# Deep Learning (for Computer Vision)

Arjun Jain | 24 March 2017

# Administrative

- 6 Lectures:
  - March 24th, 25th
  - March 31st, April 1st
  - April 7th, 8th (and more if needed)
- 1 Assignment will be given out tomorrow, to be submitted by 14th April
- Torch7 as the deep learning framework (Reason: we will discuss why later)
- Course page: **https://github.com/cs763-dl/2017Spring**

# Aim (by the end of the course)

- Understand the theory behind neural networks and its building blocks
- Be able to code using Torch7 to train models with your own data

# Agenda this Week – Today and Tomorrow

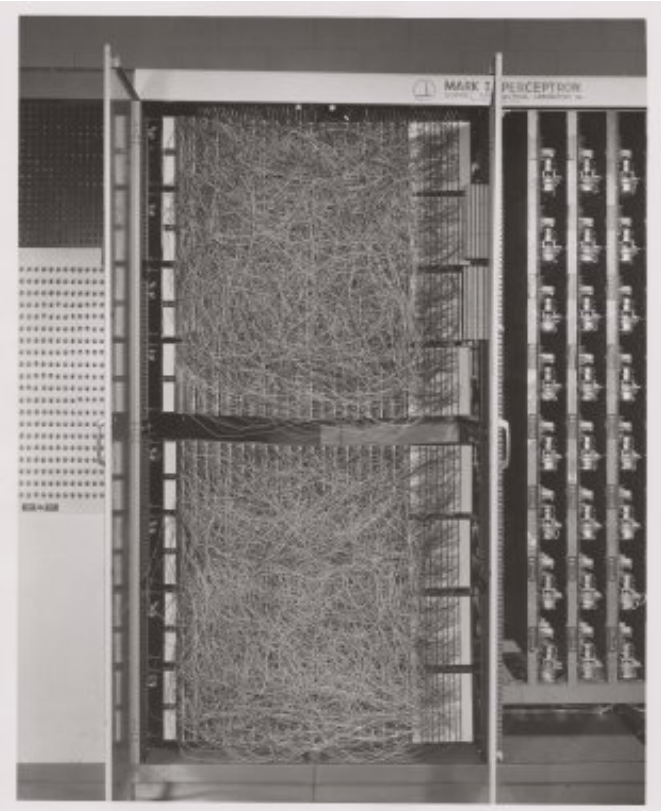
- Introduction, applications and achievements of Deep Learning
- The data driven paradigm: image classification using k-nearest neighbor and linear classification
- Optimization using gradient descent
- Feed forwards networks, training using back propagation, chain rule

# Sources

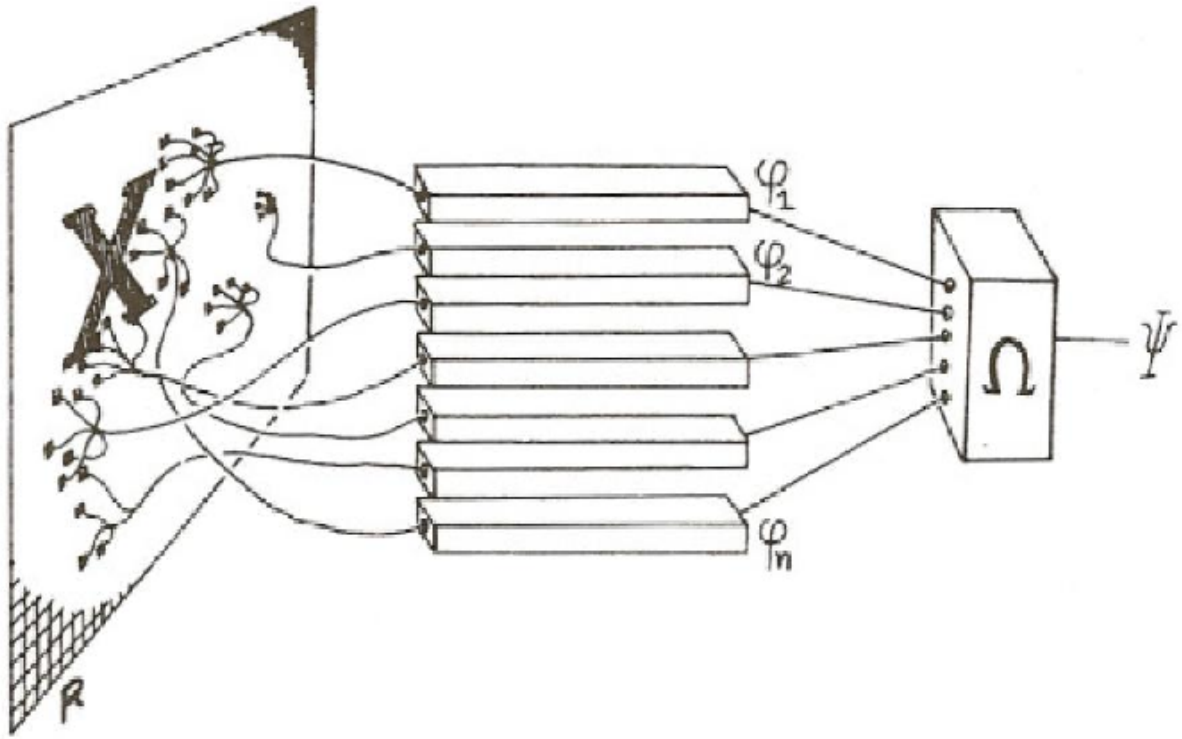
A lot of the material has been shamelessly and gratefully collected from:

- **http://cs231n.stanford.edu/**
- https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training/
- https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html
- https://research.fb.com/learning-to-segment/
- https://research.fb.com/deep-learning-tutorial-at-cvpr-2014/
- http://code.madbits.com/wiki/doku.php?id=tutorial_morestuff
- https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/practicals/practical4.pdf
- http://torch.ch/docs/developer-docs.html
- https://github.com/torch/nn/blob/31d7d2bc86a914e2a9e6b3874c497c60517dc853/doc/module.md
- https://web.stanford.edu/group/pdplab/pdphandbook/handbookch6.html
- http://neuralnetworksanddeeplearning.com/chap2.html

# Brief History – Mark I Perceptron – 1958



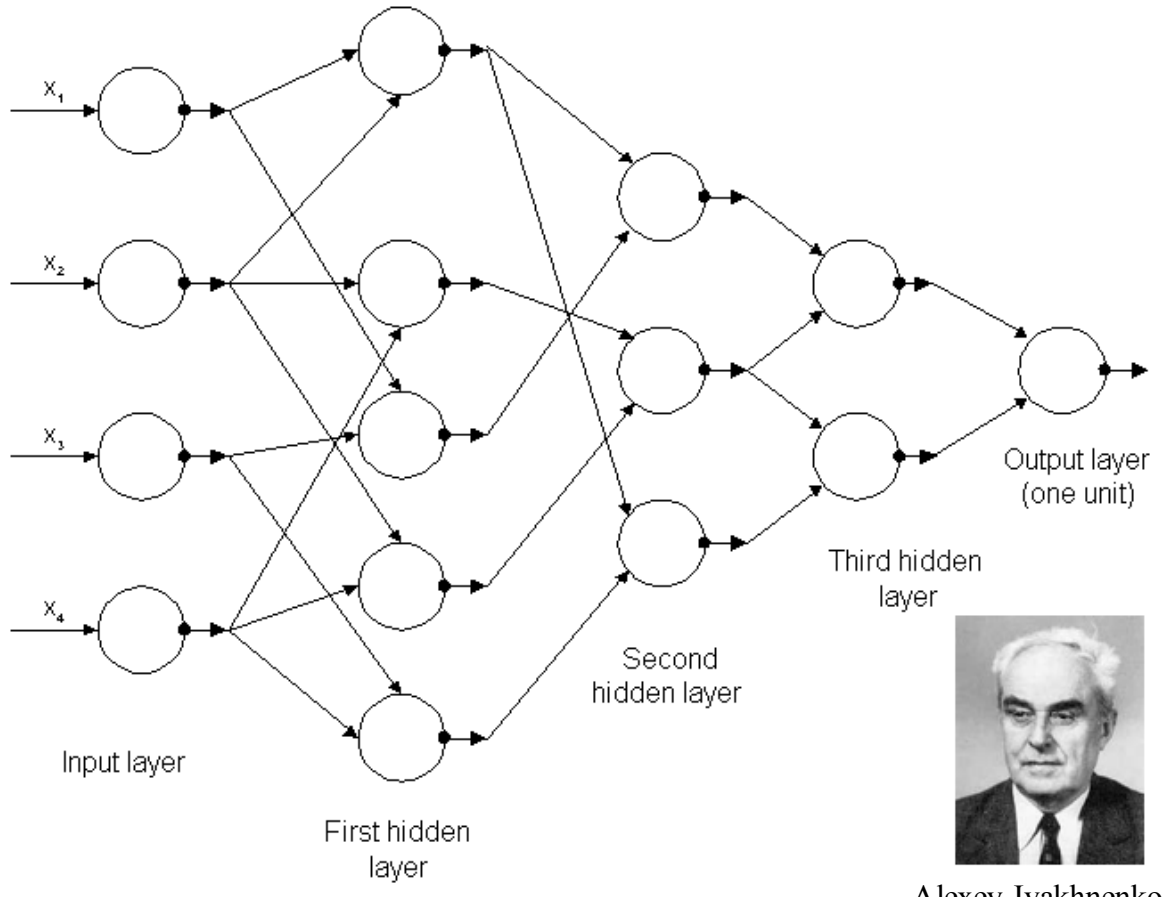

https://en.wikipedia.org/wiki/Perceptron




*Perceptrons* by M. L Minsky and S. Papert, 1969

# Brief History – The First Deep Networks

- Perceptron: single layer 1960s
- Multiple layers of non-linear features - Ivakhnenko and Lapa in 1965
- Thin but deep models with polynomial activation functions
- They did not use backpropagation

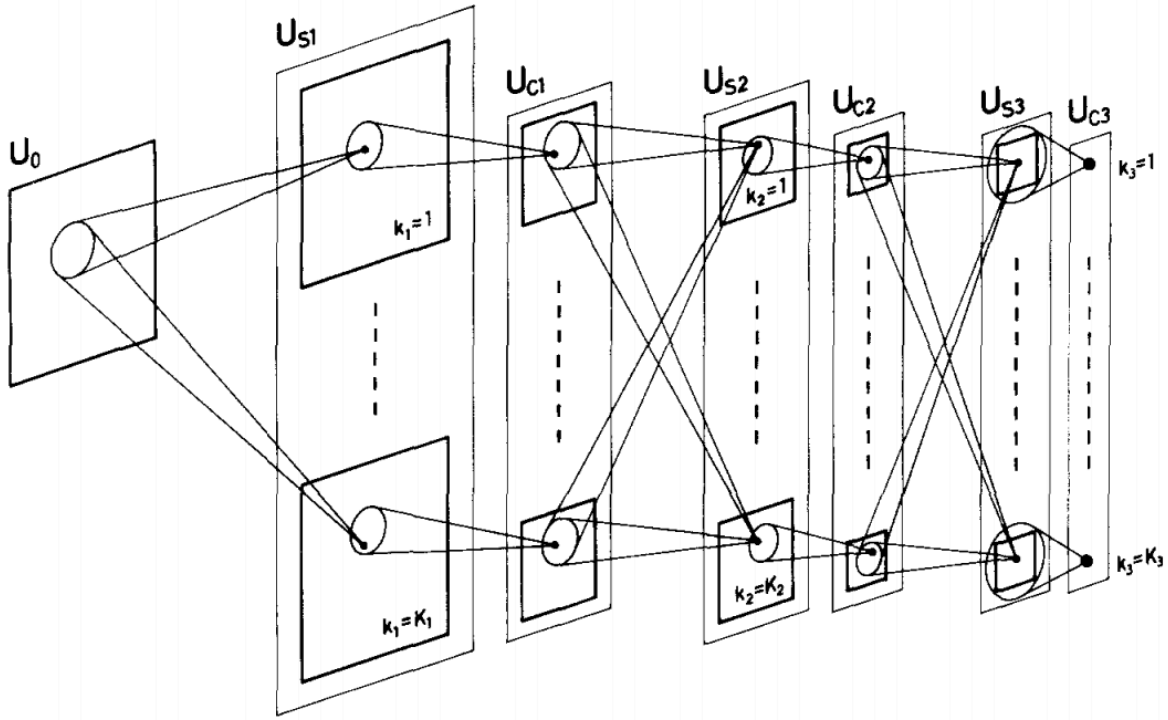


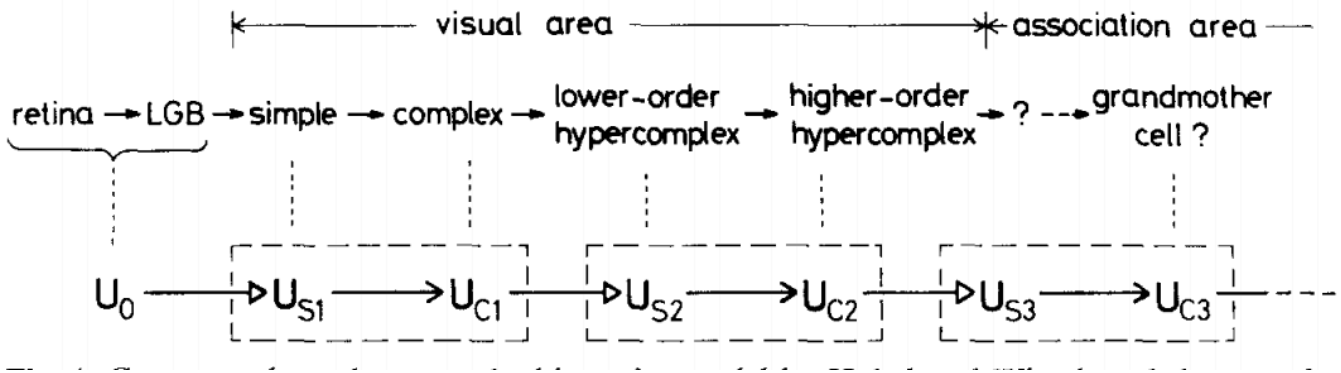


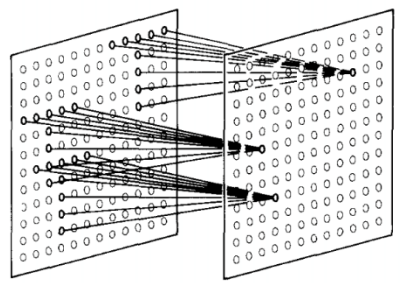

Alexey Ivakhnenko

# Brief History – The First ConvNet

- Neocognitron: multiple convolutional and pooling layers similar to modern networks, but the network was trained by using a reinforcement scheme
- Did not still use backpropagation
- Translational invariant

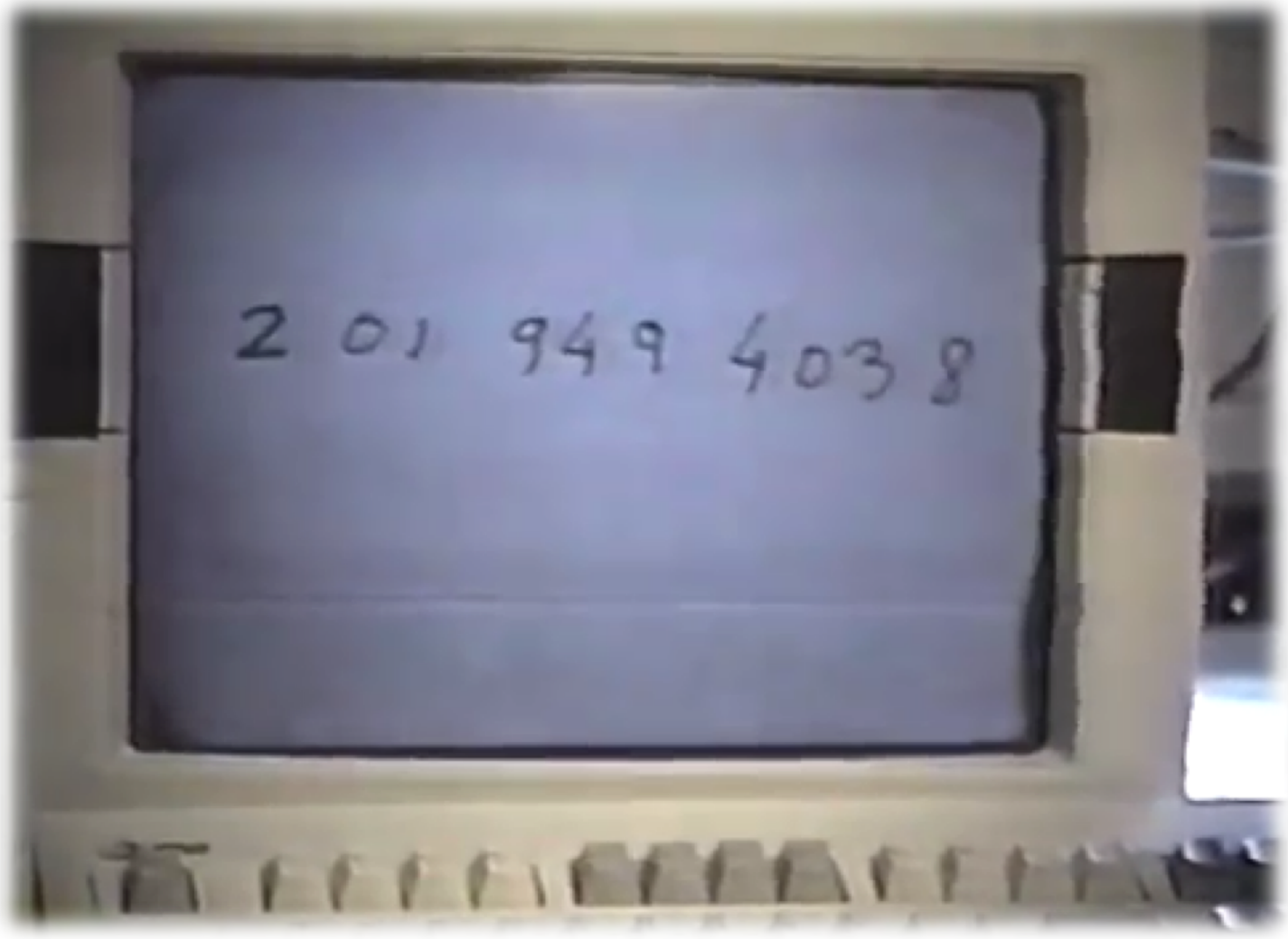


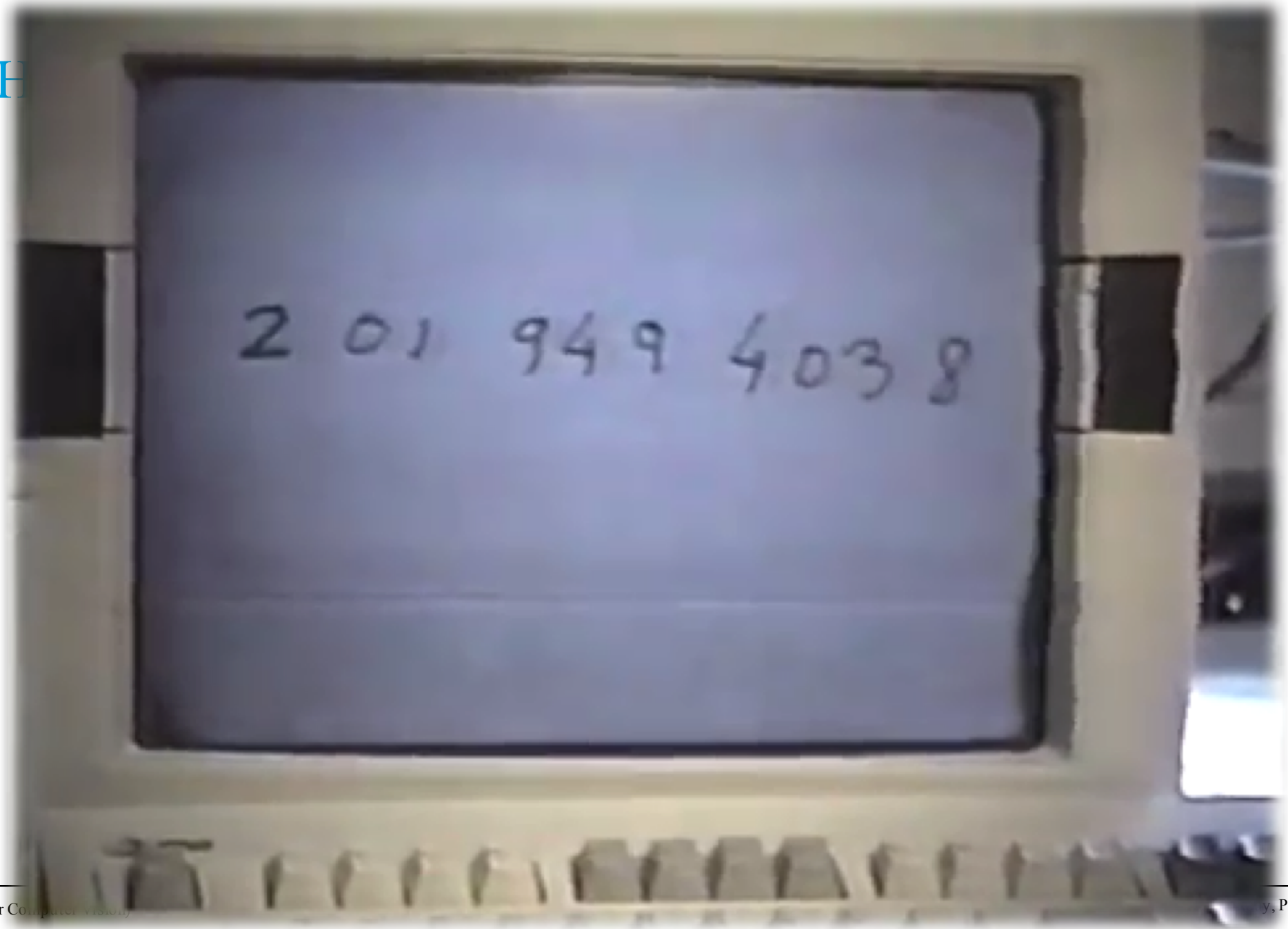



Kunihiko Fukushima

# Brief History – LeNet-5 In Action



# Brief H



# Brief History – AI Winter

- Rapid advances led to a hype of artificial intelligence (similar to the buzz around deep learning today)
- Researchers made promises to solve AI and received lots of funding
- In the 1970s it became clear that those promises could not be kept, funding was cut dramatically
- The field of artificial intelligence dropped to near pseudo-science status
- Research became very difficult (little funding; publications almost never made it through peer review)
- Further advances such as SVMs with nice properties in terms of training, provable error bounds were preferred and took the front seat
- However, a handful of researchers continued further down this path

# Brief History – AI Winter

- Those handful:


**Geoffrey Hinton:** University of Toronto & Google


**Yann LeCun:** New York University & Facebook

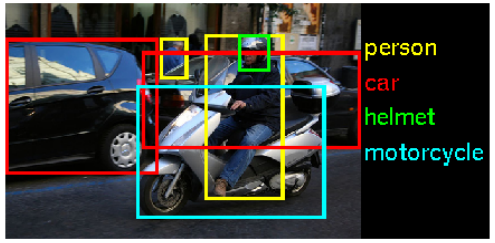

**Yoshua Bengio:** University of Montreal


**Jürgen Schmidhuber:** Swiss AI Lab & NNAISENSE


**Andrew Ng:** Stanford & Baidu

# Brief History – The Tipping Point

- 2012 ILSVRC: ImageNet Large-Scale Visual Recognition Challenge – Annual World Cup of Computer Vision
- More than a million training images and 1000 categories




**ImageNet Classification with Deep Convolutional Neural Networks**

**Alex Krizhevsky**
University of Toronto
kriz@cs.utoronto.ca

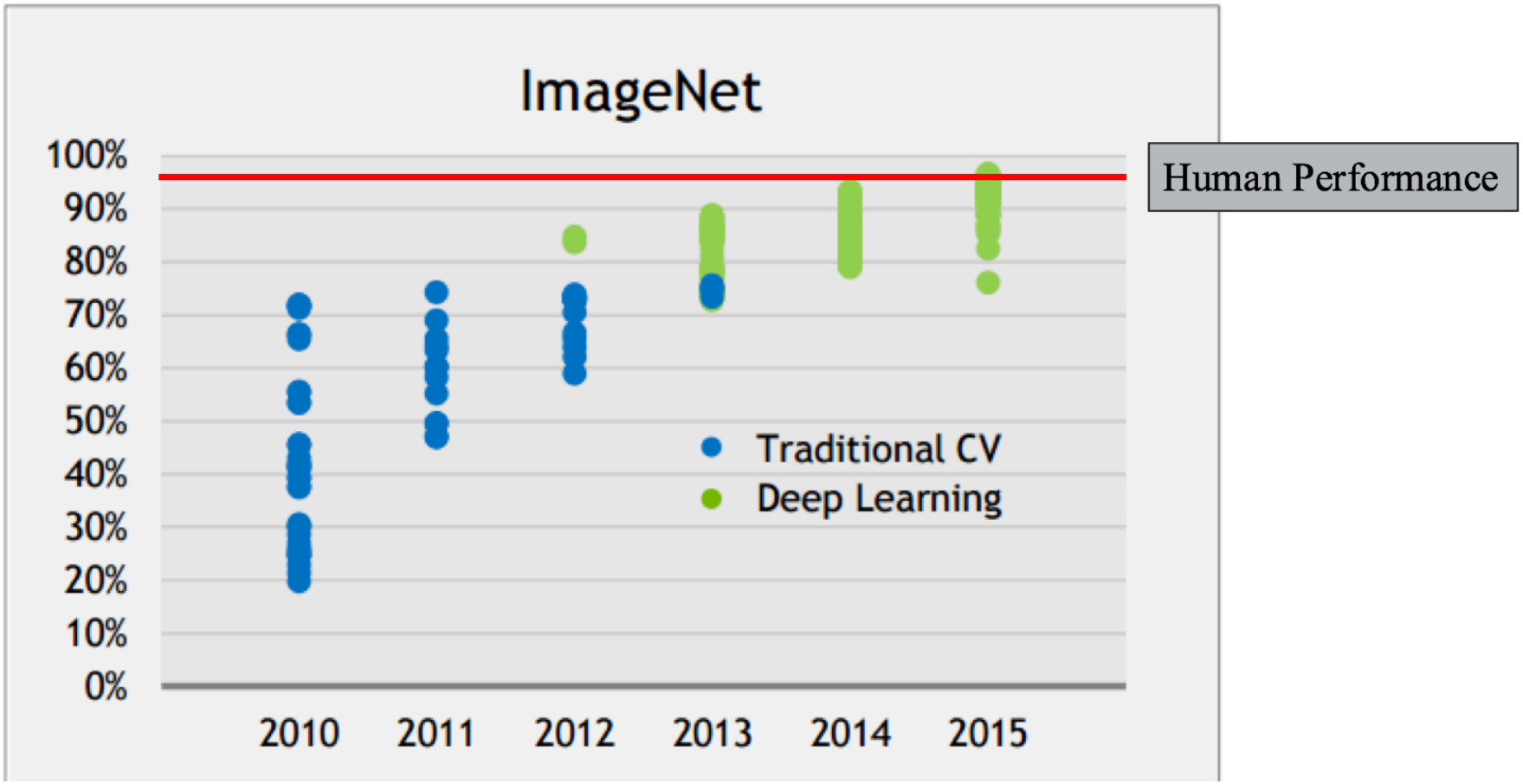
**Ilya Sutskever**
University of Toronto
ilya@cs.utoronto.ca

**Geoffrey E. Hinton**
University of Toronto
hinton@cs.utoronto.ca

# Brief History – The Tipping Point

- Reported 15.4% Top 5 error rate. The next best entry achieved an error of 26.2%
- > 8000 citations
- The coming out party for CNNs in the computer vision community
- Shocked the computer vision community. Trained end-to-end on raw pixels, without using any feature engineering methods
- From here it was apparent that deep learning would take over computer vision and that other methods would not be able to catch up

# Why ConvNets?

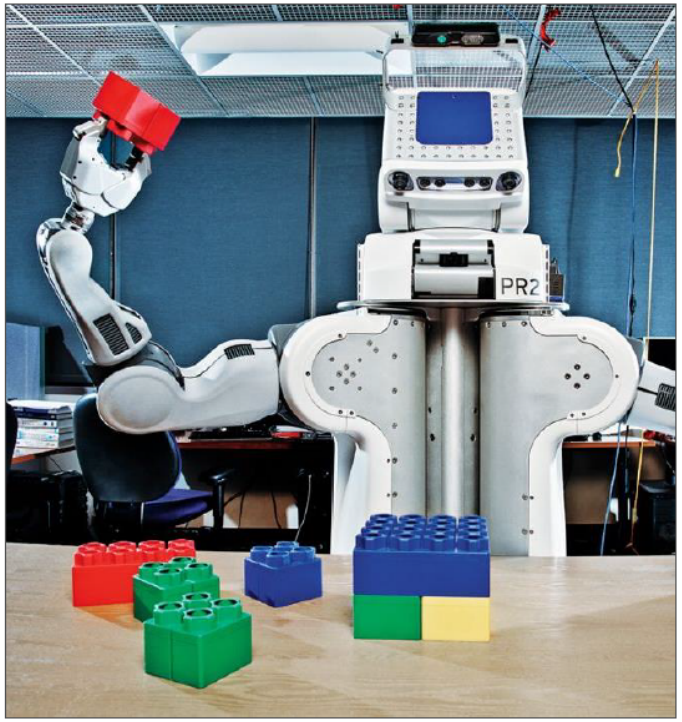


# Brief History – So What Changed (since the 1970s)?

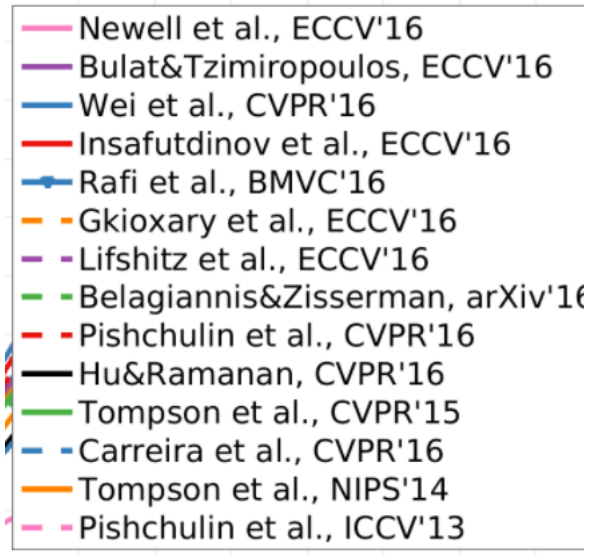
- Three things:
  - Availability of large amounts of labeled data e.g. ImageNet
  - Compute power – A single NVidia TITAN X card churns of 11 TFLOPS with ~3500 cores
  - Algorithms:
    - ReLU - Found to decrease training time
    - Dropout – prevent overfitting to the training data

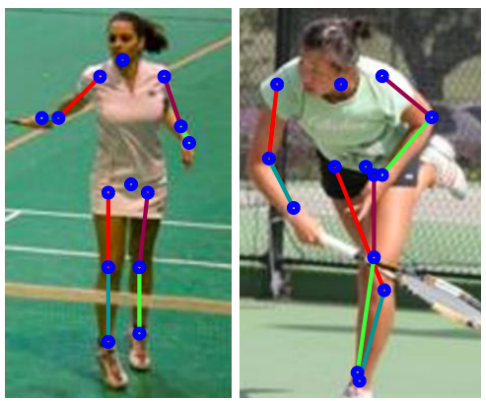
# Deep Learning – Today – One Net To Rule Them All

- Deep Learning == AI
- Solves problems previously unsolvable

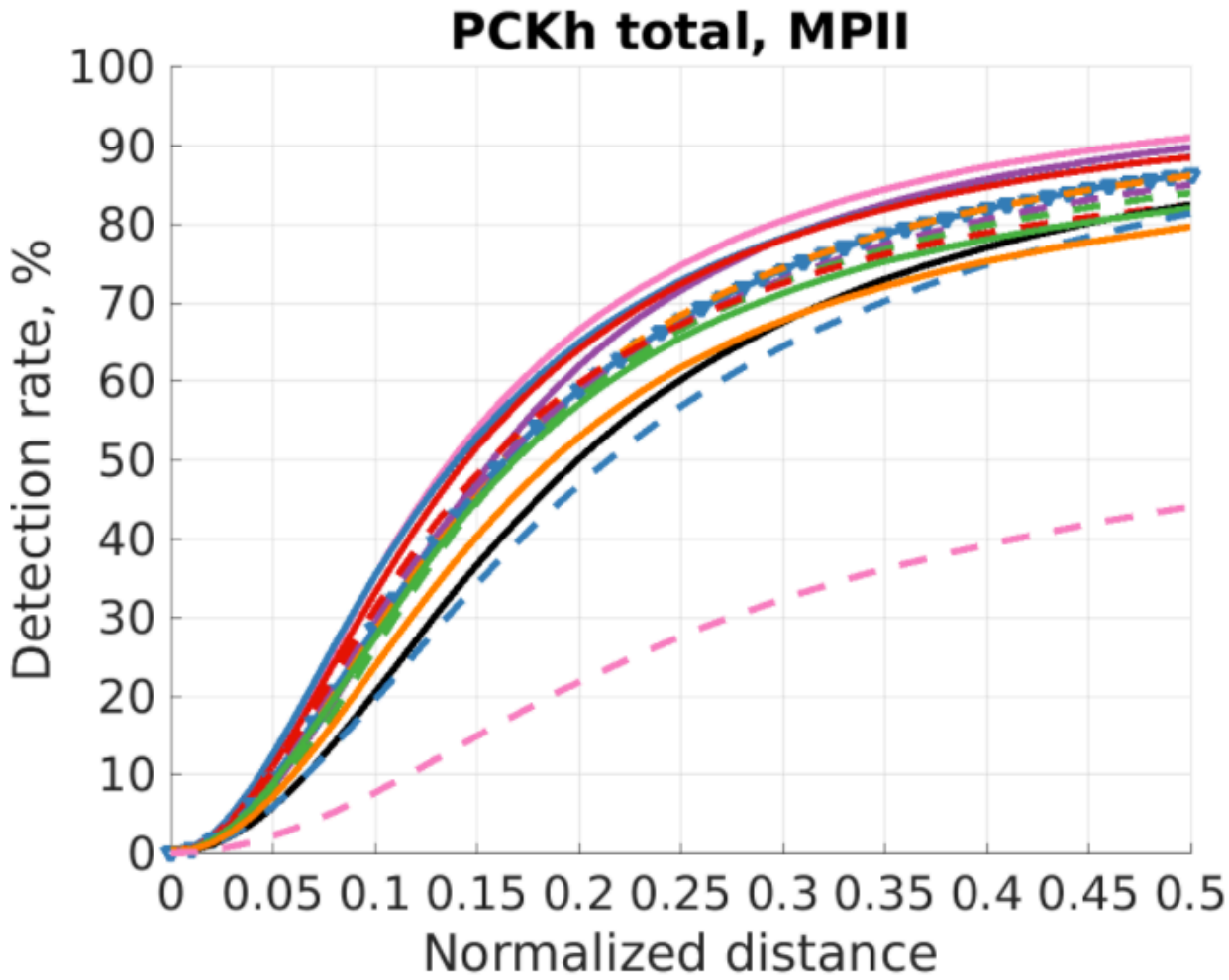




# Deep Learning – Today – Human Pose Estimation







# Deep Learning – Today – Lip Reading (using only images)


BBC One HD
07-Apr-2016 22:51:37
THE GOVERNMENT WILL PAY FOR BOTH SIDES

# Deep Learning – Today – Image Caption Generation

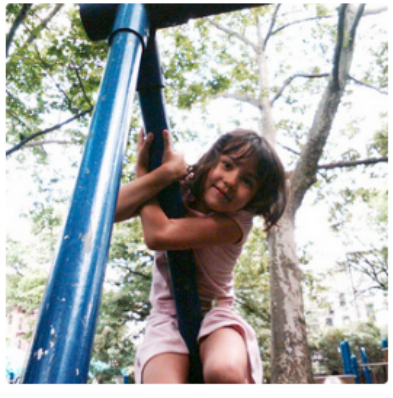
"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

"girl in pink dress is jumping in air."

"black and white dog jumps over bar."

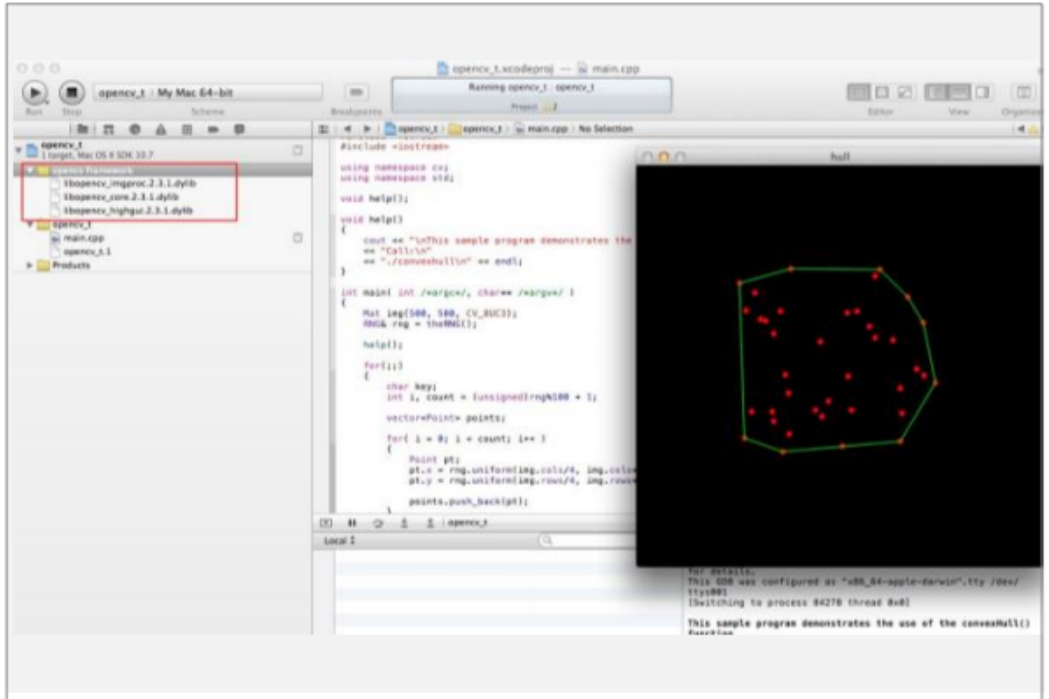


"young girl in pink shirt is swinging on swing."

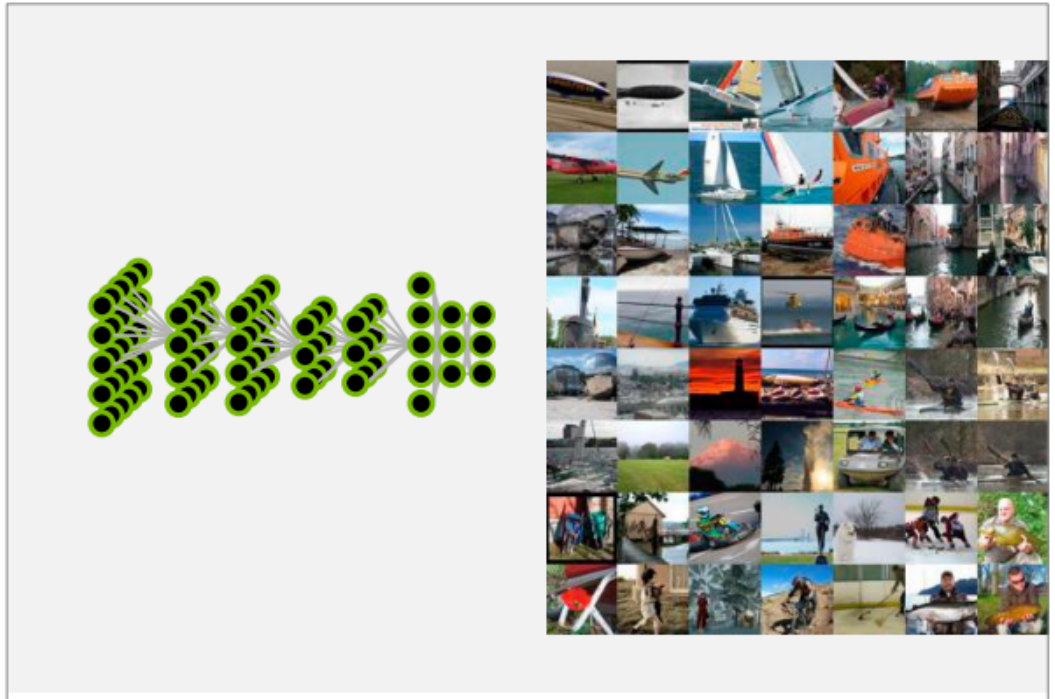
# Deep Learning – Today – Human Computer Interaction


322 mi
Welcome,
inside the 750i

# A New Programming Model – Data Driven Paradigm



Traditional Computer Vision
Experts + Time



Deep Learning Object Detection
DNN + Data + HPC

# Why Data Driven Paradigm?

- Consider Image Classification: a core task in Computer Vision

(assume given set of discrete labels)

{dog, cat, truck, plane, ...}

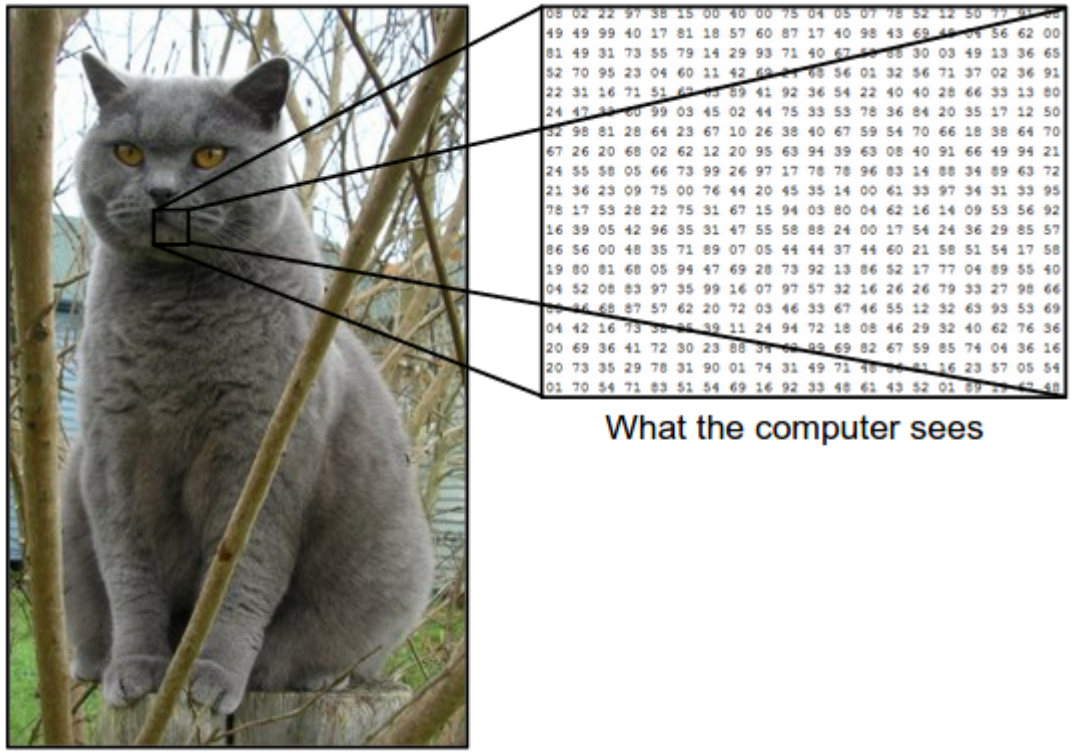
cat

# Why Data Driven Paradigm?

Images are represented as 3D arrays of numbers, with integers between [0, 255].

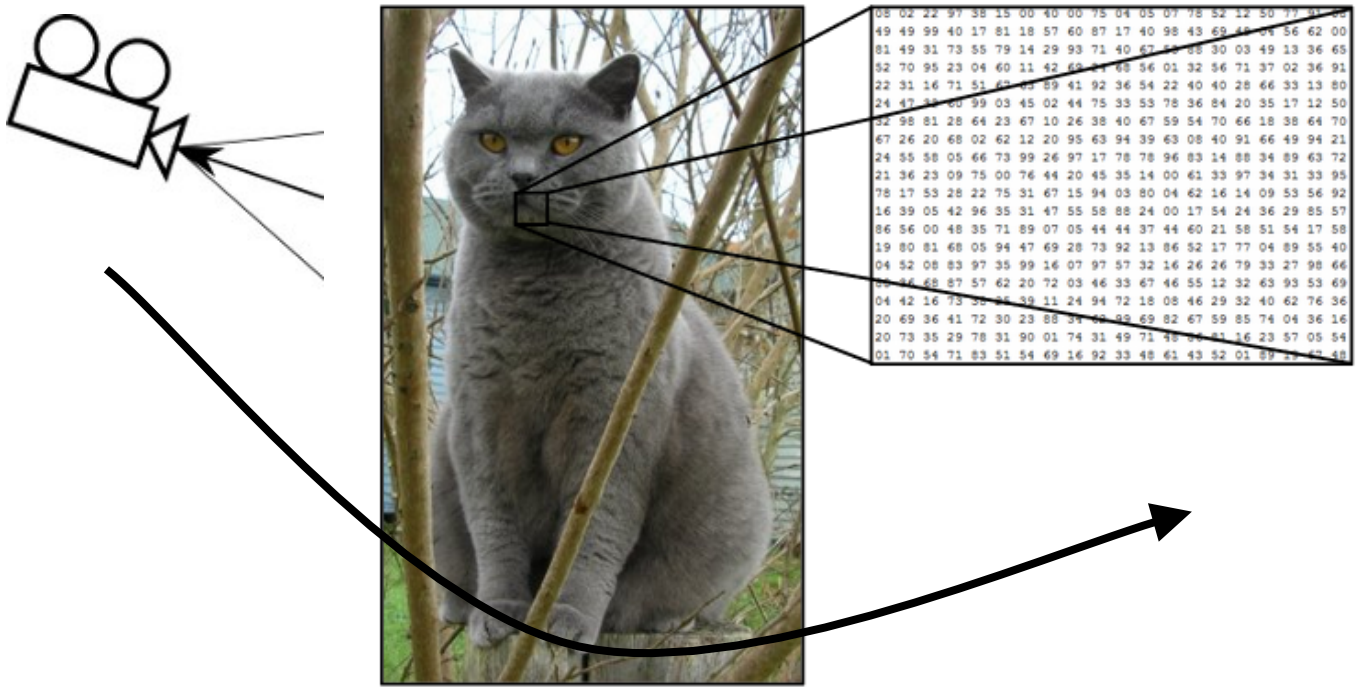
E.g.
300 x 100 x 3

(3 for 3 color channels RGB)

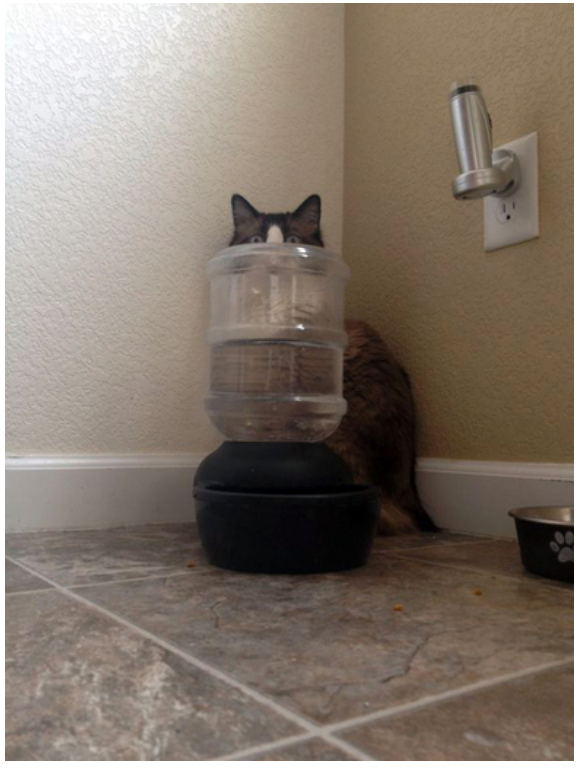


# Why Data Driven Paradigm? – Invariant to Illumination

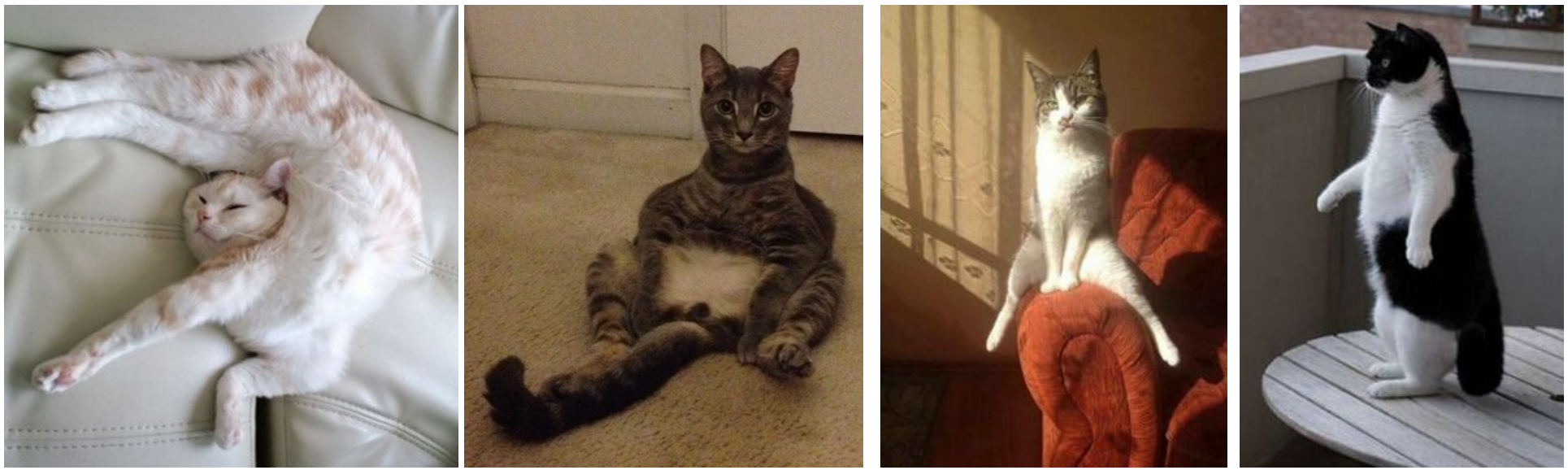
# Why Data Driven Paradigm? – Invariant to Viewpoint



# Why Data Driven Paradigm? – Deal with Occlusion



# Why Data Driven Paradigm? – Invariant to Deformation



# Why Data Driven Paradigm? – Deal with Background Clutter

# Why Data Driven Paradigm? – Deal with Intra-class Variation

# Why Data Driven Paradigm? – No Way To Hand Code It!

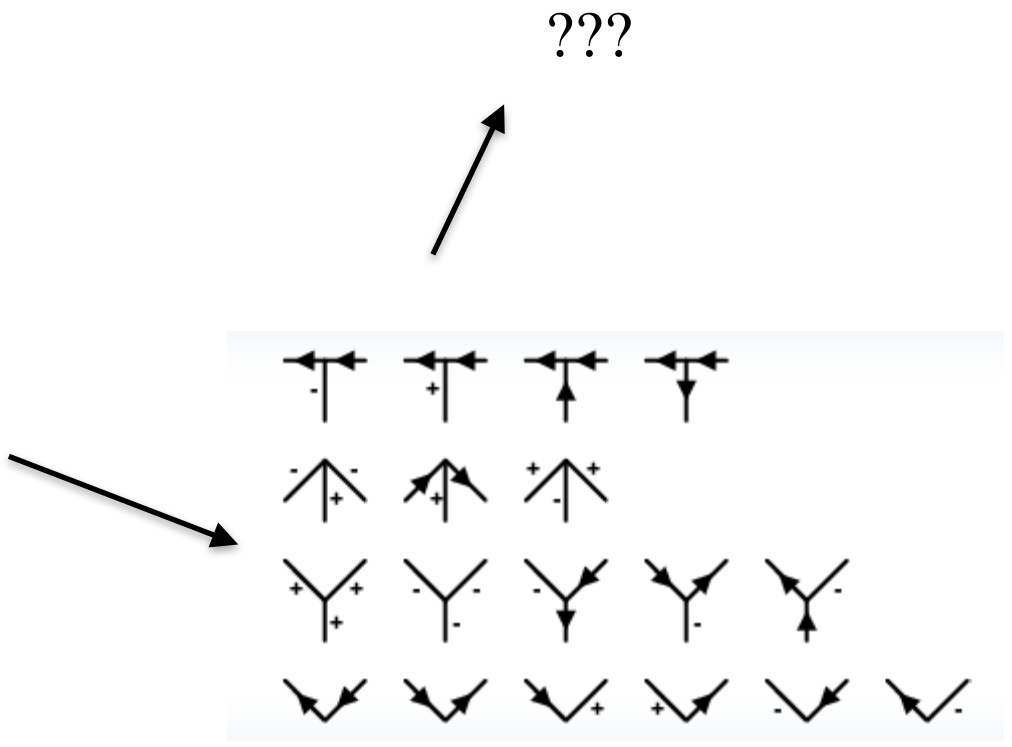
- Image classification:

```
function predict(image)
  -- ????
  return class_label
end
```

- Unlike e.g. sorting a list of numbers
- No obvious way to hard-code the algorithm for recognizing a cat, or other classes

# Why Data Driven Paradigm? – People Have Attempted

- Image classification:



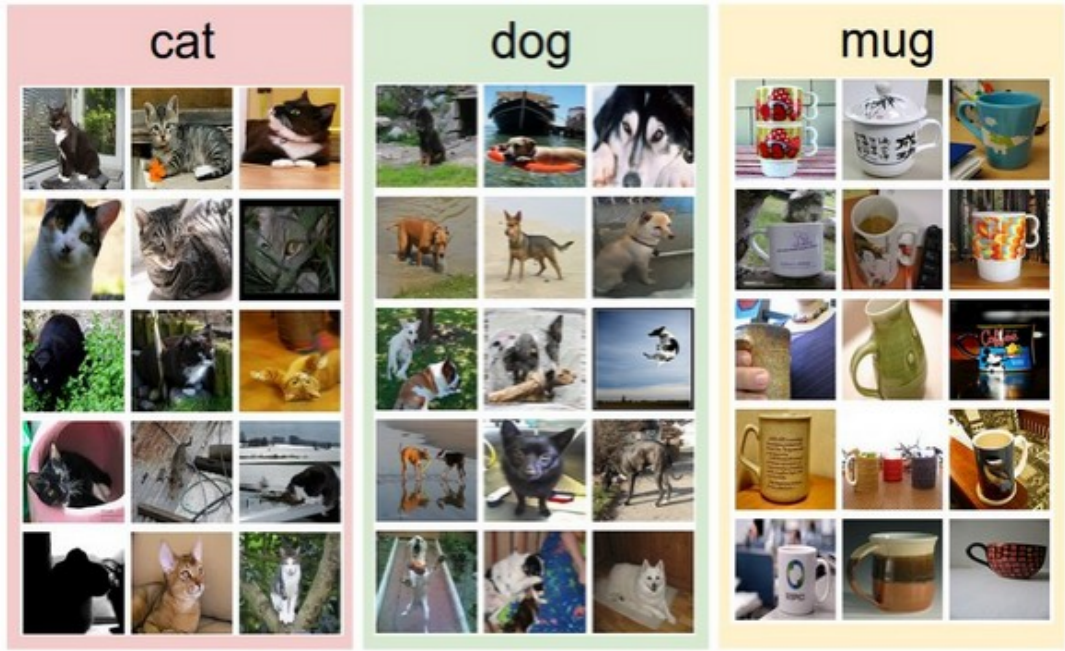
# The Data Driven Paradigm

1. Collect a dataset of images and labels
2. Use Machine Learning to train an image classifier
3. Evaluate the classifier on a withheld set of test images

```
function train(train_images, train_labels)
  -- Build model: images -> labels
  return model
end


function predict(model, test_images)
  -- Predict test_labels using the model
  return test_labels
end
```

Example Training Set



# Classifier 1: Nearest Neighbor Classifier

```
function train(train_images, train_labels)
  -- Build model: images -> labels
  return model
end
```

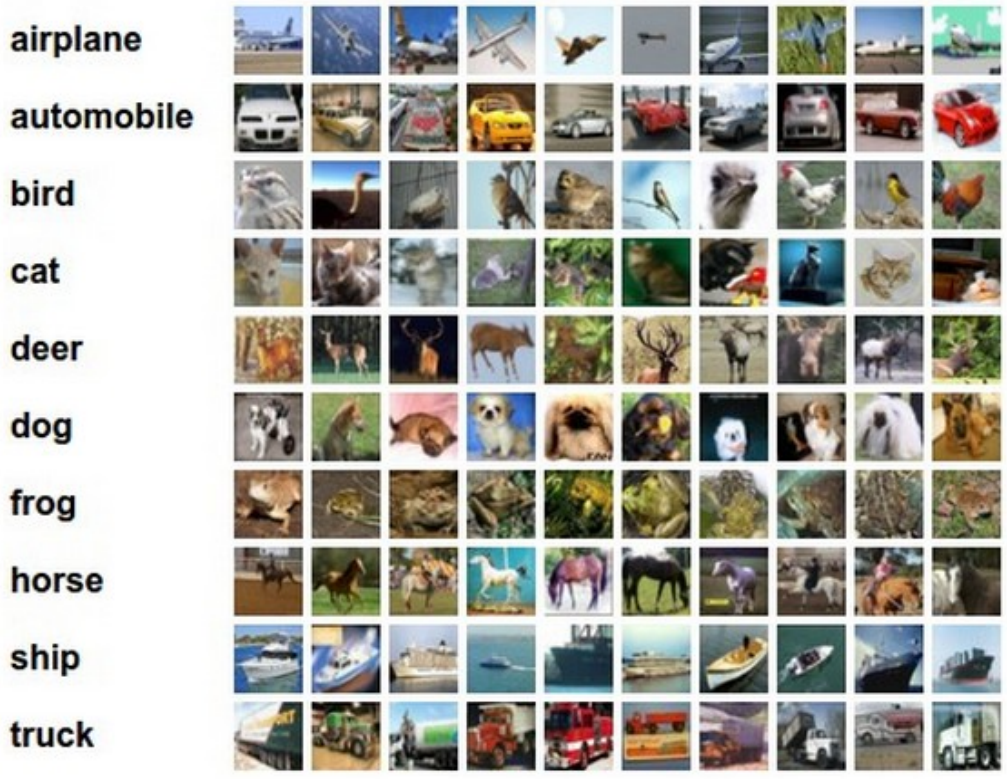
Remember all training images and their labels

```
function predict(model, test_images)
  -- Predict test_labels using the model
  return test_labels
end
```

Predict the label of the most similar training image

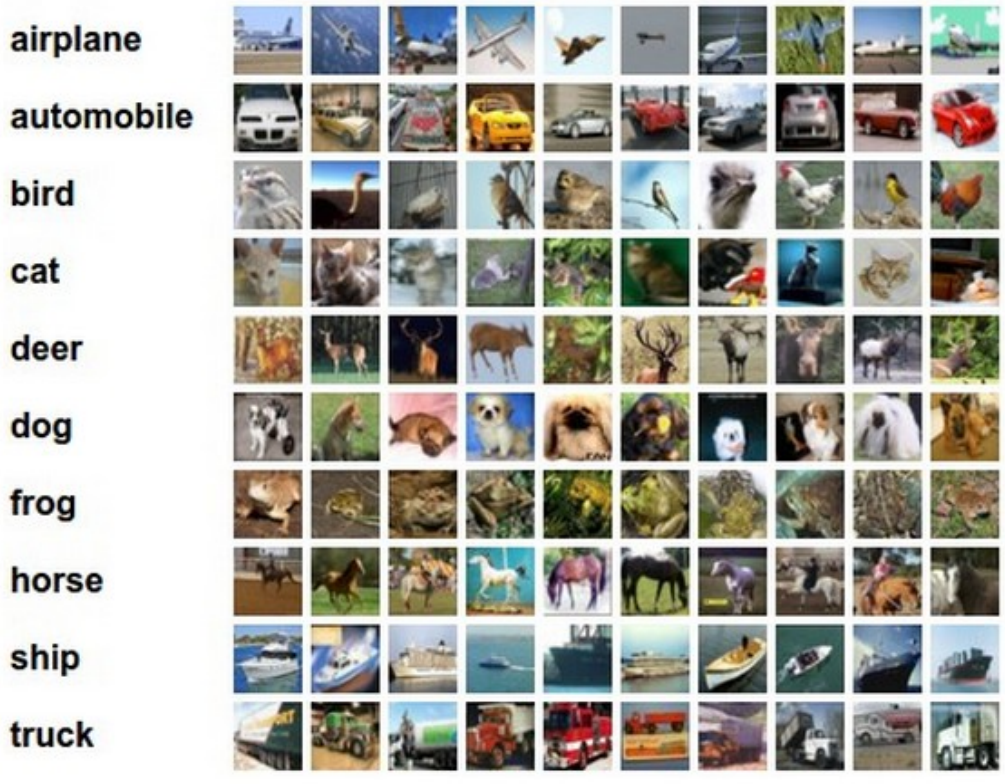
# Example Dataset: CIFAR-10

**10** labels
**50,000** training images
**10,000** test images.

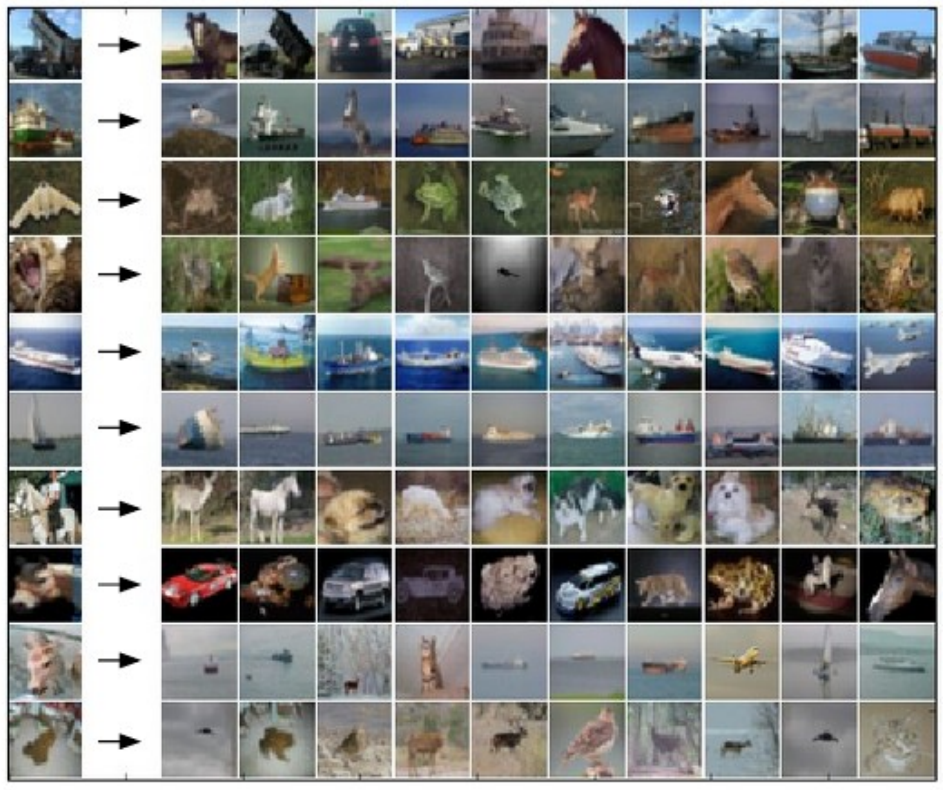


# Example Dataset: CIFAR-10

**10** labels
**50,000** training images
**10,000** test images.

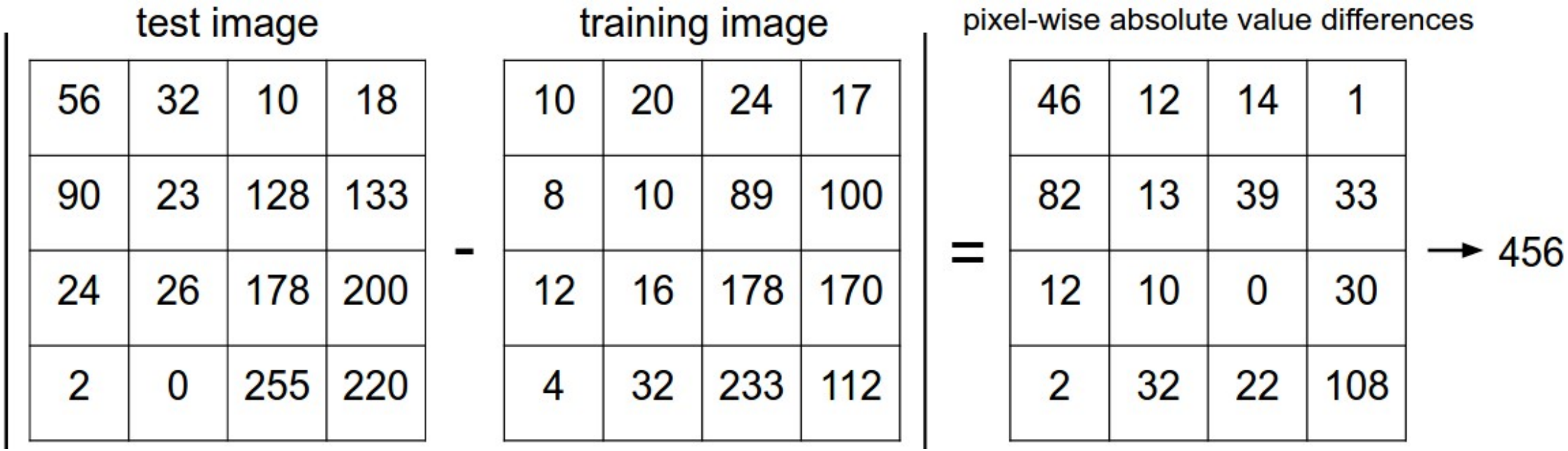



For every test image (first column), examples of nearest neighbors in rows



# How Do We Compare the Images? What is the Distance Metric?

**L1 distance:** $$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



# Nearest Neighbor Classifier

```
function predict(image)
  -- ????
  return class_label
end
```

# Nearest Neighbor Classifier

In [4]:
```
local NN = torch.class("NN")

function NN:__init()
end

function NN:train(X, y)
  -- X is 2D of size N x D = 32x32x3, so each row is an example
  -- Y is 1D of size N
  self.tr_x = X
  self.tr_y = y
end

function NN:predict(x)
  -- x is of size D = 32x32x3 for which we want to predict the label
  -- returns the predicted label for the input x
  local min_idx = nil
  local min_dist = 1e10
  for i=1, self.tr_x:size(1) do
    local dist = (self.tr_x[i] - x):float():abs():sum()
    if (dist < min_dist) then
      min_dist = dist
      min_idx = i
    end
  end
  return self.tr_y[min_idx]
end
```

# Nearest Neighbor Classifier

In [4]:

```
local NN = torch.class("NN")

function NN:__init()
end

function NN:train(X, y)
  -- X is 2D of size N x D = 32x32x3, so each row is an example
  -- Y is 1D of size N
  self.tr_x = X
  self.tr_y = y
end

function NN:predict(x)
  -- x is of size D = 32x32x3 for which we want to predict the label
  -- returns the predicted label for the input x
  local min_idx = nil
  local min_dist = 1e10
  for i=1, self.tr_x:size(1) do
    local dist = (self.tr_x[i] - x):float():abs():sum()
    if (dist < min_dist) then
      min_dist = dist
      min_idx = i
    end
  end
  return self.tr_y[min_idx]
end
```

remember the training data

# Nearest Neighbor Classifier

In [4]:

```
local NN = torch.class("NN")

function NN:__init()
end

function NN:train(X, y)
  -- X is 2D of size N x D = 32x32x3, so each row is an example
  -- Y is 1D of size N
  self.tr_x = X
  self.tr_y = y
end

function NN:predict(x)
  -- x is of size D = 32x32x3 for which we want to predict the label
  -- returns the predicted label for the input x
  local min_idx = nil
  local min_dist = 1e10
  for i=1, self.tr_x:size(1) do
    local dist = (self.tr_x[i] - x):float():abs():sum()
    if (dist < min_dist) then
      min_dist = dist
      min_idx = i
    end
  end
  return self.tr_y[min_idx]
end
```

For the test image:

- find nearest train image with L1 distance
- predict the label of nearest training image

# Nearest Neighbor Classifier

In [4]:
```
local NN = torch.class("NN")

function NN:__init()
end

function NN:train(X, y)
  -- X is 2D of size N x D = 32x32x3, so eac
  -- Y is 1D of size N
  self.tr_x = X
  self.tr_y = y
end

function NN:predict(x)
  -- x is of size D = 32x32x3 for which we w
  -- returns the predicted label for the inp
  local min_idx = nil
  local min_dist = 1e10
  for i=1, self.tr_x:size(1) do
    local dist = (self.tr_x[i] - x):float():a
    if (dist < min_dist) then
      min_dist = dist
      min_idx = i
    end
  end
  return self.tr_y[min_idx]
end
```

In [5]:
```
classifier = NN.new()
classifier:train(tr_x, tr_y)

itorch.image(te_x[100])
print (classifier:predict(te_x[100]))

itorch.image(te_x[110])
print (classifier:predict(te_x[110]))

itorch.image(te_x[120])
print (classifier:predict(te_x[120]))

itorch.image(te_x[130])
print (classifier:predict(te_x[130]))

itorch.image(te_x[140])
print (classifier:predict(te_x[140]))
```

Out[5]: 7

Out[5]: 7

Out[5]: 7

# Nearest Neighbor Classifier

In [4]:
```
local NN = torch.class("NN")

function NN:__init()
end

function NN:train(X, y)
  -- X is 2D of size N x D = 32x32x3, so each row is an example
  -- Y is 1D of size N
  self.tr_x = X
  self.tr_y = y
end

function NN:predict(x)
  -- x is of size D = 32x32x3 for which we want to predict the label
  -- returns the predicted label for the input x
  local min_idx = nil
  local min_dist = 1e10
  for i=1, self.tr_x:size(1) do
    local dist = (self.tr_x[i] - x):float():abs():sum()
    if (dist < min_dist) then
      min_dist = dist
      min_idx = i
    end
  end
  return self.tr_y[min_idx]
end
```

**Q: how does the classification speed depend on the size of the training data?**

# Nearest Neighbor Classifier

In [4]:
```
local NN = torch.class("NN")

function NN:__init()
end

function NN:train(X, y)
  -- X is 2D of size N x D = 32x32x3, so each row is an example
  -- Y is 1D of size N
  self.tr_x = X
  self.tr_y = y
end

function NN:predict(x)
  -- x is of size D = 32x32x3 for which we want to predict the label
  -- returns the predicted label for the input x
  local min_idx = nil
  local min_dist = 1e10
  for i=1, self.tr_x:size(1) do
    local dist = (self.tr_x[i] - x):float():abs():sum()
    if (dist < min_dist) then
      min_dist = dist
      min_idx = i
    end
  end
  return self.tr_y[min_idx]
end
```

**Q: how does the classification speed depend on the size of the training data?**
**linearly :(**

This is **backwards**:
- test time performance is usually much more important in practice.
- CNNs flip this: expensive training, cheap test evaluation

# The Choice of Distance is a **Hyperparameter**

Common choices:

L1 distance

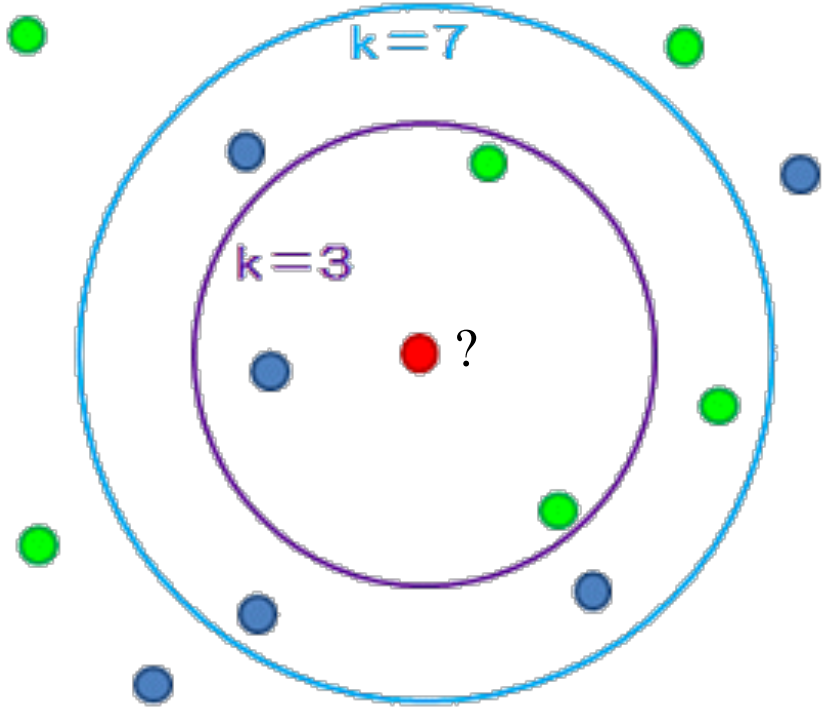
$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

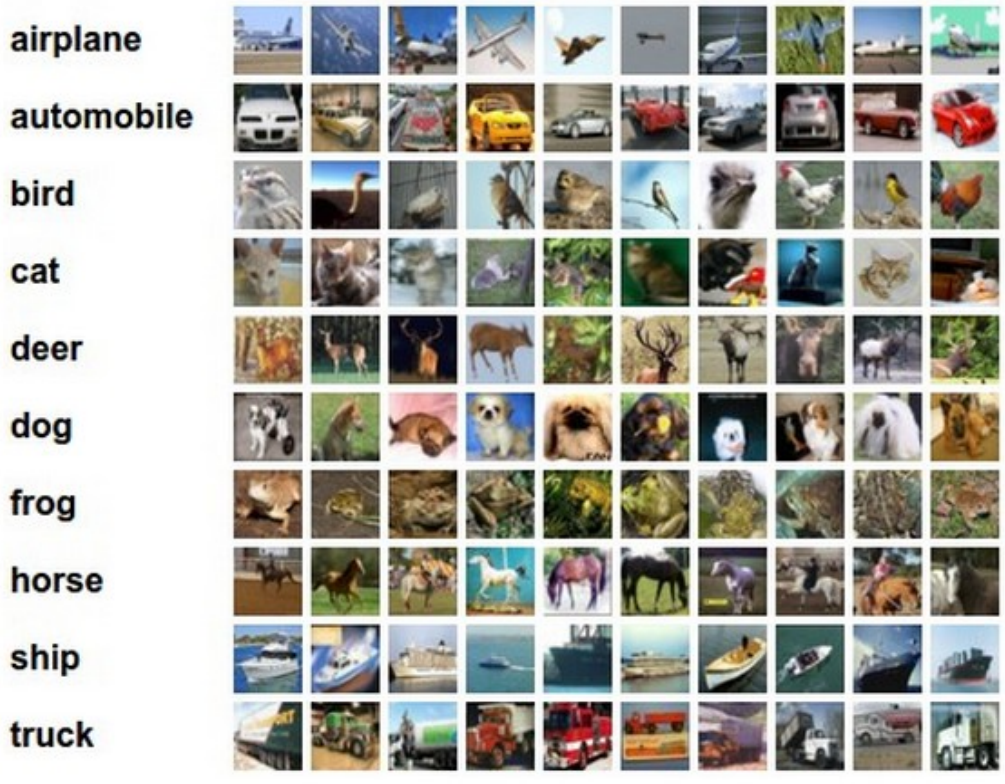
# K-Nearest Neighbor Algorithm

Find the k nearest images, have them vote on the label

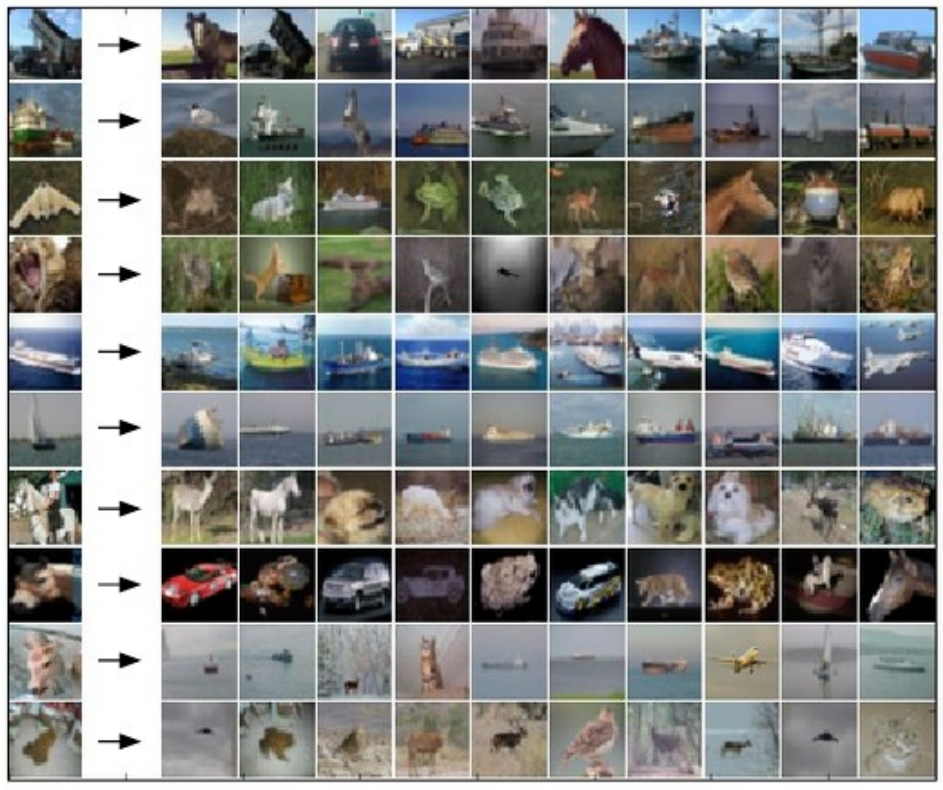


# Example Dataset: CIFAR-10

**10** labels
**50,000** training images
**10,000** test images.




For every test image (first column), examples of nearest neighbors in rows



# Example Dataset: CIFAR-10

**10** labels
**50,000** training images
**10,000** test images.

For every test image (first column), examples of nearest neighbors in rows


Q1: What is the accuracy of the nearest neighbor classifier on the **test** data, when using the Euclidean distance? What about L1 distance?

# Example Dataset: CIFAR-10

**10** labels
**50,000** training images
**10,000** test images.

For every test image (first column), examples of nearest neighbors in rows


Q2: What is the accuracy of the **k**-nearest neighbor classifier on the **test** data? What is the best value of **k**?

# How Do We Set the **Hyperparameters**?

What is the best distance to use?

What is the best value of k to use?

# How Do We Set the **Hyperparameters**?

What is the best distance to use?

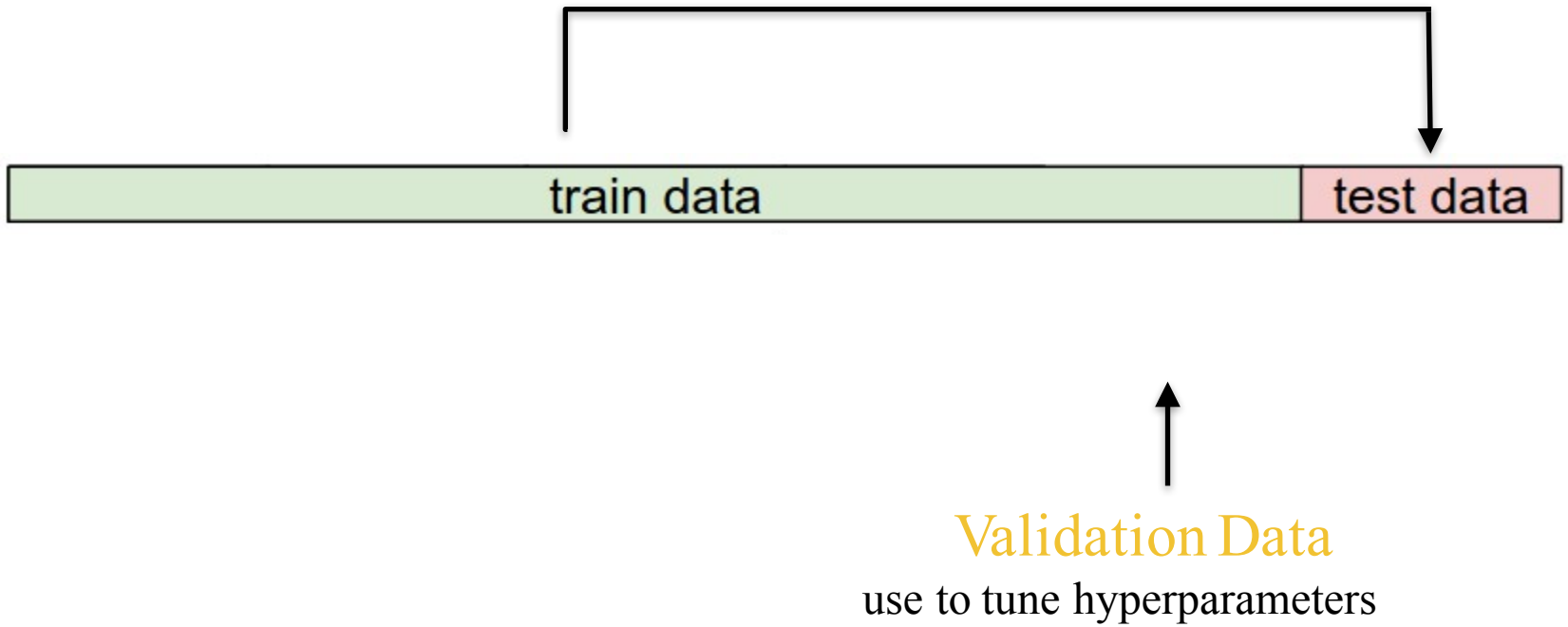
What is the best value of k to use?

Very problem-dependent.

Must try them all out and see what works best.

# Try Out What Hyperparameters that Work Best on **Test Set**

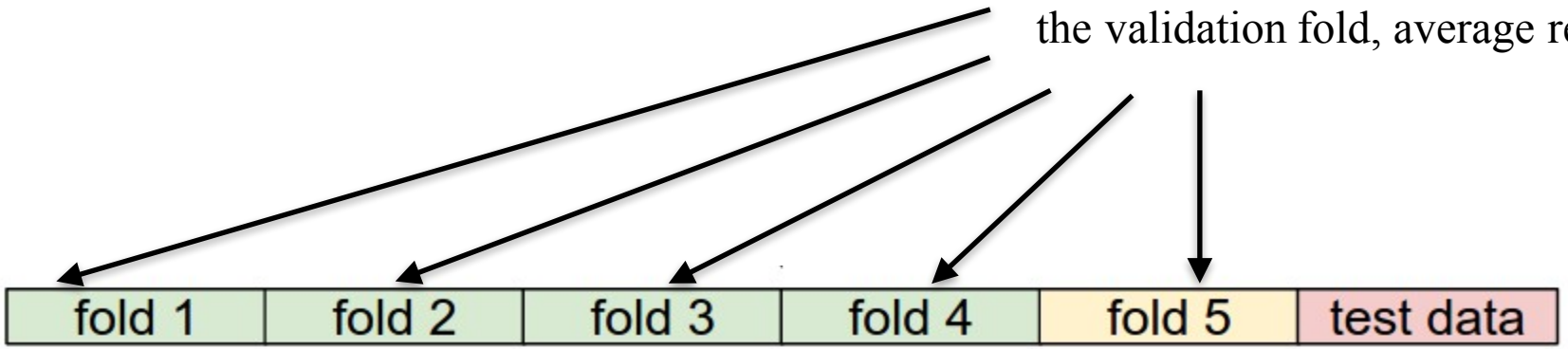
Bad idea (unless trying to win a competition where test set is given).
The test set is a proxy for the generalization performance!
Use only **VERY SPARINGLY,** at the end.



# Try Out What Hyperparameters that Work Best on **Test Set**

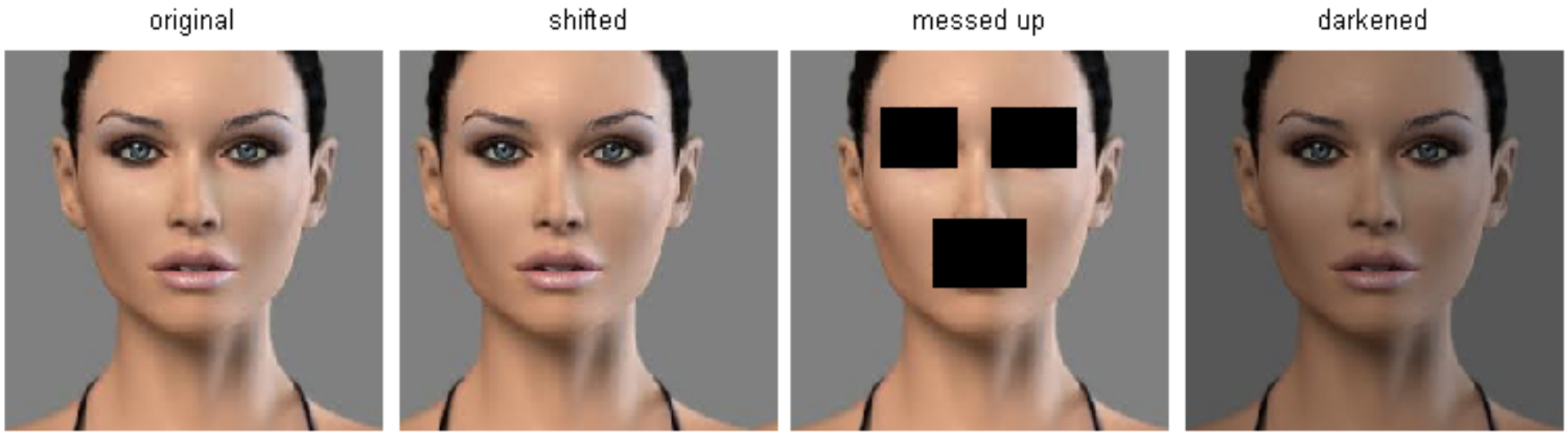
**Cross-validation**

cycle through the choice of which fold is the validation fold, average results.



# K-Nearest Neighbor on Images Never Used

- Terrible performance at test time
- Distance metrics on level of whole images can be very unintuitive




(all 3 images have same L2 distance to the one on the left)

# K-Nearest Neighbor on Images Never Used

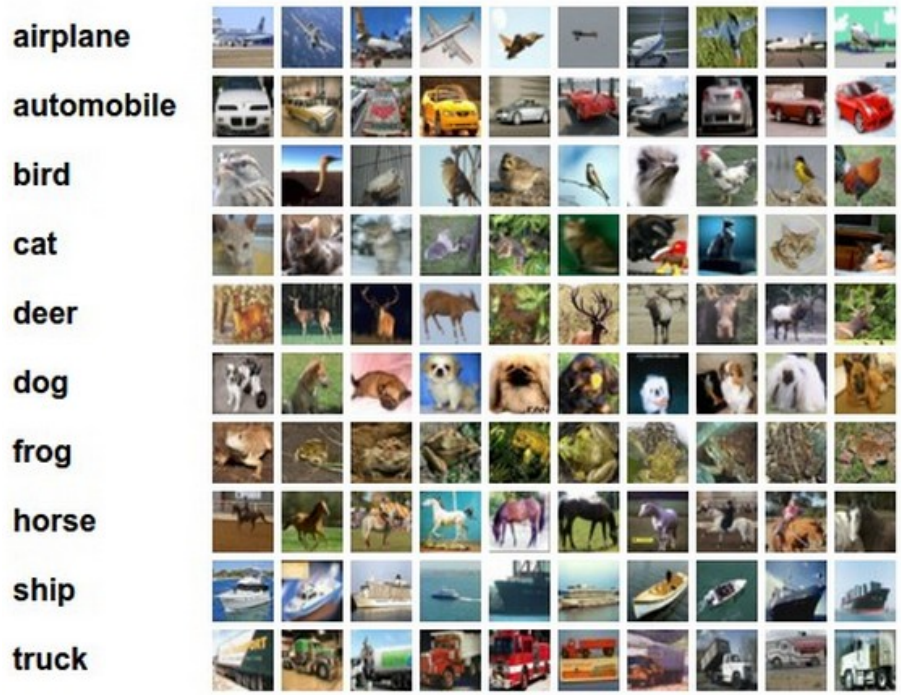
- **Image Classification:** We are given a **Training Set** of labeled images, asked to predict labels on **Test Set**. Common to report the **Accuracy** of predictions (fraction of correctly predicted images)
- We introduced the **k-Nearest Neighbor Classifier**, which predicts the labels based on nearest images in the training set
- We saw that the choice of **distance** and the value of are **hyperparameters** that are tuned using a validation set, or through **cross-validation** if the size of the dat**k**a is small.
- Once the best set of hyperparameters is chosen, the classifier is evaluated once on the test set, and reported as the performance of kNN on that data.

# K-Nearest Neighbor on Images Never Used

- k-NN classifier:
  - must remember all of the training data and store it for future comparisons with the test data
  - This is space inefficient because datasets may easily be gigabytes in size
  - Classifying a test image is expensive since it requires a comparison to all training images

# Parametric Approach: CIFAR-10

**10** labels
**50,000** training images
**10,000** test images
each image is an array of size **32 x 32 x 3 = 3072** numbers total



# **Parametric** Approach

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

**Image**

$f(x, W)$

**10** numbers, indicating class scores

parameters, or "weights"

# **Parametric** Approach: Linear Classifier

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

10x1

$f(x, W) = Wx$

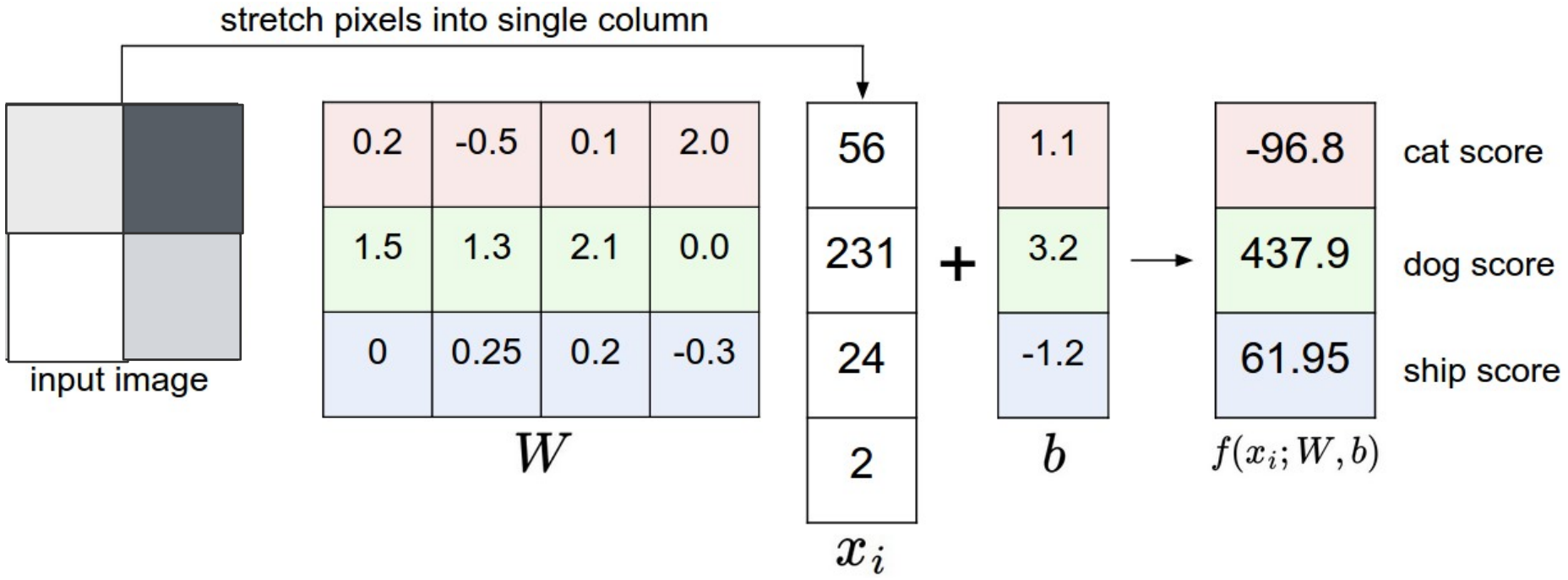
3072x1 (+b) 10x1

10x3072

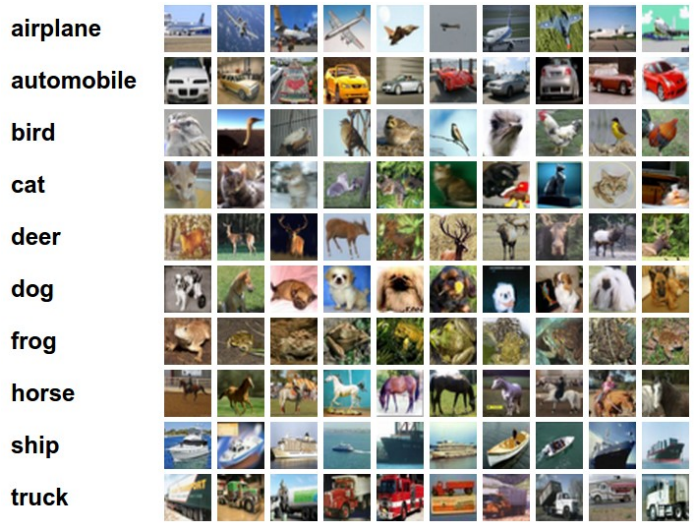
**10** numbers, indicating class scores

parameters, or "weights"

# Example with an Image with 4 Pixels, and 3 Classes (**cat**/**dog**/**ship**)

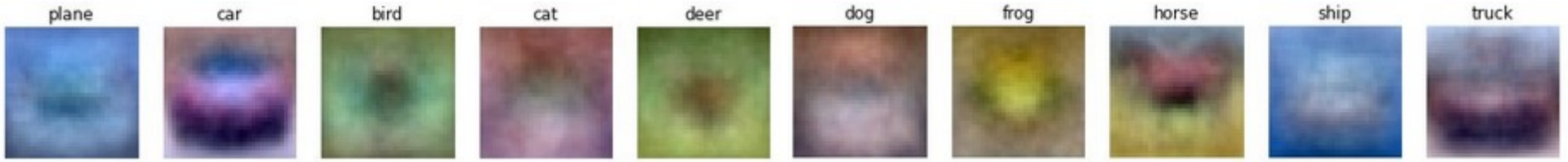


# Interpreting a Linear Classifier

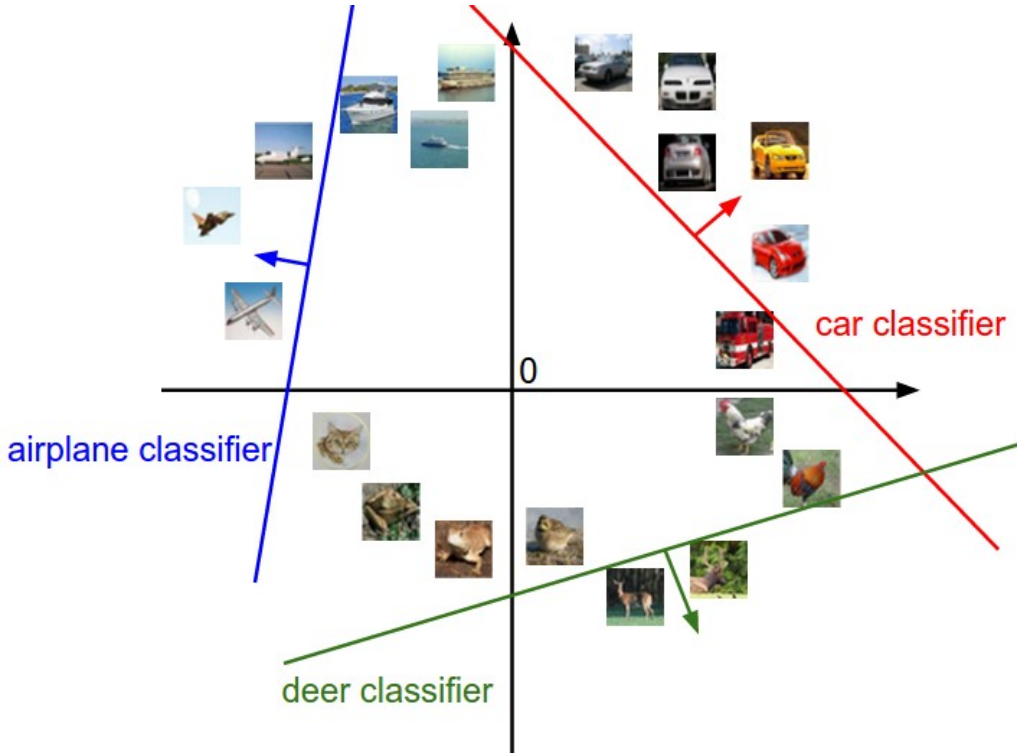



$$f(x_i, W, b) = Wx_i + b$$

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



# Interpreting a Linear Classifier




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

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

# So Far: We Defined a (Linear) Scoring Function: $f(x_i, W, b) = Wx_i + b$

Example class scores for 3 images, with a random W:



| | | | |
|---|---|---|---|
| airplane | -3.45 | -0.51 | 3.42 |
| automobile | -8.87 | **6.04** | 4.64 |
| bird | 0.09 | 5.31 | 2.65 |
| cat | **2.9** | -4.22 | 5.1 |
| deer | 4.48 | -4.19 | 2.64 |
| dog | 8.02 | 3.58 | 5.55 |
| frog | 3.78 | 4.49 | **-4.34** |
| horse | 1.06 | -4.37 | -1.5 |
| ship | -0.36 | -2.09 | -4.79 |
| truck | -0.72 | -2.93 | 6.14 |

# Going forward: Loss function / Optimzation

Example class scores for 3 images, with a random W:



| | | | |
|---|---|---|---|
| airplane | -3.45 | -0.51 | 3.42 |
| automobile | -8.87 | **6.04** | 4.64 |
| bird | 0.09 | 5.31 | 2.65 |
| cat | **2.9** | -4.22 | 5.1 |
| deer | 4.48 | -4.19 | 2.64 |
| dog | 8.02 | 3.58 | 5.55 |
| frog | 3.78 | 4.49 | **-4.34** |
| horse | 1.06 | -4.37 | -1.5 |
| ship | -0.36 | -2.09 | -4.79 |
| truck | -0.72 | -2.93 | 6.14 |

1. Define a loss function that quantifies our unhappiness with the scores across the training data.

2. Come up with a way of efficiently finding the parameters (**W**, **b**) that minimize the loss function. (optimization)

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



| | | | |
|---|---|---|---|
| cat | **3.2** | 1.3 | 2.2 |
| car | 5.1 | **4.9** | 2.5 |
| frog | -1.7 | 2.0 | **-3.1** |

# Softmax Classifier (Multinomial Logistic Regression)

**scores = unnormalized log probabilities of the classes.**

| | |
|---|---|
| cat | **3.2** |
| car | 5.1 |
| frog | -1.7 |

**scores (s)**

# Softmax Classifier (Multinomial Logistic Regression)

**scores = unnormalized log probabilities of the classes.**

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

Softmax function

| | |
|---|---|
| cat | **3.2** |
| car | 5.1 |
| frog | -1.7 |

**scores (s)**

# Softmax Classifier (Multinomial Logistic Regression)

| | |
|---|---|
| cat | **3.2** |
| car | 5.1 |
| frog | -1.7 |
| | **scores (s)** |

**scores = unnormalized log probabilities of the classes.**

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ where $$s = f(x_i; W)$$

$y = [1, 0, 0]^T$ ($y_i$ is 1 (and 0 otherwise) if and only if sample belongs to class $i$)

Want to maximize the log likelihood of the correct class, 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: $$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right) \quad L = \sum_i L_i$$

# Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1,0,0]^T$$

| | |
|---|---|
| cat | **3.2** |
| car | 5.1 |
| frog | -1.7 |

unnormalized log probabilities

# Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1,0,0]^T$$

unnormalized probabilities

| | | | |
|---|---|---|---|
| cat | **3.2** | exp → | **24.5** |
| car | 5.1 | | 164.0 |
| frog | -1.7 | | 0.18 |

unnormalized log probabilities

# Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1,0,0]^T$$

unnormalized probabilities

| | unnormalized log probabilities | exp | unnormalized probabilities | normalize | probabilities |
|---|---|---|---|---|---|
| cat | **3.2** | → | **24.5** | → | **0.13** |
| car | 5.1 | → | 164.0 | → | 0.87 |
| frog | -1.7 | → | 0.18 | → | 0.00 |

# Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1, 0, 0]^T$$

unnormalized probabilities

| | unnormalized log probabilities | exp | | normalize | probabilities |
|---|---|---|---|---|---|
| cat | **3.2** | → | **24.5** | → | **0.13** |
| car | 5.1 | → | 164.0 | → | 0.87 |
| frog | -1.7 | → | 0.18 | → | 0.00 |

L_i = -log(0.13)
= **0.89**

# Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1, 0, 0]^T$$

Q: What is the min/max possible loss L_i?

unnormalized probabilities

| | unnormalized log probabilities | exp | unnormalized probabilities | normalize | probabilities |
|---|---|---|---|---|---|
| cat | **3.2** | → | **24.5** | → | **0.13** |
| car | 5.1 | → | 164.0 | → | 0.87 |
| frog | -1.7 | → | 0.18 | → | 0.00 |

L_i = -log(0.13)
= **0.89**

# Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

In torch7:
**nn.CrossEntropyCriterion**

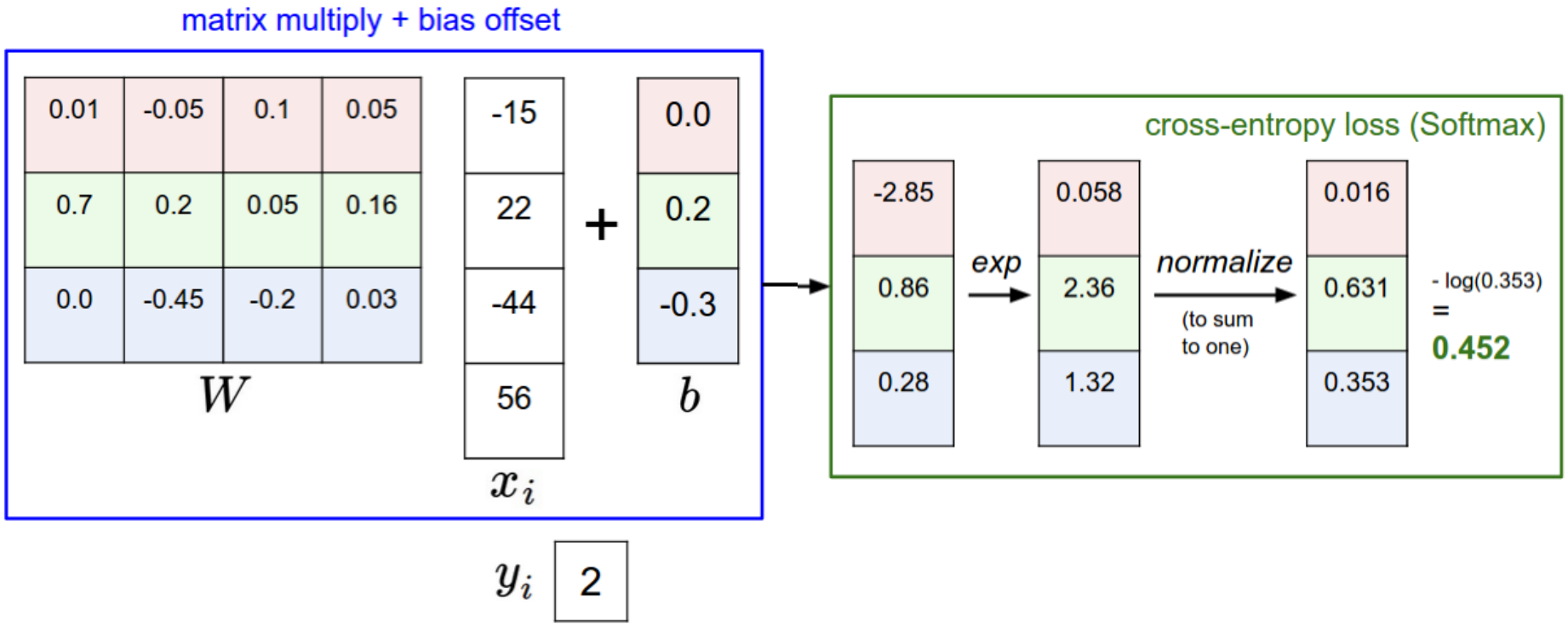
$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1,0,0]^T$$

unnormalized probabilities

| | unnormalized log probabilities | exp | unnormalized probabilities | normalize | probabilities |
|---|---|---|---|---|---|
| cat | **3.2** | → | **24.5** | → | **0.13** |
| car | 5.1 | → | 164.0 | → | 0.87 |
| frog | -1.7 | → | 0.18 | → | 0.00 |

L_i = -log(0.13)
= **0.89**

# Softmax Classifier (Multinomial Logistic Regression)



# Weight Regularization

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

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

$$L = \sum_i L_i + \lambda R(W)$$

($y$ is a one−hot vector where $y_i$ is 1 (and 0 otherwise) if and only if sample belongs to class $i$

# Weight Regularization

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

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

$$L = \sum_i L_i + \lambda R(W)$$

($y$ is a one−hot vector where $y_i$ is 1 (and 0 otherwise) if and only if sample belongs to class $i$

# Weight Regularization

$$L = \sum_i L_i + \lambda R(W)$$

Regularization strength

(another hyperparameter)

In common use:

**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) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Dropout (will see next week)

# Weight Regularization: L2 - motivation

$$x = [1, 1, 1, 1]$$

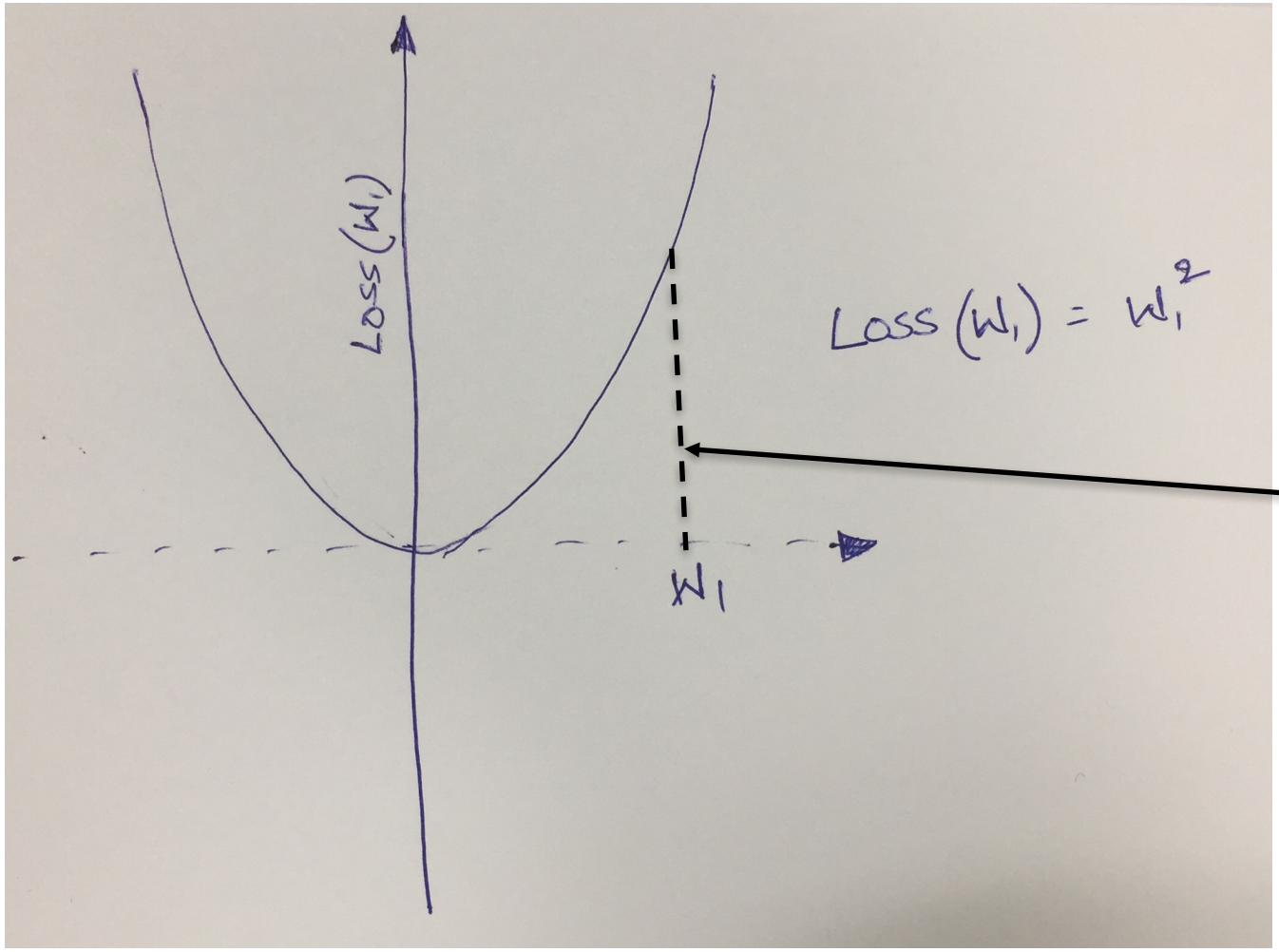
$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

Which $\boldsymbol{w}$ would L2 regularization choose?

# Optimization: How to get good values for **W**?



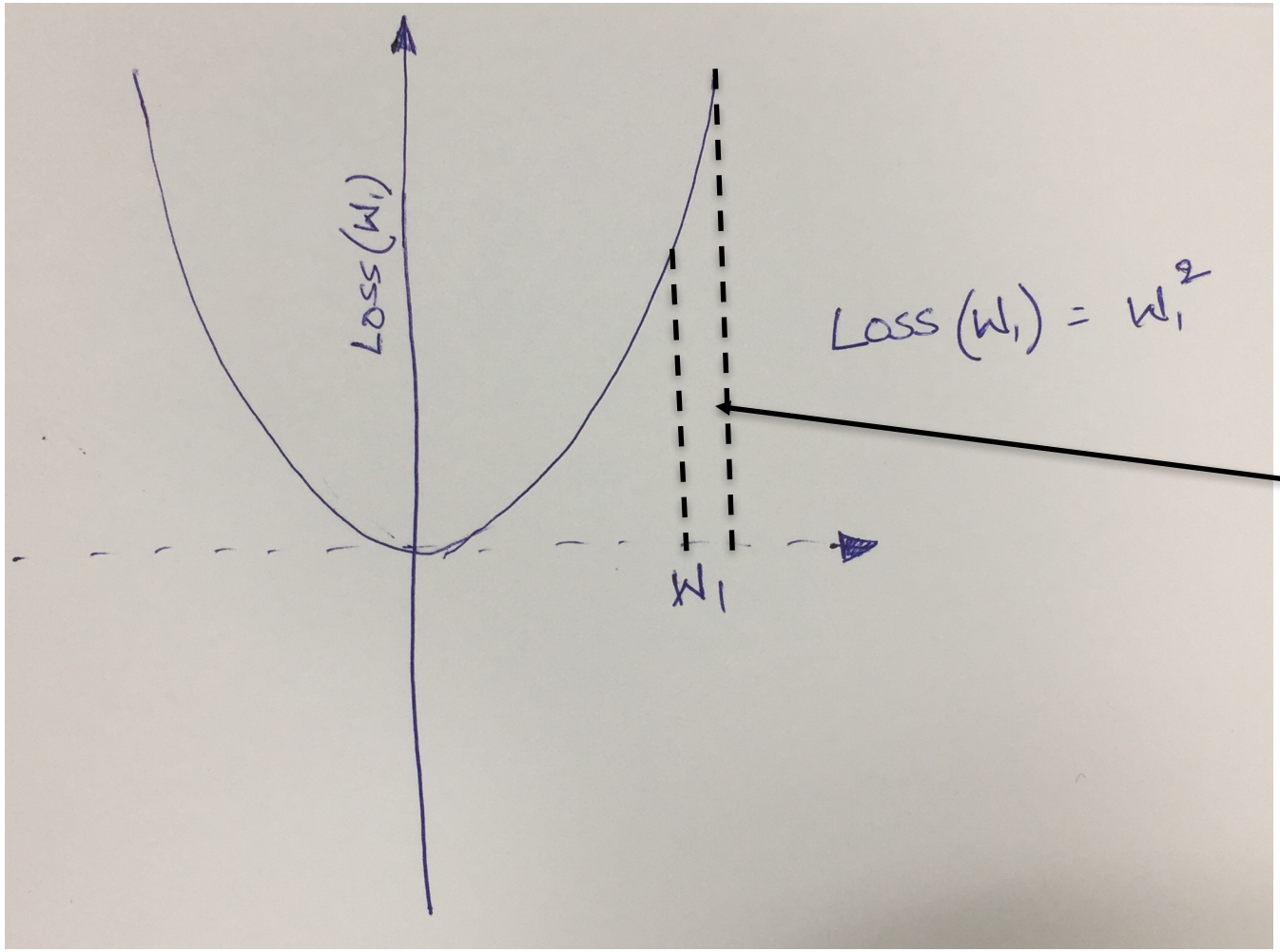

In 1-dimension, the derivative of a function:

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

$W_1 = 10,$

$Loss(W_1) = 100$

# Optimization: How to get good values for **W**?



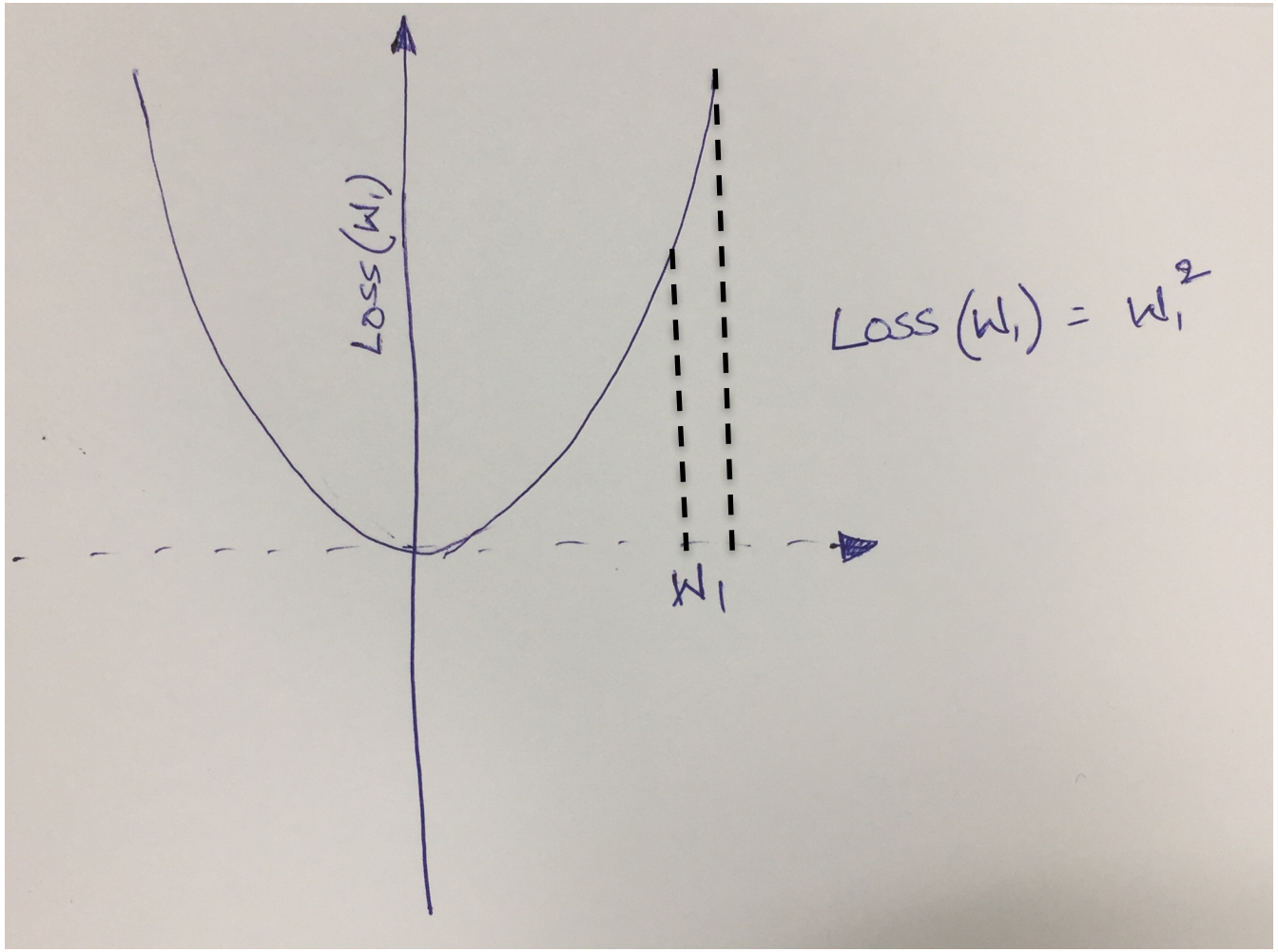

In 1-dimension, the derivative of a function:

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

$$W_1 = 10.01,$$

$$Loss(W_1) = 100.2001$$

# Optimization: How to get good values for **W**?




In 1-dimension, the derivative of a function:

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

$$\frac{dLoss(W_1)}{dW_1} = (100.2001 - 100)/0.01$$

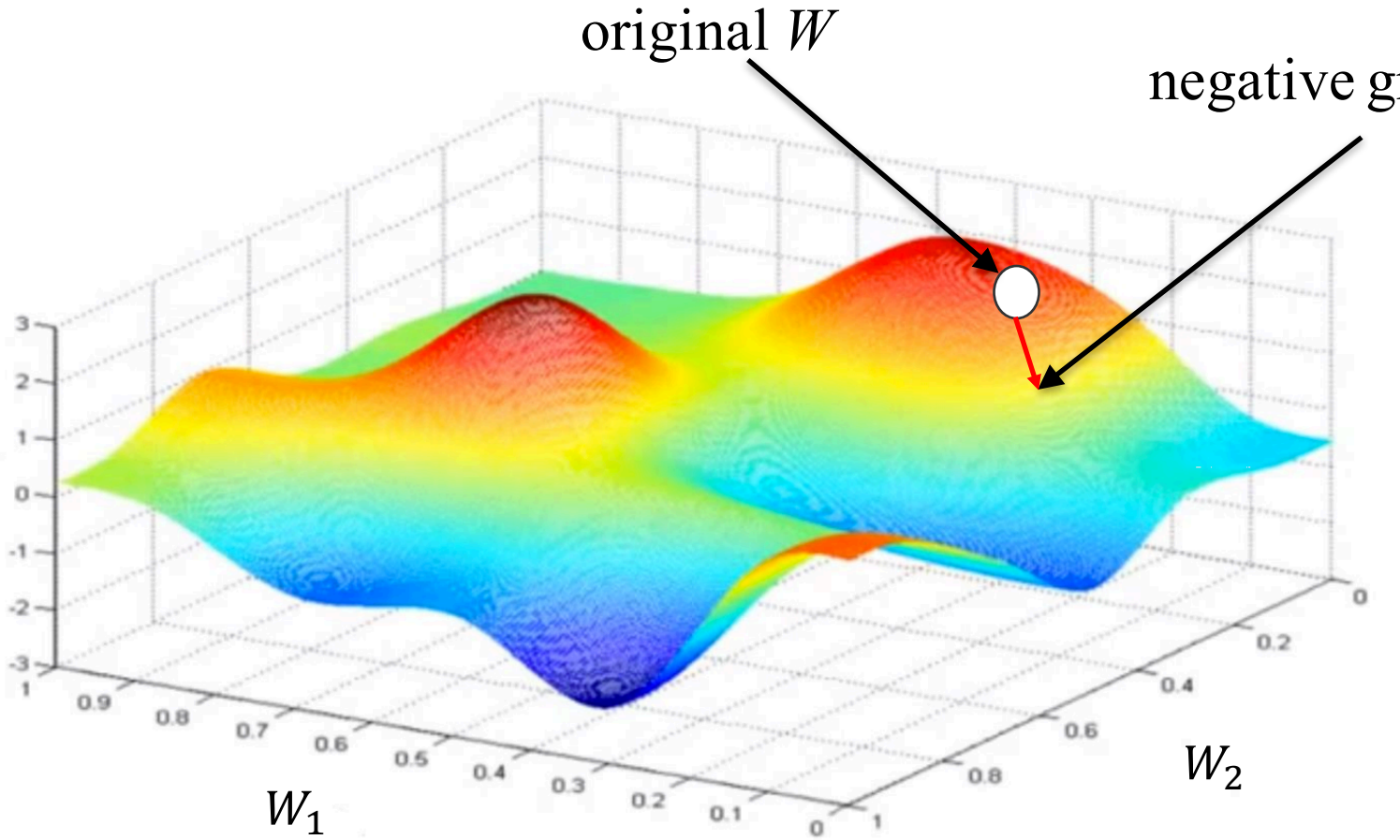
$$= 20.01$$

(assume $\lambda$=0.1)

$$W_1 = W_1 - \lambda \frac{d\ Loss(W_1)}{dW_1}$$

Move in the direction of negative slope

# Optimization: How to get good values for **W**'s?




$$\frac{\partial Loss(W)}{\partial W} = \left[\frac{\partial Loss(W)}{\partial W_1}, \frac{\partial Loss(W)}{\partial W_2}\right]$$

$$W_i = W_i - \lambda \frac{\partial Loss(W_i)}{\partial W_i}$$

In multiple dimensions, the derivative of the loss with respect to the weight vector is called the gradient, and is nothing but a row vector, each component being component wise partial derivatives (more soon)

# Optimization: How to get good values for **W**'s?

In [1]:
```
require 'nn';
model = nn.Linear(1,2)
x = torch.rand(1)
y = torch.Tensor({1})
```

In [2]:
```
op = model:forward(x)
print(x)
print(op)
print(y)
print(model.weight)
```

Out[2]:
```
 0.1854
[torch.DoubleTensor of size 1]

 0.1313
-0.4951
[torch.DoubleTensor of size 2]

 1
[torch.DoubleTensor of size 1]

-0.8399
-0.6943
[torch.DoubleTensor of size 2x1]
```

# Optimization: How to get good values for **W**'s?

```
In [3]: criterion = nn.CrossEntropyCriterion()
        e1 = criterion:forward(op, y)
        print(e1)

Out[3]: 0.78581595010034
```

$f(x)$

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.

# Optimization: How to get good values for **W**'s?

```
In [4]: model.weight[1][1] = model.weight[1][1] + 0.00001
        e2 = criterion:forward(model:forward(x), y)
        print((e2 - e1)/0.00001)

Out[4]: -0.064589666493031
```

$f(x+h)$

```
In [5]: model.weight[1][1] = model.weight[1][1] - 0.00001
        model.weight[2][1] = model.weight[2][1] + 0.00001
        e3 = criterion:forward(model:forward(x), y)
        print((e3 - e1)/0.00001)

Out[5]: 0.06458974454171
```

$f(x+h)$

# Optimization: How to get good values for **W**'s?

Loss is just a function of W:

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

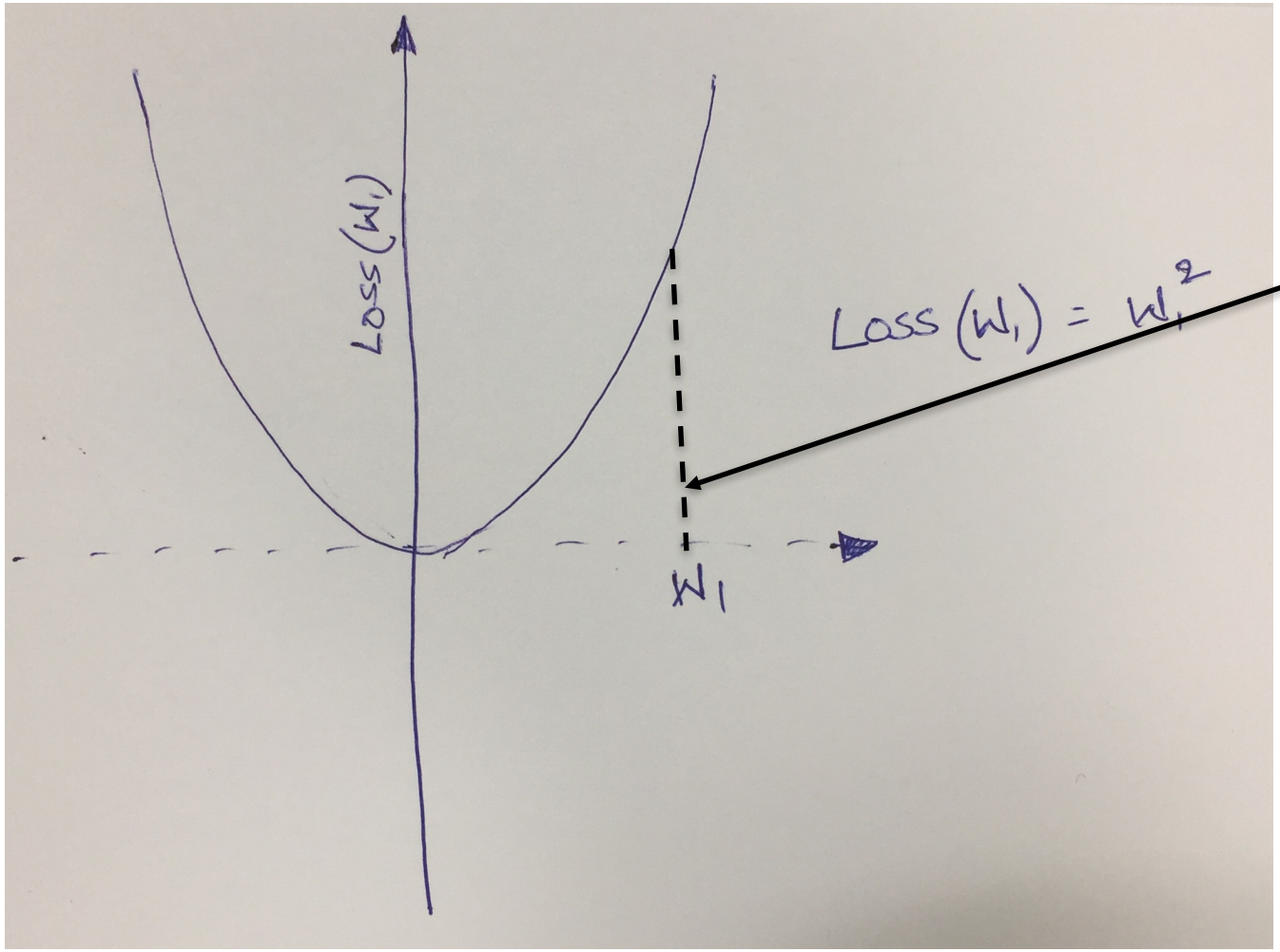
$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right)$$

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

want $\nabla_W L$

$y$ is a one−hot vector where $y_i$ is 1 (and 0 otherwise) if and only if sample belongs to class $i$

# Optimization: How to get good values for **W**?




$W_1 = 10,$

$Loss(W_1) = 100$

Assume $\lambda$=0.1

$$\frac{dLoss(W_1)}{dW_1} = 2 \times W_1$$

$$W_1 = W_1 - \lambda \frac{d\ Loss(W_1)}{dW_1}$$

Move in the direction of negative slope

# Optimization: How to get good values for **W**'s?

```
In [6]: model.weight[2][1] = model.weight[2][1] - 0.00001
        df_do = criterion:backward(model:forward(x),y)
        model:backward(x, df_do)
        print(model.gradWeight)

Out[6]: 0.01 *
         -6.4590
          6.4590
        [torch.DoubleTensor of size 2x1]
```

In summary:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

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

# Optimization: How to get good values for **W**'s?

```
In [4]: model.weight[1][1] = model.weight[1][1] + 0.00001
        e2 = criterion:forward(model:forward(x), y)
        print((e2 - e1)/0.00001)

Out[4]: -0.064589666493031
```

$f(x+h)$

```
In [5]: model.weight[1][1] = model.weight[1][1] - 0.00001
        model.weight[2][1] = model.weight[2][1] + 0.00001
        e3 = criterion:forward(model:forward(x), y)
        print((e3 - e1)/0.00001)

Out[5]: 0.06458974454171
```

$f(x+h)$

# Optimization: How to get good values for **W**'s and **b**'s?

Gradient descent:

```
while true -- for new x,y pairs
  df_do = criterion:backward(model:forward(x),y)
  model:backward(x,  df_do)
  model.weight  = model.weight  - learning_rate  * model.gradWeight
end
```

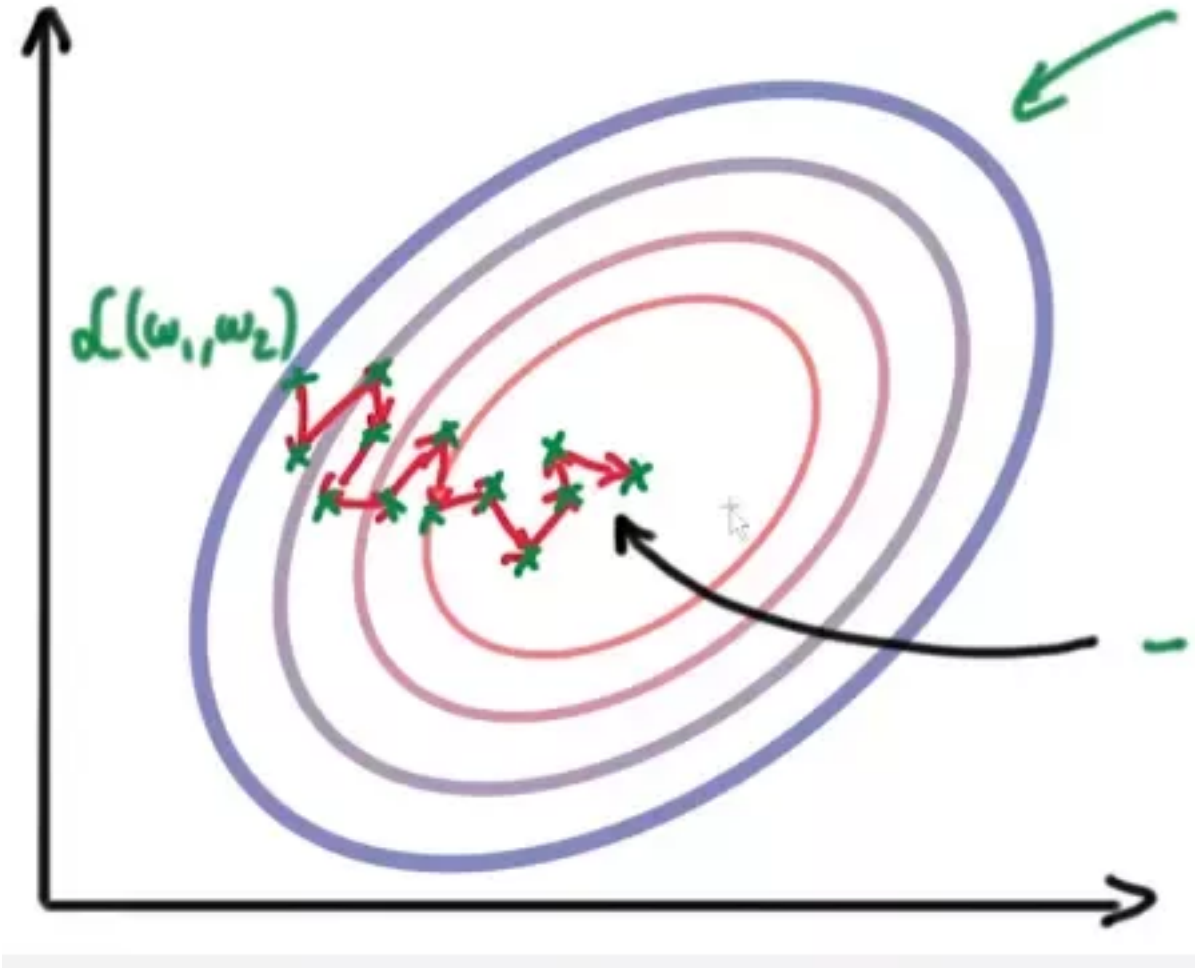
# Optimization: How to get good values for **W**'s and **b**'s?

## Gradient descent (vanila mini-batch):

```
while true -- for new x,y pairs
  df_do_all = err:backward(model:forward(batch_x), batch_y)
  df_do = mean(df_do_all)
  model:backward(batch_x, df_do)
  model.weight = model.weight - learning_rate * model.gradWeight
end
```

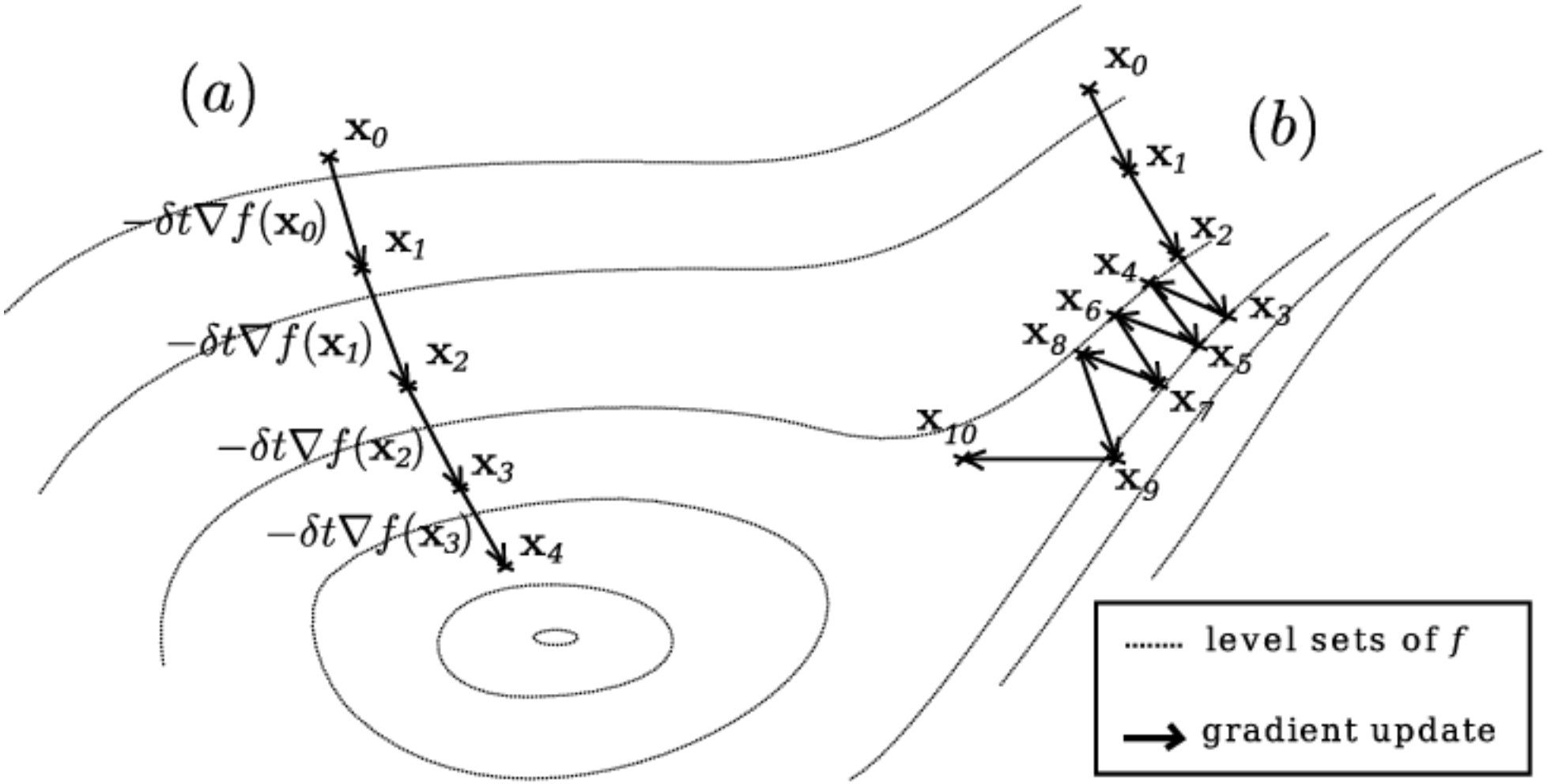
- only use a small portion of the training set to compute the gradient.

# Gradient Descent without Momemtum

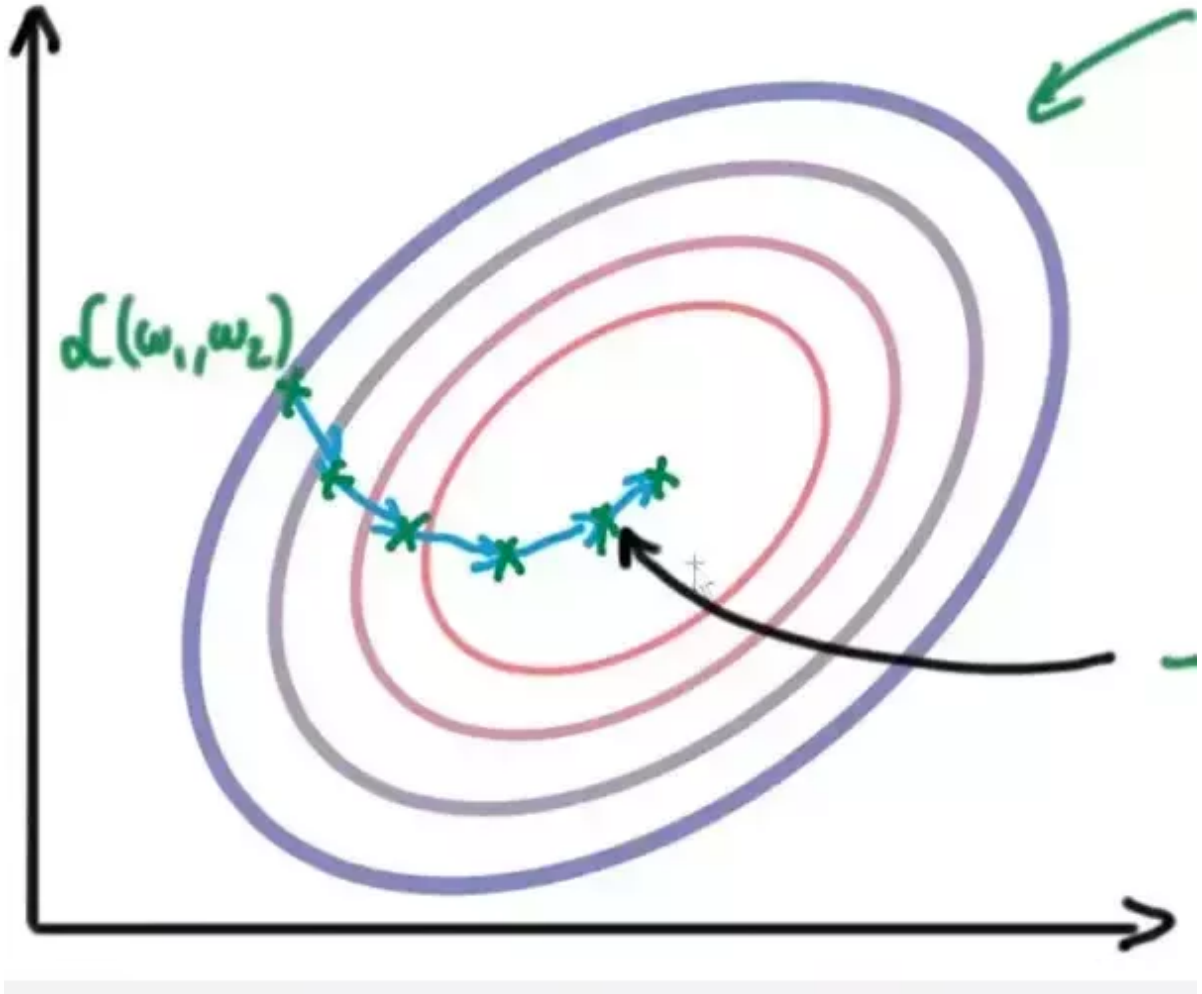



- http://sebastianruder.com/optimizing-gradient-descent/
- https://www.quora.com/What-are-differences-between-update-rules-like-AdaDelta-RMSProp-AdaGrad-and-AdaM

# Gradient Descent without Momemtum



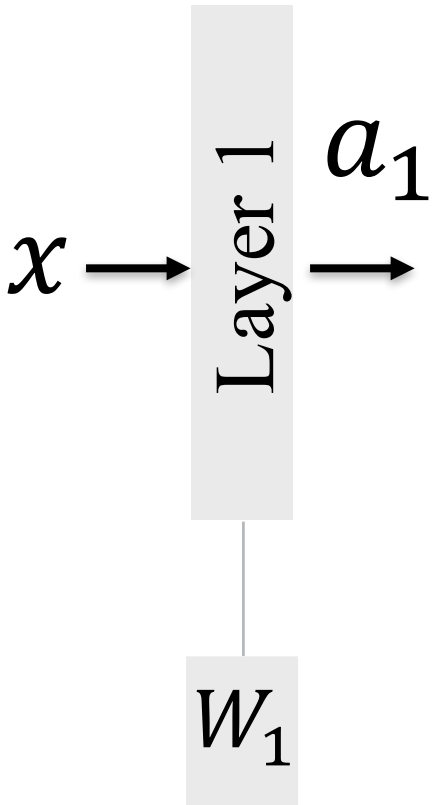
# SGD with Momemtum



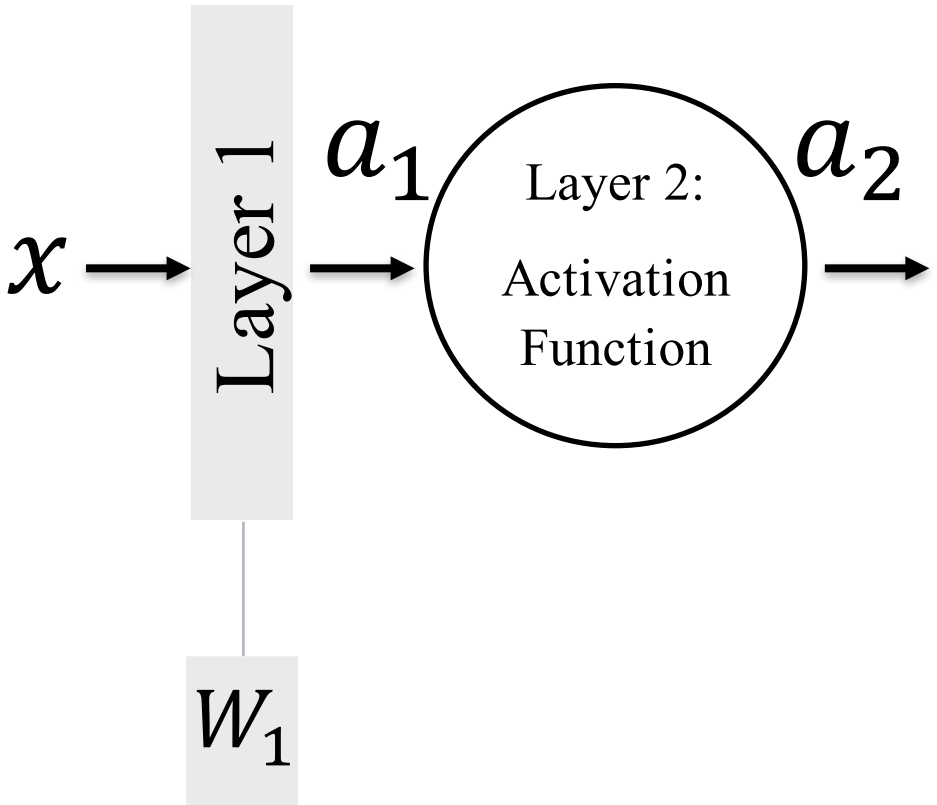

$$\boldsymbol{m_t} = \propto \boldsymbol{m_{t-1}} + \lambda \frac{\partial L}{\partial \boldsymbol{W}}$$

$$\boldsymbol{W} = \boldsymbol{W} \; - \boldsymbol{m_t}$$

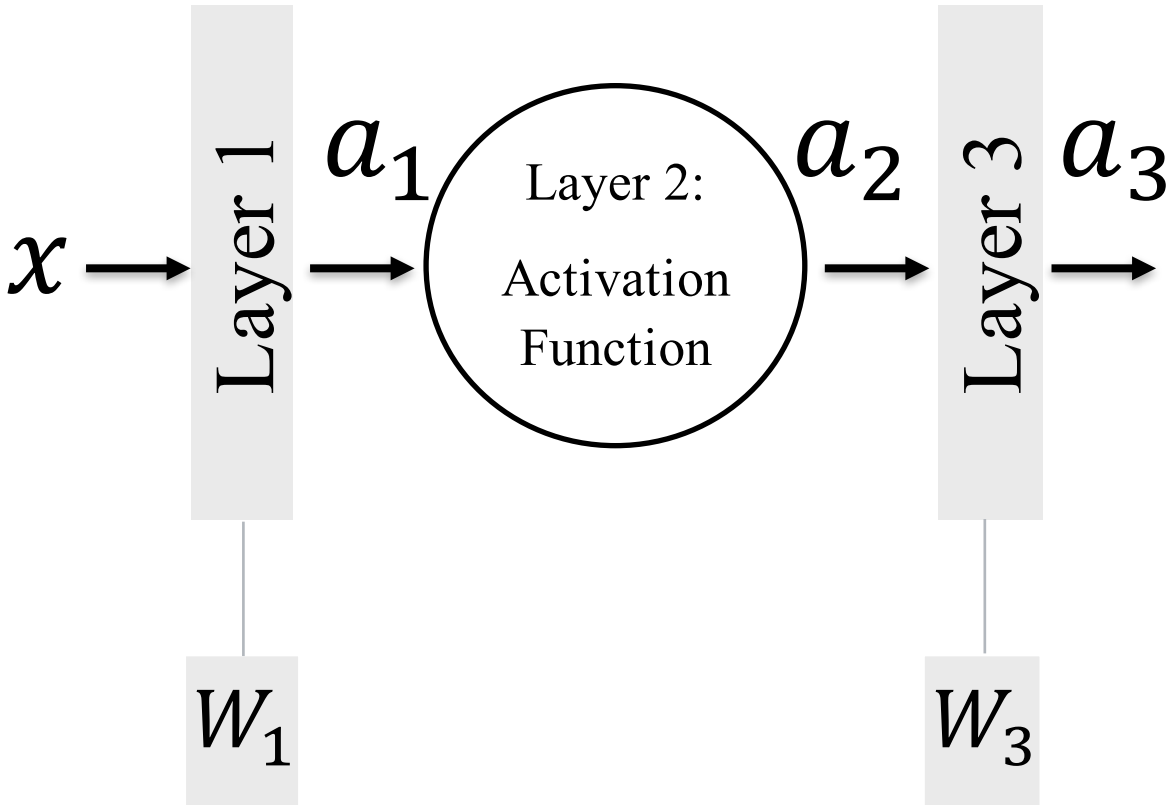
# Multiple Layers – Feed Forward



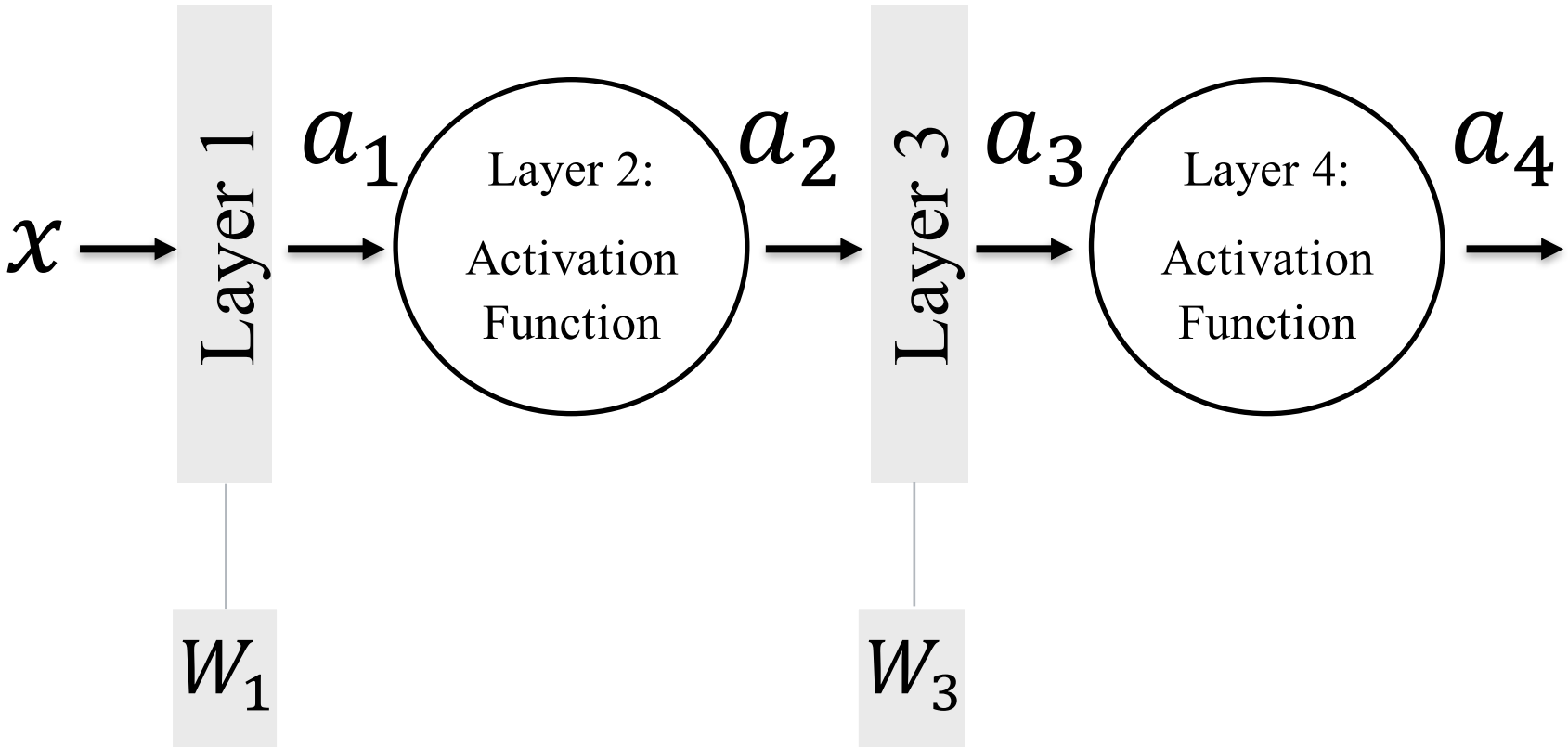
# Multiple Layers – Feed Forward



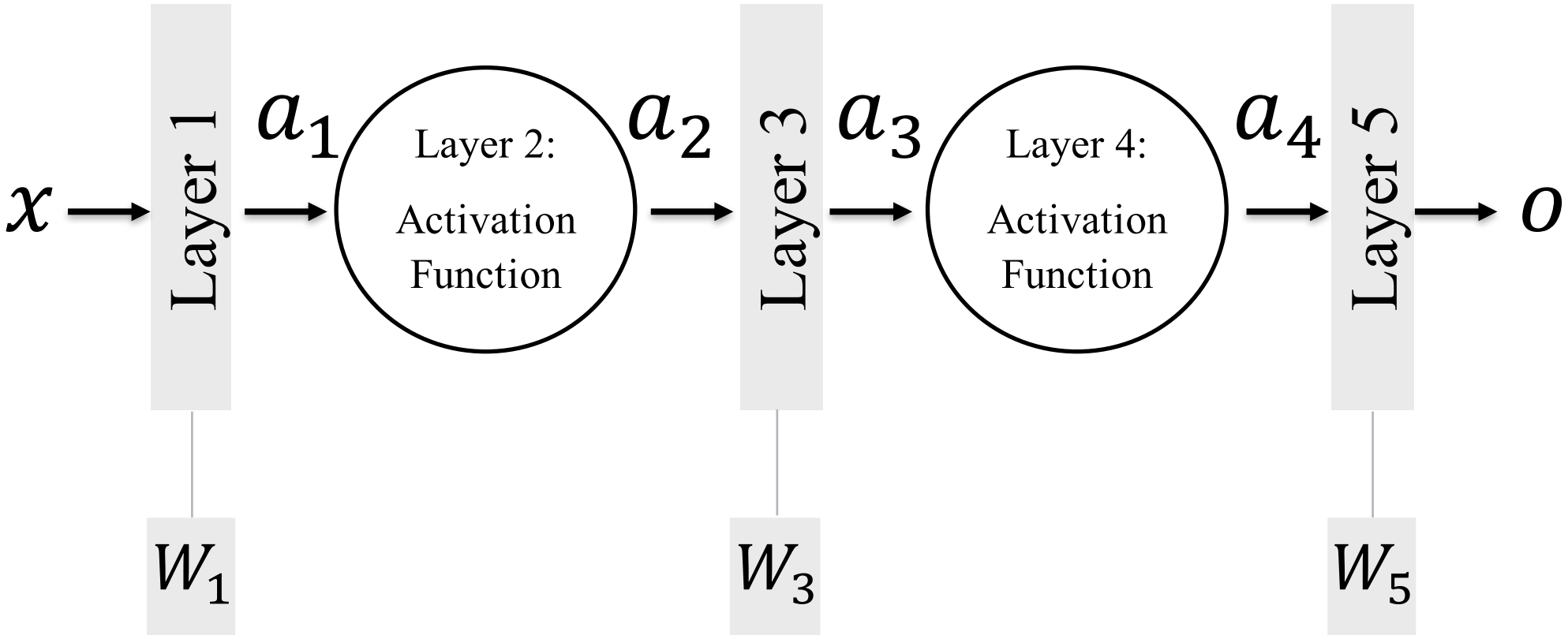
# Multiple Layers – Feed Forward



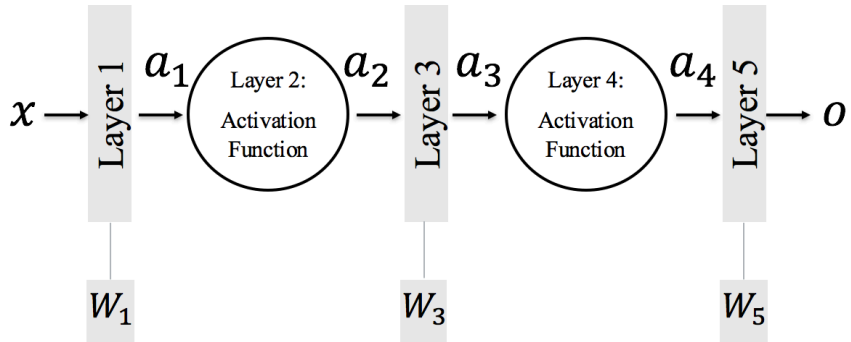
# Multiple Layers – Feed Forward



# Multiple Layers – Feed Forward



# Multiple Layers –Feed Forward - Composition of Functions




$$a_1 = F(x, W_1),\ x \in \mathbb{R}^m$$

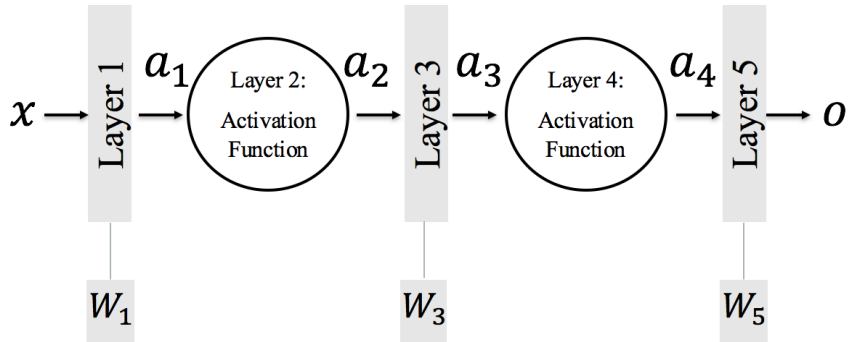
$$a_2 = G(a_1)$$

$$a_3 = H(a_2, W_3),$$

$$a_4 = J(a_3)$$

$$o = K(a_4, W_5)\ \epsilon\ \mathbb{R}^n$$

# Multiple Layers –Feed Forward - Composition of Functions




$$a_1 = F(x, W_1),\ x \in \mathbb{R}^m$$

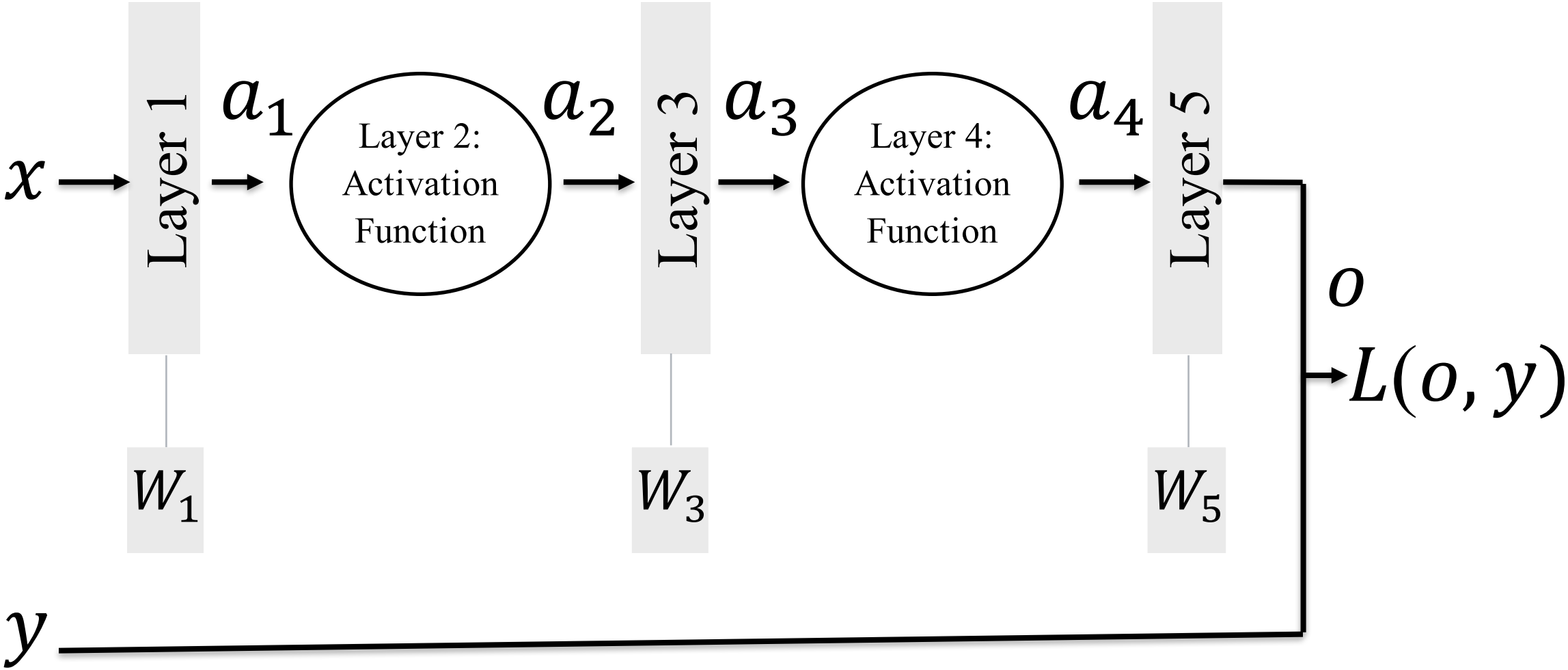
$$a_2 = G(a_1)$$

$$a_3 = H(a_2, W_3),$$

$$a_4 = J(a_3)$$

$$o = K(a_4, W_5) = K(J(H(G(F(x, W_1)), W_3)), W_5) \in \mathbb{R}^n$$

# Multiple Layers – Feed Forward - Loss



Thank you!