Neural Networks

A comprehensive overview

Table of contents

Introduction

What is a Neural Network?

Neural Networks are essentially mathematical models inspired by connected neurons in brains.

The neural network itself is not an algorithm, but rather a model that uses learning algorithms to learn to represent a function.

Neural Networks are universal function approximators, that is, they can represent any function.(*)

History

Threshold Logic Unit

The Threshold Logic Unit (McCulloch and Pitts, 1943) was the first mathematical model for a neuron. Assuming Boolean inputs and outputs, it is defined as:

$$f(x) = 1_{\sum w_i x_i + b > 0}$$

This could approximate AND, OR and NOT boolean operations, and could thus be used to construct complex boolean functions with these units.

Note

The model takes binary inputs

Perceptron

The perceptron (Rosenblatt, 1957) is very similar, except that the inputs are real:

$$f(x) = \begin{cases} 1 & \text{if } \sum w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

This model was originally motivated by biology, with w_i being synaptic weights and x_i and f firing rates.

Perceptron

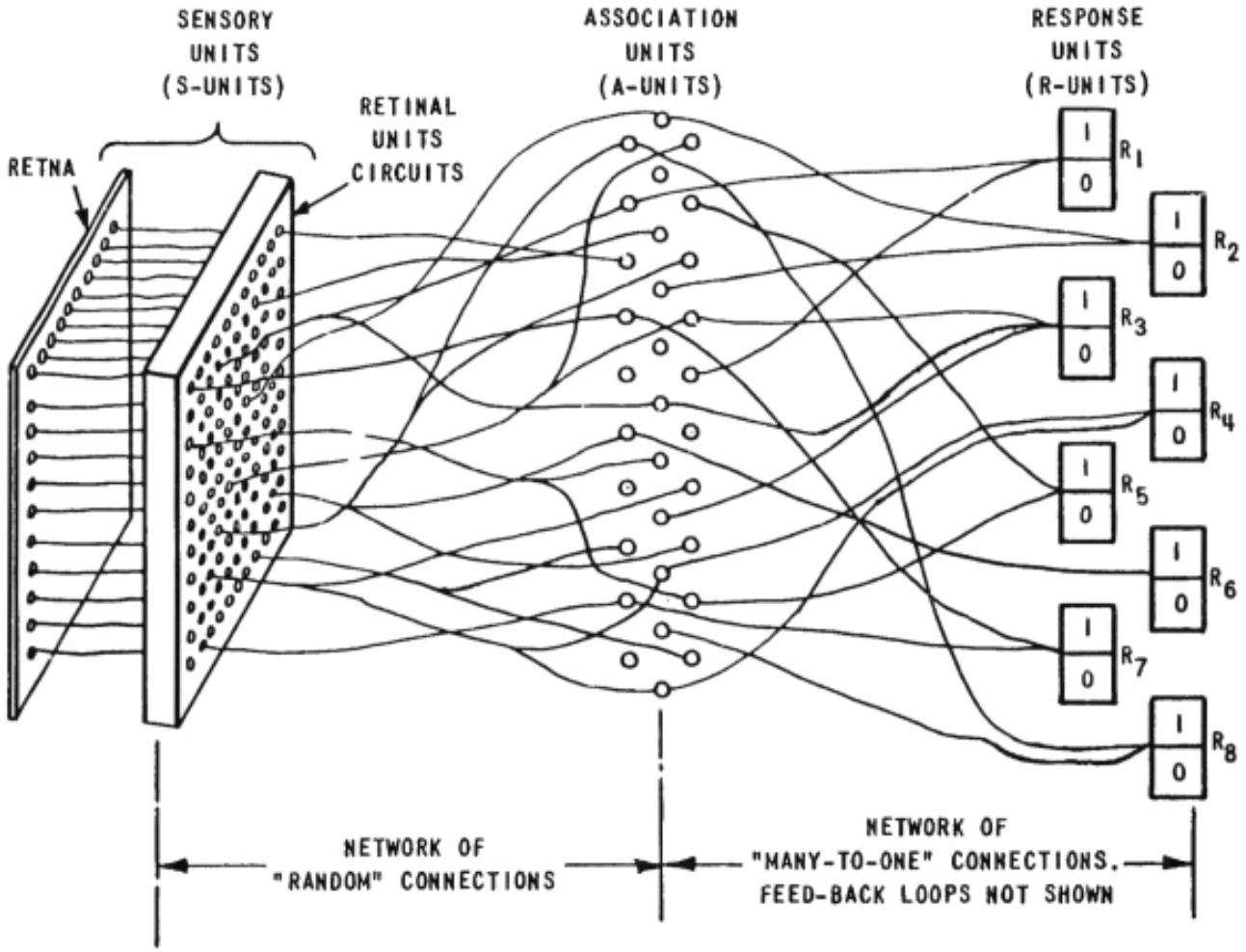


Figure 1 ORGANIZATION OF THE MARK 1 PERCEPTRON

Backpropagation

The initial idea of backpropagating errors to learn representations in Neural Networks was proposed by Werbos in his 1975 thesis.

A more concrete idea however, was presented by Rumelhart, Hinton and Williams in 1986, in the paper Learning representations by backpropagating errors.

Details

Basics

A feedforward network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation.

These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y.

Basics

Feedforward neural networks are called *networks* because they are typically represented by composing together many di□erent functions.

The model is associated with a directed acyclic graph describing how the functions are composed together.

For example, we might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the *first layer* of the network, $f^{(2)}$ is called the *second layer*, and so on.

The length of this chain is the depth of the model

Basics

Each hidden layer of the network is typically vector valued. The dimensionality of these hidden layers determines the width of the model.

Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as representing a single vector-to-vector function,we can also think of the layer as consisting of many units that act in parallel,each representing a vector-to-scalar function.

Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value.

Learning representations by back-propagating errors

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure.

Backpropagation

Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.)

The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input-output behaviour. This amounts to deciding what these units should represent.

Backpropagation

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \tag{1}$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, y_j, which is a non-linear function of its total input

$$y_j = \frac{1}{1+e^{-x_j}} \tag{2}$$

Backpropagation

It is not necessary to use exactly the functions given in equations (1) and (2). Any input-output function which has a bounded derivative will do. However, the use of a linear function for combining the inputs to a unit before applying the nonlinearity greatly simplifies the learning procedure.

The aim is to find a set of weights that ensure that for each input vector the output vector produced by the network is the same as (or sufficiently close to) the desired output vector. If there is a fixed, finite set of input-output cases, the total error in the performance of the network with a particular set of weights can be computed by comparing the actual and desired output vectors for every case. The total error, E, is defined as

$$E = \tfrac{1}{2} \sum_c \sum_j (y_{j,c} - d_{j,c})^2 \qquad (3)$$

where c is an index over cases (input-output pairs), j is an index over output units, y is the actual state of an output unit and d is its desired state. To minimize E by gradient descent it is necessary to compute the partial derivative of E with respect to each weight in the network. This is simply the sum of the partial derivatives for each of the input-output cases. For a given case, the partial derivatives of the error with respect to each weight are computed in two passes. We have already described the forward pass in which the units in each layer have their states determined by the input they receive from units in lower layers using equations (1) and (2). The backward pass which propagates derivatives from the top layer back to the bottom one is more complicated.

Backpropagation

The backward pass starts by computing $\partial E/\partial y$ for each of the output units. Differentiating equation (3) for a particular case, c, and suppressing the index c gives

$$\partial E/\partial y_j = y_j - d_j \tag{4}$$

We can then apply the chain rule to compute $\partial E/\partial x_j$

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot \mathrm{d}y_j/\mathrm{d}x_j$$

Differentiating equation (2) to get the value of $\mathrm{d}y_j/\mathrm{d}x_j$ and substituting gives

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot y_j(1-y_j) \tag{5}$$

This means that we know how a change in the total input x to an output unit will affect the error. But this total input is just a linear function of the states of the lower level units and it is also a linear function of the weights on the connections, so it is easy to compute how the error will be affected by changing these states and weights. For a weight w_{ji}, from i to j the derivative is

$$\begin{aligned}\partial E/\partial w_{ji} &= \partial E/\partial x_j \cdot \partial x_j/\partial w_{ji} \\ &= \partial E/\partial x_j \cdot y_i \end{aligned} \tag{6}$$

and for the output of the i^{th} unit the contribution to $\partial E/\partial y_i$

Backpropagation

resulting from the effect of i on j is simply

$$\partial E/\partial x_j \cdot \partial x_j/\partial y_i = \partial E/\partial x_j \cdot w_{ji}$$

so taking into account all the connections emanating from unit i we have

$$\partial E/\partial y_i = \sum_j \partial E/\partial x_j \cdot w_{ji} \tag{7}$$

We have now seen how to compute $\partial E/\partial y$ for any unit in the penultimate layer when given $\partial E/\partial y$ for all units in the last layer. We can therefore repeat this procedure to compute this term for successively earlier layers, computing $\partial E/\partial w$ for the weights as we go.

One way of using $\partial E/\partial w$ is to change the weights after every input–output case. This has the advantage that no separate memory is required for the derivatives. An alternative scheme, which we used in the research reported here, is to accumulate $\partial E/\partial w$ over all the input-output cases before changing the weights. The simplest version of gradient descent is to change each weight by an amount proportional to the accumulated $\partial E/\partial w$

$$\Delta w = -\varepsilon \partial E/\partial w \tag{8}$$

Universal Function Approximation

Cybenko 1989, Hornik et al, 1991, Leshno 1993, proved the following theorem.

Let $\sigma(.)$ be a bounded continuous function. Let I_p denote a p-dimensional hypercube and $C(I_p)$ denote the space of continuous functions on I_p. Given any $f \in C(I_p)$ and $\epsilon > 0$ there exists $q > 0$ and $v_i, w_i, i = 0, 1, ..., q$ such that $F(x) = \sum_{i<q} v_i \sigma(w_i^T x + b_i)$ such that $sup_{x \in I_p} |f(x) - F(x)| < \epsilon$.

Universal Function Approximation

It guarantees that even a single hidden-layer network can represent any classification problem in which the boundary is locally linear (smooth)

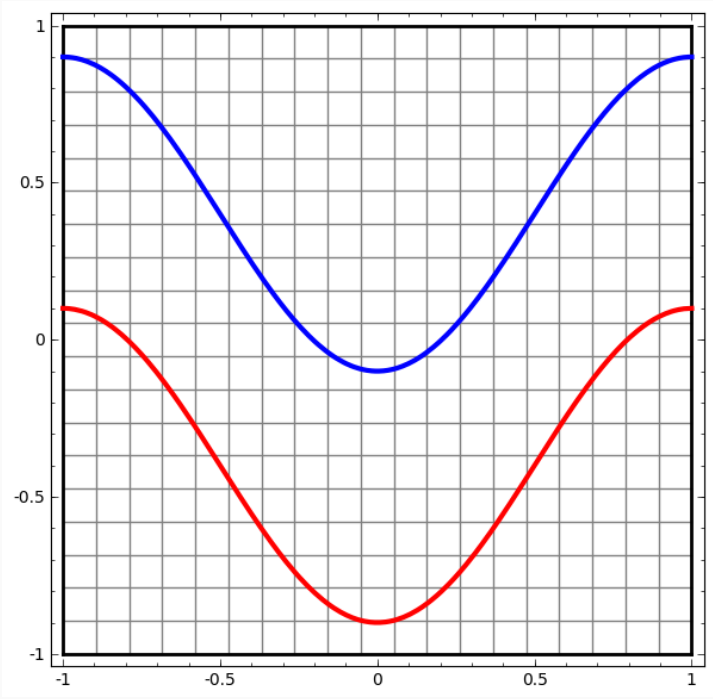
It does not inform about good/bad architectures, nor how they relate to the optimization procedure.

The universal approximation theorem generalizes to any non-polynomial (possibly unbounded) activation function, including the ReLU (Leshno, 1993).

What is the network learning?

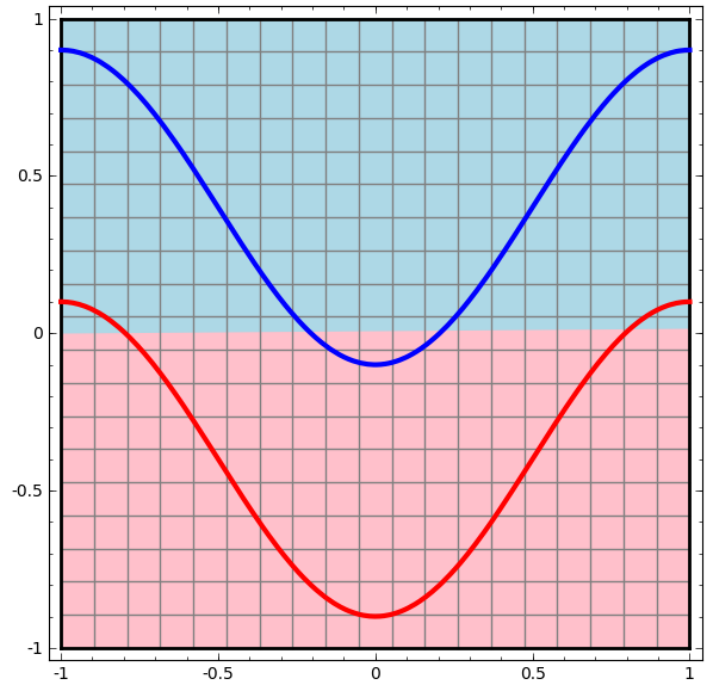
Manifold Manipulation

Consider the simple dataset, two curves on a plane. The network will learn to classify points as belonging to one or the other.



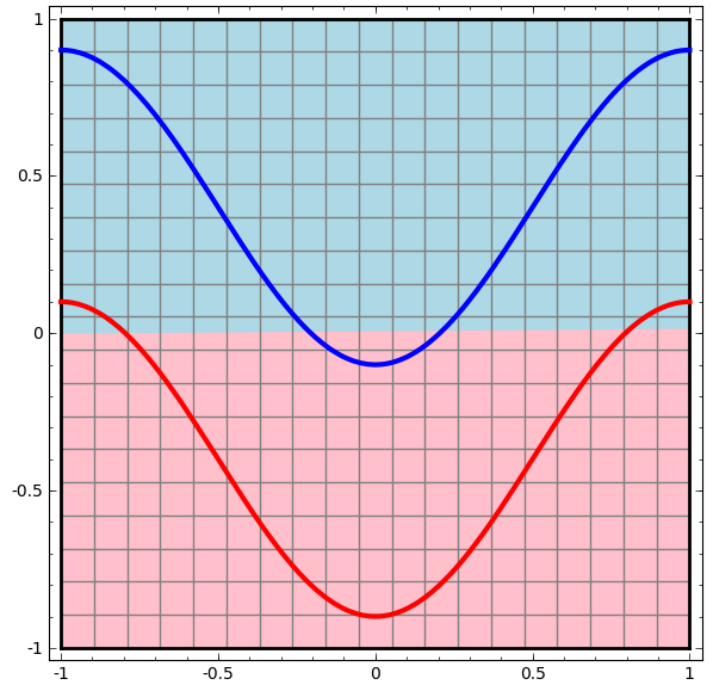
Manifold Manipulation

Consider simple network with one with only an input layer and an output layer. Such a network simply tries to separate the two classes of data by dividing them with a line.



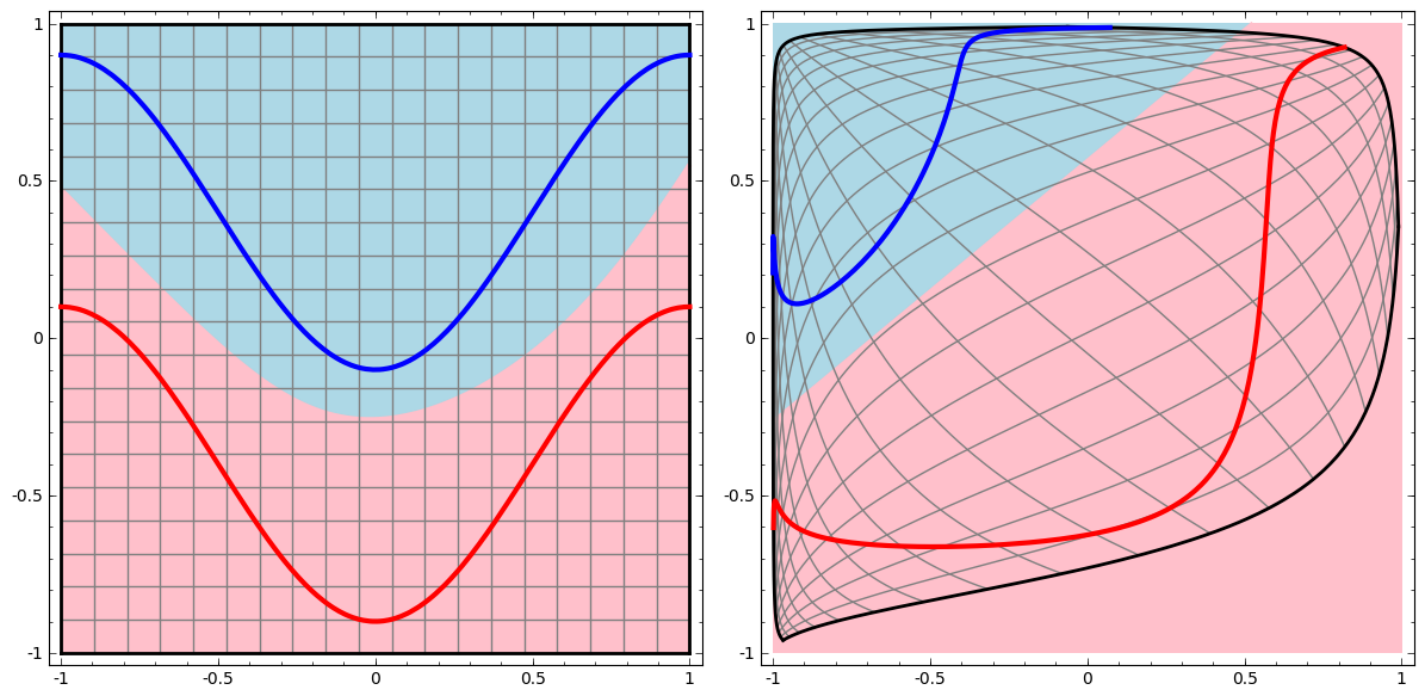
Manifold Manipulation

Consider simple network with one with only an input layer and an output layer. Such a network simply tries to separate the two classes of data by dividing them with a line.



Manifold Manipulation

A neural network with one hidden layer instead learns a curve to separate the curves. With each layer, the network transforms the data, creating a new representation.



The hidden layer learns a representation so that the data is linearly separable

Manifold Manipulation

Animations on
http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Manifold Hypothesis

The manifold hypothesis is that natural data forms lower-dimensional manifolds in its embedding space. There are both theoretical and experimental reasons to believe this to be true. The task of a classification algorithm is thus, fundamentally to separate a bunch of tangled manifolds.

Learnt Embedding Visualisation

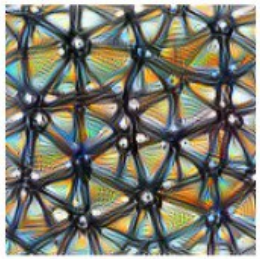
https://cs.stanford.edu/people/karpathy/cnnembed/cnn$_e$*mbed*$_4$*k.jpg*

Feature Visualization

`n` layer index
`x,y` spatial position
`z` channel index
`k` class index



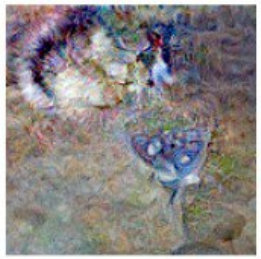

Neuron
`layer_n[x,y,z]`



Channel
`layer_n[:,:,z]`

Layer/DeepDream
`layer_n[:,:,:]^2`

Class Logits
`pre_softmax[k]`



Class Probability
`softmax[k]`

Feature Visualization

Dataset Examples show us what neurons respond to in practice

Optimization isolates the causes of behavior from mere correlations. A neuron may not be detecting what you initially thought.

Baseball—or stripes?
mixed4a, Unit 6

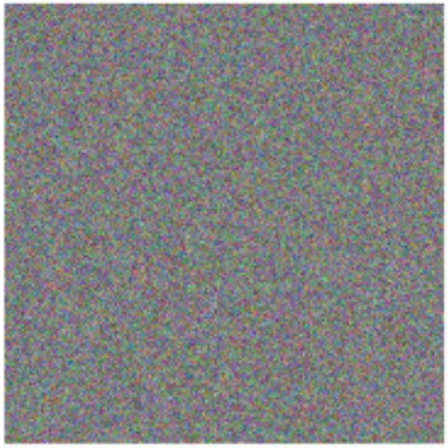
Animal faces—or snouts?
mixed4a, Unit 240

Clouds—or fluffiness?
mixed4a, Unit 453

Buildings—or sky?
mixed4a, Unit 492

Feature Visualization

Starting from random noise, we optimize an image to activate a particular neuron (layer mixed4a, unit 11).



Step 0

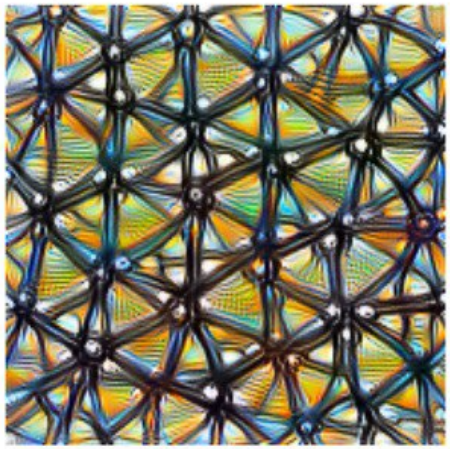
→

Step 4

→

Step 48

→

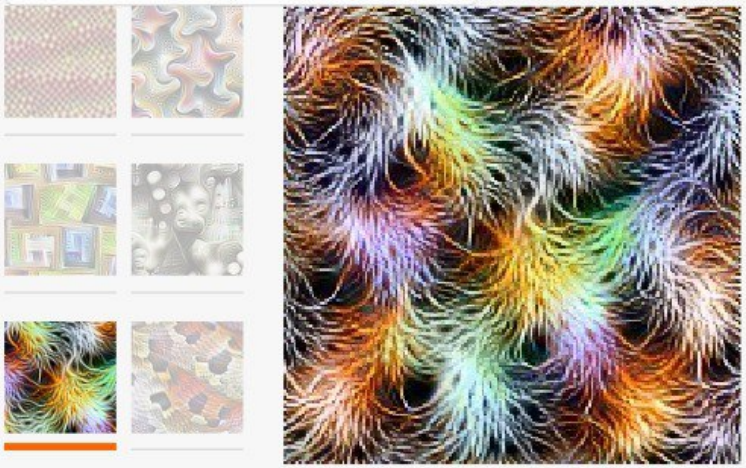


Step 2048

Feature Visualization

By jointly optimizing two neurons we can get a sense of how they interact.

REPRODUCE IN A NOTEBOOK

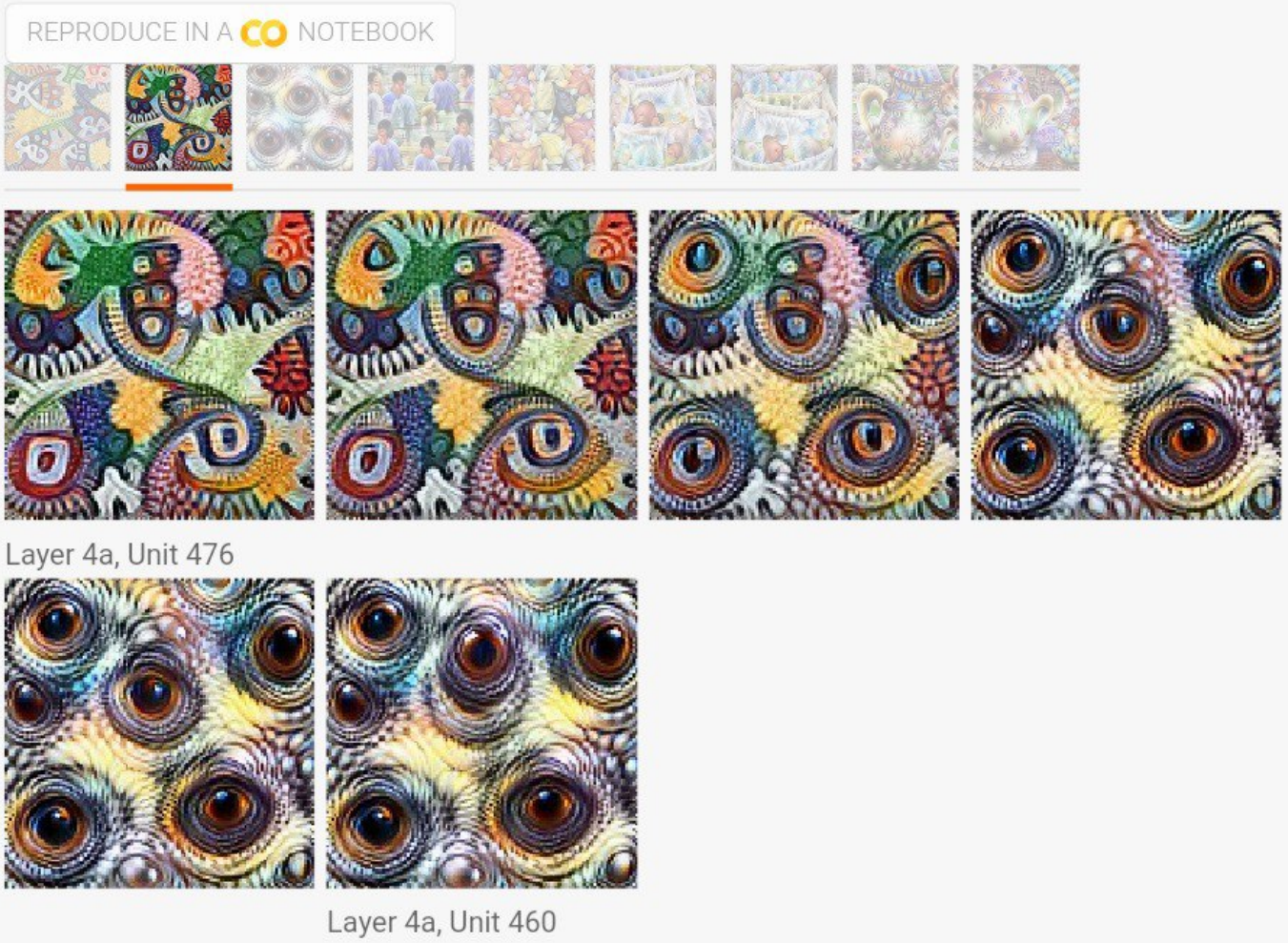


Neuron 1

Neuron 2

Jointly optimized

Feature Visualization



Memorisation Capacity of Neural Networks

A neural network was trained on CIFAR Images with random labels.

The network was able to memorise all the 60000 examples and achieve 100% training accuracy.

This indicates that the network does memorise samples, contradictory to the generalisation behavior normally observed.

Information Theory - Basics

A discrete random variable X is completely defined by the values it can take, χ, and its probability distribution $p_X(x)_{x \in X}$.

The entropy of a random variable is essentially the information content of the variable.

Information can be viewed as the decrease in uncertainty.

The rarer an event, the more information we gain if we know that it has occurred.

Information Theory - Mutual Information

Mutual Information is defined between two random variables.

It measures how information about one variable reduces uncertainty in the other variable's value.

Information Bottleneck Theory of Deep Learning

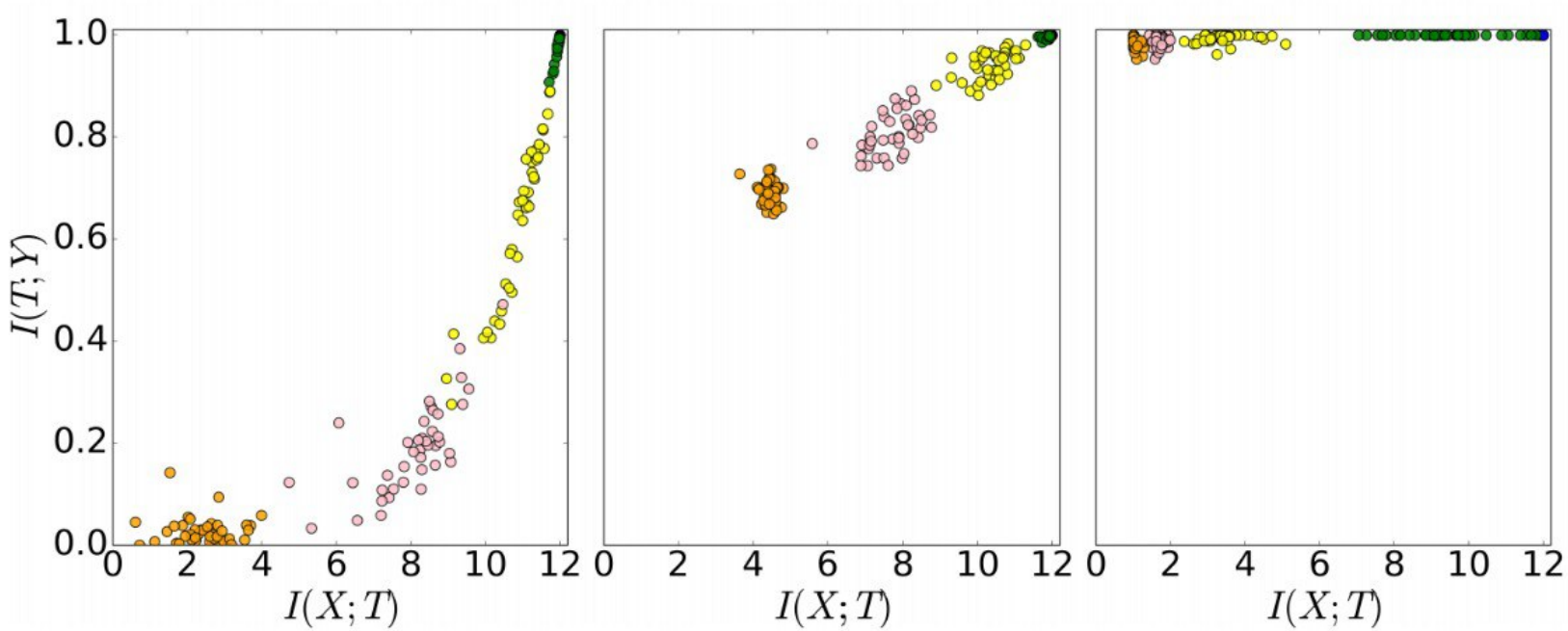


Figure 2: Snapshots of layers (different colors) of 50 randomized networks during the SGD optimization process in the *information plane* (in bits): **left** - with the initial weights; **center** - at 400 epochs; **right** - after 9000 epochs. The reader is encouraged to view the full videos of this optimization process in the *information plane* at *https://goo.gl/rygyIT* and *https://goo.gl/DQWuDD*.

Information Bottleneck Theory of Deep Learning

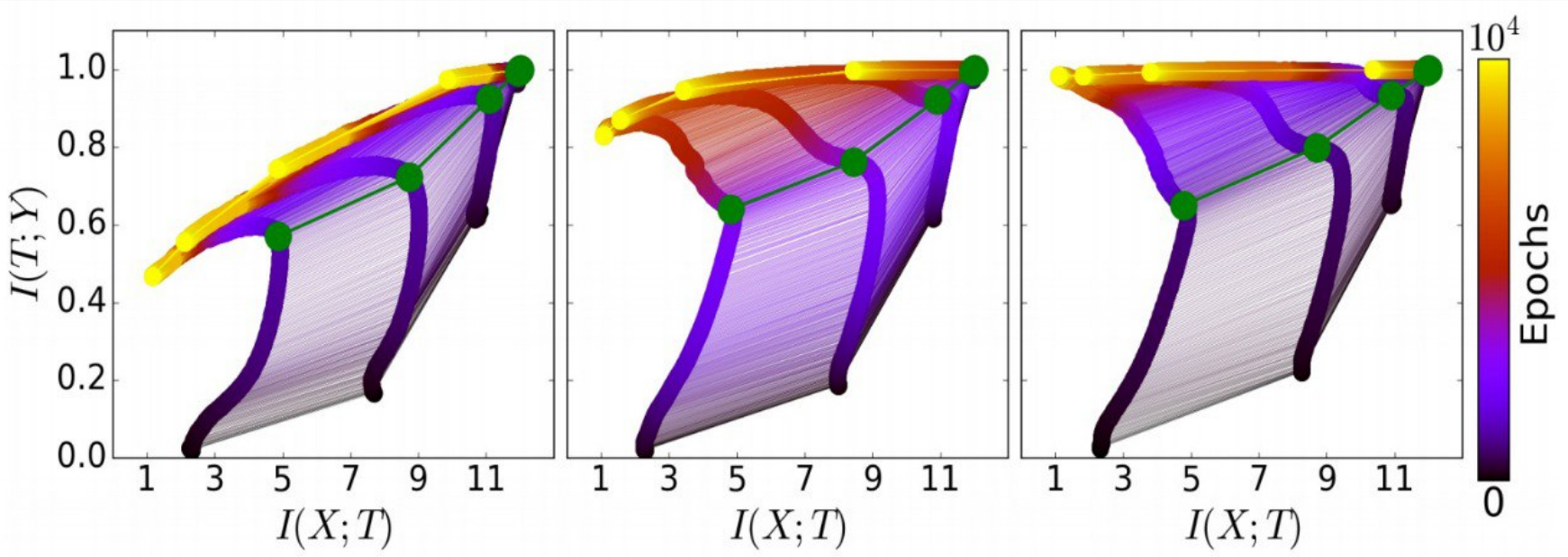


Figure 3: The evolution of the layers with the training epochs in the information plane, for different training samples. On the left - 5% of the data, middle - 45% of the data, and right - 85% of the data. The colors indicate the number of training epochs with Stochastic Gradient Descent from 0 to 10000. The network architecture was fully connected layers, with widths: input=12-10-8-6-4-2-1=output. The examples were generated by the spherical symmetric rule described in the text. The green paths correspond to the SGD drift-diffusion phase transition - grey line on Figure 4

Information Bottleneck Theory of Deep Learning

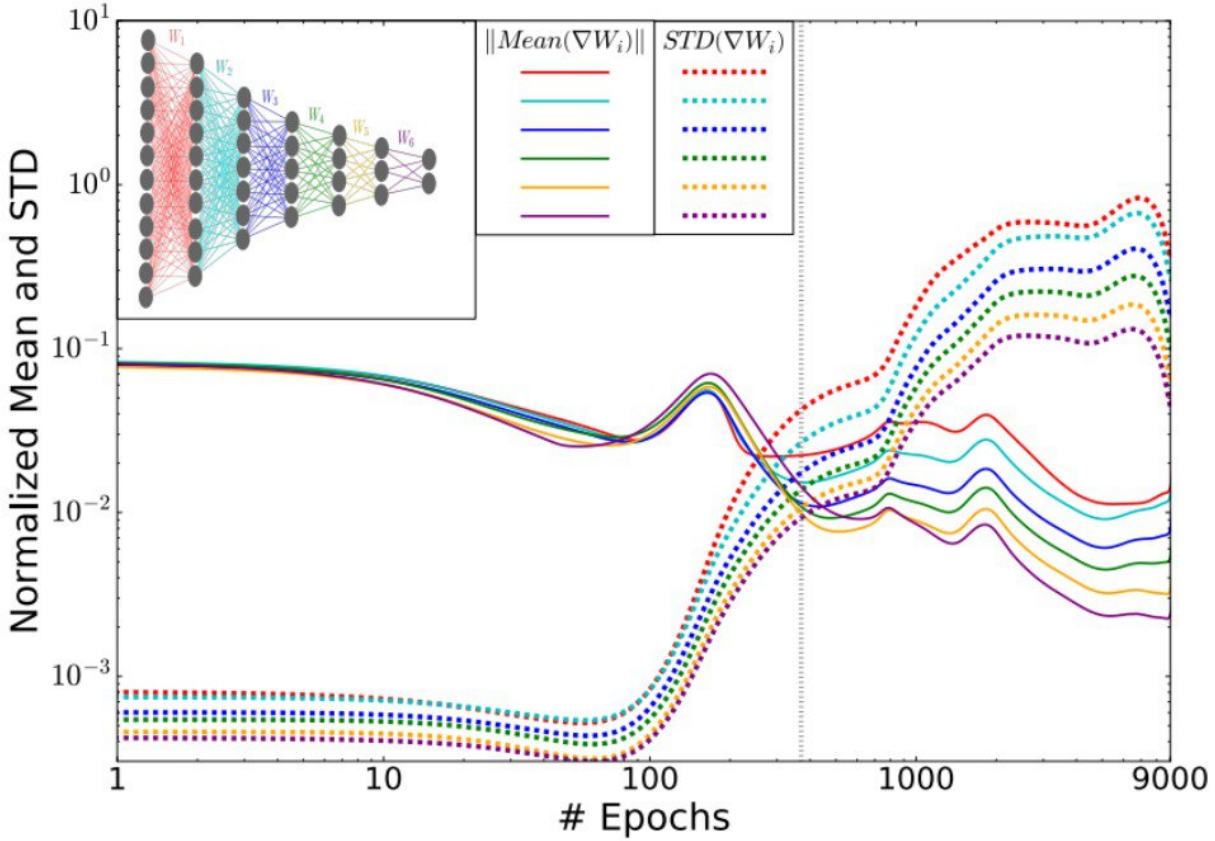


Figure 4: **The layers' Stochastic Gradients distributions during the optimization process.** The norm of the means and standard deviations of the weights gradients for each layer, as function of the number of training epochs (in log-log scale). The values are normalized by the L2 norms of the weights for each layer, which significantly increase during the optimization. The grey line ($\sim$ 350 epochs) marks the transition between the first phase, with large gradient means and small variance (*drift*, high gradient SNR), and the second phase, with large fluctuations and small means (*diffusion*, low SNR). Note that the gradients log (SNR) (the log differences between the mean and the STD lines) approach a constant for all the layers, reflecting the convergence of the network to a configuration with constant flow of relevant information through the layers!

Information Bottleneck Theory of Deep Learning

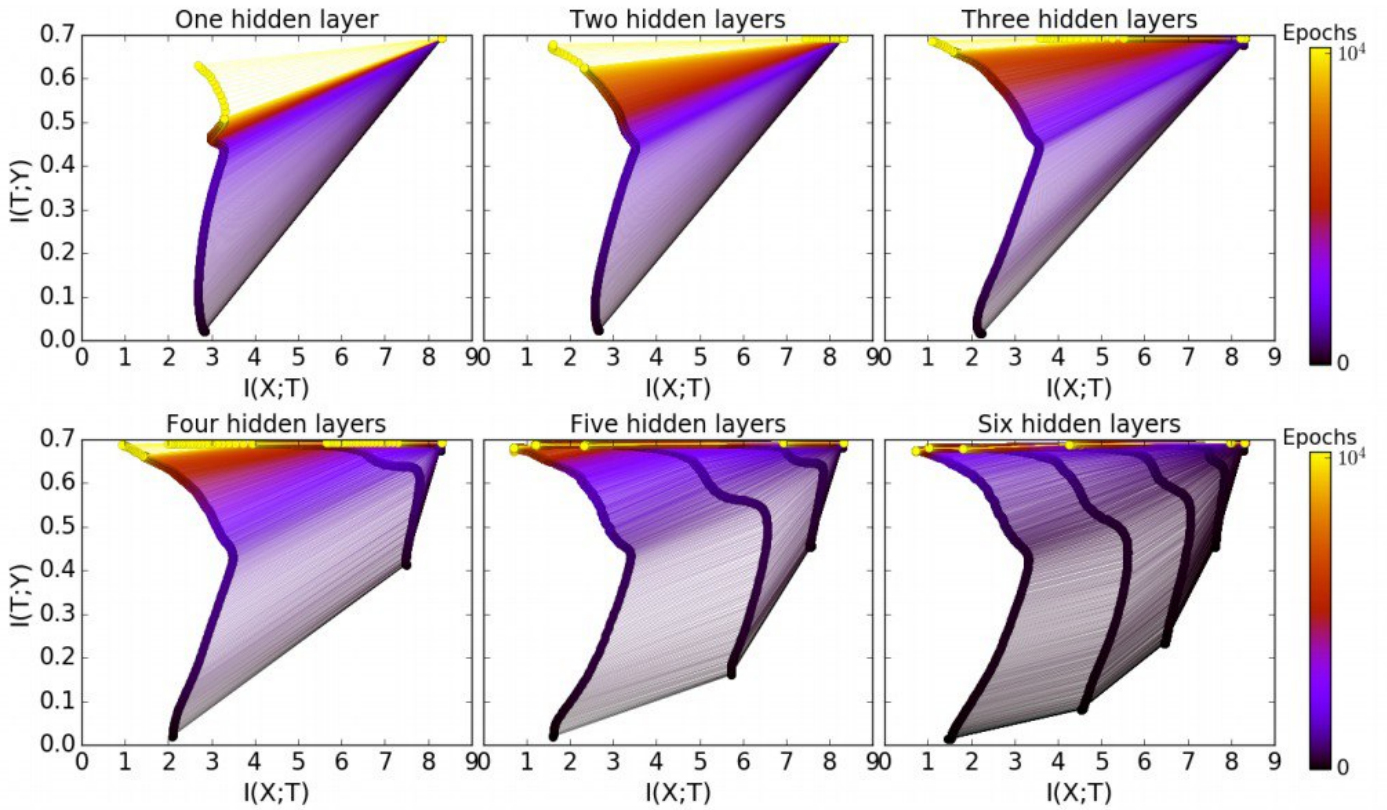


Figure 5: **The layers information paths during the SGD optimization for different architectures.** Each panel is the *information plane* for a network with a different number of hidden layers. The width of the hidden layers start with 12, and each additional layer has 2 fewer neurons. The final layer with 2 neurons is shown in all panels. The line colors correspond to the number of training epochs.

Conclusion

References i

- https://www.deeplearningbook.org/contents/mlp.html
- https://www.nature.com/articles/323533a0
- http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/ https://cs.stanford.edu/people/karpathy/cnnembed/
- http://benanne.github.io/2014/08/05/spotify-cnns.html
- https://glouppe.github.io/info8010-deep-learning/?p=lecture2.md
- https://distill.pub/2017/feature-visualization/
- https://distill.pub/2018/building-blocks/
- https://web.stanford.edu/class/ee376a/outline.html
- https://arxiv.org/abs/1611.03530
- https://arxiv.org/abs/1703.00810

Thank you.