result of this work, the team developed a "recipe" of components and training heuristics that achieves strong performance on a variety of benchmarks.



## 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. Google Brain team studied the effectiveness of scale (during pre-training) in the context of transfer learning, including transfer to tasks with very few datapoints. They investigate the interplay between computational budget (training time), architecture size, and dataset size. For this, they trained three BiT models on three large datasets: ILSVRC-2012 [46] which contains 1.3M images (BiT-S), ImageNet-21k [10] which contains 14M images (BiT-M), and JFT [51] 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, Google Brain team found 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. The combination of GN and WS is useful for training with large batch sizes and has a significant impact on transfer learning.

## Transfer to Downstream Tasks

The Google Brain team proposed 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; and try only one hyperparameter per task. Google Brain team used a heuristic rule (called 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. They found it important to set the following hyperparameters per-task: training schedule length, resolution, and whether to use MixUp regularization. They used BiT-HyperRule for over 20 tasks, with training sets ranging from 1 example per class to over 1M total examples.

During fine-tuning, they used the following standard data pre-processing: resized the image to a square, crop out a smaller random square, and randomly horizontally flip the image at training time. At test time, only resize the image to a fixed size. In some tasks, horizontal flipping or cropping destroys the label semantics, making the task impossible. An example is if the label requires predicting object orientation or coordinates in pixel space. In those cases, Google Brain team omit flipping or cropping when appropriate. 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.

It was 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.

BiT did not use any of the following forms of regularization during downstream tuning: weight decay to zero, weight decay to initial parameters, or dropout. Even though 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 enough regularization.

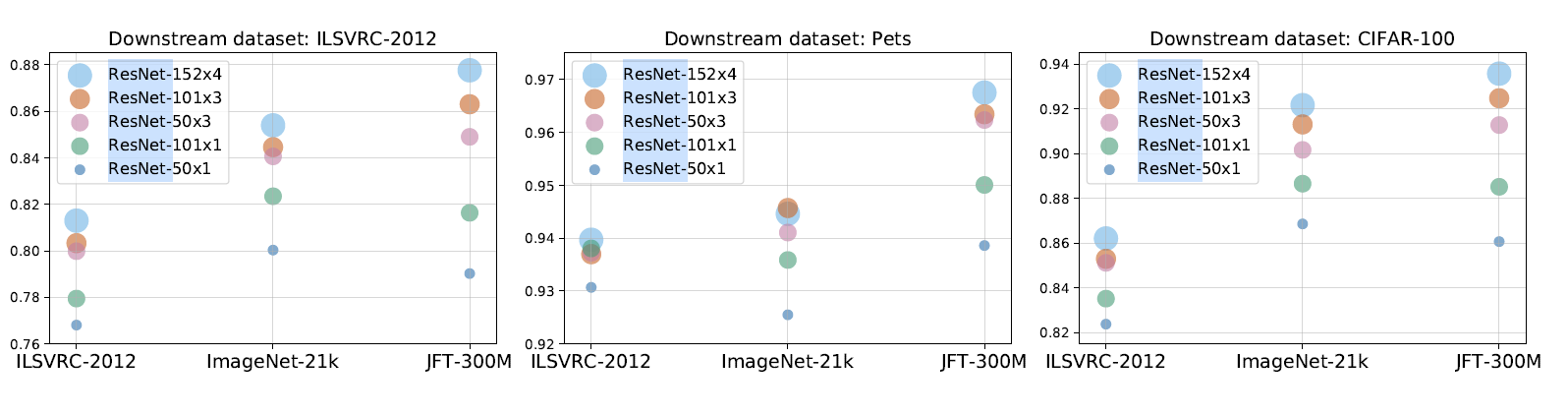
# Architectural improvements

For BiT, the Google Brain team used a ResNet-v2 neural architecture[3](#_Reference). To investigate the effects of pre-training dataset size, the team replicated their experiments on three groups of models pre-trained with different datasets: BiT-S models pre-trained on 1.28M images from ILSVRC-2012, BiT-M models pre-trained on 14.2M images from ImageNet-21k, and BiT-L models pre-trained on 300M images from JFT-300M. The models were then fine-tuned and evaluated on several common benchmarks: ILSVRC-2012, CIFAR-10/100, Oxford-IIIT Pet, and Oxford Flowers-102.

The Google Brain team noted several findings from their experiments. First, the benefits from increasing model size diminish on smaller datasets, and there is little benefit in pre-training smaller models on larger datasets. Second, the large models performed better using group normalization compared to batch normalization. Finally, to avoid an expensive hyperparameter search during fine-tuning, the team developed a heuristic called BiT-HyperRule, where all hyperparameters are fixed except "training schedule length, resolution, and whether to use MixUp regularization."

During upstream pre-training, all of BiT models uses a vanilla ResNet-v2 architecture, except that they replaced all Batch Normalization layers with Group Normalization and use Weight Standardization [43] in all convolutional layers. They trained ResNet-152 architectures in all datasets, with every hidden layer widened by a factor of four (ResNet152x4). They studied different model sizes and the coupling with dataset size.

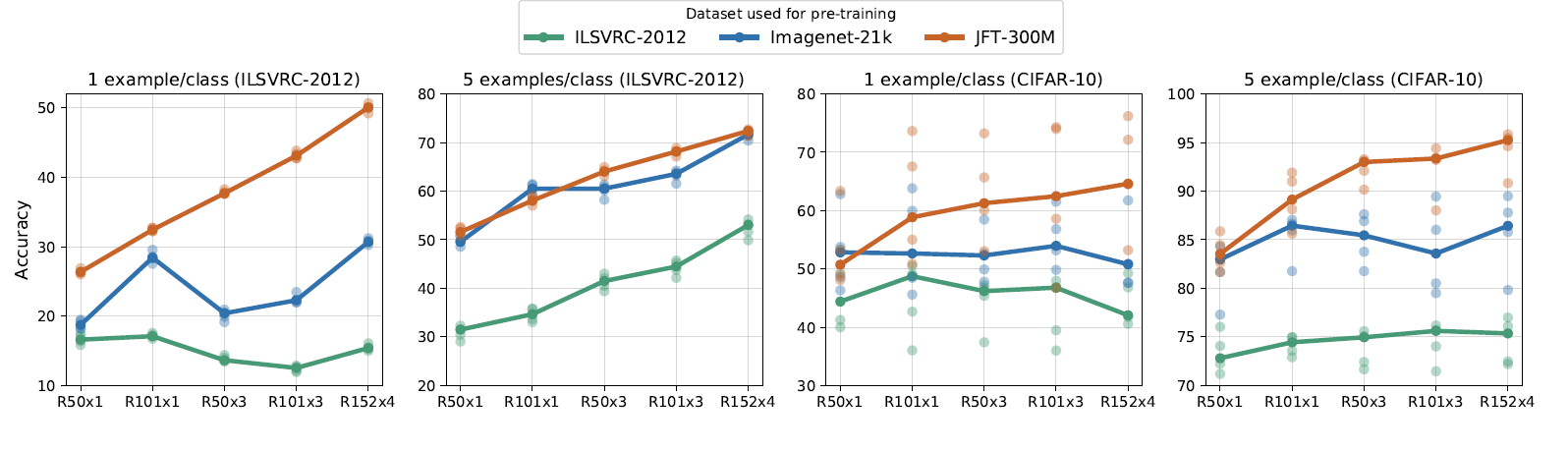
Google Brain team investigate the interplay between model capacity and upstream dataset size on downstream performance. They evaluate the BiT models of different sizes (ResNet-50x1, ResNet-50x3, ResNet-101x1, ResNet-101x3, and ResNet-152x4) trained on ILSVRC-2012, ImageNet-21k, and JFT-300M on various downstream benchmarks. These results are summarized below.



*Effect of upstream data (shown on the x-axis) and model size on downstream performance. Note that exclusively using more data or larger models may hurt performance; instead, both need to be increased in tandem.*

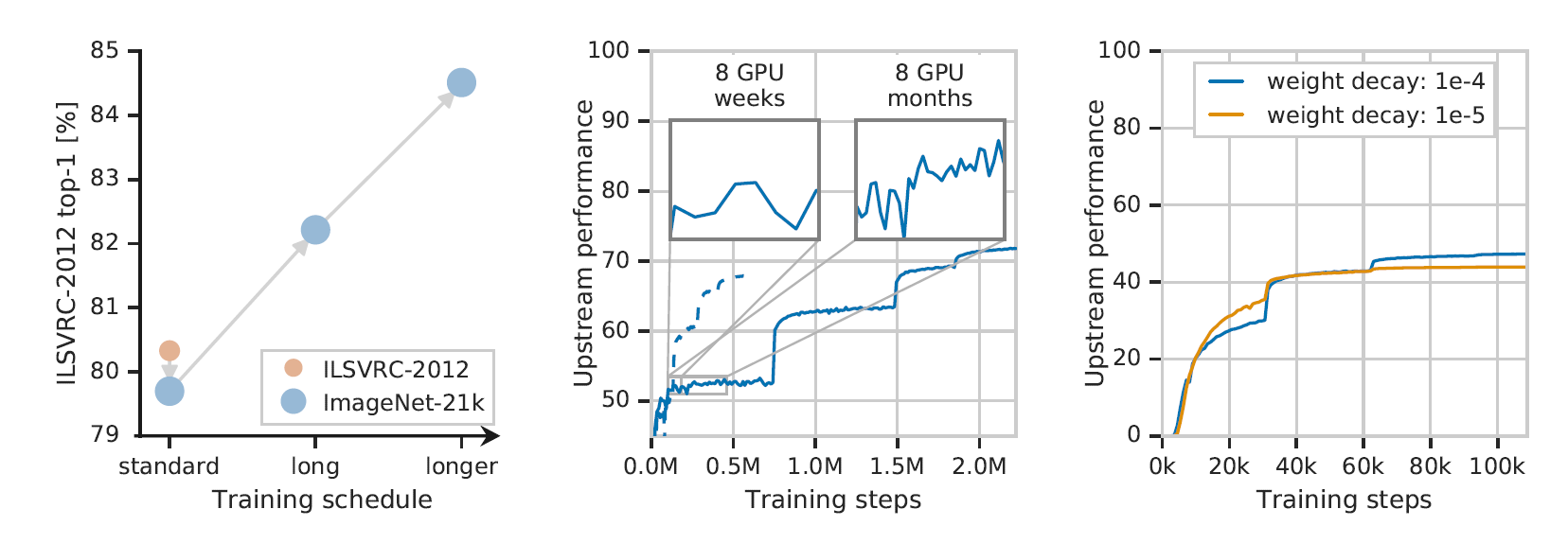
When pre-training on ILSVRC-2012, the benefit from larger models diminishes. However, the benefits of larger models are more pronounced on the larger two datasets. A similar effect is observed when training on Instagram hashtags and in language modelling. Not only is there limited benefit of training a large model size on a small dataset, but there is also limited (or even negative) benefit from training a small model on a larger dataset. Perhaps surprisingly, the ResNet-50x1 model trained on the JFT-300M dataset can even performs worse than the same architecture trained on the smaller ImageNet-21k. Thus, if one uses only a ResNet50x1, one may conclude that scaling up the dataset does not bring any additional benefits. However, with larger architectures, models pre-trained on JFT-300M significantly outperform those pre-trained on ILSVRC-2012 or ImageNet-21k.

Figure below ablates few-shot performance across different pre-training datasets and architectures. In the extreme case of one example per class, larger architectures outperform smaller ones when pre-trained on large upstream data. Interestingly, on ILSVRC-2012 with few shots, BiT-L trained on JFT-300M outperforms the models trained on the entire ILSVRC-2012 dataset itself. Note that for comparability, the classifier head is re-trained from scratch during fine-tuning, even when transferring ILSVRC-2012 full to ILSVRC-2012 few shots.



*Performance of BiT models in the low-data regime. The x-axis corresponds to the architecture, where R is for ResNet. We pre-train on the three upstream datasets and evaluate on two downstream datasets: ILSVRC-2012 (left) and CIFAR-10 (right) with 1 or 5 examples per class. For each scenario, we train 5 models on random data subsets, represented by the lighter dots. The line connects the medians of these five runs.*

Applying the “standard" computational budget of ILSVRC-2012, refer left picture below, to the larger ImageNet-21k seems detrimental. Only when we train longer (3x and 10x) do we see the benets of training on the larger dataset. The learning progress of a ResNet-101x3 on JFT-300M, refer to middle picture below, seems to be at even after 8 GPU-weeks, but after 8 GPU-months progress is clear. If one decays the learning rate too early (dashed curve), final performance is significantly worse. Faster initial convergence with lower weight decay, refer right picture below’ may trick the practitioner into selecting a sub-optimal value. Higher weight decay converges more slowly but results in a better final model.



# Batch Normalization, Group Normalization, and Weight Standardization

## Batch Normalization

In batch normalization (‘BN’), we normalize the input layer by adjusting and scaling the activations. Batch normalization is used to reduce the amount of hidden unit values shift i.e., covariance shift. Covariate shift refers to the change in the distribution of the input values to a learning algorithm. This is a problem that is not unique to deep learning. For instance, if the train and test sets come from entirely different sources (e.g. training images come from the web while test images are pictures taken on the iPhone), the distributions would differ. The reason covariance shift can be a problem is that the behavior of machine learning algorithms can change when the input distribution changes.

In the context of deep learning, a neural network changes the weights of each layer over the course of training. This means that the activations of each layer change as well. Since the activations of a previous layer are the inputs of the next layer, each layer in the neural network is faced with a situation where the input distribution changes with each step. This is problematic because it forces each intermediate layer to continuously adapt to its changing inputs.

The basic idea behind batch normalization is to limit covariate shift by normalizing the activations of each layer (transforming the inputs to be mean 0 and unit variance). This, supposedly, allows each layer to learn on a more stable distribution of inputs, and would thus accelerate the training of the network.

Calculation for all of the normalization methods is as follows:

xᵢ ← (xᵢ - 𝜇ᵢ) / √(𝜎ᵢ² + 𝜀)

for every coefficient xᵢ of an input feature x. 𝜇ᵢ and 𝜎ᵢ² are the mean and variance computed over a set Sᵢ of coefficients, and 𝜀 is a small constant added for numerical stability and to avoid division by zero. The only difference is how the set Sᵢ is chosen.

To illustrate the computation of the normalization methods, we consider a batch of size N = 3, with input features a, b, and c. They have channels C = 4, height H = 1, width W = 2:

a = [ [[2, 3]], [[5, 7]], [[11, 13]], [[17, 19]] ]

b = [ [[0, 1]], [[1, 2]], [[3, 5]], [[8, 13]] ]

c = [ [[1, 2]], [[3, 4]], [[5, 6]], [[7, 8]] ]

Hence the batch will have shape (N, C, H, W) = (3, 4, 1, 2). We take 𝜀 = 0.00001.

BN normalizes the channels and computes 𝜇ᵢ and 𝜎ᵢ along the (N, H, W) axes. Sᵢ is defined as the set of coefficients in the batch that are in the same channel as xᵢ.

For the first coefficient aᵢ = 2 of a, where i = (0, 0, 0), the corresponding 𝜇ᵢ and 𝜎ᵢ² are computed over the coefficients of a, b, and c that are in the first channel:

𝜇ᵢ = mean (2, 3, 0, 1, 1, 2) = 1.5

𝜎ᵢ² = var (2, 3, 0, 1, 1, 2) = 0.917

Plugging these into the normalization formula,

aᵢ ← (2 - 1.5) / √(0.917 + 0.00001) = 0.522

Computing all the coefficients of a gives

a ← [ [[0.522, 1.567]], [[0.676, 1.690]], [[1.071, 1.630]], [[1.066, 1.492]] ]

Higher learning rates can be used because batch normalization makes sure that there’s no activation that is high or low. And due to that, things that previously couldn’t get to train, can be trained. It reduces overfitting because it has a slight regularization effect. Like dropout, it adds some noise to each hidden layer’s activations. Therefore, if we use batch normalization, we will use less dropout, which is a good thing because we are not going to lose a lot of information. However, we should not depend only on batch normalization for regularization; we should better use it together with dropout.

To increase the stability of a neural network, batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation.

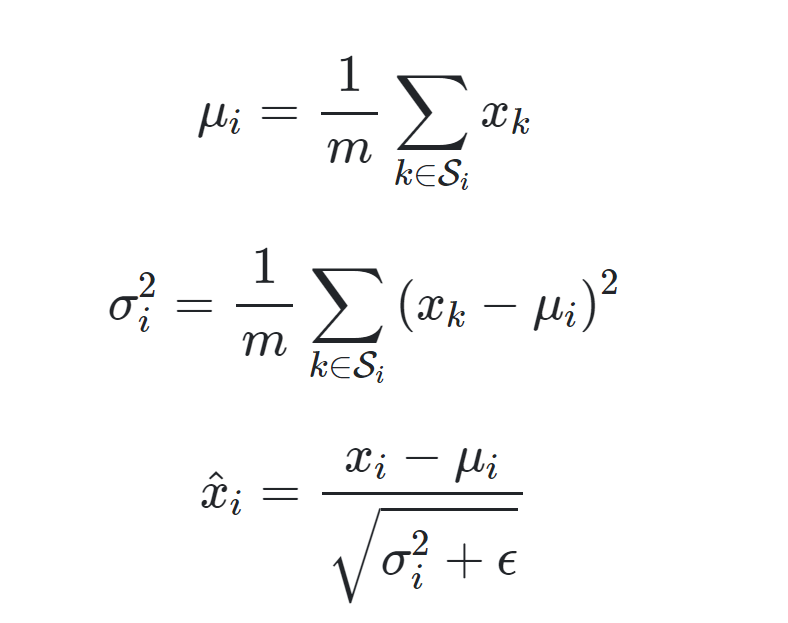
However, after this shift/scale of activation outputs by some randomly initialized parameters, the weights in the next layer are no longer optimal. SGD (Stochastic gradient descent) undoes this normalization if it’s a way for it to minimize the loss function.

Consequently, batch normalization adds two trainable parameters to each layer, so the normalized output is multiplied by a “standard deviation” parameter (gamma) and add a “mean” parameter (beta). In other words, batch normalization lets SGD do the denormalization by changing only these two weights for each activation, instead of losing the stability of the network by changing all the weights.

## Group Normalization

Group Normalization (‘GN’) is a normalization layer that divides channels into groups and normalizes the features within each group. GN does not exploit the batch dimension, and its computation is independent of batch sizes. In the case where the group size is 1, it is equivalent to Instance Normalization.

Formally, Group Normalization is defined as:



Here *x* is the feature computed by a layer, and *i* is an index. Formally, a GN layer computes and in a set S*i* defined as {}.

Here G is the number of groups, which is a pre-defined hyper-parameter ( G= 32 by default). C/G is the number of channels per group. is the floor operation, and the final term means that  *i and k* indexes are in the same group of channels, assuming each group of channels are stored in a sequential order along the C axis.

To understand GN better, let’s understand to two other normalizations methods i.e., Layer Normalization (‘LN’) and Instance Normalization (‘IN’).

Layer Normalization (‘LN’) is designed to overcome the drawbacks of BN, including its constraints on batch size. It computes 𝜇ᵢ and 𝜎ᵢ along the (C, H, W) axes, with Sᵢ defined as all the coefficients that belong to the same input feature as xᵢ. As a result, the computation for an input feature is entirely independent of other input features in a batch.

All the coefficients of a are normalized by the same 𝜇ᵢ and 𝜎ᵢ²

𝜇ᵢ = mean (2, 3, 5, 7, 11, 13, 17, 19) = 9.625

𝜎ᵢ² = var (2, 3, 5, 7, 11, 13, 17, 19) = 35.734

Therefore, applying LN to a gives

a ← [ [[-1.276, -1.108]], [[-0.773, -0.439]], [[0.230, 0.565]], [[1.234, 1.568]] ]

Instance Normalization (‘IN’) can be viewed as applying the formula of BN to each input feature (a.k.a. instance) individually as if it is the only member in a batch. More precisely, IN computes 𝜇ᵢ and 𝜎ᵢ along the (H, W) axes, and Sᵢ is defined as the set of coefficients that are in the same input feature and also in the same channel as xᵢ.

Since the computation of IN is the same as that of BN with batch size = 1, IN actually makes the situation even worse in most cases. However, for style transfer tasks, IN is better at discarding contrast information of an image, and has superior performances than BN.

For the first coefficient aᵢ = 2 of a, where i = (0, 0, 0), the corresponding 𝜇ᵢ and 𝜎ᵢ² are simply

𝜇ᵢ = mean (2, 3) = 2.5

𝜎ᵢ² = var (2, 3) = 0.25

which gives

aᵢ ← (2 - 2.5) / √(0.25 + 0.00001) = -1.000

When we apply IN to a, we get

a ← [ [[-1.000, 1.000]], [[-1.000, 1.000]], [[-1.000, 1.000]], [[-1.000, 1.000]] ]

We introduced IN as applying BN to each input feature individually as if batch size = 1. Notice that IN can also be viewed as applying LN to each channel individually as if the number of channels = 1.

Group Normalization (GN) is a middle ground between IN and LN. It organizes the channels into different groups, and computes 𝜇ᵢ and 𝜎ᵢ along the (H, W) axes and along a group of channels. Sᵢ is then the set of coefficients that are in the same input feature and also in the same group of channels as xᵢ.

The number of groups G is a pre-defined hyperparameter, which is usually required to divide C. For simplicity we group the channels in a sequential order. So channels 1, …, C / G belong to the 1st group, channels C / G + 1, …, 2C / G belong to the 2nd group, and so on. When G = C, which means each group has only 1 channel, GN becomes IN. On the other hand, when G = 1, GN becomes LN. Therefore G controls the interpolation between IN and LN.

For our example consider G = 2. To normalize the first coefficient aᵢ = 2 of a where i = (0, 0, 0), we use the coefficients of a in the first 4 / 2 = 2 channels

𝜇ᵢ = mean (2, 3, 5, 7) = 4.25

𝜎ᵢ² = var (2, 3, 5, 7) = 3.687

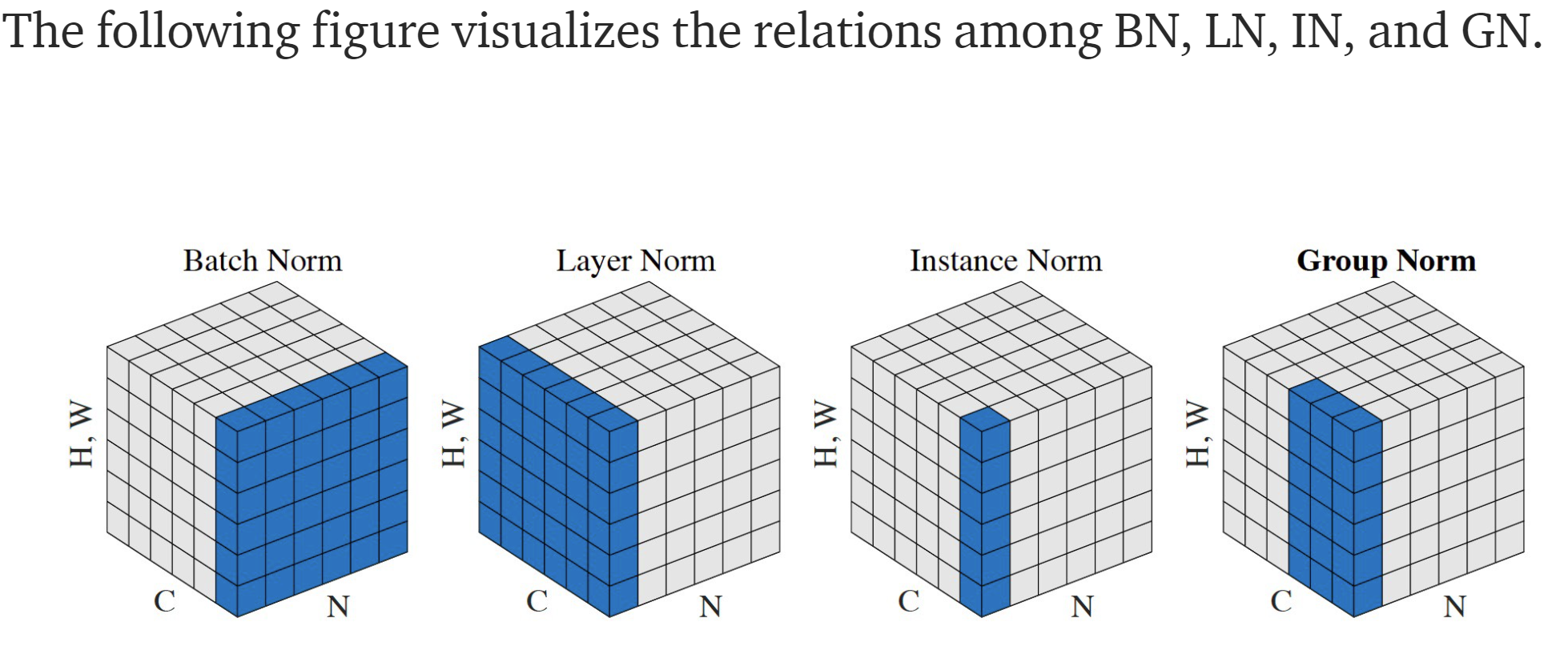
Plugging these into the normalization formula,

aᵢ ← (2 - 4.25) / √(3.687 + 0.00001) = -1.172

For other coefficients of a, the computations are similar:

a ← [ [[-1.172, -0.651]], [[0.391, 1.432]], [[-1.265, -0.633]], [[0.633, 1.265]] ]

**Comparison of Normalization Methods**



The blue regions correspond to the sets Sᵢ for the computation 𝜇ᵢ and 𝜎ᵢ, which are then used to normalize any coefficients in the blue regions.

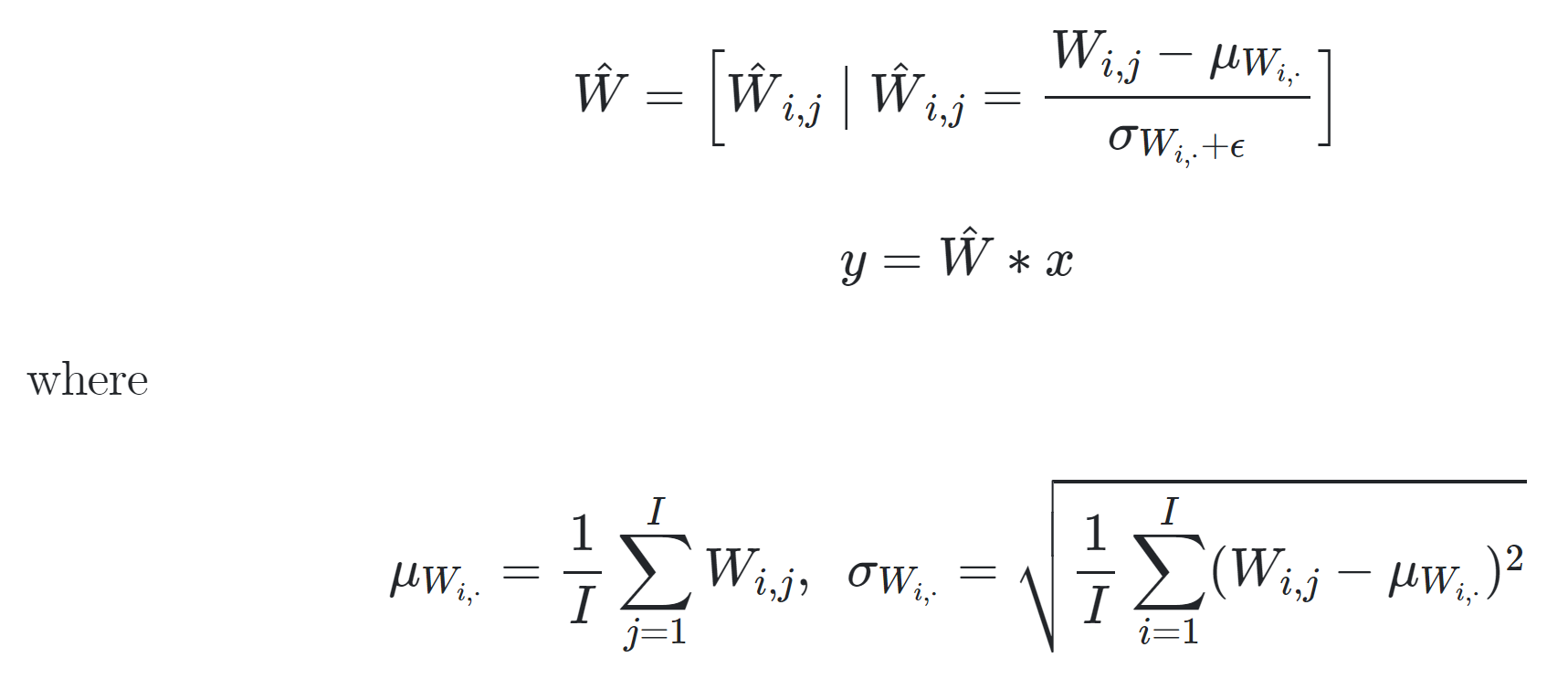
From this figure we can see how GN interpolates between IN and LN. GN is better than IN as GN can exploit the dependence across the channels. It is also better than LN because it allows different distribution to be learned for each group of channels.

When the batch size is small, GN consistently outperforms BN. However, when the batch size is significantly large, GN does not scale as well as BN and might not be able to match the performance of BN.

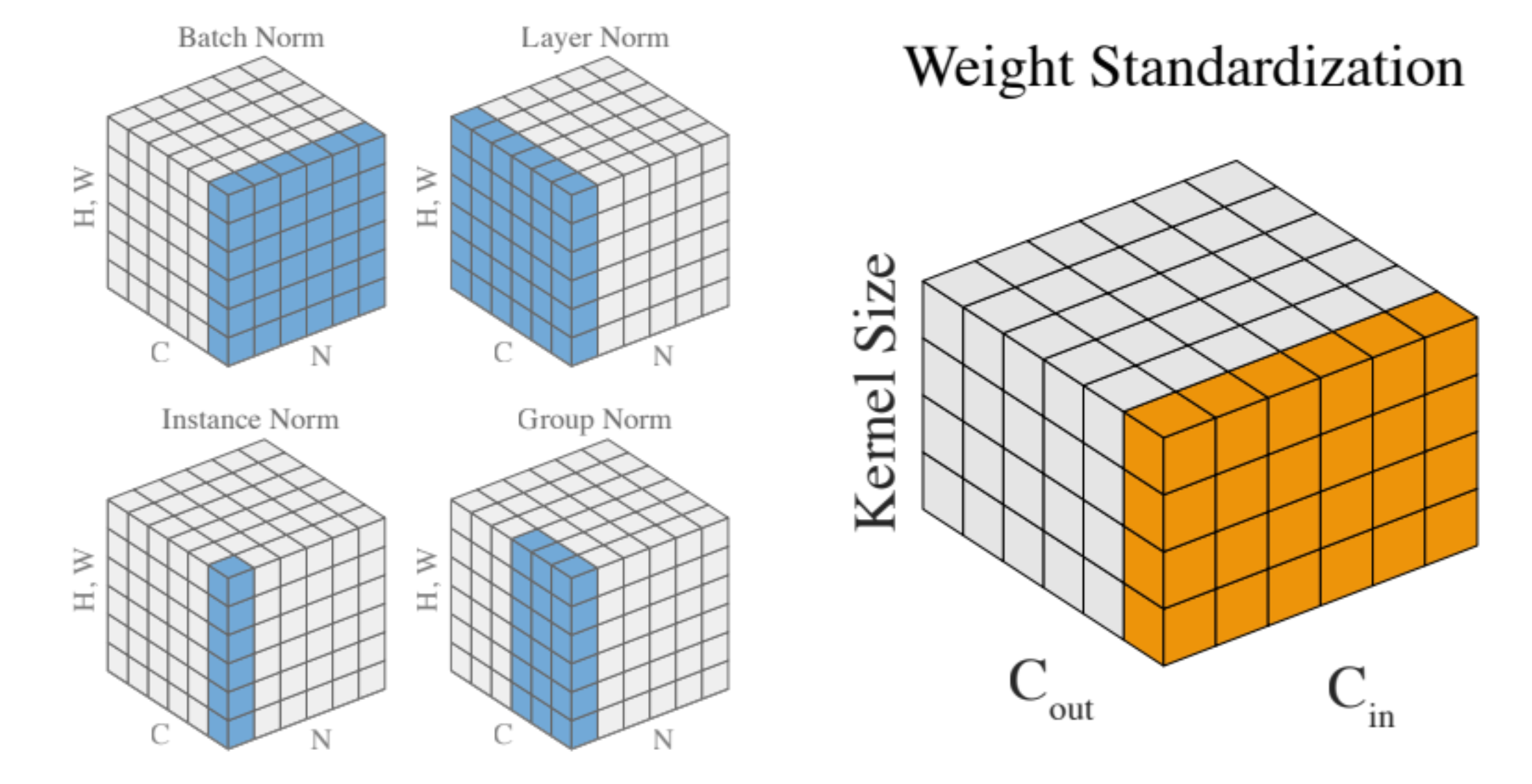
## Weight Standardization

Weight Standardization (‘WS’) is a normalization technique that smooths the loss landscape by standardizing the weights in convolutional layers. Different from the previous normalization methods that focus on activations, WS considers the smoothing effects of weights more than just length-direction decoupling. Theoretically, WS reduces the Lipschitz constants of the loss and the gradients. Hence, WS smooths the loss landscape and improves training.

In Weight Standardization, instead of directly optimizing the loss on the original weights , we reparameterize the weights as a function of , i.e. , and optimize the loss on W by SGD:



Like 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 .This is because we assume that normalization layers such as BN or GN will normalize this convolutional layer again.



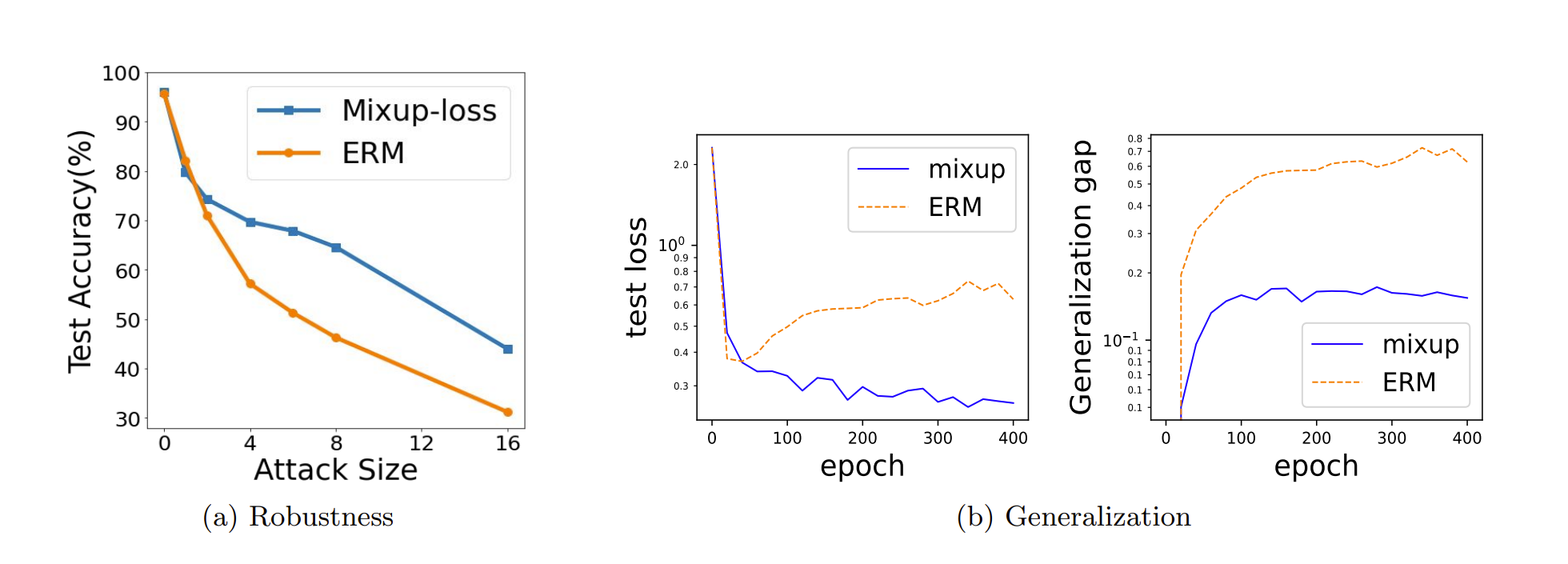
# MixUp regularization

Mixup was introduced by Zhang et al. (2018) as a data augmentation technique. It has been empirically shown to substantially improve test performance and robustness to adversarial noise of state-of-the-art neural network architectures. It has a generic and straightforward data augmentation principle. Mixup trains a neural network on convex combinations of pairs of examples and their labels. By doing so, it regularizes the neural network to favor simple linear behavior in-between training examples. Mixup’s primary motivation is to alleviate overfitting in training deep neural networks.

It is a popular data augmentation technique based on taking convex combinations of pairs of examples and their labels. This simple technique has been shown to substantially improve both the robustness and the generalization of the trained model. Mixup in training helps model robustness and generalization. Minimizing the Mixup loss corresponds to approximately minimizing an upper bound of the adversarial loss. This explains why models obtained by Mixup training exhibits robustness to several kinds of adversarial attacks such as Fast Gradient Sign Method (FGSM). Mixup augmentation corresponds to a specific type of data-adaptive regularization which reduces overfitting and thus generalization.

Adversarial robustness

Although neural networks have achieved remarkable success in many areas such as natural language processing and image recognition, it has been observed that neural networks are very sensitive to adversarial examples — prediction can be easily flipped by human imperceptible perturbations. Specifically, in Goodfellow et al. (2014), the authors use fast gradient sign method (FGSM) to generate adversarial examples, which makes an image of panda to be classified as gibbon with high confidence. Although various defense mechanisms have been proposed against adversarial attacks, those mechanisms typically sacrifice test accuracy in turn for robustness and many of them require a significant amount of additional computation time. In contrast, Mixup training tends to improve test accuracy and at the same time also exhibits a certain degree of resistance to adversarial examples, such as those generated by FGSM. Moreover, the corresponding training time is relatively modest. The model trained with Mixup loss has much robust accuracy-- when compared for robust test accuracy between a model trained with Mixup and a model trained with standard empirical risk minimization (ERM) under adversarial attacks generated by FGSM. Robustness of Mixup under other attacks have also been empirically studied in Lamb et al. (2019).



*Illustrative examples of the impact of Mixup on robustness and generalization. (a) Adversarial robustness on the SVHN data under FGSM attacks. (b) Generalization gap between test and train loss. More details regarding the experimental setup are included in* [*https://arxiv.org/pdf/2010.04819v1.pdf*](https://arxiv.org/pdf/2010.04819v1.pdf)

Generalization

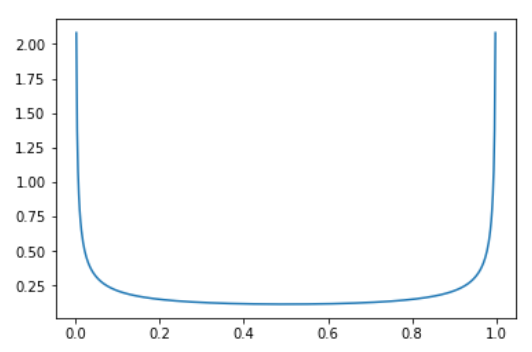
For Mixup, from figure (b) above, we observe that Mixup training results in better test performance than the standard empirical risk minimization. That is mainly due to its good generalization property since the training errors are small for both Mixup training and empirical risk minimization.

Implementation

In the original article[12](#_Reference), the authors suggested three things:

1. Create two separate dataloaders and draw a batch from each at every iteration to mix them up
2. Draw a t value following a beta distribution with a parameter alpha (0.4 is suggested in their article)
3. Mix up the two batches with the same value t.
4. Use one-hot encoded targets

Beta distribution with the same parameters alpha looks like this:



So, it means there is a very high probability of picking values close to 0 or 1 (in which case the image is almost from 1 category) and then a somewhat constant probability of picking something in the middle (0.33 as likely as 0.5 for instance).

While this works very well, it’s not the fastest way we can do this. The main point that slows down this process is wanting two different batches at every iteration (which means loading twice the number of images and applying to them the other data augmentation function). To avoid this slow down, we can be a little smarter and mixup a batch with a shuffled version of itself (this way the images mixed up are still different).

If the loss is a classic cross-entropy, we have

Loss (output, new\_target) = t \* loss (output, target1) + (1-t) \* loss(output, target2)

So we won’t one-hot encode anything and just compute those two losses then do the linear combination.

Using the same parameter t for the whole batch also seemed a bit in-efficient. Through experiments, it was noticed that the model can train faster if we draw a different t for every image in the batch (both options get to the same result in terms of accuracy, it’s just that one arrives there more slowly).  
The last trick we have to apply with this is that there can be some duplicates with this strategy: let’s say or shuffle say to mix image0 with image1 then image1 with image0, and that we draw t=0.1 for the first, and t=0.9 for the second. Then

image0 \* 0.1 + shuffle0 \* (1-0.1) = image0 \* 0.1 + image1 \* 0.9

image1 \* 0.9 + shuffle1 \* (1-0.9) = image1 \* 0.9 + image0 \* 0.1

will be the same. In practice, we could see a drop in accuracy by using this without removing those duplicates. To avoid them, replace the vector of parameters t we drew by

t = max(t, 1-t)

The beta distribution with the two parameters equal is symmetric in any case, and this way we ensure that the biggest coefficient is always near the first image (the non-shuffled batch).

# Models used in producing performance results

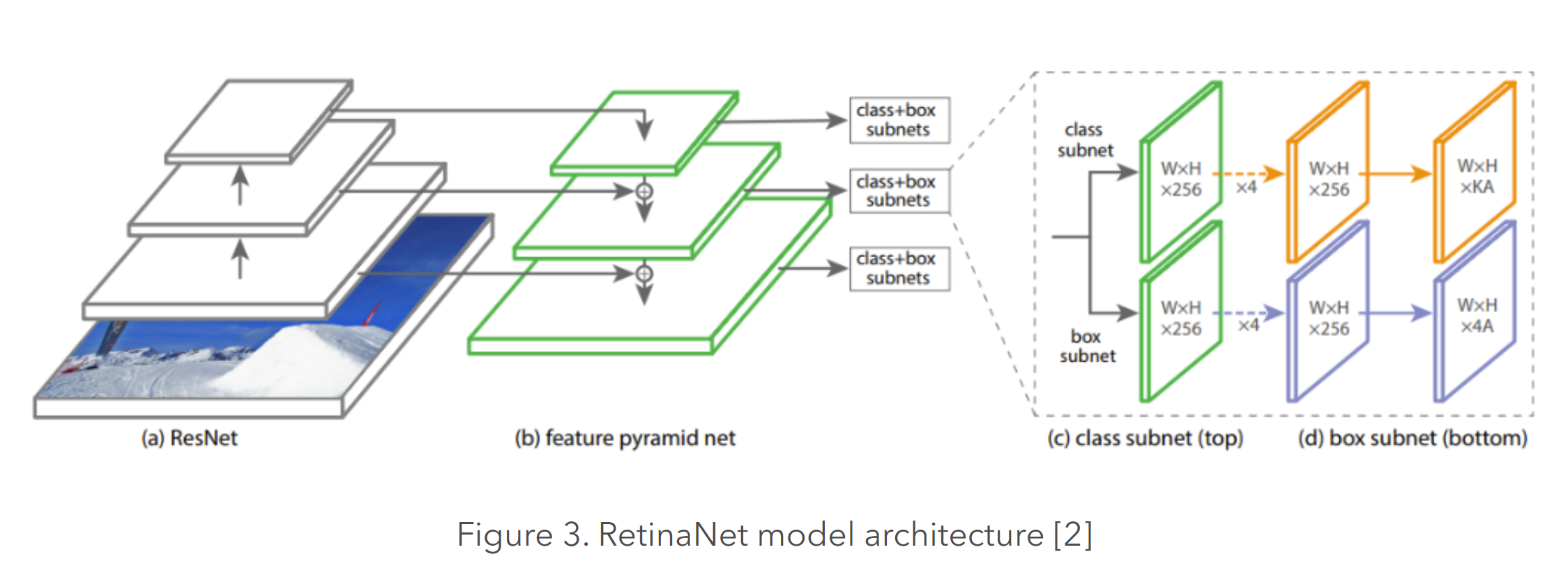
RetinaNet

RetinaNet is one of the best one-stage object detection models that has proven to work well with dense and small scale objects. For this reason, it has become a popular object detection model to be used with aerial and satellite imagery.

RetinaNet has been formed by making two improvements over existing single stage object detection models - Feature Pyramid Networks (FPN) and Focal Loss.

There are four major components of a RetinaNet model architecture:

1. 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.
2. 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.
3. Classification subnetwork - It predicts the probability of an object being present at each spatial location for each anchor box and object class.
4. Regression subnetwork - It's regresses the offset for the bounding boxes from the anchor boxes for each ground-truth object.



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). 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.

## ResNet models

Several variants of ResNet model were used for BiT experiment e.g., ResNet50, ResNet1502X4, ResNet-101X3. ResNet, short for Residual Networks is a classic neural network used as a backbone for many computer vision tasks. The fundamental breakthrough with ResNet was it allowed us to train extremely deep neural networks with 150+layers successfully. Prior to ResNet, training very deep neural networks was difficult due to the problem of vanishing gradients.

AlexNet, had only 8 convolutional layers, the VGG network had 19 and Inception or GoogleNet had 22 layers and ResNet 152 had 152 layers. ResNet-50 that is a smaller version of ResNet 152 and frequently used as a starting point for transfer learning.

In addition, as provided explained BiT (BiT-S, BiT-M, and BiT-L) model were also used for producing performance.

# Reference

1. <https://en.wikipedia.org/wiki/Transfer_learning>
2. <https://arxiv.org/abs/1912.11370>
3. <https://www.infoq.com/news/2020/06/google-big-transfer/>
4. <https://towardsdatascience.com/a-beginners-guide-to-convolutional-neural-networks-cnns-14649dbddce8>
5. <https://ai.googleblog.com/2020/05/open-sourcing-bit-exploring-large-scale.html>
6. <https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c>
7. <https://mlexplained.com/2018/01/10/an-intuitive-explanation-of-why-batch-normalization-really-works-normalization-in-deep-learning-part-1/>
8. https://paperswithcode.com/method/group-normalization
9. <https://ai.googleblog.com/2020/05/open-sourcing-bit-exploring-large-scale.html>
10. <https://arxiv.org/pdf/2010.04819v1.pdf>
11. <https://deepai.org/publication/on-mixup-regularization>
12. <https://arxiv.org/abs/2006.06049>