Episodic Learning

Dennis Kibler
Bruce Porter

Information and Computer Science Department
University of California at Irvine
Irvine, California[1]

Abstract

A system is described which learns to compose sequences of operators into episodes for problem solving. The system incrementally learns when and why operators are applied. Episodes are segmented so that they are generalizable and reusable. The idea of augmenting the instance language with higher level concepts is introduced. The technique of perturbation is described for discovering the essential features for a rule with minimal teacher guidance. The approach is applied to the domain of solving simultaneous linear equations.

1. Introduction

With the aid of a teacher, junior high school students can learn to solve simultaneous linear equations. Operators that are applied in solving these problems include multiplying an equation by a constant and combining like terms. The students are already familiar with how these operators are applied. Moreover, the teacher assumes that the students understand basic concepts about numbers, such as a number being positive, negative, or non-zero.

Our system, nicknamed PET,[2] incrementally (defined in [7]) induces correct rules from the training instances presented. The rules are correct in the sense that at any point in the learning process:

- the knowledge is consistent with all past training instances.
- sequences of rules (episodes) are guaranteed to simplify the problem state if they apply.

Learning rules for applying operators involves two stages of learning:

- Stage 1 learning involves understanding when each available operator should be applied. The concern here is with learning the enabling conditions for individual operators, without knowledge of the other operators in the solution path to provide context.
- Stage 2 learning involves understanding why each operator is applied with emphasis on the sequencing of operators. We refer to this as episodic learning. Episodic segmentation is the grouping of operators to form an episode. Episodes are discrete, reusable components for plan generation and each simplifies the problem state.

The main features of our approach to episodic learning are:

- segmentation of operator sequences into meaningful, re-usable episodes.
- augmentation of the instance language to include higher-order concepts not present in the training instance itself.
- perturbation of a training instance to create new instances.

2. Related Work

Related work in stage 1 learning includes Neves's [10] system which learned to solve one equation in one unknown from textbook traces. The system learned both the context (preconditions) of an operator as well as which operator was applied, although the operator had to be known to the system. His generalization language was simpler than ours in that a constant could only be generalized to a variable. The program LEX [8] uses version spaces to describe the current hypothesis space as well as concept trees to direct or bias the generalizations. As it is not the main point of our work, we keep only the minimal (maximally specific) generalization [11] of the examples.

MACROPS [2] is an example of stage 2, or episodic learning. This system remembers robot plans that have been generated so that the plan can be reused without re-generation. The plans are stored in triangle tables which record the

[1]This research was supported by the Naval Ocean System Center under contract N00123-81-C-1165.

[2]Please refer to [4] for a complete description of our approach including a PDL description of the learning algorithm and multiple examples of its use. PET is implemented in Prolog on Dec2020. Available upon request.

order of application of operators in the plan and how their pre-conditions are satisfied. The plans are generalized to be applicable to other instances (as are episodes).

While effective in learning plans, MACROPS has difficulty applying its acquired knowledge [1]. The central problem is that the operators in a MACROPS plan are not segmented into meaningful sequences. Any sequence of operators can be extracted from the triangle table and re-used as a macro operator. A sequence of length N defines N(N-1)/2 macros. However, few of these sequences are useful. MACROPS offers no assistance in selecting the useful sequences from a plan. If sequences are not extracted from the triangle table then the entire plan must be considered an episode. This results in a large collection of opaque, single-purpose, macro operators. Branching within an episode is made impossible. In either case, combinatorial explosion makes planning with the macros impractical.[3]

3. Operators for Solving Linear Equations

The operators applicable to solving simultaneous linear equations are described in the following table:

Operator	Semantics
combinex(Eq)	Combine x-terms in Eq.
combiney(Eq)	Combine y-terms in Eq.
combinec(Eq)	Combine constant terms in Eq.
deletezero(Eq)	Delete term with 0 coeff. or 0 constant from Eq.
sub(Eq1,Eq2)	Replace Eq2 by the result of subtracting Eq1 from Eq2
add(Eq1,Eq2)	Replace Eq2 by the result of adding Eq1 and Eq2
mult(Eq,N)	Replace Eq by the result of multiplying Eq by N

4. Description Languages

4.1. Instance Language

The instance language serves as "internal form" for training instances. We adopt a relational description of each equation, so the training instance:

a: 2x-5y=-1
b: 3x+4y=10

is stored as:

```
{term(a,2*x),term(a,-5*y),term(a,1),
 term(b,3*x),term(b,4*y),term(b,-10)}
```

where a,b are equation labels and x,y are variables in the instance language.

4.2. Generalization Language

Following Mitchell [8] and Michalski [6] we have concept trees for integers, equation labels, and variables (figure 4-1). Basically we are using the typed variables of Michalski [6].

```
                 integer              eqn     variable
                /       \             / \       / \
        non-zero          zero       a   b     x   y
        /      \             \
  positive   negative         0
   / | \       / |
  1  2 3 ...  -1 -2 ...
```

Figure 4-1: Concept trees

We permit generalizations by 1) deleting conditions, 2) replacing constants by variables (typed), and 3) climbing tree generalization. Disjunctive generalization is allowed by adding additional rules. This covers all the generalization rules discussed by Michalski [6] except for closed interval generalization.

4.3. Rule Language

Knowledge is encoded in rules which suggest operators to apply. Rules are of the form:

```
<score> -- <bag of terms expressed in
  generalization language> => <operator>
```

(The score of a rule is described in section 6.2.) PET does not learn "negative" rules to prune the search tree, as in [3].

5. Perturbation

Perturbation is a technique for stage 1 learning which enables a learning system to discover the essential features of a rule with minimal teacher involvement. A perturbation of a training instance is created by:

- deleting a feature of the instance to determine whether its presence is essential.

- if a feature is essential, modifying it slightly to determine if it can be generalized. Perturbation operators, which are added to the concept tree used for generalization, make these minor modifications.

For example, given the problem (a) 2x+3y=7 (b) 2x+3x-5y=5, the advice to combinex(b), and an empty rule base, PET first describes the rule as:

```
{term(a,2*x),term(a,3*y),term(a,-7),
 term(b,2*x),term(b,3*x),term(b,-5*y),
 term(b,-5)} => combinex(b).
```

Now PET perturbs the instance by modifying each of the coefficients individually. This is done by zeroing, incrementing and decrementing each coefficient. Some of the instances created by perturbation are:

[3]It should be recognized that MACROPS was designed to control a physical robot, not a simulation. For this reason, the designers thought it important to permit the planner to skip ahead in a plan if situation permits or to repeat a step in a plan if the operation failed due to physical difficulties.

(i)	(ii)	(iii)
3y=7	2x+3y=7	2x+3y=7
2x+3x-5y=5	3x-5y=5	2x+4x-5y=5

Since combinex(b) is effective in example i, PET generalizes (minimally) its current rule conditions with this example yielding the new rule:

```
{term(a,3*y),term(a,-7),
 term(b,2*x),term(b,3*x),term(b,-5*y),
 term(b,-5)} => combinex(b).
```

The major effect is to delete the condition on the x-term of equation(a).

The operator is not effective for example ii, and the negative information is (in the current system) unused. Generalizing with example iii, the rule becomes:

```
{term(a,3*y),term(a,-7),
term(b,2*x),term(b,pos(N)*x),term(b,-5*y),
term(b,-5)} => combinex(b).
```

And after all positive instances of the operator combinex(b) have been generated by the perturbation technique and generalized, the rule is formed:

```
{term(b,pos(N)*x),
 term(b,pos(M)*x)} => combinex(b).
```

Essentially, perturbation is a technique for creating near-examples and near-misses [13] with minimal teacher involvement upon which standard generalization techniques can be applied. Mitchell's LEX system [9] uses a similar method for creating training instances. Note that in our technique we perturb an instance and try the same operator that worked before. The perturbation is used to guide the generalization process. In LEX, perturbation is used to generate new, possibly solvable, problems. Heuristics are needed to select appropriate problems as the cost of applying the problem solver is high. See [5] for more detail on our perturbation technique.

6. Episodic Learning

6.1. Importance for Learning

As we noted in the MACROPS use of operator sequences, unless the system can select meaningfully useful sequences from the set of candidate sequences, combinatorial explosion makes re-use of generalized plans infeasible. We define an episode to be a sequence of rules which, when applied, simplifies a problem state. Our episodes are "loosely packaged" to allow branching. Rather than storing an entire plan for reaching a goal state from the start state, we segment the solution path into small, re-usable, generalizable episodes, each accomplishing a simplification of the problem.

Episodic, or stage 2, learning is concerned with these sequences of rules and their connections. These sequences are learned incrementally. Learning a rule for an operator depends on an understanding of <u>why</u> the operator is applied.[4] PET understands two reasons for selecting an operator:

1. By applying the operator, the problem state is simplified. In the domain of algebra problems, a state is simplified if the number of terms in the equations is reduced.

2. By applying the operator, the preconditions of an existing rule are satisfied. The rule being formed for the operator is then loosely linked with the rule that the operator enables. If more than one rule is enabled, then multiple branches through the episode are allowed.

PET adds a rule to the rulebase when the purpose of the rule's action (the "why" component of the operator) is understood. The first operators for which rules can be learned are those which simplify the problem state, such as combining like terms. Any operators applied before combine cannot be understood and PET must "bear with" the teacher. After rules are formed for the combine operators, subtract can be learned. For instance, sub(a,b) applied to:

```
    a: 2x+3y=5
    b: 2x-1y=1
yields:
    a: 2x+3y=5
    b: 2x-2x -1y-3y=1-5
```

Now PET can learn sub(a,b) for reason (2) above: a rule which is already understood (for a combine operator) is enabled by the subtraction. An episode can be formed connecting the rules for sub(a,b) and combine(b).

6.2. Scoring Operators

A simple scoring scheme connects rules into episodes and resolves conflicts when more than one rule is enabled. A natural scheme is to score each rule by its position in an episode. The rules for the combine operators are given a score of 0. The rule for subtract, which enables a combine operator, is given a score of 1 (0+1). Intuitively, the score is the length of the episode before something good happens (i.e. the equations get simplified). When selecting a rule, PET selects the one with the lowest score among those enabled. Ties are resolved arbitrarily.

---

[4] "Understand" is used here to mean "know-how" encoded in production rules. We do not mean to suggest a deep model of understanding which might include causality and analogy.

7. Augmentation

A description in the instance language is basically a translation of a training instance. This description is in an instance language more appropriate to computation than the surface language used to input the instance. In complex domains, more knowledge needs to be represented than is captured in a literal translation of a training instance into the instance language.

In the domain of algebra problems, augmentation serves to relate terms in the instance language. For example, a relevant relation between coefficients is productof(N,M,P) (the product of N and M is P). Augmentation of the instance language is necessary when the terms or values necessary for an operation (the RHS of a rule) are not present in the pre-conditions for the operation (the LHS of the rule). For example, consider the training instance:

```
a: 6x-15y=-3
b: 3x+4y=10
```

with the teacher advice to apply mult(b,2). The problem is that the rule formed by PET will have a 2 on the RHS (for the operation) but no 2 on the LHS. In this case, we say that the rule is not predictive of the operator.

An augmentation of the instance language is needed to relate the 2 on the RHS with some term on the LHS. In this case, the additional knowledge needed is the 3-ary predicate productof, specifically productof(2,3,6). Now the rule to cover the training instance can be formed (after perturbation and augmentation):

```
{term(a,6*x),term(b,3*x),
 productof(2,3,6)} => mult(a,2)
```

Concepts in the augmentation language form a second-order search space (figure 7-1) for generalizing an operation. The space consists of a (partial) list of concepts that a student might rely on for understanding relations between numbers. When a predictive rule cannot be found in the first-order search space then PET tries to form a rule using the augmentation as well. Concepts are pulled from the list and added to a developing rule. If the concept makes the rule predictive, then it is retained. Otherwise, it is removed and another concept is tried. If no predictive rule can be found then PET ignores the training instance.

sumof(L,M,N)	(sum of L and M is N)
productof(L,M,N)	(product of L and M is N)
squareof(M,N)	(square of M is N)

Figure 7-1: augmentation search space

Vere [12] has also addressed the problem of learning in the presence of "background information." For example, learning a general rule for a straight in a poker hand requires knowledge of the next number in sequence. This is considered background to the knowledge in the poker domain. Vere describes an "association chain" which links together each term in a rule. If a term in the rule is not linked in the chain (analogous to our test for predictiveness), then more background information must be "pulled in" until it is associated.

Augmentation is similar to selecting background knowledge. One problem with both approaches is determining how much background knowledge to incorporate. Incorporating too little knowledge, which results in an over-generalized rule, can be detected by an association chain violation or, in PET, by a non-predictive rule. However, detecting when too much knowledge has been pulled in is difficult. In this case, the rule formed will be over-specialized. We overcome this problem (to a large extent) by perturbation. Vere relies solely on forming a disjunction of rules (each overly specialized) for the correct generalization.

Vere allows only one concept in the background knowledge. This further simplifies the task of knowing how much knowledge to pull in. However, as the complexity of problem domains increase, more background knowledge must be brought to bear. Our augmentation addresses some of the problems of managing this knowledge.

8. Examples of System Performance

This section discusses highlights from PET's episodic learning for problem solving in the domain of linear equations.

8.1. Example 1—Learning Combine

The rulebase is initially empty and, as PET learns, rules are added, generalized, and supplanted. PET requests advice whenever the current rules do not apply to the problem state.

The teacher presents the training instance:

```
a: 2x+3y=5
b: 2x+4y=6
```

with the advice sub(a,b). PET applies the operator which yields:

```
a: 2x+3y=5
b: 2x-2x+4y-3y=6-5
```

PET must understand why an operator is useful before a rule is formed. The operator failed to simplify the equations (in fact the number of terms in the equations went from six to nine) and did not enable any other rules (since the rulebase is empty). PET cannot form a rule and waits for something understandable to happen.

The teacher now suggests that combinex(b) be applied, yielding:

```
a: 2x+3y=5
b: 0x+4y-3y=6-5
```

The number of terms is reduced, so PET hypothesizes a rule. Perturbation tests each term in the equations to determine which are essential and which can be generalized. PET forms the rule:

```
{term(b,pos(N)*x),
 term(b,neg(M)*x)} => combinex(b)
```

which means:

given a problem state, whenever equation b contains an x-term with a positive coefficient and an x-term with a negative coefficient, then combine the two terms.

PET is unable to apply current knowledge (i.e. the rule for combinex(b)) to the current problem state so the teacher suggests combiney(b)[5] which yields:

```
a: 2x+3y=5
b: 0x+1y=6-5
```

Stage 1 learning produces the rule:

```
{term(b,pos(N)*y),
 term(b,neg(M)*y)} => combiney(b)
```

This rule cannot be generalized with the current rulelist and is simply added.

Learning rules for the operators combinec(b) and deletezero(b) are similar and will be assumed to be completed.

Stage 2 learning of the combine operators involves relating them to episodes, or sequences of rules. Since combine simplifies a problem state immediately, it is given a score of zero. The current rulelist (with scores) is:

```
0 -- {term(b,pos(N)*x),
      term(b,neg(M)*x)} => combinex(b)
0 -- {term(b,pos(N)*y),
      term(b,neg(M)*y)} => combiney(b)
0 -- {term(b,pos(N)),
      term(b,neg(M)} => combinec(b)
0 -- {term(b,0*x)} => deletezero(b)
0 -- {term(b,0)} => deletezero(b)
```

With further training instances for the combine operators, PET forms the rules:

```
0--{term(eqn(L),int(N)*x),
  term(eqn(L),int(M)*x)}=>combinex(eqn(L))
0--{term(eqn(L),int(N)*y),
  term(eqn(L),int(M)*y)}=>combiney(eqn(L))
0--{term(eqn(L),int(N)),
  term(eqn(L),int(M)} => combinec(eqn(L))
0--{term(eqn(L),0*var(X))}
      => deletezero(eqn(L))
0--{term(eqn(L),0)} => deletezero(eqn(L))
```

8.2. Example 2—Learning Subtract

Now that the combine operators are partially learned, PET can begin to learn subtract. Since our episodes are loosely packaged, there is not a problem with further generalizing the rules for the combine operators once subtract is affiliated with them. Learning for each operator can proceed independently.

Now the training instance above which PET had to ignore can be understood. The instance with teacher advice sub(a,b) is:

```
a: 2x+3y=5
b: 2x+4y=6
```

The operator enables the partially-learned rule for combinex(b), so PET employs perturbation to form the rule:

```
{term(a,2*x),term(b,2*x),
 term(b,pos(N)*y)} => sub(a,b)
```

The rule is assigned a score of 1, loosely linking it with the rules for combine and deletezero.

Presented with further training instances, PET induces the rule:

```
{term(eqn(L1),nonzero(N)*var(X)),
 term(eqn(L2),nonzero(M)*var(X)),
 term(eqn(L2),nonzero(O)*var(Y))}
   => sub(eqn(L1),eqn(L2))
```

8.3. Example 3—Learning Multiply

The teacher presents PET with an example of multiply with the training instance:

```
a: 3x+4y=7
b: 6x+2y=8
```

No rule in the knowledge base applies, so PET requests advice. mult(a,2) is suggested which yields:

```
a: 6x+8y=14
b: 6x+2y=8
```

Now sub(a,b) applies so the rule for mult(a,2) can be learned. Mult(a,2) is given a score of 2 (one more than the score of the rule enabled). After perturbation PET forms the rule:

```
{term(a,3*x),term(b,6*x),
 term(b,pos(N)*y)} => mult(a,2)
```

At this point PET realizes that it has over-generalized since the rule is non-predictive (the 2 on the RHS does not occur on the LHS). PET augments the instance description and forms the candidate rule:

```
{term(a,3*x),term(b,6*x),term(b,pos(N)*y),
 productof(2,3,6)} => mult(a,2)
```

After additional examples, PET forms the correct rule:

```
2 -- {term(a,pos(K)*x),term(b,pos(L)*x),
      term(b,pos(M)*y),
      productof(pos(N),pos(K),pos(L)}
           => mult(a,pos(N))
```

which supplants the more specific rule in the rulebase.

8.4. Example 4—Learning "Cross Multiply"

The teacher presents the training instance:

```
a: 2x+6y=8
b: 3x+4y=7
```

Since no rule is enabled, the teacher advice to apply mult(a,3) yields:

```
a: 6x+18y=24
b: 3x+ 4y=7
```

[5]deletezero(b) could also be suggested, but we continue with a combine operator for continuity.

Since mult(b,2) is enabled, mult(a,3) can be learned. After perturbation, PET acquires the rule:

```
{term(a,2*x),term(b,3*x),term(b,pos(N)*y)}
          => mult(a,3)
```

This rule is given a score of 3 since it enables a rule with score 2. The rule will be generalized (with subsequent training instances) to:

```
3 -- {term(eqn(I),nonzero(N)*var(X)),
      term(eqn(J),nonzero(M)*var(X)),
      term(eqn(J),nonzero(L)*var(Y))}
        => mult(eqn(I),nonzero(M))
```

9. Limitations and Extensions

As with most learning programs we require that the concept to be learned be representable in our generalization language. In addition PET has to be supplied with some coarse notion of when an operator has been effective in simplifying the current state. Furthermore we assume that the teacher gives only appropriate advice and there is no "noise."

Extensions that we are considering are:

- Learning from negative instances as well as positive ones.

- Improving the use of augmentation by introducing structured concepts. These would permit climbing tree generalizations for this second-order knowledge. Another improvement would be allowing multiple concepts to be pulled into a rule from the augmentation search space. This requires a requisite change in the test for predictiveness.

- Applying the theory to learning operators in other domains. Integration problems have been attempted [8]. We would like to try our approach in the calculus problem domain.

10. Conclusions

A system, PET, has been described which learns sequences of rules, or episodes, for problem solving. The learning is incremental and thorough. The system learns when and why operators are applied. Although PET starts with an extremely general and coarse notion of why an operator should be applied it's representation becomes increasingly fine and complete as it forms rules from examples. PET detects when rules are non-predictive and augments the generalization language with higher level concepts. Due to the power of perturbation, our system can learn episodes with minimal teacher interaction. The episodes are segmented into discrete, re-usable segments, each accomplishing a recognizable simplification of the problem state. The approach is shown effective in the domain of solving simultaneous linear equations. We suspect the technique will also work for solving problems in symbolic integration and differential equations

REFERENCES

1. Carbonell, J.G. Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In Michalski,R.S., Carbonell,J.G., Mitchell,T.M., Ed., Machine Learning, Tiogo Publishing, 1983.

2. Fikes, R.E. and Nilsson, N.J. STRIPS: A new approach to the application of theorem proving to problem solving. AI 2 (1971), 189-208.

3. Kibler, D.F, and Morris, P.H. Dont be Stupid. IJCAI (1981), 345-347.

4. Kibler, D.F. and Porter, B.W. Episodic Learning. 194, University of California, Irvine, 1983.

5. Kibler, D.F. and Porter, B.W. Perturbation: A Means for Guiding Generalization. IJCAI (1983).

6. Michalski, R.S., Dietterich, T.G. Learning and Generalization of Characteristic Descriptions: Evaluation Criteria and Comparative Review of Selected Methods. IJCAI 6 (1979), 223-231.

7. Michalski,R.S., Carbonell,J.G., Mitchell,T.M. Machine Learning. Tiogo Publishing, 1983.

8. Mitchell, T.M., Utgoff, P.E., Nudel, B, and Banerji, R. Learning Problem-Solving Heuristics Through Practice. IJCAI 7 (1981), 127-134.

9. Mitchell, T.M., Utgoff, P.E., Nudel, B, and Banerji, R. Learning by Experimentation: Acquiring and Refining Problem- Solving Heuistics. In Michalski,R.S., Carbonell,J.G., Mitchell,T.M., Ed., Machine Learning, Tiogo Publishing, 1983.

10. Neves, D.M. A computer program that learns algebraic procedures by examining examples and working problems in a textbook. CSCSI II (1978), 191-195.

11. Vere, S.A. Induction of concepts in the predicate calculus. IJCAI 4 (1975), 281-287.

12. Vere, S.A. Induction of Relational Productions in the Presence of Background Information. IJCAI 5 (1977), 349-355.

13. Winston, P.H. Learning structural description from examples. In Winston, P.H., Ed., The Psychology of Computer Vision, McGraw-Hill, 1975.