

**MAI4CAREU**

Master programmes in Artificial  
Intelligence 4 Careers in Europe



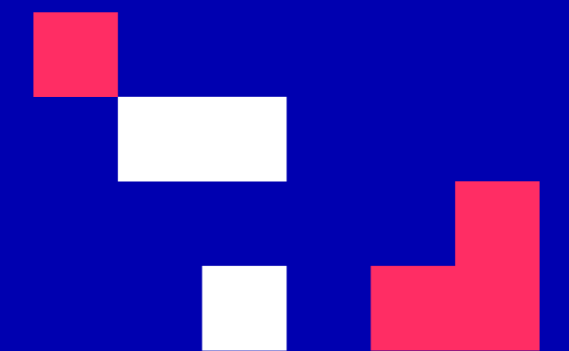
University  
of Cyprus

University of Cyprus

# **MAI645** - Machine Learning for Graphics and Computer Vision

**Andreas Aristidou, PhD**

Spring Semester 2025



## Image Classification: *Regularization, Optimization, Backpropagation*

These notes are based on the work of Fei-Fei Li, Jiajun Wu, Ruohan Gao,  
**CS231 - Deep Learning for Computer Vision**



Microsoft, its subsidiary GitHub, and OpenAI for their GitHub Copilot system received a class-action suit for mass copyright infringement. The plaintiffs argue that by producing code that doesn't give attribution to the original authors whose code is used to generate Copilot's results, the system violates open-source licenses, as well as the Digital Millennium Copyright Act. Similar suits are filed against Dalle-2, Midjourney, and ChatGPT products to protect human content creators.

## AI Tech Enables Industrial-Scale Intellectual-Property Theft, Say Critics

Are ChatGPT, Stability AI and GitHub Copilot the next big breakthroughs, huge legal and regulatory liabilities, or something else entirely?

## Recall from last time

### Image Classification: A core task in Computer Vision



This image by [Nikita](#) is  
licensed under [CC-BY 2.0](#)

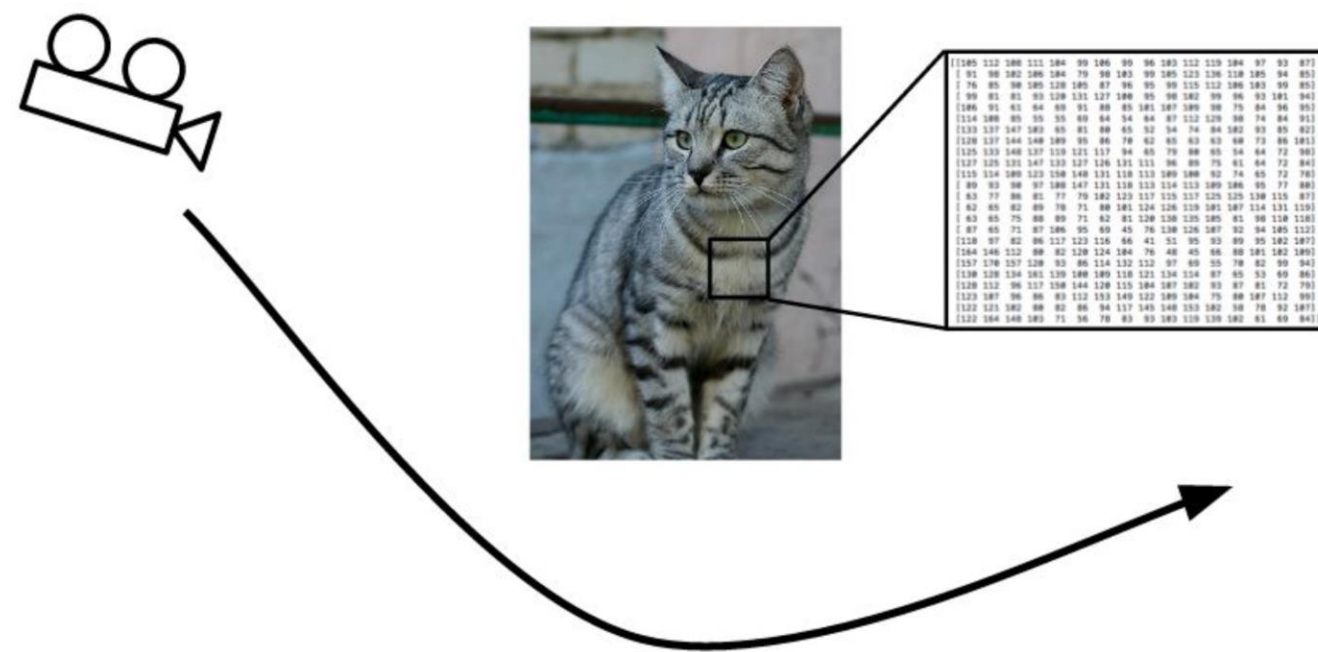
(assume given a set of possible labels)  
{dog, cat, truck, plane, ...}



cat

## Recall from last time: *Challenges of recognition*

Viewpoint

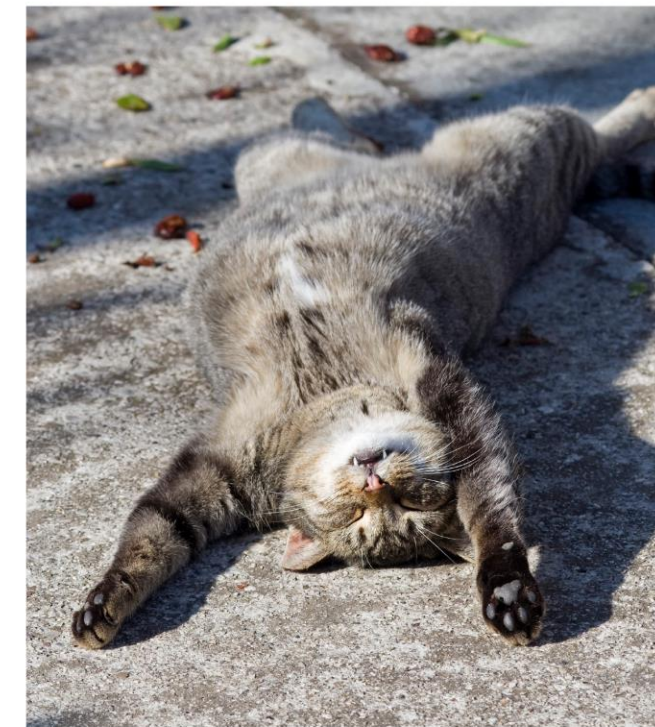


Illumination



This image is [CC0 1.0](#) public domain

Deformation



This image by [Umberto Salvagnin](#) is licensed under [CC-BY 2.0](#)

Occlusion



This image by [jonsson](#) is licensed under [CC-BY 2.0](#)

Clutter



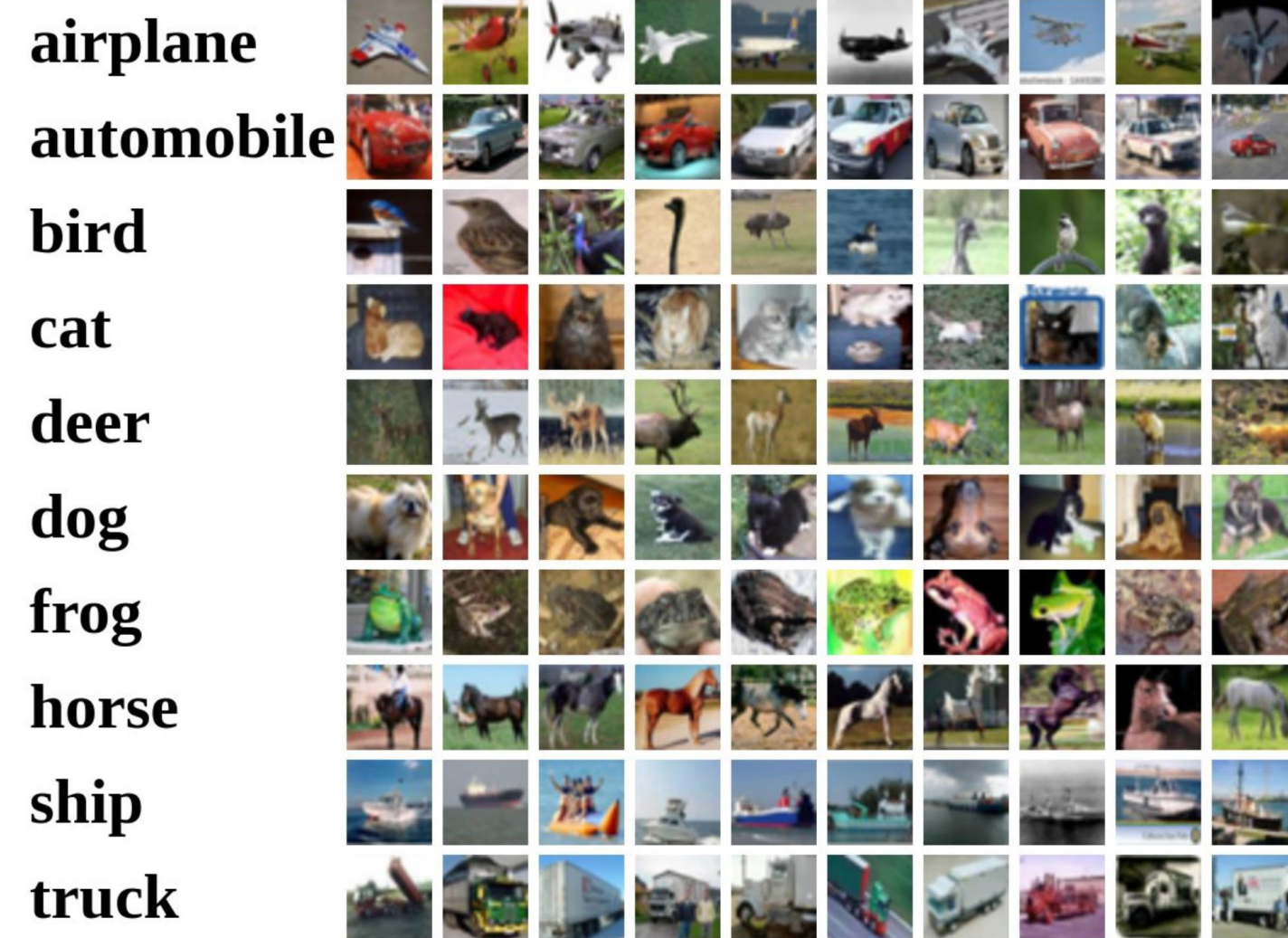
This image is [CC0 1.0](#) public domain

Intraclass Variation

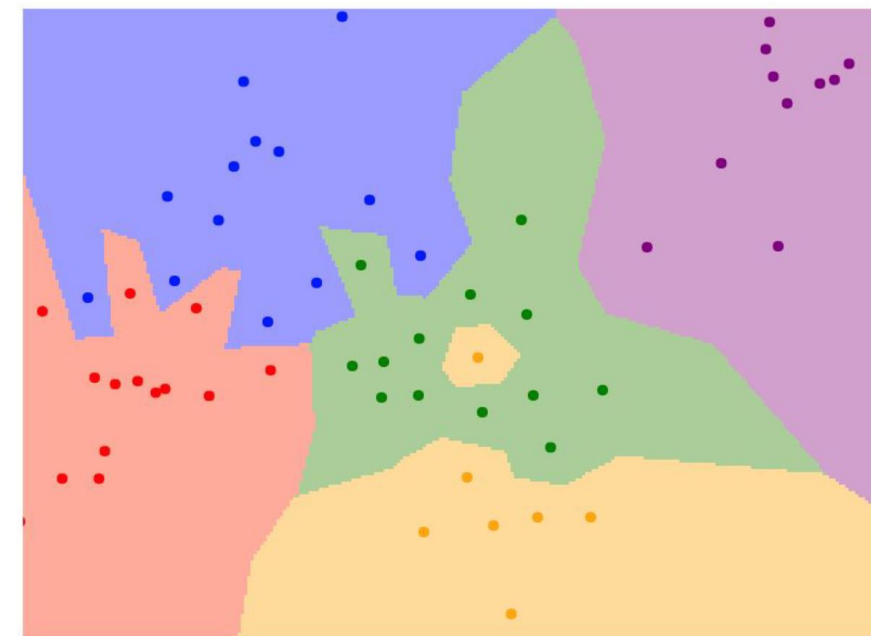


This image is [CC0 1.0](#) public domain

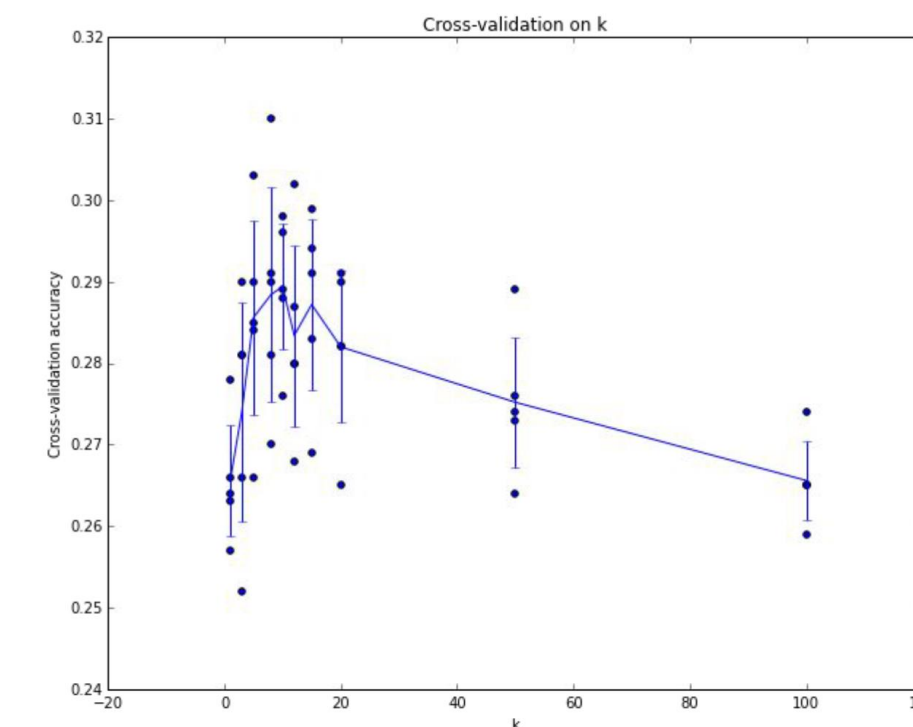
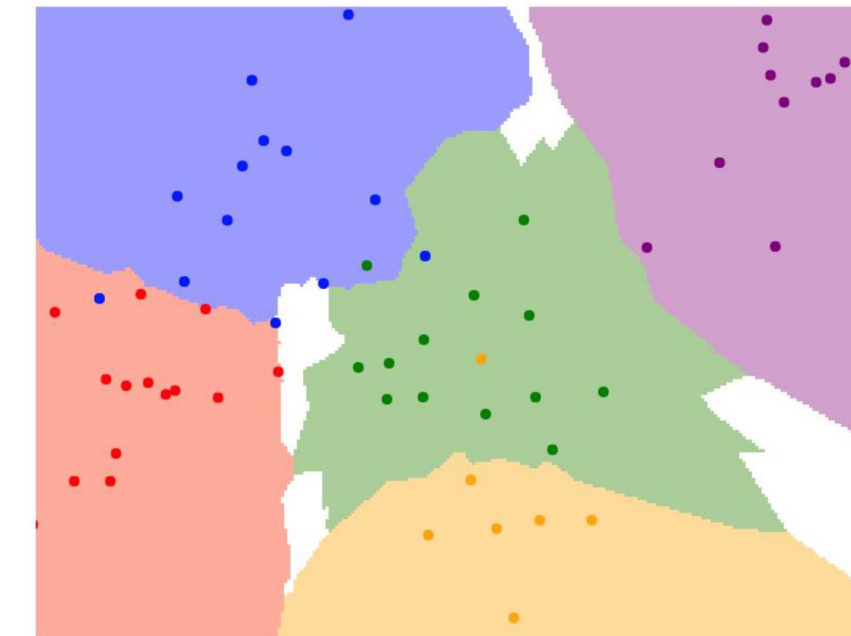
## Recall from last time: *Data-driven approaches, kNN*



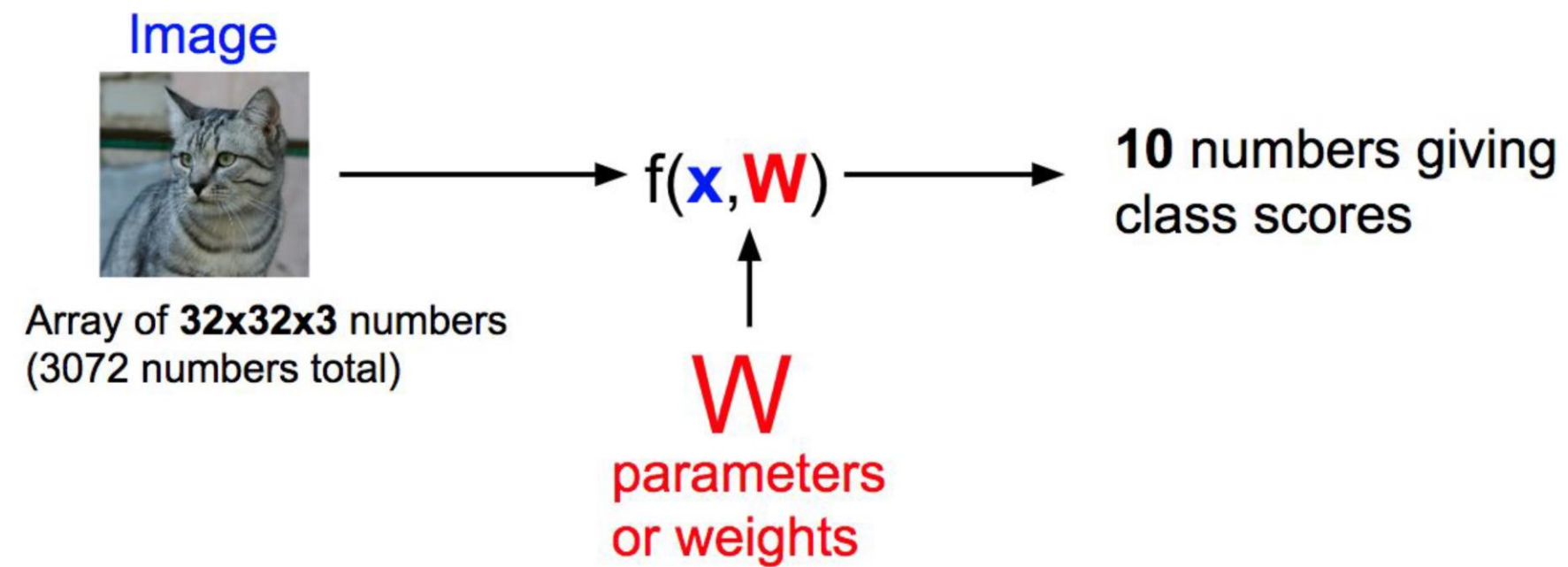
1-NN classifier



5-NN classifier



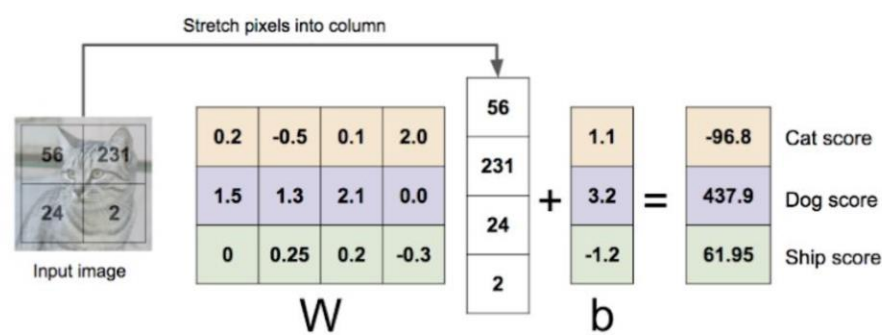
## Recall from last time: *Data-driven approaches, kNN*



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

### Algebraic Viewpoint

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



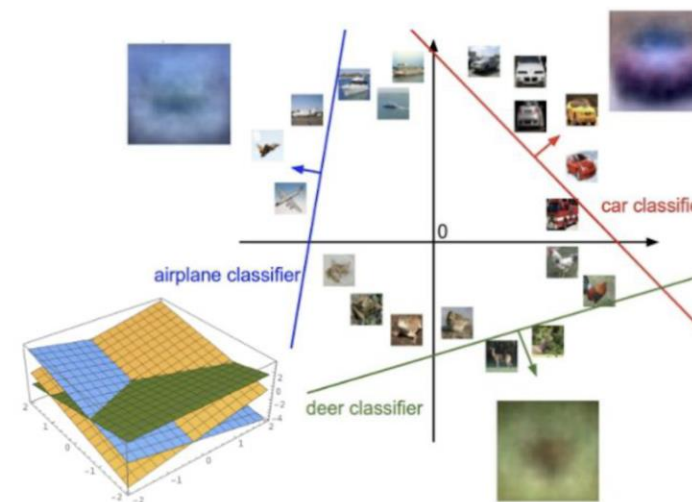
### Visual Viewpoint

One template per class



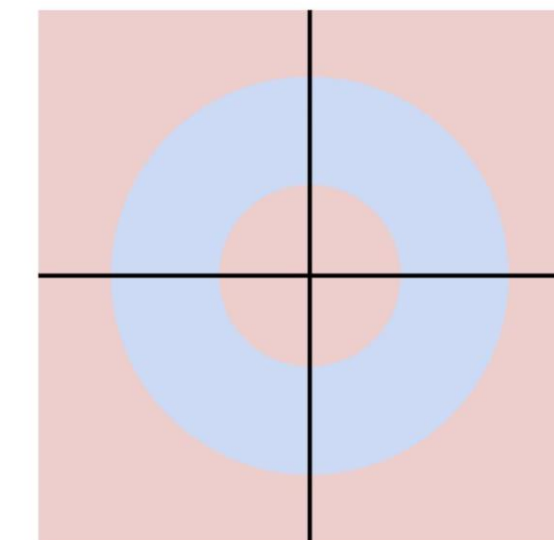
### Geometric Viewpoint

Hyperplanes cutting up space



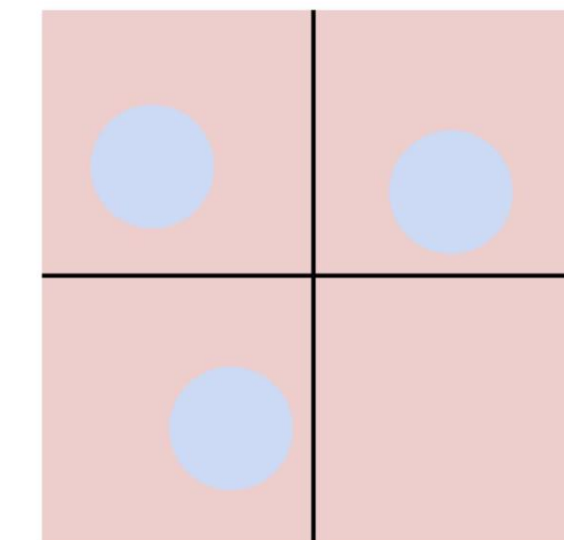
**Class 1:**  
 $1 \leq L2 \text{ norm} \leq 2$

**Class 2:**  
Everything else



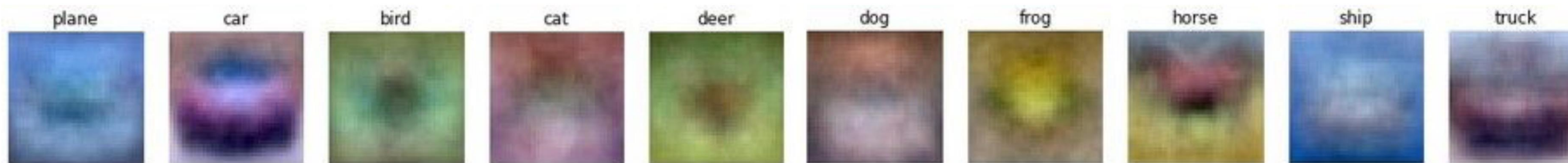
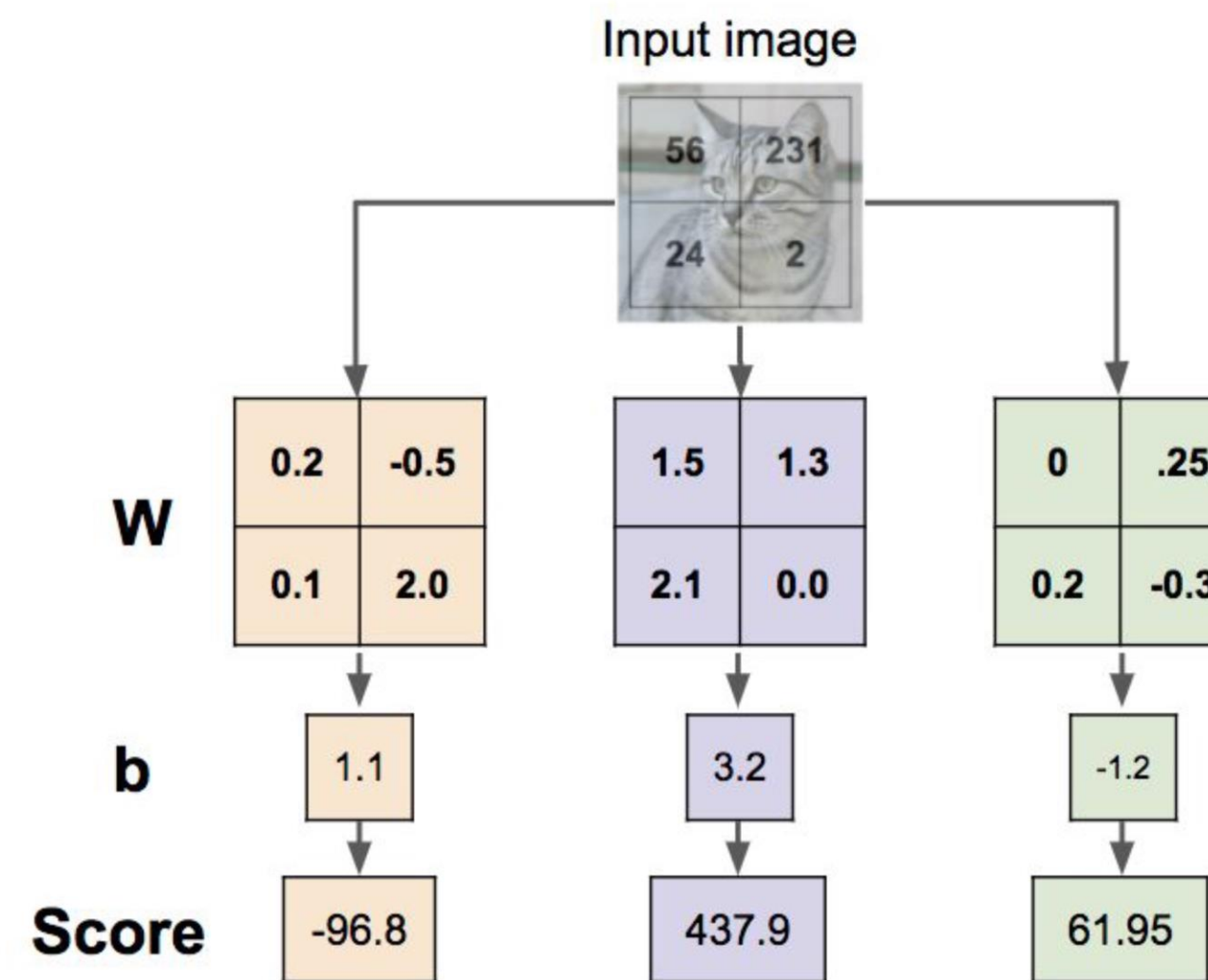
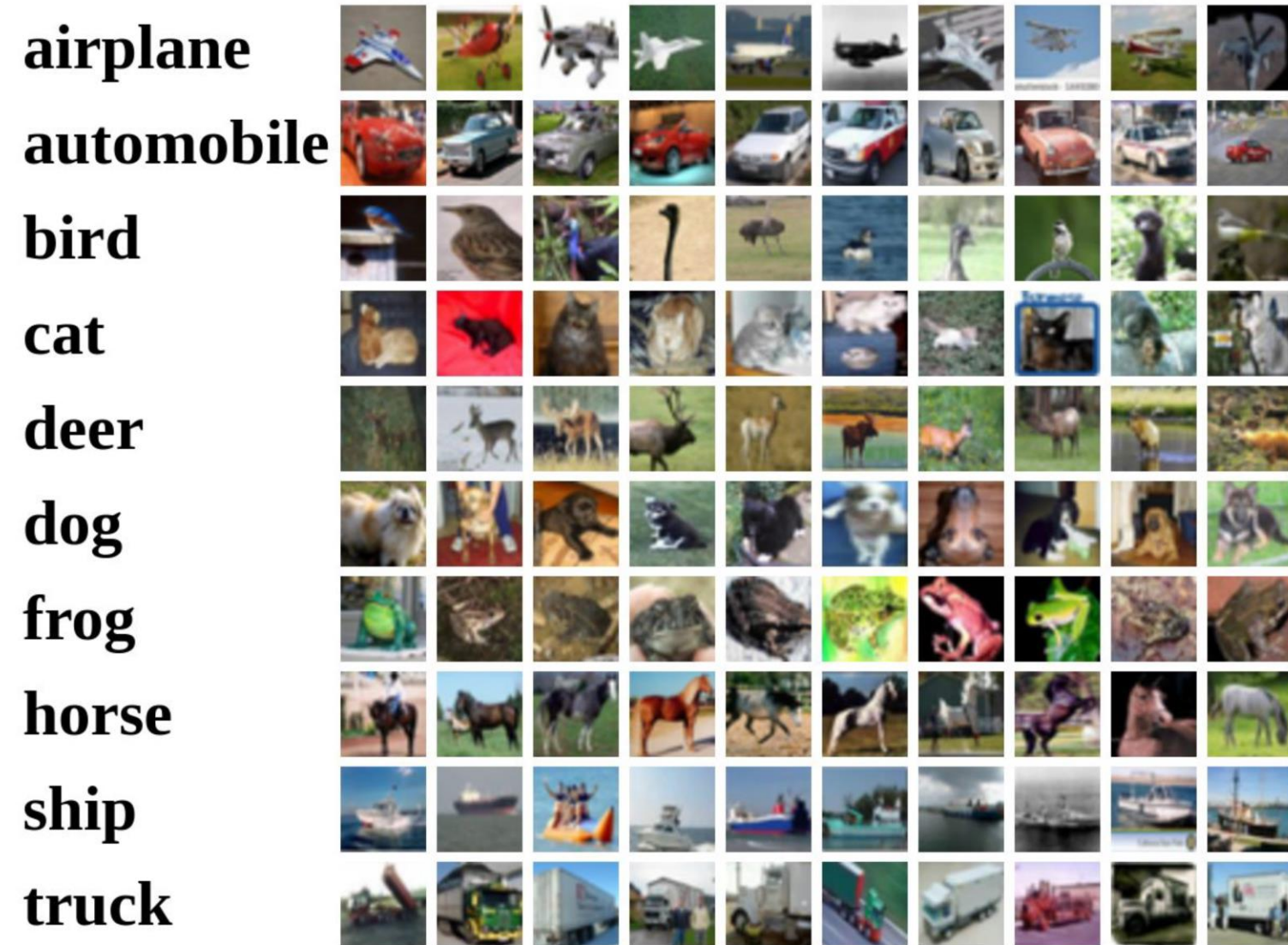
**Class 1:**  
Three modes

**Class 2:**  
Everything else



## Recall from last time: *Data-driven approaches, kNN*

# Interpreting a Linear Classifier: Visual Viewpoint



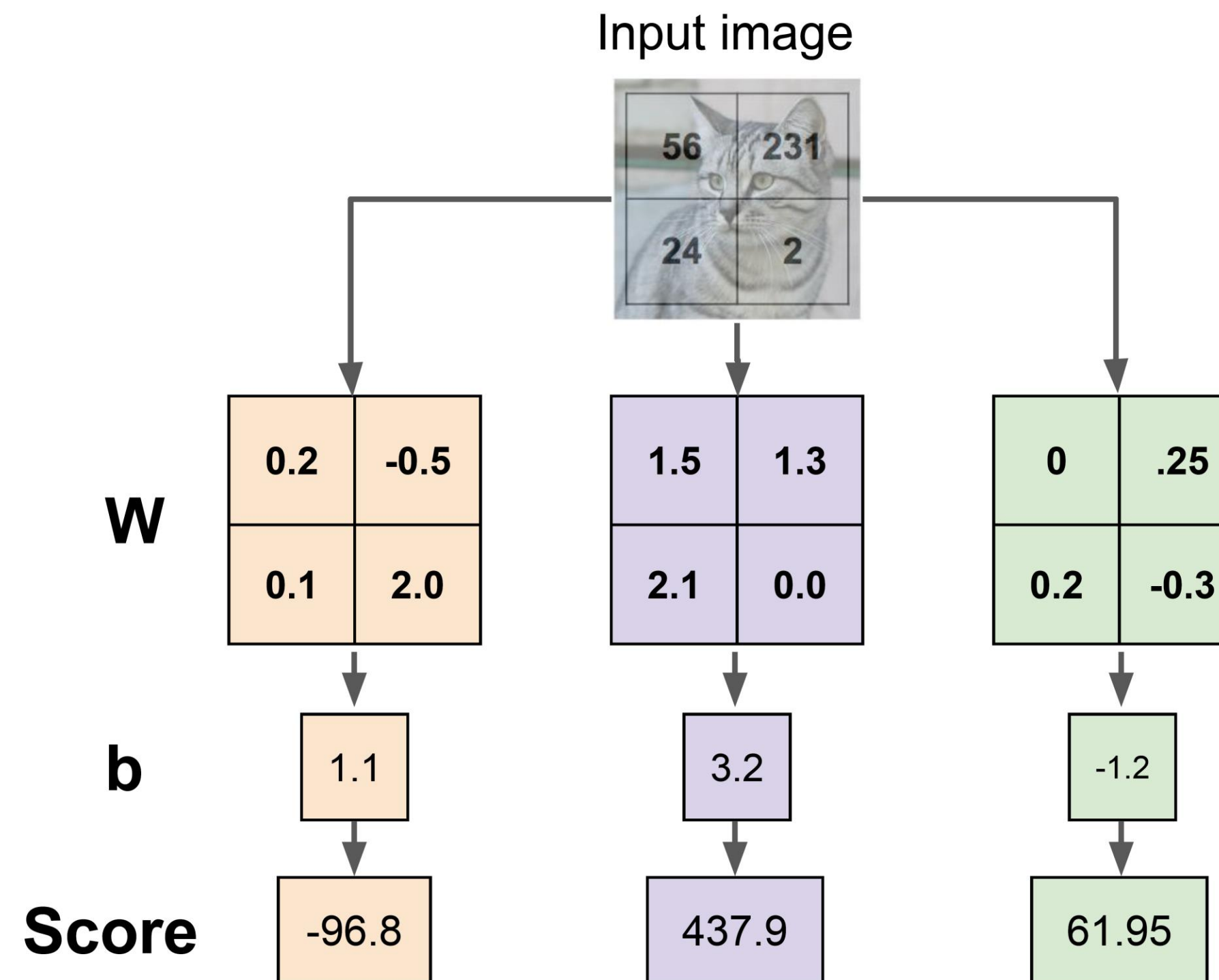
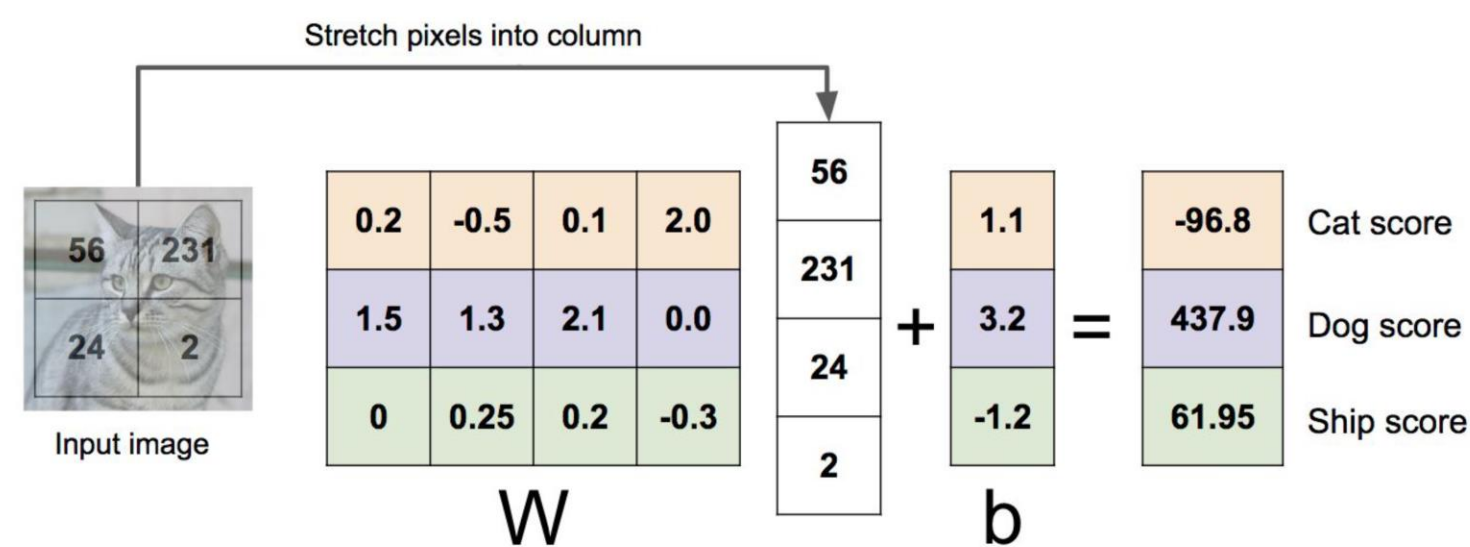


## Recall from last time: *Data-driven approaches, kNN*

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

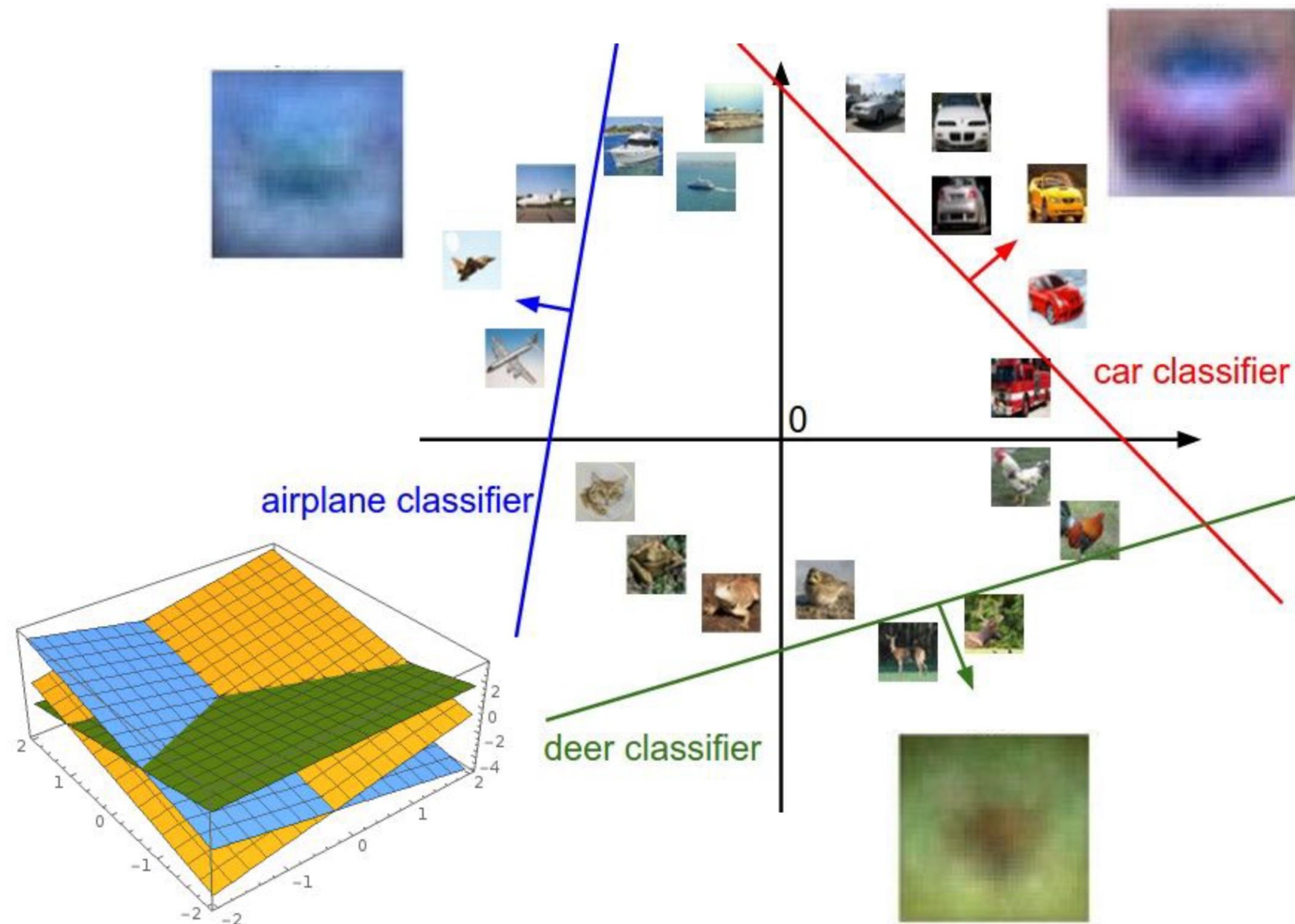
### Algebraic Viewpoint

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



Recall from last time: *Data-driven approaches, kNN*

# Interpreting a Linear Classifier: Geometric Viewpoint



Plot created using [Wolfram Cloud](#)

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



Array of **32x32x3** numbers  
(3072 numbers total)

Cat image by Nikita is licensed under [CC-BY 2.0](#)



## Recall from last time: *Data-driven approaches, kNN*

Suppose: 3 training examples, 3 classes.  
 With some  $W$  the scores  $f(x, W) = Wx$  are:



cat	<b>3.2</b>	1.3	2.2
car	5.1	<b>4.9</b>	2.5
frog	-1.7	2.0	<b>-3.1</b>

A **loss function** tells how good our current classifier is

Given a dataset of examples

$$\{(x_i, y_i)\}_{i=1}^N$$

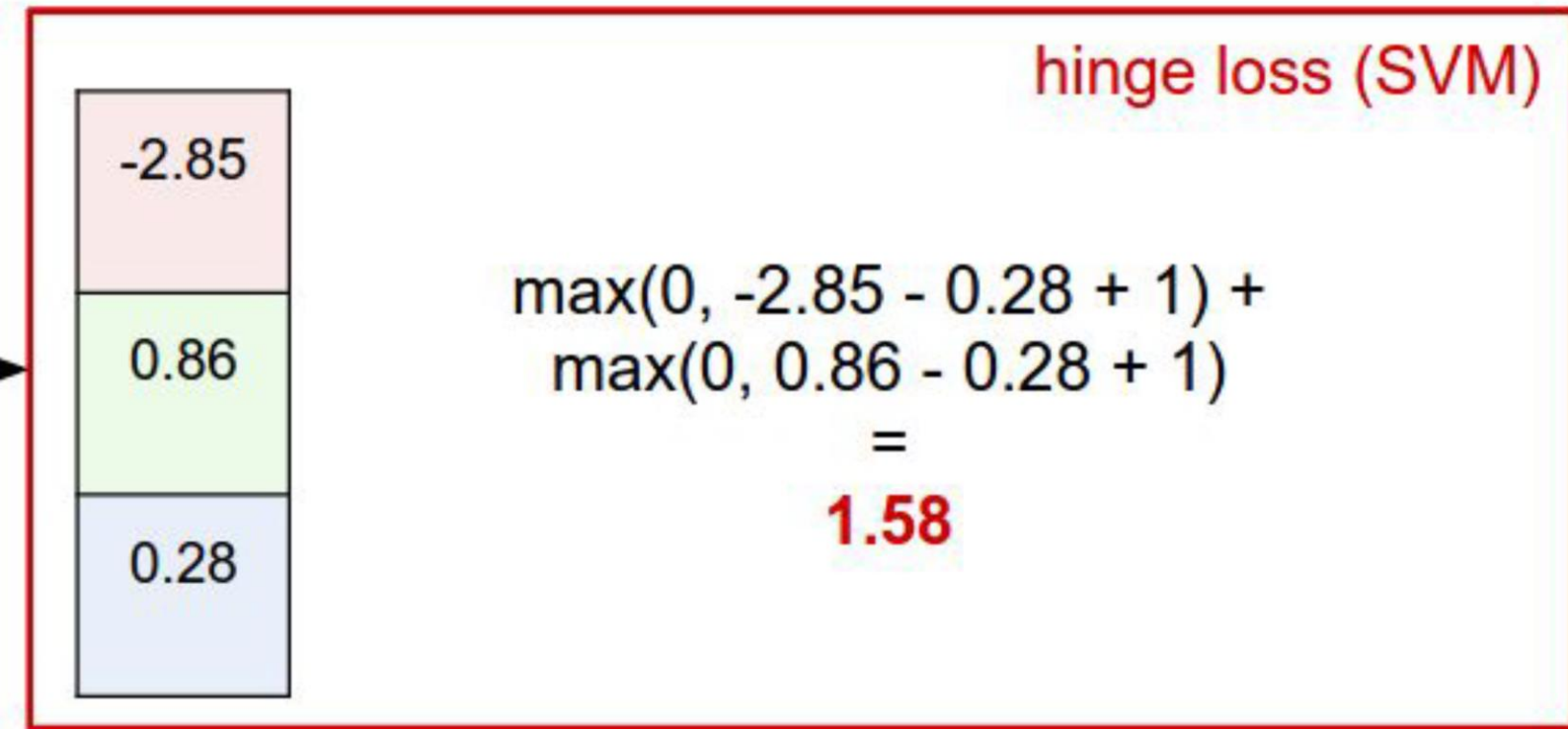
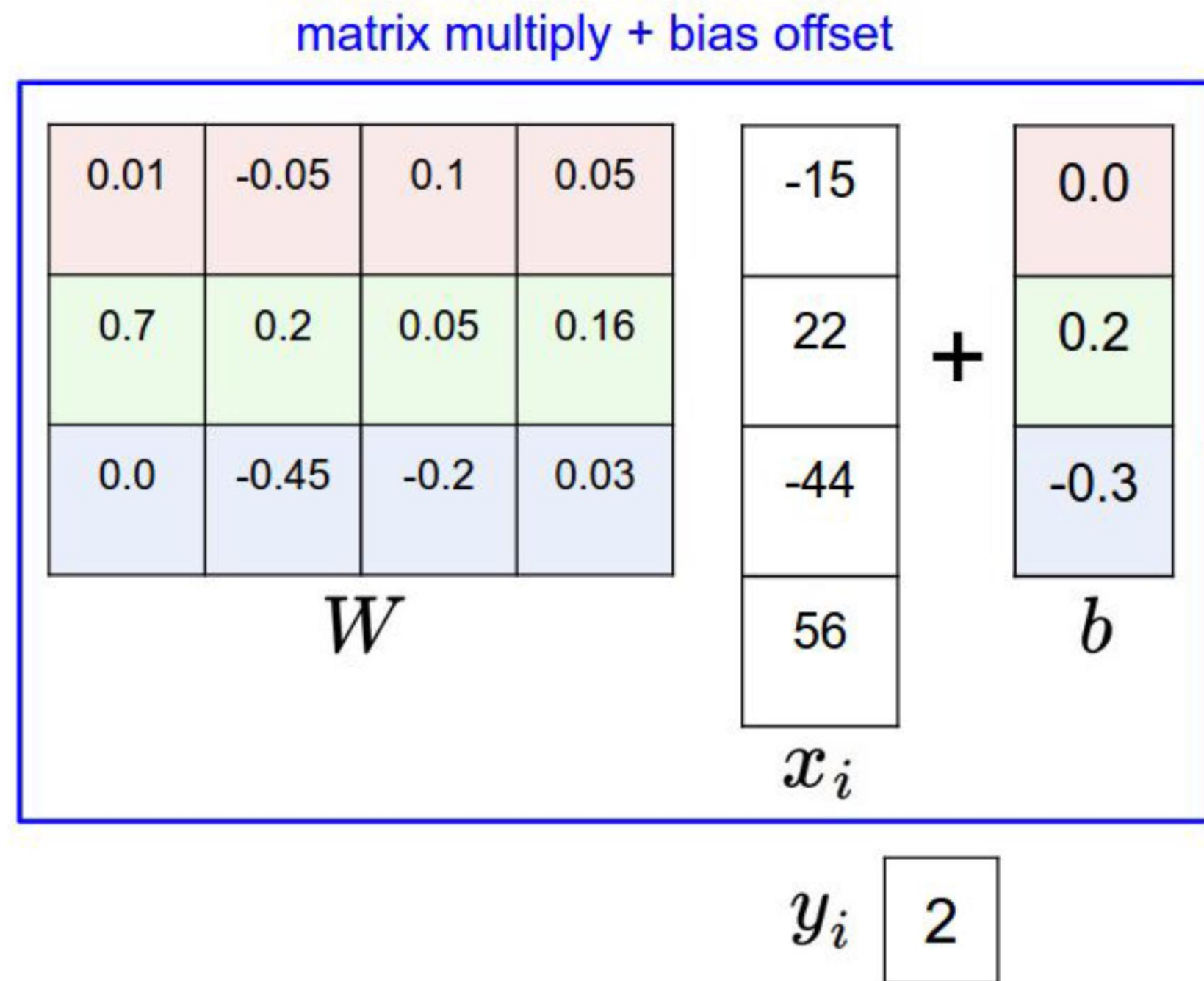
Where  $x_i$  is image and  $y_i$  is (integer) label

Loss over the dataset is a average of loss over examples:

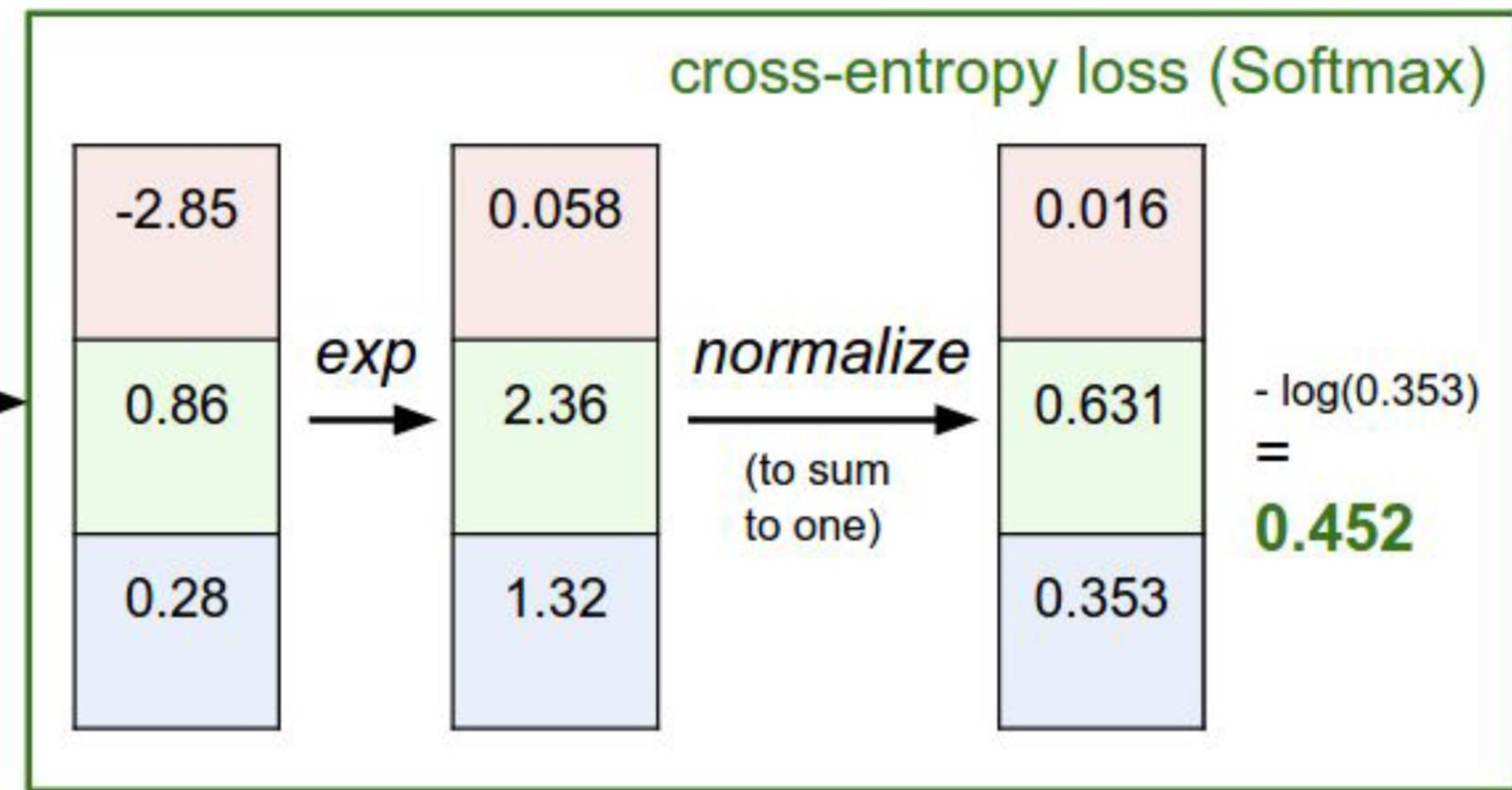
$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

**Recall from last time: *Softmax vs SVM***

**Softmax vs. SVM**



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



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



## Recall from last time: *Softmax vs SVM*

**Support Vector Machine** (SVM) is a linear classifier used for binary classification problems. It finds the best hyperplane that separates the data into two classes and classifies new data points based on which side of the hyperplane they fall on. SVM can also be extended to handle multiclass classification problems through one-vs-one or one-vs-all approaches.

The **softmax function** is a popular activation function used in machine learning, especially in multiclass classification problems. It maps a set of real-valued numbers to a probability distribution over multiple classes, such that all the probabilities sum to 1. In this way, the output of the softmax function can be interpreted as the estimated class probabilities.

In summary, Softmax is used for multiclass classification problems and SVM is used for binary classification problems.

**Recall from last time: *Softmax vs SVM***

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

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

Question:

Suppose that we found a  $W$  such that  $L = 0$ . Is this  $W$  unique?

**No!  $2W$  also has  $L = 0$ !**

## Recall from last time: *Softmax vs SVM*

Suppose: 3 training examples, 3 classes.  
With some  $W$  the scores  $f(x, W) = Wx$  are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9	0	

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

**Before:**

$$\begin{aligned}
 &= \max(0, 1.3 - 4.9 + 1) \\
 &\quad + \max(0, 2.0 - 4.9 + 1) \\
 &= \max(0, -2.6) + \max(0, -1.9) \\
 &= 0 + 0 \\
 &= 0
 \end{aligned}$$

**With  $W$  twice as large:**

$$\begin{aligned}
 &= \max(0, 2.6 - 9.8 + 1) \\
 &\quad + \max(0, 4.0 - 9.8 + 1) \\
 &= \max(0, -6.2) + \max(0, -4.8) \\
 &= 0 + 0 \\
 &= 0
 \end{aligned}$$

**How do we choose between  $W$  and  $2W$ ?**

## Regularization

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too* well on training data



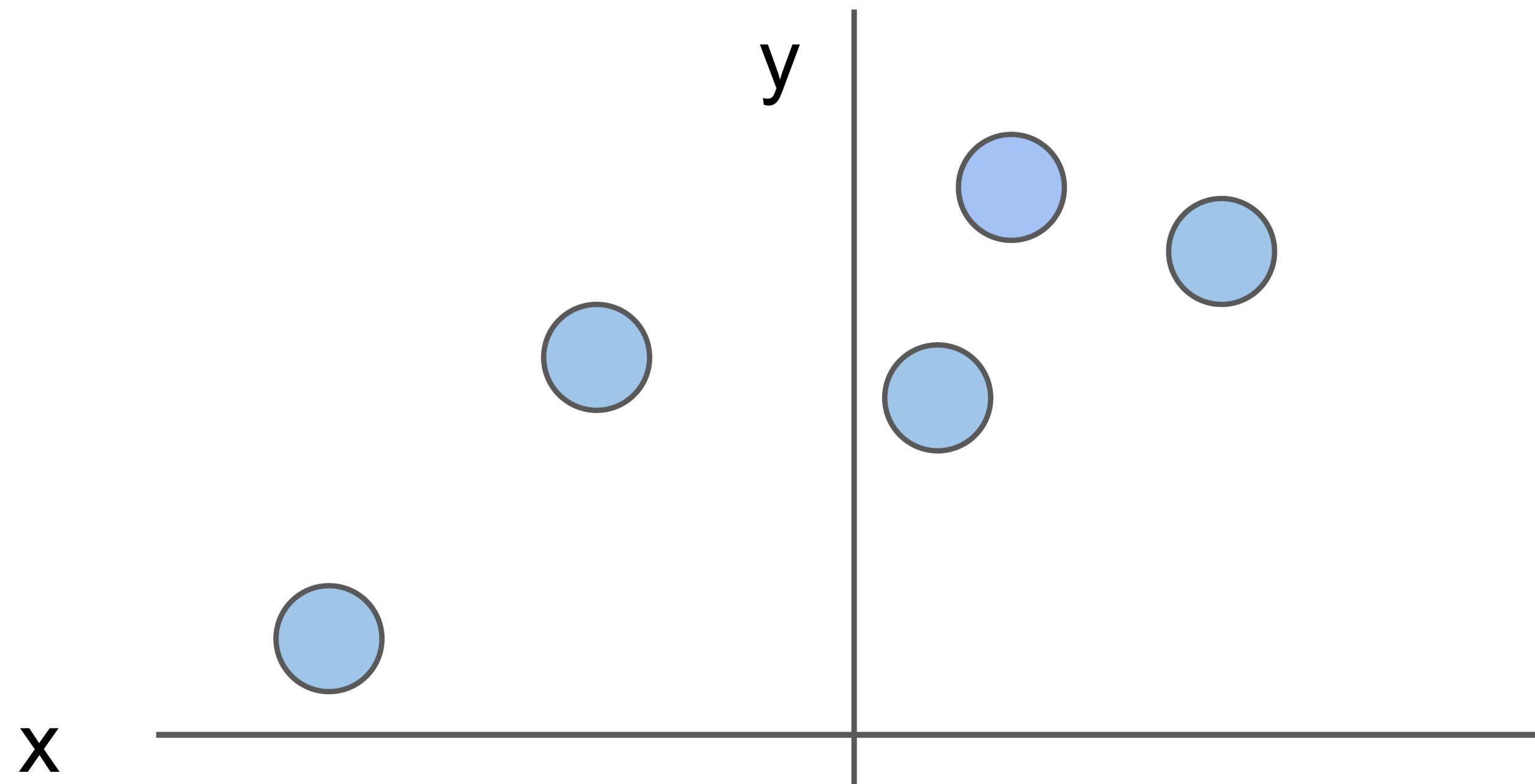
## Regularization

**Regularization** is used in machine learning to prevent overfitting, which is a common problem in complex models with many parameters. Overfitting occurs when a model fits the training data too well and memorizes the noise and random fluctuations in the data, leading to poor performance on unseen data.

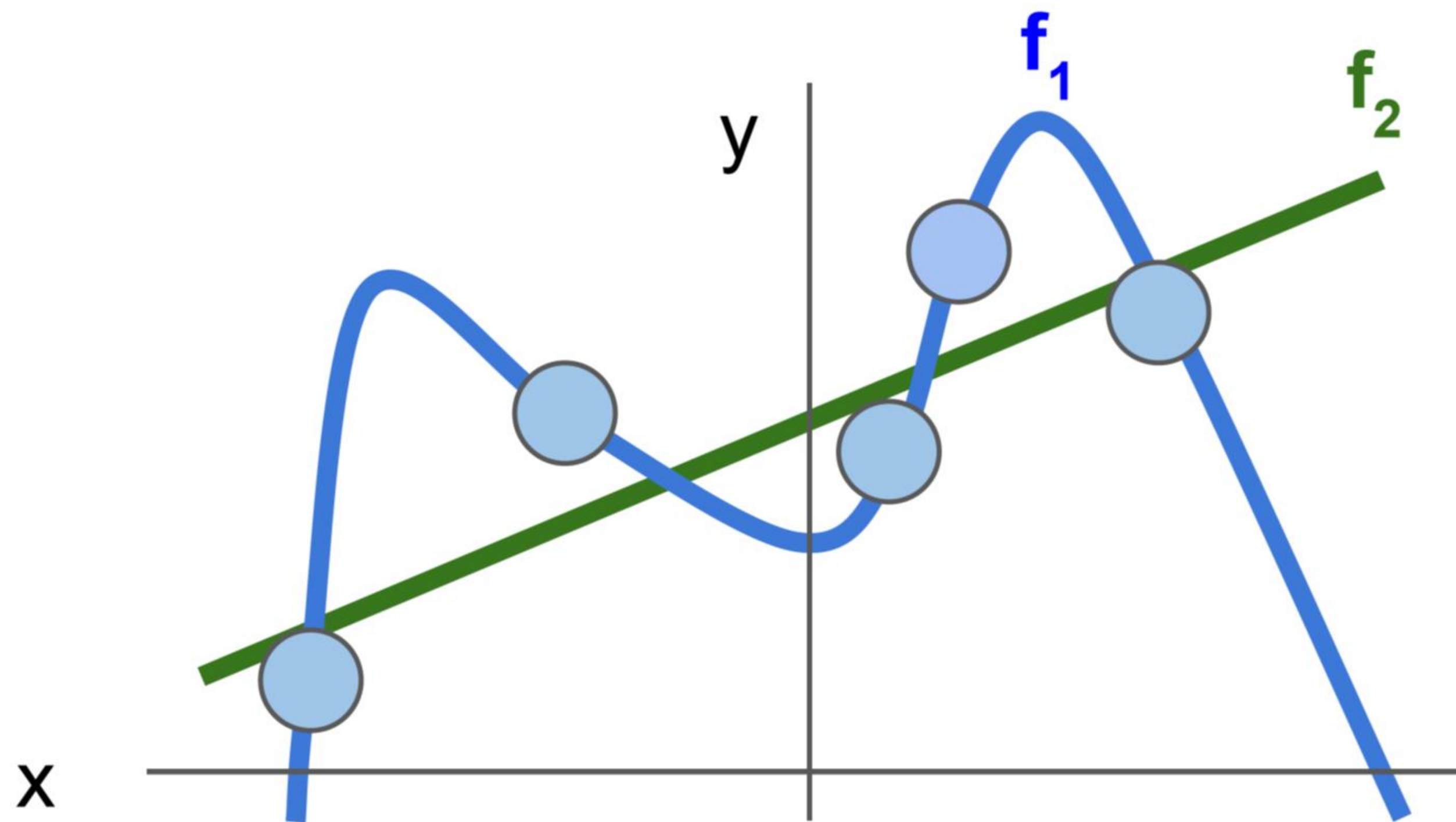
Regularization helps to address overfitting by adding a penalty term to the cost function that discourages overly complex models with large coefficients. It can be applied by adding a penalty term, such as L1 or L2 regularization, to the cost function. This term discourages overly complex models by penalizing large coefficients, leading to sparse models that are less likely to overfit the data. The goal of regularization is to find a balance between fitting the training data well and maintaining a simple model that can generalize well to unseen data.

In summary, regularization is used in machine learning to **prevent overfitting, improve the generalization performance of models, and make the models more robust and reliable.**

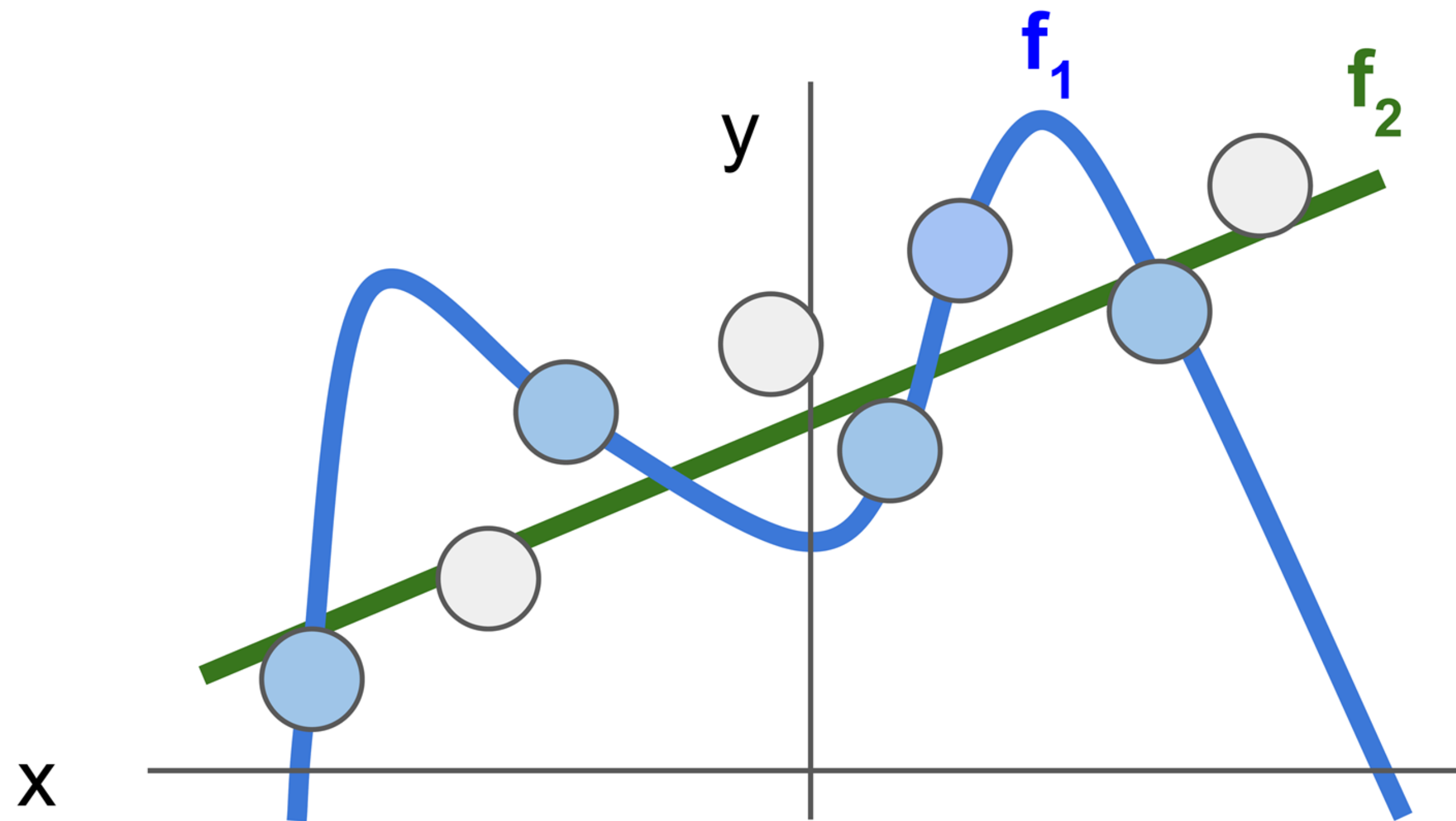
## Regularization intuition: *toy example training data*



## Regularization intuition: *Prefer Simpler Models*



## Regularization intuition: *Prefer Simpler Models*



Regularization pushes against fitting the data *too* well so we don't fit noise in the data

## Regularization

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too* well on training data

**Occam's Razar:** Among multiple competing hypotheses, the simplest is the best,  
William of Ockham 1285-1347

## Regularization

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

$\lambda$  = regularization strength (hyperparameter)

**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too* well on training data

### Simple examples

L2 regularization:  $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization:  $R(W) = \sum_k \sum_l |W_{k,l}|$

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

Why regularize?

- Express preferences over weights
- Make the model *simple* so it works on test data
- Improve optimization by adding curvature

## Regularization: *Recap*

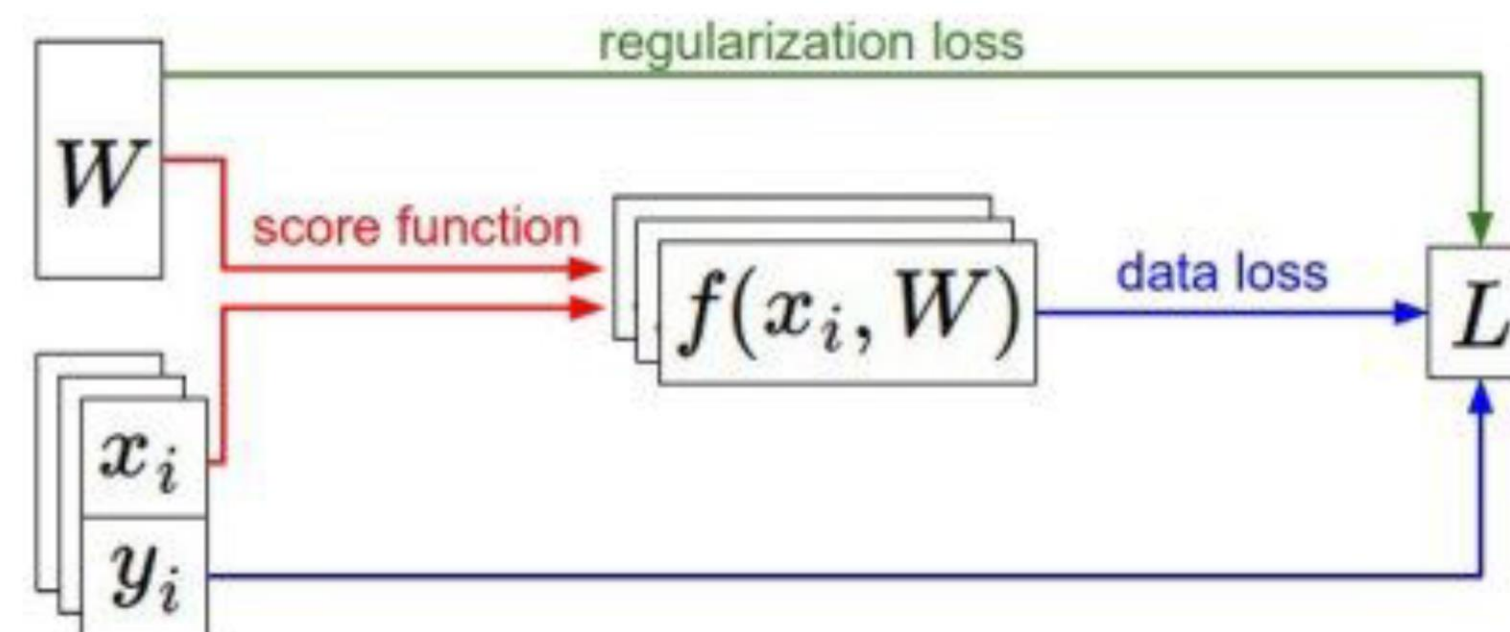
How do we find the best  $W$ ?

- We have some dataset of  $(x,y)$
- We have a **score function**:  $s = f(x; W) \stackrel{\text{e.g.}}{=} Wx$
- We have a **loss function**:

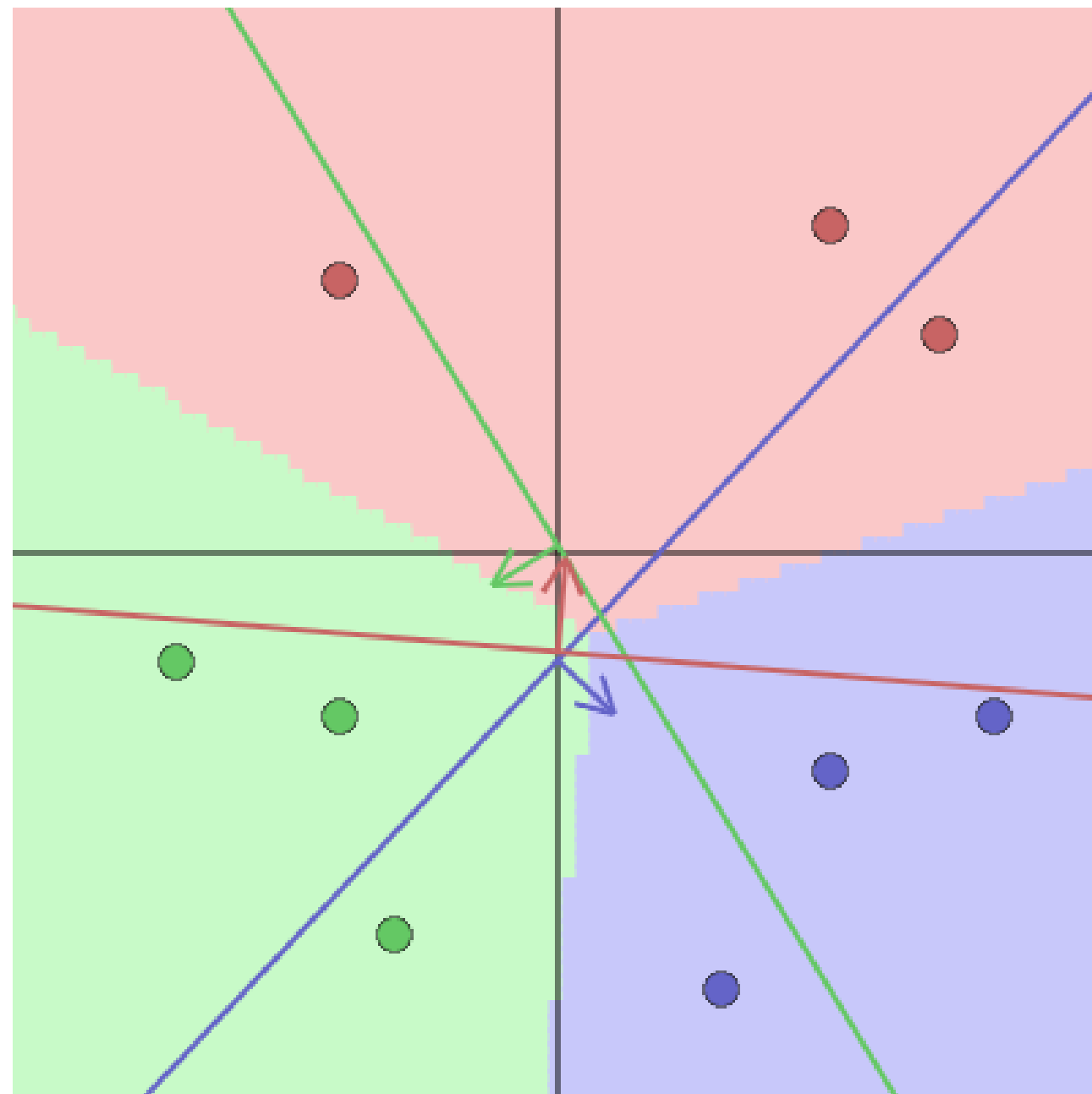
$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \quad \text{Softmax}$$

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

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



## Regularization: *Interactive Web Demo*



W[0,0]	W[0,1]	b[0]
▲	▲	▲
1.03	0.99	-0.19
0.04	-0.02	0.00
▼	▼	▼
W[1,0]	W[1,1]	b[1]
▲	▲	▲
-1.20	0.74	0.02
-0.02	0.06	0.22
▼	▼	▼
W[2,0]	W[2,1]	b[2]
▲	▲	▲
0.16	-1.72	0.30
-0.02	-0.04	-0.22
▼	▼	▼

x[0]	x[1]	y	s[0]	s[1]	s[2]	L
0.50	0.40	0	0.72	-0.28	-0.31	0.00
0.80	0.30	0	0.93	-0.72	-0.09	0.00
0.30	0.80	0	0.91	0.25	-1.03	0.35
-0.40	0.30	1	-0.31	0.72	-0.28	0.00
-0.30	0.70	1	0.19	0.90	-0.95	0.29
-0.70	0.20	1	-0.72	1.01	-0.16	0.00
0.70	-0.40	2	0.13	-1.11	1.11	0.03
0.50	-0.60	2	-0.27	-1.02	1.42	0.00
-0.40	-0.50	2	-1.10	0.13	1.10	0.03
						mean:
						0.08

Total data loss: 0.08  
 Regularization loss: 1.11  
 Total loss: 1.19

Step size: 0.10000

L2 Regularization strength: 0.15849

<http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/>





## Optimization

**Optimization** is a crucial step in machine learning as it determines the parameters of the model that lead to the best performance on a given task. It plays a crucial role in determining the performance of machine learning models and is an active area of research.

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

```
# 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
# ... (truncated: continues for 1000 lines)
```

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

```
# 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!**  
**(SOTA is ~99.7%)**

## Strategy #2: *Follow the slope*

In 1-dimension, the derivative of a function:

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

In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

The slope in any direction is the **dot product** of the direction with the gradient  
The direction of steepest descent is the **negative gradient**

## Strategy #2: Follow the slope

**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,  
?,  
?,

$$(1.25322 - 1.25347)/0.0001 = -2.5$$

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

?,  
?,...]

## Strategy #2: Follow the slope

**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**,  
?,  
?,

$$(1.25353 - 1.25347)/0.0001 = 0.6$$

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

?,...]

## Strategy #2: Follow the slope

**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**,  
?,  
?

$$(1.25347 - 1.25347)/0.0001 = 0$$

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

?, ...]

## Strategy #2: Follow the slope

**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**,  
?,  
...]

Numeric Gradient is

- Slow. Needs to loop over all dimensions
- Approximates



**Strategy #2: Follow the slope**

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

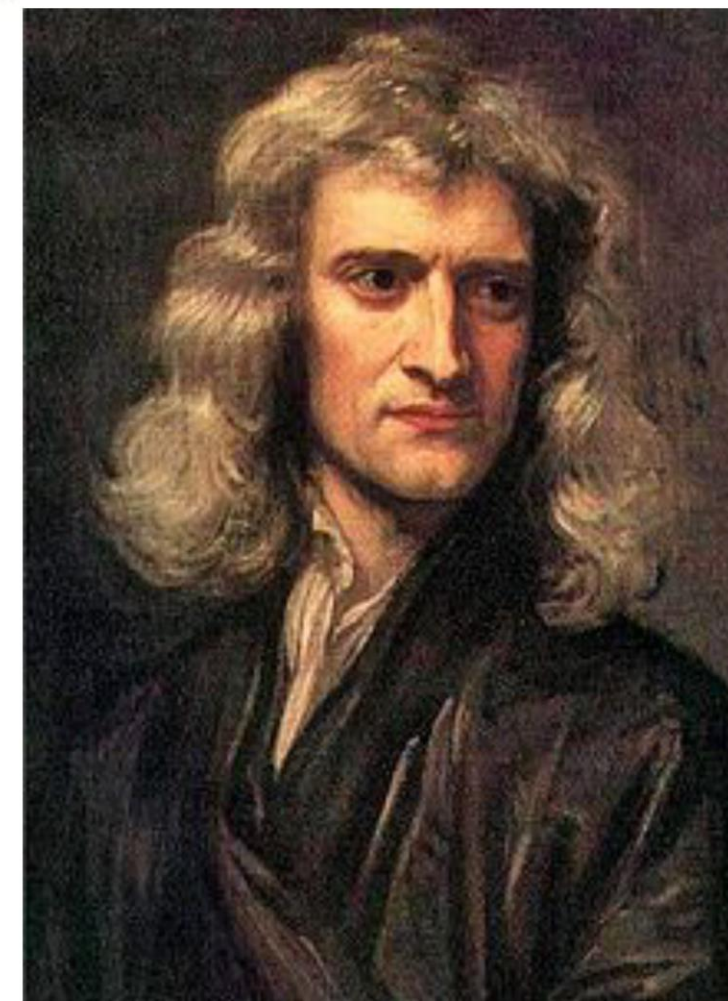
$$L = \frac{1}{N} \sum_{i=1}^N L_i + \sum_k W_k^2$$

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

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

want  $\nabla_W L$

Use calculus to compute an  
**analytic gradient**



[This image](#) is in the public domain



[This image](#) is in the public domain

## Optimization: *In summary*

**Numerical gradient:** approximate, slow, easy to write

**Analytic gradient:** exact, fast, error-prone

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

## Gradient Descent

**Gradient Descent** is an optimization algorithm used to minimize a loss function by iteratively adjusting the parameters of a model in the direction of steepest decrease of the loss function.

The algorithm works by first initializing the model parameters with random values, then iteratively computing the gradient of the loss function with respect to the parameters and updating the parameters in the direction that reduces the loss. This process continues until the gradient becomes very small or the maximum number of iterations is reached. The final set of parameters that minimize the loss is the result of the gradient descent optimization.

Gradient Descent is widely used in machine learning to train neural networks and other models. It can be implemented using different variations, such as batch gradient descent, stochastic gradient descent, and mini-batch gradient descent, depending on the size of the data set and computational resources.

## Gradient Descent

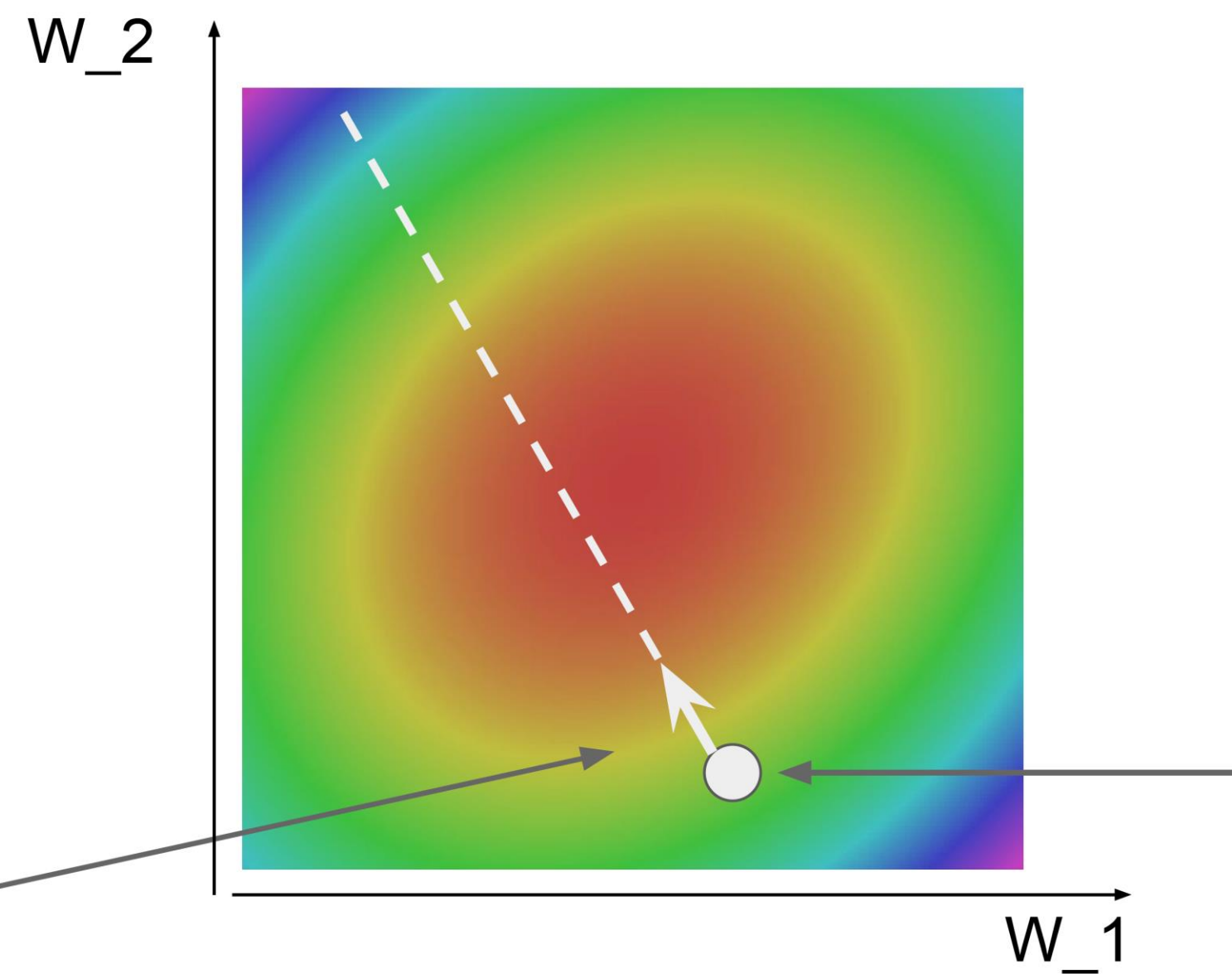
```
# Vanilla Gradient Descent

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

## Stochastic Gradient Descent

**Stochastic Gradient Descent** (SGD) is an optimization algorithm used to minimize a loss function in machine learning. It is an iterative method that updates the model parameters by computing the gradient of the loss with respect to the parameters and moving in the direction of negative gradient to reach the minimum. Unlike batch gradient descent, which computes the gradients based on the average of the entire training dataset, SGD updates the parameters using a single randomly selected sample at each iteration, making it computationally more efficient and suitable for training large-scale models

# Stochastic Gradient Descent



original W

negative gradient direction

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$



## Stochastic Gradient Descent

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch**  
of examples 32 / 64 / 128 common

```
# 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
```

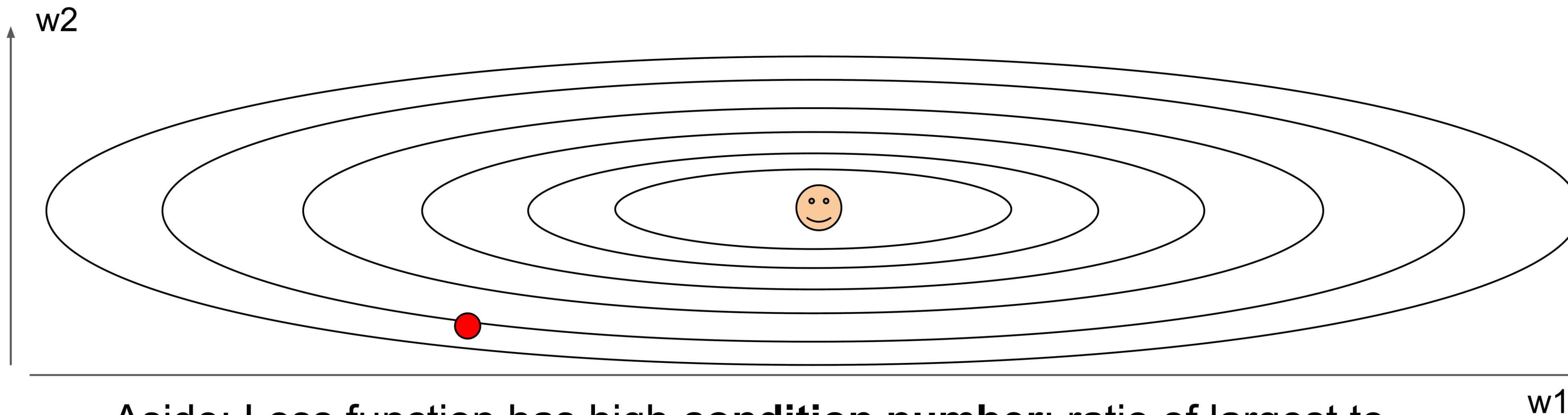
## Stochastic Gradient Descent

**Mini-batch gradient descent** is a variation of the gradient descent optimization algorithm for training machine learning models. Instead of updating the model parameters after evaluating the cost function on the entire training dataset, mini-batch gradient descent updates the parameters after evaluating the cost function on a smaller randomly selected subset of the training data, known as a mini-batch. This can result in faster convergence and a better optimization of the model parameters compared to batch gradient descent.



## Stochastic Gradient Descent

What if loss changes quickly in one direction and slowly in another?  
 What does gradient descent do?



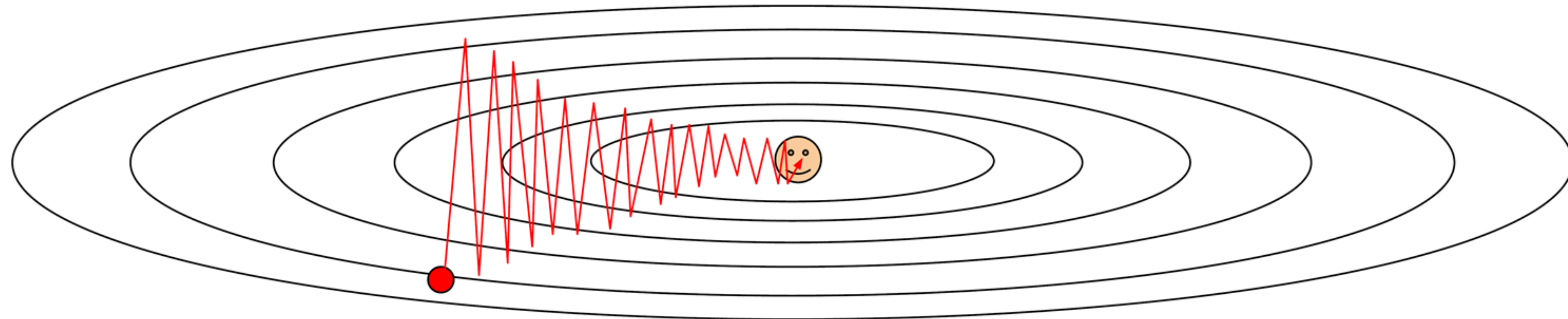
Aside: Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

## Stochastic Gradient Descent

What if loss changes quickly in one direction and slowly in another?

What does gradient descent do?

Very slow progress along shallow dimension, jitter along steep direction



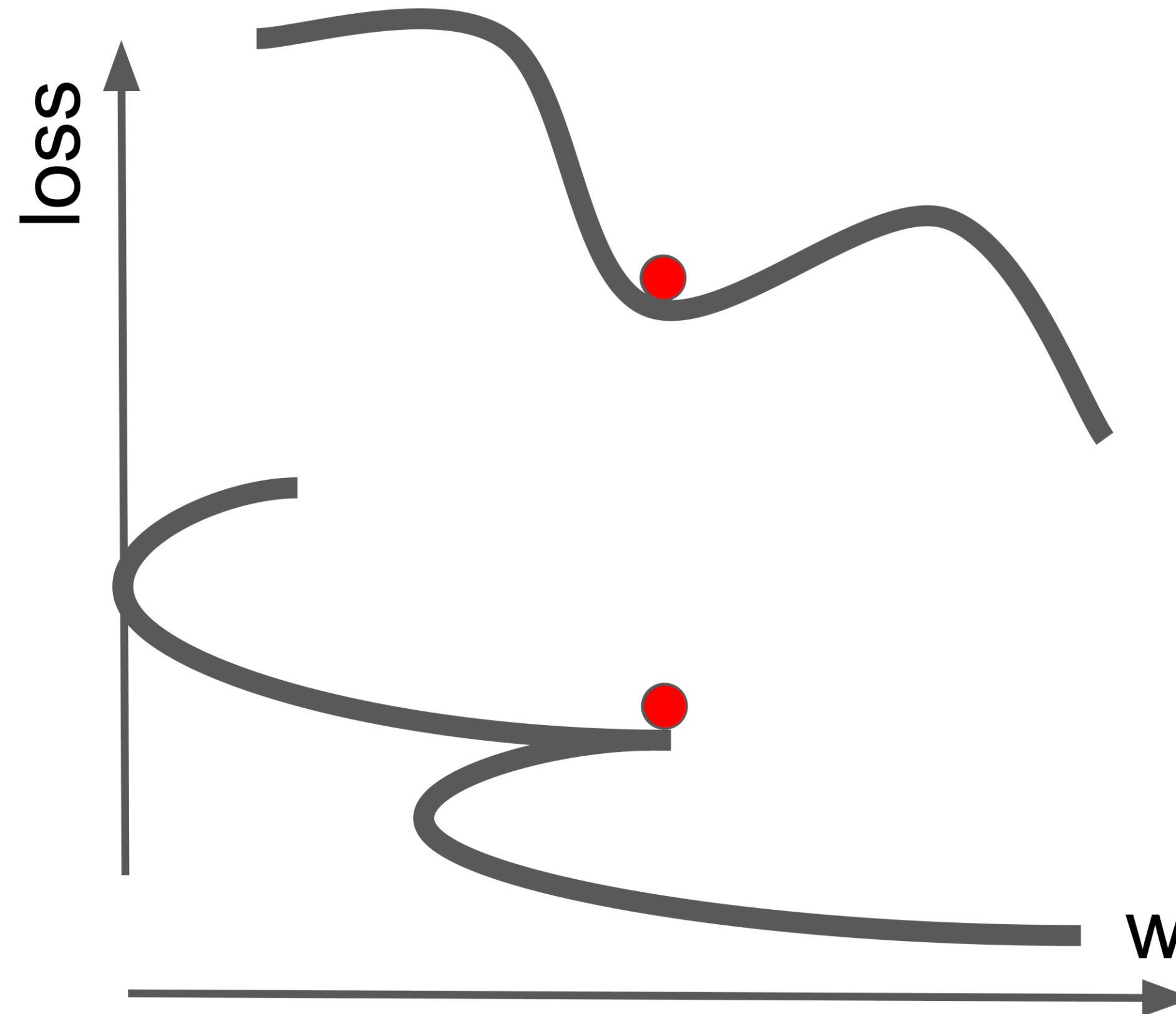
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

## Stochastic Gradient Descent

What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck

Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014



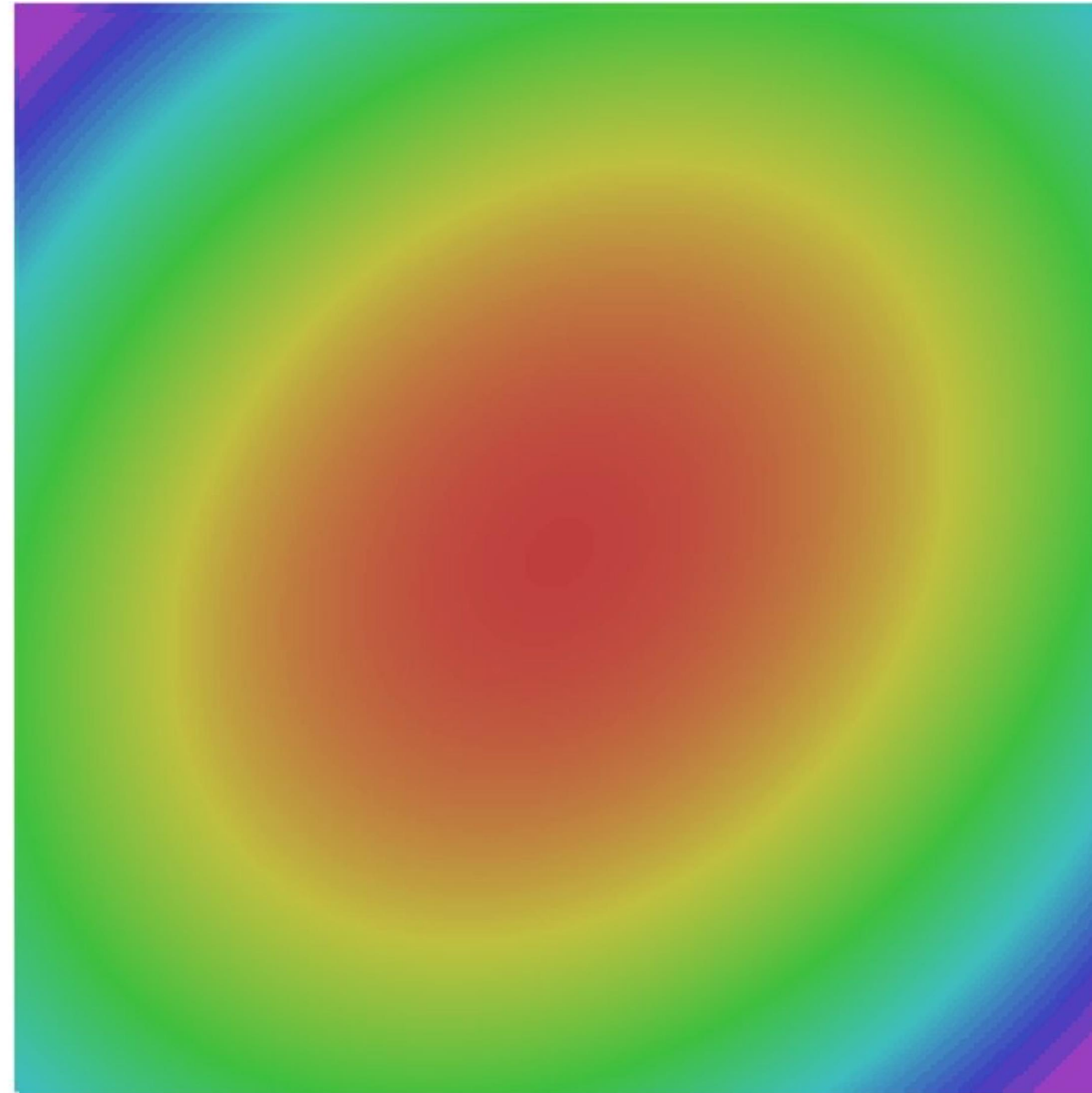
Saddle points much more common in high dimension

## Stochastic Gradient Descent

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W)$$



## SGD + Momentum

continue moving in the general direction as the previous iterations

### SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
    dx = compute_gradient(x)
    x -= learning_rate * dx
```

### SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x -= learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

## SGD + Momentum: *Alternative*

### SGD+Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx - learning_rate * dx
    x += vx
```

### SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

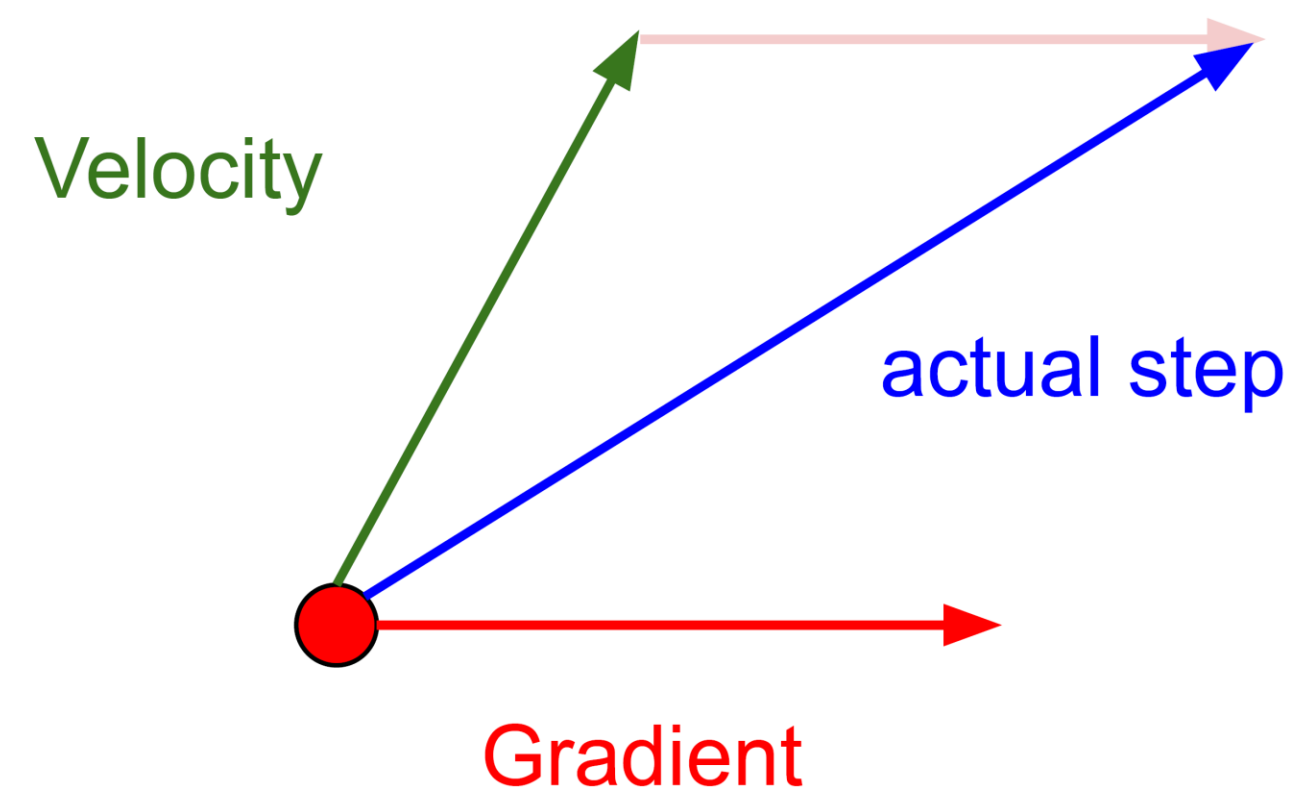
```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x -= learning_rate * vx
```

You may see SGD+Momentum formulated different ways,  
but they are equivalent - give same sequence of  $x$

Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

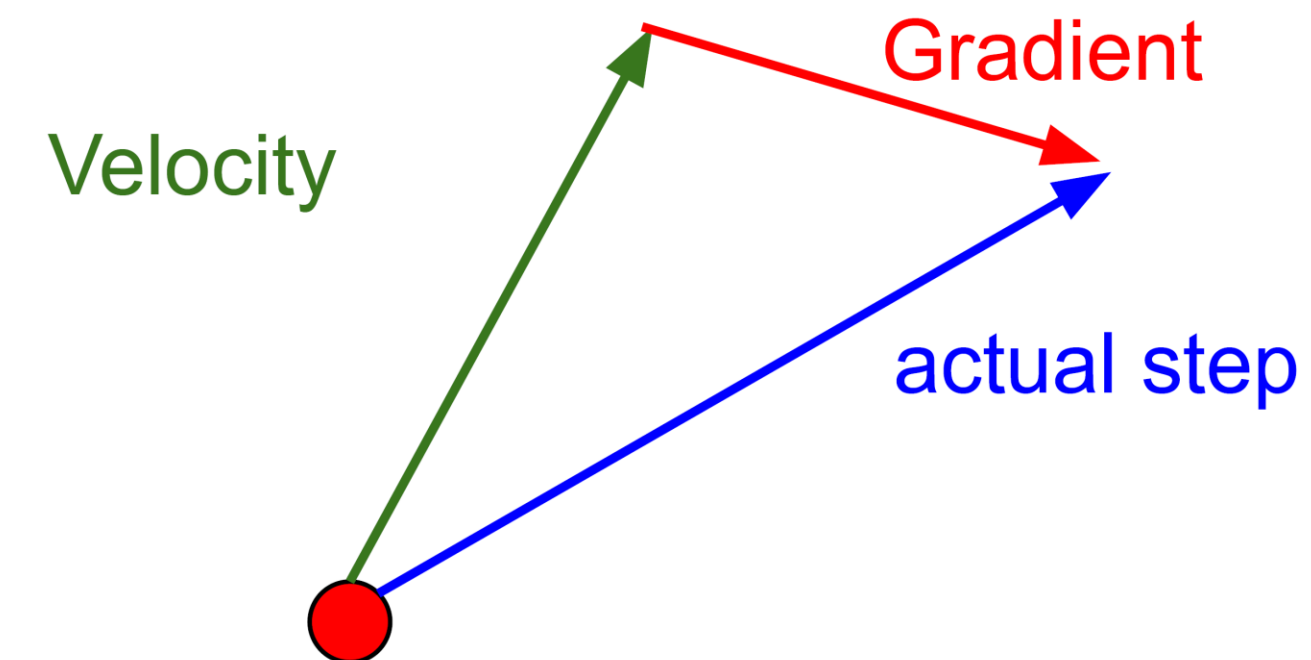
## Nesterov Momentum

Momentum update:



Combine gradient at current point with velocity to get step used to update weights

Nesterov Momentum



“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov, “A method of solving a convex programming problem with convergence rate  $O(1/k^2)$ ”, 1983

Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

## Nesterov Momentum

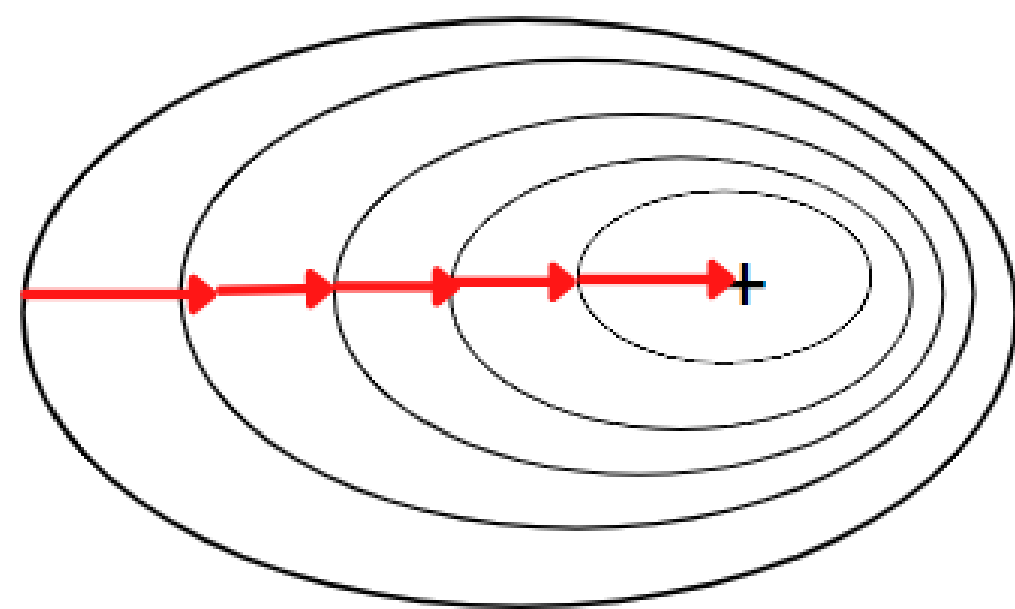
**Nesterov Momentum** is a variation of the classic momentum optimization algorithm in deep learning. It is an acceleration technique that adds a correction term to the update rule to address the issue of overshooting in traditional momentum optimization.

Nesterov Momentum uses the gradient of the future expected position in the weight space to compute the correction term, resulting in a more stable optimization trajectory and faster convergence to the optimal solution. It is a popular optimization technique that is widely used in training deep neural networks and has shown to perform well on various tasks.

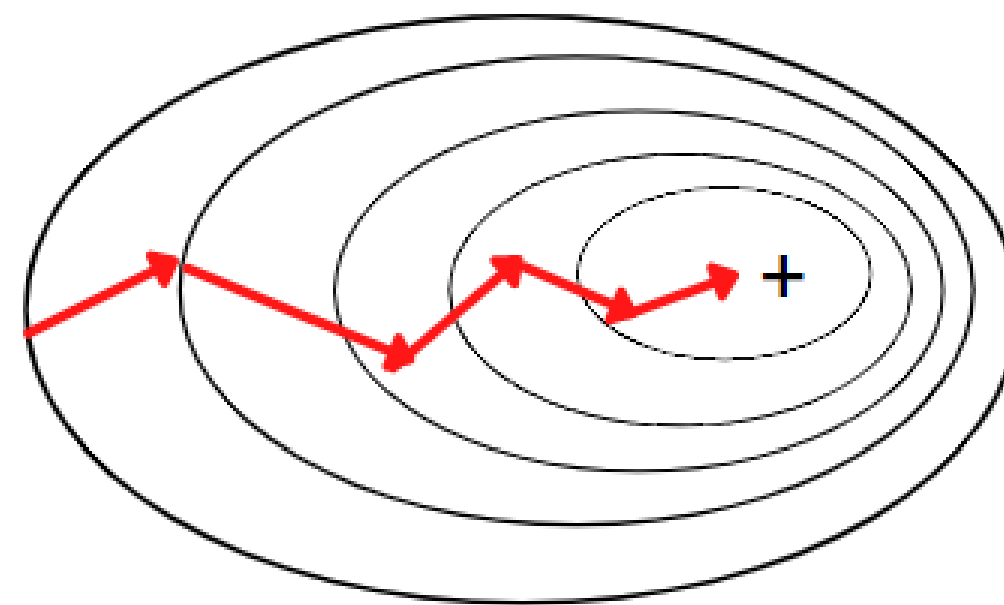


## Nesterov Momentum

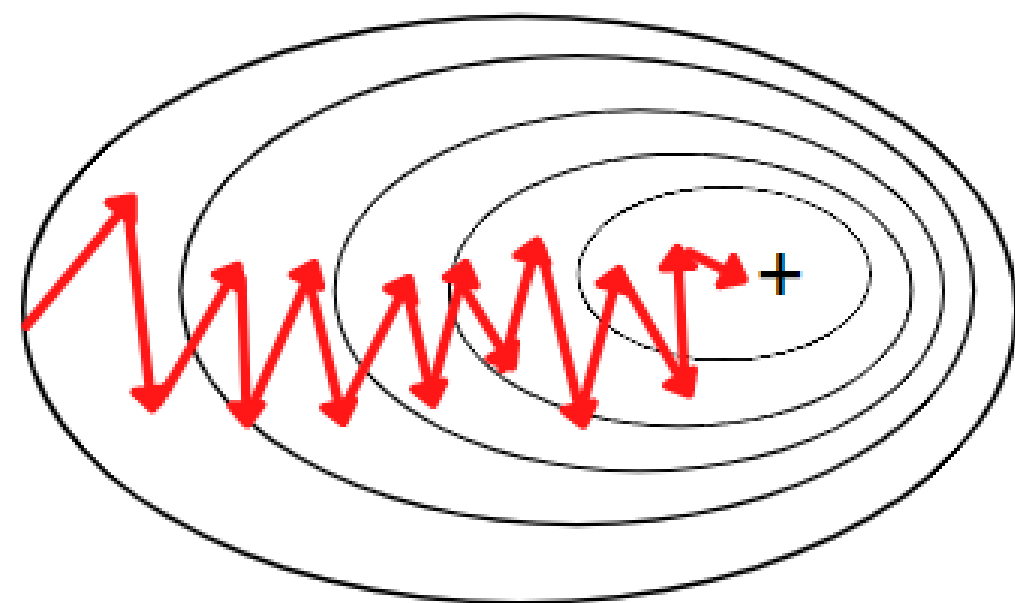
**Batch Gradient Descent**



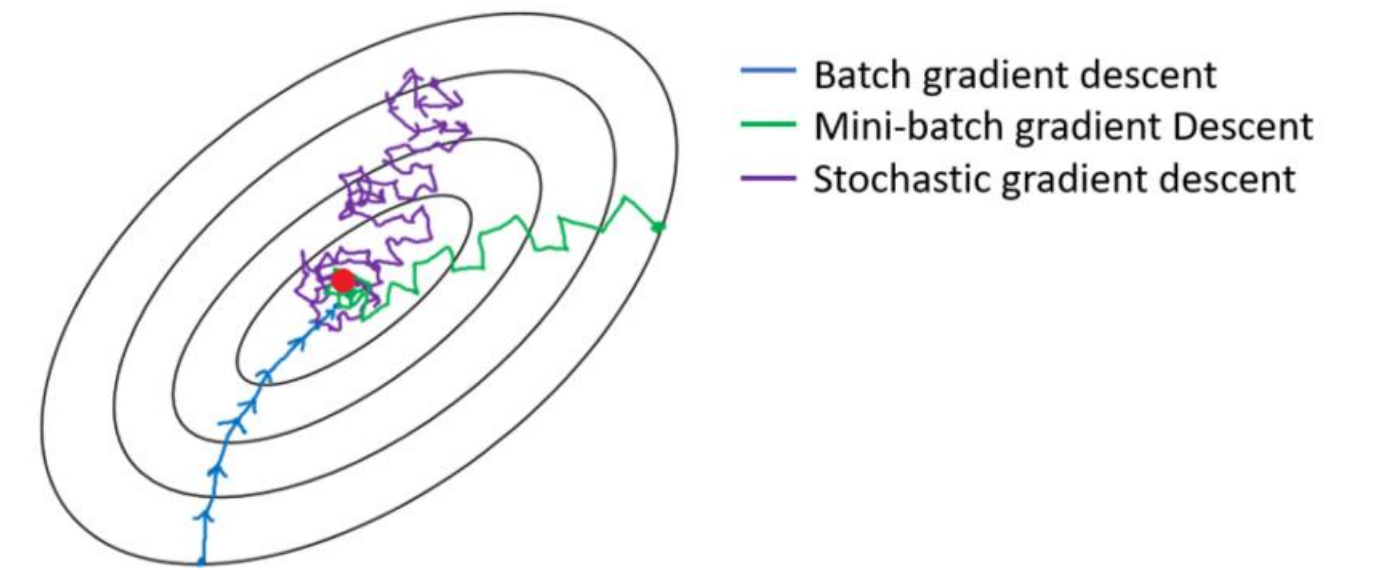
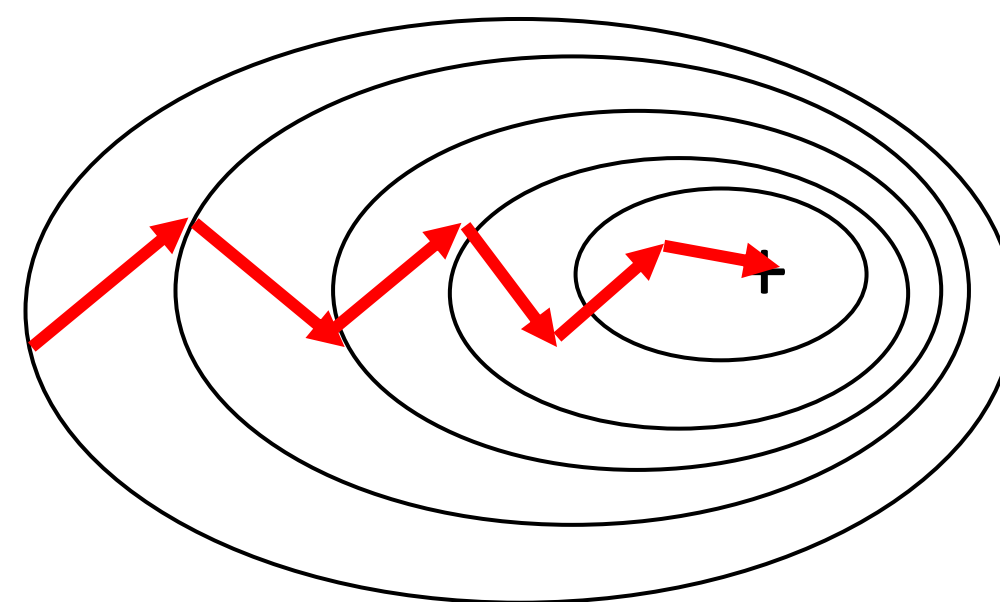
**Mini-Batch Gradient Descent**



**Stochastic Gradient Descent**



**Nesterov Momentum**



**BGD:** Slowest, with perfect gradient

**SGD:** Fastest, rough estimate of the gradient

**Mini-batch GD:** Compromise

<https://medium.datadriveninvestor.com/batch-vs-mini-batch-vs-stochastic-gradient-descent-with-code-examples-cd8232174e14>

## AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

“Per-parameter learning rates”  
or “adaptive learning rates”

**AdaGrad** is an optimization algorithm used in machine learning for training neural networks and other models. It stands for Adaptive Gradient Algorithm. It is designed to dynamically adjust the learning rate for each parameter during training, by scaling it inversely proportional to the historical gradient magnitude for that parameter. This means that the learning rate is reduced for parameters that have received a large gradient update in the past, while being increased for parameters that have received only small updates. This results in a more efficient optimization process compared to a fixed learning rate, as it can prevent overshooting or slow convergence for specific parameters.

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

## AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

“Per-parameter learning rates”  
or “adaptive learning rates”

Notice that the variable cache has size equal to the size of the gradient, and keeps track of per-parameter sum of squared gradients. This is then used to normalize the parameter update step, element-wise. Notice that the weights that receive high gradients will have their effective learning rate reduced, while weights that receive small or infrequent updates will have their effective learning rate increased. Amusingly, the square root operation turns out to be very important and without it the algorithm performs much worse. The smoothing term eps (usually set somewhere in range from  $1e-4$  to  $1e-8$ ) avoids division by zero. A downside of Adagrad is that in case of Deep Learning, the monotonic learning rate usually proves too aggressive and stops learning too early.

## RMSProp

AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

The RMSProp update adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate.

RMSProp

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

# Adam

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

## Adam

```

first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7)

```

Momentum

Bias correction

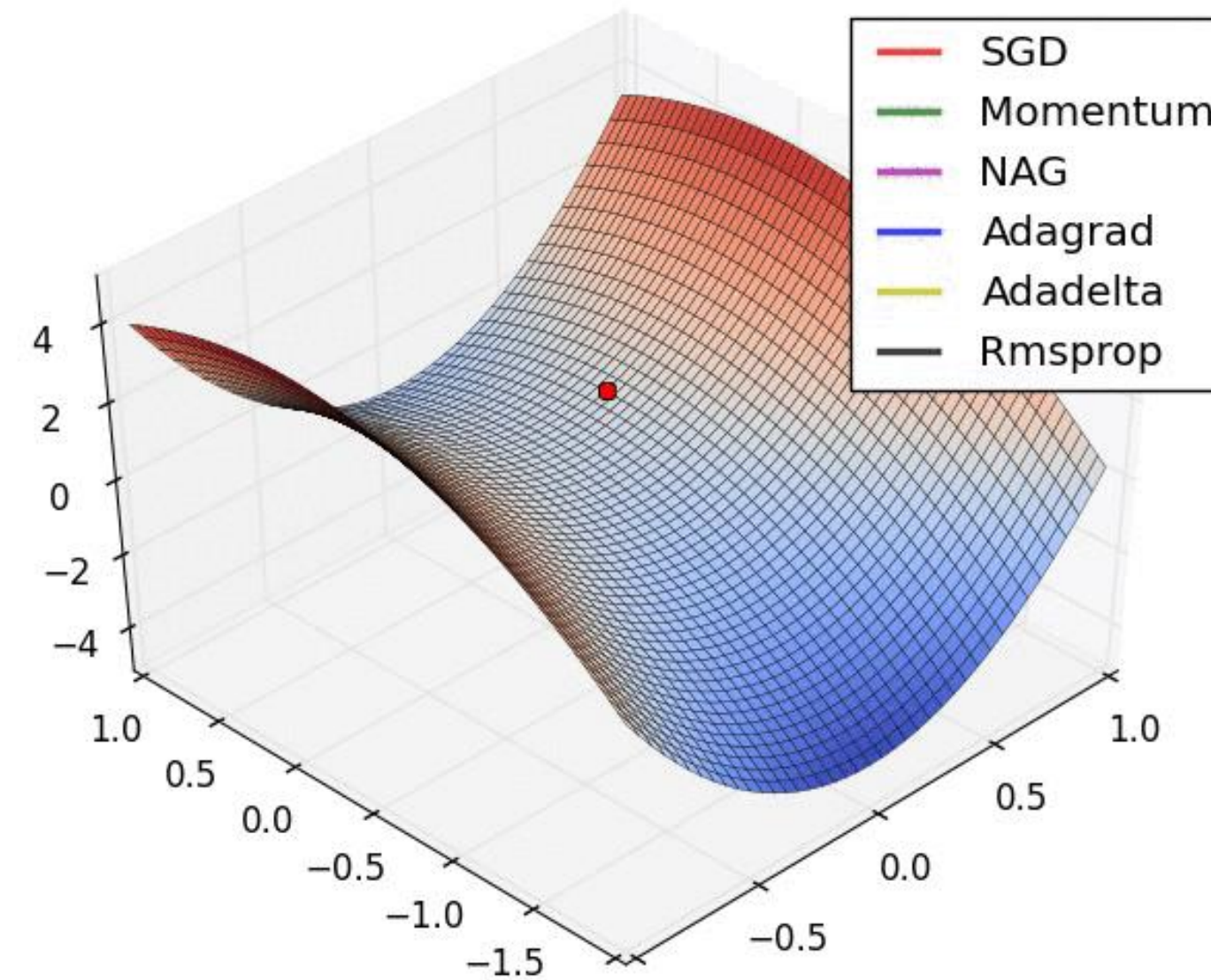
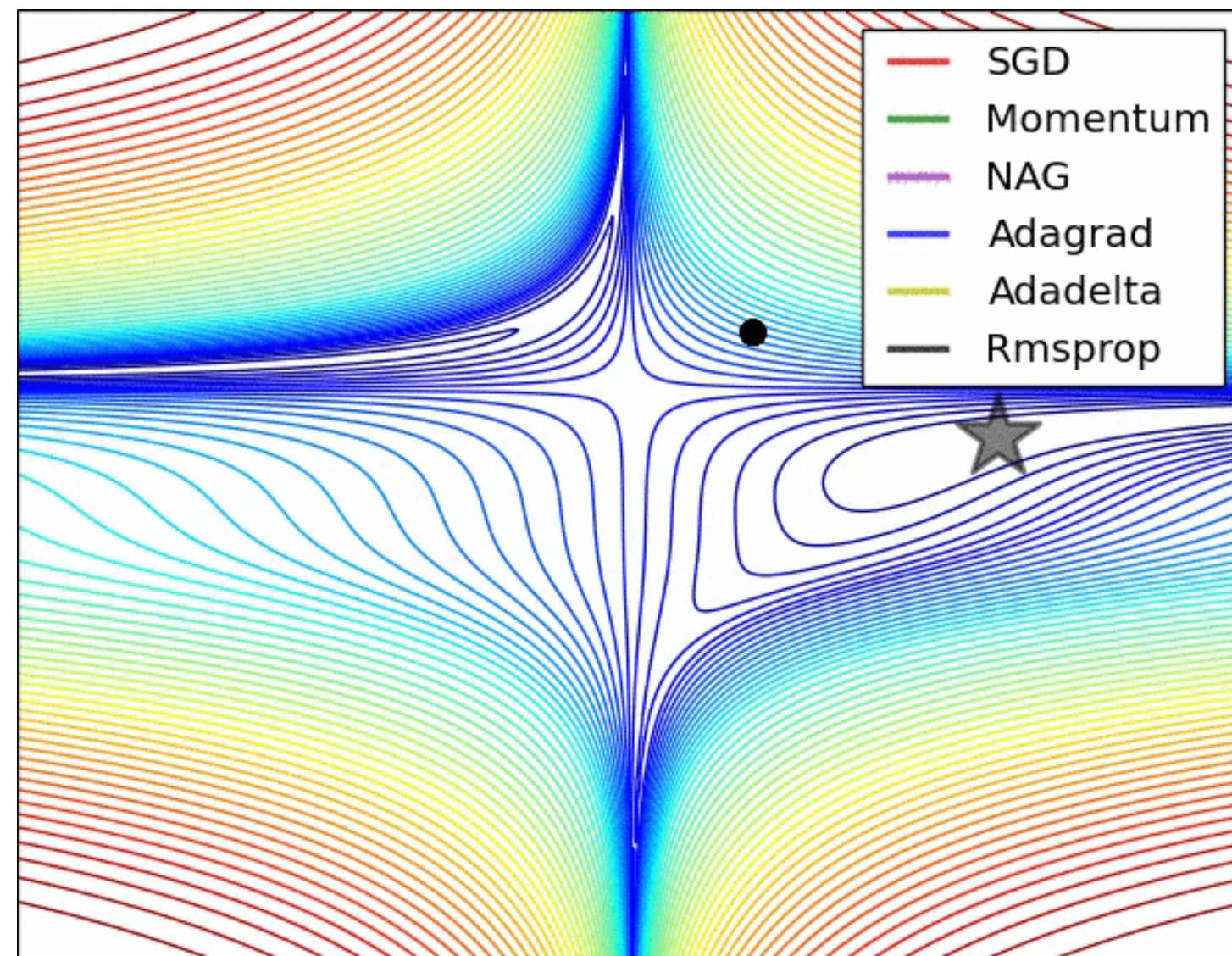
AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

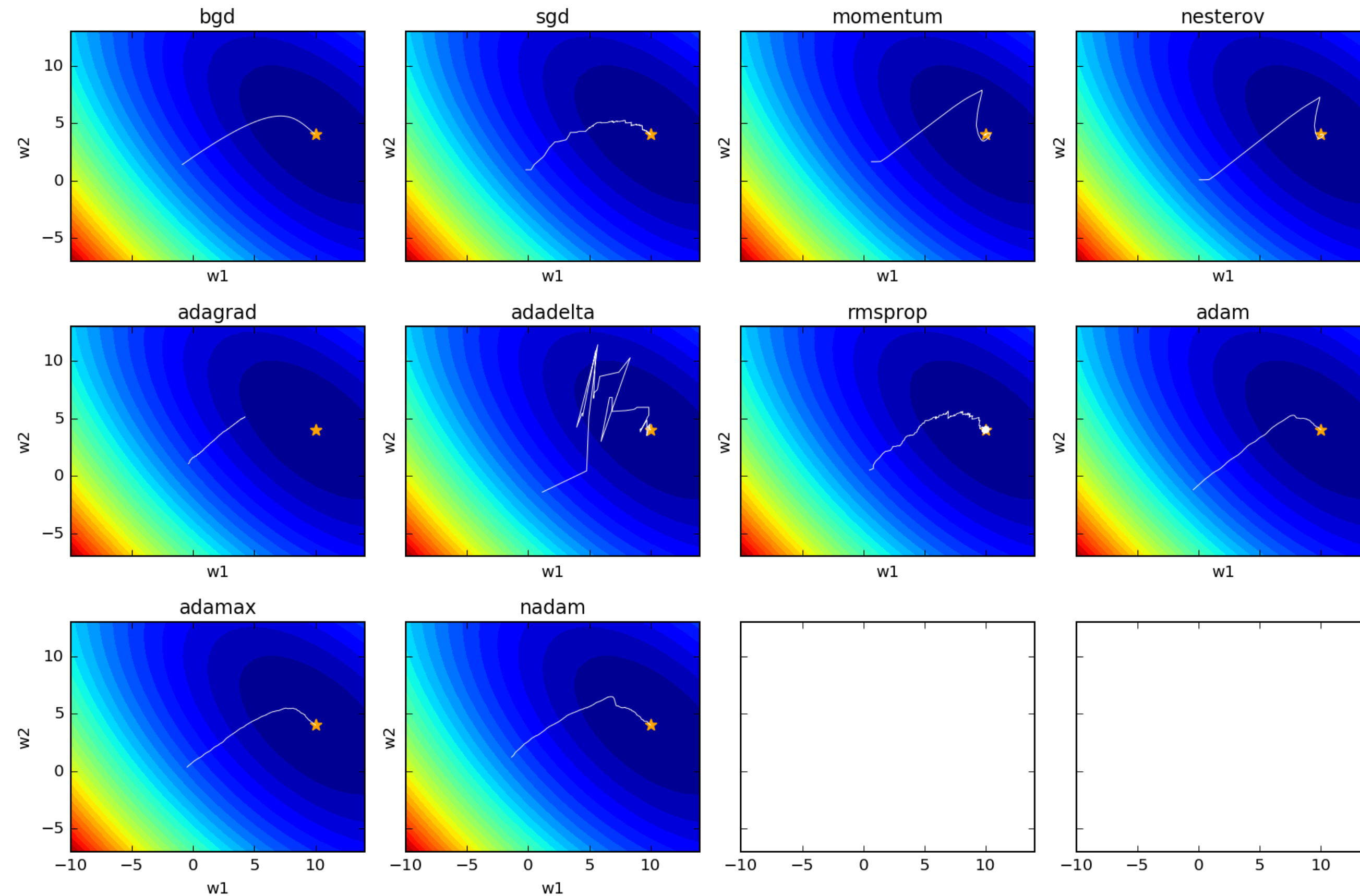
Adam with  $\text{beta1} = 0.9$ ,  
 $\text{beta2} = 0.999$ , and  $\text{learning\_rate} = 1e-3$  or  $5e-4$   
is a great starting point for many models!

## Visualize various gradient descent algorithms



Contours of a loss surface and time evolution of different optimization algorithms. Notice the "overshooting" behavior of momentum-based methods, which make the optimization look like a ball rolling down the hill. **Right:** 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. Images credit: [Alec Radford](#).

## Visualize various gradient descent algorithms



[https://github.com/zyxue/sutton-barto-rl-exercises/tree/master/supervised/gradient\\_descent](https://github.com/zyxue/sutton-barto-rl-exercises/tree/master/supervised/gradient_descent)



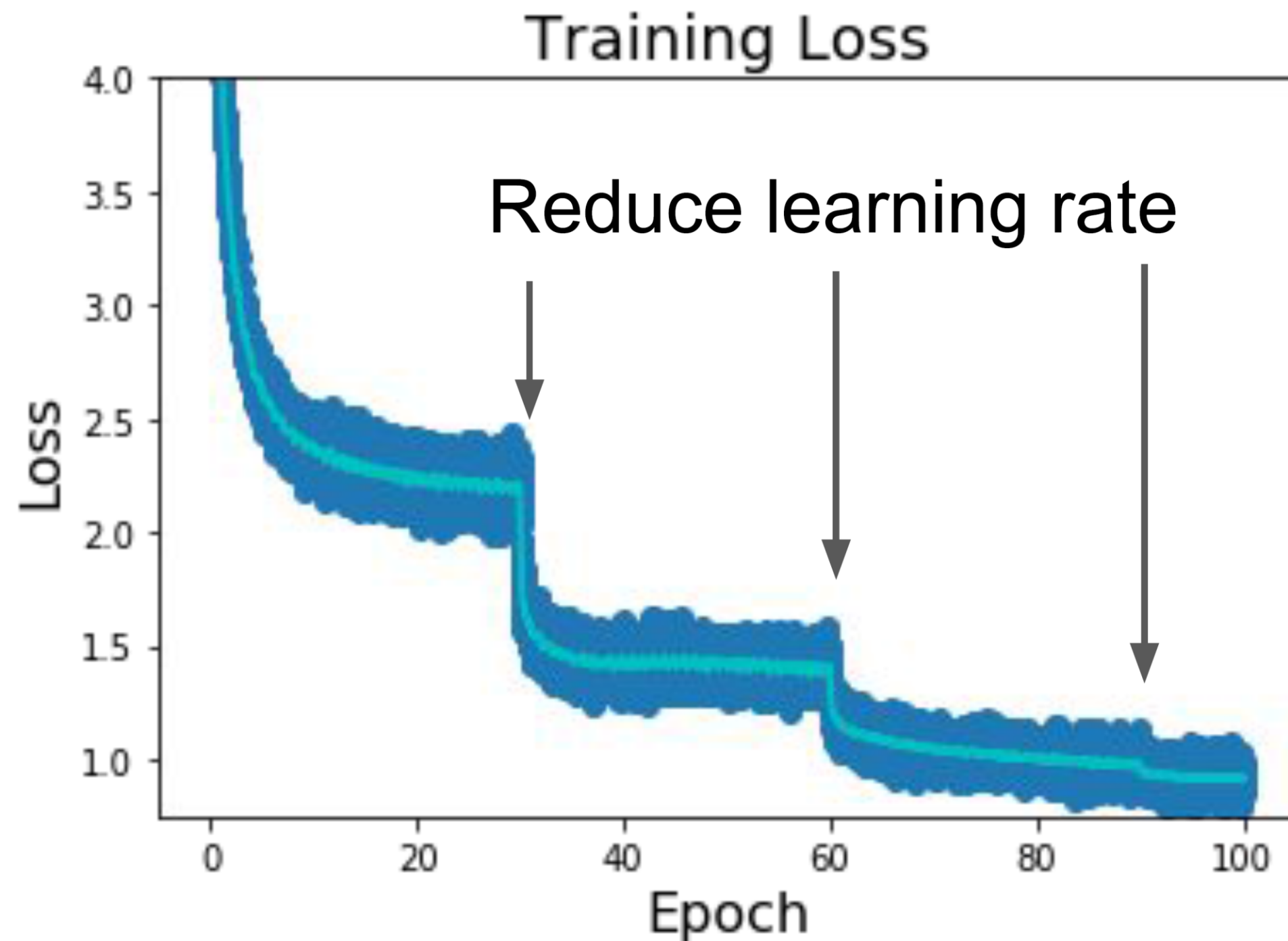


## Hyperparameter optimization

**Learning rate decay** over time refers to a method in training machine learning models, where the learning rate is gradually reduced during the optimization process. This is typically done by dividing the learning rate by a factor at certain intervals or epochs. The purpose of learning rate decay is to slow down the optimization process as it approaches a minimum in the loss function, so that the model parameters converge more precisely to the optimal values.

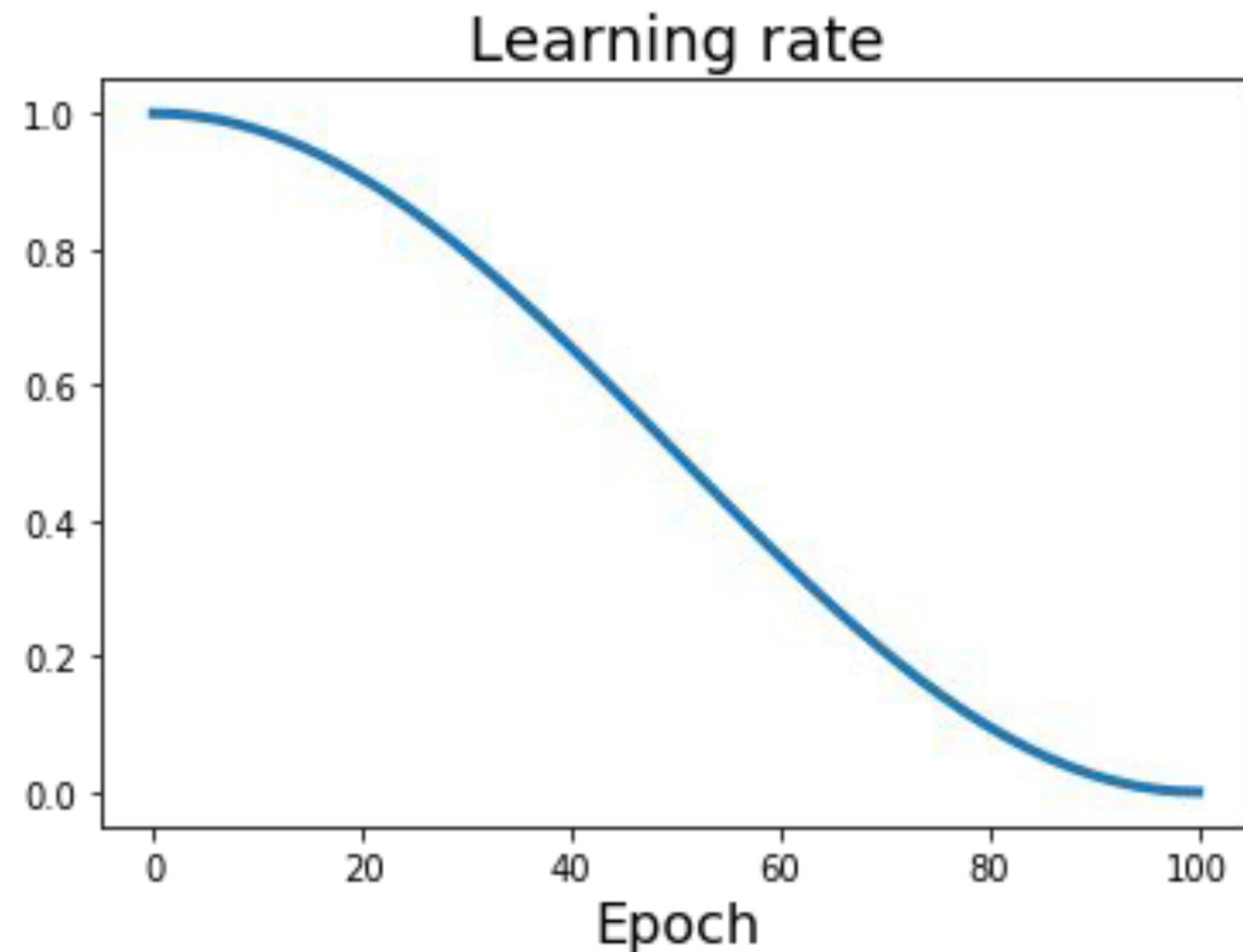
Learning rate decay helps to overcome the problem of oscillation or overshooting that can occur when the learning rate is too high, especially in the later stages of training. It allows the optimization process to converge more smoothly to the optimal values and avoid the risk of getting stuck in suboptimal local minima.

## Learning rate decay



**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

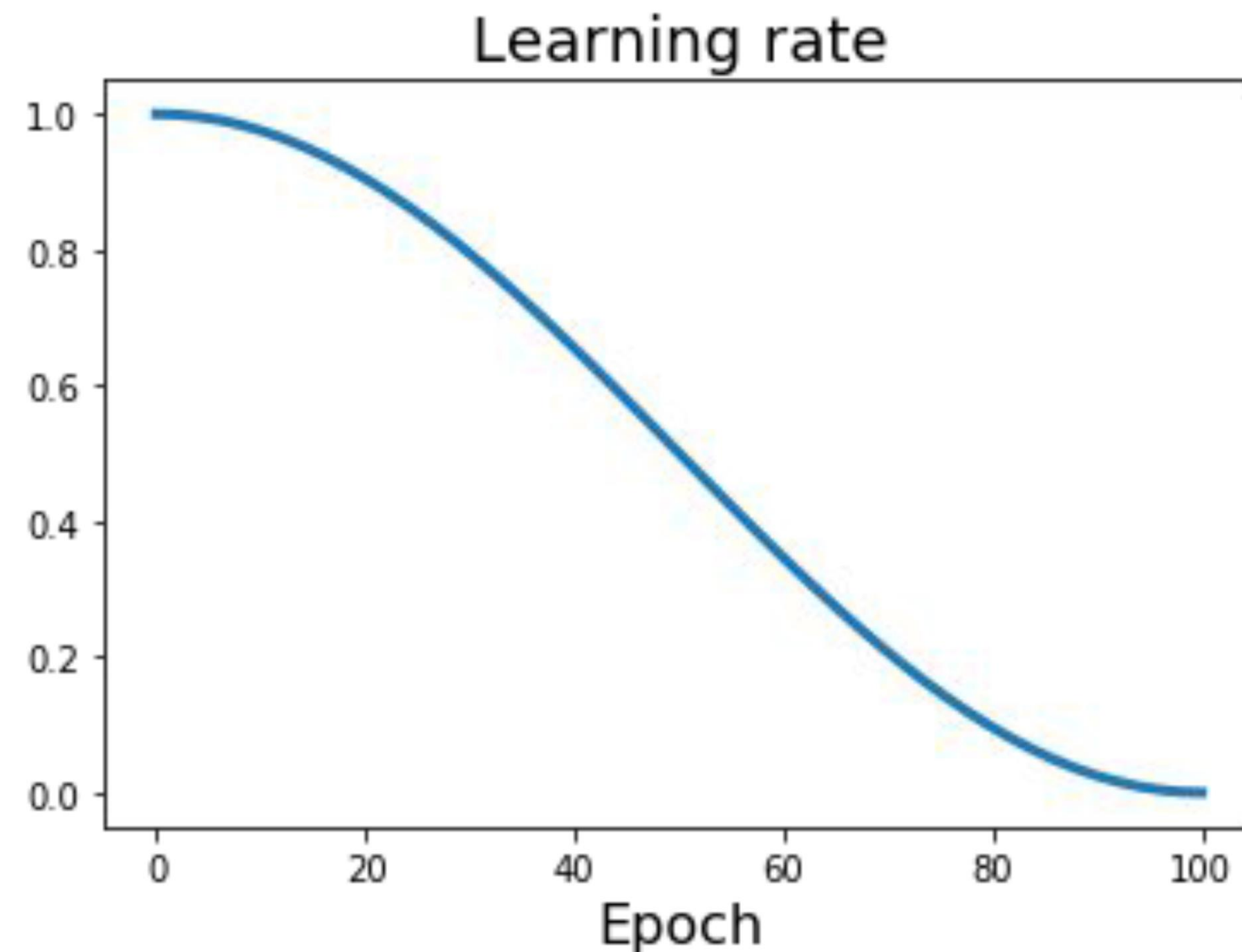
## Learning rate decay



**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**  $\alpha_t = \frac{1}{2} \alpha_0 (1 + \cos(t\pi/T))$

## Learning rate decay



**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** 
$$\alpha_t = \frac{1}{2} \alpha_0 (1 + \cos(t\pi/T))$$

$\alpha_0$  : Initial learning rate

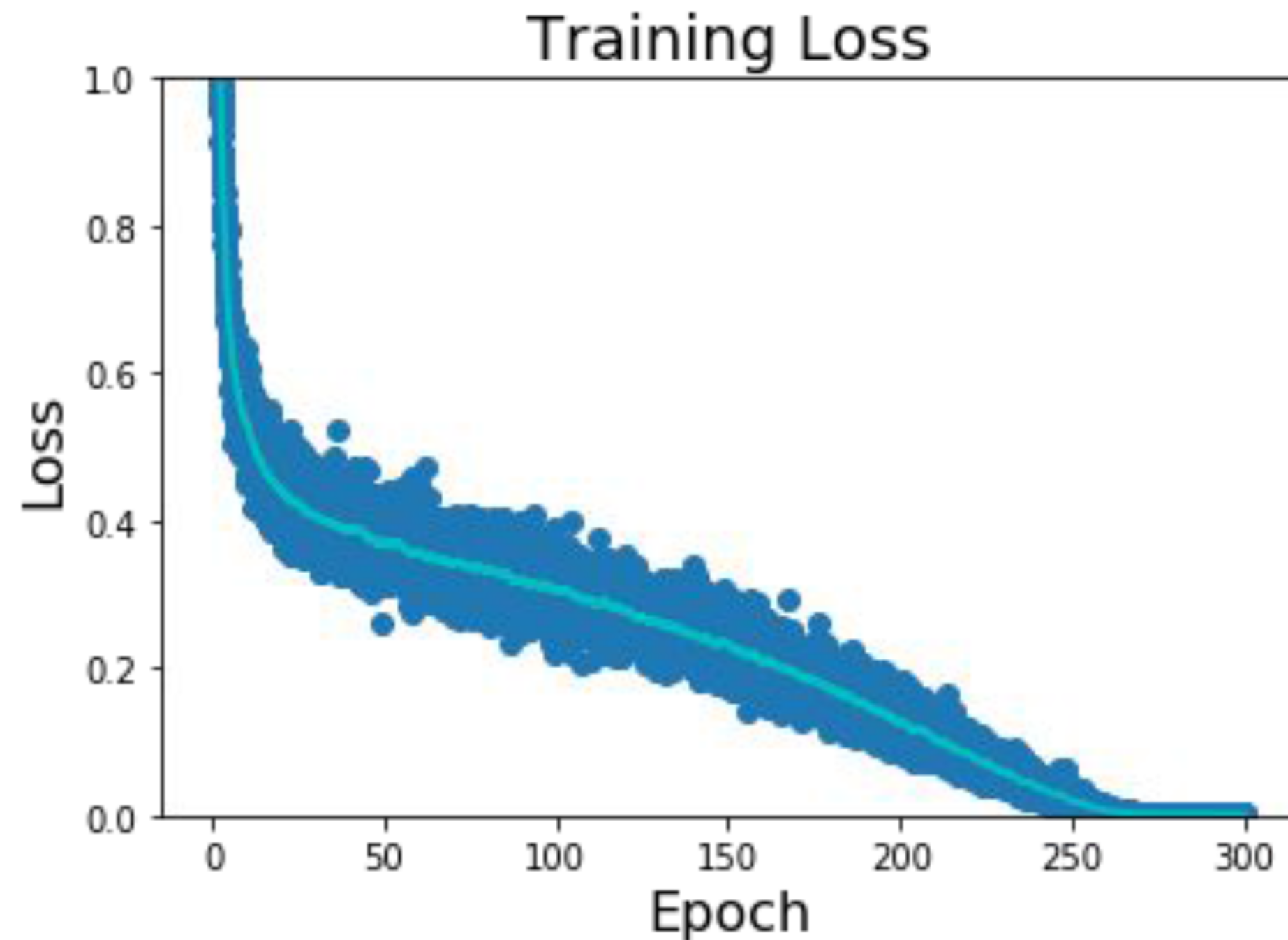
$\alpha_t$  : Learning rate at epoch  $t$

$T$  : Total number of epochs

Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017  
 Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018  
 Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018  
 Child et al, "Generating Long Sequences with Sparse Transformers", arXiv 2019



## Learning rate decay



**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:** 
$$\alpha_t = \frac{1}{2} \alpha_0 (1 + \cos(t\pi/T))$$

$\alpha_0$  : Initial learning rate

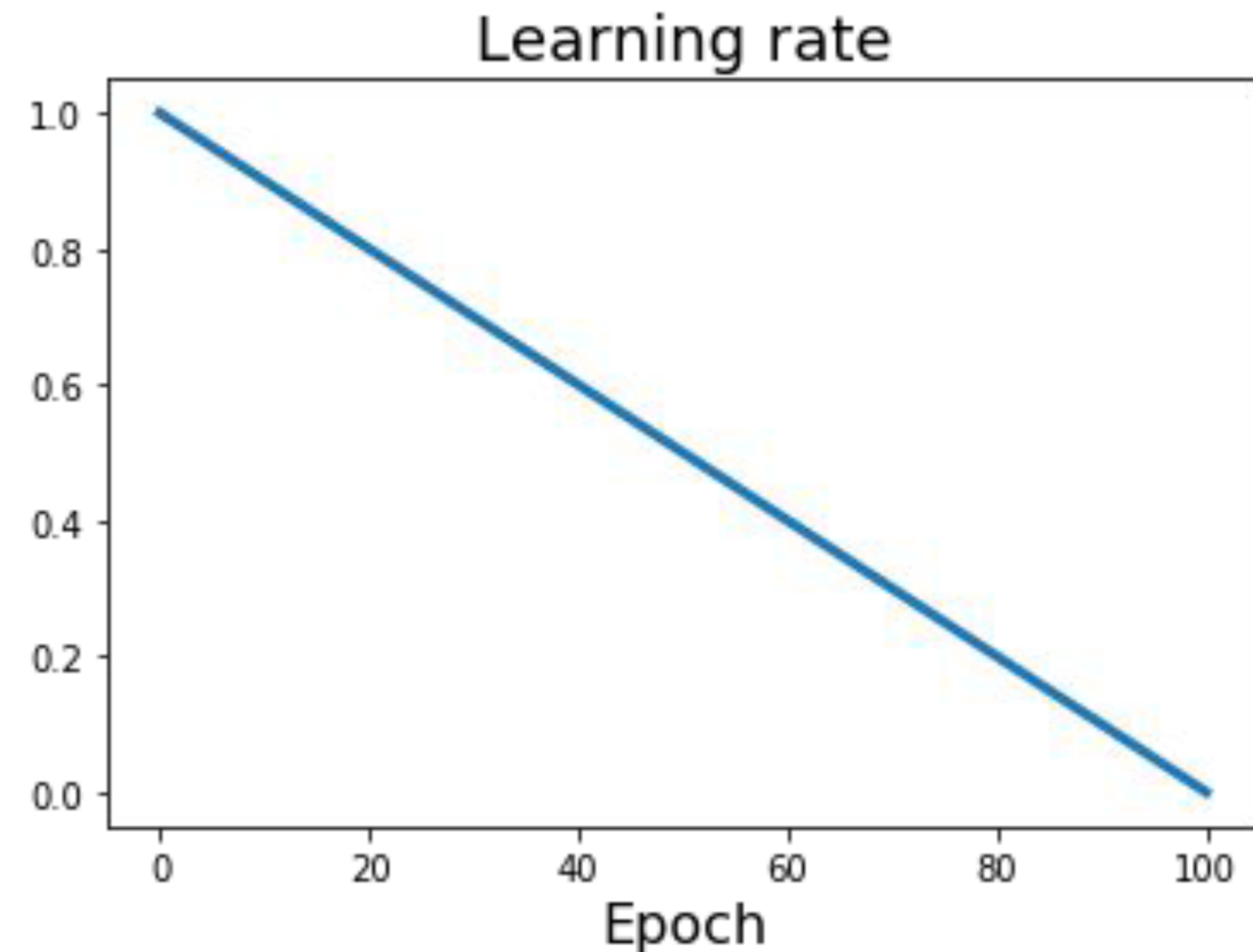
$\alpha_t$  : Learning rate at epoch  $t$

$T$  : Total number of epochs

Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017  
 Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018  
 Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018  
 Child et al, "Generating Long Sequences with Sparse Transformers", arXiv 2019



## Learning rate decay



**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**  $\alpha_t = \frac{1}{2} \alpha_0 (1 + \cos(t\pi/T))$

**Linear:**  $\alpha_t = \alpha_0 (1 - t/T)$

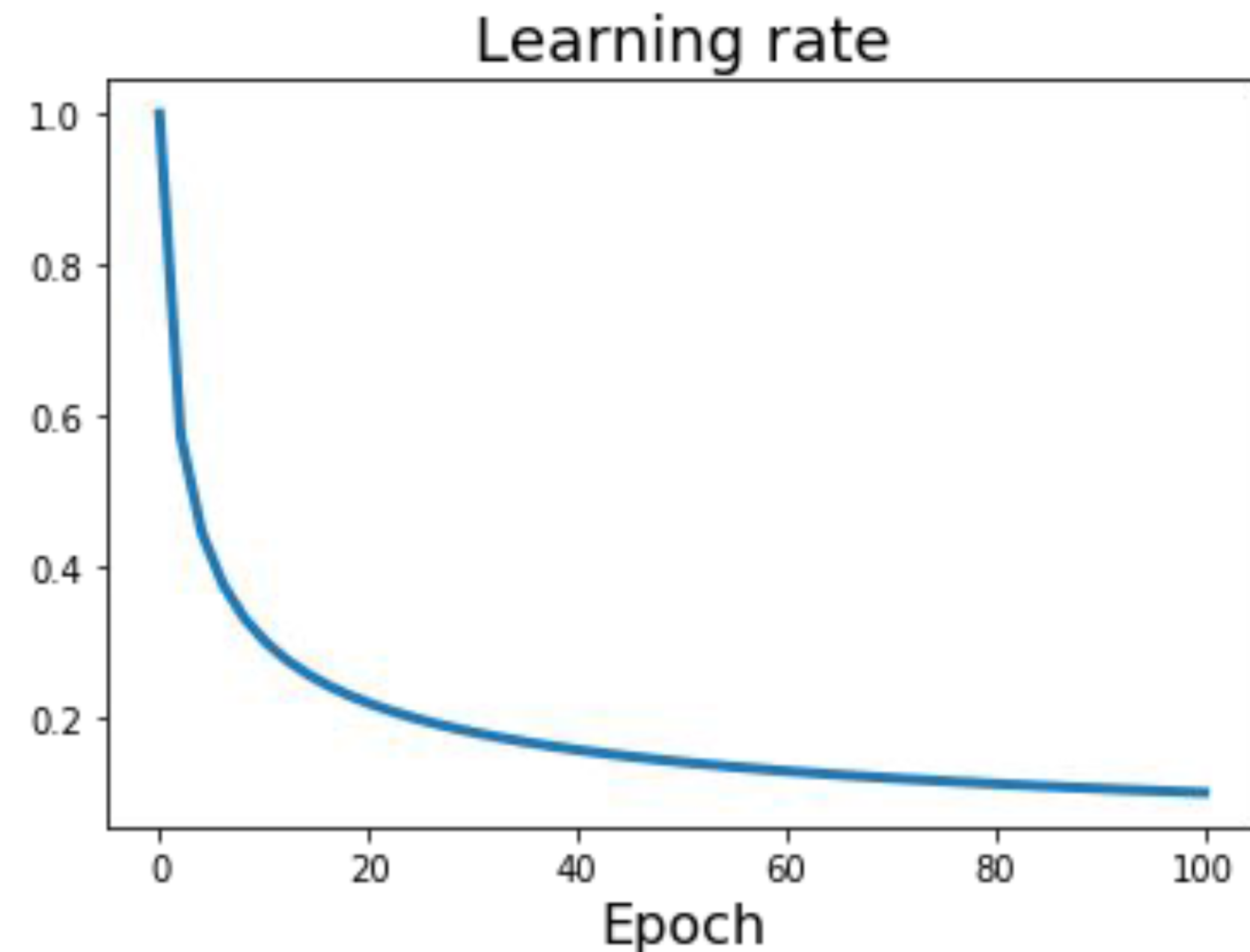
$\alpha_0$  : Initial learning rate

$\alpha_t$  : Learning rate at epoch  $t$

$T$  : Total number of epochs

Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018

## Learning rate decay



**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**  $\alpha_t = \frac{1}{2} \alpha_0 (1 + \cos(t\pi/T))$

**Linear:**  $\alpha_t = \alpha_0 (1 - t/T)$

**Inverse sqrt:**  $\alpha_t = \alpha_0 / \sqrt{t}$

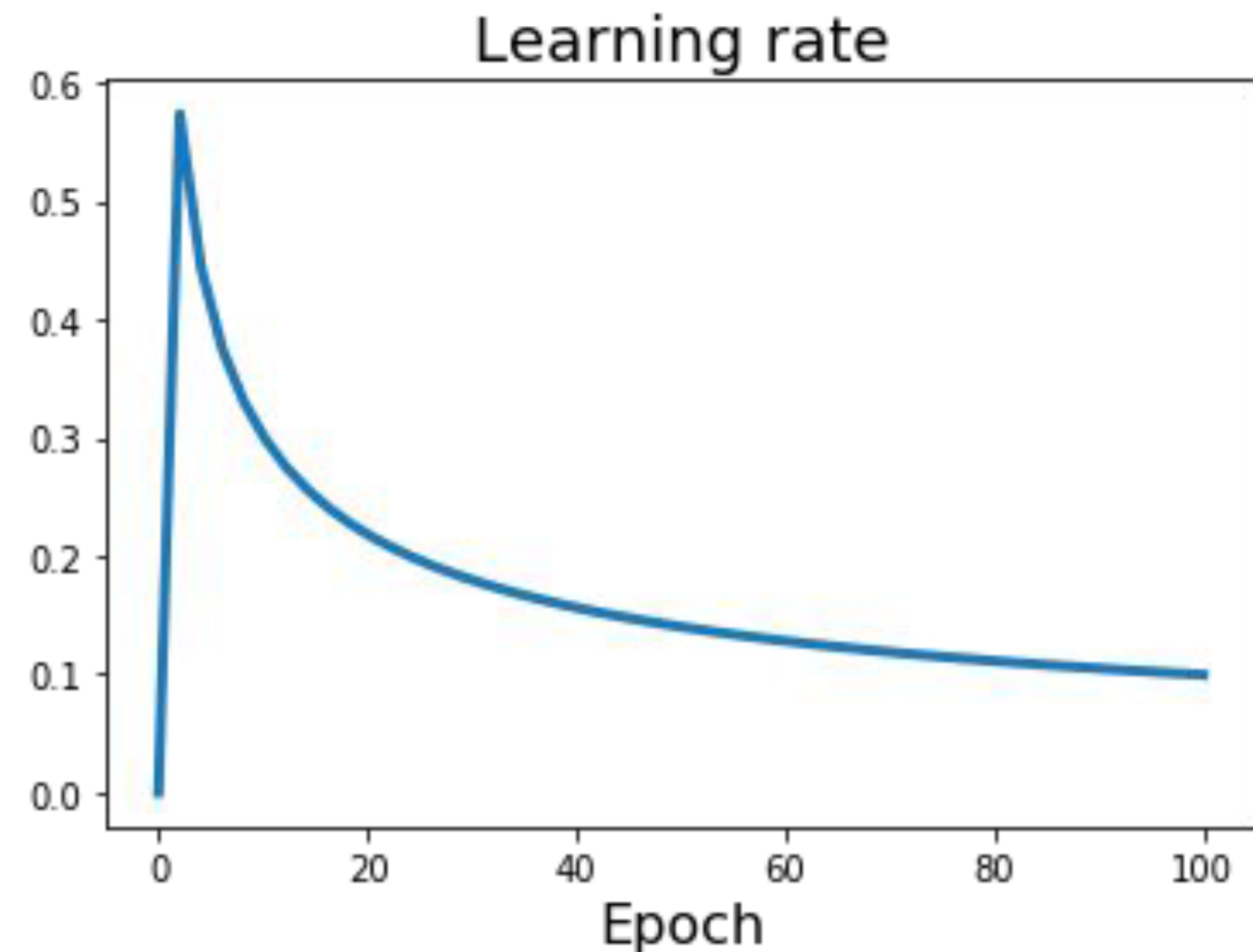
$\alpha_0$  : Initial learning rate

$\alpha_t$  : Learning rate at epoch  $t$

$T$  : Total number of epochs

Vaswani et al, "Attention is all you need", NIPS 2017

## Learning rate decay: *Linear Warmup*



High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first  $\sim 5,000$  iterations can prevent this.

Empirical rule of thumb: If you increase the batch size by  $N$ , also scale the initial learning rate by  $N$

Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017



## Limited-memory BFGS

In machine learning, **L-BFGS** is used to optimize the loss function of the model by updating the model parameters to minimize the loss. It is an iterative optimization method that uses gradient information to iteratively improve the model parameters. Unlike other optimization methods like gradient descent, L-BFGS requires less memory to store intermediate results, as it only needs to store a limited number of gradient updates.

L-BFGS is often preferred over gradient descent because it can converge faster and more accurately to the optimal solution. It is especially effective for large-scale optimization problems where the computation of the Hessian matrix is impractical, as it provides a good approximation of the second-order information. However, it can be computationally expensive compared to other optimization methods, as it requires more calculations for each iteration.

## In practice

- Adam is a good default choice in many cases; it often works ok even with constant learning rate
- SGD+Momentum can outperform Adam but may require more tuning of LR and schedule
- If you can afford to do full batch updates then try out L-BFGS (and don't forget to disable all sources of noise)

# Neural Networks

**Neural networks: *the original linear classifier***

**(Before)** Linear score function:  $f = Wx$

$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

Neural networks: *2 layers*

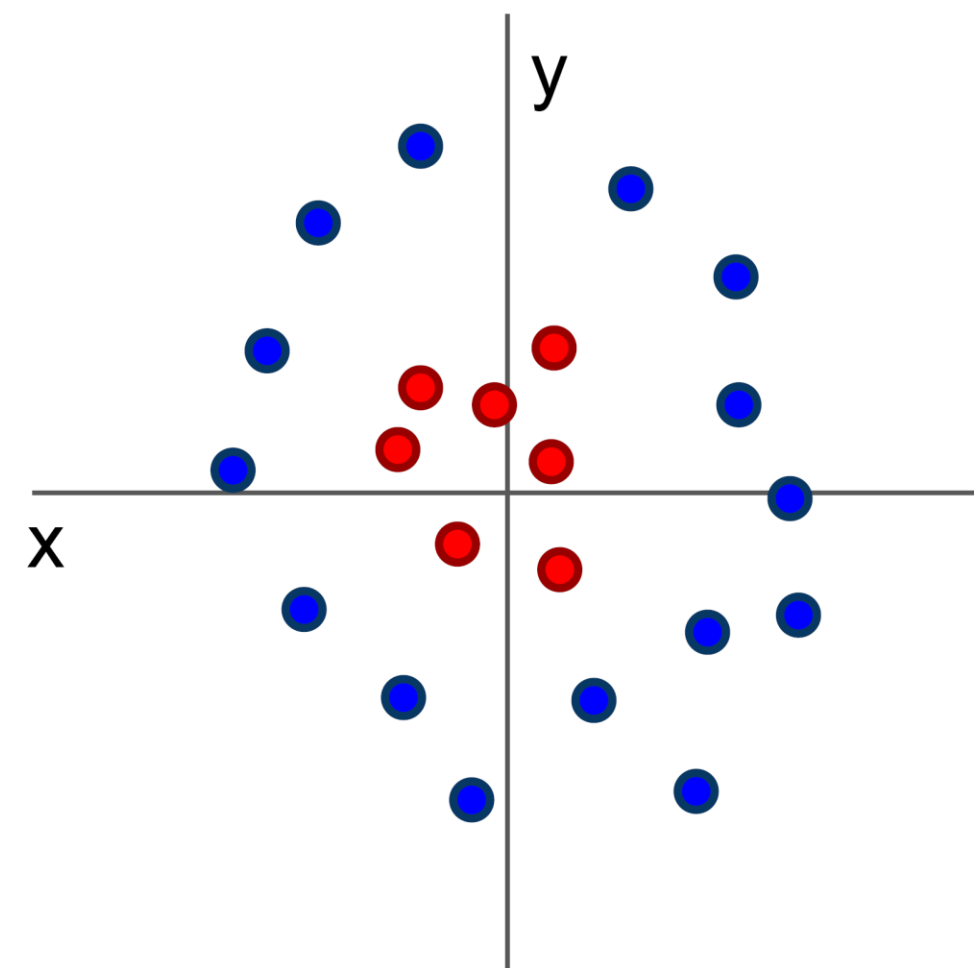
**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$  The function  $\max(0, -)$  is a non-linearity that is applied elementwise.

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

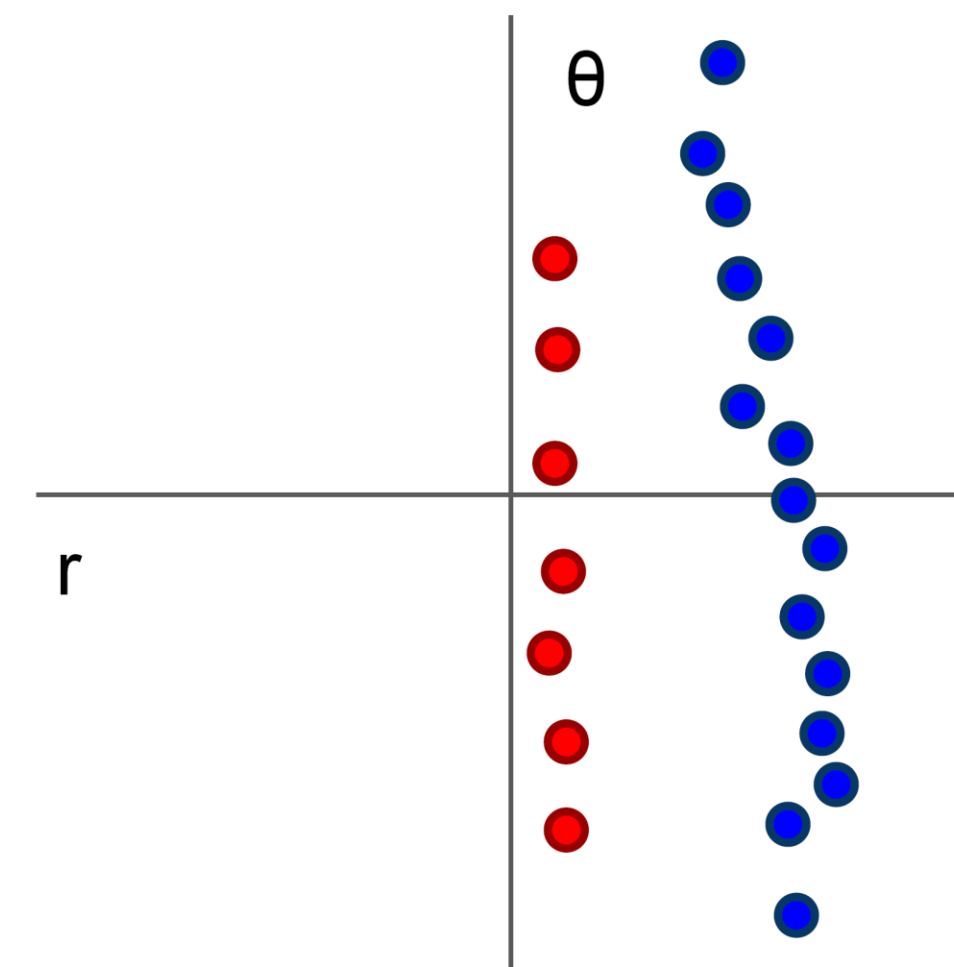
(In practice we will usually add a learnable bias at each layer as well)

## Why do we want non-linearity?



Cannot separate red and blue points with linear classifier

$$f(x, y) = (r(x, y), \theta(x, y))$$



After applying feature transform, points can be separated by linear classifier

**Neural networks: also called fully connected network**

The non-linearity is where we get the *wiggle*. The parameters  $W_2, W_1$  are learned with stochastic gradient descent, and their gradients are derived with chain rule (and computed with backpropagation).

**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

“Neural Network” is a very broad term; these are more accurately called “fully-connected networks” or sometimes “multi-layer perceptrons” (MLP)

(In practice we will usually add a learnable bias at each layer as well)

## Neural networks: 3 layers

**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  
or 3-layer Neural Network  $f = W_2 \max(0, W_1 x)$

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H_1 \times D}, W_2 \in \mathbb{R}^{H_2 \times H_1}, W_3 \in \mathbb{R}^{C \times H_2}$$

(In practice we will usually add a learnable bias at each layer as well)

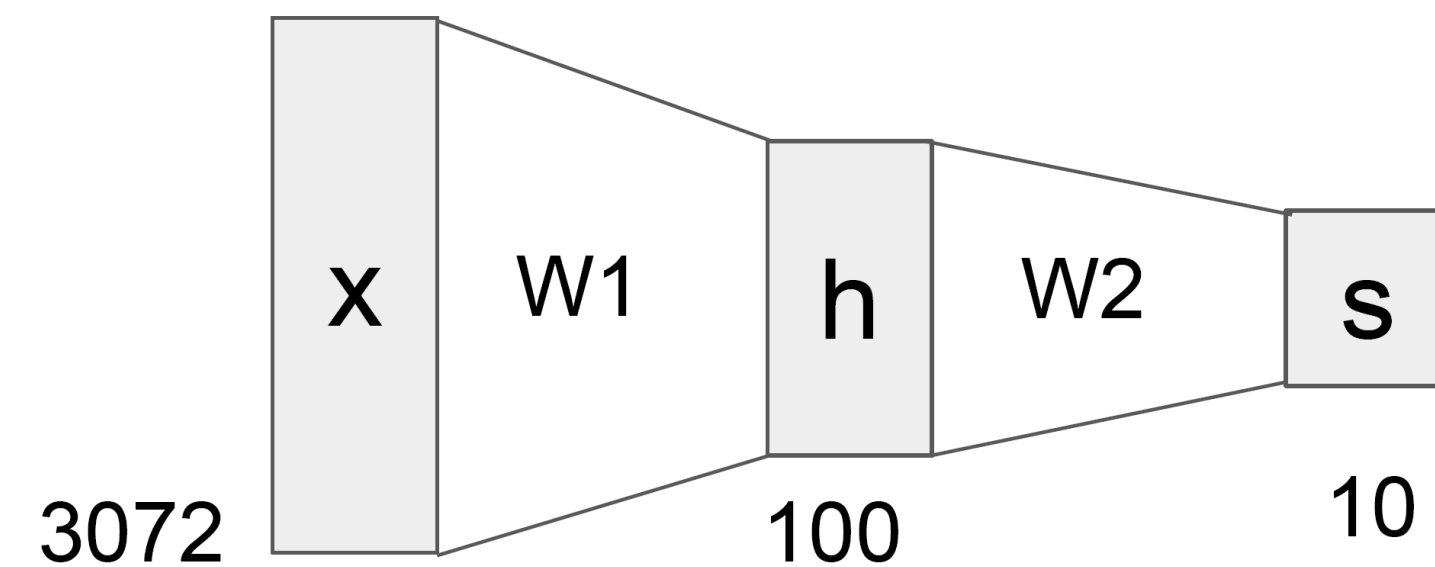
A three-layer neural network could analogously look like  $s = W_3 \max(0, W_2 \max(0, W_1 x))$ , where all of  $W_3, W_2, W_1$  are parameters to be learned. The sizes of the intermediate hidden vectors are hyperparameters of the network and we'll see how we can set them later. Lets now look into how we can interpret these computations from the neuron/network perspective.



**Neural networks: *hierarchical computation***

**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  $f = W_2 \max(0, W_1x)$

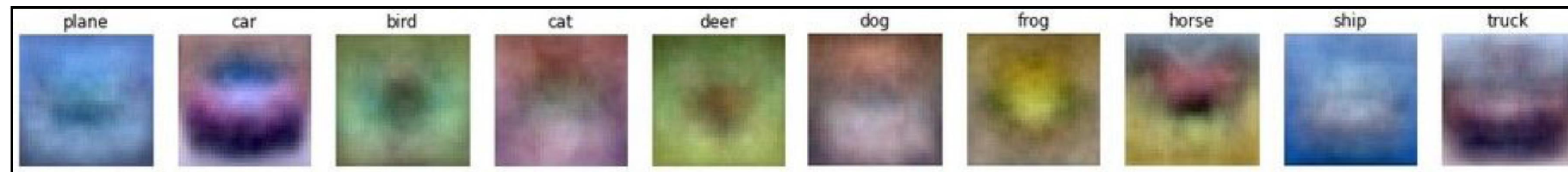
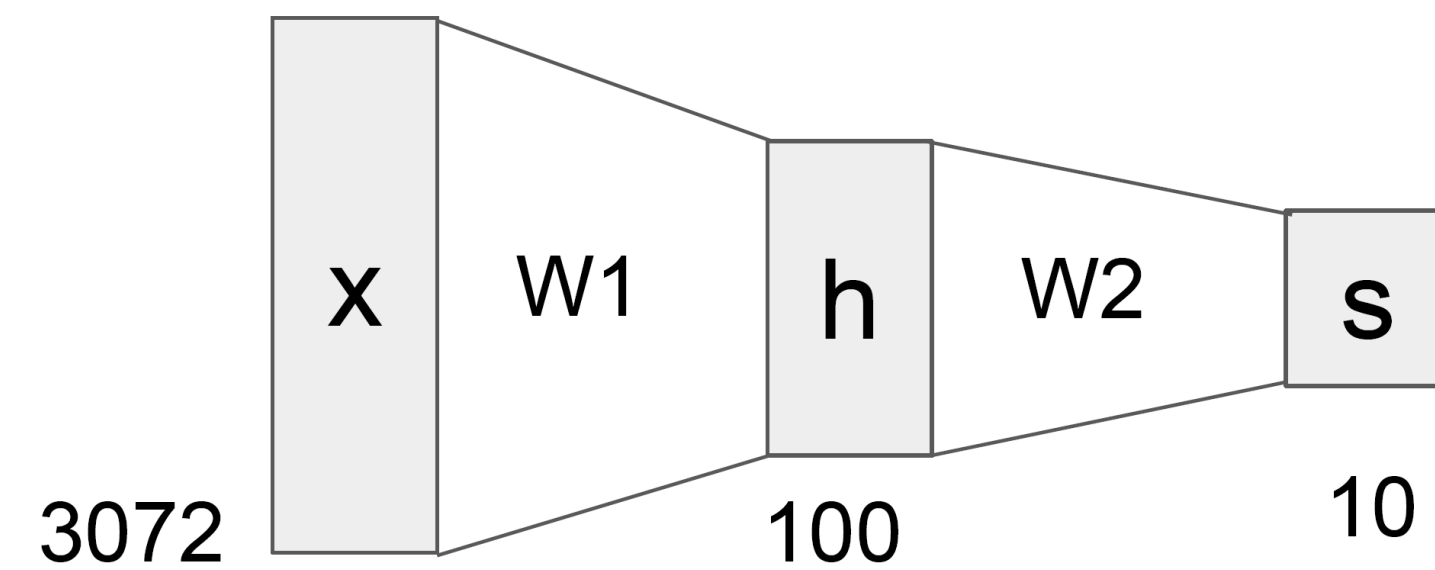


$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

**Neural networks: *learning 100s of templates***

**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  $f = W_2 \max(0, W_1x)$



Learn 100 templates instead of 10.

Share templates between classes

**Neural networks: *learning 100s of templates***

**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  $f = W_2 \max(0, W_1x)$

The function  $\max(0, z)$  is called the **activation function**.

**Q:** What if we try to build a neural network without one?

$$f = W_2W_1x \quad W_3 = W_2W_1 \in \mathbb{R}^{C \times H}, f = W_3x$$

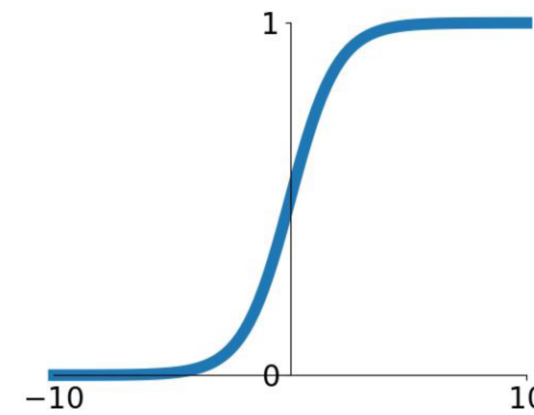
**A:** We end up with a linear classifier again!

## Activation function

### Activation functions

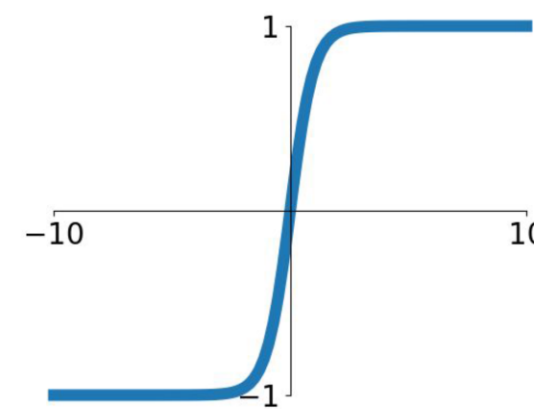
#### Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



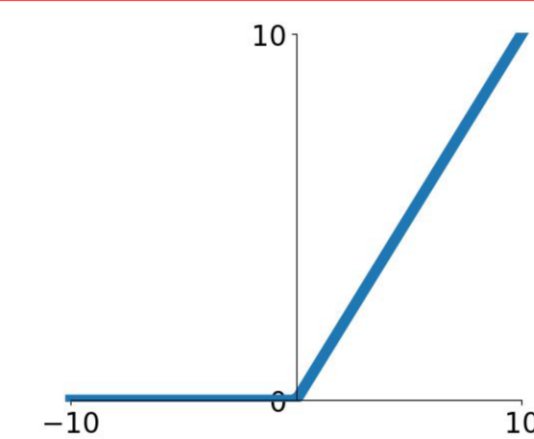
#### tanh

$$\tanh(x)$$



#### ReLU

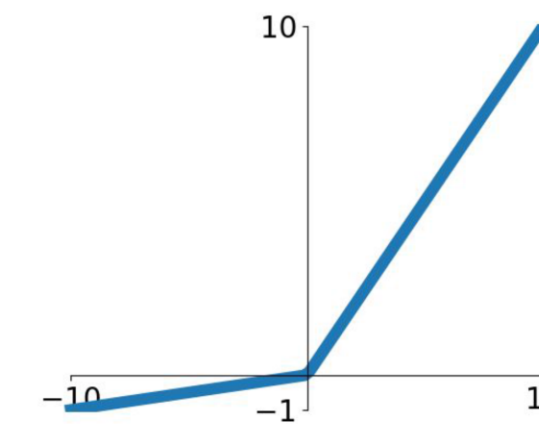
$$\max(0, x)$$



ReLU is a good default choice for most problems

#### Leaky ReLU

$$\max(0.1x, x)$$



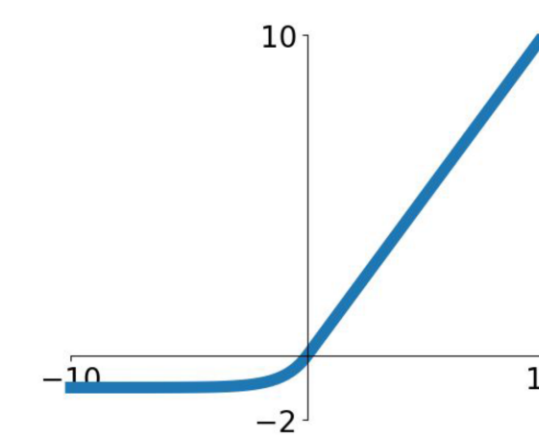
Every activation function (or *non-linearity*) takes a single number and performs a certain fixed mathematical operation on it.

#### Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

#### ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



## Activation function

**Sigmoid.** The sigmoid non-linearity takes a real-valued number and “squashes” it into range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. The sigmoid function has seen frequent use historically since it has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1). In practice, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used. It has two major drawbacks:

- *Sigmoids saturate and kill gradients.* A very undesirable property of the sigmoid neuron is that when the neuron’s activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gate’s output for the whole objective. Therefore, if the local gradient is very small, it will effectively “kill” the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn.
- *Sigmoid outputs are not zero-centered.* This is undesirable since neurons in later layers of processing in a Neural Network (more on this soon) would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive, then the gradient on the weights  $w$  will during backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression  $f$ ). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above.

**Tanh.** The tanh non-linearity is shown on the image above on the right. It squashes a real-valued number to the range  $[-1, 1]$ . Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the *tanh non-linearity is always preferred to the sigmoid nonlinearity*. Also note that the tanh neuron is simply a scaled sigmoid neuron.

## Activation function

**ReLU.** The Rectified Linear Unit has become very popular in the last few years. In other words, the activation is simply thresholded at zero (see image above on the left). There are several pros and cons to using the ReLUs:

- (+) It was found to greatly accelerate (e.g. a factor of 6 in [Krizhevsky et al.](#)) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.
- (+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.
- (-) Unfortunately, ReLU units can be fragile during training and can “die”. For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. For example, you may find that as much as 40% of your network can be “dead” (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue.

## Activation function

**Leaky ReLU.** Leaky ReLUs are one attempt to fix the “dying ReLU” problem. Instead of the function being zero when  $x < 0$ , a leaky ReLU will instead have a small positive slope (of 0.01, or so). Some people report success with this form of activation function, but the results are not always consistent. The slope in the negative region can also be made into a parameter of each neuron, as seen in PReLU neurons, introduced in [Delving Deep into Rectifiers](#), by Kaiming He et al., 2015. However, the consistency of the benefit across tasks is presently unclear.

**Maxout.** Other types of units have been proposed that do not have the functional form  $f(wTx+b)$  where a non-linearity is applied on the dot product between the weights and the data. One relatively popular choice is the Maxout neuron (introduced recently by [Goodfellow et al.](#)) that generalizes the ReLU and its leaky version. The Maxout neuron computes the function  $\max(wT_1x+b_1, wT_2x+b_2)$ . Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have  $w_1, b_1=0$ ). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, unlike the ReLU neurons it doubles the number of parameters for every single neuron, leading to a high total number of parameters.

**TLDR:** “*What neuron type should I use?*” Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of “dead” units in a network. If this concerns you, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout.

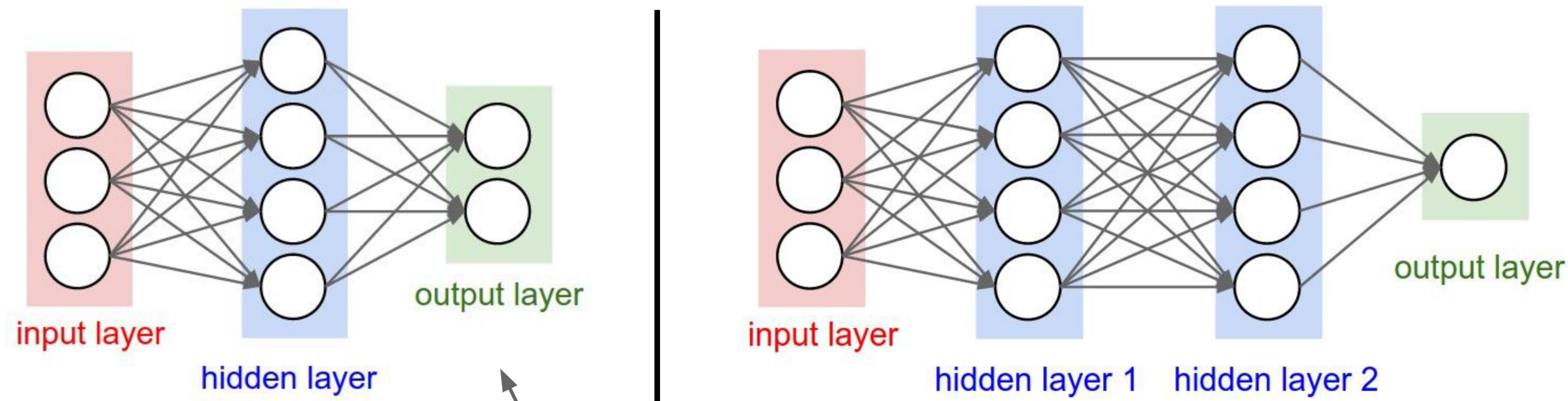
## Neural Networks: *Architectures*

### Neural Networks as neurons in graphs.

Neural Networks are modeled as collections of neurons that are connected in an acyclic graph. In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. Instead of an amorphous blobs of connected neurons, Neural Network models are often organized into distinct layers of neurons. For regular neural networks, the most common layer type is the **fully-connected layer** in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections.



## Neural Networks: Architectures



“2-layer Neural Net”, or  
“1-hidden-layer Neural Net”

“3-layer Neural Net”, or  
“2-hidden-layer Neural Net”

**“Fully-connected” layers**

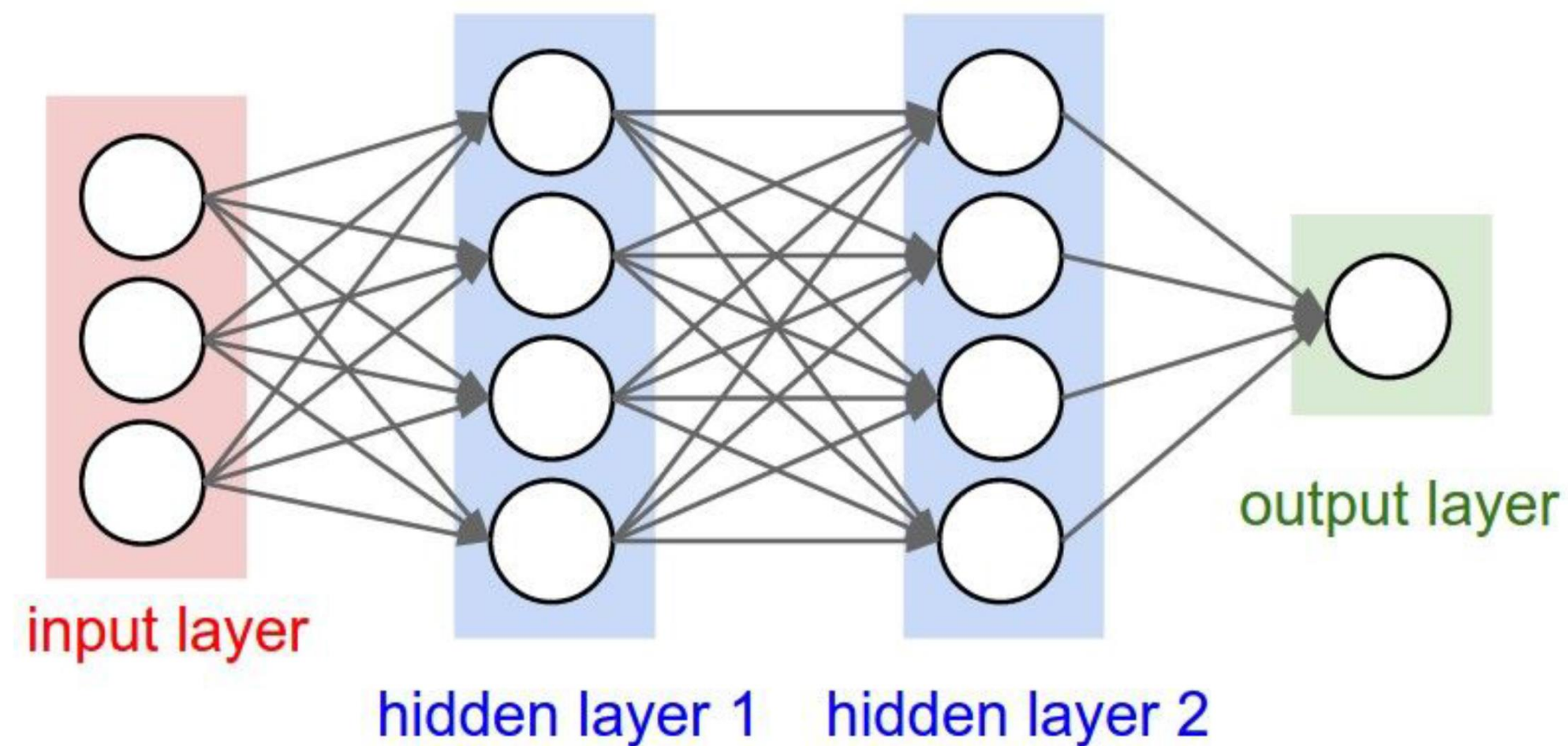
**Left:** A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs. **Right:** A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

## Neural Networks: *Architectures*

**Naming conventions.** Notice that when we say N-layer neural network, we do not count the input layer. Therefore, a single-layer neural network describes a network with no hidden layers (input directly mapped to output). In that sense, you can sometimes hear people say that logistic regression or SVMs are simply a special case of single-layer Neural Networks. You may also hear these networks interchangeably referred to as “*Artificial Neural Networks*” (ANN) or “*Multi-Layer Perceptrons*” (MLP).

**Output layer.** Unlike all layers in a Neural Network, the output layer neurons most commonly do not have an activation function (or you can think of them as having a linear identity activation function). This is because the last output layer is usually taken to represent the class scores (e.g. in classification), which are arbitrary real-valued numbers, or some kind of real-valued target (e.g. in regression)

## Neural Networks: *Example feed-forward computation*



One of the primary reasons that Neural Networks are organized into layers is that this structure makes it very simple and efficient to evaluate Neural Networks using matrix vector operations. Working with the example three-layer neural network in the diagram on the left, the input would be a  $[3 \times 1]$  vector. All connection strengths for a layer can be stored in a single matrix. For example, the first hidden layer's weights  $W1$  would be of size  $[4 \times 3]$ , and the biases for all units would be in the vector  $b1$ , of size  $[4 \times 1]$ . Here, every single neuron has its weights in a row of  $W1$ , so the matrix vector multiplication  $np.dot(W1, x)$  evaluates the activations of all neurons in that layer. Similarly,  $W2$  would be a  $[4 \times 4]$  matrix that stores the connections of the second hidden layer, and  $W3$  a  $[1 \times 4]$  matrix for the last (output) layer.

$W1, W2, W3, b1, b2, b3$  are the learnable parameters of the network.

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

## Neural Networks: *Full implementation of training a 2-layer Neural Network needs ~20 lines*

```

1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14 grad_y_pred = 2.0 * (y_pred - y)
15 grad_w2 = h.T.dot(grad_y_pred)
16 grad_h = grad_y_pred.dot(w2.T)
17 grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19 w1 -= 1e-4 * grad_w1
20 w2 -= 1e-4 * grad_w2

```

Define the network

Forward pass

Calculate the analytical gradients

Gradient descent

## Neural Networks: *Setting number of layers and their sizes*

How do we decide on what architecture to use when faced with a practical problem?

Should we use no hidden layers? One hidden layer? Two hidden layers?

How large should each layer be?

First, note that as we increase the size and number of layers in a Neural Network, the **capacity** of the network increases.

## Neural Networks: *Setting number of layers and their sizes*

When designing a neural network, several factors influence the choice of architecture, including the complexity of the problem, dataset size, computational resources, and need for interpretability. Here's how to make informed decisions:

### No Hidden Layers

**Use Case:** If the problem is linearly separable, a simple model like logistic regression (for classification) or linear regression (for regression) is sufficient.

**Example:** Predicting house prices with simple linear relationships. **Limitation:** Cannot model complex, non-linear relationships.

**Verdict:** No hidden layers are only suitable for simple problems with clear linear separability.

## Neural Networks: *Setting number of layers and their sizes*

### One Hidden Layer

**Use Case:** A single hidden layer with enough neurons can approximate any continuous function (Universal Approximation Theorem).

**When to Use:** If the problem is non-linear but not very complex (e.g., basic image recognition, simple NLP tasks, tabular data).

**Example:** Predicting loan defaults based on customer data.

**Limitation:** May struggle with high-dimensional data like images or sequential data.

**Verdict:** A single hidden layer can work for many problems, but deeper networks improve generalization in complex tasks.

## Neural Networks: *Setting number of layers and their sizes*

### Two or More Hidden Layers

**Use Case:** Deep networks are useful for capturing hierarchical patterns in data (e.g., edges → shapes → objects in image recognition).

**When to Use:** Image classification (CNNs with many layers), Time-series forecasting (RNNs, LSTMs), Text classification, NLP tasks (Transformers)

**Example:** A Convolutional Neural Network (CNN) for recognizing handwritten digits.

**Limitation:** More layers require more data and longer training times.

**Verdict:** Use 2+ hidden layers when handling complex, high-dimensional, or hierarchical data.



## Neural Networks: *Setting number of layers and their sizes*

### Guidelines for Choosing the Number of Neurons per Layer

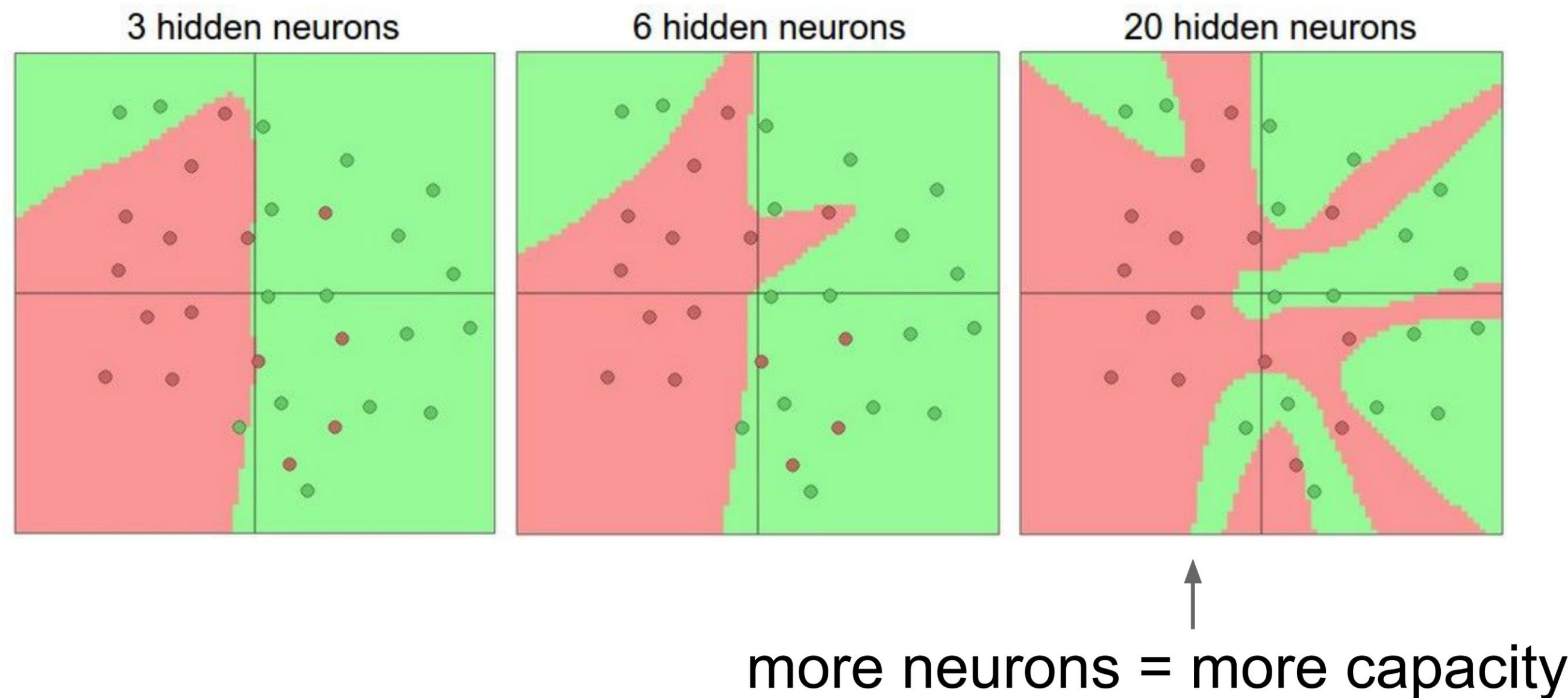
1. **Start with a power of 2:** (e.g., 32, 64, 128, 256).
2. **Keep it proportional to input features:** If input features =  $n$ , then:
  1. Small problem  $\rightarrow 2n$  to  $4n$
  2. Medium problem  $\rightarrow 4n$  to  $8n$
  3. Large problem  $\rightarrow 8n$  to  $16n$
3. **Gradually decrease neuron count per layer:** (e.g.,  $128 \rightarrow 64 \rightarrow 32$ ).
4. **Use cross-validation to fine-tune:** Test different sizes with **hyperparameter tuning**.

### Empirical Rules for Layer Size

Problem Type	Recommended Layers	Neurons Per Layer
Linear Regression	0	-
Simple Classification	1	16 - 64
Tabular Data (MLP)	1-2	64 - 256
Image Recognition (CNNs)	5-50	32 - 512 per conv layer
Text (RNNs, LSTMs)	2-3	128 - 1024
Transformers (BERT, GPT)	12+	768+

## Neural Networks: *Setting number of layers and their sizes*

### Setting the number of layers and their sizes

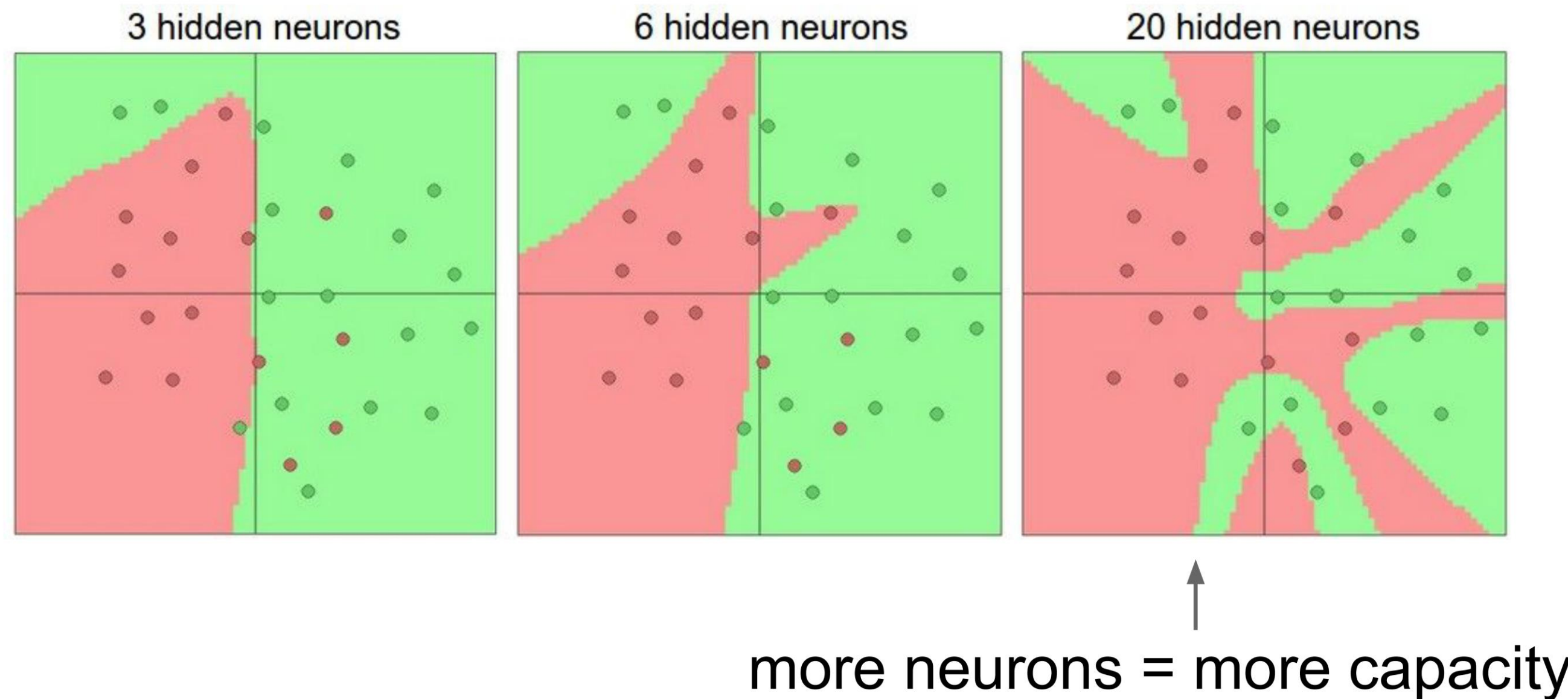


We can see that Neural Networks with more neurons can express more complicated functions. However, this is both a blessing (since we can learn to classify more complicated data) and a curse (since it is easier to overfit the training data).

**Overfitting** occurs when a model with high capacity fits the noise in the data instead of the (assumed) underlying relationship. For example, the model with 20 hidden neurons fits all the training data but at the cost of segmenting the space into many disjoint red and green decision regions. The model with 3 hidden neurons only has the representational power to classify the data in broad strokes. It models the data as two blobs and interprets the few red points inside the green cluster as **outliers** (noise). In practice, this could lead to better **generalization** on the test set.

## Neural Networks: *Setting number of layers and their sizes*

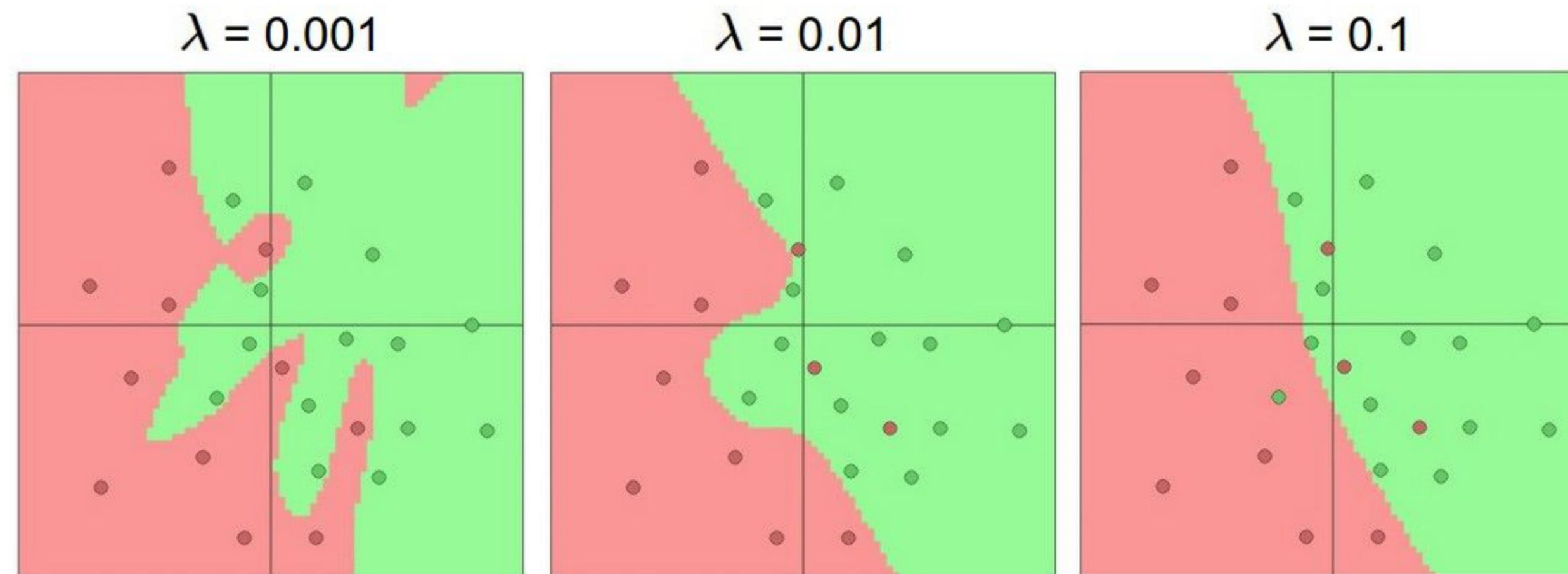
### Setting the number of layers and their sizes



Based on our discussion above, it seems that smaller neural networks can be preferred if the data is not complex enough to prevent overfitting. However, this is incorrect - there are many other preferred ways to prevent overfitting in Neural Networks. In practice, it is always better to use these methods to control overfitting instead of the number of neurons.

## Neural Networks: *Setting number of layers and their sizes*

Do not use size of neural network as a regularizer. Use stronger regularization instead:



$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i) + \lambda R(W)$$

The subtle reason behind this is that smaller networks are harder to train with local methods such as Gradient Descent: It's clear that their loss functions have relatively few local minima, but it turns out that many of these minima are easier to converge to, and that they are bad (i.e. with high loss). Conversely, bigger neural networks contain significantly more local minima, but these minima turn out to be much better in terms of their actual loss.

**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.

<https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html>

## Backpropagation

Problem: How to compute gradients?

$$s = f(x; W_1, W_2) = W_2 \max(0, W_1 x) \quad \text{Nonlinear score function}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM Loss on predictions}$$

$$R(W) = \sum_k W_k^2 \quad \text{Regularization}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W_1) + \lambda R(W_2) \quad \text{Total loss: data loss + regularization}$$

If we can compute  $\frac{\partial L}{\partial W_1}$ ,  $\frac{\partial L}{\partial W_2}$  then we can learn  $W_1$  and  $W_2$

## Backpropagation

(Bad) Idea: Derive  $\nabla_W L$  on paper

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

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

$$= \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1)$$

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

$$= \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2$$

$$\nabla_W L = \nabla_W \left( \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$$

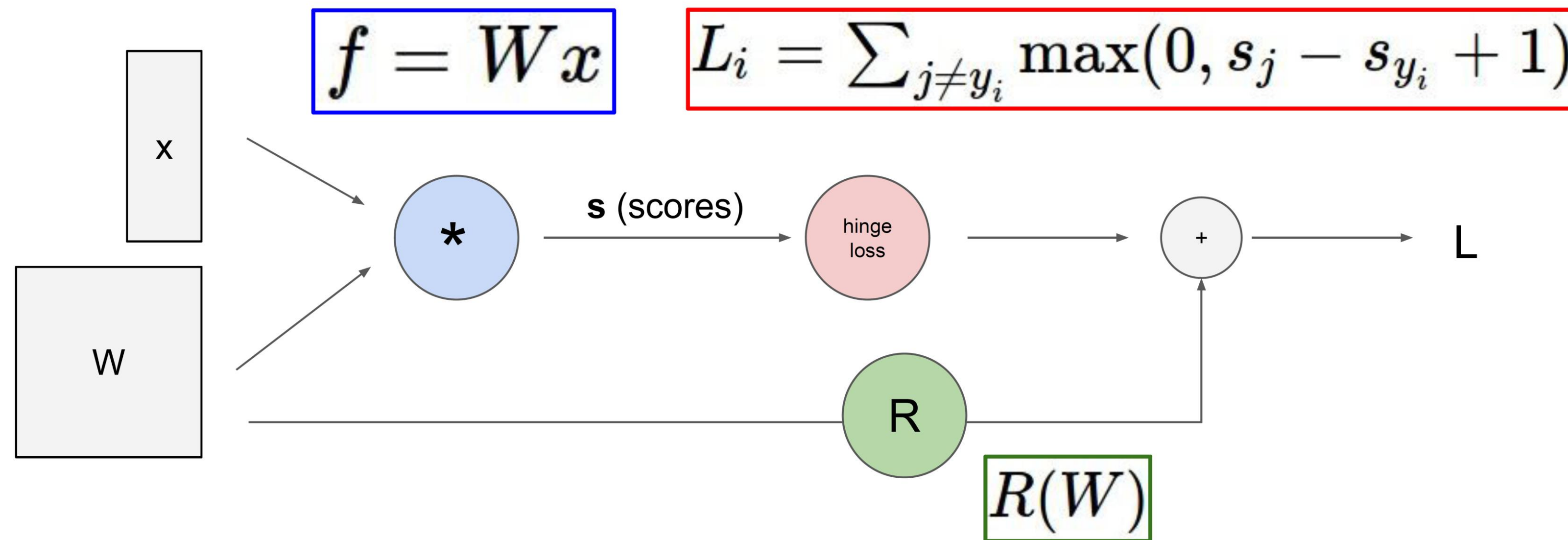
**Problem:** Very tedious: Lots of matrix calculus, need lots of paper

**Problem:** What if we want to change loss? E.g. use softmax instead of SVM? Need to re-derive from scratch =(

**Problem:** Not feasible for very complex models!

**Backpropagation**

Better Idea: Computational graphs + Backpropagation



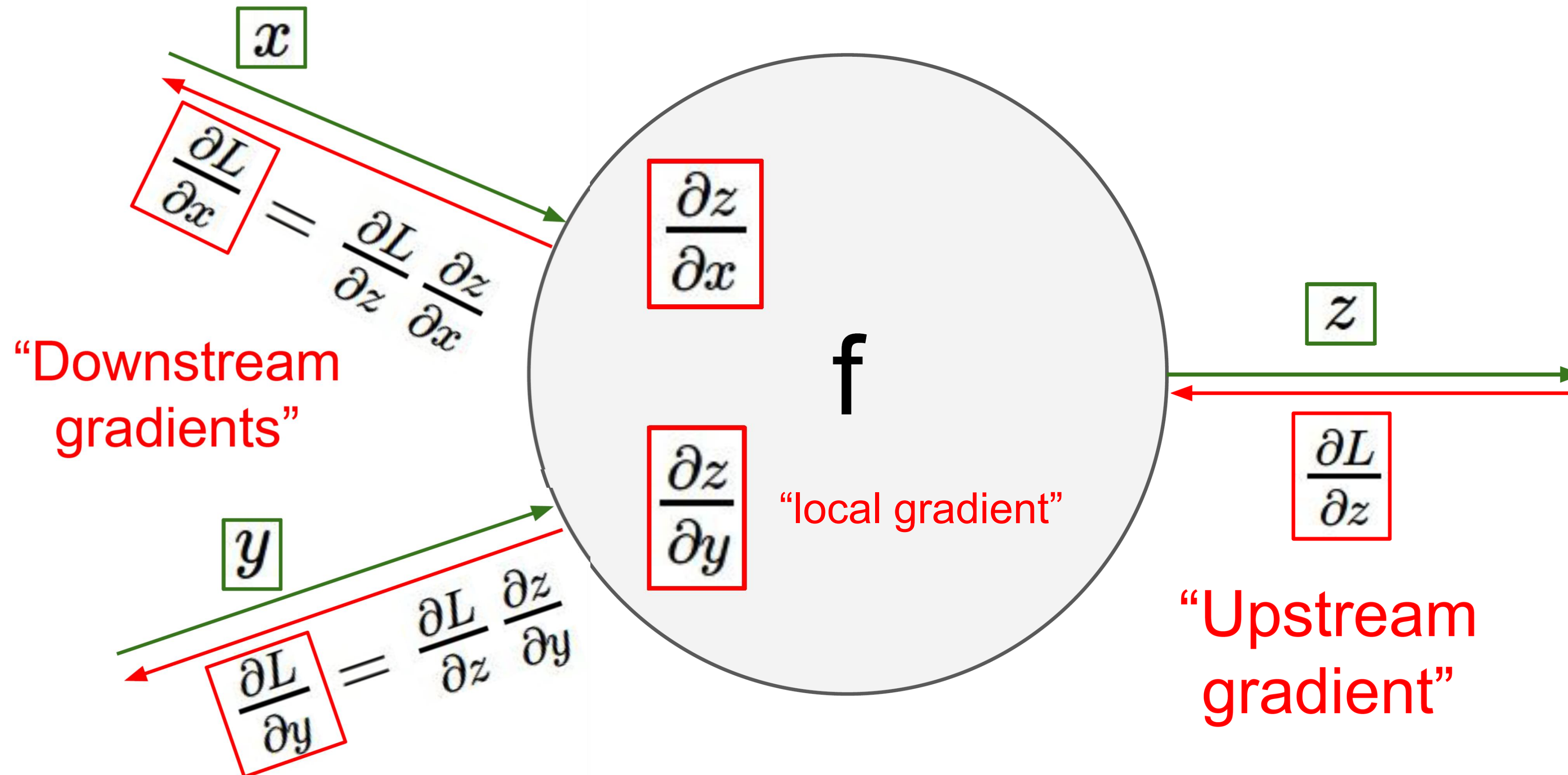
## Backpropagation

**Backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates

- Implementations maintain a graph structure, where the nodes implement the **forward()** / **backward()** API
- **Forward**: compute result of an operation and save any intermediates needed for gradient computation in memory
- **Backward**: apply the chain rule to compute the gradient of the loss function with respect to the inputs



**Backpropagation**



Backpropagation is an algorithm used in artificial neural networks to calculate the gradient of the error function with respect to the weights. It is used during the training process to adjust the weights and reduce the error between the predicted output and the actual output.



## Backpropagation

Backpropagation: a simple example

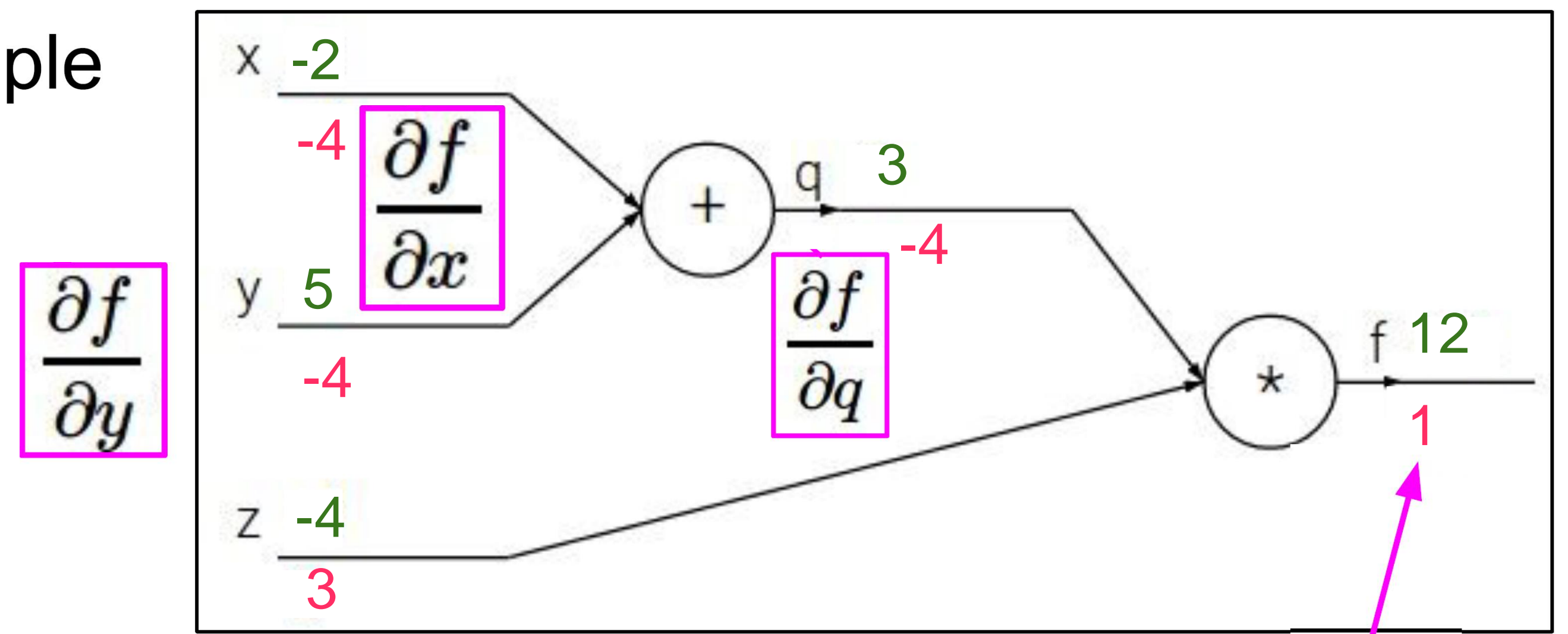
$$f(x, y, z) = (x + y)z$$

e.g.  $x = -2, y = 5, z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want:  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial z}$$

Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

Upstream gradient    Local gradient

$$\frac{\partial f}{\partial f}$$



## Next Lecture

### Convolutional Neural Networks

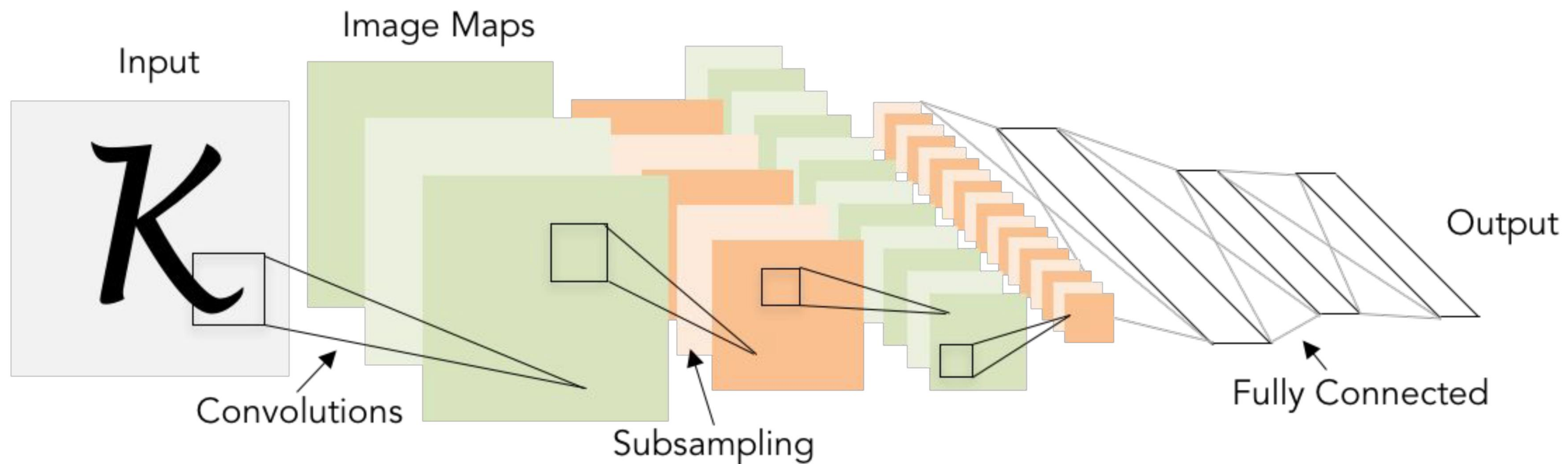


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1



# Thank you!

See you next week

