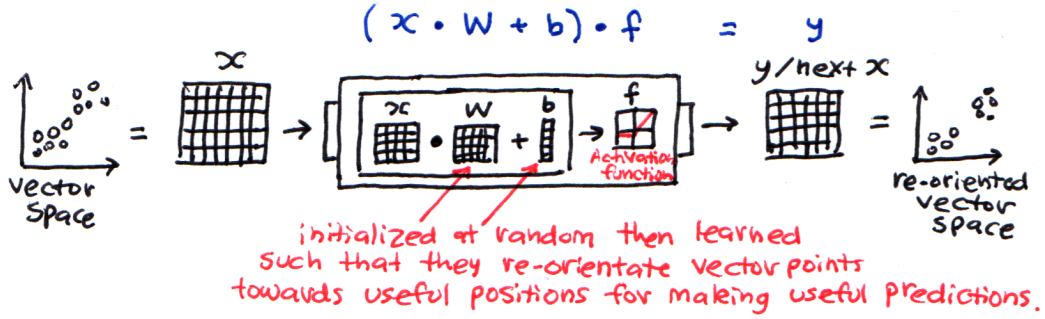
In [1]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

#for the GNN layers
from torch_geometric.nn import GCNConv, GATConv
from torch_geometric.data import Data
from torch_geometric.nn import global_mean_pool
```

In [ ]:
```
def relu(self, x):
    return np.maximum(0, x)  # ReLU activation function
```

1. Fully Connected Layer (Dense Layer) - the ANN work horse

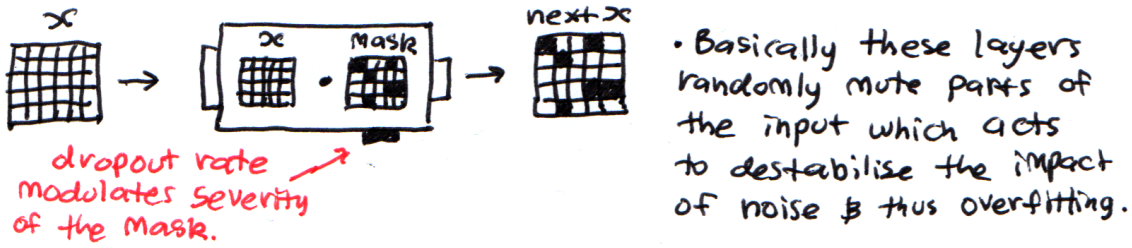
$$\mathbf{y} = f(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$$



In [ ]:
```
class FullyConnectedLayer:
    def __init__(self, input_size, output_size):
        self.W = np.random.randn(output_size, input_size)
        self.b = np.random.randn(output_size)

    def forward(self, x):
        y = self.W @ x + self.b #direction implimentation of the equation
        y = self.relu(z) #then passing it through an activation function
        return y
```

In [ ]:
```
# Example usage:
fc_layer = FullyConnectedLayer(3, 2)
input_data = np.random.randn(3)
output = fc_layer.forward(input_data)
print(output)
```

2. Dropout Layers (used for regularization in the context of ANNs)



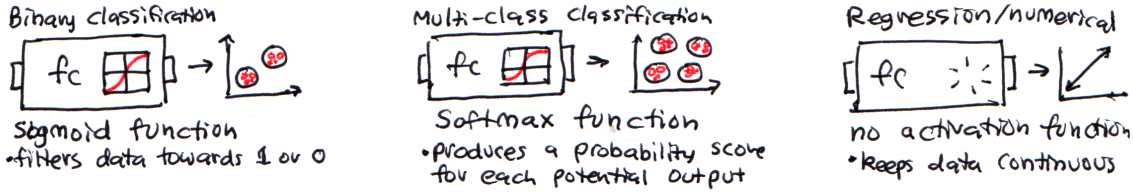
- Some of the other layer types take and express higher dimensional tensors instead of 2D matrices. The basic logic for dropout layers remains the same but instead of generate a 2D matrix as a mask, you generate a tensor matching the current state of the data. The dropout cells operate much the same, just across the entire tensor rather than just a 2D region.

In [ ]:
```
class DropoutLayer:
    def __init__(self, dropout_rate):
        self.dropout_rate = dropout_rate

    def forward(self, x, training=True):
        if training:
            mask = (np.random.rand(*x.shape) > self.dropout_rate).astype(float)
            return mask * x / (1 - self.dropout_rate)
        else:
            return x
```

In [ ]:
```
# Example usage:
dropout_layer = DropoutLayer(0.5)
input_data = np.random.randn(5)
output = dropout_layer.forward(input_data, training=True)
print(output)
```

3. Output Layer

The final layer of a network tends to be essentially just a normal fc layer, but with varying activation functions depending on the goal of the ANN.



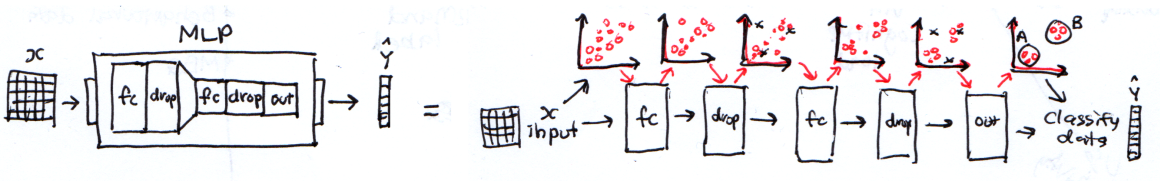
- Binary Classification ANNs end in an FC layer where the activation function is a **Sigmoid**.
- Multi-class classfication ANNs instead use a **softmax** activation function.
- Regression ANNs skip the final activation function as **the data is already continous**.

In [ ]:
```
# Define activation functions
def sigmoid(x): # Sigmoid is used for binary classification
    return 1 / (1 + np.exp(-x))

def softmax(x): # Softmax is used for multi-class classification
    exp_x = np.exp(x - np.max(x))  # Subtract max for numerical stability
    return exp_x / np.sum(exp_x)

def none_activation(x): # No activation function is used for numerical outputs
    return x
```

In [ ]:
```
class OutputLayer:
    def __init__(self, input_size, output_size, activation=None):
        self.weights = np.random.randn(output_size, input_size)
        self.biases = np.random.randn(output_size)
        self.activation = activation #set activation function based on given parameter

    def forward(self, x):
        logits = self.weights @ x + self.biases
        if self.activation is not None:
            return self.activation(logits)
        else:
            return logits  # If no activation function is provided, return raw logits
```

MLP networks (**Multilayer Perceptrons**) basically focus on re-orienting vectors in a straightforward data matrix such that the labels are coherantly distinguished. With discrete classification this means pushing them into clusters, while with numerical outputs this means pushing them towards the shape of a regression line.

-The utility of this is the ultimate point of almost all ANNs so most types (CNNs, RNNs, GNNs) will almost always functionally conclude with an MLP section. The other kinds of layers specific to other architectures generally just exist to preprocess data (applying an 'inductive bias') towards utilising these fully connected layers to the greatest degree.



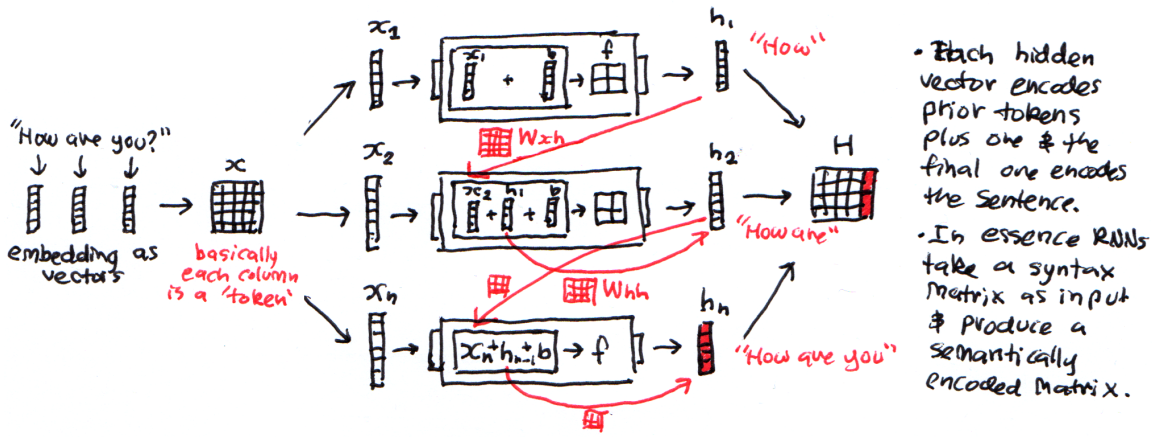
In [ ]:
```
#PyTorch Implementation
class MLP(nn.Module):
    #'hidden size' parameters dictate the size of each fc layer. This affects expressivity but is kind of arbitra
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size, dropout_prob):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)  # First fully connected layer - 'Linear' is the preconstr
        self.dropout1 = nn.Dropout(dropout_prob)        # First dropout layer
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)  # Second fully connected layer
        self.dropout2 = nn.Dropout(dropout_prob)        # Second dropout layer
        self.output = nn.Linear(hidden_size2, output_size)  # Output layer
```

```
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout1(x)
        x = F.relu(self.fc2(x))
        x = self.dropout2(x)
        x = self.output(x)
        return x
```

In [ ]:
```
# Example usage:
# Set input size, hidden layer sizes, and output size
input_size = 10  # Number of input features
hidden_size1 = 50  # Size of first hidden layer
hidden_size2 = 20  # Size of second hidden layer
output_size = 2  # Size of output layer (e.g., 2 classes for binary classification)
dropout_prob = 0.5  # Dropout probability (50% of neurons dropped during training)

# Initialize the MLP model
mlp_model = MLP(input_size, hidden_size1, hidden_size2, output_size, dropout_prob)

# Example input (batch size of 5, with 10 input features each)
input_data = torch.randn(5, input_size)

# Forward pass through the model
output = mlp_model(input_data)
print(output)
```

4. Recurrent Layer

$$\mathbf{h}_t = \tanh(\mathbf{W}_{xh} \cdot \mathbf{x}_t + \mathbf{W}_{hh} \cdot \mathbf{h}_{t-1} + \mathbf{b}_h)$$



In [ ]:
```
class RNNLayer:
    def __init__(self, input_size, hidden_size):
        self.W_xh = np.random.randn(hidden_size, input_size)
        self.W_hh = np.random.randn(hidden_size, hidden_size)
        self.b_h = np.random.randn(hidden_size)

    def forward(self, x_sequence):
        hidden_state = np.zeros(self.W_hh.shape[0])
        hidden_states = []
        for x_t in x_sequence:
            hidden_state = np.tanh(np.dot(self.W_xh, x_t) + np.dot(self.W_hh, hidden_state) + self.b_h)
            hidden_states.append(hidden_state)
        return np.array(hidden_states)
```

In [ ]:
```
# Example usage:
rnn_layer = RNNLayer(3, 4)
input_sequence = [np.random.randn(3) for _ in range(5)]
output = rnn_layer.forward(input_sequence)
print(output)
```

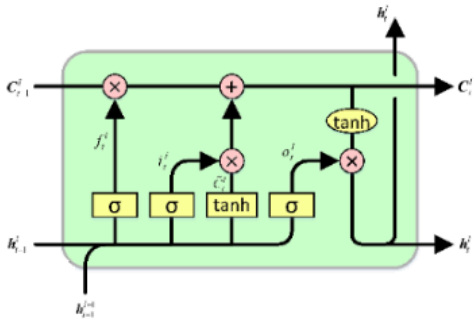
5. LSTM Layer (RNN Layer which can learn longer distance temporal dependencies)

Equations:

$$
\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}_i \cdot \mathbf{x}_t + \mathbf{U}_i \cdot \mathbf{h}_{t-1} + \mathbf{b}_i) \\
\mathbf{f}_t &= \sigma(\mathbf{W}_f \cdot \mathbf{x}_t + \mathbf{U}_f \cdot \mathbf{h}_{t-1} + \mathbf{b}_f) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_o \cdot \mathbf{x}_t + \mathbf{U}_o \cdot \mathbf{h}_{t-1} + \mathbf{b}_o) \\
\mathbf{g}_t &= \tanh(\mathbf{W}_g \cdot \mathbf{x}_t + \mathbf{U}_g \cdot \mathbf{h}_{t-1} + \mathbf{b}_g) \\
\mathbf{c}_t &= \mathbf{f}_t \cdot \mathbf{c}_{t-1} + \mathbf{i}_t \cdot \mathbf{g}_t \\
\mathbf{h}_t &= \mathbf{o}_t \cdot \tanh(\mathbf{c}_t)
\end{aligned}
$$

Where:

- $\mathbf{i}_t, \mathbf{f}_t, \mathbf{o}_t$ are the input, forget, and output gates.
- $\mathbf{c}_t$ is the cell state, and $\mathbf{h}_t$ is the hidden state.

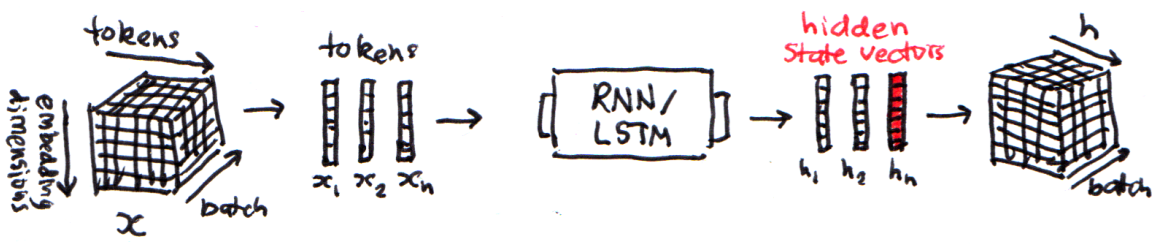


In [ ]:
```
class LSTMLayer:
    def __init__(self, input_size, hidden_size):
        self.W_i = np.random.randn(hidden_size, input_size)
        self.U_i = np.random.randn(hidden_size, hidden_size)
        self.b_i = np.random.randn(hidden_size)
        self.W_f = np.random.randn(hidden_size, input_size)
        self.U_f = np.random.randn(hidden_size, hidden_size)
        self.b_f = np.random.randn(hidden_size)
        self.W_o = np.random.randn(hidden_size, input_size)
        self.U_o = np.random.randn(hidden_size, hidden_size)
        self.b_o = np.random.randn(hidden_size)
        self.W_g = np.random.randn(hidden_size, input_size)
        self.U_g = np.random.randn(hidden_size, hidden_size)
        self.b_g = np.random.randn(hidden_size)

    def forward(self, x_sequence):
        h_t, c_t = np.zeros(self.W_i.shape[0]), np.zeros(self.W_i.shape[0])
        hidden_states = []
        for x_t in x_sequence:
            i_t = self.sigmoid(np.dot(self.W_i, x_t) + np.dot(self.U_i, h_t) + self.b_i)
            f_t = self.sigmoid(np.dot(self.W_f, x_t) + np.dot(self.U_f, h_t) + self.b_f)
            o_t = self.sigmoid(np.dot(self.W_o, x_t) + np.dot(self.U_o, h_t) + self.b_o)
            g_t = np.tanh(np.dot(self.W_g, x_t) + np.dot(self.U_g, h_t) + self.b_g)
            c_t = f_t * c_t + i_t * g_t
            h_t = o_t * np.tanh(c_t)
            hidden_states.append(h_t)
        return np.array(hidden_states)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
```

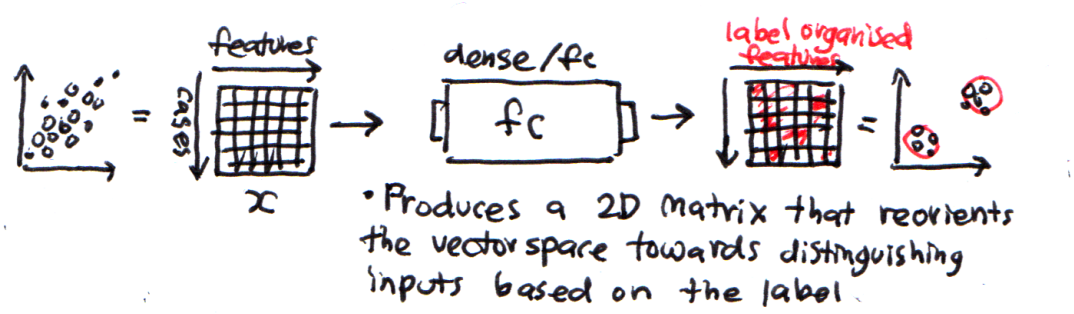
In [ ]:
```
# Example usage:
lstm_layer = LSTMLayer(3, 4)
input_sequence = [np.random.randn(3) for _ in range(5)]
output = lstm_layer.forward(input_sequence)
print(output)
```

6. Flatten & Reshape Layers (Layers used to convert between different kinds of layers)

- Recurrent neural networks tend to conclude in a fully connected layer or two- effectively meaning that RNNs tend to append an MLP onto the end of them. This kind of modular mix and matching can be generalised beyond this too, but sometimes this will need some alignment corrections to be achieved.
- With RNNs, they tend to take and express a 3D tensor as input and output (the extra dimension comes from utilising batches)



- This differs from MLP networks which tend to be given 2D matrices.



To bridge this gap, we have flatten layers. Essentially these implement some method for trimming down the tensor expressed by recurrent layers. This can be done in a few ways.

- take the final timestep tokens only and just lay them out as a matrix. Columns basically become input batches (replacing tokens) and rows become embedding dimensions.
- you can flatten the sequences so that the other timesteps are also incorporated into the end result.

The point is that the flatten layer bridges the gap between these kinds of layers.

You should also be mindful of what the outputs 'mean'.

- The output of an FC layer basically represents a distorted vectorspace that relates outputs back to inputs.
- the output of RNN layers is a contextual encoding of a sequence of tokens. If flattened, the intermediary parts of the encodings may be ommitted.

In [ ]:
```python
class LSTM_FC_Model(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(LSTM_FC_Model, self).__init__()
        self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size, num_layers=num_layers, batch_first=Tr
        self.fc = nn.Linear(hidden_size * sequence_length, num_classes)  # Ensure the input size is consistent wi

    def forward(self, x):
        lstm_out, (hn, cn) = self.lstm(x)  # lstm_out: (batch_size, sequence_length, hidden_size)
        flattened = lstm_out.reshape(lstm_out.size(0), -1) #followed by a reshaping layer
        #last_hidden_state (alt to flattened) = hn[-1]  # Take the last layer's hidden state
        out = self.fc(flattened)
        return out
```

In [ ]:
```python
# Set up example parameters
batch_size = 32
sequence_length = 10  # Length of the sequence
input_size = 20       # Size of the input features at each time step
hidden_size = 50      # Number of LSTM units
num_layers = 2        # Number of LSTM layers
num_classes = 5       # Number of output classes

# Instantiate the model
model = LSTM_FC_Model(input_size=input_size, hidden_size=hidden_size, num_layers=num_layers, num_classes=num_clas

# Example input: (batch_size, sequence_length, input_size)
input_data = torch.randn(batch_size, sequence_length, input_size)

# Forward pass through the model
output = model(input_data)

# Output shape: (batch_size, num_classes)
print("Output shape:", output.shape)
```

7. Convolutional Layers & Pooling Layers

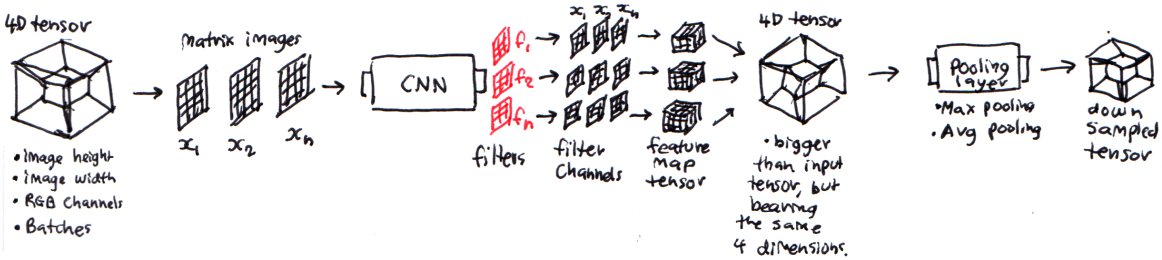
5. Convolutional Layer

Equation:

$$y(i,j) = \sum_{m=1}^{K} \sum_{n=1}^{K} w(m,n) \cdot x(i+m, j+n)$$

Where:

- $x(i+m, j+n)$ is the input patch.
- $w(m,n)$ is the filter.
- $y(i,j)$ is the output feature map.

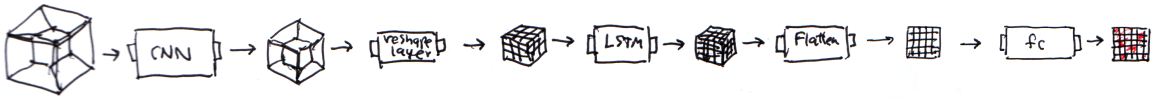


- CNN layers take and express a 4D tensor which tends to bloat on the output due to all the filters producing destinct reversions of the input tensor as feature maps. **Pooling Layers** help to reduce this bloating (but don't confuse them with reshaping layers. they lower the resolution- they don't change the amount of dimensions present).

In [ ]:
```python
class Convolution2D:
    def __init__(self, num_filters, filter_size):
        self.num_filters = num_filters
        self.filter_size = filter_size
        # Initialize filters with random values
        self.filters = np.random.randn(num_filters, filter_size, filter_size)

    def apply_filter(self, input_patch, filter):
        # Element-wise multiplication and sum the result
        return np.sum(input_patch * filter)

    def forward(self, input_image):
        # Input image size (assuming square shape and grayscale)
        input_size = input_image.shape[0]
        output_size = input_size - self.filter_size + 1

        # Create an empty output feature map
        output_feature_map = np.zeros((self.num_filters, output_size, output_size))

        # Apply each filter over the image
        for f in range(self.num_filters):
            filter = self.filters[f]
            for i in range(output_size):
                for j in range(output_size):
                    # Extract input patch of the same size as the filter
                    input_patch = input_image[i:i+self.filter_size, j:j+self.filter_size]
                    # Apply the filter to the patch and store the result
                    output_feature_map[f, i, j] = self.apply_filter(input_patch, filter)

        return output_feature_map
```

In [ ]:
```
# Example usage:
# Define a random input image (e.g., 28x28 grayscale image)
input_image = np.random.randn(28, 28)

# Create a Convolutional Layer with 8 filters, each of size 3x3
conv_layer = Convolution2D(num_filters=8, filter_size=3)

# Perform the forward pass
output = conv_layer.forward(input_image)

# Print the shape of the output feature map
print("Output feature map shape:", output.shape)
```

So If a CNN was to have a sequential component- e.g. LSTM, incorporated into it, then **reshape** layers can help convert the 4D tensor into a 3D one. It can be used in other such ways too. A **flatten** layer would then also be needed for going from LSTM to FC.



In [ ]:
```
import torch
import torch.nn as nn

# Define the custom model class
class ConvLSTM_FC_Model(nn.Module):
    def __init__(self, input_channels, conv_out_channels, lstm_hidden_size, num_classes):
        super(ConvLSTM_FC_Model, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=input_channels, out_channels=conv_out_channels, kernel_size=3, paddin
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.lstm = nn.LSTM(input_size=conv_out_channels, hidden_size=lstm_hidden_size, batch_first=True)
        self.fc = nn.Linear(lstm_hidden_size, num_classes)

    def forward(self, x):
        # x shape: (batch_size, input_channels, height, width)

        x = self.conv1(x)  # Output shape: (batch_size, conv_out_channels, height, width)
        x = self.pool(x)  # Output shape: (batch_size, conv_out_channels, height/2, width/2)

        # Reshape for LSTM: (batch_size, sequence_length, conv_out_channels)
        # Here, height * width is used as the sequence length for LSTM.
        batch_size, conv_out_channels, height, width = x.size()
        x = x.view(batch_size, height * width, conv_out_channels)  # Reshaping

        lstm_out, (hn, cn) = self.lstm(x)  # lstm_out: (batch_size, sequence_length, lstm_hidden_size)
        lstm_out_last = lstm_out[:, -1, :]  # Shape: (batch_size, lstm_hidden_size)
        out = self.fc(lstm_out_last)  # Shape: (batch_size, num_classes)

        return out
```

In [ ]:
```
# Set up example parameters
input_channels = 1  # e.g., grayscale images with 1 channel
conv_out_channels = 16  # Number of filters in the Conv layer
lstm_hidden_size = 32  # Number of LSTM units
num_classes = 10  # Number of output classes

# Instantiate the model
model = ConvLSTM_FC_Model(input_channels=input_channels, conv_out_channels=conv_out_channels, lstm_hidden_size=ls

# Example input: batch of 8 grayscale images (1 channel) of size 28x28
input_data = torch.randn(8, 1, 28, 28)

# Forward pass through the model
output = model(input_data)

# Output shape: (batch_size, num_classes)
print("Output shape:", output.shape)
```

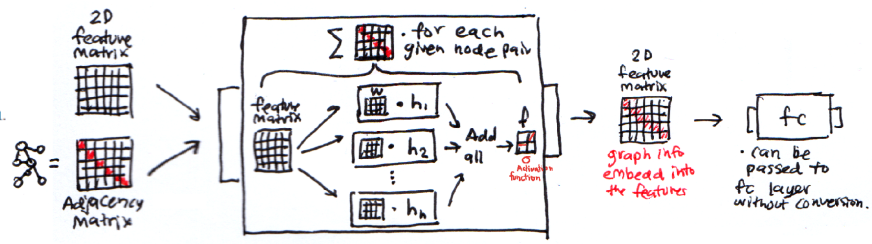
8. GNN Layers (there's a vanilla version, and two main types that see use: GCN, & GAT)

Vanilla GNN:

$$h_v^{(l+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} \mathbf{W} \cdot h_u^{(l)}\right)$$

Where:

- $h_v^{(l+1)}$ is the updated feature of node v at layer $l+1$.
- $h_u^{(l)}$ is the feature of neighboring node u at layer l.
- $\mathbf{W}$ is a shared learnable weight matrix.
- σ is a non-linear activation function (e.g., ReLU).



In [ ]:
```
class GNNLayer:
    def __init__(self, input_dim, output_dim):
        # Initialize weight matrix for node feature transformation
        self.weights = np.random.randn(input_dim, output_dim)

    def forward(self, node_features, adj_matrix):
        # Aggregate neighbor features using the adjacency matrix
        neighbor_features = np.dot(adj_matrix, node_features)
        # Apply a linear transformation to the aggregated features
        output_features = np.dot(neighbor_features, self.weights)
        return output_features
```

In [ ]:
```
# Example usage:
# Adjacency matrix (3 nodes connected in a simple graph)
adj_matrix = np.array([
    [1, 1, 0],  # Node 1 is connected to itself and Node 2
    [1, 1, 1],  # Node 2 is connected to all nodes
    [0, 1, 1]   # Node 3 is connected to itself and Node 2
])

# Node features for each of the 3 nodes (3-dimensional features per node)
node_features = np.random.randn(3, 3)

# Create a GNN layer with input dimension 3 and output dimension 2
gnn_layer = GNNLayer(input_dim=3, output_dim=2)

# Perform the forward pass
output = gnn_layer.forward(node_features, adj_matrix)

# Print the new node features
print("New node features after GNN layer:", output)
```

The equation for one layer of a GCN is:

$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{D}}^{-\frac{1}{2}}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right)$$

Where:

- $\mathbf{H}^{(l)}$ is the node feature matrix at layer l (e.g., $\mathbf{H}^{(0)}$ is the input feature matrix).
- $\hat{\mathbf{A}}$ is the **adjacency matrix** of the graph with added self-loops (i.e., edges from each node to itself).
- $\hat{\mathbf{D}}$ is the **degree matrix**, which is a diagonal matrix where each element $\hat{\mathbf{D}}_{ii}$ is the degree of node i (i.e., the number of edges connected to node i).
- $\mathbf{W}^{(l)}$ is the **weight matrix** of the layer, which is learned during training.
- σ is a non-linear activation function (e.g., ReLU).
- $\mathbf{H}^{(l+1)}$ is the updated node feature matrix at layer $l+1$.

In [ ]:
```
class GCNLayer:
    def __init__(self, in_features, out_features):
        # Initialize weight matrix
        self.W = np.random.randn(in_features, out_features)

    def forward(self, X, adj):
        # Normalize adjacency matrix
        D = np.diag(np.sum(adj, axis=1))
```

```
        D_inv_sqrt = np.linalg.inv(np.sqrt(D))
        adj_normalized = np.dot(D_inv_sqrt, np.dot(adj, D_inv_sqrt))

        # Perform feature aggregation
        H_prime = np.dot(adj_normalized, X)

        # Apply linear transformation
        H_new = np.dot(H_prime, self.W)

        # Apply activation function (ReLU)
        return np.maximum(0, H_new)
```

In [ ]:

```
# Example usage:
adj = np.array([[1, 1, 0], [1, 1, 1], [0, 1, 1]])  # Adjacency matrix with self-loops
X = np.random.randn(3, 4)  # Node features (3 nodes, 4 features per node)

gcn_layer = GCNLayer(4, 2)  # Input 4 features, output 2 features
output = gcn_layer.forward(X, adj)

print("Updated node features after GCN layer:")
print(output)
```

Graph Attention Network (GAT) Overview

In a **Graph Attention Network (GAT)**, the attention mechanism allows each node to **attend** to its neighbors with different weights, rather than treating all neighbors equally (as in Graph Convolutional Networks). This attention mechanism dynamically calculates how much **importance** each neighboring node has for a given node during message passing.

Attention Mechanism in GAT

The attention mechanism in GAT involves the following steps:

1. **Linear Transformation**: First, each node's feature vector is transformed using a shared weight matrix $\mathbf{W}$:

$$\mathbf{h}_i' = \mathbf{W}\mathbf{h}_i$$

   Where:

   - $\mathbf{h}_i$ is the input feature vector of node i.
   - $\mathbf{W}$ is the learnable weight matrix.
   - $\mathbf{h}_i'$ is the transformed feature vector for node i.

2. **Compute Attention Coefficients**: For each pair of connected nodes i and j (i.e., for each edge in the graph), we compute the **attention coefficient** e_{ij}. This coefficient determines how important node j is to node i:

$$e_{ij} = \text{LeakyReLU}\left(\mathbf{a}^T \left[\mathbf{h}_i' \parallel \mathbf{h}_j'\right]\right)$$

   Where:

   - $\mathbf{a}$ is a learnable weight vector.
   - $\parallel$ represents concatenation of the feature vectors $\mathbf{h}_i'$ and $\mathbf{h}_j'$.
   - LeakyReLU is the activation function applied to the computed score.

3. **Softmax Normalization**: The raw attention coefficients e_{ij} are then normalized across all neighbors j of node i using the softmax function to obtain the final **attention weights** α_{ij}:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})}$$

   This ensures that the attention weights sum to 1 across all neighbors of node i.

4. **Message Passing**: The new representation for node i is computed by **aggregating** the feature vectors of its neighbors j, weighted by the attention coefficients α_{ij}:

$$\mathbf{h}_i^{(new)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij}\mathbf{h}_j'\right)$$

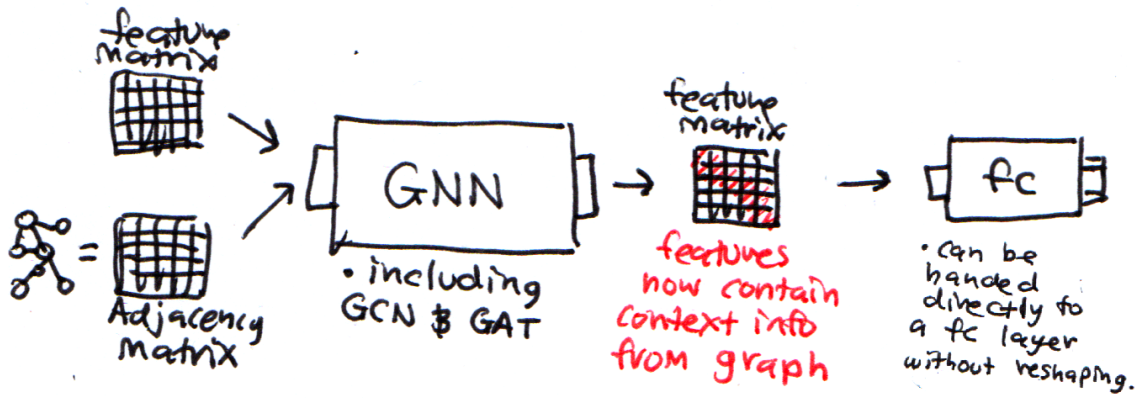
   Where:

   - $\mathbf{h}_i^{(new)}$ is the updated feature vector of node i.
   - σ is a non-linear activation function (e.g., ReLU).

In [ ]:
```python
class GATLayer:
    def __init__(self, in_features, out_features):
        # Initialize the weight matrix W and attention weights vector a
        self.W = np.random.randn(in_features, out_features)
        self.a = np.random.randn(2 * out_features)  # For concatenated h'_i and h'_j

    def forward(self, x, adj):
        # Linear transformation
        h_prime = np.dot(x, self.W)

        num_nodes = h_prime.shape[0]
        attention_coeffs = np.zeros((num_nodes, num_nodes))

        # Compute attention scores for each pair of connected nodes
        for i in range(num_nodes):
            for j in range(num_nodes):
                if adj[i, j] == 1:  # Only compute for connected nodes
                    # Concatenate h'_i and h'_j and apply attention mechanism
                    concatenated = np.concatenate([h_prime[i], h_prime[j]])
                    e_ij = np.dot(self.a, concatenated)  # Inner product with attention weights
                    attention_coeffs[i, j] = F.leaky_relu(torch.tensor(e_ij), negative_slope=0.2).item()

        # Normalize attention scores with softmax
        attention_weights = np.zeros((num_nodes, num_nodes))
        for i in range(num_nodes):
            attention_weights[i] = np.exp(attention_coeffs[i]) / np.sum(np.exp(attention_coeffs[i]))

        # Apply attention to aggregate features from neighbors
        new_features = np.zeros(h_prime.shape)
        for i in range(num_nodes):
            for j in range(num_nodes):
                if adj[i, j] == 1:
                    new_features[i] += attention_weights[i, j] * h_prime[j]

        return new_features
```

In [ ]:
```python
# Example usage:
adj = np.array([[0, 1, 1], [1, 0, 1], [1, 1, 0]])  # Adjacency matrix
x = np.random.randn(3, 4)  # Node features (3 nodes, 4-dimensional features)

gat_layer = GATLayer(4, 3)
output = gat_layer.forward(x, adj)
print("New node features after applying GAT:", output)
```

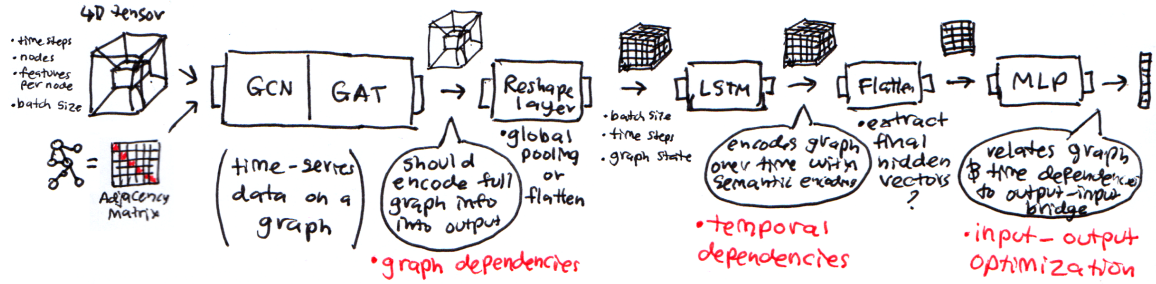
With all of the GNN layer types (GCN, GAT included), they all basically take the same input (two 2D matrices) and express one output- a 2D matrix. This makes these networks easily compatible with Fully connected layers/ MLP segments without the need for conversion, though one should be mindful of what the output represents: a feature matrix encoded with relational dependency information.



To construct a full network with both GNN and RNN elements though, this can be a little awkward as this means the temporal information needs to be available to the initial input, so a customised version of GNN layers may be needed so that the adjacency and feature information is presented while also having the sequential and batch information available.

- In simplified terms you're really just making a composition of the LSTM tensor and the GNN adjacency/ feature matrices as a 4D tensor. This can work fine as long as each layer knows where to look for its relevant information in the tensor while being free to ignore the sections it isn't using.

Proposed Hybrid Architecture



Input (2):

- 2D Adjacency Matrix (Essentially covariance matrix of all HCP parcels)
- 4D Tensor containing:
  - Nodes (length determined by amount of nodes, so 360 due to parcel count)
  - Features (length determined by amount of features used to describe nodes- degree centrality etc)
  - Time steps (length determined by steps in BOLD time series, this may use padding and LSTM will need to configure for this)
  - Batch size (e.g. 32)

In [ ]:
```
class GCN_LSTM_Model(nn.Module):
    def __init__(self, in_channels, gcn_out_channels, gat_out_channels, lstm_hidden_size, lstm_num_layers, fc_hi
        super(GCN_GAT_LSTM_Model, self).__init__()

        self.gcn = GCNConv(in_channels, gcn_out_channels) # GCN Layer
        self.gat = GATConv(gcn_out_channels, gat_out_channels, heads=1, concat=True) # GAT Layer

        self.lstm1 = nn.LSTM(input_size=gat_out_channels, hidden_size=lstm_hidden_size, num_layers=lstm_num_layer
        self.lstm2 = nn.LSTM(input_size=lstm_hidden_size, hidden_size=lstm_hidden_size, num_layers=lstm_num_layer

        self.fc1 = nn.Linear(lstm_hidden_size, fc_hidden_size)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(fc_hidden_size, num_classes)

    def forward(self, x, edge_index, batch):
        # x shape: (batch_size * time_steps, num_nodes, in_channels)
        # edge_index shape: (2, num_edges)

        x = self.gcn(x, edge_index)  #GCN Layer, Output shape: (batch_size * time_steps, num_nodes, gcn_out_chann
        x = self.gat(x, edge_index)  # GAT Layer, Output shape: (batch_size * time_steps, num_nodes, gat_out_char

        x = global_mean_pool(x, batch)  # Reshape for LSTM by pooling over nodes, Output: (batch_size * time_step
        x, _ = self.lstm1(x)  # Output shape: (batch_size, time_steps, lstm_hidden_size)
        x, _ = self.lstm2(x)  # Output shape: (batch_size, time_steps, lstm_hidden_size)

        x = x[:, -1, :]  # Use the last time step's output for the final decision (LSTM summary), Shape: (batch_s

        x = F.relu(self.fc1(x))  #FC Layer, Shape: (batch_size, fc_hidden_size)
        x = self.dropout(x)
        x = self.fc2(x)  # Shape: (batch_size, num_classes)

        return x
```

In [ ]:
```
# Example input setup
batch_size = 32
time_steps = 10  # e.g., 10 time points in the BOLD time series
num_nodes = 200  # e.g., 200 brain parcels
num_node_features = 16  # e.g., BOLD signals + additional features
gcn_out_channels = 32
gat_out_channels = 64
lstm_hidden_size = 128
lstm_num_layers = 2
fc_hidden_size = 64
num_classes = 1  # e.g., numerical prediction task (e.g., decoding a cognitive score)

# Simulate input data for batch of graphs
node_features = torch.randn((batch_size * time_steps * num_nodes, num_node_features))  # Node feature matrix (ba
edge_index = torch.tensor([[0, 1, 2], [1, 2, 0]], dtype=torch.long)  # Edge index for the graph (assuming same g
batch = torch.zeros(batch_size * time_steps * num_nodes, dtype=torch.long)  # Batch vector for pooling

# Instantiate and run the model
model = GCN_LSTM_Modell(in_channels=num_node_features,
                        gcn_out_channels=gcn_out_channels,
```

```
                        gat_out_channels=gat_out_channels,
                        lstm_hidden_size=lstm_hidden_size,
                        lstm_num_layers=lstm_num_layers,
                        fc_hidden_size=fc_hidden_size,
                        num_classes=num_classes)

# Forward pass through the model
output = model(node_features, edge_index, batch)

# Output shape should be (batch_size, num_classes)
print("Output shape:", output.shape)
```