ECE 661 COMP ENG ML & DEEP NEURAL NETS

6. CNN TRAINING – ADVANCED

YIRAN CHEN, SPRING 2024

Babysitting the CNN training

Lecture 5: Basic techniques

- Training overview
- Neural network design
  - General architecture
  - Activation functions
  - Weight initialization
- Loss function
  - Optimizer

Lecture 6: Advanced techniques

  - Regularization
  - Data preprocessing
  - Hyperparameter tuning

Babysitting the CNN training

Lecture 5: Basic techniques

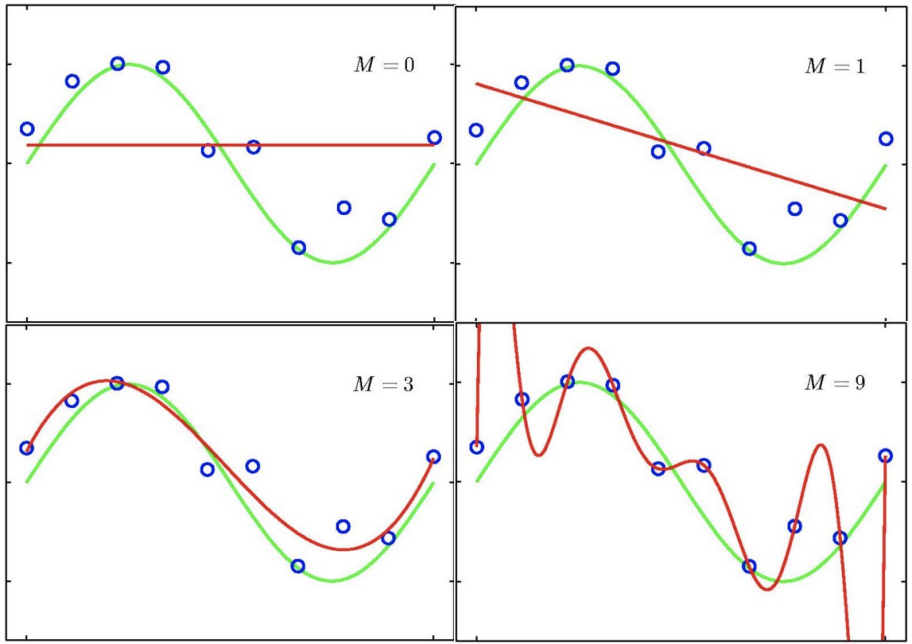
Lecture 6: Advanced techniques

- Regularization
- Data preprocessing
- Hyperparameter tuning

Regularization: overfitting issue

What is regularization?

Regularization constraints the weight values of a model and prevents it from being too well on the training data.



M[th] order polynomial fit

- $M = 9$ gives a terrible overfitting result.
- By applying regularization, we observe that high-order coefficients of the polynomial fitting are regularized to 0, and a 3[rd]-order polynomial is sufficient to do a good job in generalizing data.

L-norm regularization

How to apply regularization?

$$l(W) = CE(X, Y; W) + \lambda R(W)$$

Cross entropy loss Regularization loss

L-norm regularization is the most popular regularization method in DNN training.

- **L1 (Lasso) regularization**

$$R(W) = \sum_{i,j} |W_{ij}| = \|W\|_1$$

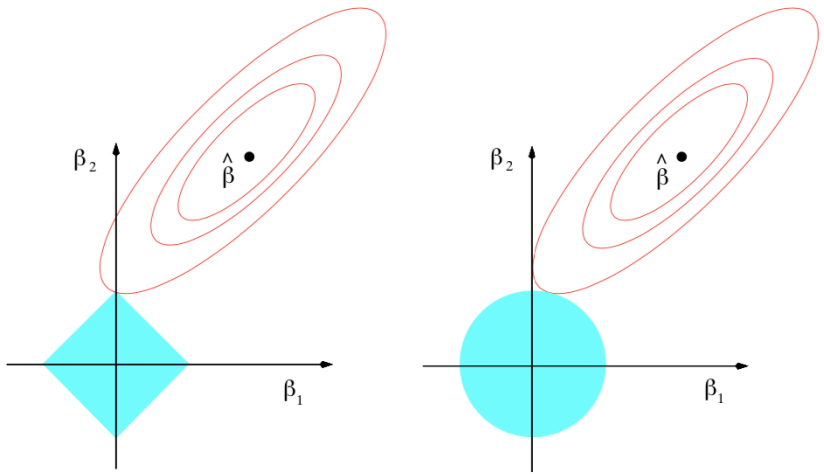
- **L2 (Ridge) regularization**

$$R(W) = \sum_{i,j} W_{ij}^2 = \|W\|_2^2$$

L-norm regularization

What does L-norm regularization look like?

Assume $W = \{\beta_1, \beta_2\}$. $\hat{\beta}$ is the set of feasible weight parameters for the solution to the neural network optimization problem.



L1 (Lasso) regularization

L2 (Ridge) regularization

L1 regularization has a **diamond** loss contour. Thus, L1 loss usually meets the feasible solution loss on the corner of the diamond shape. As a result, L1 regularization is more likely to obtain **sparse** weights.

L2 regularization has a **circular** loss contour. Thus, L2 loss usually meets the feasible solution loss on the circumference of the loss contour. Thus, L2 regularization is more likely to obtain **small weight values** compared to L1 regularization.

L-norm regularization

How to choose from L1 vs. L2 regularization?

- L2 regularization yields more stable results without too much tuning effort. Thus, we can use **L2** regularization as a good start for training CNN models.

- L1 regularization yields sparse solutions. Thus, choose **L1** regularization for compact models with high compression rate. More tuning effort is expected for L1 regularization.

- We will cover more about advanced regularization techniques and sparse solutions in Lecture: Sparse regularization.

L-norm regularization

How to implement L-norm regularization in PyTorch?

For L2 regularization, configure it in optimizer.

CLASS `torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False)` [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float*, *optional*) – momentum factor (default: 0)
- **weight_decay** (*float*, *optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float*, *optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool*, *optional*) – enables Nesterov momentum (default: False)

In Homework: For L1 regularization, add it as a part of the loss function.

L-norm Regularization

Is tuning L-norm regularization hard?

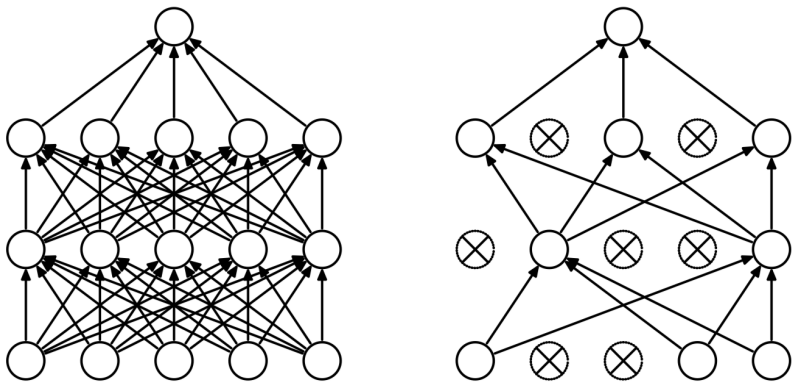
Yes. Theoretically, the regularization strength for each layer can be tuned. Thus, many efforts are spent on tuning the regularization strength of each layer as the cardinality grows exponentially with the number of layers.

Question: Suppose you have a neural network with 20 layers. Assume that for each layer, there are *n* candidate regularization choices. At most how many trials need to run for exhaustively searching all the possibility and reaching the optimal result?

Thus, most ML models employ a constant regularization strength throughout all layers.

Dropout

Dropout relieves the issue of extensive hyperparameter tuning on hyperparameters for L-norm regularization. Thus, it is an easier way to do regularization and combat overfitting.



(a) Standard Neural Net (b) After applying dropout.

Dropout **randomly** set some neurons to **zero**. Connections to zeroed neurons are also dropped.

Advantages:

- Easier configuration than L-norm regularization.
- Create an average of models to address overfitting issue. Thus, dropout obtains better empirical result than L-norm regularization.

Disadvantages:

- Extra computation cost in either training or inference.
- Sometimes dropout does not cooperate well with L-norm regularization.

Hinton, Geoffrey E., et al. "Improving neural networks by preventing co-adaptation of feature detectors." (2012)

Dropout

What do dropout layers have?

- **Dropout probability *p:*** the probability to drop (zero) a neuron and its corresponding connections in the deep neural networks.
- In most cases, we should not dropout more than half of the neurons in a DNN layer. Otherwise, we should consider to re-design this layer to have fewer capacity.

When are dropout layers enabled?

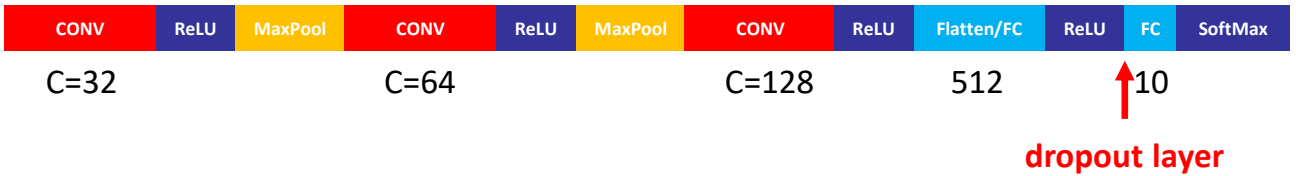
- Dropout is only enabled during the training phase to combat overfitting.
- During the testing phase, dropout is disabled.

Hinton, Geoffrey E., et al. "Improving neural networks by preventing co-adaptation of feature detectors." (2012)

Dropout

Where should we insert dropout layers?

- Traditionally, dropout is only used in **fully connected layers** with **p=0.5**.
- Recent work suggests that we can use dropout in the convolutional layers with **small *p*** values.
- We recommend inserting dropout between 2 adjacent FC layers. Under most cases, dropout is inserted after the ReLU activation between two adjacent FC layers.



- Do NOT add dropout in the last layer. The logits for softmax cross entropy calculation should never be dropped!

Hinton, Geoffrey E., et al. "Improving neural networks by preventing co-adaptation of feature detectors." (2012)

Dropout: training vs. testing

- During **training**, neurons are randomly dropped with probability.

$$Y = f(X, Z; W)$$

Y: output X: input W: weight parameters.
Z: random binary masks enabled during dropout. The binary mask has probability p of being 0 and is randomly sampled during training.

dropout only during training.

dropped neuron will be put back in testing

- During **testing**, we only need one forward pass to do the inference. Thus, we average the randomness by taking the expectation of neurons during training.

$$Y = f(X, Z; W) = E_Z[f(X, Z; W)] = \int_Z \Pr(Z \neq 0) f(X, Z; W)$$

- Since dropout probability *p* of each neuron is independent and identically distributed (i.i.d.), we approximate the integral by:

$$Y = \int_Z \Pr(Z \neq 0) f(X, Z; W) = (1 - p) f(X; W)$$

As such, **we need to scale the neurons by (1-*p*) during testing.**

Hinton, Geoffrey E., et al. "Improving neural networks by preventing co-adaptation of feature detectors." (2012)

Dropout: training vs. testing

Workflow: Dropout

Inputs: current layer activation α; dropout probability p

Training:
Step 1: Randomly select activations from α with probability p for dropout.
Step 2: Set the selected activations to 0.
Step 3: Pass the activations (many of them have been dropped to 0) to the next layer.
Step 4: Perform back propagation. Gradient of dropped neurons are 0.

Testing:
Step 1: Scale the activations α by (1-*p*).
Step 2: Pass the scaled activations α to the next DNN layer.

反 output
要 scale

However, scaling neurons during testing phase introduces extra computational cost.
This is not good for inference.

train once, inference all the time

Inverted dropout: training vs. testing

Solution: Inverted dropout integrates the scaling process into the training phase. Thus, no extra scaling in testing phase is needed. This saves computation cost for testing phase.

Workflow: inverted dropout

Inputs: current layer activation α; dropout probability p

Training:
Step 1: Randomly select activations from α with probability p for dropout.
Step 2: Set the dropped activations to 0.
Step 3: Scale the activations by 1 / (1-*p*).
Step 4: Pass the dropped activations to the next layer.
Step 5: Perform back propagation. Gradient of dropped neurons are 0.

Testing:
Step 1 Pass the activation α to the next layer.

改weight
而scale

Dropout in PyTorch

As dropout has inconsistent behavior in training / testing phase, we should be careful when using them in PyTorch.

- Use `model.train()` when training your PyTorch model with dropout enabled.
- Use `model.eval()` when testing your PyTorch model with dropout layers. Dropout will be automatically disabled in the eval mode and PyTorch will automatically handle scaling.

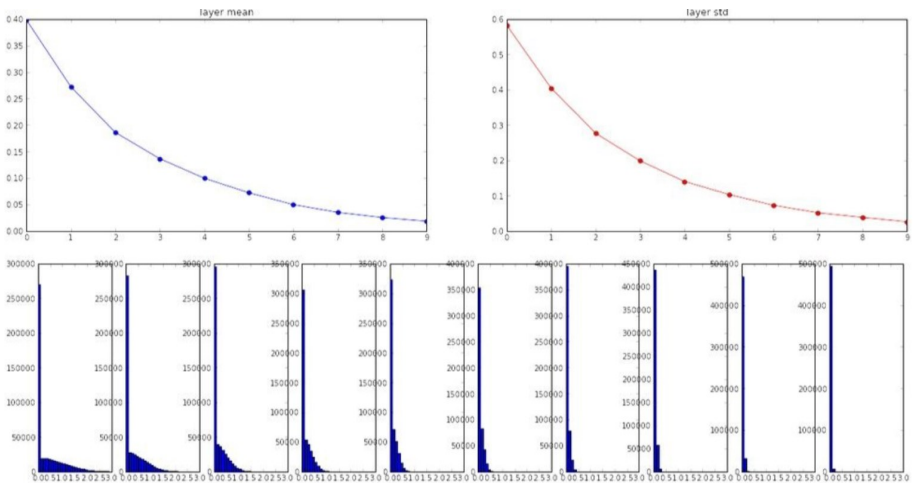
Use `torch.nn.Dropout` when training models with dropout layers enabled.

DNNs before batch normalization

Training DNNs is complicated. As the activations of the previous layers change, the proceeding layer's inputs changes. This makes the neural network more difficult to adapt to changing inputs.

- Ideally, we prefer these activations to have a distribution with zero mean and unit variance at any stage of the training process.

Activation distribution of a 10-layer DNN



Credit to Stanford CS231n, 2017

However, in this 10-layer network, activations are unscaled and has completely different distribution. This makes the DNN very difficult to train.

Batch normalization: training

Goal: Make the activations in each layer to have zero mean and unit variance.

In the **training phase**, we

1. Accumulate the moving average μ and variance σ (channel-wise)
2. Update trainable scale γ and bias β parameters (channel-wise)
   - γ and β are introduced to recover the scale of activations, but in a more controlled way

Input: values of x over a mini-batch.

Output: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathrm{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_{\mathrm{B}}^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathrm{B}})^2$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathrm{B}}}{\sqrt{\sigma_{\mathrm{B}}^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

μ_{B}: mean of the current mini-batch outputs.

σ_{B}^2: variance of the current mini-batch outputs.

γ: trainable scale parameter.

β: trainable bias parameter.

ϵ: Small value to prevent zero division.

Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." (2015)

Batch normalization: training

Two accumulated statistics: Moving mean μ, moving variance σ

- Moving mean and moving variance are accumulated during the training process, using batch statistics μ_B and σ_B.

$$\mu^{t+1} = (1-\rho)\mu^{t} + \rho\mu_{\mathrm{B}}^{t+1}$$
$$\sigma^{t+1} = (1-\rho)\sigma^{t} + \rho\sigma_{\mathrm{B}}^{t+1}$$

- A good starting value for momentum ρ is 0.1 for batch size 128.
- A good starting value for ϵ is 1e-5.
- The PyTorch default settings for momentum/epsilon in batch normalization (i.e., 0.1/1e-5) can be a good start.

When applying batch normalization
- Use `torch.nn.BatchNorm2d` for convolution layers
- Use `torch.nn.BatchNorm1d` for fully connected layers

Batch normalization: testing

In testing phase, use the accumulated moving mean μ and moving variance σ to do batch normalization.

Question: Why don't we use batch statistics from test data as moving mean / moving average?

Answer: This will use some information from test data when training a deep neural network, which is not allowed. **DO NOT update moving mean/moving variance with test data!**

Note: **Batch normalization has different behaviors in training and testing phase.**

- Be careful to use the correct phase in PyTorch when applying batch normalization.
- Recall that we use `model.train()` for training phase, and `model.eval()` for testing (evaluation) phase.

Batch normalization: layer position

- Batch normalization (BN) is usually inserted before the non-linear activation functions (e.g., ReLU, Swish etc.).

CONV-RELU-CONV-RELU

CONV-BN-RELU-CONV-BN-RELU

- Recall that dropout layers are added after non-linear activation.

CONV-BN-RELU-DROPOUT-CONV-BN-RELU-DROPOUT

- Empirically, batch normalization gives about 2% performance gain under the same training protocol.
- BN v.s. Dropout
  - While BN is helpful or essential for optimization, Dropout is more like an extra regularization technique (less essential).
  - In modern CNNs, BN is still prevalent but Dropout is not used that often.

Label smoothing

- One-hot labels in the training process is likely to lead to extreme predictions for neural networks.
  - One-hot labels encourage the logit of the true class to be much larger than the logits of incorrect classes [1].

- To alleviate this issue, we use **label smoothing** to smooth out the one-hot labels.
- In label smoothing, the original one-hot label:

$$y_{label} = [0,1,0,0,0, \dots]$$

is smoothed and replaced with

$$y_{label} = \left[\frac{\epsilon}{K-1}, 1-\epsilon, \frac{\epsilon}{K-1}, \frac{\epsilon}{K-1}, \frac{\epsilon}{K-1}, \dots\right]$$

K: number of classes

ϵ: label smoothing factor

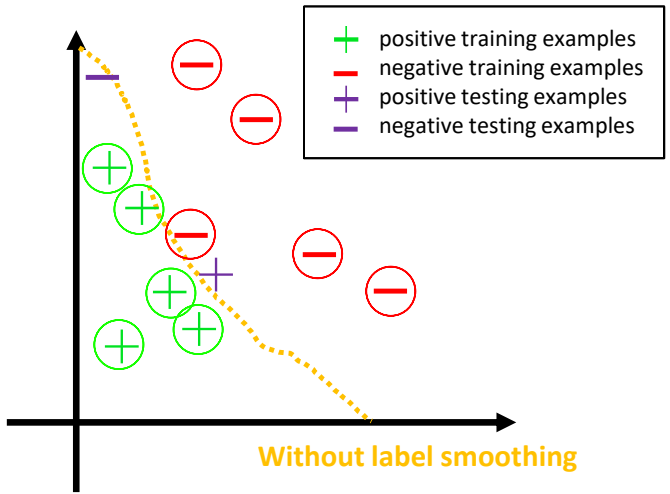
give ϵ to others to distribute evenly

- Usually, the label smoothing factor ϵ is set to 0.1 to combat overfitting.

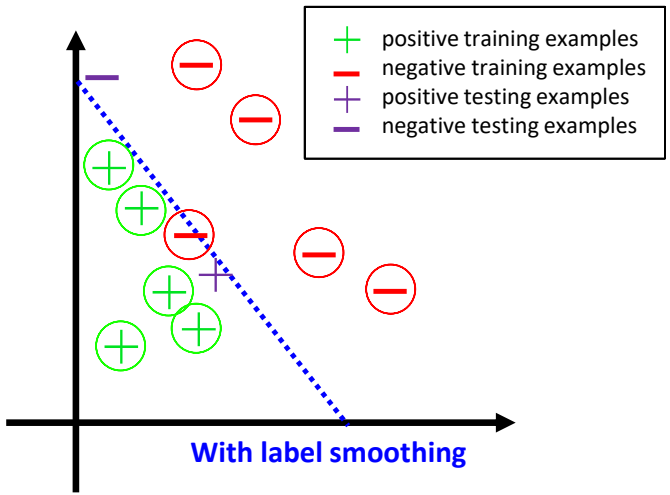
[1] https://arxiv.org/pdf/1906.02629.pdf

Label smoothing

Effect of label smoothing on a toy example



Without label smoothing, DNNs makes extreme predictions. The decision boundary is **curvy** to fit each training example. This can lead to serious overfitting. As a result, new test examples are not classified correctly.



With label smoothing, decision boundary can cross the examples with minimal violation against the one-hot labels. Thus, this decision boundary classifies new test examples correctly.

Babysitting the CNN training

Lecture 5: Basic techniques

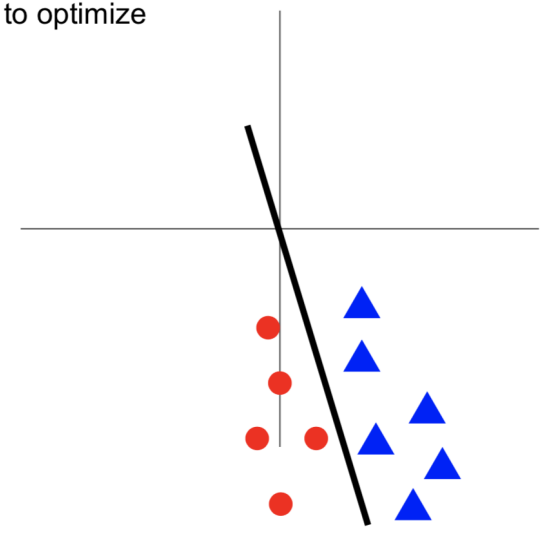
Lecture 6: Advanced techniques

- Regularization
- Data preprocessing
- Hyperparameter tuning

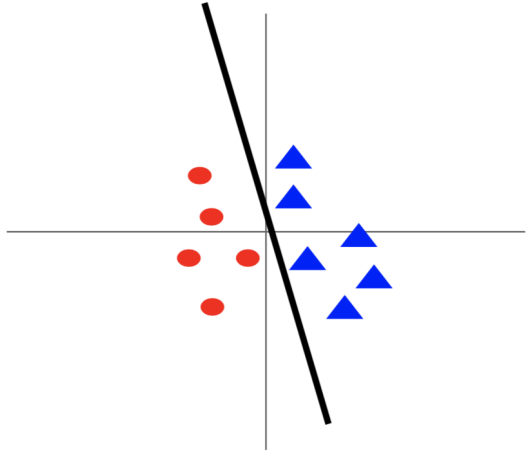
Data normalization

- Normalized data are the easiest for neural network layers to learn. Thus, we normalize the input data to have zero mean and unit variance. Empirically, normalized data yields a better result.

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



$$output = (input - mean)/std$$

`torchvision.transforms.Normalize`

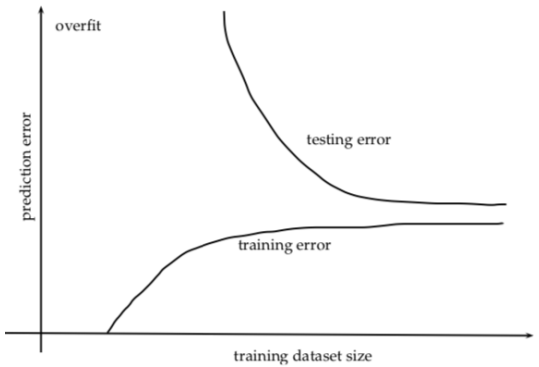
Credit to Stanford CS231n, 2017

Data augmentation

Deep neural network learns better with more data!

- We can use data augmentation to easily generate more training data. It is also a form of “regularization” to prevent neural networks from “memorizing” the original training dataset.
  - Diversity introduced by augmentation prevents the model from seeing the exact same set of images over and over again

- Common data augmentation for **images**:
  - Random horizontal/vertical shift
  - Random horizontal/vertical flip
  - Random brightness change



- Note: data augmentation is only enabled in the training process. It is recommended not to perform data augmentation on validation/test dataset.

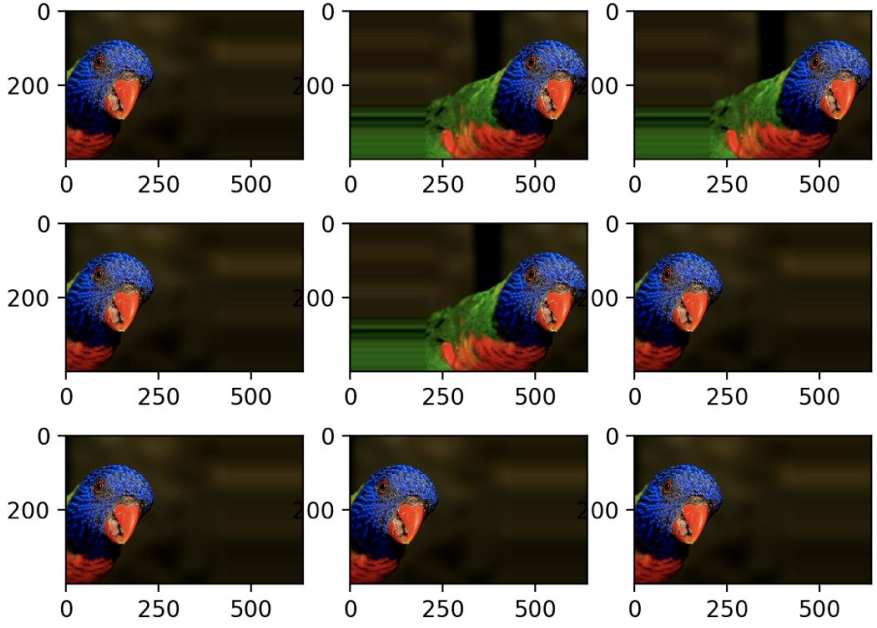
Data augmentation may hurt small models as they may underfit.

Data augmentation: random shift

- Randomly shift an input image.

`torchvision.transforms.RandomCrop`

Use **padded random crop** to do it!

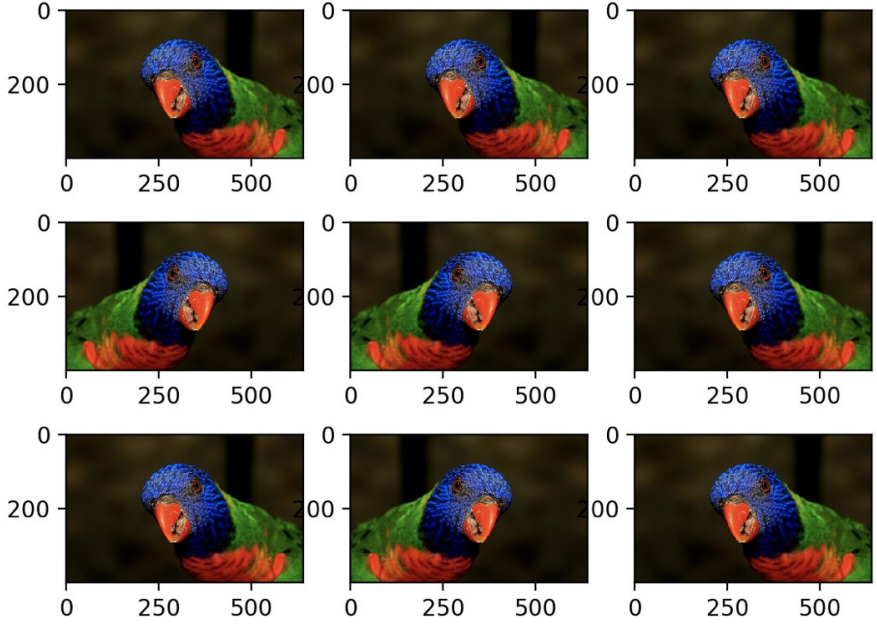


https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/

Data augmentation: random flip

- Randomly flip an input image horizontally.
- Flipping images vertically may make it very difficult to recognize, thus is not recommended.

```
torchvision.transforms.RandomHorizontalFlip
```

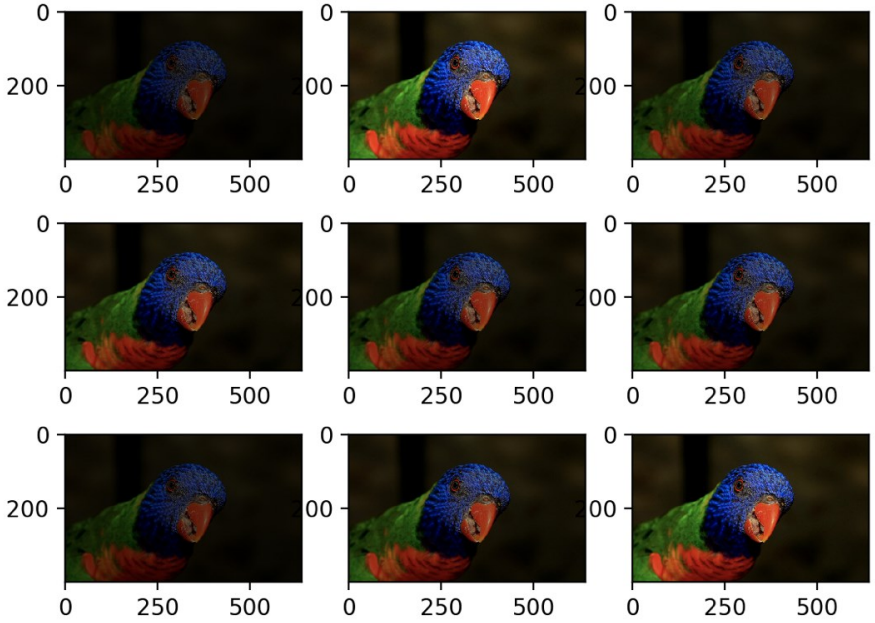


https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/

Data augmentation: brightness change

- Randomly change the brightness of an input image.
- Contrast and saturation can also be changed in this way.

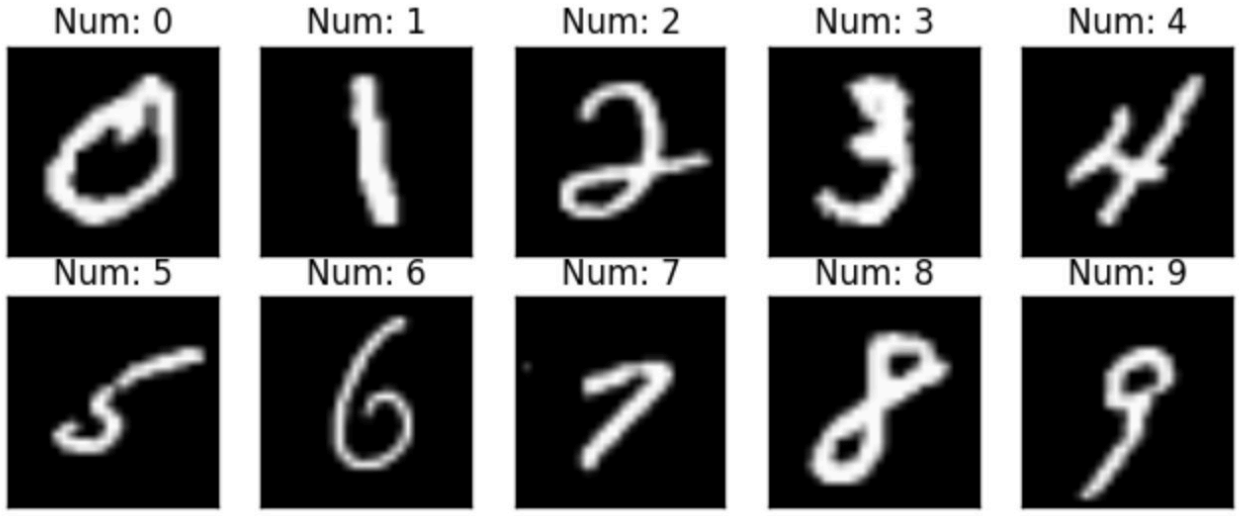
```
torchvision.transforms.functional.adjust_brightness
```



https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/

Data augmentation: use it wisely

- **Data augmentation may corrupt the dataset!**

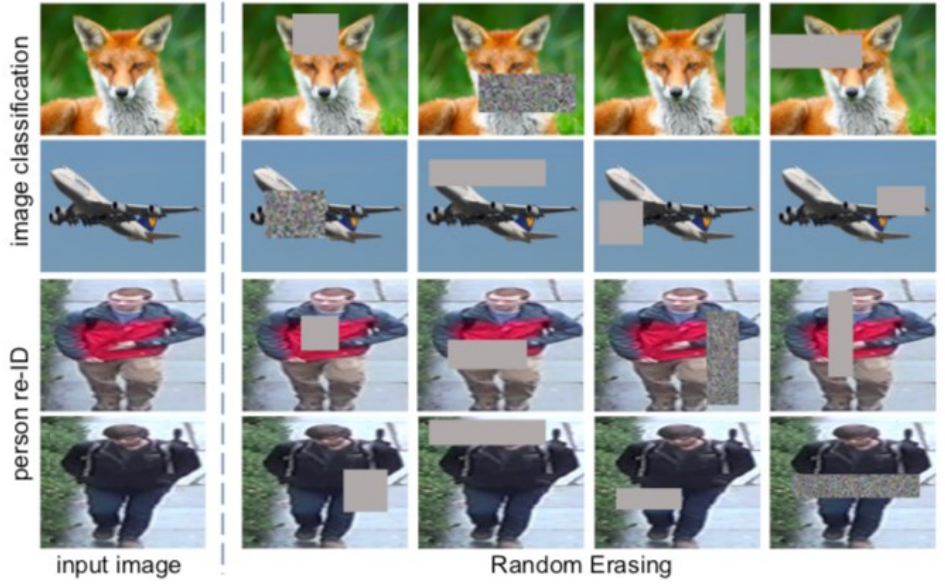


- For example, it is unreasonable to use random vertical flip during MNIST training as digit **6** and digit **9** may look similar.

- **Make sure that the images after data augmentation are reasonable.**

Random erasing

- **Random erasing** uses a random mask to occlude images to aggressively augment data.
- It is a complementary method to data augmentation and regularization methods.



`torchvision.transforms.RandomErasing`

Zhong, Zhun, et al. "Random erasing data augmentation." (2017)

Babysitting the CNN training

Lecture 5: Basic techniques

Lecture 6: Advanced techniques

- Regularization
- Data preprocessing
- Hyperparameter tuning

Hyperparameter tuning

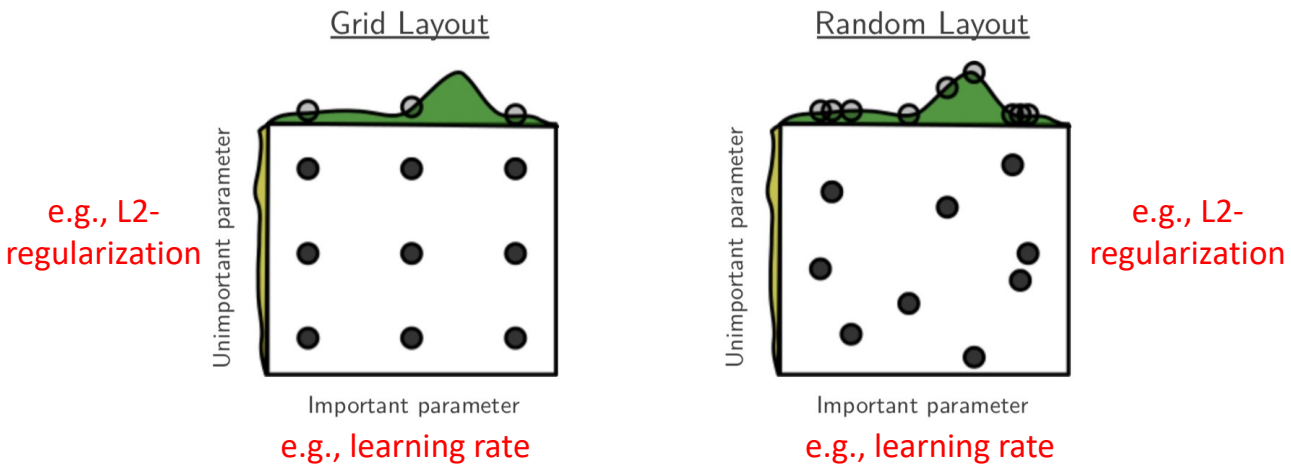
What is hyperparameter and why we need to tune it?

- Hyperparameters are **non-trainable parameters,** which are configured manually to control the learning process.
- Hyperparameters play a very important role in determining the performance of neural networks.
- Important hyperparameters include learning rate, regularization strength, dropout rate, and the used optimizer etc.
- Neural architecture is also a hyperparameter!
- Learning rate is the most sensitive hyperparameter. Tune the learning rate if only one hyperparameter can be tuned.
- The reason why a specific hyperparameter works well on a given model is still a mystery. In most cases, the hyperparameter settings are not interpretable.

Grid layout vs. Random layout

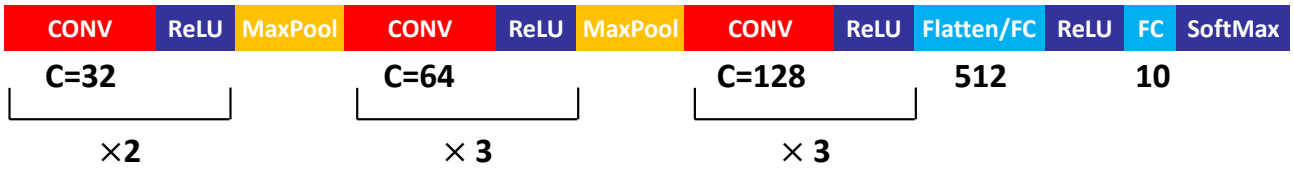
How could you tune your hyperparameter?

- **Grid layout** vs **random layout**
- The random layout might mitigate the “overshooting” issue.
- In practice, however, people often use grid layout (grid search) to tune the hyperparameters.
  - Relatively more efficient
  - There exists some heuristics/popular choices for some hyperparameters, e.g., lr [0.1, 0.01, 0.001], weight decay [5e-4, 1e-4, 5e-5]



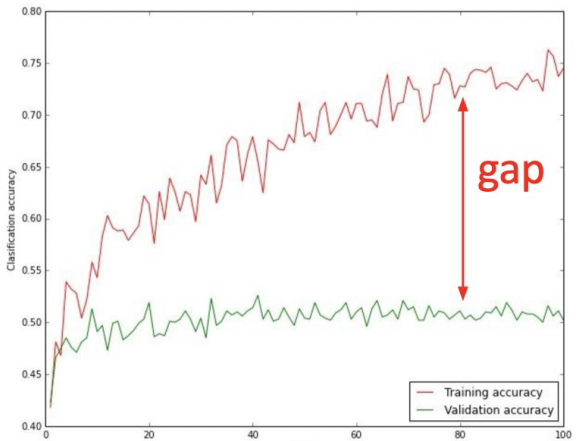
Neural architecture

- Neural network architecture is the most abstract hyperparameter.
- The best way to tune it for CNNs is to follow the design guidelines:



When should add/delete layers/channels in a neural architecture?

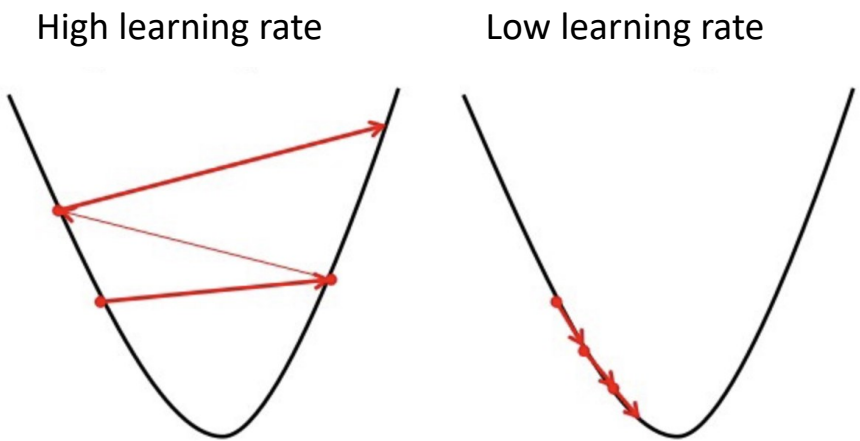
- Increase model capacity by increasing layers/channels if both training and validation accuracy are low. This denotes your model is underfitting.
- Decrease model capacity by reducing layers/channels if overfitting is observed. You may also observe a large generalization gap.



Tuning learning rate

Learning rate is the most important hyperparameter to optimize.

- High learning rate will lead to divergence of loss.
- Low learning rate slows down the learning process. Thus, the training process takes a much longer time to reach convergence, which is a waste of computing resources. Low learning rate may also lead to sub-optimal solutions.

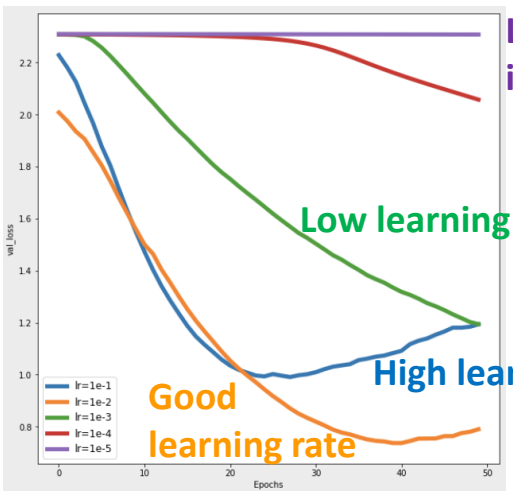


Tuning learning rate

- Different optimizers prefers different default learning rates.
  - A good starting point for learning rate with **SGD with momentum** can be 0.01.
  - For **Adam** optimizer, a good starting point is 1e-3.
- Under most cases, preferable learning rate reduces with the complexity of optimization algorithm.

General rule:

- Increase the learning rate if the loss decreases slowly.
- Decrease the learning rate if the loss oscillates drastically, goes up or diverges.

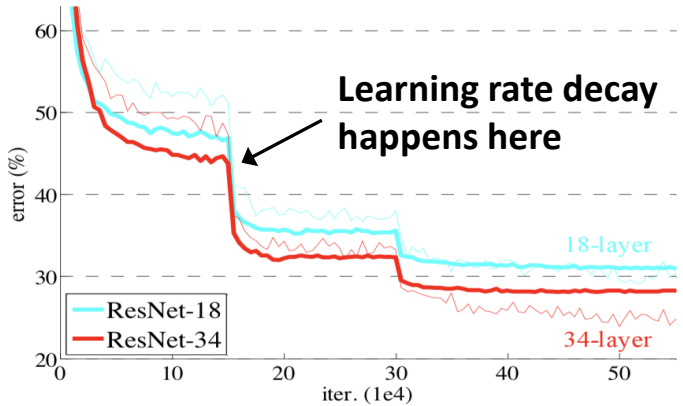


Learning rate schedule

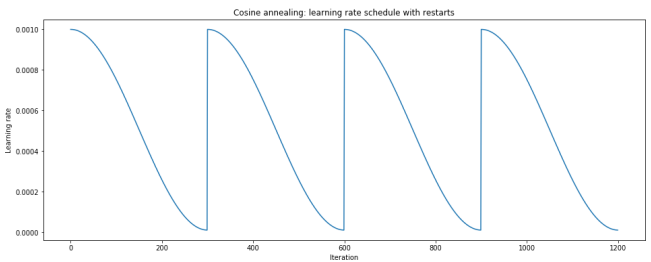
- It's not a good idea to fix the learning rate throughout the training process.
- When accuracy curve plateaus, we can lower the learning rate to further increase accuracy.

Recommended learning rate schedule:

- Step-wise lr decay
  - Decay by 0.1 every certain number epochs or when validation accuracy plateaus.
  - This should work for most of the cases in our assignment.
- Cosine lr decay
  - Schedule the lr as a cosine function.
  - Might lead to better performance; try it out if you are interested.



Bold curves: validation error
Thin curve: training error.



Source: He et al., Deep Residual Learning for Image Recognition; https://www.jeremyjordan.me/nn-learning-rate/

Tuning regularization

For L2 regularization:

- We recommend using regularization strength $\lambda = 1e - 4$ as a start for CNN training.
- Reduce the regularization strength if your CNN architecture is small.

For dropout:

- Dropout layer with probability = 0.5 is commonly used after each fully connected layer except the final output layer.
- You may try to add dropout layer with probability 0.05/0.1 after each convolutional layer or pooling layer. However, this is not guaranteed to have a better performance.

Tuning optimizer

- We recommend using Adam optimizer as a good approach for sanity check. If Adam optimizer does not work on your network, there must be some problem with the neural network design or the coding.
- SGD optimizers are usually used to reach state-of-the-art results. But be patient, they may take longer time to reach the optimal solution.

- If your model requires a high performance, use SGD optimizers with longer training regime.
- If you want to get your model within a short amount of time, use Adam optimizer.

In this and last lectures, we learned:

Lecture 5: Basic techniques

- Training overview
- Neural network design
  - General architecture
  - Activation functions
  - Weight initialization
- Loss function
  - Optimizer

Lecture 6: Advanced techniques

- Regularization
- Data preprocessing
- Hyperparameter tuning

Training log example

regularization	weight decay	epochs	lr	pretrained	top-1/5
None	0.0001	90	0.1	None	**76.15, 92.87**
3, 5e-4	0.0001	45	0.01	Yes	~60%
3, 2e-4	0.0001	45	0.01	Yes	66.902, 87.556
3, 1e-4	0.0001	45	0.01	Yes	70.032, 89.696
3, 2e-4	0.0001	70	0.1	Yes	59.772, 82.458
3, 1e-4	0.0001	70	0.1	Yes	crashed@57 epoch
3, 1e-4	0.0001	90	0.1	Yes	**64.210, 85.760**
3, 7e-5	0.0001	90	0.1	Yes	**67.118, 87.756**
3, 5e-5	0.0001	90	0.1	Yes	**69.190, 88.964**
3, 5e-5	0.0001	70	0.1	Yes	69.540, 89.166
3, 4e-5	0.0001	90	0.1	Yes	**70.658, 89.990**
3, 3e-5	0.0001	90	0.1	Yes	**71.668, 90.622**
3, 3e-5	**0**	90	0.1	Yes	74.902, 92.122
3, 2e-5	0.0001	70	0.1	Yes	crashed@32 epoch
3, 2e-5	0.0001	90	0.1	Yes	**73.038, 91.476**
3, 1e-5	0.0001	90	0.1	Yes	**74.962, 92.252**
3, 1e-5	0.0001	90	0.1	**NO**	73.154, 91.352