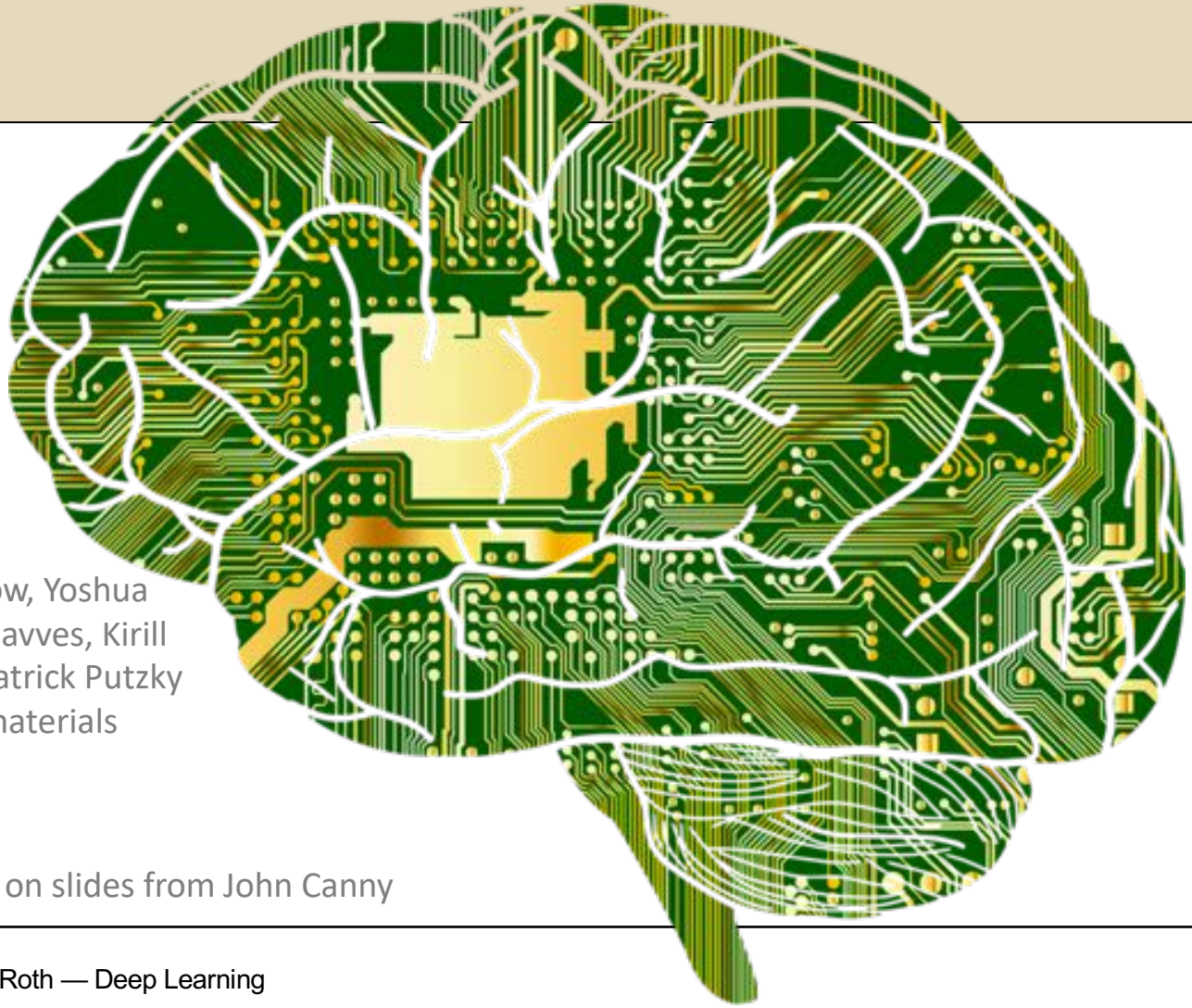
Deep Learning

Architectures and Methods:

Optimization

TECHNISCHE UNIVERSITÄT DARMSTADT



Thanks to John Canny, Ian Goodfellow, Yoshua Bengio, Aaron Courville, Efstratios Gavves, Kirill Gavrilyuk, Berkay Kicanaoglu, and Patrick Putzky and many others for making their materials publically available.

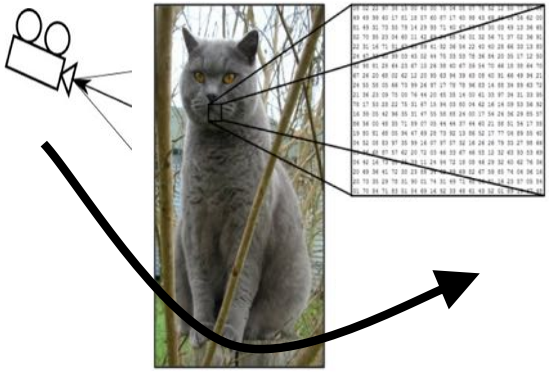
The present slides are mainly based on slides from John Canny

Recall from the skipped videos …

TECHNISCHE UNIVERSITÄT DARMSTADT

Challenges in Visual Recognition

Camera pose



Illumination

Deformation

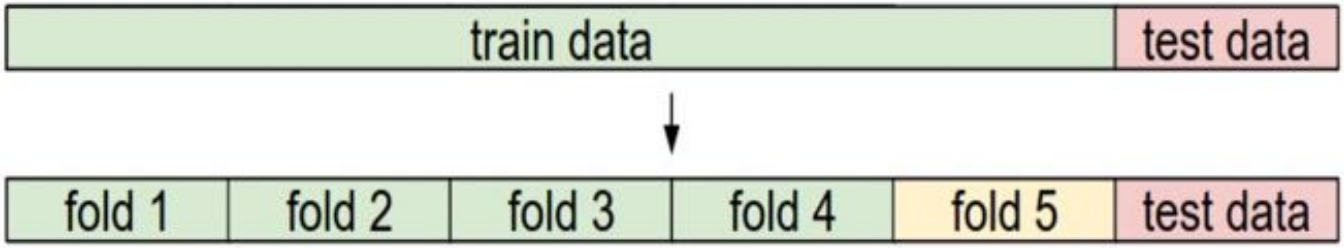
Occlusion

Background clutter

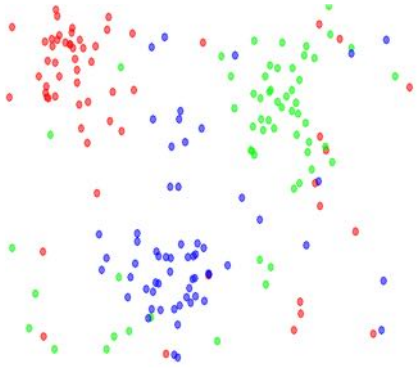
Intraclass variation

Recall from the skipped videos …

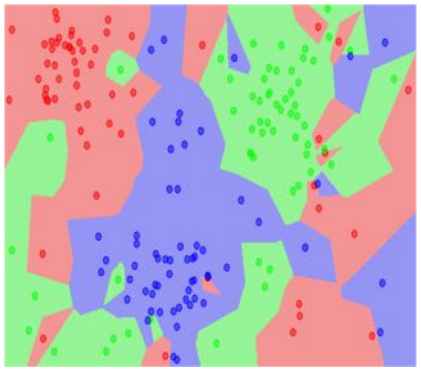
data-driven approach, kNN



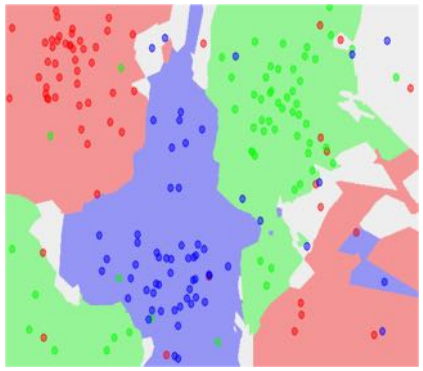
the data

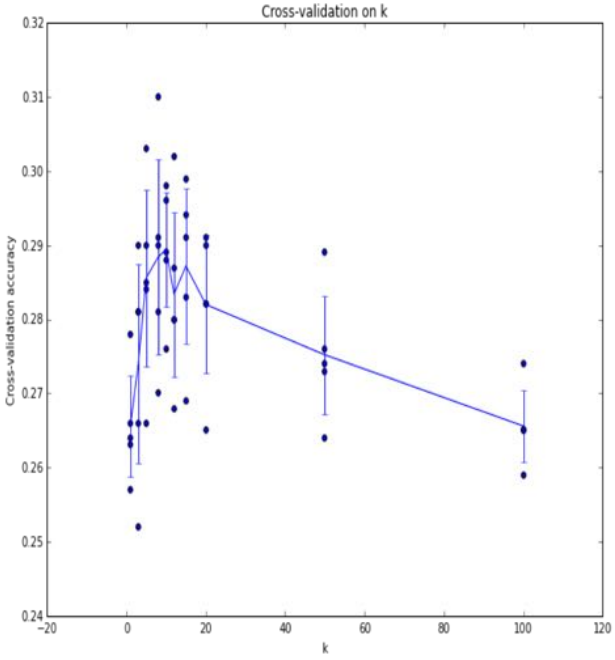


NN classifier



5-NN classifier





Parametric approach: **Linear classifier**

TECHNISCHE UNIVERSITÄT DARMSTADT

$$f(x, W) = Wx$$

3072x1

(+b) 10x1

bias

10x1

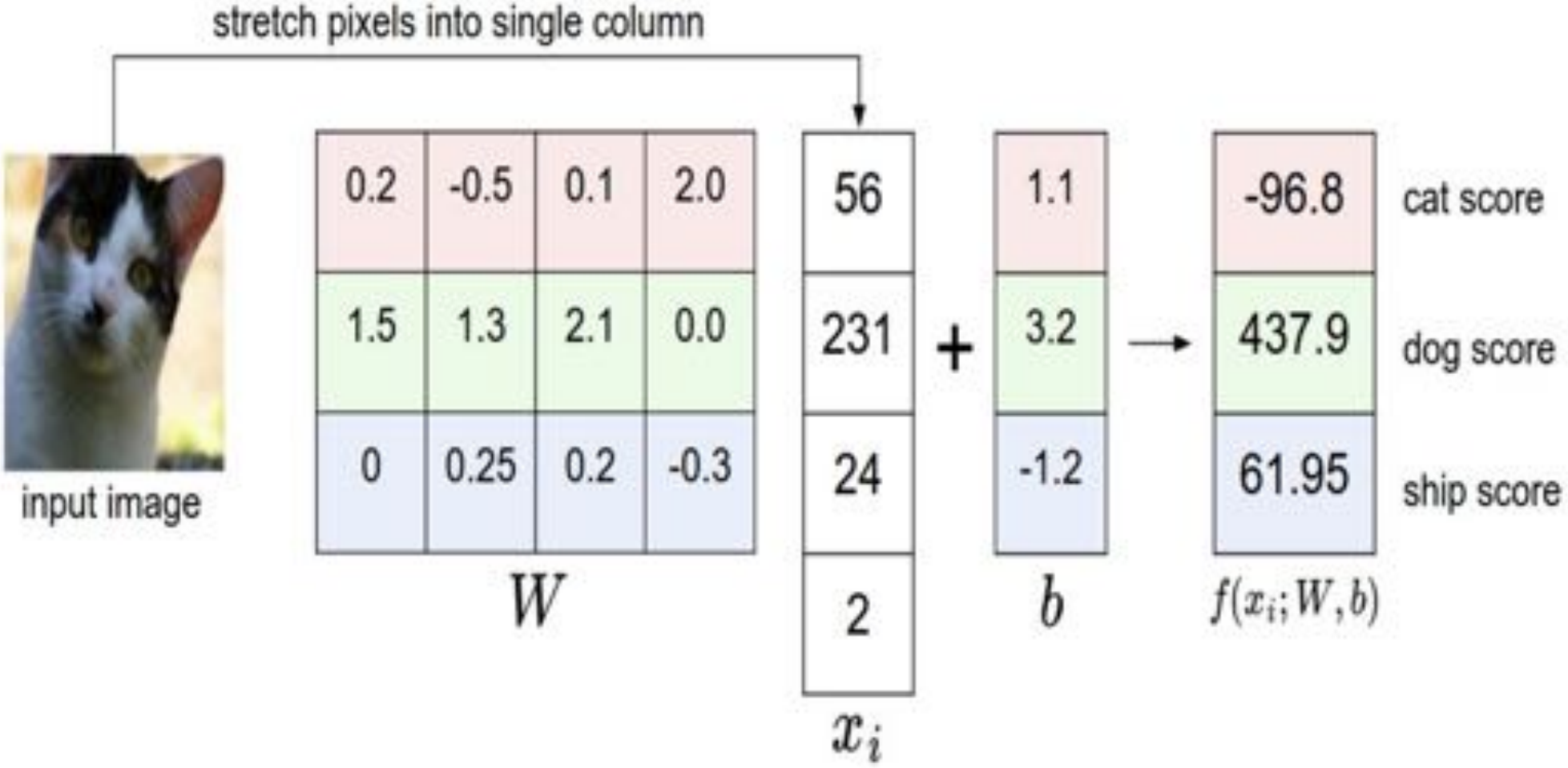
10x3072

[32x32x3]
array of numbers 0...1

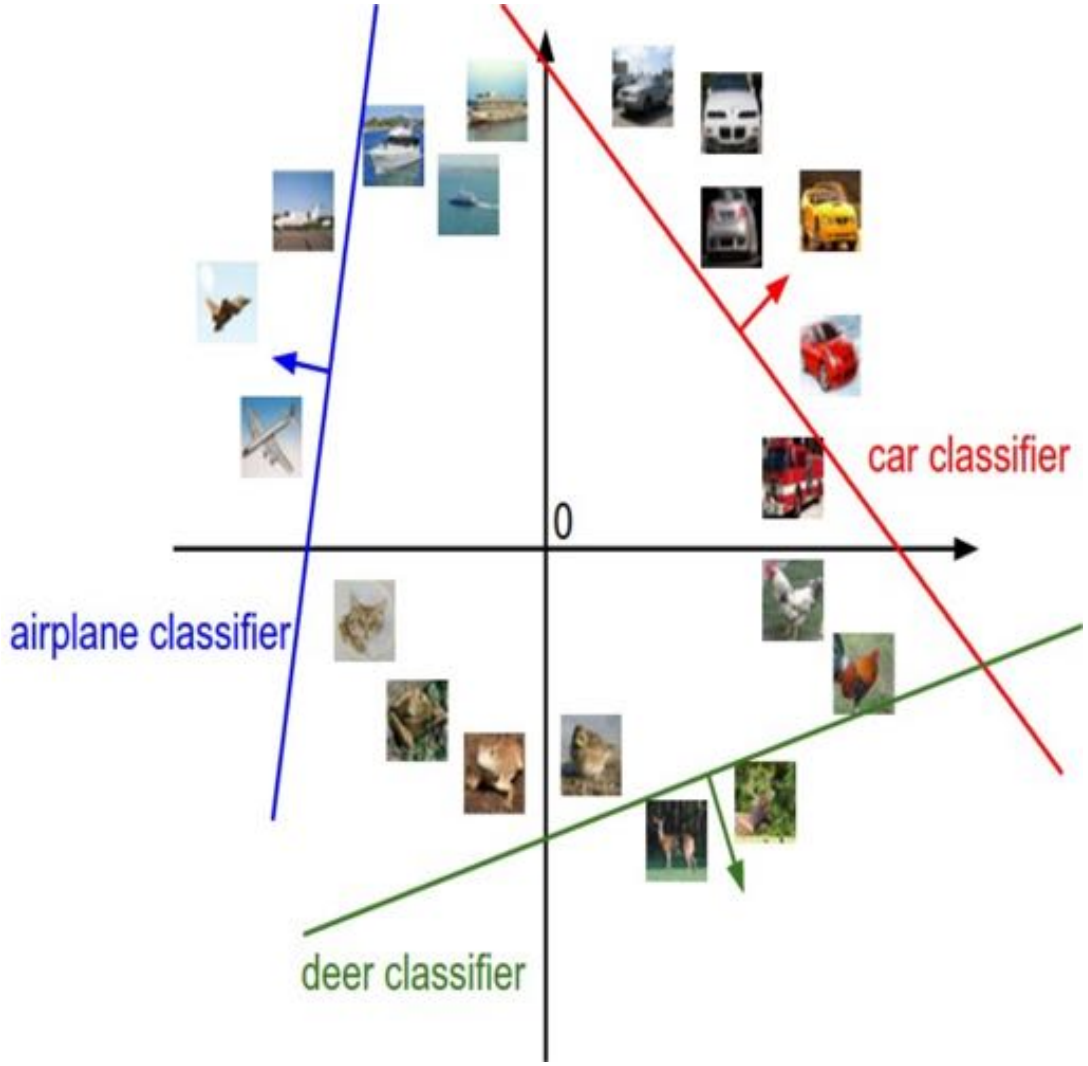
10 numbers, indicating class scores

parameters, or “weights”

Example image with 4 pixels, and 3 classes (cat/dog/ship)



Interpreting a Linear Classifier



$$f(x_i, W, b) = Wx_i + b$$

[32x32x3]
array of numbers 0...1
(3072 numbers total)

Linear Classifier

$$f(x_i, W, b) = Wx_i + b$$

Example trained weights of a linear classifier trained on CIFAR-10:

Bias and Variance

TECHNISCHE UNIVERSITÄT DARMSTADT

How do bias and variance for the linear classifier compare to kNN:

Bias: ? Higher for linear classifier. Will have errors on images which aren't linearly separable from other classes.

Variance: ? Lower: linear output is a weighted sum of all the input pixels.

In particular: Loss Functions

- We have some dataset of (x,y)
- We have a **score function:**
- We have a **loss function**:

e.g.

$$s = f(x; W) = Wx$$

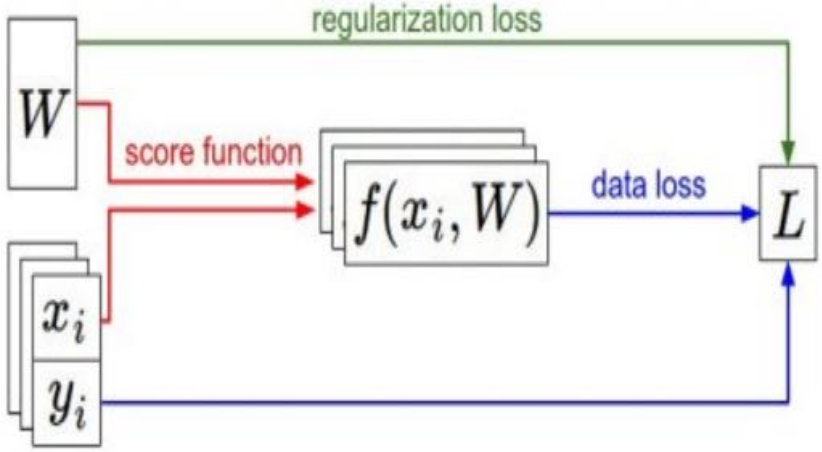
Softmax

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

SVM

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + R(W)$$ Full loss



Softmax Classifier (Multinomial Logistic Regression)

cat **3.2**

car 5.1

frog -1.7

Scores = unnormalized log prob. of the classes

$$P(Y = k|X = x_i) = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \quad \text{where} \quad s = f(x, W)$$

Want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

$$L_i = -\log P(Y = y_i|X = x_i)$$

In summary:

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Weight Regularization

TECHNISCHE UNIVERSITÄT DARMSTADT

λ= regularization strength (hyperparameter)

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max\left(0, f(x_i, W)_j - f(x_i, W)_{y_i} + 1\right) + \lambda R(W)$$

In common use:

L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$
L1 regularization	$R(W) = \sum_k \sum_l \lvert W_{k,l} \rvert$
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l} \rvert$

Dropout (will see later)

Max norm regularization (might see later)

Today: Optimization

Strategy #1: A first very bad idea solution: **Random search**

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in xrange(1000):
  W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
  loss = L(X_train, Y_train, W) # get the loss over the entire training set
  if loss < bestloss: # keep track of the best solution
    bestloss = loss
    bestW = W
  print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (trunctated: continues for 1000 lines)
```


Lets see how well this works on the test set...

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test examples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

15.5% accuracy! not bad!
(State of the art is ~95%)

TECHNISCHE UNIVERSITÄT DARMSTADT

TECHNISCHE UNIVERSITÄT DARMSTADT

Strategy #2: Follow the slope

TECHNISCHE UNIVERSITÄT DARMSTADT

In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives).

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:	**W + h** (first dim)**:**	**gradient dW:**
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…]	[0.34 + **0.0001**, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…]	[?, ?, ?, ?, ?, ?, ?, ?, ?,…]
loss 1.25347	**loss 1.25322**	

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]
loss 1.25347

W + h (first dim)**:**

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]
loss 1.25322

gradient dW:

[**-2.5**,
?,
?,
?,
?,

(1.25322 - 1.25347)/0.0001
= -2.5

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

current W:	**W + h** (second dim)**:**	**gradient dW:**
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…] **loss 1.25347**	[0.34, -1.11 + **0.0001**, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…] **loss 1.25353**	[-2.5, ?, ?, ?, ?, ?, ?, ?, ?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

W + h (second dim)**:**

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25353

gradient dW:

[-2.5,
0.6,
?,
?,
?,
?,

(1.25353 - 1.25347)/0.0001
= 0.6

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

current W:	**W + h** (third dim)**:**	**gradient dW:**
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…] **loss 1.25347**	[0.34, -1.11, 0.78 + **0.0001**, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,…] **loss 1.25347**	[-2.5, 0.6, ?, ?, ?, ?, ?, ?, ?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

W + h (third dim)**:**

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?,
?,
?

(1.25347 - 1.25347)/0.0001
= 0

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Evaluating the gradient numerically

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

```
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```


Evaluating the gradient numerically

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- approximate
- very slow to evaluate

```
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```


This is silly. The loss is just a function of W:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

This is silly. The loss is just a function of W:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

This is silly. The loss is just a function of W:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

Calculus

This is silly. The loss is just a function of W:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

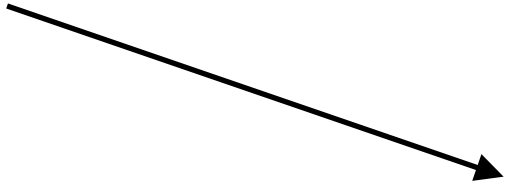
$$s = f(x; W) = Wx$$

$$\nabla_W L = \ldots$$

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

dW = ...
(some function data
and W)



gradient dW:

[-2.5,
0.6,
0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1,…]

In summary:

TECHNISCHE UNIVERSITÄT DARMSTADT

- Numerical gradient: approximate, slow, easy to write

- Analytic gradient: exact, fast, error-prone

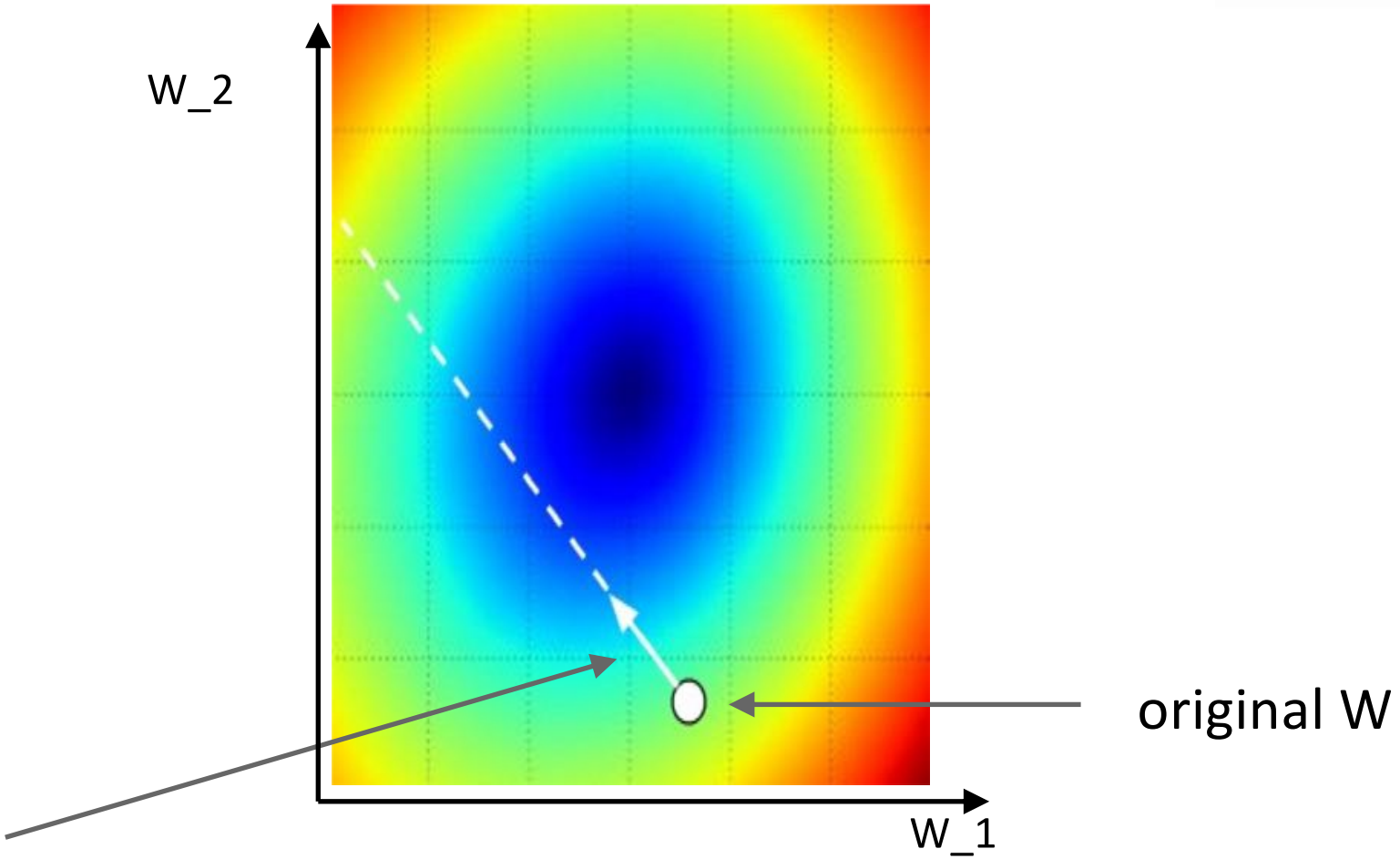
So

In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient check.**

Gradient Descent

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



W_2
W_1
original W
negative gradient direction

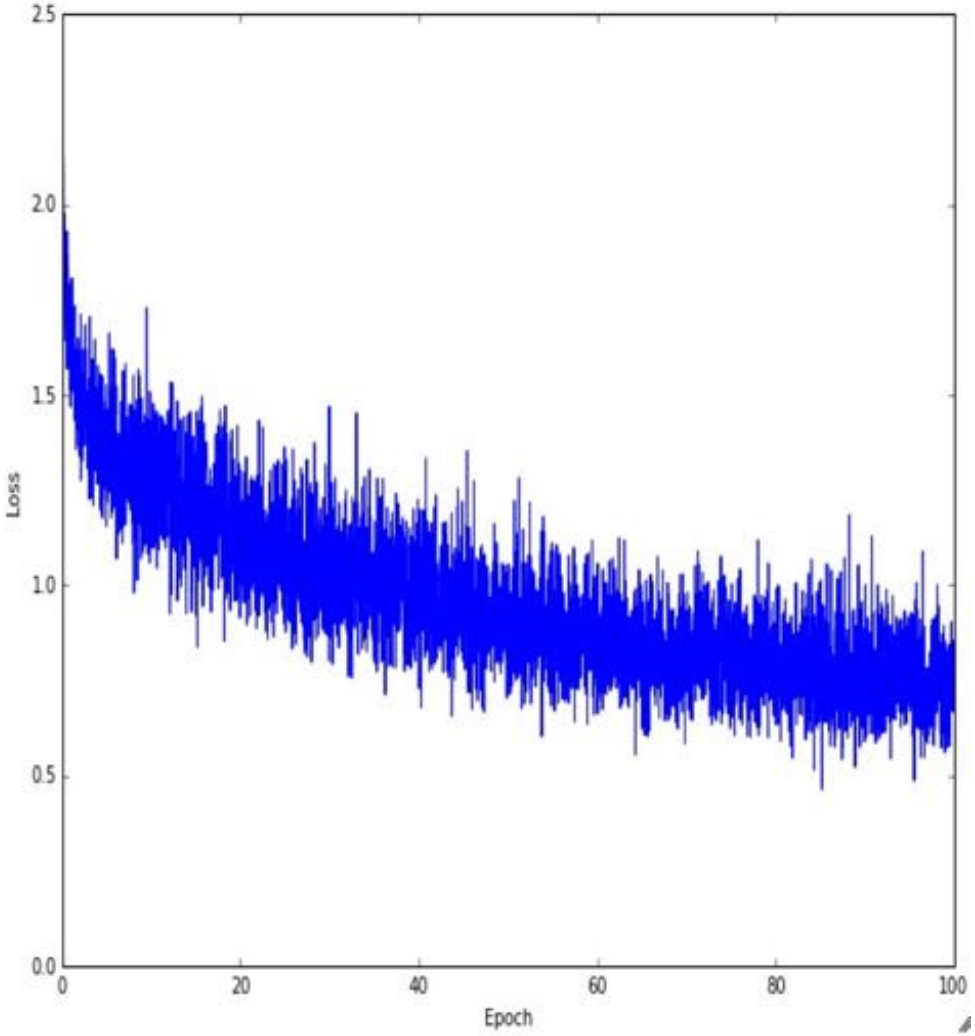
Mini-batch Gradient Descent

TECHNISCHE UNIVERSITÄT DARMSTADT

only use a small portion of the training set to compute the gradient.

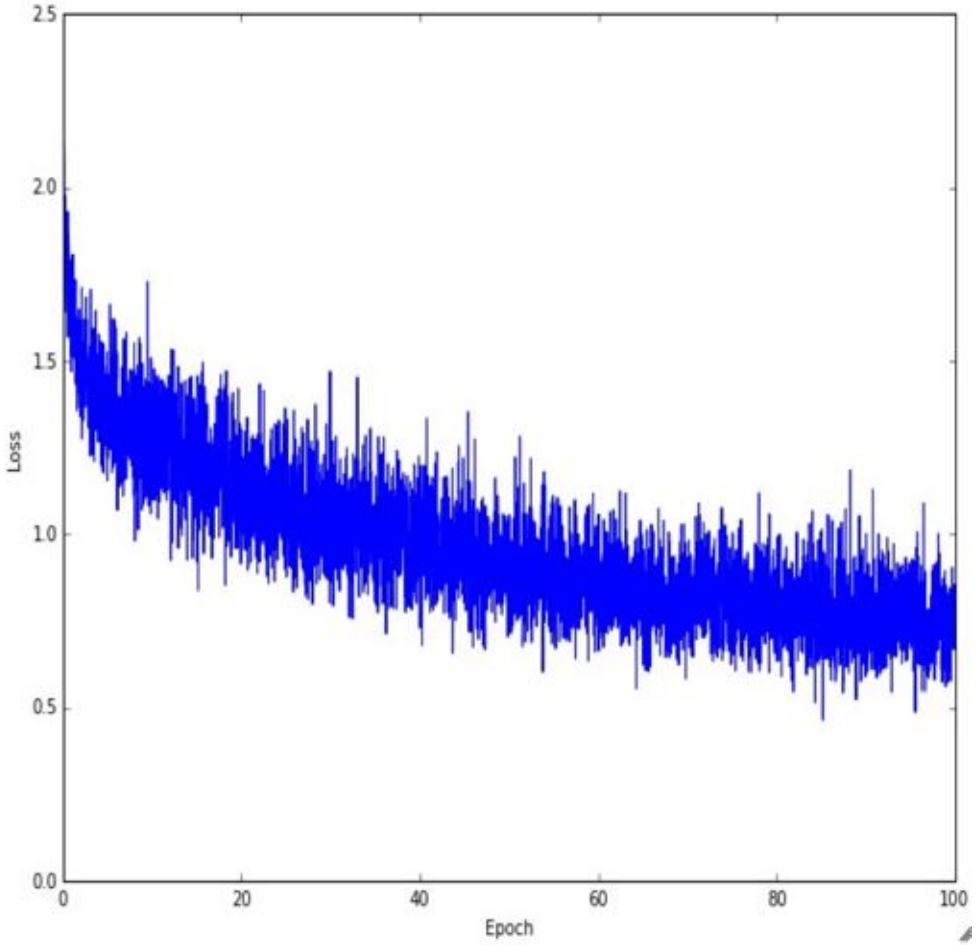
```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

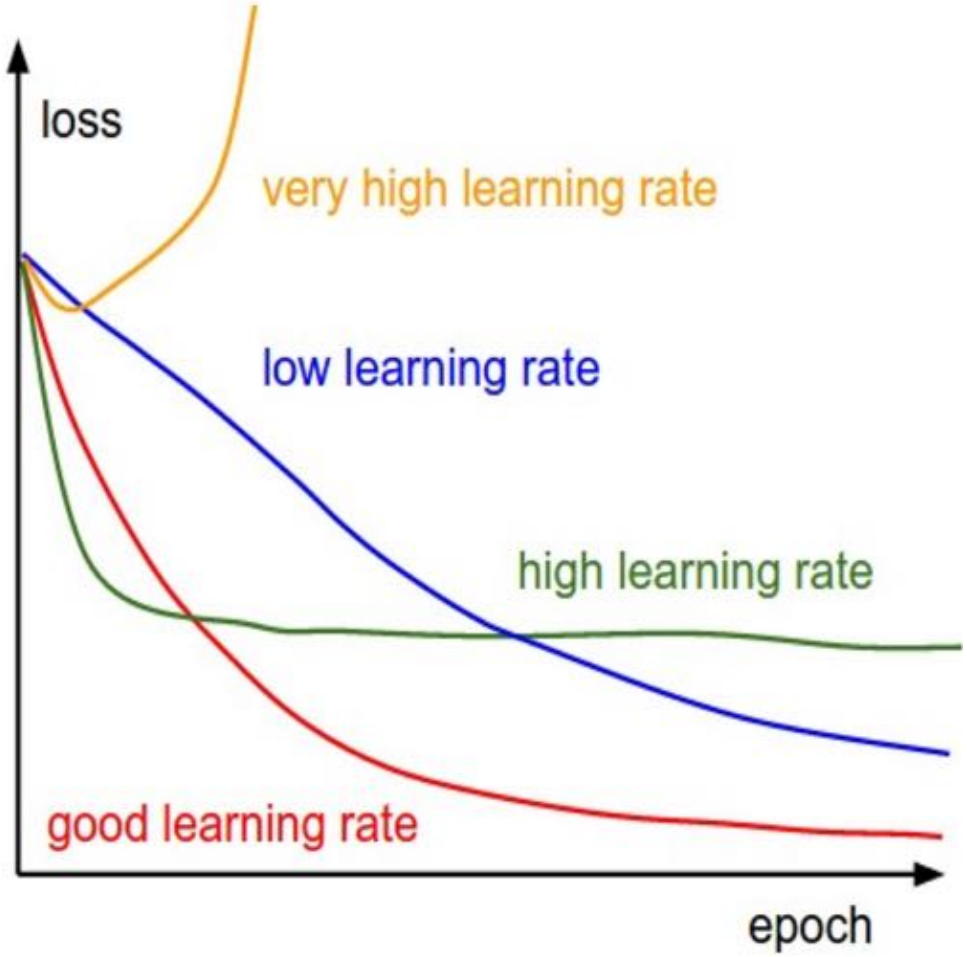
Common mini-batch sizes are
32/64/128 examples e.g. Krizhevsky
ILSVRC ConvNet used 256 examples



Example of optimization progress while training a neural network.

(Loss over mini-batches goes down over time.)


Loss
Epoch
0.0
0.5
1.0
1.5
2.0
2.5
0
20
40
60
80
100


loss
very high learning rate
low learning rate
high learning rate
good learning rate
epoch

Mini-batch Gradient Descent

TECHNISCHE UNIVERSITÄT DARMSTADT

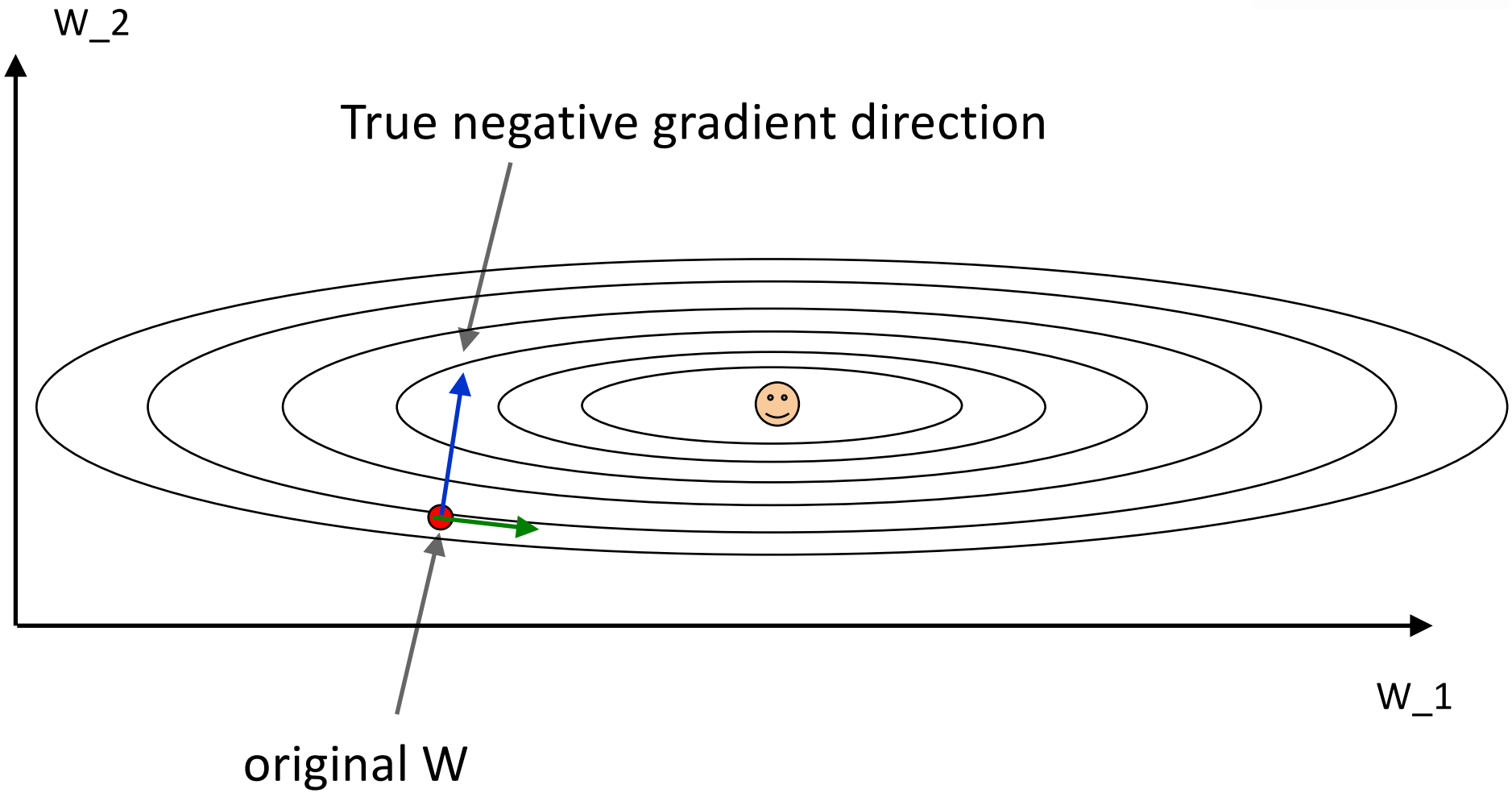
only use a small portion of the training set to compute the gradient.

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

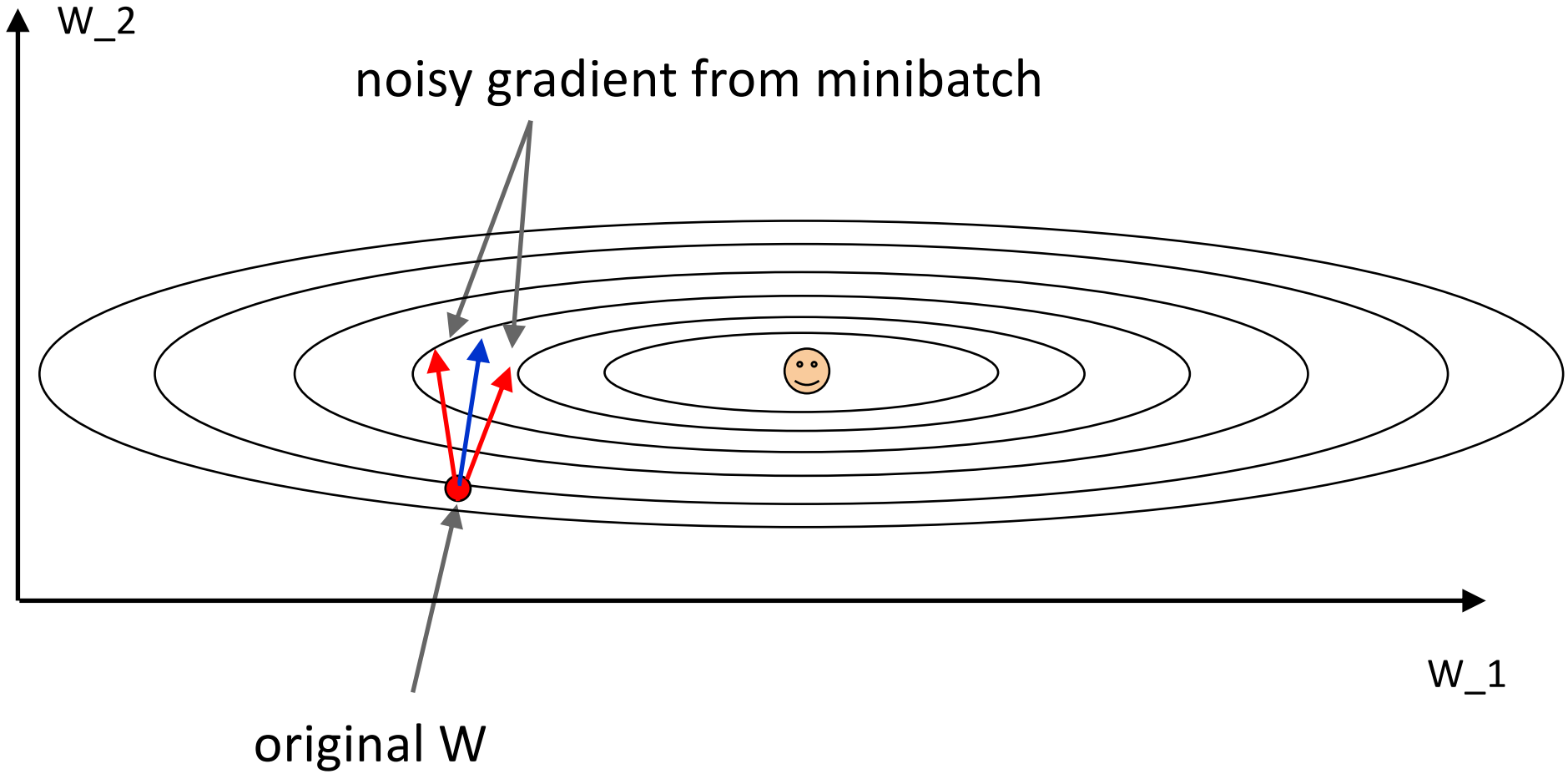
we will look at more fancy update formulas (momentum, Adagrad, RMSProp, Adam, …)

Common mini-batch sizes are 32/64/128 examples e.g. Krizhevsky ILSVRC ConvNet used 256 examples

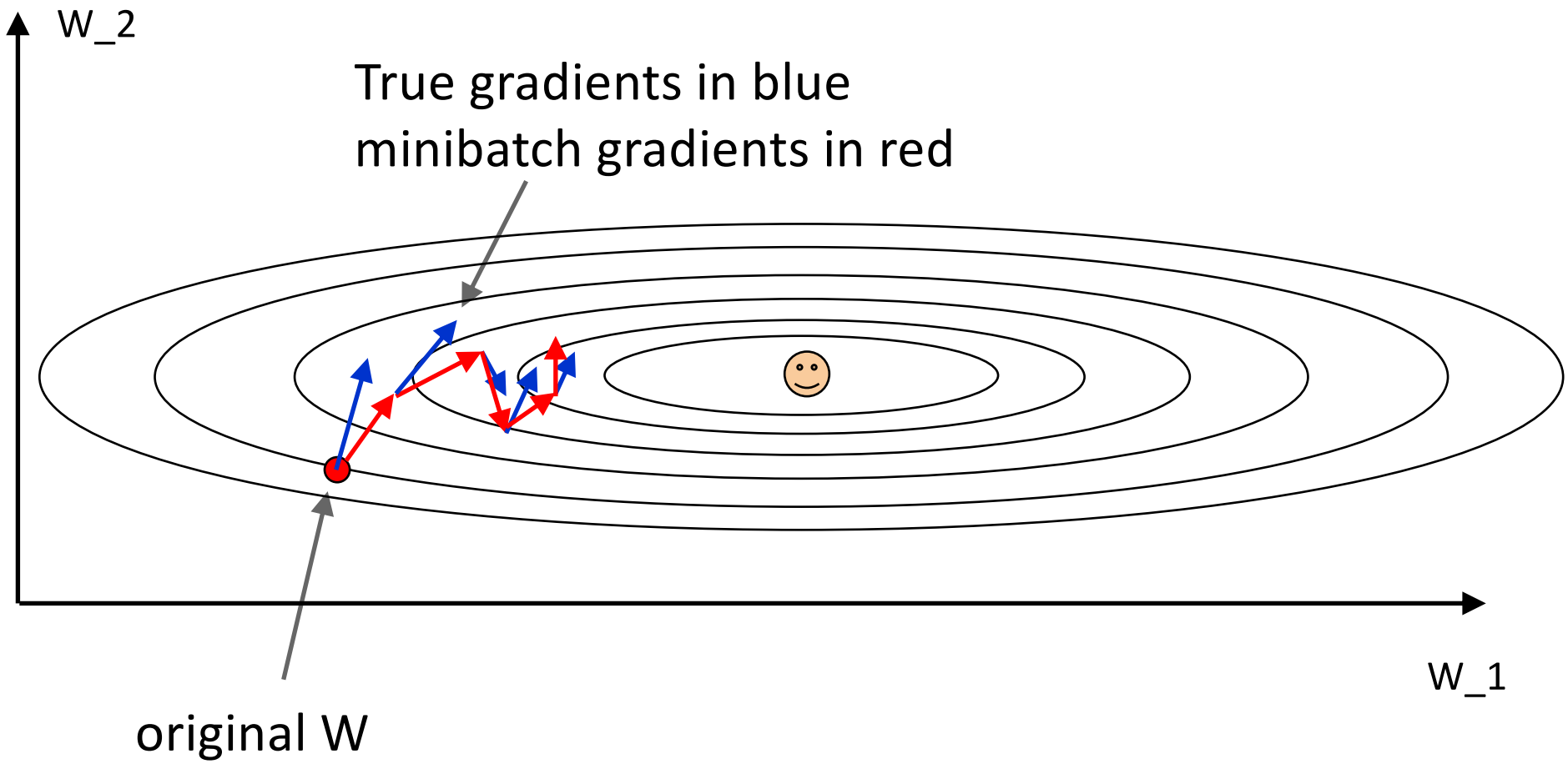
Minibatch updates



Stochastic Gradient

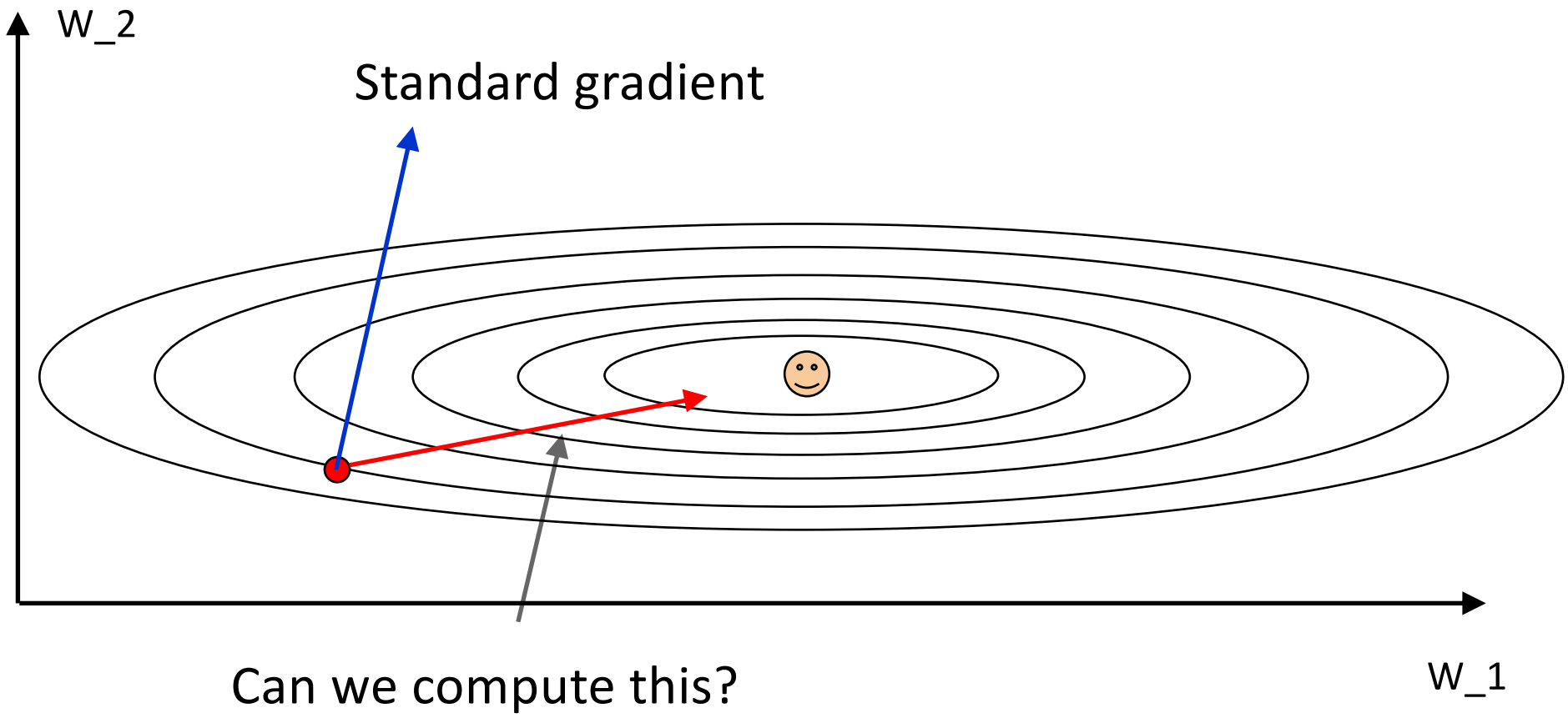


Stochastic Gradient Descent



Gradients are noisy but still make good progress on average

You might be wondering…



Newton's method for zeros of a function

Based on the Taylor Series for $f(x+h)$:

$$f(x+h) = f(x) + hf'(x) + O(h^2)$$

To find a zero of f, assume $f(x+h) = 0$, so

$$h \approx -\frac{f(x)}{f'(x)}$$

And as an iteration:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Newton's method for optima

TECHNISCHE UNIVERSITÄT DARMSTADT

For zeros of $f(x)$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

At a local optima, $f'(x) = 0$, so we use:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

If $f''(x)$ is constant (f is quadratic), then Newton's method finds the optimum in ***one step***.

More generally, Newton's method has ***quadratic converge.***

Newton's method for gradients:

TECHNISCHE UNIVERSITÄT DARMSTADT

To find an optimum of a function $f(x)$ for high-dimensional x, we want zeros of its gradient: $\nabla f(x) = 0$

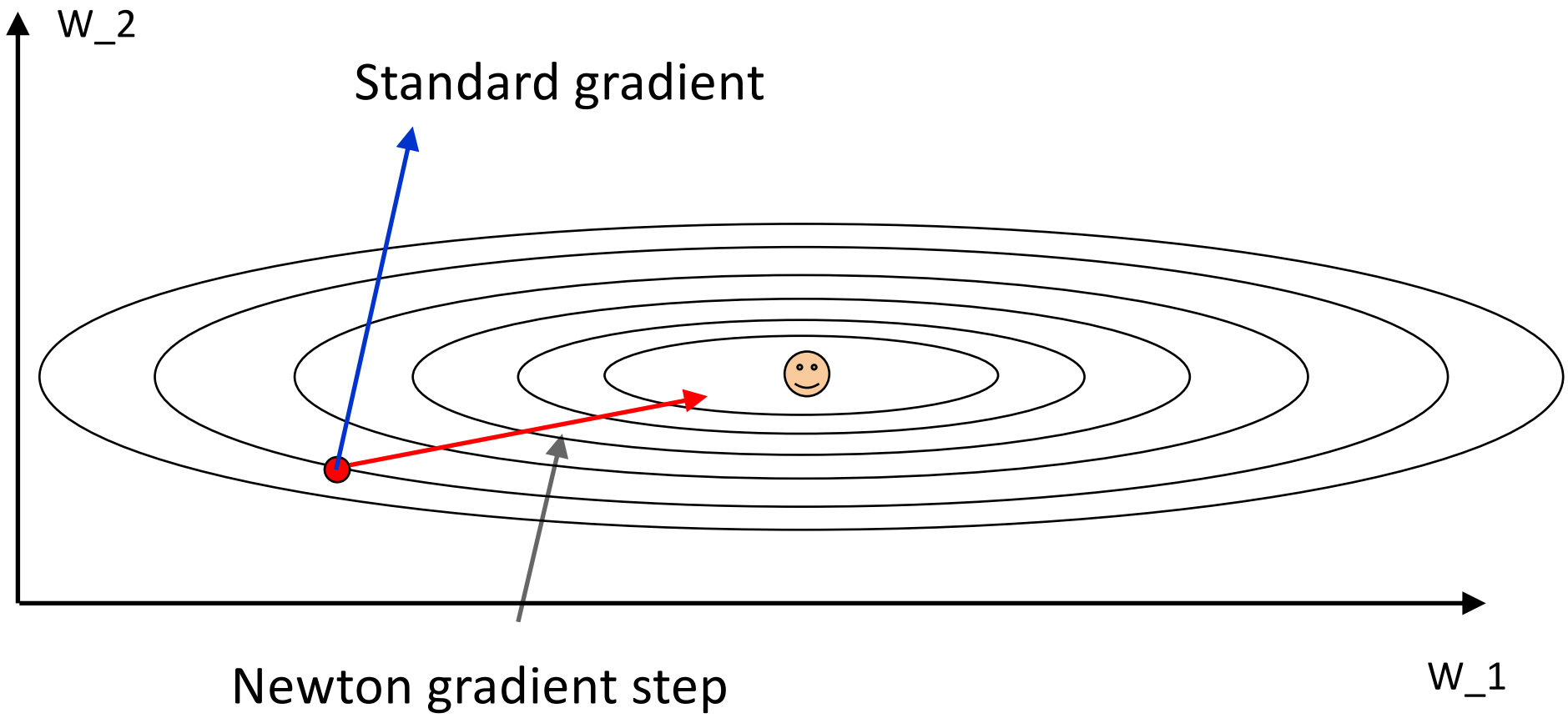
For zeros of $\nabla f(x)$ with a vector displacement h, Taylor's expansion is:

$$\nabla f(x + h) = \nabla f(x) + h^T H_f(x) + O(||h||^2)$$

Where H_f is the Hessian matrix of second derivatives of f. The update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

Newton step



Newton's method for gradients:

TECHNISCHE UNIVERSITÄT DARMSTADT

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

Converges very fast, but rarely used in DL. Why?

Newton's method for gradients:

TECHNISCHE UNIVERSITÄT DARMSTADT

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1}\nabla f(x_n)$$

Converges very fast, but rarely used in DL. Why?

Too expensive: if x_n has dimension M, the Hessian $H_f(x_n)$ has dimension M^2 and takes $O(M^3)$ time to invert.

Newton's method for gradients:

TECHNISCHE UNIVERSITÄT DARMSTADT

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1}\nabla f(x_n)$$

Converges very fast, but rarely used in DL. Why?

Too expensive: if x_n has dimension M, the Hessian $H_f(x_n)$ has dimension M^2 and takes $O(M^3)$ time to invert.

Too unstable: it involves a high-dimensional matrix inverse, which has poor numerical stability. The Hessian may even be singular.

L-BFGS

TECHNISCHE UNIVERSITÄT DARMSTADT

There is an approximate Newton method that addresses these issues called L-BGFS, (Limited memory BFGS). BFGS is Broyden-Fletcher-Goldfarb-Shanno.

Idea: compute a low-dimensional approximation of $H_f(x_n)^{-1}$ directly.

Expense: use a k-dimensional approximation of $H_f(x_n)^{-1}$, Size is O(kM), cost is O(k^2 M).

Stability: much better. Depends on largest singular values of $H_f(x_n)$.

Convergence Nomenclature

TECHNISCHE UNIVERSITÄT DARMSTADT

Quadratic Convergence: error decreases quadratically (Newton's Method):

$$\epsilon_{n+1} \propto \epsilon_n^2 \quad \text{where} \quad \epsilon_n = x_{opt} - x_n$$

Linearly Convergence: error decreases linearly:

$$\epsilon_{n+1} \leq \mu \epsilon_n \quad \text{where } \mu \text{ is the } \textit{rate of convergence}.$$

SGD: ??

Convergence Behavior

TECHNISCHE UNIVERSITÄT DARMSTADT

Quadratic Convergence: error decreases quadratically

$$\epsilon_{n+1} \propto \epsilon_n^2 \quad \text{so} \quad \epsilon_n \text{ is } O\left(\mu^{2^n}\right)$$

Linearly Convergence: error decreases linearly:

$$\epsilon_{n+1} \leq \mu \epsilon_n \quad \text{so} \quad \epsilon_n \text{ is } O(\mu^n).$$

SGD: If learning rate is adjusted as 1/n, then (Nemirofski) ϵ_n is O(1/n).

Convergence Comparison

TECHNISCHE UNIVERSITÄT DARMSTADT

Quadratic Convergence: ϵ_n is $O(\mu^{2^n})$, time $\log(\log(\epsilon))$

Linearly Convergence: ϵ_n is $O(\mu^n)$, time $\log(\epsilon)$

SGD: ϵ_n is O(1/n), time $1/\epsilon$

SGD is terrible compared to the others. Why is it used?

Convergence Comparison

TECHNISCHE UNIVERSITÄT DARMSTADT

Quadratic Convergence: ϵ_n is $O(\mu^{2^n})$, time $\log(\log(\epsilon))$

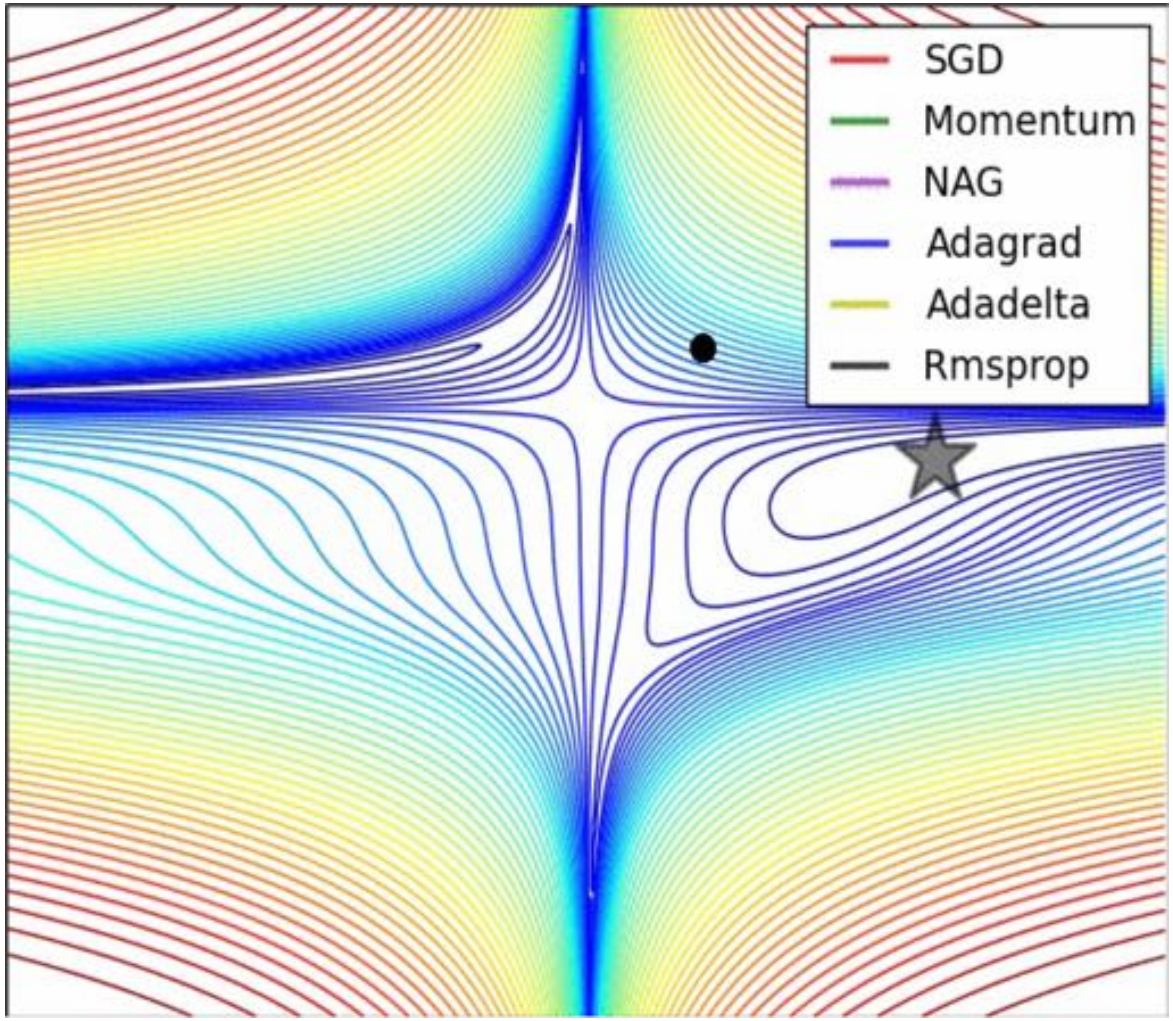
Linearly Convergence: ϵ_n is $O(\mu^n)$, time $\log(\epsilon)$

SGD: ϵ_n is O(1/n), time $1/\epsilon$

SGD is ***good enough*** for machine learning applications.
Remember "n" for SGD is minibatch count.

After 1 million updates, ϵ is order O(1/n) which is approaching floating point single precision.

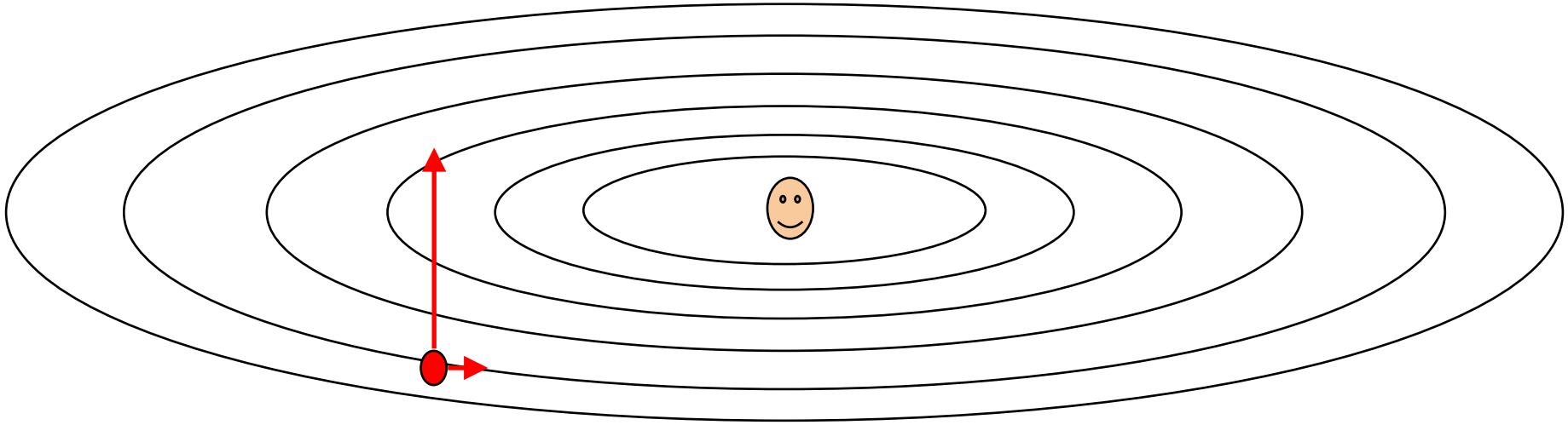
The effects of different update formulas



(image credits to Alec Radford)

Another approach to poor gradients:

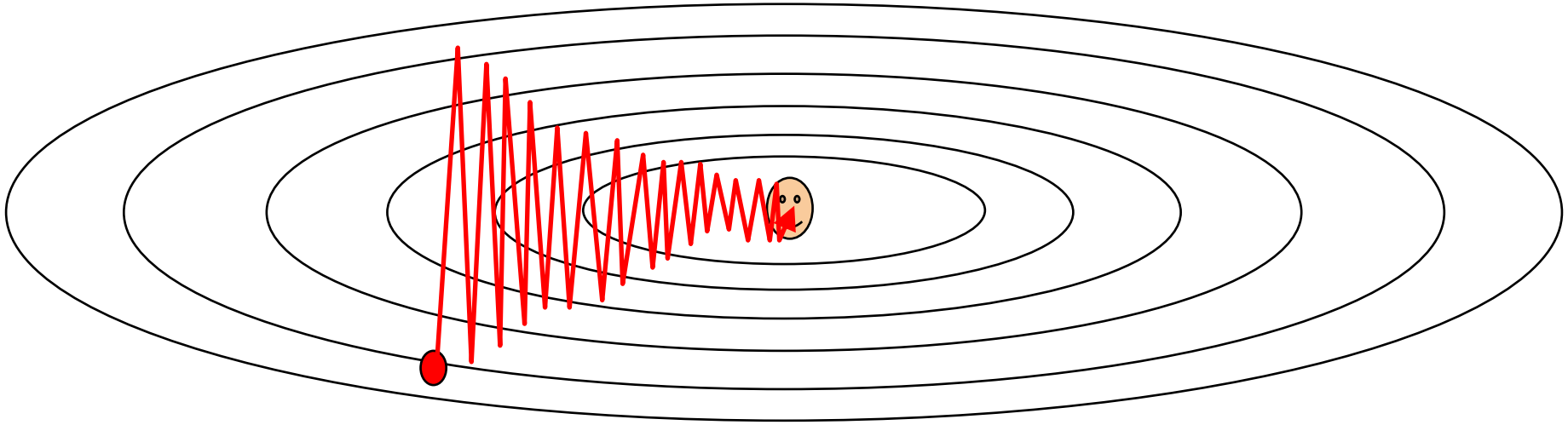
TECHNISCHE UNIVERSITÄT DARMSTADT



Q: What is the trajectory along which we converge towards the minimum with SGD?

Another approach to poor gradients:

TECHNISCHE UNIVERSITÄT DARMSTADT



Q: What is the trajectory along which we converge towards the minimum with SGD? very slow progress along flat direction, jitter along steep one

Momentum update

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# Gradient descent update
x += - learning_rate * dx
```

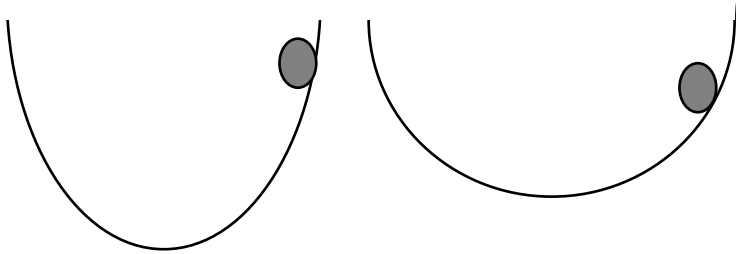
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

Physical interpretation as ball rolling down the loss function + friction (mu coefficient). mu = usually ~0.5, 0.9, or 0.99 (Sometimes annealed over time, e.g. from 0.5 -> 0.99)

Momentum update

TECHNISCHE UNIVERSITÄT DARMSTADT

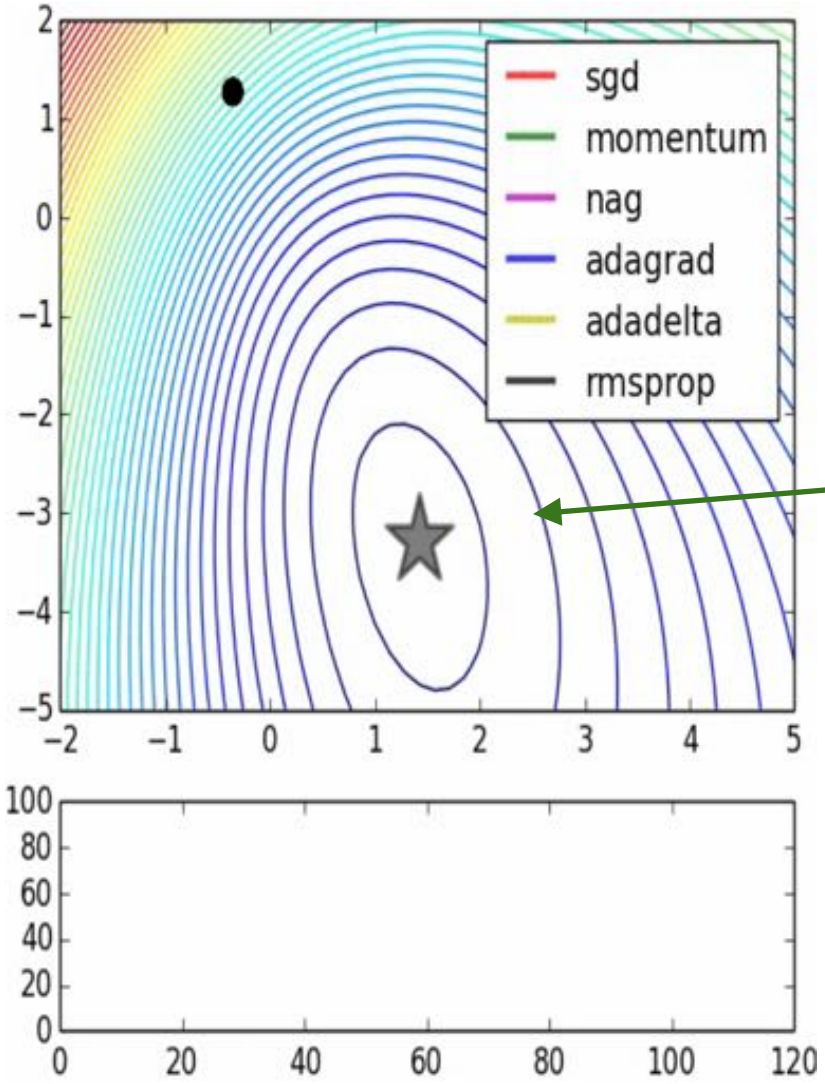
```
# Gradient descent update
x += - learning_rate * dx
```



```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

SGD vs Momentum



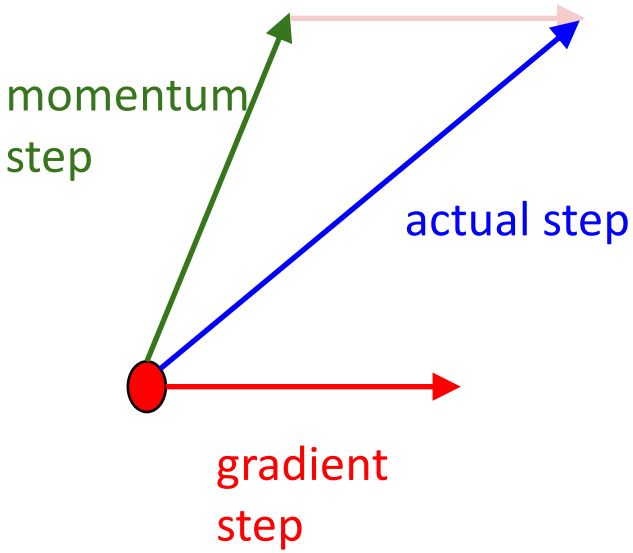
notice momentum overshooting the target, but overall getting to the minimum much faster than vanilla SGD.

Nesterov Momentum update

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

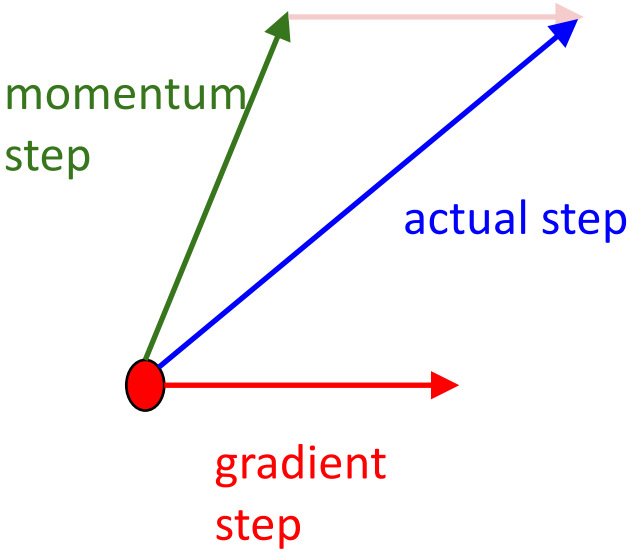
Ordinary momentum update:



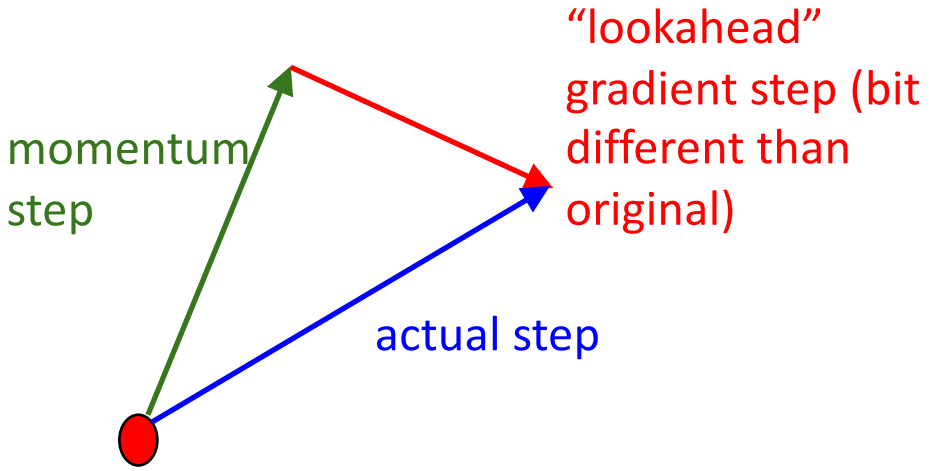
Nesterov Momentum update

TECHNISCHE UNIVERSITÄT DARMSTADT

Momentum update

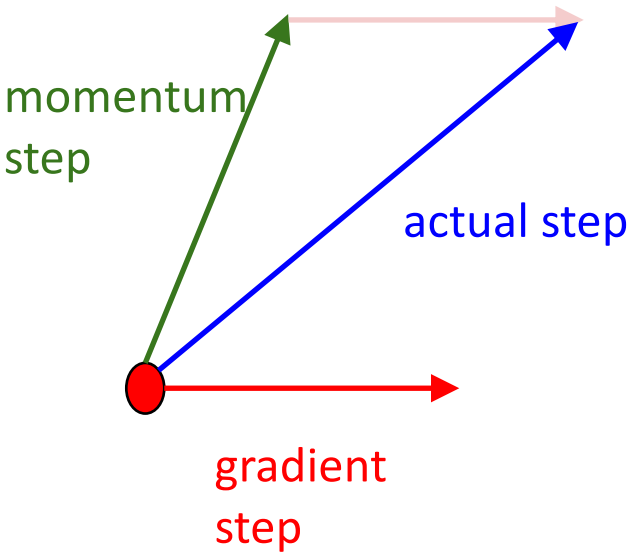


Nesterov momentum update

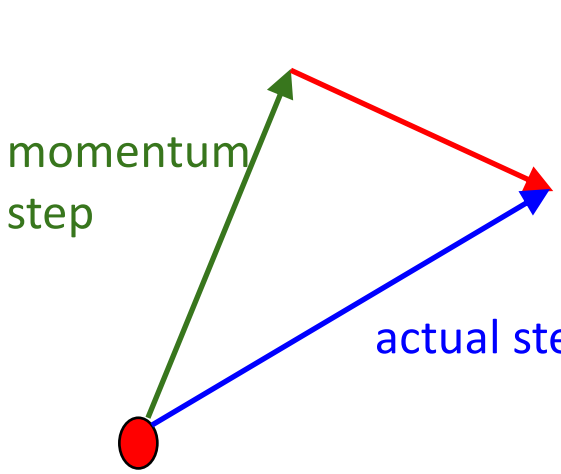


Nesterov Momentum update

Momentum update



Nesterov momentum update



Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

Nesterov Momentum update

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

Nesterov Momentum update

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

Variable transform and rearranging saves the day:

$$\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}$$

Nesterov Momentum update

TECHNISCHE UNIVERSITÄT DARMSTADT

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

Variable transform and rearranging saves the day:

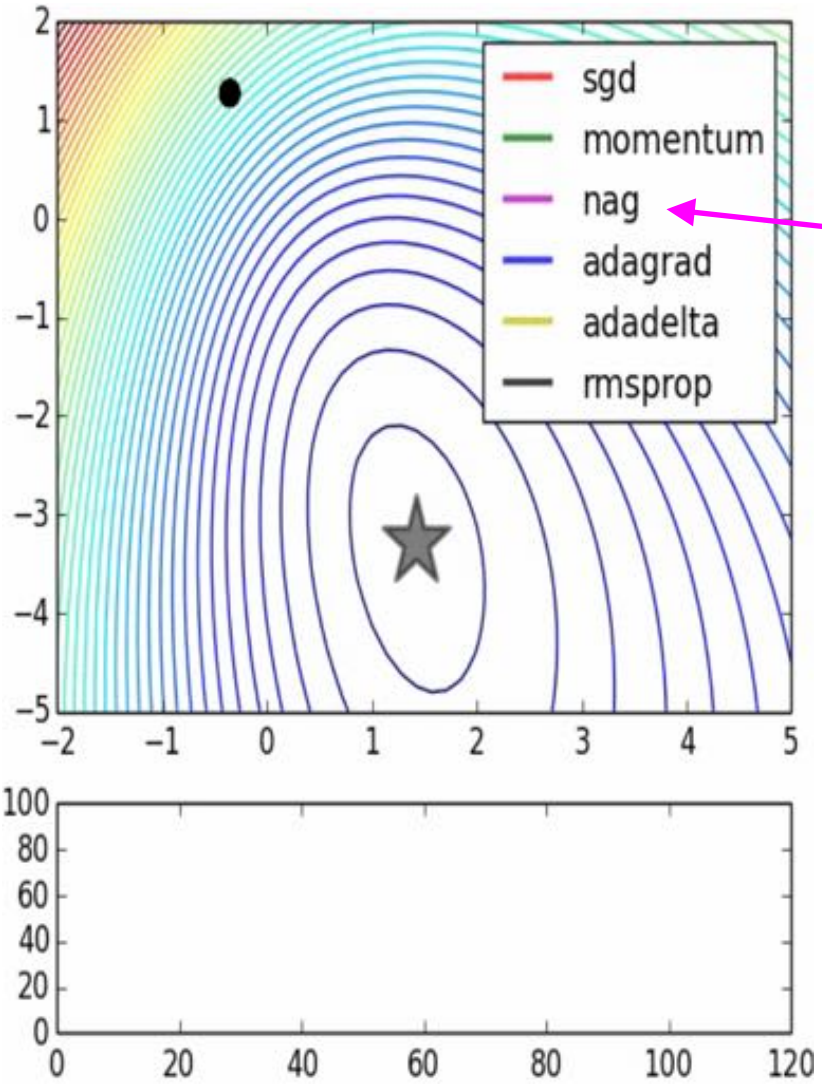
$$\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}$$

Replace all thetas with phis, rearrange and obtain:

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\phi_{t-1})$$

$$\phi_t = \phi_{t-1} - \mu v_{t-1} + (1 + \mu) v_t$$

```
# Nesterov momentum update rewrite
v_prev = v
v = mu * v - learning_rate * dx
x += -mu * v_prev + (1 + mu) * v
```

nag =
Nesterov Accelerated Gradient

AdaGrad update [Duchi et al., 2011]

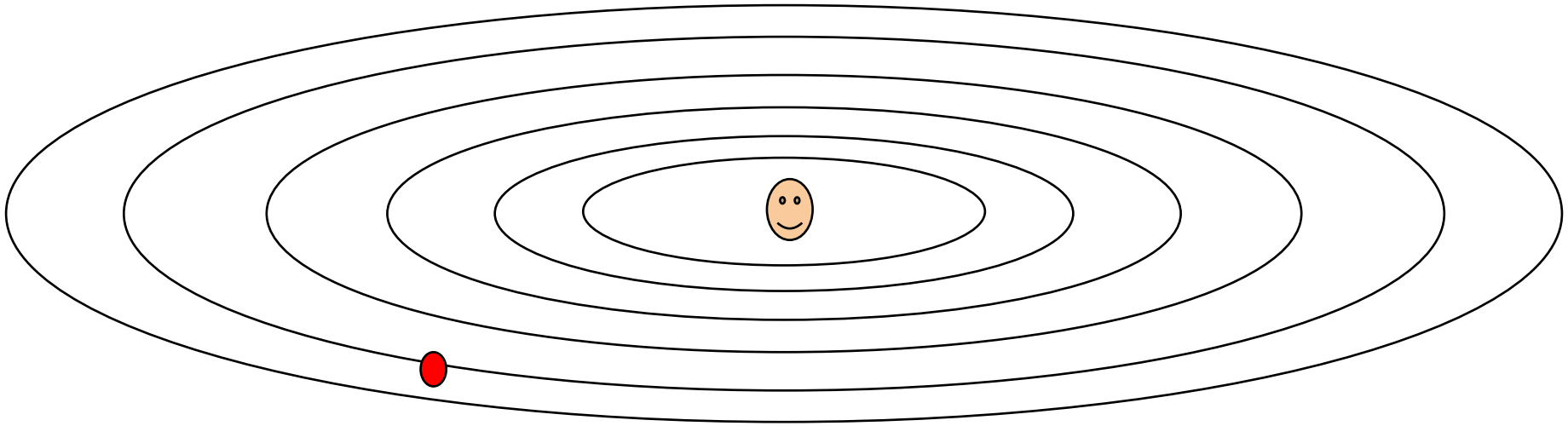
TECHNISCHE UNIVERSITÄT DARMSTADT

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

AdaGrad update [Duchi et al., 2011]

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

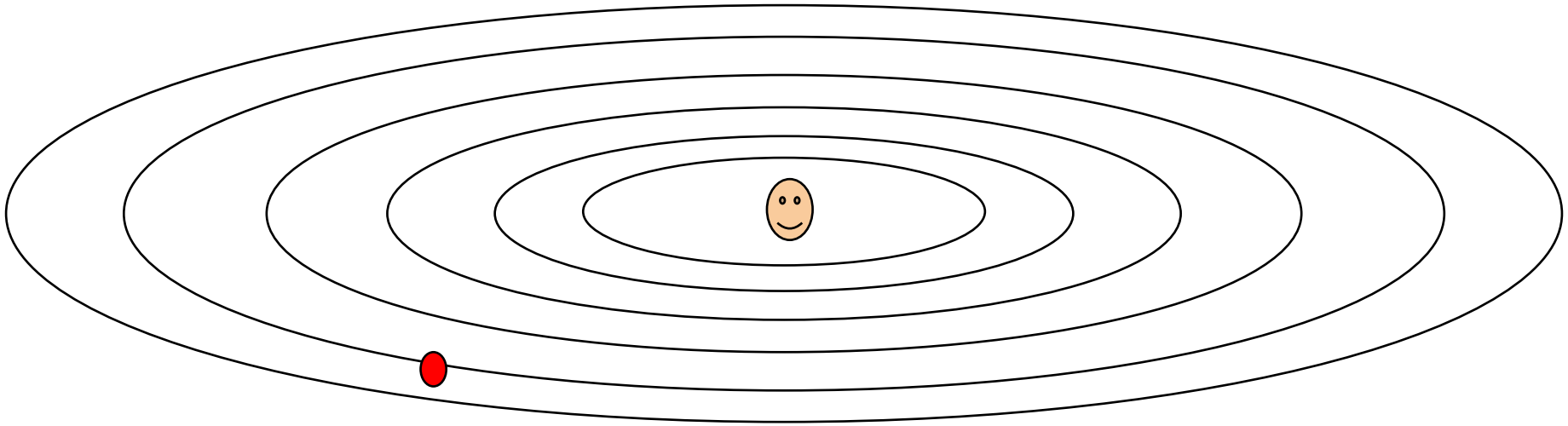


Q: What happens with AdaGrad?

AdaGrad update [Duchi et al., 2011]

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



Q2: What happens to the step size over long time?

RMSProp update [Tieleman and Hinton, 2012]

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```


rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w,t) = 0.9\, MeanSquare(w,\, t-1) + 0.1\left(\partial E / \partial w\,(t)\right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w,\, t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

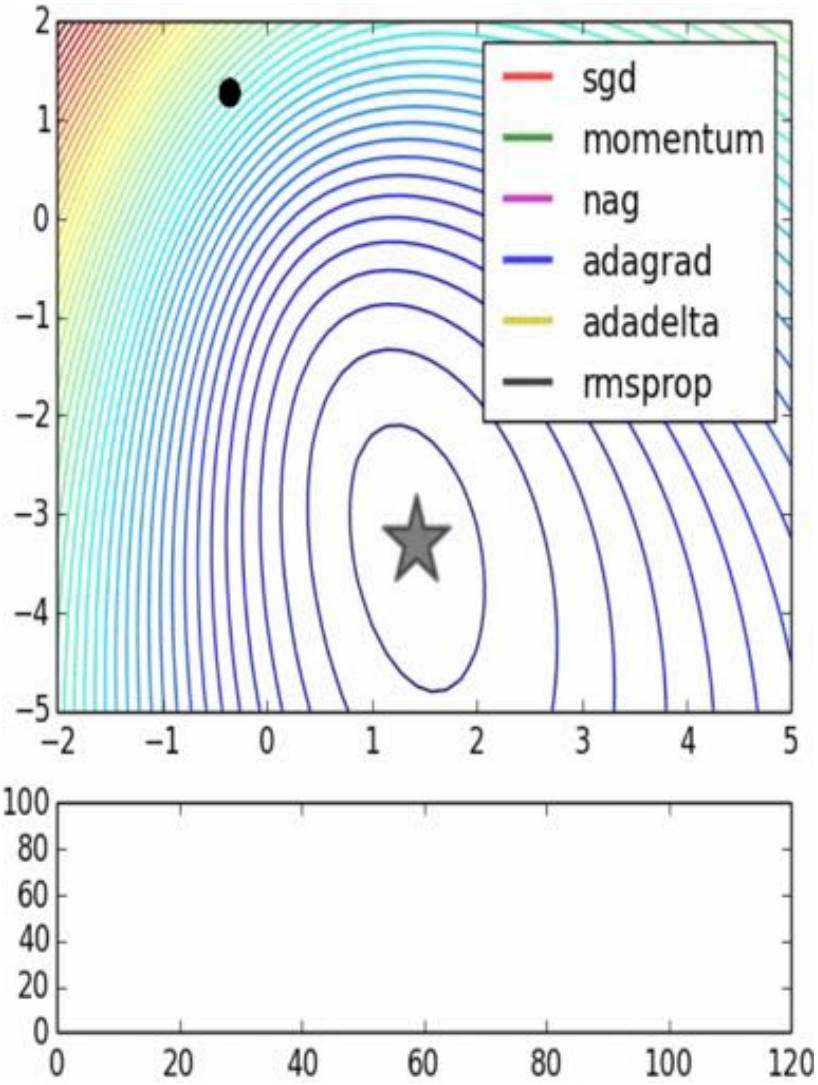
$$MeanSquare(w,t) = 0.9\, MeanSquare(w,\, t-1) + 0.1\left(\partial E/\partial w\,(t)\right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w,\, t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

Cited by several papers as:

[52] T. Tieleman and G. E. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude., 2012.



adagrad
rmsprop

Adam update

[Kingma and Ba, 2014]

TECHNISCHE UNIVERSITÄT DARMSTADT

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```


Adam update

[Kingma and Ba, 2014]

TECHNISCHE UNIVERSITÄT DARMSTADT

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

Adam update

[Kingma and Ba, 2014]

TECHNISCHE UNIVERSITÄT DARMSTADT

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```


Adam update

[Kingma and Ba, 2014]

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# Adam
m,v = #... initialize caches to zeros
for t in xrange(1, big_number):
  dx = # ... evaluate gradient
  m = beta1*m + (1-beta1)*dx # update first moment
  v = beta2*v + (1-beta2)*(dx**2) # update second moment
  mb = m/(1-beta1**t) # correct bias
  vb = v/(1-beta2**t) # correct bias
  x += - learning_rate * mb / (np.sqrt(vb) + 1e-7)
```

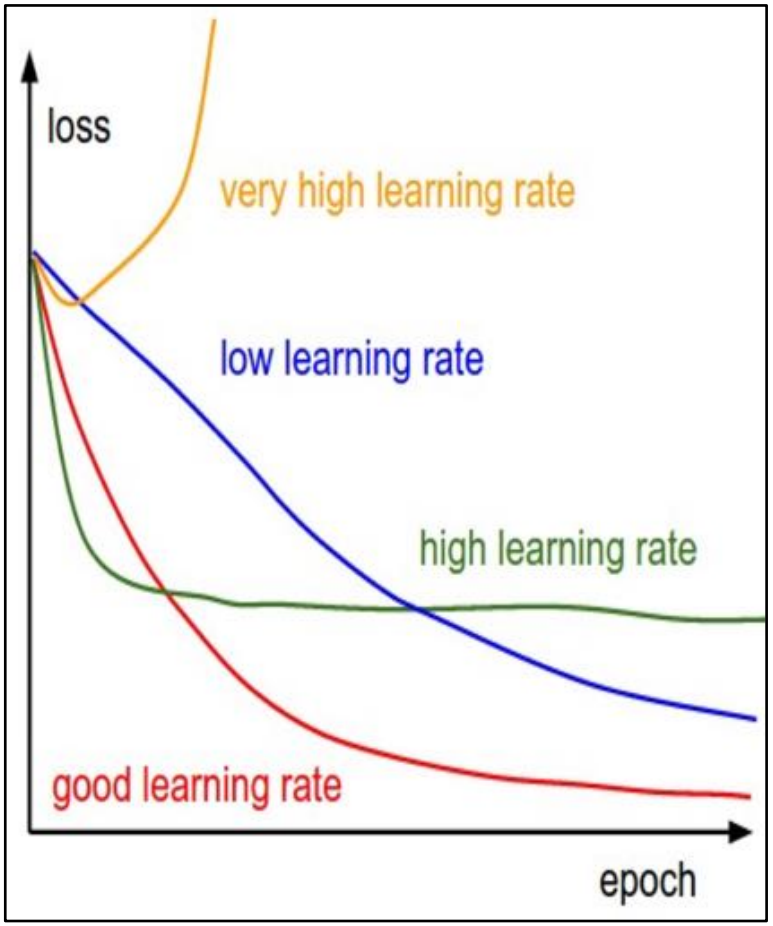
momentum

bias correction
(only relevant in first few iterations when t is small)

RMSProp-like

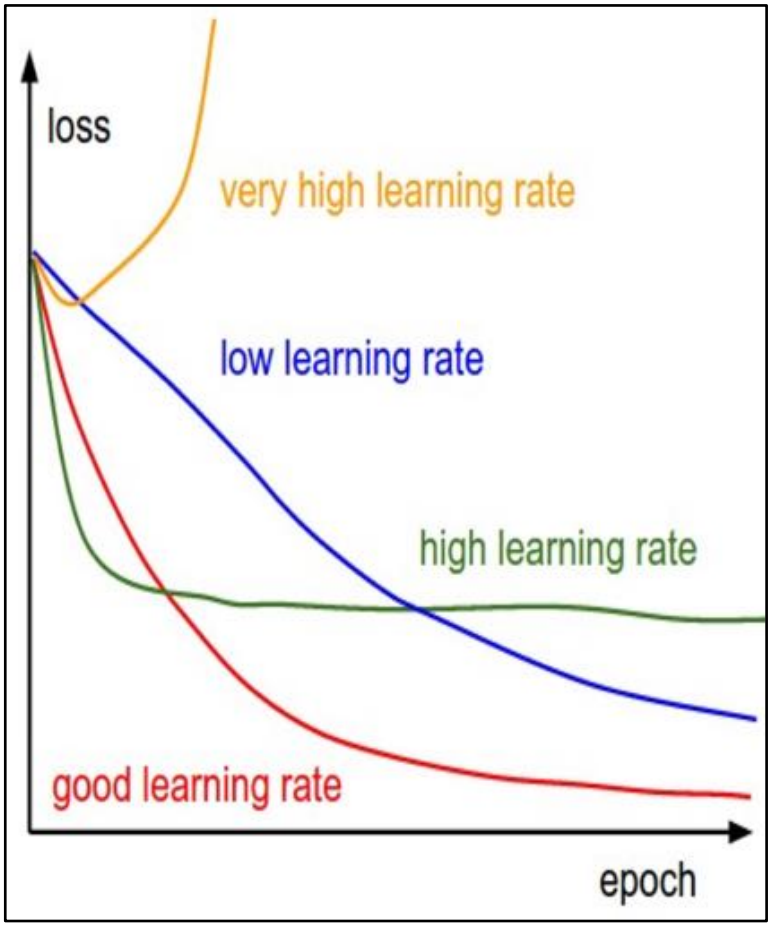
The bias correction compensates for the fact that m,v are initialized at zero and need some time to "warm up".

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

What have you learnt?

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Simple Gradient Methods** like SGD can make adequate progress to an optimum when used on minibatches of data.
- **Second-order** methods make much better progress toward the goal, but are more expensive and unstable.
- **Convergence rates:** quadratic, linear, O(1/n).
- **Momentum:** is another method to produce better effective gradients.
- ADAGRAD, RMSprop diagonally scale the gradient. ADAM scales and applies momentum.