MAP670L - Generalisation properties of algorithms in ML

Role of Tweaks in Deep Learning

Elaborated by :

Mehdi Zadem
Khmayes Abouda
Thi Hai Yen Vu

Academic Year 2019/2020

Contents

1 Introduction

Generalisation of machine learning algorithms is a real concern seeing that a model that fails to generalise will not be of much use when applied to new seen data. Deep Learning, however a very powerful technique in some situations, can be very costly to generalise if we take simple approaches. Research in the recent years has managed to come up with ways, or rather "tweaks", that help neural networks reduce overfitting without a heavy impact on performance.

2 Aim and Framework

This project aims to highlight the effect of such tweaks and to provide a brief comparison of these techniques. We'll perform a classic image classification task with a basic CNN model (convolutional neural network).

The CIFAR-10 dataset will be used for all our experimental results. It contains $60,000$ images of size 32x32 divided in 10 categories. During all of our experiments we split the data to train/validation/test of sizes $40,000/10,000/10,000$.

The CNN network used in this work is a simple CNN architecture with 2 blocks of 2DConv followed by a Maxpooling, then another 2 blocks of 2DConv followed by another Maxpooling, it is then Flattened and is passed through a Dense layer before a final Feed-Forward layer to classify the image. This results in about $1,250,000$ parameters in total.

All our experiments are performed on Google Colab using a free 12GB NVIDIA Tesla K80 GPU. The running time are roughly the same for nearly all experiments, taking up to $100s$ for each epoch (with the batch size of 32 normally).

3 Dropout

3.1 Principal and Motivation

Dropout is a simple approach designed to help neural networks achieve better generalisation proprieties by introducing a certain noise to the weights. The idea presented at [7] is simple but effective. Let us consider a neural network where each neuron is updating its weights according to the train data. Given the density in this kind of structure, it is very likely that the model will be too bound to the data fed during training, and will not generalise very well. So what happens if we just reduce the number of the neurons. Intuitively speaking, the model should be a little less attached to the training data and have better performance during the validation phase.
To explain further, the Dropout technique elaborated in [7] will be controlled by a parameter p which is the probability for a neuron to be kept in the network. And thus, the Dropout eliminates randomly a number of neurons along with their connections during training. If we have n neurons in total, the dropout will generate 2^n different subnetworks after the random elimination of the nodes. All in all, at each training case a different thinned network will be sampled and used. During the test phase, it would be impractical to average the the output of 2^n trained networks and therefore an approximation is in need. To solve this issue, it is possible to use the full network, with all its neurons present, but scale each weight by the probability p of its presence in the network. The described process ensures that a certain random noise is introduced to the network and it improves

the generalisation capabilities of the model as will be shown by the experimental findings that follow.

3.2 Mathematical Model

The mathematical formulation is as follows. Adapting the notation from [7], we consider a network composed of L hidden layers of the network, with $l \in \{1 \dots L\}$ the index of a layer of the neural network. In addition, we consider

- $z^{(l)}$: vector of inputs for layer l
- $\mathbf{y}^{(l)}$: vector of outputs of layer l
- $\mathbf{w}^{(l)}$: vector of weights of layer l
- $b^{(l)}$: vector of biases of layer l

For a simple neural network, we have for each neuron i the following forward-pass equations

$$\begin{aligned} z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)} \\ y_i^{(l+1)} &= f\left(z_i^{(l+1)}\right) \end{aligned}$$

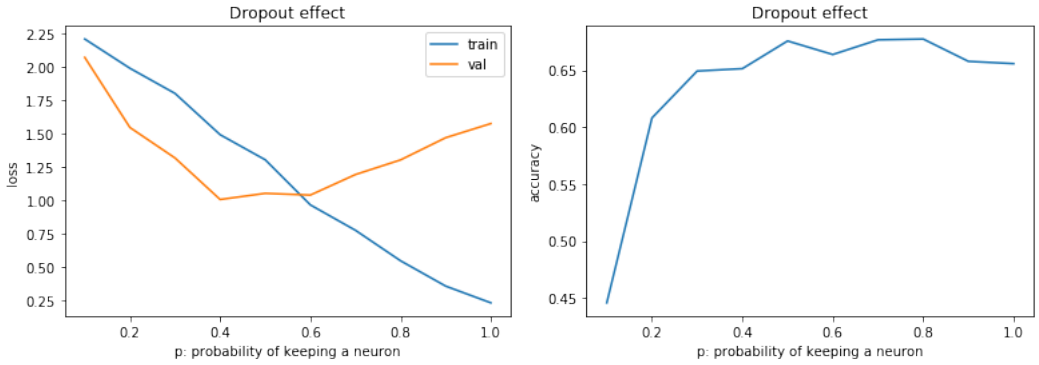
where f is an activation function (ReLu, sigmoïd..)
But when considering the dropout effect, this architecture is slightly modified to account for the elimination of some neurons. A new variable is introduced $r^{(l)}$ which represent a vector of random variables each following a Bernoulli distribution with parameter p. In other words, $r_j^{(l)}$ is equal to 1 with probability p, otherwise it is equal to 0. The new forward-pass is:

$$\begin{aligned} r_j^{(l)} &\sim \text{ Bernoulli } (p) \\ \widetilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)} \\ z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \widetilde{\mathbf{y}}^l + b_i^{(l+1)} \\ y_i^{(l+1)} &= f\left(z_i^{(l+1)}\right) \end{aligned}$$

This architecture is the one adapted for training the network. The weights updates are performed by means of backpropagation.
As previously elaborated, during the test phase, the first architecture is used with the weights being scaled down by a factor p to take into consideration the probability of retaining each unit.

3.3 Experimental Results

To illustrate the usefulness of Dropout, the CIFAR dataset is used with the provided CNN architecture. A Dropout is applied to the fully connected output layer where the hyper parameter p is to be modified in $[0, 1]$. The training is performed over 10 epochs. Displayed in 1(a), are the loss values for the training and validation data corresponding to each value of the parameter p.



(a) Training and Test losses depending on the parameter p

(b) Effect of dropout on test accuracy depending on the parameter p

Figure 1: Effect of Dropout on model performance

The following observations are made:

- The more likely a neuron is present, the lower the train loss gets.
- The validation loss present a minima around $p = 0.4$, and then picks up to reach higher values.

The Dropout effect shows that, while training loss can be sacrificed when removing neurons with probability $p = 0.4$, we manage to obtain significant decrease in validation loss which indicates better robustness and generalisation for the model. Consequently, the accuracy of the model climbs to a peak then drops again as shown in 1(b).

4 Large initial step size

4.1 Context

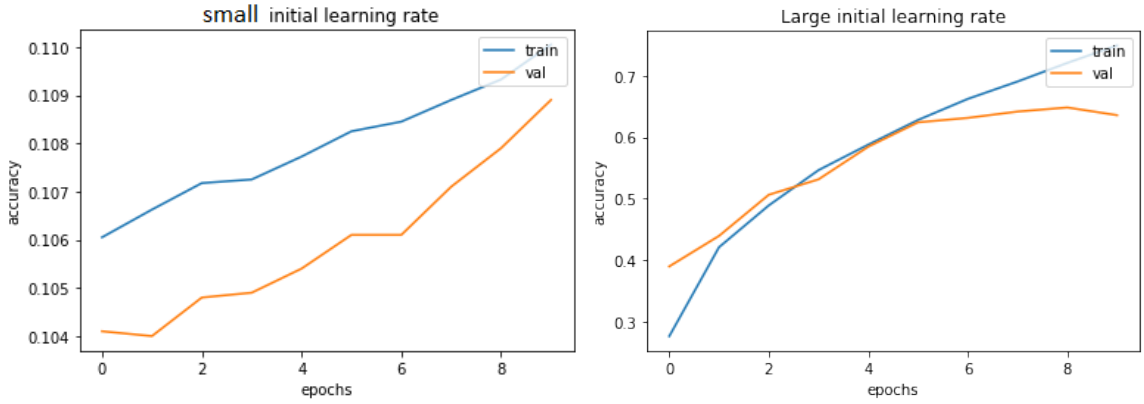
The general, yet unproven rule, when training a deep neural network is to use a large learning rate for the optimisation algorithm such as SGD. Large learning rate are almost always better for generalisation and allow to reduce overfitting. In fact, the most intuitive difference between the optimisation procedures of the large and small step sizes is the following; With larger sizes of the learning rate, the algorithm is only learning a general pattern of the data by making bigger steps during the gradient descent. On the other hand, smaller values of the learning rate tend to make the model stick closer to the provided data and learn more detailed, specific patterns. And while these values achieve faster convergence, it is very probable to land on a local minima for descent algorithms. Larger step values are more likely to converge to a global minima for the loss thus the better generalisation.
In [5], the authors elaborate a more adequate setting to test these informal theories. The idea is to feed the model a set of data that mainly contains two components $\mathcal{P}$ and $\mathcal{Q}$. $\mathcal{P}$ models a high noise and easy to fit patterns while $\mathcal{Q}$ models low noise with hard to fit pattern. Through the use of two algorithms with different learning rate values, the team tries to fit the data and see if their explanation of the behavior the better performance of large step sizes holds. The first algorithm(S) simply uses a small learning rate. The second algorithm (LS), uses a large initial step size combined with annealing. Annealing refers to reducing the learning rate a some point throughout the iterations.

The strategy proves to be a success as the obtained results confirm the expected behavior.

- **Algorithm (S)** manages to fit the pattern featured in the component $\mathcal{Q}$ with a low number of observations. But struggles with $\mathcal{P}$ as the learning rate is too dependant on the high noise
- **Algorithm (LS)** is able the successfully fit $\mathcal{P}$ in a initial phase using the large learning rate value. And when the annealing is introduced, the model switches to fitting $\mathcal{Q}$.

The authors then manage to establish two theorems concerning the behaviour of the learning rates (bounds, complexity..) in this very specific case of fitted data.

4.2 Experimental Results



(a) lr = 1e-7 (b) lr = 0.01

Figure 2: Effect of initial step size on model accuracy

The CNN model trained on the CIFAR-10 dataset is far more accurate for larger learning rates which indeed illustrates the hypothesis. The used model, and mostly the nature of the data itself, could not illustrate to a large extent the overfitting effect produced by small learning rates. But what it is clear from 2 is that a small learning rate of $1e-7$ is inadequate to correctly train the model whereas a large learning produces a far more efficient model.

5 Batch normalization

Batch Normalization [6] is a method used deep learning that has been shown to significantly speed up training process as well as having a regularization effect by adding noise to each layer in the network by normalized a portion of the dataset (mini-batches). It has also been shown that BatchNorm can help speeding up the learing phase by reducing *internal covariate shift*, i.e the deeper layers sensibility to input's distribution change due to parameter updates.

5.1 Intuition

In logistic regression framework, normalizing the input has been shown to speed up the learning process. In a deeper neural network, Batch Normalization is a way to generalise

the idea by normalizing the input of each hidden layer.

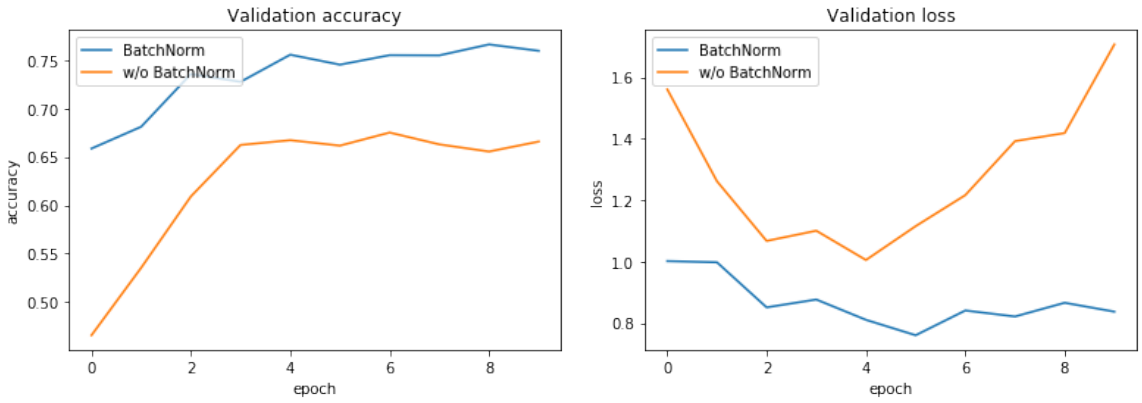
5.2 Batch-normalized network

Data set is divided into mini-batches. These mini-batches are fed to a network one in a time and for each hidden layer, we apply Batch Norm on its activation values.
The parameters of the model are the weights of the network and scaling and shifting parameters γ and β.
For a layer l, denote by x^l the activation, γ^l and β^l the parameters of the Batch Norm transformation.

1. Compute $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ the mean and the variance of the mini-batch $\mathcal{B}$.
2. Normalize
$$x^l_{norm} = \frac{x^l - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$$
ϵ is introduced for numerical stability.
3. Scale and shift
$$\tilde{x}^l = \gamma^l x^l_{norm} + \beta^l \equiv \mathrm{BN}_{\gamma^l,\beta^l}\left(x^l\right)$$

5.3 Regularization effect

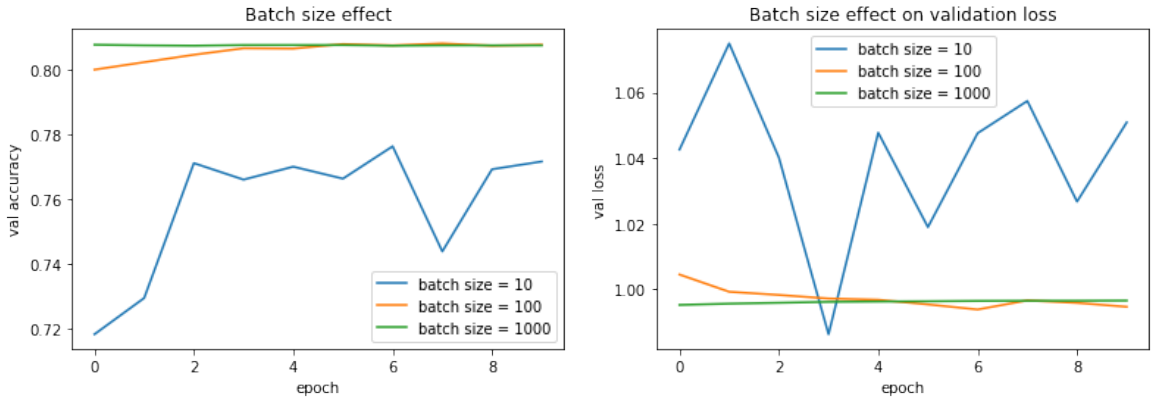
Each mini-batch is scaled by and then shifted using the mean and the variance computed on just that mini-batch. So because of the small size of the data contained in the mini-batch, we get noisy mean and variance. For each mini-batch, performing scaling and shifting on the input of each layer results in noisy values and therefore the model will generalise better and will be less sensible to changes of the input data.
The following figure illustrate the generalisation aspect of Batch Normalization.



(a) Test accuracy (b) Validation loss

Figure 3: Effect of Batch Normalization on test accuracy and validation loss. Training done on CIFAR-10 data set, with batch size of 32.

When we increase mini-batches size, we get closer to the real mean and variance of the data set. This can improve the performance and have a more apparent regularization effect.



(a) Validation accuracy

(b) Validation loss

Figure 4: Performing Batch Norm with different batch sizes

We notice that with a larger batch size, regularization effect is more evident.

5.4 Batch Normalization effect on learning process

The following figure shows that performing Batch Normalization speeds up the training phase.

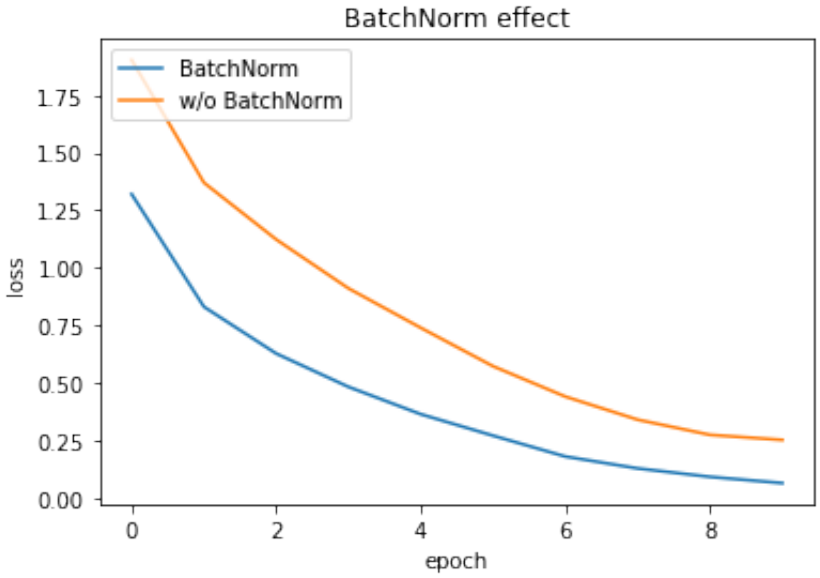


Figure 5: Training loss obtained with and without Batch Normalization

6 Weight Initialization

The starting values of the weights can have a significant effect on the training process [4]. Intuitively, initializing weights with a constant is not a good idea as the weights will always equal to a common value during training process. As a result, in general weights are chosen randomly.

In the following sections, we are going to introduce some initialization methods and then compare them on our CNN model with the CIFAR-10 dataset.

6.1 Initialization Methods

We consider the following initializations:

Constant Initialization

This is the most naive way to initialize weights in the network, where weights are set to be equal to a constant c before training. This method, however, is not efficient to train a deep neural network because different neurons in a Dense layer or different filters of a Convolutional layer will have the same outputs, which after back propagating through the network will still lead to the same values of weights. This means that the whole network will be equivalent to using only one neuron or one filter per layer.

Random Initialization

Randomly initialized weights can remedy the problem of constant initialization. Weights can be generated either from an uniform distribution $\mathcal{U}([-\sqrt{3}\sigma, \sqrt{3}\sigma])$ or a normal distribution $\mathcal{N}(0, \sigma^2)$, for some standard deviation σ. The value of σ, however, can affect greatly to the quality of the training process. For example, when using tanh or sigmoid activation functions, the gradients tend to be large for weights that are close to zero, while they tend to be very small for weights that are far away from zero. This can lead to the problems of vanishing and exploding gradients in the first layers, making it hard to train the network. We will see this in the next section.

Lecun Initialization

This initialization method was first introduced in [4]. It aims to find the appropriate σ based on the requirement that the variance of the outputs at every layer should be fixed to a constant. Formally, let's consider a linear neuron with N input neurons:

$$y = x_1 w_1 + x_2 w_2 + \cdots + x_N w_N + b,$$

where we suppose that the inputs x_i and the weights w_i are independant centered random variable, w_i have variance σ^2 and x_i and y have the same variance. We also suppose that the bias b is a constant. We have then:

$$\mathrm{Var}(y) = \sum_{i=1}^{N} \mathbb{E}\left(x_i^2\right) \mathbb{E}\left(w_i^2\right) = N \,\mathrm{Var}(x_1)\sigma^2$$

which infers that:

$$\sigma^2 = \frac{1}{N},$$

where N is the number of input neurons at the given layer.

Xavier/Glorot Initialization

Introduced in [1], this is a generalisation of Lecun Initialization. Instead of considering only equations of the forward pass, the backward equations (which comes from the backward

pass of back propagation) are also formulated which contributes to the formula of the variance:

$$\sigma^2 = \frac{2}{N_{in} + N_{out}},$$

where N_{in} and N_{out} are the number of input and output neurons of the current layer.

He Initialization

The two previous methods only concentrate on linear activations only, [2] deals with this by adressing the rectifier nonlinearities (ReLU functions). More precisely, if we consider:

$$y = f(x_1)w_1 + f(x_2)w_2 + \cdots + f(x_N)w_N + b,$$

where $f(x) = x_+$ is the ReLU function. Then we have:

$$\mathrm{Var}(y) = \sum_{i=1}^{N} \mathbb{E}\left(f(x_i)^2\right)\mathbb{E}\left(w_i^2\right) = \sum_{i=1}^{N} \mathbb{E}\left(x_{i+}^2\right)\mathbb{E}\left(w_i^2\right) = \sum_{i=1}^{N} \frac{1}{2}\mathbb{E}\left(x_i^2\right)\mathbb{E}\left(w_i^2\right) = \frac{1}{2}N\,\mathrm{Var}(x_1)\sigma^2,$$

where we used the fact that $\mathbb{E}\left(x^2\right) = \mathbb{E}\left((x_- + x_+)^2\right) = \mathbb{E}\left(x_-^2\right) + \mathbb{E}\left(x_+^2\right) = 2\mathbb{E}\left(x_+^2\right)$. We thus deduce the variance:

$$\sigma^2 = \frac{2}{N}.$$

6.2 Performance Comparison

Constant vs Random Initialization

Figure 6 compares the training loss over 10 epochs between constant initialization (initialized with 0 and 1) and randomized initialization methods (normal and uniform initializations), for different values of σ. As we expected, constant initialization does not work as the loss is not decreasing. However, not all values of σ give considerably good results, as we can see for σ too low or too high, the loss is not decreasing as well. We suggest a range for $\sigma \in [0.03, 0.07]$ in this case.

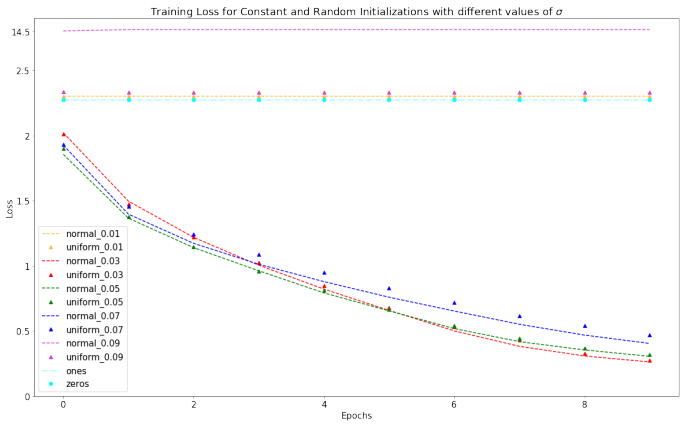
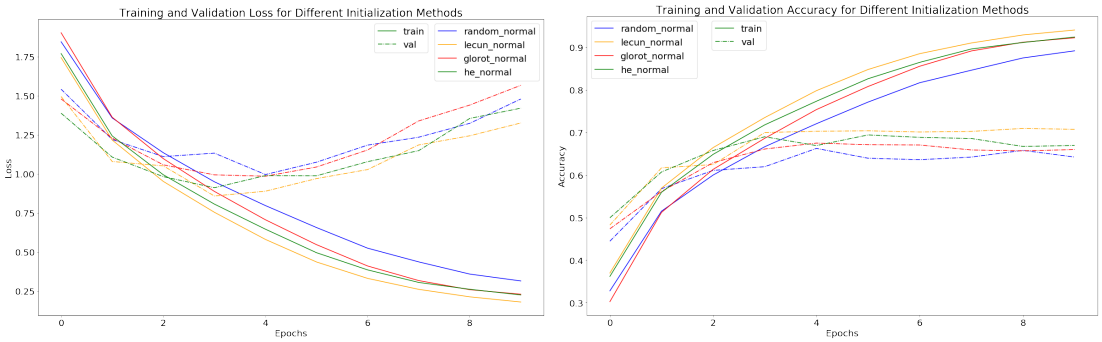


Figure 6: Constant vs Random initialization

Scaled Variance Initializations

The following figure compares the initialization methods given in the last section. We see that scaled variance initializations are better than just randomly initialize weights. Compared the methods, Lecun initialization works better than Glorot and He initializations, which give a little bit the same results.



(a) Train and Val loss

(b) Train and Val accuracy

Figure 7: Comparison of Scaled-Variance Initializations

7 Weight Decay

Apart from Dropout and Batch Normalization, another way to deal with Overfitting in deep neural network is Weight Decay [3]. This is done by adding a squared regularization term to the loss and has been shown to improve generalisation of the network.

7.1 Mathematical Formulation

In order to restrict the network's complexity, we simply add a squared $L2$-norm to the loss function of the network:

$$L(w) = L_0(w) + \frac{1}{2}\lambda \sum_{i=1}^{M} w_i^2,$$

where L_0 is the objective function of the network such as Cross-entropy Loss or Mean Squared Error, $w = (w_1, \ldots, w_M)$ denotes all the weights of the network (including biases), and λ is a parameter governing the penalization of large weights. By this way, we can prevent the weights from going too large during training.

7.2 Experimental results

We see how the regularization term influences the generalisation by training the network with varying values of λ.

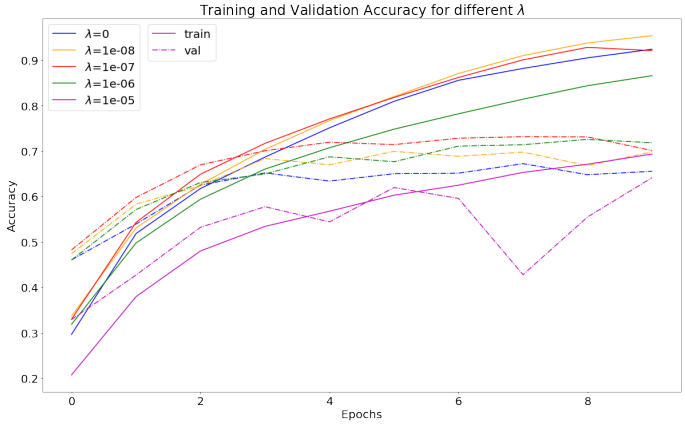


Figure 8: Impact of weight decay on train and validation accuracy

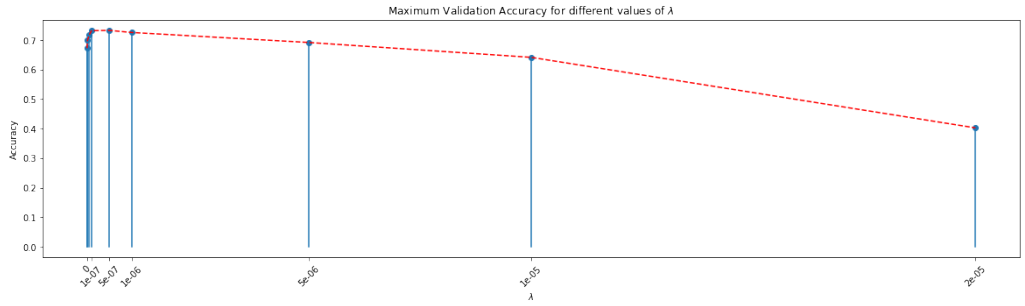


Figure 9: Effects of λ on validation result

We observe that weight decay do help generalize the network. However, a too large decay can lead to poor results as the weights tend to vanish. Therefore it also has to be tuned correctly for each problem.

8 Conclusion

We have seen several techniques that can be employed to help neural networks generalize. We studied the reguralization effect of Dropout and BatchNorm, two methodes that helps with regularization by leveraging some noise in the data. We also studied techniques that impose certain conditions on the initial parameters of the model like Weight intialization and initial step size. And finally we tried Weight Decay that consists in adding a regularization term to the loss function. We realised experiments on the CIFAR-10 dataset using a convolutional neural networks, where the task is perform multi-class classification.

References

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[3] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *NIPS*, 1991.

[4] Yann LeCun, Leon Bottou, Genevieve Orr, and Klaus Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade*, Lecture Notes in Computer Science, chapter 2, page 546. Springer Berlin / Heidelberg, 1998.

[5] Yuanzhi Li, Colin Wei, and Tengyu Ma. Towards explaining the regularization effect of initial large learning rate in training neural networks, 2019.

[6] Christian Szegedy Sergey Ioffe. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2015.

[7] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.