ECE 046211 - Technion - Deep Learning

Tal Daniel

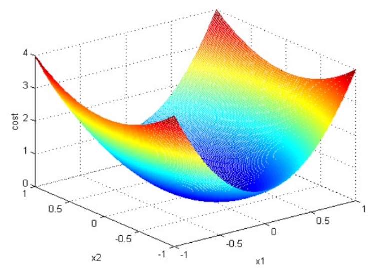
Tutorial 03 - Optimization & Gradient Descent Algorithms

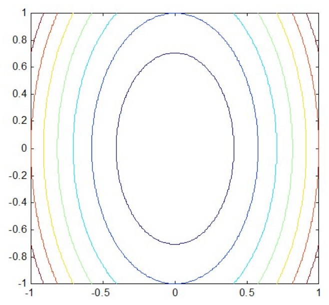
Agenda

In [1]:
```
# imports for the tutorial
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.model_selection import train_test_split
%matplotlib ipympl
# %matplotlib inline

# pytorch imports
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
```

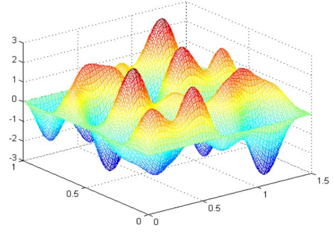
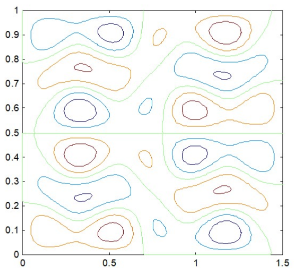
Unimodal vs. Multimodal Optimization

- **Unimodal** - only **one** optimum, that is, the *local* optimum is also global.





- **Multimodal** - more than one optimum.

Most search schemes are based on the assumption of **unimodal** surface. The optimum determined in such cases is called **local optimum design**.

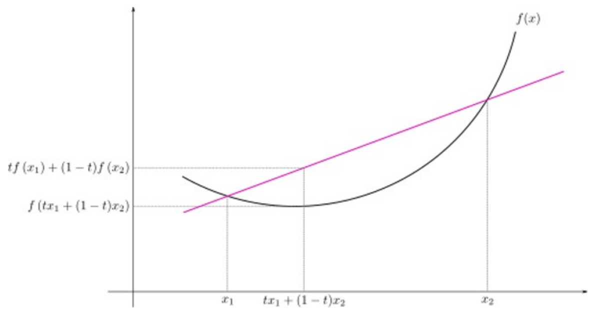
The **global optimum** is the best of all *local optimum* designs.

Convexity

- **Definition**:

$$\forall x_1, x_2 \in X, \forall t \in [0,1]:$$

$$f(tx_1 + (1-t)x_2) \le tf(x_1) + (1-t)f(x_2)$$



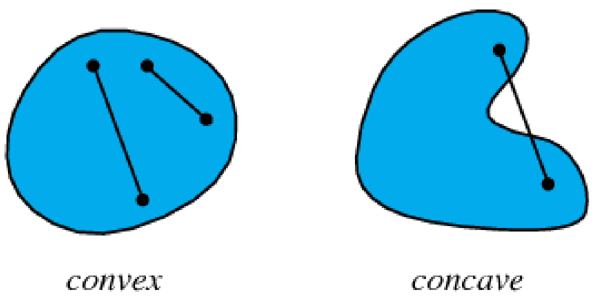
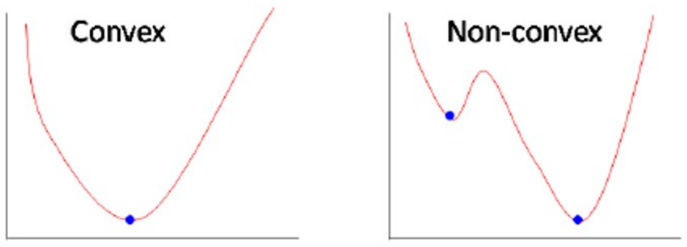


Image Source

- Convex functions are **unimodal**.
    - However, unimodal functions are not always convex, but they are usually still easy to optimize.

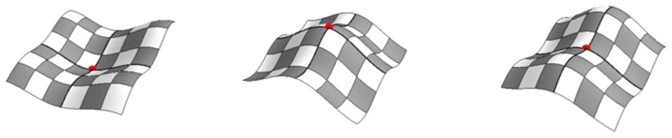


Optimality Conditions

- If $f : \mathbb{R}^d \to \mathbb{R}$ has *local* optimum at x_0 then $\nabla f(x_0) = 0$.
    - $\nabla f(x_0)$ is also called **the gradient** at x_0.
- **The Hessian Matrix** : $H(f)(x)_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(x) \in \mathbb{R}^{d \times d}$

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \frac{\partial^2 f}{\partial x_2 x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d x_1} & \frac{\partial^2 f}{\partial x_d \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_d^2} \end{bmatrix}$$

- If the **Hessian** matrix is:
    - **Positive Definite** (all eigenvalues *positive*) at $x_0 \to$ *local minimum*.
    - **Negative Definite** (all eigenvalues *negative*) at $x_0 \to$ *local maximum*.
    - Both **positive and negative** eigenvalues at $x_0 \to$ *saddle* point.



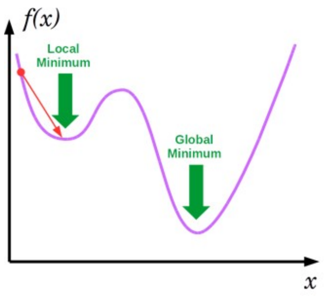
- Note: the Hessian matrix is symmetric if the second partial derivatives are continuous, but this is not always true (Schwarz's theorem).

(Batch) Gradient Descent

- Generic optimization algorithm capable of finding optimal solutions to a wide range of problems.
- The general idea is to tweak parameters **iteratively** to minimize a cost function.
- It measures the local gradient of the error function with regards to the parameter vector (θ or w), and it goes down in the direction of the descending gradient.
- **Once the gradient is zero - the minimum is reached (=convergence)**.

- **Pseudocode**:

```
- Require: Learning rate $\alpha_k$
- Require: Initial parameter vector $w$
- While stopping criterion not met do
    - Compute gradient: $g \leftarrow \nabla_w f(x, w)$
    - Apply update: $w \leftarrow w - \alpha_k g$
    - $k \leftarrow k + 1$
    end while
```



- **Convergene**: When the cost function is *convex* and its slope does not change abruptly, (Batch) GD with a (small enough) *fixed* learning rate will eventually converge to the optimal solution (but the time is depndent on the rate).

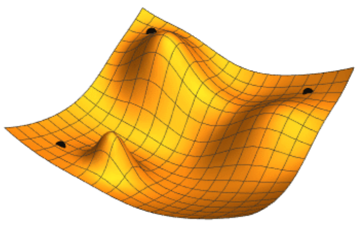


Image Source

Stochastic Gradient Descent (Mini-Batch Gradient Descent)

- The main problem with (Batch) GD is that it uses the **whole** training set to compute the gradients. But what if that training set is huge or each sample has a very large number of features? Computing the gradient can take a very long time.
- *Stochastic* Gradient Descent on the other hand, samples just one instance randomly at every step and computes the gradients based on that single instance (remember the Perceptron algorithm?). This makes the algorithm much faster but due to its randomness, it is much less stable. Instead of steady decreasing untill reaching the minimum, the cost function will bounce up and down, **decreasing only on average**. With time, it will get *very close* to the minimum, but once it is there it will continue to bounce around!
  - When we have outliers, SGD may take us to undesirable areas.
- The final parameters are good but **not optimal**.

- When the cost function is very irregular, this bouncing can actually **help the algorithm escape local minima**, so SGD has better chance to find the *global* minimum.
- How to find optimal parameters using SGD?
  - **Reduce the learning rate gradually**: this is called *learning rate scheduling*.
    - But don't reduce too quickly or you will get stuck at a local minimum or even frozen!
- *Mini-Batch* Gradient Descent - same idea as SGD, but instead of one instance each step, m samples.
  - Get a little bit closer to the minimum than SGD but a little harder to escape local minima.

- **Pseudocode**:

```
Require: Learning rate $\alpha_k$
Require: Initial parameter $w$
While stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set ($m = 1$ for SGD)
    Set $\{x_1, \dots, x_m,\}$ with corresponding targets $\{y_1, \dots, y_m\}$
    Compute gradient: $g \leftarrow \frac{1}{m}\sum_{i=1}^{m} f'(x_i, w, y_i)$
    Apply update: $w \leftarrow w - \alpha_k g$
    $k \leftarrow k + 1$
end while
```

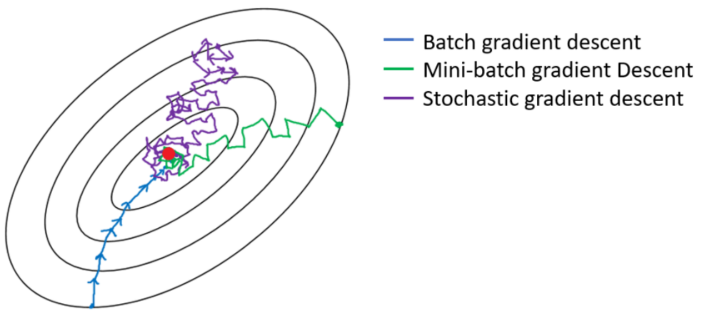


Image Source

GD Comparison Summary

Method	Accuracy	Update Speed	Memory Usage	Online Learning
Batch Gradient Descent	Good	Slow	High	No
Stochastic Gradient Descent	Good (with softening)	Fast	Low	Yes
Mini-Batch Gradient Descent	Good	Medium	Medium	Yes (depends on the MB size)

- **"Online"** - samples arrive while the algorithm runs (that is, when the algorithm starts running, not all samples exist)
- Note: all the Gradient Descent algorithms require **scaling** if the features are not within the same range!

Update Speed

- Assume: number of samples $M = 100$ and we wish to train for $N = 10$ epochs (iterating over all of the samples 10 times).
- Denote `batch_size` with m.
- `batch_size` and `num_epochs` are **hyper-parameters**: parameters that are chosen by the user prior to the training stage, they are not learned.

Method	# Gradients Updates (=iterations)
Batch Gradient Descent, $m = 100$	$\frac{M}{m} \cdot N = 1 * 10$
Stochastic Gradient Descent, $m = 1$	$\frac{M}{m} \cdot N = 100 * 10 = 1000$
Mini-Batch Gradient Descent, $m = 10$	$\frac{M}{m} \cdot N = 10 * 10 = 100$

- The update speed affects how we tune the **learning rate**, which is also a hyper-parameter (usually, larger batches enable using higher learning rates).

Challenges

- Choosing a **learning rate**.
  - Defining **learning schedule**.
- Working with features of different scales (e.g. heights (cm), weights (kg) and age (scalar)).
- Avoiding **local minima** (or *suboptimal* minima).

The Learning Rate

- **Learning Rate** hyper-parameter - it is the size of step to be taken in each iteration.
  - Too *small* $\rightarrow$ the algorithm will have to go through many iterations to converge, which will take a long time.
  - Too *high* $\rightarrow$ might make the algorithm diverge as it may miss the minimum.

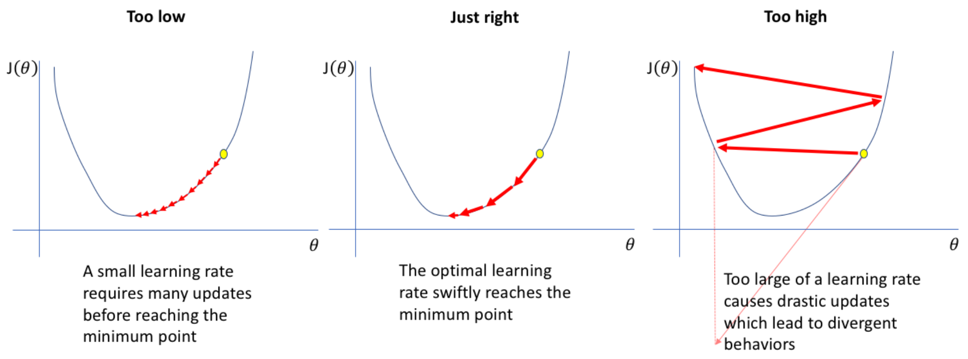

Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Image Source

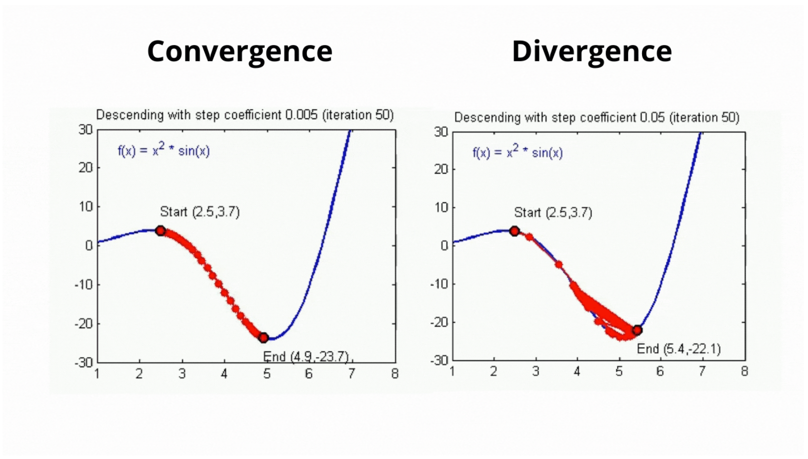


Image Source

Example - (Multivariate) Linear Least Squares

- **Problem Formulation**
  - $y \in \mathbb{R}^N$ - vector of values.
  - $X \in \mathbb{R}^{N \times L}$ - data matrix with N examples and L *features*.
  - $w \in \mathbb{R}^L$ - the *parameters* to be learned, a **weight for each feature**.
- **Goal**: find w that best fits the measurement y, that is, find a *weighted linear combination* of the feature vector to best fit the measurment y.
- Mathematiacally, the problem is:

$$\min_w f(w;x,y) = \min_w \sum_{i=1}^{N} ||x_i w - y_i||^2$$

- In vector form:

$$\min_w f(w;x,y) = \min_w ||Xw - Y||^2$$

(Multivariate) LLS - Analytical Solution

- Mathematically:

$$\min_w f(w;x,y) = \min_w ||Xw - Y||^2 = \min_w (Xw - Y)^T (Xw - Y) = \min_w (w^T X^T X w - 2w^T X^T Y + Y^T Y)$$

- The derivative:

$$\nabla_w f(w;x,y) = (X^T X + X^T X)w - 2X^T Y = 0 \rightarrow w = (X^T X)^{-1} X^T Y$$

$$X^T X \in \mathbb{R}^{L \times L}$$

- Notice how the gradient is dependent on the features, which is why scaling them is important when applying gradient descent (however, scaling is not necessary if we use the closed-form solution, the least-squares solution).

In [2]:
```
# let's load the cancer dataset
dataset = pd.read_csv('./datasets/cancer_dataset.csv')
# print the number of rows in the data set
number_of_rows = len(dataset)
# reminder, the data looks like this
dataset.sample(10)
```

Out[2]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean
489	913535	M	16.690	20.20	107.10	857.6	0.07497	0.07112
379	9013838	M	11.080	18.83	73.30	361.6	0.12160	0.21540
103	862980	B	9.876	19.40	63.95	298.3	0.10050	0.09697
460	911296201	M	17.080	27.15	111.20	930.9	0.09898	0.11100
72	859717	M	17.200	24.52	114.20	929.4	0.10710	0.18300
98	862485	B	11.600	12.84	74.34	412.6	0.08983	0.07525
391	903483	B	8.734	16.84	55.27	234.3	0.10390	0.07428
536	91979701	M	14.270	22.55	93.77	629.8	0.10380	0.11540
371	9012568	B	15.190	13.21	97.65	711.8	0.07963	0.06934
503	915143	M	23.090	19.83	152.10	1682.0	0.09342	0.12750

10 rows × 33 columns

In [3]:
```
def plot_3d_lls(x, y, z, lls_sol, title=""):
    # plot
    fig = plt.figure(figsize=(5, 5))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(x, y, z, label='Y')
    ax.scatter(x, y, lls_sol, label='Xw')
    ax.legend()
    ax.set_xlabel('Radius Mean')
    ax.set_ylabel('Area Mean')
    ax.set_zlabel('Perimeter Mean')
    ax.set_title(title)
```

In [4]:
```
def batch_generator(x, y, batch_size, shuffle=True):
    """
    This function generates batches for a given dataset x.
    """
    N, L = x.shape
    num_batches = N // batch_size
    batch_x = []
    batch_y = []
    if shuffle:
        # shuffle
        rand_gen = np.random.RandomState(0)
        shuffled_indices = rand_gen.permutation(np.arange(N))
        x = x[shuffled_indices, :]
        y = y[shuffled_indices, :]
    for i in range(N):
        batch_x.append(x[i, :])
        batch_y.append(y[i, :])
        if len(batch_x) == batch_size:
            yield np.array(batch_x).reshape(batch_size, L), np.array(batch_y).reshape(batch_size, 1)
            batch_x = []
            batch_y = []
    if batch_x:
        yield np.array(batch_x).reshape(-1, L), np.array(batch_y).reshape(-1, 1)
```

- **Pseudocode** for Linear Regression:
  - **Require**: Learning rate α_k
  - **Require**: Initial parameter w
  - **While** stopping criterion not met **do**
    - Sample a minibatch of m examples from the training set ($m = 1$ for SGD)

```
            ◦ Set $\tilde{X} = [x_1, \ldots, x_m]$ with corresponding targets $\tilde{Y} = [y_1, \ldots, y_m]$
            ◦ Compute gradient: $g \leftarrow 2\tilde{X}^T\tilde{X}w - 2\tilde{X}^T\tilde{Y}$
            ◦ Apply update: $w \leftarrow w - \alpha_k g$
            ◦ $k \leftarrow k + 1$
        ▪ **end while**
```

In [5]:
```python
# multivaraite mini-batch gradient descent
X = dataset[['radius_mean', 'area_mean']].values
Y = dataset[['perimeter_mean']].values
# Scaling
X = (X - X.mean(axis=0, keepdims=True)) / X.std(axis=0, keepdims=True)
Y = (Y - Y.mean(axis=0, keepdims=True)) / Y.std(axis=0, keepdims=True)
N = X.shape[0]
batch_size = 10
num_batches = N // batch_size
print("total batches:", num_batches)
```

```
total batches: 56
```

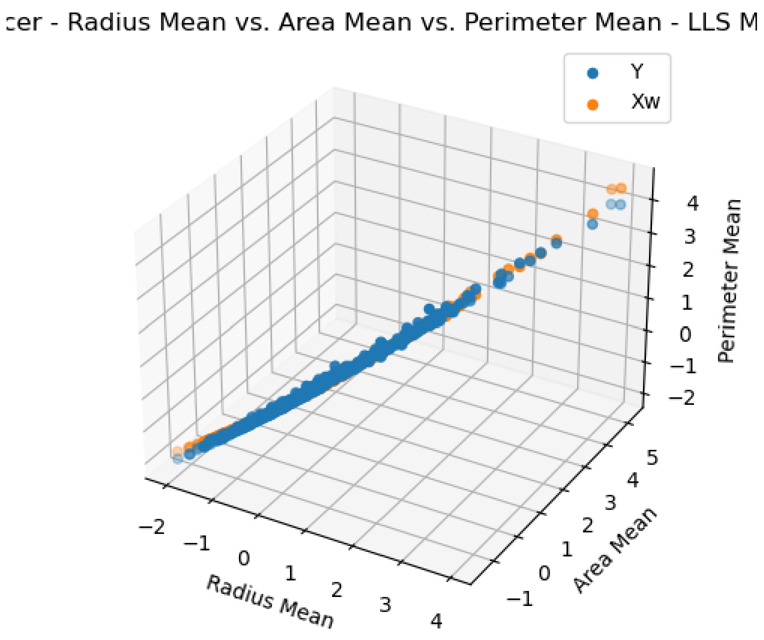
In [6]:
```python
num_epochs = 10
alpha_k = 0.001
batch_gen = batch_generator(X, Y, batch_size, shuffle=True)
L = X.shape[-1]
# initialize w
w = np.zeros((L, 1))
for i in range(num_epochs):
    for batch_i, batch in enumerate(batch_gen):
        batch_x, batch_y = batch
        if batch_i % 50 == 0:
            print("iter:", i, "batch:", batch_i, " w = ")
            print(w)
        gradient = 2 * batch_x.T @ batch_x @ w - 2 * batch_x.T @ batch_y
        w = w - alpha_k * gradient
    batch_gen = batch_generator(X, Y, batch_size, shuffle=True)

lls_sol = X @ w
```

```
iter: 0 batch: 0  w =
[[0.]
 [0.]]
iter: 0 batch: 50  w =
[[0.4391737 ]
 [0.41735388]]
iter: 1 batch: 0  w =
[[0.45621431]
 [0.43296123]]
iter: 1 batch: 50  w =
[[0.50458588]
 [0.46953377]]
iter: 2 batch: 0  w =
[[0.50659246]
 [0.46976394]]
iter: 2 batch: 50  w =
[[0.51664408]
 [0.46913792]]
iter: 3 batch: 0  w =
[[0.51715596]
 [0.46786206]]
iter: 3 batch: 50  w =
[[0.52339741]
 [0.46367398]]
iter: 4 batch: 0  w =
[[0.52374047]
 [0.4622501 ]]
iter: 4 batch: 50  w =
[[0.5295512 ]
 [0.45779193]]
iter: 5 batch: 0  w =
[[0.52985556]
 [0.456353  ]]
iter: 5 batch: 50  w =
[[0.53556725]
 [0.45194588]]
iter: 6 batch: 0  w =
[[0.53584598]
 [0.45050492]]
iter: 6 batch: 50  w =
[[0.54149192]
 [0.44617918]]
iter: 7 batch: 0  w =
[[0.54174661]
 [0.44473749]]
iter: 7 batch: 50  w =
[[0.54733087]
 [0.44049501]]
iter: 8 batch: 0  w =
[[0.54756198]
 [0.43905271]]
iter: 8 batch: 50  w =
[[0.55308576]
 [0.43489257]]
iter: 9 batch: 0  w =
[[0.55329364]
 [0.43344969]]
iter: 9 batch: 50  w =
[[0.55875783]
 [0.42937075]]
```

In [7]:
```
# plot
plot_3d_lls(X[:,0], X[:, 1], Y, lls_sol, "Breast Cancer - Radius Mean vs. Area Mean vs. Perimeter Mean - LLS Mini
print("w:")
print(w)
```

```
w:
[[0.55894282]
 [0.42792729]]
```

Figure



Learning Rate Scheduling (Annealing)

- When training deep networks, it is usually helpful to anneal (gradually change the rate) the learning rate over time.
  - **Physics intuition**: with a high learning rate, the system contains too much *kinetic energy* and the parameter vector bounces around chaotically, unable to settle down into deeper, but narrower parts of the loss function.
- Knowing when to decay the learning rate can be tricky: decay it **slowly** and you'll be wasting computation bouncing around chaotically with little improvement for a long time. But decay it **too aggressively** and the system will cool too quickly, unable to reach the best position it can.
- There are three common types of implementing the learning rate decay: step deacy, exponential decay and $1/t$ decay.
  - Recently, *cyclic* learning schedulers, such as One-cycle learning rate scheduler or *cosine* scheduling, have been gaining popularity as well.

- **(Multi) Step decay**: Reduce the learning rate by some factor every few epochs.
  - Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs. These numbers depend heavily on the type of problem and the model.
  - One heuristic you may see in practice is to watch the *validation error* while training with a fixed learning rate, and reduce the learning rate by a constant (e.g. 0.5) whenever the validation error stops improving.

- **Exponential decay**: has the mathematical form:

$$\alpha = \alpha_0 \exp(-kt),$$

  where α_0, k are hyperparameters and t is the iteration number (but you can also use units of epochs).
  - α_0 is the initial learning rate.
  - k is also referred to as the `gamma` (γ) hyperparameter.

- **1/t decay**: has the mathematical form:

$$\alpha = \frac{\alpha_0}{1+kt},$$

  where α_0, k are hyperparameters and t is the iteration number.

- **Cosine annealing**: has the mathematical form:

$$\alpha = \alpha_{min} + \frac{1}{2}(\alpha_0 - \alpha_{min})\left(1 + \cos\left(\frac{t}{t_{max}}\pi\right)\right),$$

where α_{min} is the minimum learning rate (deafult is 0) and t_{max} is number of iterations to perform a cycle.

- In practice, we usually find that the **step decay** is slightly preferable because the hyperparameters it involves (the fraction of decay and the step timings in units of epochs) are more interpretable than the hyperparameter k.
- Lastly, if you can afford the computational budget, you can try a slower decay and train for a longer time.

Learning Rate Scheduling in PyTorch

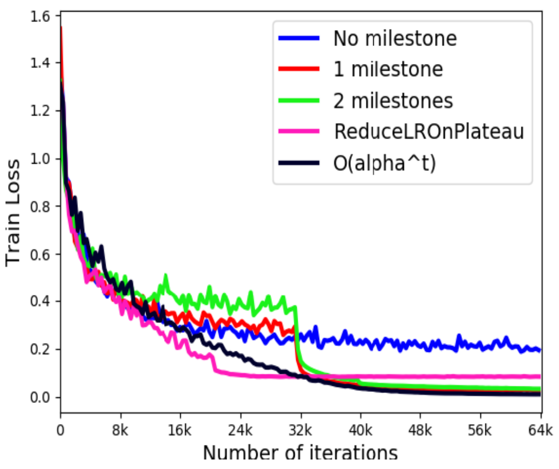
- We will use learning rate scheduling to train the neural network models later in the course.
- PyTorch offers several schedulers which can be found here.
- A typical workflow with schedulers (learning rate scheduling should be applied **after** optimizer's update):

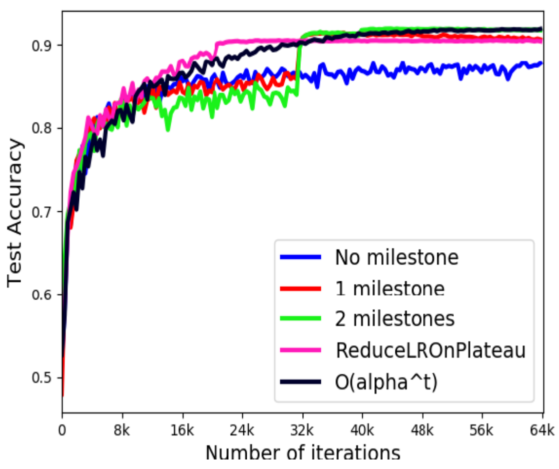
```
scheduler = ...
for epoch in range(100):
train(...)
validate(...)
scheduler.step()
```

- `torch.optim.lr_scheduler.StepLR`
- `torch.optim.lr_scheduler.MultiStepLR`
- `torch.optim.lr_scheduler.ExponentialLR`
- `torch.optim.lr_scheduler.CosineAnnealingLR`
- `torch.optim.lr_scheduler.OneCycleLR`
- `torch.optim.lr_scheduler.CyclicLR`
- And more...

Reducing LR on Plateau

- Reduce learning rate when a metric has stopped improving. Usually the validation accuracy.
- Models often benefit from reducing the learning rate by a factor of 2-10 once learning does not improve.
- This scheduler reads a metrics quantity and if no improvement is seen for a `patience` number of epochs, the learning rate is reduced.
- In PyTorch: `torch.optim.lr_scheduler.ReduceLROnPlateau`.





- Exponential Step Sizes for Non-Convex Optimization, Li et al. 2020.
- Plots of the train loss and test accuracy for training a 20-layer Residual Network to do image classification on CIFAR-10.
- The number of milestones in the legend denotes how many times we can choose to decrease the step size during training.

Momentum & Nesterov Momentum

- Gradient descent is simple and has many virtues, but **speed** is not one of them.
- For a step-size small enough, gradient descent makes a monotonic improvement at every iteration. It should always converge (sometimes to a local minimum).
- **Momentum update** is another optimization approach that *almost always* enjoys better convergence rates in deep networks.
  - It can be seen as a **"global" (equally for all parameters) adaptive learning rate**.

- This update can be motivated from a physical perspective of the optimization problem. In particular, the loss can be interpreted as the height of a *hilly terrain*.
  - Initializing the parameters with random numbers is equivalent to setting a particle with zero initial velocity at some location. The optimization process can then be seen as equivalent to the process of simulating the parameter vector (i.e. a particle) as rolling on the landscape.
  - Since the force on the particle is related to the gradient of potential energy (i.e. $F = -\nabla U$), the force felt by the particle is precisely the (negative) gradient of the loss function.
  - Moreover, $F = ma$ so the (negative) gradient is in this view proportional to the acceleration of the particle.
  - The physics view suggests an update in which the gradient only directly influences the velocity (and maintains information about the acceleration), which in turn has an effect on the position.

- Momentum proposes the following tweak to gradient descent, giving gradient descent **a short-term memory**:

$$z^{k+1} = \beta z^k - \alpha \nabla f(w^k)$$

$$w^{k+1} = w^k + z^{k+1}$$

  - α is the learning rate.
- When $\beta = 0$, we recover gradient descent. But for $\beta = 0.99$ (sometimes 0.999, if things are really bad), this appears to be the boost we need. Our iterations regain that speed and boldness it lost, speeding to the optimum with a renewed energy.

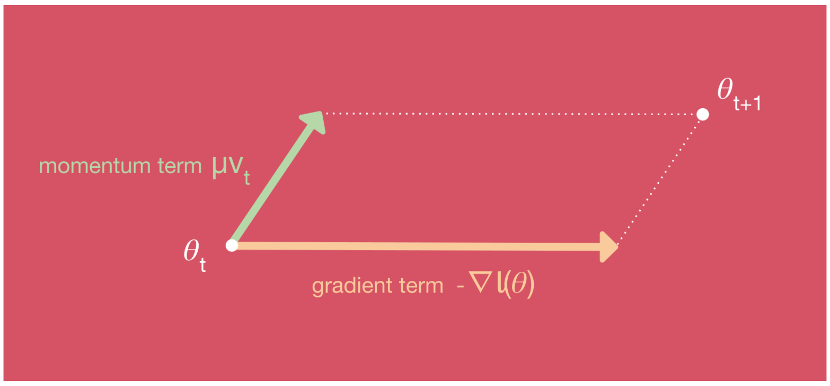
- β is a variable that is sometimes called *momentum*.

- Effectively, this variable **damps the velocity and reduces the kinetic energy of the system**, or otherwise the particle would never come to a stop at the bottom of a hill.

- With Momentum update, the parameter vector will build up velocity in any direction that has consistent gradient.

- Momentum Demo

- Note: Momentum usually works in **larger batches** and may break in smaller batches.



- Image Source

Nesterov Momentum

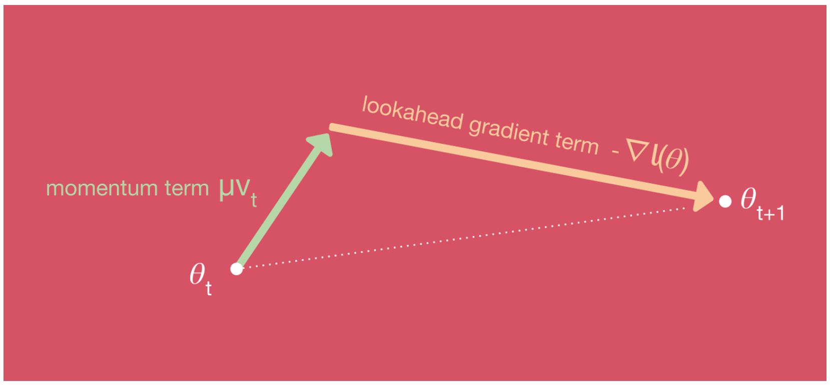
- Nesterov Momentum is a slightly different version of the momentum update that has gained popularity.
- It enjoys stronger theoretical convergence guarantees for **convex functions** and in practice it also consistently works slightly better than standard momentum.
- The core idea behind Nesterov momentum is that when the current parameter vector is at some position x, then looking at the momentum update above, we know that the momentum term alone (i.e. ignoring the second term with the gradient) is about to nudge the parameter vector by $\beta * z_k$.
- Therefore, if we are about to compute the gradient, we can treat the future approximate position $x + \beta * z_k$ as a **“lookahead”** - this is a point in the vicinity of where we are soon going to end up.

- Hence, it makes sense to compute the **gradient** at $x + \beta * z_k$ instead of at the "old/stale" position x, since while the gradient term always points in the right direction, the momentum term may not.
- If the momentum term points in the wrong direction or overshoots, the gradient can still "go back" and correct it in the same update step.

- **Nesterov Momentum**:

$$z^{k+1} = \beta z^k - \alpha \nabla f(w^k + \beta z^k)$$

$$w^{k+1} = w^k + z^{k+1}$$



- Image Source

Momentum in PyTorch

- `torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9, nesterov=True)`

In [ ]:
```
# simple optimizer and lr scheduling example
# courtesy of: deeplearningwizard.com/deep_learning/boosting_models_pytorch/lr_scheduling/

from torch.optim.lr_scheduler import ReduceLROnPlateau

'''
STEP 1: LOADING DATASET
'''

train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())

'''
STEP 2: MAKING DATASET ITERABLE
'''

batch_size = 100
n_iters = 6000
num_epochs = n_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

'''
STEP 3: CREATE MODEL CLASS
'''
class SimpleModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(SimpleModel, self).__init__()
        # Linear function
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        # Non-linearity
        self.relu = nn.ReLU()
        # Linear function (readout)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Linear function
        out = self.fc1(x)
        # Non-linearity
        out = self.relu(out)
        # Linear function (readout)
        out = self.fc2(out)
        return out
'''
STEP 4: INSTANTIATE MODEL CLASS AND DEVICE
'''
input_dim = 28 * 28
hidden_dim = 100
output_dim = 10
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

model = SimpleModel(input_dim, hidden_dim, output_dim).to(device)

'''
STEP 5: INSTANTIATE LOSS CLASS
'''
criterion = nn.CrossEntropyLoss()


'''
STEP 6: INSTANTIATE OPTIMIZER CLASS
'''
learning_rate = 0.1

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9, nesterov=True)

'''
STEP 7: INSTANTIATE STEP LEARNING SCHEDULER CLASS
'''
# lr = lr * factor
# mode='max': look for the maximum validation accuracy to track
# patience: number of epochs - 1 where loss plateaus before decreasing LR
        # patience = 0, after 1 bad epoch, reduce LR
# factor = decaying factor
scheduler = ReduceLROnPlateau(optimizer, mode='max', factor=0.1, patience=0, verbose=True)

'''
STEP 7: TRAIN THE MODEL
'''
iteration = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Send images and labels to device
        images = images.view(-1, 28 * 28).to(device)
        labeles = labels.to(device)

        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iteration += 1

        if iteration % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                # Send images and labels to device
                images = images.view(-1, 28 * 28).to(device)
                labeles = labels.to(device)

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs.data, 1)

                # Total number of labels
                total += labels.size(0)

                # Total correct predictions
                # Without .item(), it is a uint8 tensor which will not work when you pass this number to the sche
                correct += (predicted == labels).sum().item()

            accuracy = 100 * correct / total

            # Print Loss
            # print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.data[0], accuracy))

    # Decay Learning Rate, pass validation accuracy for tracking at every epoch
    print('Epoch {} completed'.format(epoch))
    print('Loss: {}. Accuracy: {}'.format(loss.item(), accuracy))
    print('-' * 20)
    scheduler.step(accuracy)  # accuracy is used to track down a plateau
```

Adaptive Learning Rate Methods

Adapative learning methods compute individual learning rates for different parameters. Previously, we performed an update for all parameters w (or θ) at once as every parameter w_i used the same learning rate α.

Popular algorithms include: AdaGrad, Rprop, RMSprop, Adam and more...

Optimizers in PyTorch

All optimizers mentioned in this tutorial and more can be imported from the `torch.optim` library. Check out the full list of available optimizers in this link.

Adagrad

- **Adagrad**: one of the first adaptive learning rate algorithm, with the basic idea of adapting the learning rate to the parameters by performing smaller updates (i.e. low learning rates) for parameters associated with *frequently* occurring features, and larger updates (i.e. high learning rates) for parameters associated with *infrequent* features.
    - For this reason, it works well with *sparse* data.
- Adagrad uses a different learning rate for every parameter w_i at every time step k.
- We denote:
    - α - the learning rate.
    - $g_k = \nabla f(w^k)$, the gradient at time step k, and $g_{i,k}$ the *partial* derivative w.r.t. the parameter w_i at time step k.
    - $G_k \in \mathbb{R}^{d \times d}$ - a *diagonal* matrix, where each element $G_{i,i}^k$ is the **sum of squares of the gradients w.r.t** w_i **up to time step** k, $G_{i,i}^k = \sum_{j=1}^k g_{i,j}^2$.
    - ϵ, a "smoothing" term that prevents division by zero, deafult is 10^{-8}, but can range from 10^{-4} to 10^{-8}.

- The Adagrad update rule:

$$w_i^{k+1} = w_i^k - \frac{\alpha}{\sqrt{G_{i,i}^k + \epsilon}} \cdot g_{i,k}$$

- Interestingly, without the square root operation, the algorithm performs much worse.
- In vectorized form, we use the matrix-vector product $\odot$ between G_k and g_k:

$$w^{k+1} = w^k - \frac{\alpha}{\sqrt{G^k + \epsilon}} \odot g_k$$

- Adagrad eliminates, or alleviates, the need to manually tune the learning rate, which is nice.
- Most implementations use a default value of 0.01 for the learning rate.
- However, its main **weakness** is the accumulation of the squared gradients (a positive quantity) in the denominator which keeps growing during training and causes the learning rate to **shrink** and eventually become very small, at which point the algorithm doesn't acquire additional knowledge.

Adagrad in PyTorch

- `torch.optim.Adagrad(model.parameters(), lr=learning_rate, initial_accumulator_value=0, eps=1e-10)`

RMSprop

- **RMSprop**: an unpublished (no official paper) optimization algorithm designed for neural networks, first proposed by Geoffrey Hinton in lecture 6 (slide 29) of the online course "Neural Networks for Machine Learning".
- The RMSProp update adjusts the **Adagrad** method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate.
- In particular, it uses **a moving average of squared gradients** instead.
- We denote:
  - α - the learning rate.
  - $g_k = \nabla f(w^k)$
  - $\mathbb{E}[g^2]$ - moving average of squared gradients (stored in a cache with squared gradients from previous iterations).
  - β - moving average parameter (good default value — 0.9).

- The RMSprop update rule:

$$\mathbb{E}[g^2]_{k+1} = \beta\mathbb{E}[g^2]_k + (1-\beta)g_k^2$$

$$w^{k+1} = w^k - \frac{\alpha}{\sqrt{\mathbb{E}[g^2]_{k+1}}}\nabla f(w^k)$$

- The learning rate is adapted by dividing by the root of squared gradient, but since we only have the estimate of the gradient on the current mini-batch, we need instead to use the moving average of it.

RMSprop in PyTorch

- `torch.optim.RMSprop(model.parameters(), lr=learning_rate, alpha=0.99)`
  - `alpha` is β from the equations above, the moving average parameter.

Adam - Adaptive Moment Estimation

- **Adam**: another optimization method that computes adaptive learning rates for each parameter.
- Adam combines the advantages of Adagrad, RMSprop and Momentum: It uses the **squared gradients to scale the learning rate** like RMSprop and it takes advantage of momentum by using **moving average of the gradient instead of the gradient itself** like SGD with momentum.
- In addition to storing an exponentially decaying average of past **squared gradients** like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past **gradients** similar to momentum.

    - Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, and thus prefers flat minima in the error surface.

- We denote:
    - α - the learning rate.
    - m - moving average of gradients. Estimates the first moment (mean) of the gardients.
    - v - moving average of squared gradients. Estimates the second momemnt (variance) of the gradients.
    - β_1 - moving average parameter for m (default: 0.9).
    - β_2 - moving average parameter for v (default: 0.999).

- The Adam update rule:

$$\mathbb{E}[g]_{k+1} = m_{k+1} = \beta_1 m_k + (1-\beta_1)\nabla f(w^k) = \beta_1 m_k + (1-\beta_1) g_k$$

$$\mathbb{E}[g^2]_{k+1} = v_{k+1} = \beta_2 v_k + (1-\beta_2)(\nabla f(w^k))^2 = \beta_2 v_k + (1-\beta_2) g_k^2$$

  Then, we use an **unbiased** estimation:

$$\hat{m}_{k+1} = \frac{m_{k+1}}{1-\beta_1^{k+1}}$$

$$\hat{v}_{k+1} = \frac{v_{k+1}}{1-\beta_2^{k+1}}$$

  (the β's are taken with the power of the current iteration)

$$w^{k+1} = w^k - \frac{\alpha}{\sqrt{\hat{v}_{k+1}}+\epsilon}\hat{m}_{k+1}$$

- ϵ, a "smoothing" term that prevents diviosn by zero, deafult's is 10^{-8}, but can range from 10^{-4} to 10^{-8}. Note that the default $\epsilon = 1e-8$ might be sub-optimal for various tasks, and thus it is recommended to test other values $\in [1e-3, 1e-12]$ as well.
    - EAdam Optimizer: How ε Impact Adam - Wei Yuan, Kai-Xin Gao.
    - ε, a A Nuisance No More - Zack Nado.

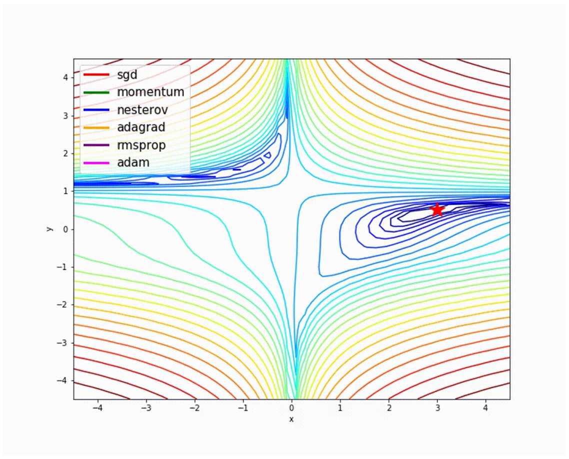
Adam in PyTorch

- `torch.optim.Adam(model.parameters(), lr=learning_rate, betas=(0.9, 0.999))`

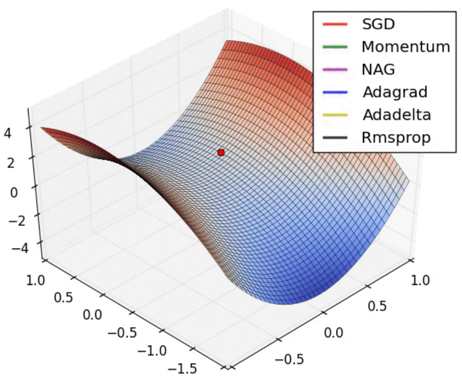
If you want to use weight decay (i.e., L_2 regularization on the weights), use `AdamW`, which adds a fix to the update rule to account for the weight regularization.

- `torch.optim.AdamW(model.parameters(), lr=learning_rate, betas=(0.9, 0.999), weight_decay=0.01)`

Comparison Between Methods



- Contours of a loss surface and time evolution of different optimization algorithms.
- Notice the "overshooting" behavior of **momentum-based methods**, which makes the optimization look like a ball rolling down the hill.
- Image Source



- A visualization of a **saddle point** in the optimization landscape, where the curvature along different dimension has different signs (one dimension curves up and another down).
- Notice that SGD has a very hard time breaking symmetry and gets *stuck* on the top.
- Conversely, algorithms such as RMSprop will see very low gradients in the saddle direction. Due to the denominator term in the RMSprop update, this will increase the effective learning rate along this direction, helping RMSProp proceed.
- Image credit: Alec Radford

Recent Advances in Optimizers

- Adam, "the king of optimizers", has been the go-to optimizer for some time now, and while there were several attempts to dethrone it, there was not a major breakthrough in the optimizers literature to replace it, and Adam is still widely used in all major deep learning domains.
- However, several notable recent works have proposed alternatives that show great promise as candidates to replace Adam, which we cover below.

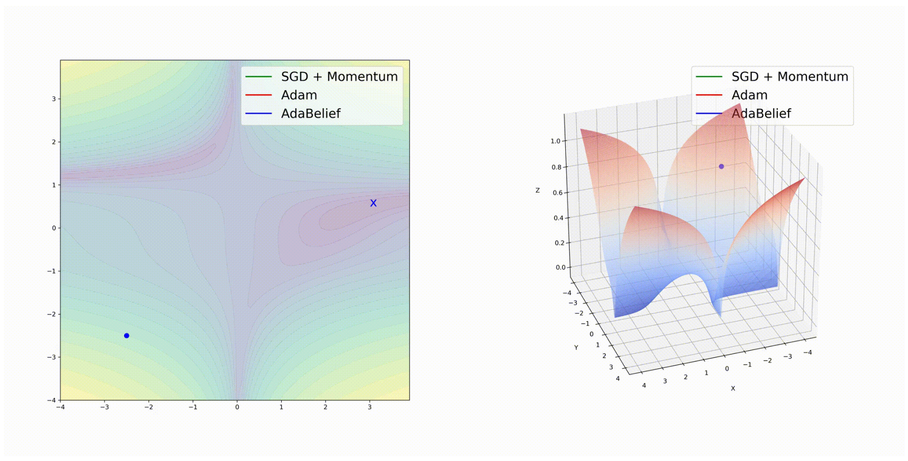
AdaBelief

- Introduced in AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients - Zhuang et al., NeurIPS 2020.
- **Motivation**: achieve three goals -- fast convergence as in adaptive methods, good generalization as in SGD, and training stability.

- **Main idea**: adapt the step-size according to the "belief" in the current gradient direction -- the moving average (EMA) of the noisy gradient serves as the prediction of the gradient at the next time step; take a small step if the prediction and current gradient have a large difference and otherwise take a large step.

```
Algorithm 1: Adam Optimizer
Initialize $\theta_0, m_0 \leftarrow 0$ , $v_0 \leftarrow 0, t \leftarrow 0$
While $\theta_t$ not converged
    $t \leftarrow t+1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
    Bias Correction
        $\widehat{m_t} \leftarrow \frac{m_t}{1-\beta_1^t}, \widehat{v_t} \leftarrow \frac{v_t}{1-\beta_2^t}$
    Update
        $\theta_t \leftarrow \prod_{\mathcal{F},\sqrt{\widehat{v_t}}} \left(\theta_{t-1} - \frac{\alpha \widehat{m_t}}{\sqrt{\widehat{v_t}}+\epsilon}\right)$
```

```
Algorithm 2: AdaBelief Optimizer
Initialize $\theta_0, m_0 \leftarrow 0$ , $s_0 \leftarrow 0, t \leftarrow 0$
While $\theta_t$ not converged
    $t \leftarrow t+1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
    $s_t \leftarrow \beta_2 s_{t-1} + (1-\beta_2)(g_t - m_t)^2 + \epsilon$
    Bias Correction
        $\widehat{m_t} \leftarrow \frac{m_t}{1-\beta_1^t}, \widehat{s_t} \leftarrow \frac{s_t}{1-\beta_2^t}$
    Update
        $\theta_t \leftarrow \prod_{\mathcal{F},\sqrt{\widehat{s_t}}} \left(\theta_{t-1} - \frac{\alpha \widehat{m_t}}{\sqrt{\widehat{s_t}}+\epsilon}\right)$
```



- Website, PyTorch Code

MADGRAD

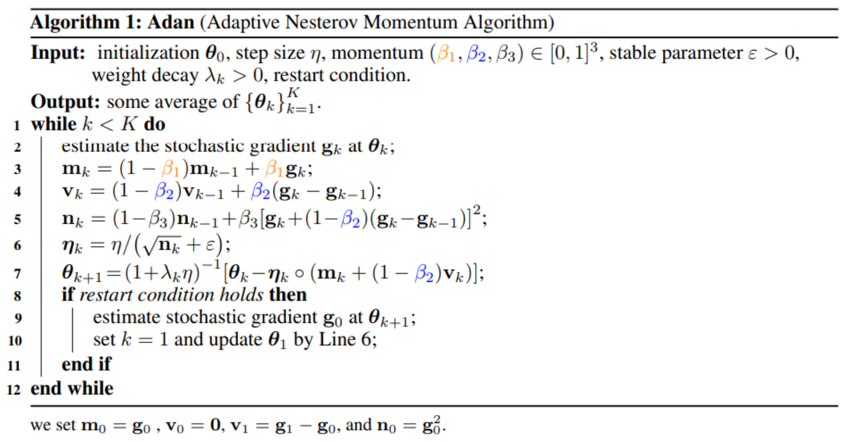
- Introduced in Adaptivity without Compromise: A Momentumized, Adaptive, Dual Averaged Gradient Method for Stochastic Optimization - Aaron Defazio and Samy Jelassi (META AI Research), JMLR 2022.
- **Motivation**: combine adaptivity with strong generalization performance.
- **Main idea**: based the dual averaging formulation of AdaGrad, combined with momentum and adaptivity to achieve the generalization performance of SGD and the fast convergence of Adam.

```
Algorithm 1 MADGRAD
Require: $\gamma_k$ stepsize sequence, $c_k$ momentum sequence, initial point $x_0$, epsilon $\epsilon$
 1: $s_0 : d = 0, \nu_0 : d = 0$
 2: for $k = 0, \ldots, T$ do
 3:     Sample $\xi_k$ and set $g_k = \nabla f(x_k, \xi_k)$
 4:     $\lambda_k = \gamma_k \sqrt{k+1}$
 5:     $s_{k+1} = s_k + \lambda_k g_k$
 6:     $\nu_{k+1} = \nu_k + \lambda_k (g_k \circ g_k)$
 7:
            $$z_{k+1} = x_0 - \frac{1}{\sqrt[3]{\nu_{k+1}} + \epsilon} \circ s_{k+1}$$
 8:     $x_{k+1} = (1 - c_{k+1}) x_k + c_{k+1} z_{k+1}.$
 9: end for
10: return $x_T$
```

- Website, PyTorch Code.
  - Notes: GPU-only. Typically, the same learning rate schedule that is used for SGD or Adam may be used. MADGRAD requires less weight decay than other methods, often as little as zero. Momentum values used for SGD or Adam's β_1 should work here.

Adan

- Introduced in Adan: Adaptive Nesterov Momentum Algorithm for Faster Optimizing Deep Models - Xie et al., 2022.
- **Motivation**: improve the deep learning model training speed by utilizing an accurate and stable estimation of the gradient moments.
- **Main idea**: a new Nesterov momentum estimation (NME) method, which avoids the extra overhead of computing gradient at the extrapolation point and used to estimate the gradient's first-order and second-order moments in adaptive gradient algorithms for convergence acceleration.



Algorithm 1: Adan (Adaptive Nesterov Momentum Algorithm)

```
Input: initialization $\boldsymbol{\theta}_0$, step size $\eta$, momentum $(\beta_1, \beta_2, \beta_3) \in [0,1]^3$, stable parameter $\varepsilon > 0$,
       weight decay $\lambda_k > 0$, restart condition.
Output: some average of $\{\boldsymbol{\theta}_k\}_{k=1}^{K}$.
1  while $k < K$ do
2      estimate the stochastic gradient $\mathbf{g}_k$ at $\boldsymbol{\theta}_k$;
3      $\mathbf{m}_k = (1-\beta_1)\mathbf{m}_{k-1} + \beta_1 \mathbf{g}_k$;
4      $\mathbf{v}_k = (1-\beta_2)\mathbf{v}_{k-1} + \beta_2(\mathbf{g}_k - \mathbf{g}_{k-1})$;
5      $\mathbf{n}_k = (1-\beta_3)\mathbf{n}_{k-1} + \beta_3[\mathbf{g}_k + (1-\beta_2)(\mathbf{g}_k - \mathbf{g}_{k-1})]^2$;
6      $\boldsymbol{\eta}_k = \eta / (\sqrt{\mathbf{n}_k} + \varepsilon)$;
7      $\boldsymbol{\theta}_{k+1} = (1+\lambda_k \eta)^{-1}[\boldsymbol{\theta}_k - \boldsymbol{\eta}_k \circ (\mathbf{m}_k + (1-\beta_2)\mathbf{v}_k)]$;
8      if *restart condition holds* then
9          estimate stochastic gradient $\mathbf{g}_0$ at $\boldsymbol{\theta}_{k+1}$;
10         set $k = 1$ and update $\boldsymbol{\theta}_1$ by Line 6;
11     end if
12 end while
```

we set $\mathbf{m}_0 = \mathbf{g}_0$, $\mathbf{v}_0 = \mathbf{0}$, $\mathbf{v}_1 = \mathbf{g}_1 - \mathbf{g}_0$, and $\mathbf{n}_0 = \mathbf{g}_0^2$.

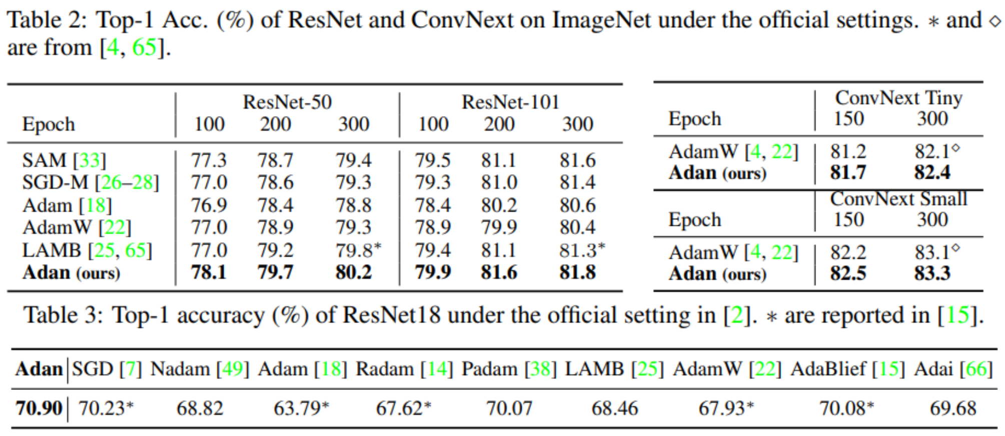


Table 2: Top-1 Acc. (%) of ResNet and ConvNext on ImageNet under the official settings. $*$ and $\diamond$ are from [4, 65].

	ResNet-50			ResNet-101		
Epoch	100	200	300	100	200	300
SAM [33]	77.3	78.7	79.4	79.5	81.1	81.6
SGD-M [26–28]	77.0	78.6	79.3	79.3	81.0	81.4
Adam [18]	76.9	78.4	78.8	78.4	80.2	80.6
AdamW [22]	77.0	78.9	79.3	78.9	79.9	80.4
LAMB [25, 65]	77.0	79.2	79.8*	79.4	81.1	81.3*
Adan (ours)	**78.1**	**79.7**	**80.2**	**79.9**	**81.6**	**81.8**

	ConvNext Tiny	
Epoch	150	300
AdamW [4, 22]	81.2	82.1◇
Adan (ours)	**81.7**	**82.4**
	ConvNext Small	
Epoch	150	300
AdamW [4, 22]	82.2	83.1◇
Adan (ours)	**82.5**	**83.3**

Table 3: Top-1 accuracy (%) of ResNet18 under the official setting in [2]. $*$ are reported in [15].

Adan	SGD [7]	Nadam [49]	Adam [18]	Radam [14]	Padam [38]	LAMB [25]	AdamW [22]	AdaBlief [15]	Adai [66]
70.90	70.23*	68.82	63.79*	67.62*	70.07	68.46	67.93*	70.08*	69.68

- PyTorch Code.
  - Notes: Adan has a slightly higher GPU memory cost than Adam/AdamW on a single node.

Recommended Videos

Warning!

- These videos do not replace the lectures and tutorials.
- Please use these to get a better understanding of the material, and not as an alternative to the written material.

Video By Subject

- Gradient Descent - Gradient Descent, Step-by-Step
  - Mathematics of Gradient Descent - Intelligence and Learning
- Stochastic Gradient Descent - Stochastic Gradient Descent, Clearly Explained
- Momentum - Gradient Descent With Momentum (C2W2L06)
- RMSProp - RMSProp (C2W2L07)
- Adam - Adam Optimization Algorithm (C2W2L08)
- Learning Rate Decay - Learning Rate Decay (C2W2L09)
- Momentum, Adagrad, RMSProp, Adam - UC Berkeley, STAT 157 - Momentum, Adagrad, RMSProp, Adam
- AdaBelief - AdaBelief Optimizer: Theory and Practical Guidelines

Credits

- Icons made by Becris from www.flaticon.com
- Icons from Icons8.com - https://icons8.com
- Datasets from Kaggle - https://www.kaggle.com/
- Examples and code snippets were taken from "Hands-On Machine Learning with Scikit-Learn and TensorFlow"
- CS231n: Convolutional Neural Networks for Visual Recognition
- Deep Learning Wizard -Learning Rate Scheduling
- Understanding Nesterov Momentum (NAG)
- Sebastian Ruder - An overview of gradient descent optimization algorithms