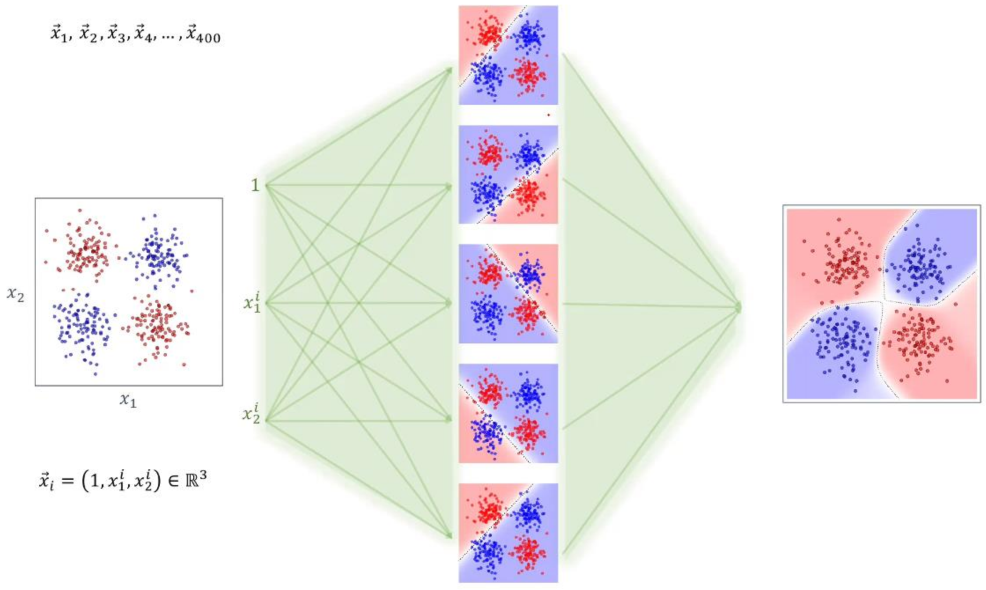
Deep Learning

Data-Driven Approach – models learn directly from data instead of explicitly programmed rules or statistical measurements.

1 neuron ≡ Linear Regression

Dense layers ≡ Fully connected ≡ Multilayer Perceptron

XOR network *– from linearity to nonlinearity:*



$\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4, \ldots, \vec{x}_{400}$
x_2
x_1
$\vec{x}_i = (1, x_1^i, x_2^i) \in \mathbb{R}^3$
1
x_1^i
x_2^i

- Each neuron in the first hidden layer is a linear combination of the input features and the weights - where the weights are the scalars that determine the contribution of each input feature to the neuron's output.

    - In the next layer each neuron receives a linear combination of linear combinations – This creates a more complex, **nonlinear** relationship between the input features and the output of the neural network.
      Therefore, **ANN** can deal with non-linear data without feature-engineering.

Sequential models are a type of neural network architecture in which the layers of the network are stacked in a linear sequence. Each layer takes the output of the previous layer as its input. Sequential models are typically used for **feedforward** neural networks in which data flows in one direction through the layers.

Activation Functions: in ANN, the activation function of a neuron defines the output of that neuron – given a weighted sum of inputs. **Must be non-linear**.

Regularization:

- Limiting the flexibility of the model in order to achieve better generalization.

- Regularization affects only the train-set, not the test.

- **L1**, also known as **Lasso** regularization, adds a penalty term to the loss function that is proportional to the absolute value of the weights. The penalty term encourages the weights to be small in magnitude, which helps prevent **overfitting**.
  The L1 penalty has the effect of making some weights exactly zero, effectively performing feature selection by removing features that are not important for the prediction task.

- **L2**, also known as **Ridge** regularization, adds a penalty term to the loss function that is proportional to the square of the magnitude of the weights.
  Unlike L1, the L2 penalty does not make any of the weights exactly zero.
  This penalty encourages smaller and more evenly distributed coefficients, which can lead to smoother and less complex models.

- The regularization strength is controlled by a hyperparameter, often denoted by **lambda**. Larger values of lambda correspond to stronger regularization.

- With **dropout** - the model randomly disables **different** neurons in each batch’s forward-propagation. So that eventually, all neurons take part in the learning process.
  By disabling some neurons during each iteration, the remaining neurons are forced to process all relevant information, even if other neurons have already processed it.
  This prevents the model from relying too heavily on any specific neuron, which can promote better generalization.

- Regularization causes each neuron in higher level to be a combination of more features. making it harder to interpret how the network arrives at its decisions.

- To account for the fact that during the test’s forward-propagation, neuron will suddenly receive more input than they did during training (due to **dropout**) causing them to fire output they never did before, we multiply the activation function by the dropout factor, or - better yet, divide the activation function by that factor during training.

Batch Normalization:

- batch normalization is a form of **regularization**.
  The idea behind it - is to normalize the input of a layer before they are passed through the activation function. This helps to ensure that the activations have a consistent distribution and that they are centered around 0, with a standard deviation of 1.

- Batch normalization is performed batch-wise: it uses the statistics (mean and variance) of each mini-batch to normalize the activations. This means that the normalization is different for each mini-batch, and the normalization statistics are updated after each batch.

- The reason for this is that the distribution of activations can change significantly during the training process, as the parameters of the network are updated. By using the mean and variance of each mini-batch to normalize the activations, batch normalization helps to ensure that the distribution of activations is consistent and well-behaved, even as the parameters change.

- This consistency is important because it helps to make the optimization problem more stable, reducing the risk of overfitting and making the training process more efficient. By normalizing the activations this way, batch normalization helps to ensure that the network is not relying on any particular activation distribution, making it more robust to changes in the data and therefore **prevent overfitting**.

- Although performing batch normalization requires additional computations during training, using this layer can often accelerate the training process by fixing mistakes in the network, such as bad weight initializations and by accelerating convergence.

- batch normalization and dropout should not be implemented together at the same network.

- Batch-norm includes **2 learnable parameters** and **2** calculated ones (**non-trainable**)

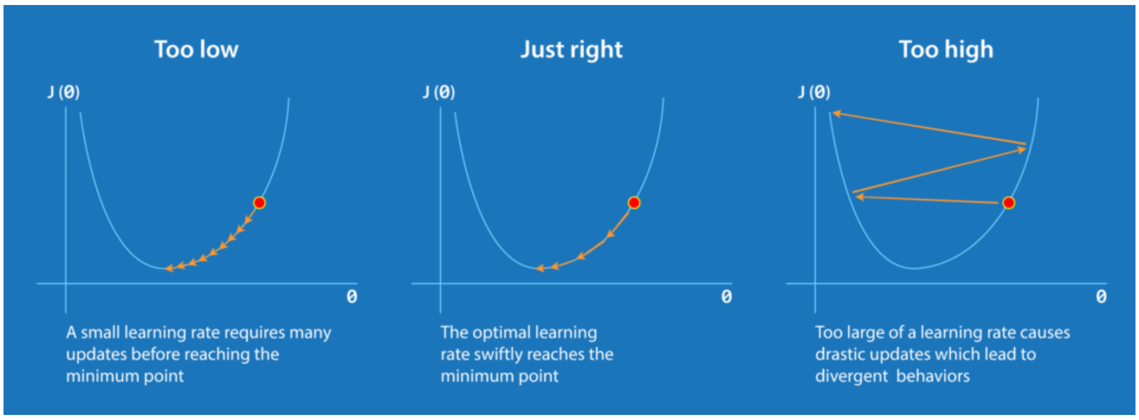
- Problem arises when running through the test data since the 2 learnable params are adapted to each batch separately - the distribution of the test data may differ from that of the training data. This is called **batch effect**. To address this issue, during **inference** (when using the trained model to make predictions on new data), we can use the **population statistics** (the global mean and variance of the entire dataset) that were calculated during training to normalize the activations of the test data.

Moving Average:

- A moving average is a statistic that captures the average change in a data series over time.
- **Weighted Moving Average** – we'll give recent values bigger wights to reflect their greater significance.
- **Exponentially Weighted Moving Average** – importance of values will drop exponentially the further they get in time.
  - We'll use this measure in **batch normalization** to calculate the STD and Mean throughout the training. Also, in Adam [see below]

Optimization:

- **Gradient-Descent** is an optimization algorithm used to minimize a **cost function** [which measures prediction error] by iteratively adjusting the model parameters in the direction of the **negative gradient** [=the direction of the steepest descent of a cost function] as to reduce the error.
  - **learning-rate** is the size of the steps taken towards the minimum point.
    - too high **lr** may result in overshooting the **minima**, so it will not be able to converge – the large updates to the **lr** causes an **Exploding Gradient**.


Too low
J (θ)
θ
A small learning rate requires many updates before reaching the minimum point
Just right
J (θ)
θ
The optimal learning rate swiftly reaches the minimum point
Too high
J (θ)
θ
Too large of a learning rate causes drastic updates which lead to divergent behaviors

- **Stochastic Gradient Descent [SGD]** gradient descent for a small batch of data in each iteration.
  - Goes through back-propagation several times → computes slower.
  - Bigger batches require more memory ≡ SGD requires less memory.
  - SGD converges faster (for each iteration) but its path is noisier.
  - Avoids **Local minima** better.
  - Can better learn nuance in the data.

- **Plateau**: when the accuracy of the model stops improving mid-training because the model has already learnt the most important patterns in the training data – leading to **saturation** or plateau in accuracy.
    - the ability of SGD to learn nuance can help the model to optimize the weights to fit less significant patterns and potentially solve the Plateau problem.

- **AdaGrad** is an adaptive gradient optimization algorithm that adjusts the **learning-rate** for each parameter in the model based on the historical gradient information.
    - it starts with a large learning rate for each parameter and decreases it over time as the gradient information for that parameter accumulates. This allows AdaGrad to handle sparse data efficiently, but it also means that the learning rate decreases too quickly for some parameters and becomes too small to make progress.
    - AdaGrad prevents the gradient from jittering around the loss function.

- **RMSprop** - Root Mean Square Propagation - a gradient descent optimization algorithm that adjusts the learning rate for each parameter in the model based on a **moving average** of the magnitudes of recent gradients. It aims to mitigate the decreasing learning rate problem in AdaGrad by using a more sophisticated learning rate adaptation method.

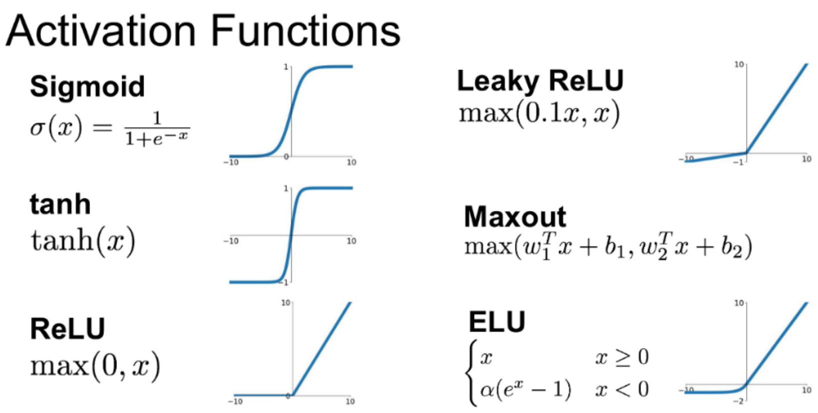
- **Momentum** is a technique used in gradient descent to help speed up convergence and avoid getting stuck in local minima. It does this by adding a fraction of the previous gradient updates to the current one, so that the updates have a "memory" of the previous gradients. This helps to speed up convergence and improve the performance of the optimization algorithm.

    - It’s Like a ball rolling down the loss function - after rolling down the hill, it will keep rolling in that direction with a little extra speed because of the momentum from the fall. That makes it easier for the ball to get out of the dips and valleys and roll all the way to the bottom of the hill faster – converge to a global minimum.

- **Adam** is an adaptive gradient-based optimization algorithm that combines the ideas of **gradient descent, momentum,** and **RMSprop** to provide a fast and effective optimization method for training deep neural networks.

- In Adam, the gradient updates are combined with the past gradient information to create a **running average** of the gradients, which helps to speed up convergence and avoid getting stuck in local minima. The momentum term in Adam is adaptive, meaning that it adjusts its magnitude based on the historical gradient information and the current gradient. This allows the algorithm to have both the stability of traditional momentum and the adaptivity of RMSprop.

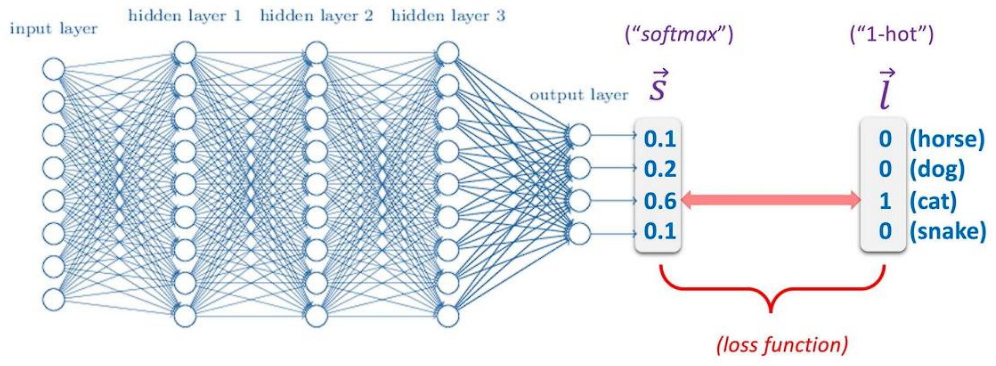
- Gradient descent adjusts **all** the weights in every iteration.
- We'll pick a loss function that penalizes larger errors [such as logistic-loss].
- **Local minima** are rarely a problem in deep learning due to the number of parameters.



- **Saturation** – when no more of something can be absorbed.
  - When a neuron predominantly outputs values close to the asymptotic ends of the bounded activation function.
  - the gradient of a **sigmoid** vanishes both when its input is samll and when it's large. $\rightarrow \text{output} = 0 \vee 1 \rightarrow$ Saturation → **Vanishing Gradient** → no optimization.
  - Saturation damages both the information capacity and the learning rate ability.
  - Sigmoid is not zero-centered.
    - We'll use Sigmoid only when we want a probability distribution – usually in the output layer of a binary classification problem.
  - **tan-h** is zero centered but still kills gradients when saturated.
  - **ReLU** – no saturation for positive values. Calculated quickly and converges fast.
    - Not zero-centered and saturates for all negative values.
    - **Dead ReLU** occurs when the input is **always** negative $\rightarrow \text{output} = 0$
  - **Leaky ReLU** solves the above saturation problem by introducing a slope for negative values.
  - **ELU**: all benefits of ReLU, does not die, closer to zero-mean outputs.

Multi-Class:

- In multi-class classification we'll use an activation function like **SoftMax** for the output layer to calculate the output for each class relatively to other classes – it will provide us with a probability distribution (not true probability of the class though).
- Then we'll use a function like **cross-entropy** to compare the output from the softmax with the true values for each class and **back-propagate** the result.



"cross-entropy" loss

$$loss(\boldsymbol{s},\boldsymbol{l}) = -\sum_i l_i \log(s_i) = -\log(s_y)$$

"softmax"

$$s_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

"1-hot"

$$l_i = \begin{cases} 1, & i = y \\ 0, & i \neq y \end{cases}$$

Weight Initialization:

- If we initialize all the weights in one uniform value → all neurons will receive the same values and essentially be equivalent to just one neuron.
- We want to initialize the weights in a **normal distribution** with a $random()$ function.
- Initializing too **small** weights will cause a **vanishing gradient**.
- Initializing too **large** weights will result in an **exploding gradient**.

- **He [Kaiming] Initialization** is a method for initializing the weights of a neural network, which uses a Gaussian distribution with zero mean and a variance proportional to the number of input units. It is particularly useful for networks with non-linear activation functions like **ReLU**, as it can help to prevent saturation (by preventing vanishing gradients) and improve learning speed.
- **Xavier** is similar but doesn't work as well with ReLU.

Image classification problem – what is the object seen in the image?

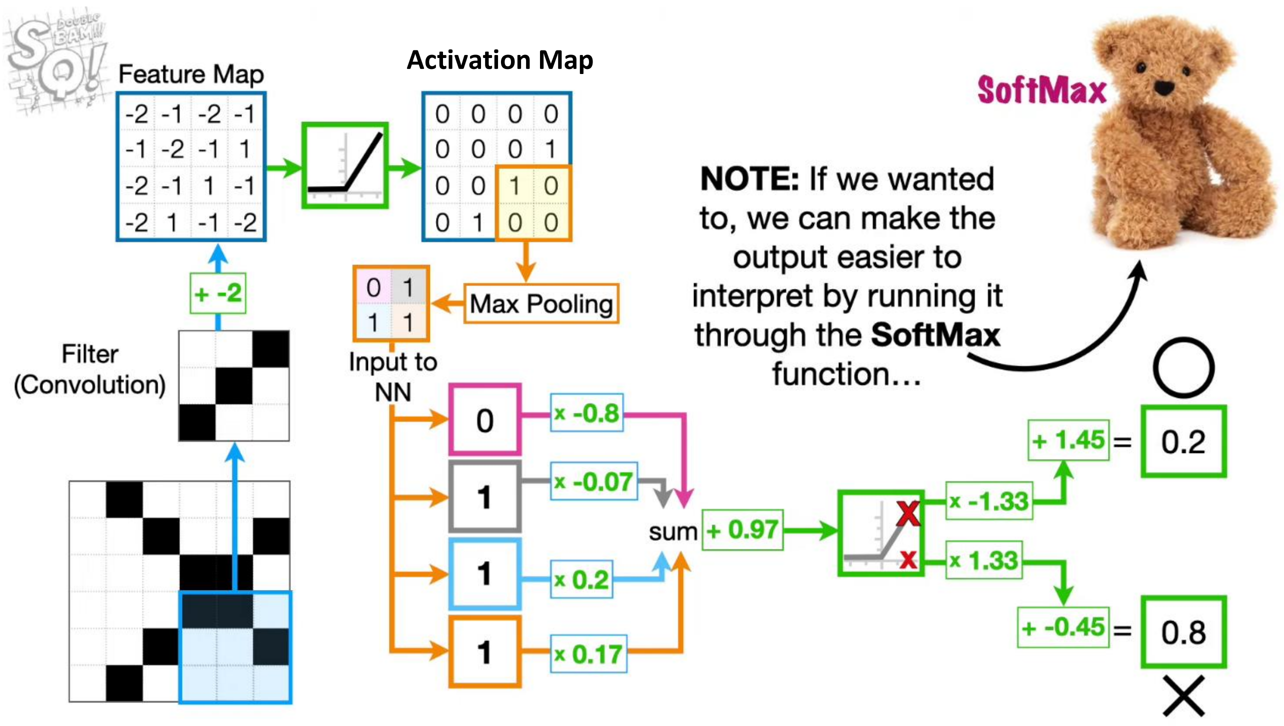
Localization problem – where is the object located within the image?

CNN

- **Convolution** is a mathematical operation, where a **filter** is applied to overlapping regions of an image to produce a **feature map**.
- Each convolution results in a feature map which provides a certain feature from The image.

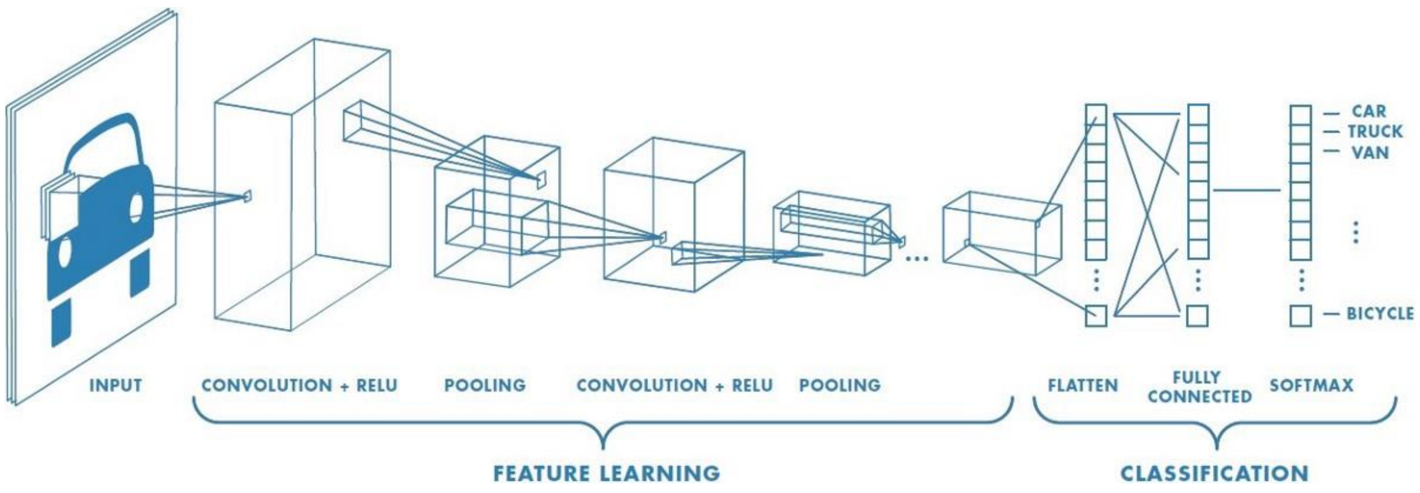
- The number of filters in each layer will determine the number of features provided from that layer.
- Each filter is meant to study a different aspect of the input image and generate a unique feature map that highlights that aspect.
- More filters will provide us with more information about the image, but too many will result in irrelevant data.

- **Max Pooling** is a pooling operation that calculates the maximum value for patches of a feature map, and uses it to create a downsampled (pooled) feature map. It is usually used after a convolutional layer.
  - Max pooling is done to prevent overfitting by providing an abstracted form of the representation. As well as to reduce the computational cost by decreasing the number of parameters and provides basic **translation invariance** to the internal representation.
    - **Translation invariance** means that if the input is shifted (translated) slightly, the output should not change drastically. In other words, the network should be able to recognize the same object regardless of where it a appears in the input image.



    - Max pooling can be problematic because down sampling - results in a lot of earlier neurons that do not participate in the learning process (those that were not pooled). also causes features to lose touch with their original position.

- ***Same padding*** is used when we need an output of the same shape as the input. If the values for the padding are zeroes then it can be called ***zero padding***.
  In max-pooling it is not appropriate to use padding.



- Each filter has the same **depth** [number of **channels**] as the previous layer.
- Each convolution will result in 1 scalar by performing a dot product between the filter and a chunk of the image (and adding a bias).
- **Flatten** transforms the activation maps from a **3D Tensnor** to a **1D array**.
    - It simply reshapes the input into a vector by stacking all the elements in the feature maps sequentially.

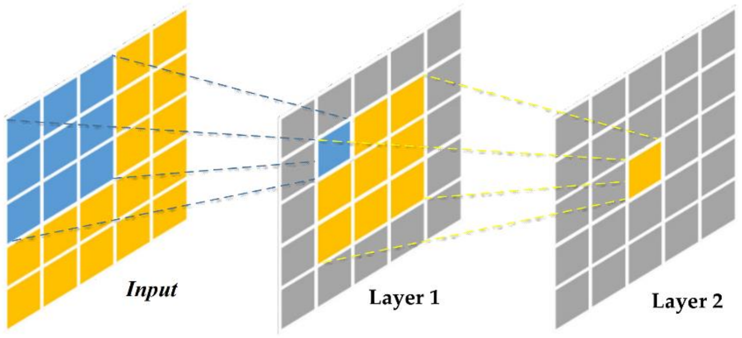
Global Average Pooling:

- Replaces flatten function and converts the 3D feature maps tensor into a 1D array.
- Transforms each feature map into a single scalar by computing its average.
- Fitting for a classification problem and not localization problems.
- This pooling will produce a number of neurons as the number of feature maps.

- *It's possible to use **KNN** on the classification stage of the CNN*
    - *The input data for the KNN will be high level feature vectors and not pixels.*

Receptive field:

A receptive field in a Convolutional Neural Network (CNN) refers to the portion of the input image that a particular neuron in the network is able to "see" and uses to compute its activation. It represents the region in the input image that has a direct impact on the output of a particular neuron in the CNN. The receptive field is determined by the size of the filters used in the convolutional layers, the stride and padding applied, and the architecture of the network.



In early layers – each filter sees a small part of the original image and represents a small detail [**low-level feature**].

In deeper layers – each filter will see a combination of filters which are already a combination of filters from earlier layers – each covering different part of the image. A filter here will represent a more significant detail of the image [**high-level feature**] – like a wheel of a car.

The formula for RF when padding = 'same':

$$\textbf{Receptive Field} = \begin{cases} \textbf{kernel}_{size} & \textbf{for first conv layer} \\ \textbf{RF} += (\textbf{kernel}_{size} - 1) \times \prod_{i=2}^{\text{this layer}} (\textbf{stride}_i) & \textbf{for next layers} \end{cases}$$

first conv layer: each neuron sees exactly its $\textbf{kernel}_{size}$
every one after that adds $(kernel_{size} - 1) \times current\ stride \times$
$all\ strides\ that\ came\ before\ except\ for\ the\ first\ conv's\ stride.$

Output shape of hidden layers: //indicates number of neurons

Default values:

Parameter	Conv2D	MaxPooling2D
Kernel	-	`pool_size = 2`
Padding	`'valid' // none`	`'valid' // none`
Stride	`1`	`None = pool_size = 2`

$$\begin{aligned} \textbf{input shape} &= \textbf{N} \times \textbf{N} \\ \textbf{filter size} &= \textbf{F} \times \textbf{F} \\ \textbf{padding} &= \frac{\textbf{F}-\textbf{1}}{\textbf{2}} \end{aligned} \qquad \textbf{output shape} = \begin{cases} \left\lfloor \dfrac{\textbf{N} - \textbf{F} + \textbf{2} \times \textbf{padding}}{\textbf{Stride}} \right\rfloor + \textbf{1} & \textbf{padding} = \textbf{'same'} \\ \left\lfloor \dfrac{\textbf{N} - \textbf{F}}{\textbf{Stride}} \right\rfloor + \textbf{1} & \textbf{padding} = \textbf{'valid'} \end{cases}$$

- where $padding = 1$ means adding 1 to each side – total of 2 rows and 2 columns.
- The above formula refers to N as the original shape of the image, before padding.

Trainable Parameters:

Number of parameters in Conv2D layer:

$$filter_{size^{3D}} = width \times length \times depth$$

$$biases = |filters|$$

$$params = |filters| \times filters_{size^{3D}} + biases$$

how much each layer adds to the sum of neurons and parameters with $padding = \text{'}same\text{'}$:

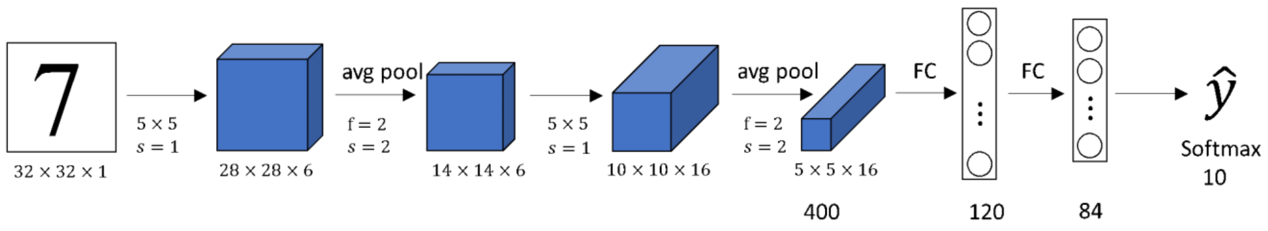
Layer	Neurons	Parameters
Input	$input_{shape^{3D}}$	*0*
Conv2D	$input_{shape^{2D}} \times \|filters\|$	$params$
Batch Normalization	*0*	$4 \times \|filters\|$ **(half of this is non-trainable)**
Activation Function	$\|previous\ nodes\|$	*0*
MaxPooling2D	$\frac{input_{shape^{3D}}}{4}$	*0*
Flatten	$input_{shape^{3D}}$	*0*
GlobalAveragePooling2D	$\|channels\|$	*0*
Output [Dense]	$\|nodes\ in\ the\ last\ dense\ layer\|$	$input_{neurons} \times output_{neurons} + output_{neurons}$

- Activation Function adds neurons only if it's done separately from the conv layer.
- Dropout does not affect the number of parameters or neurons.
- *batch-norm in CNN is done per activation map.*
- $input_{shape^{2D}} \equiv channels_{shape} \equiv (width \times length)\ of\ input$
- $|channels| \equiv depth\ of\ input$

Layer	RF	OPS
First Conv layer	$kernel_{size}$	$\frac{neurons \times params}{\|filters\|}$
Conv2D	$RF{+}{=} (kernel_{size} - 1) \times \prod_{i=2}^{this\ layer} stride_i$	
Batch Normalization	-	$4 \times \|channels\|$
Activation Function	-	$\|neurons\|$
MaxPooling2D	$RF{+}{=} (pool_{size} - 1) \times \prod_{i=2}^{this\ layer} stride_i$	$\frac{neurons \times params}{\|filters\|}$
Flatten	-	-
GlobalAveragePooling2D	-	$input_{shape^{3D}} (= w \times h \times d)$
Output [Dense]	-	$input_{neurons} \times output_{neurons} + output_{neurons}$

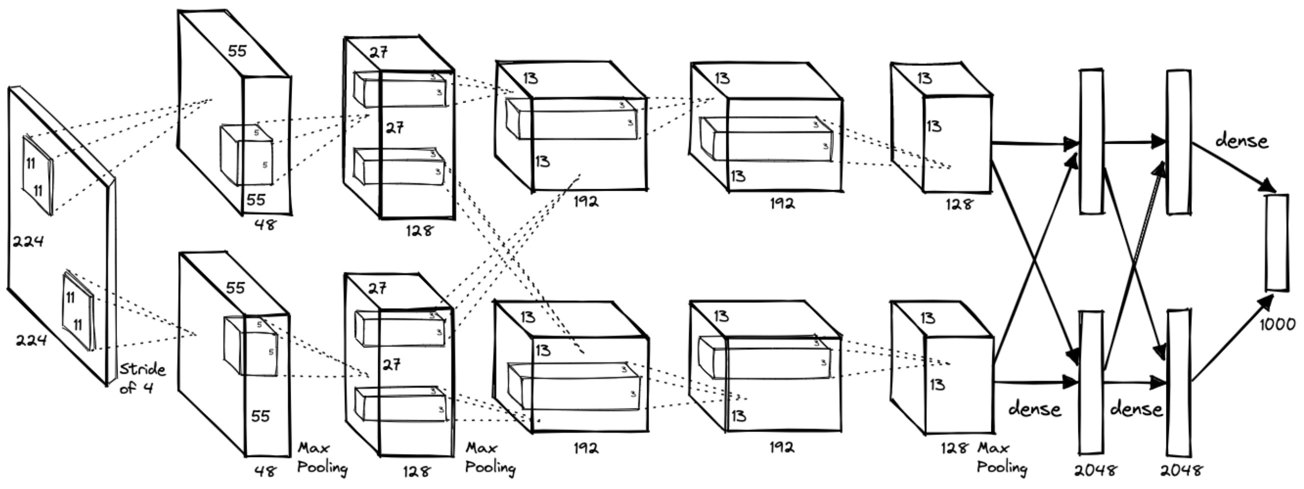
LeNet-5:

- LeNet-5 was one of the first CNNs to achieve good performance in image classification tasks. Its architecture includes several innovative elements that later became standard in CNNs [such as pooling].



AlexNet:

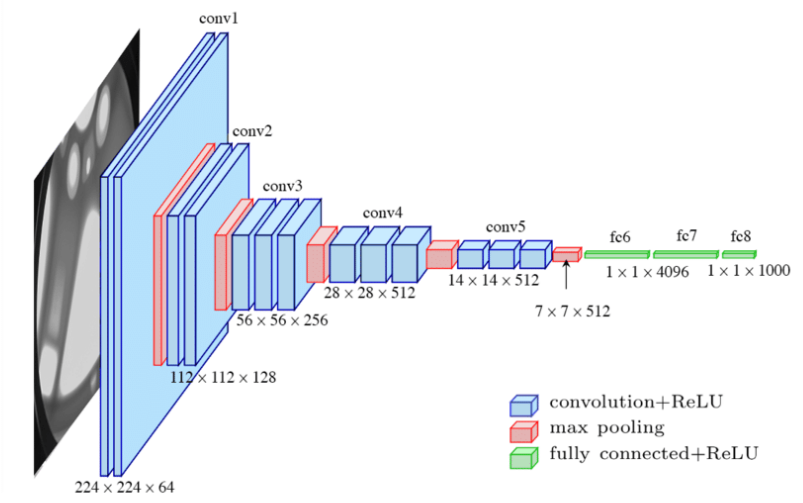
- AlexNet was one of the first large-scale CNNs to achieve state-of-the-art performance on the **ImageNet** dataset.
- It used the **ReLU** instead of the previously popular sigmoid activation function.
- It introduced the concept of **dropout** regularization to reduce overfitting.
- It made use of **data augmentation** techniques, such as horizontal flipping and random cropping, to increase the size of the training set.
- AlexNet introduced the concept of parallel computation through the use of two GPUs



VGG:

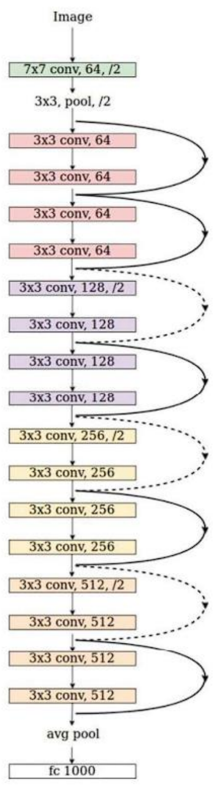
- In VGG we double the number of filters in each Conv2D to make up for the loss of data (details), after MaxPooling, with more complex combinations of features.
- $\frac{1}{4} image,\ 2 \times filters.$
- VGG downsized the filters to 3×3.

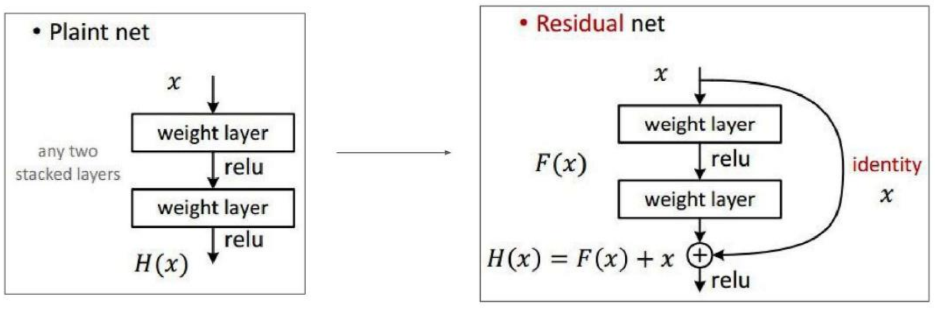
  - Instead of starting out with 7 × 7 filters, we'll start with 3 × 3 and after 3 conv layers each node will have a receptive field of 7 × 7
  - this will allow for more complex features due to the nonlinearity of multiple conv layers.
  - Less parameters to learn, but $3 \times the\ neurons$.
- VGG demonstrated how critical the depth of a network is to its performance.
- VGG is highly expansive in memory and computing time.

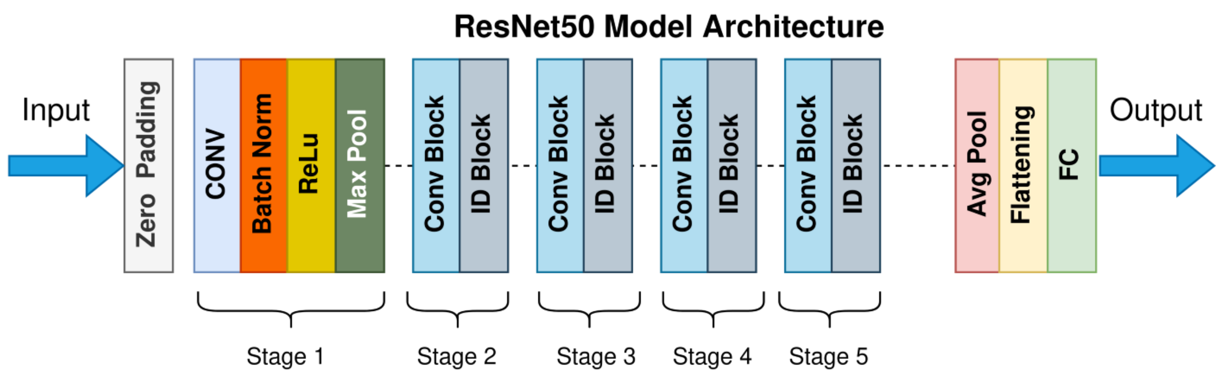


ResNet:

- **Identity**: when the network learns a specific filter.
- **residual block** consists of a series of convolutional layers summed together, with shortcut connections that skip one or more layers.
- Deeper networks are at higher risk of the **vanishing gradient problem**
  - Residual nets use addition instead of multiplication to prevent this problem.
- ResNet introduced the method of **skip connections** for deep networks in order to prevent overfitting.

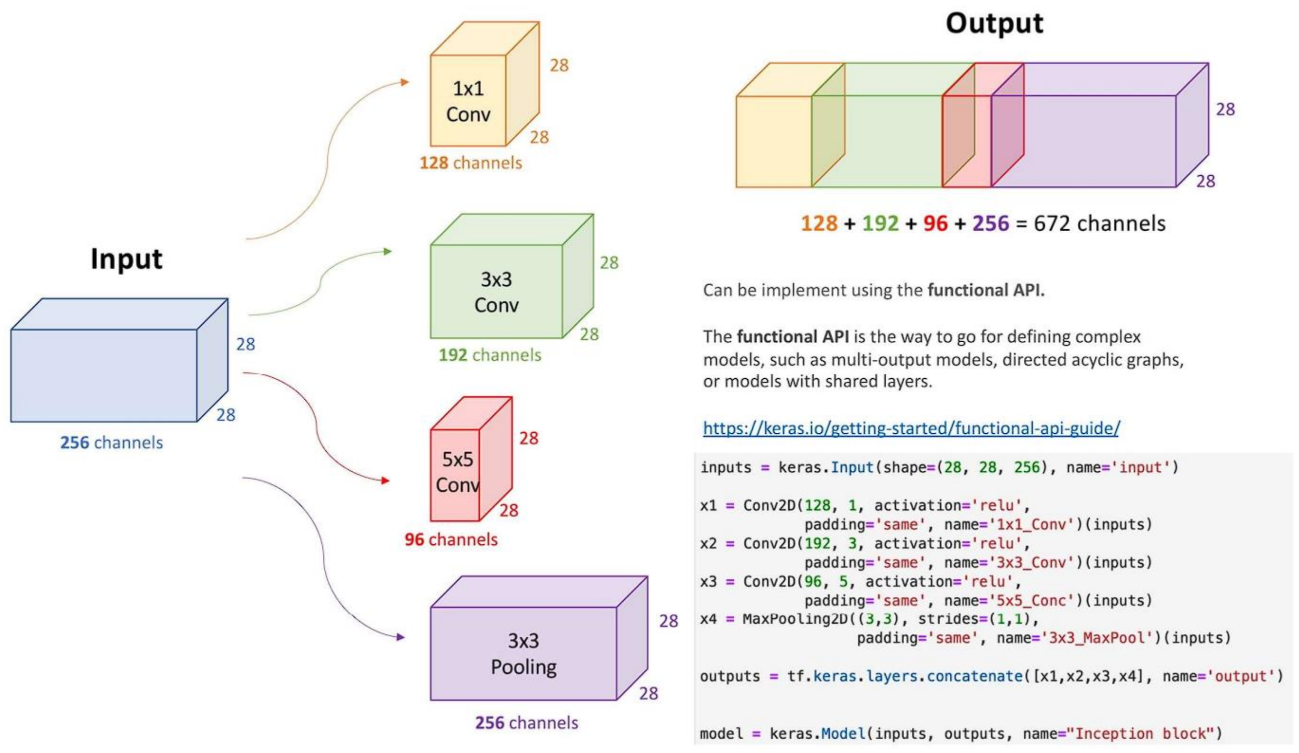






InceptionNet [GoogLeNet]:

- Non-sequential CNN.
- Dramaticly reduced the number of parameters.
- Uses different filter sizes simultaneously allowing the network to capture features at multiple scales in a single layer. The outputs of these filters are then **concatenated** and fed as input to the next layer [as illustrated bellow].



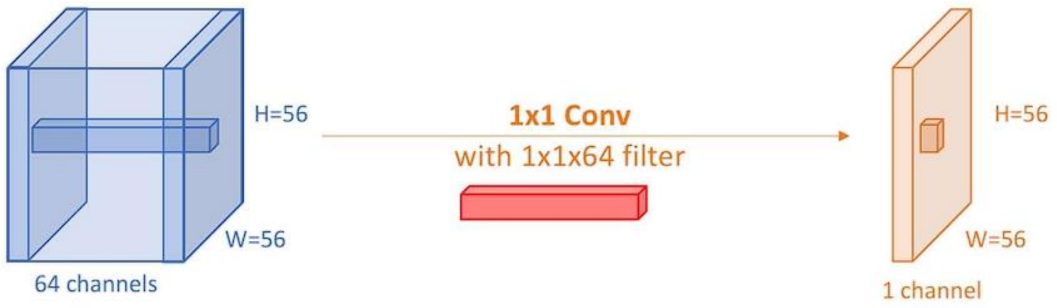
Can be implement using the **functional API.**

The **functional API** is the way to go for defining complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.

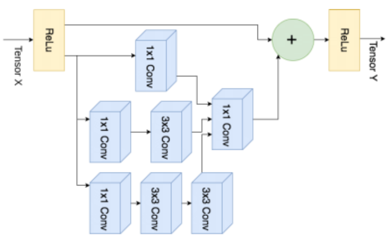
https://keras.io/getting-started/functional-api-guide/

```
inputs = keras.Input(shape=(28, 28, 256), name='input')

x1 = Conv2D(128, 1, activation='relu',
            padding='same', name='1x1_Conv')(inputs)
x2 = Conv2D(192, 3, activation='relu',
            padding='same', name='3x3_Conv')(inputs)
x3 = Conv2D(96, 5, activation='relu',
            padding='same', name='5x5_Conc')(inputs)
x4 = MaxPooling2D((3,3), strides=(1,1),
                  padding='same', name='3x3_MaxPool')(inputs)

outputs = tf.keras.layers.concatenate([x1,x2,x3,x4], name='output')


model = keras.Model(inputs, outputs, name="Inception block")
```

- maxPooling in GoogLeNet uses $stride = 1$ to keep the shape as is so it can be concatenated with the other filters.
- The concatenatoin allows the next layer to choose which filter size was best (due to each filter being connected with weights to every channel in the previous layer).
- GoogLeNet introduced **Pointwise Convoltuion** – a method of **dimensionality reduction** that uses a filter of size 1×1 in order to create a representation of the previous layer with less channels (and reduce parameters and number of operations).
- Each node in the new layer will be a linear combination of the every channel in the previous layer (in the same location).

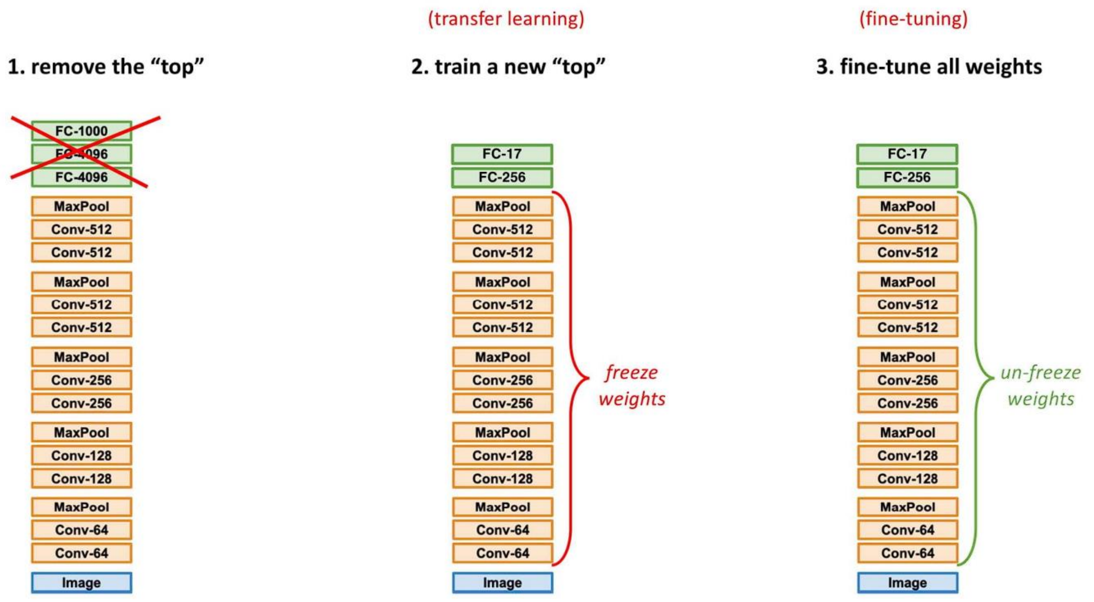


- **Inception-ResNet** is a deep neural network architecture that combines the Inception module with residual connections.



Transfer Learning:

- the process of using a pre-trained model as a starting point for a new task or problem, rather than training a model from scratch.
  - The process of tuning some of the parameters from a pre-trained model to fit the new task, is called **Fine-Tuning**.
- Usually done when we don't have a lot of data.
- We'll freeze most of the model and only train a small number of layers, to avoid overfitting from training a big model on a small dataset.
- If the data is similar to the one the original model was trained with – we can use more of the original network.
- If the data is different from the og data – we should take less of the og network – only the first few layers might be general enough to fit the new data.
- There are many ways to do tranfer learning and one must find the best point from which to re-train the model and choose which layers are best kept frozen.
- During Fine-Tuning – learning rate should be decrased.

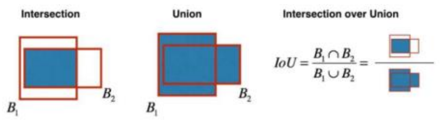


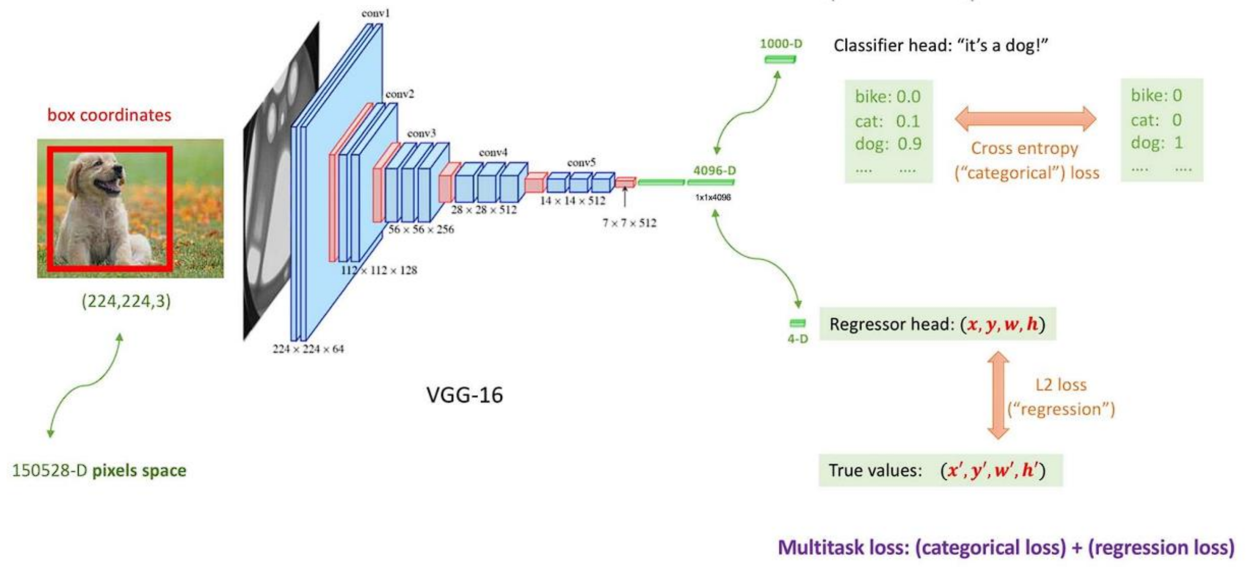
- Transferred networks have a minimum and maximum input shape.

- *To deal with overfitting - it's better to add regularization rather than decrasing the size of the model, though some networks might really be too complex for the problem at hand.*

Localization and Detection:

- Challenge: multiple objects. Bigger challenge: unkown number of objects.
- **Object localization**: locating an exact number of objects.
- **Object Detection**: identifying and locating an unkwon number of objects.
  - A good way to measure the performance of object detection is **Intersection over Union**.

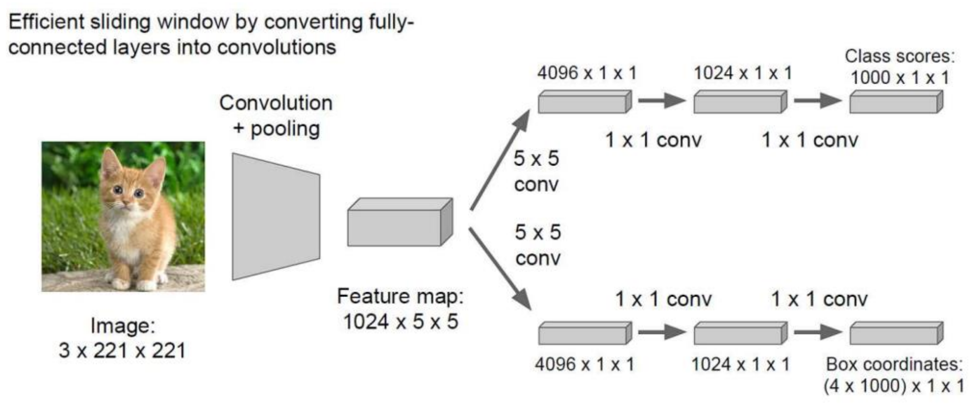




- **OverFeat** is designed for object localization tasks. It can predict bounding boxes around objects in an image and classify them into different categories.
  - Avoids resizing images by using a **Sliding Window**.
    - Each window returns a prediction of its contents [**classification**], typically in the form of a probability distribution over the possible classes in the dataset **and** the coordinates of a **bounding box** that tightly encloses the objects of interest.
      - The loss would need to take both into account.

- ***"In CNNs there is no such thing as 'fully connected layers', there are only convolutional layers with a 1×1 kernel"***

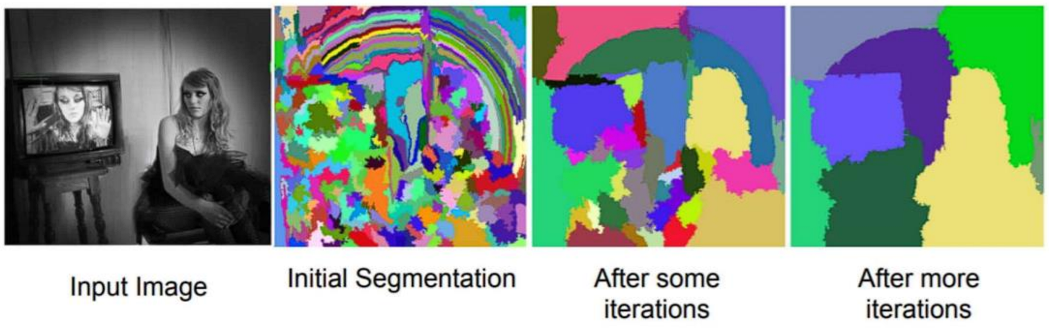
- OverFeat is a **Fully Convolutional Network**.
- Convolutions don't care about the size of the image, thanks to the use of filters and therefore FCN can work with images of different sizes.


Efficient sliding window by converting fully-connected layers into convolutions
Image: 3 x 221 x 221
Convolution + pooling
Feature map: 1024 x 5 x 5
5 x 5 conv
5 x 5 conv
4096 x 1 x 1
1 x 1 conv
1024 x 1 x 1
1 x 1 conv
Class scores: 1000 x 1 x 1
4096 x 1 x 1
1 x 1 conv
1024 x 1 x 1
1 x 1 conv
Box coordinates: (4 x 1000) x 1 x 1

- **segmentation** - The process of dividing an image into multiple regions or segments, each corresponding to a different object or part of the image.
- **region proposal** - A process of generating potential object locations in an image, which are then passed to a classifier for further processing.
- **selective search** – A **classical** computer-vision algorithm that combines low-level cues such as colour, texture and edges to generate a set of object proposals in an image, which can then be passed to a classifier for object detection.

Selective search (region proposal)

- Uses **bottom-up grouping** of image regions to generate a hierarchy of small to large regions.
- The goal is to capture **all possible objects locations**.

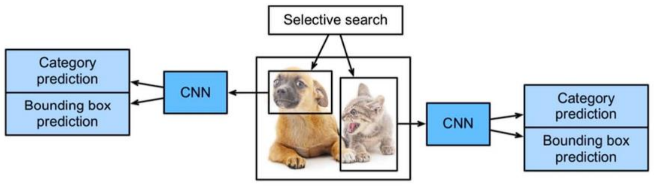


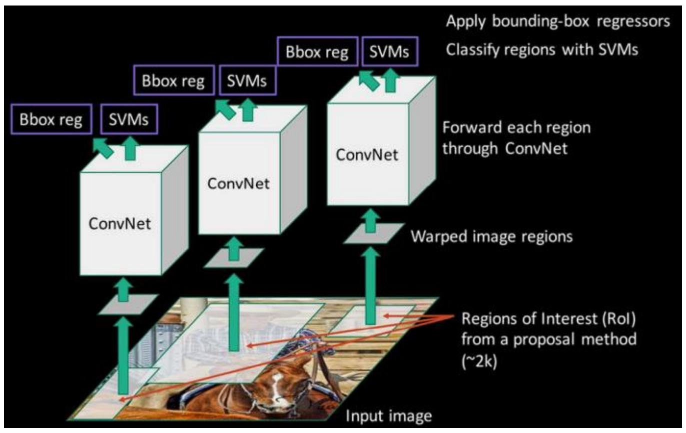
Step 1: **Generate initial sub-segmentation.**
Step 2: **Recursively combine similar regions into larger ones.**
Step 3: **Convert regions into boxes**

- **Region-based CNN [RCNN]** is an object detection framework that uses **selective search** for **segmentation** and a CNN for feature extraction.
  - The extracted features are passed to **Support Vector Machine [SVM]** to classify the object within the region.
  - The region proposals are refined using a regression model to improve the localization accuracy of the detected objects.
  - <u>Downsides:</u>
    - an overly complex framework (too much work in too many steps).
    - Slow at test-time due to independent forward passes of the CNNs.
    - selective search computes slowly.
    - training is not end to end – CNN features are not updated in response to SVMs and regression.

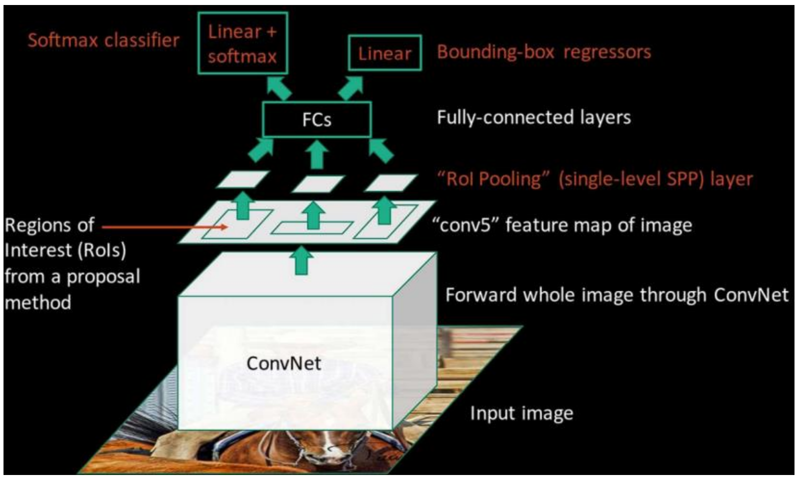
Region-based CNN

R-CNN, Fast R-CNN, Faster R-CNN

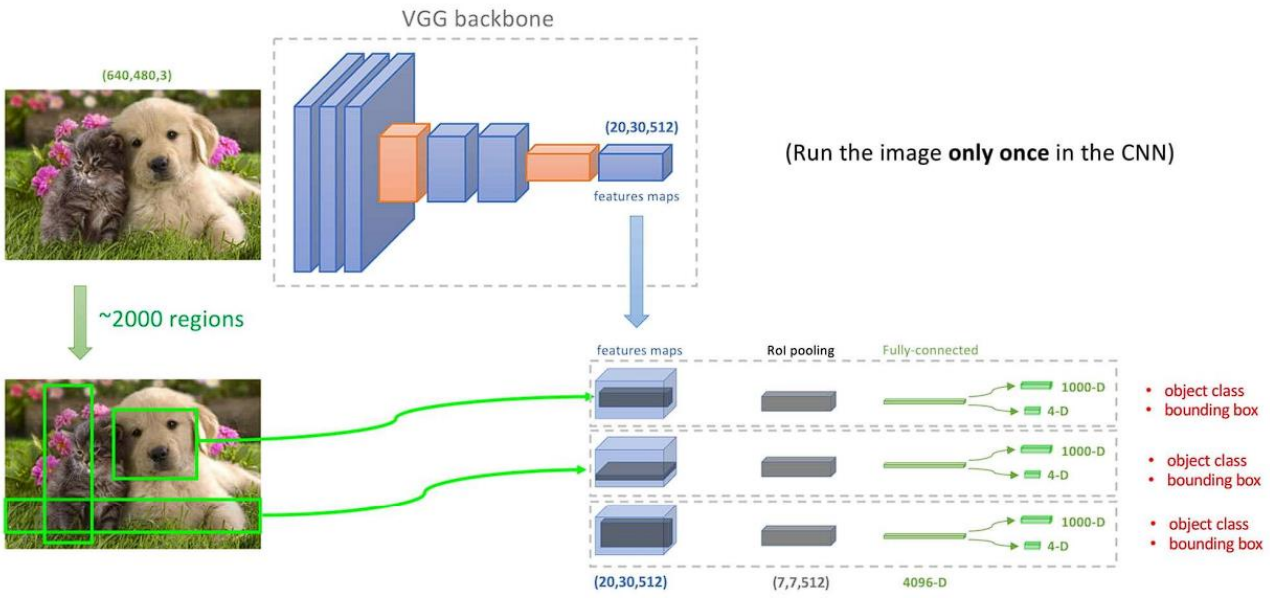



Apply bounding-box regressors
Classify regions with SVMs
Bbox reg
SVMs
Bbox reg
SVMs
Bbox reg
SVMs
ConvNet
ConvNet
ConvNet
Forward each region through ConvNet
Warped image regions
Regions of Interest (RoI) from a proposal method (~2k)
Input image

- **Fast R-CNN**:
  - **RoI Pooling**: Instead of extracting features from each object proposal using a separate forward pass through the CNN, Fast R-CNN uses RoI (Region of Interest) pooling to extract features from the shared CNN for each proposal. RoI pooling allows for efficient feature extraction by sharing the same convolutional features for all proposals within an image, and reduces the computation needed for feature extraction.

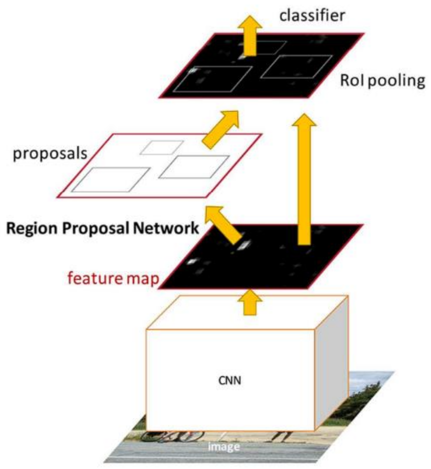

Softmax classifier
Linear + softmax
Linear
Bounding-box regressors
FCs
Fully-connected layers
"RoI Pooling" (single-level SPP) layer
Regions of Interest (RoIs) from a proposal method
"conv5" feature map of image
Forward whole image through ConvNet
ConvNet
Input image

  - Instead of using SVM – Fast-RCNN uses **fully connected layers** for classification.
  - Uses **transfer learning**.

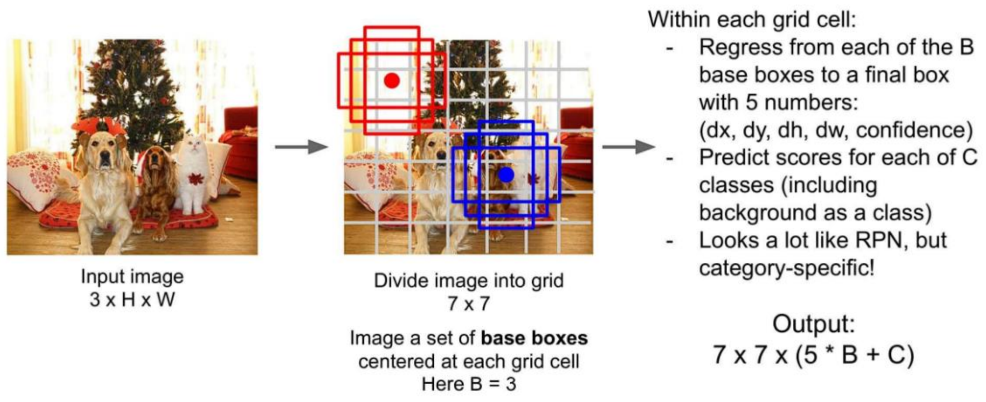


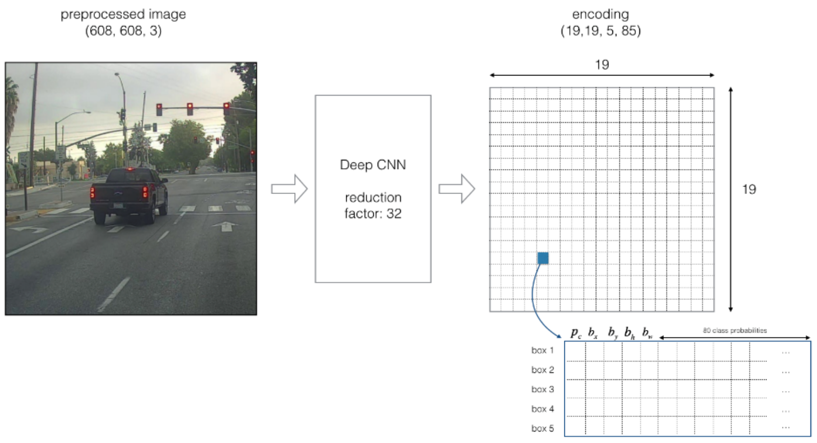
  - Still not fast enough for real-time systems.
    - Still uses **Selective Search**.

- **Faster R-CNN**: replaces Selective Search with **Region Proposal Network**
  - Still uses selective search for training but not for testing.
  - Much faster than previous frameworks, still not good enough for real-time though.



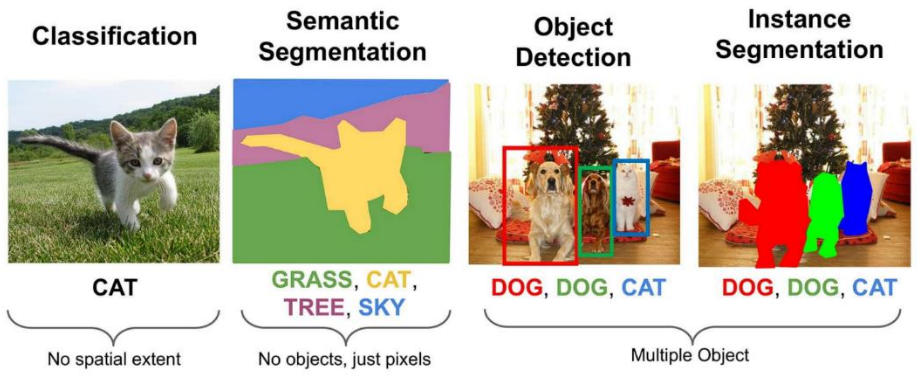
- **YOLO [You Only Look Once]**:
  - Fully Convolutional Network
  - Designed and ready for **real-time** systems.
  - Divides an Image into a **grid** of small **patches**.
    - performs classification and object detection for each patch of the grid.
  - **anchors** are pre-defined bounding boxes of different sizes and aspect ratios that are placed at different spatial locations over an input image. These anchors serve as reference points for detecting objects of different sizes and shapes.


Input image
3 x H x W
Divide image into grid
7 x 7
Image a set of base boxes centered at each grid cell
Here B = 3
Within each grid cell:
- Regress from each of the B base boxes to a final box with 5 numbers: (dx, dy, dh, dw, confidence)
- Predict scores for each of C classes (including background as a class)
- Looks a lot like RPN, but category-specific!
Output:
7 x 7 x (5 * B + C)

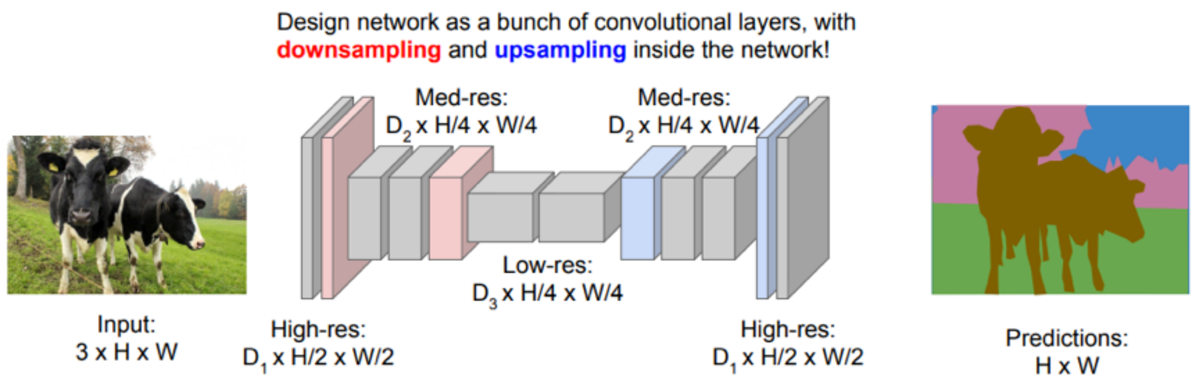


  - **Non-maximum suppression [NMS]** is a post-processing technique used in object detection to **eliminate redundant bounding boxes** that may have been produced by the object detection algorithm. NMS works by comparing the overlapping regions of bounding boxes and discarding those that have a high degree of overlap with other boxes and a lower confidence score.

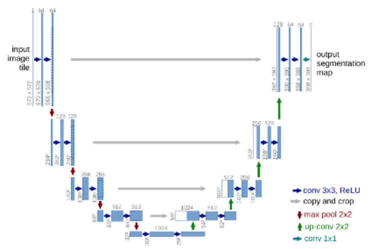
- **Segmentation**



- **Semantic Segmentation** is a computer vision task where each pixel in an image is assigned a class label, which represents the semantic meaning of that pixel. The output of semantic segmentation is a segmented image where pixels of the same class have the same colour or label. The main objective of semantic segmentation is to recognize and differentiate objects and their boundaries within an image.
  - To perform semantic segmentation, various techniques can be used, such as fully convolutional networks (**FCNs**) that use a combination of **down-sampling and up-sampling** techniques to generate pixel-level predictions
  - **skip connections** involve connecting layers in a neural network that are not necessarily adjacent to one another. This allows for the preservation of fine-grained spatial information throughout the network. In the context of semantic segmentation, skip connections can be used to connect earlier layers in the network to later ones, allowing the network to use information from earlier stages in the segmentation process to refine segmentation predictions in later stages.
  - **upsampling** is a technique used to increase the resolution of feature maps in a neural network. This is typically accomplished by applying a transformation to the feature map that increases the spatial dimensions of that map (by duplicating feature values or using interpolation).

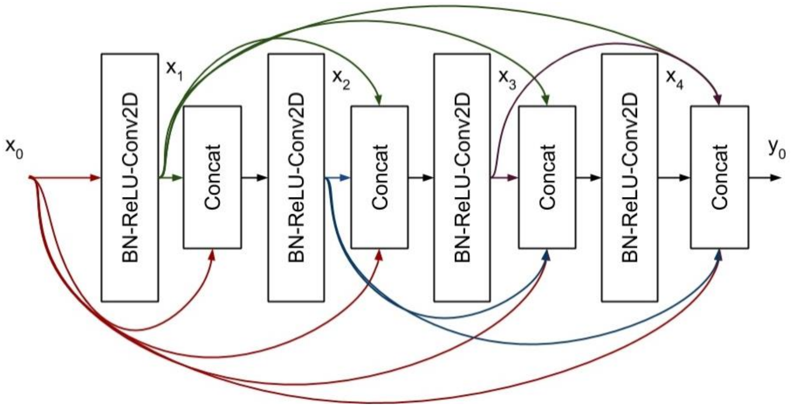

Design network as a bunch of convolutional layers, with downsampling and upsampling inside the network!
Input: 3 x H x W
High-res: D_1 x H/2 x W/2
Med-res: D_2 x H/4 x W/4
Low-res: D_3 x H/4 x W/4
Med-res: D_2 x H/4 x W/4
High-res: D_1 x H/2 x W/2
Predictions: H x W

  - **Learnable upsampling** is a technique used in deep learning that involves using trainable parameters to upsample feature maps instead of predefined upsampling methods. It allows the network to learn the best upsampling method for the given task.
  - upsampling is often used in conjunction with skip connections in semantic segmentation networks to combine high-level features learned by later layers with low-level features learned by earlier layers, resulting in more accurate and detailed segmentation predictions.
  - **U-Net** is a CNN architecture that uses skip connections to improve segmentation accuracy.

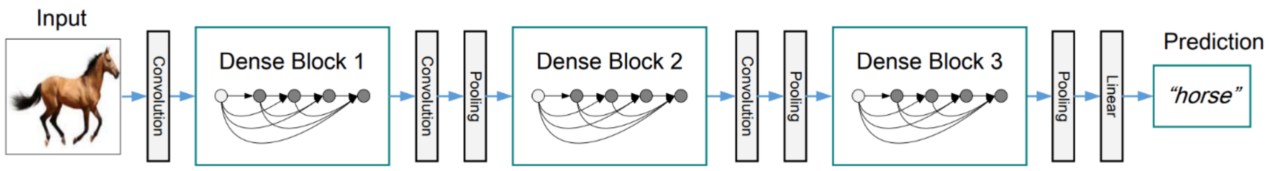


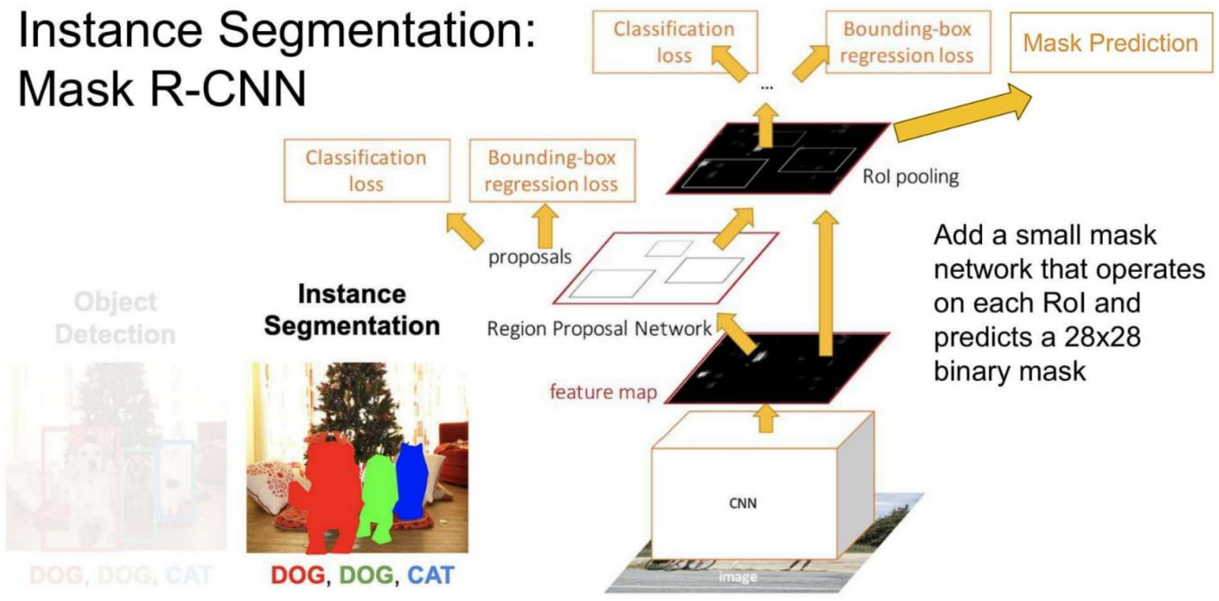
DenseNet [Densely connected convolutional network]:

- aims to address the **vanishing gradient** problem and improve feature reuse in deep networks.
- The main idea behind DenseNet is to **connect** each layer to **every** other layer in a feed-forward fashion, creating densely connected blocks of layers.



- DenseNet **concatenates** the output feature maps of each layer to the input of **all** subsequent layers. This allows the network to reuse features more efficiently and preserve gradient flow.
- used for both classification and semantic segmentation.





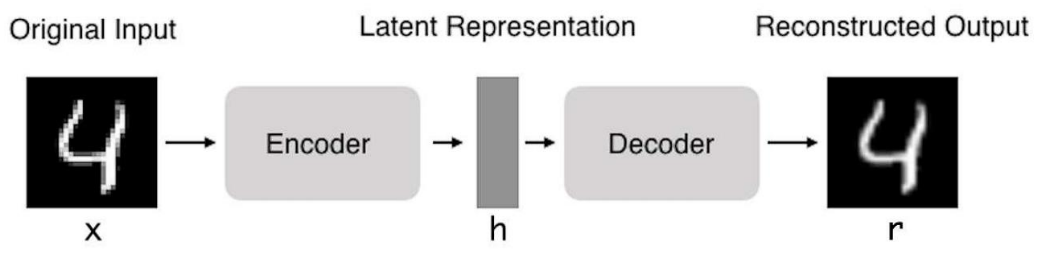
- **Mask R-CNN** is an instance segmentation framework that extends the Faster R-CNN object detection model by adding a branch for predicting segmentation masks on each Region of Interest (RoI) in parallel with the existing branch for classification and bounding box regression. This allows for pixel-level segmentation of objects in an image, providing both the object category and its precise location in the image.

Unsupervised Learning:

Unsupervised (or Self-Supervised) learning is a type of machine learning where the model is trained on **unlabeled data** without any specific output or desired result. The goal is to identify patterns or relationships within the data through techniques such as: clustering or dimensionality reduction, without any prior knowledge or labels. This approach is useful when there is a lack of labeled data, or when the patterns within the data are not well defined. Unsupervised learning can be applied in various domains, including image and speech recognition, anomaly detection, and data compression.

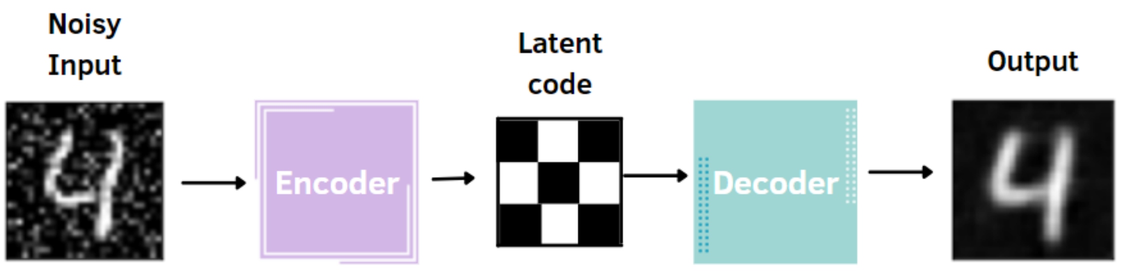
- There is a limit to the power of **discriminative models** (supervised learning).
- And so – **Generative Models** were developed.
- Generative Models can generate new data instances.

  - **Autoencoders** are designed to reproduce their input (especially for images)
    - Key point is to reproduce the input from a learned encoding.
    - Problem: loss functions are not well-suited for complex or high-dimensional data.



    - Autoencoders can be used for **pre-training** by first training an autoencoder on a large unlabeled dataset, and then **fine-tuning** the **encoder** on a smaller labeled dataset for a specific task, such as classification or segmentation.
      This allows the model to learn useful features from the larger **unlabeled** data, which can then be transferred to use on the smaller dataset in order to improve performance.

    - **de-noising**: autoencoders can be used to clean data (where small details are not important).


Noisy Input
Latent code
Output
Encoder
Decoder