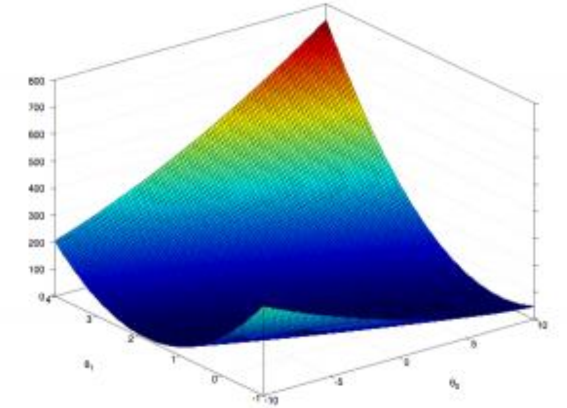
Training and Improving Neural Networks

How to train your neural network... so that it doesn't explode



Yordan Darakchiev

Technical Trainer

iordan93@gmail.com

sli.do
#DeepLearning

Table of Contents

Bias and Variance

Machine learning practices using big(ger) data

Regularization

- Usual L1 and L2 rules apply

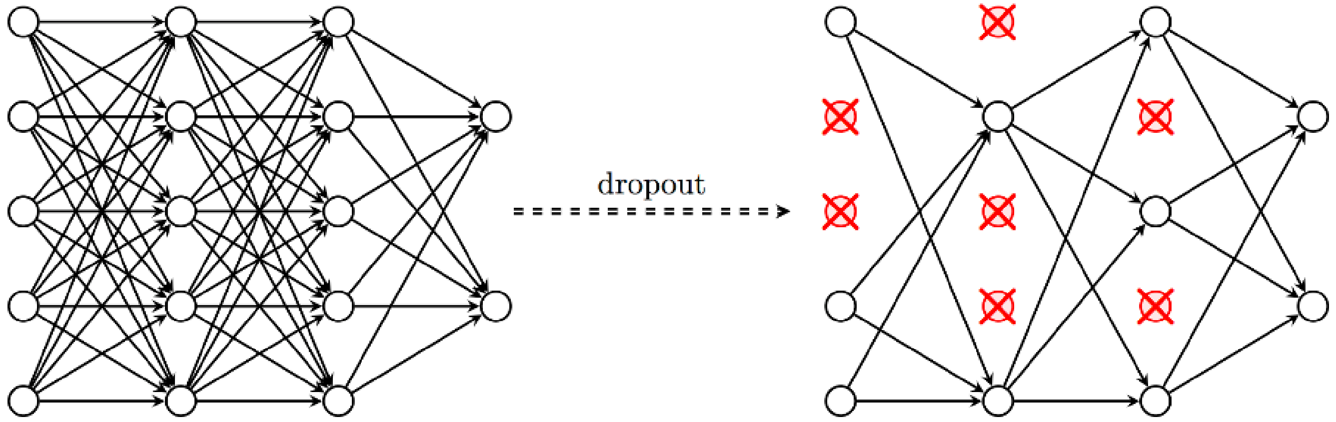
```
from tensorflow.layers import Dense
from tensorflow.keras import regularizers

Dense(
    kernel_regularizer = regularizers.L1L2(l1 = 0.5, l2 = 1),
    bias_regularizer = regularizers.L1L2(l1 = 0.5, l2 = 1),
    activity_regularizer = regularizers.L1L2(l1 = 0.3, l2 = 10))
```

  - Regularization is applied to the loss function
    - It tries to "remove" or shrink the parameters
  - We can regularize weights, biases and outputs
    - Usual steps: same regularization for weights and biases, none for outputs
- Note: using ReLU may result in activations = 0
  - This produces "dead neurons"
    - May be used as a form of regularization

Dropout

- Select a layer l
- At each training step, set a random fraction p of input weights of layer l to 0 $\Rightarrow$ keep $1 - p$ units
  - To keep the dimensions, scale the remaining weights by $\frac{1}{1-p}$




```
from tensorflow.layers import Dropout
Dropout(0.1)
```

- **Don't apply dropout during inference!**
  - `tensorflow` takes care of this

Selecting and Splitting Data

- Usually, we split the dataset like this
  - Training set – 70%
    - "Real training" set – 63%, validation set – 7%; 10 times
  - Testing set – 30%
- With many samples, this is unnecessary
  - And time consuming
- Law of big numbers
  - We can get stable results with many samples
  - ⇒ we have less chance of variance due to a small sample size
- Usual splitting for big data (e.g.,1M samples)
  - 980 000 / 10 000 / 10 000 samples
  - Alternatively, a bigger validation set: 980 000 / 16 000 / 4 000

Bias-Variance Error Analysis

- Bayes optimal error: the "real" error in data
  - No way to calculate, we need to try to come up with a measure
    - Naïve: this is 0%, the dataset is perfect
- Example: two-class classification (cats vs. dogs)
  - Metric: misclassification error ($E = 1 - A$)
  - Humans can achieve 0,5% error

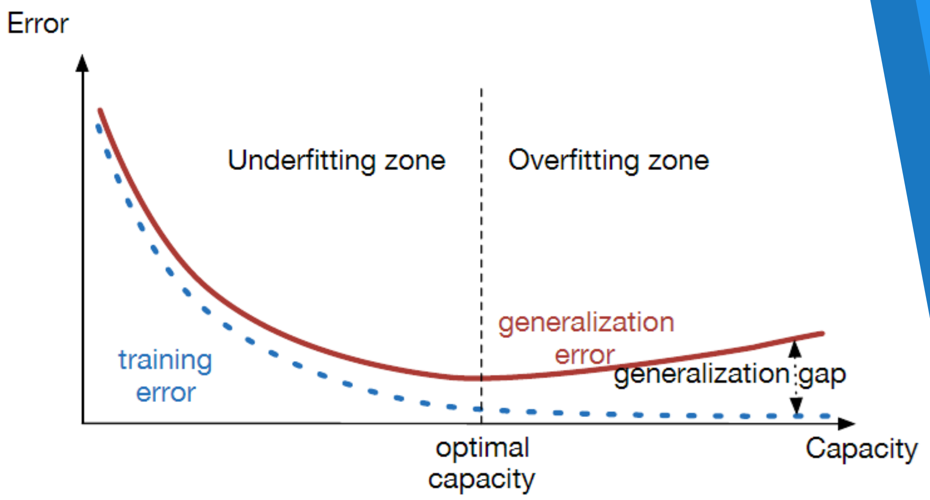
Algorithm	Train set error	Validation set error	Bias, %	Variance, %	Verdict
A1	1%	11%	0,5%	10%	High variance
A2	15%	16%	14,5%	1%	High bias
A3	15%	30%	14,5%	15%	Both
A4	0,5%	1%	0,5%	0,5%	"Neither"
A5	**0,3%**	**0,4%**	**?**	**?**	**?**

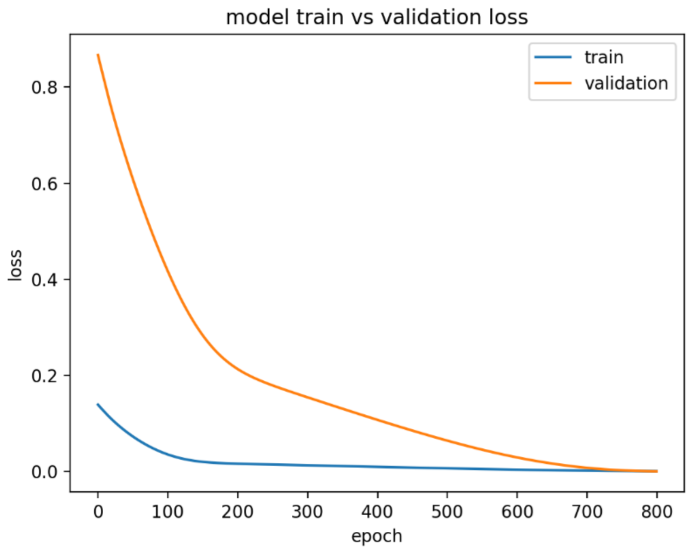
Taking the Next Step

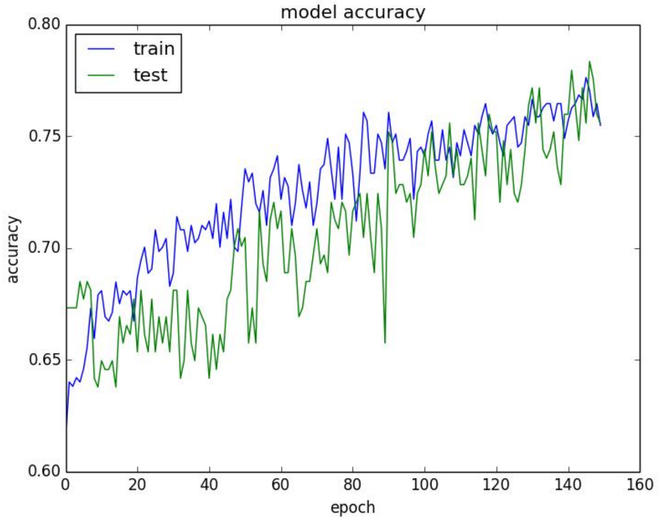
- There are no set rules, only things we can try
- High bias
  - Train a bigger network
  - Possibly, try out different architectures
    - Try to find one which is best suited for the task
  - Train longer (e.g., more epochs)
- High variance
  - Apply regularization
  - Try a smaller network architecture
  - Get more data
    - Or try to augment the current dataset
      - E.g. bootstrap sampling, image rotation, adding noise, etc.

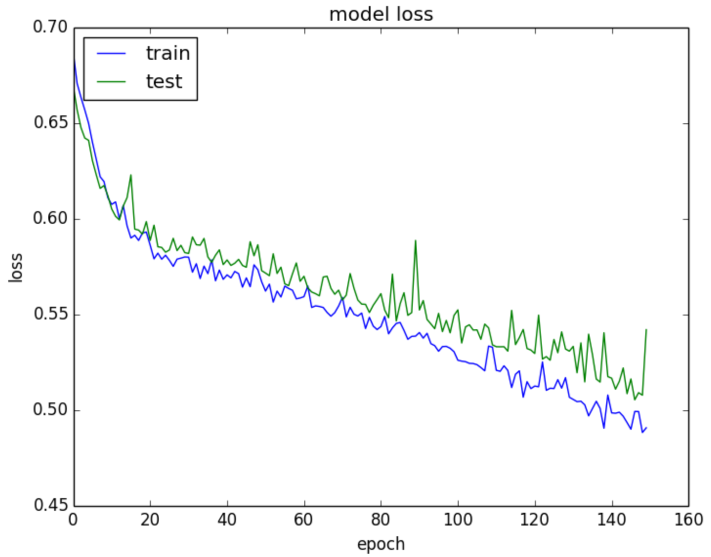
Training / Validation Curves

- The same as what we already know
  - Plot a metric (e.g., loss, accuracy…) w.r.t. the dataset size or epoch
- The shape and relative position of both curves help diagnose under- / overfitting









Optimization

"Learn smarter, not harder"

Weight Initialization

- Vanishing / Exploding gradients problem
  - Deeper networks can learn very complex functions
    - $\Rightarrow$ more layers = better
  - But let's look at what a computation looks like
  - Take, for example the activation at the 15th layer
    - Ignoring the activation functions for simplicity
    - $a^{[15]} \approx w^{[14]}a^{[14]} \approx w^{[14]}w^{[13]}a^{[13]} \approx \cdots \approx w^{[0]}w^{[1]} \ldots w^{[14]}x$
  - If the weights are similarly scaled, the product becomes $\approx w^{15}$
    - If some elements of w are $\gtrsim \mathbf{1}$, the product will become **really big**
    - Alternatively, if some elements are $\lesssim \mathbf{1}$, the product will become **really small**
  - This leads to problems when updating weights: $w = w - \nabla \mathrm{w}$
    - The gradients either become $\approx \infty$, or ≈ 0
- Solution: initialize the weights properly

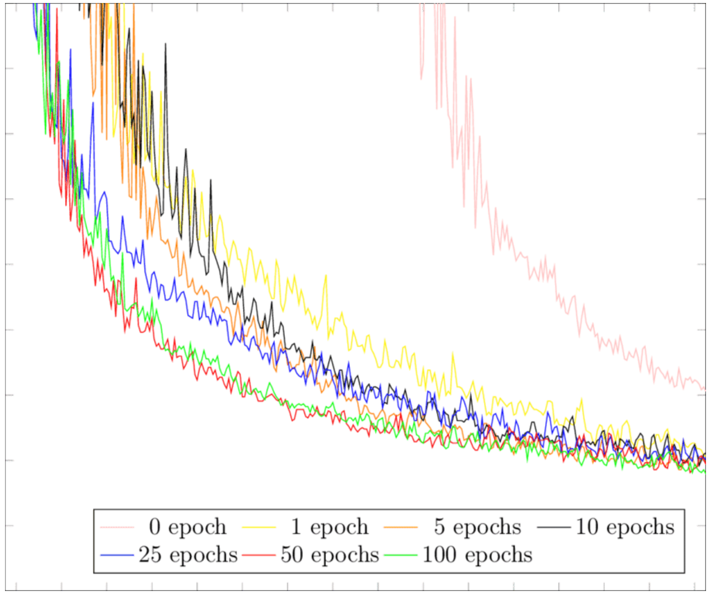
Weight Initialization (2)

- First, we know that we need random initialization
  - Gaussian, $\mu = 0, \sigma = 1$
    - $\mu = 0$ is needed because any bias has already been accounted for
- Also, initialize the weights with small numbers
  - The exploding / vanishing gradient problem affects only the first stages of training
    - After that, the NN should learn proper weights
- Glorot (Xavier) initialization
  - $\text{init} \sim N(0, \sigma)$ where $\sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}$
    - where n_{in} and n_{out} are the numbers of input and output units of the layer

```
Dense(
    kernel_initializer = tf.glorot_normal_initializer(), # or None
    bias_initializer = tf.zeros_initializer())
```

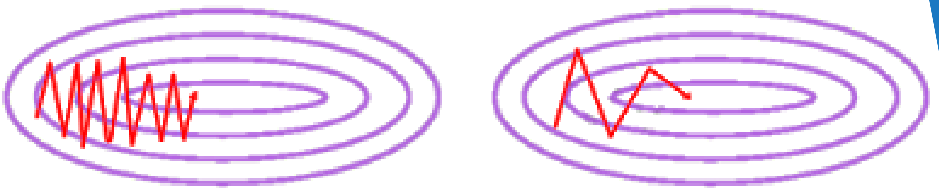
Mini-batch Gradient Descent

- It takes a lot of time to pass through the entire dataset to perform only 1 step of GD (batch gradient descent)
  - Solution: take a random sample (mini-batch) each time: mini-batch gradient descent
  - If the mini-batch contains one sample ⇒ stochastic GD (SGD)
- The cost function will not decrease smoothly
  - But will tend to decrease, also the training will be faster
- Choosing a mini-batch size (n_b)
  - Powers of 2 lead to better speed
    - E.g., **32**, **64**, 128
  - Implementation
    - Shuffle the training set
    - At each training step, pass n_b examples



Improving Gradient Descent

- Momentum
  - When updating weights,
    a fraction β_1 of the previous vector
    is added to the current:



$$v_t = \beta_1 v_{t-1} + (1 - \beta_1)\nabla J$$
$$w_t = w_{t-1} - \alpha v_t$$

  - This tends to average out the steps in the "wrong" direction and speed up convergence
- RMSprop
  - Similar to momentum, but second-order

$$S_t = \beta_2 S_{t-1} + (1 - \beta_2)(\nabla J)^2$$
$$w_t = w_{t-1} - \alpha \nabla J / (\sqrt{S_t} + \varepsilon)$$

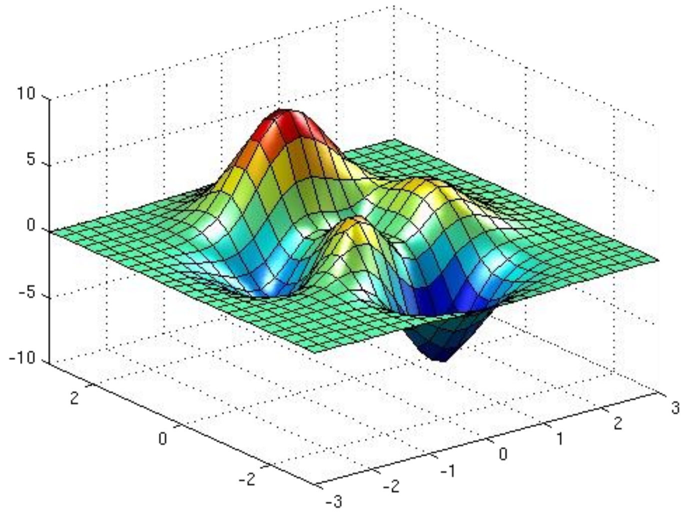
Adam Optimizer

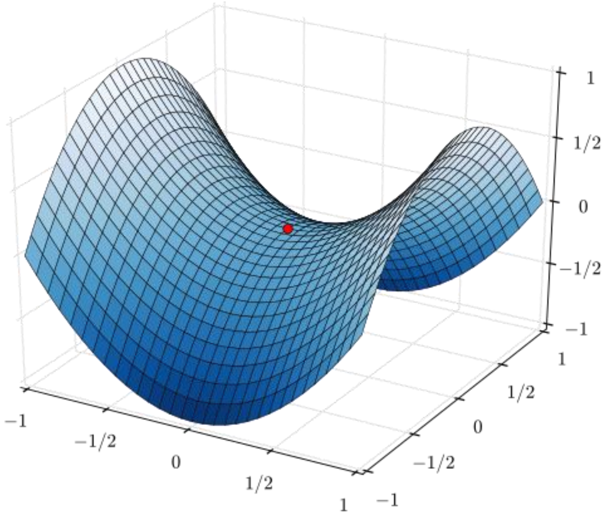
- Adam (**Ada**ptive **M**oment Estimation)
  - Combines momentum and RMSprop
  - Usually: α (tuning); $\beta_1 = 0{,}9$; $\beta_2 = 0{,}999$; $\varepsilon = 10^{-6} - 10^{-8}$
- Usage
  - In place of `GradientDescentOptimizer`
  - It's best to tune all hyperparameters but the default ones should work for most cases
    - Tuning α is non-negotiable

```
tf.train.AdamOptimizer(
    learning_rate = 0.001,
    beta1 = 0.9,
    beta2 = 0.999,
    epsilon = 1e-8)
```

A Note on Local Minima

- When training a model, GD and similar algorithms may get stuck in a local minimum
  - ML solution: different starting points
- In higher-dimensional spaces, most points with zero gradient are not local minima
  - They're instead saddle points
    - "Min" at one direction, "max" at the other
  - Example: 100 dimensions
    - Local min: all dimensions must be min
      - E.g., $p(\text{local min}) \approx 2^{-100} \approx 7{,}89.10^{-31}$
  - When an optimizer gets to a saddle point, it's able to "roll off"





Hyperparameter Tuning

Similar to "standard" machine learning

Prioritizing Hyperparameters

- Most important: learning rate α
- Momentum term β_1, mini-batch size n_b
- Number of hidden units
- Number of hidden layers
- Search methodology
  - Grid search doesn't work (too large search space)
  - Use random search instead
- Better idea: use a coarse random search first
  - When you find a good place, "zoom in" to that
  - Repeat

Hyperparameter Scales

- Usual: uniform scale
  - E.g., hidden layers = $\{2, 3, 4\}$, hidden units ∈ $[50; 100]$
- Logarithmic scale
  - E.g., $\alpha \in [0{,}00001; 10]$
  - If we pick uniformly, most values will be close to 1
  - Solution: use a log scale for better search space exploration
  - $\alpha = 10^k, k \in [-5; 1]$
- Exponentially weighted averages (β_1, β_2)
  - E.g., $\beta \in [0{,}9; 0{,}9999]$
  $\Rightarrow 1 - \beta \in [0{,}1; 0{,}0001]$
  $\Rightarrow 1 - \beta = 10^k, k \in [-4; -1]$
  - $\beta = 1 - 10^k, k \in [-4; -1]$

Batch Normalization

- Normalizing inputs: Z-score
- $x = \frac{x-\mu}{\sigma}, \mu = \frac{1}{n}\sum_{i=1}^{n} x_i, \sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n} x_i^2$
- Batch normalization
  - At a given layer l, $z_n = \frac{z-\mu}{\sqrt{\sigma^2+\varepsilon}}$
  - Use a linear transformation $\tilde{z} = \gamma z_n + \beta$ instead of the z
    - γ and β are parameters
    - γ and β are updated along with the weights w
  - Application: compute **before** activation function
  - Why does it work?
    - Doesn't allow the values to vary too much
  - Implementation

```
from tensorflow.keras.layers import BatchNormalization
BatchNormalization(input)
```

Summary

- Regularization
- Bias and variance
  - Error analysis
- Optimization algorithms
- Hyperparameter tuning
- Normalization

Questions?