Instructor: Amy Orsborn | TA: Christopher Yin
aorsborn@uw.com | c8yin@uw.com

LAB 4 - INTRODUCTION TO CONVOLUTIONAL NEURAL NETWORKS WITH APPLICATION TO AUDIO SIGNAL PROCESSING

Due: Saturday 06/04/2022 at 11:59pm

The goal of this lab is to get a gentle introductions to Convolutional Neural Networks (CNNs) and how they can be used to perform a classification task on audio signals.

LAB INSTRUCTIONS

1. You might have to spend time outside the allocated lab hours to finish the lab. In doing so, you can approach any of the course instructors be it your lab TA, the other TA(s) or the main instructor.
2. You may work in teams or groups of 1-3 members. When it comes to collaborating with students in other groups, please do not share code but only discuss the relevant concepts and implementation methods. You may change group or team members for different labs but you cannot change group members for the given lab you are working on.
3. Please document your code well by using appropriate comments, variable names, spacing, indentation, etc.
4. The starter code is not binding on you. Feel free to modify it as you wish. Everything is fine so long as you're getting the right results.
5. Please upload the .ipynb file and the PDF version of the .ipynb file to canvas. One notebook per team is fine and any one team member can upload the file. The required file(s) must be uploaded by the deadline.

1 Goal of this Lab

The goals of this lab are to get familiar with some machine learning tools used for classifying audio data, and to highlight the importance of signal processing for such tasks. Specifically, we want to design a classifier that automatically detects whether a cough sound was produced by a female or male subject. Why are we using cough sounds? The current COVID-19 pandemic has demonstrated the importance of rapid and contact-free testing. One approach that has received increasing attention is to use a person's cough sound to determine whether or not they are infected. This has led to the emergence of a few publicly available cough sound datasets containing COVID and non-COVID tasks. So why are we not classifying COVID vs. non-COVID? Detecting COVID from cough sounds is a pretty difficult problem that requires good preprocessing of the original dataset and many examples of COVID positive and COVID negative coughs; having a large number of training samples is especially important if deep learning architectures such as CNNs are used. Unfortunately, we don't have a very large dataset so that a COVID cough detection is difficult to realize.

The starter code for this lab contains code for loading the dataset and demonstrating how

neural nets are implemented in Python using Tensorflow. Furthermore, we provide target accuracy scores that you should be able to meet when completing the starter code. However, feel free to deviate from the starter code and implement different neural network architectures or different signal preprocessing pipelines to improve your classification accuracy as much as possible.

2 The Dataset

For this lab, we will use the *Coughvid* dataset[1] that contains cough sounds from over 25 000 participants across the world. A preprocessed version of this dataset with approximately 7000 cough sounds is provided to you. You can download the dataset using this link[2] (note that you have to use your UW email to download the dataset). Unzip the downloaded file. Your starter code should be in the same directory as the `coughvid/` folder. The unzipped file size of the dataset is approximately 5 GB. Talk to your TA if you run into any issues with accessing the dataset or if you don't have sufficient space on your computer.

The dataset has already been split into a training and a testing set, which you can find in the `train` and `test` folder, respectively. You will train your classifier on the training set and evaluate its performance on the test set. Note that it is an absolute no-go to use samples from the test set during the training stage. Think about why and ask your TA if you cannot find the answer.

The train set contains 80 % of all cough sounds. The test set contains the remaining 20 %. Each cough sound is exactly 8 s long, was sampled using a sampling frequency of 22 050 Hz, and has been normalized to a maximum amplitude of 1. Besides the audio recordings in the `train` and `test` folder, tables with meta data are provided for the train and test set (`train.csv` and `test.csv`, respectively). These metadata tables can be loaded in Python as `pandas.DataFrames`. A `DataFrame` is essentially a table, where each row corresponds to one cough recording and the columns are the different attributes or *features* associated with the recording. The feature that is the most important for our purpose is *gender* and has the two possible values *male* and *female*. (Unfortunately, the dataset does not have enough samples to include non-binary genders).

2.1 Assignment 1: Load the Cough Sound Dataset and Display and Listen to a Data Sample (20 %)

1. Use the function `load_dataset()` (with parameters `transform = None` and `downsample = True`) to load the train and test dataset. The code for loading the train dataset is

[1] Orlandic, L., Teijeiro, T. & Atienza, D. The COUGHVID crowdsourcing dataset, a corpus for the study of large-scale cough analysis algorithms. Sci Data 8, 156 (2021). `https://doi.org/10.1038/s41597-021-00937-4`

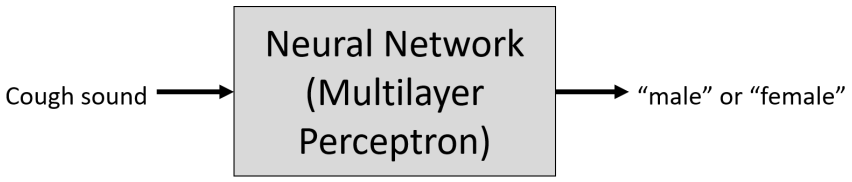
[2] This is the full URl: `https://drive.google.com/file/d/1sQEXXm-MMCjFNVcHvW2kN5xECVpFLPww/view?usp=sharing`

already provided in the starter code. You still have to write the code line for loading the test dataset (this is an easy task). `load_dataset()` returns a tuple `(data, labels)`, where `data` contains the audio time series organized in a 2D numpy array and `labels` contains the labels (*male* or *female*) for each audio sample. *male* and *female* are encoded as 0 and 1, respectively. Each row of `data` is a single cough recording and the corresponding column in `labels` is its corresponding label. That is, the audio time series and label of the i^{th} recording can be accessed by `audio = data[i]` and `label = labels[i]`. If `downsample = True`, the cough recordings are downsampled to a sampling frequency of 11 025 Hz.

2. Plot the time series of a single recording using `plt.plot()` and play the cough using the `sounddevice.play()` function from Lab 2. You can display any recording you like. The x-axis of your plot should be in seconds. Make sure to label both x- and y-axis properly. Print the label of the corresponding recording.

3 A Naive Deep Learning Implementation

Our first approach for building a classifier is to disregard any signal processing and let deep learning do all the work. We will later see that this does not perform well and add a signal processing stage to our pipeline that will significantly improve the performance. For now, our classification pipeline looks as follows:



The core of this classifier is a fully connected Multilayer Perceptron (MLP). This neural network architecture is composed of a collection of neurons that are organized in layers, whereby the neurons of one layer are connected to all neurons of the next layer (hence ”fully connected”, though ”dense” is also used). The general architecture of a neural network is shown in Fig. 1.

To train a neural network, a loss function (also called ”cost function”) is computed based on the model output. For regression tasks (i.e. predicting a numerical value such as exam score or petal width), the mean squared error between the prediction and true value is commonly used. In our case, for a binary classification task we will use the binary cross entropy function.

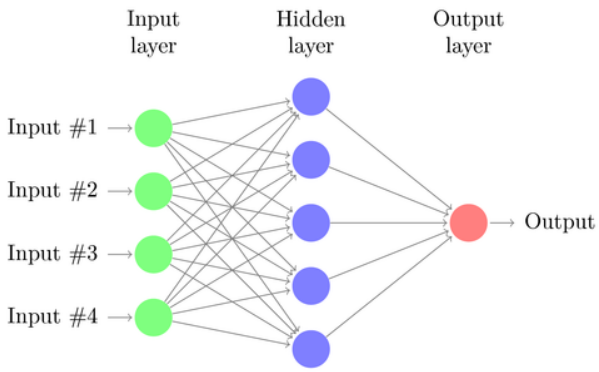


Figure 1: Illustration of neural network architecture. Image from `https://texample.net`

Once you have decided on a loss function, you iteratively compute the output of your neural network on all of your training samples, calculating the loss each time–a higher value of loss indicates worse performance. At first your model will perform very poorly, because all the neuron weights will have been randomly initialized. However, as training proceeds, the neural network will compute the gradient of the loss function (recall your calculus!), which will indicate how to adjust the weights so as to move in a direction that will reduce the value of the loss function. This approach is known as gradient descent, and the process by which all the weights of a model are updated to follow the gradient is called backpropagation. This is the key aspect of neural networks, and what enables them to "learn" how to perform a wide variety of tasks without any specialized knowledge or mathematical modeling.

As Fig. 1 shows, a neural network consists of an input, an output, and one or multiple hidden layers. Each layer contain a number of neurons that are connected to all neurons of the next layer. Each neuron performs the following series of computations that are also illustrated in Fig.2:

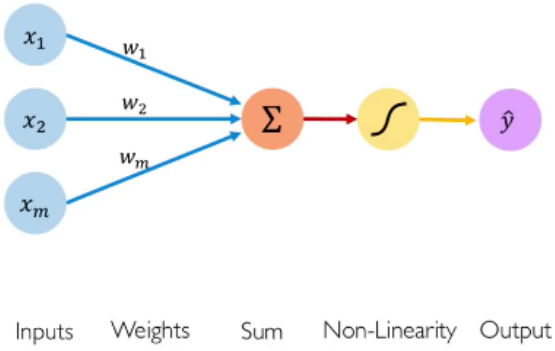


Figure 2: A single neuron inside a neural network. Image from `https://medium.com/analytics-vidhya/neural-network-part1-inside-a-single-neuron-fee5e44f1e`

1. Each input of the neuron is weighted by w_i.
2. All weighted inputs are summed together.
3. The sum of all weighted inputs (plus a bias term) is passed through a non-linear function to give the neuron's output. This non-linear function is also called the activation or squashing function; without this nonlinearity, the model would simply perform matrix multiplication, which would limit its ability to learn complex patterns.

The key for the success of neural networks is that the weights in each layer of the network are learned from training data to perform a specific task. This characteristic is what we want to leverage to design our classifier for detecting the gender associated to a cough sound.

3.1 Introduction to Tensorflow and Simple Neural Network Implementation

At first glance, implementing a neural network architecture like the one in Fig. 1 seems very daunting. Luckily, Python provides many libraries that make implementing neural networks easy. One such library is *tensorflow*, which has been developed by Google Brain and was initially released in 2015. Specifically, we will use the `tensorflow.keras.models` module that can be used to build and customize even complex neural network architectures with just a few lines of code. The input to our neural network is the 8 s long cough sound time series sampled with a frequency of 11 025 Hz. As the input layer has one unit for each input feature, the input layer of our architecture consists of $8 \cdot 11025 = 88200$ units.

To design a neural network, we first have to initialize the model by calling

```
model = model.Sequential().
```

Next we add an Input layer, which will tell the model what shape the input data will be.

```
model.add(layers.Input(data_train.shape[1]))
```

Recall that the first (axis=0) dimension of `data_train` is the sample axis, and the second (axis=1) dimension of `data_train` is the feature axis. The first dimension will therefore have length equal to the number of samples in the dataset, and the second dimension will have length equal to the number of features of a single sample: in this case, each timepoint of an audio recording counts as a single feature.

Now we can add fully connected layers to the model by calling

```
model.add(layers.Dense(100, activation='relu')).
```

The first parameter in `Dense` specifies the number of neurons for this layer (i.e., 100 in this case). The second parameter specifies the non-linearity used by each neuron. A common non-linearity is the *ReLU* function that is given by

$$f(x) = \max(0, x). \tag{1}$$

Other common non-linearities are the sigmoid function and the hyperbolic tangent (tanh) function.

Because our classification problem is binary, our output layer should only include a single neuron that returns 0 (or a number close to zero) if it detects *male* and 1 (or a number close to 1) if it detects *female*. We can implement this layer analogously to the previous layer:

`model.add(layers.Dense(1, activation='sigmoid'))`.

The sigmoid function returns an output from 0 to 1, which makes it ideal for this output layer. (An alternative to this approach is to have a 2-node output layer with softmax activation, where each node will output a probability of a different class; feel free to experiment if you wish.) Finally, to plot a summary of the model, you can call `model.summary()`. All the above steps are provided in the starter code.

Assignment 2a (10 %):

1. Add at least two additional `Dense` layers to the network architecture in the starter code. You are free to chose the number of neurons and activation function in each layer.

To train the model, we need two more lines of code. First, we have to compile the model by calling

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
loss='binary_crossentropy', metrics=['accuracy']).
```

You do not have to worry about the `optimizer` at this point; it indicates the algorithm by which backpropagation is implemented, and the ADAM optimizer is fairly standard. Recall from calculus that the gradient of a function indicates the direction of greatest change; the `learning_rate` parameter indicates how far we step in the direction specified by the gradient for each round of the optimization. Too small of a learning rate will cause training to occur very slowly; too large will cause your model performance to oscillate and possibly never converge on an optima. The provided default learning rate of `1e-3` will likely work well for your models, as the ADAM optimizer also dynamically adapts the learning rate as training progresses. The parameter `metric=['accuracy']` indicates that we will track classification accuracy throughout the training process, i.e., the percentage of correctly classified samples, as high as possible. Note that the loss function, and not accuracy, is used to update the model weights.

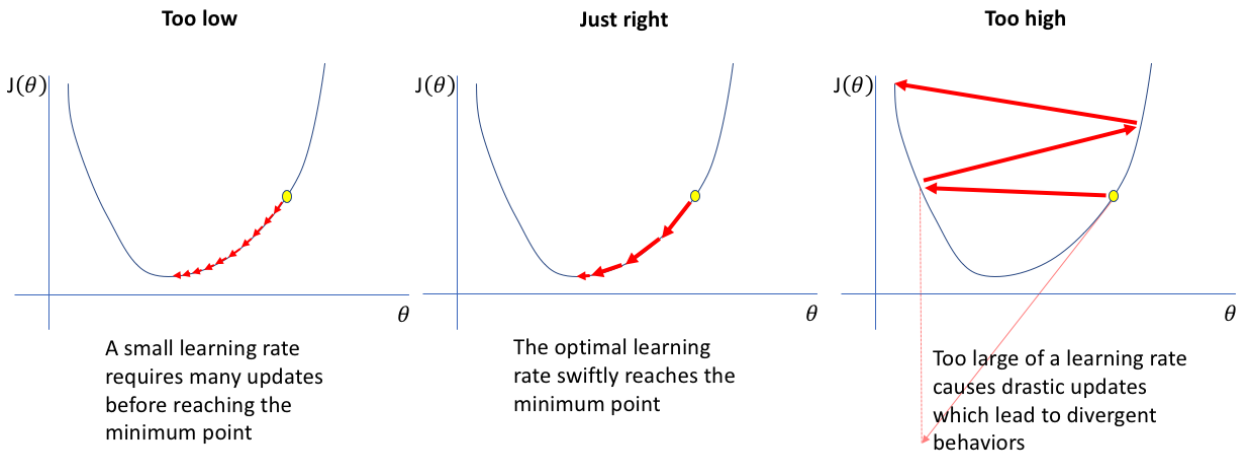


Figure 3: Effects of learning rate on model training. Image from: https://medium.com/inveterate-learner/deep-learning-book-chapter-8-optimization-for-training-deep-models-part-i-20ae75984cb2

Finally, the actual training is performed by calling

`history = model.fit(data_train, labels_train, epochs=10, validation_data=(data_test, labels_test), batch_size=128,shuffle=True)`,

where `data_train` and `labels_train` are the train dataset and labels, and `data_test` and `labels_test` are test dataset and labels, respectively. The `epoch` parameter indicates how many times the entire train dataset is used for training the model, and the `batch_size` parameter indicates how many samples are fed through the model in each forward pass of the training. Gradients are computed on batches (technically, "minibatches") rather than individual samples to stabilize the training process; ask the TA for more details.

After each epoch, we compute the accuracy on the test dataset. When you run those two lines of code, you will see a printout that contains, among other things, the classification accuracy on the train and test set after each epoch. Note that the classification accuracy on the test set is called *val_accuracy*.

In this section we have discussed many aspects of the model that can be tuned to improve performance. These are known as hyperparameters, and the process of training a good model involves finding optimal hyperparameters, as these can all affect what the model learns. There are a variety of heuristics for tuning hyperparameters (e.g. Bayesian optimization, grid search, hyperband) which are out of the scope of this assignment, but may be of interest to you if you encounter deep learning in a subsequent course/job. A summary of the hyperparameters discussed in this section is reported below:

1. Number of layers

2. Layer size
3. Activation function
4. Batch size
5. Learning rate

3.2 Assignment 2b (10 %):

Run the model fitting and describe (in words in a separate markdown cell), how the loss on the train and test set changes throughout the model training; specifically, how and when do the training and test loss curves diverge? (Hint: Don't overthink this, your model will not perform well). Plot train and test set loss as a function of epoch number, and also report the final accuracy on the test set; also plot the train and test set accuracy as a function of epoch number. (The code for that is already provided and all you have to do is execute it.) Note that if training loss decreases while test loss increases, this is an indicator of a model pathology known as overfitting, wherein your model learns only to predict specifically on the training set and can't be generalized to external data. Assume you have a new cough sound sample that the model has not seen before. How likely do you think is it that the model detects the right gender? Comment on that in the notebook.

4 Changing the Signal Representation

You should have noticed that our simple neural network model is really bad at detecting the correct gender on the test set. One reason is that we have not used a good signal representation, which we will change now. (Another reason is that a fully connected network is not built to capture spatial relationships, as opposed to the convolutional model which will be discussed in Assignment 4.) A common representation of audio signals in machine learning is the spectrogram. The spectrogram is one form of time-frequency representation and shows how the spectrum of an audio signal changes over time. It can be computed via the Short-Time Fourier Transform (STFT) given by

$$X[m,k] = \sum_{n=0}^{N-1} x[n]w[n-mT]\mathrm{e}^{-j\frac{2\pi}{N}nk}, \tag{2}$$

where $w[n]$ is a window of length N and T is the window hop size. The indices m and k in Eq. (2) represent time and frequency, respectively. Intuitively, Eq. (2) states that we break our discrete time signal $x[n]$ into a series of segments, multiply each segment by a window function $w[n]$, and finally compute the DFT of each windowed segment. This process is also illustrated in Fig. 4.

To finally obtain the spectrogram, we take the squared magnitude of the STFT:

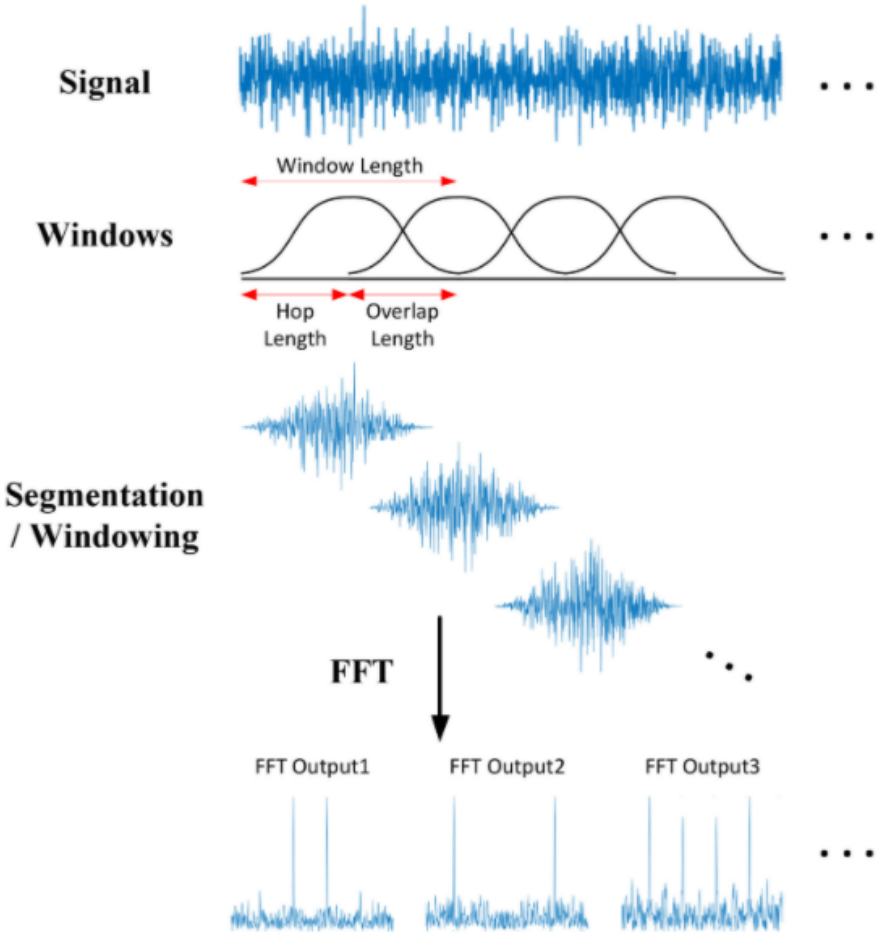


Figure 4: Illustration of short-time Fourier transform (STFT). Image from Jeon, H.; Jung, Y.; Lee, S.; Jung, Y. Area-Efficient Short-Time Fourier Transform Processor for Time–Frequency Analysis of Non-Stationary Signals. Appl. Sci. 2020, 10, 7208. `https://doi.org/10.3390/app10207208`

$$S[m,k] = |X[m,k]|^2 . \tag{3}$$

Since the spectrogram is a 2-dimensional function of time and frequency, we can visualize it like an image, where the horizontal axis indicates time and the vertical axis frequency[3].

[3]The common spectrogram representation includes some additional normalization constants, which we have ignored here for simplicity. Furthermore, following a common practice in machine learning, we will normalize $S[m,k]$ to have a maximum magnitude of 1 rendering the need for normalization constants unnecessary.

Assignment 3: Spectrogram of Cough Sounds (30 %)

1. Complete the function `spec()` that computes the spectrogram of a single audio signal. The function takes as inputs the audio signal `x` and the window length of the STFT `N`. Assume a window hop size of $T = N$. That is, adjacent segments do not overlap. For the window function we use a simple rectangular window. Your implementation should include the following steps:
   (a) split the signal x into K segments of length N.
   (b) compute the FFT of each segment and only store the frequency points from $k = 0$ to $N/2$ (corresponding to the DTFT frequency from 0 to π.).
   (c) Compute the squared magnitude according to Eq. (3).
   (d) Compute the normalized spectrogram

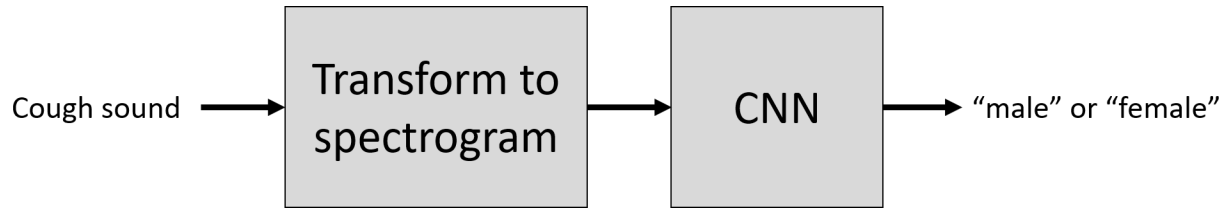
$$\tilde{S}[m,k] = \frac{S[m,k]}{\max\left(S[m,k]\right)}. \tag{4}$$

2. Once you have completed your implementation of `spec()`, run the code cell that computes the spectrogram of a single cough sound and compare it with numpy's spectrogram function used in `spec2()`. The two spectrograms you get should look identical.

Hint: You can use the function `np.fft.fft()` and might find the function `np.split()` helpful. With those functions it is possible to write an implementation without any for loops.

5 Using Spectrograms and Convolutional Neural Networks to Classify Cough Sounds

In this final section, we use the spectrogram representation to design a better classifier to detect the gender of cough sounds. Since the spectrogram is a 2-dimensional array (like an image) we replace the architecture in Section 3 by a convolutional neural network (CNN). That is, out classification pipeline will look as follows:



In a nutshell, a CNN is a neural network architecture that performs many 2-dimensional convolutions of the original input image and learned feature images with convolutional kernels. The 2-D convolution is illustrated in Fig 5. During the training procedure, the weights of the kernels are learned to perform a certain task such as classifying cough sounds.

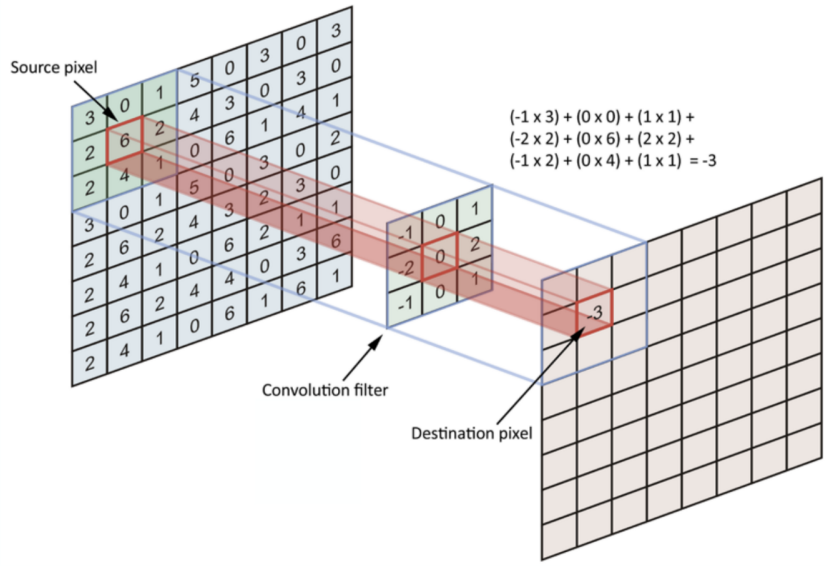


Figure 5: 2-dimensional convolution of image with kernel. Image from `https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac`

A full CNN architecture is illustrated in Fig. 6. A typical CNN architecture consist of multiple convolutional layers with varying number and size of kernels in each layer. Between convolutional layers, we typically find MaxPooling layers that aggregate samples in the images; this is considered a form of regularization that helps prevent your model from overfitting on the training set. For more details on CNNs, feel free to read this article or ask your TA. Note that you do not need a thorough understanding of CNNs to complete the next assignment.

Assignment 4: CNN Implementation (30 %)

1. Load the train and test set using the `load_dataset()` function with the parameter `transform='spec'` (you only need to run the corresponding code cell in the starter code).
2. Build the CNN architecture. An initial architecture with one convolutional layer has already been provided in the starter code. Add at least one more convolutional layer followed by a MaxPooling layer to the architecture. The important input parameters of the `Conv2D` layer are
   - `filters`: number of convolutional kernels
   - `kernel_size`: size of each kernel
   - `activation`: The activation function that is used for the output of the convolutional layer. You should use the `'relu'` function.

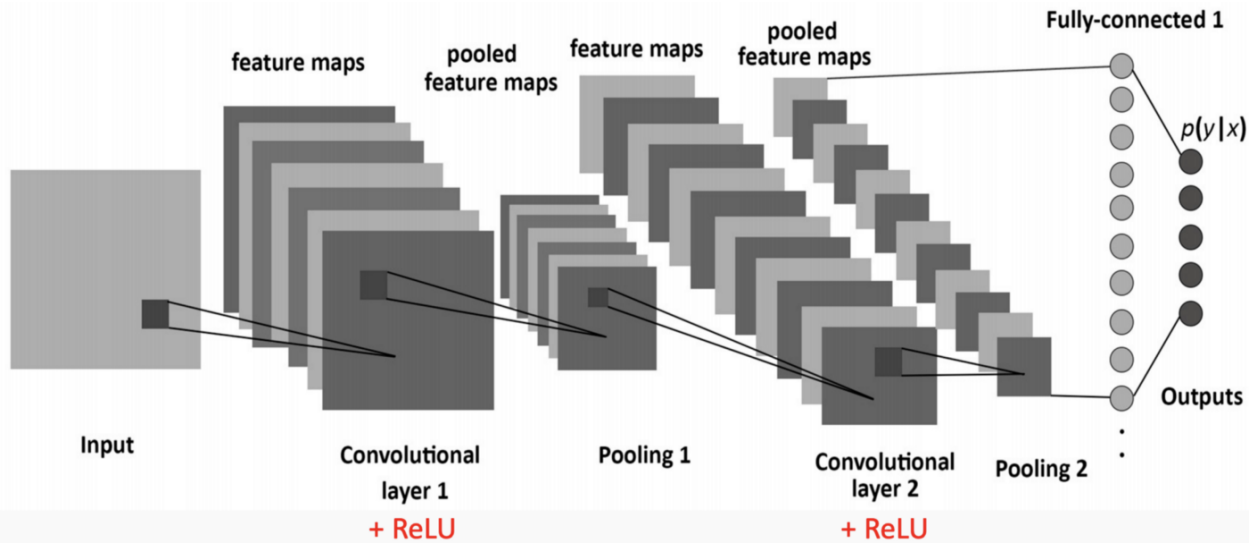


Figure 6: Illustration of CNN architecture. Image from `https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac`

The `MaxPooling` layer only takes a `pool_size` that is equivalent to the `kernel_size` of the convolutional layer.

3. Run the next code cell of the starter notebook to train your architecture. As in Assignment 2, the printout will tell you the accuracy of the classifier on the train and test set after each epoch.
4. As in Assignment 2, plot the accuracy on the train and test set as a function of epoch.
5. Change the parameters of your CNN architecture (number of layers, kernel size, etc.) to get as high of an accuracy on the test set as possible. Describe the observations you make (What changes seem to improve/decrease the test set accuracy? How does the accuracy on train and test set change throughout the training process (i.e., with increasing number of epochs)?). You should be able to get an accuracy of at least 71 % to 72 % on the test set.