

# Learning rate schedules

In addition to smarter optimizers, we can control learning rates by changing them across epochs of training.

This is very much of an art rather than a science.

We give a brief overview.

# Warm up

[Bag of Tricks for Image Classification using CNNs \(https://arxiv.org/abs/1812.01187\)](https://arxiv.org/abs/1812.01187)

- When training starts: initial values of  $\mathbf{W}$  far optimal values
- At this point, losses (and gradients) are probably large
  - large updates to  $\mathbf{W}$  might cause instability

So, we can start off "slow" with a low initial rate during a *warm-up period*.

- low learning rate compensates for high gradient

Post the warm-up, we can use a higher rate to speed training.

## Post warm-up

Typical strategy has been to decrease learning rate as the number of epochs increase.

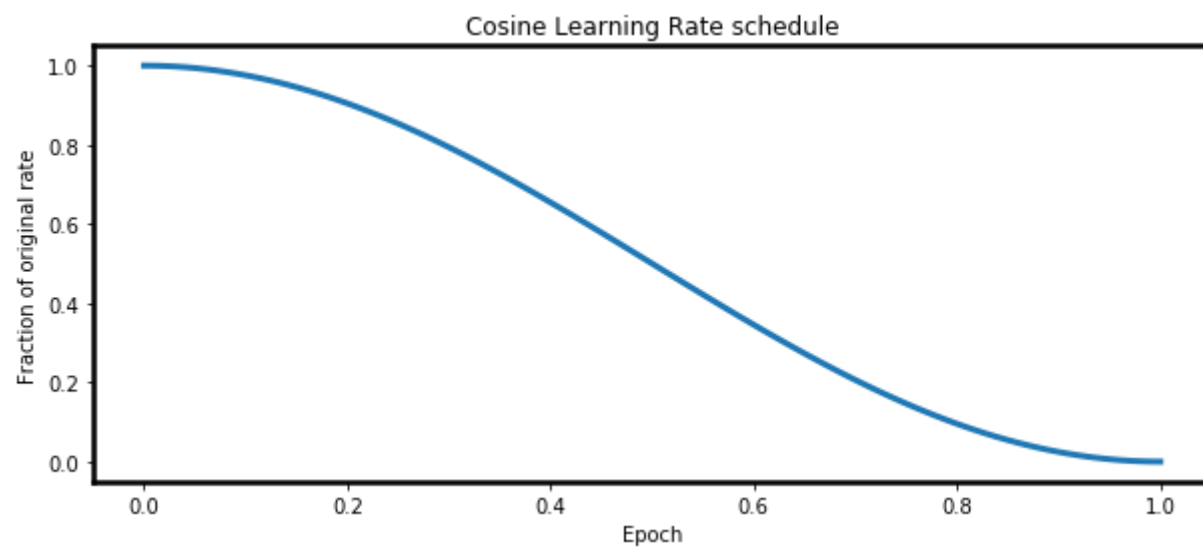
Idea is to take smaller steps as we approach the region of optimality

- don't want to overshoot

There are many ways to set a learning rate schedule (a function that maps epoch number to a rate)

- step schedule
  - vary rate by epoch
  - rate decreases as epoch increases
- cosine decay
  - decrease rate according to a cosine function
    - $\text{learning\_rate}_t = \frac{1}{2} (1 + \cos(\pi \frac{t}{T})) * \text{learning\_rate}_0$
    - where  $\text{learning\_rate}_0$  is initial learning rate,  $T$  is number of batches
    - slow decrease in rate at start
    - near-linear decrease in middle
    - slow decrease near end

```
In [5]: _ = nnh.plot_cosine_lr()
```





# Regularization

The ultimate goal of Machine Learning is out of sample prediction.

Because Neural Networks often learn a large number of parameters (weights), overfitting is a concern.

We will briefly review several methods to combat overfitting

## **Loss function: add regularization penalty**

The same methods that were applicable in Classical Machine Learning apply to Deep Learning as well.

These include regularization penalties that aim to reduce the number of parameters.

- L2 regularization
- L1 regularization



# Dropout

Dropout paper (<http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>).

Overfitting can occur because some weights in the NN adapt so as to memorize "noisy" features.

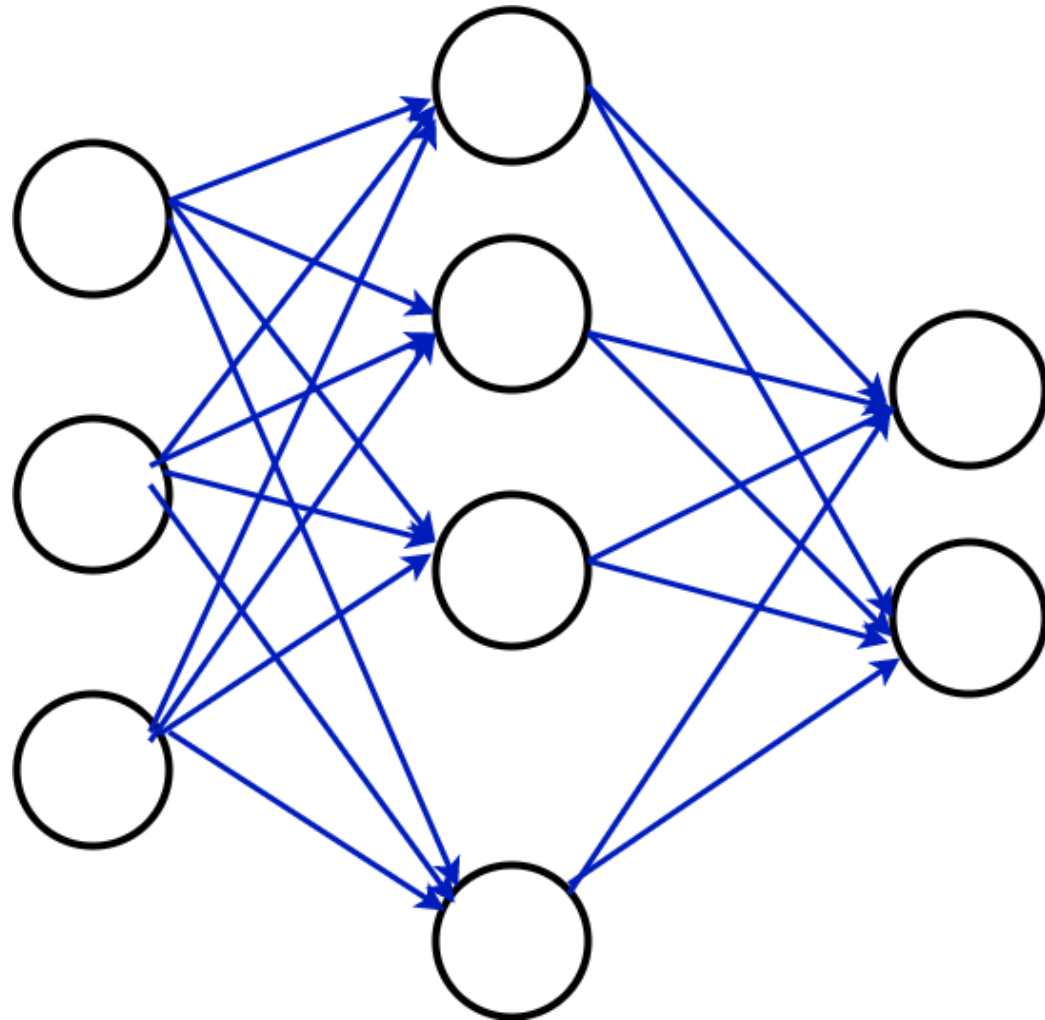
*Dropout* is a method that *randomly drops a unit in the NN*

- For each training example  $\mathbf{x}^{(i)}$
- Each unit gets dropped with probability  $p$



NN, no dropout

---



A Neural Network with  $N$  units contains  $2^N$  possible sub-networks.

Dropout can be viewed as training many of these sub-networks (with weight sub-networks.)

If a feature is truly important, the NN must adapt to robustly recognize the

If it is not important, the goal is to prevent a unit from memorizing it.

In Keras, Dropout is implemented by a layer:

`Dropout (rate)`

where `rate` is the probability of dropping a unit.

Dropout has been supplanted by Batch Normalization, but is worth studying

- for its simplicity and ease of use
- inspiration it offers.

# Data Augmentation

It is sometimes possible to expand the training set in such a way as to disoverfitting.

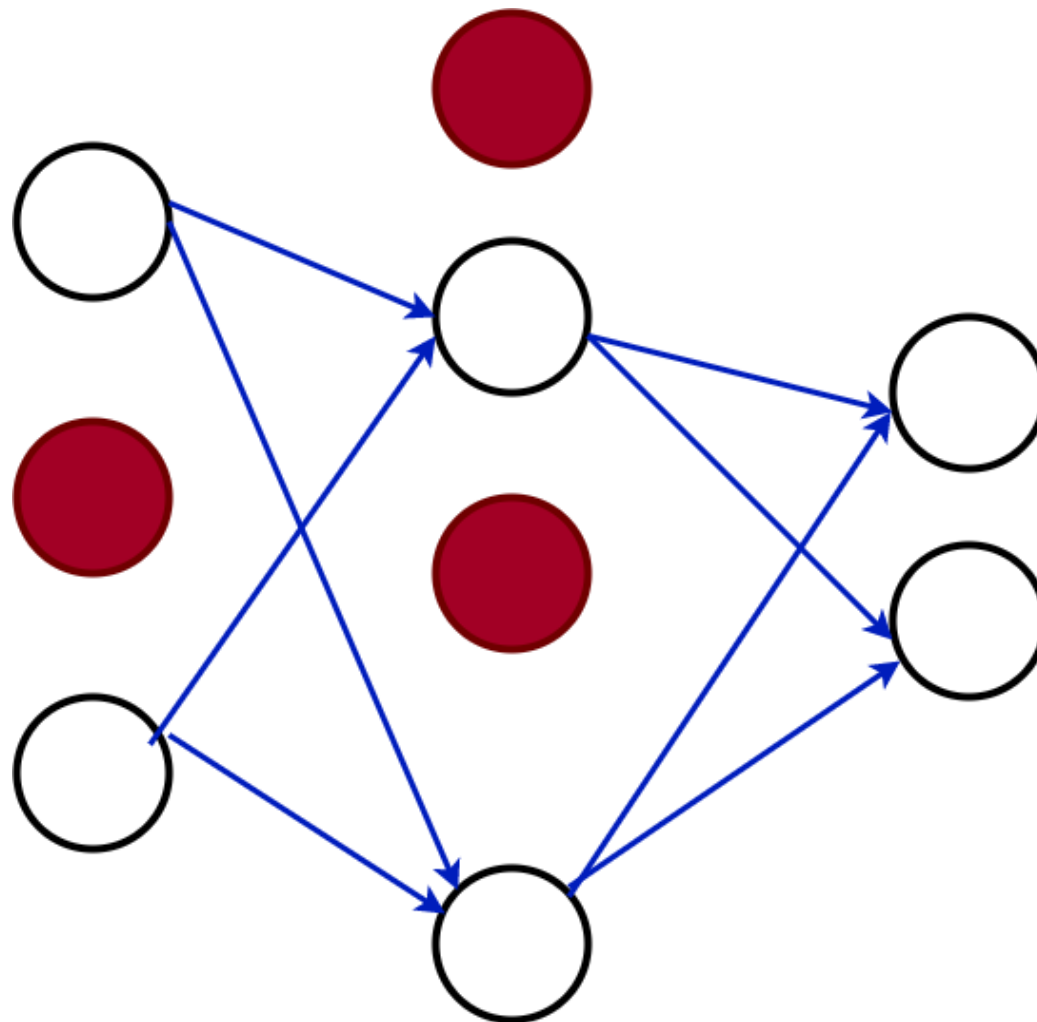
This usually involves creating variants of training examples

- make it hard to memorize them all.

## Input Transformation

Alter the image while preserving its label.

- Image transformation
  - rotate, crop, flip



Example	Smoothed label
$(\mathbf{x}^{(i)}, 0)$	$(\mathbf{x}^{(i)}, 0 + \epsilon)$
$(\mathbf{x}^{(i)}, 1)$	$(\mathbf{x}^{(i)}, 1 - \epsilon)$

So rather than using One Hot Encoding (OHE), we use " $\epsilon$  Hot Encoding"



# Mixup training

[Mixup training paper \(https://arxiv.org/abs/1905.11001\)](https://arxiv.org/abs/1905.11001)

*Mixup training* is a second solution to prevent an NN from seeking absolute confidence.

It creates additional training examples that are *mixtures* of existing examples:

Training example	Mixup ?
$(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$	original
$(\mathbf{x}^{(i')}, \mathbf{y}^{(i')})$	original
$(\mathbf{x}^{(i)} + \lambda \mathbf{x}^{(i')}, \mathbf{y}^{(i)} + \lambda \mathbf{y}^{(i')})$	Mixup

The mixing parameter  $\lambda$  is best when it is close to 0 or 1

- $(0 - \epsilon \text{ or } 1 + \epsilon)$

# Conclusion

Creating multi-layer networks seems simple.

But as with many tasks

- The difference between a design that looks good on paper and one that works well in practice
- Comes down to managing lots of details !

Network design is as much an art as it is a science.

In [6]: `print("Done")`

Done

