Towards Modular Machine Learning Pipelines

Aditya Modi [1] **Jivat Neet Kaur** [2] **Maggie Makar** [3] **Pavan Mallapragada** [1] **Amit Sharma** [2] **Emre Kıcıman** [4]
Adith Swaminathan [4]

Abstract

Pipelines of Machine Learning (ML) components are a popular and effective approach to divide and conquer many business-critical problems. A pipeline architecture implies a specific division of the overall problem, however current ML training approaches do not enforce this implied division. Consequently ML components can become *coupled* to one another after they are trained, which causes insidious effects. For instance, even when one coupled ML component in a pipeline is improved in isolation, the end-to-end pipeline performance can degrade. In this paper, we develop a conceptual framework to study ML coupling in pipelines and design new *modularity regularizers* that can eliminate coupling during ML training. We show that the resulting ML pipelines become modular (i.e., their components can be trained independently of one another) and discuss the trade-offs of our approach versus existing approaches to pipeline optimization.

1. Introduction

Many impactful systems that we interact with — like search advertising (Bottou et al., 2013), personalized recommendation (Hron et al., 2021), and embedding-based retrieval systems (Hu et al., 2022) — are powered by pipelines of Machine Learning (ML) components. Pipelines allow large teams to effectively divide-and-conquer a business-critical problem. For example in search advertising, one team can focus on the modular task of extracting the intrinsic quality of candidate ads, a second team can own the task of selecting the ads to be displayed, while yet another team can optimize the page layout for a good search experience.

[1]Microsoft [2]Microsoft Research India [3]Department of Computer Science and Engineering, University of Michigan Ann Arbor [4]Microsoft Research Redmond. Correspondence to: Aditya Modi <admodi@umich.edu>, Adith Swaminathan <adswamin@microsoft.com>.

In *ICML Workshop on Localized Learning (LLW)*, Honolulu, Hawaii, USA. 2023. Copyright 2023 by the author(s).

Architecting ML pipelines today is a trial-and-error art, despite our nuanced understanding of maintaining a single ML component in production (Sculley et al., 2015). Why so? An ML pipeline implies a specific division of the overall problem into modular tasks; however training of any single component does not *enforce* this division by default. When an ML component has an implicit dependency on another such that it violates pipeline modularity, we say that it is **coupled**. This is a very common phenomenon when an ML component consumes another ML prediction during training. Consider the 2-component pipeline example in Figure 1a, where $\mathcal{M}_1, \mathcal{M}_2$ are ML components mapping inputs $\{X_1\}$ (e.g., user and ad features) and $\{Z, X_2\}$ to outputs Z (e.g., ad quality) and Y (e.g., revenue) respectively. When the training data for the $\mathcal{M}_2$ component does not contain the ground truth values for Z (e.g., ad qualities are not directly observed in search advertising), they have to be imputed somehow. If we use a specific $\mathcal{M}_1$ snapshot to impute Z as in Figure 1b, $\hat{Z} := \mathcal{M}_1^0(X_1)$, then we inadvertently introduce coupling between $\mathcal{M}_2$ and $\mathcal{M}_1^0$. This is because when $\mathcal{M}_1$ is updated to $\mathcal{M}_1^1 \neq \mathcal{M}_1^0$ (as shown in Figure 1c), the updated Z predictions likely differ from the distribution of Z seen during $\mathcal{M}_2$ training. This distribution shift can surface unanticipated errors from the trained $\mathcal{M}_2$.

Coupled ML components can lead to insidious pipeline effects, termed as "self-defeating improvements" (Wu et al., 2021). These occur when an ML component is substantially better (e.g., $\mathcal{M}_1^0 \ll \mathcal{M}_1^1$) and yet the end-to-end pipeline performance degrades because $\mathcal{M}_2$ is coupled to $\mathcal{M}_1^0$. This is illustrated in Figure 1 where an $\mathcal{M}_2$ coupled to $\mathcal{M}_1^0$ produces a better prediction for Y than when it is composed with a substantially better $\mathcal{M}_1^1$.

We argue that ML pipelines should satisfy three desirable properties: (1) **Independently trainable**: multiple components can be trained in parallel with very limited communication or coordination needed between them; (2) **Consistent**: if a component is improved to its optimal version (i.e., replaced with the true data generating process for that component), the pipeline does not degrade; (3) **Aligned**: if a component is incrementally improved, the pipeline is again guaranteed to not degrade. Aligned pipelines may not be consistent — incremental shifts need not capture the large distribution shifts implied by consistency. When

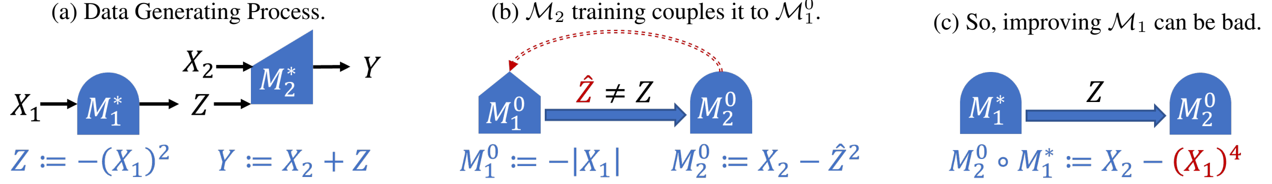

Figure 1: Demonstration of coupling in a pipeline with two ML components. (a) In the true Data Generating Process (DGP), Z is a polynomial of degree 2. (b) $\mathcal{M}_1^0$ fits a linear model to Z and $\mathcal{M}_2^0$ trained with imputed values from $\mathcal{M}_1^0$ corrects for its errors. (c) However when $\mathcal{M}_1$ is improved, $\mathcal{M}_2^0$ is now over-correcting which leads to worse pipeline performance.

components are coupled, all three properties are difficult to guarantee.

Table 1: Desired properties for pipeline training, and the training protocols that provide them. Our **Causal** proposal is the first protocol to guarantee both **Independent Trainability** and **Consistency**.

Property	Sync.	Async.	Proximal	Causal
Indep. Trainable	X	✓	✓	✓
Consistent	✓	X	X	✓
Aligned	✓	X	✓	X

In Table 1 we list the current approaches to pipeline optimization (and our proposed approach), and show the trade-offs among the properties that they provide. The approaches are — (1) **Sync.**: We train one component at a time; and whenever we update a component, we queue all dependent components to be retrained. (2) **Async.**: We train multiple components in parallel, and rely on external measurements (e.g., A/B tests) to reject component updates that cause the pipeline performance to degrade. (3) **Proximal**: we throttle the update rate of every ML component when training async., to hopefully not induce large distribution shifts in their downstream components.

Causal Pipeline Optimization: We propose new *modularity regularizers* that compose with async. training to guarantee consistency. Our regularizers are motivated by a careful analysis of the components' error sensitivities. The analysis also unifies a folk process for achieving modularity via robust optimization on perturbed inputs, along with a new complementary approach of perturbing the desired outputs of a component. We show in synthetic experiments that our regularizers effectively implement the **Causal** approach, and that the resulting consistency of trained components enables better pipeline optimization than existing protocols.

2. Preliminaries

We use uppercase X to denote random variables and lowercase x for their realizations in observed data. We reserve $\mathcal{M}$ for all ML components, with subscripts to identify each component, e.g., $\mathcal{M}_1$. Compositions describe a pipeline, e.g., $\mathcal{M}_2 \circ \mathcal{M}_1$ means $\mathcal{M}_2$ consumes $\mathcal{M}_1$'s output. Every trained snapshot of an ML component has a serial number indicated using superscripts, e.g., $\mathcal{M}_1^0$. When a random variable depends on an ML component, we use accents such as $\hat{X}$. We reserve $\mathcal{L}$ to denote loss functions, with subscripts identifying component-specific losses. We overload $\mathcal{L}$ to refer also to a component's aggregate loss, e.g., $\mathcal{L}(\mathcal{M}_2 \circ \mathcal{M}_1)$ describes a pipeline's overall loss.

Setting To simplify exposition, throughout the paper we consider a two component pipeline as shown in Figure 1a, which is the simplest example where ML coupling can occur, and which captures real-world pipelines. In a search advertising system (Figure 1a), $\mathcal{M}_1$ is an ML component that computes "user's propensity to click on ad" Z (i.e., the likelihood of the user to click-through if they are shown the ad, represented by a probability $[0, 1]$) using "user and ad features" X_1; whereas the $\mathcal{M}_2$ component assesses the "expected revenue" Y using additional "page layout features" X_2. $\mathcal{M}_1$ is typically trained by a separate team (using e.g., exogenous signals for ad quality), and the $\mathcal{M}_2$ team does not have direct access to the true Z (i.e., the true user propensities are unobservable) and imputes its training data using a prediction $\hat{Z}$ arising from a snapshot $\mathcal{M}_1^0$. The first team's goal is to predict user propensity accurately:

$$\mathcal{M}_1 : \arg\min_{\mathcal{M}} \mathbb{E}_{(X_1,Z)} \mathcal{L}_1(\hat{Z}, Z); \qquad \hat{Z} \coloneqq \mathcal{M}(X_1). \quad (1)$$

Meanwhile the second team optimizes to predict revenue:

$$\mathcal{M}_2 : \arg\min_{\mathcal{M}} \mathbb{E}_{(X_2,Z,Y)} \mathcal{L}_2(\hat{Y}, Y); \qquad \hat{Y} \coloneqq \mathcal{M}(X_2, Z). \quad (2)$$

The per-sample losses $\mathcal{L}_1, \mathcal{L}_2$ can be any classification or regression loss, and we allow $\mathcal{M}_2$ to be either a prediction component or a decision-making policy.

$$\begin{aligned} \mathcal{L}(\hat{Y}, Y) &\coloneqq (Y - \hat{Y})^2 && \text{Regression,} \\ &\coloneqq Y \text{softmax}(\hat{Y}) && \text{Classification.} \end{aligned}$$

During pipeline operation, the $\mathcal{M}_1$ and $\mathcal{M}_2$ components can be updated for several reasons (more data, better model

classes, hyper-parameter optimization, etc.). Although each component uses a component-specific loss function $\{\mathcal{L}_1, \mathcal{L}_2\}$, these losses depend on other components in general. This is because when training on data which does not have all the inputs and outputs of a particular component annotated, we have to impute values using the other components. For example, the training data available for $\mathcal{M}_2$ only contains (x_1, x_2, y) without the corresponding z, and so the learning objective for $\mathcal{M}_2$ (Equation 2) cannot be implemented directly. $\mathcal{M}_2$ thus imputes $\hat{Z}$ using a $\mathcal{M}_1^0$ snapshot $\hat{z} := \mathcal{M}_1^0(x_1)$.

$$\mathcal{M}_2 : \arg\min_{\mathcal{M}} \mathbb{E}_{(X_2,\hat{Z},Y)} \mathcal{L}_2(\hat{Y}, Y); \qquad (3)$$
$$\hat{Y} := \mathcal{M}(X_2, \hat{Z}), \quad \hat{Z} := \mathcal{M}_1^0(X_1).$$

When training $\mathcal{M}_2$ on offline data, we can verify that a new snapshot $\mathcal{M}_2^0 \mapsto \mathcal{M}_2^1$ is indeed better by checking:

$$\mathcal{L}_2(\mathcal{M}_2^1 \circ \mathcal{M}_1^0) \overset{?}{\leq} \mathcal{L}_2(\mathcal{M}_2^0 \circ \mathcal{M}_1^0).$$

For instance, this is easily assured when $\mathcal{M}_2^0$ is in the model class $\mathcal{M}$. However an independent update of $\mathcal{M}_1$ can pose a strong threat to generalization of $\mathcal{M}_\in$, so "better" $\mathcal{M}_2$ are typically validated using online A/B tests also such as,

$$\mathcal{L}_2(\mathcal{M}_2^1 \circ \mathcal{M}_1^1) \overset{?}{\leq} \mathcal{L}_2(\mathcal{M}_2^0 \circ \mathcal{M}_1^1),$$

where $\mathcal{M}_1$ has been updated independently of $\mathcal{M}_2$. Note that the $\mathcal{M}_1$ snapshot used during $\mathcal{M}_2$ training can differ from the A/B test ($\mathcal{M}_1^1 \neq \mathcal{M}_1^0$) and so online validation can reject what appeared to be "better" $\mathcal{M}_2$ on offline data.

Figure 1c shows an example of a "failed" model update. Suppose we start with an $\mathcal{M}_2^{-1}$ that is blind to Z and learns a linear mapping $\hat{Y} := X_2 + \text{constant}$. Updating $\mathcal{M}_2$ as in Figure 1b gives an $\mathcal{M}_2^0$ that is substantially better than $\mathcal{M}_2^{-1}$ but is coupled with $\mathcal{M}_1^0$. Then, if $\mathcal{M}_1$ is independently updated, we find in an A/B test that $\mathcal{M}_2^0 \circ \mathcal{M}_1^1$ is in fact worse than $\mathcal{M}_2^{-1} \circ \mathcal{M}_1^1$.

3. A Consistent Training Protocol

We seek a training procedure for ML components in a pipeline that provides all of the properties in Table 1. To build intuition, consider the Data Generating Process (DGP) of Figure 1a. Define $\mathcal{M}_1^\star$ as the DGP[1] for Z and $\mathcal{M}_2^\star$ as the DGP for Y. Observe that $\{\mathcal{M}_1^\star, \mathcal{M}_2^\star\}$ do not depend on the current pipeline or on other components' snapshots, and are instead defined solely by the underlying DGP. Note also that their composition $\mathcal{M}_2^\star \circ \mathcal{M}_1^\star$ is the Bayes-optimal predictor given the overall pipeline's inputs, $Y \equiv \mathcal{M}_2^\star \circ \mathcal{M}_1^\star \mid X_1, X_2$.

[1]Realistically, $\{Z, Y\}$ can depend on unobserved variables. In general, interpret $\mathcal{M}_1^\star \equiv \Pr(Z \mid X_1); \mathcal{M}_2^\star \equiv \Pr(Y \mid X_2, Z)$.

Recall the learning objectives of Equation 1 and 2. If the model classes are rich enough to contain $\{\mathcal{M}_1^\star, \mathcal{M}_2^\star\}$ respectively, we observe that each component independently and consistently optimizes[2] to its Bayes-optimal predictor $\mathcal{M}_1 \mapsto \mathcal{M}_1^\star; \mathcal{M}_2 \mapsto \mathcal{M}_2^\star$. Unfortunately, Equation 2 cannot be implemented because the data for $\mathcal{M}_2$ contains imputed $\hat{Z} \neq Z$. Similarly, if $\mathcal{M}_1$ is fine-tuned on data without Z annotations, Equation 1 cannot be implemented.

Figure 2a indicates the observed random variables during $\mathcal{M}_2$ training in black. We already see that Equation 2 compares the predictions of $\mathcal{M}_2$ on the unobserved Z (denoted in red) against the observed Y to drive $\mathcal{M}_2 \mapsto \mathcal{M}_2^\star$. Instead, define $\tilde{y} := \mathcal{M}_2^\star(x_2, \hat{z})$ (i.e., what would be the DGP output if we forced the Z variable to be $\hat{z}$). Consider the objective:

$$\mathcal{M}_2 : \arg\min_{\mathcal{M}} \mathbb{E}_{(X_2,\hat{Z},Y)} \mathcal{L}_2(\hat{Y}, \tilde{Y}); \qquad (4)$$
$$\hat{Y} := \mathcal{M}(X_2, \hat{Z}), \quad \tilde{Y} := \mathcal{M}_2^\star(X_2, \hat{Z}).$$

Note that when $\mathcal{M}_2^\star$ is in the model class, this objective also drives $\mathcal{M}_2 \mapsto \mathcal{M}_2^\star$. Unfortunately, $\tilde{Y}$ also remains unobserved during $\mathcal{M}_2$ training. We replace the unobserved $\{\tilde{Y}\}$ with observable surrogates, and include correction terms to capture the sensitivity of Equation 4 to their errors. Consider the Taylor expansion[3] of $\mathcal{L}_2(\hat{Y}, Y)$ around Y:

$$\mathcal{L}(\mathcal{M}(X_2, \hat{Z}), \tilde{Y}) \approx \mathcal{L}(\mathcal{M}(X_2, \hat{Z}), Y) + \frac{\partial \mathcal{L}}{\partial Y}\left[\tilde{Y} - Y\right]. \qquad (5)$$

The first term is the typical training objective of $\mathcal{M}_2$ with imputed $\hat{Z}$ data (Equation 3), but we see that an additional *modularity regularizer*[4] is needed to ensure that $\mathcal{M}_2 \mapsto \mathcal{M}_2^\star$. When $\mathcal{M}_1^0$ is such that it causes small end-to-end pipeline error (i.e., $\tilde{Y} \approx Y$), the modularity regularizer is 0. Otherwise, $\mathcal{M}_2$ training with $\hat{Z} \neq Z$ must "residualize" the prediction targets $Y \mapsto \{\tilde{Y} \mid \hat{Z}\}$ to not over-fit to $\mathcal{M}_1^0$.

The difference $\left[\tilde{Y} - Y\right]$ is akin to the conditional average treatment effect (CATE) (Semenova & Chernozhukov, 2021) with X_2 as the condition and treatment being $\left[\hat{Z} \text{ or } Z\right]$ and the outcome as $\left[\tilde{Y} \text{ or } Y\right]$. For squared loss and discrete X_2, we can easily derive a closed form for Equation 5 using a matching-based CATE estimator.

[2]In fact, in the asymptotic limit of infinite data the $\{\mathcal{M}_1^\star, \mathcal{M}_2^\star\}$ are the argmin models of their respective objectives no matter what snapshots of the other components are in use.

[3]The first order expansion is sufficiently accurate for several loss functions. For arbitrary losses, we may need additional terms of the expansion for an accurate surrogate.

[4]Taylor expansion around $\hat{Y}$ yields another modularity regularizer that can be beneficial for Lipschitz-smooth model classes.

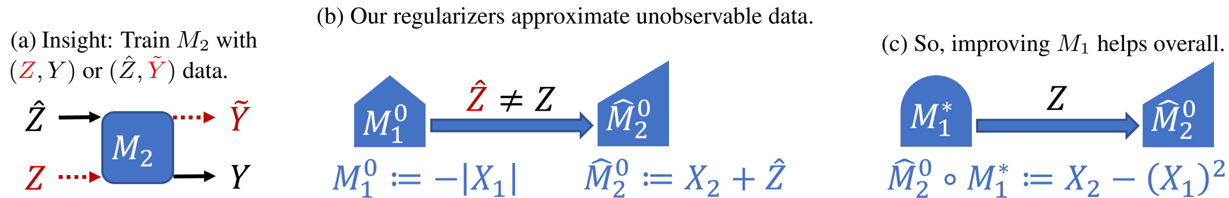

Figure 2: Demonstration of decoupling a 2-module pipeline. (a) $\mathcal{M}_2$ training only sees predictions $\hat{Z}$ and the true Y. $\tilde{Y} := \mathcal{M}_2^\star(\hat{Z})$ and true Z are never observed. If we somehow train $\mathcal{M}_2$ on $\{z, y\}$ data or on $\{\hat{z}, \tilde{y}\}$ data, then $\mathcal{M}_2 \mapsto \mathcal{M}_2^\star$ independent of any $\mathcal{M}_1$. (b) Comparing to Figure 1b, $\hat{\mathcal{M}}_2^0$ is regularized to avoid correcting $\mathcal{M}_1^0$ mistakes. (c) While $\hat{\mathcal{M}}_2^0 \circ \mathcal{M}_1^0$ is worse than Figure 1b, we are guaranteed that $\mathcal{M}_1^1$ improvements will always improve the end-to-end pipeline.

4. Experiments

We conduct experiments on synthetic datasets to answer: (1) Does the modularity regularizer indeed optimize the unobservable Equation 2? (2) How do the different pipeline training approaches fare on the three properties outlined in Table 1? For additional details, please refer to our code which will be made available before ICML 2023.

DGP We consider Figure 1a where $X_1 \in \mathbb{R}^{10}, X_2 \in \mathbb{R}^5, Z \in \mathbb{R}^3, Y \in \mathbb{R}$ and both $\mathcal{M}_1^*$ and $\mathcal{M}_2^*$ are realized by randomly initialized fully-connected neural networks with ReLU activations.

Pipeline Given that both Z and Y are real-valued outputs, we use mean squared loss for training both $\mathcal{M}_1$ and $\mathcal{M}_2$. $\mathcal{M}_1^0$ uses an under-parameterized function class, and is then updated using an over-parameterized class to simulate a consistent update $\mathcal{M}_1^1 \approx \mathcal{M}_1^*$. The incremental update $\mathcal{M}_1^1 \approx \mathcal{M}_1^0$ is implemented by adding a proximal l_2 norm regularizer during the $\mathcal{M}_1$ training so that $\mathcal{M}_1(x_1) \approx \mathcal{M}_1^0(x_1)$. Given the initial $\mathcal{M}_1^0$ as a snapshot, we train three different $\mathcal{M}_2^0$: (i) $\mathcal{M}_2^{\text{async}}$ which is the best fit model in $\mathcal{M}_2$'s class given $(X_2, \hat{Z})$, (ii) $\mathcal{M}_2^{\text{causal}}$ using the modularity regularizer in Equation 5[5] and (iii) $\mathcal{M}_2^{\text{prox}}$ which uses a distributionally robust training objective by using randomly perturbed $\hat{Z}$ inputs (so as to accommodate small distribution shifts in case $\mathcal{M}_1$ updates). As a sanity check we use the DGP to compute the counterfactual output $\tilde{Y}$ in the modularity regularizer of Equation 5. In practice, these counterfactual values are not directly available and need to be estimated from data. We anticipate that our proof-of-concept here will inspire future work on CATE-like estimators for tackling general versions of Equation 5.

Results Table 2 shows the test mean squared error for the consistent and incremental updates $\mathcal{M}_1^1$ relative to $\mathcal{M}_1^0$ error (normalized to a value of 100). We see that using an under-parameterized class causes high test loss for $\mathcal{M}_1$ (= 100), which nearly vanishes for consistent updates but remains high for incremental updates, across 10 independent runs.

Consistent $\mathcal{M}_1$ ($\approx \mathcal{M}_1^*$)	Incremental $\mathcal{M}_1$
0.642 ± 0.334	46.094 ± 0.449

Table 2: $\mathcal{M}_1$ component loss on its test set after two kinds of updates, normalized w.r.t. initial $\mathcal{M}_1$ loss. See Figure 3 for $\mathcal{M}_2$ results with these $\mathcal{M}_1$.

For each $\mathcal{M}_1$ snapshot, we evaluate all four approaches for pipeline optimization with the final pipeline mean squared error reported in Figure 3. We again show errors on a relative scale where the loss of $\mathcal{M}_2^\star \circ \mathcal{M}_1^0$ is normalized to 100. We see that synchronized updates to $\mathcal{M}_2$ (i.e., using the updated snapshots of $\mathcal{M}_1$) yield the best pipeline loss for each $\mathcal{M}_1$ update. For the initial $\mathcal{M}_1^0$, $\mathcal{M}_2^{\text{sync}}$ and $\mathcal{M}_2^{\text{async}}$ are identical, while $\mathcal{M}_2^{\text{prox}}$ has very similar performance. When $\mathcal{M}_1$ is updated to its locally optimal $\mathcal{M}_1^\star$ (*consistency*), $\mathcal{M}_2^{\text{causal}}$ achieves similar performance as the synchronized $\mathcal{M}_2$, indicating that it can recover the underlying DGP $\mathcal{M}_2^\star$ for Y. $\mathcal{M}_2^{\text{prox}}$ performs better than $\mathcal{M}_2^{\text{async}}$ because the perturbations it saw during training avoids overfitting to the specific $\hat{Z}$ from $\mathcal{M}_1^0$. When $\mathcal{M}_1$ is updated *incrementally*, the pipeline performances are all aligned, with $\mathcal{M}_2^{\text{prox}}$ leading to slightly better performance than the $\mathcal{M}_2^{\text{async}}$ loss. Sufficiently big improvements to $\mathcal{M}_1$ (Table 2) can mask the misalignment due to model coupling, and Figure 1c shows that async is not aligned in general. The observations from these experiments validate the properties outlined in Table 1. When we expect large updates from other pipeline components, we recommend the causal approach so as to produce consistent pipeline updates.

5. Discussion

We emphasize the need for causal modeling in pipeline design, and demonstrate an approach that guarantees pipeline consistency when training components independently. Our experiments demonstrate the subtle trade-offs in modular ML pipeline design. We conjecture that our causal approach and the proximal approach can be fruitfully combined to

[5]The first-order Taylor expansion in Equation 5 is exact, not an approximation, for the mean squared loss.

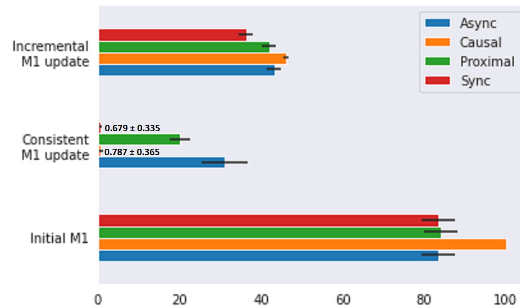

Figure 3: Pipeline optimization approaches evaluated on test set after different $\mathcal{M}_1$ updates. $\mathcal{M}_2^\star \circ \mathcal{M}_1^0$ loss is normalized to 100. Causal approach for $\mathcal{M}_2$ is best when $\mathcal{M}_1$ undergoes a big update.

produce modular ML pipelines. Our findings can be generalized to arbitrary DAG pipelines and decision-making settings, expanding their relevance and potential impact.

References

Bottou, L., Peters, J., Quiñonero-Candela, J., Charles, D. X., Chickering, D. M., Portugaly, E., Ray, D., Simard, P., and Snelson, E. Counterfactual reasoning and learning systems: The example of computational advertising. *JMLR*, 14(11), 2013.

Hron, J., Krauth, K., Jordan, M., and Kilbertus, N. On component interactions in two-stage recommender systems. In *NeurIPS*, 2021.

Hu, W., Bansal, R., Cao, K., Rao, N., Subbian, K., and Leskovec, J. Learning backward compatible embeddings. In *KDD*, 2022.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. Hidden technical debt in machine learning systems. In *NeurIPS*, 2015.

Semenova, V. and Chernozhukov, V. Debiased machine learning of conditional average treatment effects and other causal functions. *Econometrics*, 24(2), 2021.

Wu, R., Guo, C., Hannun, A., and van der Maaten, L. Fixes that fail: Self-defeating improvements in machine-learning systems. In *NeurIPS*, 2021.