Deep Learning
Lecture 5
Batch Normalization / Weight initialization / Optimizers

Prof. Dr. Rainer Herrler
Phone: 09721/ 940-8710
Email: rainer.herrler@fhws.de

Pictures from Wikipedia / Pixabay
Some Pictures generated with Stable Diffusion

Last Week

DL_006_FashionMnist.ipynb
DL_007_FashionMnistMultinomial.ipynb

Question:
What to do to make the model multinomial ?

What new did we see in DL_007 ? Conversion

- Encoding of the predictions was changed from numbers to one_hot-Encoding
  - Use to.one_hot(labels, num_classes)
  - or tf.keras.utils.to_categorical(labels, num_classes)
- Reverse function is:
  - _____
- Last Layer activation was switched from sigmoid to softmax
- Loss function was switched from “binary-crossentropy” to “categorical_crossentropy” (fitting to softmax)

What new did we see in DL_007 ? Evaluation & Analysis

- Tesorflow also provides functions for Precsion & Recall
  - sklearn.metrics.precision_score
  - sklearn.metrics.recall_score
- Accuracy, precision and recall are very close
  - Explanation: we have very balanced training data
- Confusion matrix helps us to see what classes are likely to be misclassified
  - Pandas: pd.crosstab(ytrue, ypred)
  - Scikit Learn: sklearn.metrics.confusion_matrix(ytrue, ypred)

Exercise Experiment Results

- Original Run
  - 5s per Epoch – Accuracy 82%
- Relu + categorical_hinge
  - 3s per Epoch – Accuracy 37% after 10 Epochs
  - 3s per Epoch – Accuracy 37% after 100 Epochs
- Relu in first layer + sgd
  - 3s per Epoch – Accuracy 9%
  - Explanation: Relu + Standard Weight initialization leads to very saturated
  - Batch normalization needs to be added
- Relu in first layer + BatchNorm 5s per epoch
  - Reaching 88% Accuracy at 10 Epochs
  - Reaching 95% Accuracy at 100 Epochs
- Relu in first layer + Adam without batch norm
  - Reaching 84% Accuracy at 10 Epochs

Epochs / Batches and Iteration

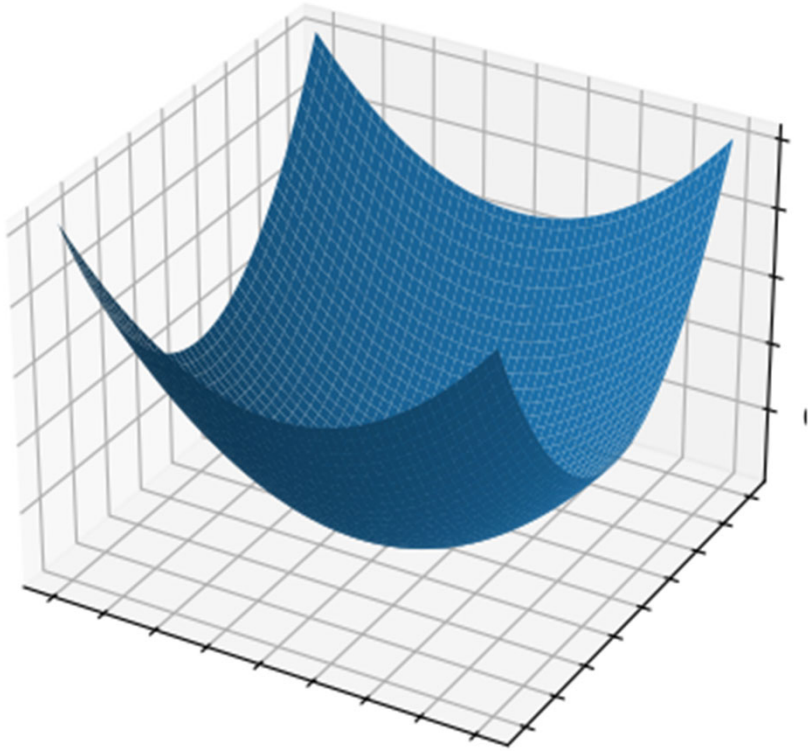
- **One Iteration** corresponds to one forward/backward pass involving one or more training samples
- A **Batch** is a set of training samples used in one iteration
  - The Batch can comprise all available training data, a part of it (mini-batch) or just a single sample
  - Larger batchsizes usually lead to a more stable learning process
  - Batches are randomly sampled from the training set
- An **Epoch** comprises several iterations until an entire dataset is passed through a Neural Network once.
  - Depending on the amount of data several epochs are required in order to “converge”

Example: 60000 Training Samples, with a batchsize of 1000 leads to 60 Iteations in one Epoch

Gradient descent variants

- **Gradient Descent aka [Full] Batch Gradient Descent**
  - Batchsize is N, where N is the number of all available training samples
  - No stochastic noise, direct path to the local minimum is possible (regarding the training set) if learningrate is selected properly
  - One iteration might take very long and data might not be possible to bet kept in the memory
- **Stochastic Gradient descent**
  - Batchsize is 1, we take one random training sample per update step
  - We make smaller updatesteps, but one iteration can be done much faster.
  - Updatesteps might go into the wrong direction (regarding the Loss for all training samples) but this will be corrected by subsequent updatesteps with other random sample
- **Minibatch Gradient descent**
  - Batchsize n is somewhere between 1 and N
  - Stochastic noise is reduced but not completely diminished
  - Is usually fastest because makes maximum usage of the GPU

Gradient Descent with different Batch Sizes



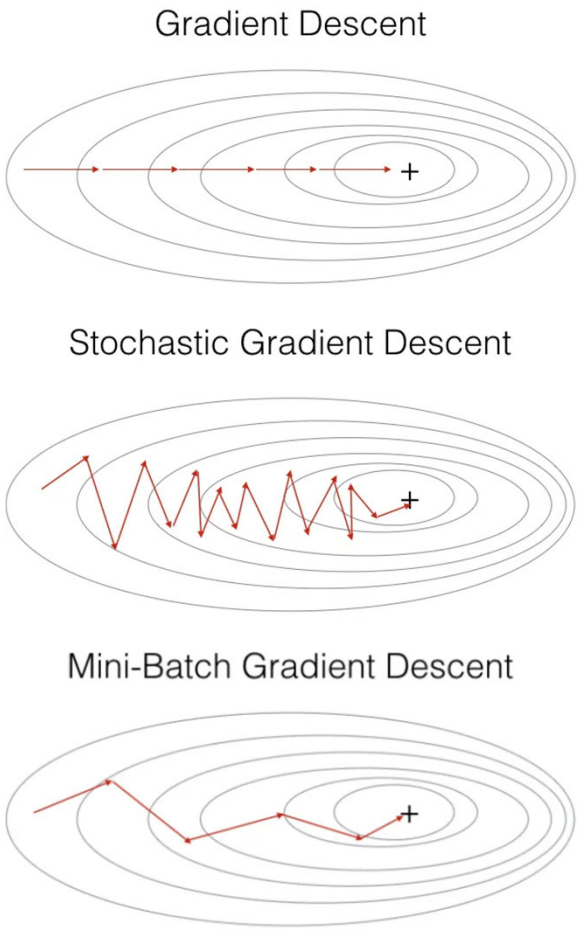
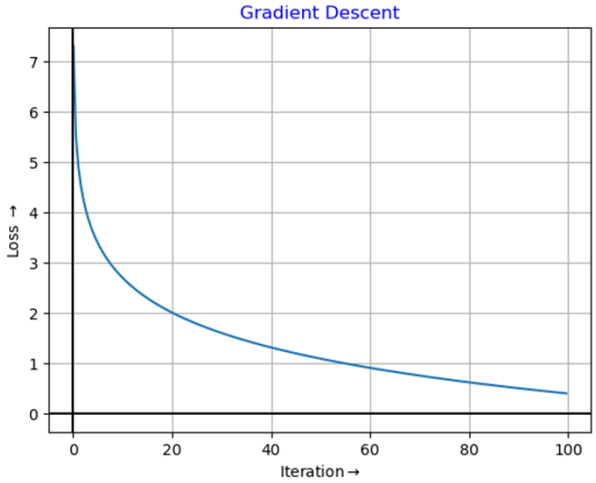
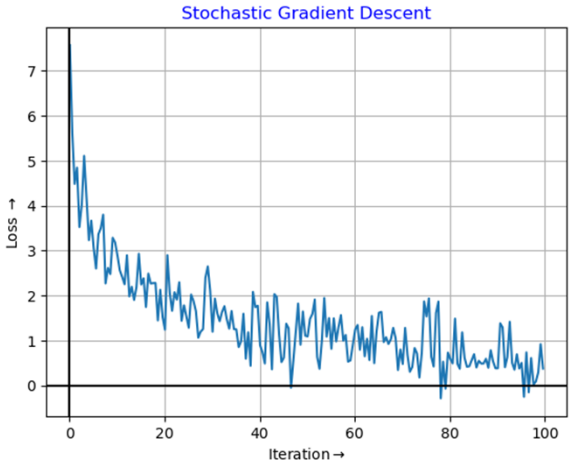
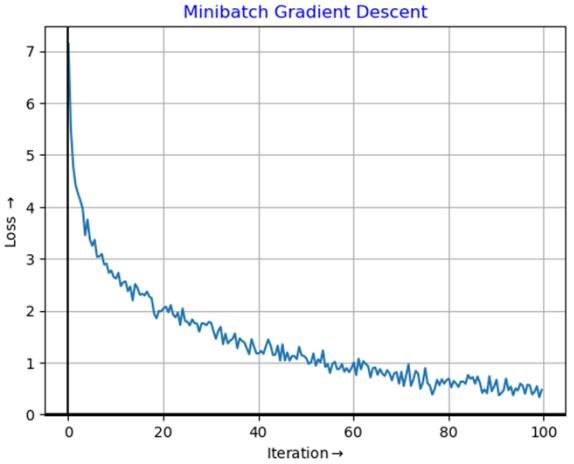


Illustration Source:
Andrew Ng's MOOC Deep Learning Course

Gradient Descent with different Batch Sizes







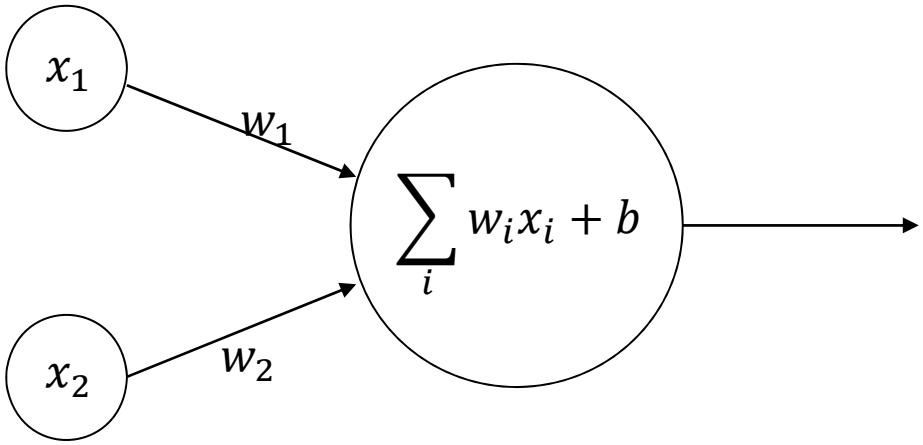
How to determine the optimal batchsize?

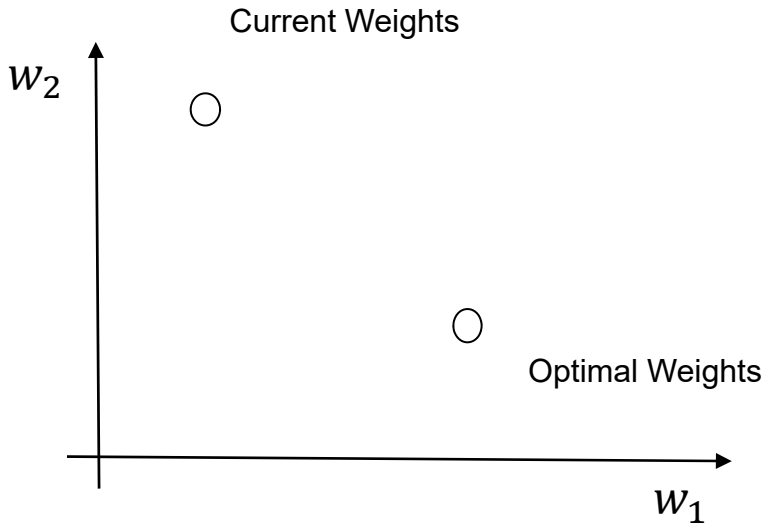
- Dependent on Size of Training Example (e.g 10 Features or 800x600*3 Pixels)
- Should fit into CPU Cache, GPU Memory
- Test different values regarding the training speed, use powers of 2 e.g. 64,128, … 1024, 2048…

How to choose batchsize and epoch-count

- Batch Size
  - Larger batch sizes result in faster progress in training, but don't always converge as fast.
  - Smaller batch sizes train slower, but *can* converge faster. It's definitely problem dependent (e.g. size of a training sample)
  - Its common to test powers of two as a batch size e.g. 23, 1024 or fractions of the number of training samples (e.g. 1000 og 60000)
  - You have to consider the graphics card memory as an upper limit
  - the preparation of batch (reading from disk, simulation) should not take longer as the network update step (second limit)
- Epochs
  - *In general*, the models improve with more epochs of training until convergence. This means they will start to plateau in accuracy.
  - Try something like 50 and plot number of epochs (x axis) vs. accuracy (y axis). If you still see significant progress in the last epochs you can raise the number, or continue training.
  - Question: Can we do too much epochs ?

Intuition about why its good to have zero mean at inputs





How can we normalized Inputs ?

- 2022 we learned in ML class
  - We can fit a StandardScaler or a MinMaxScaler on the whole training data
  - We need to apply this scaler to test data before prediction
- But what about inner layers ?

Batch Normalization Layers

- Invented in 2015 by Ioffe and Szegedy
- Batch Normalization normalizes input to the next layer
- It calculates for each feature the mean and standard deviation considering all samples of the batch
- Inputs are normalized and zero-meaned before they are passed to the next layer
- This operation is differentiable (important for backprop)

Batch Normalization Algorithm

Input: m training samples of a mini Batch $\mathcal{B} = \{x_1, \ldots x_m\}$

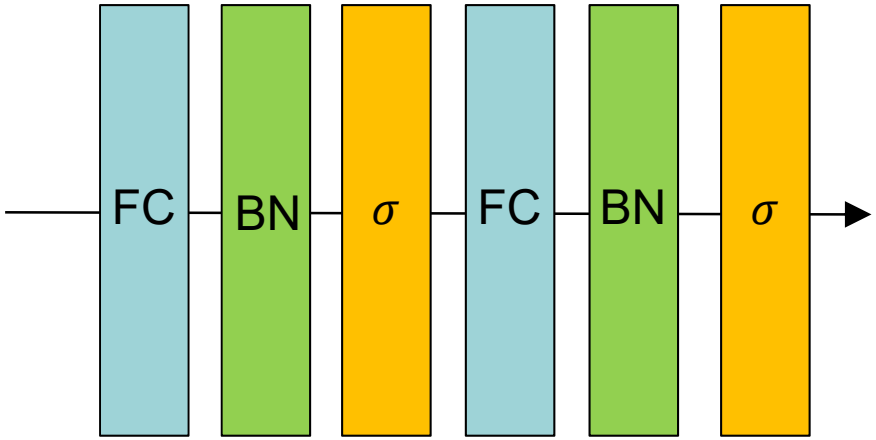
- $\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$
- $var_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$
- $\widehat{x_i} = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{var_{\mathcal{B}} + \epsilon}}$
- $\widehat{y_i} = \gamma \widehat{x_i} + \beta$

ϵ is a small number preventing to divide by zero

β and γ are trained parameters to scale and shift the normalized range to an optimal value

At test time the last mean and variance stay fixed!

Where to put batch normalization layres



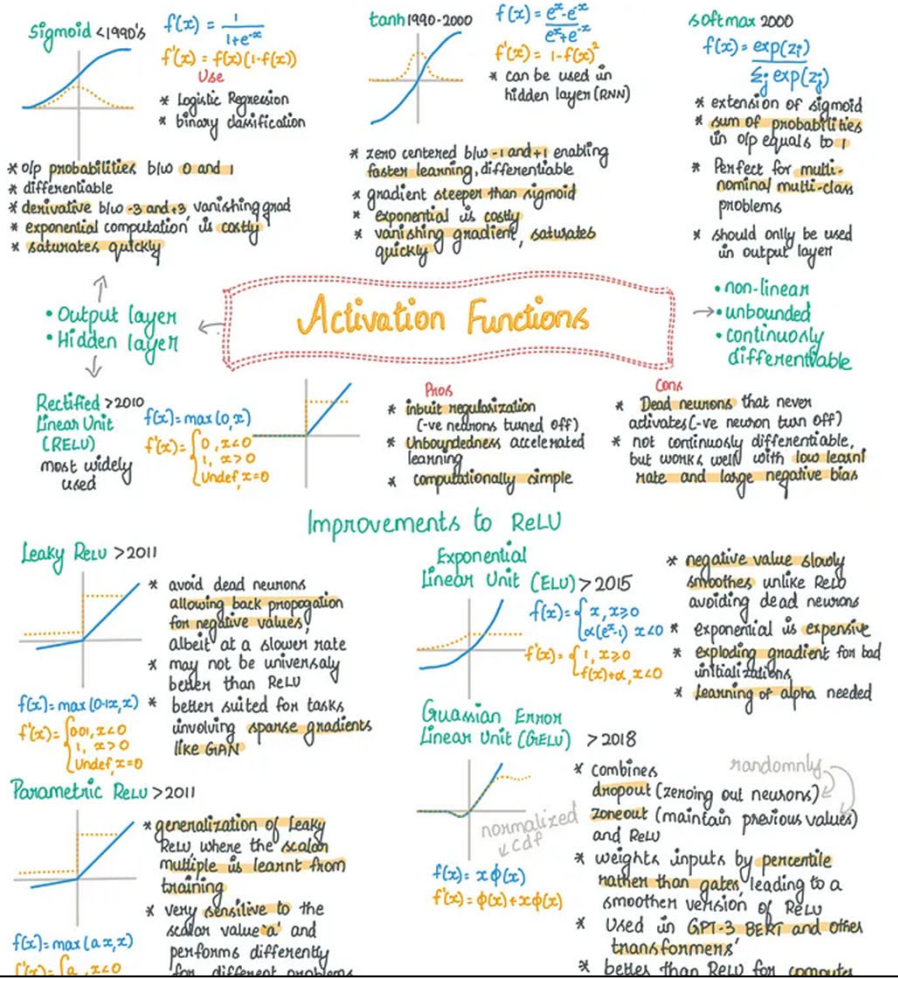

```
model = Sequential()
model.add(Dense(100,input_dim=20))
model.add(BatchNormalization())
model.add(Activation('sigmoid'))
model.add(Dense(100,input_dim=20))
model.add(BatchNormalization())
model.add(Activation('sigmoid'))
```

- Typically Batch Norm Layers are inserted between Fully Connected Layers and Activation Functions
- It also works well after the activation

Effects of Batch Normalization

- Positive
  - Improves the gradient flow through the network
  - Allowes higher Learning rates
  - Reduces the dependence on proper initialization
  - Similar effects like regularization (reduces chance to overfit, improves generalization)
- Negative
  - Does not work well with small batchsizes

Activation Functions Revisited

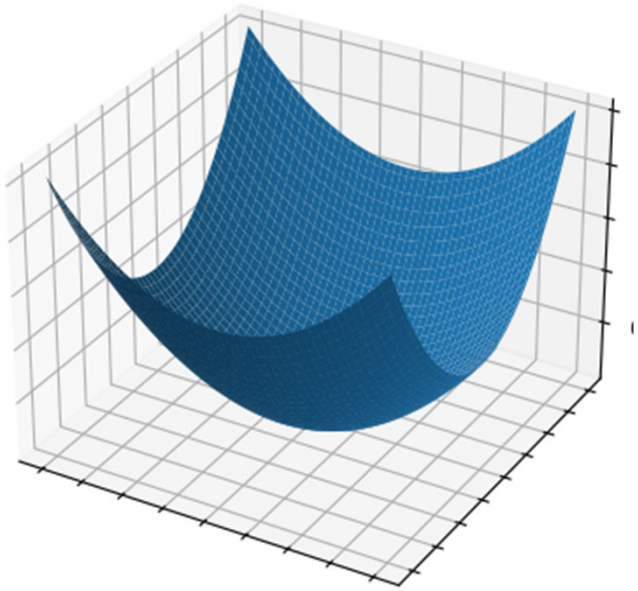


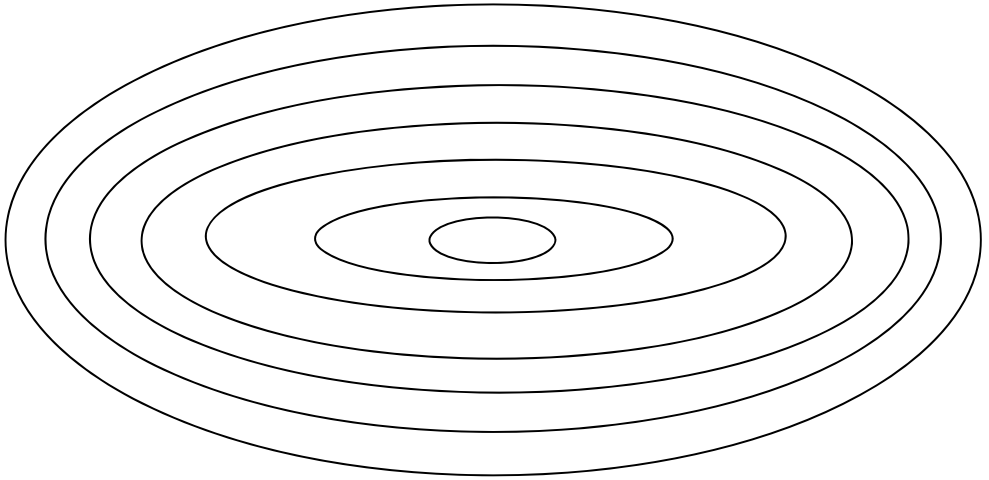
Recommended Article
https://towardsdatascience.com/fantastic-activation-functions-and-when-to-use-them-481fe2bb2bde

Weight initialization

- Initializing Weights with Constant Values
  - Not a good idea
  - All Nodes of a layer will learn the same since there is no variation
- Initialization with (small) random numbers
  - Will work just for few layers and few nodes
  - After some layers all neurons will be active and gradients will not work properly (vanishing/exploding gradients)
- **Xavier/Glorot** initialization
  - Works well for sigmoid/tanh activation function
  - $\text{np.random.randn}(outputs, innputs) * \sqrt{\frac{1}{inputs}}$
- **He** Initialization
  - Works well for ReLU activation function
  - $\text{np.random.randn}(outputs, innputs) * \sqrt{\frac{2}{inputs}}$

Optimizer - Intuition





Three most important enhancements of gradient descent:

- Momentum
- RMS-Prop
- Adam

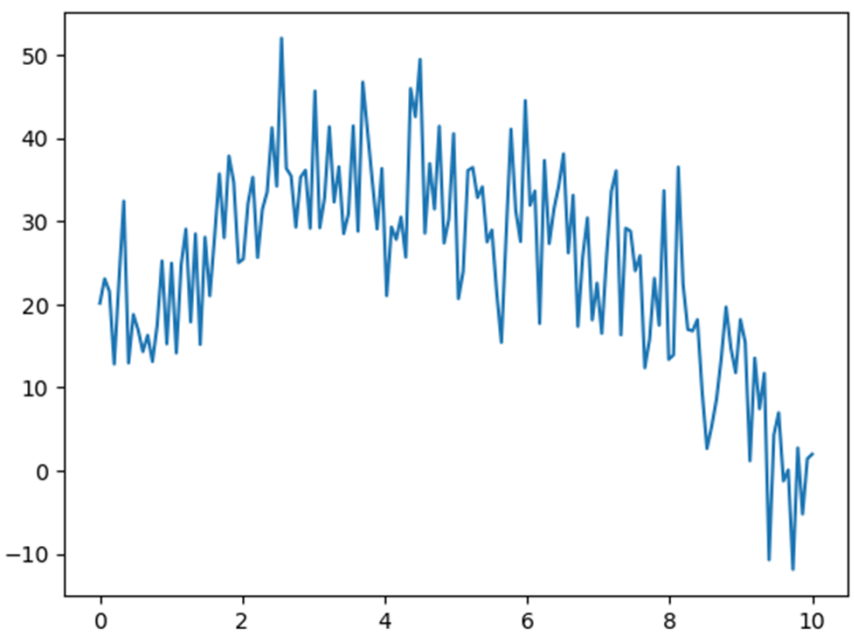
Optimizer

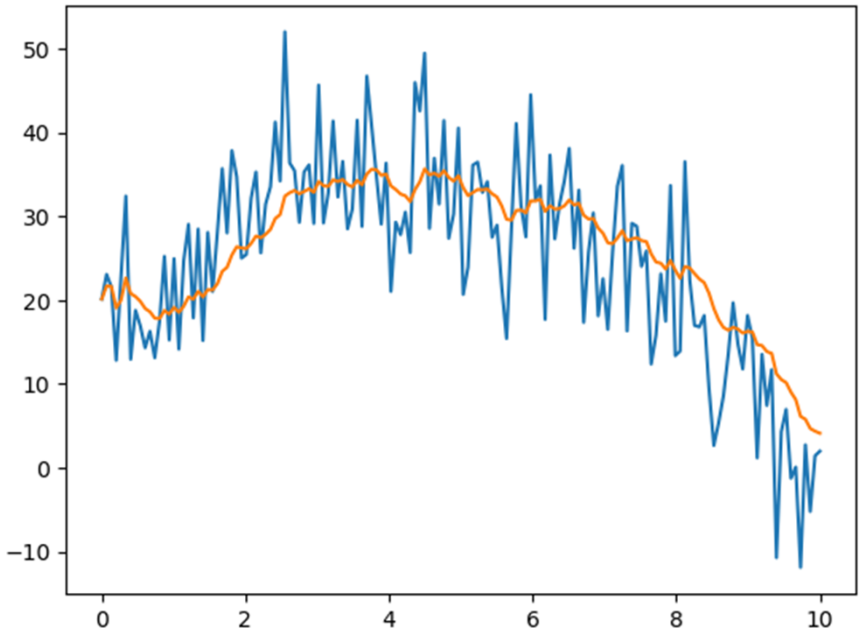
- What are optimizers ?
  - Optimizers are a enhanced strategy for updating weights
- Why optimizers ?
  - Pure gradient descent is sometimes ineffective and it can take a long time to converge
  - Pure gradient descent may overshoot the local minimia
- Idea
  - Don’t update just based on the current snapshot (gradient and learningrate) but also on the history of updates

Optimizer – introducing “Momentum”

- Don’t update just based on the current snapshot (gradient and learning rate) but also on the history of updates
- Remember gradient of the past to
  - Preserve movement in the same direction
  - Prevent movement in other directions
  - Damp oscillations
- How can this be done mathematically?

Realizing Momentum

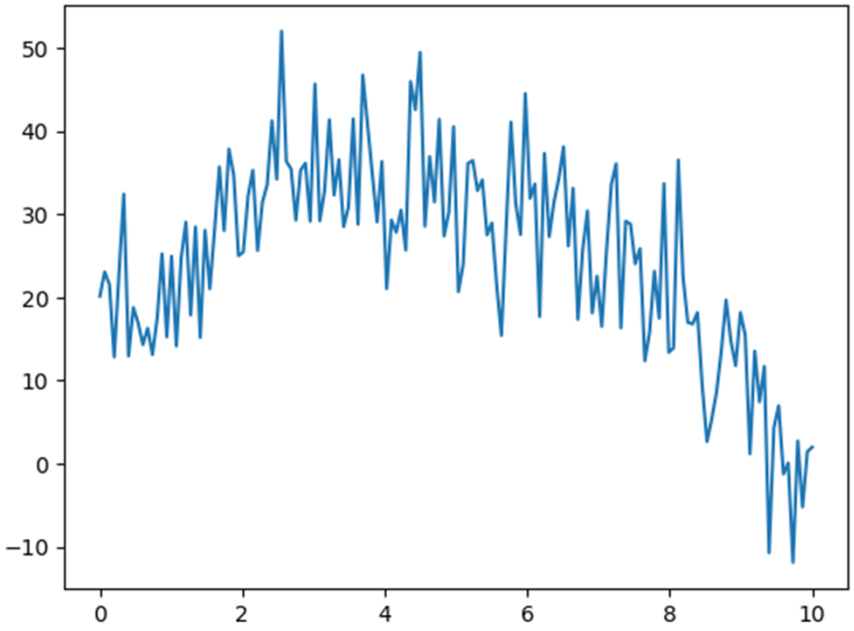


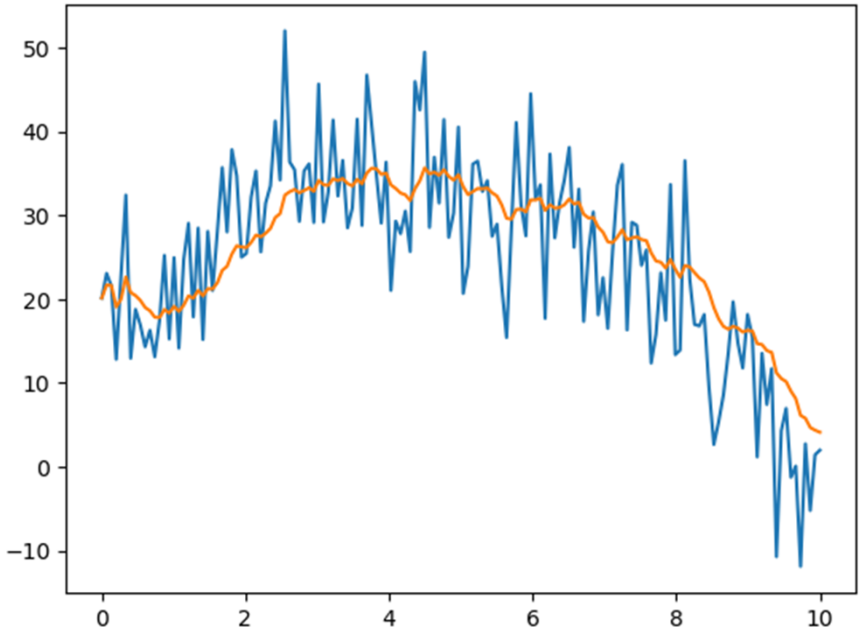


Idea:

- We calculate a “moving average” with a sliding window of relevant values
- This cuts of peaks
- Next step to I weight the newest point more than older points
- → EWMA is short for “Exponentially Weighted Moving Average”

Realizing Momentum with EWMA





Formula for a smoothed curve:

$$av(t) = \beta \cdot av(t-1) + (1-\beta) \cdot f(t)$$

Applied to weight updates:

$$\theta_i = \theta_i - \alpha \frac{\partial}{\partial \theta_i} J(\theta_i)$$

How to use Momentum in Keras

```
from keras.optimizers import SGD

model = Sequential()
sgd = SGD(lr=0.0001, momentum=0.8) # defaults are lr=0.01 momentum=0.0
model.add(Dense(100, activation="relu", input_shape=(784,)))
model.add(Dense(len(class_names), activation="softmax"))
model.compile(optimizer=sgd, loss="categorical_crossentropy“)
print(model.optimizer)
```

Optimizer – RMS Prop

- Different approach to restrict oscillations

$$E[g^2](t) = \beta E[g^2](t-1) + (1-\beta)\left(\frac{\partial}{\partial \theta_i} J(\theta_i)\right)^2$$

$$\theta_i(t) = \theta_i(t) - \frac{\alpha}{\sqrt{E[g^2]}} \frac{\partial}{\partial \theta_i} J(\theta_i)$$

- “Adaptive learning rate” taking into account how much a parameter was already changed in the past
- Exponential Weighted moving Average is also used to dacay older gradients

How to use RMSProp in Keras

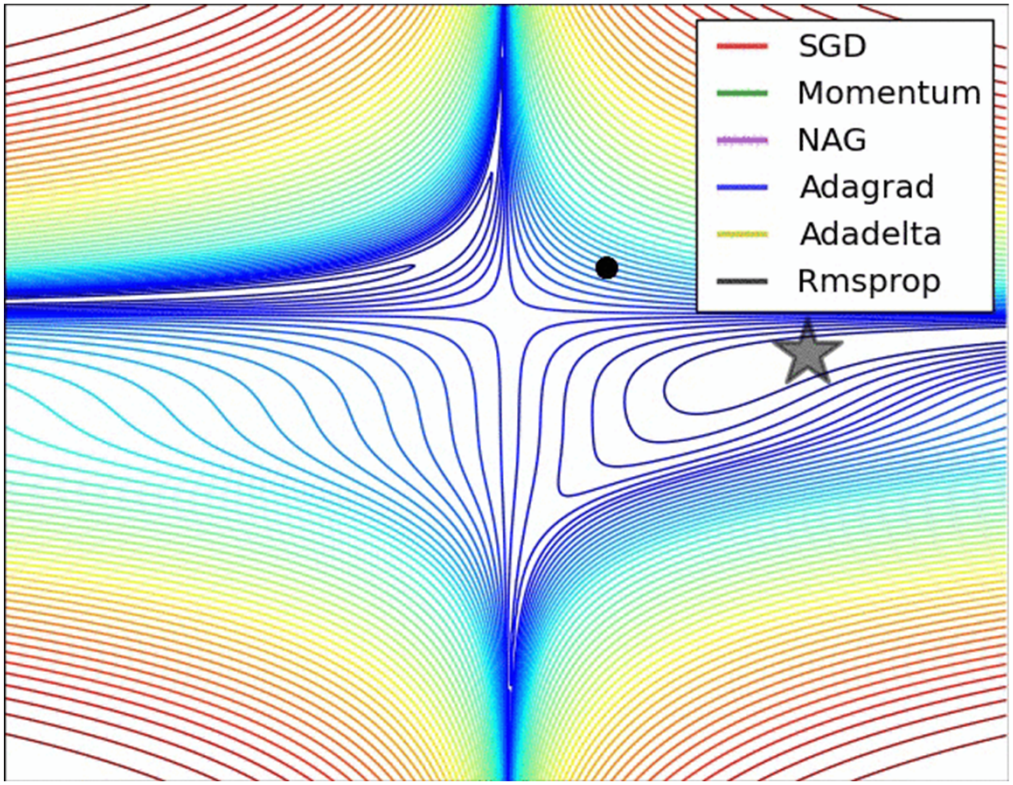
```
from tf.keras.optimizers.experimental import *

model = Sequential()
#opt = SGD(learning_rate=0.0001)
opt = RMSprop(learning_rate=0.0001)
model.add(Dense(100, activation="relu", input_shape=(784,)))
model.add(Dense(len(class_names), activation="softmax"))
model.compile(optimizer=opt, loss="categorical_crossentropy")
```

Optimizer Adam

- Invented in 2014
- Combines Ideas of Momentum and RMSProp (Exact formula not presented here)
- Slightly better than RMSProp
- Works best on a great variety of problems
- Finding the proper learning rate is not so difficult any more

Visualization of different optimizers

https://raw.githubusercontent.com/jeffheaton/t81_558_deep_learning/master/images/contours_evaluation_optimizers.gif



Summary

- Tensorflow
  - First models realized (MNIST / MNIST Multinomial)
- Iterations / Batches / Epochs
- Gradient Descent with different Batchsizes
- Normalization
  - Training data Normalization
  - Batch Normalization Layers
- Activation Functions and Weight Initialization
- Optimizers
  - SGD
  - Momentum
  - RMSprop
  - Adam