Training Deep Models

General Optimization techniques

Moritz Kirschte, INF103333

Summer Seminar, 06-05-2018

My seminar work in the summer semester 18' supervised by Prof. Dr. D. Säring goes about Optimization in Deep Machine Learning. Being a part of such a huge seminar offers me and the other attendees the ability to focus on a specific topic in the complex field of Machine Learning, rather than wasting time and losing details by giving just short overviews. Basic understandings of Linear Algebra and Machine Learning techniques as e. g. presented in the first part of the DeepLearningBook by Goodfellow et. al [GBC16], namely knowing how to build, code and train a (Deep) Neural Network, is a requirement of this paper. I decided to explicitly announce this target group restriction, to be more focused on very advanced and complex Optimization techniques. As the projects payoff goes hand in hand with the ability to train fast on big machines e. g. the NVIDIA DGX-II [NVI18], a few thoughts on optimization techniques seems to be well-suited.

I will start with introducing some advanced learning techniques, afterward stay on a powerful normalization thinking in every layer of your neural network and finally conclude with an extra and not usually mentioned chapter when talking about optimization techniques called feature optimization. After reading this paper, you should be fully aware of the most important advanced optimization techniques and you are able to train your model much faster.

Contents

1 Motivation

Before introducing first optimizations, a short moment of breath holding is appropriated.

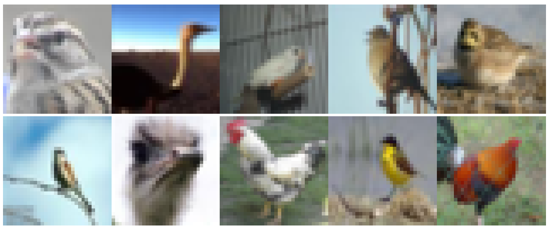
It'll be a Sisyphean task in convincing you to invest time in optimization if you're not able to realize its need. The CIFAR-10 dataset is a perfect example of concurrently visualize the progress. In the beginning, we won't be able to take our Deep Neural Network seriously, but after enhancing the Base-AI written in PYTHON chapter by chapter, advancement will be self-evident.

1.1 Example: CIFAR-10

CIFAR-10 [Kri09], created by ALEX KRIZHEVSKY, VINOD NAIR, and GEOFFREY HINTON, is a benchmark task for object recognition using 32x32 downsampled color hand-labeled images of 10 different object classes: *airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks*. It is divided into 50,000 training images and 10,000 test images.

Even modern AI algorithms are not well performing on this dataset (currently round about 2 % error rate), so it isn't as simple as an XOR-prediction, for which even a simple Logistic Regression model fits perfectly. Furthermore, you are able to visualize the error easier than with non-visual data.

Figure 1: CIFAR-10 dataset: Bird ($y = 2$)



1.2 Problem-Description

Everybody should be familiar with Moore's law claiming that every 18 to 24 month the number of transistors on ones chip gets doubled.

But according to Wirth's law [Wir95], who attributed the saying to MARTIN REISER: "The hope is that the progress in hardware will cure all software ills. However, a critical observer may observe that software manages to outgrow hardware in size and sluggishness.', software getting slower more rapidly than hardware becomes faster.

At the time optimization isn't well integrated with the program, the first results of this first intuition aren't promising. After $3,000$ iterations and half an hour of computation time with only using $10,000$ instead of $50,000$ low-resolution images in a logistic regression (1-Layer) NN, results of 41 % accuracy in the train set and 38 % in the test set are present.

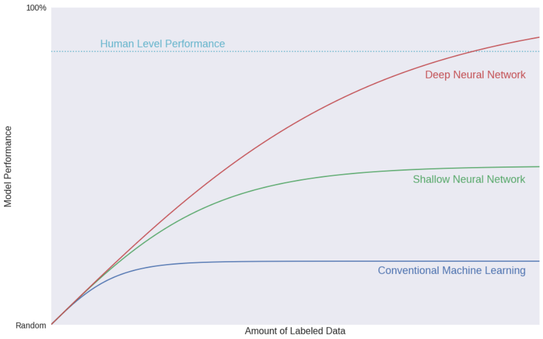
Concluding: Simply trust our computational power more than our brain might not be a good idea for Deep-NNs.

And good model performance comes only from deep and large Neural Networks.

2 Optimizer

The first topic, I want to present addresses directly the performance improve-

Figure 2: Performance comparison of Neural-Networks of different Deepness



ment. Deciding to start with this topic is not caused by actual relevance to the accuracy as regularization does, it's because building a house on sand rather than solid rock is not a good idea.

2.1 Batch Gradient Descent

Given a multi-dimensional function $\mathcal{F}(x)$ Gradient Descent takes an estimate for any input a_i on how to change the parameter a best to get a smaller function value $\mathcal{F}(a_{i+1})$ using the fact, that $\mathcal{F}(x)$ decreases fastest if one goes from a in the direction of the (negative) gradient at a: $-\nabla\mathcal{F}(a)$:

- guess x_0
- calculate next $(n \geq 0)$: $x_{n+1} = x_n - \gamma_n \nabla\mathcal{F}(x_n)$
- hopefully convergence: $\mathcal{F}(x_0) \geq \mathcal{F}(x_1) \geq \mathcal{F}(x_2) \geq \ldots$

You have to choose the *learning rate* γ carefully, because too big step sizes diverge and too small step sizes take too long per iteration but can reach a better estimate after reaching the region of convergence. A thoughtful combination would also be to introduce *learning rate decay* with a second parameter β affecting how much the learning rate will decay at the next total iteration step size in order to be fast at the beginning and slower and more precise at the end. Problematic on Gradient Descent is additionally, that there is a potential to get stuck in a suboptimal local minimum missing the global minimum, but even in higher dimensional spaces, such local minimums are very rare, because it has to be in every dimension a minimum. If not, it'll be a saddle-node and there is even one last direction to move onto. Some kinds of neural networks like e. g. logistic regression can guarantee, that they have only one minimum.

The way how to implement Gradient Descent in a Neural Network is described in the following algorithm.
Given the weights on each layer l $W^{[l]}$ and $b^{[l]}$, the training matrices $(X \in \mathbb{R}^{n,m}, Y \in \mathbb{R}^{1,m})$ and the learning rate, also called step size, α:

Listing 1: Batch Gradient Descent Algorithm

```
forward_prop(Input $\mathcal{X}$,
    Weight $\mathcal{W}^{[l]}$, Bias $b^{[l]}$)
Costs $\mathcal{J} \leftarrow \frac{1}{m} \times \sum_{i=1}^{m} \mathcal{L}oss(\hat{y^{(i)}}, y^{(i)})$
back_prop($\mathcal{A}^{[L]} = \hat{\mathcal{Y}}$, Output $\mathcal{Y}$)
$\mathcal{W}^{[l]} \leftarrow \mathcal{W}^{[l]} - \alpha \times d\mathcal{W}^{[l]}$
$b^{[l]} \leftarrow b^{[l]} - \alpha \times db^{[l]}$
```

These steps are repeated as long as a certain cost is reached or a random number of steps is taken.

The overall problem of *Gradient Descent* is, that performance issues are basically lying on m partly-redundant training examples (e.g. $320,000,000$ US-citizens). The next step of gradient descent can only be done after summing up over all m training examples ignoring the fact, that some of the training examples might be redundant and even the first 1 % offer a quite good estimate of the right direction.

2.2 Stochastic GD (SGD)

Stochastic Gradient Descent (SDG) directly addresses this problem of *Gradient Descent* by calculating the loss and updating the gradients every i^{th} training example. One step towards all training examples is called an *epoch*. It's important to note, that shuffling the data before applying *Stochastic Gradient Descent* is important, so that our AI doesn't learn something about the order of how we feed in the examples.

But sadly, this technique adds a lot of noise and even towards the minimum it dither back and forth without any convergence. Secondly, the per iteration performance trickles down significantly, due to the missing vectorization possibilities offering parallelism afforded by modern computing platforms.

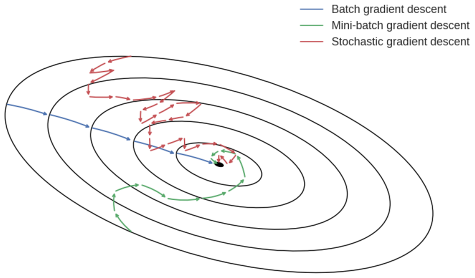
Listing 2: Stocastic Gradient Descent Algorithm

```
$shuffle([\mathcal{X}, \mathcal{Y}])$
for t in range(1,m):
    forward_prop(Input $\mathcal{X}^{(t)}$,
        Weight $\mathcal{W}^{[l]}$, Bias $b^{[l]}$)
    Costs $\mathcal{J} \leftarrow \mathcal{L}oss(\hat{y^{(t)}}, y^{(t)})$
    back_prop($\mathcal{A}^{[L](t)} = \hat{y^{(t)}}$, Output $y^{(t)}$)
    $\mathcal{W}^{[l]} \leftarrow \mathcal{W}^{[l]} - \alpha \times d\mathcal{W}^{[l]}$
    $b^{[l]} \leftarrow b^{[l]} - \alpha \times db^{[l]}$
```

2.3 Mini-Batch

As often in life the best solution is in the moderate mean rather than in extremism: *Mini-Batch* is a powerful combination of *Stochastic Gradient Descent (SDG)* and the *Batch Gradient Descent*. By choosing common batch sizes like 64, 128, 256 or 512 it offers even for big training sizes ($m > 2,000$) huge performance improvements, because it can be computed in one GPU-clock without using only parts of the GPU stucking in organization overheads and it makes progress without processing the entire training set:

Figure 3: Noise behavior in different GD-algorithms visualized on a contour plot



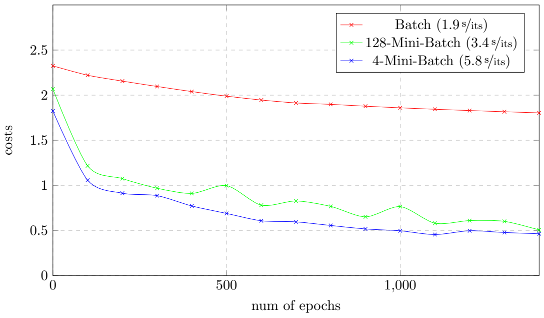
Given: Split of dataset in batches of sizes greater than one: $(x^{\{t\}}, y^{\{t\}})$

Listing 3: Mini-Batch Gradient Descent Algorithm

```
$shuffle([\mathcal{X}, \mathcal{Y}])$
for t in range(1,$batch\_size$):
    forward_prop(Input $\mathcal{X}^{\{t\}}$,
        Weight $\mathcal{W}^{[l]}$, Bias $b^{[l]}$)
    Costs $\mathcal{J} \leftarrow \frac{1}{batch\_size} \times$
        $\sum_{i=1}^{batch\_size} \mathcal{L}oss(\hat{y^{\{t\}}}, y^{\{t\}})$
    back_prop($\mathcal{A}^{[L]\{t\}} = \hat{y^{\{t\}}}$, $y^{\{t\}}$)
    $\mathcal{W}^{[l]} \leftarrow \mathcal{W}^{[l]} - \alpha \times d\mathcal{W}^{[l]}$
    $b^{[l]} \leftarrow b^{[l]} - \alpha \times db^{[l]}$
```

Mini-Batch Gradient Descents is able to improve our test accuracy by 22 % only by giving the opportunity to reach a lower

Figure 4: Comparison of the Gradient Descent family performance on CIFAR-10



costs while using the same number of Epochs $(1,500)$ despite the increasing duration per Epoch, whereas nearly *Stochastic GD* is able to decrease the costs even faster, but it takes in my example with the *CIFAR-10* dataset almost twice as long as *Mini-Batch*, so it's not recommended.

2.4 Momentum

According to Rumelhart, Hinton and Williams' paper on backpropagation learning [RHW86], *Stochastic gradient descent with momentum* remembers the update $\Delta\mathcal{W}$ at each iteration and determines the next update as a linear combination of the current gradient and the previous update:

$$\Delta\mathcal{W} \leftarrow \beta\Delta\mathcal{W} - \alpha\nabla\mathcal{L}(\mathcal{W})$$
$$\mathcal{W} \leftarrow \mathcal{W} + \Delta\mathcal{W}$$

2.4.1 Running Average

Before presenting a concrete example of the most popular linear combination used for *momentum* in deep learning, I first would like to start with a more basic idea of running average, the so called *Simple moving average*:

$$SMA_{t-\tau}^{(n)} = \frac{1}{n}\sum_{i=t-n}^{t} \mathcal{X}_i$$

$$n \mathrel{\widehat{=}} \text{memory}$$

$$\tau \mathrel{\widehat{=}} \text{group delay} = \frac{n-1}{2}$$

Just summing over the last n examples divided by n (arithmetic mean) is for most cases a good idea, but the main disadvantage is, that it needs a memory (size: n) to fulfilled before using this technique is possible and it has got a delay about τ. In cases, where the future is known and you want to draw something like a trend curve, you can shift the curve by τ. Since Gradient Descent is a causal system, this is not an option.

The *Exponential moving average* in contrast is able to handle those problems. Towards *bias correction* it has got even the ability to produce a great estimate even before the mean of $\approx \frac{1}{1-\beta}$ values is reached. With a weight of β, the current value will be added to the running average and afterward normalized for β, so that every value in the past of loses more and more influence as easily seen in the non-recursive formula: $\lim\limits_{i\to\infty} \beta^i = 0$.

Biased moment estimate EMA_t:

$$EMA_t = \beta \times EMA_{t-1} + (1-\beta) \times \mathcal{X}_t$$

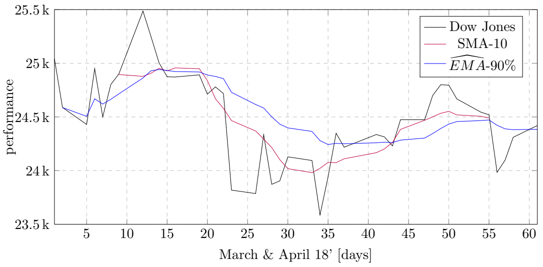
$$EMA_t = \sum_{i=0}^{t} \beta^i (1-\beta)\mathcal{X}_{t-i} \Leftarrow \text{non-recursive}$$

bias-corrected moment estimate $\widehat{EMA}_t$:

$$\widehat{EMA}_t = \frac{EMA_t}{1-\beta^t}$$

$$\beta \mathrel{\widehat{=}} \text{decay rate}$$

Figure 5: Illustration of both running averages on *Dow-Jones* share prices



2.4.2 Stochastic Gradient Descent with Momentum

Satisfying the requirements of a *momentum* algorithm, in *exponential weighted average* old values loses more and more of significance because with every step the *decay rate* as a factor of each gradient in each example gets minimized during to the increasing exponent (see non-recursive formula). These are leading factors for the acceptance of *Exponential moving average* as the algorithm behind *momentum*. To be detailed *bias correction* is not part of the so-called *Stochastic gradient descent with momentum*, because it's only relevant in the early learning period and makes computation more inefficient.

Listing 4: `Stochastic Gradient Descent with Momentum` Algorithm

```
$v_0 \leftarrow \vec{0}$
for t in range(1,batch_size):
    forward_prop(Input $x^{\{t\}}$,
        Weight $W^{[l]}$, Bias $b^{[l]}$)
    Costs $J \leftarrow \frac{1}{batch\_size} \times$
        $\sum_{i=1}^{batch\_size} Loss(\hat{y}^{\{t\}}, y^{\{t\}})$
    back_prop($A^{[L]\{t\}} = \hat{y}^{\{t\}}$, $y^{\{t\}}$)
    $v_{dW} \leftarrow \beta_1 \times v_{dW} + (1 - \beta_1) \times dW$
    $v_{db} \leftarrow \beta_1 \times v_{db} + (1 - \beta_1) \times db$
    $W^{[l]} \leftarrow W^{[l]} - \alpha \times v_{dW}$
    $b^{[l]} \leftarrow b^{[l]} - \alpha \times v_{db}$
```

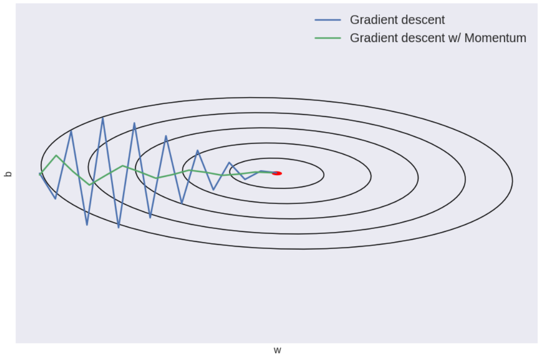
The main difference between *Stochastic Gradient Descent without Momentum* can be seen in the algorithm in Listing 4, using the results of our average system as gradients. With β we introduce a new hyperparameter in order to efficiently update new incoming values called exponential decay rate for moment estimate. According to the paper owners, a value round about 0.9 seems useful, so there is the potential, but not the actual need to tune this hyperparameter, because tuning the learning rate or the regularization factor takes better improvements to the model.

2.5 RMSProp

The principle of *RMSProp* [TH12] is in most terms the same as it of *Gradient Descent with Momentum* besides the fact, that instead of computing exponential running average without any scale, a squared version of the input into the average modulation is useful, because a massive increase in *db* (and ς) get caught by the final gradient $^{db}/_{\sqrt{\varsigma}}$ (also illustrated in Figure 6). Whenever you will, unfortunately, do an extraordinary high step going into the wrong direction e g. in oscillation, our running average ς will also increase significantly (because of the square). Due to the fact, that our final gradient is the fraction of the current gradient divided by the square root of the running average ς, the wrong and high value gets divided by a higher running average in order to get fixed in a more stable range.

E. g. if we consider, that our running average for *db* has a full memory and currently stays on $\varsigma = 2$, because all our previous gradients are in a normal behavior (final and somehow fixed gradient: $^{1.42}/_{\sqrt{2}} = 1$). Suddenly our gradient

Figure 6: Non-optimized learning algorithms often suffer of a high oscillation in b or other dimensions whereas Momentum seems to be more resistant



db jumps up to 6, so our new running average denotes by using a β_2 of 0.9 to $\varsigma = 0.9 \times 2 + 0.1 \times 36 = 5.4$, but luckily our final gradient doesn't change that much: ${}^{6}/_{\sqrt{5.4}} \approx 2.58$.

In the previously simple example, we changed our final gradient by round about half the original value. This would be a too big impact on the final learning algorithm since it should only behave as assistance system preventing *exploiting gradients* and not as a normalizing system. The plant archive this balance by choosing a very low hyperparameter $\beta_2 = 0.999$ having as same as in *Momentum* no need of real tuning.

N. B. the "hyperparameter“ ϵ can also be tuned because formally it is actually a hyperparameter, but since its only purpose is to grantee numerical stability in the denominator, there is no actual sense to handle it like this.

Listing 5: `Stochastic Gradient Descent with RMSProp` Algorithm

```
$\varsigma_0 \leftarrow \vec{0}$
$\epsilon \approx 10^{-8}$
for t in range(1, batch_size):
    forward_prop(Input $\mathcal{X}^{\{t\}}$,
        Weight $\mathcal{W}^{[l]}$, Bias $b^{[l]}$)
    Costs $\mathcal{J} \leftarrow \frac{1}{batch\_size} \times$
        $\sum_{i=1}^{batch\_size} \mathcal{L}oss(y^{\hat{\{t\}}}, y^{\{t\}}))$
    back_prop($\mathcal{A}^{[L]\{t\}} = y^{\hat{\{t\}}}$, $y^{\{t\}}$)
    $\varsigma_{d\mathcal{W}} \leftarrow \beta_2 \times \varsigma_{d\mathcal{W}} + (1 - \beta_2) \times d\mathcal{W}^2$
    $\varsigma_{db} \leftarrow \beta_2 \times \varsigma_{db} + (1 - \beta_2) \times db^2$
    $\mathcal{W}^{[l]} \leftarrow \mathcal{W}^{[l]} - \alpha \times \frac{d\mathcal{W}}{\sqrt{\varsigma_{d\mathcal{W}}}+\epsilon}$
    $b^{[l]} \leftarrow b^{[l]} - \alpha \times \frac{db}{\sqrt{\varsigma_{db}}+\epsilon}$
```

2.6 ADAM

All those optimizers have got in mind, that they offer the ability for more stable and faster learning. But in order to archive this goal, a simple implementation of the current state-of-the-art optimizer *ADAM* isn't enough, when not looking at the same time on your behind: The learning rate α can e. g. also highly increased and other hyper-parameters have to tuned another time.

The tuning process of hyper-parameters is a progress you have to think about for a moment. First, you need to be aware of all relevant and tunable hyper-parameters and its ranges ignoring such unimportant ones like ϵ, β_1 or β_2, to be more focused on the real advantages. A good way to start is with optimizing the learning rate α, the mini-batch-size and the number of hidden units; The number of layers and the learning rate decay hyper-parameter takes also some improvements. Secondly, you either sample these five hyper-parameters in a grid by choosing 10 values each, or you choose e. g. 50 random values and afterward you will zoom into the region of the lowest cost after some iteration of the development set and start sampling again in this small region. Ordering the different values for the hyper-parameters in a grid has the main disadvantage, that you only try for every parameter a small non-distinct number of values: By training the learning rate α and ϵ for numerical stability in a grid 5 values each e. g. you will ignore the fact that α is more important than ϵ and only 5 redundant values of α is trained in contrast to 25 distinct values using randomized sampling.

Transforming the scale of each hyperparameter to a logarithmically scale is also in most cases a good idea because, by this technique, you can shuffle your concrete values efficiently e. g. between 0.0001 and 1 for α or between 0.9 and 0.999 for β_1 in exponential weighted average in *Momentum*.

Listing 6: `Stochastic Gradient Descent with ADAM` Algorithm

```
for t in range(1, batch_size):
    forward_prop(Input $\mathcal{X}^{\{t\}}$,
        Weight $\mathcal{W}^{[l]}$, Bias $b^{[l]}$)
    Costs $\mathcal{J} \leftarrow \frac{1}{batch\_size} \times$
        $\sum_{i=1}^{batch\_size} \mathcal{L}oss(\hat{y^{\{t\}}}, y^{\{t\}})$)
    back_prop($\mathcal{A}^{[L]\{t\}} = \hat{y^{\{t\}}}$, $y^{\{t\}}$)
        $Momentum:$
    $\upsilon_{d\mathcal{W}} \leftarrow \beta_1 \times \upsilon_{d\mathcal{W}} + (1-\beta_1) \times d\mathcal{W}$
    $\upsilon_{db} \leftarrow \beta_1 \times \upsilon_{db} + (1-\beta_1) \times db$
        $RMSProp:$
    $\varsigma_{d\mathcal{W}} \leftarrow \beta_2 \times \varsigma_{d\mathcal{W}} + (1-\beta_2) \times d\mathcal{W}^2$
    $\varsigma_{db} \leftarrow \beta_2 \times \varsigma_{db} + (1-\beta_2) \times db^2$
        $Bias\,correction:$
    $[\widehat{\upsilon_{d\mathcal{W}}}, \widehat{\upsilon_{db}}] \leftarrow \frac{[\upsilon_{d\mathcal{W}}, \upsilon_{db}]}{1-\beta_1^{its}}$
    $[\widehat{\varsigma_{d\mathcal{W}}}, \widehat{\varsigma_{db}}] \leftarrow \frac{[\varsigma_{d\mathcal{W}}, \varsigma_{db}]}{1-\beta_2^{its}}$
        $Gradient\,update:$
    $\mathcal{W}^{[l]} \leftarrow \mathcal{W}^{[l]} - \alpha \times \frac{\widehat{\upsilon_{d\mathcal{W}}}}{\sqrt{\widehat{\varsigma_{d\mathcal{W}}}}+\epsilon}$
    $b^{[l]} \leftarrow b^{[l]} - \alpha \times \frac{\widehat{\upsilon_{db}}}{\sqrt{\widehat{\varsigma_{db}}}+\epsilon}$
```

Figure 7: Comparison of learning algorithms according to [KB15]

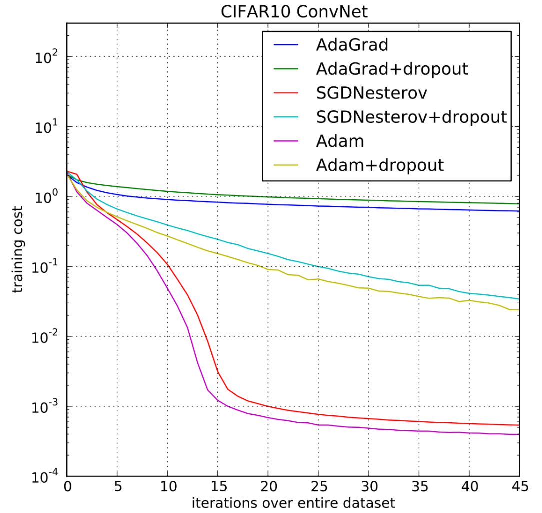
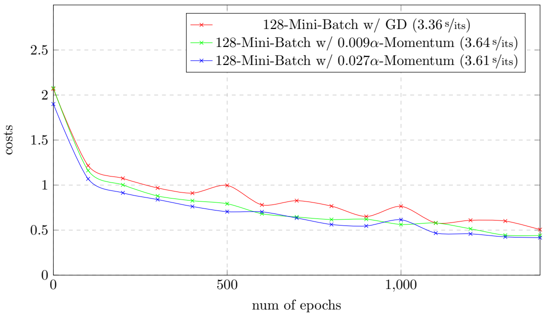


Figure 8: CIFAR-10 optimizer comparison



ADAM derived from *adaptive moment estimation* is the fastest and most intelligent state-of-the-art optimizing algorithm introduced by Kingma and Ba [KB15] combining *Momentum*, *RMSProp* and *Bias correction* with $\beta_1 = 0.9$ and $\beta_2 = 0.999$ (see algorithm in Listing 6).

2.7 Final Chapter-Remark

Concluding with one of the most complex and even most relevant topic when talking about Optimization Techniques in deep neural networks, you should note, that even

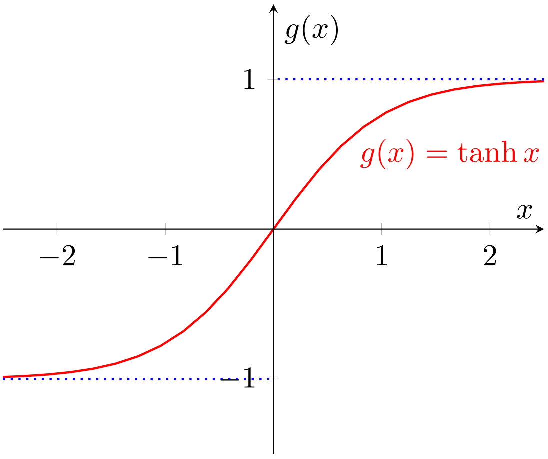
if nowadays *ADAM* is the best choice, in maybe 5 years it'll be totally outdated choosing *ADAM*. There will be better algorithms which save e. g. another minute in one hour and you'll be naive if you won't use them. Maybe you are even struggling with implementing *ADAM* on your own, but that's okay. You don't have to implement this error-like algorithm on your own, you should only know how modern optimization techniques safes times, so you confide *ADAM* or how the latest algorithm will be called and don't notice them as a mysterious black box.
As a black box, you also won't longer observe *Batch Normalization* due to the next chapter.

3 Batch Normalization

According to S. Ioffe and C. Szegedy [IS15] training a neural network is complicated, because of *Internal Covariance Shift*: The inputs to each layer are affected by the parameters of all preceding layers, so that continuously adapting to different distributions instead of real learning is necessary. Furthermore, there is the problem, that small changes to early network weights amplify as it becomes deeper. One well-known technique is to normalize the input data $\mathcal{X}$, but why not only normalize the global input and normalize the input to every layer $\mathcal{Z}$? This is exactly what Batch Norm is about.

Additionally, it introduces a slight regularization when using Mini-Batches by adding noise of different mean and variance each mini-batch (*Stochastic Gradient Descent* is, in this case, useless because normalizing only one input data isn't possible).

Figure 9: *tanh*-function



Taking a short look onto the activation function, we are able to observe, that it is useful to normalize before applying the activation function *tanh*. This lays down to the fact, that we implicit also fix the problem of vanishing gradients by keeping most weights in a range of -1 and 1 (normal distributed: $\sigma = 1$, $\mu = 0$), which is in the *tanh*-function the area, where there are the highest possible gradients. Without Batch Norm, we can get stuck in very high or low gradients slowing down the learning process. Choosing another activation function like e. g. *ReLU* - *rectified linear unit* $f(x) = x^+ = max(0, x)$ in the hidden layers and a *softmax* (multi-class classification) or *tanh* (single-class classification) for the output layer in order to make predictions between 0 and 1 establishing the probability of its occurrence might also one possible solution to the *vanishing gradient* problem. But this doesn't fix the slow learning process caused by *internal covariance shift* (see above).

Before implementing BatchNorm $BN_{\gamma,\beta}(x_i)$ with *tanh* or the outdated *sigmoid*-function you should be aware of the fact, that both are almost non-linear in the region between -1 and 1, which would reduce the complexity significantly, if you not always shift and scale to $\sigma = 1$ and $\mu = 0$, but introduce two learnable parameters γ and β (not confuse with β in *Momentum*) deciding themselves which is the best scale and shift. If the neural network learns e. g. for some layer a γ of $\sqrt{\sigma^2 + \epsilon}$ and a β of μ you can conclude that $y_i = x_i$ must hold and Batch Norm takes no effect.

Listing 7: `Batch-Norm` Algorithm

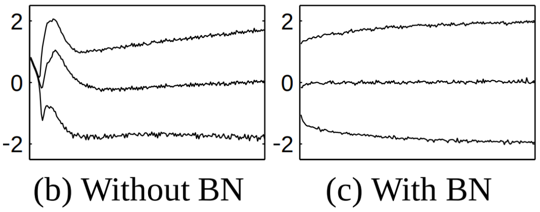
```
$\epsilon = 10^{-8}$
$\mathcal{B}= \{x_{1...m}\}, m > 1 \;\; \Leftarrow Mini - Batch$
mini-batch mean:
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$
mini-batch variance:
$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$
normalize:
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$
scale and shift:
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$
```

Listing 8: `Batch-Norm` Algorithm in a Neural-Net Layer

```
$\mathcal{Z}^{[l]} \leftarrow \mathcal{W}^{[l]} a^{[l-1]} + \cancel{b^{[l]}} \nearrow \vec{0}$
$\mathcal{Z}_{norm}^{[l]} \leftarrow \frac{\mathcal{Z}^{[l]} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$
$\hat{\mathcal{Z}^{[l]}} \leftarrow \gamma^{[l]} \mathcal{Z}_{norm}^{[l]} + \beta^{[l]}$
```

N. B. that you don't even need the bias b anymore (you can either remove it or replace it with the zero-vector $\vec{0}$), because you will shift the input by β so it would be redundant to maintain the bias, too. Secondly, you can also decide to use in contrast to my algorithms in this chapter another learning function like *ADAM*, but for simplicity, I decided to use *Gradient Descent*.

Figure 10: Distributions in the last hidden layer (15, 50, 85 percentiles) [IS15]



(b) Without BN (c) With BN

Listing 9: `Batch-Norm` Algorithm in a Deep Neural-Network

```
for t in range(1,$batch\_size$):
    forward_prop(Input $\mathcal{X}^{\{t\}}$,
        Weight $\mathcal{W}^{[l]}$, ~~Bias~~$\cancel{b^{[l]}}$)
    Costs $\mathcal{J} \leftarrow \frac{1}{batch\_size} \times$
        $\sum_{i=1}^{batch\_size} \mathcal{L}oss(\hat{y^{\{t\}}}, y^{\{t\}}))$
    back_prop($\mathcal{A}^{[L]\{t\}} = \hat{y^{\{t\}}}$, $y^{\{t\}}$)
    $\mathcal{W}^{[l]} \leftarrow \mathcal{W}^{[l]} - \alpha \times d\mathcal{W}^{[l]}$
    $b^{[l]} \leftarrow b^{[l]} - \alpha \times db^{[l]}$
    $\beta^{[l]} \leftarrow \beta^{[l]} - \alpha \times d\beta^{[l]}$
    $\gamma^{[l]} \leftarrow \gamma^{[l]} - \alpha \times d\gamma^{[l]}$
```

3.1 Final Advices

The original paper had got space left for some advanced advises how to fit *Batch Norm* best into your existing neural network. You have to keep that in mind, because just implementing *Batch Normalization* without rethinking your hyperparameter won't be that successful:

- Increase learning rate (by factor ˜5 onwards), because Batch Norm adds more stability to the learning process only focused on the decision boundary itself in a more or less fixed distribution.
- Remove dropout since Batch Norm already included a slight regularization
- Reduce L_2 Weight-Regularization (by ˜5)
- Accelerate learning rate decay because convergence gets improved
- Shuffle training examples more thoroughly because Batch Norm relies for an optimal distribution-fix on real randomness with respect to each mini-batch. Each mini-batch should have approximately the same distribution as all data in order to have a fixed distribution.

As already mentioned in my final chapter-remark in the *Optimizer*-section you can implement *Batch Norm* on your own, but you have to adapt also your backpropagation algorithm and be aware of it on testing time. *Gradient Checking* will be one of the rare techniques of verifying that you implement your backpropagation algorithm including Batch Norm correctly, but most of the time a little mistake in your program easily caused when dealing with such complex algorithms, can significantly disturb your model performance. So if you have an overall understanding Batch Normalization you are better staying at a preprogrammed AI-library as e. g. *TensorFlow* instead of program everything on your own.

4 Feature-Optimization

Personally, I realized during my seminar work that this chapter is one of the important ones when dealing with Optimization in Deep Learning, not because it has got the biggest impact, but because it's an underestimated optimization technique. Jesse Moore claims:

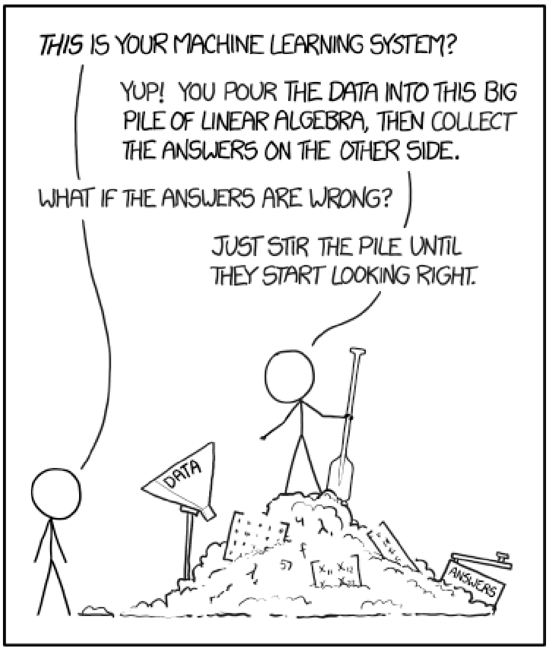
> You should spend time developing tools to analyze, present, and study your input data. [...] Avoid thinking of data as merely an input. All data has hidden features that hold immense predictive power if you can extract them. **Data is greater than model design, and model design is greater than parameter optimization.**

And that's basically how it is: More important than disturbing the pipeline in the middle and forcing with any power you can obtain in order to get your result you want, machine learning is more about thinking and understanding your features. If you don't analyze what's really relevant for your prediction you get lost in spending computational power for heating your home and not more.

Speaking from the perspective of the cartoon in Figure 11: Instead of improving how to stir your pile best, you're better with decrease your pile's height.

> With too little data, you won't be able to make any conclusions that you trust. With loads of data you will find relationships that aren't real [...] **Big data isn't about bits, it's about talent.**
> – Douglas Merrill

Figure 11: The real importance in Deep Learning @$xkcd^{1838}$

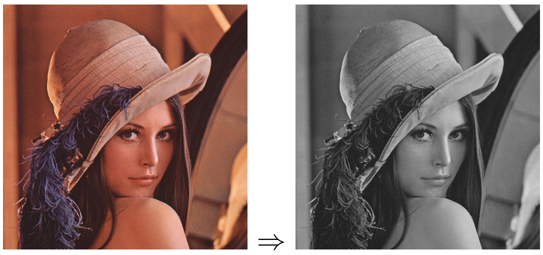

THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

4.1 Possible Feature Reductions

To get in touch with Feature Optimization we should first start asking ourselves what actual enhancements we can reach.

E. g. let's assume our Logistic Regression Network of $\{3072, 10\}$ layer dimensions on the CIFAR-10 dataset using the approach that every RGB-value on every pixel in our image should be one feature being computational inefficient. But that's actually not the best representation and has got a lot of redundancy: Nearby pixels are highly dependent and even in the corners of the image or in the color there is no new information. Shannon's entropy states that there have to be a better representation as e. g. used in *JPEG*-compression or grayscaling.

Figure 12: Lena in Color and Gray

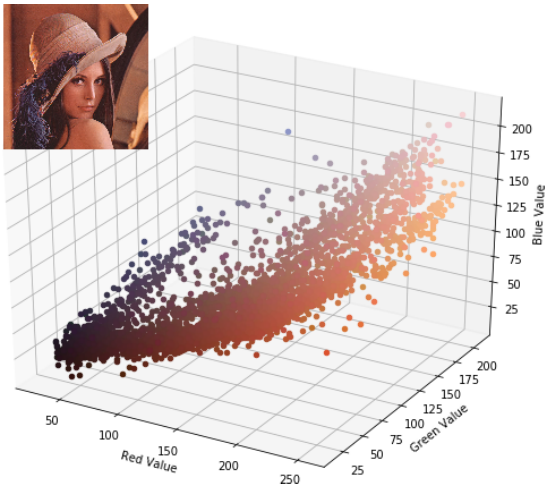


4.2 Grayscaling

Grayscaling is one example of a **input-independent technique**, because some brilliant scientists have observed a magic formula of how 3-channel-color images looks best for human eyes according to the fact that we are most sensitive for green color: $Y_{Gray} = 0.2126 \times R_{ed} + 0.7152 \times G_{reen} + 0.0722 \times B_{lue}$. With this method, we also get a high contrast. This has got huge performance benefits because the computation complexity is linear, but on the other hand, we have the main disadvantage, that we first need to observe this static formula transforming the features. In most cases and even when handling with non-image data we haven't got that ability, even when keeping in mind that the sense of making predictions with a Deep Neural Network is to be not relevant on which detailed feature correlation you might have. If you start analyzing like this you should think twice if a Neural Net is what you really want.

Let's consider we have got a colored image of *Lena* the most famous playboy-girl in computer science. If we grayscale her we can reduce our feature size by 3: Now we have $\{262.144, 10\}$ instead of $\{786.432, 10\}$.

Figure 13: Lena's scatter plot in R-G-B-domain



4.3 DCT / DFT

But there is another group of feature reduction techniques namely **input-dependent techniques**. Examples might be *Discrete Cosine Transformation – DCT* or *Discrete Fourier Transformation – DFT* being able to change an input's data to an abstract frequency representation, whereas low frequencies are that ones who gives the global structure towards an image and high frequencies are responsible for exact borders. Cutting high frequencies only has the disadvantage, that your image gets a little bit of fuzziness, but you don't lose that many information. This method is therefore state-of-the-art in image compression.

4.4 PCA

Another possible feature reduction technique is to use *Principal Component Analysis* also called *PCA*, which is state-of-the-art in Deep Learning. Conceptual it is completely the same as for *DCT*, but we transform our data in the eigenvectors domain, where each eigenvector is orthonormal to all other eigenvectors scaled by his eigenvalue. So we project our features to those top-level eigenvectors, which have got the highest eigenvalues because in this dimension there is the highest variance and we lost almost no information. This algorithm is called *principal axis theorem.*

In Figure 14 we see two orthogonal eigenvectors scaled by there eigenvectors, one with a smaller eigenvalue than the other. We would choose to project our data to the bigger eigenvector in order to retain the main information. The other direction can be interpreted as noise so we won't get information from that one.

Consider *Lena* again and how we transformed her last time. Now we use PCA and forget the static formula of how to make a good grayscale image by finding the best fitting plane for this concrete image in the scatter plot, which maximized the entropy. The results in the histogram and final image can be seen in Figure 15 and Figure 16: We observe no big difference because the difference is in the algorithm and its dynamic adaption to all kinds of features. So because we already have this static formula for grayscaling, the more complex and dynamic PCA is more or less useless, but in most cases, we haven't got that.

4.4.1 correlation matrix

The correlation of input data can be visualized by the correlation matrix R_{ij} stating how much feature i is with feature j correlated: $R_{ij} = \frac{Cov_{ij}}{\sqrt{Cov_{ii} \times Cov_{jj}}}$ (Cov is the Covariance matrix of $\mathcal{X}$).

A Value of 1 means complete redundancy, a value of -1 means reciprocal redundancy and a value of 0 means complete independence. Via plotting a histogram of this matrix we can get a quick overview of

Figure 14: PCA: Projection onto the highest eigenvector (darkblue) would retain the overall structure due to its significant larger variance as represented by the green hulls (1σ / 2σ).

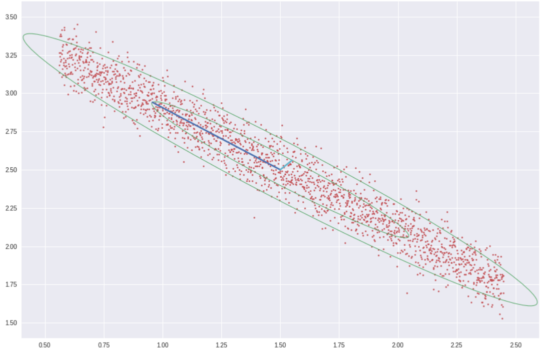


Figure 15: Histogram comparison of (a) classical and (b) PCA grayscale

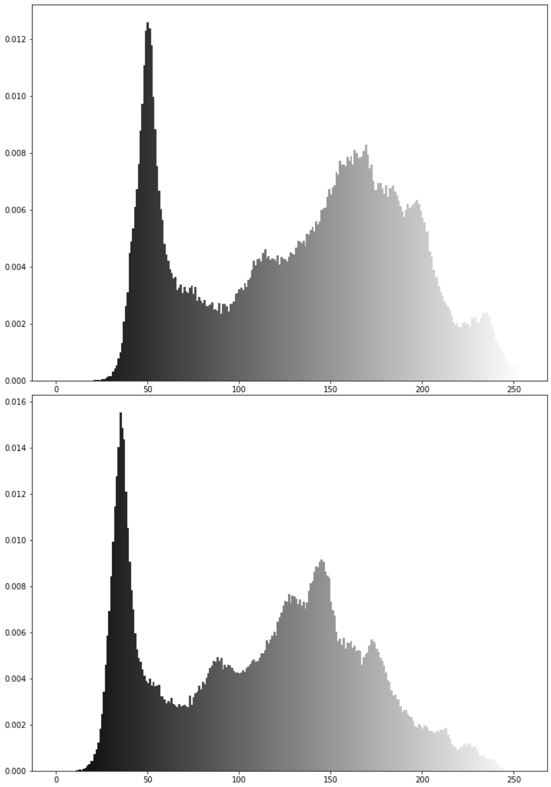


Figure 16: Image-Comparison of non-PCA and PCA grayscale

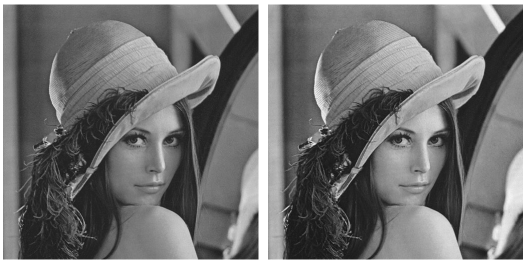
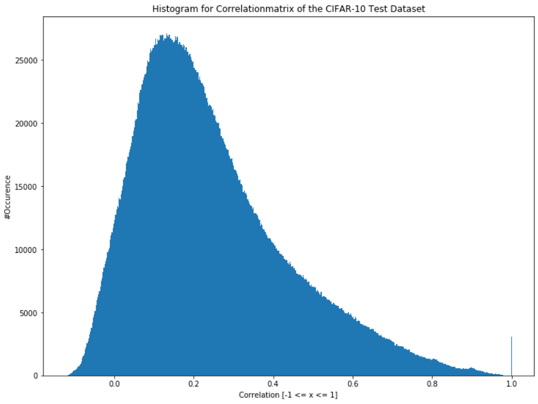


Figure 17: Pearson product-moment correlation coefficients between 0 and 1



how much *Lena* is correlated.

Thinking that everything is okay with *Lena* is in this case wrong because the most values are below 0.3 due to the fact that the upper left corner isn't much correlated with any value outside a range of 100 pixels. By looking into a certain region around the diagonal values, it's almost impossible to find values lower than 0.8. But that's how an image works. There is a high probability that the pixels around a certain one have got an equally RGB-value.

4.4.2 PCA algorithm

Formally and according to *Wikipedia* PCA is a statistical procedure using **orthogonal transformation** to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called **principal components**. It is defined in such a way that the first principal component has the **largest possible variance**, and each succeeding component, in turn, has the highest variance possible under the **constraint** that it is **orthogonal to the preceding components**.

Algorithmically spoken we try to reduce data $X \in \mathbb{R}^{n \times m}$ from n-dimensions to k-dimensions by:

- mean normalization ($\mu = 0$, $\sigma = 1$)
- Compute covariance-matrix: $\Sigma \in \mathbb{R}^{n \times n} = \frac{1}{m} \sum_{i=1}^{n} (x^{(i)})(x^{(i)})^T = \frac{1}{m} X^T \times X$
- Compute eigenvectors $U \in \mathbb{R}^{n \times n}$ descending sorted by eigenvalues of matrix Σ: $[U, S, V] = svd(\Sigma)$
- Select the first k eigenvectors from U: $Z \in \mathbb{R}^{k \times m} = U^T_{reduce} \in \mathbb{R}^{n \times k} \times X \in \mathbb{R}^{n \times m}$
- Variance retained (indicator of how much information we preserve): $\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}}$; $S \in \mathbb{R}^{n \times n} = diag(eigvalues)$

Singular Value Decomposition (SVD) offers us the ability to calculate eigenvectors and eigenvalues without assuming that the input matrix has to be a squared matrix. Since we only feed into the symmetric and squared Covariance matrix, this doesn't matter, but it's even more stable for singular covariance matrices, which could be a problem if one feature has always the same value.

Generally spoken, because of the symmetric nature of the Covariance matrix, we get by the diagonalizing process $A = P \times D \times P^{-1}$ an orthogonal eigenvectors matrix P and the diagonal eigenvalues matrix D. SVD states that $A = U \times S \times V$ whereas $U = P$ and $S = D$ so that the only difference towards the classical diagonalizing process is that we won't assume, that P needs to be invertible.

4.4.3 Eigenfaces

"Labeled faces in the wild" *(LFW)* is a dataset by the University of Massachusetts [HRBLM07] of ($X \in \mathbb{N}^{50 \times 37}$) dimensional images of popular faces, which enables huge visualization opportunities according to the idea of [SK87]. In Figure 18 we get a visualization of the Top 12 eigenfaces being our new features when training a neural network for these *LFW* dataset. For every eigenface there is a percentage available of how much information is in this particular eigenface. E. g. when using only the first eigenfaces we can retain almost 20 % of our original data, using 5/1850 of our data we retain 53 %, 20/1850 retains 75 %, 185/1850 retains 97 % and 500/1850 retains 99, 7 %.

By simply reducing our data by the factor 10 we are able to retain 97 % of our original image, which offers us the ability to train a lot faster and therefore compute more iteration giving us a better prediction. The loss of 3 % in the data is in this case negligible. Contrastive behavior can be observed if we reduce our data to 5 dimension, where we are not able to differ between e. g. Bush and Blair, so training on these images would be useless if we want to predict the different persons.

Figure 18: Eigenfaces Original samples vs. Top 12 eigenface-features

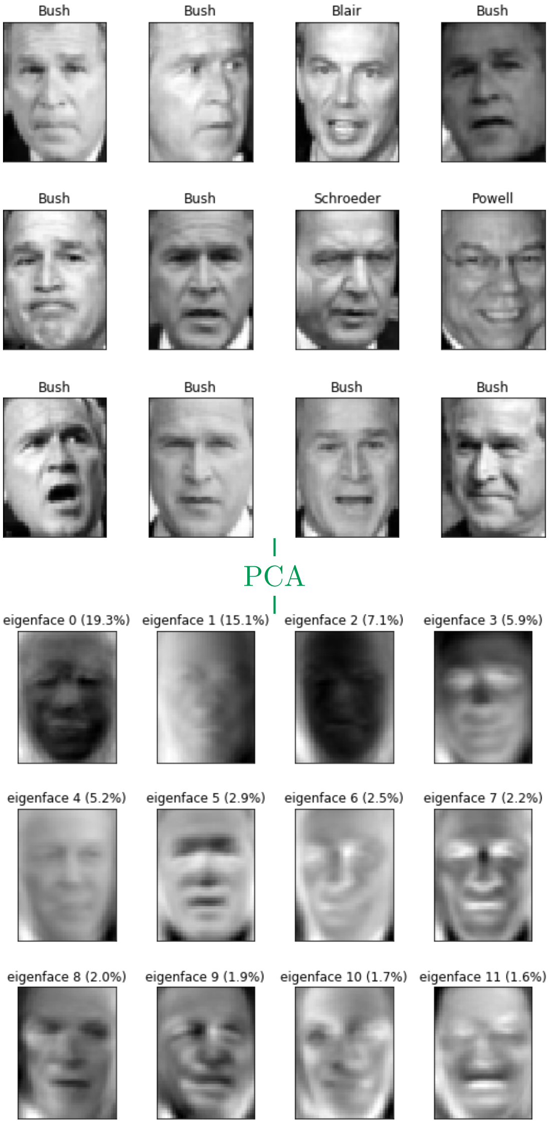


Figure 19: Eigenfaces – LFW: ${}^{5}/_{1850}$ - 53 %

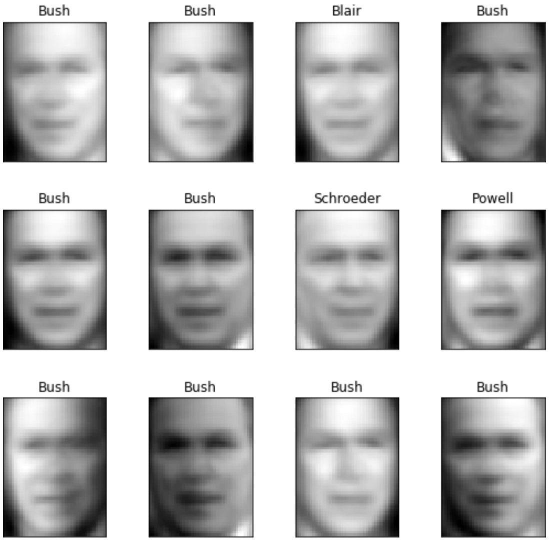


Figure 20: Eigenfaces – LFW: ${}^{20}/_{1850}$ - 75 %

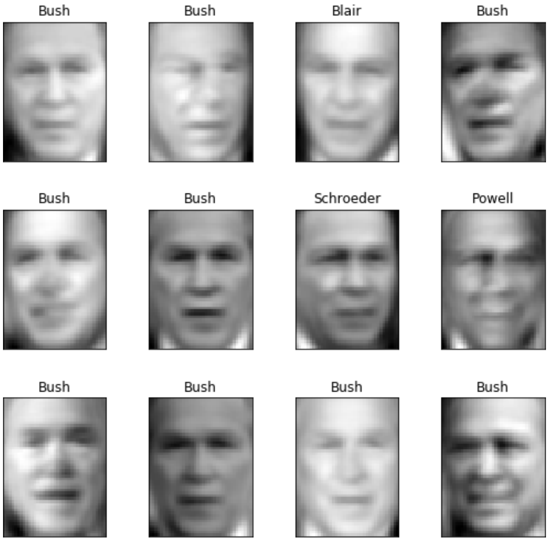


Figure 21: Eigenfaces – LFW: ${}^{185}/_{1850}$ - 97 %

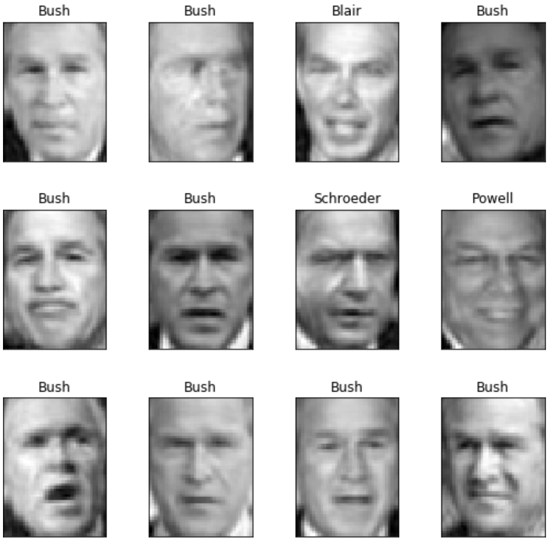


Figure 22: Eigenfaces – LFW: ${}^{500}/_{1850}$ - 99, 7 %

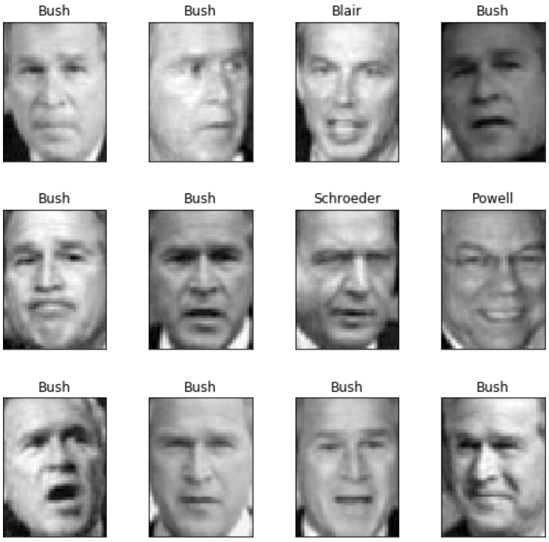
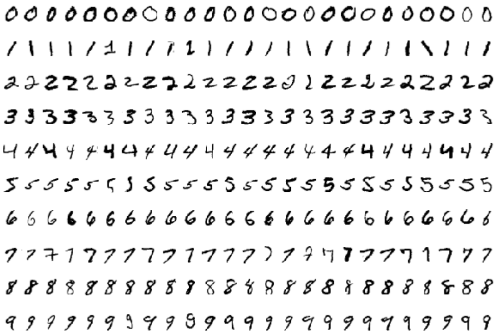


Figure 23: MNIST dataset [LCB98]



4.4.4 PCA for visualization

A last remark on PCA is about that it's also useful for visualization purposes in order to reduce very high dimensional data like e. g. the weights of our Neural Net trained towards the *MNIST*-dataset describing ten-thousands of handwritten digits in dimension 28 × 28 to 2D or 3D data in order to have the best view how to look on the dataset with the biggest variance, so you can see e. g. the predicted decision-boundary for the ten different digits and how good our network performs. Even for nun AI-purpose this visualization purpose is useful and takes advantages.

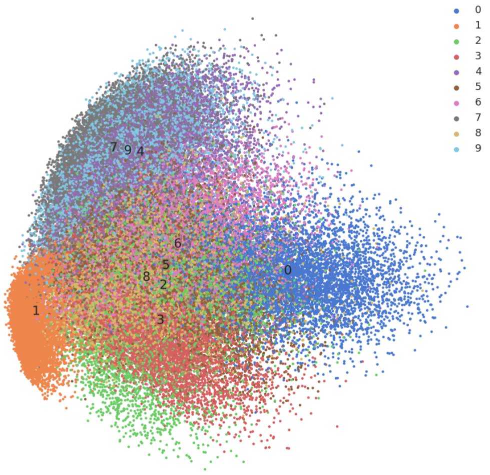
4.5 CNN

The last possible technique is to use a *CNN*, which has got all those feature reductions and optimization stuff included. But this isn't a part of my seminar work.

5 Closing remarks

To sum up this overview of common optimization techniques we learned not only what they're about but also how to use them and at which time in your project

Figure 24: PCA for Visualization – Projecting on the best fitting (i.e. maximal variance) plane throughout the 784 *dim* space of the 10 different MNIST-digits



it might useful to spend time towards certain optimization. We started with optimization in the model view with concrete *Gradient Descent* alternatives like *Momentum*, *RMSProp* or *ADAM*, going onwards to a very high-level *Batch-Normalization* and ended up with *Feature Optimization* emphasizing the importance of Data Analysis and Feature understanding.

As long as you are a master in machine learning, most time you will spend on gathering data, analyzing data and simplifying data. Since algorithms are already written in thousands of deep learning framework like *TensorFlow*, *Keras*, etc, it'll be not that useful to invest your time on that stuff even when you are already familiar with common deep learning techniques. Backpropagation algorithms can nowadays implicit inferred by how you implement forward-propagation, so you won't bother about this error-prone task anymore.

Even with techniques like *transfer learning* or *multi-task learning* you are able to use pre-trained weights and simply remove the last *softmax*-prediction-layer on that network and feed in those high-level learning features on the pre-last layer in your own new and simpler neural network. E. g. *YOLOv3* is a modern pre-trained CNN recognizing hundreds of different classes in images and videos. You can even use this network if you only want to predict traffic relevant objects like traffic lights/signs, cars, trucks, persons or bicycles by simply adding a logistic regression network on the end of *YOLOv3*s pre-last layer. This enables dozens of new business cases without relying on good learnable data and the long-time learning process.

End-to-End-Deep-Learning in contrast is something you should forget. Most of the times the amount of data needed for modeling your overall task with one single and powerful model is not countable. But on the other hand, only relying on your project on non-AI software might be also not the best solution. The combination of algorithms and AI and a feasible pipeline is currently the best way. E. g. an end-to-end model of predicting cancer or non-cancer on medical data is not realizable due to the small amount of available data. But a pipeline like first removing the background and somehow pre-marked the important area with *OpenCV* and afterward feeding this output to a neural net result in much higher prediction accuracy, because you ensure that the neural network is focused on the relevant data and doesn't get confused by wired-formed bones or other irrelevant data.

References

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[HRBLM07] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007. http://vis-www.cs.umass.edu/lfw/.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *Google*, March 2015.

[KB15] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *ICLR (conference)*, 2015.

[Kri09] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, April 2009. https://www.cs.toronto.edu/~kriz/cifar.html.

[LCB98] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits. Technical report, Courant Institute, Google Labs, Microsoft Research, 1998. http://yann.lecun.com/exdb/mnist/.

[NVI18] NVIDIA Corporation. *NVIDIA DGX-2 - The world's most powerful deep learning system for the most complex AI challenges*, April 2018. https://www.nvidia.com/en-us/data-center/dgx-2/.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, pages 533–536, October 1986.

[SK87] L. Sirovich and M. Kirby. Low-dimensional procedure for the characterization of human faces. *Journal of the Optical Society of America*, pages 4:519–524, 1987.

[TH12] Tijmen Tieleman and Geoffrey E. Hinton. Lecture 6.5 – rmsprop: Divide the gradient by a running average of its recent magnitude. Technical report, 2012.

[Wir95] Niklaus Wirth. A plea for lean software. *Computer*, pages 64–68, February 1995.

List of Figures