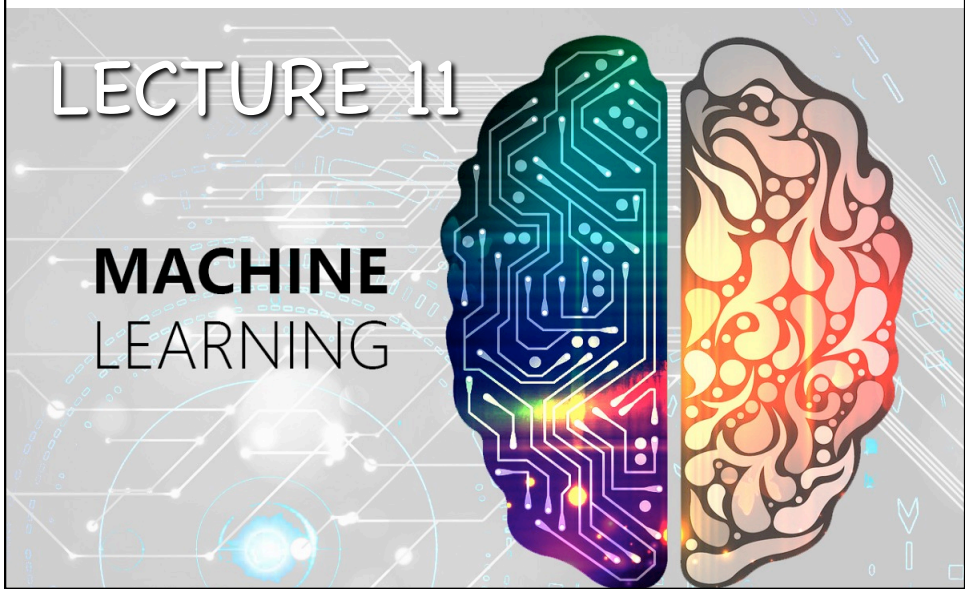
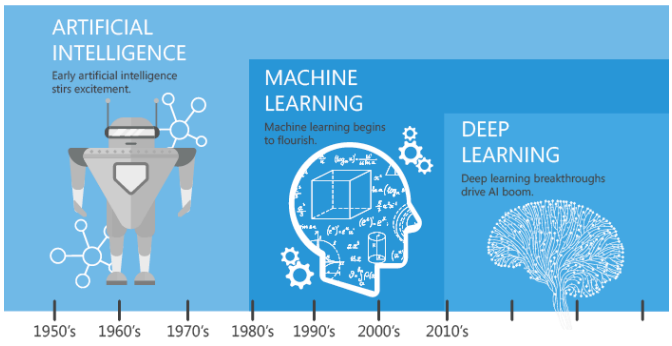
Numerical Simulation Laboratory



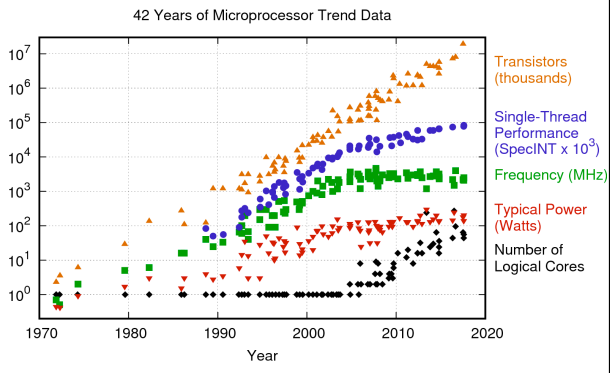
Outline

- Introduction to Machine Learning
- Introduction to feed-forward Neural Networks for supervised learning
- The backpropagation algorithm (supplementary material)

"Does your car have any idea why my car pulled it over?"



- Computers can calculate complex numerical calculations in a blink of an eye.
- In principle, they even have bigger processing power than human brain: 10^{10} transistors (but many more for a High Performance Parallel computer) with a switching time of 10^{-9} seconds against $9x10^{10}$ neurons in the whole nervous system (African elephant: $2.6x10^{11}$; human cerebral cortex: $1.6x10^{10}$; dolphin cerebral cortex: $3.7x10^{10}$), with a switching time of about 10^{-3} seconds.
- Still, brains outperform computers in many aspects … for sure from an energetic consumption point of view
- Another aspect is learning and making decisions based on knowledge/experience



3

4

Learning Machine Learning

- Machine Learning (ML), data science, and statistics are fields that describe how to learn from, and make predictions about, data. The availability of big datasets is a hallmark of modern science, including physics, where data analysis has become an important component of diverse areas, such as experimental particle physics, observational astronomy and cosmology, condensed matter physics, biophysics …
- Moreover, ML and data science are playing increasingly important roles in many aspects of modern technology … therefore, having a deep understanding of the concepts and tools used in ML is an important skill that is increasingly relevant in & out Physics … so let's start with learning machine learning!
- ML is a subfield of AI with the goal of developing algorithms capable of learning from data automatically. The basic premise of ML is to build algorithms that can receive input data and use statistical analysis to predict an output while updating outputs as new data becomes available.
- The last three decades have seen an unprecedented increase in our ability to generate and analyze large data sets.

Why study Machine Learning?

5

- This "big data" revolution has been spurred by an exponential increase in computing power and memory (Moore's law).
- Computations that were unthinkable a few decades ago can now be routinely performed on laptops. Specialized computing machines (such as GPU-based machines) are continuing this trend towards cheap, large-scale computation, suggesting that the "big data" revolution is here to stay.
- This increase in our computational ability has been accompanied by new techniques for analyzing and learning from large datasets. These techniques draw heavily from ideas in statistics, computational neuroscience, computer science, and physics.
- Physicists are uniquely situated to benefit from and contribute to ML. Many of the core concepts and techniques used in ML have their origins in Physics.
- Physicists have also been at the forefront of using "big data" and researchers are increasingly incorporating recent advances in ML and data science, and this trend is likely to accelerate in the future.

Types of Machine Learning

6

Traditional Programming

Give data to and run program on the computer to produce the output.

- ML can be divided into three broad categories:

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning

- While useful, the distinction between the three types of ML is sometimes fuzzy and fluid, and many applications often combine them in novel and interesting ways.
- For example, the recent success of Google DeepMind in developing ML algorithms that excel at tasks such as playing Go (https://doi.org/10.1038/nature16961) and video games employ deep reinforcement learning, combining reinforcement learning with supervised learning methods based on deep neural networks.

Supervised Learning

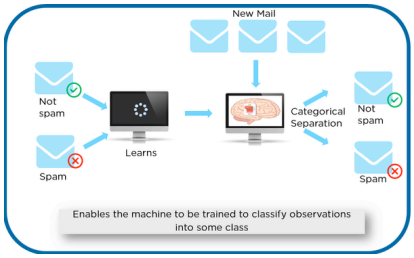
7

Traditional Programming

- Supervised learning concerns learning from labeled data, each data is tagged with the correct label(for example, a collection of pictures labeled as containing a cat or not containing a cat).

Machine Learning (supervised)

- The goal is to approximate the mapping function so well that when you have new input data (x) that you can predict the output variables (Y) for that data.



- Common supervised learning tasks include classification (A classification problem is when the output variable is a category, such as "red" or "blue" or "disease" and "no disease") and regression (the output variable is a real value, e.g. the value of a fitted function)

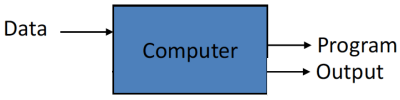
Unsupervised Learning

8

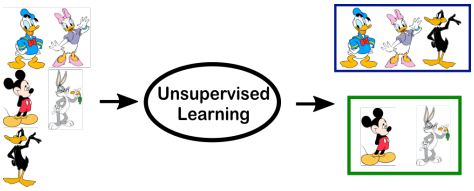
Traditional Programming

- Unsupervised learning is concerned with finding patterns and structure in unlabeled data.

Machine Learning (unsupervised)



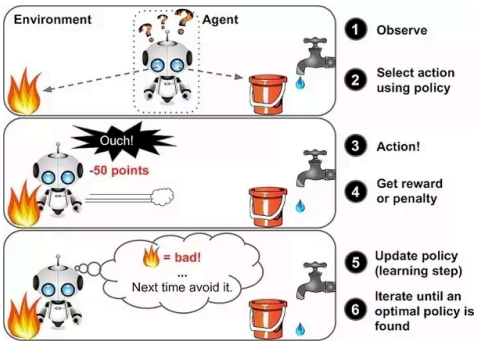
- Examples of unsupervised learning include clustering (you want to discover the inherent groupings in the data, such as grouping customers by purchasing behavior), association (you want to discover rules that describe large portions of your data, such as people that buy X also tend to buy Y), generative modeling, etc.



- In our training data, we don't provide any label to the corresponding data. The unsupervised model is able to separate both the characters by looking at the type of data and models the underlying structure or distribution in the data in order to learn more about it.

Reinforcement Learning

- Finally, a reinforcement learning algorithm, or agent, learns by interacting with its environment. The agent receives rewards by performing correctly and penalties for performing incorrectly.
- The agent learns without intervention from a human by maximizing its reward and minimizing its penalty.
- In the example, the agent is given 2 options i.e. a path with water or a path with fire.
- A reinforcement algorithm works on reward a system i.e. if the agent uses the fire path then the rewards are subtracted and agent tries to learn that it should avoid the fire path.



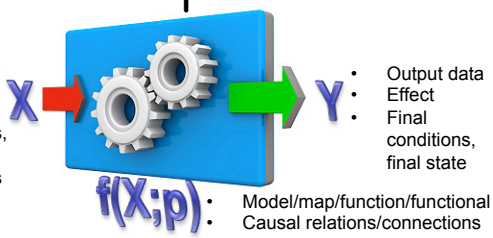
- If it had chosen the safe path then the reward points would have been added. The agent then would try to learn what path is safe and what path isn't.

Reinforcement Learning in action!

- In the following application of reinforcement learning agent has only two possible moves/actions: right or left
- It (or he?) can adjust also the rapidity of the moves

Setting up a problem in supervised ML

Many problems in supervised ML and data science starts with the same ingredients:



- Input data
- Cause
- Initial conditions, boundary conditions

- Output data
- Effect
- Final conditions, final state

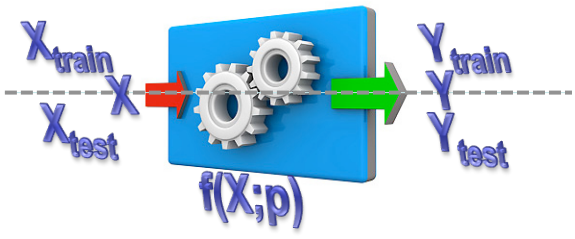
- Model/map/function/functional
- Causal relations/connections
- Theory/parameters

- The first ingredient is the dataset D = (X; Y) where X is a set of independent variables and Y is a set of dependent variables.
- The second is the model f(X;p), which is a function f: x→y of the parameters p. That is, f is a function used to predict an output from a vector of input variables.
- The final ingredient is the cost function C[y;f(X;p)] that allows us to judge how well the model performs on the observations y. The model is fit by finding the value of p that minimizes the cost function. For example, one commonly used cost function is the squared error. Minimizing the squared error cost function is known as the method of least squares

Train & test

ML researchers and data scientists follow a standard recipe to obtain models that are useful for prediction problems.

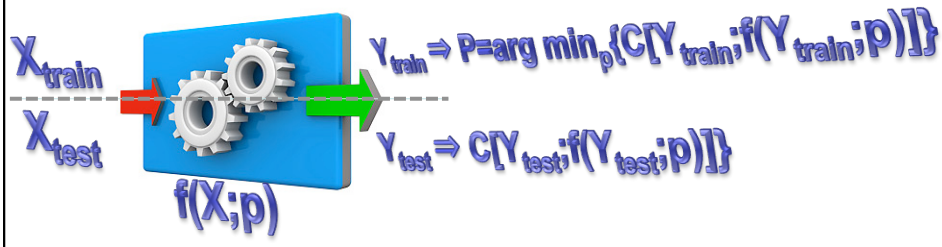
- The first step in the analysis is to randomly divide the dataset D into two mutually exclusive groups D_{train} and $D_{test/validation}$ called the training and test/validation sets. Performing some analysis (such as using the data to select important variables) before partitioning the data is a common pitfall that can lead to incorrect conclusions.



- Typically, the majority of the data are partitioned into the training set (e.g., 90%) with the remainder going into the test set.

13

- The model is fit by minimizing the cost function using only the data in the training set $P=\arg\min_p\{C[Y_{train};f(X_{train};p)]\}$



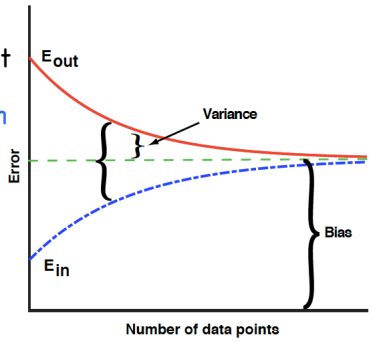
- Finally, the performance of the model is evaluated by computing the cost function using the test set $C[Y_{test};f(X_{test};p)]$
- The value of the cost function for the best fit model on the training set is called the in-sample (or training) error $E_{in} = C[Y_{train};f(X_{train};p)]$ and the value of the cost function on the test set is called the out-of-sample (or generalization) error $E_{out} = C[Y_{test};f(X_{test};p)]$

14

- One of the most important observations we can make is that $E_{out} \geq E_{in}$ *almost* always. This is because the test data was now used in the fitting procedure.
- Splitting the data into mutually exclusive training and test sets provides an unbiased estimate for the predictive performance of the model; this is known as cross-validation in the ML and statistics literature.
- In many applications of classical statistics, we start with a mathematical model that we assume to be true (e.g., we may assume that Hooke's law, a linear relation among displacement and force, is true if we are observing a mass-spring system) and our goal is to estimate the value of some unknown model parameters (the spring stiffness).
- Problems in ML, by contrast, typically involve inference about complex systems where we do not know the exact form of the mathematical model that describes the system. Therefore, it is not uncommon for ML researchers to have multiple candidate models that need to be compared. This comparison is usually done using E_{out}; the model that minimizes E_{out} is chosen as the best model

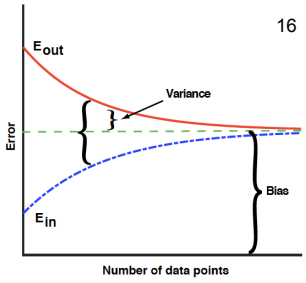
Statistical Learning Theory

- This raises a natural question: Can we say something general about the relationship between E_{in} and E_{out}? This is the domain of statistical learning theory, and we give a brief overview of the main results in this section.
- The basic intuitions of statistical learning can be summarized in three simple observations. The first one, shown in the following figure, shows the typical out-of-sample error, E_{out}, and in-sample error, E_{in}, as a function of the amount of training data.
- In making this graph, we have assumed that the true data is drawn from a complicated distribution, so that we cannot exactly learn f(x).
- Hence, after a quick initial drop (not shown in figure), E_{in} will increase with the number of data points, because our models are not powerful enough to learn the true function we are seeking to approximate.



- In contrast, E_{out} will decrease with the number of data points.

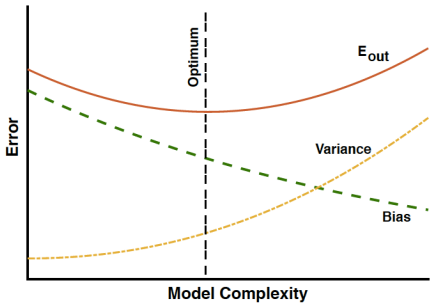
- As the number of data points gets large, the training data set becomes more representative of the true distribution from which the data is drawn.
- For this reason, in the infinite data limit, E_{in} and E_{out} must approach the same value, which is called the "bias" of our model.



- In general, the more complex the model class we use, the smaller the bias. However, we do not generally have an infinite amount of data. For this reason, we cannot know the bias and we have to minimize E_{out}
- E_{out} can be naturally decomposed into a bias, which measures how well we can hypothetically do in the infinite data limit, and a variance, which measures the typical errors introduced in training our model due to having a finite training set
- The final quantity shown in figure is the difference between the generalization and training error. It measures how much worse we would do on a new data set compared to our training data. Models with a large difference between the in-sample and out-of-sample errors are said to "overfit" the data.

17

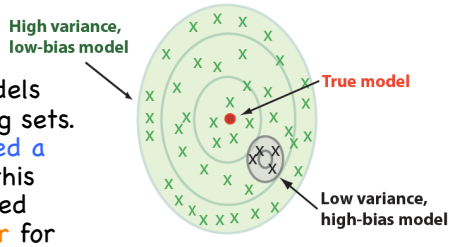
- One of the lessons of statistical learning theory is that it is not enough to simply minimize the training error, because the out-of-sample error can still be large.
- The second observation shows E_{out} as a function of "model complexity". In many cases, model complexity is related to the number of parameters we are using to approximate the true function f(x)



- If we consider a training dataset of a fixed size, E_{out} will be a non-monotonic function of the model complexity, and is generally minimized for models with intermediate complexity
- In fact, even though using a more complicated model always reduces the bias, at some point the model becomes too complex for the amount of training data and the generalization error becomes large due to high variance.
- Thus, to minimize E_{out} and maximize our predictive power, it may be more suitable to use a more biased model with small variance than a less-biased model with large variance.

18

- This important concept, our third observation, is commonly called the bias-variance tradeoff and gets at the heart of why machine learning is difficult
- Another way to visualize the bias-variance tradeoff is shown in the new figure. Here, we imagine training a complex model (shown in green) and a simpler model (shown in black) many times on different training sets of a fixed size N.

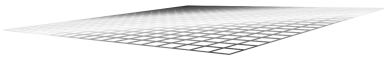


- Due to the uncertainty from having finite size data sets, the learned models will differ for each choice of training sets. In general, more complex models need a larger amount of training data. For this reason, the fluctuations in the learned models (variance) will be much larger for the more complex model than the simpler model.
- However, if we consider the asymptotic performance as we increase the size of the training set, it is clear that the complex model will eventually perform better than the simpler model. Thus, depending on the amount of training data, it may be more favorable to use a less complex, high-bias model to make predictions.

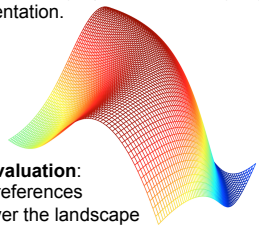
Key Elements of Machine Learning

All machine learning algorithms are combinations of three components:

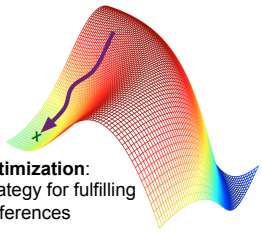
- Representation: how to represent knowledge. Examples include decision trees, graphical models, neural networks, support vector machines and others.
- Evaluation: the way to evaluate candidate programs (hypotheses). Examples include accuracy, squared error, likelihood, posterior probability, cost, entropy, KL divergence and others.
- Optimization: the way candidate programs are generated, known as the search process. For example, stochastic gradient descent, quasi-Newton methods, convex optimization, constrained optimization, etc.



Representation: the *landscape* of possible models, the playing field allowed by a given representation.



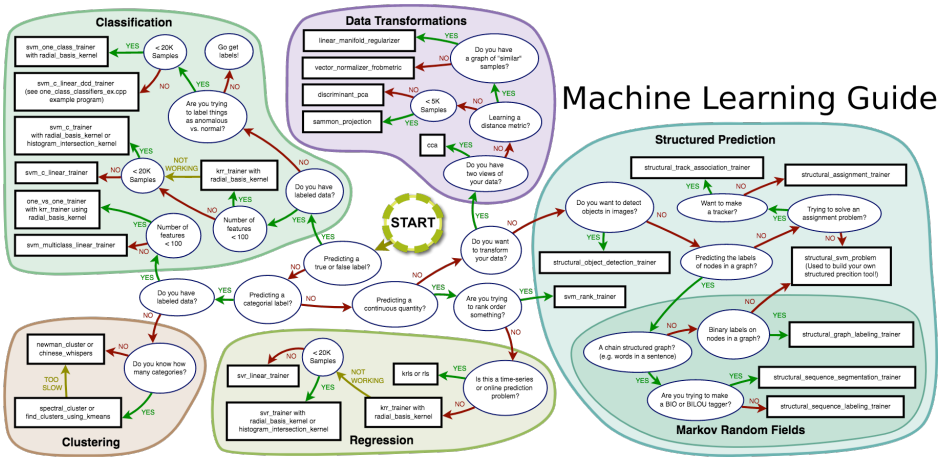
Evaluation: Preferences over the landscape



Optimization: Strategy for fulfilling preferences

Machine Learning is huge!

The picture above is just to show that ML researcher over the years have developed a large set of techniques to solve specific problems to learn from data …



We will go on introducing one of the most powerful and widely used algorithms when it comes to the subfield of ML called deep learning

An introduction to feedforward Neural Networks

- Over the last decade, Neural Networks (NNs) have emerged as the one of most powerful and widely-used supervised learning techniques.
- NNs have a long history, but re-emerged to prominence after a rebranding as "DeepLearning" and Deep Neural Networks (DNNs) in the mid 2000s.
- DNNs truly caught the attention of the wider machine learning community and industry in 2012 when a group of data scientists used a GPU-based DNN model (AlexNet) to lower the error rate on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) by an incredible twelve percent from 28% to 16%. Just three years later, a machine learning group from Microsoft achieved an error of 3.57% using an ultradeep residual neural network (ResNet) with 152 layers!
- Since then, DNNs have become the workhorse technique for many image and speech recognition based machine learning tasks. The large-scale industrial deployment of DNNs has given rise to a number of high-level libraries and packages (Caffe, Keras, Pytorch, and TensorFlow) that make it easy to quickly code and deploy DNNs.

NNs categories

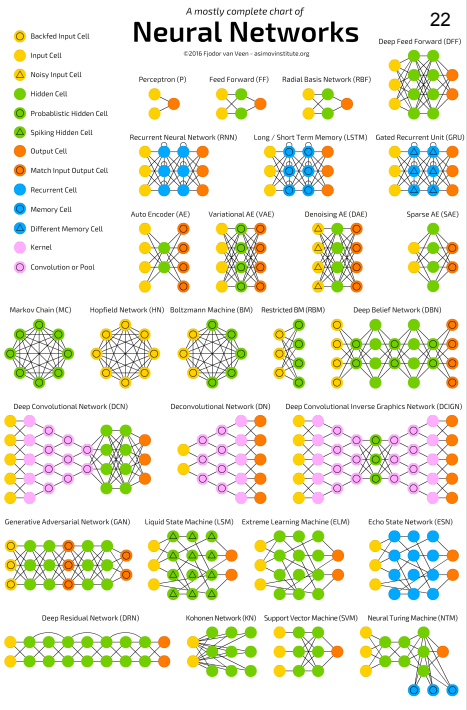
- Conceptually, it is helpful to divide neural networks into four categories:

(i) general purpose neural networks for supervised learning,

(ii) neural networks designed specifically for image processing, the most prominent example of this class being Convolutional Neural Networks (CNNs)

(iii) neural networks for sequential data such as Recurrent Neural Networks (RNNs), and

(iv) neural networks for unsupervised learning such as Deep Boltzmann Machines.



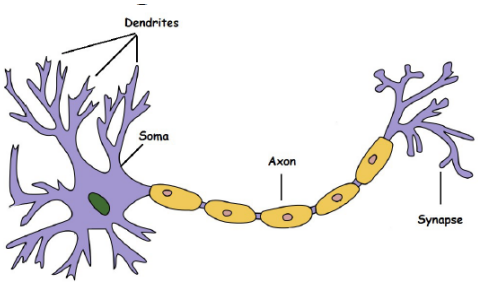
NN & Physics

In physics, DNNs and CNNs have already found numerous applications:

- In classical and quantum statistical physics, they have been applied to detect phase transitions
- At the same time, methods from statistical physics have been applied to the field of deep learning to study the efficiency of learning rules, to explore the representation capabilities of DNNs, make analogies between training DNNs and spin glasses
- Deep CNNs were used in lensing reconstruction of the cosmic microwave background
- DNNs have been used to improve the efficiency of Monte-Carlo algorithms
- Representing quantum states as DNNs and quantum state tomography are among some of the impressive achievements to reveal the potential of DNNs to facilitate the study and control of quantum systems.
- For a recent review on ML in Physics: G. Carleo, et. al. , *Machine learning and the physical sciences*, arXiv:1903.10563 (Mar 2019)

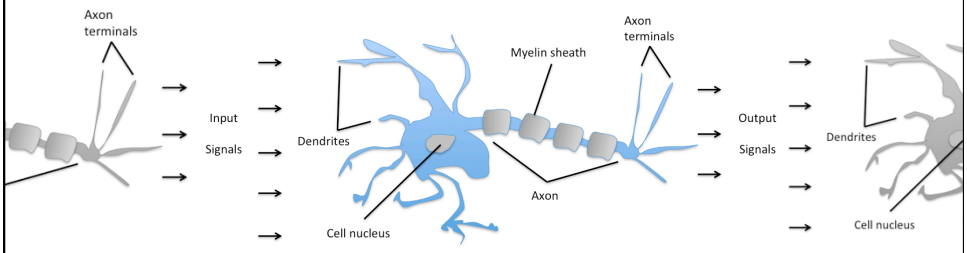
Biology behind the idea

- So, what nature's concepts are Neural Networks trying to imitate?
- As you are probably aware, the smallest unit of the nervous system is a neuron. These are cells with similar and simple structure. Yet, by continuous communication, these cells achieve enormous processing power.
- If you put it in simple terms, neurons are just switches. These switches generate an output signal if they receive a certain amount of input stimuli. This output signal is input for another neuron.
- Each neuron has these components: Body (also known as soma), Dendrites, Axon
- Body (soma) of a neuron carries out the basic life processes of a neuron. Every neuron has a single axon. This is a long part of the cell; it acts like a wire and it is an output of the neuron.



25

- Dendrites, on the other hand, are inputs of neuron and each neuron has multiple dendrites. These inputs and outputs, axons and dendrites, of different neurons come close via axon terminals: the synapses



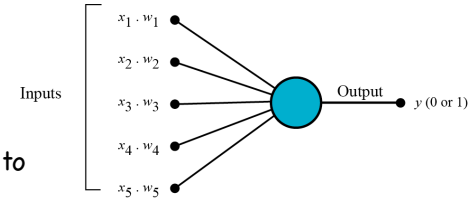
Schematic of a biological neuron.

- Trough the synapses signals are carried by neurotransmitter molecules, and each serves a different type of neuron (e.g. the famous serotonin and dopamine).
- The amount and type of these chemicals will dictate how "strong" the input to the neuron is. And, if there is enough input on all dendrites, the soma will "fire up" the signal on the axon, and transmit it to the next neuron.

Perceptrons

26

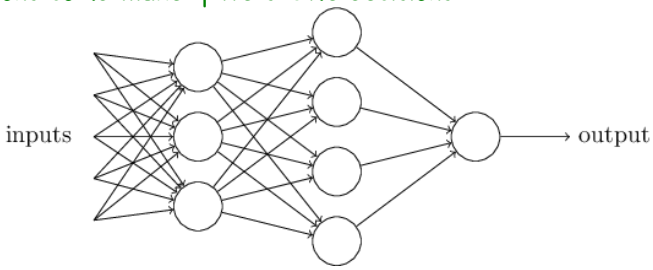
- What is a neural network? To get started, I'll explain a type of artificial neuron called a perceptron. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts.
- Today, it's more common to use other models of artificial neurons - in much modern work on neural networks, the main neuron model used is one called the sigmoid neuron. We'll get to sigmoid neurons shortly. But to understand why sigmoid neurons are defined the way they are, it's worth taking the time to first understand perceptrons.
- So how do perceptrons work? A perceptron takes several binary inputs, x_1, x_2, ..., and produces a single binary output:



- Rosenblatt proposed a simple rule to compute the output. He introduced weights, w_1, w_2, ..., real numbers expressing the importance of the respective inputs to the output. The output, 0 or 1, is determined by whether the weighted sum $\Sigma_j w_j x_j$ is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron.

27

- A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence/data.
- Obviously, the perceptron isn't a complete model of human decision-making! But it should seem plausible that a complex network of perceptrons could make quite subtle decisions:

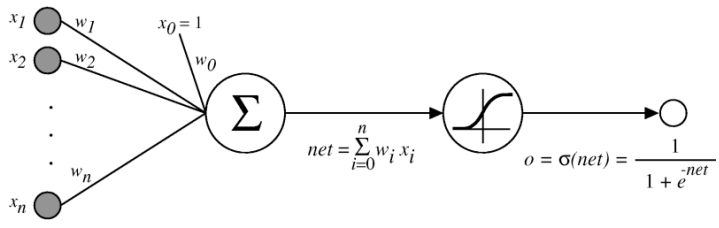


- There is however problem when our network contains perceptrons. In fact, a small change in the weights or threshold of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way
- That makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to the desired behaviour.

28

Sigmoid neurons

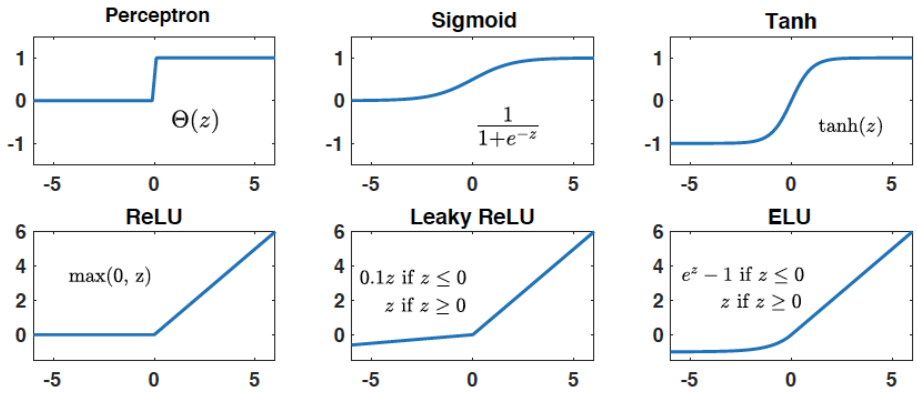
- We can overcome this problem by introducing a new type of artificial neuron called a sigmoid neuron. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.



- Just like a perceptron, the sigmoid neuron has inputs, x_1, x_2, ... But the output is not 0 or 1. Instead, it's $\sigma(z)=1/(1+\exp(-z))$, where σ is called the sigmoid function, $z=\mathbf{w}\cdot\mathbf{x}+w_0$ and w_0 (or b) the bias.
- Note that It's only when $\mathbf{w}\cdot\mathbf{x}+b$ is of modest size that there's much deviation from the perceptron model. The exact form of σ isn't so important - what really matters is the shape of the function.

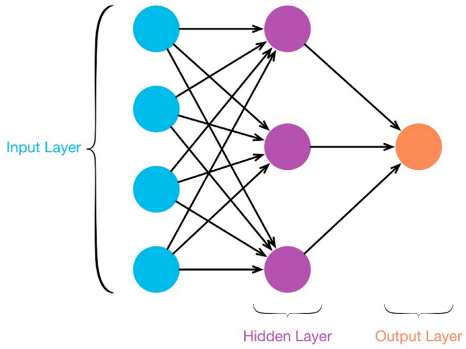
Activation functions

- Various possible non-linear activation functions for neurons:



- In modern DNNs, it has become common to use non-linear functions that do not saturate for large inputs (bottom row) rather than saturating functions (top row).

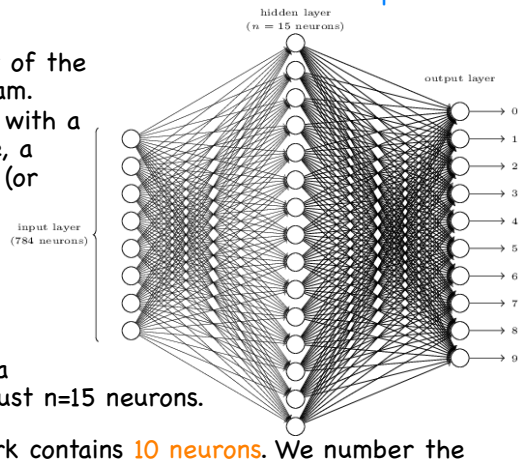
- If σ had in fact been a step function, then the sigmoid neuron would be a perceptron, since the output would be 1 or 0 depending on whether $\mathbf{w} \cdot \mathbf{x} + w_0$ was positive or negative
- It's the smoothness of the σ function that is the crucial fact, not its detailed form. The smoothness of σ means that small changes Δw_j in the weights and Δw_0 in the bias will produce a small change Δ_{out} in the output from the neuron.
- The basic idea of all neural networks is to layer neurons in a hierarchical fashion; suppose to have a complex network obtained combining more sigmoid neurons:
- The leftmost layer in this network is called the input layer, and the neurons within the layer are called input neurons. The rightmost or output layer contains the output neurons, or, as in this case, a single output neuron. The middle layer is called a hidden layer.



- The design of the input and output layers in a network is often straightforward because they should adapt to the dataset D = (X; Y)

31

- For example, suppose we're trying to determine the digit in a handwritten image. A natural way to design the network is to encode the intensities of the image pixels into the input neurons. If the image is a 28 by 28 greyscale image, then we'd have 784=28 × 28 input neurons
- For simplicity I've omitted most of the 784 input neurons in the diagram. The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0 representing black (or the opposite!), and in between values representing gradually darkening shades of grey.
- The second layer of the network is a hidden layer. The example shown illustrates a small hidden layer, containing just n=15 neurons.



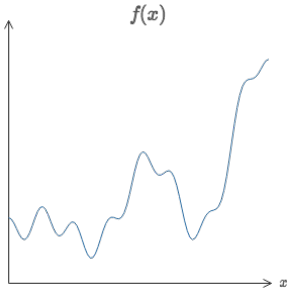
- The output layer of the network contains 10 neurons. We number the output neurons from 0 through 9, and figure out which neuron has the highest activation value. If that neuron is, say, neuron number 6, then our network will guess that the input digit was a 6. And so on for the other output neurons.

Universal approximation theorems

- Thus, the whole neural network can be thought of as a complicated nonlinear transformation of the inputs X into an output Y that depends on the weights and biases of all the neurons in the input, hidden, and output layers.
- The use of hidden layers greatly expands the representational power of a NN. Perhaps, the most formal expression of the increased representational power of neural networks is the universal approximation theorem which states that a neural network with a single hidden layer can approximate any continuous, multi-input/multi-output function with arbitrary accuracy (G. Cybenko, 1989, for sigmoid; Kurt Hornik, 1991, etc. etc.)
- Although feed-forward networks with a single hidden layer are universal approximators, the width of such networks has to be exponentially large
- In 2017 Lu et al. proved universal approximation theorem for width-bounded deep neural networks. In particular, they showed that width-n+4 networks with ReLU activation functions can approximate any Lebesgue integrable function on n-dimensional input space if the depth of the network is allowed to grow

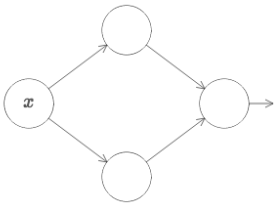
32

- These results tells us that neural networks have a kind of universality. No matter what function we want to compute, we know that there is a neural network which can do the job.
- One can give a simple and mostly visual explanation of the universality theorem
- The basic idea is that hidden neurons allow neural networks to generate step functions (characteristic functions) with arbitrary offsets and heights. These can then be added together to approximate arbitrary functions.
- The proof also makes clear that the more complicated a function, the more hidden units (and free parameters) are needed to approximate it. Hence, the applicability of the approximation theorem to practical situations should not be overemphasized
- To understand why the universality theorem is true, let's start by understanding how to construct a neural network which approximates a function with just one input and one output ...

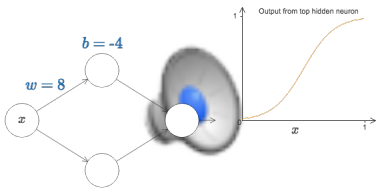


33

- Once we've understood this special case it's actually pretty easy to extend to functions with many inputs and many outputs.
- To build insight into how to construct a network to compute f, let's start with a network containing just a single hidden layer, with two hidden neurons, and an output layer containing a single output neuron:



- To get a feel for how components in the network work, let's focus on the top hidden neuron. What's being computed by the hidden neuron is $\sigma(wx+b)$, where $\sigma(z) \equiv 1/(1+e^{-z})$ is the sigmoid function.

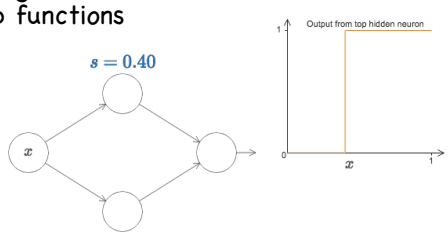


- By increasing the weight, w, the curve gets steeper, until eventually it begins to look like a step function. On the contrary, by decreasing the weight, the curve broadens out
- As the bias decreases the graph moves to the right, but its shape doesn't change; as the bias increases the graph moves to the left, but, again, its shape doesn't change.

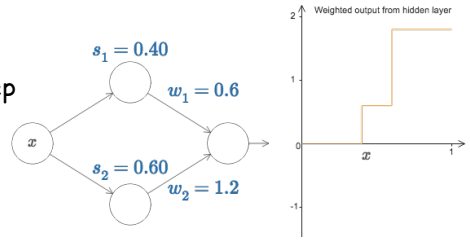
34

35

- It's actually quite a bit easier to work with step functions than general sigmoid functions. The reason is that in the output layer we add up contributions from all the hidden neurons. It's easy to analyze the sum of a bunch of step functions, but rather more difficult to reason about what happens when you add up a bunch of sigmoid shaped curves. And so it makes things much easier to assume that our hidden neurons are outputting step functions



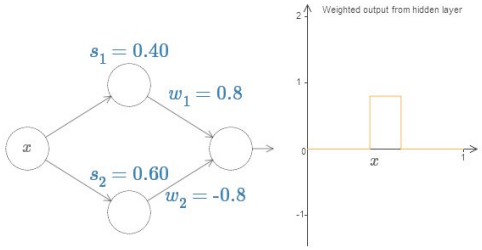
- We do this by fixing the weight w to be some very large value. It will greatly simplify our lives to describe hidden neurons using just a single parameter, s, which is the step position, s=-b/w



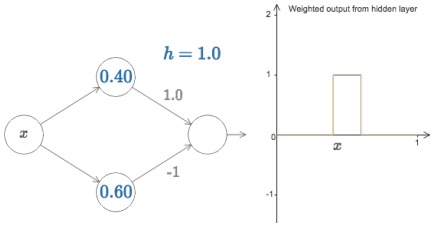
- Let's take a look at the behavior of the entire network. In particular, we'll suppose the hidden neurons are computing step functions parameterized by step points s_1 (top neuron) and s_2 (bottom neuron). And they'll have output weights w_1 and w_2

36

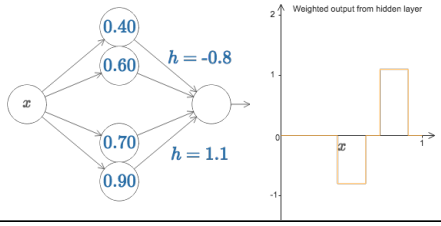
- Finally, setting w_1 to be positive and w_2=-w_1 negative, you can get a "bump" function, which starts at point s_1, ends at point s_2, and has height h. For instance, the weighted output might look like this:



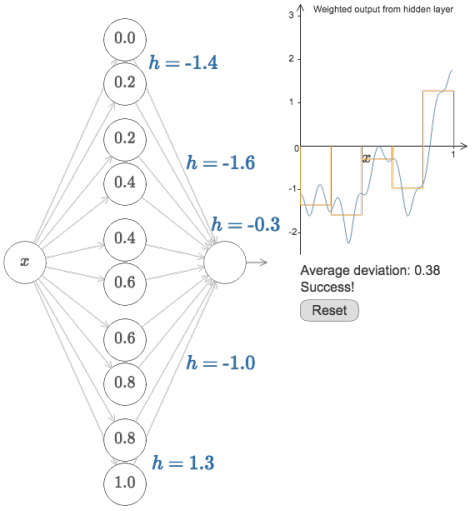
- Of course, we can rescale the bump to have any height at all. Let's use a single parameter, h, to denote the height.

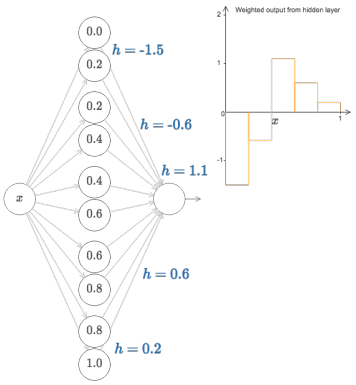


- We can use our bump-making trick to get two bumps, by gluing two pairs of hidden neurons together into the same network:



- And we can use this idea to get as many peaks as we want, of any height. In particular, we can divide the interval [0,1] up into a large number, N, of subintervals, and use N pairs of hidden neurons to set up peaks of any desired height.



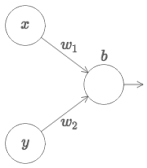


- You've now figured out all the elements necessary for the network to approximately compute the function f(x)! It's only a coarse approximation, but we could easily do much better, merely by increasing the number of pairs of hidden neurons, allowing more bumps.

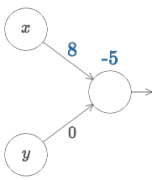
37

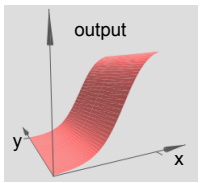
Many input variables

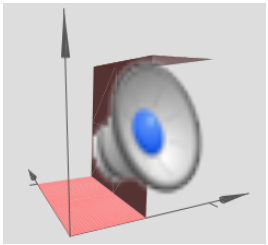
38



- Let's extend our results to the case of many input variables. We'll start by considering what happens when we have two inputs to a neuron:
- As you can see, with w_2=0 the input y makes no difference to the output from the neuron. It's as though x is the only input.
- Given this, what do you think happens when we increase the weight w_1, with w_2=0? Just as in our earlier discussion, as the input weight gets larger the output approaches a step function. The difference is that now the step function is in 3D. Also as before, we can move the location of the step point around by modifying the bias. The actual location of the step point is s_x=-b/w_1.

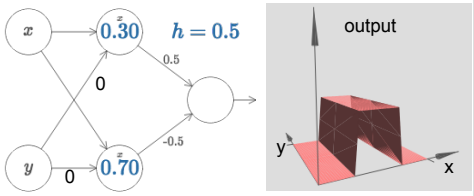




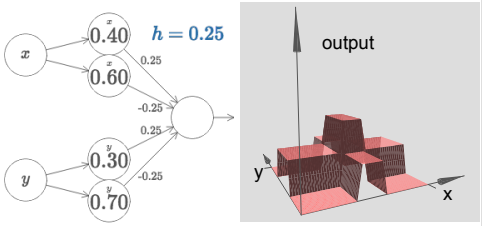


39

- Of course, it's also possible to get a step function in the y direction, by making the weight on the y input very large, and the weight on the x equal to 0
- We can use the step functions we've just constructed to compute a three-dimensional bump function. To do this, we use two neurons, each computing a step function in the x direction. Then we combine those step functions with weight h and -h, respectively, where h is the desired height of the bump.

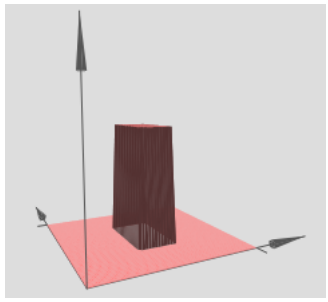


- Of course, we can easily make a bump function in the y direction, by using two step functions in the y direction
- Let's consider what happens when we add up two bump functions, one in the x direction, the other in the y direction, both of height h (To simplify the diagram I've dropped the connections with zero weight.):

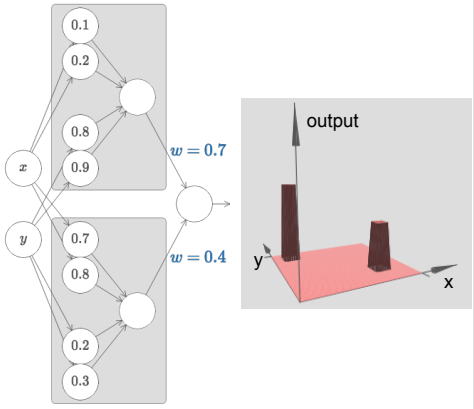


40

- If we choose the threshold appropriately - say, a value of 3h/2, which is sandwiched between the height of the plateau and the height of the central tower - by adjusting the bias we could squash the plateau down to zero, and leave just the tower standing.

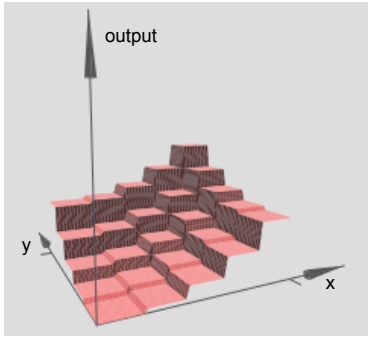


- Let's try gluing two such networks together, in order to compute two different tower functions. To make the respective roles of the two sub-networks clear I've put them in separate boxes, below: each box computes a tower function, using the technique described above. The graph on the right shows the weighted output from the second hidden layer, that is, it's a weighted combination of tower functions.



41

- The same idea can be used to compute as many towers as we like.
- We can also make them as thin as we like, and whatever height we like. As a result, we can ensure that the weighted output from the second hidden layer approximates any desired function of two variables:



- By using more input neurons and adjusting the complexity of the NN you can approximate functions of many variables
- If you want to practice directly yourself with step functions, go to: http://neuralnetworksanddeeplearning.com
Especially, see Chapter 4

42

- Modern neural networks generally contain multiple hidden layers (hence the "deep" in deep learning).
- Increasing the number of layers increases the number of parameters and hence the representational power of neural networks.
- Indeed, recent numerical experiments suggests that as long as the number of parameters is larger than the number of data points, certain classes of neural networks can fit arbitrarily labeled random noise samples. This suggests that large neural networks of the kind used in practice can express highly complex functions
- Other works suggest that it is computationally and algorithmically easier to train deep networks rather than shallow, wider nets, though this is still an area of major controversy and active research
- Choosing the exact network architecture for a neural network remains an art that requires extensive numerical experimentation and intuition, and is often times problem-specific. Both the number of hidden layers and the number of neurons in each layer can affect the performance of a neural network.
- There seems to be no single recipe for the right architecture for a neural net that works best.

43

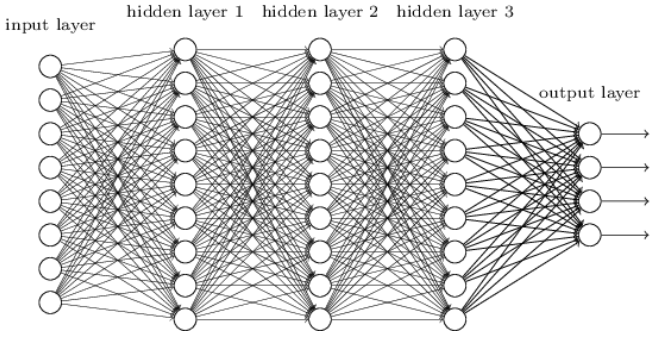
- However, a general rule of thumb that seems to be emerging is that the number of parameters in the neural net should be large enough to prevent underfitting

- Empirically, the best architecture for a problem depends on the task, the amount and type of data that is available, and the computational resources at one's disposal. Certain architectures are easier to train, while others might be better at capturing complicated dependencies in the data and learning relevant input features.
- Finally, there have been numerous works that move beyond the simple deep, feed-forward neural network architectures discussed here. For example, modern neural networks for image segmentation often incorporate "skip connections" that skip layers of the neural network (He et al., 2016). This allows information to directly propagate to a hidden or output layer, bypassing intermediate layers and often improving performance.

44

Training deep networks

Neural networks differ from simpler supervised procedures in that generally they contain multiple hidden layers that make taking the gradient computationally more difficult.



How can we fit a model with a NN & DNN? The basic procedure for training neural nets is:

- construct a cost/loss function and then use some gradient descent technique to minimize the cost function and find the optimal weights and biases.

The cost/loss function

- Recall that in most cases, the top output layer of our neural net is either a continuous predictor or a classifier that makes discrete (categorical) predictions. Depending on whether one wants to make continuous or categorical predictions, one must utilize a different kind of loss function
- For continuous data, the loss functions that are commonly used include the mean squared error

$$E(\vec{w}) = \frac{1}{n}\sum_{i=1}^{n}\left|\vec{Y}_i - f(\vec{Y}_i;\vec{w})\right|^2$$

- And the mean absolute error

$$E(\vec{w}) = \frac{1}{n}\sum_{i=1}^{n}\left|\vec{Y}_i - f(\vec{Y}_i;\vec{w})\right|$$

- The full cost function often includes additional terms that implement some sort of regularization
- For categorical data, when the output layer is taken to be a classifier for binary data with only two types of labels (or a softmax classifier if there are more than two types of labels), the most commonly used loss function is the cross-entropy.

- The cross-entropy between the true labels $Y_i \in \{0;1\}$ and the predictions is given by

$$E(\vec{w}) = -\sum_{i=1}^{n} Y_i \log f(Y_i;\vec{w}) + (1-Y_i)\log\left[1 - f(Y_i;\vec{w})\right]$$

- Note, in fact, that if the weights **w** predict the correct labeling of the data, for example $f(Y_i;\mathbf{w}) \approx 0$ when $Y_i=0\ \forall i$ except when i=k, where $Y_k=1$ and $f(Y_k;\mathbf{w})\approx 1$, the cross-entropy is

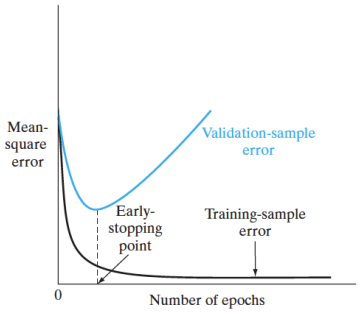
$$E(\vec{w}) \approx -Y_k \log f(Y_k;\vec{w}) - \sum_{i=1(\neq k)}^{n} \log\left[1 - f(Y_i;\vec{w})\right] \approx 0$$

  otherwise $E(\mathbf{w})>0$

- Having defined an architecture and a cost function, we must now train the model. Similar to other supervised learning methods, we make use of gradient descent-based methods to optimize the cost function.
- Depending on the architecture, data, and computational resources, different optimizers may work better on the problem, though vanilla Stochastic Gradient Descent is a good first choice.
- Finally, we note that calculating the gradients for a neural network requires a specialized algorithm, called Backpropagation which forms the heart of any neural network training procedure.

Early-Stopping Method of Training

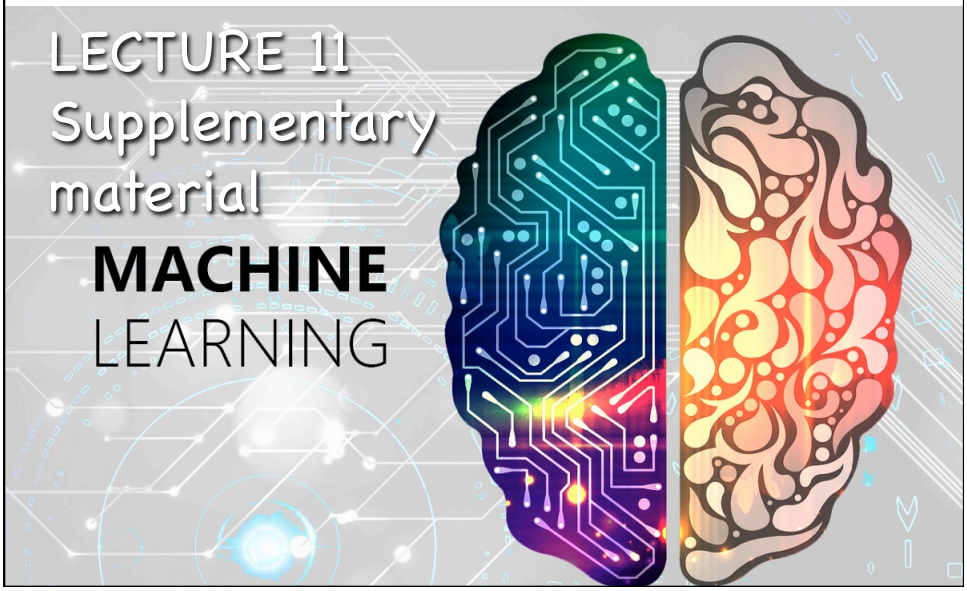
- Ordinarily, a DNN trained with the back-propagation algorithm learns in stages, moving from the realization of fairly simple to more complex mapping functions as the training session progresses.
- This process is exemplified by the fact that in a typical situation, the mean-square error decreases with an increasing number of epochs used for training, and it is very difficult to figure out when it is best to stop training if we were to look at the learning curve for training all by itself
- In particular, it is possible for the network to end up overfitting the training data if the training session is not stopped at the right point.
- We may identify the onset of overfitting through the use of cross-validation by comparing E_{in} & E_{out}
- This heuristic suggests that the minimum point on the validation learning curve be used as a sensible criterion for early-stopping the training session



Lecture 11: Suggested books

- P. Mehta, C.H. Wang, A.G.R. Day, and C. Richardson, *A high-bias, low-variance introduction to Machine Learning for physicists*, arXiv:1803.08823 (Feb 2019)
- M. Nielsen, *Neural Networks and Deep Learning*, http://neuralnetworksanddeeplearning.com, on-line book
- S. Haykin, *Neural Networks and Learning Machines*, Pearson (2009)
- S. Theodoridis, *Machine Learning. A Bayesian and Optimization Perspective*, Elsevier (2015)
- K.P. Murphy, *Machine Learning. A Probabilistic Perspective*, MIT press (2012)

Numerical Simulation Laboratory



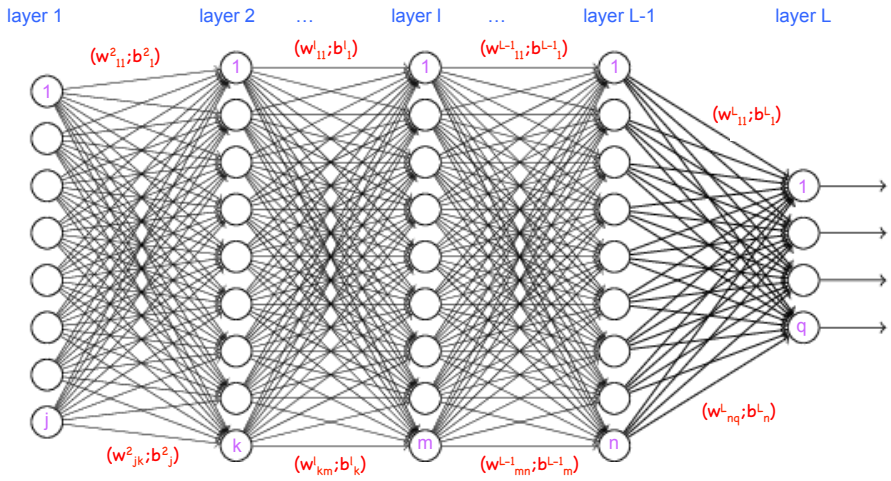
The backpropagation algorithm

- The training procedure requires us to be able to calculate the derivative of the cost function with respect to all the parameters of the neural network (the weights and biases of all the neurons in the input, hidden, and visible layers).
- A brute force calculation is out of the question since it requires us to calculate as many gradients as parameters at each step of the gradient descent.
- The backpropagation algorithm (Rumelhart and Zipser, 1985) is a clever procedure that exploits the layered structure of neural networks to more efficiently compute gradients
- At its core, backpropagation is simply the ordinary chain rule for partial differentiation (derivative of the composition of two or more functions), and can be summarized using four equations.
- In order to see this, we must first establish some useful notation. We will assume that there are L layers in our network with $l = 1, \ldots, L$ indexing the layer.

51

Backpropagation equations

- Let's denote by w^l_{jk} the weight for the connection from the k-th neuron in layer l-1 to the j-th neuron in layer l. Also, we denote the bias of this neuron by b^l_j .



52

- By construction, in a feed-forward neural network the activation a^l_j of the j-th neuron in the l-th layer can be related to the activities of the neurons in the layer l-1 by the equation

$$a^l_j = \sigma\left(\sum_k w^i_{jk} a^{l-1}_k + b^l_j\right) = \sigma\left(z^l_j\right)$$

where we have defined the linear weighted sum $z^l_j = \sum_k w^i_{jk} a^{l-1}_k + b^l_j$

- By definition, the cost function E depends directly on the activities of the output layer a^L_j. It of course also indirectly depends on all the activities of neurons in lower layers in the neural network through iteration of the previous equation for a^l_j
- Let us define the error Δ^L_j of the j-th neuron in the L-th layer as the change in cost function with respect to the weighted input z^L_j

$$\Delta^L_j = \frac{\partial E}{\partial z^L_j} \qquad (I)$$

This definition is the first of the 4 backpropagation equations

- We can analogously define the error of neuron j in layer l, Δ^l_j , as the change in the cost function with respect to the weighted input z^l_j: $\Delta^l_j = \frac{\partial E}{\partial z^l_j} = \frac{\partial E}{\partial a^l_j}\sigma'(z^l_j)$

- Where $\sigma'(x)$ denotes the derivative of the non-linearity $\sigma(\cdot)$ with respect to its input evaluated at x.
- Notice that the error function Δ^l_j can also be interpreted as the partial derivative of the cost function with respect to the bias b^l_j , since

$$\Delta^l_j = \frac{\partial E}{\partial z^l_j} = \frac{\partial E}{\partial b^l_j}\frac{\partial b^l_j}{\partial z^l_j} = \frac{\partial E}{\partial b^l_j} \qquad (II)$$

  where in the last line we have used the fact that $\partial b^l_j / \partial z^l_j = 1$. This is the second of the 4 backpropagation equations.
- We now derive the final 2 backpropagation equations using the chain rule. Since the error depends on neurons in layer l only through the activation of neurons in the subsequent layer l+1, we can use the chain rule to write

$$\Delta^l_j = \frac{\partial E}{\partial z^l_j} = \sum_k \frac{\partial E}{\partial z^{l+1}_k}\frac{\partial z^{l+1}_k}{\partial z^l_j} = \sum_k \Delta^{l+1}_k \frac{\partial z^{l+1}_k}{\partial z^l_j} = \left(\sum_k \Delta^{l+1}_k w^{l+1}_{jk}\right)\sigma'(z^l_j) \qquad (III)$$

  This is the third backpropagation equation.

- The final equation can be derived by differentiating of the cost function with respect to the weight w^l_{jk} as

$$\frac{\partial E}{\partial w^l_{jk}} = \frac{\partial E}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{jk}} = \Delta^l_j a^{l-1}_k \qquad (IV)$$

- Together, Eqs. (I), (II), (III), and (IV) define the 4 backpropagation equations relating the gradients of the activations of various neurons a^l_j , the weighted inputs z^l_j and the errors Δ^l_j .
- These equations can be combined into a simple, computationally efficient Backpropagation Algorithm to calculate the gradient with respect to all parameters:

1. **Activation at input layer**: calculate the activations a^1_j of all the neurons in the input layer.
2. **Feedforward**: starting with the first layer, compute z^l and a^l for each subsequent layer.
3. **Error at top layer**: calculate the error of the top layer using Eq.(I) (this requires to know the expression for the derivative of both $E(w) = E(a^L)$ and $\sigma(z)$.
4. **"Backpropagate" the error**: use Eq.(III) to propagate the error backwards and calculate Δ^1_j **for all layers.**
5. **Calculate gradient**: use Eqs.(II)&(IV) to calculate $\partial E/\partial b^1_j$ and $\partial E/\partial w^1_{jk}$

55

- We can now see where the name backpropagation comes from. The algorithm consists of a forward pass from the bottom layer to the top layer where one calculates the weighted inputs and activations of all the neurons.
- One then backpropagates the error starting with the top layer down to the input layer and uses these errors to calculate the desired gradients.
- This description makes clear the incredible utility and computational efficiencyof the backpropagation algorithm.
- We can calculate all the derivatives using a single "forward" and "backward" pass of the neural network. This computational efficiency is crucial since we must calculate the gradient with respect to all parameters of the neural net at each step of gradient descent
- Now it seems like it should be straightforward to train any neural network. However, until fairly recently it was widely believed that training deep networks was an extremely difficult task. One reason for this was that even with backpropagation, gradient descent on large networks is extremely computationally expensive. However, the great advances in computational hardware has made this a much less vexing problem than even a decade ago.