

Supervised Learning

Alexandros Iosifidis

diiP Summer School

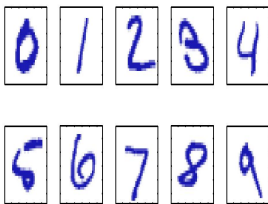
June 11, 2024

Machine Learning (ML)

- The main idea is to “give machines access to data and let them learn and improve for themselves”
- Relevant to problems which are hard to describe or model e.g., face detection
- Solution: data-driven approach as opposed to model-based approach



Face detection



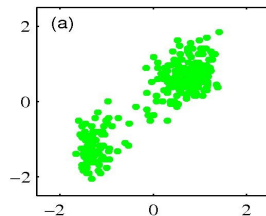
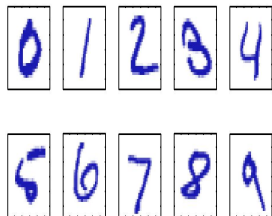
Text recognition



Movie recommendation

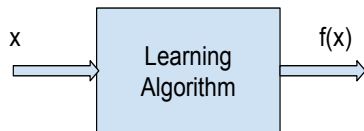
Goals of Learning

- The two main goals of learning are:
 - ① **Data Understanding**: means extracting useful features or patterns from data to improve our understanding of the data and underlying process(es) e.g., clustering, dimensionality reduction
 - ② **Prediction**: making important decisions about system of interest e.g., hand-written digit recognition; wind power forecasting (on new/unseen data during training)
- **Challenges**: data may be complex/nonstationary, multi-modal, high-dimensional, noisy and depending on the application may be prohibitively large or inadequately small.



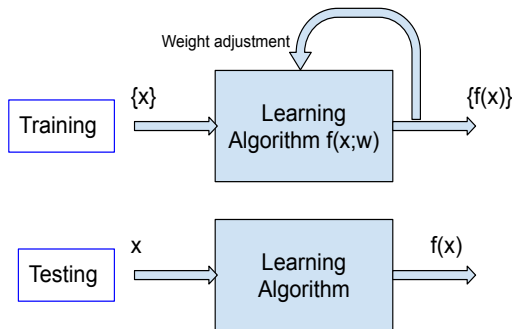
Prediction: Classification vs Regression

- **Classification:** problems deal with assigning input data to a specified category. Here, the output $f(x)$ belong to a discrete set of values or labels. Examples are
 - Classifying email as spam or not
 - Classification of hand written digit
- **Regression:** are prediction problems in which the output $f(x)$ corresponds to a continuous range of values (continuous variable) e.g., wind power prediction.



Supervised Learning

- Labeled training data includes input along with corresponding correct output
- The training data is used to teach model to yield the desired outputs



- Input x may be raw input or relevant features from input e.g., scaled and translated images of digits; average image intensity over rectangular regions of images for face recognition
- **Examples:**
 1. Classification problems (email spam detection, hand-written digit classification, delivering content that matches users interests in social media, streaming and online shopping platforms)
 2. Regression Problems (stock index price prediction; wind power forecasting)
- **Algorithms:**
 1. Linear Regression, Logistic Regression, Perceptron
 2. Neural networks (Multilayer Perceptron, Convolutional NN, Deep NN, Recurrent NN, Transformers)

Three approaches to classification

- 1 Finding a function $y(\mathbf{x})$ called **discriminant function** which maps a new input \mathbf{x}_* onto a class label.
- 2 **Generative models:** Determining **class-conditional densities** $p(\mathbf{x}|\mathcal{C}_k)$ or joint distributions $p(\mathbf{x}, \mathcal{C}_k)$, followed by the estimation of posterior densities:

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})} = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_j p(\mathbf{x}|\mathcal{C}_j)p(\mathcal{C}_j)}$$

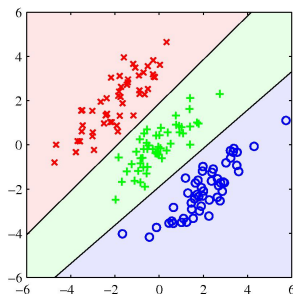
Finally, decision theory is used to determine class membership for new \mathbf{x} .

- 3 **Discriminative models:** Obtaining **posterior class probabilities** $p(\mathcal{C}_k|\mathbf{x})$ directly, followed by discrimination.

Linear classification

The **goal** of classification is to assign an input vector \mathbf{x} to one of the K classes \mathcal{C}_k , $k = 1, \dots, K$:

- The input space is divided into K **decision regions**, separated by **decision boundaries** or decision surfaces
- **Linear models**: are those for which decision boundaries are **linear functions** of \mathbf{x} i.e., $\mathbf{w}^T \mathbf{x} + w_0$



Decision boundaries for a linearly separable data set

Discriminant Functions: Two classes

Linear discriminant function:

$$y(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$$

\mathbf{w} is called *weight vector* and w_0 is called *bias* ($-w_0$ is called *threshold*).

Classification rule:

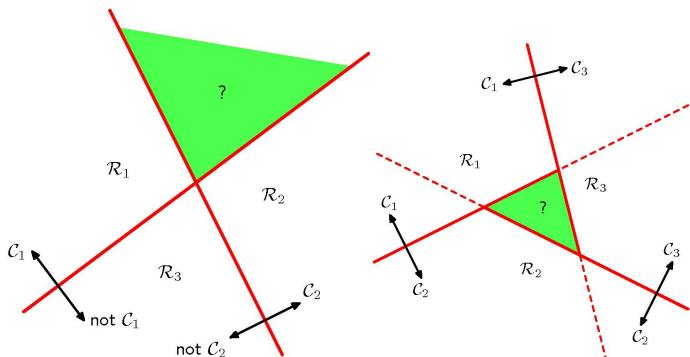
- Assign \mathbf{x} to class \mathcal{C}_1 if $y(\mathbf{x}) \geq 0$
- Assign \mathbf{x} to class \mathcal{C}_2 if $y(\mathbf{x}) < 0$

The **decision boundary** at $y(\mathbf{x}) = 0$, which corresponds to a $(D - 1)$ -dimensional hyperplane in \mathbb{R}^D .

Discriminant Functions: $K > 2$ classes

We can extend binary discriminant functions to K -class discriminant functions using two schemes:

- **One-versus-rest**: Use $K - 1$ binary discriminant functions, each of which separating points in \mathcal{C}_k and not in that class
- **One-versus-one**: Use $K(K - 1)/2$ binary discriminant functions, one for every possible pair of classes $\mathcal{C}_k, \mathcal{C}_{j \neq k}$.



Discriminant Functions: $K > 2$ classes

Single K class Discriminant: comprises K linear functions of the form:

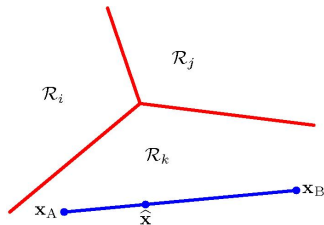
$$y_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + w_{k0}$$

The classification rule:

assigning \mathbf{x} to class \mathcal{C}_k if $y_k(\mathbf{x}) > y_j(\mathbf{x})$ for all $j \neq k$.

The decision boundary between \mathcal{C}_k and \mathcal{C}_j is given by $y_k(\mathbf{x}) = y_j(\mathbf{x})$, corresponding to

$$(\mathbf{w}_k - \mathbf{w}_j)^T \mathbf{x} + (w_{k0} - w_{j0}) = 0.$$



Least Squares for classification

Each class \mathcal{C}_k , $k = 1, \dots, K$ is described by:

$$y_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + w_{k0} = \tilde{\mathbf{w}}_k^T \tilde{\mathbf{x}}$$

where $\tilde{\mathbf{w}}_k = [w_{0k}, \mathbf{w}_k^T]^T$, and $\tilde{\mathbf{x}} = [1, \mathbf{x}^T]^T$.

We can group all K outputs together:

$$\mathbf{y}(\mathbf{x}) = \tilde{\mathbf{W}}^T \tilde{\mathbf{x}}$$

where $\tilde{\mathbf{W}} = [\tilde{\mathbf{w}}_1, \dots, \tilde{\mathbf{w}}_K]$.

The **classification rule** is:

assign \mathbf{x} to class \mathcal{C}_k if $y_k(\mathbf{x}) > y_j(\mathbf{x})$ for all $j \neq k$.

Least Squares for classification

Problem: Given a training set $\{\mathbf{x}_n, \mathbf{t}_n\}$ for $n = 1, \dots, N$, we want to estimate the parameters $\tilde{\mathbf{W}}$ of the regression model.

We use the 1-of- K binary coding scheme for \mathbf{t} . Denoting

$$\tilde{\mathbf{X}} = \begin{bmatrix} \tilde{\mathbf{x}}_1^T \\ \tilde{\mathbf{x}}_2^T \\ \vdots \\ \tilde{\mathbf{x}}_N^T \end{bmatrix} \text{ and } \tilde{\mathbf{T}} = \begin{bmatrix} \tilde{\mathbf{t}}_1^T \\ \tilde{\mathbf{t}}_2^T \\ \vdots \\ \tilde{\mathbf{t}}_N^T \end{bmatrix} \quad (1)$$

Cost function: The sum-of-squares error for all training data points is

$$E_D(\tilde{\mathbf{W}}) = \frac{1}{2} \text{Tr}\{(\tilde{\mathbf{X}}\tilde{\mathbf{W}} - \tilde{\mathbf{T}})^T(\tilde{\mathbf{X}}\tilde{\mathbf{W}} - \tilde{\mathbf{T}})\}$$

Least Squares for classification

By **minimizing** the above cost function w.r.t \tilde{W} , we get

$$\tilde{W} = \tilde{X}^\dagger T$$

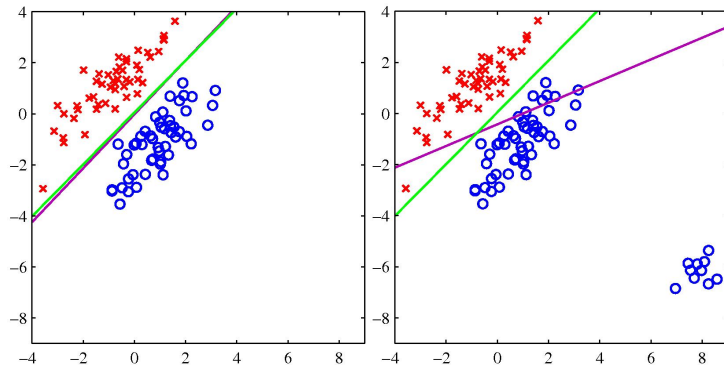
where \tilde{X}^\dagger is the pseudo-inverse of the matrix and is given by

$$\tilde{X}^\dagger = (\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T$$

How is this different from the least squares solution for the regression problem?

Least Squares for classification

Least squares-based classification is not robust to outliers.



Magenta line corresponds to least squares-based regression and green line corresponds to *logistic regression*.

The Perceptron algorithm

Given an input vector \mathbf{x} , the **Perceptron algorithm**:

- uses a fixed nonlinear transformation $\phi(\mathbf{x})$ (also including $\phi_0(\mathbf{x}) = 1$)
- uses a generalized linear model:

$$y(\mathbf{x}) = f(\mathbf{w}^T \phi(\mathbf{x}))$$

where:

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0. \end{cases}$$

is the **nonlinear activation** function.

- **Target values:** $t = +1$ for C_1 and $t = -1$ for C_2

Goal: Finding \mathbf{w} that is optimal for classification (in some sense).

The Perceptron algorithm

We want a \mathbf{w} such that:

- for all $\mathbf{x}_n \in \mathcal{C}_1$ we have $\mathbf{w}^T \phi(\mathbf{x}_n) > 0$
- for all $\mathbf{x}_n \in \mathcal{C}_2$ we have $\mathbf{w}^T \phi(\mathbf{x}_n) < 0$

Using $t_n \in \{-1, +1\}$ we can unify the two cases to: $\mathbf{w}^T \phi(\mathbf{x}_n) t_n > 0$.

The Perceptron criterion:

- assigns zero error for any correctly classified data point
- assigns an error equal to $-\mathbf{w}^T \phi(\mathbf{x}_n) t_n$ to \mathbf{x}_n if it is misclassified.

$$E_P(\mathbf{w}) = - \sum_{n \in \mathcal{M}} \mathbf{w}^T \phi(\mathbf{x}_n) t_n$$

where \mathcal{M} is the set of misclassified data points.

The Perceptron algorithm

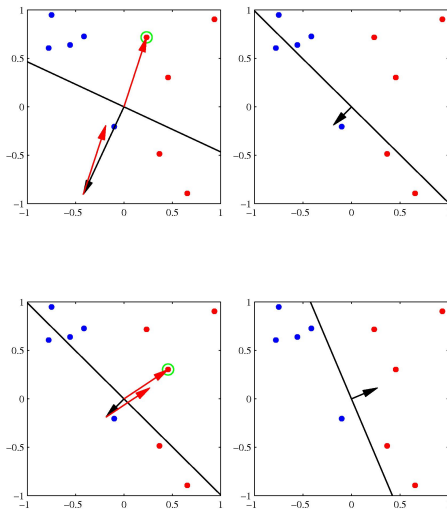
- 1 Randomly initialize \mathbf{w}^0
- 2 Iterate (until convergence)
 - 1 shuffle the training vectors \mathbf{x}_n , $n = 1, \dots, N$
 - 2 set $E_P(\mathbf{w}) = 0$
 - 3 iterate through the training vectors \mathbf{x}_τ
 - 4 if \mathbf{x}_τ is misclassified:
 - Compute the gradient $\nabla E_P(\mathbf{w}) = -\phi(\mathbf{x}_\tau)t_n$
 - Update the weight vector

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta \nabla E_P(\mathbf{w}) = \mathbf{w}^{(\tau-1)} + \eta \phi(\mathbf{x}_\tau)t_n$$

where $\eta > 0$ is a *learning rate* parameter.

Limitations: Applicable only for linearly separable data and for $K = 2$ classes

The Perceptron algorithm



Probabilistic Generative Models

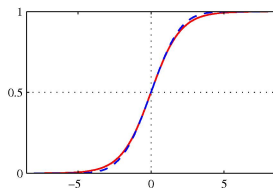
Consider a two-class problem; the posterior probability for \mathcal{C}_1 is

$$\begin{aligned} p(\mathcal{C}_1|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1) + p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} \\ &= \frac{1}{1 + \exp(-\alpha)} = \sigma(\alpha) \end{aligned}$$

where:

$$\alpha = \ln \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)}$$

and $\sigma(\alpha)$ is the *logistic sigmoid* function.



Logistic sigmoid function

Properties:

- **squashing function**: maps the whole real axis into a finite interval.
- $\sigma(-\alpha) = 1 - \sigma(\alpha)$
- $d\sigma/d\alpha = \sigma(1 - \sigma)$
- Inverse (known as *logit function*): $\alpha = \ln\left(\frac{\sigma}{1-\sigma}\right)$
- α represents the **log of the ratio of conditional probabilities for the two classes**:

$$\alpha(\mathbf{x}) = \ln\left(\frac{p(C_1|\mathbf{x})}{p(C_2|\mathbf{x})}\right)$$

also known as the *log odds*.

Logistic Regression

- We define the **error function** as the negative log-likelihood, which is called *cross-entropy* error function (suitable for minimization):

$$E(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{w}) = -\sum_{n=1}^N \left(t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \right)$$

where $y_n = \sigma(\alpha_n)$ and $\alpha_n = \mathbf{w}^T \mathbf{x}_n$.

- We optimize the parameters \mathbf{w} by applying **stochastic gradient descent**. **But why not obtain \mathbf{w} directly as in LS?**
- The gradient of error function w.r.t \mathbf{w} is

$$\nabla E(\mathbf{w}) = \sum_{n=1}^N (y_n - t_n) \mathbf{x}_n$$

Iterative Reweighted Least Squares*

Reminder: Newton-Raphson method defines the update rule:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \mathbf{H}^{-1} \nabla E(\mathbf{w}^{(\tau)})$$

where \mathbf{H} is the *Hessian matrix* having elements $H_{ij} = \frac{\partial^2 E_n(\mathbf{w})}{\partial w_i \partial w_j}$.

Using the cross-entropy error function:

$$\begin{aligned} \nabla E(\mathbf{w}) &= \sum_{n=1}^N (y_n - t_n) \mathbf{x}_n = \mathbf{X}^T (\mathbf{y} - \mathbf{t}) \\ \mathbf{H} = \nabla \nabla E(\mathbf{w}) &= \sum_{n=1}^N y_n (1 - y_n) \mathbf{x}_n \mathbf{x}_n^T = \mathbf{X}^T \mathbf{R} \mathbf{X}, \end{aligned}$$

where $\mathbf{R} = \text{diag}(y_n(1 - y_n))$ is a $N \times N$ square diagonal depending on \mathbf{w} .

Iterative Reweighted Least Squares*

Using the property $0 < y_n < 1$:

- \mathbf{H} is positive definite: for any \mathbf{u} the value $\mathbf{u}^T \mathbf{H} \mathbf{u} > 0$
- the error function $E(\mathbf{w})$ is a *convex* function of \mathbf{w}
- the error function has a unique (global) minimum.

The **iterative update rule** (due to dependence of \mathbf{R} on \mathbf{w}) is:

$$\begin{aligned}\mathbf{w}^{(\tau+1)} &= \mathbf{w}^{(\tau)} - (\mathbf{X}^T \mathbf{R} \mathbf{X}^{-1} \mathbf{X}^T (\mathbf{y} - \mathbf{t})) \\ &= (\mathbf{X}^T \mathbf{R} \mathbf{X})^{-1} (\mathbf{X}^T \mathbf{R} \mathbf{X} \mathbf{w}^{(\tau)} - \mathbf{X}^T (\mathbf{y} - \mathbf{t})) \rightarrow\end{aligned}$$

$$\mathbf{w}^{(\tau+1)} = (\mathbf{X}^T \mathbf{R} \mathbf{X})^{-1} \mathbf{X}^T \mathbf{R} \mathbf{z}$$

where $\mathbf{z} \in \mathbb{R}^N$:

$$\mathbf{z} = \mathbf{X} \mathbf{w}^{(\tau)} - \mathbf{R}^{-1} (\mathbf{y} - \mathbf{t}).$$

Softmax function

For $K > 2$ we obtain the **normalized exponential**:

$$p(C_k|\mathbf{x}) = \frac{p(\mathbf{x}|C_k)p(C_k)}{\sum_{j=1}^K p(\mathbf{x}|C_j)p(C_j)} = \frac{\exp(\alpha_k)}{\sum_{j=1}^K \exp(\alpha_j)}$$

where:

$$\alpha_k = \ln p(\mathbf{x}|C_k)p(C_k)$$

The normalized exponential is also called **softmax** function:

- when $\alpha_k \gg \alpha_j$ for all $j \neq k$, then $p(C_k|\mathbf{x}) \simeq 1$ and $p(C_j|\mathbf{x}) \simeq 0$

Multiclass logistic regression

Using 1-of- K coding for the target vectors \mathbf{t}_n forming $\mathbf{T} \in \mathbb{R}^{N \times K}$, the **likelihood function** is:

$$p(\mathbf{T} | \mathbf{w}_1, \dots, \mathbf{w}_K) = \prod_{n=1}^N \prod_{k=1}^K p(C_k | \mathbf{x}_N)^{t_{nk}} = \prod_{n=1}^N \prod_{k=1}^K y_{nk}^{t_{nk}}$$

with $y_{nk} = y_k(\mathbf{x}_n)$, and t_{nk} is the $\{n, k\}$ -th element of \mathbf{T} .

The negative log-likelihood, which is called **cross-entropy** error function, (suitable for minimization) is:

$$E(\mathbf{w}_1, \dots, \mathbf{w}_K) = -\ln p(\mathbf{T} | \mathbf{w}_1, \dots, \mathbf{w}_K) = -\sum_{n=1}^N \sum_{k=1}^K t_{nk} \ln y_{nk}$$

Multiclass logistic regression*

Derivatives of $E(\mathbf{w})$:

$$\begin{aligned}\nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \dots, \mathbf{w}_K) &= \sum_{n=1}^N (y_{nj} - t_{nj}) \mathbf{x}_n \\ \nabla_{\mathbf{w}_k} \nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \dots, \mathbf{w}_K) &= - \sum_{n=1}^N y_{nk} (I_{kj} - y_{nj}) \mathbf{x}_n \mathbf{x}_n^T.\end{aligned}$$

We can use $\nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \dots, \mathbf{w}_K)$ for applying SGD-based optimization, or both $\nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \dots, \mathbf{w}_K)$ and $\nabla_{\mathbf{w}_k} \nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \dots, \mathbf{w}_K)$ to apply IRLS-based optimization.

The artificial neuron

The basic building block of a neural network is called *neuron*:

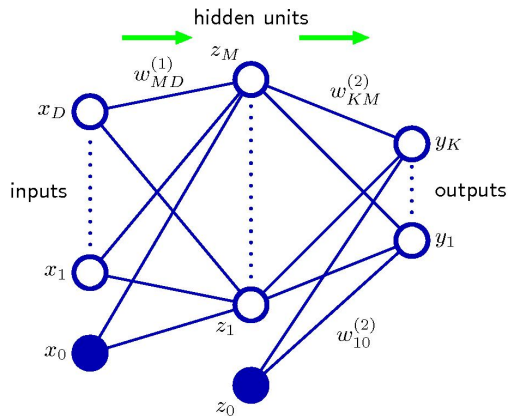
- It receives as input a vector, e.g. $\mathbf{x} \in \mathbb{R}^D$ and applies the following transformation:

$$\begin{aligned}\alpha &= \sum_{i=1}^D w_i x_i + w_0 = \mathbf{w}^T \mathbf{x} + w_0 \\ z &= h(\alpha)\end{aligned}$$

where:

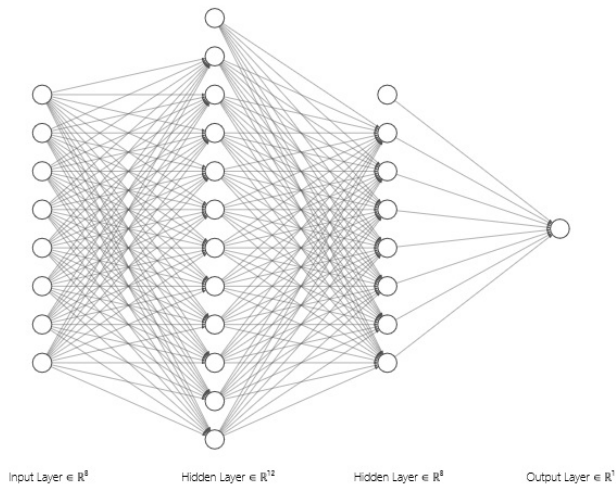
- $\{\mathbf{w}, w_0\}$ are the parameters of the neuron
- \mathbf{w} is called *weight* and w_0 is called *bias*
- α is known as the *activation*
- $h(\cdot)$ is a nonlinear *activation function*
- $h(\cdot)$ can take many forms, depending on the position of the neuron in the neural network and the problem at hand

A two-layer feed-forward neural network

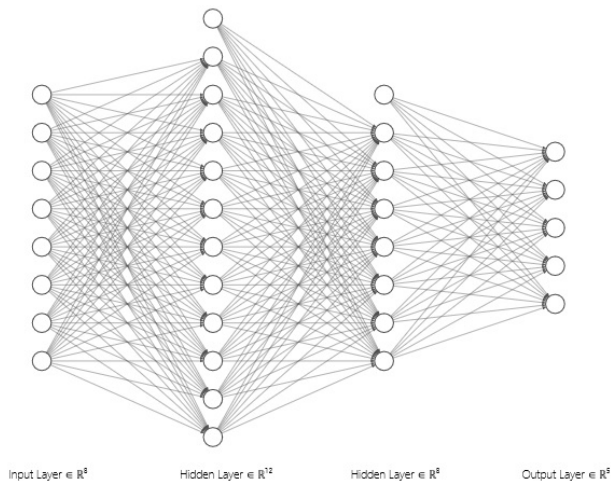


$$y_k(\mathbf{x}, \mathbf{w}) = \sigma \left(\sum_{j=1}^M W_{kj}^{(2)} h \left(\sum_{i=1}^D W_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right)$$

Multi-layer feed-forward neural network



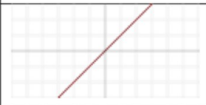
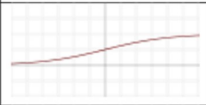
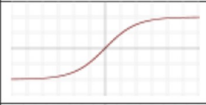
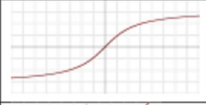

Multi-layer feed-forward neural network



Activation functions

Name	Equation	Derivative
Identity	$f(x) = x$	$f'(x) = 1$
Logistic	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
HyperbTan	$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan	$f(x) = \tanh^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
ReLU	$f(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ 1, & \text{if } x > 0 \end{cases}$
Softmax	$f_i(\mathbf{x}) = \frac{e^{x_k}}{\sum_{l=1}^K e^{x_l}}, k = 1, \dots, K$	$\frac{\partial f_i(\mathbf{x})}{\partial x_j} = f_i(\mathbf{x})(\delta_{ij} - f_j(\mathbf{x}))$

Activation functions

Name	Plot
Identity	
Logistic	
HypTan	
ArcTan	
ReLU	

A two-layer feed-forward neural network

If we 'absorb' the bias parameters in the corresponding weight vectors using additional dimensions (having a value equal to 1) on the input and hidden-layer output vectors:

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma \left(\sum_{j=0}^M W_{kj}^{(2)} h \left(\sum_{i=0}^D W_{ji}^{(1)} x_i \right) \right) = \sigma \left(\mathbf{W}^{(2)T} \underbrace{h(\mathbf{W}^{(1)T} \tilde{\mathbf{x}})}_{\tilde{\mathbf{z}}} \right)$$

where $\tilde{\mathbf{z}} \in \mathbb{R}^{M+1}$ and:

$$\mathbf{W}^{(1)} = [\mathbf{w}_1^{(1)}, \dots, \mathbf{w}_M^{(1)}] \in \mathbb{R}^{(D+1) \times M}$$

$$\mathbf{W}^{(2)} = [\mathbf{w}_1^{(2)}, \dots, \mathbf{w}_K^{(2)}] \in \mathbb{R}^{(M+1) \times K}$$

Note that the activation functions are applied *element-wise* (on each dimension)

Multilayer Perceptron

The above neural network is called *Multilayer Perceptron* (or MLP):

- an important property is that the activation functions of all neurons are differentiable w.r.t. their parameters

The use of nonlinear activation functions is crucial:

- If we use linear activation functions in the above two-layer neural network:

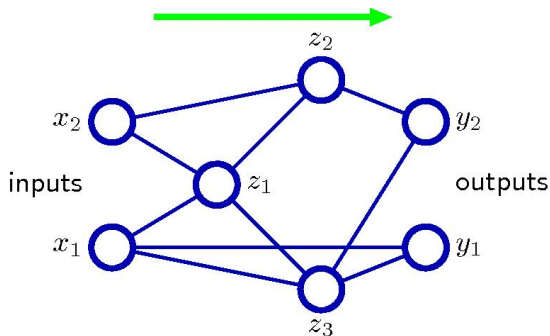
$$y_k(\mathbf{x}, \mathbf{w}) = \mathbf{W}^{(2)T} \mathbf{W}^{(1)T} \tilde{\mathbf{x}} = \mathcal{W}^T \tilde{\mathbf{x}}$$

where $\mathcal{W} = \mathbf{W}^{(2)T} \mathbf{W}^{(1)T} \in \mathbb{R}^{(D+1) \times K}$.

- Thus, any neural network with more than one layers and linear activation functions correspond to an one-layer neural network.

Skip connections

The connections in the network do not need to be restricted between neurons of successive layers (however they need to be feed-forward):



Neural Network Training: Regression*

Given a set of data points \mathbf{x}_n , $n = 1, \dots, N$, the corresponding target values t_n and including all the weights of the neural network in a parameter \mathbf{w} :

- we assume that t follows a Gaussian distribution:

$$p(t|\mathbf{x}, \mathbf{w}) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1})$$

where β is the precision (inverse variance) of the distribution.

- The likelihood function is (we use linear output neurons):

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^N p(t_n|\mathbf{x}_n, \mathbf{w}, \beta)$$

- The error function is the negative log-likelihood (discarding the terms not depending on \mathbf{w} and scaling factors):

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \left(y(\mathbf{x}_n, \mathbf{w}) - t_n \right)^2$$

Minimizing $E(\mathbf{w})$ will lead to \mathbf{w}_{ML} :

- Due to the use of nonlinear activation functions for the hidden neurons, \mathbf{w}_{ML} will be a local minimum of $E(\mathbf{w})$ (non-convex optimization)
- Using \mathbf{w}_{ML} we can optimize for β by minimizing the negative log-likelihood function:

$$\frac{1}{\beta_{ML}} = \frac{1}{N} \sum_{n=1}^N \left(y(\mathbf{x}_n, \mathbf{w}_{ML}) - t_n \right)^2$$

The optimizations w.r.t. \mathbf{w} and β are performed by using an iterative optimization process.

When the targets are vectors \mathbf{t}_n , $n = 1, \dots, N$:

- We obtain \mathbf{w}_{ML} by minimizing the **mean-squared error between the outputs and the targets**:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2$$

- We use \mathbf{w}_{ML} to optimize for β_{ML} :

$$\frac{1}{\beta_{ML}} = \frac{1}{NK} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}_{ML}) - \mathbf{t}_n\|^2$$

Neural Network Training: Binary classification*

Consider that the targets are $t_n \in \{0, 1\}$, where:

- $t_n = 1$ means that $\mathbf{x}_n \in \mathcal{C}_1$
- $t_n = 0$ means that $\mathbf{x}_n \in \mathcal{C}_2$

The neural network will have one output neuron with logistic sigmoid activation function:

$$y = \sigma(\alpha) = \frac{1}{1 + \exp(-\alpha)}$$

ensuring that $0 \leq y(\mathbf{x}, \mathbf{w}) \leq 1$.

Neural Network Training: Binary classification*

The conditional distribution of t w.r.t. \mathbf{x} and \mathbf{w} is then a Bernoulli distribution:

$$p(t|\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t (1 - y(\mathbf{x}, \mathbf{w}))^{1-t}$$

The negative log-likelihood function becomes the **cross-entropy error function**:

$$E(\mathbf{w}) = - \sum_{n=1}^N \left(t_n \ln y(\mathbf{x}_n, \mathbf{w}) + (1 - t_n) \ln(1 - y(\mathbf{x}_n, \mathbf{w})) \right)$$

Neural Network Training: Multi-class classification*

The target vectors $\mathbf{t}_n \in \mathbb{R}^K$ follow the 1-of- K coding scheme.

The error function is:

$$E(\mathbf{w}) = - \sum_{n=1}^N \sum_{k=1}^K t_{kn} \ln y_k(\mathbf{x}_n, \mathbf{w})$$

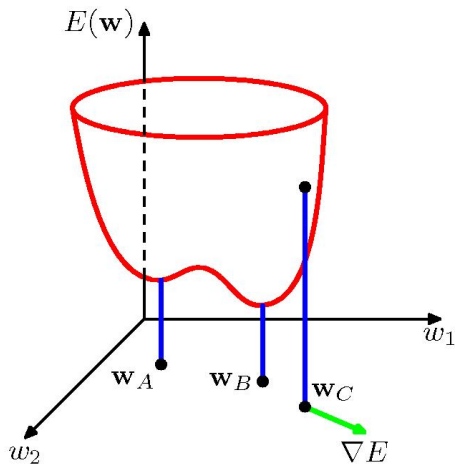
The output neuron activation function is the **softmax function**:

$$y_k(\mathbf{x}, \mathbf{w}) = \frac{\exp(\alpha_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(\alpha_j(\mathbf{x}, \mathbf{w}))}$$

ensuring that $0 \leq y_k(\mathbf{x}, \mathbf{w}) \leq 1$ and $\sum_k y_k(\mathbf{x}, \mathbf{w}) = 1$.

Parameter optimization

Local minima correspond to $\nabla E(\mathbf{w}) = 0$.



Parameter optimization

Process:

- 1 Choose an initial set of parameter values $\mathbf{w}^{(0)}$
- 2 Update the parameters until reaching a local minimum using:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{\tau} + \Delta\mathbf{w}^{(\tau)}$$

Several optimization techniques which use different choices for $\Delta\mathbf{w}^{(\tau)}$:

- Local quadratic approximation
- **Gradient descent optimization**
- **Error Backpropagation**
- Recent schemes: RMSprop, Adagrad, Adadelata, **Adam**, Adamax, Nadam

Parameter optimization: Gradient descent

Gradient descent updates \mathbf{w} using:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^\tau - \eta \nabla E(\mathbf{w}^{(\tau)})$$

To find a sufficiently good minimum, it may be necessary to run a gradient-based algorithm multiple times, each time using a different randomly chosen starting point.

When

$$E(\mathbf{w}) = \sum_{n=1}^N E_n(\mathbf{w})$$

stochastic gradient descent (SGD) can be used:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^\tau - \eta \nabla E_n(\mathbf{w}^{(\tau)})$$

Mini-batch gradient descent uses small chunks (e.g. 64) data points for each update.

Parameter optimization: Learning Rate

In the gradient descent equation $\mathbf{w}^{(\tau+1)} = \mathbf{w}^\tau - \eta \nabla E(\mathbf{w}^{(\tau)})$, η is the learning rate which determines the size of the update step.

Small learning rates:

- Lead to slow convergence and to a long training time
- Get easier stuck in local minima

Large learning rates:

- Lead to unstable training
- Are easier to diverge

How can we find the correct learning rate?

- Try multiple learning rates and pick the best one (hyper-parameter optimization).
- Adapt the learning rate to the landscape.

Error Backpropagation

Iterative process for updating the weights of a neural network formed by two steps:

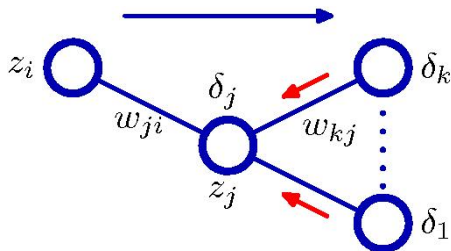
- Step 1: Evaluate the derivatives of the error function with respect to the weights
- Step 2: Adjust the weights using the derivatives

The contribution of the Backpropagation algorithm is that it provides a computationally efficient method for evaluating the derivatives.

Backpropagation algorithm can be used for:

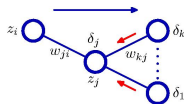
- a general network having arbitrary feed-forward topology
- arbitrary *differentiable nonlinear activation functions*,
- a broad class of error functions

Error Backpropagation*



Forward and backward pass at a neuron

Error Backpropagation*



At neuron j the forward pass is:

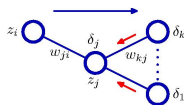
$$\alpha_j = \sum_i w_{ij} z_i \quad z_j = h(\alpha_j).$$

The gradient of E_n w.r.t. w_{ji} can be expressed as (*chain rule*):

$$\frac{\partial E_n}{\partial w_{ji}} = \underbrace{\frac{\partial E_n}{\partial \alpha_j}}_{\delta_j} \underbrace{\frac{\partial \alpha_j}{\partial w_{ji}}}_{z_i} = \delta_j z_i$$

Calculating the derivative of the error at neuron j is equal to multiplying the value of the *error signal* at neuron j (δ_j) with the value of the output of neuron j (z_j , for 'dummy' neurons interacting with the bias $z = 1$).

Error Backpropagation*



The error signal at neuron j is given by:

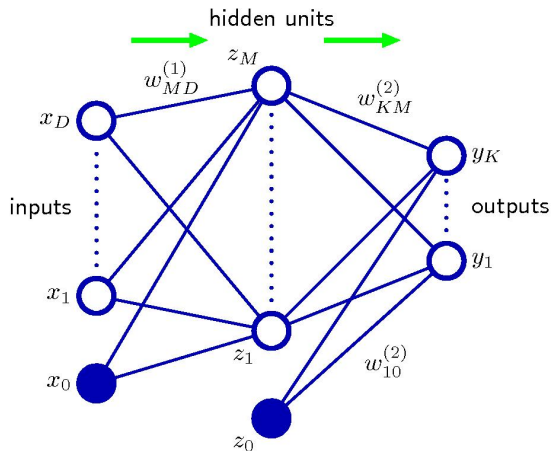
$$\delta_j = \frac{\partial E_n}{\partial \alpha_j} = \sum_k \frac{\partial E_n}{\partial \alpha_k} \frac{\partial \alpha_k}{\partial \alpha_j} = h'(\alpha_j) \sum_k w_{kj} \delta_k$$

Thus, the error signals at a neuron j are obtained by *backpropagating* the error signals from units higher up in the network.

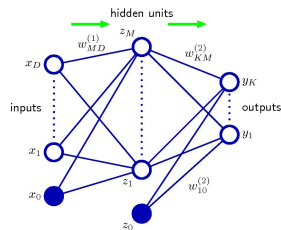
For (mini-)batch methods, when $E(\mathbf{W}) = \sum_n E_n(\mathbf{w})$ we have:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}$$

Error Backpropagation: Example*



Error Backpropagation: Example*



We use:

- Sum-of-squares error:

$$E_n = \frac{1}{2} \sum_{k=1}^K (y_k - t_k)^2$$

- linear output neurons
- hidden neurons with activation function:

$$h(\alpha) = \tanh(\alpha) = \frac{e^\alpha - e^{-\alpha}}{e^\alpha + e^{-\alpha}}$$

Error Backpropagation: Example*

Forward pass:

- For each input data point \mathbf{x}_n calculate:

$$\alpha_j = \sum_{i=0}^D W_{ji}^{(1)} x_i, \quad z_j = \tanh(\alpha_j), \quad y_k = \sum_{j=1}^M W_{kj}^{(2)} z_j$$

Calculate the error signals for each output neurons: $\delta_k = y_k - t_k$.

Backward pass:

- Backpropagate the error signals:

$$\delta_j = (1 - z_j^2) \sum_{k=1}^K w_{kj} \delta_k$$

- Calculate the gradients w.r.t. the weights $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$:

$$\frac{\partial E_n}{\partial W_{ji}^{(1)}} = \delta_j x_i, \quad \frac{\partial E_n}{\partial W_{ji}^{(2)}} = \delta_k z_j$$

Regularization in Neural Networks

Weight decay:

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2} \mathbf{w}^T \mathbf{w}$$

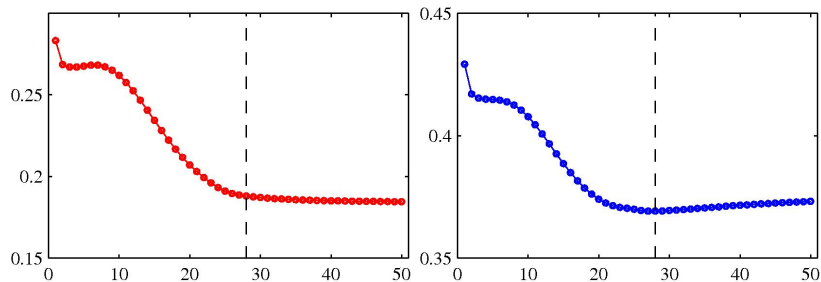
To allow the regularizer to be invariant under linear transformations of the input data (for a network with L layers):

$$\frac{\lambda_1}{2} \sum_{w \in \mathcal{W}_1} w^2 + \frac{\lambda_2}{2} \sum_{w \in \mathcal{W}_2} w^2 + \dots + \frac{\lambda_L}{2} \sum_{w \in \mathcal{W}_L} w^2$$

where \mathcal{W}_l denotes the set of weights in layer l .

Regularization in Neural Networks

Early stopping:



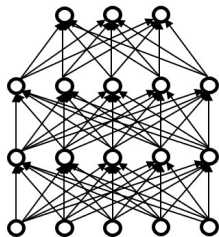
Left: Error function on the training data

Right: Error function on the validation data

Regularization in Neural Networks: Dropout

Dropout in iterative optimization: A probabilistic process to 'augment' the training set during iterative training and increase invariance:

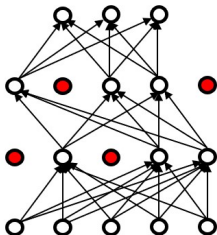
Standard neural network training



All iterations

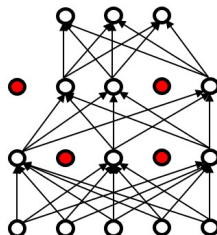
Dropout-based training

At each iteration, each neuron is active with probability p (using Bernoulli distribution and cut-off value of e.g. $p = 0.5$)



Training iteration 1

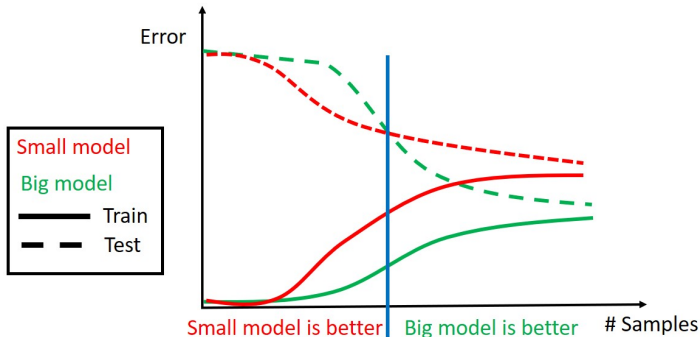
...



Training iteration t

Regularization in Neural Networks: Training set size

When the number of training samples is small (smaller than the number of the model's parameters) the model tends to overfit (under-determined problem).



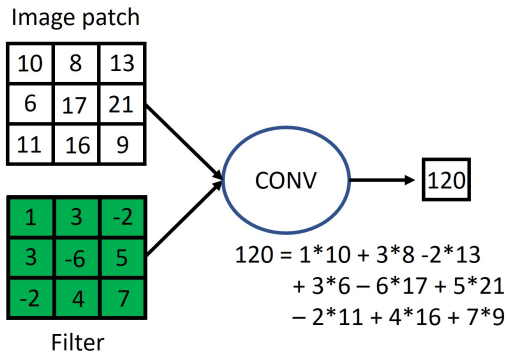
Regularization in Neural Networks: Training set size

In neural networks, this problem is usually addressed by using:

- Data augmentation: create new samples by applying small variations on the training data. For example, for images: geometric variations (shift, rotations, scaling), crops, noise
- Transfer learning:
 - 1 Initialize the model's parameters with those of a pre-trained model on a big data set (having similar properties to the problem we want to solve)
 - 2 fine-tune the model using the small data set



Convolution Operation



Convolutional Neural Networks

Convolutional Neural Network (CNN, ConvNet):

- A neural network with at least one layer where the matrix multiplication operation is replaced by the convolution operator.

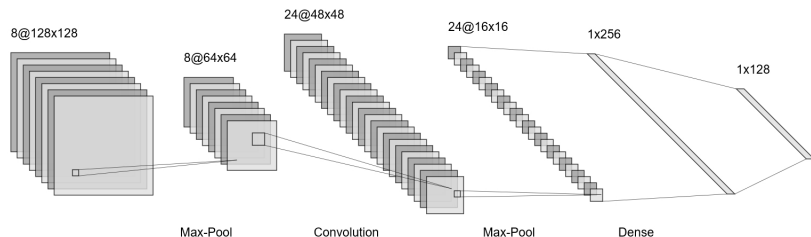


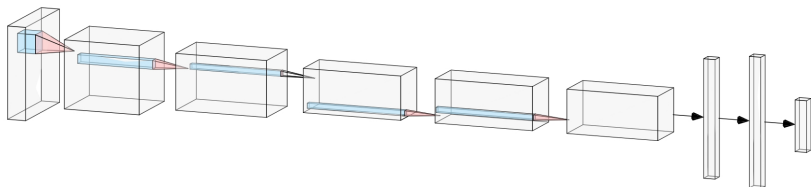
Figure: LeNet

Figure created using <http://alexlenail.me/NN-SVG/LeNet.html>

Convolutional Neural Networks

Convolutional Neural Network (CNN, ConvNet):

- A neural network with at least one layer where the matrix multiplication operation is replaced by the convolution operator.



The Convolution Layer

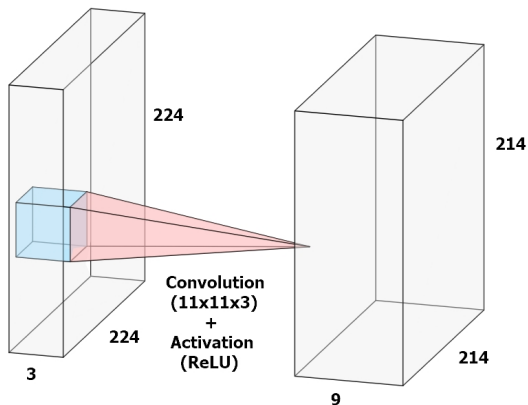
A *tensor filter* convolves the input third-order tensor (which is the image in the first layer):

- at each position it produces a value.
- follows a *sliding window* approach to process all the locations in the input tensor.
- leads to one *output map* (either a matrix or a third-order tensor with its last dimension equal to 1).

Convolving an input third-order tensor with K tensor filters leads to the creation of K output maps (a third-order tensor with its last dimension equal to K)

The Convolution Layer

A convolution with 9 filters, each having a size of $11 \times 11 \times 3$:

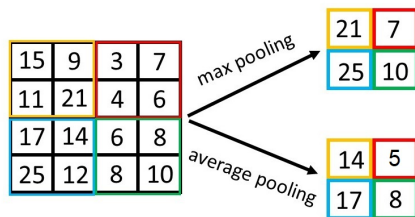


Pooling

Pooling is used to:

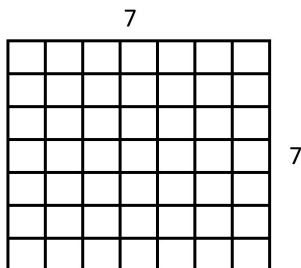
- Reduces the memory footprint of the CNN over the successive layers.
- Summarizes information in local neighborhoods (leads to small invariance w.r.t. shifting the input signal).

At later layers of the CNN, each *tensor fiber* encodes information of a larger area in the input image (*receptive field*).



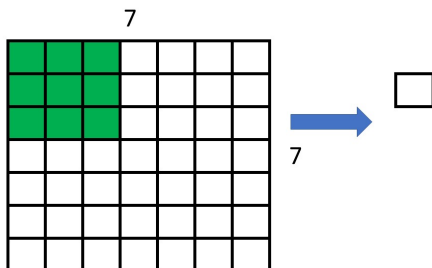
Convolution stride

Convolution with a stride of one:



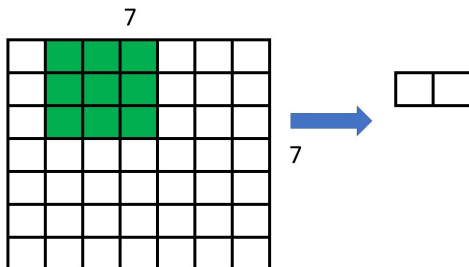
Convolution stride

Convolution with a stride of one:



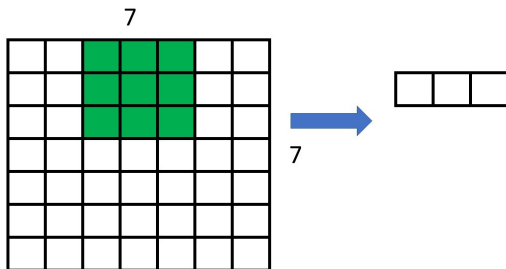
Convolution stride

Convolution with a stride of one:



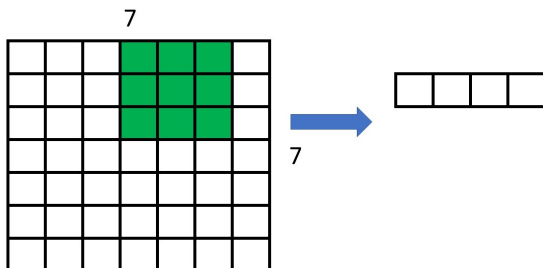
Convolution stride

Convolution with a stride of one:



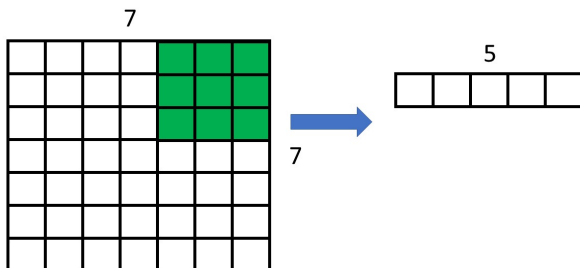
Convolution stride

Convolution with a stride of one:



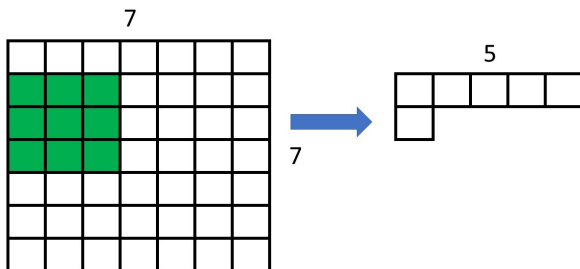
Convolution stride

Convolution with a stride of one:



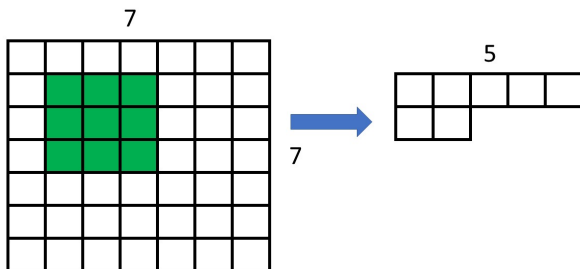
Convolution stride

Convolution with a stride of one:



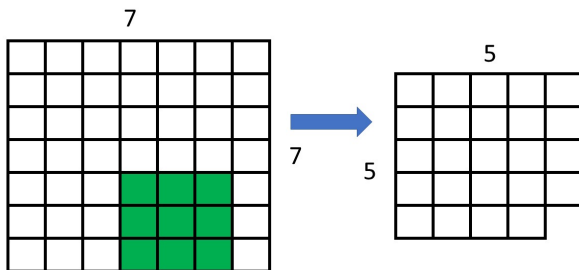
Convolution stride

Convolution with a stride of one:



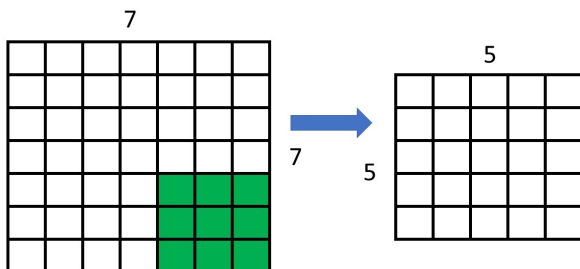
Convolution stride

Convolution with a stride of one:



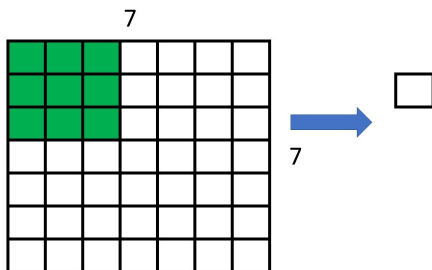
Convolution stride

Convolution with a stride of one:



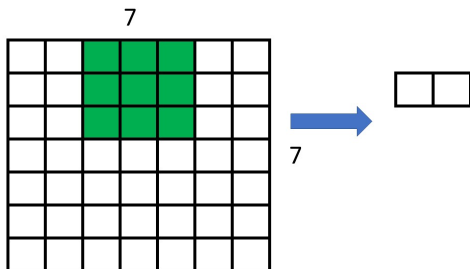
Convolution stride

We can achieve similar effects to pooling by using a stride greater than one:



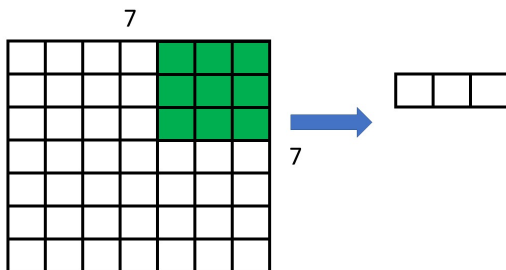
Convolution stride

We can achieve similar effects to pooling by using a stride greater than one:



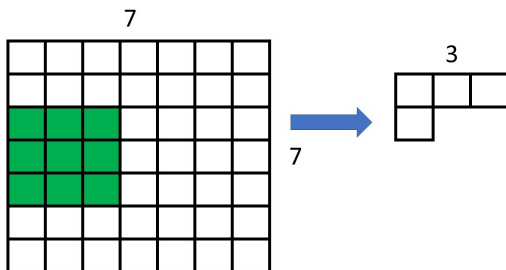
Convolution stride

We can achieve similar effects to pooling by using a stride greater than one:



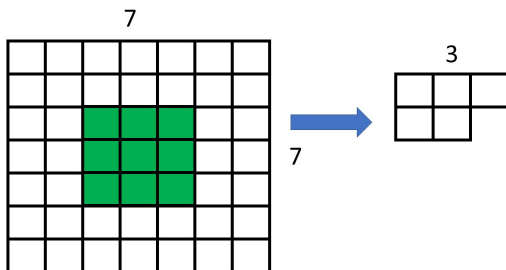
Convolution stride

We can achieve similar effects to pooling by using a stride greater than one:



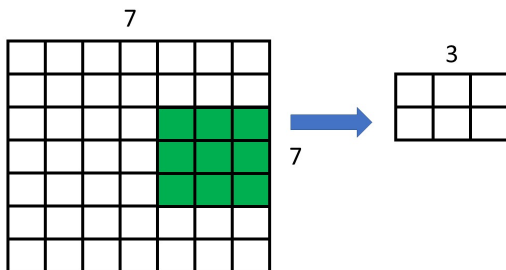
Convolution stride

We can achieve similar effects to pooling by using a stride greater than one:



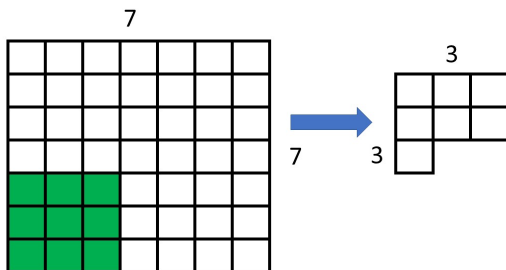
Convolution stride

We can achieve similar effects to pooling by using a stride greater than one:



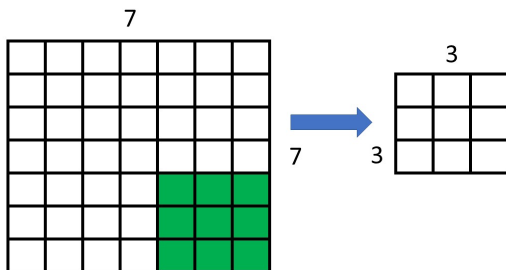
Convolution stride

We can achieve similar effects to pooling by using a stride greater than one:



Convolution stride

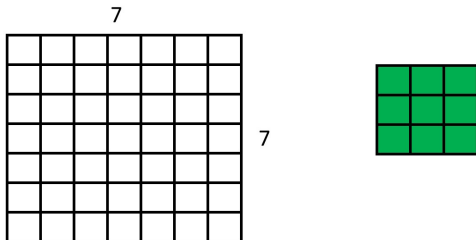
We can achieve similar effects to pooling by using a stride greater than one:



Zero padding

To create an output tensor for which the first two dimensions match those of the input tensor we use zero padding with size:

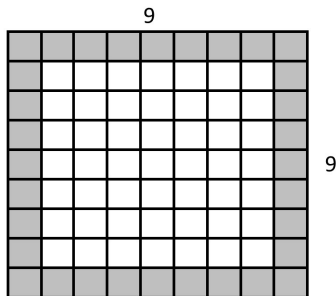
$$\#zeros = (\#image\ dims - \#filter\ dims) / stride + 1$$



Zero padding

To create an output tensor for which the first two dimensions match those of the input tensor we use zero padding with size:

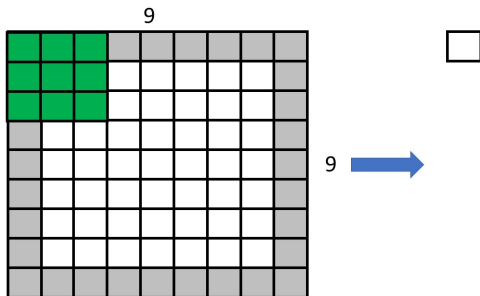
$$\#zeros = (\#image\ dims - \#filter\ dims) / stride + 1$$



Zero padding

To create an output tensor for which the first two dimensions match those of the input tensor we use zero padding with size:

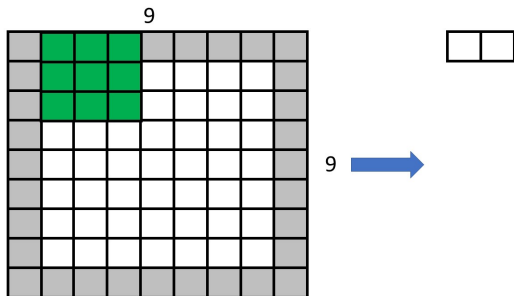
$$\#zeros = (\#image\ dims - \#filter\ dims) / stride + 1$$



Zero padding

To create an output tensor for which the first two dimensions match those of the input tensor we use zero padding with size:

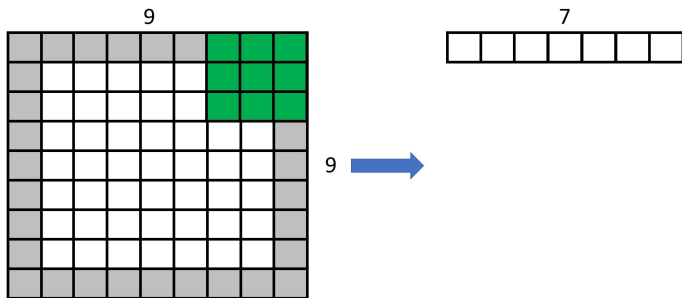
$$\#zeros = (\#image\ dims - \#filter\ dims) / stride + 1$$



Zero padding

To create an output tensor for which the first two dimensions match those of the input tensor we use zero padding with size:

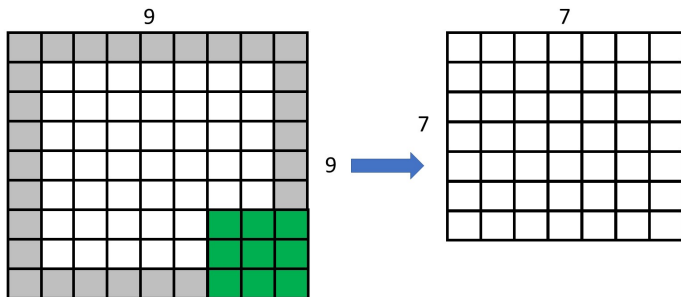
$$\#zeros = (\#image\ dims - \#filter\ dims) / stride + 1$$



Zero padding

To create an output tensor for which the first two dimensions match those of the input tensor we use zero padding with size:

$$\#zeros = (\#image\ dims - \#filter\ dims) / stride + 1$$



State-of-the-art (SOTA) neural network for image classification in 2012 using convolutions, pooling and MLP.

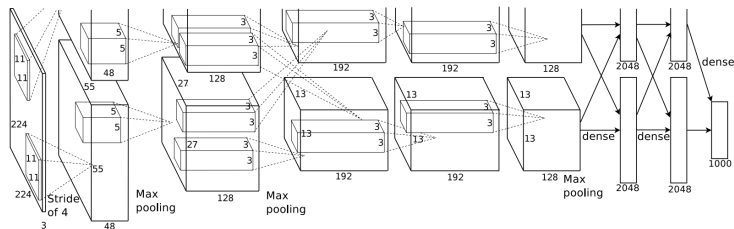


Figure: AlexNet Architecture

Image taken from "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012

SOTA neural network for image classification in 2014, using the same architectural elements but more layers.

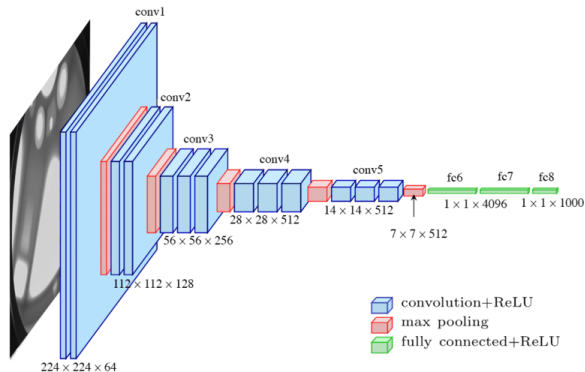


Figure: VGG Architecture

Image taken from "Automatic localization of casting defects with convolutional neural networks", IEEE Big Data 2017

Residual Connections

Residual connections make training more stable, allowing for deeper networks.

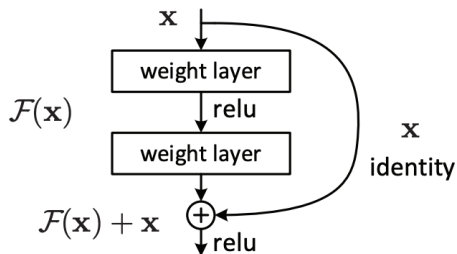


Figure: Residual Connection

Image taken from "Deep Residual Learning for Image Recognition", CVPR 2016

Using residual connections, ResNet can have many more layers than VGG.

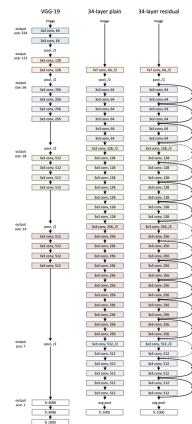


Figure: Residual Connection

Image taken from “Deep Residual Learning for Image Recognition”, CVPR 2016

Inception

It is difficult to determine the convolution parameters (e.g. filter size).

The Inception module includes multiple possibilities in parallel.

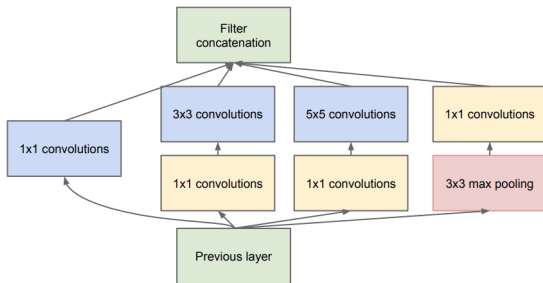


Figure: Inception Module

Image taken from “Going deeper with convolutions”, CVPR 2015

3D Convolution

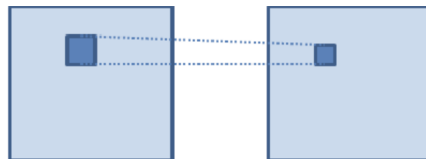
An image is represented by a 3D tensor $H \times W \times C$.

A video can be represented as a series of images, resulting in a 4D tensor $T \times H \times W \times C$, where T is the temporal dimension.

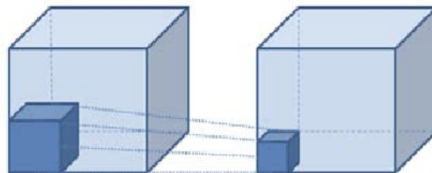
To process a video with deep learning, it is not enough to process each image individually, as the changes that occur between frames over time contain valuable information.

3D convolution is a type of convolution where the kernel slides in three dimensions instead of two, and can be used for processing videos.

3D Convolution



(a) 2D convolution



(b) 3D convolution

3. Comparison of (a) 2D convolution and (b)

Figure: 3D convolution, image taken from: L. Fan, Z. Xia, X. Zhang and X. Feng, "Lung nodule detection based on 3D convolutional neural networks," 2017 International Conference on the Frontiers and Advances in Data Science (FADS), 2017, pp. 7-10.

Continual 3D CNNs

3D CNNs perform redundant computations for a temporal convolution during online inference.

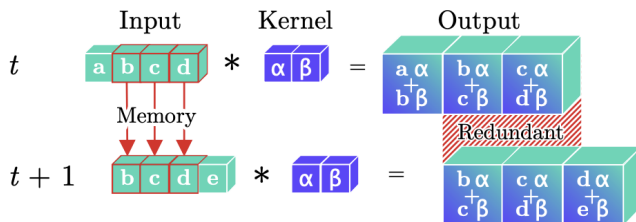


Figure: Redundant computation, image taken from: Hedegaard, Lukas, and Alexandros Iosifidis. "Continual 3D Convolutional Neural Networks for Real-Time Processing of Videos." European Conference on Computer Vision (ECCV), 2022.

Continual 3D CNNs

This redundancy can be removed by using continual convolutions instead. The intermediate feature maps corresponding to all but the last temporal position are stored, while the last feature map and prior memory are summed to produce the resulting output.

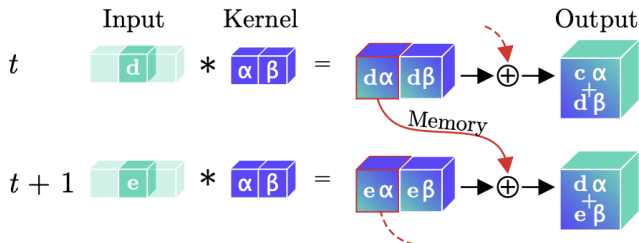


Figure: Continual convolution, image taken from: Hedegaard, Lukas, and Alexandros Iosifidis. "Continual 3D Convolutional Neural Networks for Real-Time Processing of Videos." European Conference on Computer Vision (ECCV), 2022.

Vision Transformer

Alternative architecture for computer vision tasks that does not include convolutions.

Based on the Transformer architecture which has been widely used in natural language processing since 2017.

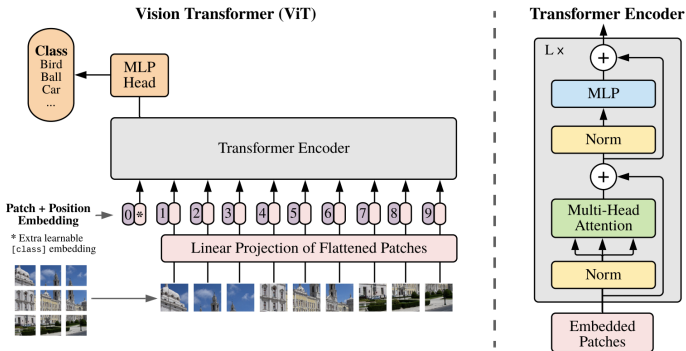


Figure: Image taken from "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

Vision Transformer: Self-Attention

If the key, query and value vectors are packed into matrices $Q = XW^Q$, $K = XW^K$ and $V = XW^V$, where W^Q , W^K and W^V are learnable weight matrices, the self-attention operation can be rephrased as follows:

$$Z = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V$$

Self-attention is extended to *multi-head attention* in order to enable the model to capture more than one type of relationship between the entities in the sequence.

Positional embeddings are additional learnable parameters that are added to each patch in order to solve this issue.

Classification token is an additional token which is added alongside the input sequence of tokens.

Continual Transformers

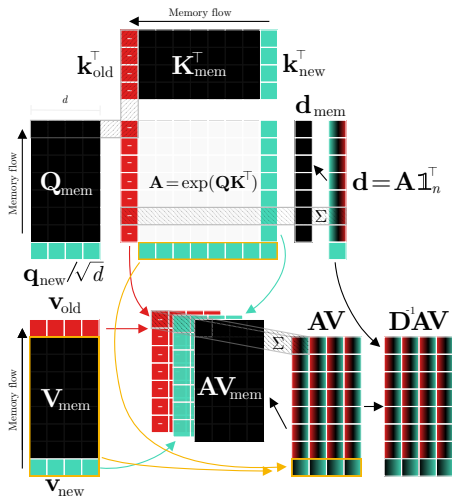


Figure: Continual Transformer, image taken from: Hedegaard, Lukas, Arian Bakhtiarnia, and Alexandros Iosifidis. "Continual Transformers: Redundancy-Free Attention for Online Inference." arXiv preprint arXiv:2201.06268 (2022).

Hands-on examples

Regression-based classification

Perceptron-based classification

Multilayer Perceptron and CNN on MNIST

Transfer Learning for image classification on CIFAR10

3D Convolutional Neural Network based classification on 3D MNIST

Vision Transformer for image classification on CIFAR10