DEEPFAKE SPECCH RECOGNITION

Importing Libraries :-

- torch / torch.nn → Model construction & Training
  - → Building & Training deeplearning models
  - → defining layers & operations
- tqdm.auto.tqdm → adds a progress bar for long-running loops
  - → Progress monitoring

```
def readtxtfile(path):
    with open(path, 'r') as file:
        text = file.read().splitlines()
        return text
```

function reads a protocol file (→ ASVspoof2019.LA.cm.train.trn.txt) a text file, splitting it into individual lines → stored in a list

ASVspoof Protocol file format

speaker_id	file_id	attack	label

e.g.: LA_0079 LA_T_100137 -- bonafide

→ extracts file_name (index 1) & label (index -1)

```
def getlabels(path):
    text = readtxtfile(path)
    filename2label = {}
    for item in tqdm(text):
        key = item.split(' ')[1]
        value = item.split(' ')[-1]
        filename2label[key] = value

    return filename2label
```

→ empty dictonary created

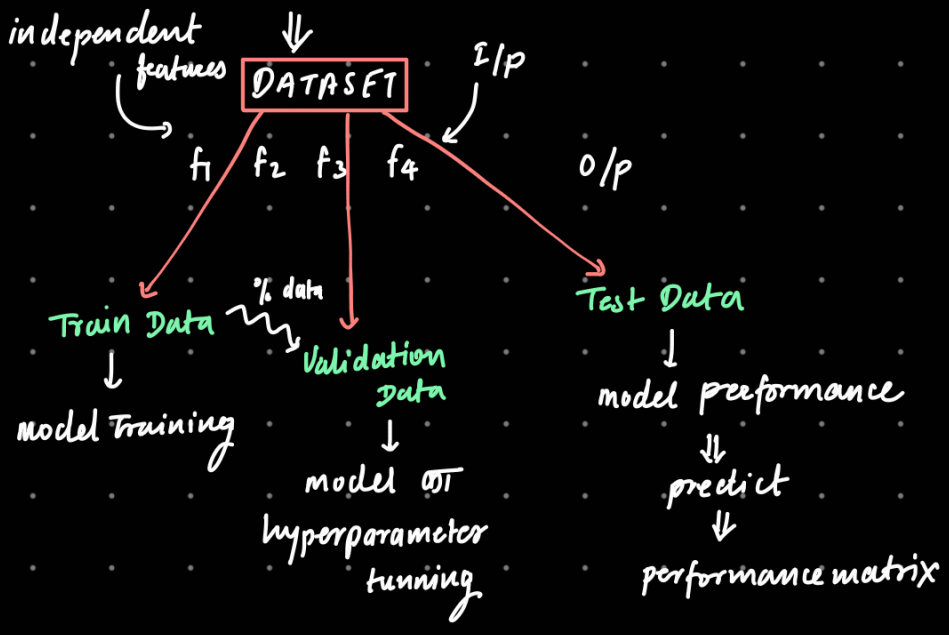
tqdm progress bar wraps around for-loop, showing progress as protocol file is parsed.

index = -1 → denotes last element

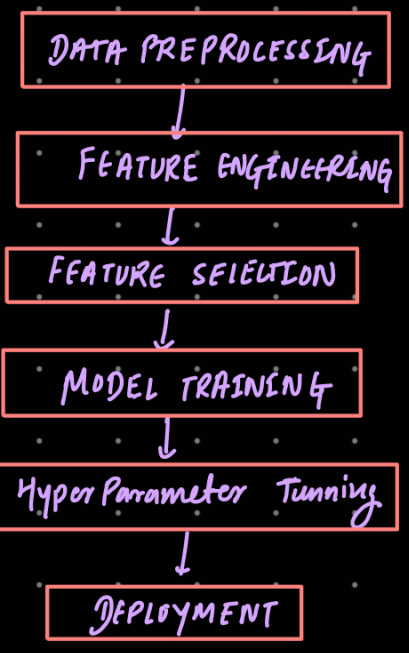
each file_name is mapped to its respective label (key) (value).

Training data vs Validation data vs Testing data

Regression
Classification
} supervised learning



ML - lifecycle



CROSS VALIDATION → C.V.

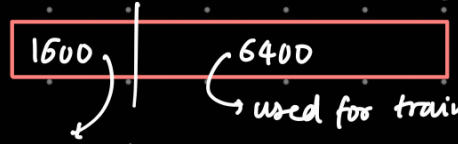
e.g.: Total data → 10k
Training data → 8000
Testing data → 2000

Now let, C.V. = 5

Hence, $\frac{8000}{5} = 1600$

Thus, for every cross validation,
training data = 8000 − 1600 = 6400
Testing data = 2000
Validation data = 1600

Therefore, C.V. = 1



⇒ Accuracy = A_1

C.V. = 2

⇒ Accuracy = A_2

.
.
.

⇒ Accuracy = A_5 at C.V. = 5

find mean of $A_1, A_2 \dots A_5$

Accuracy or any performance meterices

→ handling audio file, loading & Basic preprocessing

```python
class ASVSpoof(torch.utils.data.Dataset):
    def __init__(self, audio_dir_path, num_samples, filename2label, transforms):
        super().__init__()
        self.audio_dir_path = audio_dir_path
        self.num_samples = num_samples
        self.audio_file_names = self.get_audio_file_names(filename2label)
        self.labels, self.label2id, self.id2label = self.get_labels(filename2label)
        self.transforms = transforms
```

Constructor

→ fixed number of audio samples (frames) to be used for each file

→ any additional audio transformation, such as mel spectrogram

→ loads audio file & returns signal & sample rate

```python
def __getitem__(self, index):
    signal, sr = torchaudio.load(os.path.join(self.audio_dir_path, self.audio_file_names[index]))
    signal = self.mix_down_if_necessary(signal)
    signal = self.cut_if_necessary(signal)
    signal = self.right_pad_if_necessary(signal)
    label = (self.labels[index])
    return signal, label
```

→ converts stereo audio to mono by averaging the channels

→ cuts the audio if it exceeds num_samples

→ pads the audio signal to ensure it has exactly num_samples frames

Padding & Cutting → especially important in Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs)

because, they expect consistent input shape

```python
def get_audio_file_names(self, filename2label):
    audio_file_names = list(filename2label.keys())
    audio_file_names = [name + '.flac' for name in audio_file_names]
    return audio_file_names

def get_labels(self, filename2label):
    labels = list(filename2label.values())
    id2label = {idx: label for idx, label in enumerate(set(labels))}
    label2id = {label: idx for idx, label in enumerate(set(labels))}
    labels = [label2id[label] for label in labels]
    return labels, label2id, id2label
```

→ append .flac extension to each file name

→ converts the string labels (bonafide, spoof) into Integer IDs.

```python
num_samples = 4 * 16000  # IMPORTANT!!
```

→ defines audio sample (frames) that will be used for each audio file

4 * 16000 → represents the no. of samples for 4 seconds of audio at sampling rate = 16000 Hz

→ each audio file will be processed to include 4 seconds worth of audio data

Sample Rate :- Audio files typically have sample rate, which refers to the number of samples captured per second.

By defining, `num_samples`, the audio will be truncated or padded to ensure every input has exactly 64,000 samples (which represent 4 seconds of audio)

Data Augmentation :

```
ain_dataset = ASVSpoof(train_audio_files_path, num_samples, filename2label, None)
l_dataset = ASVSpoof(val_audio_files_path, num_samples, val_filename2label, None)
st_dataset = ASVSpoof(test_audio_files_path, num_samples, test_filename2label, None)
```

→ No data augumentation : No transforms are applied, the dataset loads as-it is.

In many audio processing tasks, you might apply additional transformations [e.g., generating spectrograms, performing data augmentation] to enrich the dataset or convert the raw audio into a more suitable format for the model. [e.g., Mel. spectrogram]

`torchaudio` → audio processing in PyTorch

`timm` → (PyTorch image models) for access to pre-trained models & utilities

```
train_loader = torch.utils.data.DataLoader(train_dataset, shuffle=True, batch_size=8
val_loader = torch.utils.data.DataLoader(val_dataset, shuffle=True, batch_size=8)
test_loader = torch.utils.data.DataLoader(test_dataset, shuffle=True, batch_size=8)
```

→ data is shuffled at the begining of epoch to prevent the model from memorizing the data order

→ the loader provides batches of 8 samples at a time

Batch Size: The batch size controls how many samples the model processes before updating the weights.

† Large Batch sizes can speed up training but may require more memory.

```
t_steps = len(train_loader)
v_steps = len(val_loader)
ts_steps = len(test_loader)
```

→ number of batches (or steps) required to complete one pass through the entire dataset for training, validation & testing

→ steps per epoch = $\frac{\text{num\_samples}}{\text{batch\_size}}$

```
torch.autograd.set_detect_anomaly(True)
```

- **Gradient Issues**: During training, if any invalid gradient operations occur (e.g., division by zero, NaN values), it can be hard to trace the source. `torch.autograd.set_detect_anomaly(True)` helps by raising an error the moment a problematic operation is detected, allowing you to debug issues more easily.
- **Vanishing/Exploding Gradients**: These issues often arise during deep learning training. Vanishing gradients occur when gradients are too small to update weights meaningfully. Exploding gradients happen when the gradients grow too large, causing instability.

CustomWav2VecForClassification → class defines a custom neural network based on wav2vec2 architecture from Hugging Face's transformers library. This model is designed for binary classification tasks, like deepfake detection (bonafide vs spoofed).

wav2vec2: A pre-trained model designed for Automatic Speech Recognition (ASR) but is often repurposed for other tasks like audio classification by leveraging its ability to extract meaningful audio features.

LSTM: (Long Short-Term Memory) a type of Recurrent Neural Network (RNN) that helps capture temporal dependencies in sequence, such as audio

→ designed to learn from sequence of data, making it particularly effective for tasks involving time series or sequential inputs, like audio signals

→ Role in this project

→ audio signals are sequential
→ consists of series of time-dependent samples

* Temporal Dependencies → LSTMs remember past information while considering current input

→ LSTM layer in your model helps capture the temporal features of the audio sequence after wav2vec2 model has extracted high-level representations.

* Bidirectional Dependencies:

LSTM → bidirectional i.e., it processes the input data in both forward & backward directions. This allows the model to capture context from both past & future audio samples providing a more comprehensive understanding of audio sequence

* Combining features: LSTM layer takes the feature vectors output by wav2vec2 model & models the relationships between these features over time. By using LSTMs after wav2vec2, the model can combine powerful feature extraction with temporal learning.

Sigmoid Activation Function → maps any real-valued number into a value between 0 & 1.

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

→ Binary classification tasks

→ Sigmoid for the last layer allows your model to output probabilistic predictions

Alternatives to LSTM

1. **Gated Recurrent Unit (GRU)**
   - **Overview:** GRU is a simpler and computationally efficient variant of LSTM. It combines the forget and input gates into a single update gate and uses fewer parameters than LSTM.
   - **Benefit:** Often performs comparably to LSTM while being faster to train due to its reduced complexity. It can effectively capture temporal dependencies in sequences like audio.
2. **Convolutional Neural Networks (CNNs)**
   - **Overview:** CNNs are designed for processing grid-like data and are commonly used in image and audio processing. For audio, they can operate on spectrograms or other time-frequency representations.
   - **Benefit:** CNNs are highly efficient at capturing local patterns in data, and they can be used for feature extraction, potentially providing better accuracy in audio classification tasks.
3. **Temporal Convolutional Networks (TCNs)**
   - **Overview:** TCNs use 1D convolutions to process sequential data, similar to LSTMs but with causal convolutions. They can capture long-range dependencies through dilated convolutions.
   - **Benefit:** They have been shown to perform well on sequential tasks, and they offer advantages in terms of parallelization and training speed compared to RNNs.
4. **Transformers**
   - **Overview:** Transformers have gained popularity for their ability to model long-range dependencies without the sequential limitations of RNNs. They use self-attention mechanisms to weigh the importance of different input elements.
   - **Benefit:** Transformers can handle longer sequences more efficiently and have shown state-of-the-art performance in various tasks, including speech recognition and classification.

Alternatives to Sigmoid Activation

1. **Softmax Activation**
   - **Overview:** The Softmax function is used in multi-class classification tasks. It converts the raw logits into probabilities for each class.
   - **Benefit:** If your task can be treated as multi-class (e.g., different types of spoofing), using Softmax can provide a more robust probability distribution across multiple classes.
2. **Binary Cross-Entropy Loss with Sigmoid Output**
   - **Overview:** While this uses Sigmoid for the output layer, you might consider optimizing the loss function. Combining binary cross-entropy with a more sophisticated weighting scheme can help if class imbalances exist.
   - **Benefit:** Adjusting class weights can improve the model's sensitivity to the minority class (e.g., spoofed audio) and lead to better overall performance.
3. **Swish or Mish Activation**
   - **Overview:** Swish and Mish are newer activation functions that have been shown to improve performance in some deep learning models. They are smooth and non-monotonic.
   - **Benefit:** These activations can potentially lead to better gradient flow and higher model accuracy compared to traditional functions like ReLU and Sigmoid.

```python
class CustomWav2Vec2ForClassification(nn.Module):
    def __init__(self, checkpoint):
        super(CustomWav2Vec2ForClassification, self).__init__()
        config = Wav2Vec2Config.from_pretrained(checkpoint)
        self.feature_extractor = Wav2Vec2FeatureExtractor.from_pretrained(checkpoint)
        self.wav2vec2 = Wav2Vec2Model.from_pretrained(checkpoint, config=config)
        self.blstm = nn.LSTM(config.hidden_size, config.hidden_size // 2, bidirectional=True, num_layers=2, batch_first=True)
        self.relu1 = nn.ReLU()
        self.dropout1 = nn.Dropout(0.1)
        self.pool = nn.AdaptiveAvgPool1d(128)
        self.linear = nn.Linear(199 * 128, 1)
        self.sigmoid = nn.Sigmoid()
```

defines the model architecture

extracts features from raw audio

two-layer Bidirectional LSTM network

Rectified linear unit → introduces non-linearity into model

helps prevent overfitting by randomly zeroing out some neuron during training

→ reduces the feature map size down to a fixed size of 128 making it easier to handle variable-length input

converts output into a probability (range [0,1]), indicating whether the input is bonafide or spoof.

fully connected layer that flattens the pooled features into a single output (as a binary classification task)

Linear layer + Sigmoid → typical setup for Binary classification problems

wav2vec2 + BLSTM + fully connected layers → capture both spatial (wav2vec2) & temporal (LSTM) features from the audio & make accurate binary classification predictions.

(power architecture)

```python
def forward(self, input_ids, attention_mask):
    input_features = self.feature_extractor(input_ids, return_tensors="pt", sampling_rate=16000)
    ff = input_features.input_values
    ff = ff.squeeze(0).to(device)

    attention_mask = attention_mask.to(device)
    features = self.wav2vec2(ff, attention_mask=attention_mask, output_hidden_states=True).last_hidden_state

    res = features
    x, (h, c) = self.blstm(features)
    x = x + res  # Residual connection

    x = self.pool(x)  # Adaptive average pooling
    x = x.view(x.shape[0], -1)  # Flatten
    x = self.linear(x)  # Fully connected layer
    x = self.sigmoid(x)  # Sigmoid activation
    return x
```

↳ converts raw audio into input features compatible with wav2vec2

passes the input features through the wav2vec2 model to extract audio representations → acoustic & phonetic properties represented in high-level feature vectors

features passed through BLSTM to capture temporal dependencies in the audio

combines the original wav2vec2 features with LSTM output

hidden states from LSTM are averaged down to fixed size of 128

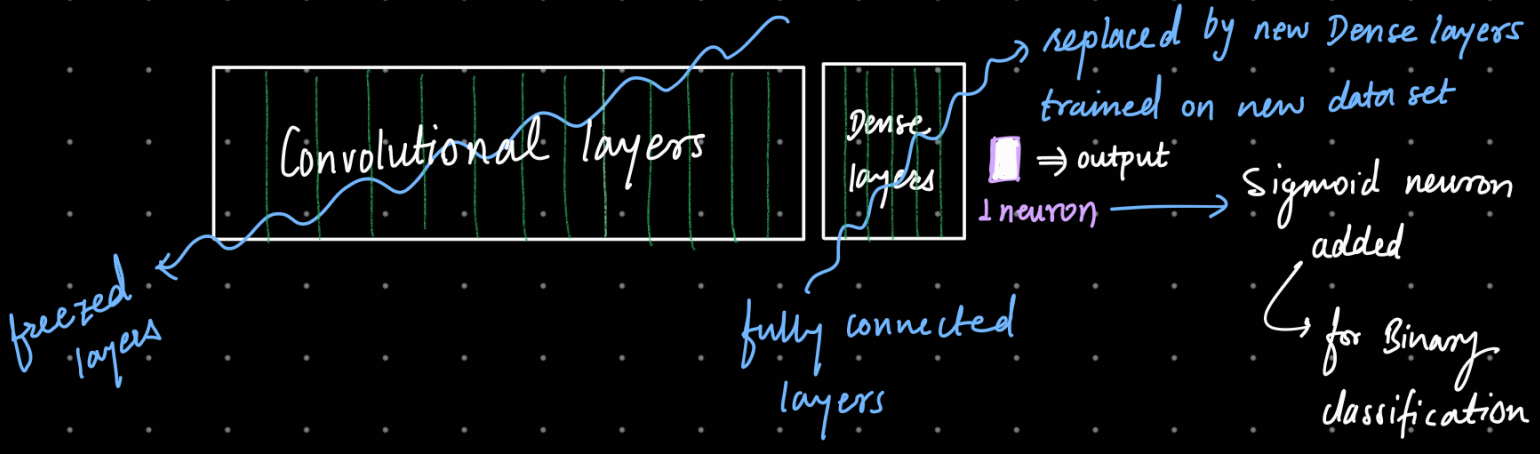
converts 3-D tensor (batchsize, timesteps, features) into 2-D so it can be passed into a linear layer

Residual Connections help preserve information from earlier layers by bypassing certain transformations.

```
checkpoint = 'facebook/wav2vec2-base-960h'
model = CustomWav2Vec2ForClassification(checkpoint)
```

→ includes wav2vec2 feature extractor + additional layers for fine-tuning
→ specific to classification task

Transfer learning → this approach allows you to fine-tune the pre-trained model with fewer labelled data, improving performance for specific tasks (like audio classification)




```
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-6)
```

→ learning rate
→ used to update the model's weights during training