Deep Learning with Python and Keras (TensorFlow)

Dr Amita Kapoor

Artificial Intelligence and Data Analytics Expert, Educator, Author, Expert in Remote Team Management. Founder, NePeur

FOUNTECH.AI®

knowledgehut upGrad

edureka!

simplilearn

DIGITTY

UDACITY

coursera education for everyone

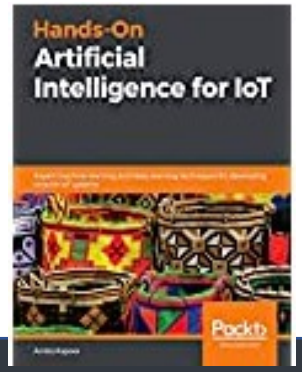
FEYNLABS.AI

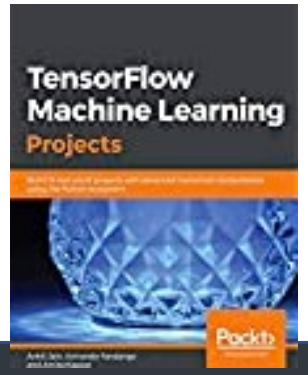
neuromatch

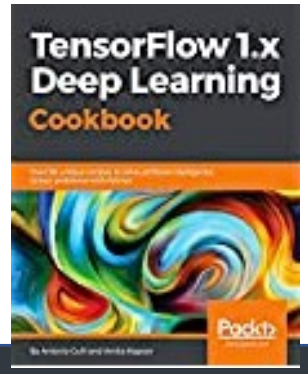
DeepSight AILabs

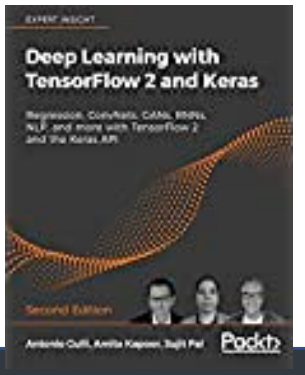
WOMEN WHO CODE

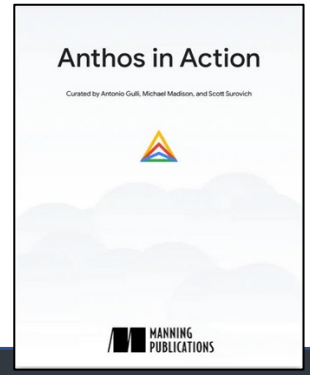
UNIVERSITY OF OXFORD











Outline

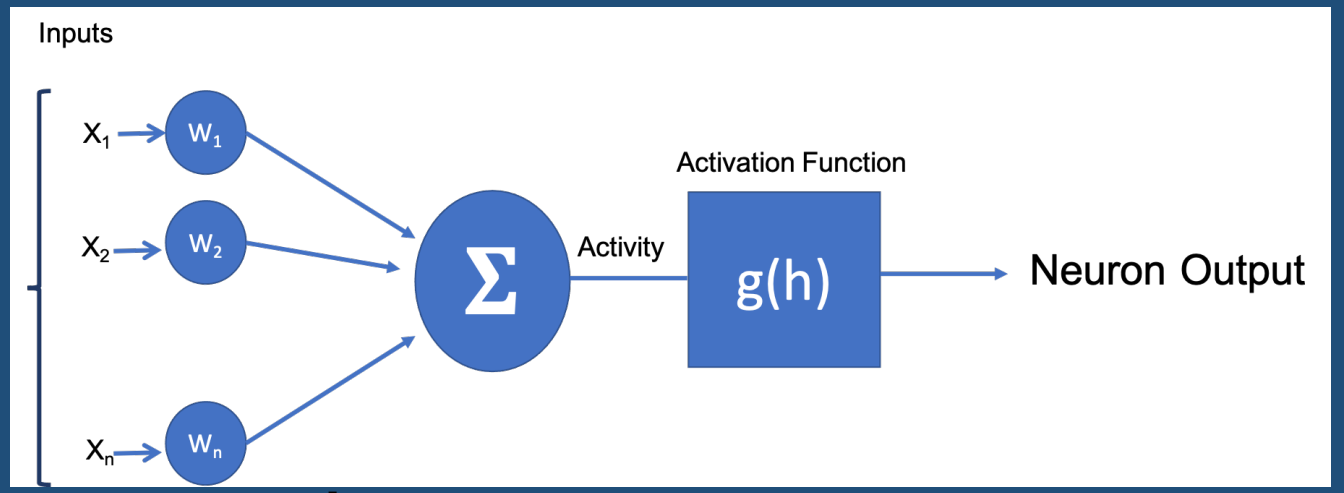
- Understanding the Keras API
- Complex model building and training
- Bringing it all together- TensorFlow Ecosystem

Deep Learning 101

Deep Learning and Neural Networks

Artificial Intelligence

Perceptrons

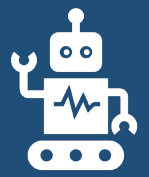


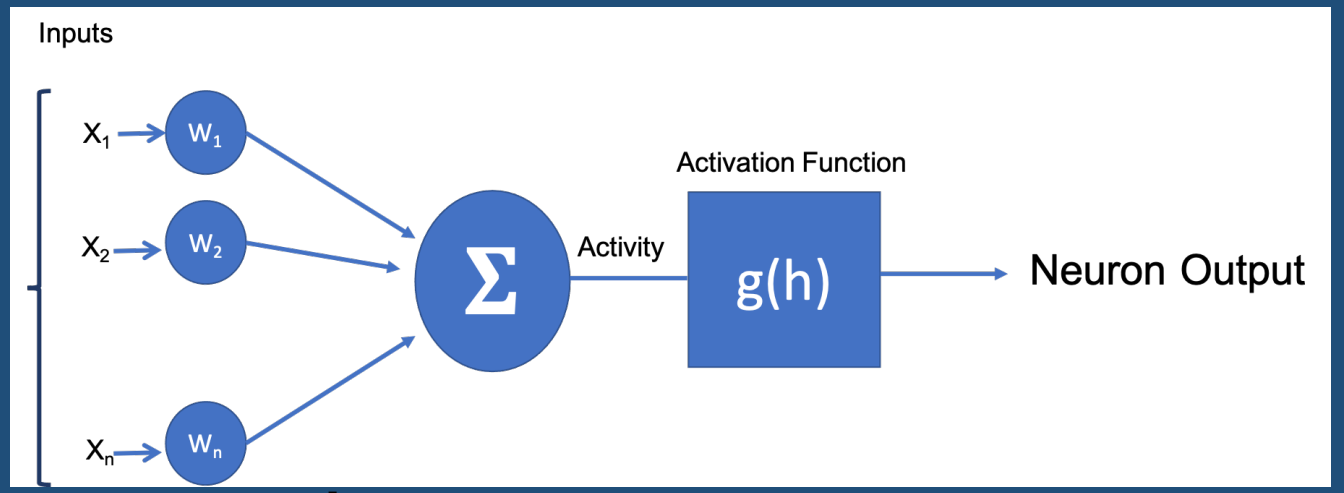
1950s 1980s 2011

Deep Learning and Neural Networks

Artificial Intelligence

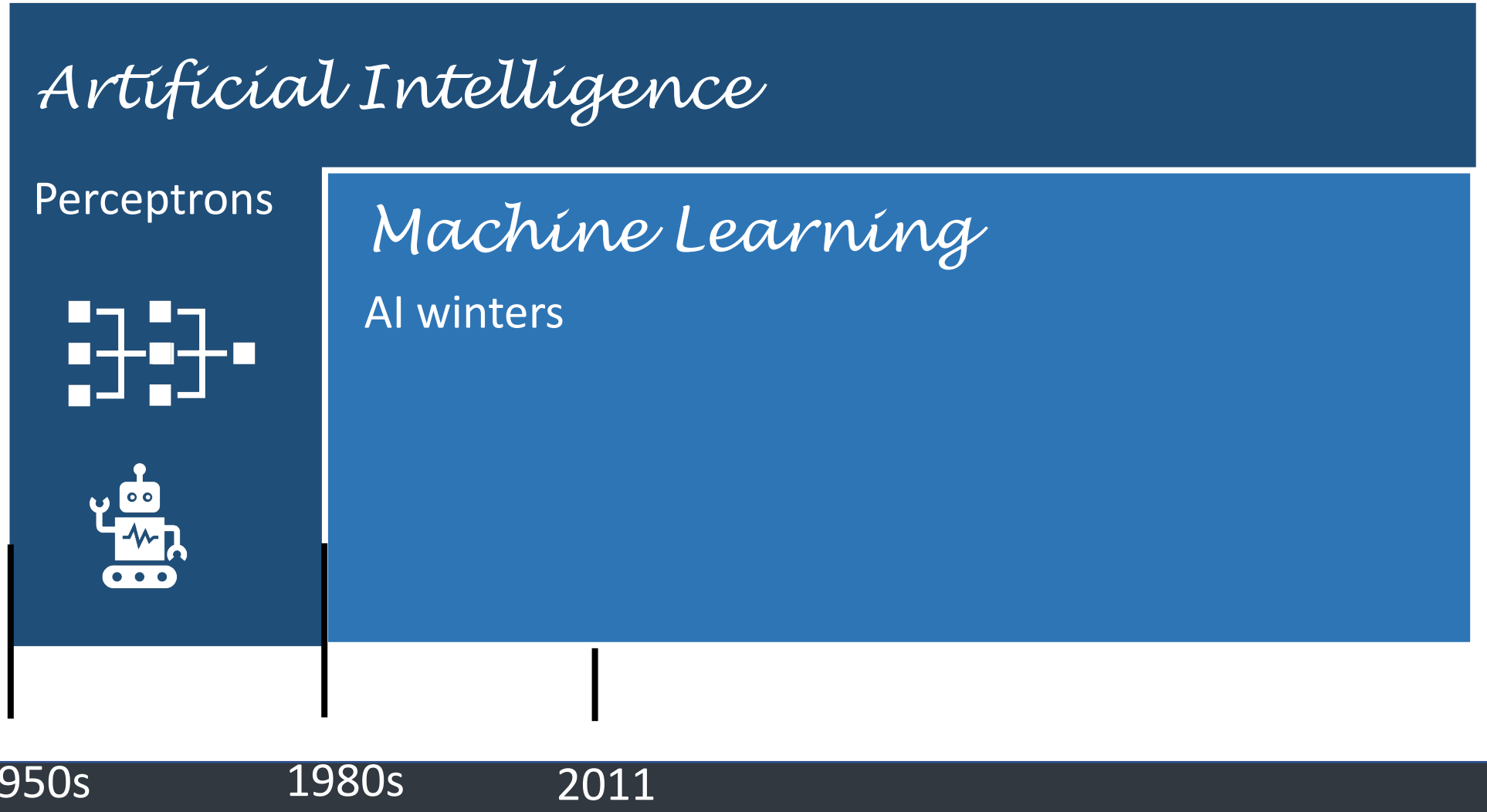
Perceptrons



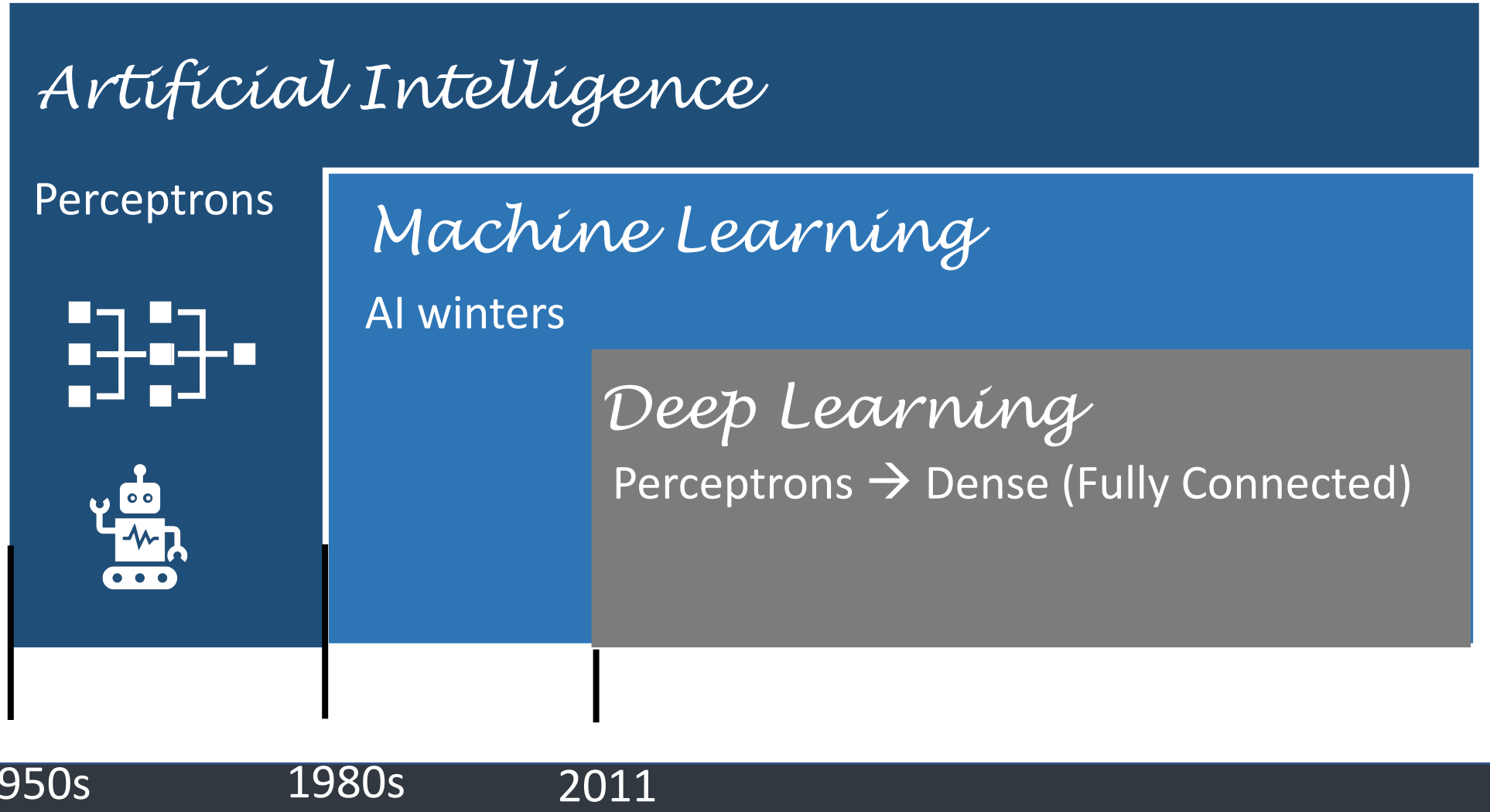


1950s 1980s 2011

Deep Learning and Neural Networks



Deep Learning and Neural Networks



Deep Learning and Neural Networks

- Dense Neural Networks
- Convolutional Neural Networks
- Recurrent Neural Networks
- Generative Adversarial Networks
- Auotoencoders
- Transformers

Types of Learning

Supervised

Data: (x, y)
x is data and y the label

Goal
Learn function mapping
f: $x \rightarrow y$

Unsupervised

Data: (x)
x is data, but no labels

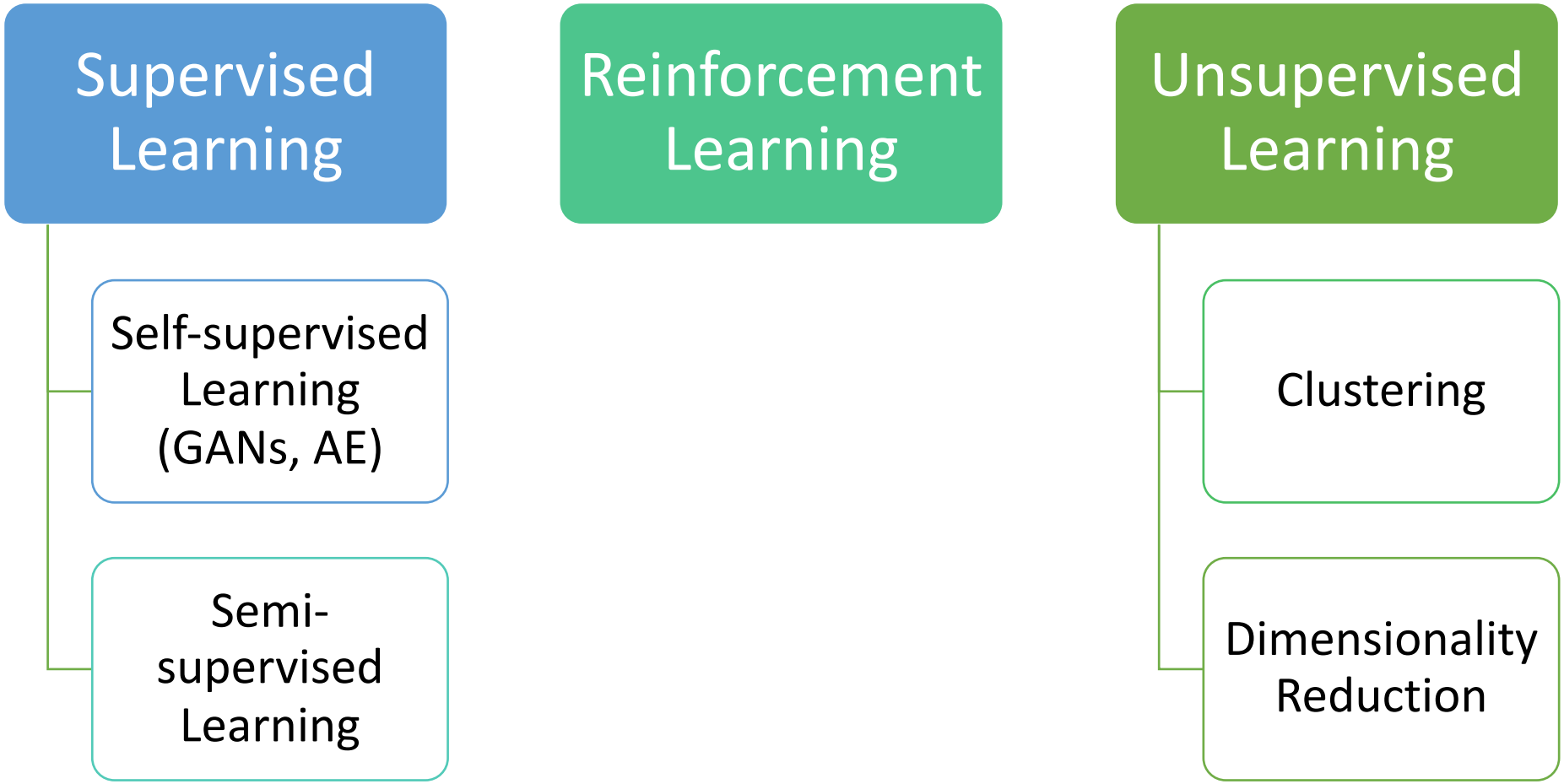
Goal
Learn underlying structure

Reinforcement

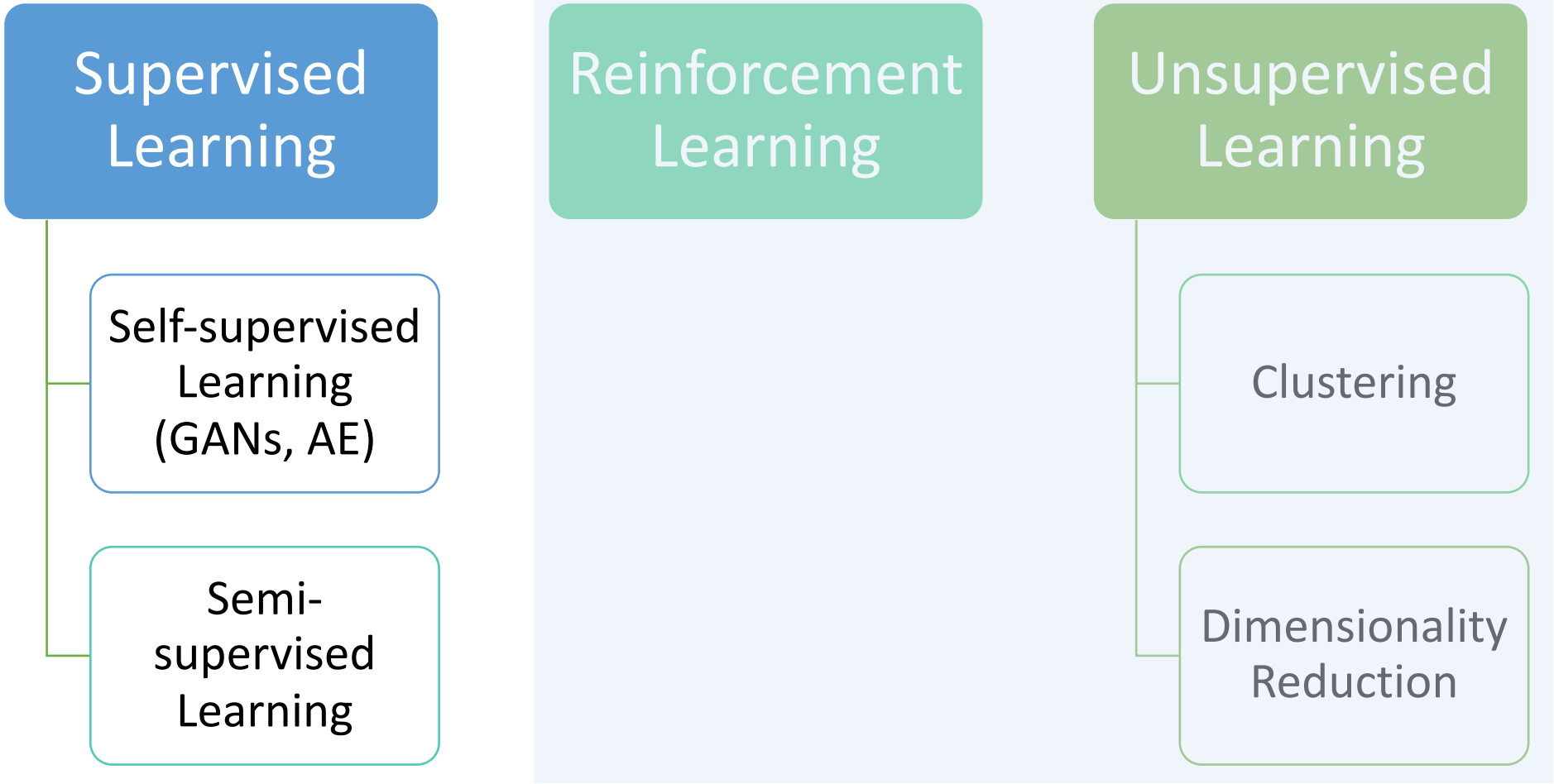
Data: (s, a)
s is state, a the $a$ction

Goal
Maximize future rewards

Deep Learning and Neural Networks



Deep Learning and Neural Networks

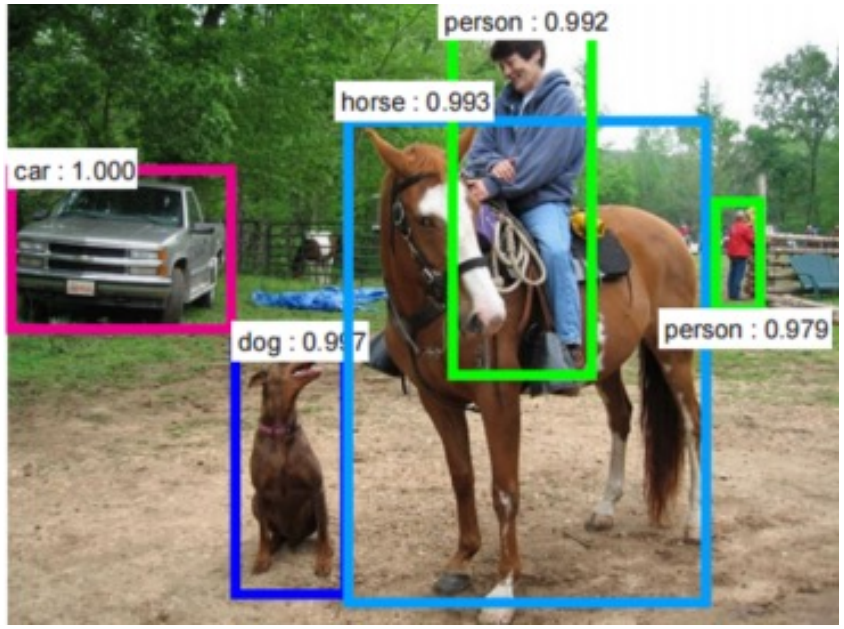


Deep Learning: Success Stories

- Automatic Colorization of Black and White Images
- Automatically Adding Sounds To Silent Movies
- Automatic Machine Translation
- Object Classification and Detection in Photographs
- Automatic Handwriting Generation
- Automatic Text Generation
- Automatic Image Caption Generation
- Automatic Game Playing

CNN

- A type of Neural Networks.
- Effective in areas like image recognition and classification.



History

- Was proposed first by Yann LeCun in 1988 for the recognition of handwritten digits (MNIST).

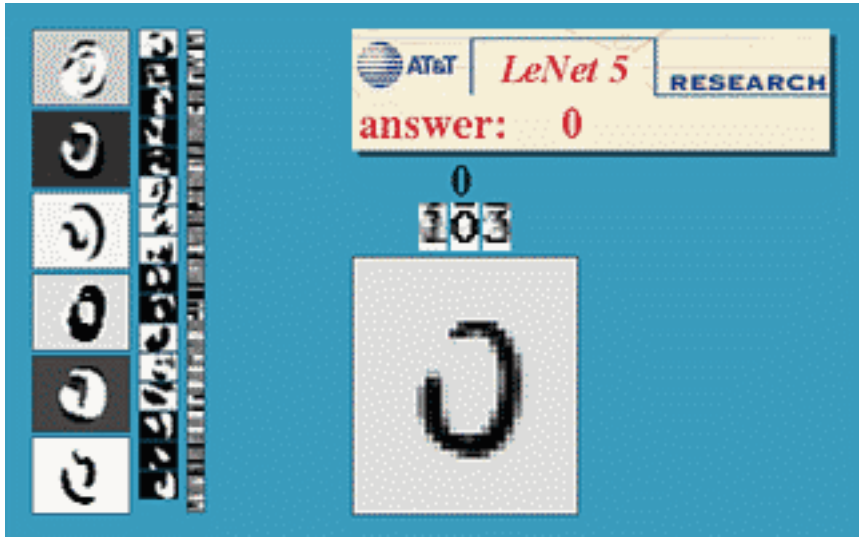


Image Source: http://yann.lecun.com/

CNN-Architecture

- CNN consists of four main parts:
  - Convolution
  - Non Linearity
  - Pooling
  - Classification

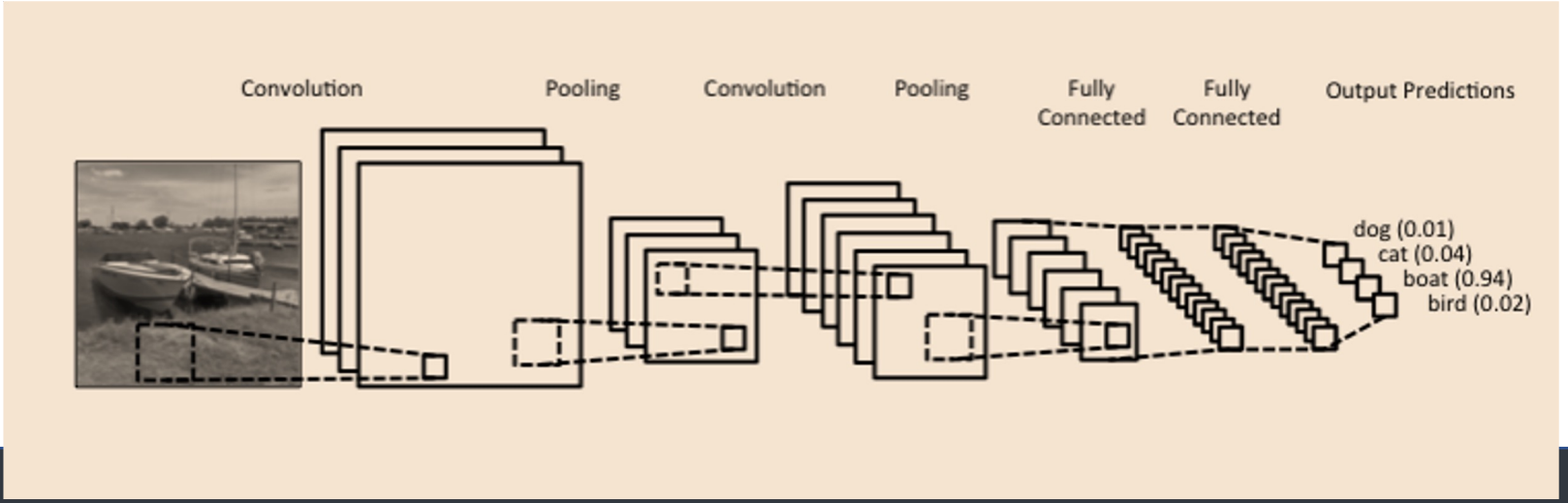
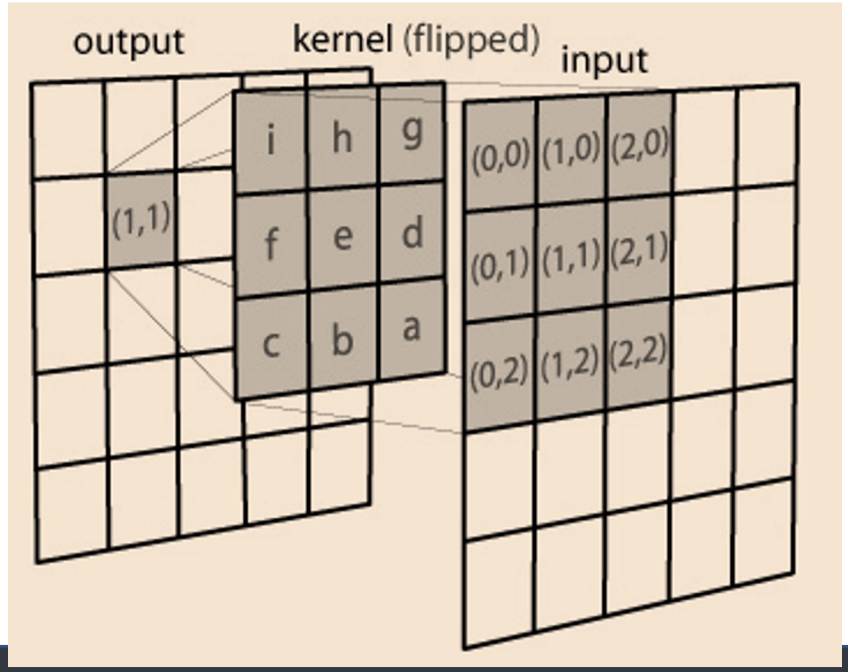


Image Source: https://www.clarifai.com/technology

Convolution and filters

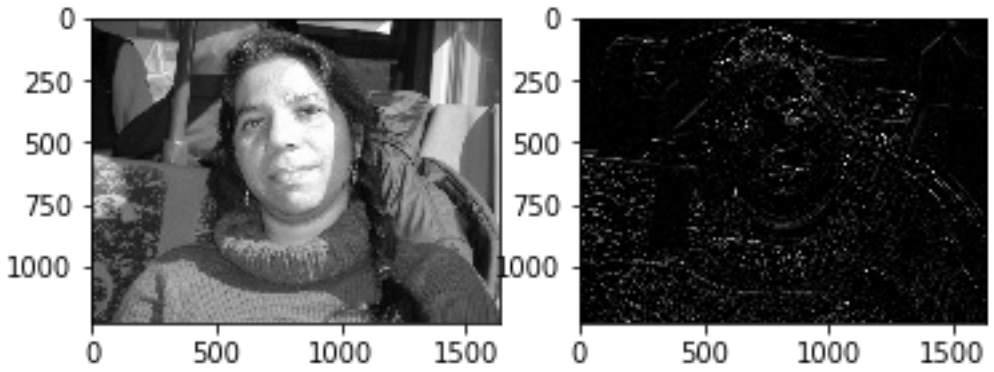
- Convolution is an old signal processing trick
- Process of adding each element of the image to its local neighbors, weighted by the kernel (filter).
- Traditionally it involves flipping both the rows and columns of the kernel and then multiplying locally similar entries and summing.

a	b	c
d	e	f
g	h	i



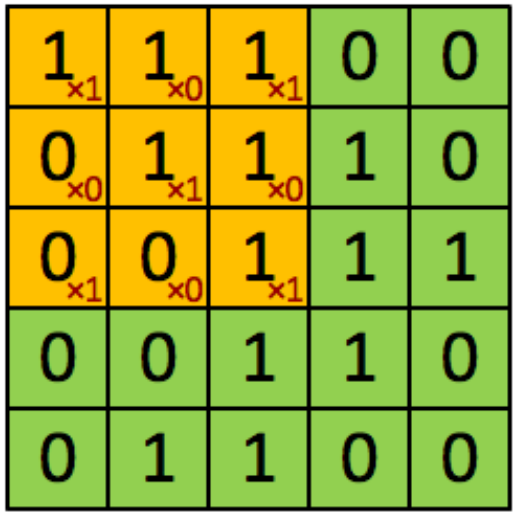
Convolution and filters

```
kernel = np.array([[ 1,  2,  1],
                   [ 0,   0,   0],
                   [-1,  -2,  -1]])
k2 = np.flip(np.fliplr(kernel),0)
filtered = cv2.filter2D(src=image, kernel=k2, ddepth=-1)
plt.subplot(121)
plt.imshow(image, cmap='gray')
plt.subplot(122)
plt.imshow(filtered, cmap='gray')
```



Convolution in CNN

- No flip is needed



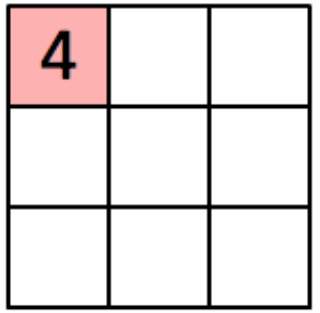


Image Source: http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution/

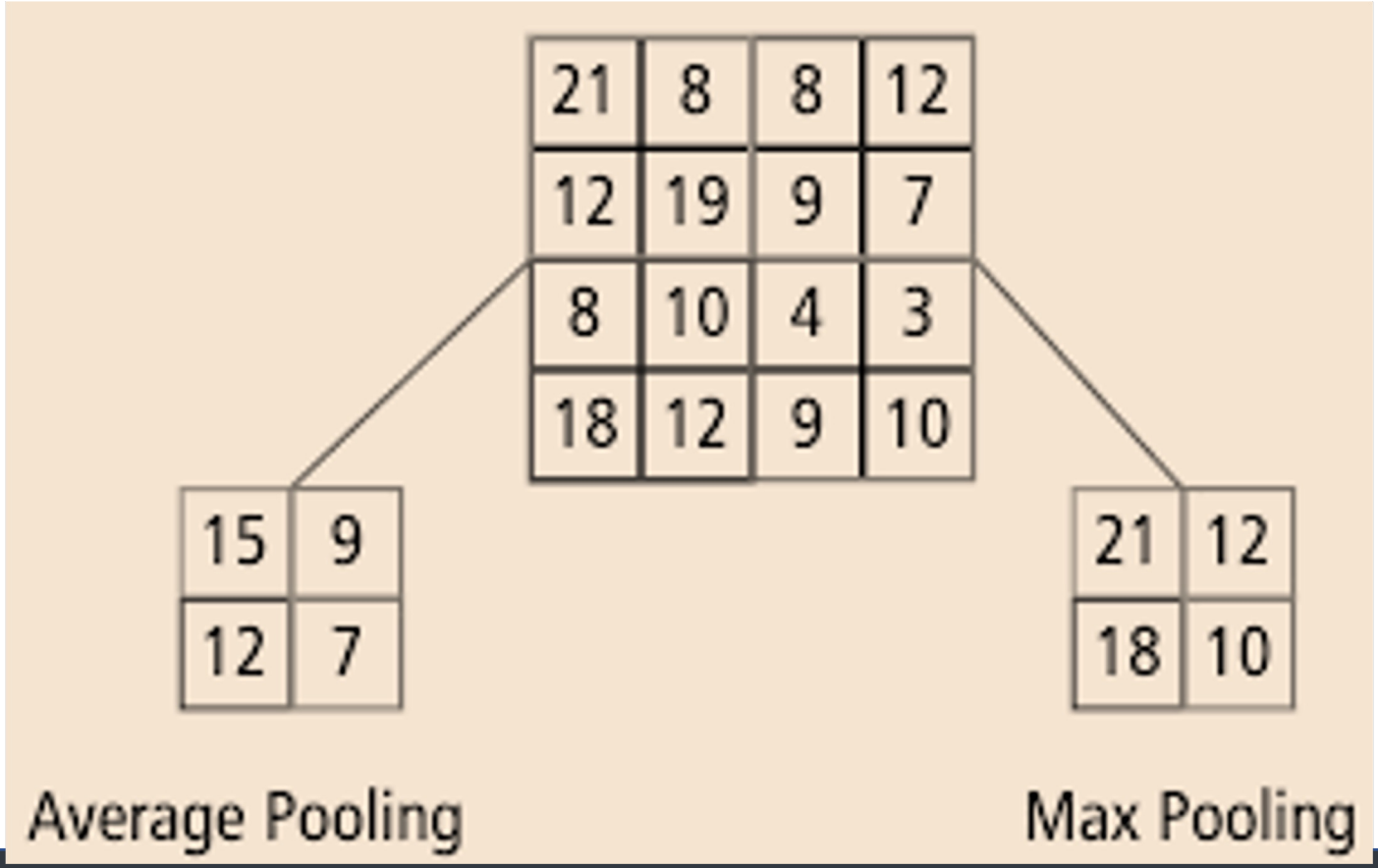
Convolution in CNN

Convolution provides three important ideas that help improve machine learning systems

1. Sparse interactions: kernel smaller than input
2. Parameter sharing: The network has

3. Equivariant representations: parameter sharing causes

Average/Max Pooling



Stride and padding

- Stride: the step of the convolution operation.
- It defines the shift in filter on an image at each step.
- When the stride is 1 then we move the filters one pixel at a time.
- It is convenient to pad the input volume with zeros around the border.
- The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes.
- Both stride and padding are hyperparameters

Stride and padding

- The size of the output image is affected by stride size and by padding:

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

n_{in}: number of input features
n_{out}: number of output features
k: convolution kernel size
p: convolution padding size
s: convolution stride size

Convolution

Max Pooling

A new image

Convolution

Max Pooling

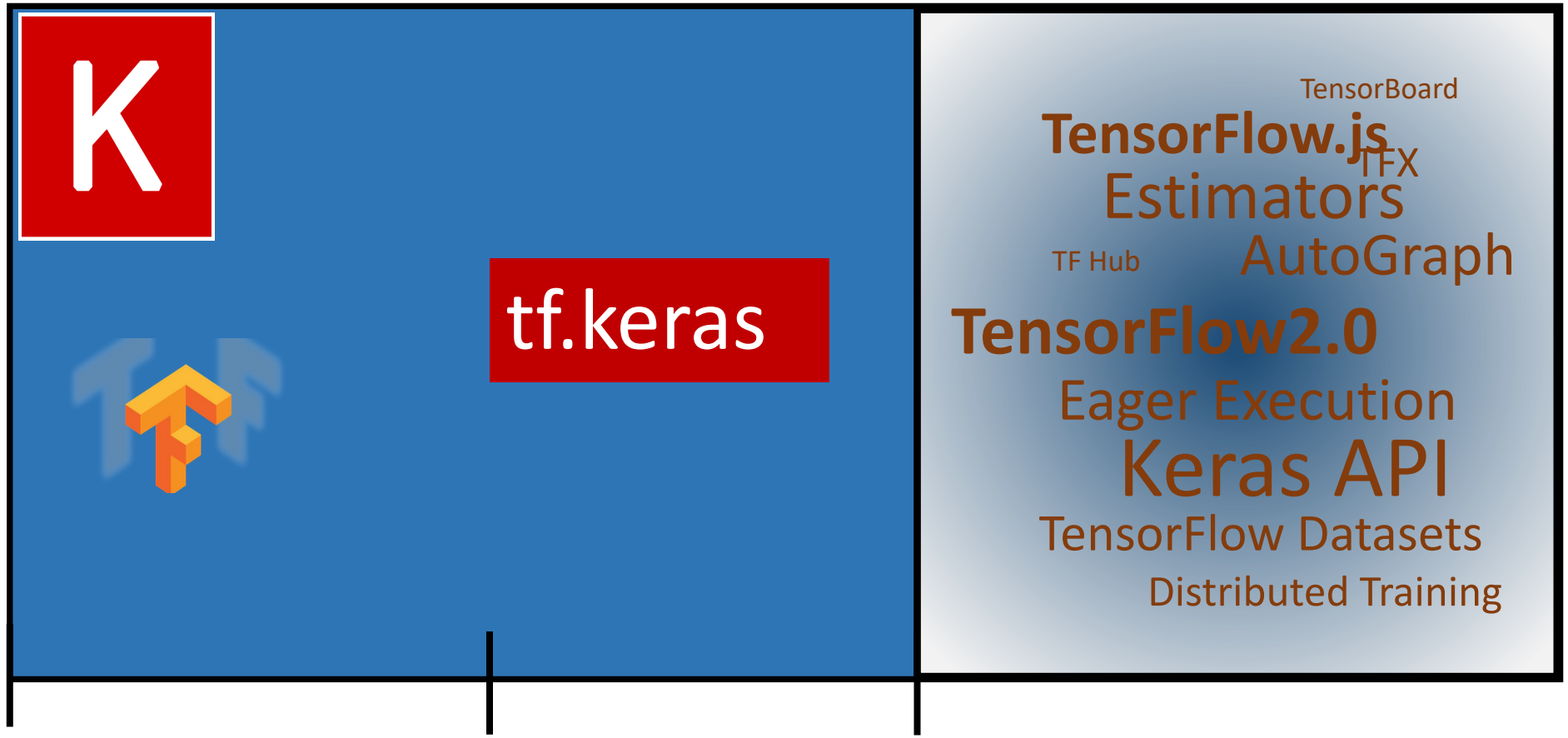
A new image

Flattened

Fully Connected
Feedforward network

Cat/Dog …

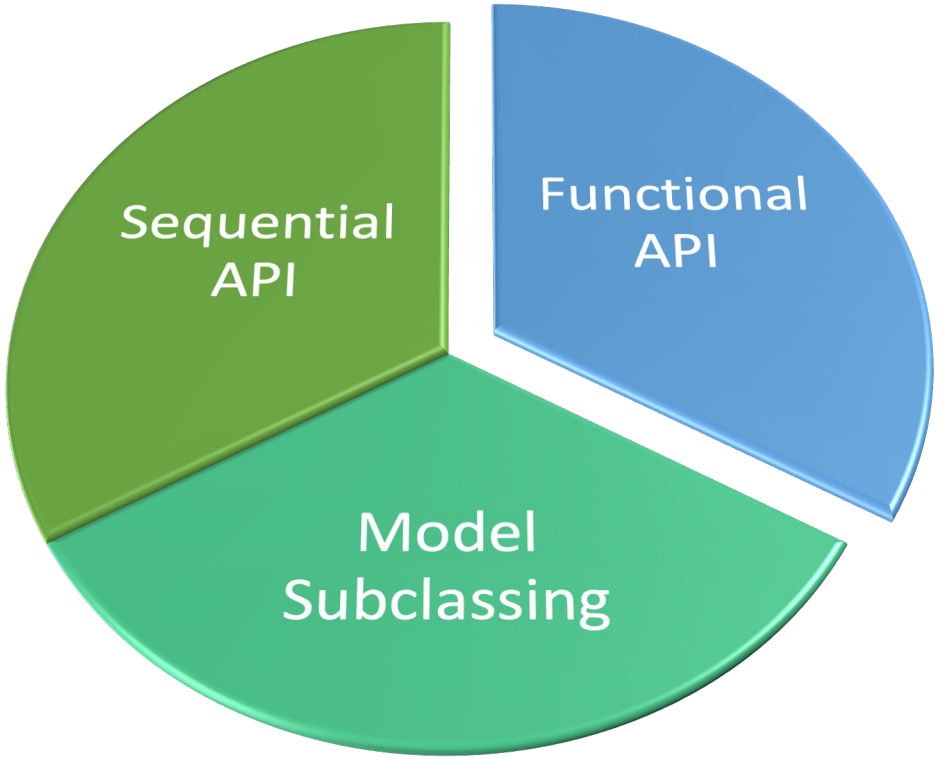
TensorFlow-Keras: Journey



Understanding the Keras API

TensorFlow Keras API

- Three different programming models
  - Sequential API
  - Functional API
  - Model Subclassing
- You can mix and match the three styles according to your needs



Sequential API

Sequential API

- Simple and Intuitive
- Can think of it as stack of layers
- Add layers one by one in a sequential fashion
- Layers Supported in tensorflow.keras.layers
  - Dense
  - Convolution
  - RNN, LSTM, GRU
  - Dropout
  - Batch Normalization
  - And many more...

Sequential API- How to?

1. Start with Sequential class available in *tensorflow.keras.models*
2. Define input dimensions in the first layer using *input_shape* parameter (Optional)
3. Add the layers you want one by one in the order you require using add method
4. And output dimensions in the last layer

```
from tensorflow.keras.models import Sequential
my_model = Sequential()

my_model.add(Dense(4,input_shape=(100,)))
my_model.add(Dense(122))
my_model.add(Dense(10))
```

Weight Initializations

- Choosing the right range and method for weight initialization is crucial when building a neural n
- Goal of weight initialization is to prevent layer activation outputs from exploding or vanishing during the training of the DL technique.
- Training the network without a useful weight initialization can lead to a very slow convergence or an inability to converge network.

Weight Initializations

- Using *kernel_initializer*
  - **random_uniform**: Weights are initialized to uniformly random small values in the range -0.05 to 0.05.
  - **random_normal**: Weights are initialized according to a Gaussian distribution, with zero mean and a small standard deviation of 0.05.
  - **zero**: All weights are initialized to zero.

```
model.add(keras.layers.Dense(NB_CLASSES, input_shape=(RESHAPED,),
                             kernel_initializer='zeros', name='dense_layer', activation='softmax'))
```

Full list: https://www.tensorflow.org/api_docs/python/tf/keras/initializers.

Sequential API - compile

- Once the model is made
- Use compile to specify loss, metrics and optimizer

```
model.compile( optimizer=K.optimizers.RMSprop(), # Optimizer
# Loss function to minimize
loss=K.losses.SparseCategoricalCrossentropy(),
# List of metrics to monitor
metrics=[K.metrics.SparseCategoricalAccuracy()], )
```

Sequential API - compile

Losses	Metrics	Optimizer
• MeanSquaredError() • KLDivergence() • CosineSimilarity() etc.	• AUC() • Precision() • Recall() etc.	• SGD() (with or without momentum) • RMSprop() • Adam() etc.

Training the model- fit()

- Use *fit* to train the model
- The returned history object holds a record of the loss values and metric values during training.

```
history = model.fit( x_train, y_train, batch_size=64, epochs=2,
    # We pass some validation for
    # monitoring validation loss and metrics
    # at the end of each epoch
    validation_data=(x_val, y_val), )
```

Sequential API - Evaluation and Prediction

- Evaluate the model on the test data using *evaluate*
- Generate predictions (probabilities -- the output of the last layer) on new data using *predict*

```
results = model.evaluate(x_test, y_test, batch_size=128)
predictions = model.predict(x_test[:3])
```

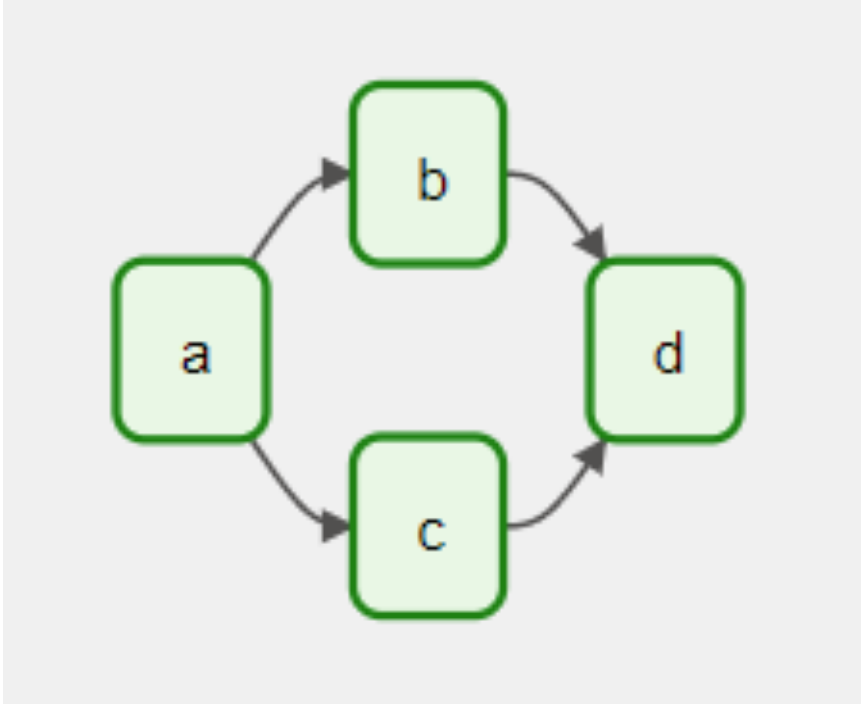
Sequential API - When not to use

- Model has multiple inputs and outputs
- Any layer has multiple inputs or multiple outputs
- Need to share a layer
- Have a non-linear topology.

Functional API

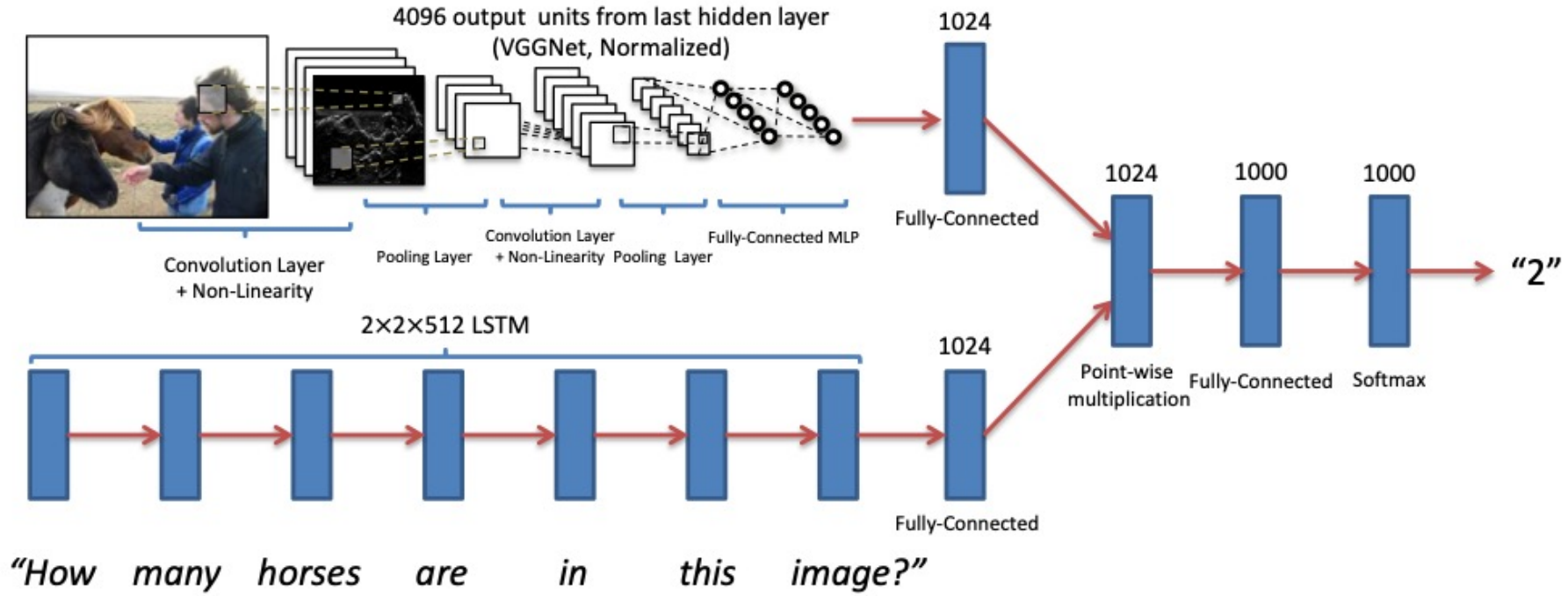
Functional API

- If instead of stack your model is a **DAG**
- Multiple Inputs
- Multiple Outputs
- Shared layers
- Residual Connections with non-sequential flow



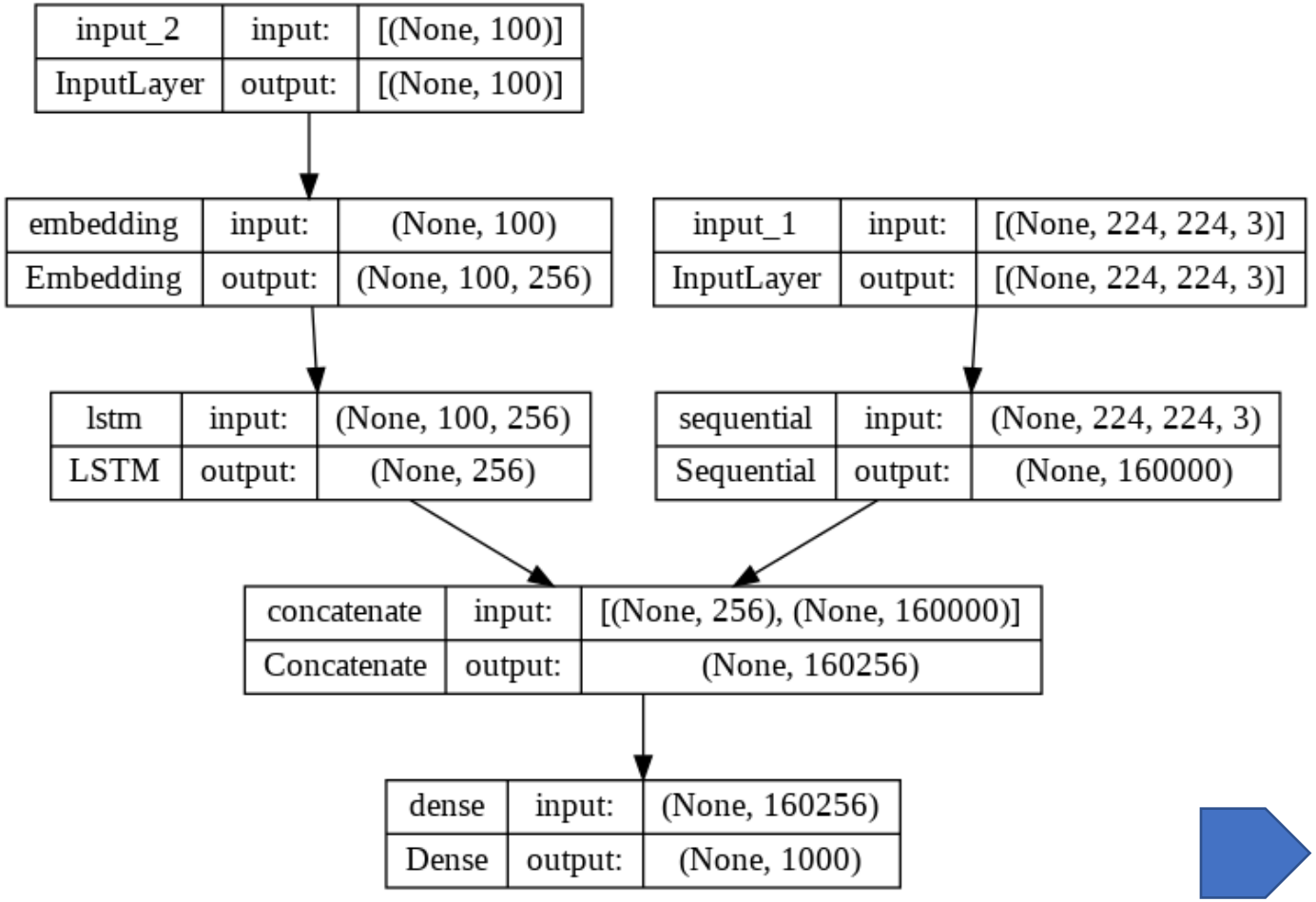
Functional API- Multiple Inputs

- Visual Question Answering (VQA): ICCV- 2013



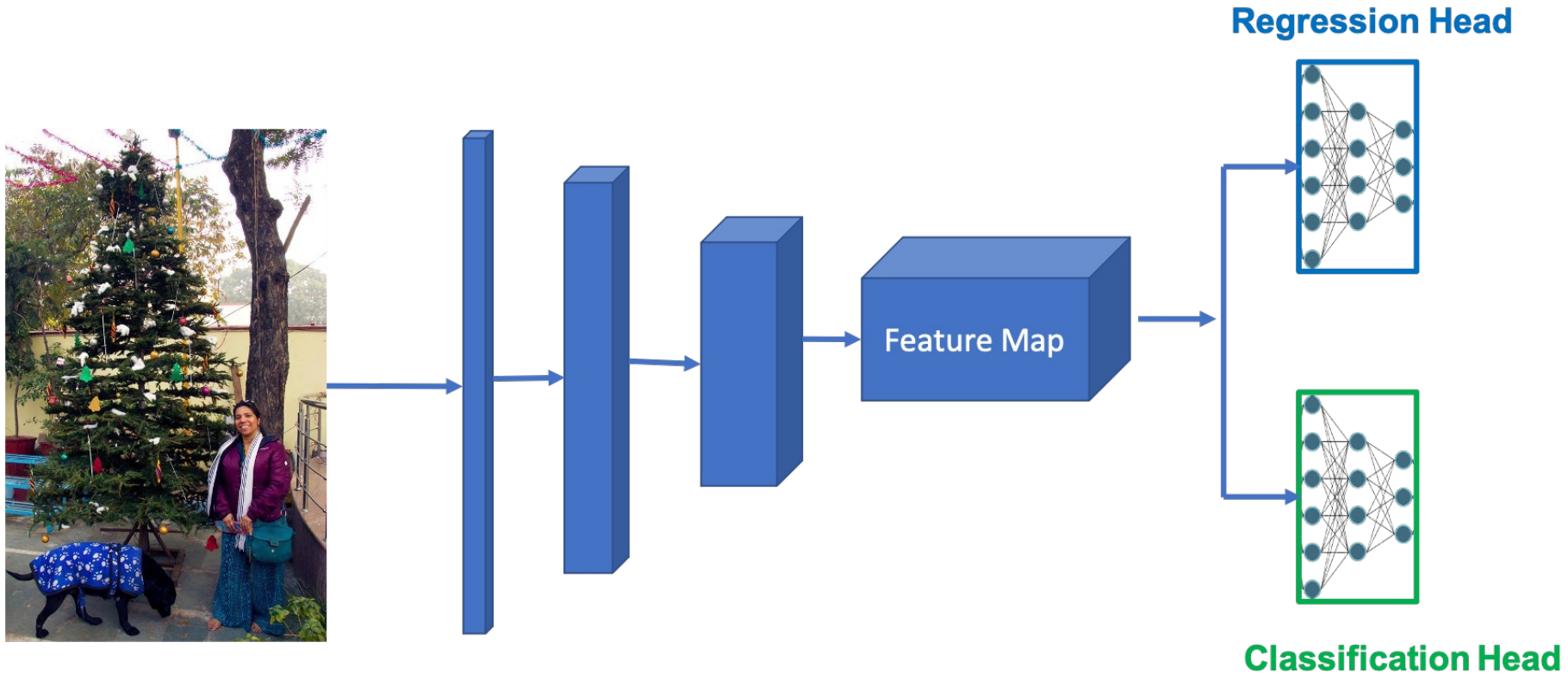
Functional API- Multiple Inputs

- VQA:
- Model with a combination of CNN for images and RNN for text.

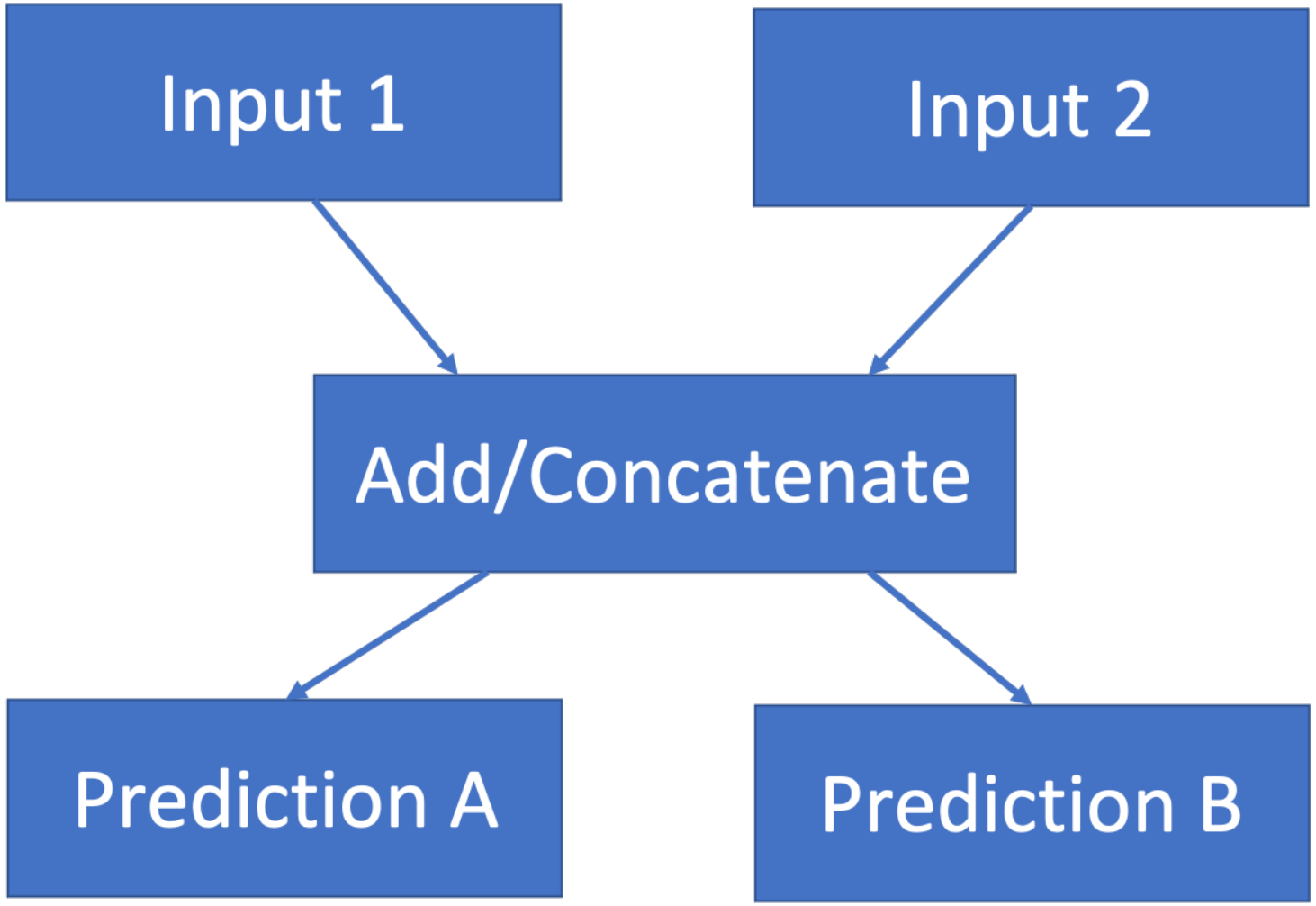


Functional API-Multiple outputs

- Image Classification and Localization



Functional API - Shared and Reusable layers

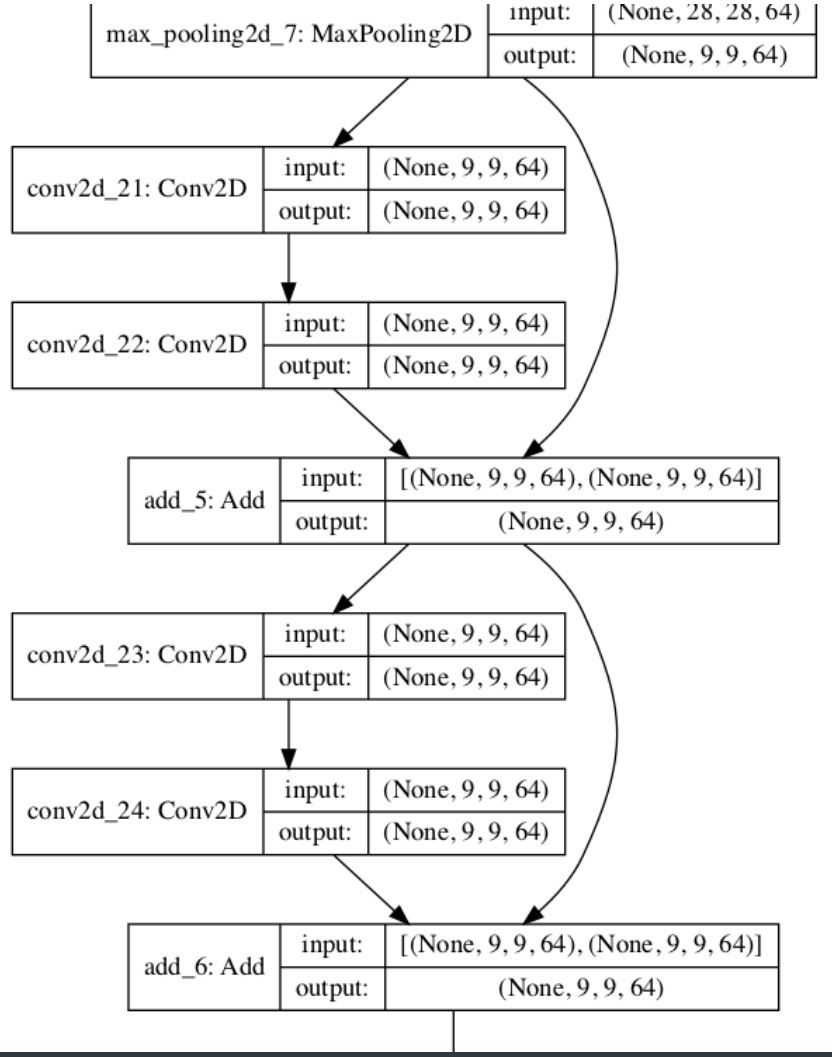


Functional API - Shared and Reusable layers

```
# Embedding for 500 unique words mapped to 128-dimensional vectors
shared_embedding = Embedding(500, 128)

# Variable-length sequence of integers
text_input_a = Input(shape=(None,), dtype="string")

# Variable-length sequence of integers
text_input_b = Input(shape=(None,), dtype="string")

# Reuse the same layer to encode both inputs
encoded_input_a = shared_embedding(text_input_a)
encoded_input_b = shared_embedding(text_input_b)

prediction_a = Dense(5)(encoded_input_a)
prediction_b = Dense(10)(encoded_input_b)

model =  Model(inputs = [text_input_a, text_input_b], outputs= [prediction_a,
prediction_b], name="shared_embedding")
model.summary()
```

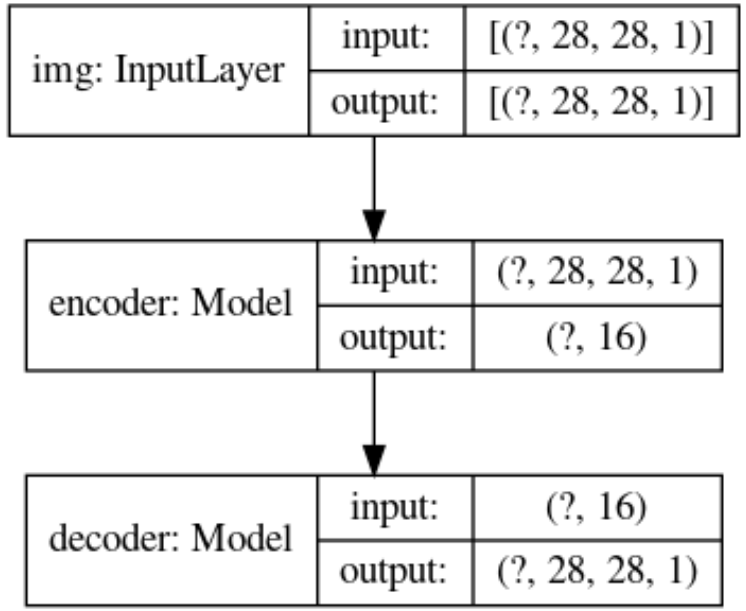
Functional API- Residual connections

```
inputs = Input(shape=(32, 32, 3), name="img")
x = Conv2D(32, 3, activation="relu")(inputs)
x = Conv2D(64, 3, activation="relu")(x)
block_1_output = MaxPooling2D(3)(x)

x = Conv2D(64, 3, activation="relu", padding="same")(block_1_output)
x = Conv2D(64, 3, activation="relu", padding="same")(x)
block_2_output = add([x, block_1_output])

x = Conv2D(64, 3, activation="relu", padding="same")(block_2_output)
x = Conv2D(64, 3, activation="relu", padding="same")(x)
block_3_output = add([x, block_2_output])

x = Conv2D(64, 3, activation="relu")(block_3_output)
x = MaxPooling2D()(x)
outputs = Dense(10)(x)

model = Model(inputs, outputs, name="toy_resnet")
```



Functional API

- You can treat any model as if it were a layer by invoking it on an Input or on the output of another layer.
- By calling a model you aren't just reusing the architecture of the model, you're also reusing its weights.
- Model can be nested
- Model can contain sub-models (since a model is just like a layer).



Functional API – Model Nesting

```
def get_model():
        inputs = keras.Input(shape=(128,))
        outputs = layers.Dense(1)(inputs)
        return keras.Model(inputs, outputs)

model1 = get_model()
model2 = get_model()
model3 = get_model()

inputs = keras.Input(shape=(128,))
y1 = model1(inputs)
y2 = model2(inputs)
y3 = model3(inputs)
outputs = layers.average([y1, y2, y3])
ensemble_model = keras.Model(inputs=inputs, outputs=outputs)
```

Functional API

```
:
prediction_a = Dense(5, name='prediction_a')(encoded_input_a)
prediction_b = Dense(1, name='prediction_a')(encoded_input_b)
:
:
model.compile(loss = {'prediction_a`: `crossentropy`, 'prediction_b`: `mse`} )
```

- If you have more than one output layer you can specify separate *loss* function for each output, by using the name you assigned to them while creating.

Functional API

```
:
:
model.fit({'input_1': x_train, 'input_2': x_string},
        {'prediction_a': y_train, 'prediction_b`: estimate},
        validation_split=0.2, epochs=30)
:
```

- Similarly, you can specify names to input layers, and feed different inputs and outputs in the *fit* function

Functional API- Limitations

- Does not support dynamic architectures
  - Recursive networks
  - Tree RNNs etc

Imperative API

Imperative API

- Also called **model subclassing**.
- Give higher flexibility and control.
- Lower-level API - More opportunity for **bugs!!**
- Difficult to debug

Use with Caution!

Imperative API – When to use

- Recursive structures
- Tree RNNs
- Customizable layers
- Use when require full control over model, layer, and training procedure implementation.

Imperative API – When to use

1. Create a child of Model Class
   - MyModel is a subclass- child of *Model* class.
   - It inherits from the Model class directly.
2. Create layers in the initializer
3. Define the forward pass in *call* method

Imperative API- Example

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Dropout
class MyModel(Model):
    def __init__(self, num_classes, **kwargs):   ## Initializer
        super(MyModel, self).__init__(**kwargs)
        self.input_ = Dense(10, activation='relu')
        self.drop_ = Dropout(0.2)
        self.output_ = Dense(num_classes, activation='softmax')

    def call(self, inputs, training=False):
        x = self.input_(inputs)
        if training:
            x = self.drop_(x)   ## x = self.drop_(x, training=training)
        return self.output_(x)
```

Model Definition

```
model = MyModel(5)
```

Model Instantiation

```
model.build((128,4))
```

Model building

Imperative API - Limitations

- Instantiating a model does not build the model
  - The model is build when either you specifically call the build function
  - Or you send it input via fit function
- More difficult to reuse
  - Increases **tech debt**
- Difficult to inspect, copy, or clone.

Debugging happens during execution, as opposed to when you're defining your model.

Complex Model Building and Training

TensorFlow Datasets

- Public research dataset as tf.data.Datasets and NumPy arrays.
- A large repository of datasets – for wide range of inputs- audio, Image, Graph, Object Detection, question-answers, text, and video

```
import tensorflow_datasets as tfds
mnist_data = tfds.load("mnist")
mnist_train, mnist_test = mnist_data["train"], mnist_data["test"]
```

Image

- Image Classification
- Image Segmentation
- Object Detection
- Pose Estimation
- Depth Estimation

Audio

- Audio Classification
- Speech Detection

Text

- Text Classification
- Text Summarization
- Question Answering

TensorFlow Keras Application

- Contains many pretrained models for image classification.
- Trained on ImageNet Data
- Models can be used for prediction, feature extraction, and fine-tuning.

TensorFlow Hub

- Library with many pretrained models.
- Contains hundreds of trained, ready-to-deploy deep learning models.
- Models for image classification, image segmentation, object detection, text embedding, text classification, video classification and generation, and much more.
- Models in TF Hub are available in SavedModel, TFLite, and TF.js formats.
- Use these pretrained models directly for inference or fine-tune them.

TF Callback

A callback is a powerful tool to customize the behavior of a model during

- Training
- Evaluation
- Inference

```
model.fit(x, y, callbacks=list_of_callbacks)
```

TF Callback - EarlyStopping

- This allows to monitor the metrics, and stop model training when it stops improving/when the minimum of loss has been reached

```
tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                 min_delta=0,
                                 patience=0,
                                 verbose=0,
                                 mode='auto',
                                 baseline=None,
                                 restore_best_weights=False)
```

TF Callback – ModelCheckpoint

- Allows to save the model regularly during training.
- Useful when training deep learning models which take a long time to train.

```
tf.keras.callbacks.ModelCheckpoint(filepath,
                                   monitor='val_loss',
                                   verbose=0,
                                   save_best_only=False,
                                   save_weights_only=False,
                                   mode='auto',
                                   save_freq='epoch')
```

TF Callback – TensorBoard

- Useful for visualization of the training summary for the model.
- Generates the logs for TensorBoard, which can later be launched to visualize the progress of the training.

```
tf.keras.callbacks.TensorBoard(log_dir='logs',
                               histogram_freq=0,
                               write_graph=True,
                               write_images=False,
                               update_freq='epoch',
                               profile_batch=2,
                               embeddings_freq=0,
                               embeddings_metadata=None,
                               **kwargs)
```

Using Custom Loss Function

- Define your custom loss function:

```
def custom_loss(y_true, y_pred):
        # calculate loss, using y_pred
    return loss
```

```
model.compile(loss=custom_loss, optimizer='adam')
```

Custom Training – Gradient Tape

- Used to write custom training loops.
- Record operations for automatic differentiation.
- Trainable variables are automatically watched.

```
tf.GradientTape( persistent=False, watch_accessed_variables=True)
```

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
      g.watch(x)
     y = x * x
dy_dx = g.gradient(y, x)
print(dy_dx)
>>> tf.Tensor(6.0, shape=(), dtype=float32)
```

Data Augmentation

- Used to artificially enlarge the dataset.
- Improve performance and outcomes of machine learning models.
- Some augmenting methods:
  - padding
  - random rotating
  - re-scaling,
  - vertical and horizontal flipping
  - translation
  - cropping
  - zooming
  - darkening & brightening
  - grayscaling
  - changing contrast
  - adding noise
  - random erasing

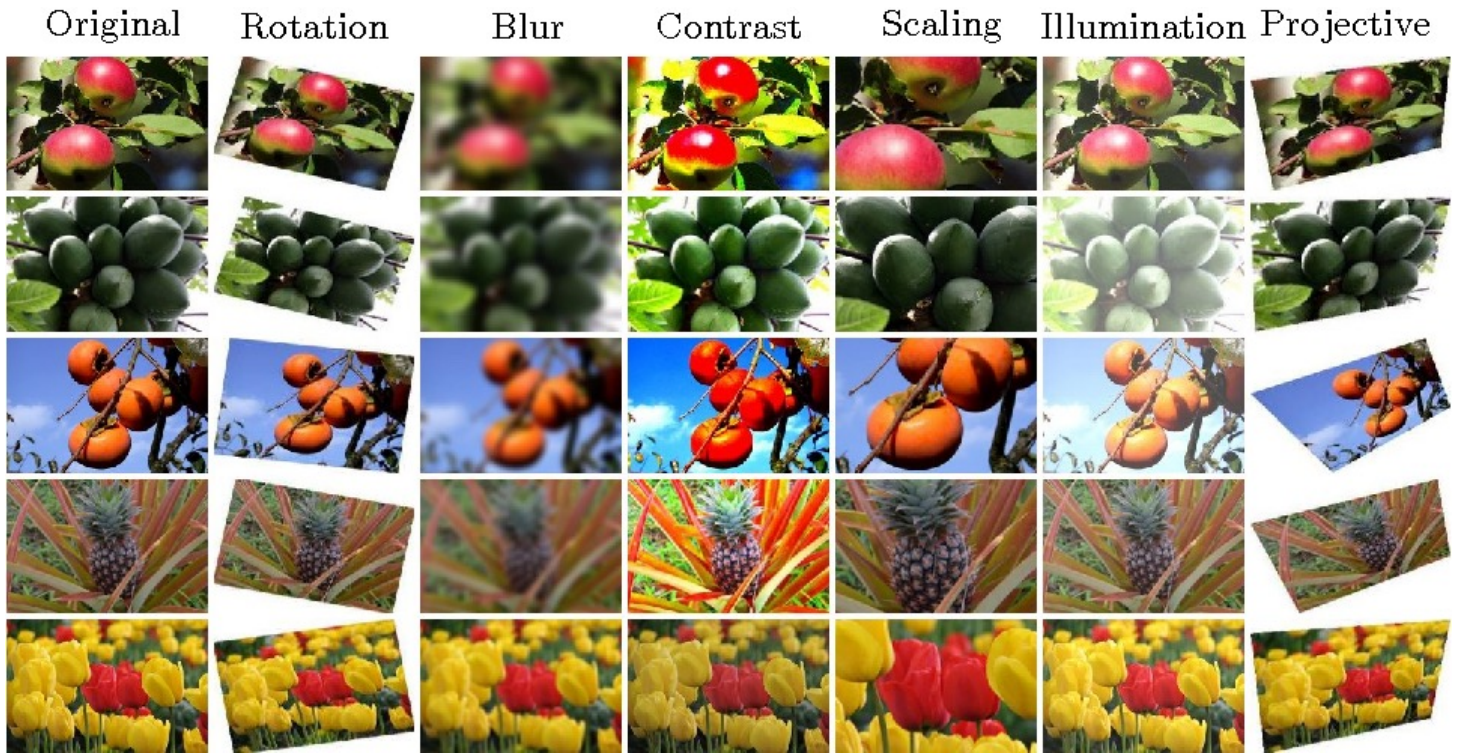


Image Source: https://medium.com/analytics-vidhya/data-augmentation-is-it-really-necessary-b3cb12ab3c3f

Traditional Deep Neural Networks

DNNs are able to learn effectively from large amounts of data.
Data is presented in input-output pairs (supervised learning).
Requires enormous computing power

Traditional Deep Neural Networks

DNNs are able to learn effectively from large amounts of data.	Large Data is not always available
Data is presented in input-output pairs (supervised learning).	Labelled Data is not always available
Requires enormous computing power	Is expensive and Time consuming

Solution

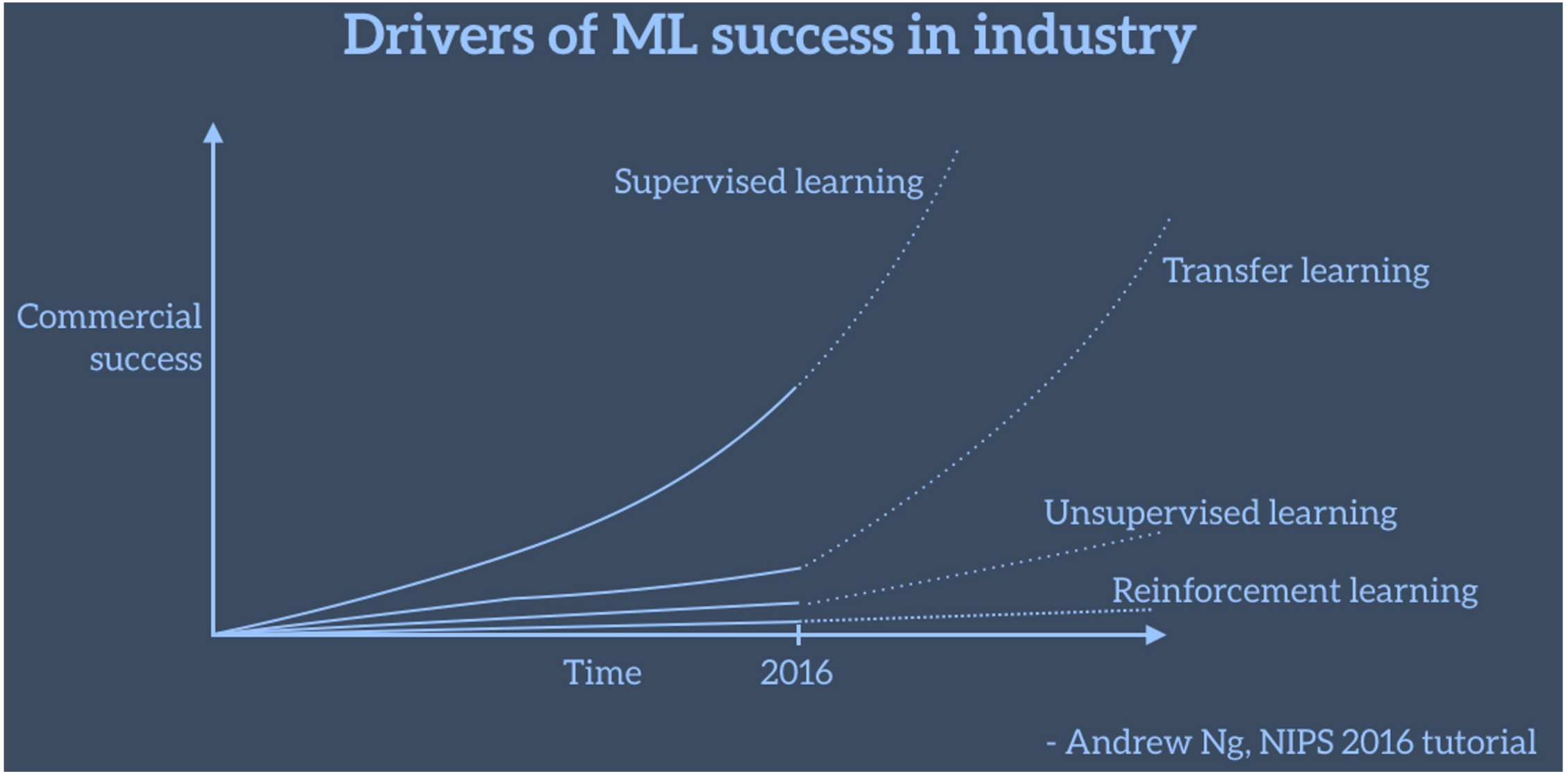
Transfer Learning

Every time we see a similar problem, instead of learning from scratch we use previously trained agent.

Transfer Learning

Transfer learning will become a key driver of Machine Learning success in industry.

-Andrew Ng (NIPS 2016)



Taken From: http://ruder.io/transfer-learning/

What is Transfer Learning?

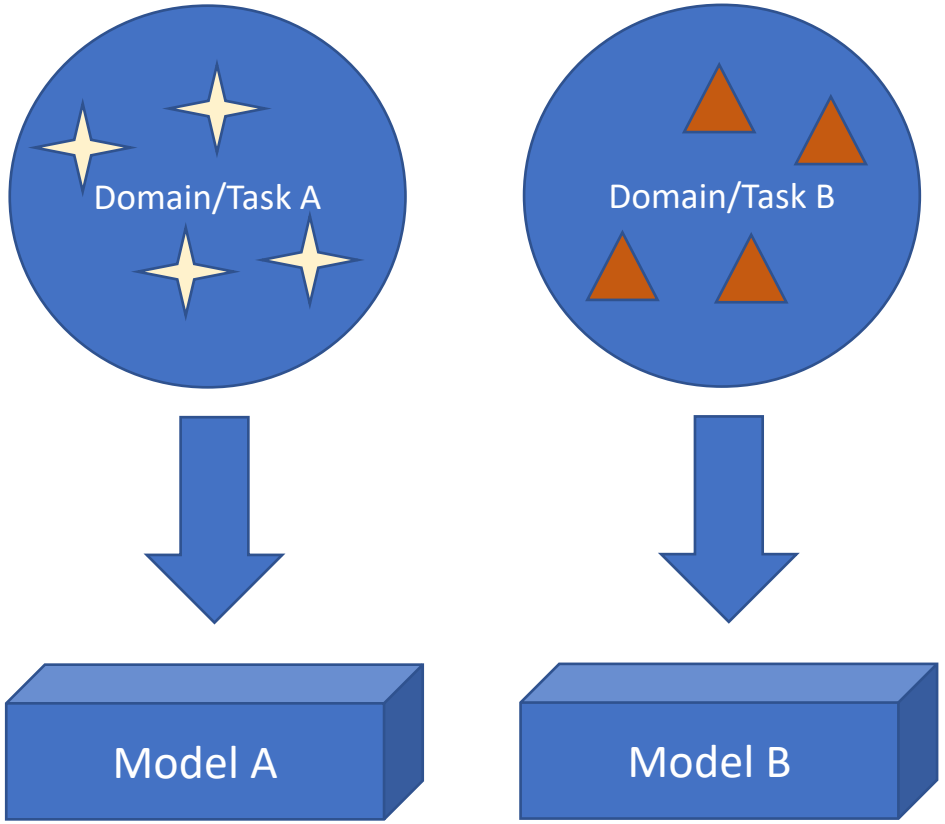
- People can intelligently apply knowledge learned previously to solve new problems.
- Not a new concept:
  - NIPS-95 Learning to Learn: *Need for lifelong machine learning methods that retain and reuse previously learned knowledge*.
  - DARPA 2005: *The ability of a system to recognize and apply knowledge and skills learned in previous task to novel tasks*.

Transfer Learning- ML Commercial Success-Key?

- Traditional Models have reported Super human performance in certain tasks.
- Yet, when they are used in production, the performance deteriorates.
- Real world is very different from the structured data used for training and testing.
- Individual users can have slightly different preferences.
- Transfer Learning can help deal with these and allow us to use ML beyond tasks and domains where
  - labelled data is plentiful,
  - data is outdated
- Boost productivity by reducing time to implement new projects

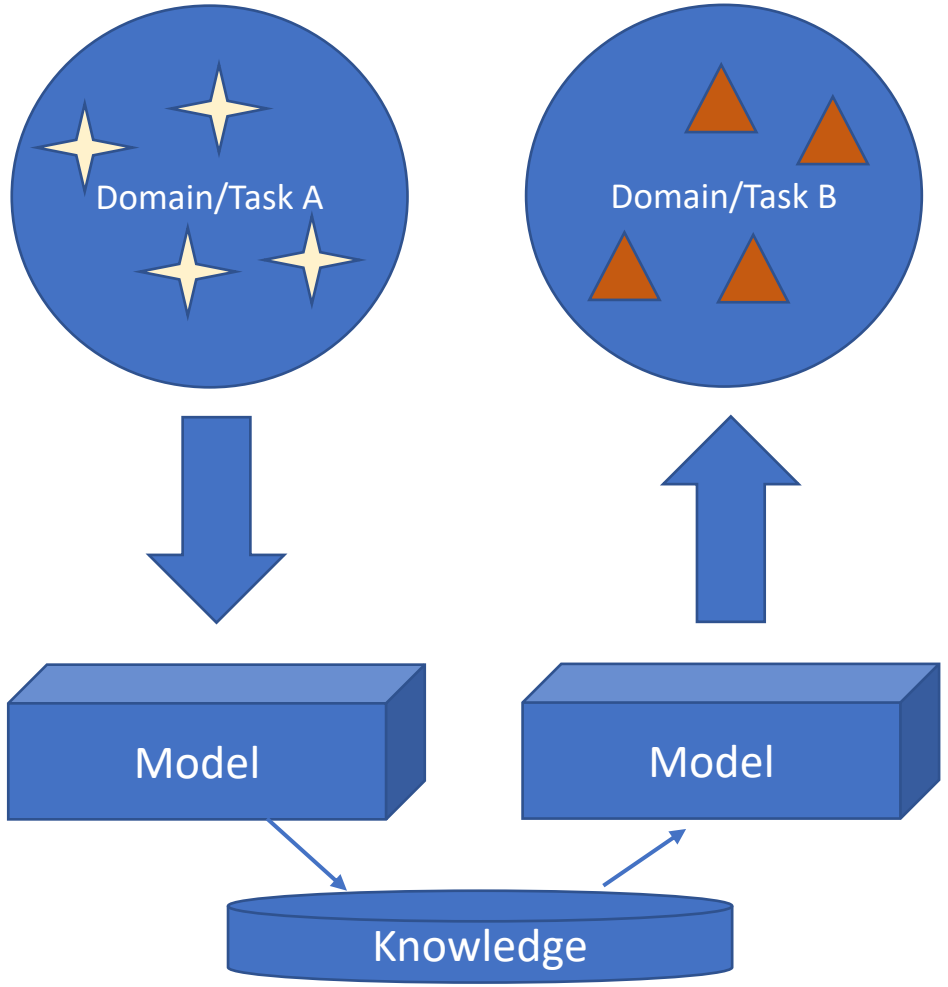
Traditional vs Transfer Learning

$D_s = D_T \quad and \quad T_s = T_T$



Training and Evaluation on same task/domain

$D_s \neq D_T \quad or \quad T_s \neq T_T$



Training and Evaluation on different task/domain

Transfer Learning

- Transfer knowledge to new conditions.
- Reuse of some or all of the training (data) of a prior model:
    - feature representations
    - neural-node layering
    - Weights
    - training method
    - loss function
    - learning rate etc.
- Tap into the knowledge gained on prior projects: Supervised, Unsupervised, Reinforcement Learning
- Extracts knowledge from one or more source tasks and apply the knowledge to a target task.

Pre-trained Models

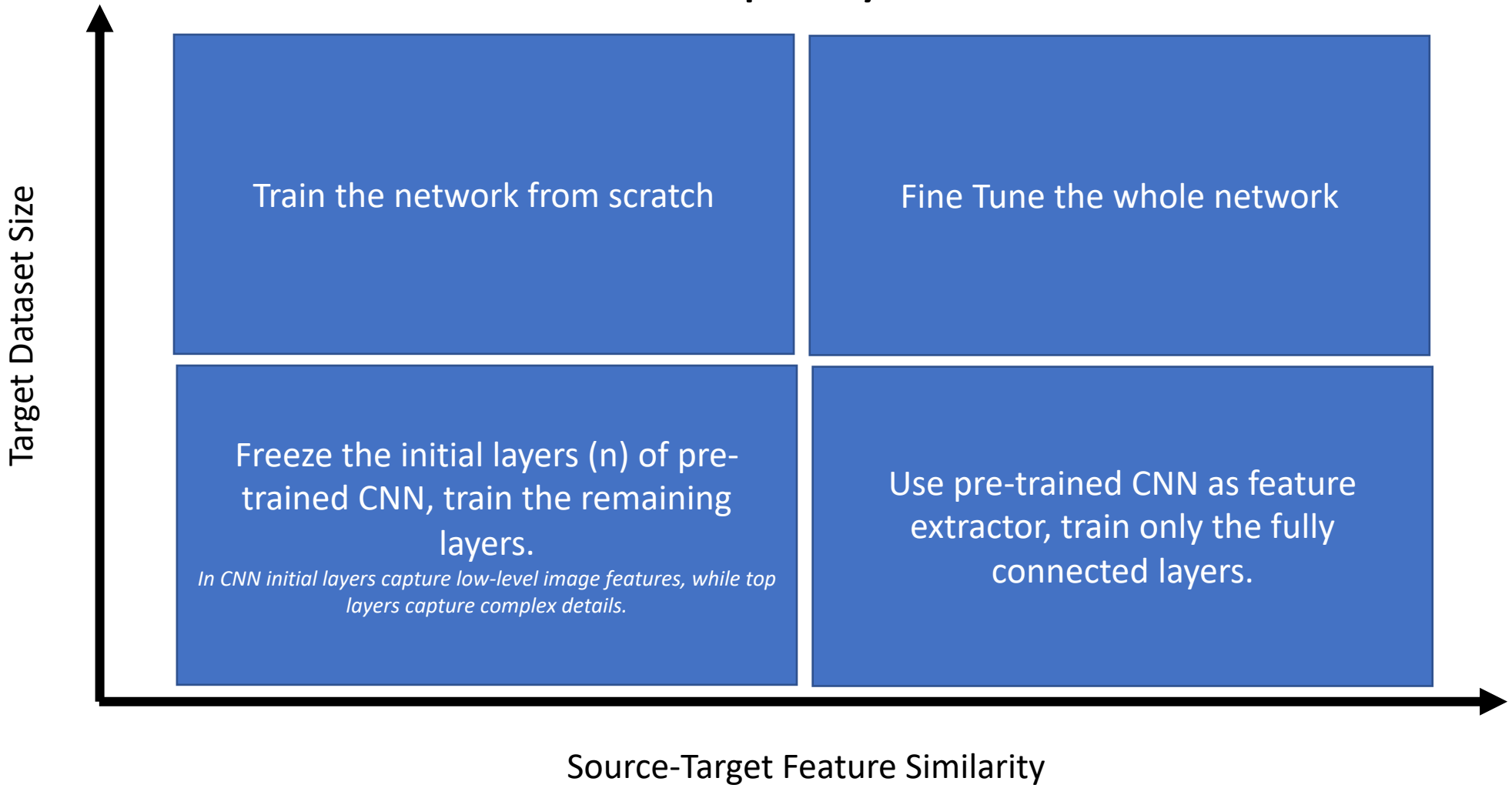
Pre-trained CNN as feature extractor:

Take a pre-trained CNN model, remove the last fully connected layers, use the remaining CNN as feature extractor for the new dataset. The output of this CNN feature extractor is fed to a classifier. The classifier is trained for the new dataset.

Fine tune pre-trained CNN:

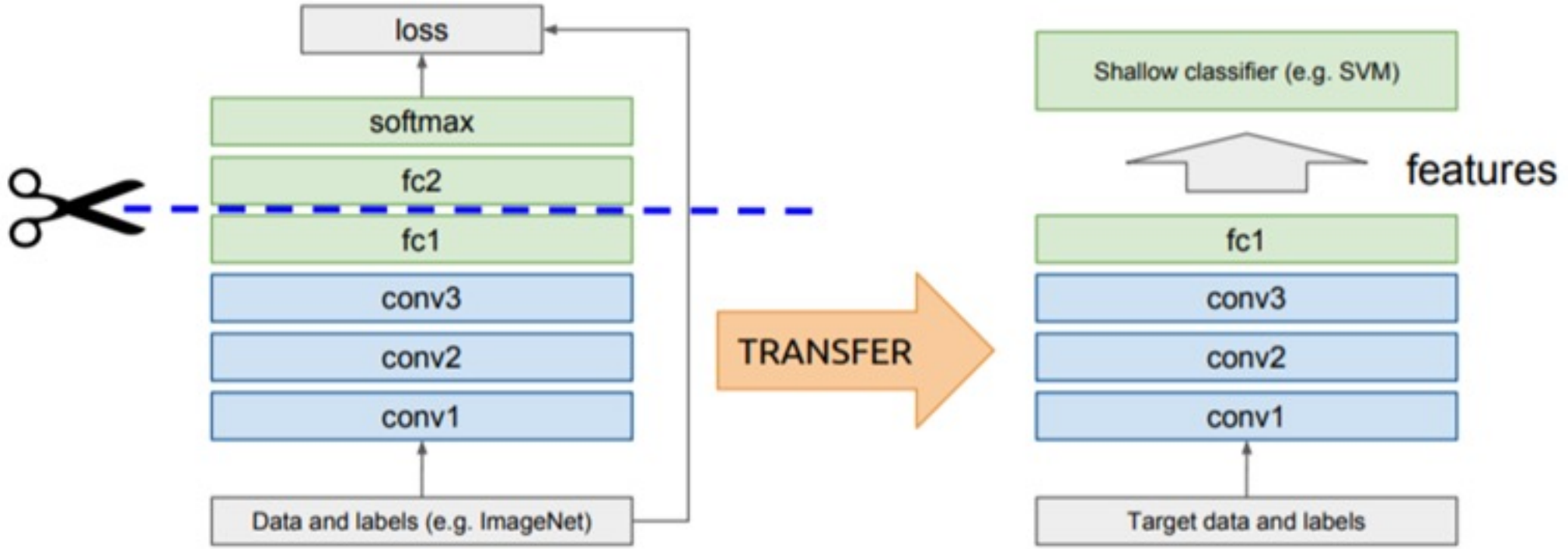
The weights of both the pre-trained CNN layers (all or some) and fully connected classifier layer/s are fine tuned using backpropagation.

Which method to employ?



Pre-trained CNN as feature extractor

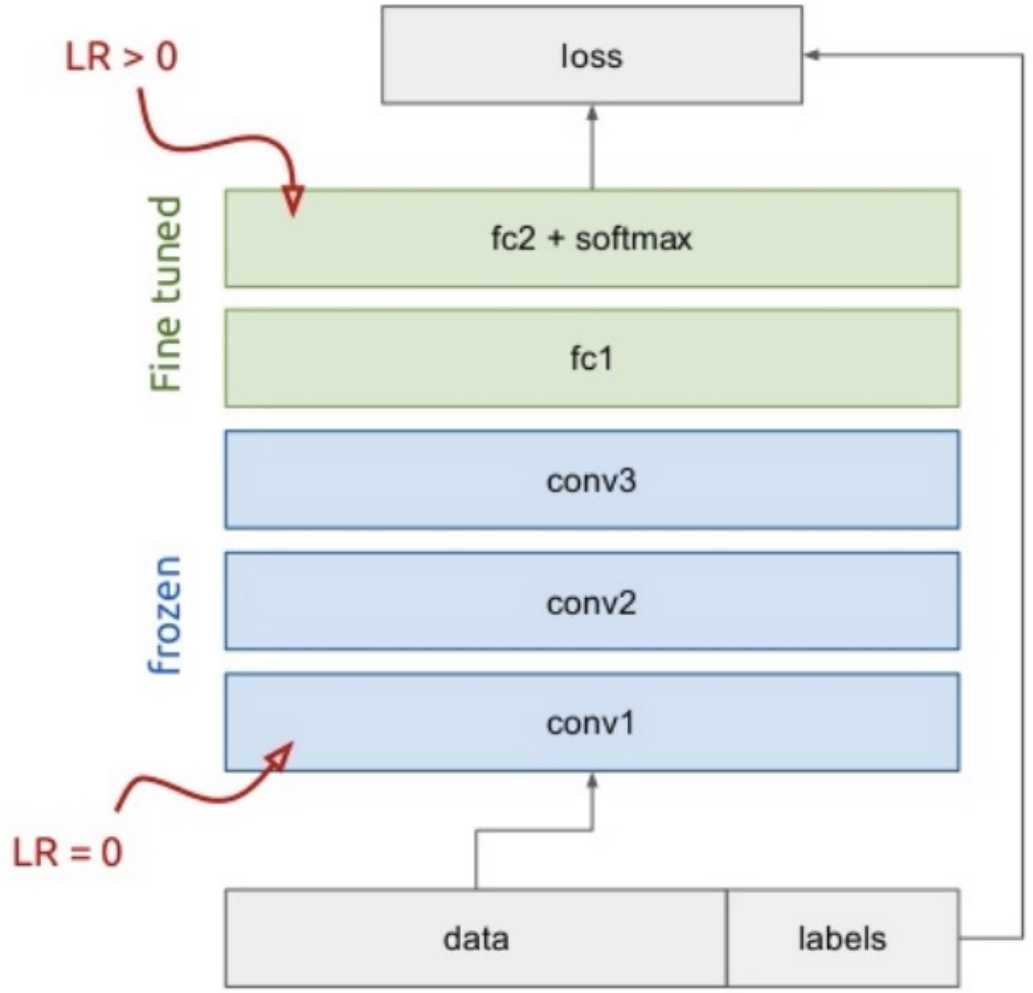
Assumes that $D_S = D_T$



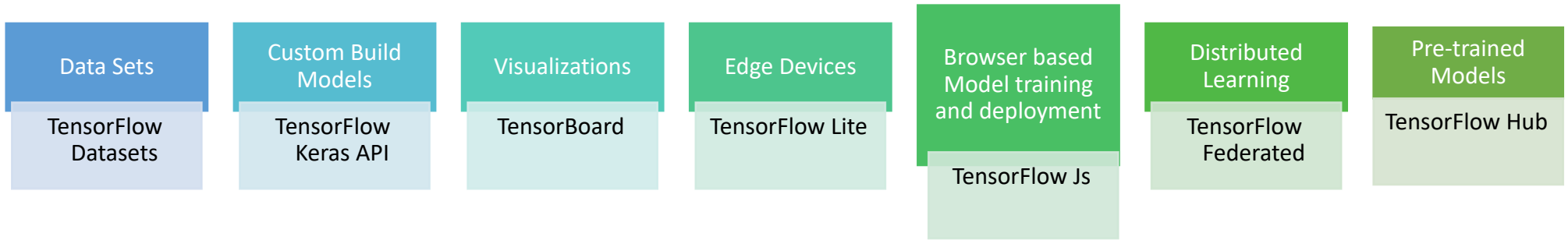
Pre-Trained Model: TensorFlow Hub

Models for Image classification with weights trained on ImageNet

- Xception
- VGG16
- VGG19
- ResNet50
- InceptionV3
- MobileNet

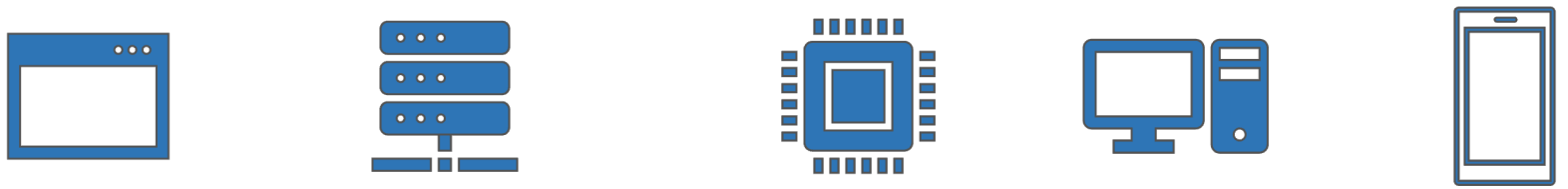


TensorFlow Ecosystem


Data Sets
TensorFlow Datasets
Custom Build Models
TensorFlow Keras API
Visualizations
TensorBoard
Edge Devices
TensorFlow Lite
Browser based Model training and deployment
TensorFlow Js
Distributed Learning
TensorFlow Federated
Pre-trained Models
TensorFlow Hub

TensorFlow.js

- A collection of APIs to train and run inference directly in browsers or in Node.js.
- Execute Machine Learning models: Client side in-browser
  - Low Latency
  - Data Privacy
  - Easy deployment
- TensorFlow.js scripts can run anywhere the Javascript can run



TensorFlow Lite

- Enables machine learning inference directly on mobile and embedded devices
  - Small binary and model size to save on memory,
  - Low energy consumption to save on the battery,
  - Low latency for efficiency.
- Quantization
- FlatBuffers
- Mobile Interpreter
- Mobile Converter

Federated Learning and Federated Core

- Distributed machine learning- both data and devices are distributed.
- Enables training for decentralized data
  - Distributed data
  - Private data

Bringing code to the data

Thanks