Deep Learning

Improving the way neural networks learn

weights initialization, loss functions, optimizers, regularization

POLITÉCNICO DE LEIRIA

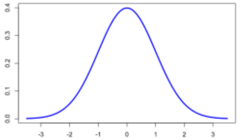
Slides by Carlos Grilo & Rolando Miragaia

Weights initialization

Weights initialization

- In order to avoid units saturation, weights are usually initialized with small values

- Now, suppose that the weights of some layer were generated following a normal distribution with mean 0 and standard deviation 1



- Thus, the number of inputs of the layer is not taken into consideration

- This leads to larger weighted sums when we have more inputs, which may lead to units saturation as well

Xavier/Glorot initialization

- The Xavier initialization, also called the Glorot initialization, initializes weights as follows:
  - First, the weights are generated following a normal distribution with mean 0 and standard deviation 1
  - After this, weights are multiplied by $\sqrt{1/n}$, where n is the number of inputs of the layer to which the weights belong
- This way, the weights will be normally distributed with mean 0 and standard deviation $\sqrt{1/n}$
- That is, the values will get smaller as the number of inputs grows

Xavier/Glorot initialization in Keras

- In Keras, in order to use this type of weights initialization, we use the tf.keras.initializers.GlorotNormal class
- The difference regarding previous slide is that the standard deviation of the normal distribution is equal to

$$\sqrt{\frac{2}{fan_{in} + fan_{out}}}$$

where fan_{in} is the number of inputs of the layer and fan_{out} is the number of outputs of the layer

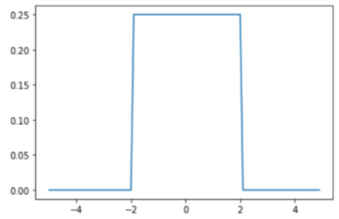
Xavier/Glorot initialization in Keras

- Usage example:

```
initializer = tf.keras.initializers.GlorotNormal()
layer = tf.keras.layers.Dense(3, kernel_initializer=initializer)
```

Keras default weights initializer

- By default, Keras uses the Glorot uniform weights initializer (tf.keras.initializers.GlorotUniform class)
- Weights are initialized following a uniform distribution within $[-limit, limit]$, where



$$limit = \sqrt{\frac{6}{fan_{in} + fan_{out}}}$$

where fan_{in} is the number of inputs of the layer and fan_{out} is the number of outputs of the layer

Loss functions

Problems, activation and loss functions

Problem Type	Last layer activation function	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multi-label classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

Mean Squared Error

$$L(w) = \frac{1}{2N}\sum_{i}^{N} \|y_i - a_i\|^2$$

where

- N is the number of samples
- y_i is the desired output array for sample i
- a_i is the network output array for sample i

MSE computation example 1 Regression to arbitrary values

- If we have two training samples s_1 and s_2 for which
  - $y_{s_1} = [3, 4]$ and $a_{s_1} = [2.1, 2.5]$
  - $y_{s_2} = [5, 2]$ and $a_{s_2} = [6.3, 2.7]$
- Then, $L(w) = \frac{1}{2N} \sum_i^N \|y_i - a_i\|^2 = \frac{1}{2\times 2}\left(\left\| \begin{bmatrix} 3 \\ 4 \end{bmatrix} - \begin{bmatrix} 2.1 \\ 2.5 \end{bmatrix} \right\|^2 + \left\| \begin{bmatrix} 5 \\ 2 \end{bmatrix} - \begin{bmatrix} 6.3 \\ 2.7 \end{bmatrix} \right\|^2 \right) =$

$$= \frac{1}{4}\left(\left\| \begin{bmatrix} 3 - 2.1 \\ 4 - 2.5 \end{bmatrix} \right\|^2 + \left\| \begin{bmatrix} 5 - 6.3 \\ 2 - 2.7 \end{bmatrix} \right\|^2 \right) =$$

$$= \frac{1}{4}\left(\sqrt{(3-2.1)^2 + (4-2.5)^2}^2 + \sqrt{(5-6.3)^2 + (2-2.7)^2}^2 \right) =$$

$$= \frac{1}{4}\left[(3-2.1)^2 + (4-2.5)^2 + (5-6.3)^2 + (2-2.7)^2 \right] = \dots$$

MSE computation example 2 Regression to values between 0 and 1

- If we have two training samples s_1 and s_2 for which
  - $y_{s_1} = [1, 0]$ and $a_{s_1} = [0.8, 0.2]$
  - $y_{s_2} = [1, 1]$ and $a_{s_2} = [0.7, 0.9]$
- Then, $L(w) = \frac{1}{2N}\sum_i^N \|y_i - a_i\|^2 = \frac{1}{2\times 2}\left(\left\|\begin{bmatrix}1\\0\end{bmatrix} - \begin{bmatrix}0.8\\0.2\end{bmatrix}\right\|^2 + \left\|\begin{bmatrix}1\\1\end{bmatrix} - \begin{bmatrix}0.7\\0.9\end{bmatrix}\right\|^2\right) =$

$$= \frac{1}{4}\left(\left\|\begin{bmatrix}1-0.8\\0-0.2\end{bmatrix}\right\|^2 + \left\|\begin{bmatrix}1-0.7\\1-0.9\end{bmatrix}\right\|^2\right) =$$

$$= \frac{1}{4}\left(\sqrt{(1-0.8)^2+(0-0.2)^2}^2 + \sqrt{(1-0.7)^2+(1-0.9)^2}^2\right) =$$

$$= \frac{1}{4}[(1-0.8)^2+(0-0.2)^2+(1-0.7)^2+(1-0.9)^2] = \ldots$$

Binary crossentropy loss for binary classification

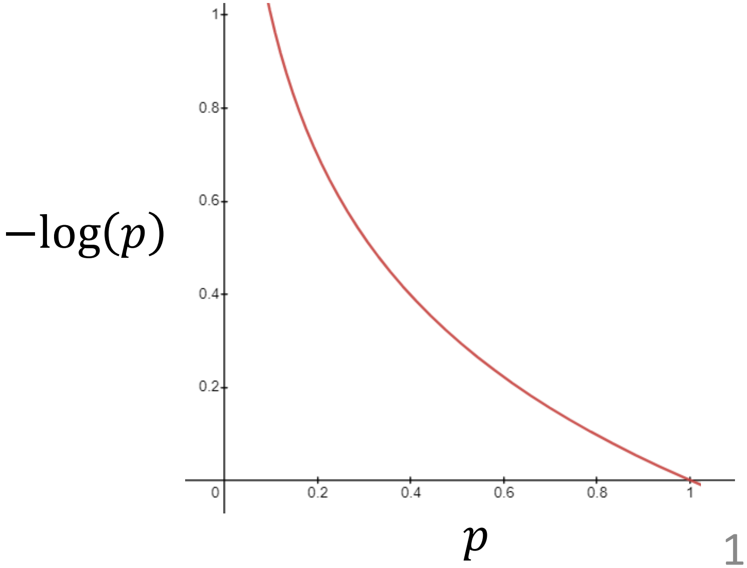
- Each sample should be labelled with label 0 or 1
- The last layer of the network should be composed of one sigmoid unit
- The output of the output unit is the probability that the sample belongs to the class labelled as 1

Binary crossentropy loss

$$L = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p_i) + (1 - y_i)\log(1 - p_i)$$

where

- N is the number of samples
- y_i is the desired output for sample i
- p_i is the current output for sample i



Binary crossentropy loss example

- Consider a binary classification problem where the desired and actual network outputs for 4 samples are as follows:
  - Desired: [1, 0, 1, 0] Actual: [0.8, 0.1, 0.3, 0.8]
- The loss for each of these samples is computed as follows:
  - 1 x log(0.8) + (1 - 1) x log(1 - 0.8) = -0.0969
  - 0 x log(0.1) + (1 - 0) x log(1 - 0.1) = 1 x log(0.9) = -0.0458
  - 1 x log(0.3) + (1 - 1) x log(1 - 0.3) = -0.5229
  - 0 x log(0.8) + (1 - 0) x log(1 - 0.8) = 1 x log(0.2) = -0.6990

Then,

L = -1/4 x (-0.0969 - 0.0458 - 0.5229 - 0.6990) = 0.34

Binary crossentropy loss example

- Consider a binary classification problem where the desired and actual network outputs for 4 samples are as follows:
  - Desired: [1, 0, 1, 0]   Actual: [0.8, 0.1, 0.3, 0.8]

- The loss for each of these samples is computed as follows:
  - 1 x log(0.8) + (1 - 1) x log(1 - 0.8) = -0.0969
  - 0 x log(0.1) + (1 - 0) x log(1 - 0.1) = 1 x log(0.9) = -0.0458
  - 1 x log(0.3) + (1 - 1) x log(1 - 0.3) = -0.5229
  - 0 x log(0.8) + (1 - 0) x log(1 - 0.8) = 1 x log(0.2) = -0.6990

Note that, for each sample, only one term contributes to the loss function (the one corresponding to the class to which the sample belongs)

Binary crossentropy in Keras

- Example:

```
import tensorflow as tf
model.compile(
      loss='binary_crossentropy’,
      optimizer=tf.keras.optimizers.RMSprop(learning_rate=1e-4),
      metrics=['acc'])
```

Categorical crossentropy loss

- The categorical crossentropy loss is used when we have multiclass, single-label classification problems
- Each sample should be labelled with a one-hot encoding vector (only one of the vector elements is 1; all the others have value 0)
- The last layer of the network should be composed of as many units as the number of classes
- The softmax activation function should be used in the output layer units

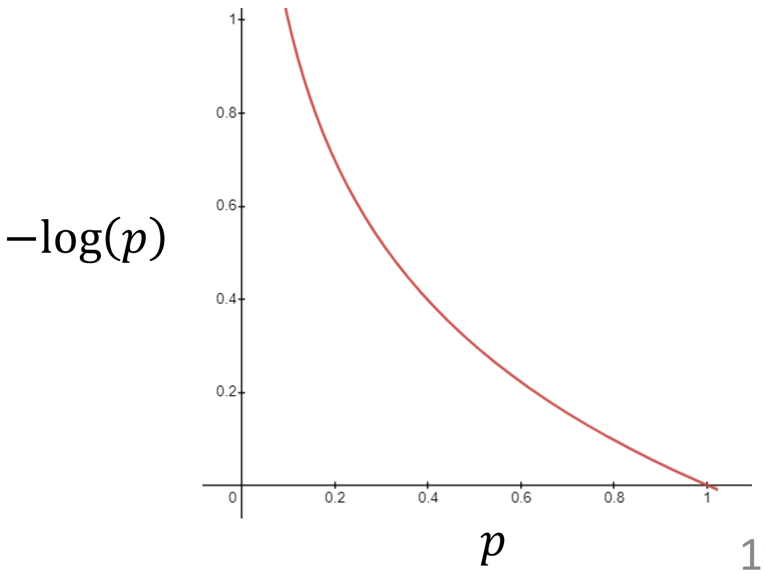
Categorical crossentropy loss

$$L_{s_i} = -\sum_{j=1}^{C} y_{ij} \log(p_{ij})$$

$$L = \frac{1}{N} \sum_{i=1}^{N} L_{s_i}$$

where

- L_{s_i} is the loss for sample s_i
- C is the number of classes
- N is the number of samples
- y_{ij} is the desired output for unit j for sample s_i
- p_{ij} is the current output for output unit for sample s_i



Categorical crossentropy loss example

- Consider a multiclass single-label classification problem where the desired and actual network output for 2 samples are as follows:
  - $y_{s_1} = [1, 0, 0]$ and $a_{s_1} = [0.8, 0.1, 0.2]$
  - $y_{s_2} = [0, 0, 1]$ and $a_{s_2} = [0.04, 0.06, 0.9]$
- The loss for each of these samples is computed as follows:
  - -(1 x log(0.8) + 0 x log(0.1) + 0 x log(0.2)) = 0.0969
  - -(0 x log(0.04) + 0 x log(0.06) + 1 x log(0.9)) = 0.0458

Then,

L = 1/2 x (0.0969 + 0.0458)
= 0.07135

Categorical crossentropy in Keras

- Example:

```
model.compile(
      loss='categorical_crossentropy',
      optimizer='rmsprop’,
      metrics=['accuracy'])
```

Binary crossentropy loss for multiclass, multi-label classification

- The binary crossentropy loss is also used when we have more than two classes, where each sample can belong to more than one class
- Each sample should be labelled with a vector where the elements corresponding to the classes to which the sample belongs have value 1 and all others have value 0
- The last layer of the network should be composed of as many units as the number of classes
- The last layer of the network should be composed of units using the sigmoid activation function
- The output of each unit in the output layer is the probability that the sample belongs to the class the unit stands for

Binary crossentropy loss for multiclass, multi-label classification

$$L_{s_i} = -\sum_{j=1}^{C} y_{ij} \log\left(p_{ij}\right) + (1 - y_{ij}) \log\left(1 - p_{ij}\right)$$

$$L = \frac{1}{N}\sum_{i=1}^{N} L_{s_i}$$

where

- L_{s_i} is the loss for sample s_i
- C is the number of classes
- N is the number of samples
- y_{ij} is the desired output for unit j for sample s_i
- p_{ij} is the current output for unit j for sample s_i

Binary crossentropy loss example

- Consider a multiclass, multi-label, classification problem, where the desired and actual network outputs for 2 samples are as follows:
  - $y_{s_1} = [1, 0, 1]$ and $a_{s_1} = [0.8, 0.3, 0.9]$
  - $y_{s_2} = [1, 1, 0]$ and $a_{s_2} = [0.7, 0.9, 0.1]$
- The loss for each of these samples is computed as follows:
  - $L_{s_i} = -(\sum_{j=1}^{C} y_{ij} \log(p_{ij}) + (1 - y_{ij}) \log(1 - p_{ij})$
  - -(1 x log(0.8) + 0 x log(0.2) + 0 x log(0.3) + 1 x log(0.7) + 1 x log(0.9) + 0 x log(0.1)) = 0.2976
  - -(1 x log(0.7) + 0 x log(0.3) + 1 x log(0.9) + 0 x log(0.1) + 0 x log(0.1) + 1 x log(0.9)) = 0.2464

Binary crossentropy loss example (cont.)

- We now compute the average of these values
  - $L = \frac{1}{N}\sum_{i=1}^{N} L_{s_i}$= 1/2 x (0.2976 + 0.2464) = 0.272

Exercise

- Show that the binary crossentropy loss function is equivalent to the categorical crossentropy loss function when there are two classes

Optimizers

Optimizers

- Optimizers are algorithms used to change the parameters of neural networks such as weights and bias in order to reduce the loss

- Some optimizers also change the learning rate during the training process

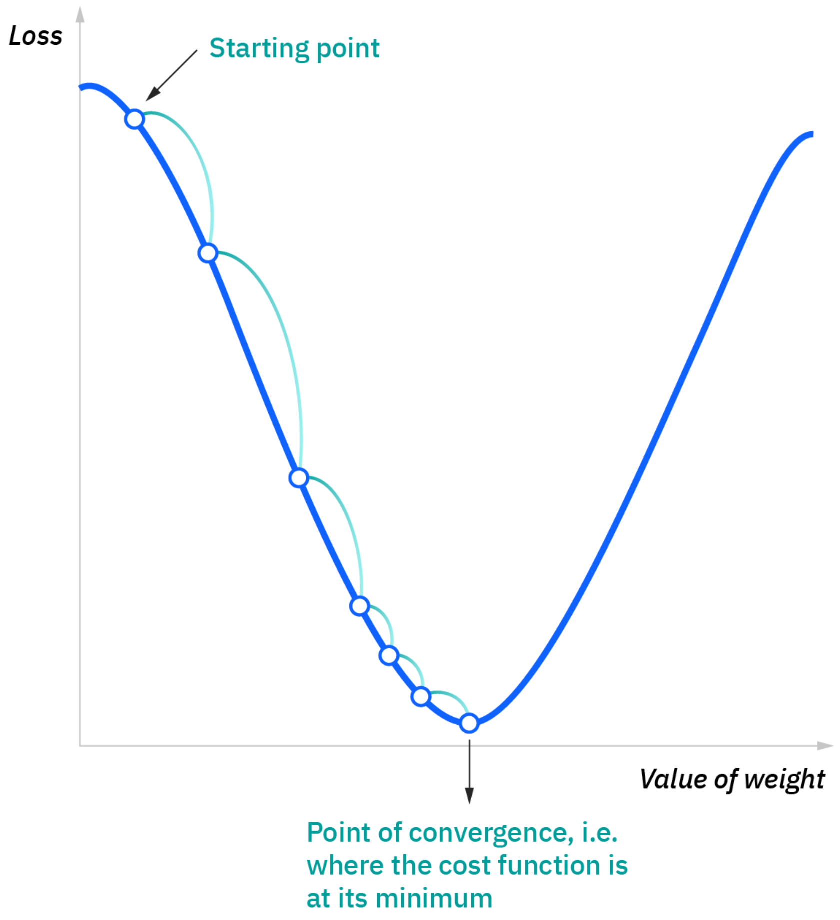
- How we should change the weights or learning rates of the neural network to reduce the loss is defined by the optimizers we use

Gradient Descent

- Gradient descent is an optimization algorithm which is commonly used to train machine learning models and neural networks
- Training data helps these models learn over time, and the loss function within gradient descent specifically acts as a barometer
- The model will continue to adjust its parameters to yield the smallest possible error until it is close to or equal to zero
- The main idea is to change the weights in the opposite direction of the loss gradient

Gradient Descent

- From the starting point, we find the derivative (or slope) of the loss function
- The slope will inform the updates to the parameters (weights and bias)
- The slope at the starting point will be steeper, but as new parameters are generated, the steepness should gradually reduce until it reaches the lowest point on the curve, known as the point of convergence
- The goal of gradient descent is to minimize the cost function, or the error between predicted and actual y
- The variation of each weight depends on two values – the partial derivative of the loss function along that weight and a learning rate. This process is iterative, allowing to gradually arrive at the local or global minimum (i.e. point of convergence)


Loss
Starting point
Value of weight
Point of convergence, i.e. where the cost function is at its minimum

Gradient Descent

- Gradient Descent is the most basic and most popular optimization algorithm
- The main idea is to perform the loss gradient along weights and bias

$$\Delta w = -\eta \frac{\partial L}{\partial w} \equiv w_{new} = w_{old} - \eta \frac{\partial L}{\partial w} \equiv w_{t+1} = w_t - \eta \frac{\partial L}{\partial w_t}$$

Gradient Descent

- There are three types or variants of gradient descent learning algorithms
  - Batch gradient descent
  - Stochastic gradient descent
  - Mini-batch gradient descent

- These types differ from each other in what concerns the number of examples from the training set used to compute the loss and update the weights

Gradient Descent – Batch gradient descent

- Batch gradient descent sums the error for each point in a training set, updating the model only after all training samples have been evaluated. This process is referred to as a training epoch
- While this batching provides computation efficiency, it can still have a long processing time for large training datasets as it still needs to store all of the data into memory
- Batch gradient descent also usually produces a stable error gradient and convergence, but sometimes the convergence point isn’t the most ideal, finding the local minimum versus the global one

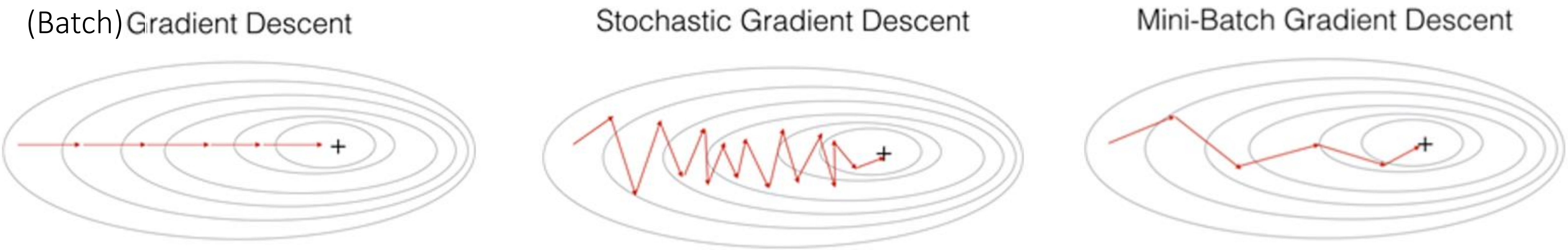
Gradient Descent – Stochastic gradient descent

- Stochastic gradient descent (SGD) runs a training step for each example within the dataset and it updates each training example's parameters one at a time
- Since we only need to hold one training example, they are easier to store in memory
- While these frequent updates can offer more detail and speed, it can result in losses in computational efficiency when compared to batch gradient descent
- Its frequent updates can result in noisy gradients, but this can also be helpful in escaping the local minimum and finding the global one

Gradient Descent – Mini-batch gradient descent

- Mini-batch gradient descent combines batch gradient descent and stochastic gradient descent
- It splits the training dataset into small batch sizes and performs updates on each of those batches
- This approach strikes a balance between the computational efficiency of batch gradient descent and the speed of stochastic gradient descent

Gradient Descent



(Batch) Gradient Descent

Stochastic Gradient Descent

Mini-Batch Gradient Descent

Gradient Descent with momentum

- Momentum reduces high variance in Gradient Descent and softens the convergence
- It accelerates the convergence towards the relevant direction and reduces the fluctuation to irrelevant directions
- One more hyperparameter is used in this method known as momentum, here μ

$$v_o = 0$$

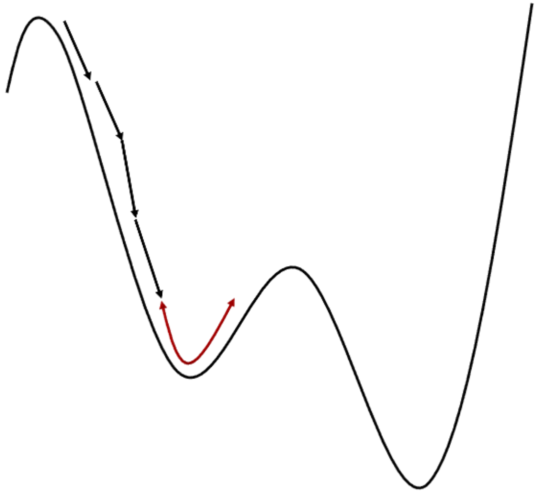
$$v_t = \mu v_{t-1} + (1 - \mu)\ \frac{\partial L}{\partial w_t}$$

Typically, $\mu = 0.90$ or 0.95

$$w_{t+1} = w_t - \eta v_t$$

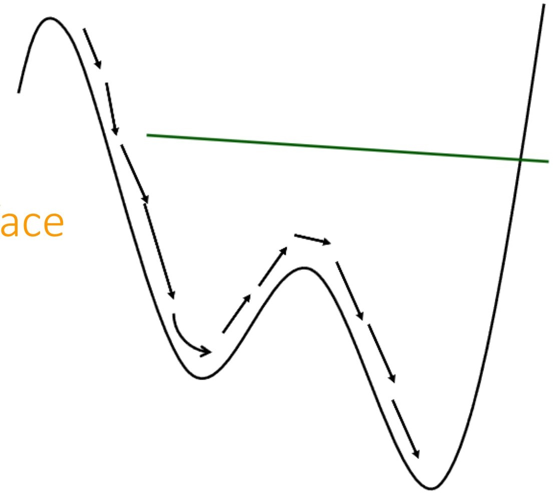
Gradient Descent with momentum

Without momentum



Loss surface

With momentum

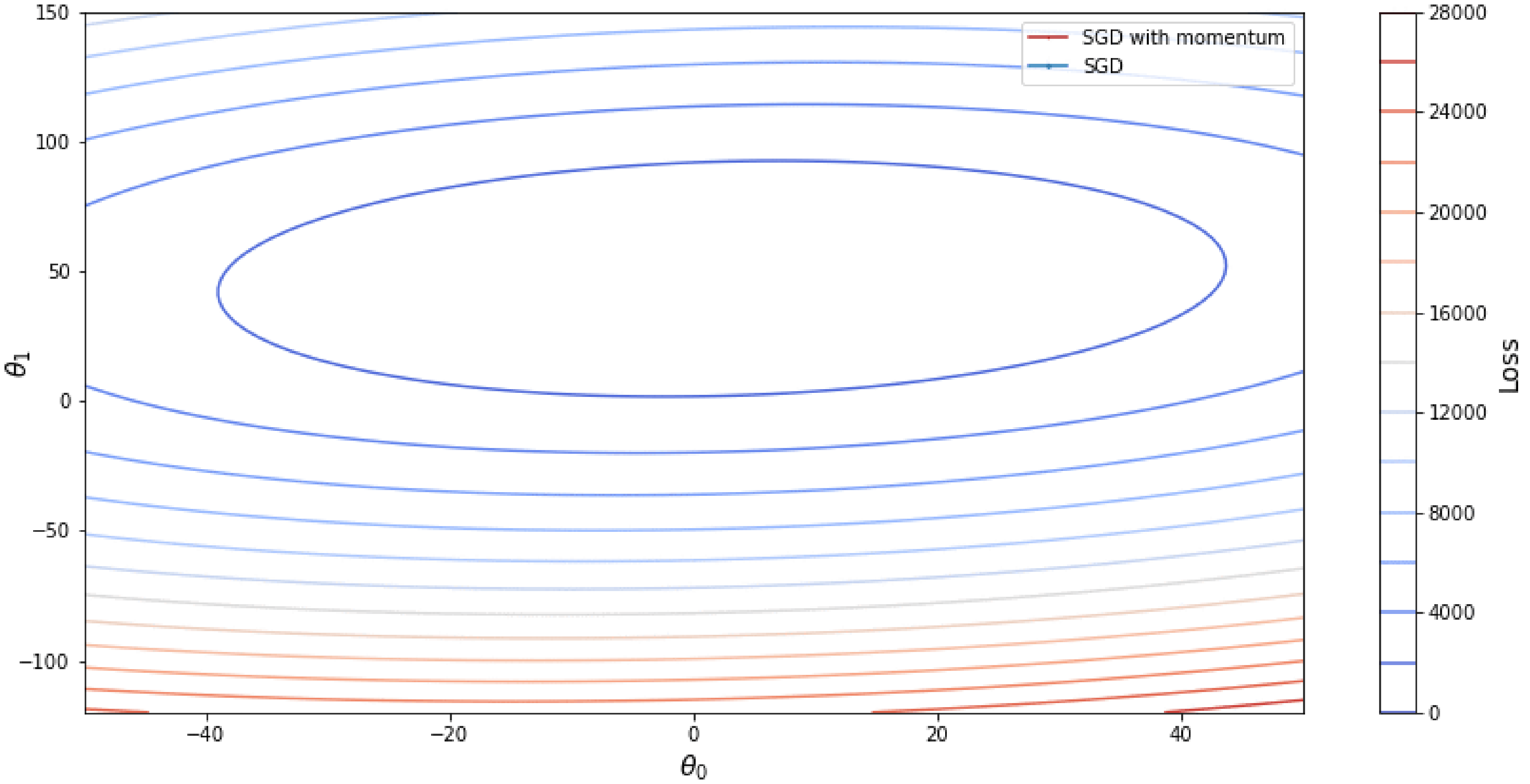


Sucessive updates in the same direction tend to increase the magnitude of the modifications, leading to a faster convergence

It’s harder to escape from local minima due to oscilations

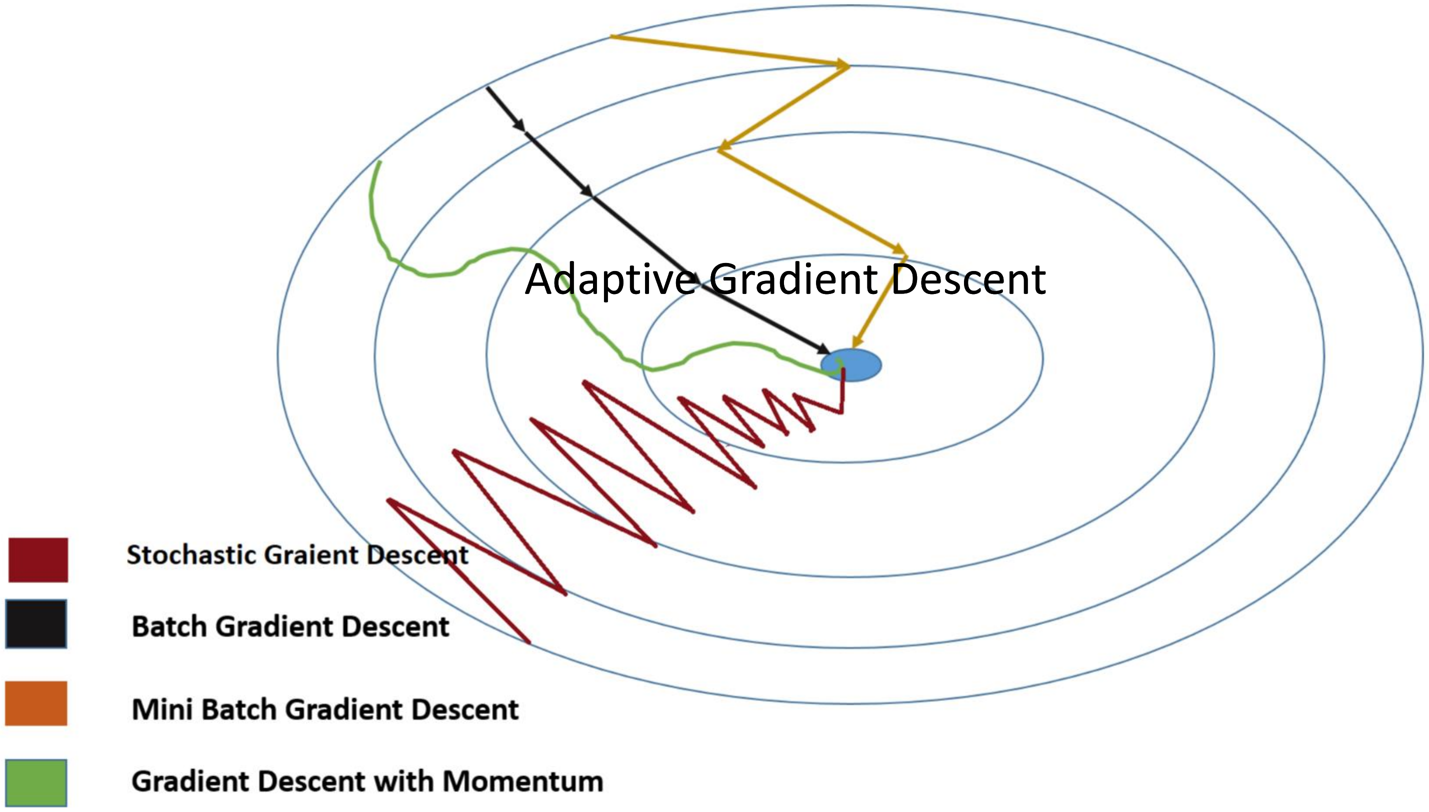
It’s easear to escape from local minima: weights tend to be modified in the direction of previous modifications

Gradient Descent with momentum



Gradient Descent with momentum

Gradient descent flavours



🟥 Stochastic Graient Descent

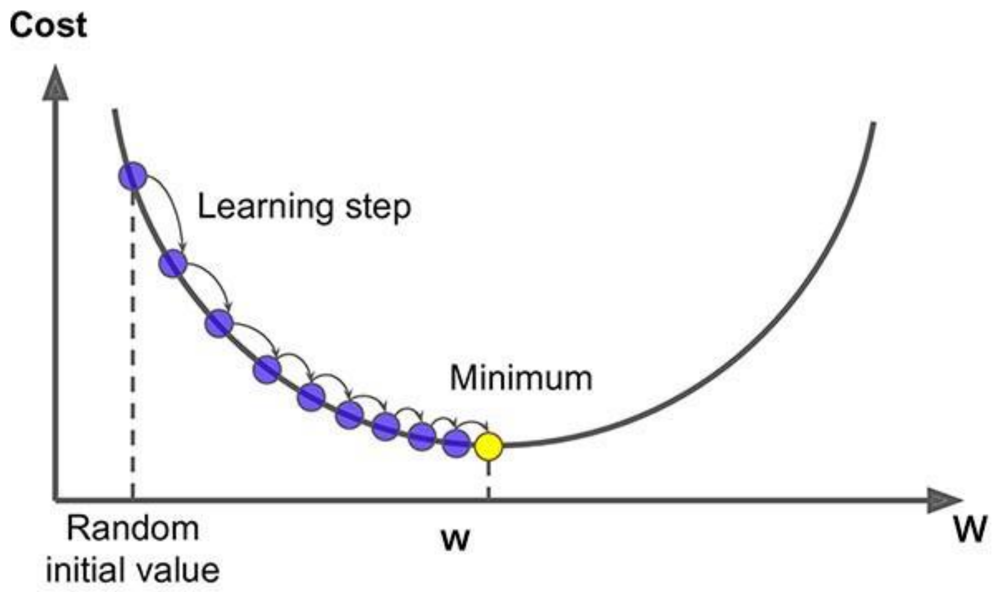
⬛ Batch Gradient Descent

🟧 Mini Batch Gradient Descent

🟩 Gradient Descent with Momentum

Adaptive Gradient Descent (AdaGrad)

- With AdaGrad, the learning rate η varies dynamically
- This feature is expected to help us reach the optimum faster than those methods with constant learning rate
- The closer the loss gets to a minimum, the learning rate is reduced for smoother convergence



Adaptive Gradient Descent (AdaGrad)

- Weights are updated as follows

$$w_{t+1} = w_t - \eta_t' \frac{\partial L}{\partial w_t}$$

where

$$\eta_t' = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

$$\alpha_t = \sum_{i=1}^{t} (\frac{\partial L}{\partial w_i})^2$$

η is the initial learning rate

η_t' is the dynamical learning rate

α_t is the sum of the squared gradients until time t

ϵ is a small positive number added to avoid zero division error

Adaptive Gradient Descent (AdaGrad)

- Gradients until *t* are computed, squared (to increase the emphasis), and summed together
- This value is expected to increase with time
- A small positive value (ϵ) is added to avoid the #DIV/0! error
- So, for every step, the initial learning rate is divided by the square root of $(\alpha_t + \epsilon)$ (which increases with steps), thereby reducing the learning rate, resulting in smoothing

Adaptive Gradient Descent (AdaGrad)

- Problem with AdaGrad:
  - Since the learning rate decreases with time, the changes to the weights become smaller with time
  - In convex loss surfaces this may be a good thing
  - It becomes a problem in non-convex loss surfaces because it may turn difficult escaping to local minima

Root Mean Square Propagation (RMS Prop)

- RMS Prop is a widely used optimization technique for mini-batch learning of neural networks
- This process averages the gradients over successive mini-batches so that weights can be finely calibrated and, thus, regulating the diminishing gradient problem by Adagrad
- RMS Prop deals with the diminishing gradient problem by ”exponentially smoothing the learning rate (like for gradients in Momentum with SGD)”

Root Mean Square Propagation (RMS Prop)

- Weights are updated as follows

$$s_t = \rho s_{t-1} + (1-\rho)(\frac{\partial L}{\partial w_t})^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \frac{\partial L}{\partial w_t}$$

s_t is the moving average of the squared gradients

η is the (initial) learning rate

$\frac{\eta}{\sqrt{s_t + \epsilon}}$ is the dynamical learning rate

ϵ is a small positive number added to avoid zero division error

Root Mean Square Propagation (RMS Prop)

- The moving averages for squared gradient helps us regulate the problem of Adagrad
- It uses the magnitude of the recent gradient descents to normalize the gradient
- In this process, the learning rate never becomes too small, hence solving the problem

Adaptive Moment (ADAM)

- Intuitively, ADAM is a combination of the gradient descent with momentum algorithm and the RMSProp algorithm
- This optimizer updates the parameters based on the learning rate from RMSProp and by using smoothing of gradients from momentum with SGD

Adaptive Moment (ADAM)

- Weights are updated as follows

$$w_{t+1} = w_t - \eta_t' \widehat{v_t}$$

where

$$\eta_t' = \frac{\eta}{\sqrt{\widehat{s_t} + \epsilon}}$$

$$\widehat{v_t} = \frac{v_t}{1-\beta_1}, \qquad \widehat{s_t} = \frac{s_t}{1-\beta_2}$$

$$v_t \ = \beta_1 v_{t-1} + (1-\beta_1)\frac{\partial L}{\partial w_t}$$

$$s_t = \beta_2 s_{t-1} + (1-\beta_2)(\frac{\partial L}{\partial w_t})^2$$

η is the initial learning rate

η_t' is the dynamical learning rate

v_t is the moving average of the gradients

s_t is the moving average of the squared gradients

ϵ is a small positive number added to avoid zero division error

Comparison

Optimizer	Advantages	Disadvantages
BGD	• Easily interpretable • Less complex as it is a first-order derivative	• With a huge dataset, it can take a long time to converge • Massive memory requirements to compute gradient on the whole dataset
SGD	• Frequent parameter updating, thus converging in less time • Requires less memory storage for gradient computation	• High variance in model parameters • The learning rate has to be modified for the model to perform similarly to GD or for the same learning rate, data has to be processed in batches to reduce parameter variance
MBGD	• MBGD clearly reduces parameter variance compared to SGD and updates frequently compared to GD (best of both worlds) • Requires less storage compared to GD	• As the learning rate is constant, the convergence is not smooth

Comparison

Optimizer	Advantages	Disadvantages
SGD with momentum	• Reduces variance of parameters and converges faster than SGD	• Constant learning rate might take more steps to converge • One more hyper-parameter (μ) to be selected manually (usually 0.9/0.95)
Adagrad	• No need to manually tune the learning rate • Converges smoothly compared to previous techniques • Every parameter can have different learning rates (as RMS Prop and ADAM)	• Computationally expensive when compared to the SGD family optimizers • For very deep neural networks, the parameters may radically diminish leading to a *Vanishing gradient problem*
RMS Prop	• Learning rate adjusts based on recent gradients and not all gradients • Can better handle sparse gradients on datasets (as well as ADAM)	
ADAM	• Default values work well in most cases • Requires less storage and computationally efficient • Works well on large datasets	• Can suffer weight decay and in some cases may not converge to an optimal solution

Regularization

Regularization

- Regularization techniques are used to avoid/reduce overfitting
- Techniques we will talk about:
  - L2 regularization
  - L1 regularization
  - Dropout
  - Dataset augmentation
  - Reducing the model size
  - Batch normalization

L2 regularization

- The L2 regularization technique consists in adding an extra term to the cost function:

$$L(w) = L_0 + \frac{\lambda}{2N}\sum_{w} w^2$$

- where
  - L_0 is the original loss function (MSE or crossentropy loss, for example)
  - N is the number of samples
  - λ is a constant called the regularization parameter
  - w are the weights of the model

L2 regularization

- Intuitively, L2 regularization leads the network to “prefer” to learn small weights, all other things being equal
- Large weights will only be allowed if they considerably improve the first part of the cost function
- Put another way, regularization can be viewed as a way of compromising between finding small weights and minimizing the original cost function

L2 regularization and λ

- The relative importance of the two terms of the loss function depends on the value of λ:
  - when λ is small, we prefer to minimize the original cost function
  - when λ is large, we prefer small weights

L2 regularization – weights update

- The expression used to update the weights is as follows:

$$w(t+1) = w - \frac{\eta\lambda}{N} w - \eta \frac{\partial L_0}{\partial w}$$

- This is exactly the same as the usual gradient descent learning rule, except we first rescale the weight w by a factor $1 - \frac{\eta\lambda}{N}$
- This rescaling is sometimes referred to as weight decay, since it makes the weights smaller

Why does regularization helps avoiding overfitting?

- Empirically, regularized neural networks usually generalize better than unregularized networks
- However, no-one has yet developed an entirely convincing theoretical explanation for why regularization helps networks generalize
- While it often helps, we don’t have an entirely satisfactory systematic understanding of what’s going on, merely incomplete heuristics and rules of thumb

What follows is an intuitive explanation for why does regularization helps avoiding overfitting

Why does regularization helps avoiding overfitting?

- Consider a regularized network with small weights
- The behavior of this network won't change too much if we do small changes to the inputs
- That is, regularized networks resist to learn the noise in the training data and this leads them to learn simpler models based on patterns seen often in the training data
- The hope is that this will force these networks to do real learning about the phenomenon at hand, and to generalize better from what they learn

Why does regularization help avoiding overfitting?

- A network with large weights may change its behaviour considerably in response to small changes in the input
- An unregularized network can use large weights to learn a complex model that carries a lot of information about the noise in the training data

L1 regularization

- The L1 regularization technique consists in adding the following extra term to the cost function:

$$L(w) = L_0 + \frac{\lambda}{N} \sum_{w} |w|$$

- where
  - L_0 is the original loss function (MSE or crossentropy loss, for example)
  - N is the number of samples
  - λ is a constant called the regularization parameter
  - w are the weights of the model

L1 regularization – weights update

- The expression used to update the weights is as follows:

$$w(t+1) = w - \frac{\eta\lambda}{N}\operatorname{sgn}(w) - \eta\,\frac{\partial L_0}{\partial w}$$

- $\operatorname{sgn}(w)$: signal of w (-1 if $w < 0$; 1, otherwise)

L1 and L2 regularization

- Both L1 and L2 shrinks the weights
- In L1 regularization, the weights decrease by a constant amount
- In L2 regularization, the weights decrease by an amount proportional to w
- When a weight has a large magnitude, $|w|$, L1 regularization shrinks the weight much less than L2 regularization does
- When $|w|$ is small, L1 regularization shrinks the weight much more than L2 regularization
- The net result is that L1 regularization tends to concentrate the weight of the network in a relatively small number of high-importance connections, while the other weights are driven toward zero

L1 regularization in Keras

- Usage examples:

```
layer = layers.Dense(
  units=5,
  kernel_regularizer=regularizers.L1(0.01))

#Using the default parameters
dense = tf.keras.layers.Dense(3, kernel_regularizer='l1')
```

The l1 parameter corresponds to the value of λ (see previous slides)

L2 Regularization in Keras

- Usage examples:

The l2 parameter corresponds to the value of λ (see previous slides)

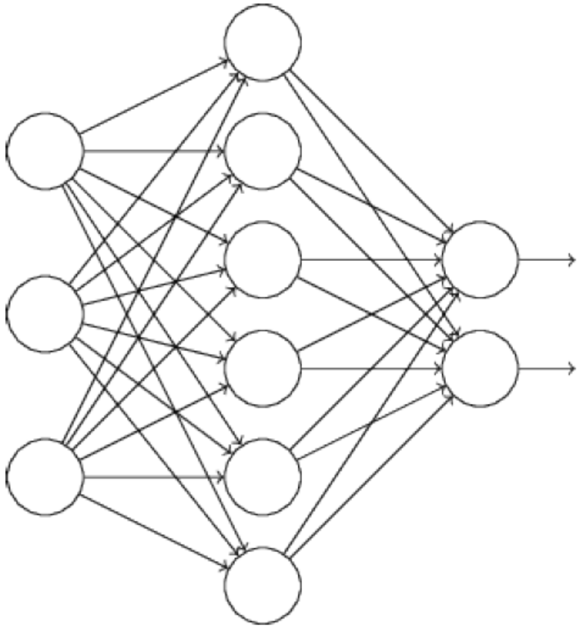
```
layer = layers.Dense(
   units=5,
   kernel_regularizer=regularizers.L2(0.01))

#Using the default parameters
dense = tf.keras.layers.Dense(3, kernel_regularizer='l2')
```

Keras also includes the L1L2 regularizer that applies both L1 and L2 regularization

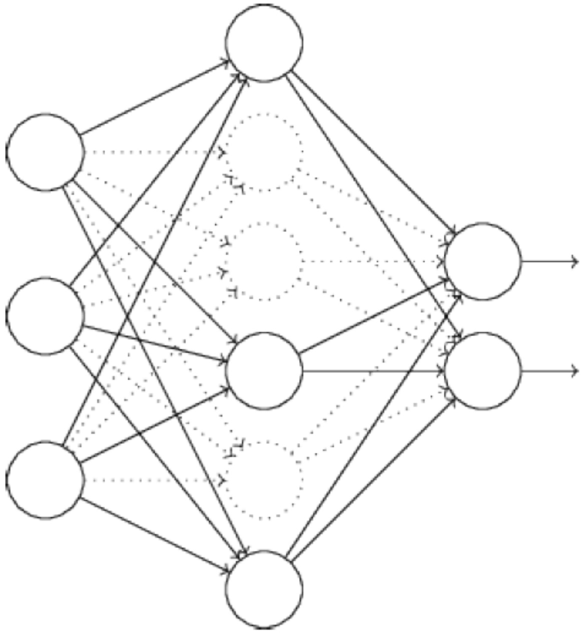
Dropout

- Consider the following network:



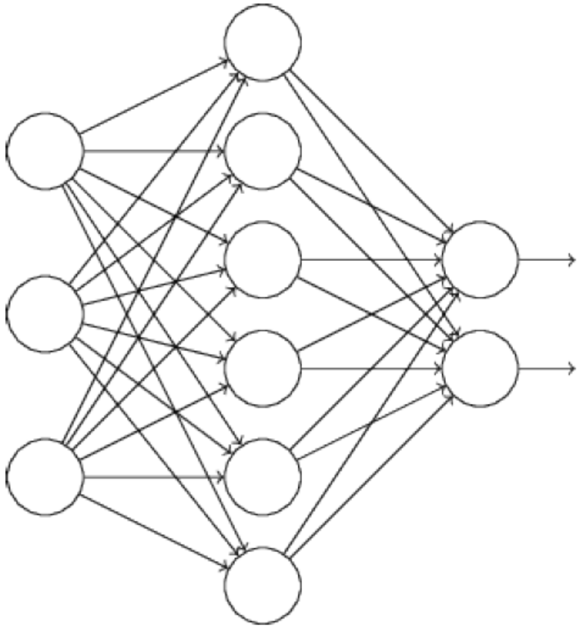
Dropout consists in, for each batch, switch off (set their output to 0) some proportion of randomly chosen units of one layer

This is done both in the forward and in the backward steps



Dropout

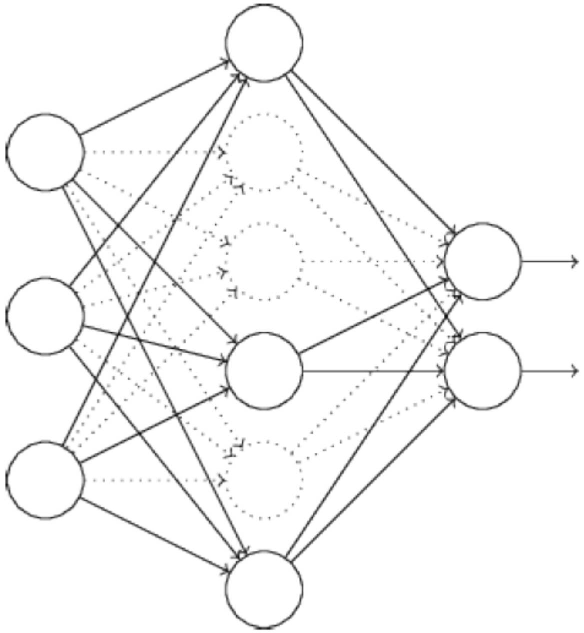
Common dropout rate: 0.5



The weights are learnt with some proportion of units dropped out

But, after the training process, all the units become active

To compensate for that, the weights are scaled by the chosen dropout rate



Why does dropout help avoiding overfitting?

- “This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.”

Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. ImageNet Classification with Deep Convolutional Neural Networks, 2012.

Dataset augmentation

- The best way to make a machine learning model generalize better is to train it on more data
- Data augmentation consists in generating more training data from existing training samples, by augmenting the samples via a number of random transformations that yield believable-looking images
- This technique is used mainly when the dataset is small

Reducing model size

- We can reduce
  - the number of layers
  - The number of units in the layers

Batch normalization

- During training, the outputs for every layer change, which means that the distribution of input data for every layer will change every iteration
- This is called **internal covariance shift**
- Internal covariance shift slows down learning
- One way of solving this is to use batch normalization

Batch normalization (training)

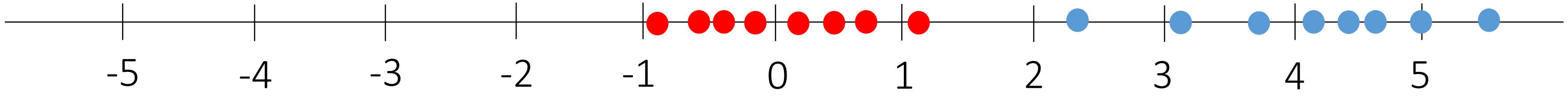
- When we apply batch normalization to one layer, the output of the units of that layer during the training process is first normalized as follows:

$$h_i^{k\,norm} = \frac{h_i^k - \mu^k}{\sigma^k}$$

This transformation sets the mean of the layer outputs close to 0 and the standard deviation close to 1

- where
  - h_i^k is the original output of unit *k* to the *i*th sample of the batch
  - $\mu^k = \frac{1}{m}\sum_{i=1}^{m} h_i^k$ is the average of the output for unit *k* along all the samples in the batch
  - $\sigma^k = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h_i^k - \mu^k)^2 + \epsilon}$ is a modified standard deviation of the output for unit *k* (ϵ is added to avoid division by 0 in cases where the variance is 0)

Normalization



- ● Original values
- ● Normalized values (centered around 0 and with standard deviation 1

Batch normalization (training)

- Then, the final output of the layer units becomes

$$h_i^{k^{final}} = \gamma^k h_i^{k^{norm}} + \beta^k$$

- $\gamma^k$and $\beta^k$are learned parameters that allow the output of the units to have any mean and standard deviation
- $\gamma^k$and $\beta^k$are learned as other parameters (weights)
- When $\beta^k$is used, there is no need to use biases (in the normalized layer)

Batch normalization (training)

- Besides, the moving averages of μ^k and σ^k are also computed

$$\mu_{mov}^k = \alpha\mu_{mov}^k + (1 - \alpha)\mu^k$$

$$\sigma_{mov}^k = \alpha\sigma_{mov}^k + (1 - \alpha)\sigma^k$$

- These moving averages are used only after the training process (see next slide)

Batch normalization (inference)

- For inference, the output of units of the normalized layers is computed as follows:

$$h_i^{k\,norm} = \frac{h_i^k - \mu_{mov}^k}{\sigma_{mov}^k} \quad , \quad h_i^{k\,final} = \gamma^k h_i^{k\,norm} + \beta^k$$

- where
  - h_i^k is the original output of unit *k* to the *ith* sample of the batch
  - μ_{mov}^k is the moving average of μ^k computed during the training process
  - σ_{mov}^k is the moving average of σ^kcomputed during the training process

Batch normalization (inference)

- Ideally, we could have calculated and saved the average and standard deviation for the full data during training
- But that would be very expensive as we would have to keep values for the full dataset in memory during training
- Instead, the moving average acts as a good approximation for the average and standard deviation of the data
- It is much more efficient because the calculation is incremental — we have to remember only the most recent moving average

Batch normalization

- Benefits attributed to batch normalization:
  - It reduces **internal covariance shift**
  - Higher learning rates can be used
  - It improves the generalization capabilities of the model

Batch normalization in Keras

```
tf.keras.layers.BatchNormalization(
    axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True,
    beta_initializer="zeros",
    gamma_initializer="ones",
    moving_mean_initializer="zeros",
    moving_variance_initializer="ones",
    beta_regularizer=None,
    gamma_regularizer=None,
    beta_constraint=None,
    gamma_constraint=None,
    **kwargs
)
```

In Keras, we introduce batch normalization as a layer after the layer whose outputs we want to normalize

We can use it both after convolutional layers and dense ones

Bibliography

- Michael Nielsen, Neural networks and deep learning (Chapter 3), 2019, http://neuralnetworksanddeeplearning.com/
- Ian Goodfellow and Yoshua Bengio and Aaron Courville, Deep Learning (Chapter 8), MIT Press, 2016, https://www.deeplearningbook.org/