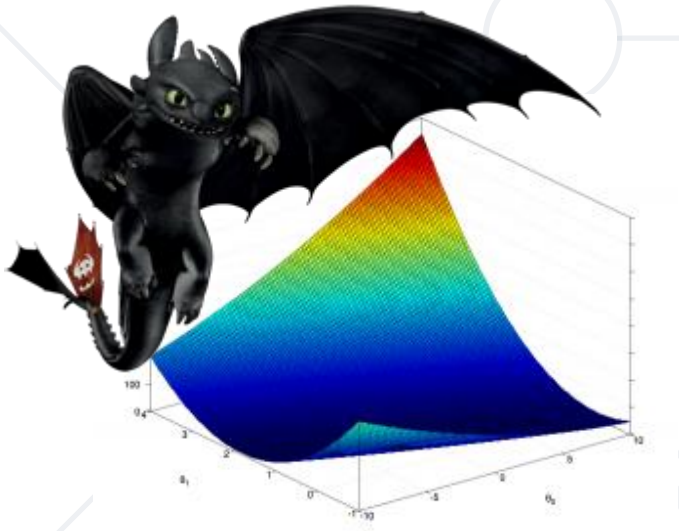
# Training and Improving Neural Networks

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




**Yordan Darakchiev**

**Technical Trainer**

Software University


SoftUni


**Software University**

https://softuni.bg

# Have a Question?

Software University

sli.do

#DeepLearning

# Table of Contents



# Building Models

"Complexity is not supposed to be complex”

# Building Models

Software University

- All base classes may be inherited
  - Just like estimators in `sklearn` (without the mixin complexity)

```
class MyLinear(Layer):
    def __init__(self, units = 32, input_dim = 32):
        super().__init__()
        self.w = self.add_weight(
            shape = (input_dim, units),
            initializer = "random_normal")
        self.b = self.add_weight(
            shape = (units,),
            initializer = "zeros")

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b
```

```
class MyLinear(nn.Module):
    def __init__(self, in_units, units):
        super().__init__()
        self.weight = nn.Parameter(
                torch.randn(in_units, units))
        self.bias = nn.Parameter(
                torch.randn(units,))


def forward(self, X):
        return torch.matmul(
            X, self.weight.data) + self.bias.data
```

# tensorflow's Functional API

Software University

- Models and layers are callable
  - We may specify multiple inputs and outputs

```
inputs = Input(shape = (784,))
layer = Dense(64, activation = "relu")
x = layer(inputs)
# Or, more concisely:
x = Dense(64, activation = "relu")(inputs)
```

  - Reusing a variable many times

```
x1 = Dense(64, activation = "relu")(inputs)
x2 = Dense(128, activation = "relu")(inputs)
```

  - Combining variables

```
result = keras.layers.Concatenate()([x1, x2])
```

# Multiple Inputs and Outputs in pytorch

Software University

- The same approach applies
  - Since we already have all variables assigned, just use them as needed in `forward()`

```
class AdvancedModel(nn.Module):
    def __init__(self):
        super(AdvancedModel, self).__init__()
        self.layer1 = nn.Linear(3, 64)
        self.layer2 = nn.Linear(3, 128)

    def forward(self, x):
        # Reusing a variable many times
        x1 = self.layer1(x)
        x2 = self.layer2(x)
        # Combining variables
        result = torch.cat((x1, x2), 1)
        return result
```

# Some Notes on Data Pipelines

Software University

- In all real scenarios, data must be read sequentially
  - Doesn't fit in RAM
  - Some preprocessing needed
    - e.g., normalization, feature engineering, embedding
- Interplay between CPU (data and I/O) and GPU must be fast
- In tensorflow: `data.Dataset`
  - A tutorial on creating performant data pipelines
- In pytorch: `utils.data.Dataset` and `utils.data.DataLoader`
  - A tutorial on how to set up datasets and data loaders

# Bias and Variance

Machine learning practices using big(ger) data

# Regularization

- Usual L1 and L2 rules apply

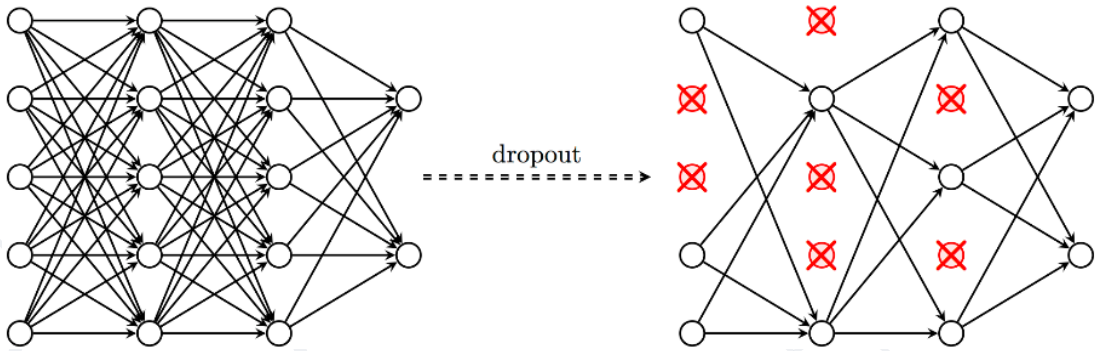
```
from tensorflow.layers import Dense
from tensorflow.keras import regularizers

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 Regularization

- 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
- If used just after the input layer
  - Performs a sort of "feature selection" / "data denoising"

# Selecting and Splitting Data

Software University

- With many samples a 70 / 30 split 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

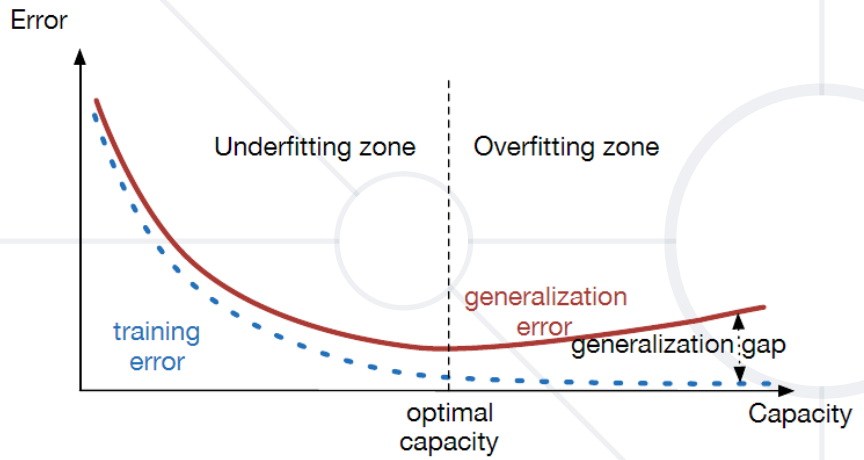
| Algorithm | Train set error | Validation set error | Bias, % | Variance, % | Verdict |
|---|---|---|---|---|---|
| A1 | 1% | 11% | 0,5% | 10% | High variance |
| A2 | 15% | 16% | 14,5% | 1% | High bias |
| A3 | 15% | 30% | 14,5% | 15% | Both |
| A4 | 0,5% | 1% | 0,5% | 0,5% | "Neither" |
| A5 | 0,3%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

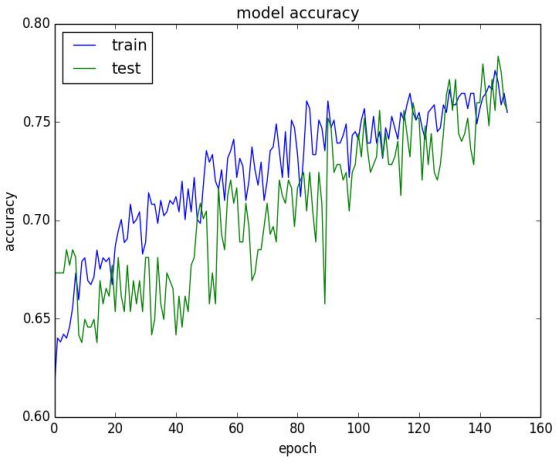
# Taking the Next Step

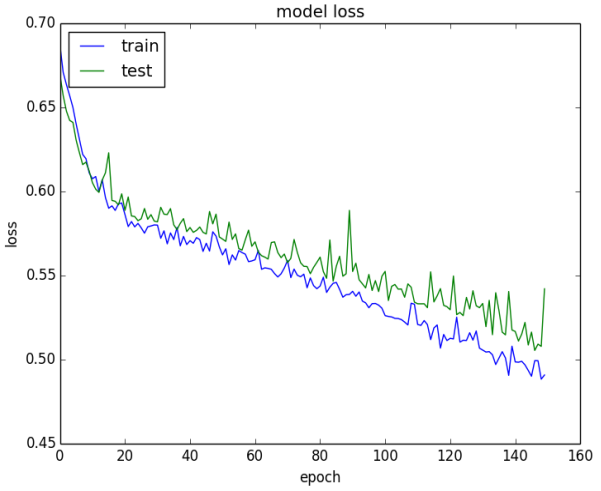
- There are no set rules, only things we can try
- High bias
  - Train a bigger network
  - Possibly, try out different architectures
    - Try to find one which is best suited for the task
  - Train longer (e.g., more **epochs**)
- High variance
  - Apply regularization
  - Try a smaller network architecture
  - 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
- We use model monitoring tools such as `tensorboard` or `mlflow`



# 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 $15^{th}$ layer
    - Ignoring the activation functions for simplicity
    - $a^{[15]} \approx w^{[14]}a^{[14]} \approx w^{[14]}w^{[13]}a^{[13]} \approx \dots \approx w^{[0]}w^{[1]} \dots w^{[14]}x$

# Weight Initialization

- 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 $\approx 0$
- Solution: **initialize the weights properly**

# Weight Initialization

- 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

# Weight Initialization

- tensorflow

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

- In pytorch, we need to set the weights manually

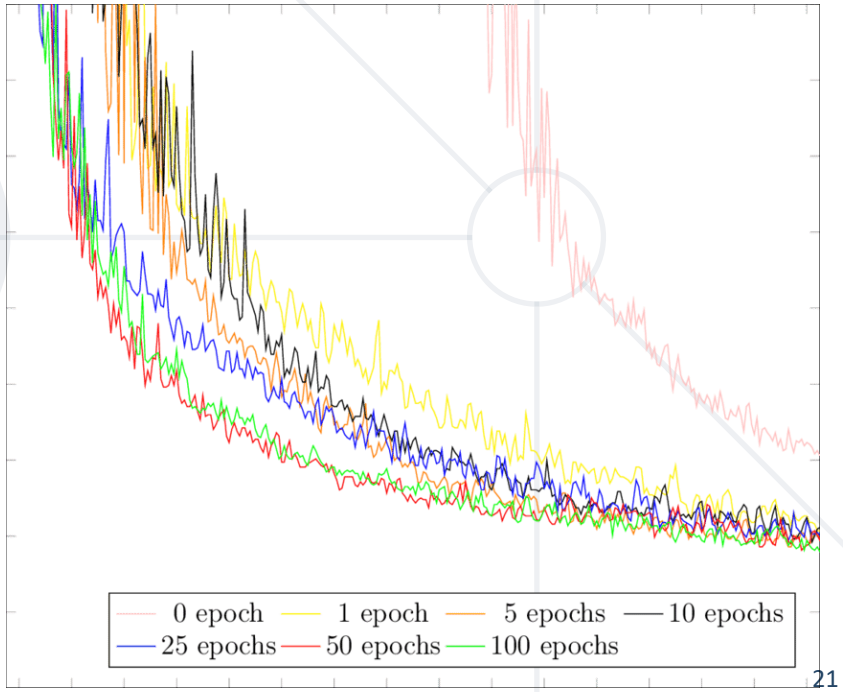
```
layer = torch.nn.Linear(20, 30)
torch.nn.init.uniform_(layer.weight)
```

  - We could also use a custom function for a class (Module)

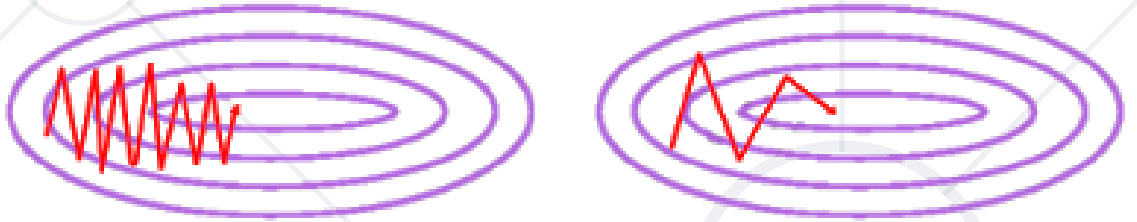
```
@torch.no_grad()
def init_weights(m):
    print(m)
    if type(m) == nn.Linear:
        m.weight.fill_(1.0)
        print(m.weight)

net = ...
net.apply(init_weights)
```

# 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 (sometimes)
    - E.g., 16, **32**, **64**, 128
  - Implementation
    - Shuffle the training set at the start of the epoch
    - At each training step, pass $n_b$ examples



# Improving Gradient Descent



- Momentum
  - When updating weights,
    a fraction $\beta_1$ of the previous vector
    is added to 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

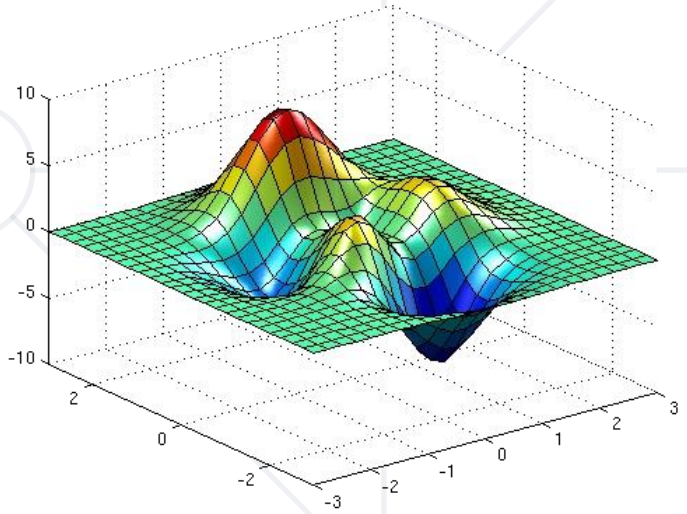
- Adam (**Ada**ptive **M**oment Estimation; Kingma & Ba; 2014)
  - Combines momentum and RMSprop
  - Usually: $\alpha$ (tuning); $\beta_1 = 0{,}9$; $\beta_2 = 0{,}999$; $\varepsilon \in [10^{-6}; 10^{-8}]$
- Usage
  - In place of `GradientDescentOptimizer`
  - It's best to tune all hyperparameters but we may skip $\beta_1, \beta_2$
    - **Tuning $\alpha$ is non-negotiable!**

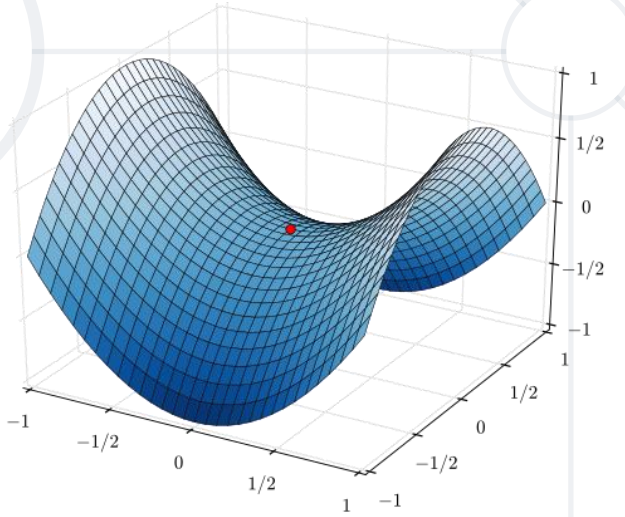
```
tf.train.AdamOptimizer(
    learning_rate = 0.001,
    beta1 = 0.9,
    beta2 = 0.999,
    epsilon = 1e-8)
```

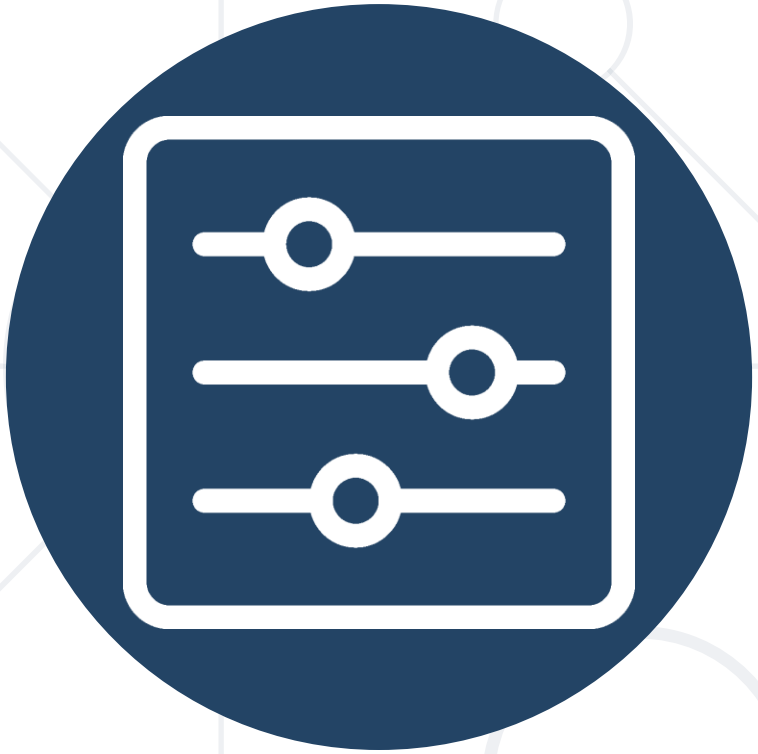
```
torch.optim.Adam(
    model.parameters(),
    lr = 0.001,
    betas = (0.9, 0.999),
    eps = 1e-8)
```

# 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"



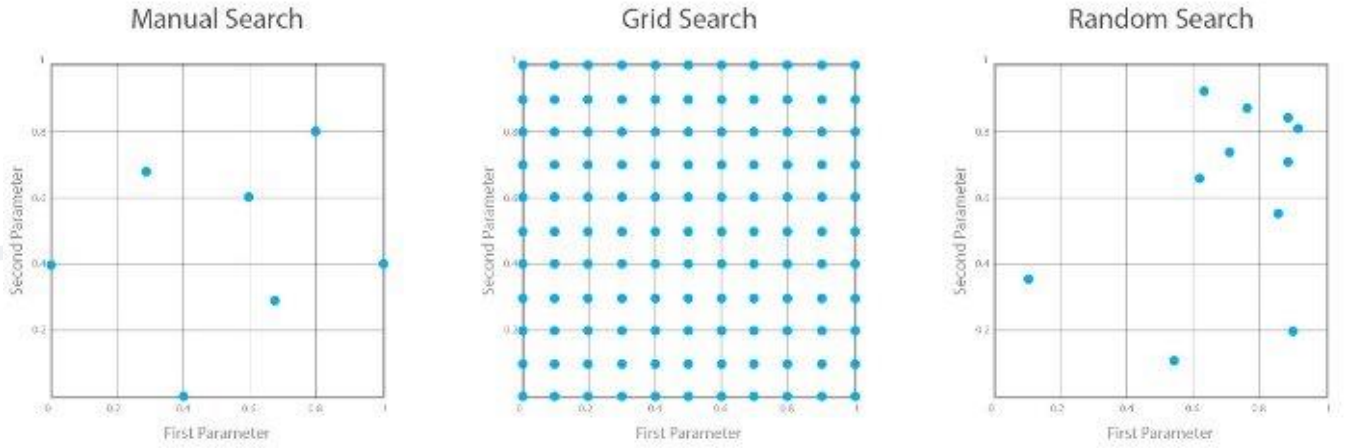


# Hyperparameter Tuning

Similar to "standard" machine learning

# Prioritizing Hyperparameters

- Most important: learning rate $\alpha$
- Momentum term $\beta_1$, 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 or Bayesian search instead
    - Optuna examples: tensorflow / pytorch



# Hyperparameter Scales

- Uniform scale
  - E.g., hidden layers = $\{2, 3, 4\}$, hidden units $\in [50; 100]$
- 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 ($\beta_1, \beta_2$)
  - 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

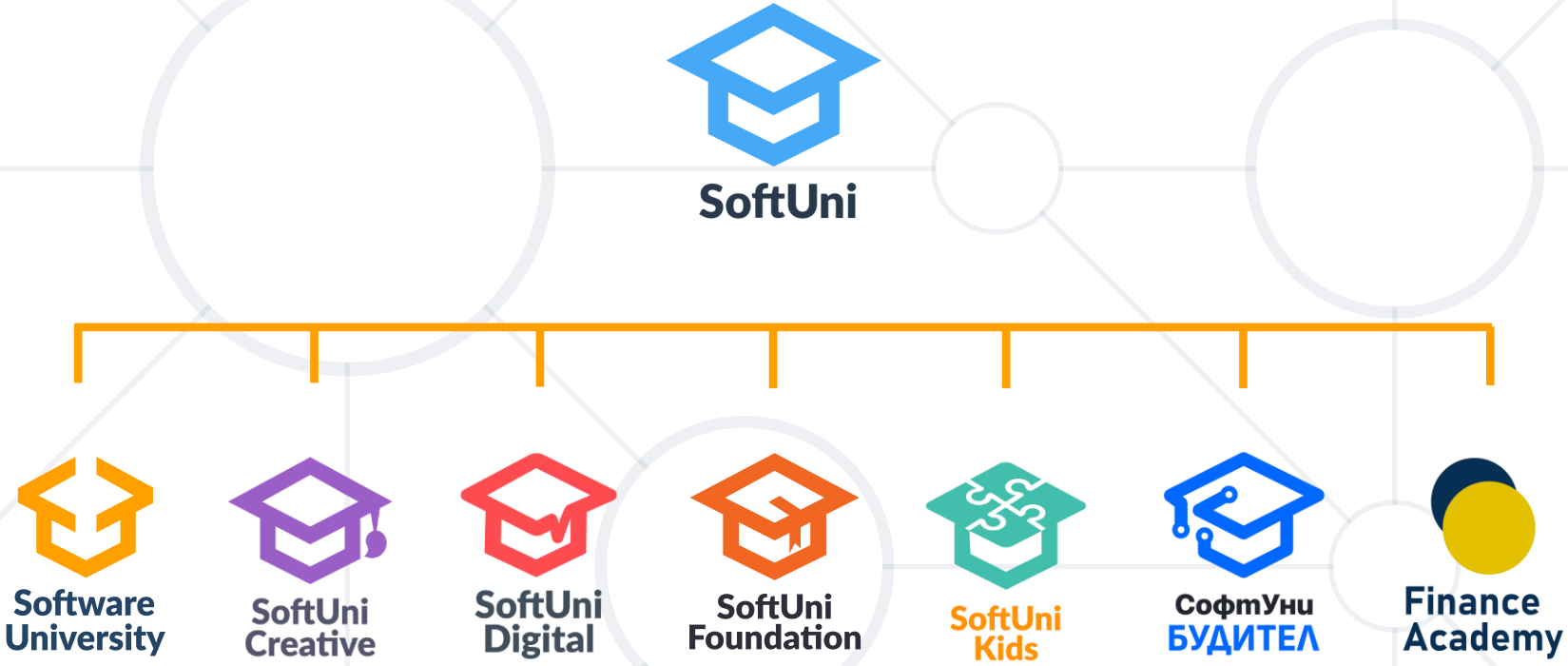
- Normalizing inputs: Z-score
- $x = \frac{x-\mu}{\sigma}, \mu = \frac{1}{n}\sum_{i=1}^{n} x_i, \sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n} x_i^2$
- Batch normalization
  - At a given layer $l$, $z_n = \frac{z-\mu}{\sqrt{\sigma^2+\varepsilon}}$
  - Use a linear transformation $\tilde{z} = \gamma z_n + \beta$ instead of the $z$
    - $\gamma$ and $\beta$ are parameters
    - $\gamma$ and $\beta$ 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

1. Regularization
2. Bias and variance
   - Error analysis
3. Optimization algorithms
4. Hyperparameter tuning
5. Normalization

# Questions?



# SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

top EMPLOYER България Bulgaria 2024 FOR A BETTER WORLD OF WORK

DRAFT KINGS

THE CROWN IS YOURS

VIVACOM

INDEAVR

Serving the high achievers

AMBITIONED

PHAR VISION

# Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

# License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg