DD2363/2023 – Lecture 12
Learning from data (ch.16)

Johan Hoffman

Gradient descent minimization

ALGORITHM 15.1. x = gradient_descent_method(f, x0).
Input: objective function **f**, and initial guess **x0**.
Output: solution vector **x**.

```
1: x[:] = x0[:]
2: Df = compute_gradient(f, x)
3: while norm(Df) > TOL do
4:     Df = compute_gradient(f, x)
5:     alpha = get_step_length(f, Df, x)
6:     x[:] = x[:] - alpha*Df
7: end while
8: return x
```

Newton's method

ALGORITHM 15.3. x = newton_minimization(f, x0).
Input: function **f**, initial guess **x0**.
Output: solution vector **x**.

```
1: x[:] = x0[:]
2: Df = compute_gradient(f, x)
3: while norm(Df) > TOL do
4:     Df = compute_gradient(f, x)
5:     Hf = compute_hessian(f, x)
6:     dx = solve_linear_system(Hf, -Df)
7:     x[:] = x[:] + dx
8: end while
9: return x
```

Stochastic gradient method

Now consider a minimization problem for which the objective function is a sum,

$$\min_{x \in D} f(x), \quad f(x) = \sum_{i=1}^{n} f_i(x).$$

ALGORITHM 15.4. x = stochastic_gradient_descent_method(f_array, alpha, x0, no_iter).
Input: array of functions **f_array**, initial guess **x0**, step size **alpha**, number of iterations **no_iter**.
Output: solution vector **x**.

```
1: x[:] = x0[:]
2: for i=0:no_iter -1 do
3:     f = random_select(f_array)
4:     Df = compute_gradient(f, x)
5:     x[:] = x[:] - alpha*Df
6: end for
7: return x
```

Regularization

Minimization often represents an *ill-posed problem*, for which no unique solution exists. To make the problem *well-posed*, with a unique solution, one can favour or exclude certain solutions by modifying the objective function using *Tikhonov regularization*,

$$\min_{x \in D} f(x) + \|\Gamma x\|^2,$$

where $\Gamma \in R^{n \times n}$ is the *Tikhonov matrix*.

Example 15.7. *L^2-regularization* corresponds to $\Gamma = \epsilon I$, with $\epsilon \in R$ and I the $n \times n$ identity matrix, which favours solutions to the minimization problem which are small in the L^2 norm.

Constrained minimization

Often the search space is defined by a set of equality and inequality constraints, for which the *constrained minimization problem* in R^n takes the following form,

$$\begin{aligned} &\min_{x \in R^n} f(x), \\ &g(x) = c, \\ &h(x) \leq d, \end{aligned} \tag{15.8}$$

with the objective function $f : R^n \rightarrow R$, and the constraints defined by the two functions $g : R^n \rightarrow R^m$ and $h : R^n \rightarrow R^l$, and the vectors $c \in R^m$ and $d \in R^l$. If the objective function is linear, then any optimal solution x^* must be located on the boundary of the search space. But if the objective function is nonlinear, minima can exist both as boundary points and as interior points. Hence, for a linear objective function we seek minima on the boundary of the search space, whereas for a nonlinear objective function we must also search for minima inside the search space.

Lagrange multipliers

In the case we have only equality constraints in the constrained minimization problem (15.8), we can define a *Lagrangian* $L : R^{n+m} \to R$, which takes the form

$$L(x, \lambda) = f(x) - \lambda^T (g(x) - c),$$

for the *primal variable* $x \in R^n$ and the *dual variable* $\lambda \in R^m$. λ is also referred to as the *adjoint variable*, or the *Lagrangian multipliers*. By seeking the critical point of the Lagrangian, we obtain the necessary optimality conditions

$$\nabla_x L(x, \lambda) = \nabla f(x) - \lambda^T \nabla g(x) = 0, \tag{15.10}$$

$$\nabla_\lambda L(x, \lambda) = g(x) - c = 0, \tag{15.11}$$

$n + m$ equations from which we can determine the solution

$$(x, \lambda) \in R^{n+m}.$$

Optimal control

If a dynamical system is governed by a system of initial value problems in which there are free parameters, we can seek to formulate an optimal control strategy to modify these parameters over time to achieve a certain goal. Consider a dynamical system of the form

$$\dot{u}(t) = f(m(t), u(t), t), \quad u(0) = u_0, \quad t \in [0, T], \tag{15.13}$$

where $u(t) \in R^n$ is the state variable and $m(t) \in R^d$ is the control variable. For L a running cost and g a final cost, the optimal control problem is then to minimize a cost functional

$$J(m, u, T) = g(u(T)) + \int_0^T L(m(t), u(t), t)\, dt. \tag{15.14}$$

Optimal control

Pontryagin's principle states that an optimal control $m_{opt}(t)$ only exists provided that a certain primal-dual system is satisfied, which represents necessary optimality conditions in the context of dynamical systems. The construction of the optimality conditions is a generalization of Hamiltonian mechanics, by formulating a Hamiltonian

$$H(m, u, \lambda, t) = \lambda^T f(m, u, t) + L(m, u, t), \tag{15.15}$$

$$\begin{aligned} \dot{u} &= \frac{\partial H}{\partial \lambda} = f, \\ -\dot{\lambda}^T &= \frac{\partial H}{\partial u} = \lambda^T (\nabla_u f) + (\nabla_u L), \end{aligned}$$

with a final condition for the adjoint variable $\lambda(T) = \nabla_u g(u(T))$. These equations offer a method to compute an optimal solution, by minimization of the Hamiltonian with respect to the control variable $m(t)$.

Unsupervised learning/clustering

To measure the distance between data points we can use the Euclidian distance in the observation space R^d,

$$d(x_i, x_j) = \|x_i - x_j\|.$$

Alternatively, we can use *feature extraction*, where we introduce a Hilbert space H defined by a feature map $\phi : R^d \to H$, the feature space, in which we measure the distance between data points by the metric

$$d_H(x_i, x_j) = \|\phi(x_i) - \phi(x_j)\|_H.$$

Unsupervised learning/clustering

To compute the distance in the feature space we form a kernel

$$k(x_i, x_j) = (\phi(x_i), \phi(x_j))_H,$$

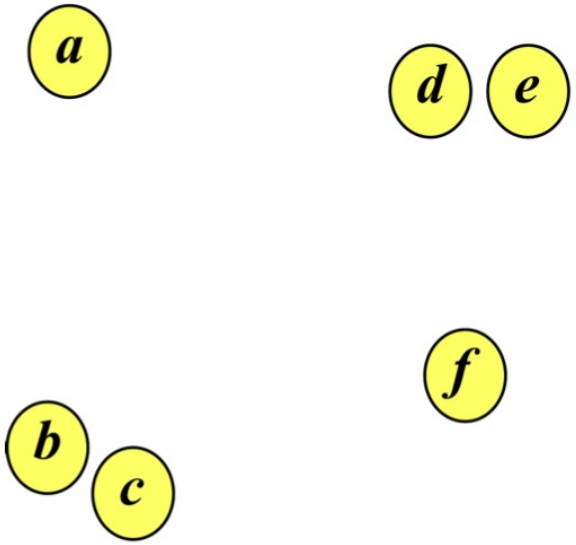
with $(\cdot, \cdot)_H$ the inner product in H, from which the distance can be computed as

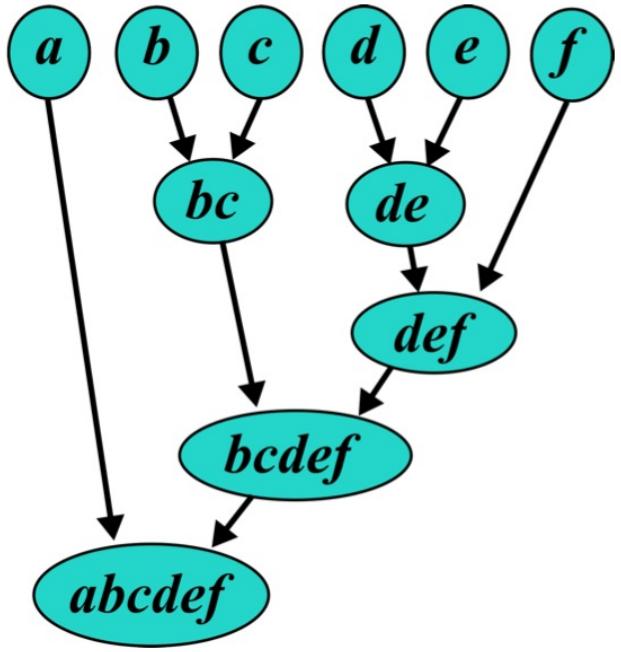
$$\|\phi(x_i) - \phi(x_j)\|_H = \sqrt{k(x_i, x_i) + k(x_j, x_j) - 2k(x_i, x_j)},$$

since

$$\|\phi(x_i) - \phi(x_j)\|_H^2 = (\phi(x_i), \phi(x_i))_H + (\phi(x_j), \phi(x_j))_H - 2(\phi(x_i), \phi(x_j))_H.$$

Unsupervised learning/clustering





Unsupervised learning/clustering

Related algorithms are based on the topological concept of a *connected set*, defined as a set C in which the open balls $B(x, \epsilon)$ of all elements $x \in C$ include at least one other element $y \in C$. In R^d the open ball is defined in terms of the metric as

$$B(x, \epsilon) = \{y \in R^d : d(x, y) < \epsilon\}.$$

Unsupervised learning/clustering

Instead of computing the pairwise distances between data points, we can assign a data point x_i to a cluster C_j based on its distance to the cluster mean, or center,

$$\bar{x}_j = \frac{1}{|C_j|} \sum_{i \in C_j} x_i,$$

with $|C_j|$ the number of data points x_i in the set C_j. To reduce the sensitivity to outliers, we can use the median of the cluster data points instead of the mean.

Example 16.3 (k-means). The *k-means* algorithm is initialized by k clusters C_j, each with a mean $\bar{x}_j \in R^d$. In every step the data points x_i are assigned to the cluster C_j for which the distance to the cluster mean $\bar{x}_j$ is the shortest, after which the cluster mean is recomputed for the updated clusters. The algorithm converges when the cluster means no longer change.

Supervised learning/classification

Once we have divided the data $x = \{x_1, ..., x_n\} \subset R^d$ into k clusters $\{C_j\}_{j=1}^k$, a natural question is what to do when new data points are added. Since we want to avoid to recompute the clustering algorithm each time a new data point is added, we instead develop *classification algorithms* that directly assign a new data point to the category, or class, corresponding to one of the clusters. One approach to do this is to partition the observation space R^d into k disjoint *decision regions* Ω_j separated by *decision boundaries*, with the property that

$$C_j \subset \Omega_j, \quad j = 1, ..., k.$$

Supervised learning/classification

Example 16.5. *Support vector machines* (SVM) are algorithms that seek to construct an optimal decision boundary in the form of a hyperplane that separates two decision regions Ω_1 and Ω_2. SVM is based on the principle to maximize the *margin*, the distance between the hyperplane and the closest data points of each cluster C_1 and C_2, the *support vectors*. Hence, only the small subset of support vectors determines the hyperplane, all other data points are unimportant for an SVM algorithm. A hyperplane H_0 in R^d is defined as the set of vectors $x \in R^d$ which satisfy the equation $w \cdot x - b = 0$, with the normal vector $w \in R^d$ and the scalar $b \in R$.

Supervised learning/classification

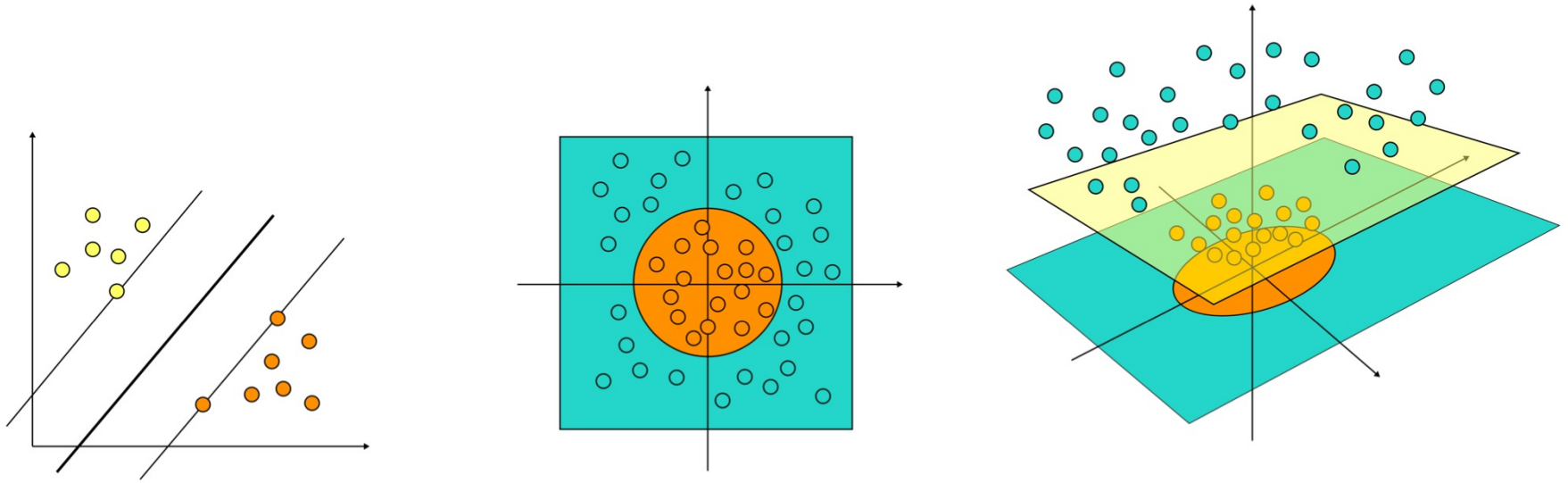


Figure 16.2. *Linear decision boundaries in R^2 computed by SVM, using the Euclidian distance in the data domain (left), and an illustration of how data points in the observation space R^2 are mapped to the feature space R^3 by the feature map $\phi : (y_1, y_2) \mapsto (y_1, y_2, y_1^2 + y_2^2)$, to allow for nonlinear separability of the data points inside and outside the circle (right).*

Statistical decision theory

We now approach the classification problem from a statistical point of view, where we introduce a discrete random variable S to model a set of source categories which can take N_s different values j, each with a *prior* probability mass function

$$P_S(j) = P(S = j), \quad \sum_{j=1}^{N_s} P_S(j) = 1.$$

The feature vector x which is extracted from data is assumed to be an observation of a random vector X, with a probability distribution that depends on the source category S, characterized by the conditional probability density function $f_{X|S}(x|j)$, the conditional density probability to observe $X = x$ given that $S = j$, for which

$$\int_{\Omega} f_{X|S}(x|j)\, dx = 1, \quad j = 1, ..., N_s,$$

with the integral over the full observation space Ω.

Statistical decision theory

The output from the classification algorithm, or classifier, is a decision determined by a discrete decision function $d(\cdot)$, where

$$d(x) \in \{1, ..., N_d\}.$$

If the task of the classifier is to identify the source category S, then $N_d = N_s$ and each decision represents the most likely source category $S = j$ under the observation $X = x$, which is the *posterior* probability mass function

$$P_{S|X}(j|x) = \frac{f_{X|S}(x|j)P_S(j)}{f_X(x)}$$

Statistical decision theory

To design a classifier we first construct a *loss matrix* $L = (l_{ij})$ of dimension $N_d \times N_s$, where each component $l_{ij} = L(d = i|S = j)$ represents the loss if the classifier chooses $d = i$ when the truth is $S = j$. We then define the *conditional expected loss*, or *conditional risk*, as the average loss made by the decision $d(x) = i$ based on observation x,

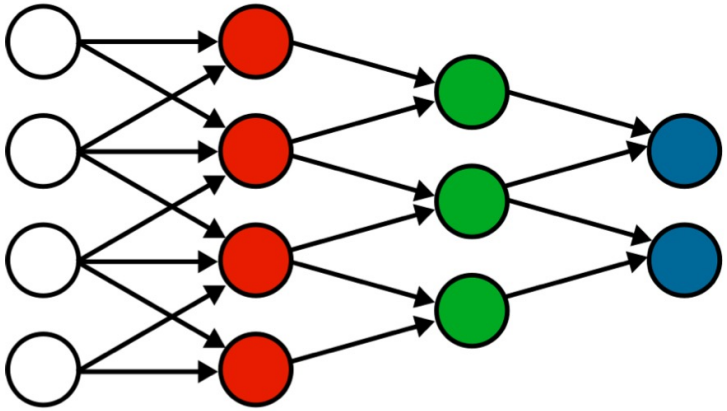
$$R(i|x) = \sum_{j=1}^{N_s} L_{ij} P_{S|X}(j|x),$$

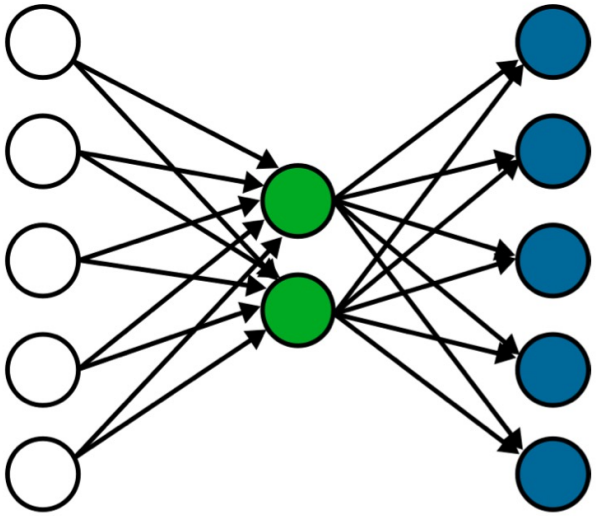
which leads us to *Bayes minimum-risk decision rule*,

$$d(x) = \arg\min_i R(i|x).$$

Deep neural networks

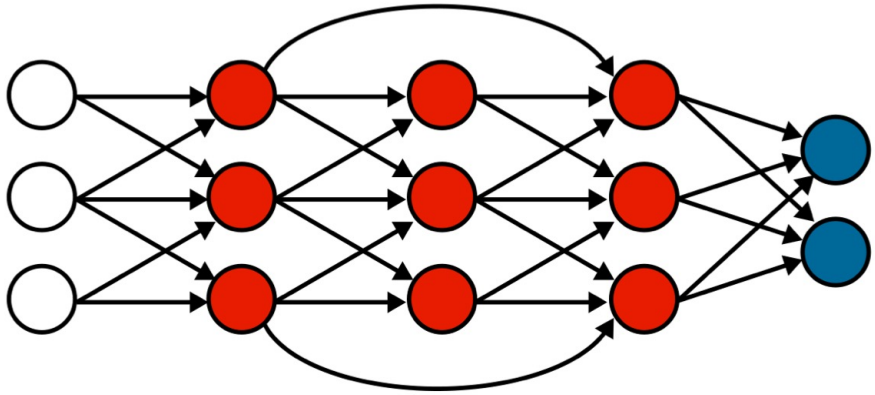
The number of hidden layers is the *depth* of the network, and the number of neurons in each layer is the *width* of that layer. A standard *feedforward* network architecture can be represented by a directed acyclic graph (DAG) where information is flowing in one direction from the input layer to the output layer, one layer at the time, whereas a *residual neural network* (ResNet) can use *skip connections* to bypass certain layers.





Deep neural networks

The number of hidden layers is the *depth* of the network, and the number of neurons in each layer is the *width* of that layer. A standard *feedforward* network architecture can be represented by a directed acyclic graph (DAG) where information is flowing in one direction from the input layer to the output layer, one layer at the time, whereas a *residual neural network* (ResNet) can use *skip connections* to bypass certain layers.



Forward propagation equation

Consider two adjacent layers n and $n-1$ in a feedforward neural network, with widths N_n and N_{n-1}, respectively. The input for layer n is the output from layer $n-1$, and hence we get the *forward propagation equation*

$$x^{(n)} = \phi \odot (W^{(n)} x^{(n-1)} + b^{(n)}),$$

with the state vectors and bias vectors $x^{(n)}, b^{(n)} \in R^{N_n}$ and $x^{(n-1)} \in R^{N_{n-1}}$, the weight matrix $W^{(n)} \in R^{N_n \times N_{n-1}}$, and the component-wise extension of the scalar activation function $\phi \odot (\cdot)$, or in component form,

$$x_i^{(n)} = \phi(w_{ij}^{(n)} x_j^{(n-1)} + b_i^{(n)}),$$

with $b_i^{(n)}$ the bias of neuron i in layer n, $w_{ij}^{(n)}$ the weight of the input state $x_j^{(n-1)}$, and $\phi(\cdot)$ the nonlinear activation function. In a residual neural network the forward propagation equation for layer n can also involve state vectors from other layers than $n-1$.

Activation functions

Neural networks are designed to include nonlinear transformations of the input data, and the nonlinearity is enforced through the activation function $\phi(\cdot)$. The classical activation function is the *perceptron*, which corresponds to the *Heaviside function*

$$H(x) = \begin{cases} 0, & x \leq 0, \\ 1, & x > 0. \end{cases}$$

But binary output have proven problematic in training of the network, and therefore regularized activation functions have been constructed, such as the *sigmoid function* (or *logistic function*),

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Activation functions

Another popular activation function is the *rectified linear unit* $ReLU$,

$$ReLU(x) = x^+ = \max(0, x),$$

or its smooth approximation the *softplus function*

$$sReLU(x) = \log(1 + e^x).$$

We note that the derivative of the ReLU function is the Heaviside function, and that the derivative of the softplus function is the sigmoid function.

Deep learning

For a DNN with L hidden layers, the feedforward network represents a function

$$g_w : R^{N_0} \rightarrow R^{N_{L+1}},$$

with N_0 and N_{L+1} the number of neurons in the input and output layers, respectively. The function is parameterized by

$$w = (W^{(1)}, ..., W^{(L+1)}, b^{(1)}, ..., b^{(L+1)}).$$

The input layer operates directly on the raw data $x^{(0)}$ and the output layer acts as a classifier, whereas in the hidden layers the extracted features are recursively processed.

Deep learning

Hence, a constrained minimization problem is formulated based on the cost function and an equality constraint given by the forward propagation equation,

$$\hat{w} = \arg\min_w C(\mathcal{D}^{train}; g_w). \tag{16.2}$$

Example 16.10. The cost function may take the form

$$C(\mathcal{D}^{(train)}; g_w) = \sum_{\alpha=1}^{N_\alpha} \frac{1}{2} \| x_\alpha^{(L+1)} - \hat{x}_\alpha \|^2, \tag{16.3}$$

with the training data set $(x_\alpha^{(0)}, \hat{x}_\alpha) \in \mathcal{D}^{(train)}$ of size N_α, and with the model output given by the forward propagation equation $x_\alpha^{(L+1)} = g_w(x_\alpha^{(0)})$. Here the training data is labelled, connecting input data $x_\alpha^{(0)}$ to output data $\hat{x}_\alpha$, which allows for a supervised learning algorithm.

Deep learning

Hence, a constrained minimization problem is formulated based on the cost function and an equality constraint given by the forward propagation equation,

$$\hat{w} = \arg\min_w C(\mathcal{D}^{train}; g_w). \tag{16.2}$$

Example 16.11. DNN can also be used for unsupervised learning without any labeled data, with only input data available $x_\alpha^{(0)} \in \mathcal{D}^{(train)}$. For instance, to train an autoencoder we can use the cost function (16.3) but with $\hat{x}_\alpha = x_\alpha^{(0)}$, so that the autoencoder seeks to reconstruct the input data after downsampling.

Deep learning

The in-sample error is defined as

$$E_{in} = C(\mathcal{D}^{(train)}, g_{\hat{w}}),$$

and the out-of-sample error

$$E_{out} = C(\mathcal{D}^{(test)}; g_{\hat{w}}).$$

If we also want to perform model selection with respect to E_{out}, we instead partition the data according to $\mathcal{D} = \mathcal{D}^{(train)} \cup \mathcal{D}^{(test)} \cup \mathcal{D}^{(validation)}$, where the validation data set is used to evaluate the performance of the selected model. It is important to be aware of the trade-off between model expressiveness and generalization, the *bias-variance trade-off*, where a model with many parameters but not enough data may run the risk of overfitting, so that performance for the training data is high but much worse for the test (or validation) data.

Training: backpropagation algorithm

To determine the weights and biases of a DNN, we can solve a constrained minimization problem of the type (16.2), with training data $\hat{x} \in R^{N_{L+1}}$ and cost function

$$C(x^{(L+1)}) = \frac{1}{2}\|x^{(L+1)} - \hat{x}\|^2 = \frac{1}{2}\sum_{i=1}^{N_{L+1}} |x_i^{(L+1)} - \hat{x}_i|^2.$$

Note that these updates are based on only one data vector $\hat{x}$, but the same methodology can be used if additional data vectors are added to the cost function. If $\hat{x}$ is chosen randomly from the training data set $\mathcal{D}^{(train)}$, the update formula corresponds to a stochastic gradient descent method.

Training: backpropagation algorithm

Under the constraint of the forward propagation equation we formulate a Lagrangian,

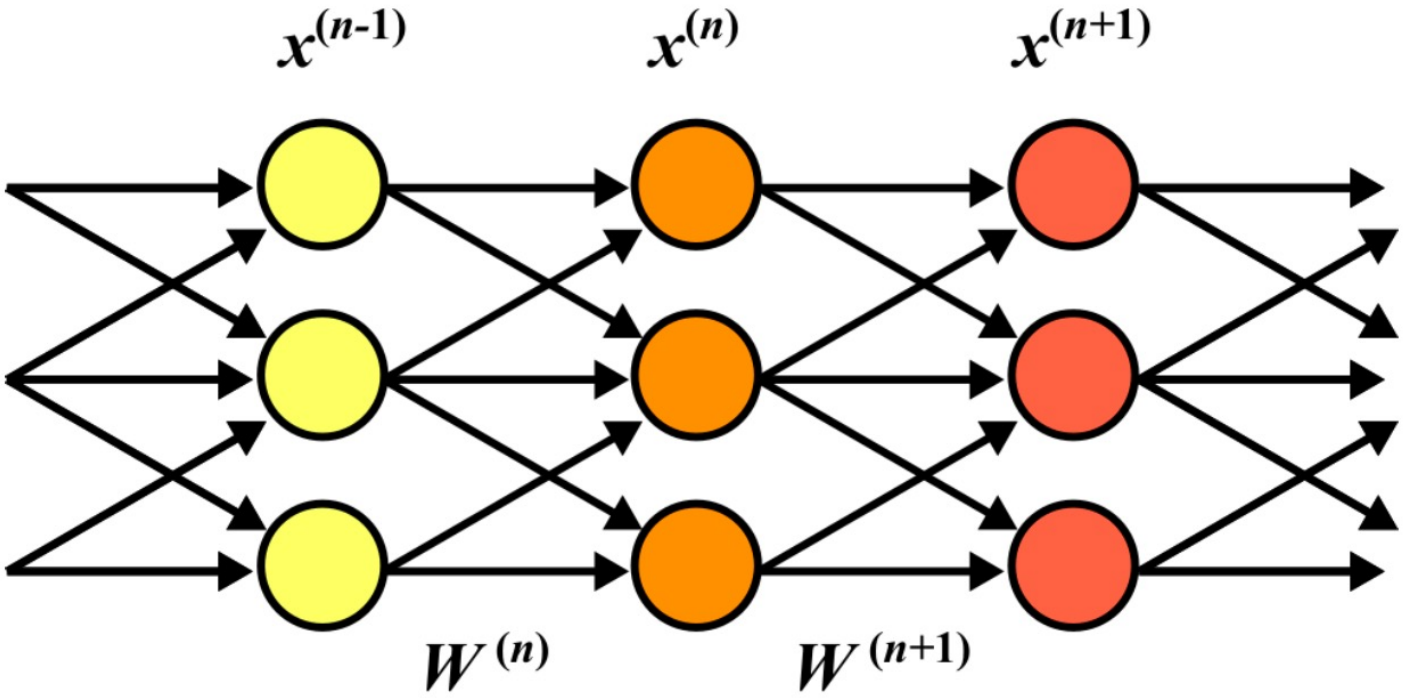
$$L(W, x, \lambda) = C(x^{(L+1)}) + \sum_{n=1}^{L+1} (\lambda^{(n)})^T (x^{(n)} - \phi \odot (W^{(n)} x^{(n-1)} + b^{(n)})),$$

with $W = (W^{(n)}, b^{(n)})$ the weights and biases, $x = (x^{(n)})$ the primal variables, and $\lambda = (\lambda^{(n)})$ the adjoint variables. The forward propagation equation corresponds to the following optimality condition

$$\frac{\partial L(W, x, \lambda)}{\partial \lambda^{(n)}} = 0 \quad \Rightarrow \quad x^{(n)} = \phi \odot (W^{(n)} x^{(n-1)} + b^{(n)}),$$

for $n = 1, .., L+1$, with the initial condition $x^{(0)}$.

Training: backpropagation algorithm



Training: backpropagation algorithm

Similarly, we obtain the backward propagation equation from the optimality condition

$$\frac{\partial L(W,x,\lambda)}{\partial x^{(n)}} = 0 \quad \Rightarrow \quad \lambda^{(n)} = (W^{(n+1)})^T \nabla\phi(y^{(n+1)})\lambda^{(n+1)},$$

for $n = 0, .., L$, with the final condition

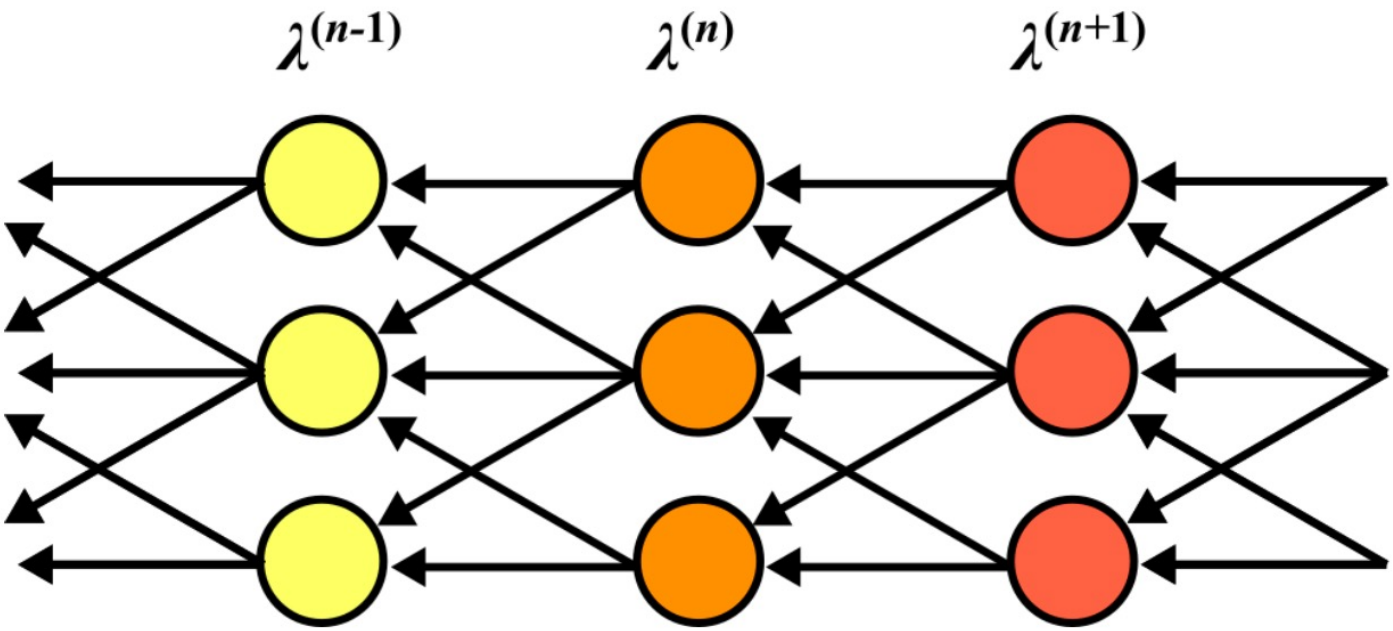
$$\lambda^{(L+1)} = (x^{(L+1)} - \hat{x}).$$

Here $\nabla\phi(y^{(n+1)})$ is a Jacobian matrix evaluated at

$$y^{(n+1)} = W^{(n+1)}x^{(n)} + b^{(n+1)}.$$

For a standard feedforward network architecture the Jacobian is a diagonal matrix since the activation function acts component-wise.

Training: backpropagation algorithm



Training: backpropagation algorithm

Example 16.12. For example, in the case of a sigmoid activation function $\phi(x) = (1+e^{-x})^{-1}$, the diagonal components of the Jacobian matrix are

$$(\nabla\phi(y^{(n+1)}))_{ii} = \phi(y_i^{(n+1)})(1-\phi(y_i^{(n+1)})) = x_i^{(n+1)}(1-x_i^{(n+1)}). \tag{16.4}$$

By the transformation $\Lambda^{(n)} = \nabla\phi(y^{(n)})\lambda^{(n)}$, the adjoint equation takes the form

$$\Lambda^{(n)} = \nabla\phi(y^{(n)})(W^{(n+1)})^T\Lambda^{(n+1)}, \tag{16.5}$$

for $n = 0, .., L$, with final condition

$$\Lambda^{(L+1)} = \nabla\phi(y^{(L+1)})(x^{(L+1)} - \hat{x}).$$

Training: backpropagation algorithm

The third optimality condition gives the Jacobian matrix with respect to the weights, which are the control variables,

$$\frac{\partial L(W,x,\lambda)}{\partial W^{(n)}} = \nabla\phi(y^n)\lambda^{(n)}(x^{(n-1)})^T = \Lambda^{(n)}(x^{(n-1)})^T = 0,$$

so that the Jacobian matrix takes the form

$$\Lambda^{(n)}(x^{(n-1)})^T,$$

which leads to a gradient descent method with an update map

$$W^{(n)} \to W^{(n)} + \Delta W^{(n)}$$

for the increment

$$\Delta W^{(n)} = -\alpha^{(n)}\Lambda^{(n)}(x^{(n-1)})^T,$$

where $\alpha^{(n)}$ is a step length.

Training: backpropagation algorithm

If the model parameter search space is too large in relation to the amount of available training data there is a risk of overfitting, where the model performs well on training data but generalizes poorly to new data. To reduce the risk for overfitting, the model parameter search space may be reduced by Tichonov regularization, or l^2 regularization, which penalizes the l^2 norm of the parameters,

$$C(x^{(L+1)}) = \frac{1}{2}\|x^{(L+1)} - \hat{x}\|^2 + \lambda\|W\|_2^2,$$

with a penalty parameter $\lambda \geq 0$. Similarly the l^1 norm $\|W\|_1$ can be penalized, referred to as l^1 *regularization*. A different type of regularization is based on normalization of the state variables in the hidden layers, typically subtracting a mean and dividing by a standard variance based on all the neurons in that layer or the neurons in a batch of training data, referred to as *batch normalization*. Alternatively, in a *weight normalization* strategy only the weights in the hidden layers are normalized, for example, by a suitable norm.

Generative adversarial networks

Apart from the classifier DNNs, we distinguish between *generative* and *adversarial* deep neural networks. A generative network learns to generate output data that belongs to a certain class C_{true}, whereas an adversarial network learns to determine if input data belongs to the class C_{true} or not. From an input data set, typically random, the generative network g_w^{gen} generates output data which we say belongs to the class C_{gen}.

Generative adversarial networks

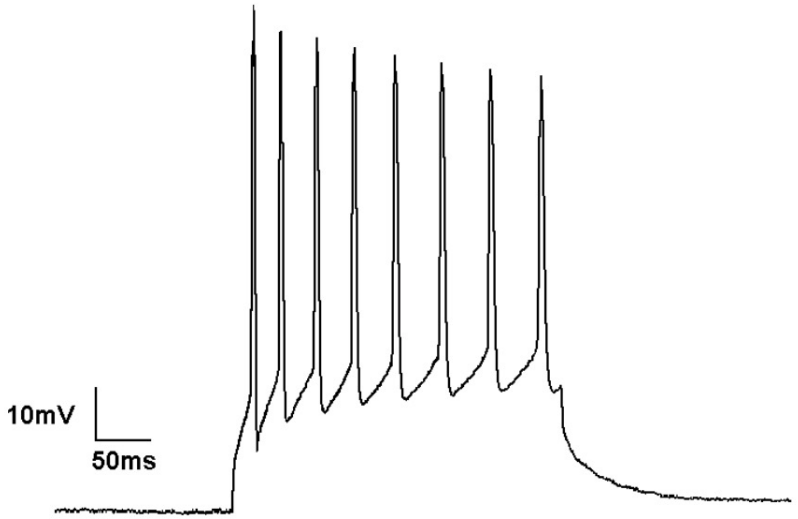
Spiking neural network

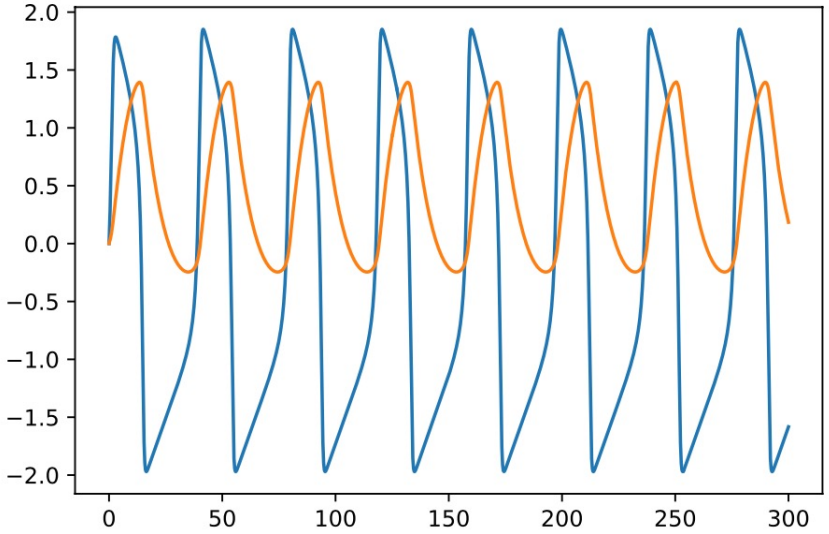
A *spiking neural network* (SNN) seeks to model information processing in the human brain, where the artificial neurons, or *spiking neurons*, are connected through *synaptic connections* in a network with adjustable weights. Hence, a spiking neural network can be understood as a dynamical system over a graph neural network. The dynamics of *action potentials*, or *spike trains*, is determined for each neuron by a system of ordinary differential equations.

Example 16.13. A famous model of action potentials is the *Hodgkin-Huxley model*, which rendered a Nobel Prize in 1963. The *FitzHugh-Nagumo model* is a simplification of the Hodgkin-Huxley model which still captures the spiking dynamics of an action potential $v(t)$,

$$\begin{aligned} \dot{v}(t) &= v(t) - \frac{v(t)^3}{3} - w(t) + I(t), \\ \dot{w}(t) &= \tau^{-1}(v(t) - a - bw(t)). \end{aligned}$$

Spiking neural network





Data-driven dynamical systems

A model of a dynamical system expresses change, often in terms of a differential equation which provides a recipe for how to construct a discrete approximation of the dynamical system by a time stepping algorithm. The essence of a time stepping algorithm is that the present state x_t follows from the previous state x_{t-1} by a transition map

$$A_t : x_{t-1} \mapsto x_t.$$

If the map is time-independent $A_t = A$, the dynamical system is said to be time-invariant. Hence, the elements of a time stepping algorithm for a time-invariant dynamical system is an initial state x_0 and a transition map A, which can be derived from a differential equation.

Data-driven dynamical systems

Example 16.15. In the case of the initial value problem,

$$\dot{u}(t) = \sin(u(t)) - 5,$$

explicit Euler time stepping with time step Δt leads to a time-invariant dynamical system, with the transition map $A : U_{n-1} \mapsto U_n$, defined by

$$U_n = A(U_{n-1}) = U_{n-1} + \Delta t(\sin(U_{n-1}) - 5).$$

Data-driven dynamical systems

If we do not have access to a differential equation which describes the dynamical system, but only data from measurements, we can take a data-driven approach to the problem where we seek to learn the transition map A from data. To implement a time stepping algorithm for x_t we need two basic data structures. Apart from the transition map A, we need to store the previous state x_{t-1}, which functions as a memory in the algorithm. Therefore, to adapt deep learning algorithms to model dynamical systems we need to equip the model with a memory, or internal state, similar to a time stepping algorithm. One mechanism to add a memory to a DNN is to depart from a standard feedforward network architecture by adding loops, referred to as a *recurrent neural network* (RNN).

Data-driven dynamical systems

Loops keep previous states in memory so that we can express a hidden state $h_t \in R^n$ as

$$h_t = h_{t-1} + \phi(W_t h_{t-1} + b_t),$$

together with an initial state $h_0 = x_0$, which takes the form of an explicit Euler time stepping algorithm for an ordinary differential equation

$$\dot{h} = f(h), \quad h(0) = x_0.$$

Here

$$f(h) = \lim_{\Delta t \to 0} \frac{1}{\Delta t} \phi(Wh + b),$$

where $\Delta t = 1/K$, with K the number of unrolled loops.

Stability of DNN

Stability concerns the growth of perturbations in a system, for a deep neural network this can be perturbations in initial data, in the weights or biases. The effect of perturbations can be measured by a cost function

$$C(x^{(L+1)}, \hat{x}^{(L+1)}) = \frac{1}{2}\|x^{(L+1)} - \hat{x}^{(L+1)}\|^2,$$

where the unperturbed model output vector is given by a forward propagation equation

$$x^{(L+1)} = g_w(x^{(0)}),$$

and the perturbed model output vector by the perturbed forward propagation equation

$$\hat{x}^{(L+1)} = g_{\hat{w}}(\hat{x}^{(0)}),$$

with the perturbed initial data $\hat{x}^{(0)} = x^{(0)} + \epsilon_0$, and the perturbed model parameters $\hat{w} = w + \epsilon_w$.

Stability of DNN

Stability related to perturbations in the model parameters measures how robust the network is with respect to training, and stability related to perturbations in initial data characterizes the accuracy of the classification. If a small perturbation in the input vector can lead to a large effect in the output vector, the result cannot be trusted.

Perturbations that exhibit a strong growth are referred to as *adversarial perturbations*. Specifically, a class of adversarial perturbations have been identified that show strong growth over a wide range of different deep neural networks, so called *universal adversarial perturbations*.

Hidden Markov models

In a stochastic context, the analogy of a dynamical system is a Markov process, a stochastic process where each state is statistically dependent on the previous state, but independent of all other states. With access to a stochastic differential equation that describes the dynamical system, we can simulate the system by a Monte Carlo method where we construct an ensemble of paths for which we can compute statistics. If we have access to data, we can model the dynamical system by a *hidden Markov model* (HMM) defined in terms of the parameters

$$H = \{q, A, B\}.$$

The elements of a hidden Markov model is (i) a discrete Markov chain $\{q, A\}$, represented by a vector of *initial state probability distributions* q and a *transition probability matrix* A, and (ii) a vector of *output probability distributions* B that define the conditional probabilities for each observation given the underlying state of the Markov chain.

Black, white and grey box methods

The data-driven methods we have discussed are sometimes referred to as *black box methods* in the field of system identification. These can be contrasted with theory-driven *white box methods*, for example, in the form of differential equations. We may also choose a hybrid approach, or *grey box method*, in which we combine the benefits of established theory and access to data.

Black, white and grey box methods

Example 16.17. Weather forecasting is a prime example of data assimilation, where the Navier-Stokes equations which models the conservation laws of the atmosphere are combined with data, for example, the wind speed, pressure, temperature and humidity.

Example 16.18. To model the dispersion of air pollution in a city we can use a grey box method, where the passive transport of the pollutants

$$u = (u_1, ..., u_n)^T$$

by the wind $\beta(x, t)$ is described by the left hand side of the differential equation

$$\dot{u} + \beta \cdot \nabla u = f(u),$$

but where the chemical reactions $f(u)$ are described by a deep neural network, which is trained using measurement data.

Black, white and grey box methods

Example 16.19. For an example of so called physics-informed neural networks, consider a deep neural network for which the input layer consists of the spatial and temporal coordinates (x, t), and the output layer consists of $u(x, t)$, the solution to a partial differential equation in residual form,

$$R(u(x, t)) = 0.$$

When we train the network with the training data

$$\{\hat{x}_i, \hat{t}_i, \hat{u}_i(\hat{x}_i, \hat{t}_i)\}_{i=1}^{N},$$

and use the norm of the residual as cost function

$$\sum_{i=1}^{N} \|R(\hat{u}_i(\hat{x}_i, \hat{t}_i))\|^2,$$

we hope that the network can be trained to generate approximations of $u(x, t)$ for coordinates (x, t) outside the training data. This is a general construction, for any partial differential equation and any domain, but to what degree this approach could be reliable and efficient is still under investigation.