

Dynamically Delayed Postdictive Completeness and Consistency in Learning

John Case, **Timo Kötzing**
University of Delaware

October 14, 2008

- ▶ Let $\mathbb{N} = \{0, 1, 2, \dots\}$.
- ▶ For any total function $g : \mathbb{N} \rightarrow \mathbb{N}$ and for all k we let $g[k]$ denote the sequence $g(0), \dots, g(k-1)$.
- ▶ Given a learner h and a learner $g : \mathbb{N} \rightarrow \mathbb{N}$ we define the **learning-in-the-limit sequence p of h on g** by

$$\forall k : p(k) := h(g[k]).$$

- ▶ We say that h **Ex-learns** g iff, for some i , $p(i)$ computes g and $p(i) = p(i+1) = p(i+2) = \dots$.
- ▶ A set of computable functions is called **Ex-learnable** iff there is a learner learning every function in that set.
- ▶ For example: **PF** is **Ex-learnable**.

- ▶ Let $\mathbb{N} = \{0, 1, 2, \dots\}$.
- ▶ For any total function $g : \mathbb{N} \rightarrow \mathbb{N}$ and for all k we let $g[k]$ denote the sequence $g(0), \dots, g(k-1)$.
- ▶ Given a learner h and a learnee $g : \mathbb{N} \rightarrow \mathbb{N}$ we define the **learning-in-the-limit sequence p of h on g** by

$$\forall k : p(k) := h(g[k]).$$

- ▶ We say that h **Ex-learns** g iff, for some i , $p(i)$ computes g and $p(i) = p(i+1) = p(i+2) = \dots$.
- ▶ A set of computable functions is called **Ex-learnable** iff there is a learner learning every function in that set.
- ▶ For example: **PF** is **Ex-learnable**.

- ▶ Let $\mathbb{N} = \{0, 1, 2, \dots\}$.
- ▶ For any total function $g : \mathbb{N} \rightarrow \mathbb{N}$ and for all k we let $g[k]$ denote the sequence $g(0), \dots, g(k-1)$.
- ▶ Given a learner h and a learner $g : \mathbb{N} \rightarrow \mathbb{N}$ we define the **learning-in-the-limit sequence p of h on g** by

$$\forall k : p(k) := h(g[k]).$$

- ▶ We say that h **Ex-learns** g iff, for some i , $p(i)$ computes g and $p(i) = p(i+1) = p(i+2) = \dots$.
- ▶ A set of computable functions is called **Ex-learnable** iff there is a learner learning every function in that set.
- ▶ For example: **PF** is **Ex-learnable**.

- ▶ Let $\mathbb{N} = \{0, 1, 2, \dots\}$.
- ▶ For any total function $g : \mathbb{N} \rightarrow \mathbb{N}$ and for all k we let $g[k]$ denote the sequence $g(0), \dots, g(k-1)$.
- ▶ Given a learner h and a learner $g : \mathbb{N} \rightarrow \mathbb{N}$ we define the **learning-in-the-limit sequence p of h on g** by

$$\forall k : p(k) := h(g[k]).$$

- ▶ We say that h **Ex-learns** g iff, for some i , $p(i)$ computes g and $p(i) = p(i+1) = p(i+2) = \dots$.
- ▶ A set of computable functions is called **Ex-learnable** iff there is a learner learning every function in that set.
- ▶ For example: **PF** is **Ex-learnable**.

- ▶ Let $\mathbb{N} = \{0, 1, 2, \dots\}$.
- ▶ For any total function $g : \mathbb{N} \rightarrow \mathbb{N}$ and for all k we let $g[k]$ denote the sequence $g(0), \dots, g(k-1)$.
- ▶ Given a learner h and a learner $g : \mathbb{N} \rightarrow \mathbb{N}$ we define the **learning-in-the-limit sequence p of h on g** by

$$\forall k : p(k) := h(g[k]).$$

- ▶ We say that h **Ex-learns** g iff, for some i , $p(i)$ computes g and $p(i) = p(i+1) = p(i+2) = \dots$.
- ▶ A set of computable functions is called **Ex-learnable** iff there is a learner learning every function in that set.
- ▶ For example: **PF** is **Ex-learnable**.

- ▶ Let $\mathbb{N} = \{0, 1, 2, \dots\}$.
- ▶ For any total function $g : \mathbb{N} \rightarrow \mathbb{N}$ and for all k we let $g[k]$ denote the sequence $g(0), \dots, g(k-1)$.
- ▶ Given a learner h and a learner $g : \mathbb{N} \rightarrow \mathbb{N}$ we define the **learning-in-the-limit sequence p of h on g** by

$$\forall k : p(k) := h(g[k]).$$

- ▶ We say that h **Ex-learns** g iff, for some i , $p(i)$ computes g and $p(i) = p(i+1) = p(i+2) = \dots$.
- ▶ A set of computable functions is called **Ex-learnable** iff there is a learner learning every function in that set.
- ▶ For example: **PF** is **Ex-learnable**.

Some Feasibility Restrictions on Learning Time

- ▶ We sometimes want the computation of h on $g[k]$ ($= g(0), \dots, g(k-1)$) to take no more than **polytime** (in $k + \sum_{i=0}^{k-1} |g(i)|$).
- ▶ **Fact (Pitt '89)**: For every Ex-learnable set of functions \mathcal{S} , there is such a polytime computable learner learning \mathcal{S} .
- ▶ **Why?** A polytime learner can be obtained from **postponing** necessary computations to a later time, where sufficient computing time is available (due to having a longer input).
- ▶ As a result, polytime learning as above introduces no actual **efficiency**, as **unfair postponement tricks** can be used.
- ▶ Hence, correspondingly we seek to **limit unfair postponement tricks** by putting additional restrictions on the intermediate hypotheses $p(0), p(1), \dots$.

Some Feasibility Restrictions on Learning Time

- ▶ We sometimes want the computation of h on $g[k]$ ($= g(0), \dots, g(k-1)$) to take no more than **polytime** (in $k + \sum_{i=0}^{k-1} |g(i)|$).
- ▶ **Fact (Pitt '89)**: For every Ex-learnable set of functions \mathcal{S} , there is such a polytime computable learner learning \mathcal{S} .
- ▶ **Why?** A polytime learner can be obtained from **postponing** necessary computations to a later time, where sufficient computing time is available (due to having a longer input).
- ▶ As a result, polytime learning as above introduces no actual **efficiency**, as **unfair postponement tricks** can be used.
- ▶ Hence, correspondingly we seek to **limit unfair postponement tricks** by putting additional restrictions on the intermediate hypotheses $p(0), p(1), \dots$.

Some Feasibility Restrictions on Learning Time

- ▶ We sometimes want the computation of h on $g[k]$ ($= g(0), \dots, g(k-1)$) to take no more than **polytime** (in $k + \sum_{i=0}^{k-1} |g(i)|$).
- ▶ **Fact (Pitt '89)**: For every Ex-learnable set of functions \mathcal{S} , there is such a polytime computable learner learning \mathcal{S} .
- ▶ **Why?** A polytime learner can be obtained from **postponing** necessary computations to a later time, where sufficient computing time is available (due to having a longer input).
- ▶ As a result, polytime learning as above introduces no actual **efficiency**, as **unfair postponement tricks** can be used.
- ▶ Hence, correspondingly we seek to **limit unfair postponement tricks** by putting additional restrictions on the intermediate hypotheses $p(0), p(1), \dots$.

Some Feasibility Restrictions on Learning Time

- ▶ We sometimes want the computation of h on $g[k]$ ($= g(0), \dots, g(k-1)$) to take no more than **polytime** (in $k + \sum_{i=0}^{k-1} |g(i)|$).
- ▶ **Fact (Pitt '89)**: For every Ex-learnable set of functions \mathcal{S} , there is such a polytime computable learner learning \mathcal{S} .
- ▶ **Why?** A polytime learner can be obtained from **postponing** necessary computations to a later time, where sufficient computing time is available (due to having a longer input).
- ▶ As a result, polytime learning as above introduces no actual **efficiency**, as **unfair postponement tricks** can be used.
- ▶ Hence, correspondingly we seek to **limit unfair postponement tricks** by putting additional restrictions on the intermediate hypotheses $p(0), p(1), \dots$.

Some Feasibility Restrictions on Learning Time

- ▶ We sometimes want the computation of h on $g[k]$ ($= g(0), \dots, g(k-1)$) to take no more than **polytime** (in $k + \sum_{i=0}^{k-1} |g(i)|$).
- ▶ **Fact (Pitt '89)**: For every Ex-learnable set of functions \mathcal{S} , there is such a polytime computable learner learning \mathcal{S} .
- ▶ **Why?** A polytime learner can be obtained from **postponing** necessary computations to a later time, where sufficient computing time is available (due to having a longer input).
- ▶ As a result, polytime learning as above introduces no actual **efficiency**, as **unfair postponement tricks** can be used.
- ▶ Hence, correspondingly we seek to **limit unfair postponement tricks** by putting additional restrictions on the intermediate hypotheses $p(0), p(1), \dots$.

Some Feasibility Restrictions on Learning Time

- ▶ We sometimes want the computation of h on $g[k]$ ($= g(0), \dots, g(k-1)$) to take no more than **polytime** (in $k + \sum_{i=0}^{k-1} |g(i)|$).
- ▶ **Fact (Pitt '89)**: For every Ex-learnable set of functions \mathcal{S} , there is such a polytime computable learner learning \mathcal{S} .
- ▶ **Why?** A polytime learner can be obtained from **postponing** necessary computations to a later time, where sufficient computing time is available (due to having a longer input).
- ▶ As a result, polytime learning as above introduces no actual **efficiency**, as **unfair postponement tricks** can be used.
- ▶ Hence, correspondingly we seek to **limit unfair postponement tricks** by putting additional restrictions on the intermediate hypotheses $p(0), p(1), \dots$.

Postdictive Completeness (Special Case)

- ▶ A learner h is called **postdictively complete** iff $\forall g, k, h(g[k])$ correctly postdicts $g[k]$ ($= g(0), \dots, g(k-1)$), i.e., the programs $h(g[k])$ correctly compute g on all inputs $< k$.
- ▶ Akama & Zeugmann '07 presented, for each $n \in \mathbb{N}$, a relaxation to postdictive completeness: each conjecture is required to postdict only all except the last n seen data points.
- ▶ Essentially, such a delay n learner could, for each k , after seeing $g[k]$, run a counter down from n to 0 to see which future hypotheses must correctly compute $g[k]$.
- ▶ We extend this notion of delayed postdictive completeness from constant delays to **dynamically computed** delays; firstly, a different delay may be used for each datum seen; secondly, countdowns from constants can be replaced by dynamic countdowns. We explain below.

Postdictive Completeness (Special Case)

- ▶ A learner h is called **postdictively complete** iff $\forall g, k, h(g[k])$ correctly postdicts $g[k]$ ($= g(0), \dots, g(k-1)$), i.e., the programs $h(g[k])$ correctly compute g on all inputs $< k$.
- ▶ Akama & Zeugmann '07 presented, for each $n \in \mathbb{N}$, a relaxation to postdictive completeness: each conjecture is required to postdict only all except the last n seen data points.
- ▶ Essentially, such a delay n learner could, for each k , after seeing $g[k]$, run a counter down from n to 0 to see which future hypotheses must correctly compute $g[k]$.
- ▶ We extend this notion of delayed postdictive completeness from constant delays to **dynamically computed** delays; firstly, a different delay may be used for each datum seen; secondly, countdowns from constants can be replaced by dynamic countdowns. We explain below.

Postdictive Completeness (Special Case)

- ▶ A learner h is called **postdictively complete** iff $\forall g, k, h(g[k])$ correctly postdicts $g[k]$ ($= g(0), \dots, g(k-1)$), i.e., the programs $h(g[k])$ correctly compute g on all inputs $< k$.
- ▶ Akama & Zeugmann '07 presented, for each $n \in \mathbb{N}$, a relaxation to postdictive completeness: each conjecture is required to postdict only all except the last n seen data points.
- ▶ Essentially, such a delay n learner could, for each k , after seeing $g[k]$, run a counter down from n to 0 to see which future hypotheses must correctly compute $g[k]$.
- ▶ We extend this notion of delayed postdictive completeness from constant delays to **dynamically computed** delays; firstly, a different delay may be used for each datum seen; secondly, countdowns from constants can be replaced by dynamic countdowns. We explain below.

Postdictive Completeness (Special Case)

- ▶ A learner h is called **postdictively complete** iff $\forall g, k, h(g[k])$ correctly postdicts $g[k]$ ($= g(0), \dots, g(k-1)$), i.e., the programs $h(g[k])$ correctly compute g on all inputs $< k$.
- ▶ Akama & Zeugmann '07 presented, for each $n \in \mathbb{N}$, a relaxation to postdictive completeness: each conjecture is required to postdict only all except the last n seen data points.
- ▶ Essentially, such a delay n learner could, for each k , after seeing $g[k]$, run a counter down from n to 0 to see which future hypotheses must correctly compute $g[k]$.
- ▶ We extend this notion of delayed postdictive completeness from constant delays to **dynamically computed** delays; firstly, a different delay may be used for each datum seen; secondly, countdowns from constants can be replaced by dynamic countdowns. We explain below.

Postdictive Completeness (Special Case)

- ▶ A learner h is called **postdictively complete** iff $\forall g, k, h(g[k])$ correctly postdicts $g[k]$ ($= g(0), \dots, g(k-1)$), i.e., the programs $h(g[k])$ correctly compute g on all inputs $< k$.
- ▶ Akama & Zeugmann '07 presented, for each $n \in \mathbb{N}$, a relaxation to postdictive completeness: each conjecture is required to postdict only all except the last n seen data points.
- ▶ Essentially, such a delay n learner could, for each k , after seeing $g[k]$, run a counter down from n to 0 to see which future hypotheses must correctly compute $g[k]$.
- ▶ We extend this notion of delayed postdictive completeness from constant delays to **dynamically computed** delays; firstly, a different delay may be used for each datum seen; secondly, countdowns from constants can be replaced by dynamic countdowns. We explain below.

Constructive Ordinals and Notations

- ▶ One of the ways we consider is (algorithmic) counting down from a notation for a constructive ordinal (each to be explained).
- ▶ Ordinals represent well-orderings, for example:

$$0, 1, 2, \dots \underbrace{\quad}_{\omega} \quad 0, 1, 2, \dots \underbrace{\quad}_{\omega+\omega}$$

is the ordinal $\omega + \omega = \omega \cdot 2$.

- ▶ The constructive ordinals, like $\omega \cdot 2$ above, are those having a program, called a notation, for specifying how to lay them out left-to-right.
- ▶ Everyone knows how to employ finite ordinals for counting down. Notations enable us to count down also from infinite (constructive) ordinals, like $\omega \cdot 2$ above.

Constructive Ordinals and Notations

- ▶ One of the ways we consider is (algorithmic) counting down from a notation for a constructive ordinal (each to be explained).
- ▶ Ordinals represent well-orderings, for example:

$$0, 1, 2, \dots \overbrace{) }^{\omega} 0, 1, 2, \dots \overbrace{) }^{\omega + \omega}$$

is the ordinal $\omega + \omega = \omega \cdot 2$.

- ▶ The constructive ordinals, like $\omega \cdot 2$ above, are those having a program, called a notation, for specifying how to lay them out left-to-right.
- ▶ Everyone knows how to employ finite ordinals for counting down. Notations enable us to count down also from infinite (constructive) ordinals, like $\omega \cdot 2$ above.

Constructive Ordinals and Notations

- ▶ One of the ways we consider is (algorithmic) counting down from a notation for a constructive ordinal (each to be explained).
- ▶ Ordinals represent well-orderings, for example:

$$0, 1, 2, \dots \overbrace{) }^{\omega} 0, 1, 2, \dots \overbrace{) }^{\omega + \omega}$$

is the ordinal $\omega + \omega = \omega \cdot 2$.

- ▶ The constructive ordinals, like $\omega \cdot 2$ above, are those having a program, called a notation, for specifying how to lay them out left-to-right.
- ▶ Everyone knows how to employ finite ordinals for counting down. Notations enable us to count down also from infinite (constructive) ordinals, like $\omega \cdot 2$ above.

Constructive Ordinals and Notations

- ▶ One of the ways we consider is (algorithmic) counting down from a notation for a constructive ordinal (each to be explained).
- ▶ Ordinals represent well-orderings, for example:

$$0, 1, 2, \dots \overbrace{) }^{\omega} 0, 1, 2, \dots \overbrace{) }^{\omega + \omega}$$

is the ordinal $\omega + \omega = \omega \cdot 2$.

- ▶ The constructive ordinals, like $\omega \cdot 2$ above, are those having a program, called a notation, for specifying how to lay them out left-to-right.
- ▶ Everyone knows how to employ finite ordinals for counting down. Notations enable us to count down also from infinite (constructive) ordinals, like $\omega \cdot 2$ above.

Constructive Ordinal Countdown

$$0, 1, 2, \dots \overset{\omega}{\underbrace{)} } 0, 1, 2, \dots \overset{\omega+\omega}{\underbrace{)} }$$

- ▶ **Example** countdown for $\omega \cdot 2$ above: start from the right-most end, leap left to, say, right-most copy of 2, then walk left to adjacent 1, then left to adjacent 0, then leap to left-most 42, then leap left to 1, then walk left to 0.
- ▶ Counting down from (notations for) infinite constructive ordinals is more powerful than counting down from finite ordinals. E.g., some counting down from a notation for $\omega + 1$ is equivalent to counting down once and, then, deciding dynamically how many further but finite number of times to count down.

Constructive Ordinal Countdown

$$0, 1, 2, \dots \overset{\omega}{\underbrace{)} } 0, 1, 2, \dots \overset{\omega+\omega}{\underbrace{)} }$$

- ▶ **Example** countdown for $\omega \cdot 2$ above: start from the right-most end, leap left to, say, right-most copy of 2, then walk left to adjacent 1, then left to adjacent 0, then leap to left-most 42, then leap left to 1, then walk left to 0.
- ▶ Counting down from (notations for) infinite constructive ordinals is more powerful than counting down from finite ordinals. E.g., some counting down from a notation for $\omega + 1$ is equivalent to counting down once and, then, deciding dynamically how many further but finite number of times to count down.

Constructive Ordinal Countdown

$$0, 1, 2, \dots \underbrace{\quad}_{\omega} \quad 0, 1, 2, \dots \underbrace{\quad}_{\omega+\omega}$$

- ▶ **Example** countdown for $\omega \cdot 2$ above: start from the right-most end, leap left to, say, right-most copy of 2, then walk left to adjacent 1, then left to adjacent 0, then leap to left-most 42, then leap left to 1, then walk left to 0.
- ▶ Counting down from (notations for) infinite constructive ordinals is more powerful than counting down from finite ordinals. E.g., some counting down from a notation for $\omega + 1$ is equivalent to counting down once and, then, deciding dynamically how many further but finite number of times to count down.

Use of Countdown – Hierarchy Results

- ▶ For our use of algorithmic countdown, for example, from a notation for a constructive ordinal: a fresh countdown is started for each datum seen; this datum has to be correctly postdicted once the countdown terminates.

Results:

- ▶ Akama and Zeugmann showed that there is a strict hierarchy for the learning power of postdictively complete learning with finite delay.
- ▶ We extended this hierarchy into the constructive transfinite.

Use of Countdown – Hierarchy Results

- ▶ For our use of algorithmic countdown, for example, from a notation for a constructive ordinal: a fresh countdown is started for each datum seen; this datum has to be correctly postdicted once the countdown terminates.

Results:

- ▶ Akama and Zeugmann showed that there is a strict hierarchy for the learning power of postdictively complete learning with **finite** delay.
- ▶ We extended this hierarchy into the constructive **transfinite**.

Use of Countdown – Hierarchy Results

- ▶ For our use of algorithmic countdown, for example, from a notation for a constructive ordinal: a fresh countdown is started for each datum seen; this datum has to be correctly postdicted once the countdown terminates.

Results:

- ▶ Akama and Zeugmann showed that there is a strict hierarchy for the learning power of postdictively complete learning with **finite** delay.
- ▶ We extended this hierarchy into the constructive **transfinite**.

Use of Countdown – Hierarchy Results

- ▶ For our use of algorithmic countdown, for example, from a notation for a constructive ordinal: a fresh countdown is started for each datum seen; this datum has to be correctly postdicted once the countdown terminates.

Results:

- ▶ Akama and Zeugmann showed that there is a strict hierarchy for the learning power of postdictively complete learning with **finite** delay.
- ▶ We extended this hierarchy into the constructive **transfinite**.

Feasibility Results

Regarding feasibility: We use only **feasible** notations (which every constructive ordinal has) **and** use polytime computable functions again, this time to handle the counting down.

Now we have, regarding **delayed postdictively complete** learning:

- ▶ PF is feasibly learnable without delay;
- ▶ EXPF is not feasibly learnable without delay;
- ▶ EXPF is feasibly learnable with ω delay;
- ▶ EXPF2 is not feasibly learnable with ω delay;
- ▶ EXPF2 is feasibly learnable with $\omega \cdot 2$ delay;
- ▶ EXPF3 is not feasibly learnable with $\omega \cdot 2$ delay.

Feasibility Results

Regarding feasibility: We use only **feasible** notations (which every constructive ordinal has) **and** use polytime computable functions again, this time to handle the counting down.

Now we have, regarding **delayed postdictively complete** learning:

- ▶ **PF** is feasibly learnable without delay;
- ▶ **EXPF** is **not** feasibly learnable without delay;
- ▶ **EXPF** is feasibly learnable with ω delay;
- ▶ **EXPF2** is **not** feasibly learnable with ω delay;
- ▶ **EXPF2** is feasibly learnable with $\omega \cdot 2$ delay;
- ▶ **EXPF3** is **not** feasibly learnable with $\omega \cdot 2$ delay.

⋮

Feasibility Results

Regarding feasibility: We use only **feasible** notations (which every constructive ordinal has) **and** use polytime computable functions again, this time to handle the counting down.

Now we have, regarding **delayed postdictively complete** learning:

- ▶ **PF** is feasibly learnable without delay;
- ▶ **EXPF** is **not** feasibly learnable without delay;
- ▶ **EXPF** is feasibly learnable with ω delay;
- ▶ **EXPF2** is **not** feasibly learnable with ω delay;
- ▶ **EXPF2** is feasibly learnable with $\omega \cdot 2$ delay;
- ▶ **EXPF3** is **not** feasibly learnable with $\omega \cdot 2$ delay.

⋮

Feasibility Results

Regarding feasibility: We use only **feasible** notations (which every constructive ordinal has) **and** use polytime computable functions again, this time to handle the counting down.

Now we have, regarding **delayed postdictively complete** learning:

- ▶ **PF** is feasibly learnable without delay;
- ▶ **EXPF** is **not** feasibly learnable without delay;
- ▶ **EXPF** is feasibly learnable with ω delay;
- ▶ **EXPF2** is **not** feasibly learnable with ω delay;
- ▶ **EXPF2** is feasibly learnable with $\omega \cdot 2$ delay;
- ▶ **EXPF3** is **not** feasibly learnable with $\omega \cdot 2$ delay.

⋮

Feasibility Results

Regarding feasibility: We use only **feasible** notations (which every constructive ordinal has) **and** use polytime computable functions again, this time to handle the counting down.

Now we have, regarding **delayed postdictively complete** learning:

- ▶ **PF** is feasibly learnable without delay;
- ▶ **EXPF** is **not** feasibly learnable without delay;
- ▶ **EXPF** is feasibly learnable with ω delay;
- ▶ **EXPF2** is **not** feasibly learnable with ω delay;
- ▶ **EXPF2** is feasibly learnable with $\omega \cdot 2$ delay;
- ▶ **EXPF3** is **not** feasibly learnable with $\omega \cdot 2$ delay.

⋮

Feasibility Results

Regarding feasibility: We use only **feasible** notations (which every constructive ordinal has) **and** use polytime computable functions again, this time to handle the counting down.

Now we have, regarding **delayed postdictively complete** learning:

- ▶ **PF** is feasibly learnable without delay;
- ▶ **EXPF** is **not** feasibly learnable without delay;
- ▶ **EXPF** is feasibly learnable with ω delay;
- ▶ **EXPF2** is **not** feasibly learnable with ω delay;
- ▶ **EXPF2** is feasibly learnable with $\omega \cdot 2$ delay;
- ▶ **EXPF3** is **not** feasibly learnable with $\omega \cdot 2$ delay.

⋮

Feasibility Results

Regarding feasibility: We use only **feasible** notations (which every constructive ordinal has) **and** use polytime computable functions again, this time to handle the counting down.

Now we have, regarding **delayed postdictively complete** learning:

- ▶ **PF** is feasibly learnable without delay;
- ▶ **EXPF** is **not** feasibly learnable without delay;
- ▶ **EXPF** is feasibly learnable with ω delay;
- ▶ **EXPF2** is **not** feasibly learnable with ω delay;
- ▶ **EXPF2** is feasibly learnable with $\omega \cdot 2$ delay;
- ▶ **EXPF3** is **not** feasibly learnable with $\omega \cdot 2$ delay.

⋮

Feasibility Results

Regarding feasibility: We use only **feasible** notations (which every constructive ordinal has) **and** use polytime computable functions again, this time to handle the counting down.

Now we have, regarding **delayed postdictively complete** learning:

- ▶ **PF** is feasibly learnable without delay;
- ▶ **EXPF** is **not** feasibly learnable without delay;
- ▶ **EXPF** is feasibly learnable with ω delay;
- ▶ **EXPF2** is **not** feasibly learnable with ω delay;
- ▶ **EXPF2** is feasibly learnable with $\omega \cdot 2$ delay;
- ▶ **EXPF3** is **not** feasibly learnable with $\omega \cdot 2$ delay.

⋮

Feasibility Results

Regarding feasibility: We use only **feasible** notations (which every constructive ordinal has) **and** use polytime computable functions again, this time to handle the counting down.

Now we have, regarding **delayed postdictively complete** learning:

- ▶ **PF** is feasibly learnable without delay;
- ▶ **EXPF** is **not** feasibly learnable without delay;
- ▶ **EXPF** is feasibly learnable with ω delay;
- ▶ **EXPF2** is **not** feasibly learnable with ω delay;
- ▶ **EXPF2** is feasibly learnable with $\omega \cdot 2$ delay;
- ▶ **EXPF3** is **not** feasibly learnable with $\omega \cdot 2$ delay.

⋮

Generalized Countdown

- ▶ For algorithmic countdowns we allow arbitrary countdown graphs: binary relations without infinite **computable** descending chains (ordinals: special case). In our paper we characterize the relative learning power wrt computable countdown graphs.
- ▶ Suppose $A \subseteq \mathbb{N}$. A is **ce** iff there exists an algorithm listing A . A is **immune** iff it is inf. and does not have any inf. ce subset.
- ▶ For example, there exists computable R on \mathbb{N} :



1. There is a set \mathcal{S} learnable with alg. countdown wrt R , s.t., $\forall \alpha$ constructive, \mathcal{S} is not learnable with alg. countdown wrt α ;
2. $\forall \alpha$ constructive: any set \mathcal{S} learnable with algorithmic countdown wrt α is learnable with algorithmic countdown wrt R iff $\alpha < \omega \cdot 2$.

Generalized Countdown

- ▶ For algorithmic countdowns we allow arbitrary countdown graphs: binary relations without infinite **computable** descending chains (ordinals: special case). In our paper we characterize the relative learning power wrt computable countdown graphs.
- ▶ Suppose $A \subseteq \mathbb{N}$. A is **ce** iff there exists an algorithm listing A . A is **immune** iff it is inf. and does not have any inf. ce subset.
- ▶ For example, there exists computable R on \mathbb{N} :



1. There is a set \mathcal{S} learnable with alg. countdown wrt R , s.t., $\forall \alpha$ constructive, \mathcal{S} is not learnable with alg. countdown wrt α ;
2. $\forall \alpha$ constructive: any set \mathcal{S} learnable with algorithmic countdown wrt α is learnable with algorithmic countdown wrt R iff $\alpha < \omega \cdot 2$.

Generalized Countdown

- ▶ For algorithmic countdowns we allow arbitrary countdown graphs: binary relations without infinite **computable** descending chains (ordinals: special case). In our paper we characterize the relative learning power wrt computable countdown graphs.
- ▶ Suppose $A \subseteq \mathbb{N}$. A is **ce** iff there exists an algorithm listing A . A is **immune** iff it is inf. and does not have any inf. ce subset.
- ▶ For example, there exists computable R on \mathbb{N} :



1. There is a set \mathcal{S} learnable with alg. countdown wrt R , s.t., $\forall \alpha$ constructive, \mathcal{S} is not learnable with alg. countdown wrt α ;
2. $\forall \alpha$ constructive: any set \mathcal{S} learnable with algorithmic countdown wrt α is learnable with algorithmic countdown wrt R iff $\alpha < \omega \cdot 2$.

Generalized Countdown

- ▶ For algorithmic countdowns we allow arbitrary countdown graphs: binary relations without infinite **computable** descending chains (ordinals: special case). In our paper we characterize the relative learning power wrt computable countdown graphs.
- ▶ Suppose $A \subseteq \mathbb{N}$. A is **ce** iff there exists an algorithm listing A . A is **immune** iff it is inf. and does not have any inf. ce subset.
- ▶ For example, there exists computable R on \mathbb{N} :

$$\bullet, \bullet, \bullet, \dots) \underbrace{(\dots, \bullet, \bullet, \bullet)}_{\text{immune}}$$

1. There is a set \mathcal{S} learnable with alg. countdown wrt R , s.t., $\forall \alpha$ constructive, \mathcal{S} is not learnable with alg. countdown wrt α ;
2. $\forall \alpha$ constructive: any set \mathcal{S} learnable with algorithmic countdown wrt α is learnable with algorithmic countdown wrt R iff $\alpha < \omega \cdot 2$.

Generalized Countdown

- ▶ For algorithmic countdowns we allow arbitrary countdown graphs: binary relations without infinite **computable** descending chains (ordinals: special case). In our paper we characterize the relative learning power wrt computable countdown graphs.
- ▶ Suppose $A \subseteq \mathbb{N}$. A is **ce** iff there exists an algorithm listing A . A is **immune** iff it is inf. and does not have any inf. ce subset.
- ▶ For example, there exists computable R on \mathbb{N} :

$$\bullet, \bullet, \bullet, \dots) \underbrace{(\dots, \bullet, \bullet, \bullet)}_{\text{immune}}$$

1. There is a set \mathcal{S} learnable with alg. countdown wrt R , s.t., $\forall \alpha$ constructive, \mathcal{S} is not learnable with alg. countdown wrt α ;
2. $\forall \alpha$ constructive: any set \mathcal{S} learnable with algorithmic countdown wrt α is learnable with algorithmic countdown wrt R iff $\alpha < \omega \cdot 2$.

Generalized Countdown

- ▶ For algorithmic countdowns we allow arbitrary countdown graphs: binary relations without infinite **computable** descending chains (ordinals: special case). In our paper we characterize the relative learning power wrt computable countdown graphs.
- ▶ Suppose $A \subseteq \mathbb{N}$. A is **ce** iff there exists an algorithm listing A . A is **immune** iff it is inf. and does not have any inf. ce subset.
- ▶ For example, there exists computable R on \mathbb{N} :

$$\bullet, \bullet, \bullet, \dots) \underbrace{(\dots, \bullet, \bullet, \bullet)}_{\text{immune}}$$

1. There is a set \mathcal{S} learnable with alg. countdown wrt R , s.t., $\forall \alpha$ constructive, \mathcal{S} is not learnable with alg. countdown wrt α ;
2. $\forall \alpha$ constructive: any set \mathcal{S} learnable with algorithmic countdown wrt α is learnable with algorithmic countdown wrt R iff $\alpha < \omega \cdot 2$.

Generalized Countdown

- ▶ We generalize what structures are allowed to be used for algorithmic countdowns.
- ▶ For example, there exists computable R on \mathbb{N} :

$$\bullet, \bullet, \bullet, \dots) (\dots, \bullet, \bullet, \bullet$$

such that there are no infinite **computable** descending chains wrt R (although, unlike for ordinals, there are infinite descending chains wrt R).

Generalized Countdown

- ▶ We generalize what structures are allowed to be used for algorithmic countdowns.
- ▶ For example, there exists computable R on \mathbb{N} :

$$\bullet, \bullet, \bullet, \dots) (\dots, \bullet, \bullet, \bullet$$

such that there are no infinite **computable** descending chains wrt R (although, unlike for ordinals, there are infinite descending chains wrt R).