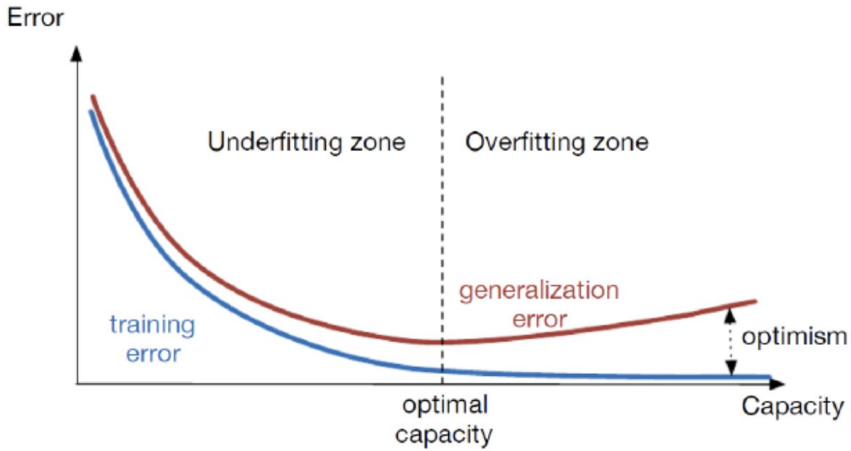
Training Neural Networks

University of Victoria - PHYS 555

What do we want to optimize?

- Optimization Goal: reduce the training error
- Deep Learning Goal: reduce the generalization error
- We need to pay attention to overfitting



Empirical Risk to minimize training error

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(\mathbf{x}_i, \theta) + \lambda \Omega(\theta))$$

Empirical Risk to minimize training error

Empirical Risk

$$J(\theta) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} \ell(\mathbf{x}_i, \theta)}_{\text{Cost: fit to the data}} + \lambda\Omega(\theta))$$

regularization strength (can be different on layers)

penalty to constraint the weights (not biases), data independent

Cost and loss

To optimize the network parameters build a cost function on the training dataset:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, f(\mathbf{x}_i; \theta)).$$

Cost function: average loss over the training sample

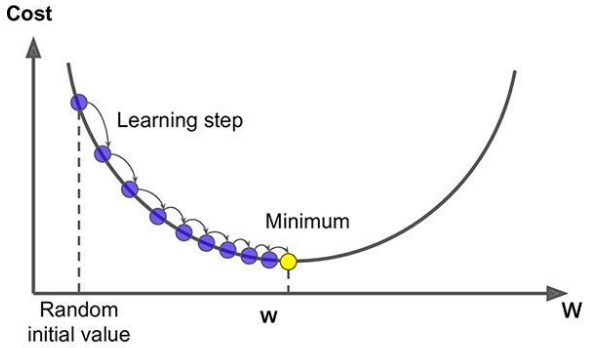
loss function: compares to the data

NN model function

How to optimize?

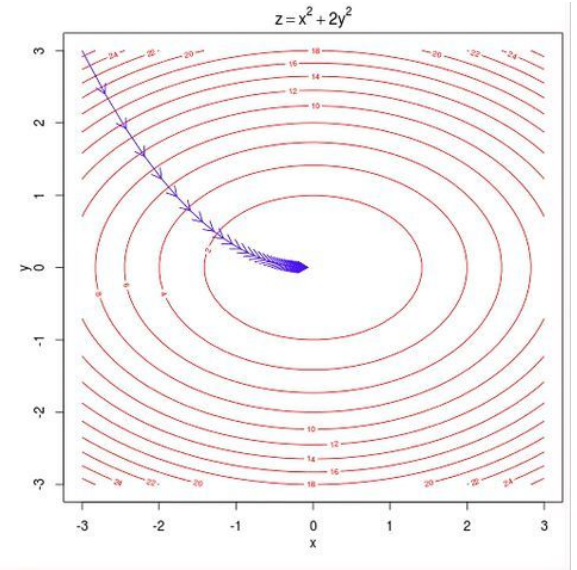
Gradient descent recap

We can use gradient descent to estimate
our network parameters by minimizing the cost.



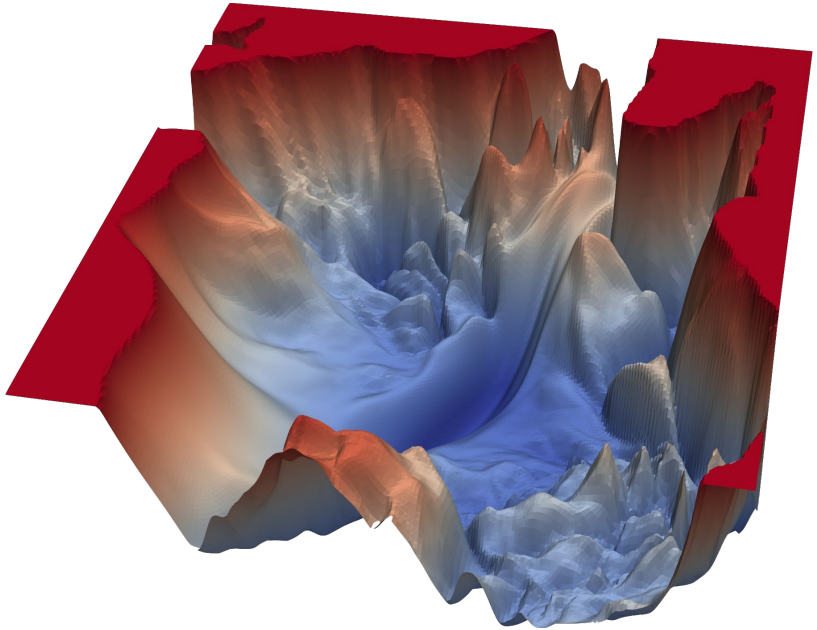
Update rule with learning rate α

$$\theta_{k+1} = \theta_k - \alpha \nabla \ell(y_i; f(\mathbf{x}_i; \theta))$$



Gradient descent in deep learning

- Gradient descent does not guarantee local minimum
- Highly non-convex functions
- Very high-dimensional data
- Datasets too large to compute all gradients
- For large neural networks most local minima yield similar performance



Full Batch Gradient Descent

$$\nabla J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla \ell(y_i, f(\mathbf{x}_i, \theta))$$

Stochastic Gradient Descent (SGD)

$$\nabla J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla \ell(y_i, f(\mathbf{x}_i, \theta))$$

Evaluate loss over a random sample of the training data

Mini-batch Stochastic Gradient Descent (SGD)

Compute gradient on a random mini-batch of B examples.

$$\theta_{k+1} = \theta_k - \alpha \frac{1}{B} \sum_{i=1}^{B} \nabla \ell(y_i; f(\mathbf{x}_i; \theta))$$

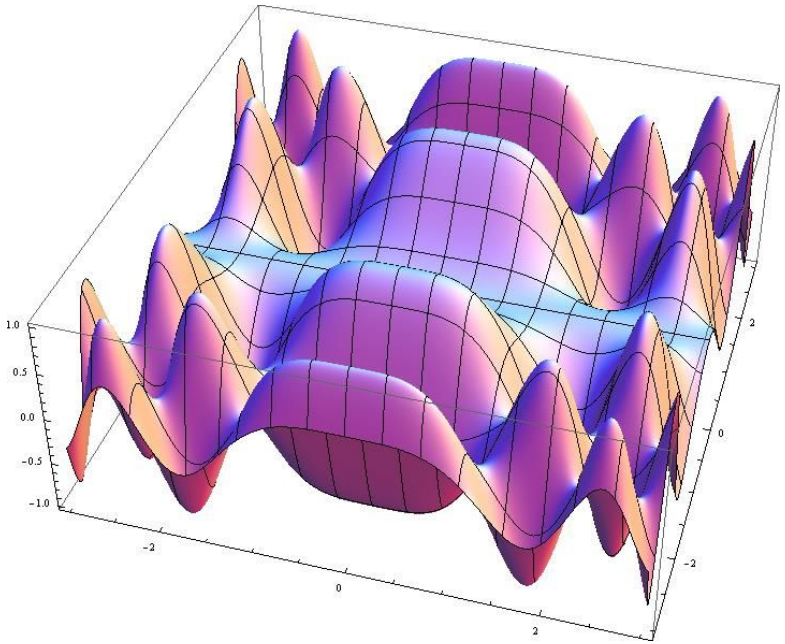
Learning rate and minibatch size
are hyper-parameters

SGD Algorithm with mini-batches

```
for t in range(1, num_epochs):
    shuffle(data)
    for batch in get_batches():
        loss = compute_loss(batch, w)
        dw = compute_gradient(loss)
        w = w - alpha * dw
```

Does global minimum matter?

- Most local minima are close to the global minimum
- Predictions from one local minimum to another seems to vary little
- Use ensemble of training models



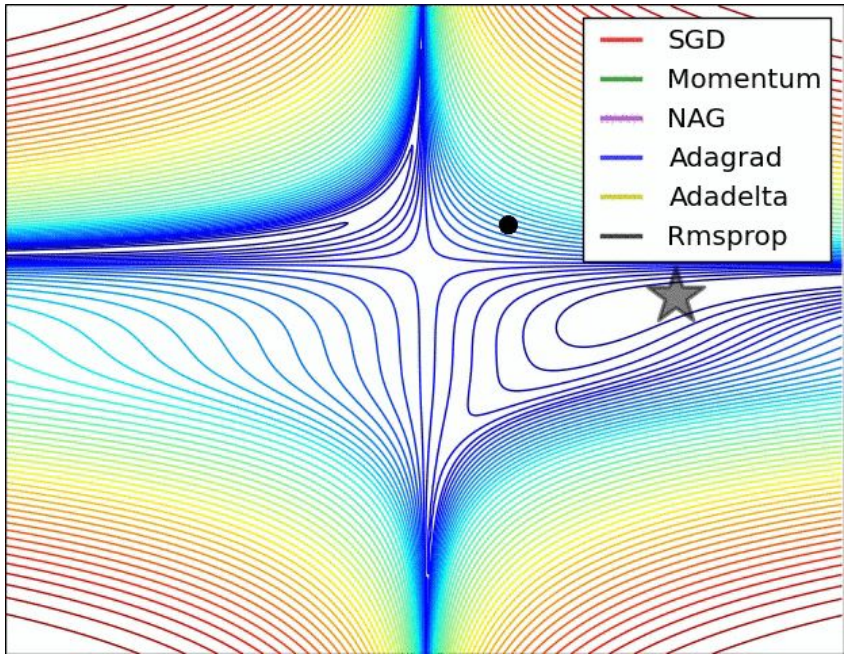
What matters: generalisation to unseen data!

SGD in practice

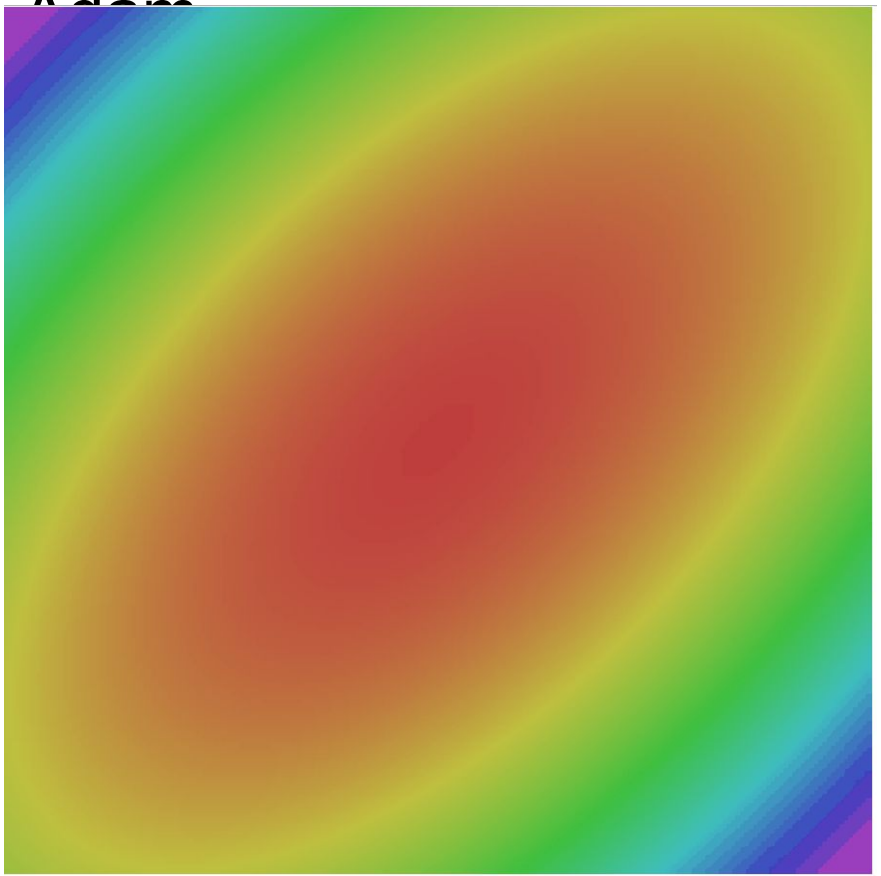
Many algorithms to choose from which implements update rules to accelerate the minimum search:

- SGD + momentum
- Nesterov
- AdaGrad
- RMSProp
- Adam: good default
- RAdam
- ….

Credit: S. Ruder blog



Adam



SGD

SGD+Momentum

RMSProp

Adam

Adam

$$
\begin{aligned}
m_{t+1} &= \beta_1 m_t + (1-\beta_1)\nabla J(\theta_t) \\
v_{t+1} &= \beta_2 v_t + (1-\beta_2)\nabla J(\theta_t)^2 \\
\hat{m}_{t+1} &= \frac{m_{t+1}}{1-\beta_1^{t+1}} \\
\hat{v}_{t+1} &= \frac{v_{t+1}}{1-\beta_2^{t+1}} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon}\hat{m}_{t+1}
\end{aligned}
$$

$\hat{m}_t$ and $\hat{v}_t$ are estimates for the first and second moments of the gradients. Because $m_0 = v_0 = 0$, these estimates are biased towards 0, the factors $(1-\beta^{t+1})^{-1}$ are here to counteract these biases.

Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

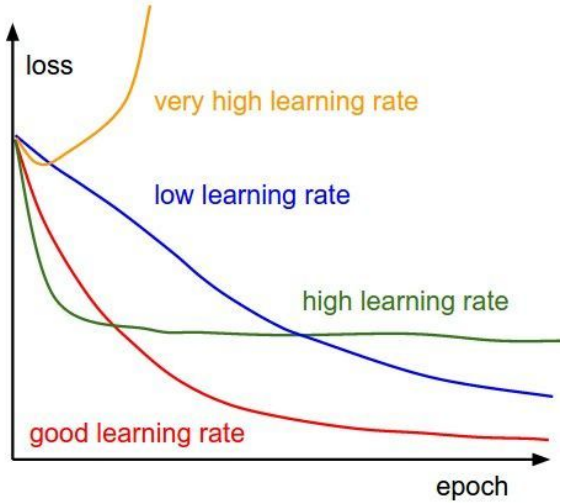
Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-4 is a great starting point for many models!

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

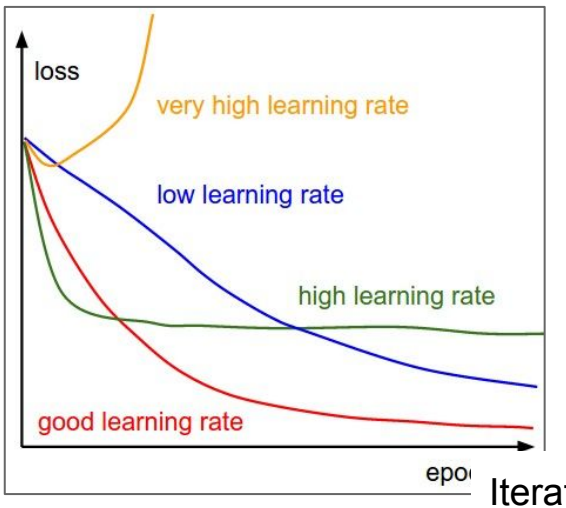
Learning rate

- All optimizers have it
- Very sensitive
  - Too low -> slow convergence
  - Too high -> early plateau or divergence
  - Try first large value such as 0.1 or 1, then divide by 10 in case of divergence
- Large LR prevents final convergence
  - monitor LR, multiply or divide after each update



Learning rate: hyperparameter

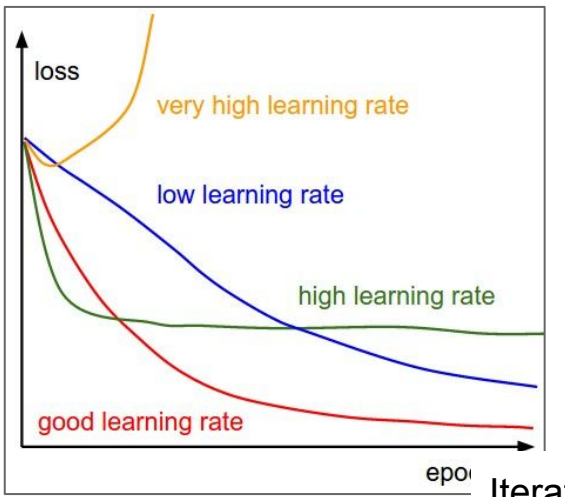
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

Learning rate: hyperparameter

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

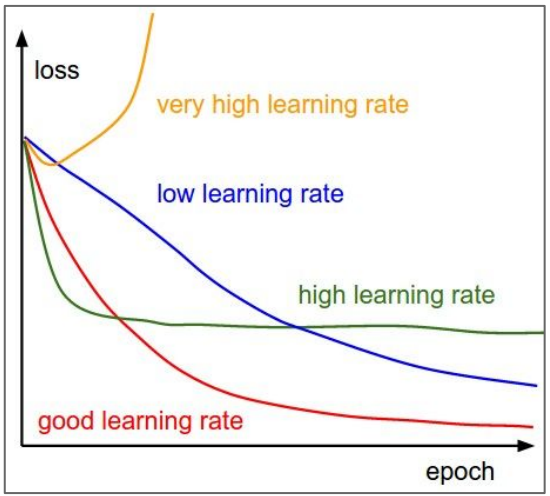
expor $\alpha = \alpha_0 e^{-kt}$ **/:**

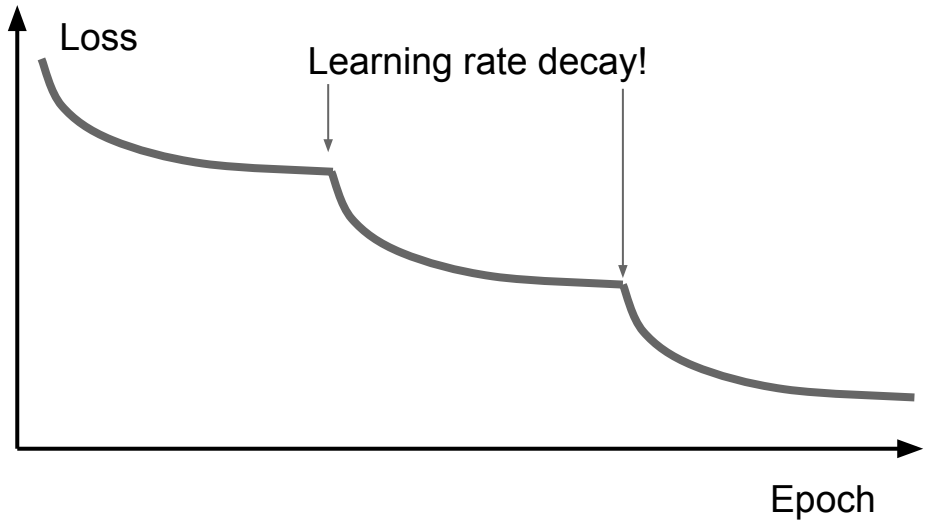
$$\alpha = \alpha_0 / (1 + kt)$$

1/t decay:

Learning rate: hyperparameter

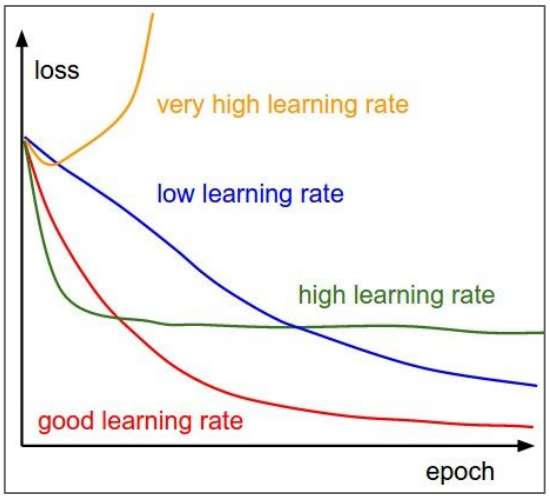
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

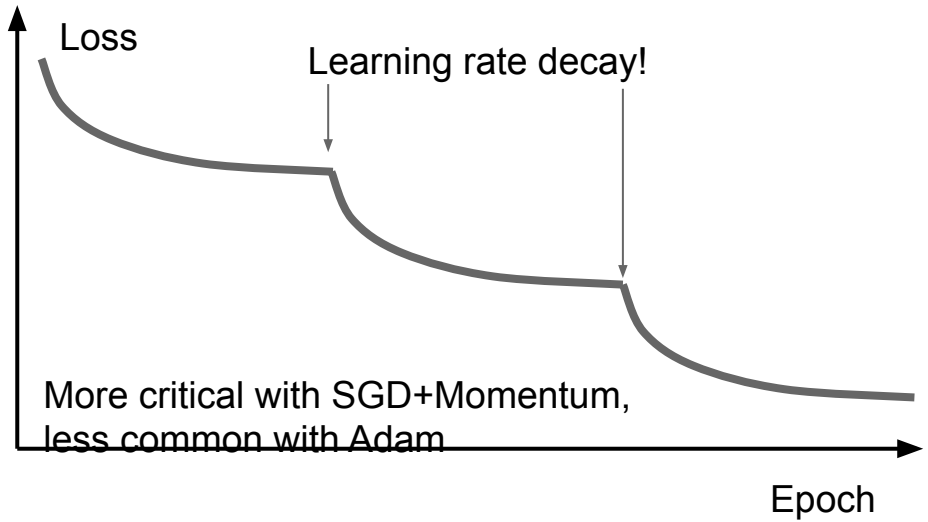




Learning rate: hyperparameter

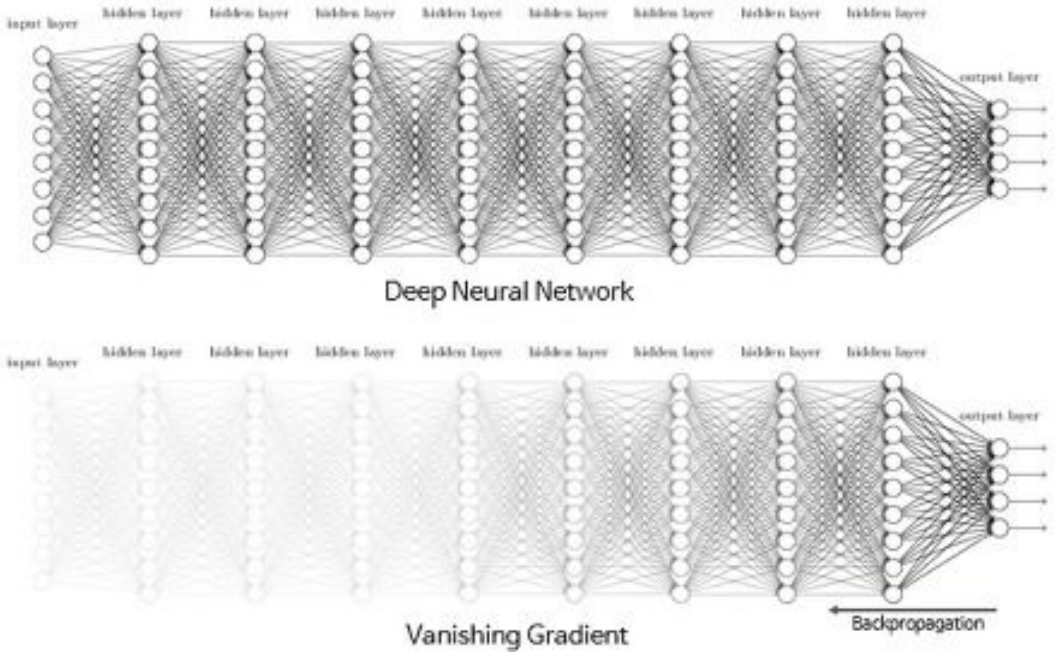
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



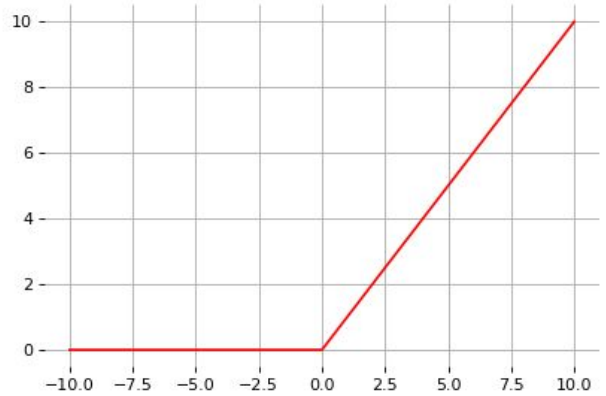


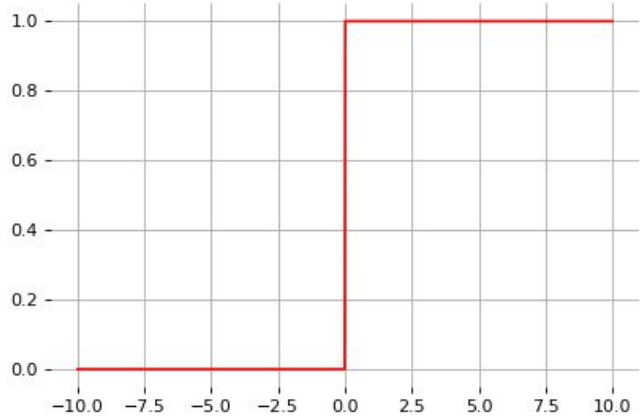
Vanishing gradients

- Small gradients slow down, and eventually prevent SGD to converge
- Limits learning capacity



Gradient control: ReLU



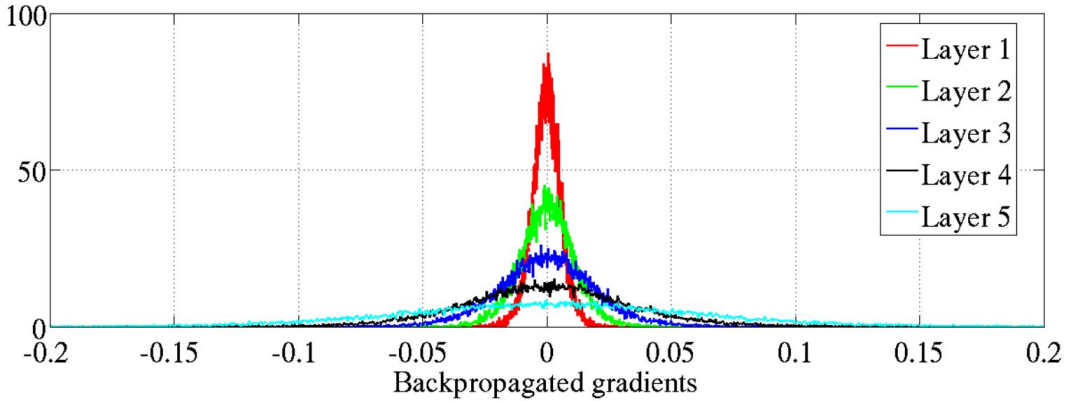


$$\mathrm{ReLU}(x) = \max(0, x)$$

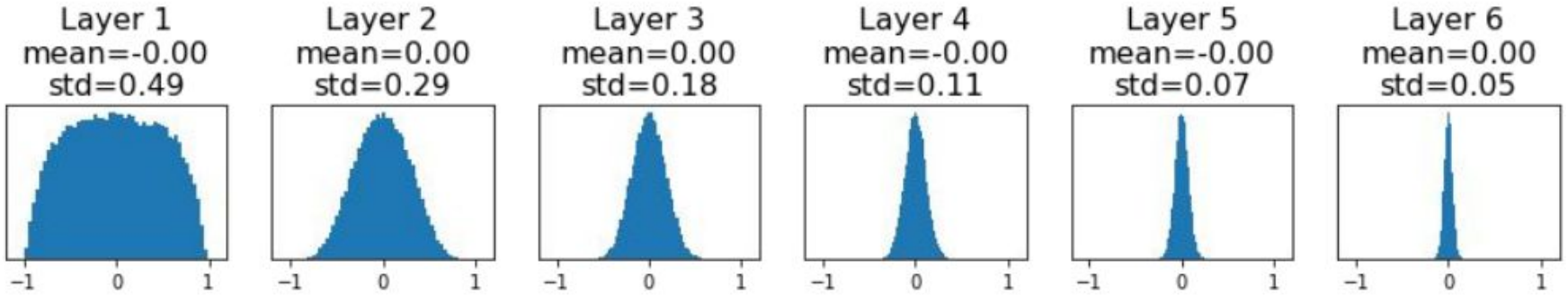
$$\frac{\mathrm{d}}{\mathrm{d}x}\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

Gradient control: weights initialization

- Neural networks weights have to be randomly initialized to break the symmetry
- Normal distribution initialization with a constant σ works okay for small networks but kills gradients for deeper networks



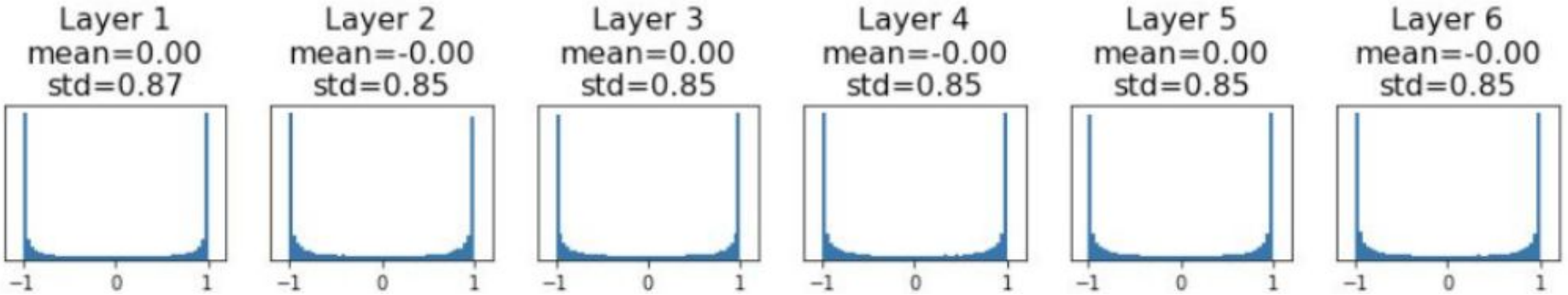
Weights initialization



Initialize with: $\theta \sim 0.01 \times \mathcal{N}(0, 1)$

Tiny activations → tiny gradients

Weights initialization

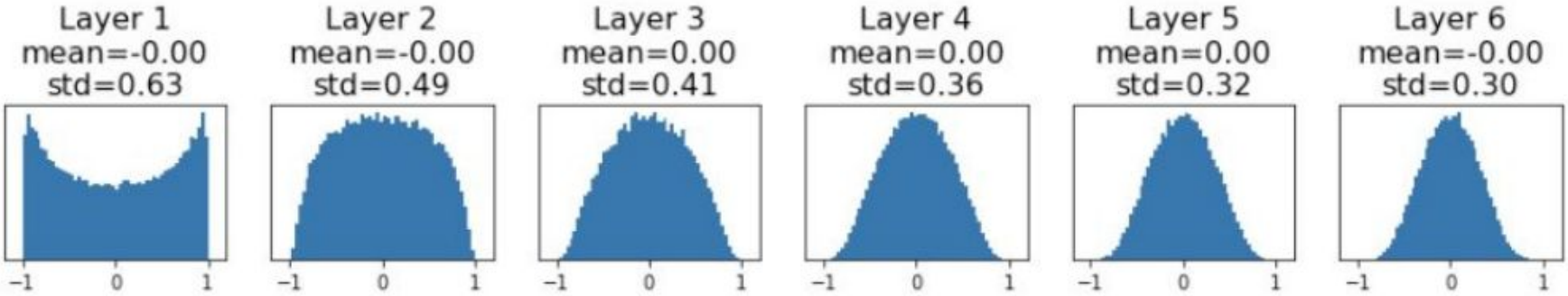


$$\theta \sim 0.05 \times \mathcal{N}(0, 1)$$

Activation saturate → zero gradients

From: CS231n Lecture 7

Weights initialization: Xavier (sigmoid, tanh)



Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

Xavier (also called Glorot) initialization works around the issue by sampling the weight according to a truncated Gaussian:

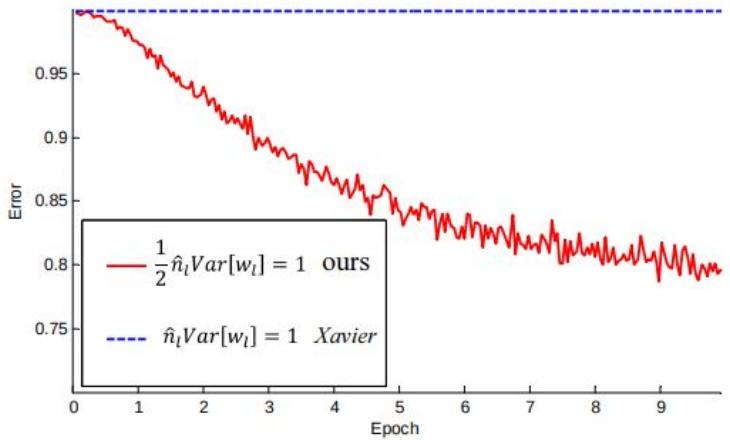
$$\theta \sim \mathcal{N}(0, \frac{2}{N_{\text{in}} + N_{\text{out}}})$$

From: CS231n Lecture 7

Weights initialization: He (ReLU)

- But: Xavier initialization issue with ReLU activated networks
- Scale the incoming weight to have a O(1) variable
- The factor 2 depends on activation: ReLUs ground to 0 the linear activation about half the Time -> Double weight variance for ReLU to adapt
- Kaming He initialization:

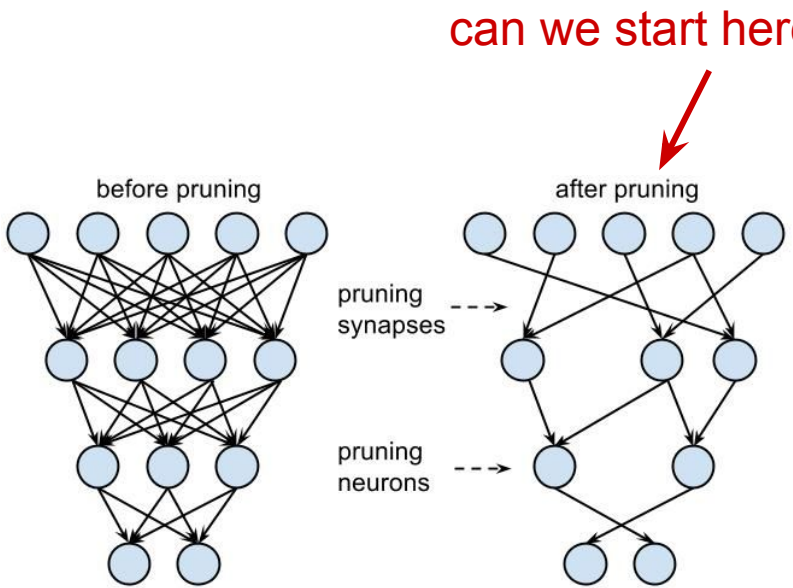
$$\theta \sim \mathcal{N}(0, \frac{2}{N_{\text{in}}})$$



Weights initialization: lottery ticket hypothesis

1. Train network with random init
2. Prune a fraction of the network
3. Reset the weights of pruned network to initialization values from step 1
4. Train the pruned network

A randomly-initialized, dense neural network contains a subnetwork that is initialized such that — when trained in isolation — it can match the test accuracy of the original network after training for at most the same number of iterations



Frankle & Carbin, [arXiv:1803.03635]

Regularization: weight decay and Lasso

Same as traditional linear regression, two types of common regularizations can be added to the cost function to minimize:

$$J_R(\mathbf{w}) = J(\mathbf{w}) + \lambda\Omega(\mathbf{w})$$

- L2 = weight decay : $\Omega(\mathbf{w}) = \frac{1}{2}\sum_i w_i^2$
- L1 = Lasso: $\Omega(\mathbf{w}) = \frac{1}{2}\sum_i |w_i|$

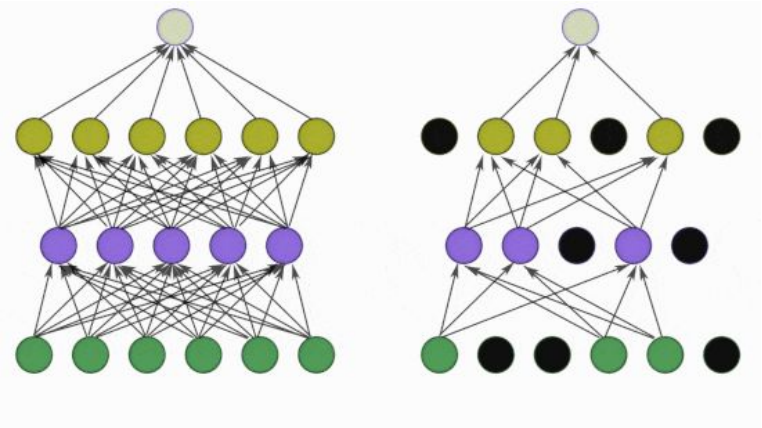
Question: what do they effectively do?

Regularization : dropouts

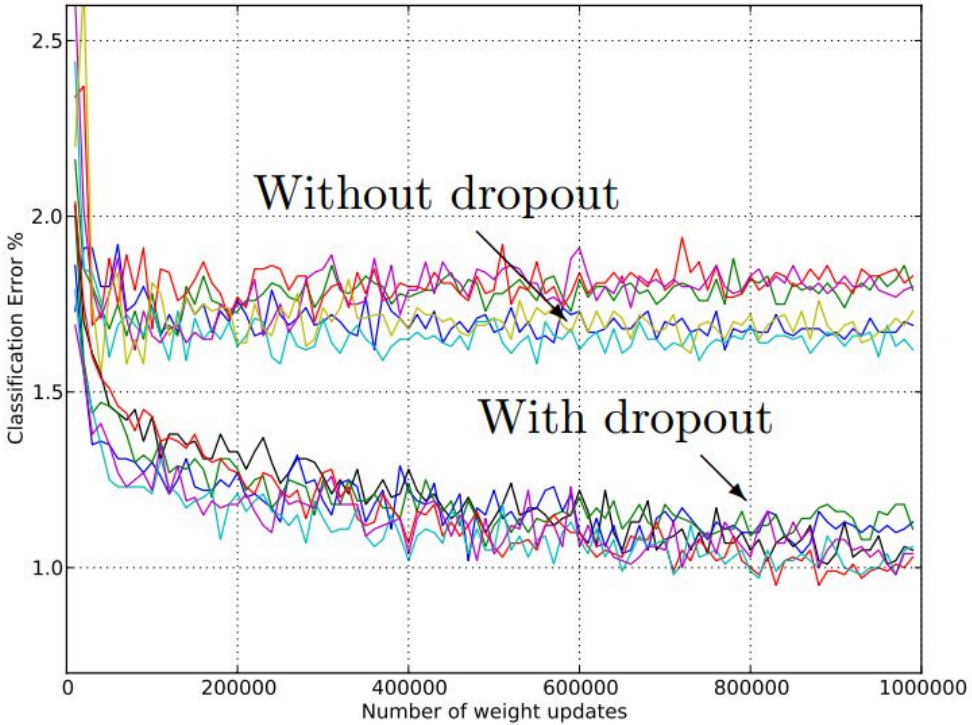
Original procedure:

- Training: randomly dropping out network connections with a fixed probability *p* in the forward pass

- Inference: keep all units, but scale them by *p*



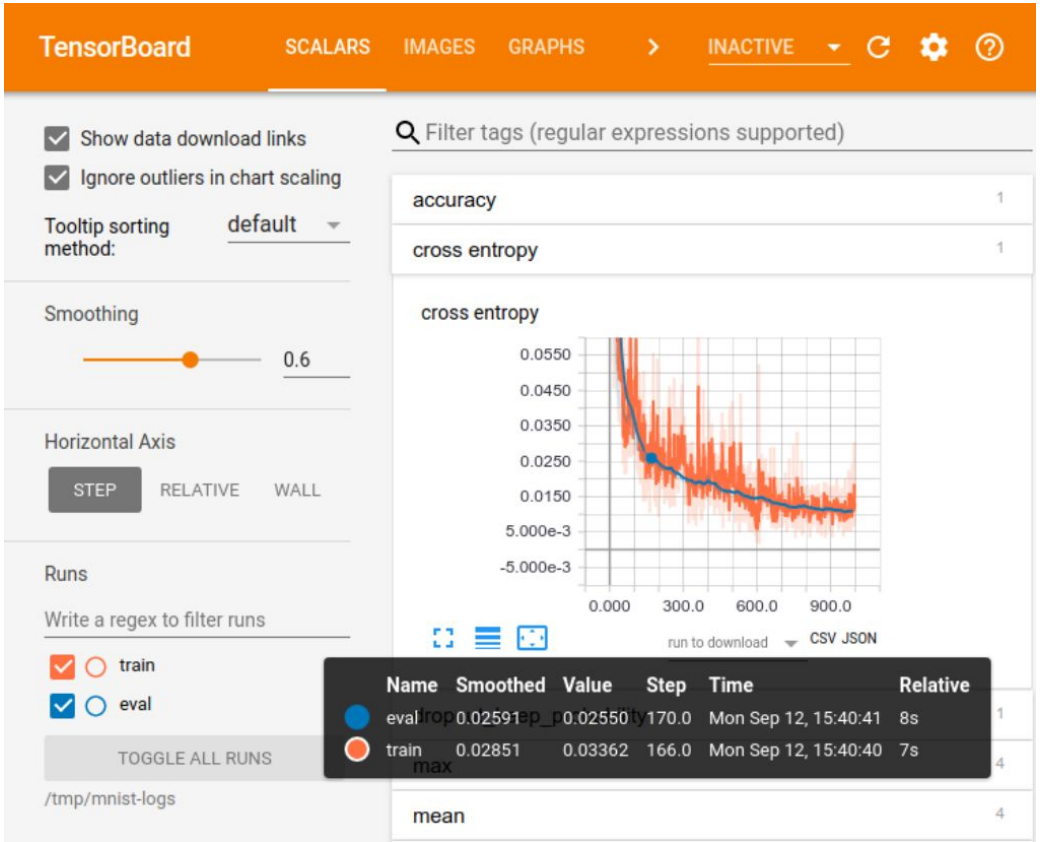
Regularization: dropouts



Srivastava et al. 2014

Training Practices

ALWAYS PLOT YOUR LOSSES !!!

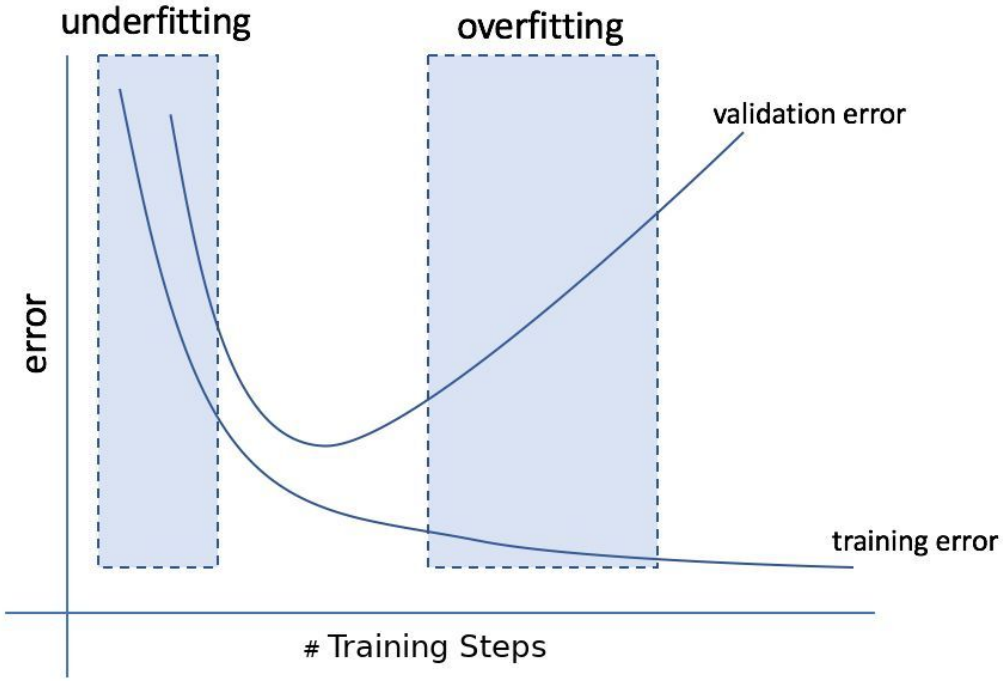


Monitoring the training

- monitor loss on training set
- monitor loss on validation set

Training is slow:

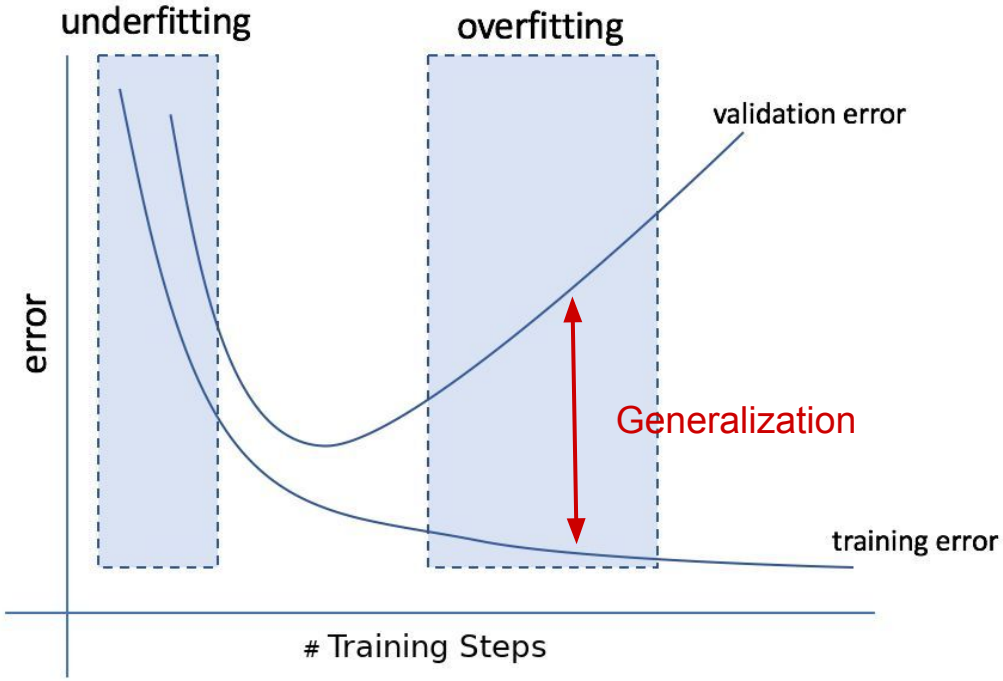
- Cross-validation can be costly
- Test set still held out



Monitoring the training

Overfitting!

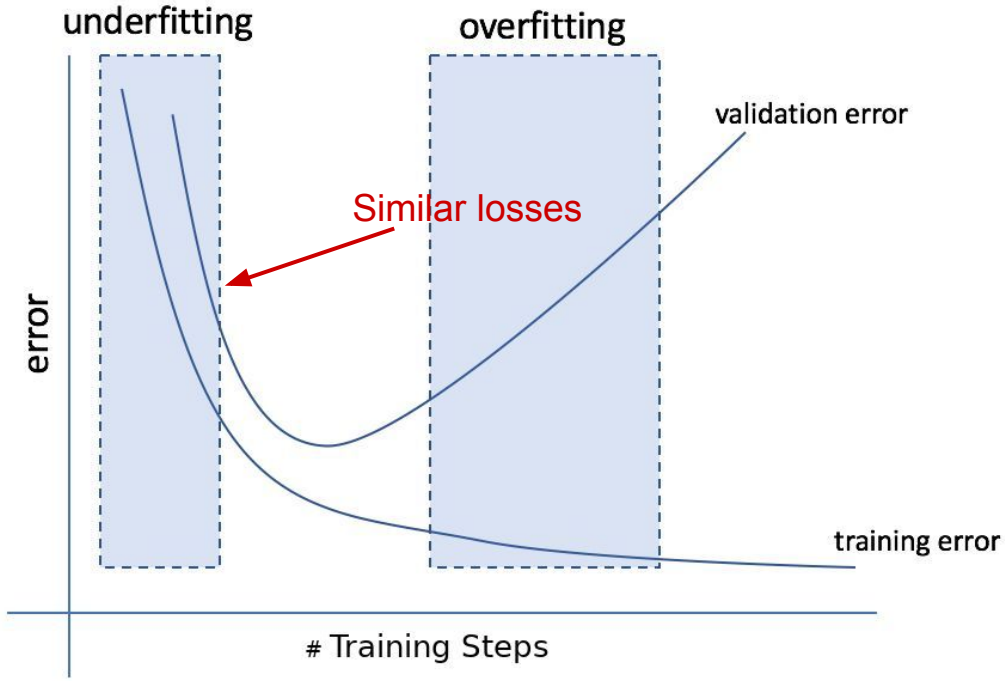
- Data:
  - do you have enough?
  - can you artificially augment it?
- Algorithm:
  - Reduce model complexity
  - Tune learning rate
  - Use regularization



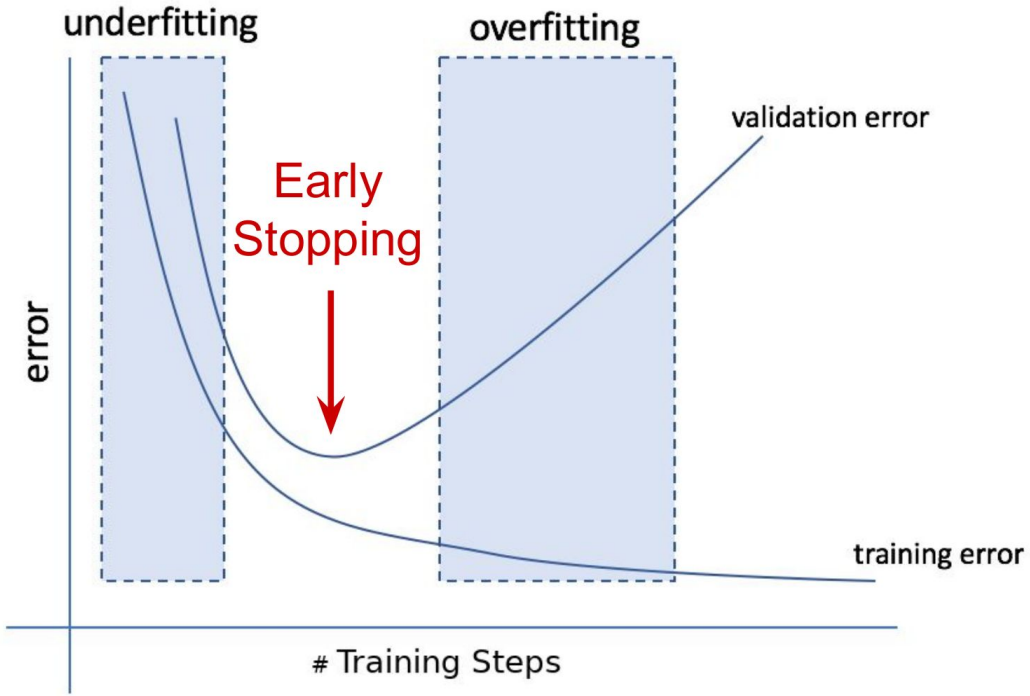
Monitoring the training

Underfitting!

- Data:
  - ...
- Algorithm:
  - Train longer
  - Complexify your architecture
  - Check learning rate
  - Check hyper-parameters



Early stopping



What does Early Stopping mean?

Procedure to find the optimal number of iterations

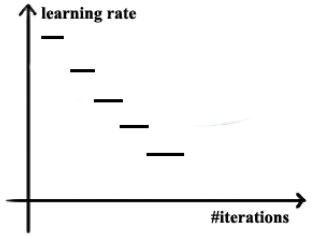
1. Keep a validation set (you already are doing it, right???)
2. At each epoch, check validation loss (or accuracy)
3. When validation does not improve for a a number of patience epochs, stop!

Decaying learning rate

- Minimization process => gradients goes to zero.
  But: SGD would still have noise due to its stochastic nature

- Learning rate should satisfy these sufficient conditions (Robbins-Monro)
  - ensures it explores enough space
  - ensure it converges

⇨ Decay the learning rate



Decaying learning rate

There are various **learning rate schedulers** that are common in practice:

- Linear decay
- Exponential decay
- Cosine
- Inverse square-root

These are applied as a function of SGD step or epoch.

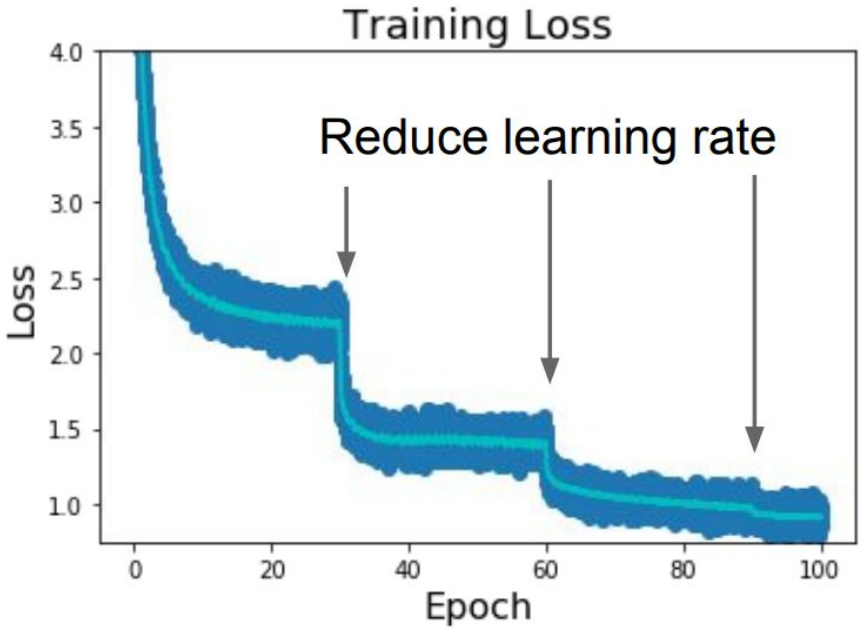
- Example: exponential decay would be:
  - α_0 : initial learning rate
  - t : step or epoch number
  - T : total number of steps or epochs.

$$\alpha = \alpha_0(1 - \frac{t}{T})$$

Decaying learning rate

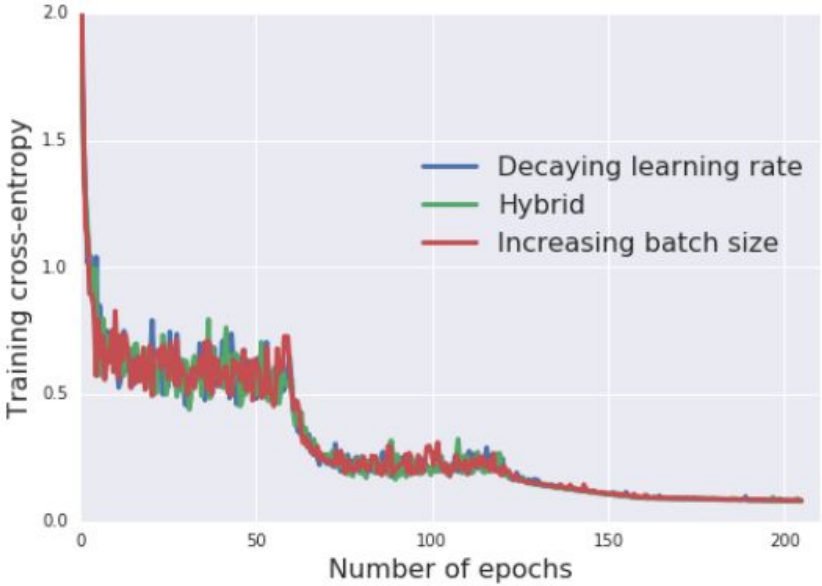
Another approach is to monitor the training curves and reduce the learning rates on plateaus, e.g. divide learning rate by 10 when validation error plateaus

`torch.optim.lr_scheduler.ReduceLROnPlateau`



Alternative to rate decay: increase the batch size

SGD is a mix between stochastic noise learning rate and batch size



arXiv.org > cs > arXiv:1711.00489

Computer Science > Machine Learning

Don't Decay the Learning Rate, Increase the Batch Size

Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, Quoc V. Le

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

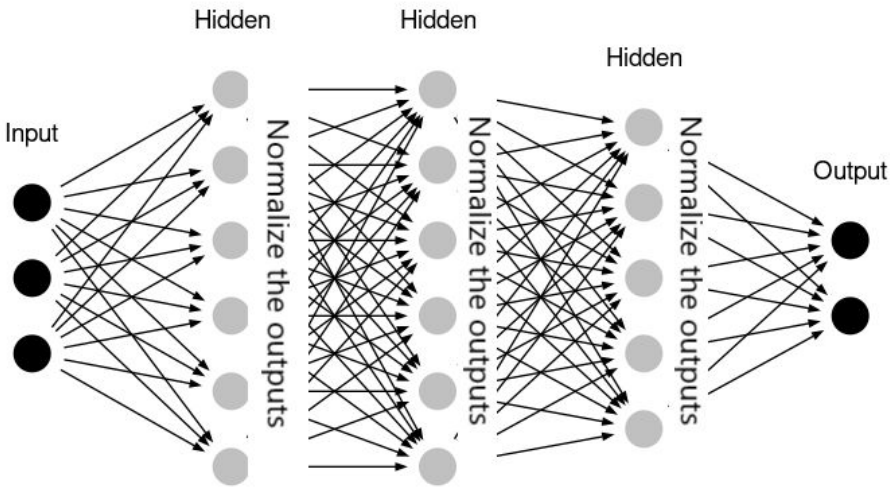
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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



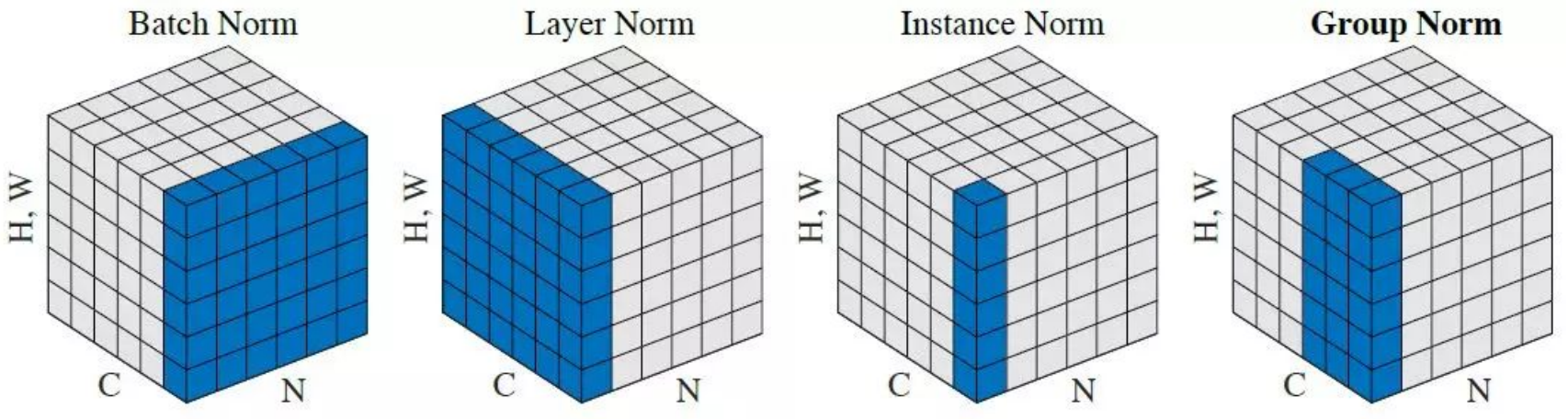
Batch Normalization, Loffe & Szegedy, [arXiv:1502.03167]

Batch Normalization

- Gradients are easier to flow through networks
- Optimization less sensitive to some hyperparameters
  - can use higher learning rates → faster convergence
  - less sensitive to initialization
- smoother optimization landscape (arXiv:1805.11604).
- “implicit regularizer”: affects network capacity and generalization
- Typically inserted before activation.

Behaves differently at training and test time, since test time does not have "mini-batches".

Batch, Layer, Instance, and Group Normalization

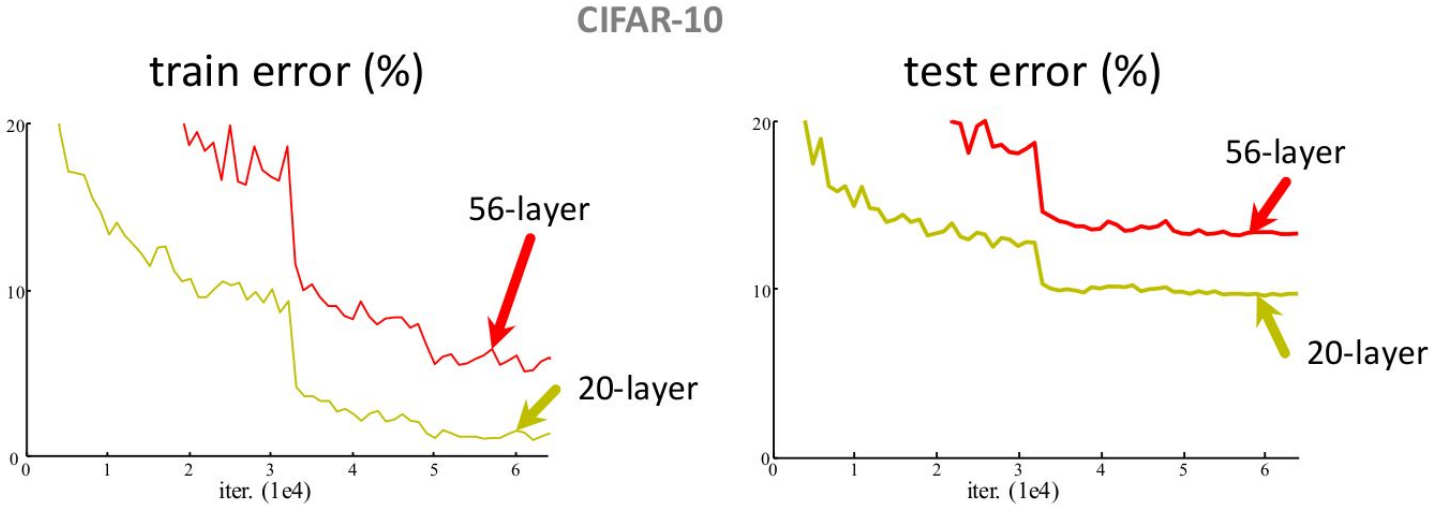


There are various types of normalization layers that use mean and variance statistics along different dimensions. Only BatchNorm is batch dependent.

More details in: Group Normalization, Wu & He, arXiv:1803.08494, CS231n Spring 2019, Lecture 7

Can we keep going deeper?

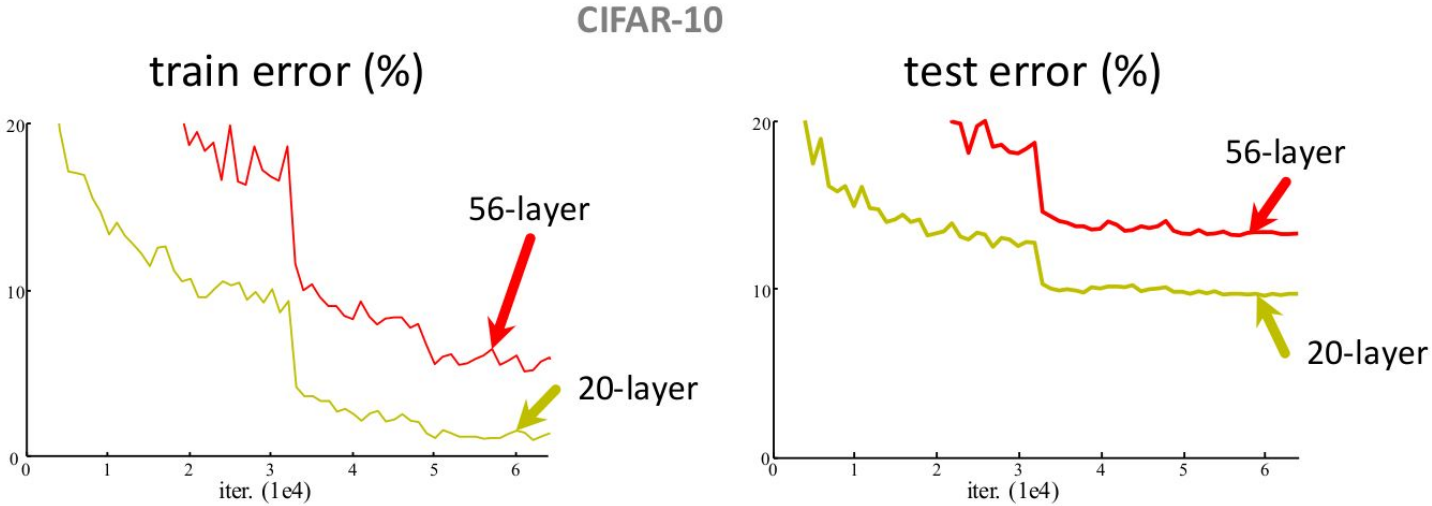
He 2016
CVPR



- ConvNets with 3x3 filters
- What's wrong ?

Can we keep going deeper?

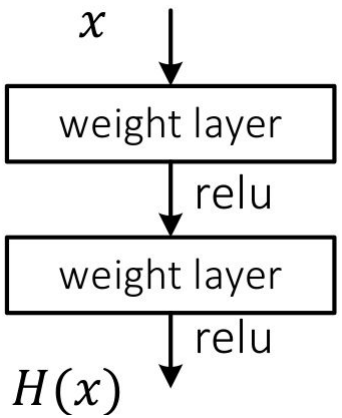
He 2016
CVPR



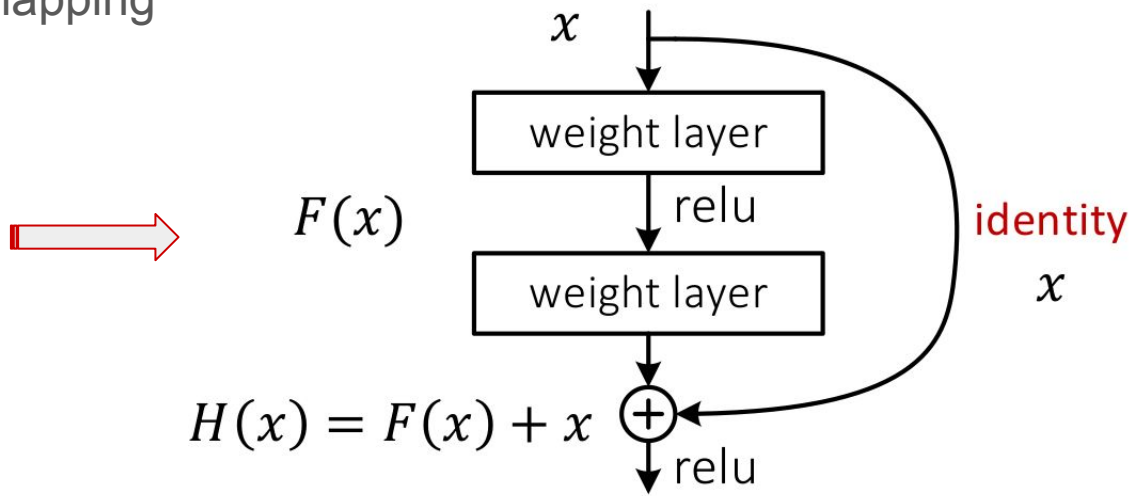
- ConvNets with 3x3 filters
- 56-layer net has higher training error than 20-layer net!
- General phenomenon observed in many datasets
- We can not stack simple layers and improve performance

Residual connection

- Optimization difficulties: solvers cannot find the solution when going deeper
- Solution: use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

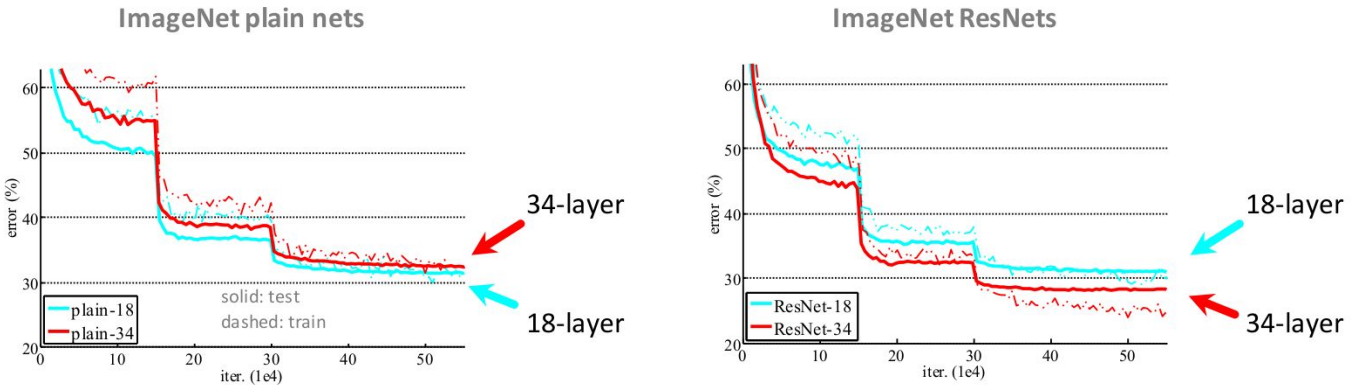


Plain Connection



Residual Connection
aka "Skip Connection"

Residual connection: effect on ImageNet



- ConvNets with residual connections can be trained with less difficulties
- Deeper ResNets have both lower training and testing errors

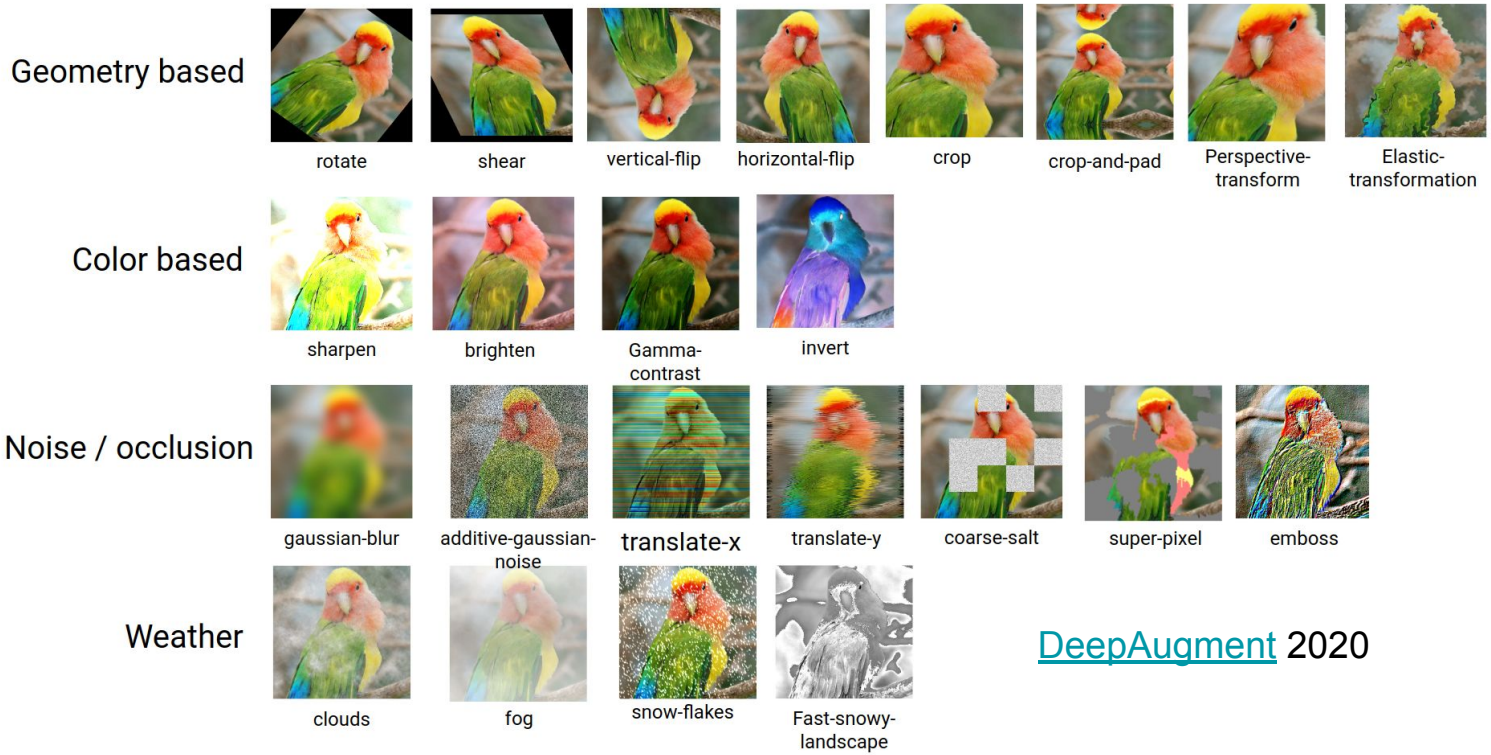
Not enough data?

What are the solutions?

- Collect more data
- Synthetic data
- Augment the data set with generating new data with simple transformations

Data Augmentation

Data augmentation

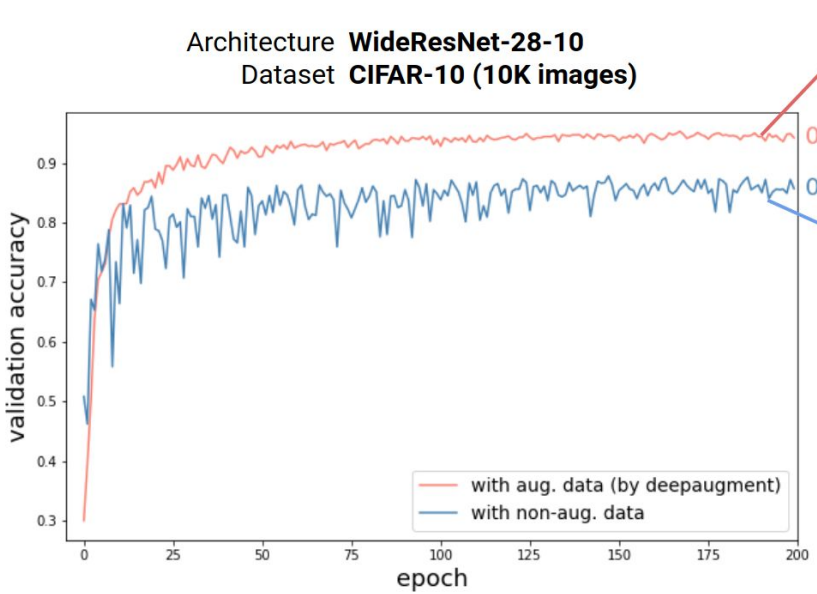


DeepAugment 2020

Data augmentation example

Data Augmentation Libraries:
- Augmentor
- AugLy

DeepAugment applied



8.5% accuracy increase
= **60%** reduction in error

DeepAugment not applied
(no augmentation)

Training Practices Summary

Preprocessing time

1. Data Augmentation?

Designing the network

2. Skip Connections?
3. Batch Normalization?
4. [Dropout]?

Optimization

5. Initialization
6. Learning rate decay, batch size
7. Early Stopping?
8. L1/L2 Regularization?

tinyurl.com/Karpathy-recipe

Others

- Ensemble of models?
- Start by overfitting to debug

Ensembles of deep networks

Just like decision trees, you can build ensembles of deep nets.

1) Train multiple versions of model (initialization, hyper-parameters, …)
2) Ensemble the results, e.g.:
    - Average the prediction of models at test time.
    - Average weights of multiple models (Polyak-Ruppert Averaging)
    - Use multiple snapshots of the same model

May give extra ~2% performance and more robustness to unseen data.

Recipe to architecture a deep network

- Data preparation: rescaling
- Select number of layers, neurons
- Select activation functions
- Add other layers: dropouts, normalization
- Final output, with according activation

Then train.

The Right Balance to Design a Network