



Stage d'été – Semaine 4

QGNN-TimeCausality

Avancement du projet



Stage d'été – Semaine 4

01

Updates

Update concernant l'accès au IBM Q



02

Implementation (clustering)

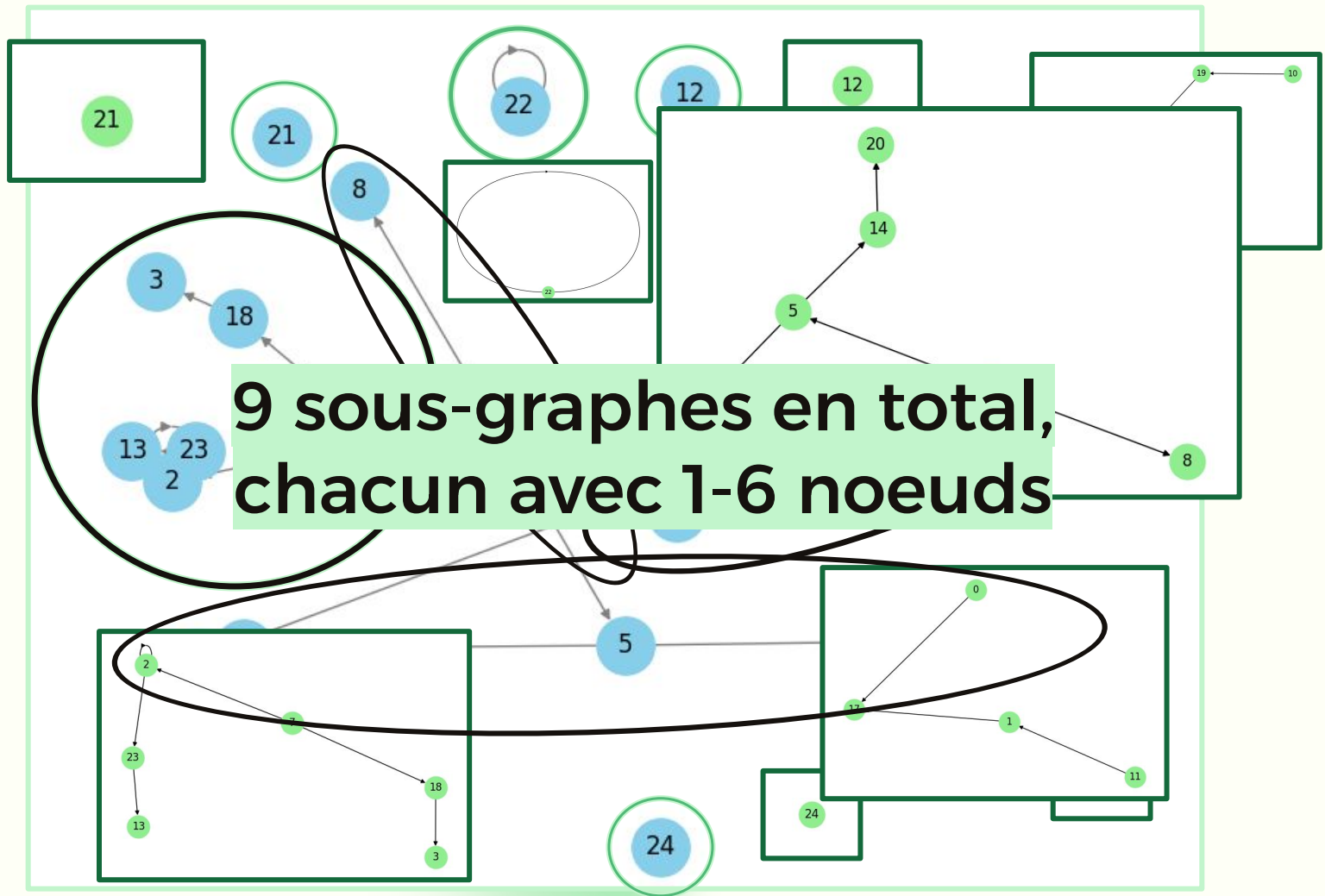
Application du tuto de pennylane

- Le graphe contient **25 nœuds** (variables: **25** portefeuilles sur **4000** jours.). Mais, Implémenter un QGRNN avec 25 qubits n'est pas optimal (le tutoriel PennyLane utilise seulement **4 qubits**).

→ Regrouper les nœuds en sous-graphes de taille réduite (environ **1-6 qubits chacun**) via une méthode de **clustering**.

```
# 2. CLUSTERING DES NŒUDS =====  
  
def cluster_graph(G):  
    communities = list(greedy_modularity_communities(G))  
    return [sorted(list(c)) for c in communities] # transforme chaque ensemble de nœuds c en liste triée  
    # retour final est donc une liste de listes ( chaque sous-liste représentant un cluster de nœuds. )
```

- Méthode utilisée : Utilisation de l'algorithme `greedy_modularity_communities` de NetworkX, basé sur la modularité.
- → Cet algorithme détecte les communautés de nœuds *densément connectés entre eux* et *faiblement connectés aux autres*.
- But du clustering : Regrouper les variables ayant **beaucoup de relations causales entre elles**, formant des sous-systèmes cohérents à analyser localement avec un circuit quantique.
- → L'algorithme de clustering **ignore** le sens des arêtes. (Il traite le graphe comme non orienté, ce qui signifie que la *direction des relations causales n'est pas prise en compte à cette étape.*) → Est-il nécessaire d'utiliser un algorithme de clustering qui prend en compte l'orientation des relations causales pour une meilleure fidélité au graphe causal ?





03

Implémentation (QGRNN)

Définition du circuit QGRNN (QGNN_layer)

L'objectif est de traiter chaque cluster du graphe comme un petit sous-graphe, encodé sur un nombre réduit de qubits.

- Fonction 'qgrnn_layer' : implémente une couche de message passing quantique :

→ Nous utilisons une **angle embedding** via une **porte de rotation RY**, qui **encode l'information** (ou "feature") du nœud **dans un qubit** avec un *angle de rotation*. [valeur réelle → état quantique d'un qubit]

```
# 3. DÉFINITION DU CIRCUIT QGRNN =====  
  
def qgrnn_layer(graph, weights, index_map): # Couche QGNN simple : RY + CZ pour message passing  
    for node, idx in index_map.items():  
        qml.RY(weights[idx], wires=idx) # Rotation paramétrée RY pour encoder les messages des nœuds  
  
    for i, j in graph.to_undirected().edges():  
        if index_map[i] != index_map[j]:  
            qml.CZ(wires=[index_map[i], index_map[j]]) # Entanglement entre qubits voisins avec CZ (message passing)
```


Définition du circuit QGRNN (QGNN_layer)

Ensuite, on convertit le graphe a un graphe non-orienté pour appliquer les portes CZ de manière symétrique entre les qubits connectés et pour chaque arête du graphe on applique une porte CZ entre les qubits connectés.

- → Cela crée de l'intrication (entanglement) entre les qubits, simulant ainsi la propagation d'informations (message passing) entre nœuds voisins dans le graphe ce qui est essentielle pour capturer les dépendances structurelles entre les nœuds.

```
# 3. DÉFINITION DU CIRCUIT QGRNN =====  
  
def qgrnn_layer(graph, weights, index_map): # Couche QGNN simple : RY + CZ pour message passing  
    for node, idx in index_map.items():  
        qml.RY(weights[idx], wires=idx) # Rotation paramétrée RY pour encoder les messages des nœuds  
  
    for i, j in graph.to_undirected().edges():  
        if index_map[i] != index_map[j]:  
            qml.CZ(wires=[index_map[i], index_map[j]]) # Entanglement entre qubits voisins avec CZ (message passing)
```

→ Ceci est une simplification du modèle QGNN proposé dans le tutoriel PennyLane, et il est plus adapté à notre cas où les sous-graphes (clusters) peuvent être traités indépendamment.

Définition du circuit QGRNN

L'objectif est de traiter chaque cluster du graphe comme un petit sous-graphe, encodé sur un nombre réduit de qubits.

- Fonction 'create_qgrnn_cluster_circuit' : génère dynamiquement un "QNode" (circuit quantique) pour un cluster donné :
 - Nombre de qubits : #de nœuds dans le cluster.
 - Encodage des features : Chaque feature (valeur à un instant donné) est encodée via une rotation RY sur un qubit dédié.
 - Propagation quantique : La couche "qgrnn_layer" est ensuite appliquée pour propager l'information dans le graphe.
 - Sortie du circuit : Le circuit retourne les espérances de mesure selon PauliZ pour chaque qubit, ce qui donne une représentation réelle du cluster.

```
def create_qgrnn_cluster_circuit(graph):  
    n_qubits = graph.number_of_nodes()  
    dev = qml.device("default.qubit", wires=n_qubits)  
  
    @qml.qnode(dev)  
    def circuit(inputs, weights):  
        node_list = list(graph.nodes())  
        index_map = {node: idx for idx, node in enumerate(node_list)}  
  
        for idx, node in enumerate(node_list):  
            qml.RY(inputs[idx], wires=idx) # Encodage des features avec  
  
        qgrnn_layer(graph, weights, index_map)  
  
        return [qml.expval(qml.PauliZ(idx)) for idx in range(n_qubits)]  
  
    return circuit, dev
```

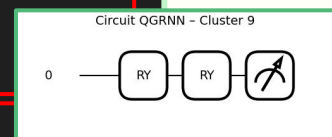
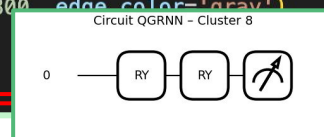
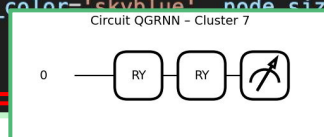
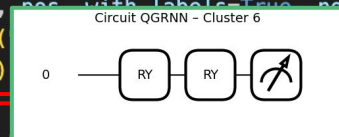
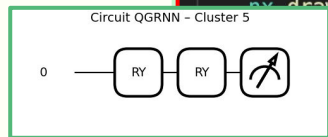
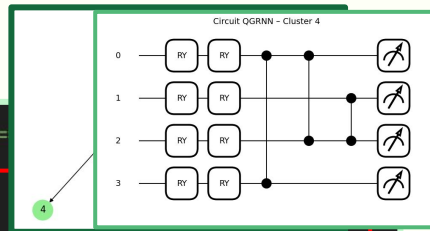
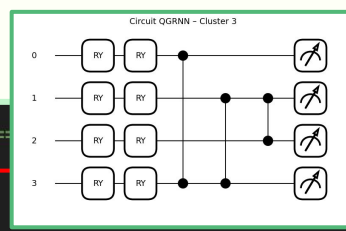
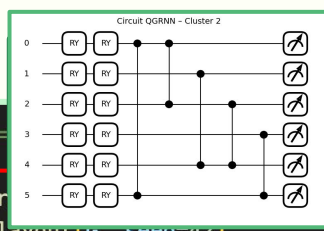
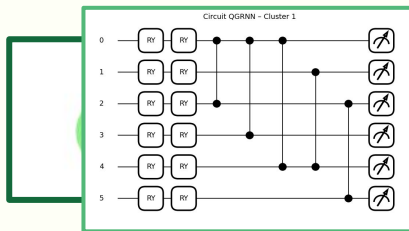
(À la fin du circuit, chaque qubit est mesuré selon l'observable Pauli-Z. Cette opération retourne une valeur réelle entre -1 et +1, représentant l'état final moyen du qubit [L'espérance de mesure : la moyenne des résultats qu'on obtiendrait si on répétait la mesure plusieurs fois])

NB : Ici j'ai pas encore spécifié le nombre de shots, j'ai juste utilisé le mode analytique (par défaut)



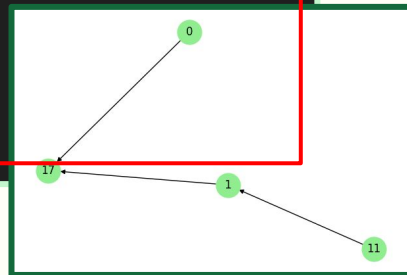
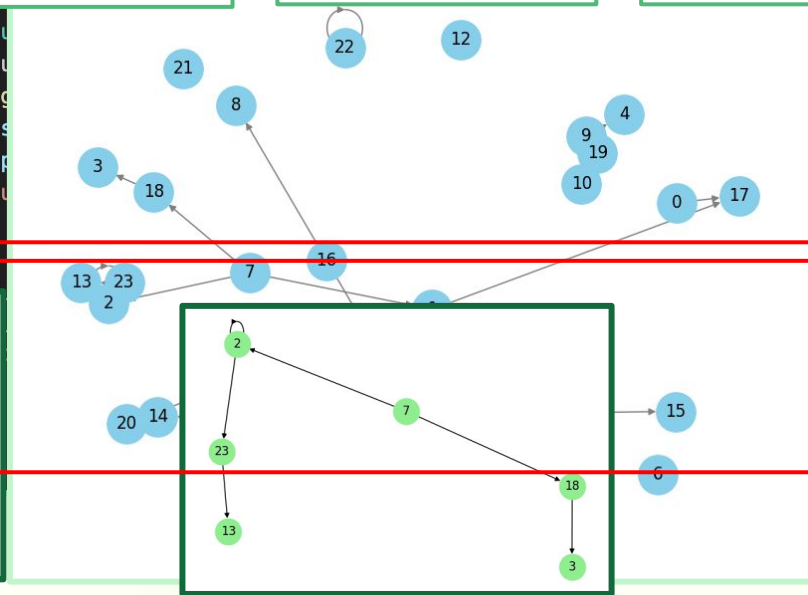
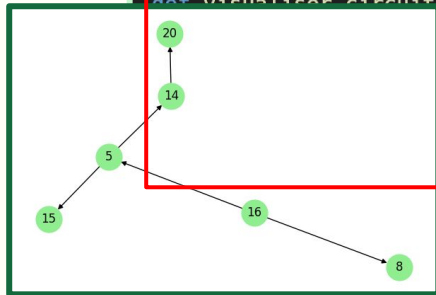
04

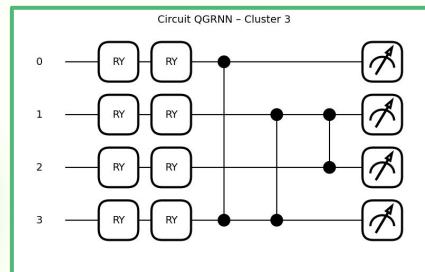
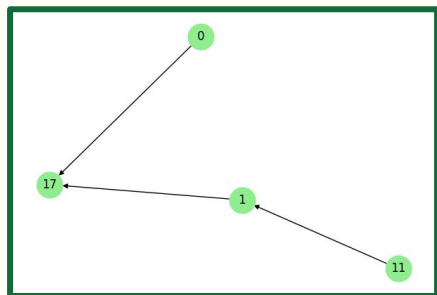
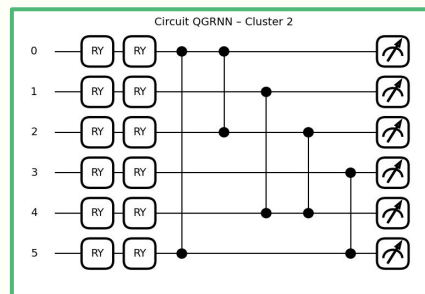
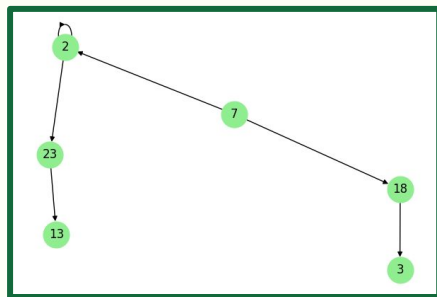
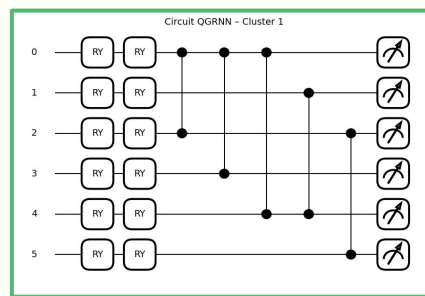
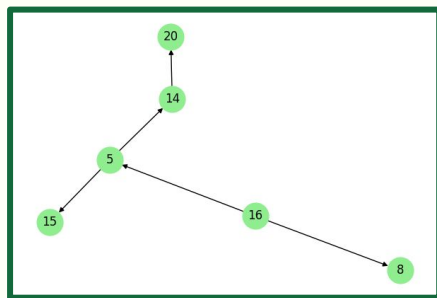
Implementation (Visualisation)

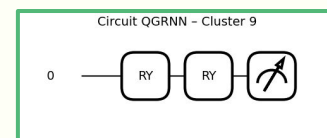
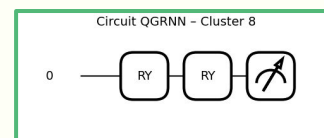
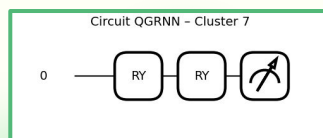
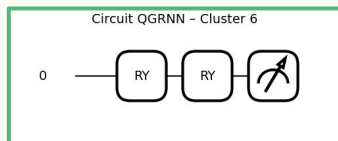
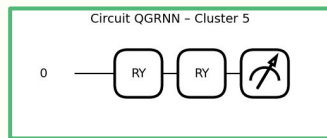
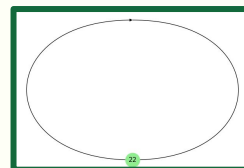
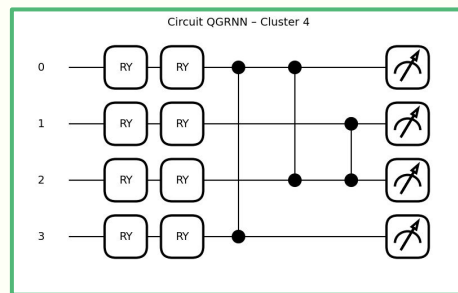
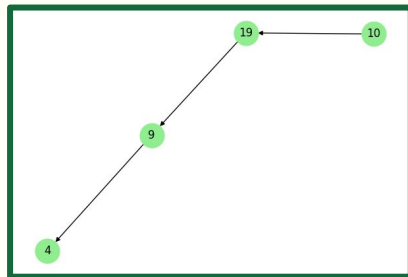


```
pos = nx.spring_layout(G, seed=42)
plt.figure(figsize=(8, 6))
nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=800, edge_color='gray')
plt.title('Graph visualization')
plt.show()

for i, nodes in enumerate(clusters):
    subgraph = G.subgraph(nodes)
    pos = nx.spring_layout(subgraph, seed=42)
    plt.figure(figsize=(8, 6))
    nx.draw(subgraph, pos, with_labels=True, node_color='skyblue', node_size=800, edge_color='black')
    plt.title(f'Cluster {i+1} visualization')
    plt.show()
```









Stage d'été – Semaine 4

05

Output

Pipeline principal

Chaque cluster de nœuds est traité par un circuit QGRNN dédié. La sortie du circuit est une liste de valeurs réelles correspondant aux espérances de mesure PauliZ sur chaque qubit. Ces valeurs permettent d'obtenir une représentation numérique du cluster après propagation des informations entre qubits.

```
def create_qgrnn_cluster_circuit(graph):
    n_qubits = graph.number_of_nodes()
    dev = qml.device("default.qubit", wires=n_qubits)

    @qml.qnode(dev)
    def circuit(inputs, weights):
        node_list = list(graph.nodes())
        index_map = {node: idx for idx, node in enumerate(node_list)}

        for idx, node in enumerate(node_list):
            qml.RY(inputs[idx], wires=idx) # Encodage des features avec

        qgrnn_layer(graph, weights, index_map)

        return [qml.expval(qml.PauliZ(idx)) for idx in range(n_qubits)]

    return circuit, dev
```

```
# 5. PIPELINE PRINCIPAL (exécution comme dans le tutoriel PennyLane) =====

# Paramètres
data_path = "/Users/mohamedsaoudi/Desktop/stage/Implementation/dataset1/FinanceCPT/re
rels_path = "/Users/mohamedsaoudi/Desktop/stage/Implementation/dataset1/FinanceCPT/re
num_vars = 25

# Étape 1 : Charger les données et construire le graphe
df, rels = load_data(data_path, rels_path, num_vars)
start_t = max(rels["lag"])
G_t = build_graph_at_time_t(start_t, df, rels, num_vars)

# Étape 2 : Clustering
clusters = cluster_graph(G_t)

# Étape 3 : Visualisation des graphes
visualize_full_graph(G_t)
visualize_clusters(G_t, clusters)

# Étape 4 : Appliquer QGRNN à chaque cluster
for cluster_id, nodes in enumerate(clusters):
    subgraph = G_t.subgraph(nodes).copy()

    circuit, dev = create_qgrnn_cluster_circuit(subgraph)

    features = np.array([subgraph.nodes[i]['feature'] for i in subgraph.nodes()])
    weights = np.random.randn(len(subgraph.nodes()), requires_grad=True)

    output = circuit(features, weights)
    print(f"Cluster {cluster_id+1} output: {output}")

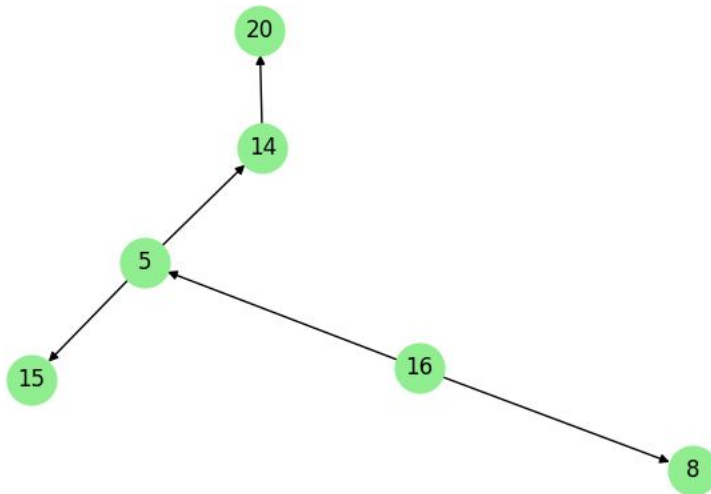
    visualiser_circuit_qgrnn(circuit, features, weights, cluster_id + 1)
```


Pipeline principal

L'output :

```
mohamedsaoudi@MBP-de-mohamed Implementation % /usr/local/bin/python3 /Users/mohamedsaoudi/Desktop/stage/Implementation/second_draft.py
Cluster 1 output: [tensor(-0.81401396, requires_grad=True), tensor(-0.45506842, requires_grad=True), tensor(0.22329344, requires_grad=True), tensor(0.19526051, requires_grad=True), tensor(0.86055782, requires_grad=True), tensor(0.4847411, requires_grad=True)]
/Users/mohamedsaoudi/Desktop/stage/Implementation/second_draft.py:88: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
Cluster 2 output: [tensor(0.91019226, requires_grad=True), tensor(0.24375791, requires_grad=True), tensor(0.99685017, requires_grad=True), tensor(-0.80555094, requires_grad=True), tensor(-0.33002715, requires_grad=True), tensor(-0.13555407, requires_grad=True)]
/Users/mohamedsaoudi/Desktop/stage/Implementation/second_draft.py:88: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
Cluster 3 output: [tensor(0.91019226, requires_grad=True), tensor(0.24375791, requires_grad=True), tensor(0.99685017, requires_grad=True), tensor(-0.80555094, requires_grad=True), tensor(-0.33002715, requires_grad=True), tensor(-0.13555407, requires_grad=True)]
/Users/mohamedsaoudi/Desktop/stage/Implementation/second_draft.py:88: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
Cluster 4 output: [tensor(0.91019226, requires_grad=True), tensor(0.24375791, requires_grad=True), tensor(0.99685017, requires_grad=True), tensor(-0.80555094, requires_grad=True), tensor(-0.33002715, requires_grad=True), tensor(-0.13555407, requires_grad=True)]
/Users/mohamedsaoudi/Desktop/stage/Implementation/second_draft.py:88: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
Cluster 5 output: [tensor(-0.80793011, requires_grad=True), tensor(0.91019226, requires_grad=True), tensor(0.24375791, requires_grad=True), tensor(0.99685017, requires_grad=True), tensor(-0.80555094, requires_grad=True), tensor(-0.33002715, requires_grad=True)]
/Users/mohamedsaoudi/Desktop/stage/Implementation/second_draft.py:88: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
Cluster 6 output: [tensor(0.98869483, requires_grad=True), tensor(0.91019226, requires_grad=True), tensor(0.24375791, requires_grad=True), tensor(0.99685017, requires_grad=True), tensor(-0.80555094, requires_grad=True), tensor(-0.33002715, requires_grad=True)]
/Users/mohamedsaoudi/Desktop/stage/Implementation/second_draft.py:88: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
Cluster 7 output: [tensor(0.92005776, requires_grad=True), tensor(0.91019226, requires_grad=True), tensor(0.24375791, requires_grad=True), tensor(0.99685017, requires_grad=True), tensor(-0.80555094, requires_grad=True), tensor(-0.33002715, requires_grad=True)]
/Users/mohamedsaoudi/Desktop/stage/Implementation/second_draft.py:88: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
Cluster 8 output: [tensor(0.94681781, requires_grad=True), tensor(0.91019226, requires_grad=True), tensor(0.24375791, requires_grad=True), tensor(0.99685017, requires_grad=True), tensor(-0.80555094, requires_grad=True), tensor(-0.33002715, requires_grad=True)]
/Users/mohamedsaoudi/Desktop/stage/Implementation/second_draft.py:88: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
Cluster 9 output: [tensor(0.23519855, requires_grad=True), tensor(0.91019226, requires_grad=True), tensor(0.24375791, requires_grad=True), tensor(0.99685017, requires_grad=True), tensor(-0.80555094, requires_grad=True), tensor(-0.33002715, requires_grad=True)]
/Users/mohamedsaoudi/Desktop/stage/Implementation/second_draft.py:88: UserWarning: This figure includes Axes that are not compatible with tight_layout, so results might be incorrect.
  plt.tight_layout()
```

Espérances de
mesures pour 6
qubits!



Pipeline principal

A quoi servent ces espérances de mesures ?

- Ces valeurs reflètent l'**état final du qubit**, donc indirectement le **résultat du traitement de l'information dans le cluster**.
- On peut voir les espérances comme une représentation vectorielle du cluster, qu'on pourrait :
 - Utiliser comme entrée dans un classifieur classique (SVM, réseau de neurones, etc.).
 - Comparer entre clusters pour voir lesquels se ressemblent.
 - Réinjecter dans un pipeline ML pour une tâche supervisée ou non-supervisée.
- On peut les optimiser lors de l'entraînement : (e.g: Si je fais de la classification, j'aurais une fonction de coût (loss) qui compare les espérances obtenues aux vraies étiquettes, et j'entraînerai les weights du circuit pour minimiser cette perte)

→ Ces espérances permettent de transformer l'information quantique en un **vecteur réel exploitable classiquement**, soit pour de la classification, de l'analyse de similarité, ou toute autre tâche ML.




06

Questions & Prochaines étapes

Résumé :

Jusqu'à maintenant, j'ai un pipeline qui :

- Charge les données
 - Construit un graphe causal à l'instant t
 - Regroupe les nœuds en clusters (sous-graphes)
 - Applique un circuit QGRNN à chaque cluster
 - Extrait les espérances de mesure comme représentations vectorielles.
 - Donne des visualisations des graphes et des circuits
- 

Questions

- Y a-t-il des cas d'usage concrets ou une implémentation envisagée pour ce projet ? J'essaie de trouver le contexte pour interpréter les sous graphes/clusters
- Quelle tâche d'apprentissage sera plus pertinente à appliquer ?

Prochaines etapes

- Ajouter une tâche d'apprentissage (classification, ou autres)
- Faire l'analyse du temps
- L'optimisation (définir une loss function)
- Interpreter les sous-graphes
- GNN
- Regarder le message dans l'output (petit detail de visualisation)
- Preciser le #shots
- Organiser le code sur différents fichiers et le structurer encore mieux
- Vérifier un peu plus le circuit quantique
- Appliquer sur des différents datasets
- Ajouter les ressources consultées sur mon tracker et la présentation sur github



Merci de votre attention!