Automated Feature Extraction with Machine Learning and Image Processing

PD Stefan Bosse

University of Siegen - Dept. Maschinenbau
University of Bremen - Dept. Mathematics and Computer Science

Training and Validation of data-driven Models

Adapting dynamic parameters of a functional network is an iterative optimization problem

Training and Validation of data-driven Models

Adapting dynamic parameters of a functional network is an iterative optimization problem

Commonly the solution space is infinite, i.e., there is no one valid solution of the optimization problem.

Training and Validation of data-driven Models

Adapting dynamic parameters of a functional network is an iterative optimization problem

Commonly the solution space is infinite, i.e., there is no one valid solution of the optimization problem.

Basic training is demonstrated for an Artificial Neural Network

A simple Artificial Neuron

A simple neuron (perecptron) is a mapping function *f* (a model) that maps an n-dimensional input vector ***v*** on a scalar value *u*:

$$f(\vec{x}, \vec{w}, b) = g\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Here ***w*** is weight vector and *b* an offset (dynamic parameters). The function *g* is called transfer or activation function, normally not parametrized.

A simple Artificial Neuron

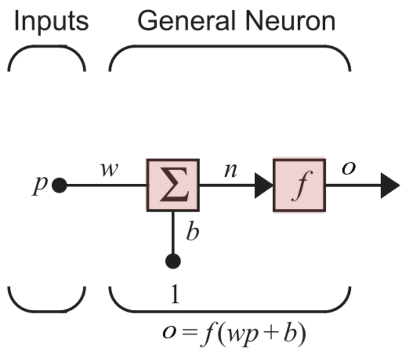


Fig. 1. A single neuron with a single input *p* and an output *o*. *w* is a weighting factor (a weight for incoming *p*) and *b* is a bias (offset)

A Multi-input Artificial Neuron

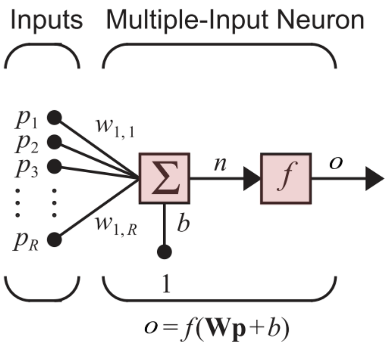


Fig. 2. A single neuron with an input vector ***p*** and a scalar output *o*. ***w*** is a weighting factor vector (a weight for incoming *p*) and *b* is a bias (offset)

Artificial Neural Network

A ANN is a function graph consisting of interconnected neurons. It is a graph $G(V,N)$ with a set of nodes (neurons) and vertices connecting the nodes.

Commonly neurons are arranged and grouped in layers, but this is not mandatory. There is always an input and one output layer. Hidden layers are between input and output layers.

Artificial Neural Network

- The input layer (commonly) consists of *n* neurons for *n* input variables (attributes).
- The output layer (commonly) consists of *m* neurons for *m* output variables (regression) or *m* target classes (classification)
- Commonly, but not mandatory, each neuron of a layer *i* is connected with the outputs of all neurons of the previous layer *i*-1

Artificial Neural Network

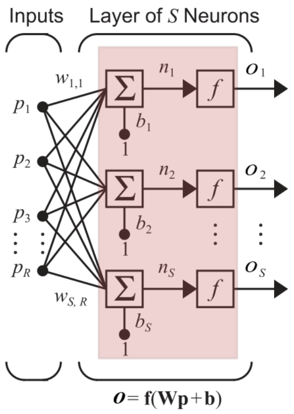


Fig. 3. Neural network with neurons arranged in one layer

Loss and Error Functions

Assume there is a set of data samples ***D***, each sample contains the ***x*** input feature vector and output target feature vector ***y***.

- The goal of the model training is to find a model function that maps ***x*** on ***y*** with minimal error for **all instances** (at least averaged)
- The loss or error function defines the mismatch of a training or test sample with the output of the function *f* (here for one scalar output *y*):

$$y = f(\vec{x})$$
$$MAE(y, y_0) = |y_0 - y|$$
$$MBE(y, y_0) = y_0 - y$$
$$MSE(y, y_0) = (y_0 - y)^2$$

Loss and Error Functions

- For multiple outputs (***y***) we get:

$$\vec{y} = f(\vec{x})$$

$$MAE(\vec{y}, \vec{y}_0) = \frac{\sum_{i=1}^{n} \left| y_i - y_{0,i} \right|}{n}$$

$$MBE(\vec{y}, \vec{y}_0) = \frac{\sum_{i=1}^{n} y_i - y_{0,i}}{n}$$

$$MSE(\vec{y}, \vec{y}_0) = \frac{\sum_{i=1}^{n} \left(y_i - y_{0,i} \right)^2}{n}$$

Training by Error Backpropagation

Most of the CNN layers involve parameters which are required to be tuned appropriately for a given computer vision task (e.g., image classification and object detection).

- Assume again a single perceptron neuron with only two inputs *a* and *b*.
- Then we can change the respective weight parameter *w* just by computing the "forward" application error, and subtracting the error multiplied with the current input value from the weight *w* (***Rough approximation!***):

$$w\acute{}_i = w_i - \alpha(y - y_0)x_i$$

WorkBook Live

CLEAR LOAD + - Neuron

Example

```
data = {
  {x1=0,  x2=0,  y=0},
  {x1=1,  x2=0,  y=0.3},
  {x1=0,  x2=1,  y=0.5},
  {x1=1,  x2=1,  y=1},
}
function sigmoid(x) {
  1/(1+exp(-x))
}
function neuron(x1,x2,w,b) {
  accu = x1w[1]+x2w[2]
  sigmoid(accu+b)
}
```

Ex. 1. Some training data and the implementation of the sigmoid (logstic regression) activation and neuron function with two inputs

Example

```
w = [0,0] b=0 samples=1:4 rate=0.01
for (run in 1:1000) {
  set=sample(samples,1)
  row=data[[set]]
  y=neuron(row$x1,row$x2,w,b)
  err=y-row$y
  w[1]=w[1]-rate*err*row$x1
  w[2]=w[2]-rate*err*row$x2
  b=b-rate*err
}
print(w) print(b)
```

Ex. 2. Training with randomized selected sample instances

Example

```
for (index in 1:4) {
  row=data[[index]]
  y=neuron(row$x1,row$x2,w,b)
  print(paste('Index',index,'Predicted',y,'Error',y-row$y))
}
```

Ex. 3. Test with sample instances

Gradient Descent Method

- Indeed, the gradient of the output error with respect to the weight parameter w_i is computed and subtracted from the current weight parameter value:

$$w\acute{}_i = w_i - \alpha \frac{\partial (y - y_0)}{\partial w_i}$$

- That means, the weight parameter is corrected by a term that corresponds to the amount of the change of the error by changing the weight by a small delta value.

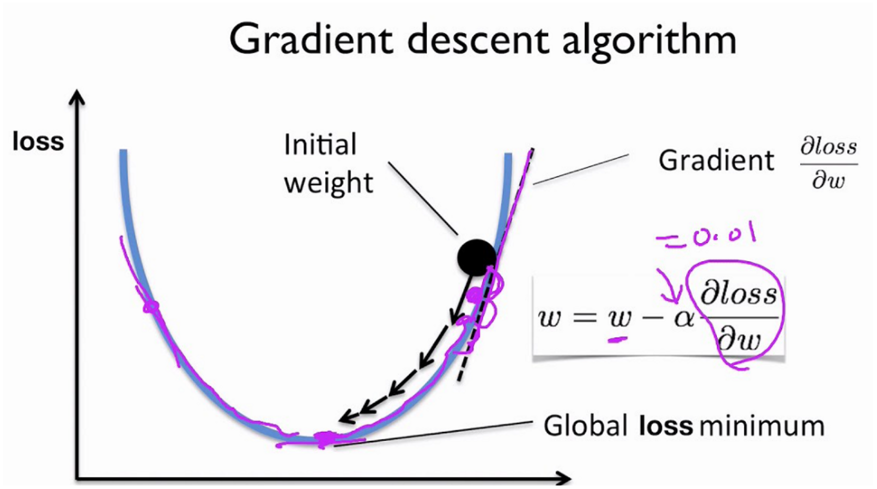

Gradient descent algorithm
loss
Initial weight
Gradient $\frac{\partial loss}{\partial w}$
$= 0.01$
$w = w - \alpha \frac{\partial loss}{\partial w}$
Global **loss** minimum

Fig. 4. The learning rate α determines the steps to be taken along the slope to achieve the goal. Too large steps could result in jumping over or missing the point of global minimum(also known as overshooting) and too small steps results in a very slow process of achieving the goal. This is a hyperparameter that needs to be tuned. In practice, people often start with 0.01, and either decrease or increase accordingly. (Aminah Mardiyyah Rufai)

Learning Rate

- But: We have a lot of different training samples, and if we change the parameter only based on the error from the current sample we will not converge to an average!

- Therefore, only a small fraction given by the learning rate parameter α is used!

Error backpropagation in layered Networks

- Up to here we considered only one functional node (one neuron).
- If parameters of functions of previous nodes/layers must be adapted, the process is a little bit more complicated, although, the same principle is applied, i.e., in general the derivative of the error function by the respective weight/parameter to be adjusted must be computed:

$$\frac{\partial E}{\partial w_i}$$

[Matt Mazur, https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/]

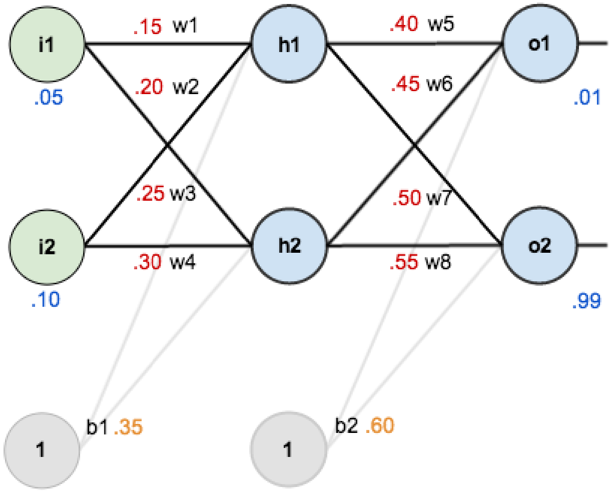
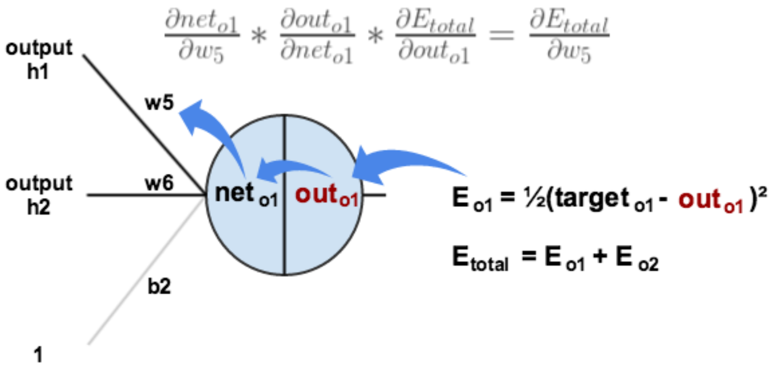


Fig. 5. Example network with teo input node, two inner nodes, and two output nodes

- Let us now consider one node with an input vector ***x***, a product-sum result *net* (***x***,***w***) applied to the transfer function *f*, and a resulting output *out*, then we can write by a simple chain rule:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial out_i} \cdot \frac{\partial out_i}{\partial net_i} \cdot \frac{\partial net_i}{\partial w_i}$$



$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$
output h1
output h2
1
w5
w6
b2
net o1
out o1
$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2$
$E_{total} = E_{o1} + E_{o2}$

- In the hidden inner layer, we start with the same formula, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons.

- We know that out_{h1} affects both out_{o1} and out_{o2} therefore the gradient needs to take into consideration its effect on the both output neurons:

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial out_{h1}} \cdot \frac{\partial out_{h1}}{\partial net_{h1}} \cdot \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

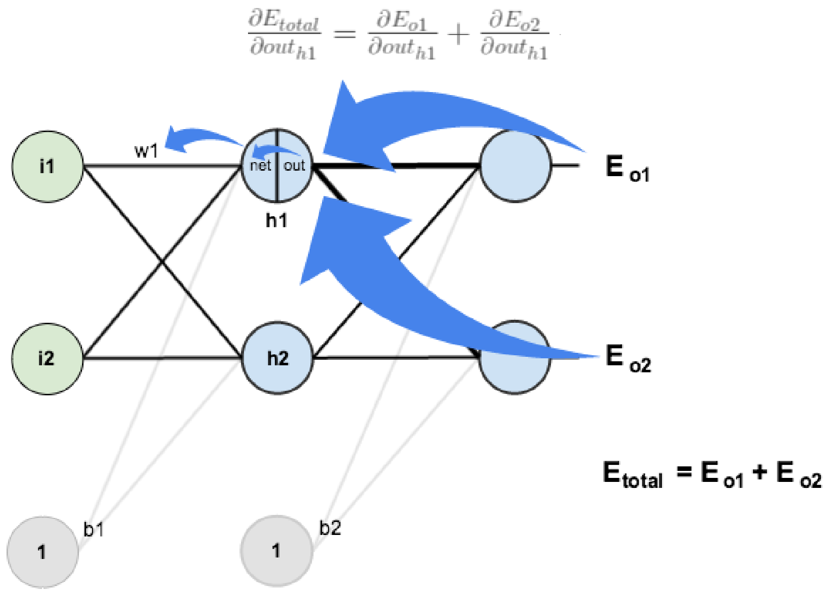

$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$
i1
i2
w1
net
out
h1
h2
E_{o1}
E_{o2}
$E_{total} = E_{o1} + E_{o2}$
1
b1
1
b2

Fig. 6. Error backpropagation from output to inner layer nodes must consider error accumulation by multiple nodes

Weight Initialization

A correct weight initialization is the key to stably train very deep networks. An ill-suited initialization can lead to the vanishing or exploding gradient problem during error back-propagation.

Gaussian Random Initialization

A common approach to weight initialization in CNNs is the Gaussian random initialization technique. This approach initializes the convolutional and the fully connected layers using ran- dom matrices whose elements are sampled from a Gaussian distribution with zero mean and a small standard deviation (e.g., 0.1 and 0.01).

Uniform Random Initialization

The uniform random initialization approach initializes the convolutional and the fully connected layers using random matrices whose elements are sampled from a uniform distribution (instead of a normal distribution as in the earlier case) with a zero mean and a small standard deviation (e.g., 0.1 and 0.01).

- The uniform and normal random initializations generally perform identically.
- However, the training of very deep networks can become a problem with a random initializa- tion of weights from a uniform or normal distribution.
    - The reason is that the forward and backward propagated activations can either diminish or explode when the network is very deep.

Xavier Initialization

A random initialization of a neuron makes the variance of its output directly proportional to the number of its incoming connections (a neuron's fan-in measure).

- To alleviate this problem, Glorot and Bengio [2010] proposed to randomly initialize the weights with a variance measure that is dependent on the number of incoming and outgoing connections (n_{fin} and n_{fout} respectively) from a neuron,

$$Var(w) = \frac{2}{n_{fin} + n_{fout}}$$

where w are network weights. Note that the fan-out measure is used in the variance above to balance the back-propagated signal as well. Xavier initialization works quite well in practice and leads to better convergence rates.

ReLU-scaled Initialization

Neurons (or filters with transfer functions) with a ReLU non-linearity do not follow the assumptions made for the Xavier initialization.

- Precisely, since the ReLU activation reduces nearly half of the inputs to zero, therefore the variance of the distribution from which the initial weights are randomly sampled should be

$$Var(w) = \frac{2}{n_{fin}}$$

- The ReLU aware scaled initialization works better compared to Xavier initialization for recent architectures which are based on the ReLU nonlinearity.

Pre-training

One approach to avoid the gradient diminishing or exploding problem is to use layer-wise pre-training in an unsupervised fashion.

- The unsupervised pre-training can be followed by a supervised fine-tuning stage to make use of any available annotations.
- However, due to the new hyper-parameters, the considerable amount of effort involved in such an approach and the availability of better initialization techniques, layer-wise pre-training is seldomused now to enable the training of CNN-based very deep networks.

(not a good idea)

Supervised Pre-Training

In practical scenarios, it is desirable to train very deep networks, but we do not have a large amount of annotated data available for many problem settings.

- A very successful practice in such cases is to first train the neural network on a related but different problem, where a large amount of training data is already available.
- Afterward, the learned model can be “adapted” to the new task by initializing with weights pre-trained on the larger dataset.

This process is called “fine-tuning” and is a simple, yet effective, way to transfer learning from one task to another.

Training and Validation (Test)

The set of data samples are commonly split in two sub-sets:

1. Training data samples only used to compute model errors for model parameter optimization;
2. Test (or validation) data samples only used to check and assess the current model accuracy.

For gradient error back-propagation commonly linear error functions are used. For the validation, higher-order funtions (like MSE) can be used.

Regularization

Since deep conv. and neural networks have a large number of parameters, they tend to over-fit on the training data during the learning process.

- Over-fitting meana that the model performs really well on the training data but it fails to generalize well to unseen data.
- It, therefore, results in an inferior performance on new data (usually the test set).

Regularization approaches aim to avoid this problem using several intuitive ideas.

Regularization

We can categorize common regularization approaches into the following classes, based on their central idea:

- approaches which regularize the network using data level techniques (e.g., data augmentation);
- approaches which introduce stochastic behavior in the neural activations (e.g., dropout and drop connect);
- approaches which aligns parameters of "saturated" nodes to bring the back in the non-saturation range;
- approaches which normalize batch statistics in the feature activations (e.g., batch normalization);
- approaches which use decision level fusion to avoid over-fitting (e.g., ensemble model averaging);
- approaches which introduce constraints on the network weights (e.g., `1 norm`,2 norm, max-norm, and elastic net constraints); and
- approaches which use guidance from a validation set to halt the learning process (e.g., early stopping).

Data Augmentation

Data augmentation is the easiest, and often a very effective way of enhancing the generalization power of CNN models. Especially for cases where the number of training examples is relatively low, data augmentation can enlarge the dataset (by factors of 16x, 32x, 64x, or even more) to allow a more robust training of large-scale models.

Data augmentation is performed by making several copies from a single image using straightforward operations such as rotations, cropping, flipping, scaling, translations, and shearing. These operations can be performed separately or combined together to form copies, which are both flipped and cropped.

[Khan, 2018]

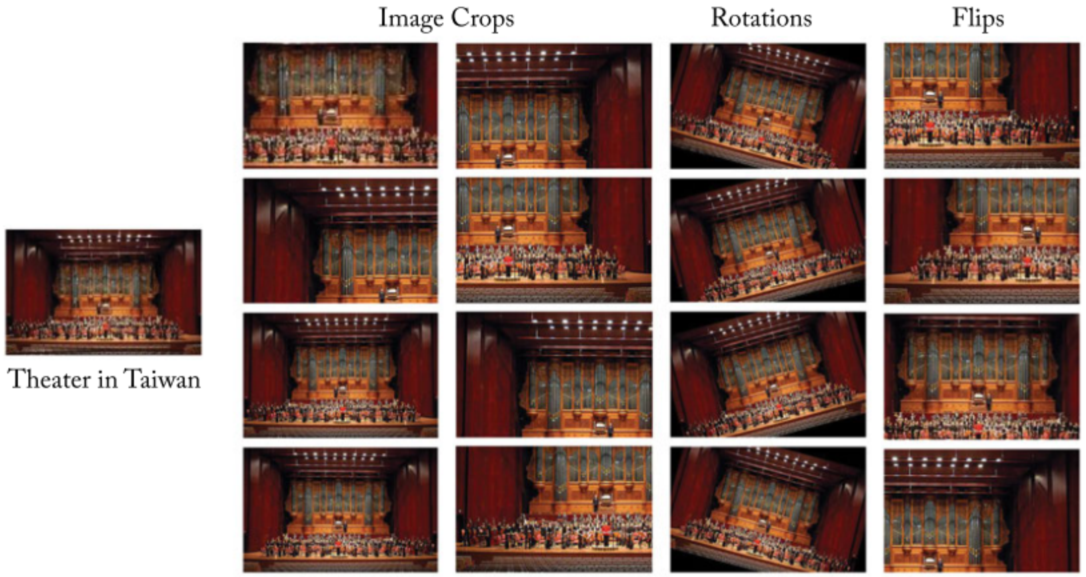


Fig. 7. Examples of data augmentation using image cropping, flipping, and rotation

Drop Out

One of the most popular approaches for neural network regularization is the dropout technique.

- During network training, each neuron is activated with a fixed probability (usually 0.5 or set using a validation set).
- This random sampling of a sub-network within the full-scale network introduces an ensemble effect during the testing phase, where the full network is used to perform prediction.
- Activation dropout works really well for regularization purposes and gives a significant boost in performance on unseen data in the test phase.

A random dropout layer generates a mask $\boldsymbol{m} \in B^{m}$, where each element m_{i} is indepently sampled from a Bernoulli distribution a probability p being on (or 1-p being off).

- This mask is used to modify the output activations from the previous layer, i.e.:

$$\vec{a}_l = \vec{m} \odot f\left(\hat{W} \cdot \vec{a}_{l-1} + \vec{b}_l\right)$$

Here, $\boldsymbol{a} \in \mathbb{R}^{n}$ and $\boldsymbol{b} \in \mathbb{R}^{m}$ denote the activations and biases respectively. $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ is the weight matrix, and f the transfer function.

Ensemble Model Averaging

The ensemble averaging approach is another simple, but effective, technique where a number of models are learned instead of just a single model.

- Each model has different parameters due to different random initializations, different hyper-parameter choices (e.g., architecture, learning rate) and/or different sets of training inputs.
- The output from these multiple models is then combined to generate a final prediction score.

Ensemble Model Averaging

- The prediction combination approach can be a simple output averaging, a majority voting scheme or a weighted combination of all predictions.
  - The final prediction is more accurate and less prone to over-fitting compared to each individual model in the ensemble.
  - The committee of experts (ensemble) acts as an effective regularization mechanism which enhances the generalization power of the overall system.

Early Stopping

The overfitting problem occurs when a model performs verywell on the training set but behaves poorly on unseen data.

- Early stopping is applied to avoid overfitting in the iterative gradient-based algorithms.
- This is achieved by evaluating the performance on a held-out validation set at different iterations during the training process.
    - The training algorithm can continue to improve on the training set until the performance on the validation set also improves.
    - Once there is a drop in the generalization ability of the learned model, the learning process can be stopped or slowed down.

[Khan, 2018]

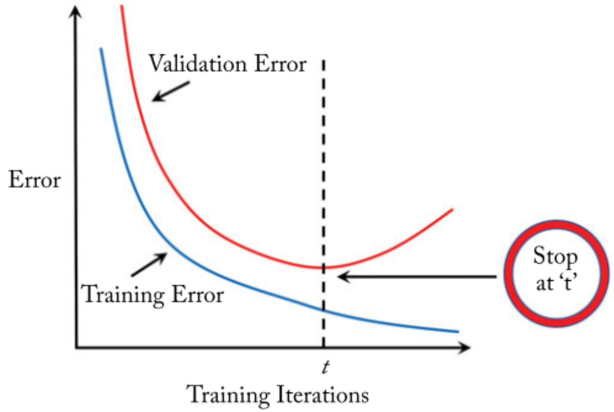


Fig. 8. An illustration of the early stopping approach during network training using the validation error for decision making instead a pre-defined training error threshold.

Gradient-based CNN Learning

The CNN learning process tunes the parameters of the network such that the input space is correctly mapped to the output space.

- At each training step, the current estimate of the output variables is matched with the desired output (often termed the "ground-truth" or the "label space").
- This matching function serves as an objective function during the CNN training and it is usually called the loss function or the error function.
- The CNN training process involves the optimization of its parameters such that the loss function is minimized.

Each iteration which updates the parameters using the complete training set is called a “training epoch".

Each training iteration at time *t* using the following parameter update equation modifies the parameters (same for linear filter mask wieghts as well as for non-linear neuronal functions):

$$\theta_t = \theta_{t-1} - \alpha\delta_t$$
$$\delta_t = \nabla_\theta F(\theta_t)$$

But in contrast to neuron with fixed input data for a given data sample, the filter mask of a convolution operation moves the window over the entire input matrix!

Let's say we have 3x3 image, ***I***, and a 2x2 filter ***W***. Sliding this filter over the image will produce 2x2 output (no padding).

- The for elements of this output would be:

$$
\begin{aligned}
O_{11} &= I_{11} W11 + I_{12} W_{12} + I_{21} W_{21} + I_{22} W_{22} \\
O_{12} &= I_{12} W11 + I_{13} W_{12} + I_{22} W_{21} + I_{23} W_{22} \\
O_{21} &= I_{21} W11 + I_{22} W_{12} + I_{31} W_{21} + I_{32} W_{22} \\
O_{22} &= I_{22} W11 + I_{23} W_{12} + I_{32} W_{21} + I_{33} W_{22}
\end{aligned}
$$

- The next layer can be pooling and then this output can be fed into a dense layer, after flattening if necessary. For example, if it's average pooling with 2x2 pool size, we have a single output:

$$o = \frac{O_{11} + O_{12} + O_{21} + O_{22}}{4}$$

- If *L* is the loss function, then we get:

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial W_{11}} & \frac{\partial L}{\partial W_{21}} \\ \frac{\partial L}{\partial W_{12}} & \frac{\partial L}{\partial W_{22}} \end{bmatrix}$$

The error must be computed and accumulated for all pixels of the input image!

Batch Gradient-Descent

- Gradient descent algorithms work by computing the gradient of the objective function with respect to the network parameters, followed by a parameter update in the direction of the steepest descent.

- The basic version of the gradient descent, termed "batch gradient descent," computes this gradient on the **entire training set**.
    - It is guaranteed to converge to the global minimum for the case of convex problems.
    - For non-convex problems, it can still attain a local minimum.

- However, the training sets can be very large in computer vision problems, and therefore learning via the batch gradient descent can be prohibitively slow because for each parameter update, it needs to compute the gradient on the complete training set.

Stochastic Gradient-Descent

⚠ Stochastic Gradient Descent (SGD) performs a parameter update for each set of input and output that are present in the training set.

- As a result, it converges much faster compared to the batch gradient descent. Furthermore, it is able to learn in an “online manner”, where the parameters can be tuned in the presence of new training examples.
- The only problem is that its convergence behavior is usually unstable, especially for relatively larger learning rates and when the training datasets contain diverse examples.
- When the learning rate is appropriately set, the SGD generally achieves a similar convergence behavior, compared to the batch gradient descent, for both the convex and non-convex problems.

Gradient Computation

- A gradient ∇ can be approximated by small difference terms:

$$\nabla = \frac{\partial u}{\partial v} \approx \frac{\Delta u}{\Delta v} = \frac{u_i - u_{i-1}}{v_i - v_{i-1}}$$

- But such a difference formula tends to be very inaccurate for large gradients (not known in advance and dynamic). So analytical differentiation (of a node function) is preferred if possible.

- On the other hand, analytically deriving the derivatives of complex expressions is time-consuming and laborious. Furthermore, it is necessary to model the layer operation as a closed-form mathematical expression. However, it provides an accurate value for the derivative at each point.

Gradients of functions *f* can be computed by:

1. Numerical differentation (approximation from samples)

$$\frac{\Delta f}{\Delta x} = \frac{f(x+h) - f(x)}{h}$$

2. Analytical differentiation (for simple functions)
3. Symbolic differentation (for complex functions)
4. Programmed differentiation

Every computer program is implemented using a programming language, which only supports a set of basic functions (e.g., addition, multiplication, exponentiation, logarithm and trigonometric functions). Automatic differentiation uses this modular nature of computer pro- grams to break them into simpler elementary functions. The derivatives of these simple functions are computed symbolically and the chain rule is then applied repeatedly to compute any order of derivatives of complex programs.

[Khan, 2018]

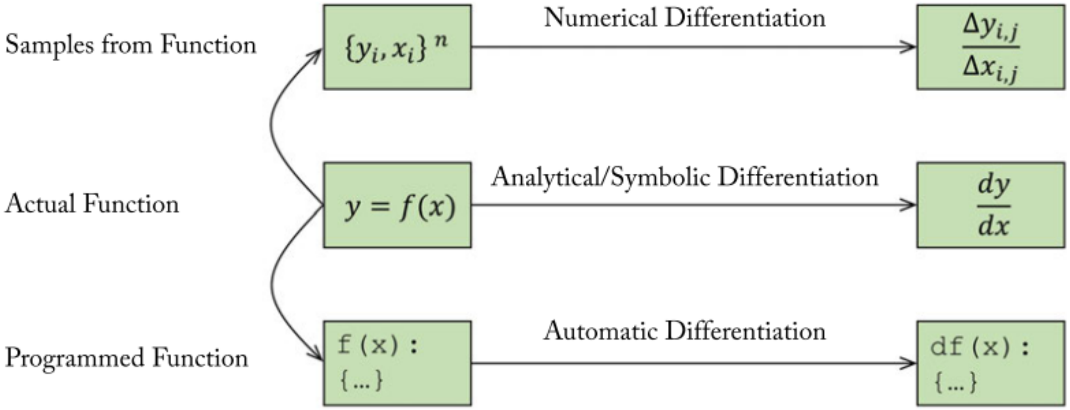

Samples from Function
Actual Function
Programmed Function
$\{y_i, x_i\}^n$
$y = f(x)$
f(x): {…}
Numerical Differentiation
Analytical/Symbolic Differentiation
Automatic Differentiation
$\frac{\Delta y_{i,j}}{\Delta x_{i,j}}$
$\frac{dy}{dx}$
df(x): {…}

Fig. 9. Relationships between different differentiation methods

Summary

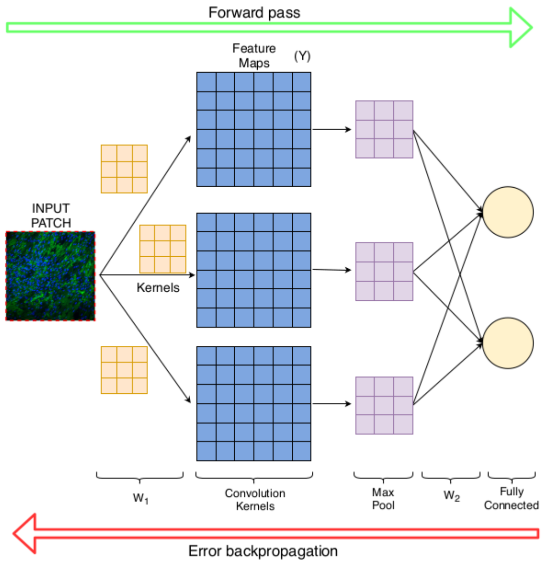


Fig. 10. Error backpropagation requires a previous forward computation to get the error and to compute the errror gradients (Bazaga et al., 2019).

Understanding CNN by Visualization

- Convolutional networks are large-scale models with a huge number of parameters that are learned in a data driven fashion.
  - Plotting an error curve and objective function on the training and validation sets against the training iterations is one way to track the overall training progress.
  - However, this approach does not give an insight into the actual parameters and activa- tions of the CNN layers.
  - It is often useful to visualize what CNNs have learned during or after the completion of the training process.

The visualization can be categorized into three types depending on the network signal that is used to obtain the visualization, i.e., weights, activations, and gradients.We summarize some of these three types of visualization methods below

Relevant Regions (ROIs)

Visualization of regions which are important for the correct prediction from a deep network.

This is an iteative method to get either an heatmap of regions to show their contribution in a classification problem or to mask out irrelevant regions.

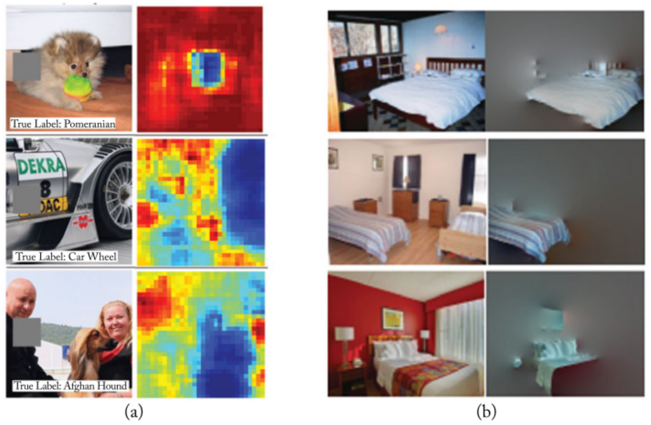


Fig. 11. (a) The grey regions in input images is sequentially occluded and the output probability of correct class is plotted as a heat map (blue regions indicate high importance for correct classification). (b) Segmented regions in an image are occluded until the minimal image details that are required for correct scene class prediction are left