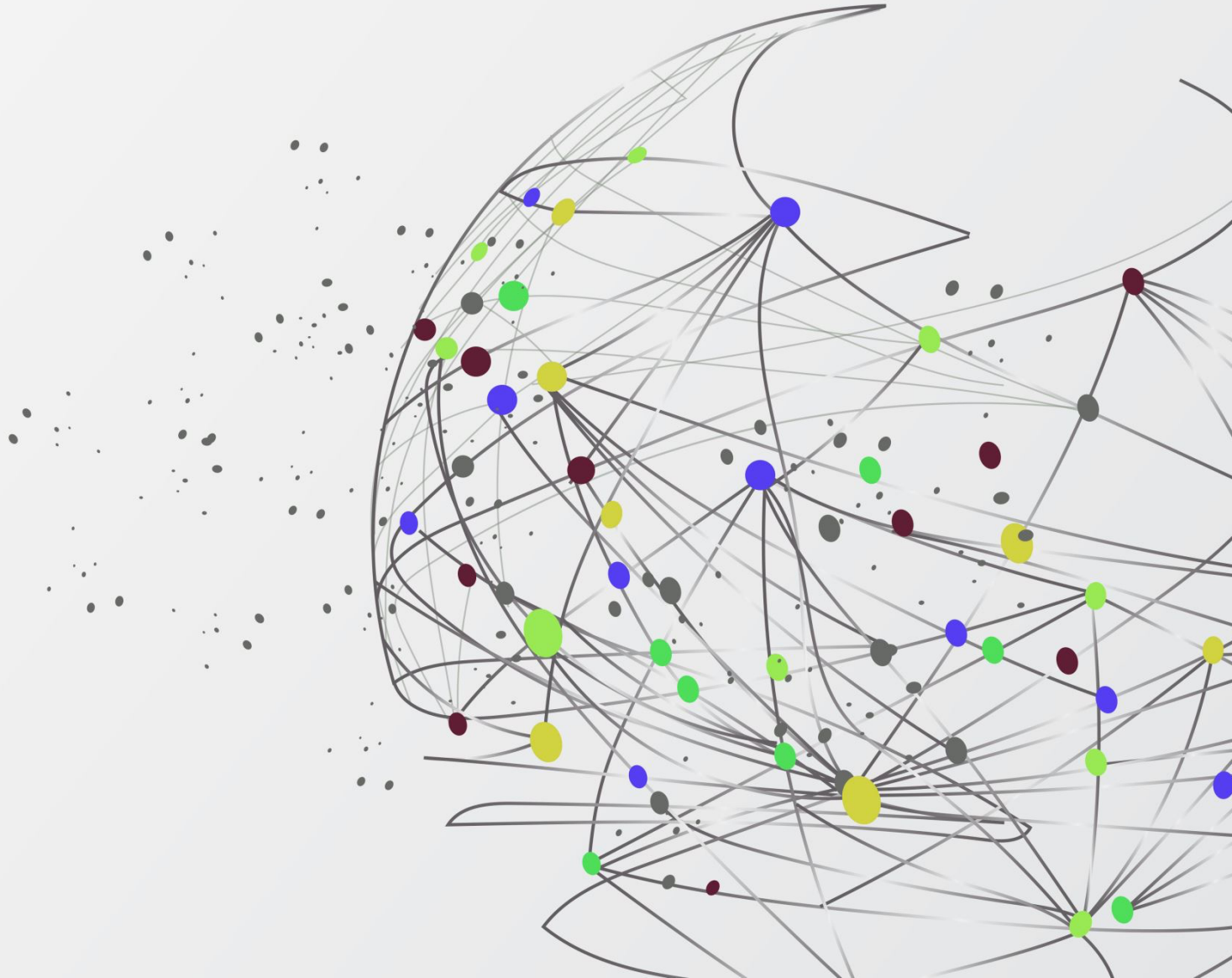
Mentor Session
Deep Learning Part 2

Machine Learning &
Data Science II

Yingjie(Chelsea) Wang



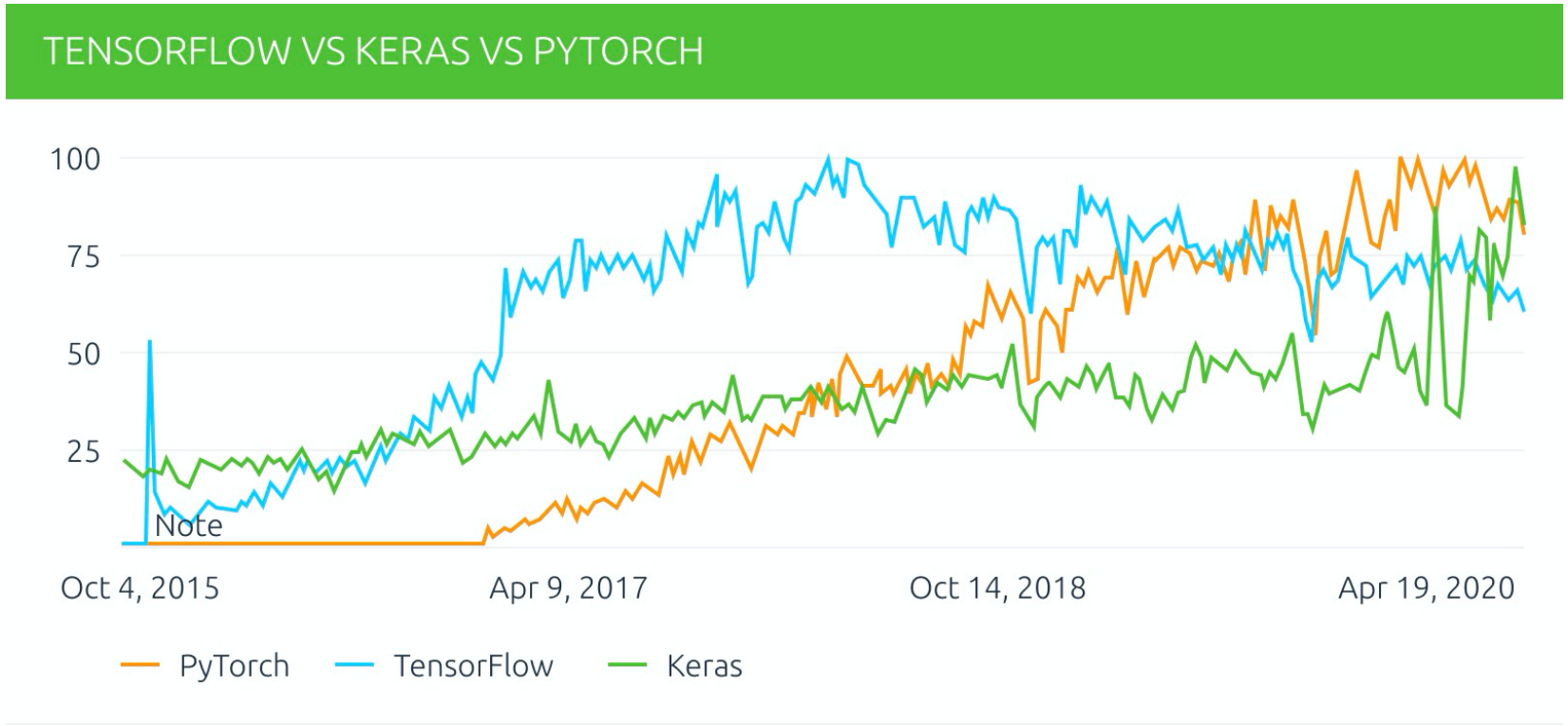
Today's Plan

- Deep Learning Framework: Keras vs PyTorch
- Review on Concepts for Deep Learning
  - Multi-Layer Perceptron (MLP)
    - Feedforward, Backpropagation
    - Use Case: Function Approximation, XOR, MNIST Image Classification
  - Convolutional Neural Networks (CNN)
    - Convolution layer
    - Pooling layer
    - Use Case: MNIST Image Classification, Flowers Recognition, Malaria Cell Image Classification
  - Recurrent Neural Networks (RNN)
    - A little review
    - Use Case: Chirp Signal Time Series, Text Classification

Keras vs PyTorch

- Keras – For beginners, school projects, small dataset research
- PyTorch – For experts, publication, large and complex dataset research

KERAS	PYTORCH
Keras was released in **March 2015**.	While PyTorch was released in **October 2016**.
Keras has a **high-level** API.	While PyTorch has a **low-level** API.
Keras is comparatively **slower in speed**.	While PyTorch has a **higher speed** than Keras, suitable for high performance.
Keras has a **simple architecture**, making it more readable and easier to use.	While PyTorch has very low readablility due to a **complex architecture**.
Keras has a smaller community support.	While PyTorch has a stronger community support.
Keras is mostly used for **small datasets** due to its slow speed.	While PyTorch is preferred for **large datasets** and high performance.
Debugging in Keras is **difficult** due to presence of computational junk.	While **debugging** in PyTorch is **easier** and faster.
Backend for Keras include: TensorFlow, Theano and Microsoft CNTK backend.	While PyTorch has no backend implementation.



Keras vs PyTorch

- Keras – Great access to tutorials and reusable code
- PyTorch – Excellent community support and active development

Keras vs PyTorch

- Keras – more concise, simpler API
- PyTorch – more flexible, encouraging deeper understanding of deep learning concepts

Keras

```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPool2D())
model.add(Conv2D(16, (3, 3), activation='relu'))
model.add(MaxPool2D())
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

PyTorch

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3)
        self.conv2 = nn.Conv2d(32, 16, 3)
        self.fc1 = nn.Linear(16 * 6 * 6, 10)
        self.pool = nn.MaxPool2d(2, 2)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 6 * 6)
        x = F.log_softmax(self.fc1(x), dim=-1)
        return x
model = Net()
```

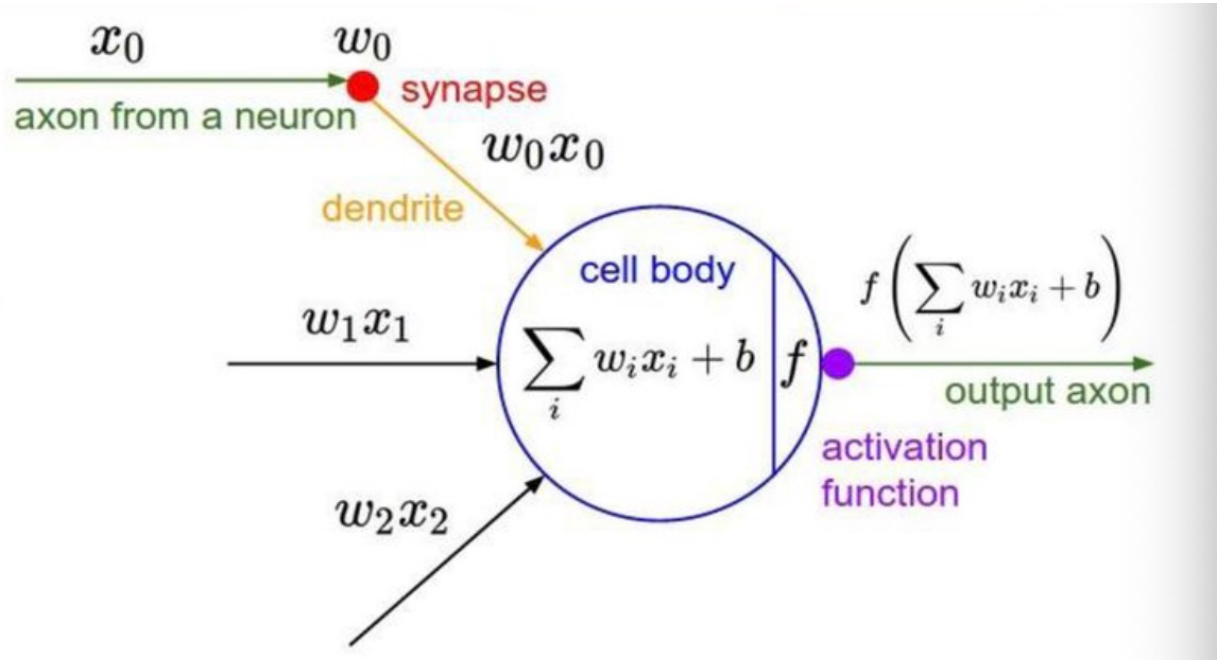
Types of Neural Networks in Deep Learning

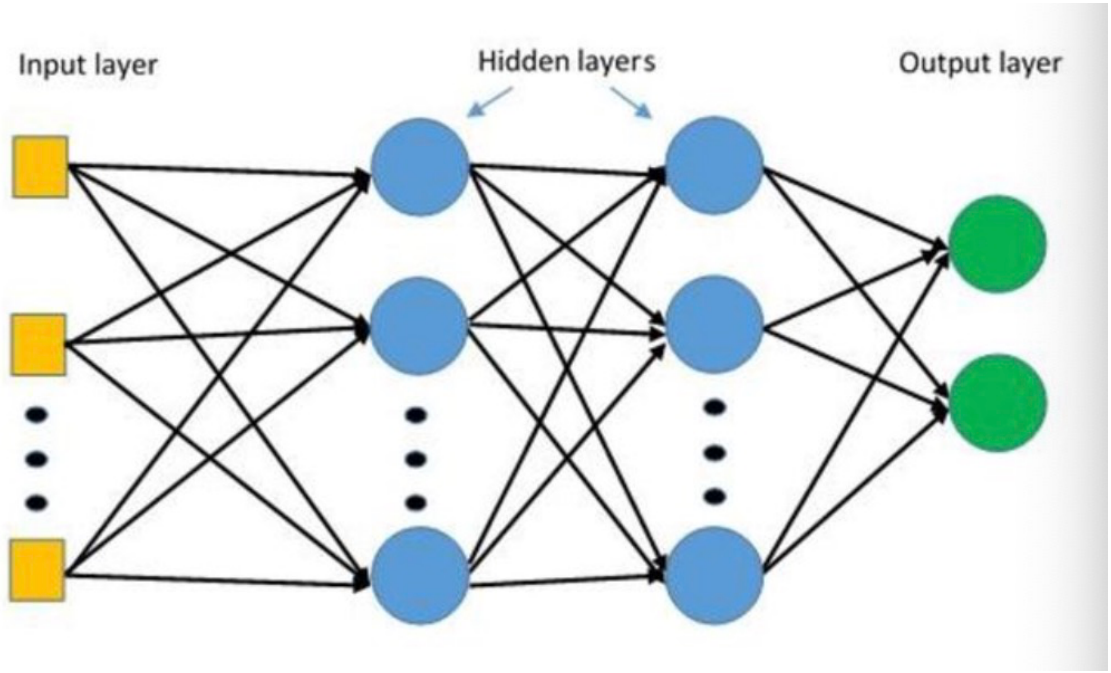
- Three important types of neural networks that form the basis for most pre-trained models in deep learning:
  - Artificial Neural Networks (ANN)
  - Convolution Neural Networks (CNN)
  - Recurrent Neural Networks (RNN)

	MLP	RNN	CNN
Data	Tabular data	Sequence data (Time Series,Text, Audio)	Image data
Recurrent connections	No	Yes	No
Parameter sharing	No	Yes	Yes
Spatial relationship	No	No	Yes
Vanishing & Exploding Gradient	Yes	Yes	Yes

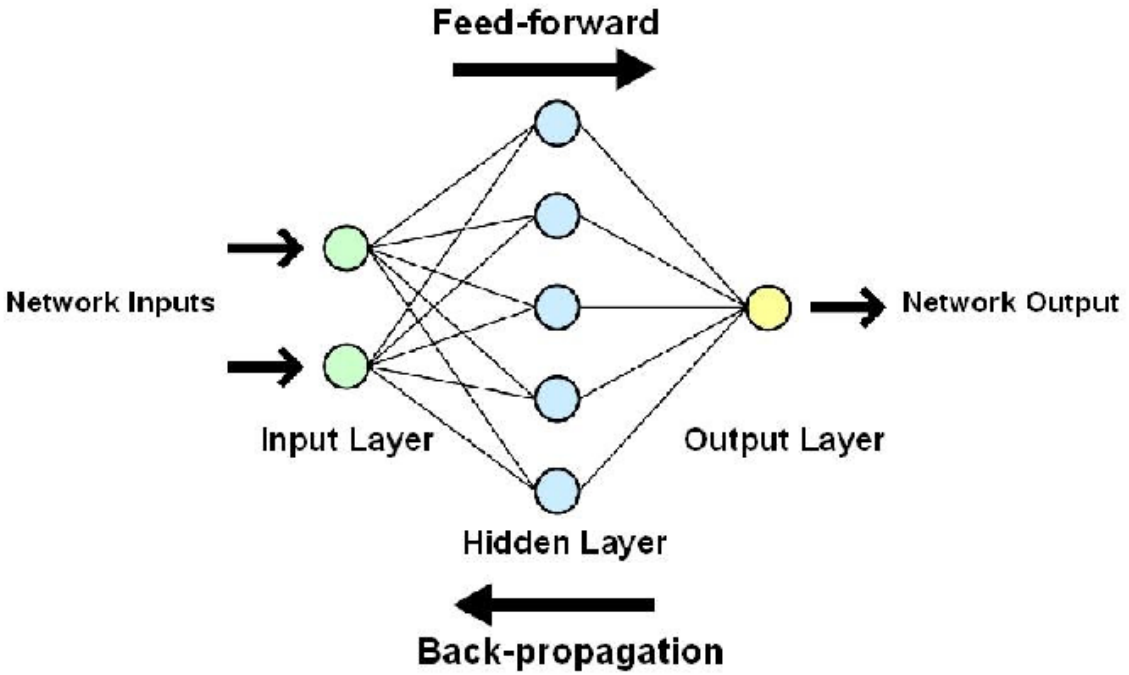
Perceptrons and Multi-Layer Perceptrons

- Perceptron
  - A simple binary classification algorithm
- Multi-layer Perceptron (MLP)
  - A perceptron that teams up with additional perceptrons, stacked in several layers, to solve complex problems.
  - Capability to learn non-linear models.





Feedforward and Backpropagation



Forward pass

$$x \xrightarrow{r = w_x x} r \xrightarrow{y = w_r r} y \xrightarrow{} L = \frac{1}{2}(y - y_{\text{target}})^2$$

(y_{target} feeds into the loss)

Backward pass

$$\frac{\partial L}{\partial r} \longleftarrow \frac{\partial L}{\partial y}$$

$$\frac{\partial L}{\partial w_x} = \frac{\partial L}{\partial r}\frac{\partial r}{\partial w_x} \qquad \frac{\partial L}{\partial r} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial r} = \frac{\partial L}{\partial y} w_r$$

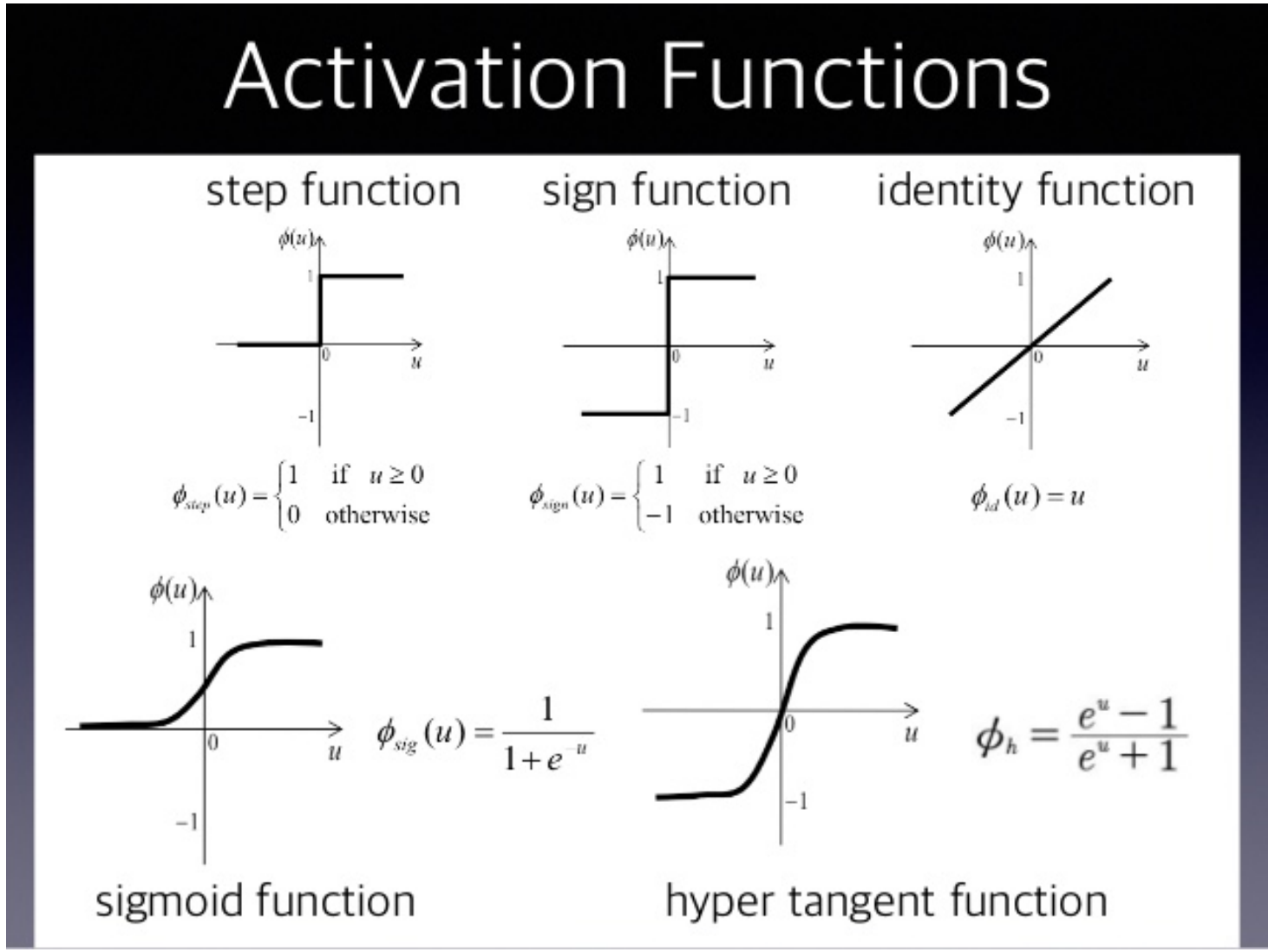
Non-local: $\frac{\partial L}{\partial y} w_r$ — Local: x

6 Stages of Neural Network Learning

- **Initialization** – initial weights are applied to all the neurons.
- **Forward propagation** – the inputs from a training set are passed through the neural network and an output is computed.
- **Error function** – because we are working with a training set, the correct output is known. An error function is defined, which captures the delta between the correct output and the actual output of the model, given the current model weights (in other words, “how far off” is the model from the correct result).
- **Backpropagation** – the objective of backpropagation is to change the weights for the neurons, in order to
- **Weight update –** weights are changed to the optimal values according to the results of the backpropagation algorithm.
- **Iterate until convergence** – because the weights are updated a small delta step at a time, several iterations are required in order for the network to learn. After each iteration, the gradient descent force updates the weights towards less and less global loss function. The amount of iterations needed to converge depends on the learning rate, the network meta-parameters, and the optimization method used.
- At the end of this process, the model is ready to make predictions for unknown input data. New data can be fed to the model, a forward pass is performed, and the model generates its prediction.

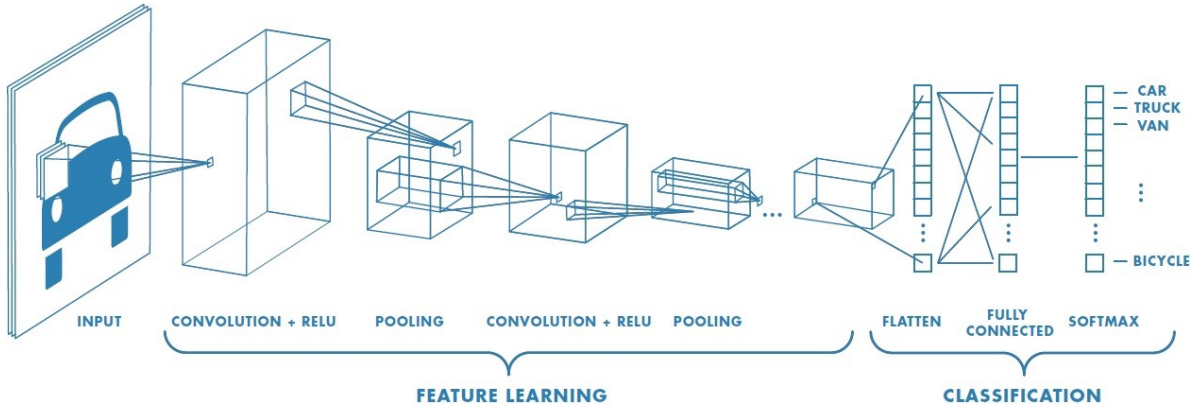
Multi-layer Perceptron (MLP) Application

- MLPs as universal approximators
  - A one-layer MLP can model an arbitrary function of a single input.
- MLPs as universal classifiers
  - A one-layer MLP can model any classification boundary
- One of the main reasons behind these applications is the activation function. Activation functions introduce nonlinear properties to the network. This helps the network learn any complex relationship between input and output.



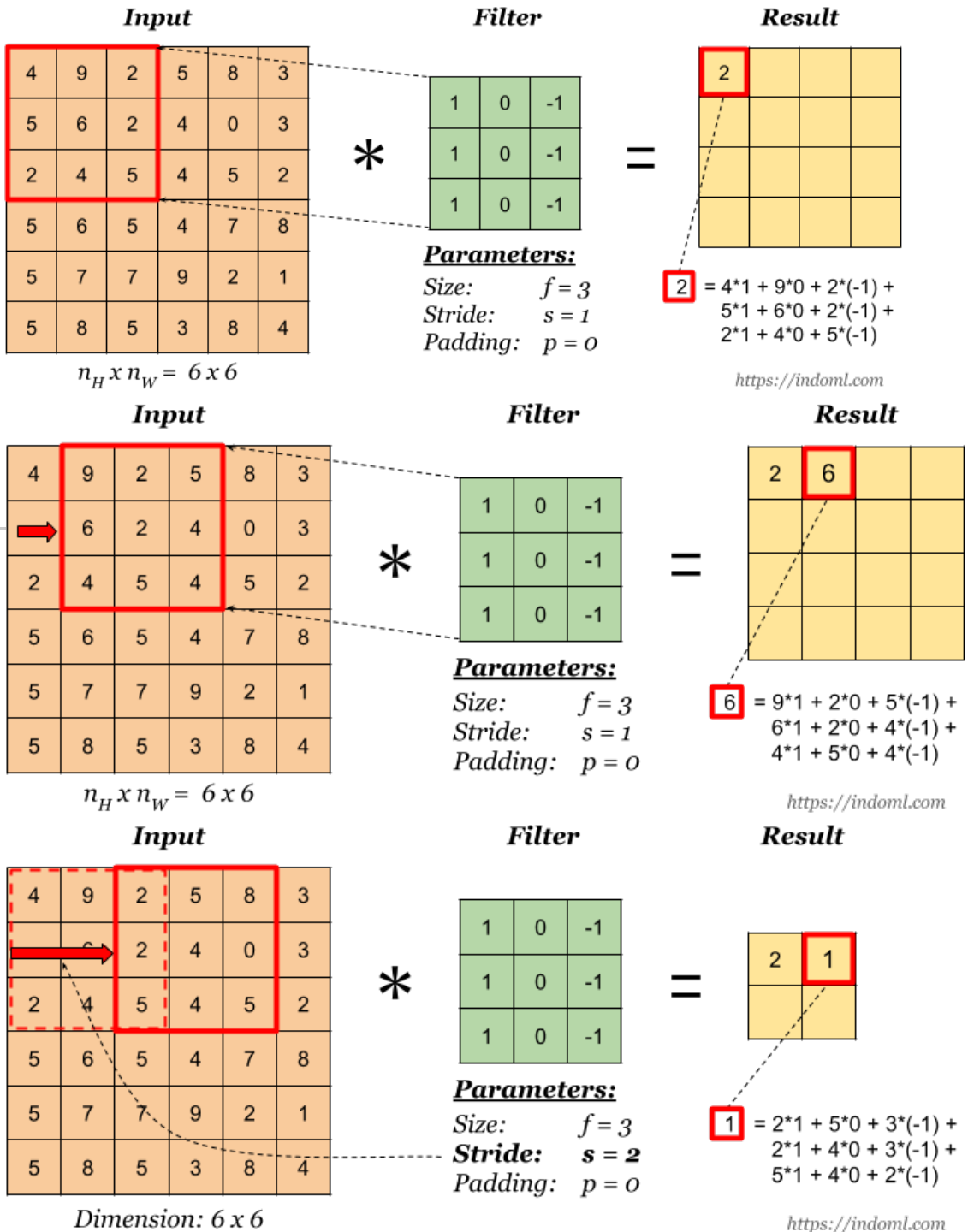
Convolution Neural Network (CNN)

- Convolutional neural networks, also known as CNNs, are a specific type of neural networks that is especially prevalent in image and video processing projects.
- Procedure:
  - Provide input image into convolution layer.
  - Choose parameters, apply filters with strides, padding if requires. Perform convolution on the image and apply ReLU activation to the matrix.
  - Perform pooling to reduce dimensionality size.
  - Add as many convolutional layers until satisfied.
  - Flatten the output and feed into a fully connected layer.
  - Output the class using an activation function and classify the images.



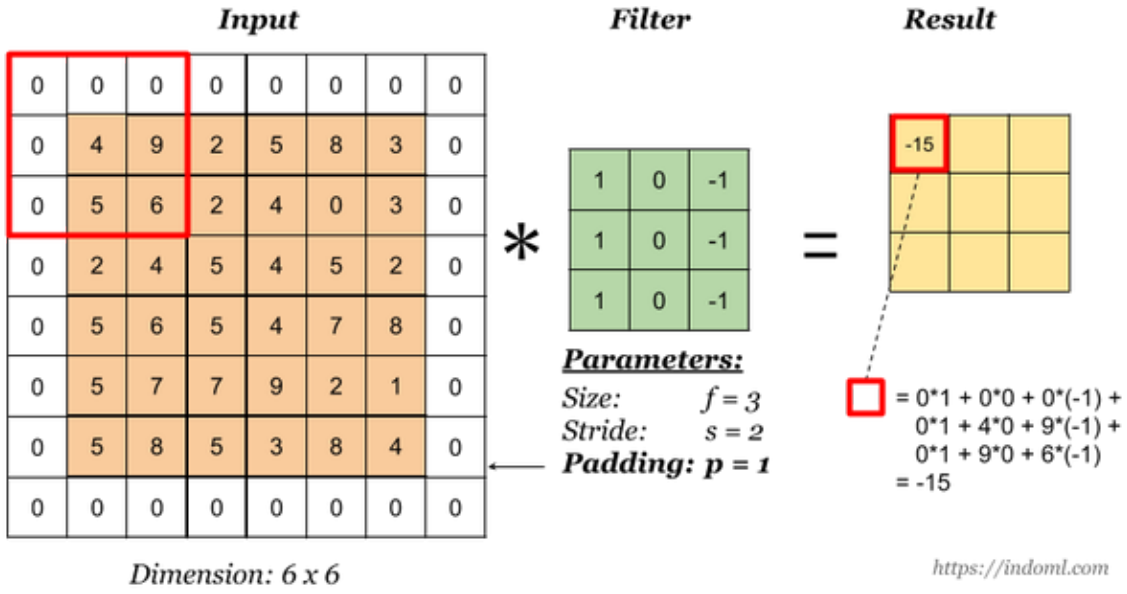
CNN: Types of Layer

- Convolution layer (CONV) uses filters that perform convolution operations as it is scanning the input with respect to its dimensions.
- Its hyperparameters include the filter size and stride. The resulting output is called feature map or activation map.
- Stride governs how many cells the filter is moved in the input to calculate the next cell in the result.



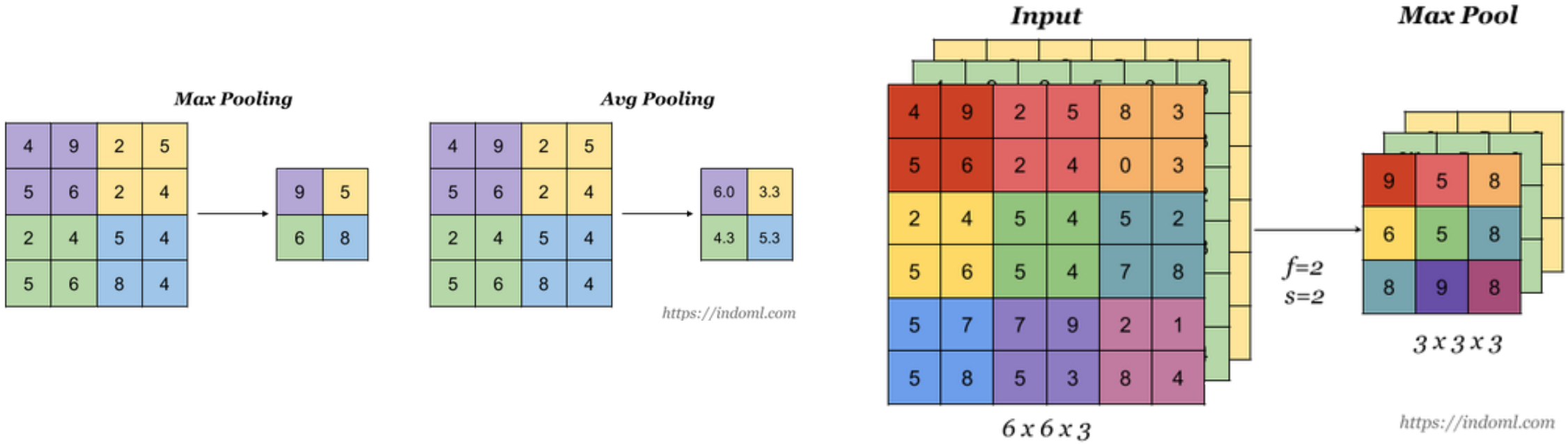
CNN: Types of Layer

- Padding has the following benefits:
  - It allows us to use a CONV layer without necessarily shrinking the height and width of the volumes. This is important for building deeper networks, since otherwise the height/width would shrink as we go to deeper layers.
  - It helps us keep more of the information at the border of an image. Without padding, very few values at the next layer would be affected by pixels as the edges of an image.
- Some padding terminologies:
  - “valid” padding: no padding
  - “same” padding: padding so that the output dimension is the same as the input
- Note that the dimension of the result has changed due to padding.



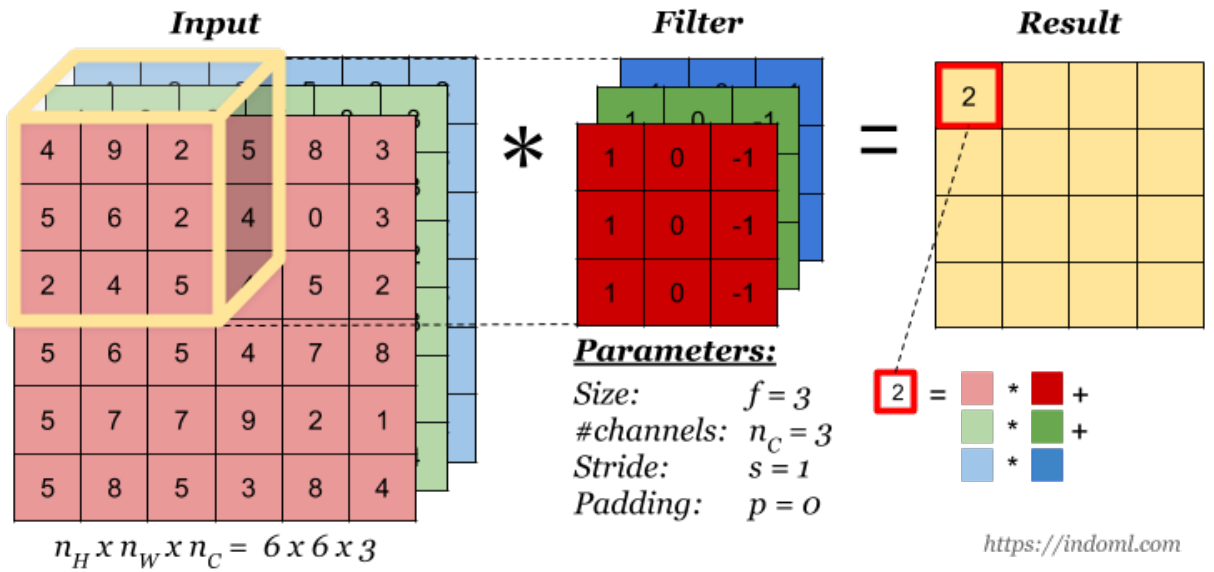
CNN: Types of Layer

- The pooling layer (POOL) is a downsampling operation, typically applied after a convolution layer, which does some spatial invariance. In particular, max and average pooling are special kinds of pooling where the maximum and average value is taken, respectively.
- It has hyper-parameters: size (f), stride (s), type (max or avg).
- But it doesn't have parameter so that there's nothing for gradient descent to learn



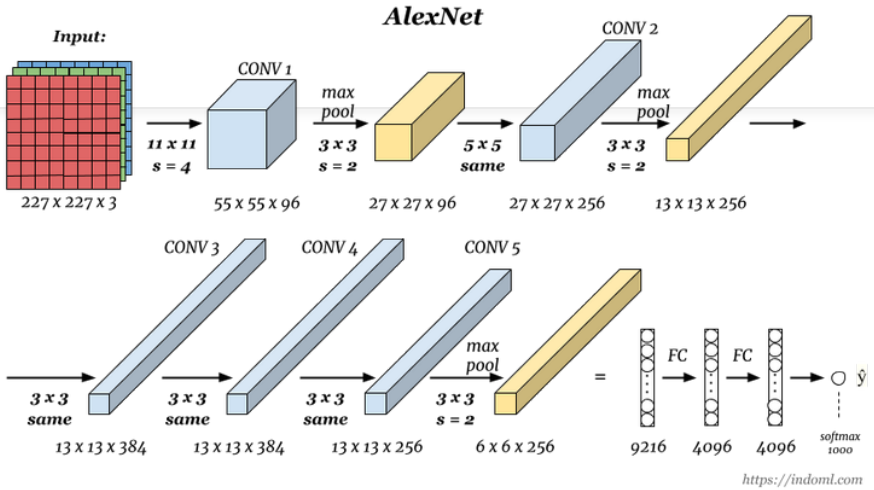
CNN: Convolution Operation on Volume

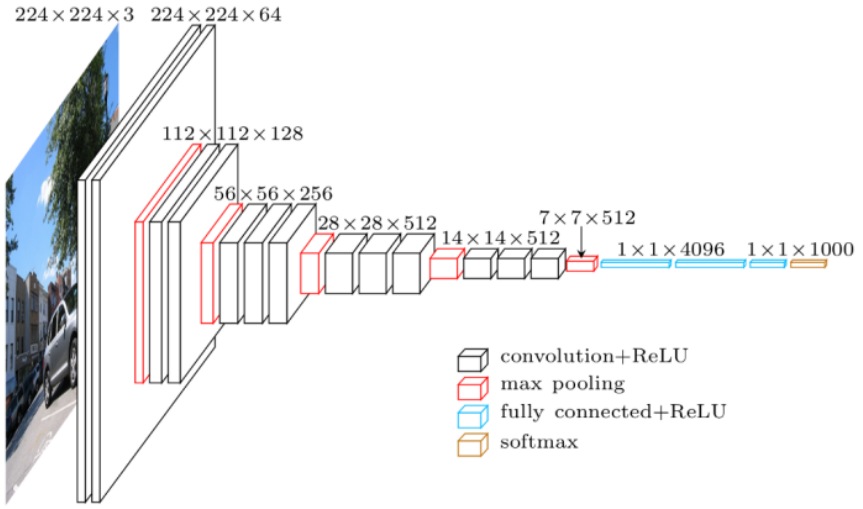
- When the input has more than one channels (e.g. an RGB image), the filter should have matching number of channels. To calculate one output cell, perform convolution on each matching channel, then add the result together.
- The total number of multiplications to calculate the result is
  - (4 x 4) x (3 x 3 x 3) = 432.



CNN: Classic Network

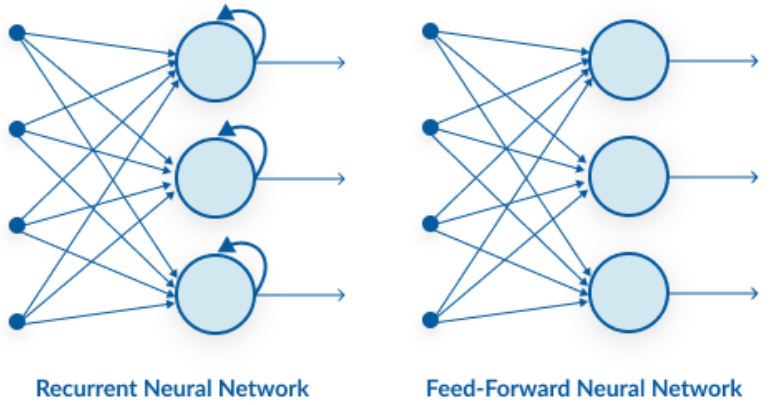
- AlexNet
  - AlexNet is another classic CNN architecture from ImageNet Classification with Deep Convolutional Neural Networks paper by Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever (2012).
  - Number of parameters: ~ 60 millions.
- VGG-16
  - VGG-16 from Very Deep Convolutional Networks for Large-Scale Image Recognition paper by Karen Simonyan and Andrew Zisserman (2014). The number 16 refers to the fact that the network has 16 trainable layers (i.e. layers that have weights).
  - Number of parameters: ~ 138 millions.
  - The strength is in the simplicity: the dimension is halved and the depth is increased on every step (or stack of layers)

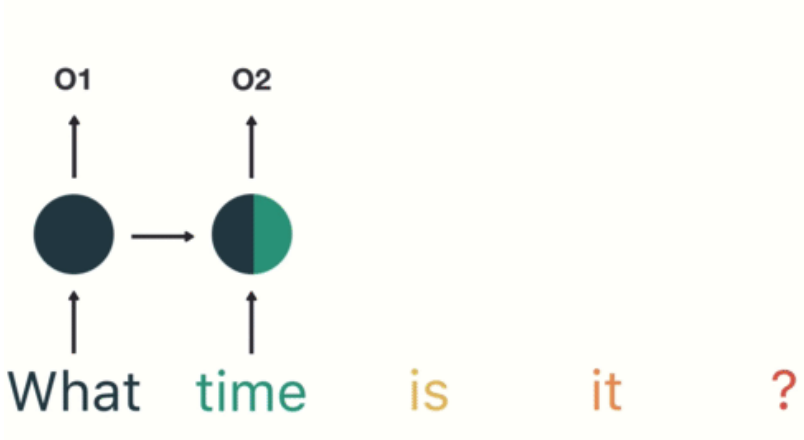




Recurrent Neural Network (RNN)

- Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. This looping constraint ensures that sequential information is captured in the input data.
- We can use recurrent neural networks to solve the problems related to:
  - Time Series data
  - Text data
  - Audio data
- RNN models are mostly used in the fields of natural language processing and speech recognition.





Regularization Strategies

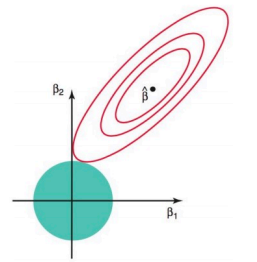
- L1 and L2 Loss
- Early Stopping
- Dropout and
- Dataset augmentation

L1 and L2 Loss

- Parameter Penalization
  - Just like with LASSO and Ridge Regression, the premise here is that we can constrain the magnitudes of the learned parameters
  - The regularization approaches are based on limiting the capacity of models by adding a parameter norm penalty $g(\theta)$ to the objective function L. We denote the regularized objective function by L.
    - $L = L\,(\theta; X, y) + \lambda g(\theta)$
  - where $\lambda \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, $g(\theta)$, relative to the standard objective function L. Setting λ to zero results in no regularization. Larger values of λ correspond to more regularization.
  - We typically choose to use a parameter norm penalty $g(\theta)$ that penalizes only the weights of the affine transformation at each layer and leaves the biases unregularized.
    - The biases typically require less data to fit accurately than the weights.
    - Regularizing the bias parameters can introduce a significant amount of underfitting.

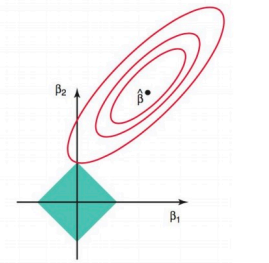
L2 Regularization

$$g(\theta) = ||\theta||_2 = \sum_k \theta_k^2$$



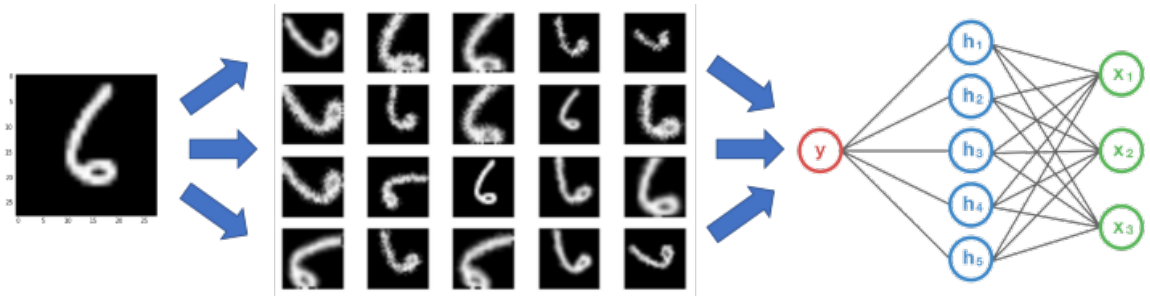
L1 Regularization

$$g(\theta) = ||\theta||_1 = \sum_k |\theta_k|$$



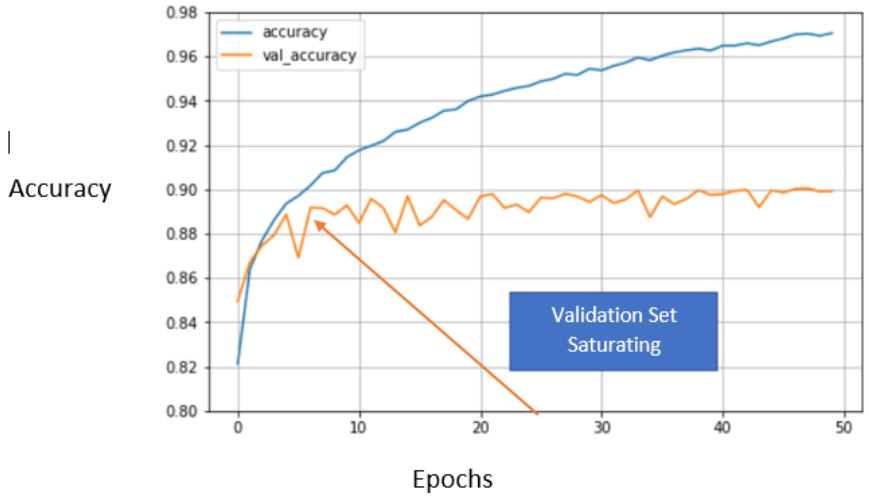
Data Augmentation

- Data Augmentation
  - Overfitting is one of the greatest challenges for deep networks. With large models and (relatively) small datasets, it becomes hard to avoid memorizing the training data.
  - Data Augmentation is a set of techniques that allows us to create more training data in a way that preserves important aspects and improve generalizability.
    - Flip, Rotation, Shift, Zoom, Scaling, Brightness etc.
    - Image data augmentation is used to expand the training dataset in order to improve the performance and ability of the model to generalize.
  - Data Augmentation has been mostly applied to images and might be more challenging in other domains.



Early Stopping

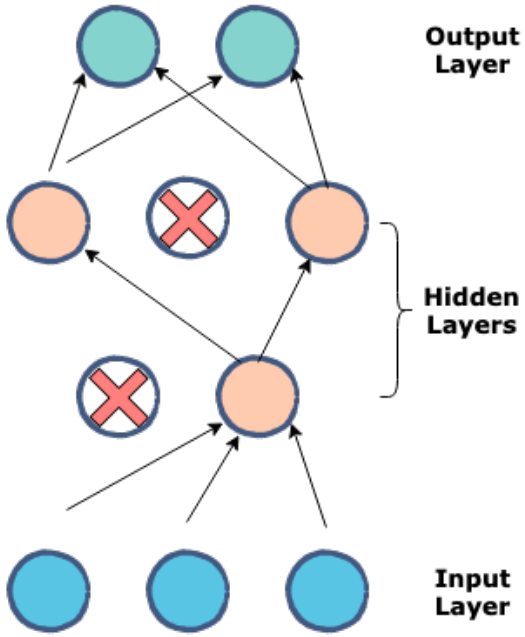
- Early Stopping
  - As training iteration increases, we are more likely to be optimizing the weights toward areas that over-fit the training data.
  - Solution: When going on training, we keep a record of the loss function on the validation data, and when we see that there is no improvement on the validation set, we stop, rather than going all the epochs. This strategy of stopping early based on the validation set performance is called Early Stopping.
- Arguments
  - monitor: Quantity to be monitored.
  - min_delta: Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.
  - patience: Number of epochs with no improvement after which training will be stopped.
  - mode: One of {"auto", "min", "max"}. In min mode, training will stop when the quantity monitored has stopped decreasing; in "max" mode it will stop when the quantity monitored has stopped increasing; in "auto" mode, the direction is automatically inferred from the name of the monitored quantity.
  - baseline: Baseline value for the monitored quantity. Training will stop if the model doesn't show improvement over the baseline.
  - restore_best_weights: Whether to restore model weights from the epoch with the best value of the monitored quantity. If False, the model weights obtained at the last step of training are used.




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

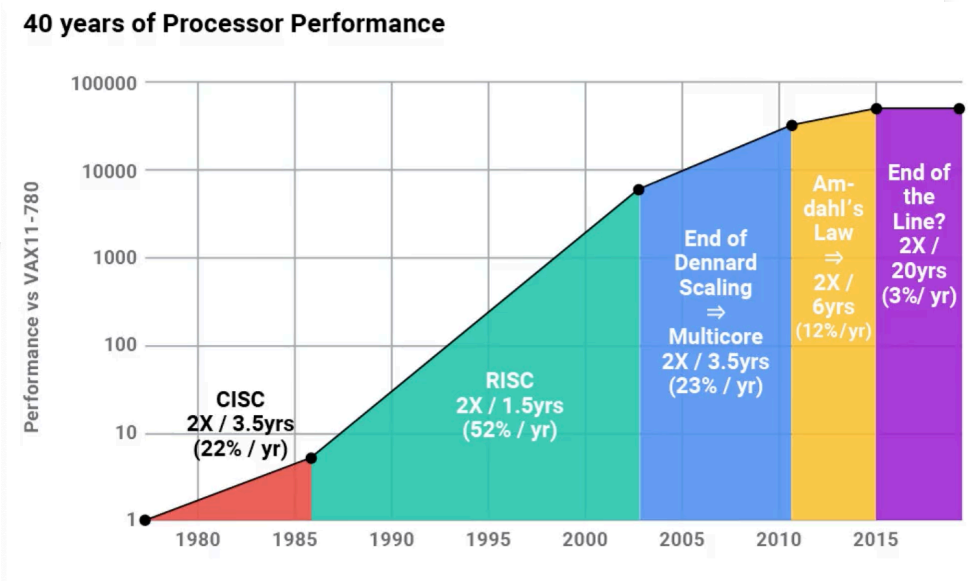
Dropout

- Problem
  - When a fully-connected layer has a large number of neurons, co-adaption is more likely to happen. Co-adaptation refers to when multiple neurons in a layer extract the same, or very similar, hidden features from the input data. This leads to overfitting if the duplicate extracted features are specific to only the training set.
- Dropout
  - In dropout, we randomly shut down some fraction of a layer's neurons at each training step by zeroing out the neuron values. The fraction of neurons to be zeroed out is known as the dropout rate r_d. The remaining neurons have their values multiplied by $\frac{1}{1-r_d}$ so that the overall sum of the neuron values remains the same.
  - Dropout regularization is a generic approach. It can be used with most, perhaps all, types of neural network models, including multilayer perceptron, convolutional neural networks, and LSTM networks.
  - The default interpretation of the dropout hyperparameter is the probability of training a given node in a layer, where 1 means no dropout, and 0 means no outputs from the layer. A good value for dropout in a hidden layer is between 0.5 and 0.8. Input layers use a larger dropout rate, such as of 0.8.



Dropped-Out Network

CPU vs GPU vs TPU



- The end of Moore's Law
  - End of exponential growth on cpu processing performance
- How to overcome?
  - GPU
    - Just like CPU, but it has thousands of tiny multiplier in a single processor
    - Super efficient on applications with massively parallel tasks
  - TPU
    - Specifically designed as a matrix processor
    - It places tens of thousands of operators connected as a very large matrix, just for doing the one task: matrix operations
    - TPU executes massive matrix operations with a very large pipeline, with significantly less memory access
- https://storage.googleapis.com/nexttpu/index.html