# Probabilistic Machine Learning:
# 9. Bayesian Neural Networks

Tomasz Kajdanowicz, Piotr Bielak, Maciej Falkiewicz, Kacper Kania, Piotr Zieliński

Department of Computational Intelligence
Wroclaw University of Science and Technology

# Pre-reading

No pre-reading was offered.

**Disclamer**

The lecture does not go in-depth about neural networks. Expand your knowledge with arxiv.org and by a starting point: Deep Learning by Ian Goodfellow and Yoshua Bengio and Aaron Courville.

# Table of contents

# Table of Contents

# Introduction to Neural Networks

- one of the most popular tools used in machine learning
- many applications (e.g. medicine, automotive, document
- classifications, translation)
- popular through easyness of usage and application

**Math**

behind neural networks is not the easiest part, but can be trivially automated (apparatus invented in previous centuries), e.g. *chain rule for calculating derivatives* over parameters of ested functions.

Known approaches and popular approaches in NN mplemented in well-known, libraries such as:

- tensorflow
- eras (which is, in fact, a wrapper for the tensorflow)
- pytorch

# Table of Contents

# Neural Networks

**Definition**

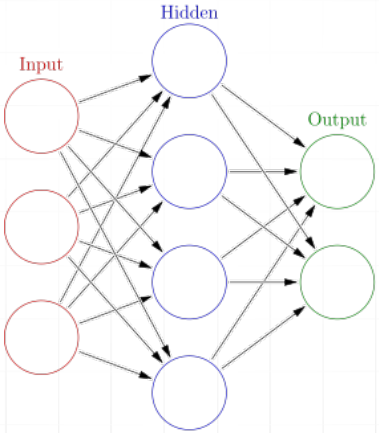
Neural networks are said to be connectionist systems that are trained in a frequentionist approach.

*Connectionism is an approach to the study of **human cognition** that utilizes mathematical models, known as connectionist networks or artificial neural networks. Often, these come in the form of highly interconnected, **neuron-like** processing units. There is no sharp dividing line between connectionism and computational neuroscience, but connectionists tend more often to abstract away from the specific details of neural functioning to focus on high-level cognitive processes (for example, recognition, memory, comprehension, grammatical competence and reasoning). During connectionism's ideological heyday in the late twentieth century, its proponents aimed to replace theoretical appeals to formal rules of inference and sentence-like cognitive representations with appeals to the parallel processing of diffuse patterns of neural activity.[1]*

# Representation

Artificial neural networks are often represented as:

- layers of circles, where each circle represents a neuron
- lines between neurons in two consecutive layers, which represent learnable parameter



# Representation - symbolic approach

- there exist a symbolic approach
- out of the scope of this course

In short, it allows manipulating symbols rather than not-that-meaningful scalars.
The equation of the neural network can be described as:

$$y_j = \sum_{i=1}^{M} w_{ij} x_i + b_i \tag{1}$$

where the $j$-th output neuron is a weighted sum of $M$ input signals with an additional shifting values $b$.

# Training

By *training*, we mean fitting network parameters.

- we initialize these parameters randomly and let known algorithms to optimize these parameters such that a general criterion $\mathcal{L}$ is minimized
- a criterion can be described as "How wrong our neural network is about some decision"
- the optimization tells how the network should correct its parameters to perform better
- it is performed through gradient descent $\nabla$ of parameters $\theta$ over criterion $\mathcal{L}$
- it can be seen as going down the slope of the loss function landscape

$$\nabla_\theta \mathcal{L} \tag{2}$$

# Training

An example loss landscape for two parameters. During our optimization, we search for a global optimum (blue areas) and the network "descents" on that landscape.

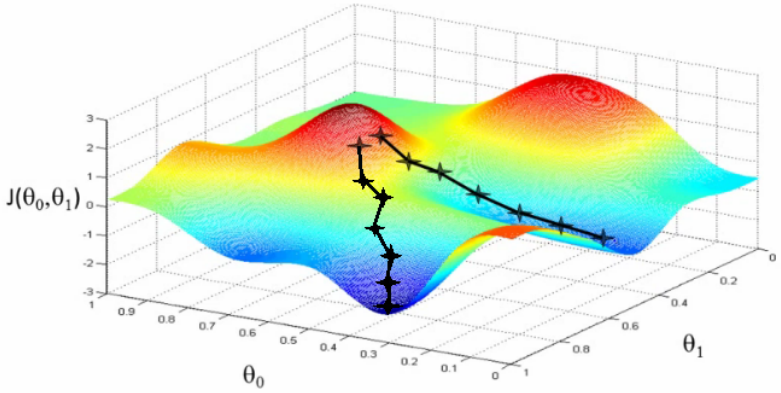



Figure: Credits to Andrew Ng (Gradient Descent lecture)

# Frequentionist vs bayesian and optimization

- frequentionist is only a part of the bayesian
- in the bayesian framework, we encode our prior knowledge about model parameters before seeing any data

With the right hit, we could achieve high accuracy without any training. However, in the frequentionist approach, we derive learning from the maximum likelihood estimation of the parameters.
It means maximizing the probability of the data, given the model parameters:

$$\arg\max_{\theta} p(\mathcal{D}|\theta) \tag{3}$$

# Calculations in training

- calculating is not trivial and almost always impossible to find an analytical solution
- we opt for optimizing by using gradient descents
- we are looking for a set of parameters, where the gradient is equal to 0:

$$\nabla_\theta p(\mathcal{D}|\theta) = 0 \tag{4}$$

From that equation, we are able to derive a *weight update* $\Delta\theta_i$ that improves the performance of the neural network, when the update is applied.

# Table of Contents

# How to measure the goodness of the Neural Network?

- to measure how bad or good our neural network is, we need to define a *criterion*.
- generally, int measures performance

There are few requirements to be met by a function so that the function can be considered a proper criterion for gradient optimization:

- it needs to be at least $C^1$ differentiable (it has at least one derivative)
- it has to be a minimized function, so the best parameters are obtained through $\arg\min_\theta \mathcal{L}$ (note, that for maximized functions, we can simply use $-\mathcal{L}$)
- choice of criterion depends on the application of our network

# Common criterions

For a ground truth $y_i$ and network's prediction $\hat{y}_i = f_\theta(x_i)$ for a $i$-th sample, we can use:

1. squared error (for the regression task)

$$\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2 \tag{5}$$

   *it measures how far our prediction is from the ground truth; the higher the difference, the higher the loss value*

2. binary crossentropy (for the classication task with binary class only):

$$\mathcal{L}(y, \hat{y}) = -y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \tag{6}$$

   *origins in information theory; the lower the entropy, the more we know about the system; the smallest possible value is obtained when, intuitively,* $y_i = 1 \wedge \hat{y}_i = 1$ *and* $y_i = 0 \wedge \hat{y}_i = 0$.

# Common criterion (2)

3. categorical crossentropy (for the classification task with many exclusive classes)

$$\mathcal{L}(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k \tag{7}$$

The loss assumes that both vectors $y$ and $\hat{y}$ follows the definition of the *proper probability distribution* (values in range [0, 1], and sum to 1), which is typical for the softmax outputs (one of the activation functions). When a particular activation is applied, the gradient of that function simplifies significantly, which stabilizes learning. Features of that loss are the same as for its binary equivalent.

Having one of these losses chosen, we can train our neural network.

# Table of Contents

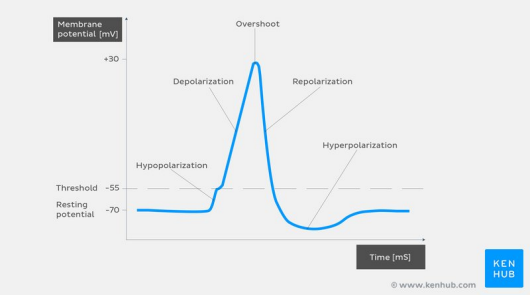
# Activation functions

**Activations**

are simple, non-linear functions that transform input data into output data with the same dimensions.

Activations were introduced mainly for two reasons:

1. It is proven, that biological neural network, *perform some non-linearity* on the input signal coming from the input dendrites and the potential action is measured as a function over the sum of the input signals



# Activation functions (2)

The biological neuron aggregates the signal, and when the aggregation hits a specific threshold value, the neuron "unlocks" and passes the signal to the synapse (output).

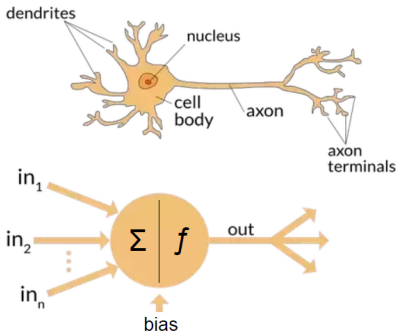



Figure: Credits medium.com

In the era of biological analogies (1950s), the activation function followed the idea of the threshold function.

# Activation functions

2. strictly mathematical reasons of activation functions
   - weighted sum is just a linear operation
   - stacking multiple layers that perform the same weighted sum with different sets of weights is also a linear operation
   - it implies that multiple layers can be combined into a single layer
   - causes, that problems that are *not linearly separable* cannot be solved

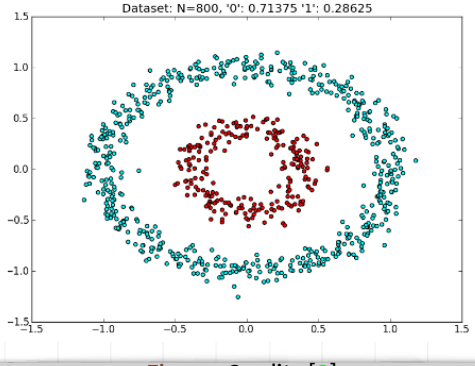



Figure: Credits [2]

# Non-linear separability

**Non-linear separability**

means that there is no such an $N + 1$ dimensional hyperplane or a plane that can split the $N$ dimensional data into two classes

We have to introduce some non-linearity to neural networks. When stacked in multiple layers, they can approximate any function (theoretically).
Activation function:

- for the criterion must be differentiable
- there are some exceptions, but we will not dive into them

Some popular activations (denoted in general as $\sigma(x)$):

1. Sigmoid
2. Hyperbolic tangent (tanh)
3. Rectified linaer unit (ReLU)
4. Softmax

# Sigmoid

Equation: $\sigma(x) = \frac{1}{1+e^{-x}}$
Values range: $[0, 1]$
Good for:

- binary classification
- Bernoulli probability description
- before ReLU, the sigmoid was a standard for hidden units

Shape:

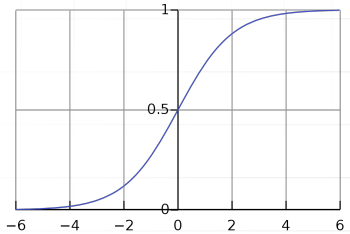



Figure: Credits to Wikipedia

Bad for:

- deep neural networks (vanishing gradient problem)
- time sensitive applications (in fact, many such sigmoids is time consuming to calculate for low-end hardware)

# Hyperbolic tangent (tanh)

Shape:

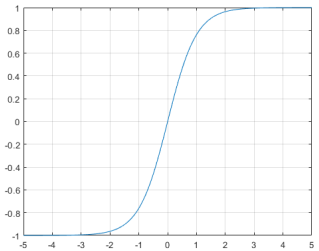



Figure: Credits to Mathworks

Equation: $\sigma(x) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
Values range: $[-1, 1]$
Good for:

- binary classification for $y \in \{-1, 1\}$
- hidden units bounded on both sides (due to symmetrical nature), for example in recurrent neural networks

Bad for:

- deep neural networks (vanishing gradient problem)
- time sensitive applications (in fact, many such sigmoids is time consuming to calculate for low-end hardware)

# Rectified linaer unit (ReLU)

Equation: $\sigma(x) = \max(0, x)$
Values range: $[0, \infty]$
Good for:

- deep neural networks (no vanishing gradient)
- bounded linear regression

Shape:

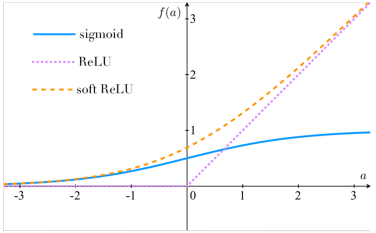



Figure: Credits to [3]

Bad for:

- badly defined neural network (exploding gradient)
- very deep neural networks (more than 20 layers) due to dying neurons (activation equal to 0 due to negative values of the neuron)

# Softmax

Equation: $\sigma(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

Values range: $[0, 1]$

Good for:

- last layer as a multiclass classification
- describing probabilities of the multinomial distribution
- the best with categorical crossentropy

Bad for:

- hidden units activation

Shape:

(no explicit shape due to dependence on a scale of other values)

# Other activation functions

- many more
- the above mentioned in 99% of the papers about different NN architectures
- some modifications to the standard activation function can be applied (e.g. using some alterations of the ReLU to mitigate the problem of dying neurons - LeakyReLU that allows negative values to pass through neural network with activation as $\max(\alpha x, x)$)

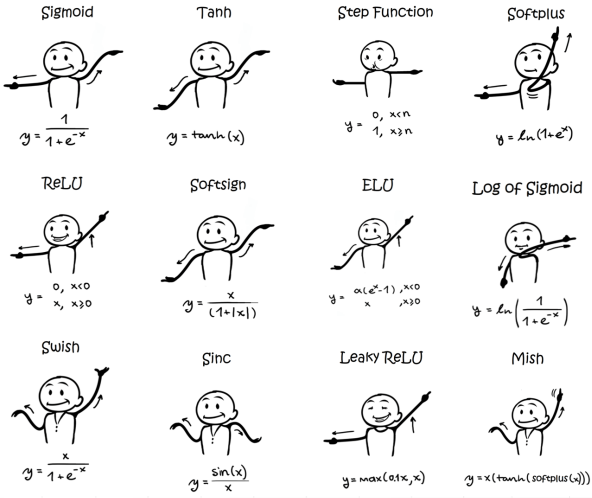



Figure: Credits to https://sefiks.com/2020/02/02/dance-moves-of-deep-learning-activation-functions/

# Table of Contents

# So how to learn Neural Network ?

We can derive equations necessary to learn our neural networks. Assumptions:

- for neural network presented in the first figure
- training using stochastic gradients descent
- hidden layer and the output layer are activated with the sigmoid function

*Stochastic means that we take few samples at once calculating gradients for them, average gradients over these samples, and then apply as corrections to learnable parameters. These batches of data are called mini-batches.*

# Backpropagation algorithm

The neural networks consists of:

- $N$ input neurons
- $M$ hidden neurons
- $K$ output neurons

Each mini-batch consists of $B$ samples. Weights $w$ and $v$ are learnable. Output of the network is described then as:

$$\hat{y}_{bk} = \sigma_{bk}\left(\sum_{m=1}^{M} v_{mk}\sigma_{bm}\left(\sum_{n=1}^{N} w_{nm}x_{bn}\right)\right)$$

- indices $k$ means the $k$-th of the sample
- all equations are applied independently to each sample in the mini-batch
- calculated gradients of weights are averaged over samples in the mini-batch
- look for dropped biases, but gradients for biases are derived in an analogical way as for weights

# Backpropagation algorithm (2)

Now, we need a criterion. Let's say that the task is a binary classification task (hence, the sigmoid function in the last layer). We will use binary crossentropy:

$$C = \frac{1}{2B} \sum_{b=1}^{B} \sum_{k=1}^{K} (y_{bk} - \hat{y}_{bk})^2 \tag{8}$$

where:

- $y_{bk}$ are ground truth classes encoded as one-hot vectors. It means that 3rd class out of possible 5 classes is encoded as a vector: $[0, 0, 1, 0, 0]$

# Backpropagation algorithm (2)

To update weights $w$ and $v$, we need gradients of these weights over defined criterion:

$$\frac{\partial C}{\partial w_{nm}}, \frac{\partial C}{\partial v_{mk}} \tag{9}$$

- direct computing these gradients would require a brilliant mind and great memory
- to overcome the lack of these abilities, we use the chain rule, that help us decompose heavy calculation into simpler ones
- we drop $b$ indices for notational simplicity
- gradients are calculated for each sample independently and averaged over samples in the mini-batch (this can be proven by including all the indices)

# Backpropagation algorithm (3)

$$\frac{\partial C}{\partial v_{mk}} = \frac{\partial C}{\partial \sigma_k} \frac{\partial \sigma_k}{\partial \delta_k} \frac{\partial \delta_k}{\partial v_{mk}}$$

$$\frac{\partial C}{\partial w_{nm}} = \frac{\partial C}{\partial \sigma_k} \frac{\partial \sigma_k}{\partial \delta_k} \frac{\partial \delta_k}{\partial \sigma_m} \frac{\partial \sigma_m}{\partial \delta_n} \frac{\partial \delta_n}{\partial x_n}$$

where

- $\sigma_k$ and $\sigma_m$ denotes activated hidden and input units respectively for $b$-th sample
- $\delta_k$ and $\delta_m$ are raw values (before the activation function)

At this point, we forgot that in the forward equation to get $\hat{y}_k$, we had sums. We have to include them in these chains. Notice, that some variables do not depend on these sums, for example, $w_{nm}$ does not depend on the sum over $K$ classes. So full equations can be written as:

$$\frac{\partial C}{\partial v_{mk}} \rightarrow \text{unchanged}$$

$$\frac{\partial C}{\partial w_{nm}} = \frac{\partial \sigma_m}{\partial \delta_n} \frac{\partial \delta_n}{\partial x_n} \sum_{k=1}^{K} \frac{\partial C}{\partial \sigma_k} \frac{\partial \sigma_k}{\partial \delta_k} \frac{\partial \delta_k}{\partial \sigma_m}$$

# Backpropagation algorithm (4)

Having these equation simplified, we can calculate derivative for each component. We recall that a derivatives for each equation are as follows:

- sigmoid: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- tanh: $1 - \tanh^2(x)$
- - relu:

$$\sigma'(x) = \begin{cases} 1, & x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

- softmax:
  for the equation:

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

  it equals to:

$$\sigma'(x_i) = \begin{cases} x_i(1 - x_i), & i = j \\ -x_i x_j, & i \neq j \end{cases}$$

# Backpropagation algorithm (5)

No we have all components to solve these derivative chains. We assume that $\hat{y}_k = \sigma_k$ and both $\sigma_k$ and $\sigma_m$ are sigmoid functions.

$$\delta_m = \sum_{n=1}^{N} x_n w_{nm} \tag{10}$$

$$\delta_k = \sum_{m=1}^{M} v_{mk} \sigma_m \left(\delta_m\right) \tag{11}$$

$$\frac{\partial C}{\partial \sigma_k} = \frac{\partial}{\partial \sigma_k} \frac{1}{2} \sum_{k=1}^{K} \left(y_k - \sigma_k(\delta_k)\right)^2 = -\sigma_k(\delta_k) \left(y_k - \sigma_k(\delta_k)\right) \tag{12}$$

$$\frac{\partial \sigma_k}{\partial \delta_k} = \sigma_k(\delta_k)(1 - \sigma_k(\delta_k)) \tag{13}$$

$$\frac{\partial \delta_k}{\partial v_{mk}} = \frac{\partial}{\partial v_{mk}} \sum_{m=1}^{M} v_{mk} \sigma_m(\delta_m) = \sigma_m(\delta_m) \tag{14}$$

# Backpropagation algorithm (6)

At this point we have, gradients ready to be applied for weights $v$. We can further derive updates for $w$.

$$\frac{\partial \delta_k}{\partial \sigma_m} = \frac{\partial}{\partial \sigma_m(\delta_m)} \sum_{m=1}^{M} v_{mk}\sigma_m(\delta_m) = v_{mk} \tag{15}$$

$$\frac{\partial \sigma_m}{\partial \delta_m} = \sigma_m(\delta_m)(1 - \sigma_m(\delta_m)) \tag{16}$$

$$\frac{\partial \delta_m}{\partial w_{nm}} = \frac{\partial}{\partial w_{nm}} \sum_{n=1}^{N} x_n w_{nm} = x_n \tag{17}$$

# Backpropagation algorithm (7)

Now, we are ready to apply these partial derivatives to the chains mentioned above.

$$\frac{\partial C}{\partial v_{mk}} = -\sigma_k(\delta_k)(y_k - \sigma_k(\delta_k))\sigma_k(\delta_k)(1 - \sigma_k(\delta_k))\sigma_m(\delta_m) \tag{18}$$

$$\frac{\partial C}{\partial w_{nm}} = \sigma_m(\delta_m)(1 - \sigma_m(\delta_m))x_n \sum_{k=1}^{K} -\sigma_k(\delta_k)(y_k - \sigma_k(\delta_k))\sigma_k(\delta_k)(1 - \sigma_k(\delta_k))v_{mk} \tag{19}$$

# Backpropagation algorithm (8)

- in the same way, we can derive full updates for any neural network architecture with any loss function and any activation function
- framework comes from the requirement that each function is differentiable
- implmentation through 'for' loops in a programming language of choice?
- most programming libraries use matrix notation, and these sums can be easily replaced with dot products - *we can calculate multiple gradients at once*

**Ok, but how can we use these partial derivatives?** The general formula for a weight update from a step $t$ in step $t+1$ is described as:

$$w^{t+1} = w^t - \alpha \frac{\partial C}{\partial w}$$

where

- $\alpha$ is a learning rate

This is a standard update formula for gradient descent. Stochastic gradient descent would include averaging (or summing, depending on a framework) over $B$ samples. Many other adaptive optimizers account for additional derivate moments (averages over last $S$ steps) to provide so-called momentum and stabilize learning. These optimizers are often a standard choice.

# Further references

We ask you to refer to the following works if you would like to learn more about them:

1. An overview of gradient descent optimization algorithms - one of the most complete overview
2. ADADELTA: An Adaptive Learning Rate Method
3. Overview of mini-batch gradient descent - it is said that RMSprop was invented during this lecture by prof. Hinton
4. Adam: A Method for Stochastic Optimization

# Backpropagation algorithm (Summary)

Now, we have to just plug these equations as:

$$v_{mk}^{t+1} = v_{mk}^{t} - \alpha \frac{\partial C}{\partial v_{mk}} \tag{20}$$

$$v_{nm}^{t+1} = v_{nm}^{t} - \alpha \frac{\partial C}{\partial w_{nm}} \tag{21}$$

We remind that the algorithm above is called backpropagation and is widely used for neural networks to this date. $\alpha$ is a hyperparameter that has to be tuned manually.

# Table of Contents

# What about deeper networks?

- it would be daunting to derive all partial derivates for deeper neural networks
- e.g. complex neural architectures with over 100 layers, 4 million parameters
- multiple parallel branches that are joined at some stage to get the final prediction

**Idea:**

**Simplification**

simplyfy the equations(computations) and left just additions and multiplications of scalars, without any functions such as a sigmoid

Then, we would have to write simple rules into nodes of the graph, where each node represents an atomic operation, such as an addition, and edges represent derivates of nodes. Programmatically, it is achievable in a short amount of time.
Current frameworks use such simplifications and automation in their auto differentiating methods. This way, the only thing that we have to implement is the forward inference. The backward pass is calculated automatically. But how these methods are constructed. Let us introduce directed acyclic graphs of computation.

# Directed acyclic graphs of computation

We start with a simple function. We want to calculate something like:

$$y = A(x)$$

and for simplicity, introduce some intermediate variable $a = A(x)$ that will hold the value before returning it to $y$. Now, we build our computation graph:

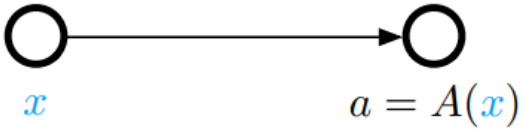



Figure: Credits to Matthew Johnson (Automatic Differentiation lecture)

# Directed acyclic graphs of computation(2)

We want to calculate $\frac{\partial y}{\partial x}$. We proceed as follows:

1. Start with calculating $\frac{\partial y}{\partial y} = 1$
2. Then, calculate $\frac{\partial y}{\partial a} = 1$ (because it is an intermediate result $= y$)
3. End with $\frac{\partial a}{\partial x} = A'(x) \cdot 1$

All these computations are multiplied to get our final: $\frac{\partial y}{\partial x} = 1 \cdot 1 \cdot A'(x)$. The final graph is as follows:

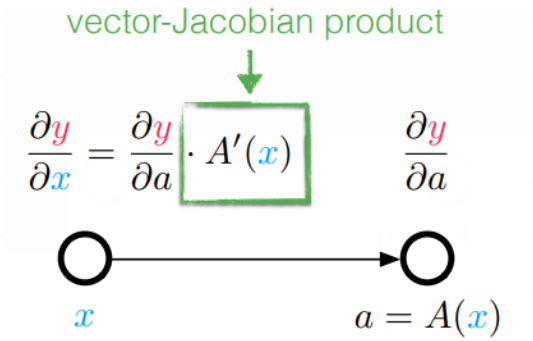



Figure: Credits to Matthew Johnson (Automatic Differentiation lecture)

# Directed acyclic graphs of computation(3)

To summarize:

1. Nodes are simple operations.
2. Edges are derivates of nodes on each end.
3. Derivatives in a sequence of edges are multiplied.
4. Derivatives in parallel branches joined together, are summed.

These graphs can be arbitrarily complex, but local computations are still simple.

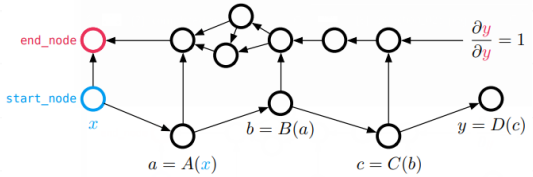



Figure: Credits to Matthew Johnson (Automatic Differentiation lecture)

# Other computation methods

There are also different approaches on how to calculate a jacobian efficiently (row-wise or column-wise). Still, we will not dive into them (you will these during your 2nd semester in Deep Learning course).

To understand it further, please checkout the code by Andrey Karpathy (CTO of Tesla and one of the pioneers in deep learning). The author implemented the autograd in a minimal number of LOC.

Also, for more information, we ask you to refer to this lecture.

# Other computation methods

It is worth mentioning that different frameworks implemented these graphs differently.

- TensorFlow, before version 2.0, used static computation fact. That way, the TensorFlow was the fastest library, and it was easy to distribute computation over a cluster.
- Nowadays, both PyTorch and TensorFlow 2.0, derive the graph dynamically by inspecting call stack
- Each operation, even ‘+‘ and ‘-‘ causes that the framework handles this operation and registers it in its own graph
- Each simple operation is overloaded (for example, PyTorch uses ‘Function‘ class), so it implements both forward and backward pass

# Table of Contents

# Overfitting

When overfitting occurs, our model performs well on the data that *it was trained on*, but performs poorly any *testing data*, that model did not see.

To visualize it: imagine that you obtained a dataset that consists of 1000 samples. You built a neural network that has 10 000 learnable parameters. Theoretically, each neuron "encoded" one sample, so it serves as a switch - if the example occurs, it switches on and switches off otherwise. For new samples, that were not available in the dataset, the model will crash (ex. return wrong classification decision), since it does not have any switches for these samples.

# ...and underfitting

*Underfitting* occurs when our model has so few parameters (there are also other factors, of course), that it is unable to match any point.

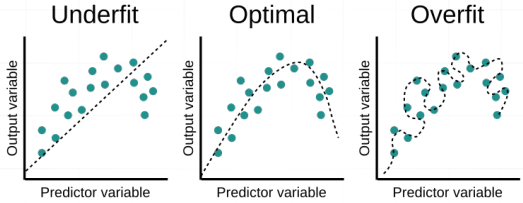



Figure: Credits to www.educative.io

Preventing overfitting comes from *regularization* techniques.

## Regularization

Regularization means reducing degrees of freedom, so the model, theoretically, generalizes better to unseen data.

You have already met a few of them ($L_1$ and $L_2$ regularization), but the list is long.

## $L_1$

$L_1$ reduces magnitudes of weights $w$. We introduce it in a loss function as ($N$ learnable parameters):

$$\mathcal{L}_\lambda = \mathcal{L} + \lambda \sum_{i=1}^{N} |w_i|$$

where $|\cdot|$ means taking an absolute value. By deriving the gradients of that loss, we can notices what happens to weights during an update:

$$\frac{\partial L_1}{\partial w_i} = \lambda \frac{w_i}{|w_i|} \tag{22}$$

$$w_i^{t+1} = w_i^t - \alpha \frac{\partial C}{\partial w_i} - \lambda \frac{w_i}{|w_i|} \tag{23}$$

Such a loss leads to the network sparsification, since many parameters will be led to 0. $\lambda$ is a hyperparameter that has to be tuned manually.

## $L_2$

Less restrictive loss, is an $L_2$ regularization. It is defined as:

$$\mathcal{L}_\lambda = \mathcal{L} + \lambda \sum_{i=1}^{N} w_i^2$$

Similarily, it reduces parameter's magnitude but the strength of the regularization is decreases with the magnitude of the weight since:

$$\frac{\partial L_1}{\partial w_i} = \frac{\lambda}{2} w_i \tag{24}$$

$$w_i^{t+1} = w_i^t - \alpha \frac{\partial C}{\partial w_i} - \frac{\lambda}{2} w_i \tag{25}$$

We can see that it does not lead to zeroing weights.

# Dropout

The technique was first described in Dropout: A Simple Way to Prevent Neural Networks from Overfitting.

It works as follows:

1. We sample a mask *m* from a Bernoulli distribution with a probability *p*. The mask is the size of the input (for *x* of size 16, we create a mask of 16 elements consisting only 0s and 1s)
2. We multiply the mask by the mask *m* and pass values forward.
3. Since we changed the total distribution of activations (to be more precise - decreased their sum), we have to divide all activations by *p* to mitigate to the problem. Otherwise, the learning would be unstable.
4. During backpropagation, we pass the signal through neurons that stayed activated during the forward inference.
5. For testing, we disable dropout like it never existed.

# Dropout (2): Importance

- neurons are highly correlated
  Let us explain why it is necessary. During the learning of a standard network, we sample initial weights from some distribution (commonly, normal). If weights are almost the same in value and are responsible for features that are correlated, they will be updated in the same fashion. In short, it causes neurons to be highly correlated, thus making them redundant and prone to overfitting. The dropout decorrelates these weights because sometimes one weight is updated, while the others are not.
- correlated networks
  Another interpretation is that we train multiple neural networks at once, that are correlated (correlation of networks is not the same as correlation of particular weights). This creates an ensemble of neural networks that, theoretically, should increase generalization power by variance reduction. We can, of course, hold the variance to be at a similar lever by increasing the number of learnable weights.

Even though the method was invented in 2014, it, together with its variants, is commonly used to this date.

# Final NN remarks

We can encourage you to read more about neural networks on your own. One reason is that these are a method of choice in the presence of high volumes of data. Also, they drop the necessity to analyze datasets in depth, since they learn automatically what features are necessary and how they are correlated.

- Feedforward Networks and Backpropagation
- Deep Learning. Deep Feedforward Networks

# Table of Contents

# Bayesian Neural Networks

We can proceed with bayesian neural networks. Their main advantages are two:

- we can measure uncertainty about our predictions with a different confidence level
- we can encode prior knowledge about weights distribution (in principle, it is harder than it sounds)

# Table of Contents

# Bayesian approach

Recall, that for normal neural networks, we used a maximum likelihood estimation of parameters that would maximize the probability of producing the data under these parameters:

$$\arg\max_{\theta} p(\mathcal{D}|\theta)$$

However, this formulation leads to obtaining point estimates of parameters. In short, it means no uncertainty measure. Fortunately, we can use Bayes theorem to derive equations necessary to learn distribution of parameters:

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})}$$

# Bayesian approach (2)

This is our posterior distribution of parameters, and $p(\theta)$ is our prior belief about these parameters. Using bayesian inference, we can obtain predictions by integrating over all parameters $\theta$

$$p(y|x, \mathcal{D}) = \int p(y|x, \theta) p(\theta|\mathcal{D}) \, d\theta$$

As you can see, these equations are intractable since there is an infinite number of configurations of parameters, and we do not know the distribution of the data.

# Bayesian approach(3)

- We already know a few algorithms that can deal with, for example, Monte Carlo estimation (see Practical Variational Inference for Neural Networks). The article was published in 2011.
- In 2014, Stochastic Variational Inference was introduced, and we will look deep into this idea. You already used stochastic variational inference algorithms during previous tasks, but treated it as a black box. Its power comes from introducing variational distributions, that can be parameterized with learnable parameters (through the reparametrization trick).

# Table of Contents

# What is a Variational Method?

**Definition**

Variational methods are a general family of methods for approximating complicated densities by a simpler class of densities. They turn inference into optimization.

# Variational Inference

Variational Inference:

- Suppose we have some data $\mathcal{D}$, and some latent variables $z$ (e.g. paramaters neural network)
- We're interested in doing posterior inference over $z$
- This would consist of calculating:

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})} = \frac{p(\theta,\mathcal{D})}{p(\mathcal{D})} = \frac{p(\theta,\mathcal{D})}{\int_{\theta'} p(\theta',\mathcal{D})}$$

- the numerator is easy to compute for given $\theta$, $\mathcal{D}$
- the denominator is, in general, intractable

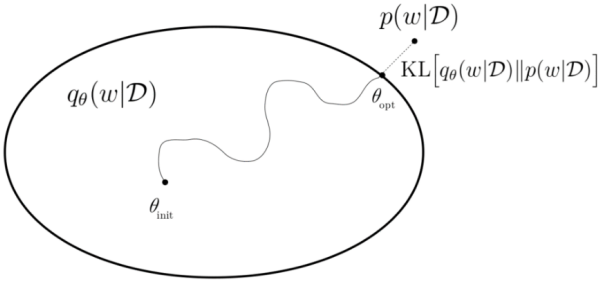
# The Variational Distribution

- Rather than calculate the posterior $p(\theta|\mathcal{D})$ exactly, we can approximate it with some distribution $q$
- The approximating distribution $q$ is called the variational distribution
- We'll choose $q$ to have some nice form, so that we can feasibly calculate $q(\theta|\mathcal{D})$

# KL Divergence

**$p$ and $q$**

Since $q$ is intended to approximate $p$, we want them to be as similar as possible.




**Kullback–Leibler**

We can measure this in many ways - the most common is KL divergence

$$\mathrm{KL}\left[q_\phi(\theta|\mathcal{D})||p(\theta|\mathcal{D})\right] = \int q_\phi(\theta|\mathcal{D}) \log \frac{q_\phi(\theta|\mathcal{D})}{p(\theta|\mathcal{D})}\, d\theta$$

We can't calculate this expression, since we don't have $p(\theta|\mathcal{D})$. But can handle it in a batch manner by sampling weights once (do not have to integrate over all possible configurations) and calculate the KL for a batch of data.

We can also perform monte carlo estimation by running multiple samplings of weights. This will lead to an unbiased estimator of weights. Summarizing, finding optimal weights can be described as:

$$\phi_{opt} = \arg\min_{\phi} \sum_{i=1}^{n} \log q_{\phi}(\theta^{(i)}|\mathcal{D}) - \log p(\theta^{(i)}) - \log p(\mathcal{D}|\theta^{(i)})$$

where $n$ are $n$ monte carlo samples and $\theta^{(i)}$ are sampled from $q_{\phi}(\theta|\mathcal{D})$. Notice, that the last term is our likelihood estimate from a standard neural networks.

# Table of Contents

# Reparametrization trick

The variational parameterization above is convenient; however it lacks differentiability that is required in neural networks. Therefore, we introduce reparametrization trick that can be easily incorporated in bayesian neural networks with a gaussian parametrization (refer to Variational Dropout and the Local Reparameterization Trick for more information). In such a framework, each learnable parameter consists of two parameters: the mean and variance of a gaussian.

$$\phi = (\mu, \sigma^2)$$

We apply it then to reparametrization framework:

$$f(\epsilon) = \theta = \mu + \sigma \cdot \epsilon$$

where:

$$\epsilon \sim \mathcal{N}(0, 1)$$

# Reparametrization trick

The only nondifferentiable part is $\epsilon$, but we can care in fact about $\mu$ and $\sigma$. Thus, we can derive gradients as (note that these are automatically derived in the autograd framework):

$$\Delta_\mu = \frac{\partial f}{\partial \theta} + \frac{\partial f}{\partial \mu} \tag{26}$$

$$\Delta_\sigma = \frac{\partial f}{\partial \theta}\frac{\epsilon}{\sigma} + \frac{\partial f}{\partial \sigma} \tag{27}$$

And update as:

$$\mu^{(t+1)} = \mu^t - \alpha\Delta_\mu \tag{28}$$

$$\sigma^{(t+1)} = \sigma^t - \alpha\Delta_\sigma \tag{29}$$

This framework is called *Bayes by Backprop* and is a core component to introduce bayesian neural networks.

# Table of Contents

We have already introduced a few concepts that are popular among the neural networks community. These include standard neural networks and their bayesian formulation. However, training bayesian neural nets can be daunting: you have to implement initialization carefully and store twice as many parameters. It also requires retraining (so no knowledge transfer between neural networks can happen), and the learning process has high variance.

To tackle these problems Yarin Gal and Zoubin Ghahramani in "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning" asked the following question: is it possible to approximate the bayesian behavior using existing well-established techniques? Look for practice session notebooks for details.

# Table of Contents

# Dropout in BNN

We have already shown you how the dropout technique works. It bases on sampling mask from the bernoulli mask with a probability $1 - p$ as $M \sim B(1 - p)$ and multiply layers inputs by that mask as $\hat{x} = x \odot M$, where $\odot$ is a hadamard product (elementwise multiplication). This operation is performed during training and turned off during validation. What if we left the dropout during the validation?

It turns out that we get other architecture of the same neural network. Thanks to the possibility of multiple samplings, we get a Gaussian Process with a covariance function marginalized over its weights.

# Table of Contents

# Uncertainty

For recall, the predictive distribution for a pair $(x, y)$ of test point and weights $\theta$ is equal to:

$$q(y|x) = \int p(y|x;\theta)q(\theta)\,d\theta$$

The equation can be understood as the integration over multiple instances of $\theta$, where multiple of these instances are obtained by multiple forward passes with dropout turned on. We have to find the first two raw moments (mean and standard deviation) to obtain the full uncertainty. These can be easily approximated through Monte Carlo sampling with $T$ samplings of bernoulli mask as:

$$\mathbb{E}_{q(y|x)} \approx \frac{1}{T}\sum_{t=1}^{T}\hat{f}(x;\theta^t)$$

where $\hat{f}$ is our neural network. Predictive variance however is equal to:

$$\mathrm{Var}_{q(y|x)}(y) \approx \tau^{-1} + \frac{1}{T}\sum_{t=1}^{T}\hat{f}(x;\theta^t)^2 - \mathbb{E}^2_{q(y|x)}$$

# uncertainty (2)

Note that these equations are simplified versions for the case when only a single scalar is predicted. Fortunately, if no dependencies occur between classes, then each predicted scalar in a vector can be treated independently (ex. for categorical prediction). For full derivation, please refer to the original work.

Taking advantage of the central limit theorem, we can see that these equations lead to a description of normal distributions, where in fact, each forward inference uses bernoulli distribution. This where $\tau$ appears - square root of its inverse is our width of the normal distribution of multiple bernoulli trials. Recall that $\tau$ is an inverse of the variance. It is described as:

$$\tau = \frac{(1-p)l^2}{2N\lambda}$$

where $p$ is a probability of the dropout, $N$ is the number of samples in the training data, $\lambda$ - $L_2$ strength, and $l$ - a prior scale of the distribution. Notice that with the increase of data, width decreases since we are given more data, and we are more sure about the predicted values.

Note, that both moments simplify to normal mean and variance calculation of predictions. Finally, we can obtain log-likelihood of the model as:

$$\log p(y|x;\theta) \approx \text{logsumexp}\left(-\frac{1}{2}\tau(y-y^t)^2\right) - \log T - \frac{1}{2}\log 2\pi - \frac{1}{2}\log \tau^{-1}$$

logsumexp trick is a mathematical equation, used in summing probabilites for a numerical stability. It is defined as:

$$\text{logsumexp}\left(-\frac{1}{2}\tau(y-y^t)^2\right) = \log\left(\sum_{t=1}^{T}\exp\left(-\frac{1}{2}\tau(y-y^t)^2\right)\right)$$

It is implemented ‘pytorch‘ in method ‘torch.logsumexp(...)‘

# Bibliography I


[1] *Connectionism | Internet Encyclopedia of Philosophy*. https://www.iep.utm.edu/connect/. [Online; accessed 2020-05-08].

[2] *Data Science Notebook*. https://sites.google.com/site/datasciencenotebook1/. [Online; accessed 2020-05-08].

[3] Chao Zhang. "Joint Training Methods for Tandem and Hybrid Speech Recognition Systems using Deep Neural Networks". PhD thesis. University of Cambridge, 2017.