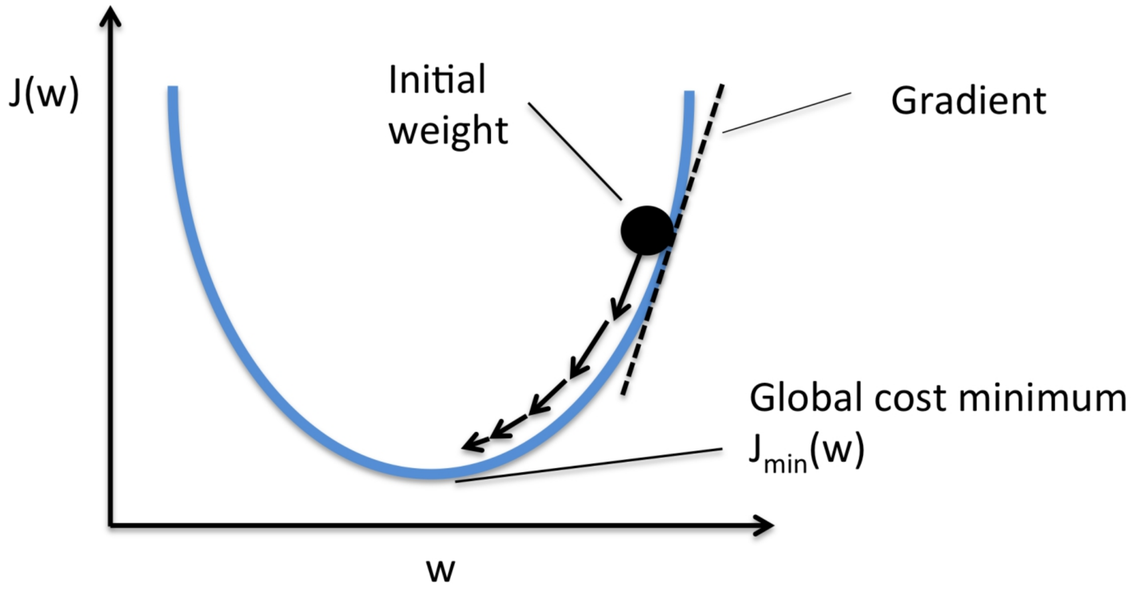
hillel
IT school

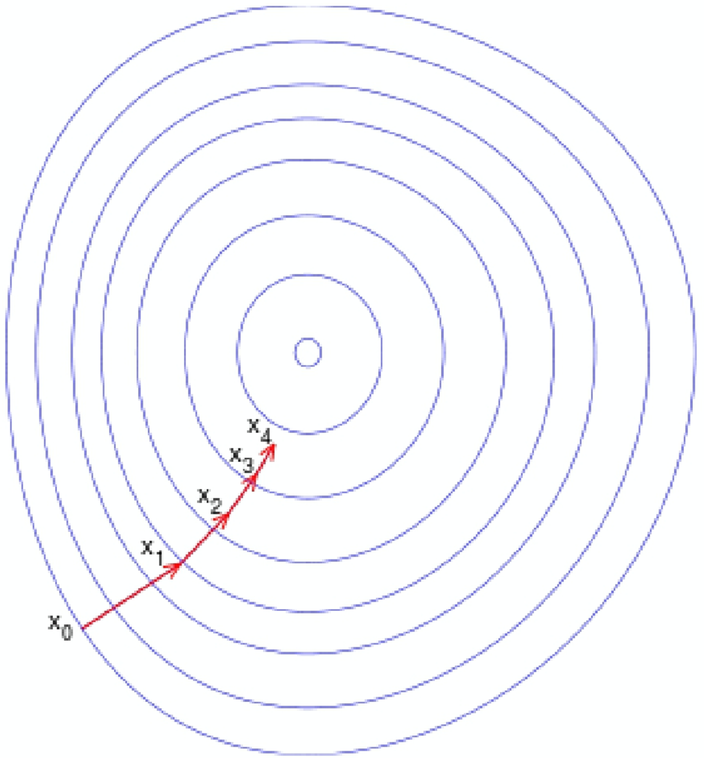
Lesson 11
Deep Learning: Optimization

Gradient Descent

Повторювати до зближення:

$$\omega \leftarrow \omega - \alpha \frac{\partial \text{Loss}}{\partial \omega}$$





Gradient Descent

Плюси:
- Може знайти мінімуми

Мінуси:
- Потрібно розрахувати градієнт на всьому наборі даних
- Може застрягти на плато або неглибоких локальних мінімумах

Stochastic Gradient Descent

Batch Gradient Descent:

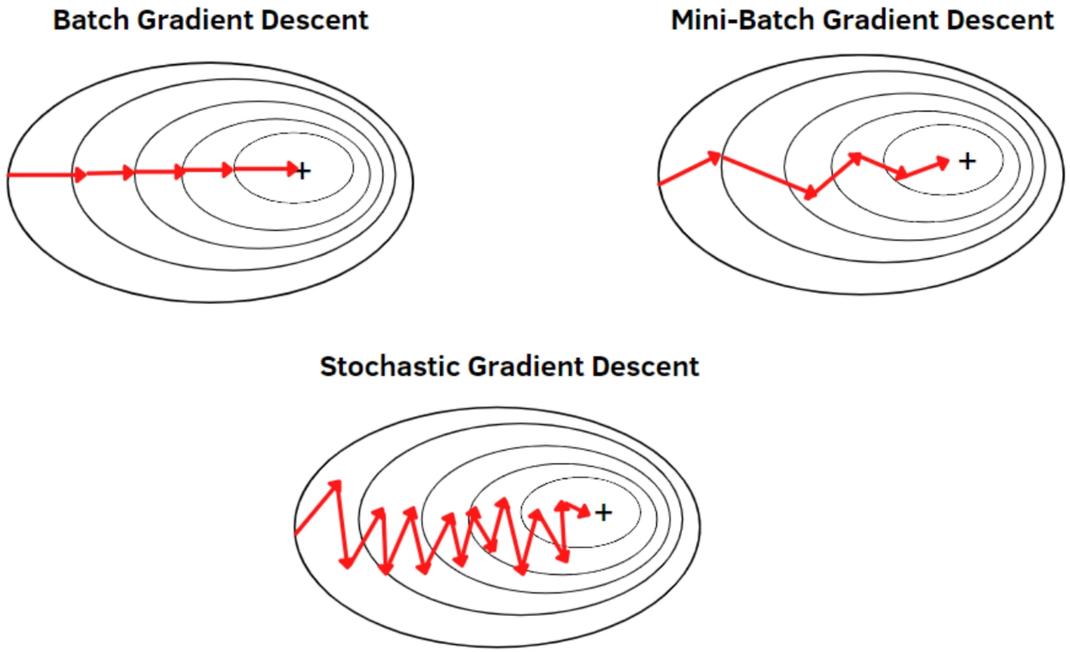
- розмір партії дорівнює розміру навчального набору

Stochastic Gradient Descent:

- розмір партії дорівнює 1

Mini-batch Gradient Descent:

- 1 < розмір партії < розмір навчального набору



SGD with Momentum

Чому SGD з Momentum?

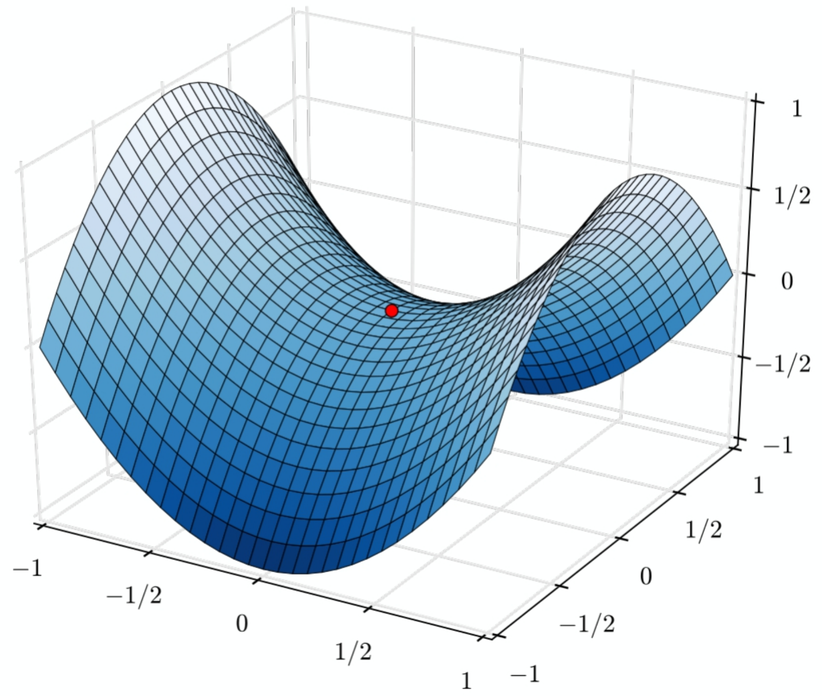
SGD не працює ідеально, тому що під час глибокого навчання ми отримуємо невипуклий графік функції втрат, і якщо використовувати простий SGD, це призводить до низької продуктивності.

Є 3 причини, чому це не працює:

1. Ми потрапляємо в локальні мінімуми і не можемо досягти глобальних мінімумів.

2. Сідлова точка буде зупинкою для досягнення глобальних мінімумів.

3. Висока кривизна.



SGD with Momentum

Чому SGD з Momentum?

Імпульс (Momentum) накопичує градієнт минулих кроків, щоб визначити напрямок руху.
Іншими словами, SGD з Momentum запам'ятовує оновлення
$\Delta\omega$ на кожній ітерації та визначає наступне оновлення
як лінійну комбінацію градієнта та попереднього оновлення:

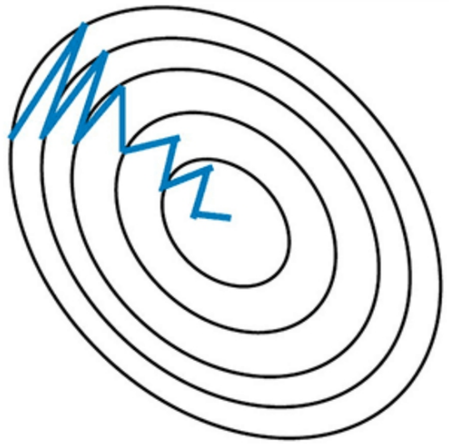
$$\Delta\omega \leftarrow \eta\Delta\omega - \alpha\frac{\partial \text{Loss}}{\partial\omega}$$

$$\omega \leftarrow \omega + \Delta\omega$$

Це призводить до:

$$\omega \leftarrow \omega - \alpha\frac{\partial \text{Loss}}{\partial\omega} + \eta\Delta\omega$$

Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

SGD with Momentum

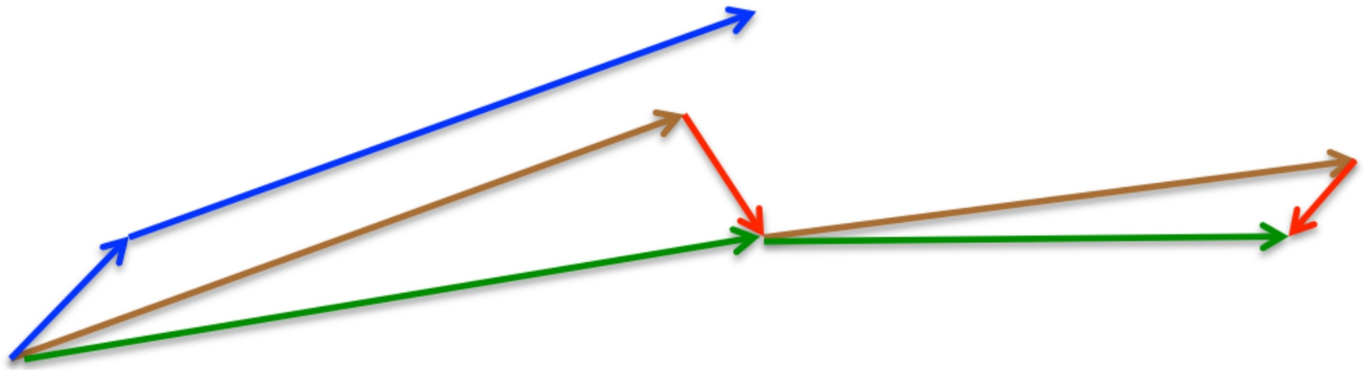
Модифікації Momentum

Стандартний метод Momentum спочатку обчислює градієнт у поточному місці, а потім робить великий стрибок у напрямку оновлення накопиченого градієнта.

Натхненний методом Нестерова для оптимізації опуклих функцій, Ілля Суцкевер запропонував нову форму Momentum, яка часто працює краще:

- **Спочатку** зробіть великий стрибок у напрямку попереднього накопиченого градієнта.
- **Потім** виміряйте градієнт там, де ви закінчите, і внесіть корекцію. (Краще виправити помилку після того, як ви її зробили!)

Пояснення графіку:
- коричневий вектор = стрибок
- червоний вектор = корекція
- зелений вектор = накопичений градієнт
- сині вектори = стандартний Momentum



Математичний вираз виглядає так:

$$\Delta\omega \leftarrow \eta\Delta\omega - \alpha\frac{\partial \text{Loss}}{\partial\omega}(\omega - \Delta\omega)$$

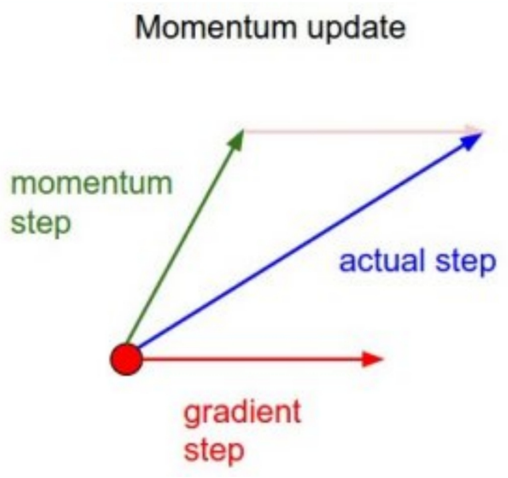
$$\omega \leftarrow \omega + \Delta\omega$$

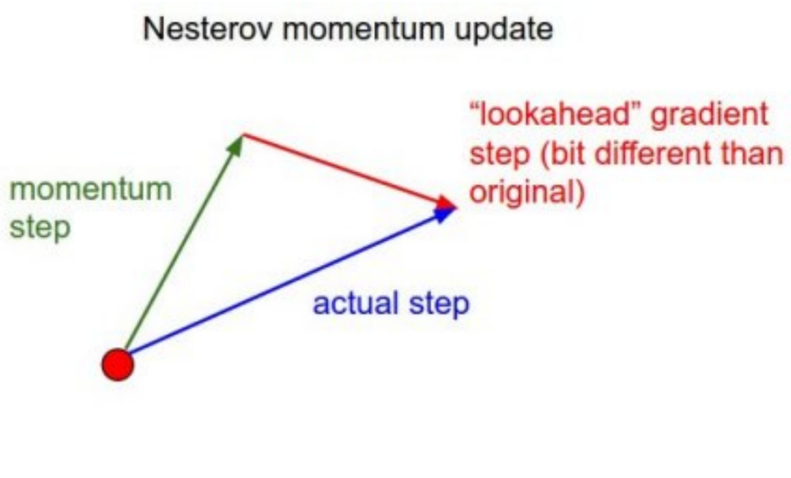
Це призводить до:

$$\omega \leftarrow \omega \leftarrow \eta\Delta\omega - \alpha\frac{\partial \text{Loss}}{\partial\omega}(\omega - \Delta\omega)$$

SGD with Momentum

Замість того, щоб оцінювати градієнт у поточній позиції (червоне коло), ми знаємо, що наш Momentum перенесе нас до кінчика зеленої стрілки. Таким чином, за допомогою Nesterov Momentum ми оцінюємо градієнт у позиції «прогнозованого вперед (lookahead)».





RMSProp

Root Mean Square Propagation

RMSProp — це метод, винайдений Джеффрі Хінтоном у 2012 році, у якому швидкість навчання адаптується для кожного з параметрів. Ідея полягає в тому, щоб розділити швидкість навчання для ваги на поточне середнє значення величин останніх градієнтів для цієї ваги.

Отже, спочатку «running average» обчислюється в термінах середнього квадрата,

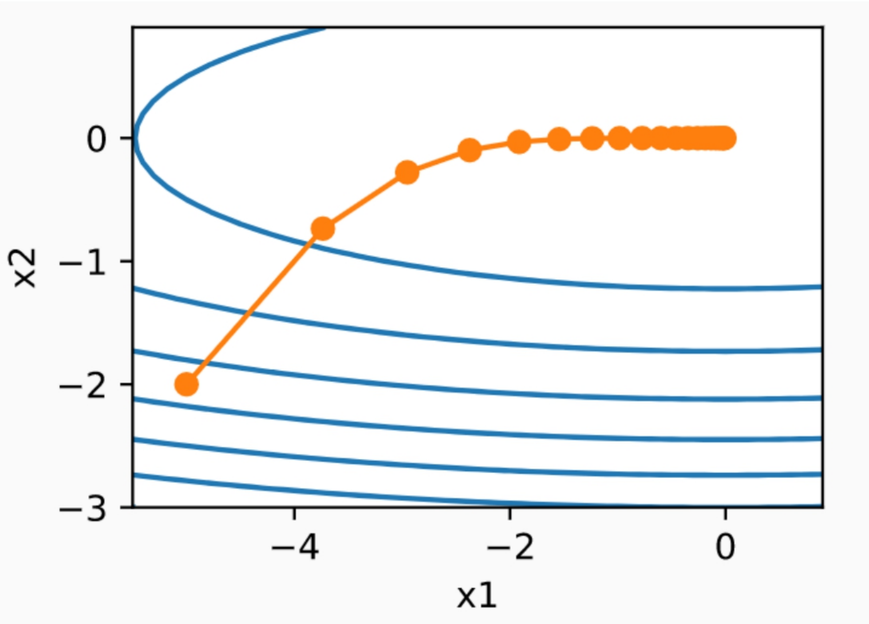
$$\nu(\omega, t) := \gamma\nu(\omega, t-1) + (1-\gamma)(\frac{\partial \text{Loss}}{\partial \omega})^2$$

де γ (gamma) – коефіцієнт забування. Концепція збереження історичного градієнта як суми послідовностей запозичена з Adagrad, але «забуття» введено, щоб вирішити проблему зниження рівня навчання Adagrad у невипуклих задачах шляхом поступового зменшення впливу старих даних.

Параметри оновлюються так:

$$\omega \leftarrow \omega - \frac{\alpha}{\sqrt{\nu(\omega t)}} \frac{\partial \text{Loss}}{\partial \omega}$$

RMSProp показав хорошу адаптацію швидкості навчання в різних програмах.



Траєкторія алгоритму RMSProp
для $\min f(x) = 0.1x_1^2 + 2x_2^2$

Adam

Adaptive Moment Estimation

Adam — це оновлення оптимізатора *RMSProp* 2014 року, яке поєднує його з основною функцією *Momentum Method*. У цьому алгоритмі оптимізації використовуються «running averages» з експоненціальним забуванням як градієнтів, так і других моментів градієнтів. Враховуючи параметри $w^{(t)}$ і функцію втрат $L^{(t)}$, де t індексує поточну ітерацію навчання (починається з 0), оновлення параметра Адама визначається як:

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\frac{\partial L^{(t)}}{\partial w}$$

$$\nu_w^{(t+1)} \leftarrow \beta_2 \nu_w^{(t)} + (1-\beta_2)\frac{\partial^2 L}{\partial w^2}$$

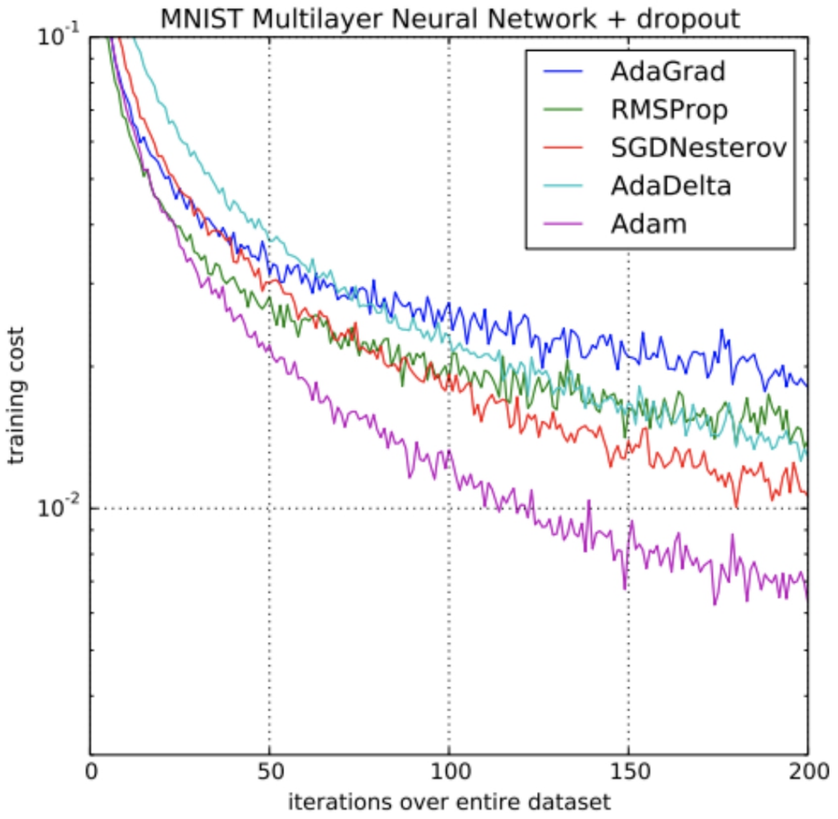
$$\hat{m}_w = \frac{m_w^{(t+1)}}{1-\beta_1^t}$$

$$\hat{\nu}_w = \frac{\nu_w^{(t+1)}}{1-\beta_2^t}$$

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha\frac{\hat{m}_w}{\sqrt{\hat{\nu}_w}+\epsilon}$$

Де ϵ — це малий скаляр (наприклад, 10^{-8}), який використовується для запобігання ділення на 0, а β_1 (наприклад, 0.9) і β_2 (наприклад, 0.999) — коефіцієнти забування для градієнтів і секундних моментів градієнтів відповідно. Зведення в квадрат і квадратний корінь виконується поелементно.

$AdamW$ є пізнішим оновленням, яке пом'якшує неоптимальний вибір алгоритму зменшення ваги в $Adam$.



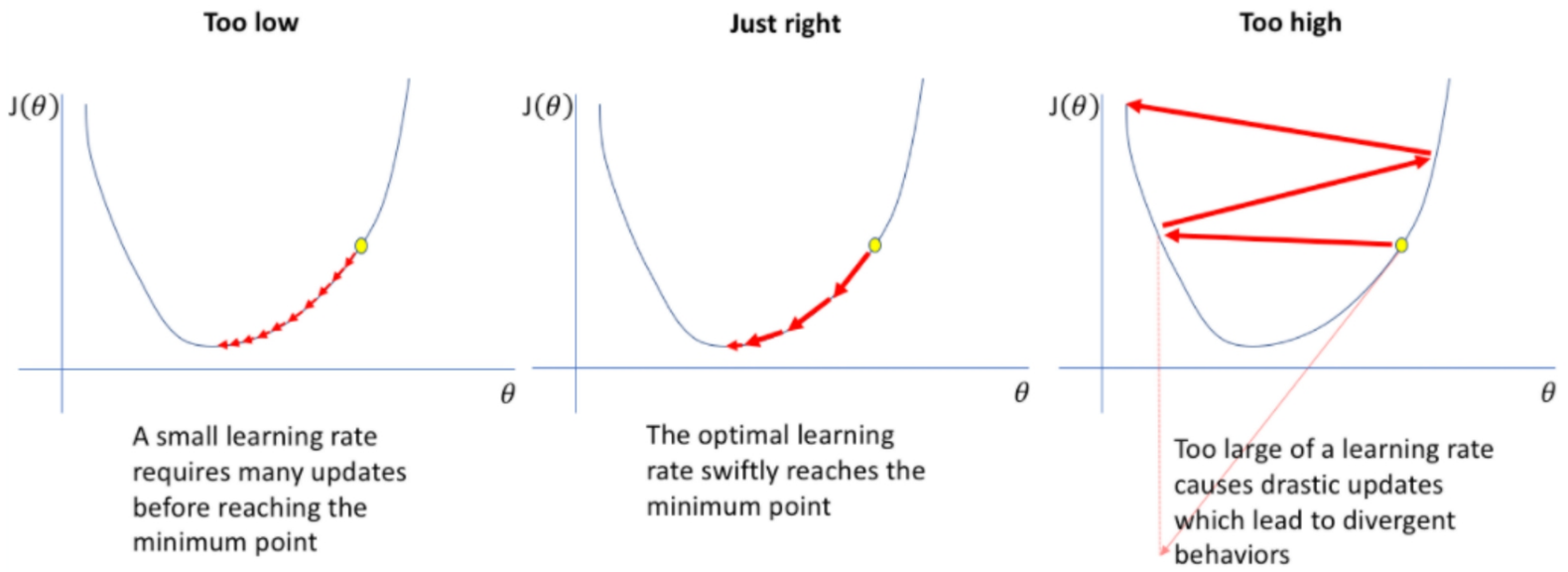
Learning the PyTorch

PyTorch:

- Basics
- Tensors
- Datasets & DataLoaders
- Build the Neural Network
- Optimizing Model Parameters
- Save and Load the Model

Learning Rate Schedulers

Learning Rate Schedulers регулюють швидкість навчання під час процесу тренування моделі. Ідея полягає в тому, щоб почати з відносно високої швидкості навчання, щоб швидко прогресувати, а потім зменшити її для «точного налаштування» ваги в міру тренування. Цей підхід часто призводить до швидшої конвергенції та кращої кінцевої продуктивності.



Learning Rate Schedulers

Псевдокод для використання Learning Rate Schedulers виглядає так:

```
import torch.optim as optim


# ... inside training script ...
model = ...
model = model.to(device)
optimizer = optim.SGD(model.parameters(), lr=1e-3)
# define the learning rate scheduler
scheduler = MyCoolLearningRateScheduler(optimizer,
...)
n_epochs = 123
for epoch in range(n_epochs):
    # ... training loop ...

    # change learning rate
    scheduler.step()
```

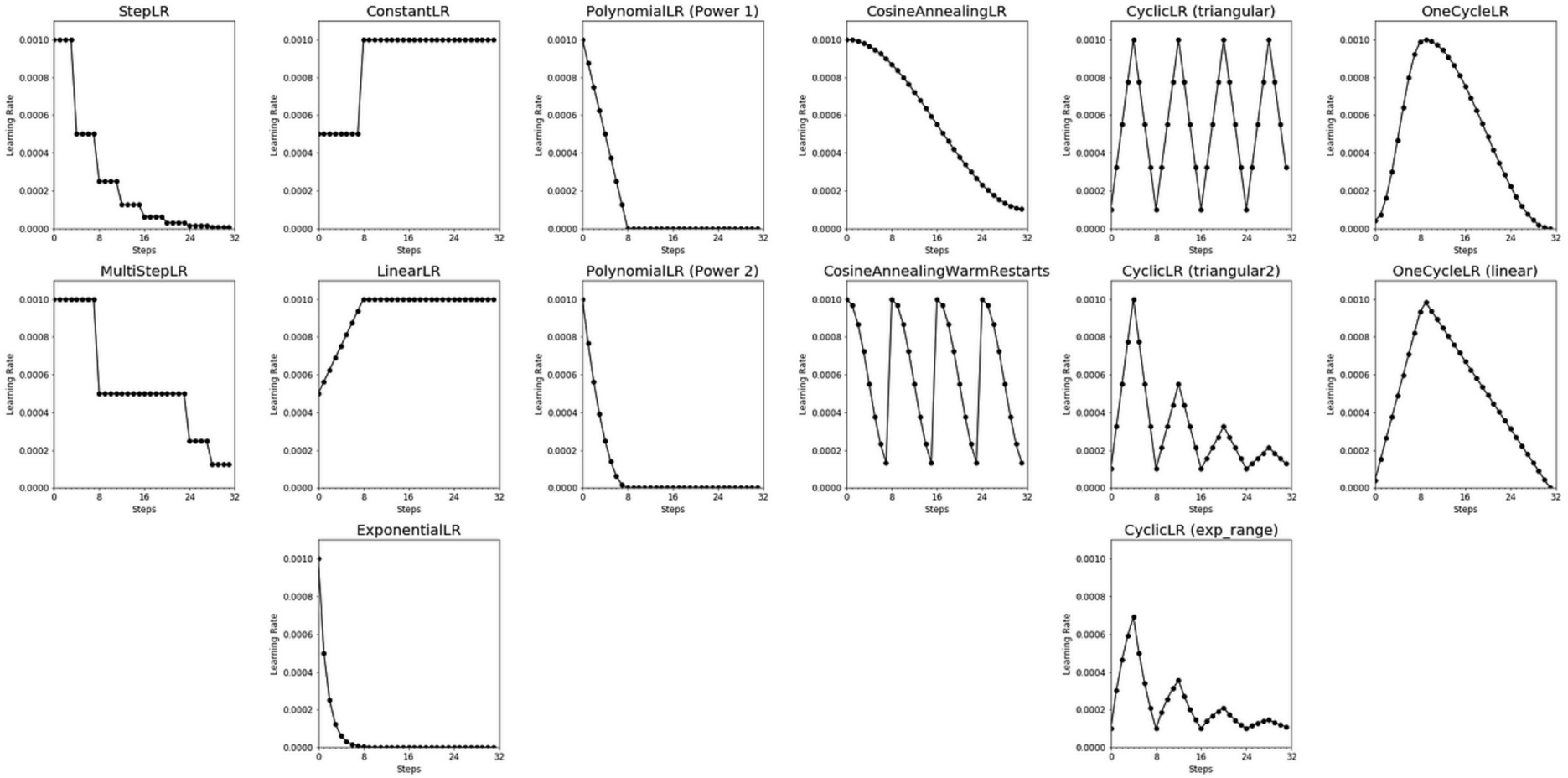
Learning Rate Schedulers

Learning Rate Schedulers:

- **Step Decay** - зменшує швидкість навчання на коефіцієнт кожні кілька епох.
- **Exponential Decay** - експоненціально знижує швидкість навчання, тобто $lr = lr_0 \times e^{-k \times epoch}$, де k - це константа.
- **Cosine Annealing** - регулює швидкість навчання відповідно до косинусної функції.
- **ReduceLROnPlateau** - відстежує метрику валідації (наприклад, функцію втрат) і знижує швидкість навчання, коли метрика перестає покращуватися.
- **Cyclic Learning Rates** - регулярні коливання темпів навчання між нижньою та верхньою межею. Цей підхід може допомогти моделі уникнути локальних мінімумів.
- **One-cycle Learning Rate** - починається з лінійного збільшення швидкості навчання від мінімального до максимального значення, а потім лінійно зменшується. Зазвичай це робиться протягом попередньо визначеної кількості епох або ітерацій.
- **Warm-up** - починається з дуже низького темпу навчання і поступово його збільшує. Це може бути корисним для забезпечення стабільного навчання на початкових етапах, особливо для дуже глибоких мереж або тих, хто використовує такі методи, як transfer learning.

High Level Example

Learning Rate Schedulers



hillel
IT school

Thanks for your attention