SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Deep Learning for Computer Vision

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Chapter Goals

After completing this chapter, you should be able to understand :

- Convolutional Neural Networks
- CNNs for Image Classification
- CNNs with Keras
- Known CNN architectures
- Vision Transformers
- ViT for Image classification
- ViT with Pytorch

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Convolutional Neural Networks

Classical Computer vision

Subject matter expertise → Feature definition → Feature extraction → Image classification

Intraclass Variation

Illumination Changes

Occlusion

Fine-Grained Categories

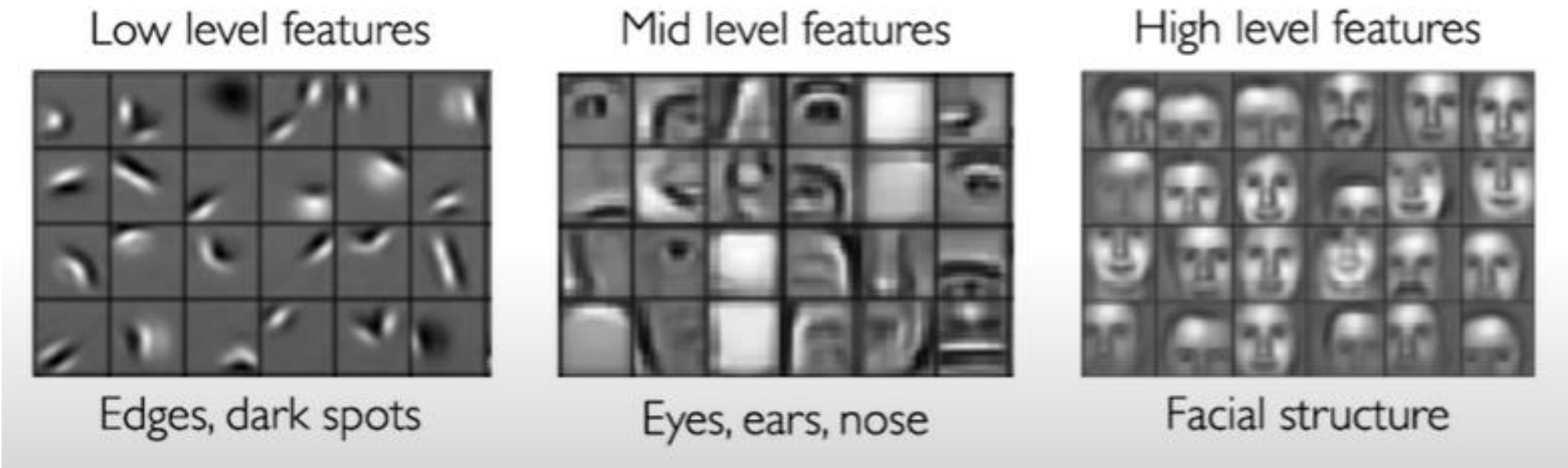
Maine Coon | Ragdoll | American Shorthair

Background Clutter

Deformation

Convolutional Neural Networks: Learning Feature Representations

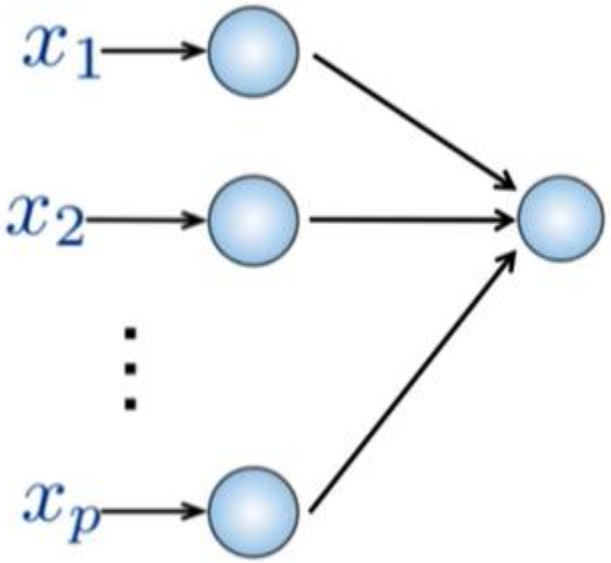
CNNs allow for learning a hierarchy of features directly from data



Dense (Fully Connected) Neural Network

Input:

- 2D image
- Vector of pixel values



Fully Connected:

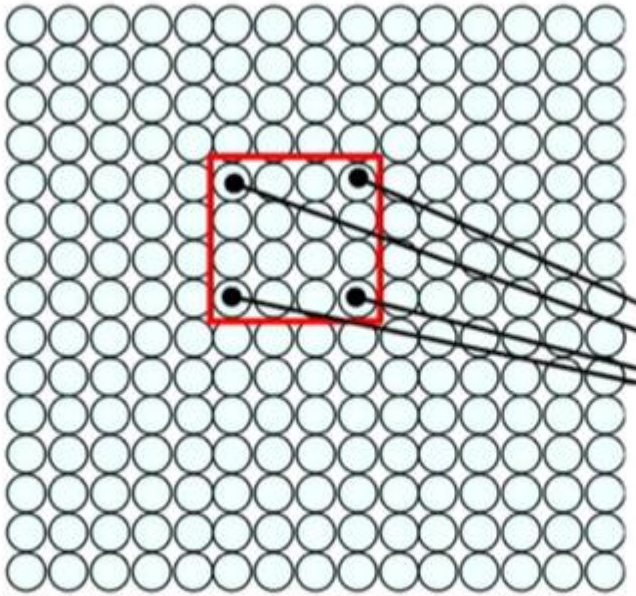
- Connect neuron in hidden layer to all neurons in input layer
- No spatial information!
- And many, many parameters!

How can we use **spatial structure** in the input to inform the architecture of the network?

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Using Spatial Structure

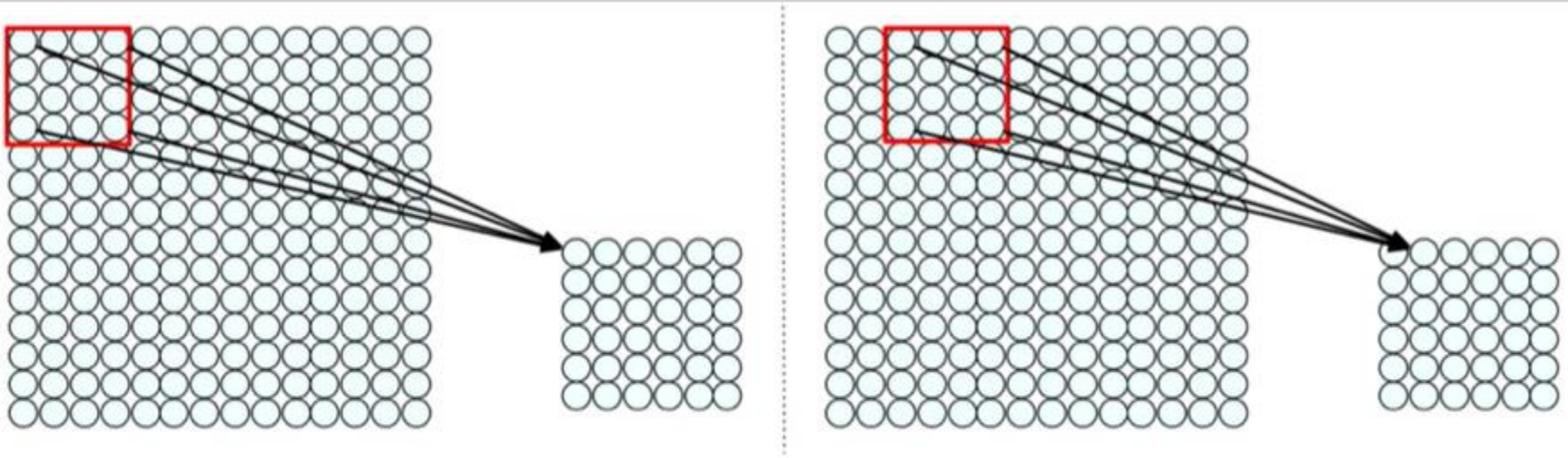
Input: 2D image.
Array of pixel values



Idea: connect patches of input to neurons in hidden layer.

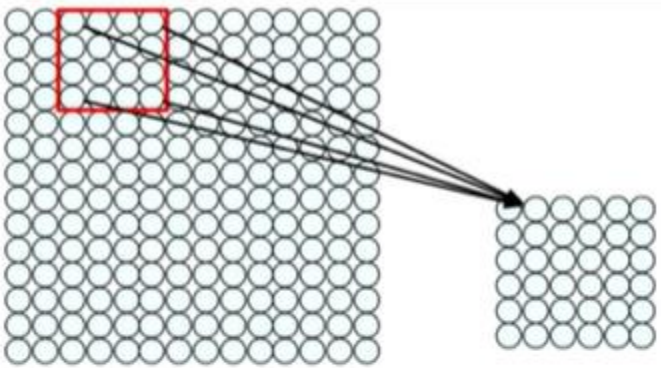
Neuron connected to region of input. Only "sees" these values.

Using Spatial Structure



Connect patch in input layer to a single neuron in subsequent layer.
Use a sliding window to define connections.
How can we **weight** *the patch to detect particular features?*

Feature Extraction with Convolution



- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

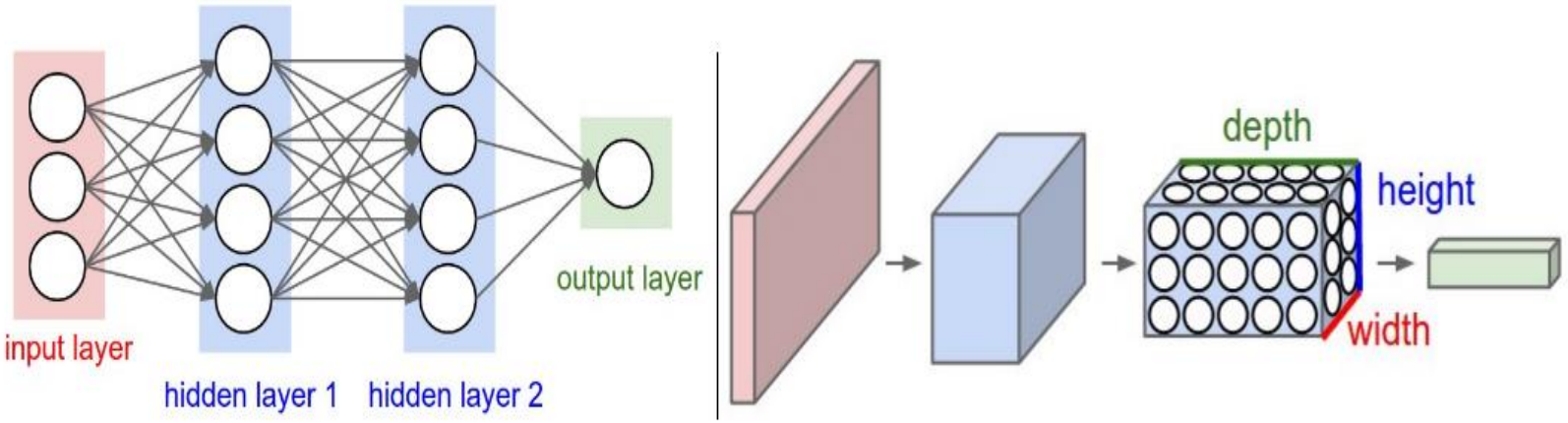
This "patchy" operation is **convolution**

1) Apply a set of weights – a filter – to extract **local features**

2) Use **multiple filters** to extract different features

3) **Spatially share** parameters of each filter

Dense Neural Network and Convolutional Neural Network

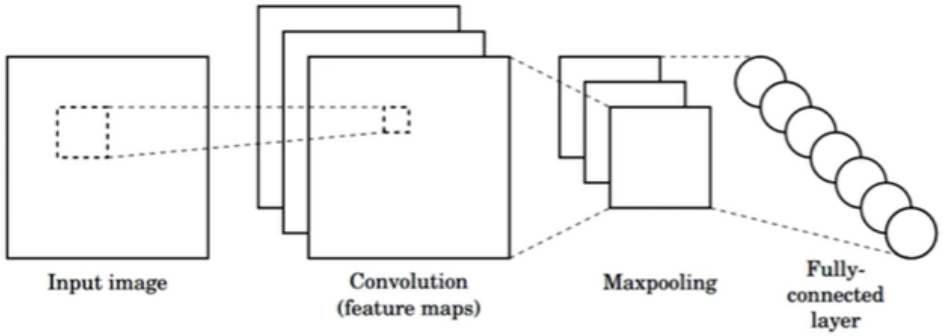


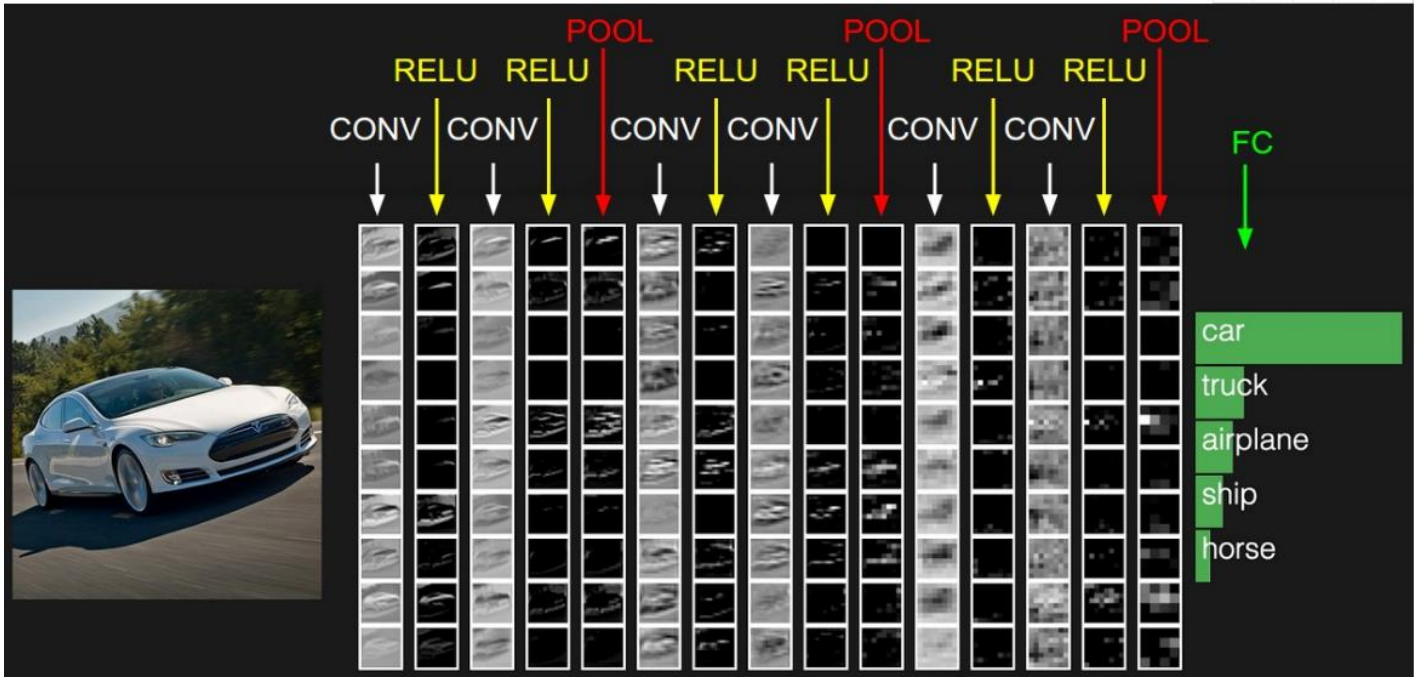
Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Dense Neural Network limitation for computer vision:

- Connect neuron in hidden layer to all neurons in input layer
- No spatial information
- Many parameters

A simple CNN structure





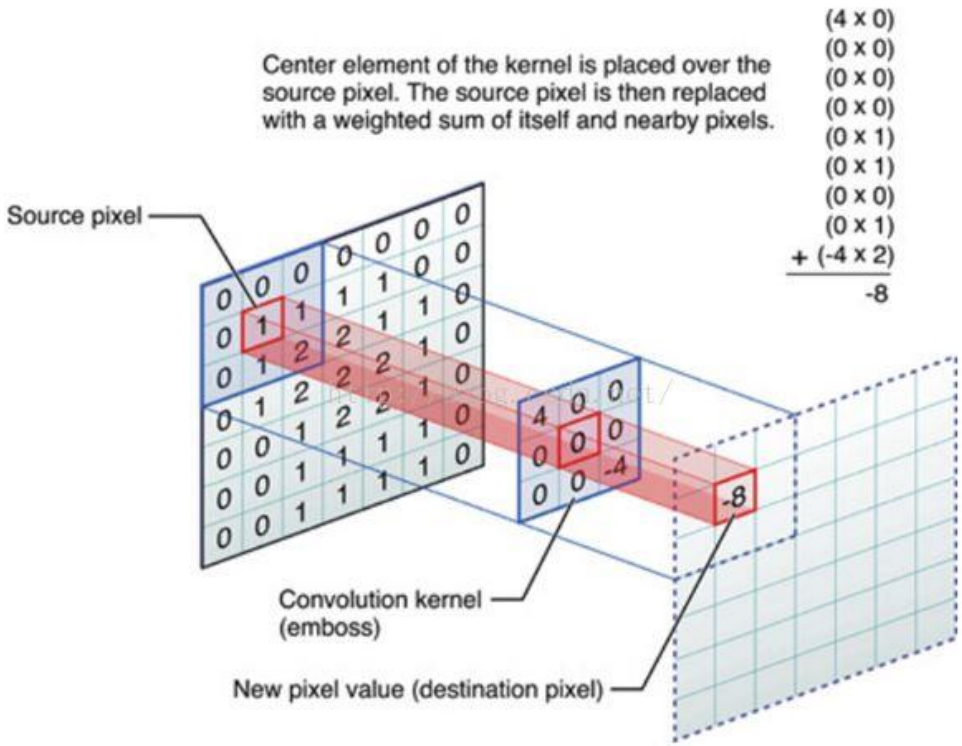
*CONV: Convolutional layer *RELU: Activation layer *POOL: Pooling or Dimension reduction layer *FC: Fully connected layer

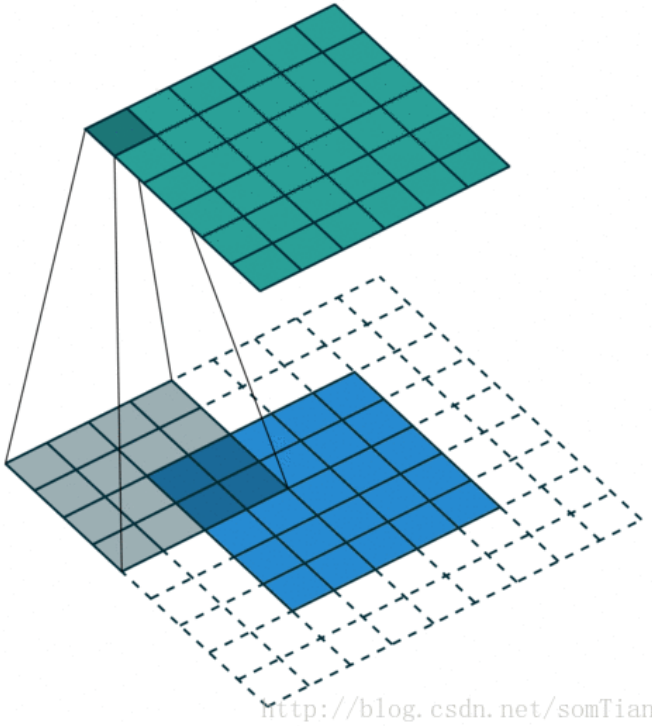
CNNs: Feature Extraction+Classification

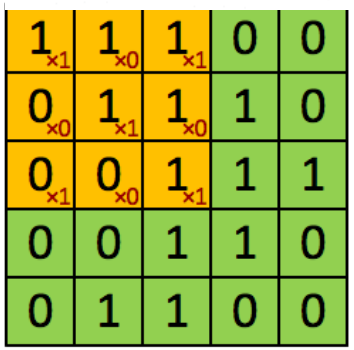
CNN is a combination of two components:

- Feature extraction: The convolution + pooling layers perform feature extraction and make possible to detect features.
- Classification: The fully connected layers act as a classifier **on top** of the extracted features, and assign a probability for the input image being a representative of certain class

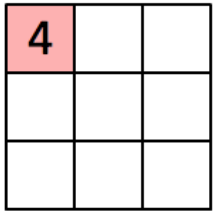
Convolutional layer







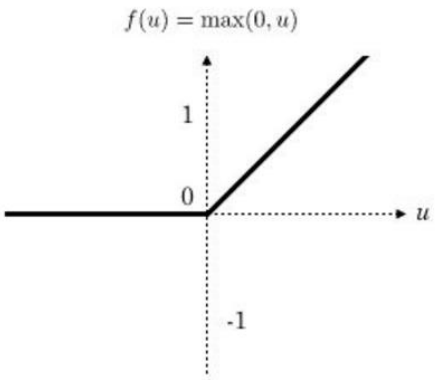
Image



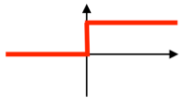
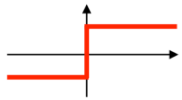
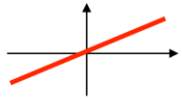
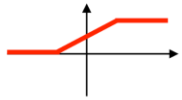
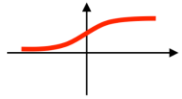
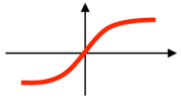
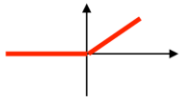
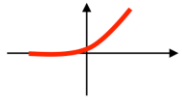
Convolved Feature

Padding on the input volume with zeros in such way that the convolution layer does not alter the spatial dimensions of the input

Activation layer: Rectified linear unit, ReLU

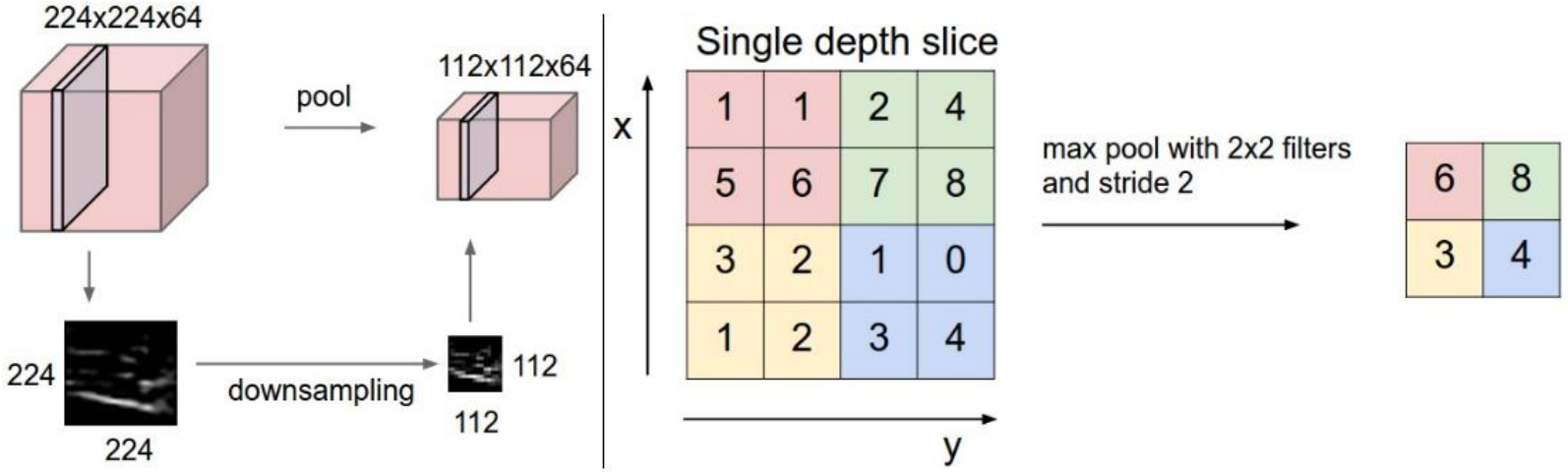


RELU

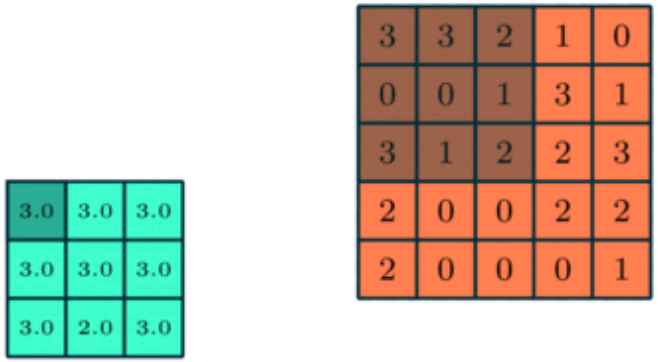
Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

Other activation functions

Pooling layer

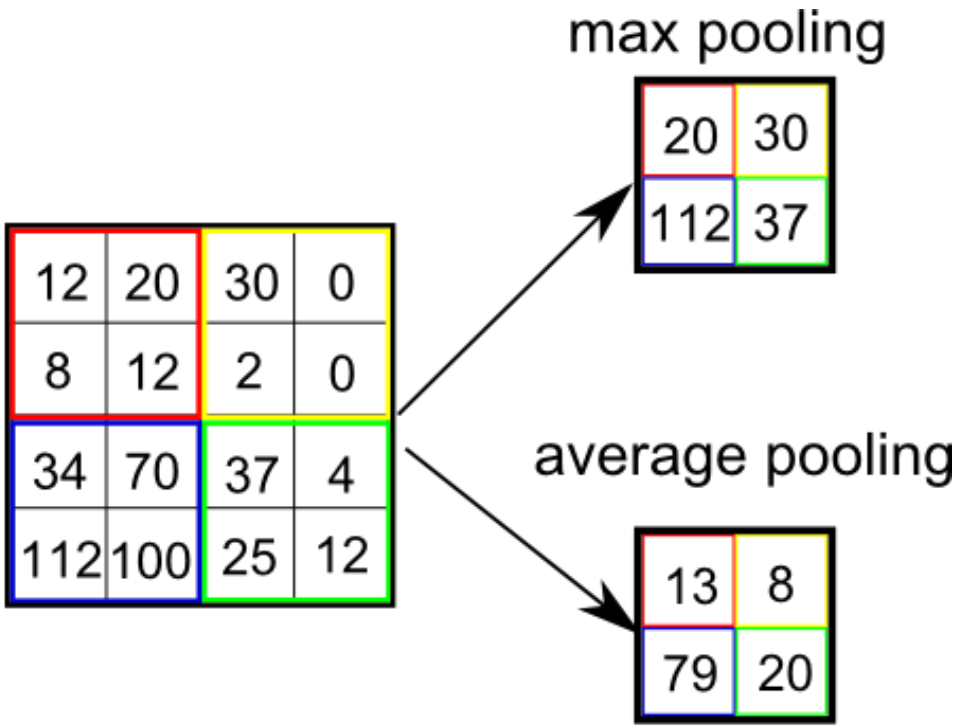


Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).



Pooling

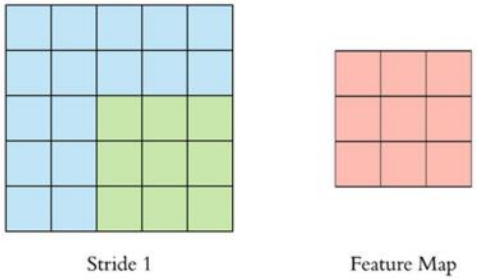
- Max pooling selects the brighter pixels from the image. It is useful when the background of the image is dark, and we are interested in only the lighter pixels of the image
- Average pooling method smooths out the image and hence the sharp features may not be identified when this pooling method is used.

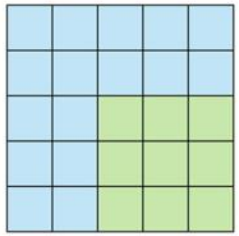


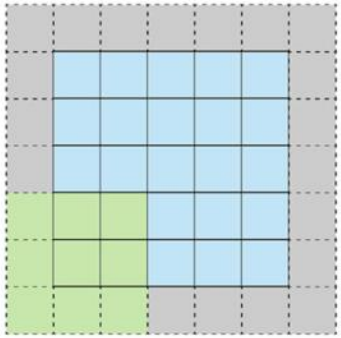
Convolution layer: Further properties

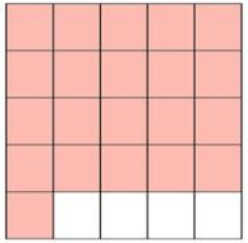
Stride specifies how much we move the convolution filter at each step.

- We can have bigger strides if we want less overlap between the fields making the resulting feature map **smaller**.
- We can use **padding** to maintain the feature map's dimensionality similar to the input by surrounding input with zeros or the values on the edge



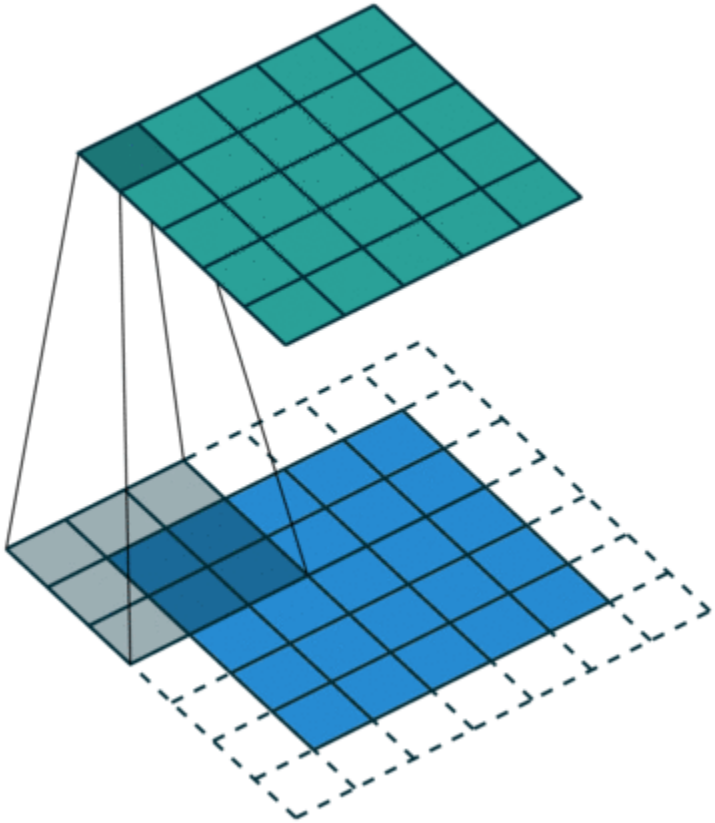






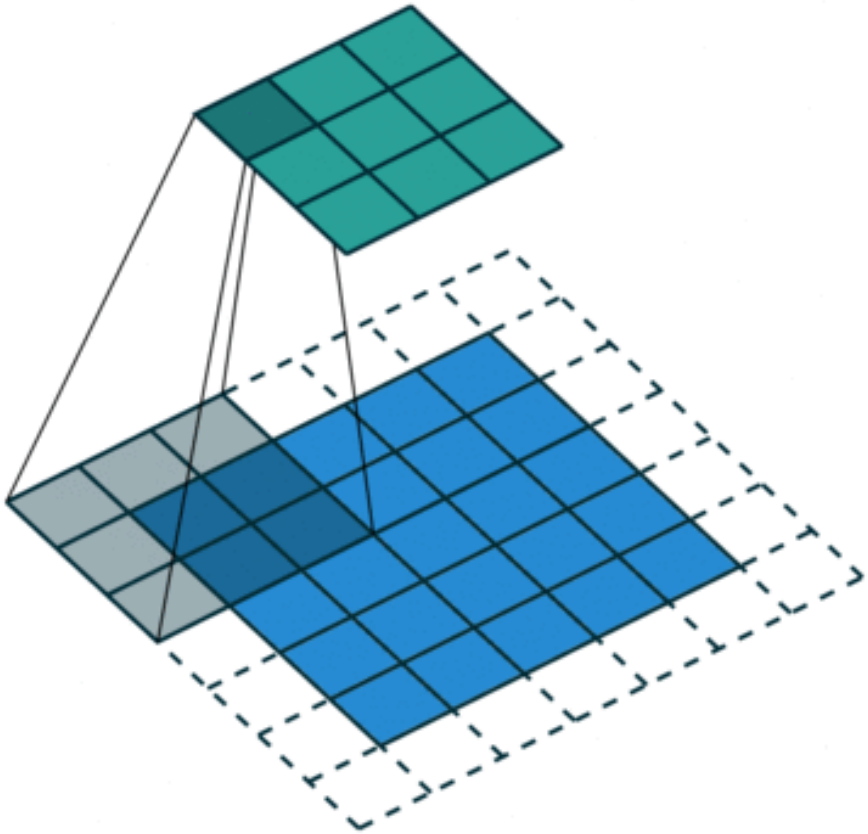
Rule: output size [(W – K + 2P) / S]+ 1
Where S:stride P:padding

Convolution layer: Padding

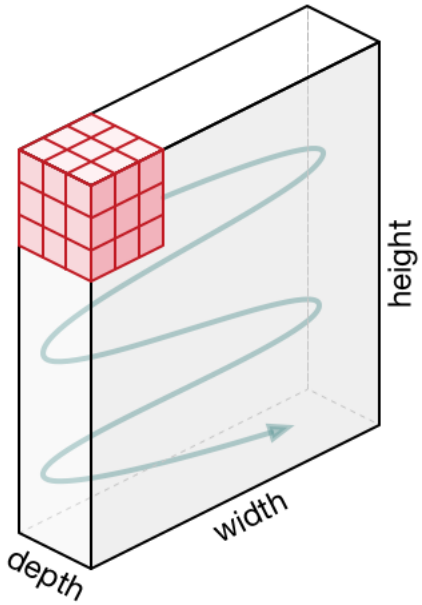


Padding: 5x5x1 image is padded with 0s to create a 6x6x1 image
The convolved matrix is of dimensions 5x5x1

Convolution layer: Stride



Convolution Operation with Stride Length = 2



The filter moves to the right with a certain Stride Value till it parses the complete width. Moving on, it hops down to the beginning (left) of the image with the same Stride Value and repeats the process until the entire image is traversed.

CNN: Training

CNN is trained the same way as ordinary NN using **Backpropagation** with gradient descent to adjust its filter values (weights):

- **forward pass:** All of the weights/filter values were randomly initialized, and the output will not give preference to any classification class in particular
- **loss function:** having a training data with both an image and a label, we will use labels to compute a loss against the output of previous step (forward pass). One of the common ways to define a loss function we want to minimize is **cross entropy** between target and actual output:
- **backward pass:** solving an optimization problem, we find out which weights most directly contributed to the loss of the network.

We have to calculate a derivative of the loss dL/dW (where W are the weights at a particular layer) to determining which weights contributed most to the loss and find ways to adjust them so that the loss decreases.

- **weight update:** update all the weights of the filters so that they change in the opposite direction of the gradient. Where the learning rate is a parameter chosen by the Data scientist to define a step in weight change

$$w = w_i - \eta \frac{dL}{dW}$$

w = Weight
w_i = Initial Weight
η = Learning Rate

CNN: Cross entropy

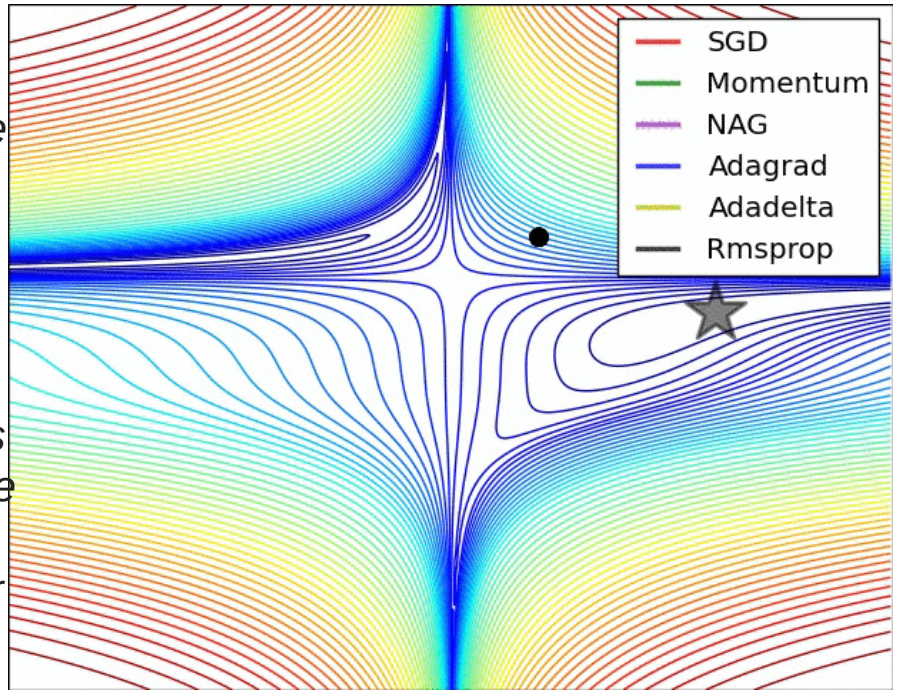
Consider two probability distributions P and Q where
P is Expected Probability (*y*): The known probability of each class label for an example in the dataset
Q is Predicted Probability (*yhat*): The probability of each class label an example predicted by the model

Cross-entropy can be calculated using the probabilities of the events from P and Q, as follows:
H(P, Q) = – sum x in X P(x) * log(Q(x))
Where P(x) is the probability of the event x in P, Q(x) is the probability of event x in Q and log is the base-2 logarithm,

CNN training: Optimization methods

Gradient Descent: Gradient descent is an optimization algorithm used to minimize the cost function by iteratively moving in the direction of steepest descent. It calculates the gradient of the cost function with respect to the model parameters and adjusts the parameters in the opposite direction of the gradient.

Stochastic Gradient Descent (SGD): SGD is an optimization algorithm that randomly selects a small subset of training examples (also known as a mini-batch) to calculate the gradient and update the model parameters. This method is computationally efficient and can converge faster than batch gradient descent when dealing with large datasets.



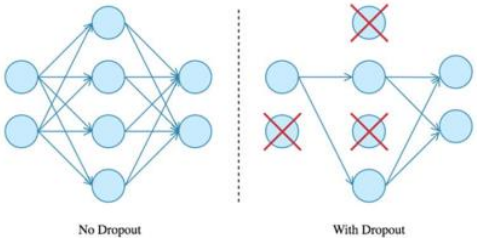
Momentum: Momentum is a modification of SGD that adds a "momentum term" to the gradient update rule. It helps the optimizer to continue moving in the previous direction and smooth out the oscillations in the gradient updates. This can lead to faster convergence and better results.

Adaptive methods: Adaptive methods such as Adagrad, RMSprop, and Adam adjust the learning rate for each parameter based on its previous gradients. This allows them to adaptively change the learning rate for each parameter and improve the convergence of the optimization algorithm.

CNN-overfitting

We may face a case, when training loss keeps going down, but the validation loss starts increasing after certain epoch. It is a sign of ***overfitting***. It tells us that our model is memorizing the training data, but it's failing to generalize to new instances.

- The most popular regularization technique for deep neural networks is **Dropout**.
- It is used to prevent overfitting via temporarily "dropping"/disabling a neuron with probability *p* (a hyperparameter called the *dropout rate* that is typically a number around 0.5) at each iteration during the training phase.
- The dropped-out neurons are resampled with probability *p* at every training step (a dropped out neuron at one step can be active at the next one). (Dropout is not applied during test time after the network is trained).
- Dropout can be applied to input or hidden layer nodes but not the output nodes



Reason. Dropout forces every neuron to be able to operate independently and prevents the network to be too dependent on a small number of neurons.

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Batch Normalization

Batch Normalization We normalize not only the input layer, but the values in the hidden layers as well, improving training speed and overall accuracy.
Batch normalization allows each layer of a network to learn by itself a little bit more independently of other layers.

- **Speeds up training:** it should converge much quicker, even though each training iteration will be slower because of the extra normalization calculations during the forward pass and the additional hyperparameters to train during back propagation.
- **Allows higher learning rates:** as networks get deeper, initially small gradients get even smaller during back propagation, and so require even more iterations. Batch normalization allows much higher learning rates, increasing the speed at which networks train.
- **Simplifies weights initialization:** batch normalization helps reduce the sensitivity to the initial starting weights.

Batch Normalization

Batch normalization adds two trainable parameters to each layer, so the normalized output is multiplied by a "standard deviation" parameter and add a "mean" parameter.
Being used before the activation function and dropout, Batch Normalization layer applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

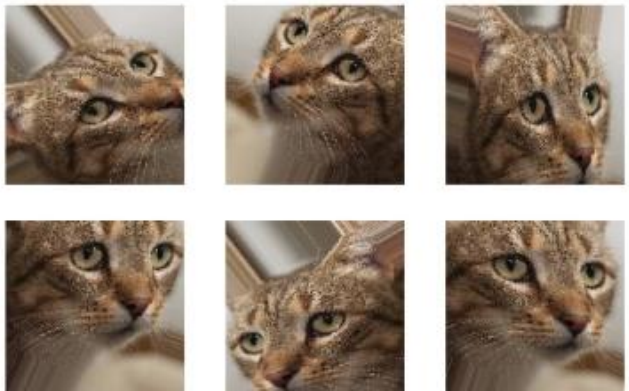
- **Makes activation functions viable:** as batch normalization regulates the values going into each activation function, nonlinearities that don't work well in deep networks tend to become viable again.

-**Reduces overfitting:** it has a slight regularization effects and, similarly to dropout, it adds some noise to each hidden layer's activations. If we use batch normalization, we will use less dropout, which is a good thing because we are not going to lose a lot of information. However, we should not depend only on batch normalization for regularization; we should better use it together with dropout.

```
keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True, beta_initializer='zeros', gamma_initializer='ones',
    moving_mean_initializer='zeros', moving_variance_initializer='ones', beta_regularizer=None, gamma_regularizer=None, beta_constraint=None, gamma_constraint=None)
```

CNN-Data Augmentation

- **Overfitting** happens when we are training on very few examples (e.g. 1000 images per category), and even use of dropout does not help us much.
- To start thinking about Deep Learning, we have to operate with at least 100K training examples. On a small dataset we will overfit no matter which regularization technique is applied.
- Fortunately, there is a **data augmentation** method which enables us to train deep models on small datasets. It allows us to artificially boost the size of the training set through enrichment of the training data by generating new examples via random transformation of existing ones.
- Data augmentation is done dynamically during training time. To generate realistic images, the common transformations are: rotation, shifting, resizing, exposure adjustment, contrast change etc. Also, some data cleaning tricks on images can be applied (e.g. image normalization).



CNNs-Keras: Data preparation

Prepare the Data

Before we can train the model, we need to prepare the data. We'll divide the feature values by 255 to normalize them as floating point values between 0 and 1, and we'll split the data so that we can use 70% of it to train the model, and hold back 30% to validate it. When loading the data, the data generator will assign "hot-encoded" numeric labels to indicate which class each image belongs to based on the subfolders in which the data is stored.

In this case, there are three subfolders - *circle*, *square*, and *triangle*, so the labels will consist of three *0* or *1* values indicating which of these classes is associated with the image - for example the label [0 1 0] indicates that the image belongs to the second class (*square*).

```
from keras.preprocessing.image import ImageDataGenerator

batch_size = 30

print("Getting Data...")
datagen = ImageDataGenerator(rescale=1./255, # normalize pixel values
                             validation_split=0.3) # hold back 30% of the images for validation

print("Preparing training dataset...")
train_generator = datagen.flow_from_directory(
    training_folder_name,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='categorical',
    subset='training') # set as training data

print("Preparing validation dataset...")
validation_generator = datagen.flow_from_directory(
    training_folder_name,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='categorical',
    subset='validation') # set as validation data
```

```
Getting Data...
Preparing training dataset...
Found 840 images belonging to 3 classes.
Preparing validation dataset...
Found 360 images belonging to 3 classes.
```

CNNs-Keras:NN Definition

Define the CNN

Now we're ready to train our model. This involves defining the layers for our CNN, specifying an *optimizer*, and compiling the model for multi-class classification. In this example, we'll use an optimizer based on the *Adam* algorithm and set its *learning rate* parameter (which determines how much the weights are adjusted after backpropagation identifies their affect on loss). These settings can have a significant impact on how well your model (and how quickly) your model learns the optimal weights and bias values required to predict accurately.

Note: For information about the optimizers available in Keras, see https://keras.io/optimizers/

```
# Define a CNN classifier network
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Flatten, Dense
from keras import optimizers

# Define the model as a sequence of layers
model = Sequential()

# The input layer accepts an image and applies a convolution that uses 32 6x6 filters and a rectified linear unit activation function
model.add(Conv2D(32, (6, 6), input_shape=train_generator.image_shape, activation='relu'))

# Next we'll add a max pooling layer with a 2x2 patch
model.add(MaxPooling2D(pool_size=(2,2)))

# We can add as many layers as we think necessary - here we'll add another convolution layer and another and max poolinglayer
model.add(Conv2D(32, (6, 6), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Now we'll flatten the feature maps and generate an output layer with a predicted probability for each class
model.add(Flatten())
model.add(Dense(train_generator.num_classes, activation='softmax'))

# We'll use the ADAM optimizer
opt = optimizers.Adam(lr=0.001)

# With the layers defined, we can now compile the model for categorical (multi-class) classification
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

print(model.summary())
```

CNNs-Keras: Model summary

model summary shows the output shape from each layer:

- In the input layer, we pass an image, which is 128x128 pixels x 3 channels.
- A convolution layer then applies 32 6x6 filters. The 6x6 filter loses 5 pixels around the edge of the image, so the output is 123x123x32.
- A pooling layer then applies a 2x2 filter, halving the size of each pixel layer; which when rounded gives us an output of 61x61x32.
- Then another convolution layer uses 32 6x6 filters, losing 5 pixels to produce an output of 56x56x32.
- Another pooling layer halves the pixel dimensions, so now we have 28x28x32.
- When we flatten this, 28x28x32 gives us 25,088 individual values
- Finally, we feed these into a dense layer that produces 3 outputs - a probability value for each of our three classes.

Note that the flattened layer that defines the input to our fully-connected neural network always expects 25,088 values; so working backward from here, our convolutional layers must start with a 128x128 image. Images of a different size or shape will not work with this model.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 123, 123, 32)      3488
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 61, 61, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 56, 56, 32)        36896
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 75267
=================================================================
Total params: 115,651
Trainable params: 115,651
Non-trainable params: 0
_________________________________________________________________
None
```

CNNs-Keras: Model training

Train the Model

With the layers of the CNN defined, we're ready to train the model using our image data. In the example below, we use 5 iterations (*epochs*) to train the model in 30-image batches, holding back 30% of the data for validation. After each epoch, the loss function measures the error (*loss*) in the model and adjusts the weights (which were randomly generated for the first iteration) to try to improve accuracy.

Note: We're only using 5 epochs to reduce the training time for this simple example. A real-world CNN is usually trained over more epochs than this. CNN model training is processor-intensive, so it's recommended to perform this on a system that can leverage GPUs (such as the Data Science Virtual Machine in Azure) to reduce training time.

Status will be displayed as the training progresses.

```
[ ]: # Train the model over 5 epochs
num_epochs = 5
history = model.fit_generator(
    train_generator,
    steps_per_epoch = train_generator.samples // batch_size,
    validation_data = validation_generator,
    validation_steps = validation_generator.samples // batch_size,
    epochs = num_epochs)
```

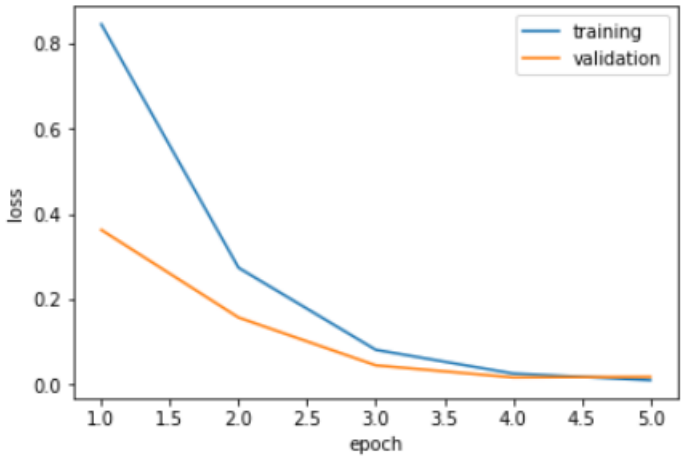
```
28/28 [==============================] - 198s 7s/step - loss: 0.8453 - accuracy: 0.6107 - val_loss: 0.3629 - val_accuracy: 0.8222
Epoch 2/5
28/28 [==============================] - 167s 6s/step - loss: 0.0815 - accuracy: 0.9869 - val_loss: 0.0451 - val_accuracy: 1.0000
Epoch 4/5
28/28 [==============================] - 184s 7s/step - loss: 0.0259 - accuracy: 1.0000 - val_loss: 0.0168 - val_accuracy: 1.0000
Epoch 5/5
28/28 [==============================] - 186s 7s/step - loss: 0.0102 - accuracy: 1.0000 - val_loss: 0.0180 - val_accuracy: 1.0000
```

CNNs-Keras: Training/validation loss

View the Loss History

We track average training and validation loss history for each epoch. We can plot these to verify that loss reduced as the model was trained, and to detect *over-fitting* (which is indicated by a continued drop in training loss after validation loss has levelled out or started to increase.

```
%matplotlib inline
from matplotlib import pyplot as plt

epoch_nums = range(1,num_epochs+1)
training_loss = history.history["loss"]
validation_loss = history.history["val_loss"]
plt.plot(epoch_nums, training_loss)
plt.plot(epoch_nums, validation_loss)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(['training', 'validation'], loc='upper right')
plt.show()
```



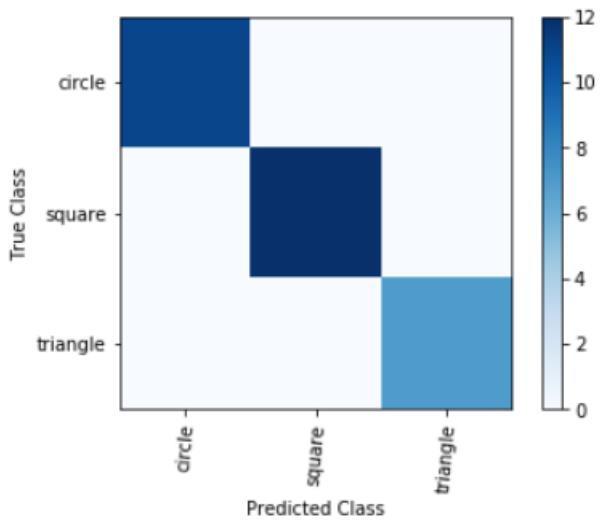
CNNs-Keras: Model evaluation

Evaluate the model

With our model trained, we'll use it to predict labels for the test data and evaluate its precision, recall, and simple accuracy using the known labels. Then we'll plot the confusion matrix to evaluate how well the model performs for each class label. Keras does not provide a built-in confusion matrix, so we'll use Scikit-Learn.

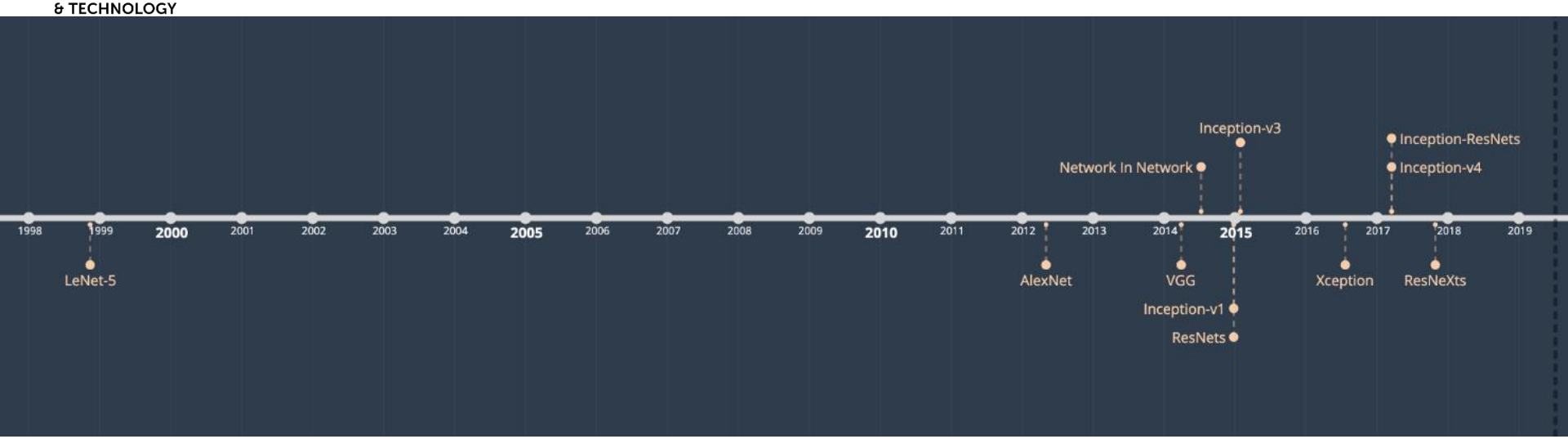
```
import numpy as np
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
%matplotlib inline

print("Generating predictions from validation data...")
# Get the image and label arrays for the first batch of validation data
x_test = validation_generator[0][0]
y_test = validation_generator[0][1]

# Use the moedl to predict the class
class_probabilities = model.predict(x_test)

# The model returns a probability value for each class
# The one with the highest probability is the predicted class
predictions = np.argmax(class_probabilities, axis=1)

# The actual labels are hot encoded (e.g. [0 1 0], so get the one with the value 1
true_labels = np.argmax(y_test, axis=1)

# Plot the confusion matrix
cm = confusion_matrix(true_labels, predictions)
plt.imshow(cm, interpolation="nearest", cmap=plt.cm.Blues)
plt.colorbar()
tick_marks = np.arange(len(classes))
plt.xticks(tick_marks, classes, rotation=85)
plt.yticks(tick_marks, classes)
plt.xlabel("Predicted Class")
plt.ylabel("True Class")
plt.show()
```

```
Generating predictions from validation data...
```



SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Example CNNs

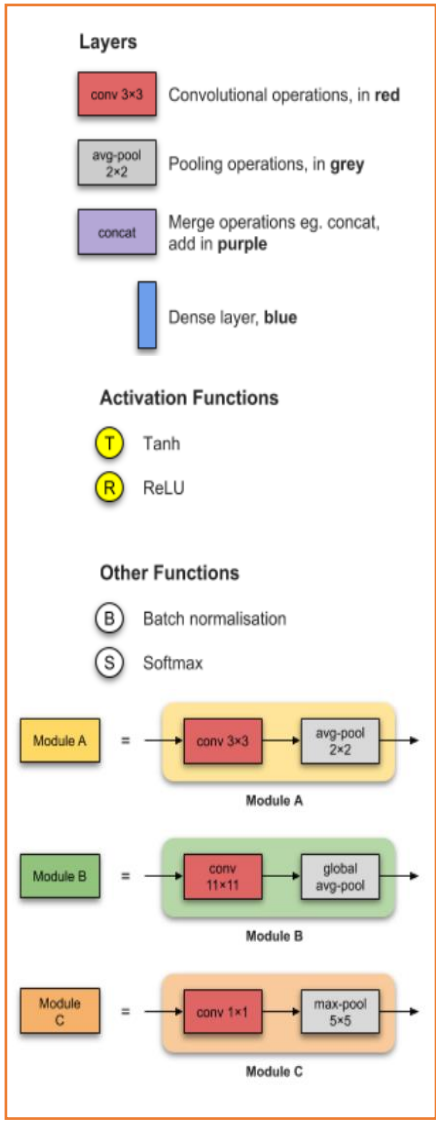


Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
VGG16	528 MB	0.713	0.901	138,357,544	23
InceptionV3	92 MB	0.779	0.937	23,851,784	159
ResNet50	98 MB	0.749	0.921	25,636,712	-
Xception	88 MB	0.790	0.945	22,910,480	126
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
ResNeXt50	96 MB	0.777	0.938	25,097,128	-

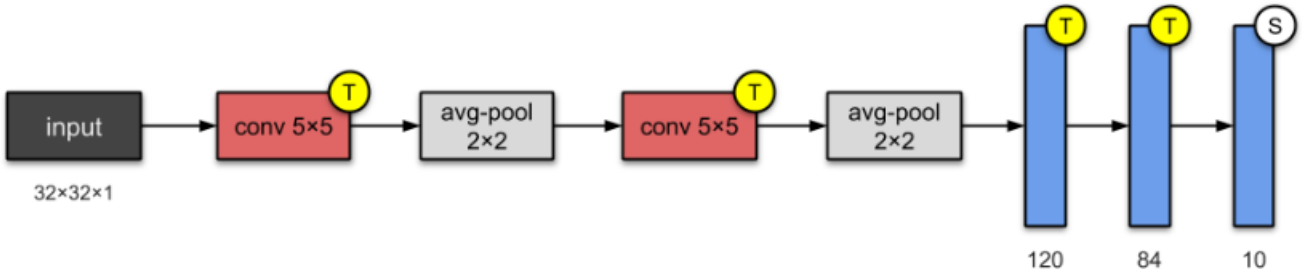
The top-1 and top-5 accuracy refers to the model's performance on the ImageNet validation dataset.

Depth refers to the topological depth of the network. This includes activation layers, batch normalization layers etc.

Example CNNs

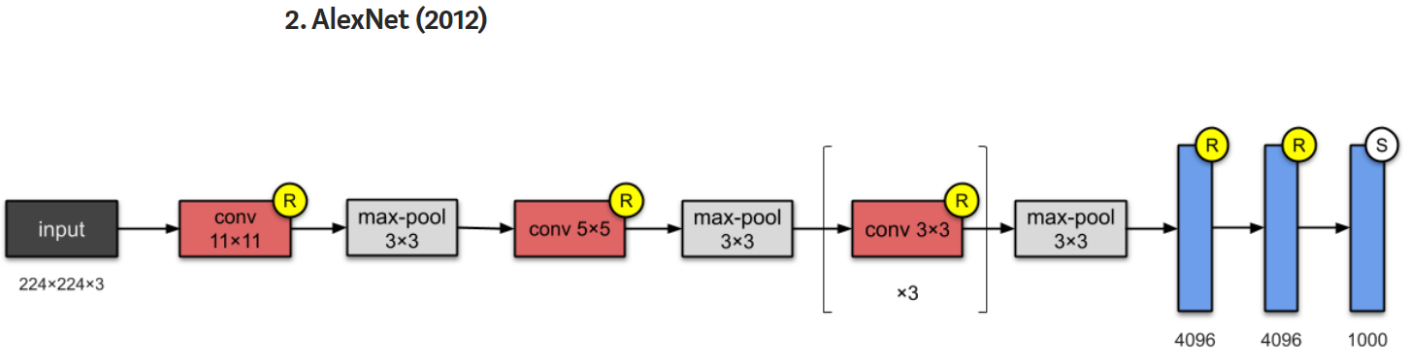


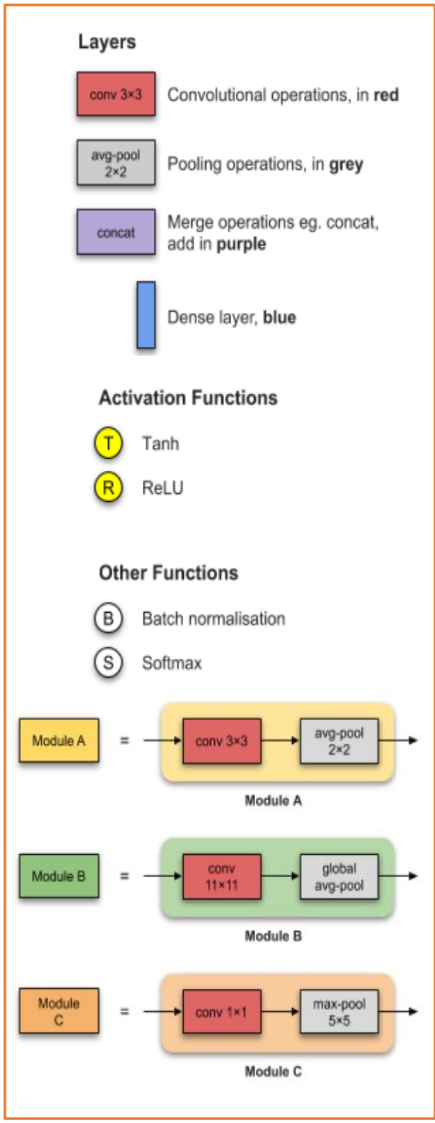
1. LeNet-5 (1998)



LeNet was introduced in the research paper "Gradient-Based Learning Applied To Document Recognition" in the year 1998 by Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner.

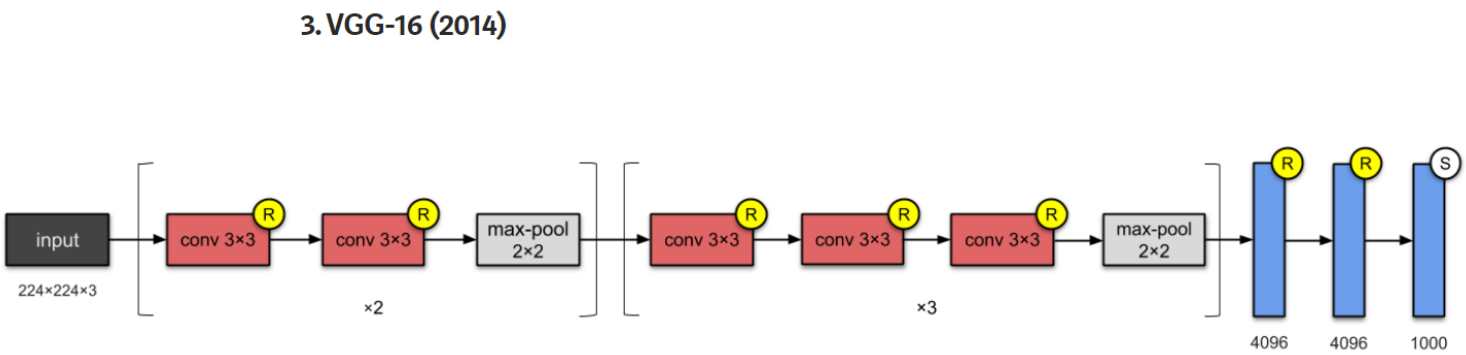
Example CNNs

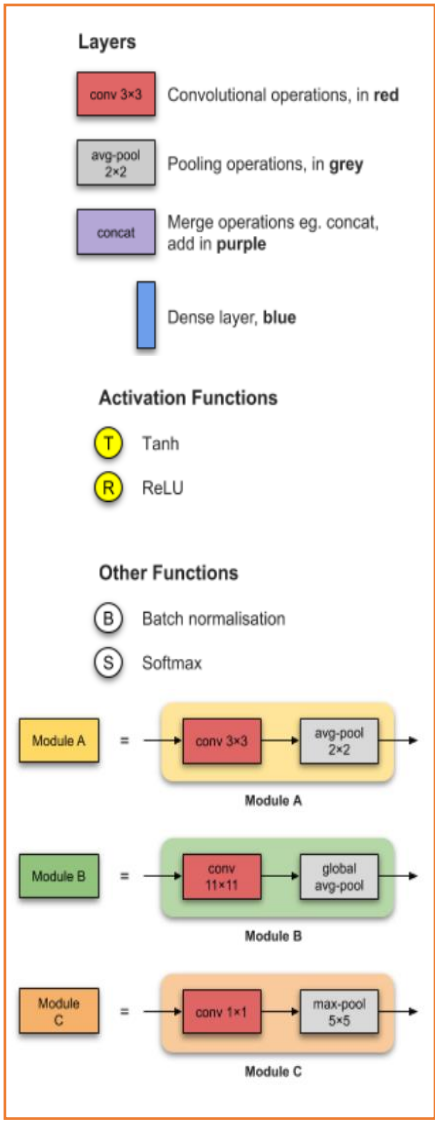




AlexNet contained eight layers; the first five were convolutional layers, some of them followed by max-pooling layers, and the last three were fully connected layers. It used the non-saturating ReLU activation function, which showed improved training performance over tanh and sigmoid

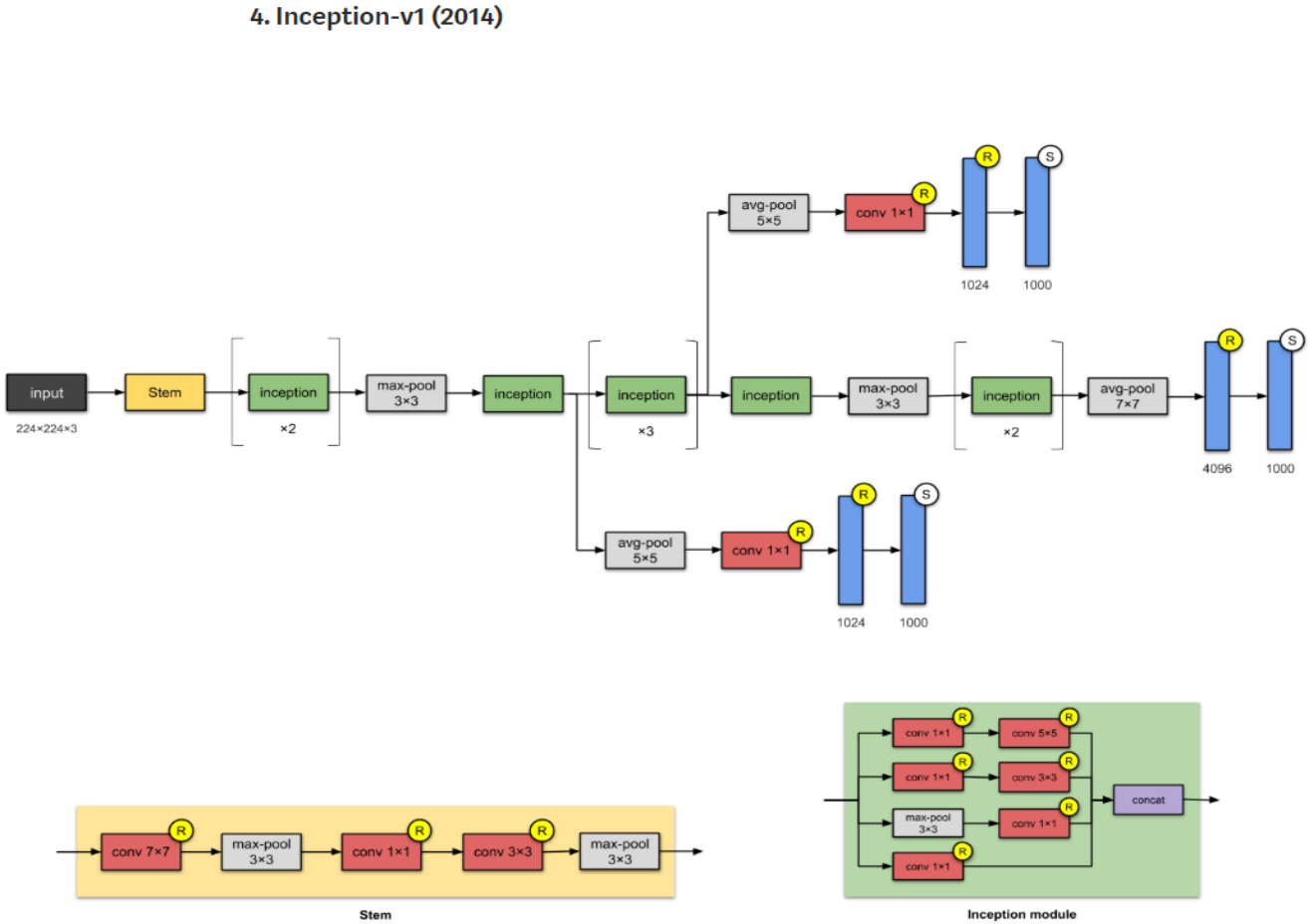
Example CNNs

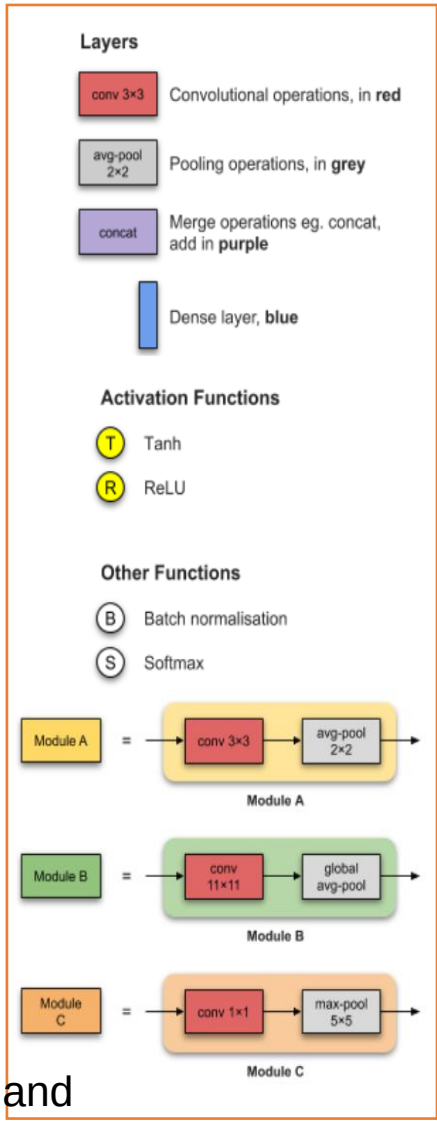




It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another.

Example CNNs

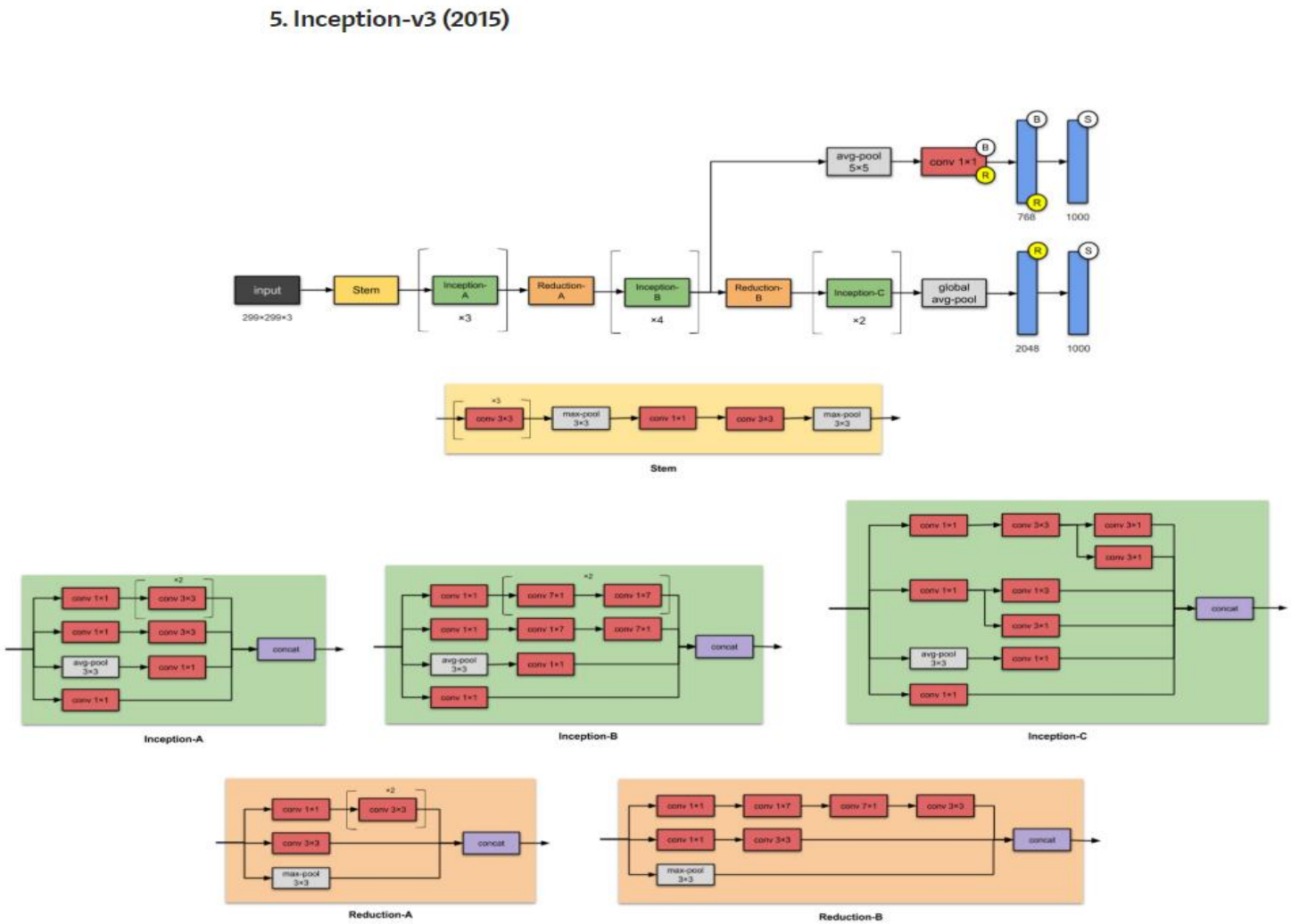


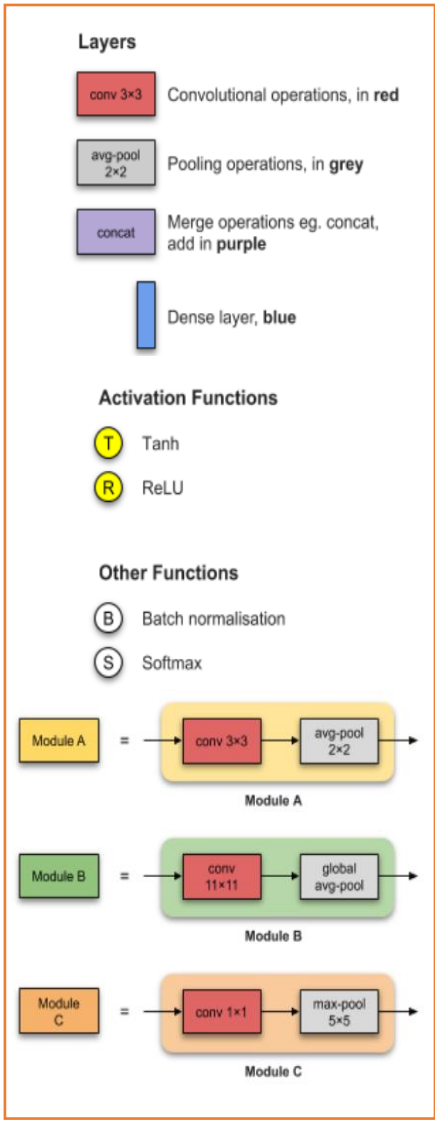


Inception-v1 uses a lot of tricks to push performance, both in terms of speed and accuracy.

1 X 1 convolution: A *1×1* convolution maps an input pixel with all its respective channels to an output pixel. *1×1* convolution is used as a dimensionality reduction module to reduce computation to an extent.

Example CNNs

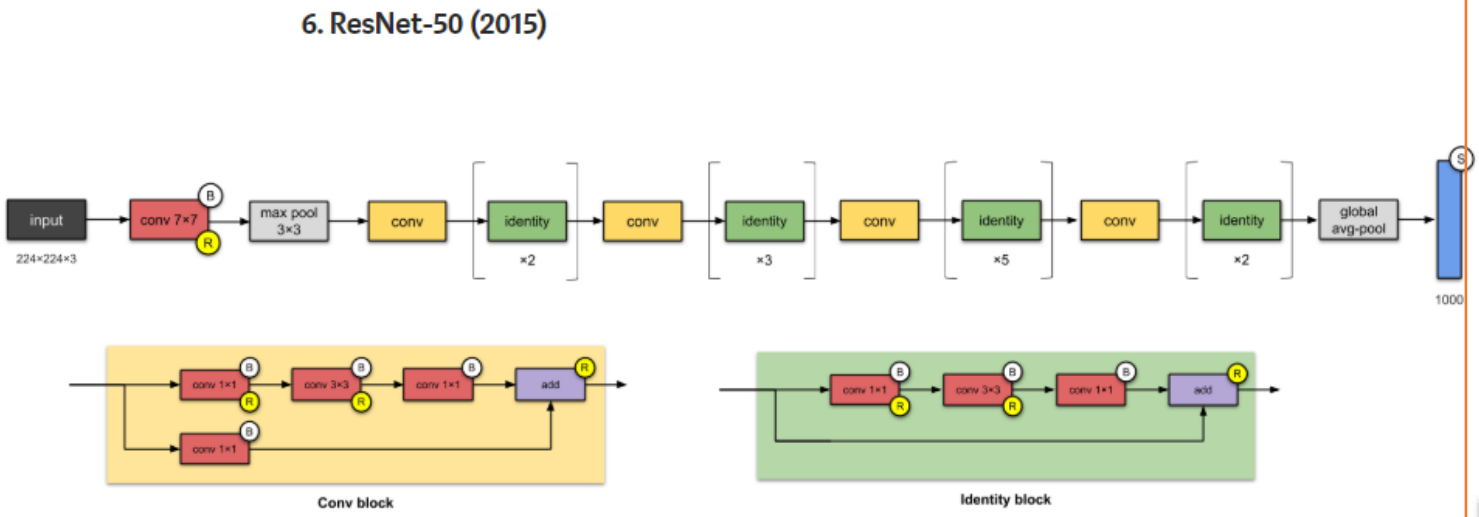


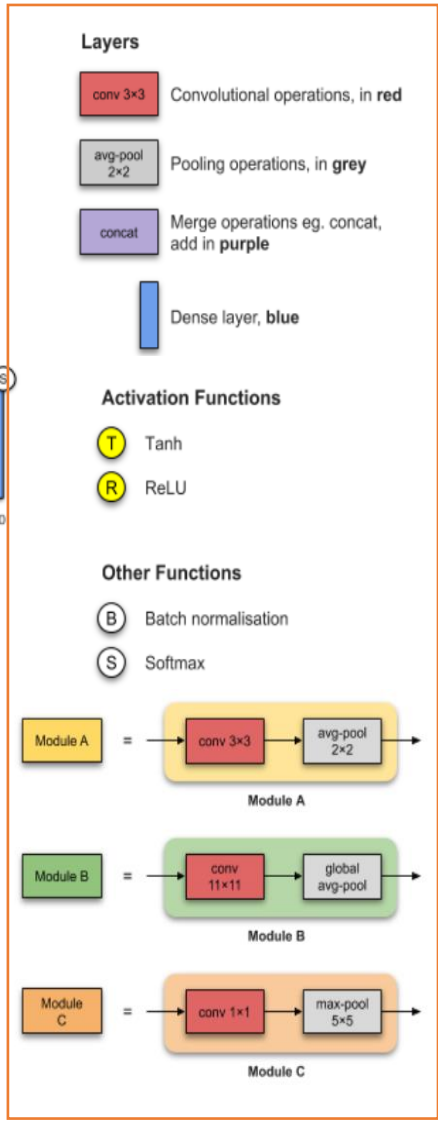


Inception-v3 is a variant of Inception-v2 which adds BN-auxiliary.

BN auxiliary refers to the version in which the fully connected layer is also-normalized.

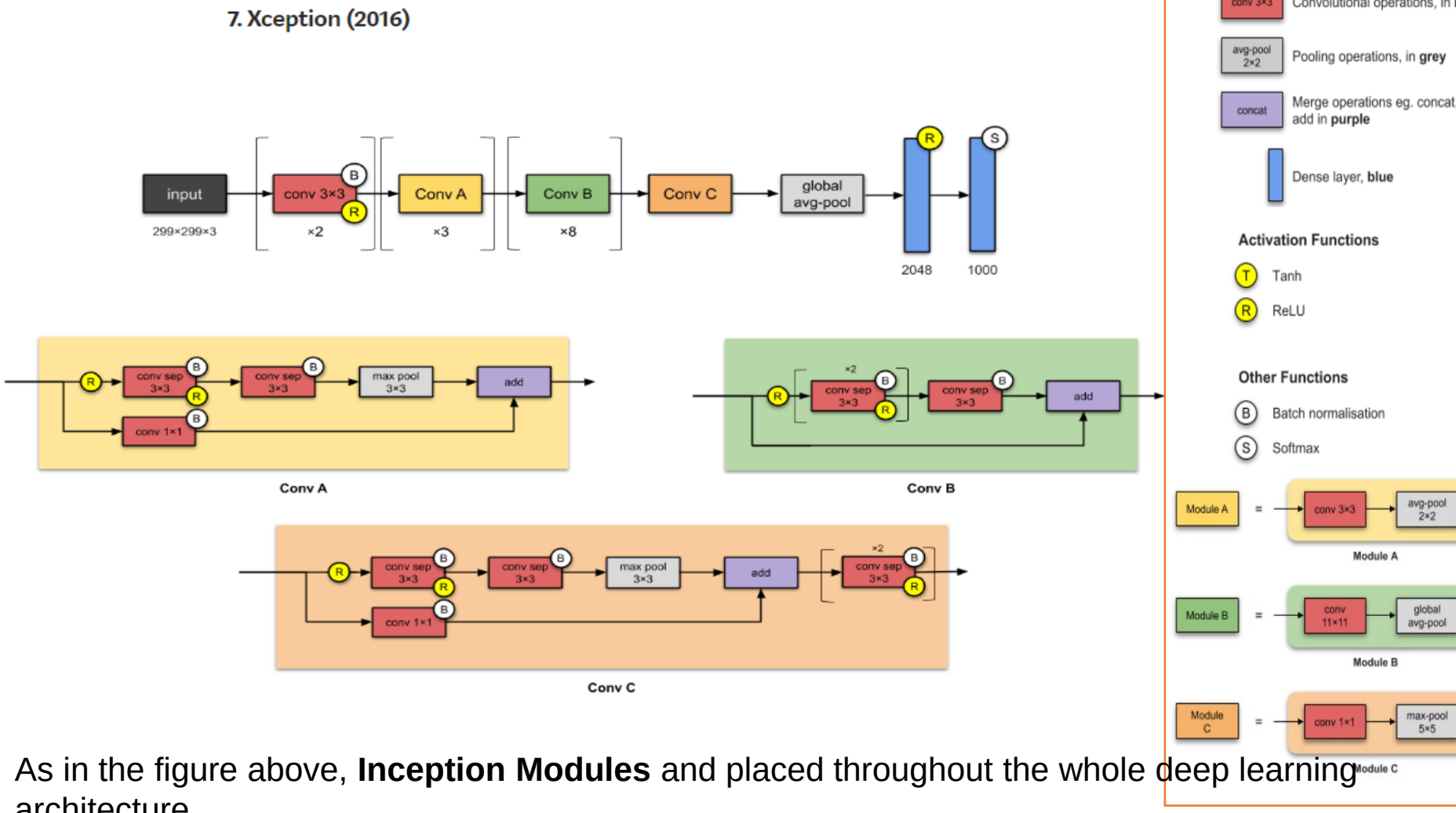
Example CNNs





The “50” refers to the number of layers it has.
The main innovation of ResNet is the skip connection. Deep networks often suffer from vanishing gradients, ie: as the model backpropagates, the gradient gets smaller and smaller.
The skip connection in the diagram above is labeled “identity.” It allows the network to learn the identity function, which allows it to pass the input through the block without passing through the other weight layers!

Example CNNs



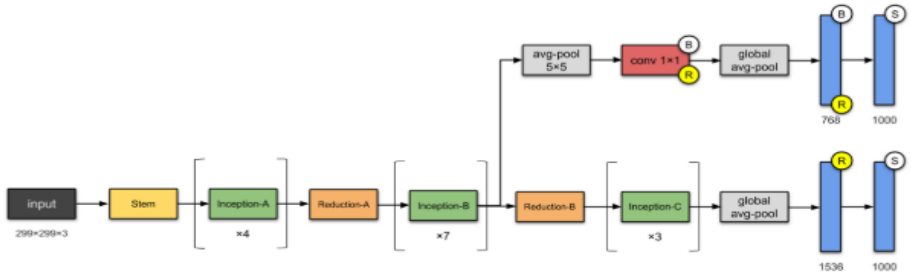
As in the figure above, **Inception Modules** and placed throughout the whole deep learning architecture.

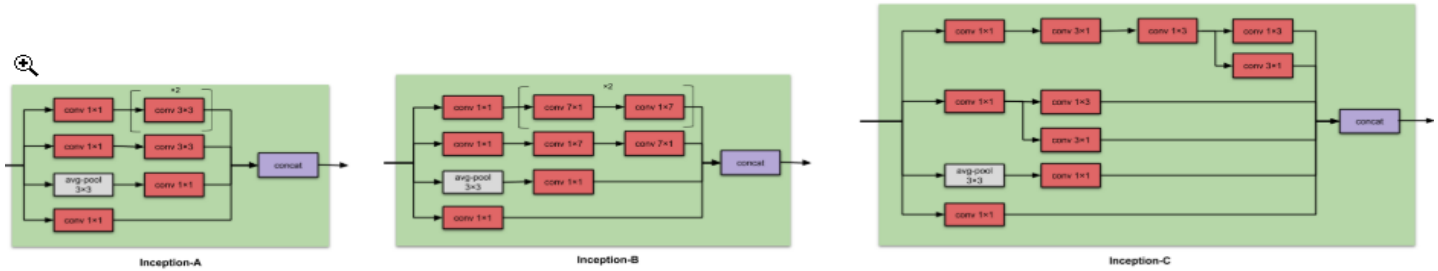
And there are **residual (skip) connections**, originally proposed by ResNet, placed for all flows.

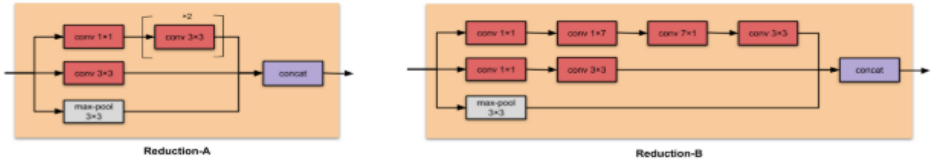
Example CNNs

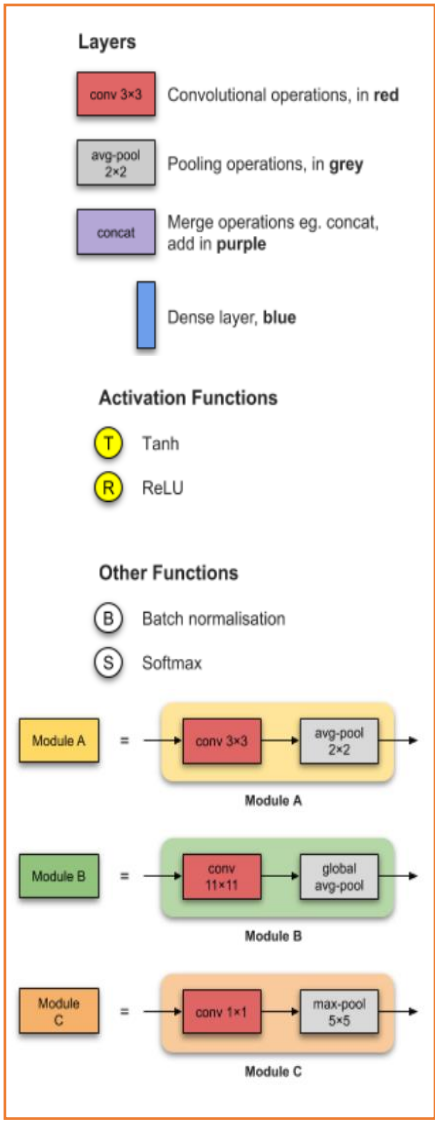
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

8. Inception-v4 (2016)



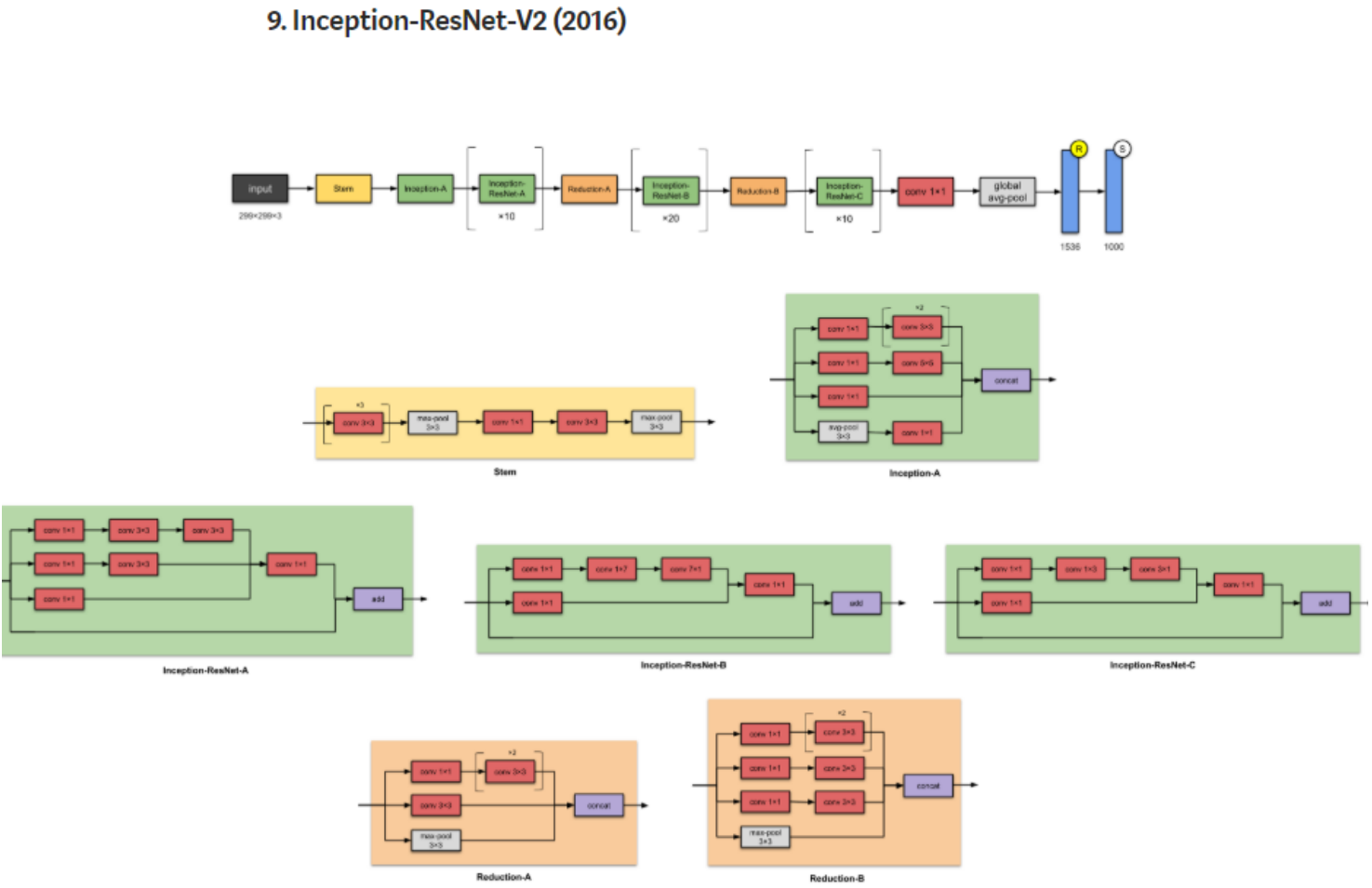


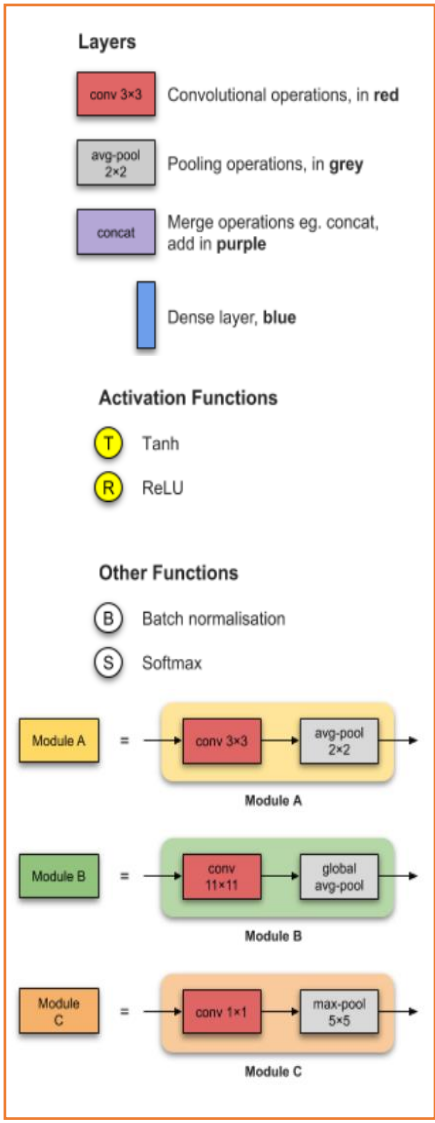




Inception-v4 is a convolutional neural network architecture that builds on previous iterations of the **Inception** family by simplifying the architecture and using more **inception** modules than **Inception**-v3.

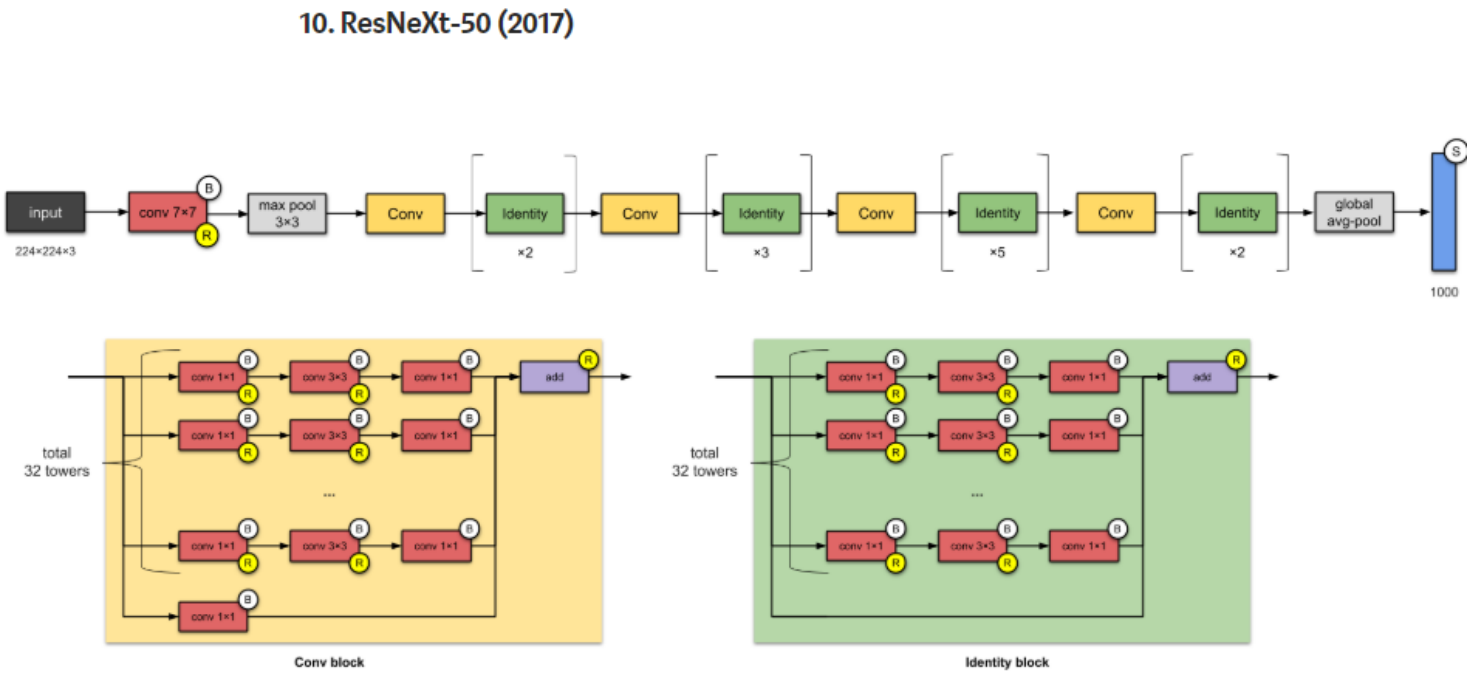
Example CNNs

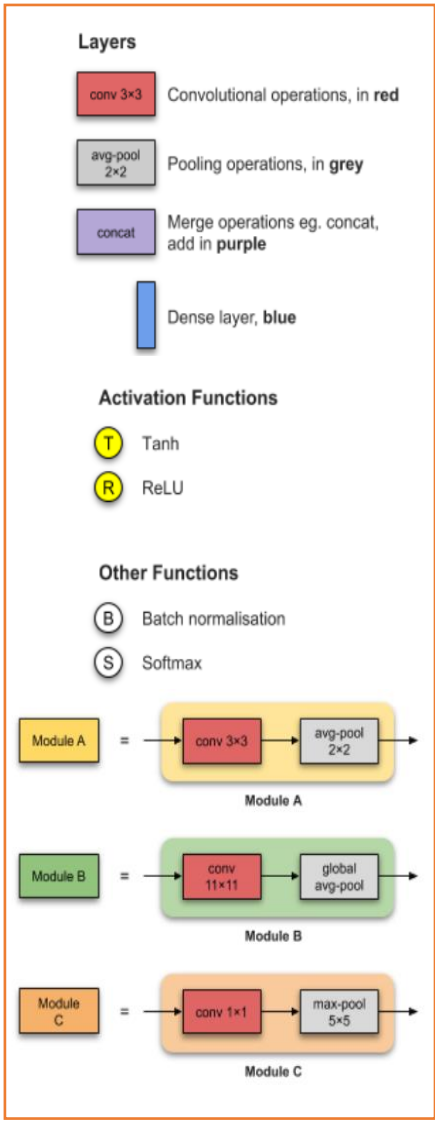




Inception-ResNet-v2 is a convolutional neural architecture that builds on the Inception family of architectures but incorporates residual connections (replacing the filter concatenation stage of the Inception architecture).

Example CNNs





ResNeXt is a simple, highly modularized network architecture for image classification. ResNeXt is constructed by repeating a building block that aggregates a set of transformations with the same topology.

SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Vision Transformers

ViT-Image Classification

- CNNs, e.g., ResNet, were the best solutions to image classification.
- Vision Transformer (ViT) [1] beats CNNs (by a small margin), if the dataset for pretraining is sufficiently large (at least 100 million images).
- ViT is based on Transformer (for NLP) [2].

Reference

- 1. Dosovitskiy et al. An image is worth 16×16 words: transformers for image recognition at scale. In ICLR, 2021.
- 2. Vaswani et al. Attention Is All You Need. In NIPS, 2017.

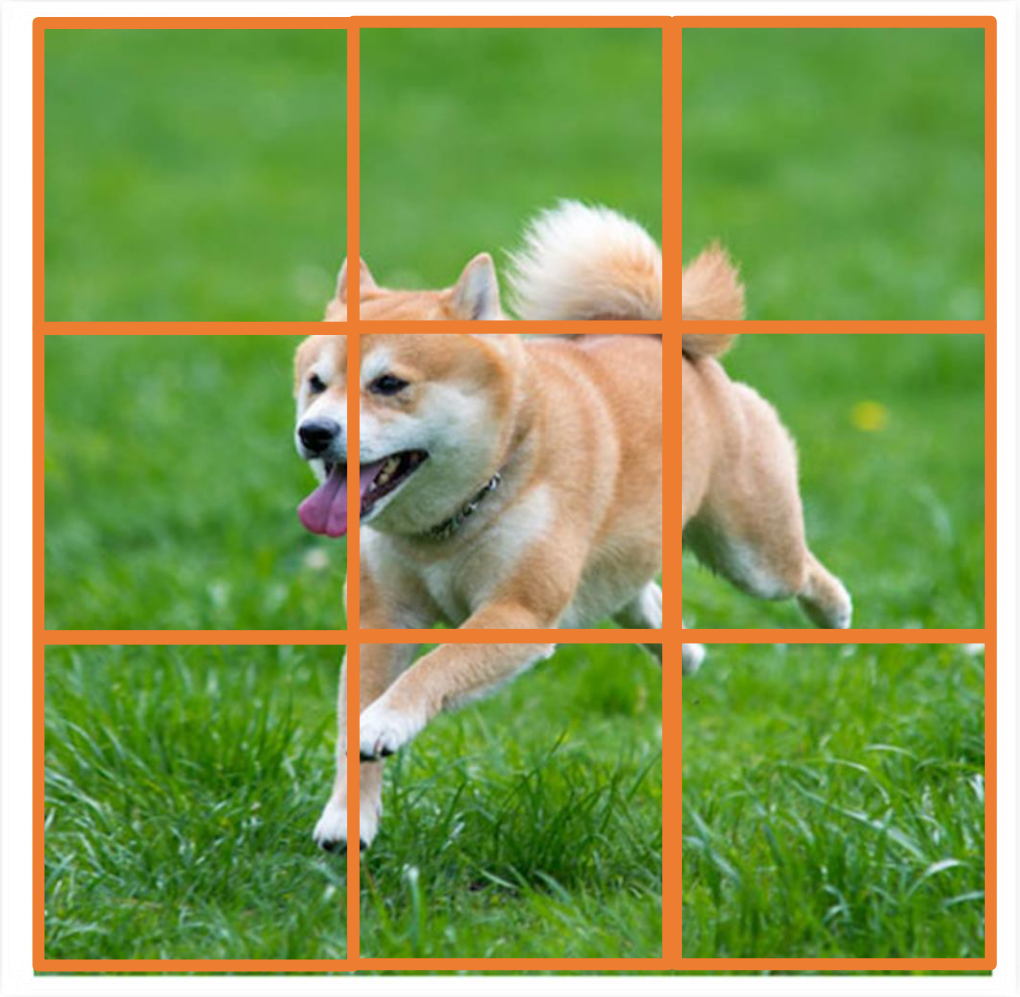
VIT is exactly the Encoder Network of Transformer

ViT-Image Classification

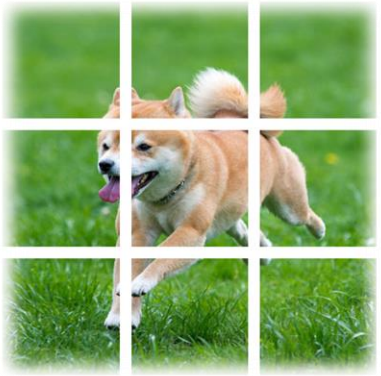
- Split the images into patches

Here, the patches do not overlap.

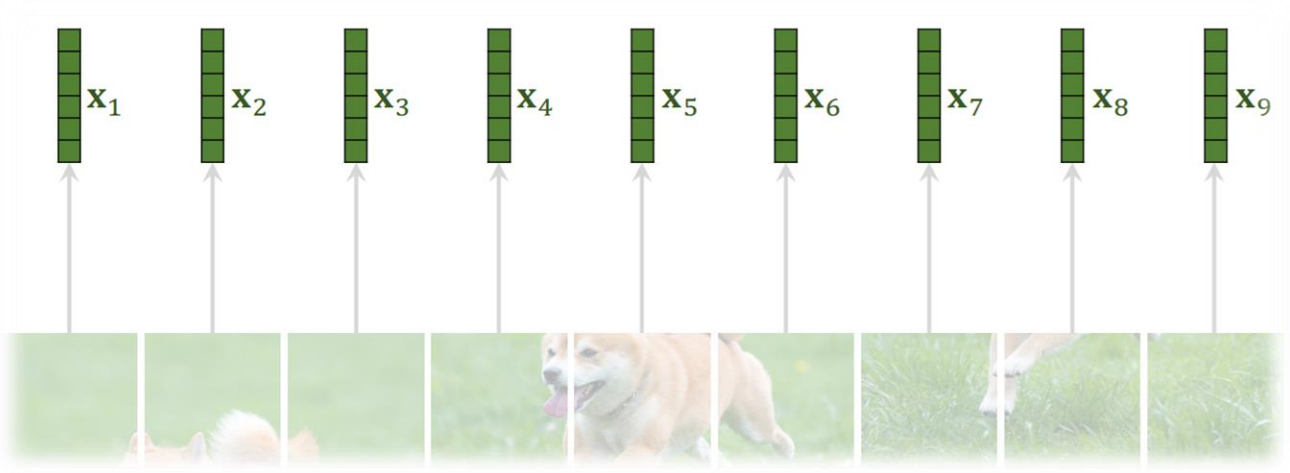
- The patches can overlap.
- User specifies:
  - patch size, e.g., 16×16;
  - stride, e.g., 16×16

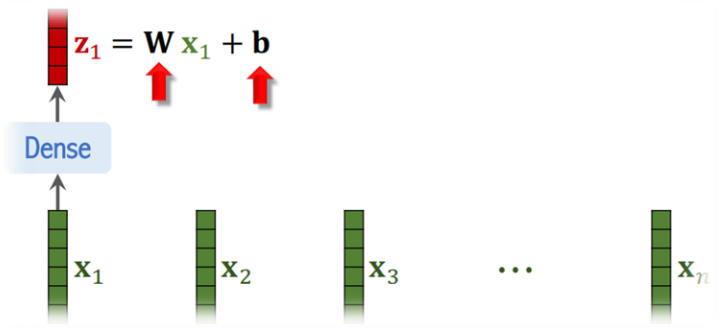


ViT-Vectorization

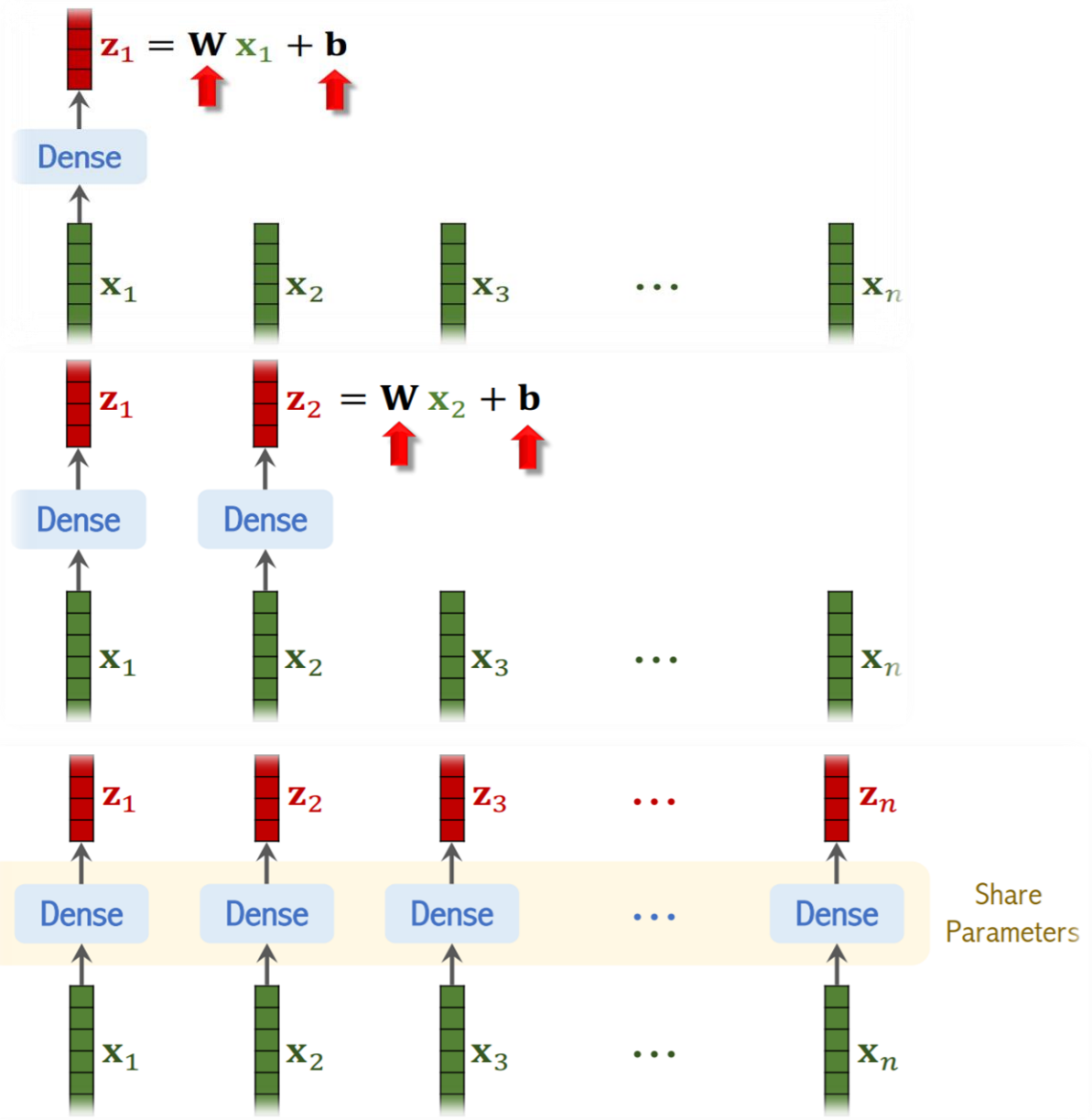


- If the patches are d1xd2xd3 tensors, then the vectors are d1d2d3×1.



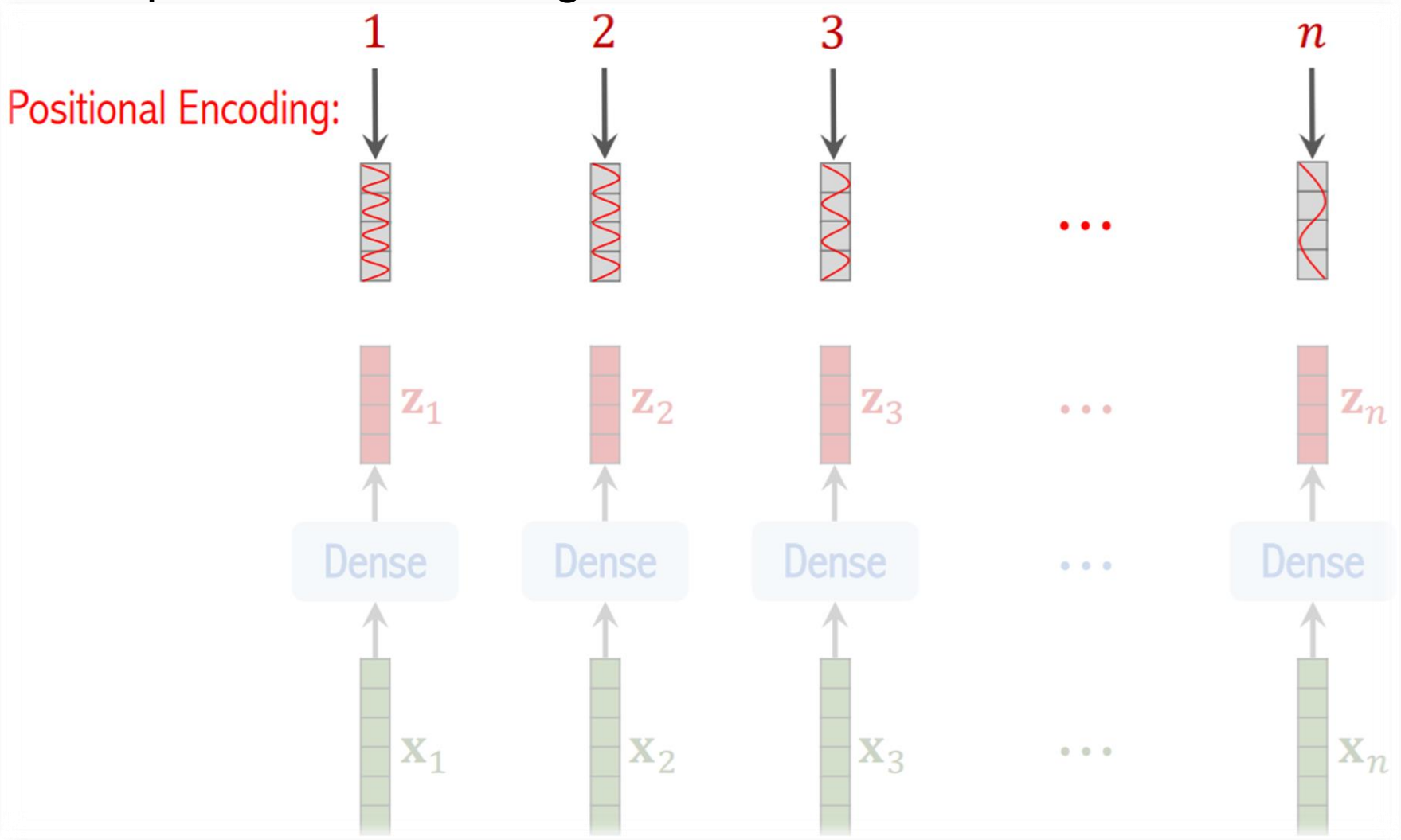


ViT-Vectorization

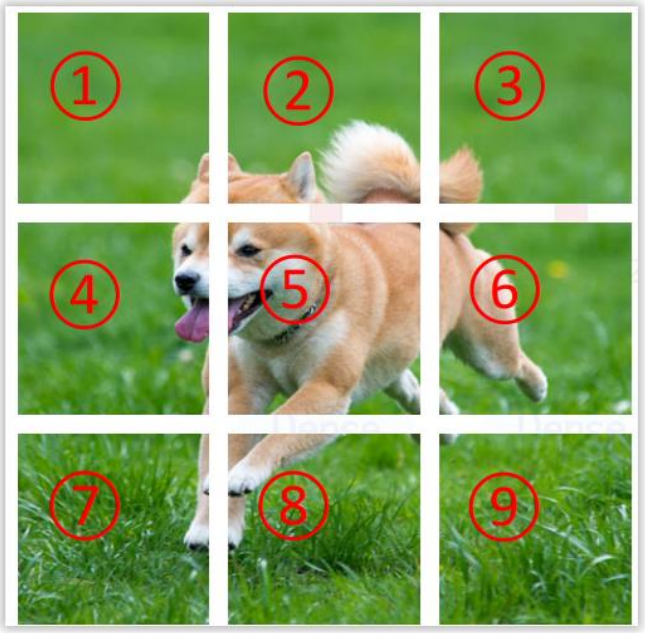


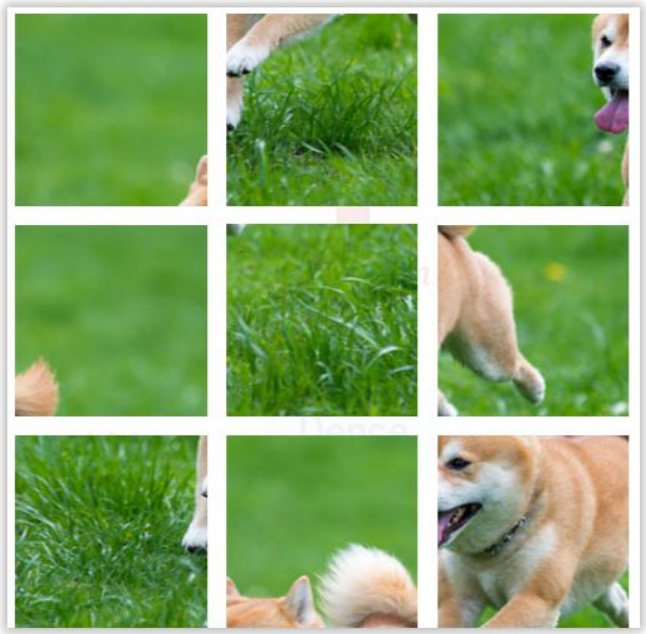
ViT-Vectorization

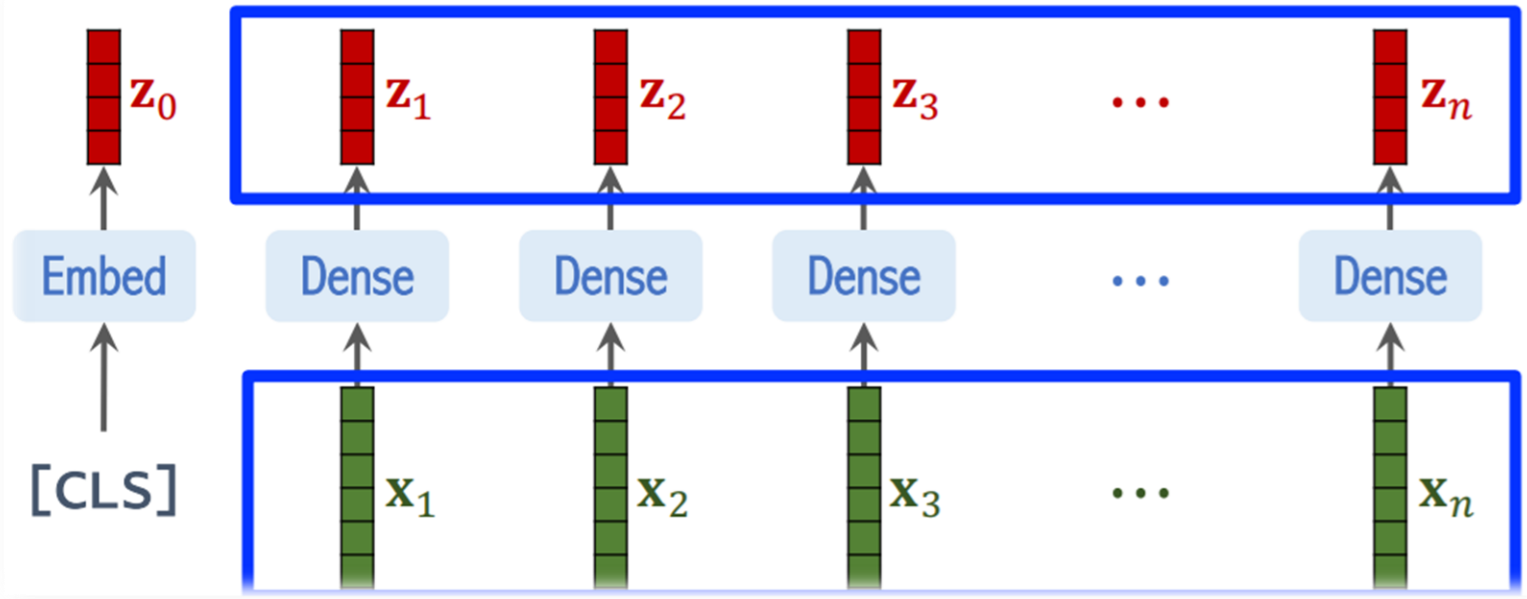
Add positional encoding vectors to z1, z2,…,zn



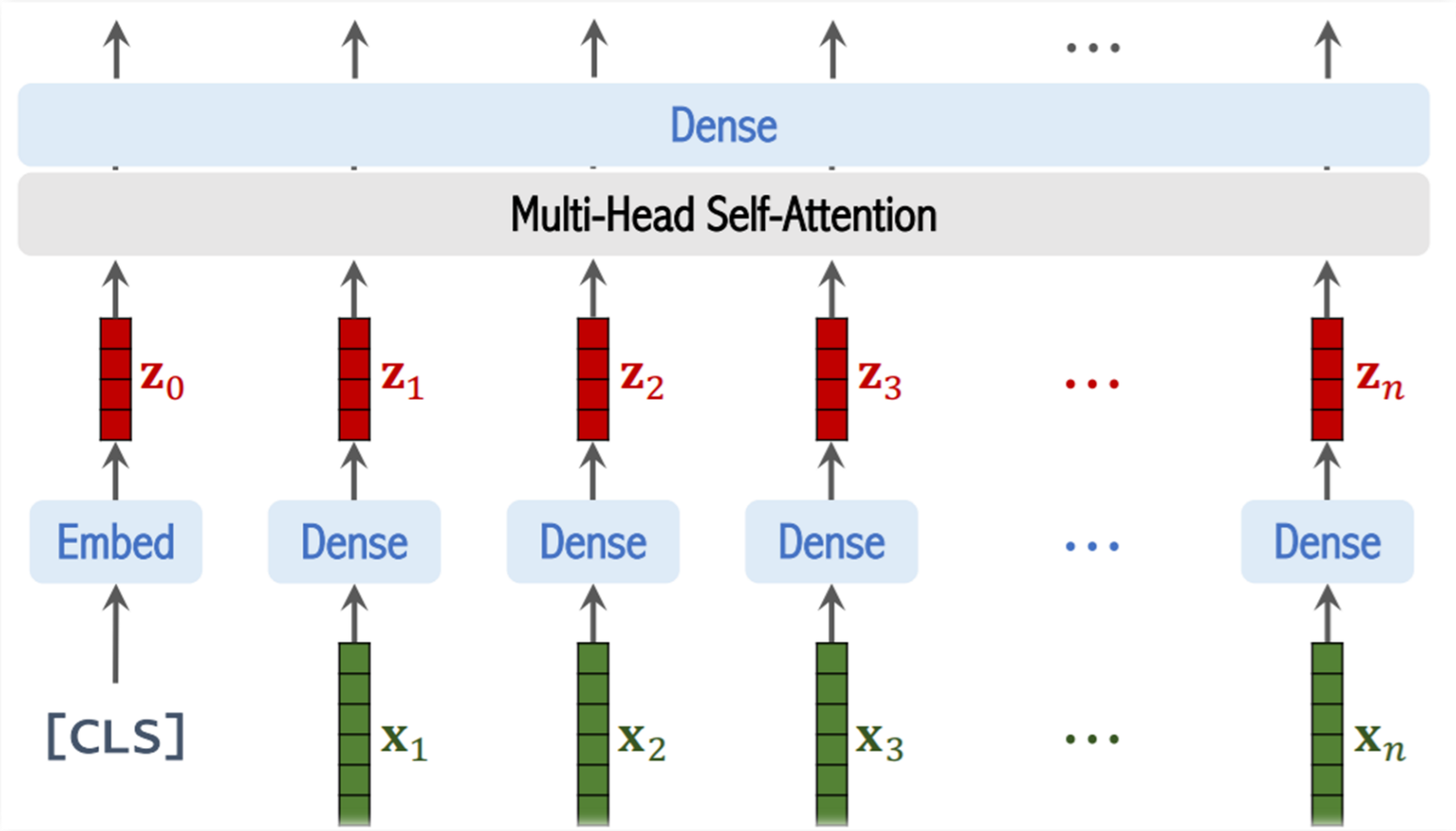
ViT-Positional Encoding



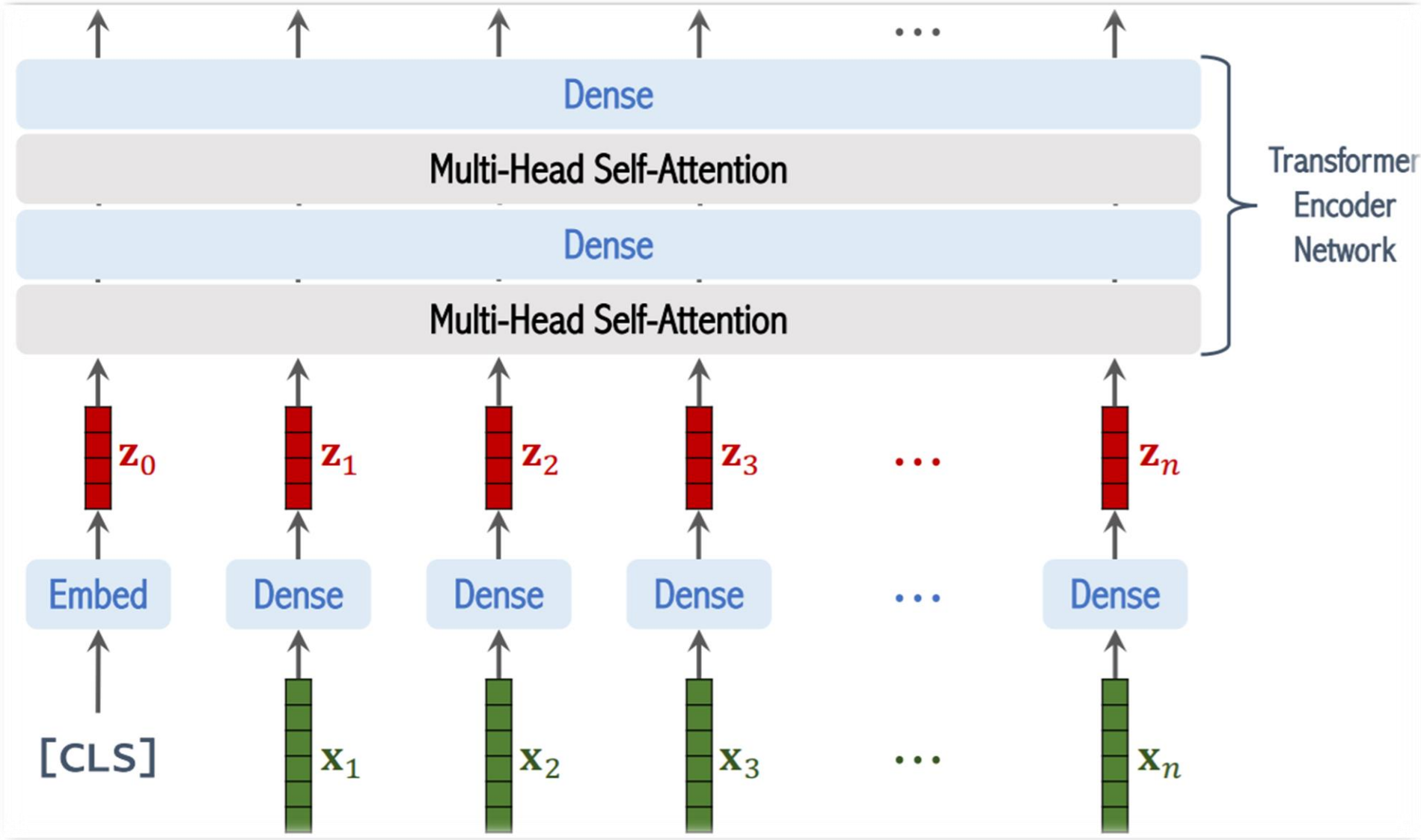




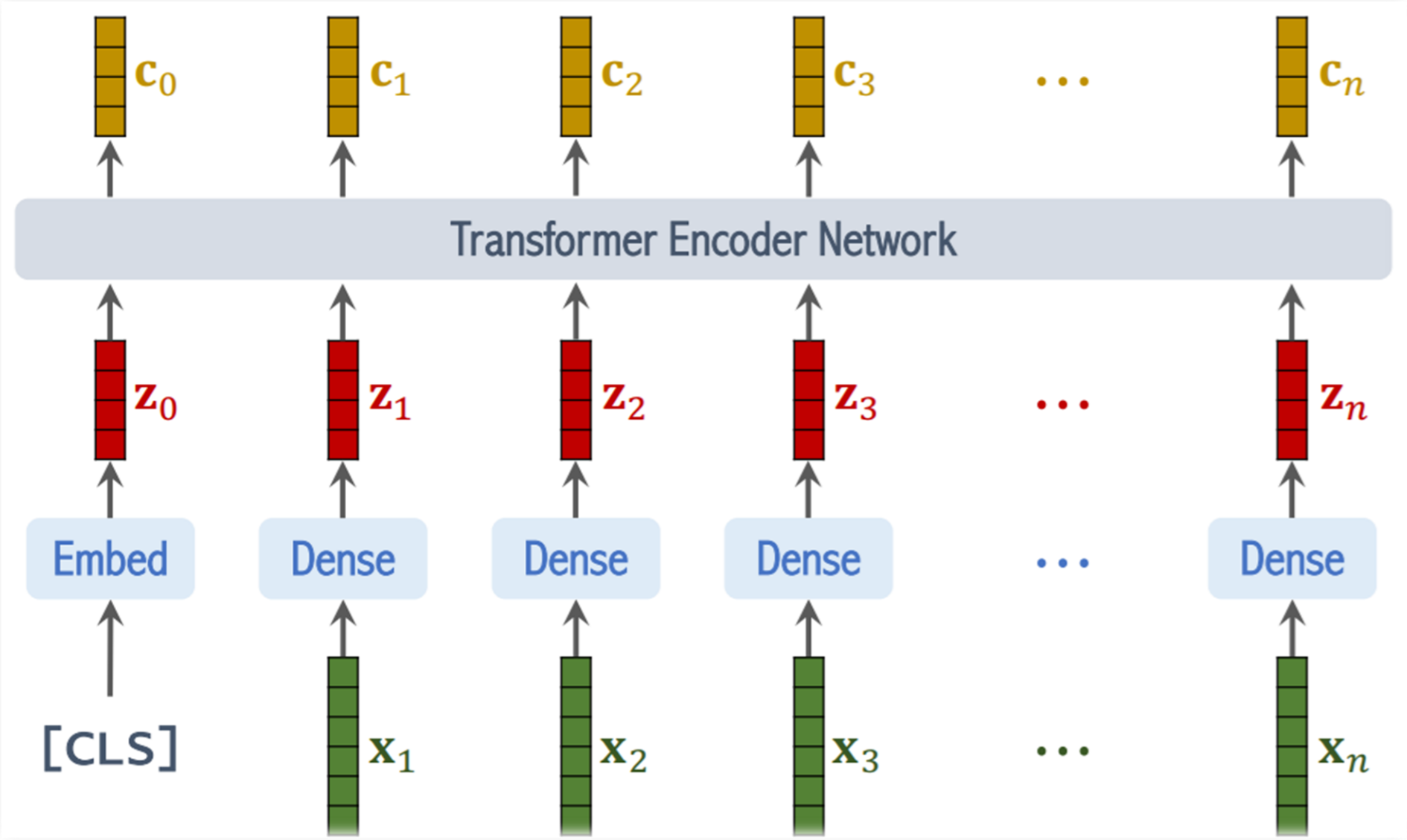
ViT-Positional Encoding



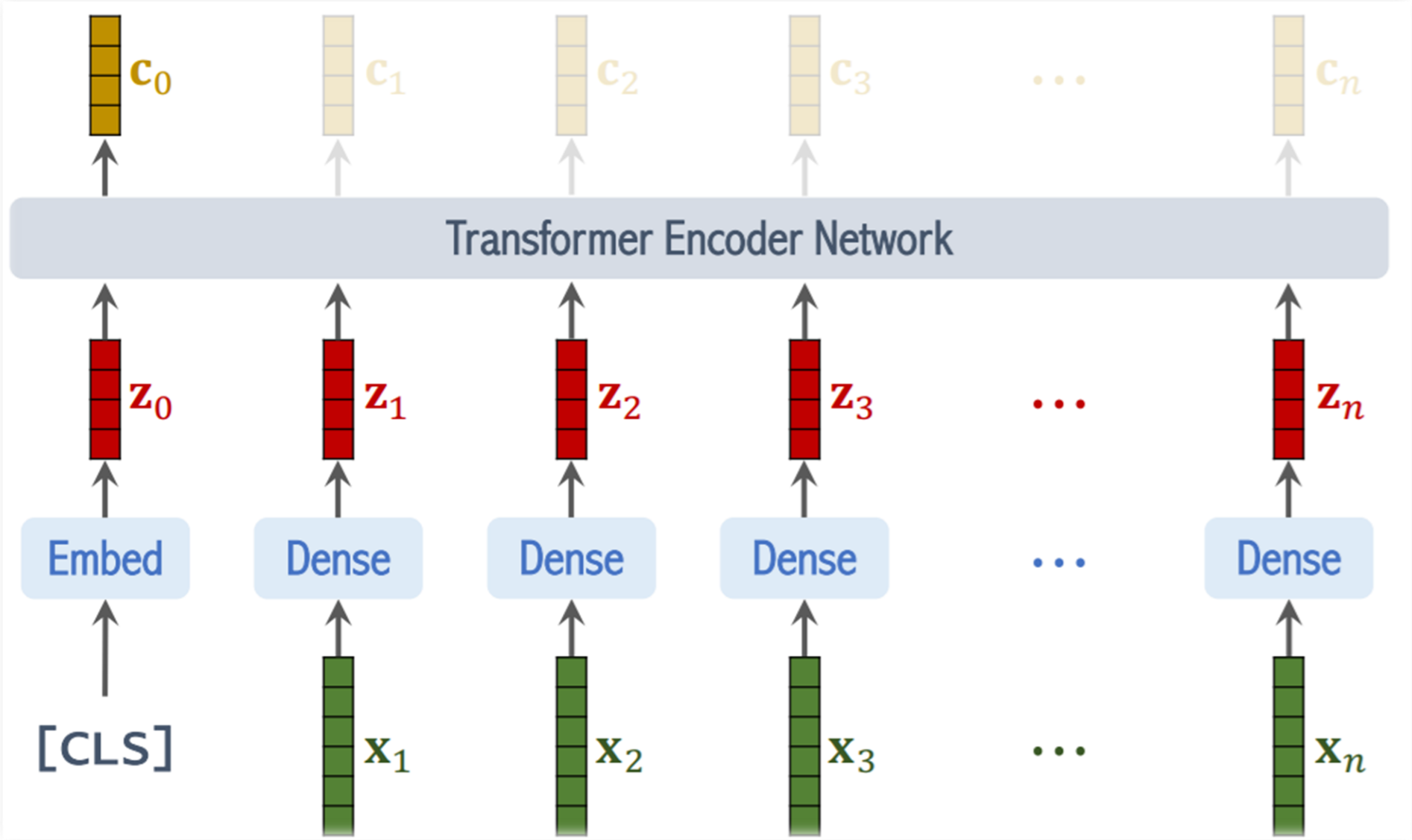
ViT-Positional Encoding



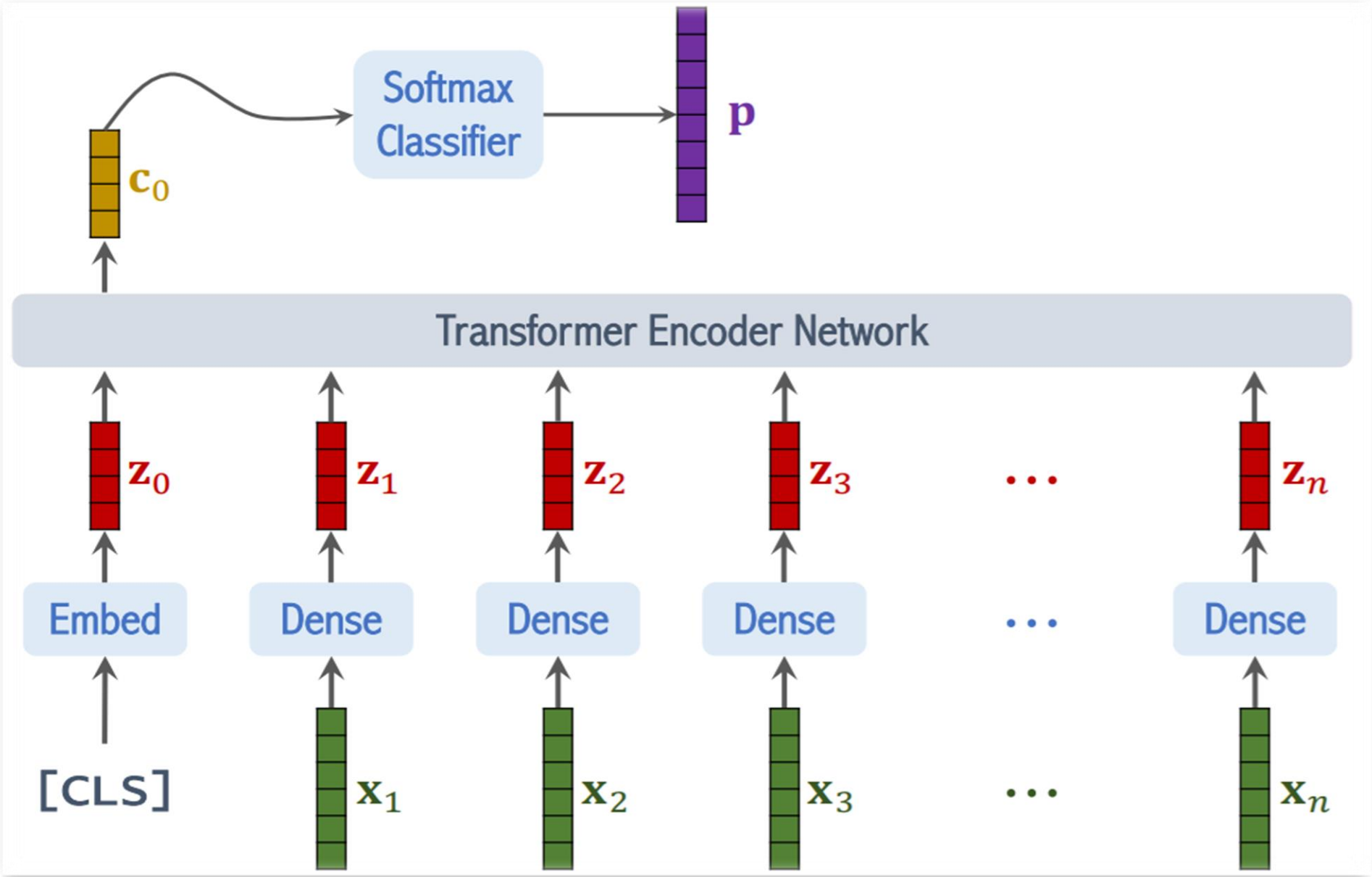
ViT


c_0
c_1
c_2
c_3
...
c_n
Transformer Encoder Network
z_0
z_1
z_2
z_3
...
z_n
Embed
Dense
Dense
Dense
...
Dense
[CLS]
x_1
x_2
x_3
...
x_n

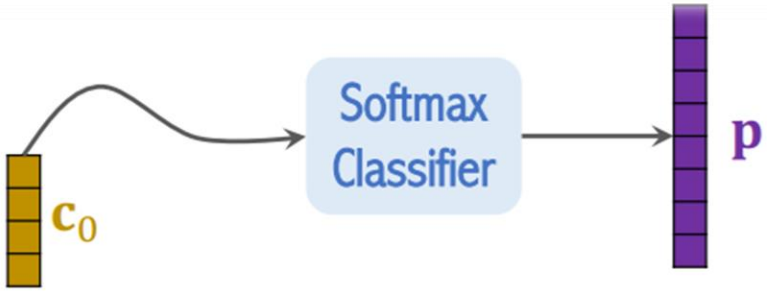
ViT

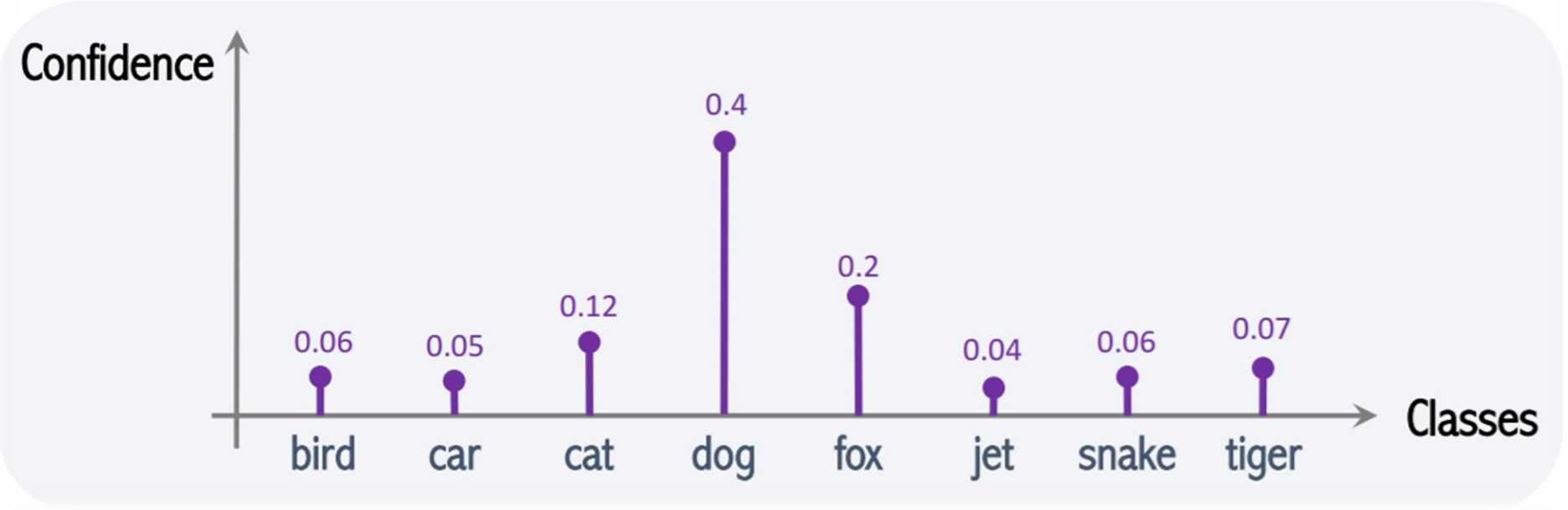

c_0
c_1
c_2
c_3
...
c_n
Transformer Encoder Network
z_0
z_1
z_2
z_3
...
z_n
Embed
Dense
Dense
Dense
...
Dense
[CLS]
x_1
x_2
x_3
...
x_n

ViT


Softmax Classifier
p
c_0
Transformer Encoder Network
z_0
z_1
z_2
z_3
z_n
Embed
Dense
Dense
Dense
Dense
[CLS]
x_1
x_2
x_3
x_n

ViT

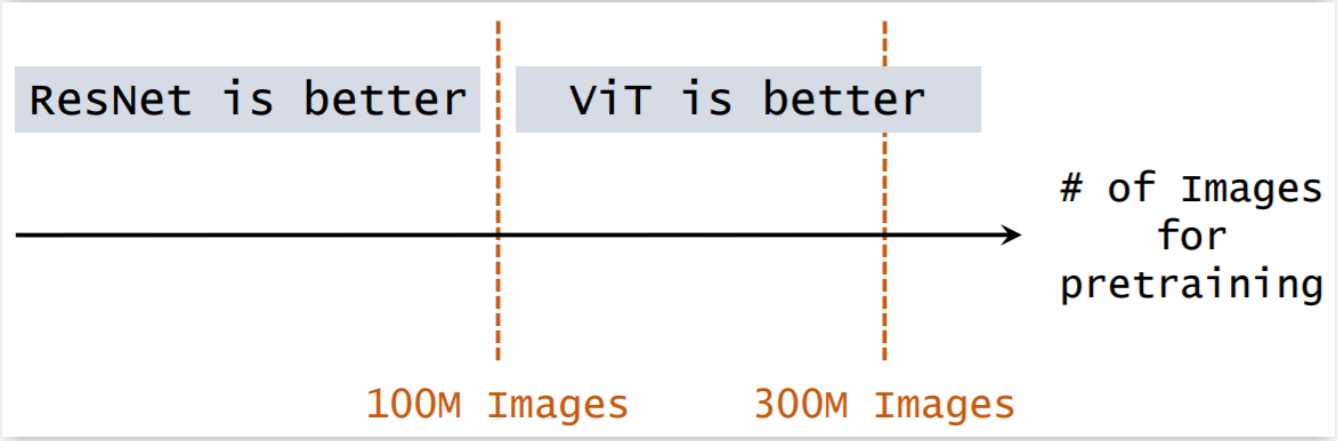

Softmax
Classifier
c_0
p


Confidence
0.06
0.05
0.12
0.4
0.2
0.04
0.06
0.07
bird
car
cat
dog
fox
jet
snake
tiger
Classes

ViT- Image Classification Accuracies

- Pretrained on ImageNet (small), ViT is slightly worse than ResNet.
- Pretrained on ImageNet-21K (medium), ViT is comparable to ResNet.
- Pretrained on JFT (large), ViT is slightly better than ResNet

	# of Images	# of Classes
ImageNet (Small)	1.3 Million	1 Thousand
ImageNet-21K (Medium)	14 Million	21 Thousand
JFT (Big)	300 Million	18 Thousand



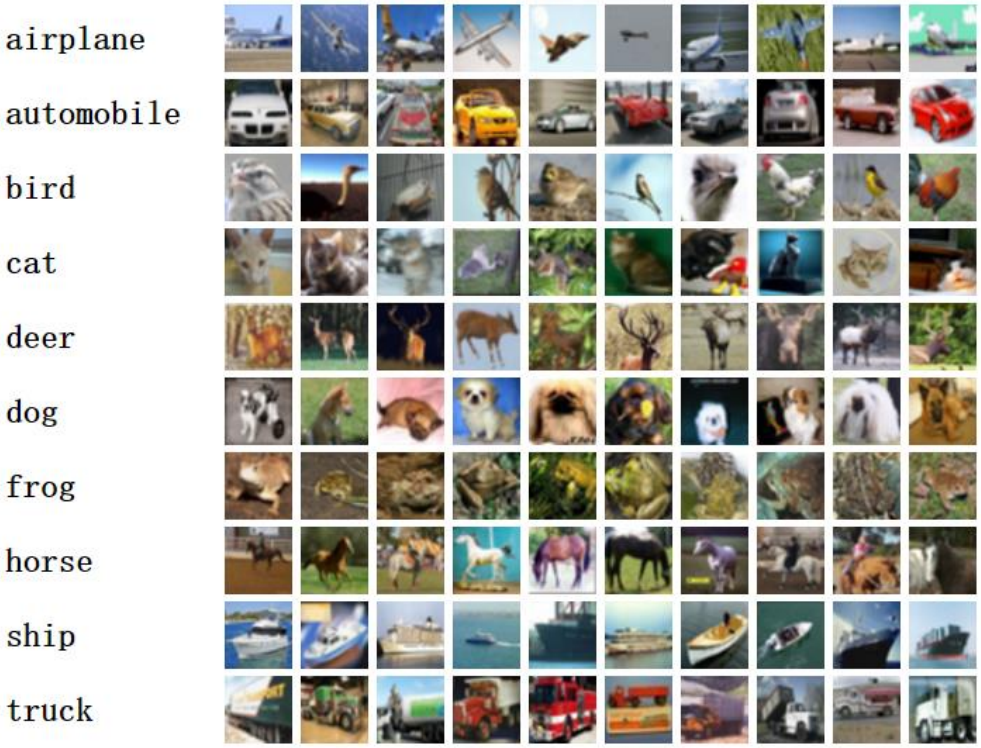
ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Chapter Summary

- We saw the definition of Convolutional Neural Networks
- We visited how to use CNNs for Image Classification
- We discovered how to implement CNNs with Keras
- We reviewed known CNN architectures
- We explained Vision Transformers
- We saw ViT for Image classification
- We mentioned ViT with Pytorch

CIFAR10 dataset and state of the art

The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.



Accuracy

Model	Acc.
VGG16	92.64%
ResNet18	93.02%
ResNet50	93.62%
ResNet101	93.75%
ResNeXt29(32x4d)	94.73%
ResNeXt29(2x64d)	94.82%
DenseNet121	95.04%
PreActResNet18	95.11%
DPN92	95.16%