Backpropagation-Free Parallel Deep Reinforcement Learning

William H. Guss
Machine Learning at Berkeley
2650 Durant Ave, Berkeley CA, 94720
wguss@ml.berkeley.edu

Mike Zhong
Machine Learning at Berkeley
Berkeley CA, 94720
mlyzhong@berkeley.edu

Utkarsh S
Machine Learning at Berkeley
Berkeley CA, 94720
philkuz@ml.berkeley.edu

Max Johansen
Machine Learning at Berkeley
Berkeley CA, 94720
max@ml.berkeley.edu

Abstract

In this paper we conjecture that an agent, envirionment pair (π, E) trained using DDPG with an actor network μ and critic network Q^π can be decomposed into a number of sub-agent, sub-environment pairs (π_n, E_n) ranging over every neuron in μ; that is, we show empircially that treating each neuron n as an agent $\pi_n : \mathbb{R}^n \to \mathbb{R}$ of its inputs and optimizing a value function Q^{π_n} with respect to the weights of π_n is dual to optimizing Q^π with respect to the weights of μ. Finally we propose a learning rule which simultaneously optimizes each π_n without error backpropogation achieving state of the art performance and speed across a variety of OpenAI Gym environments.

Todo list

1 Introduction

Introduction to DDPG and recent advances in deep RL.

Biological diffusion of dopamine in the brain $\implies$ error backpropagation is not biologically feasible.

Synthetic gradients are a step in the right direction, but still require eventual back propogation.

Therefore it is feasible that each neuron is maximizing the expectation on his future dopamine intake, and so we propose the following theorem.

2 Agent-Environment Value Decomposition

A high level description of the section.

Fix the time thing. Remove ℓ.

2.1 Background

Recall the standard reinforcement learning setup. We say E is an *environment* if $E \stackrel{\text{def}}{=} (\mathcal{S}, \mathcal{A}, \mathcal{R}, T, r)$ where T describes transition probability measure $T\left(s_{t+1} \mid s_t, a_t\right)$ and $r : \mathcal{S} \times \mathcal{A} \rightarrow$

$\mathcal{R}$ is a reward function. Furthermore $\mathcal{S}$, $\mathcal{A}$, $\mathcal{R}$ are the *state space, action space, and reward space* respectively. We restrict $\mathcal{R}$ to a compact subset of $\mathbb{R}$ and action space and state space to finite dimensional real vector spaces. As in DDPG we assume that the environment E is *fully observed*

FIX THIS WHEN FIXING ℓ. The environment is NOT fully observed.

; that is, at any time step the state s_t is fully described by the observation presented, x_t, and not by the history $(x_1, a_1, \dots, a_{t-1})$.

We define the policy for an agent to be $\pi : \mathcal{P}(\mathcal{A}) \times \mathcal{S} \to [0, 1]$. In general the policy is a probability measure on some σ-algebra $\mathcal{M} \subset \mathcal{P}(\mathcal{A})$ conditioned on $\mathcal{S}$ so that $\pi(\mathcal{A} \mid s \in \mathcal{S}) = 1$. However, we will deal only with *deterministic* policies where for every s_t there is unique a_t so that $\pi(\{a_t\} \mid s = s_t) = 1$ and the measure is 0 otherwise. Thus we will abuse notation and define a *deterministic agent* by a policy function $\pi : \mathcal{S} \to \mathcal{A}$. Additionally we denote the state-space trajectories of π by

$$\Gamma_\pi(\mathcal{S}) = \{(s_1, s_2, \dots) \mid s_1 \sim T(s_0), s_{t+1} \sim T(s \mid s_t, \pi(s_t))\}. \tag{2.1.1}$$

For a policy π the action-value function is the expected future reward under π by performing a_t at state s_t using the Bellman equation

$$Q^\pi(s_t, a_t) = \mathop{\mathbb{E}}_{s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1}))] \tag{2.1.2}$$

with $\gamma \in (0, 1)$ a discount factor, and the second expectation removed because π is deterministic. [Some survey] provides an extensive exposition into a justification of this equation and choice for the action-value of π, so we will assume such a choice is a valid measure of performance.

In deterministic policy gradient methods, we define an actor $\mu : \mathcal{S} \to \mathcal{A}$ and a critic $Q^\mu : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and optimize $Q^\mu(s_t, \mu(s_t|\theta^\mu))$ with respect to the paramers θ^μ of μ. This method is provably the true policy gradient of μ if Q^μ is known. Recently (DDPG) utilizes the universality of DNNs in order to approximate both μ and Q^μ along with delayed weight-transfer networks to stabilize learning and prevent divergence as depicted in Figure 1. In order to decompose the action-value function we will make heavy use of this methodology at a scale local to each neuron in the flavor of (Synthetic gradients.)

Insert reference and make this a footnote

Cite lillicrap

Make DDPG figure

Cite deepmind

2.2 Towards Neurocomputational Decomposition of Q^μ

In order to decompose the Q^μ algorithm we will abstractly define a neurocomputational agent in terms of an operator on voltages with no restrictions on the topology of the network, and then relate the action-value function of the whole agent to those which are defined for each individual neuron in the network.

If $\mathcal{V}$ is an N-dimensional vector space then a *neurocomputational agent* is a tuple $\mathcal{N} = (\mu, \epsilon, \delta, K, \Theta, \sigma, D)$ such that:

- $\epsilon : \mathcal{S} \to \mathcal{V}$ encodes the state into the voltages. Realistically, only a subset of all neurons are input neurons, denoted as $N_I \subset \mathcal{V}$, so $\epsilon(s_t) = proj_{N_I}(\epsilon(s_t))$.
- $\delta : \mathcal{V} \to \mathcal{A}$ decodes the voltages of the *output neurons* $N_O \subset V$ into an action, so that $\delta(v_t) = \delta(proj_{N_O}(v_t))$.
- $K : \mathcal{V} \to \mathcal{V}$ is the linear voltage graph transition function of the graph representing the topology of $\mathcal{N}$, parameterized by θ.
- $\Theta : \mathcal{V} \to \mathcal{V}$ is a nonlinear inhibition function.
- $\sigma : \mathcal{V} \to \mathcal{V}$ is the elementwise application of some activation function to the voltage vector.
- $D : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ is called voltage dynamic of $\mathcal{N}$ such that

$$v_{cur+1} \stackrel{\text{def}}{=} D(v_{cur}, v_{in}) \stackrel{\text{def}}{=} \sigma(\Theta K[v_{cur}]) + v_{in} \tag{2.2.1}$$

  where v_{cur} is the internal voltage vector of $\mathcal{N}$ and v_{in} is an input voltage. We will occasionally abuse notation an say that $D(v_{cur}) = D(v_{cur}, 0)$ when v_{in} is 0.

- $\mu : \mathcal{S} \to \mathcal{A}$ is the deterministic policy for $\mathcal{N}$. For some agents, the internal time τ is not in sync with the discrete time step t of E. Therefore for every t there is an evaluation delay ℓ so that

$$\begin{aligned} v_{t+1} &= D(v_t) + \epsilon(s_t) \\ &= D(v_{t-1}, \epsilon(s_t)) \\ \mu(s_t) &= \delta(v_t) \end{aligned} \tag{2.2.2}$$

It is not hard to see that this definition encompasses any DQN or DDPG network with either reccurent or non recurrent layers. Additionally other paradigms such as the leaky integrattor are neurocomputational agents. (See appendix.)

If n is some neuron in $\mathcal{N}$, we say $E^n = (\mathcal{S}^n, \mathcal{A}^n, \mathcal{R}^n, T^n, r^n)$ is deterministic *sub-environment* of E with respect to $\mathcal{N}$ if it has the following properties:

- an environment $\mathcal{S}^n = \mathcal{V}$, in that the environment that neuron n sees is the voltages of all other neurons. We assume that the graph is fully connected so that all neurons are connected to all other neurons. Realistically, each neuron only sees the voltages of a subset of all neurons. We will deal with this in a further iteration of the paper.
- an action $\mathcal{A}^n = \mathbb{R}$ denotes the output voltage of neuron n one time step later, i.e. $\alpha_t^n = v_{t+1}^n$.
- a reward $\mathcal{R}^n = \mathcal{R}$, denoting the reward that neuron n receives. It is the same as the reward that the entire agent sees.
- a transition function $T^n : \mathcal{S}^n \times \mathcal{A}^n \to \mathcal{S}^n$ such that $T^n(v_t, \alpha_t^n) = (I - \delta_{n,n})D(v_t, \epsilon(s_t)) + e_n \alpha_t^n$
- and a reward function $r^n(v_\tau, \alpha) = r(s_t, \mu(s_t))$ if $I(\tau) \neq 0$ and 0 otherwise. Essentially the state space of E^n at time step $\tau + 1$ is just the normal dynamics of $\mathcal{N}$ applied to the previous state along with a possible encoded input state $\epsilon(s_t)$ from E except for at neuron n. Lastly an agent $\mu^n : \mathcal{V} \to \mathbb{R}$ is called *neuromorphically local* to $\mathcal{N}$ if $v_\tau \mapsto \langle D(v_\tau), e_n \rangle$.

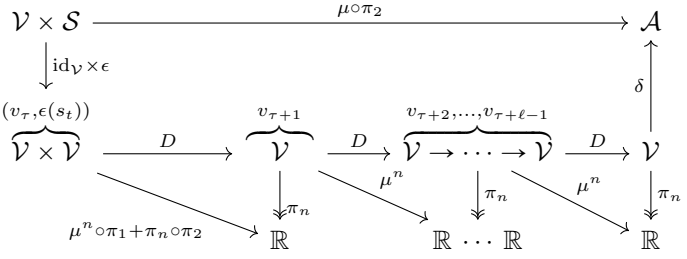
We now can think of every neuron in $\mathcal{N}$ as an agent in its own environment, acting on its inputs, and we can extend the action-value definition to μ^n as follows

$$Q^{\mu^n}(v_\tau, \alpha_\tau) = \mathop{\mathbb{E}}_{v_{\tau+1} \sim E^n} \left[r^n(v_\tau, a_\tau) + \gamma Q^{\mu^n}(v_{\tau+1}, \mu^n(v_{\tau+1})) \right]. \tag{2.2.3}$$

2.2.1 Results

Provided with the previous definitions, the following question arises: does deterministic policy gradient learning on $\mathcal{N}$, specifically μ on E, *commute* with performing the same operation simultaneously on every neuromorphically local agent μ^n comprising $\mathcal{N}$ and their respective sub-environments E^n? Supposing that we have the true Q^μ function and μ is optimal with respect to Q^μ, then it is intuitive, but not obvious, that every μ^n should behave optimally with respect to an infinite time horizon – but will the reverse hold? We give the following results:

Theorem 2.2.1. *Let E and $\mathcal{N}$ be defined as before. Then for every $n \in \mathcal{N}$, it follows that $\Gamma_\mu(\mathcal{S})$ is equal to $\Gamma_{\mu^n}(\mathcal{V})$ up to bijection and the following diagram commutes.*



(2.2.4)

Proof. We first show that (2.2.4) commutes. Let $v \in \mathcal{V}$ and $s \in \mathcal{S}$. Observe that $\mu(\mathcal{S}) =$ is clear that $V[b,c] \circ V[a,b] = V[a,c]$ by (**??**), so the upper part of the diagram is equivalent to

$$\begin{array}{ccc} \mathcal{S} & \xrightarrow{\mu} & \mathcal{A} \\ \downarrow{\scriptstyle \epsilon} & & \uparrow{\scriptstyle \delta} \\ \mathcal{V} & \xrightarrow{V[\tau,\ell]} & \mathcal{V} \end{array} \tag{2.2.5}$$

and by definition μ with an evaluation time ℓ we have that

$$\begin{aligned} \mu(s_t) &= \delta(V[\tau+\ell](\sigma\left(K\Theta[V(\tau)]\right) + \epsilon(s_t))) \\ &= \delta(V[\tau+\ell](V(\tau))). \end{aligned} \tag{2.2.6}$$

fix this equation, ϵ doesn't directly commute

Next for each $V[\tau+k,1]$, $k \in \mathbb{N} \cup \{0\}$ observe the cooresponding trangle in the diagram. When π_n is the cannonical projection, we have

$$\left(\pi_n \circ V[\tau+k,1]\right)(v_\tau) = \langle V[\tau+k+1](v), e_n \rangle \tag{2.2.7}$$

and by (**??**)

□

Theorem 2.2.2. *If $\mathcal{N}$ is a nuerocomputational agent in E and for every $n \in \mathcal{N}$ there is a nueromorphically local agent μ^n in sub-environment E^n, then policy gradient for μ agrees with the simultaneous policy gradients of every neuromorphically local agent; that is*

$$\prod_{n=1}^{N} \nabla_{K^n} Q^{\mu^n}(v,a)\Big|_{v=v_t, a=\mu^n(v_t)} = \nabla_K Q^{\mu}(s,a)\Big|_{s=s_t, a=\mu(s_t)} \tag{2.2.8}$$

for every time step t, where K^n represents the nth column of the linear voltage graph transition function, i.e. the weights of the connections from all neurons to neuron n.

Suggestion (from Mike):

If $\mathcal{N}$ is a nuerocomputational agent in E and for every $n \in \mathcal{N}$ there is a nueromorphically local agent μ^n in sub-environment E^n, then policy gradient for μ agrees with the simultaneous policy gradients of every neuromorphically local agent; that is

$$\forall n, \nabla_{K^n} Q^{\mu^n}(v,a)\Big|_{v=v_t, a=\mu^n(v_t)} = \nabla_{K^n} Q^{\mu}(s,a)\Big|_{s=s_t, a=\mu(s_t)} \tag{2.2.9}$$

for every time step t, where K^n represents the nth column of the linear voltage graph transition function, i.e. the weights of the connections from all neurons to neuron n.

Proof. Prove the Decomposition

□

Emperical justification of the iff using the following experiment (s).

Intuitively, it should be the case that by treating each neuron in the neural network as its own Q-learner, the policy gradient of an individual agent should be the same as for the policy gradient entire agent, if each neuron sees the same reward r_t as the entire agent. In other words, the optimal way of updating the entire connection matrix K should be the optimal way for updating the weights connected to each neuron w.r.t. each neuron as its own Q-learner, given the same rewards.

3. Plot the output of both Q^μ and Qn using TENSORFLOW summaries.

From Experiment 1, we see that there is a correlation between the Q-gradient for neuron n, and the nth column of the gradient of the agent's Q function:

from strong to weak:

$$\nabla_{K^n} Q^{\mu^n}(v,a) = \nabla_{K^n} Q^{\mu}(s,a)$$

$$\nabla_{K^n} Q^{\mu^n}(v,a) \propto \nabla_{K^n} Q^{\mu}(s,a)$$
$$corr[\nabla_{K^n} Q^{\mu^n}(v,a), \nabla_{K^n} Q^{\mu}(s,a)] \approx EMPERICALVALUE$$

although (assuming we get a weak experimental result), we propose that it is due to our neurons not being the ideal Q-learner (convergence issues, etc.).

1. Training a network on Atari using DDPG and plotting average critic functions for neurons using window.

EXPERIMENT 1 SPECIFICATION. 1. Set up a standard DDPG to play the set of atari games in OpenAi Gym using TENSORFLOW (this will be mac,linux, or windows bash only). If on Windows bash install Xming (its an X server) and run all OpenAI Gym commands with `DISPLAY=localhost:0.0 python3 src/experiment1/some__script_in_src.py`. This will pipe the visual output fo the OpenAI Gym simulators to the display. If you cannot get this to work on your screen, do not do env.render. We can also stop using the atari games, since this works for a fact on the basic Box2d versions. **We are going to write all of this in Python3, make sure to install gym in python3.**
2. DDPG has a Q^{μ} network which we use to optimize $\mu(s_t \mid \theta)$ with respect to θ. The goal of this experiment is to train a standard DDPG network to play one of these OpenAI Gym simulations, whislt concurrently estimating and viewing the Q functions for every single neuron. THEREFORE, we need to select a subset of neurons in the fully connected layers (for example) of the μ network (actor) and concurrently train a network $Q^n(s,a)$ to estimate the Q function of the neuron based on its inputs s and its SINGLE output voltage s. This can be a 3 layer fully connected network with $|s|+1$ inputs (one for each input the neuron n and 1 for the voltage of the neuron after receiving that output.) Tensorflow is a dataflow language so the output of a layer looks like $O2 = \sigma(W * O1)$. Therefore you just need to make another "network" whose dataflow could be like $Qn = \sigma(W_3^n * \sigma(W_2^n * \sigma(W_1^n * \text{concat}(O1, O2[i]))))$ where n is the ith neuron on layer $O2$. And $O1, O2, O3$ are the outputs of the neurons on those layers in the network. Then as in standard DQN you tain Qn with a lag network $W(Qn') = W(Qn)(1-\tau) + \tau W(Qn)$ where W denotes the weights of Qn, say $W_3^n, W_2^n, W_1^n, \ldots$. And then actually do gradient decent on the weights of Qn not Qn' by minimizing the following bellman equation

$$L(s_t, a_t, r_t, s_{t+1}, a_{t+1}) = (Qn(O1(s_t), O2[i](s_t)) - r_t - Qn'(O1(s_{t+1}), O2[i](s_{t+1})))^2$$

with respect to the parameters $W(Qn)$. Note I didn't actually use a_t above since really the Qn function takes in the input of the previous layer (to n) as its input, say $O1(s_t)$ and the action for that same time step which is just the output of the neuron n, say $O2(s_t)[i]$. The same goes for Qn' but at the next time step.

3 Experimentation

3.1 Experiment 1: Learning Q functions on components of the Actor network.

3.2 Experiment 2: Treating each neuron as its Actor-Critic network using linear approximators

Now that we have established that the Critic networks for each of the individual neuron learns the same Q-function as does the entire agent, we now treat each neuron as its own actor. Each neuron changes the weights of its presynaptic neurons' connections, as parameters for its actor – its voltage on the next timestep – to optimize its learned approximated Q function. We use the linear approximation:

$$Q^n(v,a) \approx \theta_{n,v}^T v + \theta_{n,a} a = \theta_n^T (v,a)^T$$
$$\mu^n(v) = \sigma(K^n v)$$

GOT RID OF Θ in that it's not very necessary ...

The algorithm for learning (very much INSPIRED by [CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING]):

For each neuron n:

Initialize random weights $\theta_{n,v}$, $\theta_{n,a}$, and K^n

Initialize target network $Q^{n\prime}$ and $\mu^{n\prime}$ with same weights, $\theta'_{n,v}$, $\theta'_{n,a}$, and K'^n

Initialize replay buffer R

For each episode:

Receive initial observation voltages v_1

for t=1, T, do:

follow dynamics, $a^n_{t+1} = \sigma(K^n v_t)$

Observe reward r_t, observe new voltages v_{t+1}

Store transition (v_t, a_t, r_t, v_{t+1}) in R.

Sample a random minibatch of N transitions (v_i, a_i, r_i, v_{i+1}) from R

Set $y_i = r_i + \gamma Q'(v_{i+1}, \mu'(v_{i+1}))$

Update critic weights θ_n by minimizing loss $L := \frac{1}{N}\sum_i (y_i - Q(v_i, a_i))^2$

Update actor weights (connections) using the sample policy gradient:

$$\nabla_{K^n} J \approx \frac{1}{N}\sum_i \nabla_a Q(v,a)|_{v=v_i, a=\mu^n(v_i)} \nabla_{K^n}\mu^n(v)|_{v_i}$$

update the target networks:

$$\theta'_n \leftarrow \tau\theta_n + (1-\tau)\theta'_n$$
$$K^{n\prime} \leftarrow \tau K^n + (1-\tau)K^{n\prime}$$

end for

end for

We train the critic weights $(\theta_{n,v}, \theta_{n,a})$ by minimizing the loss:

$$L = \frac{1}{2}((r_i + \gamma Q^{n\prime}(v_{i+1}, a_{i+1})) - Q(v_i, a_i)) = \frac{1}{2}D_i^2$$

where $D_i := (r_i + \gamma\theta_n^{T\prime}(v_{i+1}, a_{i+1})) - \theta_n(v_i, a_i)$

We thus have:

$$\nabla_{\theta_n} L = -D_i(v_i, a_i)$$

or, using gradient descent with learning rate η_Q, we have:

$$\theta_n \mathrel{-}= \eta_Q \nabla_{\theta_n} L = -\eta_Q D_i(v_i, a_i)$$

or:

$$\theta_{n,v} \mathrel{+}= \eta_Q D_i v_i$$

and

$$\theta_{n,a} \mathrel{+}= \eta_Q D_i a_i$$

As for updating the actor policy, with learning rate eta_A, we have:

$$K^n \mathrel{+}= \eta_A \nabla_a Q(v,a)|_{v=v_i, a=\mu^n(v_i)} \nabla_{K^n}\mu^n(v)|_{v_i} = \eta_A \theta_{n,a}\sigma'(K^n v_i)v_i$$

IN CONCLUSION:

$$\theta_{n,v} \mathrel{+}= \frac{1}{N}\sum_i \eta_Q D_i v_i$$

$$\theta_{n,a} \mathrel{+}= \frac{1}{N}\sum_i \eta_Q D_i a_i$$

$$K^n \mathrel{+}= \frac{1}{N}\sum_i \eta_A \theta_{n,a} \sigma'(K^n v_i) v_i$$

where $D_i := (r_i + \gamma \theta_n^{T\prime}(v_{i+1}, a_{i+1})) - \theta_n(v_i, a_i)$

Therefore we propose the following learning rule in aims to evidence the reverse, training μ using simultaneous optimization on all Q_n w.r.t π_n's weights.

4 Decentralized Deep Determinstic Policy Gradient Learning

Proposal of the rule. Linear approximation of the Q function for every neuron is good enough, (experimentally).

Implications of the rule to DDPG

Implications of the rule to entirely recurrent networks (infinite time horizion and NO unrolling since the environment the local actions of the neuron which globally recur to that neuron again are *encoded* into Q_n; large time horizion probably implies that better regresser needed for Q_n.)

Parallelism, no error backprop, and only 2x operations, but no locking on GPU, so all can be run sumultaneously if we cache!

5 Results

To validate the new learning rule we throw a fuck ton of experiments together on the following list (or better using OpenAI Gym).

```
blockworld1 1.156 1.511 0.466 1.299 -0.080 1.260
blockworld3da 0.340 0.705 0.889 2.225 -0.139 0.658
canada 0.303 1.735 0.176 0.688 0.125 1.157
canada2d 0.400 0.978 -0.285 0.119 -0.045 0.701
cart 0.938 1.336 1.096 1.258 0.343 1.216
cartpole 0.844 1.115 0.482 1.138 0.244 0.755
cartpoleBalance 0.951 1.000 0.335 0.996 -0.468 0.528
cartpoleParallelDouble 0.549 0.900 0.188 0.323 0.197 0.572
cartpoleSerialDouble 0.272 0.719 0.195 0.642 0.143 0.701
cartpoleSerialTriple 0.736 0.946 0.412 0.427 0.583 0.942
cheetah 0.903 1.206 0.457 0.792 -0.008 0.425
fixedReacher 0.849 1.021 0.693 0.981 0.259 0.927
fixedReacherDouble 0.924 0.996 0.872 0.943 0.290 0.995
fixedReacherSingle 0.954 1.000 0.827 0.995 0.620 0.999
gripper 0.655 0.972 0.406 0.790 0.461 0.816
gripperRandom 0.618 0.937 0.082 0.791 0.557 0.808
hardCheetah 1.311 1.990 1.204 1.431 -0.031 1.411
hopper 0.676 0.936 0.112 0.924 0.078 0.917
hyq 0.416 0.722 0.234 0.672 0.198 0.618
movingGripper 0.474 0.936 0.480 0.644 0.416 0.805
pendulum 0.946 1.021 0.663 1.055 0.099 0.951
reacher 0.720 0.987 0.194 0.878 0.231 0.953
reacher3daFixedTarget 0.585 0.943 0.453 0.922 0.204 0.631
reacher3daRandomTarget 0.467 0.739 0.374 0.735 -0.046 0.158
reacherSingle 0.981 1.102 1.000 1.083 1.010 1.083
walker2d 0.705 1.573 0.944 1.476 0.393 1.397
```

2. Show that training decentralized policy gradient $\implies$ total policy optimization

3. Show speed improvements on update step through parallelism (samples per second vs DDPG).

4. Show results are comparable with the state of the art.

6 Conclusion

We wrecked deep reinforcement learning using biological inspiration.

6.1 Future Work

Would like to try the method with full recurrent networks and purely asynchronous implementation of leaky integration networks.

Would like to prove the conjecture. List possible methods of proof.