Martin Isaksson

Bayesian Approach to Deep Convolutional Neural Networks

Barcelona, Spain

Martin Isaksson
2017-01-06

Table of Contents

Introduction

Within the subject of data mining, an important task is to find patterns in the data. To do this, an adequate process should be considered, see *Figure 1*.

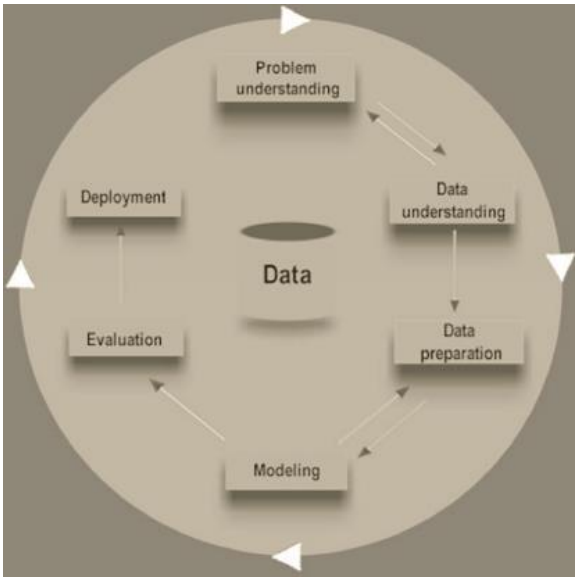

Figure 1 The process in data mining to consider, this report focus mostly on the data preprocessing (preparation) and modeling.

Before entering the model to be used for pattern recognition, the importance of preprocessing the data has to be carefully considered. The preprocessing of the data depends on the other hand of the task, the sort of data and what kind of model that is going to be used.

This report will focus on the model-part of the data mining process. More specifically, an investigation of a Bayesian approach to deep convolutional neural networks (CNN) will be performed. To end up at that investigation, some earlier yet important, concepts will be explained first.

The Bayesian approach to neural networks was explored in the early 90's and showed great results. What was made was to make artificial neural networks (ANN) more efficient with clever data preprocessing and also to find a way to choose model architecture, hyper-parameters and weights of the network in a much faster way than before.

The Bayesian approach to CNNs was investigated during the year 2015 and an article was published about that in 2016, which showed promising results. It is not hard to understand from this that the topic is very new and hopefully will be furthered explored to make CNNs more efficient.

CNNs consist of much more hyper-parameters than ANNs, so the Bayesian approach to CNNs is not a direct translation of the Bayesian approach to neural networks. The Bayesian approach to CNNs is about the weights only to being able to perform good results with small datasets.

Both ANNs and CNNs are often used for classification and pattern recognition of the input datasets. CNNs are preferred over ANNs when the input data are images. This is because of several reasons, but the perhaps most important one is that CNNs can work with much less parameters compared to ANNs (if the input are images).

Bayesian probability is the use of random variables (or unknown quantities) to model sources of uncertainty in statistical models. Or as in the case of hyper-parameters, the uncertainty resulting from lack of information.

The areas of neural networks, Bayesian probability and CNNs are huge and they include too many details that can be explained. Therefore, only a brief description of the areas will be carried through and the focus will lie on the data preprocessing, Bayesian ANNs and the Bayesian approach to CNNs.

The ANNs and CNNs that will be discussed are feed-forward networks.

Artificial Neural Networks

The human brain can be seen as a supercomputer, with its primary visual cortex (also known as V1) containing 140 million neurons and tens of billions of connections between them, and the rest of the visual cortices (V2, V3, V4 and V5) which are doing more complex image processing. It is no wonder that the human race has a habit of seeing patterns in most of things (even if there exist none).

History

The development of Artificial neural networks first started in 1943 when Warren McCulloch and Walter Pitts tried to figure out how the brain works. The first model was built as an electrical circuit of a simple neural network.

In 1949 Donald Hebb discovered that the more time a neural path is used, the more it will be strengthened.

The first system ever built for commercial use was MADALINE (the none commercial system which did the same thing was called ADALINE), a work done by Bernard Widrow and Marcian Hoff from Stanford in 1959. The system read streaming bits from a phone line, and from that it could predict the next bit.

Despite the later success of neural nets, the traditional von Neumann architecture took over the computing scene.

In 1972 Kohonen and Anderson developed a neural network that used matrices. Without knowing it at the time, they were creating an array of analog ADALINE circuits. The difference was that it activated a set of outputs instead of just one.

The first multilayer perceptron (MLP) network was an unsupervised network created in 1975.

Thanks to John Hopfield of Caltech, and a paper he published in 1982, the interest in the field was renewed.

In 1986, an attempt of extending the Widrow-Hoff rule to multiple layers was made. It was at that moment the, as we call it today, backprogagation method was born. However, back then it didn't exist powerful enough computers, which was needed for the computationally heavy backpropagation. There is a reason why it took all the way from then till 2011 to start using that theory, because by then the computers were powerful enough (even though it still takes a long time to train networks with multiple layers today).

Perceptrons

Artificial neural networks consist of multiple layers of neurons (also called units). There is an input layer, a hidden layer or several hidden layers (it doesn't necessarily need to exist a hidden layer), and an output layer. Normally a neural network is defined as a MLP if there are two or more hidden layers.

Perceptrons are one sort of neurons and they were developed in the 1950's and 1960's by a scientist called Frank Rosenblatt (inspired of the work McCulloch and Pitts, mentioned earlier). Very briefly, the perceptron takes several binary inputs and produces a single binary output (See *Figure 2*).

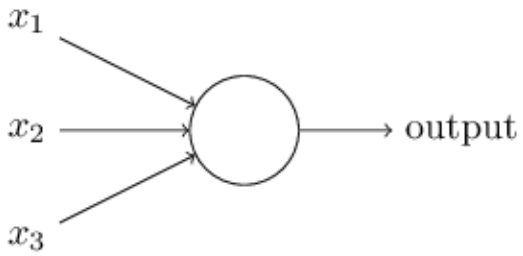

Figure 2 The perceptron.

Each binary input is multiplied with a *weight*, and if the sum of all these input-multiplications is greater than some *threshold value* the output will be 1, else it will be 0. A common approach is to move the threshold value to the same side as the weight and input, and call it the *bias*.

$$output = \begin{cases} 1 \ , if \ \sum_i w_i x_i + b > 0 \\ 0 \ , if \ \sum_i w_i x_i + b \leq 0 \end{cases} \quad (1)$$

ANNs of Today

Today it is more common with neurons that can take inputs others than binary values, and also give an output that is not binary.

Each neuron has several inputs, each multiplied by a weight. The summation of all these multiplications are added with the threshold value, also called the *bias* as mentioned above. The final value of this is used as an input to an *activation function* (the *sigmoid function* is often used as an activation function), which gives the output which is sent to the next layer of neurons. This output differs from the

perceptrons since it is not binary. The great difference is that here a small change in the weights and bias will only cause a small change in the output, which is beneficial for the learning process.

When training the system, it is common to dividing the data (supervised learning is used, which means that each input data has a label of its ground truth) into training data, validation data and test data (where the training data is of greater size than the validation data and test data combined). The training data is used as input to the system, where the biases are initialized (can be set to zero) and weights are randomly initialized (the initialization part can be done in several ways, but random is the most common one). The system is then evaluated by a cost-function at the end of the network. This is the function that tells us if the weights and biases are chosen good, or if they have to change. The goal is to find the global minimum of the cost function (though it is easy to get stuck in a local minimum). To minimize the cost function, the earlier mentioned backpropagation-method is used combined with gradient descent. As the name of the method reveals, the error is propagated backwards in the network, and simultaneously update the biases and weights with help of the backpropagated cost-function. Within this method, and the whole network actually, it exists several important parameters that control other parameters. These *hyper-parameters* have to be decided. It is here the validation data is used. After each epoch (one run over the network), when new weights and biases have been calculated, the network is tested with the validation data. To iteratively do this after each epoch, one can plot the cost-function of the training data and the validation data. From this plot it is possible to extract information that says if a certain hyper-parameter should be changed. Though this is an iterative process, and training neural networks consists of lots of trial-and-error to find the right hyper-parameters, if it is even possible.

After this whole procedure the test data is used to evaluate the network as a final product.

Data Preprocessing

The simplest form of preprocessing might be a linear transformation of the input data. A more complex form is dimensionality reduction, which can improve the result even though information is reduced.

To gain some prior knowledge and use it in the preprocessing stage can improve a neural networks' result dramatically.

Regarding the dimensionality reduction, it can be as simple as discarding a subset of the original input data, or it can consider approaches involving forming linear or non-linear combinations of the original variables to generate inputs for the neural network. These combinations of input are called *features*, and the process of generating the features is called *feature extraction*.

The motivation of dimensional reduction is that it can reduce the impact of the worst effects of the consequences of high dimensionality. If a network has fewer

inputs it has fewer adaptive parameters (e.g. weights) to be determined, which often means that the network gets better generalization properties. Also, a neural network with fewer weights may be faster to train.

Just consider the example of having an image of size 250 x 250 as input. This means that there are 62 500 input units and therefore 62 501 weights (including the bias) for every hidden unit to learn. A huge computational resource would be required to find the minimum of the cost-function. To tackle this problem, preprocessing the data can be a solution. A technique of dimensionality reduction can be considered, where *pixel averaging* is used on a block of pixels of the input image. The averaged pixels are examples of features. This procedure reduces the information, and can therefore lead to poor results. However, this problem doesn't exist for convolutional neural networks, which will be described further on.

Regarding the simpler preprocessing task, it is important to normalize and rescale (linear transformation) the data before it is used as input to the network. Each input variable should have a zero mean and a unit standard deviation over the transformed training set. Because of this, the weights can be initialized randomly.

For ANNs there are way more preprocessing-tasks to consider, compared to CNNs. Another example (others than mentioned above) is *missing value*, which means that an input variable is missing its input data. There exist different techniques to handle this issue (some better than the other), e.g. to express these variables in terms of regression over the other variables using the available data, and then use the regression-function to fill in the missing values.

However, these issues are not discussed regarding CNNs (in the same way as for ANNs). For CNNs there are two main preprocessing tasks:

- Normalization
- Augmentation

The main reason why this is the case is that the CNN itself perform a feature extraction, which will be described further on.

Normalization

The pixel values often lie in the range [0,255], and feeding these values into the network can cause the neuron to saturate (it basically stops learning). Therefore the learning will be really slow. Some preprocessing techniques are:

- *Mean image subtraction*
- *Per-channel normalization* (the method mentioned above, with zero mean and unit standard deviation)
- *Per-channel mean subtraction*
- *Whitening* (turn the distribution into a normal distribution)
- *Dimensionality reduction* (e.g. PCA, but this is not that common in deep learning)

The rule that is important to keep in mind is, always use the simplest method (the easier, the more effective).

Augmentation

To prevent the network from overfitting, a great way is to create artificial data. This is often done by simply mirror, rotate, etc. the input image.

Overfitting is when the network learns the training-data too well, and doesn't get generalized.

Bayesian Probability

Bayesian probability is an interpretation of the concept probability, where the probability represents belief.

The Bayesian view is that a *hypothesis* is assigned a probability, and this probability is evaluated by using a *prior probability*, which is then updated to a *posterior probability* with the use of new relevant data, or also called *evidence*. This procedure is called *Bayesian learning*.

Bayes' rule:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \qquad (2)$$

Where A (the proposition) and B (the evidence) are events, P(B)≠0, P(A) is the prior probability and p(A|B) is the posterior probability.

So instead of that a hypothesis is either *true* or *false* (1 or 0), it takes on a value in the interval [0,1].

Bayesian Artificial Neural Networks

In the early 90's a Bayesian approach to neural networks was explored. This approach is to make it easier to optimize the network in a faster way, rather than the slow process of trial-and-error. Some other beneficial properties are:

- The training method of error-minimization arises from an approximation to the Bayesian approach
- With the use of a Bayesian framework, the regularization term is given as a natural interpretation
- The regularization-coefficients can be selected using only the training-data. I.e. no validation-data is needed, and a lot of computations can therefore be prevented.
- Bayesian approach allows different models to be compared, using only the training-data (what distinguish models are different number of units, hidden layers, cost-function, etc.).

Bayesian ANN is divided into three levels:

- Weight selection
- Hyper-parameters selection
- Model selection

There is a hierarchical nature of the Bayesian framework (to be described), which will become obvious. The evidence at level two and level three is given by the denominator of Bayes' theorem at the previous level.

Weight Selection

The normal way of retrieving the "perfect" weights in a neural network is to train the network by the use of the backpropagation-method while minimizing the cost-function which finds a single set of weights, as mentioned earlier. The Bayesian approach, on the other hand, considers a probability distribution function over weight space that represent the relative degrees of belief in different values of the weights.

First this function is set to some prior distribution, p(**w**). Here **w** = (w_1, … , w_W) is a vector containing the adaptive weights and biases. When data D = (t^1, … , t^N) is observed, it can be converted to a posterior distribution, by using Bayes' theorem:

$$p(\boldsymbol{w}|D) = \frac{p(D|\boldsymbol{w})p(\boldsymbol{w})}{p(D)} \qquad (3)$$

where the denominator (which is a normalization factor) can be written as:

$$p(D) = \int p(D|\boldsymbol{w})p(\boldsymbol{w})d\boldsymbol{w} \qquad (4)$$

The prior distribution p(**w**) is typically a broad distribution, since there is generally little idea at the prior state of what the weights should look like. When the data are observed, the posterior distribution is much more compact than the prior distribution, since the evidence (data) gives an idea of how the network will behave.

The prior distribution is usually defined as:

$$p(\boldsymbol{w}) = \frac{1}{Z_W(\alpha)}\exp(-\alpha E_W) \qquad (5)$$

$$E_W = \frac{1}{2}||\boldsymbol{w}||^2 \qquad (6)$$

where E_W corresponds to the use of a simple weight-decay regularizer, and α is a hyper-parameter (since it controls the distribution of other parameters, weights and biases) and the denominator

$$Z_W(\alpha) = \int \exp\left(-\frac{\alpha}{2}\left|||\boldsymbol{w}||\right|^2\right) d\boldsymbol{w} = \left(\frac{2\pi}{\alpha}\right)^{W/2} \qquad (7)$$

which is a normalization factor.

In general, the likelihood function in Bayes' theorem, that defines the posterior distribution, can be written:

$$p(D|\boldsymbol{w}) = \frac{1}{Z_D(\beta)}\exp(-\beta E_D) \qquad (8)$$

where E_D is an error function, β is another hyper-parameter, and the denominator (a normalization factor):

$$Z_D(\beta) = \int \exp(-\beta E_D)\, dD = \left(\frac{2\pi}{\beta}\right)^{N/2} \qquad (9)$$

This will give the posterior distribution:

$$p(\boldsymbol{w}|D) = \frac{1}{Z_S}\exp(-\beta E_D - \alpha E_W) = \frac{1}{Z_S}\exp\big(-S(\boldsymbol{w})\big) \qquad (10)$$

and the denominator:

$$Z_S(\alpha,\beta) = \int \exp(-S(\boldsymbol{w}))\, d\boldsymbol{w} \qquad (11)$$

The maximum of the posterior distribution, the weight vector $\mathbf{w}_{MP}$, can now be found by minimizing the negative logarithm of the expression of the posterior distribution, with respect to the weights. This can be done if the hyper-parameters are considered as fixed (the estimation of them will be described in next section).

Though, in practice $Z_S(\alpha,\beta)$ cannot be evaluated analytically. In practice the interesting parts to evaluate are the probability distribution of the network predictions and also the evidences for the hyper-parameters and for the model. This means that integration over weight space is required, and therefore simplifying approximations needs to be introduces. Bishop recommends the Markov chain Monte Carlo integration in [7].

Hyper-Parameters Selection

Hyper-parameters are in general the trickiest part of a neural network to decide. This is because of the lack of knowledge of how neural networks work. With "ordinary" neural networks, the hyper-parameters are decided with trial-and-error (and validation-data).

The Bayesian treatment of these unknown hyper-parameters is to integrate them out of any predictions, e.g. in the posterior distribution:

$$p(\boldsymbol{w}|D) = \iint p(\boldsymbol{w},\alpha,\beta|D)d\alpha d\beta = \iint p(\boldsymbol{w}|\alpha,\beta,D)p(\alpha,\beta|D)d\alpha d\beta \qquad (12)$$

where

$$p(\alpha,\beta|D) = \frac{p(D|\alpha,\beta)p(\alpha,\beta)}{p(D)} \qquad (13)$$

which can be assumed to have its highest peaks around the most probable hyper-parameters α_{MP} and β_{MP}. Then the posterior distribution $p(\mathbf{w}|D)$ can be rewritten as:

$$p(\boldsymbol{w}|D) \approx p(\boldsymbol{w}|\alpha_{MP}, \beta_{MP}, D) \qquad (14)$$

I.e. the hyper-parameters should be chosen so that this posterior distribution is being maximized (and then use these values of the hyper-parameters for further calculations).

The posterior distribution of the hyper-parameters contains a term on the right hand-side, p(α,β), called the *hyperprior*. Since the initial values of the hyper-parameters often are without prior knowledge, the best way of initializing this hyperprior is to do it *non-informative*, i.e. choose the hyperprior so it in some sense gives equal weight to all possible values. The hyper-parameters are scale-parameters (since they determine the scale of $\mathbf{||w||^2}$), and non-informative priors for scale-parameters are generally chosen so that they are uniform on a logarithmic scale.

To maximize p(α,β|D) is equal as to maximize p(D|α,β), which is the *evidence* for α and β. If the dependency the hyper-parameters α and β is made explicit, the evidence can be written as:

$$p(D|\alpha,\beta) = \int p(D|\boldsymbol{w},\alpha,\beta)p(\boldsymbol{w}|\alpha,\beta)d\boldsymbol{w} = \int p(D|\boldsymbol{w},\beta)p(\boldsymbol{w}|\alpha)d\boldsymbol{w} \qquad (15)$$

where the fact that the prior is independent of β and the likelihood function is independent of α, has been used. By using the equations from the last section, the evidence can be written as:

$$p(D|\alpha,\beta) = \frac{1}{Z_D(\beta)}\frac{1}{Z_W(\alpha)}\int \exp(-S(\boldsymbol{w}))d\boldsymbol{w} = \frac{Z_S(\alpha,\beta)}{Z_D(\beta)Z_W(\alpha)} \qquad (16)$$

The maximum of the logarithm of this equation is what is needed to be found. To achieve this, α and β needs to be considered separately, i.e. first maximizing the equation w.r.t. α and then to β. These values of the hyper-parameters are then used to find $\mathbf{w}_{MP}$. This is done by using a standard iterative training algorithm, where the hyper-parameters are re-estimated as:

$$\alpha^{new} = W/2E_W \qquad (17)$$

$$\beta^{new} = N/2E_D \qquad (18)$$

which are an approximation of the outcome of maximizing the logarithm of the evidence.

This is one approach, but there are alternatives that also are interesting to investigate, e.g. to analytically perform integrations over α and β. This can be read about in [7].

Model Selection

When selecting the model (number of layers, units, etc.) to use for the neural network, it is important to have in mind that the least complex model (that gets the job done) is the best model (see *Figure 3*). The great thing about the Bayesian formalism is that it automatically penalizes complex models.

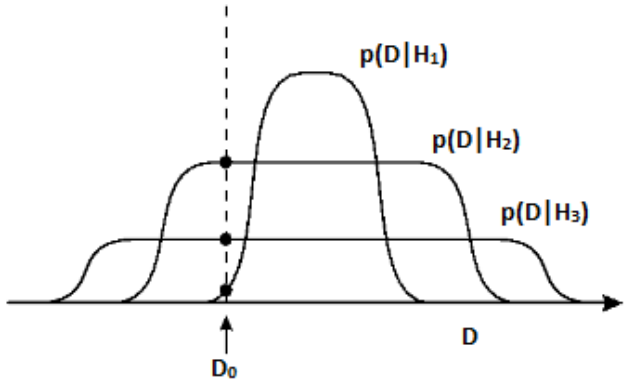

Figure 3 Model selection.

If H_i is a set of models, the posterior probabilities of those various models (after observing the data D) can be written as:

$$p(H_i|D) = \frac{p(D|H_i)p(H_i)}{p(D)} \qquad (19)$$

where $p(H_i)$ is the prior probability assigned to model H_i and $p(D|H_i)$ is the evidence. Actually, this evidence is exactly the denominator in equation (13), in which the conditional dependence of model H_i has been made explicit.

The evidence can be written as:

$$p(D|H_i) = \int p(D|\boldsymbol{w}, H_i)p(\boldsymbol{w}|H_i)d\boldsymbol{w} \qquad (20)$$

If (as will be assumed) the posterior distribution is sharply peaked around a the most probable value w_{MP}, then the integral can be approximated as the maximum times the width $\Delta w_{posterior}$ of the peak:

$$p(D|H_i) \approx p(D|w_{MP}, H_i)p(w_{MP}|H_i)\Delta w_{posterior} \qquad (21)$$

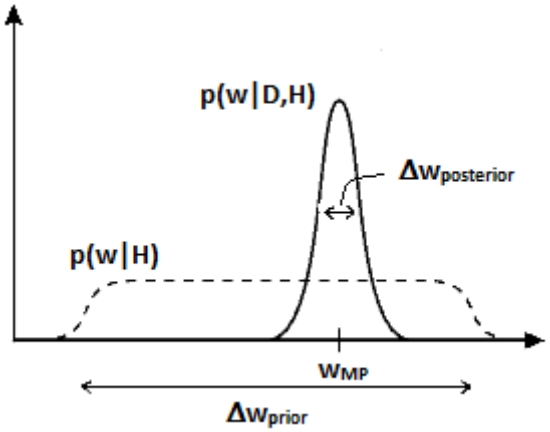

Figure 4 The sizes of the distributions.

If the prior is chosen to be uniform over a large interval Δw_{prior} (see *Figure 4*), the evidence can be re-written as:

$$p(D|H_i) \approx p(D|w_{MP}, H_i)\left(\frac{\Delta w_{posterior}}{\Delta w_{prior}}\right) \qquad (22)$$

The first term on the right hand-side is the likelihood evaluated of the most probable weight values, and the second term is called an *Occam factor* (<1), which penalizes the network. A small Occam factor indicates on a complex model, and a higher value indicates that the network is less complex. The model with the largest evidence will be determined by the balance between needing a large likelihood (which will be the case when the model should fit the data well) and needing a large Occam factor (so that the model isn't too complex).

Deep Convolutional Neural Networks

In an ANN, the layers are said to be fully connected. This means that each unit, or neuron, in a layer is connected to all the units in adjacent layers.

When patterns are to be recognized in images, CNN is recommended to use. This is because of several reasons (e.g. that ANN doesn't take the spatial structure into account), but one important reason is the computational complexity. E.g. if the input is an image of size 50 x 50 pixels, this means that there is a need of 2500 units in the input layer. And then to have several hidden layers, with several hidden units, with thousands of training-images, one can easily imagine the computational effort, especially if the trial-and-error of finding the correct hyper-parameters is also considered.

So, CNNs are great for classification of images and are fast for training deep, multi-layer networks. Though, with CNN there are more hyper-parameters to decide/optimize (as far as possible, since optimizing all the hyper-parameters is not a converging process), which will be mentioned further on.

Convolutional neural networks use three basic ideas:

- Local receptive fields
- Shared weights and biases
- Pooling

Local Receptive Fields

In fully connected layers, the input can be seen as a vertical line of neurons. In CNNs, the input can be seen as a square of the same size as the input image, where every pixel is replaced by a neuron (see *Figure 5*).

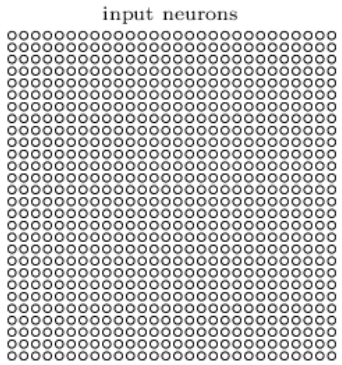

Figure 5 The input neurons.

As in the ordinary neural network, the input neurons/pixels are connected to the hidden layer of neurons. But instead of the case where every neuron in the input is connected to every neuron in the hidden layer, the connections are made in small localized regions of the input image. I.e. each neuron in the first hidden layer is connected to a region of input neurons, as in *Figure 6*.

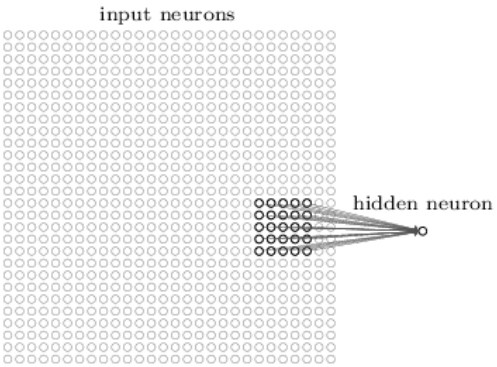

Figure 6 Local perceptive field.

The region in the input image, which are connected to a hidden neuron, is called the *local receptive field* for that hidden neuron (size m x m). It can be thought of as a little window on the input pixels where each connection learns a weight and the hidden neuron learns an overall bias.

The local receptive field is slid over the whole image, and for each local receptive field there is a different hidden neuron in the first hidden layer (this is the convolution part). When the field is slid across the image, it can move one or

several pixels at a time. How many pixels the window is moving is called the *stride length*.

Shared Weights and Biases

For each hidden neuron, there are m x m weights and a bias. What is special about this is that all the hidden neurons have the same m x m weights and bias. This means that all the hidden neuron in the first hidden layer detect the same feature (a feature could be a vertical line, corner, etc.) in the input image, just at different locations. Therefore, CNNs are well adapted to the translational invariance of images. Because of this, the *map* from the input layer to the hidden layer is called a *feature map*. The weights and bias defining the feature maps is called the *shared weights* and the *shared bias*.

In a CNN, each hidden layer (also called *kernel* or *filter*) has several feature maps, since each feature map only recognize one feature in an image.

The advantage of using shared weights and shared bias is that it reduces the number of parameters in the CNN.

Pooling

CNNs contain layers called *pooling layers*, which are used after the convolutional layer. The pooling layer simplify the information in the output from the convolutional layer (make each feature map condensed).

A common pooling-method is the *max-pooling*. This method takes the maximum activation output in a k x k (e.g. k=2) region, which become the pooling unit. The pooling is done to each feature map separately.

Max-pooling can be thought of as a way of asking the network if any feature is found in the image. If so, the exact positional information is thrown away. The only important part for the network is the position relative to other features.

The Network

The CNN is these three areas put together. Usually the network ends with a fully connected layer (or several fully connected layers), which are exactly the same as described for ANNs.

The backpropagation-method is also used for CNNs for training, as it was described for ANNs.

Bayesian Deep Convolutional Neural Networks

When applying CNNs on small datasets, overfitting can be seen as almost a fact. It is a drawback for the CNNs that they overfitting the data so easily when the dataset is too small. This has until now been handled with techniques as *drop-out*, adding a *regularization term* to the cost-function, and by *augmenting* the dataset (as a preprocessing step by creating artificial data).

A Bayesian approach to deep convolutional neural networks will prevent this drawback, by placing a probability distribution aver the CNN's kernels (compared to the Bayesian ANNs weights). This correspond to perform a drop-out after every convolutional layer in practice. Though, not the *standard drop-out*, but the *Monte Carlo drop-out* (or MC drop-out, which will be described further on in this section).

To not increase the number of parameters (as in the Bayesian ANNs) in the network, *Bernouilli approximating variational distributions* are used, since the Bernouilli variables doesn't require any additional parameters for the approximate posteriors.

To be able to understand how the MC drop-out works, some background-thoughts have to be considered. So if $\{x_1, \dots, x_N\}$ is the training inputs and $\{y_1, \dots, y_N\}$ is the corresponding output. In probabilistic modelling it would be interesting to estimate the function that performs the function y=f(x) that is likely to have generated the output y. First a prior distribution over all the possible functions is needed, p(f). Then to capture the process where observations are generated given a specific function, the likelihood p(**Y**|f, **X**) is defined. As usual in a Bayesian approach, the posterior distribution over the space of functions given the dataset p(f|**X**, **Y**) is wanted. With this distribution an output for a new input point x* can be predicted by integrating over all possible functions f:

$$p(y^*|x^*, \boldsymbol{X}, \boldsymbol{Y}) = \int p(y^*|f^*)p(f^*|x^*, \boldsymbol{X}, \boldsymbol{Y})df^* \qquad (23)$$

The integral in the equation is not so easy to handle, therefore it needs to be approximated. To do this approximation, the model is conditioned on a finite set of random variables w. Then the integral would look like:

$$p(y^*|x^*, \boldsymbol{X}, \boldsymbol{Y}) = \int p(y^*|f^*)p(f^*|x^*, w)p(w|\boldsymbol{X}, \boldsymbol{Y})df^*dw \qquad (24)$$

Now another distribution, p(w|**X**,**Y**), cannot be analytically analyzed. Therefore, an approximate *variational* distribution q(w), which is easy to analyze, is defined. The goal is to have q(w) as similar to p(w|**X**,**Y**) as possible. As a measure of the similarity between two distibutions, *Kullback-Leibler* (KL) divergence is minimized to achieve this goal. The approximate predictive distribution looks like the following:

$$q(y^*|x^*) = \int p(y^*|f^*)p(f^*|x^*, w)q(w)df^*dw \qquad (25)$$

Minimizing the KL divergence is equivalent to maximizing the *log evidence lower bound* with respect to the variational parameters defining q(w):

$$\mathcal{L}_{VI} \coloneqq \int q(w)p(\boldsymbol{F}|\boldsymbol{X}, w)\log p(\boldsymbol{Y}|\boldsymbol{F})d\boldsymbol{F}dw - KL(q(w)||p(w)) \qquad (26)$$

where KL(q(w)||p(w)) is the KL divergence.

This is known as *variational inference*, a standard technique in Bayesian modeling. This is related to drop-out in neural networks by defining the approximating variational distribution q($\mathbf{W}_i$) (here $\mathbf{W}_i$ is an array of weights) for every layer i as:

$$\boldsymbol{W_i} = M_i \cdot diag(\left[z_{i,j}\right]_{j=1}^{K_i}) \qquad (27)$$

$$z_{i,j} \sim Bernouilli(p_i) for\ i = 1, \dots, L, j = 1, \dots, K_{i-1} \qquad (28)$$

where $z_{i,j}$ random variables with a Bernouilli distribution, with a probability p_i, and M_i are variational parameters to be optimized. To sample from q($\mathbf{W}_i$) is identical to drop-out in layer i in a network which weights are $(M_i)_{i=1}^{L}$. The variable $z_{i,j}$=0 (which is binary) corresponds to a drop-out in unit j in layer i-1 as an input to the ith layer.

The expression in equation (26) is intractable and needs to be approximated. It is done by approximating the equation with *Monte Carlo integration* over w, which leads to an unbiased estimator of $\mathcal{L}_{VI}$:

$$\hat{\mathcal{L}}_{VI} \coloneqq \sum_{i=1}^{N} E\left(y_i, \hat{f}(x_i, \hat{w}_i)\right) - KL(q(w)||p(w)) \qquad (29)$$

$$\hat{w}_i \sim q(w) \qquad (30)$$

where E(·,·) corresponds to the *softmax lost*, a certain cost-function. Also in equation (25) with the posterior p(w|**X**,**Y**) replaced by the approximate posterior q(w), the integral can be approximated with a Monte Carlo integration:

$$p(y^*|x^*, \boldsymbol{X}, \boldsymbol{Y}) = \int p(y^*|x^*, w)q(w)dw \approx \frac{1}{T}\sum_{t=1}^{T} p(y^*|x^*, \hat{w}_t) \qquad (31)$$

$$\hat{w}_i \sim q(w) \qquad (32)$$

This is known as MC drop-out.

This theoretical development results in the Bernouilli approximate variational inference and is implemented in CNNs by performing a drop-out after each convolutional layer and before the pooling layer. So a prior distribution is placed over each kernel and each kernel-feature map is approximately integrated with Bernouilli variational distributions. Bernouilli random variables $z_{i,j,n}$ (n because each kernel has a deep, a number of feature maps) are sampled and each feature map n is multiplied by:

$$\boldsymbol{W_i} = M_i \cdot diag(\left[z_{i,j,n}\right]_{j=1}^{K_i}) \qquad (33)$$

This distribution randomly sets different feature maps to zero in kernels.

Conclusion

A Bayesian approach is in general not as popular as the “ordinary” approach. This is mainly because of the lack of implementations of Bayesian neural networks

compared to the “ordinary” neural networks, which can be derived from back in the 90’s, when the “ordinary” neural networks were more popular and more focused on.

Also an important reason is that the scientists that mainly worked with neural networks weren’t statisticians. It is not until recently, as the statistical area has influenced the world of machine learning.

The Bayesian neural networks seem very promising theoretically, but in practice one have to approximate the posterior in the network, which can lead to heavily computations. This is because Gaussian approximating distribution increase the number of parameters considerably, when not increasing the capacity that much.

The results of the predictive performance of Bayesian ANNs haven’t been reported better than any neural network using drop-out, when the Bayesian ANNs also increase the complexity. This is why the same approach is not suitable for CNNs, since an increase of parameters are too costly. That is why Bernouilli approximating variational distributions are used for the Bayesian CNNs, since this approach doesn’t increase the number of parameters.

While ANNs can handle small dataset very well, CNNs are overfitting easy when it comes to too small datasets. This is of course a drawback and to prevent the overfitting from happening, one can use data preprocessing as augmentation of the data by generating artificial data. However, the Bayesian approach to deep convolutional neural networks is supposed to handle this drawback when using small datasets.

References

Artificial Neural Networks:

[3] M. Nielsen, ”Neural Networks and Deep Learning,” 2016. [Online]. Available: http://neuralnetworksanddeeplearning.com/index.html. [Använd December 2016].

[7] C. Bishop, Neural Networks for Pattern Recognition, 1995.

[1] R. Rojas, Neural Networks A Systemaic Introduction, Berlin: Springer, 1996.

[2] ”cs.stanford.edu,” 2000. [Online]. Available: http://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html. [Använd December 2016].

Data Preprocessing:

[7] C. Bishop, Neural Networks for Pattern Recognition, 1995.

Bayesian Probability:

[5] ”Wikipedia,” 26 November 2016. [Online]. Available: https://en.wikipedia.org/wiki/Bayesian_probability. [Använd December 2016].

Bayesian Artificial Neural Networks:

[7] C. Bishop, Neural Networks for Pattern Recognition, 1995.

Deep Convolutional Neural Networks:

[3] M. Nielsen, ”Neural Networks and Deep Learning,” 2016. [Online]. Available: http://neuralnetworksanddeeplearning.com/index.html. [Använd December 2016].

[4] I. G. a. Y. B. a. A. Courville, ”Deep Learning Book,” MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org/. [Använd December 2016].

Bayesian Deep Convolutional Neural Networks:

[6] Y. G. a. Z. Ghahramani, ”Bayesian Convolutional Neural Networks with Bernoulli Approximate Variational Inference,” *ICLR 2016,* 2016.