Machine Learning for Music

Module 2: predictive pipelines for music & MIR

Music information retrieval

Main references for this module:

“Fundamentals of Music Processing” (Meinard Müller, International Audio Laboratories Erlangen):

https://www.audiolabs-erlangen.de/resources/MIR/FMP/C0/C0.html

“Generative AI for Music and Audio Creation” (Michigan):

https://hermandong.com/teaching/pat498_598_fall2024/

and **“Music Classification: Beyond Supervised Learning, Towards Real-world Applications” (Minz Won, Janne Spijkervet, Keunwoo Choi),** specifically:

https://music-classification.github.io/tutorial/part2_basics/input-representations.html

Music Information Retrieval

Music Information Retrieval (or MIR) is a fairly broad term…

Historically it might have included tasks such as (content-based) search (i.e., “retrieval”), but also classification tasks (chord labeling), or other predictive tasks (source separation, onset detection, etc.; see later)

These days, the term might be used to cover *any* ML involving music quite broadly, e.g. conferences like ISMIR have come to include all sorts of papers on generative AI etc.

But in this class, we’ll use the term “MIR” to describe more “traditional” approaches (usually *discriminative*) compared to *generative* approaches (Modules 3 and 4)

Introduction to Music Information Retrieval (MIR)

What is Information Retrieval?

Obtaining relevant information from a large collection of data

International Society for Music Information Retrieval (ISMIR)



- Non-profit organization that focuses on music information retrieval (MIR) research
- Annual conference (ISMIR conference)
- Vibrant community (ISMIR community)

Example task: Genre classification

Let's start with the task of *genre classification:* that is, we'd like to build a pipeline that makes predictions like:



from https://music-classification.github.io/tutorial/part1_intro/what-is-music-classification.html

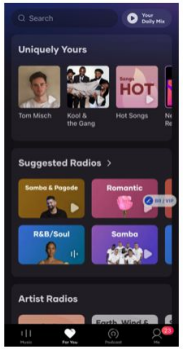
Why genre classification?

Recommendation: e.g. features may be useful “downstream” for a content-based recommender system

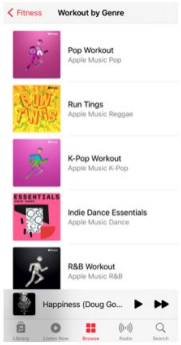
Curation: e.g. just to browse and organize content more efficiently

Playlist generation

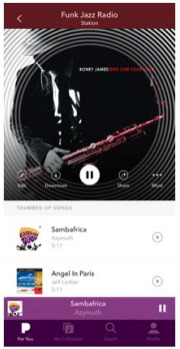
Listening behavior analysis



Recommendation



Curation



Playlist generation


Analysis

Related tasks

Similarly, one could imagine a number of related tasks:

- Genre classification, see e.g. *Tzanetakis and Cook: Musical genre classification of audio signals*
- Mood classification, see e.g. *Kim et al.: Music emotion recognition: a state of the art review*
- Instrument classification, see e.g. *Herrera-Boyer et al.: Automatic classification of musical instrument sounds*
- Music tagging, see e.g. *Lamere: Social tagging and music information retrieval*

Of these, ***tagging*** is perhaps the most general, as tags could potentially subsume any other types of labels

Predictive pipelines for music & MIR

2.1: (More) ML fundamentals

ML fundamentals

Sorry!

We have to take a break from musical fun for a bit to talk about ML :-(

ML fundamentals

In the interests of time, let's regard this component as mostly being **self-study;**

If I finish this module early (and anyone is interested…) I can spend some more time covering these fundamentals

See also:

- workbook2.ipynb (course webpage) which shows you how to apply many of these concepts
- CSE158/258 slides (https://cseweb.ucsd.edu/classes/fa24/cse258-b/) and (free) course textbook (https://cseweb.ucsd.edu/~jmcauley/pml/) which cover these in *way* more detail

ML fundamentals

To those less interested in ML: Feel free to skim over the “foundational” stuff and focus on getting the code in the notebooks working, and understanding the notebook functions in terms of inputs and outputs (i.e., think of everything as a black box)

As usual, think of this as a list of concepts that *might* come up or which might be mentioned when presenting case studies later, not a list of concepts you *need* to know for this class

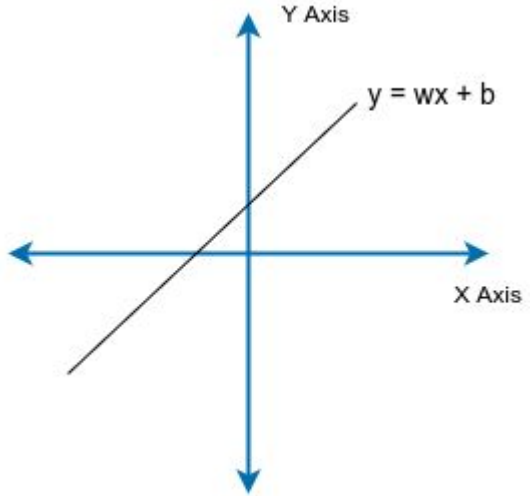
Linear models

Basic equation

$$y = wx + b$$

- y is what we want to predict (output)
- x is our input data (features)
- w is the weight (slope)
- b is the bias (y-intercept)

E.g., think of *x* as the number of hours studied and *y* as your test score



https://medium.com/analytics-vidhya/activation-functions-part-1-d80b0e8ee89d

Linear models

Basic equation

- However, most ML models require multiple inputs

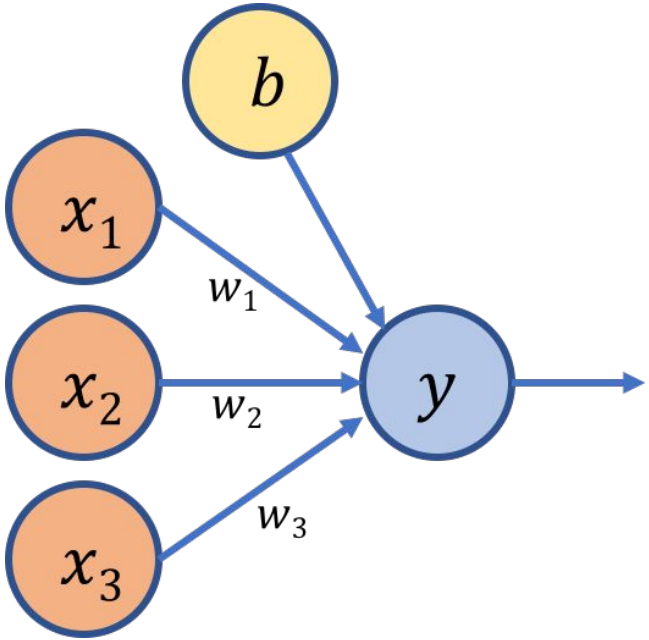
  (for example, we can predict your test score based on studying hours *and* your prior knowledge).

- So instead of y = wx + b, we generalize this as...

$$y = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$

Linear models

$$y = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$



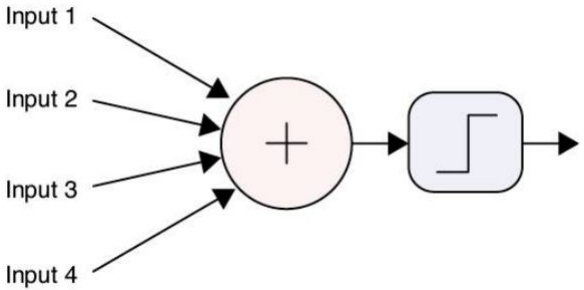
https://medium.com/@melodious/understanding-deep-neural-networks-from-first-principles-logistic-regression-bd2f01c9e263

Linear models

Artificial neuron

This mathematical function, which takes multiple weighted inputs, applies a bias term, and produces an output is called an artificial neuron (or simply a *neuron*)

$$y = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

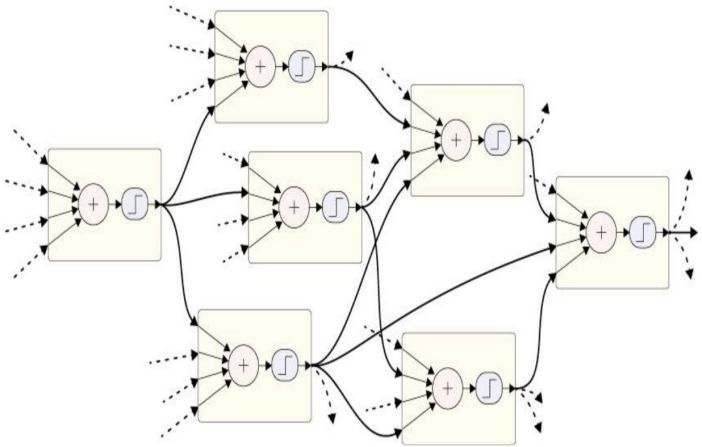


artificial neuron (bias term and weights not shown)
("Deep Learning A Visual Approach")

Linear models

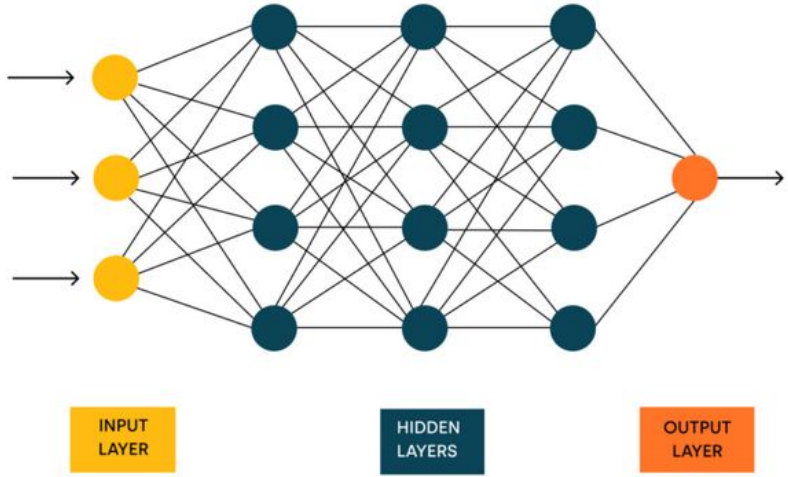
Artificial Neural Network

Like real neurons, artificial neurons can be wired up in networks, where each input comes from the output of another neuron; this is called a *neural network*



Linear models

A **Feed Forward Network:** a type of neural network where information flows in one direction only without any loops or cycles in the connections between neurons

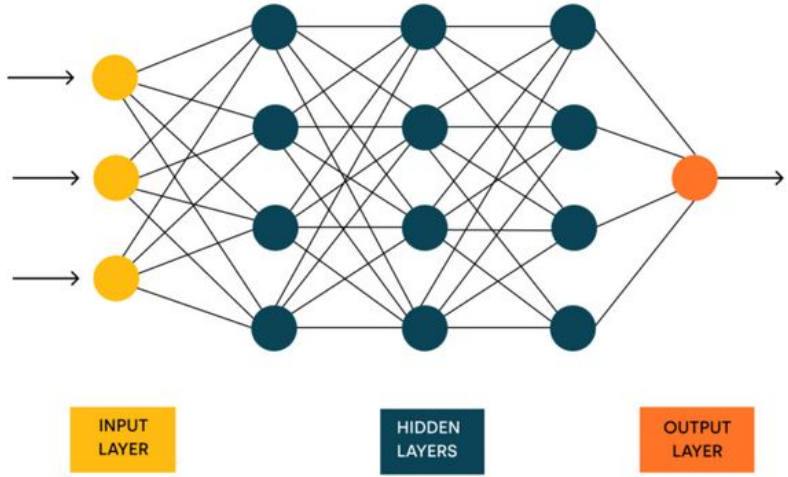


https://www.qtravel.ai/blog/what-are-neural-networks-and-what-are-their-applications/

Linear models

A **Feed Forward Network** is composed of input layer, hidden layer, and output layer:

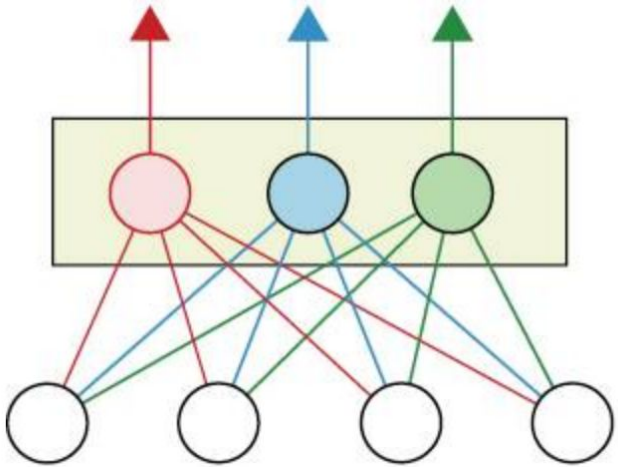
The ***input layer*** receives raw data; ***hidden layers*** process and transform that data through multiple artificial neurons; and the ***output layer*** produces the final predictions or results based on the network's learned patterns



https://www.qtravel.ai/blog/what-are-neural-networks-and-what-are-their-applications/

Linear models

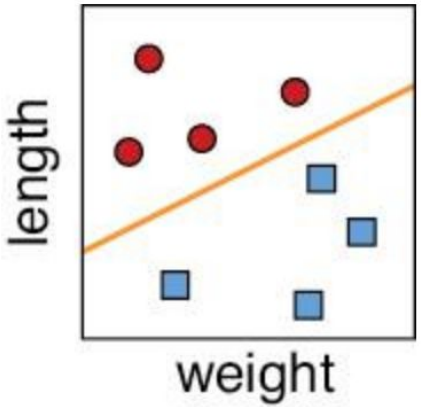
A **Fully Connected Layer** is a layer in which each neuron receives input from every neuron in the previous layer.



Linear models

When to use linear models vs. when not to use linear models

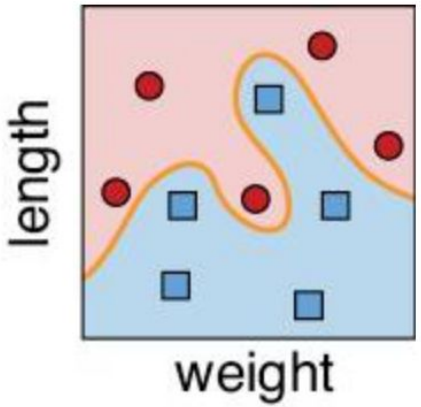
Linear models assume that the underlying relationship between features and outputs is linear), and are a good choice where that assumption is roughly true!



Linear models

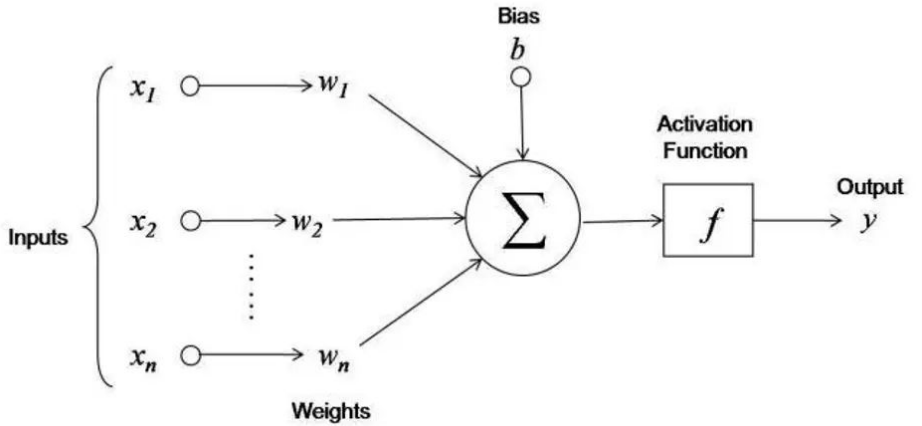
When to use linear models vs. when not to use linear models

They are a bad choice when relationships are non-linear!



Activation function

When relationships are complex and nonlinear, we apply **activation function**s to neurons (note: not every activation function is non-linear)



https://www.superannotate.com/blog/activation-functions-in-neural-networks

Activation function

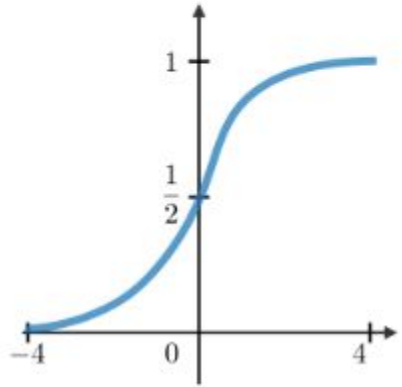
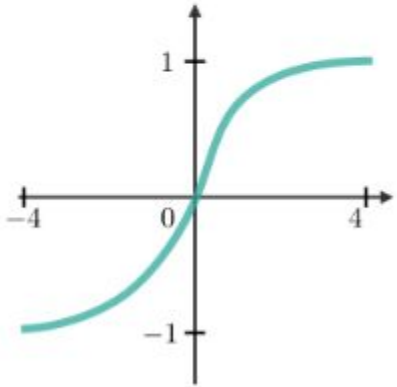
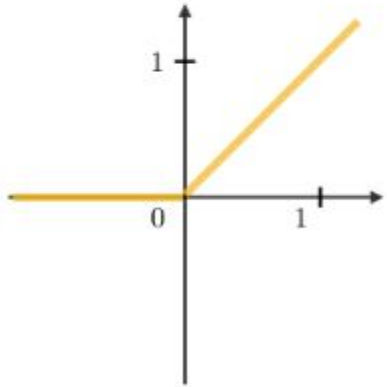
Mathematical notation

$$f(y) = f(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b)$$

$$f(y) = f(\sum_{i=1}^{n} w_i x_i + b)$$

https://www.superannotate.com/blog/activation-functions-in-neural-networks

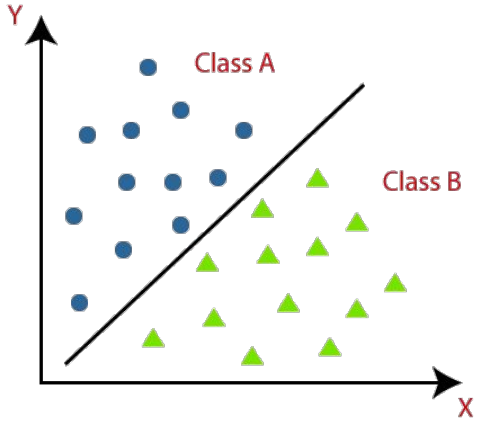
Some common activation functions

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$g(z) = \max(0, z)$
		

https://studymachinelearning.com/activation-functions-in-neural-network/

Classification: Recap

- **What is classification?** Models that sort things into categories
- **Real world examples**
  - Mood classification for playlists (Happy? Sad?)
  - Instrument recognition in mixed audio (Guitar? Piano?)
  - Music recommendation systems



https://www.javatpoint.com/classification-algorithm-in-machine-learning

Classification

Binary Classification



Multiclass Classification



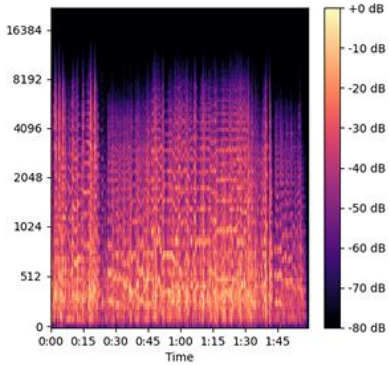
Multilabel Classification

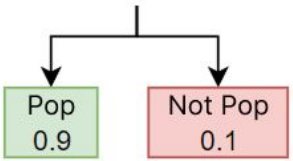


https://www.javatpoint.com/classification-algorithm-in-machine-learning

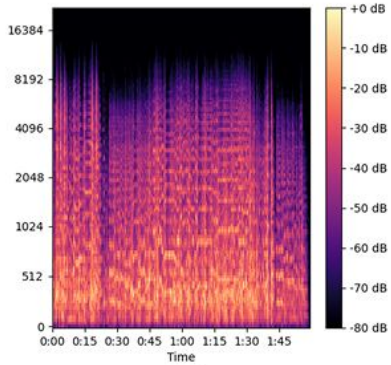
Classification

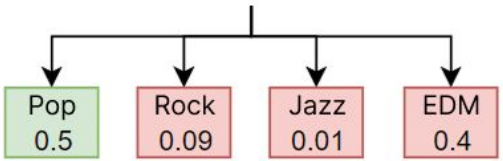
Binary Classification



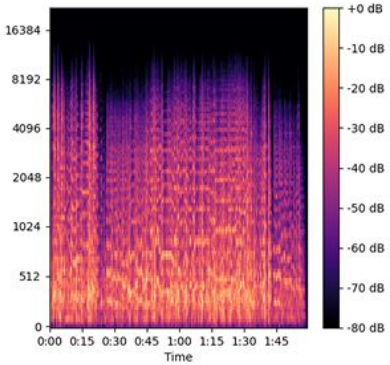


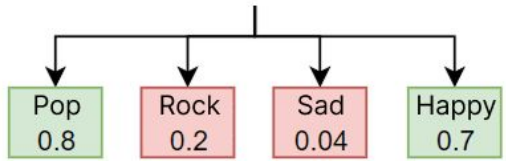
Multiclass Classification





Multilabel Classification





edited from https://www.mathworks.com/help/deeplearning/ug/multilabel-image-classification-using-deep-learning.html

Classification

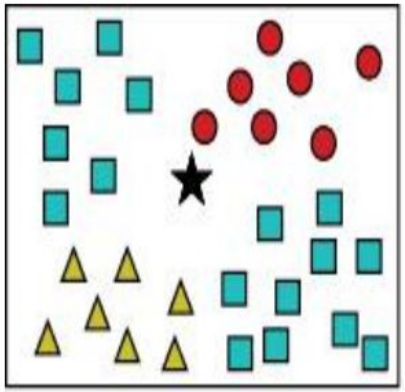
Types of classifier

- Logistic Regression (despite the name, it's for classification!)
- Decision Trees
- Support Vector Machines (SVM)
- K-Nearest Neighbors (KNN)
- Random Forests
- ... and many more!

Classification

K-Nearest Neighbors (KNN)

Finding the k closest samples to the new sample

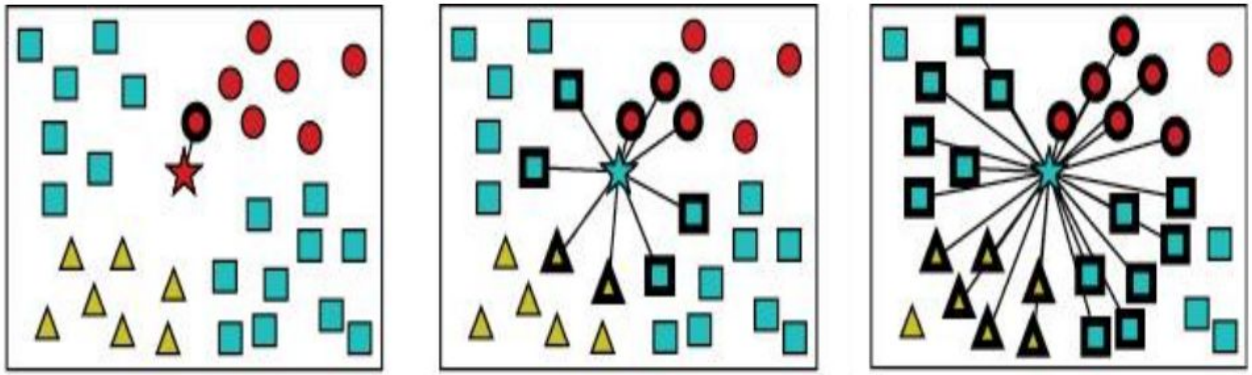


How can we determine a class for the star among three classes (square, circle, triangle).

Classification

K-Nearest Neighbors (KNN)

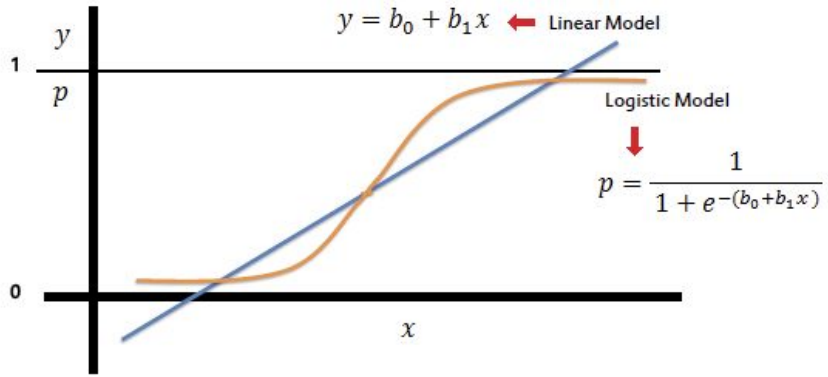
When k is 1 we only use the nearest sample to determine the class, but can have k > 1



Classification

Logistic Regression

Like a linear model but with a special activation function

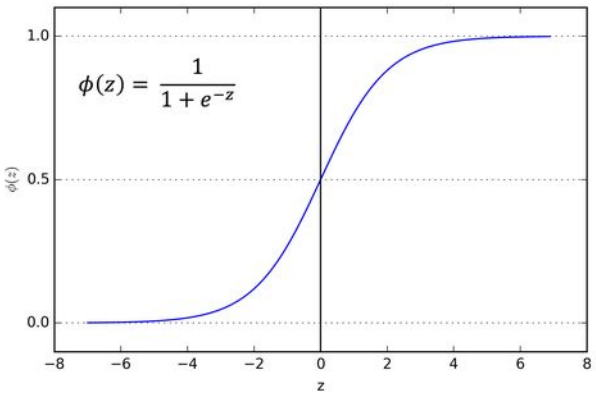


https://www.saedsayad.com/logistic_regression.htm

Classification

Logistic Regression

1. Linear model: y = wx + b
2. Problem: Linear output can be any number (-∞ to +∞)
3. But classification needs probability (0 to 1)
4. Solution: Add **sigmoid activation function**
5. Logistic Formula: ŷ = σ(wx + b) where σ(z) = 1/(1 + e^(-z))



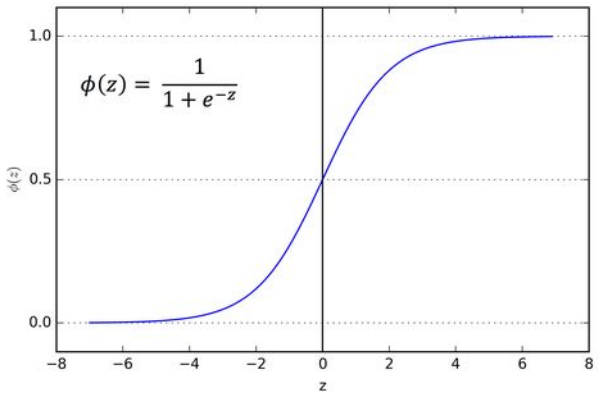
Sigmoid activation function

https://www.linkedin.com/pulse/understanding-sigmoid-function-logistic-regression-piduguralla

Classification

Sigmoid Function

- Takes any input and squishes it between 0 and 1
- Properties:
  - Output is always between 0 and 1
  - Steepest at middle (z = 0)
  - Flattens at extremes



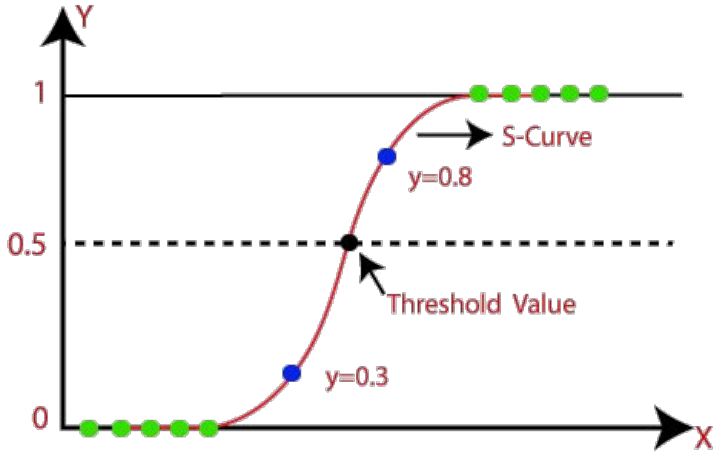
Sigmoid activation function

https://www.linkedin.com/pulse/understanding-sigmoid-function-logistic-regression-piduguralla

Classification

Logistic Regression

- When the output is larger than (or equal to) the threshold, label it as True (1)
- When the output is smaller than the threshold value, label it as False (0)



Classification

Logistic Regression Example

Features (x):

- x_1: Average beats per minute (BPM)
- x_2: Frequency of distorted guitar sounds
- x_3: Presence of orchestral instruments

$$\sigma(z) = \sigma(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b)$$

Weights (w_1, w_2, w_3) learned from data
Output: Probability that a song is rock music $\hat{y} = \sigma(z)$

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

If $\sigma(z) \geq 0.5$: Classify as rock

If $\sigma(z) < 0.5$: Classify as classical

Classification

Loss Function

A (binary) logistic regressor uses a (binary) **Cross Entropy Loss**:

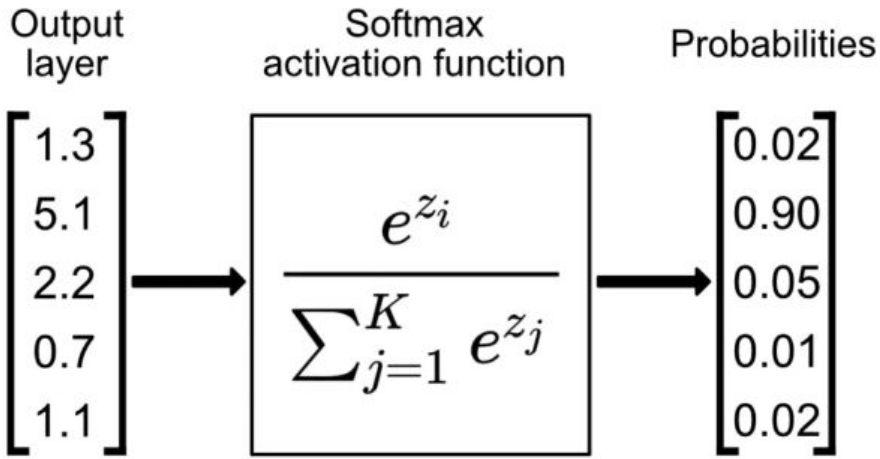
$$L = -(y * log(\hat{y}) + (1 - y) * log(1 - \hat{y}))$$

where *y* is the label (1 when True 0 when False) and ŷ is the model's predicted probability

Classification

Multiclass Logistic Regression

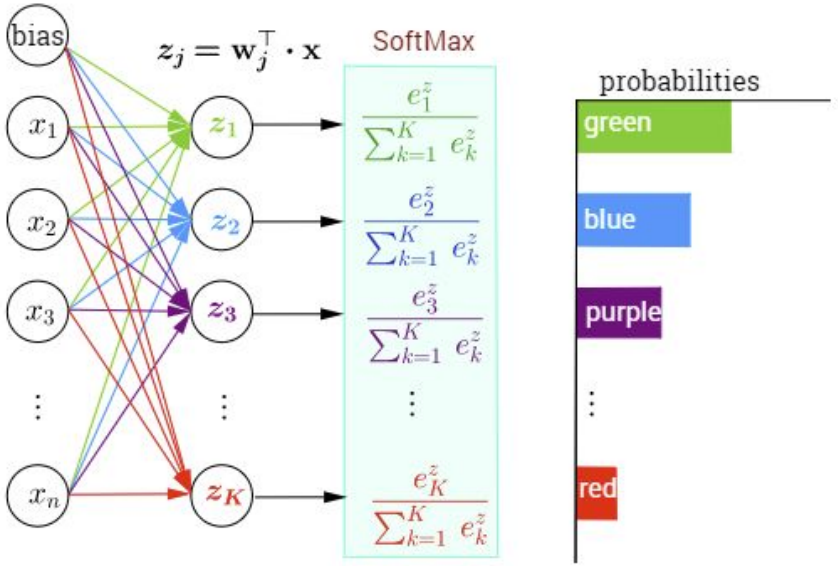
Similar to binary logistic regression, but we apply the **softmax function** to ensure that the sum of probabilities equals 1



https://www.singlestore.com/blog/a-guide-to-softmax-activation-function/

Classification

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_K \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \\ \mathbf{w}_3^\top \\ \vdots \\ \mathbf{w}_K^\top \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$



https://rinterested.github.io/statistics/softmax.html

Classification

Support Vector Machines (SVM)

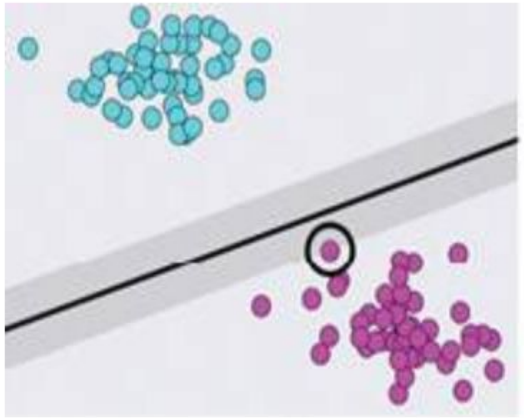
Let’s say we have samples consisting of two classes; how can we set a boundary line that best separates the classes?

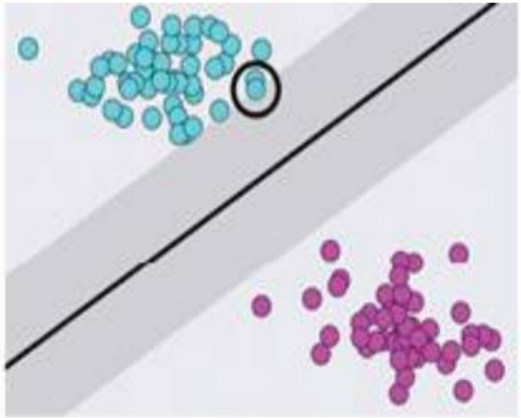


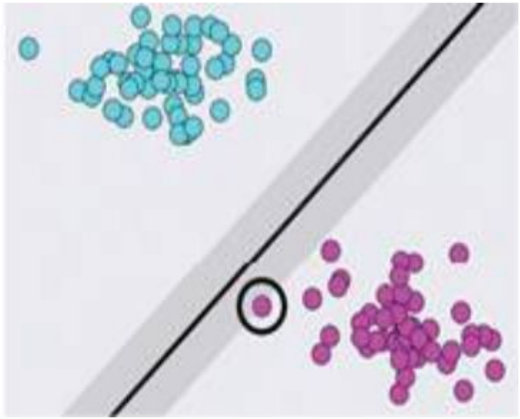
Classification

Support Vector Machines (SVM)

Let’s say we have samples consisting of two classes; how can we set a boundary line that best separates the classes? (a circle around the sample that’s closest to the line)





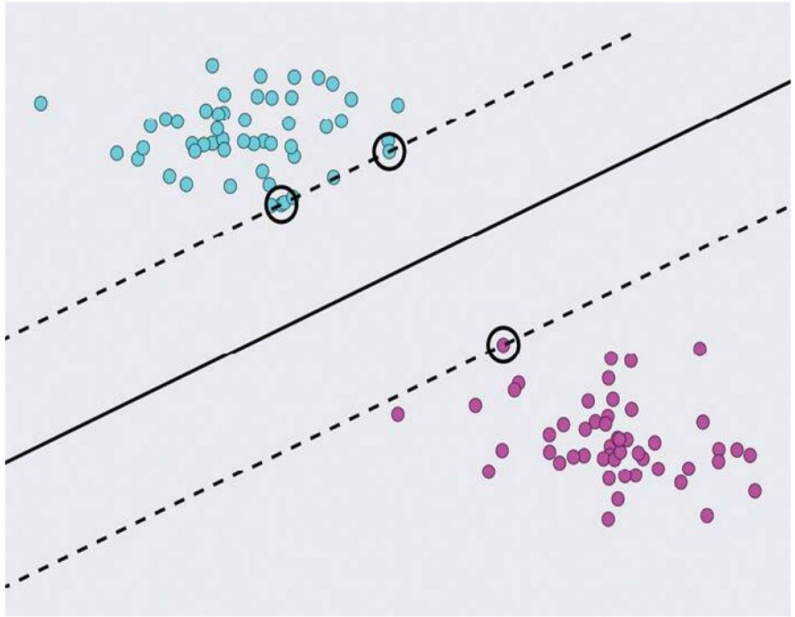


Classification

Support Vector Machines (SVM)

An algorithm that finds the line that has the greatest distance from samples

(really just here to show you that there's not just one way to build a classifier)



Evaluating classification models

Accuracy:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} = \frac{TP+TN}{TP+TN+FP+FN}$$

Error Rate: 1- Accuracy

Evaluating classification models

Accuracy is not enough

Imagine we're building a model to detect a rare disease that only affects 1% of the population:

```
Total_patients = 10000
Healthy = 9900 (99%)
Sick = 100 (1%)

Model prediction: "Healthy" for ALL patients
Accuracy = 9900/10000 = 99%
```

Despite having 99% accuracy, this model is completely useless because:

1. It fails to detect ANY sick patients
2. missing sick patients (false negative) can be life-threatening

True and false positives (and negatives)

TP, TN, FP, FN

- True: the system was correct in its prediction
- False: the system was wrong in its prediction.
- Positive: it recommended the song.
- Negative: it didn't recommend the song.

True and false positives (and negatives)

TP, TN, FP, FN

- **True Positive**: When the system recommends a song and you actually like it. For example, it suggests a new jazz track, and you end up enjoying it.
- **True Negative**: When the system correctly doesn't recommend a song you wouldn't like. For instance, it knows you dislike heavy metal and doesn't suggest those songs to you.
- **False Positive**: When the system recommends a song but you don't like it. Like when it suggests a country song thinking you'll enjoy it, but you actually don't.
- **False Negative**: When the system fails to recommend a song you would have liked. For example, it doesn't suggest a pop song that you would have loved because it incorrectly assumed you wouldn't enjoy it.

Balanced error rate

Balanced Error Rate (BER)

a metric specifically designed for imbalanced datasets that gives equal importance to errors in each class, regardless of their size

Formula

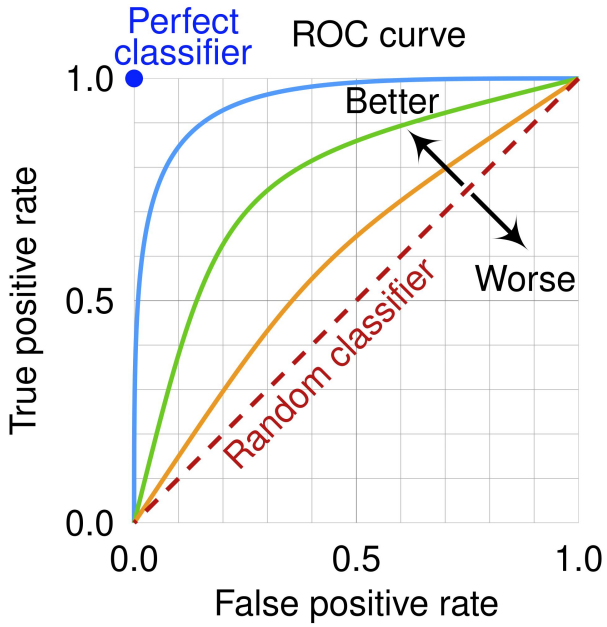
$$\begin{aligned} \mathrm{BER} &= 0.5 \times (\mathrm{FPR} + \mathrm{FNR}) \\ &= 0.5 \times \left(\frac{\mathrm{FP}}{\mathrm{FP}+\mathrm{TN}} + \frac{\mathrm{FN}}{\mathrm{FN}+\mathrm{TP}} \right) \\ &= 1 - 0.5 \times (\mathrm{TPR} + \mathrm{TNR}) \end{aligned}$$

Potentially desirable because any “trivial” model (e.g. always predict true, always predict false; predict at random) will have BER of 0.5; only “useful” models have lower BER

Other metrics

ROC Curve

- **X-axis:** False Positive Rate (FPR)
  - **Formula:** FPR = FP / (FP + TN)
- **Y-axis:** True Positive Rate (TPR)
  - **Formula:** TPR = TP / (TP + FN)



Other metrics

Precision / recall / F1 / etc.

These don’t show up much in this class, so I won’t spend time covering them.

For self study see https://cseweb.ucsd.edu/~jmcauley/pml/pml_book.pdf Section 3.3.3

Generalization, overfitting and underfitting

Need to address two issues when training a model:

1. We should not evaluate a model on the same data that was used to train it. Rather we should use a held-out dataset (i.e., a test set).
2. Features that improve performance on the training data will not necessarily improve performance on the held-out data.

Evaluating a model on held-out data gives us a sense of how well we can expect that model to work ‘in the wild.’

This held-out data, known as a **test set,** measures how well our model can be expected to **generalize** to new data.

Generalization, overfitting and underfitting

Overfitting occurs when our model works well on our training data, but does not generalize well to held-out (test) data

Underfitting occurs when our model is insufficiently complex to capture the underlying dynamics in a dataset (meaning that both its training error and test error are high)

Generalization, overfitting and underfitting

Q: But how can we **detect** overfitting or underfitting?

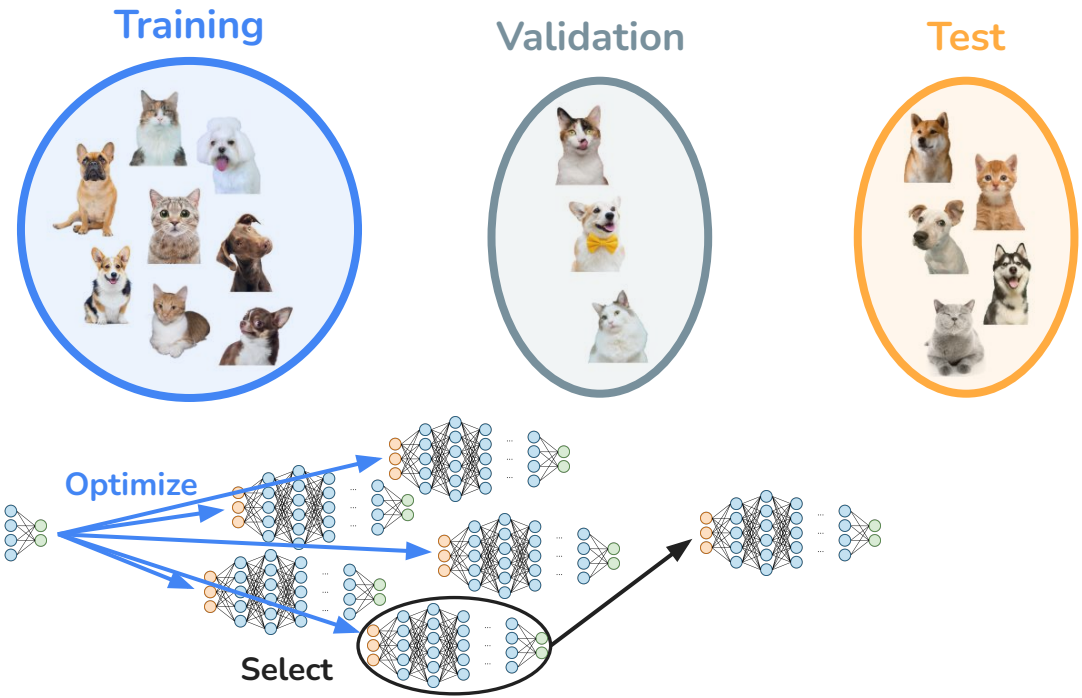
- (Just about) *all* models will work better on seen data compared unseen data, so *having worse performance on the training set doesn't necessarily mean we've overfit* (i.e., there may be no model that generalizes better)
- Likewise, poor performance on the training set doesn't necessarily mean we've underfit: possibly *no model is capable of fitting the data* (based on the available features)

A: We'll need to try several models of varying complexity to find which one generalizes best

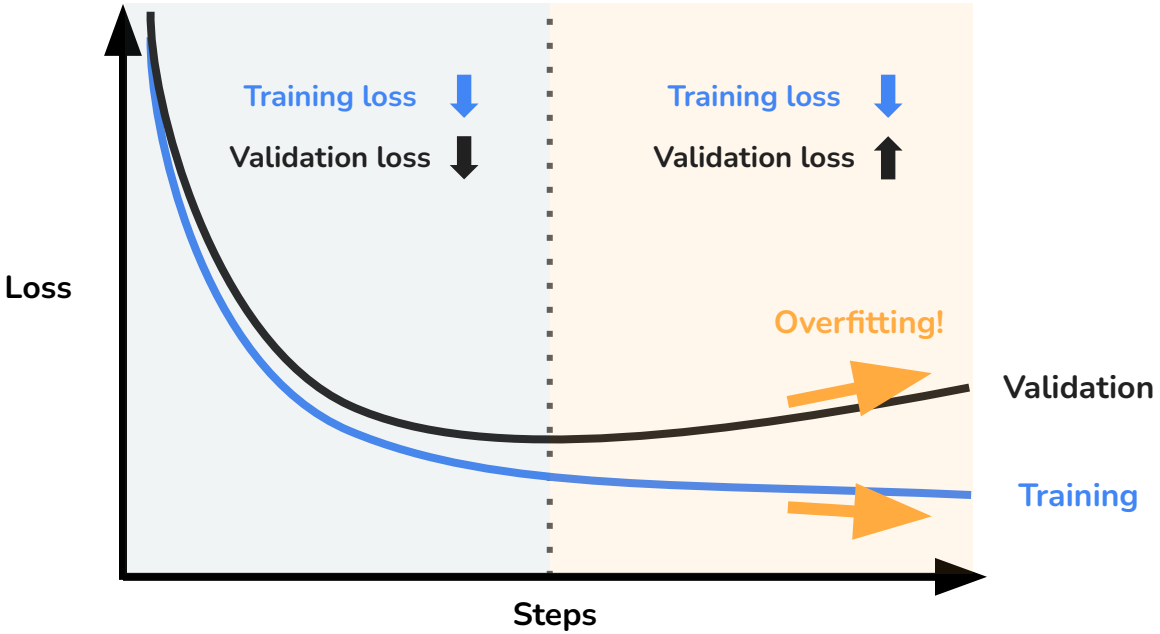
The learning pipeline

A validation set is “like” a test set but can be probed multiple times to select our favorite model

(whereas the test set should in theory be seen only once and not used to select models)



The learning pipeline



Some “deep learning” stuff

We’ll cover the specific models that we really need to in their respective modules

Below are just a few building blocks that show up in different models

Pooling

Pooling is a *downsampling* operation that attempts to extract (or “summarize”) the most important information from a (learned) model layer

Max pooling

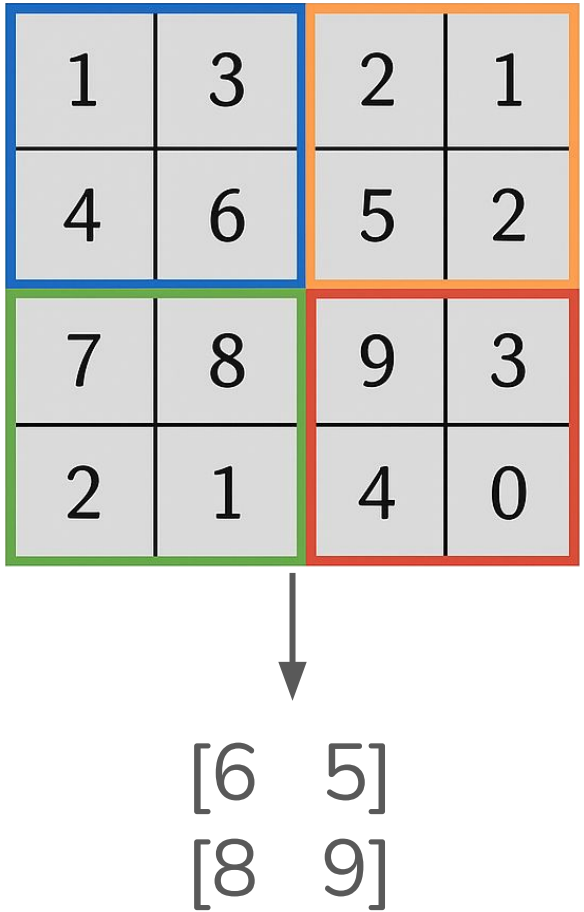
An input feature map $X \in R$; a size of $H \times W$; a pooling window size $k \times k$; a stride size s; (optional) padding p

Output size

$$H' = \left\lfloor \frac{H - k + 2p}{s} \right\rfloor + 1, \quad W' = \left\lfloor \frac{W - k + 2p}{s} \right\rfloor + 1$$

Max pooling:

$$Y_{i,j} = \max_{0 \le mk,\, 0 \le nk} X_{si+m,\, sj+n}$$



Mean pooling

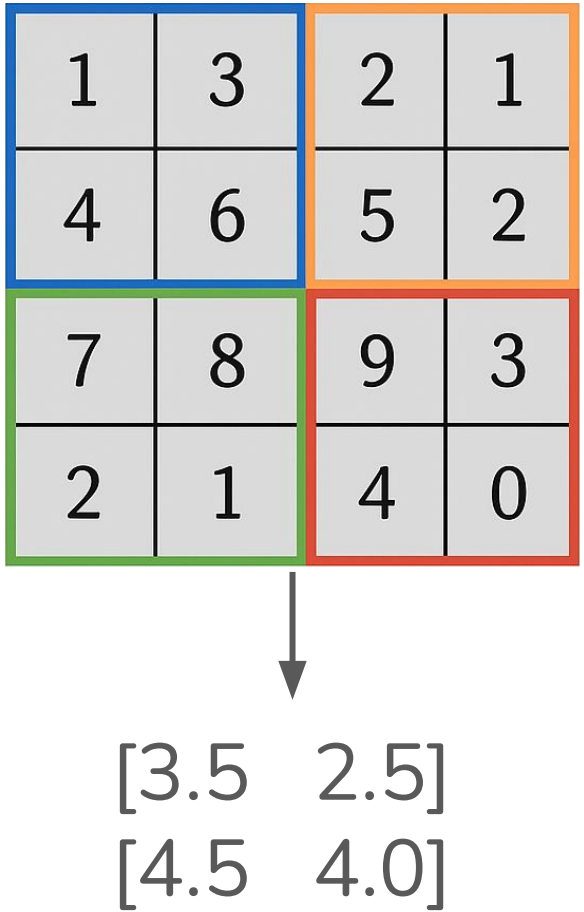
An input feature map $X \in R$; a size of $H \times W$; a pooling window size $k \times k$; a stride size s; (optional) padding p

Output size

$$H' = \left\lfloor \frac{H - k + 2p}{s} \right\rfloor + 1, \quad W' = \left\lfloor \frac{W - k + 2p}{s} \right\rfloor + 1$$

Mean pooling:

$$Y_{i,j} = \frac{1}{k^2} \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} X_{si+m,\, sj+n}$$



ML fundamentals

Some basic operators (in torch)

`torch.nn.Linear`: Applies a linear transformation y=Wx+by.

`torch.nn.ReLU`: Applies the ReLU activation f(x)=max(0,x) adds non-linearity.

`torch.nn.Sigmoid`: Maps inputs to (0, 1) useful in binary classification.

`torch.nn.Tanh`: Maps inputs to (-1, 1) used in regression tasks.

`torch.nn.Dropout`: Randomly zeroes elements during training; regularization to prevent overfitting.

`torch.nn.BatchNorm1d` / `torch.nn.BatchNorm2d`: Normalizes input across the batch; stabilizes learning.

`torch.nn.Sequential`: Combines layers into a sequential block; simplifies model definition

ML fundamentals

Classification Operators

`torch.nn.CrossEntropyLoss`: Combines `LogSoftmax` and `NLLLoss` used for multi-class classification.

`torch.nn.Softmax`: Converts logits to probabilities used for output interpretation.

Regression Operators

`torch.nn.MSELoss`: Mean Squared Error penalizes squared difference between prediction and target.

`torch.nn.L1Loss`: Mean Absolute Error penalizes absolute differences.

Optimization Operators

`torch.optim.SGD`: Stochastic Gradient Descent optimizer.

`torch.optim.Adam`: Popular optimizer with adaptive learning rates.

`optimizer.step()`: Updates parameters using computed gradients.

`optimizer.zero_grad()`: Clears old gradients before backpropagation.

`loss.backward()`: Computes gradients via backpropagation.

ML fundamentals

CNN Operators

`torch.nn.Conv2d`: Applies a 2D convolution over input images, core of CNNs.

`torch.nn.Conv1d`: Applies 1D convolution used for time series or sequential data.

`torch.nn.Conv3d`: Applies 3D convolution used for volumetric data (e.g., videos, medical scans).

`torch.nn.MaxPool2d`: Downsamples using max pooling; reduces spatial dimensions.

`torch.nn.AvgPool2d`: Downsamples using average pooling.

`torch.nn.Flatten`: Converts multi-dimensional tensor to a 1D vector used before fully connected layers.

ML fundamentals

(see some of the above operations “in context” in workbook2.ipynb)

Predictive pipelines for music & MIR

2.2: Input representations (& Fourier transforms)

The Fourier Transform in a nutshell

Mostly from **“Fundamentals of Music Processing” (Meinard Müller, International Audio Laboratories Erlangen):**

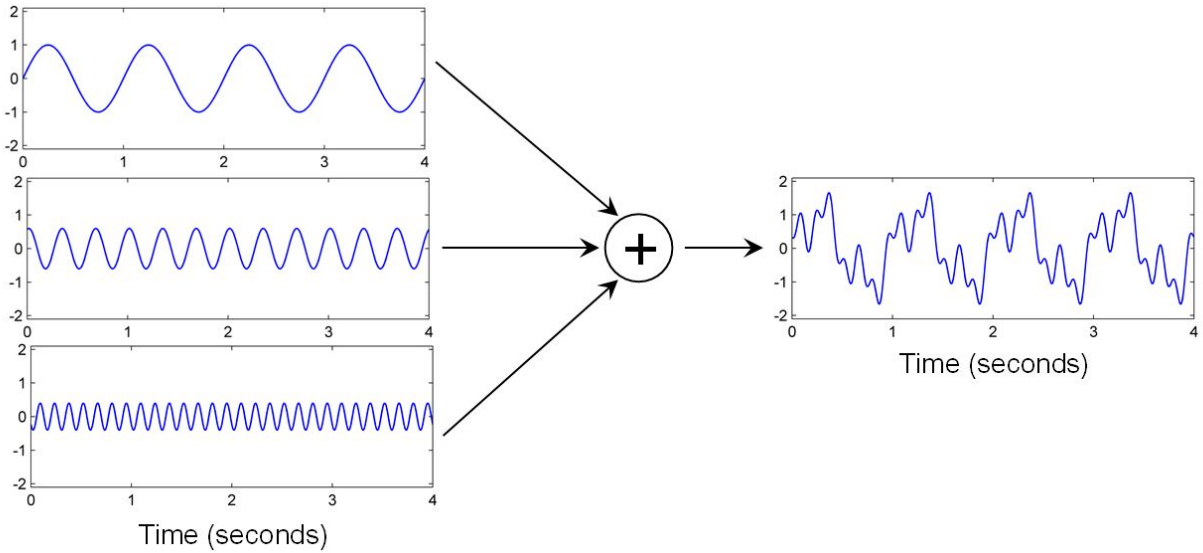
https://www.audiolabs-erlangen.de/resources/MIR/FMP/C0/C0.html

and **“Music Classification: Beyond Supervised Learning, Towards Real-world Applications” (Minz Won, Janne Spijkervet, Keunwoo Choi),** specifically:

https://music-classification.github.io/tutorial/part2_basics/input-representations.html

What are we trying to do?

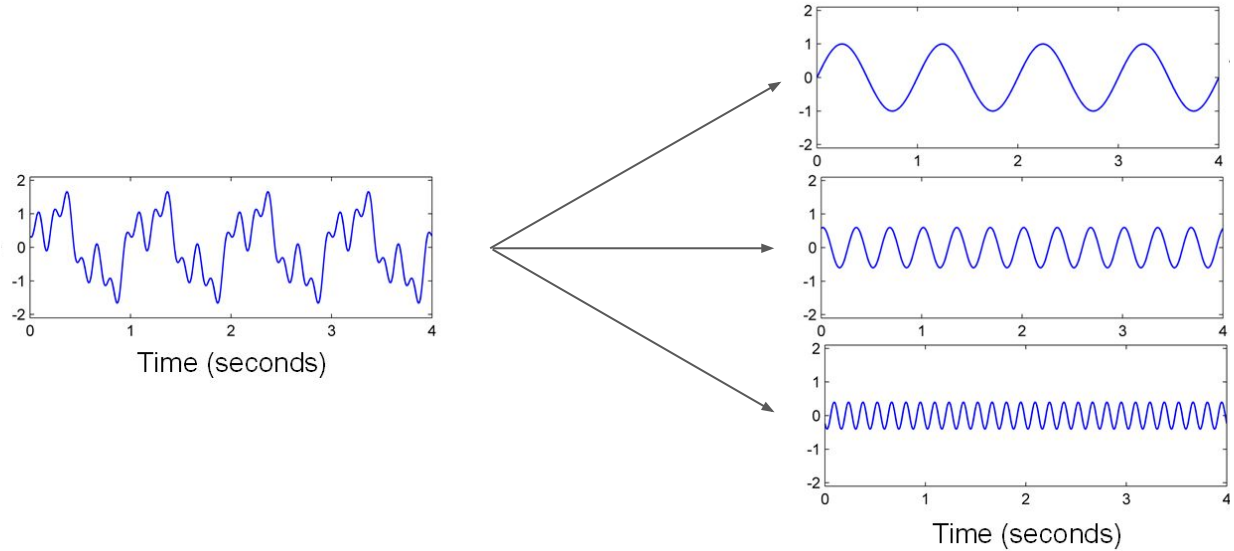
Recall (Module 1): audio waveforms can be composed by summing several wave functions



From: Meinard Muller: Fundamentals of Music Processing

What are we trying to do?

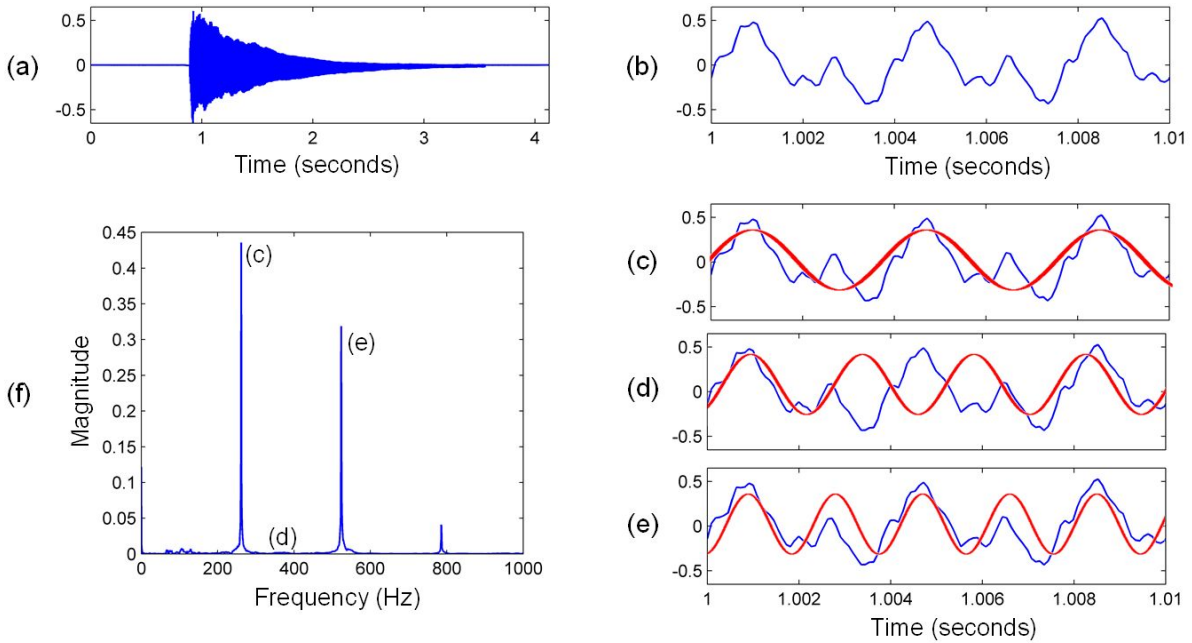
Recall (Module 1): can we reverse this process, to understand what simple waveforms generated a signal?



From: Meinard Muller: Fundamentals of Music Processing

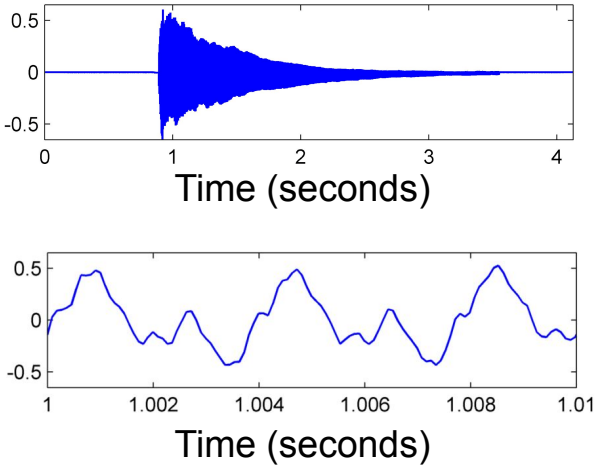
What are we trying to do?

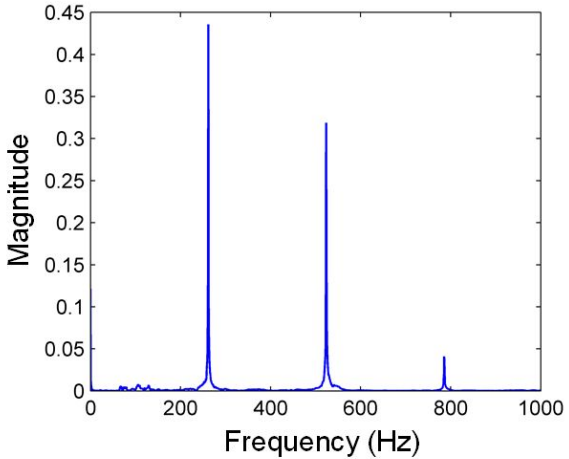
(a) waveform
(b) zoomed in on some location
(c-e) *approximately*, and *locally* sum to b
(f) frequency magnitude representation: (roughly) at what magnitude is each frequency represented in (b)?



From: Meinard Muller: Fundamentals of Music Processing

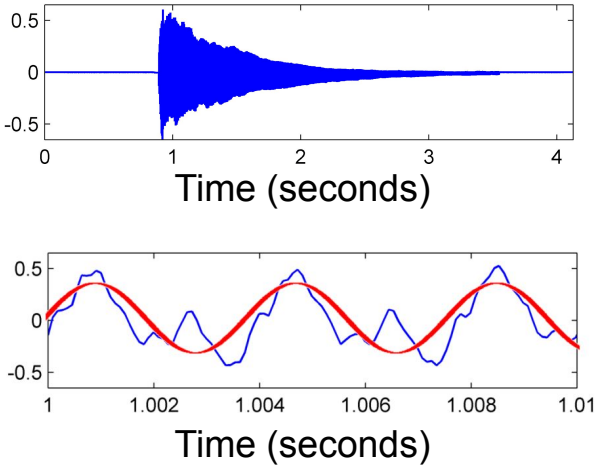
What are we trying to do?

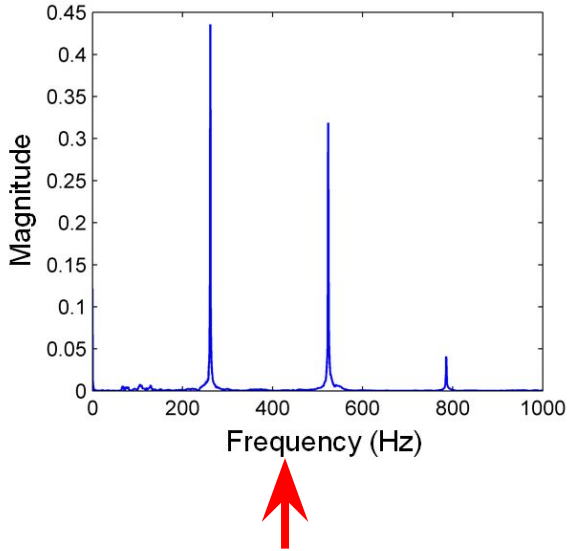




From: Meinard Muller: Fundamentals of Music Processing

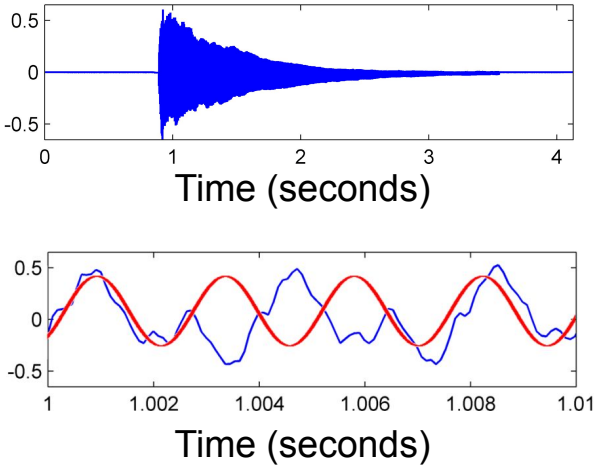
What are we trying to do?

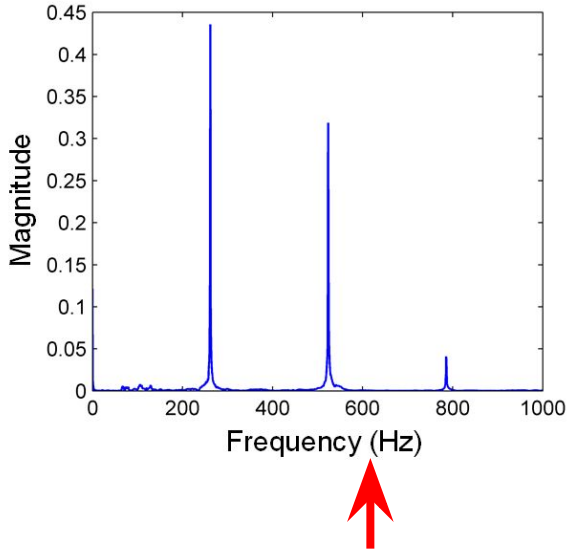




From: Meinard Muller: Fundamentals of Music Processing

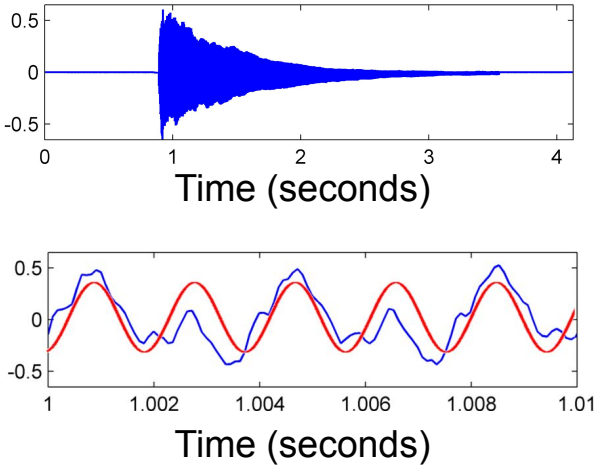
What are we trying to do?

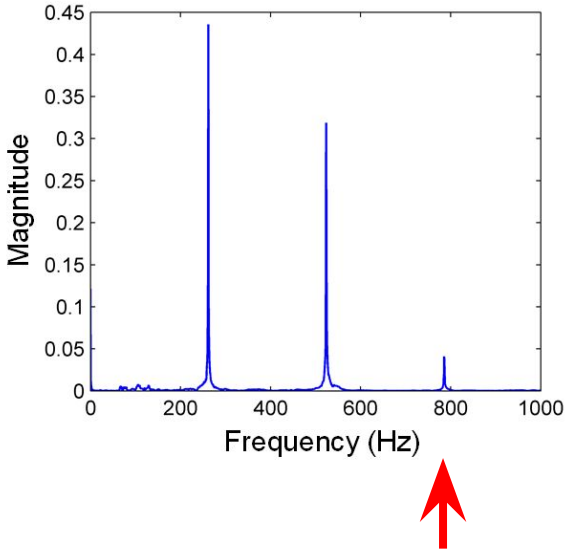




From: Meinard Muller: Fundamentals of Music Processing

What are we trying to do?





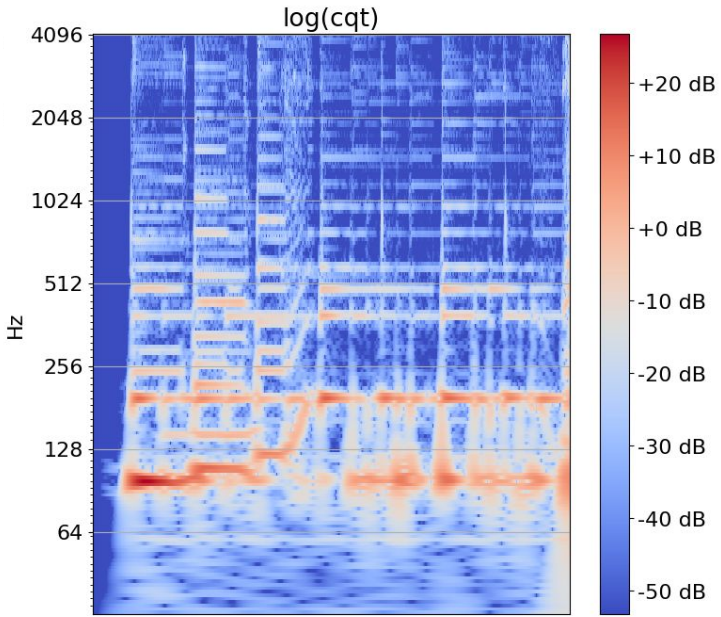
From: Meinard Muller: Fundamentals of Music Processing

What are we trying to do?

Above describes which frequencies are present *locally* (i.e., at a given point in time)

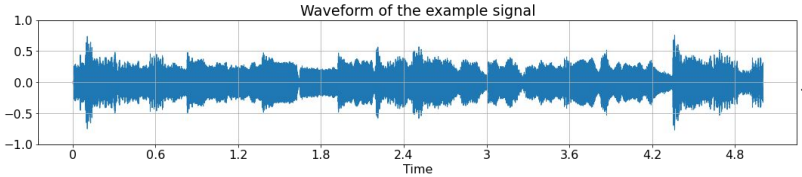
Following this, we'd like to see how frequencies *vary over time*

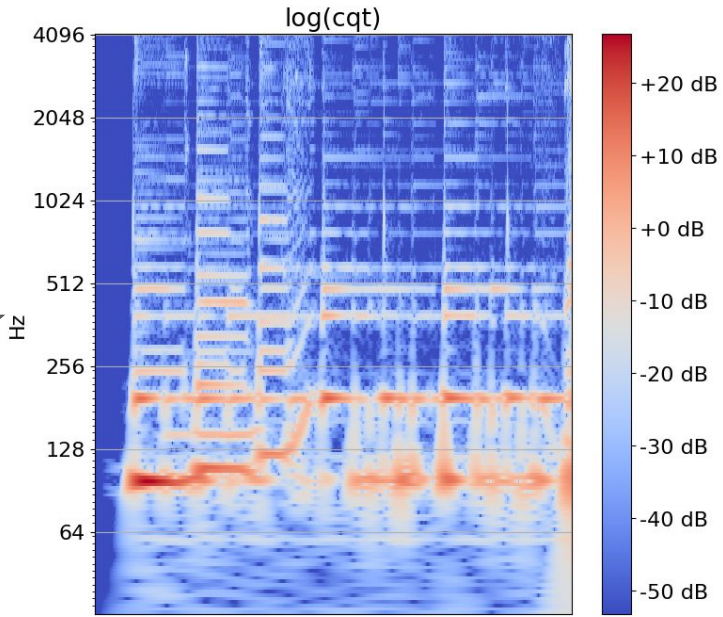
This gives us a *time-frequency* representation of the waveform (called a "spectrogram")



What are we trying to do?

Basically, we want to from a waveform (1d) to a time-frequency (2d) representation





Code walkthrough: Input representations

This rest of this section (slides and code) is based on: https://music-classification.github.io/tutorial/part2_basics/input-representations.html

(note, the code from the above link requires a few small modifications to work properly, see working version in **workbook2.ipynb**)

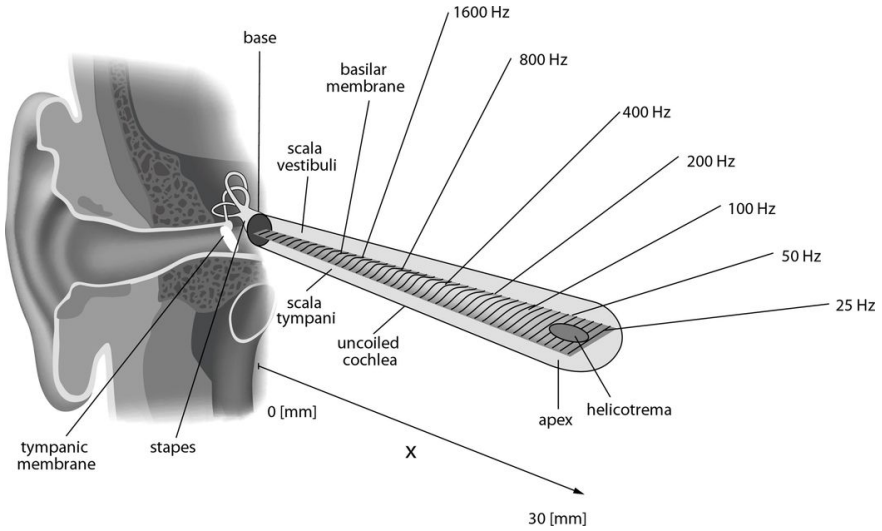
Input representations: waveforms

What do we want from an audio representation?

- (Reasonably) compact
- (Depending on the method) fixed-dimension (e.g. CNN image classifiers generally operate over images of fixed scale)
- Biologically plausible (?)

Input representations

“Biologically plausible” means that the representation should have some relationship to how the human ear receives (or maybe how the brain processes) audio (we’ll come back to this later)



(from wikipedia)

Input representations: waveforms

Should it be biologically plausible?

- Probably! We generally care about tasks related to human perception of sound, e.g. beat detection, classification of genres; put differently, we are solving tasks that are “natural” for humans, so we should seek a representation that corresponds closely to human biology
- **Q:** Can you think of any (music or audio) tasks that are “easy” for machines but hard for humans? Would a different representation be suitable?

In any case, we’ll focus on “biologically plausible” representations in the following code

Input representations: waveforms

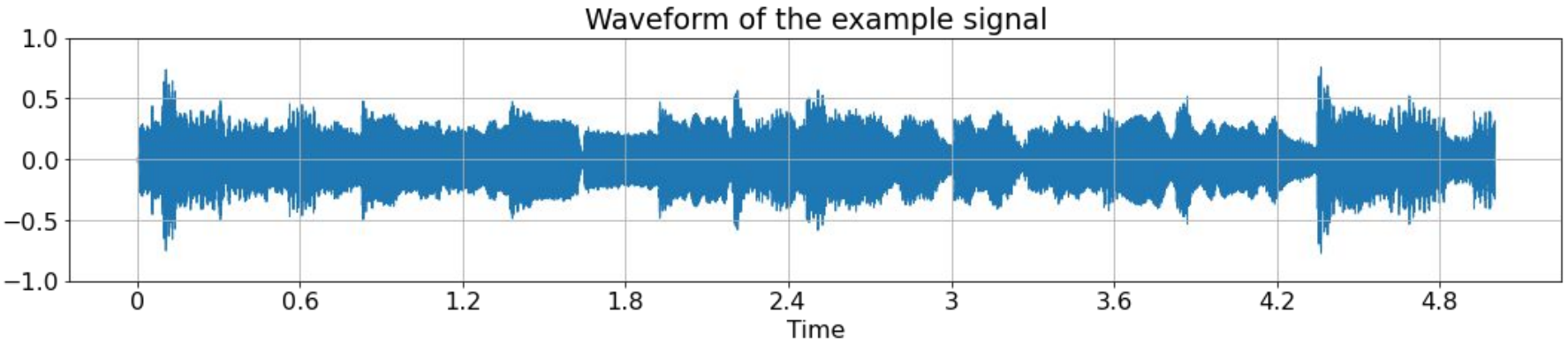
Step 1: Load a waveform (see workbook2.ipynb code)

A few libraries:

```
import matplotlib.pyplot as plt # make nice plots
import librosa                  # music and audio analysis
import IPython.display as ipd   # play audio in notebook
```

Input representations: waveforms

Step 1: Load a waveform



Input representations: waveforms

Step 1: Load a waveform

- This is 5-seconds of audio
- The shape is a 110,250-dimensional vector
- I.e., we have sampled (mono) audio at a rate of 22,050 samples per second (i.e., 44.1kHz for stereo)

Input representations: waveforms

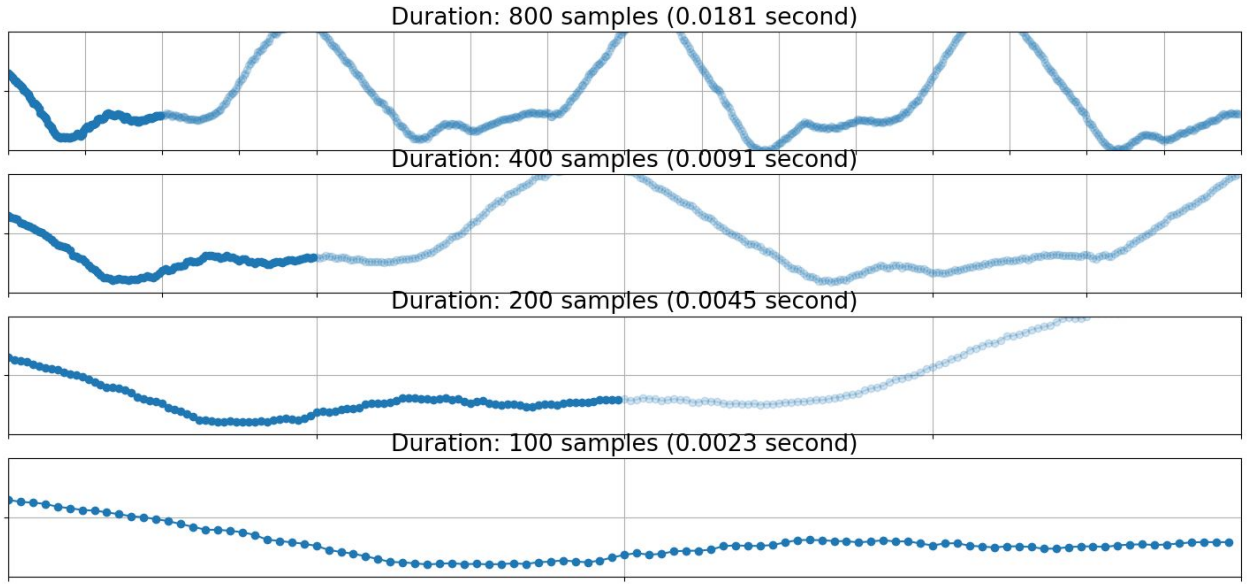
Step 1: Load a waveform

- This is 5-seconds of audio
- The shape is a 110,250-dimensional vector
- I.e., we have sampled (mono) audio at a rate of 22,050 samples per second (i.e., 44.1kHz for stereo)
- This “raw” representation is intended for audio devices (the diaphragms of speakers) whose goal is to produce the sound (the change of pressure in the air) correctly

Input representations: waveforms

Step 1: Load a waveform

- Zoom into the waveform:



Input representations: waveforms

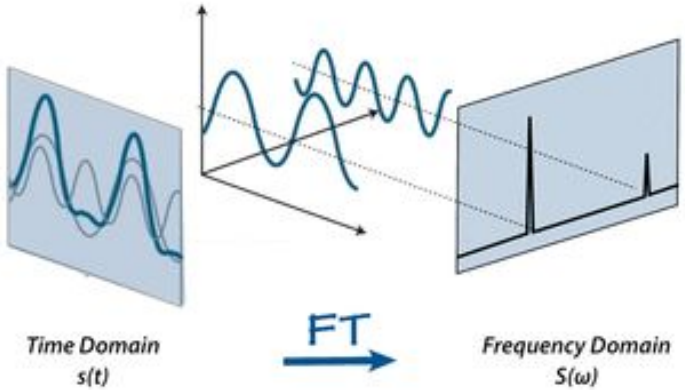
So, we can represent our data as a raw waveform; is this likely to be useful for downstream approaches?

- We might see applications later on that use waveforms as inputs and/or outputs, but for e.g. a classification task, do these fit our desired criteria (compact, fixed dimensional, biologically plausible, etc.)?

Fourier Transform

Fourier Transform (FT): A mathematical tool that breaks down any signal into its constituent frequencies: like splitting white light into a rainbow through a prism

- Takes a signal in time domain (waveform) → converts it to frequency domain (spectrum)
- **Core idea:** Any complex signal can be represented as a sum of simple sine waves



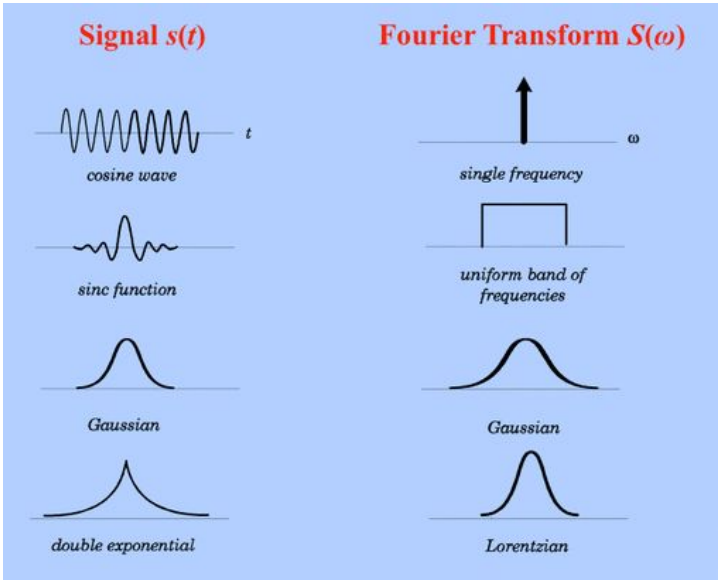
https://mriquestions.com/fourier-transform-ft.html

Fourier Transform

$$S(\omega) = \int_{-\infty}^{\infty} s(t) e^{-i\omega t} dt$$

Fourier Transform (FT): A mathematical tool that breaks down any signal into its constituent frequencies: like splitting white light into a rainbow through a prism

- Takes a signal in time domain (waveform) → converts it to frequency domain (spectrum)
- **Core idea:** Any complex signal can be represented as a sum of simple sine waves



https://mriquestions.com/fourier-transform-ft.html

Fourier Transform

So what do we still need to do?

1. We don't have continuous signals! We have (quantized) ***discrete*** observations: we need the *Discrete* Fourier Transform **(DFT)**
2. We don't actually want to describe an entire waveform in terms of its frequency components: we want to describe it ***locally,*** i.e., to measure what frequency components are present at a particular point in time **(STFT)**
3. (we also need to compute these things efficiently, but I won't go into that much) **(FFT)**

https://mriquestions.com/fourier-transform-ft.html

Fourier Transform

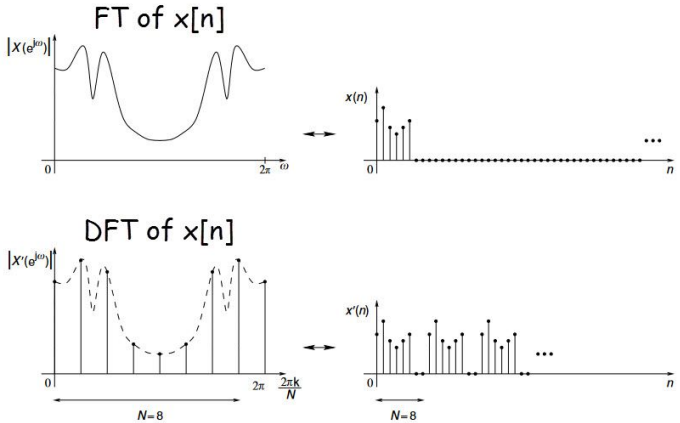
I'll cover each of the above at a pretty high-level, and mostly focus on code from here on out

(follow along in workbook2.ipynb)

Discrete Fourier Transform (DFT)

Discrete Fourier Transform

- Fourier Transform that operates on discrete signals (rather than continuous signals)



$$S(\omega) = \int_{-\infty}^{\infty} s(t)e^{-i\omega t}dt$$ (previous slide)

vs

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{(\frac{-j2\pi}{N}nk)}$$

for $0 \leq k \leq N-1$

https://www.ee-diary.com/2023/05/what-is-discrete-fourier-transformdft.html

Fast Fourier Transform (FFT)

Computing $X[k] = \sum_{n=0}^{N-1} x[n] e^{(\frac{-j2\pi}{N} nk)}$

$\text{for } 0 \leq k \leq N-1$

requires O(N^2) operations (there are N “outputs” *X*, and N “inputs” *x*)

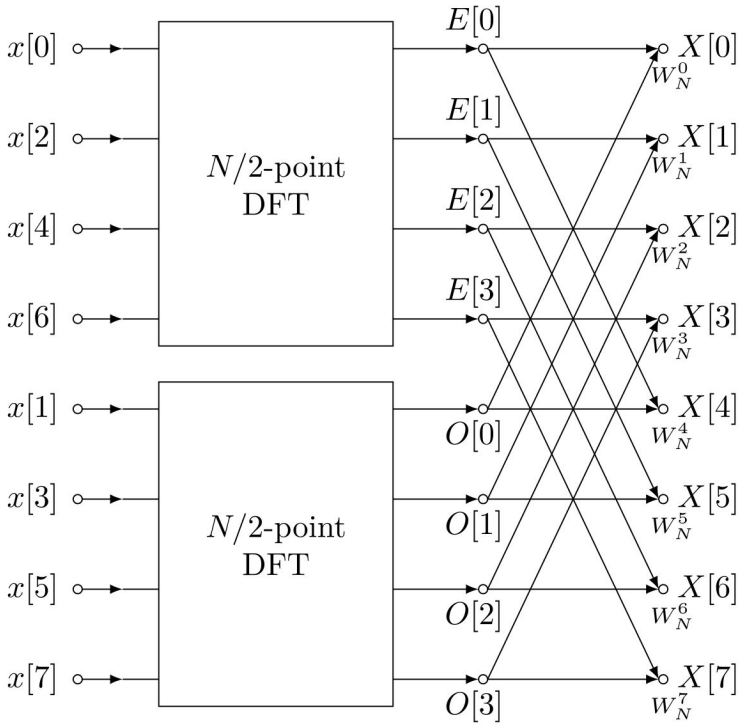
But! e^(-j2πnk/N) is periodic; this means that many calculations in DFT are redundant

The **Fast Fourier Transform (FFT)** reuses these calculations

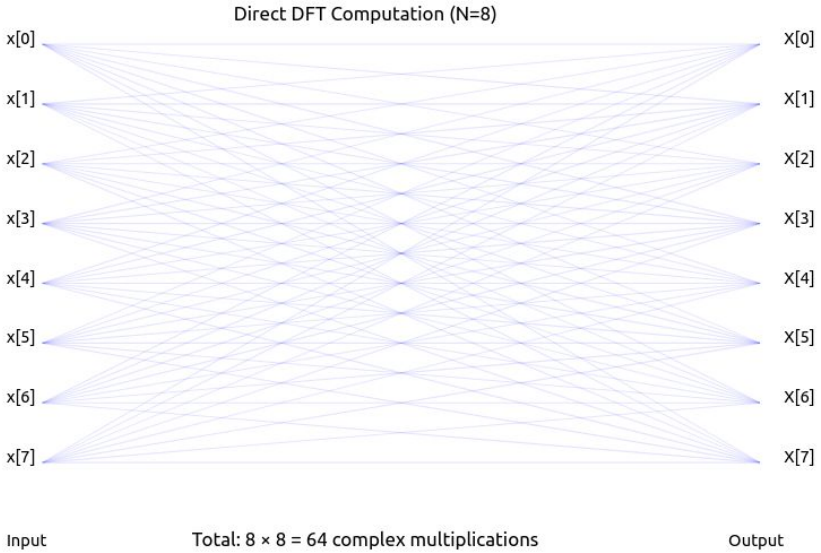
Fast Fourier Transform (FFT)

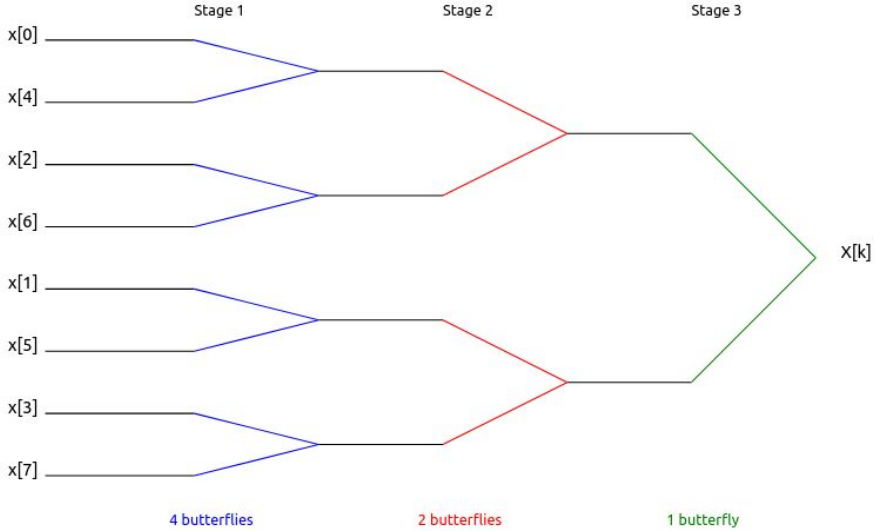
Fast Fourier Transform

- Just an efficient algorithm to compute DFT
- Gives exactly the same results as Discrete Fourier Transform
- “Divide-and-conquer”-type approach: break the problem down into two subproblems of size N/2
- Final complexity is O(NlogN) (rather than O(N^2))



Direct DFT vs FFT Computation





Direct DFT O(N^2)

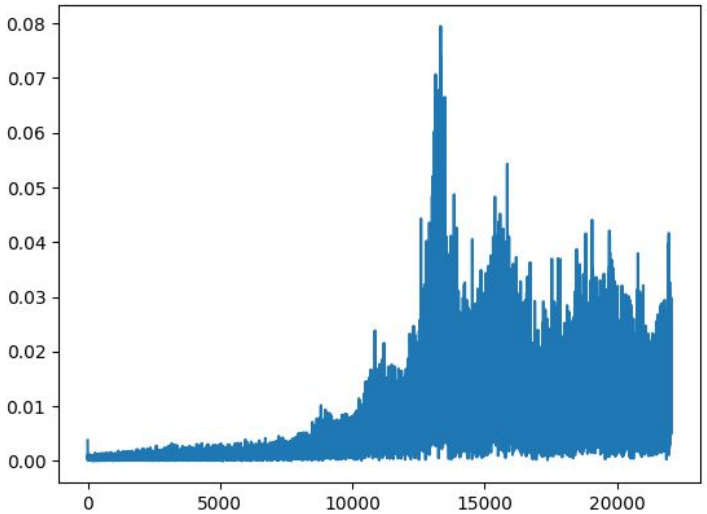
FFT O(NlogN)

Example Code: Loading audio and showing spectrum

```
from scipy.fft import fft, fftfreq#import library for signal processing
y, sr = librosa.load("sample.wav", sr=44100)#Load audio (sr is sampling rate)
yf = fft(y)#Perform FFT
yf = yf[0:sr//2]#Select first half
yf = np.abs(yf)#Take magnitude of complex FFT values
yf = 2.0/sr * yf#Compensate for using only half of the spectrum and scale
xf = fftfreq(sr, 1/sr)[:sr//2]#Create x-axis(frequency) values for the plot
plt.plot(xf, yf)#Plot
plt.savefig("spectrum.png")#Save
plt.close()#Finish plotting
```

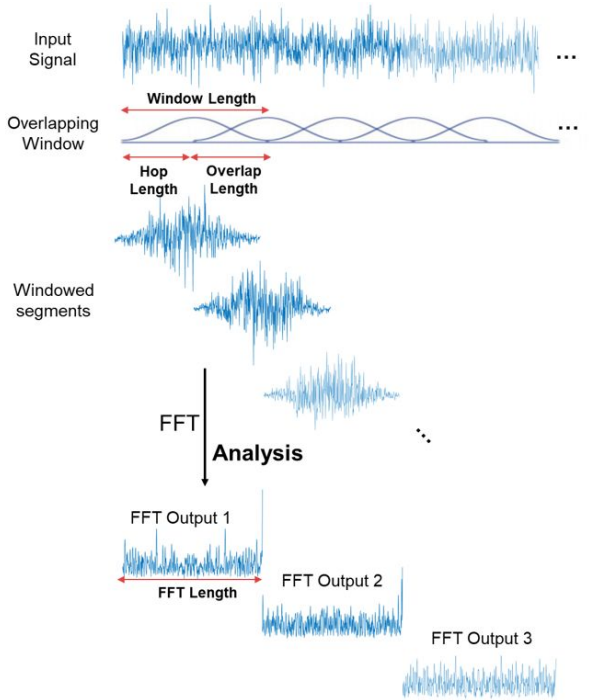
Loading audio and showing spectrum

Example Output:



STFT (Short-time Fourier Transform)

- **Next issue:** The Fourier Transform (FT) doesn't *localize* temporal events
- Short-Time Fourier Transform (STFT)
  - Time-frequency representation of signals
  - Slide short windows over the input signal and then compute the FFT **locally** for each window

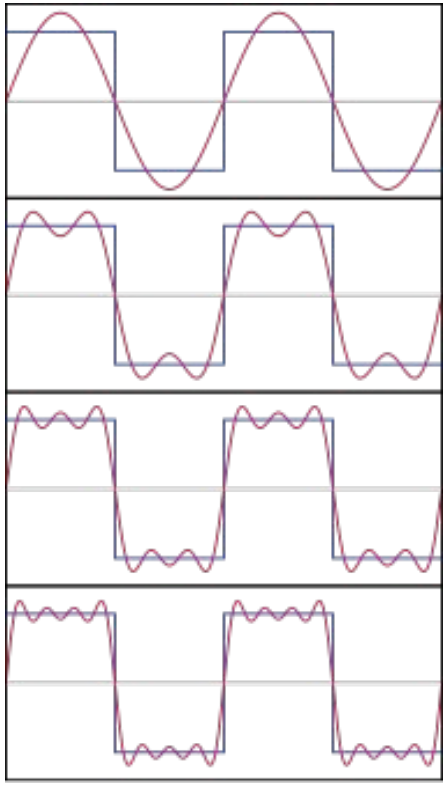


https://ww2.mathworks.cn/help/dsp/ref/dsp.stft.html

Input representations: STFT

We'll use the Short-time Fourier Transform (STFT) on our magnitude (waveform) representation

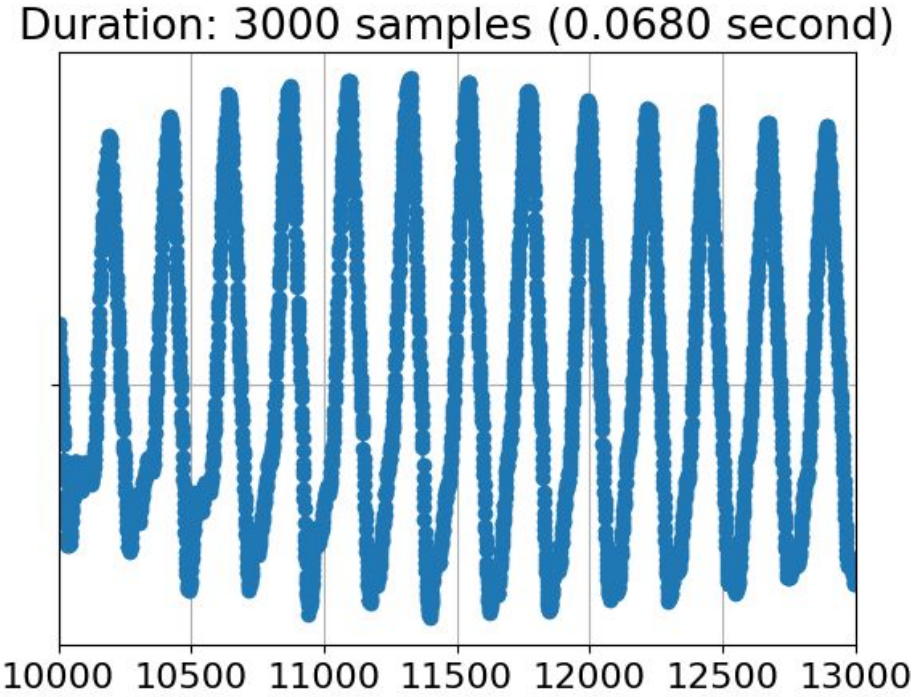
recall: the Fourier transform approximates a (periodic) signal using a series of trigonometric functions



from wikipedia: https://en.wikipedia.org/wiki/Fourier_series

Input representations: STFT

Our sequences are not periodic *globally*: they are periodic *locally;* by computing the Fourier transform on short segments, the coefficients tell us the frequencies and amplitudes associated with that segment



Input representations: STFT

I won't go into this in a tonne of detail, some high-level points:

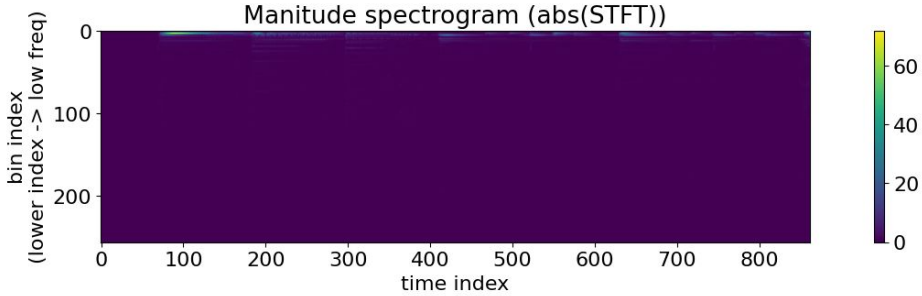
- We can control (using a library) the number of samples we take, and the number of frequency bins
- **One more issue!** Fourier coefficients are *complex* numbers (which encode phase information), but we'll typically consider only the *magnitude*, since humans are not sensitive to phase information
- See https://music-classification.github.io/tutorial/part2_basics/input-representations.html for a bit more explanation of the library function

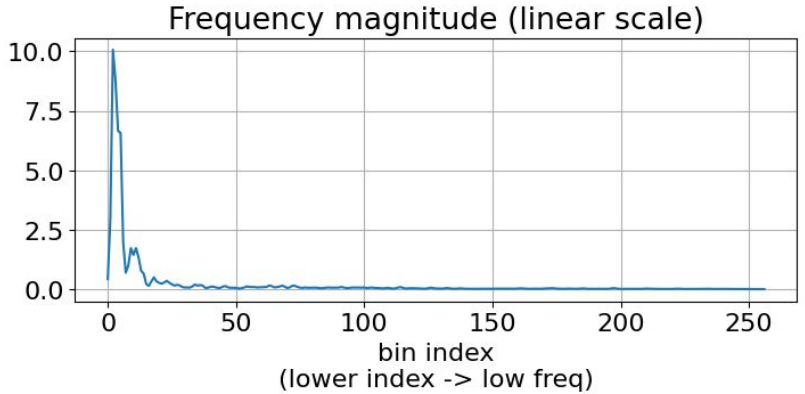
Input representations: STFT

Output looks like this (left)...

Note: everything interesting happens at the lowest frequencies!

We can confirm this (right) by plotting the frequency distribution



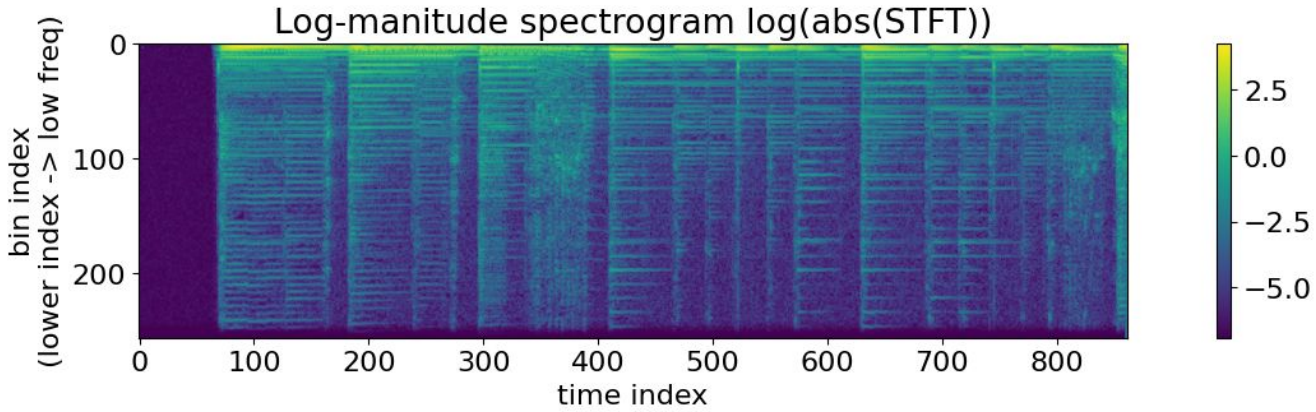


Input representations: STFT

- This (extremely sparse) representation wouldn't be very useful for downstream models
- Basically, magnitudes are higher in low frequency regions
- However, humans don't perceive magnitude linearly, rather we perceive changes in magnitude logarithmically (see: decibels)
- So, let's take the *log of the magnitude*

Input representations: STFT

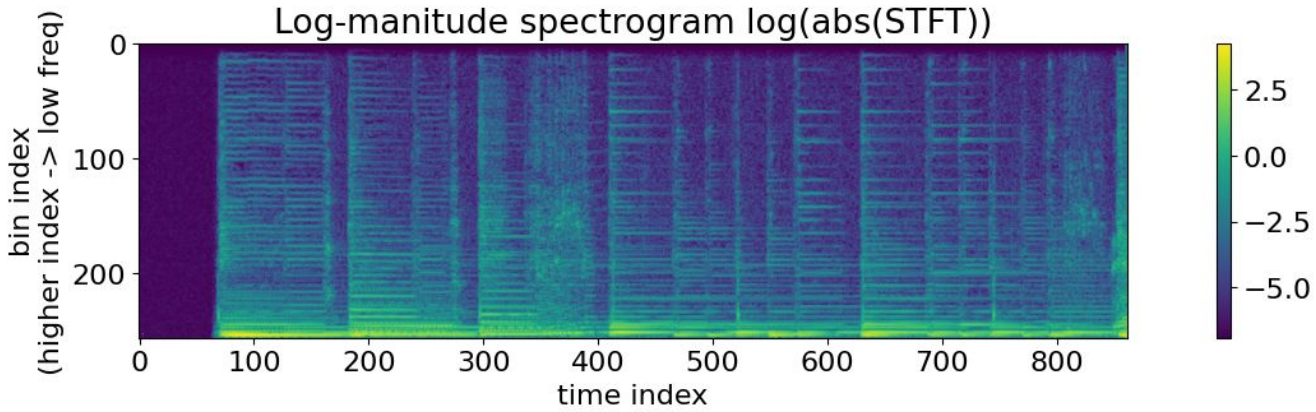
Take logarithms!



This procedure (the *log* of the *magnitude*) is common enough that this is usually what's called STFT in papers

Input representations: STFT

(usually drawn the other way up)



Input representations: STFT

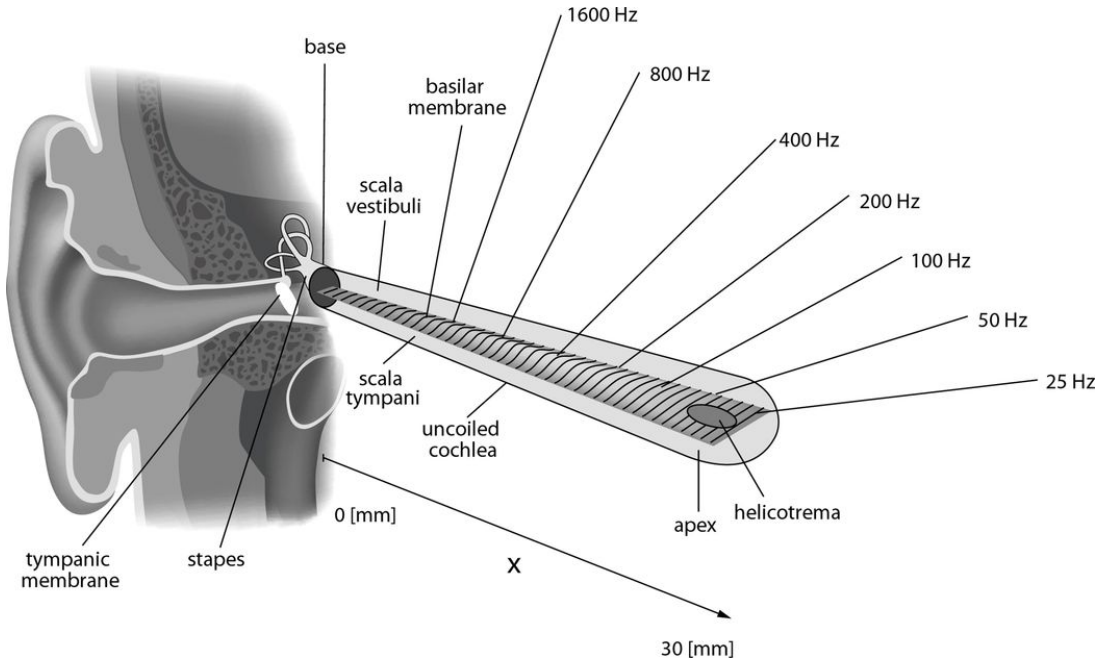
- **Note:** we did not change the range of the frequencies that we looked at, rather we changed the relative scale of the values (by taking a logarithm); that is, we're not "zooming in" on part of the frequency spectrum
- See ref for decibel scaling, which has some different choices of constants (e.g. log10 rather than log)

Input representations: spectrograms

What about a 2d representation?

- In our ear, the amplitudes of acoustic waves – waveforms – are what our eardrums respond to
- The sound travels through our auditory system; at some point, it is converted to a (sort of) 2-dimensional representation in the cochlear to perform frequency analysis; this is done by the basilar membrane physically responding to a certain frequency component

Input representations: spectrograms



(from wikipedia)

Input representations: spectrograms

What about a 2d representation?

- Perhaps put more simply, biologically, the ear responds to both *frequency* and *amplitude* (in other words, as you listen to music over time, you are “aware” of both of those things); we might want the same from our representation
- So, we’d like a representation that measures *frequency and amplitude over time*

Input representations: Melspectrograms

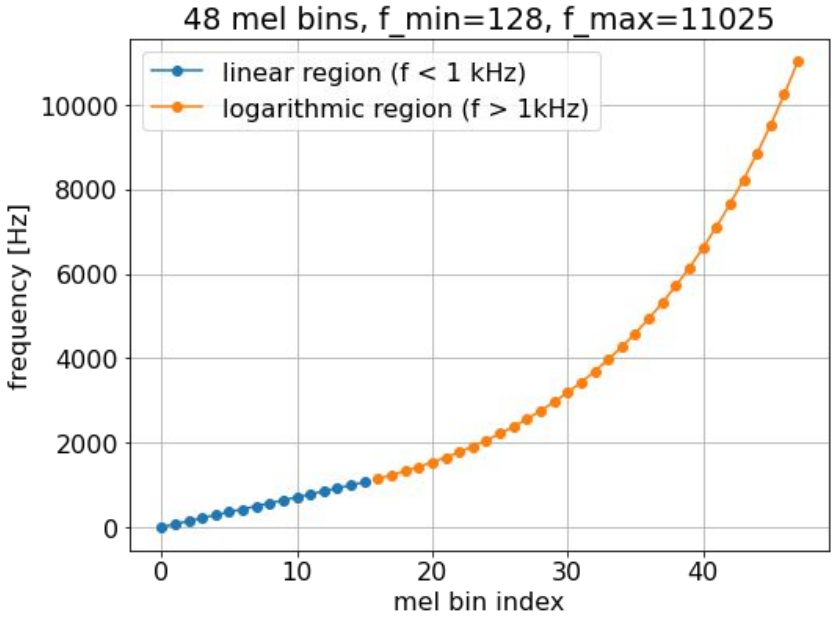
More modifications:

- Just as our perception of amplitude is non-linear, our perception of frequency is also non-linear: recall how we defined *octaves* in Module 1 – moving one octave up the keyboard corresponds to doubling the frequency
- Can also make some more minor changes: we can mostly ignore information in high frequency ranges (e.g. > 10kHz) which is barely audible, sparse, and arguably not useful

Input representations: Melspectrograms

Melspectrograms further adjust the frequency representations to mimic human perception using the *Mel scale* (https://en.wikipedia.org/wiki/Mel_scale)

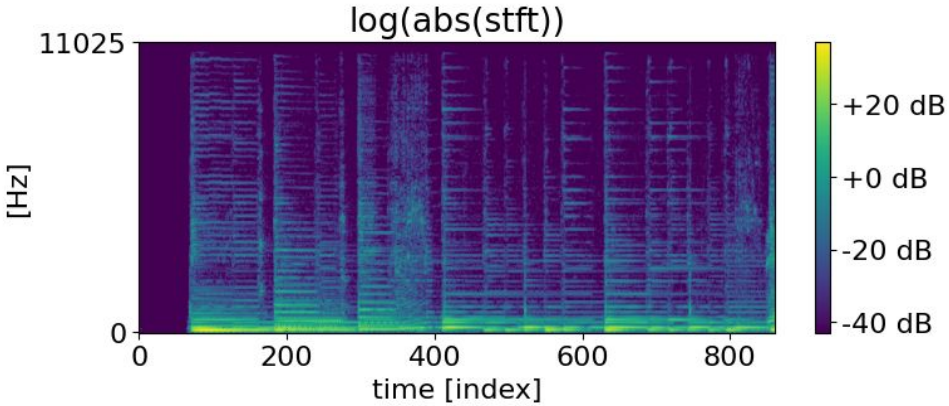
Commonly, they are linear below 1kHz and logarithmic above 1kHz

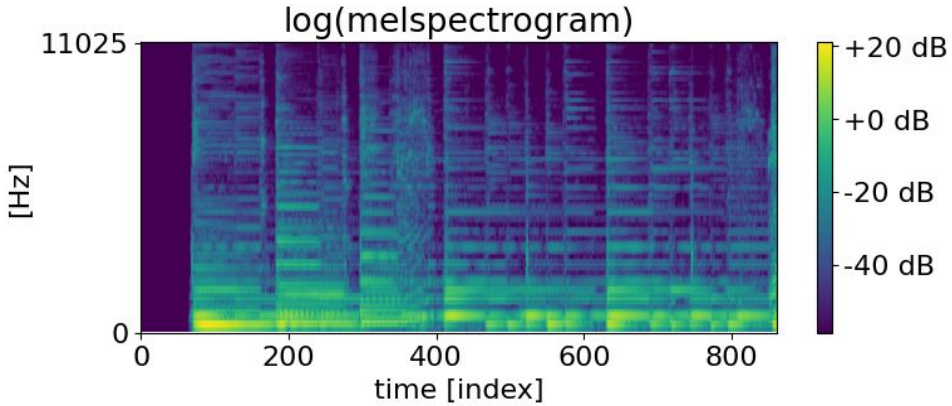


from https://music-classification.github.io/tutorial/part2_basics/input-representations.html

Input representations: Melspectrograms

STFT compared to Melspectrogram:





Input representations: Constant-Q transform

The implementation of the Melspectrogram is a pretty straightforward modification of the STFT: high-frequency bins are just aggregated into one

As such, while it's more *compact* than the STFT, it contains less information than the the STFT (even though it arguably contains most of the *relevant* information)

Although we motivated the Melspectrogram by human perception, if were interested specifically in *music* representation, why not use a representation based on frequencies of musical notes?

Input representations: Constant-Q transform

The **Constant-Q Transform** is a time-frequency representation particularly well-suited for musical signals. It decomposes audio into frequency bins whose widths are geometrically spaced, using a log_2 frequency scale in musical pitch perception.

The CQT uses a logarithmic frequency scale, where each bin corresponds to a musical note, making it analogous to a piano roll representation. The frequency of each bin is defined as:

$$f_k = f_{\min} \cdot 2^{\frac{k}{B}}$$

- ***f_min*** is the minimum frequency,
- ***B*** is the number of bins per octave,
- ***k*** is the bin index.

Input representations: Constant-Q transform

Why “Constant Q”?

Q is a “quality factor”, defining the complexity of the transform (determines the window length):

$$Q = \frac{f_k}{\delta f_k}.$$

frequency of kth filter

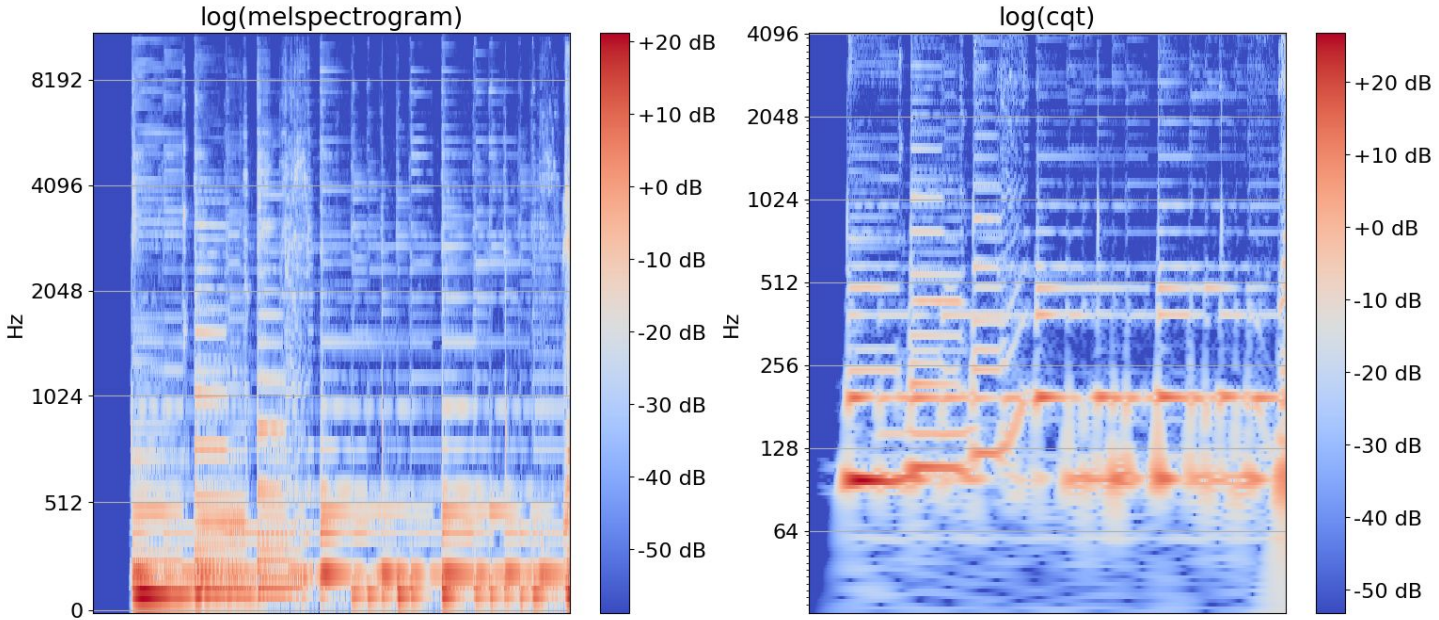
filter width

This is in contrast to the “Variable Q” transform

Disclaimer: I don’t really understand the significance of this (anyone else?) though the question of why it is named this way came up; I basically just think of this as a transform with geometrically-spaced bins which is useful for music representation (clearer when we see the output on the next slide)

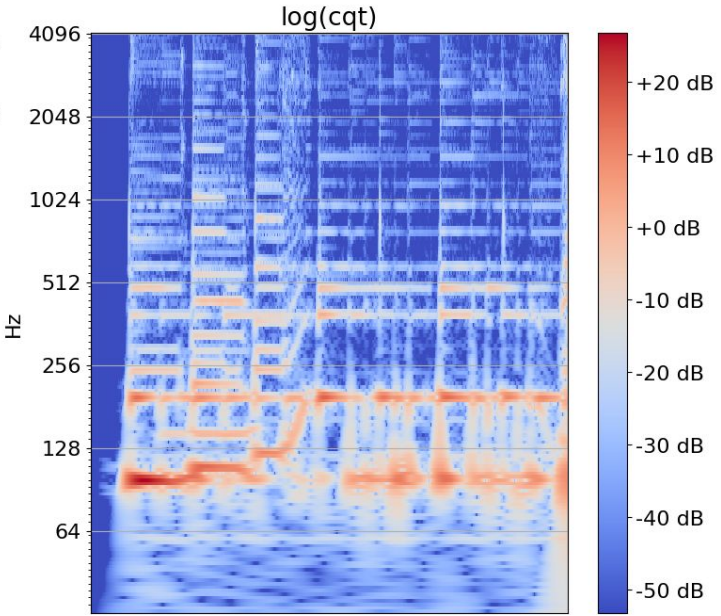
Input representations: Constant-Q transform
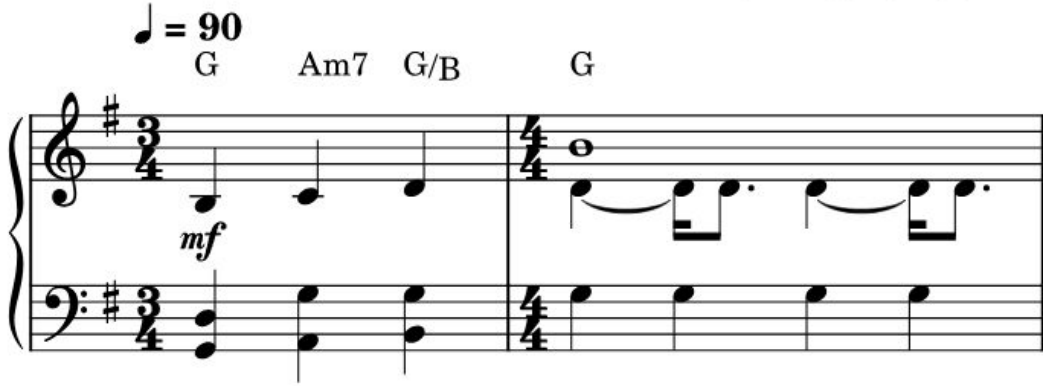
Melspectrogram vs Constant-Q transform:



Input representations: Constant-Q transform

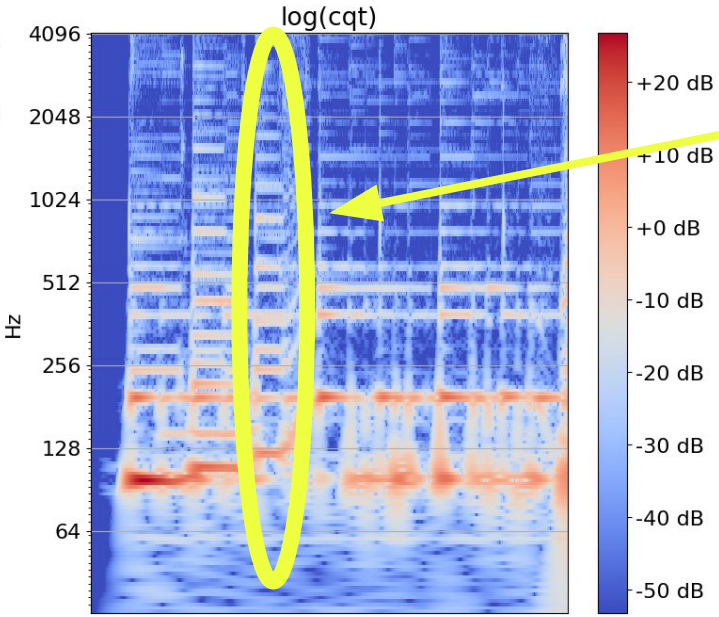
From this we can easily (?) “see” the music!





(from musescore, not the same arrangement)

Harmonics and overtones



What are these? Aren't only a few notes being played at any point?

Harmonics and overtones

Musical instruments (other than a synthesizer) won't generate "pure" sine waves: instead, playing a note will generate a range of frequencies

The **fundamental** frequency is the lowest frequency of a periodic waveform; this is what is *perceived as* being the pitch of the note

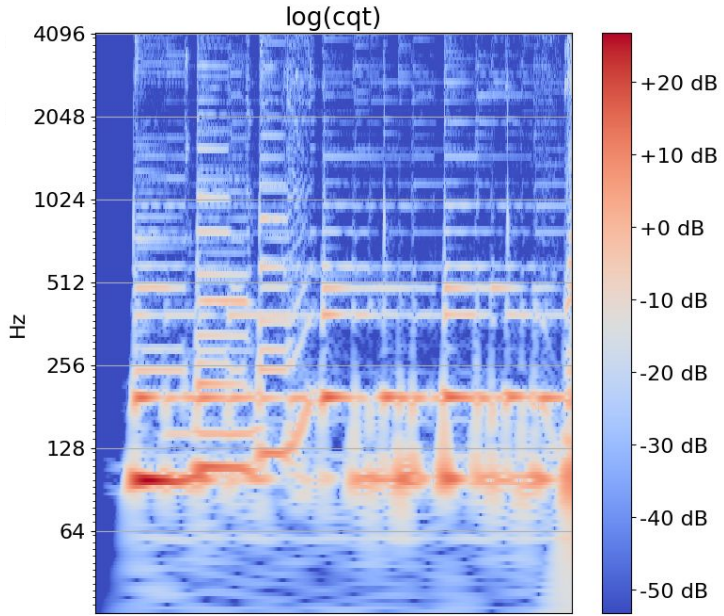
Musical instruments (or the human voice) produce **overtones:** frequencies above the fundamental:

- **Harmonic overtones** are integer multiples of the fundamental frequency (2x, 3x, etc.); these are produced by string instruments, various woodwinds, or the human voice; this series of tones is called the **harmonic series**
- Some instruments (like drums) produce **inharmonic overtones,** i.e., overtones that aren't integer multiples of the fundamental frequency

Harmonics and overtones

The above perhaps isn't particularly relevant to this class, though does give some sense of why some tasks (like music transcription, or extracting notes from an audio file) are quite hard: basically, the spectrogram isn't *quite* like a continuous piano roll which just has high activation for the notes being played, but rather it has high activation for the notes and overtones

What happens if we change the timing, tempo, or pitch?



Timing changes: *translate* the spectrogram *horizontally*

Tempo changes: *stretch or shrink* the spectrogram *horizontally*

Pitch changes: *translate* the spectrogram *vertically*

What happens if we change the timing, tempo, or pitch?

So, to learn from spectrograms, we'll need models that can handle (or "discover" / "learn") *invariances* to translation and scale

CNNs! (next topic)

Mel-frequency cepstral coefficients (MFCC)

Finally: can we represent spectrograms more compactly?

(based on “Deep Learning for Music Analysis and Generation: Fundamentals for Musical Audio”:

https://github.com/affige/DeepMIR/blob/main/2024/lecture02_fundamentals_audio.pdf)

Mel-frequency cepstral coefficients (MFCC)

I won't go into details about this one but the basic procedure is as follows:

1. Compute the spectrogram
2. Group the FFT bins according to the perceptually motivated Mel-filter bank (linear until 1,000 Hz and logarithmic above it, usually with triangular overlapping windows)
3. Take logs and apply a DCT

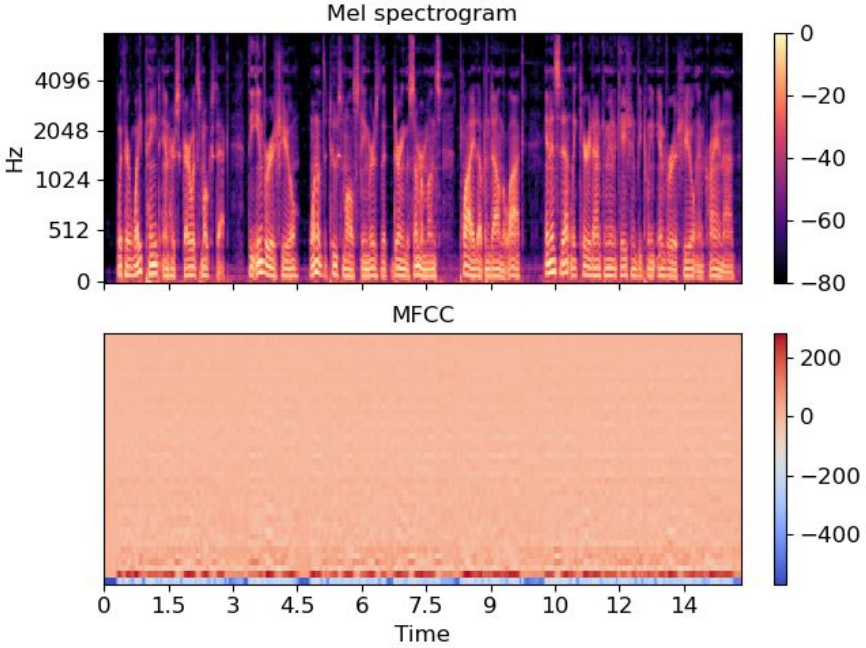
Compact, though perhaps less interpretable representation of the spectrum (1,024-dim → 13 coefficients), which somehow captures the overall energy distribution in the spectrum

(we'll use this a bit in homeworks and assignments to keep runtimes manageable)

https://github.com/affige/DeepMIR/blob/main/2024/lecture02_fundamentals_audio.pdf

Mel-frequency cepstral coefficients (MFCC)

MFCC compared to Melspectrogram:



Input representations

Software (from https://music-classification.github.io/tutorial/part2_basics/input-representations.html):

- CPU
  - **librosa:** librosa.stft internally uses stft but with a more tailored and carefully chosen API with default values. It also includes functions to compute a melspectrogram and a CQT [MRL+15].
  - **scipy:** scipy.signal.stft is a cpu-based implementation of STFT using FFT [VGO+20].
  - Essentia provides C++-based implementation of STFT and melspectrogram as well as their python bindings [BWGomezGutierrez+13].
- CPU & GPU
  - **Torchaudio:** One can compute STFT using `torchaudio.functional.spectrogram` . It also includes a wide variety of functions and utilities such as `amplitude to DB` and resampling [YHN+21].
  - **nnAudio:** It has multiple versions of CQT computation functions as well as others e.g., STFT and melspectrogram. Its STFT computation is based on Conv1D, not FFT [CAAH20].
  - **Tensorflow:** It has a native support of stft. However, it is slow on cpu, which might be critical if your model is going to be deployed on cpu-only machines. It, by the way, even has a function to convert an STFT to melspectrogram [ABC+16].
  - **kapre:** kapre.time_frequency includes tf.keras layers such as STFT and melspectrogram as well as utilities as decibel conversion. Its STFT layer is a wrapper of tf.signal.stft [CJK17].

Take-homes

- Try to get a solid grasp of *what a spectrogram is representing* (how it is actually computed is less important)
- Think about the differences between e.g. a spectrogram and other representations like a piano roll
- Try to understand the similarities and differences between spectrograms and images, especially in terms of invariances (e.g. what do translation and scaling represent on a spectrogram? Should predictive models be invariant to these operations or not?)

Predictive pipelines for music & MIR

2.3: Convolutional Neural Networks

Convolutional Neural Networks

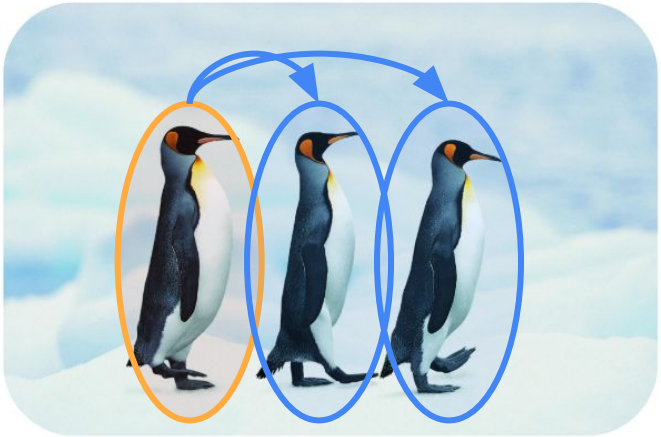
From: **“Generative AI for Music and Audio Creation” (Michigan):**

https://hermandong.com/teaching/pat498_598_fall2024/

CNNs: Reusable pattern detectors

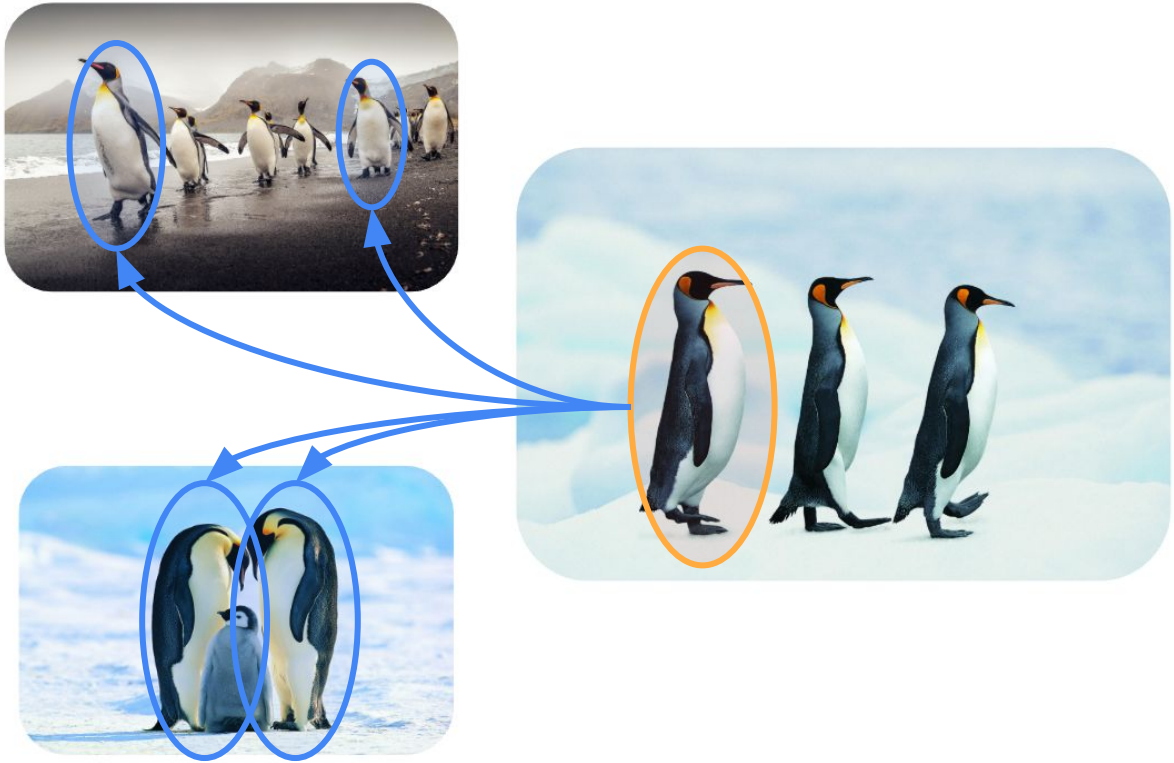
Intuition: Want to learn a **reusable local pattern detector**

- Widely used in computer vision
- Also used for music and audio
  - Representing music as piano rolls
  - Representing audio as spectrograms



Objects have similar **local patterns** (coloring, beak shape, etc.); the patterns learned from one penguin can be reused to recognize similar features in the others.

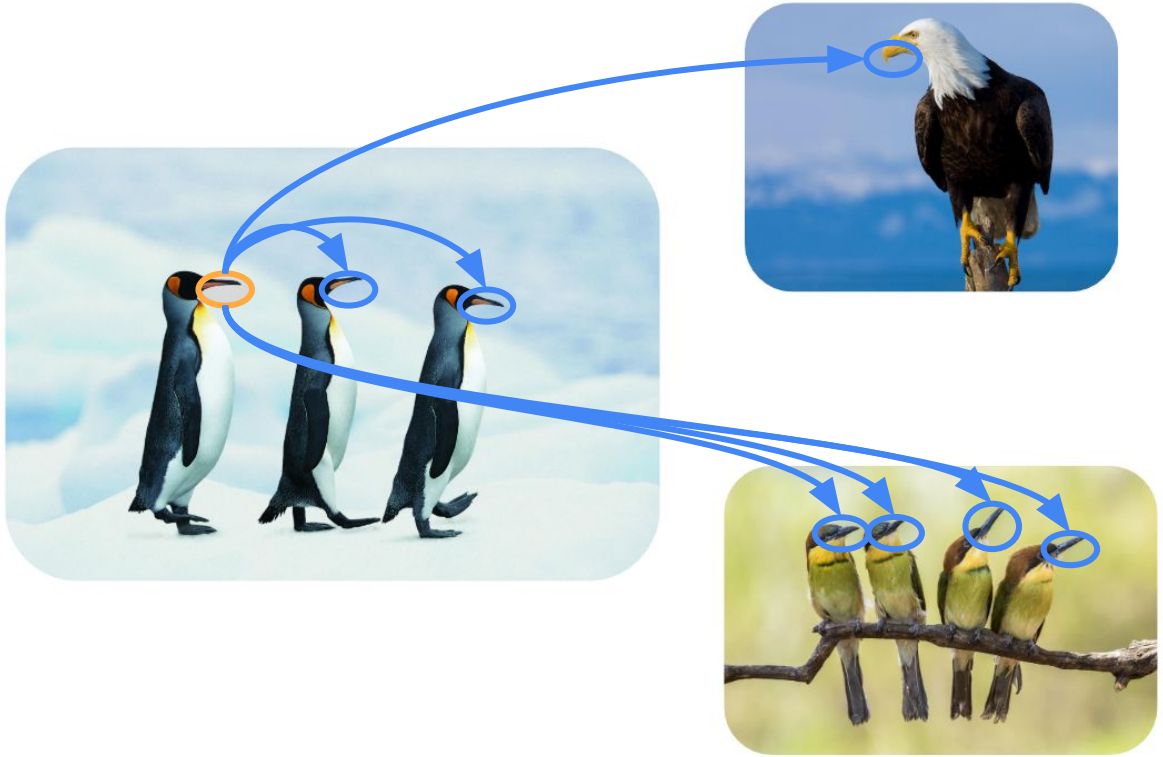
CNNs: Reusable pattern detectors



Once a CNN learns what a penguin “looks like” in one image, it can apply that knowledge to detect penguins in new scenes, regardless of the background, pose, or surrounding elements

CNNs: Reusable pattern detectors

Detected patterns might generalize: parts are shared not only across different penguins in different scenes, but also across other types of birds

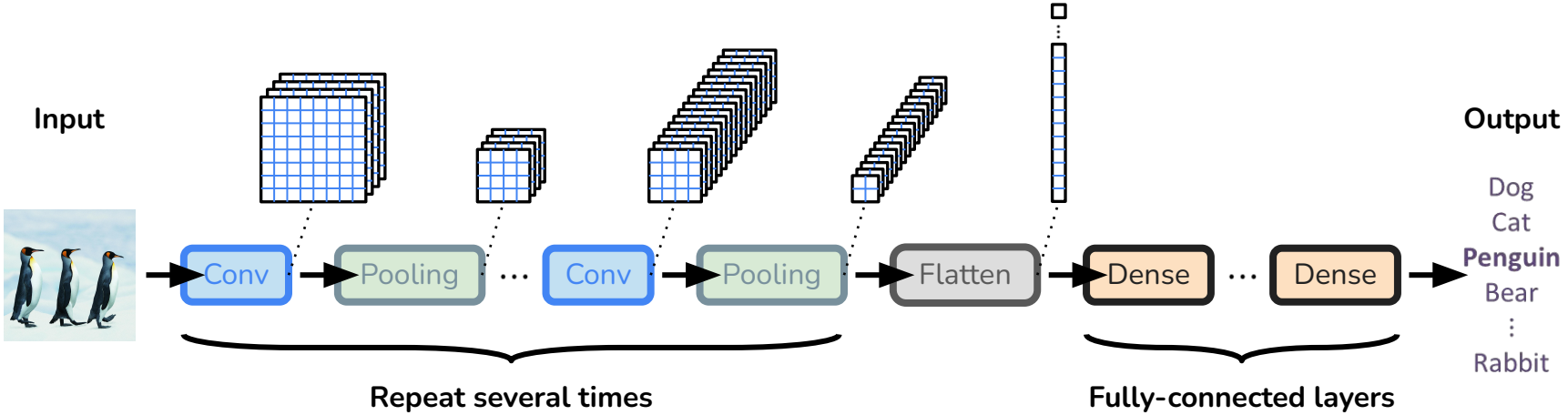


CNNs: Reusable pattern detectors



Parts of a CNN

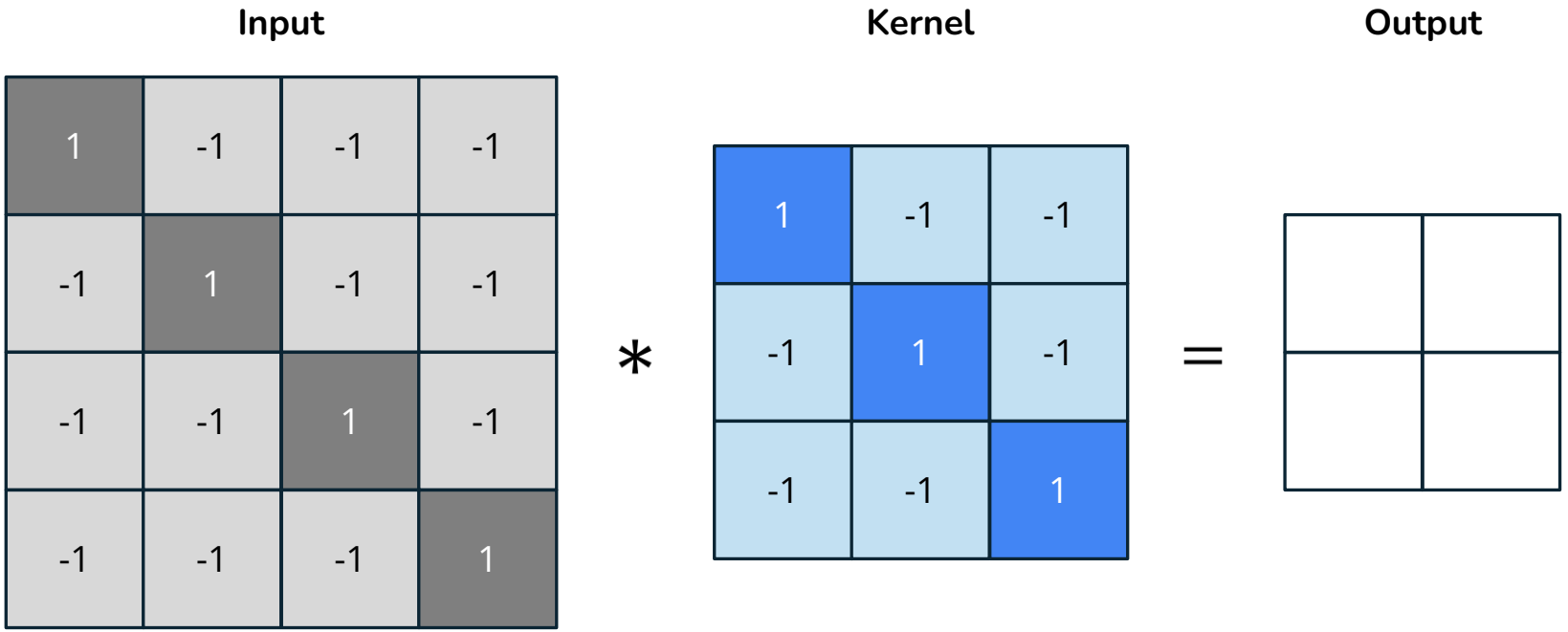
(explanation on next slide)



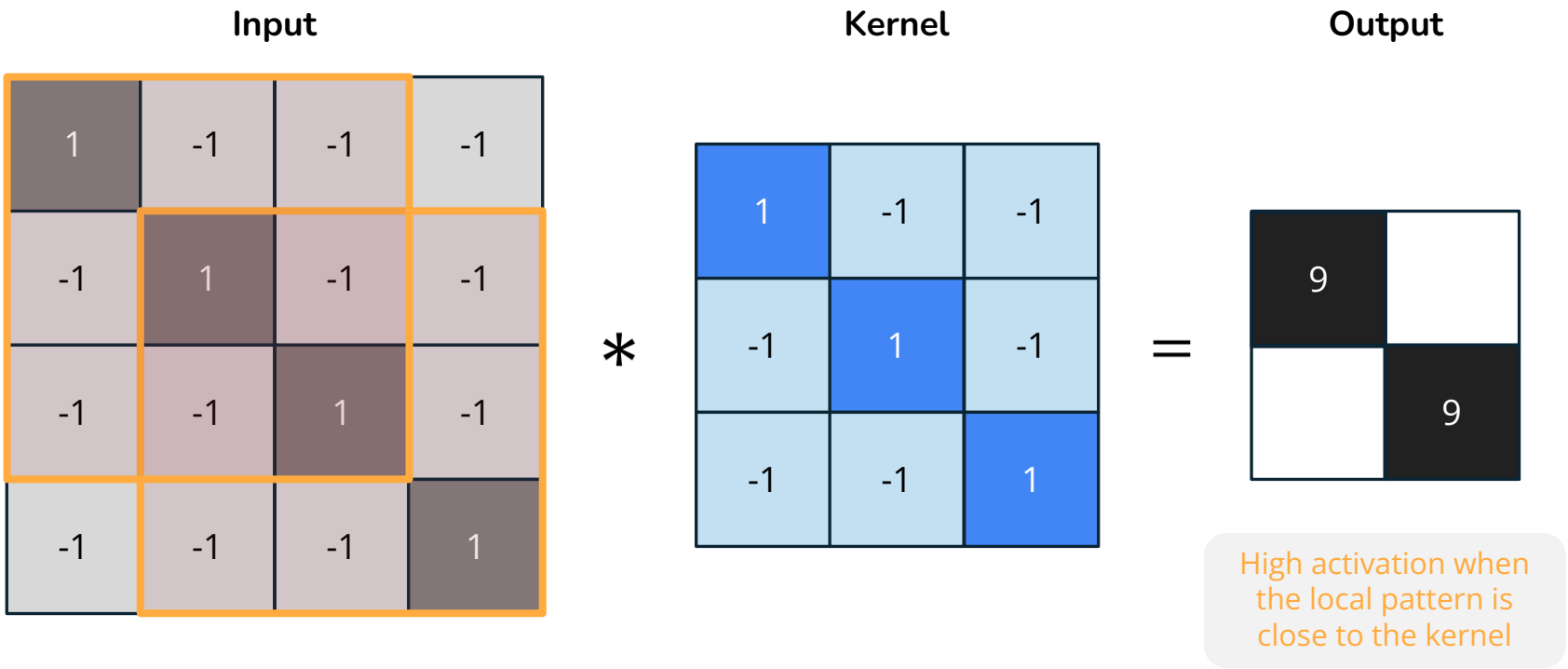
Parts of a CNN

- An image passes through multiple layers of convolutions and pooling
- Early convolutional layers extract low level features, such as edges, shapes, colors
- Later layers extract high level features (flippers, beak, eyes, etc.)
- Pooling layers are responsible for “aggregating” information into a few important features
- After several rounds of convolution and pooling, data is flattened into a 1D vector, which is passed through fully connected (dense) layers that interpret the extracted features and make the final prediction

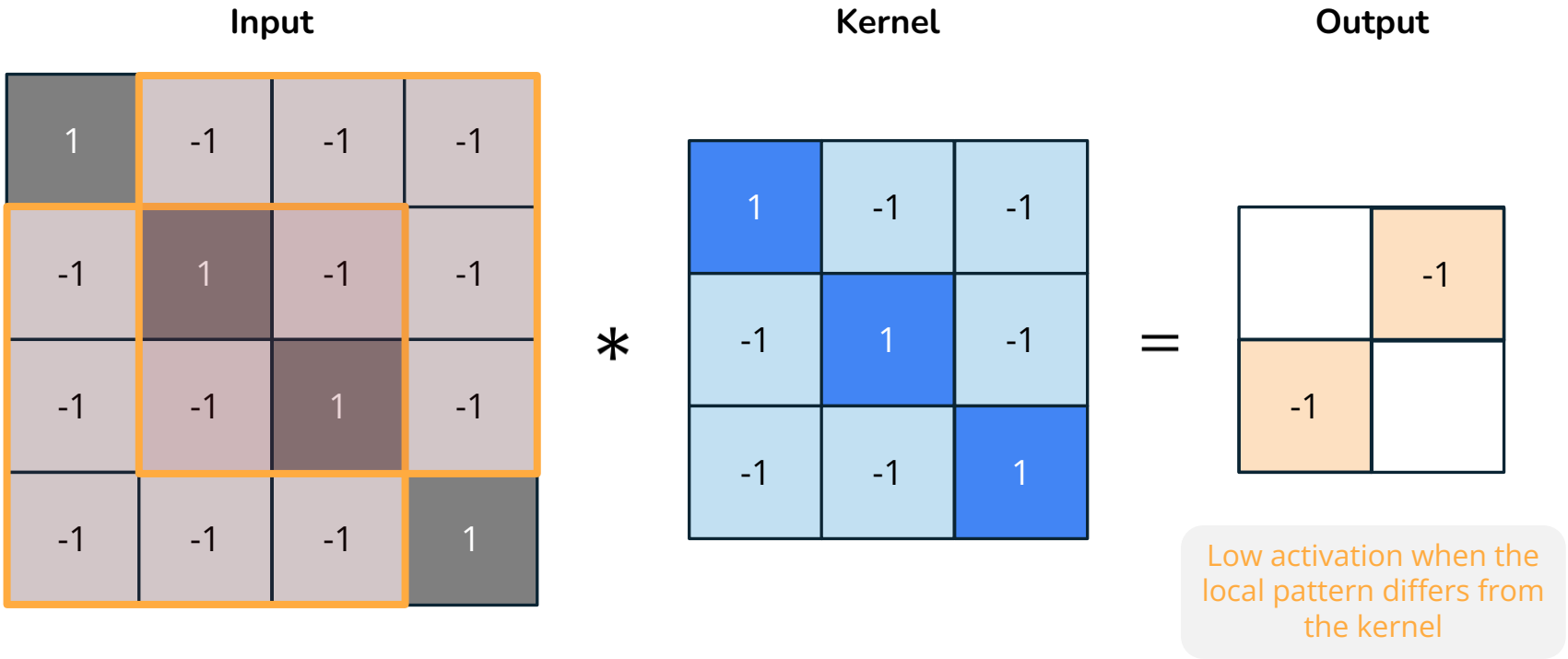
What is “convolution”?



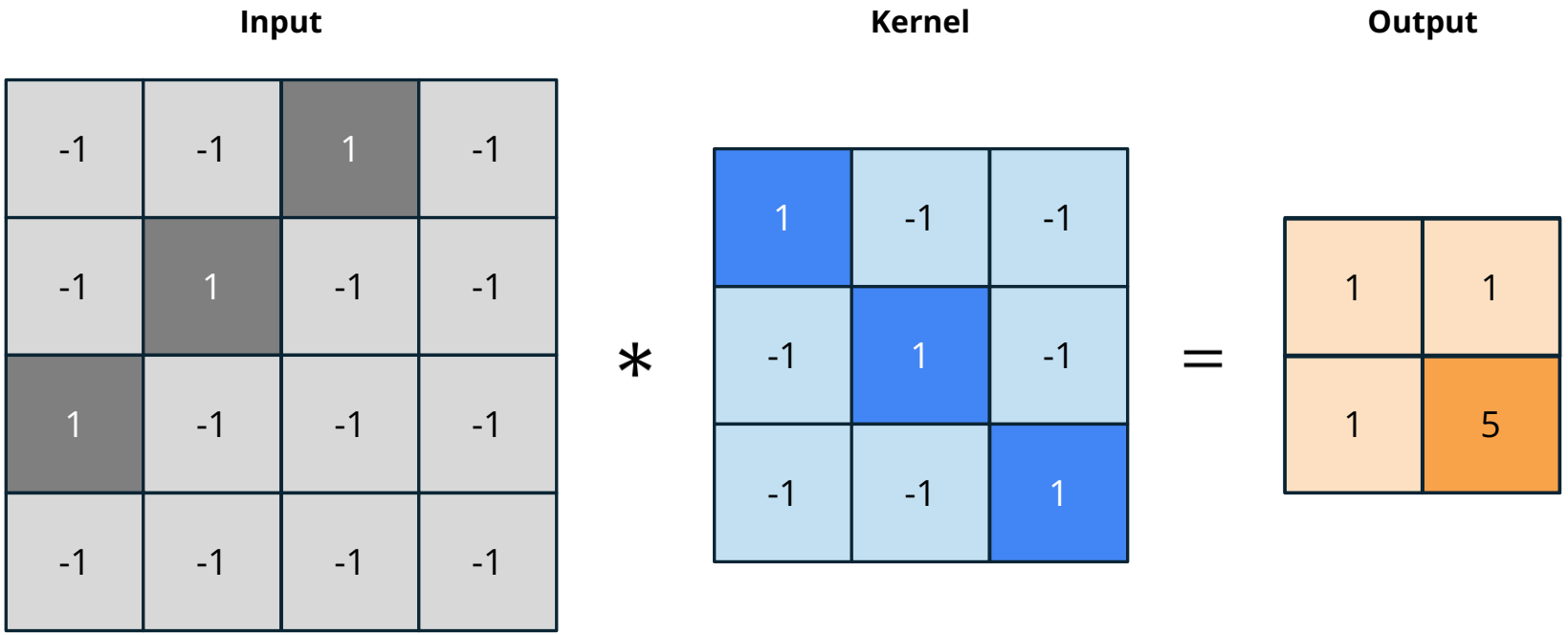
2D Convolution



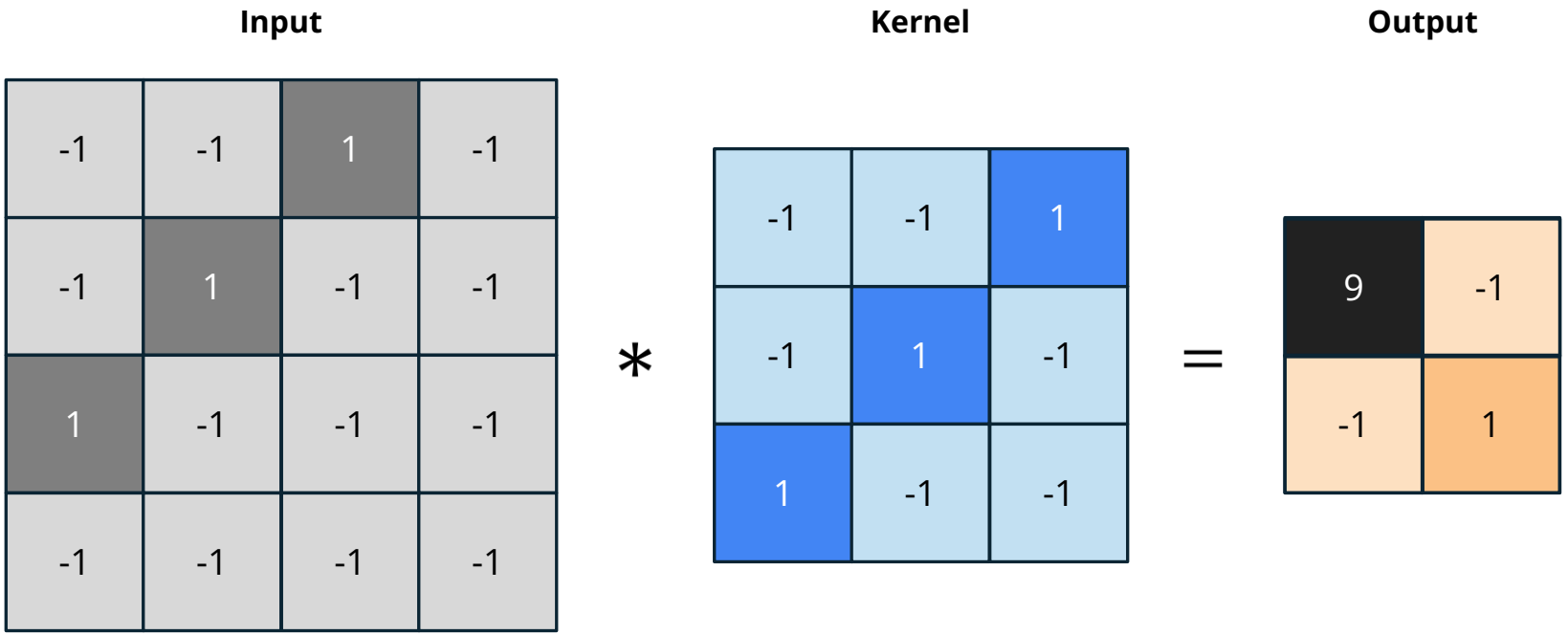
2D Convolution



2D Convolution



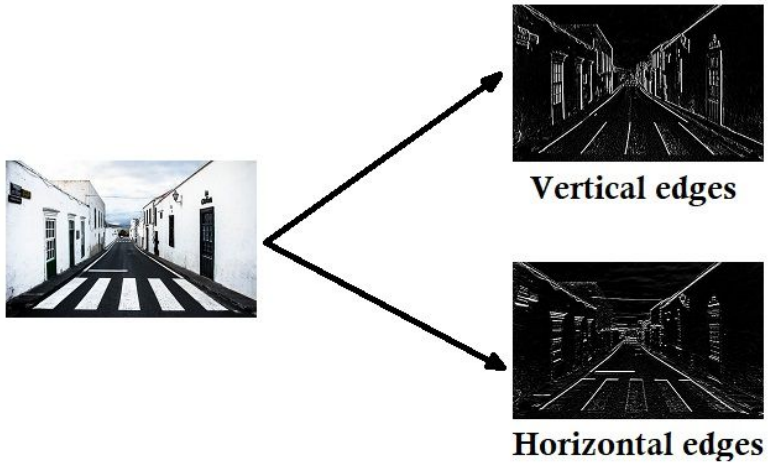
2D Convolution



2D Convolution

Now imagine these filters being applied over an entire image

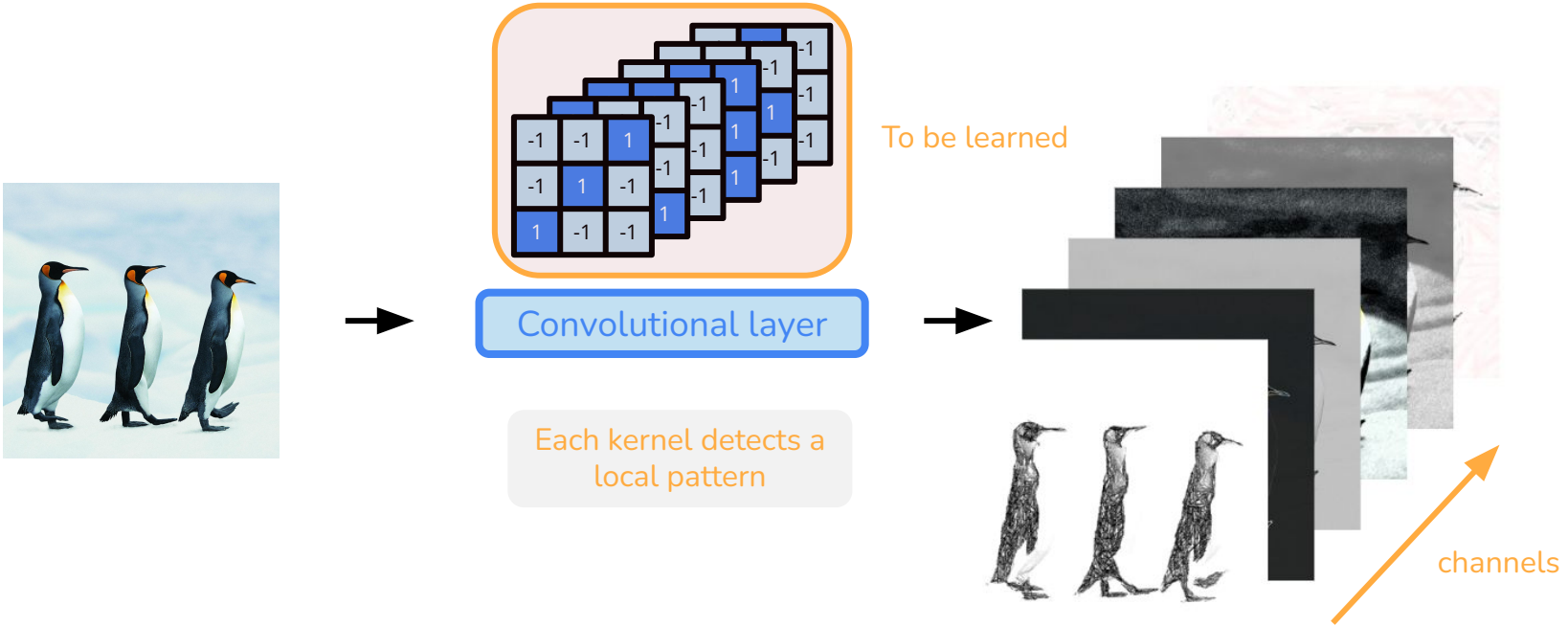
The ***response*** *to the filter* will capture how much different parts of the image capture different properties



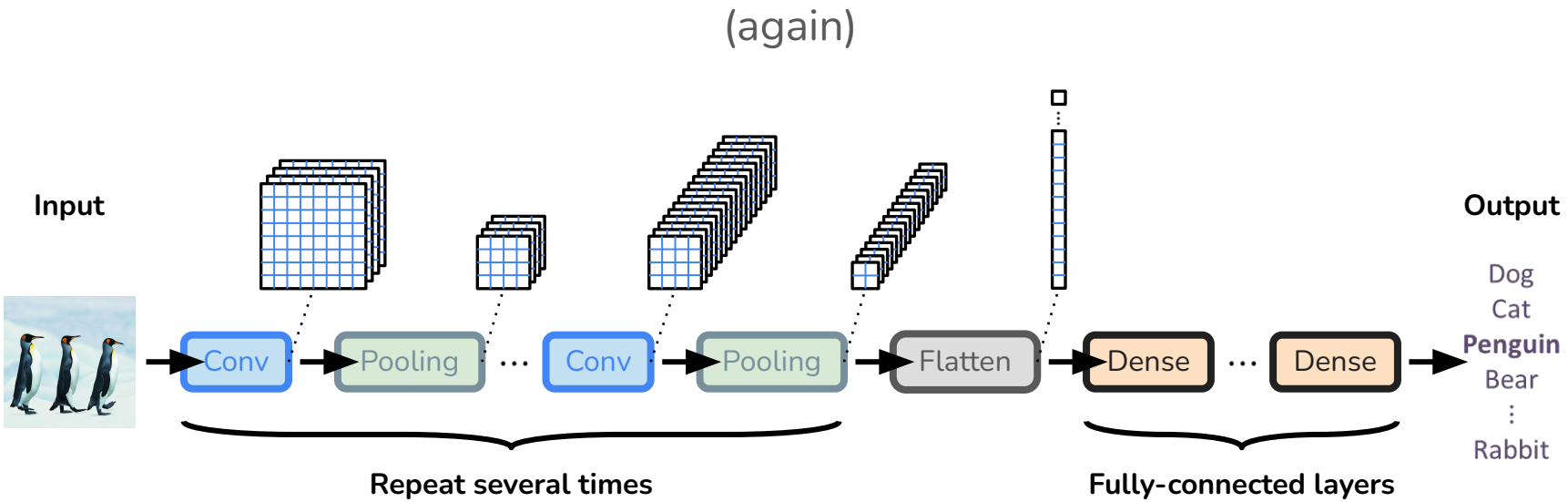
from https://datahacker.rs/edge-detection/

2D Convolution

A convolutional **layer** in a CNN consists of many **learnable kernels** (channels)

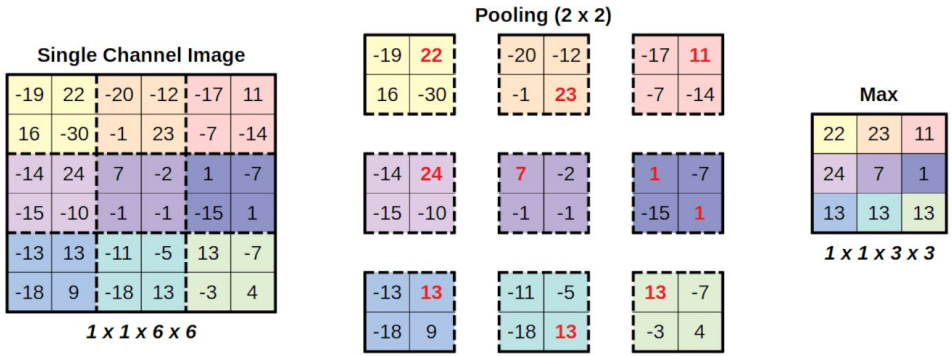


Parts of a CNN

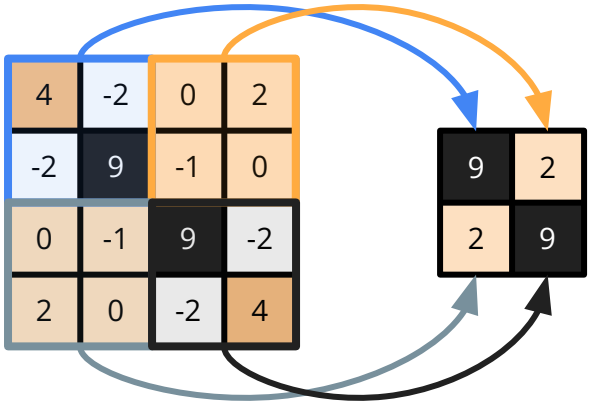


Pooling layers

Pooling layers downsample and aggregate information between convolutional layers



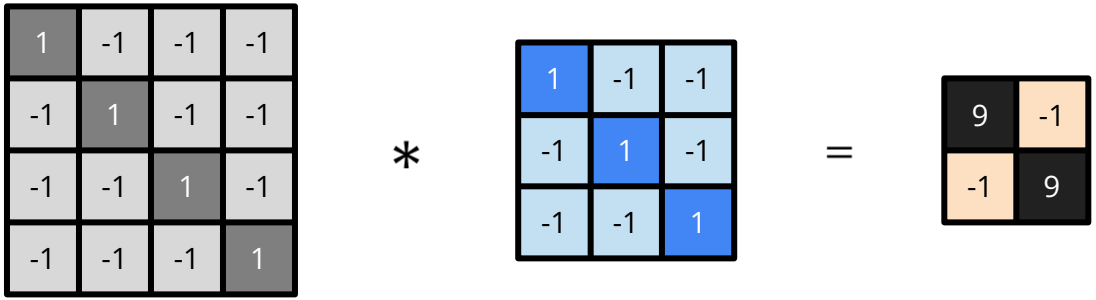
(from Daniel Voigt Godoy - https://github.com/dvgodoy/dl-visuals/)



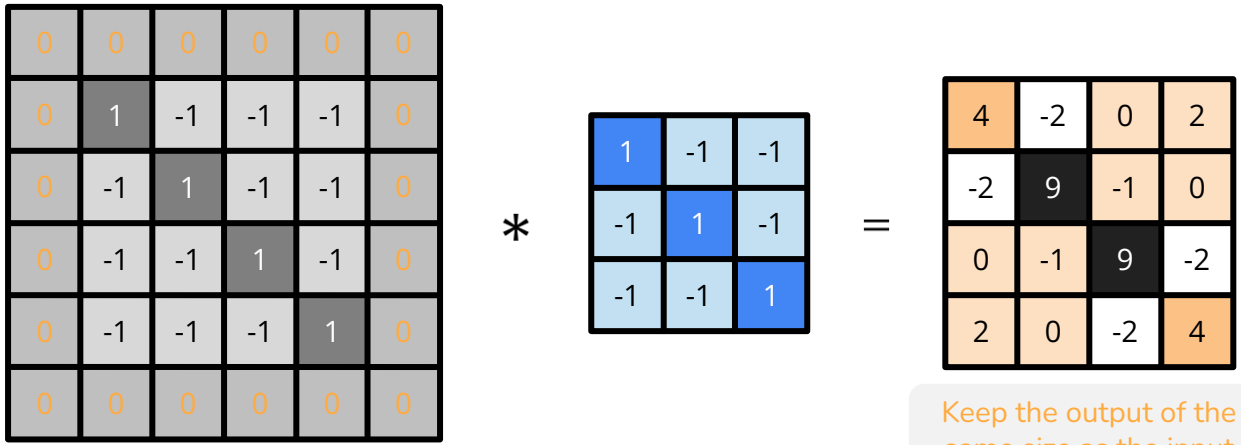
Downsample and keep the strongest activation in each block

More (mostly not relevant) details: padding & shape

`padding="valid"`



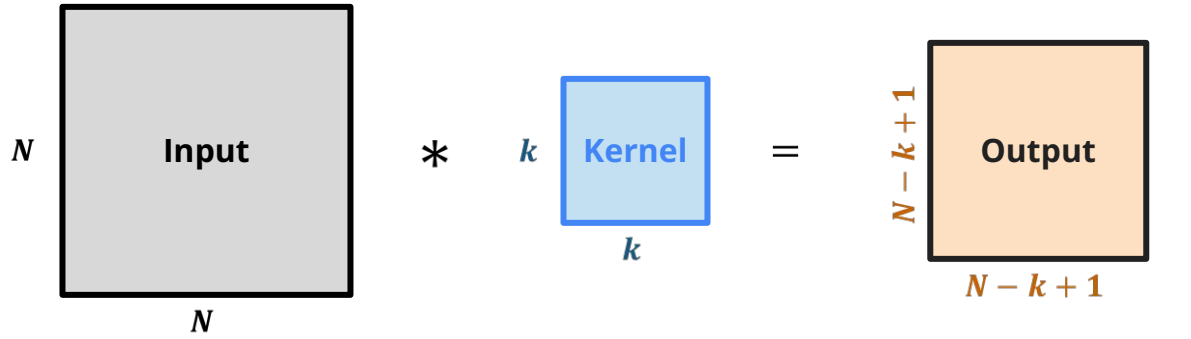
`padding="same"`



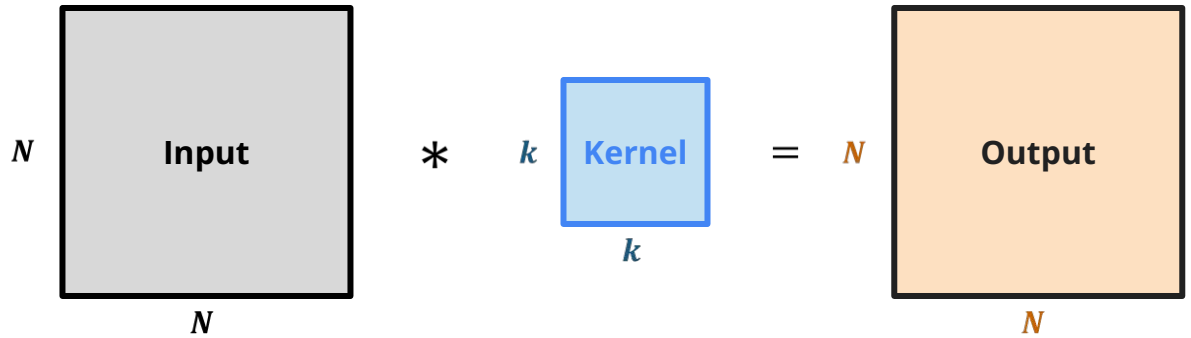
Keep the output of the same size as the input

More details: padding & shape

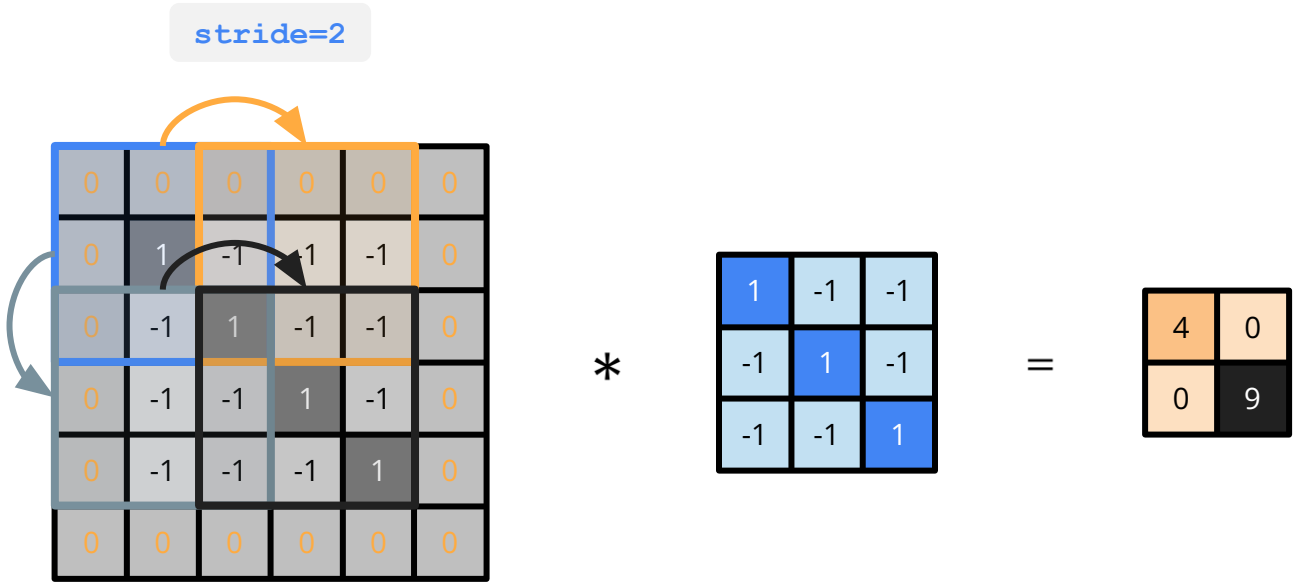
`padding="valid"`



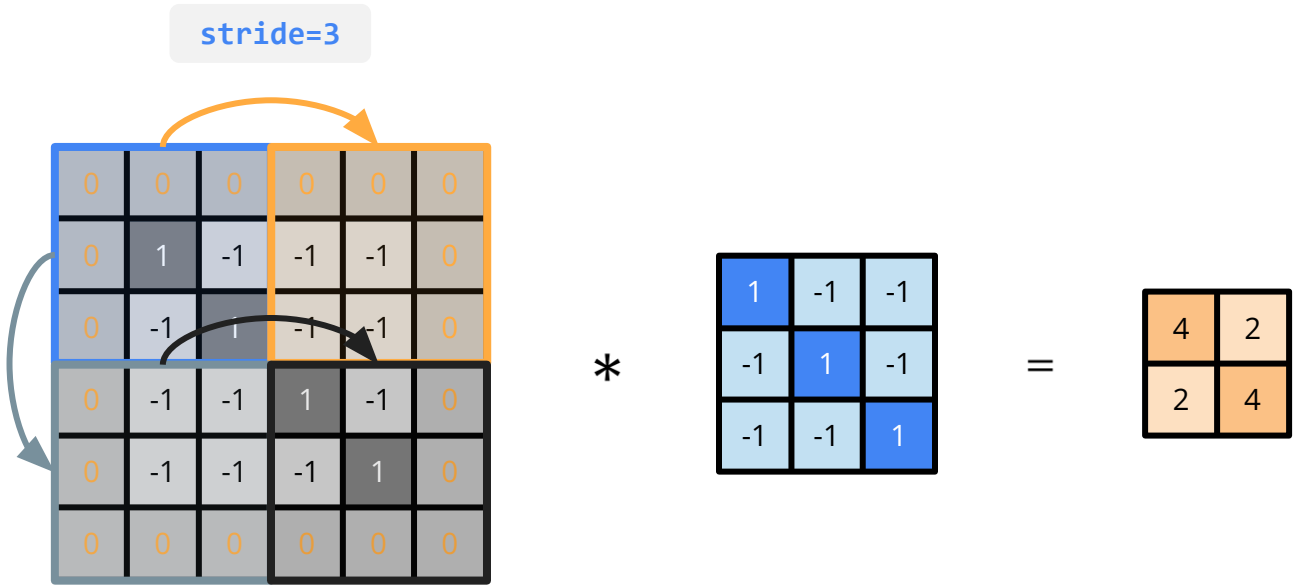
`padding="same"`



More details: stride



More details: stride



A real example

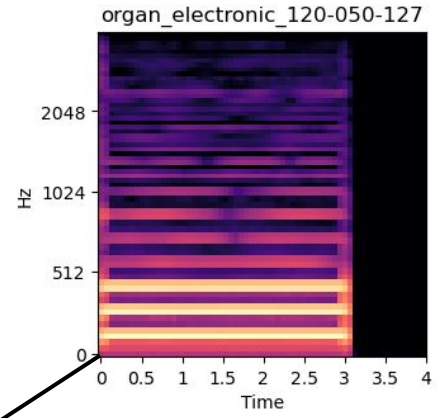
```python
class CNN(nn.Module):
    """A basic convolutional neural network."""
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, 3, padding="same")
        self.conv2 = nn.Conv2d(16, 32, 3, padding="same")
        self.conv3 = nn.Conv2d(32, 64, 3, padding="same")
        self.conv4 = nn.Conv2d(64, 128, 3, padding="same")
        self.pool = nn.MaxPool2d(2, 2)
        self.fc = nn.Linear(128 * 4 * 4, n_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x
```

Input channels

Output channels

Kernel size



Input: 1 x 64 x 64

16 x 32 x 32 = 16384

32 x 16 x 16 = 8192

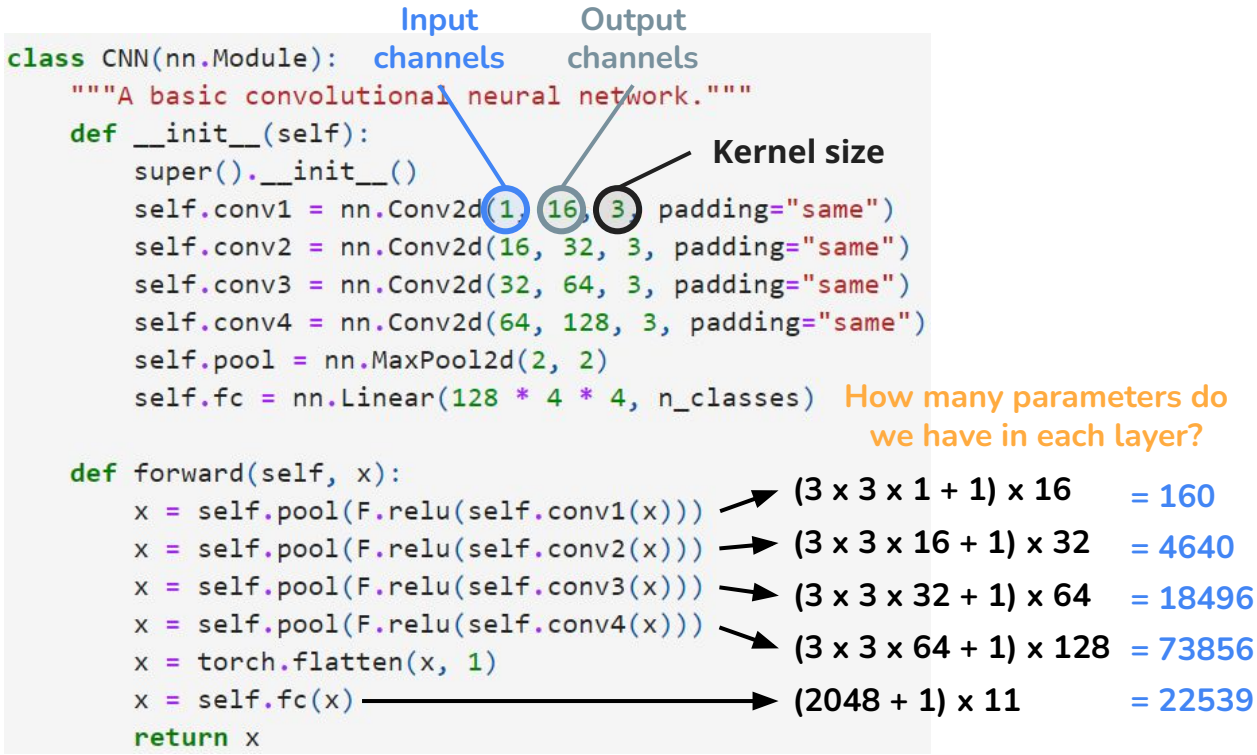
64 x 8 x 8 = 4096

128 x 4 x 4 = 2048

Total number of features decreases as we go deeper

More channels, finer resolution as we go deeper

A real example



CNNs

- Learn local patterns: in computer vision we might care about *shapes*, *colors*, *textures*, etc.; in audio we might care about
  - repeated musical patterns
  - timbral / textural characteristics (what does a drum spectrogram “look like”?)
  - “shapes” corresponding to chords, etc.
- Are invariant to “shifts”: in computer vision we care about invariance to translation and scale; in audio we might care about
  - invariance to pitch (vertical translation in spectrogram)
  - invariance to timing (horizontal translation in spectrogram)
  - invariance to tempo (horizontal scaling)

CNNs for the symbolic and audio domain

Note: we can potentially use CNNs for both symbolic *and* audio-domain representations of music

- Audio-domain: spectrograms (as input and/or output)
- Symbolic: piano rolls! (Module 3)

What does a CNN learn? (kernels in AlexNet)

Layer 1:

Learned CNN kernels

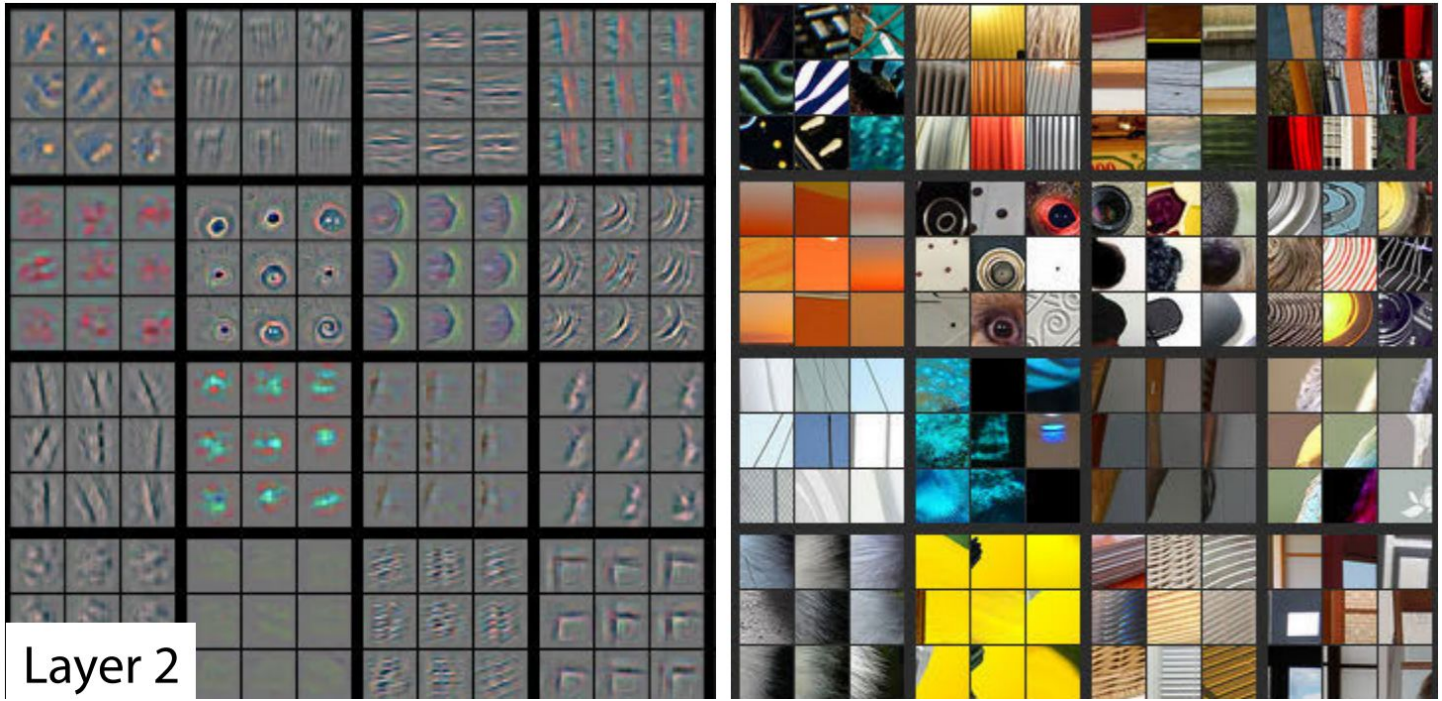


Top activations



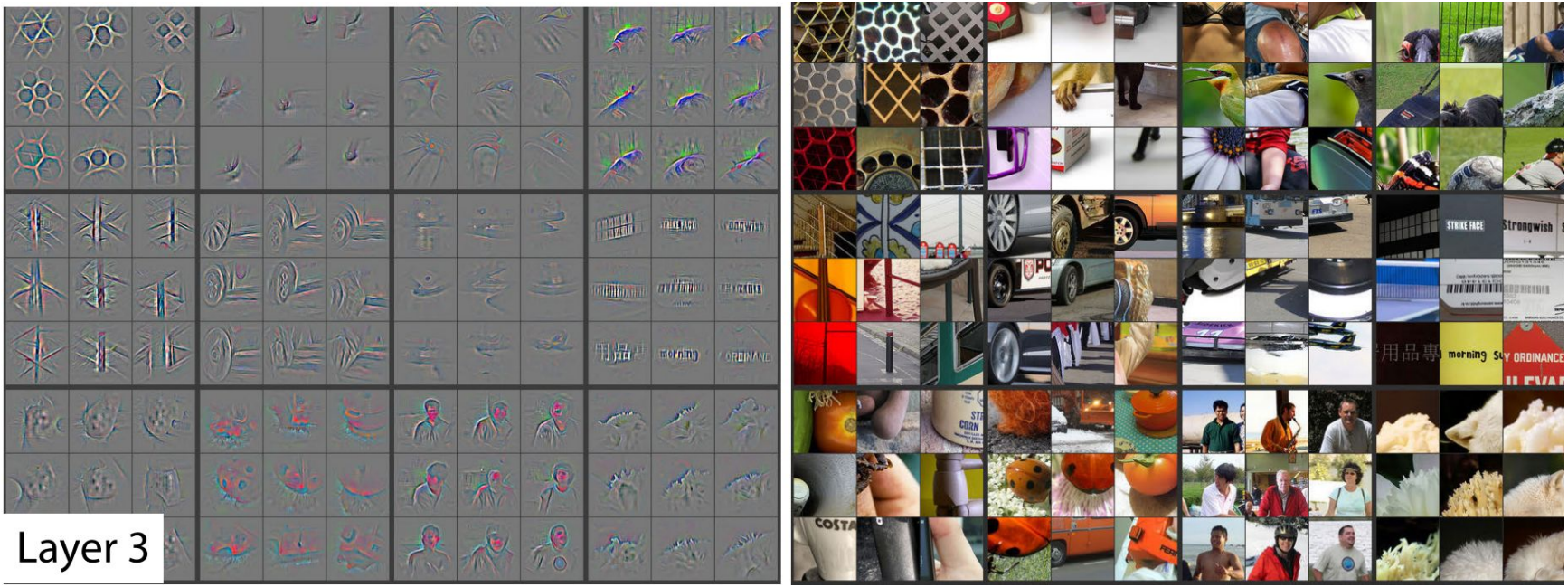
Matthew D. Zeiler and Rob Fergus, "Visualizing and Understanding Convolutional Networks," *ECCV*, 2014.

What does a CNN learn?



Matthew D. Zeiler and Rob Fergus, "Visualizing and Understanding Convolutional Networks," *ECCV*, 2014.

What does a CNN learn?



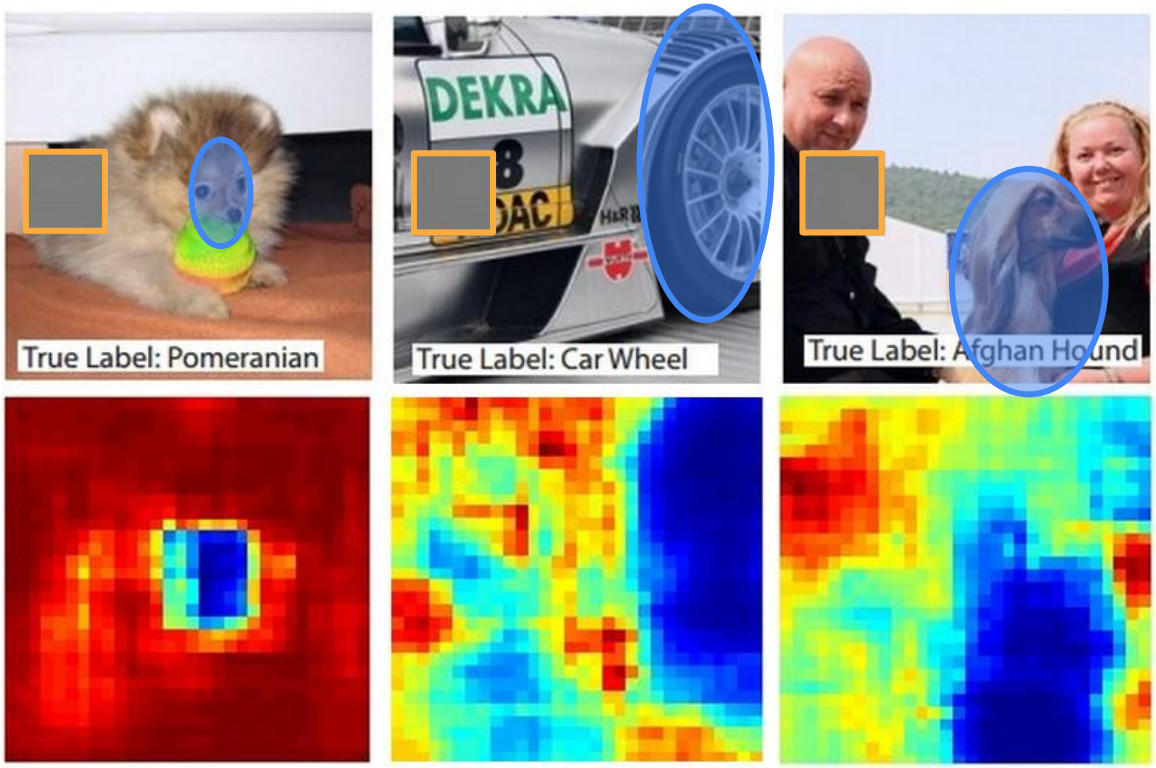
Matthew D. Zeiler and Rob Fergus, "Visualizing and Understanding Convolutional Networks," *ECCV*, 2014.

What does a CNN learn?



Matthew D. Zeiler and Rob Fergus, "Visualizing and Understanding Convolutional Networks," *ECCV*, 2014.

What does a CNN learn?



cs231n.github.io/understanding-cnn/

Take-homes

- Understand how CNNs capture different patterns via “filters” that respond to parts of an image (or spectrogram)
- Understand how this type of model can capture variances, invariances, and think about how they’re relevant to music understanding (we’ll see MIR examples in the next section)

Predictive pipelines for music & MIR

2.4: CNN architectures for MIR

CNN architectures for MIR

Based on “Music Classification: Beyond Supervised Learning, Towards Real-world Applications” (Architectures): https://music-classification.github.io/tutorial/part3_supervised/architectures.html

(which also has excellent code)

CNN architectures for MIR

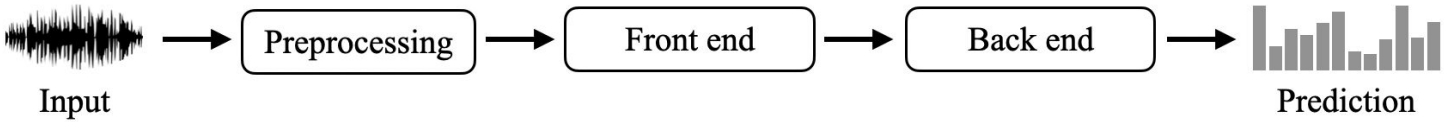
Now that we know how a CNN works, how is this type of model used in an MIR context?

Two topics:

1. Survey of some architectures showing how CNNs can be adapted for MIR
2. Data processing (really “augmentation”) techniques to build datasets for CNN training

CNN architectures for MIR

Basic components:



Preprocessing: (e.g.) convert audio to Mel-spectrogram; extract different samples from an audio sequence, etc.

Front end: (e.g.) 2D CNN on samples

Back end: aggregate the outputs (e.g. pooling), basically, summarize a bunch of extracted features

from https://music-classification.github.io/tutorial/part3_supervised/architectures.html#

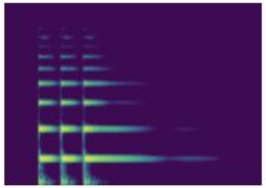
Summary of architectures

Model	Preprocessing	Input length	Front end	Back end	Training	Aggregation
FCN	STFT	29.1s	2D CNN	.	song-level	.
VGG-ish / Short-chunk CNN	STFT	3.96s	2D CNN	Global max pooling	instance-level	Average
Harmonic CNN	STFT	5s	2D CNN	Global max pooling	instance-level	Average
MusiCNN	STFT	3s	2D CNN	1D CNN	instance-level	Average
Sample-level CNN	.	3s	1D CNN	1D CNN	instance-level	Average
CRNN	STFT	29.1s	2D CNN	RNN	song-level	.
Music tagging transformer	STFT	5s-30s	2D CNN	Transformer	instance-level	Average

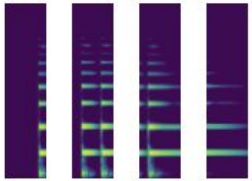
from https://music-classification.github.io/tutorial/part3_supervised/architectures.html#

CNN architectures for MIR

Song vs instance level tagging:



Song-level



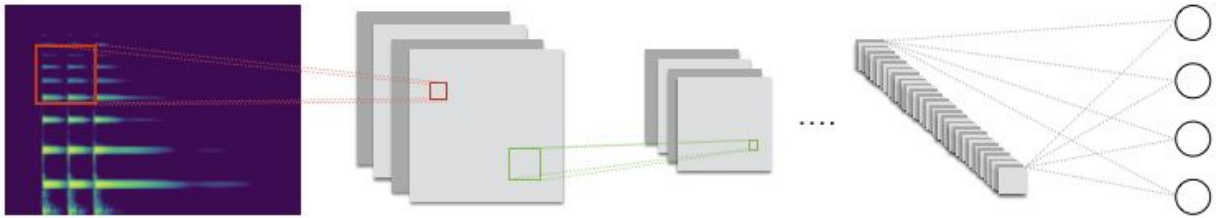
Instance-level

What is the right *granularity* at which to make predictions? **“Instance level”** features, extracted from short segments, may capture more local properties (e.g. timbral characteristics) versus longer segments which may capture larger-scale properties but miss details

from https://music-classification.github.io/tutorial/part3_supervised/architectures.html#

Fully Convolutional Networks (FCNs)

Keunwoo Choi, György Fazekas, and Mark Sandler. *“Automatic tagging using deep convolutional neural networks”*, 2016



- Melspectrogram
- 3 CNN layers
- (all of these examples are for ***tagging***)

from https://music-classification.github.io/tutorial/part3_supervised/architectures.html#

VGG-ish / Short-chunk CNNs

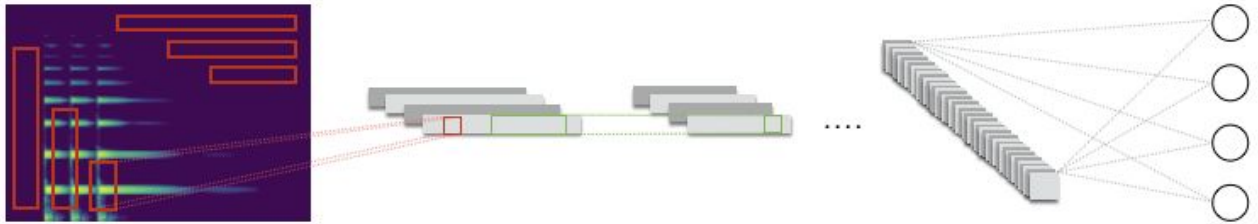
e.g. Minz Won, Andres Ferraro, Dmitry Bogdanov, and Xavier Serra. “*Evaluation of cnn-based automatic music tagging models*”, 2020

- Uses instance level rather than song-level training
- 3.69s audio chunks
- Max-pooling to summarize features

from https://music-classification.github.io/tutorial/part3_supervised/architectures.html#

MusiCNN

Jordi Pons and Xavier Serra. *“Musicnn: pre-trained convolutional neural networks for music audio tagging”*, 2019

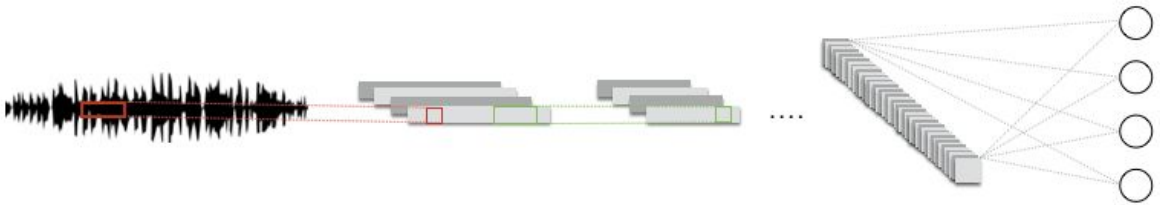


- Manually designed filter shapes for music tagging
- “tall” filters are designed to capture timbral characteristics;
- “long” filters are designed to capture rhythmic patterns and tempo

from https://music-classification.github.io/tutorial/part3_supervised/architectures.html#

Sample-level CNNs

Jongpil Lee, Jiyoung Park, Keunhyoung Luke Kim, and Juhan Nam. *"Sample-level deep convolutional neural networks for music auto-tagging using raw waveforms"*, 2017



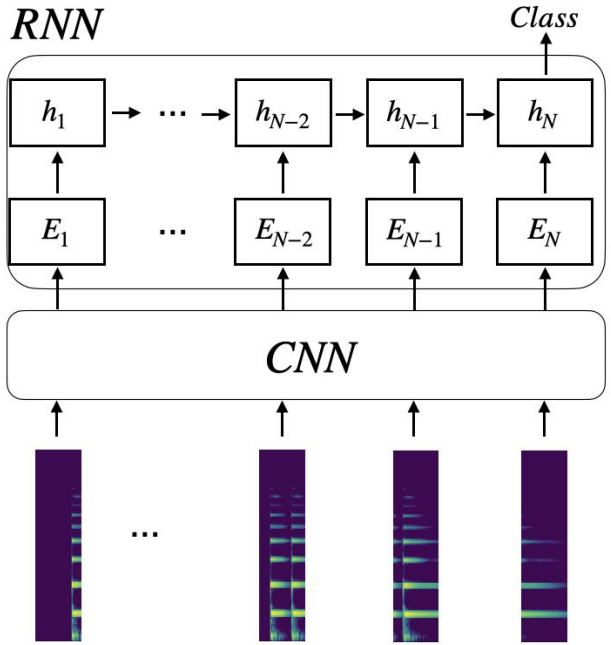
- Uses a 1d-CNN
- In theory, can learn filters similar to what STFT is already capturing (e.g. responses to different wavelengths)

from https://music-classification.github.io/tutorial/part3_supervised/architectures.html#

Convolutional Recurrent Neural Networks (CRNNs)

Keunwoo Choi, György Fazekas, Mark Sandler, and Kyunghyun Cho. *"Convolutional recurrent neural networks for music classification"*, 2017

- Combination of a CNN and RNN
- The CNN captures local characteristics, while the RNN can summarize the sequence
- (**see case study:** Dance Dance Convolution)

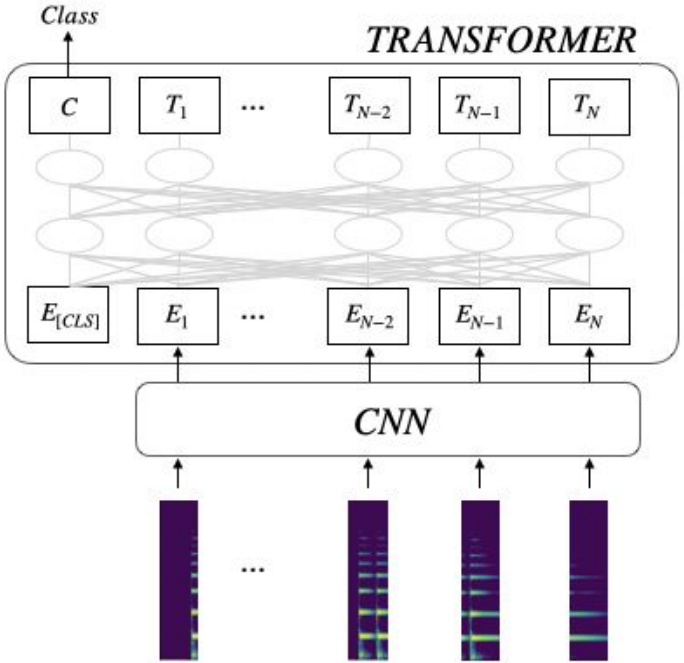


from https://music-classification.github.io/tutorial/part3_supervised/architectures.html#

Music tagging transformer

(e.g.) Minz Won, Keunwoo Choi, and Xavier Serra. *"Semi-supervised music tagging transformer"*, 2019

- Similar to CRNN but uses a transformer (it works better!)
- Can also use it to interpret



from https://music-classification.github.io/tutorial/part3_supervised/architectures.html#

CNN architectures for MIR

Lots of others: this survey is already a few years old

But main thing to understand is how CNNs can be used to capture useful invariances in music and audio, and how CNNs can be combined with sequence models to capture both local and sequential characteristics (though more of this in Module 3)

MIR fundamentals: data augmentation

Data augmentation: In computer vision (for example), it is common to *augment* datasets with transformations of existing instances, e.g. by rotation, translation, cropping, etc.

These transformations modify the image while (in principle) preserving its label; thus we can synthesize a larger dataset, and potentially mitigate overfitting (etc.)

Can also apply transformations / augmentation to music!

Data augmentation

Random crop (by time)

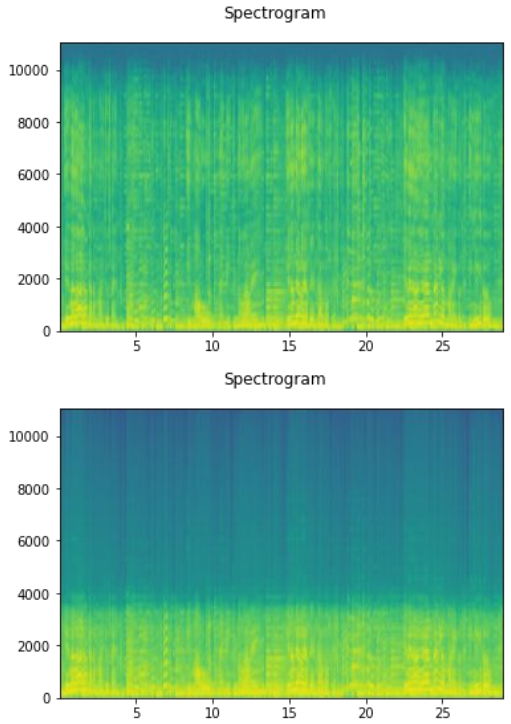
Take a random sample of an audio signal between two timepoints

Data augmentation

Frequency Filter

A frequency filter is applied to the signal:

- Low-pass: keep low frequencies, discard high frequencies
- High-pass: keep high frequencies, discard low frequencies



low-pass (from https://music-classification.github.io/)

Data augmentation

Delay: The signal is delayed by some value and added to the original signal with a volume factor

Comb filter: Delay with a short time and high volume (which causes interference)

Reverb: Apply a (Schroeder) reverberation effect; this gives the illusion that the sound is played in a larger space, in which it takes longer for the sound to reflect

Data augmentation

Pitch shift: Shifts the pitch up or down by semitones without affecting tempo (genre classification, music tagging, chord recognition)

Time stretch: Changes the speed of the audio without altering pitch (rhythm analysis, beat tracking, and dance music recognition)

Background noise injection: Adds ambient or synthetic noise to simulate real-world conditions (music tagging and music transcription)

Random cropping / segment slicing: Takes random chunks from longer tracks

SpecAugment: Applies time/frequency masking to spectrograms (music tagging architectures like PANNs, AST, MAEST)

Data augmentation

Generally speaking, data is augmented via lots of operations that change the acoustic qualities of the piece, but shouldn't change its *label*

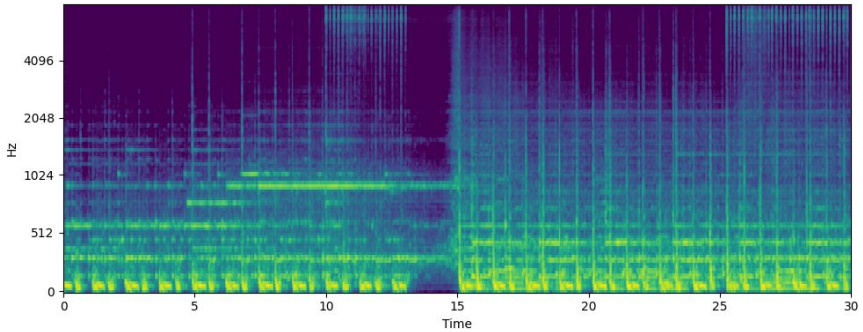
In this way, we will learn models that are *invariant* to these irrelevant details (as opposed to overfitting on them)

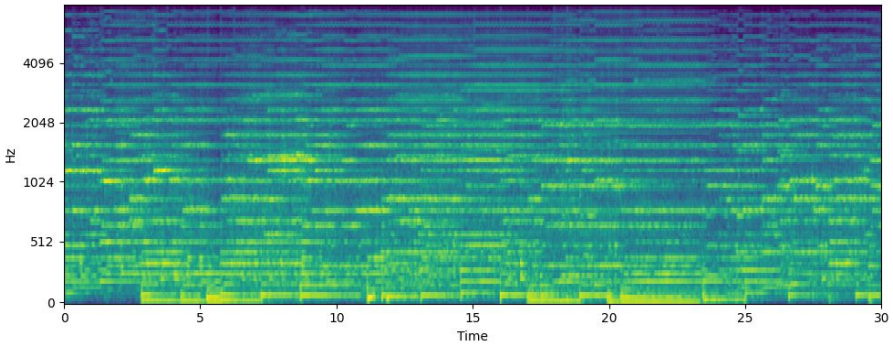
Worked example: genre classification

Try it out: a simple architecture is implemented in **workbook2.ipynb**

(and we'll expand upon this in the homework)

Worked example: genre classification





Label: 0 (not classical)

Label: 1 (classical)

Note: the audio samples are AI-generated music (Copet et al., 2023)

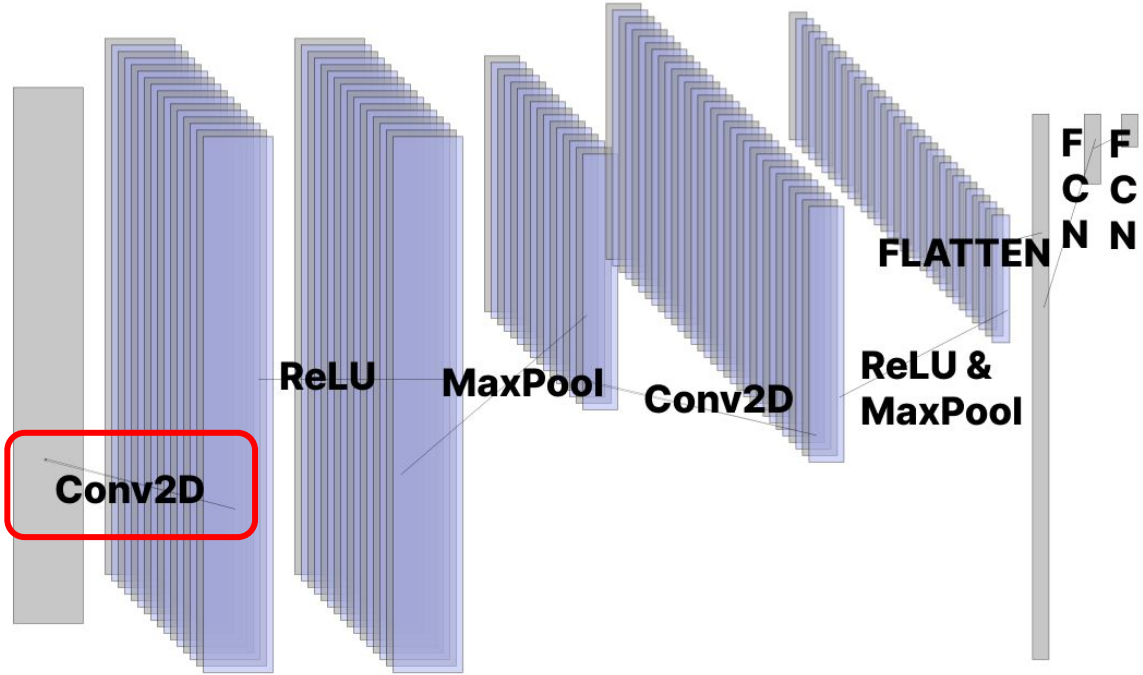
Worked example...

```
class MelSpectrogramClassifier(nn.Module):
    def __init__(self, input_channels=1, n_mels=128, fixed_length=938, num_classes=2):
        super(MelSpectrogramClassifier, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(32 * (n_mels // 4)  * (fixed_length // 4), 128)
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = x.transpose(2, 3)
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv1(x)))

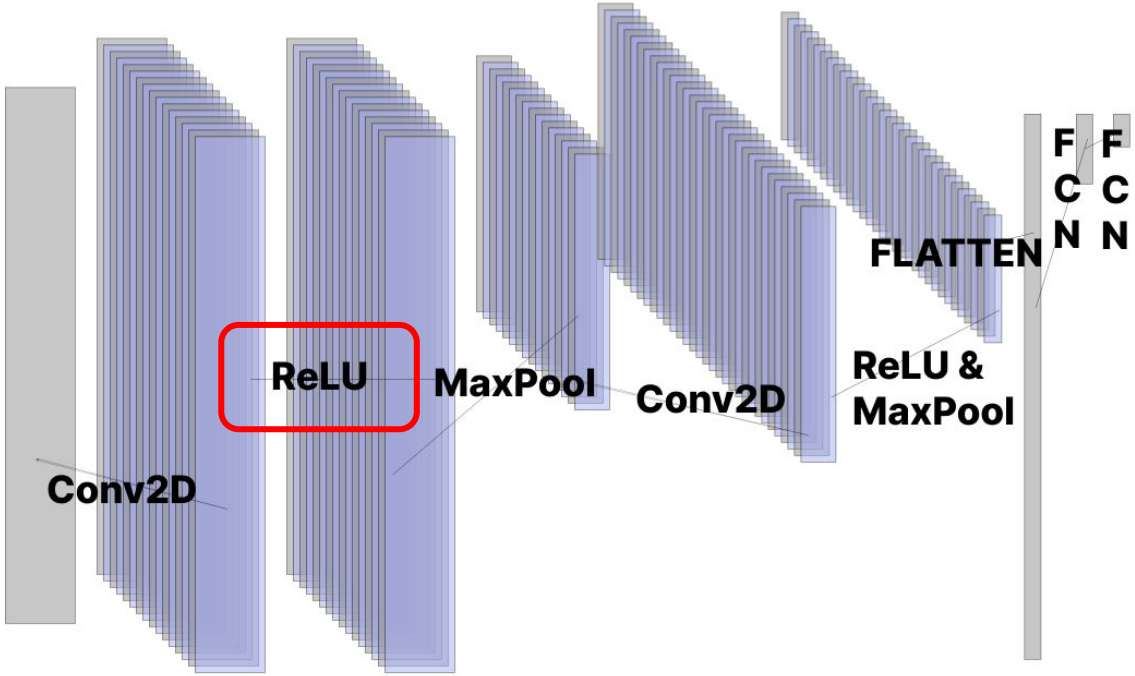
        x = x.view(x.shape[0], -1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

Worked example...



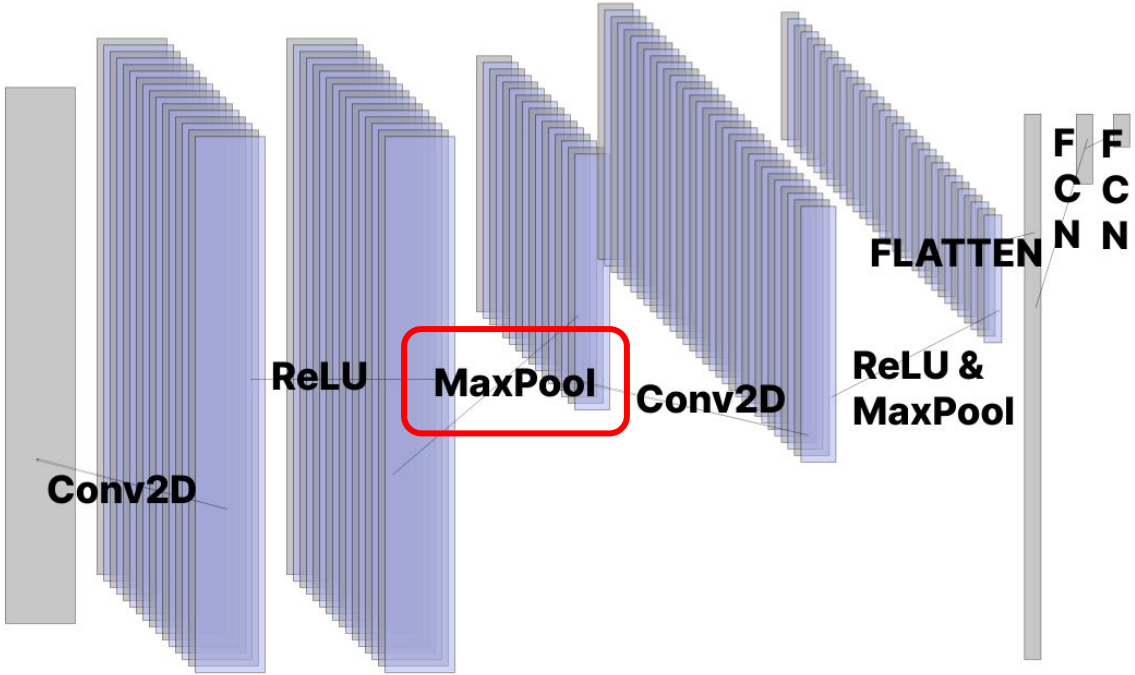

```
nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1,
padding=1)
```

Worked example...



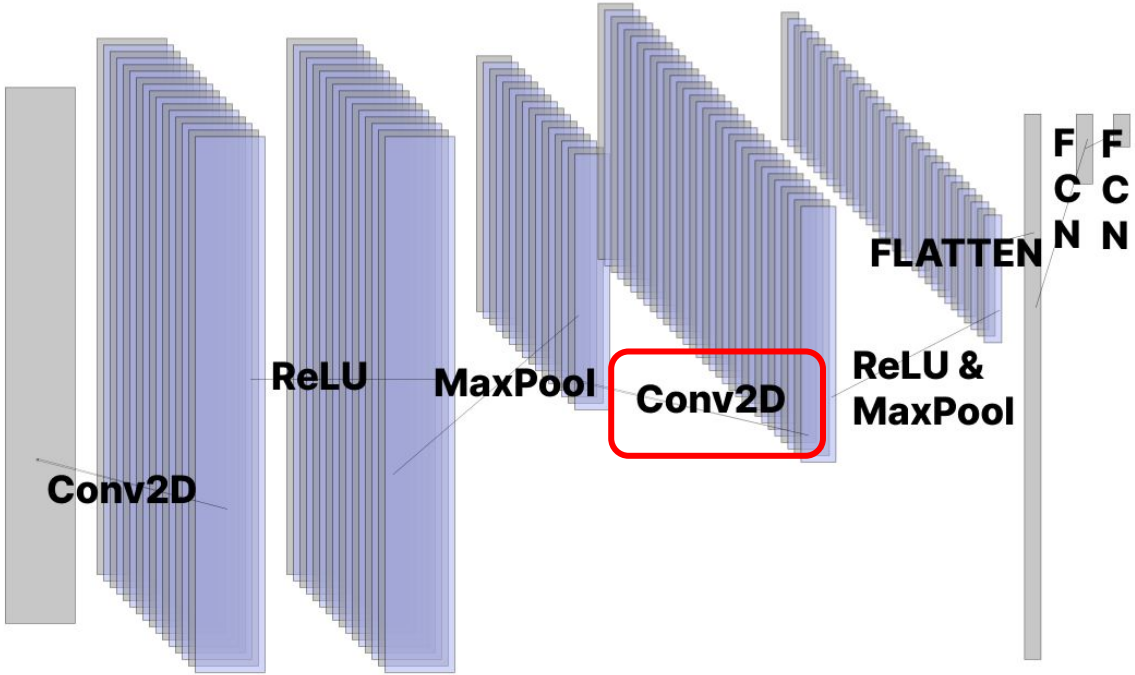
`nn.ReLU()`

Worked example...



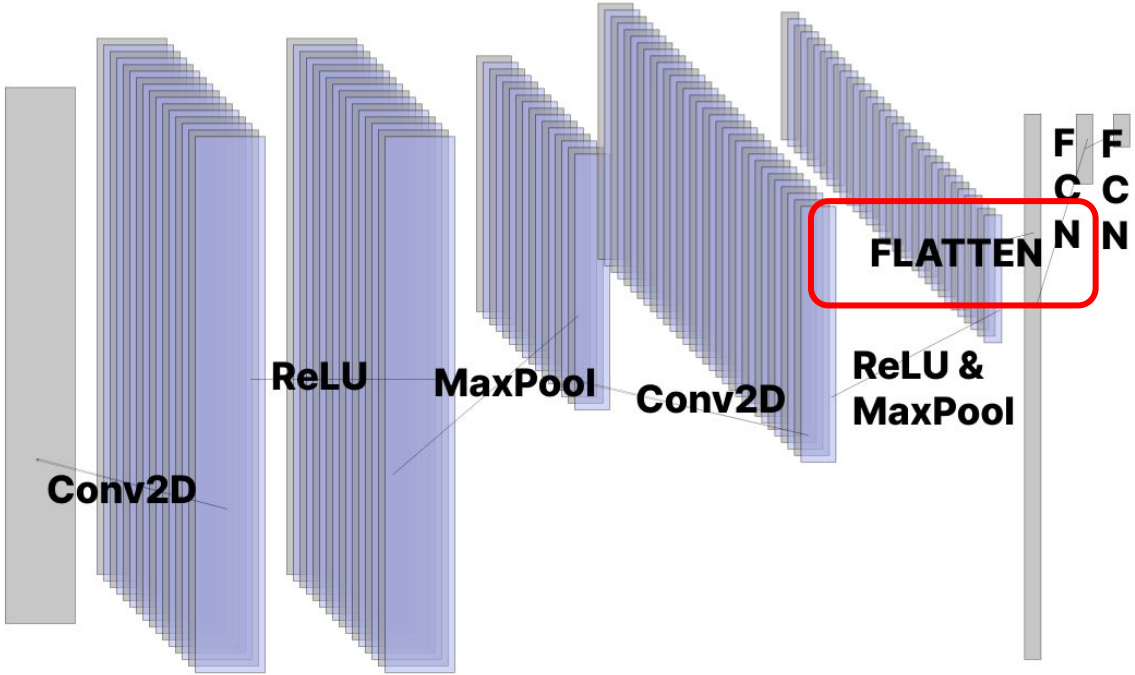

```
nn.MaxPool2d(kernel_size=2, stride=2)
```

Worked example...



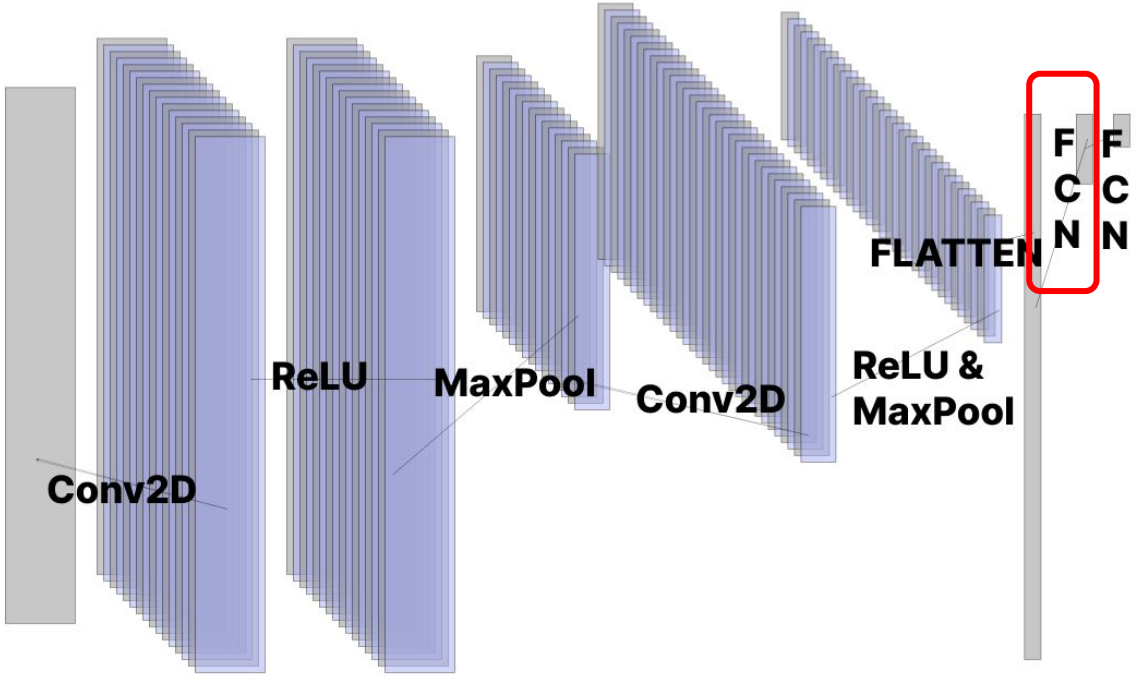

```
nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1,
padding=1)
```

Worked example...



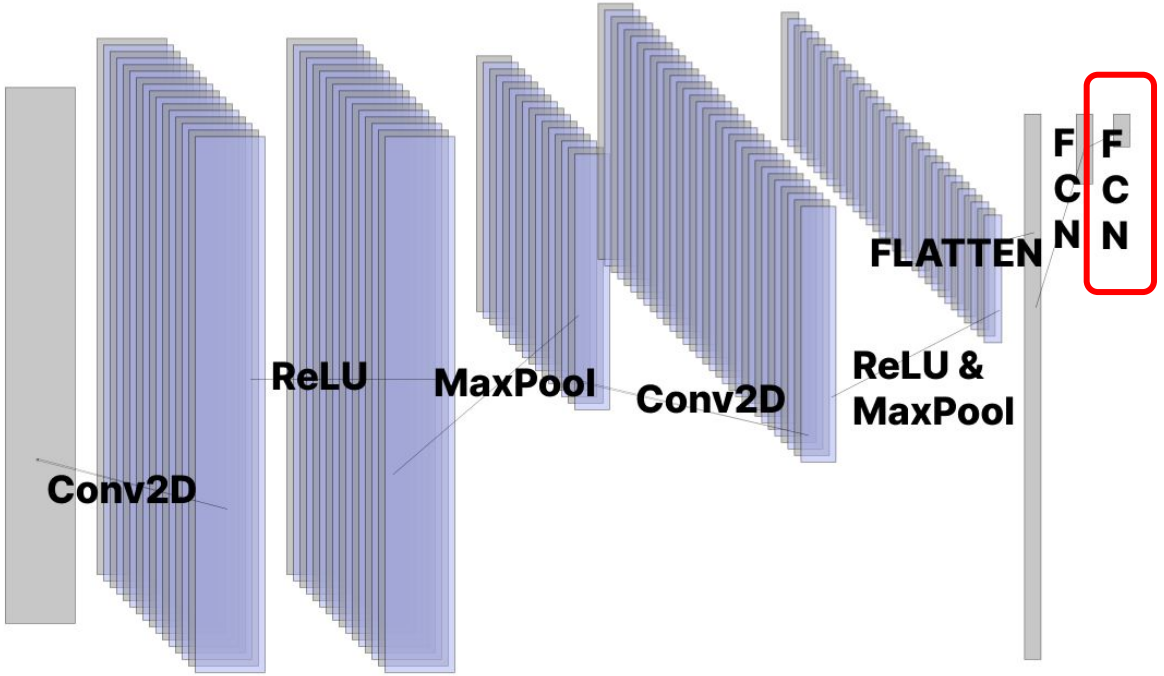

```
x.view(x.shape[0], -1)
```

Worked example...



`nn.Linear(... ,128)`

Worked example...



`nn.Linear(128, 2)`

Take-homes

- By now, you should have an overall sense of the *pipeline* of building an MIR system (for some tagging-ish classification task)
- Try training a simple model, and implementing some data augmentations to try and improve performance (Homework?)

Predictive pipelines for music & MIR

2.5: A few more MIR tasks
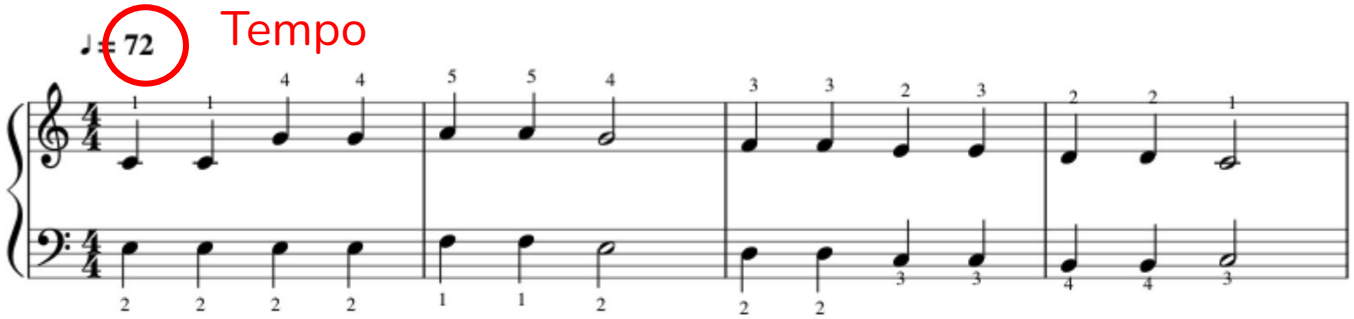
More MIR tasks

I won’t cover these in depth: we’ve covered the basics of feature extraction which can then be used for a lot of related tasks

But, I want to mention a few applications that are part of MIR, but not quite like those above, mostly to give a sense of how the fundamental techniques might be adapted to slightly different settings

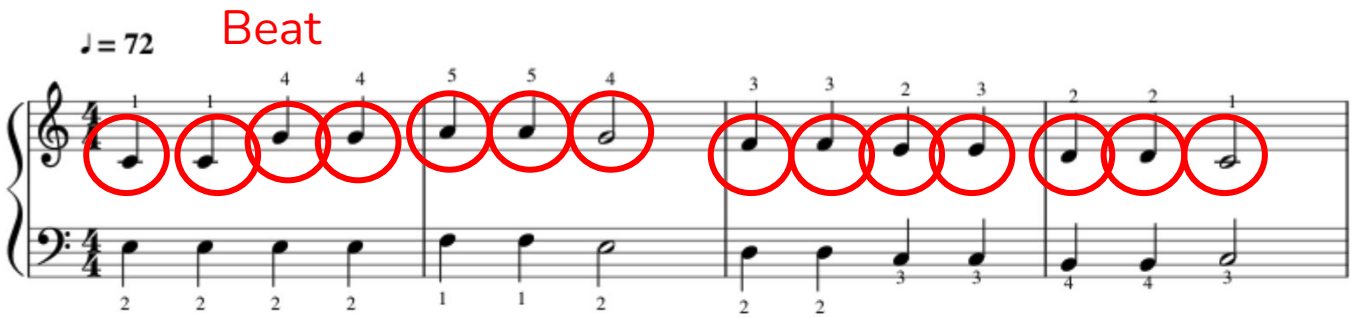
Beat-tracking

- Tempo Estimation: Estimating the value of BPM (Beats per minute)
- Beat Estimation: Estimating a sequence of beats
- Downbeat Estimation: Estimating a sequence of downbeats (the first beats of each bar)
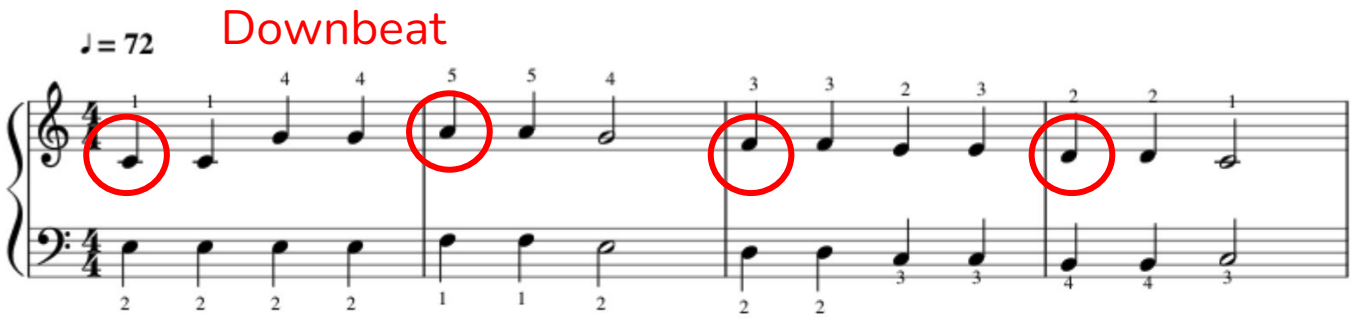


Beat-tracking

- Tempo Estimation: Estimating the value of BPM (Beats per minute)
- Beat Estimation: Estimating a sequence of beats
- Downbeat Estimation: Estimating a sequence of downbeats (the first beats of each bar)

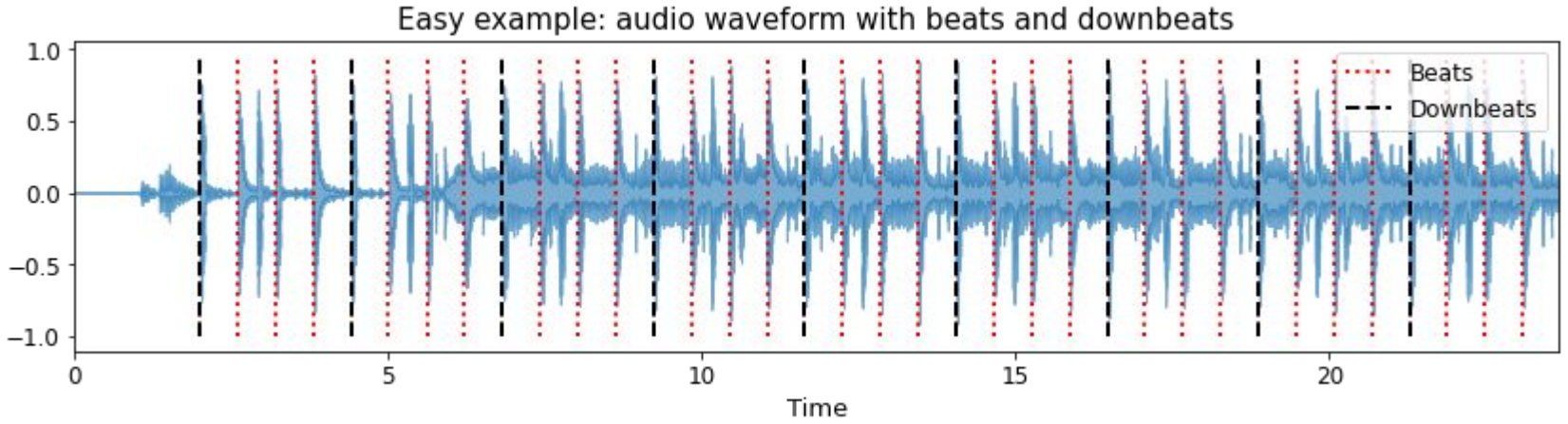


Beat-tracking

- Tempo Estimation: Estimating the value of BPM (Beats per minute)
- Beat Estimation: Estimating a sequence of beats
- Downbeat Estimation: Estimating a sequence of downbeats (the first beats of each bar)



Beat-tracking - Sound Example



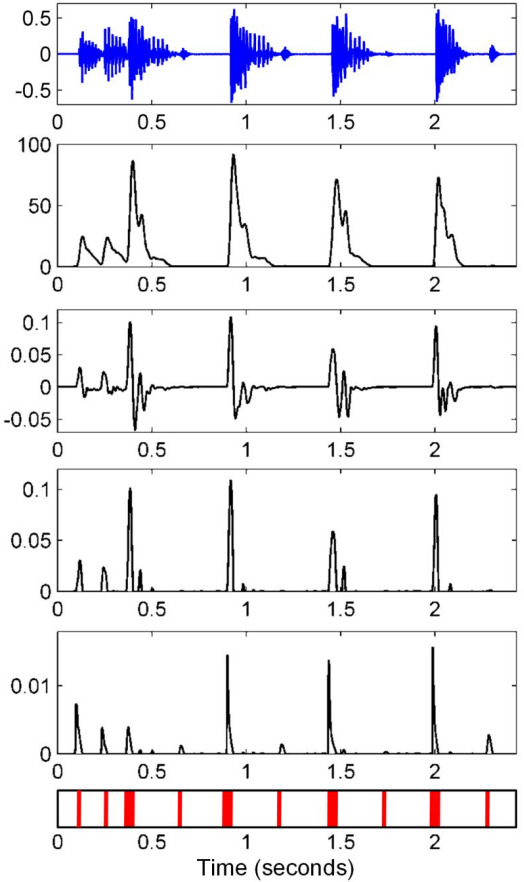
https://tempobeatdownbeat.github.io/tutorial/ch2_basics/definition.html

Beat-tracking

See also e.g. https://www.audiolabs-erlangen.de/resources/MIR/FMP/C6/C6S1_PeakPicking.html

which has a great description of beat-tracking (or “onset detection”)

The solution described there is more “traditional”: based on “peak picking” and not actually involving ML



Beat-tracking: CNN

Böck, Sebastian, and Matthew EP Davies. "Deconstruct, Analyse, Reconstruct: How to improve Tempo, Beat, and Downbeat Estimation." ISMIR. 2020.

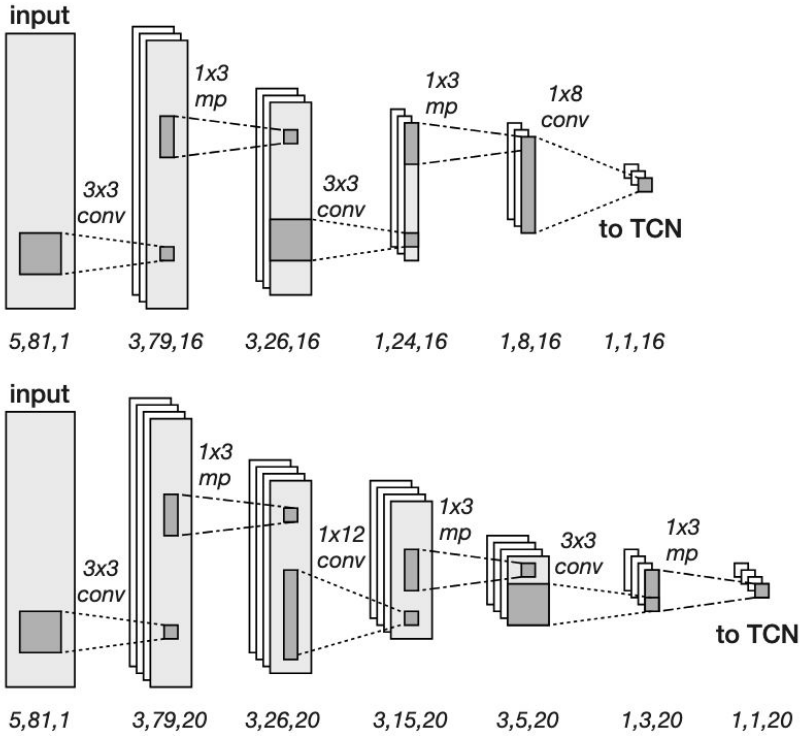


Figure 1: Comparison of the convolution (*conv.*) and max pooling (*mp*) layers. The architecture from [20, 22] (top). Our proposed architecture (bottom). The dimensions of the tensors are shown below each layer.

Chord recognition

Having separated the music into beats, the goal of **chord recognition** is to predict the underlying chords from the raw audio segments

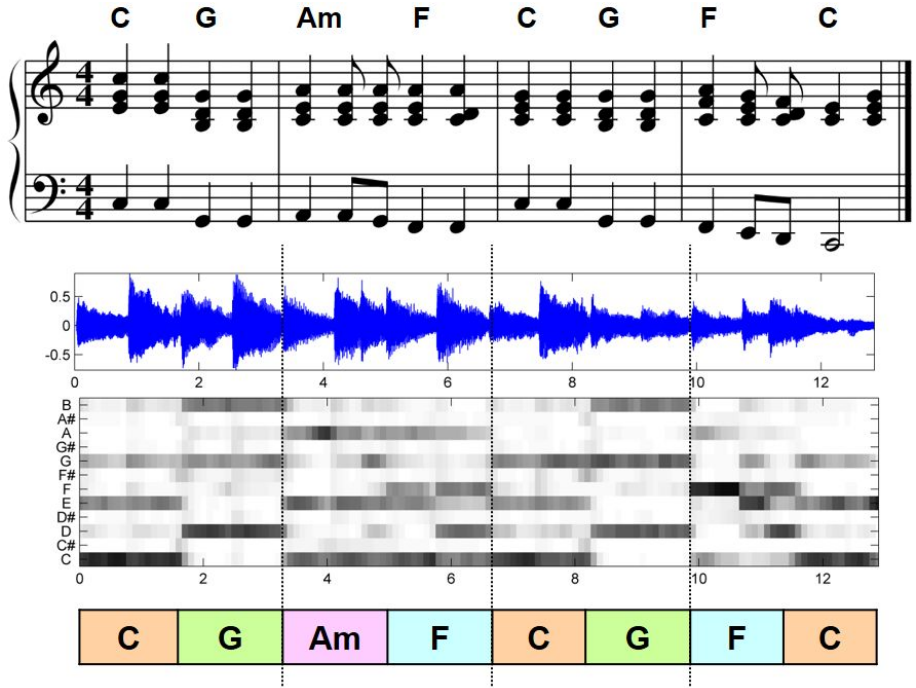


Figure 5.1 from [Müller, FMP, Springer 2015]

https://www.audiolabs-erlangen.de/resources/MIR/FMP/C5/C5S2_ChordRec_Templates.html

Music synchronization: Dynamic Time Warping (DTW)

What is Dynamic Time Warping? Finding the optimal alignment between two temporal sequences by stretching or compressing them along the time axis. The lengths of sequences are often different from one another.

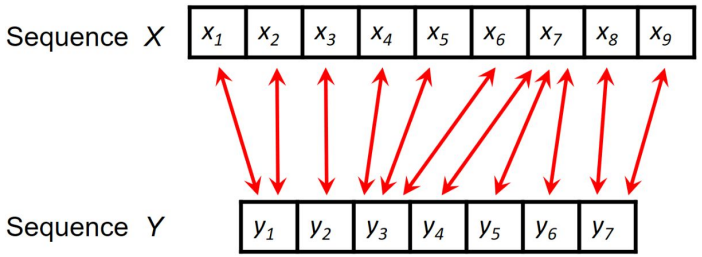
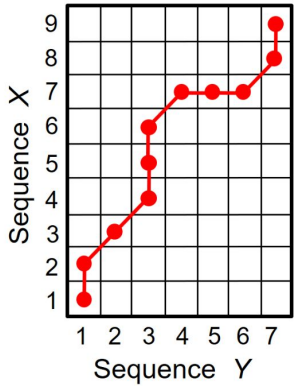


Figure 3.12 from [Müller, FMP, Springer 2015]



see https://www.audiolabs-erlangen.de/resources/MIR/FMP/C3/C3S2_DTWbasic.html

Dynamic Time Warping (DTW)

DTW application: cover song identification

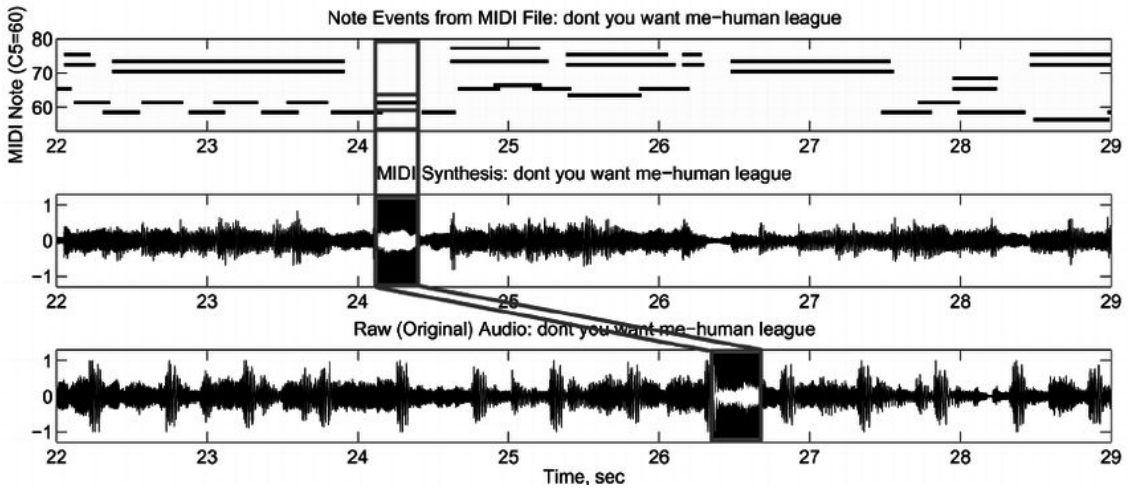
- DTW enables matching between different versions of the same song, even with:
  - Different tempos
  - Key changes
  - Arrangement variations
- Simplified Process: Extract chroma features or mel-spectrograms
  - Apply DTW to find optimal alignment
  - Calculate similarity score
- Real-world application: Shazam-like services for cover song detection



Dynamic Time Warping (DTW)

DTW application: music alignment

Dong et al. (2022) used dynamic time warping to align a music score (synthesized audio) with a music performance recording



https://www.researchgate.net/figure/Goal-of-MIDI-Alignment-The-upper-graph-is-the-transcription-of-the-first-verse-of-Dont_fig1_2882342

Take-homes

Hopefully you can see that we've just scratched the surface of a big topic

Time-permitting, we'll revisit MIR in **Module 5,** investigating various applications that don't quite fit the "*train a CNN to make a prediction*" mold

Examples include:

- Music recommender systems, content-based retrieval systems
- Playlist generation and continuation
- Lyric translation (but more related to Modules 3/4)
- (etc.)

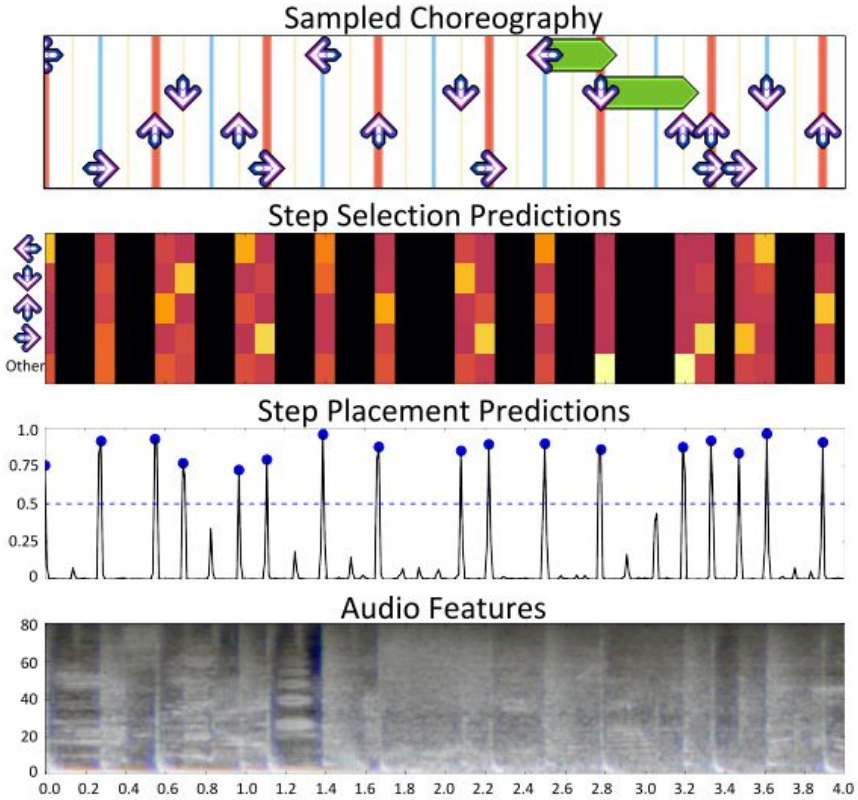
Predictive pipelines for music & MIR

Case study: Dance Dance Convolution

Dance Dance Convolution (2017)

Proposed and presented the task of ***learning to choreograph***

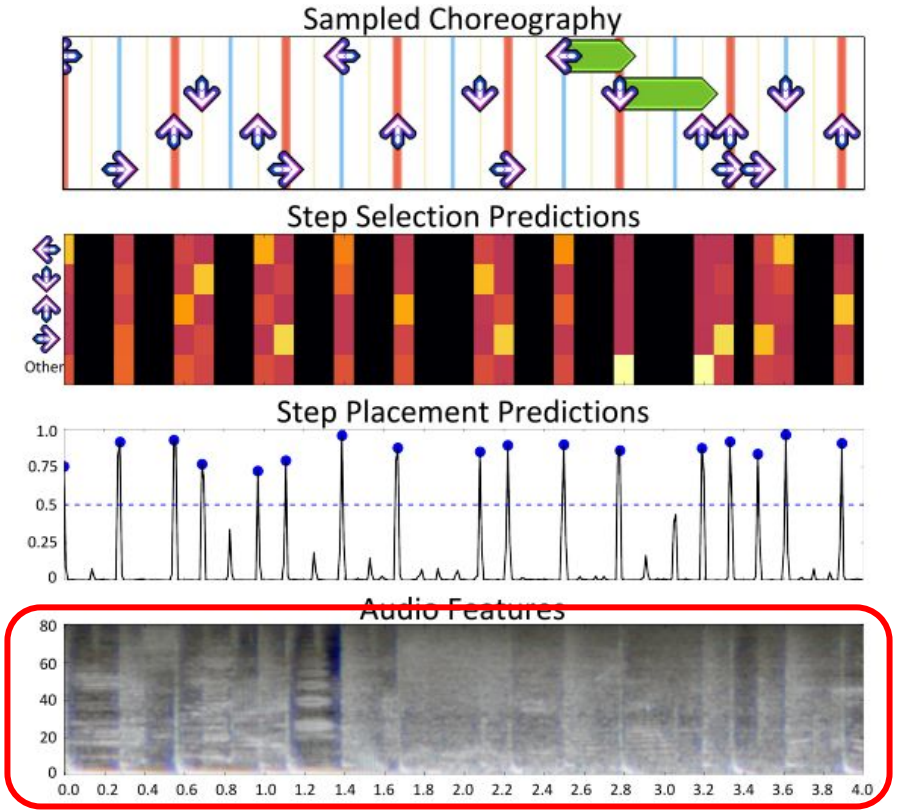
What is *learning to choreograph?*: Learning to produce a step chart (a sequence of arrow steps that players need to hit during a song) in Dance Dance Revolution given music audio



Dance Dance Convolution (2017)

Audio representation

1. Applied STFT with three different window lengths
2. Converted it into mel-spectrogram (with 80 frequency bins)
3. Scaled the magnitude logarithmically in order to better represent human perception



Dance Dance Convolution (2017)

Audio representation

1. Applied STFT with three different window lengths (23ms, 46ms, 93ms)
    - **Shorter window sizes:** low-level features such as pitch and timbre while
    - **Larger window sizes:** high-level features such as melody and rhythm
2. Converted into mel-spectrogram (with 80 frequency bins)
3. Scaled the magnitude logarithmically in order to better represent human perception

Dance Dance Convolution (2017)

Audio representation

1. Applied STFT
2. Converted it into mel-spectrogram (with 80 frequency bins)
3. Scaled the magnitude logarithmically in order to better represent human perception

```
import numpy as np
import librosa

audio_signal, sr = librosa.load("audio_signal", 44100)
stft = librosa.stft(audio_signal,
                    n_fft=2048,
                    hop_length=512)
magnitude_spectrogram = np.abs(stft)
mel_spectrogram = librosa.feature.melspectrogram(
       S=magnitude_spectrogram,
       sr=sr,
       n_mels=80,
       fmax=sr/2)
log_mel_spectrogram = librosa.power_to_db(mel_spectrogram,
ref=np.max)
```

Dance Dance Convolution (2017)

Audio representation

1. Applied STFT
2. Converted it into mel-spectrogram (with 80 frequency bins)
3. Scaled the magnitude logarithmically in order to better represent human perception

```
import numpy as np
import librosa

audio_signal, sr = librosa.load("audio_signal", 44100)
stft = librosa.stft(audio_signal,
                    n_fft=2048,
                    hop_length=512)
magnitude_spectrogram = np.abs(stft)
mel_spectrogram = librosa.feature.melspectrogram(
       S=magnitude_spectrogram,
       sr=sr,
       n_mels=80,
       fmax=sr/2)
log_mel_spectrogram = librosa.power_to_db(mel_spectrogram,
ref=np.max)
```

Dance Dance Convolution (2017)

Audio representation

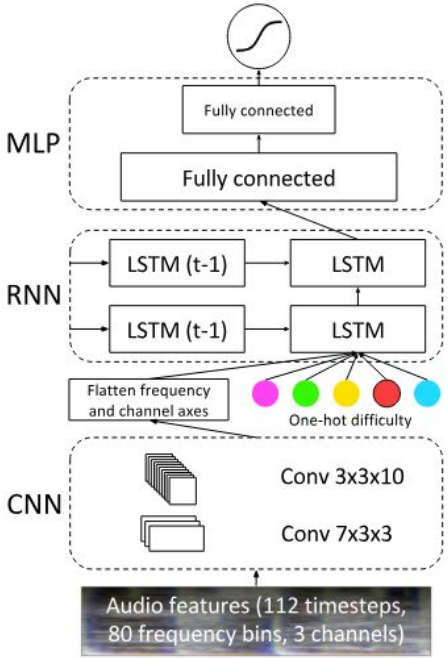
1. Applied STFT
2. Converted it into mel-spectrogram (with 80 frequency bins)
3. Scaled the magnitude logarithmically in order to better represent human perception

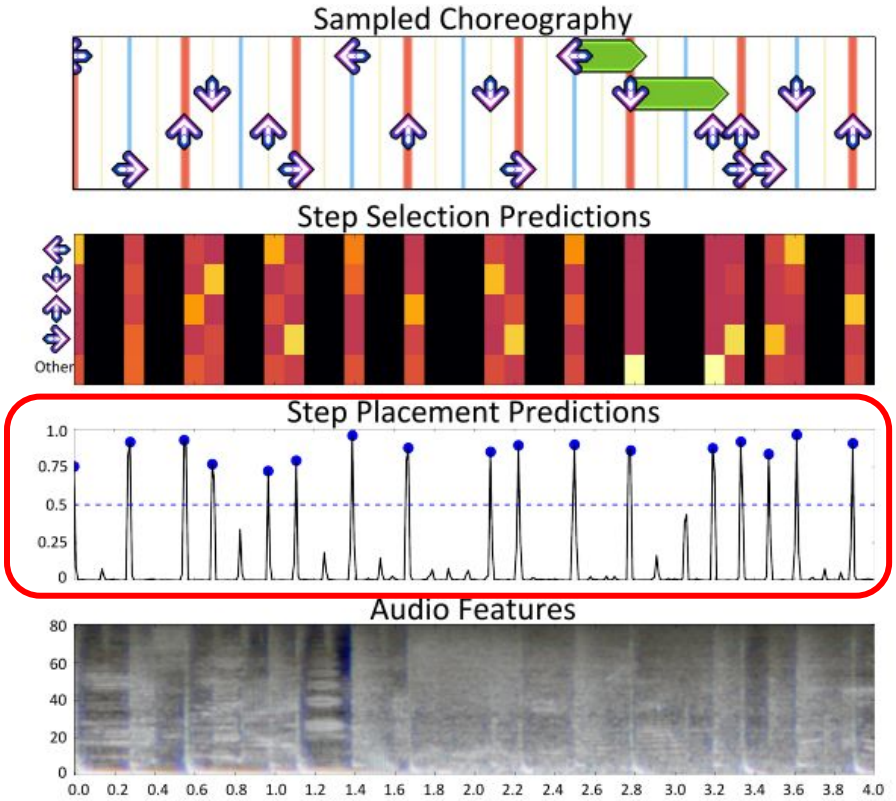
```
import numpy as np
import librosa

audio_signal, sr = librosa.load("audio_signal", 44100)
stft = librosa.stft(audio_signal,
                    n_fft=2048,
                    hop_length=512)
magnitude_spectrogram = np.abs(stft)
mel_spectrogram = librosa.feature.melspectrogram(
       S=magnitude_spectrogram,
       sr=sr,
       n_mels=80,
       fmax=sr/2)
log_mel_spectrogram = librosa.power_to_db(mel_spectrogram,
ref=np.max)
```

Dance Dance Convolution (2017)

Step Placemer





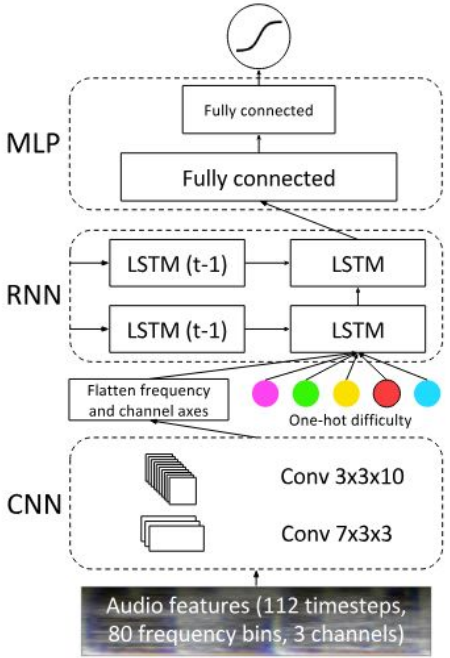
Dance Dance Convolution (2017)

Step placement

Built upon onset detection model

1. Applied CNN to mel-spectrograms
2. Applied RNN to the output of CNN
3. Applied Fully Connected Layers to the output of RNN

(we'll look at RNN models in Module 3)



Dance Dance Convolution (2017)

Step selection

Predicting next step based on current step

- Also based on a language model (LSTM)
- Has to learn “semantics” e.g., steps should follow common patterns, and should avoid requiring the dancer to turn around (etc.)

