TP Apprentissage profond sur système embarqué (Xilinx ARTYX)

INP- ENSEIRB-MATMECA – Option SE –

Objectif

L'objectif est d'essayer d'optimiser un réseau de neurones profonds pour l'exécuter dans un circuit ARTYX 7 sur la carte NEXYS A7-100T. La référence du composant est **xc7a100tcsg324-1**.

La première étape est de créer un modèle de réseau de neurones non optimisé qui servira de référence pour comparaison avec le modèle optimisé.

La seconde étape est de réaliser l'élagage et la quantification des coefficients du réseau. Nous synthétiserons alors un IP en VHDL prêt à être utilisé dans un système numérique. Pour cela, nous étudierons deux façons de réaliser ces optimisations.

Pour cela, vous avez l'ensemble des codes des démonstrations réalisées en cours pour construire les scripts pythons nécessaires.

Les fichiers DNN_model_1.py, DNN_model_2.py et DNN_model_3.py vous permettent d'écrire vos codes et de les exécuter sous python et l'éditeur **« spyder ».**

Vous devez rendre les trois fichiers scripts et un **rapport complet** de l'analyse de l'étude avec les ***succés, les echecs montrant votre comprehension de cette technologie.***

L'environnement

Vous allez travailler sur une machine virtuel Linux Ubuntu. Suivez les directives ci-dessous pour gérer votre projet de TP.
Pour accéder à la VM, vous devez vous connecter en choisissant via l'icône en haut à droite en forme de clé à molette la VM FPGA_IA. Il faut rentrer son nom, son mot de passe puis choisir sa VM avant de cliquer sur login (ou appuyer sur Enter).
Une fois la VM FPGA_IA opérationnelle, il y a 2 comptes se01 pour le groupe 1 et se02 pour le groupe 2. Les mots de passe sont (username/passwd) :
se01/se01 et **se02/se02**
Une fois connecté, vous lancez dans un terminal la commande :
$ **mbsdk**
pour avoir l'environnement Xilinx. Le prompt est changé en **[Xilinx EDK 2019.2]$**

⇨ copier les étapes du TP :

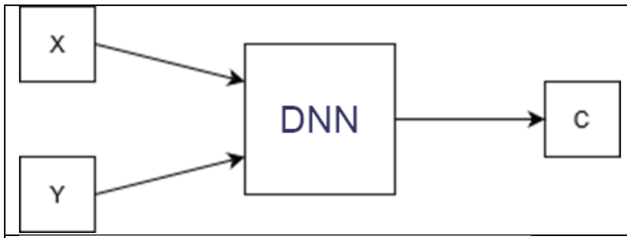
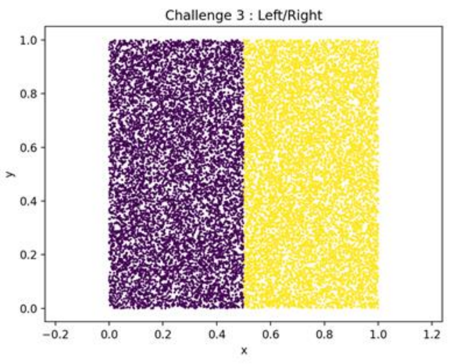
```
mkdir tp
cd tp
cp /home/druillole/TP_HLS4ML.zip .
unzip TP_HLS4ML.zip
```

L''environnement Anaconda est accessible dans le terminal.
Pour lancer spyder, vous aurez juste à taper :

[Xilinx EDK 2019.2]$ spyder

Le réseau de neurones

Nous allons utiliser un réseau de neurones de test qui n'a pas d'intérêt sauf celui d'apprendre la méthode d'optimisation d'un réseau de neurone pour système embarqué. L'objectif est d'apprendre les étapes pour descendre un réseau de neurones dans un SoC. Cela permet d'avoir une compréhension du réseau sans se poser de question sur la nature des données utilisées.

	L'objectif du réseau est de classer les coordonnées (X, Y) selon la région de provenance selon une catégorie binaire c=0 ou c=1. Les deux régions séparent un plan en deux selon un carré de (0,1) par (0,1).
	Ci-contre est la représentation des régions du plan.

Dix mille données labellisées sont générées automatiquement suivant le code suivant :

```
##############################################
# Generation Des Données                 #
##############################################
coords=[]
color=[]
for i in range(10000):
    x=np.random.uniform(low=0.,high=0.5)
    y=np.random.uniform(low=0.,high=1.)
    coords.append([x,y])
    color.append(0)
    x=np.random.uniform(low=0.5,high=1.)
    y=np.random.uniform(low=0.,high=1.)
    coords.append([x,y])
    color.append(1)

save_challenge(normalize_coords(coords),color,"ch3")
plot_challenge(normalize_coords(coords),color,"Challenge 3 : Left/Right")
```

Le réseau est décrit par 4 couches profondes selon la séquence : 2x200x100x50x1. La commande en tensorflow/Keras pour une couche est :

Keras_model = Sequential()
Keras_model.add(Input(shape=(***nbre_variables_entree***,)))

Keras_model.add(Dense(***nbre_neurones***, name= ''fc1'', kernel_initializer=''lecun_uniform''))
Keras_model.add(Activation(*activation.relu*, name=''relu1''))

La dernier fonction d'activation pour sélectionner la région est la fonction ''sigmoid'' : activation.sigmoid

Modèle 1

Utiliser le fichier **DNN_model_1.py** dans le répertoire **TP_DNN_MODEL_1** pour :

- Construire le réseau de neurones
- Réaliser l'apprentissage profond en créant un jeu de données d'entrainement et un jeu de données de test.
- Réaliser le profil des coefficients grâce à la bibliothèque HLS4ML
- Changer la configuration du modèle HLS pour tenir compte du profile des coefficients.
- Convertir le modèle en projet HLS
- Implémenter et synthétiser le modèle.

Rédiger un rapport donnant le nombre total de coefficients du réseau, la precision des réponses prévues avant et après conversion du modèle en HLS, le profilage du modèle et l'empreinte des ressources sur le composant Zynq.

Décrire l'ensemble des répertoires et leur contenu créés lors de l'exécution du script.

Modèle 2

L'objectif est maintenant d'optimiser le modèle pour réduire l'empreinte matériel dans le circuit Zynq en essayant de conserver les performances. Pour cela, nous utilisons deux outils :

-l'élagage : mettre à zéro l'ensemble des coefficients proche de zéro, réduisant le nombre de calcul à effectuer.

-la quantification : réduire la taille des coefficients en virgule fixe, réduisant le nombre de circuits logiques du composant.

Utiliser le fichier DNN_model_2.py pour :

Convertir le modèle défini au script DNN_MODEL_1.py en utilisant QKeras dans le répertoire **TP_DNN_MODEL_2**. Le fichier de script s'appelle **DNN_MODEL_2.py**:

qkeras_model.add(Input(shape=(2,)))

qkeras_model.add(QDense(***nbre_neurones***, name="fc1",
                kernel_quantizer=quantized_bits(**16,2**,alpha=1), bias_quantizer=quantized_bits(**16,2**,alpha=1),
                kernel_initializer="lecun_uniform", kernel_regularizer=l1(0.0001)))

qkeras_model.add(QActivation(activation=quantized_relu(9,2), name="relu1"))

Récupérer dans les fichiers annexes les commandes pour réaliser l'élagage du modèle.

- Construisez le réseau de neurones
- Réaliser l'apprentissage profond en créant un jeu de données d'entrainement et un jeu de données de test.
- Réaliser l'élagage.
- Convertir le modèle en projet HLS en configurant le réseau suivant le modèle QKeras
- Implémenter et synthétiser le modèle.

Rédiger un rapport donnant le nombre total de coefficients du réseau, la précision des réponses prévues avant et après conversion du modèle en HLS, le profilage du modèle et l'empreinte des ressources sur le composant Zynq.

Modèle 3

Optimiser le modèle sans utiliser QKeras mais en forçant la configuration du modèle HLS comme vu en cours avec HLS4ML. Ajouter l'élagage des coefficients comme pour le modèle 2. Vous créerez un fichier DNN_MODELE_3.py à partir du fichier DNN_MODEL_1.py et DNN_MODEL_2.py.

Créer le fichier **DNN_model_3.py** dans le répertoire **TP_DNN_MODEL_1** pour :

- Construire le réseau de neurones
- Réaliser l'apprentissage profond en créant un jeu de données d'entrainement et un jeu de données de test.
- Réaliser le profil des coefficients grâce à la bibliothèque HLS4ML
- Changer la configuration du modèle HLS pour tenir compte du profile des coefficients.
- Convertir le modèle en projet HLS
- Implémenter et synthétiser le modèle.

Rédiger un rapport donnant le nombre total de coefficients du réseau, la precision des réponses prévues avant et après conversion du modèle en HLS, le profilage du modèle et l'empreinte des ressources sur le composant Zynq.

Amélioration de l'IP VIVADO

Après implémentation en HLS, que constatez-vous sur les ressources utilisées par le SoC ? Est-ce normal ?

Après avoir identifié où se trouve le modèle HLS généré par HLS4ML, analyser le code HLS généré, expliquer la présence et l'utilité des différents #pragma.

Que constatez-vous sur les ressources utilisées par le SoC pour le modèle ? Trouver une solution à l'éventuel problème constaté. Concluez.

Conclusion

Suivant les différentes synthèses faites, conclure sur la possibilité ou non de l'intégration du réseau dans le circuit ARTYX. Quel besoin en termes de ressource a-t-on besoin ?

ANNEXES

Installation des outils

La conversion du code python en VHDL utilise l'outil HLS4ML qui ne fonctionne que sur Linux. L'ensemble des plateformes d'intégration de réseaux de neurones est compatible Linux car elles sont liées aux outils de compilation C++ comme pour les outils HLS de Xilinx. Pour cette raison, on peut soit utiliser une machine virtuelle Linux sous Windows soit utiliser une machine PC sous OS Linux. Nous avons validé l'installation sous Centos8.

On installe les logiciels suivants sur la base Linux :

- GCC / C++ (**yum install gcc-c++**)
- Graphviz (**yum install graphviz**)
- VIVADO HLS (6h d'installation) version 2019.2 + les composants et cartes pour Zynq, ultrascale ….
- ANACONDA (plateforme python) :
  - Jupyter lab
  - Tensorflow & Keras
  - QKeras
  - pyTorch
  - pyQt5
  - pyDot
  - HLS4ML

Lors du test de fonctionnement de la plateforme HLS4ML, il se peut que la fonction « compile() » renvoie une erreur de compilation. Cela vient du fichier modèle de construction du modèle HLS, *build_lib.sh*. La version installée de HLS4ML peut ne pas contenir la bonne valeur pour la variable $OSTYPE identifiant le nom du système d'exploitation.

Il faut donc ajouter une ligne dans le fichier template build_lib.sh situé dans l'installation anaconda3.IPython sous ***/lib/****[pythonx.y]****/sites-packages/hls4ml/templates/vivado***.

```
#!/bin/bash

CC=g++
if [[ "$OSTYPE" == "linux" ]]; then
   CFLAGS="-O3 -fPIC -std=c++11 -fno-gnu-unique"
elif [[ "$OSTYPE" == "linux-gnu" ]]; then
   CFLAGS="-O3 -fPIC -std=c++11"
elif [[ "$OSTYPE" == "darwin"* ]]; then
   CFLAGS="-O3 -fPIC -std=c++11"
fi
LDFLAGS=
INCFLAGS="-Ifirmware/ap_types/"
PROJECT=myproject
LIB_STAMP=mystamp

echo ${CC} ${CFLAGS} ${INCFLAGS} -c firmware/${PROJECT}.cpp -o ${PROJECT}.o
${CC} ${CFLAGS} ${INCFLAGS} -c firmware/${PROJECT}.cpp -o ${PROJECT}.o
echo ${CC} ${CFLAGS} ${INCFLAGS} -c ${PROJECT}_bridge.cpp -o ${PROJECT}_bridge.o
${CC} ${CFLAGS} ${INCFLAGS} -c ${PROJECT}_bridge.cpp -o ${PROJECT}_bridge.o
```

```
echo ${CC} ${CFLAGS} ${INCFLAGS} -shared ${PROJECT}.o ${PROJECT}_bridge.o -o firmware/${PROJECT}-${LIB_STAMP}.so
${CC} ${CFLAGS} ${INCFLAGS} -shared ${PROJECT}.o ${PROJECT}_bridge.o -o firmware/${PROJECT}-${LIB_STAMP}.so
rm -f *.o
```

Attention, la variable $OSTYPE peut avoir d'autres valeurs selon l'installation de la machine. Identifier la valeur en tapant : >>*echo $OSTYPE*

Code 1

```
#!/usr/bin/env python
# coding: utf-8
# # Part 1: Getting started

from tensorflow.keras.utils import to_categorical
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
import numpy as np
get_ipython().run_line_magic('matplotlib', 'inline')
seed = 0
np.random.seed(seed)
import tensorflow as tf
tf.random.set_seed(seed)
import os
os.environ['PATH'] = 'D:\Xilinx\Vivado\2019.1\bin:' + os.environ['PATH']

data = fetch_openml('hls4ml_lhc_jets_hlf')
X, y = data['data'], data['target']

# ### Let's print some information about the dataset
# Print the feature names and the dataset shape
print(data['feature_names'])
print(X.shape, y.shape)
print(X[:5])
print(y[:5])

# As you saw above, the `y` target is an array of strings, e.g. \['g', 'w',...\] etc.
# We need to make this a "One Hot" encoding for the training.
# Then, split the dataset into training and validation sets
le = LabelEncoder()
y = le.fit_transform(y)
y = to_categorical(y, 5)
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
print(y[:5])

scaler = StandardScaler()
X_train_val = scaler.fit_transform(X_train_val)
X_test = scaler.transform(X_test)

np.save('X_train_val.npy', X_train_val)
np.save('X_test.npy', X_test)
np.save('y_train_val.npy', y_train_val)
np.save('y_test.npy', y_test)
np.save('classes.npy', le.classes_)

# ## Now construct a model
# We'll use 3 hidden layers with 64, then 32, then 32 neurons. Each layer will use `relu` activation.
# Add an output layer with 5 neurons (one for each class), then finish with Softmax activation.

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l1
from callbacks import all_callbacks

model = Sequential()
model.add(Dense(64, input_shape=(16,), name='fc1', kernel_initializer='lecun_uniform',
kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='relu', name='relu1'))
model.add(Dense(32, name='fc2', kernel_initializer='lecun_uniform', kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='relu', name='relu2'))
model.add(Dense(32, name='fc3', kernel_initializer='lecun_uniform', kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='relu', name='relu3'))
model.add(Dense(5, name='output', kernel_initializer='lecun_uniform', kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='softmax', name='softmax'))

# ## Train the model
# We'll use Adam optimizer with categorical crossentropy loss.
# The callbacks will decay the learning rate and save the model into a directory 'model_1'
# The model isn't very complex, so this should just take a few minutes even on the CPU.
# If you've restarted the notebook kernel after training once, set `train = False` to load the trained model.

train = True
if train:
   adam = Adam(lr=0.0001)
   model.compile(optimizer=adam, loss=['categorical_crossentropy'], metrics=['accuracy'])
   callbacks = all_callbacks(stop_patience = 1000,
                          lr_factor = 0.5,
                          lr_patience = 10,
                          lr_epsilon = 0.000001,
                          lr_cooldown = 2,
                          lr_minimum = 0.0000001,
                          outputDir = 'model_1')
   model.fit(X_train_val, y_train_val, batch_size=1024,
           epochs=30, validation_split=0.25, shuffle=True,
           callbacks = callbacks.callbacks)
else:
   from tensorflow.keras.models import load_model
   model = load_model('model_1/KERAS_check_best_model.h5')

# ## Check performance
# Check the accuracy and make a ROC curve
```

```
import plotting
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
y_keras = model.predict(X_test)
print("Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_keras, axis=1))))
plt.figure(figsize=(9,9))
_ = plotting.makeRoc(y_test, y_keras, le.classes_)


# # Convert the model to FPGA firmware with hls4ml
# Now we will go through the steps to convert the model we trained to a low-latency optimized FPGA firmware with
hls4ml.
# First, we will evaluate its classification performance to make sure we haven't lost accuracy using the fixed-point data
types.
# Then we will synthesize the model with Vivado HLS and check the metrics of latency and FPGA resource usage.
#
# ## Make an hls4ml config & model
# The hls4ml Neural Network inference library is controlled through a configuration dictionary.
# In this example we'll use the most simple variation, later exercises will look at more advanced configuration.

import hls4ml
print (dir("hls4ml"))
backend_list = hls4ml.templates.get_available_backends()
print (backend_list)

config = hls4ml.utils.config_from_keras_model(model, granularity='model')
print("-----------------------------------")
print("Configuration")
plotting.print_dict(config)
print("-----------------------------------")
hls_model = hls4ml.converters.convert_from_keras_model(model,
                                        hls_config=config,
                                        output_dir='model_1/hls4ml_prj',
                                        part='xc7z020clg400-1')

# Let's visualise what we created. The model architecture is shown, annotated with the shape and data types


hls4ml.utils.plot_model(hls_model, show_shapes=True, show_precision=True, to_file=None)

# ## Compile, predict
# Now we need to check that this model performance is still good. We compile the hls_model, and then use
`hls_model.predict` to execute the FPGA firmware with bit-accurate emulation on the CPU.

hls_model.compile()
X_test = np.ascontiguousarray(X_test)
y_hls = hls_model.predict(X_test)

# ## Compare
# That was easy! Now let's see how the performance compares to Keras:

print("Keras  Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_keras, axis=1))))
print("hls4ml Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_hls, axis=1))))
```

```
fig, ax = plt.subplots(figsize=(9, 9))
_ = plotting.makeRoc(y_test, y_keras, le.classes_)
plt.gca().set_prop_cycle(None) # reset the colors
_ = plotting.makeRoc(y_test, y_hls, le.classes_, linestyle='--')

from matplotlib.lines import Line2D
lines = [Line2D([0], [0], ls='-'),
        Line2D([0], [0], ls='--')]
from matplotlib.legend import Legend
leg = Legend(ax, lines, labels=['keras', 'hls4ml'],
          loc='lower right', frameon=False)
ax.add_artist(leg)

# ## Synthesize
# Now we'll actually use Vivado HLS to synthesize the model. We can run the build using a method of our `hls_model` object.
# After running this step, we can integrate the generated IP into a workflow to compile for a specific FPGA board.
# In this case, we'll just review the reports that Vivado HLS generates, checking the latency and resource usage.

# While the C-Synthesis is running, we can monitor the progress looking at the log file by opening a terminal from the notebook home, and executing:
#
# `tail -f model_1/hls4ml_prj/vivado_hls.log`
hls_model.build(csim=False)

# ## Check the reports
# Print out the reports generated by Vivado HLS. Pay attention to the Latency and the 'Utilization Estimates' sections
hls4ml.report.read_vivado_report('model_1/hls4ml_prj/')
```

Code 2

```
#!/usr/bin/env python
# coding: utf-8

# # Part 2: Advanced Configuration

from tensorflow.keras.utils import to_categorical
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import accuracy_score
import numpy as np
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
import plotting
import os
os.environ['PATH'] = '/opt/Xilinx/Vivado/2019.2/bin:' + os.environ['PATH']

# ## Load the dataset

X_train_val = np.load('X_train_val.npy')
X_test = np.ascontiguousarray(np.load('X_test.npy'))
y_train_val = np.load('y_train_val.npy')
```

```
y_test = np.load('y_test.npy', allow_pickle=True)
classes = np.load('classes.npy', allow_pickle=True)

# ## Load the model
# Load the model trained in 'part1_getting_started'. **Make sure you've run through that walkthrough first!**

from tensorflow.keras.models import load_model
model = load_model('model_1/KERAS_check_best_model.h5')
y_keras = model.predict(X_test)

# ## Make an hls4ml config & model
# This time, we'll create a config with finer granularity. When we print the config dictionary, you'll notice that an entry is
created for each named Layer of the model. See for the first layer, for example:
# ```LayerName:
#     fc1:
#         Precision:
#             weight: ap_fixed<16,6>
#             bias:   ap_fixed<16,6>
#             result: ap_fixed<16,6>
#         ReuseFactor: 1
# ```
# Taken 'out of the box' this config will set all the parameters to the same settings as in part 1, but we can use it as a
template to start modifying things.


import hls4ml
config = hls4ml.utils.config_from_keras_model(model, granularity='name')
print("-----------------------------------")
plotting.print_dict(config)
print("-----------------------------------")

# ## Profiling
# As you can see, we can choose the precision of _everything_ in our Neural Network. This is a powerful way to tune the
performance, but it's also complicated. The tools in `hls4ml.model.profiling` can help you choose the right precision for
your model. (That said, training your model with quantization built in can get around this problem, and that is
introduced in Part 4. So, don't go too far down the rabbit hole of tuning your data types without first trying out
quantization aware training with QKeras.)
#
# The first thing to try is to numerically profile your model. This method plots the distribution of the weights (and biases)
as a box and whisker plot. The grey boxes show the values which can be represented with the data types used in the
`hls_model`. Generally, you need the box to overlap completely with the whisker 'to the right' (large values) otherwise
you'll get saturation & wrap-around issues. It can be okay for the box not to overlap completely 'to the left' (small
values), but finding how small you can go is a matter of trial-and-error.
#
# Providing data, in this case just using the first 1000 examples for speed, will show the same distributions captured at
the output of each layer.

get_ipython().run_line_magic('matplotlib', 'inline')
for layer in config['LayerName'].keys():
    config['LayerName'][layer]['Trace'] = True
hls_model = hls4ml.converters.convert_from_keras_model(model,
                                       hls_config=config,
                                       output_dir='model_1/hls4ml_prj_2',
```

```
                                        part='xcu250-figd2104-2L-e')
hls4ml.model.profiling.numerical(model=model, hls_model=hls_model, X=X_test[:1000])

# ## Customize
# Let's just try setting the precision of the first layer weights to something more narrow than 16 bits. Using fewer bits can save resources in the FPGA. After inspecting the profiling plot above, let's try 8 bits with 1 integer bit.
#
# Then create a new `HLSModel`, and display the profiling with the new config. This time, just display the weight profile by not providing any data '`X`'. Then create the `HLSModel` and display the architecture. Notice the box around the weights of the first layer reflects the different precision.

config['LayerName']['fc1']['Precision']['weight'] = 'ap_fixed<8,2>'
hls_model = hls4ml.converters.convert_from_keras_model(model,
                                        hls_config=config,
                                        output_dir='model_1/hls4ml_prj_2',
                                        part='xcu250-figd2104-2L-e')
hls4ml.model.profiling.numerical(model=model, hls_model=hls_model)
hls4ml.utils.plot_model(hls_model, show_shapes=True, show_precision=True, to_file=None)

# ## Trace
# When we start using customised precision throughout the model, it can be useful to collect the output from each layer to find out when things have gone wrong. We enable this trace collection by setting `Trace = True` for each layer whose output we want to collect.

for layer in config['LayerName'].keys():
   config['LayerName'][layer]['Trace'] = True
hls_model = hls4ml.converters.convert_from_keras_model(model,
                                        hls_config=config,
                                        output_dir='model_1/hls4ml_prj_2',
                                        part='xcu250-figd2104-2L-e')

# ## Compile, trace, predict
# Now we need to check that this model performance is still good after reducing the precision. We compile the `hls_model`, and now use the `hls_model.trace` method to collect the model output, and also the output for all the layers we enabled tracing for. This returns a dictionary with keys corresponding to the layer names of the model. Stored at that key is the array of values output by that layer, sampled from the provided data.
# A helper function `get_ymodel_keras` will return the same dictionary for the Keras model.
#
# We'll just run the `trace` for the first 1000 examples, since it takes a bit longer and uses more memory than just running `predict`.

hls_model.compile()
hls4ml_pred, hls4ml_trace = hls_model.trace(X_test[:1000])
keras_trace = hls4ml.model.profiling.get_ymodel_keras(model, X_test[:1000])
y_hls = hls_model.predict(X_test)

# ## Inspect
# Now we can print out, make plots, or do any other more detailed analysis on the output of each layer to make sure we haven't made the performance worse. And if we have, we can quickly find out where. Let's just print the output of the first layer, for the first sample, for both the Keras and hls4ml models.

print("Keras layer 'fc1', first sample:")
print(keras_trace['fc1'][0])
print("hls4ml layer 'fc1', first sample:")
print(hls4ml_trace['fc1'][0])

# ## Compare
# Let's see if we lost performance by using 8 bits for the weights of the first layer by inspecting the accuracy and ROC curve.

print("Keras  Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_keras, axis=1))))
print("hls4ml Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_hls, axis=1))))

fig, ax = plt.subplots(figsize=(9, 9))
_ = plotting.makeRoc(y_test, y_keras, classes)
plt.gca().set_prop_cycle(None) # reset the colors
_ = plotting.makeRoc(y_test, y_hls, classes, linestyle='--')

from matplotlib.lines import Line2D
lines = [Line2D([0], [0], ls='-'),
        Line2D([0], [0], ls='--')]
from matplotlib.legend import Legend
leg = Legend(ax, lines, labels=['keras', 'hls4ml'],
           loc='lower right', frameon=False)
ax.add_artist(leg)

# ## Profiling & Trace Summary
# We lost a small amount of accuracy compared to when we used `ap_fixed<16,6>`, but in many cases this difference will be small enough to be worth the resource saving. You can choose how aggressive to go with quantization, but it's always sensible to make the profiling plots even with the default configuration. Layer-level `trace` is very useful for finding when you reduced the bitwidth too far, or when the default configuration is no good for your model.
#
# With this 'post training quantization', around 8-bits width generally seems to be the limit to how low you can go before suffering significant performance loss. In Part 4, we'll look at using 'training aware quantization' with QKeras to go much lower without losing much performance.
#
# ## ReuseFactor
# Now let's look at the other configuration parameter: `ReuseFactor`.
# Recall that `ReuseFactor` is our mechanism for tuning the parallelism:

# ![reuse.png](attachment:reuse.png)

# So now let's make a new configuration for this model, and set the `ReuseFactor` to `2` for every layer:
# we'll compile the model, then evaulate its performance. (Note, by creating a new config with `granularity=Model`, we're implicitly resetting the precision to `ap_fixed<16,6>` throughout.) Changing the `ReuseFactor` should not change the classification results, but let's just verify that by inspecting the accuracy and ROC curve again!
# Then we'll build the model.

config = hls4ml.utils.config_from_keras_model(model, granularity='Model')
print("-----------------------------------")
print(config)
print("-----------------------------------")
# Set the ReuseFactor to 2 throughout
config['Model']['ReuseFactor'] = 2
hls_model = hls4ml.converters.convert_from_keras_model(model,
                                                       hls_config=config,
                                              output_dir='model_1/hls4ml_prj_2',
                                              part='xcu250-figd2104-2L-e')
hls_model.compile()
y_hls = hls_model.predict(X_test)
print("Keras  Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_keras, axis=1))))
print("hls4ml Accuracy: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_hls, axis=1))))
plt.figure(figsize=(9, 9))
_ = plotting.makeRoc(y_test, y_keras, classes)
plt.gca().set_prop_cycle(None) # reset the colors
_ = plotting.makeRoc(y_test, y_hls, classes, linestyle='--')

# Now build the model
#
# **This can take several minutes.**
#
# While the C-Synthesis is running, we can monitor the progress looking at the log file by opening a terminal from the
notebook home, and executing:
#
# `tail -f model_1/hls4ml_prj_2/vivado_hls.log`
hls_model.build(csim=False)

# And now print the report, compare this to the report from Exercise 1
hls4ml.report.read_vivado_report('model_1/hls4ml_prj_2')
hls4ml.report.read_vivado_report('model_1/hls4ml_prj')
```

Code 3

```
#!/usr/bin/env python
# coding: utf-8
# # Part 3: Compression

from tensorflow.keras.utils import to_categorical
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
import numpy as np
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
seed = 0
np.random.seed(seed)
import tensorflow as tf
tf.random.set_seed(seed)
import os
os.environ['PATH'] = '/opt/Xilinx/Vivado/2019.2/bin:' + os.environ['PATH']

# ## Fetch the jet tagging dataset from Open ML


X_train_val = np.load('X_train_val.npy')
X_test = np.load('X_test.npy')
y_train_val = np.load('y_train_val.npy')
y_test = np.load('y_test.npy')
classes = np.load('classes.npy', allow_pickle=True)
```

```
# ## Now construct a model
# We'll use the same architecture as in part 1: 3 hidden layers with 64, then 32, then 32 neurons. Each layer will use
`relu` activation.
# Add an output layer with 5 neurons (one for each class), then finish with Softmax activation.

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l1
from callbacks import all_callbacks

model = Sequential()
model.add(Dense(64, input_shape=(16,), name='fc1', kernel_initializer='lecun_uniform', kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='relu', name='relu1'))
model.add(Dense(32, name='fc2', kernel_initializer='lecun_uniform', kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='relu', name='relu2'))
model.add(Dense(32, name='fc3', kernel_initializer='lecun_uniform', kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='relu', name='relu3'))
model.add(Dense(5, name='output', kernel_initializer='lecun_uniform', kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='softmax', name='softmax'))

# ## Train sparse
# This time we'll use the Tensorflow model optimization sparsity to train a sparse model (forcing many weights to '0'). In
this instance, the target sparsity is 75%

from tensorflow_model_optimization.python.core.sparsity.keras import prune, pruning_callbacks, pruning_schedule
from tensorflow_model_optimization.sparsity.keras import strip_pruning
pruning_params = {"pruning_schedule" : pruning_schedule.ConstantSparsity(0.75, begin_step=2000, frequency=100)}
model = prune.prune_low_magnitude(model, **pruning_params)

# ## Train the model
# We'll use the same settings as the model for part 1: Adam optimizer with categorical crossentropy loss.
# The callbacks will decay the learning rate and save the model into a directory 'model_2'
# The model isn't very complex, so this should just take a few minutes even on the CPU.
# If you've restarted the notebook kernel after training once, set `train = False` to load the trained model rather than
training again.

train = True
if train:
    adam = Adam(lr=0.0001)
    model.compile(optimizer=adam, loss=['categorical_crossentropy'], metrics=['accuracy'])
    callbacks = all_callbacks(stop_patience = 1000,
                              lr_factor = 0.5,
                              lr_patience = 10,
                              lr_epsilon = 0.000001,
                              lr_cooldown = 2,
                              lr_minimum = 0.0000001,
                              outputDir = 'model_2')
    callbacks.callbacks.append(pruning_callbacks.UpdatePruningStep())
    model.fit(X_train_val, y_train_val, batch_size=1024,
              epochs=30, validation_split=0.25, shuffle=True,
              callbacks = callbacks.callbacks)
    # Save the model again but with the pruning 'stripped' to use the regular layer types
    model = strip_pruning(model)
    model.save('model_2/KERAS_check_best_model.h5')
else:
    from tensorflow.keras.models import load_model
    model = load_model('model_2/KERAS_check_best_model.h5')

# ## Check sparsity
# Make a quick check that the model was indeed trained sparse. We'll just make a histogram of the weights of the 1st
layer, and hopefully observe a large peak in the bin containing '0'. Note logarithmic y axis.

w = model.layers[0].weights[0].numpy()
h, b = np.histogram(w, bins=100)
plt.figure(figsize=(7,7))
plt.bar(b[:-1], h, width=b[1]-b[0])
plt.semilogy()
print('% of zeros = {}'.format(np.sum(w==0)/np.size(w)))

# ## Check performance
# How does this 75% sparse model compare against the unpruned model? Let's report the accuracy and make a ROC
curve. The pruned model is shown with solid lines, the unpruned model from part 1 is shown with dashed lines.
# **Make sure you've trained the model from part 1**

import plotting
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from tensorflow.keras.models import load_model
model_ref = load_model('model_1/KERAS_check_best_model.h5')

y_ref = model_ref.predict(X_test)
y_prune = model.predict(X_test)

print("Accuracy unpruned: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_ref, axis=1))))
print("Accuracy pruned:   {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_prune, axis=1))))

fig, ax = plt.subplots(figsize=(9, 9))
_ = plotting.makeRoc(y_test, y_ref, classes)
plt.gca().set_prop_cycle(None) # reset the colors
_ = plotting.makeRoc(y_test, y_prune, classes, linestyle='--')

from matplotlib.lines import Line2D
lines = [Line2D([0], [0], ls='-'),
         Line2D([0], [0], ls='--')]
from matplotlib.legend import Legend
leg = Legend(ax, lines, labels=['unpruned', 'pruned'],
            loc='lower right', frameon=False)
ax.add_artist(leg)

# # Convert the model to FPGA firmware with hls4ml
# Let's use the default configuration: `ap_fixed<16,6>` precision everywhere and `ReuseFactor=1`, so we can compare
with the part 1 model. We need to use `strip_pruning` to change the layer types back to their originals.
#
# **The synthesis will take a while**
```

```
#
# While the C-Synthesis is running, we can monitor the progress looking at the log file by opening a terminal from the notebook home, and executing:
#
# `tail -f model_2/hls4ml_prj/vivado_hls.log`

import hls4ml

config = hls4ml.utils.config_from_keras_model(model, granularity='model')
print(config)
hls_model = hls4ml.converters.convert_from_keras_model(model,
                                        hls_config=config,
                                        output_dir='model_2/hls4ml_prj',
                                        part='xcu250-figd2104-2L-e')
hls_model.compile()
hls_model.build(csim=False)

# ## Check the reports
# Print out the reports generated by Vivado HLS. Pay attention to the Utilization Estimates' section in particular this time.

hls4ml.report.read_vivado_report('model_2/hls4ml_prj/')


# Print the report for the model trained in part 1. Remember these models have the same architecture, but the model in this section was trained using the sparsity API from tensorflow_model_optimization. Notice how the resource usage had dramatically reduced (particularly the DSPs). When Vivado HLS notices an operation like `y = 0 * x` it can avoid placing a DSP for that operation. The impact of this is biggest when `ReuseFactor = 1`, but still applies at higher reuse as well. **Note you need to have trained and synthesized the model from part 1**

hls4ml.report.read_vivado_report('model_1/hls4ml_prj')
```

Code 4

```
#!/usr/bin/env python
# coding: utf-8

# # Part 4: Quantization

from tensorflow.keras.utils import to_categorical
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
import numpy as np
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
seed = 0
np.random.seed(seed)
import tensorflow as tf
tf.random.set_seed(seed)
import os
os.environ['PATH'] = '/opt/Xilinx/Vivado/2019.2/bin:' + os.environ['PATH']

# ## Fetch the jet tagging dataset from Open ML
```

```
X_train_val = np.load('X_train_val.npy')
X_test = np.load('X_test.npy')
y_train_val = np.load('y_train_val.npy')
y_test = np.load('y_test.npy')
classes = np.load('classes.npy', allow_pickle=True)

# ## Construct a model
# This time we're going to use QKeras layers.
# QKeras is "Quantized Keras" for deep heterogeneous quantization of ML models.
#
# https://github.com/google/qkeras
#
# It is maintained by Google and we recently added support for QKeras model to hls4ml.
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l1
from callbacks import all_callbacks
from tensorflow.keras.layers import Activation
from qkeras.qlayers import QDense, QActivation
from qkeras.quantizers import quantized_bits, quantized_relu

# We're using `QDense` layer instead of `Dense`, and `QActivation` instead of `Activation`. We're also specifying
`kernel_quantizer = quantized_bits(6,0,0)`. This will use 6-bits (of which 0 are integer) for the weights. We also use the
same quantization for the biases, and `quantized_relu(6)` for 6-bit ReLU activations.

model = Sequential()
model.add(QDense(64, input_shape=(16,), name='fc1',
           kernel_quantizer=quantized_bits(6,0,alpha=1), bias_quantizer=quantized_bits(6,0,alpha=1),
           kernel_initializer='lecun_uniform', kernel_regularizer=l1(0.0001)))
model.add(QActivation(activation=quantized_relu(6), name='relu1'))
model.add(QDense(32, name='fc2',
           kernel_quantizer=quantized_bits(6,0,alpha=1), bias_quantizer=quantized_bits(6,0,alpha=1),
           kernel_initializer='lecun_uniform', kernel_regularizer=l1(0.0001)))
model.add(QActivation(activation=quantized_relu(6), name='relu2'))
model.add(QDense(32, name='fc3',
           kernel_quantizer=quantized_bits(6,0,alpha=1), bias_quantizer=quantized_bits(6,0,alpha=1),
           kernel_initializer='lecun_uniform', kernel_regularizer=l1(0.0001)))
model.add(QActivation(activation=quantized_relu(6), name='relu3'))
model.add(QDense(5, name='output',
           kernel_quantizer=quantized_bits(6,0,alpha=1), bias_quantizer=quantized_bits(6,0,alpha=1),
           kernel_initializer='lecun_uniform', kernel_regularizer=l1(0.0001)))
model.add(Activation(activation='softmax', name='softmax'))

# ## Train sparse
# Let's train with model sparsity again, since QKeras layers are prunable.

from tensorflow_model_optimization.python.core.sparsity.keras import prune, pruning_callbacks, pruning_schedule
from tensorflow_model_optimization.sparsity.keras import strip_pruning
pruning_params = {"pruning_schedule" : pruning_schedule.ConstantSparsity(0.75, begin_step=2000, frequency=100)}
model = prune.prune_low_magnitude(model, **pruning_params)

# ## Train the model
# We'll use the same settings as the model for part 1: Adam optimizer with categorical crossentropy loss.
# The callbacks will decay the learning rate and save the model into a directory 'model_2'
# The model isn't very complex, so this should just take a few minutes even on the CPU.
# If you've restarted the notebook kernel after training once, set `train = False` to load the trained model rather than
training again.

train = True
if train:
    adam = Adam(lr=0.0001)
    model.compile(optimizer=adam, loss=['categorical_crossentropy'], metrics=['accuracy'])
    callbacks = all_callbacks(stop_patience = 1000,
                              lr_factor = 0.5,
                              lr_patience = 10,
                              lr_epsilon = 0.000001,
                              lr_cooldown = 2,
                              lr_minimum = 0.0000001,
                              outputDir = 'model_3')
    callbacks.callbacks.append(pruning_callbacks.UpdatePruningStep())
    model.fit(X_train_val, y_train_val, batch_size=1024,
              epochs=30, validation_split=0.25, shuffle=True,
              callbacks = callbacks.callbacks)
    # Save the model again but with the pruning 'stripped' to use the regular layer types
    model = strip_pruning(model)
    model.save('model_3/KERAS_check_best_model.h5')
else:
    from tensorflow.keras.models import load_model
    from qkeras.utils import _add_supported_quantized_objects
    co = {}
    _add_supported_quantized_objects(co)
    model = load_model('model_3/KERAS_check_best_model.h5', custom_objects=co)

# ## Check performance
# How does this model which was trained using 6-bits, and 75% sparsity model compare against the original model?
Let's report the accuracy and make a ROC curve. The quantized, pruned model is shown with solid lines, the unpruned
model from part 1 is shown with dashed lines.
#
#
# We should also check that hls4ml can respect the choice to use 6-bits throughout the model, and match the accuracy.
We'll generate a configuration from this Quantized model, and plot its performance as the dotted line.
# The generated configuration is printed out. You'll notice that it uses 7 bits for the type, but we specified 6!? That's just
because QKeras doesn't count the sign-bit when we specify the number of bits, so the type that actually gets used needs
1 more.
#
# We also use the `OutputRoundingSaturationMode` optimizer pass of `hls4ml` to set the Activation layers to round,
rather than truncate, the cast. This is important for getting good model accuracy when using small bit precision
activations. And we'll set a different data type for the tables used in the Softmax, just for a bit of extra performance.
#
#
# **Make sure you've trained the model from part 1**

import hls4ml
import plotting
hls4ml.model.optimizer.OutputRoundingSaturationMode.layers = ['Activation']
hls4ml.model.optimizer.OutputRoundingSaturationMode.rounding_mode = 'AP_RND'
```

```
hls4ml.model.optimizer.OutputRoundingSaturationMode.saturation_mode = 'AP_SAT'

config = hls4ml.utils.config_from_keras_model(model, granularity='name')
config['LayerName']['softmax']['exp_table_t'] = 'ap_fixed<18,8>'
config['LayerName']['softmax']['inv_table_t'] = 'ap_fixed<18,4>'
print("-----------------------------------")
plotting.print_dict(config)
print("-----------------------------------")
hls_model = hls4ml.converters.convert_from_keras_model(model,
                                                       hls_config=config,
                                                       output_dir='model_3/hls4ml_prj',
                                                       part='xcu250-figd2104-2L-e')
hls_model.compile()

y_qkeras = model.predict(np.ascontiguousarray(X_test))
y_hls = hls_model.predict(np.ascontiguousarray(X_test))

get_ipython().run_line_magic('matplotlib', 'inline')
from sklearn.metrics import accuracy_score
from tensorflow.keras.models import load_model

model_ref = load_model('model_1/KERAS_check_best_model.h5')
y_ref = model_ref.predict(X_test)

print("Accuracy baseline:  {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_ref, axis=1))))
print("Accuracy pruned, quantized: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_qkeras, axis=1))))
print("Accuracy hls4ml: {}".format(accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_hls, axis=1))))

fig, ax = plt.subplots(figsize=(9, 9))
_ = plotting.makeRoc(y_test, y_ref, classes)
plt.gca().set_prop_cycle(None) # reset the colors
_ = plotting.makeRoc(y_test, y_qkeras, classes, linestyle='--')
plt.gca().set_prop_cycle(None) # reset the colors
_ = plotting.makeRoc(y_test, y_hls, classes, linestyle=':')

from matplotlib.lines import Line2D
lines = [Line2D([0], [0], ls='-'),
        Line2D([0], [0], ls='--'),
        Line2D([0], [0], ls=':')]
from matplotlib.legend import Legend
leg = Legend(ax, lines, labels=['baseline', 'pruned, quantized', 'hls4ml'],
           loc='lower right', frameon=False)
ax.add_artist(leg)

# # Synthesize
# Now let's synthesize this quantized, pruned model.
#
# **The synthesis will take a while**
#
# While the C-Synthesis is running, we can monitor the progress looking at the log file by opening a terminal from the
notebook home, and executing:
#
# `tail -f model_3/hls4ml_prj/vivado_hls.log`
```

```
hls_model.build(csim=False)

# ## Check the reports
# Print out the reports generated by Vivado HLS. Pay attention to the Utilization Estimates' section in particular this
time.

hls4ml.report.read_vivado_report('model_3/hls4ml_prj')

# Print the report for the model trained in part 1. Now, compared to the model from part 1, this model has been trained
with low-precision quantization, and 75% pruning. You should be able to see that we have saved a lot of resource
compared to where we started in part 1. At the same time, referring to the ROC curve above, the model performance is
pretty much identical even with this drastic compression!
#
# **Note you need to have trained and synthesized the model from part 1**

hls4ml.report.read_vivado_report('model_1/hls4ml_prj')

# Print the report for the model trained in part 3. Both these models were trained with 75% sparsity, but the new model
uses 6-bit precision as well. You can see how Vivado HLS has moved multiplication operations from DSPs into LUTs,
reducing the "critical" resource usage.
#
# **Note you need to have trained and synthesized the model from part 3**

hls4ml.report.read_vivado_report('model_2/hls4ml_prj')
```