Lecture 4-2 Better Accuracy

Tian Sheuan Chang

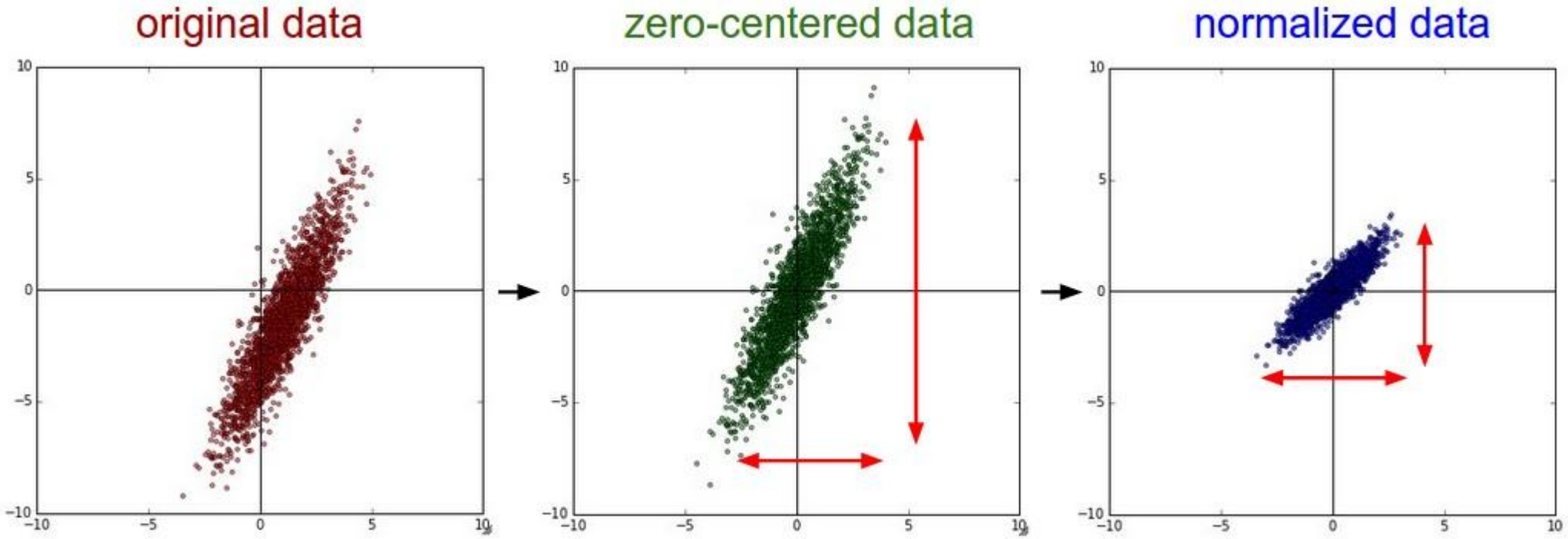
Outline

- Babysitting the learning process
  - Quick check on what's wrong
- Better accuracy
  - Detailed method to improve training and test accuracy

BABYSITTING THE LEARNING PROCESS QUICK CHECK

Step 1: Preprocessing the data

- Mean subtraction: zero centered
- Normalization: data at approximately the same scale



X -= np.mean(X, axis = 0)

```
X /= np.std(X, axis = 0)
```

Source: cs231n

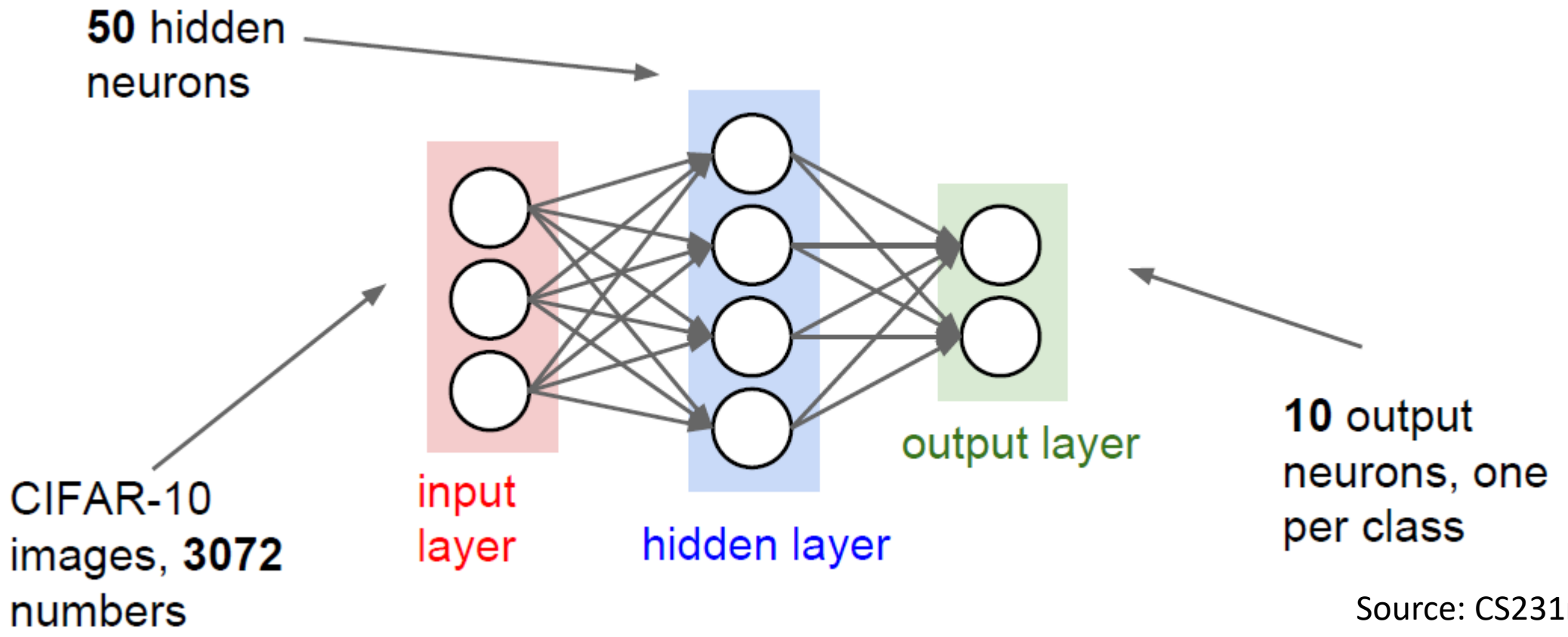
Data Splitting: Training/Validation/Test

- Ratio to split depends on dataset size and data distribution
- Old machine learning era (for small data set)
  - 70/30% or 60/20/20%
- Big data era
  - million examples: 98% train, 1% dev, 1% test.
  - More than a million examples: 99.5% train and 0.25% dev, 0.25% test.

Source: Andrew Ng, coursera

Step 2: Choose the architecture

- Choose model capacity to fit or overfit your data to get training error low
  - Trial and error, transfer learning, receptive field, scaling laws



Source: CS231n

Step 3. Trial Run

- 3 or 5 epochs to test everything is fine before long epoch run

Double check that the loss is reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0)
print loss
```

disable regularization

```
2.30261216167
```

loss ~2.3.
"correct " for
10 classes

returns the loss and the
gradient for all parameters

- Disable regularization to check data cost first

10 classes, avg prob = 0.1, Softmax −ln(0.1) = 2.302

Source: CS231n

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

crank up regularization

```
3.06859716482
```

loss went up, good. (sanity check)

- Add regularization, loss will go up

Source: CS231n

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

The above code:

- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla 'sgd'

Source: CS231n

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
```

Very small loss, train accuracy 1.00, nice!

```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```


Source: CS231n

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)

Source: CS231n

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
```

Now let's try learning rate 1e6.

Source: CS231n

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

cost: NaN almost always means high learning rate...

Source: CS231n

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

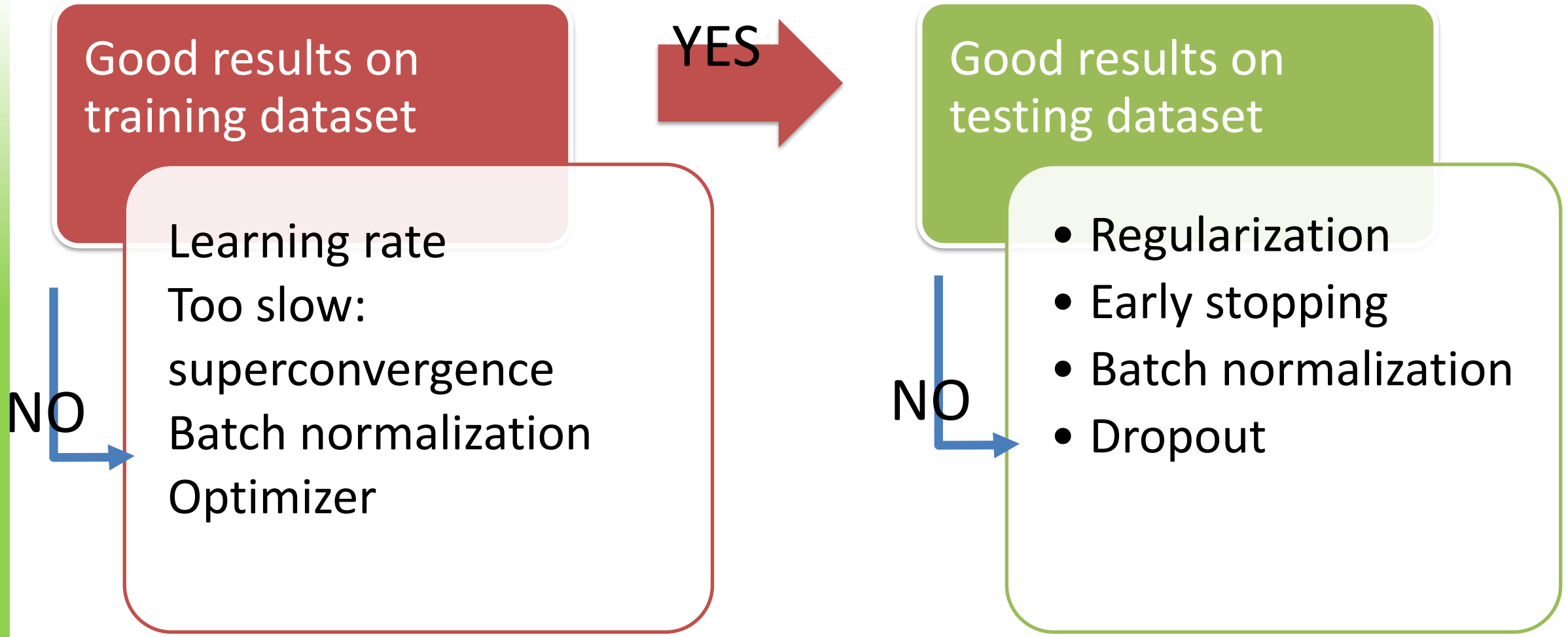
3e-3 is still too high. Cost explodes….

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-5]

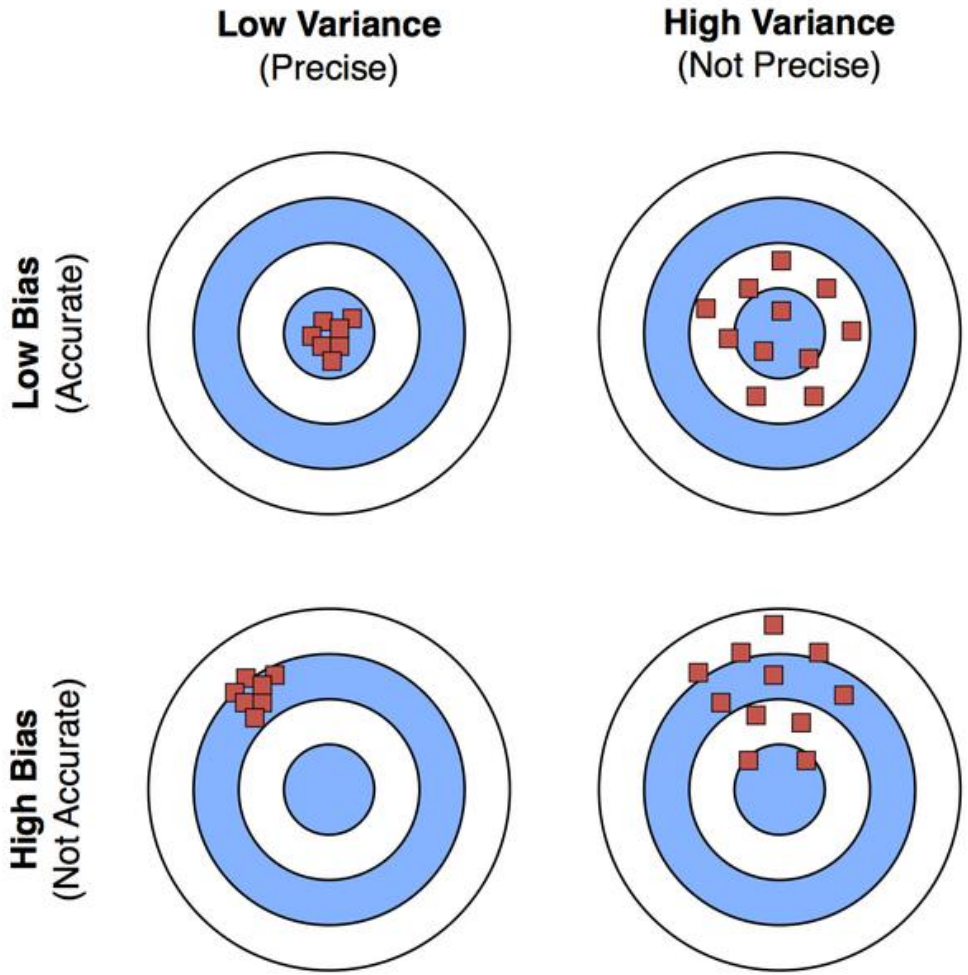
Source: CS231n

BETTER ACCURACY?
BABYSITTING YOUR TRAINING PROCESS

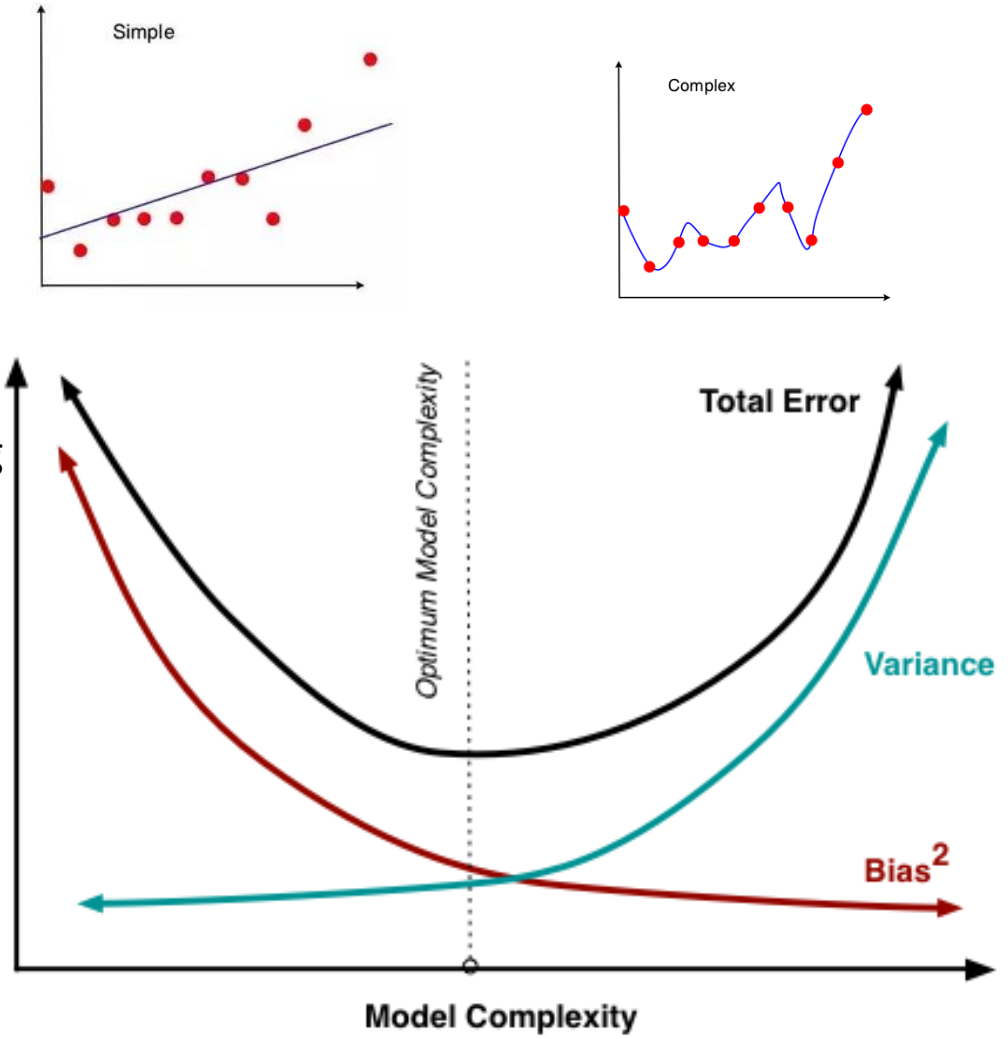
Tips for Deep Learning



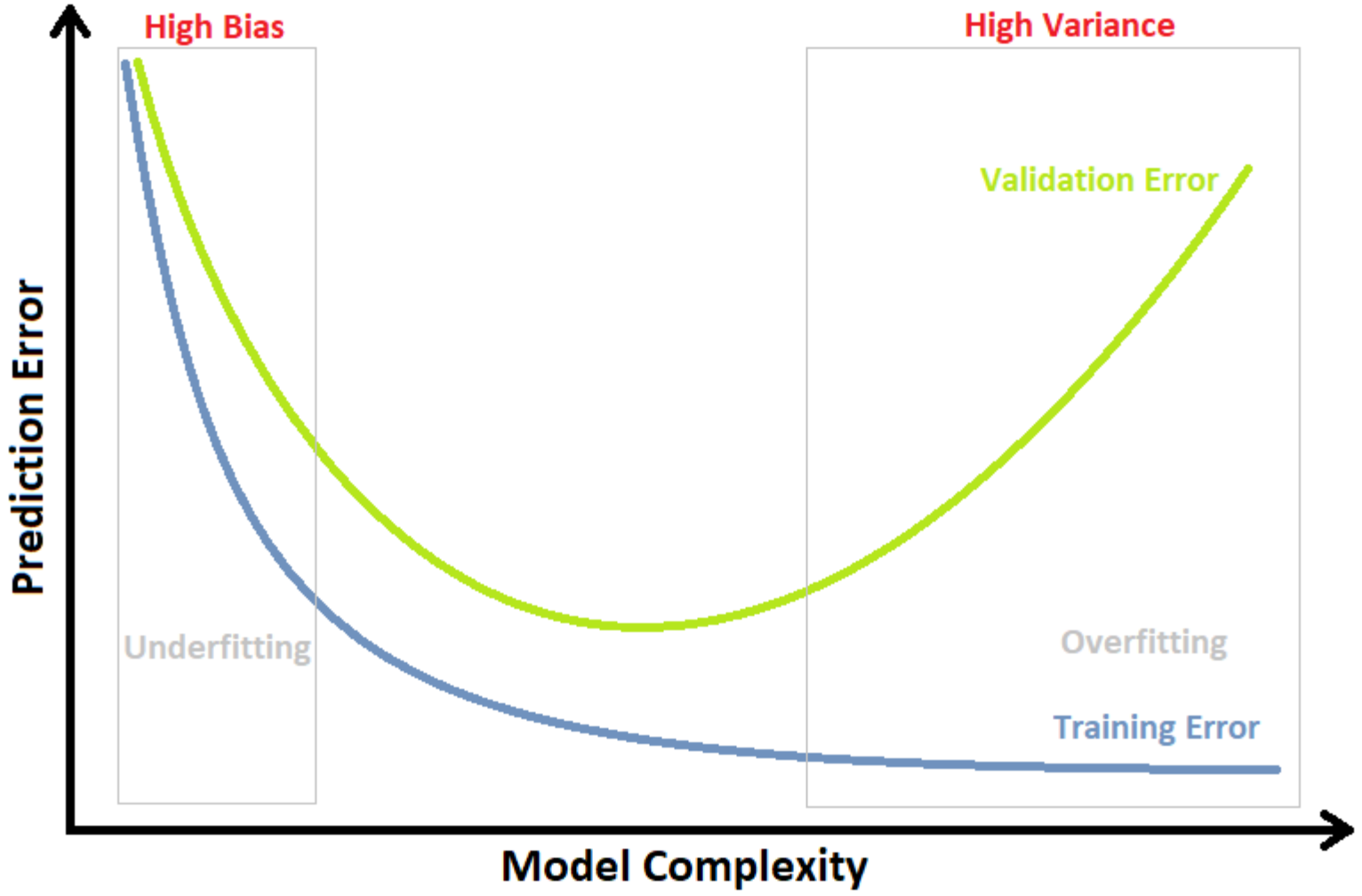
Bias and Variance



Underfitting



Training 目標: 至少overfitting


High Bias
High Variance
Validation Error
Prediction Error
Underfitting
Overfitting
Training Error
Model Complexity

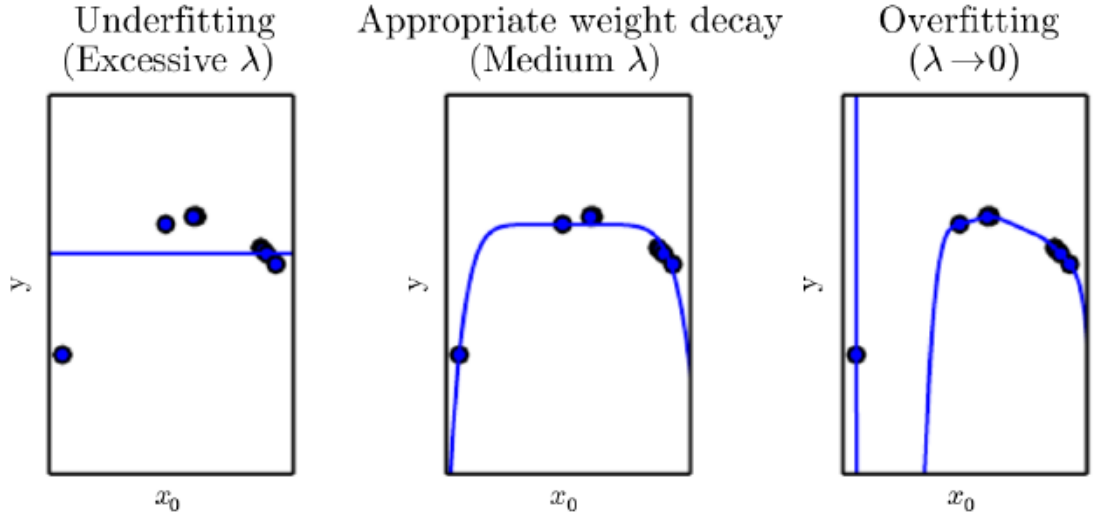
Controlling the Capacity: Regularization

- Modification made to the learning algorithm to reduce generalization error but not its training error (usually at the cost of higher training error)

Example: To include *weight decay* in the training criterion

$$J(\boldsymbol{w}) = \mathrm{MSE}^{(\mathrm{train})} + \lambda \boldsymbol{w}^T \boldsymbol{w}$$

where λ controls preference for small $\boldsymbol{w}$ and is determined a priori



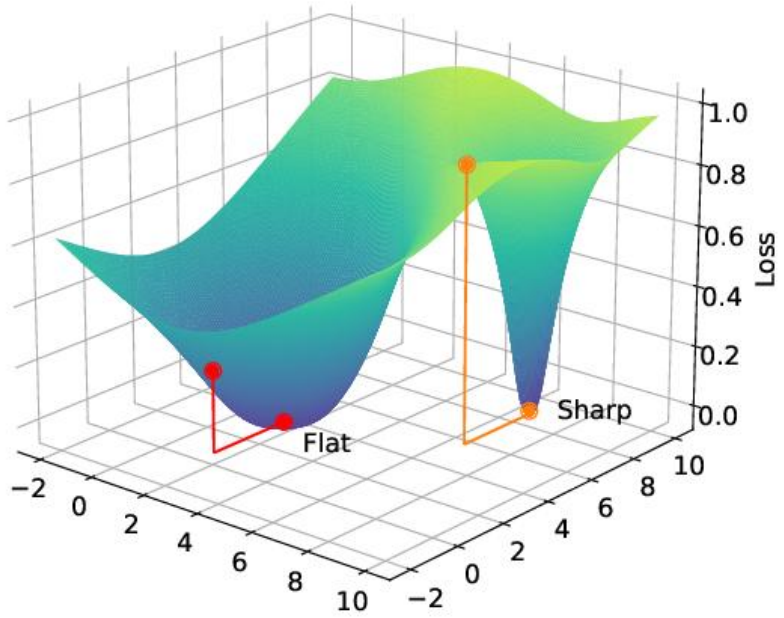
A degree-9 polynomial model fitted to quadratic data

傳統機器學習的觀點

- **泛化 (generalization)**：模型能否從「訓練樣本」學到規律，並應用到「未見資料」。
- **過擬合 (overfitting)**：模型太依賴訓練樣本細節（包含噪音），在測試資料表現不佳。
- **經典 Bias-Variance Tradeoff**：
  - **低容量模型**：偏差高、變異低 → 訓練/測試都差 (欠擬合)。
  - **高容量模型**：偏差低、變異高 → 訓練好、測試差 (過擬合)。
  - 理論上應該有一個「最佳模型容量」讓誤差最小。

Flat vs Sharp Minima

- Optimizer (SGD) 偏好找到『平滑、寬廣的谷底 (flat minima)』
  - Flat minima → 對資料小變動不敏感 → 泛化好
  - Sharp minima → 過度依賴某些樣本 → 泛化差



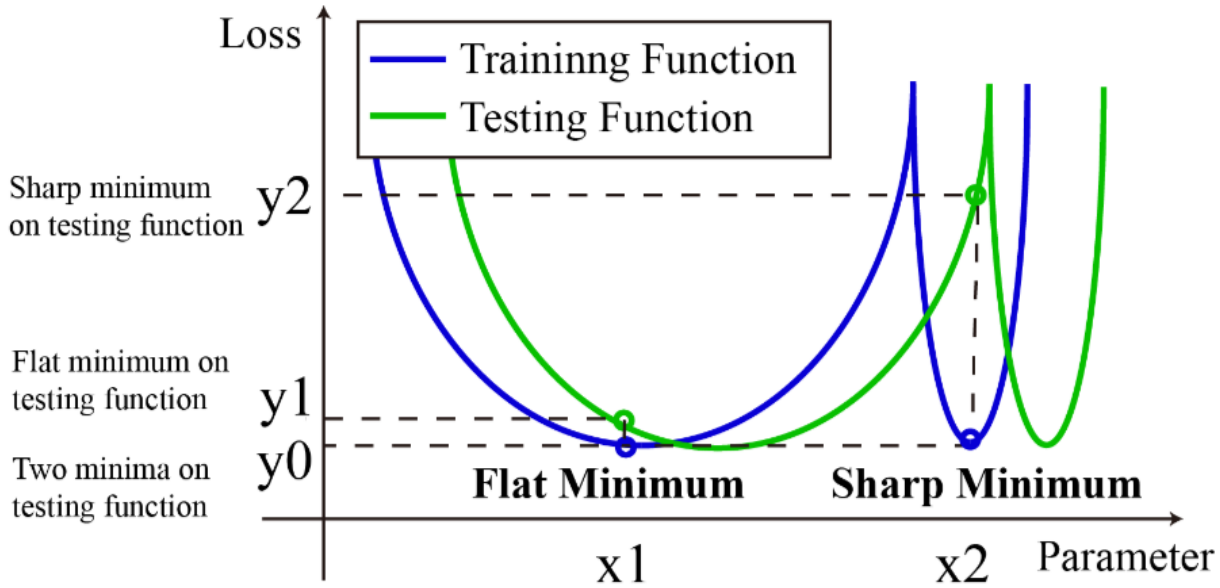


Figure 3. An intuitive explanation of the difference between flat and sharp minima [20].

https://arxiv.org/pdf/1905.08094

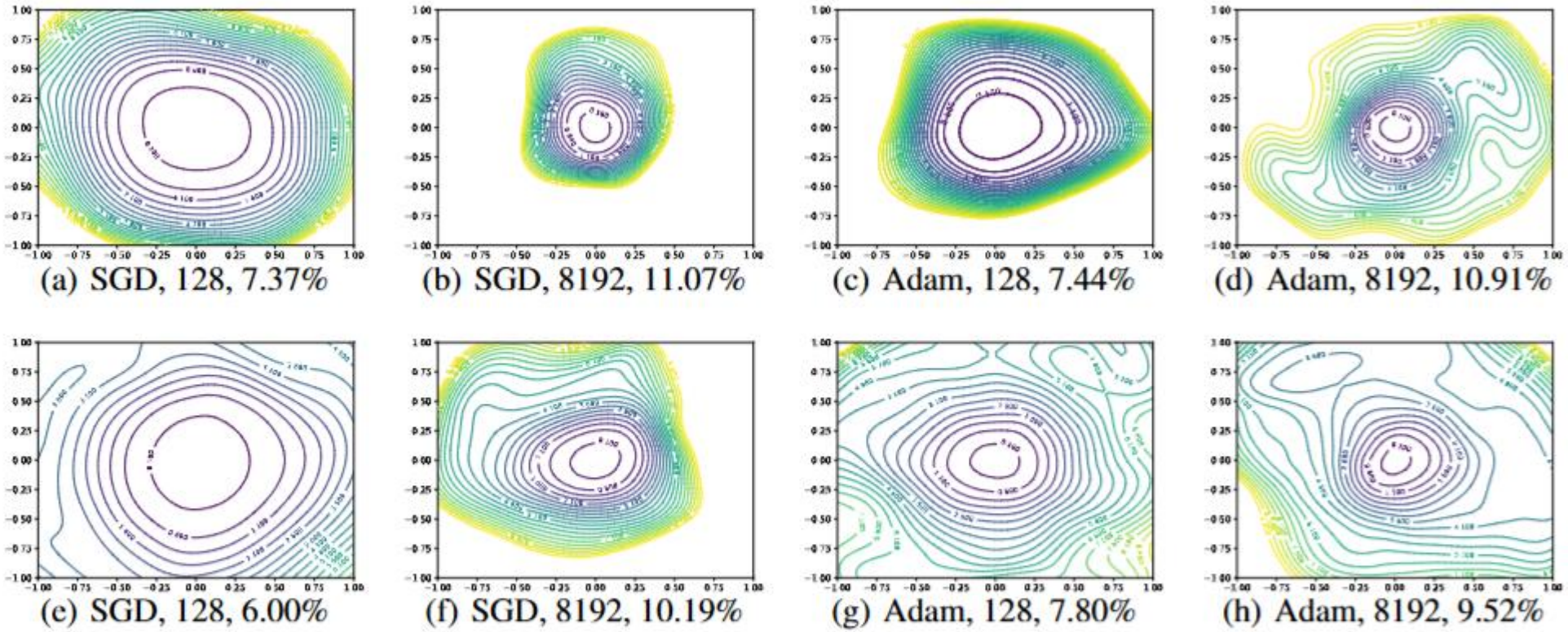
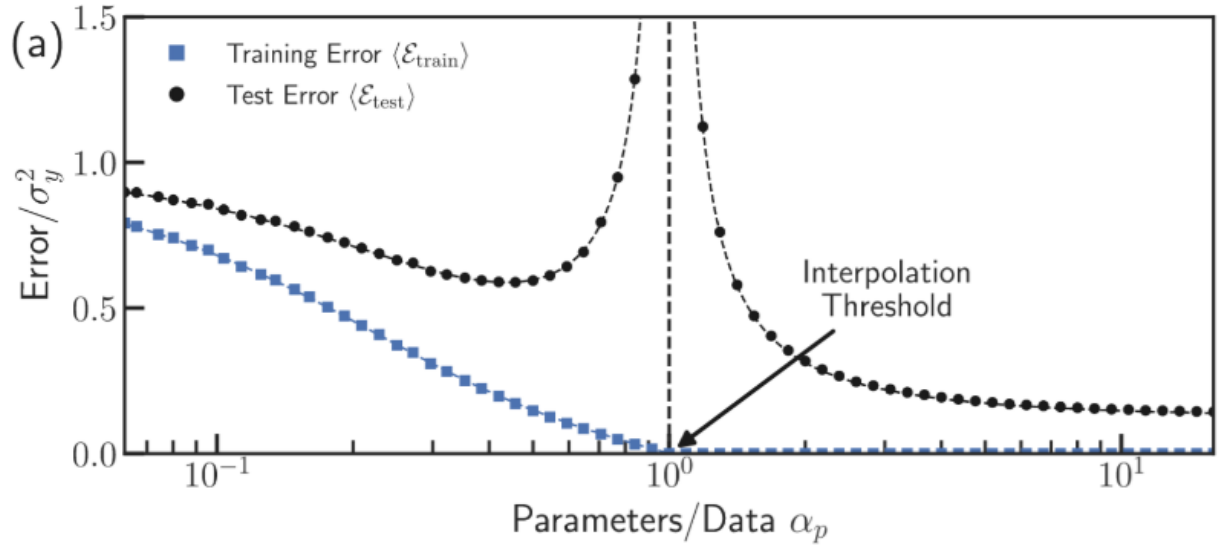


Figure 5: 2D visualization of solutions obtained by SGD with small-batch and large-batch. Similar to Figure 4, the first row uses zero weight decay and the second row sets weight decay to 5e-4.

深度學習的「反直覺」現象

- 然而在深度學習裡：
  - 大網路（幾百萬甚至幾十億參數）常常**比小網路泛化更好**。
  - **經典的 bias-variance 曲線不再成立**。
  - 現象稱為 **double descent**：誤差隨模型容量先下降→上升→再下降。
- 👉 引發問題：**為什麼「嚴重 overparameterization」反而幫助泛化？**



Double descent: test error falls, rises, then falls as a ratio of parameters to data

https://arxiv.org/abs/2303.14151

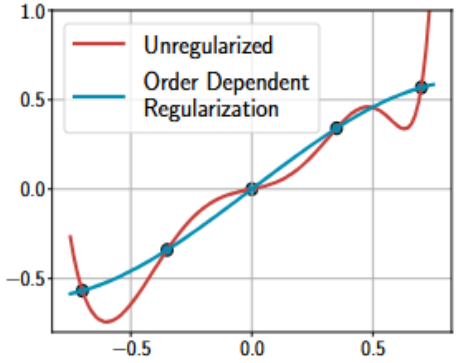
直覺化比喻

- **背書 vs 理解**
  - 小模型 = 只能背幾條公式 → 不會解新題。
  - 大模型 = 書讀很多，但能提煉出「共通規律」→ 解題更靈活。
- **山谷比喻**
  - 過擬合 = 掉進一個又深又窄的山谷 → 訓練誤差極低，但一點點資料改變就跌倒。
  - 泛化好的模型 = 在一個寬廣平緩的谷底 → 對資料變動不敏感。
- **拼圖**
  - 小模型：只能拼出簡單的圖案。
  - 大模型：雖然可以拼出亂七八糟的東西，但訓練算法會「偏好」拼出有結構的圖案。

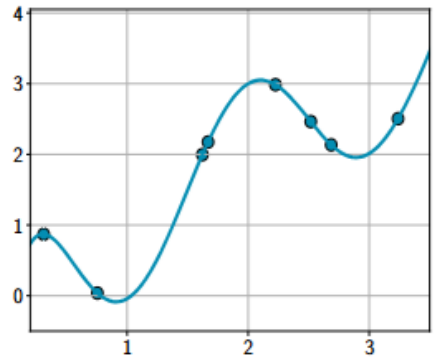
Implicit Regularization in DL Models

- 深度學習的許多「怪現象」其實早在物理/統計學中就被觀察過。
- **核心觀念：隱式正則化 (implicit regularization)**
  - 即使沒有顯式加上 L1/L2 正則化，SGD、Adam 等優化器也會「偏好簡單解」。
  - 深度網路雖然能表達複雜函數，但實際訓練過程傾向找到「平滑的、結構化的」解，而不是記憶訓練樣本的噪音。
- **「能量景觀」直覺**
  - 訓練過程像在能量函數裡找到最低谷 (minimum)。
  - 在高維度下，很多 minimum 存在，但 SGD 傾向落在「flat minimum」（寬而平的谷）→ 泛化好。
  - 而不是「sharp minimum」（尖而深的谷）→ 過擬合。
- **Double Descent 與老理論呼應**
  - 在 **過參數**區間，模型雖能完全記憶訓練集，但 SGD 會自動偏向找到「最簡單」的零訓練誤差解。
  - 這和物理裡「phase transition」的行為相似。

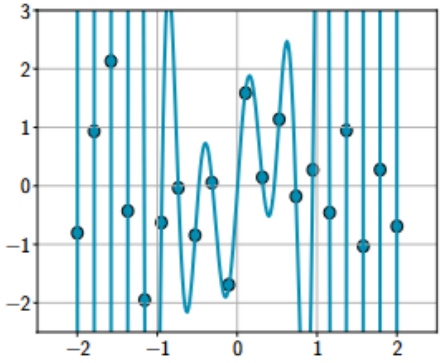
https://arxiv.org/abs/2503.02113



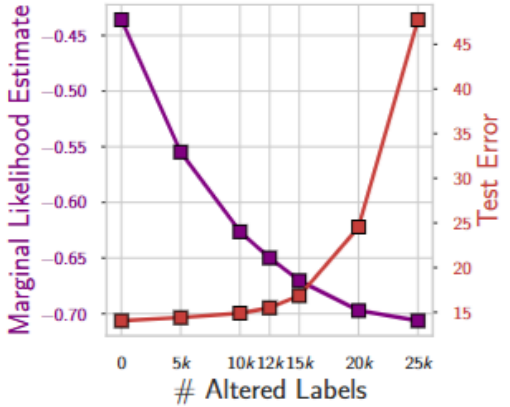
(a) Simple Data



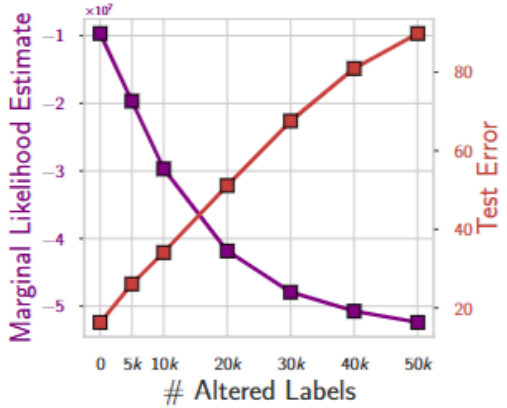
(b) Structured Data



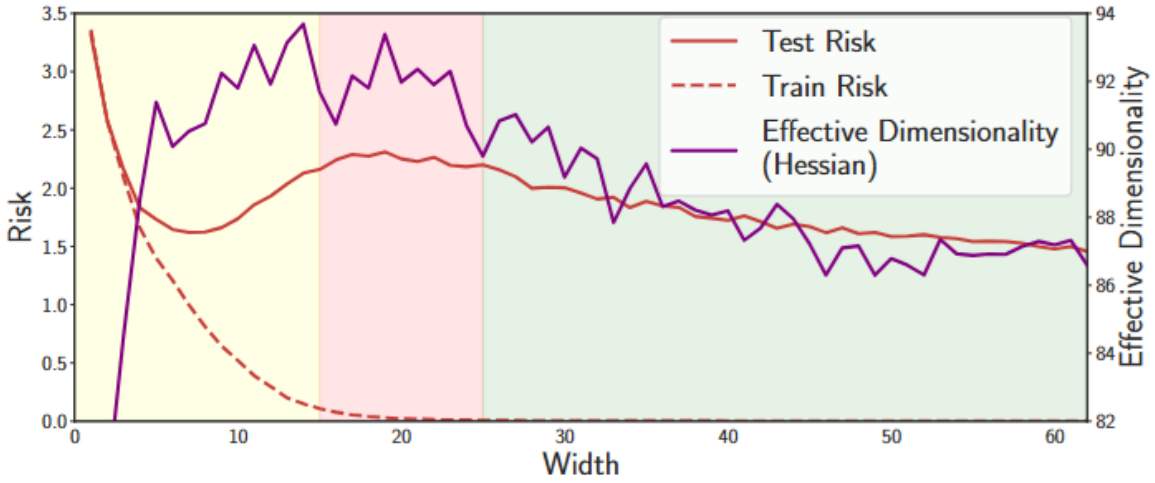
(c) Noisy Data



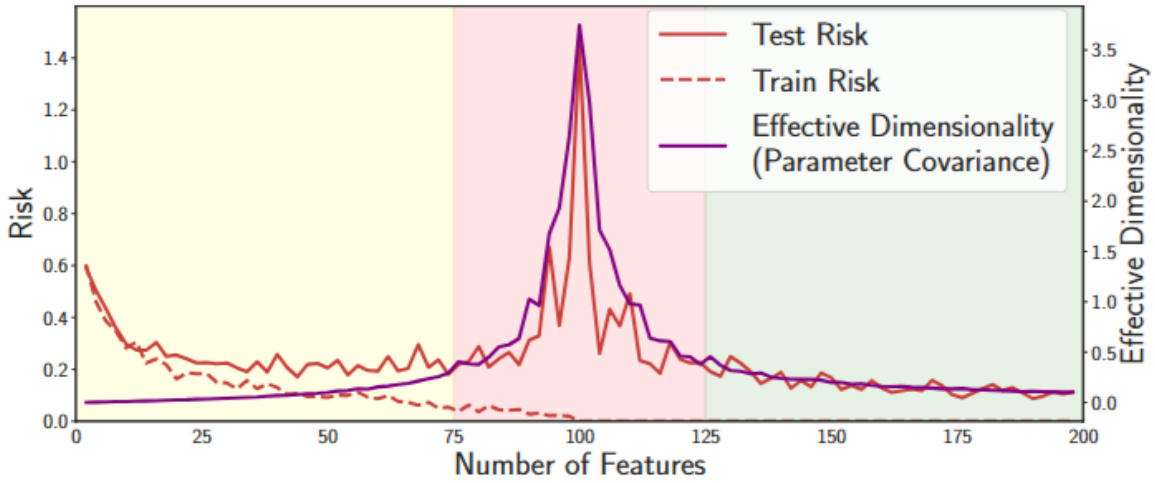
(d) GP on CIFAR-10



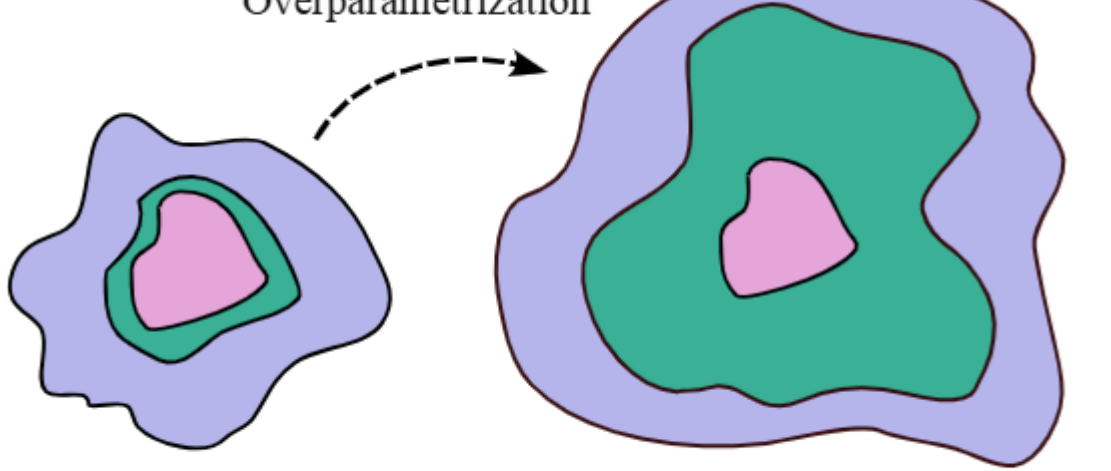
(e) ResNet-20 on CIFAR-10



(f) ResNet-18



(g) Linear Random Features



High loss Low loss, overfitting Low loss, good generalization

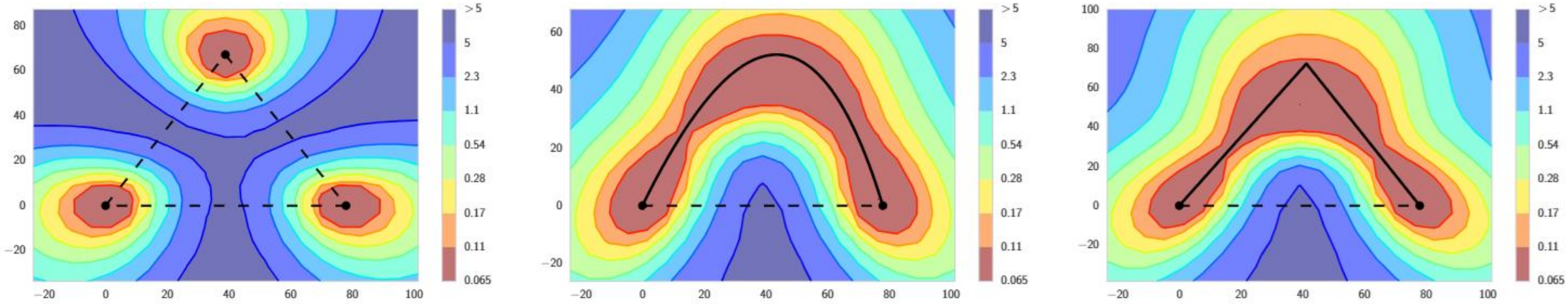
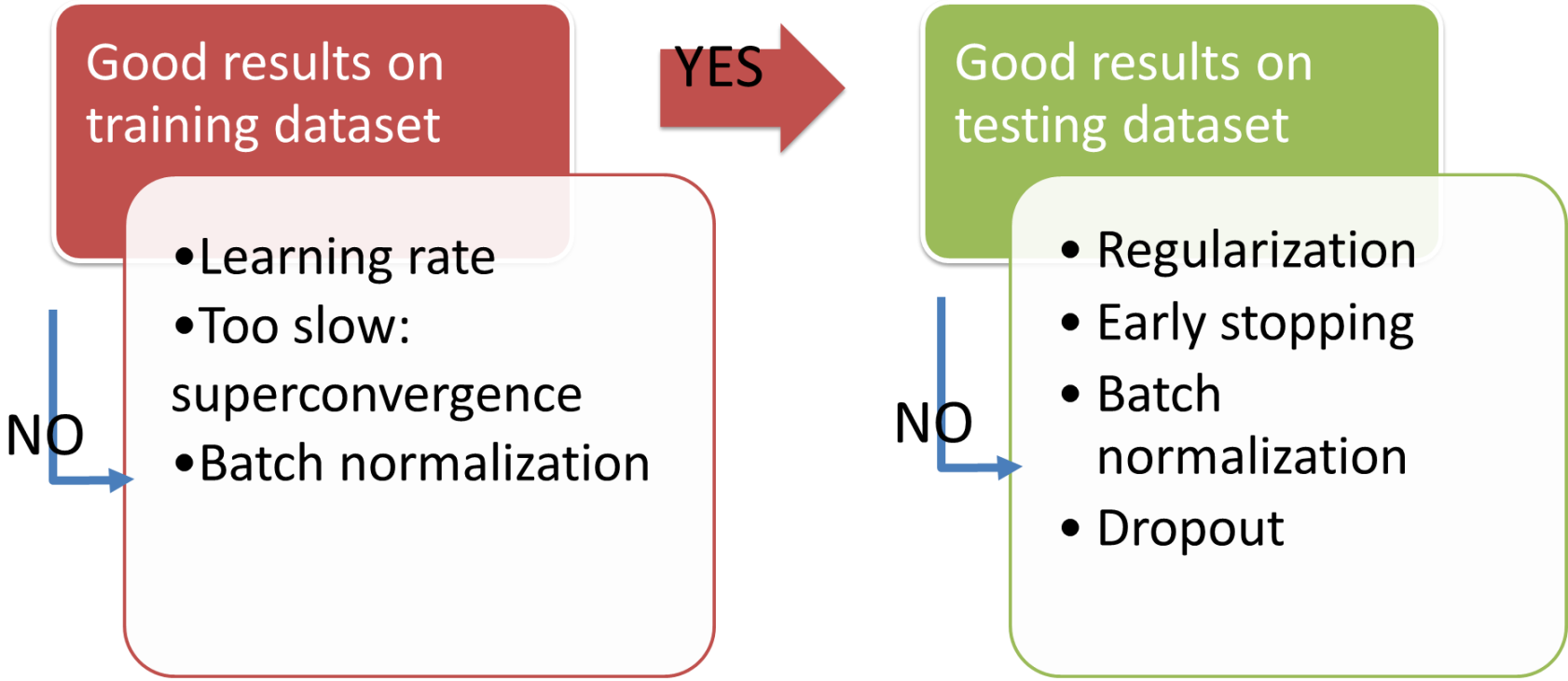


Figure 7. **Modes in the neural network landscape are connected along curves.** Three different two-dimensional subspaces of the ℓ_2-regularized cross-entropy loss landscape of a ResNet-164 on CIFAR-100 as a function of network weights. The horizontal axis remains fixed, anchored to the optima of two independently trained networks, while the vertical axis varies across panels. **Left:** Conventional assumption of isolated optima. **Middle** and **Right:** Alternative planes where optima are connected via simple curves while maintaining near-zero loss. *Mode connectivity* is relatively distinct to deep neural networks. Figure adapted from Garipov et al. (2018).

實務建議

- **檢查泛化能力**：一定要保留 validation/test 集。
- **避免過擬合的方法**：
  - 更多資料 (Data Augmentation, Synthetic data)
  - 正則化 (Dropout, Weight Decay, Early Stopping)
  - 適度的模型大小 (不一定小，反而過大模型 + 正則化更好)
- **理解「過參數 ≠ 必然過擬合」**：深度學習優化的隱式偏好，讓大網路仍能泛化。

- 傳統上，我們怕「模型太大 → 過擬合」。
  - 深度學習時，大模型不一定過擬合，因為 SGD 幫助找到泛化更好的解。
- 直覺上：
  - 背死書 = 過擬合
  - 理解原理 = 泛化
  - 平坦谷底 = 泛化，尖銳谷底 = 過擬合



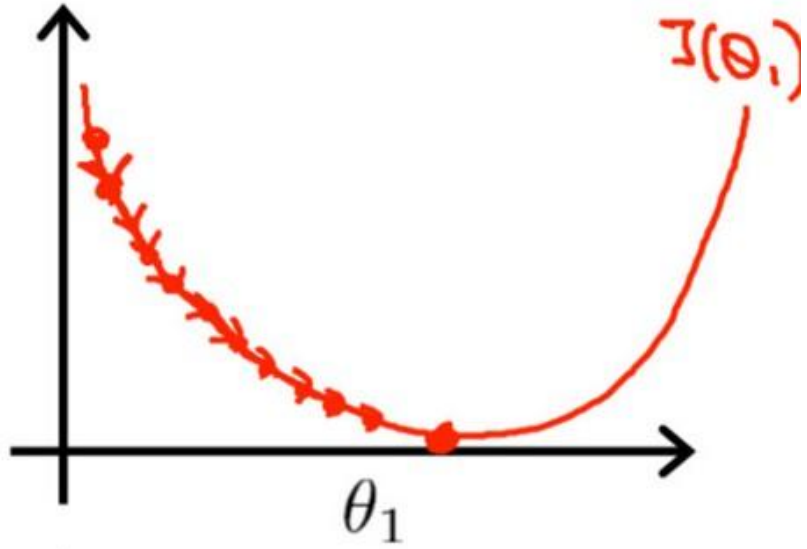
LEARNING RATE

Learning Rate

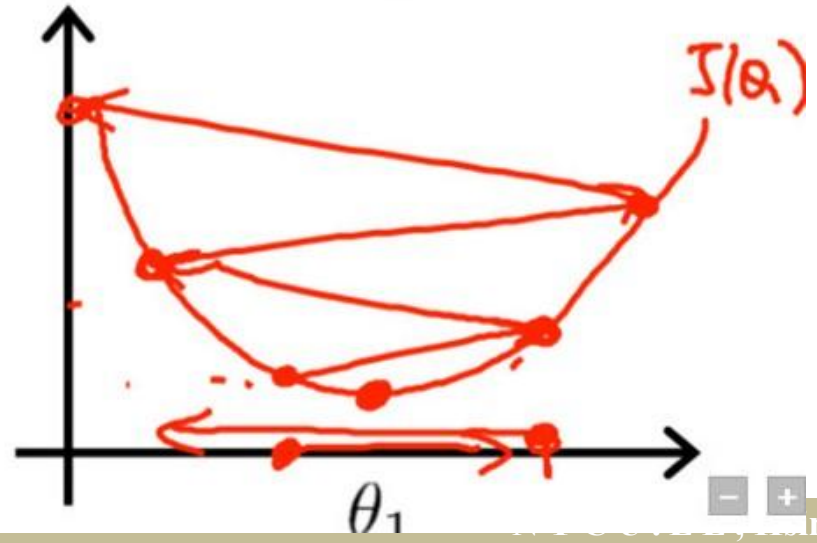
對收斂速度的影響

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

If α is too small, gradient descent can be slow.

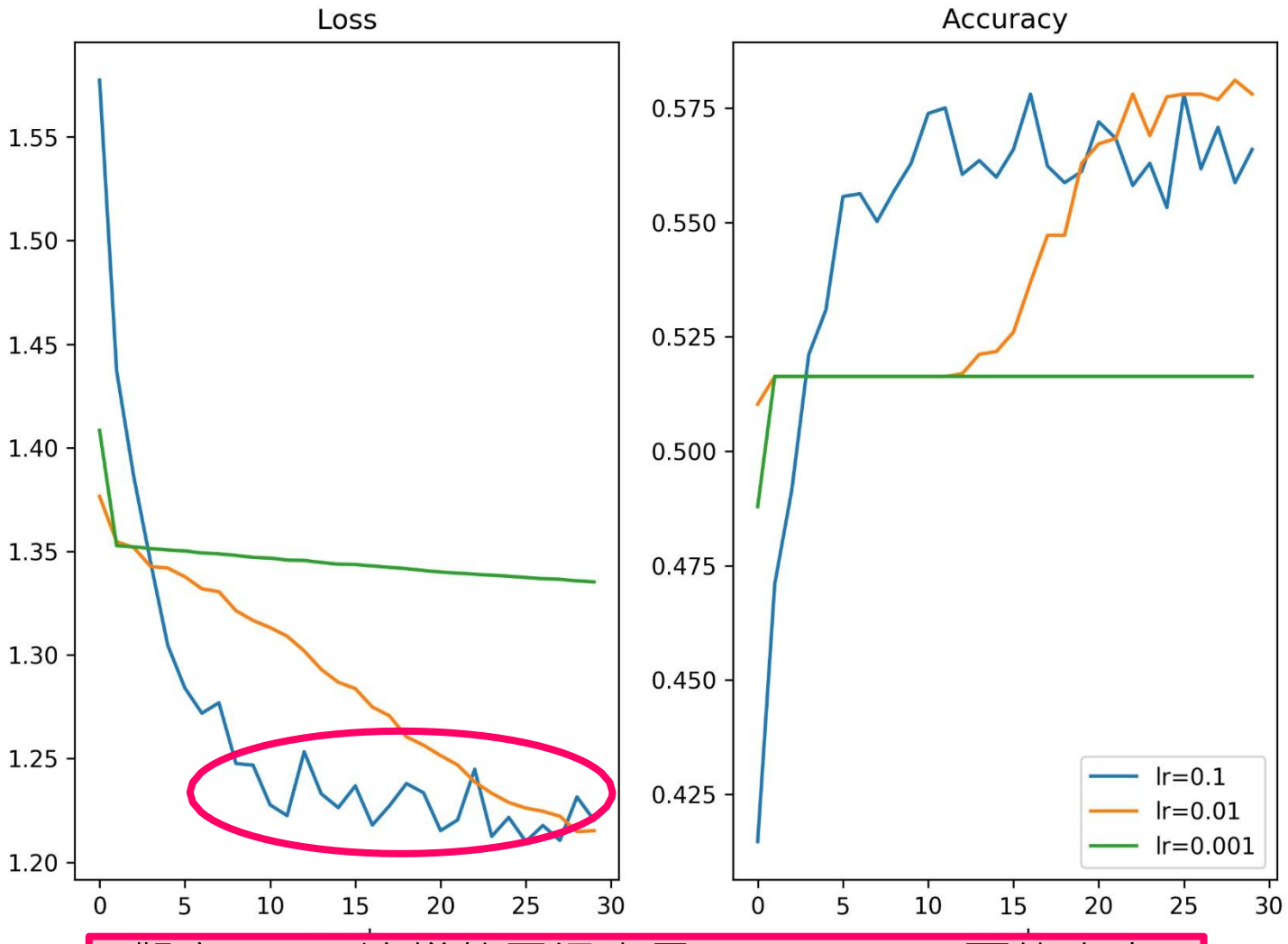


If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.



source: Andrew Ng

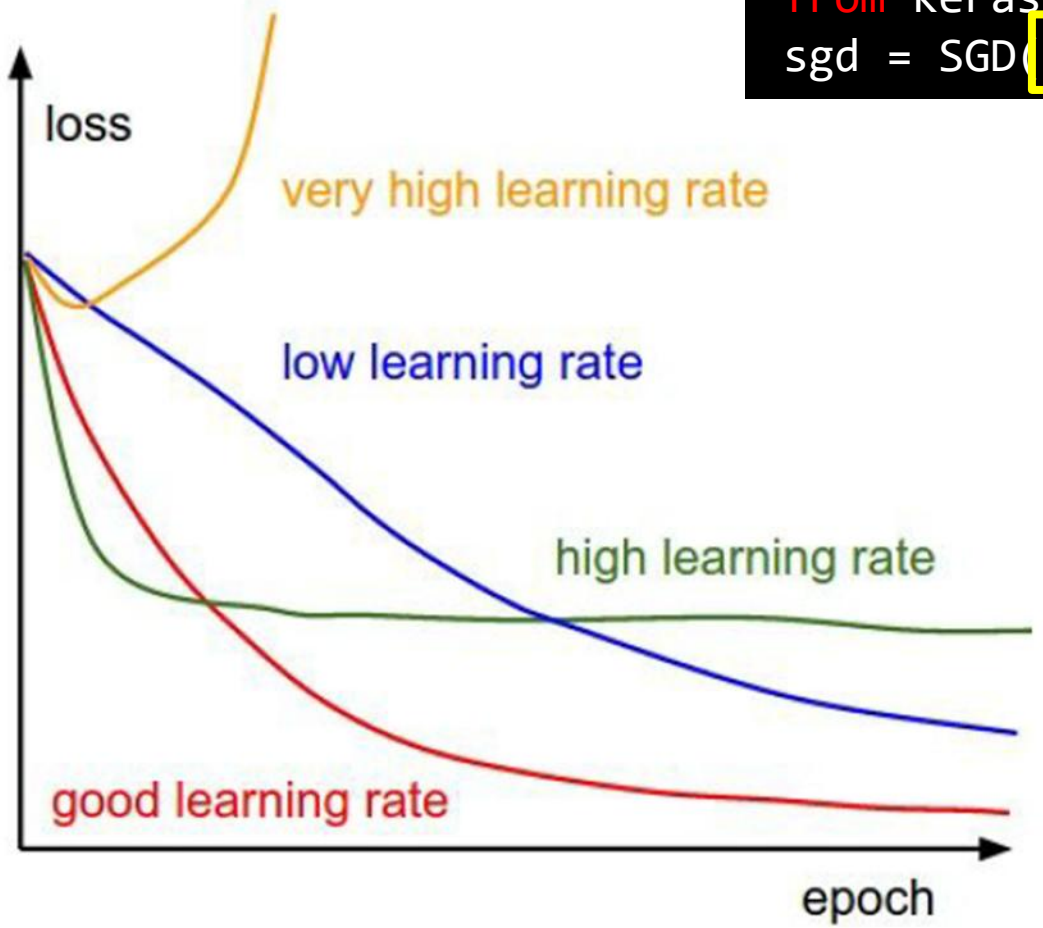
Learning Rate Selection



觀察 loss，這樣的震盪表示 learning rate 可能太大

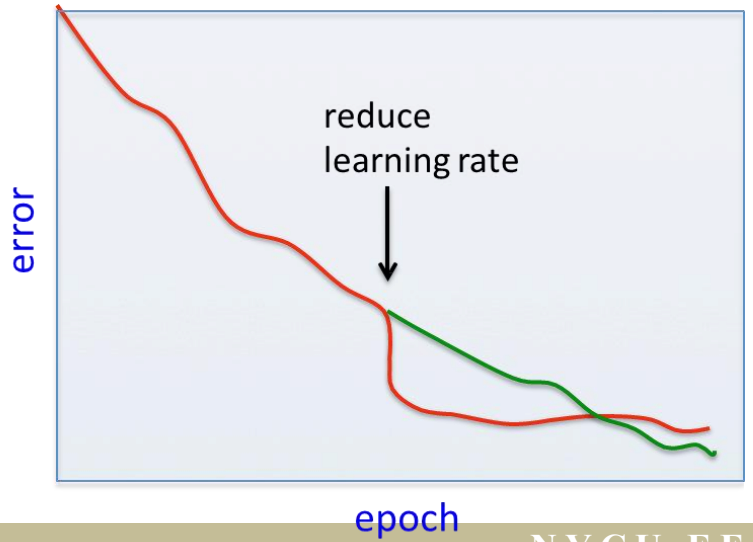
Learning Rate and decay

```
# 指定 optimizier
from keras.optimizers import SGD, Adam, RMSprop, Adagrad
sgd = SGD(lr=0.01,momentum=0.0,decay=0.0,nesterov=False)
```



建議一次降一個數量級，如: 0.01 vs 0.001，挑選出最好的 learning rate

一開始較大，後面較小避免上下震盪



如何設Learning Rate

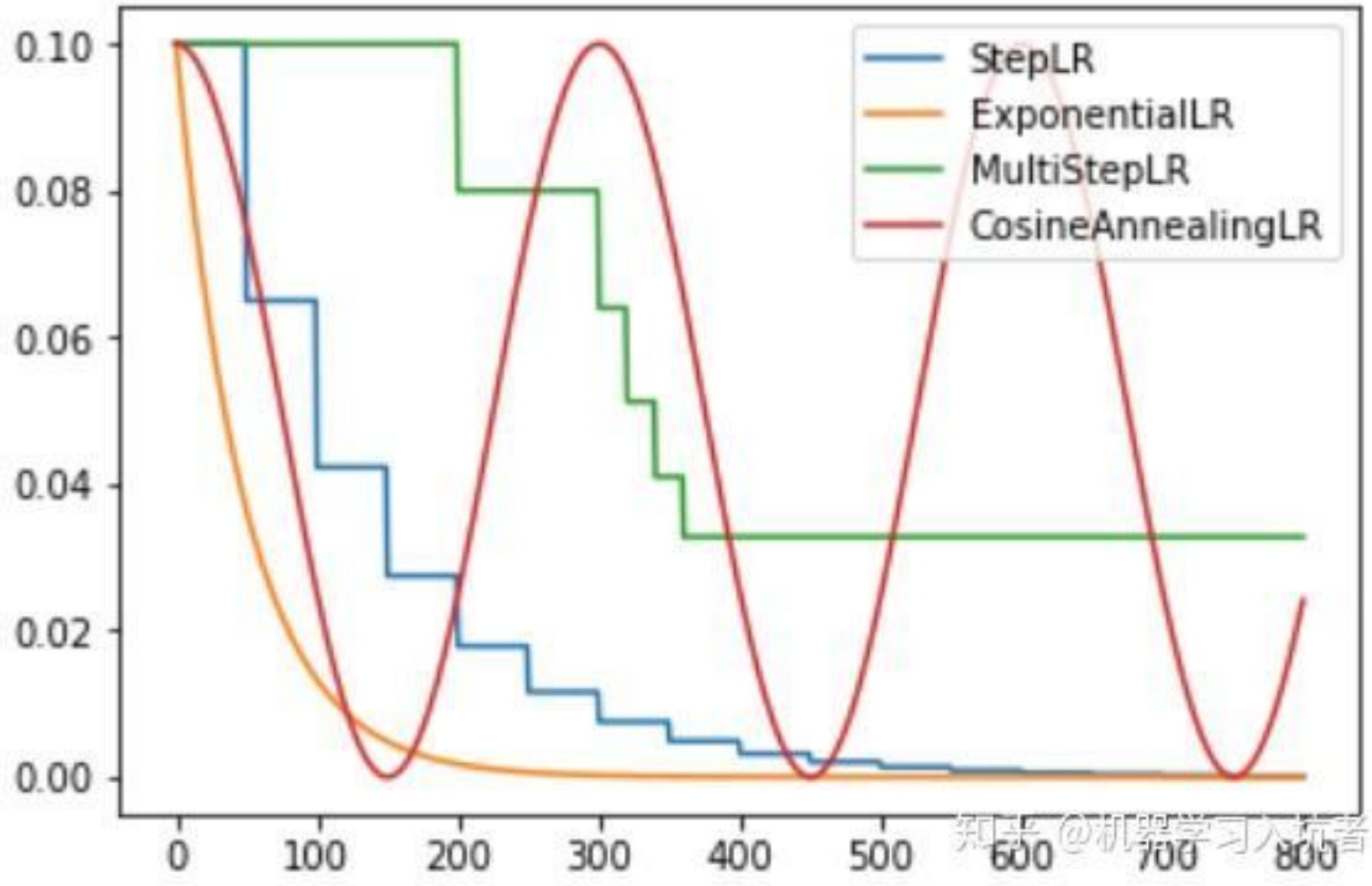
- 經驗值
  - 大多要試試看才知道，通常不會大於 0.1
  - 一次調一個數量級: 0.1 => 0.01 => 0.001，
- 初始值
  - 0.01 或 0.001 還蠻常用的
- 訓練過程中，還可以再降 learning rate decay
  - 10^{-4}, 10^{-5} 也有可能

- Strategy
  - we hope that the learning rate in the early stage of training will be larger, so that the network will converge quickly,
  - and the learning rate will be smaller in the later stage of training, so that the network will better converge to the optimal solution

LEARNING RATE DECAY

Source: Practical Recommendations for Gradient-Based Training of Deep Architectures by Yoshua Bengio

Learning Rate Decay



Learning Rate Schedules

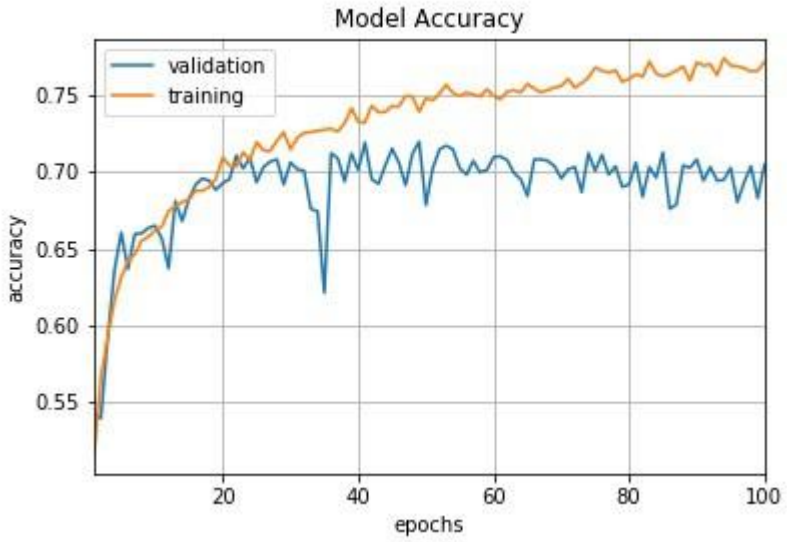
- Constant Learning Rate

```
keras.optimizers.SGD(lr=0.1, momentum=0.0, decay=0.0, nesterov=False)
```

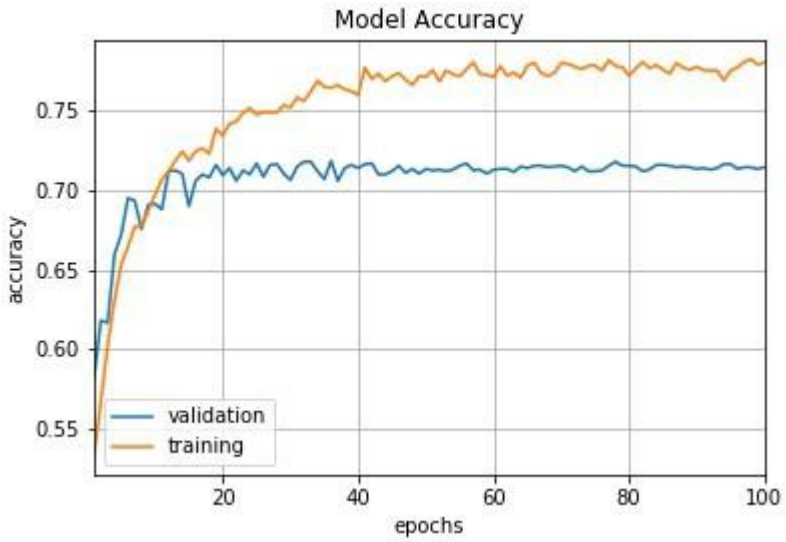
- time-based decay
  - Math form `lr = lr0/(1+kt)`
    - `lr, k are hyperparameters and t is the iteration number`

```
learning_rate = 0.1
decay_rate = learning_rate/epochs
sgd = SGD(lr=0.1, momentum=0.8, decay=decay_rate, nesterov=False)
```

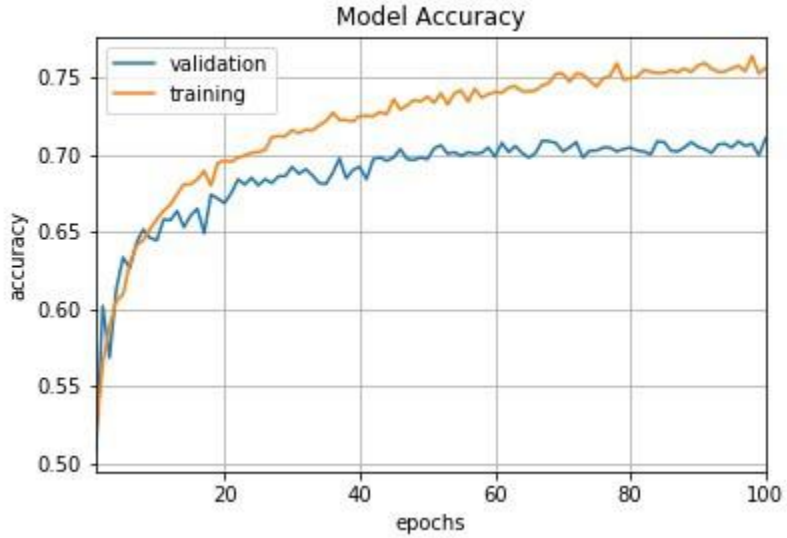
- step decay
  - drops the learning rate by a factor every few epochs
  - Math form `lr = lr0 * drop^floor(epoch / epochs_drop)`
- exponential decay
  - Math form `lr = lr0 * e^(-kt)`
- Manual decay



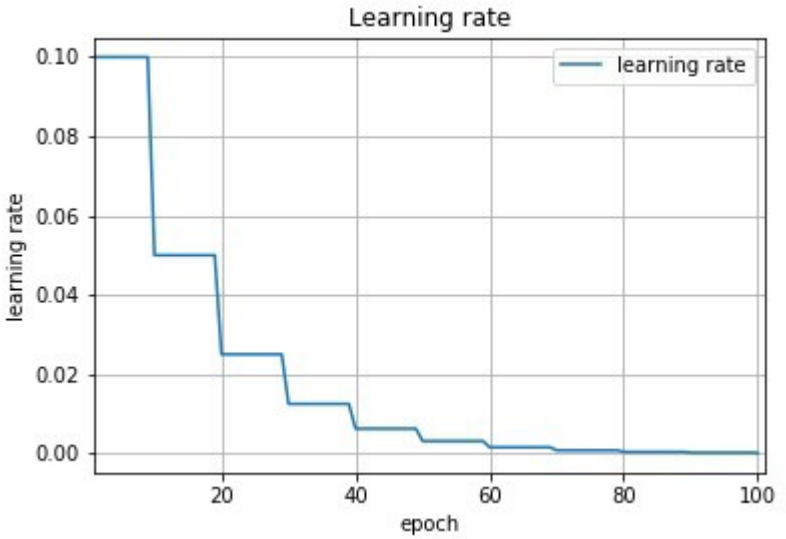
Constant Learning Rate



step decay

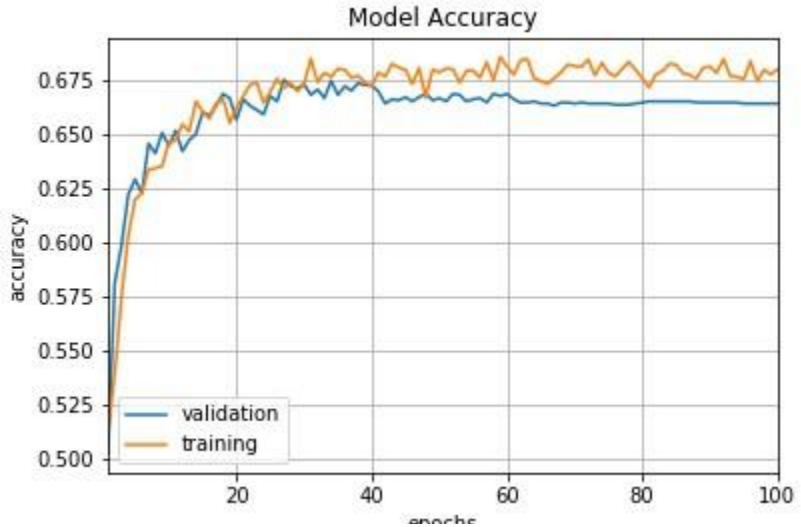


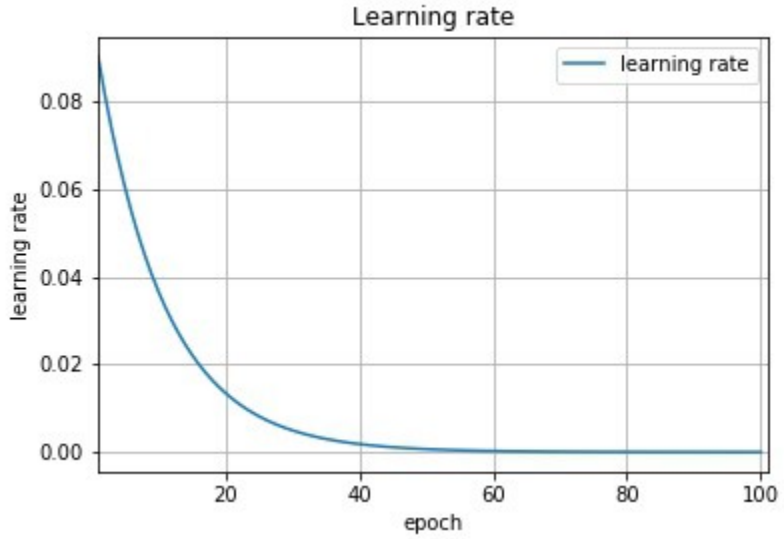
time-based decay



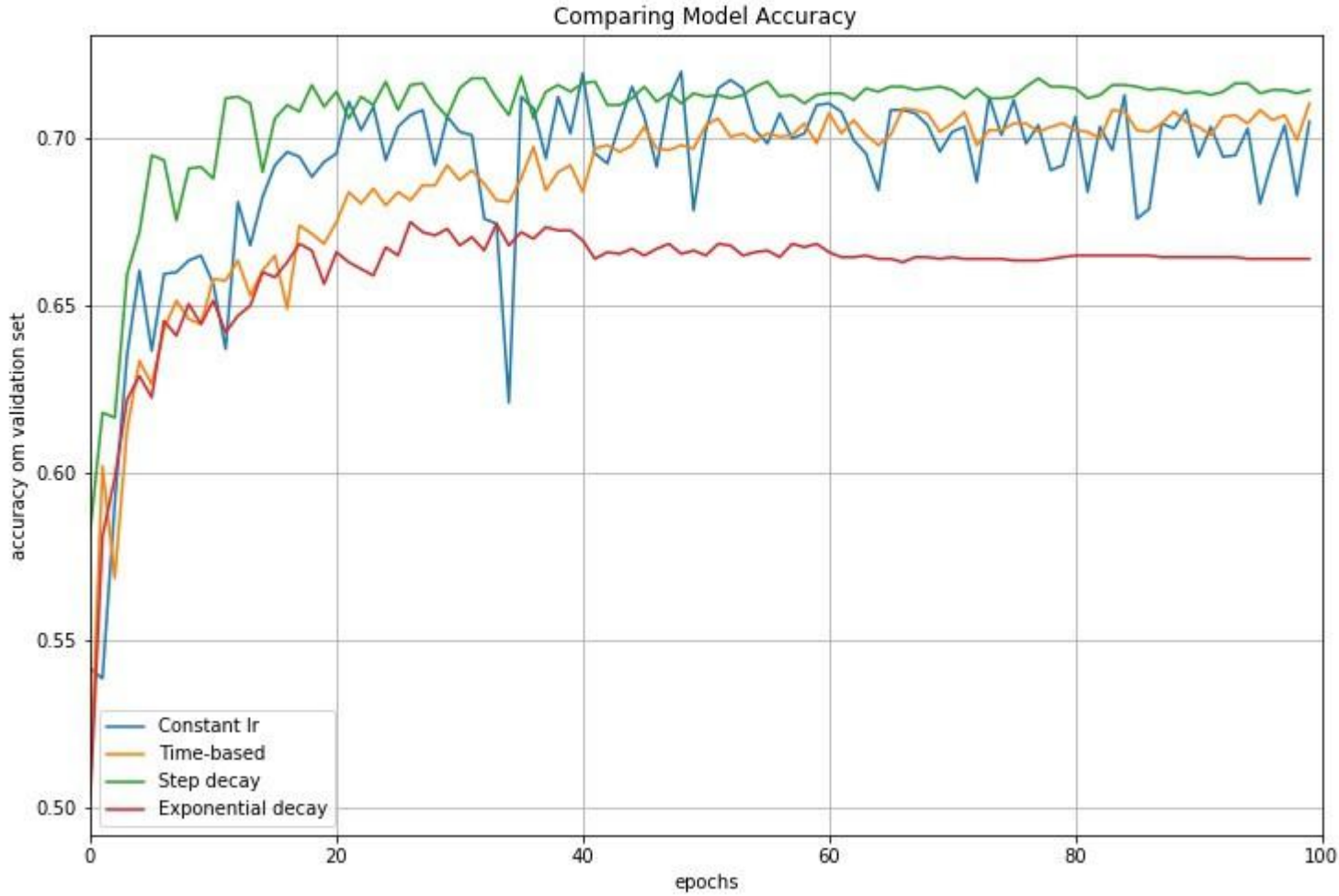
step decay schedule

Exponential rate decay



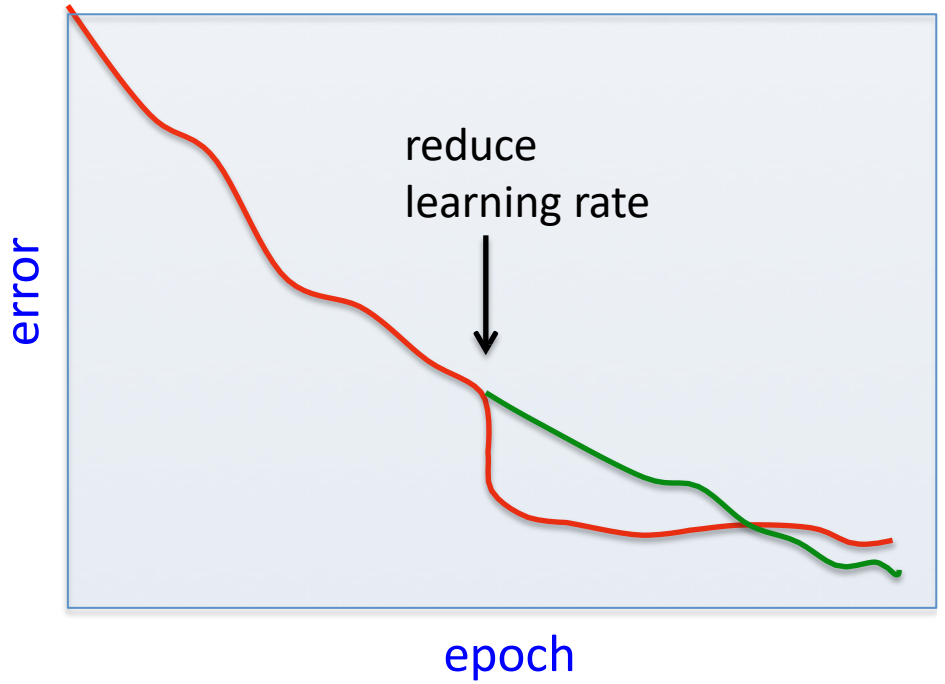


Step decay has the best accuracy and stable learning curve

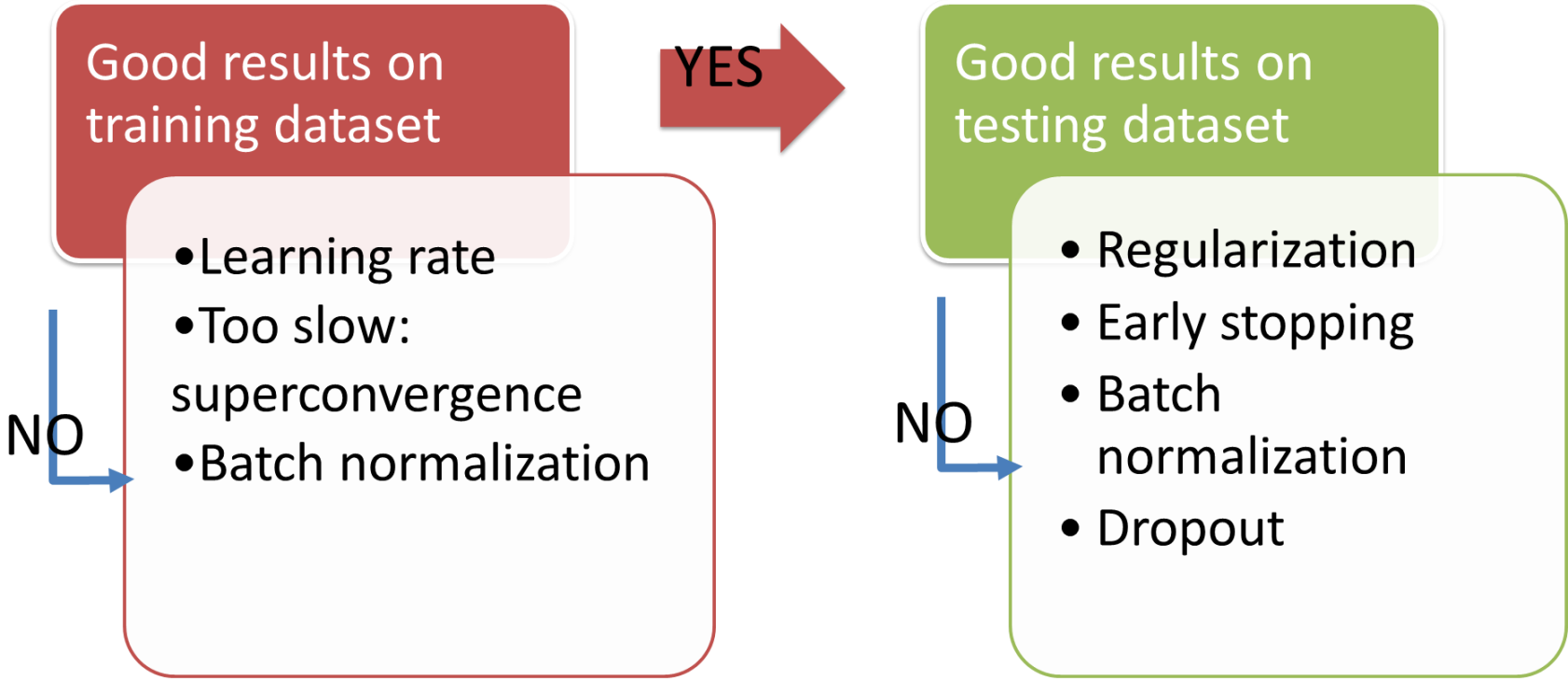


Tips

- Use constant learning rate first to get a good result
- Be careful about turning down the learning rate
  - Low learning rate reduce fluctuation error
  - But slow down learning process as well
  - Don't turn down the learning rate too soon!

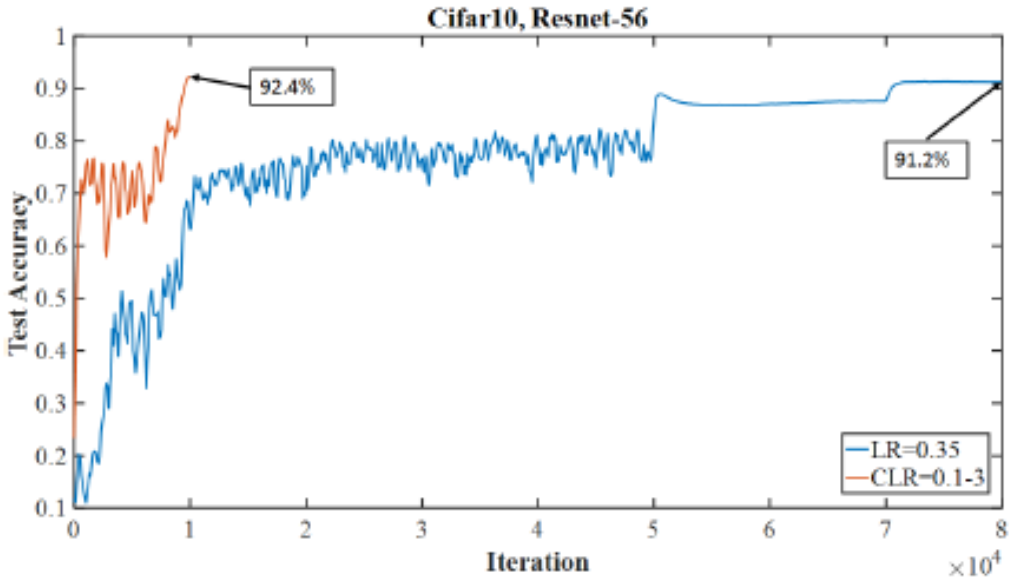


Source: Hinton, neural network for machine learning, lecture 6a

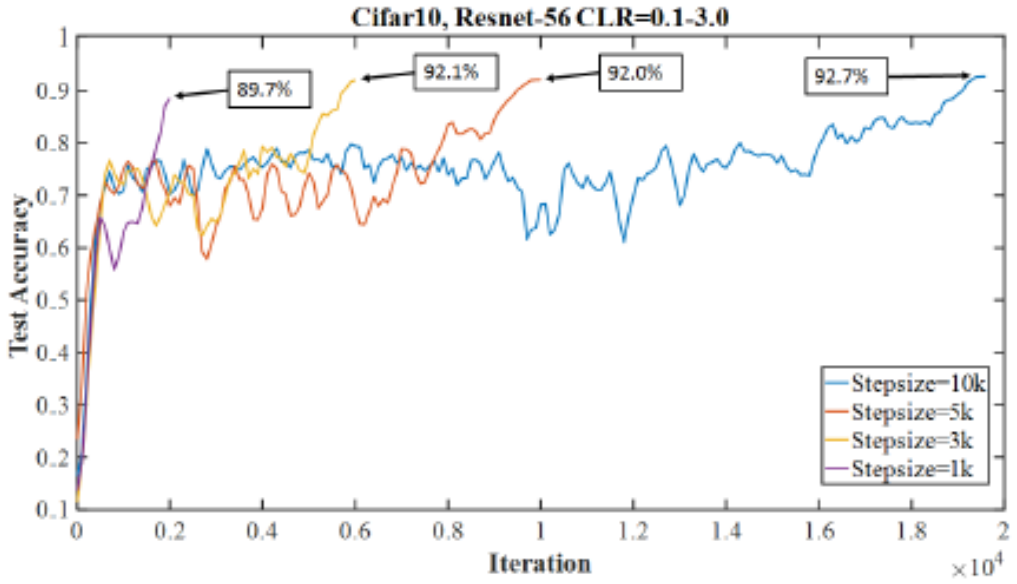


CONVERENCE TOO SLOW: SUPERCONVERGENCE

SuperConvergence: Training Using Large LR



(a) Comparison of test accuracies of super-convergence example to a typical (piecewise constant) training regime.



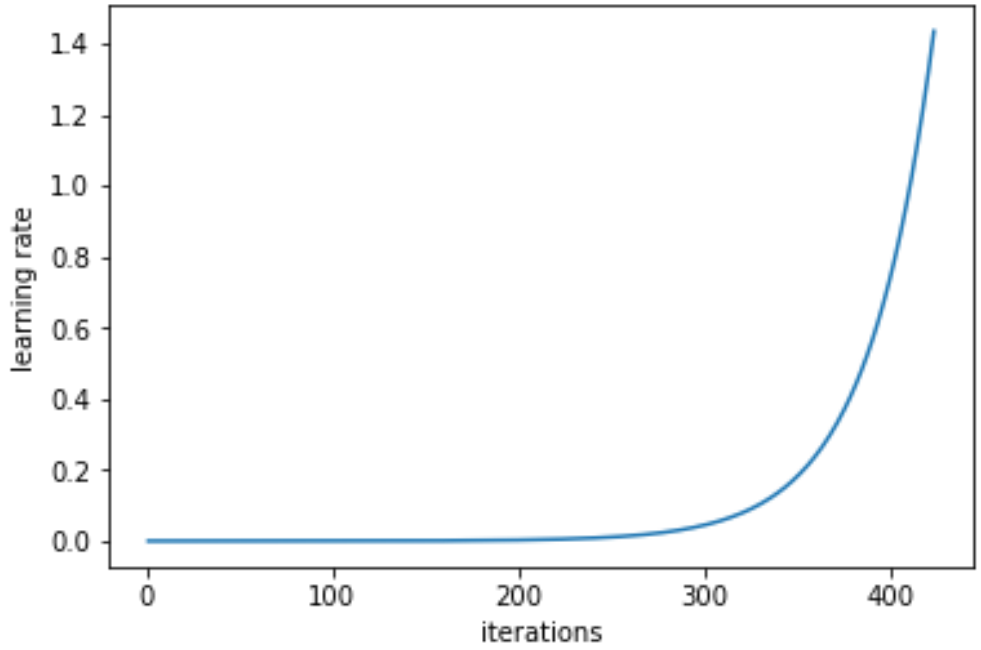
(b) Comparison of test accuracies of super-convergence for a range of stepsizes.

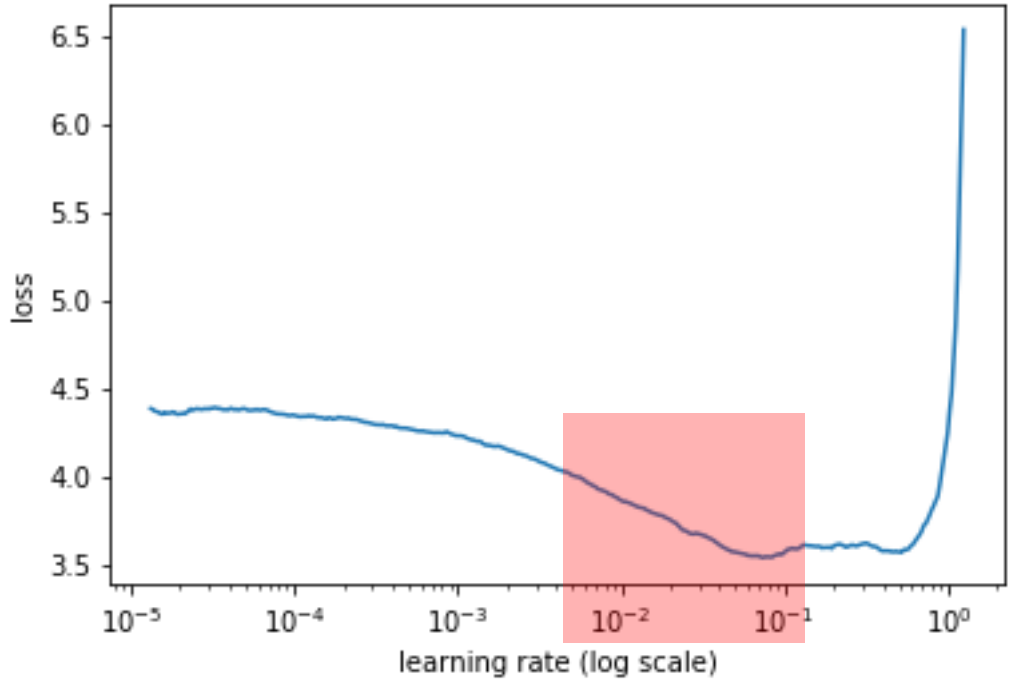
Figure 1: Examples of super-convergence with Resnet-56 on Cifar-10.

Smart Way to Set Learning Rate

train a network starting from a low learning rate and increase the learning rate exponentially for every batch

Record the learning rate and training loss for every batch





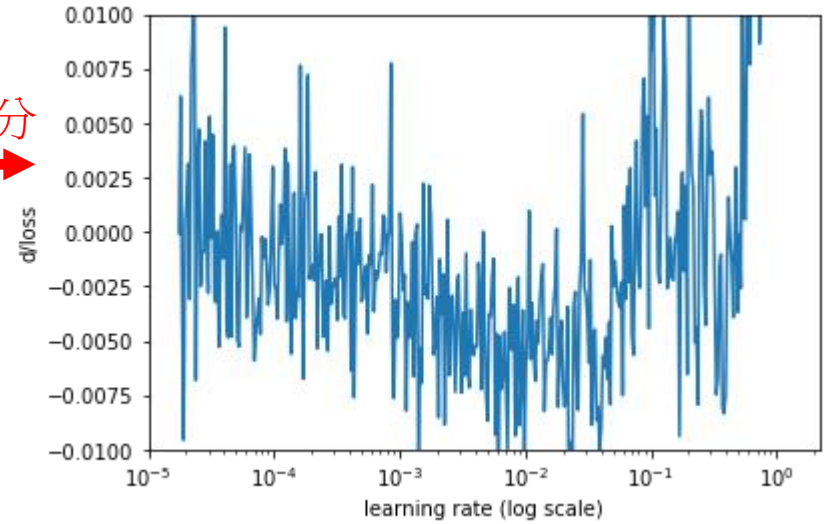
選loss 下降最多的點

Cyclical Learning Rates for Training Neural Networks

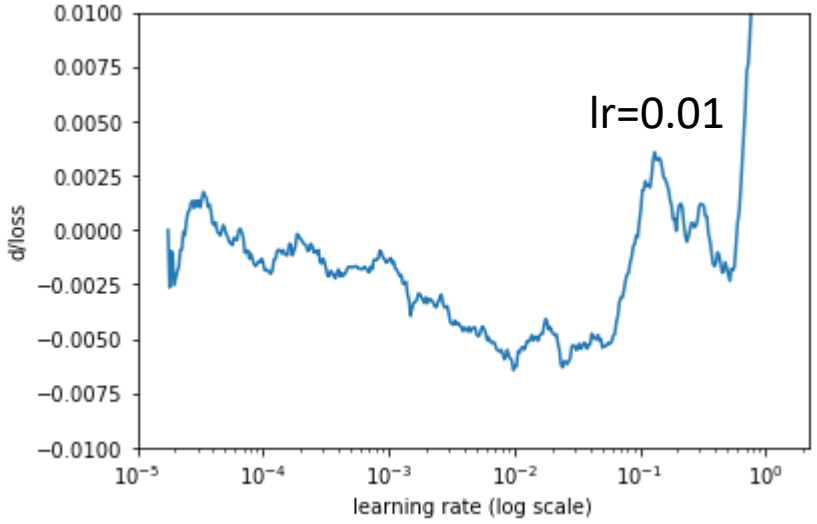
Smart Way to Set Learning Rate

calculating the rate of change of the loss (a derivative of the loss function with respect to iteration number)

微分



too noisy, let's smooth it out using simple moving average



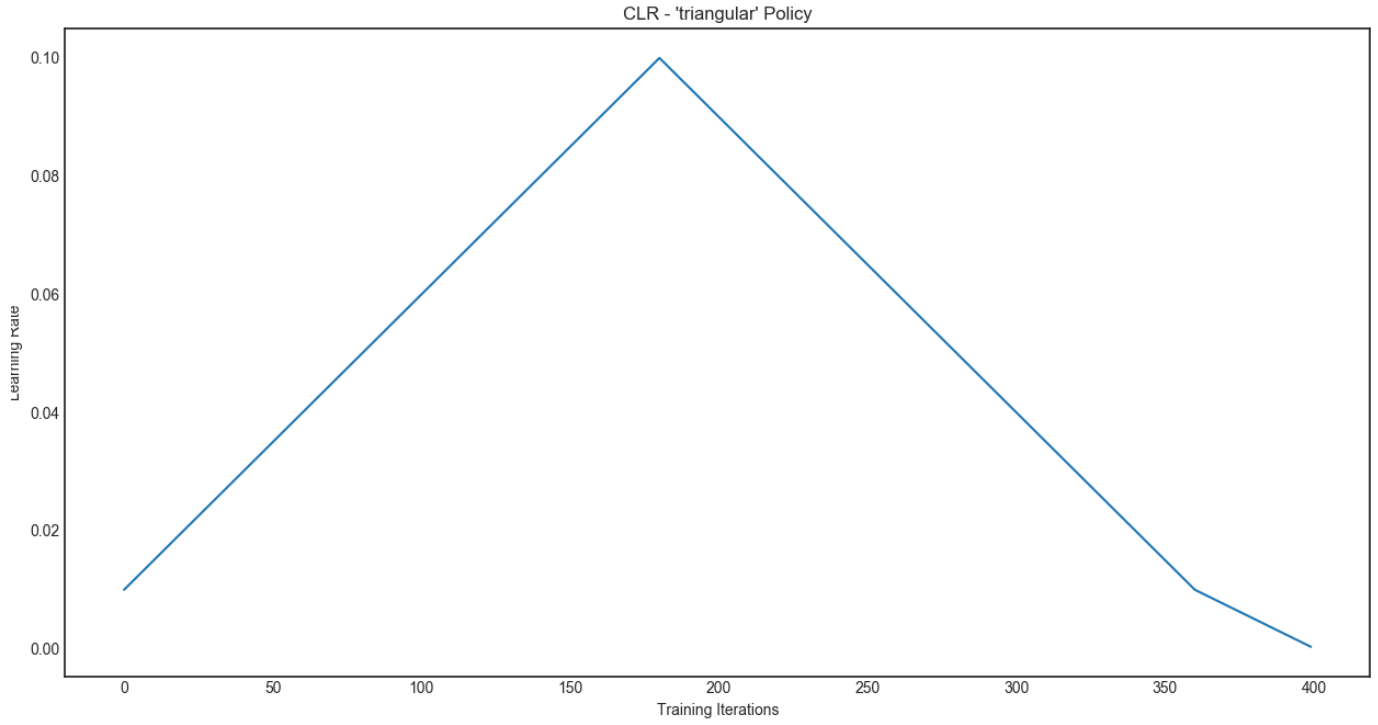
learn.lr_find()
learn.sched.plot_lr()

rerun the same learning rate search procedure periodically to find the learning rate at a later point in the training process

Pytorch: fast.ai https://github.com/fastai/fastai
Keras: https://github.com/surmenok/keras_lr_finder

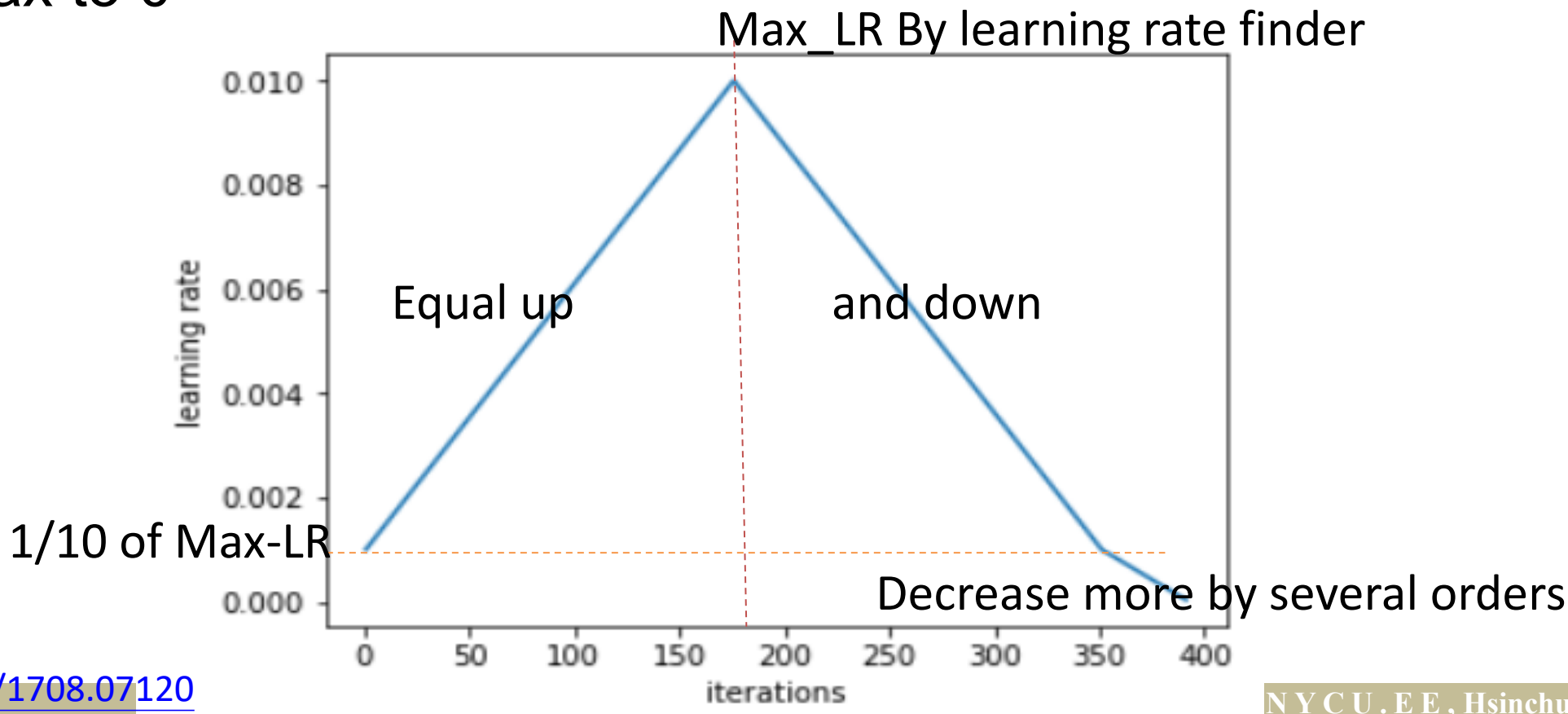
One Cycle Learning Rate Policy

- Cyclic Learning Rate (CLR): Learning rate schedule



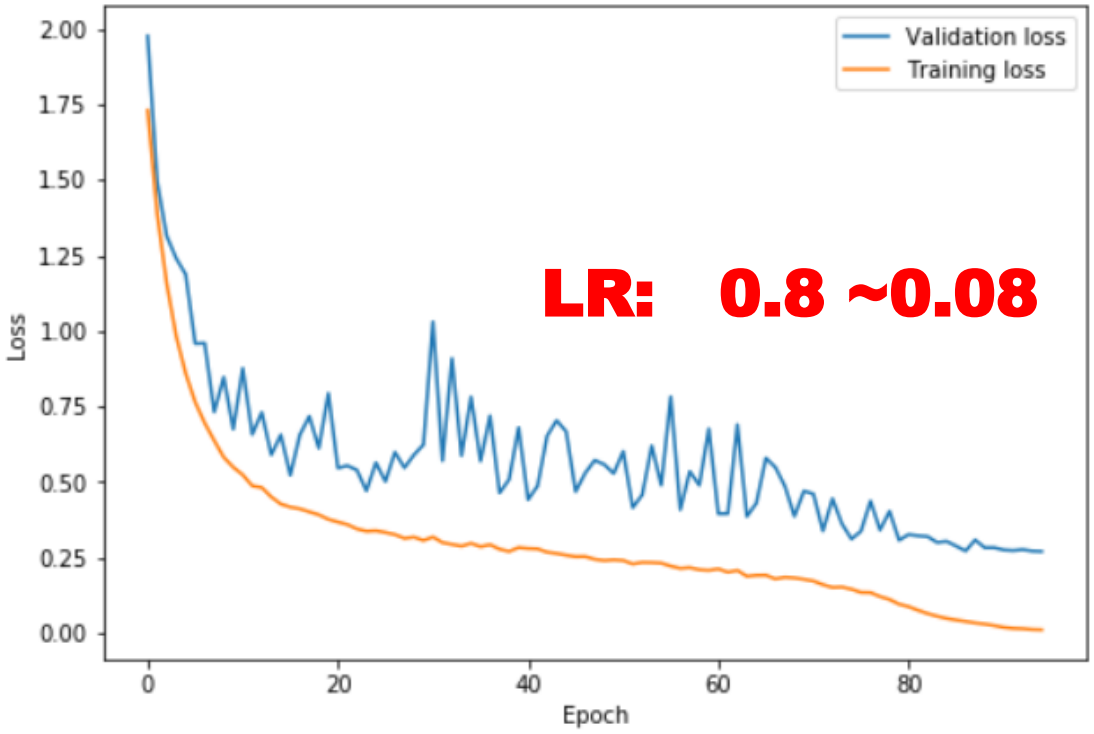
Cyclical Learning Rate aka. Super Convergence

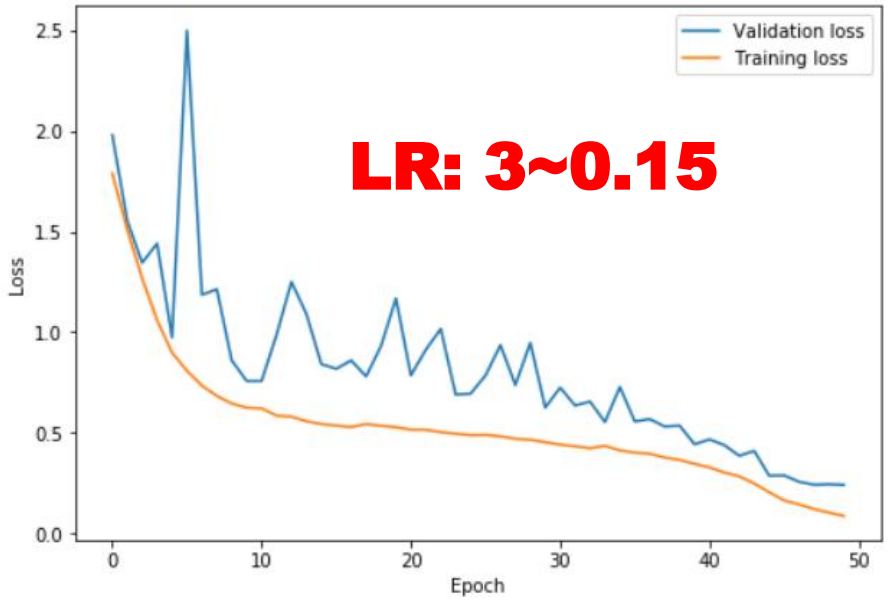
- Use high learning rate, but need some warmup
- Two phase is even better, with the second phase use cosine annealing for lr_max to 0

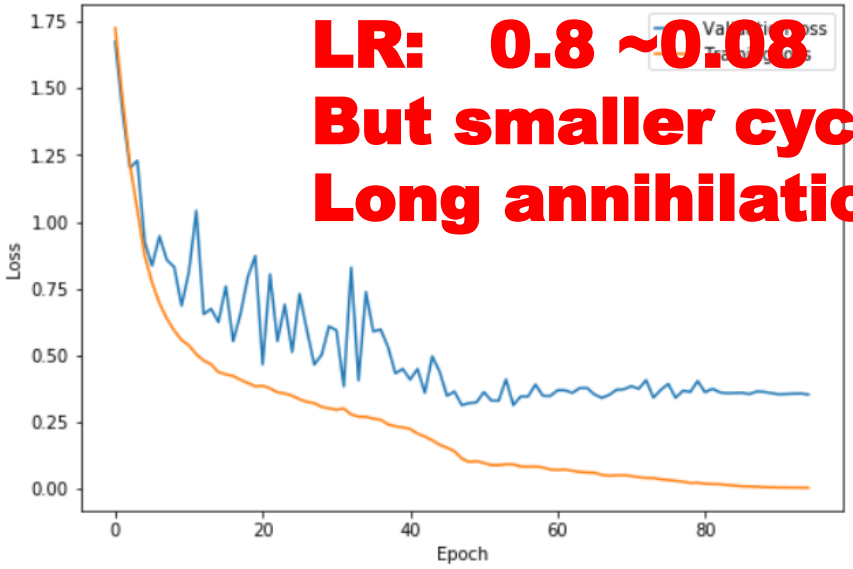


https://arxiv.org/abs/1708.07120

FastAI: The 1cycle policy

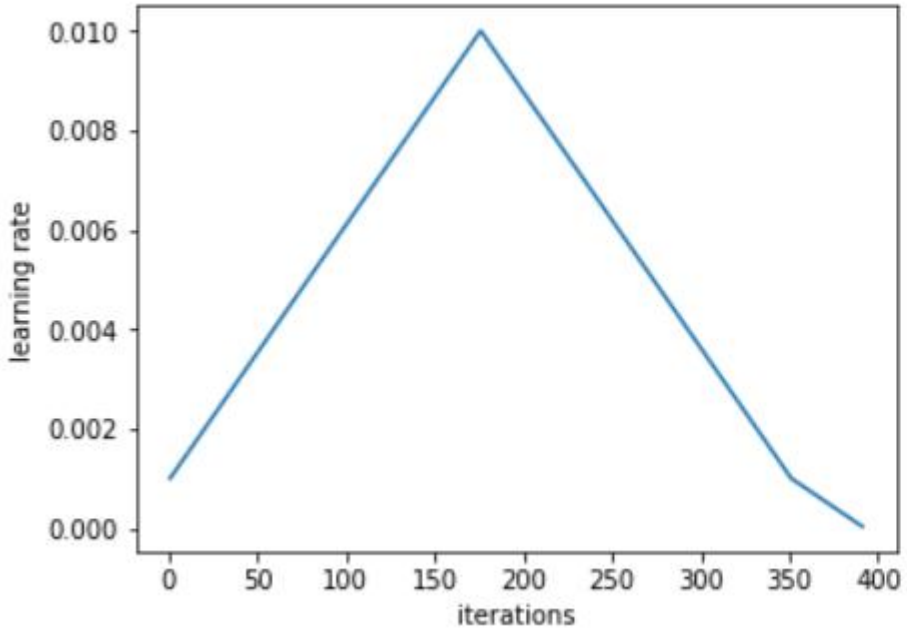


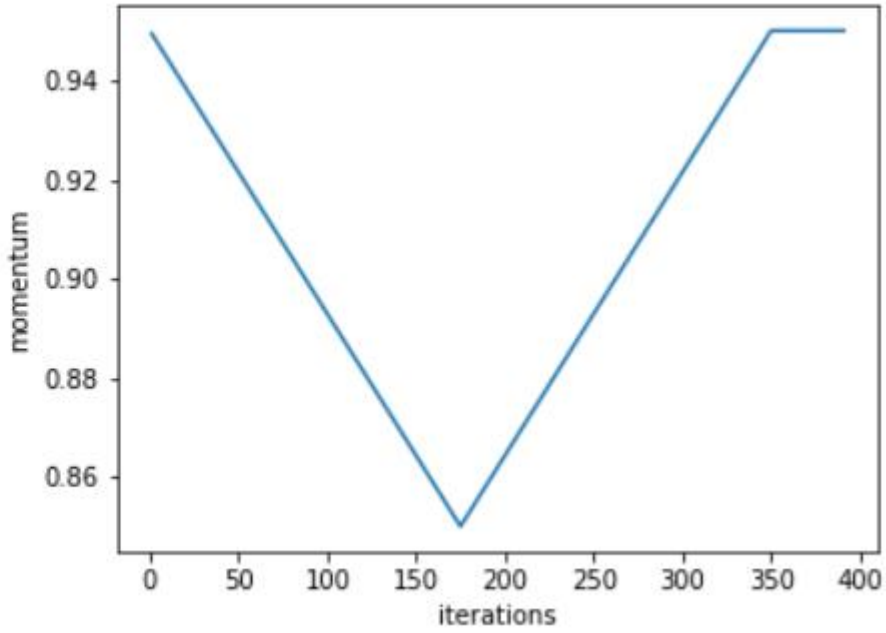




The 1cycle policy

- Cyclical momentum
  - When toward large learning rate, decrease the momentum
  - Momentum: 0.85 ~ 0.95 and keep flat when annihilation



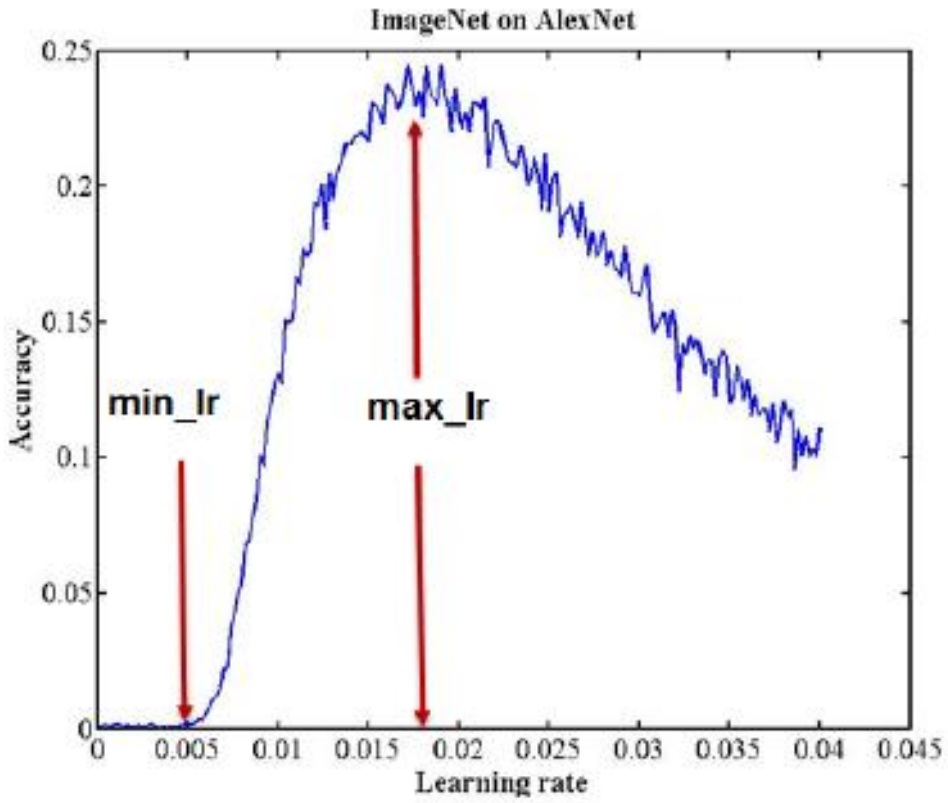


The 1cycle policy

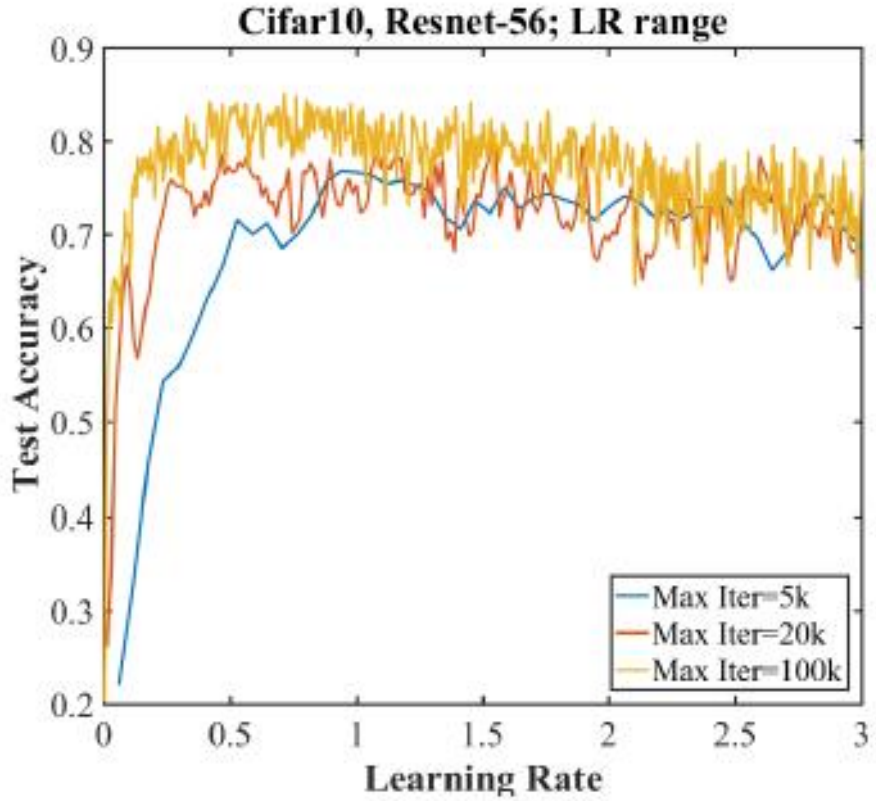
- Other hyper-parameters
  - Batch size: set the high one to fit in the available memory
  - Weight decay
    - run the learning rate finder for a few values of weight decay, and pick the largest one that will still let us train at a high maximum learning rate
  - Dropout
    - Same as above approach for weight decay

- Concept behind this
  - Training with the 1cycle policy at high learning rates is a method of regularization

- large learning rates regularize the training



(a) Typical learning rate range test result where there is a peak to indicate max_lr.



(b) Learning rate range test result with the Resnet-56 architecture on Cifar-10.

Learning Rate Scheduler with Warm Start

- CyclicLR – triangular

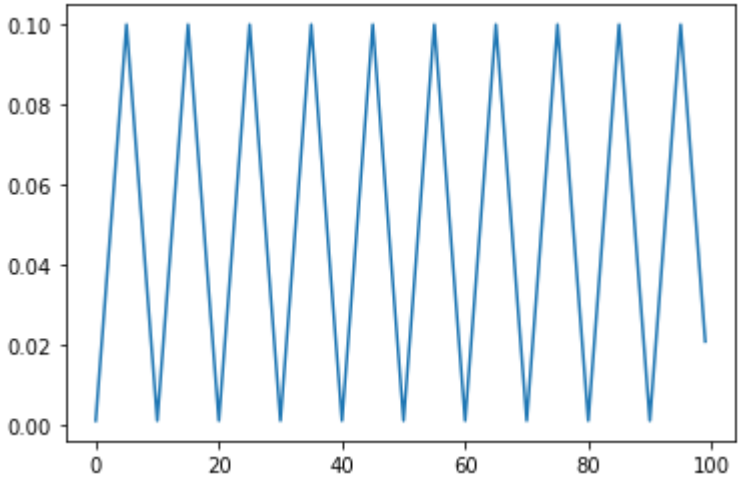
```
scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer, base_lr=0.001, max_lr=0.1, step_size_up=5,
mode="triangular")

scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer, base_lr=0.001, max_lr=0.1, step_size_up=5,
mode="triangular2")
```

- OneCycleLR - cos

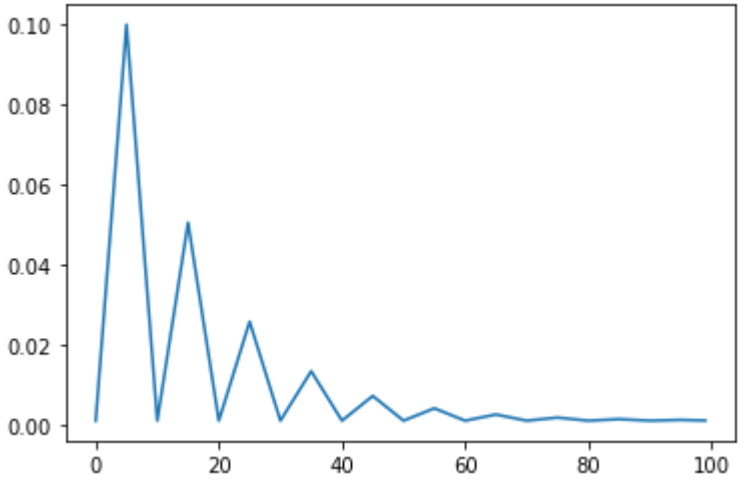
```
scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.1, steps_per_epoch=10, epochs=10,
anneal_strategy='linear')
```


https://www.kaggle.com/isbhargav/guide-to-pytorch-learning-rate-scheduling

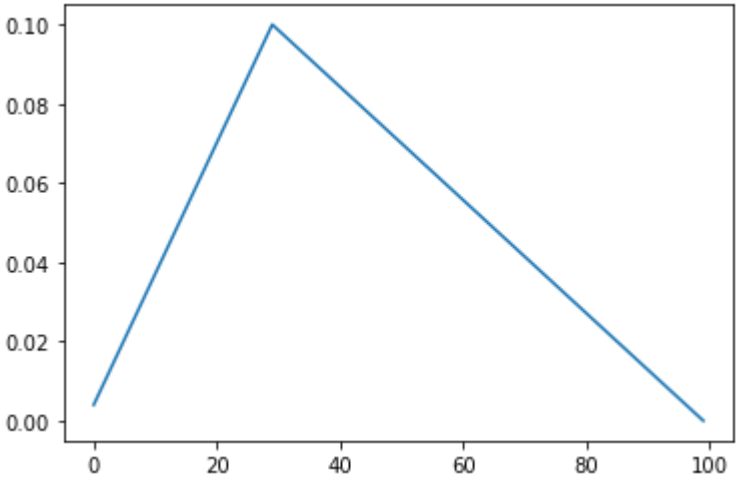
CyclicLR - triangular



CyclicLR - triangular2



OneCycleLR - linear



Train Imagenet in 18 minutes

- Progressive resizing
  - using small images at the start of training, and gradually increasing size as training progresses.
  - That way, when the model is very inaccurate early on, it can quickly see lots of images and make rapid progress, and later in training it can see larger images to learn about more fine-grained distinctions.
- Dynamic batch size
  - use larger batch sizes for some of the intermediate epochs – this allowed us to better utilize the GPU RAM and avoid network latency.
- Remove weight decay from BN layers

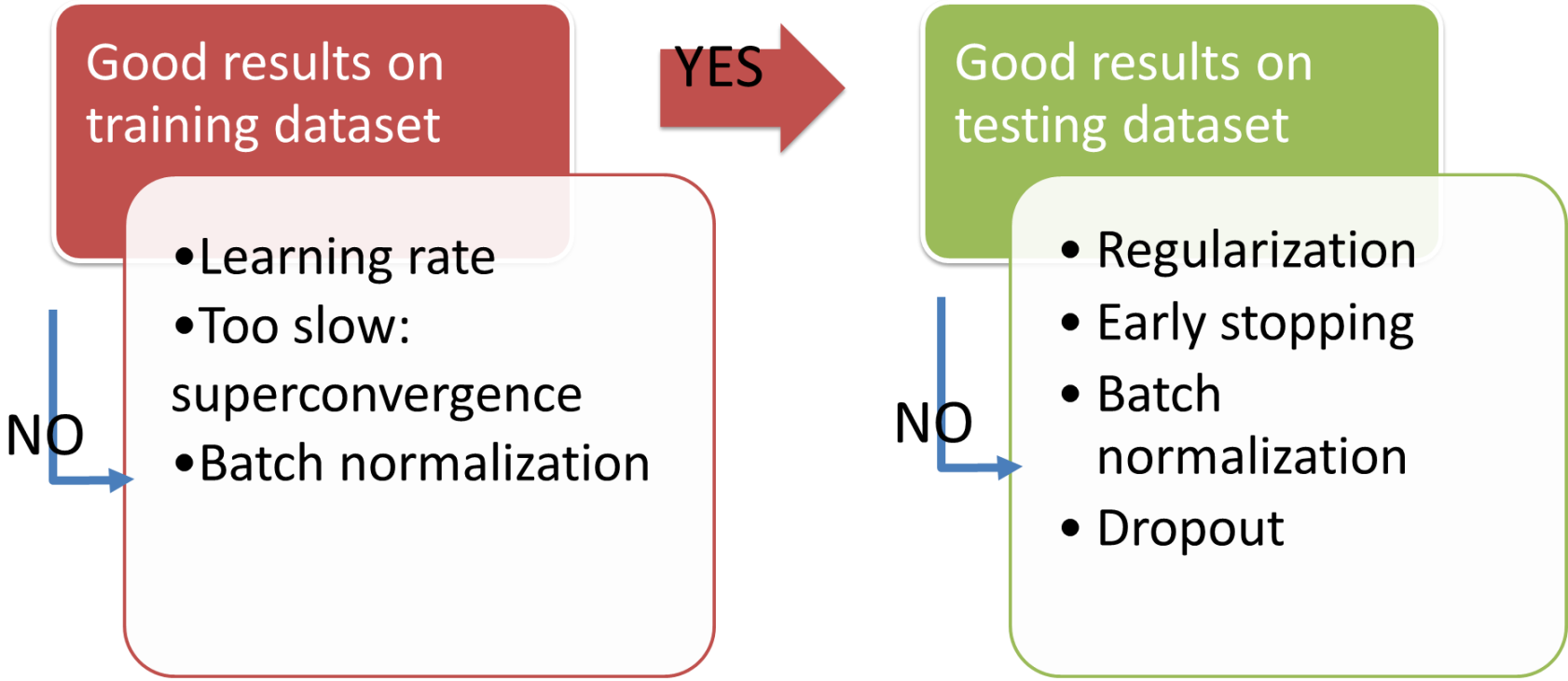
如何挑選 Learning Rate

- **先做 Learning Rate Finder**
  - 嘗試從很小的 LR 開始，逐步增加，觀察 loss 何時開始下降、何時開始爆炸。
  - 在下降區間內取一個最佳值。
- **依模型與任務選擇策略**
  - **CNN 圖像分類**：常用 step decay / cosine annealing。
  - **Transformer / LLM**：幾乎都用 warmup + cosine。
  - **小數據或收斂困難**：可嘗試 cyclical learning rate。
- **典型數值範圍**
  - Adam/AdamW：10^{-5}~10^{-3}
  - SGD (有 momentum)：10^{-3}~10^{-1}

策略	何時用	典型參數與做法	優點	注意事項
固定 LR	小模型/短訓練、穩定任務	AdamW: 1e-4~3e-4；SGD: 1e-2~1e-1	簡單	常錯過更好極值
Step Decay	傳統 CNN from scratch	step_size=30, gamma=0.1	易用有效	不連續跳變
Exponential	需平滑下降	k 依總步數反解	連續平滑	可能過早太小
Cosine	CV/NLP 通用、長訓練	T_max=總步數，eta_min=1e-6~1e-5	收斂穩定	需搭 warmup
Cosine + Restarts	多階段訓練/超參搜尋	週期縮短或保持	多次再啟探索	調參較多
Warmup（線性）	Transformer/LLM/ViT 幾乎必備	前 1%~5% 步數線性升至峰值	防初期發散	峰值 LR 要搭配 batch
One-Cycle	單階段加速收斂	max_lr 設為 LR Finder 峰值附近	快速且常準	需估總步數
ReduceLROnPlateau	指標停滯時自動降 LR	patience=3~5，factor=0.1	自調整	反應落後、噪聲敏感
Cyclical (CLR)	想避免局部谷	base_lr~max_lr 循環	探索性強	可能抖動

最新進展

- **One-cycle policy**：
  - 結合 cyclical + warmup，效果在 NLP/vision 都很不錯。
- **Adaptive LR (AdaFactor, Lion, Shampoo)**：
  - 部分優化器自帶動態調整 LR 的機制。
- **LLM scaling law**：
  - 大模型訓練有推薦的 learning rate 與 batch size 對應規則（例如 linear scaling rule）。



BATCH NORMALIZATION

Batch Normalization

- Problem
  - Model training is hard due to a lot of parameter tunings
    - Could we simplify this?
  - Deep models composes many layers. Weight update of one layer depends strongly on all the other layers
- Concept: inspired by input normalization
  - "you want unit Gaussian activations? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

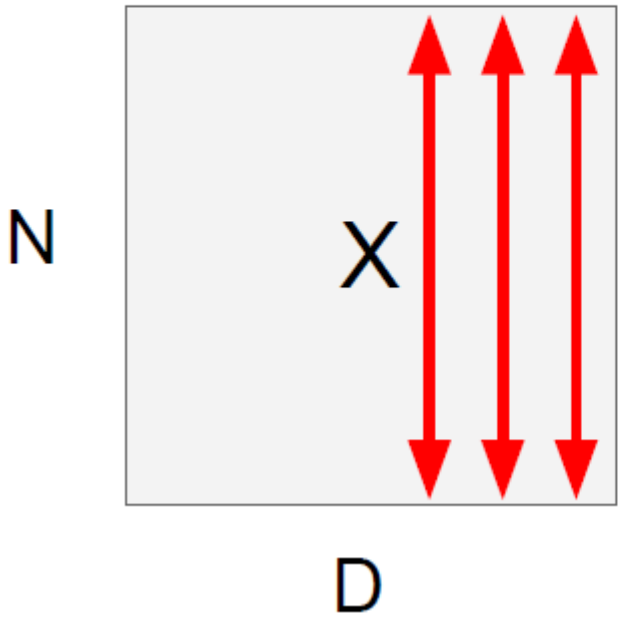
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

Source: CS231n

Batch Normalization

"you want unit gaussian activations?
just make them so."



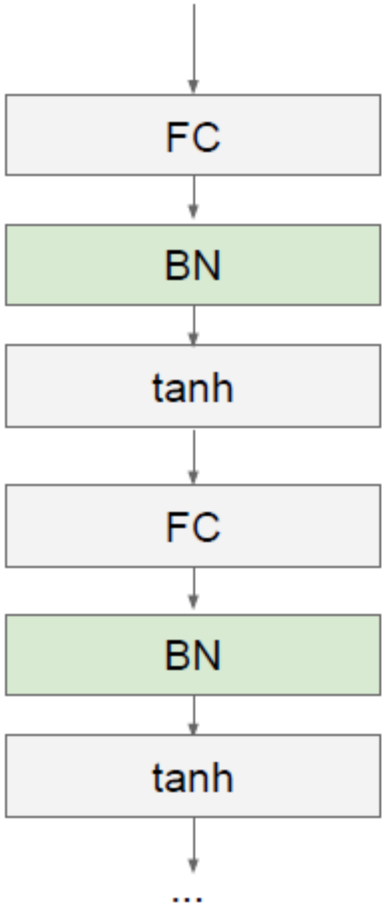
1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Source: CS231n

Batch Normalization



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Source: CS231n

Batch Normalization

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

γ，β are learned parameters to allow the new variable to have any mean and standard deviation

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

Source: CS231n

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

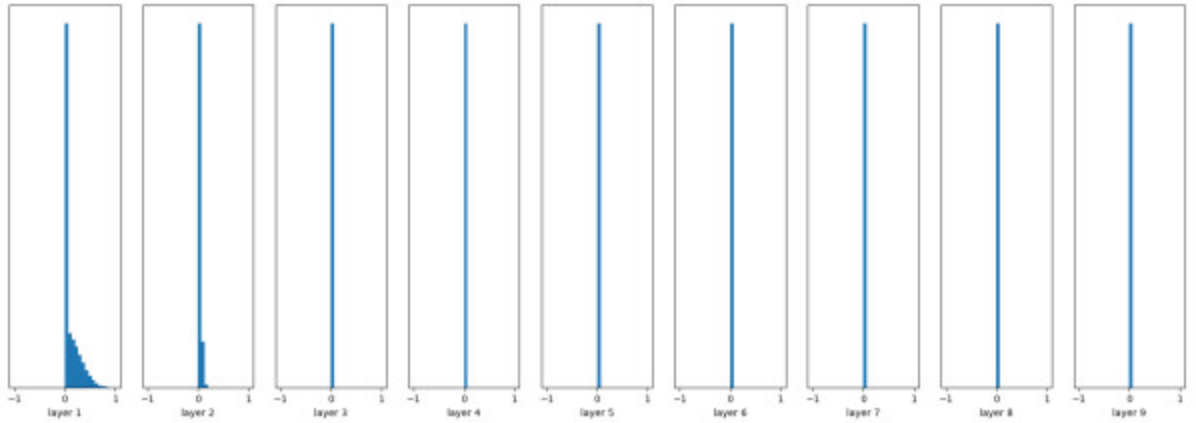
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Source: CS231n

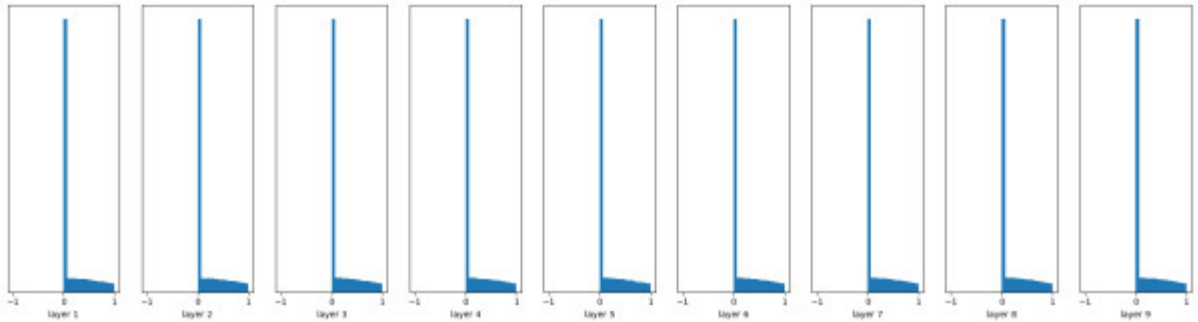
Random initialization without BN

```
W = tf.Variable(np.random.randn(node_in, node_out)) * 0.01
......
fc = tf.nn.relu(fc)
```



Random initialization with BN

```
W = tf.Variable(np.random.randn(node_in, node_out)) * 0.01
......
fc = tf.contrib.layers.batch_norm(fc, center=True, scale=True,
                                  is_training=True)
fc = tf.nn.relu(fc)
```



Source: CS231n

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- How to Get μ γ σ β

- Training time
  - Just as the algorithm, based on the minibatch
  - Mean and variance of minibatch is an approximate of whole batch
    - Good result if these two are matched
    - Required a certain size of mini-batch (better larger than 8 or 16, bad for 1 or 2)
- Test time
  - μ σ can be estimated with running averages during training time (fixed in inference)
  - γ will be merged with weight before inference

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Source: CS231n

TLDR

- Use batch normalization to reduce effect of initialization

Source: CS231n

BN results and effects

- BN -- before or after ReLU?

Name	Accuracy	LogLoss	Comments
Before	0.474	2.35	As in paper
Before + scale&bias layer	0.478	2.33	As in paper
After	**0.499**	**2.21**	
After + scale&bias layer	0.493	2.24	

Explanation of BatchNorm by Ian Goodfellow.
BN is better understood as a technique which reduces **second-order relationships between parameters of different layers** than a method to reduce covariate shift. Thus, the before/after distinction doesn't matter, and differences in performance could simply be because of other particular artefacts of the model.
Source: the deep learning book
https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md

- BN and activations

Name	Accuracy	LogLoss	Comments
ReLU	0.499	2.21	
RReLU	0.500	2.20	
PReLU	**0.503**	**2.19**	
ELU	0.498	2.23	
Maxout	0.487	2.28	
Sigmoid	0.475	2.35	
TanH	0.448	2.50	
No	0.384	2.96	

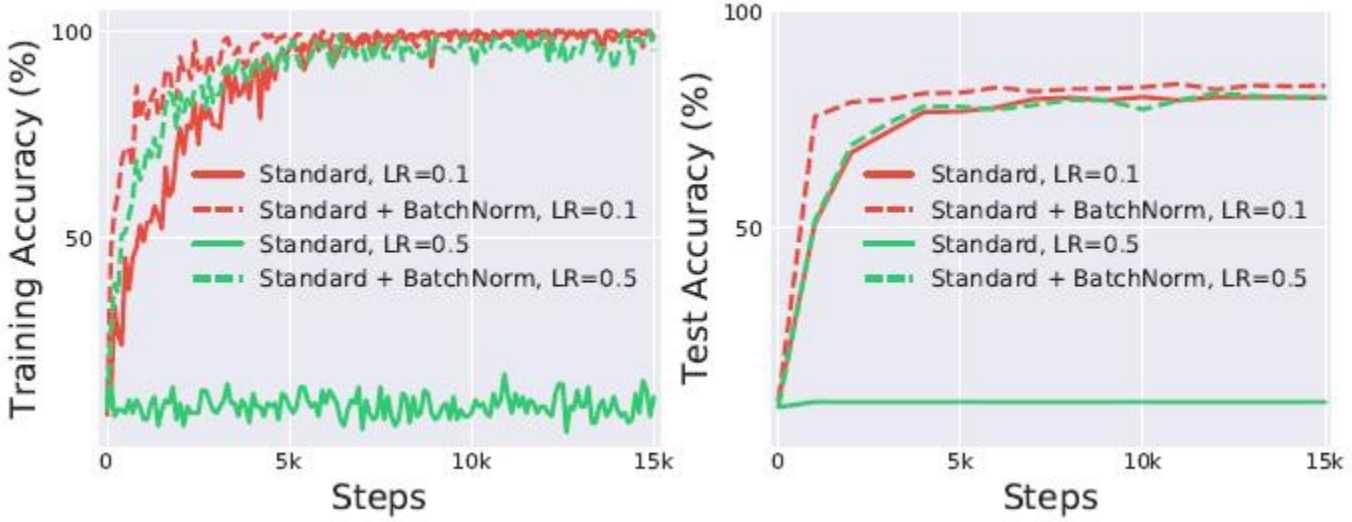
Name	Accuracy	LogLoss	Comme
GoogLeNet128	0.619**	1.61	
GoogLeNet BN Before + scale&bias layer LSUV	0.603	1.68	
GoogLeNet BN Before + scale&bias layer Ortho	0.607	1.67	
GoogLeNet BN After LSUV	0.596	1.70	
GoogLeNet BN After Ortho	0.584	1.77	
[GoogLeNet128_BN_lim0606] [https://github.com/lim0606/caffe-googlenet-bn]	**0.645**	**1.54**	BN before ReLU + scale bias, linear LR, batch_size = 128, base_lr = 0.005, 640K iter, LSUV init. 5x5 replaced with 3x3 + 3x3. 3x3 replaced with 3x1+1x3

ReLU non-linearity, fc6 and fc7 layer only

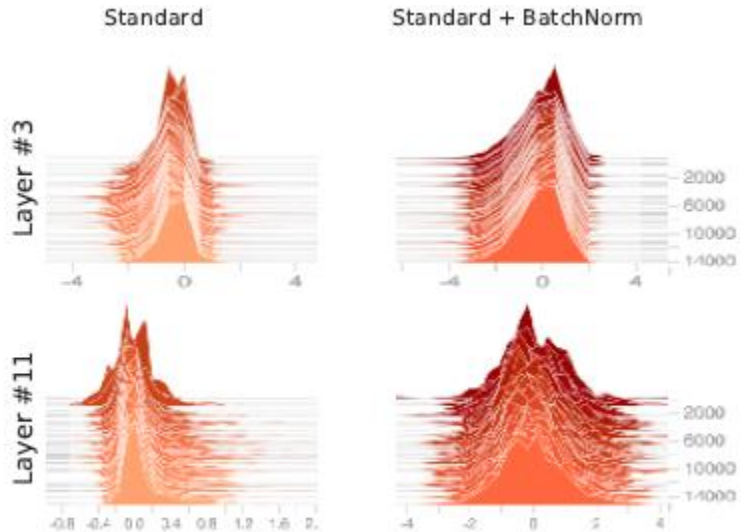
Name	Accuracy	LogLoss	Comments
Dropout = 0.5	0.499	2.21	
Dropout = 0.2	**0.527**	**2.09**	
Dropout = 0	0.513	2.19	

How Does Batch Normalization Help Optimization?

- Does BatchNorm's performance stem from controlling internal covariate shift?
  - Small differences in the layer input distribution between one with or without BN

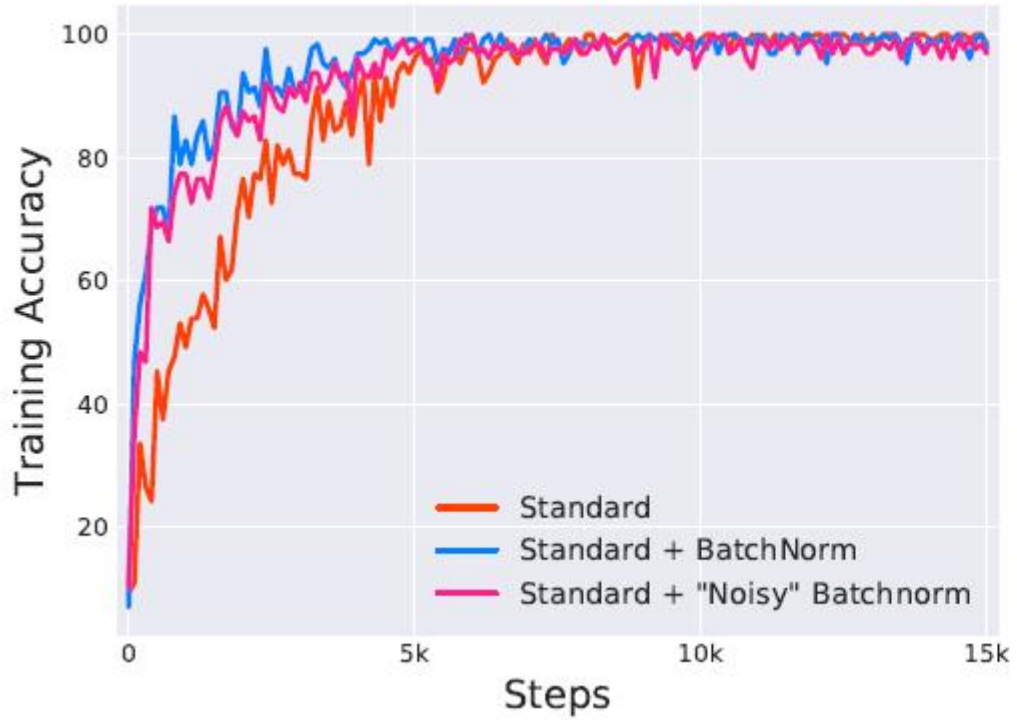


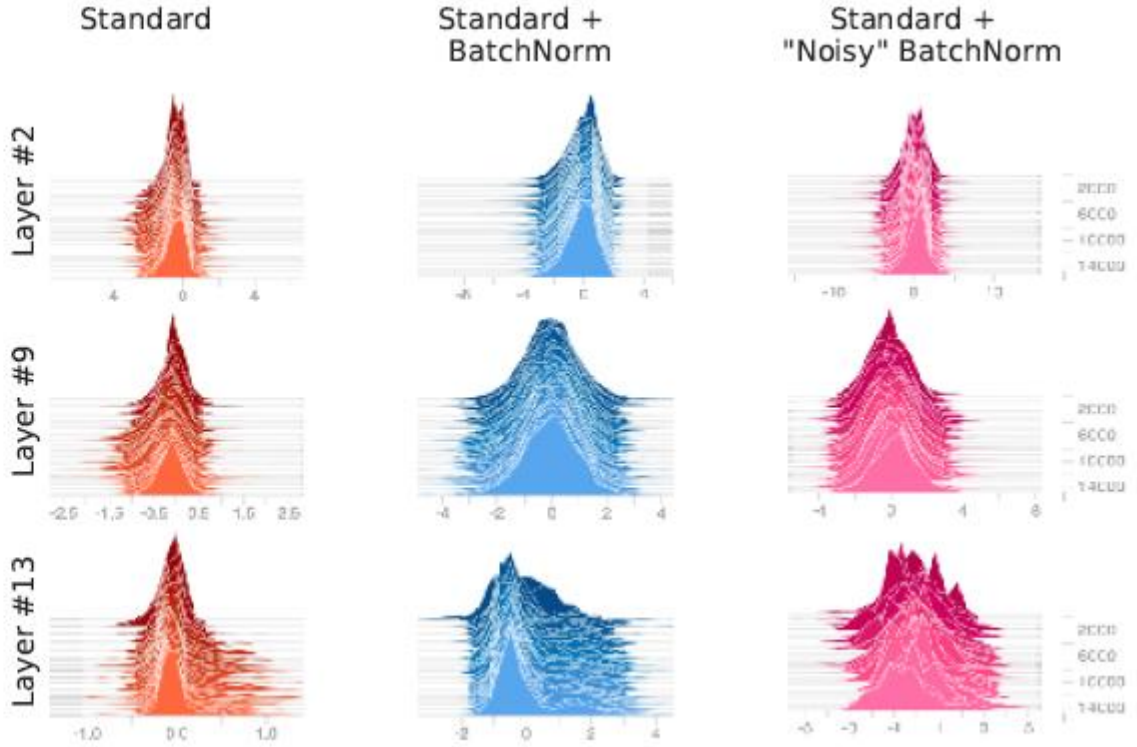
有BN，model initialization, learning rate choice 選擇較不敏感



difference in distributional stability (change in the mean and variance) in networks with and without BatchNorm layers seems to be marginal.

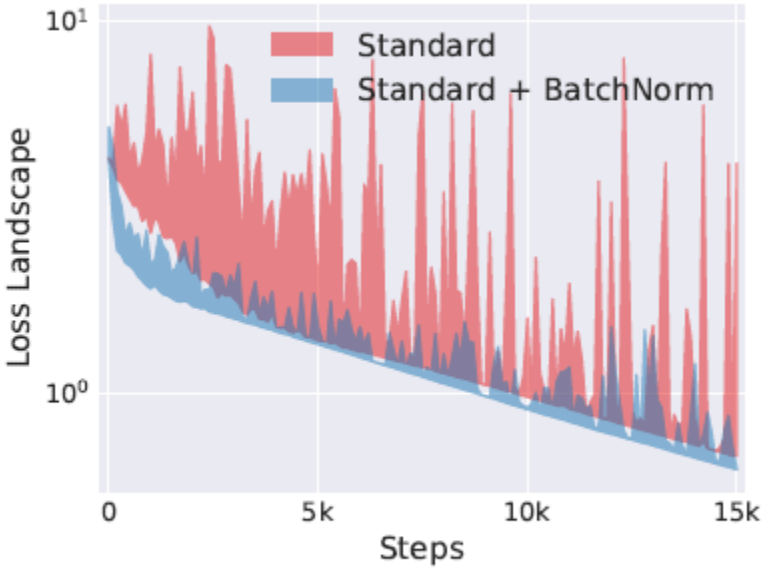
- Noisy BN has similar performance and input distribution
- BN performance does not relate to the stability of input distribution



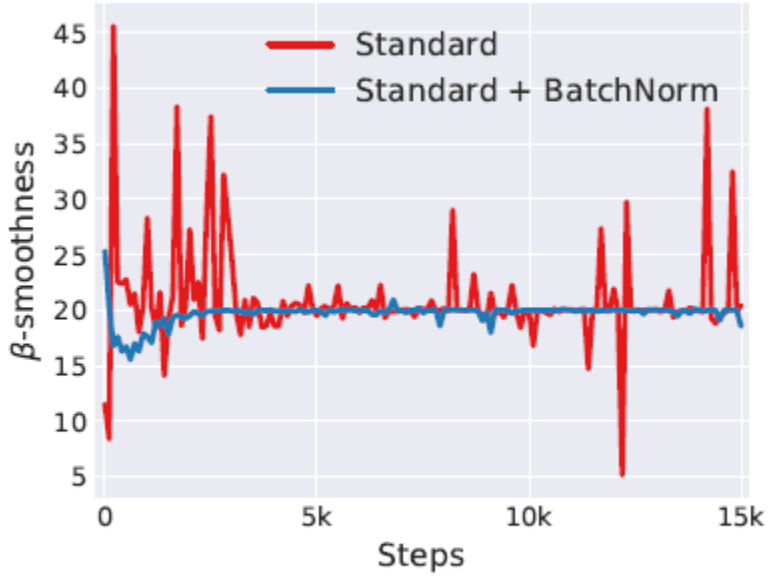


BN Smooth Gradient

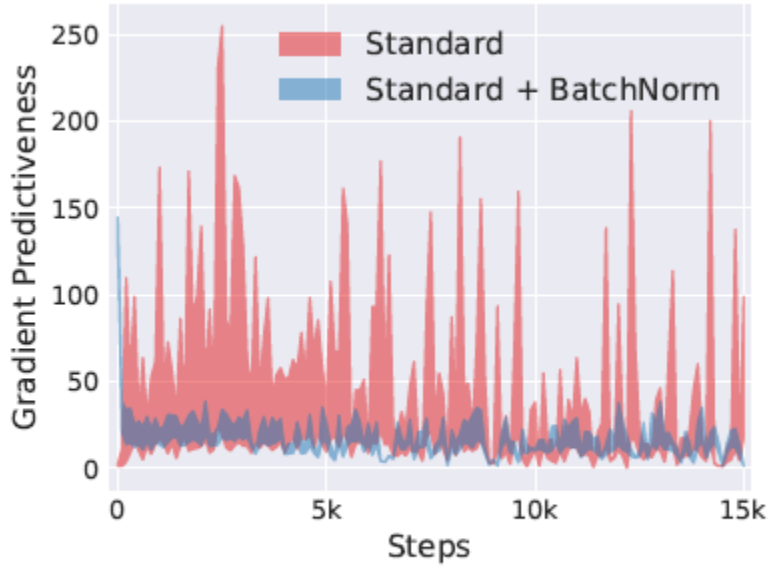
- BN makes loss landscape and gradient smoother
  - Gradient more reliable and predictable
  - Easy for any gradient descent algorithm to work well
  - Larger step size without trap to local min or kink



(a) loss landscape

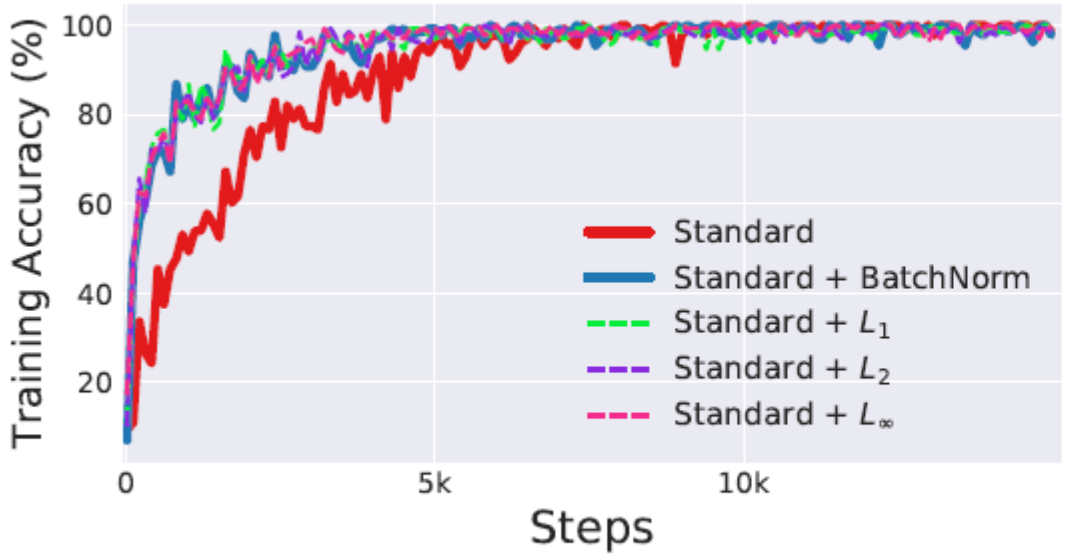


(b) "effective" β-smoothness

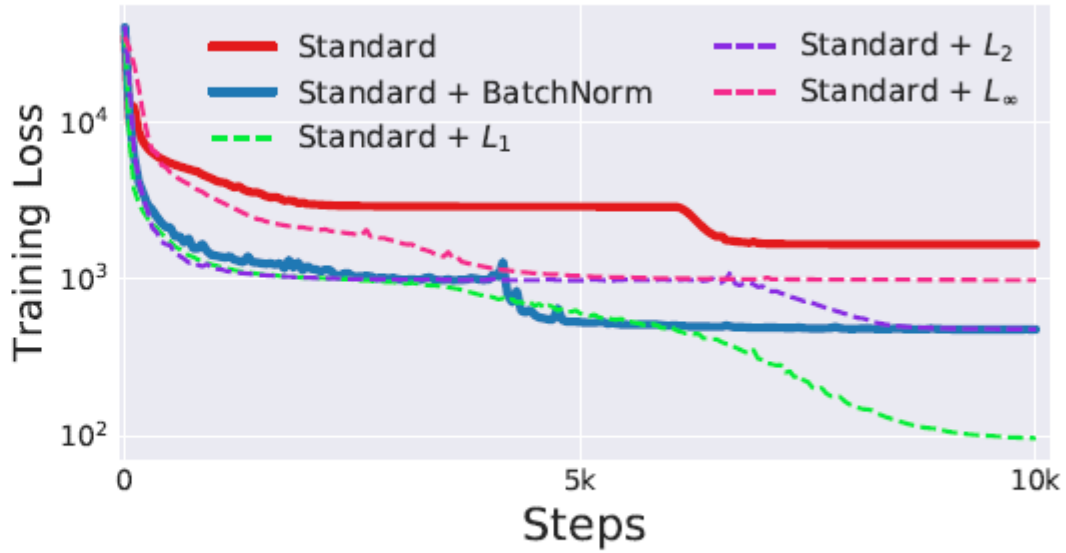


(c) gradient predictiveness

Other Normalization Works As Well



(a) VGG



(b) Deep Linear Model

L1-norm variance BN

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}} + \epsilon} \qquad \sigma_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} |x_i - \mu_{\mathcal{B}}|$$

$$\gamma_{L_2} = \sqrt{\frac{\pi}{2}} \cdot \gamma_{L_1} \qquad \sigma_{L_2} = \sqrt{\frac{\pi}{2}} \cdot \sigma_{L_1}$$

Beyond BN: Layer BN/Instance BN/Group BN

- 輸入的圖像shape記為[N, C, H, W] (N: batch, C: channel)
  - batchNorm是在batch上，對NHW做歸一化，對小batchsize效果不好；
  - layerNorm在通道方向上，對CHW歸一化，主要對RNN, transformer
  - instanceNorm在圖像像素上，對HW做歸一化，用在風格化遷移；
  - GroupNorm將channel分組，然後再做歸一化；
  - SwitchableNorm是將BN、LN、IN結合，賦予權重，讓網絡自己去學習歸一化層應該使用什麼方法

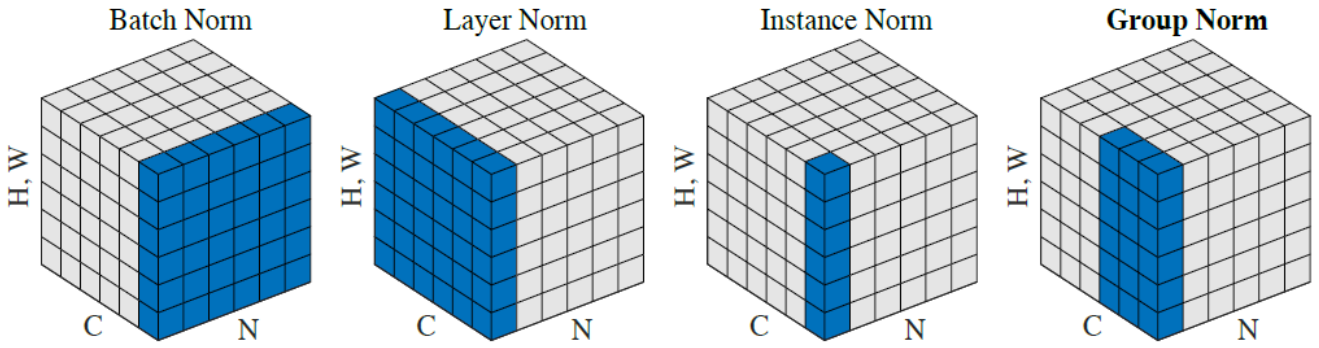
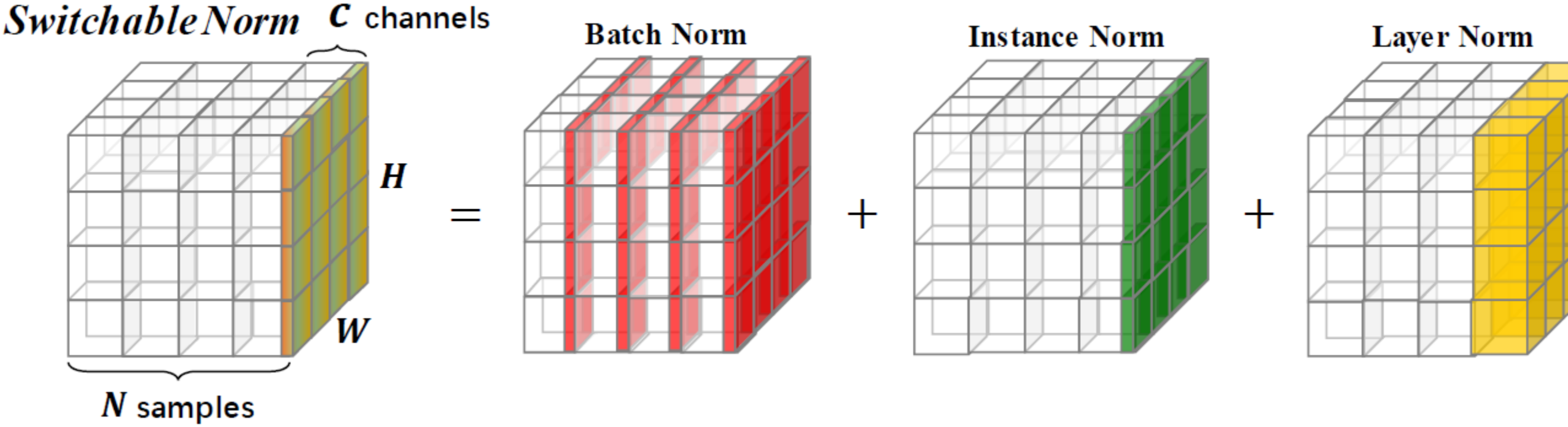


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

https://arxiv.org/pdf/1803.08494.pdf

Switchable-Normalization

- learn different normalization operations for different normalization layers in a deep neural network in an end-to-end manner
  - Up to 3% accuracy improvement



Layer Normalization

- LN: apply normalization to certain layer input

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2} \tag{3}$$

  - LN中同層神經元輸入擁有相同的均值和方差，不同的輸入樣本有不同的均值和方差
    - LN不依賴於mini-batch的大小和輸入sequence的深度，因此可以用於bath-size為1和RNN中對邊長的輸入sequence的normalize操作
  - BN中則針對不同神經元輸入計算均值和方差，同一個minibatch中的輸入擁有相同的均值和方差
  - For RNN, Normalization term會對summed input的進行尺度變換，使RNN在training和inference時更加穩定

$$\mathbf{h}^t = f\left[\frac{\mathbf{g}}{\sigma^t} \odot (\mathbf{a}^t - \mu^t) + \mathbf{b}\right] \qquad \mu^t = \frac{1}{H}\sum_{i=1}^{H} a_i^t \qquad \sigma^t = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^t - \mu^t)^2} \tag{4}$$

Instance Normalization

- Same principle
  - BN: on batch 統計量會受到batch中其他樣本的影響
  - IN: single image 由自己計算出的統計量
- Why IN, especially for style transfer
  - 通過調整BN統計量，或學習的參數beta和gamma，BN可以用來做domain adaptation
  - Style Transfer是一個把每張圖片當成一個domain的domain adaptation問題。
- BN or IN
  - 圖片視訊分類等特徵提取網絡中大多數情況BN效果優於IN，在生成式類任務中的網絡IN優於BN

Group Normalization

- BN: batch size dependent
  - Training work with large batch size (~32)
  - Testing without small batch size (1?)
  - Inconsistant statistics between training and testing ?
- IN and LN are special cases of GN

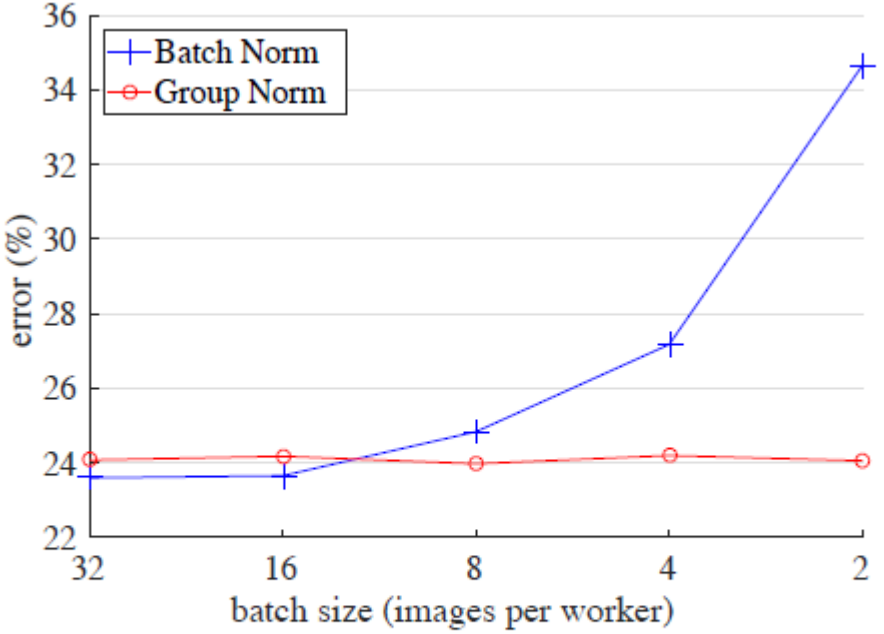
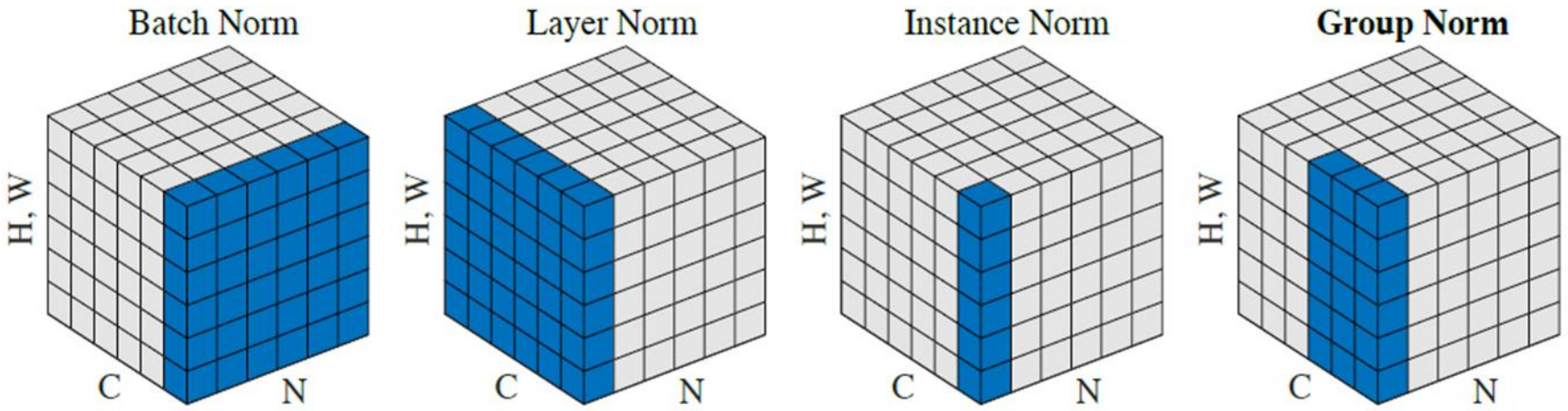


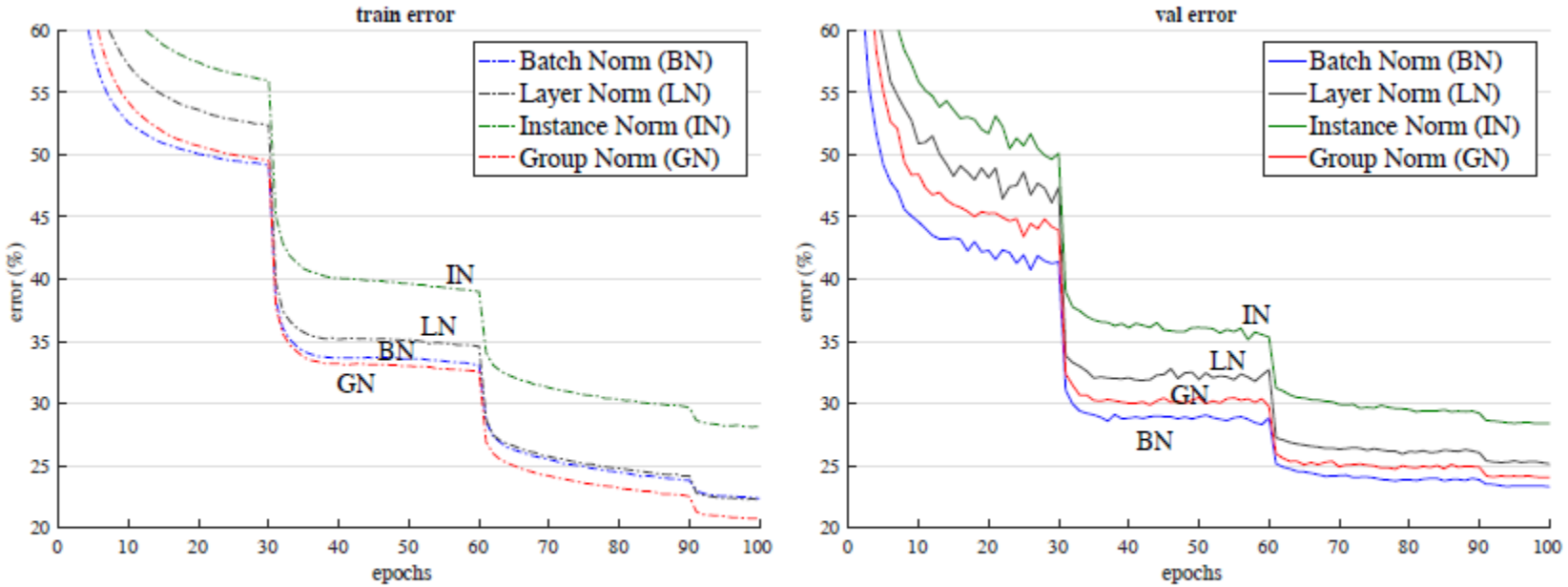
Figure 1. **ImageNet classification error *vs.* batch sizes.** This
a ResNet-50 model trained in the ImageNet training set using
workers (GPUs), evaluated in the validation set.

GN介於LN和IN之間，其首先將channel分為許多組（group），對每一組做歸一化



Why GN Works?

- Before DL
  - Group wise feature or group wise norm in SIFT, HOG, GIST
- DL
  - Feature is not totally independent
  - Same or similar features can be grouped together



- Robust to batch size

	BN	LN	IN	GN
val error	**23.6**	25.3	28.4	24.1
△ (*vs.* BN)	-	*1.7*	*4.8*	***0.5***

Table 1. **Comparison of error rates** (%) of ResNet-50 in the ImageNet validation set, trained with a batch size of **32 images**/GPU. The error curves are in Figure 4.

batch size	32	16	8	4	2
BN	**23.6**	**23.7**	24.8	27.3	34.7
GN	24.1	24.2	**24.0**	**24.2**	**24.1**
△	*0.5*	*0.5*	*-0.8*	*-3.1*	*-10.6*

Table 2. **Sensitivity to batch sizes**. We show ResNet-50's validation error (%) in ImageNet. The last row shows the differences between BN and GN. The error curves are in Figure 5. This table is visualized in Figure 1.

- Good for detection and segmentation

backbone	AP^{bbox}	AP^{bbox}_{50}	AP^{bbox}_{75}	AP^{mask}	AP^{mask}_{50}	AP^{mask}_{75}
BN*	37.7	57.9	40.9	32.8	54.3	34.7
GN	**38.8**	**59.2**	**42.2**	**33.6**	**55.9**	**35.4**

Table 4. Detection and segmentation **ablation results in COCO**, using Mask R-CNN with **ResNet-50 C4**. BN* means BN is frozen.

	AP^{bbox}	AP^{bbox}_{50}	AP^{bbox}_{75}	AP^{mask}	AP^{mask}_{50}	AP^{mask}_{75}
R50 BN*	38.6	59.8	42.1	34.5	56.4	36.3
R50 GN	40.3	61.0	44.0	35.7	57.9	37.7
R50 GN, long	**40.8**	**61.6**	**44.4**	**36.1**	**58.5**	**38.2**
R101 BN*	40.9	61.9	44.8	36.4	58.5	38.7
R101 GN	41.8	62.5	45.4	36.8	59.2	39.0
R101 GN, long	**42.3**	**62.8**	**46.2**	**37.2**	**59.7**	**39.5**

Table 6. **Detection and segmentation results in COCO** using

PRE-NORM VS POST-NORM

Pre-Activation BN vs post-Activation BN

- Post-Activation BN: Conv / Linear -> Activation (ReLU) -> BN
- **設計思路：**
  - 原始論文的作者認為，「內部協變量轉移」指的是網路中每一層**輸入**的分佈在訓練過程中不斷變化。對於下一層來說，前一層激活函數的輸出 a 就是它的輸入。因此，對 a 進行標準化是最直觀的想法。
- **潛在問題：**
  - BN 的目的是將數據分佈變得更穩定（例如，接近標準正態分佈），而激活函數（特別是 ReLU）本身就會極大地改變數據分佈。
  - 以 ReLU 為例，它會將所有小於 0 的值都變為 0。
  - BN 會對這些已經被「截斷」過的、分佈不對稱的數據進行標準化。
  - 這意味著 BN 和 ReLU 的目標可能存在衝突。BN 試圖將數據的均值移回 0，但 ReLU 已經移除了所有負值部分。這可能會限制模型的學習能力，因為 BN 處理的是一個已經損失了部分信息的數據分佈。

Pre-Activation BN vs post-Activation BN

- Pre-Activation BN: Conv / Linear -> BN -> Activation (ReLU)
  - 在 ResNet-v2 等研究中被證明更有效的方法，並成為了現代網路架構的標準做法
- **設計思路：**
  - 這種設計認為，BN 的主要好處應該是為**非線性激活函數**提供一個穩定、健康的輸入分佈。
  - 卷積層或全連接層的輸出 z 通常呈現一個更接近對稱（例如高斯）的分佈。
  - 在 z 進入 ReLU 之前對其進行標準化，可以確保 ReLU 總是在一個穩定的、均值為0、方差為1的分佈上進行操作。
  - 這樣做可以讓更多的神經元處於激活狀態，避免大量的神經元輸出為0（即 "Dying ReLU" 問題），從而使梯度流動更順暢。
- **主要優點：**
  - **更健康的梯度流**：BN 確保了數據不會輕易地進入 ReLU 的飽和區（負值區），有助於梯度傳播，這在非常深的網路（如 ResNet）中尤其重要。
  - **更強的的正則化效果**：研究表明，Pre-Activation BN 的模型對初始化更不敏感，且表現出更好的泛化能力。
  - **目標一致**：BN 和 ReLU 的目標不再衝突。BN 負責穩定數據分佈，ReLU 負責引入非線性，分工明確。

BN and Activation

- 為什麼BN有效?
  - 傳統觀點認為，它減少「內部共變數偏移」(Internal Covariate Shift, ICS)
  - 新觀點
    - 降低梯度的利普希茨常數 (Reducing the gradient Lipschitz constant)。
    - 降低隨機梯度平方的期望值 (Reducing the expectation of the squared stochastic gradient)。
    - 降低隨機梯度的變異數 (Reducing the variance of the stochastic gradient)。
  - 傳統BN只有在與ReLU激活函數配合使用時，才能間接引發這三種效應 。當使用Sigmoid、Tanh等其他激活函數時，其效果會大打折扣

Complete Batch Normalization (CBN)

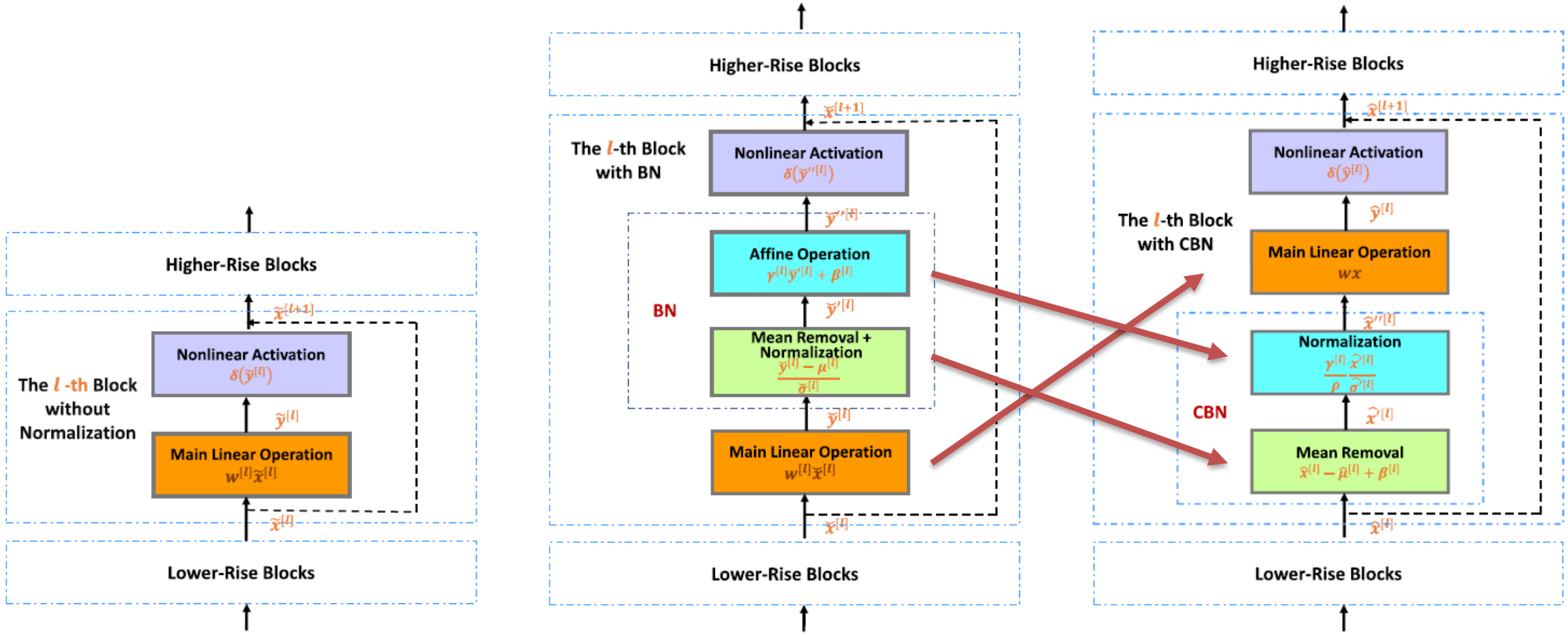


Fig. 1. Structures of a standard block embedded with (a) no normalization, (b) vanilla BN, and (c) CBN. The shortcut is optional. Vanilla BN is placed behind the linear activation layer, while CBN is located in the front of the linear activation layer. Moreover, mean removal and normalization with ℓ_2 norm are tangled in vanilla BN, but the two parts are decoupled in CBN. Additionally, CBN adds ρ to tune the magnitude of normalization for better performance.

- CBN與傳統BN相比，有四個關鍵的結構性修改 ：
  - **位置改變**：CBN將正規化層放置在線性操作 (如卷積層) 之前，而傳統BN則放置在線性操作之後、非線性激活函數之前 。論文從理論上證明，將正規化放在非線性激活函數之後 (即下一個區塊的線性操作之前) 是強制性的，並且更有益 。
  - **解構操作**：CBN將傳統BN中混合在一起的「均值移除」(Mean Removal) 和「l_2 範數正規化」(Normalization with l_2 norm) 兩個步驟解耦，使其成為獨立的步驟 。
  - **引入超參數 rho**：CBN加入了一個新的超參數 rho 來調整正規化的強度，這有助於更好地降低梯度利普希茨常數和梯度平方期望值 。
  - **採用自適應移動平均統計 (AMAT)**：為了降低隨機梯度的變異數，CBN在訓練期間使用基於歷史資訊的自適應移動平均統計數據，而非僅僅依賴當前批次的數據

TABLE III
COMPARISON OF MEAN TEST ACCURACY AND STANDARD DEVIATION OF 10 TRIALS FOR BN AND CBN EMBEDDED RESNET-20 WITH DIFFERENT ACTIVATION FUNCTIONS ON CIFAR100

Methods	Before Opera.	Decouple	Tune ρ	AMAT	ResNet-20 Sigmoid	Tanh	ReLU	SELU	Swish
① BN (Baseline)	-	-	-	-	46.05 $\pm$ 0.20	63.15 $\pm$ 0.09	67.21 $\pm$ 0.12	63.81 $\pm$ 0.14	67.82 $\pm$ 0.10
② CBN Improv.	✓	✓	✓	✓	66.92 $\pm$ 0.08 (+20.87)	67.13 $\pm$ 0.07 (+3.98)	68.55 $\pm$ 0.05 (+1.34)	67.43 $\pm$ 0.08 (+3.62)	69.52 $\pm$ 0.16 (+1.70)
③	✓	-	-	-	65.06 $\pm$ 0.09 (+19.01)	66.42 $\pm$ 0.11 (+3.09)	66.78 $\pm$ 0.12 (-0.43)	66.13 $\pm$ 0.09 (+2.32)	67.25 $\pm$ 0.12 (-0.57)
④	✓	✓	-	-	66.64 $\pm$ 0.08 (+20.59)	66.24 $\pm$ 0.10 (+3.27)	67.90 $\pm$ 0.08 (+0.69)	66.22 $\pm$ 0.16 (+2.41)	68.31 $\pm$ 0.13 (+0.49)
⑤	✓	✓	✓	-	66.82 $\pm$ 0.11 (+20.77)	67.13 $\pm$ 0.15 (+3.98)	68.28 $\pm$ 0.09 (+1.07)	66.40 $\pm$ 0.09 (+2.59)	69.17 $\pm$ 0.15 (+1.35)
⑥	-	-	✓	-	48.82 $\pm$ 0.24 (+2.77)	63.26 $\pm$ 0.14 (+0.11)	67.38 $\pm$ 0.11 (+0.17)	65.03 $\pm$ 0.14 (+1.22)	67.93 $\pm$ 0.10 (+0.11)
⑦	-	-	✓	✓	49.88 $\pm$ 0.15 (+3.83)	63.45 $\pm$ 0.10 (+0.40)	67.51 $\pm$ 0.09 (+0.30)	65.25 $\pm$ 0.08 (+1.44)	68.01 $\pm$ 0.11 (+0.19)

Pre-LN vs Post-LN

- Post-LN
  - Used in original transformer model (original BERT and GPT)
  - Work well when training is stable and not very deep
- Pre-LN
  - Applied before residual connection
  - Stability and Training Depth
  - Support deeper network due to help in mitigating the vanishing gradient problem
  - Less learning rate/initialization sensitivity
    - Allow aggressive training schedules
  - Used in GPT-3 and later

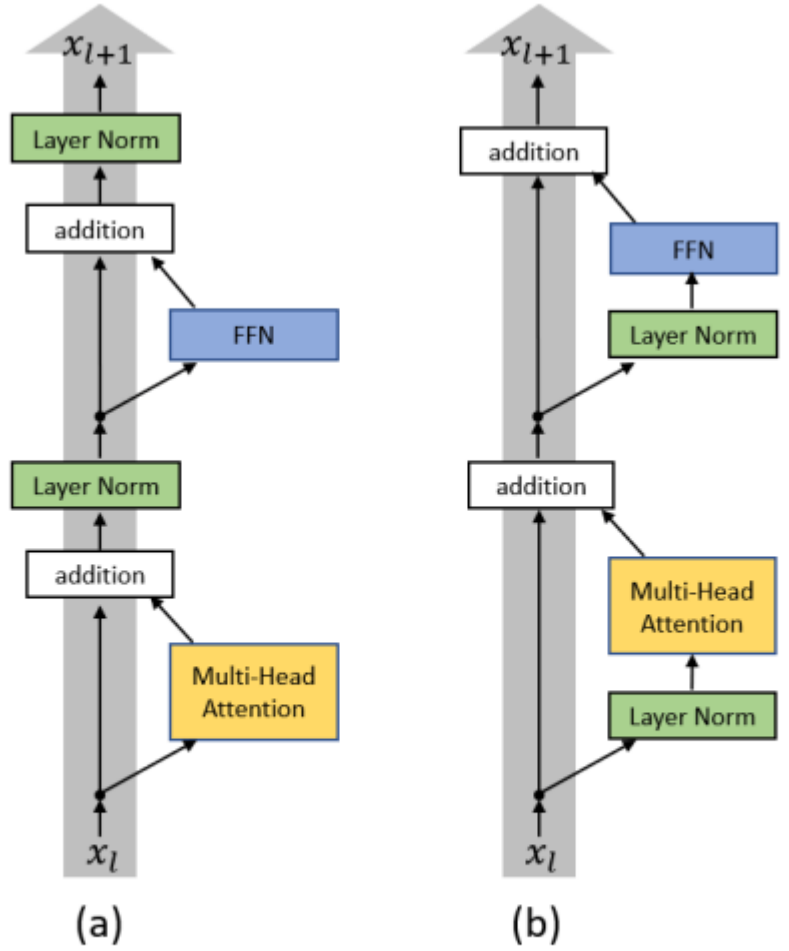


Figure 1. (a) Post-LN Transformer layer; (b) Pre-LN Transforme layer.

https://arxiv.org/abs/2002.04745

Post-LN 在深層模型中，梯度可能在殘差加法後放大

為什麼會「放大」？

- **殘差加法的作用**：殘差連接本意是讓梯度穩定傳遞，但如果 $f(x)$的梯度貢獻很大（例如，權重初始化不當或層數深時，子層的輸出變異數增加），則在加法後 z的梯度會被放大。想像一下：如果捷徑梯度是 1，子層梯度是 1.5，總和就變成 2.5，傳到上一層時更大。
- **LN 的位置問題**：在 Post-LN 中，LN 是放在加法之後，所以它只能正規化最終輸出 o，但無法防止加法過程中的梯度放大。結果是，梯度在層間傳遞時，可能在深層累積變大（尤其是從模型頂層向下傳時）。
- **深層模型的放大效應**：在淺層模型（層數少，如 6-12 層）中，這問題不明顯，因為累積層數少。但在深層模型（如 100+ 層的 LLM）中，梯度要穿過許多層，每層的殘差加法都可能輕微放大，導致指數級增長。這就像複利效應：每層放大 1.1 倍，10 層後是 2.59 倍，100 層後可能爆炸到天文數字。

- 何時用誰？
  - NLP/Transformer：LN / RMSNorm 為主（BN 對序列長度/批量敏感）。
  - CV/object detecion：大 batch → BN；小 batch → GN/SyncBN。
  - Norm-free：需更嚴謹初始化與學習率控制（進階主題）。

Normalization Free

- Approach
  - 透過合適的初始化與殘差結構設計，使得不需要 normalization 層，也能保持梯度不爆炸/不消失。
- 典型做法：
  - 使用合適的 weight initialization（例如根據深度調整縮放因子）。
  - 控制殘差連接的 scale，使 forward/backward 的 variance 穩定。
  - 用 deterministic scaling (像 $1/\sqrt{L}$ for L 層) 來避免累積放大。
  - Example
    - Fixup Initialization (2019) 就是一個 Norm-free 方法

HYPERPARAMETER OPTIMIZATION

Cross-validation strategy

- **coarse -> fine** cross-validation in stages
- **First stage**: only a few epochs to get rough idea of what params work
- **Second stage**: longer running time, finer search
- … (repeat as necessary)

- Tip for detecting explosions in the solver:
  - If the cost is ever > 3 * original cost, break out early

Source: CS231n

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                              model, two_layer_net,
                                              num_epochs=5, reg=reg,
                                              update='momentum', learning_rate_decay=0.9,
                                              sample_batches = True, batch_size = 100,
                                              learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice

Source: CS231n

Now run finer search

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

But this best cross-validation result is worrying. Why?

Source: CS231n

Hyperparameters to play with

- network architecture => capacity
- learning rate, its decay schedule, update type
  - The most important hyperparameter
- regularization (L2/Dropout strength)

- Two ways of search based on resources
  - One model with serial search for CPU based resource
  - Parallel models for abundant resource

Source: CS231n

Grid Search v.s. Random Search

- In practice, random search is recommeded

Random Search for
Hyper-Parameter Optimi
Bergstra and Bengio, 20

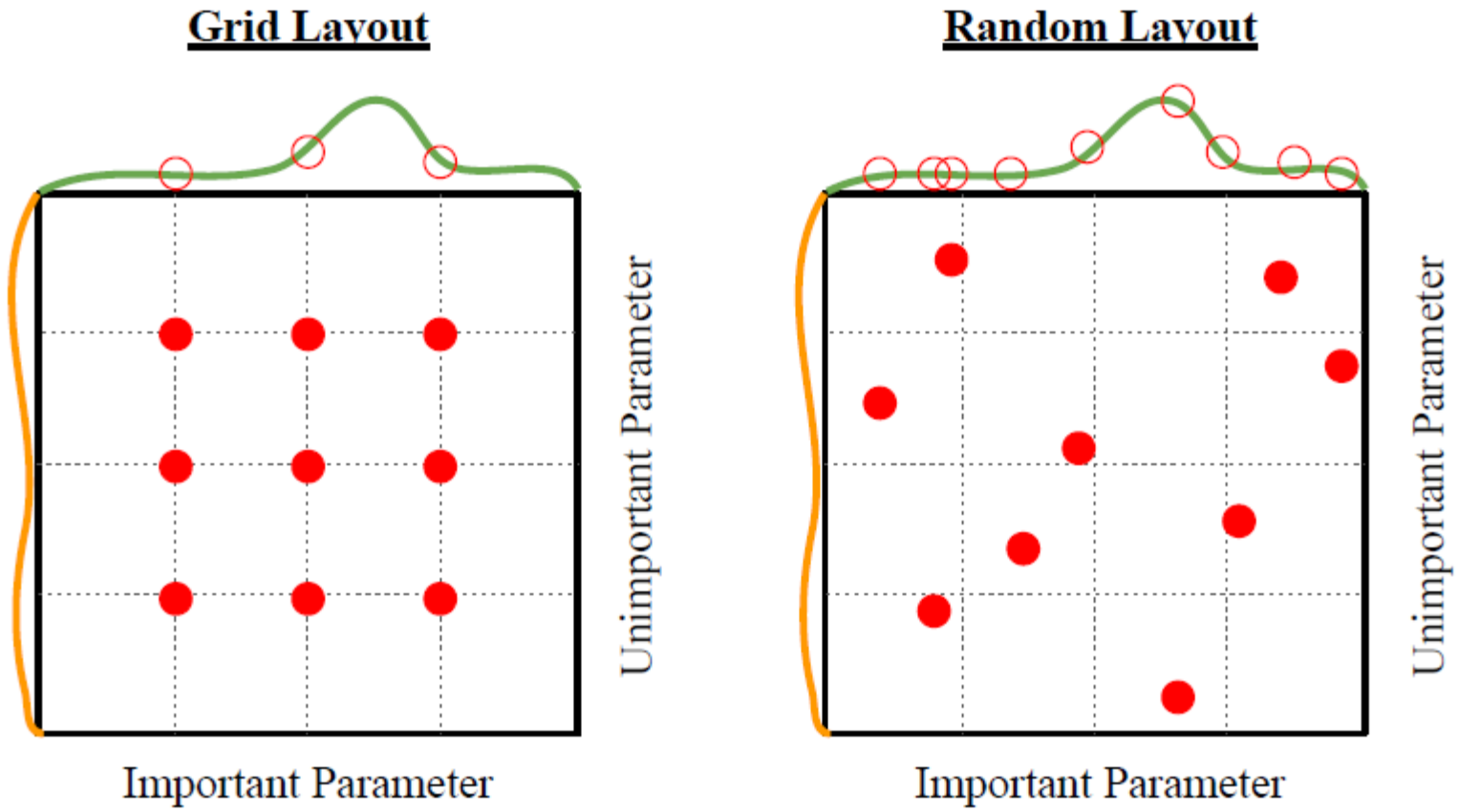
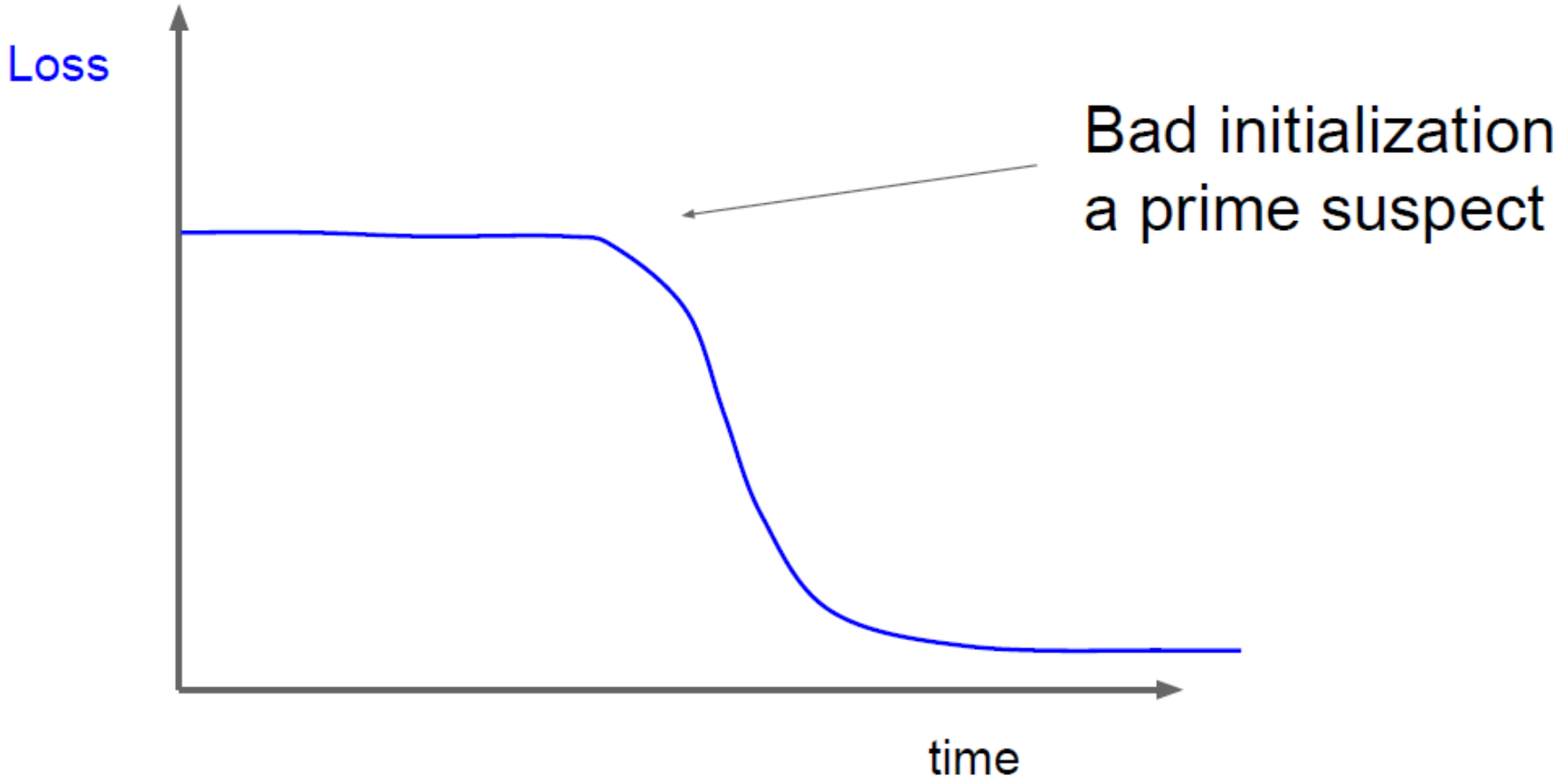
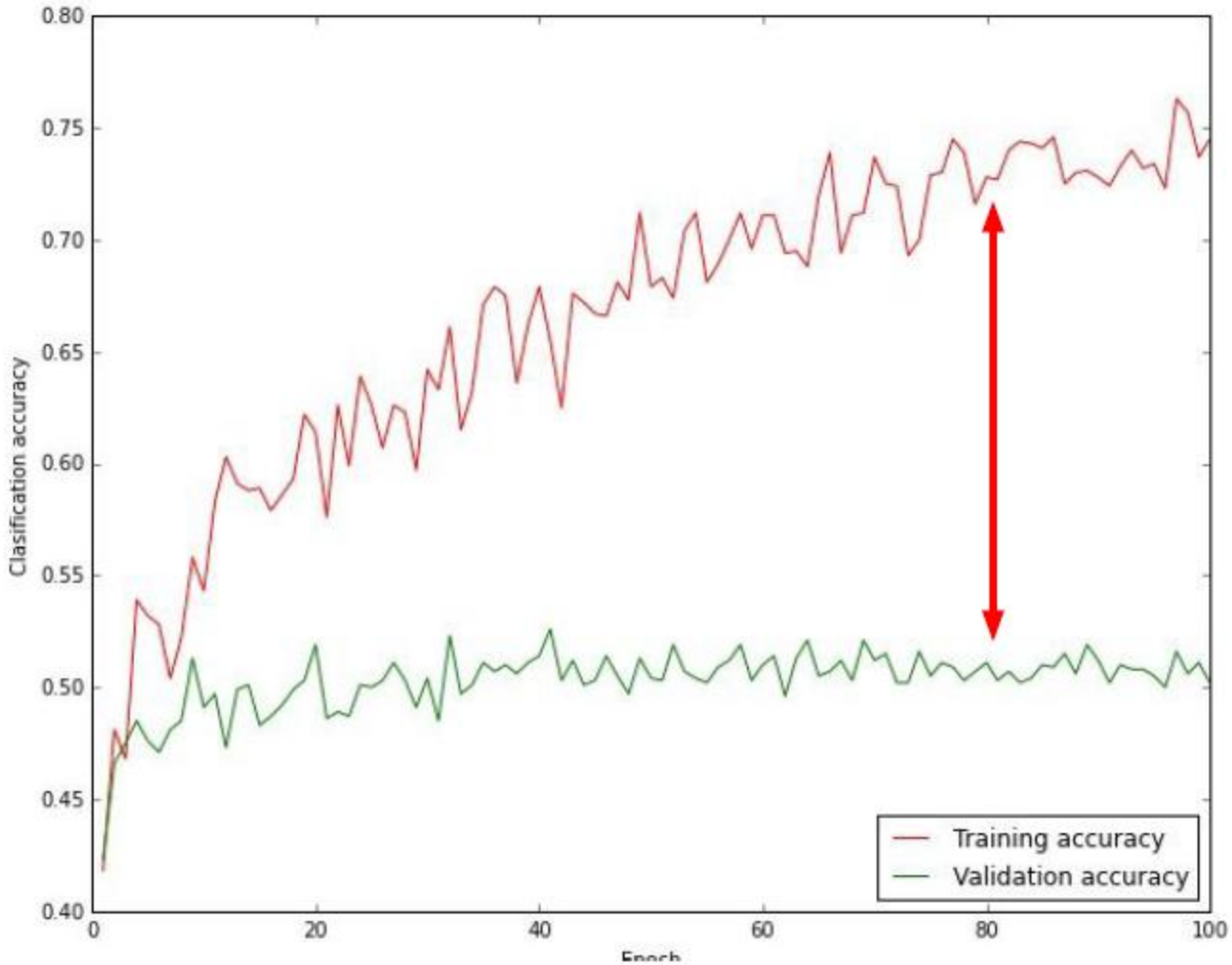


Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017



Source: CS231n



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

Source: CS231n

Summary

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization
  - (random sample hyperparams, in log space when appropriate)

Source: CS231n

Appendix

BAG OF TRICKS FOR IMAGE CLASSIFICATION WITH CONVOLUTIONAL NEURAL NETWORKS

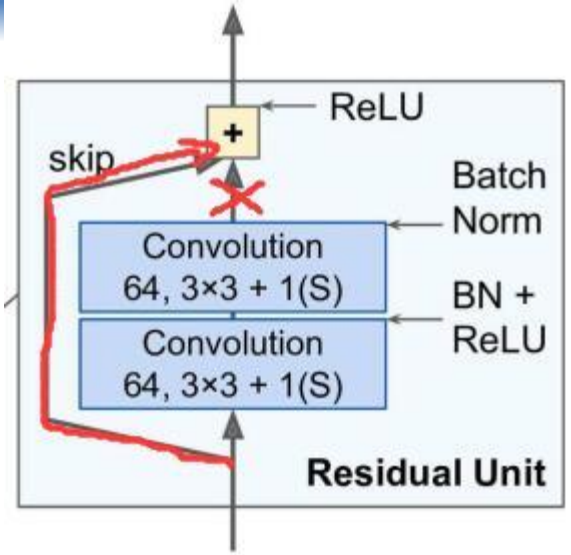
Tricks

- model architecture
  - input stem and downsample
- data augmentation
  - mixup
- loss function
  - label smoothing
- learning rate schedule
  - cosine learning rate decay、lr warmup
- Optimization
  - lr warmup、zero γ、no bias decay、cosine decay

資料增強 data augmentation (baseline)

- 訓練
  - 隨機抽樣並將uint8轉換為float32
  - 隨機裁剪出比例尺在[34,43][34,43]之間、尺寸大小在[8%,100%][8%,100%]之間的圖像塊，並resize為224x224的圖像
  - 以0.5的概率水準翻轉
  - 隨機使用[0.6,1.4][0.6,1.4]的係數對亮度、飽和度、對比度進行擾動
  - 用正態分佈N(0,0.1)N(0,0.1)的隨機係數為圖像添加PCA雜訊
  - 圖像數值去均值，除以方差
- 預測
  - 保持比例尺，將短邊縮放到256
  - 從中央裁剪出224x224的圖像
  - 與訓練減去相同的均值，除以相同的方差

Efficient Training



- Large-batch training
  - Linear scaling learning rate
  - Learning rate warmup
  - Zero γ for second BN of residual net or short cut for easy training at initial
  - No bias decay
- Low-precision training

Heuristic	BS=256		BS=1024	
	Top-1	Top-5	Top-1	Top-5
Linear scaling	75.87	92.70	75.17	92.54
+ LR warmup	76.03	92.81	75.93	92.84
+ Zero γ	76.19	93.03	76.37	92.96
+ No bias decay	76.16	92.97	76.03	92.86
+ FP16	76.15	93.09	76.21	92.97

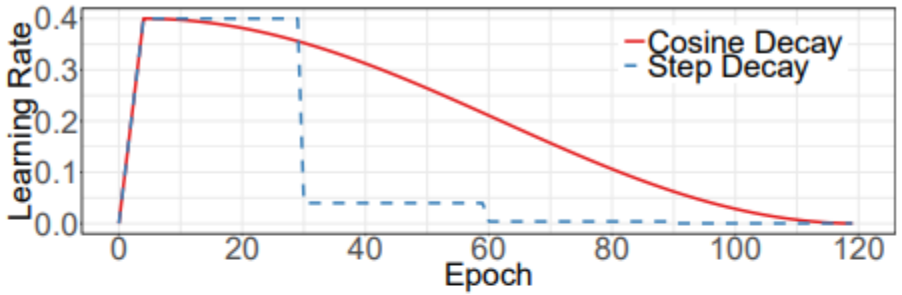
Model	Efficient			Baseline		
	Time/epoch	Top-1	Top-5	Time/epoch	Top-1	Top-5
ResNet-50	**4.4 min**	**76.21**	**92.97**	13.3 min	75.87	92.70
Inception-V3	**8 min**	**77.50**	**93.60**	19.8 min	77.32	93.43
MobileNet	**3.7 min**	**71.90**	**90.47**	6.2 min	69.03	88.71

Table 3: Comparison of the training time and validation accuracy for ResNet-50 between the baseline (BS=256 with FP32 and a more hardware efficient setting (BS=1024 with FP16).

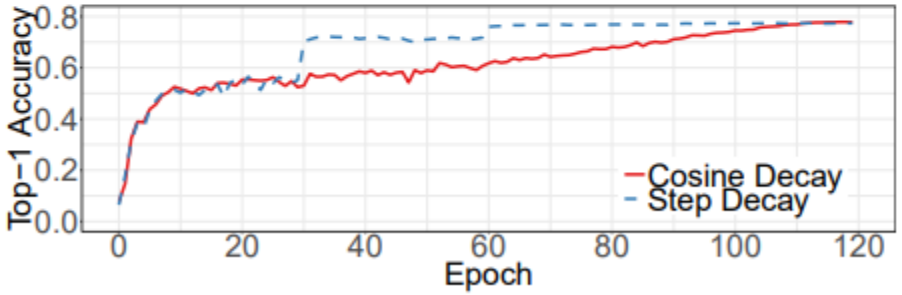
Training Refinements

- Cosine Learning Rate Decay
  - decreases the learning rate slowly at the beginning, and then becomes almost linear decreasing in the middle, and slows down again at the end
- Knowledge Distillation
- Mixup Training
  - Data augmentation with weighted linear interpolation of two examples
  - Only use the new example in mix-up

$$\hat{x} = \lambda x_i + (1-\lambda)x_j,$$
$$\hat{y} = \lambda y_i + (1-\lambda)y_j,$$



(a) Learning Rate Schedule



(b) Validation Accuracy

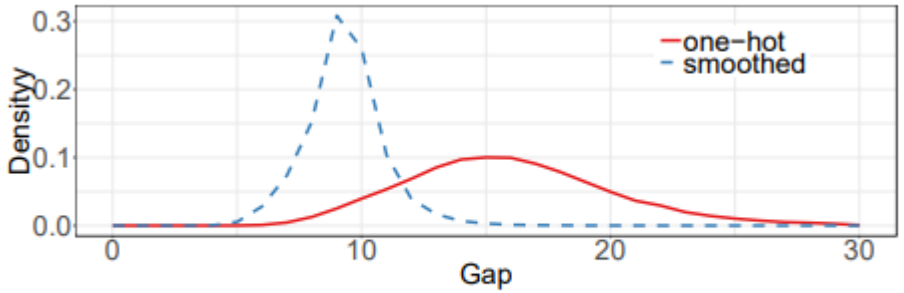
Training Refinements

- Label Smoothing
  - Change true probability to

$$q_i = \begin{cases} 1-\varepsilon & \text{if } i = y, \\ \varepsilon/(K-1) & \text{otherwise,} \end{cases}$$

    - K = 1000 for imagenet
    - ε = 0.1
  - Optimal solution becomes

$$z_i^* = \begin{cases} \log((K-1)(1-\varepsilon)/\varepsilon) + \alpha & \text{if } i = y, \\ \alpha & \text{otherwise,} \end{cases}$$



(b) Empirical gap from ImageNet validation set

The empirical distributions of the gap between the maximum prediction and the average of the rest.

Refinements	ResNet-50-D		Inception-V3		MobileNet	
	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
Efficient	77.16	93.52	77.50	93.60	71.90	90.53
+ cosine decay	77.91	93.81	78.19	94.06	72.83	91.00
+ label smoothing	78.31	94.09	78.40	94.13	72.93	91.14
+ distill w/o mixup	78.67	94.36	78.26	94.01	71.97	90.89
+ mixup w/o distill	79.15	94.58	**78.77**	**94.39**	**73.28**	**91.30**
+ distill w/ mixup	**79.29**	**94.63**	78.34	94.16	72.51	91.02

Table 6: The validation accuracies on ImageNet for stacking training refinements one by one. The baseline models are obtained from Section 3.

Model	FLOPs	top-1	top-5
ResNet-50 [9]	3.9 G	75.3	92.2
ResNeXt-50 [27]	4.2 G	77.8	-
SE-ResNet-50 [12]	3.9 G	76.71	93.38
SE-ResNeXt-50 [12]	4.3 G	78.90	94.51
DenseNet-201 [13]	4.3 G	77.42	93.66
ResNet-50 + tricks (ours)	4.3 G	**79.29**	**94.63**

Table 1: **Computational costs and validation accuracy of various models.** ResNet, trained with our "tricks", is able to outperform newer and improved architectures trained with standard pipeline.

Object Detection and Semantic Segmentation

Refinement	Top-1	mAP
B-standard	76.14	77.54
D-efficient	77.16	78.30
+ cosine	77.91	79.23
+ smooth	78.34	80.71
+ distill w/o mixup	78.67	80.96
+ mixup w/o distill	79.16	81.10
+ distill w/ mixup	79.29	**81.33**

Table 8: Faster-RCNN performance with various pre trained base networks evaluated on Pascal VOC.

Refinement	Top-1	PixAcc	mIoU
B-standard	76.14	78.08	37.05
D-efficient	77.16	78.88	38.88
+ cosine	77.91	**79.25**	**39.33**
+ smooth	78.34	78.64	38.75
+ distill w/o mixup	78.67	78.97	38.90
+ mixup w/o distill	79.16	78.47	37.99
+ mixup w/ distill	79.29	78.72	38.40

Table 9: FCN performance with various base networks evaluated on ADE20K.

trick	referenced paper
xavier init	Understanding the difficulty of training deep feedforward neural networks
warmup training	Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour
no bias decay	Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes
label smoothing	Rethinking the inception architecture for computer vision)
random erasing	Random Erasing Data Augmentation
cutout	Improved Regularization of Convolutional Neural Networks with Cutout
linear scaling learning rate	Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour
cosine learning rate decay	SGDR: Stochastic Gradient Descent with Warm Restarts

trick	acc
baseline	64.60%
+xavier init and warmup training	66.07%
+no bias decay	70.14%
+label smoothing	71.20%
+random erasing	does not work, drops about 4 points
+linear scaling learning rate(batchsize 256, lr 0.04)	71.21%
+cutout	does not work, drops about 1 point
+cosine learning rate decay	does not work, drops about 1 point

baseline(training from sctrach, no ImageNet pretrain weights are used):
vgg16 64.60% on CUB_200_2011 dataset, lr=0.01, batchsize=64
effects of stacking tricks

Linear Scaling Rule: *When the minibatch size is multiplied by k, multiply the learning rate by k.*

$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in \mathcal{B}} \nabla l(x, w_t).$$

$$w_{t+k} = w_t - \eta \frac{1}{n} \sum_{j<k} \sum_{x \in \mathcal{B}_j} \nabla l(x, w_{t+j})$$

$$\hat{w}_{t+1} = w_t - \hat{\eta} \frac{1}{kn} \sum_{j<k} \sum_{x \in \mathcal{B}_j} \nabla l(x, w_t).$$

setting ηˆ = kη would yield wˆt+1 ≈ wt+k, and the updates from small and large minibatch SGD would be similar

Test Time Augmentation





```
[[0.277 0.135 0.026 0.011 0.005 0.002 0.005 0.009 0.498 0.032]]
```

Prediction 1

```
[[0.085 0.411 0.002 0.001 0.    0.    0.002 0.    0.452 0.046]]
```

Prediction 2

```
[[0.252 0.411 0.009 0.003 0.002 0.    0.003 0.002 0.257 0.061]]
```

Prediction 3

```
[[0.11  0.182 0.004 0.001 0.    0.    0.002 0.    0.677 0.023]]
```

Prediction 4

```
[[0.225 0.361 0.02  0.01  0.004 0.002 0.006 0.005 0.312 0.055]]
```

Prediction 5

```
[[0.19  0.3   0.012 0.005 0.002 0.001 0.004 0.003 0.439 0.044]]
```

Average of the 5 predictions