Rubik Cube Solver

Author: Christopher Athaide, EYE Numerics, LLC

This notebook provides the code to solve the Rubik cube using a deep learning framework.

Directory information

In[1]:=
```
mainDirectory = "D:\\Apps\\Projects\\Wolfram\\SampleProjects\\RubikCube\\";
dataDirectory = mainDirectory <> "data\\";
tmpDirectory = mainDirectory <> "tmp\\";
modelDirectory = mainDirectory <> "models\\";
```

Basic definitions

In[5]:=
```
mkFace[index_Integer] := ArrayReshape[Range[1, 9] + (index - 1) * 9, {3, 3}]
arr2Faces[arr_] := Map[Partition[#, 3] &]@Partition[arr, 9]
rotateCounterClockwise[mat_] := Reverse@Transpose@mat
rotateClockwise[mat_] := Transpose@Reverse@mat
rCC[mat_] := Reverse@Transpose@mat
rC[mat_] := Transpose@Reverse@mat
flipUpDown[mat_] := Reverse@mat
flipLeftRight[mat_] := Reverse /@ mat
flipLeftRightUpDown[mat_] := Map[Reverse]@(Reverse@mat)
{frontI, topI, rightI, backI, leftI, bottomI} = Range@6;
allFaces = mkFace /@ Range[6];
allFacesArray = Flatten@allFaces;
centerIndexes = Range[1, 6] * 9 - 4;
nonCenterIndexes = Complement[Range@54, centerIndexes];
nonCenterIndexDict = Association@Map[#[[2]] → #[[1]] &]@il@nonCenterIndexes;
nonCenterIndexArray = Last /@ SortBy[First]@(Join @@
      {Map[{#[[2]], #[[1]]} &]@il@nonCenterIndexes, Map[{#, -1} &]@centerIndexes});
cornerIndexes = {{1, 39, 16}, {3, 18, 19}, {7, 46, 45}, {9, 25, 48},
   {28, 21, 12}, {30, 10, 37}, {34, 54, 27}, {36, 43, 52}};
sideIndexes = {{2, 17}, {4, 42}, {6, 22}, {8, 47}, {20, 15}, {26, 51},
   {38, 13}, {44, 49}, {29, 11}, {31, 24}, {33, 40}, {35, 53}};
cornerLocators = First /@ cornerIndexes;
sideLocators = First /@ sideIndexes;
mainLocators = cornerLocators ~ Join ~ sideLocators;
cornerIndexArray = Flatten@cornerIndexes;
sideIndexArray = Flatten@sideIndexes;

locatorArray = cornerLocators ~ Join ~ sideLocators;
cornerMaps =
  SortBy[First]@Flatten[{#, RotateLeft[#], RotateRight[#]} & /@ cornerIndexes, 1];
cornerDict = Association@Map[First[#] → # &]@cornerMaps;
sideMaps = SortBy[First]@Flatten[{#, RotateLeft[#]} & /@ sideIndexes, 1];
sideDict = Association@Map[First[#] → # &]@sideMaps;
rotateCubeAboutFrontClockwise[faces_] := {rC[faces[[frontI]]], rC[faces[[leftI]]],
  rC[faces[[topI]]], rCC[faces[[backI]]], rC[faces[[bottomI]]], rC[faces[[rightI]]]}
rotateCubeAboutFrontCounterClockwise[faces_] := {rCC[faces[[1]]], rCC[faces[[rightI]]],
```

```
  rCC[faces[[bottomI]]], rC[faces[[backI]]], rCC[faces[[topI]]], rCC[faces[[leftI]]]}
rotateCubeAboutRightClockwise[faces_] := {faces[[bottomI]], faces[[frontI]],
  rC[faces[[rightI]]], flipLeftRightUpDown[faces[[topI]]],
  rCC[faces[[leftI]]], flipLeftRightUpDown[faces[[backI]]]}
rotateCubeAboutRightCounterClockwise[faces_] :=
 {faces[[topI]], flipLeftRightUpDown@faces[[backI]], rCC[faces[[rightI]]],
  flipLeftRightUpDown[faces[[bottomI]]], rC[faces[[leftI]]], faces[[frontI]]}
rotateCubeAboutTopClockwise[faces_] := {faces[[rightI]], rC@faces[[topI]],
  faces[[backI]], faces[[leftI]], faces[[frontI]], rCC@faces[[bottomI]]}
rotateCubeAboutTopCounterClockwise[faces_] := {faces[[leftI]], rCC@faces[[topI]],
  faces[[frontI]], faces[[rightI]], faces[[backI]], rC@faces[[bottomI]]}
fillCube[corners_, sides_, centers_ : centerIndexes] :=
 With[{arr = (Thread[{cornerIndexArray, Flatten@Map[cornerDict[#] &]@corners}] ~
       Join ~ Thread[{sideIndexArray, Flatten@Map[sideDict[#] &]@sides}] ~
       Join ~ Thread[{centerIndexes, centers}])},
  (Map[Partition[#, 3] &]@Partition[#, 9] &)@(Last /@ SortBy[First]@arr)]
hexadecimalCharacters = ToString /@ (Range@9) ~ Join ~ {"A", "B", "C", "D", "E", "F"};
hexadecimalDict = Association@Map[#[[1]] → #[[2]] &]@
    (Reverse /@ (Thread[{Range[1, 13] ~ Join ~ {13, 13}, hexadecimalCharacters}]));
hexadecimalCharacters = ToString /@ (Range@9) ~ Join ~ {"A", "B", "C", "D", "D", "D"};
digit2Str = Association[
   Map[# → "0" <> ToString[#] &]@Range[0, 9] ~ Join ~ Map[# → ToString[#] &]@Range[10, 99]];
getCurrentTime[] := With[{tmp = TimeObject[] //. TimeObject[aA_, _, _] → aA},
                         ToString[digit2Str[tmp[[1]]]] <> ":" <>
   ToString[digit2Str[tmp[[2]]]] <> ":" <> ToString[digit2Str[IntegerPart@tmp[[3]]]]
                         ]
splitNumber[n_Integer] := Map[{#, n - #} &]@Range[1, n - 1]
indexRemapFunction[positions54Vec_] :=
 (nonCenterIndexArray[[#[[nonCenterIndexes]]]] &) /@ positions54Vec
(*fillCube[{7,1,9,3,28,30,34,36},{4,8,2,6,20,26,38,44,29,31,33,35}]*)
```

Some more definitions

In[47]:=
```
extractAnchorPositions[faces_] := {Part[faces, cornerLocators], Part[faces, sideLocators]}
extractCenterPositions[faces_] := Part[faces, centerIndexes]
rotFaceFrontClockwiseArray = {7, 4, 1, 8, 5, 2, 9, 6, 3, 10, 11, 12, 13, 14,
   15, 45, 42, 39, 16, 20, 21, 17, 23, 24, 18, 26, 27, 28, 29, 30, 31, 32, 33, 34,
   35, 36, 37, 38, 46, 40, 41, 47, 43, 44, 48, 25, 22, 19, 49, 50, 51, 52, 53, 54};
rotFaceFrontCounterClockwiseArray = {3, 6, 9, 2, 5, 8, 1, 4, 7, 10, 11, 12, 13, 14,
   15, 19, 22, 25, 48, 20, 21, 47, 23, 24, 46, 26, 27, 28, 29, 30, 31, 32, 33, 34,
   35, 36, 37, 38, 18, 40, 41, 17, 43, 44, 16, 39, 42, 45, 49, 50, 51, 52, 53, 54};
rotFaceTopClockwiseArray = {19, 20, 21, 4, 5, 6, 7, 8, 9, 16, 13, 10, 17, 14, 11,
   18, 15, 12, 28, 29, 30, 22, 23, 24, 25, 26, 27, 37, 38, 39, 31, 32, 33, 34,
   35, 36, 1, 2, 3, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54};
rotFaceTopCounterClockwiseArray = {37, 38, 39, 4, 5, 6, 7, 8, 9, 12, 15, 18, 11,
   14, 17, 10, 13, 16, 1, 2, 3, 22, 23, 24, 25, 26, 27, 19, 20, 21, 31, 32, 33, 34,
   35, 36, 28, 29, 30, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54};
rotFaceRightClockwiseArray = {1, 2, 48, 4, 5, 51, 7, 8, 54, 10, 11, 3, 13, 14, 6,
   16, 17, 9, 25, 22, 19, 26, 23, 20, 27, 24, 21, 18, 29, 30, 15, 32, 33, 12, 35,
   36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 34, 49, 50, 31, 52, 53, 28};
rotFaceRightCounterClockwiseArray = {1, 2, 12, 4, 5, 15, 7, 8, 18, 10, 11, 34, 13,
```

```
    14, 31, 16, 17, 28, 21, 24, 27, 20, 23, 26, 19, 22, 25, 54, 29, 30, 51, 32, 33,
    48, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 3, 49, 50, 6, 52, 53, 9};
rotFaceLeftClockwiseArray = {10, 2, 3, 13, 5, 6, 16, 8, 9, 36, 11, 12, 33, 14,
    15, 30, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 52, 31, 32, 49, 34,
    35, 46, 43, 40, 37, 44, 41, 38, 45, 42, 39, 1, 47, 48, 4, 50, 51, 7, 53, 54};
rotFaceLeftCounterClockwiseArray = {46, 2, 3, 49, 5, 6, 52, 8, 9, 1, 11, 12, 4, 14,
    15, 7, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 16, 31, 32, 13, 34,
    35, 10, 39, 42, 45, 38, 41, 44, 37, 40, 43, 36, 47, 48, 33, 50, 51, 30, 53, 54};
rotFaceBackClockwiseArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 21, 24, 27, 13, 14, 15,
    16, 17, 18, 19, 20, 54, 22, 23, 53, 25, 26, 52, 34, 31, 28, 35, 32, 29, 36, 33,
    30, 12, 38, 39, 11, 41, 42, 10, 44, 45, 46, 47, 48, 49, 50, 51, 37, 40, 43};
rotFaceBackCounterClockwiseArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 43, 40, 37, 13, 14,
    15, 16, 17, 18, 19, 20, 10, 22, 23, 11, 25, 26, 12, 30, 33, 36, 29, 32, 35, 28,
    31, 34, 52, 38, 39, 53, 41, 42, 54, 44, 45, 46, 47, 48, 49, 50, 51, 27, 24, 21};
rotFaceBottomClockwiseArray = {1, 2, 3, 4, 5, 6, 43, 44, 45, 10, 11, 12, 13, 14,
    15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 7, 8, 9, 28, 29, 30, 31, 32, 33, 25, 26,
    27, 37, 38, 39, 40, 41, 42, 34, 35, 36, 52, 49, 46, 53, 50, 47, 54, 51, 48};
rotFaceBottomCounterClockwiseArray = {1, 2, 3, 4, 5, 6, 25, 26, 27, 10, 11, 12, 13,
    14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 34, 35, 36, 28, 29, 30, 31, 32, 33,
    43, 44, 45, 37, 38, 39, 40, 41, 42, 7, 8, 9, 48, 51, 54, 47, 50, 53, 46, 49, 52};
rotateFaceFrontClockwise[faces_] :=
 With[{tmp = Flatten@{rC@faces[[frontI]], Sequence @@ faces[[2 ;; 6]]}},
  fillCube[Sequence @@ extractAnchorPositions[tmp], extractCenterPositions[tmp]]]
rotateFaceFrontCounterClockwise[faces_] :=
 With[{tmp = Flatten@{rCC@faces[[frontI]], Sequence @@ faces[[2 ;; 6]]}},
  fillCube[Sequence @@ extractAnchorPositions[tmp], extractCenterPositions[tmp]]]
rotateFaceTopClockwise[faces_] := With[
  {tmp = Flatten@rotateFaceFrontClockwise@rotateCubeAboutRightCounterClockwise[faces]},
  rotateCubeAboutRightClockwise[
   fillCube[Sequence @@ extractAnchorPositions[tmp], extractCenterPositions[tmp]]]]
rotateFaceTopCounterClockwise[faces_] :=
 With[{tmp = Flatten@rotateFaceFrontCounterClockwise@
       rotateCubeAboutRightCounterClockwise[faces]}, rotateCubeAboutRightClockwise[
   fillCube[Sequence @@ extractAnchorPositions[tmp], extractCenterPositions[tmp]]]]
rotateFaceRightClockwise[faces_] :=
 With[{tmp = Flatten@rotateFaceFrontClockwise@rotateCubeAboutTopClockwise[faces]},
  rotateCubeAboutTopCounterClockwise[
   fillCube[Sequence @@ extractAnchorPositions[tmp], extractCenterPositions[tmp]]]]
rotateFaceRightCounterClockwise[faces_] :=
 With[{tmp = Flatten@rotateFaceFrontCounterClockwise@rotateCubeAboutTopClockwise[faces]},
  rotateCubeAboutTopCounterClockwise[
   fillCube[Sequence @@ extractAnchorPositions[tmp], extractCenterPositions[tmp]]]]
rotateFaceLeftClockwise[faces_] := With[
  {tmp = Flatten@rotateFaceFrontClockwise@rotateCubeAboutTopCounterClockwise[faces]},
  rotateCubeAboutTopClockwise[
   fillCube[Sequence @@ extractAnchorPositions[tmp], extractCenterPositions[tmp]]]]
rotateFaceLeftCounterClockwise[faces_] := With[{tmp =
    Flatten@rotateFaceFrontCounterClockwise@rotateCubeAboutTopCounterClockwise[faces]},
  rotateCubeAboutTopClockwise[fillCube[Sequence @@ extractAnchorPositions[tmp],
    extractCenterPositions[tmp]]]]
rotateFaceBackClockwise[faces_] := With[
```

```
  {tmp = Flatten@{faces[[1 ;; 3]], rC@faces[[backI]], Sequence @@ faces[[5 ;; 6]]}},
  fillCube[Sequence @@ extractAnchorPositions[tmp], extractCenterPositions[tmp]]]
rotateFaceBackCounterClockwise[faces_] :=
 With[{tmp = Flatten@{faces[[1 ;; 3]], rCC@faces[[backI]], Sequence @@ faces[[5 ;; 6]]}},
  fillCube[Sequence @@ extractAnchorPositions[tmp], extractCenterPositions[tmp]]]
rotateFaceBottomCounterClockwise[faces_] := With[
  {tmp = Flatten@rotateFaceFrontCounterClockwise@rotateCubeAboutRightClockwise[faces]},
  rotateCubeAboutRightCounterClockwise[
   fillCube[Sequence @@ extractAnchorPositions[tmp], extractCenterPositions[tmp]]]]
rotateFaceBottomClockwise[faces_] :=
 With[{tmp = Flatten@rotateFaceFrontClockwise@rotateCubeAboutRightClockwise[faces]},
  rotateCubeAboutRightCounterClockwise[
   fillCube[Sequence @@ extractAnchorPositions[tmp], extractCenterPositions[tmp]]]]
rotateFaces[arr1 : {__Integer}] := arr1
rotateFaces[{arr1 : {__Integer}}] := arr1
rotateFaces[arr1 : {__Integer}, arr2__ : {__Integer}] := arr1[[rotateFaces@arr2]]
rotateFaces[{arr1 : {__Integer}, arr2__ : {__Integer}}] := arr1[[rotateFaces@arr2]]
moveTensor = {{{rotFaceFrontClockwiseArray, rotFaceFrontCounterClockwiseArray},
    {rotFaceTopClockwiseArray, rotFaceTopCounterClockwiseArray},
    {rotFaceRightClockwiseArray, rotFaceRightCounterClockwiseArray}},
   {{rotFaceBackClockwiseArray, rotFaceBackCounterClockwiseArray},
    {rotFaceLeftClockwiseArray, rotFaceLeftCounterClockwiseArray},
    {rotFaceBottomClockwiseArray, rotFaceBottomCounterClockwiseArray}}};
moveArray = {rotFaceFrontClockwiseArray, rotFaceFrontCounterClockwiseArray,
   rotFaceTopClockwiseArray, rotFaceTopCounterClockwiseArray,
   rotFaceRightClockwiseArray, rotFaceRightCounterClockwiseArray,
   rotFaceBackClockwiseArray, rotFaceBackCounterClockwiseArray,
   rotFaceLeftClockwiseArray, rotFaceLeftCounterClockwiseArray,
   rotFaceBottomClockwiseArray, rotFaceBottomCounterClockwiseArray, Range@54};
rotateFacesByIndex[i1 : _Integer] := rotateFaces[ moveArray[[i1]]]
rotateFacesByIndex[arr1 : {__Integer}] := rotateFaces[ (moveArray[[#]] &) /@ arr1 ]
getRedundantMaps[{}, moveArr_] := {}
getRedundantMaps[indexes_, moveArr_] :=
     Module[{tmpArr, result},
          tmpArr = SortBy[First]@
     Map[Function[singleIndex, {Length@moveArr[[singleIndex]], singleIndex} ]]@indexes;
          result = Last@tmpArr[[1]];
          Map[{#, result} &]@(Last /@ tmpArr)
          ]
arr2Hex[arr_] := StringJoin[ (hexadecimalCharacters[[#]] &) /@ arr]
inverseMove[moveStr_String] := arr2Hex@Map[If[OddQ[#], # + 1, # - 1] &]@
   Flatten[Lookup[hexadecimalDict, Characters[StringReverse[moveStr]]]]
getShortestMove[key_, indexes_, moveArr_, resultDict_] :=
     Module[{tmpArr, result},
          tmpArr = SortBy[First]@
     Map[Function[singleIndex, {Length@moveArr[[singleIndex]], singleIndex} ]]@indexes;
          result = Last@tmpArr[[1]];
          result → resultDict[key]
         ]
```

Dictionary of short move sequences

```
In[86]:= short6 = Import[
    "D:\\Apps\\Projects\\Wolfram\\SampleProjects\\RubikCube\\data\\shortestMoves_6.du"];
redundantKeys6 = Import[
    "D:\\Apps\\Projects\\Wolfram\\SampleProjects\\RubikCube\\data\\redundantKeyDict_6.du"];
rotateUsingMoveStr[moveStr_String] :=
      If[StringLength@moveStr ≤ 6,
   Flatten@fillCube[Sequence @@ short6[Lookup[redundantKeys6, moveStr, moveStr]]],
                      First[(rotateFaces[#[[1]], #[[2]]] &) /@
     ((Flatten@fillCube[Sequence @@ #] &) /@

          {With[{ky = StringTake[#, 6]}, Lookup[short6, ky, short6[redundantKeys6[ky]]]],

           With[{ky1 = StringDrop[#, 6]},

            If[StringLength@ky1 ≤ 6, Lookup[short6, ky1, short6[redundantKeys6[ky1]]],

             extractAnchorPositions@rotateUsingMoveStr@ky1
                                                                                  ]
                                                                              ]
                                                                  } &) /@ {moveStr}]
          ]
```

Build larger moves from smaller ones

```
In[107]:= createLargerMoves[{}] := Range@54
move8Lst =
   Import["D:\\Apps\\Projects\\Wolfram\\SampleProjects\\RubikCube\\data\\Moves_8.du"];
reducedNumberPairDict =
      With[
    {numberPairDict = Block[{allNumberPairs = Join @@ (splitNumber[#] & /@ Range[2, 16]), f},
                                  f[numPairs_] :=
        Function[pair, If[Total@pair ≤ 14, pair[[1]] * 10 + pair[[2]] →
           Select[ #[[1]] ≤  pair[[1]] && #[[2]] ≤  pair[[2]] &]@numPairs, Nothing]];
                                  Association@Map[f[allNumberPairs]]@
          Tuples[Range@20, 2]
                                  ]
             },
               Association@
     KeyValueMap[#1 → Select[#2, Function[lst, Total@lst ≤ 6]] &]@numberPairDict
            ];
redundantMoveSequenceQ[move1_String, move2_String] :=
          With[{dictKey = Min[StringLength@move1 * 10 + StringLength@move2, 66]},
                      (If[KeyExistsQ[redundantKeys6, #],
        StringLength@redundantKeys6[#] < StringLength@#, False] &) /@

    (Map[StringTake[move1, -Min[StringLength@move1, #[[1]]]] <> StringTake[move2,
          Min[StringLength@move2, #[[2]]]] &, reducedNumberPairDict[dictKey]])
                ]
createLargerMoves[move1_String] := rotateFaces[rotateUsingMoveStr@move1]
createLargerMoves[move1_String, move2_String] :=
```

```
 Block[{keyPair = {move1, move2}, flags, p1, p2},
          flags = redundantMoveSequenceQ[move1, move2];
          If[MemberQ[flags, True], Null,
                   rotateFaces[rotateUsingMoveStr@move1, rotateUsingMoveStr@move2]
           ]
       ]
createLargerMoves[move1_String, move2_String, move3_String] :=
 Block[{keyPair = {move1, move2}, flags, p1, p2},
          flags = redundantMoveSequenceQ[move1, move2] ~
    Join ~ redundantMoveSequenceQ[move2, move3];
          If[MemberQ[flags, True], Null,
                   rotateFaces[rotateUsingMoveStr@move1,
    rotateUsingMoveStr@move2, rotateUsingMoveStr@move3]
           ]
       ]
getShortKeys[shortKeyDict_Association, m_Integer] :=
 shuffleList@Flatten@KeyValueMap[RandomChoice[shortKeyDict[#1], #2] &]@
    (KeySort@Counts[RandomChoice[Range@6, m]])
getRandomMoveDoubles[n_Integer] := Block[{vec, keys, shortKeys},
                          vec = Flatten@RandomChoice[{1, 2, 3}, {n, 2}];

  keys = With[{dict = Counts[vec], keysByLength = GroupBy[StringLength]@(Keys@short6)},
                                        {getShortKeys[keysByLength, dict[1]],
     getShortKeys[keysByLength, dict[2]], RandomSample[move8Lst, dict[3]]}
                                    ];

  Partition[Last /@ SortBy[First]@Thread[{Flatten@{Position[vec, 1],
         Position[vec, 2], Position[vec, 3]}, Flatten@keys}], 2]
                   ]
getRandomMoveTriples[n_Integer] := Block[{vec, keys},
                          vec = Flatten@RandomChoice[{1, 2, 3}, {n, 3}];

  keys = With[{dict = Counts[vec], keysByLength = GroupBy[StringLength]@(Keys@short6)},
                                        {getShortKeys[keysByLength, dict[1]],
     getShortKeys[keysByLength, dict[2]], RandomSample[move8Lst, dict[3]]}
                                    ];

  Partition[Last /@ SortBy[First]@Thread[{Flatten@{Position[vec, 1],
         Position[vec, 2], Position[vec, 3]}, Flatten@keys}], 3]
                   ]
```

Building the Solver

A) Create a few test moves

In[117]:=
```
SeedRandom[123457]
joinedMoves = Map[StringJoin]@
    (Select[getRandomMoveDoubles[50], StringLength[#[[1]] <> #[[2]]] > 6 &] ~
      Join ~ getRandomMoveTriples[50]);
```

In[119]:=
```
ClearAll[move8Lst]
choiceIndex = 1;
arr1 = rotateUsingMoveStr[joinedMoves[[choiceIndex]]];
arr3 = rotateUsingMoveStr[joinedMoves[[3]]];
{joinedMoves[[choiceIndex]], inverseMove[joinedMoves[[choiceIndex]]]}
arr2 = rotateUsingMoveStr[inverseMove[joinedMoves[[2]]]];
```

B) Build the library of up to 6 moves

In[126]:=
```
getUptoMove6Results[] := Module[{tmp3,  tmp4, sKeys},
      tmp3 = Import[FileNameJoin[{dataDirectory, "allUpToNMoves_6.du"}]];
      tmp4 = Import[FileNameJoin[{dataDirectory, "redundantKeyDict_6.du"}]];
      sKeys = Complement[arr2Hex /@ tmp3, Keys@tmp4];

  Association@Map[With[{res = rotateUsingMoveStr@#}, Hash@(res) → {#, res}] &, sKeys]
 ]
move6Results = getUptoMove6Results[];
move5Results = Select[move6Results, StringLength[#[[1]]] ≤ 5 &];
move4Results = Select[move6Results, StringLength[#[[1]]] ≤ 4 &];
move3Results = Select[move6Results, StringLength[#[[1]]] ≤ 3 &];
(*move7Hashes=Import[FileNameJoin[{dataDirectory,"move7Hashes.du"}]];*)
move8Hashes =
  AssociationMap[1 &]@Import[FileNameJoin[{dataDirectory, "move8Hashes.du"}]];
```

C) Load the neural net prediction function.

In[133]:=
```
predictor = Import[FileNameJoin[{modelDirectory, "trainedNet5j.du"}]];
```

D) Solve the Rubik cube.

```
ClearAll[solveCube, iterateOverMoves,
 evaluateNextMoves, recordTimeElapsed, getMoveAndState]
ClearAll[checkfinalZoneState, getNextStates2,
 finalizeMoveSequence, recordTimeElapsed, getStateHistory]
recordTimeElapsed[t_] := DateDifference[t, getCurrentTime[], "Second"]
getStateHistory[stateLst_, stateAndMoves_] :=
 Table[({#[[1]], rotateUsingMoveStr[#[[2]]], #[[2]]} &)@
   {stateLst[[pair[[1]]]], pair[[2]]}, {pair, stateAndMoves}]
finalizeMoveSequence[initialMoves_, stateMoveTriples_(* start state,
  move state, move string *), moveDict_] :=
       Module[{finalStates, finalHashStates1, movePairs,
   indexes, positionsAndValues1, getCompleteMove1, currTime},
                finalStates = Table[triples[[1]][[triples[[2]]]],
    {triples, stateMoveTriples}];
                movePairs = Select[Values@moveDict, StringLength[First@#] == 2 &];
  (* Select length 2 moves *)
                positionsAndValues1[mainDict_, subSet_] :=
   ({#, subSet[[#]]} &) /@ (Flatten@Position[Lookup[mainDict, subSet, 0], 1]);
                getCompleteMove1[state_] :=
   With[{moveHashes = Hash /@ Partition[state[[Flatten[Last /@ movePairs]]], 54]},

    positionsAndValues1[moveDict, moveHashes]
                                              ];
```

```
                currTime = getCurrentTime[];
                finalHashStates1 = Table[

    Table[{First@indexStatePair, item}, {item, getCompleteMove1[Last@indexStatePair]}],
                                          {indexStatePair, il@finalStates}
                                       ];
                finalHashStates1 =
   Select[Flatten[finalHashStates1, 1], KeyExistsQ[moveDict, #[[2]][[2]]] &];
  (* Final filter to ensure that all keys are valid *)
                indexes =
   Table[{initialMoves[[item[[1]]]], stateMoveTriples[[item[[1]]][[3]],
                                        First@movePairs[[item[[2]][[1]]]],
     inverseMove[ Lookup[moveDict, item[[2]][[2]]][[1]]]},
                                     {item, finalHashStates1}
                                   ];
                StringJoin /@ indexes
                ]
checkfinalZoneState[dbKeys_, nextMoveList_, currStateList_] :=
     With[{n = Length@nextMoveList,
                            futureStates =
    Flatten@Outer[Hash[#1[[#2[[-1]]]]] &, currStateList, nextMoveList, 1, 1]},

  (Thread[{IntegerPart[(# - 1) / n] + 1 , First /@ nextMoveList[[Mod[# - 1, n] + 1]] }] &)[
   Flatten@Position[Lookup[dbKeys, futureStates, -1], 1]]
         ]
getMoveAndState = Function[{startArr, dictStateIndexes, moveDict},
   Map[{First@#(*String*), startArr[[Last@#]] (*State*)} &]@
    Values@Part[moveDict, dictStateIndexes]];
getNextStates2[True, statePredictionFn_, dbKeys_, nextMoveList_,
  currStateList_, numStates_] :=
     Module[{n = Length@nextMoveList, currTime, futureStates, commonStates, f1, g1, h1},

  h1 := Function[ranks, Part[First /@ (SortBy[Last]@(il@ranks)), 1 ;; numStates]];
              g1 = Function[arr, h1@Flatten@statePredictionFn[
       Map[nonCenterIndexArray[[#[[nonCenterIndexes]]]] &]@arr, TargetDevice → "GPU"]];
              f1 = Function[state,  With[
     {tbl = Table[({#, Hash@#} &)[state[[move[[-1]]]]], {move, nextMoveList}] },

     {g1[First /@ tbl] (* Best moves *), Last /@ tbl (* All move hashes *)}
                                        ]
                                  ];
              currTime = getCurrentTime[];
              futureStates = Map[f1]@currStateList;

  commonStates = Flatten@Position[Lookup[dbKeys, Flatten[Last /@ futureStates], -1], 1];
              If[Length@commonStates > 0,

   {True, Thread[{IntegerPart[(commonStates - 1) / n] + 1 (* Destination State Index *),
      First /@ nextMoveList[[Mod[commonStates - 1, n] + 1]] }]},
                        {False, First /@ futureStates}
```

```
                    ]
          ]
getNextStates2[False, statePredictionFn_,
  dbKeys_, nextMoveList_, currStateList_, numStates_] :=
        Module[{n = Length@nextMoveList, futureStates,
   commonStates, f1(*, f2,*)(*g1,h1*)},
            f1 = Table[Hash@#[[move]], {move, Last /@ nextMoveList}] &;

  commonStates = Flatten@Position[Lookup[dbKeys, Flatten[f1 /@ currStateList], -1], 1];
            If[Length@commonStates > 0,

   {True, Thread[{IntegerPart[(commonStates - 1) / n] + 1 (* Destination State Index *),
      First /@ nextMoveList[[Mod[commonStates - 1, n] + 1]] }]},
                {False, {}}
             ]
          ]

evaluateNextMoves[statePredictionFn_, currStateArr54_, results_, moveDict1_, moveDict2_,
  moveHashes3_(*, numEvalStates_*)] := Module[{states, res2, chainFn, solved},
          states = getMoveAndState[currStateArr54, results, moveDict1];
          {solved, res2} = getNextStates2[True, statePredictionFn, moveHashes3,
    Values@moveDict2, Last /@ states(*[[1;;numEvalStates]]*), 20];
          chainFn = (Function[s, (s <> # &) /@#[[2]]][#[[1]]] &)@
     {#[[1, 1]], First /@#[[2]]} &;
          If[solved,
                       With[{sh = {currStateArr54, rotateUsingMoveStr@#, #} & /@
       StringJoin /@ Thread[{First /@ Part[states, First /@ res2], Last /@ res2}]},
                            {True, (MinimalBy[Union[#], StringLength] &)@
      finalizeMoveSequence[Map[{} &]@sh, sh, moveDict2]}
                        ],
                       {False, Flatten[chainFn /@
      Thread[{states(*[[1;;20]]*), Map[Part[Values@move4Results, #] &]@res2 }]]}
                ]
 ]

iterateOverMoves[statePredictionFn_, currStateArr54_, results_,
  moveDict1_, moveDict2_, moveHashes3_, numCases_, iterLimit_] :=
        Module[{solvedA = False, secondResults, indexes, iter = 0, cumulativeMoves = {}},
                    indexes = Partition[Range@numCases, numCases / iterLimit];
                    While[! solvedA && iter++ < iterLimit,
                             {solvedA, secondResults} =
    evaluateNextMoves[statePredictionFn, currStateArr54, Part[results, indexes[[iter]]],

     moveDict1, moveDict2, moveHashes3];
                             If[! solvedA, AppendTo[cumulativeMoves, secondResults]];
                         ];
                    {solvedA, If[solvedA, secondResults, cumulativeMoves]}
                ]
solveCube[statePredictionFn_, currStateArr54_,
```

```
  firstDict_, secondDict_, numCases_, iterLimit_] :=
        Module[{solved = False, reportResults, secondResults,
   currTime, numStates = 5000, firstResults},
                 reportResults[result_] :=
   With[{sh = getStateHistory[{currStateArr54}, result]},

    {True, Union[(MinimalBy[Union[#], StringLength] &)@
       finalizeMoveSequence[Map[{} &]@sh, sh, firstDict]]}
                                                    ];
                 {solved, firstResults} = getNextStates2[True, statePredictionFn,
    move8Hashes, Values@firstDict, {currStateArr54}, numStates];
                 (*Echo[firstResults[[1]][[1;;5]]];*)
                 If[solved, reportResults[firstResults],

   {solved, secondResults} = iterateOverMoves[statePredictionFn, currStateArr54,
     firstResults[[1]], firstDict, secondDict, move8Hashes, numCases, iterLimit];
                          If[solved,
                               {solved, Union[secondResults]},
                                   {solved, secondResults}
                           ]
                      ]
                 (*{solved,firstResults}*)
             ]

Do[
 currTime = getCurrentTime[];
 res1 = solveCube[predictor,
   rotateUsingMoveStr@joinedMoves[[index]], move6Results, move4Results, 5000, 50];
 Echo[{"Cumulative: ", {index, recordTimeElapsed[currTime], First@res1}}];
 , {index, {1, 2, 3(*13,30,35,44,45,47,48,50*)}}]
```

» {Cumulative: , {1, 16 s , True}}

» {Cumulative: , {2, 17 s , True}}

» {Cumulative: , {3, 35 s , True}}