Alpha Zero Symbolic Regression Algorithm

August 28, 2023

1) Create game object

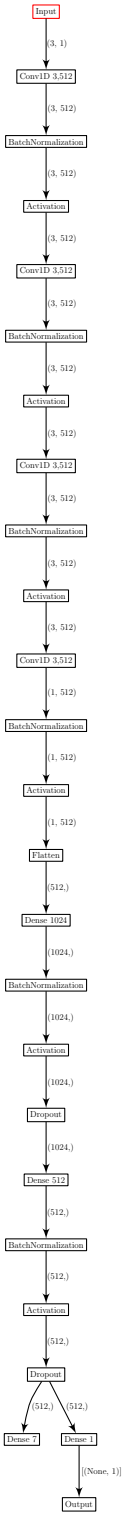
- $\mathbf{n}$ = length of desired expression
- b = board of game
  - → $\mathbf{n}$ pieces (all 0 initially)
  - → possible initial moves
  - → legal moves stemming from all possible previous moves

2) Create neural network wrapper object from game object

- Create symbolic regression neural network
  - → size of input board = $\mathbf{n}$
  - → size of output = # of tokens considered = $\mathbf{o}$
  - → **NN** model, see below example for $\mathbf{n} = 3$ and $\mathbf{o} = 7$

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 3, 1)]	0	[]
conv1d (Conv1D)	(None, 3, 512)	2048	['input_1[0][0]']
batch_normalization (BatchNormalization)	(None, 3, 512)	2048	['conv1d[0][0]']
activation (Activation)	(None, 3, 512)	0	['batch_normalization[0][0]']
conv1d_1 (Conv1D)	(None, 3, 512)	786944	['activation[0][0]']
batch_normalization_1 (BatchNormalization)	(None, 3, 512)	2048	['conv1d_1[0][0]']
activation_1 (Activation)	(None, 3, 512)	0	['batch_normalization_1[0][0]']
conv1d_2 (Conv1D)	(None, 3, 512)	786944	['activation_1[0][0]']
batch_normalization_2 (BatchNormalization)	(None, 3, 512)	2048	['conv1d_2[0][0]']
activation_2 (Activation)	(None, 3, 512)	0	['batch_normalization_2[0][0]']
conv1d_3 (Conv1D)	(None, 1, 512)	786944	['activation_2[0][0]']
batch_normalization_3 (BatchNormalization)	(None, 1, 512)	2048	['conv1d_3[0][0]']
activation_3 (Activation)	(None, 1, 512)	0	['batch_normalization_3[0][0]']
flatten (Flatten)	(None, 512)	0	['activation_3[0][0]']
dense (Dense)	(None, 1024)	525312	['flatten[0][0]']
batch_normalization_4 (BatchNormalization)	(None, 1024)	4096	['dense[0][0]']
activation_4 (Activation)	(None, 1024)	0	['batch_normalization_4[0][0]']
dropout (Dropout)	(None, 1024)	0	['activation_4[0][0]']
dense_1 (Dense)	(None, 512)	524800	['dropout[0][0]']
batch_normalization_5 (BatchNormalization)	(None, 512)	2048	['dense_1[0][0]']
activation_5 (Activation)	(None, 512)	0	['batch_normalization_5[0][0]']
dropout_1 (Dropout)	(None, 512)	0	['activation_5[0][0]']
pi (Dense)	(None, 7)	3591	['dropout_1[0][0]']
v (Dense)	(None, 1)	513	['dropout_1[0][0]']

Total params: 3,431,432
Trainable params: 3,424,264
Non-trainable params: 7,168


Input
(3, 1)
Conv1D 3,512
(3, 512)
BatchNormalization
(3, 512)
Activation
(3, 512)
Conv1D 3,512
(3, 512)
BatchNormalization
(3, 512)
Activation
(3, 512)
Conv1D 3,512
(3, 512)
BatchNormalization
(3, 512)
Activation
(3, 512)
Conv1D 3,512
(1, 512)
BatchNormalization
(1, 512)
Activation
(1, 512)
Flatten
(512,)
Dense 1024
(1024,)
BatchNormalization
(1024,)
Activation
(1024,)
Dropout
(1024,)
Dense 512
(512,)
BatchNormalization
(512,)
Activation
(512,)
Dropout
(512,)
(512,)
Dense 7
Dense 1
[(None, 1)]
Output

3) Create coach object, responsible for executing self.play + learning

- Game Object **(1)**
- Neural Network Object **(2)**
- Arguments
  - → # of iterations
  - → # of episodes per iteration
  - → # of training examples (of the form $[p_i, v, r]$)
  - → # of arena games to determine if new neural network will be accepted
- MCTS object
  - → Game Object **(1)**
  - → Neural Network Object **(2)**
  - → Arguments (same as above)
  - → Dictionaries (all initially empty)
    - $Q(s, a)$: Average value of cumulative rewards received after taking action a from state s; agent's current (at state s) estimate of action a's expected value
    - $N(s, a)$: visit counts for action a at state s
    - $N(s)$: visit counts for state s
    - $P(s)$: estimated vector of prior probabilities of taking actions $(a_1, a_2, \ldots, a_{\mathbf{o}})$ from state s (estimated initially by neural network)
    - $E(s)$: stores end state for state s (-1 = not finished, $0 \leq E(s) \leq 1$ = finished score (0 is the worst possible score and 1 is the best possible score))
    - $V(s)$: vector of 1's and 0's of size $\mathbf{o}$ representing if token i $(0, 1, \ldots, \mathbf{o} - 1)$ is valid (1) or not (0)
- trainExampleHistory (initially empty): list of deques of training examples for 1 episode of the form $[s_i, p_i, r]$
  - → s_i: state (expression list in our case)
  - → p_i: policy vector of size $\mathbf{o}$
  - → r: eventual score of the episode, same for all training examples in trainExampleHistory

- skipFirstSelfPlay: whether or not to skip numEpisodes of initial self-play

4) Load training examples from file if available

5) **Learn!**

A.) For each iteration i $(1, 2, \ldots, \#$ of iterations)

I.) If not skipFirstSelfPlay or $i >= 2$

i.) iterationTrainExamples initialized to empty deque with hard-coded maximum possible length

ii.) For each episode on ith iteration

a.) reinitialize MCTS attribute

b.) execute episode and add result of the form $[s, p, r]$ to iterationTrainExamples, i.e., "execute episode"

execute episode

1.) initialize "trainExamples" as empty list

2.) get initial board (in other words, the initial state s)

3.) set game step = 0

4.) For each game step until game completed, do:

a.) get action prob. vector p for state s using a variable "temp" $= 1$ if episodeStep $<$ tempThreshold else 0, i.e., "Get Action Prob"

Get Action Prob

1.) For a predetermined number of Monte-Carlo simulations

a.) perform a Monte-Carlo tree search (MCTS) for the current state s, i.e., "MCTS search"

MCTS search

1.) If current state s does not have its end state status stored in $E(s)$, store end state status, i.e., "getGameEnded"

getGameEnded(s)

1.) create a copy of the state s (which is a list of integers that encode different tokens)

2.) return -1 (not ended) if the integer 0 is in the list of pieces for the state s

3.) else, get the score for the finished game ($0 \leq$ score ≤ 1), i.e., "is_win" ,and return it

is_win()

1.) Check (again) that the expression list for state s is complete (no zeros). If not, return -1

2.) else

a.) add custom operators (i.e. grad) as sympy "implemented functions"

b.) convert the list of integer pieces into a symbolic expression string

c.) count how many constants are in the expression string

d.) register the input variable (currently only x is supported (univariate) but more can be added in the future)

e.) register symbolic transformations for sympy to help parse expression string

f.) store training features in X (only 1d supported at the moment) and training labels in Y

g.) if the expression string has numConsts > 0

i.) register "const" sympy symbols as y{0:numConsts} and replace each "const" substring with "y{0}", y"{1}", ..., "y{numConsts-1}"

ii.) create a temporary dictionary to map "y{0}", y"{1}", ..., "y{numConsts-1}" strings to sympy symbols

iii.) add custom operators (currently only grad) to temporary dictionary

iv.) attempt to parse expression with temporary dictionary and transformations
- if failed:
  - ⋆ First fail: add input variable x to expression string
  - ⋆ subsequent fails: remove last token from expression string & try to parse again
    - ∘ If string becomes empty, then return a score of 0 (worst possible score)

v.) create a lambda function using input variable x and the constants "y{0}", y"{1}", ..., "y{numConsts-1}"

vi.) try to obtain best-fit values for "y{0}", y"{1}", ..., "y{numConsts-1}"
- if failed: set parameters to random values between 0 and 1

vii.) obtain the predicted labels using the best-fit model and compute the loss

h.) else

i.) create temporary dictionary to map custom operator strings to implemented functions (currently only grad)

ii.) attempt to parse expression with temporary dictionary and transformations
- if failed:
  - ⋆ First fail: add input variable x to expression string
  - ⋆ subsequent fails: remove last token from expression string

& try to parse again

- If string becomes empty, then return a score of 0 (worst possible score)

iii.) create a lambda function using input variable x

iv.) obtain the predicted labels using the model and compute the loss

i.) if loss $<$ current best loss: store corresponding best expression & loss

j.) return score as $\exp\{-0.005 \cdot \text{loss}\}$

2.) if the end state status of state s is status = completed (i.e., not -1, but instead $0 \leq$ score ≤ 1) then return the end-state status (i.e., $0 \leq$ score ≤ 1)

3.) (else) if the current state s does not have a corresponding vector of prior probabilities of taking actions $a_0, a_1, \ldots, a_{\mathbf{o}}$ from state s stored in $P(s)$, then do:

a.) store the neural network's prediction of the vector of prior probabilities and value for the current state s in the dictionary $P(s)$ and temporary variable v, respectively

b.) store the valid moves "valids" for the current state s, i.e., "getValidMoves"

getValidMoves(s)

1.) Copy the current state s

2.) return the legal moves for the current state s, i.e., "getLegalMoves"

getLegalMoves()

1.) If the expression list is complete (meaning it contains no zeros), then return the list of supported operators of size $\mathbf{o}$ (this never happens)

2.) else if the expression list is all zeros (i.e., empty), then return a list of 1's and 0's (1 means operator i in the list of supported operators of size **o** is legal, and 0 means operator i is illegal)

3.) else

a.) get the index i of the current move (i.e., token) that has to be decided upon

b.) return a list of 1's and 0's of size **o** indicating whether the operator i is legal (1) or illegal (0) from move $i-1$; this is obtained from a hard-coded dictionary

c.) the corresponding vector of prior probabilities for state s is stored in $P(s)$ as the dot-product of the neural network's prediction and the binary list of valid moves. Therefore, even if the neural network predicts a non-zero probability for operator i given state s, if it's not valid (i.e. 0) then the actual probability is automatically 0

d.) store the sum of the prior probability vector (obtained in the previous step c.)) in a variable called "sum_Ps_s"

e.) if sum_Ps_s > 0, then divide all the probabilities in $P(s)$ by sum_Ps_s

f.) else (in practice, when $P(s)$ is a 0-vector of size **o**)

i.)

$$P(s) = \underbrace{P(s)}_{\text{just all 0's}} + \overbrace{\text{valids}}^{\text{list of 1's (valid) and 0's (not-valid) for move } i}$$

ii.)

$$P(s) = \frac{P(s)}{\text{sum(P(s))}} \qquad \text{(for normalization)}$$

g.) store the list of 1's and 0's for if move i is valid (1) or not (0) for state s in the dictionary $V(s)$

h.) store the visit count for state s as 0 in the dictionary $N(s)$

i.) return the variable v (i.e., the value (i.e., the cumulative reward))

4.) (else)

a.) store the list of 1's and 0's for if move i is valid (1) or not (0) that's in the dictionary $V(s)$ in a temporary variable called "valids"

b.) search for the index of the action a with the highest upper-confidence bound (u.c.b.) u

i.) initialise the highest u.c.b. "cur_best" to $-\infty$ and the corresponding index of the best action "best_act" to -1

ii.) for index a in $0, 1, \ldots, \mathbf{o} - 1$

– if action a is valid (i.e., stored as 1 in position a of list "valids")

⋆

$$u = \begin{cases} Q(s,a) + c \cdot P(s,a) \cdot \sqrt{\frac{N(s)}{1+N(s,a)}} & \text{if } Q(s,a) \text{ exists} \\ c \cdot P(s,a) \cdot \sqrt{N(s) + \epsilon} & \text{otherwise} \end{cases}$$

$Q(s,a) =$ estimated value of a given s

$c =$ exploration-exploitation trade-off

$P(s,a) =$ est. prior prob. of taking action a from s

$N(s) =$ visit counts for state s

$N(s,a) =$ visit counts for action a at state s

$\epsilon =$ just a small, hard-coded number

- if $u >$ cur_best:
  - ⋆ cur_best $= u$
  - ⋆ best_act $= a$

iii.) $a =$ best_act

iv.) store the next state given the current state s and the index of the best action a in a variable "next_s," i.e., computed by "getNextState"

getNextState(s, a)

1.) If the action index a is equal to the number of supported operators **o** (i.e., 1 past the last element) then return the state s

2.) else

a.) copy the state s

b.) store the action corresponding to index a in a temporary variable called "move"

c.) assign the first 0-element of the current pieces of the copy of state s the move "move"

d.) return the updated pieces (i.e., the current state expression list)

v.) repeat the MCTS search routine given the "next_s"' state until the recursion reaches the base case where the cumulative reward v is returned, and assign this to a temporary variable called v

vi.) If the current state s and the corresponding index of the current best action a are store in the dictionary $Q(s, a)$ (which, as a reminder, is the agent's prediction of the value of action a from state s), then

–

$$Q(s,a) = \frac{N(s,a) \cdot Q(s,a) + v}{N(s,a) + 1}$$
$$N(s,a) = N(s,a) + 1$$

where:

$Q(s,a)$ = estimated value of a given s
$N(s,a)$ = visit counts for action a at state s
v = neural network prediction of cum. reward

vii.) else

–

$$Q(s,a) = v$$
$$N(s,a) = 1$$

where:

$Q(s,a)$ = estimated value of a given s
$N(s,a)$ = visit counts for action a at state s
v = neural network prediction of cum. reward

viii.) $N(s) = N(s) + 1$, where $N(s)$ is the number of visit counts for state s

ix.) return v

2.) Store the visit counts for action a at state s (obtained from the $N(s,a)$ dictionary) for all possible states **o** in a temporary list "counts" (so "counts" is therefore also a list of size **o**)

3.) If the variable "temp" is 0

a.) Store the indices corresponding to the most visited actions from state s in a temporary variable "bestAs"

b.) Store a randomly selected index from "bestAs" in a temporary variable called "bestA"

c.) Initialize a temporary size-**o** list "probs" with 0's

d.) assign 1 to the bestA'th element of "probs"

e.) return "probs"

4.) else

a.) counts = counts$^{1/\text{temp}}$

b.) create a temporary variable called "counts_sum" that stores the sum of "counts"

c.) create a temporary variable "probs" that stores $\frac{\text{counts}}{\text{counts\_sum}}$

d.) return "probs"

b.) append a list consisting of the current state s and the action probability vector p in the trainExamples list

c.) store a random index of a possible action weighted by p in a temporary variable called "action"

d.) get the next state (i.e. getNextState(s, a), see above) given the current state s and the chosen action "action" and then update the current state s with this next state

e.) check if the game has ended (i.e. getGameEnded(s), see above), i.e., get the score and store it in a temporary variable called "r"

f.) if r is not equal to -1 (meaning the episode is complete and thus r is the score of the episode, i.e., $0 \leq r \leq 1$), then append r to every element of the trainExamples list, and finally, return trainExamples

iii.) append iterationTrainExamples to the current "trainExamplesHistory"

II.) if the size of the current "trainExamplesHistory" is greater than the desired length (which is hard-coded), then remove the oldest entry from "trainExamplesHistory"

III.) save the training examples (i.e., "trainExamplesHistory") of the i'th iteration to a file

IV.) copy trainExamplesHistory to a temporary variable called "trainExamples"

V.) shuffle "trainExamples"

VI.) save the weights of the current neural network to a file

VII.) Train the neural network with "trainExamples," where the neural network uses the state s as x (features) and the probability vectors and target values as y (labels)

VIII.) Arena tournament

i.) Create a MCTS object with the game attribute of the current Coach object, the current neural network, and the configuration arguments as arguments

ii.) Create an Arena object with the game attribute of the Coach object, and a player, i.e., a function that uses the MCTS object to always choose the action with the highest probability (i.e., via argmax(Get Action Prob), see above)

iii.) have this current neural network play "arenaCompare" number of games. If the average score is better than the current best score, then save this current neural network's weights, else just save the weights of the best, previous neural network's weights to a checkpoint file.