Audio Event Detection via Deep Learning in Python

Coop, Ph.D.
Director of Machine Learning
phData, inc.

About the author

Robert Coop, Ph.D. → Coop

Director of Machine Learning, phData

Machine learning since 2008

https://www.linkedin.com/in/rcoop/

Agenda

01 **Introduction**

Objective and motivation

02 **Representing audio**

Time/signal domains, spectrograms

03 **Deep learning for audio**

Convolutional neural networks, VGG, VGGish

04 **Use case: Recognizing commands**

Publicly available data for research

01

Introduction

Objective and motivation

Objective

What should you take from this?

Enable the listener to use open source tools and pre-trained neural networks for audio analytics projects

Motivation

Scenario: audio event detection

Audio recorded from security devices and classified as hostile or non-hostile

Use best practices

Prefer approaches that have been researched and tested

Create a solution quickly

Get from zero to prototype as fast as possible

02

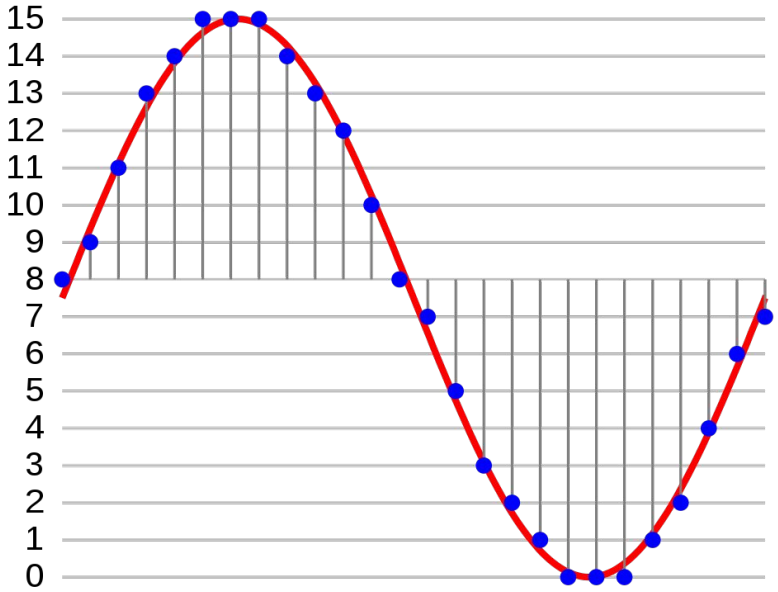
Representing audio

Digital audio, frequency scales, time and frequency domains, spectrograms

Digital audio

Waveform Audio File (WAV)

- Analog-to-digital converter (ADC)
- Pulse-code modulation (PCM)
  - Sampling rate
  - Bit depth



Frequency scales

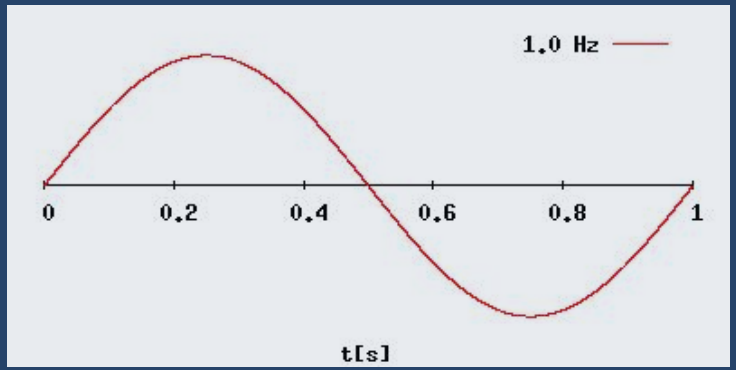
Hertz (Hz) scale

Based on physical characteristics of sound wave

One Hertz = one cycle per second

Middle C (C_4) - 261.626 Hz

Treble C (C_5) - 523.251 Hz



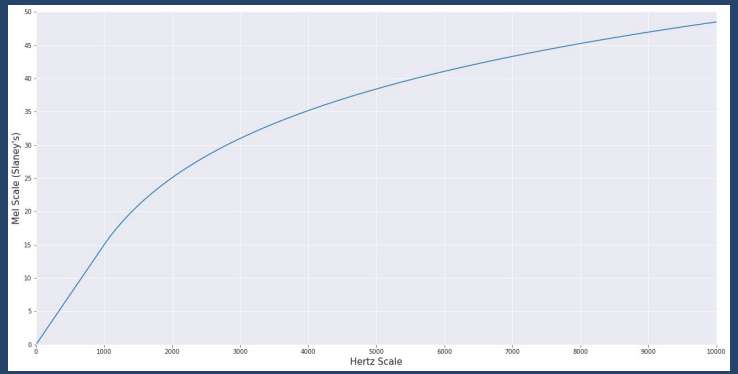
Mel scale

Based on human perception of sound

Several different formulas exist

Slaney, M. Auditory Toolbox Version 2, Technical Report #1998-010 Interval Research Corporation, 1998.

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right)$$



Frequency interval comparison - 100 Hz

100 Hz

200 Hz

300 Hz

400 Hz

6000 Hz

6100 Hz

6200 Hz

6300 Hz

Frequency interval comparison - 1.5 Mel

1.5 Mel (100 Hz)

3 Mel (200 Hz)

4.5 Mel (300 Hz)

6 Mel (400 Hz)

~41 Mel (6000 Hz)

~42.5 Mel (6652 Hz)

~44 Mel (7374 Hz)

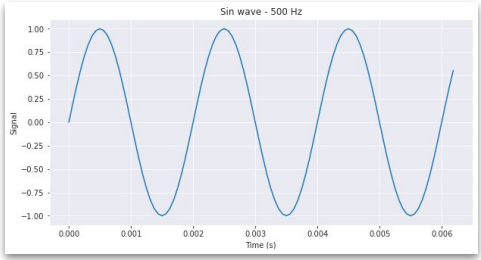
~45.5 Mel (8176 Hz)

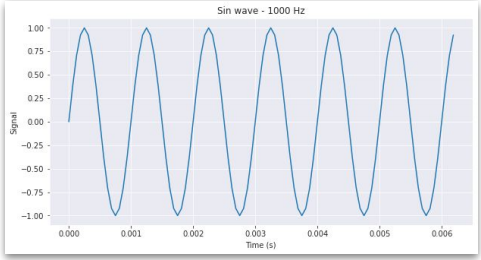
Time domain and frequency domain

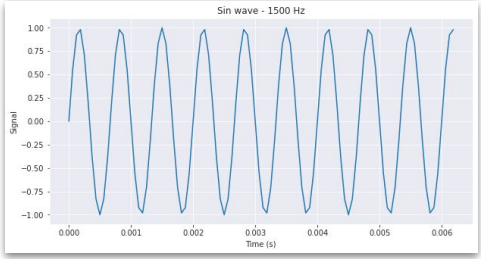
$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i x \xi}\,dx$$

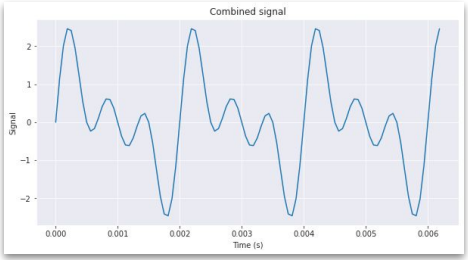
Fourier transform

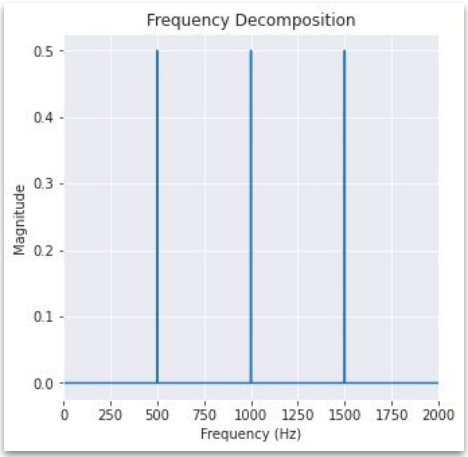
Decomposes signal into constituent frequencies









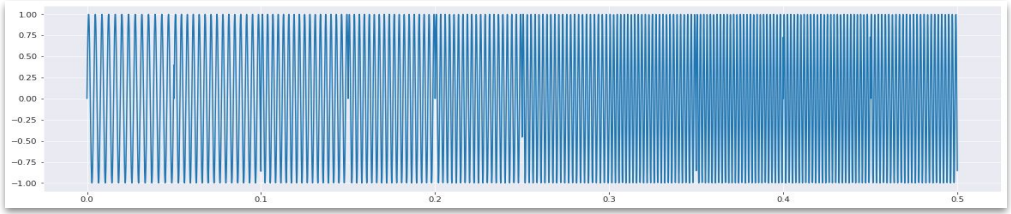


Time domain and frequency domain

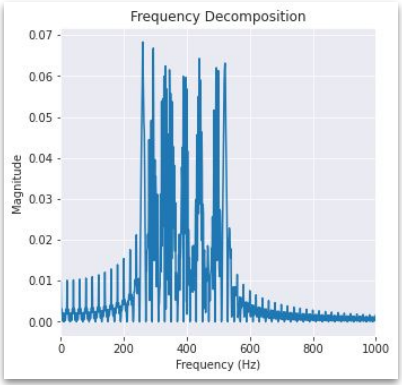
Spectrograms

Frequency decompositions over time

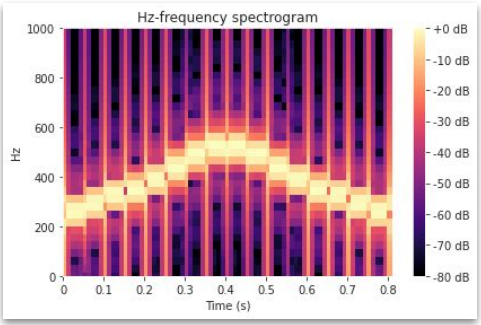
C-major scale



Frequency decomposition is a poor representation



Spectrograms show progressions over time



03

Deep learning for audio

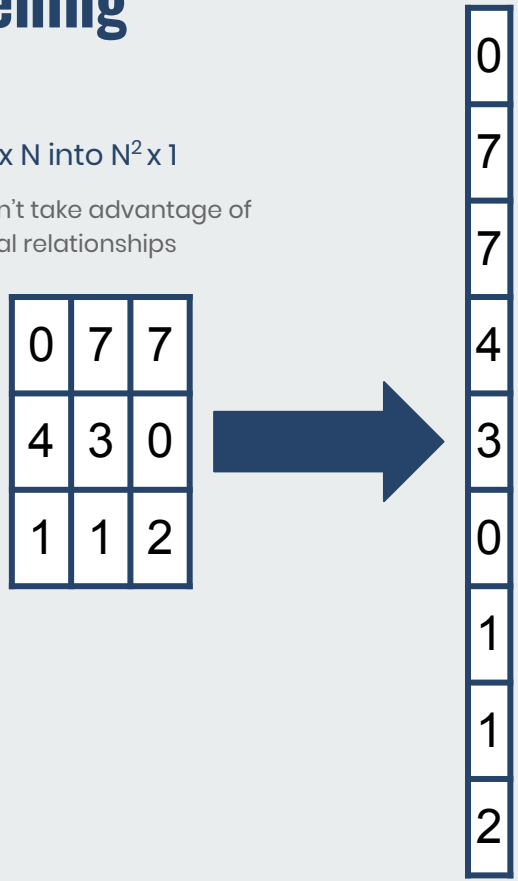
Convolutional neural networks,
VGG16, VGGish

Processing spatial data

Flattening

Flatten N x N into N^2 x 1

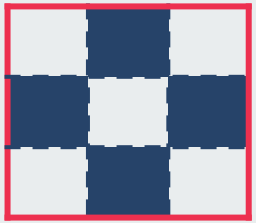
Doesn't take advantage of spatial relationships



Processing spatial data

Convolution

Use kernels to reduce dimensionality while retaining spatial relationships



Kernel

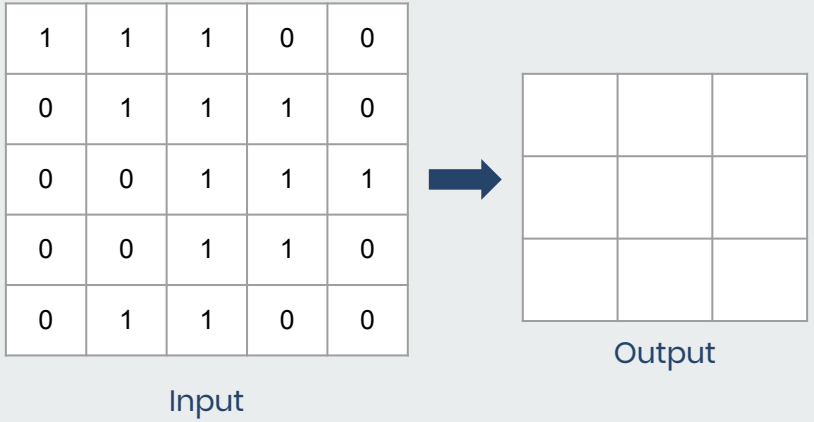
1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input

Processing spatial data

Convolution

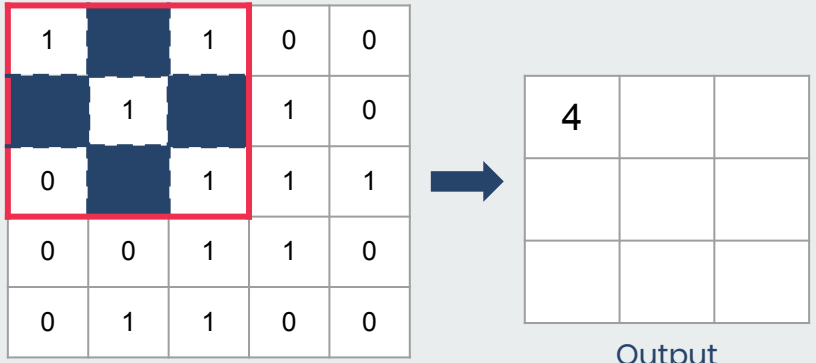
Use kernels to reduce dimensionality while retaining spatial relationships



Processing spatial data

Convolution

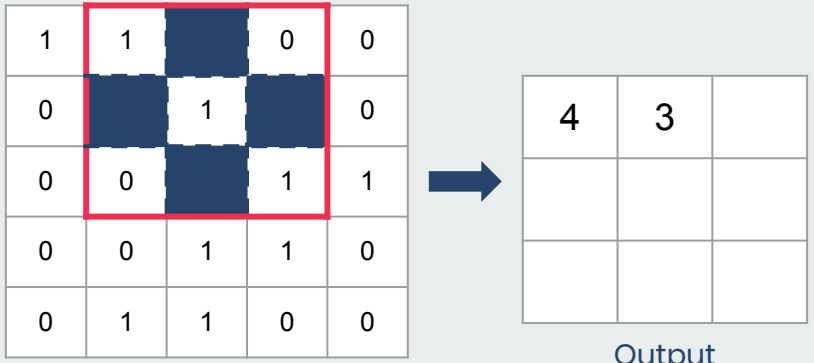
Use kernels to reduce dimensionality while retaining spatial relationships



Processing spatial data

Convolution

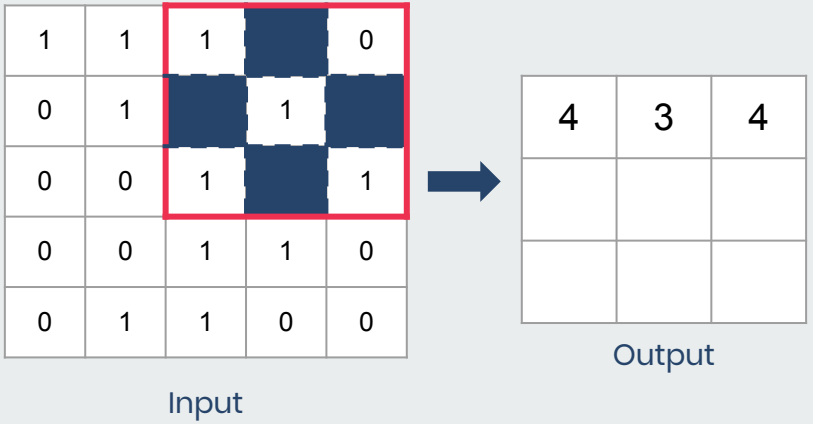
Use kernels to reduce dimensionality while retaining spatial relationships



Processing spatial data

Convolution

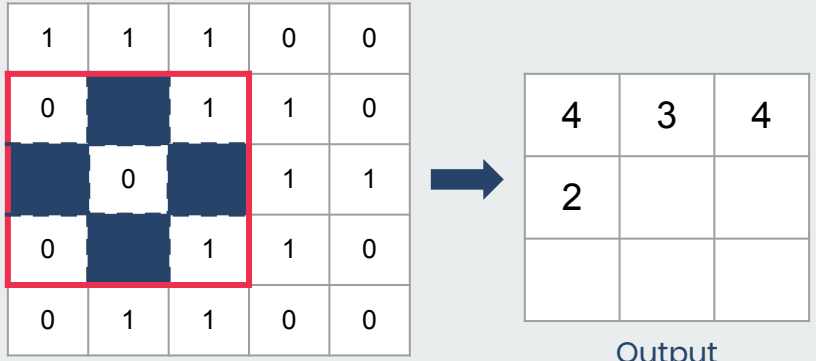
Use kernels to reduce dimensionality while retaining spatial relationships



Processing spatial data

Convolution

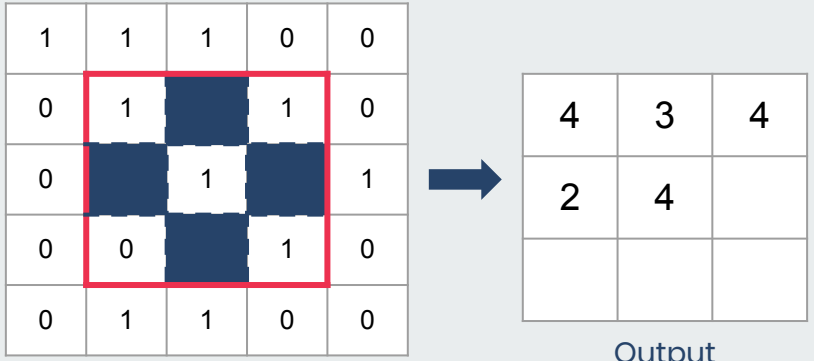
Use kernels to reduce dimensionality while retaining spatial relationships



Processing spatial data

Convolution

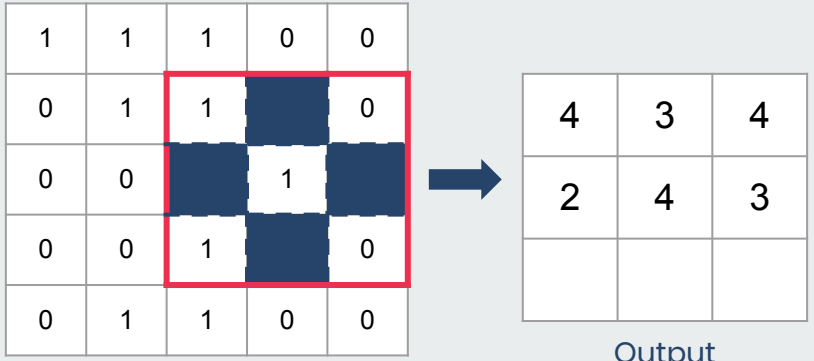
Use kernels to reduce dimensionality while retaining spatial relationships



Processing spatial data

Convolution

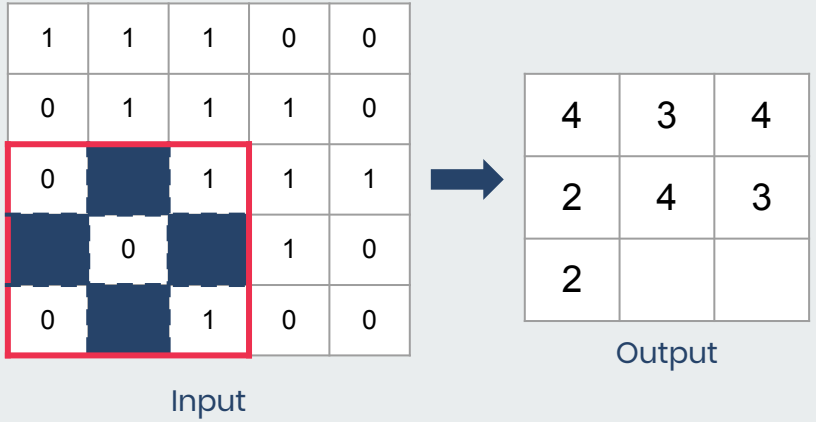
Use kernels to reduce dimensionality while retaining spatial relationships



Processing spatial data

Convolution

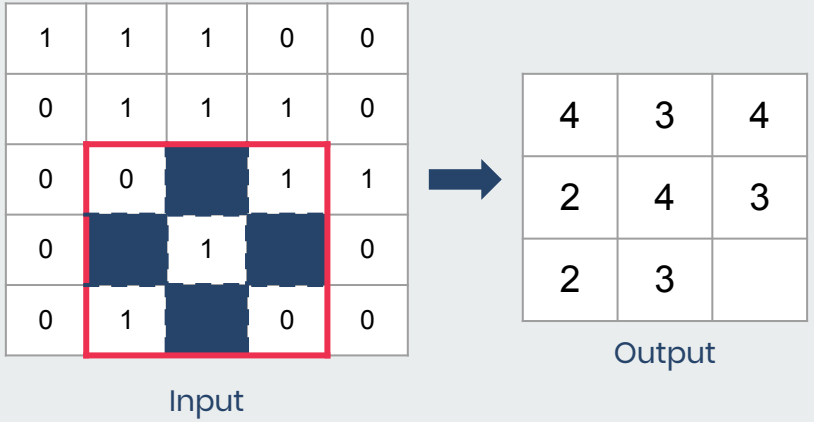
Use kernels to reduce dimensionality while retaining spatial relationships



Processing spatial data

Convolution

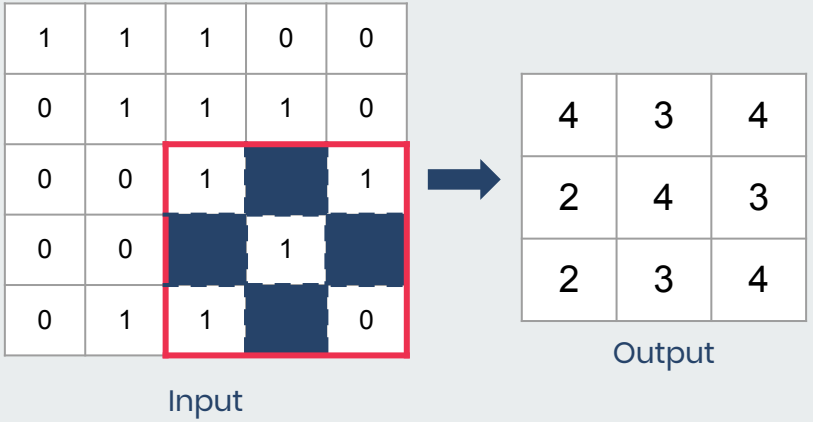
Use kernels to reduce dimensionality while retaining spatial relationships



Processing spatial data

Convolution

Use kernels to reduce dimensionality while retaining spatial relationships



Processing spatial data

Convolution

Use kernels to reduce dimensionality while retaining spatial relationships

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input

4	3	4
2	4	3
2	3	4

Output

Processing spatial data

Pooling

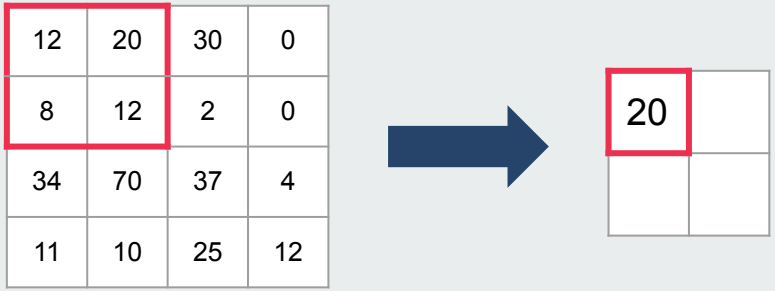
Max (or mean) pooling extracts dominant features while suppressing noise from noisy activations

12	20	30	0
8	12	2	0
34	70	37	4
11	10	25	12

Processing spatial data

Pooling

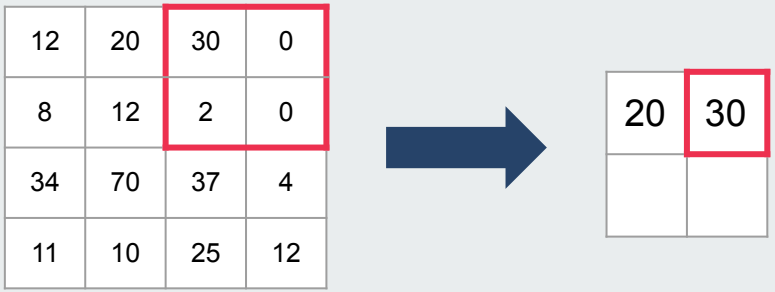
Max (or mean) pooling extracts dominant features while suppressing noise from noisy activations



Processing spatial data

Pooling

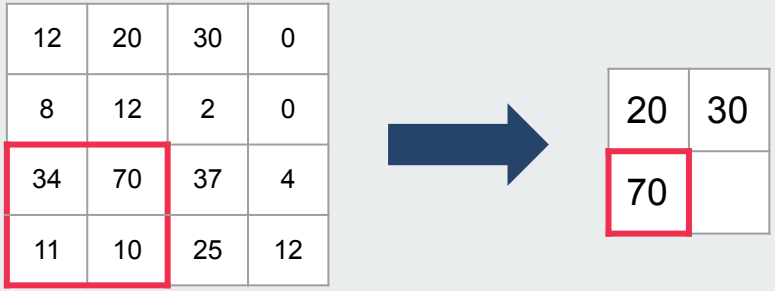
Max (or mean) pooling extracts dominant features while suppressing noise from noisy activations



Processing spatial data

Pooling

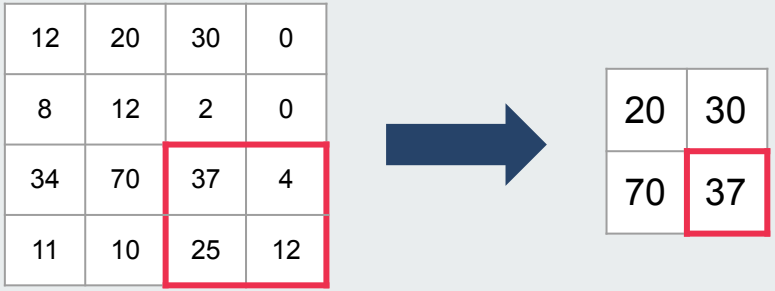
Max (or mean) pooling extracts dominant features while suppressing noise from noisy activations



Processing spatial data

Pooling

Max (or mean) pooling extracts dominant features while suppressing noise from noisy activations



Processing spatial data

Pooling

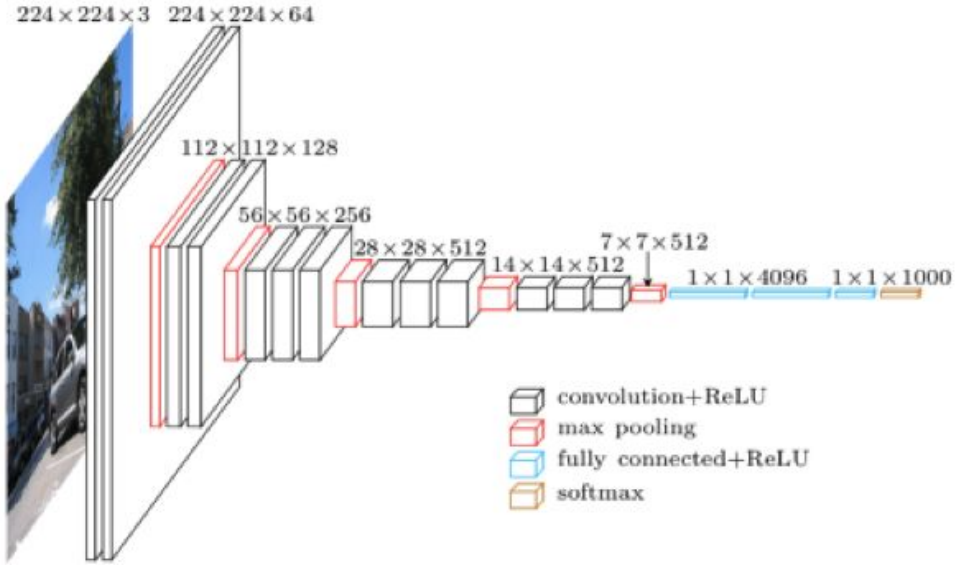
Max (or mean) pooling extracts dominant features while suppressing noise from noisy activations

12	20	30	0
8	12	2	0
34	70	37	4
11	10	25	12

20	30
70	37

VGG[1]

A convolutional deep neural network for image classification



Architecture

11-19 layers, adjustable weights

Advantages

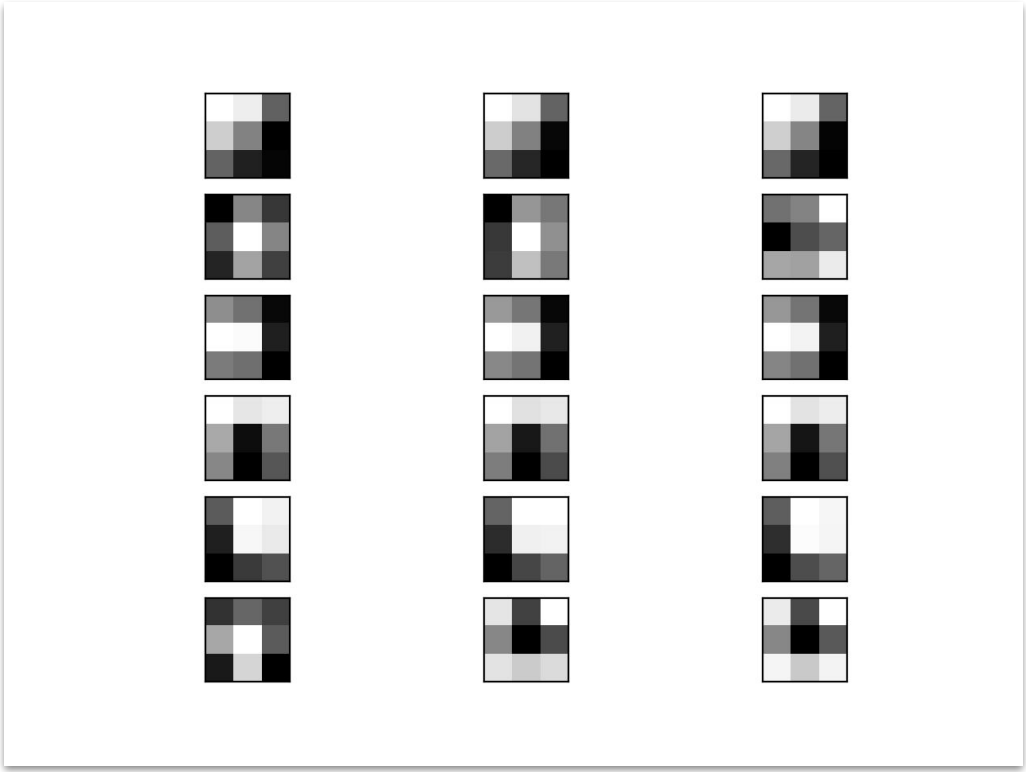
Consistently performs very well on benchmarks

Several pre-trained versions exist

[1] Simonyan, Karen, and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." arXiv preprint arXiv:1409.1556 (2014).

Visualizing Convolutional Layers of VGG16 trained on ImageNet[1]

Kernel visualization - first 6 filters



[1] Browinee, Jason. "How to Visualize Filters and Feature Maps in Convolutional Neural Networks."
https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/

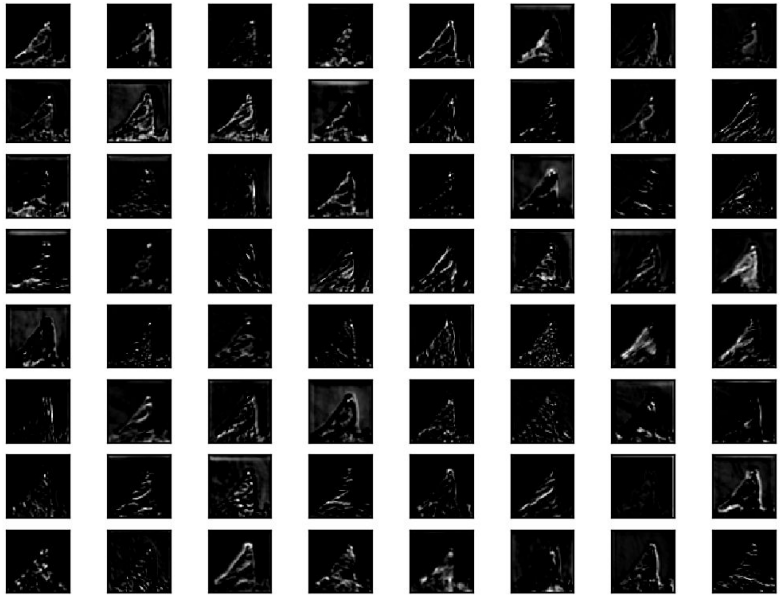
Visualizing Convolutional Layers of VGG16 trained on ImageNet[1]

Feature maps - block 1

[1] Browinee, Jason. "How to Visualize Filters and Feature Maps in Convolutional Neural Networks."
https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/

Visualizing Convolutional Layers of VGG16 trained on ImageNet[1]

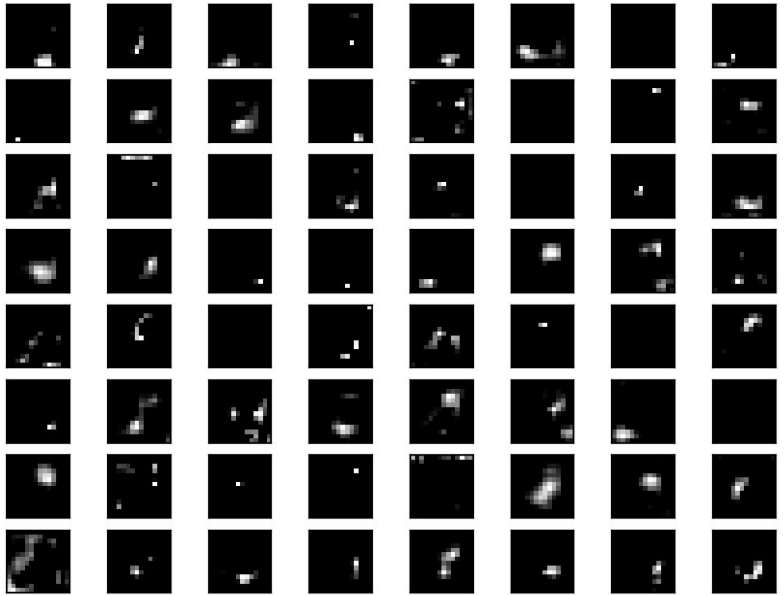
Feature maps - block 3



[1] Browinee, Jason. "How to Visualize Filters and Feature Maps in Convolutional Neural Networks."
https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/

Visualizing Convolutional Layers of VGG16 trained on ImageNet[1]

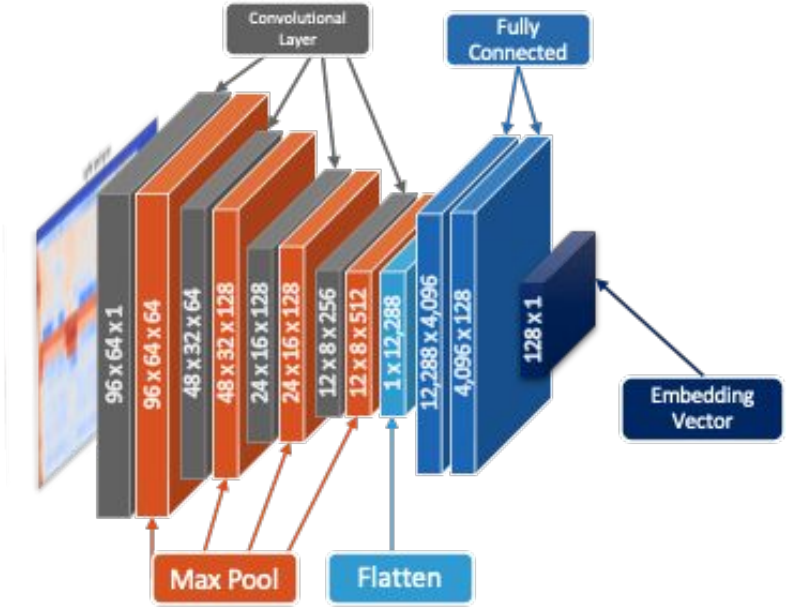
Feature maps - block 5



[1] Browinee, Jason. "How to Visualize Filters and Feature Maps in Convolutional Neural Networks."
https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/

VGGish[1]

A convolutional deep neural network for audio classification



Architecture

VGG adapted for audio processing

11 layers, adjustable weights

Operates on log-mel spectrograms

Advantages

Pre-trained model trained on Google's AudioSet[2] data

[1] Hershey, Shawn, et al. "CNN architectures for large-scale audio classification." 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2017.
[2] Gemmeke, Jort F., et al. "Audio set: An ontology and human-labeled dataset for audio events." 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2017.

04

Use case: Recognizing Commands

Overview, datasets, and tools

Use case: Recognizing Commands

Dataset - Speech Commands[1] v0.02

105,829 utterances of 35 different words

2,618 different speakers

1 second WAV files- 16-bit PCM at a 16 KHz sample rate

Word	Number of Utterances
Backward	1,664
Bed	2,014
Bird	2,064
Cat	2,031
Dog	2,128
Down	3,917
Eight	3,787
Five	4,052
Follow	1,579
Forward	1,557
Four	3,728
Go	3,880
Happy	2,054
House	2,113
Learn	1,575
Left	3,801
Marvin	2,100
Nine	3,934
No	3,941
Off	3,745
On	3,845
One	3,890
Right	3,778
Seven	3,998
Sheila	2,022
Six	3,860
Stop	3,872
Three	3,727
Tree	1,759
Two	3,880
Up	3,723
Visual	1,592
Wow	2,123
Yes	4,044
Zero	4,052

[1] Warden, Pete. "Speech commands: A dataset for limited-vocabulary speech recognition." arXiv preprint arXiv:1804.03209 (2018).

Workflow

01 **Preprocess**

Convert audio to input frames

02 **Load**

Define VGGish structure, load checkpoint

03 **Evaluate**

Feed spectrograms into VGGish

04a **Feature extraction**

Use output as vector embedding

04b **Use for transfer learning**

Add trainable layers onto frozen network

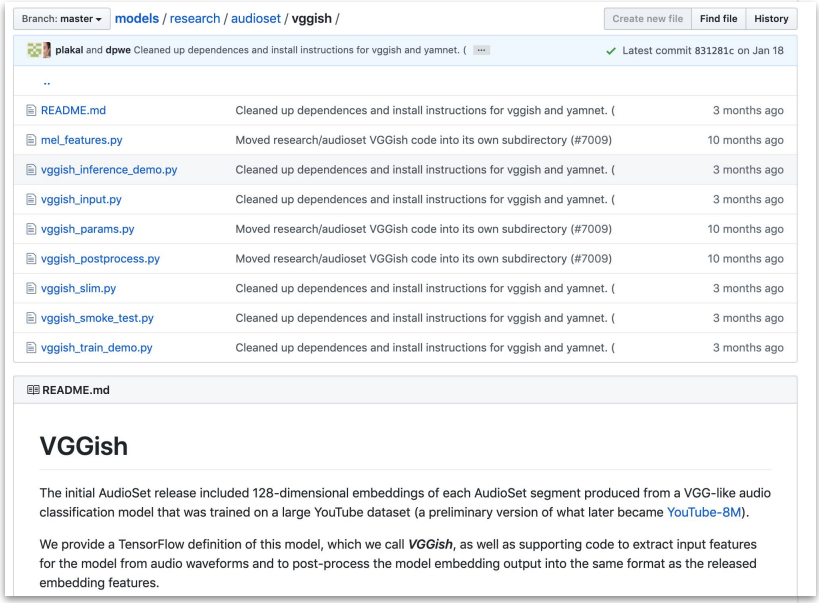
04c **Use as warm start**

Use pre-trained weights as starting point

VGGish library

Publicly available source code and utilities

https://github.com/tensorflow/models/tree/master/research/audioset/vggish



Getting set up

Install dependencies

```
pip install numpy resampy tensorflow==1.15 tf_slim six soundfile
```

Download VGGish library and checkpoint files

https://github.com/tensorflow/models/tree/master/research/audioset/vggish

Develop using preferred tools

Loading audio for use with VGGish

Processing performed by VGGish library

- Audio resampled to 16 kHz mono.
- Compute spectrogram.
- Map the spectrogram to 64 mel bins covering the range 125-7500 Hz and take the log of the mel spectrogram.
- Each audio sample mapped to non-overlapping 0.96 second segments, where segment is composed of 64 mel bands covering 96 10 ms frames

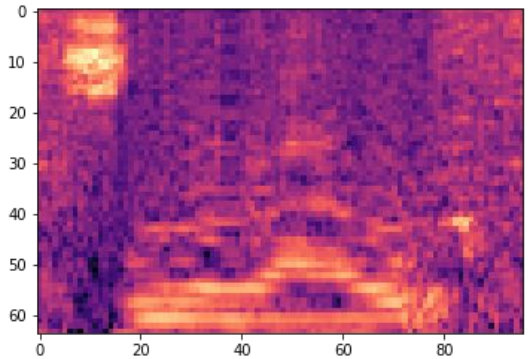
Loading audio and generating spectrograms

```
import sys
import matplotlib.pyplot as plt
import numpy as np
sys.path.append("/home/ubuntu/odsc/vggish/lib/models/research/audioset/vggish")

import vggish_input

fn = '/home/ubuntu/audio/speech_commands/zero/8a90cf67_nohash_0.wav'
spectrogram = vggish_input.wavfile_to_examples(fn)

plt.imshow(np.flip(spectrogram[0,:,:].T, axis=0), cmap='magma');
```



Feature extraction with VGGish

Using VGGish to generate vector embeddings of audio

Steps:

1. Define network
2. Load checkpoint (pre-trained weights)
3. Use network to process spectrogram

Defining network, loading weights, generating embeddings

```
import tensorflow as tf
tf.disable_v2_behavior()

import vggish_slim, vggish_params

ckpt = '/home/ubuntu/odsc/vggish/lib/vggish_model.ckpt'

with tf.Graph().as_default(), tf.Session() as sess:
    vggish_slim.define_vggish_slim(training=False)
    vggish_slim.load_vggish_slim_checkpoint(sess, ckpt)
    features_tensor = sess.graph.get_tensor_by_name(
        vggish_params.INPUT_TENSOR_NAME)
    embedding_tensor = sess.graph.get_tensor_by_name(
        vggish_params.OUTPUT_TENSOR_NAME)

    [embedding_batch] = sess.run([embedding_tensor],
                                 feed_dict={features_tensor: spectrogram})
```

VGGish post-processing

```
import vggish_postprocess

pca_params = '/home/ubuntu/odsc/vggish/lib/vggish_pca_params.npz'
pproc = vggish_postprocess.Postprocessor(pca_params)

postprocessed_batch = pproc.postprocess(embedding_batch)
```

VGGish output: raw vs post-processed

Raw network output

```
array([[0.        , 0.        , 0.5756371 , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.8052461 , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.09647734,
        0.        , 0.22157027, 0.36531267, 0.        , 0.        ,
        0.1050349 , 0.        , 0.4312849 , 0.        , 0.        ,
        0.        , 0.        , 0.09738243, 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.7165066 , 0.45736638,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.0754849 , 0.        , 0.04537962, 0.        , 0.        ,
        0.28914464, 0.        , 0.42669168, 0.        , 0.7256194 ,
        0.0045549 , 0.        , 0.        , 0.        , 0.        ,
        0.        , 1.1186049 , 0.        , 0.19068974, 0.01635714,
        0.        , 0.        , 0.        , 0.        , 0.07885516,
        0.        , 0.        , 0.15197024, 0.        , 0.02502064,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.61964595, 0.4623407 , 0.64971626, 0.        ,
        0.35118967, 0.6092059 , 0.        , 0.        , 0.02227053,
        0.        , 0.12507549, 0.        , 0.        , 0.5445715 ,
        0.        , 0.        , 0.16406971, 0.76613045, 0.        ,
        0.11071926, 0.        , 0.        , 0.        , 0.7489714 ,
        0.        , 0.        , 0.        , 0.34730738, 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.00565438, 0.11365021, 0.        ,
        0.        , 0.        , 0.02634984, 0.        , 0.        ,
        0.        , 0.09891586, 0.39740086, 0.        , 0.        ,
        0.        , 0.08000719, 0.        ]], dtype=float32)
```

VGGish output: raw vs post-processed

Post-processed output

```
array([[158,  14, 154, 100, 205,  72, 121,  65, 132, 249,  96,  86, 101,
        154,  70, 161, 100, 100, 163, 121,  16, 255, 134,  67,  66, 131,
        168, 210,  64, 186, 228, 102,  32,  75,   0, 219,  46,   0, 148,
        152,   0, 197,  96,  92, 187, 111, 255, 193,  93, 225, 160,  82,
         91,  76, 115, 106, 255,  42, 149, 137, 117,  93,  45, 220,  83,
         90, 144,   4, 129, 190, 136, 140, 172,  64, 108, 132,   0, 255,
         15,  48,  16,  92, 161, 101,  82, 158, 127, 145, 255,  32, 255,
        129,  52,   6, 149, 255, 218,  98, 253, 218,  47, 135, 255, 173,
          0,   0,  50,  45, 255,  78, 140,  85,  84,  41, 255,   0,  76,
        247,   0, 167, 123, 116,  13,   0, 168,   0, 178, 255]],
      dtype=uint8)
```

Why post-process?

Native output of VGGish: 128-dimensional encoding of each example frame evaluated by the network

VGGish also includes a post-processing capability in order to maintain compatibility with YouTube-8M (which has released visual/audio embeddings for millions of videos). Post-processor performs PCA, whitening, and quantizes output to 8 bits.

Why post-process?

- Only if you need to work with other embeddings
- Typically easier to work with raw version of embeddings

Next steps

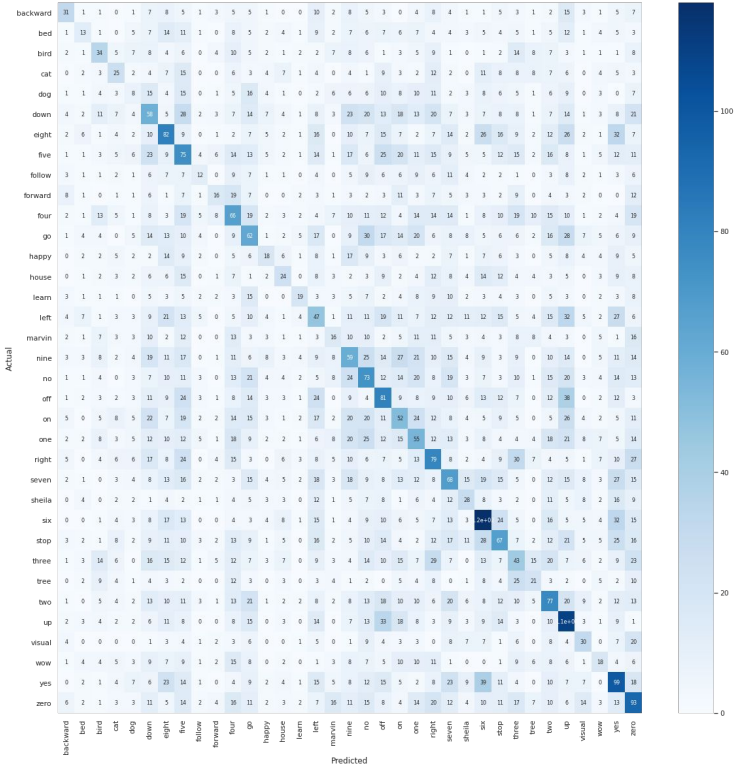
Using vector embeddings

Embeddings can be used in place of the raw audio in other algorithms (e.g. k-means) or models (e.g. xgboost)

Experimental results

XGBoost using embeddings

Overall accuracy - 18.4%



Transfer learning with VGGish

Using the VGGish network as a base for additional layers

Steps:

1. Define network - with frozen layers
2. Load checkpoint (pre-trained weights)
3. Add trainable layers onto network
4. Train network - only additional layers

Defining and extending the network

```
g = tf.Graph()
sess = tf.Session(graph=g)

with g.as_default():
    # Define VGGish.
    embeddings = vggish_slim.define_vggish_slim(training=False)

    # Grab last layer of network before compression to embeddings
    vggish_fc1 = sess.graph.get_tensor_by_name('vggish/fc1/fc1_2/Relu:0')

    # Define a shallow classification model and associated training ops on top
    # of VGGish.
    with tf.variable_scope('mymodel'):
      # Add fully connected layer with 500 units.
        num_units = 500
        fc1 = slim.fully_connected(vggish_fc1, num_units)

        # Add a classifier layer at the end, consisting of parallel logistic
        # classifiers, one per class. This allows for multi-class tasks.
        logits = slim.fully_connected(
          fc1, num_classes, activation_fn=None, scope='logits')

        # Add softmax output layer for classification
        tf.nn.softmax(logits, name='prediction')

        prediction_tensor = sess.graph.get_tensor_by_name('mymodel/prediction:0')

      # Add training ops.
        with tf.variable_scope('train'):
            global_step = tf.Variable(
                0, name='global_step', trainable=False,
                collections=[tf.GraphKeys.GLOBAL_VARIABLES,
                             tf.GraphKeys.GLOBAL_STEP])

            # Labels are assumed to be fed as a batch multi-hot vectors, with
            # a 1 in the position of each positive class label, and 0 elsewhere.
            labels = tf.placeholder(
                tf.float32, shape=(None, num_classes), name='labels')

            # Cross-entropy label loss.
            xent = tf.nn.softmax_cross_entropy_with_logits(
                logits=logits, labels=labels, name='xent')
            loss = tf.reduce_mean(xent, name='loss_op')
            tf.summary.scalar('loss', loss)

            # We use the same optimizer and hyperparameters as used to train VGGish.
            optimizer = tf.train.AdamOptimizer(
                learning_rate=vggish_params.LEARNING_RATE,
                epsilon=vggish_params.ADAM_EPSILON)
            optimizer.minimize(loss, global_step=global_step, name='train_op')
```

Initializing and training

```
with g.as_default():
    # Initialize all variables in the model, and then load the pre-trained
    # VGGish checkpoint.
    sess.run(tf.global_variables_initializer())
    vggish_slim.load_vggish_slim_checkpoint(sess, ckpt)

    # Locate all the tensors and ops we need for the training loop.
    features_tensor = sess.graph.get_tensor_by_name(
        vggish_params.INPUT_TENSOR_NAME)
    labels_tensor = sess.graph.get_tensor_by_name('mymodel/train/labels:0')
    global_step_tensor = sess.graph.get_tensor_by_name(
        'mymodel/train/global_step:0')
    loss_tensor = sess.graph.get_tensor_by_name('mymodel/train/loss_op:0')
    train_op = sess.graph.get_operation_by_name('mymodel/train/train_op')
```

```
# The training loop.
for _ in range(200):
    for i in range(num_batches):
        labels = labels_train_batches[i]
        features = audio_train_batches[i]

        [num_steps, loss, _] = sess.run(
          [global_step_tensor, loss_tensor, train_op],
          feed_dict={features_tensor: features, labels_tensor: labels})
        print('Step %d: loss %g' % (num_steps, loss))
```

Training:

- 100,000 batches of 500
- Run on EC2 (p3.2xlarge)
- One Tesla V100-SXM2-16GB GPU

Final loss

- **1.17431**

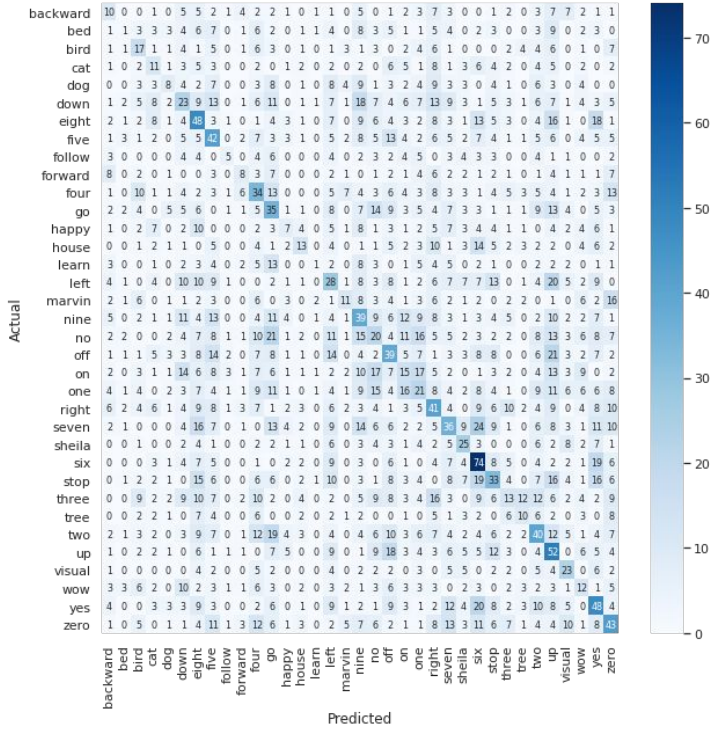
Run time

- **28 min**

Experimental results

Extending VGGish

Overall accuracy - 30.4%



VGGish as a warm start

Retraining and fine-tuning VGGish

Steps:

1. Define network - with trainable layers
2. Load checkpoint (pre-trained weights)
3. Add trainable layers onto network
4. Train network - including base layers

Defining and extending the network

```
g = tf.Graph()
sess = tf.Session(graph=g)

with g.as_default():
    # Define VGGish.
    embeddings = vggish_slim.define_vggish_slim(training=True)

    # Grab last layer of network before compression to embeddings
    vggish_fc1 = sess.graph.get_tensor_by_name('vggish/fc1/fc1_2/Relu:0')

    # Define a shallow classification model and associated training ops on top
    # of VGGish.
    with tf.variable_scope('mymodel'):
      # Add fully connected layer with 500 units.
        num_units = 500
        fc1 = slim.fully_connected(vggish_fc1, num_units)

        # Add a classifier layer at the end, consisting of parallel logistic
        # classifiers, one per class. This allows for multi-class tasks.
        logits = slim.fully_connected(
          fc1, num_classes, activation_fn=None, scope='logits')

        # Add softmax output layer for classification
        tf.nn.softmax(logits, name='prediction')

        prediction_tensor = sess.graph.get_tensor_by_name('mymodel/prediction:0')

      # Add training ops.
        with tf.variable_scope('train'):
            global_step = tf.Variable(
                0, name='global_step', trainable=False,
                collections=[tf.GraphKeys.GLOBAL_VARIABLES,
                             tf.GraphKeys.GLOBAL_STEP])

            # Labels are assumed to be fed as a batch multi-hot vectors, with
            # a 1 in the position of each positive class label, and 0 elsewhere.
            labels = tf.placeholder(
                tf.float32, shape=(None, num_classes), name='labels')

            # Cross-entropy label loss.
            xent = tf.nn.softmax_cross_entropy_with_logits(
                logits=logits, labels=labels, name='xent')
            loss = tf.reduce_mean(xent, name='loss_op')
            tf.summary.scalar('loss', loss)

            # We use the same optimizer and hyperparameters as used to train VGGish.
            optimizer = tf.train.AdamOptimizer(
                learning_rate=vggish_params.LEARNING_RATE,
                epsilon=vggish_params.ADAM_EPSILON)
            optimizer.minimize(loss, global_step=global_step, name='train_op')
```

Initializing and training

```
with g.as_default():
    # Initialize all variables in the model, and then load the pre-trained
    # VGGish checkpoint.
    sess.run(tf.global_variables_initializer())
    vggish_slim.load_vggish_slim_checkpoint(sess, ckpt)

    # Locate all the tensors and ops we need for the training loop.
    features_tensor = sess.graph.get_tensor_by_name(
        vggish_params.INPUT_TENSOR_NAME)
    labels_tensor = sess.graph.get_tensor_by_name('mymodel/train/labels:0')
    global_step_tensor = sess.graph.get_tensor_by_name(
        'mymodel/train/global_step:0')
    loss_tensor = sess.graph.get_tensor_by_name('mymodel/train/loss_op:0')
    train_op = sess.graph.get_operation_by_name('mymodel/train/train_op')
```

```
# The training loop.
for _ in range(200):
    for i in range(num_batches):
        labels = labels_train_batches[i]
        features = audio_train_batches[i]

        [num_steps, loss, _] = sess.run(
          [global_step_tensor, loss_tensor, train_op],
          feed_dict={features_tensor: features, labels_tensor: labels})
        print('Step %d: loss %g' % (num_steps, loss))
```

Training:

- 100,000 batches of 500
- Run on EC2 (p3.2xlarge)
- One Tesla V100-SXM2-16GB GPU

Final loss

- **1.45928e-07**

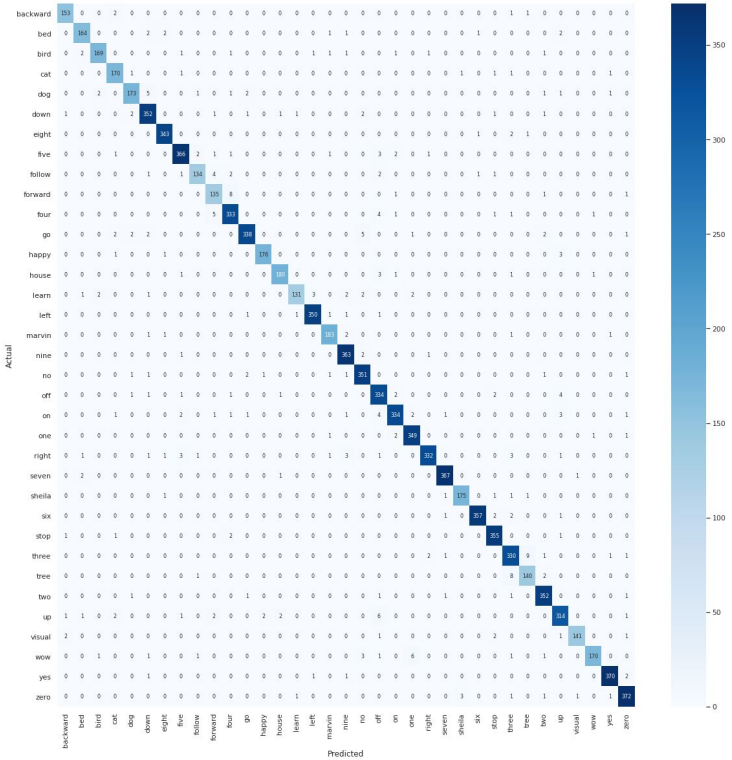
Run time

- **1 hr 20 min**

Experimental results

Retraining VGGish

Overall accuracy - 96.6%



In closing...

What have we learned?

phData

Code available on GitHub -

https://github.com/RobertCoop/ODSCEast2020