Quantum Machine Learnig for cybersecurity, Part 2 — training the model:

Marek Kowalik · Follow

9 min read · Sep 7, 2022

DDoS-type attacks recognition with Hybrid Quantum Neural Network (H-QNN):

Introduction:

Imagine you're trying to access your government's website. You just wanted to print some form or read about law changes, but the site doesn't load. It's possible that the servers are under cyberattack. How can we prevent that scenario? With Machine Learning methods of course. They are irreplaceable in the cybersecurity field by capturing data nuances, imperceptible by humans. In this short series, we'll talk about detecting a DDoS-type attack with a new technology: quantum computing.

Quantum computing offers alternative computing ways by using quantum phenomena. Recently there is a lot of hype around this technology and quite a few Quantum Machine Learning (QML) algorithms appeared. However, finding and training Quantum Machine Learning models that present high level of performance is not an easy task. Finding a QML model that obtains this level of performance on a real dataset is truly outstanding. Let's take a look at one of these models and its implementation in `PennyLane` and `Tensorflow`. We'll train a Hybrid Quantum Neural Network (H-QNN) on a DDoS-type attacks dataset from paper "*Quantum machine learning for intrusion detection of distributed denial of service attacks: a comparative overview*" [1]. The code in this article is a slightly changed version from the repository attached to the paper. In this article we'll train the model on the data prepared in the previous part.

H-QNN: A bird's-eye view:

Current quantum devices allow us to train effectively only small quantum models (using as few gates as possible with quantum states, with only a few qubits); this is due to high error rates. Also, models with that size allow to train them on quantum simulators in a reasonable time. In this blog we'll train them this way.

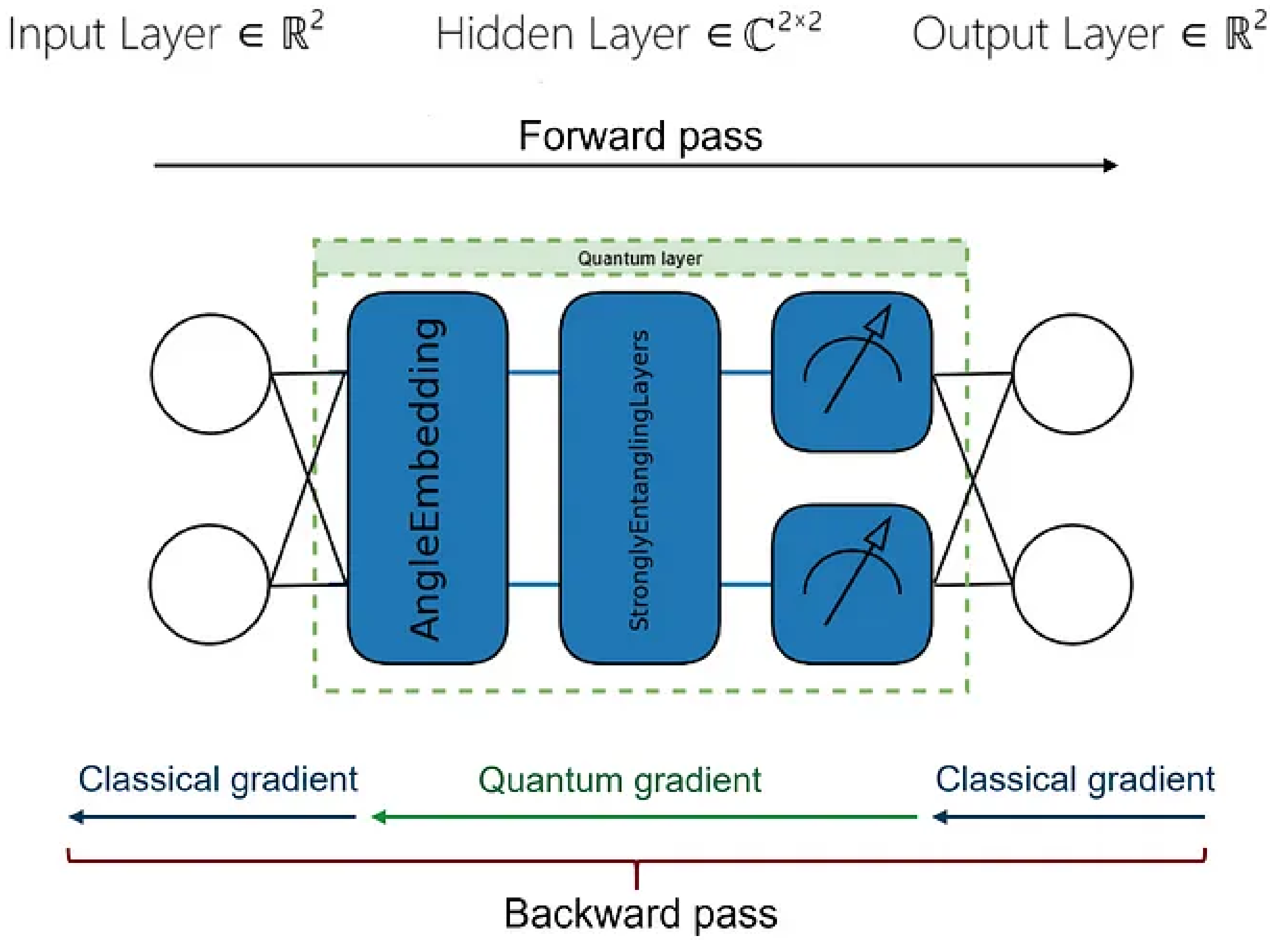
There's a bunch of QML algorithms available, but today we'll use a Hybrid Quantum Neural Network. In the original paper [1] this H-QNN model performed the best in comparison to two other tested models. H-QNNs are classical Neural Networks (NN) with one or more layers substituted with quantum circuits. This way, we can swap for example small, dense layers at the end of NN with quantum circuits with just a few qubits and then simulate them fast and locally or quickly train them on quantum devices (even with thousands of data points).

How can we use a quantum circuit as a quantum layer in a Deep Neural Network? We need to:

1. Encode the input to the quantum state (e.g. Angle Embedding)
2. Use a parametrized set of quantum gates (aka ansatz), to modify the encoded state (e.g. Strongly Entangling Layer)
3. Decode the output from modified quantum state (obtain (quasi)-probabilities of every possible state from measurement and then, for example, calculating expectation values).

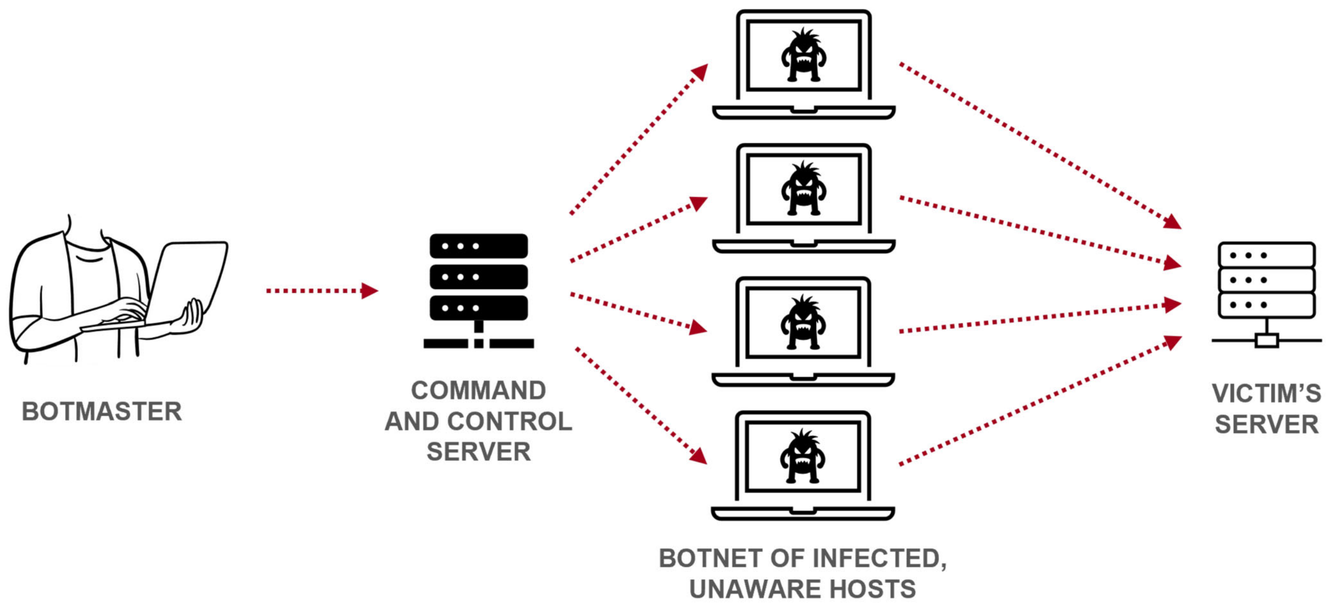
`Qiskit` and `Pennylane` have good, high-level implementations of quantum layers with gradients evaluation. This way we can truly incorporate them into `Torch` or `TensorFlow` Neural Network models.

Here's schematic representation of H-QNN. It's a Deep Neural Network with two nodes per layer, with one hidden layer swapped to quantum one:



DDoS attacks (reminder):

Distributed Denial of Service (**DDoS**) is a type of cyberattack, where many hosts are trying to connect with a victim's server, until it crashes and is unable to process a legitimate request. It's coordinated action from one central point, usually performed with malicious software, which infects devices of unaware owners:



Recognizing this type of attack is important in the early stages of connection to the server. This prevents the attackers from taking resources, swamping, and finally shutting down the website or application. We need small, robust models to quickly classify, for example, a user request as benign or potential DDoS, without slowing down the whole process. As we can see, this fits small H-QNN, which can be easily simulated locally, without sending it to a quantum device for prediction. `Qiskit`, `PennyLane` and other Python software development kits provide efficient quantum simulators for classical computers.

Further reading: [What is a Distributed Denial-of-Service Attack?]

Let's begin with importing all needed packages. Here's our stack:

Data preprocessing	Quantum Part of the H-QNN	Classical Part of the H-QNN	Evaluation	Visualization
pandas NumPy	PENNYLANE	TensorFlow	scikit learn	matplotlib seaborn

```
# QML
import pennylane as qml

# ML
```

```
import tensorflow as tf
from tensorflow.keras.callbacks import ReduceLROnPlateau

# Data processing:
import numpy as np

# Utils
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, recall_score,
confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import f1_score, accuracy_score, precision_score,
make_scorer

# Visuals:
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
```

Next, let’s set up global variables:

```
# Set random seed:
GLOBAL_SEED = 419514
np.random.seed(GLOBAL_SEED)
tf.random.set_seed(GLOBAL_SEED)
# Computation data format:
comp_dtype = 'float32'
```

Training data preparation:

Train/test/validation split:

As a train and test dataset, we took the chosen `10'000` samples (from the previous part). Then, we split them in ratio `70%` - `15%` - `15%` (training - validation - testing, respectively).

```
# set ratio:
train_val_test_split = [0.7, 0.15, 0.15]
test_val_size = sum(train_val_test_split[1:])
val_size = train_val_test_split[2]/test_val_size

# split dataset:
trainX, testX, trainy, testy = train_test_split(x, y, stratify=y,
test_size=test_val_size, random_state=GLOBAL_SEED)
testX, valx, testy, valy = train_test_split(testX, testy,
stratify=testy, test_size=val_size, random_state=GLOBAL_SEED)
```

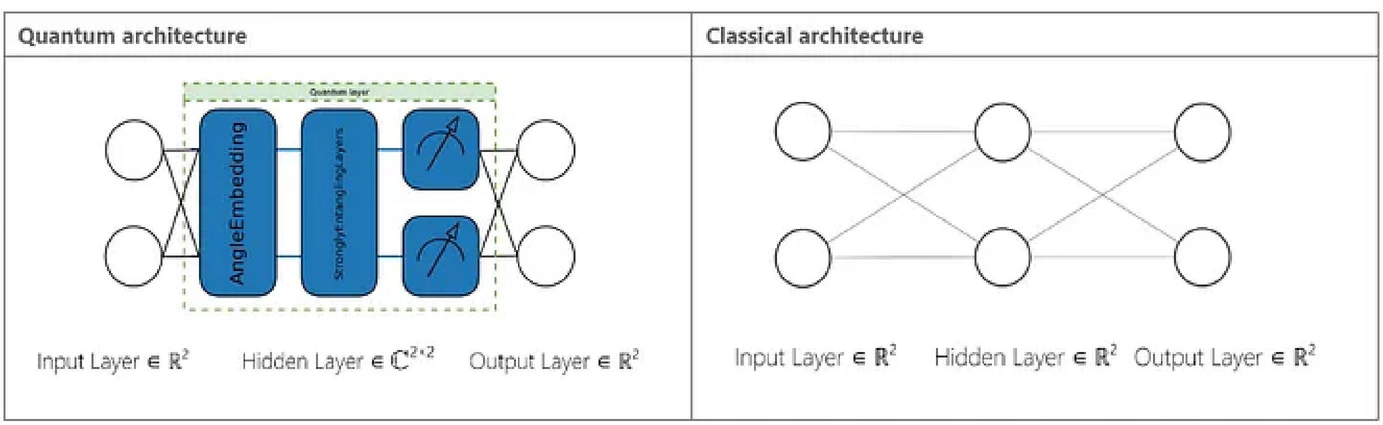
```
# One-hot encode:
trainy = tf.one_hot(trainy, depth=n_features, dtype=comp_dtype)
testy = tf.one_hot(testy, depth=n_features, dtype=comp_dtype)
valy = tf.one_hot(valy, depth=n_features, dtype=comp_dtype)
```

H-QNN and DNN models training:

Model:

As mentioned before, we'll use Hybrid Quantum Neural Network (H-QNN) architecture with fully connected layers as an input and output. The hidden layer is a quantum layer implemented in `PennyLane`. It's a parametrized quantum circuit, but then we convert it to a trainable `Keras` layer. To find out more about `Tensorflow-PennyLane` integration; see this tutorial.
Quantum layer performs Angle Embedding to encode the data to a quantum state. Then we add Strongly Entangling Layers, since it's commonly used in QML e.g. [2].
For comparison, as a classical model we used Deep Neural Network (DNN) architecture with one hidden layer. The input, hidden and output layers have inputs and outputs of shape `2`. The 2-qubits quantum layer in H-QNN encode the data into 4 complex numbers and transform them this way. That allows quantum neural networks for higher expressivity (in a sense of approximation broader classes of functions).

Both architectures schematically:



Let's specify our training parameters:

```
n_qubits = 2
layers = 1
```

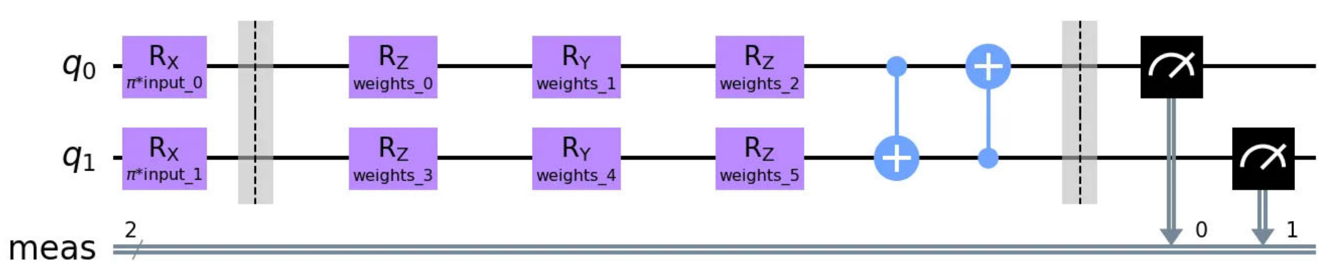
```
data_dimension = n_features

batch_size = 5
n_epochs = 1
```

Quantum layer

Pennylane has a bunch of high-level wrappers for constructing quantum circuits for Quantum Neural Networks. We'll use them to make `qnode` (Pennylane quantum circuit object) used later as our Quantum Layer in H-QNN. We need to implement:

- **Classical data encoding:** For that, we'll use `AngleEmbedding`, which adds rotation gates for each qubit. Default gate is rotation around X-axis and we'll stick with that. For this part, we need to pass two input parameters, to specify rotation angle for the gate. We'll scale the input by `π` to cover all possible qubit states within range `(-1, +1)` which is the output from the previous layer with sigmoid activation function.
- **Trainable ansatz:** For that, we'll use `StronglyEntanglingLayers`, which apply 3 rotations around `z`, `y` and `z` axis respectively. At the end, we have CNOT gates in both possible ways on 2 qubits. We can apply them as many times as we want, but will stick with only one set.
- **Data decoding:** For that, we'll use expectation values in Z basis. This is the most common method, which is just the sum of probabilities of measuring, given qubit in a state `|1>` with coefficient `-1` and in a state `|0>` with coefficient `1`. We'll get two numbers between the range `(-1, +1)` as an output.

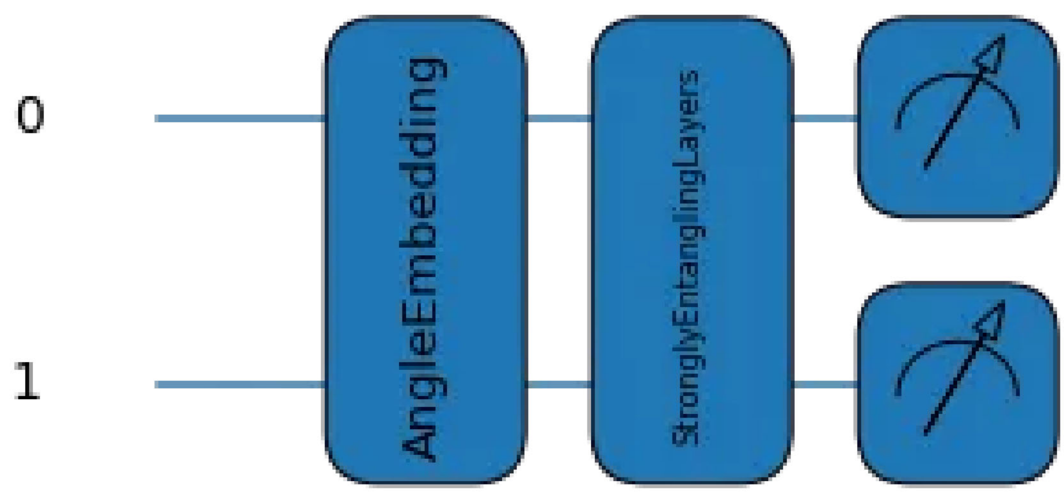
Our quantum layer schematically will looks like this:



Let’s code it:

```
dev = qml.device("default.qubit", wires=n_qubits)

@qml.qnode(dev)
def qnode(inputs, weights):
    qml.templates.AngleEmbedding(np.pi*inputs, wires=range(n_qubits))
    qml.templates.StronglyEntanglingLayers(weights, 
wires=range(n_qubits))

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


weight_shapes = {"weights": (layers,n_qubits,3)}

inputs = np.random.rand(n_qubits).astype(comp_dtype)
weights = np.random.rand(layers, n_qubits, 3).astype(comp_dtype)

plt.figure(figsize=(10,10))
qml.draw_mpl(qnode)(inputs, weights)
plt.show()
```

```
<Figure size 720x720 with 0 Axes>
```



To complete the model, we'll use two dense layers as input and output layers. The first one will have `sigmoid` activation function and the second one `softmax`, since we have binary classification with two numbers as output from our model. To make our `qnode` proper `Keras` layer, we need to put it into `qml.qnn.KerasLayer()` wrapper:

```
q_layer = qml.qnn.KerasLayer(qnode, weight_shapes,
output_dim=n_qubits, dtype=comp_dtype)
q_layer.build(2)

q_model = tf.keras.models.Sequential()
q_model.add(tf.keras.layers.Dense(n_qubits, activation='sigmoid',
input_dim=data_dimension))
q_model.add(q_layer)
q_model.add(tf.keras.layers.Dense(data_dimension,
activation='softmax'))

q_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 2)                 6
_________________________________________________________________
keras_layer (KerasLayer)     (None, 2)                 6
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 6
=================================================================
Total params: 18
Trainable params: 18
Non-trainable params: 0
_________________________________________________________________
```

H-QNN training:

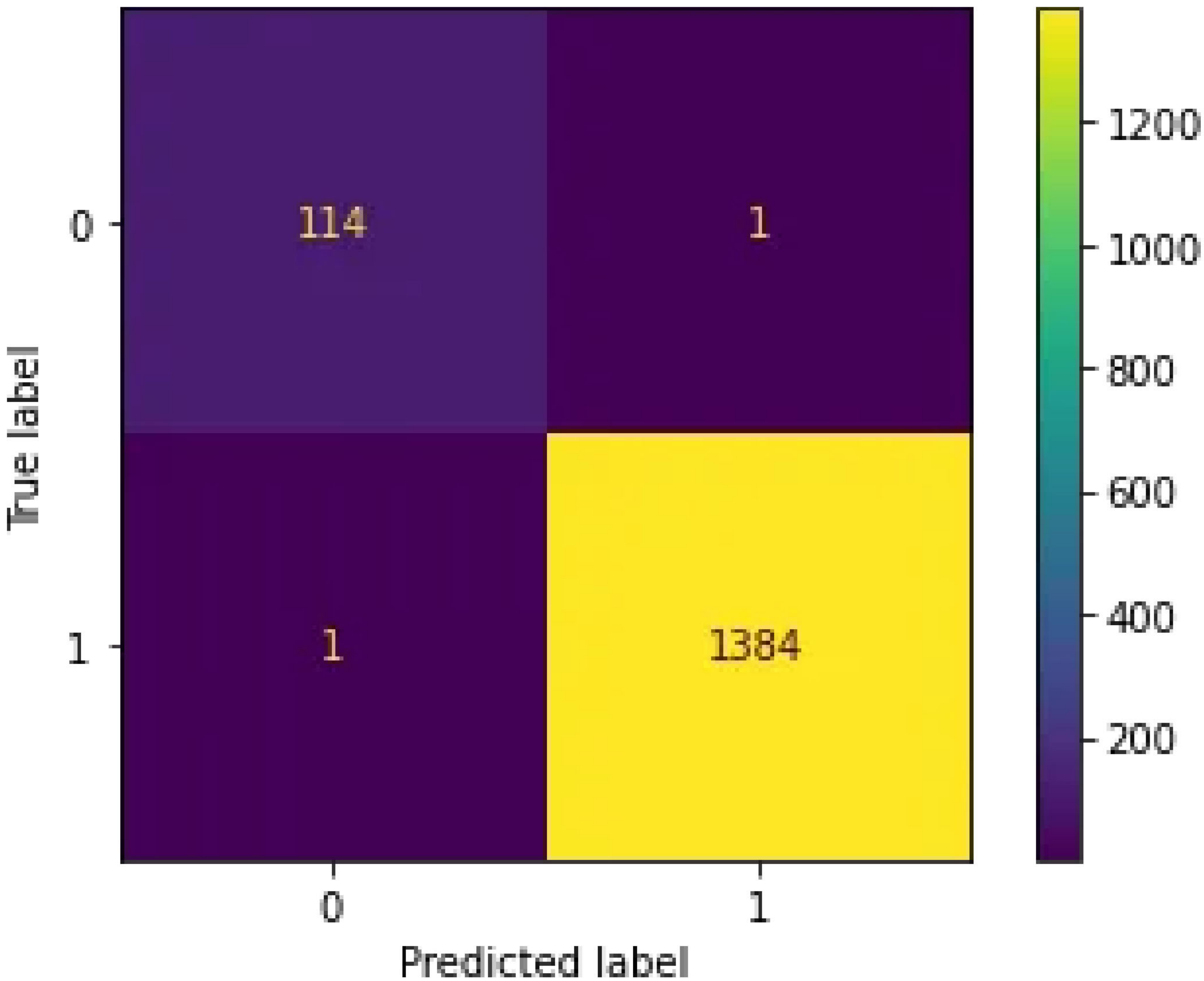
Finally, we are ready to run the training:

```
%%time

opt = tf.keras.optimizers.Adam(learning_rate=0.05)
q_model.compile(loss='categorical_crossentropy',
optimizer=opt,metrics=["accuracy"])

q_history = q_model.fit(trainX, trainy, validation_data=(testX,
testy), epochs=n_epochs, batch_size=batch_size)
```

```
1400/1400 [==============================] - 279s 199ms/step - loss:
0.2310 - accuracy: 0.9268 - val_loss: 0.0110 - val_accuracy: 0.9993
Wall time: 4min 38s
```

As we can see, our model trained in just one epoch. Let's evaluate it and see the results in a confusion matrix:

```
# predicition:
valpredy = q_model.predict(valx)
valpredy_round = np.round(valpredy)

# metrics calculation:
q_classification = classification_report(valy[:,1],
valpredy_round[:,1])
q_confusion = confusion_matrix(valy[:,1], valpredy_round[:,1])

q_accuracy = round(accuracy_score(valy[:,1],
valpredy_round[:,1])*100,5)
q_recall = round(recall_score(valy[:,1], valpredy_round[:,1],
average='macro')*100,5)
q_precision = round(precision_score(valy[:,1], valpredy_round[:,1],
average='weighted')*100,5)
q_f1 = round(f1_score(valy[:,1], valpredy_round[:,1],
average='weighted')*100,5)

print(f'Accuracy:\t {q_accuracy:.2f}%')
print(f'Recall:\t\t {q_recall:.2f}%')c
print(f'Precision:\t {q_precision:.2f}%')
print(f'F1:\t\t {q_f1:.2f}%')

Accuracy:        99.87%
Recall:          99.53%
Precision:       99.87%
F1:              99.87%

disp = ConfusionMatrixDisplay(confusion_matrix=q_confusion)
disp.plot()
plt.show()
```



Classical analog:

This shows that H-QNN can be very effective in a simple classification. For comparison, we'll train a classical DNN model. As activation, we'll use relu instead of sigmoid, but the layers shape stays the same:

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(n_qubits, activation='relu',
input_dim=data_dimension))
model.add(tf.keras.layers.Dense(data_dimension, activation='relu'))
model.add(tf.keras.layers.Dense(data_dimension, activation='softmax'))

model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 2)                 6
_________________________________________________________________
dense_9 (Dense)              (None, 2)                 6
_________________________________________________________________
dense_10 (Dense)             (None, 2)                 6
=================================================================
Total params: 18
Trainable params: 18
Non-trainable params: 0
_________________________________________________________________
```

Training:

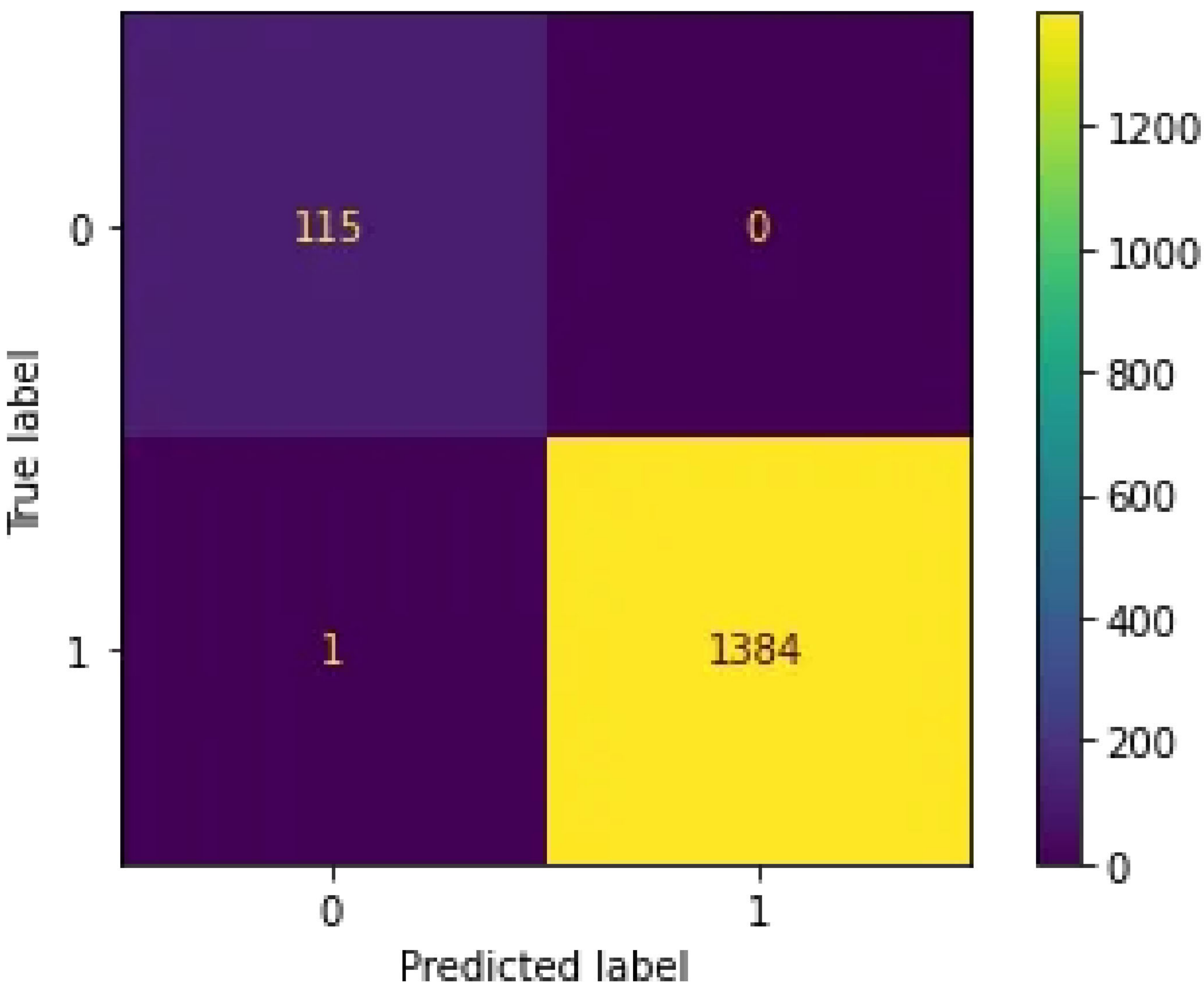
```
%%time

opt = tf.keras.optimizers.Adam(learning_rate=0.02)
model.compile(loss='categorical_crossentropy', optimizer=opt,metrics=
["accuracy"])

history = model.fit(trainX, trainy, validation_data=(testX, testy),
epochs=n_epochs, batch_size=batch_size)
```

```
1400/1400 [==============================] - 2s 1ms/step - loss:
0.0625 - accuracy: 0.9804 - val_loss: 0.0030 - val_accuracy: 0.9993
Wall time: 2.41 s
```

```
# prediction:
valpredy = model.predict(valx)
valpredy_round = np.round(valpredy)

# metrics calculation:
classification = classification_report(valy[:,1], valpredy_round[:,1])
confusion = confusion_matrix(valy[:,1], valpredy_round[:,1])

accuracy = round(accuracy_score(valy[:,1], valpredy_round[:,1])*100,5)
recall = round(recall_score(valy[:,1], valpredy_round[:,1],
average='macro')*100,5)
precision = round(precision_score(valy[:,1], valpredy_round[:,1],
average='weighted')*100,5)
f1 = round(f1_score(valy[:,1], valpredy_round[:,1],
average='weighted')*100,5)

print(f'Accuracy:\t {accuracy:.2f}%')
print(f'Recall:\t\t {recall:.2f}%')
print(f'Precision:\t {precision:.2f}%')
print(f'F1:\t\t {f1:.2f}%')
```

```
Accuracy:        99.93%
Recall:          99.96%
Precision:       99.93%
F1:              99.93%
```

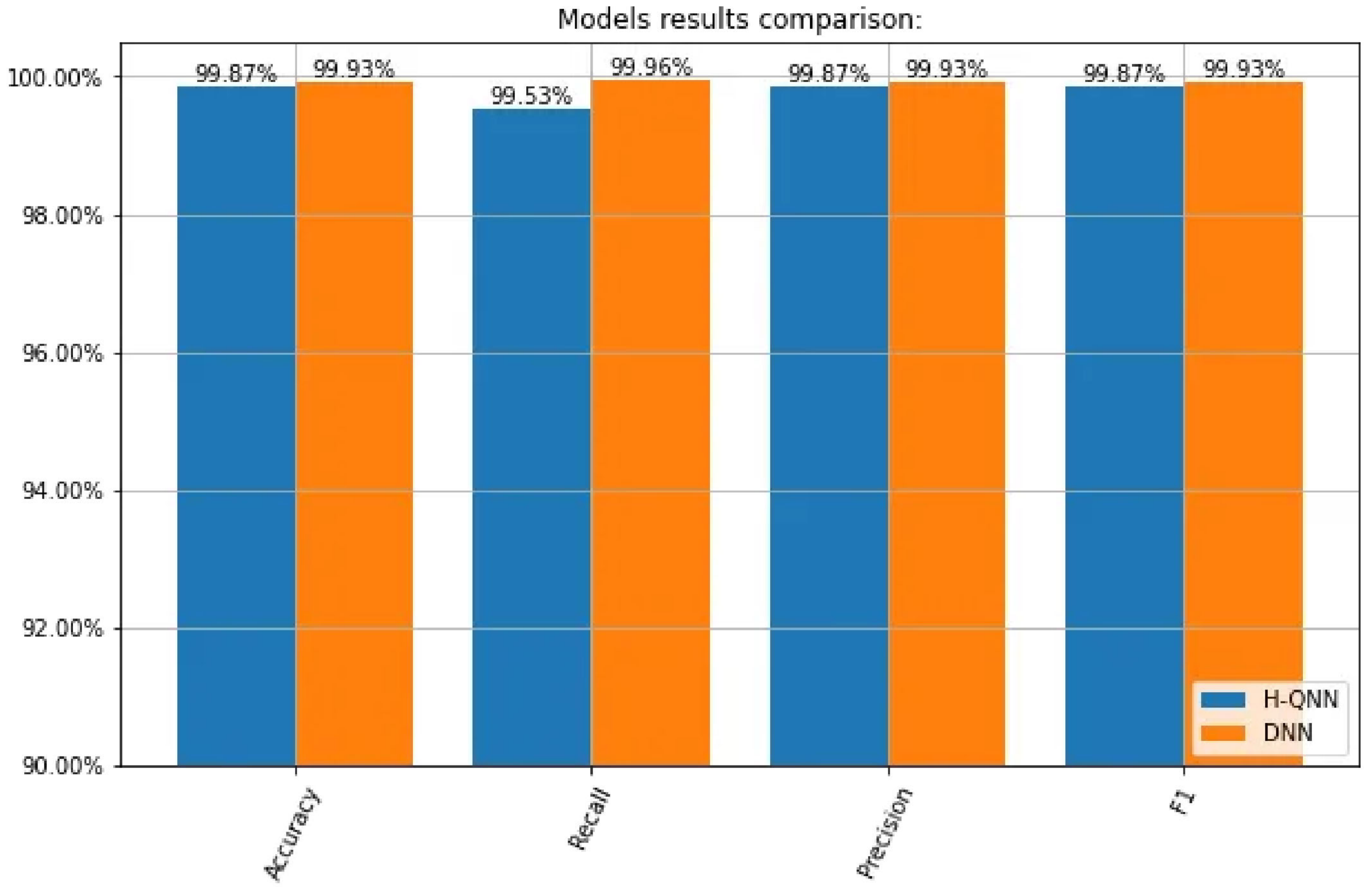
```
# plotting confusion matrix:
disp = ConfusionMatrixDisplay(confusion_matrix=confusion)
disp.plot()
plt.show()
```



As we can see, the results are similar. Let’s put all the metrics on the same plot:

```
c_results = np.asarray([accuracy, recall, precision, f1])
q_results = np.asarray([q_accuracy, q_recall, q_precision, q_f1])
results_description = ['Accuracy', 'Recall', 'Precision', 'F1']
```

```
# plot preparation:
fig, ax = plt.subplots(figsize = (10,6))
idx = np.asarray([i for i in range(4)])
width = 0.4

# plotting:
q_bars = ax.bar(idx-width/2, q_results, width=width, label='H-QNN')
c_bars = ax.bar(idx+width/2, c_results, width=width, label='DNN')

# setting ticks:
ax.set_xticks(idx)
ax.set_title('Models results comparison:')
ax.set_xticklabels(results_description, rotation=65)
fmt = '%.2f%%'
yticks = mtick.FormatStrFormatter(fmt)
ax.yaxis.set_major_formatter(yticks)

# add bars labels:
ax.bar_label(c_bars, fmt=fmt)
ax.bar_label(q_bars, fmt=fmt)

# set y-axis limits
ax.set_ylim(90, 100.5)

ax.legend(loc=4)
plt.grid()

plt.show()
```



Summary:

In this series, we have trained and evaluated Hybrid Quantum Neural Network (H-QNN). We adapted one of the state-of-the-art approaches, feeding a robust, small quantum architecture with heavily reduced data in terms of features. This allows us to train and evaluate the model using a lot more data points, since quantum models inference is a major bottleneck of these architectures. We obtained a high level of model performance, comparable with classical analog. The code is ready to run on real quantum devices, since the quantum layer is very shallow, and there's no need to use error correction.

References:

[1] Quantum machine learning for intrusion detection of distributed denial of service attacks: a comparative overview, *E. D. Payares and J. C. Martinez-Santos, 2021*

[2] *QDNN: deep neural networks with quantum layers*, Zhao, Gao 2021 https://doi.org/10.1007/s42484-021-00046-w

Quantum Computing Machine Learning Cybersecurity Neural Networks

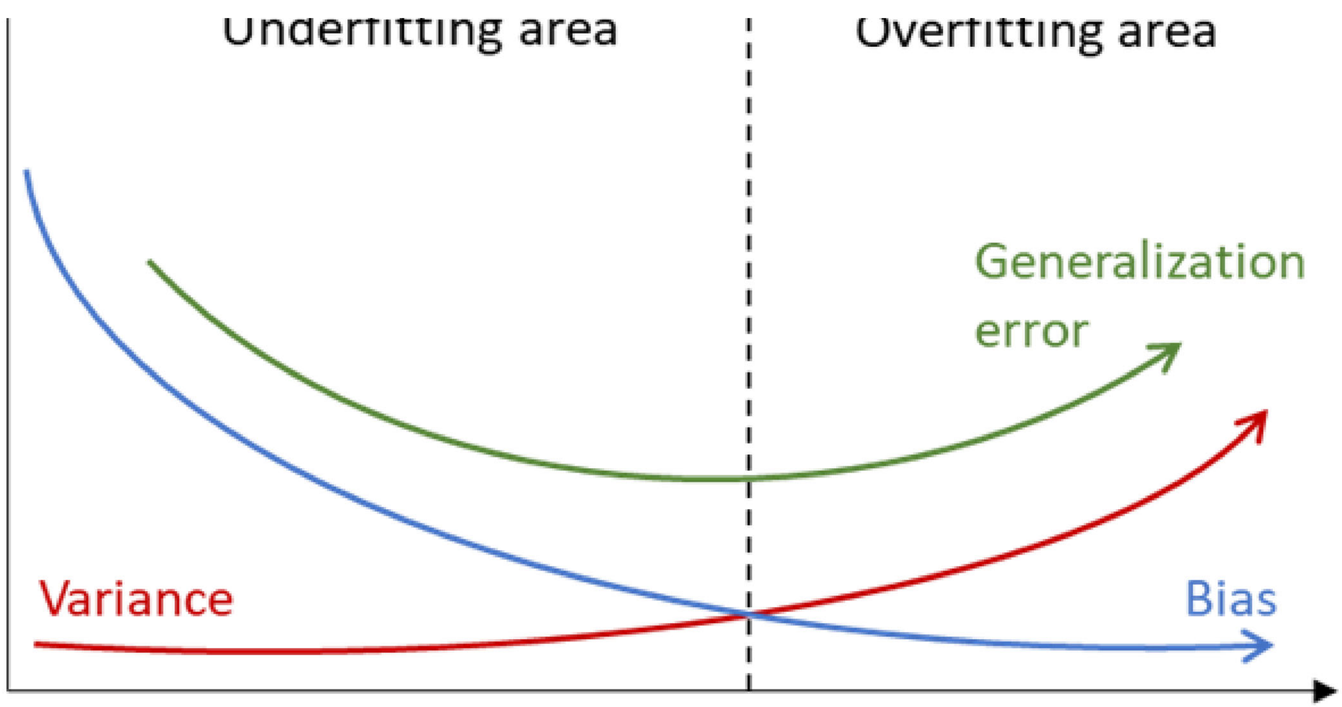
Follow

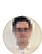
Written by Marek Kowalik

17 Followers

Junior Data Scientist and Quantum Developer in Capgemini Quantum Lab.

More from Marek Kowalik



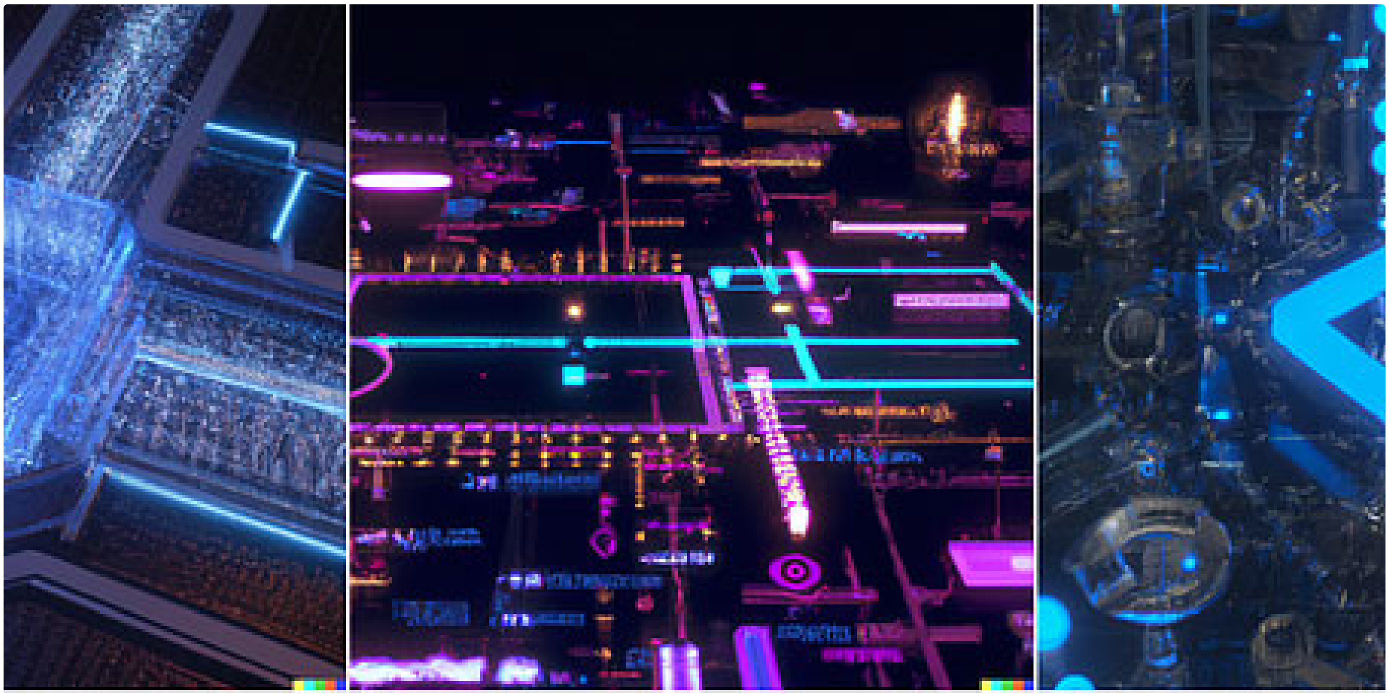


Marek Kowalik

Capacities of Quantum Neural Networks, Part 1

On a way to compare quantum and classical machine learning models

7 min read · 3 days ago

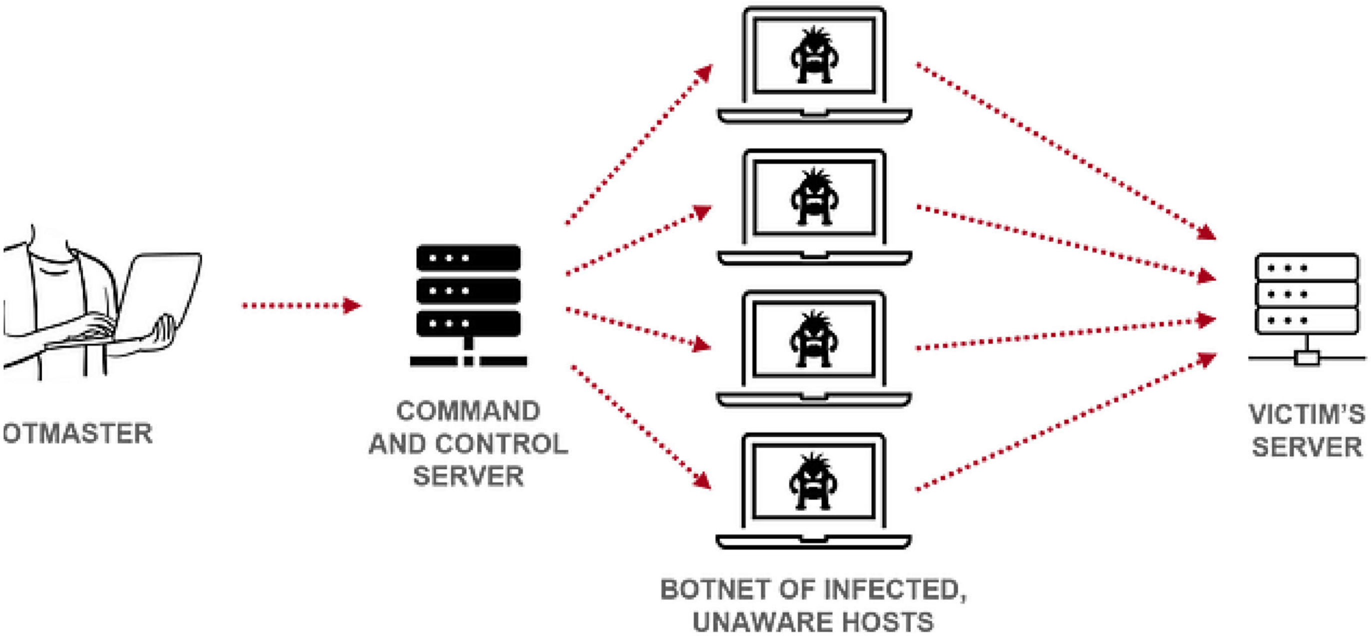


Marek Kowalik

Parallelization for Quantum Computers — multi-programming

Parallelization is a game changer when it comes to the optimization of programs; particularly for training deep learning models, it has...

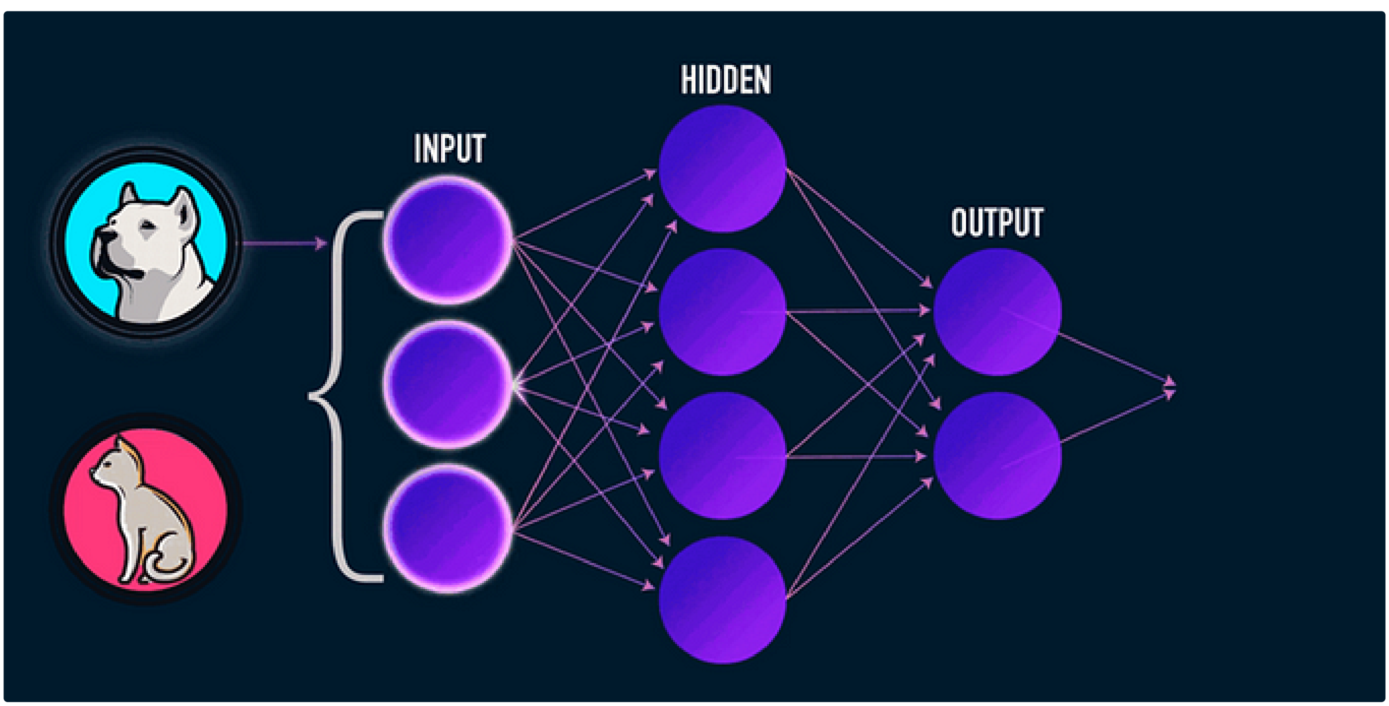
7 min read · Oct 12, 2022



Marek Kowalik

Quantum Machine Learning for Cybersecurity, Part 1

7 min read · Sep 7, 2022



Marek Kowalik

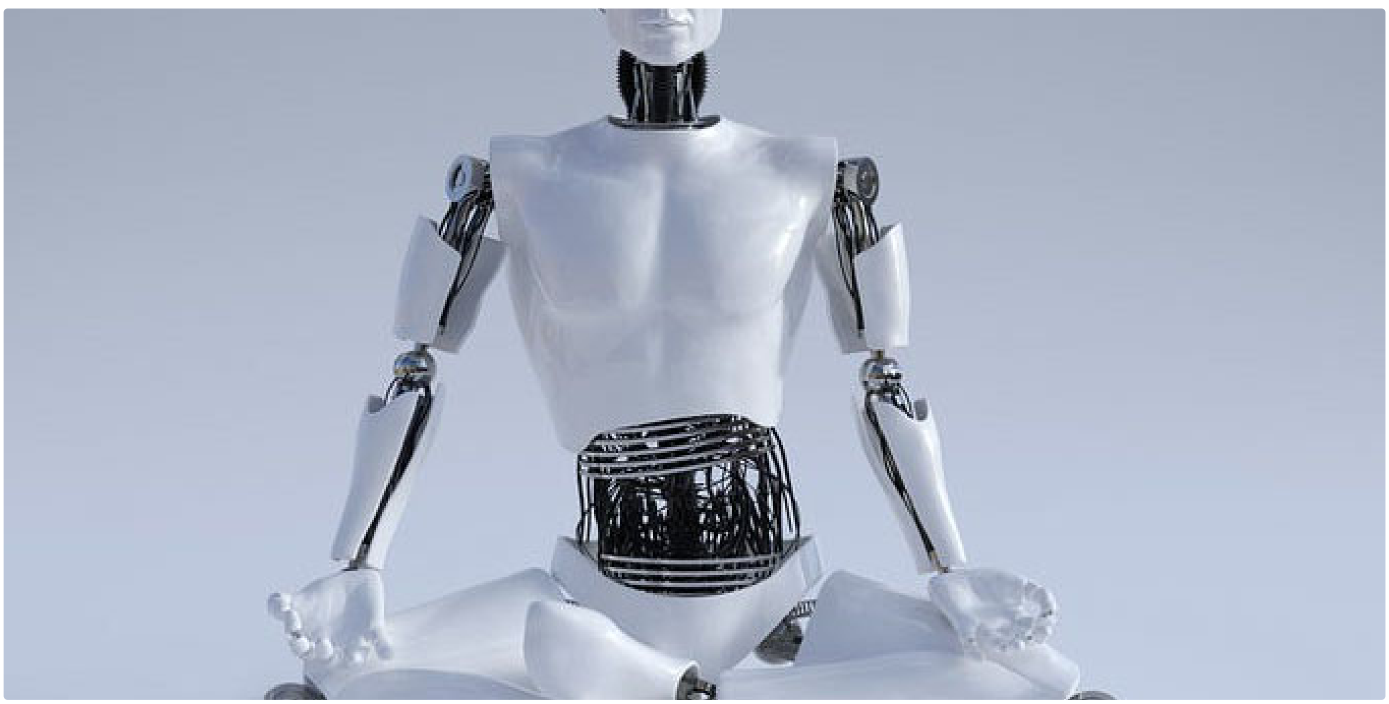
Capacities of Quantum Neural Networks, Part 2

On a way to compare quantum and classical machine learning models

8 min read · 3 days ago

See all from Marek Kowalik

Recommended from Medium



The PyCoach in Artificial Corner

You're Using ChatGPT Wrong! Here's How to Be Ahead of 99% of ChatGPT Users

Master ChatGPT by learning prompt engineering.

· 7 min read · Mar 17

17.9K 326

Tim Lou, PhD in Towards Data Science

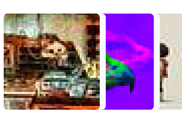
How Quantum Physics Broke the Laws of Statistics

Demystifying the Data Science Behind 2022's Physics Nobel Prize

· 12 min read · May 3

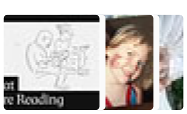
411 5

Lists



What is ChatGPT?

9 stories · 21 saves



Staff Picks

300 stories · 59 saves

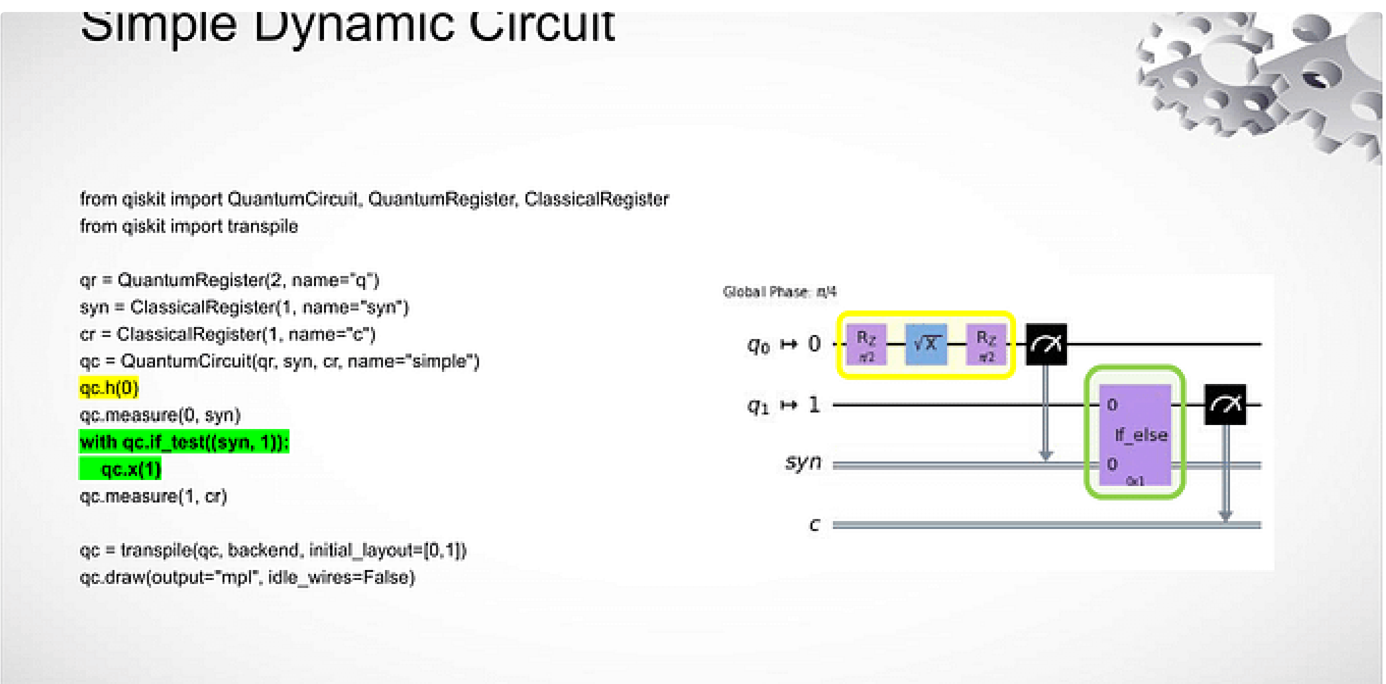
Dennis Ganzaroli in MLearning.ai

The Best kept Secret in Data Science is KNIME

Discover KNIME, the best kept secret in data science. This powerful and versatile open source platform offers a visual interface and wide...

· 14 min read · Feb 2

367 3



Brian N. Siegelwax in Level Up Coding

Dynamic Quantum Circuits, Lesson 1

FYI: code snippets won't work.

· 2 min read · Dec 5, 2022

54

Alexander Nguyen in Level Up Coding

Why I Keep Failing Candidates During Google Interviews…

They don't meet the bar.

· 4 min read · Apr 13

3.5K 108

Salvatore Raieli in Level Up Coding

Welcome Back 80s: Transformers Could Be Blown Away by Convolution

The Hyena model shows how convolution could be faster than self-attention

· 10 min read · May 2

1K 11

See more recommendations