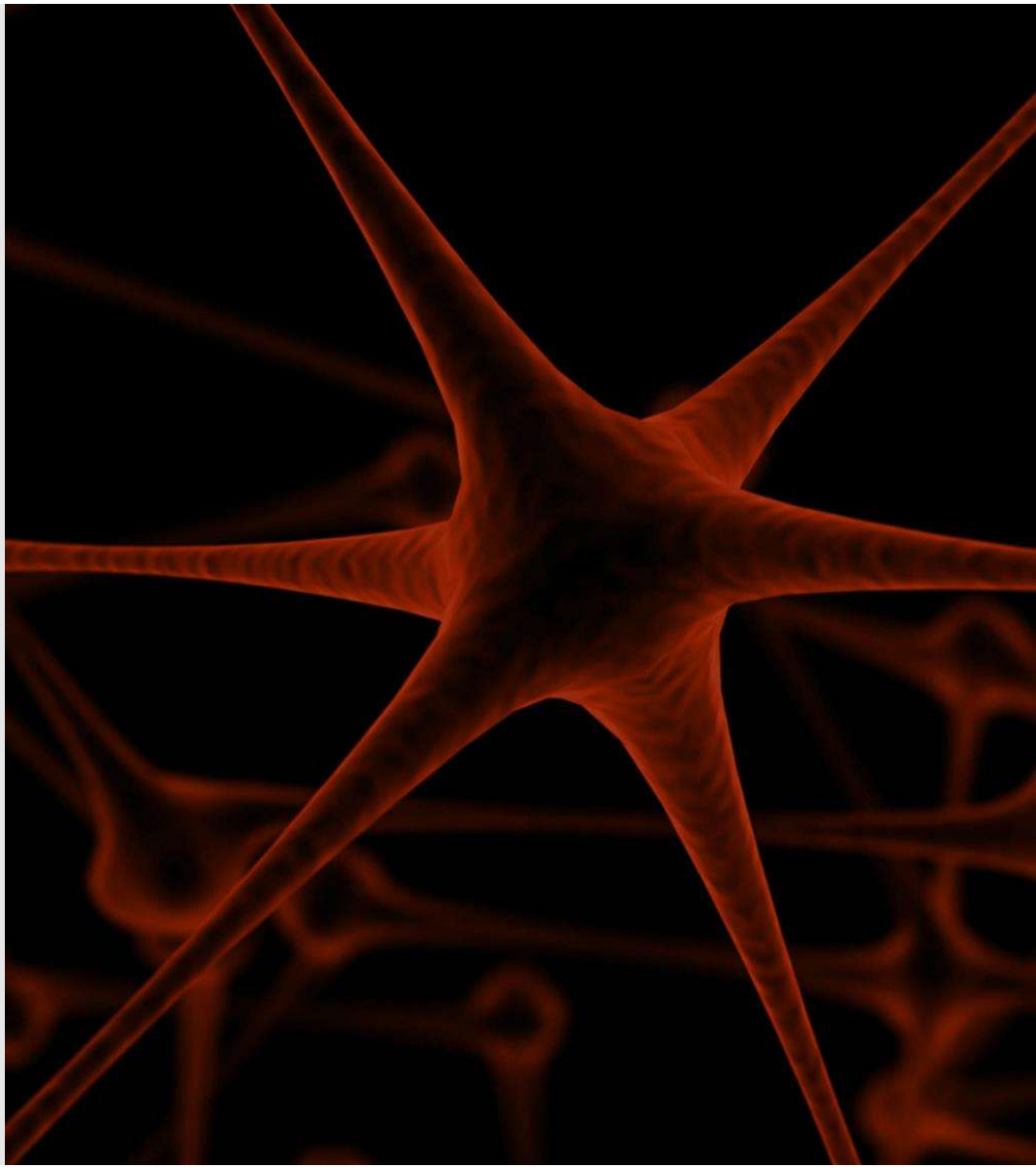
Microsoft

# De-mystifying Deep Learning

*Anusua Trivedi*
*Data Scientist*
*Email: antriv@microsoft.com*
*Twitter: @anurive*



# Talk Outline

- ❑ Deep Learning (DL)
- ❑ Deep Neural Networks (DNN)
- ❑ Types of DNNs
- ❑ DL Frameworks
- ❑ Use Cases

# Resources

https://github.com/antriv/DeepLearning-1-Day-Training

# Traditional ML Vs DL

Traditional ML requires manual feature extraction/engineering

Feature extraction for unstructured data is very difficult

Deep learning can automatically learn features in data

Deep learning is largely a "black box" technique, updating learned weights at each layer

# Why is DL popular?

- DL models has been here for a long time
  - Fukushima (1980) – Neo-Cognitron
  - LeCun (1989) – Convolutional Neural Network

- DL popularity grew recently
  - With growth of Big Data
  - With the advent of powerful GPUs

# Deep learning begins with a little function

It all starts with a humble linear function called a perceptron.

weight1 ✕ input1
weight2 ✕ input2
\+ weight3 ✕ input3
sum

Perceptron:
If sum > threshold: output 1
Else: output 0

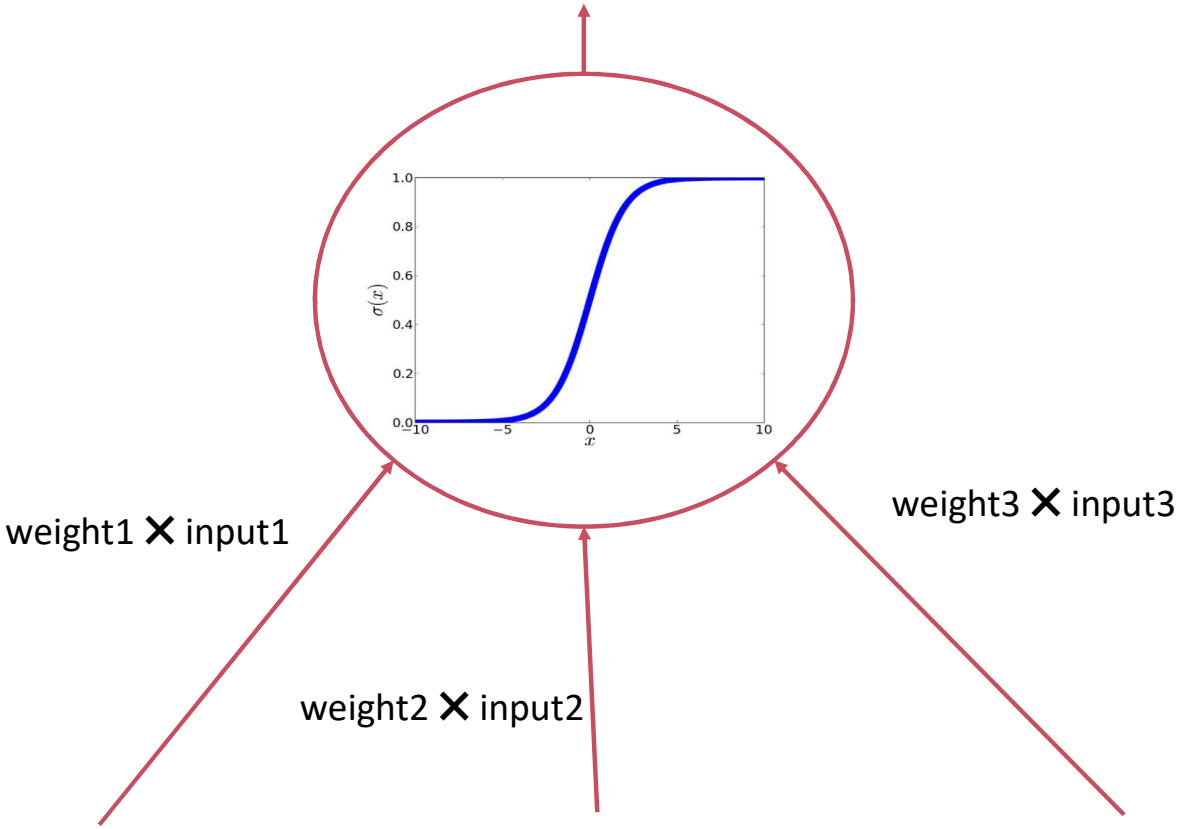
Example: The inputs can be your data. Question: Should I buy this car?

0.2 ✕ gas mileage
0.3 ✕ horsepower
\+ 0.5 ✕ num cup holders
sum

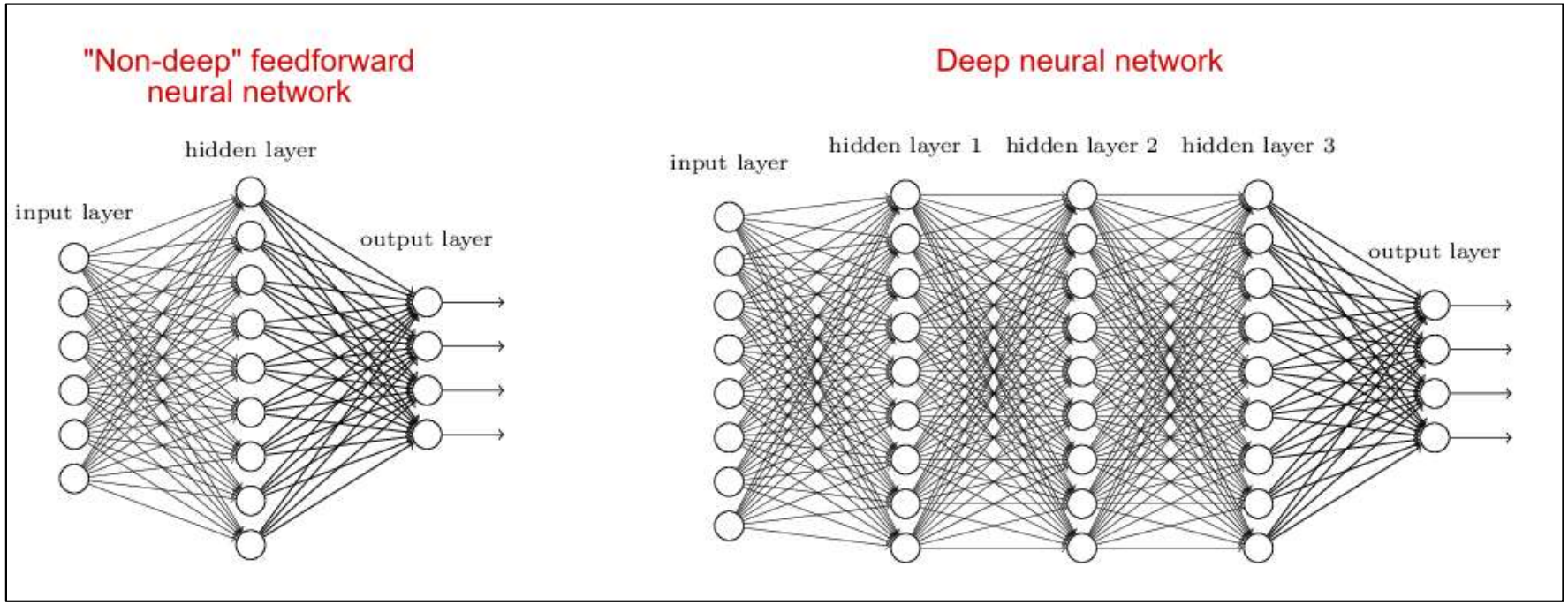
Perceptron:
If sum < threshold: buy
Else: walk

# These little functions are chained togeter

Deep learning comes from chaining a bunch of these little functions together.
Chained together, they are called neurons.

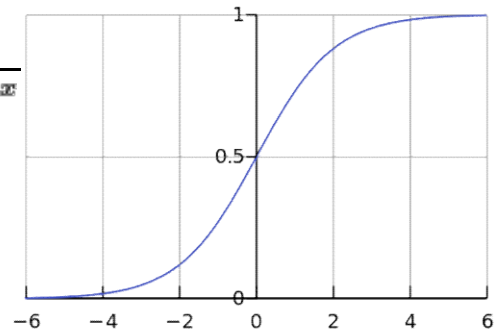


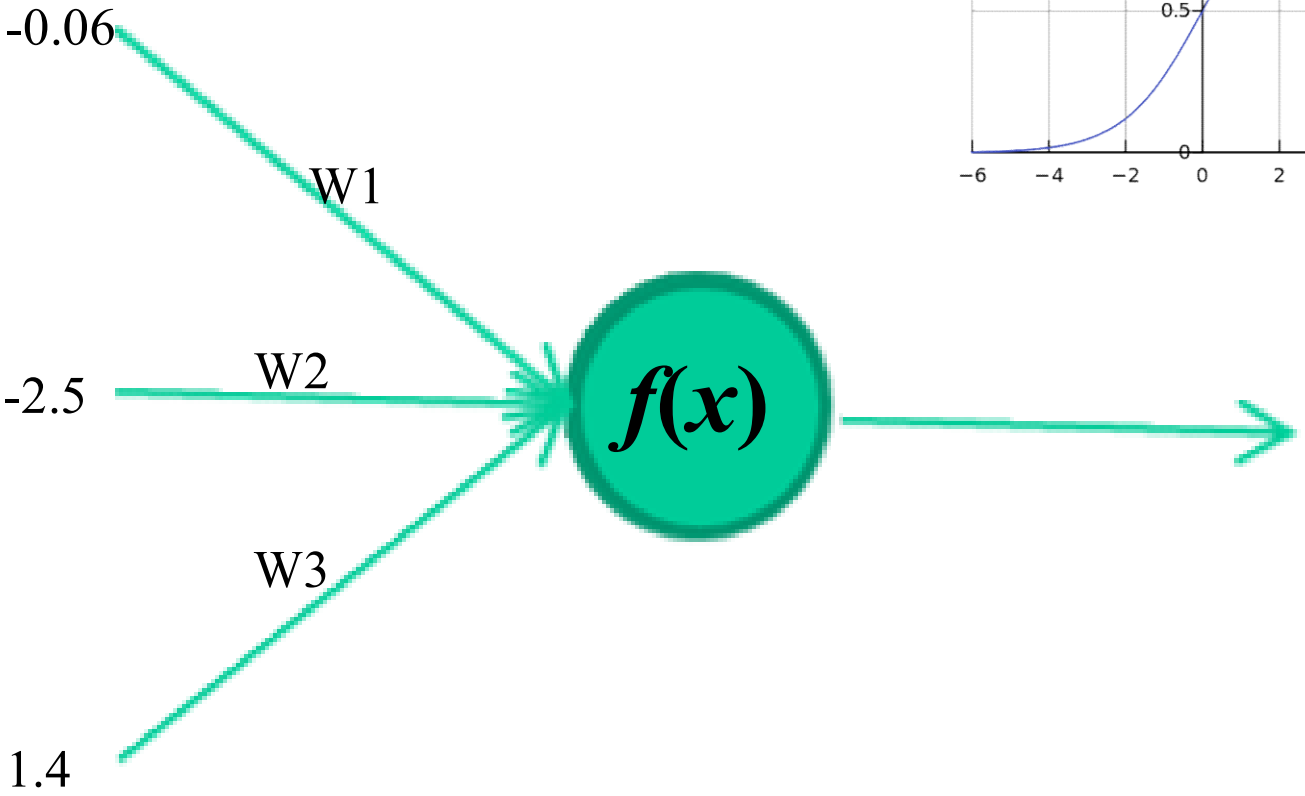
# Deep Neural Network (DNN)



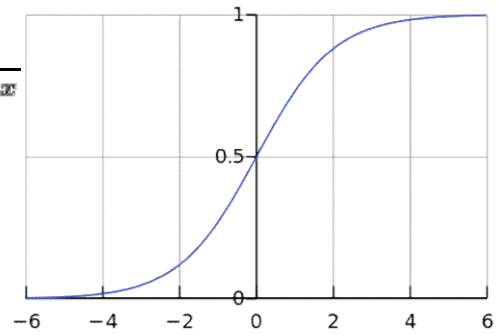

*http://neuralnetworksanddeeplearning.com/chap5.html

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



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



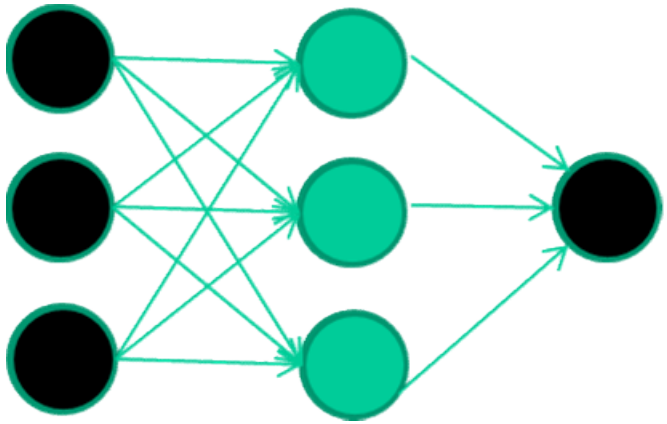




$x = -0.06 \times 2.7 + 2.5 \times 8.6 + 1.4 \times 0.002 = 21.34$

*A dataset*

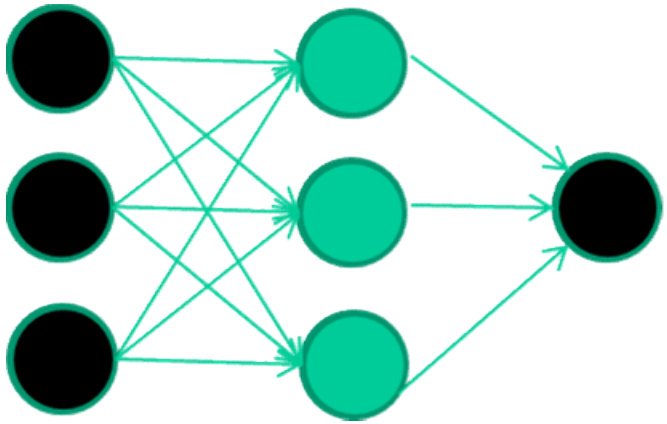
| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc … | | | |



*Training the neural network*

| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |

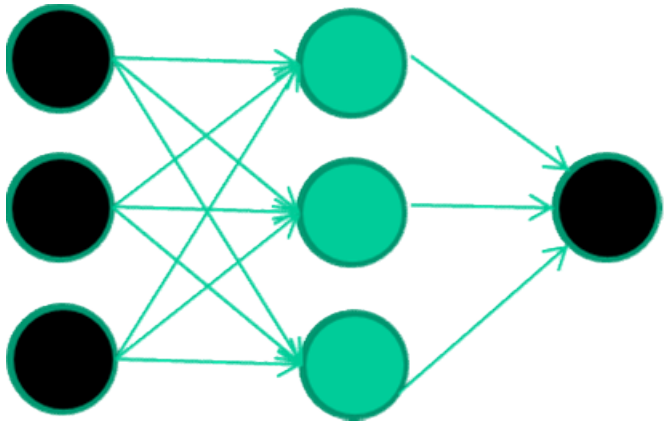
etc …



*Training data*

| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc … | | | |

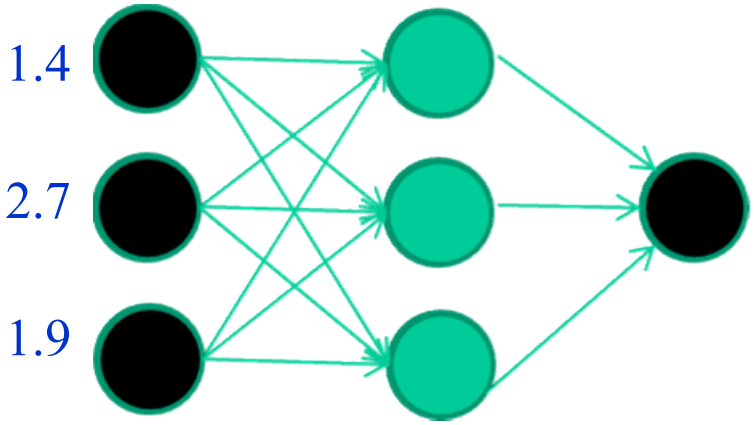
**Initialise with random weights**



*Training data*

| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc … | | | |

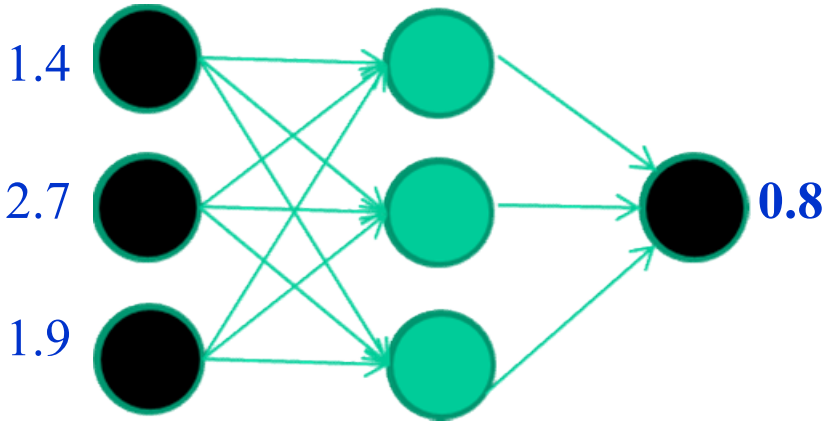
**Present a training pattern**



*Training data*

| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc … | | | |

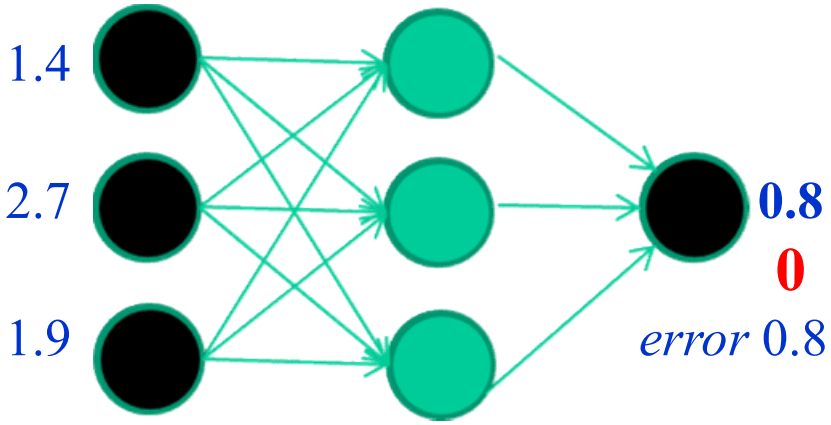
**Feed it through to get output**



*Training data*

| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc … | | | |

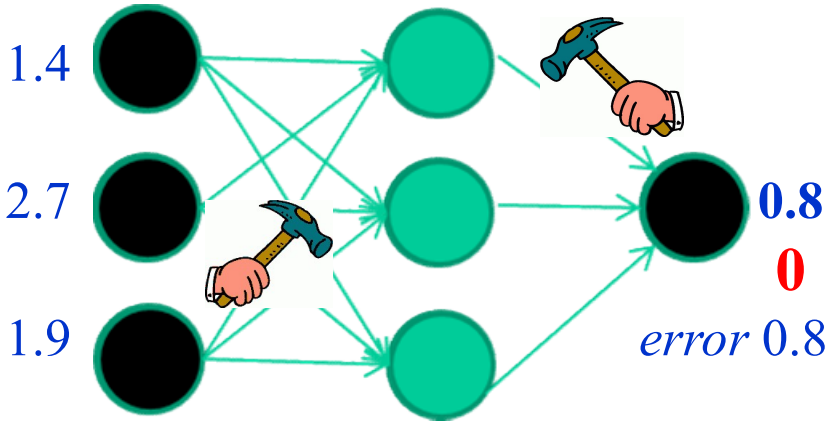
**Compare with target output**



*Training data*

| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc … | | | |

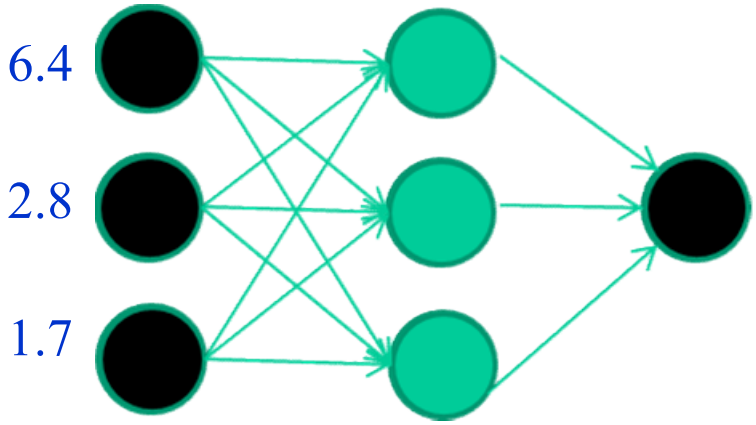
**Adjust weights based on error**



*Training data*

| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc … | | | |

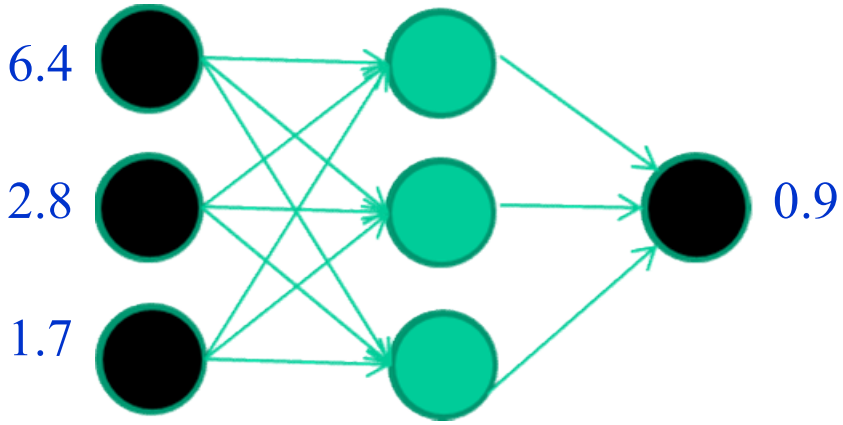
**Present a training pattern**



*Training data*

| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc … | | | |

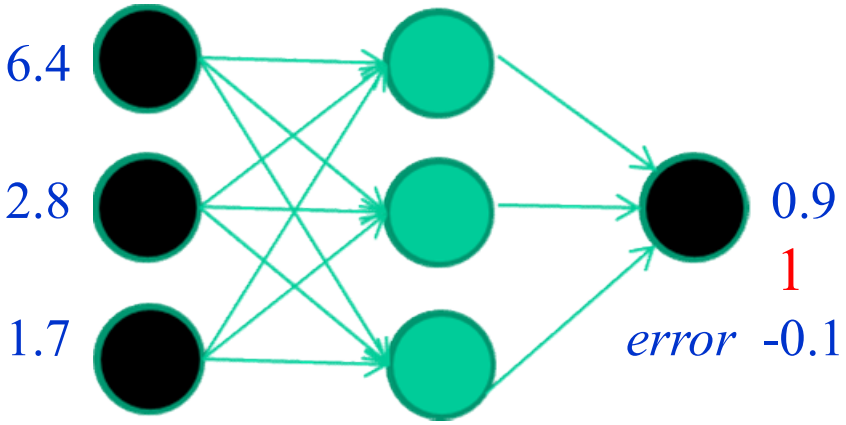
**Feed it through to get output**



*Training data*

| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc … | | | |

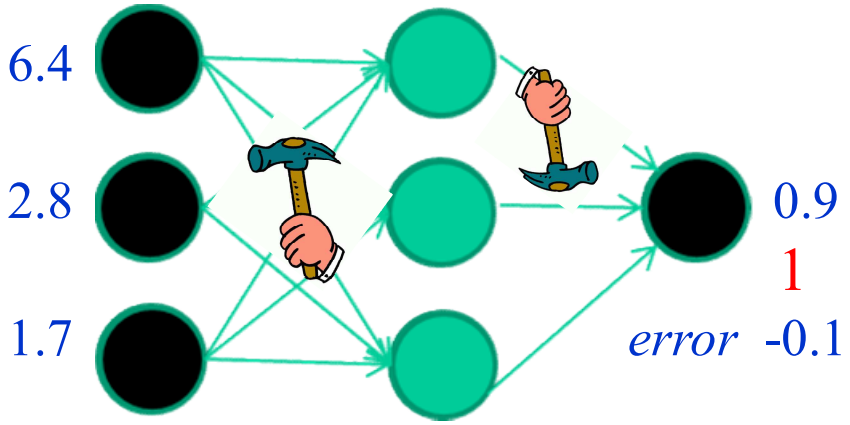
**Compare with target output**



*Training data*

| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc … | | | |

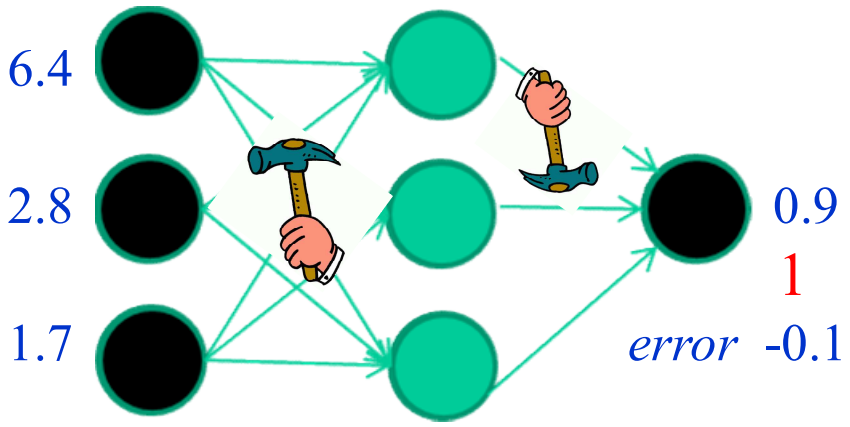
**Adjust weights based on error**



*Training data*

| ***Fields*** | | | ***class*** |
|---|---|---|---|
| 1.4 | 2.7 | 1.9 | 0 |
| 3.8 | 3.4 | 3.2 | 0 |
| 6.4 | 2.8 | 1.7 | 1 |
| 4.1 | 0.1 | 0.2 | 0 |
| etc … | | | |

**And so on ….**

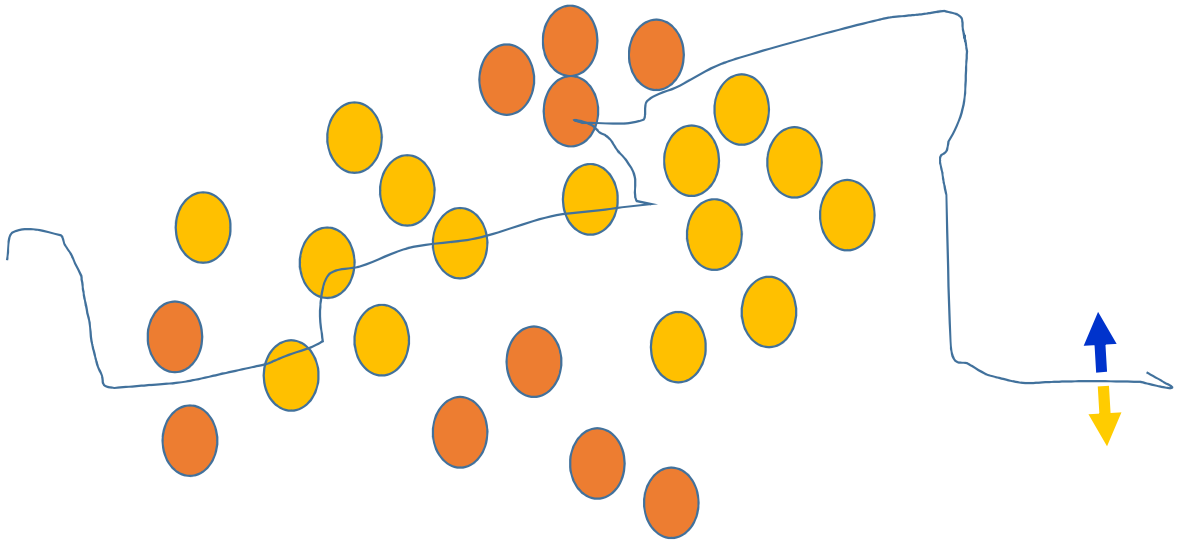



**Repeat this thousands, maybe millions of times – each time taking a random training instance, and making slight weight adjustments**

*Algorithms for weight adjustment are designed to make changes that will reduce the error*

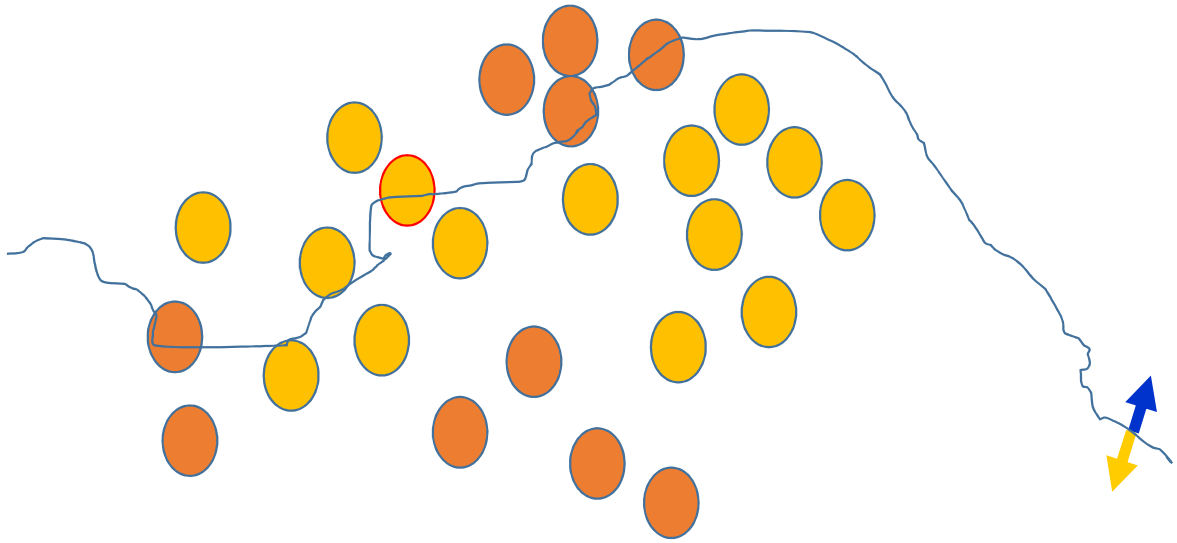
# The decision boundary perspective…

**Initial random weights**



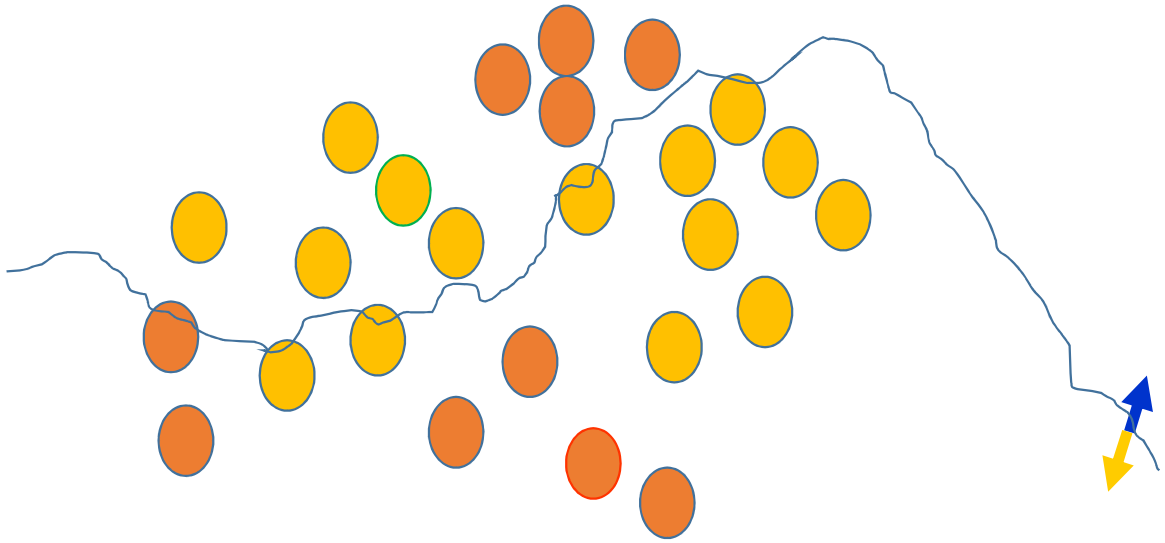
# The decision boundary perspective…

**Present a training instance / adjust the weights**



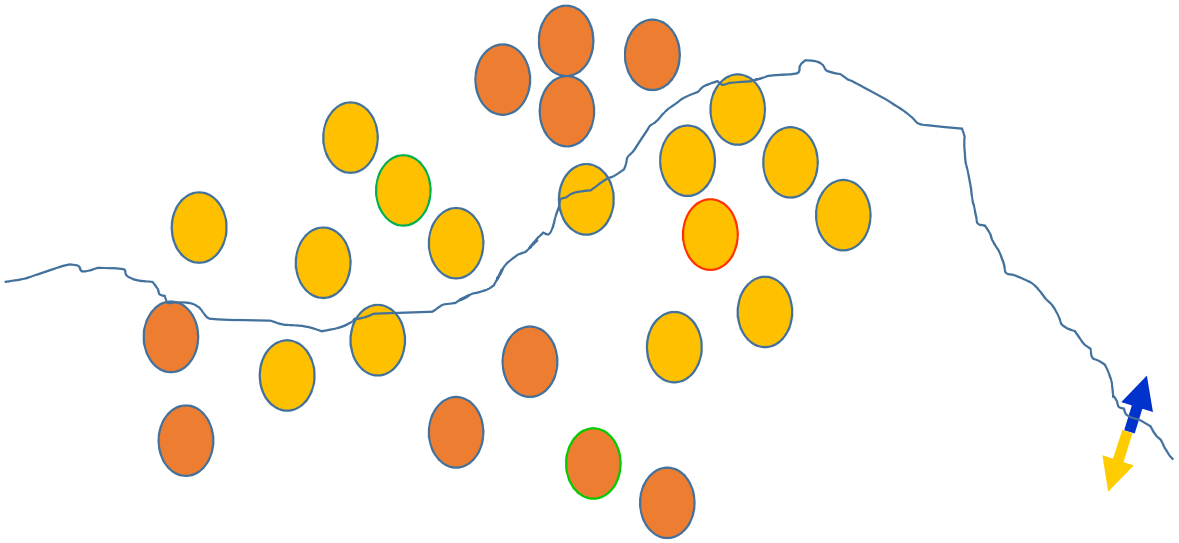
# The decision boundary perspective…

**Present a training instance / adjust the weights**



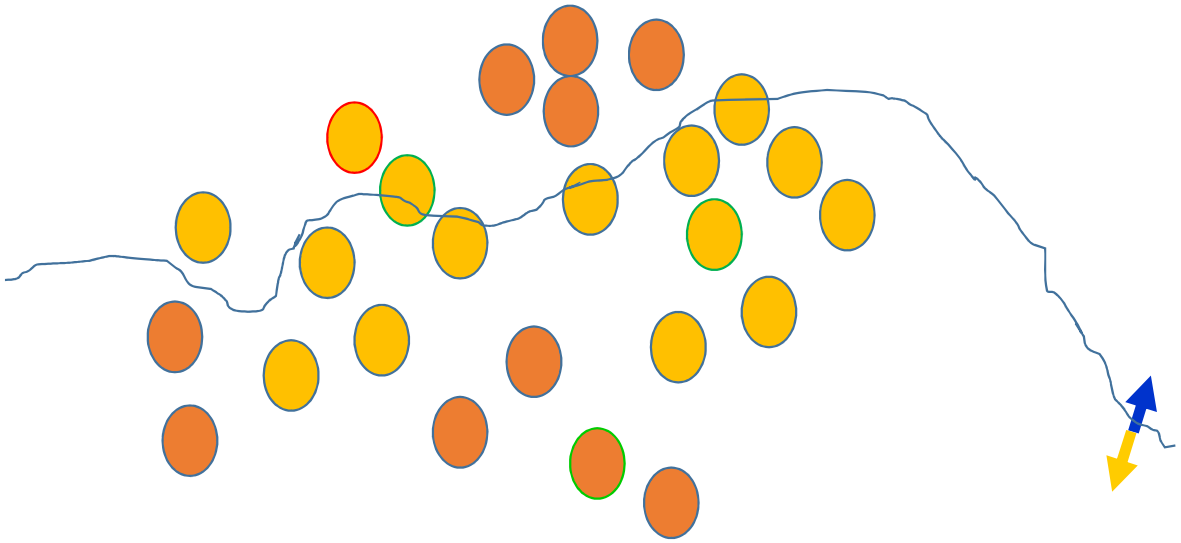
# The decision boundary perspective…

**Present a training instance / adjust the weights**



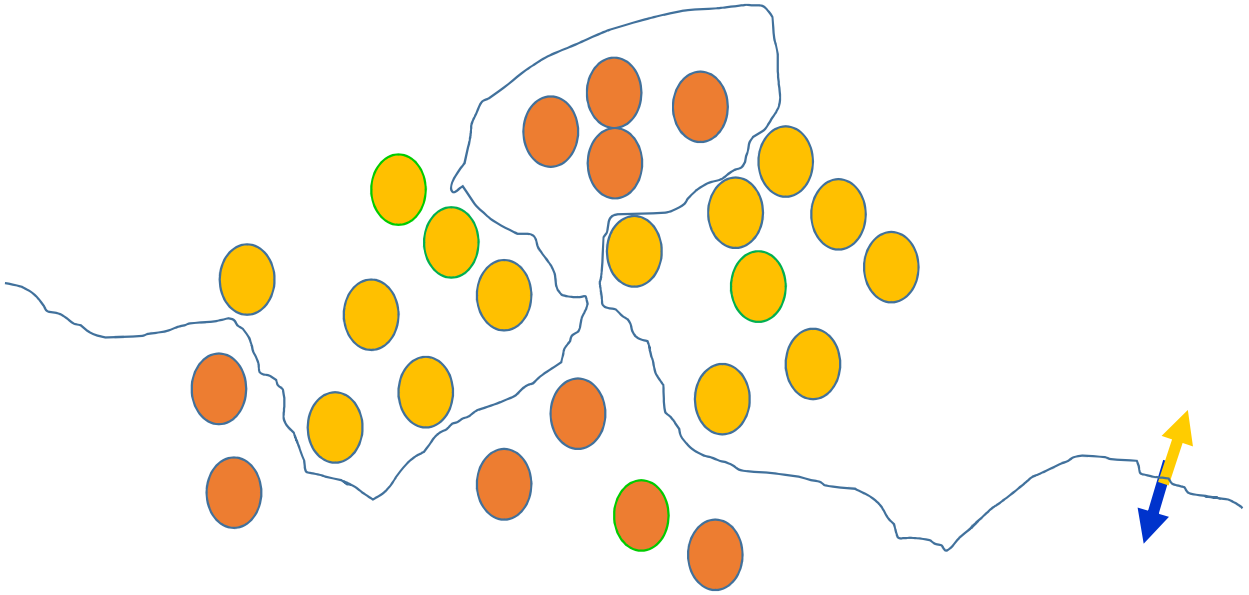
# The decision boundary perspective…

**Present a training instance / adjust the weights**



# The decision boundary perspective…

**Eventually ….**

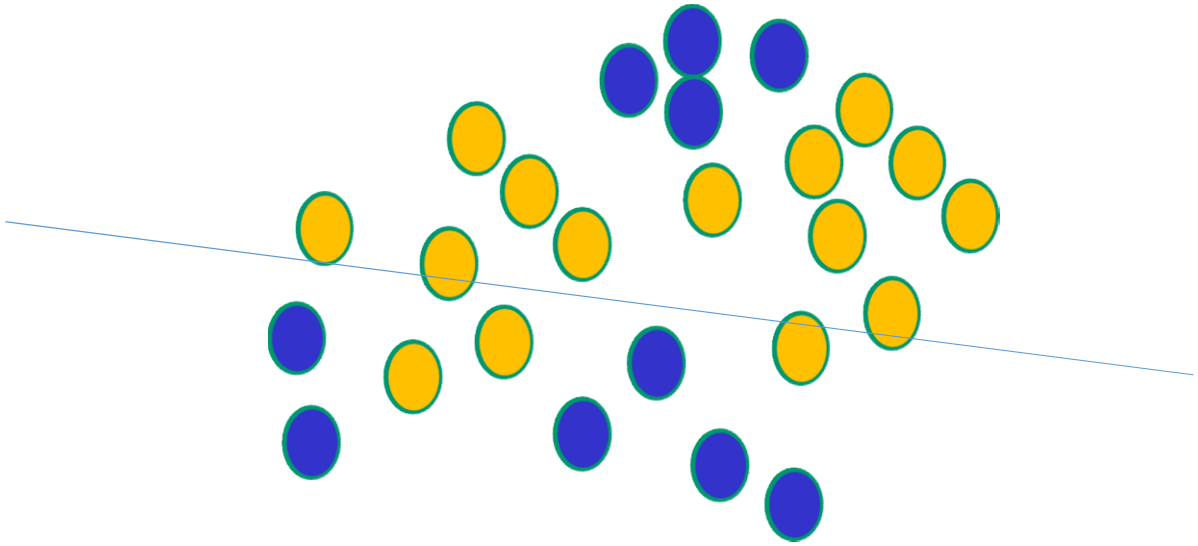


# The point we are trying to make here

- Weight-learning algorithms for NNs are dumb
- They work by making thousands and thousands of tiny adjustments, each making the network work better
- Eventually the cumulation of all these bits tends to be good enough to learn effective classifiers for many real applications

# Some other ‘by the way’ points

If $f(x)$ is linear, the NN can **only** draw straight decision boundaries (even if there are many layers of units)

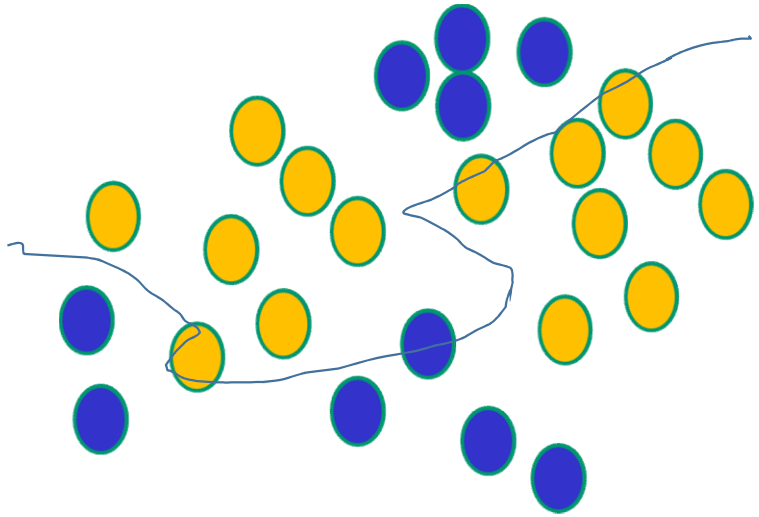


# Some other ‘by the way’ points

NNs use nonlinear *f*(x) so they

can draw complex boundaries,

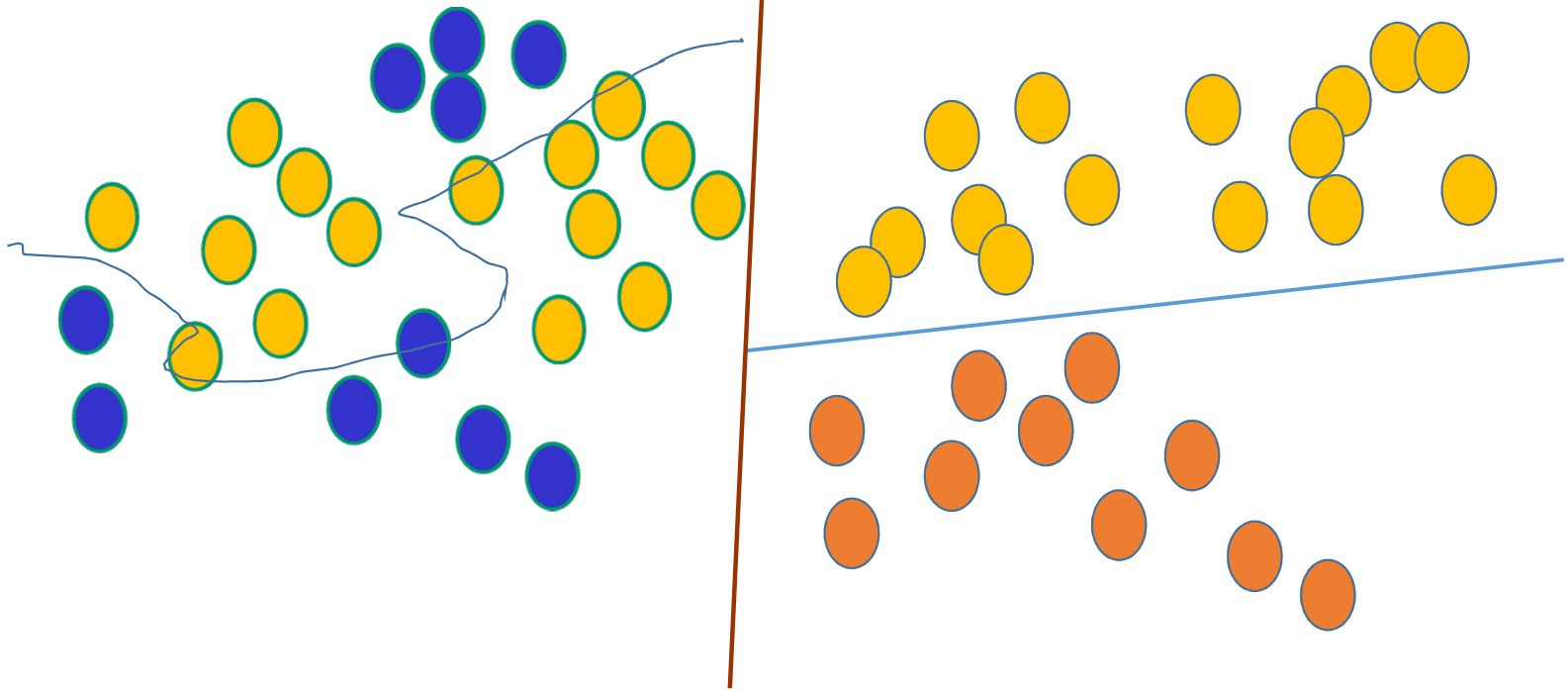
but keep the data unchanged

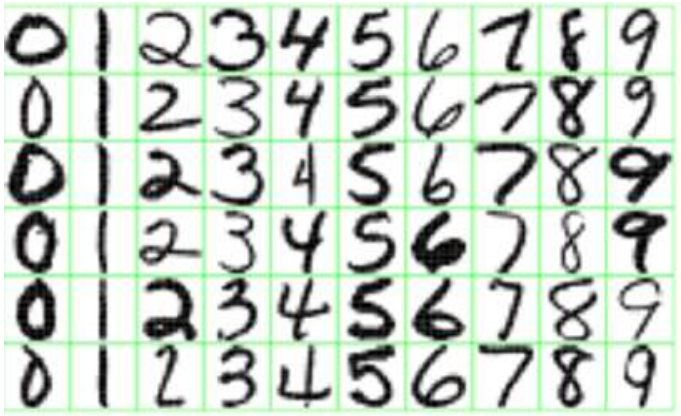


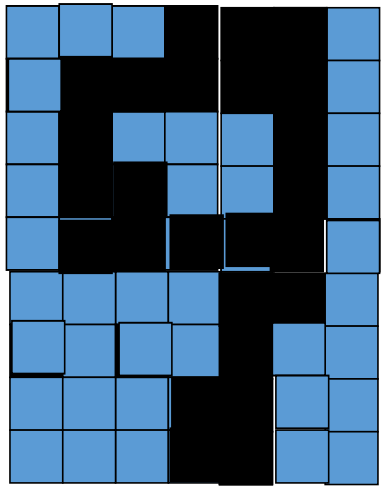
# Some other ‘by the way’ points

NNs use nonlinear *f*(x) so they can draw complex boundaries, but keep the data unchanged

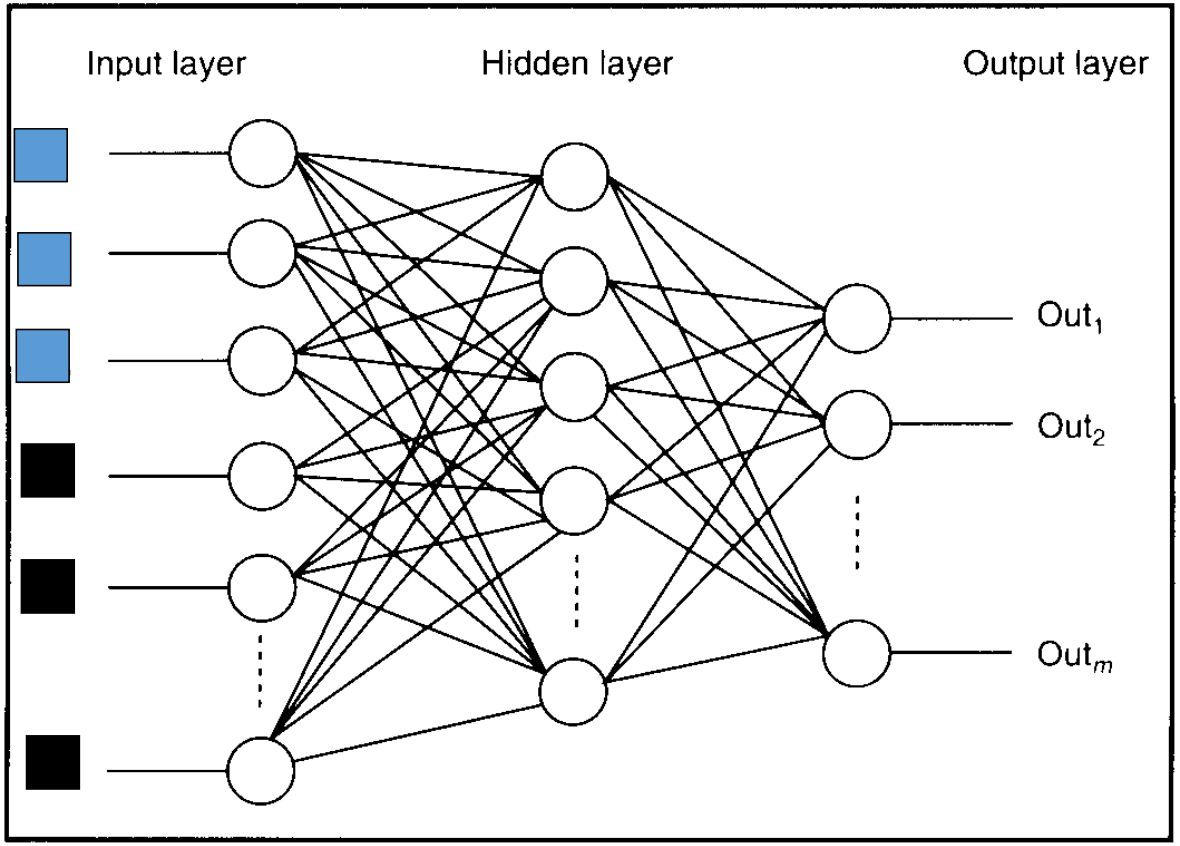
SVMs only draw straight lines, but they transform the data first in a way that makes that OK

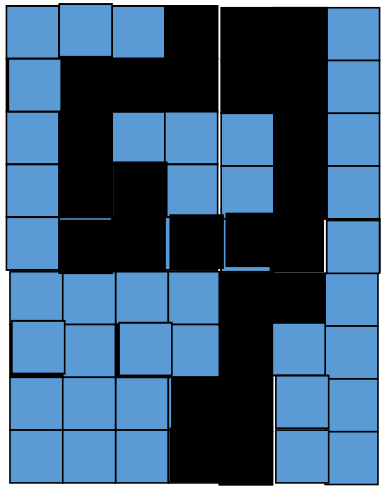




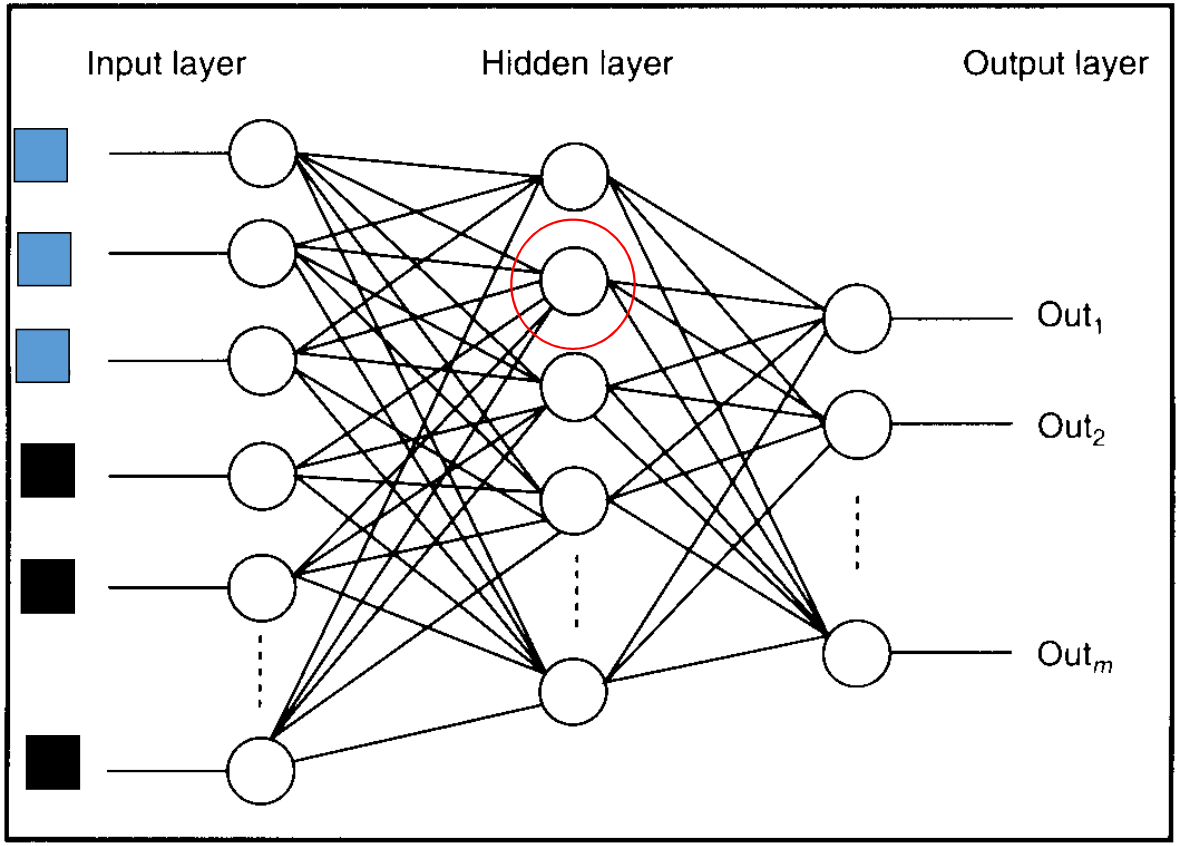


# Feature detectors

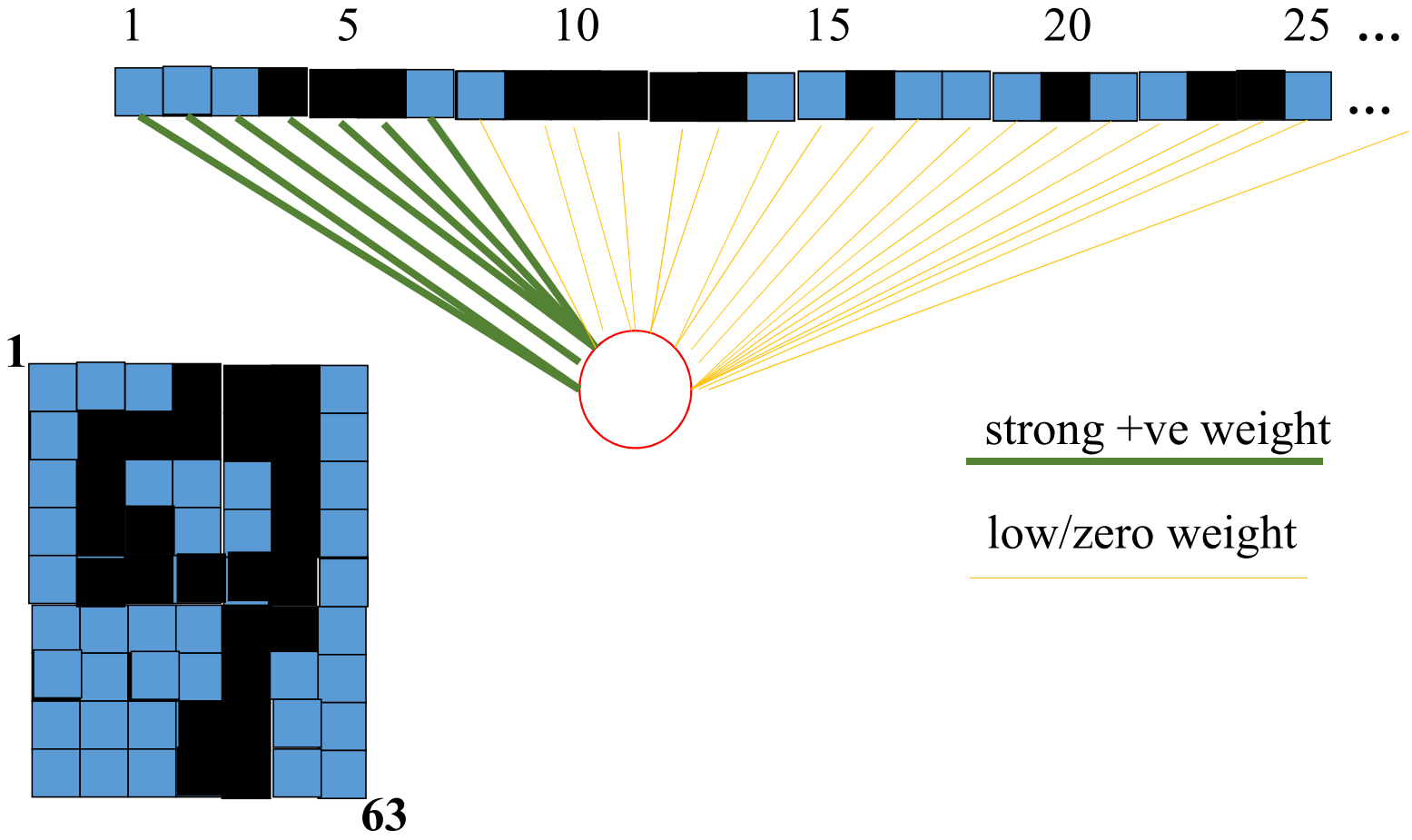




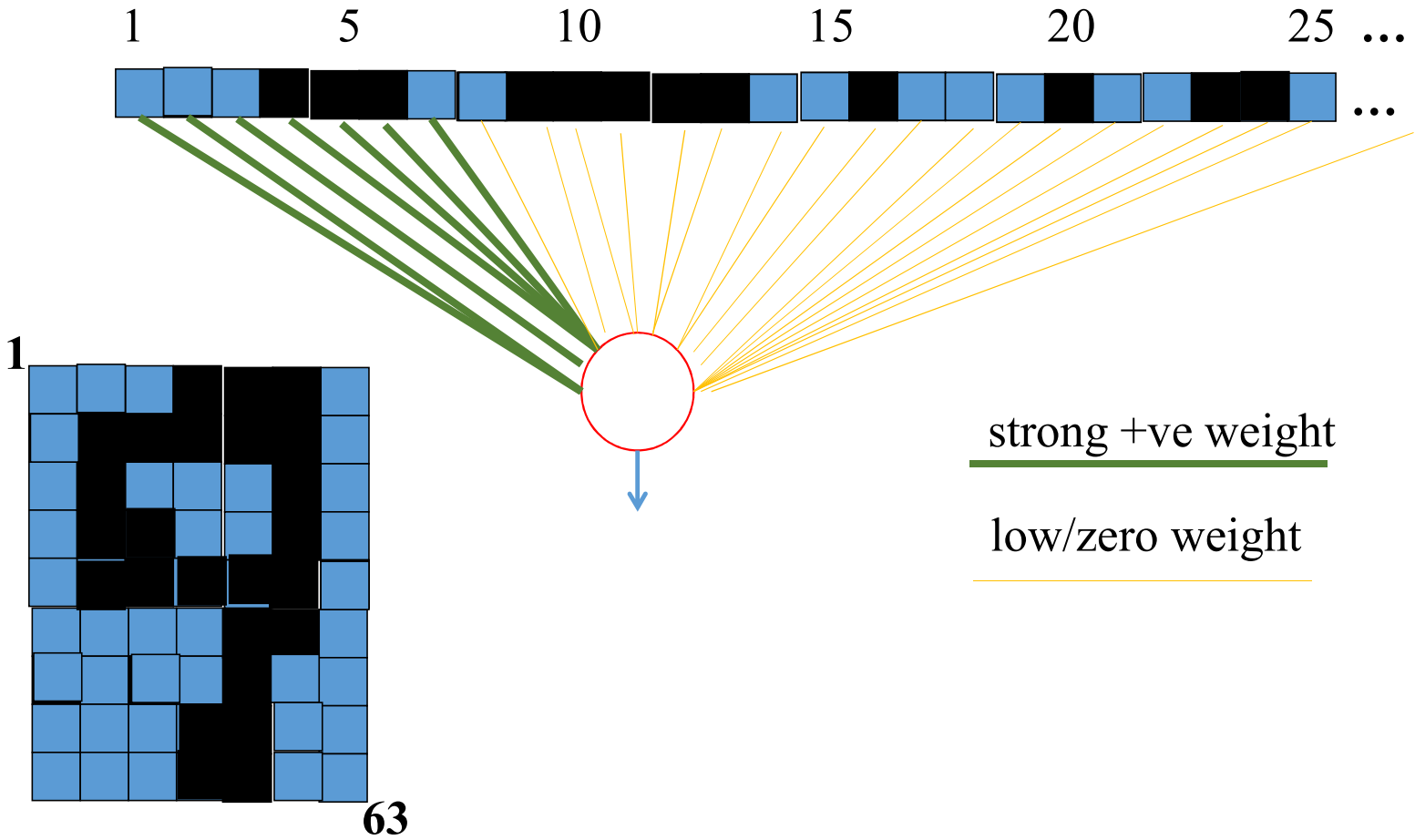
*what is this unit doing?*



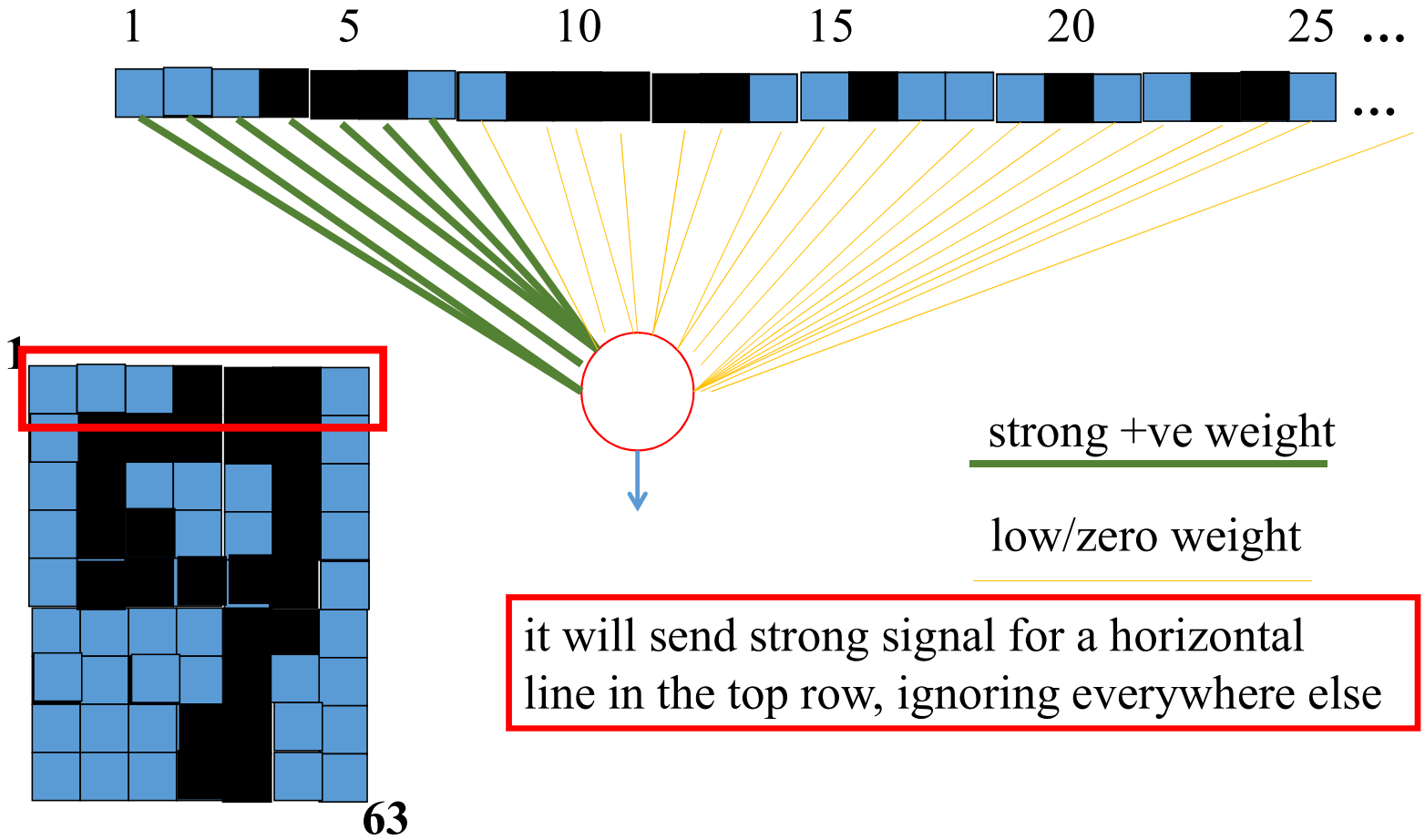
# Hidden layer units become *self-organised feature detectors*



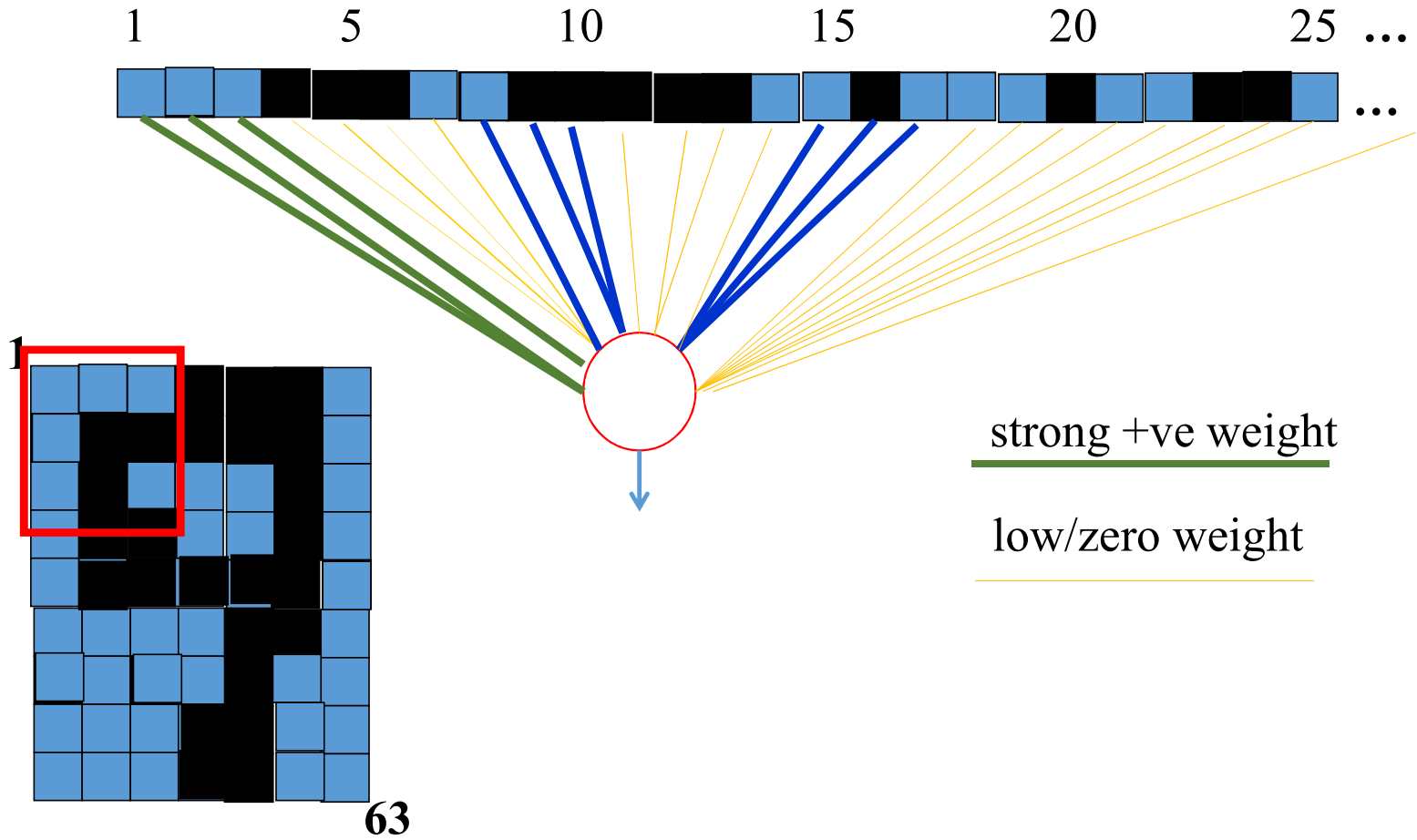
# What does this unit detect?



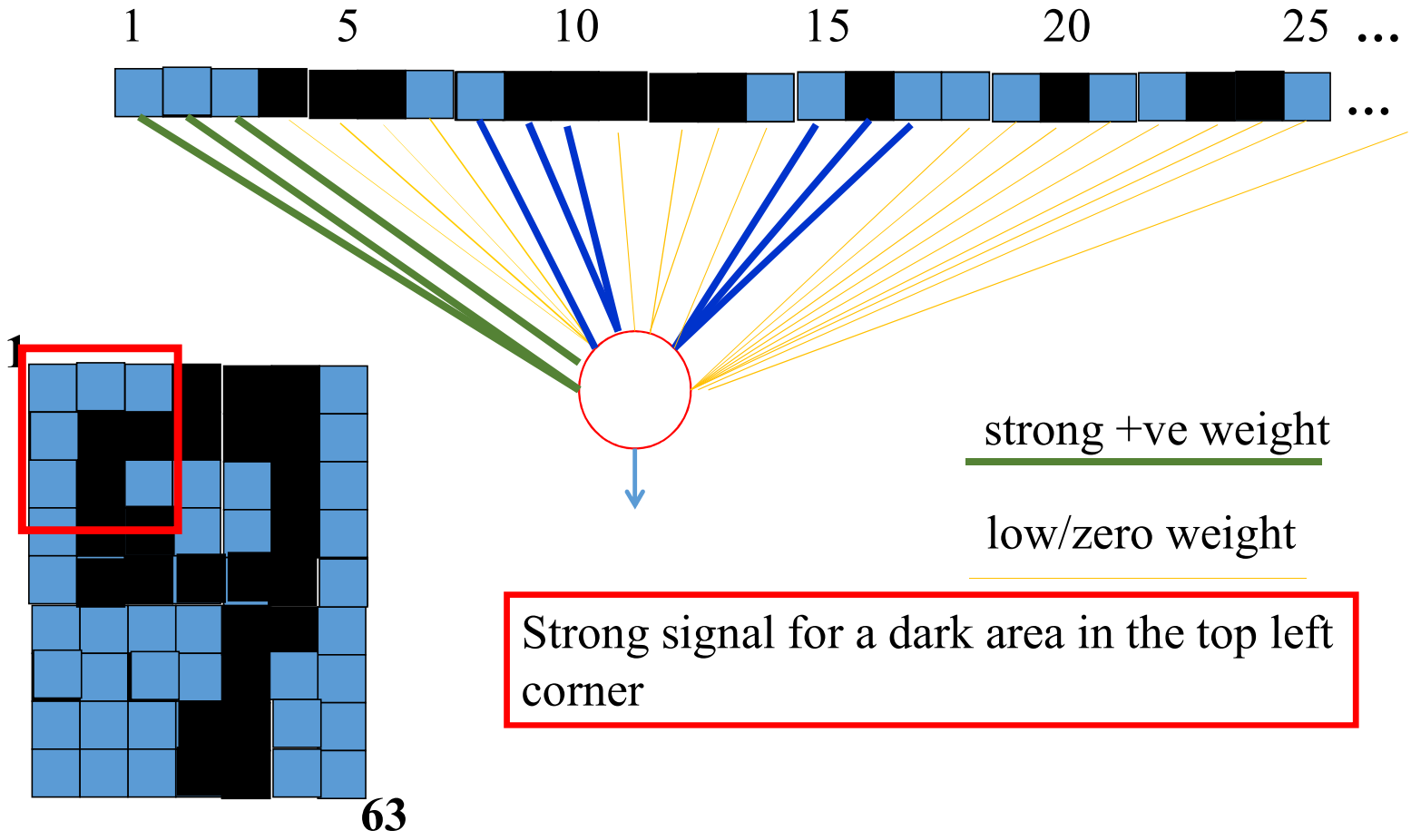
# What does this unit detect?

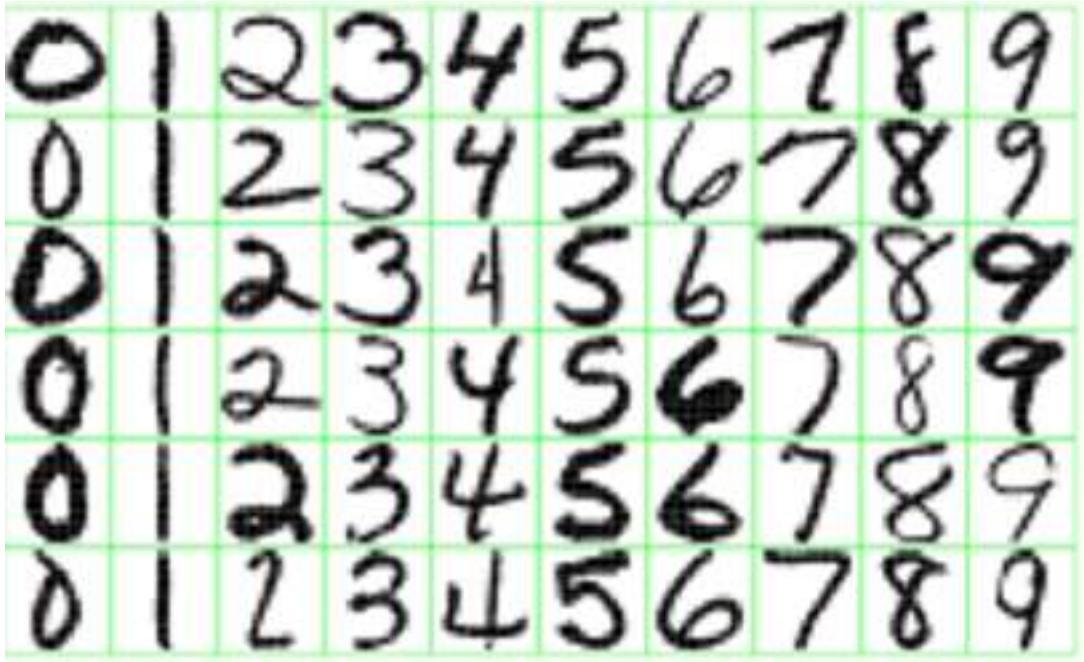


# What does this unit detect?

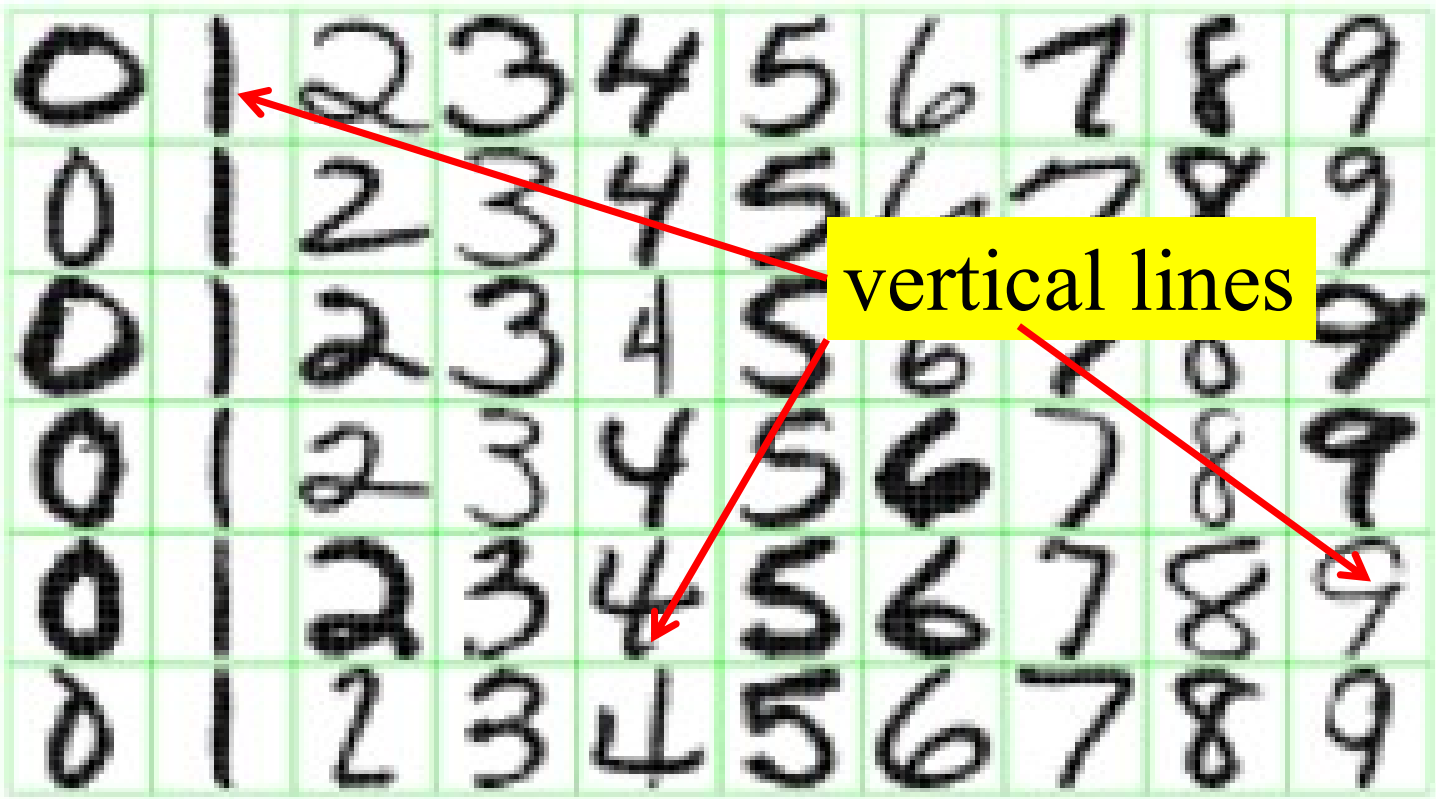


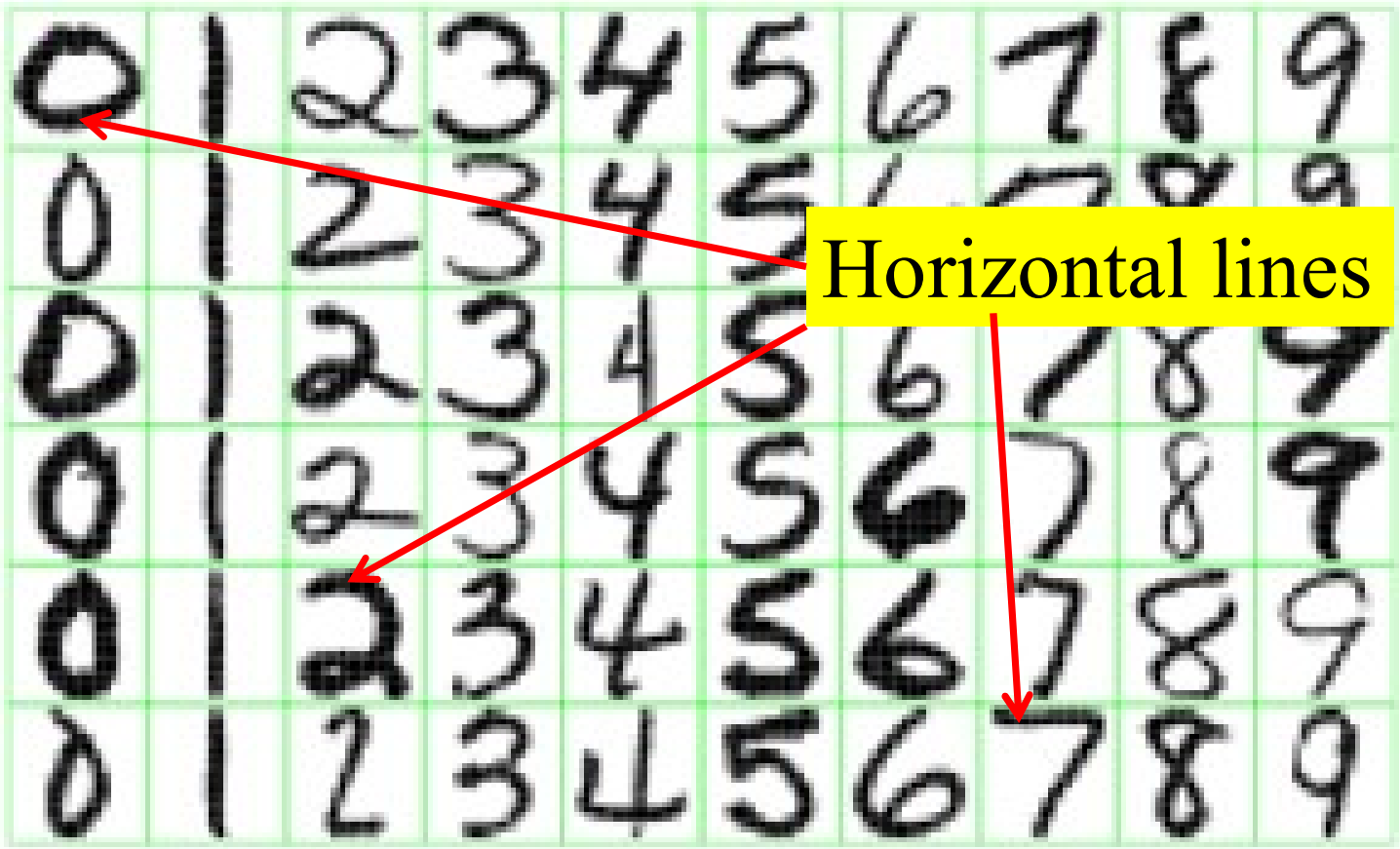
# What does this unit detect?

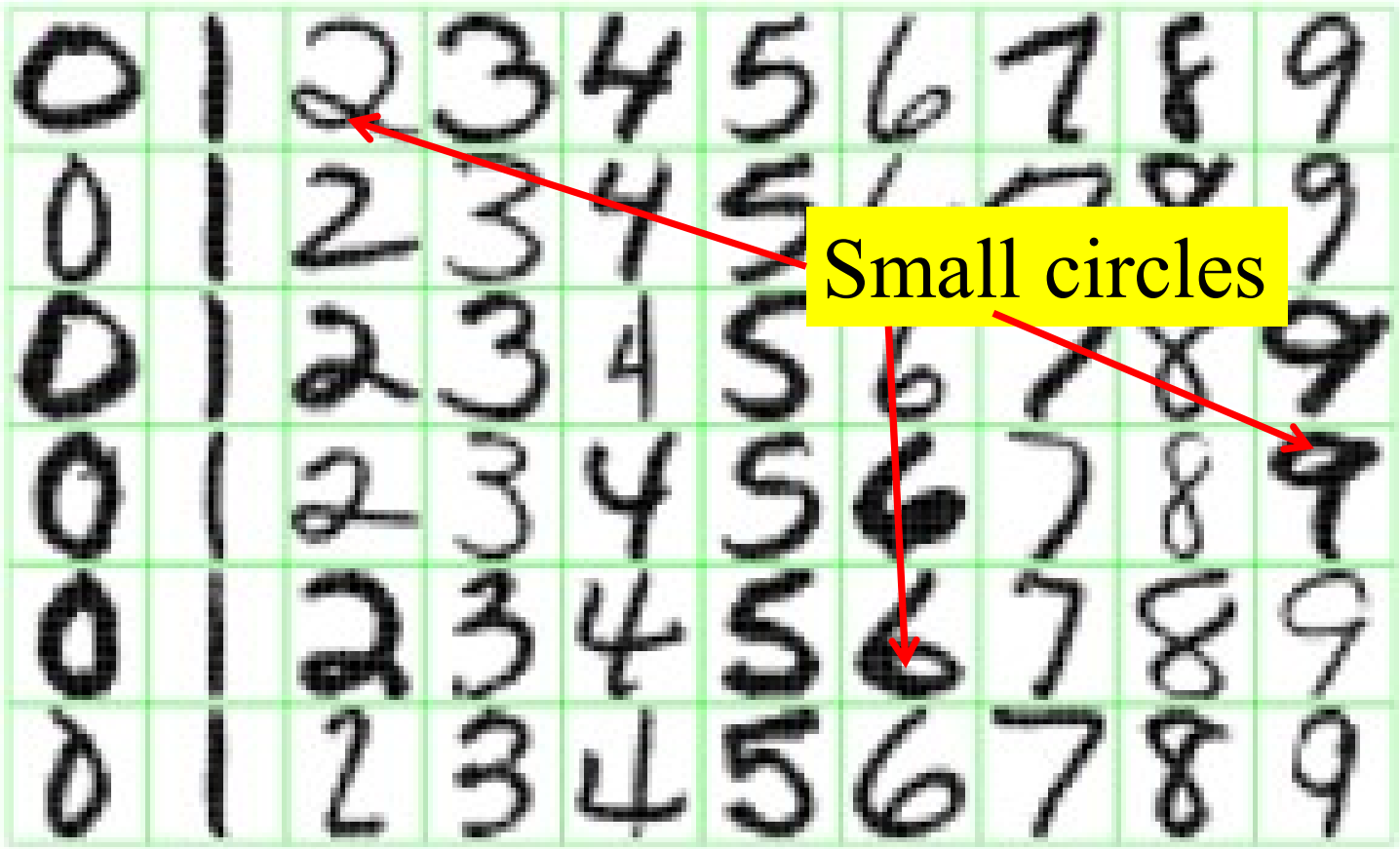




What features might you expect a good NN to learn, when trained with data like this?


vertical lines

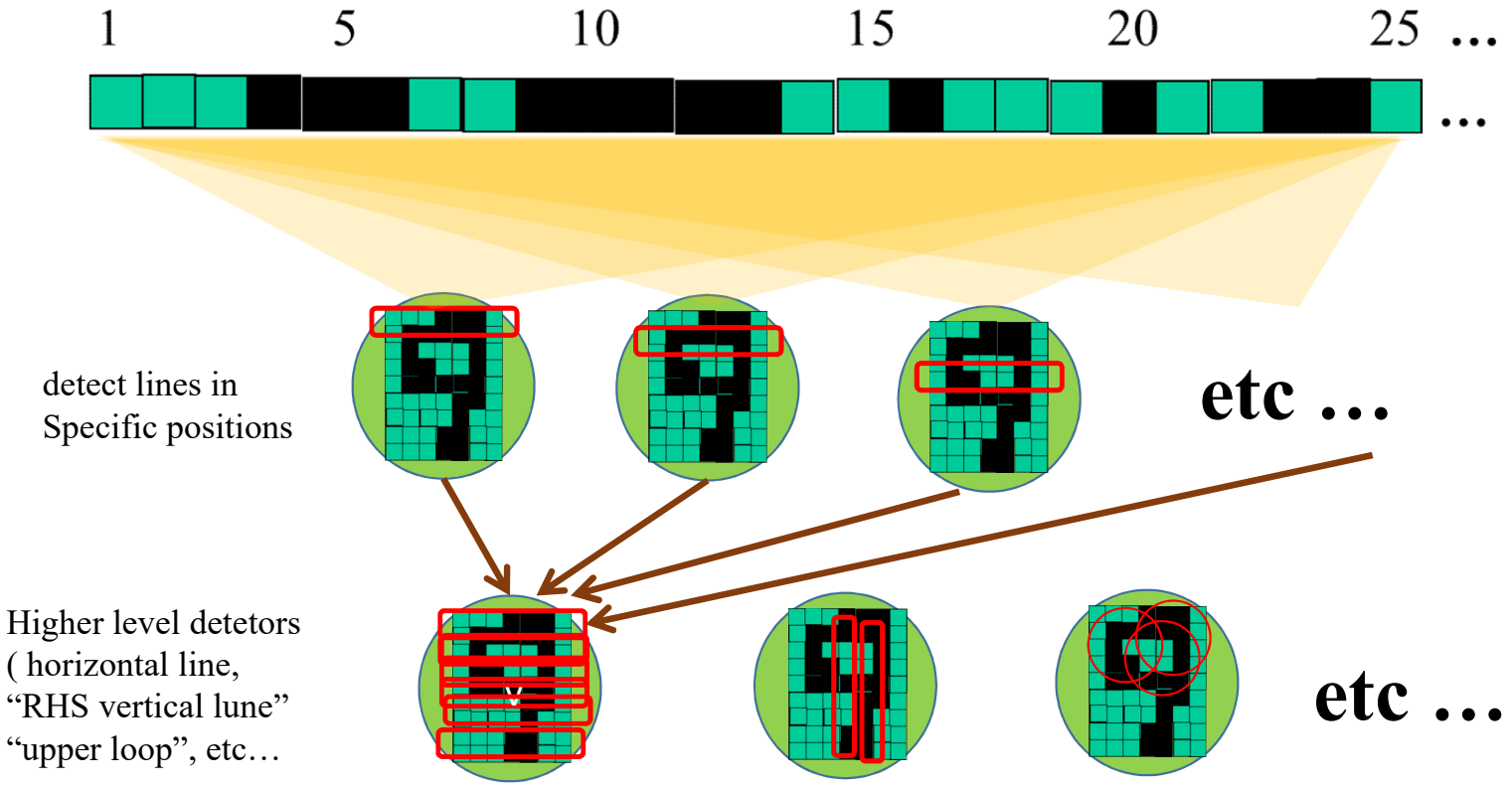

Horizontal lines


Small circles

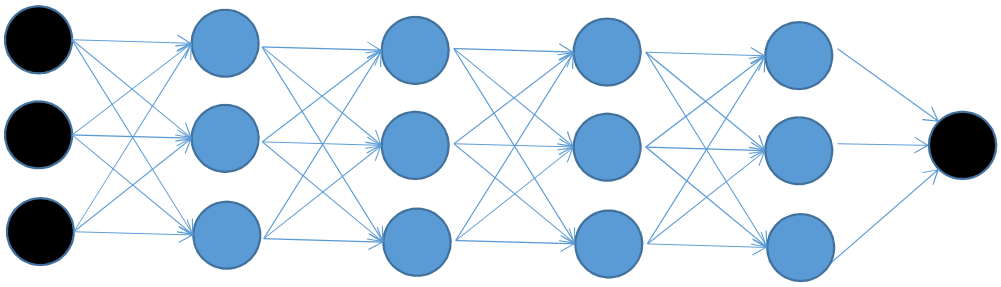

1

But what about position invariance ???
our example unit detectors were tied to specific parts of the image

successive layers can learn higher-level features …

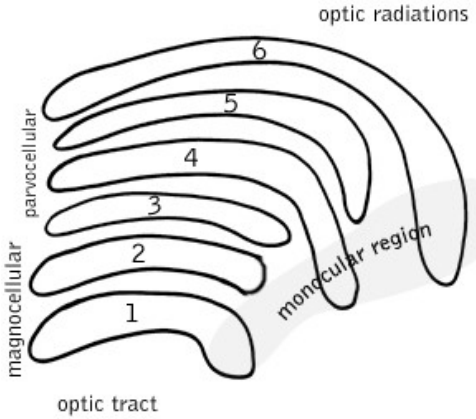


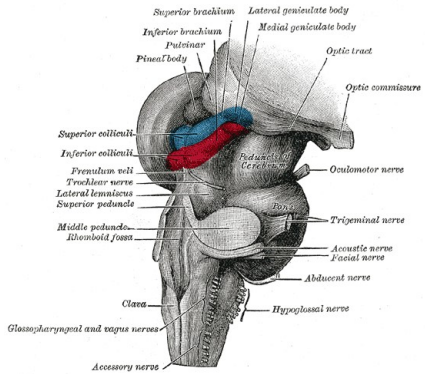
# So: *multiple layers make sense*



# So: *multiple layers make sense*

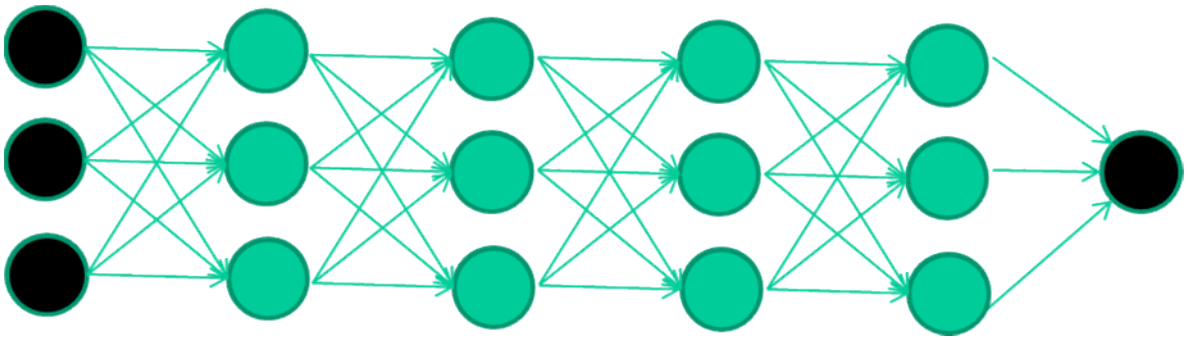
Our brain works that way



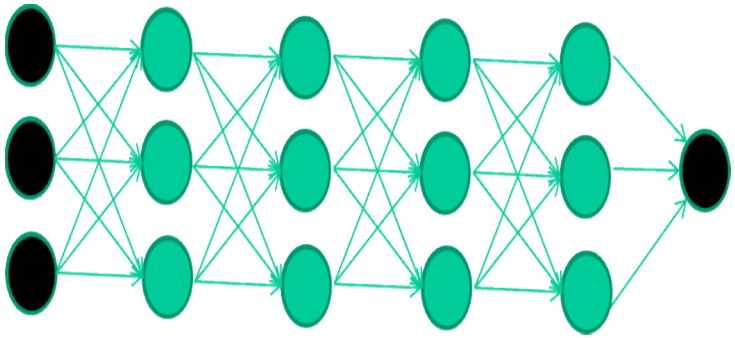



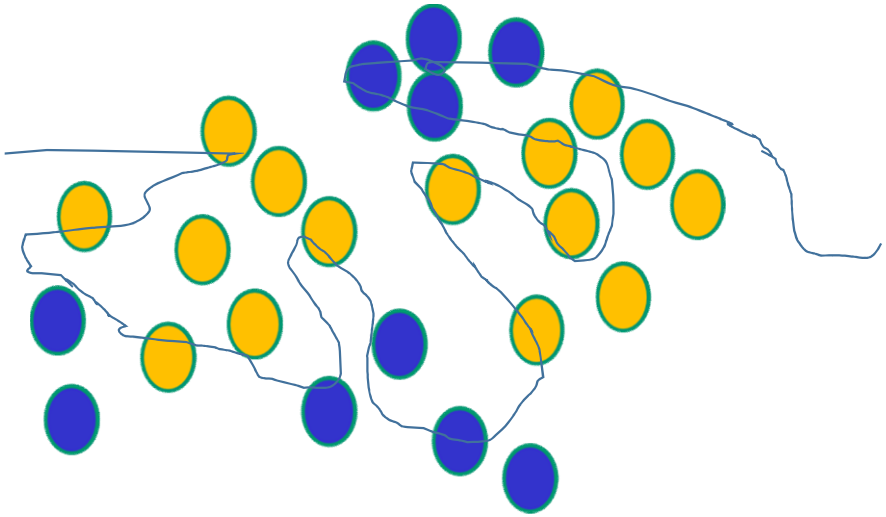
# So: *multiple layers make sense*

Many-layer neural network architectures should be capable of learning the true underlying features and 'feature logic', and therefore generalise very well …



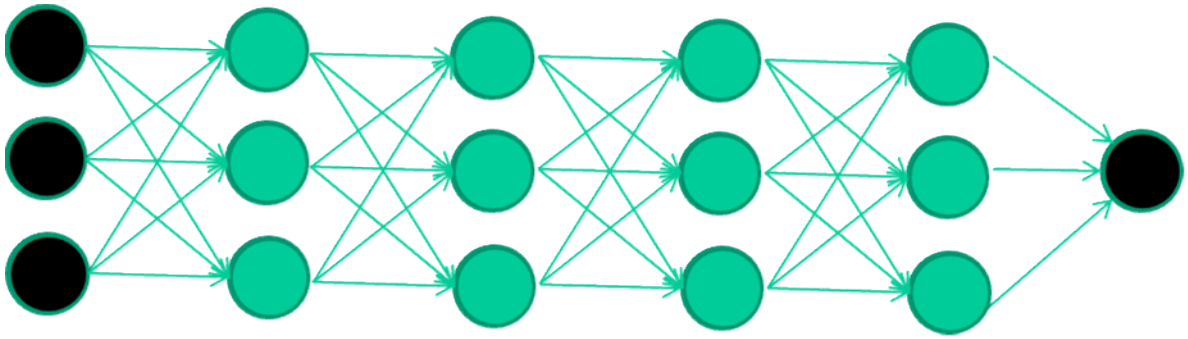
But, until very recently, our weight-learning algorithms simply did not work on multi-layer architectures



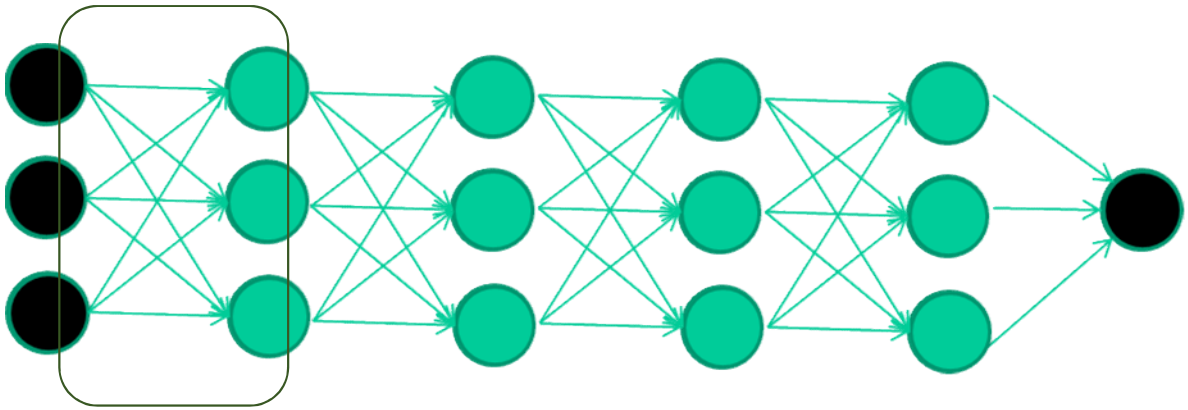


Along Came Deep Learning…….

# The new way to train multi-layer NNs…

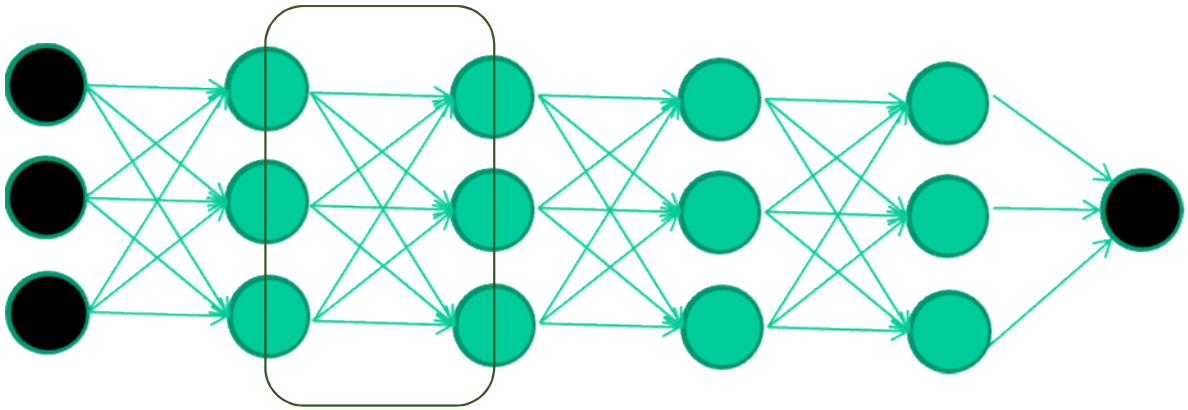


# The new way to train multi-layer NNs…



Train **this** layer first

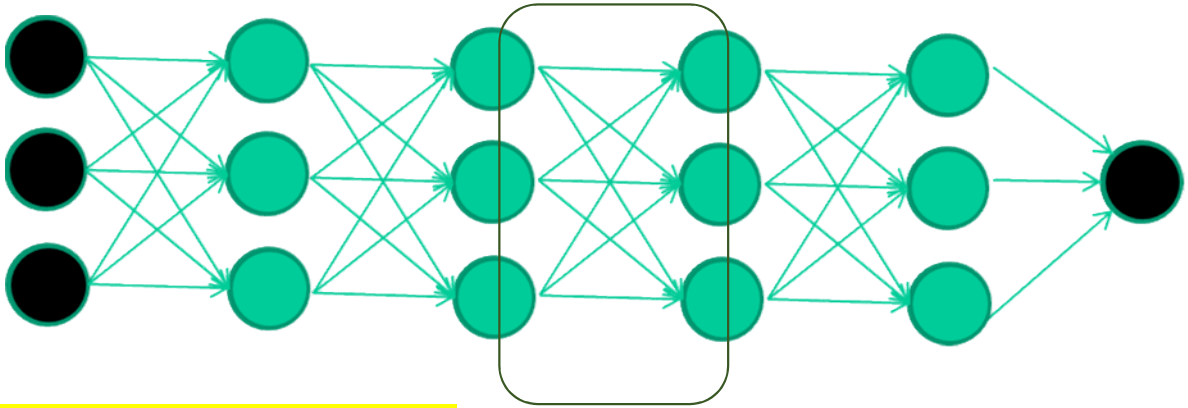
# The new way to train multi-layer NNs…



Train **this** layer first

then **this** layer

# The new way to train multi-layer NNs…

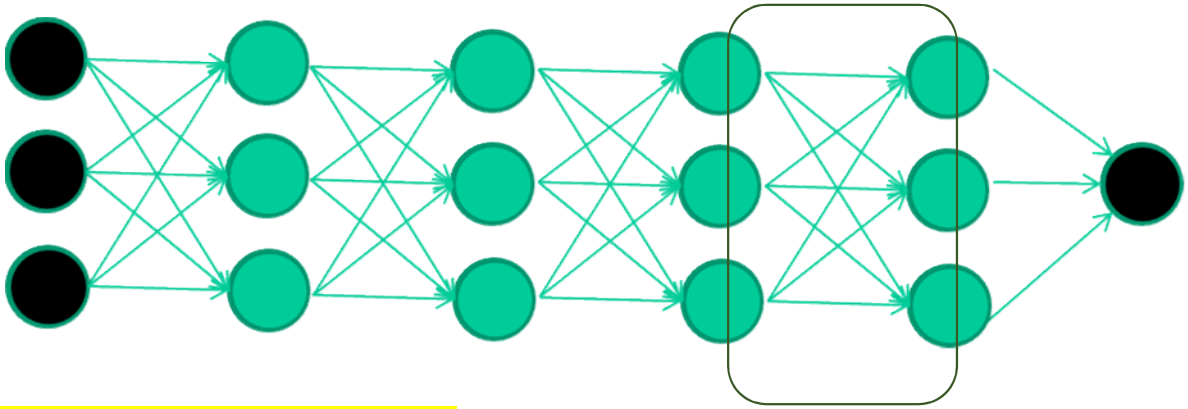


Train **this** layer first

then **this** layer

then **this** layer

# The new way to train multi-layer NNs…



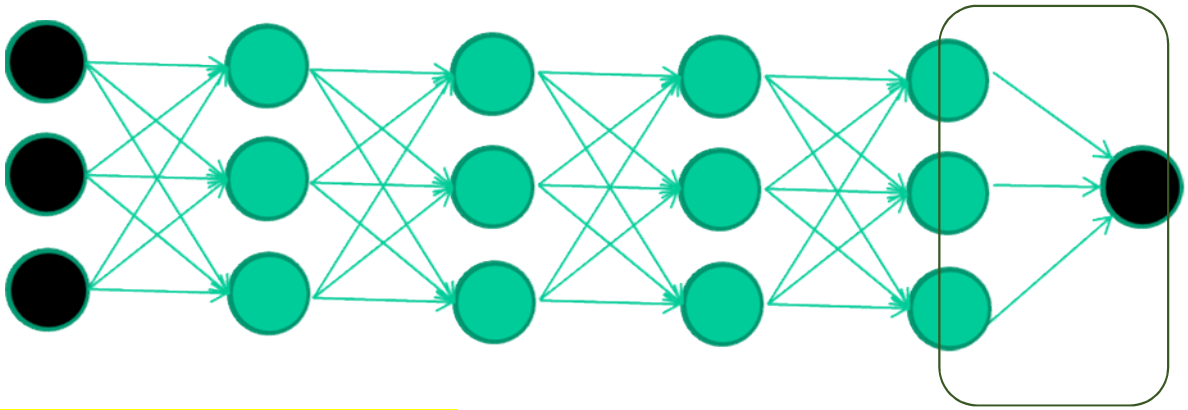
Train **this** layer first

then **this** layer

then **this** layer

then **this** layer

# The new way to train multi-layer NNs…



Train **this** layer first

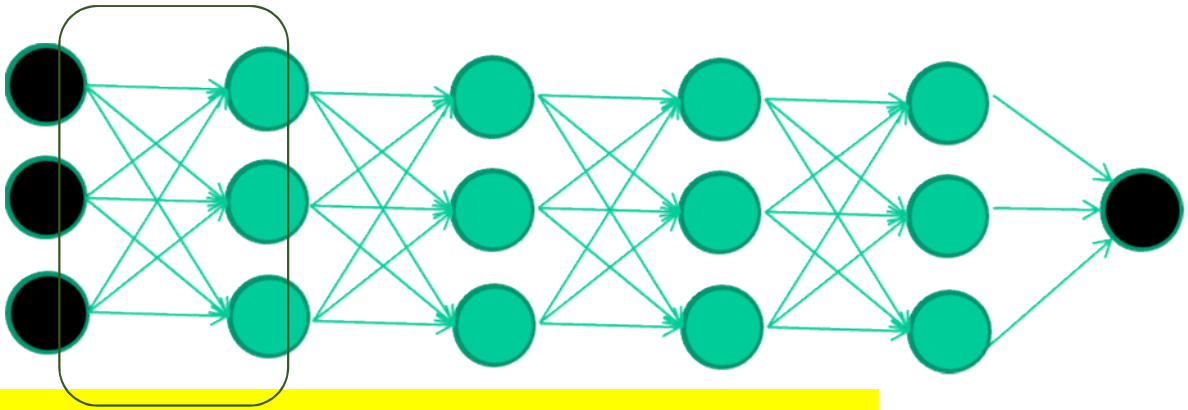
then **this** layer

then **this** layer

then **this** layer

finally **this** layer

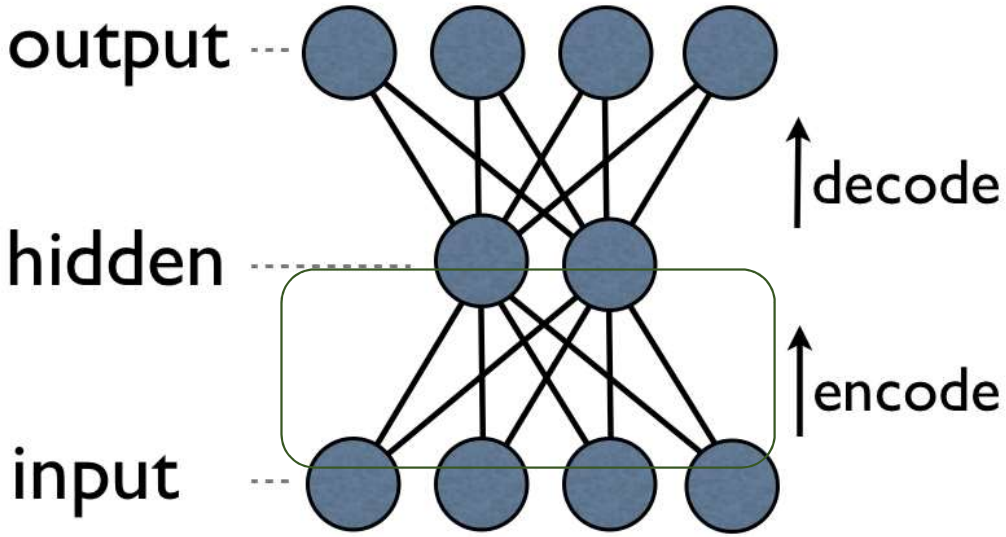
# The new way to train multi-layer NNs…



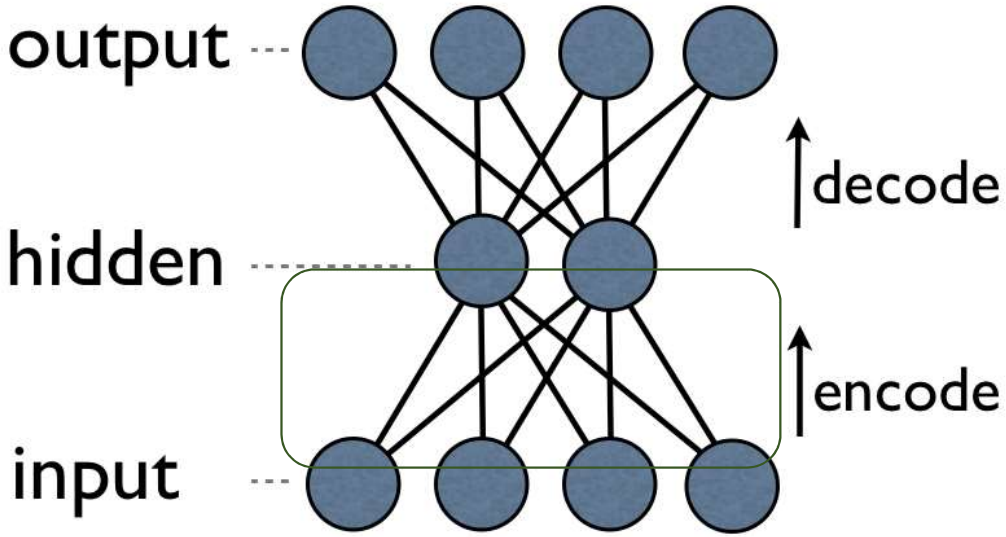
*EACH of the (non-output) layers is trained to be an* ***auto-encoder***

*Basically, it is forced to learn good features that describe what comes from the previous layer*

an auto-encoder is trained, with an absolutely standard weight-adjustment algorithm to reproduce the input

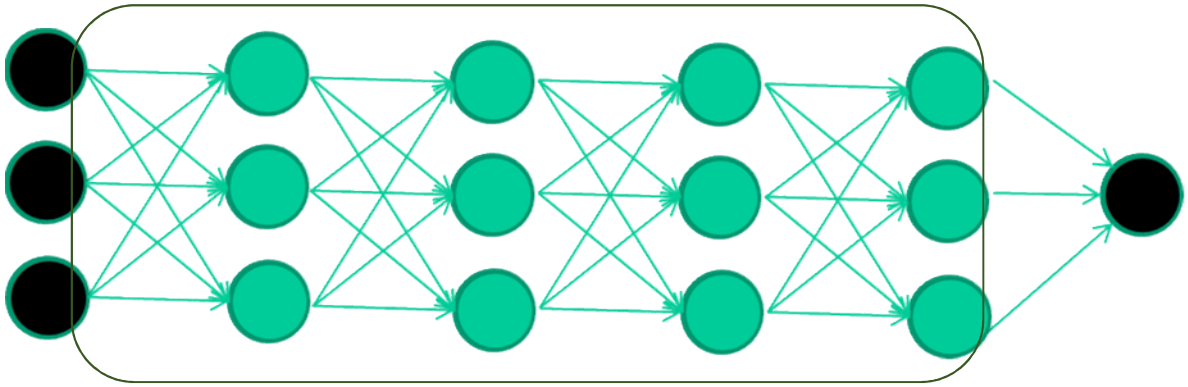


an auto-encoder is trained, with an absolutely standard weight-adjustment algorithm to reproduce the input

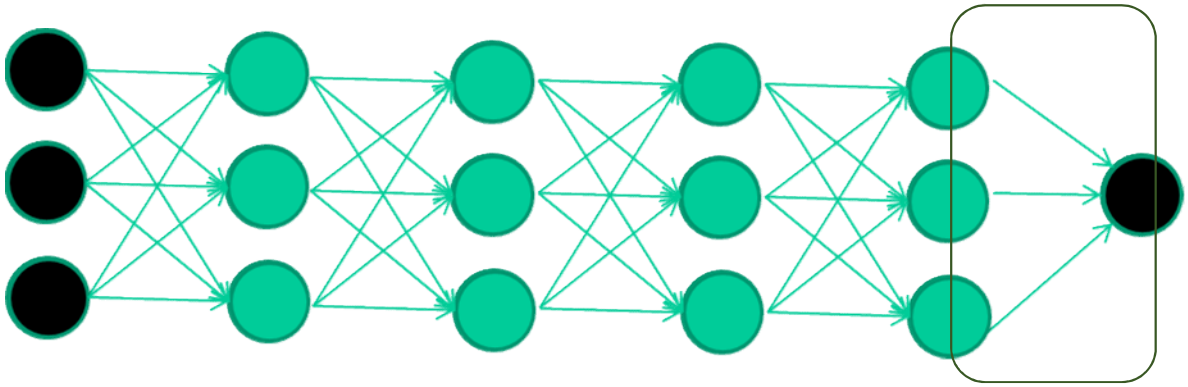



By making this happen with (many) fewer units than the inputs, this forces the 'hidden layer' units to become good feature detectors

intermediate layers are each trained to be auto encoders (or similar)



Final layer trained to predict class based on outputs from previous layers



# And That's the basic idea behind deep learning

- There are many types of deep learning
- Different kinds of autoencoder, variations on architectures and training algorithms etc…
- It's a fast growing area …

# Common DNNs

- Deep Convolutional Neural Network (DCNN)
  - To extract representation from images
- Recurrent Neural Network (RNN)
  - To extract representation from sequential data
- Deep Belief Neural Network (DBN)
  - To extract hierarchical representation from a dataset
- Deep Reinforcement Learning (DQN)
  - To prescribe how agents should act in an environment in order to maximize future cumulative reward (e.g., a game score)

**We cover DCNN & RNN today**

# Comparing Representations

## Vector representation

taco = $[17.32, 82.9, -4.6, 7.2]$

- Vectors have a similarity score.
  A taco is not a burrito but similar.
- Vectors have internal structure [Mikolov et al., 2013].
  Italy – Rome = France – Paris
  King – Queen = Man – Woman
- Vectors are grounded in experience.
- Meaning relative to predictions.

## Symbolic representation

taco = $taco$

- Symbols can be the same or not.
- A taco is just as different from a burrito as a Toyota.
- Symbols have no structure.
- Symbols are arbitrarily assigned.
- Meaning relative to other symbols.

# Common Open-Source DL Frameworks

**Non-symbolic**

Torch, Caffe

Manual optimization

Memory intensive

**Symbolic**

CNTK, MXNET, Theano, TensorFlow

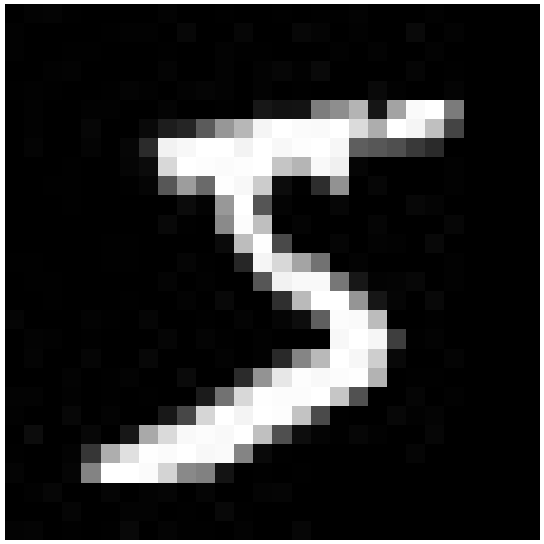
Automatic optimization

Memory reuse

# Deep learning and computer vision

# Vision is hard

Vision is hard because images are big matrices of numbers.



Example from MNIST handwritten digit dataset [LeCun and Cortes, 1998].

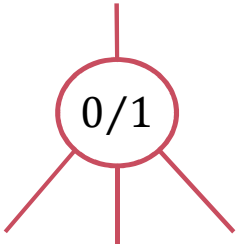
How a computer sees an image

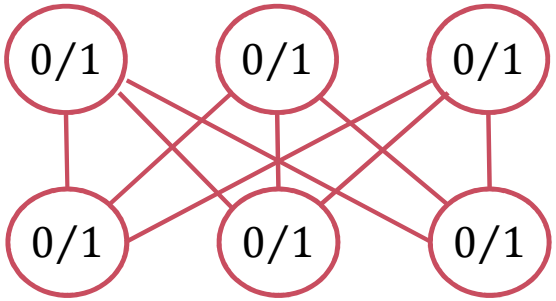
[22, 81, 44, 88, 17, 0, ..., 45]

- Even harder for 3D objects.
- You move a bit, and everything changes.

# RBM: Unsupervised Model

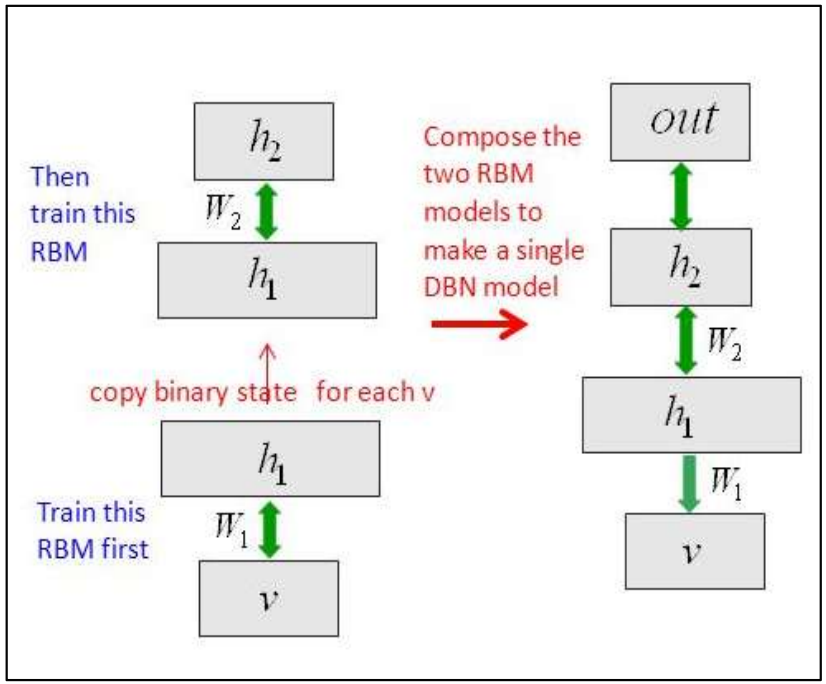
- Stochastic binary neuron
- Probabilistically outputs 0 (turns off) or 1 (turns on) based on the weight of the inputs from on units.



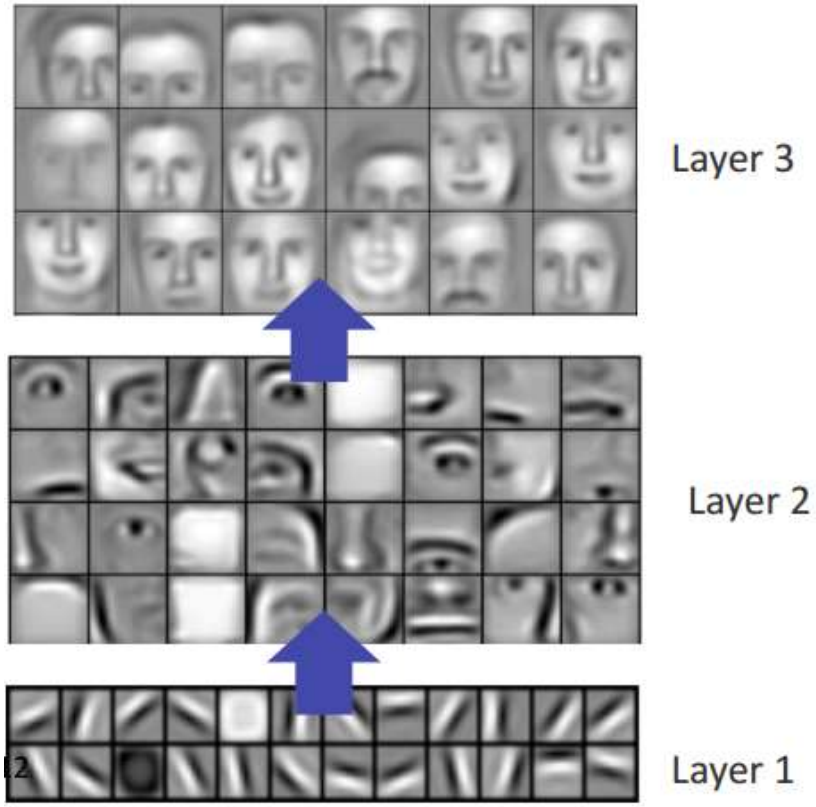

- Limit connections to be from one layer to the next.
- Fast because decisions are made locally.
- Trained in an unsupervised way to reproduce the data.

# Stack up the layers to make a DBN




*Huang et al., Communications ACM 01/2014

# Computer vision, scaling up



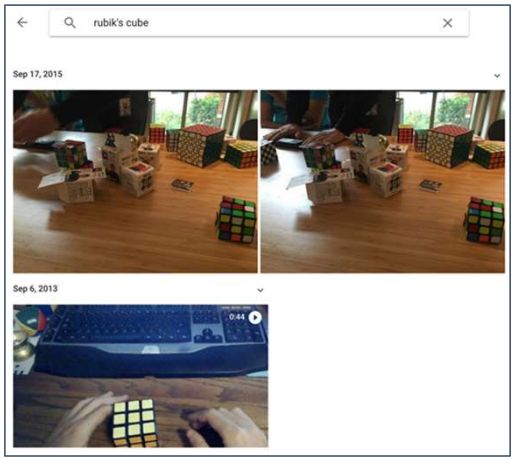

[Lee et al., 2009]

Unsupervised learning was scaled up by Honglak Lee et al. [2009] to learn high-level visual features.

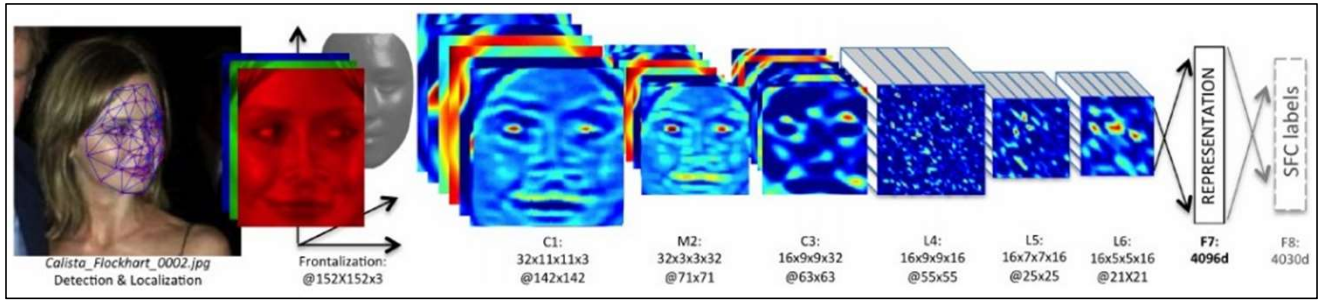
Further scaled up by Quoc Le et al. [2012].

- Used 1,000 machines (16,000 cores) running for 3 days to train 1 billion weights by watching YouTube videos.
- The network learned to identify cats.
- The network wasn't told to look for cats, it naturally learned that cats were integral to online viewing.

# Supervised: ConvNets are everywhere




e.g. Google Photos search




Face Verification, Taigman et al. 2014 (FAIR)

[Goodfellow et al. 2014]

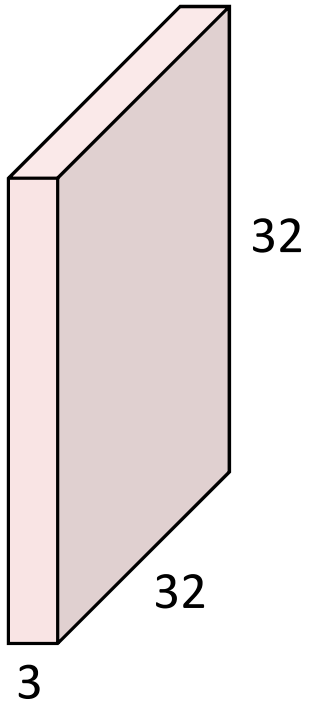

Self-driving cars

*Andrej Karpathy's recent presentation

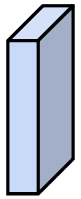
# Convolution Layer

32x32x3 image

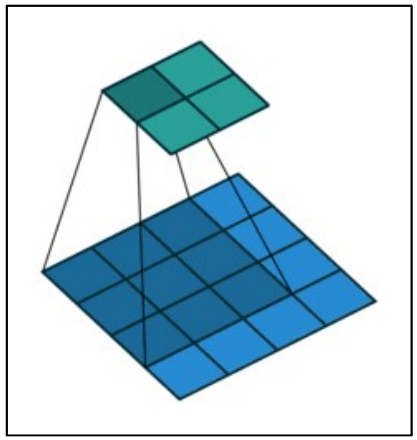
Filters always extend the full depth of the input volume




5x5x3 filter



**Convolve** the filter with the image i.e. “slide over the image spatially, computing dot products”

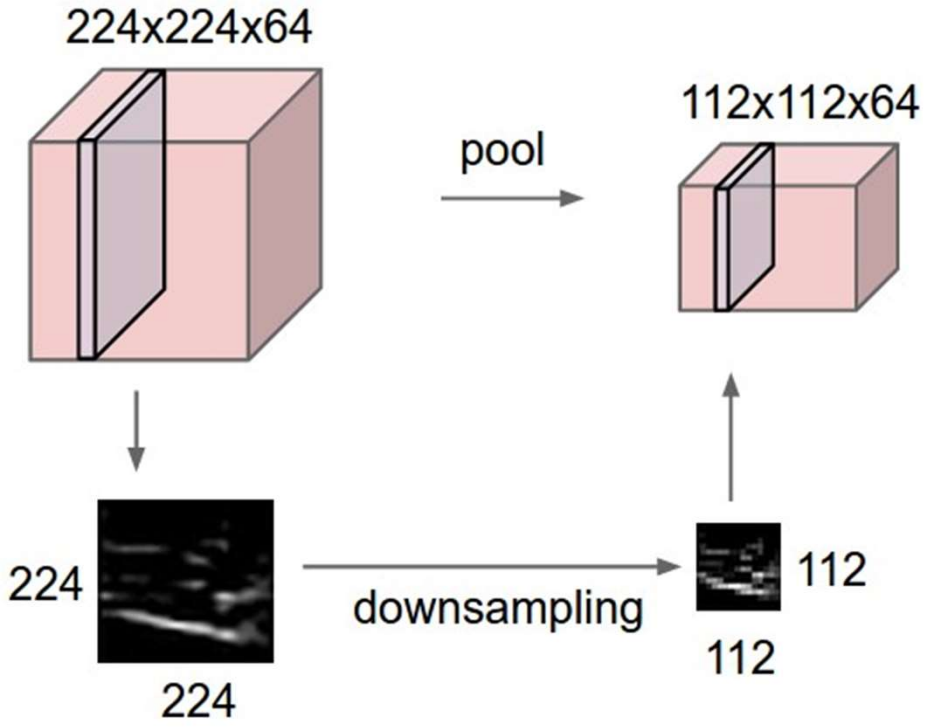


Convolving a 3 X 3 kernel over a 4 X 4 input

*Andrej Karpathy’s recent presentation
*http://iamaaditya.github.io/2016/03/one-by-one-convolution/

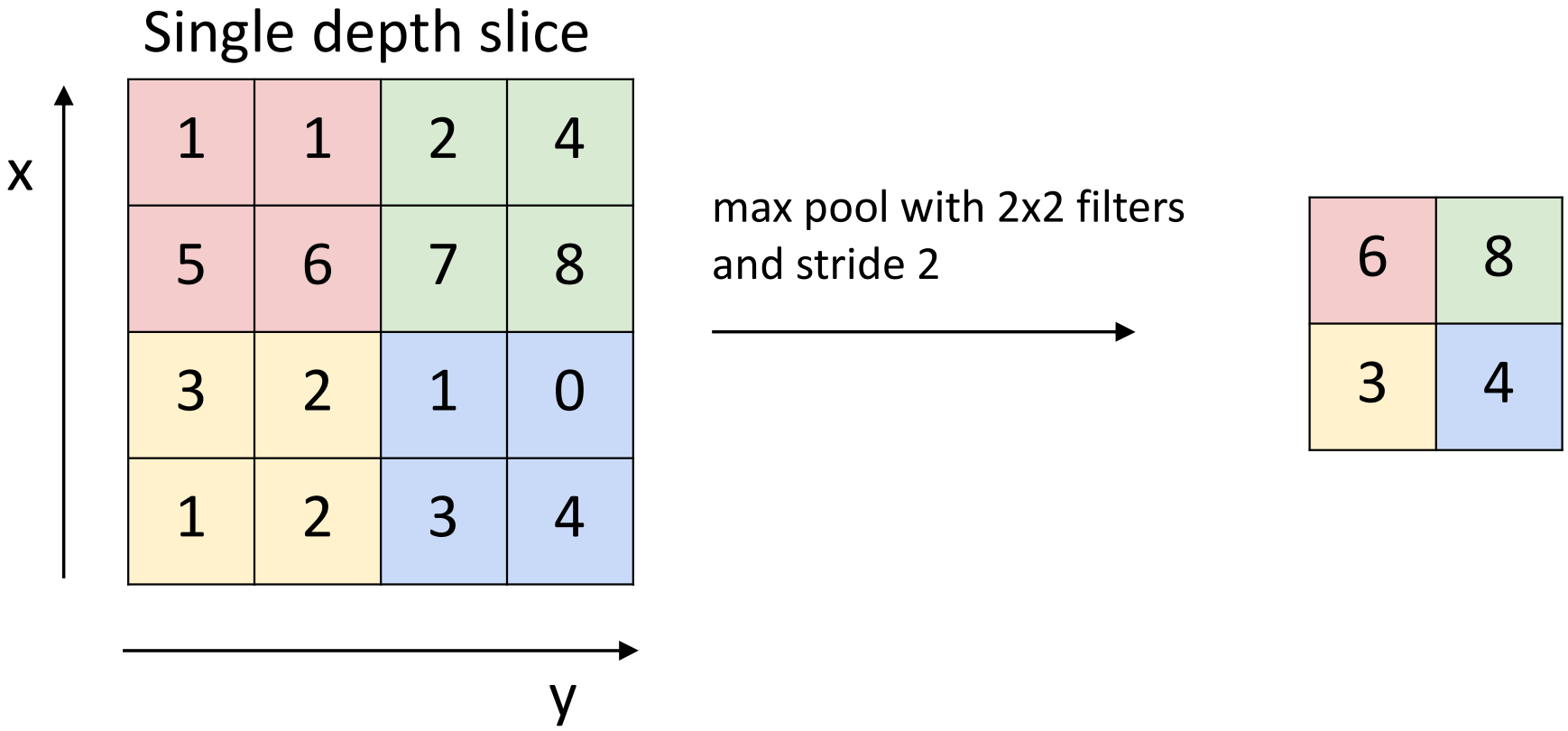
# Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently



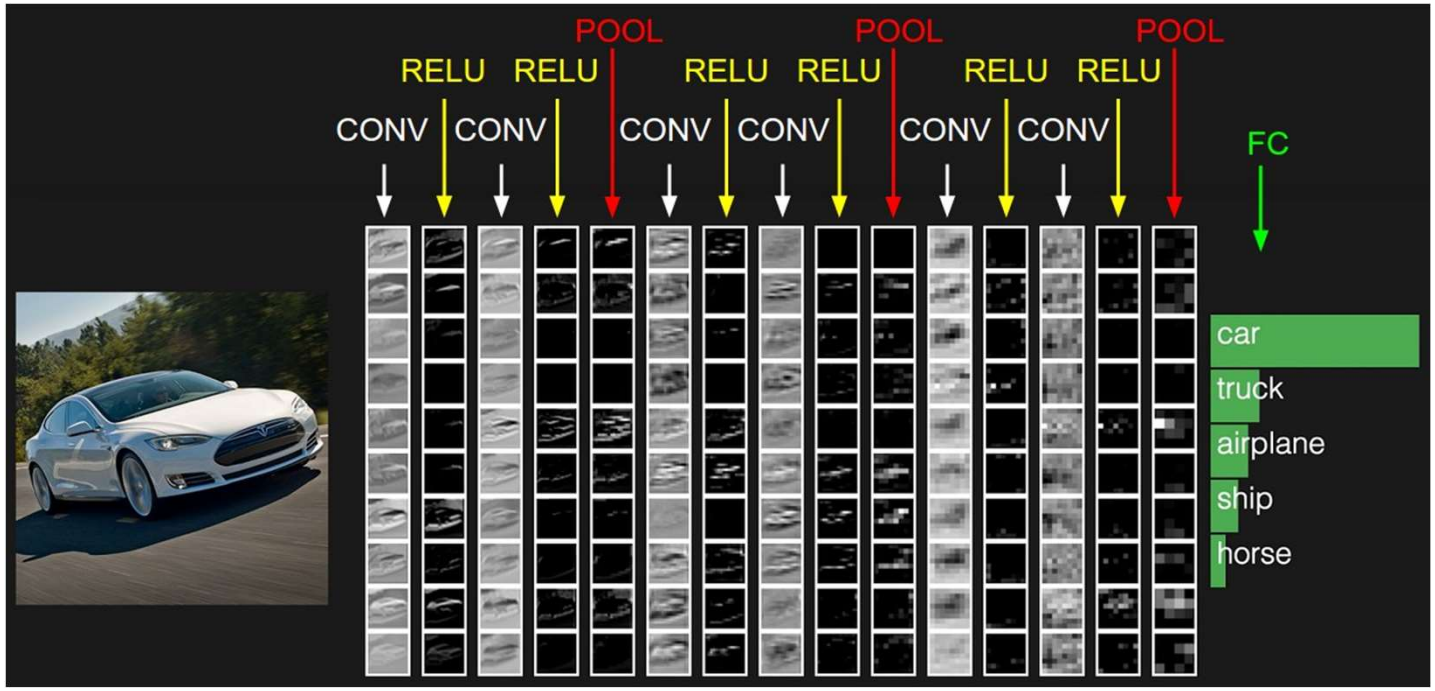

*Andrej Karpathy's recent presentation

# Max Pooling



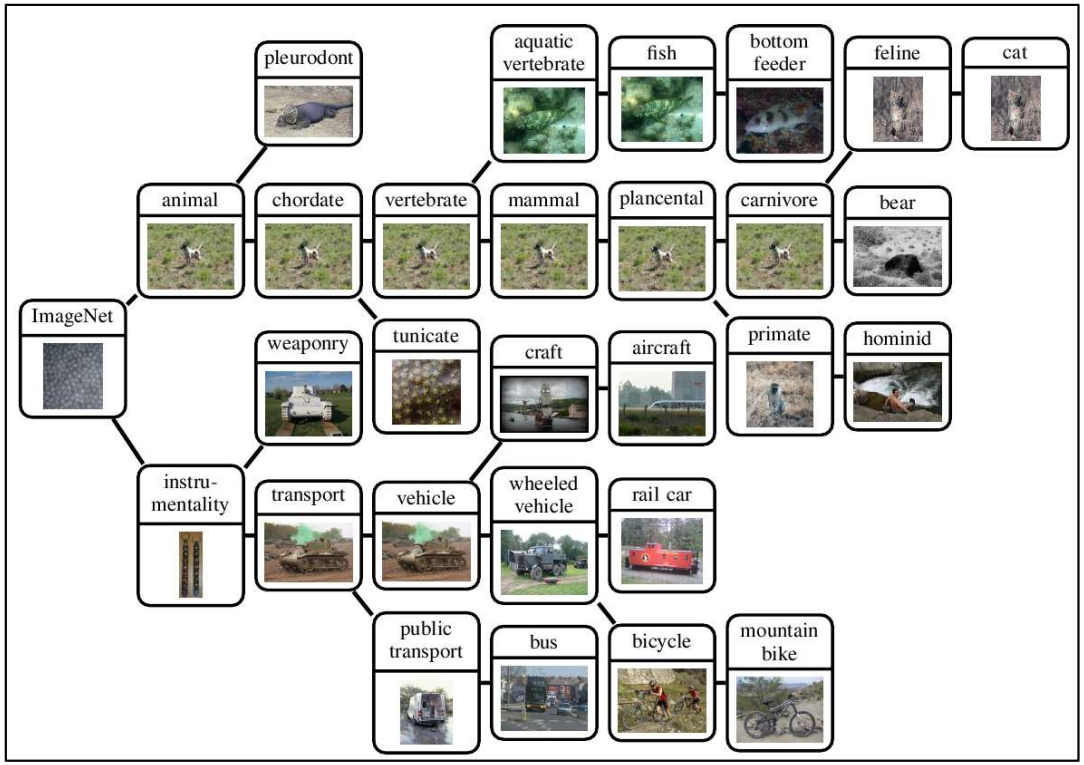

*Andrej Karpathy's recent presentation

# Fully Connected Layer (FC layer)



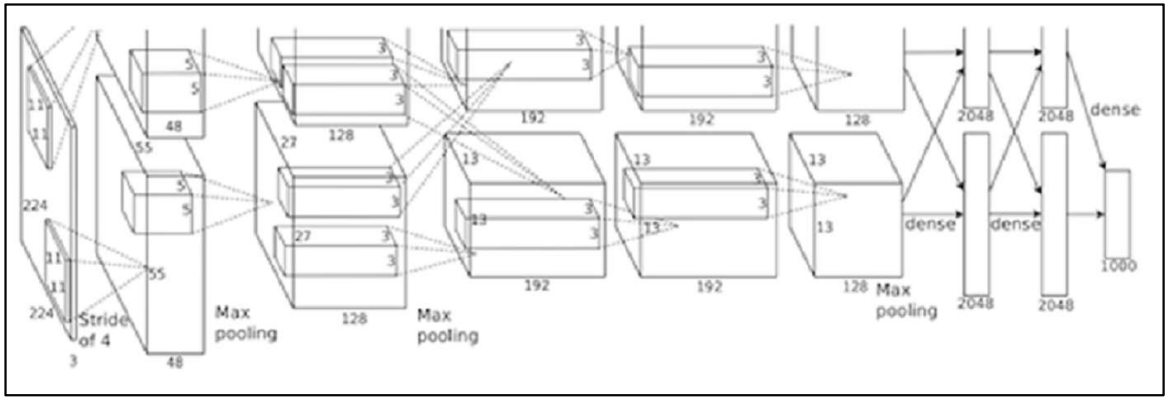

*Andrej Karpathy's recent presentation

# IMAGENET




*http://groups.inf.ed.ac.uk/calvin/imagenet/prototypes.html

# AlexNet



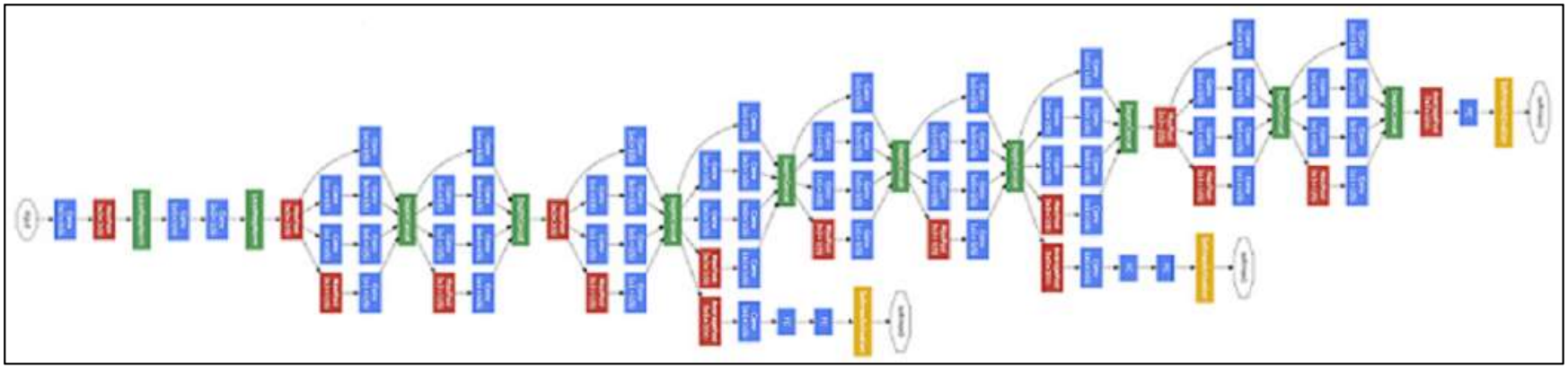

*https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

# VGGNet

| ConvNet Configuration | | | | | |
|---|---|---|---|---|---|
| A | A-LRN | B | C | D | E |
| 11 weight layers | 11 weight layers | 13 weight layers | 16 weight layers | 16 weight layers | 19 weight layers |
| input (224 × 224 RGB imag ) | | | | | |
| conv3-64 | conv3-64<br>**LRN** | conv3-64<br>**conv3-64** | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 |
| maxpool | | | | | |
| conv3-128 | conv3-128 | conv3-128<br>**conv3-128** | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 |
| maxpool | | | | | |
| conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256<br>**conv1-256** | conv3-256<br>conv3-256<br>**conv3-256** | conv3-256<br>conv3-256<br>conv3-256<br>**conv3-256** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| FC-4096 | | | | | |
| FC-4096 | | | | | |
| FC-1000 | | | | | |
| soft-max | | | | | |

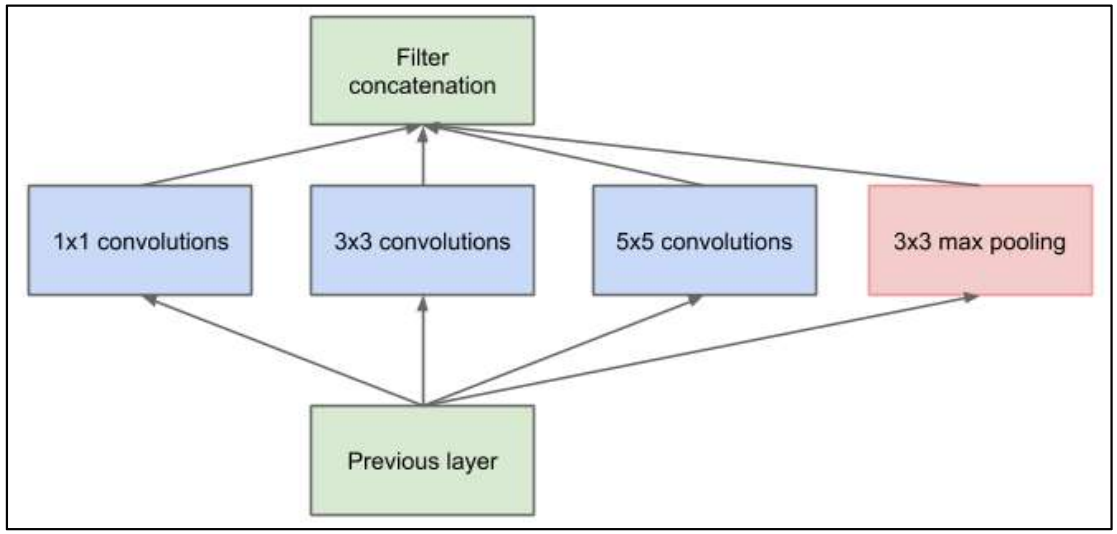
*http://arxiv.org/pdf/1409.1556v6.pdf

# GoogLeNet



*http://www.csc.kth.se/~roelof/deepdream/bvlc_gnet.html

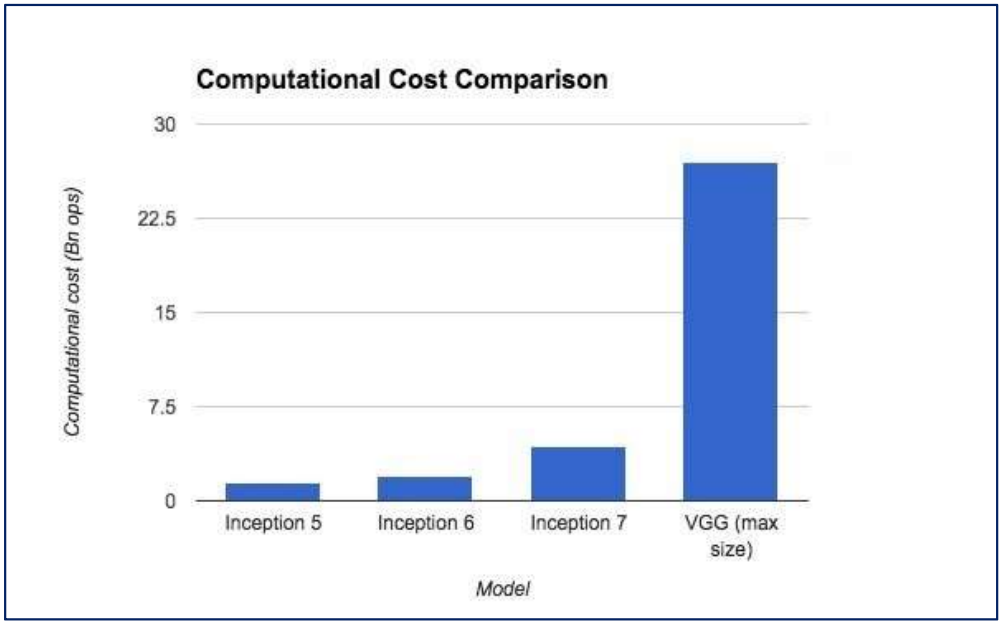
# GoogLeNet uses Inception




Inception Module

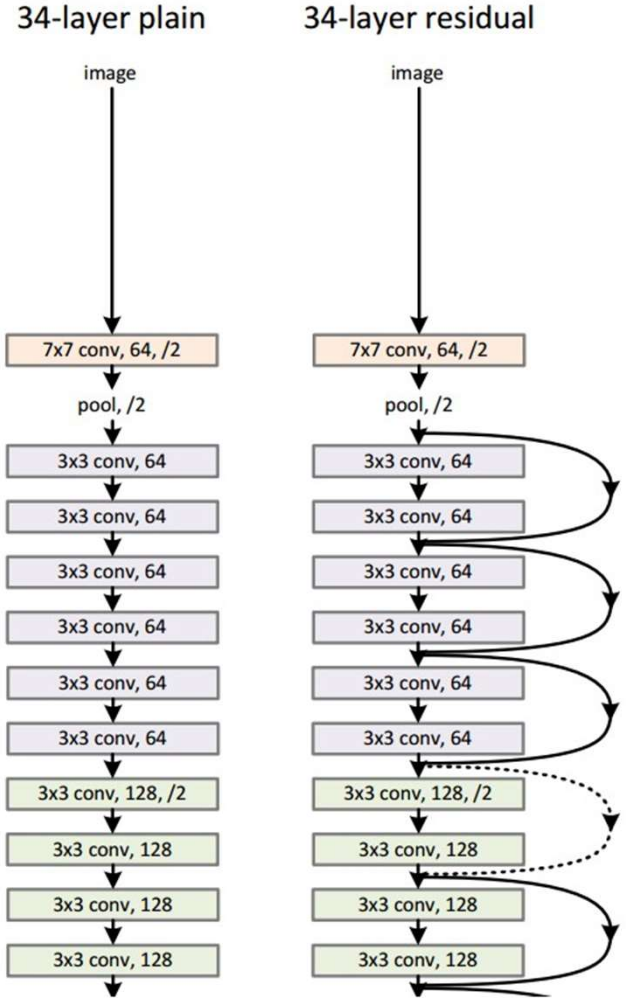
*https://arxiv.org/pdf/1409.4842v1.pdf

# Inception Performance comparison

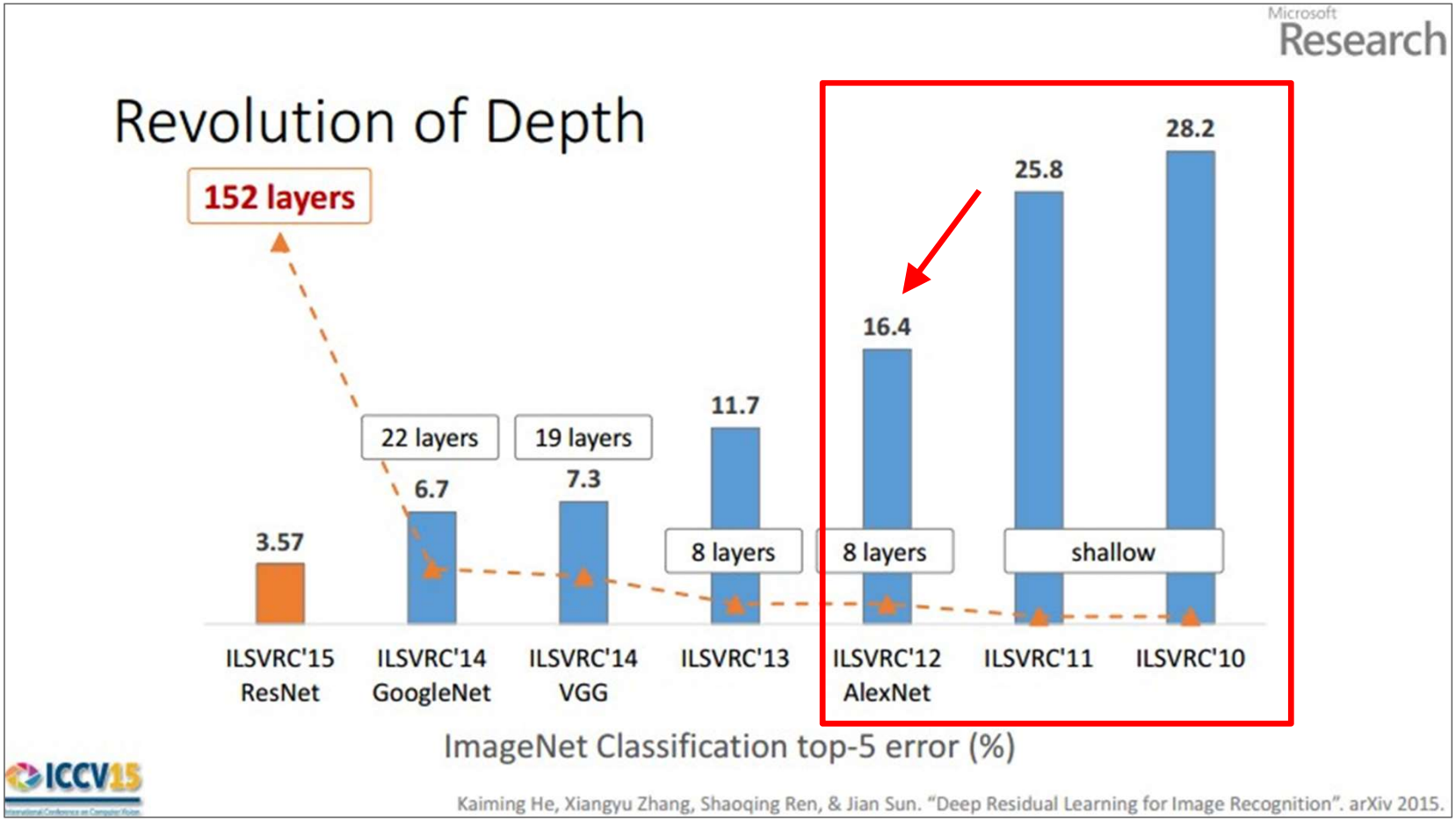



*http://lsun.cs.princeton.edu/slides/Christian.pdf

# Microsoft ResNet




*https://arxiv.org/pdf/1512.03385v1.pdf




*Andrej Karpathy's recent presentation

# Deep learning and natural language processing

# Deep learning enables sub-symbolic processing

| | |
|---|---|
| I | <i> |
| bought | <bought> |
| a | <a> |
| car | <car> |
| . | <.> |

You have to remember to represent “purchased” and “automobile.”

What about “truck”?

How do you encode the meaning of the entire sentence?

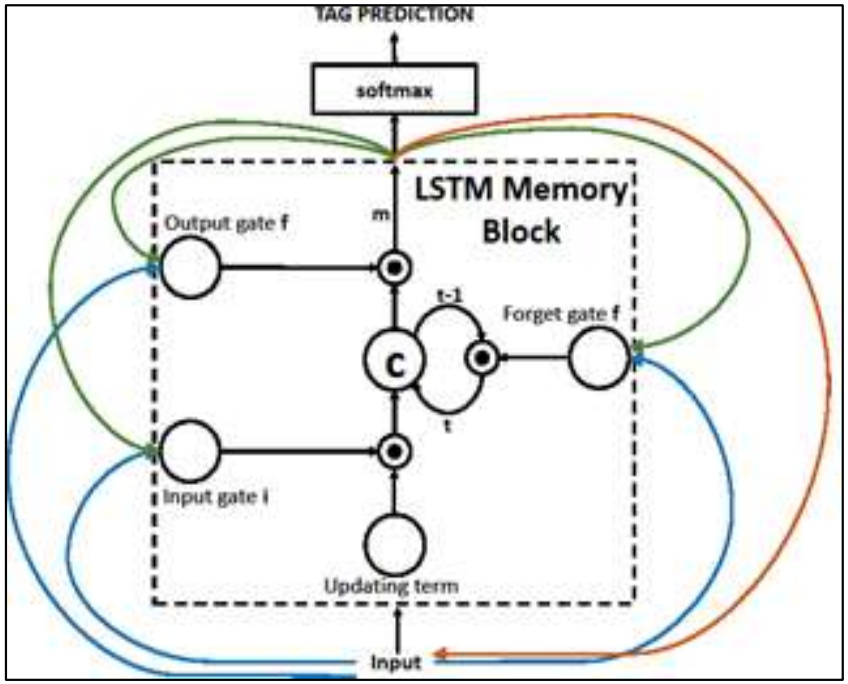
# But what about a sentence?

Algorithm for generating vectors for sentences

1. Make the sentence vector be the vector for the first word.
2. For each subsequent word, combine its vector with the sentence vector.
3. The resulting vector after the last word is the sentence vector.

Can be implemented using a recurrent neural network (RNN)

# LSTM RNN




*Huang et al., Communications ACM 01/2014

# What can you do with a Sentence Vector?

You can feed it to a classifier.

You can unwind in the other direction to do machine translation.

Called a seq2seq model, or Neural Machine Translation, or encoder-decoder model.

# Deep learning and question answering

Bob went home.
Tim went to the junkyard.
Bob picked up the jar.
Bob went to town.
Where is the jar? A: town

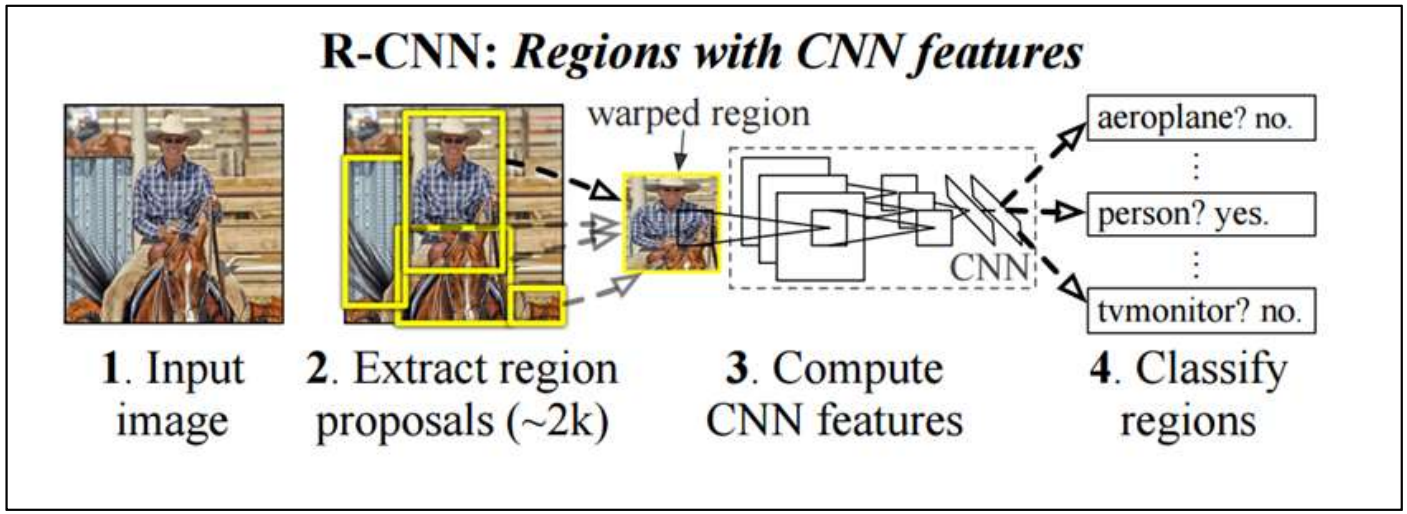
Memory Networks [Weston et al., 2014]: Updates memory vectors based on a question and finds the best one to give the output.

The office is north of the yard.
The bath is north of the office.
The yard is west of the kitchen.
How do you go from the office to the kitchen? A: south, east

Neural Reasoner [Peng et al., 2015]: Encodes the question and facts in many layers, and the final layer is put through a function that gives the answer.

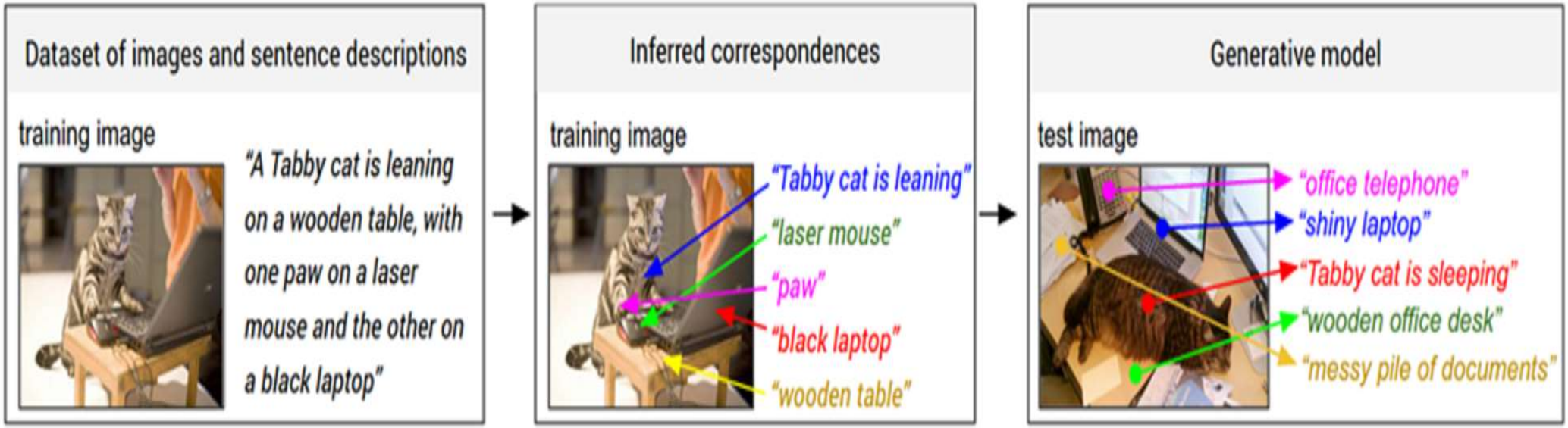
# Other commonly used DNNs

# Region Based CNN (RCNN)



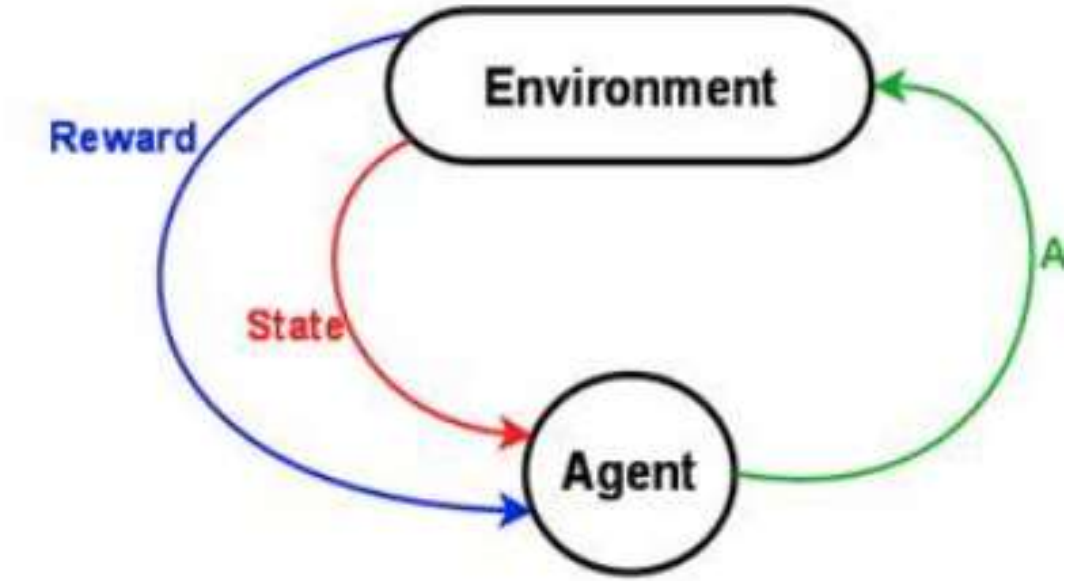

*https://arxiv.org/pdf/1311.2524v5.pdf

# Generating Image Descriptions (CNN-RNN)




*https://arxiv.org/pdf/1412.2306v2.pdf

# Deep Reinforcement Learning



# Deep Learning – Some use-cases in next 5 years?

# Advanced Melanoma Screening and Detection

- Researchers at the University of Michigan are putting advanced image recognition to work, detecting one of the most aggressive, but treatable in early stages, types of cancer.

- The team trained a neural network to isolate features (texture and structure) of moles and suspicious lesions for better recognition and detecting melanoma known to date.

# Neural Networks for Brain Cancer Detection

- A team of French researchers is working on spotting invasive brain cancer cells during surgery.

- They found that using neural networks in conjunction with Raman spectroscopy during operations allows them to detect the cancerous cells easier and reduce residual cancer post-operation.

# Neural Networks in Weather Forecasting and Event Detection

- This traditional area for large-scale supercomputers is now becoming a hotbed for neural network development, particularly when it comes to weather event (pattern) detection.

- In one such use case, computational fluid dynamics codes are matched with neural networks and other genetic algorithm approaches to detect cyclone activity.

# Neural Networks in Finance

- Trading and risk management are two areas where we would expect to see developments for neural networks

- Popular technical indicators along with neural networks and genetic algorithms are used to generate buy and sell signals for each stock and for portfolios of stocks

# What deep learning still can’t do?

# What deep learning still can't do

- System 1: Fast and Parallel
  - With computer vision, we seem to be on the right track
  - Reinforcement learning is useful in increasingly large worlds

- System 2: Slow and Serial
  - Still lacking in common sense
  - Language processing needs a grounded understanding

# Limitations of deep learning

- The encoded meaning is grounded with respect to other words.
- There is no linkage to the physical world.

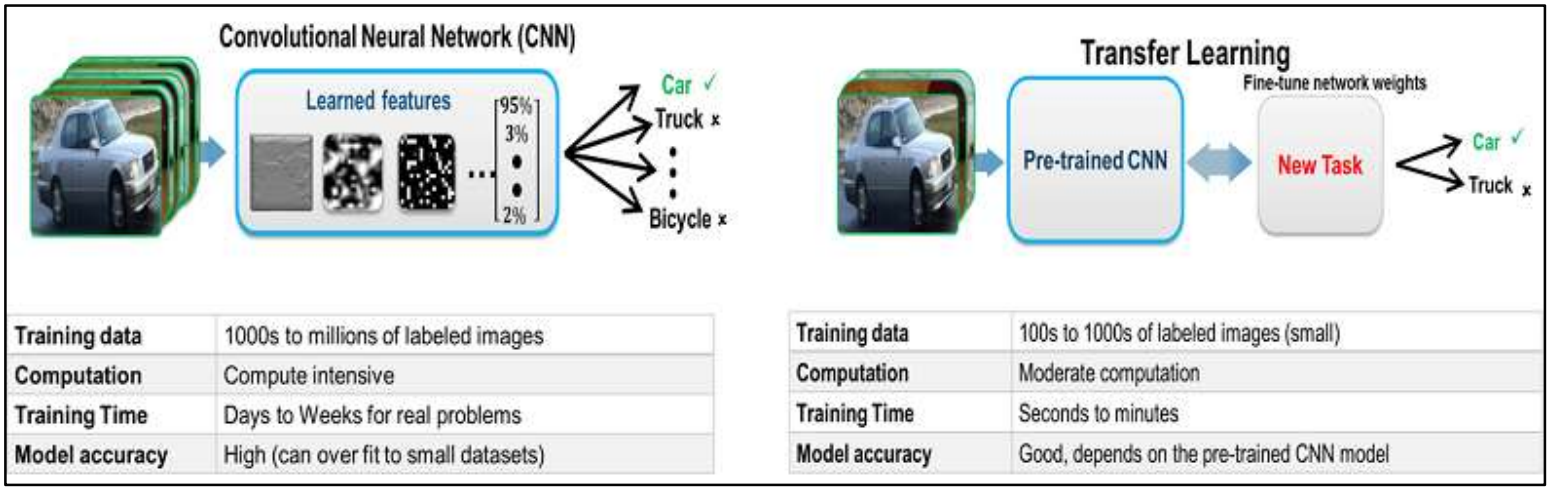
Bob went home.
Tim went to the junkyard.
Bob picked up the jar.
Bob went to town.
Where is the jar? A: town

Deep learning has no understanding of what it means for the jar to be in town.

For example that it can't also be at the junkyard. Or that it may be in Bob's car, or still in his hands.

# Increasing Re-usability of Deep Learning models

# Transfer Learning & Fine-tuning

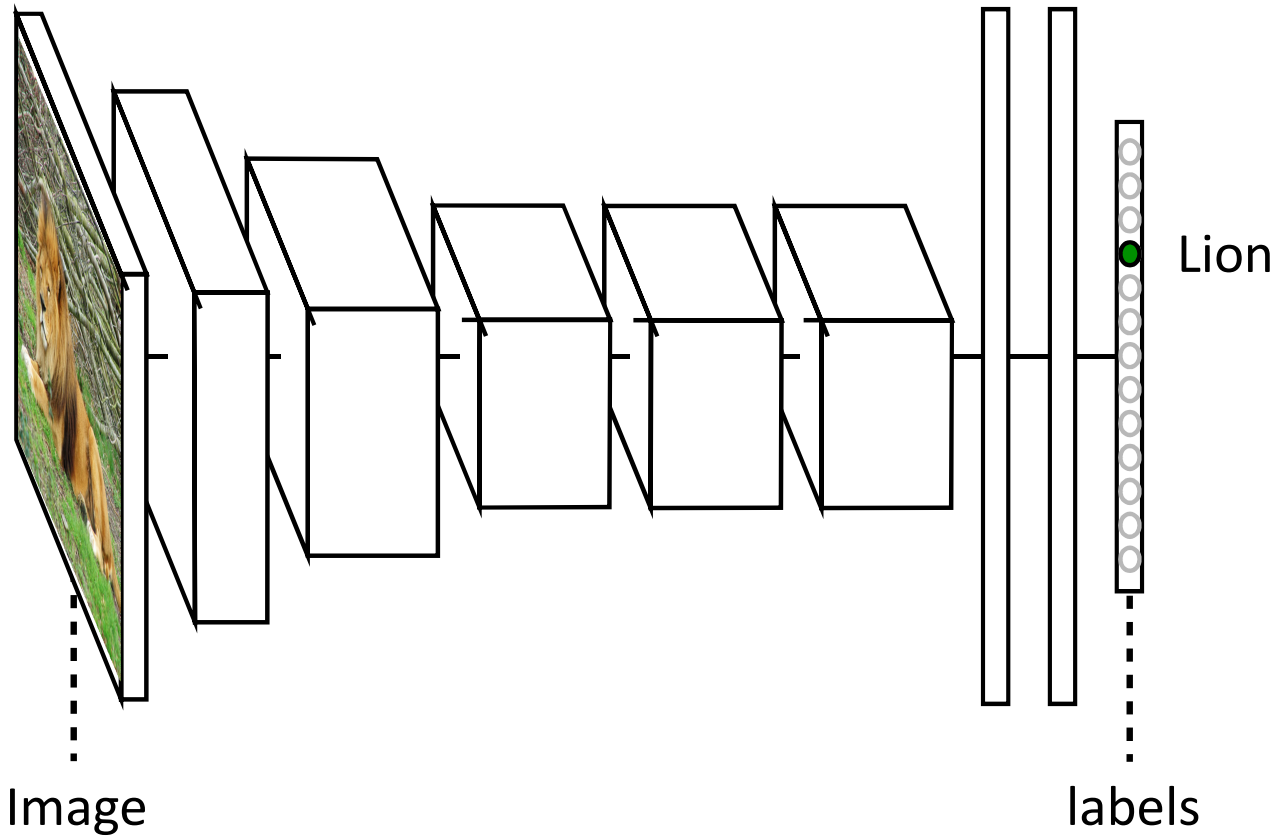



| | |
|---|---|
| **Training data** | 1000s to millions of labeled images |
| **Computation** | Compute intensive |
| **Training Time** | Days to Weeks for real problems |
| **Model accuracy** | High (can over fit to small datasets) |

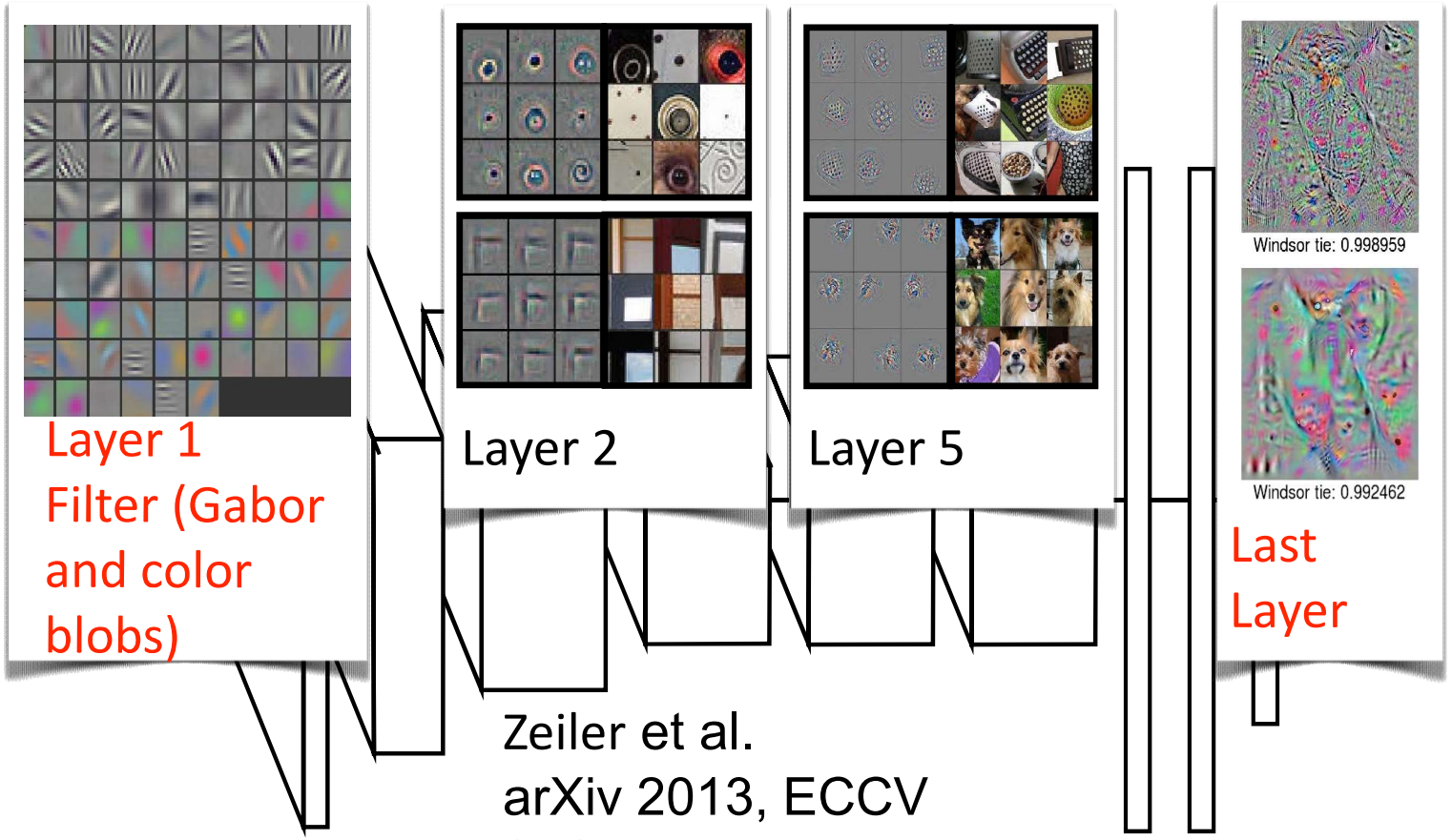
| | |
|---|---|
| **Training data** | 100s to 1000s of labeled images (small) |
| **Computation** | Moderate computation |
| **Training Time** | Seconds to minutes |
| **Model accuracy** | Good, depends on the pre-trained CNN model |

*http://www.mathworks.com/discovery/deep-learning.html?requestedDomain=uk.mathworks.com

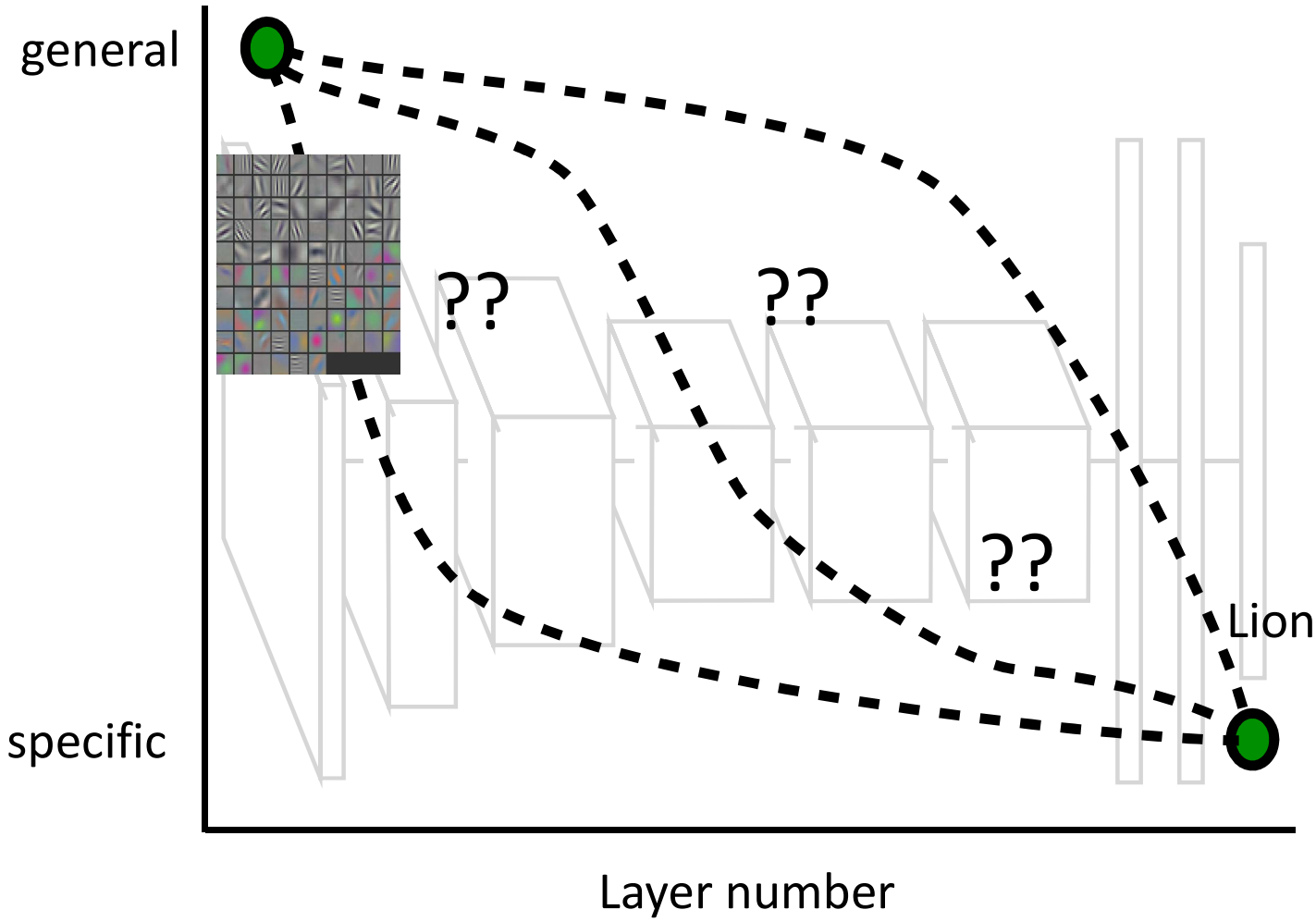
# Convolutional Neural Networks: AlexNet



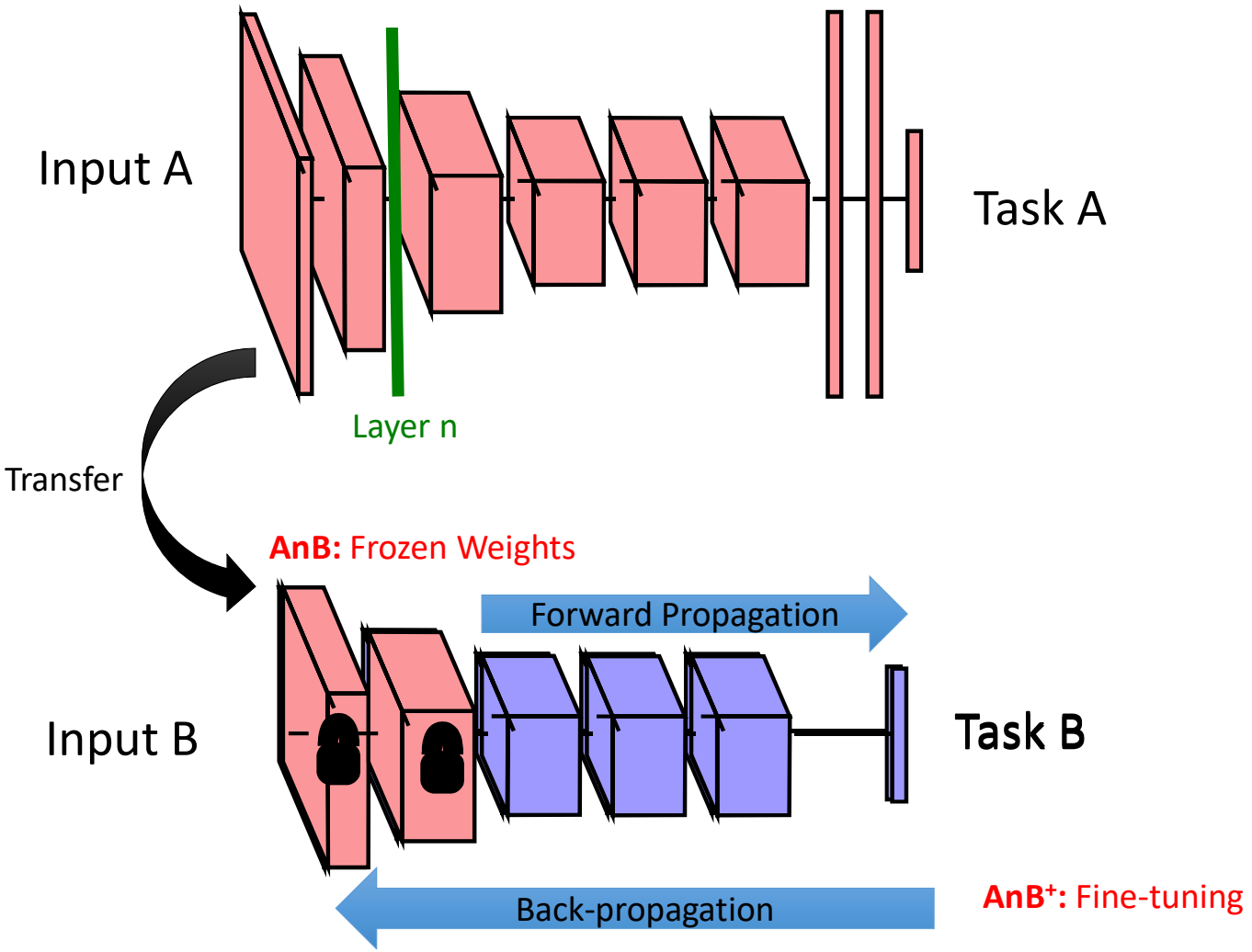

Krizhevsky, Sutskever, Hinton — NIPS 2012

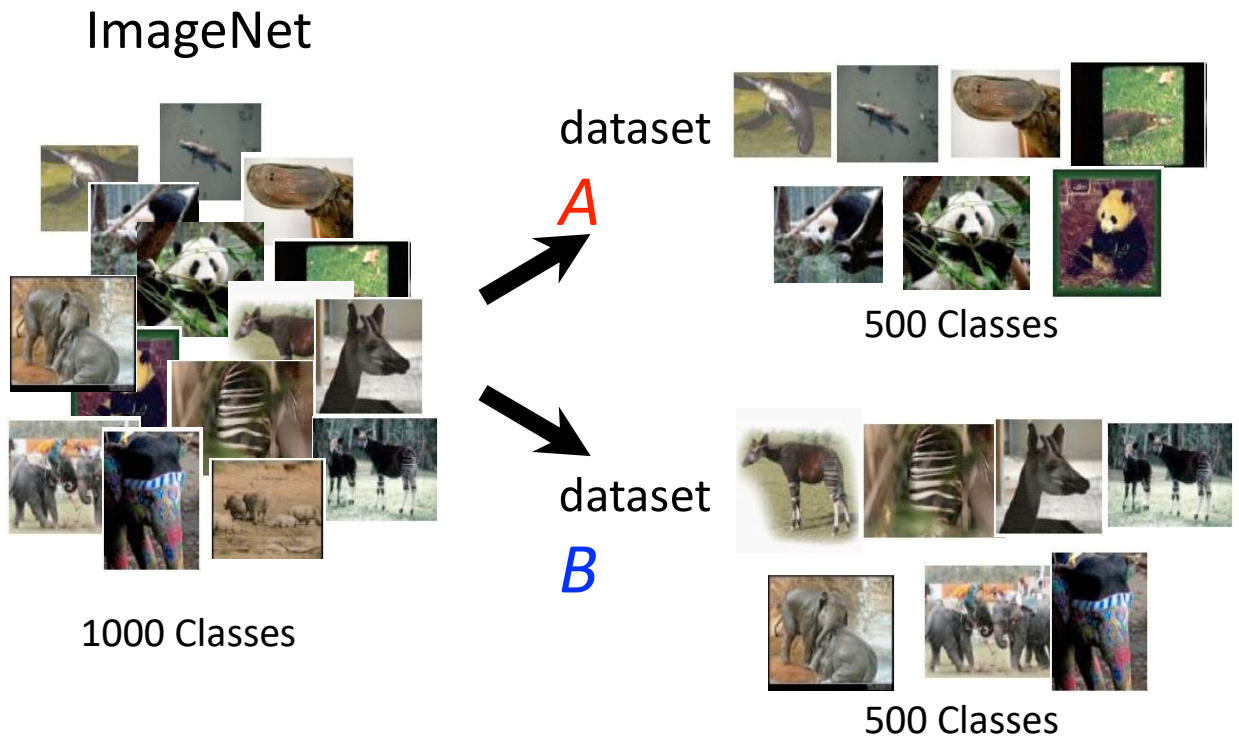


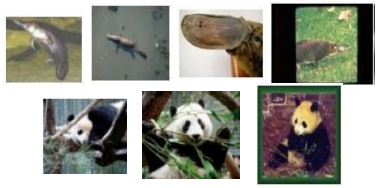

**Gabor filter:** linear filters used for edge detection with similar orientation representations to the human visual system


general
specific
??
??
??
Lion
Layer number

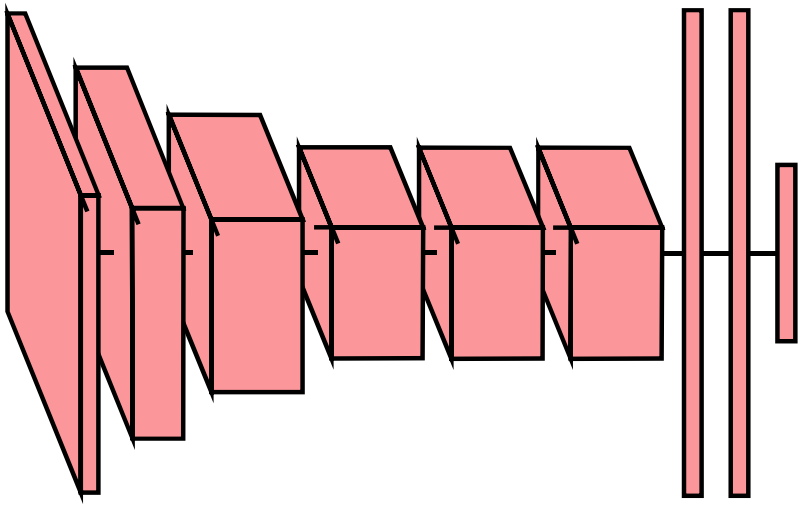
# Transfer Learning Overview




ImageNet
1000 Classes
dataset
A
500 Classes
dataset
B
500 Classes

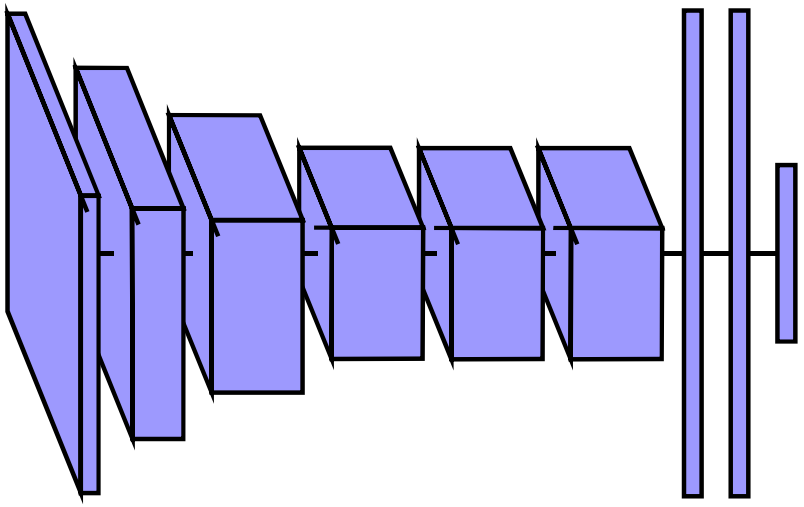


A Images

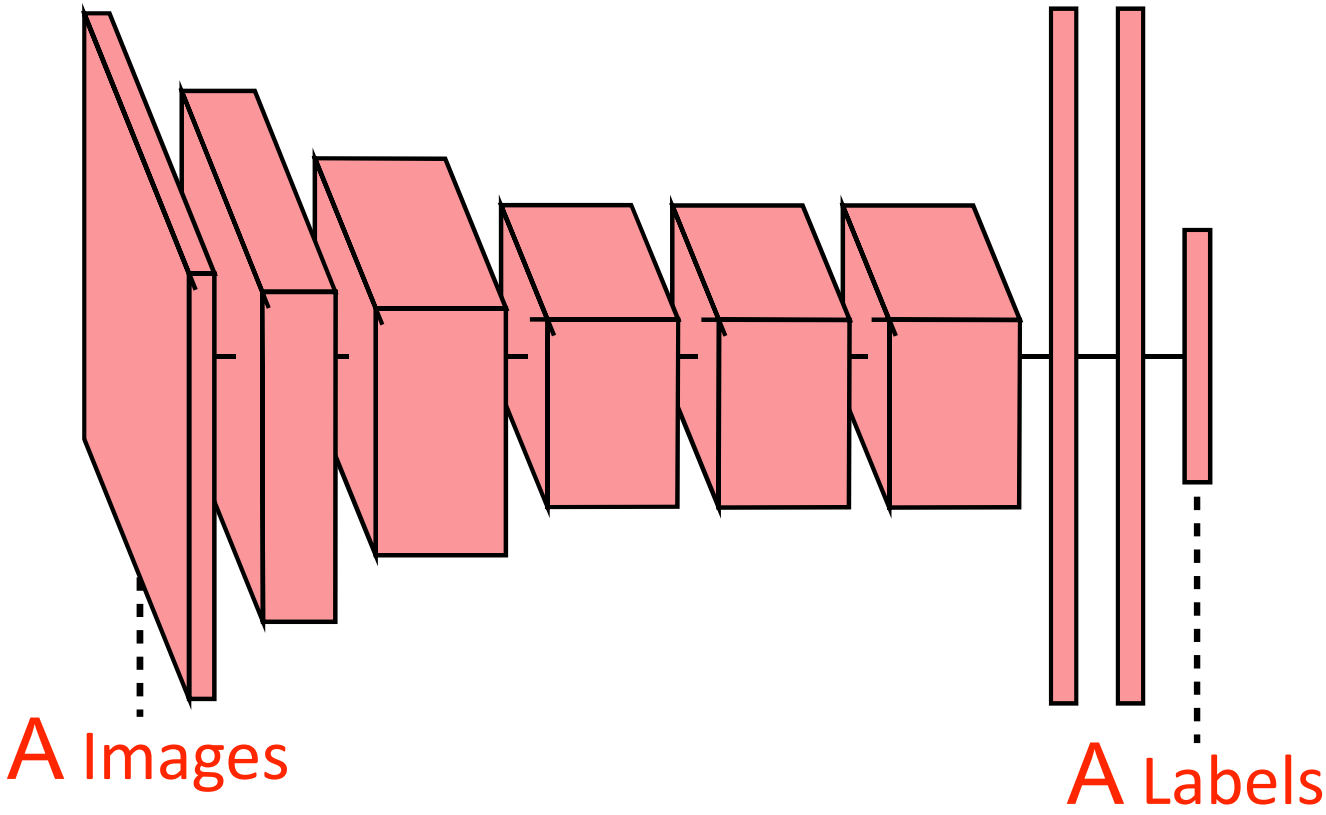


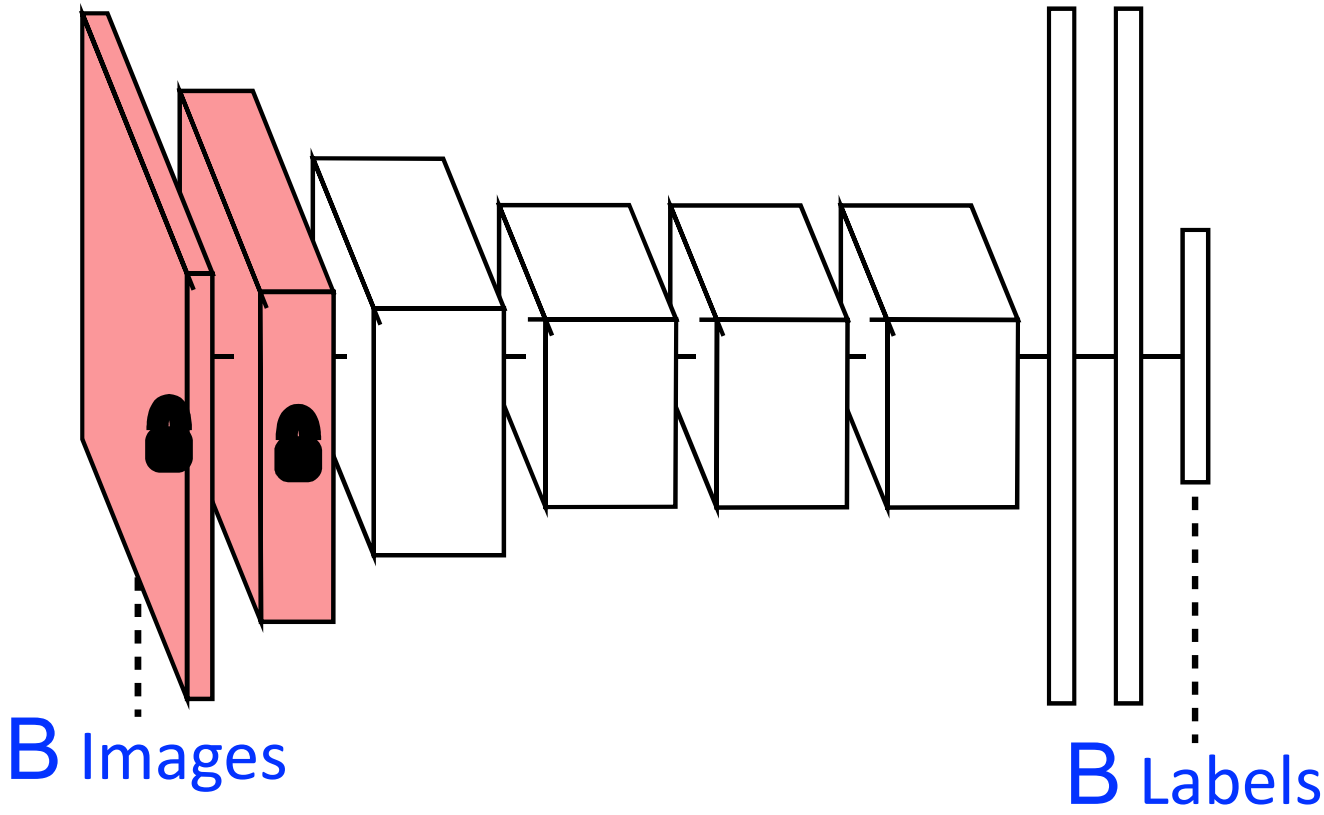
baseA

B Images

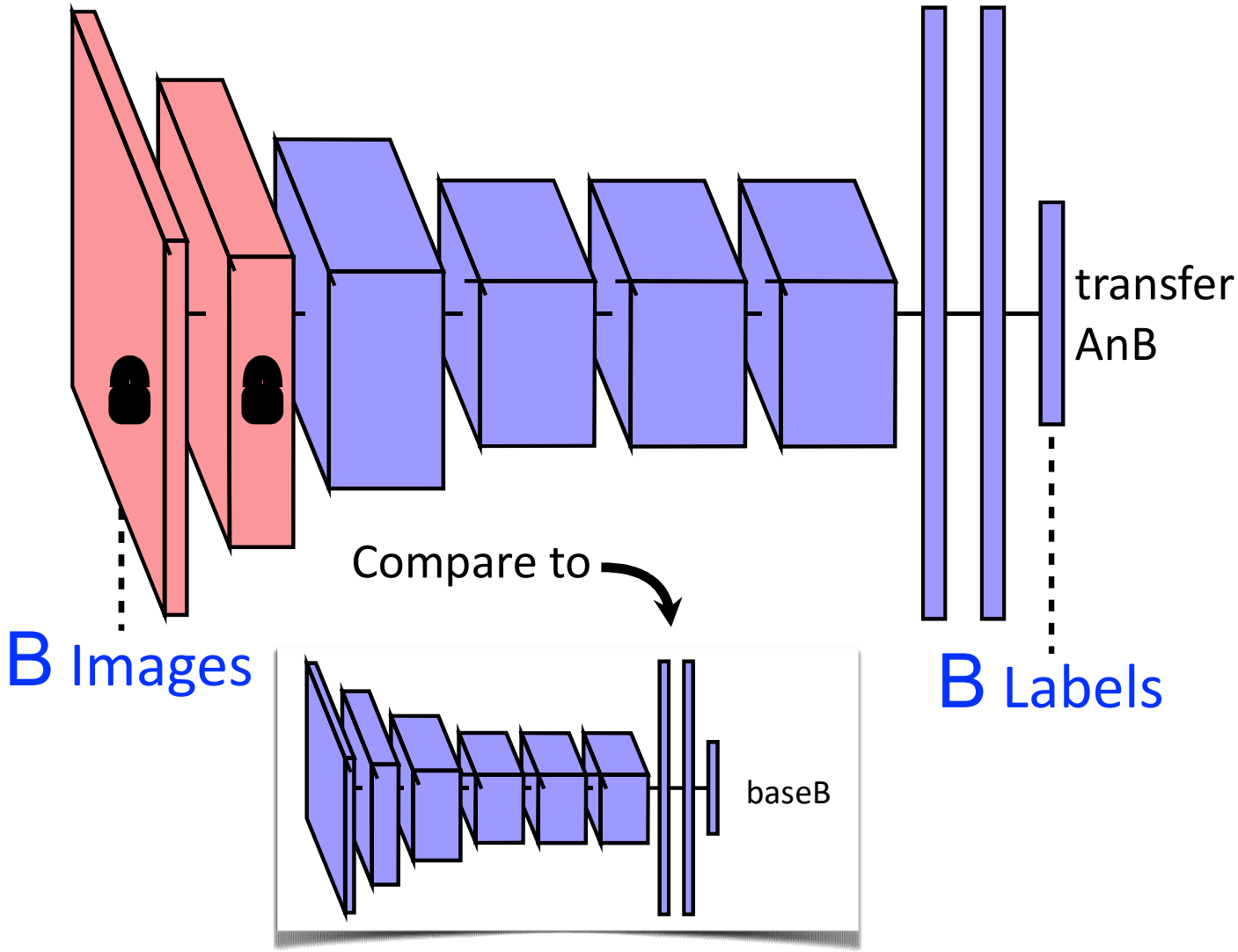


baseB

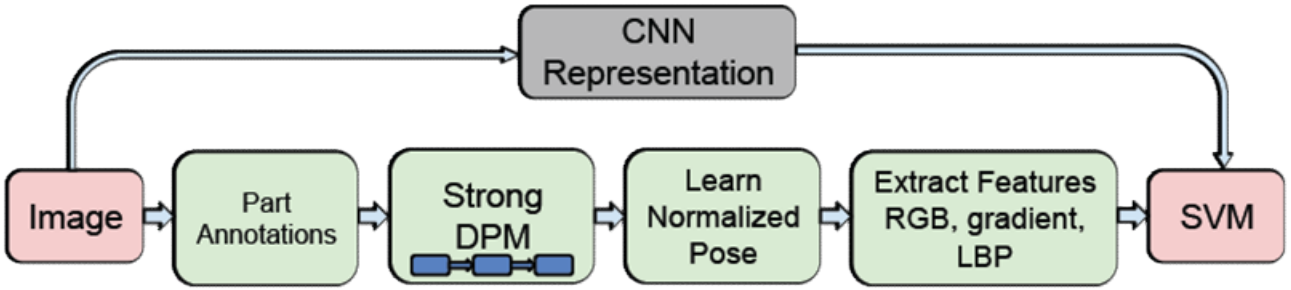

A Images
A Labels




Hypothesis: if transferred features are specific to task A, performance drops. Otherwise the performance should be the same.


transfer
AnB
Compare to
B Images
B Labels
baseB

# Can the features be exploited for any vision tasks?

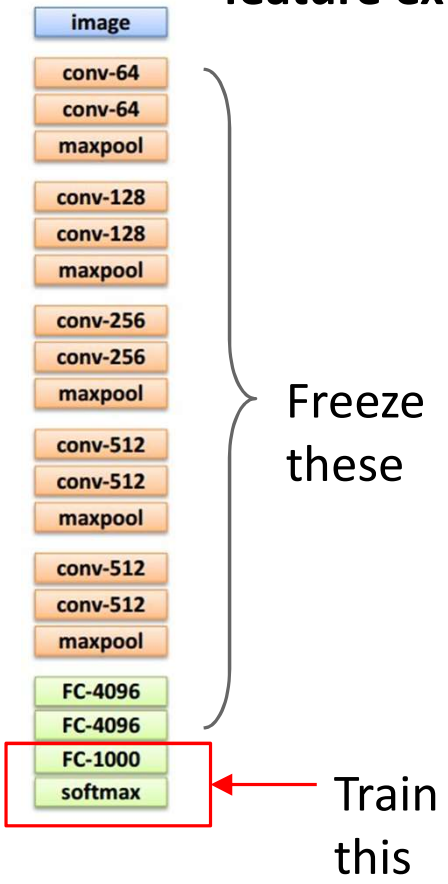



**CNN representation** replaces pipelines of service-oriented architecture (s.o.a) methods and achieve better results.

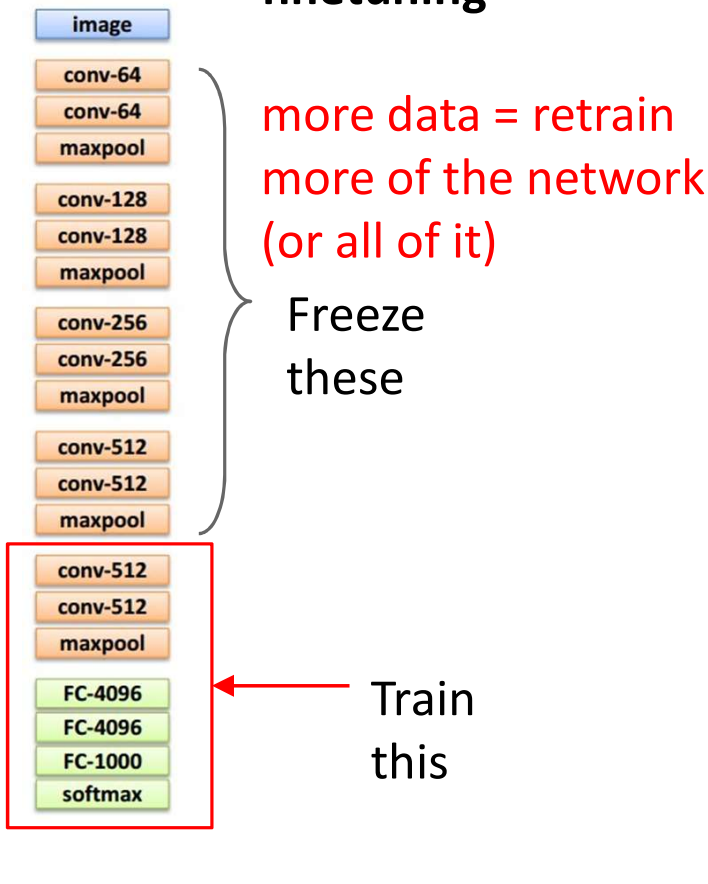
# 4 types of Transfer Learning

1. Train on Imagenet


2. Small dataset: **feature extractor**

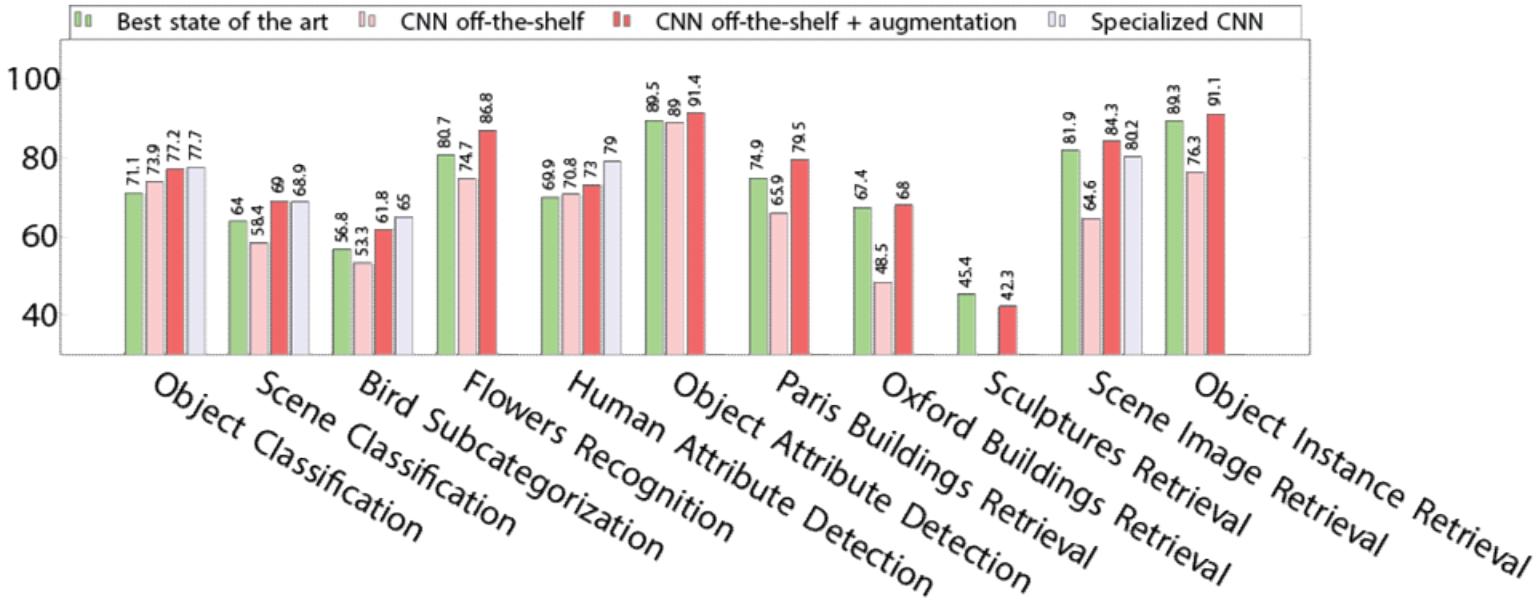



3. Medium dataset: **finetuning**



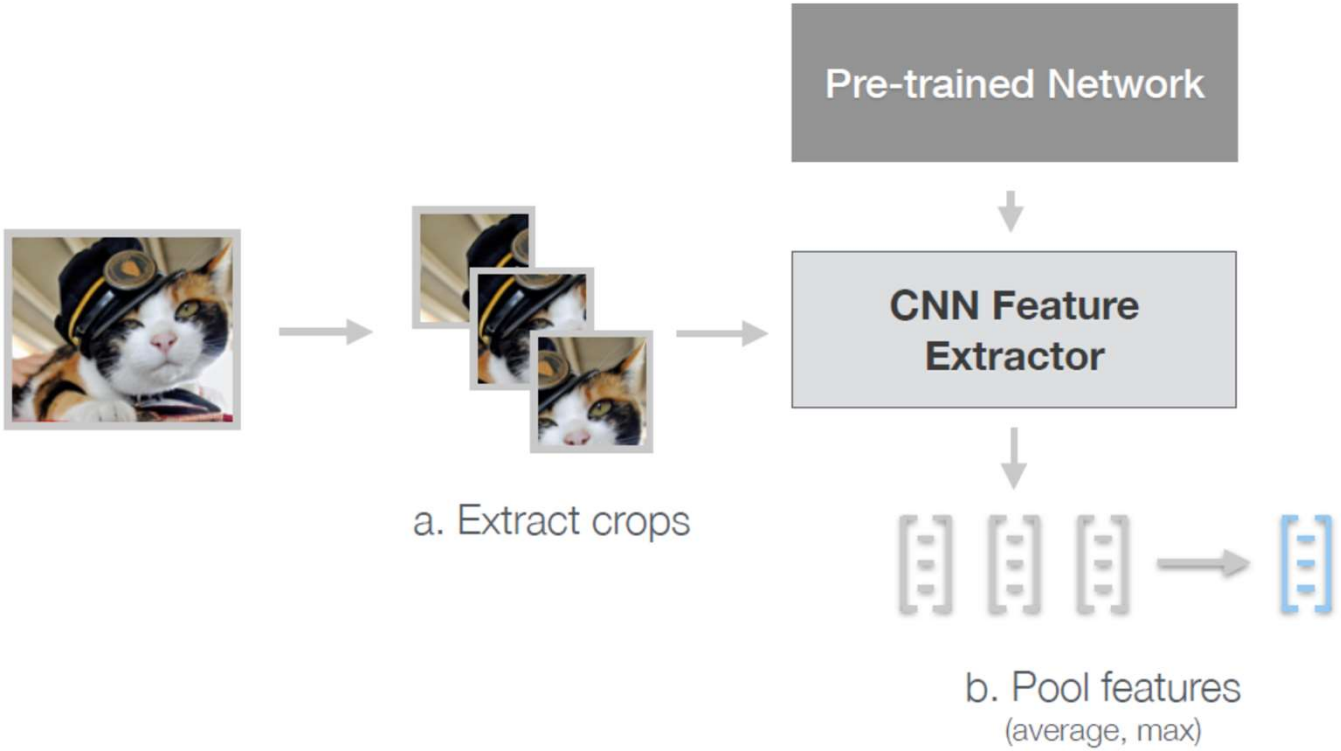

*Andrej Karpathy's recent presentation

# Transfer Learning



**Data Augmentation:**

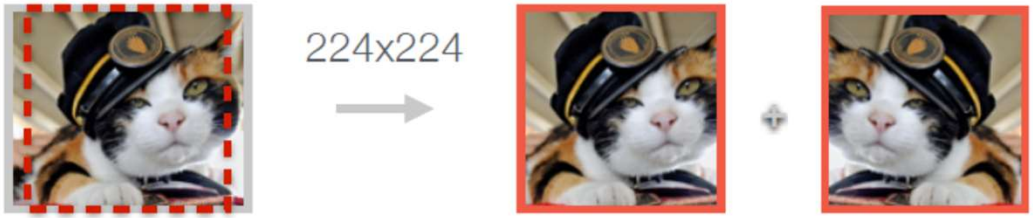
*Given* pre-trained ConvNet, augmentation applied at test time



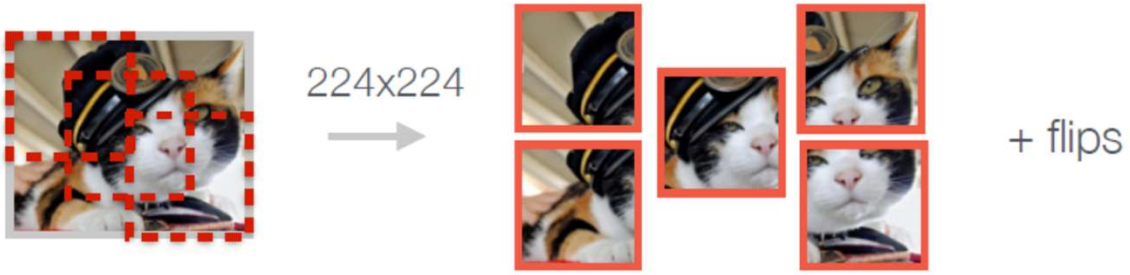
**Data Augmentation:**

a. No augmentation (= 1 image)


b. Flip augmentation (= 2 images)

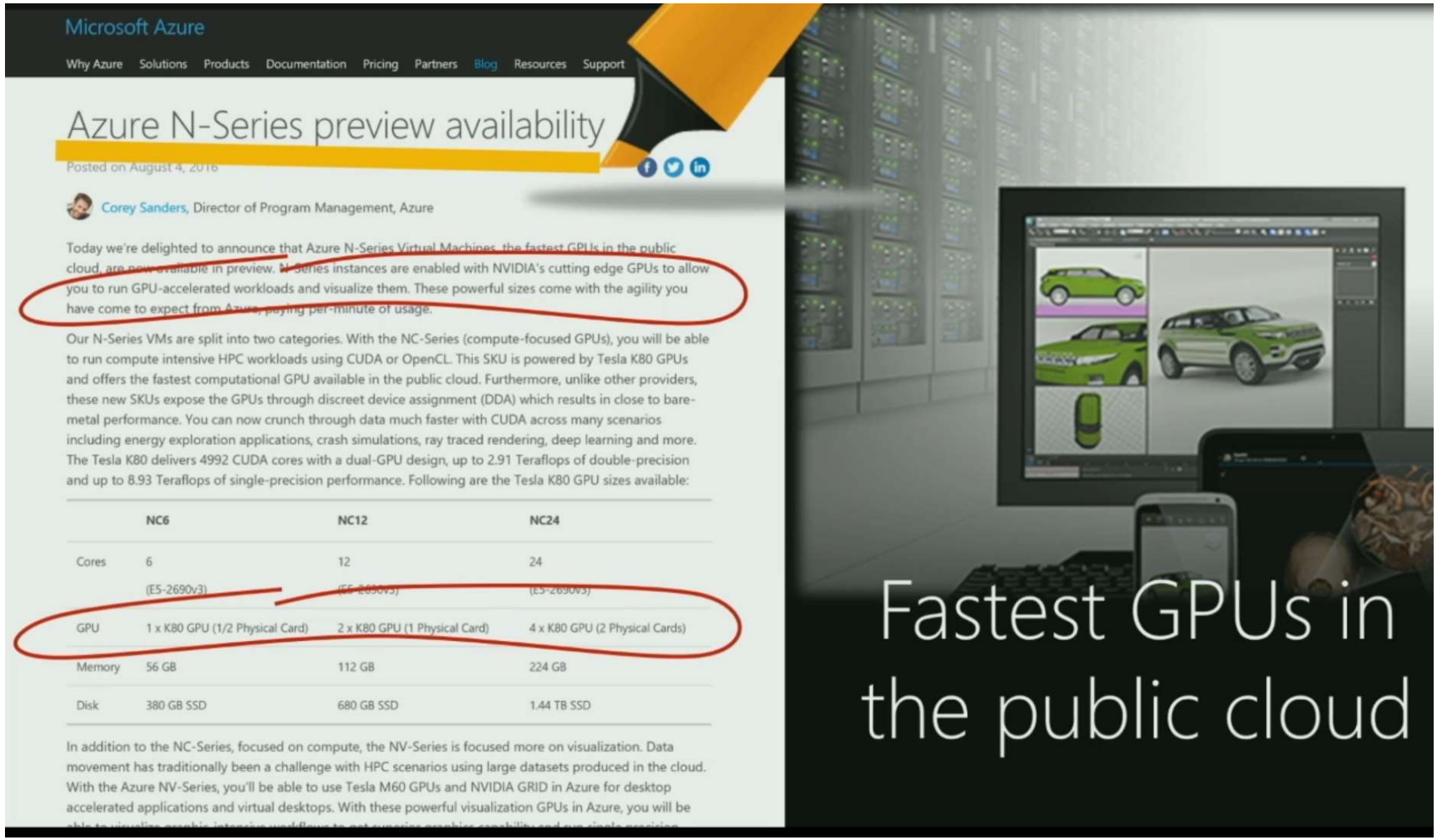



c. Crop+Flip augmentation (= 10 images)



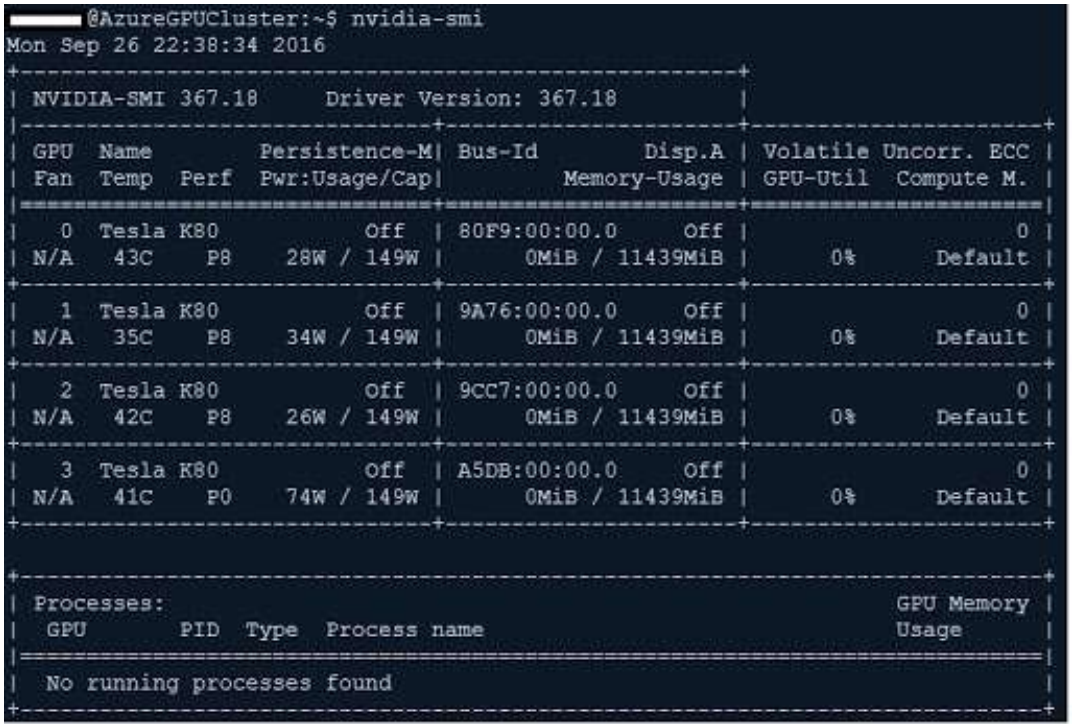
# Use Cases

# GPUs in Azure



# Azure GPU VM

- Used Ubuntu 16.04 N-Series VM
- NC24 VM with 4 NVIDIA Tesla K80

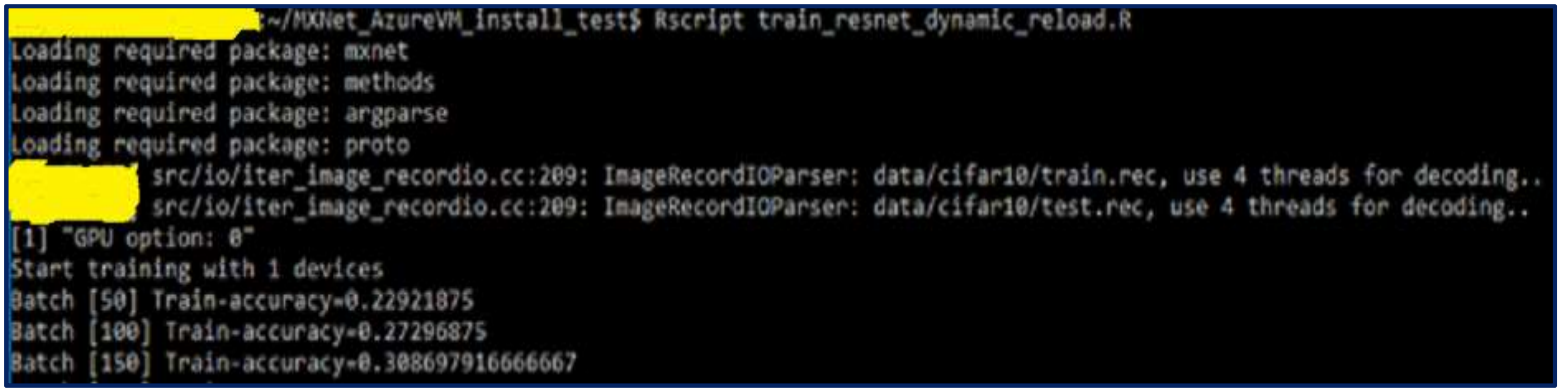


# DL Framework: MXNET

- DL toolkit by DMLC (2015)
- Supports almost all common DL models
- Supports multi-GPU training
- Supports distributed training
- Python/R support

# Case Study I: ResNet on CIFAR 10

- Based on Microsoft Blog on how to use MXNET with MRO/MRS
- Used MXNET with MRO and 1 GPU




*Tao Wu and Team, Microsoft

# Case Study II: Finetuning pre-trained Inception BN Network

- Used an ImageNet pretrained Inception BN Network
- Used MXNET with anaconda python and 4 GPUs
- Try to predict contents of a new unseen image

# Using trained Inception BN Model

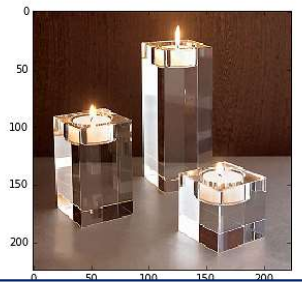
In [21]:
```
# Load the pre-trained model
prefix = "./Inception/model/Inception_BN"
num_round = 39
devices = [mx.gpu(i) for i in range(3)]
model = mx.model.FeedForward.load(prefix, num_round, ctx=devices, numpy_batch_size=1)
```

In [22]:
```
# load mean image
mean_img = mx.nd.load("Inception/model/mean_224.nd")["mean_img"]
```

In [23]:
```
# load synset (text label)
synset = [l.strip() for l in open('Inception/model/synset.txt').readlines()]
```

```
top5 = [synset[pred[i]] for i in range(5)]
print("Top5: ", top5)
```

```
('Original Image Shape: ', (552, 550, 3))
('Top1: ', 'n02948072 candle, taper, wax light')
('Top5: ', ['n02948072 candle, taper, wax light', 'n03666591 lighter, light, igniter, ignitor', 'n04456115 torch', 'n03729826 m
atchstick', 'n03347037 fire screen, fireguard'])
```



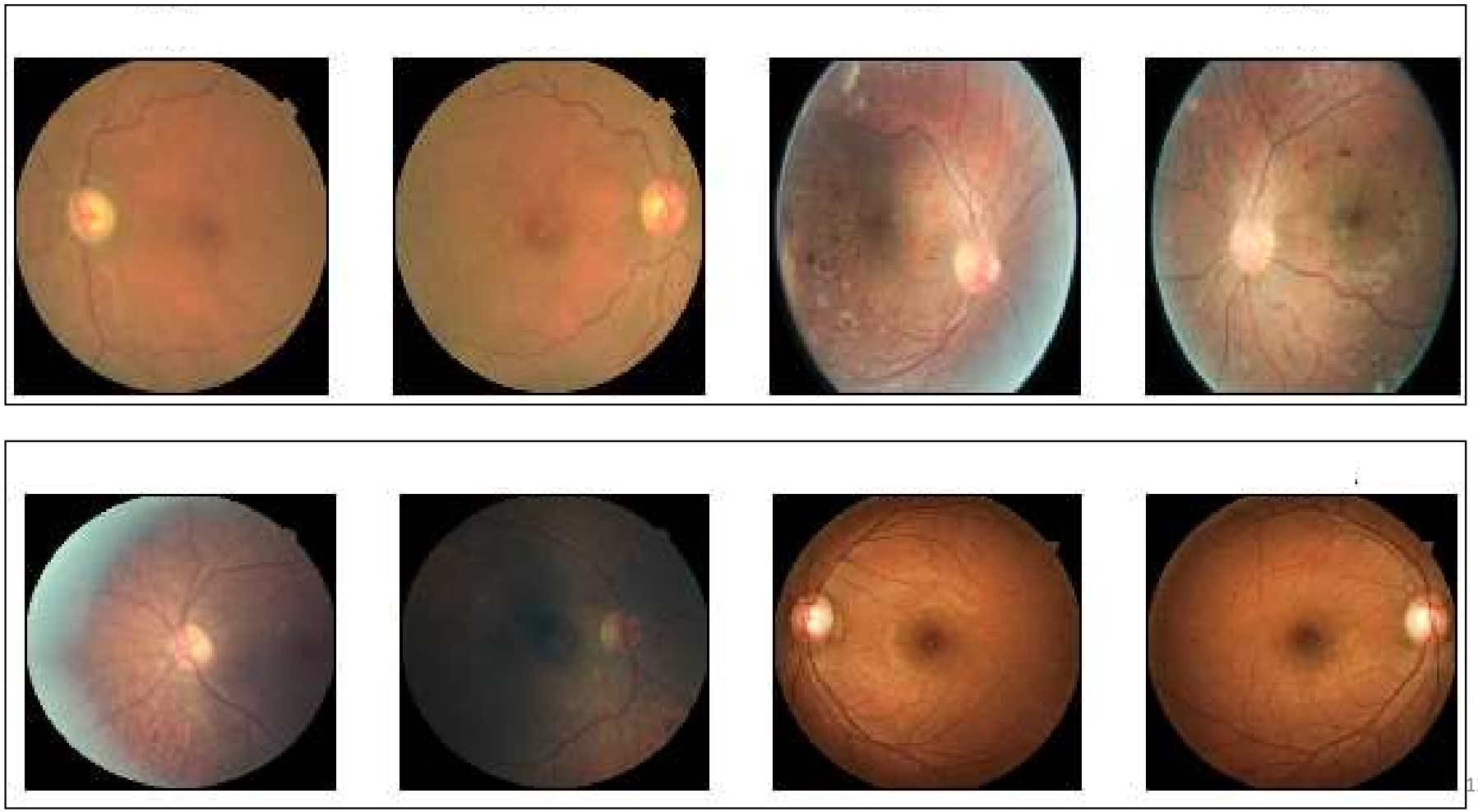
# DL Framework : Theano (Lasagne/Keras)

- Developed by a group at Université de Montréal (2007)
- Easy to customize with domain-specific data
- Experimental multi-GPU support (through Platoon)
- Experimental distributed (only through Keras)
- Great Python support

# Case Study III: Diabetic Retinopathy Prediction

- Used labeled fluorescein angiography images of eyes to improve Diabetic Retinopathy (DR) prediction (Kaggle)
- Used a DCNN to improve DR prediction
- Used Theano+Lasagne+Keras with anaconda python and 1 GPU

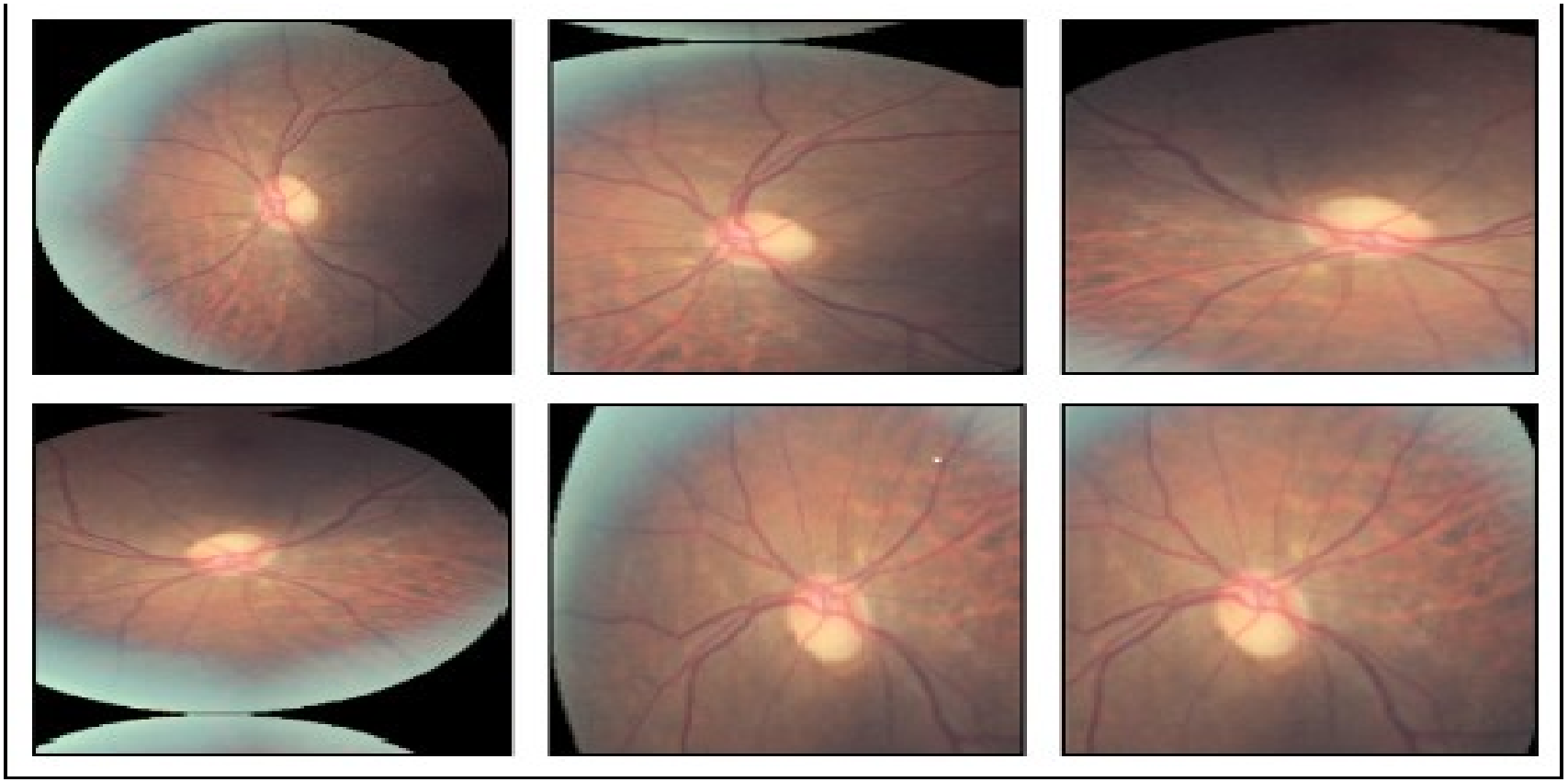
# Diabetic Retinopathy



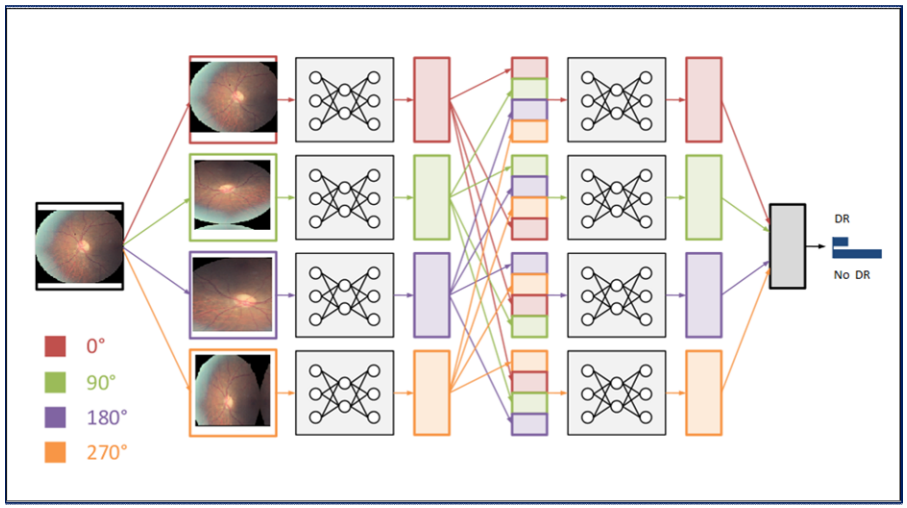
# Fine-tuning DCNN model

- ImageNet pre-trained GoogleNet
- Trained ImageNet features transferred as initial weights for DR prediction.
- Lower layers (first 2 layers) of the pre-trained DCNN contain generic features/weights that can be used for the DR prediction task.

# Reduce Overfitting: Augmentation



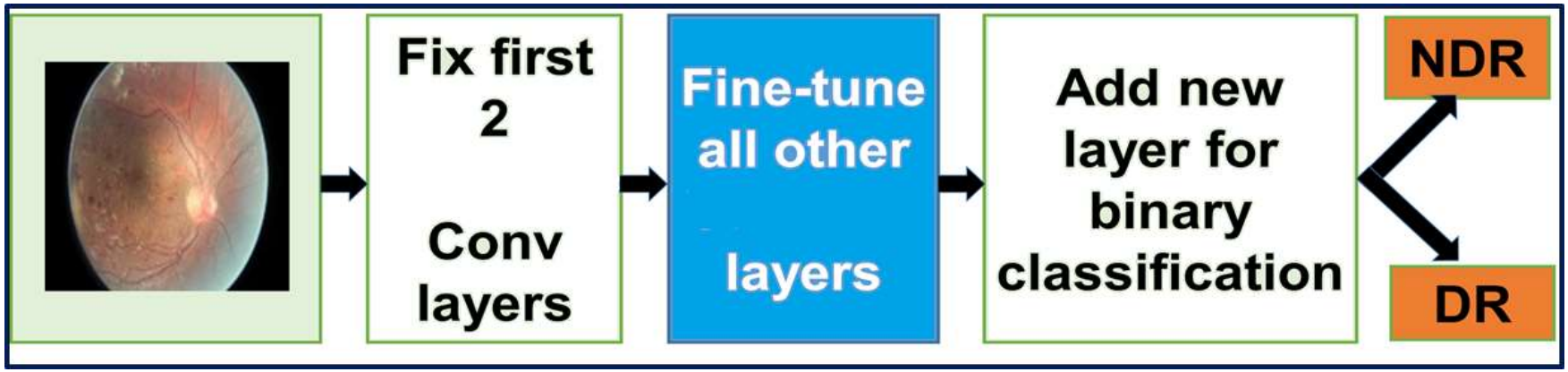
# Transfer Learning DCNN




```
net['loss3/classifier'] = DenseLayer(net['pool5/7x7_s1'],
                                     num_units=1000,
                                     nonlinearity=linear)
net['features'] = DenseLayer(net['loss3/classifier'],
                                     num_units=2,
                                     nonlinearity=linear)
net['prob'] = NonlinearityLayer(net['features'],
                                nonlinearity=softmax)

googlenet = cPickle.load(open(('%smodelzoo/blvc_googlenet.pkl' % persistency_dir),'rb'))
lasagne.layers.set_all_param_values(net['loss3/classifier'], googlenet['param values'])
return net, MEAN_VALUES.astype(np.int16)
```

*http://benanne.github.io/2015/03/17/plankton.html

# Fine-tuning GoogleNet

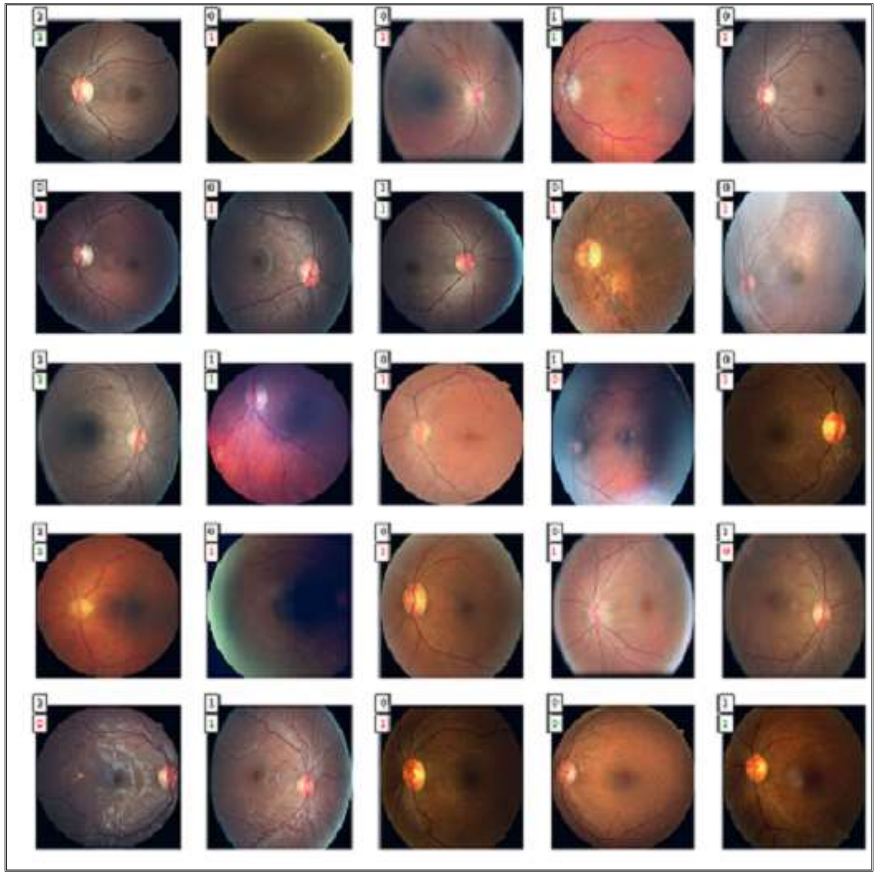



```
# Define the parameters to be adapted during training
# Note that we fine-tune all layers except for 'conv1_1' and 'conv1_2'
# If you want to fine-tune all layers,
# you may use: model_params = lasagne.layers.get_all_params(net['prob'], trainable=True)
model_params = [net['loss3/classifier'].W, net['loss3/classifier'].b,
                net['features'].W, net['features'].b]
feature_layer = net['loss3/classifier']
output_layer = net['prob']
```

# Training Fine-tuned DCNN model

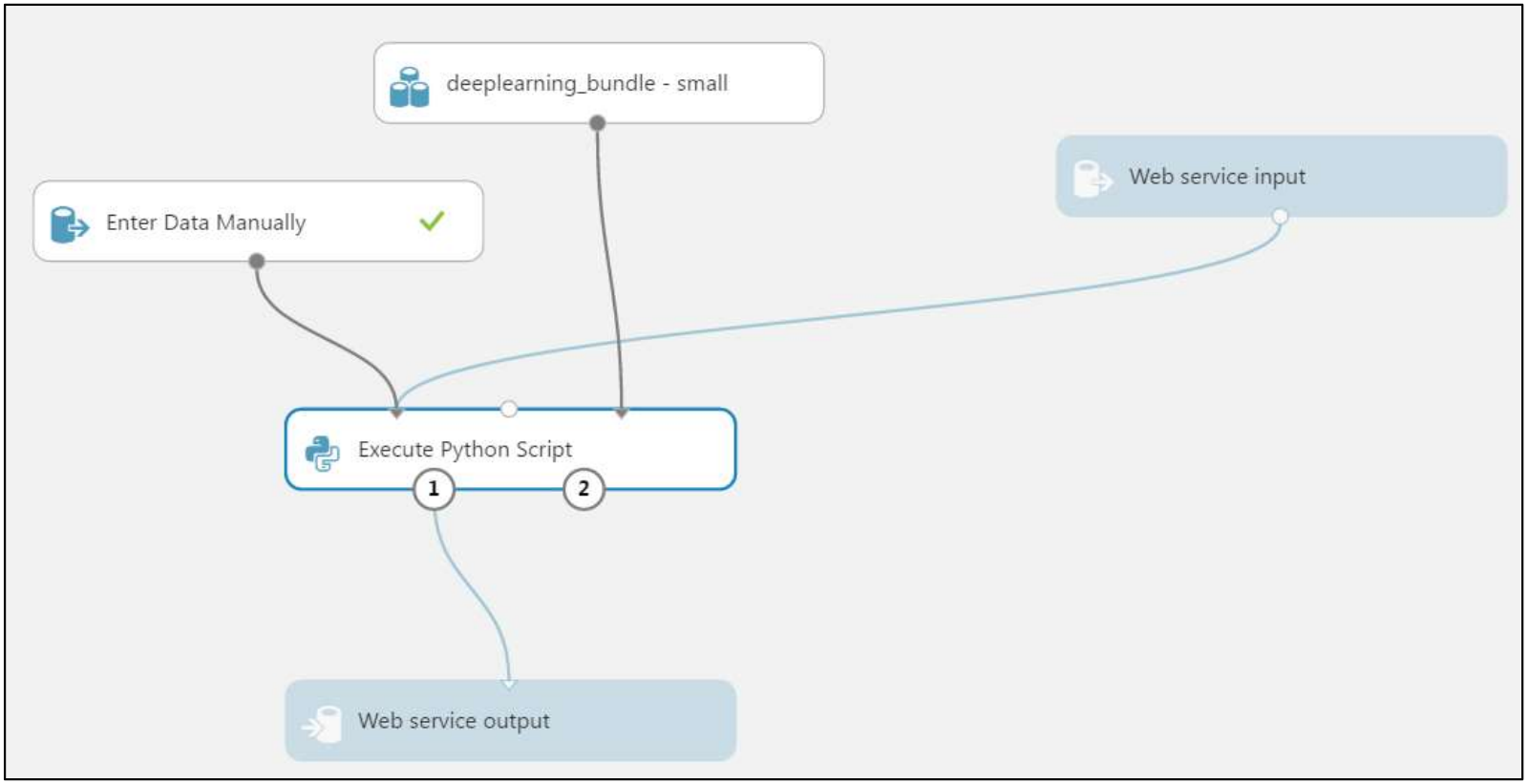
```
preds = np.empty((data_X_test.shape[0], n_repetitions))
for repetition in range(n_repetitions):
    print "##### Repetition %s #####" % repetition
    np.random.seed(repetition)
    random.seed(repetition)

    if do_train:
        print "Train"
        print repetition
        print n_epochs
        param_values = train_model(repetition, n_epochs=n_epochs)
        cPickle.dump(param_values, open('%sblvc_googlenet_%s.pkl' % (persistency_dir, repetition), 'wb'),
                     protocol=-1)
```

# Diabetic Retinopathy Prediction

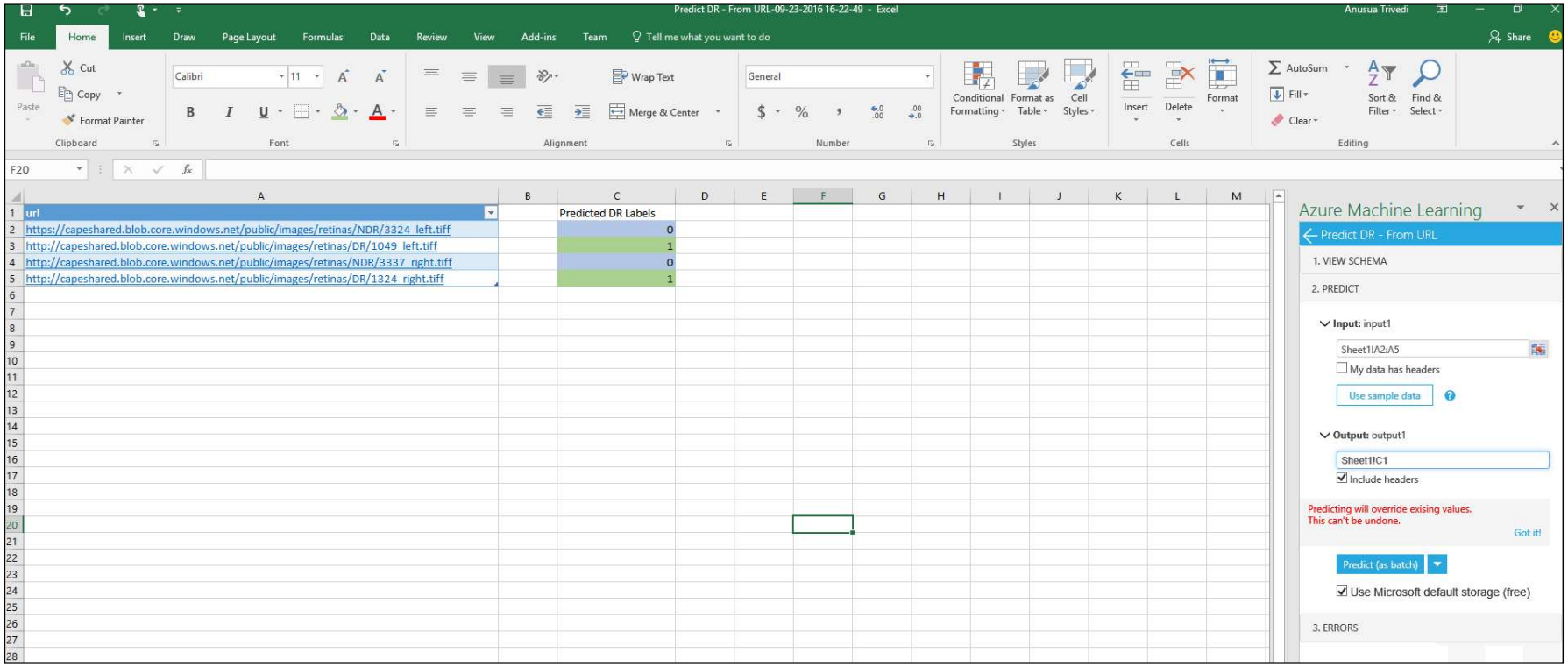


Epoch 449: Train cost 8.994, Train acc 0.607, val acc 0.798, val AUC 0.849

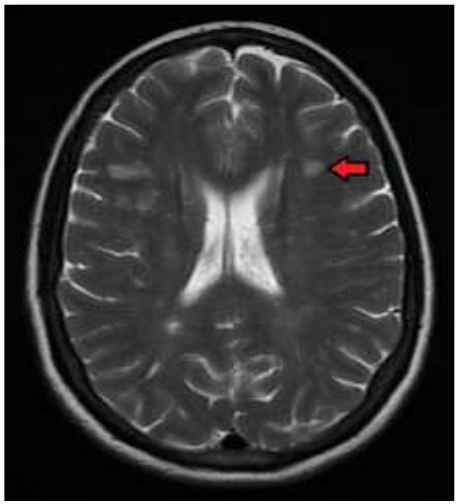
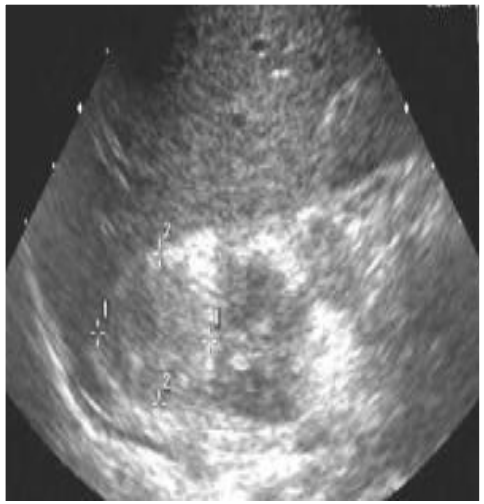
# Using GPU-Trained model in AML



# AML Web Service Batch-Test



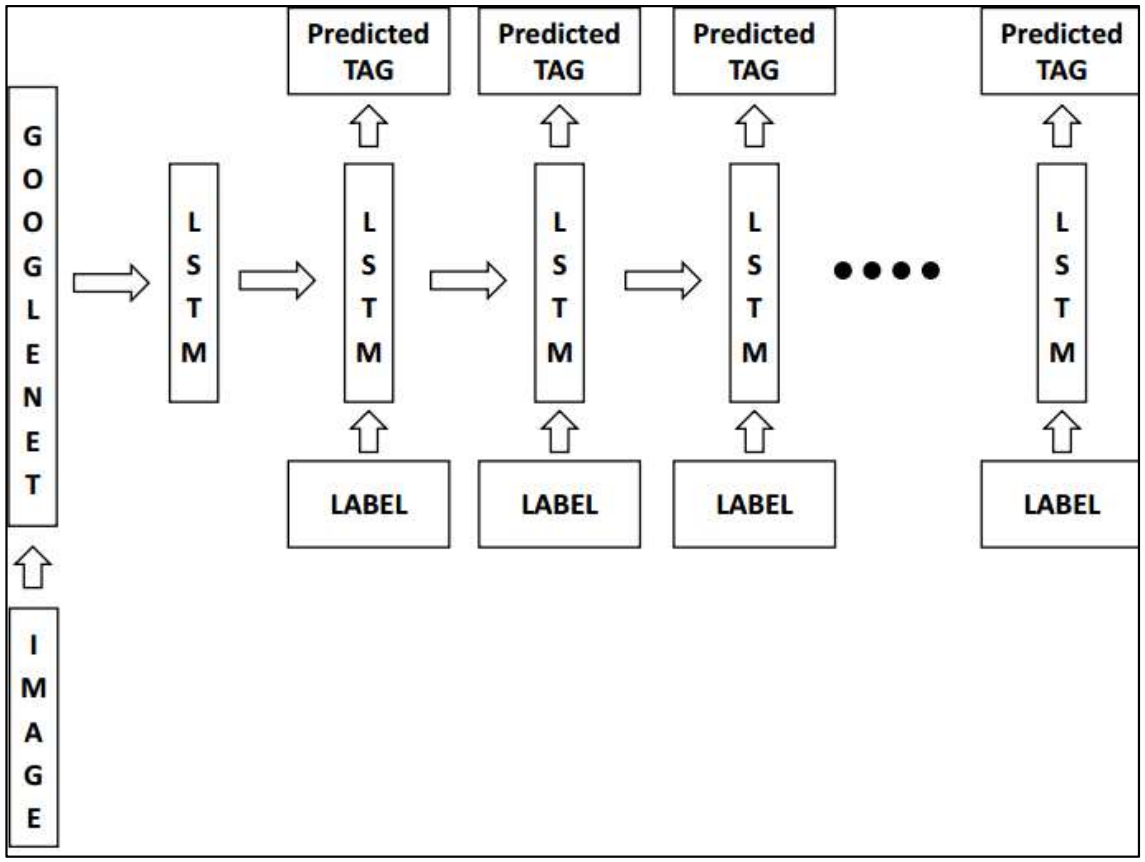
# Other Uses of this DCNN Model

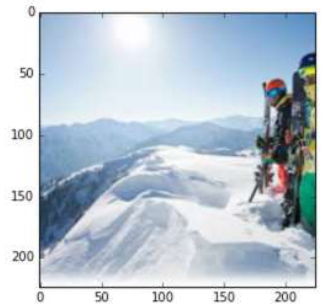
# Case Study IV: Image Caption Generation

- Used the ImageNet-trained DCNN and learn MS COCO image features through transfer learning and fine-tuning
- Used a Long short-term memory (LSTM) Recurrent Neural Network (RNN) on these learned image features for the image caption generation

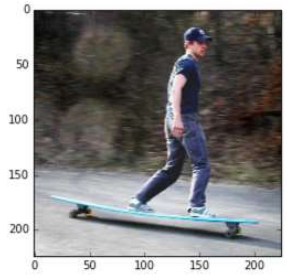
# Deep CNN-RNN Model



# Caption generation using fine-tuned CNN-RNN model



```
In [31]: p = get_cnn_features(cnn_im)
         CLASSES = pickle.load(open('blvc_googlenet.pkl'))['synset words']
         print(CLASSES[p.argmax()])

         alp

In [32]: x_cnn = get_cnn_features(cnn_im)

In [33]: # Sample some predictions
         for _ in range(5):
             print(predict(x_cnn))

         a man riding skis on a snow covered slope
         a skier is standing in the snow on a snowy day
         a person riding a snowboard down a snow covered hill
         a man riding a snowboard down a snow covered slope
         a man riding a snowboard down a snowy hill
```



```
In [52]: p = get_cnn_features(cnn_im)

         CLASSES = pickle.load(open('blvc_googlenet.pkl'))['synset words']

         print(CLASSES[p.argmax()])

         ski

In [53]: x_cnn = get_cnn_features(cnn_im)

         # Sample some predictions

         for _ in range(5):

             print(predict(x_cnn))

         a man riding a snowboard down a snow covered slope
         a man riding a skateboard down a hill
         a man riding a skateboard on a street
         a man riding a wave on a surfboard
         a man riding skis down a snow covered slope
```

# Re-usability of this CNN-RNN Model

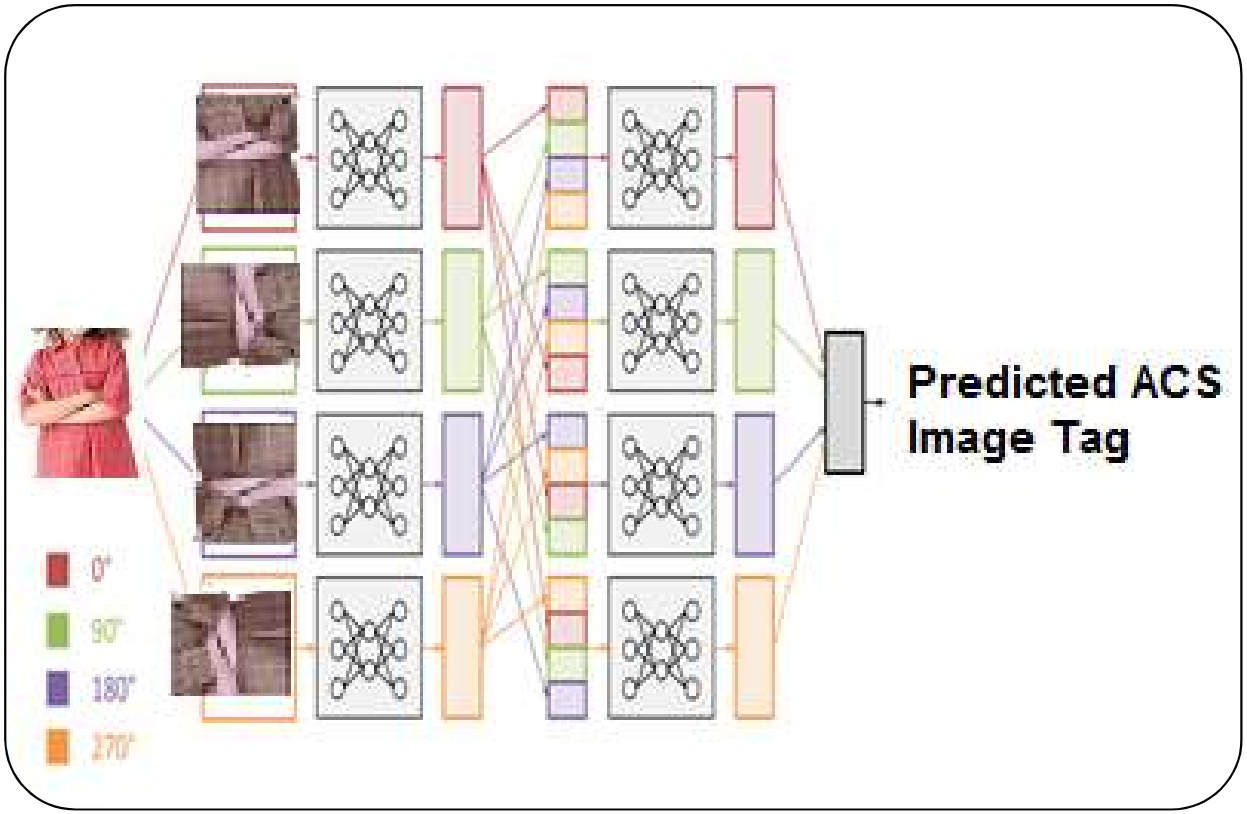
- Used CNN-RNN model for natural image caption generation
- Can use the same CNN-RNN model in a completely different domain

# Case Study V: Fashion Image Tag Prediction

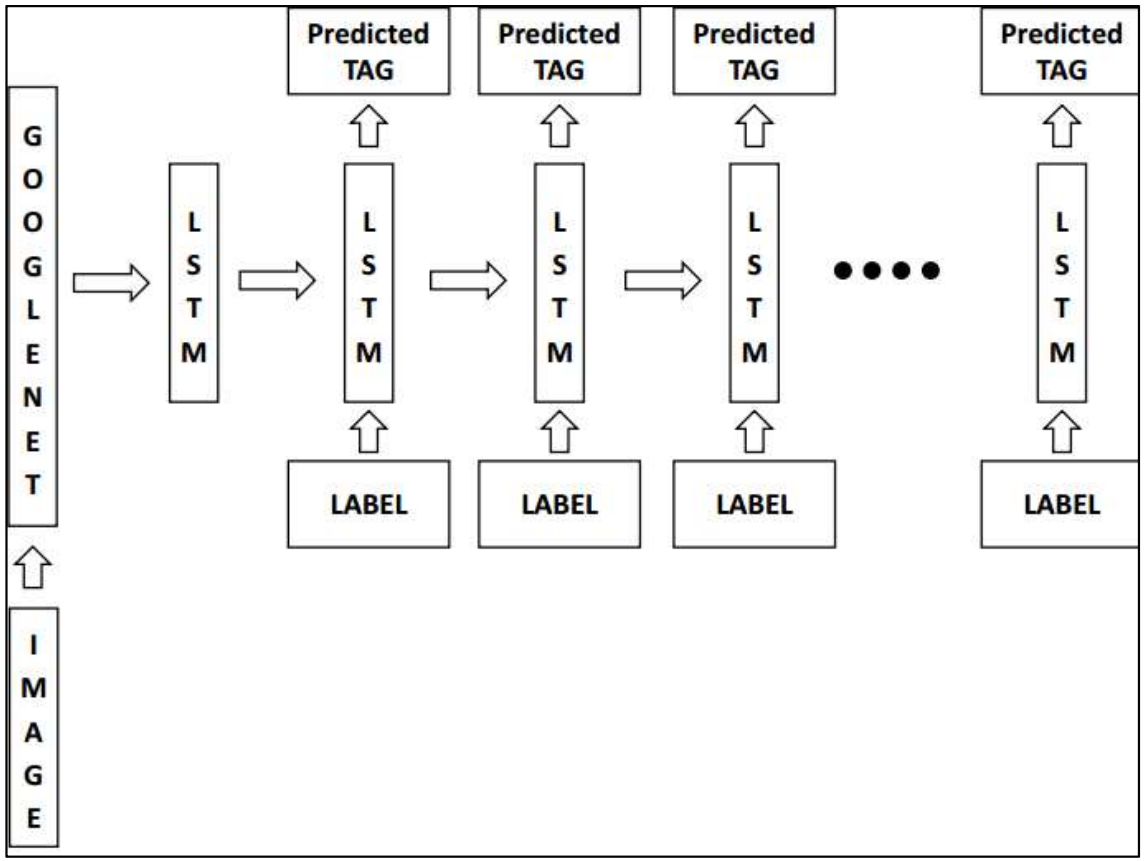
- We use the ImageNet-trained DCNN and learn Apparel Classification with Style (ACS) image features through transfer learning and fine-tuning

- Then we use a Long short-term memory (LSTM) Recurrent Neural Network (RNN) on the learned image features for the image caption generation

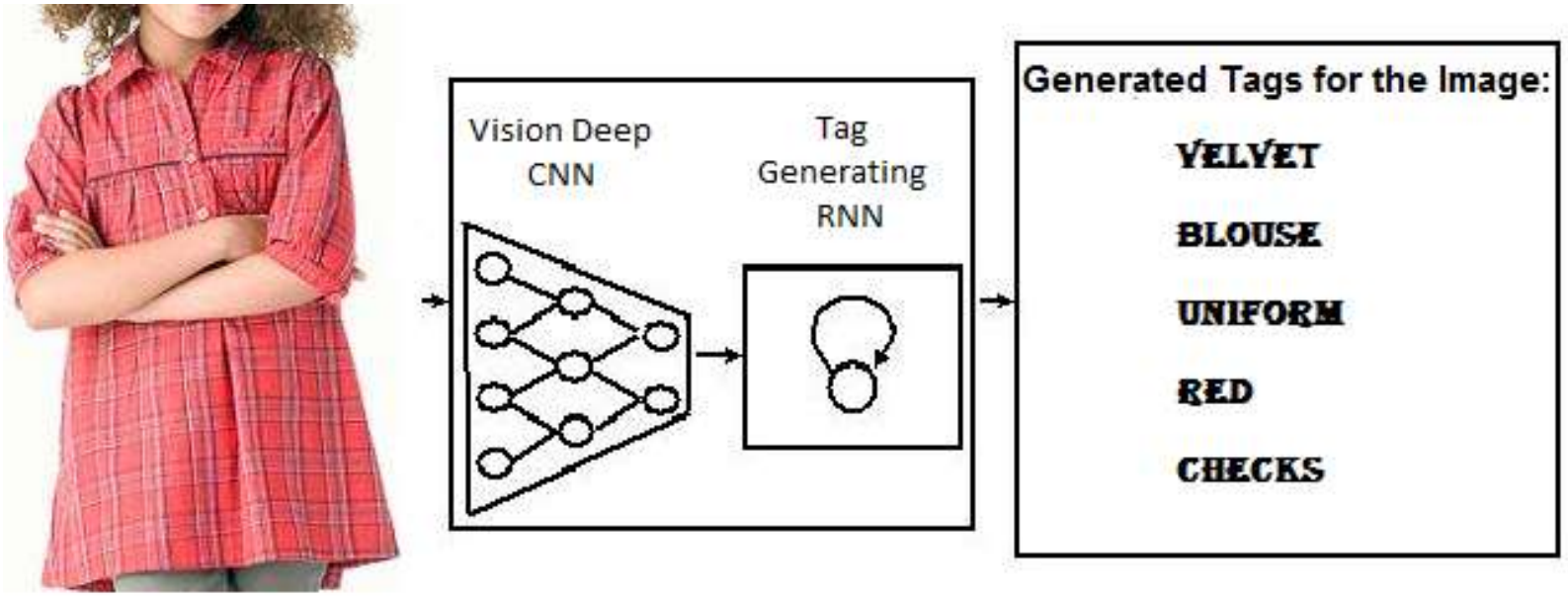
# Augmentation + Transfer Learning



# Using the same CNN-RNN Model



# ACS Images Tag Prediction



# Using fine-tuned CNN-RNN model


```
p = get_cnn_features(cnn_im)
CLASSES = pickle.load(open('blvc_googlenet.pkl'))['synset words']
print(CLASSES[p.argmax()])
```

```
crutch
```

```
x_cnn = get_cnn_features(cnn_im)
```

```
# Sample some predictions
for _ in range(5):
    if (predict(x_cnn)!=''):
        print(predict(x_cnn))
```

```
robe
```


```
p = get_cnn_features(cnn_im)
CLASSES = pickle.load(open('blvc_googlenet.pkl'))['synset words']
print(CLASSES[p.argmax()])
```

```
bow tie, bow-tie, bowtie
```

```
x_cnn = get_cnn_features(cnn_im)
```

```
# Sample some predictions
for _ in range(5):
    if (predict(x_cnn)!=''):
        print(predict(x_cnn))
```

```
robe
coat
uniform
```

# DL Framework: TensorFlow

- DL toolkit by Google (2016)
- Supports all common DL models
- Supports multi-GPU training
- Supports native distributed training
- Great Python Support

# Case Study VI: Transfer Learning on pre-trained AlexNet

**Create new layer, attached to `fc7`**

In [11]:

```
# Create new final layer
with graph.as_default():
    x = graph.get_operation_by_name('input').outputs[0]

    with tf.name_scope('transfer'):
        labels = tf.placeholder(tf.int32, [None])
        one_hot_labels = tf.one_hot(labels, 2)

        with tf.name_scope('cat_dog_final_layer'):
            weights = tf.Variable(tf.truncated_normal([4096, 2], stddev=0.001),
                                  name='final_weights')
            biases = tf.Variable(tf.zeros([2]), name='final_biases')
            logits = tf.nn.xw_plus_b(fc7, weights, biases, name='logits')

        prediction = tf.nn.softmax(logits, name='cat_dog_softmax')
        cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, one_hot_labels)
        loss = tf.reduce_mean(cross_entropy, name='cat_dog_loss')

        global_step = tf.Variable(0, trainable=False, name='global_step')
        inc_step = global_step.assign_add(1)

        cat_dog_variables = [weights, biases]
        train = tf.train.GradientDescentOptimizer(0.01).minimize(loss, global_step=global_step,
                                                                 var_list=cat_dog_variables)

    with tf.name_scope('accuracy'):
        label_prediction = tf.argmax(prediction, 1, name='predicted_label')
        correct_prediction = tf.equal(label_prediction, tf.argmax(one_hot_labels, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))


    init = tf.initialize_all_variables()
```

# Training on Cats & Dogs Images

**Train our model!**

In [37]:
```
for data_batch, label_batch in get_batch(32, train_data, 1, should_distort=True):
    data_batch = np.squeeze(data_batch)
    feed_dict = {x: data_batch, labels: label_batch}
    err, acc, step, _ = sess.run([loss, accuracy, inc_step, train],
                                 feed_dict=feed_dict)
    if step % 50 == 0:
        print("Step: {}\t Accuracy: {}\t Error: {}".format(step, acc, err))
```

```
Step: 50        Accuracy: 0.9375        Error: 0.0700903013349
Step: 100       Accuracy: 0.875         Error: 0.175362020731
Step: 150       Accuracy: 0.9375        Error: 0.150483578444
Step: 200       Accuracy: 0.9375        Error: 0.149619147182
Step: 250       Accuracy: 0.90625       Error: 0.124697074294
Step: 300       Accuracy: 0.90625       Error: 0.12353708595
Step: 350       Accuracy: 0.96875       Error: 0.187275096774
Step: 400       Accuracy: 1.0   Error: 0.0302645843476
Step: 450       Accuracy: 0.96875       Error: 0.108887523413
Step: 500       Accuracy: 1.0   Error: 0.0338761284947
Step: 550       Accuracy: 0.96875       Error: 0.0903983265162
Step: 600       Accuracy: 1.0   Error: 0.0455834493041
Step: 650       Accuracy: 0.90625       Error: 0.178182154894
Step: 700       Accuracy: 1.0   Error: 0.0333553180099
Step: 750       Accuracy: 0.90625       Error: 0.149356365204
Step: 800       Accuracy: 0.90625       Error: 0.187900155783
Step: 850       Accuracy: 0.96875       Error: 0.0794009119272
Step: 900       Accuracy: 1.0   Error: 0.0737376660109
Step: 950       Accuracy: 0.96875       Error: 0.148608088493
Step: 1000      Accuracy: 0.9375        Error: 0.0959873497486
Step: 1050      Accuracy: 0.9375        Error: 0.279115825891
```

# Validating on Cats & Dogs Images

**Validate**

In [38]:
```
num_correct = 0
total = len(valid_data)
i = 0
for data_batch, label_batch in get_batch(batch_size, valid_data, 1):
    feed_dict = {x: data_batch, labels: label_batch}
    correct_guesses = sess.run(correct_prediction,
                               feed_dict=feed_dict)
    num_correct += np.sum(correct_guesses)
    i += batch_size
    if i % (batch_size * 10) == 0:
        print('\tIntermediate accuracy: {}'.format((float(num_correct) / float(i))))
acc = num_correct / float(total)
print('\nAccuracy: {}'.format(acc))
```

In [39]:
```
check_accuracy(valid_data)
```

```
        Intermediate accuracy: 0.96
        Intermediate accuracy: 0.944
        Intermediate accuracy: 0.939333333333
        Intermediate accuracy: 0.9445
        Intermediate accuracy: 0.9448
        Intermediate accuracy: 0.945333333333
        Intermediate accuracy: 0.946285714286
        Intermediate accuracy: 0.945
        Intermediate accuracy: 0.945555555556
        Intermediate accuracy: 0.9452
        Intermediate accuracy: 0.944727272727
        Intermediate accuracy: 0.945666666667
        Intermediate accuracy: 0.945846153846
        Intermediate accuracy: 0.947142857143
        Intermediate accuracy: 0.946533333333

Accuracy: 0.953066666667
```

# Classifying Cats & Dogs

```
    feed_dict = {x: [image]}
    guess = sess.run(label_prediction, feed_dict=feed_dict)
    if guess[0] == 1:
        print('Guess: dog')
    else:
        print('Guess: cat')
    plt.imshow(image)
    plt.show()
```

In [43]:
```
spot_check()
```

```
Guess: cat
```

# Case Study VII: Similar Image Retrieval

**Create Nearest Neighbors model!**

In [11]:
```
nbrs = NearestNeighbors(n_neighbors=5, algorithm='ball_tree').fit(extracted_features)
```

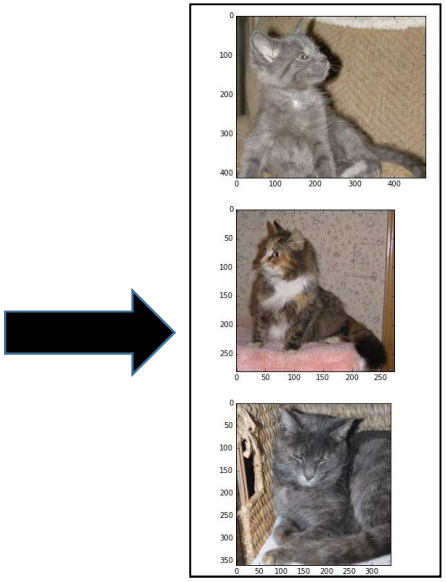
In [12]:
```
distances, indices = nbrs.kneighbors(extracted_features)
```

In [13]:
```
indices
```

Out[13]:
```
array([[   0, 2775,  304, 1876, 3865],
       [   1, 1845, 1391, 1882, 1608],
       [   2,  806, 1302,  544, 1213],
       ...,
       [4997, 3050, 3414, 4303, 1800],
       [4998,  584, 2363,  221, 2283],
       [4999, 4216, 1417, 3899,  935]])
```

**Print out the five nearest neighbors**

In [14]:
```
    for i, neighbor in enumerate(neighbors):
        image = ndimage.imread(filenames[neighbor])
        plt.figure(i)
        plt.imshow(image)
    plt.show()
```



# Case Study VIII: Similar Image Retrieval

# Computational Network Toolkit (CNTK)

- DL toolkit by Microsoft Research (2016)
- Supports almost all common DL models
- Supports multi-GPU training
- Supports native distributed training
- C++/Python Support
- Example: https://github.com/Microsoft/CNTK/wiki/Object-Detection-using-Fast-R-CNN

# Demos/Hands-on