Introduction and topics of the module

- Overview of hybrid knowledge representation
- Neural networks introduction
- Learning in multilayer and recurrent networks
- Localist, distributed learning and shape recognition
- Neural network architectures
- Hybrid architectures
- Neuroscience-inspired architectures
- MLP, SOM, SRN, ESN, CNN Architectures
- Bioinspired robotic architectures
- …

Agents need to reason, communicate, learn and develop many complex tasks…

How to do it? Rule-Based Systems?

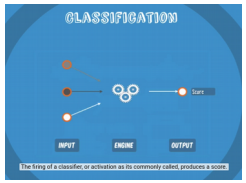
- Use rules to represent knowledge in an **IF…THEN…** form or more complex formalisms
- Use an "inference engine" to **chain the rules** together in different ways
- Allows the ability to **explain decision** by tracing which rules are used and when
- Strict symbolic logic rules alone may be brittle… extensions for learning and robustness possible?

Fuzzy Logic

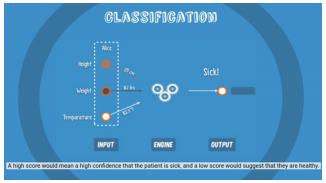
- Based on human reasoning which is imprecise
- Uses a "membership function" to describe how strongly something belongs to a group
- Membership functions can be learned adaptively using for instance neural techniques
- Leads to interesting hybrid neural symbolic architectures

NN: There is an interconnected web of nodes, which are called neurons, and the edges that join them together.
A NN main func is to receive a set of inputs, and then use the output to solve a problem.

NN app: classification. A process of categorizing a group of objects, while only using some basic data features that describe them.
Classification: Logistic Regression, SVM, Naive Bayes, and NN.
The firing of a classifier / activation as its commonly called, produces a score.



The classifier would receive this data about the patient, process it and fire out a confidence score.

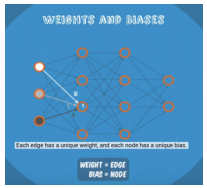


A neural net can be viewed as the result of spinning classifiers together in a layered web. This is because each node in the hidden and output layers has its own classifier.

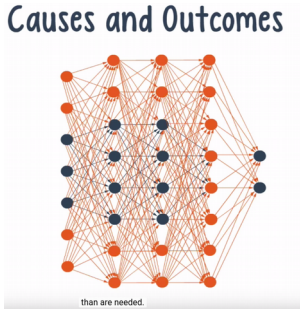
In hidden layer, it gets its inputs from input layer, and activates. its score is then passed on as input to the next hidden layer for further activation.
forward prop is a nerual nets way of classifying a set of inputs.

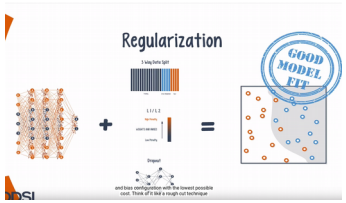
MLP
The first NN were born out of the need to address the inaccuracy of an early classifier, the perceptron.
it was shown that by using a layered web of perceptrons, the acc could be inproved.

set of inputs is modified by unique weights and biases.
Each edge has a unique weight and each node has a unique bias.
weight = edge
bias = node



this means that the combination used for each activation is also unique, which explains why the nodes fire differently.





Exam 1 (Prof. Wermter):

1. What is the difference between MLPs and Deep Learning?
2. How can we reduce the vanishing gradient problem in Deep Learning?
3. What are Rectified Linear Units (because I mentioned them)? What advantage do they have? What disadvantages could you think of? What is Softplus?
4. When we want to evaluate the output of a network/model, what kind of measurements can we use?
5. How does a Convolutional Neural Network work and what is trained? What is the difference between Convolution and Pooling (he referred to Pooling as Subsampling)
6. Can you see a connection between Momentum, Hysteresis (in Recurrent Plausibility Networks), Leaking Rate (in Echo State Networks)? (I added the τ in Multiple Timescale Recurrent Neural Networks)
7. What is Dropout and why do we use it?

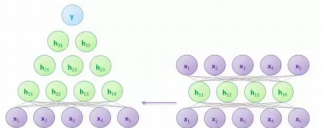
others have said, MLP is not really different than deep learning, but arguably just one type of deep learning.

Back-propagation (which has existed for decades) theoretically allows you to train a network with many layers. But before the advent of deep learning, researchers did not have widespread success training neural networks with more than 2 layers.

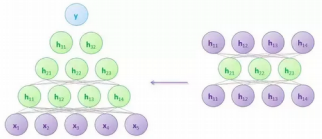
This was mostly because of vanishing and/or exploding gradients. Prior to deep learning MLPs were typically initialized using random numbers. Like today, MLPs used the gradient of the network's parameters w.r.t. to the network's error to adjust the parameters to better values in each training iteration. In back propagation, to evaluate this gradient involves the chain rule and you must multiply each layer's parameters and gradients together across all the layers. This is a lot of multiplication, especially for networks with more than 2 layers. If most of the weights across many layers are less than 1 and they are multiplied many times then eventually the gradient just vanishes into a machine-zero and training stops. If most of the parameters across many layers are greater than 1 and they are multiplied many times then eventually the gradient explodes into a huge number and the training process becomes intractable.

Deep learning proposed a new initialization strategy: use a series of single layer networks - which do not suffer from vanishing/exploding gradients - to find the initial parameters for a deep MLP. The pictures below attempt to illustrate this process:

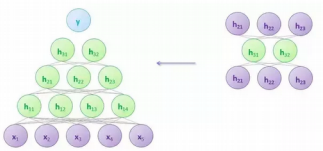
1.) A single layer autoencoder network is used to find initial parameters for the first layer of a deep MLP.



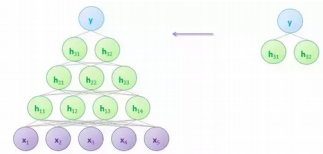
2.) A single layer autoencoder network is used to find initial parameters for the second layer of a deep MLP.



3.) A single layer autoencoder network is used to find initial parameters for the third layer of a deep MLP.



4.) A softmax classifier (logistic regression) is used to find initial parameters for the output layer of a deep MLP.



Now that all the layers have been initialized through this pre-training process to values that are more suitable for the data, you can usually train the deep MLP using gradient descent techniques without the problem of vanishing/exploding gradients.

Of course the field of deep learning has moved forward since this initial breakthrough, and many researchers now argue pre-training is not necessary. But even without pre-training, reliably training a deep MLP requires some additional sophistication, either in the initialization or training process beyond the older MLP training approaches of random initialization followed by standard gradient descent.

The vanishing gradient problem requires us to use small learning rates with gradient descent which then needs many small steps to converge. This is a problem if you have a slow computer which takes a long time for each step. If you have a fast GPU which can perform many more steps in a day, this is less of a problem.

There are several ways to tackle the vanishing gradient problem. I would guess that the largest effect for CNNs came from switching from sigmoid nonlinear units to rectified linear units. If you consider a simple neural network whose error E depends on weight w_{ij} only through y_j, where

$$y_j = f\left(\sum_i w_{ij} x_i\right),$$

its gradient is

$$\frac{\partial}{\partial w_{ij}} E = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \cdot f'\left(\sum_i w_{ij} x_i\right) x_i.$$

If f is the logistic sigmoid function, f' will be close to zero for large inputs as well as small inputs. If f is a rectified linear unit,

$$f(u) = \max(0, u),$$

its gradient is zero only for negative inputs and large for large inputs. Another important contribution comes from properly initializing the weights. This paper looks like a good source for understanding the challenges in more details (although I haven't read it yet):

Problem

Gradient based methods learn a parameter's value by understanding how a small change in the parameter's value will affect the network's output. If a change in the parameter's value causes very small change in the network's output - the network just can't learn the parameter effectively, which is a problem.

This is exactly what's happening in the vanishing gradient problem -- the gradients of the network's output with respect to the parameters in the early layers become extremely small. That's a fancy way of saying that even a large change in the value of parameters for the early layers doesn't have a big effect on the output. Let's try to understand when and why does this problem happen.

Cause

Vanishing gradient problem depends on the choice of the activation function. Many common activation functions (e.g sigmoid or tanh) 'squash' their input into a very small output range in a very non-linear fashion. For example, sigmoid maps the real number line onto a "small" range of [0, 1]. As a result, there are large regions of the input space which are mapped to an extremely small range. In these regions of the input space, even a large change in the input will produce a small change in the output - hence the gradient is small.

This becomes much worse when we stack multiple layers of such non-linearities on top of each other. For instance, first layer will map a large input region to a smaller output region, which will be mapped to an even smaller region by the second layer, which will be mapped to an even smaller region by the third layer and so on. As a result, even a large change in the parameters of the first layer doesn't change the output much.

We can avoid this problem by using activation functions which don't have this property of 'squashing' the input space into a small region. A popular choice is Rectified Linear Unit which maps x to $max(0, x)$.

Hopefully, this helps you understand the problem of vanishing gradients. I'd also recommend reading along this ☞ IPython notebook which does a small experiment to understand and visualize this problem, as well as highlights the difference between the behavior of sigmoid and rectified linear units.

Two additional major benefits of ReLUs are sparsity and a reduced likelihood of vanishing gradient. But first recall the definition of a ReLU is $h = \max(0, a)$ where $a = Wx + b$.

One major benefit is the reduced likelihood of the gradient to vanish. This arises when $a > 0$. In this regime the gradient has a constant value. In contrast, the gradient of sigmoids becomes increasingly small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning.

The other benefit of ReLUs is sparsity. Sparsity arises when $a \leq 0$. The more such units that exist in a layer the more sparse the resulting representation. Sigmoids on the other hand are always likely to generate some non-zero value resulting in dense representations. Sparse representations seem to be more beneficial than dense representations.

sparsity refers to several weights, or coefficients in regression, being equal to zero. – rafaelvalle Mar 2 at 7:01

Advantage:

- Sigmoid: not blowing up activation
- Relu : not vanishing gradient

In other words, **Disadvantage:**

- Sigmoid: tend to vanish gradient (cause there is a mechanism to reduce the gradient as "a" increase, where "a" is the input of a sigmoid function. Gradient of Sigmoid: $S'(a) = S(a)(1 - S(a))$. When "a" grows to infinite large , $S'(a) = S(a)(1 - S(a)) = 1 \times (1 - 1) = 0$.
- Relu : tend to blow up activation (there is no mechanism to constrain the output of the neuron, as "a" itself is the output)

There are two main benefits:

- ReLUs are much simpler computationally. The forward and backward passes through an ReLU are both just a simple if statement. Compare this to the sigmoid activation, which requires computing an exponent. This advantage is huge when dealing with big networks with many neurons, and can significantly reduce both training and evaluation times.
- Sigmoid activations are easier to saturate. There is a comparatively narrow interval of inputs for which the sigmoid's derivative is *sufficiently nonzero*. In other words, once a sigmoid reaches either the left or right plateau, it is almost meaningless to make a backward pass through it, since the derivative is very close to 0. On the other hand, ReLUs only saturate when the input is less than 0. And even this saturation can be eliminated by using leaky ReLUs. For very deep networks, saturation hampers learning, and so ReLUs provide a nice workaround.

Sigmoid unit :
$f(x) = \frac{1}{1+exp(-x)}$

Tanh unit:
$f(x) = tanh(x)$

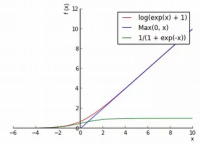
Rectified linear unit (ReLU):
$f(x) = \sum_{i=1}^{inf} \sigma(x - i + 0.5) \approx log(1 + e^x)$

we refer

- $\sum_{i=1}^{inf} \sigma(x - i + 0.5)$ as **stepped sigmoid**
- $log(1 + e^x)$ as **softplus function**

The **softplus** function can be approximated by **max function** (**or hard max**) ie $max(0, x + \mathcal{N}(0, 1))$. The max function is commonly known as **Rectified Linear Function (ReL).**

In the following figure, we graphically compare ReL function (soft/hard) with sigmoid function.



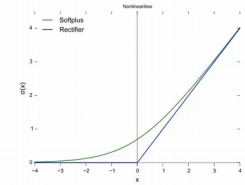
The major differences between the sigmoid and ReL function are:

- Sigmoid function has range [0,1] whereas the ReL function has range $[0, \infty]$. Hence sigmoid function can be used to model probability, whereas ReL can be used to model positive real number. NOTE: The view of **softplus** function as approximation of stepped sigmoid units relates to the binomial hidden units as discussed in http://machinelearning.wustl.edu... ☞
- The gradient of the sigmoid function vanishes as we increase or decrease x. However, the gradient of the ReL function doesn't vanish as we increase x. In fact, for max function, gradient is defined as $\begin{pmatrix} 0 & if\ x < 0 \\ 1 & if\ x > 0 \end{pmatrix}$.

The advantages of using Rectified Linear Units in neural networks are

- If hard max function is used as activation function, it induces the sparsity in the hidden units.
- ReLU doesn't face gradient vanishing problem as with sigmoid and tanh function. Also, It has been shown that deep networks can be trained efficiently using ReLU even without pre-training.

ReLU is defined as The **softplus** ☞ is its differential surrogate and is defined as .



Both the ReLU and **Softplus** are largely similar, except near 0 where the **softplus** is enticingly smooth and differentiable. It's much easier and efficient to compute ReLU and its derivative than for the **softplus** function which has log(.) and exp(.) in its formulation. Interestingly, the derivative of the **softplus** function is the logistic function: .

In deep learning, computing the activation function and its derivative is as frequent as addition and subtraction in arithmetic. By switching to ReLU, the forward and backward passes are much faster while retaining the non-linear nature of the activation function required for deep neural networks to be useful.

Precision

Precision is the number of True Positives divided by the number of True Positives and False Positives. Put another way, it is the number of positive predictions divided by the total number of positive class values predicted. It is also called the Positive Predictive Value (PPV).

Precision can be thought of as a measure of a classifiers exactness. A low precision can also indicate a large number of False Positives.

- The precision of the All No Recurrence model is 0/(0+0) or not a number, or 0.
- The precision of the All Recurrence model is 85/(85+201) or 0.30.
- The precision of the CART model is 10/(10+13) or 0.43.

The precision suggests CART is a better model and that the All Recurrence is more useful than the All No Recurrence model even though it has a lower accuracy. The difference in precision between the All Recurrence model and the CART can be explained by the large number of False Positives predicted by the All Recurrence model.

Recall

Recall is the number of True Positives divided by the number of True Positives and the number of False Negatives. Put another way it is the number of positive predictions divided by the number of positive class values in the test data. It is also called Sensitivity or the True Positive Rate.

Recall can be thought of as a measure of a classifiers completeness. A low **recall** indicates many False Negatives.

- The **recall** of the All No Recurrence model is 0/(0+85) or 0.
- The **recall** of the All Recurrence model is 85/(85+0) or 1.
- The **recall** of CART is 10/(10+75) or 0.12.

As you would expect, the All Recurrence model has a perfect **recall** because it predicts "recurrence" for all instances. The **recall** for CART is lower than that of the All Recurrence model. This can be explained by the large number (75) of False Negatives predicted by the CART model.

F1 Score

The F1 Score is the 2*((precision*recall)/(precision+recall)). It is also called the F Score or the F Measure. Put another way, the F1 score conveys the balance between the precision and the **recall**.

- The F1 for the All No Recurrence model is 2*((0*0)/0+0) or 0.
- The F1 for the All Recurrence model is 2*((0.3*1)/0.3+1) or 0.46.
- The F1 for the CART model is 2*((0.43*12)/0.43+12) or 0.19.

If we were looking to select a model based on a balance between precision and **recall**, the F1 measure suggests that All Recurrence model is the one to beat and that CART model is not yet sufficiently competitive.

In machine learning, a **convolutional neural network** (**CNN**, or **ConvNet**) is a class of deep, feed-forward artificial neural network that have successfully been applied to analyzing visual imagery.

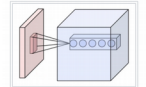
CNNs use a variation of multilayer perceptrons designed to require minimal preprocessing.[1] They are also known as **shift invariant** or **space invariant artificial neural networks** (**SIANN**), based on their shared-weights architecture and translation invariance characteristics.[2][3]

Convolutional networks were inspired by biological processes[4] in which the connectivity pattern between neurons is inspired by the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.

They have applications in image and video recognition, recommender systems[5] and natural language processing.[6]

A CNN architecture is formed by a stack of distinct layers that transform the input volume into an output volume (e.g. holding the class scores) through a differentiable function. A few distinct types of layers are commonly used. We discuss them further below:



Neurons of a convolutional layer (blue), connected to their receptive field (red)

Convolutional layer [edit]

The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input.

Stacking the activation maps for all filters along the depth dimension forms the full output volume of the convolution layer. Every entry in the output volume can thus also be interpreted as an output of a neuron that looks at a small region in the input and shares parameters with neurons in the same activation map.

Local connectivity [edit]

When dealing with high-dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume because such a network architecture does not take the spatial structure of the data into account. Convolutional networks exploit spatially local correlation by enforcing a local connectivity pattern between neurons of adjacent layers: each neuron is connected to only a small region of the input volume. The extent of this connectivity is a hyperparameter called the receptive field of the neuron. The connections are local in space (along width and height), but always extend along the entire depth of the input volume. Such an architecture ensures that the learnt filters produce the strongest response to a spatially local input pattern.

Spatial arrangement [edit]

Three hyperparameters control the size of the output volume of the convolutional layer: the depth, stride and zero-padding.

- The depth of the output volume controls the number of neurons in a layer that connect to the same region of the input volume. These neurons learn to activate for different features in the input. For example, if the first convolutional layer takes the raw image as input, then different neurons along the depth dimension may activate in the presence of various oriented edges, or blobs of color.
- Stride controls how depth columns around the spatial dimensions (width and height) are allocated. When the stride is 1 then we move the filters one pixel at a time. This leads to heavily overlapping receptive fields between the columns, and also to large output volumes. When the stride is 2 (or rarely 3 or more) then the filters jump 2 pixels at a time as they slide around. The receptive fields overlap less and the resulting output volume has smaller spatial dimensions.[33]
- Sometimes it is convenient to pad the input with zeros on the border of the input volume. The size of this padding is a third hyperparameter. Padding provides control of the output volume spatial size. In particular, sometimes it is desirable to exactly preserve the spatial size of the input volume.

Parameter sharing [edit]

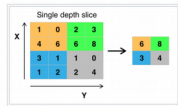
A parameter sharing scheme is used in convolutional layers to control the number of free parameters. It relies on one reasonable assumption: That if a patch feature is useful to compute at some spatial position, then it should also be useful to compute at other positions. In other words, denoting a single 2-dimensional slice of depth as a **depth slice**, we constrain the neurons in each depth slice to use the same weights and bias.

Since all neurons in a single depth slice share the same parameters, then the forward pass in each depth slice of the CONV layer can be computed as a convolution of the neuron's weights with the input volume (hence the name: convolutional layer). Therefore, it is common to refer to the sets of weights as a filter (or a kernel), which is convolved with the input. The result of this convolution is an activation map, and the set of activation maps for each different filter are stacked together along the depth dimension to produce the output volume. Parameter sharing contributes to the translation invariance of the CNN architecture.

Sometimes the parameter sharing assumption may not make sense. This is especially the case when the input images to a CNN have some specific centered structure, in which we expect completely different features to be learned on different spatial locations. One practical example is when the input are faces that have been centered in the image: we might expect different eye-specific or hair-specific features to be learned in different parts of the image. In that case it is common to relax the parameter sharing scheme, and instead simply call the layer a locally connected layer.

Pooling layer [edit]

Another important concept of CNNs is pooling, which is a form of non-linear down-sampling. There are several non-linear functions to implement pooling among which *max pooling* is the most common. It partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum. The intuition is that the exact location of a feature is less important than its rough location relative to other features. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters and amount of computation in the network, and hence to also control overfitting. It is common to periodically insert a pooling layer between successive convolutional layers in a CNN architecture. The pooling operation provides another form of translation invariance.



Max pooling with a 2x2 filter and stride = 2

The pooling layer operates independently on every depth slice of the input and resizes it spatially. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples at every depth slice in the input by 2 along both width and height, discarding 75% of the activations. In this case, every max operation is over 4 numbers. The depth dimension remains unchanged.

In addition to max pooling, the pooling units can use other functions, such as average pooling or L2-norm pooling. Average pooling was often used historically but has recently fallen out of favor compared to max pooling, which works better in practice.[34]

Due to the aggressive reduction in the size of the representation, the trend is towards using smaller filters[35] or discarding the pooling layer

Region of Interest pooling (also known as RoI pooling) is a variant of max pooling, in which output size is fixed and input rectangle is a parameter.[37]

Pooling is an important component of convolutional neural networks for object detection based on Fast R-CNN[38] architecture.

ReLU layer [edit]

ReLU is the abbreviation of Rectified Linear Units. This layer applies the non-saturating activation function $f(x) = \max(0, x)$. It increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer.

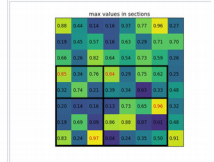
Other functions are also used to increase nonlinearity, for example the saturating hyperbolic tangent $f(x) = \tanh(x)$, $f(x) = |\tanh(x)|$, and the sigmoid function $f(x) = (1 + e^{-x})^{-1}$. ReLU is preferable to other functions, because it trains the neural network several times faster[39] without a significant penalty to generalisation accuracy.

Fully connected layer [edit]

Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have connections to all activations in the previous layer, as seen in regular neural networks. Their activation can hence be computed with a matrix multiplication followed by a bias offset.

Loss layer [edit]

The loss layer specifies how training penalizes the deviation between the predicted and true labels and is normally the final layer. Various loss functions appropriate for different tasks may be used there. Softmax loss is used for predicting a single class of K mutually exclusive classes. Sigmoid cross-entropy loss is used for predicting K independent probability values in $[0, 1]$. Euclidean loss is used for regressing to real-value labels $(-\infty, \infty)$.



RoI pooling to size 2x2. In this example region proposal (an input parameter) has size 7x5.

Dropout [edit]

Because a fully connected layer occupies most of the parameters, it is prone to overfitting. One method to reduce overfitting is dropout.[40][41] At each training stage, individual nodes are either "dropped out" of the net with probability $1 - p$ or kept with probability p, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed. Only the reduced network is trained on the data in that stage. The removed nodes are then reinserted into the network with their original weights.

In the training stages, the probability that a hidden node will be dropped is usually 0.5; for input nodes, this should be much lower, intuitively because information is directly lost when input nodes are ignored.

At testing time after training has finished, we would ideally like to find a sample average of all possible 2^n dropped-out networks; unfortunately this is unfeasible for large values of n. However, we can find an approximation by using the full network with each node's output weighted by a factor of p, so the expected value of the output of any node is the same as in the training stages. This is the biggest contribution of the dropout method: although it effectively generates 2^n neural nets, and as such allows for model combination, at test time only a single network needs to be tested.

By avoiding training all nodes on all training data, dropout decreases overfitting in neural nets. The method also significantly improves the speed of training. This makes model combination practical, even for deep neural nets. The technique seems to reduce node interactions, leading them to learn more robust features that better generalize to new data.

DropConnect [edit]

DropConnect[42] is the generalization of dropout in which each connection, rather than each output unit, can be dropped with probability $1 - p$. Each unit thus receives input from a random subset of units in the previous layer.

DropConnect is similar to dropout as it introduces dynamic sparsity within the model, but differs in that the sparsity is on the weights, rather than the output vectors of a layer. In other words, the fully connected layer with DropConnect becomes a sparsely connected layer in which the connections are chosen at random during the training stage.

Stochastic pooling [edit]

A major drawback to Dropout is that it does not have the same benefits for convolutional layers, where the neurons are not fully connected.

In stochastic pooling,[43] the conventional deterministic pooling operations are replaced with a stochastic procedure, where the activation within each pooling region is picked randomly according to a multinomial distribution, given by the activities within the pooling region. The approach is hyperparameter free and can be combined with other regularization approaches, such as dropout and data augmentation.

An alternate view of stochastic pooling is that it is equivalent to standard max pooling but with many copies of an input image, each having small local deformations. This is similar to explicit elastic deformations of the input images,[44] which delivers excellent MNIST performance. Using stochastic pooling in a multilayer model gives an exponential number of deformations since the selections in higher layers are independent of those below.

10.9 Leaky Units and Other Strategies for Multiple Time Scales

One way to deal with long-term dependencies is to design a model that operates at multiple time scales, so that some parts of the model operate at fine-grained time scales and can handle small details, while other parts operate at coarse time scales and transfer information from the distant past to the present more efficiently. Various strategies for building both fine and coarse time scales are possible. These include the addition of skip connections across time, "leaky units" that integrate signals with different time constants, and the removal of some of the connections used to model fine-grained time scales.

Method	Mean performance for 50 networks (%)		
	Recall	Precision	F_1 Measure
Randomised	92.72	92.12	92.42
Original Corpus	92.59	91.73	92.16
Reversed Original	92.26	91.39	91.83

$$precision = \frac{tp}{tp + fp} \qquad recall = \frac{tp}{tp + fn}$$

$$F_{user} = \frac{\left(1 + N^2\right) \cdot precision \cdot recall}{precision + \left(N^2 \cdot recall\right)} \qquad F_1 \text{ Measure} : F_{user} \text{ with } N = 1$$

8.3.2 Momentum

While stochastic gradient descent remains a very popular optimization strategy, learning with it can sometimes be slow. The method of momentum (Polyak, 1964) is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. The effect of momentum is illustrated in Fig. 8.5.

Formally, the momentum algorithm introduces a variable $\boldsymbol{v}$ that plays the role of velocity—it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient. The name *momentum* derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton's laws of motion. Momentum in physics is mass times velocity. In the momentum learning algorithm, we assume unit mass, so the velocity vector $\boldsymbol{v}$ may also be regarded as the momentum of the particle. A hyperparameter $\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay. The update rule is given by:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}\right) \right), \tag{8.15}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}. \tag{8.16}$$

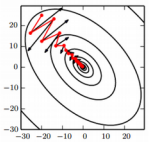


Figure 8.5: Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient. Here, we illustrate how momentum overcomes the first of these two problems. The contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix. The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point. We can see that a poorly conditioned quadratic objective looks like a long, narrow valley or canyon with steep sides. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon. Compare also Fig. 4.6, which shows the behavior of gradient descent without momentum.

multiple timescales by restricting update frequencies to temporal constraints. A different method with the same effect is the use of leakage and hysteresis parameters that constrain the amount of change within a system between time steps. The concept of leakage is most popularly used in the Echo State Network (ESN [4]) but has also been shown to improve the Simple Recurrent Network (SRN [5]). A related concept can be found in the Recurrent Plausibility Network (RPN [6]) which introduces a related hysteresis parameter φ to perform time-averaging. It also has shortcut connections, which provide shorter error propagation paths for the temporal context layers. Shortcuts have been shown to allow better training in very deep networks [7]. Both shortcuts and leaky units are used in the Structurally Constrained Recurrent Network (SCRN [8]) that additionally partitions its layer into modules, similarly to the CWRNN.

Exam 2 (Prof. Wermter):

1. Explain the difference between localist and distributed representations. When are they used? How does sparse coding compare?
2. What is the difference between MLP and deep learning? Is a recurrent plausability net a kind of deep learning? How can we combine recurrency and deep learning?
3. How does a CNN work? (Explain normalization/convolution/pooling, draw a filter kernel, explain how it is applied to an image, ...)
4. How does learning work in a CNN? What exactly is learned? (neocognitron was given as a hint)
5. What are pros and cons of using sigmoid functions vs ReLU? When are they typically used? How about x=0 in ReLU (not differentiable), is there an alternative? (softplus)
6. How do you evaluate a network? (He wanted me to bring up training/test/validation sets and scores such as precision, recall, F1)
7. How can you determine when a SOM should grow? What is the difference between "growing when required" and "growing neural gas"?

Method	Mean performance for 50 networks (%)		
	Recall	Precision	F_1 Measure
Randomised	92.72	92.12	92.42
Original Corpus	92.59	91.73	92.16
Reversed Original	92.26	91.39	91.83

$$precision = \frac{tp}{tp + fp} \qquad recall = \frac{tp}{tp + fn}$$

$$F_{user} = \frac{(1 + N^2) \cdot precision \cdot recall}{precision + (N^2 \cdot recall)} \qquad F_1 \text{ Measure} \quad : F_{user} \text{ with } N = 1$$

Given a big, yellow Volkswagen car. How many memory units will you use to store this information?

You might tempt to use a single unit to do that. Given the input, if the unit is activated (its value is 1), you know it is a big, yellow Volkswagen. If it is not, then the input is not a big yellow Volkswagen.

Now let's say you need to store a small gray Lexus, a huge green Toyota, and Optimus Prime. Do you see where that goes? Right, you might need to store another small blue BMW, in addition to, well, Bumblebee, Ironhide and all the gang. If you allocate one memory unit for each car model, then you might need lots of units to memorize all those models. The number of units you might need, in fact, can grow exponentially. This is an example of non-distributed representation (or "localist representation", as called by Geoff Hinton: http://www.cs.toronto.edu/~bonne...).

Now let us take another approach. We are going to use 3 units: 1 for the size (small, medium, big, huge, and transformers), 1 for color pattern and 1 for the brand. So now, depending on the output of each unit, we can have all kinds of combinations between size, color pattern and brand. We can store loads of car models using only 3 units. This is a distributed representation.

Consider the well-known Hidden markov model. At each time step, the hidden state of the model can take only one value. At each hidden state, we have a different observation probabilities. This, in fact, requires that the number of hidden states in HMM must be big enough to capture all the combinations of its internal states. Take any working model of HMM for speech recognition, you will see how huge the hidden states can be (and must be, to make sure it works).

Now consider a neural network with a hidden layer of sigmoid units. The values of the hidden units can be anything between 0 and 1. The value of the output unit is a non-linear function of the linear combination of all the hidden units. Therefore every combination of the hidden units is possible.

In the simplest case, a hidden layer of n binary units can represent 2^n states, while an HMM with n hidden states can only represent n states. This is where we see the difference between distributed and local representation.

Sparse coding

Sparse coding minimizes the objective

$$\mathcal{L}_{sc} = \underbrace{||WH - X||_2^2}_{\text{reconstruction term}} + \underbrace{\lambda||H||_1}_{\text{sparsity term}}$$

where W is a matrix of bases, H is a matrix of codes and X is a matrix of the data we wish to represent. λ implements a trade of between sparsity and reconstruction. Note that if we are given H, estimation of W is easy via least squares.

In the beginning, we do not have H however. Yet, many algorithms exist that can solve the objective above with respect to H. Actually, this is how we do inference: **we need to solve an optimisation problem** if we want to know the h belonging to an unseen x.

Auto encoders

Auto encoders are a family of unsupervised neural networks. There are quite a lot of them, e.g. deep auto encoders or those having different regularisation tricks attached--e.g. denoising, contractive, sparse. There even exist probabilistic ones, such as generative stochastic networks or the variational auto encoder. Their most abstract form is

$$D(d(e(x;\theta');\theta''), x)$$

but we will go along with a much simpler one for now:

$$\mathcal{L}_{ae} = ||W\sigma(W^T X) - X||^2$$

where σ is a nonlinear function such as the logistic sigmoid $\sigma(x) = \frac{1}{1+\exp(-x)}$.

Similarities

Note that $\mathcal{L}_{ae}$ looks *almost* like $\mathcal{L}_{sc}$ once we set $H = \sigma(W^T X)$. The difference of both is that i) auto encoders do not encourage sparsity in their general form ii) an autoencoder uses a model for finding the codes, while sparse coding does so by means of optimisation.

For natural image data, regularized auto encoders and sparse coding tend to yield very similar W. However, auto encoders are *much more efficient* and are easily generalized to much more complicated models. E.g. the decoder can be highly nonlinear, e.g. a deep neural network. Furthermore, one is not tied to the squared loss (on which the estimation of W for $\mathcal{L}_{sc}$ depends.)

Also, the different methods of regularisation yield representations with different characteristica. Denoising auto encoders have also been shown to be equivalent to a certain form of RBMs etc.

But why?

If you want to solve a prediction problem, you will not need auto encoders *unless* you have only little labeled data and a lot of unlabeled data. Then you will generally be better of to train a deep auto encoder and put a linear SVM on top instead of training a deep neural net.

However, they are very powerful models for capturing characteristica of distributions. This is vague,

Recurrent Plausibility Networks

Recurrent neural networks introduce previous states and extend feedforward networks with short-term incremental memories. In fully recurrent networks, all the information is processed and fed back into one single layer. Partially recurrent networks, such as simple recurrent networks, have recurrent connections between the hidden layer and context layer (Elman 1990) or Jordan networks have connections between the output and context layer (Jordan 1986).

In other research (Wermter 1995), different decay memories were introduced by using distributed recurrent delays over the separate context layers representing the contexts at different time steps. At a given time step, the network with n hidden layers processes the current input as well as the incremental contexts from the $n-1$ previous time steps.

Figure 1 shows the general structure of our recurrent plausibility network. It combines the features of recurrent networks with distributed context layers and self-recurrent connections of the context layers. The input to a hidden layer L_n is constrained by the underlying layer L_{n-1} as well as the incremental context layer C_n. The activation of a unit $L_{ni}(t)$ at time t is computed on the basis of the weighted activation of the units in the previous layer $L_{(n-1)i}(t)$ and the units in the current context of this layer $C_{ni}(t)$ limited by the logistic function f.

$$L_{ni}(t) = f(\sum_k w_{ki} L_{(n-1)i}(t) + \sum_l w_{li} C_{ni}(t))$$

The units in the context layers perform a time-averaging of the information using the equation

$$C_{ni}(t) = (1 - \varphi_n) L_{ni}(t-1) + \varphi_n C_{ni}(t-1)$$

where $C_{ni}(t)$ is the activation of a unit in the context layer at time t. We represent the self-recurrent connections using the hysteresis value φ_n. The hysteresis value of the context layer C_{n-1} is lower than the hysteresis value of the next context layer C_n. This ensures that the context layers closer to the input layer will perform as memory that represents a more dynamic context. Having higher hysteresis values, the context layers closer to the output layer will incrementally build more stable sequential memory. Therefore the larger context is built on the more recent, dynamic one.

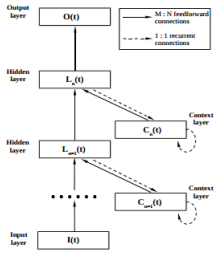


Figure 1: Recurrent plausibility network.

The reason that sigmoid functions are being replaced by rectified linear units, is because of the properties of their derivatives.

Let's take a quick look at the sigmoid function σ which is defined as $\frac{1}{1+e^{-x}}$. The derivative of the sigmoid function is

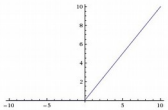
$$\sigma'(x) = \sigma(x) * (1 - \sigma(x))$$

The range of the σ function is between 0 and 1. The maximum of the σ' derivative function is equal to $\frac{1}{4}$. Therefore when we have multiple stacked sigmoid layers, by the backprop derivative rules we get multiple multiplications of σ'. And as we stack more and more layers the maximum gradient decreases exponentially. This is commonly known as the vanishing gradient problem. The opposite problem is when the gradient is greater than 1, in which case the gradients explode toward infinity (exploding gradient problem).

Now let's check out the ReLU activation function which is defined as:

$$R(x) = max(0, x)$$

The graph of which looks like



If you look at the derivatives of the function (slopes on the graph), the gradient is either 1 or 0. In this case we do not have the vanishing gradient problem or the exploding problem. And since the general trend in neural networks has been deeper and deeper architectures ReLU became the choice of activation.

Hope this helps.

There are two main benefits:

- ReLUs are much simpler computationally. The forward and backward passes through an ReLU are both just a simple if statement. Compare this to the sigmoid activation, which requires computing an exponent. This advantage is huge when dealing with big networks with many neurons, and can significantly reduce both training and evaluation times.
- Sigmoid activations are easier to saturate. There is a comparatively narrow interval of inputs for which the sigmoid's derivative is *sufficiently nonzero*. In other words, once a sigmoid reaches either the left or right plateau, it is almost meaningless to make a backward pass through it, since the derivative is very close to 0. On the other hand, ReLUs only saturate when the input is less than 0. And even this saturation can be eliminated by using leaky ReLUs. For very deep networks, saturation hampers learning, and so ReLUs provide a nice workaround.

Self-Organizing Map (SOM)

The Self-Organizing Map is one of the most popular neural network models. It belongs to the category of competitive learning networks. The Self-Organizing Map is based on unsupervised learning, which means that no human intervention is needed during the learning and that little needs to be known about the characteristics of the input data. We could, for example, use the SOM for clustering data without knowing the class memberships of the input data. The SOM can be used to detect features inherent to the problem and thus has also been called SOFM, the Self-Organizing Feature Map.

The Self-Organizing Map was developed by professor Kohonen [20]. The SOM has been proven useful in many applications [22]. For closer review of the applications published in the open literature, see section 2.3.

The SOM algorithm is based on unsupervised, competitive learning. It provides a topology preserving mapping from the high dimensional space to map units. Map units, or neurons, usually form a two-dimensional lattice and thus the mapping is a mapping from high dimensional space onto a plane. The property of topology preserving means that the mapping preserves the relative distance between the points. Points that are near each other in the input space are mapped to nearby map units in the SOM. The SOM can thus serve as a cluster analyzing tool of high-dimensional data. Also, the SOM has the capability to generalize. Generalization capability means that the network can recognize or characterize inputs it has never encountered before. A new input is assimilated with the map unit it is mapped to.

The Self-Organizing Map is a two-dimensional array of neurons:

$$\mathbf{M} = \{\mathbf{m}_1, .., \mathbf{m}_{900}\}$$

This is illustrated in Figure 2.3. One neuron is a vector called the codebook vector

$$\mathbf{m}_i = [m_{i1}, .., m_{in}]$$

This has the same dimension as the input vectors (n -dimensional). The neurons are connected to adjacent neurons by a neighborhood relation. This dictates the topology, or the structure, of the map. Usually, the neurons are connected to each other via rectangular or hexagonal topology. In the Figure 2.3 the topological relations are shown by lines between the neurons.

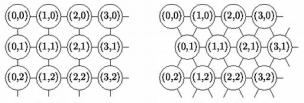


Figure 2.3: Different topologies

Variants [edit]

A number of variants of the neural gas algorithm exists in the literature so as to mitigate some of its shortcomings. More notable is perhaps Bernd Fritzke's growing neural gas,[6] but also one should mention further elaborations such as the Growing When Required network[7] and also the incremental growing neural gas.[8]

Growing neural gas [edit]

Fritzke describes the growing neural gas (GNG) as an incremental network model that learns topological relations by using a "Hebb-like learning rule",[6] only, unlike the neural gas, it has no parameters that change over time and it is capable of continuous learning.

Growing when required [edit]

Having a network with a growing set of nodes, like the one implemented by the GNG algorithm was seen as a great advantage, however some limitation on the learning was seen by the introduction of the parameter λ, in which the network would only be able to grow when iterations were a multiple of this parameter.[7] The proposal to mitigate this problem was a new algorithm, the Growing When Required network (GWR), which would have the network grow more quickly, by adding nodes as quickly as possible whenever the network identified that the existing nodes would not describe the input well enough.

Incremental growing neural gas [edit]

Another neural gas variant inspired in the GNG algorithm is the incremental growing neural gas (IGNG). The authors propose the main advantage of this algorithm to be "learning new data (plasticity) without degrading the previously trained network and forgetting the old input data (stability)."[8]

Example: Allergy or Not?

Hunter says she is itchy. There is a test for Allergy to Cats, but this test is not always right:

- For people that **really do** have the allergy, the test says "Yes" **80%** of the time
- For people that **do not** have the allergy, the test says "Yes" **10%** of the time ("false positive")

Here it is in a table:

	Test says "Yes"	Test says "No"
Have allergy	**80%**	**20% "False Negative"**
Don't have it	**10% "False Positive"**	**90%**

Question: If 1% of the population have the allergy, and **Hunter's test says "Yes"**, what are the chances that Hunter really has the allergy?

Do you think 75%? Or maybe 50%?

A similar test was given to Doctors and most guessed around 75% ...
... but they were very wrong!

(Source: "Probabilistic reasoning in clinical medicine: Problems and opportunities" by David M. Eddy 1982, which this example is based on)

There are three good ways to solve this: "Imagine a 1000", "Tree Diagrams" or "Bayes' Theorem", use which you like.

Try Imagining A Thousand People

When trying to understand questions like this, just imagine a large group (say 1000) and play with the numbers:

- Of 1000 people, only **10** really have the allergy (1% of 1000 is 10)
- The test is 80% right for people who **have** the allergy, so it will get **8 of those 10 right**.
- But 990 **do not** have the allergy, and the test will say "Yes" to 10% of them, which is **99 people** it says "Yes" to **wrongly** (false positive)
- So out of 1000 people the test says "**Yes**" to (8+99) = **107 people**

As a table:

	1% have it	Test says "Yes"	Test says "No"
Have allergy	10	**8**	**2**
Don't have it	990	**99**	**891**
	1000	107	893

So 107 people get a "Yes" but only 8 of those really have the allergy:

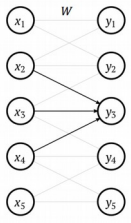
8 / 107 = about 7%

So, even though Hunter's test said "Yes", it is still only **7% likely** that Hunter has a Cat Allergy.

Why so small? Well, the allergy is so rare that those who actually have it are greatly **outnumbered** by those with a false positive.

Convolutional Layers

- **x** = input signal
- **y** = filter map
- **W** = kernel (filter)
- Sliding the filters over the input signal
- y_j is a dot product of the local neighborhood and the filter weights
- Apply an activation function on each filter map



- Different filters
  - Extract different structures
- All the filters are applied to the same input signal
- Each layer will produce N outputs where N is the number of filters

- Deeper layers will receive N inputs from previous layers
- Each filter sums over previous layer output
- N filter maps will produce one output

Exam 3 (Prof. Wermter):

1. Explain how Echo State Networks help in sequencial tasks? Why do we have a reservoir and how does it help? Compare ESN with SRN. What's different? What does leak rate do in an ESN and why do we need that?
2. Compare Plausibility Networks with ESN? What is the Basic principle about RPNs and what is the effect of different contexts and Hysteresis?
3. What is the effect of multiple timescales in an MTRNN? What is the need for that? What do slow and fast layers do?
4. Compare Clock-work RNN with MTRNN? How does this work? (Drew the network diagram and explained working). Effect of clock speeds? This is discrete, can we do it in a continuous way? I said something about Spike Coding Neurons and Neural Oscillators tuning into specific frequency to look at a particular part of the input. (Visual Attention Model one).
5. Is there a concept of time in the brain? (Yes I presume, with sequencing and neural oscillations (then the neural oscillation visual attention model again)). Do you know any networks which work on this such as alpha, beta, gamma networks?. (Acknowledged that it was not a part of the curriculum, so left it.)
6. ReLU - Why? How does it help? What are the limitations? What is Softplus? (Wrote the function formula and drew a graph for ReLU and Softplus.)

2 Recurrent Neural Networks

2.1 Recurrent Plausibility Network

The Recurrent Plausibility Network (RPN) was originally developed to learn and represent semantic relationships while disambiguating contextual relationships [9]. It is based on the state of an unfolded SRN during truncated BPTT (see Fig. 1(a)), i.e. each hidden layer $\mathbf{h}$ has its own set of m context layers $\mathbf{c}_k$ ($k \in \{1, \ldots, m\}$) which store past activations. The main difference to an unfolded SRN is the use of temporal shortcut connections for shorter context propagation paths, making vanishing or exploding gradients less likely (compare Fig. 1(b)). For time step t, the units of the hidden layer $\mathbf{h}$ are activated as follows:

$$\mathbf{h}^{(t)} = f_h\left(\mathbf{x}^{(t)}\ \mathbf{W}_{xh} + \sum_{k=1}^{m} \mathbf{c}_m^{(t-1)}\ \mathbf{W}_{mh}\right), \tag{1}$$

where the vector $\mathbf{c}$ denotes the context layers, that are activated by shifting their contents with $\mathbf{c}_{m-1}^{(t)} = \mathbf{c}_m^{(t-1)}$. The respective context activation for units in $\mathbf{c}_m$ is further constrained under the hysteresis parameter φ [10]:

$$\mathbf{c}_k^{(t)} = \begin{cases} (1-\varphi_n)\cdot \mathbf{h}^{(t-1)} + \varphi \cdot \mathbf{c}_k^{(t-1)} & \text{iff } k = 1, \\ \mathbf{c}_{k-1}^{(t-1)} & \text{otherwise} \end{cases} \tag{2}$$

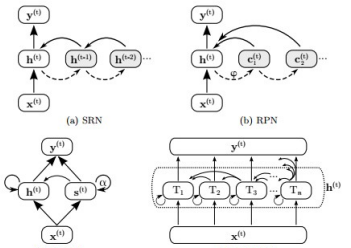


Fig. 1. Comparison of investigated RNN architectures. Figure (a) shows an SRN unfolded in time. The RPN (b) extends the SRN with its temporal shortcuts and the hysteresis φ. In case of a deep RPN, each vertical layer $\mathbf{h}_n^{(t)}$ can have its own hysteresis value φ_n. The SCRN (c) has an additional layer $\mathbf{s}^{(t)}$ that learns slower than in $\mathbf{h}^{(t)}$ due to its high leakage $\alpha = 0.95$. The modules T_k of the CWRNN (d) are sorted by increasing numbers from left to right and are only updated for $t \bmod T_i = 0$.

The hysteresis mechanism allows for a finer adjustment of context memory than in the SRN. Rather than accumulating past activations in a single feedback loop, the network is able to specifically learn the contribution between specific time frames due to the temporal shortcuts.

2.2 Structurally Constrained Recurrent Network

The Structurally Constrained Recurrent Network (SCRN) was recently proposed by Mikolov et al. [8]. The motivation behind the architecture is to achieve specialization of hidden layers by partitioning them into parallel "modules" that operate independently and under distinct temporal constraints. This theoretically allows to train on multiple timescales. While the left path in the SCRN equals a SRN with a regular hidden layer $\mathbf{h}^{(t)}$, the additional module $\mathbf{s}^{(t)}$ has units with different temporal characteristics (compare Fig. 1(c)). It is initialized with the recurrent identity matrix and its updates constrained by a leakage parameter $\alpha \in [0, 1]$. The authors set this leakage to 0.95, causing the states to change on a much slower scale than in $\mathbf{h}^{(t)}$. Similarly to the RPN, this architecture makes use of shortcut connections ($\mathbf{W}_{sh}$) that allow $\mathbf{h}^{(t)}$ to access long-term context which is learned in $\mathbf{s}^{(t)}$. The update rules of the SCRN are as follows:

$$\mathbf{s}^{(t)} = (1-\alpha)\mathbf{W}_{xs}\ \mathbf{x}^{(t)} + \alpha\ \mathbf{s}^{(t-1)}, \tag{3}$$

$$\mathbf{h}^{(t)} = f_h(\mathbf{W}_{sh}\ \mathbf{s}^{(t)} + \mathbf{W}_{xh}\ \mathbf{x}^{(t)} + \mathbf{W}_{hh}\ \mathbf{h}^{(t-1)}), \tag{4}$$

$$\mathbf{y}^{(t)} = f_y(\mathbf{W}_{hy}\ \mathbf{h}^{(t)} + \mathbf{W}_{sy}\ \mathbf{s}^{(t)}), \tag{5}$$

where f_h and f_y are the respective activation functions for the hidden and output layers.

2.3 Clockwork Recurrent Neural Network

The discussed idea of partitioning the hidden layer into parallel modules with distinct temporal properties can also be found in the Clockwork Recurrent Neural Network (CWRNN). However, the main difference is that multiple timescales are not achieved by varying leakage but rather an external clock that determines *when* a module gets updated. This means that a module k is only updated if its clock period T_k satisfies the criterion $t \bmod T_k = 0$. Otherwise, the module is inactive in which case the previous activation $\mathbf{h}_k^{(t-1)}$ gets copied over:

$$\mathbf{h}_k^{(t)} = \begin{cases} f_h\left(\mathbf{x}^{(t)}\ \mathbf{W}_{xk} + \sum_{l=k}^{n} \mathbf{h}_l^{(t-1)}\ \mathbf{W}_{lk}\right) & \text{iff } t \bmod T_k = 0, \\ \mathbf{h}_k^{(t-1)} & \text{otherwise} \end{cases} \tag{6}$$

An additional constraint is that $T_l > T_k$ for $l < k$, i.e. the modules are ordered by increasing numbers from left to right (compare Fig. 1(d)). Therefore, modules on the left are updated more frequently than those on the right. Consequently, modules with greater periods (on the right) will self-organize slower and to long-term dependencies while those with small periods (on the left) change more often, focusing on short-term dependencies.

In this paper, we have explored various design concepts that allow emergence of multiple timescales and long-term memory in RNNs. Leaky and clocked activations have been investigated together with partitioning hidden layers into modules and using shortcut connections by comparing a number of architectures on the tasks of sequence generation and learning embedded Reber grammar.

Our results show that parallel hidden layers, which learn under different temporal constraints can lead to an emergence of multiple timescales in RNNs. Furthermore, shared weights in the form of shortcut connections (such as in the SCRN and CWRNN) allow units which self-organize to short-term context, to take long-term dependencies into account from specialized units that operating on a larger timescale. While the SCRN achieves this by means of leakage, the CWRNN utilizes clocked module activations. For the sequence generation task, the CWRNN was the only architecture to learn the decomposition of the trained sinusoid wave into all its subfrequencies. All other networks converged to the mean or a single subfrequency. This suggests that the CWRNN is able to store the entire sequence in the memory of the clocked modules, although it has half as much parameters as the SRN [3]. For the second task, the complete opposite can be observed; the SCRN is able to outperform all other networks for sequence lengths of 20 while the CWRNN has difficulties.

Our findings suggest that the SCRN and RPN seem to work better for discrete, symbolic long-term decisions while the CWRNN is better at decomposing real-valued signals. Partitioning hidden layers with distinct temporal constraints has shown to be a viable method to capture different timescales. Future research should therefore concentrate on further exploring time scaling mechanisms on more challenging tasks such as sequence classification or language modeling.

Curator: Herbert Jaeger

Echo state networks (ESN) provide an architecture and supervised learning principle for recurrent neural networks (RNNs). The main idea is (i) to drive a random, large, fixed recurrent neural network with the input signal, thereby inducing in each neuron within this "reservoir" network a nonlinear response signal, and (ii) combine a desired output signal by a trainable linear combination of all of these response signals.

The basic idea of ESNs is shared with *Liquid State Machines* (LSM), which were developed independently from and simultaneously with ESNs by Wolfgang Maass (Maass W., Natschlaeger T., Markram H. 2002). Increasingly often, LSMs, ESNs and the more recently explored *Backpropagation Decorrelation* learning rule for RNNs (Schiller and Steil 2005) are subsumed under the name of *Reservoir Computing*. Schiller and Steil (2005) also showed that in traditional training methods for RNNs, where all weights (not only the output weights) are adapted, the dominant changes are in the output weights. In cognitive neuroscience, a related mechanism has been investigated by Peter F. Dominey in the context of modelling sequence processing in mammalian brains, especially speech recognition in humans (e.g., Dominey 1995, Dominey, Hoen and Inui 2006). Dominey was the first to explicitly state the principle of reading out target information from a randomly connected RNN. The basic idea also informed a model of temporal input discrimination in biological neural networks (Buonomano and Merzenich 1995). The earliest known clear formulation of the reservoir computing idea, however, is due to K. Kirby who exposed this concept in a largely forgotten (1 Google cite, as of 2017) conference contribution (Kirby 1991).

For an illustration, consider the task of training an RNN to behave as a tunable frequency generator (download the MATLAB code of this example). The input signal $u(n)$ is a slowly varying frequency setting, the desired output $y(n)$ is a sinewave of a frequency indicated by the current input. Assume that a training input-output sequence $D = (u(1), y(1)), \ldots, (u(n_{max}), y(n_{max}))$ is given (see the input and output signals in ; here the input is a slow random step function indicating frequencies ranging from 1/16 to 1/4 Hz). The task is to train a RNN from these training data such that on slow test input signals, the output is again a sinewave of the input-determined frequency.

In the ESN approach, this task is solved by the following steps.

- **Step 1: Provide a random RNN.** (i) Create a random *dynamical reservoir* RNN, using any neuron model (in the frequency generator demo example, non-spiking leaky integrator neurons were used). The reservoir size N is task-dependent. In the frequency generator demo task, $N = 200$ was used. (ii) Attach input units to the reservoir by creating random all-to-all connections. (iii) Create output units. If the task requires output feedback (the frequency-generator task does), install randomly generated output-to-reservoir connections (all-to-all). If the task does not require output feedback, do not create any connections to/from the output units in this step.
- **Step 2: Harvest reservoir states.** Drive the dynamical reservoir with the training data D for times $n = 1, \ldots, n_{max}$. In the demo example, where there are output-to-reservoir feedback connections, this means to write both the input $u(n)$ into the input unit and the teacher output $y(n)$ into the output unit ("teacher forcing"). In tasks without output feedback, the reservoir is driven by the input $u(n)$ only. This results in a sequence $\mathbf{x}(n)$ of N-dimensional reservoir states. Each component signal $x(n)$ is a nonlinear transform of the driving input. In the demo, each $x(n)$ is an individual mixture of both the slow step input signal and the fast output sinewave (see the five exemplary neuron state plots in Figure 1).
- **Step 3: Compute output weights.** Compute the output weights as the linear regression weights of the teacher outputs $y(n)$ on the reservoir states $\mathbf{x}(n)$. Use these weights to create reservoir-to-output connections (dotted arrows in Figure 1). The training is now completed and the ESN ready for use. Figure 2 shows the output signal obtained when the trained ESN was driven with the slow step input shown in the same figure.

Echo state property. In order for the ESN principle to work, the reservoir must have the *echo state property* (ESP), which relates asymptotic properties of the excited reservoir dynamics to the driving signal. Intuitively, the ESP states that the reservoir will asymptotically wash out any information from initial conditions. The ESP is guaranteed for additive-sigmoid neuron reservoirs, if the reservoir weight matrix (and the leaking rates) satisfy certain algebraic conditions in terms of singular values. For such reservoirs with a tanh sigmoid, the ESP is violated *for zero input* if the spectral radius of the reservoir weight matrix is larger than unity. Conversely, it is empirically observed that the ESP is granted for any input if this spectral radius is smaller than unity. This has led in the literature to a far-spread but erroneous identification of the ESP with a spectral radius below 1. Specifically, the larger the input amplitude, the further above unity the spectral radius may be while still obtaining the ESP. An abstract characterization of the ESP for arbitrary reservoir types, and algebraic conditions for additive-sigmoid neuron reservoirs are given in Jaeger (2001a); for an important subclass of reservoirs, tighter algebraic conditions are given in Buehner and Young (2006) and Yildiz et al. (2012); for leaky integrator neurons, algebraic conditions are spelled out in Jaeger et al. (2007). The relationship between input signal characteristics and the ESP are explored in Manjunath and Jaeger (2012), where a fundamental 0-1-law is shown: if the input comes from a stationary source, the ESP holds with probability 1 or 0.

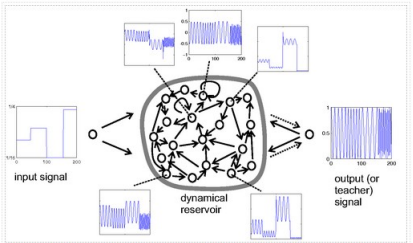


Figure 1: The basic schema of an ESN, illustrated with a tuneable frequency generator task. Solid arrows indicate fixed, random connections; dotted arrows trainable connections.

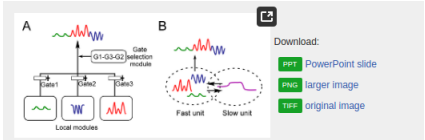


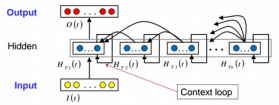
Figure 1. Schematic drawings of (A) local representation model and (B) multiple timescale model.
(A) Curves colored red, blue, and green represent sensori-motor sequences corresponding to motor primitives. Output of the system consists of behavior sequences made up of combinations of these primitives. In the local representation model, functional hierarchy is realized through the use of explicit hierarchical structure, with local modules representing motor primitives in the lower level, and a higher module representing the order of motor primitives switched via additional mechanisms such as gate-selection. (B) In the multiple timescale model, primitives are represented by fast context units whose activity changes quickly, whereas sequences of primitives are represented by slow context units whose activity changes slowly.

In order to overcome difficulties associated with the local representation model, we introduce in the current study a different type of representation for functional hierarchy. The representation we use neither makes use of separate local modules to represent primitives, nor introduces explicit hierarchical structure to manipulate these primitives. Instead of setting up an explicit hierarchy, we attempt to realize the self-organization of a functional hierarchy by means of neural activity with multiple timescales. This functional hierarchy is made possible through the use of two distinct types of neurons, each with different temporal properties. The first type of neuron is the "fast" unit, whose activity changes quickly over the short term. The second type of neuron is the "slow" unit, whose activity changes over the long term (Figure 1B).

The idea that multiple timescales may carry advantages for neural systems in interacting with complex environments is intuitively understandable. Indeed, the importance of multiple timescales in neural systems has been emphasized in a number of earlier studies from various different fields. For example, at the level of behavior, it has been shown that the process of acquiring motor skills develops through multiple timescales [19],[20]. Biological observations on motor adaptation, such as for example saccade adaptation and force field adaptation, likewise suggest that these processes involve distinct subsystems with differing timescales [21],[22]. At the level of neural synchrony, meanwhile, it is thought that differing timescales in neural synchrony are involved at different levels of information processing, such as for example in local and global interactions of brain regions [23],[24]. These previous studies strongly suggest the possibility that multiple timescales may be essential for the emergence of functional hierarchy in neural systems.

At the neuron level, the use of timescale variation has also been proposed as a means of representing different levels of functionality. In a study of auditory perception, for example, Poeppel [25] hypothesized that different temporal integration windows in neural activities correspond to a perceptual hierarchy between formant transition level and syllable level. In a study of an evolutional neural network model using a mobile robot, Nolfi [26] showed that a model with differing temporal integration windows is superior to the normal model in cases in which the robot is required to achieve two different tasks: collision avoidance, which requires short-term sensori-motor control, and self-localization, which requires long-term sensory integration. Furthermore, Paine [27] showed that, using a similar evolutional neural network model with a mobile robot, it was possible to achieve hierarchical functionality of motor primitives (wall avoidance) and execution of a given sequence of primitives (global goals) through a particular constraint

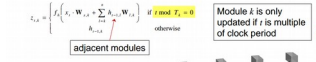
Clockwork Recurrent Neural Network



- Assign a clock period T_k to each module k
- For each time step t:
  - Apply activation function if module is active
  - Otherwise, keep module activations from previous time step

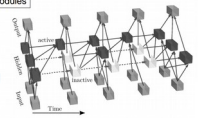
Clockwork RNN activation function

- Activation value of the neuron in kth module:



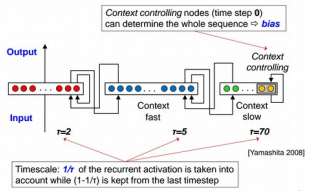
Clockwork RNN unfolded in time

- Training:
  - Error propagated according to "activeness"



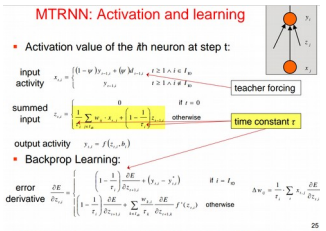
[Koutnik et al. 2014]

Multiple Timescale Recurrent Neural Network



MTRNN: Activation and learning

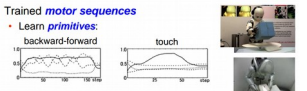
- Activation value of the kth neuron at step t:



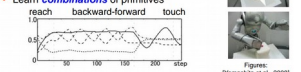
25

MTRNN experiment A (Yamashita 2008)

- Trained ***motor sequences***
  - Learn ***primitives***:



  - Learn ***combinations*** of primitives



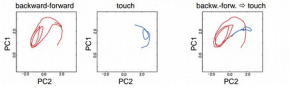
…and test for the ***emergence*** of a hierarchy

26

MTRNN experiment A: Analysis

- Question: How does the network ***self-organize***?
- Approach: Run a ***Principle Component Analysis*** on the neural activity

- Result: Emergence of a ***functional hierarchy***

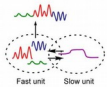


MTRNN experiment B (Hinoshita 2011)

- Trained ***sentences*** and tested for emulation and recognition

Interesting observation:

- Linguistic ***hierarchy*** emerges in the network:
  - Word representations in the Cf
  - Sentence representations in the Cs
- Linguistic structure can produce sentences from the ***inferred*** grammar.
  - Even if they where not learned explicitly!



Exam 4 (Dr. Weber):

1. Started with SOM: What is SOM, how does it work?
2. What is the Mexican Hat inspired from? Didn't know the exact answer, said it's similar to the center-surround receptive fields of cells in V1.
3. How can we visualize SOMs from high-dimensional data? He gave me a hint: the U matrix (went on and explained U matrix)
4. Can a SOM's topology change? Which models do you know that enable this? (Explained growing when required)
5. Is there something else but the winner-takes-all coding?
6. What's the binding problem?
7. What is reinforcement learning?
8. Which of these are similar (after I explained there's Q-Learning, SARSA and Actor Critic) and what's the difference between them?
9. How can we implement this using a neural net?
10. What if we have distributed input? Said we could use classification before that.
11. Can Reinforcement learning overfit? How do we detect overfitting in general?

Winner-take-all is a computational principle applied in computational models of neural networks by which neurons in a layer compete with each other for activation. In the classical form, only the neuron with the highest activation stays active while all other neurons shut down; however, other variations allow more than one neuron to be active, for example the soft winner take-all, by which a power function is applied to the neurons.

Neural Networks and RL: You have two options. The first one is to use a network which you will have as input your feature vectors (states) and output probability of each action. This is called a policy network and you can find a very detailed tutorial with Python Code in order to implement it by A. Karpathy. Your second option is to use the Q-Network approach. Your input will be again the same but the output will be values of your Q function for each action you have ($Q(a_i)$). You will use the Q-learning equation $Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \max_a' Q(s_{t+1}, a') - Q(s_t, a_t)]$. The details of the implementation can be found in the paper of V. Mnih. Also do not worry about the delayed rewards as the discount factor γ will "help" your agent to be affected by the future rewards.

In order to calculate your states, I would suggest you to create a simulation of the environment and a step function. You don't mention what kind of game you are dealing with but the general idea is that the step function will take as input your current state and current action and output the next state and reward (don't mind about the continuous space of your features as you can discretize it - you can use Kalman filters or other models to have a better state estimation as well).

My advice would be to choose your approach (Policy net or Q-net) and read the blog or the paper, create a simulation of the environment and a step function for your game. You can find tons of implementations of the Deep Q-net although I would suggest you to start with a very simple network so you don't get in trouble by tuning the Deep net.

I am about to pull my hair out trying to figure out, how exactly, a U-matrix is constructed for visualization of Self-Organizing-Maps. (SOMS, aka Kohonen Nets).

Every last google result I have found does not help, is contradictory, has a massive number of typos, or otherwise very broad.

I am asking a simple question: I have an output grid of 3x3 output units: **How do I construct a U-matrix from this?**

Links so far, (including here on stack exchange):

1) Original Paper. (RIDDLED with errors, typos, and misleading information. The U-matrix part is so full of errors I do not know how this paper got published.)

2) The SOM toolbox manual that quotes the above paper. (Explains how to do it for an output *line*, but does not explain how to do it for an output *grid*).

3) Another paper. (Explains how to make a U-matrix, but completely contradicts his first paper, and the SOM toolbox that it is based on).

4) A similar question on SE that didnt really get anywhere.

5) Another similar question on SE that is good, but doesnt explain how-exactly to make a U-matrix.

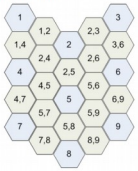
To facilitate this ... I have made up a very simple example:

I have a 3x3 output grid, that means, 3x3 output neurons that have already been trained. All neurons have dimension, say, 4. Now I want to make a U-matrix.

How exactly do I do that?

A U-matrix is a visual representation of the distances between neurons in the input data dimension space. Namely you calculate the distance between adjacent neurons, using their trained vector. If your input dimension was 4 then each neuron in the trained map corresponds to a 4-dimensional vector. Lets say you have a 3x3 hexagonal map.

The umatrix will be a 5x5 matrix with interpolated elements for each connection between two neurons like this



The {x,y} elements are the distance between neuron x and y and the values in {x} elements are the mean of the surrounding values. eg {4,5} = distance(4,5) and {4} = mean({1,4},{2,4},{4,5},{4,7}). For the calculation of the distance you use the trained 4-dimensional vector of each neuron and the distance formula that you used for the training of the map (usually Euclidian distance). So the values of the umat are only numbers (no vectors). Then you can assign a light gray colour to the largest of these values and a dark gray to the smallest and the other values to corresponding shades of gray. You can use these colours to paint the cells of the umat and have a visualized representation of the distances between neurons.

The **binding problem** is a term used at the interface between neuroscience, cognitive science and philosophy of mind that has multiple meanings.

Firstly, there is the **segregation problem**: a practical computational problem of how brains segregate elements in complex patterns of sensory input so that they are allocated to discrete "objects". In other words, when looking at a blue square and a yellow circle, what neural mechanisms ensure that the square is perceived as blue and the circle as yellow, and not vice versa? The segregation problem is sometimes called BP1.

Secondly, there is the **combination problem**: the problem of how objects, background and abstract or emotional features are combined into a single experience.[1] The combination problem is sometimes called BP2.

However, the difference between these two problems is not always clear. Moreover, the historical literature is often ambiguous as to whether it is addressing the segregation or the combination problem.[1][2]

The segregation problem [edit]

Definition [edit]

The segregation problem is the problem of how brains segregate elements in complex patterns of sensory input so that they are allocated to discrete "objects".[2]

Smythies defined BP1 in these terms: "How is the representation of information built up in the neural networks that there is one single object 'out there' and not a mere collection of separate shapes, colours and movements?"[2] Revonsuo refers to this as the problem of "stimulus-related binding" – of sorting stimuli. Although usually referred to as a problem of binding, the computational problem is arguably one of discrimination.[1] Thus, in the words of Canales et al.: "to bind together all the features of one object and segregate them from features of other objects and the background".[3] Bartels and Zeki describe it as "determining that it is the same (or a different) stimulus which is activating different cells in a given visual area or in different visual areas".[4]

The combination problem [edit]

Definition [edit]

Smythies[2] defines BP2 as "How do the brain mechanisms actually construct the phenomenal object?". Revonsuo[1] equates this to "consciousness-related binding", emphasizing the entailment of a phenomenal aspect. As Revonsuo explores in 2006,[22] there are nuances of difference beyond the basic BP1:BP2 division. Smythies speaks of constructing a phenomenal object ("local unity" for Revonsuo) but philosophers such as Descartes, Leibniz, Kant and James (see Brook and Raymont[23]) have typically been concerned with the broader unity of a phenomenal experience ("global unity" for Revonsuo) – which, as Bayne[24] illustrates may involve features as diverse as seeing a book, hearing a tune and feeling an emotion. Further discussion will focus on this more general problem of how sensory data that may have been segregated into, for instance, "blue square" and "yellow circle" are to be re-combined into a single phenomenal experience of a blue square next to a yellow circle, plus all other features of their context. There are a wide range of views on just how real this "unity" is, but the existence of medical conditions in which it appears to be subjectively impaired, or at least restricted, suggests that it is not entirely illusory.[*citation needed*]

The Binding Problem

Objects have different features such as color, shape, sound, and smell. Some, such as color and sound, are represented separately from the instant they hit our sensory receptors. Other features, such as color and shape, are initially encoded together but subsequently analyzed by separate areas of the brain. Despite this separation, in perception the brain must represent which features belong to the same object. This is the binding problem. Any case of the brain representing as associated two features or stimuli that are initially represented separately can be called binding, but this entry will focus on a subset of these: the pairing of features that belong to a common object.

Solutions to the spatial binding problem

A simple solution to the binding problem is to have a single neuron (or other representational unit) for each possible combination of features. However, considering that different feature dimensions such as color, shape, and texture may each have hundreds of values, it is impractical to dedicate a unit to each combination. Still, the visual system does contain neurons selective for certain combinations of features, and these may suffice to solve the binding problem in certain cases (Risenhuber & Poggio 1999).

Wolf Singer has championed the theory that binding is represented via synchronous rhythmic firing of the neurons selective for the paired features (von der Malsburg 1981; Gray et al. 1989). The idea is that the joint activity of the feature representations allows other brain areas to process the features together, to the exclusion of features belonging to other objects. Groups of neurons in many parts of the brain frequently do synchronize their responses, and attention to visual stimuli can enhance the effect, but the precise relationship of the phenomenon to perceptual binding remains unclear (Fries et al. 2001; Thiele & Stoner 2003; Dong et al. 2008).

Vanishing gradient between sigmoid and ReLu:
https://cs224d.stanford.edu/notebooks/vanishing_grad_example.html
http://machinelearningmastery.com/classification-accuracy-is-not-enough-more-performance-measures-you-can-use/
https://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications