Unsupervised Learning in Neural Networks

Lecture Notes

Contributors: Fabian Brix, Renato Kempter

December 28, 2013

1 Neuroscience

1.1 Visual Cortex

1.1.1 Primary Visual Cortex

The primary visual cortex is the simplest, earliest cortical visual area. It is highly specialized for processing information about static and moving objects and is excellent in pattern recognition.

1.2 Receptive Fields

The term receptive field originally described an area of the body service where a stimulus could elicit a reflex (Sherrington, 1906). The definition was later extended to sensory neurons defining the receptive field as a restricted region of visual space where a luminous stimulus could drive electrical responses in a retinal ganglion cell: *"Responses can be obtained in a given optic nerve fiber only upon illumination of a certain restricted region of the retina, termed the receptive field of the fiber"* (Hartline, 1938). $http://www.scholarpedia.org/article/Receptive_field$

General definition spanning different types of neurons across sensory modalities: the receptive field is a portion of sensory space that can elicit neuronal responses when stimulated. The sensory space can be defined in a single dimension (e.g. carbon chain length of an odorant), two dimensions (e.g. skin surface) or multiple dimensions (e.g. space, time and tuning properties of a visual receptive field). The neuronal response can be defined as FIRING RATE (i.e. **number of action potentials generated by**

a neuron) or include also subthreshold activity (i.e. depolarizations and hyperpolarizations in membrane potential that do not generate action potentials).

1.2.1 Visual Receptive Fields

ICA 2, discourse receptive fields.

Receptive Field Development

2 Supervised vs. Unsupervised learning

Supervised learning a system is trained for regression or classification via a set of labeled training data. The goal is to optimize the parameters of an objective function using the labeled data (How to correctly predict the labelling "l" or "f" of input patterns according to the training data?). Supervised learning is dealt with in Prof. Seegers class "Pattern Classification and Machine Learning".

Unsupervised learning In unsupervised learning one tries to extract patterns from the data without having any information about their classification/outputs. One therefore tries to detect the intrinsic structure of the data. (How does an hand-written "l" or "f" look like?).

Reinforcement learning In this context an "agent" is employed and given a certain choice of actions and rewards upon chosing the correct action. The actions exist according to training input patterns (animal conditioning: a mouse sees an "l" or an "f" and chooses action a_1/a_2 accordingly and is given a reward upon success). Reinforcement learning can be either conducted in a supervised or an unsupervised manner.

3 Hebbian Learning Rule

change in a given synaptic weight is proportional to both the pre-synaptic input and the output activity of the post-synaptic neuron.

- time-dependent
- local
- strongly interactive

3.1 Synaptic plasticity

change in connection strength

> Hebb, 1949: *"When an axon of cell j repeatedly or persistently takes part in firing (through synapses to dendrites of) cell i, then j's efficiency as one of the cells firing is increased".*

The Hebbian learning rule is a local UNSUPERVISED LEARNING rule, because it depends solely on the states of the respective pre-synaptic i and post-synaptic j neurons and the present efficacy w_{ij}, but not on the state of other neurons k. The repsective pre-synaptic neurons and post-synaptic neuron can however be simultaneously active, so that correlations in their activity lead to firing of the neuronal cell i. In Hebbian learning these correlations are used to learn the synaptic transmission efficacies w_{ij}, so that in essence it is correlation-based learning.

3.2 Homeostatic plasticity

Homeostatic plasticity refers to the capacity of neurons to regulate their own excitability relative to network activity, a compensatory adjustment that occurs over the timescale of days. The term homeostatic plasticity derives from two opposing concepts: "homeostatic" (a product of the Greek words for "same" and "state" or "condition") and plasticity (or "change"), thus homeostatic plasticity means "staying the same through change."

3.3 Rate-based Hebbian Learning

Goal is to find a mathematically formulated learning rule based on Hebb's postulate.
We begin by focusing on a single synapse with efficacy w_{ij} transmitting signals from a pre-synaptic neuron i to a post-synaptic neuron j. The activity ((mean) firing rate - average number of spikes per unit time) of i is denoted by v_i and that of j by v_j. Based on the (1) `locality` aspect of Hebbian Learning we can write a general description of the change of the synaptic efficacy.

$$\frac{d}{dt}w_{ij} = \mathbf{F}(w_{ij}; v_j^{pre}, v_i^{post})$$

The membrane potential is further uniquely defined by the postsynaptic firing rate $v_i = g(u_i)$ with a monotone gain function g and therefore doesn't have to be included in F.
The (2) `cooperativity` aspect of Hebb's postulates implies that pre- and

postsynaptic neuron have to be active simultaneously for a synaptic weight change to occur and this property can be used to learn something about the function F. Taylor series expansion around $v_i = v_j = 0$ leads to:

$$\frac{d}{dt}w_{ij} = c_0(w_{ij}) + c_1^{post}(w_{ij})v_i + c_1^{pre}(w_ij)v_j + c_2^{pre}(w_{ij})v_j^2 + c_2^{post}(w_{ij})v_i^2 + c_2^{corr}v_iv_j + \mathcal{O}(v^3)$$

The term with c_2^{corr} is bilinear in pre- and postsynaptic activity showing that the change of the synaptic efficacy w_{ij} is indeed subject to a combination of changes in both activities.
Simplest choice of F is to fix $c_2^{corr} = c > 0$ and set all other terms of the taylor expansion to zero resulting in the prototype of hebbian learning.

$$\frac{d}{dt}w_{ij} = c_2^{corr}v_iv_j$$

This learning rule includes only first-order terms and therefore models non-Hebbian plasticity. More complicated learning rules include higher-order terms of the taylor expansion. If $c_2^{corr} = c < 0$, the learning rule is anti-hebbian.
F is dependet on the efficacy w_{ij}, so that it will not grow without limit. A saturation of synaptic weights can be achieved if parameter c_2^{corr} goes to zero as w_{ij} approaches its maximum value:

$$c_2^{corr}(w_{ij}) = \gamma_2(1 - w_{ij})$$

Originally Hebb did not consider a rule for decreasing the synaptic weights. In order to have a feasible model however, the synaptic efficacy should decrease in the absence of stimulation.

$$c_0(w_{ij}) = -\gamma_0 w_{ij}$$

The combination results in the simplest feasible learning rule:

$$\frac{d}{dt}w_{ij} = \gamma_2(1 - w_{ij})v_iv_j - \gamma_0 w_{ij}$$

3.3.1 Gating

Postsynaptic gating A weight change occurs only if the postsynaptic neuron is active, $v_i > 0$. $\Rightarrow$ the weight changes are "gated" by the postsynaptic neuron. Only the efficacies of highly active presynaptic neurons $v_j > v_\phi$ are strengthened.

$$\frac{d}{dt}w_{ij} = \gamma v_i(v_j - v_\phi(w_{ij})), \gamma > 0$$

Presynaptic gating role of pre- and postsynaptic firing rate are exchanged. Now, a change in synaptic weights can only occur if the presynaptic neuron is active, $v_j > 0$ and the activity of the postsynaptic neuron determines the direction of the change.

$$\frac{d}{dt}w_{ij} = \gamma v_j(v_i - v_\phi))$$

3.3.2 BCM rule

The "Bienenstock-Cooper-Munroe" rule is a generalization of the presynaptic gating rule, where Φ is a non-linear function and the reference rate v_ϕ is the running average of the postsynaptic activity v_i.

$$\frac{d}{dt}w_{ij} = \eta\Phi(v_i - v_\phi))v_j - \gamma w_{ij}$$

3.4 Learning in Rate Models (functional consequences of hebbian learning)

Goal: understand how activity-dependent learning rules influence the formation of connections between neurons in the brain.
Plasticity is controlled by the statistical properties of the presynaptic input for the postsynaptic neuron.

3.4.1 Evolution of synaptic weights

For the sake of simplicity we model the presynaptic input as a set of static patterns $\{x^\mu \in \mathcal{R}^N; 0 \leq \mu \leq p\}$. At each time step one of the patterns is selected at random and the presynaptic rates are fixed to $v_i = x_i^\mu$. The synaptic weights are modified according to a Hebbian learning rule dependent on the correlation of pre- and postsynaptic activity: $\frac{d}{dt}w_{ij} = a_2^{corr}v_i^{post}v_j^{pre}$.
In a general rate model the firing rate v_i^{post} is given by a nonlinear function of the total input: $v^{post} = g(\sum_i w_i v_i^{pre})$. For simplification reasons we focus on a linear rate model:

$$v_i^{post} = \sum_k w_{ik}v_k^{pre}$$

We can now combine the learning rule and the linear rate model to form:

$$\frac{d}{dt}w_{ij} = a_2^{corr}\sum_k w_{ik}v_k^{pre}v_j^{pre}$$

We are interested in the long-term behaviour of the synaptic weights and therefore consider the expectation value of the weight vector, i.e., the weight vector averaged over the sequence $(x^{\mu_1}, x^{\mu_2}, ..., x^{\mu_n})$ that have so far been presented to the network.

$$\langle \frac{d}{dt} w_{ij} \rangle = a_2^{corr} \sum_k \langle w_{ik} \rangle \underbrace{\langle x_k^\mu x_j^\mu \rangle}_{C_{kj}} \qquad C_{kj} = \langle x_k x_j \rangle = \frac{1}{p} \sum_{\mu=1}^{p} x_k^\mu x_j^\mu$$

C_{kj} are the correlation matrices for respective components i and j in the sum of the learning rule. Using PCA we can detect the direction of maximal variance in the set of patterns. $\Rightarrow$ Express weight vector w in eigenvectors of C. What do principal components mean in this respect?

PCA think of neuronal output as projection of weight vector, fill in PCA equations

3.5 Oja's rule

The Oja learning rule is a mathematical formalization of the Hebbian learning rule, such that over time the neuron actually learns to compute a principal component of its input stream. The forgetting term added to balance the growth of the weights this time is not only proportional to the value of the weight, but also to the square of the output of the neuron v_i.

$$\frac{d}{dt} w_{ij} = a_2^{corr} v_j v_i - \underbrace{a_2^{post}}_{a_2^{corr} w_{ij}} (v_i^{post})^2 = a_2^{corr} (v_j v_i - w_{ij} (v_i^{post})^2)$$

Now, the forgetting term balances the growth of the weight. The squared output $(v_i^{post})^2$ guarantees that the larger the output of the neuron becomes, the stronger is this balancing effect.

3.5.1 Oja's rule and PCA

$$v_i^{post} = \mathbf{w}^T \mathbf{x}^\mu$$

$$\frac{d}{dt} \mathbf{w} = a_2^{corr} (\mathbf{x}^\mu \mathbf{x}^{\mu T} \mathbf{w} - \mathbf{w}^T \mathbf{x}^\mu \mathbf{x}^{\mu T} \mathbf{w} \mathbf{w}), \{x^\mu, 0 \leq \mu \leq p\}$$

This is the incremental change for just one input vector x^μ. When the algorithm is run for a long time, changing the input vector at every step, one can look at the average behaviour. An especially interesting question

is what is the value of the weights when the average change in the weight is zero. This is the point of convergence of the algorithm. Averaging right hand side over x^μ while w stays constant yields the following equation.

$$\frac{d}{dt}\mathbf{w} = a_2^{corr}(\underbrace{\langle \mathbf{x}\mathbf{x}^\mathbf{T} \rangle}_{C}\mathbf{w} - \underbrace{(\mathbf{w}^T \underbrace{\langle \mathbf{x}\mathbf{x}^\mathbf{T} \rangle}_{C}\mathbf{w})\mathbf{w}}_{\lambda}) = 0, \overline{x} = 0$$

Considering that the quadratic form $w^T C w$ is a scalar, this equation clearly is the eigenvalue-eigenvector equation for the covariance matrix C. This analysis shows that if the weights converge in the Oja learning rule $C\mathbf{w} = \lambda\mathbf{w}$, then the weight vector becomes one of the eigenvectors of the input covariance matrix $\mathbf{w} = \mathbf{e_1}$ (ONLY THIS STATE IS STABLE), and the output of the neuron becomes the corresponding principal component. Principal components are defined as the inner products between the eigenvectors and the input vectors. For this reason, the simple neuron learning by the Oja rule becomes a principal component analyzer (PCA). Although not shown here, it has been proven that the neuron will find the the `first principal component` and the norm of the weight vector tends to one. PCA IS USED FOR HEBBIAN LEARNING IN **linear** NEURONS

3.6 Independent Component Analysis (ICA)

Goal: Compute a transformation $\mathbf{y} = A\mathbf{x}$ such that the entries y_i of the transformed vector (components) are mutually independent. Statistical independence is a stronger constraint than uncorrelatedness required by the PCA and actually implies it. (The two conditions are equivalent *only* if the sources x_i are generated from Gaussian random variables - one random vector).

Assumptions: Random input vector $\mathbf{x}$ is generated by a linear combination of statistically independent $p(\mathbf{x}) = p(x_1, \ldots, x_n) = p_1(x_1) \ldots p_n(x_n)$ and stationary components (sources), $\mathbf{x} = A\mathbf{y}$. The task is now is to find a matrix W so as to recover the original signal sources, the components of $\mathbf{y} = \begin{bmatrix} s_1 & s_2 & \ldots & s_n \end{bmatrix}^T$ by exploiting the information hidden in the samples of the mixed signal patterns $\mathbf{x}$ (blind source separation). $\mathcal{A}$ is known as the mixing and W as the demixing matrix, respectively.
In contrast to PCA which can always be performed, ICA is meaningful only if the involved random variables are non-Gaussian. Each of the resulting independent components is `uniquely` estimated up to a multiplicative constant.

For this reason the components are considered to be of unit variance. (Multidimensional Gaussians are isometric/rotationally symmetric and therefore the orientation/principal axis of the data can't be recovered). Searching for independent rather than uncorrelated features gives us the means of exploiting a lot more information, hidden in the higher order statistics of the data.

Ambiguities permutation of speakers (ICA can't tell which is which) sign of the speakers (ICA can't guarantee outputting the right sign of the signal) for many applications (audio) one doesn't care about these facts

Preprocessing (whitening)

- Zero mean data
- Whitening (unit variance). Perform PCA on mixed signals and divide each component by the square-root of its eigenvalue $x_k^n = \frac{1}{\sqrt{\lambda_k}} x_k \Rightarrow C^n = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. After PCA entries of projections $\mathbf{y}$ of random vectors are uncorrelated and independence can be rewritten to that of projections $p(y_1, \dots y_n) = p_1(y_1) \dots p_n(y_n)$

How to find natural axis? We need measure for non-gaussianty. If we look at two-dimensional data, an arbitrary projection will be close to a Gaussian distribution which doesn't tell us anything about our data (neuronal inputs). $\Rightarrow$ search for direction which is maximally non-Gaussian.

3.6.1 Gaussianity vs. non-gaussianity

This approach in performing ICA is a direct generalization of the PCA technique. The Karhunen-Love transform focuses on the second-order statistics and demands the cross-correlations $\langle y_i y_j \rangle$ to be zero. In ICA demanding that the components of $\mathbf{y}$ be statistically independent is equivalent to demanding all the higher order cross-cumulants to be zero (going up to the fourth-order cumulants is sufficiant for many applications).careful: Cross-cumulant vs auto-cumulant cumulants vs. moments? These are actually

moments?

$$\begin{aligned}
&\mathcal{K}_1(y(i)) = \mathbb{E}[y(i)] = 0, \quad \textbf{mean} \\
&\mathcal{K}_2(y(i)y(j)) = \mathbb{E}[y(i)y(j)], \quad \textbf{(Co)variance} \\
&\mathcal{K}_3(y(i)y(j)y(k)) = \mathbb{E}[y(i)y(j)y(k)], \quad \textbf{skewness}
\end{aligned}$$

$$\begin{aligned}
\mathcal{K}_4(y(i)y(j)y(k)y(r)) &= \mathbb{E}[y(i)y(j)y(k)y(r)] - \mathbb{E}[y(i)y(j)]\,\mathbb{E}[y(k)y(r)] \\
&\quad - \mathbb{E}[y(i)y(k)]\,\mathbb{E}[y(j)y(r)] - \mathbb{E}[y(i)y(r)]\,\mathbb{E}[y(j)y(k)] \\
&= \mathbb{E}[y^4] - 3\,\mathbb{E}[y^2], \quad \textbf{kurtosis}
\end{aligned}$$

We need a measure for non-gaussianity, which we find in the higher order cumulants, because for Gaussian distributions there is no difference in ICA & PCA. The kurtosis (4-th order cumulant) is such a measure of non-gaussianity. The smaller it is the more it describes a leptokurtic curve with large values around the mean descending fast towards the first standard deviation. The problem can now be reduced to finding a matrix, W, for that the second-order (covariacnes) and fourth-order cross-cumulants, kurtosis, of the transformed patterns are zero.
DOES ANYONE UNDERSTAND THE MEANING OF THE FOLLOWING EQUATION?

$$J(w) = [\mathbb{E}(F(y)) - \mathbb{E}(F(v))]^2$$

TODO: MINIMIZE NON-GAUSSIANITY

3.6.2 ICA by neuronal rule

batch

$$\Delta\mathbf{w} = \eta\,\mathbb{E}\left[\mathbf{x}g(\mathbf{w}^T\mathbf{x})\right]$$

online

$$\Delta\mathbf{w} = \eta\mathbf{x}g(\mathbf{w}^T\mathbf{x})$$

TODO

3.6.3 FastICA

1. Initialize w
2. Newton step $\mathbf{w}^{new} = \mathbb{E}\left[\mathbf{x}(g(\mathbf{w}^T\mathbf{x}) - \mathbf{w}g'(\mathbf{w}^T\mathbf{x}))\right]$
3. Renormalize $\mathbf{w} = \frac{\mathbf{w}^{new}}{\|\mathbf{w}^{new}\|}$
4. if algorithm as not converged, go to 2)

3.6.4 Temporal ICA

Goal: Find unmixing matrix W such that outputs of time-dependent signals are independent.

Assumptions: Signals are time-dependendent and the different components may be recorded with delays τ_i

$$\langle y_i(t) y_k(t - \tau_k) \rangle = \delta_{ik} \lambda(\tau)$$

4 Competitive Learning

4.1 Clustering

4.1.1 Nearest Neighbour

Initialize prototypes p_k to represent groups of data points. Assign datapoints to nearest neighbouring prototype $|\mathbf{p}_k - \mathbf{x}| \leq |\mathbf{p}_i - \mathbf{x}|$ resulting in Voronoi tesselation of input space. For data compression the input $\mathbf{x}$ is classified and the index k transmitted. This gives rise to the construction error

$$\mathbb{E}(\mathbf{w}_1, \ldots, \mathbf{w}_n) = \sum_k \sum_{\mu \in C_k} (\mathbf{w}_k - \mathbf{x}^\mu)^2$$

But how to choose initialization of prototypes? To minimize the reconstruction error one has discretize the input space adaptively.

4.1.2 K-Means

Standard K-means is based on a random initialization of prototypes.

1. For every data pattern x^μ the winning prototype $\mathbf{p}_k$is determined with the nearest neighbour rule.

2. The winning prototype is moved closer to the classified $\mathbf{x}_k^\mu$ by updating it according to $\Delta \mathbf{p}_k = \eta(\mathbf{p}_k - \mathbf{x}^\mu)$.

Showing that the error decreases Gradient descent on the error surface in the following manner

$$\Delta \mathbf{p}_j = -\eta \frac{\partial \mathbb{E}}{\partial \mathbf{w}_j} = -\eta \frac{\partial}{\partial \mathbf{w}_j} \sum_k \sum_{\mu \in C_k} (\mathbf{w}_k - \mathbf{x}^\mu)^2$$

$$\textbf{batch update}: \quad \Delta \mathbf{p}_j = -2\eta \sum_{\mu \in C_j} (\mathbf{x}^\mu - \mathbf{w}_j)$$
$$\textbf{online update}: \quad \Delta \mathbf{p}_j = 2\eta (\mathbf{x}^\mu - \mathbf{w}_j)$$

In the batch update rule distances from the data points in the prototype's class are summed up and the prototype is moved towards the mean with step size η.
After convergence: $\Delta w_k = 0$

$$\sum_{\mu \in C_k} \mathbf{w}_k - \sum_{\mu \in C_k} \mathbf{x}^\mu = N_k \mathbf{w}_k - \sum_{\mu \in C_k} \mathbf{x}^\mu$$
$$\Rightarrow \mathbf{w}_k = \frac{1}{N_k} \sum_{\mu \in C_k} \mathbf{x}^\mu$$

So with the batch update rule each prototype is at the center of its data cloud after convergence.
The online update has a stochastic component, because one has to choose a data point at random and move into its direction by the step size η. This causes the algorithm to jitter around the mean. To alleviate this problem the trick is to reduce η so that the algorithm eventually freezes.

Problem of dead units Dead units are prototypes without data clouds. The error surface allows the algorithm to converge into a state permitting dead units through several local minima. To overcome this problem it is useful to initialize the prototypes to data points in k-means.

4.2 Neuronal Implementation of Clustering

(now multiple postsynaptic neurons for the same input?) In the neuronal implementation the data patterns take the form of sensory inputs to neurons in the brain, one postsynaptic neuron for every prototype $\mathbf{p}_j$. The activity of these neurons is defined by a non-linear neuron-model $v_1^{post} = y_1 = g(\sum_k \mathbf{w}_{ik} \mathbf{x}_k^\mu) = g(\mathbf{w}^T \mathbf{x}^\mu)$ which performs a non-linear transform of the scalar product of the prototypes used as weights $\mathbf{w}_j = \mathbf{w}_k$ which are now synaptic weights and the sensory inputs.
For NORMALIZED WEIGHT VECTORS the clustering expression $|\mathbf{p}_k - \mathbf{x}| \leq |\mathbf{p}_i - \mathbf{x}|$ can be rewritten to $\mathbf{w}_k^T \mathbf{x} \leq \mathbf{w}_k \mathbf{x}$. This means that the smallest distance to the prototype $\mathbf{w}_k$ results in the largest scalar product. We reformulate our clustering aim to finding the postsynaptic neuron with the maximal scalar product ("biggest total stimulation").

Winner take all circuit How can it be assured that only one neuron wins, that is to say that the respective input pattern is classified to a single prototype? Simply by introducing a `lateral term` incorporating activities of the other postsynaptic neurons for the other prototypes. The winner is the neuron that receives the most stimulation from outside (strongest outside drive) inhibiting the others which is equivalent to saying input pattern is classified to the prototype with which it forms the largest scalar product $\mathbf{w}_k \leq \mathbf{x}^\mu$.

$$y_i(t+1) = g(\text{total drive}) = g(\sum_k w(t+1)_{ik}\mathbf{x}_k^\mu + \sum_j B_{ij}y(t)_j), \quad B_{ij} < 0$$

$$\Rightarrow y_i(t^{final}) = \begin{cases} 1 \text{ for winner} \\ 0 \text{ for others} \end{cases}$$

How to update weights?

Hebbian Learning Rule (no inhibition) For every pattern $\mathbf{x}^\mu$ find winning prototype $\mathbf{w}_k = \mathbf{p}_k$ and update it according to the following learning rule:

$$\begin{aligned} \frac{d}{dt}w_{ki} &= \eta(y_k x_i^\mu - \gamma w_{ki} y_k) \\ &= \eta(\delta_{ki}x_i - w_{ki}\delta_{ki}) \\ &= \eta(x_i - w_{ki}) \end{aligned}$$

update for winner k (only neuron active) is same eq. as before (k-means), now derived as a hebbian learning rule:

$$\frac{d}{dt}\mathbf{w}_k = \eta(\mathbf{x}^\mu - \mathbf{w}_k)$$

Kohonen Learning Rule

1. choose input $\mathbf{x}^\mu$
2. determine winner k according to current weights for input (scalar product)
3. update weights of winner k

$$\frac{d}{dt}w_{ki} = \eta(y_k x_i^\mu - \gamma w_{ki} y_k)$$

and neighbours j

$$\frac{d}{dt} = \eta\Lambda(j,k)(\mathbf{x}^{\mu} - \mathbf{w}_j), \quad \text{on-line}$$

Every neighbour j is inhibited according to inhibition coefficients of k for j.

4. decrease size of neighbourhood function $\Lambda(j,k)$ to ensure convergence?

5. reiterate from 1) until stabilization

Thus, the neuron whose weight vector was closest to the input vector is updated to be even closer. The result is that the winning neuron is more likely to win the competition the next time a similar vector is presented, and less likely to win when a very different input vector is presented. As more and more inputs are presented, each neuron in the layer closest to a group of input vectors soon adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors will have a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input vectors it sees.

5 Reinforcement learning

5.1 Introduction

RL vs. supervised learning Supervised learning, used in statistical pattern recognition and artifical neural networks is learning from examples provided by a knowledgable external supervisor whereas in RL an agent is instructed to learn from its own experience. It is a "behavioral" learning problem: Through interaction between the learning system and its environment, whereas the system seeks to achieve a specific goal.

trade-off exploration vs. exploitation An agent has to exploit what it ALREADY KNOWS in order to obtain a reward, but it also has to EXPLORE in order to make BETTER ACTION SELECTIONS in the future. The agent must try a variety of actions and progressively favour those that appear to be best, because neither schemes can be pursued exclusively without failing a the task. Estimate reward probabilities vs. take action to maximize reward.

RLs main characteristic NOT ONE-SHOT DECISION MAKING PROBLEM, harder than supervised learning RL explicitly considers the WHOLE problem of a goal-directed agent INTERACTING with an uncertain environment. All reinforcement learning agents have explicit goals, can sense aspects of their environments, and can choose actions to influence their environments. Moreover, it is usually assumed from the beginning that the agent has to operate despite significant UNCERTAINTY ABOUT THE ENVIRONMENT it faces. When reinforcement learning involves PLANNING, it has to address the *interplay between planning and real-time action selection*, as well as the question of how environmental models are acquired and improved.
An agent's actions are permitted to affect the future state of the environment (e.g., the next chess position, the level of reservoirs of the refinery, the next location of the robot), thereby affecting the options and opportunities available to the agent at later times. Correct choice requires taking into account indirect, delayed consequences of actions, and thus may require foresight or planning. The knowledge the agent brings to the task at the start–either from previous experience with related tasks or built into it by design or evolution–influences what is useful or easy to learn, but interaction with the environment is essential for adjusting behavior to exploit specific features of the task.

5.1.1 Elements of reinforcement learning

there are four main subelements of a reinforcement learning system.

Policy commonly denoted by π, it defines the learning agents way of behaving at a given time. Mapping from the perceived states of the environment to the actions that present themselves to the agent in these states. The policy alone is sufficient to determine the behaviour of an agent. In oder words, a policy is a rule used by the learning system to decide what to do, given the current state of the environment.

Reward function defines the goal in a reinforcement learning problem. Maps each perceived state of the environment to a single number, a reward, which describes the intrinsic desirability of the state. The agent's sole objective is to maximize its total gain from the rewards it receives.
The reward function may serve as a basis for altering the agent's policy.

Value function While the reward function indicates which actions are immediately beneficial to the agent, a **value function** specifies what is

good for the agent in the long run. The value of a state is the EXPECTED total amount of rewards the agent will accumulate from this state onwards. For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards.

Model of the environment A model mimics the behaviour of the environment. *For example, given a state and action, the model might predict the resultant next state and next reward.* Models are used for PLANNING, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced. The incorporation of models and planning into reinforcement learning systems is a relatively NEW DEVELOPMENT. Early reinforcement learning systems were explicitly trial-and-error learners; what they did was viewed as almost the opposite of planning. Nevertheless, it gradually became clear that reinforcement learning methods are closely related to dynamic programming methods, which do use models, and that they in turn are closely related to state-space planning methods.

5.2 Markov Decision Problems (MDP)

Reinforcement learning algorithms model the world as a Markov Decision Problem (MDP)

MDP $(S, A, P^a_{s \to S'}, \gamma, \mathcal{R})$

S, state space

A, contains possible actions associated with states

$\sum_{s'} P^2_{s \to s'} = 1$, outgoing action prob. sum to 1

γ, discount factor for future rewards

$\mathcal{R} : S \to \mathbb{R}$, reward function maps states to reward values

policy/strategy profile $\pi(s, a)$, describes which action a to take in state s

5.3 Reward-based action learning: Q-values

In the case of a `1-step horizon` the agent starts of in an initial state s from where he can get to several states s'_i by choosing an associated action a_i. A reward $P^a_{s \to s'}$ is associated with every action.

Goal: The agent should find the optimal strategy a^* resulting in the maximization $Q(s, a^*) > Q(s, a_j)$ of the expected reward $Q(S, a) = \sum_{s'} P^a_{s \to s'} R^a_{s \to s'}$.

Update rule: The Q values are however not known at the beginning, because the reward probabilities $P^a_{s \to s'}$ are not known. Therefore, the agent needs to learn them via iterative updates:

$$\Delta Q(s, a) = \eta \left[R^a_{s \to s'} - Q(s, a) \right] = \eta \left[r - Q(s, a) \right]$$

These consist of the difference between received reward and expected reward multiplied by a learning rate η. Over multiple episodes the Q-values converge to the true value of the reward r, expressed by convergence of the mean update $< \Delta Q(s, a) >= 0$.

5.4 SARSA and Bellman equation

For `Multi-step horizon` reward-based action learning we reformulate the update rule to take into account the Q-values from the next step multiplied by a discount factor γ:

$$\begin{aligned} \Delta Q(s, a) &= \eta \left[r - (Q(s, a) - \gamma Q(s', a')) \right] \\ &= \eta \underbrace{\left[r + \gamma Q(s', a') - Q(s, a) \right]}_{\delta_t} = \eta \delta_t \end{aligned}$$

This update rule is called **SARSA** for `State Action Reward State Action` because it uses the state and action of time t along with the reward, state, and action of time t + 1 to form the so-called **temporal difference (TD) error** δ_t. This update rule is called `associative` because the values that it learns are associated with particular states in the environment $s \in S$. We can write the same update rule without any s terms. This update rule would learn the value of actions without associating them with any environment state, thus it is `nonassociative`. *For instance, if a learning agent were playing a slot machine with two handles, it would be learning only action values, i.e. the value of pulling lever one versus pulling lever two.* The associative case of learning state-action values is commonly thought of as the full reinforcement learning problem, although the related state value and action value cases can be learned by the same TD method.

Bellman equation The Bellman equation expresses the expected reward $Q(s, a)$ recursively over an infinite/multi-step horizon:

$$Q(s, a) = \sum_{s'} P^a_{s \to s'} [R^a_{s \to s'} + \gamma \sum_{a'} \pi(s', a') Q(s', a')]$$

The $Q(s', a')$ are weighted differently according to the chosen policy $\pi(s, a)$ derived from $Q(s, a)$.

SARSA algorithm The Bellman equation can be used for the SARSA algorithm implementing the SARSA update rule:

1. Being in state s, chose action a according to **policy**
2. Observe reward r, next state s'
3. Choose action a' in state s' according to **policy**
4. Update $\Delta Q(s, a)$
5. $s' \rightarrow s$, $a' \rightarrow a$
6. Goto 1)

5.5 Strategies for exploration vs. exploitation

There are different strategies for exploitation/`ways to find optimal policy` while correct Q-values are not known:

5.5.1 Choosing a policy

The action choices for all states $s \in S$ are specified by a policy π. A policy consists of a rule for choosing the next state based on the value or predictions for possible next states (and in our case actions). More specifically, a policy maps state values to actions. Policy choice is very important due to the need to balance exploration and exploitation. This often means taking suboptimal actions in order to learn more about the value of an action or state.

- **Greedy policy**: Consider an agent who takes the highest-valued action a^*, $Q(s, a^*) > Q(s, a)$ at every step. This policy is called a greedy policy, because the agent is only exploiting its current knowledge of state or action values to maximize the expected reward and is not exploring to improve its estimates of other states. *A greedy policy is likely undesirable for learning because there may be a state which is actually good but whose value cannot be discovered because the agent currently has a low value estimate for it, precluding/preventing its selection by a greedy policy.*

- **ε-greedy policy**: take action which looks best with . A common way of balancing the need to explore states with the need to exploit current knowledge in maximizing rewards is to follow an ε-greedy policy which simply follows a greedy policy with probability $P = 1 - \varepsilon$ and takes a random action with probability ε. *This method is quite effective for a large variety of tasks.*

- **Softmax strategy**: take action a' which looks best with probability Another common policy is softmax. You may recognize softmax as the stochastic activation function for Boltzmann machines, often called the Gibbs or Boltzmann distribution. With softmax, the probability of choosing action a at time t is

$$P(a') = \frac{\exp(\beta Q(a'))}{\sum_a \exp(\beta Q(a))}$$

  where the denominator sums over all the exponentials of all possible action values and $\tau = \frac{1}{\beta}$ is the temperature coefficient. A high temperature causes all actions to be equiprobable, while a low temperature skews the probability toward a greedy policy. *The temperature coefficient can also be annealed over the course of training, resulting in greater exploration at the start of training and greater exploitation near the end of training.*

- **Optimistic greedy**: start off with Q values that are too big

Action Values Q versus State Values V EXPECTED PAYOFFS Almost all reinforcement learning algorithms are based on estimating value functions. These are functions of states V (or of state-action pairs Q) that estimate how good it is for the agent to be in a given state V_s(or how good it is to perform a given action in a given state Q_a). The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular policies.

Recall that a policy π is a mapping from states $s \in S$ and actions $a \in A$ to the probability $\pi(s, a)$ of taking action a when in state s. Informally, the value of a state s under a policy π, denoted by $V^\pi(s)$ is the expected return when starting in s and FOLLOWING π THEREAFTER. The Bellman equation

for state values V becomes:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \overbrace{\sum_{s'} P^{a}_{s \to s'} [R^{a}_{s \to s'} + \gamma \underbrace{\sum_{a'} \pi(s',a') Q(s',a')}_{V^{\pi}(s')}]}^{Q(s,a)}$$

$$= \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{s \to s'} [R^{a}_{s \to s'} + \gamma V^{\pi}(s')], \quad \text{in form of itself}$$

The difference between Q and V is the following: The value function is the expected return starting from state s_t and following the policy π. The state-action value function $Q(s,a)$ is the expected return starting from state s_t, taking action a and `only thereafter` following policy π. We however use the state-action values Q instead of the mere state values V

5.6 On-policy vs off-policy learning

TODO: part doesn't fully match slides?
On-policy means SARSA; off-policy means Q-learning. The difference lies in the method used to compute the update for $Q(s,a)$. While for SARSA, we use the policy π to select the action a' for the discounted term $(\gamma \sum_{a'} \pi(s',a') Q(s',a'))$, the Q-learning algorithm simply uses the greedy approach to select a'. Off-policy Q-learning by default uses a greedy update rule and the policy derived from Q is only employed for selection of the first action a from the current state:

$$\Delta Q(s,a) = \alpha \left[r(a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

The `Problem` with these algorithms is that learning is slow, that is to say the propagation of Q-values is slow.

5.7 Eligibility traces

An eligibility trace is a temporary record of the `occurrence of an event`, such as the visiting of a state or the `taking of an action` (relevant in our case). The trace marks the memory parameters associated with the event as eligible for undergoing learning changes by weighting the learning rule accordingly.
At the moment of the reward update, we want to also update previous action values along trajectories, because otherwise the information diffusion

across several states takes too much time. Every time an action a is chosen according to a policy, the eligibility traces are evaluated thus:

- If a = action taken in state j: $e(j,a)(t) = \gamma\lambda e(j,a)(t-\Delta t) + 1$
- else: $e(s,a)(t) = \gamma\lambda e_(s,a)(t-\Delta t)$

The γ here is the memory reduction parameter.
Then, for all states s and actions a use the respective eligibility trace as an additional coefficient for the SARSA update rule with TD error $\delta_t = r_t + \gamma Q(s',a') - Q(s,a)$.

$$\Delta Q(s,a) = \eta\delta_t e(j,a)$$

5.8 Continuous states

Q-values for continuous states approximation by a weighted sum of basis functions

$$Q(s,a) = \sum_j w_{aj} * r_j(s) = \sum_j w_{aj} * \phi(s - s_j)$$

s_j center of basis function, Φ shape of basis functions - determines how much of the weights are used.
If the time interval between subsequent weight updates approaches zero, the difference between two consecutive states $s' - s = v(a) * \Delta t$, where $v(a)$ is the velocity resulting from the choice of the action a. $s' - s$ approximates zero, therefore $s' \approx s$. Similarly, the finer the time resolution becomes, the likelier it is that $a \approx a'$. Thus, for small time steps, an online gradient descent on the error term E_t becomes closer to a δ_t modulated Hebbian rule. TODO: Exercise $\frac{dQ(s',a')}{dw_{aj}}$

5.9 Temporal Difference TD(λ) algorithms

TD algorithms are often used in reinforcement learning to predict a measure of the total amount $r_{tot.}$ of reward expected over the future, but they can be used to predict other quantities such as the expected reward Q as well. Eligibility traces are usually implemented in this context by an exponentially-decaying memory trace, with decay parameter λ. This generates a family of TD algorithms **TD**(λ), $0 \leq \lambda \leq 1$ with **TD**(0) corresponding to updating only the immediately preceding prediction (as described above - scholarpedia), and **TD**(1) corresponding to equally updating all the preceding

predictions (`which is actually Q-learning`).

$$e(j,a)(t) = \gamma\lambda e(j,a)(t-\Delta t) + \begin{cases} 1 * r_j & \text{if} \quad a = a' \\ 0 & \text{else} \end{cases}$$
$$= \gamma\lambda e(j,a)(t-\Delta t) + r_j \delta_{a,a'}$$

Eligibility traces do not have to be exponentially-decaying traces, but these are usually used since they are relatively easy to implement and to understand theoretically.

5.9.1 Neuronal interpretation

TD incorporates eligibility and continuous state space. We do a function approximation using weights and instead of updating $Q(s,a)$. TD method with a function approximator like a neural network, we update V_t^π by finding the output gradient with respect to the weights. This step is not captured in the previous equation $\delta_t = r_t + \gamma Q(s',a') - Q(s,a)$. We will discuss these implementation specific details later.

$$Q_w(s,a) = \sum_{i=1}^{n} w_i^a * r_i(s)$$

$$\Delta w_{aj} = \eta \delta_t e_{aj}$$

Eligibility and continous TD

1. choose action a according to policy (greedy)
2. observe reward r in state s', from there choose next action a'
3. calculate TD error δ_t in SARSA
4. update eligibility trace of action a in current state $e(j,a)$
5. perform weight update $\Delta w_{ja} = \eta \delta_t e(j,a)(t)$
6. update Q-value approximation for state j $Q_w(j,a) = \sum_{s' \in S'} w_{s'}^a r_{s'}(j)$ (rewards we can get through actions a leading to states s' in s) for next episode.
7. $a \leftarrow a'$, $s \leftarrow s'$, return to 2)

eligibity traces implement (short-term) memory The weights grow/connection is reinforced if action a at state s is successful in obtaining a reward

TODO: DOES SOMEBODY HAVE THE BLACKBOARD: REPRESENTATION OF STATE-VALUE Lecture 7 (Backgammon)

We can write the eligibility trace as $e_{aj}(t) = \gamma\lambda e_{aj}(t - \Delta t) + r_j\delta_{a,a'}$

6 Energy-Based Models (EBM)

A scalar energy is associated to each configuration of the variables of interest in the model (cost function). Learning then corresponds to modifying the energy function so that its shape has desirable properties, for example we would like desirable configurations to have low energy. The probabilistic distribution of for the model is define by an energy function:

$$p(x) = \frac{\exp(-E(x))}{Z}$$

This distribution is the Boltzmann or Gibbs distribution which originates from classical statistical mechanics.
The normalization factor Z is called the partition function $Z = \sum_x \exp(-E(x))$.
An energy-based model can be learnt by performing (stochastic) gradient descent on the empirical negative log-likelihood of the training data.

6.1 EBMs with hidden units

Often one does not observe the examples x fully or one wants to introduce some non-observed variables to increase the expressive power of the model. So we consider a hidden part y additionally to the observed part x, giving:

$$P(x) = \sum_y P(x, y) = \sum_y \frac{\exp(-E(x, y))}{Z}$$

By introducing the notation of FREE ENERGY

$$\mathcal{F} = -\log \sum_y \exp(-\mathcal{F})$$

we can the rewrite the probability distribution in the condensed of the gibbs distribution:

$$P(x) = \frac{exp(-\mathcal{F}(x))}{Z}, with Z = \sum_x \exp(-\mathcal{F}(x))$$

6.2 Restricted Boltzmann Machines (RBM)

Boltzmann Machines (BMs) are a particular form of log-linear (Gibbs distribution) Markov Random Fields (MRF), where the energy function is linear in its free parameters. In order to represent complicated distributions (limited parametric setting $\rightarrow$ non-parametric setting) we consider some of the variables to be hidden (not observed). One can increase the modeling capacity of the Boltzmann Machine (BM) by adding for hidden variables/units. Furthermore a RBM is restricted to a bipartite Graph, because there no lateral connections visible-visible and hidden-hidden are allowed. We introduce the notation of $v = x$ for visible units and $h = y$ for hidden units.
The energy function therefore is defined as:

$$E(v, h) = -b^t v - c^t h - h^t W_{u,v} u$$

where $W_{u,v}$ represents a connection matrix of weights for hidden and visible units and b, c are the offsets of the visible and hidden layers respectively.
By plugging in the energy function we get the following representation of the free energy formula:

$$F(v) = -b^t v - \sum_i \log \sum_{h_i} \exp(h_i(c_i + W_i v))$$

Because of the specific structure of RBMs, visible and hidden units are conditionally independent of one-another

A particular set of "synaptic" weights is said to constitute a perfect model of the environmental structure if it leads to exactly the same probability distribution of the states of the visible units as when these units are clamped by the environmental input vectors.

RBMs with binary units

Commonly RBMs are studied with binary units $v_i, h_i \in \{0, 1\}$ ... The free energy simplifies to:

$$\mathcal{F}(v) = -b^t v - \sum_i \log(1 + \exp(c_i + W_i v))$$

deeplearning.net/tutorial/rbm.html
sigmoid function (sigm):

$$\sigma(t) = sig(t) = \frac{1}{1 + \exp(-t)} = \frac{1}{2} \cdot (1 + \tanh(\frac{t}{2}))$$