Deep Equilibrium Nets for Solving Dynamic Stochastic Models

MCC, 03-04 September 2024

Lecture 2: Training the network

Aleksandra Friedl

https://github.com/alexmalova/DEQN_workshop

friedl@ifo.de

This course is inspired by Simon Scheidegger and based on his various teaching materials that I was relying on as a student and those which came as a result of our scientific collaboration

Outline of the presentation

Setting the learning rate

- Too small learning rate: slow convergence
- Too high learning rate: no convergence
- Optimal learning rate: good convergence

Learning rate can be also adaptive!

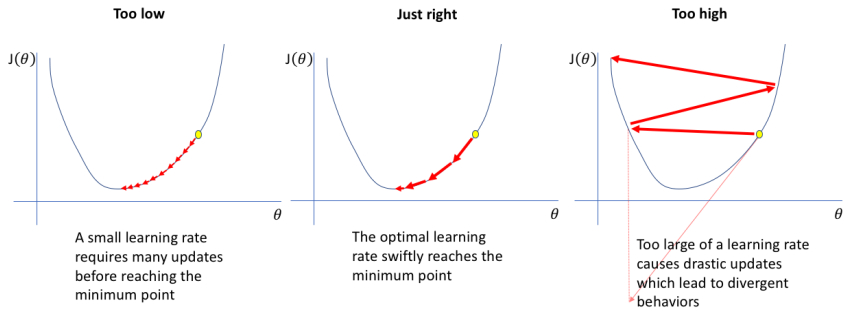


figure from https://www.jeremyjordan.me/content/images/2018/02/Screen-Shot-2018-02-24-at-11.47.09-AM.png

Choice of the stochastic gradient descent

Method	Formula
Learning Rate	$w^{(t+1)} = w^{(t)} - \eta \cdot \nabla \ell(w^{(t)}, z) \quad = w^{(t)} - \eta \cdot \nabla w^{(t)}$
Adaptive Learning Rate	$w^{(t+1)} = w^{(t)} - \eta_t \cdot \nabla w^{(t)}$
Momentum [Qian 1999]	$w^{(t+1)} = w^{(t)} + \mu \cdot (w^{(t)} - w^{(t-1)}) - \eta \cdot \nabla w^{(t)}$
Nesterov Momentum [Nesterov 1983]	$w^{(t+1)} = w^{(t)} + v_t; \quad v_{t+1} = \mu \cdot v_t - \eta \cdot \nabla \ell(w^{(t)} - \mu \cdot v_t, z)$
AdaGrad [Duchi et al. 2011]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A_{i,t}} + \varepsilon}; \quad A_{i,t} = \sum_{\tau=0}^{t} \left(\nabla w_i^{(t)}\right)^2$
RMSProp [Hinton 2012]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A'_{i,t}} + \varepsilon}; \quad A'_{i,t} = \beta \cdot A'_{t-1} + (1-\beta)\left(\nabla w_i^{(t)}\right)^2$
Adam [Kingma and Ba 2015]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot M_{i,t}^{(1)}}{\sqrt{M_{i,t}^{(2)}} + \varepsilon}; \quad M_{i,t}^{(m)} = \frac{\beta_m \cdot M_{i,t-1}^{(m)} + (1-\beta_m)\left(\nabla w_i^{(t)}\right)^m}{1-\beta_m^t}$

See https://arxiv.org/pdf/1609.04747.pdf

Outline of the presentation

Weights initialization

- Before the training process starts: all weights vectors must be initialized with some numbers.
- There are many initializers of which random initialization is one of the most widely known ones (e.g., with a normal distribution).
  - Specifically, one can configure the mean and the standard deviation, and once again seed the distribution to a specific (pseudo-)random number generator.
  - which distribution to use, then?
  - random initialization itself can become problematic under some conditions: you may then face the vanishing gradients and exploding gradients problems.
- What to do against these problems?
  - e.g. Xavier He initialization (available in Keras)
  - They are different in the way how they manipulate the drawn weights to arrive at approximately 1. By consequence, they are best used with different activation functions.
  - Specifically, He initialization is developed for ReLU based activating networks and by consequence is best used on those. For others, Xavier (or Glorot) initialization generally works best.

See, e.g., Glorot Bengio (2010) - http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf; He et al. (2015) - https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/ He_Delving_Deep_into_ICCV_2015_paper.pdf

Outline of the presentation

Vanishing gradients

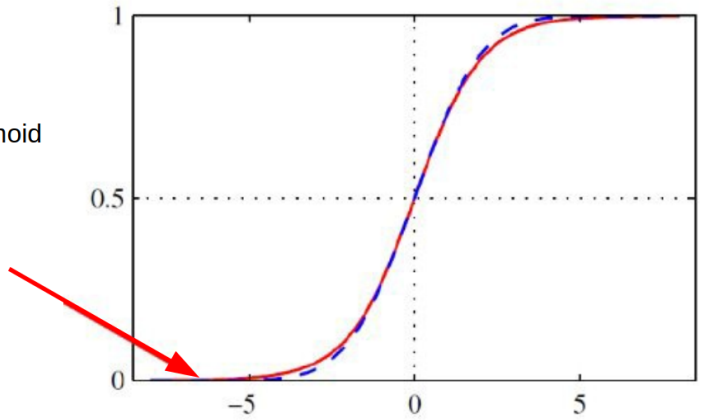
Deep learning community often deals with two types of problems during training: vanishing gradients (and exploding) gradients.

- Vanishing gradients
  - the backpropagation algorithm, which chains the gradients together when computing the error backwards, will find really small gradients towards the left side of the network (i.e., farthest from where error computation started).
  - This problem primarily occurs e.g. with the Sigmoid and Tanh activation functions, whose derivatives produce outputs of $0 < x' < 1$, except for Tanh which produces $x' = 1$ at $x = 0$.
  - Consequently, when using Tanh and Sigmoid, you risk having a suboptimal model that might possibly not converge due to vanishing gradients.
  - ReLU does not have this problem – its derivative is 0 when $x < 0$ and is 1 otherwise.
  - It is computationally faster. Computing this function – often by simply maximizing between $(0, x)$ – takes substantially fewer resources than computing e.g. the sigmoid and tanh functions. By consequence, ReLU is the de facto standard activation function in the deep learning community today.

Vanishing gradients

Problem with Sigmoid
→ Saturation

Gradient too small



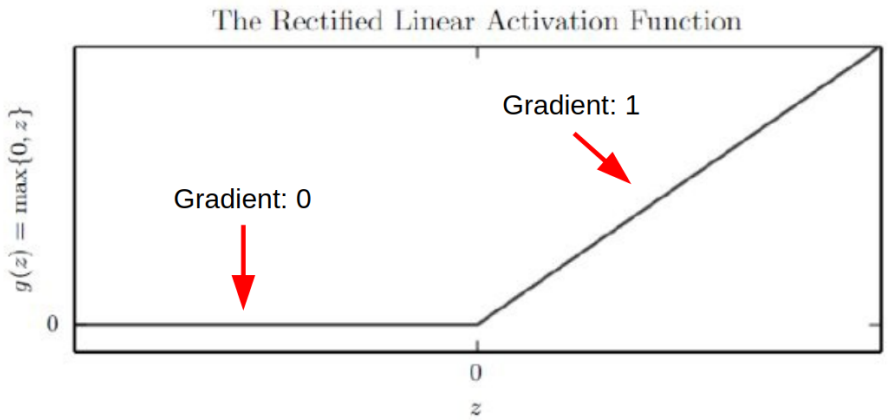
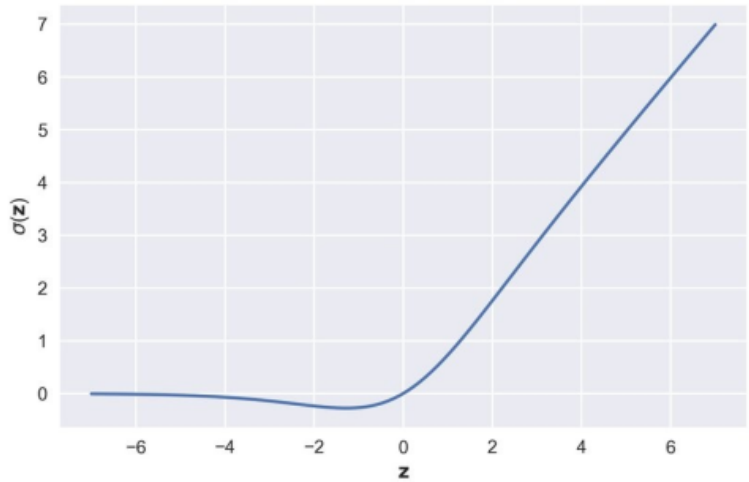


figure from Fig. From Goodfellow et al. (2016)

Vanishing gradients: swish activation function

- Nevertheless, it does not mean that it cannot be improved.
  - Swish activation function.
  - Instead, it does look like the de-facto standard activation function, with one difference: the domain around 0 differs from ReLU.
- Swish is a smooth function. That means that it does not abruptly change direction like ReLU does near x = 0.
  - Swish is non-monotonic. It thus does not remain stable or move in one direction, such as ReLU.
  - It is in fact this property which separates Swish from most other activation functions, which do share this monotonicity.
- In applications - Swish could be better than ReLU

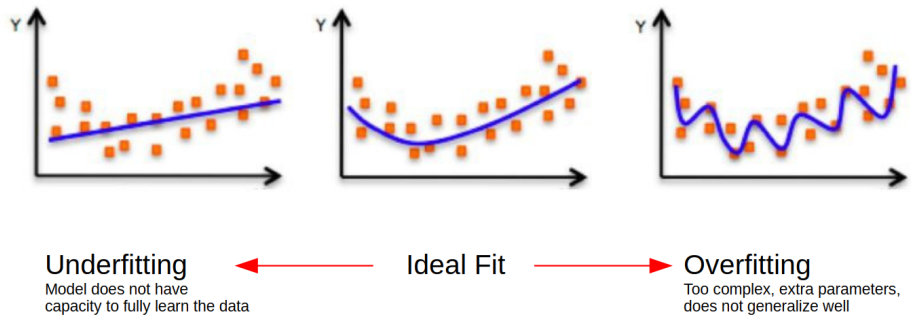
$$f(x) = x * \text{sigmoid}(x)$$
$$= x * \left(1 + e^{-x}\right)^{-1}$$



Outline of the presentation

Overfitting

- **Underfitting**: model does not have capacity to fully learn the data
- **Ideal fit**: the model is appropriate for the data
- **Overfitting**: the model is too complex, a lot of extra parameters and does not generalize well



Overfitting: early stop

Stop training before we have a chance to overfit

Outline of the presentation

Model quality assessment

- So far, we have assessed the prediction quality of our model based on the same data that we used for training.
- This is a very bad idea, since we can not accurately measure how well our model works for previously unseen data points (e.g., for cars not in our dataset)
- Our model may over-fit to the training data and loose its ability to make predictions.

Coming next

We'll see some best practices for evaluating machine learning models

Model evaluation: split the data

- To avoid over-fitting, it is good practice to assess the quality of a model based on test data that must not be used for training the model.
- The key idea is to split the available data (randomly) into training, validation, and test data.

Model evaluation: split the data

One common approach to reliably assess the quality of a machine learning model and avoid over-fitting is to randomly split the available data into:

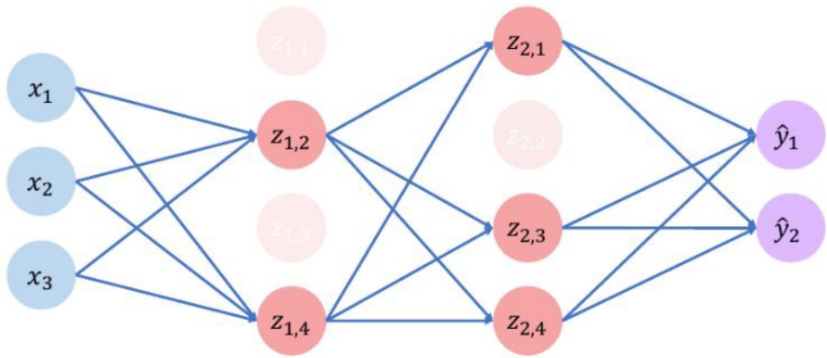
- training data (70% of the data) is used for determining optimal coefficients.
- validation data (20% of the data) is used for model selection (e.g., fixing degree of polynomial, selecting a subset of features, etc.)
- test data (10% of the data) is used to measure the quality that is reported

Regularization

- Regularization is a technique that constrains our optimization problem to discourage complex models
- We use it to improve generalization of our model on unseen data

Regularization: dropout

- During training, randomly set some activations to 0
- Typically 'drop' 50 % of activations in layer
- Forces network to not rely on any node



See Srivastava et al., 2014 – http://jmlr.org/papers/v15/srivastava14a.html

Regularization: batch normalization

- Speeds up the training process
- Decreases the importance of initial weights
- Regularizes the model

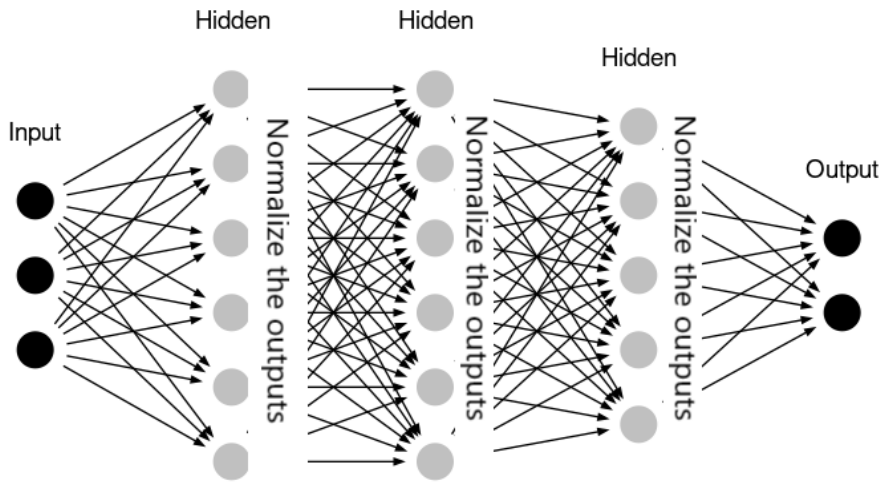


Fig. from and for the reference https://medium.com/@abheerchrome/batch-normalization-explained-1e78f7eb1e8a

Outline of the presentation

Inputs and outputs

Select inputs and outputs for your problem

- Before anything else, you need to think about the problem you are trying to solve, and make sure that you have data for the problem, both input vectors and target outputs.
- At this stage you need to choose what features are suitable for the problem and decide on the output encoding that you will use — standard neurons, or linear nodes.
- These things are often decided for you by the input features and targets that you have available to solve the problem.
- Later on in the learning it can also be useful to re-evaluate the choice by training networks with some input feature missing to see if it improves the results at all

Preparing the data

- Normalize inputs
  - Re-scale the data by subtracting the mean value from each element of the input vector, and divide by the variance (or alternatively, either the maximum or minus the minimum, whichever is greater).
- Split the data into training, testing, and validation sets
  - You cannot test the learning ability of the network on the same data that you trained it on, since it will generally fit that data very well (often too well, overfitting and modeling the noise in the data as well as the generating function).
  - Recall: we generally split the data into three sets, one for training, one for testing, and then a third set for validation, which is testing how well the network is learning during training.

Network architecture

Select a network architecture

- You already know how many input nodes there will be, and how many output neurons.
- You need to consider whether you will need a hidden layer at all, and if so how many neurons it should have in it.
- You might want to consider more than one hidden layer.
- The more complex the network, the more data it will need to be trained on, and the longer it will take.
- It might also be more subject to over-fitting.
- The usual method of selecting a network architecture is to try several with different numbers of hidden nodes and see which works best

Training and testing

- Train the network
  - The training of the NN consists of applying the MLP algorithm to the training data.
  - This is usually run in conjunction with early stopping, where after a few iterations of the algorithm through all of the training data, the generalization ability of the network is tested by using the validation set.
  - The NN is very likely to have far too many degrees of freedom for the problem, and so after some amount of learning it will stop modeling the generating function of the data, and start to fit the noise and inaccuracies inherent in the training data. At this stage the error on the validation set will start to increase, and learning should be stopped.
- Test the network
  - Once you have a trained network that you are happy with, it is time to use the test data for the first (and only) time. This will enable you to see how well the network performs on some data that it has not seen before, and will tell you whether this network is likely to be usable for other data, for which you do not have targets.

How to do it practically: Keras and Tensorflow

- Software
  - TensorFlow tensorflow.org
  - Keras API: https://www.tensorflow.org/guide/keras/sequential_model
  - TensorFlow is an open-sourced end-to-end platform, a library for multiple machine learning tasks, while Keras is a high-level neural network library that runs on top of TensorFlow
- Data
  - Fun data sets to play with: https://www.kaggle.com/datasets
  - Some “clean” data to play with: https://archive.ics.uci.edu/ml/index.php
- Debugging
  -
  - Help for debugging – Tensorboard: https://www.tensorflow.org/tensorboard

Intorduction to DNN

- Lets look at the notebook: 02_Gentle_DNN.ipynb.
- This Notebook contains all the basic functionality from a theoretical point of view.
- 2 simple examples, one regression (now), and one classification (for homework).

To do:

Select one of the functions below:

- **Polynomial Functions**:
  - $f(x, y, z) = x^2 + y^2 + z^2$
  - $g(x, y, z) = x^3 + xy + y^3 + z$
- **Trigonometric Functions**:
  - $f(x, y, z) = \sin(\pi x)\cos(\pi y)\sin(\pi z)$
  - $g(x, y, z) = \sin(\pi x) + \cos(\pi y) + \sin(\pi z)$
- **Exponential Functions**:
  - $f(x, y, z) = e^x e^y e^z = e^{x+y+z}$
  - $g(x, y, z) = e^{x^2-y^2+z^2}$
- **Mixed Functions**:
  - $f(x, y, z) = xe^y \sin(\pi z)$
  - $g(x, y, z) = \sin(\pi x)e^y \cos(\pi z)$

- Approximate a 3-dimensional function with Neural Nets based 10, 50, 100, 500 points randomly sampled from $[0, 1]^2$. Compute the average and maximum error.
- The errors should be computed by generating 1,000 uniformly distributed random test points from within the computational domain.
- Plot the maximum and average error as a function of the number of sample points.

To do:

Play with the architecture:

- Number of hidden layers.
- activation functions.
- choice of the stochastic gradient descent algorithm.
- Monitor the performance with respect to the architecture.

Questions?