Course 395: Machine Learning - Lectures

- Lecture 1-2: Concept Learning (M. Pantic)
- Lecture 3-4: Decision Trees & CBC Intro (M. Pantic & S. Petridis)
- Lecture 5-6: Evaluating Hypotheses (S. Petridis)
- Lecture 7-8: Artificial Neural Networks I (S. Petridis)
- ➢ Lecture 9-10: Artificial Neural Networks II (S. Petridis)
- Lecture 11-12: Instance Based Learning (M. Pantic)
- Lecture 13-14: Genetic Algorithms (M. Pantic)

Output Weights Update Rule: Example

- Update rule for output units: $\Delta w_{kj} = -\eta \frac{\partial E}{\partial o_k} \frac{\partial \sigma(net_k)}{\partial net_k} y_j$

- Error function $E = \frac{1}{2}\sum_{k=1}^{K}(t_k - o_k)^2$

- $\frac{\partial E}{\partial o_k} = -(t_k - o_k)$

- $\frac{\partial \sigma(net_k)}{\partial net_k} = \sigma(net_k)(1 - \sigma(net_k)) = o_k(1 - o_k)$

  when σ is sigmoid

Output Weights Update Rule: Example

- $\Delta w_{kj} = -\eta \frac{\partial E}{\partial o_k} \frac{\partial \sigma(net_k)}{\partial net_k} y_j = \eta(t_k - o_k) o_k (1 - o_k) y_j$
- When the output is 0 or 1 then Δw is 0 as well
- No matter if our prediction is right or wrong Δw will be 0 if the output is either 0 or 1
- When the output activation function is sigmoid it is not a good idea to use the quadratic error function
- See http://neuralnetworksanddeeplearning.com/chap3.html

Cross Entropy Error as Error Function

- A good error function when the output activation functions are sigmoid is the binary cross entropy defined as follows:

$$E = -\sum_{k=1}^{K}\left(t_k \ln o_k + (1-t_k)\ln(1-o_k)\right)$$

- $\Delta w_{kj} = -\eta \frac{\partial E}{\partial o_k} \frac{\partial \sigma(net_k)}{\partial net_k} y_j$
- $\frac{\partial E}{\partial o_k} = \frac{o_k - t_k}{o_k(1-o_k)}$
- $\frac{\partial \sigma(net_k)}{\partial net_k} = \sigma(net_k)(1-\sigma(net_k)\ ) = o_k(1-o_k)$

Cross Entropy Error as Error Function

- $\Delta w_{kj} = -\eta \frac{\partial E}{\partial o_k} \frac{\partial \sigma(net_k)}{\partial net_k} y_j$
- $\Delta w_{kj} = -\eta \frac{o_k - t_k}{o_k(1-o_k)} o_k(1 - o_k) y_j = \eta(t_k - o_k)\, y_j$
- The higher the error the higher the weight update

Softmax output activation functions

- A popular output activation function for classification is softmax $o_k = \frac{e^{net_k}}{\sum_k e^{net_k}}$
- The output can be interpreted as a discrete probability distribution
- The right error function is the negative log likelihood cost
  E = $-\sum_k t_k ln o_k$
- Target vectors = [0 0 1 … 0] → E = $-ln o_L$ where L is the position of the active target, i.e., it is 1.

Output activation functions: Summary

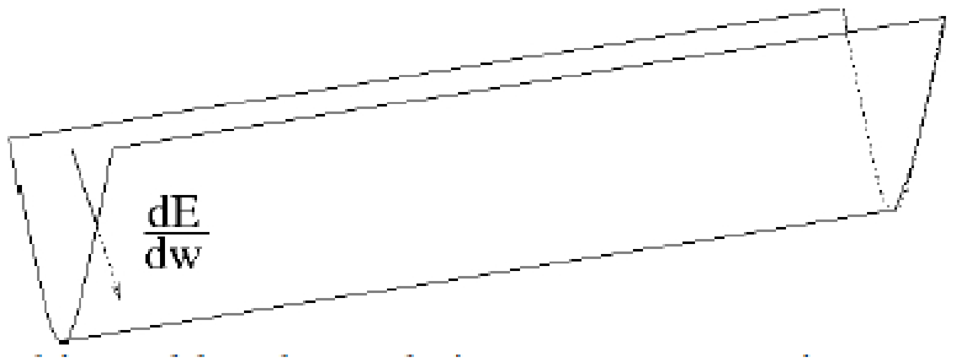
- For each output activation function the right error function should be selected
- Sigmoid → Cross entropy error (useful for classification)
- Softmax → negative log likelihood cost (useful for classification)
- Both combinations work well for classification problems, Softmax has the advantage of producing a discrete probability distribution over the outputs
- Linear → Quadratic loss (useful for regression)

SGD with momentum

- Standard backpropagation

$$w_i \leftarrow w_i + \Delta w_i \quad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

- If the error surface is a long and narrow valley, gradient descent goes quickly down the valley walls, but very slowly along the valley floor.



From https://www.cs.toronto.edu/~hinton/csc2515/notes/lec6tutorial.pdf

SGD with momentum

- Standard backpropagation

$$w_i \leftarrow w_i + \Delta w_i \quad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

- Backpropagation with momentum

$$\Delta w_i(t) = \mu\, \Delta w_i(t-1) + (1-\mu)\left(-\eta \frac{\partial E}{\partial w_i(t)}\right) \textbf{ OR}$$

$$\Delta w_i(t) = \mu\, \Delta w_i(t-1) + \left(-\eta \frac{\partial E}{\partial w_i(t)}\right)$$

- $\mu = \text{momentum constant, usually } 0.9, 0.95$

- It is like giving momentum to the weights

- We do not take into account only the local gradient but also recent trends in the error surface

Other Training Algorithms

- Adam (usually works quite well)
- Adagrad
- Adadelta
- RMSprop
- Nesterov momentum
- …and others

Learning Rate Decay

- In the beginning weights are random so we need large weight updates, then as training progresses we need smaller and smaller updates.

- It’s a good idea to start with a “high” (depends on the problem/dataset) learning rate and decay it slowly.

- Typical values for initial learning rate, 0.1, 0.01. It’s problem dependent

- Step decay: Reduce the learning rate by some factor every few epochs, e.g., divide by 2 every 50 epochs

Learning Rate Decay

- Keep learning rate constant for T epochs and then decrease as follows:$lr_t = \frac{lr_0 * T}{\max(t,T)}$

- Keep learning rate constant for T epochs and then decrease as follows: lr_t =$lr_{t-1} * scalingFactor$ (e.g. 0.99)

- Decrease as follows: $lr_t = \frac{lr_0}{1+\frac{t}{T}}$, T is the epoch where the learning rate is halved

- You can think of many other ways to decay the learning rate

Momentum

- It's usually a good practice to increase the momentum during training.
- Typically the initial value is 0.5 and the final value is 0.9, 0.95
- Increase is usually linear
- It's also common to start increasing the momentum when the learning rate starts decreasing.

Weight Initialisation

- We said we start with random weights…but how?
- Some of the most common weight initialisation techniques are the following:

1. Sample from a gaussian distribution, we need to define mean (usually 0) and standard deviation (e.g. 0.1 or 0.01)
2. Sample from a uniform distribution, we need to define the range [-b,b]
3. Sparse initialisation: Use gaussian/uniform distributions to initialise weights and then set most of them to 0. You need to define sparsity level, e.g. 0.8 (80% weights in each layer are set to 0).

Weight Initialisation

4. Glorot Initialisation: Sample from a gaussian distribution with 0 mean and st. dev. = $\sqrt{2/(n1+n2)}$

   - n1, n2 are the number of neurons in the previous and next layers, respectively.

   - Glorot, Bengio, Understanding the difficulty of training deep feedforward neural networks, JMLR, 2010

Weight Initialisation

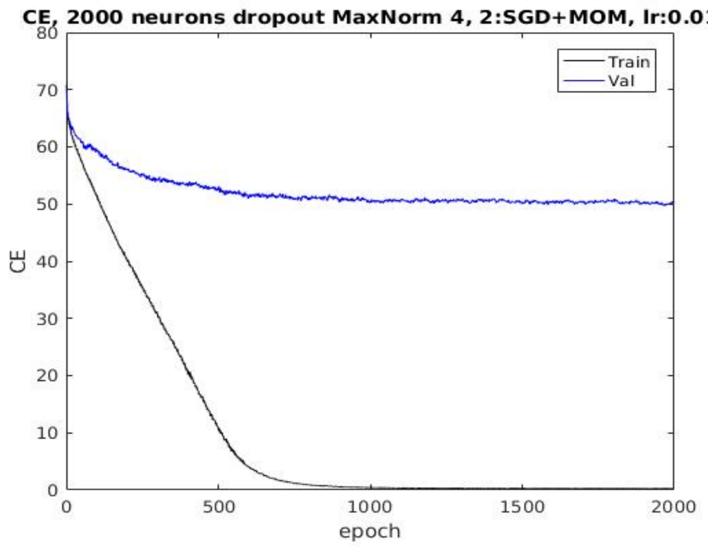
5. He Initialisation: Sample from a gaussian distribution with 0 mean and st. dev. = $\sqrt{2/n1}$
   - n1 is the number of inputs to the neuron (i.e. the size of the previous layer).
   - Designed for neurons which use ReLu as activation functions.
   - He et al., Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, ICCV 2015

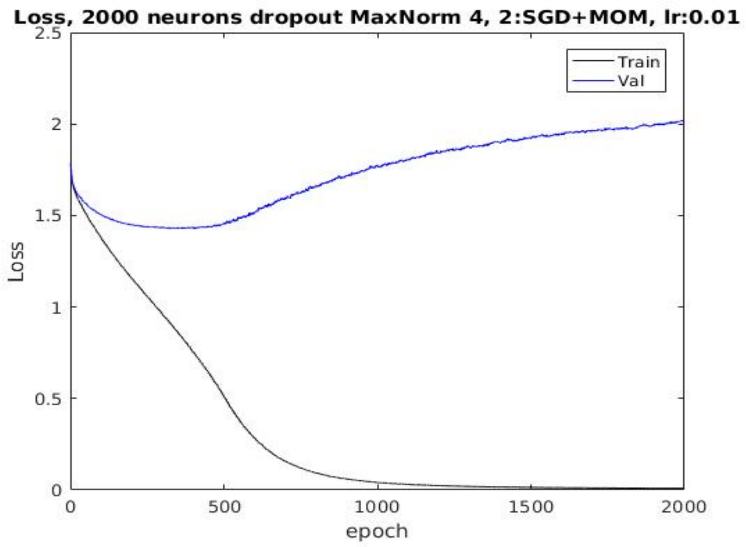
6. You can find many other approaches in the literature

Ways to avoid overfitting

- Early stopping (see slide 55, part 1)
- L1 Regularisation
- L2 Regularisation
- Dropout
- Max-norm Constraint
- Data augmentation

Early Stopping





- Early stopping: should we use loss or Classification error?
- It’s common that classification error can go down while the loss goes up!

L2 Regularisation

- $E = E_0 + \lambda \sum_{all\ Weights} w^2$
- E_0 is the original error function, e.g., quadratic loss, negative log-likelihood
- It is NOT applied to the bias
- We wish to minimise the original error function (E_0)
- We also wish to penalise large weights, keep the weights small (second term)
- Small λ → we prefer to minimise E_0
- Large λ → we prefer small weights

L1 Regularisation

- $E = E_0 + \lambda \sum_{all\ Weights} |w|$
- E_0 is the original error function, e.g., quadratic loss, negative log-likelihood
- It is NOT applied to the bias
- We wish to minimise the original error function (E_0)
- We also wish to penalise large weights, keep the weights small (second term)
- Small λ → we prefer to minimise E_0
- Large λ → we prefer small weights

L1/L2 Regularisation

- So what’s the difference between L1 and L2 regularisation?

- L2: $\frac{\partial E}{\partial w} = \frac{\partial E_0}{\partial w} + \lambda w \rightarrow \Delta w = -\eta \frac{\partial E_0}{\partial w} - \eta\lambda w$
- L1: $\frac{\partial E}{\partial w} = \frac{\partial E_0}{\partial w} + \lambda sign(w) \rightarrow \Delta w = -\eta \frac{\partial E_0}{\partial w} - \eta\lambda sign(w)$

- L1: The weights shrink by a constant amount towards 0

- L2: The weights shrink by an amount proportional to w

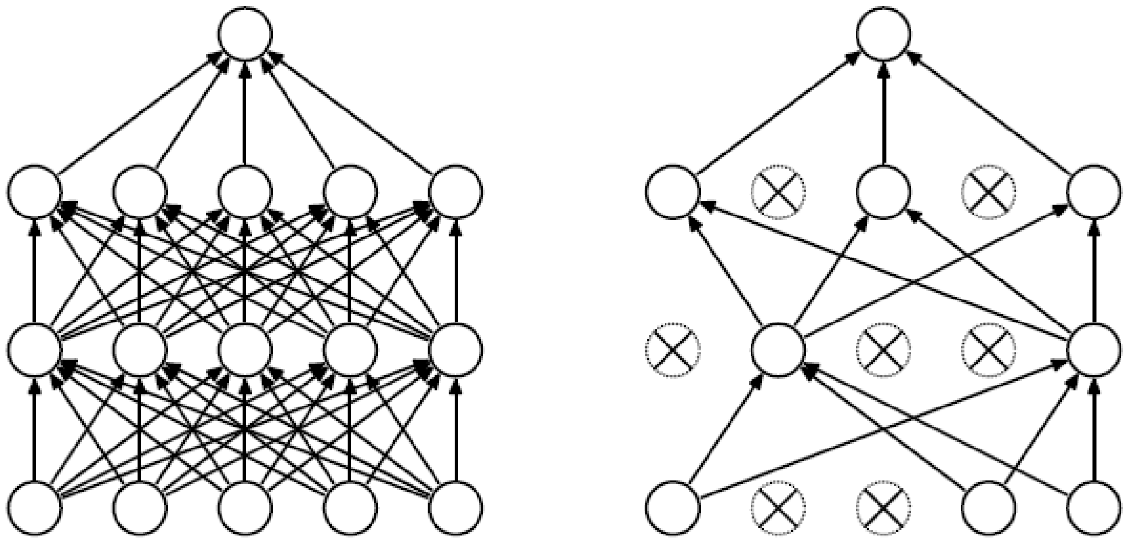
- L1 drives small weights to zero

L1/L2 Regularisation

- Why small weights prevent overfitting?
- When weights are 0 or close to zero this equivalent to removing the corresponding connection between the neurons
- Simpler architecture → avoids overfitting
- Network has the right capacity
- It is like we start with a high capacity (complex) network until we find a network with the right capacity for the problem

Dropout

- We don’t modify the error function but the network itself
- During training neurons are randomly dropped out
- The probability that a neuron is present is p



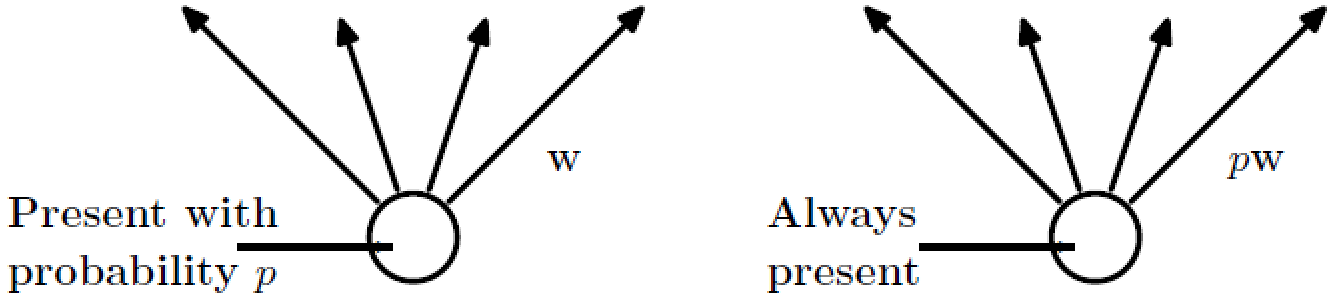
(a) Standard Neural Net (b) After applying dropout.

From Dropout: A simple way to prevent neural networks from overfitting by Srivastava et al., JMLR 2014

Dropout

- Dropout prevents overfitting because it provides a way of approximately combining exponentially many different neural network architectures.

- Typical values for p: 0.8/0.5 for input/hidden neurons
- At test time the outgoing weights of a neuron are multiplied by p



(a) At training time (b) At test time

From Dropout: A simple way to prevent neural networks from overfitting by Srivastava et al., JMLR 2014

Dropout - Tips

- If a network with n neurons in the hidden layer works well for a given task then a good dropout network should have n/p neurons.

- Dropout introduces a significant amount of noise in the gradients, a lot of gradients cancel each other → you should use higher learning rate (and maybe higher momentum)

- More epochs are needed

- The above heuristics do not always work!

Max-Norm Regularisation

- Constrain the norm of the incoming weight vector at each hidden unit to be upper bounded by a fixed constant c.
- Weight vector length: $L = \sqrt{{w_{j1}}^2 + {w_{j2}}^2 + \ldots + {w_{jN}}^2}$
- w_{ji} corresponds to incoming weights to neuron j from the N neurons of the previous layer
- If L > c then multiply all the incoming weights by c/L
- The new vector length is c
- Another approach to keep the weights small
- Usually used in combination with dropout

Data Augmentation

- One of the best ways to avoid overfitting is more data
- So we can artificially generate more data, usually a bit noisy, so we introduce more variation
- We should apply operations that correspond to real-world variations.
- For images: flip left-right, rotate, translate, etc

Data Normalisation

- It is not desirable that some inputs are orders of magnitude larger than other inputs
- Map each input *x(i)* to [-1/0, +1]
- Min value is mapped to -1/0
- Max value is mapped to 1

Data Normalisation

- Standardize inputs to mean=0 and 1 std. dev.=1

  $$y = \frac{x - x_{mean}}{x_{std}}$$

- Useful for continuous inputs/targets

- It’s called z-normalisation

- Scaling is needed if inputs take very different values. If e.g., they are in the range [-3, 3] then scaling is probably not needed

Data Normalisation

- $x_{mean,}\, x_{std}$ are computed on the training set and then applied to the validation and test sets.

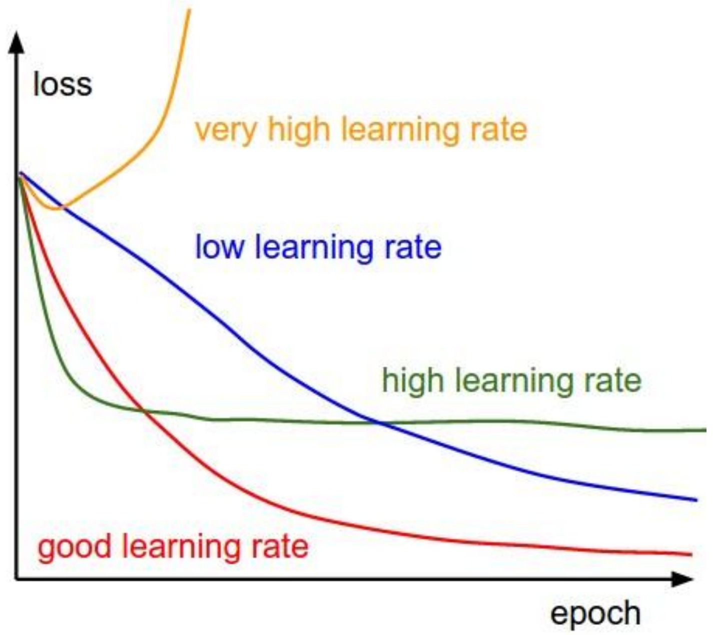
- It is not correct to normalise each set separately.

Image Normalisation

- When the input data are images then you can simply remove the mean image computed on the training set.

- Alternatively, you can compute the mean and standard deviation of all the pixels in each image and z-normalise each image independently.

Monitoring the learning process



From http://cs231n.github.io/neural-networks-3/

- If loss increases or oscillates then the learning rate is too high

- If loss goes down slowly the the learning rate is low

- Find a learning rate value at which the loss on the training data immediately begins to decrease.
- It's a good idea to turn off regularisation at this point

Monitoring the learning process Other tips

- Compute the mean and standard deviation of hidden neurons activations for all examples in a mini-batch
- They should be different than 0 (this is important when ReLu is used since the neurons can easily die)
- For each layer compute the norm of the weights and the norm of the weight updates Δw.
- The ratio norm(Δw) / norm(w) should be 0.01 – 0.0001
- If ratio is significantly different then something could be wrong

Hyperparameter Optimisation

- Once a good initial learning rate value is found then we can optimise the hyperparameters on the validation set
- Network architecture: number of layers, number of neurons per layer.
- Learning rate: when to start decaying, type of decay
- Regularisation: type of regularisation, values for regularisation parameters
- Training algorithm, SGD+Momentum, Adam, RMSprop
- Maybe we wish to optimise again the initial learning rate

(Hyper)Parameters / Weights

- (Hyper)Parameters are what the user specifies, e.g. number of hidden neurons, learning rate, number of epochs etc
- They need to be optimised
- Weights: They are also parameters but they are optimised automatically via gradient descent

Vanishing/Exploding gradient

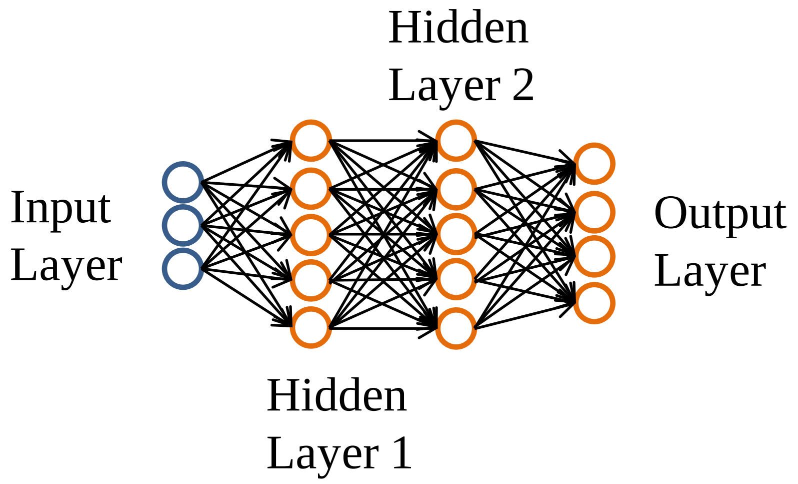
$$\Delta \boldsymbol{w_{ji}} = -\boldsymbol{\eta} \frac{\boldsymbol{\partial E}}{\boldsymbol{\partial w_{ji}}} = -\boldsymbol{\eta \delta_j x_i} \; where \;\; \boldsymbol{\delta_j} = \sum_{\boldsymbol{k=1}}^{\boldsymbol{K}} (\boldsymbol{\delta_k w_{kj}}) \frac{\boldsymbol{\partial \sigma(net_j)}}{\boldsymbol{\partial net_j}}$$

- As we backpropagate through many layers:

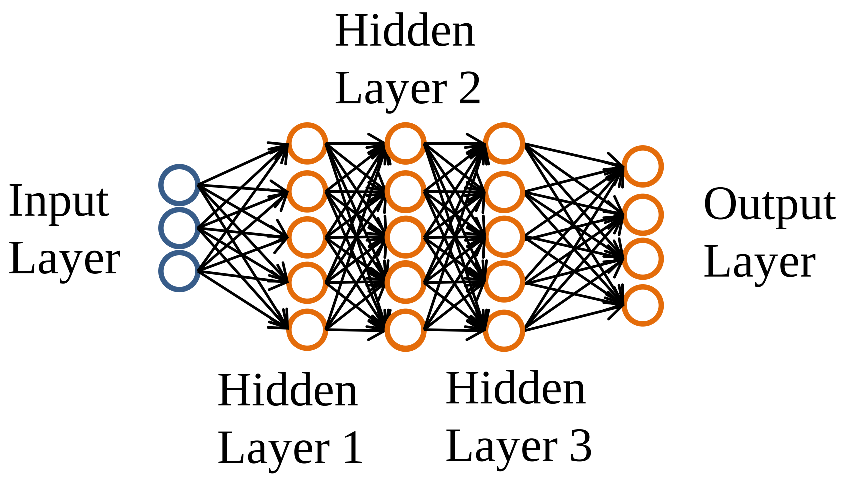
1. If the weights are small -> δ_i shrink exponentially
2. If the weights are big -> δ_i grow exponentially

- So either the network stops learning (case 1) or becomes unstable (case 2)
- That is why it is not possible to train deep networks with backpropagation

Deep NNs



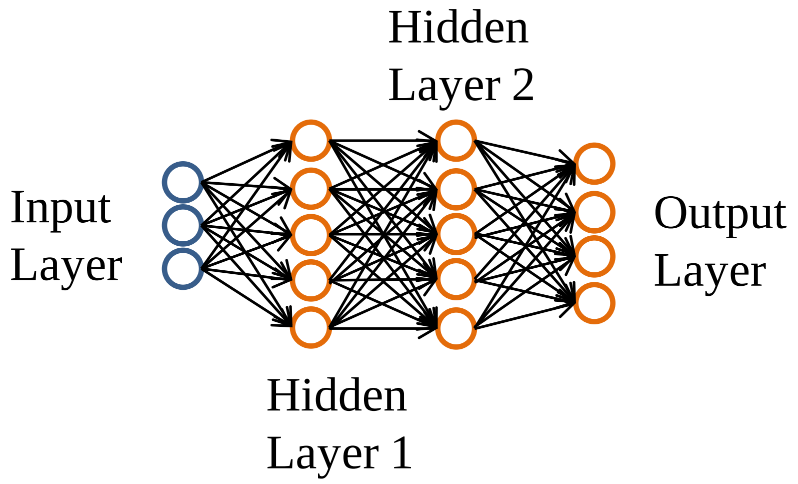
3-layer feed-forward network



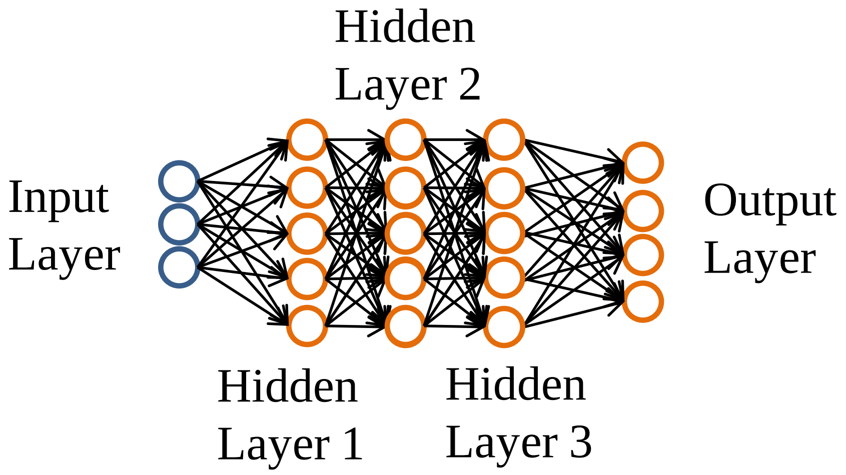
4-layer feed-forward network

- Two ways to train
- A lot of data (data augmentation), ReLu, dropout etc
- Pre-training

Deep NNs



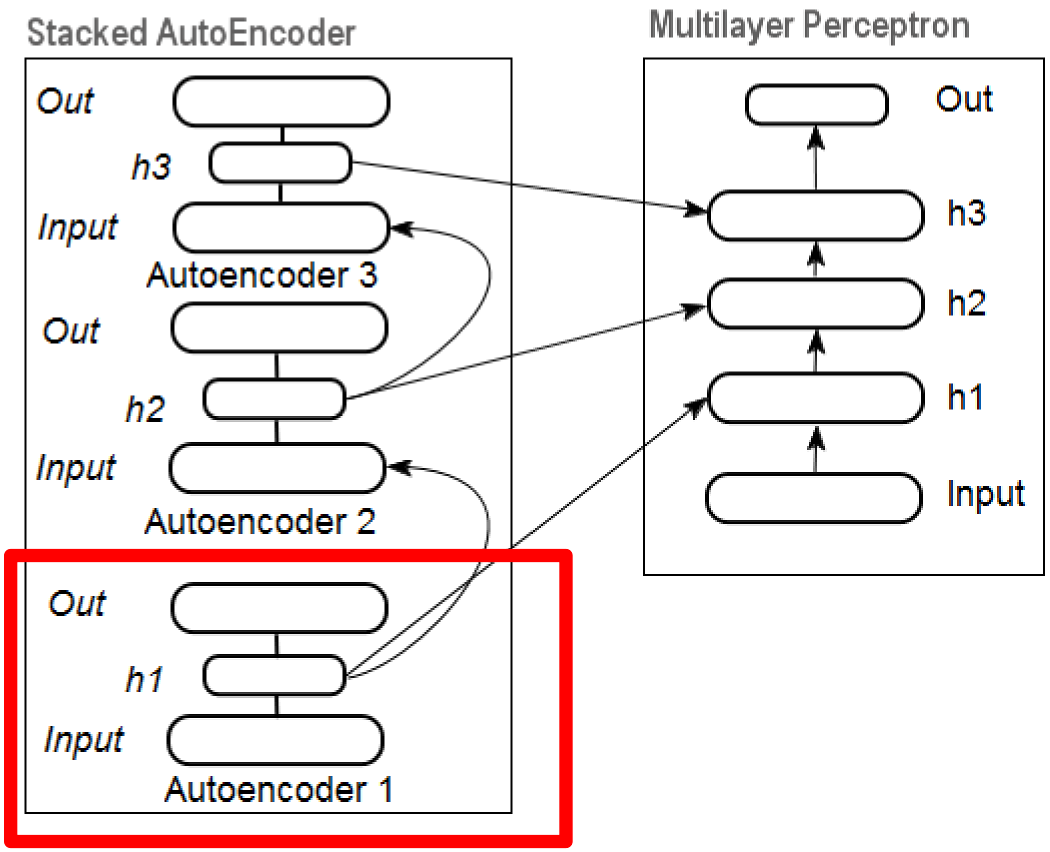
3-layer feed-forward network



4-layer feed-forward network

- There is a pre-training phase where weights are initialised to a good starting point.
- Pre-training is performed per layer using Restricted Boltzmann Machines or Stacked Denoising Autoencoders
- Then backpropagation is used to fine-tune the weights starting from a good initialisation point.

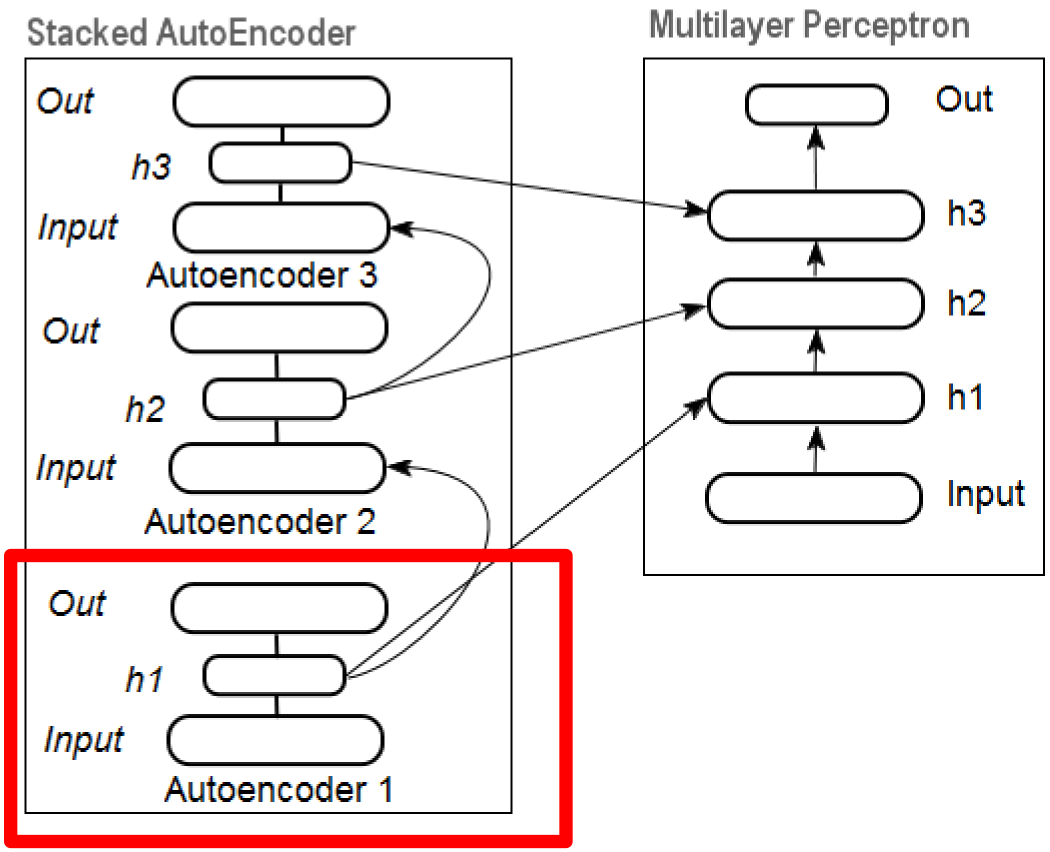
Stacked Denoising Autoencoder



From https://www.mql5.com/en/articles/1103#2_2

- Train a network to reproduce its input
- This network is called an Autoencoder (AE)
- The idea is that the middle layer represents the main variations in the data
- The problem is that the AE may simply learn the identity function

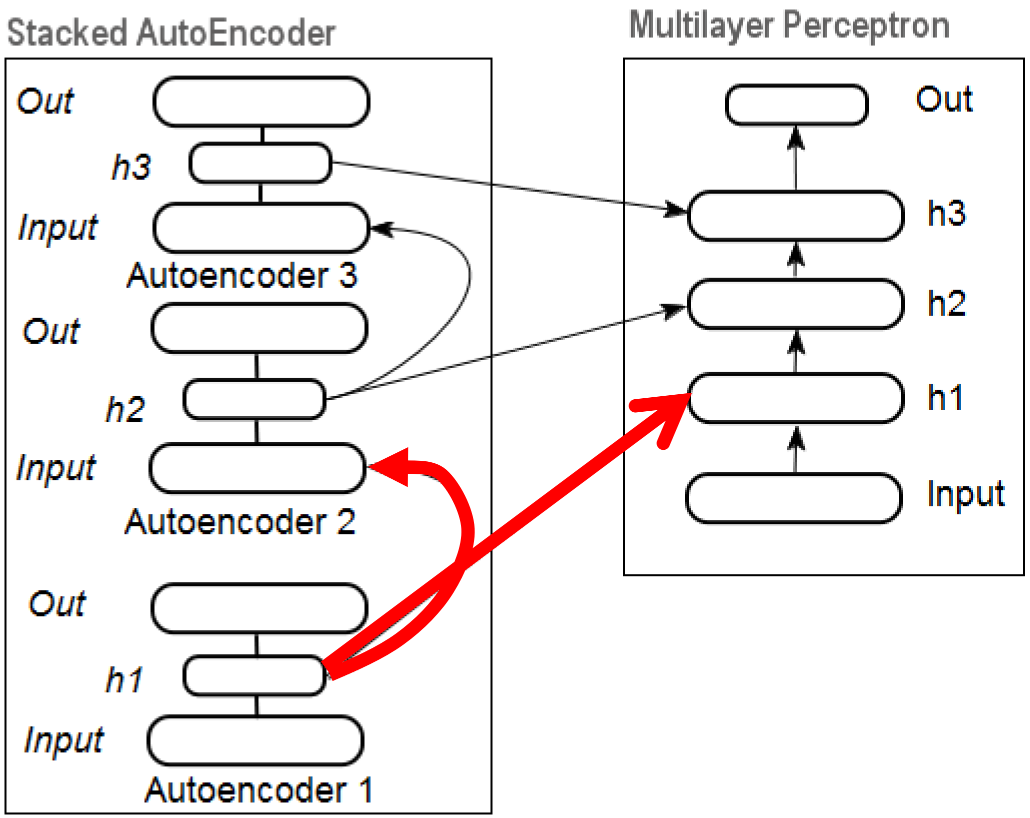
Stacked Denoising Autoencoder



From https://www.mql5.com/en/articles/1103#2_2

- Denoising AE: we add noise to the input so the network learns to reconstruct (output) the “denoised” input
- We usually set as many as half of the inputs to 0
- The network tries to reconstruct the input and undo the effect of noise
- The hidden layer is “forced” to learn the main variations in the data

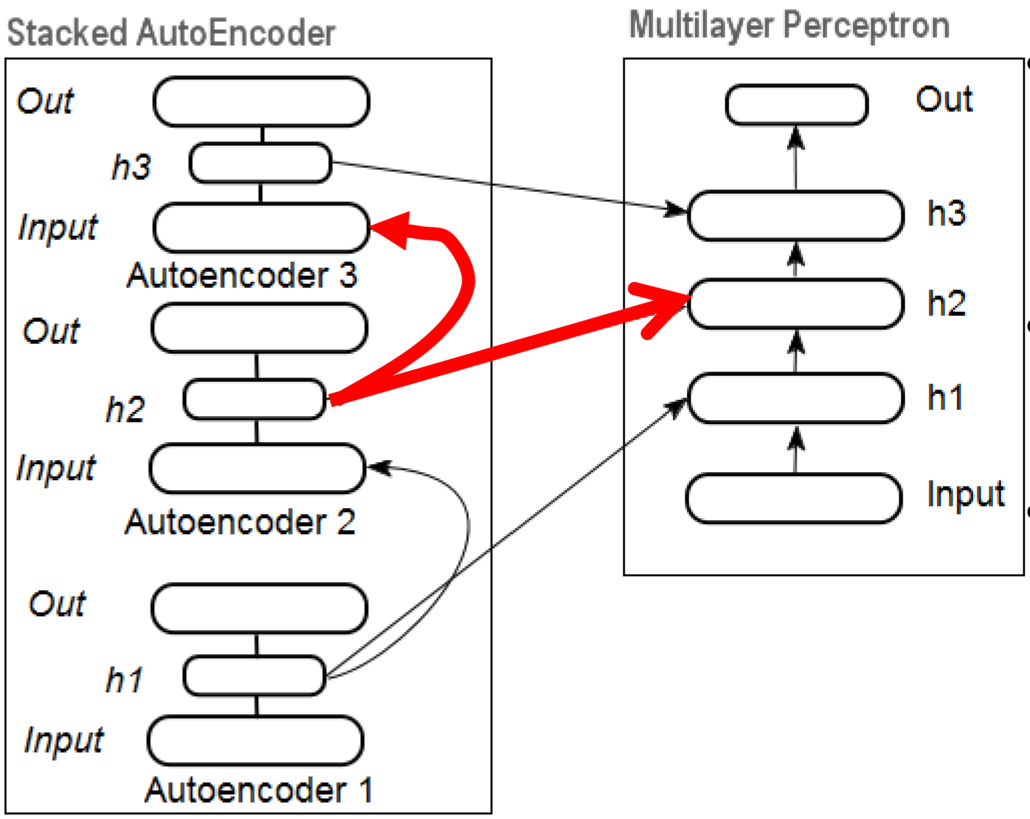
Stacked Denoising Autoencoder



- The hidden layer weights of the AE are copied to the feed-forward NN
- The output of the hidden layer is used as input for the 2nd AE
- Noise is added to this new input and the 2nd AE learns to "denoise" its input

From https://www.mql5.com/en/articles/1103#2_2

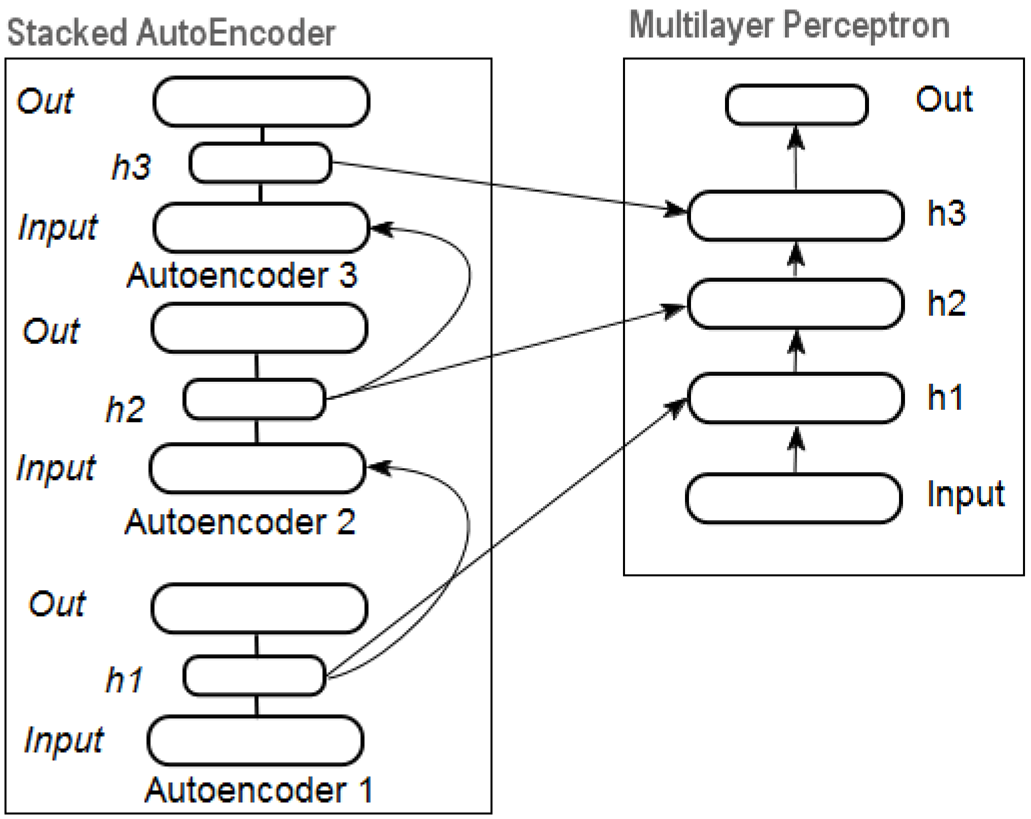
Stacked Denoising Autoencoder



- The hidden layer weights of the 2nd AE are copied to the feed-forward NN
- The output of the hidden layer is used as input for the 3rd AE
- Using this approach we can add as many as layers as we want

From https://www.mql5.com/en/articles/1103#2_2

Stacked Denoising Autoencoder



From https://www.mql5.com/en/articles/1103#2_2

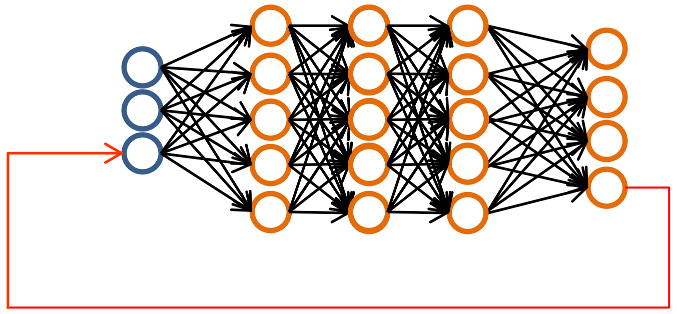
- This approach is used to initialise the NN
- This is called pre-training
- It results in good initialisation of the weights
- Then we fine-tune the network using stochastic gradient descent

Deep Networks for Time Series

- Deep feedforward NNs are good at various tasks but not at handling time series data

- Recurrent Neural Networks are suitable for time series

- They also suffer from the vanishing gradient problem



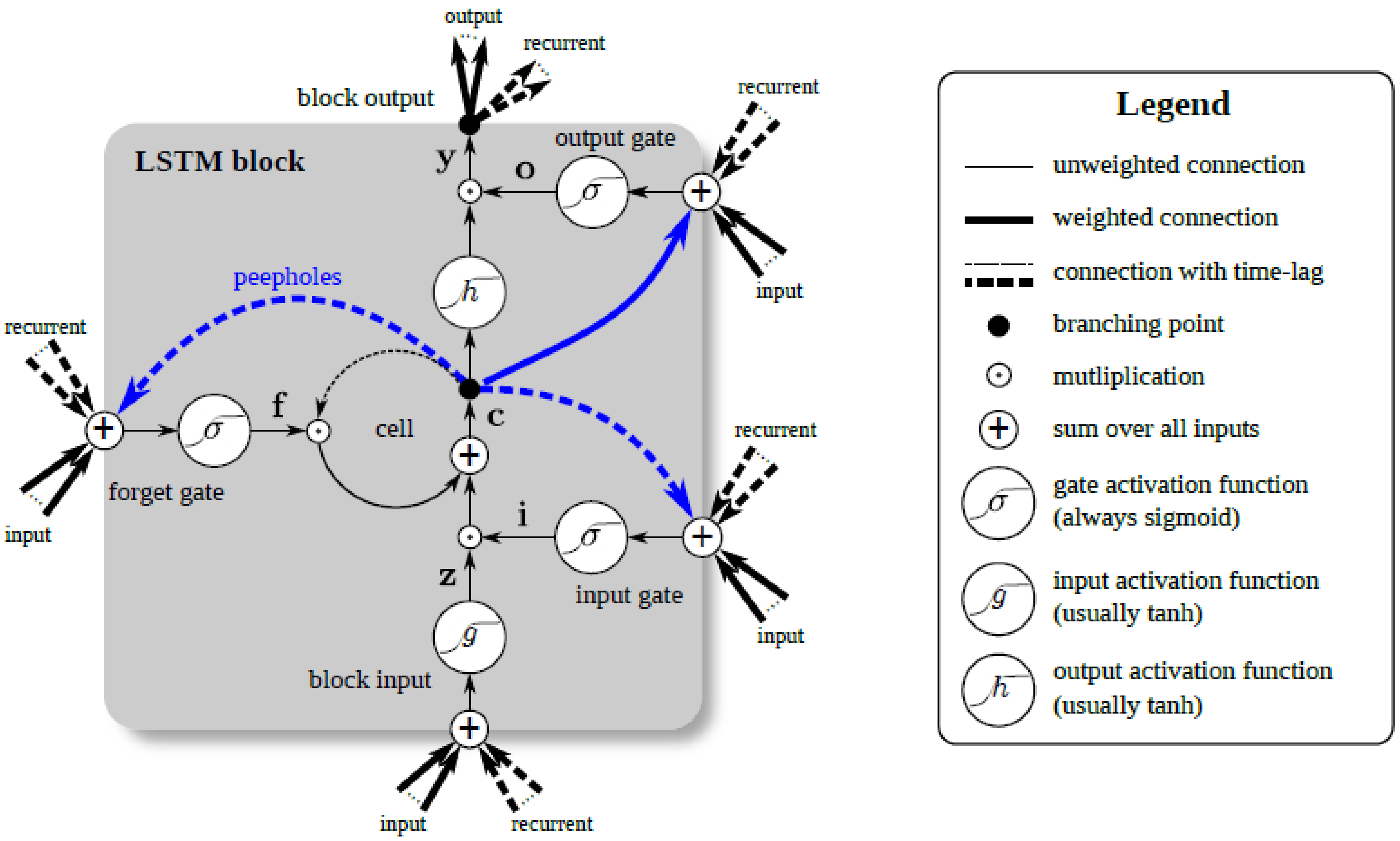
LSTMs

- A type of recurrent network that can be effectively trained is the Long-Short Term Memory Recurrent Neural Network (LSTM-RNN). Introduced in 1990s

- We replace the neuron with a memory cell

- There are input, output and forget gates which control when information flows in / out of the cell and when to reset the state of the cell

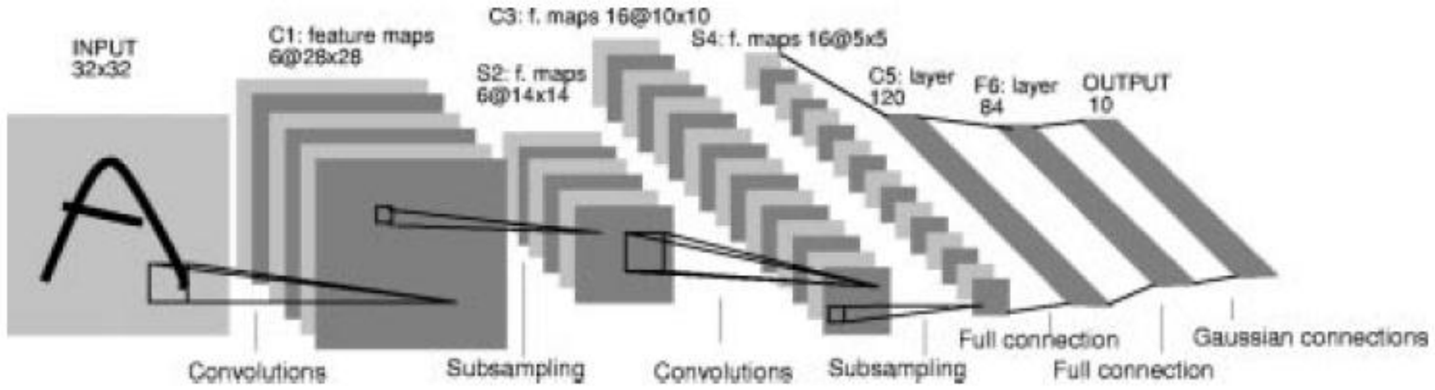
LSTMs



From LSTM: A search space odyssey by Greff et al., arXiv Mar 2015

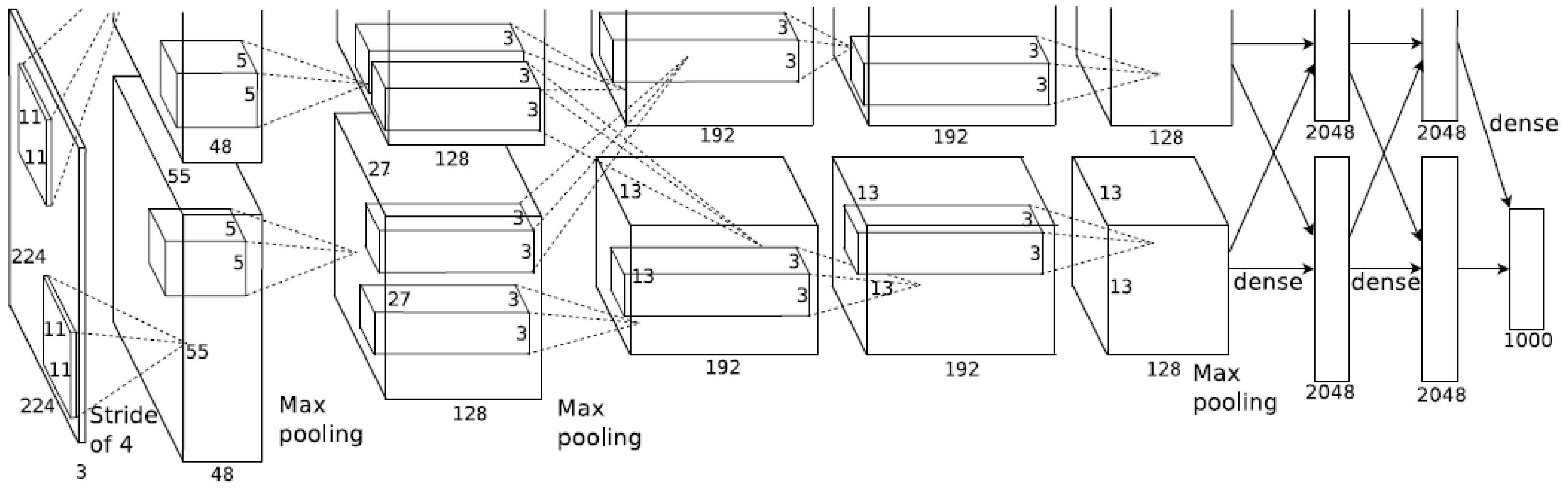
Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) have been very successful in computer vision
- First version was introduced in 1980s (neocognitron)
- Improved by LeCun et al., “Gradient-Based Learning Applied to Document Recognition”, Proc. IEEE, 1998



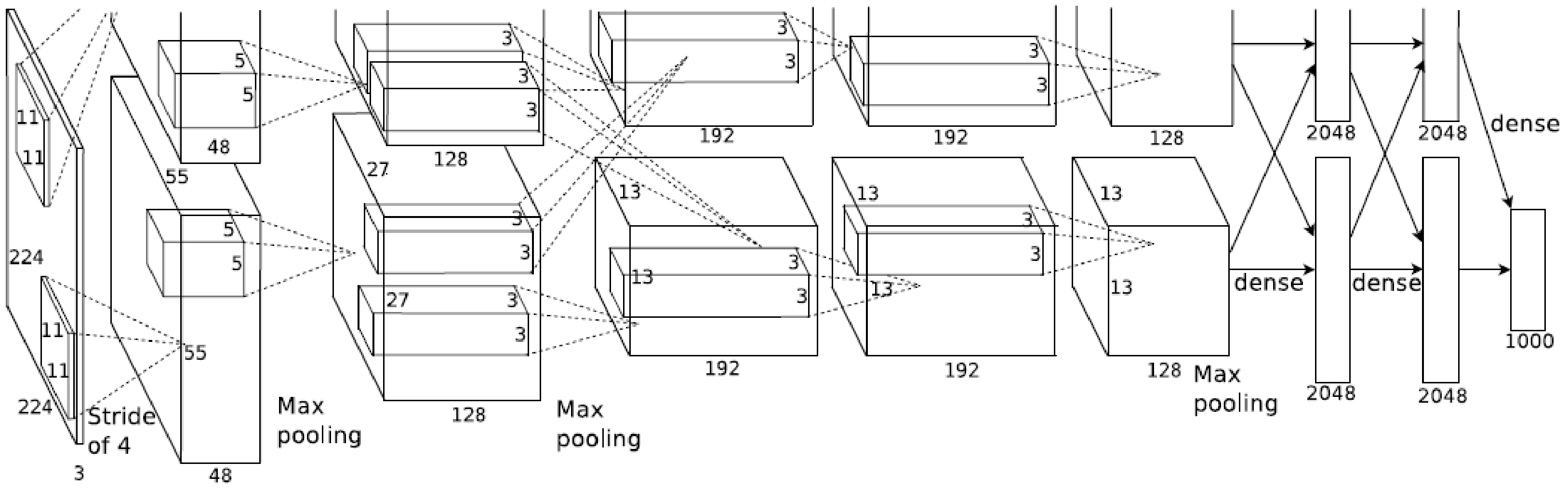
Convolutional Neural Networks

- Became popular in 2012 after winning the ImageNet competition
- "ImageNet Classification with Deep Convolutional Neural Networks", by Krizhevsky et al., NIPS 2012
- Tricks: Data augmentation, Dropout, ReLu + GPUs

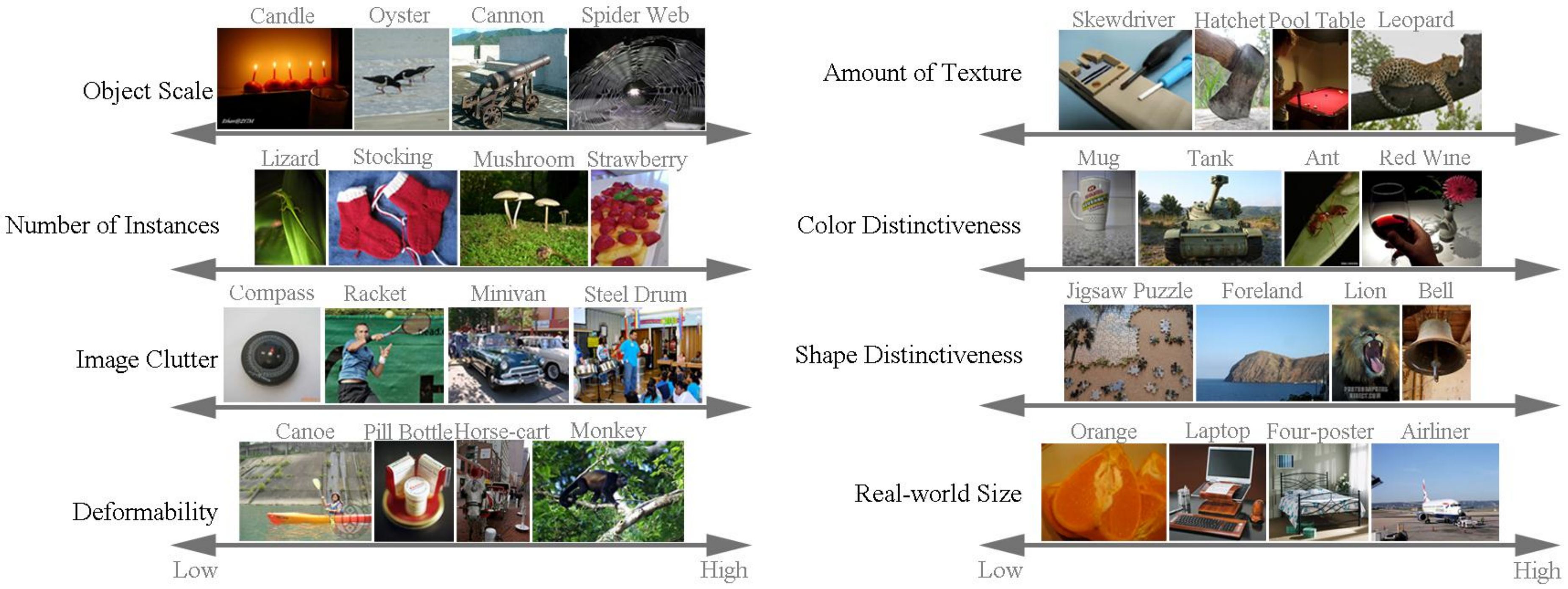


Convolutional Neural Networks

- It's a deep network = many layers
- Each layer is either a convolutional layer or subsampling layer
- Final layers are fully connected layers



ImageNet Competition – Object Classification



- Classification of 1000+ objects
- State-of-the-art before 2012: ~26%
- New state-of-the-art in 2012 with deep networks: ~15%