17. Training your neural net - 2

EEC454 Introduction to Machine Learning Systems

2023 Fall, Jaeho Lee

Contents

- **Part 1.** Setting up training
    - Activation functions
    - Data pre-processing
    - Batch normalization
    - Weight initialization

- **Part 2.** Training Dynamics
    - Learning rate
    - Regularization
    - Babysitting the learning process
    - Hyperparameter optimization

Learning Rate

Recall that...

- **SGD.** Can be written as

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla_\theta \left(\sum_{i=1}^{B} \ell(y_i, f_\theta(\mathbf{x}_i)) \right)$$

- **Variants.** Adam, Adagrad, RMSProp ... are all based on this.
- **Hyperparameters.** There are two key HPs.
    - Learning rate η
    - Batch size B

Common Practice

- **Question.** How should we select η^*, B^*?

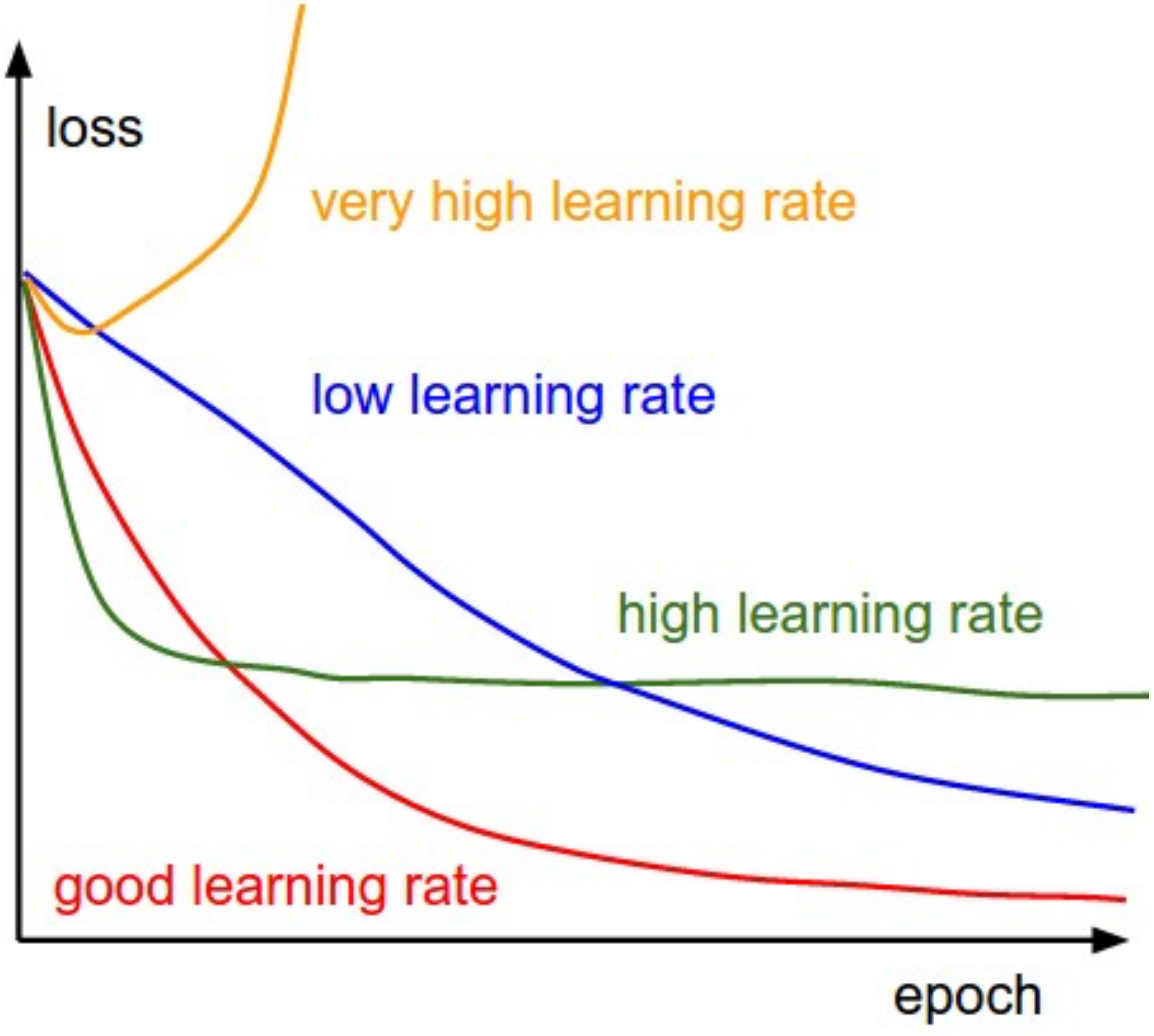
- **Practice.** Choose the largest possible B, then find the optimal η.

  Memory constraints
  + generalization issues

Typical case

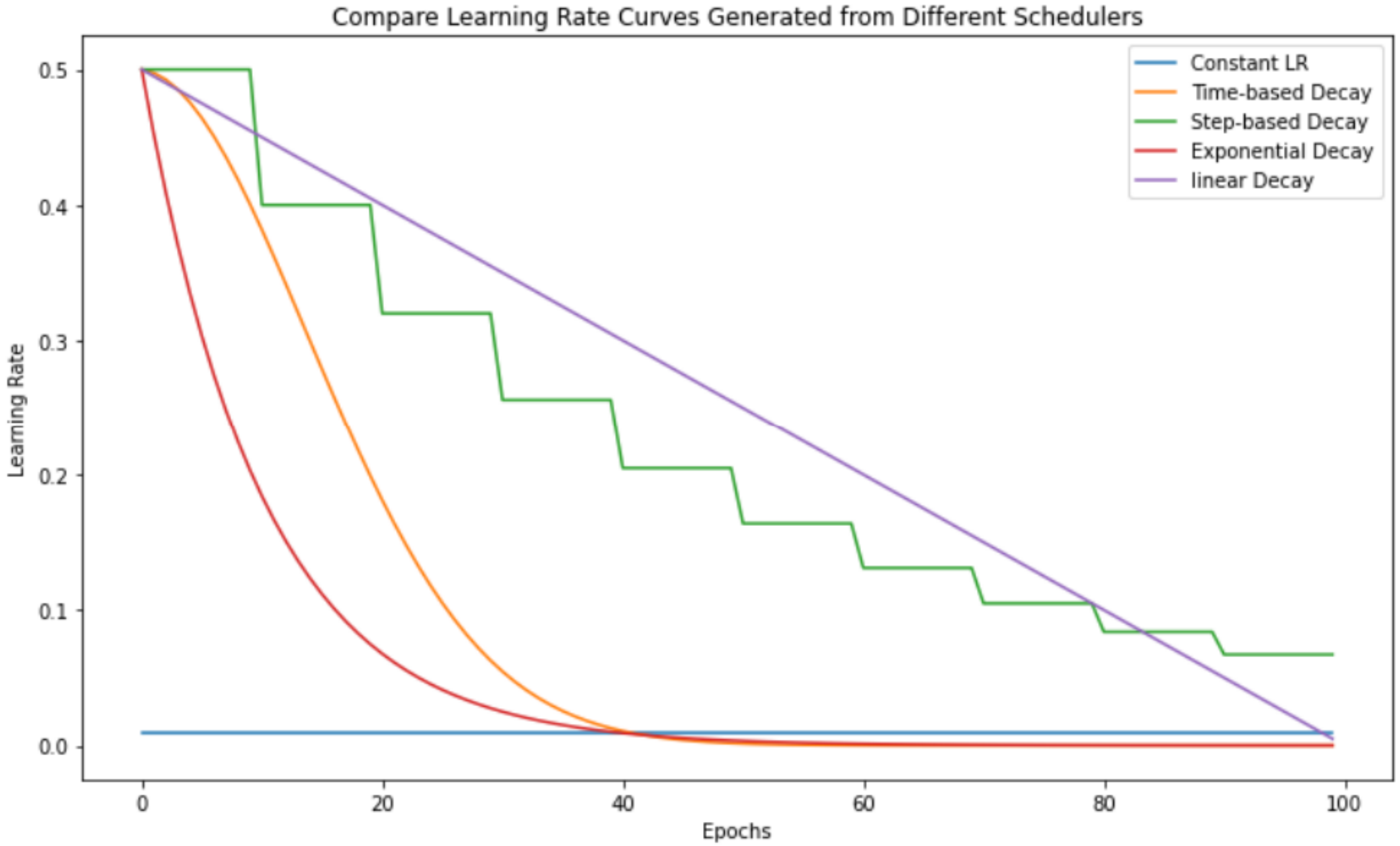
- **High LR**
  - Faster loss drop 👍
  - Converge at high loss 😢
- **Low LR**
  - Slow loss drop 😢
  - Converge at low loss 👍

- **Q.** How to enjoy both benefits?



LR decay

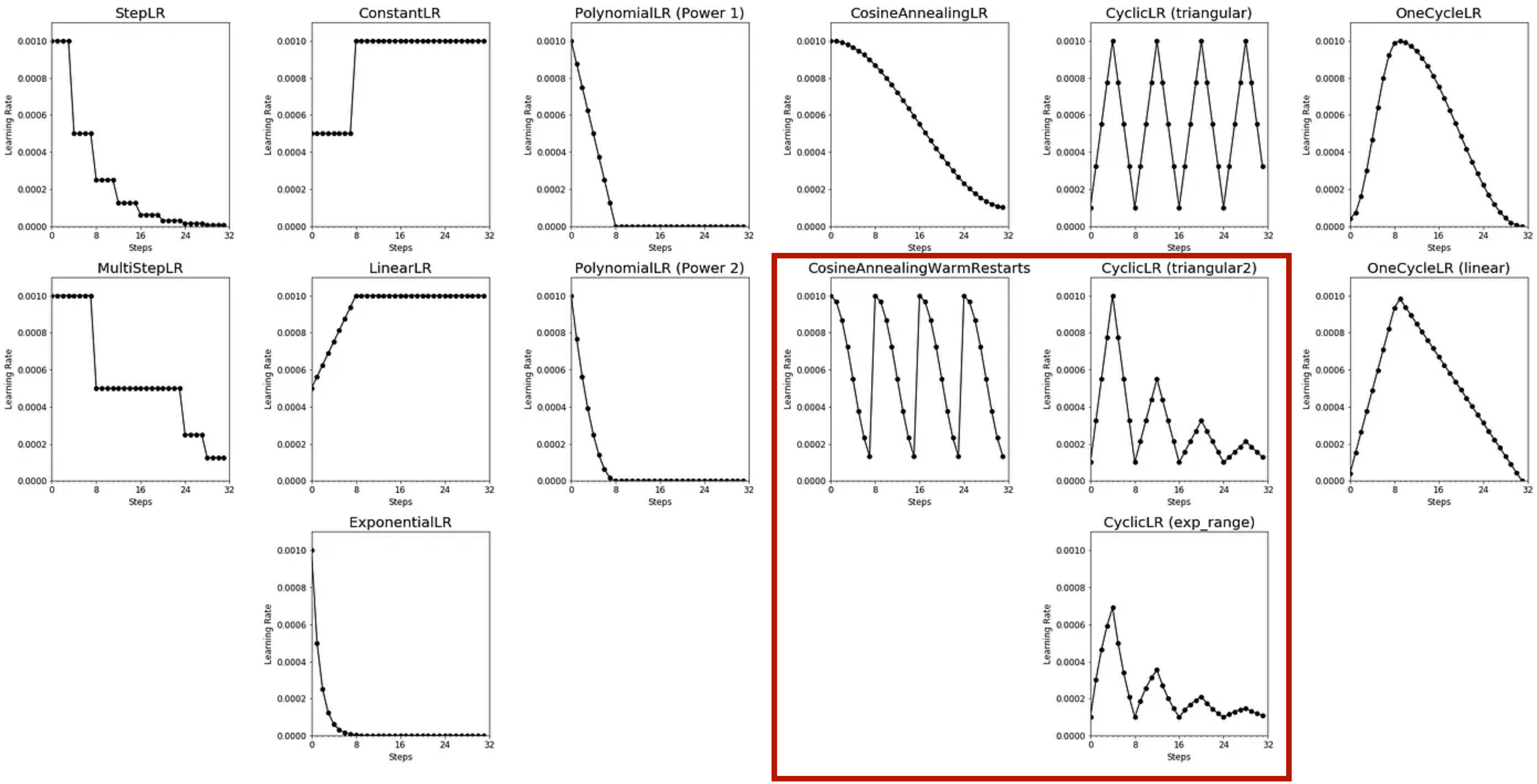
- **Decay.** A typical solution is to use ***learning rate decay***.



Note. Optimizers have different sensitivities to lr decay
(e.g., LR decay is less critical in Adam than SGD + Momentum)

LR schedule — more of an art

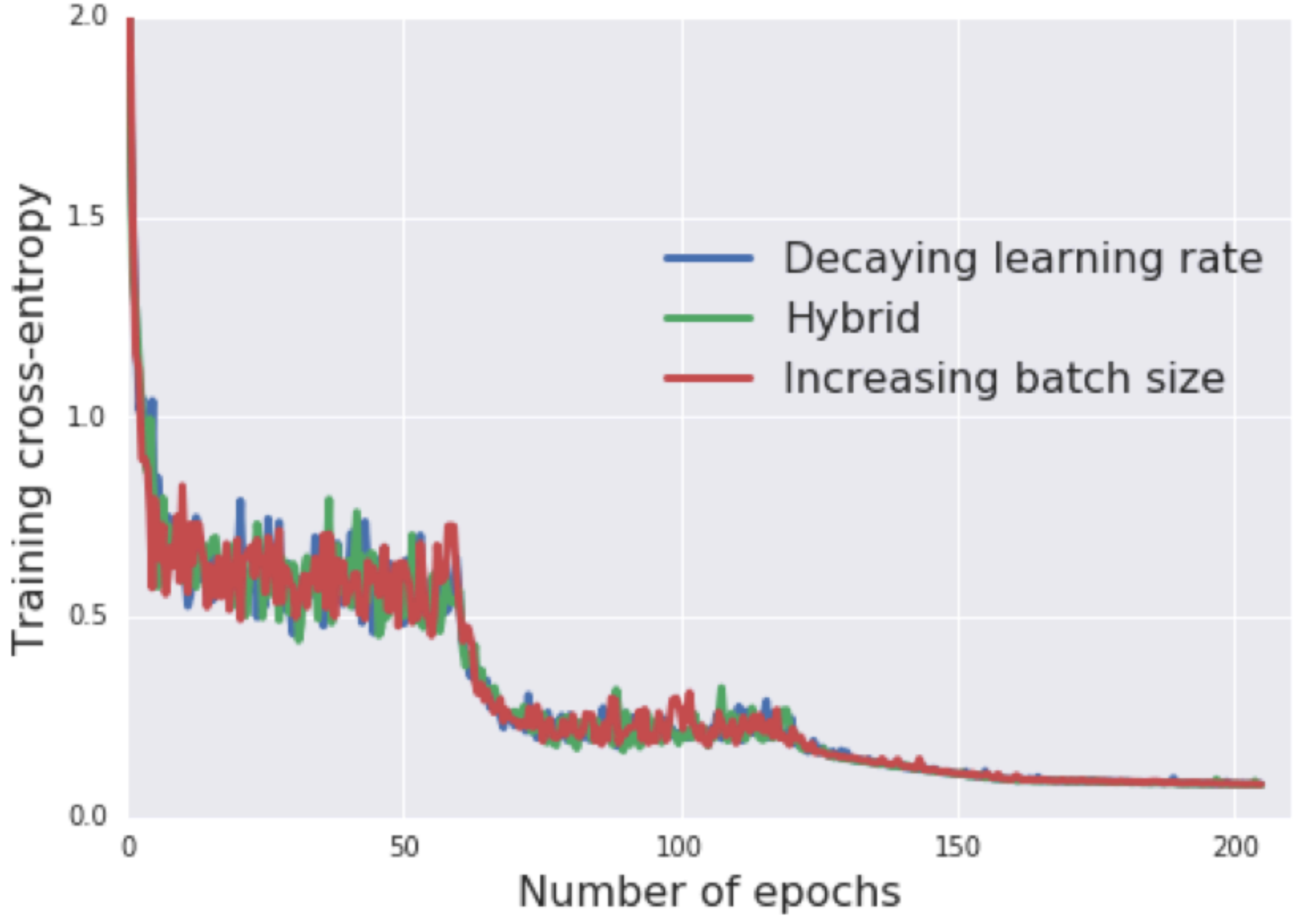
- Nowadays, it is quite common to use cosine/cyclic LR with warmup.



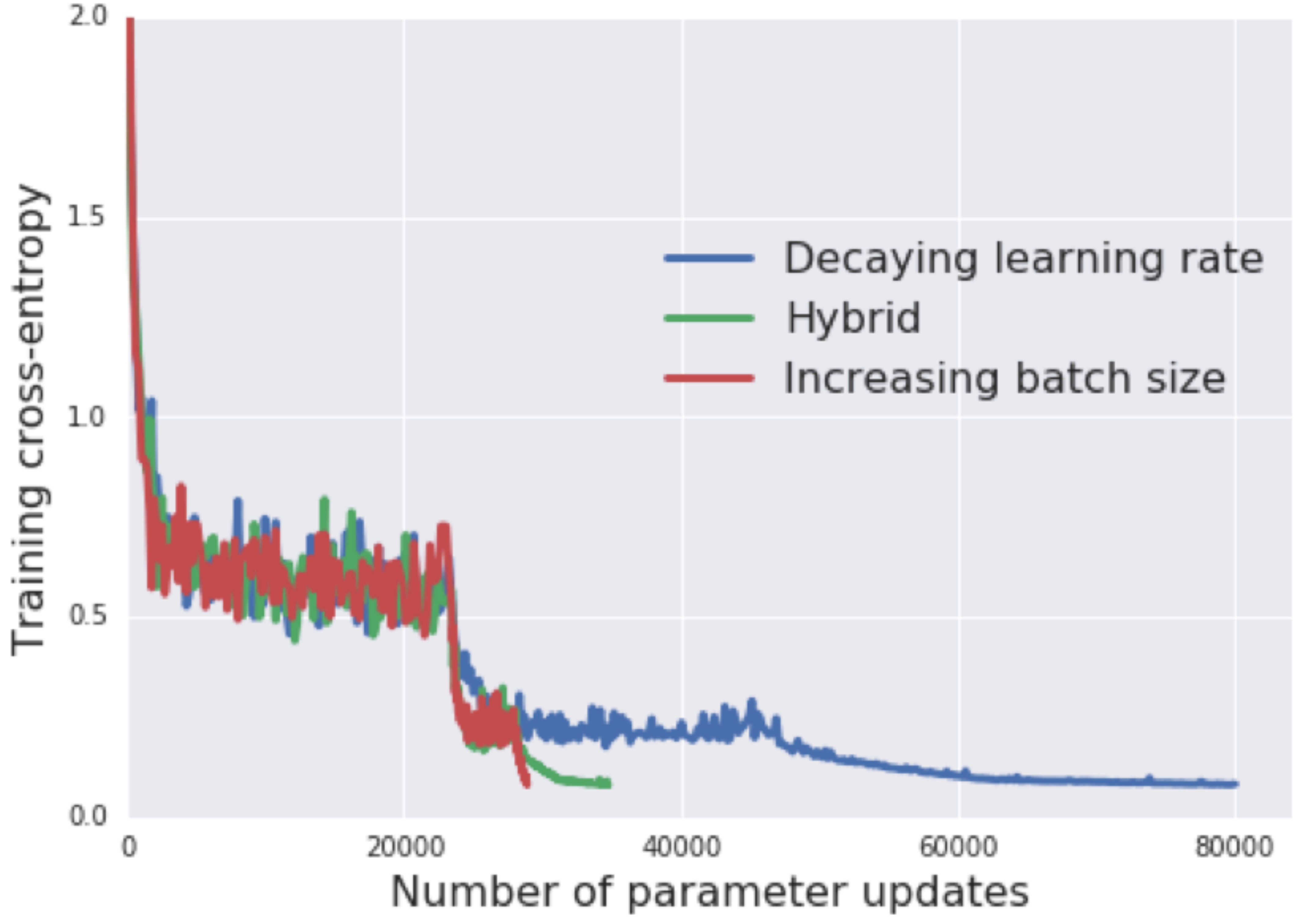
Smith et al., “Don’t decay the learning rate, increase the batch size,” ICLR 2018

Efficiency — the batch size

- Increasing the batch size ≈ Decreasing the learning rate



(a)

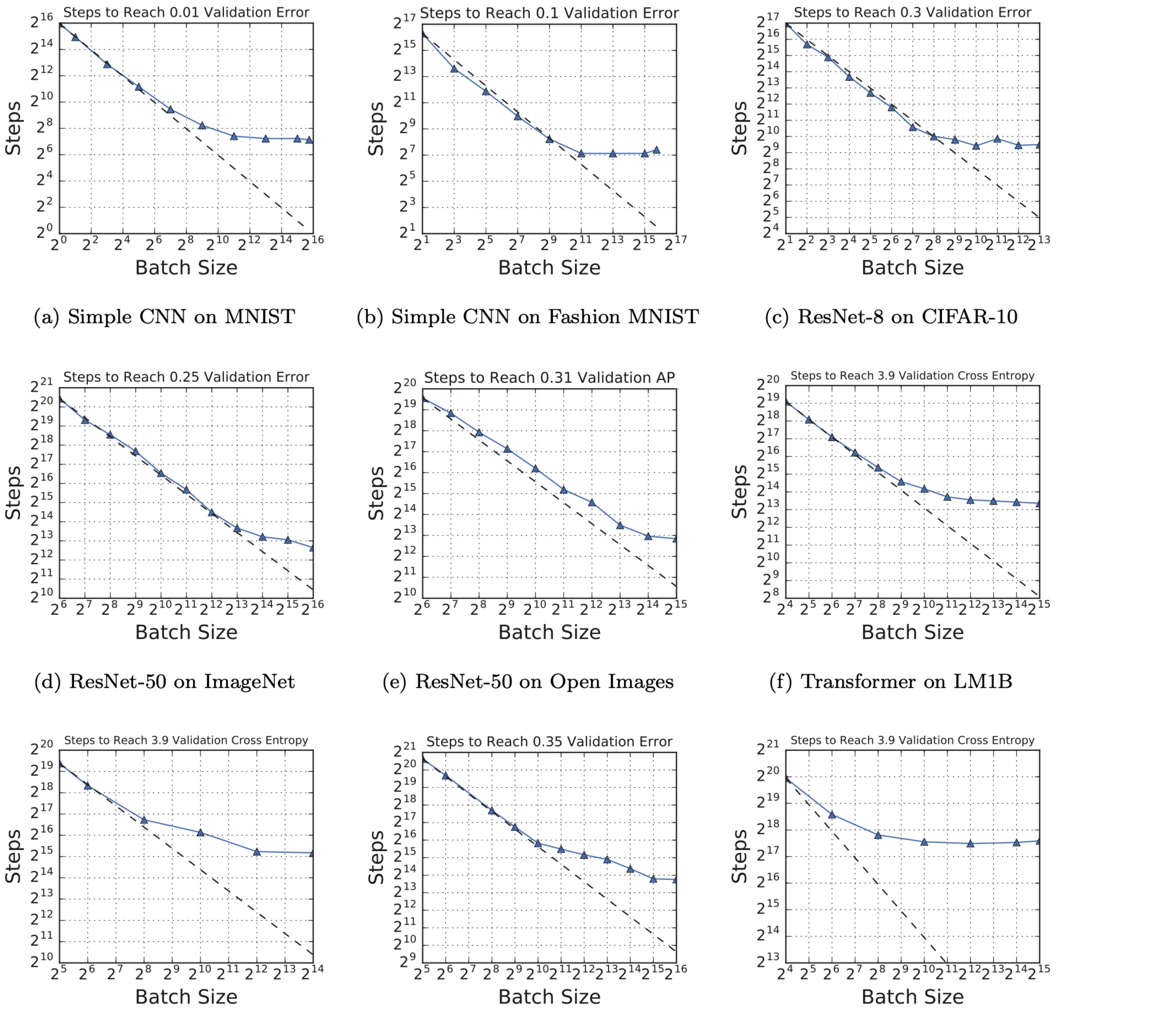


(b)

Efficiency — the batch size

- Using the larger batch ***speeds up*** the overall training procedure.
    - but the benefit ***saturates***

Note. Interestingly, the optimal LR scales linearly with the batch size.



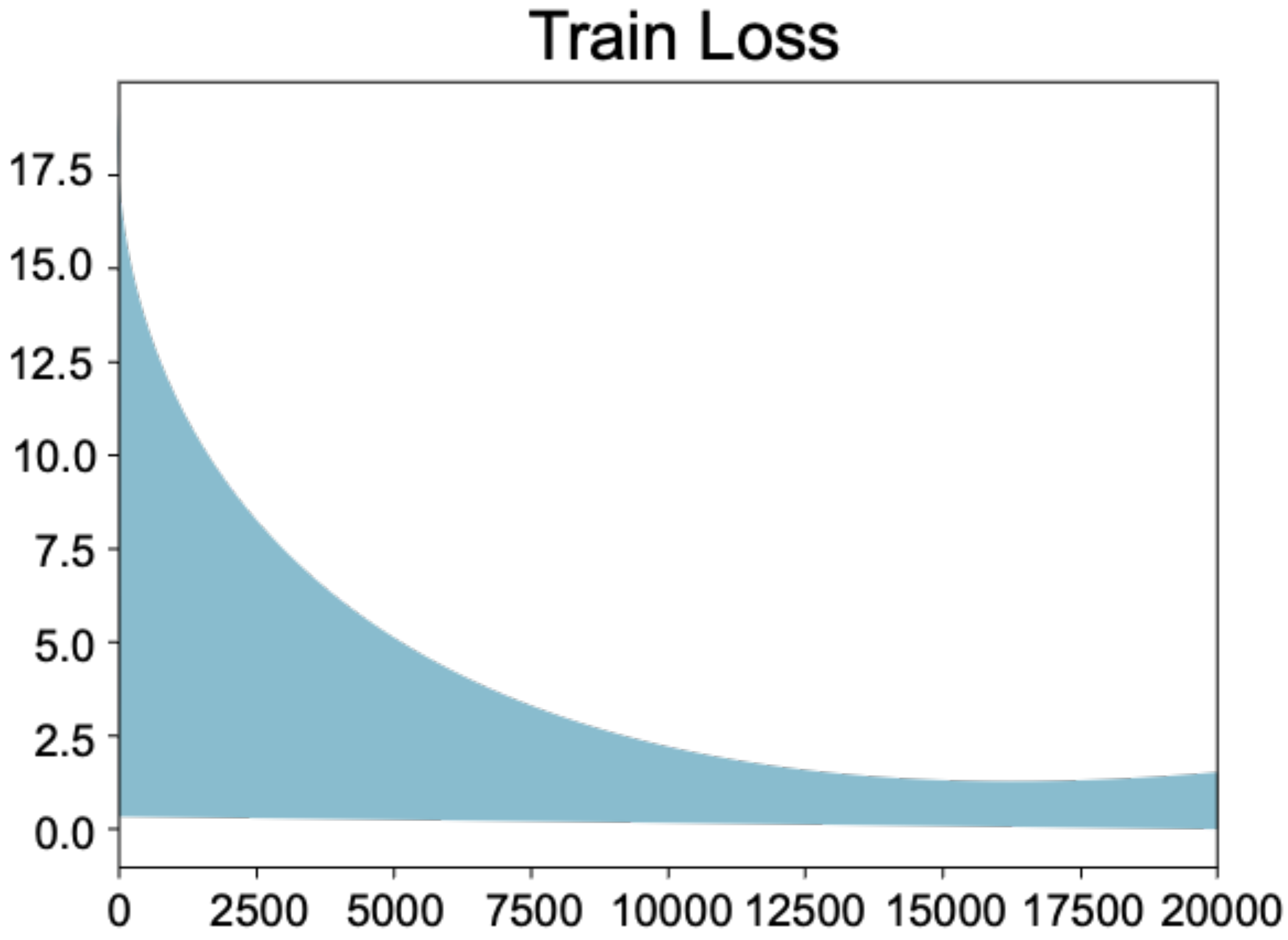
(a) Simple CNN on MNIST
(b) Simple CNN on Fashion MNIST
(c) ResNet-8 on CIFAR-10
(d) ResNet-50 on ImageNet
(e) ResNet-50 on Open Images
(f) Transformer on LM1B
(g) Transformer on Common Crawl
(h) VGG-11 on ImageNet
(i) LSTM on LM1B

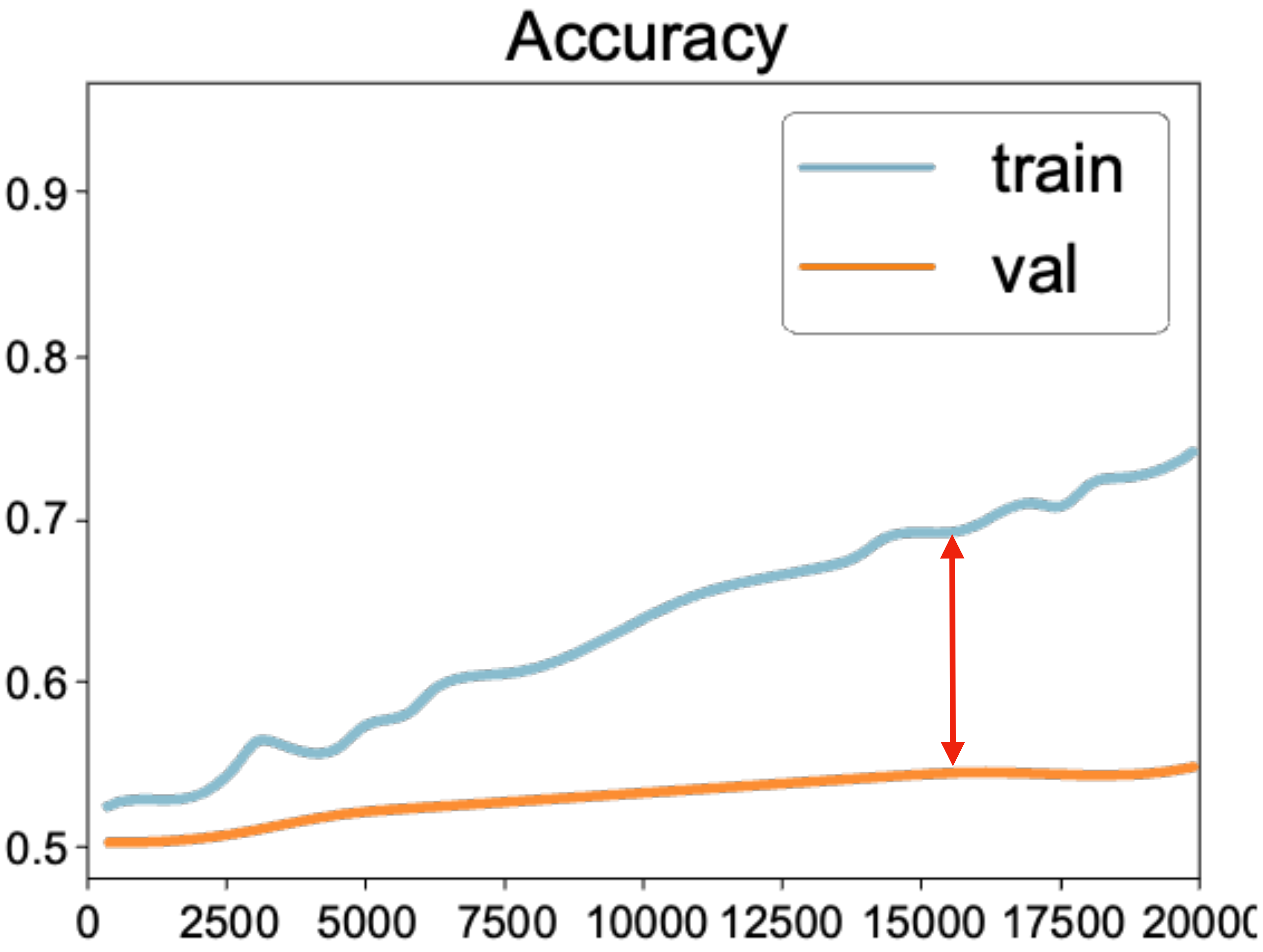
What we did not cover

- Detailed discussions on how advanced optimization algorithms work.
  - **Momentum.** https://distill.pub/2017/momentum/
  - **Adam.** https://optimization.cbe.cornell.edu/index.php?title=Adam
  - **Others.** https://cs231n.github.io/neural-networks-3/

Regularization

Beyond Training Error





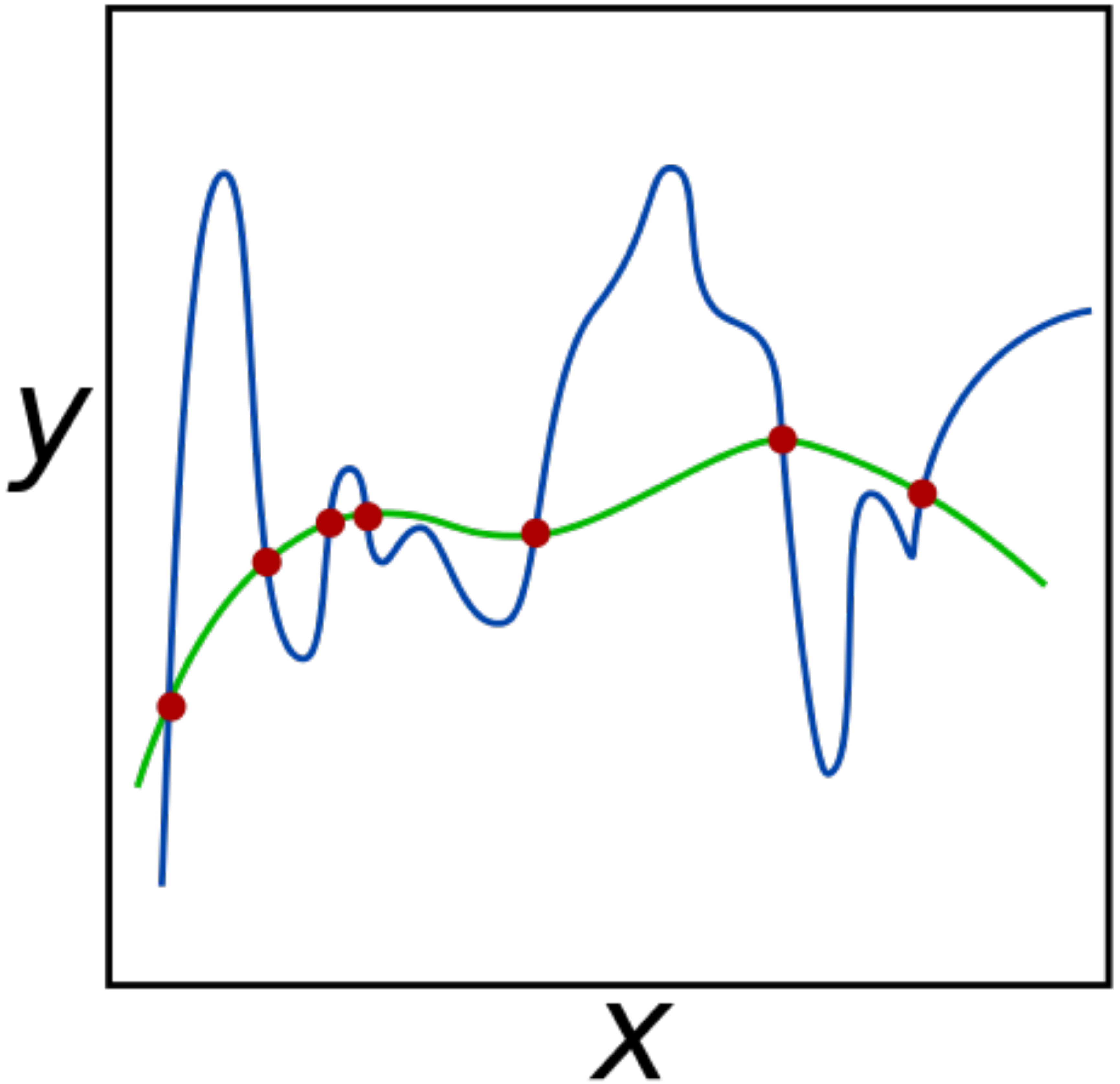
Better optimization algorithms help reduce the training loss

But we actually care about the test performance—how to reduce the gap?

Core Philosophy

- Most regularization methods follow the principle of ***Occam's razor***:

"Whenever possible, use simpler models"



Core Philosophy

- **Simplicity of the model?**
  - Many definitions—smaller norm weights
    sparse weights
    have smaller prediction confidence…
- **How to force simplicity?**
  - Add penalty to the loss.
  - Modify the architecture…
- **Note.** Also eases the optimization — recall the midterm!

Case 1. L2 Regularization

- **Simplicity.** Whenever possible, use smaller ℓ_2 norm weights.

- **Method.** Directly adding to the regularization term

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla_\theta (L(\theta) + \lambda \cdot \|\theta\|_2)$$

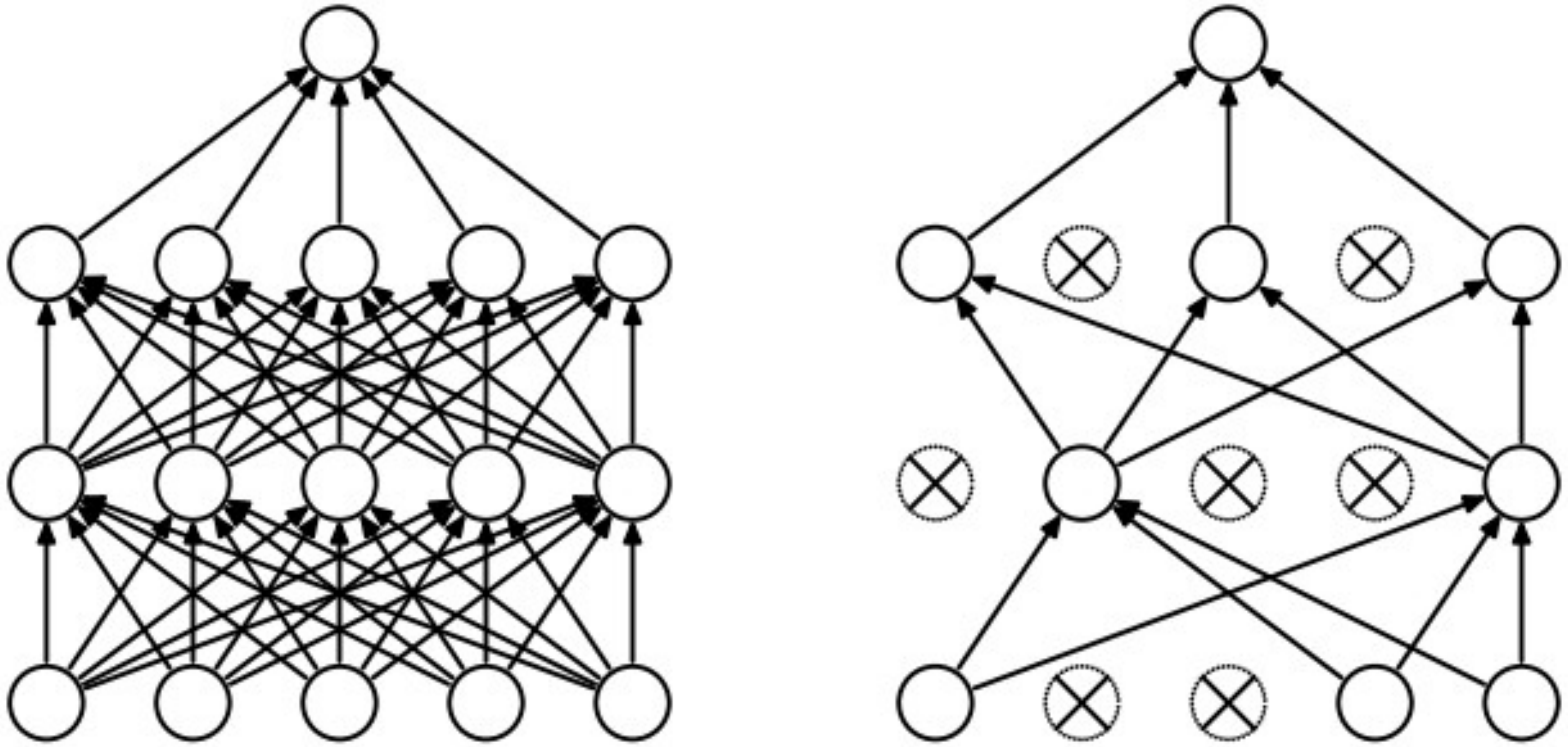
In fact, this is equivalent to a simpler-to-implement form:

$$\theta^{(t+1)} = (1 - \eta\lambda)\theta^{(t)} - \eta \cdot \nabla_\theta L(\theta)$$

(thus often called "weight decay")

Case 2. Dropout

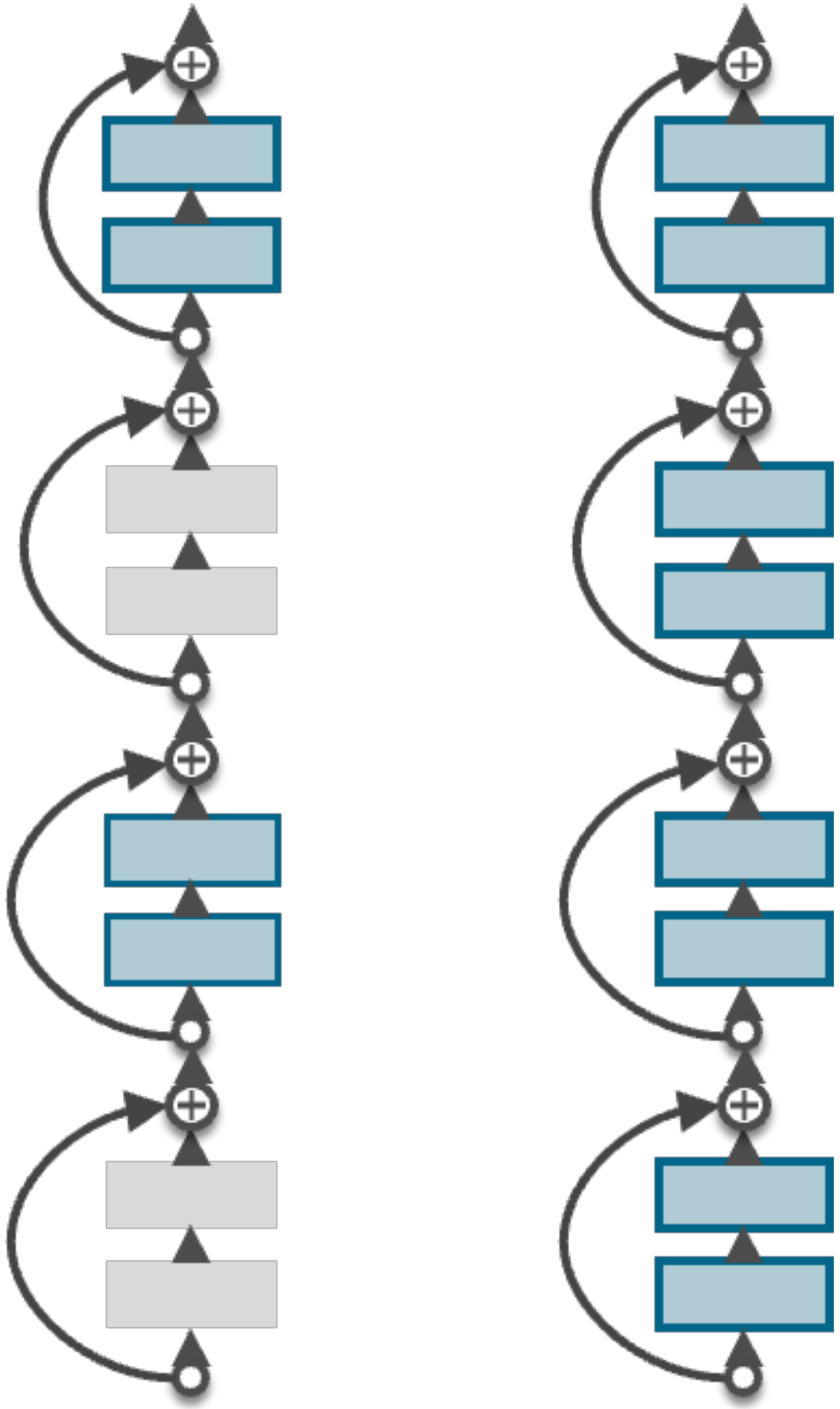
- **Simplicity.** Whenever possible, use smaller subnetwork.
- **Method.** During the training, randomly remove each neuron, w.p. p.
  - For the inference, rescale the weights back to $1/p$.



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

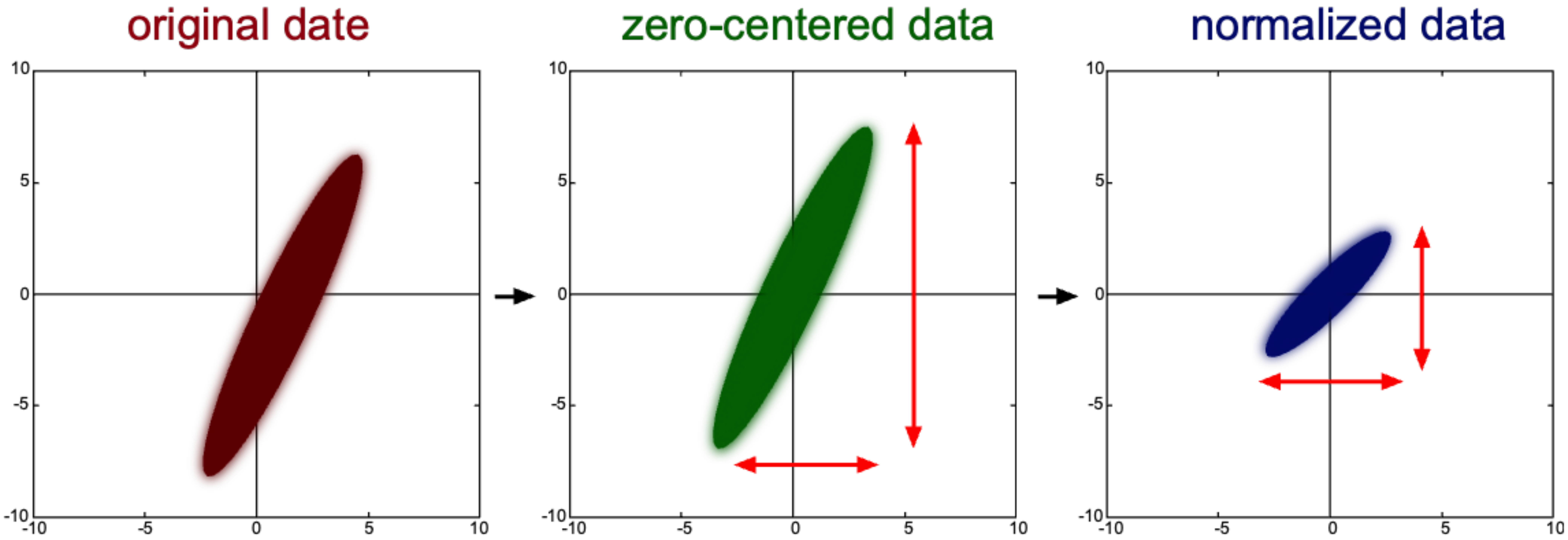
Case 2. Dropout

- **Note.** This is actually being used for training models like ChatGPT.
    - e.g., “Stochastic depth” removes some layers



Babysitting the learning process

Step 1. Preprocess the data




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

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

Here, we assume that $\mathbf{X} = \mathbb{R}^{n \times d}$, so that the first axis is along the data indices

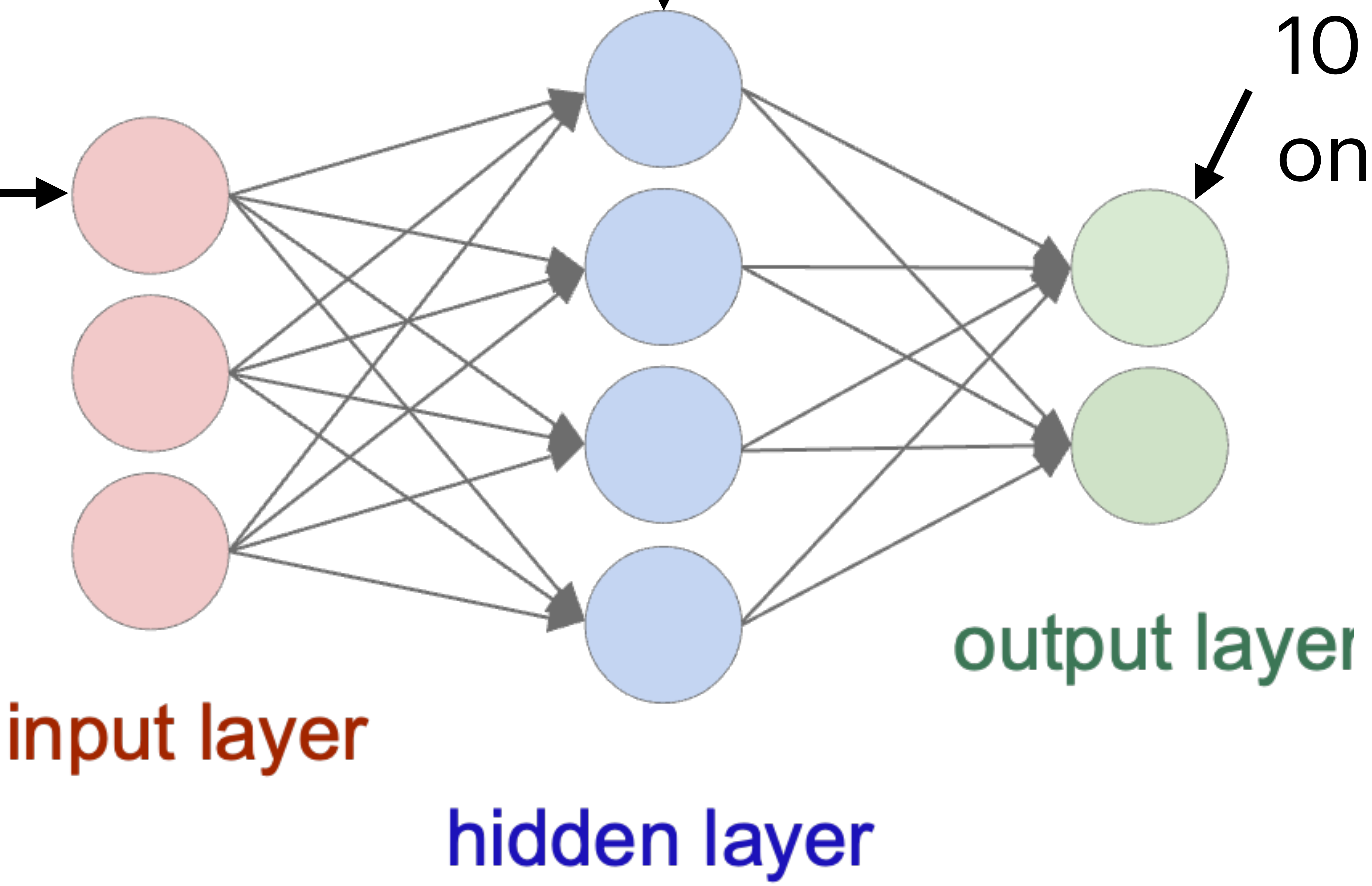
Step 2. Choose the architecture

50 hidden neurons

10 output neurons; one per class

CIFAR-10 images

$32 \times 32 \times 3 = 3072$



Step 3. Set up the loss

```
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
```

```
2.30261216167
```

disabled regularization

loss—looks reasonable for an untrained model

$$\ln(1/10) \approx -2.302585$$

Step 3. Set up the loss

```
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
```

```
3.06859716482
```

cranked up reg

loss went up— sanity check passed.

Step 4. Train

Tip. Make sure you can perfectly fit the 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)
```

Can we fit the first 20 samples from CIFAR-10, using SGD without regularization?

Step 4. Train

Tip. Make sure you can perfectly fit the 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
```

Step 4. Train

Tip. Make sure you can perfectly fit the very small portion of the training data

```
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
```

training accuracy is small, so we can train indeed!

Step 4. Train

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

```
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)
```

Step 4. Train

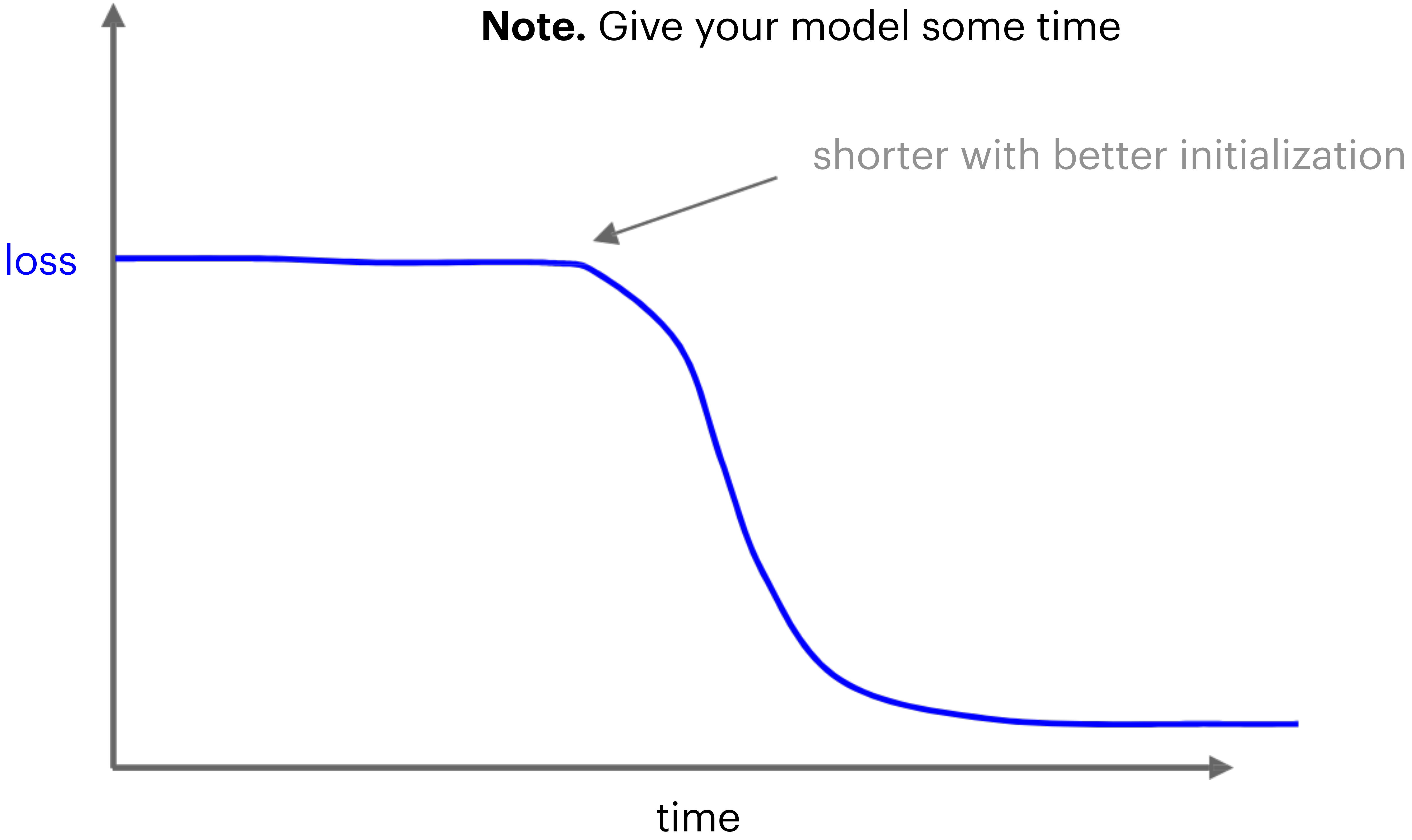
Start with small regularization and find the learning rate that makes the loss go down.

```
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
```

the loss stays similar... maybe LR too low

Step 4. Train



Note. Give your model some time
shorter with better initialization
loss
time

Step 4. Train

If the LR is too high, you'll see NaNs...

(or nondecreasing losses)

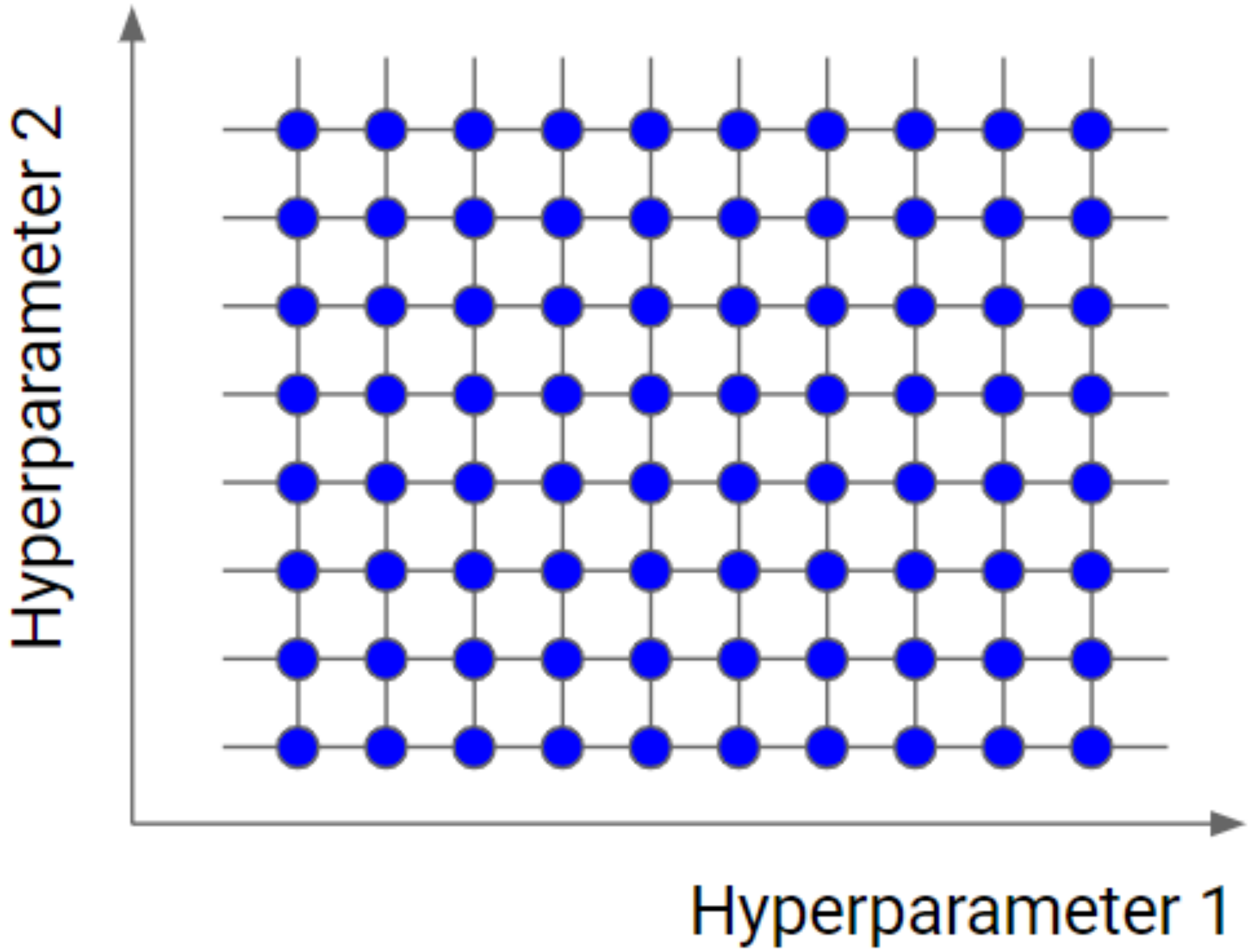
```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,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

Hyperparameter Optimization

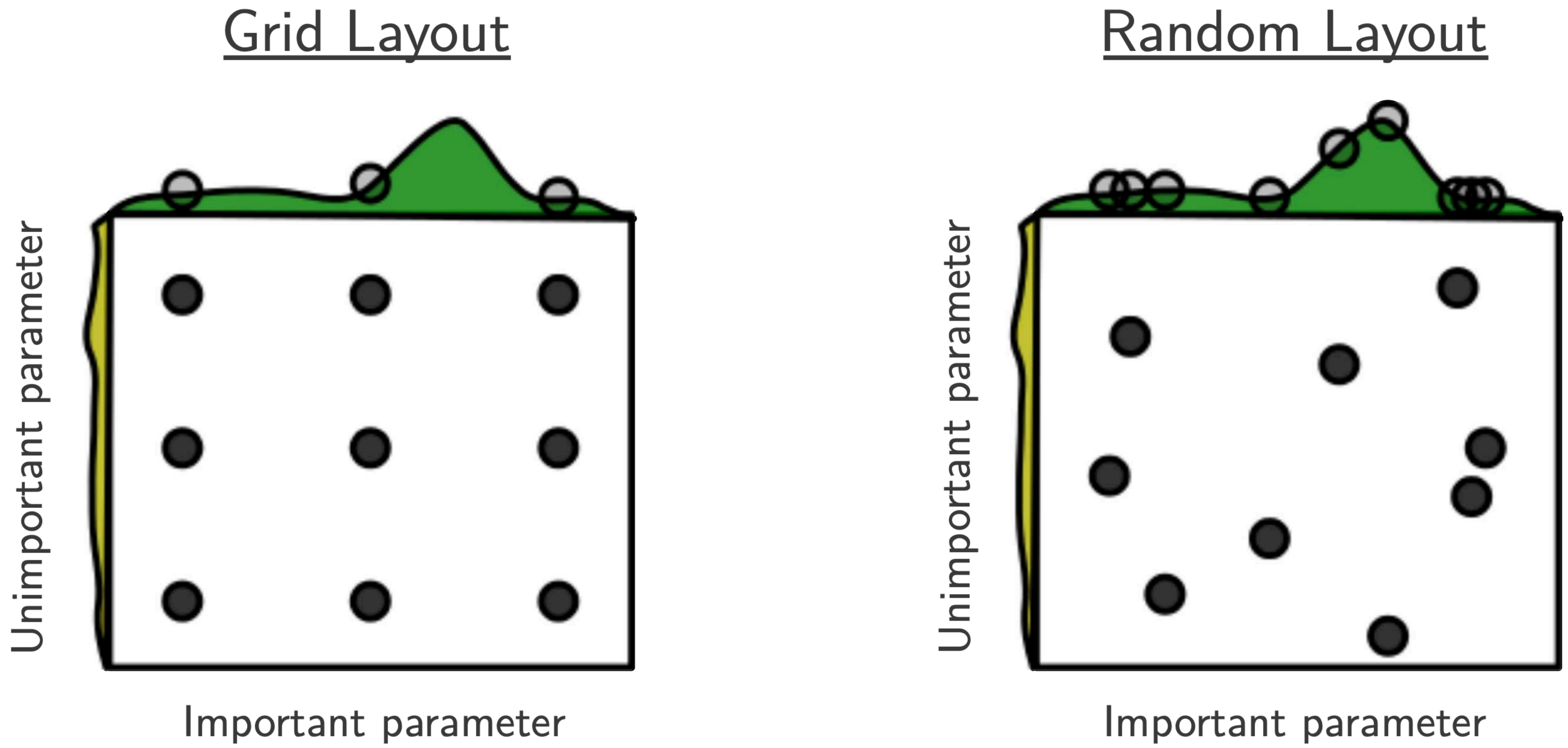
Strategy

- The elementary strategy is the **grid search**
  - use coarse-to-fine grids, to reduce #trials
  - sometimes we use log-scales
    - **LR.** $10^{-2}, 10^{-3}, 10^{-4}$



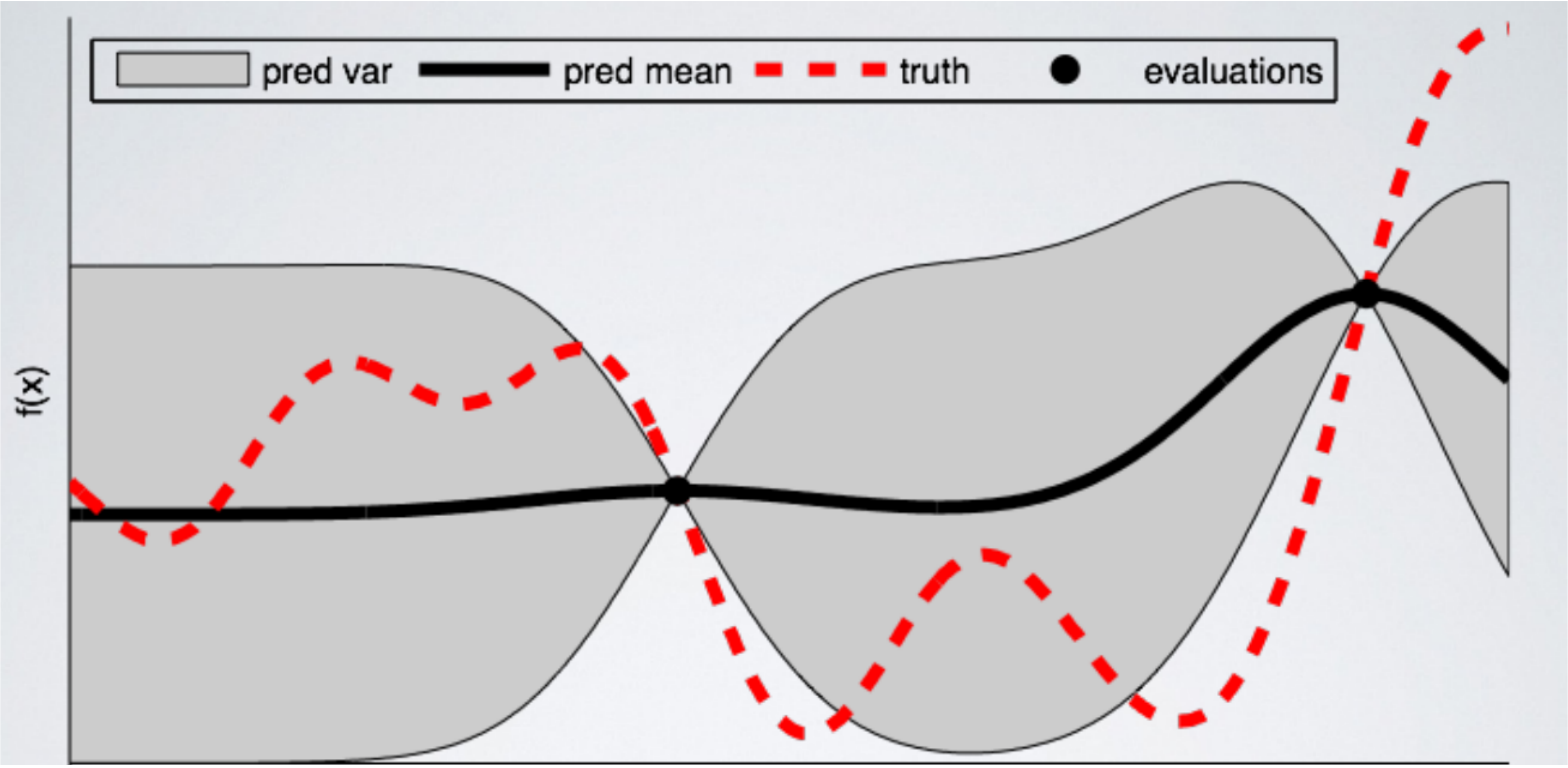
Strategy

- Also quite common to use the **random search**
  - Larger “effective sample size.”



More sophisticated...

- In some cases, we use Bayesian HP optimization techniques...
  - Predicting the performance, with Gaussian Processes



More sophisticated...

- In some cases, we can use the hyperparameter transfer...

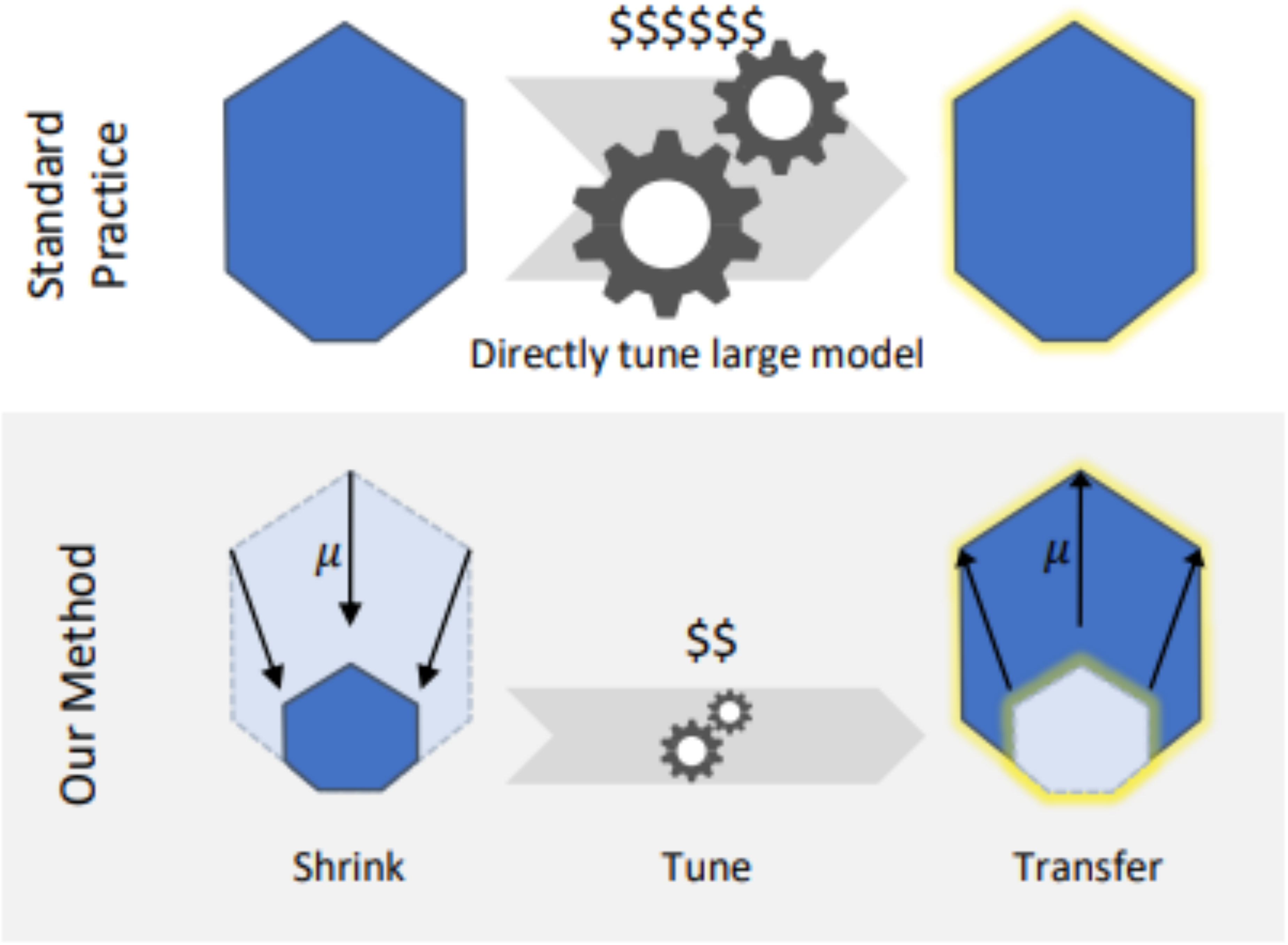


Figure 2: Illustration of μTransfer

Cheers

- *Next up.* Tasks that deep learning solves