Part 1

My Kaggle ID: mungsoo

Best accuracy: 0.604

Network Architecture

I change the BaseNet to a shallow ResNet. The basic block of my ResNet is Bottleneck layer. As shown below

Layer No.	Layer Type	Kernel Size	Input | Output Channels (for conv layers)
1	Bn2d	-	A | A
2	Relu	-	A | A
3	Conv2d	1	A | B
4	Bn2d	-	B | B
5	Relu	-	B | B
6	Conv2d	3	B | B
7	Bn2d	-	B | B
8	Relu	-	B | B
9	Conv2d	1	B | A / B | 2 * A

Bottleneck consists of three basic conv layers. The first one is a 1x1 layers used to decrease the input channels. It helps to increase the non-linearity which is helpful to build deep networks, and to decrease the number of operations significantly. The second conv layer is a 3x3 conv layer. It was firstly proposed by VGG, said that the 3x3 conv layer, which is the smallest size of kernels, reduces the number of parameters significantly compared to the larger conv layers with the same receptive field.

The stride of the second conv layer can be 1 or 2, which maintain the output resolution and downsample it by 2 respectively.

The number of channels is A and B. If A is 4 times B, then the output channel of Bottleneck is same as input. If A is not 2 times B, then the number of output channel is increased to 4 * B.

There is also an identity mapping of the input added to the output.

Notice that this implementation of Bottleneck uses full pre-activation as addressed in https://arxiv.org/pdf/1603.05027.pdf (He at el.). It has the best performance in their experiment. It introduces an identity mapping from the very beginning to the output without any interference. The gradient of such architecture is also easier to compute and more stable (unlikely vanishing).

The ResNet architecture is shown below

Layer No.	Layer Type	Kernel Size	Input | Output dimension	Input | Output Channels (for conv layers)
1	Conv2d	3	32 | 32	3 | 64
2	Bn2d	-	32 | 32	64 | 64
3	Relu	-	32 | 32	64 | 64
4	Bottleneck	-	32 | 32	64 | 128
5	Bottleneck	-	32 | 32	128 | 128
6	Bottleneck	-	32 | 32	128 | 128
7	Bottleneck	-	32 | 16	128 | 256
8	Bottleneck	-	16 | 16	256 | 256
9	Bottleneck	-	16 | 16	256 | 256
10	Bottleneck	-	16 | 8	256 | 512
11	Bottleneck	-	8 | 8	512 | 512
12	Bottleneck	-	8 | 8	512 | 512
13	Bn2d	-	8 | 8	512 | 512
14	Relu	-	8 | 8	512 | 512
15	Avgpool2d	-	8 | 1	512 | 512
16	Fc	-	100	-

Factors helped to improve performance and ablation studies

Batch Normlization: It improves the network performance a lot. Firstly, I added several more conv layers to the initial network and without BN. The result acc was about 0.32. Then I added BN after every layer, except for the one before the last fc layer (I forgot to put one there). The network was still hard to converge and acc had no improvement. Then I fixed the bug and found the performance had a huge improvement, reach 0.44. It shows that BN is very important as it addresses the internal covariate shift problem. After BN, the input of each layer has the same distribution. And the activation, like sigmoid, can avoid saturate after BN.

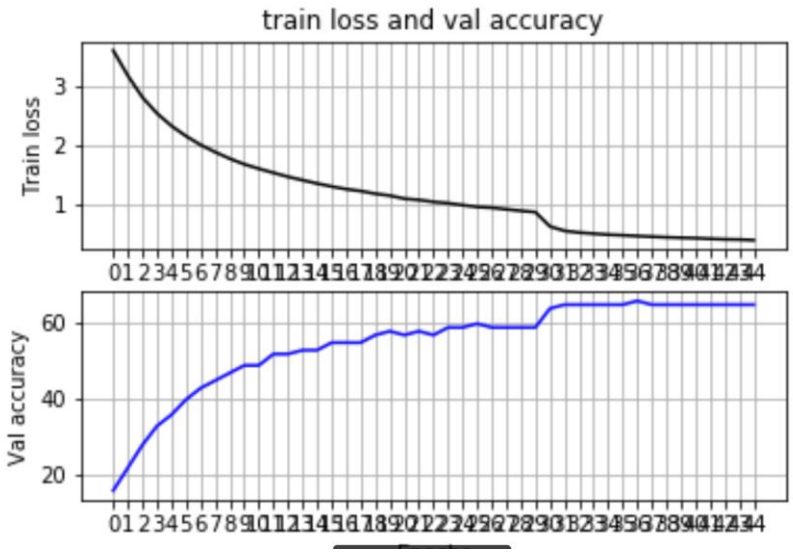
Data augmentation and normalization: Data augmentation helps to improve the generalization of the model and alleviate the overfitting problem. After BN, data normalization can be omitted as they are doing the same thing but BN is more powerful. However, I still normalized the input. I tried to train the network without data augmentation and noticed overfitting at the end of the training process.

Deeper Network: A shallow and naïve network can easily reach 0.50. A deeper architecture can reach over 0.60. I do not add extra fc layers as it is computationally intensive. I use 1x1 conv layers to reduce channel dimension. All these efforts help to reduce the size of the network. I don't add more layers to the network as it takes much time to train.

Dropout and weight decay: I do not use dropout as it slows down the training process and do not improve acc. I use 0.0001 weight decay to alleviate overfitting.

Training technique: I tried Adam and SGD and find SGD converges to a better result. I use 0.1 initial learning rate and multiple it by 0.1 every 30 epoches. I noticed a huge accuracy improvement as the learning rate got smaller (generally 0.03~0.06).

Loss and acc:



Extra credit:

As stated above, I implemented a simple version ResNet(refering https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py). Then I adapt the full pre-activation raised by He et al.

I use 1x1 conv layer to reduce computation and to increase network depth. I use global average pooling raised by M Lin et al. instead of adding fc layers.

I also use several training techniques like weight decay and learning rate decay, so that the network can converge faster and have a better result.

Part 2

The train acc I got by using ResNet as a fixed feature extractor is 0.83.

The test acc I got by using ResNet as a fixed feature extractor is 0.48.

Batch size is 32, learning rate is initialized with 0.01 and divided by 10 every 20 epoches. Weight decay is set with 0.0001. The data is augmented with RandomResizedCrop(224), RandomHorizontalFlip() and normalized with Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]).

The train acc I got by fine-tuning the whole network is 0.93.

The test acc I got by fine-tuning the whole network is 0.54.

Batch size is 32, learning rate is initialized with 0.01 and divided by 10 every 20 epoches. Weight decay is set with 0.0001. The data is augmented with RandomResizedCrop(224), RandomHorizontalFlip() and normalized with Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]).