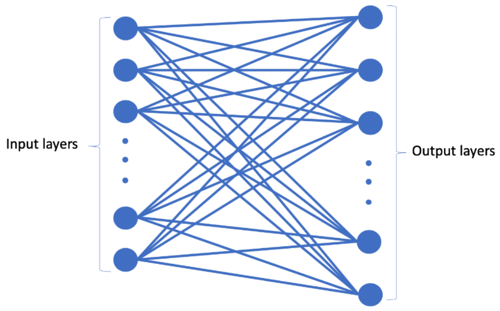
Homework 5 for 20 spring EE569
CNN Training On LeNet-5
Chengyao Wang
69615991816
chengyao@usc.edu

Problem 1, CNN Architecture:

1. Fully Connected Layer:
   a. *FC* layer in neural network inherited from *multi-layer perceptron (MLP)* and got its name "*fully connected*" because every input & output neuron are pairwise connected in most cases, except for cases where weights are zero.



   b. Computation done in *fc* layers, apart from activation functions, could be generalized into matrix multiplications and thus weights in *fc* layers can be represented by *linear operators* or *matrices*. The number of total parameters in *fc* layers are:
$$\#_{Input-Neuron} \times \#_{Output\_Neuron}$$
   And the computations:
$$\overrightarrow{w_{output}} = f(W^{M\times n} \times \overrightarrow{w_{input}})$$
   c. ***fc* layers are often located at the end of the network model in forward computation, playing the role of *decision makers (discriminators)* – utilizing upstream data to perform tasks like classification & regression.** There are some works being done in computer vision areas where people truncate the *fc* layers and train *kernel machines* or *random forests* on top of previous, intact layer's outputs. And they received similar performances.
2. Convolutional Layer:
   a. **Convolutional layers in general plays feature extraction role in the network, thus they generally appear at the early stage of the forward pass.**
   b. **In facing image data, conv layers also substantially lowered the number of parameters originally introduced by *fc* layers. Also according to LeCun[1], conv layers are also good at preserving local spatial information as opposed to *fc* layers.**
   c. We can find many similarities between *conv* layer computation & traditional computer vision feature extractors or more generally operators like *gaussian denoising filters & Lowe's filters*. Their biggest difference lies in trainability of *conv* filters. Filter weights in *conv* layers are learned by gradient descent under the dataset's instruction, while the latter are often guided by *Fourier & frequency domain analysis*.
   d. Also in addition, one convolution filter in deep learning are often defined as an aggregation of traditional 2D kernels, thus they convolve over all input channels and often have 3 dimensions:
$$conv_i = W^{Input-channel\times K\times K}$$
   One such kernel produce 1-channel response map and we often have many *conv* kernels in one *conv* layer to produce multi-channel outputs.
3. Max Pooling:
   a. **Max pooling are often used to perform dimension reduction along the 2D spatial wise response map, and they are also believed to increase the model's robustness to affine transformations.**
   b. Convolutional kernels with stride 1 have limited ability down-sampling responses to feed to *fc* layers for classification. Thus, pooling is introduced to conduct efficient feature reduction.
   c. Pooling layers works quite the same as conv layers, expect that they are no longer restricted to linear, numerical operations and they are not trainable. We swipe 2D window across the response map (they're often do not overlap) and do some operations.
   d. There are many variants of pooling, but *max-pooling* is used most, especially in computer vision tasks. Since we are looking for certain patterns by convolution operations, we could use the strongest response to represent its neighborhood which naturally justifies *max-pooling*.
   e. Worth noticing that dimension reduction is a must in image-based tasks, but *max-pooling* is not. ResNet[2] used stride 2 *conv* layers to conduct the same task. We could also introduce *dilation*[3] to enlarge the receptive field of *conv* kernels which also another choice of dimension reduction.
4. Activation function:

[1] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324
[2] Kaiming He, et al. "Deep Residual Learning for Image Recognition" 2015, https://arxiv.org/abs/1512.03385
[3] Fisher Yu, Vladlen Koltun. "Multi-Scale Context Aggregation by Dilated Convolutions", https://arxiv.org/abs/1511.07122

   a. **Activation function is the key for neural networks to approximate non-linear functions. It also supports the layers from collapsing because all layers their selves are linear operations.**
   b. Though the necessity of the activation functions is never doubted from the very birth of *multi-layer perceptron (MLP),* which to implement has evolved along with the understanding of neural networks & empirical experiences. People refer to "*sigmoid / tanh*" from the beginning, but the two suffers from gradient saturation & gradient vanishing problems. Now people use "*ReLU*"s as a default choice, which kind of alleviate the problem and also makes back propagation process much faster.
5. Softmax Function:
   a. **Softmax function are often located at the very end of a classification network to turn per-class response into per-class probabilities. To the best of our knowledge, it has at least the following merits:**
      i. Works in perfect consistency with one-hot vector label encoding methods. And also restrains the range of output vector to $[0,1]^N$ which allows for many other tricks, e.g. label smoothing[4].
      ii. Probability distributions is compatible for a much wider range of loss functions other than L-p losses, e.g. *cross entropy used in most classification tasks, KL divergence used in Variational Auto Encoders (VAE) and Wasserstein metric in Wasserstein GAN[5].*

$$CrossEntropy = \sum p_i ln q_i$$
$$KL - Divergence = \sum p_i ln \frac{p_i}{q_i}$$
$$Wasserstein\ Metric^{6} = [infE[d(P,Q)^p]]^{1/p}$$

      iii. Softmax function also is strongly related to *Bayes' Theorem[7]* :

$$p(w_0|x) = \frac{p(x|w_0)p(w_0)}{p(x|w_0)p(w_0) + p(x|w_1)p(w_1)} = \frac{1}{1 + exp(-ln(\frac{p(x|w_1)}{p(x|w_0)}\frac{p(w_0)}{p(w_1)}))} = \frac{1}{1+e^{-f(x)}}$$

      $f(x)$ here is our neural network, and it's named *log-likelihood ratio* in statistics. We presented the 2-class case, but it's generally the same in multi-class situations.
   b. $Softmax/\ Cross\ entropy$ pair is widely used in classification networks. But in fact, such combinations first appeared in *Logistic Regressions*, and that's why the latter is later named "single layer neural network".
6. Overfitting.
   a. Though to our personal opinion, overfit in machine learning & deep learning are not exactly the same. **But generally speaking, overfit refers to the situation where the model achieves a substantially higher training accuracy than test accuracy.** Intuitively speaking, we say the network is memorizing the training data rather than finding the underlying distribution, which gives low predictive power outside the training set.
   b. **From *VC-theory*, inappropriately larger model capacity is the necessary condition for overfit to exist when the dataset we have to train the model is insufficient to restrain all the degrees of freedom in a parameterized model**. The "*free*" parameters are the source of performance degradation, or in VC-theory "deterministic noise". Thus in statistical learning, we often talk about overfit w.r.t to model capacity, while in deep learning, epochs. People are recently aware of this topic and are trying to approach this problem starting from why DNN could still achieve such a great performance with excessive model capacity[8].
   c. Common tricks to prevent overfitting include:
      i. *Early Stopping*. We monitor network's performance using the test set performance. If the performance starts to degrade, we stop training.
      ii. *Weight Decay.* It degrades the coefficient of weights in the weight update function, and can be proved being mathematically equivalent to L2-regularization:

$$W_t^{m+1} = (1-\eta\alpha)W_t^m - \alpha\Delta W_t$$

      iii. *Batch Normalization.* Batch normalization (layers) mainly address the gradient vanishing / explosion problems in DNN by manually restraining the responses to zero-mean, unit-variance across batches. On the other hand, by restraining the responses, the weights could also be regularized.
      iv. *Dropout.* Restrain effective model capacity every epoch during training. Usually applied to *fc* layers.
      v. Choosing network structure comparable to task & dataset at hand.
7. **The advantages of CNN as opposed to traditional methods to the best of our knowledge include:**
   a. *Large enough model capacity*. Necessary condition for getting a good performance. **This includes powerful enough feature extractors (numerous convolution operators) and discriminators.**
   b. *Adaptive feature extractors*. The feature weights are learned from the dataset given, thus is more "*suitable*" for the task we're facing.
   c. *Large dataset*. Deep learning models are often provided with large dataset.
8. Loss functions & backpropagation:
   a. **Loss functions are metrics that quantitatively describe how well our model is doing upon the training set. It has to be differentiable because network learns by its gradients. They are often convex functions and is closely related to accuracy in classification tasks.**
   b. The learning process of neural networks is an optimization problem – we are searching through the parameters space looking for the point where the model behaves the best on the ***training set***. Traditional convex optimizations have both 1st order & 2nd

[4] https://towardsdatascience.com/label-smoothing-making-model-robust-to-incorrect-labels-2fae037ffbd0
[5] Martin Arjovsky, Soumith Chintala, Léon Bottou. "Wasserstein GAN", https://arxiv.org/abs/1701.07875
[6] https://en.wikipedia.org/wiki/Wasserstein_metric
[7] https://crazyoscarchang.github.io/2018/08/29/why-the-softmax-function/
[8] Preetum Nakkiran, et al. "Deep Double Descent: Where Bigger Models and More Data Hurt" 2019, https://arxiv.org/abs/1912.02292

order approaches but it's long been a consensus across the community that the complexity of the problem makes only 1st order feasible, which is *gradient descent*. Its iterative characteristic also makes it an automatic procedure on computers.

c. **Backpropagation is a way to backpropagate error gradient to every layer of the network. It can be mathematically described as chain differentiation.** In *fc* layers, the forward pass shall be:

$$\overrightarrow{x_{t+1}} = f(W_t{}^T \times \overrightarrow{x_t} + \overrightarrow{b_t})$$

And thus the backpropagation of error shall be:

$$\overrightarrow{err_t} = [f^{`}(W_t \times \overrightarrow{x_t} + \overrightarrow{b_t}) \odot W_t] \times \overrightarrow{err_{t+1}}$$

Where $\odot$ is the Hardmard product. We are approximating non-linear activation functions with its tangent line at each element of $W_t \times \overrightarrow{x_t} + \overrightarrow{b_t}$. We could update the weights in $(W_t, \overrightarrow{b_t})$ as:

$$\Delta W_t = \frac{\alpha}{2} \overrightarrow{x_t} \times \overrightarrow{err_{t+1}}^T$$
$$\Delta b_t = \frac{\alpha}{2} \overrightarrow{ones} \times \overrightarrow{err_{t+1}}^T$$

We could observe some possibilities for memory optimization, thus no matter the size of the network, we could still iteratively backpropagate the error gradient & train it. Situations are similar for *conv* layers, there the error gradient in each point of the 3D response map $\overrightarrow{X_t}$ will be contributed by the all related point in $\overrightarrow{X_{t+1}}$ during forward pass.

d. Batch normalization layers, dropouts & pooling layers have their own small tricks during *BP* process.
   i. Coefficients in *BN* layers are not trainable. Since the layer applies a simple linear transformation on each response point respectively, *BP* is trivial.
   ii. *Dropouts*. All the neurons keep their on/off status during *BP*, that simply means some $\overrightarrow{x_t}$ are 0. It doesn't even affect the update formula.
   iii. Pooling. Error only propagates into the chosen point which is the largest during forward pass for *max-pooling* cases. This is done by recording the indices every iteration respectively, which is both time & memory efficient. *Average pooling* is the equivalent to *conv* kernels.

e. There are now 2nd order update method being forwarded[9].

Problem 2, CIFAR-10 Classification:

Here are some comments of the approaches to this problem:

1. We're using PyTorch to construct and train the model.
2. We used ***GTX 1060 6G*** to accelerate training process, to ~ 2.6s / epoch with batch size 128. At first, the speed of CPU feeding the dataset batch to GPU became the bottleneck, and we achieved the speed above by involving all cores of CPU – specifying the number of workers of ***pytorch.dataset.DatasetLoader***. We could prefetch all the dataset into GPU, but *pytorch* doesn't provide this functionality by default thus we are not applying such method.
3. All the specific configurations, including network structure, optimizer & learning rate schedule and training accuracy & test accuracy are all manually organized & written into a JSON file upon completion of the whole process.
4. The script will generate three files: ***saved model (model.state_dict() format), train_acc / test_acc / loss to epoch plots and JSON file mentioned above***. They share the same name. We record the starting time of the script to be the ID of this particular execution, and the three files take same ID.
5. We defined a default configuration, upon which we apply slightly different parameters settings:

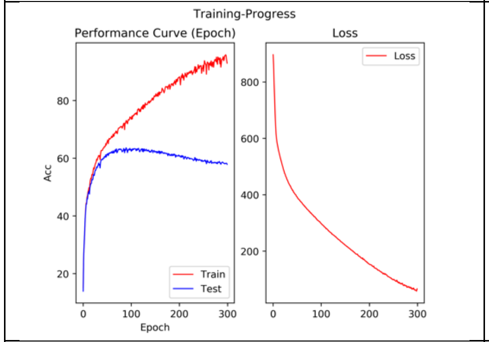
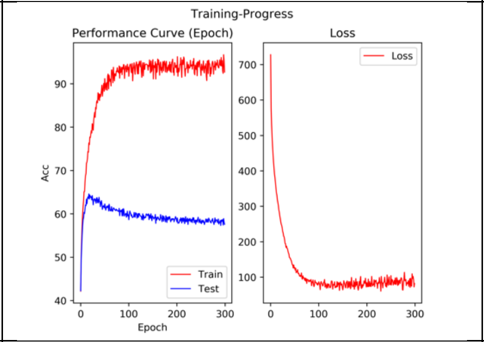
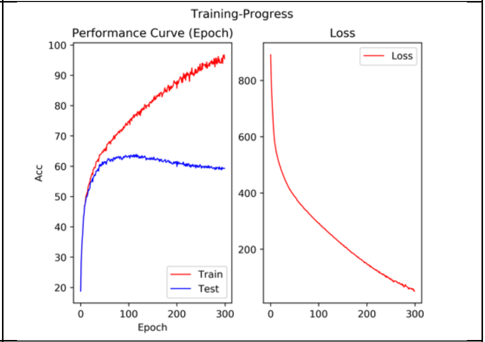
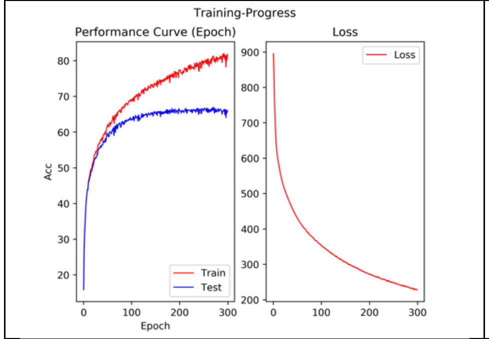
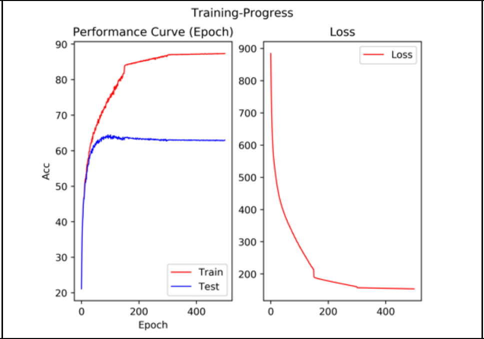
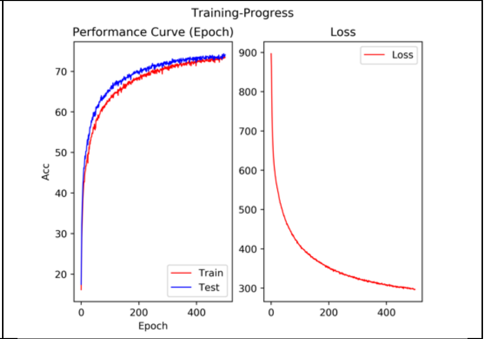
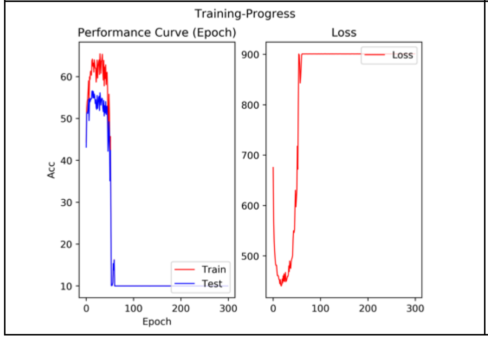
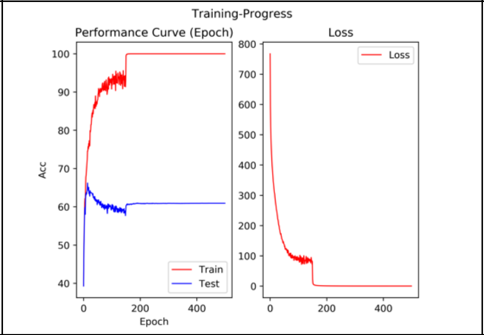
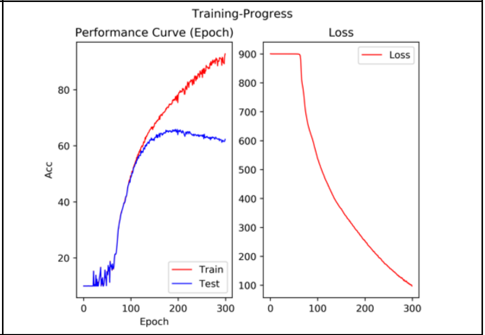
	Default	**Setting(a)**	**Setting(b)**	**Setting(c)**	**Setting(d)**	**Setting(e)**	**Setting(f)**
Weight Decay	0.0	0.0	0.0	**5e-3**	0.0	0.0	0.0
Initial Learning Rate	0.001	**0.1/0.01**	0.001	0.001	**0.001**	0.001	0.001
Initialization (*fc*)	XavierUni(1)	XavierUni(1)	**XavierNorm(1)**	XavierUni(1)	XavierUni(1)	XavierUni(1)	**XavierUni(.1)**
Batch Size	128	128	128	128	128	128	128
Data Normalization (μ, σ)	(0.5, 1.0)	(0.5, 1.0)	(0.5, 1.0)	(0.5, 1.0)	(0.5, 1.0)	(0.5, 1.0)	(0.5, 1.0)
Total Epoch	300	300	300	300	**500**	**500**	**300**
Learning Rate Schedule	Fixed	Fixed	Fixed	Fixed	**[150, 300], 0.1**	Fixed	Fixed
Dataset Augmentation	None	None	None	None	None	**Crop+Flip**	**None**
Optimizer	Stochastic gradient descent (Momentum = 0.9)						
Activation Function	ReLU						

a. "*None*" in dataset augmentation includes *pytorch.ToTensor()* & *Data Normalization* (μ, σ).

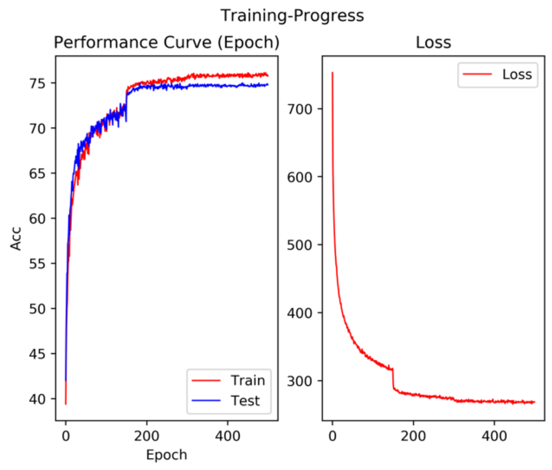
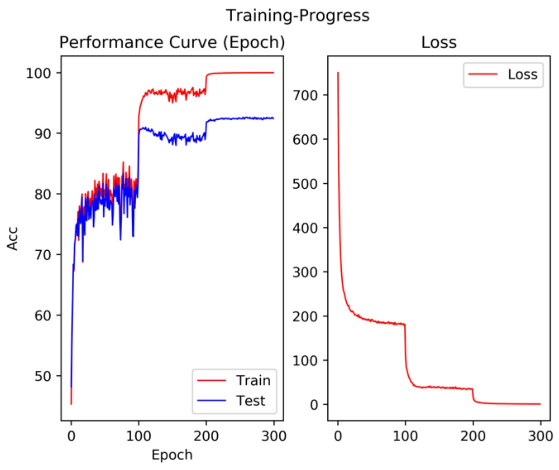
[9] Jacob Rafati, et al."Quasi-Newton Optimization Methods For Deep Learning Applications", https://arxiv.org/abs/1909.01994

b. We are NOT monitoring test (valid) accuracy to apply early stopping techniques to prevent overfitting. We have no need to save the best performing model throughout the process. The increase of the epochs is required under some configurations because its models require more time to converge.
c. We just left initialization of *conv* layers to its default way.
d. Activation functions are not changed because its related to network structure, not parameters settings. And to the best of our knowledge, there are no striking new activation functions introduced.
e. We constructed ResNetv1 of all depth variants and trained ResNet18. Best accuracy on CIFAR10 achieved reported ~92.50%.

Result

Default	Setting(a) (lr=0.01)	Setting(b)
		
Setting(c)	Setting(d)	Setting(e)
		
	Other Trials	
Setting(a) (lr=0.1)	Setting(d) (init_lr=0.01)	Setting(f)
		

Best Performing Model & ResNet18v1

Multi-Stage Learning Rate + Data Augmentation	ResNet18v1
	

	Best Train Accuracy	Final Train Accuracy	Best Test Accuracy	Final Test Accuracy	Loss (Final)
Default	95.894%	92.952%	63.64%	57.89%	66.486
Setting(a), lr=0.01	96.714%	92.702%	64.6%	57.59%	82.141
Setting(b)	96.804%	95.436%	63.93%	59.29%	50.951
Setting(c)	81.994%	81.834%	67%	66.11%	228.277
Setting(d)	87.378%	87.342%	64.49%	62.98%	153.65
Setting(e)	73.68%	73.444%	74.38%	74.16%	296.472
Setting(f)	92.814%	92.814%	65.87%	62.34%	97.329
ResNet18v1	99.998%	99.992%	92.64%	92.42%	0.890
Best Model	76.142%	75.778%	75%	74.81%	268.720

Some Comments & Analysis:

1. Reference parameters setting are guided by previous experiences and the target model we're facing.
   a. **Weight decay (5e-3)**. Weight decay usually takes on $0.1^k, k = 4, 5, 6, 7$. Weight decay too large will make the training process hardly progress.
   b. **Learning Rate (0.001)**. Learning rates often takes values on $0.1^k, k = 1, 2, 3, 4, 5, 6$, and larger models often generally needs larger learning rate. LeNet5 is rather small, thus the initial values are set to 0.1^3.
   c. **Initialization**. We tried standard normal / uniform distribution initialization tricks, but the model is found to be not learning in the few epochs. The most impact initialization has on model training could be the ***magnitude of the weights***. The values should be on relatively the same scale of the input, i.e. the standardization of the input image. *BN* also kind of deals with this problem to some extent.
   d. **Batch size**. We're using GPU, and first trained on 64. But we later changed to 128, and batch feeding speed by CPU again became the bottleneck.
   e. **Optimizer**. SGD is used for all parameter settings. It's one of the simplest optimizing strategy but is the most suitable for customizing learning rate schedule because the learning rate in other optimizers are coupled with other variables. There are more novel learning rate tricks, e.g. cosine decay, exponential decay or warm start.
2. ***Default*** **and other setting that doesn't include overfit preventing tricks all exploits typical patterns of overfitting.** As the epoch goes, training accuracy always goes up and the test accuracy first rises then slightly decreases.
3. **Large Learning rate is leading to oscillation & instability in training**. For example, ResNet18v1 we trained takes ***[0.1, 0.01, 0.001]***, but as we can see, the first two leads to obvious instability for LeNet5, model is not even converging under ***0.1***.
4. **We could observe foundations to support 1(c), which is related to initialization. Setting(f)** is generating 1 order of magnitude smaller of weights than input data, and it leads to instability in the new few epochs.

$$W_t^{m+1} = W_t^m - \frac{\alpha}{2}\overrightarrow{x_t} \times \overrightarrow{err_{t+1}}^T$$

Making $||W_{i\cdot}^m|| \approx ||\overrightarrow{x_t}||$ will obviously promises more stable training process, because the error gradient always has the same magnitude. **And we don't observe big difference between *Xavier uniform & Xavier normal* (Default & (b)).**

5. **Weight decay alleviates the overfit issue and generates a slightly better accuracy (~3%).** We do not observe a degradation of test accuracy at least within 300 epochs.
6. **(For LetNet5) Changing learning rate schedule seem not to have a positive impact on best test accuracy but seem to be combat overfit by a slight better final accuracy (~2%).** Because smaller learning rate has limited parameter search ability and thus less risk to overfit. But different learning rate is essential for larger models such as ResNet18 showed above. Decrease of learning rate leads to boost of accuracy every time.

7. **Data augmentation is the most effective way of boosting performance in this case (~10%+).** Though we are applying rather simple augmentation types, LeNet5 exploited ***underfitting*** in this scenario where it cannot fully consume the (augmented) dataset. The training accuracy are not rising to 100% anymore. In the **best model** case, test set accuracy is even higher than training accuracy during early epochs because the model is learning partially from another *virtual "dataset"* which is a different dataset.
8. **LeNet5 is over parameterized w.r.t original CIFAR-10, but not much. But overparameterization doesn't mean better performance.** Such result is kind of denying the possibility for further boosting performance under such model / dataset pair.

Problem 3, State-of-the-art CIFAR-10 Classification – EfficientNet[10]:

This method comes the 4th in a quite up-to-date leaderboard[11] with reported accuracy at 98.9%. The models that come before it involves advanced, carefully designed transfer learning methods or GPU-architecture optimized ResNet. They are all published in 2018 ~ 2020.

Neural Architecture Search (NAS)[12]:

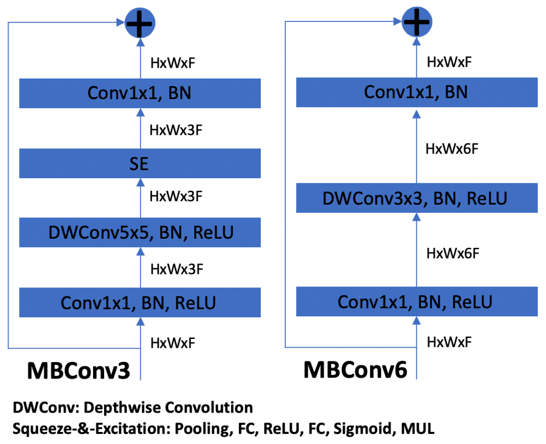
NAS is a subfield of *AutoML* where people are dedicated to design automatic deep learning systems. *NAS* focuses on hyperparameter space searching, especially *ANN* architectures. Three of the most important topics are:

1. **Search space**. The ensemble of all combinations of possible architecture designs that we are facing, possibly infinite.
2. **Search Strategy.** Strategies to explore over the search space. Since the search space scales up exponentially with the structure interested, possibly infinite, a good search strategy is essential for time efficiency & output result. It could be formulated into a RL problem.
3. **Performance Estimation.** Neural network generally takes much time to train & evaluate, thus we are usually don't apply cross validation like in machine learning. Thus, how to efficiently & accurately estimate the performance of our current model **without constructing & training it** is also quite critical.

As we can see, most of the efforts are devoted to lowering the time complexity of this problem. *Nested parameter + hyperparameter search* in tedious in time consumption w.r.t current hardware computing resources. *NAS* method produces the baseline *"EfficientNet-B0"* for downstream scaling method inspired by MnasNet[13]. It's proposed for mobile proposes, thus is small in size and suitable for scaling.

EfficientNet-B0 (#FLOPS=400M)

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224x224	32	1
2	MBConv1, k3x3	112x112	16	1
3	MBConv6, k3x3	112x112	24	2
4	MBConv6, k5x5	56x56	40	2
5	MBConv6, k3x3	28x28	80	3
6	MBConv6, k5x5	14x14	112	3
7	MBConv6, k5x5	14x14	192	4
8	MBConv6, k3x3	7x7	320	1
9	Conv1x1 & Pooling & FC	7x7	1280	1



EfficientNet:

EfficientNet first uses *NAS* similar to [13] to generate *"EfficientNet-B0"* and then scales it up using novel **compound scaling method**. High level speaking, [10] scales up width, depth & input data resolution uniformly by increasing ϕ:

[10] Mingxing Tan, Quoc V. Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", https://arxiv.org/abs/1905.11946
[11] https://paperswithcode.com/sota/image-classification-on-cifar-10
[12] https://en.wikipedia.org/wiki/Neural_architecture_search
[13] Tan, et al. "MnasNet: Platform-Aware Neural Architecture Search for Mobile", https://arxiv.org/abs/1807.11626

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

It produces *"EfficientNet-B1"* ~ *"EfficientNet-B7"* by different ϕ. $\{\alpha, \beta, \gamma\}$ are determined using simple grid search after *"EfficientNet-B0"* is determined. It searches for the best parameter sets that can achieve the highest accuracy under some constraints such as memory occupation, FLOPs & $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$. The last equation is possibly constructed to expand the model size every time by times of 2. [10] presented the best parameter set $\{\alpha, \beta, \gamma\} = \{1.2, 1.1, 1.15\}$. And quantization methods like flooring or ceiling is done to apply d, w, r to the network. The largest model *"B7"* has 64M parameters, in comparison, ResNet18v1 has 11M.

Comparison:

LeNet5 is the pioneer of CNN, especially convolution layers. But EfficientNet is introduced by Google Brain in 2019. The rapid, wild growth of deep convolutional network has introduced many tricks to climb the leaderboard of famous datasets other than merely scaling deeper, including something like Residual / Highway block to avoid training problems, all kinds of optimizers & learning rate schedules, transfer learning, *NAS* etc. Under such perspective, LeNet5 seems more like a prototype. AlexNet could be taken as scaled version of LeNet5, with no novel modern tricks, is reported ~74%, which is also not comparable to the smallest ResNet. **LeNet5 of course is smaller in size, thus have obvious advantages in the simplicity of training methods, memory occupancy & inference time. But as shown in the results above, LeNet5 has limited ability in achieving a better accuracy on CIFAR10 dataset. Pros-and-Cons of EfficientNet introduced above is obviously just the opposite, thus is not repeated anymore.**