Unsupervised and Reinforcement Learning in Neural Networks

Lecture Notes

Contributors: Fabian Brix, Renato Kempter, Stig Viaene

January 5, 2014

Contents

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 I highly recommend reading the section of Gerstner's book on this subject at http://icwww.epfl.ch/ gerstner/SPNM/node77.html#SECTION042140000000000000.

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 not considered in this class because there is no easy biological analogy. It 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 choosing 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

Synaptic plasticity is basically the change in connection strength of the synapses over time. According to the Hebbian learning rule, its changes are governed by the following postulate:

> 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 i 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 respective 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. (A rigorous derivation is presented in Section 3.4.2.)

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 j to a post-synaptic neuron i. 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 dependent on the efficacy w_{ij}, so that it will not, in reality, 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

Gating is the process of restricting the number of changes that occur. We discern between postsynaptic and presynaptic gating.

Postsynaptic gating A weight change occurs only if the postsynaptic neuron is active, $v_i^{post} > 0$. $\Rightarrow$ the weight changes are "gated" by the postsynaptic neuron. Only the efficacies of highly active presynaptic neurons $v_j^{pre} > v_\phi$, which is a function of w_{ij}, are strengthened.

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

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, $v_j^{pre} > 0$, and the activity of the postsynaptic neuron determines the direction of the change.

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

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.

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

Note that the dependency of v_ϕ on w_{ij} (via v_i^{post}) implements a homeostatic plasticity.

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.

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

C_{kj} are the correlation matrix components.

3.4.2 Detection of correlation

We denote here the weight vector between the post- and presynaptic neuron as $\vec{w}$, its components are w_i for i from 1 to N. (Note that we left out the second index because we are only considering a single postsynaptic neuron.)

From what we had before, we get:

$$\frac{d\vec{w}}{dt} = a_2 C \vec{w}$$

which is an ordinary matrix differential equation[1].

[1] Some information on the topic can be found on http://en.wikipedia.org/wiki/Matrix_differential_equation

Solving the differential equation We know that particular solutions to the equation have the form $\vec{w} = \alpha(t)\vec{e}_n$, with $\vec{e}_n$ the n-th eigenvector of C. The general solution is hence a sum of such terms, incorporating all eigenvectors.

Note also that C is symmetric and as such has real eigenvalues. We can write $C\vec{e}_k = \lambda_k \vec{e}_k$, for $1 \leq k \leq N$.

With this knowledge we easily derive:

$$\begin{aligned}\frac{d\vec{w}}{dt} = \frac{d}{dt}\alpha(t)\vec{e}_n &= a_2 C\vec{e}_n\alpha(t) \\ &= a_2\lambda_n\vec{e}_n\alpha(t)\end{aligned}$$

And:

$$\vec{w}(t) = \sum_n \exp(a_2\lambda_n t)\vec{e}_n$$

Wrapping it up We know that the eigenvalues are both real and positive, so we can order them as follows:

$$\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_N > 0$$

Hence, we can write:

$$\vec{w}(t) = e^{\lambda_1 a_2 t}\sum_k \exp(a_2(\lambda_k - \lambda_1)t)\vec{e}_k$$

It is immediately clear that since λ_1 is the largest eigenvalue, in the limit $\vec{w}$ will become parallel to $\vec{e}_1$, which is the principal component of the input set[2]. Since the postsynaptic neuron firing rate is equal to the dot product of the input rates with the weight vector, the output is actually the projection of the input onto the principal component. It is the size of the input along the axis of maximal variance. Hence, the neuron detects correlations in the input.

3.4.3 PCA

Using PCA we can detect the direction of maximal variance in the set of patterns, hence the Hebbian learning rule basically implements PCA where only the first principal component is retained.

[2]In fact, this is only true when $\langle\vec{x}\rangle = 0$, in which case the covariance equals the correlation.

We consider C to be the correlation matrix, as before. Now, also consider

$$C^0_{kj} = \langle (x_k - \langle x_k \rangle)(x_j - \langle x_j \rangle) \rangle$$

i.e. C^0 is the covariance matrix of the input.

Now, let's denote the eigenvectors of C^0 as $\vec{e}_n$ and the corresponding eigenvalues as λ_n:

$$C^0 \vec{e}_n = \lambda_n \vec{e}_n$$

Let's assume that the eigenvectors are ordered and that the vector with largest associated eigenvalue is $\vec{e}_1$. This vector is called the **principal component**.

The PCA algorithm is summarized in Listing 1.

Listing 1: PCA-algorithm

```
1. Substract mean
2. Calculate covariance matrix
3. Calculate eigenvectors
```

Dimensionality reduction PCA can be used to perform a dimensionality reduction. In this case, the principal components of a data set are determined and only the first (i.e. most significant) components are retained.

Theorem: the first component has maximal variance. We can easily prove this as follows. Let's still consider y to be the output of a neuron, as such being the dot product of the input and weight vectors. We now need to find the direction of $\vec{w}$ that will cause y to have maximal variance. The variance V is defined as:

$$\begin{aligned} V = \langle y^2 \rangle &= \langle (\vec{w} \cdot \vec{x})^2 \rangle \\ &= \vec{w}^T \langle \vec{x}\vec{x}^T \rangle \vec{w} \\ &= \vec{w}^T C^0 \vec{w} \end{aligned}$$

Now assume $\vec{w}$ is normalized and $\vec{e}_k$ are the normalized eigenvectors of C^0. Since C^0 is symmetric (see above), its eigenvectors form a orthogonal basis. Hence, we can always write:

$$\vec{w} = \sum \alpha_k \vec{e}_k$$

And since $\vec{w}$ is normalized, we have

$$\sum \alpha_k^2 = 1$$

Hence:

$$V = \sum \alpha_k^2 \vec{e}_k^T C^0 \vec{e}_k = \sum \alpha_k^2 \lambda_k$$

We can easily see that since λ_1 is the largest eigenvalue, the largest variance will be achieved when $\vec{w} = \vec{e}_1$ (because of the normalization). This proves the theorem.

We could also try to solve the problem of finding the first principal component by using Lagrange multipliers. The method of Lagrange multipliers provides a strategy for finding the local maxima and minima of a function subject to equality constraints. Consider the optimization problem maximize $f(\omega)$, subject to $g(\omega) = c$. Both f, g need to have continuous first partial derivatives. A new variable, λ called Lagrange multiplier is introduced and the Lagrange function is defined by

$$L(\lambda, \omega) = f(\omega) - \lambda(g(\omega) - c)$$

If ω_0 is a maximizer of $f(\omega)$, for the original constraint problem, then there exists λ_0 such that (ω_0, λ_0 is a stationary point for the Lagrange function. In the case of PCA, the function we'd like to maximize is the variance. If the data has zero-mean, we can write

$$f(\omega) =< y^2 >=< (\omega^T * x)^2 >= \omega^T < xx^T > \omega = \omega^T C \omega$$

And we can observe that the optimal ω is an eigenvector of the covariance matrix C. Among all the eigenvectors, ω^* is the one corresponding to the largest eigenvalue, as we try to maximize $\omega^T C \omega$ which is equal to λ.

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.

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

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.

This has the result that the norm of the weight vector tends to one. Compare this to the result obtained in Section 3.4.2, where we ended up with weights that grew exponentially without bound.

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 the value of the weights is when the average change in weights is zero. This is the point of convergence of the algorithm. Averaging the 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` (as shown in the second exercise session) and the norm of the weight vector tends to one (can be found on Wikipedia[3]). PCA IS USED FOR HEBBIAN LEARNING IN **linear** NEURONS

[3] http://en.wikipedia.org/wiki/Oja's_rule under 'Derivation'

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 statistically 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).

Structure of the section We'll first describe the model in which ICA works and sketch the rationale behind it. Then we will delve deeper into the details, focusing on preprocessing and non-gaussianity measures. Then we'll derive an ICA algorithm, first for a neural case, then a faster variant. Finally, we'll discuss some applications and also a variant on the method, temporal ICA.

3.6.1 Assumptions and rationale

A random input vector $\mathbf{x}$ is generated by a linear combination of statistically independent and stationary components (sources), $\mathbf{x} = A\mathbf{s}$. This implies:

$$p(\mathbf{s}) = p(s_1, \ldots, s_n) = p_1(s_1) \ldots p_n(s_n)$$

The task is now is to find a matrix W so as to recover the original signal sources, the components of $\mathbf{y} = \mathbf{s} = \begin{bmatrix} s_1 & s_2 & \ldots & s_n \end{bmatrix}^T$ 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 reality, recovering $\mathbf{s}$ exactly is not always possible and one tries to find an estimate $\mathbf{y}$ of $\mathbf{s}$, which is what ICA does.
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.
The central limit theorem plays a crucial role in the case of a mixture of independent random variables. This theorem states that the distribution of a sum of independent random variables tends towards a gaussian distribution.

Hence, we try to maximize the non-gaussianity of the output variables $\mathbf{y}$. Note that even the mixture of only two independent variables has a much 'more gaussian' distribution than the independent variables themselves (assuming of course that their distribution is not gaussian). This is a vital property in making ICA work for even low dimensionalities.
Independence is another crucial fact: source signals are independent, but the signal mixtures are not independent anymore, as they are built from the same source signals.

Ambiguities One application of ICA is to the cocktail party problem: we hear a mixture of many different sources. With ICA, we can try to find the sources. However, ICE will not be able to tell which speaker is which (i.e. the solution is determined but for a permutation of the columns of $\mathbf{A}$ and components of $\mathbf{y}$). Also, assuming unit variance, ICA cannot distinguish between signs and may hence output the negative of the original source. (If the variance is not normalized, the solution is determined but for any scale factor.)
For most applications, however, (e.g. audio; separation of speakers), one doesn't care about these facts.

3.6.2 Preprocessing

- 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 the natural axis? We need a 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.

3.6.3 Gaussianity vs. non-gaussianity

This approach in performing ICA is a direct generalization of the PCA technique. The Karhunen-Loève 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 sufficient for many applications).For Gaussians, the PCA is equal to ICA, as the normal distribution is only characterized by its zero, first and second moment. Gaussians are therefore statistically independent if the covariance is zero. For other distributions, this is not the case and variables are only decorrelated but not statistically independent if the covariance is equal to zero. Hence, in order to separate the variables, we need to apply ICA. careful: Cross-cumulant vs auto-cumulant cumulants vs. moments? These are actually moments?

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

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

Kurtosis The kurtosis (4-th order cumulant) can be used as a measure of non-gaussianity. The kurtosis is zero for gaussians. For super-gaussian[4] (also termed *leptokurtic*) variables, the kurtosis is > 0, whereas a sub-gaussian (also termed *platykurtic*) variable has a kurtosis < 0. This has implications in how to recover the source. In the case of super-gaussian variables, we need to maximize the kurtosis, in the case of sub-gaussian variables, we need to minimize! Without making assumptions on the actual distribution of the data, we can simply minimize the absolute value or square of the kurtosis.
The problem can now be reduced to finding a matrix, W, for which the second-order (covariances) and fourth-order cross-cumulants, kurtosis, of the transformed patterns are zero.
A disadvantage of using kurtosis is that it is very sensitive to outliers in the data set. Hence, we look into another measure of non-gaussianity.

Negentropy Negentropy[5] is based on the idea of entropy in information theory. It is only defined for discrete random variables, but an analogy exists

[4] A supergaussian variable has a spiky (large mean) pdf with heavy tails.

[5] This was, to my knowledge, not mentioned in class, but it is necessary to understand the concept of the formulas that followed.

for continuous variables: differential entropy, which is defined as:

$$H(\mathbf{y}) = -\int f(\mathbf{y}) \log f(\mathbf{y}) \mathrm{d}\mathbf{y}$$

It has been proved that *a gaussian variable has the largest (differential) entropy amongst all random variables of equal variance.* This indicates that entropy can be used as a measure of non-gaussianity.
However, to obtain a measure that is always nonnegative and zero for a gaussian variable, **negentropy** has been defined as:

$$J(\mathbf{y}) = H(\mathbf{y}_{gauss}) - H(\mathbf{y})$$

where $\mathbf{y}_{gauss}$ is a Gaussian random variable of the same covariance matrix as $\mathbf{y}$.
The advantage of negentropy is that it is a statistically justifiable measure and in some cases it even is the optimal estimator of nongaussianity. The major drawback is that it is computationally intensive to calculate it, as it needs an estimate of the pdf. Because of this, we consider approximations of the negentropy.

Approximations of negentropy A lot of approximations of negentropy have been proposed, but we'll only discuss one (the most useful one) here. This approximation is as follows:

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

where v is a Gaussian variable of zero mean and unit variance, y is also assumed to have zero mean and unit variance and F is a nonquadratic function. This measure is always non-negative and is zero when y is Gaussian. If one takes $F(y) = y^4$, the expression results in a kurtosis-based approach, but with better functions F[6], one can achieve much better results. This measure can be computed fast, which the negentropy could not, and is robust, which the kurtosis is not.
Finally, since the expression $E\{F(v)\}$ will have a constant value (F is known and v is a Gaussian variable) and since y depends on w, we can also write:

$$J(w) = [\gamma(w) - a]^2$$

[6]For instance, $G_1(u) = \frac{1}{a_1} \log \cosh a_1 u$ or $G_2(u) = -\exp(-u^2/2)$

3.6.4 ICA by neuronal rule

Let's assume a nongaussianity measure J, defined as:

$$J(\mathbf{w}) = E\{F(\mathbf{y})\}$$

where F is basically any (nonquadratic) function. If we perform gradient ascent on J, we get:

$$\begin{aligned}\Delta\mathbf{w} &= \eta\frac{\partial J}{\partial \mathbf{w}} \\ &= \eta E\{g(y)\mathbf{x}\}\end{aligned}$$

where $g = F'$. Now, assume that $y = \mathbf{w}\cdot\mathbf{x}$. We then get, working out the expectation:

$$\Delta w_n = \eta\frac{1}{p}\sum_{\mu} x_n^{\mu} g(\mathbf{w}\cdot\mathbf{x})$$

which is the batch update rule. We can see that indeed, this can be implemented by a neuron, as it naturally computes the dot product. The function g can then also be implemented by a nonlinear neuron. In vector notation, the batch rule is:

$$\Delta\mathbf{w} = \eta\,\mathbb{E}\left[\mathbf{x}g(\mathbf{w}^T\mathbf{x})\right] = \eta\frac{1}{p}\sum_{\mu}\mathbf{x}^{\mu}g(\mathbf{w}\cdot\mathbf{x})$$

Converting this to an online rule means dropping the averaging over different inputs. If we assume that η is very small, this amounts to the same result. We have:

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

3.6.5 FastICA

The algorithm is based on a fixed-point iteration scheme maximizing non-Gaussianity as a measure of statistical independence. It can be derived as an approximative Newton iteration. To derive the Newton step, we start with a Taylor expansion:

$$f(x+\Delta x) \approx f(x) + \nabla f(x)^T\Delta x + \frac{1}{2}\Delta x^T B\Delta x$$

where B is the Hessian matrix. The gradient of the Taylor approximation with respect to Δx can be written as

$$\nabla f(x+\Delta x) \approx \nabla f(x) + B\Delta x$$

Setting this gradient to zero provides the Newton step

$$\Delta x = -B^{-1} * \nabla f(x)$$

Now, again consider a measure for non-gaussianity $J(y) = E\{F(y)\}$. Also assume that we want the weight vector (i.e. the natural axis) to be normalized: $\|\mathbf{w}\| = 1$. Using Lagrange multipliers (β), we find that the optima of J will be found in poits where:

$$E\{\mathbf{x}g(\mathbf{x} \cdot \mathbf{w}) - \beta\mathbf{w}\} = 0$$

We then find the Jacobian JF of the lefthand side of this equation as:

$$JF(\mathbf{w}) = E\{\mathbf{x}\mathbf{x}^T g'(\mathbf{w} \cdot \mathbf{x})\} - \beta\mathbf{I}$$

Now we can approximate the first term of this expression as $E\{\mathbf{x}\mathbf{x}^T\}E\{g'(\mathbf{w}^T\mathbf{x})\} = E\{g'(\mathbf{w}^T\mathbf{x})\}\mathbf{I}$, which makes the Jacobian diagonal and easily invertible. This results in the following approximate Newton step:

$$\Delta\mathbf{w} = \frac{E\{\mathbf{x}g(\mathbf{w}^T\mathbf{x})\} - \beta\mathbf{w}}{E\{g'(\mathbf{w}^T\mathbf{x}) - \beta\}}$$

This can be further simplified, which yields the following algorithm, fastICA:

1. Initialize $\mathbf{w}$
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 has not converged, go to 2

3.6.6 Applications

Blind separation of sources. Separation of a set of source signals from a set of mixed signals, without the aid of information about the source signals or the mixing process.

3.6.7 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 $|\mathbf{w}_k - \mathbf{x}| \leq |\mathbf{w}_i - \mathbf{x}|$ resulting in a Voronoi tesselation of the input space. For data compression the input $\mathbf{x}$ is classified and the index k transmitted. This gives rise to the construction error

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

where C_k are the different clusters. But how do we choose the prototypes? To minimize the reconstruction error one has discretize the input space adaptively, cfr. vector quantization.

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 $\mathbf{w}_k$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 $\Delta \mathbf{w}_k = \eta(\mathbf{x}^\mu - \mathbf{w}_k)$.

Showing that the error decreases Gradient descent on the error surface yields:

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

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

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

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: $\Delta \mathbf{w}_k = 0$

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

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. We could reason on the fact that $\mathbf{w}_k$ is the center of all its assigned data points, so that after $n+1$ points, we have:

$$\begin{aligned} \mathbf{w}_k &= \frac{1}{n+1} \sum_{\mu=1}^{n+1} \mathbf{x}^\mu \\ &= \frac{1}{n+1} \sum_{\mu=1}^{n} \mathbf{x}^\mu + \frac{1}{n+1} \mathbf{x}^{n+1} \end{aligned}$$

So we have an (approximating) update rule:

$$\Delta \mathbf{w} = \frac{1}{n+1} \mathbf{x}^{n+1}$$

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. (Note that this does not guarantee the avoidance of local minima, the algorithm can still converge to a sub-optimal solution. In practice, the algorithms is sometimes repeated a few times until convergence to overcome this.)

4.2 Neuronal Implementation of Clustering

We now consider multiple postsynaptic neurons, all connected to (at least partially) the same presynaptic neurons. In the neuronal implementation the

data patterns take the form of sensory inputs to neurons in the brain, one postsynaptic neuron for every prototype $\mathbf{w}_j$. 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 $|\mathbf{w}_k - \mathbf{x}| \leq |\mathbf{w}_i - \mathbf{x}|$ can be rewritten to $\mathbf{w}_k^T\mathbf{x} \geq \mathbf{w}_i\mathbf{x}$ (just write the equation for the squares and then expand it). 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 takes 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 that the input pattern is classified to the prototype with which it forms the largest scalar product $\mathbf{w}_k \cdot \mathbf{x}^\mu$.

$$y_i(t+1) = g(\text{total drive}) = g(\sum_k w(t+1)_{ik}\mathbf{x}_k^\mu + \sum_j B_{ij}y(t)_j)$$
$$\Rightarrow y_i(t^{final}) = \begin{pmatrix} 1 \text{ for winner} \\ 0 \text{ for others} \end{pmatrix}$$

where $B_{ij} < 0$ for inhibition (and positive for excitation). Remains the question how we should update the weights.

Hebbian Learning Rule For every pattern $\mathbf{x}^\mu$, we assume that the winning prototype, or neuron, is k, with weight vector $\mathbf{w}_k$ and update all the weights according to the following (Oja) learning rule:

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

with δ_{ki} the Kronecker-delta for equality between k and i, as elaborated in the previous paragraph. The update for winner k (only neuron active) is

now the same as before (k-means), here 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

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

   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

Here Λ is a neighborhood function, whose value represents the strength of coupling between unit i and unit k during the training process. 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, where 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 elements to a reinforcement learning system.

Policy Commonly denoted by π, it defines the learning agents way of behaving at a given time. Given a perceived state and possible actions in that state, it will output the action to be taken. The policy alone is sufficient to determine the behaviour of an agent. In other 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 off in an initial state s where it has the choice between different actions a_i. Every (state,action) pair has associated with it a set of possible next states s'_j, each of which is reached with a probability $P^a_{s\to s'_j}$. A reward $R^a_{s\to s'}$ is associated with a state transition induced by a specific action a.

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 estimate them and improve this estimate 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 $Q(s,a)$.
It can be shown that aftter the mean update, using the learning rule described above, has converged, yields Q-values that solve the Bellman equation.

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

1. Being in state s, choose 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.

$$\Delta V^\pi(s) = \eta\delta(s) = \eta(R^a_{s\to s'} - V(s) + \gamma V(s'))$$

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 discount 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 in a state` (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/memories are reduced by a factor λ, except for the memory corresponding to current state j and action a^* which is additionally incremented by 1.

- 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.
Now, unlike the case without eligibility traces, for all states s and actions a we use the respective eligibility trace as an additional coefficient for the SARSA update rule and modify the TD error to be $\delta_t = r_t + \gamma Q(s', a') - Q(j, a^*)$.

$$\Delta Q(s, a) = \eta \delta_t e(j, a) = \eta \left[r_t + \gamma Q(s', a') - Q(j, a^*) \right] e(s, a), \quad \forall (s, a)$$

Reset eligibility traces at the beginning of each trial.

5.8 Continuous states

The state-action value function Q(s,a) for continuous states is approximated 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 (expresses the importance of following states s_i with respect to current s, "the signal/reward these states give to the current state", $\Phi(s - s_j)$ activation of pre-synaptic neuron) $\Rightarrow$ weights will become specific to certain action resulting in specific reward which has most influence on Q-value of current state.
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 (check sheet 8, ex.5). $\frac{dQ(s',a')}{dw_{aj}} = \Phi(s - s_j)$, the direction of the gradient is the basis function itself.

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`).

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

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

Recap: Hebbian learning is a theoretical concept exploiting statistical correlations in passive changes of the activations of presynaptic and postsynaptic neurons. It is useful for development, i.e. the wiring of neurons for receptive fields in the brain). Reinforcement learning in contrast relies on conditional changes (reward is received for policy choosing certain actions) to maximize a reward and is therefore useful for learning new behaviours.

Hebbian Learning of Actions: example: current states as input and cues as inputs for action neurons. How to model reward on top of action neurons after they have become selective to a certain combination of states and inputs?. We only want to learn the actions that are rewarded! Hebbian Learning fails. Most models of Hebbian Learning/STDP do not take into account neuromodulators, so cannot describe success/shock TODO paragraph on Spike Timing Dependent Plasticity (STDP)

Neuromodulation Neuromodulation is the physiological process by which a given neuron uses one or more neurotransmitters to regulate diverse populations of neurons. This is in contrast to classical synaptic transmission, in which one presynaptic neuron directly influences a single postsynaptic partner. Neuromodulators secreted by a small group of neurons diffuse through large areas of the nervous system, affecting multiple neurons. Examples of neuromodulators include dopamine, serotonin, acetylcholine, histamine and others.
In neuroscience, an excitatory postsynaptic potential (EPSP) is a temporary depolarization of postsynaptic membrane potential caused by the flow of positively charged ions into the postsynaptic cell as a result of opening of ligand-gated ion channels. They are the opposite of inhibitory postsynaptic potentials (IPSPs), which usually result from the flow of negative ions into the cell or positive ions out of the cell. A postsynaptic potential is defined as excitatory if it makes the neuron more likely to fire an action potential. EPSPs can also result from a decrease in outgoing positive charges, while IPSPs are sometimes caused by an increase in positive charge outflow. The flow of ions that causes an EPSP is an excitatory postsynaptic current (EPSC). EPSPs, like IPSPs, are graded (i.e. they have an additive effect). When multiple EPSPs occur on a single patch of postsynaptic membrane, their combined effect is the sum of the individual EPSPs. Larger EPSPs result in greater membrane depolarization and thus increase the likelihood that the postsynaptic cell reaches the threshold for firing an action potential.

extend theory should be good for memory formation as well as `action learning`. Therefore global factors have to be included for feedback (neuromodulator)

$$\Delta w_{ij} \propto F(\underbrace{v_i, v_j}_{local}, \underbrace{\text{SUCCESS}}_{\text{global}})$$

Timing issues in Reinforcement learning success signal is delayed (spike time scale:1-10ms, reward delay: 100-5000ms) → need memory trace (eligibility trace) implement eligibility traces as memory at synapse, same definition as in **TD**(λ)

what is winner takes all interaction here? correlation of reward in current state and reward obtained from action

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

if a is successful the elibility trace contains its reward

$$e(j,a)(t) = \gamma\lambda e(j,a)(t-\Delta t) + r_j(s_t) r_a(t)$$

Plasticity Model for Reinforcement learning spike-based model

$$\Delta w_{ij} \propto \text{success} \cdot (v_i v_j + \dots)$$

in contrast to old spike-timing dependent plasticity

$$\Delta w_{ij} \propto v_i v_j + \dots$$

synaptic update of current action a

$$\begin{aligned} \Delta w_{aj} &= \eta \delta_t r_j \\ \Delta w_{aj} &= \eta \delta_t r_j r_a \quad \text{if a taken} \end{aligned}$$

success= reward - exp. reward δ_t (dopamine neurons), success is maximized when $delta_t$ is minimized → correct prediction

Molecular mechanism changes in synaptic connections synaptic terminal (left), dendrite (right): calcium ions poor into the nerve terminal and these release neurotransmitters from small vesicles. Some of these neurotransmitters activate receptors on the target neuron which in turn opens channels letting sodium ions in TODO

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

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

5.9.2 Learning algorithm for state values V

This error will be multiplied by a learning rate α and then used to update our weights. The general update rule is

$$V_t = V_t + \alpha \left[r_{t+1} + \gamma V_{t+1} - V_t \right]$$

5.10 Biological Principles of Learning

5.10.1 Spatial Learning

Map brain: place neuronal cells - sensitive to spatial location, Environment box: place fields which are then mapped to place cells

Hippocampal place cells, depends on visual cues, works also in the dark

navigation to a hidden goal (Morris Water maze) different starting positions, task learning depends on the hippocampus code for position of the animal, learn action (North, East, South, West, etc.) towards goal after many trials the network should have learnt the right actions to take from positions in the water maze (the weights for the right actions should become specific to the appropriate positions.)

5.10.2 Unsupervised learning of place cells

from reinforcement learning of the appropriate actions for states to the unsupervised learning of place cells in the hippocampus?

Goal: in order to have the code for the position of the animal in the maze we have to learn the place cells. Derive a model that stores views of visited places. Robot in an environment: visual input at each time step:

Local view: activation of set of 9000 gabor wavelets $L(\mathbf{p}, \Phi) = \{F_k\}$ store views of visited places single view cell (VC) stores a local view all local views are stored in an incrementally growing view cells population.

Extracting direction views are aligned by current gaze directions Φ_i and their differences $\Delta\Phi = \Phi_i - \Phi_k$

Extracting position small difference between local views - spatially close positions F_i from set of filter responses

$$\text{Difference:} \quad \Delta L = |F_i - F(t)|$$
$$\text{Similarity Measure:} \quad r_i^{VC} = \exp\left(-\frac{(\Delta L)^2}{2\sigma_{VC}^2}\right)$$

Learning place cells Population of place cells created incrementally during exploration. Activity of a place: place cells are connected to grid cells of view cells? $\mathbf{w}_{ij}^{p}c$ - weight vector for place cell i

$$r^{pc} = \sum w_{ij}^{pc} r_j^{pre}$$

competitive hebbian learning:

$$\delta w_{ij}^{pc} = \eta \, |r_i^{pc} - \phi| \, (r_j^{pre} - w_{ij}^{pc})$$

Place cell based vs. landmark-based navigation Place cells: all place cells connected to all possible action cells(nucleus accumbens) (important weights w_{ij} are learnt). Learnt actions result in correct movement towards goal.
Landmark-based navigation: Visual filters connected to action cells (dorsal striatum), so that agent turns towards landmark

- Unsupervised learning leads to interaction between vision and path integration
- Model of landmark-based navigation
- Model of map-based navigation
- Learning of actions: `reinforcement learning`
- Learning of views and places: `unsupervised learning`

Biological mechanisms: grid cells firing grids extend over the whole space. Subpopulation with different grid sizes exist. Grid cells project to the CA1 area. Grid-cell firing fields rotate following the rotation of the cue. Biological locus of the path integration (elecrode in dorsomedial entorhinal cortex)

5.11 Policy Gradient Methods

Policy gradient methods are a type of reinforcement learning techniques that rely upon optimizing `parametrized policies` π_θ with respect to the `expected return` (long-term cumulative reward) by gradient descent.

5.11.1 Limitations of TD methods (Q-learning & SARSA)

Reinforcement learning is probably the most general framework in which reward-related learning problems of animals, humans or machine can be phrased. However, most of the methods proposed in the reinforcement learning community are not yet applicable to many problems such as robotics, motor control, etc. due to the following reasons. The motivation for policy gradient methods is that they do not suffer from many of the problems that have been marring traditional reinforcement learning approaches:

- **Lack of guarantees of a value function?:** In continuous observation domains (environment) function approximations which are hard to select/tune are necessary for these methods
- **Intractability:** In only partially observable, `non-markovian` settings, TD algorithms are shown to be fundamentally flawed. Because of the uncertain state information the learning systems need to be modeled as partially observable Markov decision problems which often results in excessive computational cost.
- **Convergence:** Even in fully observable, `markovian` settings, TD algorithms, e.g. Q-learning, most traditional reinforcement learning methods using function approximators have no convergence guarantees and there exist even divergence examples.
- **Complexity:** Continuous states and actions in high dimensional spaces cannot be treated by most reinforcement learning approaches using TD methods, because they are difficult to represent using TD methods.

Policy gradient methods differ significantly as they do not suffer from these problems in the same way. For example, uncertainty in the state might degrade the performance of the policy (if no additional state estimator is being used) but the optimization techniques for the policy do not need to be changed. Continuous states and actions can be dealt with in exactly the same way as discrete ones while, in addition, the learning performance is often increased. `Convergence` at least to a local optimum is `guaranteed`.

5.11.2 Mathematical Formulation

UNDERSTAND: by associating certain actions with certain stimuli in a stochastic fashion???

Goal: In policy optimization the goal is to optimize the expected reward/reward directly

$$J(\theta) = E\left[\sum_{k=0}^{H} a_k r_k\right]$$

according to the parameters $\theta \in \mathbb{R}^K$ of the policy π_θ, where a_k denote time-step dependent weighting factors, often set to $a_k = \gamma y_k$ for discounted reinforcement learning (where γ is in [0,1]) or $a_k = \frac{1}{H}$ for the average reward case.

Basic approach: We model the learning system for a policy gradient method in a discrete-time manner and we will denote the current time step by k. The input to the system is described using a probability distribution $s_{k+1} \sim (s_{k+1}|s_k, a_k)$ as model where $s_k \in \mathbb{R}^M$ denotes the current action, and $s_k,\ s_{k+1} \in \mathbb{R}^N$ denote the current and next state, respectively. We furthermore assume that actions are generated by a policy $a_k \sim \pi_\theta(a_k \| s_k)$ which is modeled as a probability distribution in order to incorporate exploratory actions. The policy is assumed to be parameterized by K policy parameters $\theta \in \mathbb{R}^K$ The sequence of states and actions forms a `trajectory` denoted by $\tau = [s_0 : H, a_0 : H]$ where H denotes the `horizon` which can be infinite.The words trajectory, history, trial, or roll-out are used interchangeably. At each instant of time, the learning system receives a reward denoted by $r_k = r(s_k, a_k) \in \mathbb{R}$.

Optimization: For real-world applications, any change to the policy parameterization has to be smooth because drastic changes can be hazardous for the actor. Furthermore useful initializations of the policy based on domain knowledge would otherwise vanish after a single update step. Therefore the policy method of choice is steepest descent on the expected return. The policy parametrization is thus updated according to the following gradient update rule

$$\theta_{h+1} = \theta_h + \alpha_h \nabla_\theta J|_{\theta=\theta_h}$$

where $\alpha \in \mathbb{R}^+$ denotes a learning rate and $h \in \{0, 1, 2, \dots\}$ the current update number. The time step k and update number h are two different

variables. In actor-critic-based policy gradient methods, the frequency of updates of h can be nearly as high as of k. However, in most episodic methods, the policy update h will be significantly less frequent.
The `main problem` in policy gradient methods is to obtain a good estimator of the policy gradient $\nabla_\theta J|_{\theta=\theta_h}$

5.11.3 Likelihood Ratio Method

Assume that trajectories τ are generated from the policy distribution of a system $\tau \sim p\theta(\tau) = p(\tau|\theta)$ with return $r(\tau) = \sum_{k=0}^{H} a_k r_k$ which leads to $J(\theta) = \mathbb{E}[r(\tau)] = \int_T p_\theta(\tau) r(\tau) d\tau_k$. In this case, the policy gradient can be estimated using the likelihood ratio better known as REINFORCE (Williams, 1992) trick, i.e., by using

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau) \nabla_\theta \log p_\theta(\tau)$$

from standard differential calculus: $\nabla_\theta \log p_\theta(\tau) = \nabla_\theta p_\theta(\tau) / p_\theta(\tau)$ we obtain

$$\begin{aligned} \nabla_\theta J(\theta) &= \int_{\mathbf{T}} \nabla_\theta p_\theta(\tau) r(\tau) d\tau \\ &= \int_{\mathbf{T}} p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) r(\tau) d\tau \\ &= E[\nabla_\theta \log p_\theta(\tau) r(\tau)]. \end{aligned}$$

As the expectation $\mathbb{E}[\cdot]$ can be replaced by sample averages, denoted by $\langle \cdot \rangle$, only the derivative $\nabla_\theta \log p_\theta(\tau)$ is needed for the estimator. Importantly, this derivative can be computed without knowledge of the generating distribution $p_\theta(\tau)$ as

$$p_\theta(\tau) = p(\mathbf{s}_0) \prod_{k=0}^{H} p(\mathbf{s}_{k+1} | \mathbf{s}_k, \mathbf{a}_k) \pi_\theta(\mathbf{a}_k | \mathbf{s}_k)$$

implies that

$$\nabla_\theta \log p_\theta(\tau) = \sum_{k=0}^{H} \nabla_\theta \log \pi_\theta(\mathbf{a}_k | \mathbf{s}_k)$$

as only the policy depends on θ. By taking the sample average as Monte Carlo (MC) approximation of this expectation by taking N trial histories we get

$$\nabla_\theta J = \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \log p_\theta(\tau_\mathbf{n}) r(\tau_\mathbf{n}).$$

which is a fast approximation of the policy gradient for the current policy.
N TRIALS OR N TRIAL HISTORIES OF ONE TRIAL

5.11.4 Drawbacks of Policy Gradient

Of course, policy gradients are not the salvation to all problems having significant problems themselves. They are by definition on-policy (note that tricks like importance sampling can slightly alleviate this problem) and need to `forget data very fast` in order to avoid the `introduction of a bias` to the gradient estimator. **Hence, the use of sampled data is not very efficient**. In tabular representations **???**, value function methods are guaranteed to converge to a global maximum while policy gradients only converge to a local maximum and there may be many maxima in discrete problems.

5.12 Neuronal implementation of Policy Gradient Methods

TODO: Biology, Neurons, Hebb, Perceptron → mathing law

6 3 Factor rules

6.1 Integrate and fire model

- the membrane potential v(t) defines the state of the neuron at time t
- Each spike causes an excursion of the membrane potential with amplitude w
- The membrane potential decays towards its resting value V_0
- If v(t) reaches a threshold value V_th, a spike is produced and v(t) is reset to its resting value

membrane potential?

$$C_m \frac{dv(t)}{dt} = I_{\text{leak}}(t) + I_S(t) + I_{\text{inj}}(t)$$

decay towards resting value V_0

$$I_leak(t) = -\frac{C_m}{\tau_m}\left[v(t) - V_0\right]$$

Response to constant input sub-threshold stimulus

$$v(t) = V_0 + IR_m(1 - \exp(-(t - t_0)/\tau_m))$$

supra-theshold stimulus

$$T_{\text{ISI}} = -\tau_m \ln\left[1 - \frac{\theta}{IR_m}\right], \quad (IR_m > \theta)$$

where $\theta = (V_{th} - V_0)$

Response to spike(delta) input supra-threshold stimulus $I_leak(t)$ at Φ leads to post-synaptic potential with Weight J and Delay D:

$$v_\delta(t) = J \cdot \exp(-\frac{(t - t_{sp} - D)}{\tau_m})$$

Response to arbitrary input for any linear system, response to an arbitrary stimulus is given by the convolution of the impulse response with the stimulus

$$\begin{aligned} v(t) &= V_0 + \frac{1}{C_m}\int_{t_0}^{t} \exp(-\frac{(t - t')}{\tau_m}) \cdot I_{\text{Inj}}(t')dt' \\ &= V_0 + \frac{1}{C_m}(v_\delta * I_{\text{Inj}})(t) \end{aligned}$$

Linear system means:

- if v(t) is a solution, then $\alpha v(t)$ is also a solution, for all $\alpha \in \mathbb{R}$
- If u(t) and v(t) are solutions, then u(t)+v(t) is also a solution

6.2 Spike response model with stochastic firing

membrane potential **u** is potential

$$u_i(t|\text{input}) = u_{rest} + \sum_{t_j^f} \varepsilon(t - t_j^f) + \sum_{t_j^f} \eta(t - t_j^f \eta(t - t_j^f)$$

t_j^f - time t wich presynaptic neuron fires

probabilistic spike generation the higher the potential (of the postsynaptic neuron) the more likely the neuron fires

$$p(t|u_i) = g(u(t)) \propto \exp(\beta u(t))$$

Probability of Spike Train $x_j(t) = \sum_j \delta(t - t_j^f)$, $\varepsilon(t - t_j^f)$, $y(t) = \sum_n \delta(t - t^n)$

$$P(y|x) = \prod_n p(t|\hat{t}^n) \exp\left[-\int_{\hat{t}^{n-1}}^{\hat{t}^n} p(t'|\hat{t}^{n-1})dt'\right]$$

potential, $\hat{t}$=refractoriness, last spike

$$u(t) = \sum_{j,f} w_j \varepsilon \underbrace{(t - t_j^f)}_{\text{all spikes, all neurons}} + \eta(t - \hat{t})$$

stochastic firing

$$p(t) = g(u(t))$$

6.3 R-max

6.3.1 Derivation

optimization of an objective function, reward maximization by gradient ascent

$$\frac{d}{dt} w_{ij}(t) = \eta \frac{d}{dw_{ij}} \langle R \rangle \quad = \eta \langle R(\text{spiketrain, input}) \frac{d}{dw_{ij}} \log P(\text{spiketrain}|\text{ input}) \rangle$$

Batch to online

$$\frac{d}{dt} w_{ij}(t) \propto R(t) \int_{start}^{t} dt' \underbrace{\varepsilon(t' - t_j^{pre})}_{\text{EPSP}} \left[\underbrace{\delta(t' - t_i^f) - p(u(t'))}_{\text{Post}}\right]$$

6.3.2 Rule

Simple activity model Hebb rule:

$$\frac{d}{dt} = H_{aj}(t) = r(s_t) r_a(t) - g \cdot H_a j(t)$$

spiking hebb rule model:

$$\frac{d}{dt} H_{aj}(t) \underbrace{\varepsilon(t' - t_j^{pre})}_{\text{EPSP}} \left[\underbrace{\delta(t' - t_i^f) - p(u(t'))}_{\text{Post}}\right] - g \cdot H_a j(t)$$

$$\frac{d}{dt} w_{aj} = \eta R_t H_{aj}$$

R_t success signal=phasic dopamine Hebb learning rule is EPSP $\left[\delta(t - t_i^f) - p(u)\right]$

$\langle H_{ij} \rangle = 0$, repeating a fixed input has no Hebb-learning effect

6.4 R-STDP

$$\Delta w_{ij} \propto S(R) \times \text{STDP}$$

$$\Delta w_{ij} \propto S(R) \times f(\text{pre}, \text{post}) \qquad = S(R) \times H_{ij}$$

Hebbian Learning Rule is STDP
$\langle H_{ij} \rangle \neq 0$, repeating a fixed input gives a hebb-learning effect

6.5 When do 3-factor rules work?

both rules discussed have the form

$$\begin{aligned} \Delta w_{ij} &\propto S \cdot H_{ij} \\ &= \underbrace{\langle S \cdot H_{ij} \rangle - \langle S \rangle \langle H_{ij} \rangle}_{\text{good: cov(R,H)}} + \underbrace{\langle S \rangle \langle H_{ij} \rangle}_{\text{bad}} \end{aligned}$$

$\langle S \rangle \langle H_{ij} \rangle$ has to be neutralized, Always the case for R-max ($\langle H_{ij} \rangle = 0$)
For R-STDP use $R \to S(R) = R - \langle R \rangle$ (Success=Reward-exp.Reward)
Good three factor rules are either

- `useless` for unsupervised hebbian learning
- or the average neuromodulatory reward signal S must be zero $S = R - \langle R \rangle \Rightarrow$ system must know running average of expected reward $\langle R \rangle$

6.6 Application to Hand Movements

6.6.1 How are movements represented

- muscle coordinates, vector of muscle activations over time
- Joint coordinates - vector of joint angles over time
- World coordinates - vector of positions in 3D space

6.6.2 Detour: neural encoding of movement direction

monkey pushes red button - experiment, reaching movements involve many muscles, neurons in motor cortex are tuned to movement direction, tuning curve for motor neurons, accuracy of population code

7 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.

7.1 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:

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

7.2 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):

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