# Lecture 6
# Regularization and training recipes

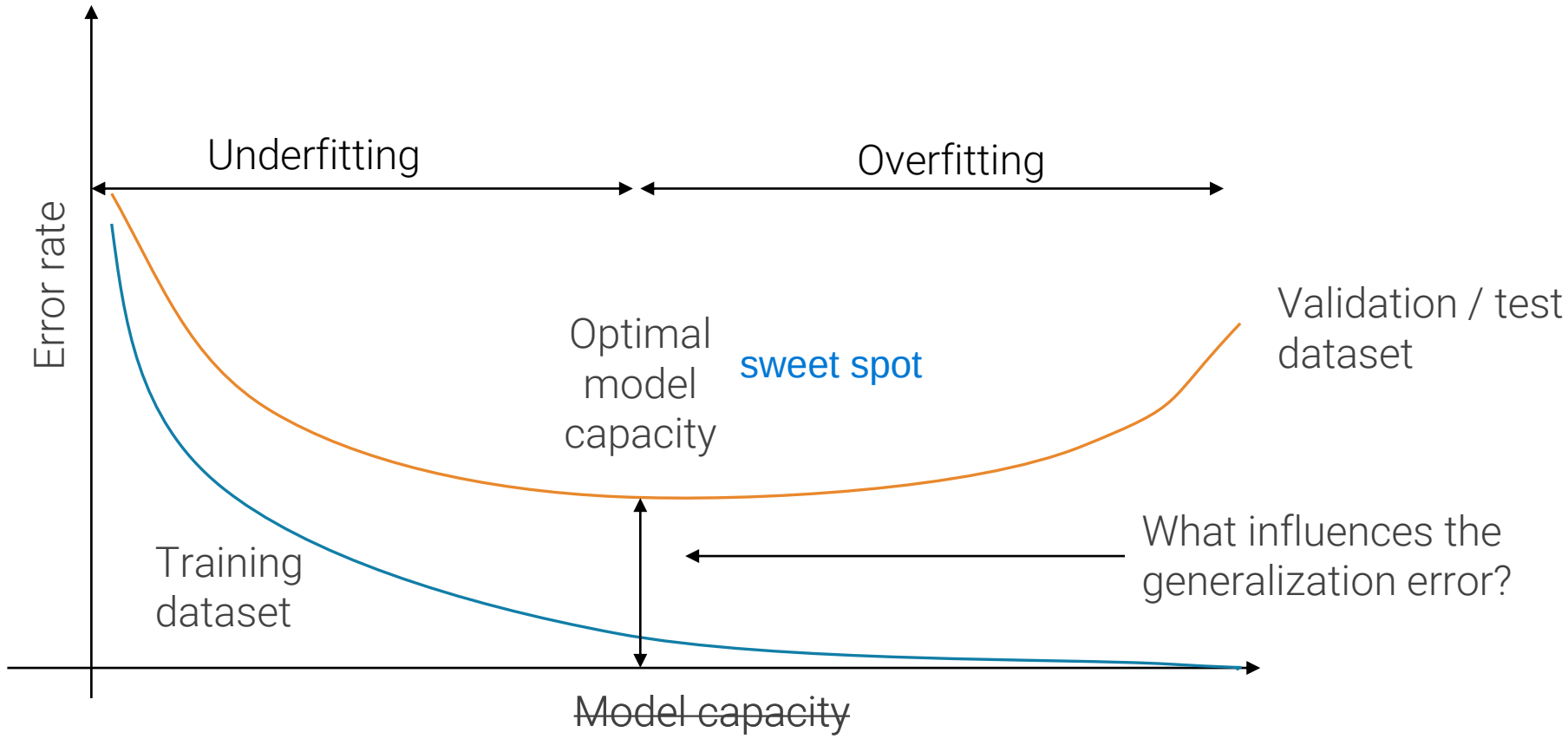
IMAGE PROCESSING AND COMPUTER VISION – PART 2

SAMUELE SALTI

# Generalization error

hyperparams change behavior of model

Model Capacity: higher capacity means the model can capture more complex patterns in the data, and it can be increased using more complex models or higher-degree polynomial features.




I can use VC dim to measure model capacity

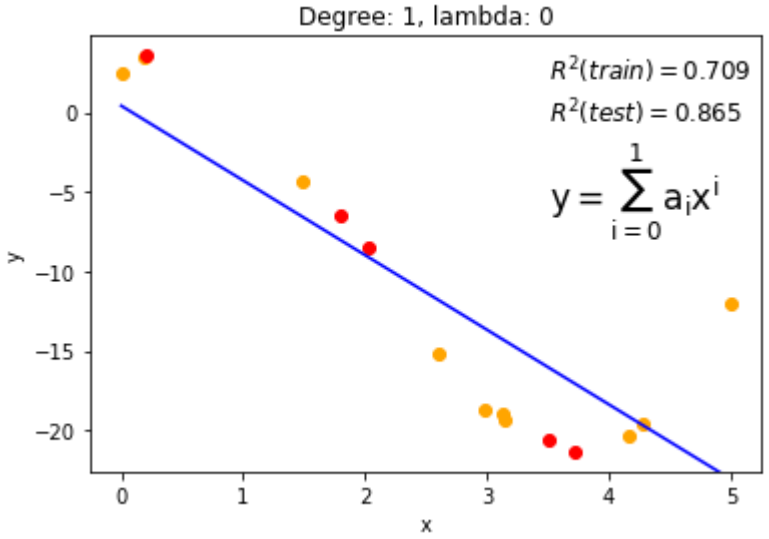
# Example: generalized linear regression

noise, no perfect models then

Suppose we fit $N$ data points, generated by a noisy cubic equation $y = (ax^3 + bx^2 + cx + d) + \epsilon$ with different models, which are polynomials of degree $K$ of the form $y = \sum_{i=0}^{K} a_i x^i = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$

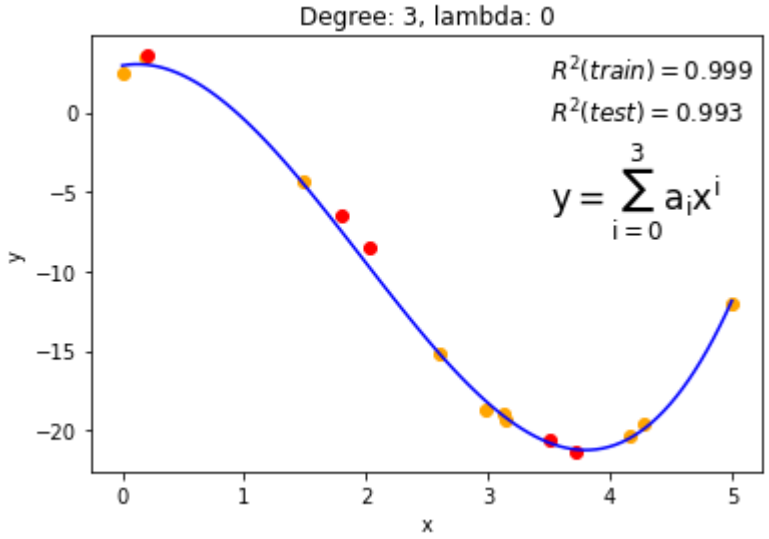
the capacity

Linear model
Underfitting



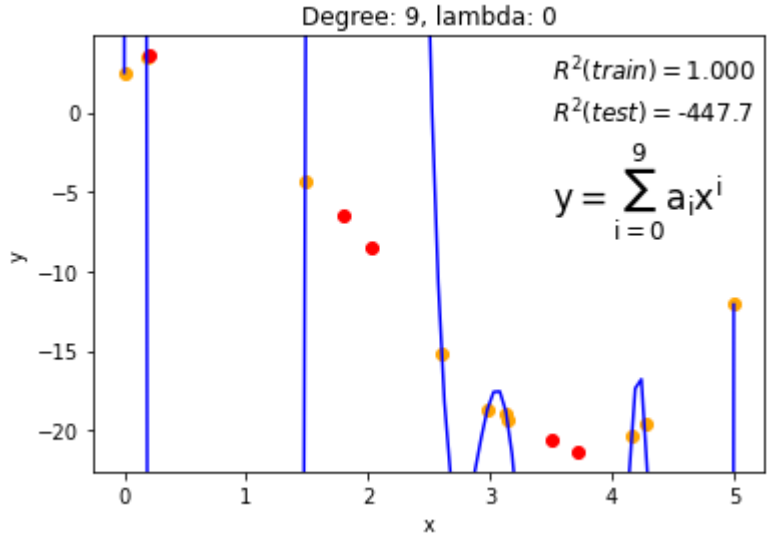

"Right" model

with the right capacity (degree here)




"Interpolating" model
Overfitting

it perfectly fits training data



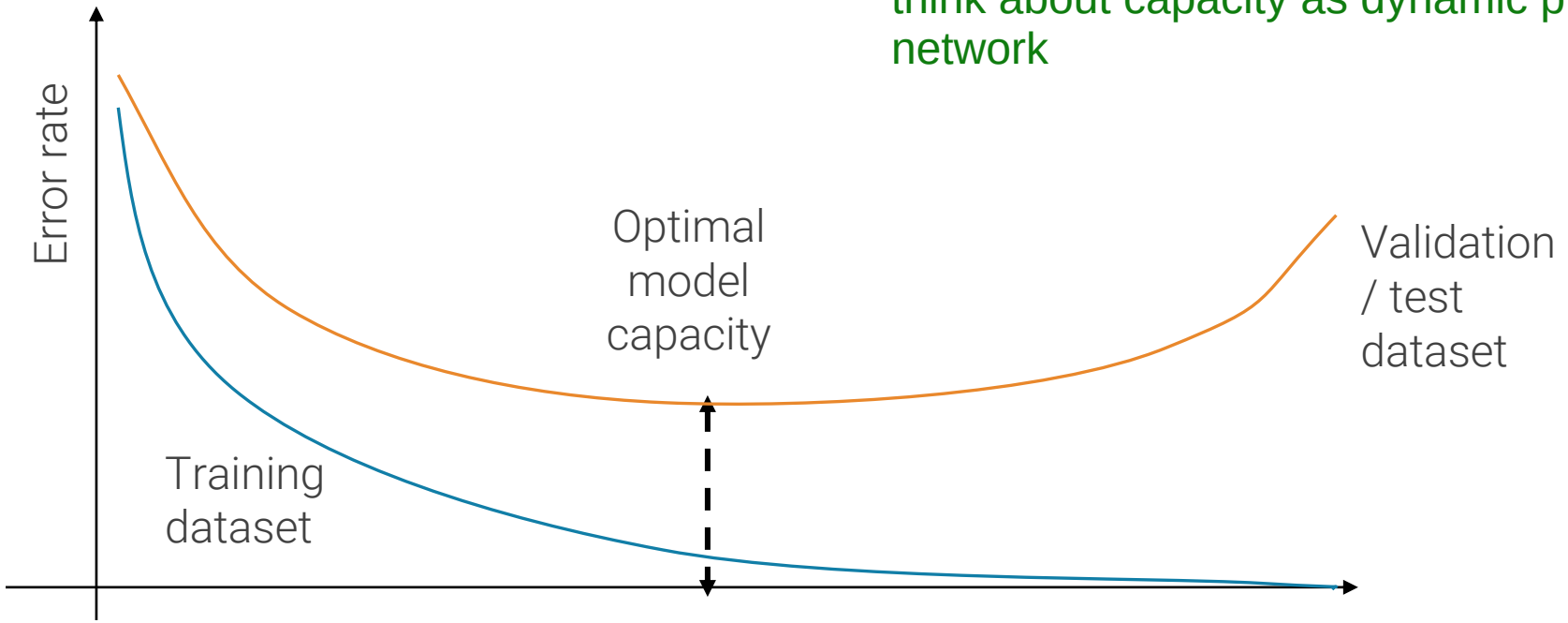

http://www.deeplearningbook.org/contents/ml.html

# Effective capacity

how we can retireve the best results and the best configuration setting of our model. let it doing its best

In practice, we very often train a specific architecture, whose theoretical capacity is fixed and large. Yet, the **effective capacity** of an architecture can still changed due to **optimization hyperparameters** and **regularization**.

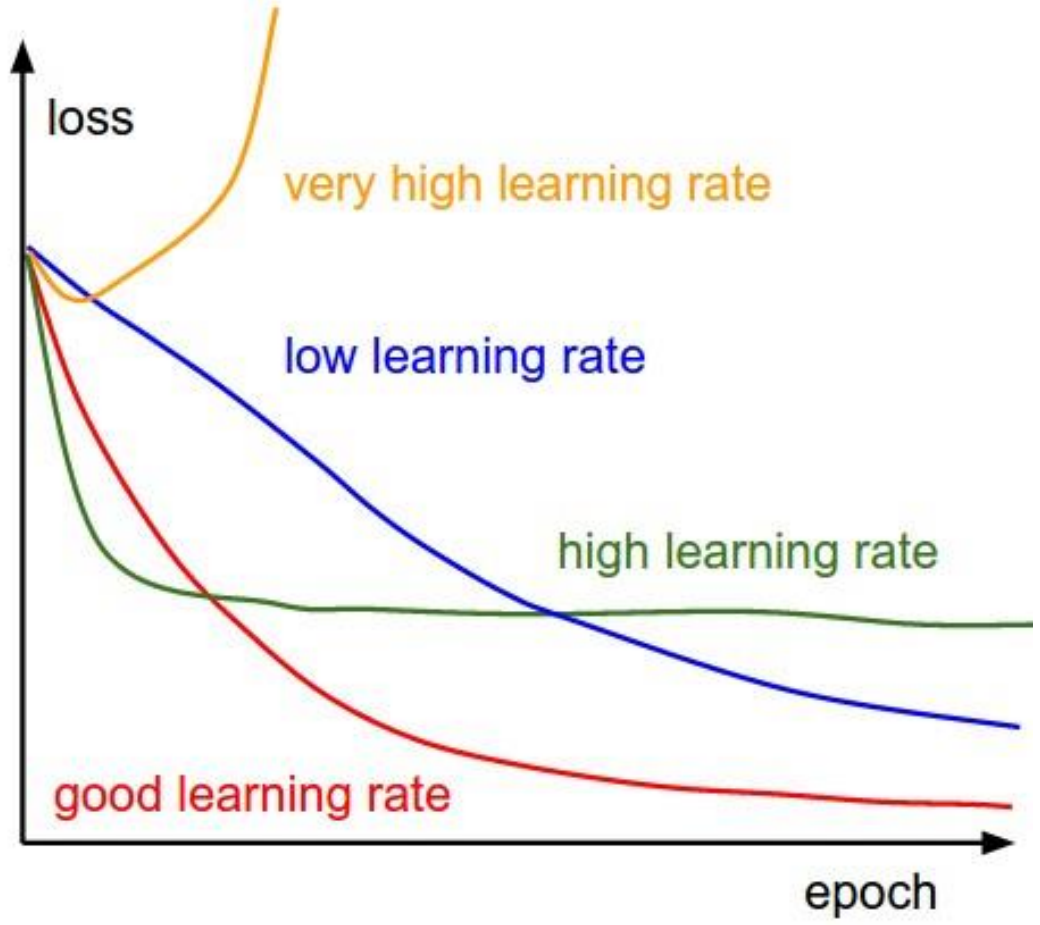
think about capacity as dynamic property of our network




~~Theoretical capacity: network width, network depth, input resolution, etc..~~

Effective capacity: learning rate, favor small values for parameters, training time, ...

# Learning rate is a key hyperparameter



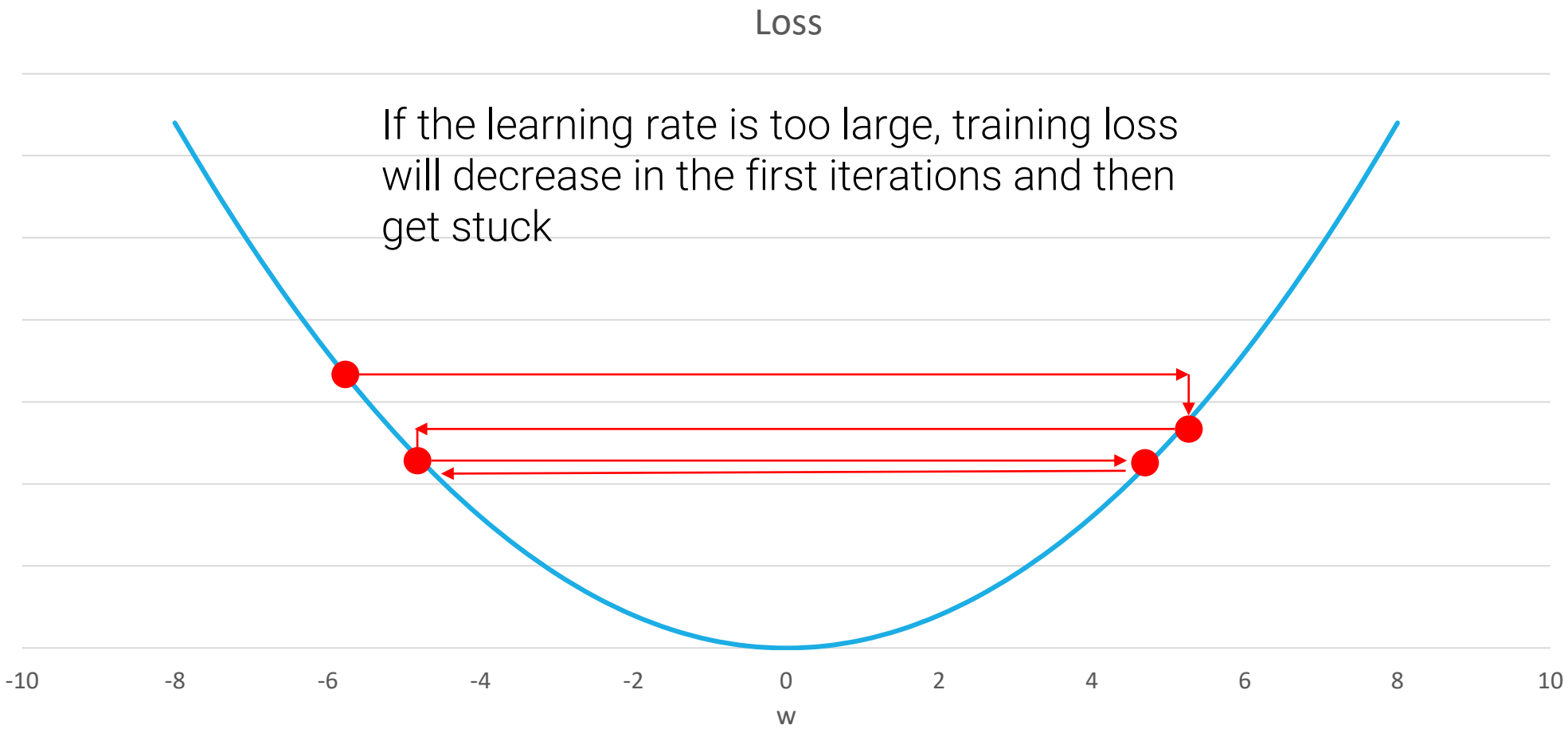

Problem: hard to find the good learning rate to reach good fitting

Solution: use a mixture of high and low.

Source: https://cs231n.github.io/neural-networks-3/

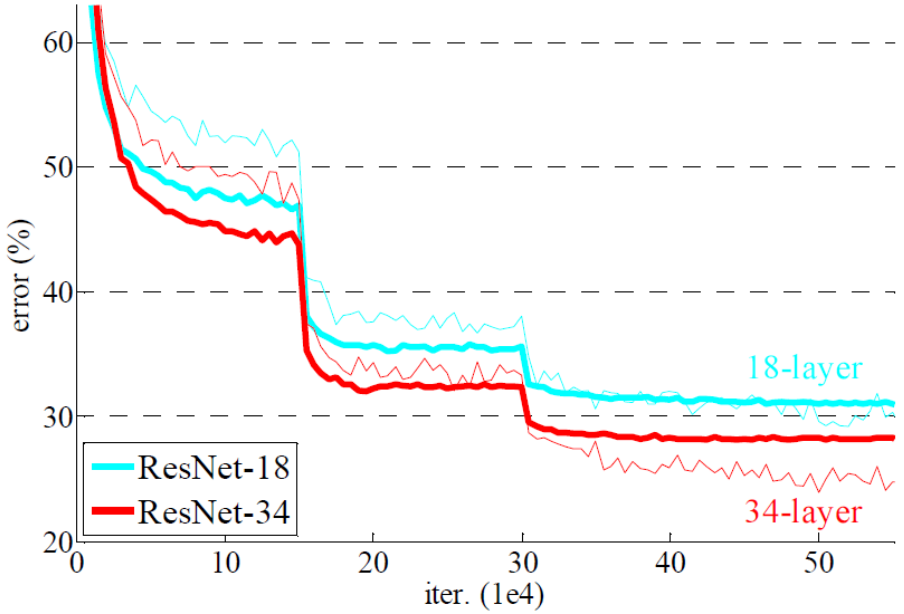
# Learning rate and gradient descent

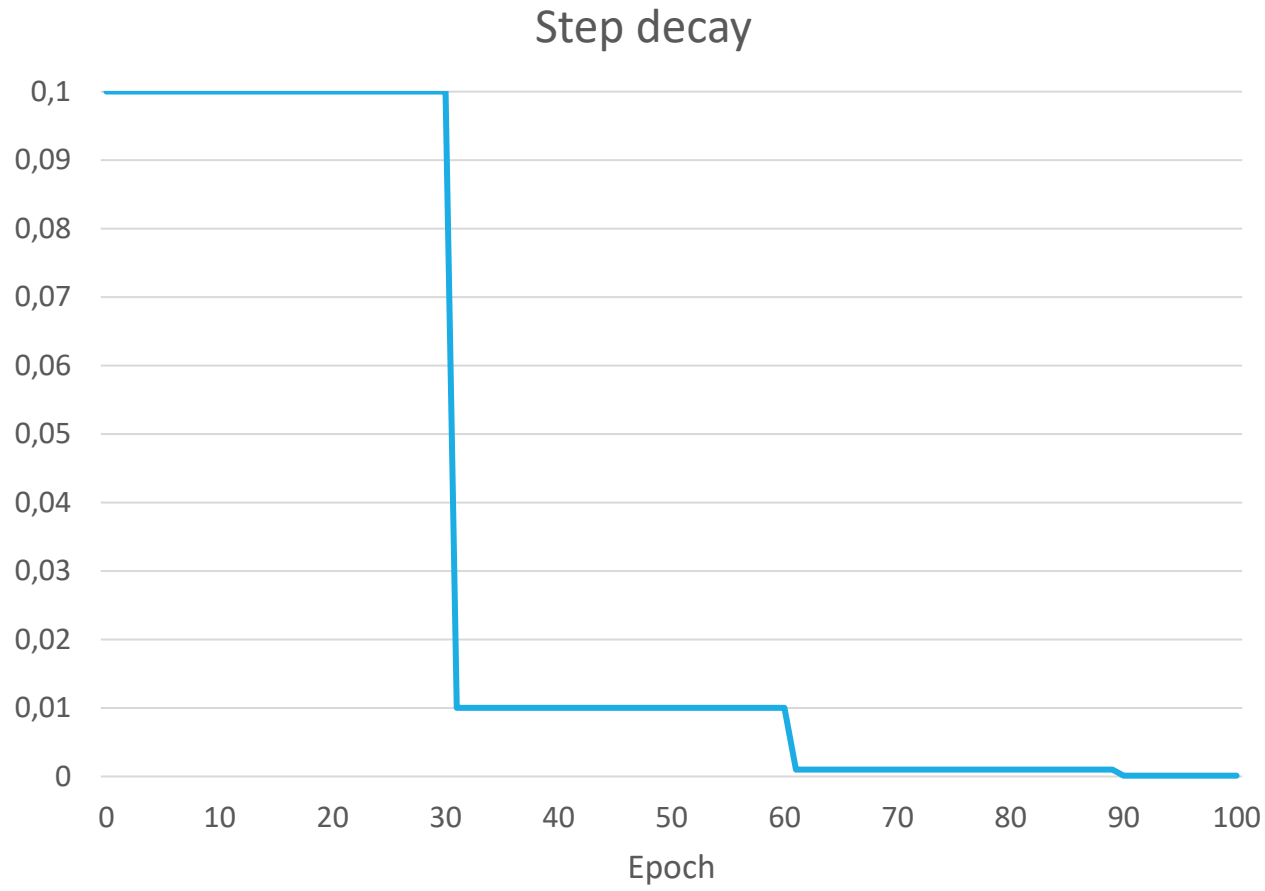
bouncing behavior



# Learning rate schedule: step decay

plan how and when its value has to change (decrease)



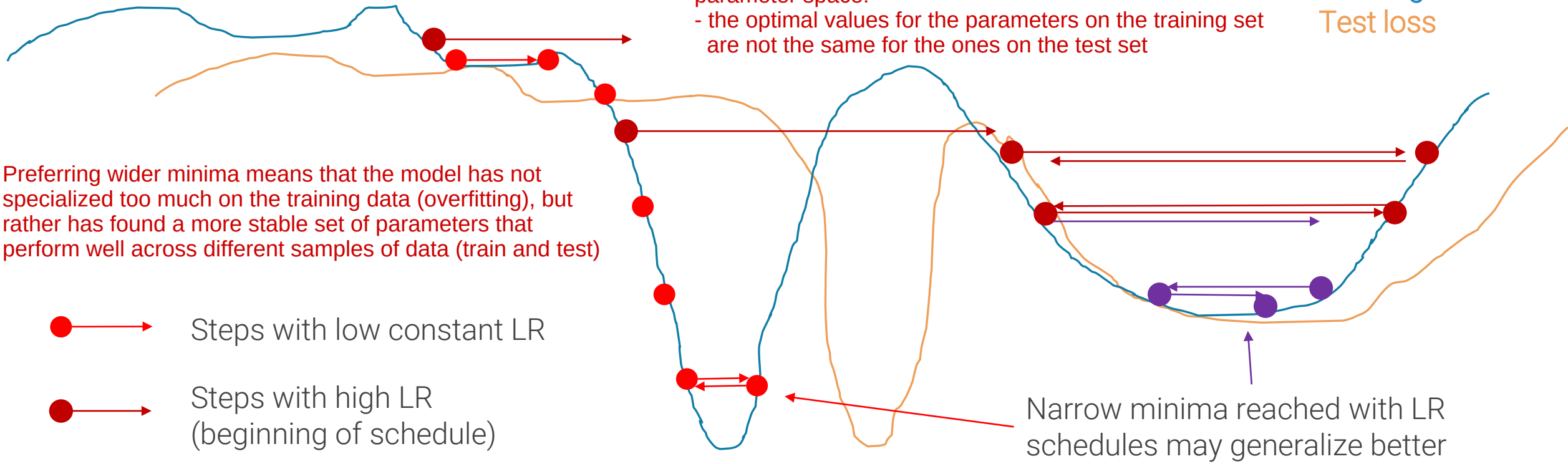

Step schedule (or decay): start with high learning rate (e.g. 0.1) and divide by 10 when the error plateaus
Used e.g., in ResNets

# Intuition: favor wide minima

why no overfitting? why use lr schedulers?

the test loss is kinda shifted wrt train loss in the parameter space:
- the optimal values for the parameters on the training set are not the same for the ones on the test set

Training loss
Test loss



Preferring wider minima means that the model has not specialized too much on the training data (overfitting), but rather has found a more stable set of parameters that perform well across different samples of data (train and test)

Steps with low constant LR

Steps with high LR (beginning of schedule)

Steps with low LR (end of schedule)

Narrow minima reached with LR schedules may generalize better

narrow minima are less robust to this shift, rather than wider minima, where the loss value is pretty the same

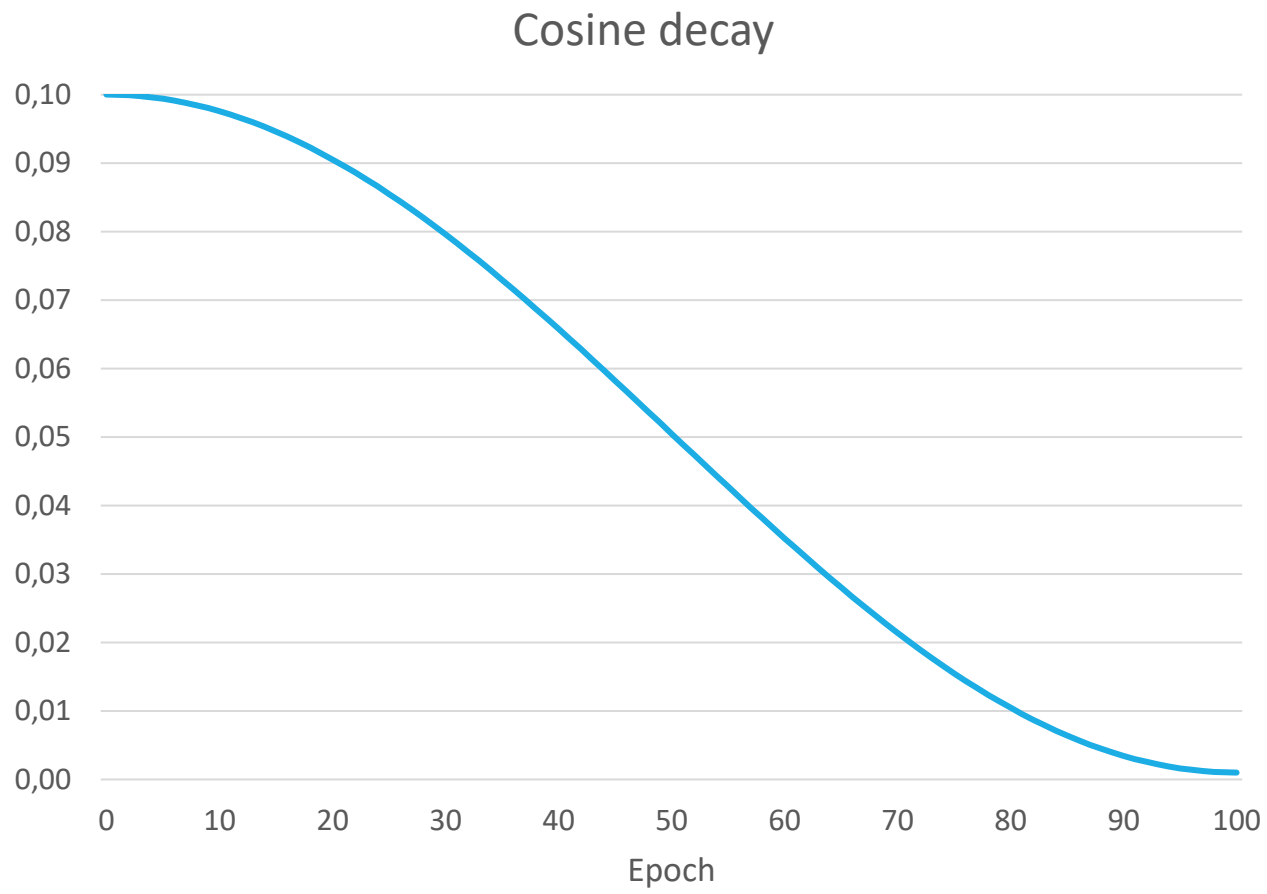
Sepp Hochreiter and Jurgen Schmidhuber. "Flat minima". Neural Computation, 9(1):1–42, 1997.
Laurent Dinh et al., "Sharp minima can generalize for deep nets.", ICLR 2017.
Nitish Shirish Keskar et al., "On large-batch training for deep learning: Generalization gap and sharp minima." ICLR 2017.

# Learning rate schedule: cosine

Cosine schedule (or decay): if training for $E$ epochs, learning rate for epoch $e$ is

$$lr_e = lr_E + \frac{1}{2}(lr_0 - lr_E)\left(1 + \cos\left(\frac{e\,\pi}{E}\right)\right)$$

It lets the optimizer follow a similar path to step decays with less hyper parameters to tune.



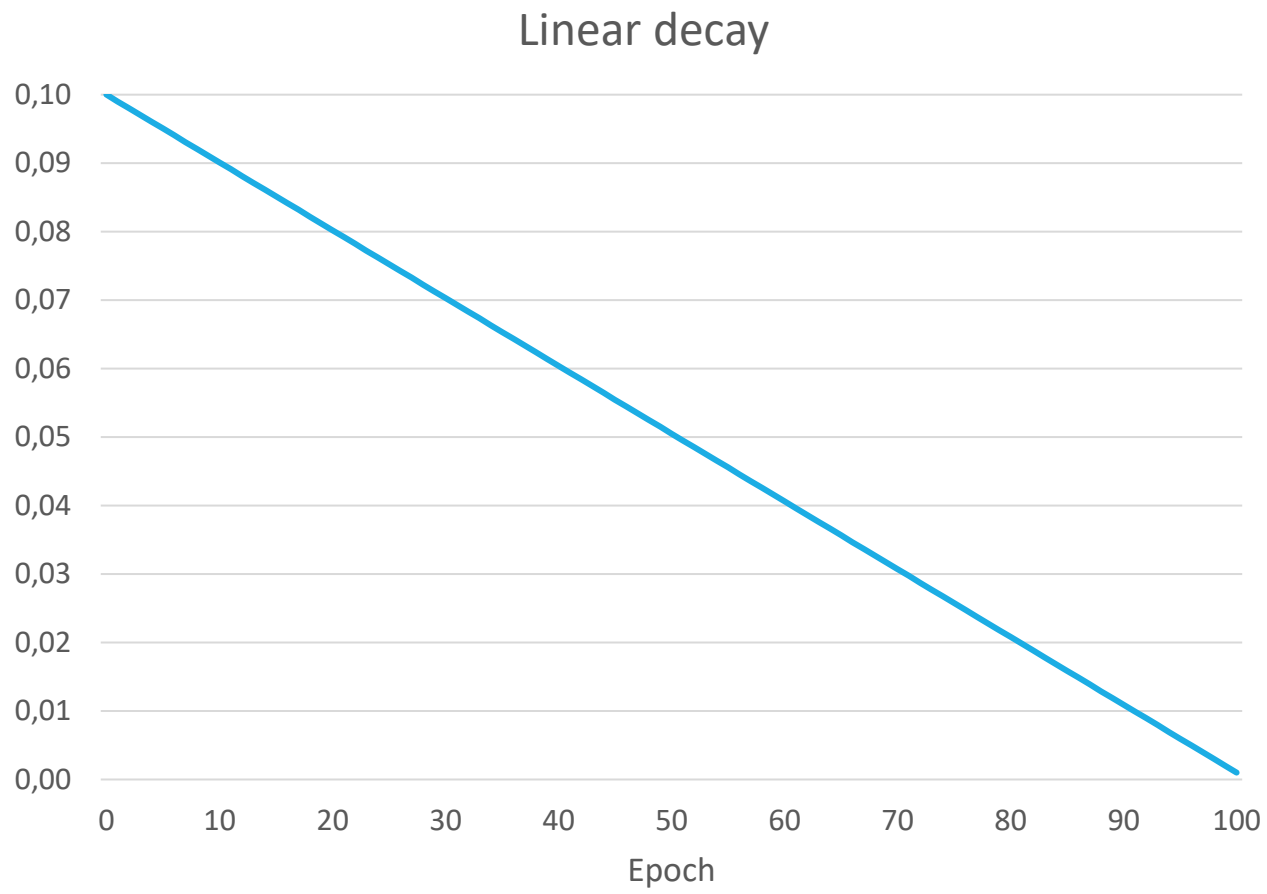

Ilya Loshchilov, Frank Hutter "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017

# Learning rate schedule: linear

Linear schedule (or decay): if training for $E$ epochs, learning rate for epoch $e$ is

$$lr_e = lr_E + (lr_0 - lr_E)\left(1 - \frac{e}{E}\right)$$

An even simpler alternative to coarsely emulate step decay evolution



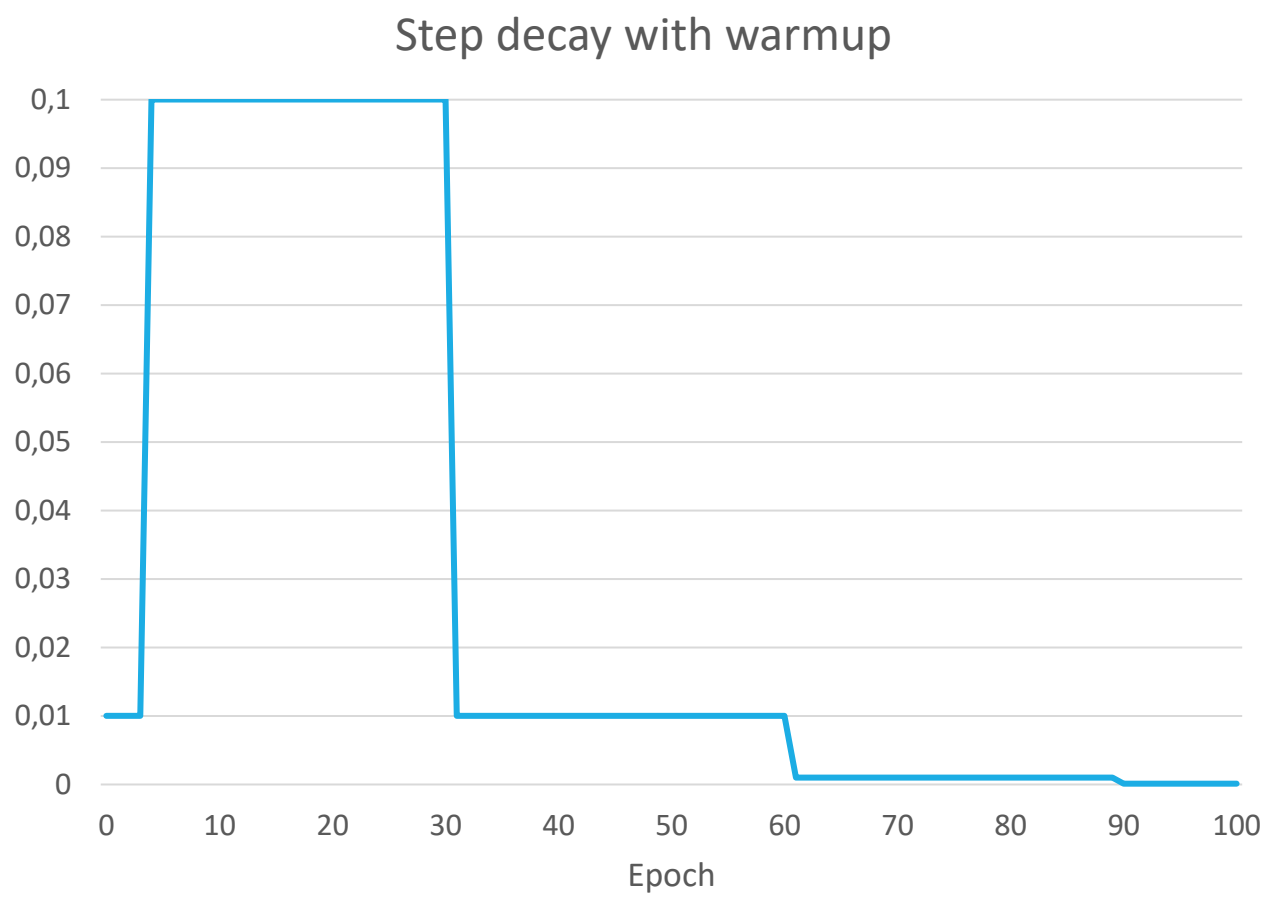
# Learning rate schedule: warm-up

best: start cautios, then try big jumps with high lr

starting with very high lr at the begininng gives instability but it helps to move out of the random init

Schedules usually start with high learning rates. For very deep networks (e.g. ResNet-110 on CIFAR-10) a high learning rate can slow down convergence at the beginning of training (i.e. accuracy remains at chance levels for several epochs). This is usually a symptom of poor initialization: a way to counteract it is to use a lower learning rate for a few epochs (even one or less, e.g. until accuracy increases).

Usually beneficial also for shallower, randomly initialized networks, as at the beginning of training we may be in a hard-to-navigate part of the loss landscape.




Kaiming He et al., "Deep Residual Learning for Image Recognition", CVPR 2016

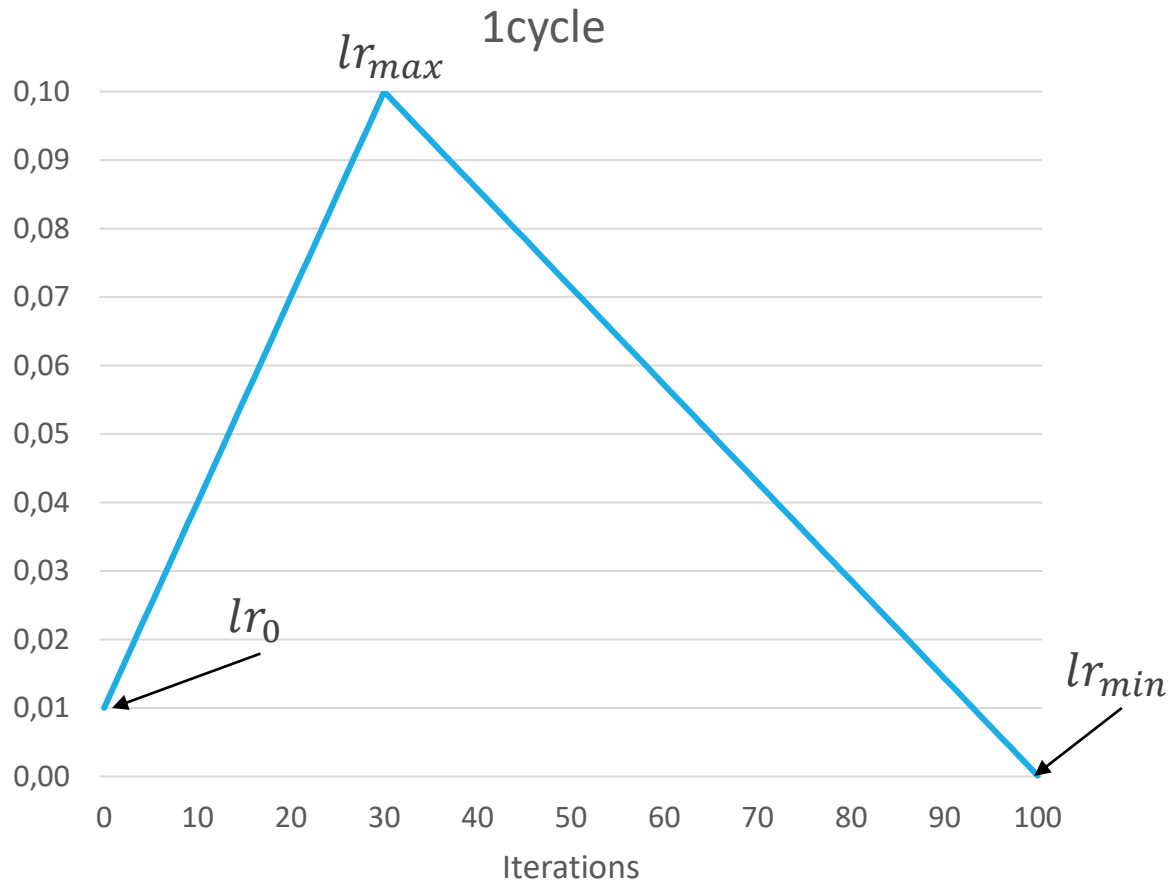
# Learning rate schedule: One cycle

it is different from the previous schedules because it is changed after each minibatch, not after each epoch

One cycle schedule (or decay) modifies the learning rate after each mini-batch (also referred to as iteration or step).
If training for a total number of iterations $I$, learning rate for each iteration $i$ is

$$lr_i = \begin{cases} lr_{max} + (lr_0 - lr_{max}) \left(1 - \frac{i}{pI}\right) & if\ i < pI \\ lr_{min} + (lr_0 - lr_{min}) \left(1 - \frac{i - pI}{I - pI}\right) & if\ i \geq pI \end{cases}$$

where $p$ is the fraction of iterations where we increase the learning rate (e.g. 0.3).
Original proposal had 3 phases, but PyTorch and other popular implementations provide two. It is usually realized with cosine segments.




Leslie N. Smith & Nicholay Topin, "Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates", ICLR 2017

# Regularization

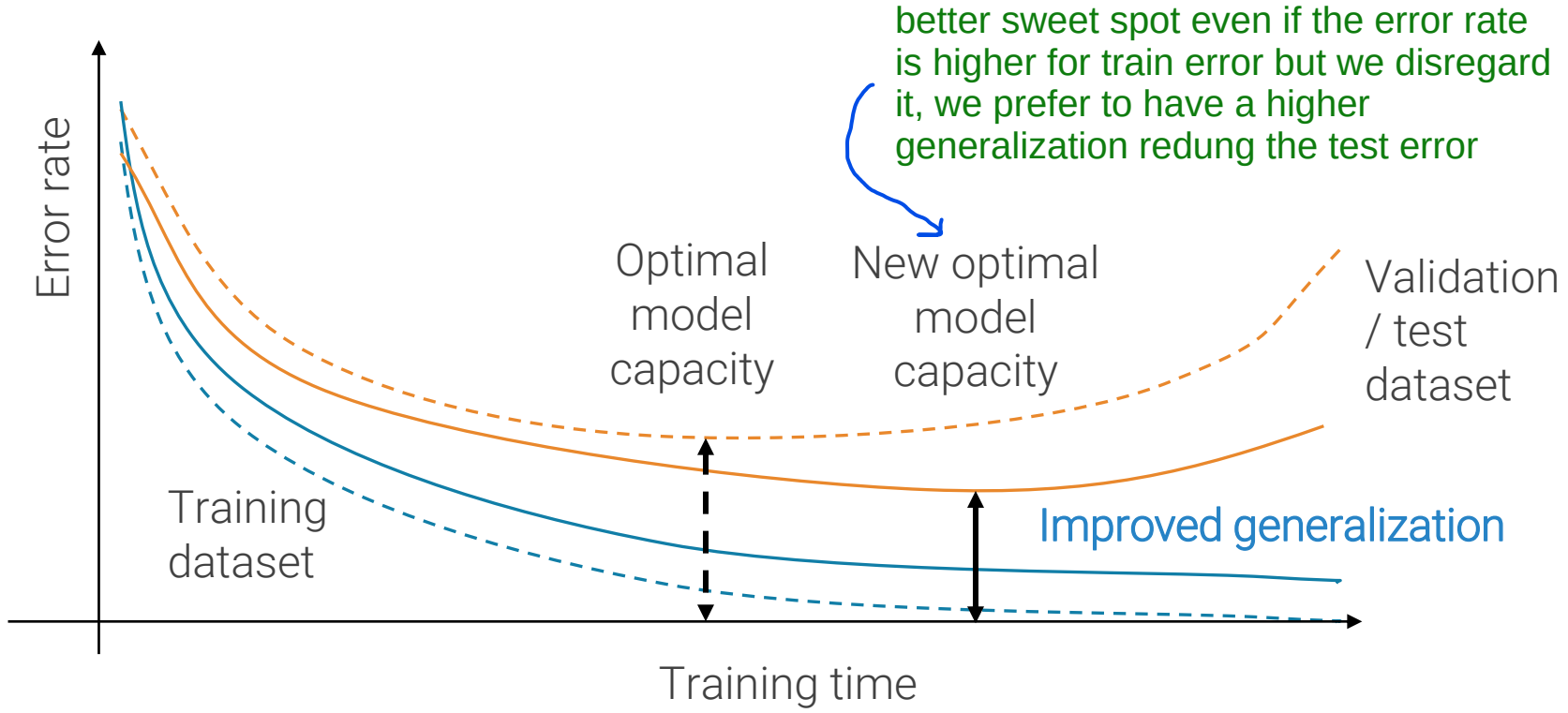
Regularization is any modification we make to the training recipe that is intended to

- reduce its generalization (test) error
- but not its training error.

regularization techniques

We will review:

- parameter norm penalties
- early stopping
- label smoothing
- dropout
- stochastic depth
- data augmentation



# Parameter norm penalties

impose constraints, penalizing large weights, such that the model does not become too complex and do not consider too much important training data alone. We dont want models bounded to specific high level structure of the training data, but instead we prefer simpler model, thus more general, that can behave better on unseen data, the test data

When using parameter norm penalties, we (implicitly!) add a term to the loss

$$L(\theta; D^{train}) = L^{task}(\theta; D^{train}) + \lambda\, L^{reg}(\theta)$$

Task (or data) loss: let training match the model to the input data and label, e.g. cross-entropy loss

Regularization loss: guide training to prefer "simpler" models

Commonly used norms are:

$\boldsymbol{l_2}$ regularization $L^{reg}(\boldsymbol{\theta}) = \sum_i \boldsymbol{\theta_i^2}$

$\boldsymbol{l_1}$ regularization $L^{reg}(\boldsymbol{\theta}) = \sum_i |\boldsymbol{\theta_i}|$

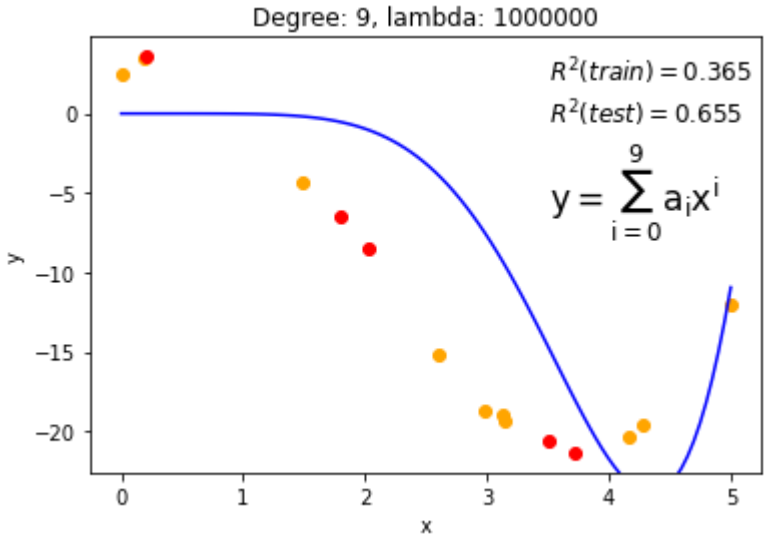
Intuition: even if a model has millions of parameters, a constraint on the overall norm of the parameters forces them to be small, thereby limiting the effective capacity of my model

# Example: generalized linear regression with $l_2$ norm

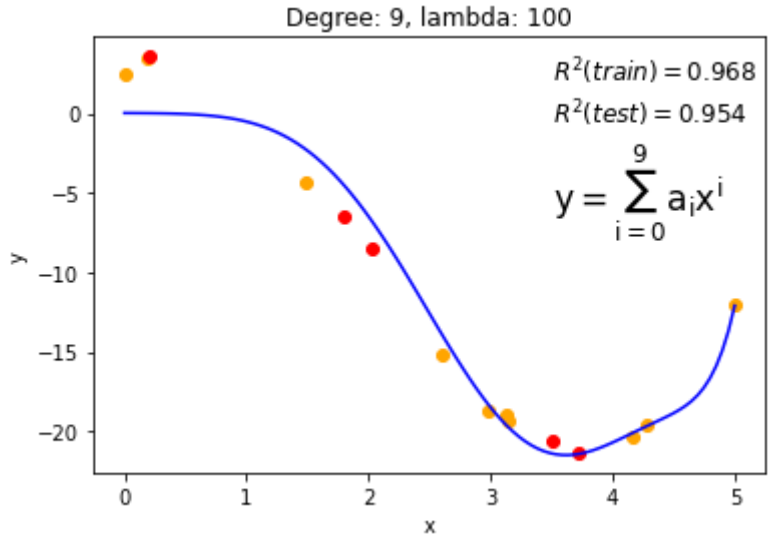
We can fit the same $N$ data points presented earlier with high-capacity models, i.e. polynomials of degree $N$, and control the ability of the model to generalize through appropriate regularization.

we are not adding nothing, so we obtain the same model of before

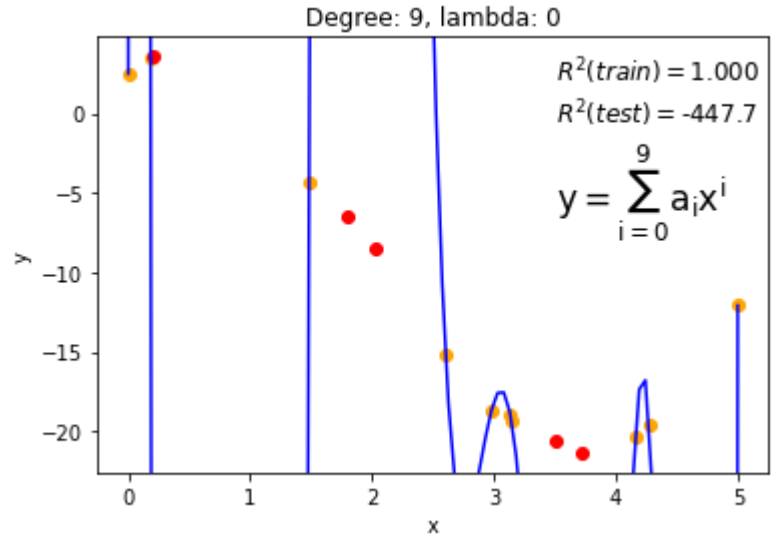
"Linear" model
$\lambda \to +\infty$




"Right" model
appropriate $\lambda$




"Interpolating" model
$\lambda \to 0$




http://www.deeplearningbook.org/contents/ml.html

# $l_2$ regularization or weight decay

our aim is then to weight "less" the params and to control the magnitude of the weights to prevent them from growing too large.

$$L\left(\theta; D^{train}\right) = L^{task}\left(\theta; D^{train}\right) + \frac{\lambda}{2}\|\theta\|_2^2$$

$l_2$ regularization is also known as ridge regression or Tikhonov regularization

In deep learning, it also goes under the name of weight decay because for plain SGD every gradient descent step drives the weights toward the origin before applying the update used when regularization is not used

$$\theta^{(i+1)} = \theta^{(i)} - lr\,\nabla_\theta L\left(\theta^{(i)}; D^{train}\right)$$

$$= \theta^{(i)} - lr\,\nabla_\theta \left[L^{task}\left(\theta^{(i)}; D^{train}\right) + \frac{\lambda}{2}\left\|\theta^{(i)}\right\|_2^2\right]$$

don't use theta_i as a starting point, but a number smaller than one mult by theta_i

$$= \theta^{(i)} - lr\left[\nabla_\theta L^{task}\left(\theta^{(i)}; D^{train}\right) + \lambda\theta^{(i)}\right]$$

$= (1 - lr\,\lambda)\theta^{(i)}$ Decayed parameter vector

$- lr\,\nabla_\theta L^{task}\left(\theta^{(i)}; D^{train}\right)$ Gradient without regularization

CLASS `torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0,`

$\lambda$ `weight_decay=0, nesterov=False, *, maximize=False, foreach=None)` [SOURCE]

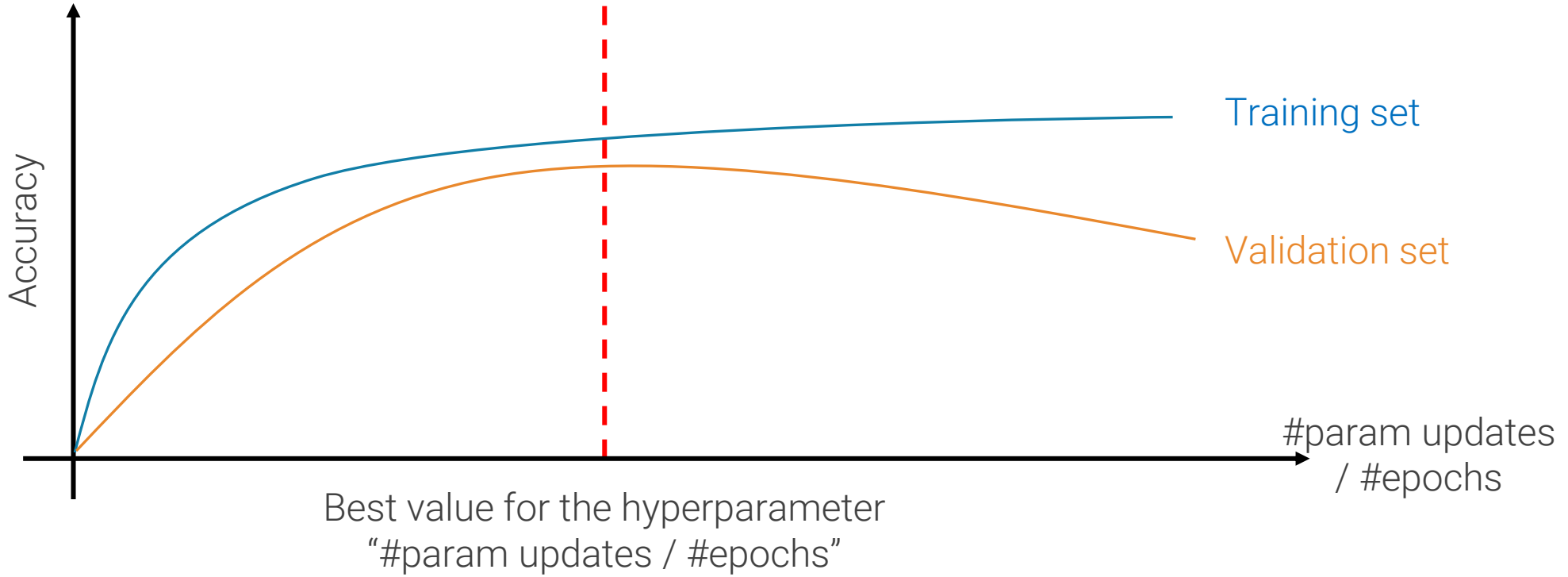
Stephen Jose Hanson and Lorien Y Pratt. "Comparing biases for minimal network construction with back-propagation.", NIPS 1988.

dont use loss cause its only a proxy, use metrics instead

# Early stopping

this or continue training until I do not see any further improvement of our metric of interest on validation set (patience hyperparam)

Training time (i.e. number of parameter updates) is an hyperparameter controlling the effective capacity of the model. By using, at inference time, the best performing model on the validation set, we are effectively selecting the best value for this hyperparameter.




http://www.deeplearningbook.org/contents/regularization.html

# Label smoothing

without label smoothing, we push scores of the correct class to +inf and the ones of the incorrect classes to -inf. But this approach does not bring any benefit because this lead to overfitting

we want that the arg of the log to be 1

$$s^{(i)} \rightarrow -\log\left(\frac{\exp\left(s_{y^{(i)}}\right)}{\sum_{k=1}^{C}\exp\left(s_k\right)}\right) \leftarrow \mathbb{I}\left(y^{(i)}\right)$$

$$LS\left(y^{(i)}, \epsilon\right)$$

| | |
|---|---|
| 0 (plane) | 2,1 |
| 1 (car) | 5,3 |
| 2 (bird) | 7,6 |
| ... | 0,2 |
| | -1,3 |
| | 2,5 |
| | 3,5 |
| | 4,5 |
| | 5,5 |
| | 6,5 |

Cross-entropy loss has the one-hot encoding of the true label as target. Yet, it can not reach it: to have loss equal to 0, softmax of the correct label should be 1, hence $s_{y^{(i)}} \rightarrow +\infty$ and $s_{k,k\neq y^{(i)}} \rightarrow -\infty$

Pushing scores in this way may cause overfitting

| | |
|---|---|
| 0 (plane) | 0 |
| 1 (car) | 0 |
| 2 (bird) | 1 |
| ... | 0 |
| | 0 |
| | 0 |
| | 0 |
| | 0 |
| | 0 |
| | 0 |

If we assume labels are corrupted by a small uniform noise $\epsilon$, we can mitigate the problem by using a label smoothing encoding.

This also accounts for mislabeled examples.

| | | |
|---|---|---|
| 0 (plane) | 0.01 | $= \epsilon/C$ |
| 1 (car) | 0.01 | $= \epsilon/C$ |
| 2 (bird) | 0.91 | $= 1 - \epsilon\,(C-1)/C$ |
| ... | 0.01 | $= \epsilon/C$ |
| | 0.01 | $= \epsilon/C$ |
| | 0.01 | $= \epsilon/C$ |
| | 0.01 | $= \epsilon/C$ |
| | 0.01 | $= \epsilon/C$ |
| | 0.01 | $= \epsilon/C$ |
| | 0.01 | $= \epsilon/C$ |

# Label Smoothing in PyTorch v1.10 and above

```
CLASS  torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-
       100, reduce=None, reduction='mean', label_smoothing=0.0)  [SOURCE]
```

Simply pass the correct class and set `label_smoothing` > 0

# Label Smoothing in PyTorch before v1.10

CLASS `torch.nn.CrossEntropyLoss(weight: Optional[torch.Tensor] = None, size_average=None, ignore_index: int = -100, reduce=None, reduction: str = 'mean')` [SOURCE]

This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.

It is useful when training a classification problem with *C* classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class.

*input* has to be a Tensor of size either $(\text{minibatch}, \text{C})$ or $(\text{minibatch}, \text{C}, \text{d}_1, \text{d}_2, ..., \text{d}_\text{K})$ with $\text{K} \geq 1$ for the *K*-dimensional case (described later).

This criterion expects a class index in the range $[0, \text{C} - 1]$ as the *target* for each value of a 1D tensor of size *minibatch*; if *ignore_index* is specified, this criterion also accepts this class index (this index may not necessarily be in the class range).

CE loss expects only the value of the correct class for each example in the mini-batch, we can't pass the label smoothing encoding of $y_i$ to it...

# Why cross-entropy?

The name cross-entropy loss comes from the cross-entropy $H(p, q)$, a distance used in information theory between two probability distributions $p$ and $q$ over the same set of events, defined as

$$H(p, q) = -\mathbb{E}_p[\log q]$$

If $p$ and $q$ are **discrete** probability distributions, this becomes

$$H(p, q) = -\sum_{x \in \mathbb{X}} p(x) \log q\,(x)$$

In classification, the set of events $\mathbb{X}$ is the set of the classes $1, \dots, C$, and a vector of $C$ positive values which sum to one is a probability mass function over this set.

Therefore, we can think of the one-hot encoding of the true label as the distribution $p$, and the output of the model after the softmax, as the distribution $q$. Then, we recover the expression of the standard loss

$$H\left(\mathbb{I}(y^{(i)}), p_{model}\left(Y \middle| x^{(i)}; \theta\right)\right) = -\sum_{k=1}^{C} \mathbb{I}\left(y^{(i)}\right)_k \log p_{model}\left(Y = k \middle| x^{(i)}; \theta\right) = -\log\, p_{model}\left(Y = y^{(i)} \middle| x^{(i)}; \theta\right)$$

# KLDiv Loss

With the interpretation of the loss as the cross-entropy, it is easy to change the one-hot encoding with the label smoothing encoding of the true label.

$$H\left(\boldsymbol{LS}\left(y^{(i)}, \boldsymbol{\epsilon}\right), p_{model}\left(Y \middle| x^{(i)}; \theta\right)\right) = -\sum_{k=1}^{C} \boldsymbol{LS}\left(y^{(i)}, \boldsymbol{\epsilon}\right)_k \log p_{model}\left(Y = k \middle| x^{(i)}; \theta\right)$$

One last step: cross-entropy can be expressed in terms of other two information-theory quantities, the Kullback-Leibler divergence $\boldsymbol{D_{KL}}$ and the entropy $\boldsymbol{H}$.

$$H(p, q) = H(p) + D_{KL}(p \,||\, q)$$

Entropy of the labels $H\left(\boldsymbol{LS}\left(y^{(i)}, \boldsymbol{\epsilon}\right)\right)$ is constant, hence minimizing CE is equivalent to minimizing KL divergence from the model $q$ to the labels $p$ (in this order, divergence is NOT commutative).

CLASS `torch.nn.KLDivLoss(size_average=None, reduce=None, reduction: str = 'mean', log_target: bool = False)`

[SOURCE]

The Kullback-Leibler divergence Loss

`target` is $p$, i.e. $\boldsymbol{LS}\left(y^{(i)}, \boldsymbol{\epsilon}\right)$,
`input` is $\log(q)$, i.e. $\log\left(p_{model}\left(Y = k \middle| x^{(i)}; \theta\right)\right)$

This criterion expects a *target Tensor* of the same size as the *input Tensor*.

# Dropout

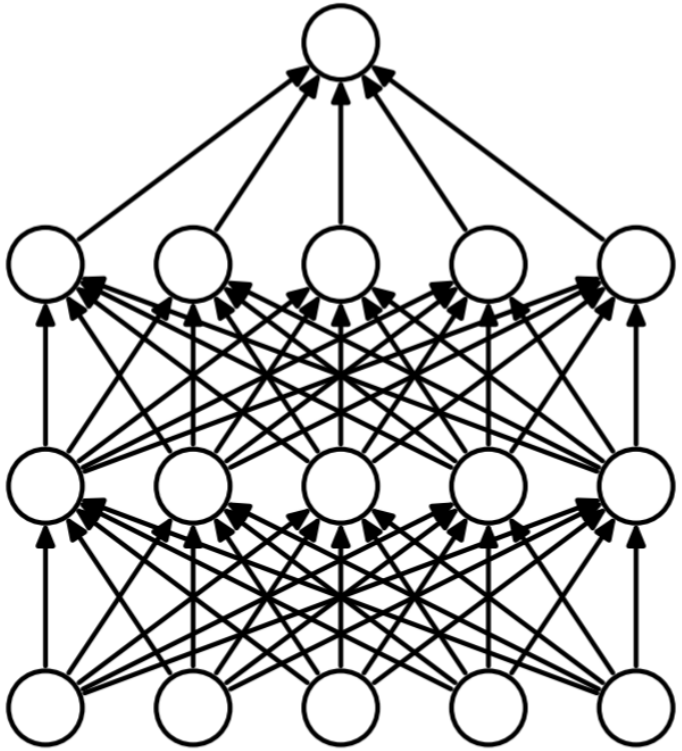
cancelling activations

In each forward pass, randomly set some activations to zero

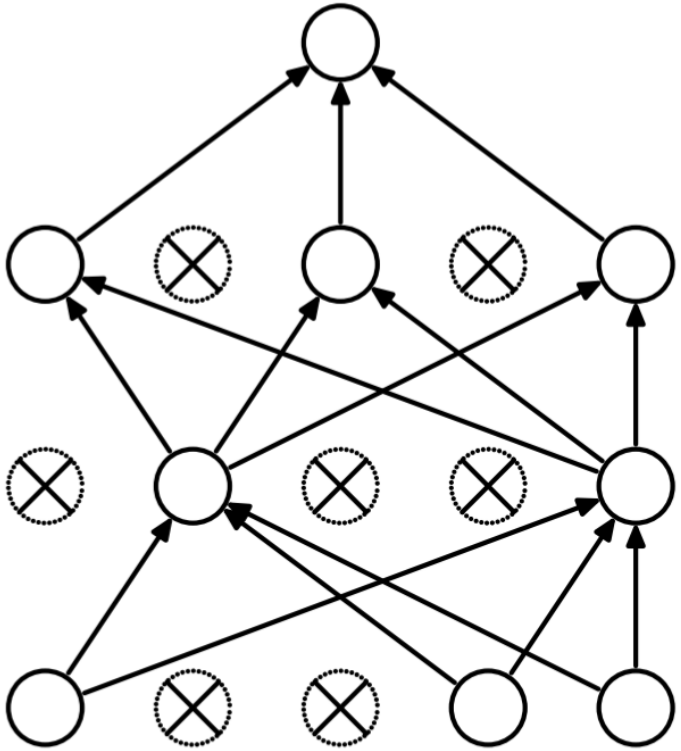
Probability of keeping an activation is a hyperparameter $p$, usually set to 0.5 and as high as 0.8 or 0.9.

Every time we train on a new minibatch of examples, we sample a new binary mask of active nodes.

We may apply it between any two layers, but never to the output activations!



Full network



Network used for one forward pass

Nitish Srivastava et al., "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2015.

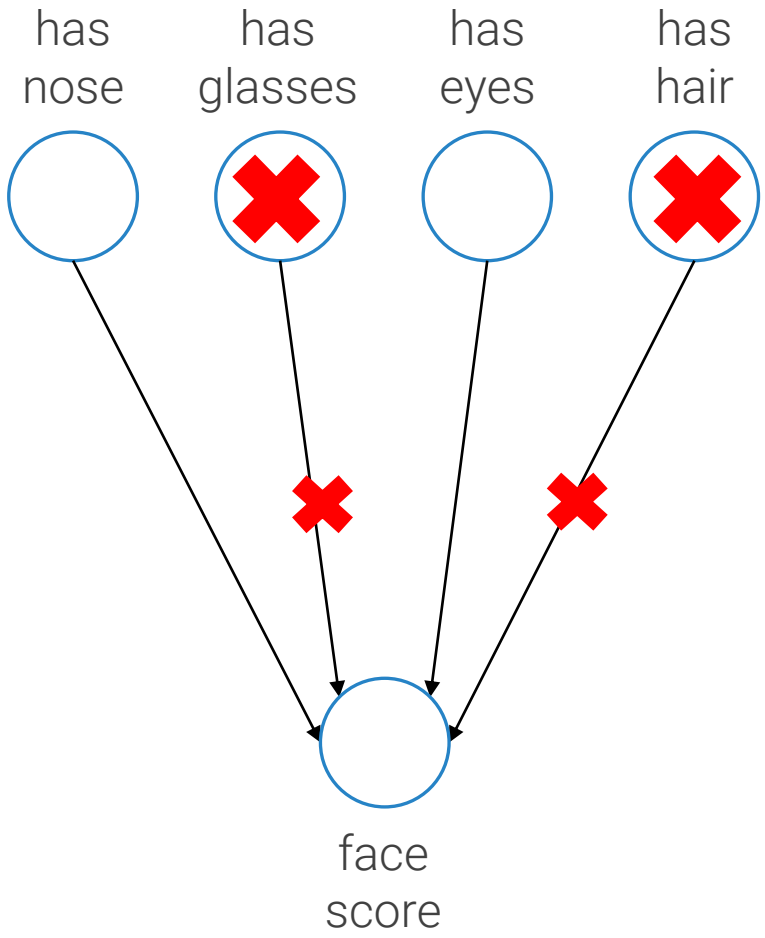
# Dropout as anti-conspiracy strategy

The idea of dropout was inspired to Hinton by his bank.

“I went to my bank. The tellers kept changing and I asked one of them why. He said he didn’t know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting.”

Each hidden unit must be able to perform well regardless of which other hidden units are in the model, it regularizes units to be useful in many context.

As disruptive noise is applied to hidden units even in deep layers, makes it able to destroy selective information from the input, e.g. erases noses from faces and ask the network to recognize them anyway.



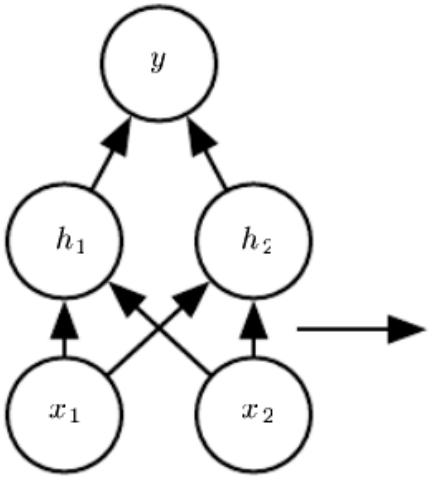
# Alternative interpretation: ensemble

We can think of dropout as training a large ensemble of models: each random binary mask is a new “model”.

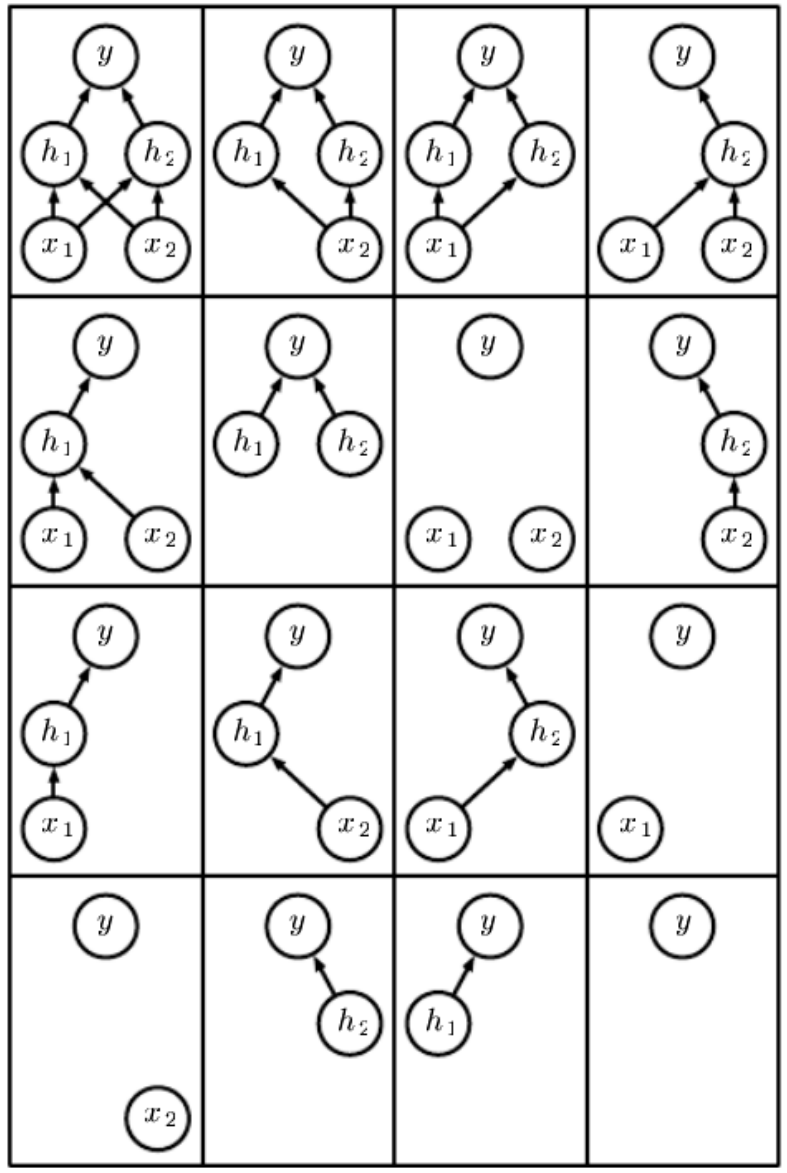
Ensemble of models are a widely used technique to improve performance.

Key difference: “models” in dropout share weights

We can think of this ensemble as a special form of bagging, which we will study when discussing Random Forests



Base network



Ensemble of subnetworks

NOTE: configurations without connections between input and output highly unlikely in large models

http://www.deeplearningbook.org/contents/regularization.html

# Dropout – Test time

we've introduced stochasticity: at training time a random mask is sampled and it changes scores -> how to use a network that works stochastically?

Dropout makes predictions at training time stochastic, but we want output at test time to be deterministic

$$scores = f(x; \theta, m)$$

Random mask

Principled solution: "average out" stochasticity at test time

we sample all the possible binary masks, run the networks with each of them with our image and then average out each prediction

$$scores = f(x; \theta) = \mathbb{E}_m[f(x; \theta, m)] = \sum_{all\ masks\ m} p(m) f(x; \theta, m)$$

We can approximate the sum, but everything is slowed down: e.g. for each test example $x$, pass it through the network several times sampling a different mask at each step (the more the better) and then average predictions.

but it is too costly, then we do such an approximation -->

# Dropout – Weight scaling

Batch norm showed us how dangerous is to have discrepancies between distributions inside the network between training time and test time

A faster approximation is provided by weight scaling

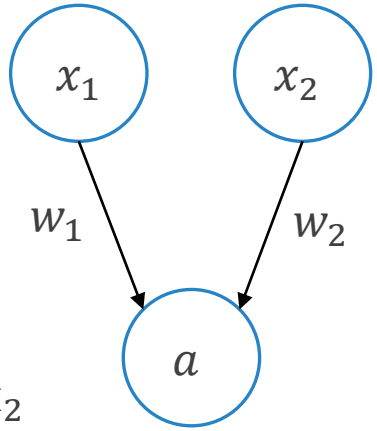
Consider an activation obtained by merging two linear neurons: in this case the summation can be solved explicitly, and it shows that at training time we obtain on average the same activation without dropping rescaled by $p$

To make training and test time equal, we can either:

1. Rescale value at test time by $p \rightarrow p\ a_{test} = \mathbb{E}_m[a_{train}]$
2. Rescale value at training time by $\frac{1}{p} \rightarrow a_{test} = \mathbb{E}_m\left[\frac{a_{train}}{p}\right]$

The latter is referred to as inverted dropout and is preferred as it leaves test time unchanged.

While this is exact only for linear layers, it is used as a fast approximation also in the presence of non-linearities.




Value at test time (without dropping):

$a_{test} = \mathrm{w}_1\mathrm{x}_1 + \mathrm{w}_2\mathrm{x}_2$

a is a RV

Expected value at training time with $p = 0.5$:

$$\mathbb{E}_m[a_{train}]$$
$$= \frac{1}{4}(w_1x_1 + w_2x_2) + \frac{1}{4}(w_1x_1 + 0)$$
$$+ \frac{1}{4}(0 + w_2x_2) + \frac{1}{4}(0 + 0)$$
$$= \frac{1}{2}(\mathrm{w}_1\mathrm{x}_1 + \mathrm{w}_2\mathrm{x}_2) = p\ a_{test}$$

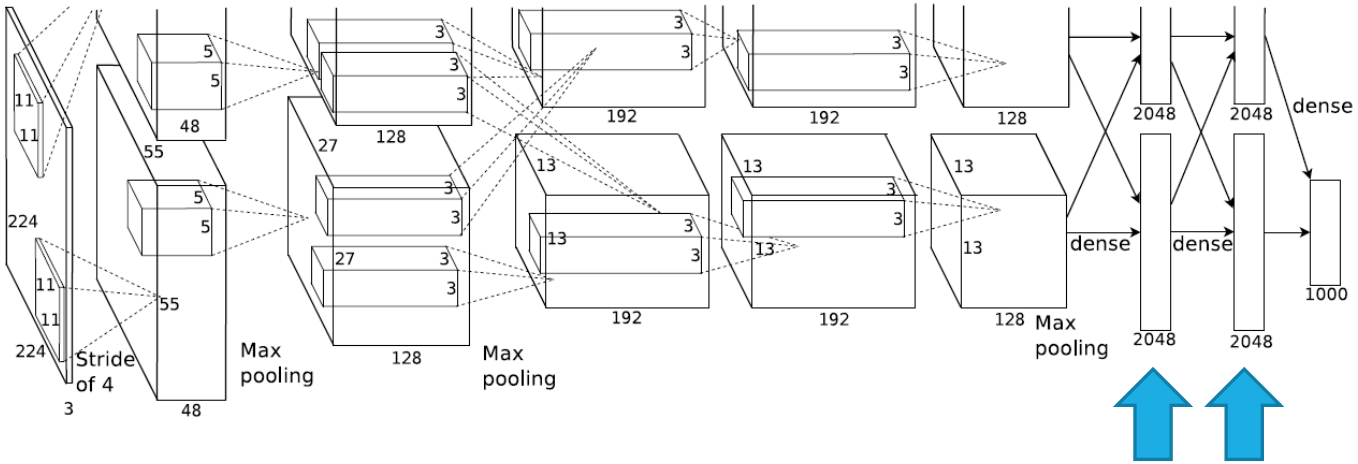
obv, we consider all cases: x1 dropped, x2 dropped, both of them, none of 'em

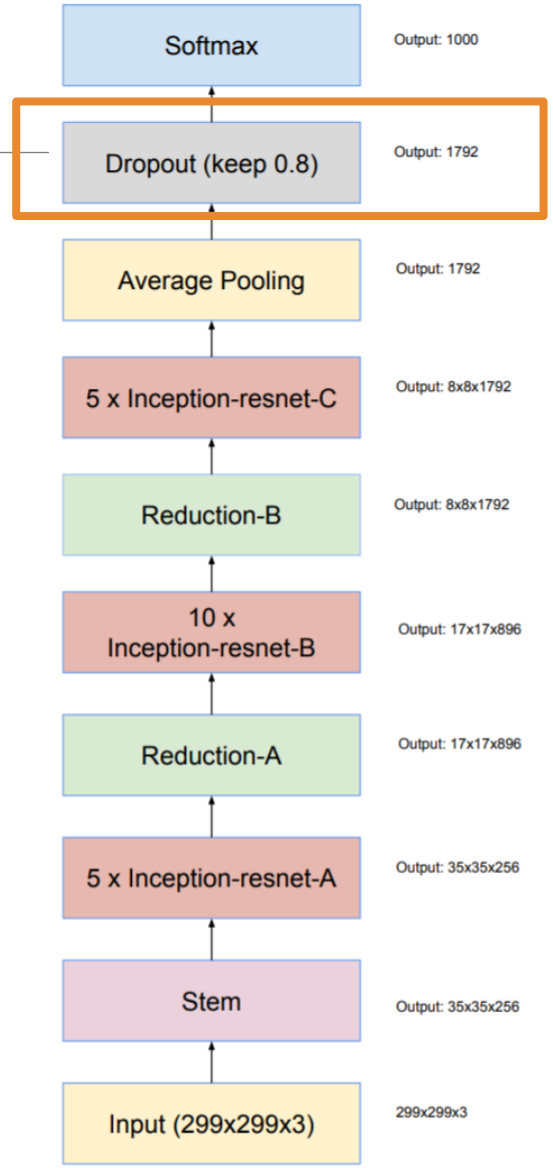
p factor of discrepancy between excepted value at training time and the value at test time, without dropout

don't use between conv because they have already strong inductive bias and then few params

# Dropout: where

use them where there are a lot of parameters




Usually applied to layers with many parameters, e.g. all but last FC layers in AlexNet, VGG, between GlobalAveragePooling and last fc layer in Inception-ResNet-v2,…

Not used in ResNets and variants, for its unclear interactions with BatchNorm.

EfficientNet uses it again, and scales $p$ linearly from 0.2 to 0.5 going from B0 to B7.

Xiang Li, Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift, CVPR 19

# Regularization: a general template

Training time: add randomness

making train difficult, increasing training loss

$$scores = f(x; \theta, m)$$

Test time: average it out ( sometimes approximately )

$$f(x; \theta) = \sum_{all\ masks\ m} p(m) f(x; \theta, m)$$

Example: batch norm

batch norm is actually a regularization technique, because it follows the same template above

At training time, each activation is randomly influenced by other samples in the mini-batch

$$\hat{a}_j^{(i)} = \frac{a_j^{(i)} - \mu_j}{\sqrt{v_j + \epsilon}}$$

$\mu_j$ and $v_j$ fixed to averages seen at training time, BN becomes a deterministic linear operation

# Stochastic depth

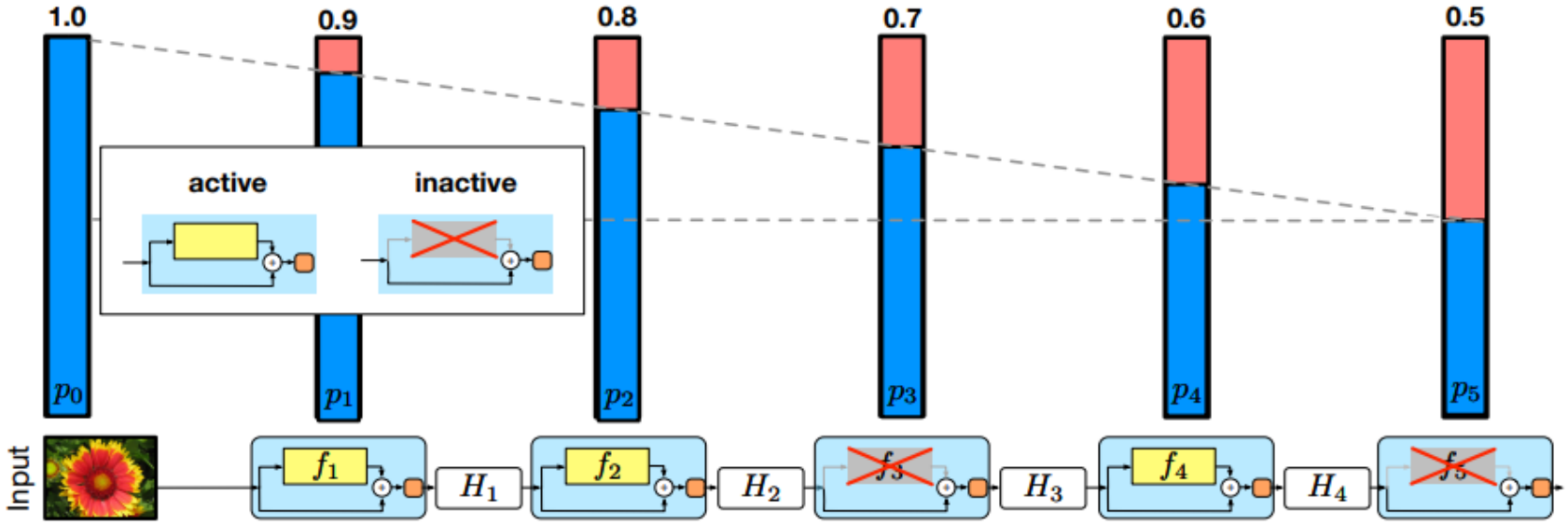
skip it

Training time: shrink depth

For ResNet-like architectures, drop with probability $1 - p_l$ the convolutional path of a ResNet-block, i.e. keep active only the residual identity path.

The survival probability decreases with depth

$$p_l = 1 - \frac{l}{L}(1 - p_L)$$




Test time: unmodified network

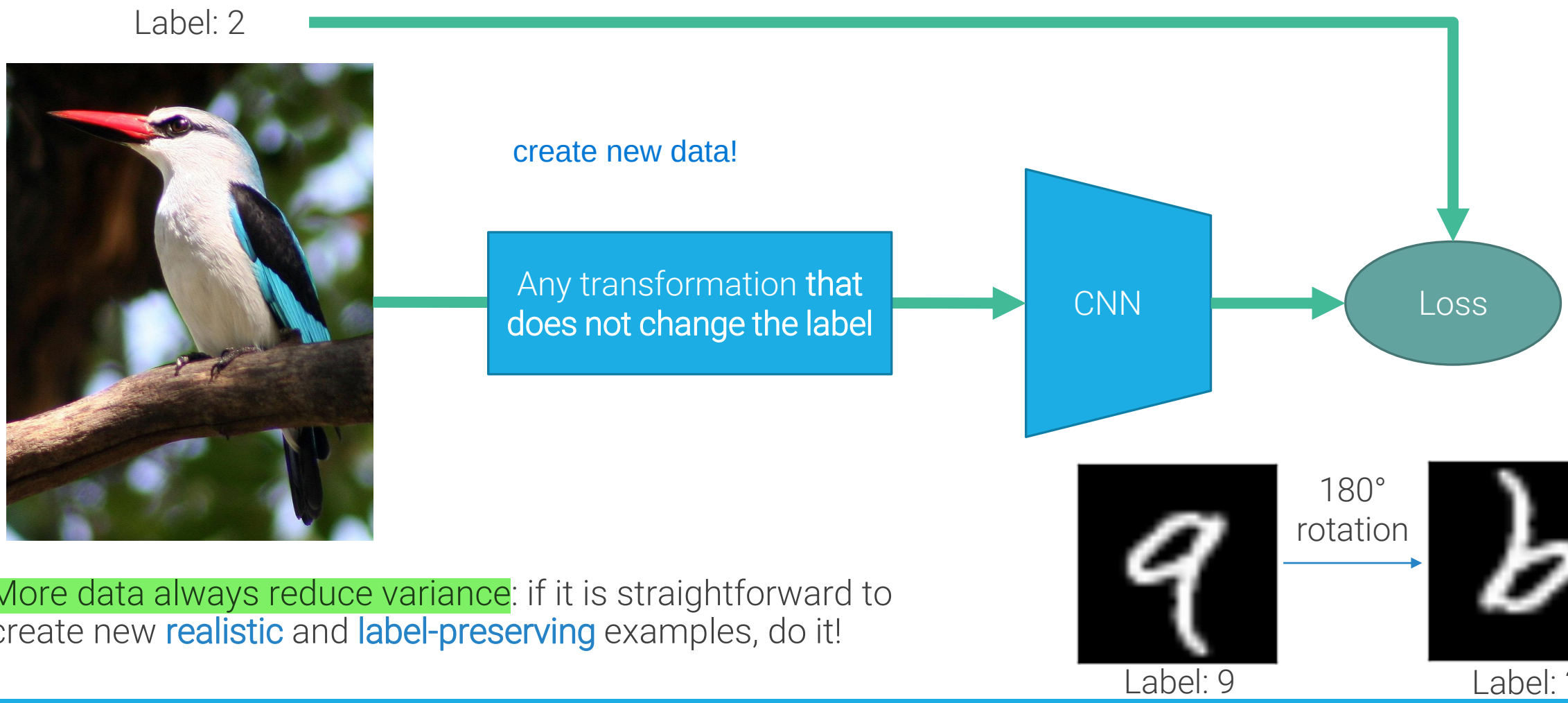
Huang et al., "Deep Networks with Stochastic Depth", ECCV16

super hard to augment meaningfully

always visually check the augmented data

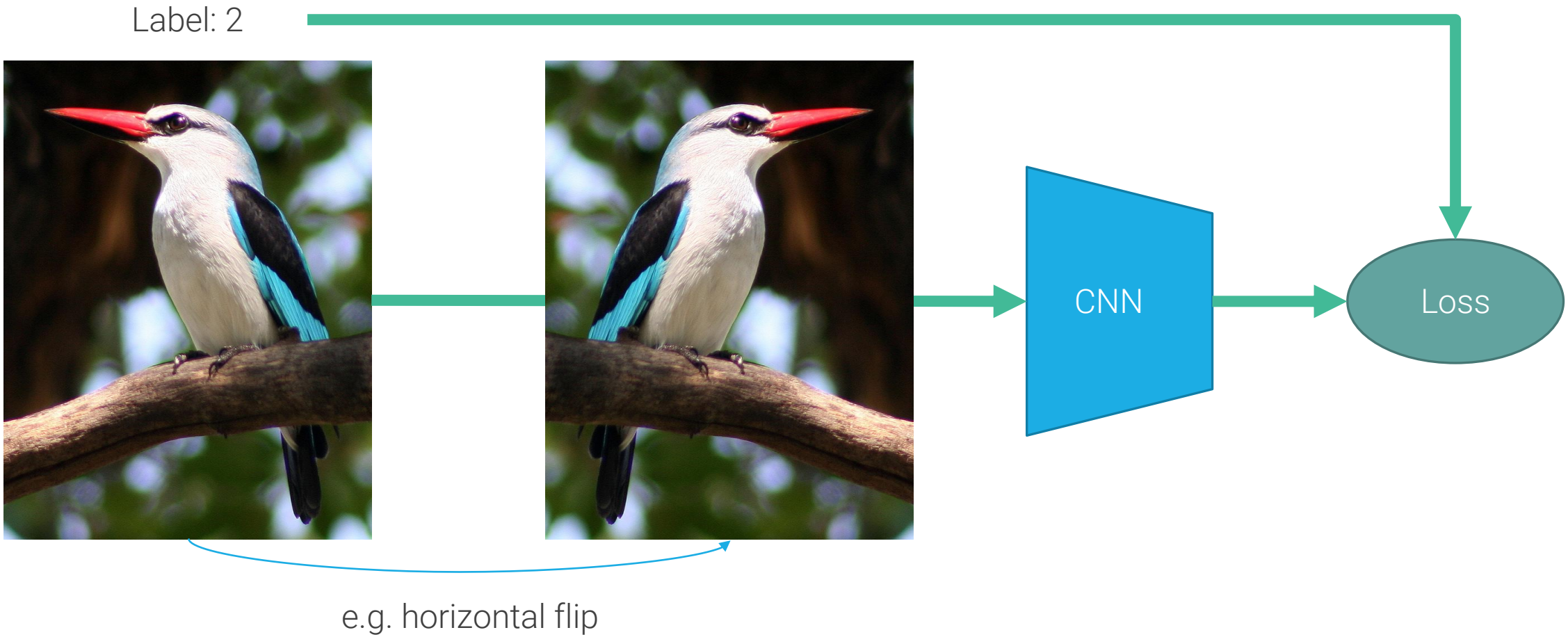
# Data augmentation

the most important regularization technique to try when overfitting




More data always reduce variance: if it is straightforward to create new realistic and label-preserving examples, do it!

# Data augmentation: example



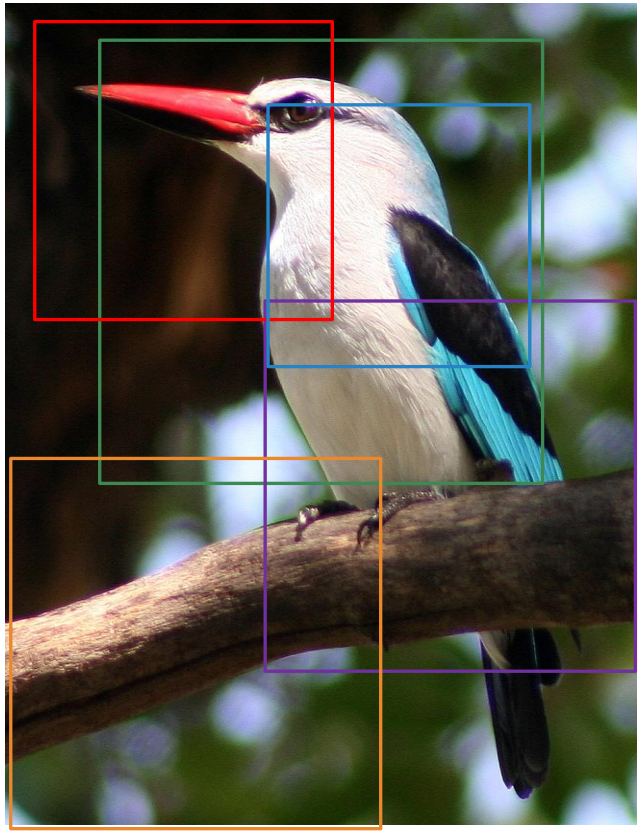
# Data augmentation: multi-scale training

Employed by ResNet, inherited from VGG

Training: sample random crops / scales

1. Pick random $S$ in range $[S_{\min}, S_{\max}] = [256, 480]$
2. Isotropically resize training image so that short side = $S$
3. Sample random 224 x 224 patch

For smaller $S$, similar to 224, the crop will capture whole-image statistics, while for $S \gg 224$ the crop will correspond to a small part of the image, containing a small object or an object part.

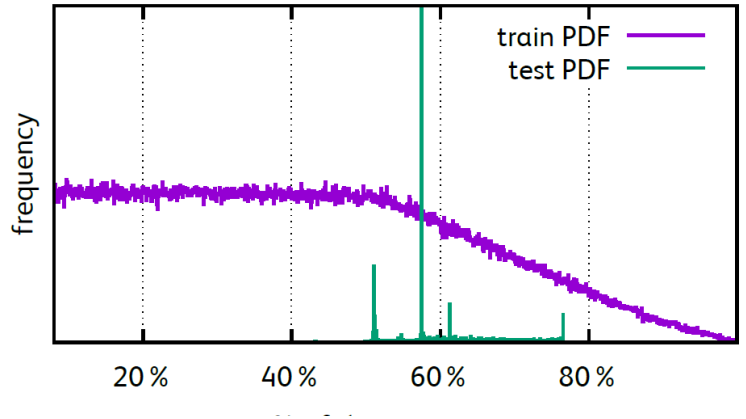


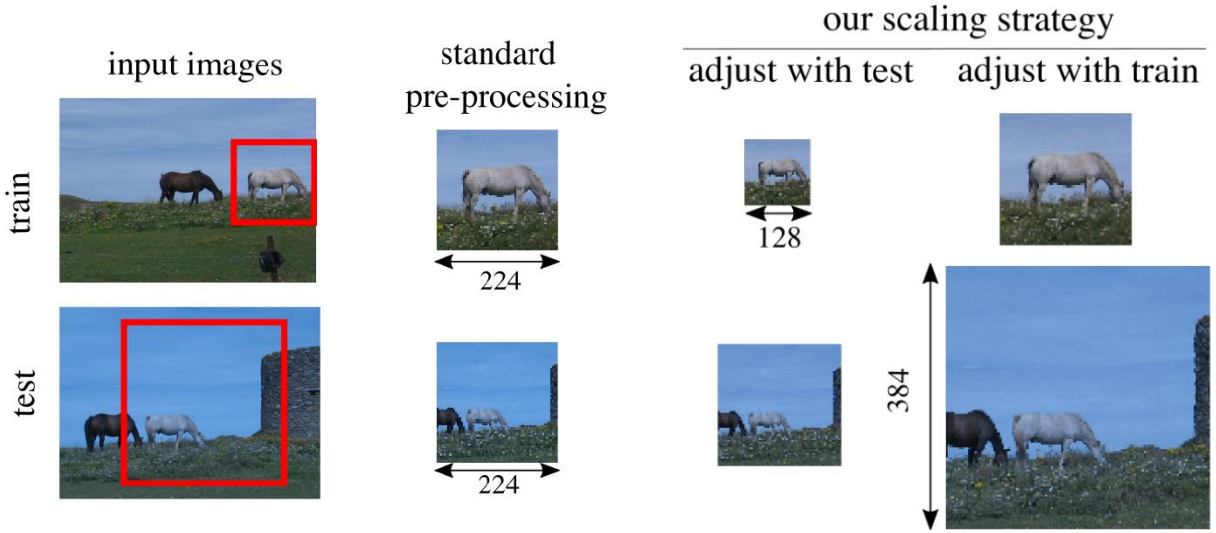
# FixRes

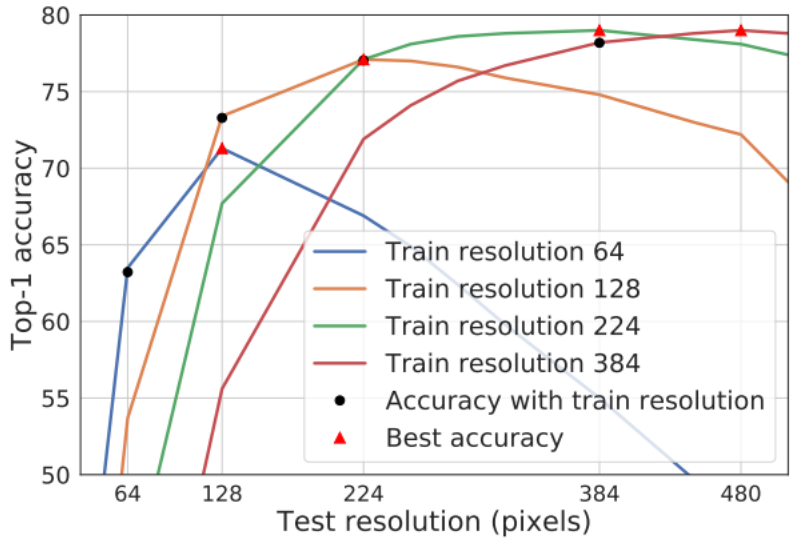
if train at lower res and test at higher resolution better performances!!

Data augmentations induce a significant discrepancy between the size of the objects seen by the classifier at train and test time: in fact, a lower train resolution improves the classification at test time!

FixRes is a simple strategy to optimize the classifier performance, that fix the discrepancy by: (a) calibrating the object sizes by adjusting the crop size and (b) adjusting statistics of global average pooling by fine-tuning on larger crops the classifier and the last batch-norm layer before pooling.



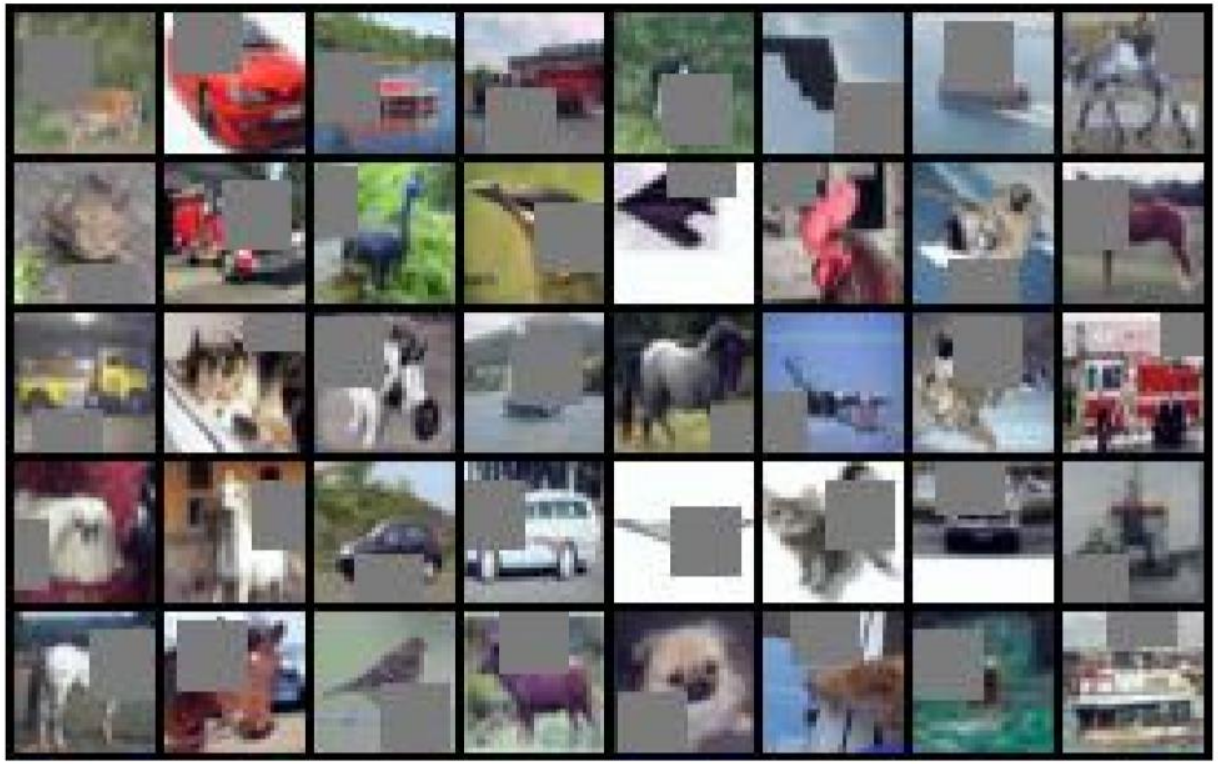



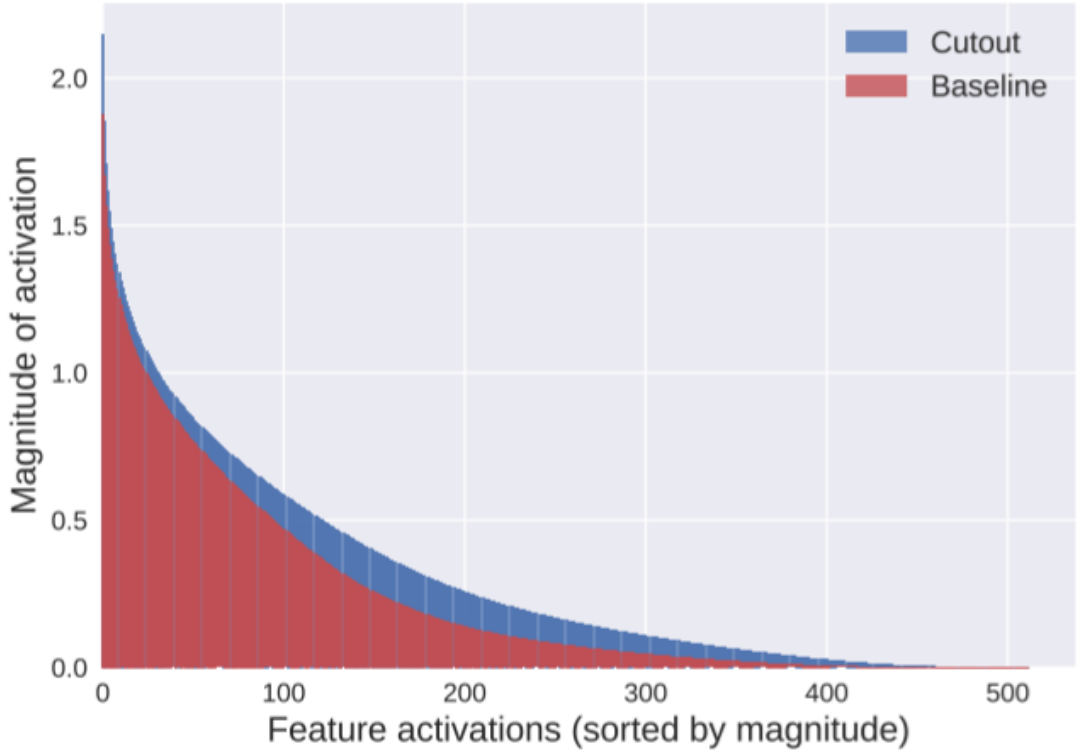

Hugo Touvron et al., "Fixing the train-test resolution discrepancy", NeurIPS 2019.

# Cutout (Random Erasing)

Remove a random square region of the input image with 50% probability = “Dropout of the input space”.

It forces the network to use a more diverse set of features, improving generalization.






Terrance DeVries and Graham W. Taylor, Improved Regularization of Convolutional Neural Networks with Cutout, 2017

Zhun Zhong et al., “Random Erasing Data Augmentation”, AAAI 2020.

# Mixup

Training: random blending of images

Given two samples $(x^{(i)}, y^{(i)}), (x^{(j)}, y^{(j)})$

1. Sample random blend probability $\lambda$ from a PDF $Beta(\alpha, \alpha),\ \alpha \in (0, +\infty)$

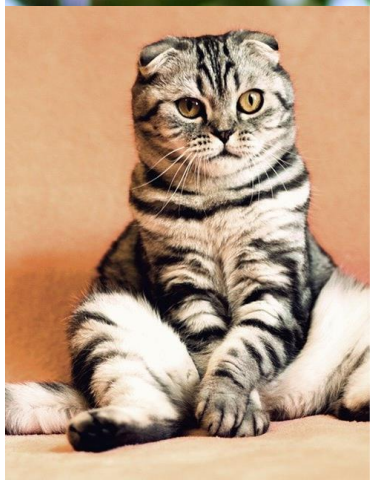
2. Perform linear interpolation in the input space **and in the one-hot encoded label space** to define actual input sample and label

$$x = \lambda\, x^{(i)} + (1 - \lambda)\, x^{(j)}$$

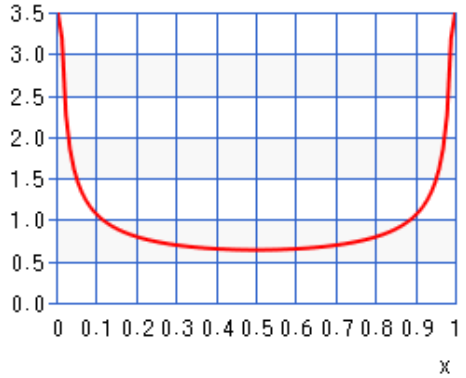
$$y = \lambda\, \mathbb{I}(y^{(i)}) + (1 - \lambda)\, \mathbb{I}(y^{(j)})$$

Testing: unmodified input

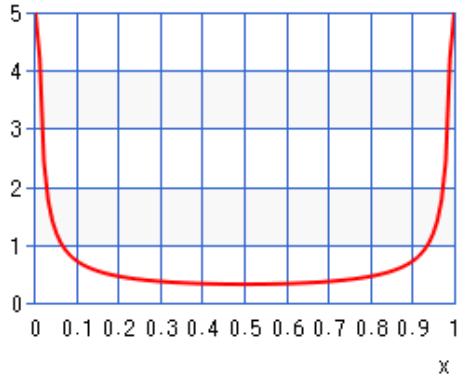
Intuition: it forces the network to act linearly in between classes and have smoother transitions from one class decision boundary to another. Both effects help generalization.

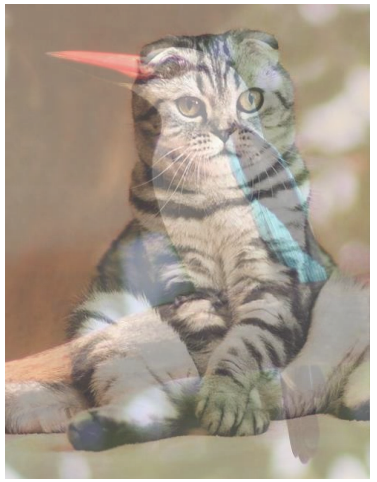


$Beta(0.5, 0.5)$




$Beta(0.2, 0.2)$



| | |
|---|---|
| 0 (plane) | 0 |
| 1 (cat) | 0.6 |
| 2 (bird) | 0.4 |
| … | 0 |
| | 0 |

Hongyi Zhang et al., "mixup: BEYOND EMPIRICAL RISK MINIMIZATION", ICLR 2018.

# CutMix

Training: patches are cut and pasted among training images

Given two samples $(x^{(i)}, y^{(i)}), (x^{(j)}, y^{(j)})$ in a mini-batch

1. Sample random blend probability $\lambda$ from a PDF $Beta(\alpha, \alpha),\ \alpha \in (0, +\infty)$

2. Sample a rectangular region $(r_x, r_y, r_h, r_w)$ whose aspect ratio is proportional to the original image and use it to define a binary mask $M$ filled with 0 inside the region, 1 outside.

$$r_x \sim U(0, W)\ r_y \sim U(0, H)\ r_w = W\sqrt{1-\lambda}\ \ r_h = H\sqrt{1-\lambda}$$

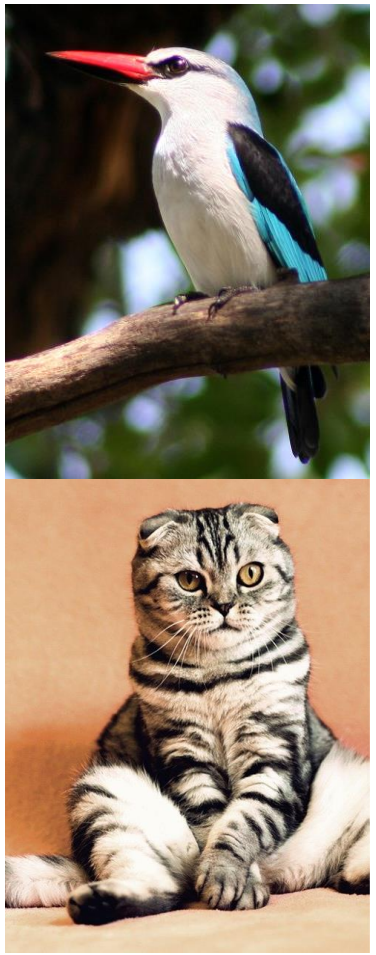
Note that the ratios of areas between region and image is $\frac{r_w r_h}{WH} = 1 - \lambda$

3.Combine images according to mask and perform linear interpolation **in the one-hot encoded label space** to define the label

$$x = M \odot x^{(i)} + (1 - M) \odot x^{(j)}$$

$$y = \lambda\, \mathbb{I}(y^{(i)}) + (1 - \lambda)\, \mathbb{I}(y^{(j)})$$

Testing: unmodified input



| | |
|---|---|
| 0 (plane) | 0 |
| 1 (cat) | 0.8 |
| 2 (bird) | 0.2 |
| ... | 0 |
| | 0 |

Sangdoo Yun et al., "CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features", ICCV 19

# Random Hyper-parameters search

Grid search (log-linear):

$\forall\ lr \in 10\text{^}\{-5, -4, -3, -2\}$

$\forall\ wd \in 10\text{^}\{-5, -4, -3, -2\}$

train a model

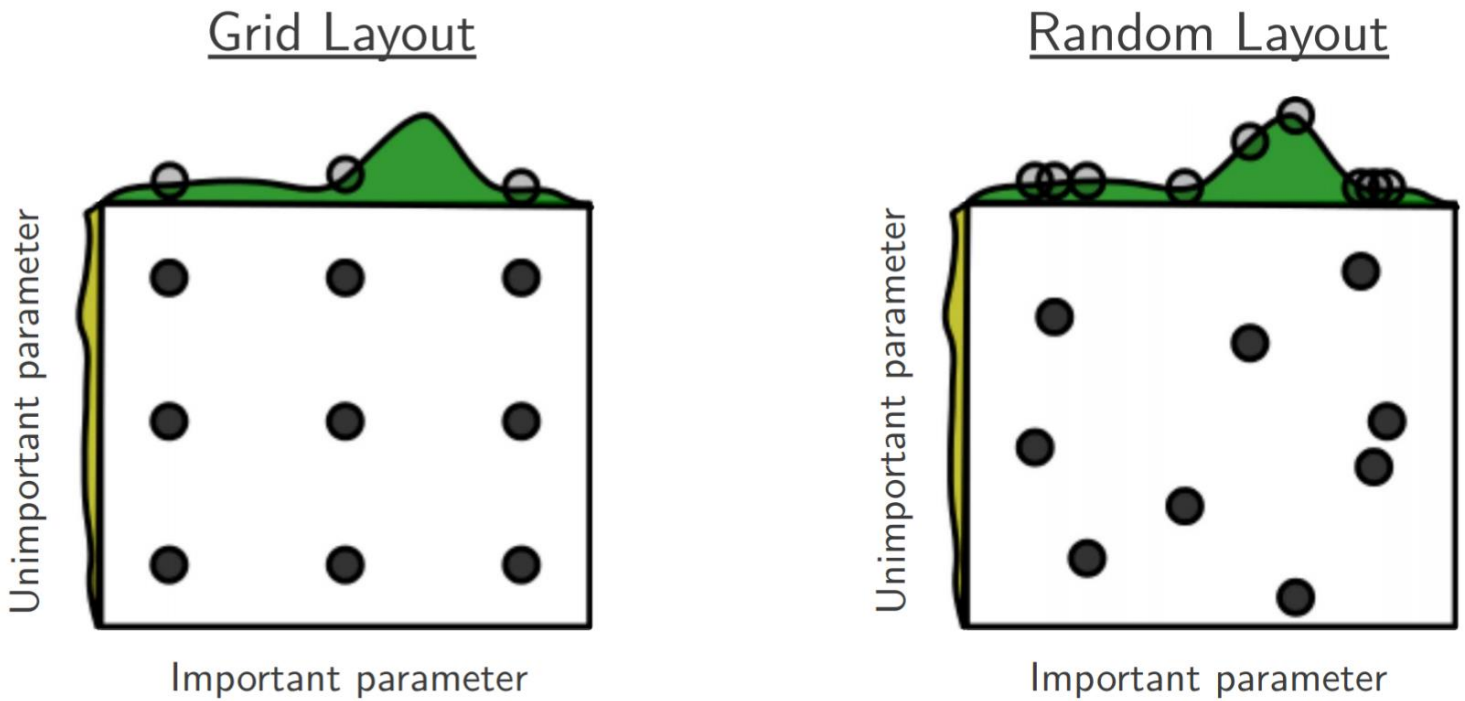
Random search: sample

$lr$ from $10\text{^}U[-5, -2]$

$wd$ from $10\text{^}U[-5, -2]$

and train a model.

It leads to a more efficient exploration of the space (but beware of the curse of dimensionality!)




Bergstra and Bengio, "Random Search for Hyper-Parameter Optimization", JMLR 2012

# Test time good practice: ensembles

1. Train multiple (randomly initialized) models on the same dataset.

2. Run each model over a test image, average the results (e.g. take average of logits, then take argmax)

This usually increases the overall performance by 1-2%: even if networks have similar error rates, they tend to make different mistakes.

Downside 1: we have to train a lot of networks from scratch

Downside 2: we have to run a lot of networks at test-time

# Exponential Moving Average (EMA)

To avoid building and running a costly ensemble, we can average snapshots in weight space.

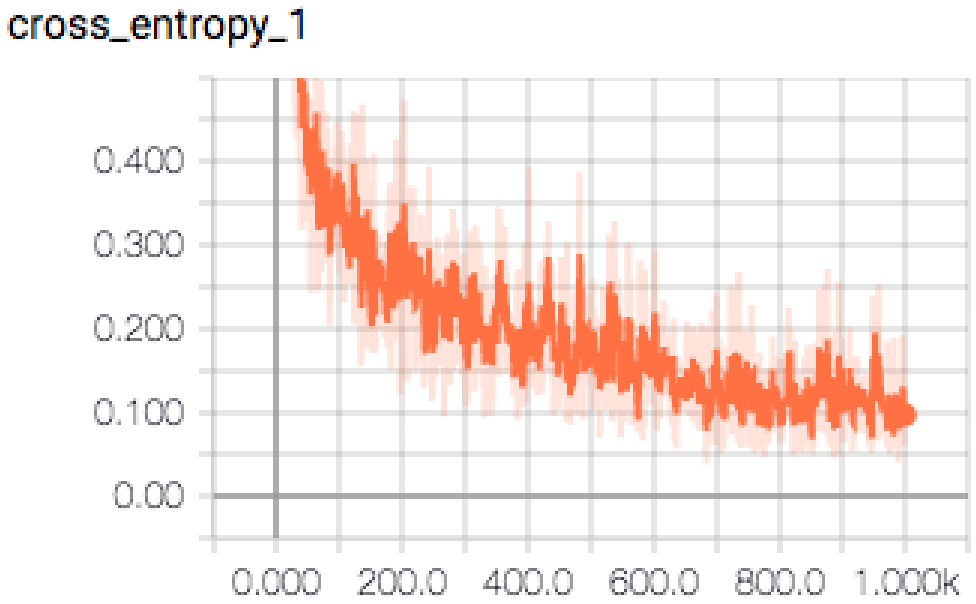
EMA (sometimes called Polyak average): store and update with SGD the usual vector of parameters used at training time $\theta$

$$\theta^{(i+1)} = \theta^{(i)} - lr\ \nabla_\theta L\left(\theta; D^{(train)}\right)$$

but also store a vector of parameters to be used at test time, updated with an exponential moving average every $k$ step

$$\theta^{(test)} = (1-\rho)\ \theta^{(i+1)} + \rho\ \theta^{(test)} \quad \rho \in [0,1]$$

($\rho$ usually very large)




Polyak and Juditsky, "Acceleration of stochastic approximation by averaging", SIAM Journal on Control and Optimization, 1992.

# A “recipe” for training NNs

“… suffering is a perfectly natural part of getting a neural network to work well, but it can be mitigated by being thorough, defensive, paranoid, and obsessed with visualizations of basically every possible thing. The qualities that in my experience correlate most strongly to success in deep learning are patience and attention to detail.”

1. **Become one with the data**: collect statistics but also look at the data, understand what is relevant
2. **Set up the end-to-end training/evaluation skeleton** + get dumb baselines: check all the infrastructure code before training complex models, check init loss, overfit a small dataset, etc..
3. Overfit: reach low bias by starting with known models + Adam, then explore
4. Regularize: apply data augmentation, norm penalties, dropout, stochastic depth, cutmix, etc..
5. **Tune**: random search for better hyper parameters around what worked in 3-4, use LR schedules
6. **Test-time optimizations**: ensembles and/or distillation

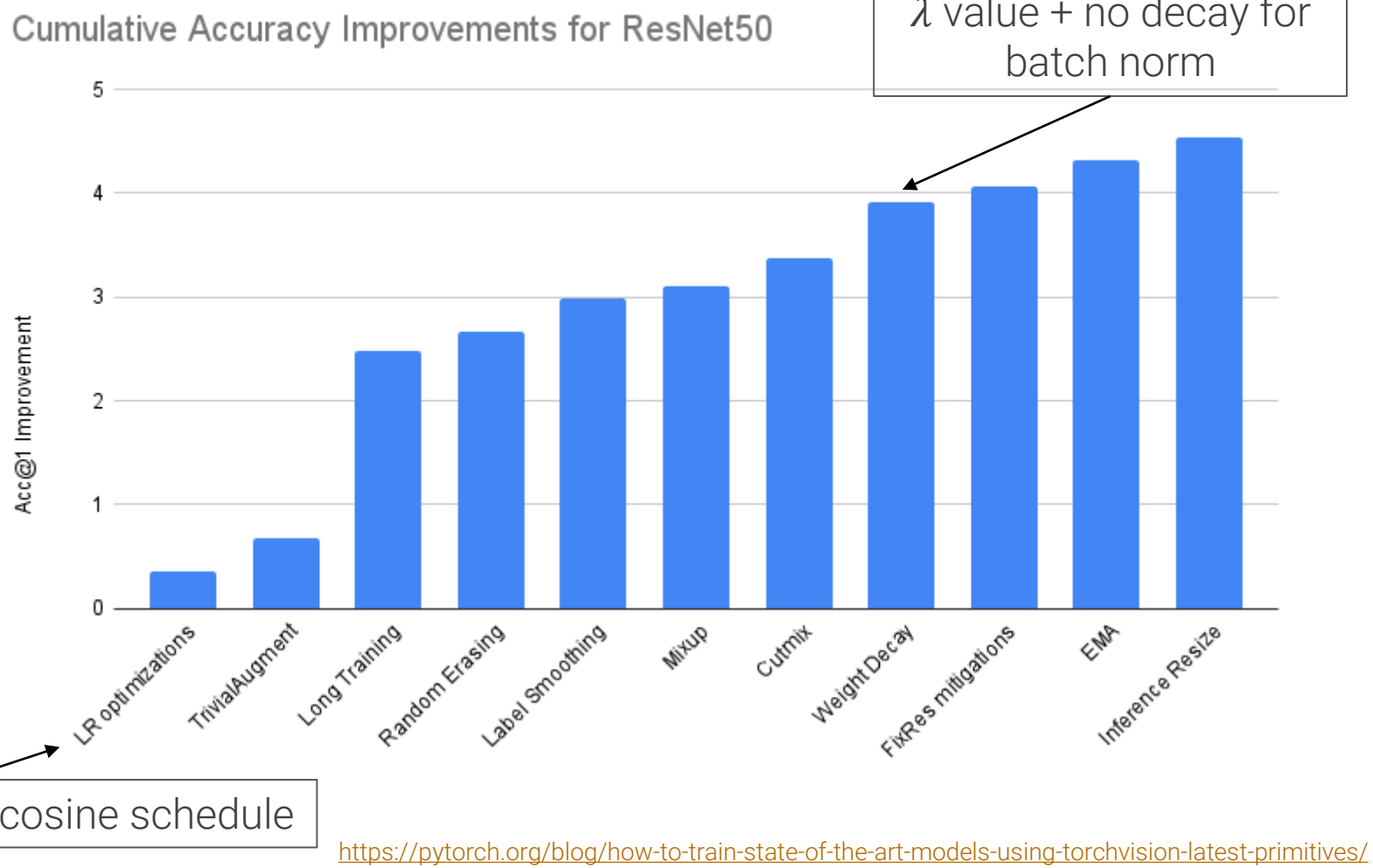
it is a sanity check, it means that the model can at least perform well on training data

https://karpathy.github.io/2019/04/25/recipe/

# A case study: new weights in PyTorch

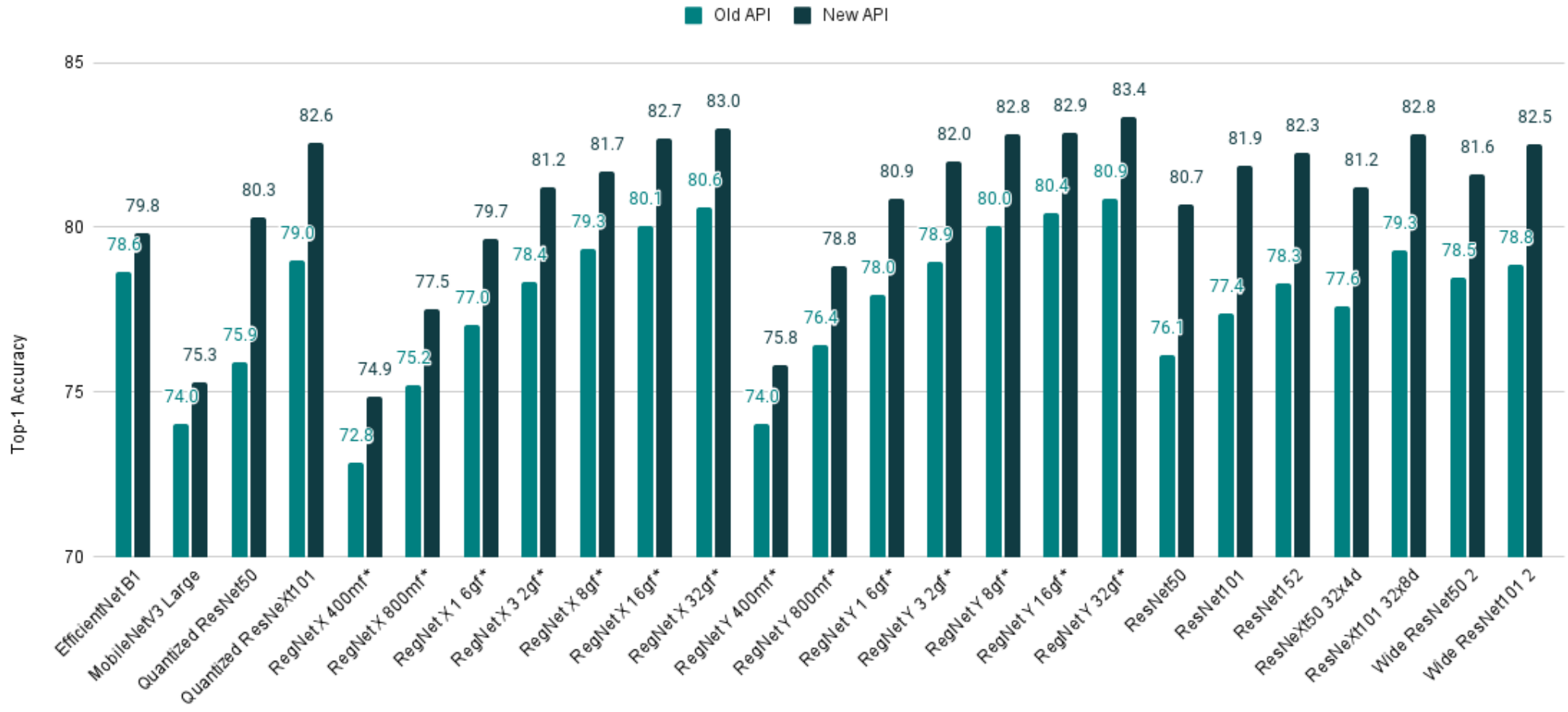
In 2021, PyTorch refreshed their pre-trained weights and API.

"... training models is not a journey of monotonically increasing accuracies and the process involves a lot of backtracking. To quantify the effect of each optimization, below we attempt to show-case an idealized linear journey of deriving the final recipe starting from the original recipe of TorchVision. We would like to clarify that this is an oversimplification of the actual path we followed and thus it should be taken with a grain of salt."




https://pytorch.org/blog/how-to-train-state-of-the-art-models-using-torchvision-latest-primitives/

# Training recipe matters




https://pytorch.org/blog/introducing-torchvision-new-multi-weight-support-api/