NTNU | Norwegian University of Science and Technology

DEEP LEARNING
DEEP NEURAL NETWORKS

Tor Andre Myrvoll
2020-10-12

Introduction - Outline

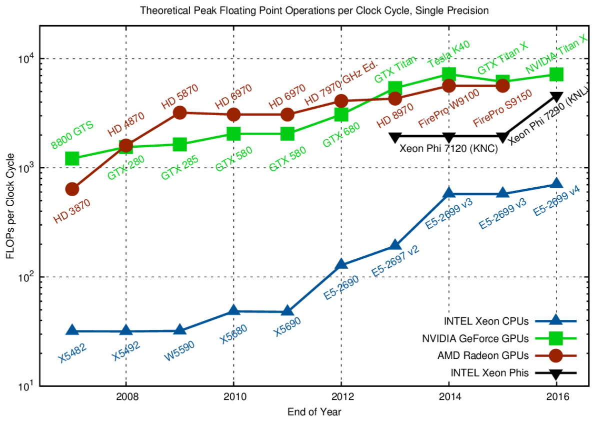
- Training DNNs – Common techniques
  - Batch, mini-batch and stochastic gradient descent
  - Normalization
  - Dropout
  - Variations on the gradient search
- Best practice
  - Training, validation and test data
  - Accuracy, precision, recall

Common techniques - Training on a GPU

Training a deep neural network using a GPU

- Training any reasonably sized DNN demands a massive amount of processing power
- Even the latest CPUs are no match for this challenge
- Luckily, the problem is *massively parallel*
- This means that we can use a large number of simple, specialized processing units, instead of a few very powerful, but generic ones
- A source of a large number of small, but efficient processing units is your graphics card (GPU)

Common techniques - Training on a GPU



Common techniques - Training on a GPU

- Another strength of a GPU is the very fast memory that it contains
- Deep learning problems are often very large, and accessing main memory of any computer system is very slow (relatively speaking)
- GPUs, on the other had, have very fast RAM, so if we can put the data and the model onto the GPU, we will have a significant speedup
- To make use of a GPU we should therefore
  - Have a parallel formulation of the problem
  - Use proper memory management

Common techniques - Batch, mini-batch, SGD

We remember that the objective function, our total loss, was written

$$L(W, B, X, Y) = \frac{1}{|X|} \sum_n l(y_n, f(x_n; W, B))$$

which in turn gave us the following expression of the *gradient* of the objective function

$$\nabla_W L(W, B, X, Y) = \frac{1}{|X|} \sum_n \nabla_W l(y_n, f(x_n; W, B))$$

We see that the gradient is the sum of contributions for each training sample $\{x_n, y_n\}$. This means that we can compute each contribution independently and *in parallel*.

Common techniques - Batch, mini-batch, SGD

We define three approaches to computing the gradient $\nabla_W L(W, B, X, Y)$ (note: a complete pass through the training set is called *an epoch*):

- Batch: Compute all elements of the gradient and sum them.
  - Parallelizable
  - Slow convergence per epoch
- Mini-batch: Pick a random subset of training samples and compute an approximation to the true gradient
  - Parallelizable
  - Fast convergence per epoch
- Stochastic Gradient Descent: Pick a single training example on random and compute an approximation to the gradient
  - Not parallelizable
  - Fast convergence per epoch

Common techniques - Batch, mini-batch, SGD

- Batch should never be used as it’s convergence is very slow. The reason for this is that we will only do one update of the model every we go through the entire training set.
- Mini-batch is the most popular approach by far. Selecting the number of examples used carefully, one can get a good approximation to the gradient, and update the model many times per epoch.
- The SGD gradient is very noisy, but on average the model will improve more per epoch than when using regular batch. Not parallelizable, but still used for some particular problems.

Common techniques - Normalization

Although feature extraction can be avoided in many cases for DNNs, some pre-processing is still in order:

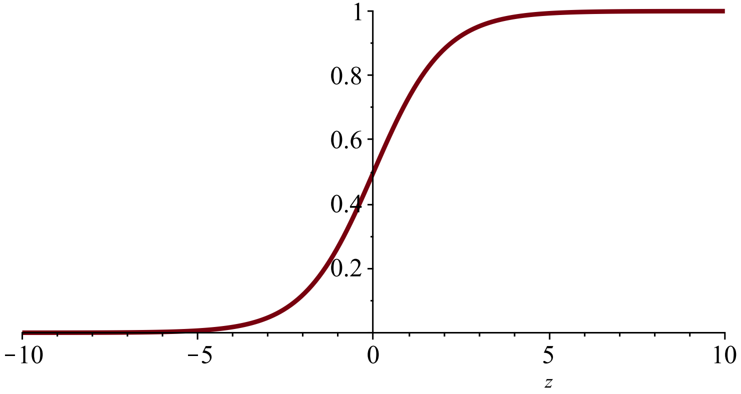
- Global normalization:
  - Mean and variance: Compute new features

$$x'_n = \frac{x_n - \bar{x}}{\bar{\sigma}}$$

where $\bar{x}$ and $\bar{\sigma}$ is the sample mean and standart deviation respectively.

- Global normalization is used to mitigate numerical problems, in particular those involving gradients.

Common techniques - Normalization



Common techniques - Normalization

Local Normalization

- Sometimes there are variation within the dataset that is detractive to learning
- Example: When recording speech from a number of speakers, the microphones and recording conditions may vary.
- This will in turn yield an additive mean value to the feature vectors that is unique per speaker
- In this case it will be usefull to compute the mean per speaker and subtract this

Common techniques - Regularization

Overtraining is one of the biggest problems across all machine learning methods, and DNNs are subject to this as well.

- We can control the overtraining by adjusting the number of free parameters, that is nodes and layers
- This is not very efficient however, as we need to retrain the network for every topology and test for overtraining.
- A better approach is to use *regularization*
- Regularization techniques were originally constructed to address ill-posed problems, for instance solving systems of linear equations $Ax = y$ where the matrix A is alsmost non-invertible.
- Regularization techniques can also be used to control the flexibility of a machine learning system

Common techniques - Regularization

L_p-regularization:

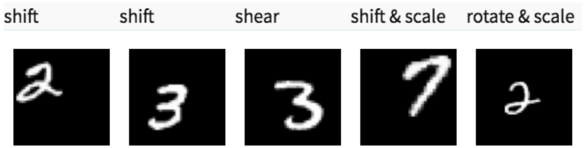
- In L_p-regularization we add a *penalty* to our optimization criterion

$$L(W, B, X, Y) = \frac{1}{|X|} \sum_n l(y_n, f(x_n; W, B)) + \lambda_W \|W\|_p + \lambda_B \|B\|_p$$

- The most common values for p are $p = 1, 2$, which corresponds to absolute values or sum of squares.
- Adding the L_p term will penalize the expression into avoiding large values for w_{ij}^l and b_i^l.
- This will reduce the flexibility of the network and avoid overfitting

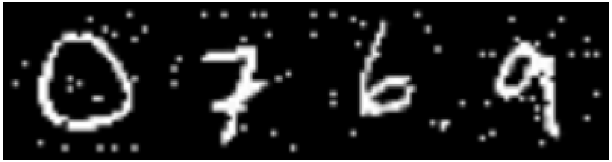
Common techniques - Data augmentation

- Sometimes the reason for overtraining is the lack of data, in the sense that the data is not spanning the space of possible observations
- We can *augment* the data set by adding carefully (and sometimes not so carefully), perturbed versions of existing data



Common techniques - Noisy augmentation

- We can also augment the data by adding noise to the training examples



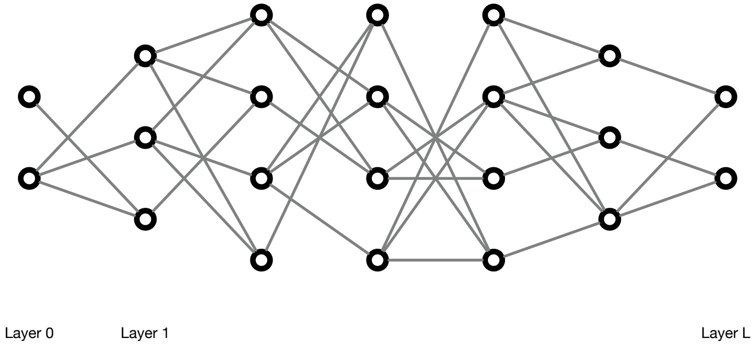
- Addind noise may seem counter productive, but if done correctly it will provide some *smoothing* of the learning machine, as it will need to fit the model to a large number of noisy examples instead of tuning the model to a small number of clean examples.

Common techniques - Dropout

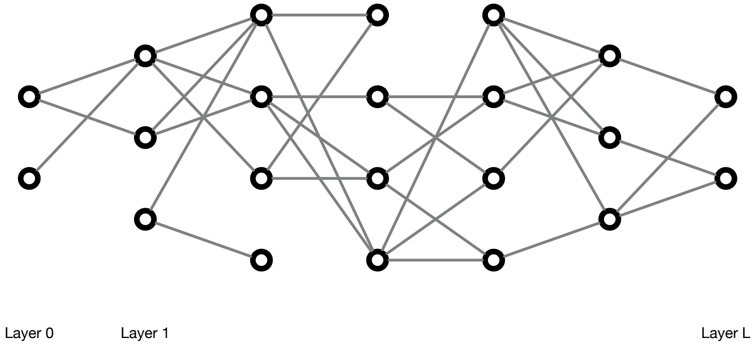
One of the most popular regularization methods is *dropout*

- For every iteration where the model is to be updated, remove a fraction c of the weigths, so that they don’t contribute
- Choose the weights to remove randomly every iteration

Common techniques - Dropout



Common techniques - Dropout



Common techniques - Dropout

- Dropout is in a sense a type of additive noise, behaving similarly to the noise augmentation training
- In addition, dropout *breaks dependencies*
- In principle, if two weights are summed at a processing node, the network may become dependent on the *difference* $w_{ij}x_i + w_{kj}x_k$.
- Dependencies like this are unwanted for a number of reasons. The weigths may for instance become unbounded, while still computing a valid difference *for the training set*

Common techniques - Gradient search

There are many variations on the concept of gradient search:

- Simple gradient search

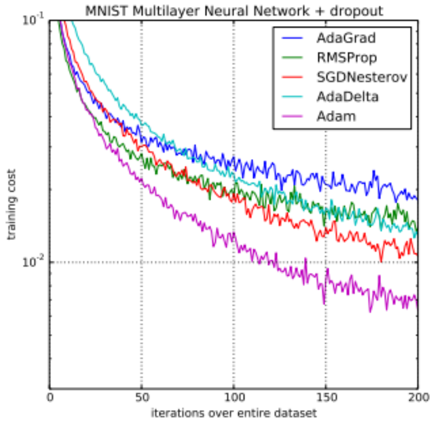
$$x^{i+1} = x^i - \epsilon \nabla_x f(x)$$

- Time dependent gradient search

$$x^{i+1} = x^i - \epsilon_i \nabla_x f(x)$$

- Gradient search with momentum (Nesterov)
- Adaptive learning algorithms: Adam, AdaGrad, AdaDelta...

Common techniques - Gradient search



NTNU | Norwegian University of Science and Technology

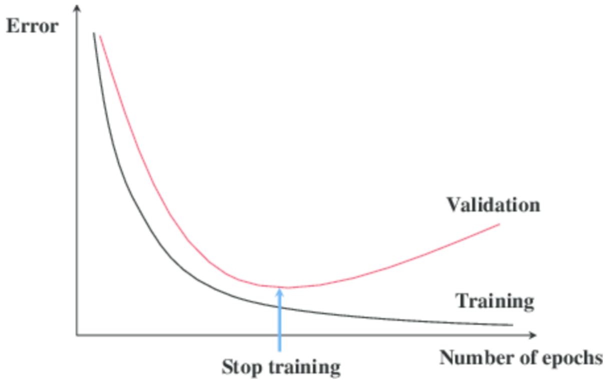
Best practice - Train, validation and test

One of the most important practices when doing any machine learning work is to define and use he following data sets correctly:

- Training data: This is the data that the model is based on directly. We use this data to compute gradients and update the model.
- Test data: This is the ultimate indicator of the performance of our model. This data should never be used in training, nor should it be used to optimize any hyper-parameters of the model.
- Validation data: This data set is to tune hyper-parameters, like model topology, learning rates, dropout rates and so on. It is never used to compute gradients and as such directly comtributing to the model.

Best practice - Use of validation set – early stopping

The validation set is often used to tune regularization parameters. One example is *early stopping*



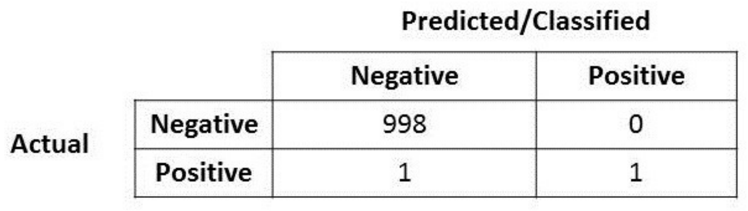
Best practice - Cross-validation

If resources permit it, it is beneficial to use *N-fold cross-validation*

- Take the training set and divide it into N equal parts
- Use one of the parts as validation data set
- Perform whatever performance tuning needed using the other $N - 1$ parts as the training set, measuring the performance on the valiation set.
- Do this N times.
- Choose optimal parameters by eg. majority vote, train the system using all available data and test on the test set

Best practice - Accuracy, precision, recall

For many machine learning tasks, the data may be severely skewed

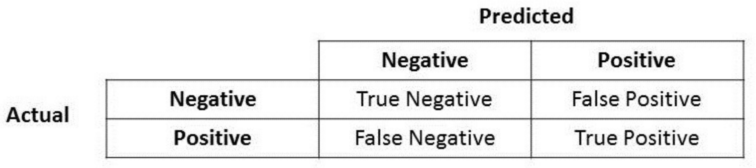


		Predicted/Classified	
		Negative	Positive
Actual	Negative	998	0
	Positive	1	1

In this case, even a classifier that *always* guesses negative, will be right in 99.8% of the cases.
But is this a good classifier?

Best practice - Accuracy, precision, recall

Let us define accuracy, precision and recall in terms of the confusion matrix



		Predicted	
		Negative	**Positive**
Actual	**Negative**	True Negative	False Positive
	Positive	False Negative	True Positive

Best practice - Accuracy, precision, recall

Definitions:

- Accuracy

$$\text{Accuracy} = \frac{\text{True positive} + \text{True negative}}{\text{Number of test examples}}$$

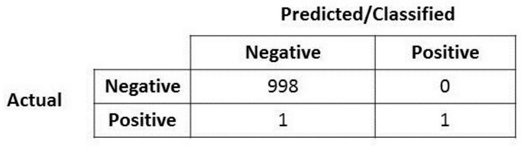
- Precision:

$$\text{Precision} = \frac{\text{True positive}}{\text{True positive} + \text{False positive}} = \frac{\text{True positive}}{\text{Total positive preditions}}$$

- Recall:

$$\text{Recall} = \frac{\text{True positive}}{\text{True positive} + \text{False negative}} = \frac{\text{True positive}}{\text{Total positive examples}}$$

Best practice - Accuracy, precision, recall



		Predicted/Classified	
		Negative	Positive
Actual	Negative	998	0
	Positive	1	1

- Accuracy: 0.999 (99.9 %)
- Precision: 1.0 (100 %)
- Recall: 0.5 (50 %)

Thank you for your attention

NTNU | Norwegian University of Science and Technology