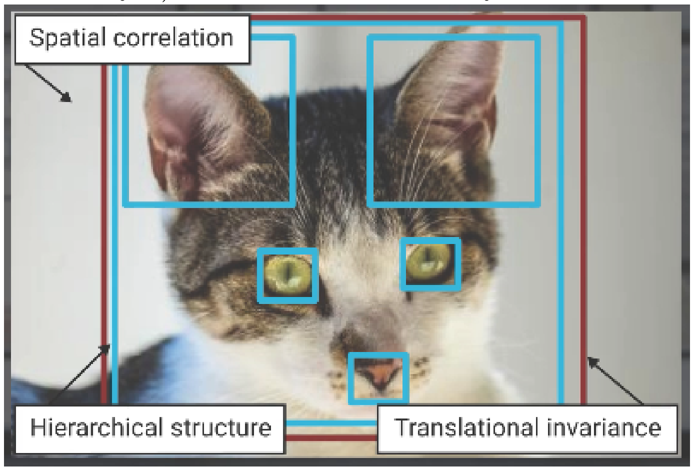
1 Module 22: Convolutional Neural Networks

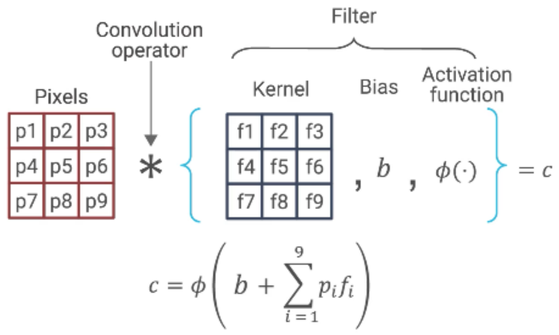
Contents

1.1 Approach

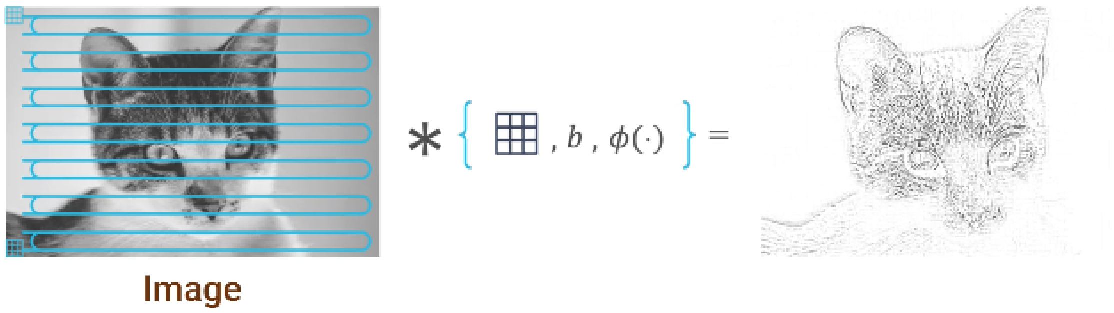
Simply flattening and passing an image through an NN conventionally destroys spatial information. Natural images have regularities, i.e. hierarchical structure (head $\rightarrow$ face $\rightarrow$ eyes) that are translationally invariant.



A **convolution** can be performed on a patch of pixels of the same shape as the kernel.



To produce a **featuremap**, we sweep this patch across the entire image. Each filter will highlight different characteristics.



This convolution is done for a specified number of K filters for the number of features in the convolution layer.

Number of parameters in a convolution layer: (kernel size + 1 (bias term)) $\times$ K. For example, if the kernel size is 3:

$$N_{\text{params}} = (3 \times 3 + 1) \times K \tag{1}$$

Kernel weights and bias factor are learned through training. The modeler manually sets the following:

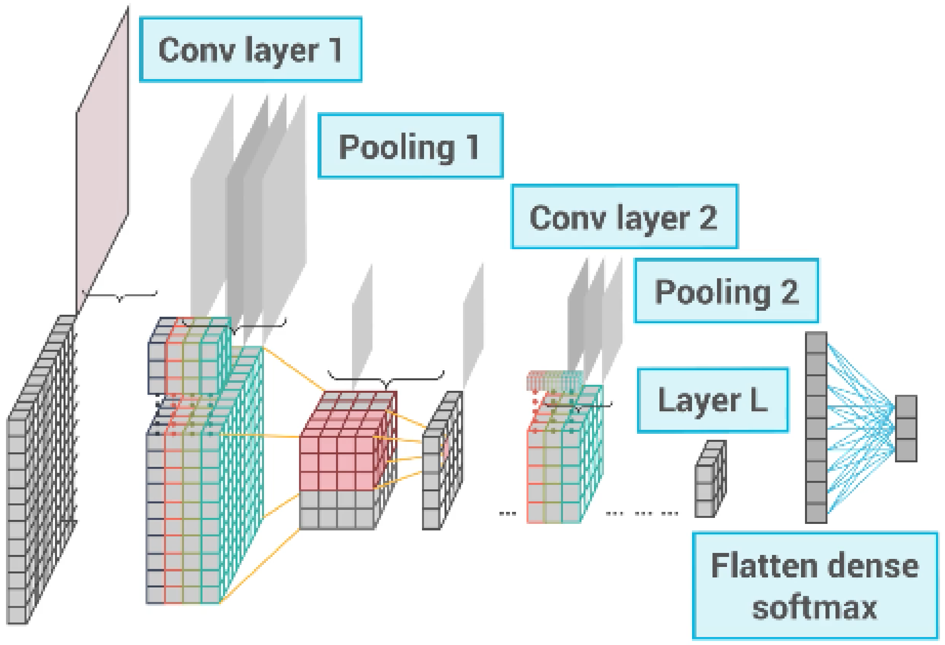
- Padding: whether the include zeros on the margins to keep size(featuremap) = size(image). (Keras: Valid = no padding, Same = same size as image)
- Stride: $n > 1$ calculates every n convolutions. Decreases number of convolutions by a factor of n which decreases featuremap's number of pixels by a factor of n^2.

Convolution layers are often followed by a pooling layer, compressing information down. Most common: **max pooling**, aggregate e.g. 4 tiles and output their max. Sort-of a "zooming out" of the featuremap. Features from the previous level combine to form composite features.

Maxpooling K featuremaps, we will have a 3D featuremap of size(convolution)/tiling factor (e.g. 4) $\times$ K.

We then continue stacking convolution + max pooling layers.

Lastly, we apply a flatten and softmax. This last layer represents only the highest-level features. Then we make a dense layer and produce the outputs.



Essentially: convolution/pooling layers are a process to learn highly nonlinear features, which are used as inputs to a multinomial logistic regression at the end.

Visualize a network: `from keras.utils.vis_utils import plot_model`

1.2 Sequential vs. Functional API

```
model = keras.Sequential([
    layers.Conv2D(filters = 32, kernel_size = 3, activation = 'relu',
    ↪  input_shape:optional = (l, w, 1)),
    layers.MaxPooling2D(pool_size = 2),
    layers.Conv2D(filters = 64, kernel_size = 3, activation = 'relu'),
    layers.MaxPooling2D(pool_size = 2),
    layers.Conv2D(filters = 128, kernel_size = 3, activation = 'relu'),
    layers.Flatten(),
    layers.Dense(10, activation = 'softmax')
])

inputs = keras.Input(shape = (l, w, 1))
x = layers.Conv2d(filters = 32, kernel_size = 3, activation = 'relu')(inputs)
x = layers.MaxPooling2D(pool_size = 2)(x)
x = layers.Conv2d(filters = 64, kernel_size = 3, activation = 'relu')(x)
x = layers.MaxPooling2D(pool_size = 2)(x)
x = layers.Conv2d(filters = 128, kernel_size = 3, activation = 'relu')(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation = 'softmax')(x)
model = keras.Model(inputs = inputs, outputs = outputs)
```

1.3 Data Preprocessing

- Data augmentation: distort, rescale, rotate images.
- Dropout: Randomly disable connections between some neurons.
- Shuffling: NNs learn fastest from the "most random" samples. Choose consecutive samples from different classes.
- Early stopping: Use the weights from the stage of training where validation loss is at minimum.
- Inequal class distribution (prior class probabilities): either remove some higher-frequency classes or duplicate lower-frequency classes. Subsampling → remove patterns randomly or with heuristics wrt. regions with low ambiguity. Can cause lost inforamtion.
- Divide-and-conquer: impose a hierarchichal approach to modularize classification tasks. Useful in large-scale application domains.

1.4 Using a pre-trained network

Chop off the top of the model (the stages Flatten → Dense)

```
base_model = keras.applications.vgg16.VGG16(weights = 'imagenet', include_top =
↪  False)

# Freeze the weights in the pretrained model
base_model.trainable = False

model = keras.Sequential([
    keras.applications.vgg16.preprocess_input # VGG16 has diff. RGB chan config
    base_model,
    layers.Flatten(),
    layers.Dense(1, activation = 'sigmoid') # e.g. for binary classif.
])
```

1.5 Fine-tuning

With a pre-trained network, start by training for several epochs. Then, unfreeze the last few layers of the pretrained model and train again with a lower learning rate.

Bottom-most layers capture universal features. Top-most layers capture high-level layers are generally more custom to the problem.

```
# ... after initial training
base_model.trainable = True # Unfreeze the whole model
for layer in base_model.layers[:-4]: # Then freeze individual layers
    layer.trainable = False
```

1.6 Understanding CNNs

https://towardsdatascience.com/how-to-visualize-convolutional-features-in-40-lines-o

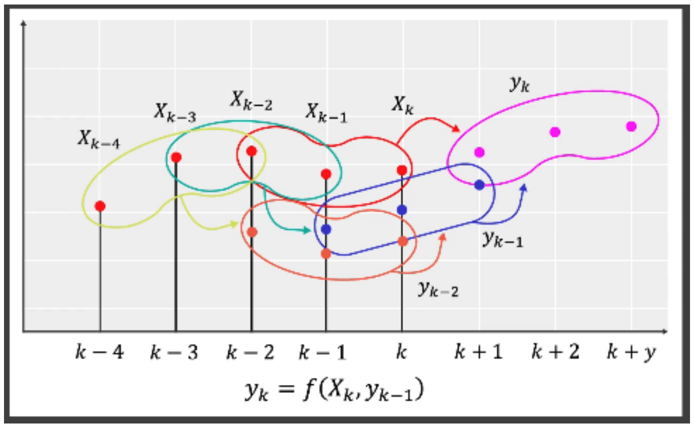
Bottom layers ”look” for broad-scale patterns. Top layers begin to resemble images that humans would recognize.

Auditing layers and being able to understand the network’s approach could be important to understanding weaknesses in the model.

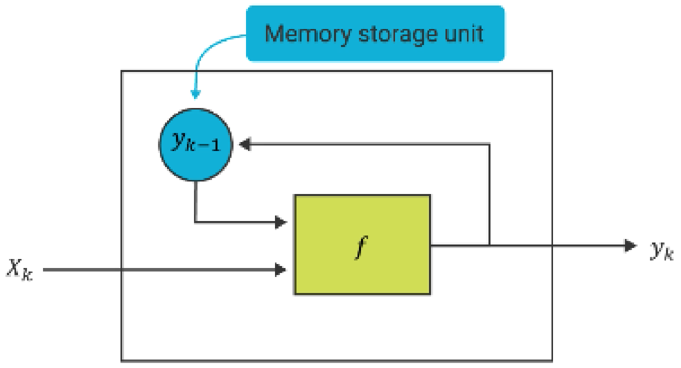
One pixel attack for fooling deep neural networks: https://arxiv.org/abs/1710.08864

1.7 Sequential data

Recurrent neural networks: Cascade predictions from previous data. That previous data is made from the basis of all data previous, recurrently (autoregression). Therefore we have implicit information for all previous data.

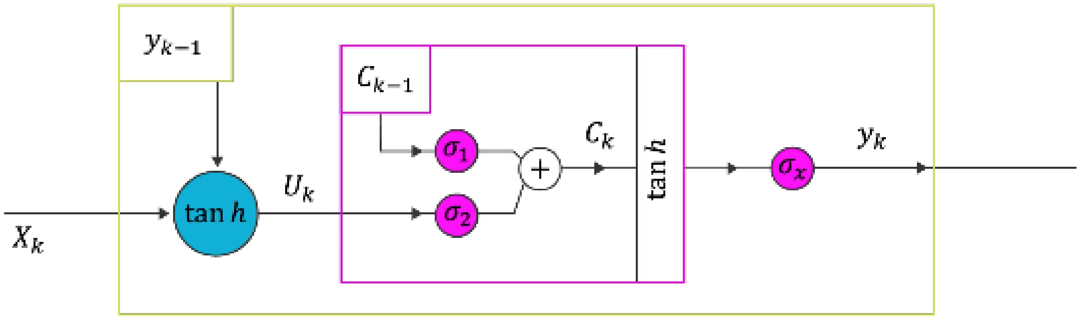


RNNs are then composed of **memory cells** in which (multiple) previous outputs are held in memory. The function f is a dense layer with some activation function.



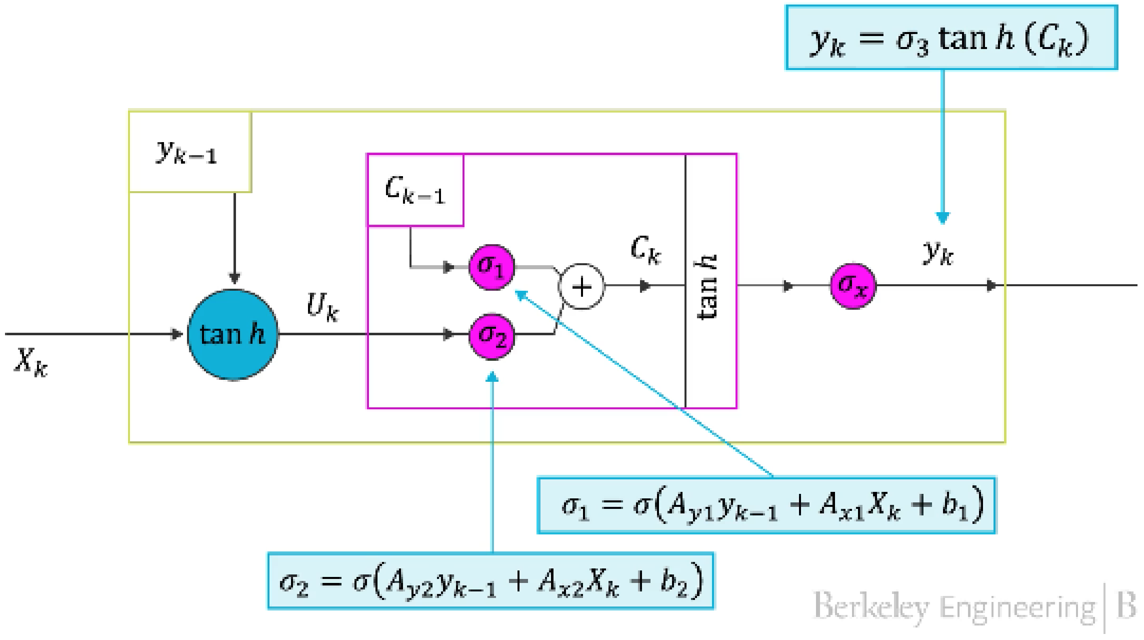
$$y_k = f(X_k, y_{k-1})$$

The drawback is that these memory cells can only predict in short-term. **Long short-term memory cells (LSTMs)** improve performance by nesting two memory cells within one.



Each σ acts a bit like a dimmer switch.

$$C_k = \sigma_1 C_{k-1} + \sigma_2 U_k \tag{2}$$


$y_k = \sigma_3 \tan h\,(C_k)$
y_{k-1}
C_{k-1}
σ_1
σ_2
$+$
C_k
$\tan h$
σ_x
y_k
U_k
X_k
$\sigma_1 = \sigma(A_{y1}y_{k-1} + A_{x1}X_k + b_1)$
$\sigma_2 = \sigma(A_{y2}y_{k-1} + A_{x2}X_k + b_2)$
Berkeley Engineering | B

```
# Prepare data for forecasting
def build_window(data, # data
        h, # history length
        f): # forecast length
    dataX, dataY = [], []

    for i in range(len(data)-h-f):
        dataX.append(data[i:(i+h)])
        dataY.append(data[(i+h):(i+h+f)])
    dataX = np.array(dataX)
    dataY = np.array(
        np.reshape(dataX, ( data.shape[0], 1, dataX.shape[1] ))
        )

    return dataX, dataY

# Example of 3 different model types
model_dense = Sequential([
    Flatten(),
    Dense(16, activation = 'relu'),
    Dense(24)
])
model_RNN = Sequential([
    SimpleRNN(16, input_shape = (1, 6)),
    Dense(24)
])
model_LSTM = Sequential([
    LSTM(16, input_shape = (1, 6)),
    Dense(24)
])
```

1.8 NNs for regression

Final output layer doesn't have an activation (instead of softmax or sigmoid). Final stage is then simply a linear regression on the features generated by the previous stage.

```
# e.g.
model = keras.Sequential([
    layers.Dense(64, activation = 'relu'),
    layers.Dense(64, activation = 'relu'),
    layers.Dense(64, activation = 'relu'),
    # ...
    layers.Dense(1, activation = None)
])

model.compile(optimizer = 'rmsprop', loss = 'mse', metrics = ['mae'])
```