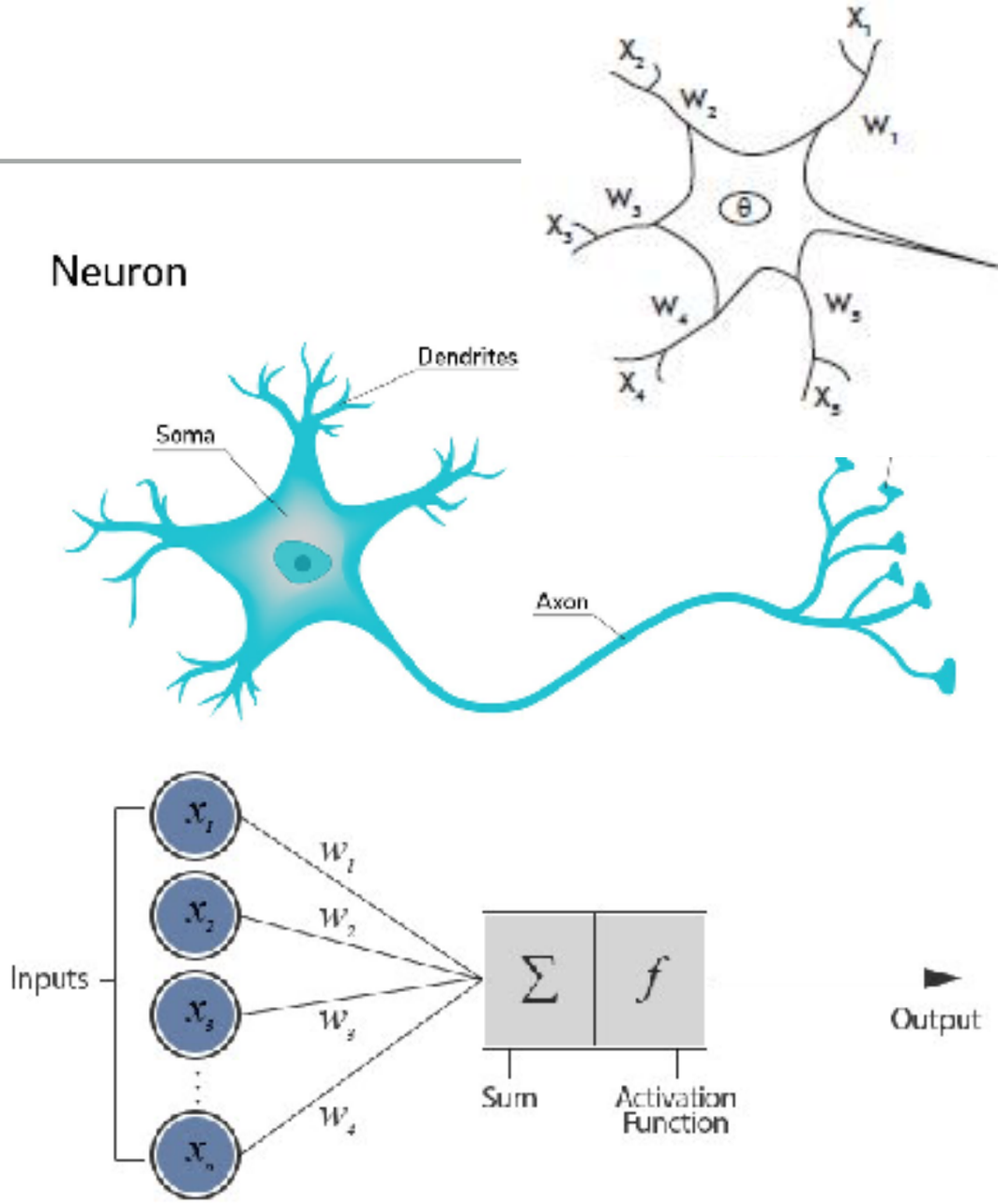
# NEURAL NETWORKS II.

# ANALOGY WITH REAL NEURONS

- Brain:
  - The inputs of a real neuron are weighted
    - Due to the position of synapses (distance from the soma, signal attenuates!), the properties of the dendrites (attenuation), and synapse strength (amount of signalling molecules and receptors)
  - Learning is changing the synapse strength and wiring
  - Hebbian learning: *"Cells that fire together wire together"* (modern: Spike-timing-dependent plasticity (STDP) )
- Artificial neural network:
  - Information is also stored in weights (connections are fixed)
  - Learning is updating weights  with gradient descent using error back propagation: an "error" value stored at every neuron (e.g.: "diff" in caffe)
- The brain can not and does not learn with backprop, but the principles may be similar (paper1, paper2, paper3)



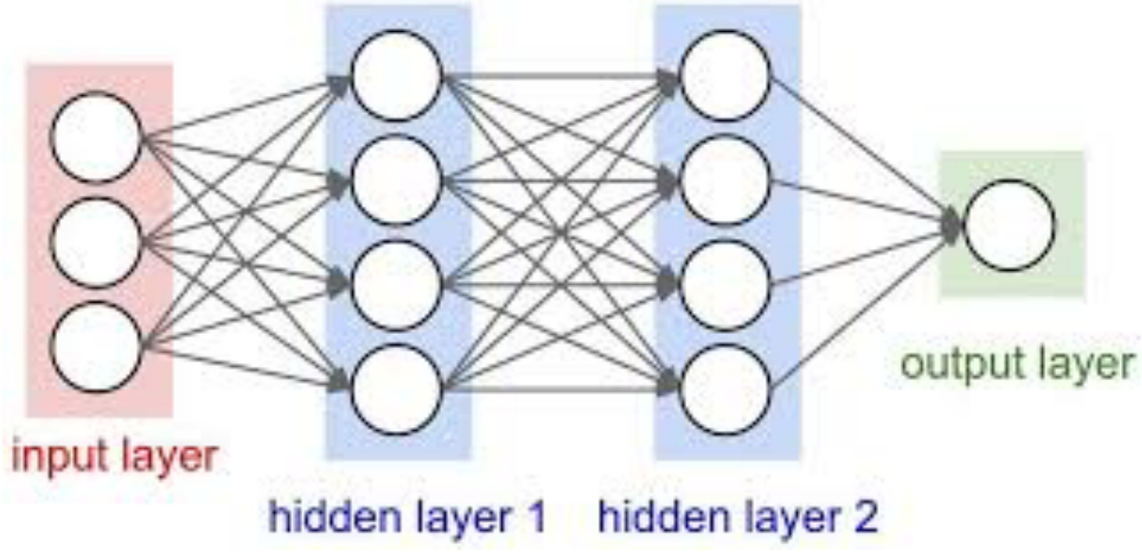

$$o_i = K(s_i) = K(\sum w_{ij} o_j + b_i)$$

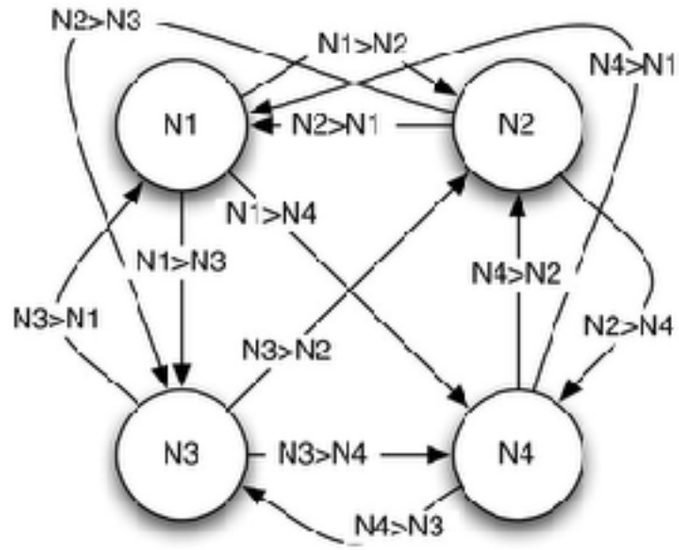
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_i}\frac{\partial o_i}{\partial s_i}\frac{\partial s_i}{\partial w_{ij}} = \frac{\partial E}{\partial o_i} K'(s_i)\, o_j$$

$$\sum_{l \in R} \frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial o_i} = \sum_{l \in R} \frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial s_l}\frac{\partial s_l}{\partial o_i} = \sum_{l \in R} \frac{\partial E}{\partial o_l} K'(s_l) w_{li}$$

# DYNAMIC NEURAL NETWORKS

- Why this course is not about decision trees for example? Interest in NNs not only for function approximation!
- A real neural network is a complex dynamic system. (states of neurons, rules dictated by physics/chemistry/ biology)
- Artificial neural network are interesting complex dynamic system models! ( states: activations (often binary), rules for state updates and learning)
- Hopfield network (fully connected)
  - Binary states, natural update (weighted sum of inputs above threshold -> 1)
  - "Energy": either stays the same or decreases with random asynchronous updates, converges to local minima. (Ising model!)
  - Associative memory within the basin of attractor
  - Learning is changing the weights to create the attractors

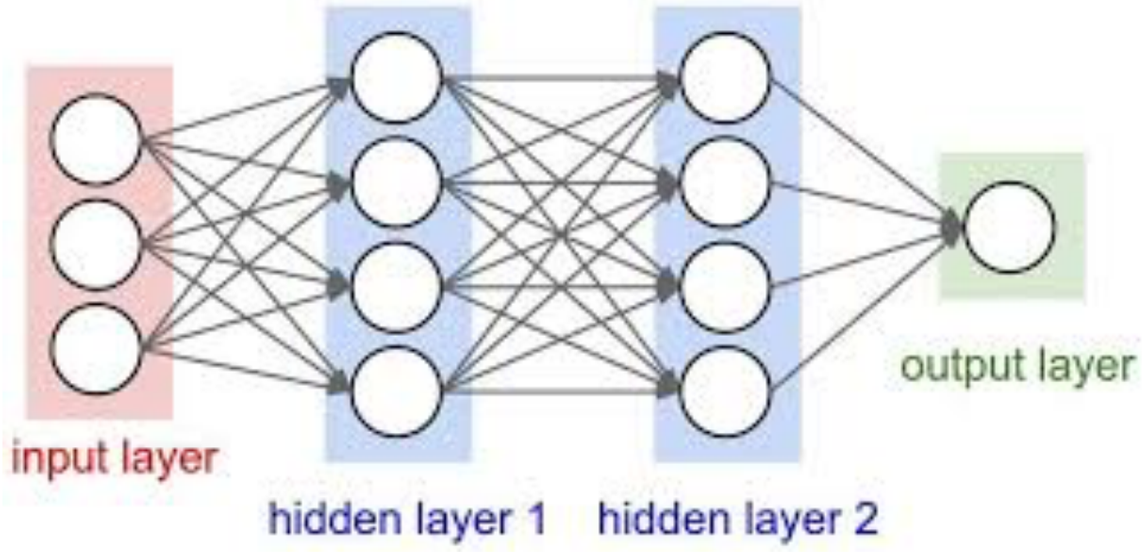



$$E = -\frac{1}{2}\sum_{i,j} w_{ij} s_i s_j + \sum_i \theta_i s_i$$

# DYNAMIC NEURAL NETWORKS

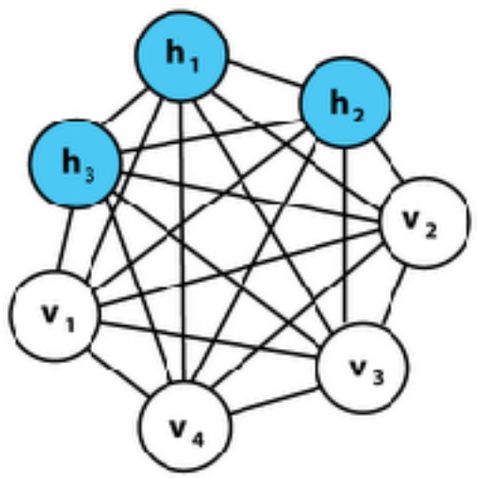
- Generative models are attractive!
  - The brain is not only able to classify dogs/cat, but it can also imagine new ones (you can dream, draw etc a cat)
  - Generative models are believed to be able to learn meaningful things without strong supervised signal!
- Boltzmann machine (fully connected)
  - Hopfield model + random update rule for units, using the “Energy”
  - Visible units (data), hidden units (representation)
  - Training: set weights so that the thermal equilibrium visible state probabilities approximate the training data sample probabilities
  - A Boltzmann Machine generates data with correct distribution.
- Restricted Boltzmann Machine:Only inter-layer connections
- Deep Boltzmann Machine: Multiple layers with connections only between neighbours

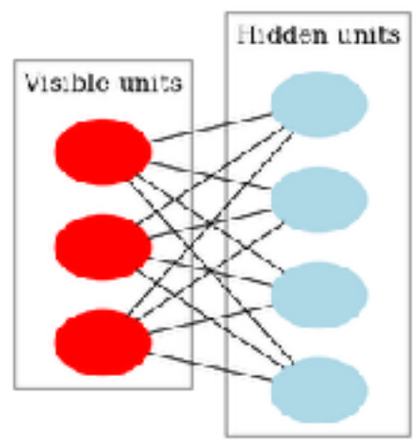



$$E = -\frac{1}{2}\sum_{i,j} w_{ij} s_i s_j + \sum_i \theta_i s_i$$

$$\Delta E_i = -k_B T \ln(p_{\mathrm{i=off}}) - (-k_B T \ln(p_{\mathrm{i=on}}))$$

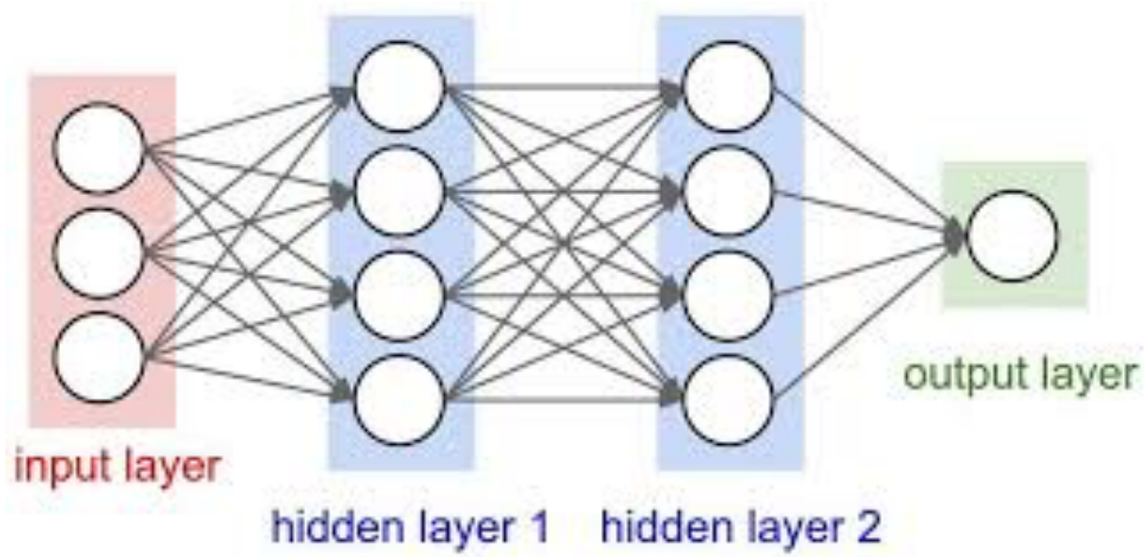
$$p_{\mathrm{i=on}} = \frac{1}{1 + \exp(-\frac{\Delta E_i}{T})}$$

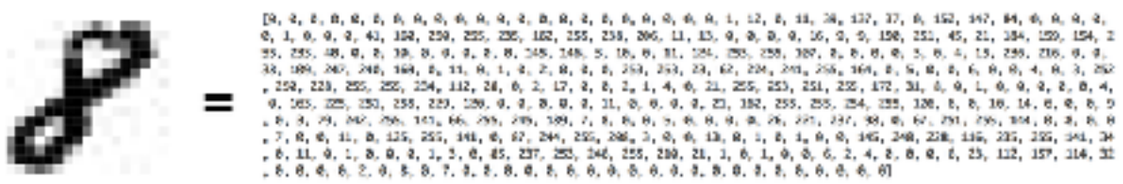


# INTERNAL REPRESENTATIONS

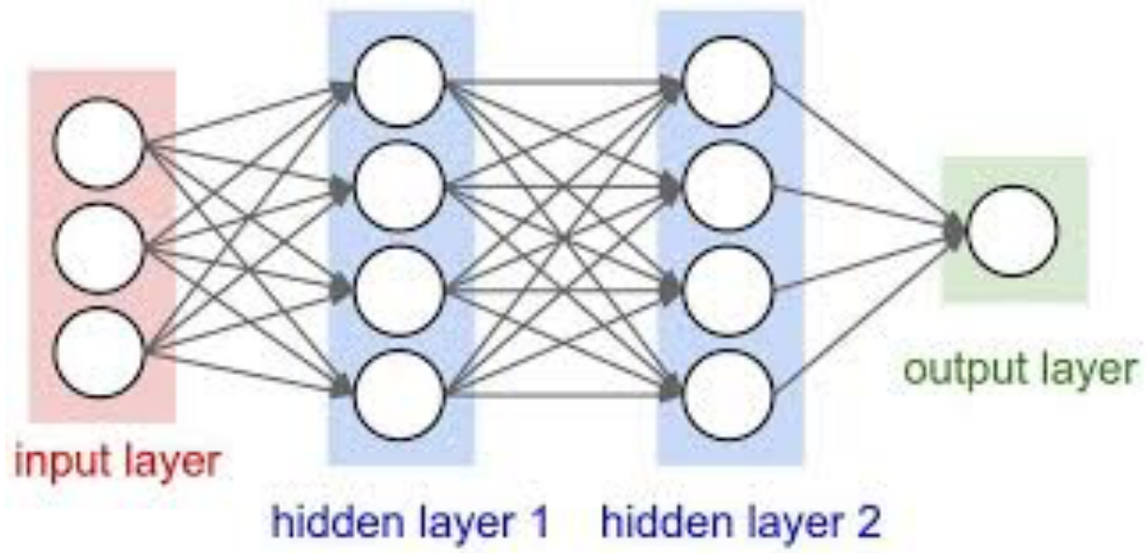
- Boltzmann Machines are directly trained to learn an internal representation of the data!
- A restricted (deep) Boltzmann Machine looks very similar to a feedforward neural network! Layers of a feed forward neural network are also representation of the input data!
- There are no complex learned representations in linear models, or nearest neighbours or decision trees or support vector machines!
- Why do we need internal representations?
  - Very high dimensional complex input space (image, sound)
  - Distance in original space is a bad metric: logical machine learning approaches fail
  - There is a simple low dimensional structure in the data which is not evident from the raw data, e.g.: elephants, dogs



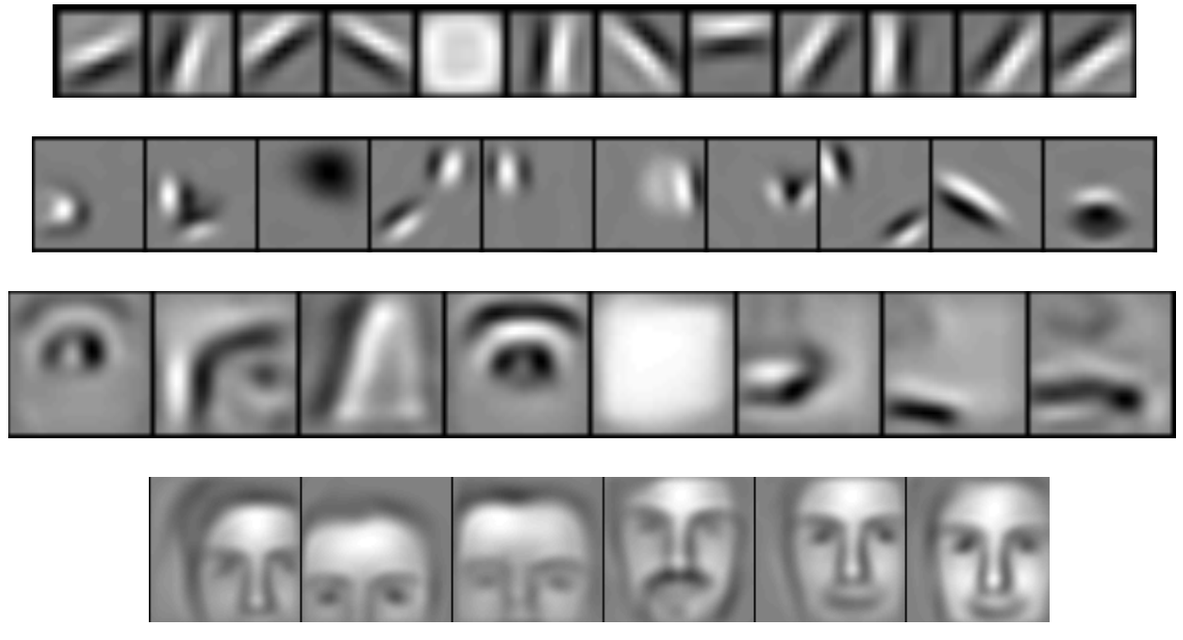



# INTERNAL REPRESENTATIONS

- It must be useful for the neural network to represent the true underlying factors when learning to solve the task, be it generating the true distribution or approximating a function

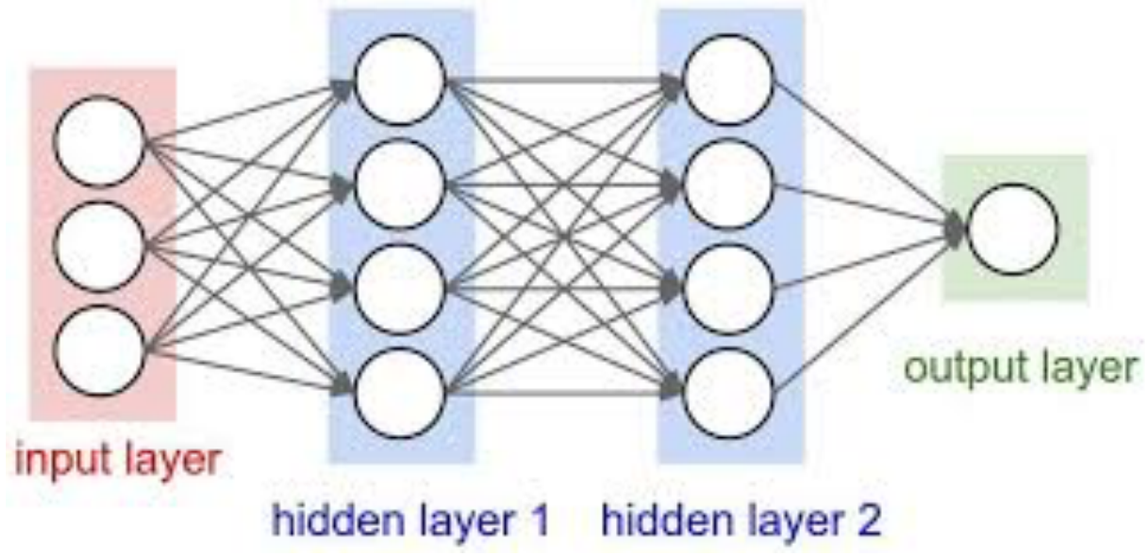



- Neural networks layers with “few” units: learned low dimensional representations
- Layered neural networks can learn hierarchical representations, which can potentially very useful if the problem is well described with hierarchical features!



# WHY LAYERED & FEED FORWARD?

- Error **BACK** propagation
- Backward connections or connection in a layer, no “back” direction  for easy error propagation: no easy recursive formula in the chain rule
- Recurrent neural networks: Backprop through time!



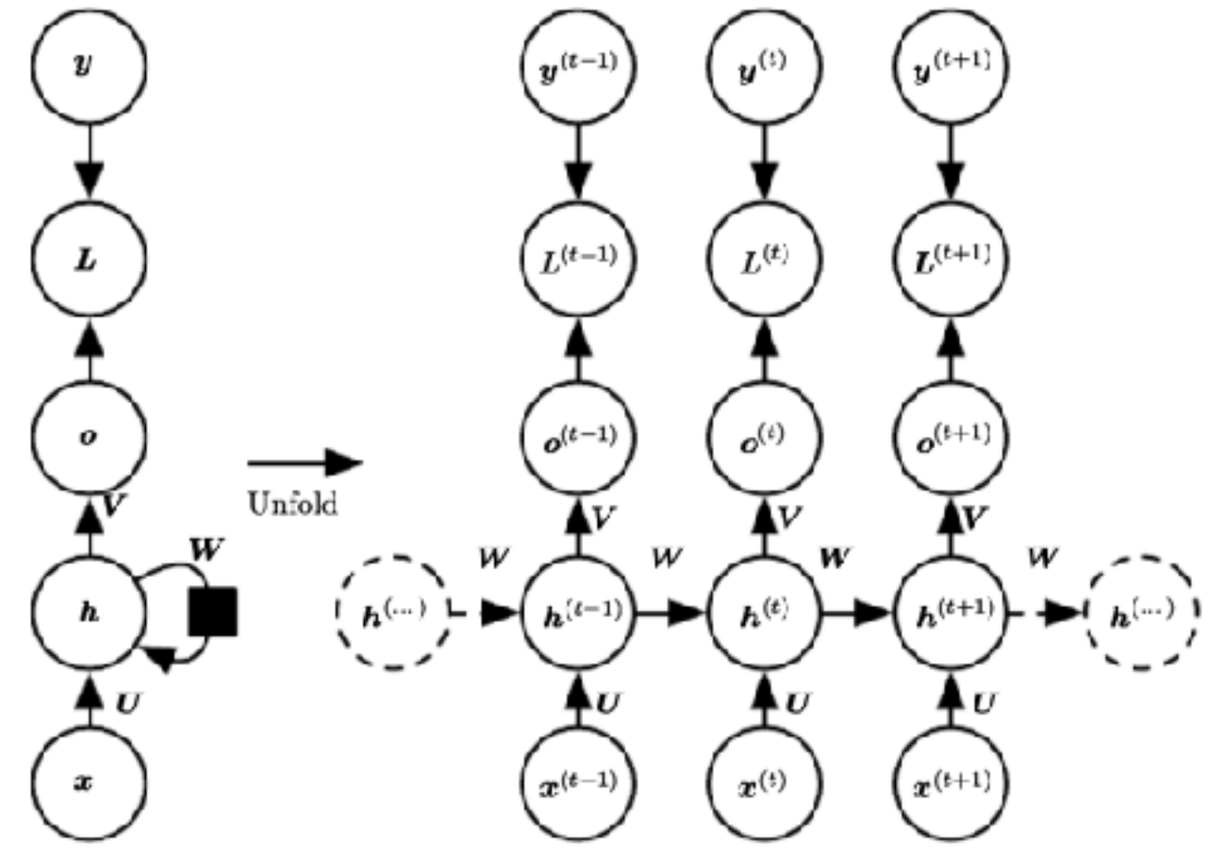

$$o_i = K(s_i) = K(\sum w_{ij} o_j + b_i)$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_i}\frac{\partial o_i}{\partial s_i}\frac{\partial s_i}{\partial w_{ij}} = \frac{\partial E}{\partial o_i} K'(s_i)\, o_j$$

$$\sum_{l \in R} \frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial o_i} = \sum_{l \in R} \frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial s_l}\frac{\partial s_l}{\partial o_i} = \sum_{l \in R} \frac{\partial E}{\partial o_l} K'(s_l) w_{li}$$

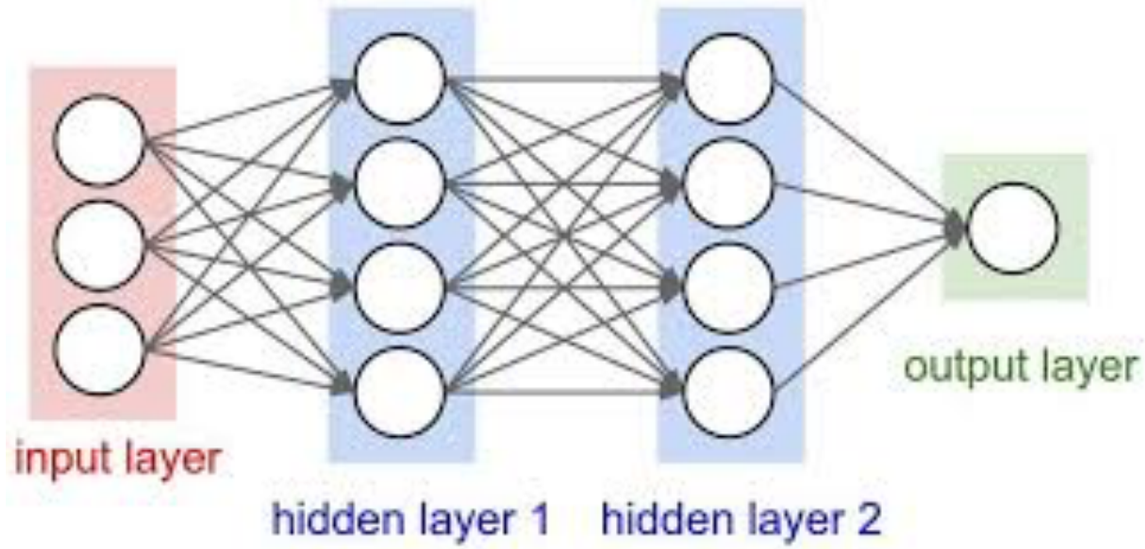
# WHY LAYERED & FEED FORWARD?

- Error **BACK** propagation
- Backward connections or connection in a layer, no “back” direction  for easy error propagation: no easy recursive formula in the chain rule
  - Recurrent neural networks: Backprop through time!



# WHY LAYERED & FEED FORWARD?

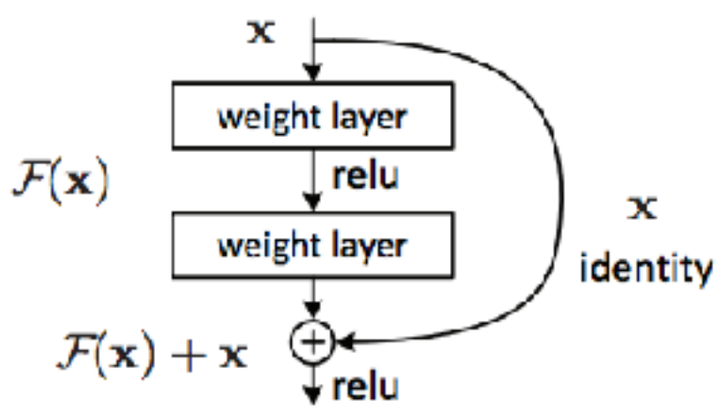
- Error **BACK** propagation
- Backward connections or connection in a layer, no "back" direction for easy error propagation: no easy recursive formula in the chain rule
- Recurrent neural networks: Backprop through time!
- Skip connections are handled!




$$o_i = K(s_i) = K(\sum w_{ij} o_j + b_i)$$

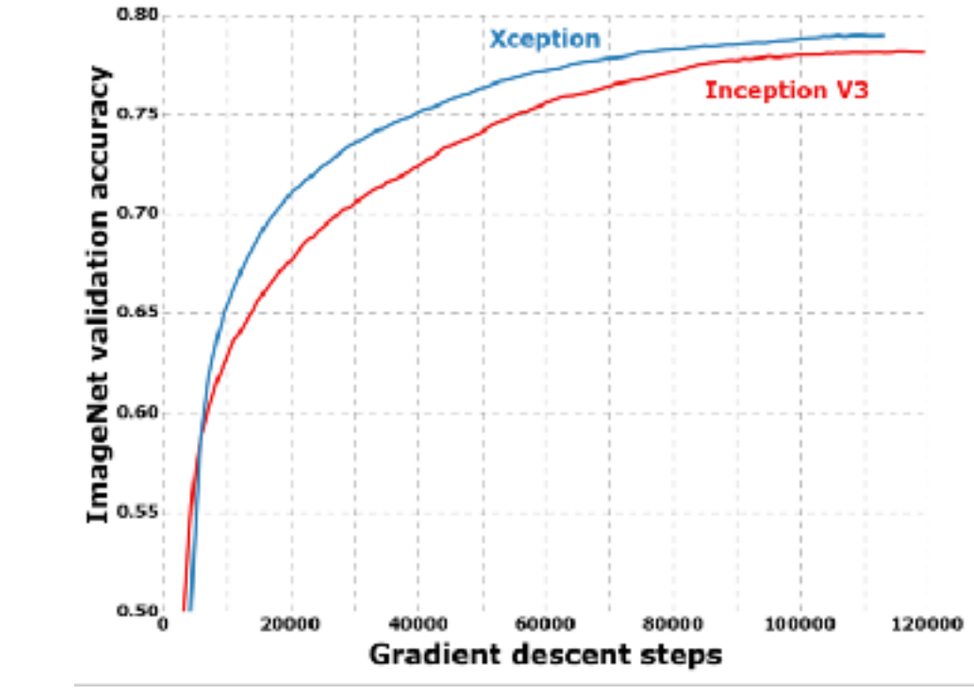
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_i}\frac{\partial o_i}{\partial s_i}\frac{\partial s_i}{\partial w_{ij}} = \frac{\partial E}{\partial o_i} K'(s_i)\, o_j$$

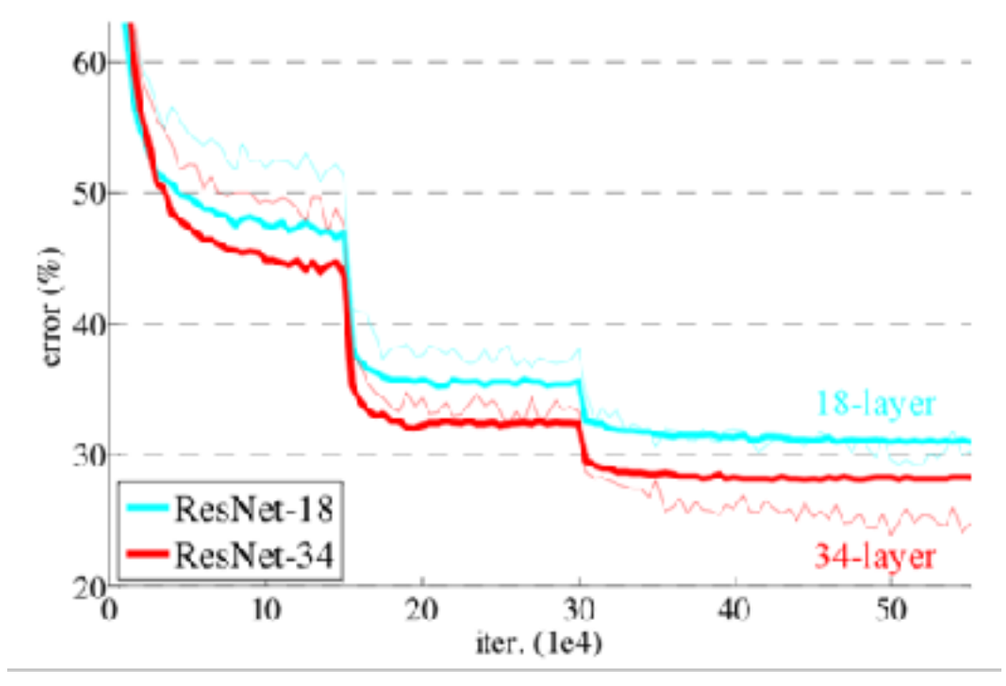
$$\sum_{l \in R} \frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial o_i} = \sum_{l \in R} \frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial s_l}\frac{\partial s_l}{\partial o_i} = \sum_{l \in R} \frac{\partial E}{\partial o_l} K'(s_l) w_{li}$$



# TRAINING NEURAL NETWORKS

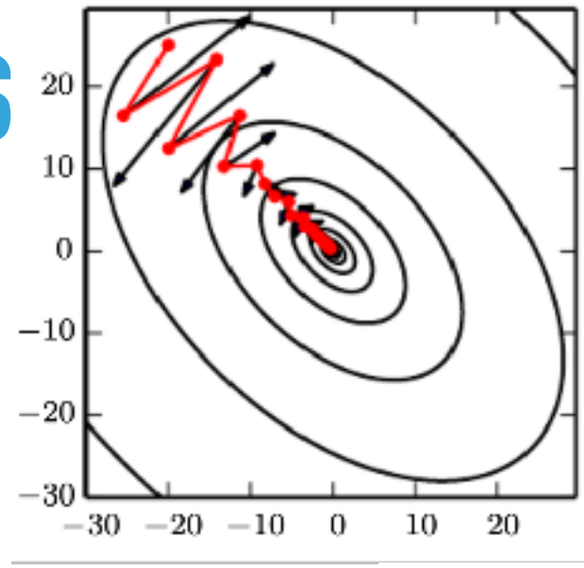
- Instead of full gradient, stochastic gradient (SGD): Gradient is only calculated from a few examples - a minibatch - at a time (1-512 samples usually, but can be larger)
- Larger batches estimate the garden more accurately, but take longer: the efficiency diminishes!
- Small batches can have regularisation effect! (noisy gradients)
- 1 full pass over the whole training dataset is called an *epoch*
- Stochasticity: order of data points. Shuffle in each epoch, to provide more variation.
  - Can not be read from disk via streaming! Fast random access memory is essential with powerful GPUs: large memory or SSD
- Note: use permutations of data, not random sampling, in order to use the whole dataset in a balanced way!
- Note: online training, can basically handle **unlimited data**!



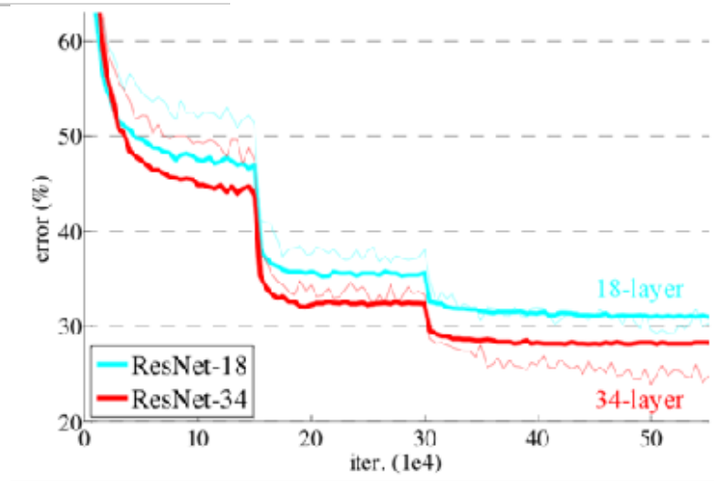
# TRAINING NEURAL NETWORKS

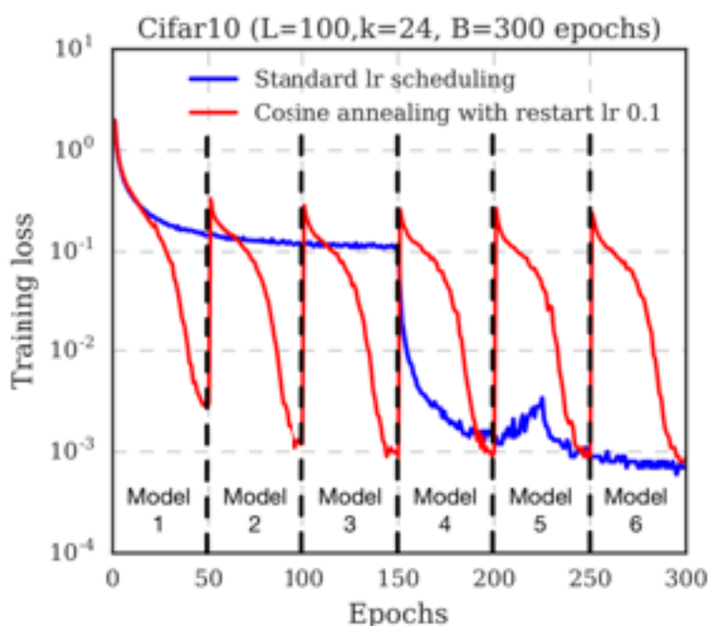
- Local minima? Nah
- Momentum
  - Noisy gradients, keep going!
- Decaying learning rates:Large initial steps, smaller ones close to a good solution to not jump around the minima. Similarly to simulated annealing.
  - Decay, step-decay, cyclic decay (figs)

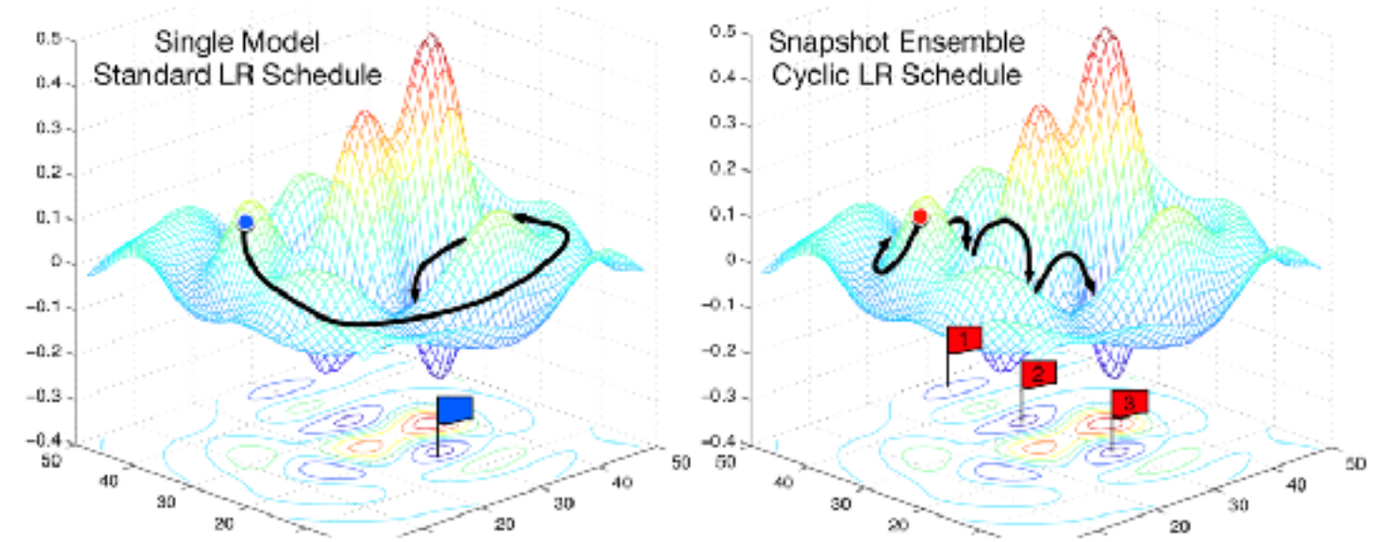



$$\alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[ \frac{1}{m} \sum_{i=1}^{m} L\left( \boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)} \right) \right]$$

$$\alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left( \frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$$



# WEIGHT INITIALISATION

- How to chose initial parameters?
- Full 0? Each weight has the same, and not meaningful gradients. Random!
- Uniform or Gauss? Both Ok.
- Mean? 0
- Scale?
  - Serious problems with exploding training or slow convergence for bad values!)
  - Avoid exploding or vanishing passes, (forward backward too)
  - For ReLU: variance = 2/(n_inputs) (He et al 2015)
  - Some variations for tanh, sigmoid or linear non-linearities
- Even in 2014 they trained a 16 layer neural networks with greedy layer-wise pre training, because of exploding gradients. Then they realised these simple schemes allow training from scratch!

$$\mathbf{y}_l = \mathrm{W}_l \mathbf{x}_l + \mathbf{b}_l.$$

$$Var[y_l] = n_l Var[w_l x_l],$$

$$Var[y_l] = n_l Var[w_l] E[x_l^2].$$

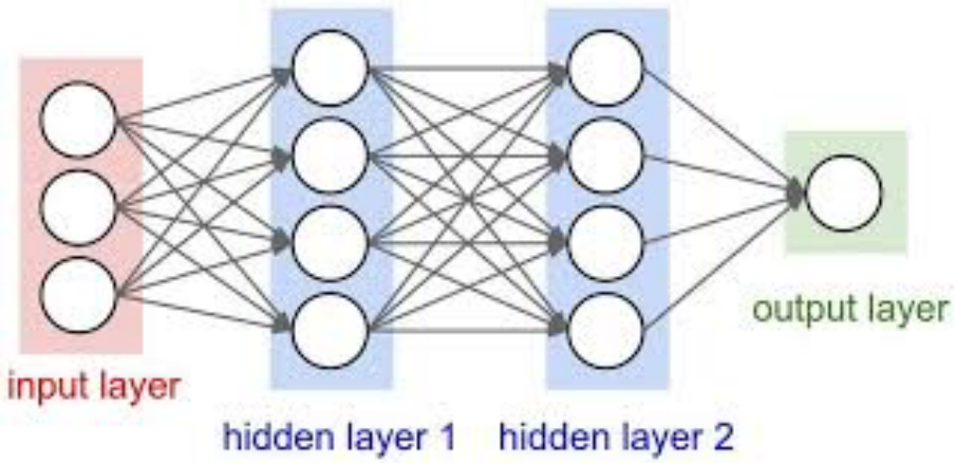
$$Var[y_l] = \frac{1}{2} n_l Var[w_l] Var[y_{l-1}].$$

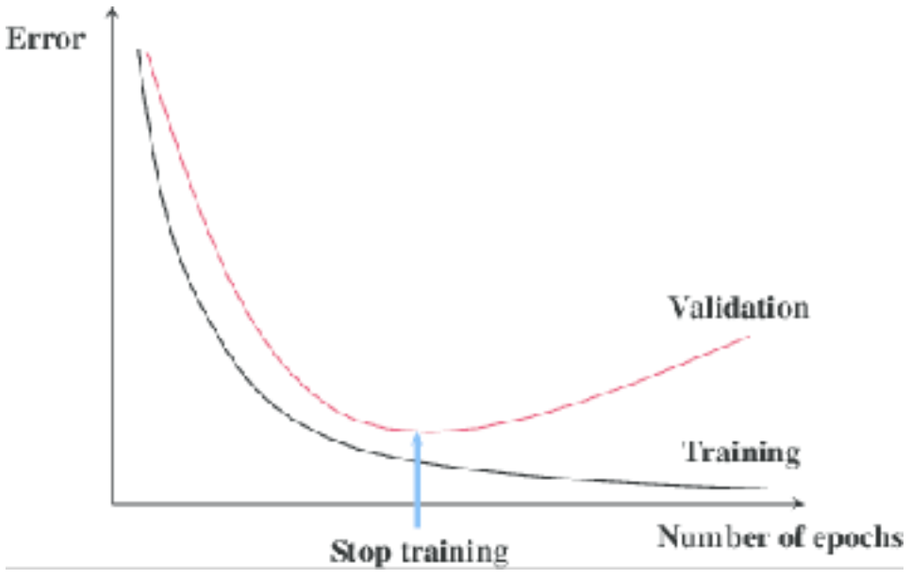
$$Var[y_L] = Var[y_1] \left( \prod_{l=2}^{L} \frac{1}{2} n_l Var[w_l] \right)$$

$$\frac{1}{2} n_l Var[w_l] = 1, \quad \forall l.$$

# REGULARISATION IN NEURAL NETWORKS, EARLY STOPPING

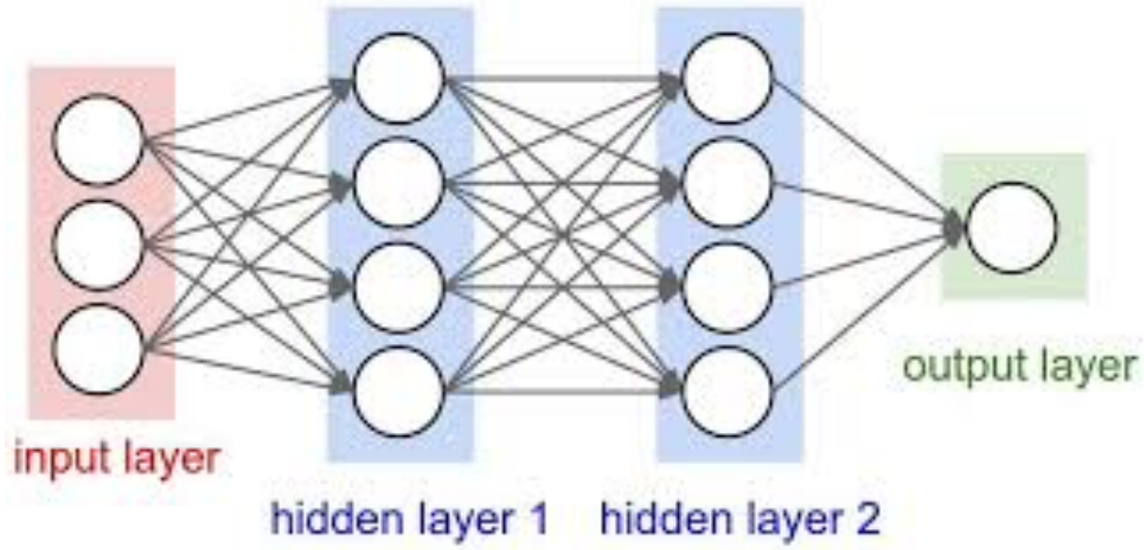
- Neural networks with many units and layers can easily memorise any data
- (modern image recognition networks can memorise 1.2 million, 224x224 pixel size, fully random noise images)
- L2 penalty of weights can be useful but still!
- How long should we train? “Convergence” is often 0 error on training data, fully memorised.
- Early stopping: Train-val-test splits, and stop training when error on validation does not improve. (Train-test only split will “overfit” the test data)!
- Early stopping is a regularisation! It does not improve training accuracy, but it does improve testing accuracy. It is essentially a limit, how far we can wander from the random initial parameter point.

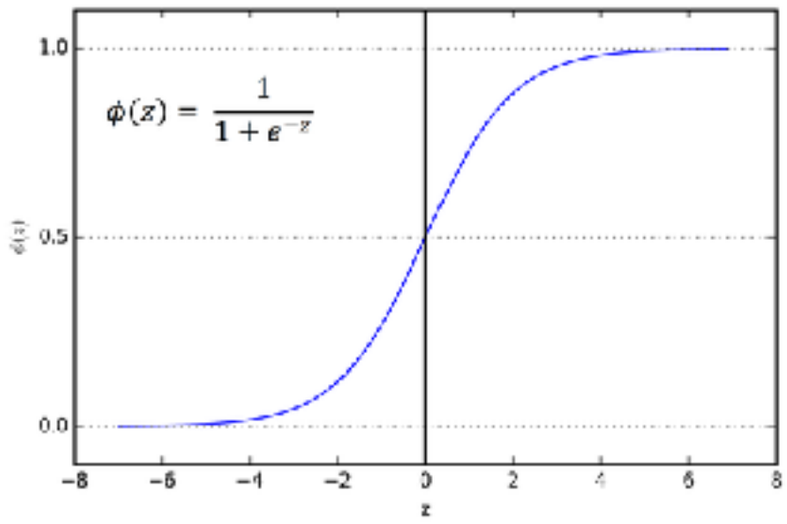


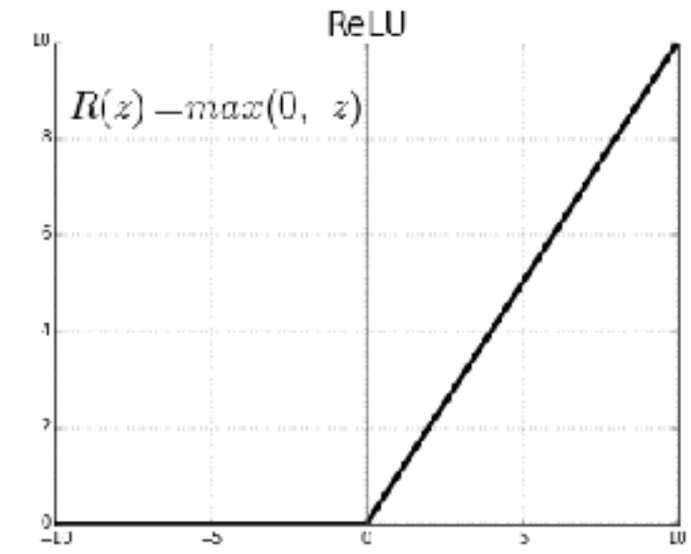



# RELU VS SATURATING UNITS

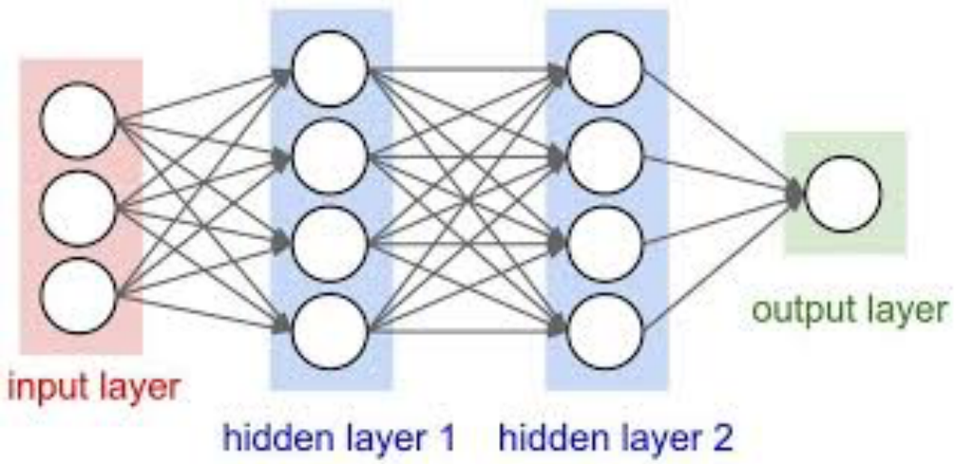
- Small, few layer neural networks generally use saturating units: sigmoid, tanh
- With many more layers it becomes a problem
  - The gradient of a function composition is the product of each gradient
    - When an activation is large: the  saturating units have small gradient
    - Product of 10+ gradients: vanishing gradients at the first layers!
    - Solution: non-linear function which is linear above 0: ReLU constant gradients at high activations!
- Recurrent neural networks also use saturating units!



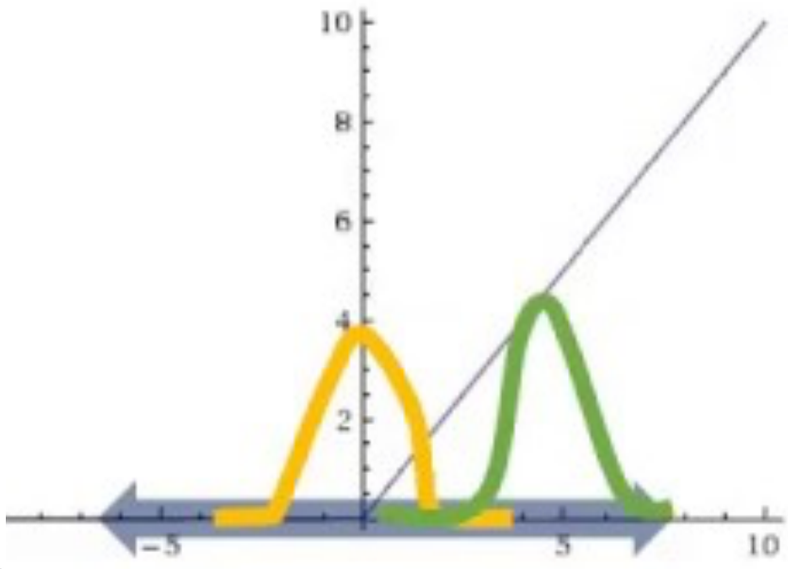



# BATCH NORMALISATION




- After each gradient step weights are changed
- Due to the change in weights, the distribution of inputs to a neuron change in scale, and it shifts
  - *Internal covariate shift*
- A neuron has to accommodate to a new distribution in every single step
  - Creates exploding gradients, slows down training
- Normalise (0 mean, unit variance) (and scale) the inputs to each neuron. Use batch samples for normalisation.
  - Rule of thumb: batch size >= 16
- Even 100x larger learning rates do not explode. Radically speeds up training
- Improves generalisation too! Different mini batch sample in each epoch. Large initial steps



# REFERENCES

- ESL chapter 11.
- https://www.deeplearningbook.org