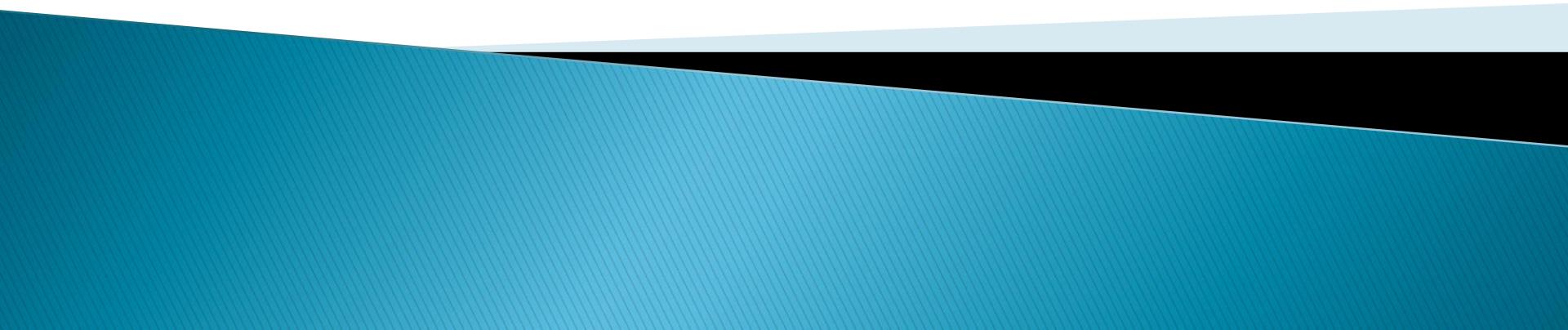
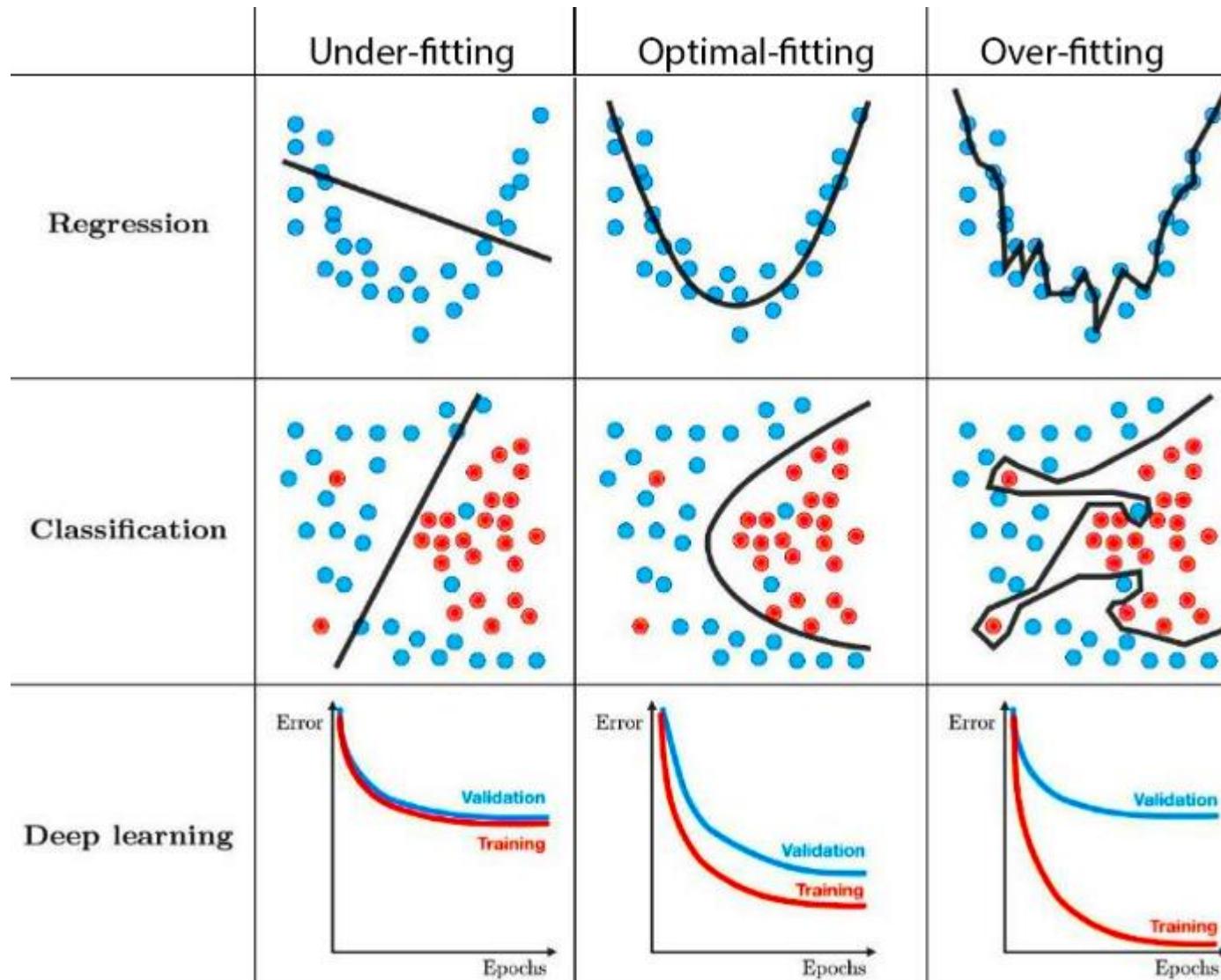
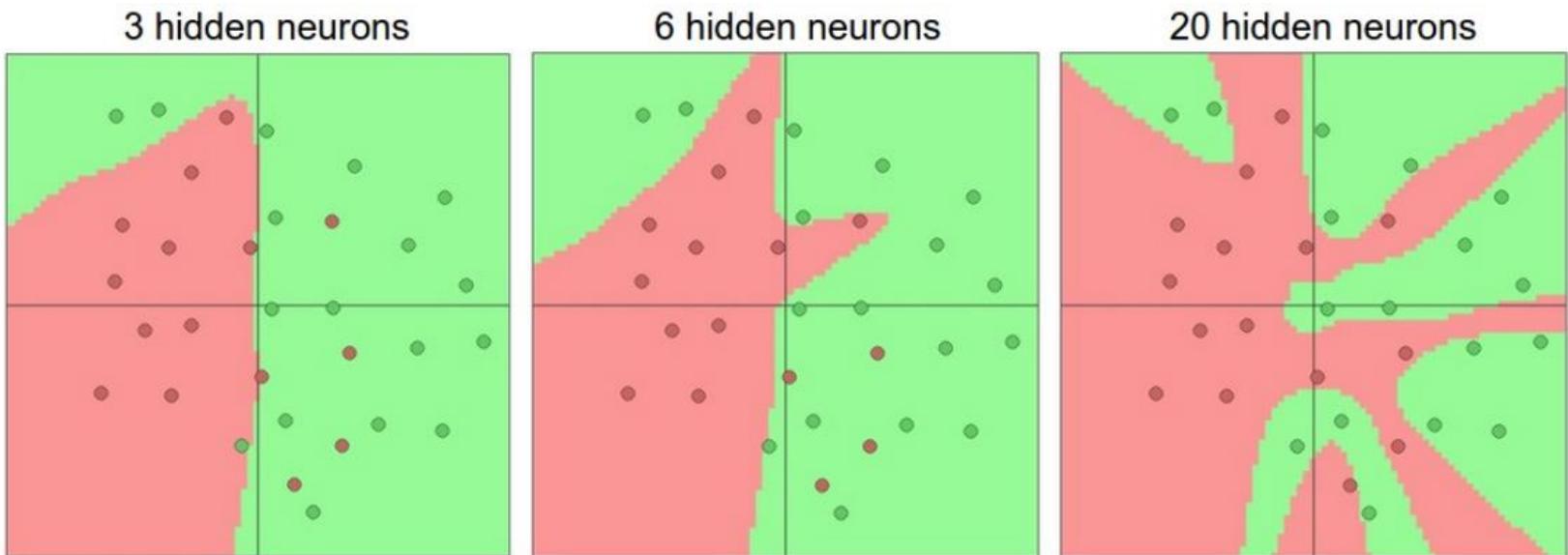


# Overfitting





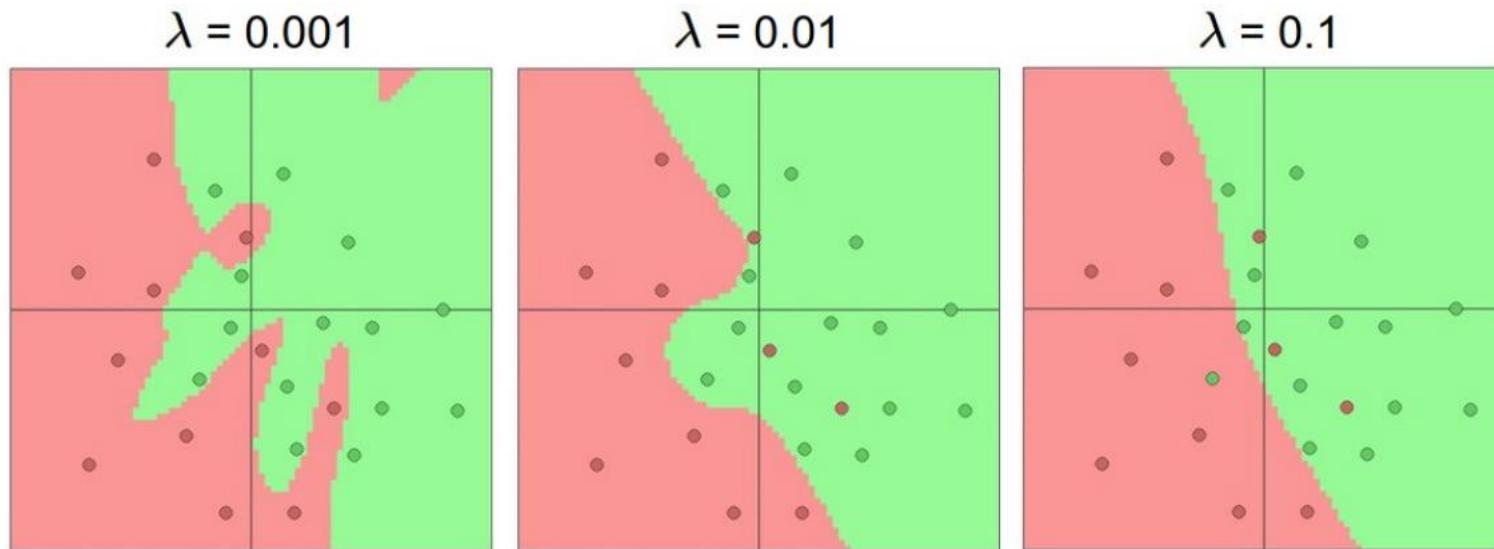


Larger Neural Networks can represent more complicated functions. The data are shown as circles colored by their class, and the decision regions by a trained neural network are shown underneath.

# Regularization

- $L_2 = \lambda \sum_w w^2;$
- $L_1 = \lambda \sum_w |w|.$

Додається до Loss Function

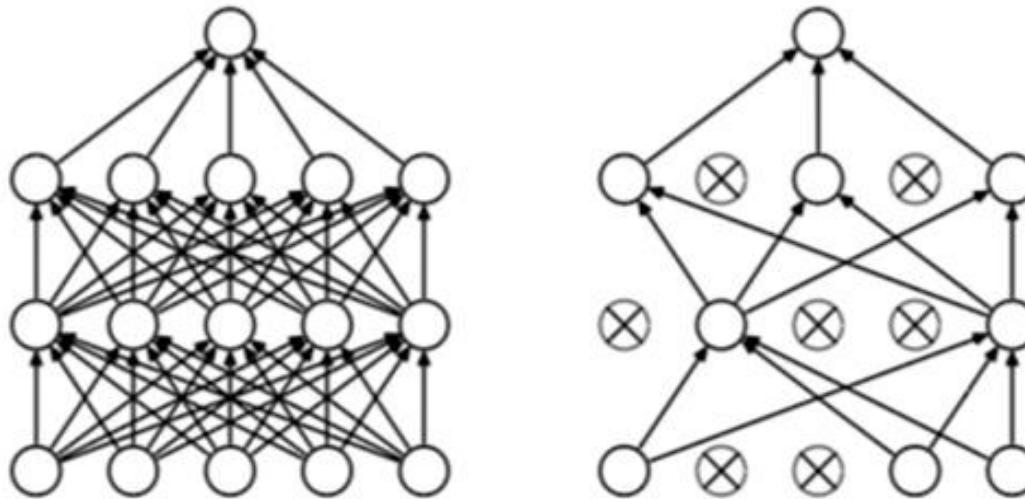


The effects of regularization strength: Each neural network above has 20 hidden neurons, but changing the regularization strength makes its final decision regions smoother with a higher regularization.

```
tf.keras.layers.Dense(  
    units, activation=None, use_bias=True,  
    kernel_initializer='glorot_uniform',  
    bias_initializer='zeros', kernel_regularizer=None,  
    bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,  
    bias_constraint=None, **kwargs  
)
```

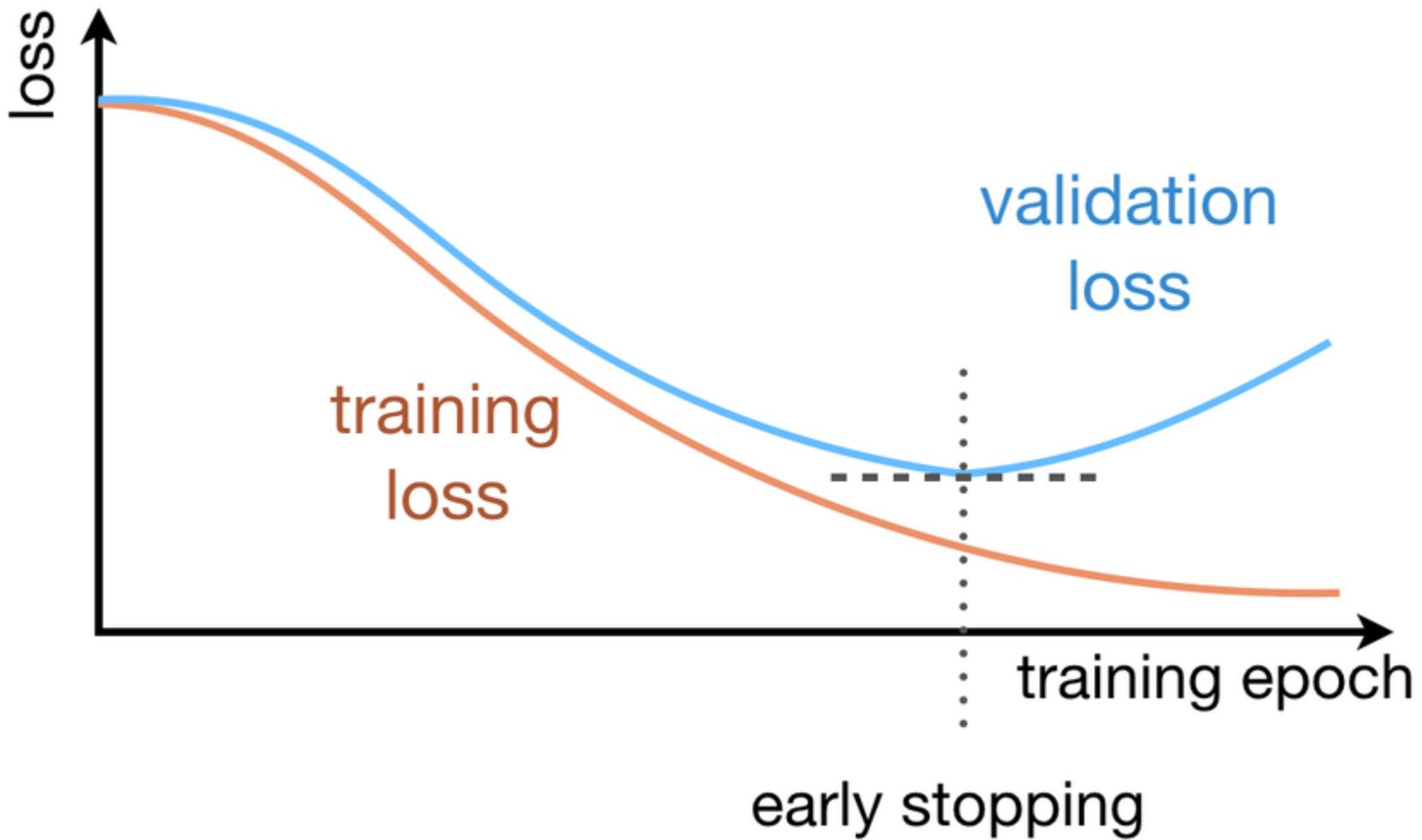
```
tf.keras.regularizers.L1(0.3) # L1 Regularization Penalty  
tf.keras.regularizers.L2(0.1) # L2 Regularization Penalty  
tf.keras.regularizers.L1L2(l1=0.01, l2=0.01) # L1 + L2 penalties
```

# Dropout



```
tf.keras.layers.Dropout(  
    rate, noise_shape=None, seed=None, **kwargs  
)
```

```
model = tf.keras.Sequential([  
    keras.layers.Dense(512, activation='relu', input_shape=(784,)),  
    keras.layers.Dropout(0.2),  
    keras.layers.Dense(10)  
])
```



Filter

Model

Sequential

▶ activations

▶ applications

▶ backend

▼ callbacks

Overview

BackupAndRestore

BaseLogger

CSVLogger

Callback

CallbackList

**EarlyStopping**

History

LambdaCallback

LearningRateScheduler

ModelCheckpoint

ProgbarLogger

ReduceLROnPlateau

RemoteMonitor

SidecarEvaluatorModelExport

TensorBoard

TerminateOnNaN

TensorFlow &gt; API &gt; TensorFlow v2.15.0.post1 &gt; Python

Was this helpful?

# tf.keras.callbacks.EarlyStopping



View source on GitHub

Stop training when a monitored metric has stopped improving.

Inherits From: [Callback](#)

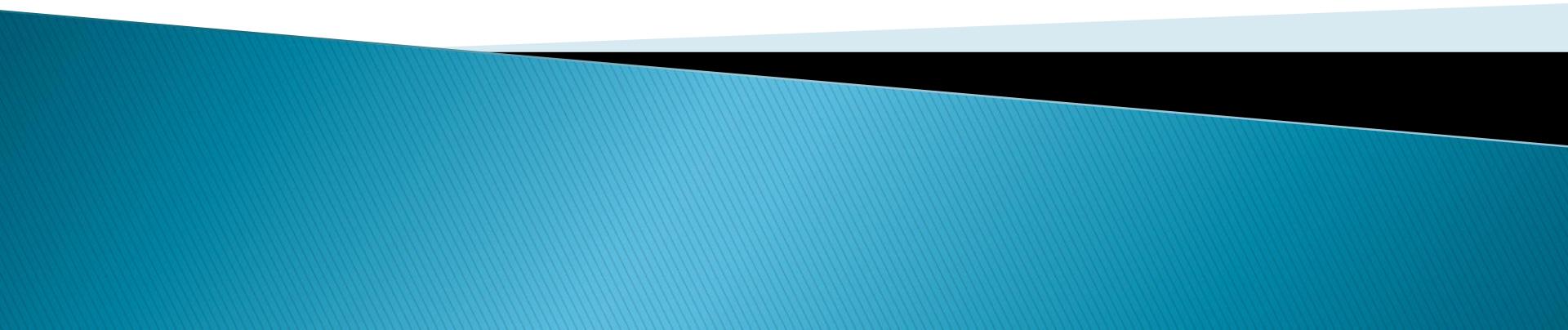
```
tf.keras.callbacks.EarlyStopping(  
    monitor='val_loss',  
    min_delta=0,  
    patience=0,  
    verbose=0,  
    mode='auto',  
    baseline=None,  
    restore_best_weights=False,  
    start_from_epoch=0  
)
```



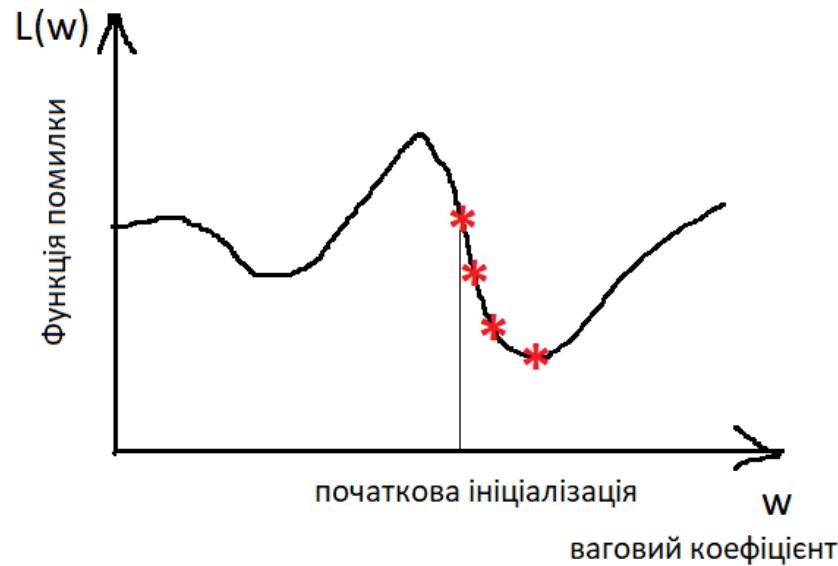
## Example:

```
>>> callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)  
>>> # This callback will stop the training when there is no improvement in  
>>> # the loss for three consecutive epochs.  
>>> model = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])  
>>> model.compile(tf.keras.optimizers.SGD(), loss='mse')  
>>> history = model.fit(np.arange(100).reshape(5, 20), np.zeros(5),  
...                      epochs=10, batch_size=1, callbacks=[callback],  
...                      verbose=0)  
>>> len(history.history['loss']) # Only 4 epochs are run.  
4
```

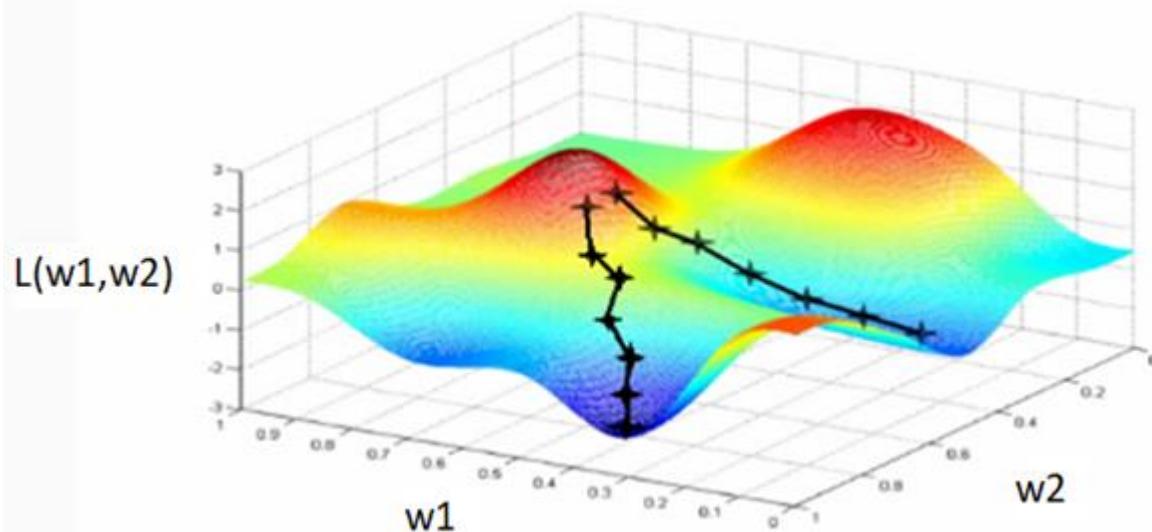
# Learning rate. Optimizers



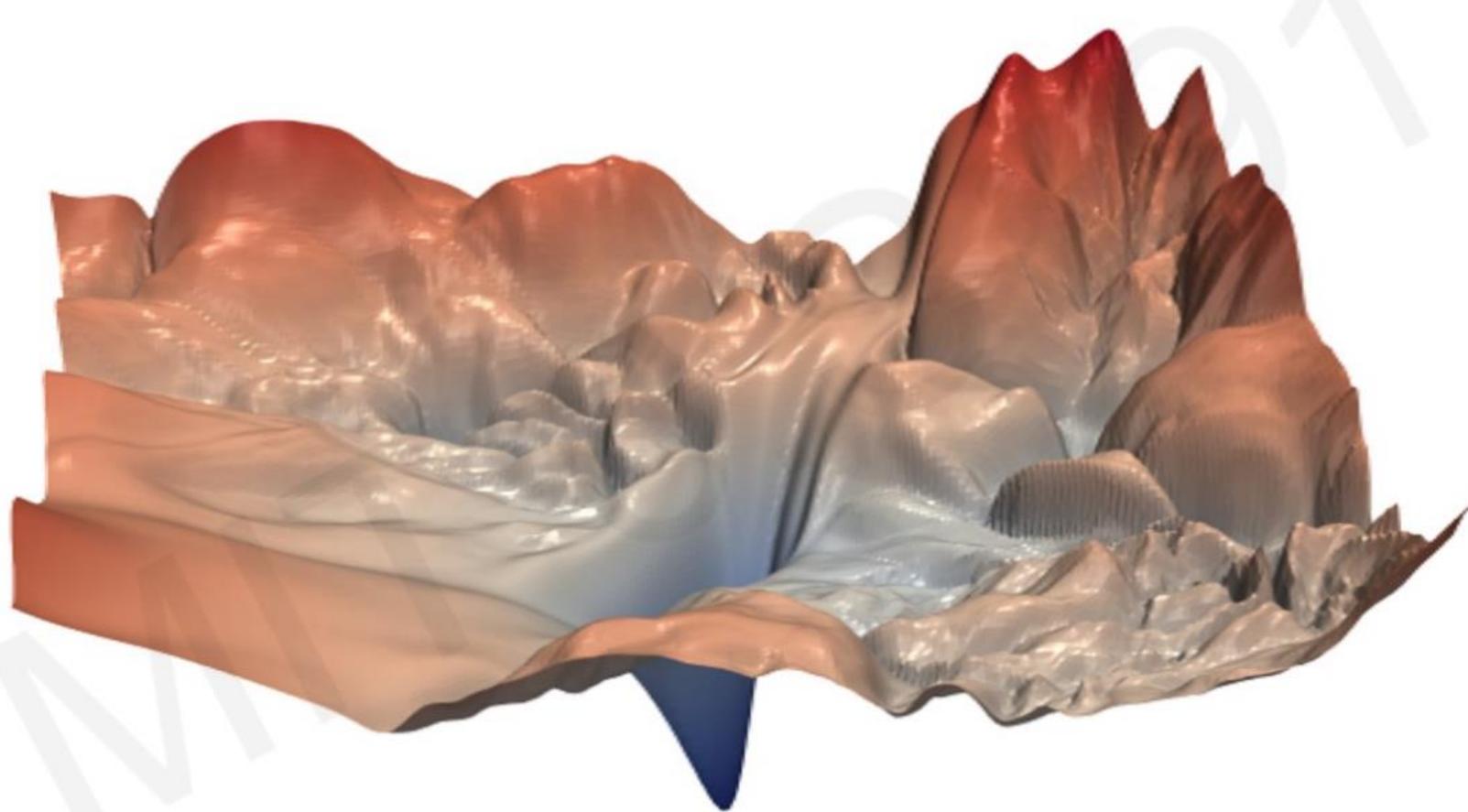
# Градієнтний спуск (Gradient descent)

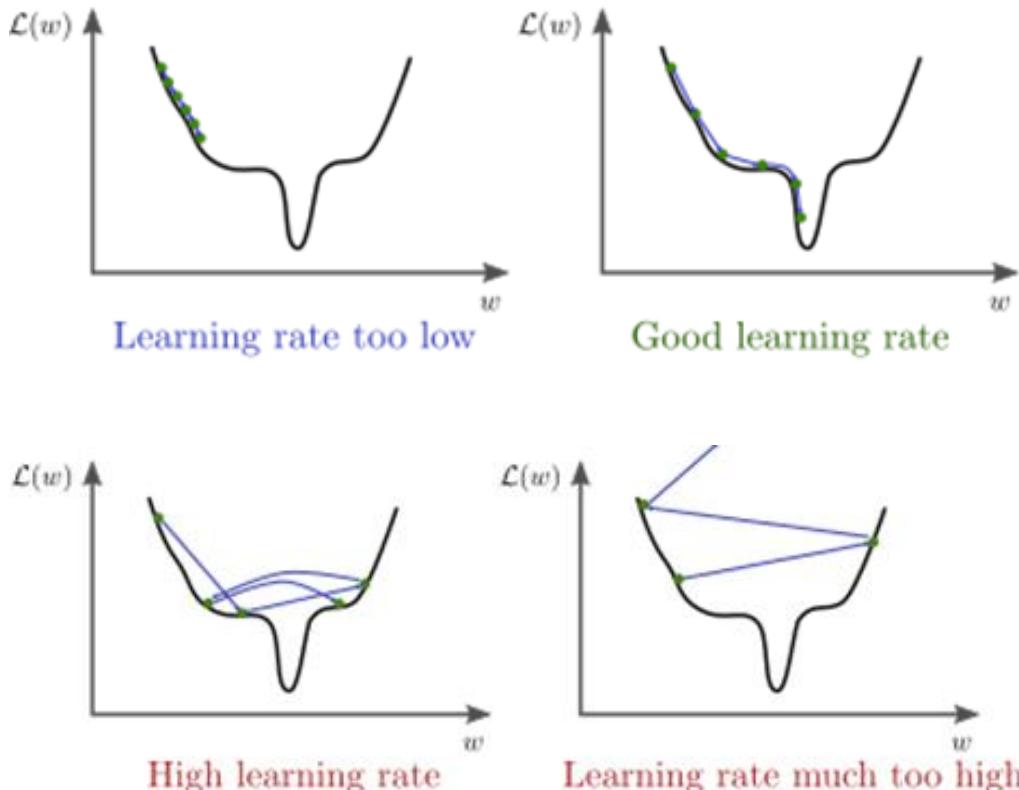
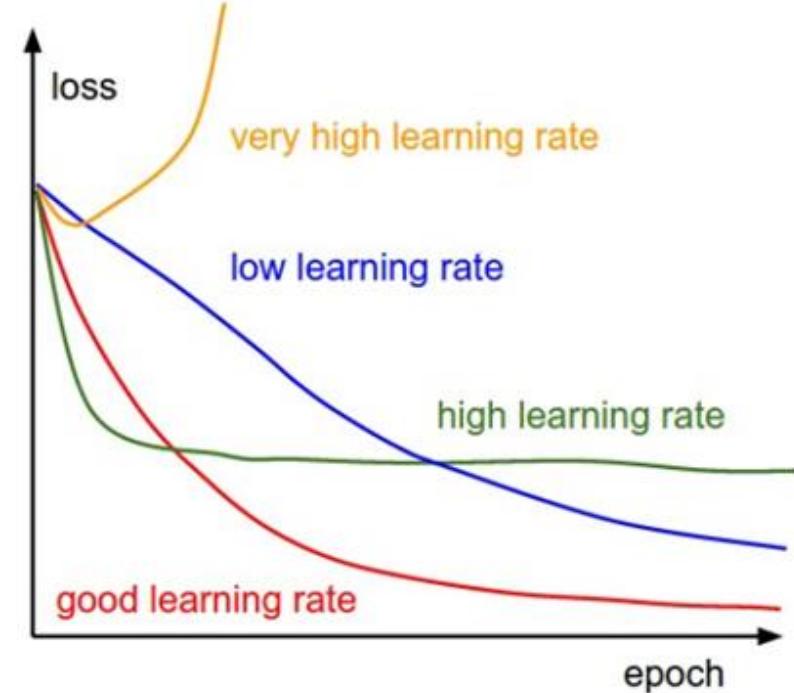


$$\vec{w} = \vec{w} - \eta \vec{\nabla}_w L$$
$$\vec{b} = \vec{b} - \eta \vec{\nabla}_b L$$



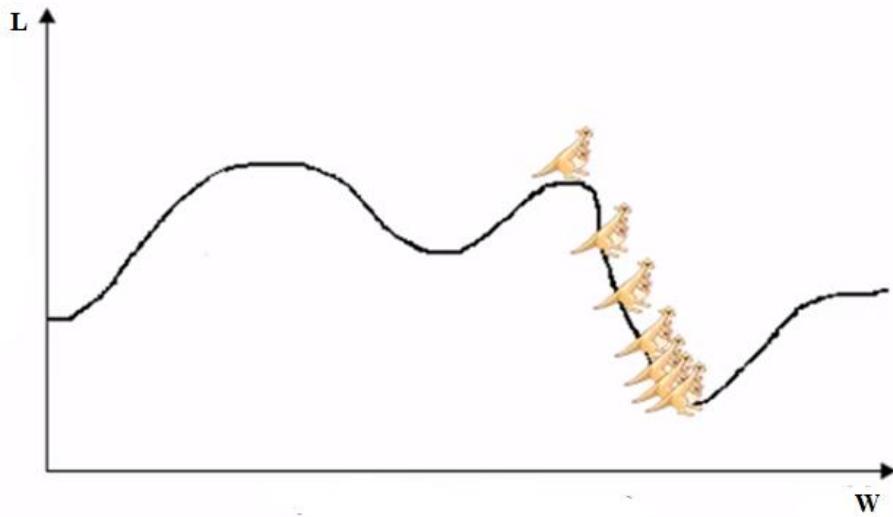
# Visualizing the Loss Landscape of Neural Nets





```
tf.keras.optimizers.SGD(  
    learning_rate=0.01, momentum=0.0, nesterov=False, name='SGD', **kwargs  
)
```

# learning rate schedule



`tf.keras.optimizers.schedules`

[https://www.tensorflow.org/api\\_docs/python/tf/keras/optimizers/schedules](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules)

- ▶ Step decay

Зменшення learning rate  
через кожні декілька епох  
(кожні x епох помножуємо  
на  $\lambda < 1$  )

- ▶ Linear decay:

$$\eta = \eta_0 \left( 1 - \frac{t}{T} \right)$$

- ▶ Exponential decay:

$$\eta = \eta_0 e^{-\frac{t}{T}}.$$

- ▶ Cosine Decay...



Filter

Adam

AdamW

Lion

Optimizer

deserialize

get

serialize

▶ experimental

▶ legacy

▼ schedules

Overview

CosineDecay

CosineDecayRestarts

ExponentialDecay

InverseTimeDecay

LearningRateSchedule

PiecewiseConstantDecay

PolynomialDecay

deserialize

serialize

▶ preprocessing

▶ regularizers

▶ saving

▶ utils

▶ tf.linalg

▶ tf.lite

▶ tf.lookup

▶ tf.math

# Module: tf.keras.optimizers.schedules

See Stable

See Nightly

Public API for `tf.keras.optimizers.schedules` namespace.

View aliases

## Classes

`class CosineDecay`: A LearningRateSchedule that uses a cosine decay schedule.

`class CosineDecayRestarts`: A LearningRateSchedule that uses a cosine decay schedule with restarts.

`class ExponentialDecay`: A LearningRateSchedule that uses an exponential decay schedule.

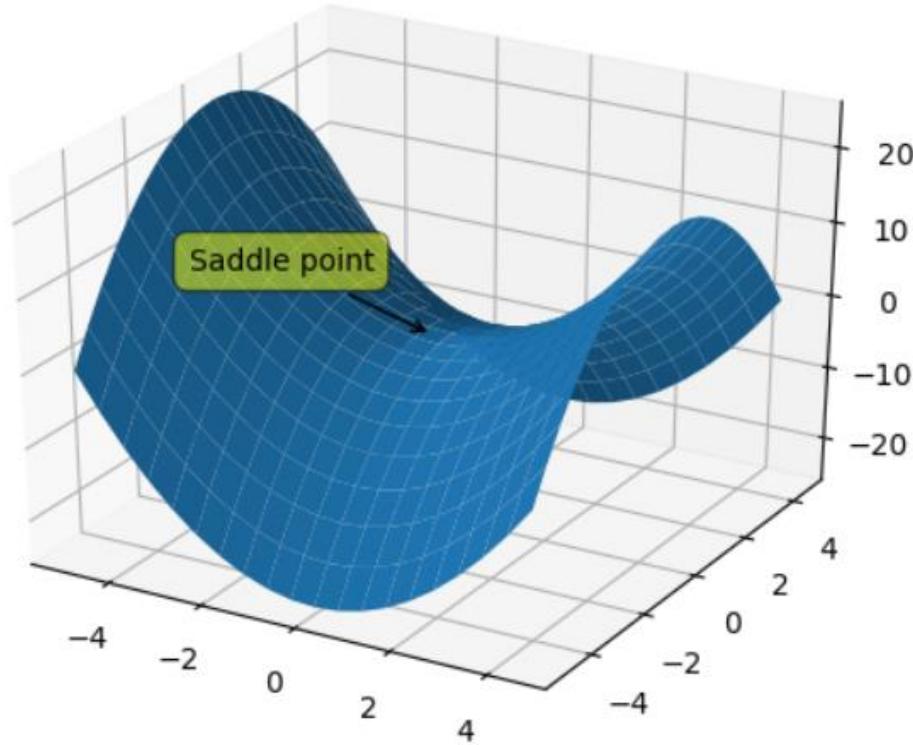
`class InverseTimeDecay`: A LearningRateSchedule that uses an inverse time decay schedule.

`class LearningRateSchedule`: The learning rate schedule base class.

`class PiecewiseConstantDecay`: A LearningRateSchedule that uses a piecewise constant decay schedule.

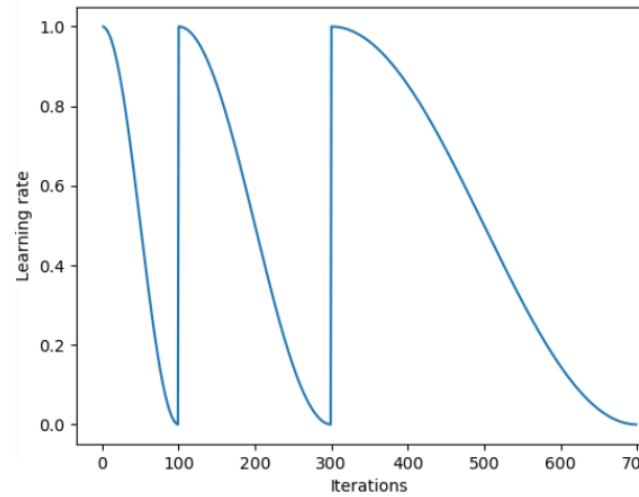
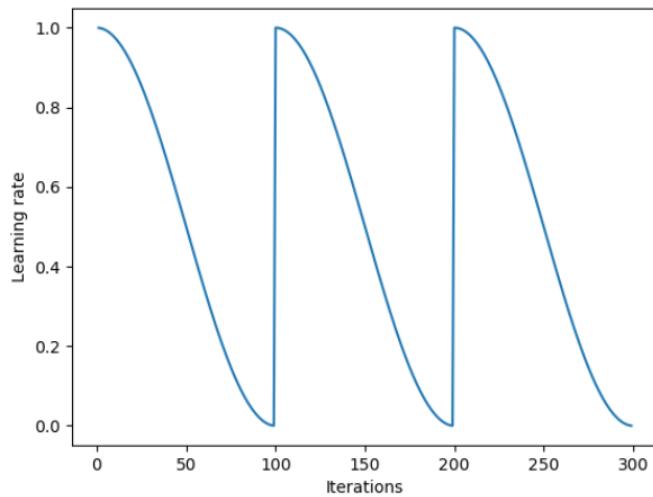
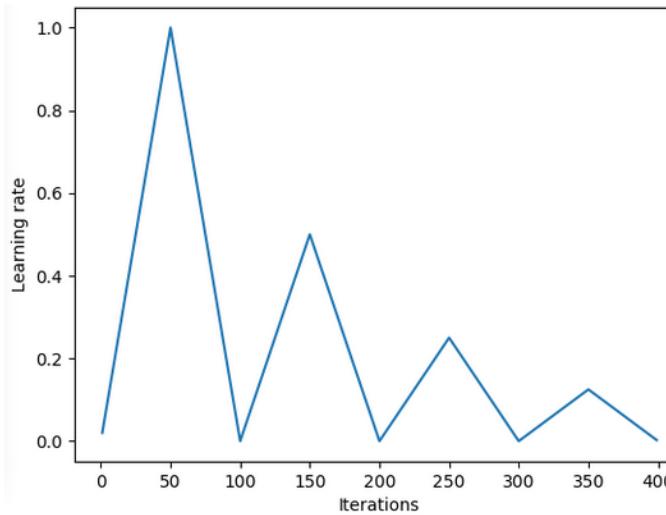
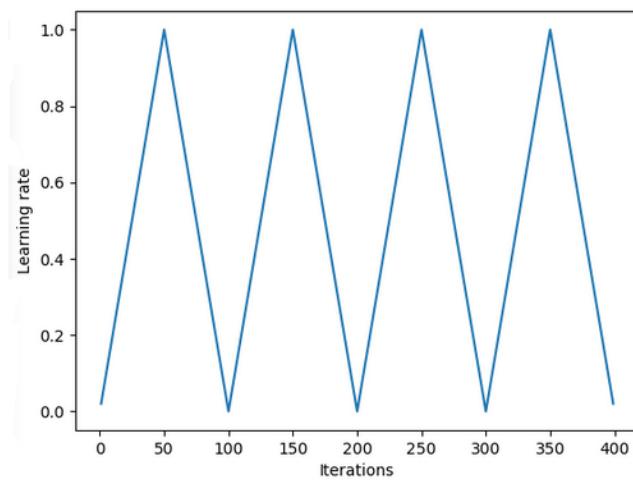
`class PolynomialDecay`: A LearningRateSchedule that uses a polynomial decay schedule.

## Functions



**Сідлова точка** в математичному аналізі – така точка з області визначення функції, яка є стаціонарною для даної функції, однак не є її локальним екстремумом.

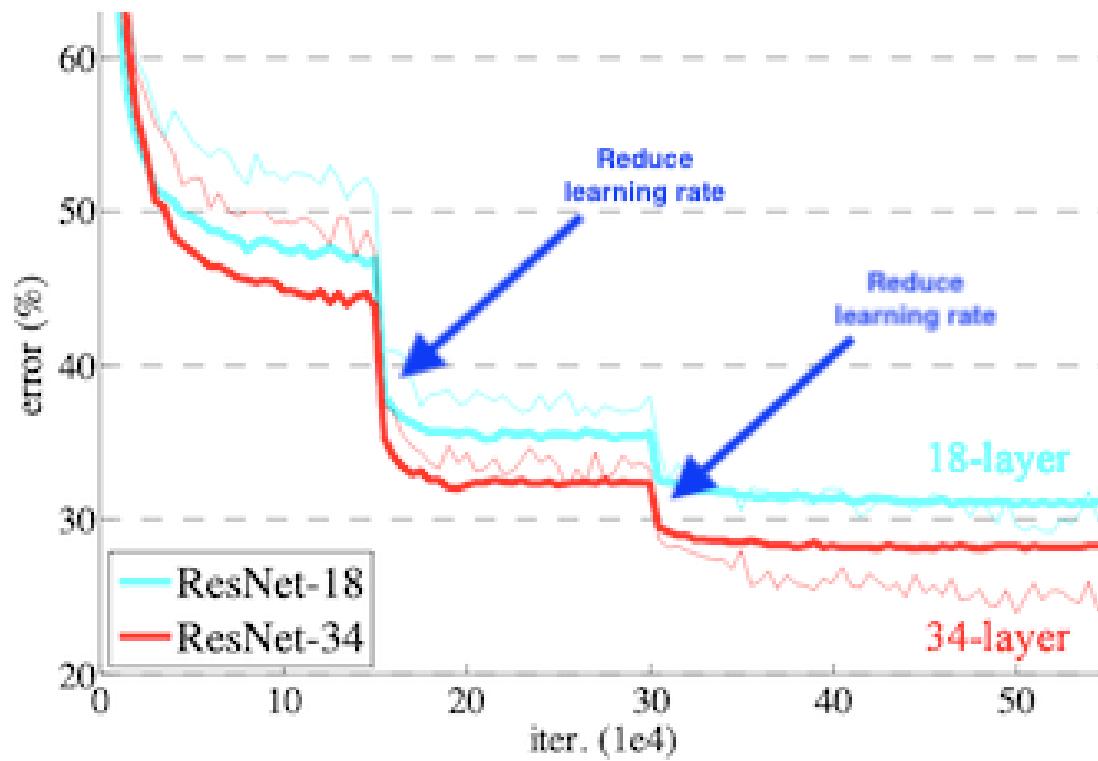
# cycling learning rate



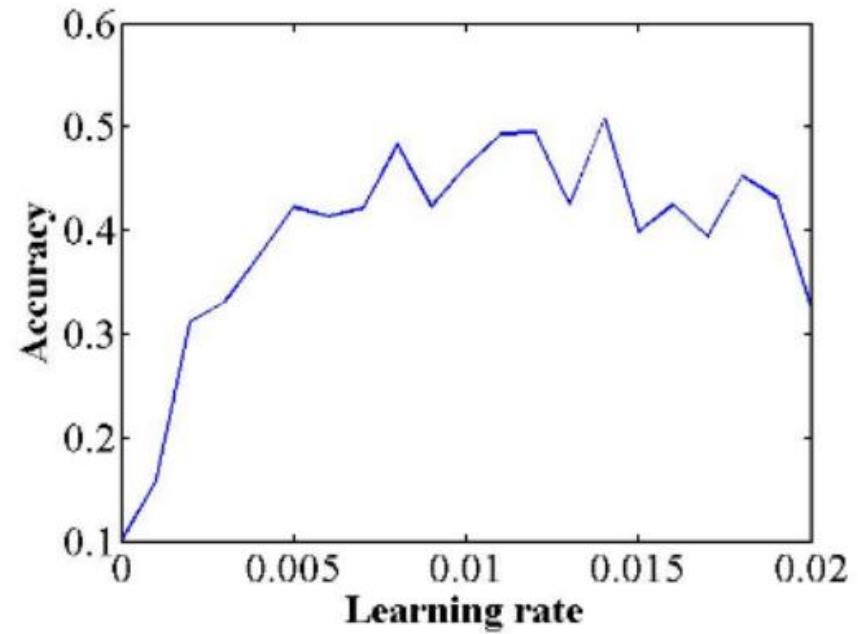
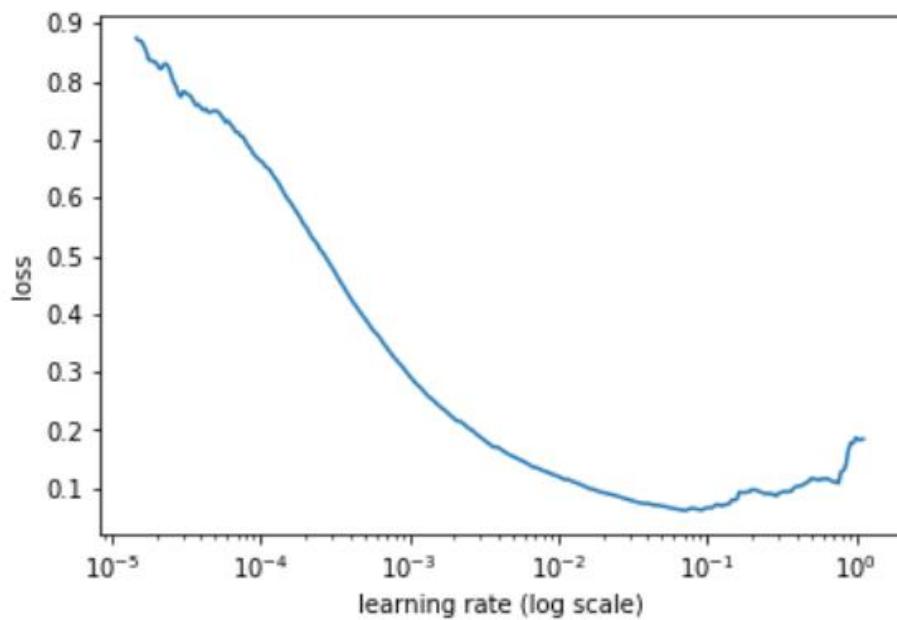
Triangular methods

cosine annealing

# Reduce learning rate on plateau



# Picking the right value for learning rate

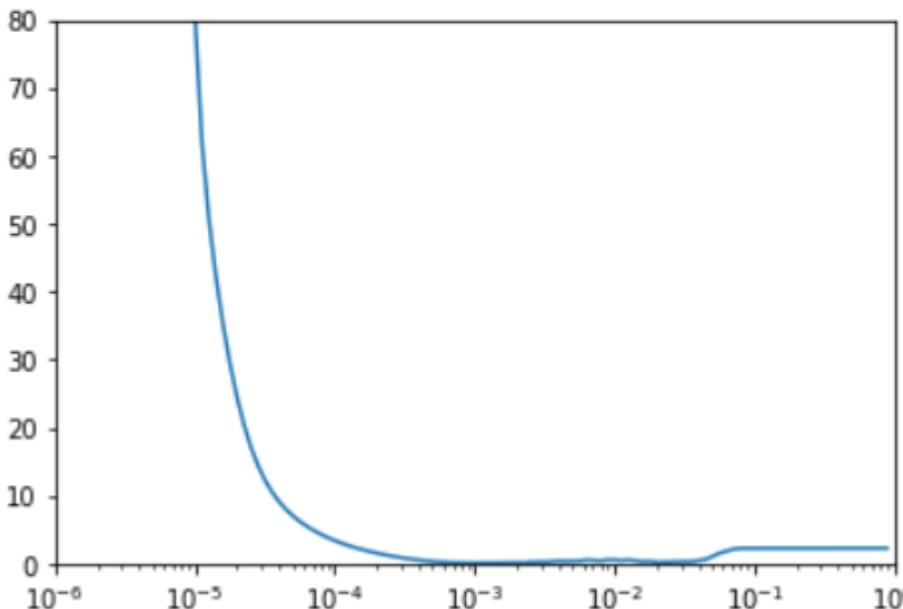


```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-6),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=[ 'accuracy'])

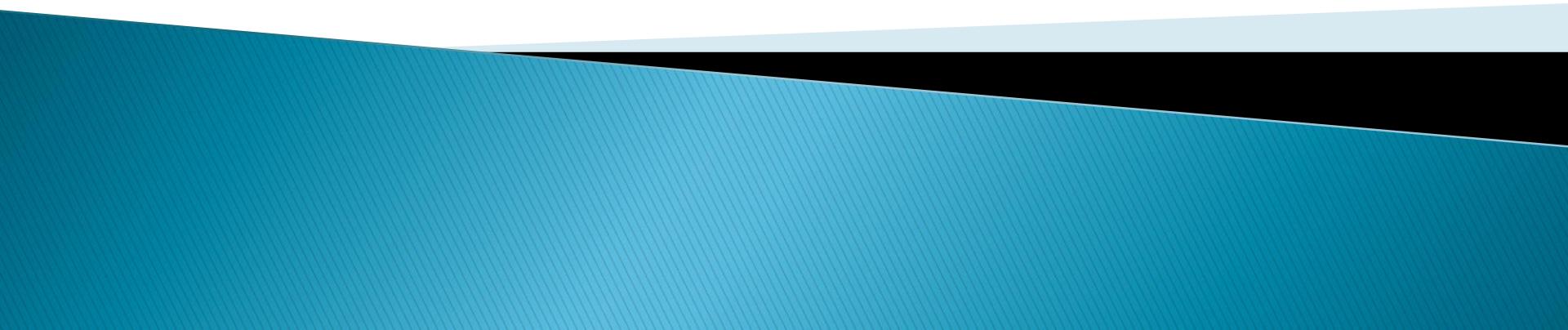
lr_schedule = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-6 * 10 ** (epoch / 20))
history = model.fit(ds_train_b,
                     epochs=100,
                     validation_data=ds_val_b, callbacks=[lr_schedule])
```

```
lrs = 1e-5 * 10 ** (np.arange(100) / 20)
plt.semilogx(lrs, history.history['loss'])
plt.axis([1e-6, 1, 0, 80])
```

(1e-06, 1.0, 0.0, 80.0)



# **Адаптивні варіанти градієнтного спуску та метод моментів**



# Module: tf.keras.optimizers



TensorFlow 1 version

Public API for `tf.keras.optimizers` namespace.

## Modules

`schedules` module: Public API for `tf.keras.optimizers.schedules` namespace.

## Classes

`class Adadelta`: Optimizer that implements the Adadelta algorithm.

`class Adagrad`: Optimizer that implements the Adagrad algorithm.

`class Adam`: Optimizer that implements the Adam algorithm.

`class Adamax`: Optimizer that implements the Adamax algorithm.

`class Ftrl`: Optimizer that implements the FTRL algorithm.

`class Nadam`: Optimizer that implements the NAdam algorithm.

`class Optimizer`: Base class for Keras optimizers.

`class RMSprop`: Optimizer that implements the RMSprop algorithm.

`class SGD`: Gradient descent (with momentum) optimizer.

# Введемо такі позначення:

$\theta_t$

– значення параметрів (вагових коефіцієнтів) у момент часу  $t$

$\nabla L(\mathbf{x}_t, \theta_{t-1}, y_t)$  – значення градієнта в точці  $\theta_{t-1}$

$\eta$

– швидкість навчання (learning rate)

Тоді оновлення значень вагових коефіцієнтів класичним методом SGD можна записати у такому вигляді:

$$\theta_t = \theta_{t-1} - \eta \nabla L(\mathbf{x}_t, \theta_{t-1}, y_t)$$

# Метод моментів

Ідея методу: розглядаємо рух по поверхні функції втрат як рух матеріальної точки, зберігаючи частину швидкості.

Тобто замість того щоб рухатися строго в напрямку градієнта в конкретній точці, ми намагаємося продовжити рух в тому ж напрямку, в якому рухалися раніше + враховуємо значення градієнта. Звідси і назва методу: у нашої «матеріальної точки», яка спускається по поверхні, з'являється імпульс, вона рухається по інерції і прагне цей імпульс зберегти.

$$u_t = \gamma u_{t-1} + \eta \nabla_{\theta} L(\theta),$$
$$\theta = \theta - u_t.$$

```
tf.keras.optimizers.SGD(  
    learning_rate=0.01, momentum=0.0, nesterov=False, name='SGD', **kwargs  
)
```

# Метод Нестерова

Ідея методу: оскільки застосовуючи метод моментів, ми знаємо, що прямуємо у точку  $\gamma u_{t-1}$ ,

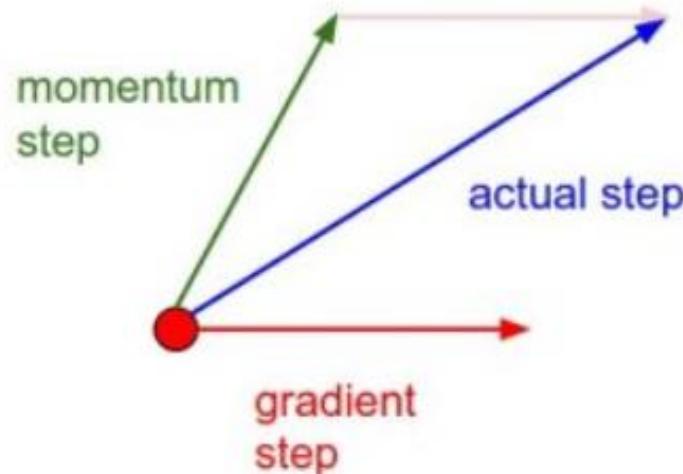
Метод Нестерова пропонує обчислювати градієнт одразу у цій точці

$$u_t = \gamma u_{t-1} + \eta \nabla_{\theta} \mathcal{L}(\theta - \gamma u_{t-1}).$$

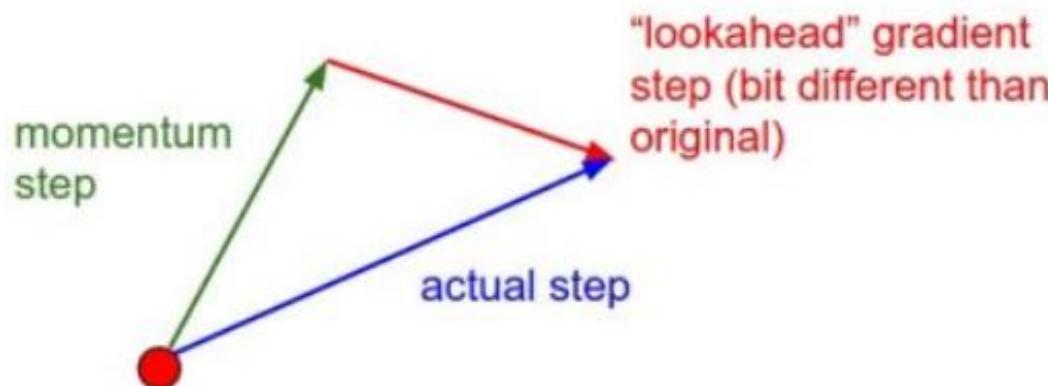
$$\theta = \theta - u_t.$$

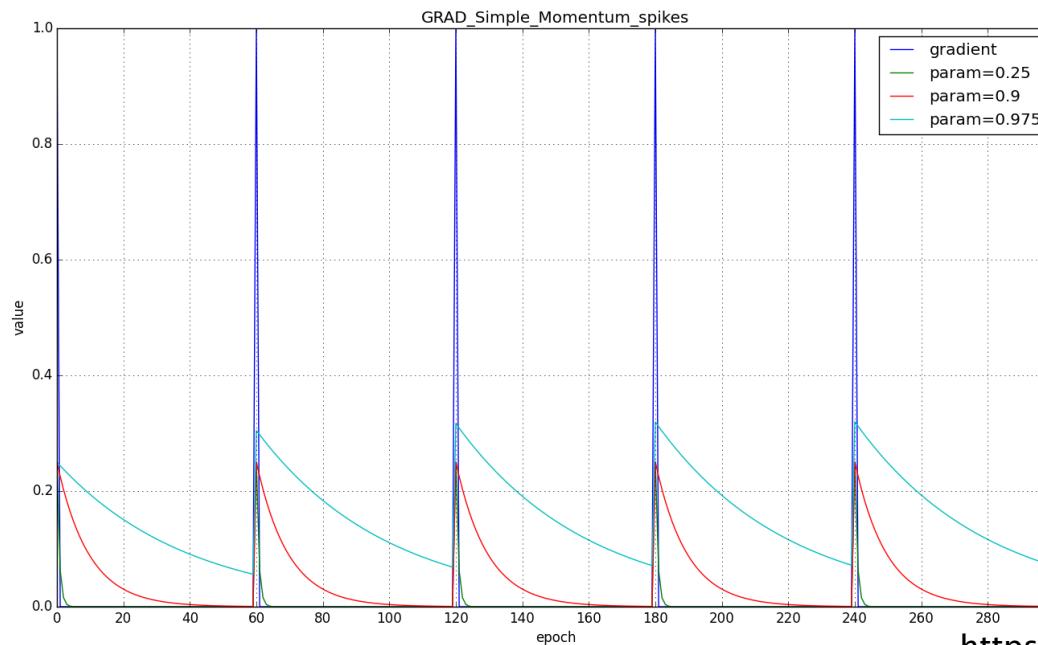
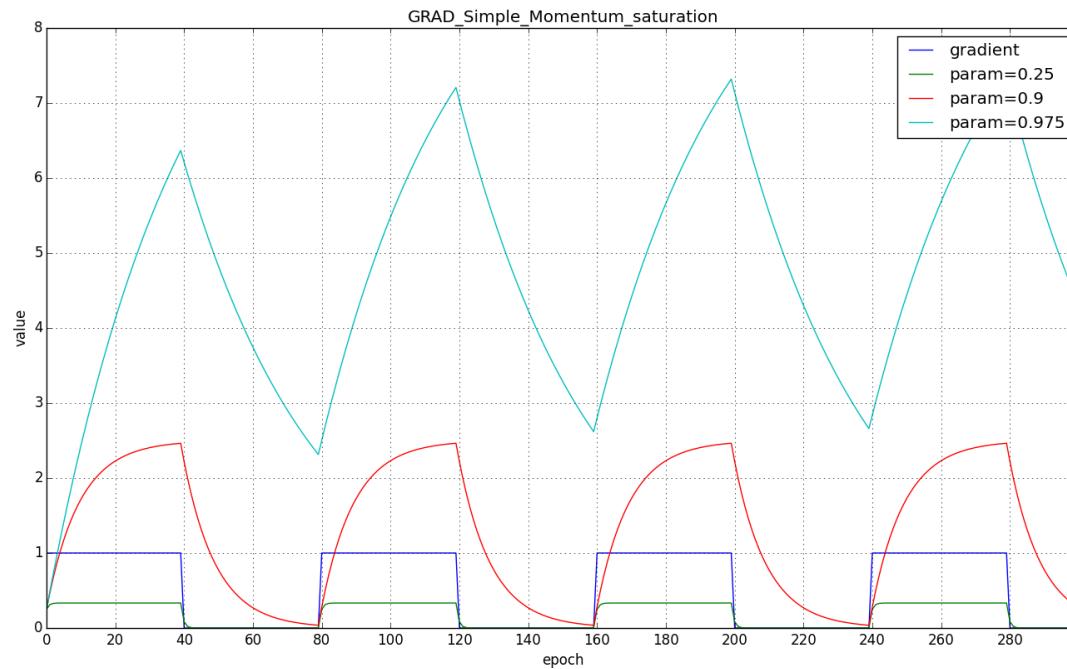
```
tf.keras.optimizers.SGD(  
    learning_rate=0.01, momentum=0.0, nesterov=True, name='SGD', **kwargs  
)
```

## Momentum update

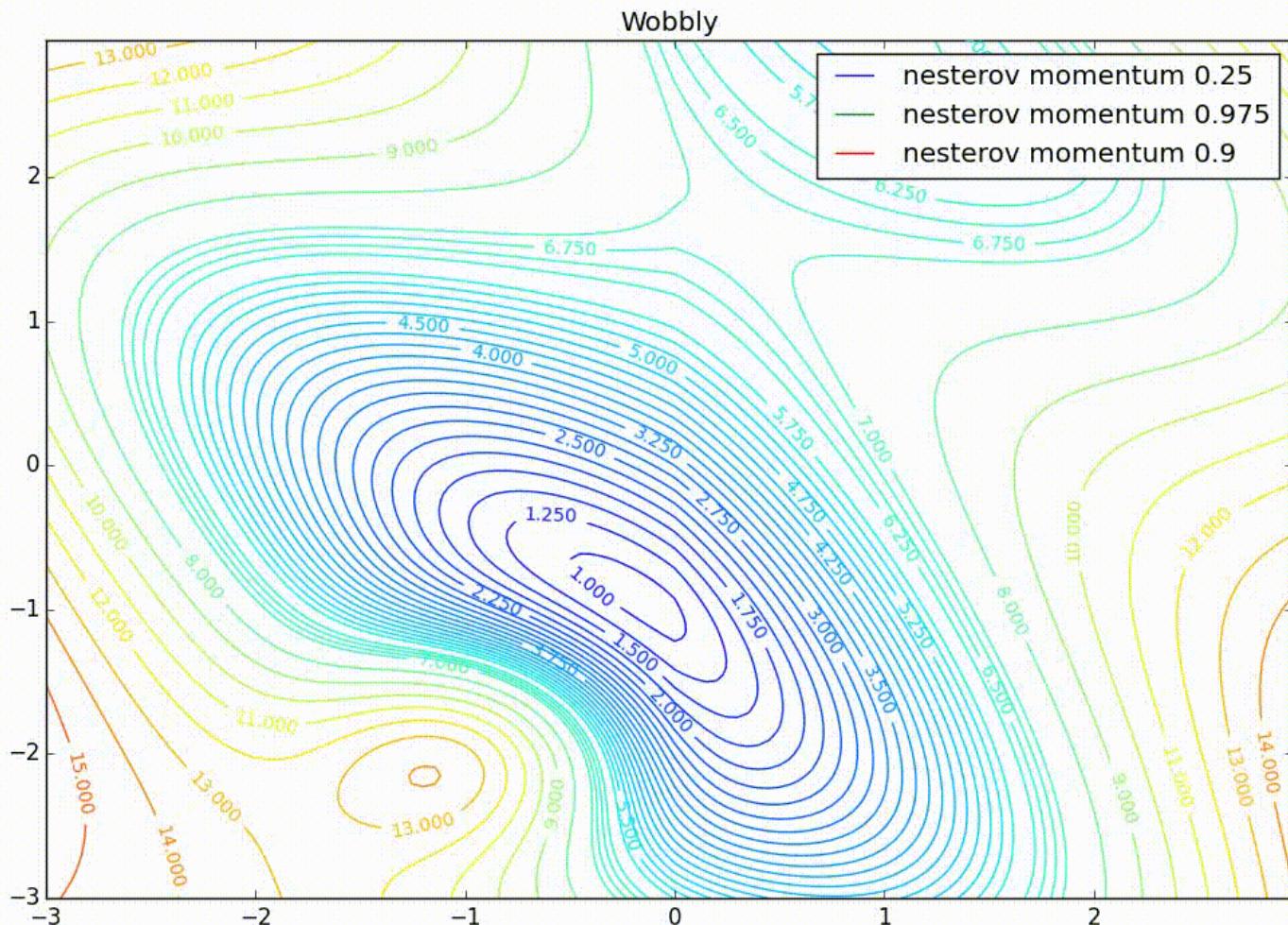


## Nesterov momentum update

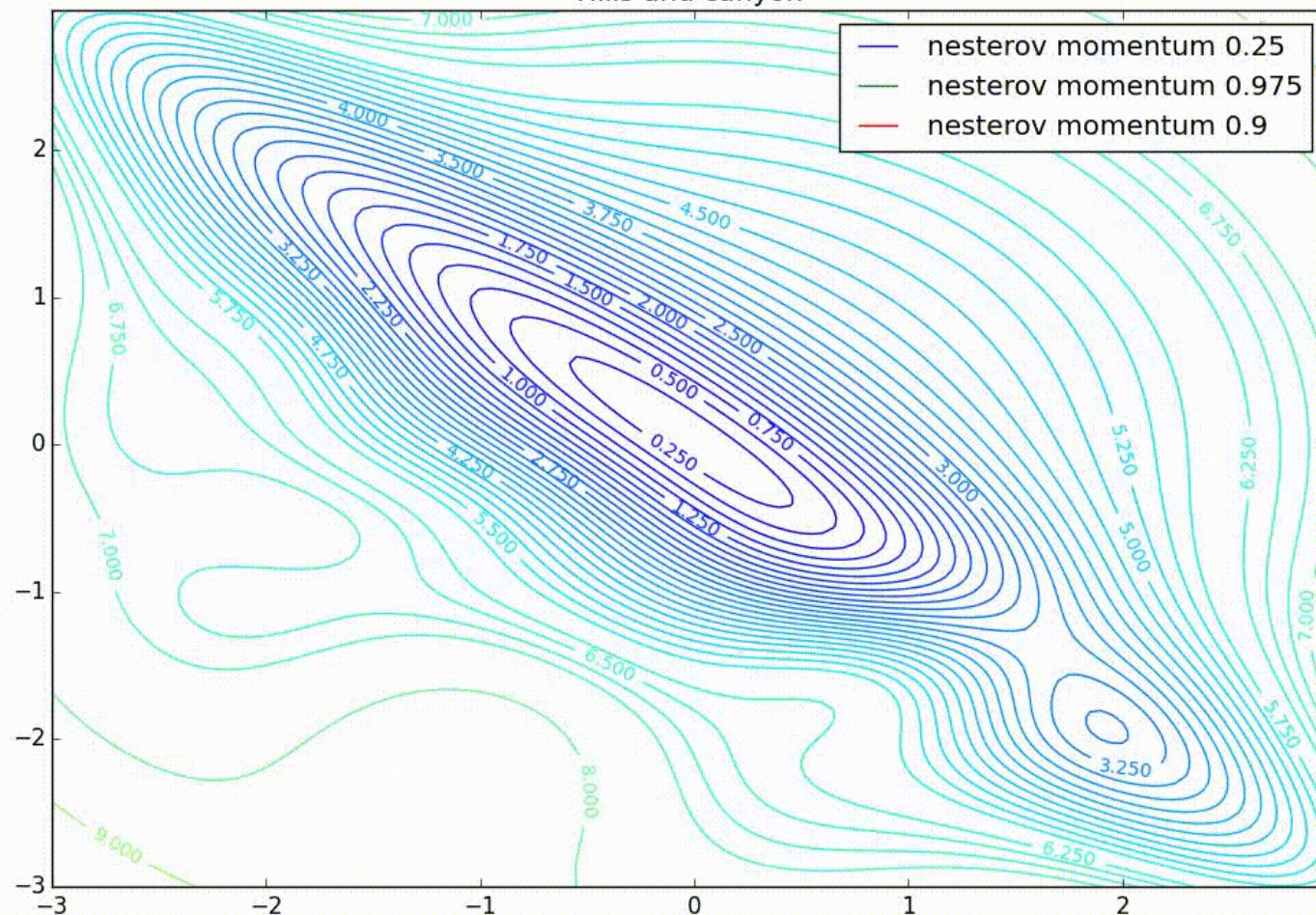




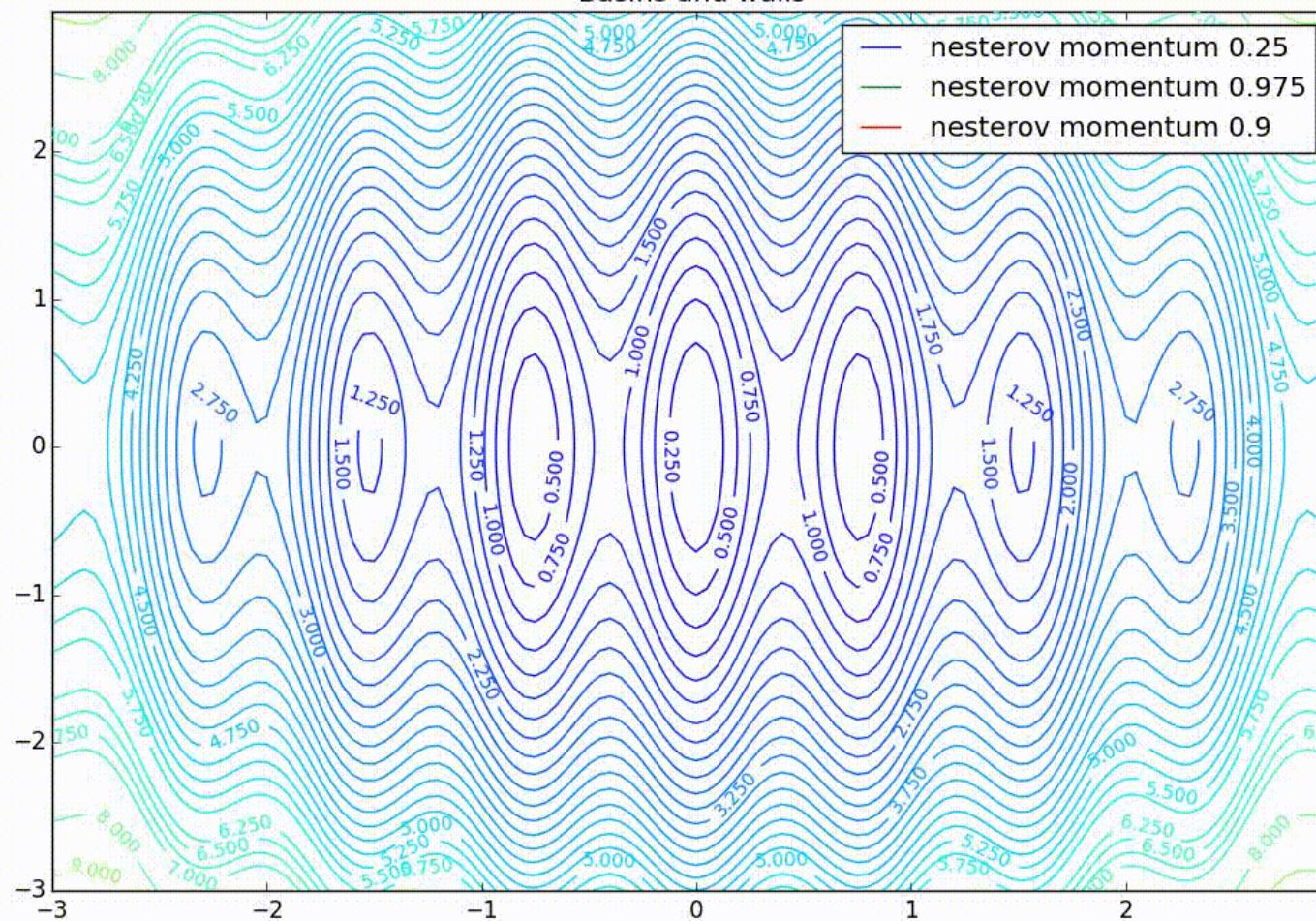
# Експерименти



Hills and canyon



Basins and walls



# Адаптивні методи

У попередніх методах швидкість навчання була однакова для всіх напрямків

Ідея адаптивних методів: давайте рухатися швидше по тим параметрам, які не сильно змінюються, і повільніше - по параметрам, що швидко змінюються

- ▶ Adagrad
- ▶ RMSProp
- ▶ Adadelta
- ▶ Adam
- ▶ ...

# Adagrad (adaptive gradient)

Ефективно перемасштубує крок навчання кожного параметра окремо, враховуючи історію всіх минулих градієнтів цього параметра. Це робиться шляхом поділу кожного елемента у градієнти на квадратний корінь суми квадратів попередніх відповідних елементів градієнта. Також метод зменшує сам крок навчання з часом, так як сума квадратів збільшується з кожною ітерацією.

Позначимо для стисlosti запису:  $g_t \equiv \nabla_{\theta} L(\theta_t)$

$$G_t = G_t + g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

# RMSprop (Root Mean Square propagation)

Проблема попереднього методу полягає в тому, що сума квадратів градієнта постійно збільшується і швидкість навчання іноді зменшується надто швидко.

У цьому методі замість повної суми оновлень обчислюється експоненціальне ковзне середнє

$$v_t = \gamma v_{t-1} + (1 - \gamma)x \quad 0 < \gamma < 1 \text{ - коефіцієнт збереження}$$

Позначимо  $E[g^2]_t$  - ковзне середнє в момент  $t$

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g_t^2 \quad \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Позначимо  $RMS[g]_t = \sqrt{E[g^2]_t + \epsilon}$

$$\theta_{t+1} = \theta_t - \frac{\eta}{RMS[g]_t} g_t$$

# Adadelta

$$\theta_{t+1} = \theta_t - \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma) \Delta\theta_t^2$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

# Adam (Adaptive Moment Estimation)

Метод поєднує у собі кращі ідеї як методу моментів, так і адаптивних методів. Найчастіше застосовується на практиці та потребує мінімальне налаштування параметрів.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

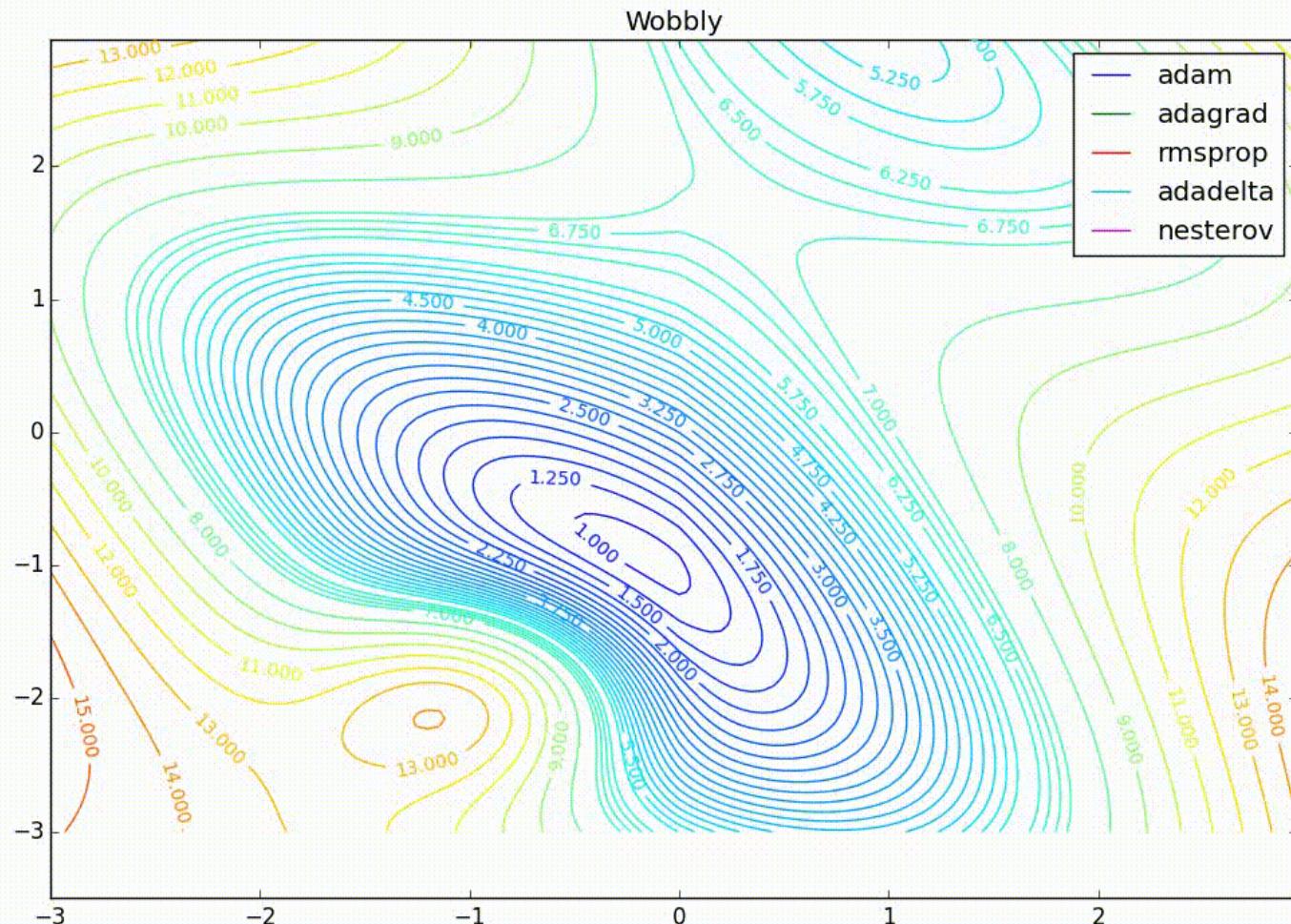
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

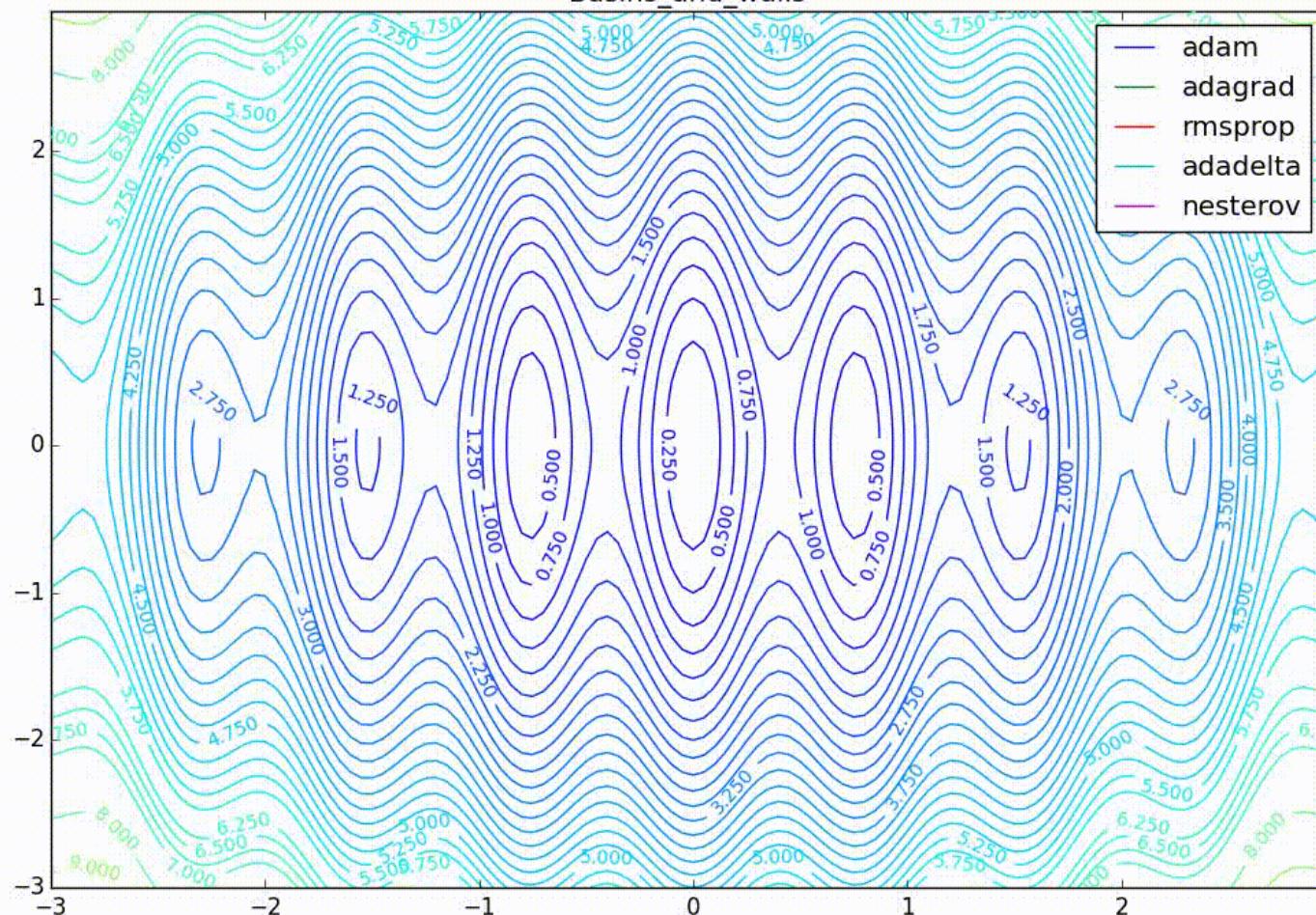
Автори методу пропонують обирати значення параметрів такими:

$$\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8} \quad \eta = 0.001$$

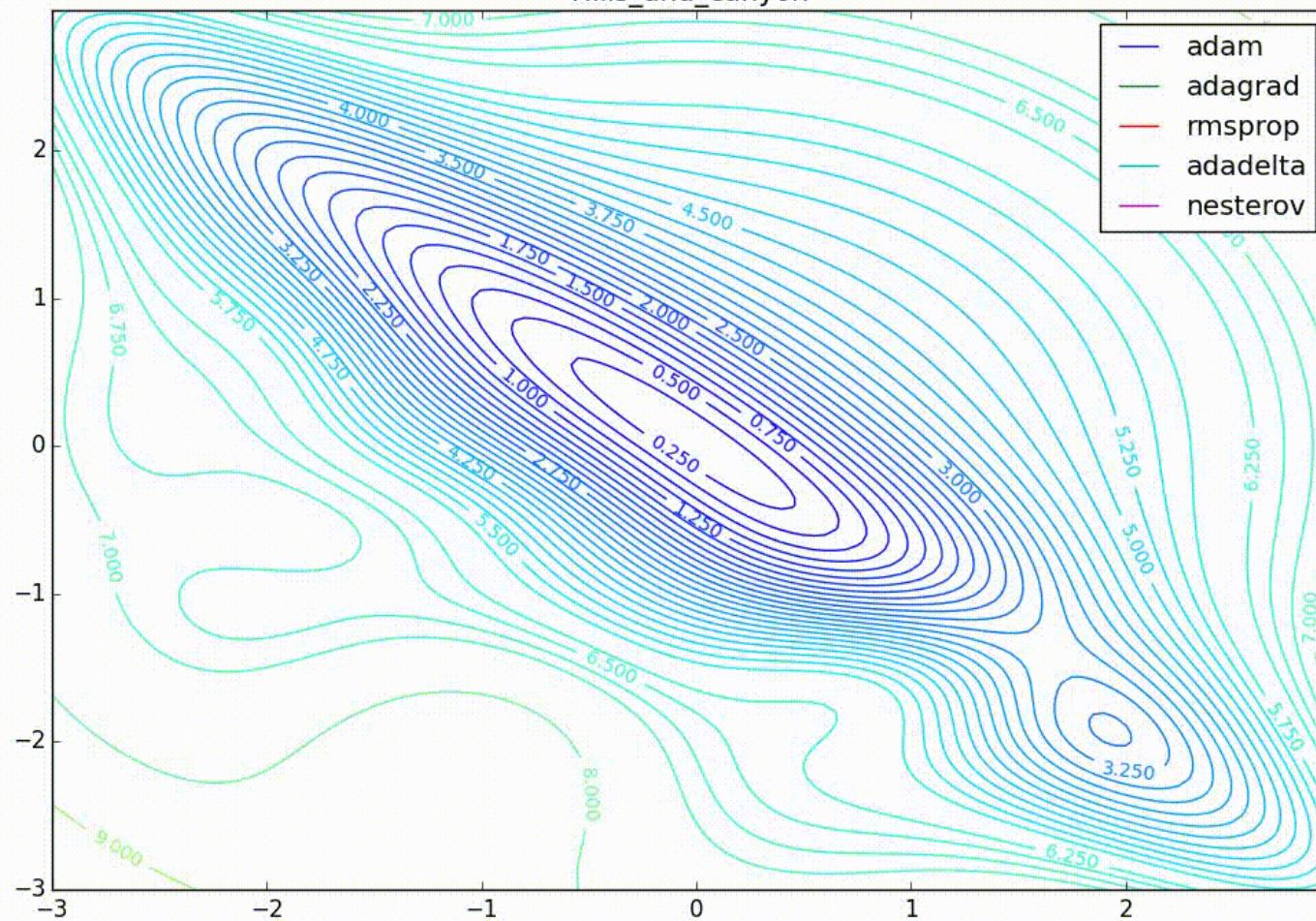
# Експерименти



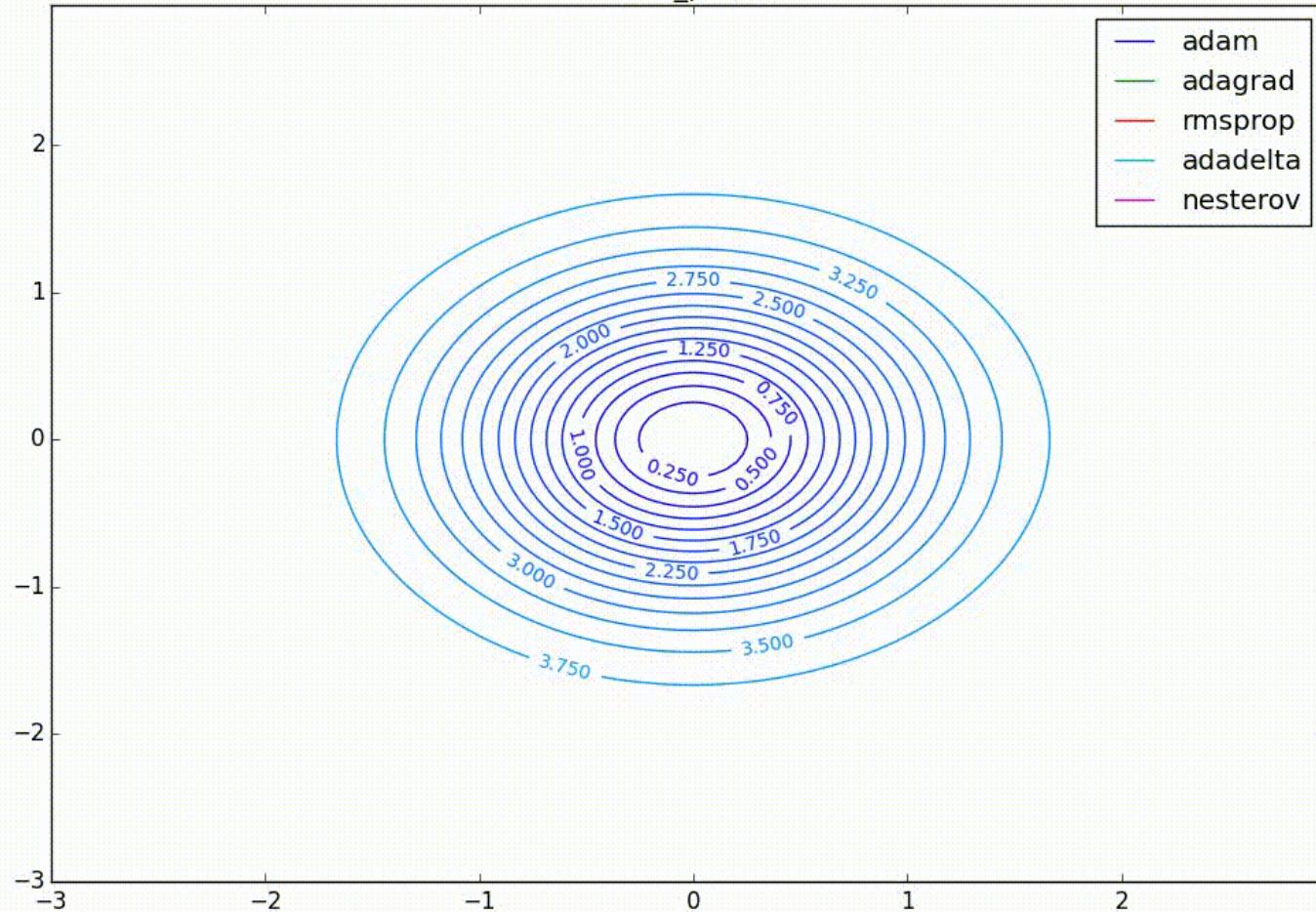
Basins\_and\_walls



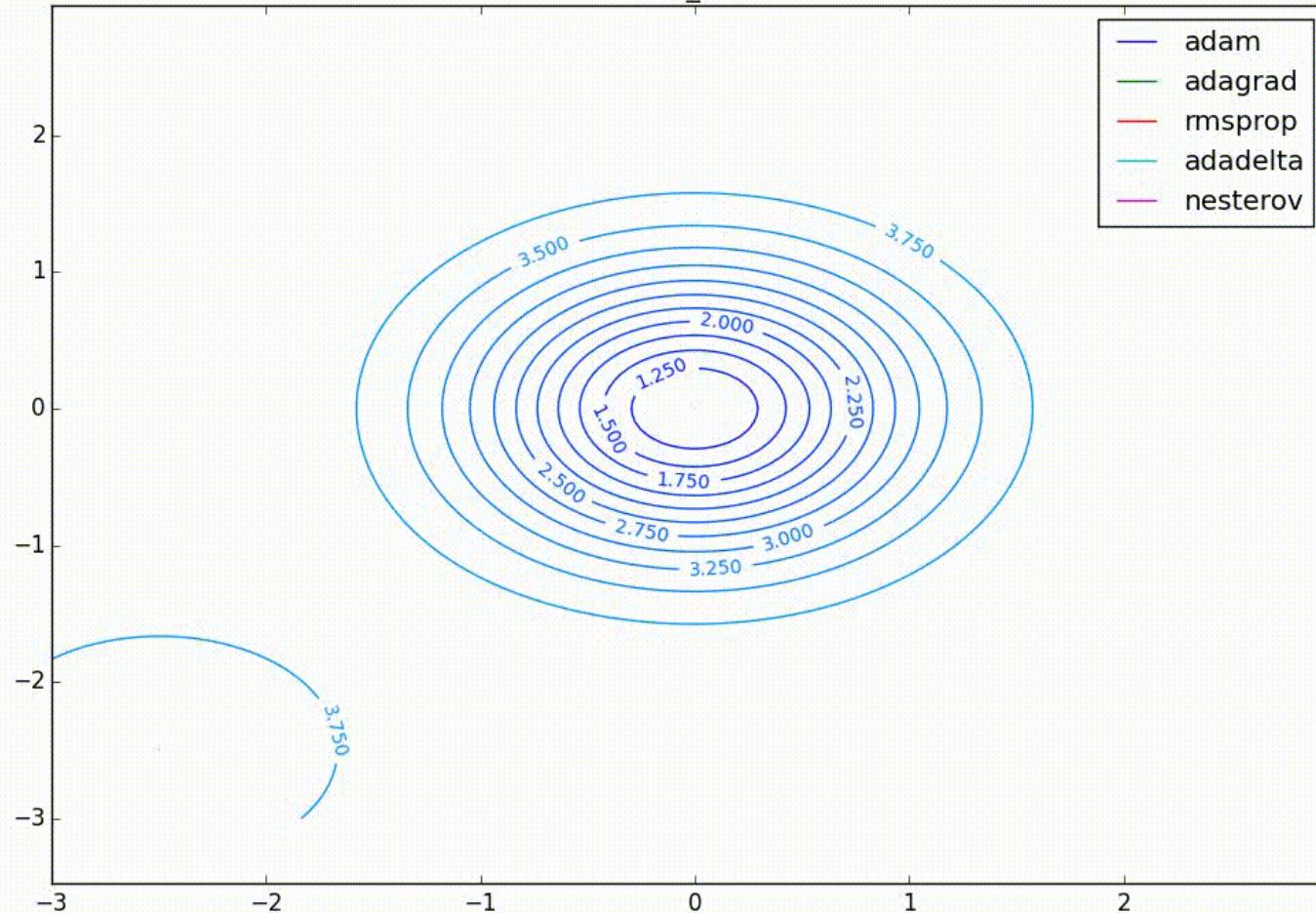
Hills\_and\_canyon



Giant\_plateau



Bad\_init



# Ініціалізація вагових коефіцієнтів

Xavier Glorot ініціалізація

$$w_i \sim U \left[ -\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}} \right]$$

для симетричних функцій активації

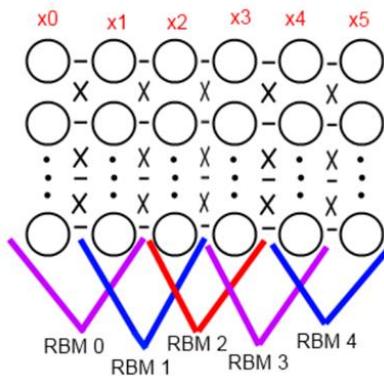
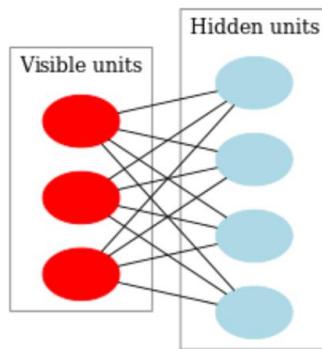
Ініціалізація Хе

$$w_i \sim \mathcal{N} \left( 0, \sqrt{\frac{2}{n}} \right)$$

для *ReLU*

bias = 0

Restricted Boltzmann machine



<https://proceedings.mlr.press/v9/glorot10a.html>

<https://arxiv.org/abs/1502.01852>

## Module: tf.keras.initializers

### Classes

```
class Constant: Initializer that generates tensors with constant values.  
class GlorotNormal: The Glorot normal initializer, also called Xavier normal initializer.  
class GlorotUniform: The Glorot uniform initializer, also called Xavier uniform initializer.  
class HeNormal: He normal initializer.  
class HeUniform: He uniform variance scaling initializer.  
class Identity: Initializer that generates the identity matrix.  
class Initializer: Initializer base class: all Keras initializers inherit from this class.  
class LecunNormal: Lecun normal initializer.  
class LecunUniform: Lecun uniform initializer.  
class Ones: Initializer that generates tensors initialized to 1.  
class Orthogonal: Initializer that generates an orthogonal matrix.  
class RandomNormal: Initializer that generates tensors with a normal distribution.  
class RandomUniform: Initializer that generates tensors with a uniform distribution.  
class TruncatedNormal: Initializer that generates a truncated normal distribution.  
class VarianceScaling: Initializer capable of adapting its scale to the shape of weights tensors.  
class Zeros: Initializer that generates tensors initialized to 0.  
  
class constant: Initializer that generates tensors with constant values.  
class glorot_normal: The Glorot normal initializer, also called Xavier normal initializer.  
class glorot_uniform: The Glorot uniform initializer, also called Xavier uniform initializer.  
class he_normal: He normal initializer.  
class he_uniform: He uniform variance scaling initializer.  
class identity: Initializer that generates the identity matrix.  
class lecun_normal: Lecun normal initializer.  
class lecun_uniform: Lecun uniform initializer.  
class ones: Initializer that generates tensors initialized to 1.  
class orthogonal: Initializer that generates an orthogonal matrix.  
class random_normal: Initializer that generates tensors with a normal distribution.  
class random_uniform: Initializer that generates tensors with a uniform distribution.  
class truncated_normal: Initializer that generates a truncated normal distribution.  
class variance_scaling: Initializer capable of adapting its scale to the shape of weights tensors.  
class zeros: Initializer that generates tensors initialized to 0.
```

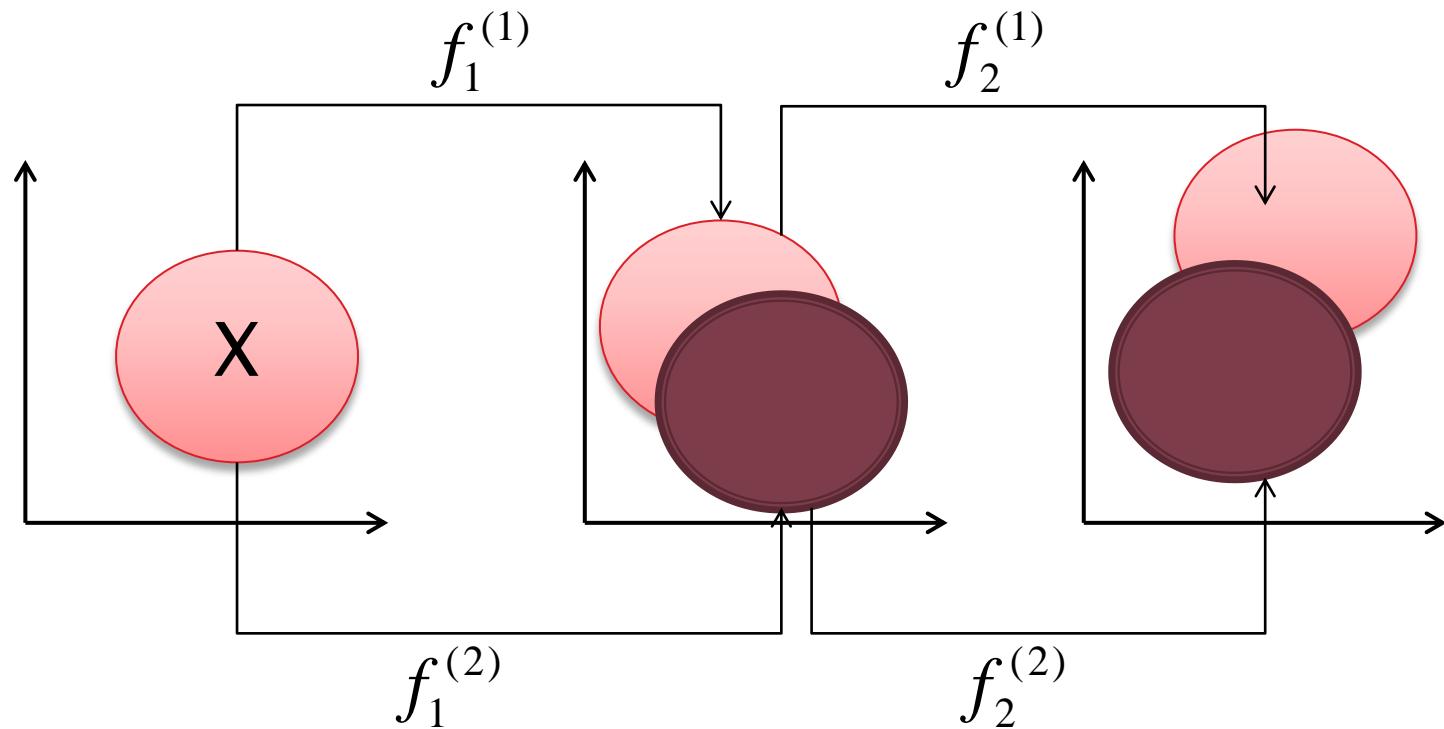
## tf.keras.initializers.HeNormal

```
tf.keras.initializers.HeNormal(  
    seed=None  
)
```

```
# Usage in a Keras layer:  
initializer = tf.keras.initializers.HeNormal()  
layer = tf.keras.layers.Dense(3, kernel_initializer=initializer)
```

# Batch Normalization

# Problem: internal covariate shift



**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots m\}$ ;

Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

```
layer = keras.layers.BatchNormalization()
```

