The Journal of Open Source Software

idinn: A Python Package for Inventory-Dynamics Control with Neural Networks

Jiawei Li [1¶], **Thomas Asikis** [2], **Ioannis Fragkos**[3], **and Lucas Böttcher** [1,4]

1 Department of Computational Science and Philosophy, Frankfurt School of Finance & Management **2** Game Theory, University of Zurich **3** Department of Technology and Operations Management, Rotterdam School of Management, Erasmus University Rotterdam **4** Laboratory for Systems Medicine, Department of Medicine, University of Florida ¶ Corresponding author

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

Identifying optimal policies for replenishing inventory from multiple suppliers is a key problem in inventory management. Solving such optimization problems requires determining the quantities to order from each supplier based on the current inventory and outstanding orders, minimizing the expected ordering, holding, and out-of-stock costs. Despite over 60 years of extensive research on inventory management problems, even fundamental dual-sourcing problems—where orders from an expensive supplier arrive faster than orders from a low-cost supplier—remain analytically intractable (Barankin, 1961; Fukuda, 1964). Additionally, there is a growing interest in optimization algorithms that can handle real-world inventory problems with non-stationary demand (Song et al., 2020).

We provide a Python package, `idinn`, which implements inventory dynamics-informed neural networks designed to control both single-sourcing and dual-sourcing problems. In single-sourcing problems, a single supplier delivers an ordered quantity to the firm within a known lead time (the time it takes for orders to arrive) and at a known unit cost (the cost of ordering a single item). Dual-sourcing problems are more complex. In dual-sourcing problems, the company has two potential suppliers of a product, each with different known lead times and unit costs. The company's decision problem is to determine the quantity to order from each of the two suppliers at the beginning of each period, given the history of past orders and the current inventory level. The objective is to minimize the expected order, inventory, and out-of-stock costs over a finite or infinite horizon. idinn implements neural network controllers and inventory dynamics as customizable objects using PyTorch as the backend, allowing users to identify near-optimal ordering policies for their needs with reasonable computational resources.

The methods used in `idinn` take advantage of advances in automatic differentiation (Paszke et al., 2019, 2017) and the growing use of neural networks in dynamical system identification (Chen et al., 2018; Fronk & Petzold, 2023; Wang & Lin, 1998) and control (Asikis et al., 2022; Böttcher et al., 2022, 2024; Böttcher, 2023; Böttcher & Asikis, 2022; Mowlavi & Nabi, 2023).

Statement of need

Inventory management problems arise in many industries, including manufacturing, retail, hospitality, fast fashion, warehousing, and energy. A fundamental but analytically intractable inventory management problem is dual sourcing (Barankin, 1961; Fukuda, 1964; Xin & Van Mieghem, 2023). `idinn` is a Python package for controlling dual-sourcing inventory dynamics with dynamics-informed neural networks. The classical dual-sourcing problem we consider is usually formulated as an infinite-horizon problem focusing on minimizing average cost while considering stationary stochastic demand. Using neural networks, we minimize costs over

multiple demand trajectories. This approach allows us to address not only non-stationary demand, but also finite-horizon and infinite-horizon discounted problems. Unlike traditional reinforcement learning approaches, our optimization approach takes into account how the system to be optimized behaves over time, leading to more efficient training and accurate solutions.

Training neural networks for inventory dynamics control presents a unique challenge. The adjustment of neural network weights during training relies on propagating real-valued gradients, while the neural network outputs - representing replenishment orders - must be integers. To address this challenge in optimizing a discrete problem with real-valued gradient descent learning algorithms, we apply a problem-tailored straight-through estimator (Asikis, 2023; Dyer et al., 2023; Yang et al., 2022). This approach enables us to obtain integer-valued neural network outputs while backpropagating real-valued gradients.

`idinn` has been developed for researchers, industrial practitioners and students working at the intersection of optimization, operations research, and machine learning. It has been made available to students in a machine learning course at the Frankfurt School of Finance & Management, as well as in a tutorial at California State University, Northridge, showcasing the effectiveness of artificial neural networks in solving real-world optimization problems. In a previous publication (Böttcher et al., 2023), a proof-of-concept codebase was used to compute near-optimal solutions of dozens of dual-sourcing instances.

Example usage

Single-sourcing problems

The overarching goal in single-sourcing and related inventory management problems is for companies to identify the optimal order quantities to minimize inventory-related costs, given stochastic demand. During periods when inventory remains after demand is satisfied, each unit of excess inventory incurs a holding cost h. If demand exceeds available inventory in one period, the excess demand incurs an out-of-stock cost b. To solve this problem using `idinn`, we first initialize the sourcing model and its associated neural network controller. Then, we train the neural network controller using costs generated by the sourcing model. Finally, we can use the trained neural network controller to compute near-optimal order quantities that depend on the state of the system.

Initialization

We use the 'SingleSourcingModel' class to initialize a single-sourcing model. The single-sourcing model considered in this example has a lead time of 0 (i.e., the order arrives immediately after it is placed) and an initial inventory of 10. The holding cost, h, and the out-of-stock cost, b, are 5 and 495, respectively. Demand is generated from a discrete uniform distribution within $[1, 4]$. We use a batch size of 32 to train the neural network, i.e., the sourcing model generates 32 samples simultaneously. In code, the sourcing model is initialized as follows.

```
import torch
from idinn.sourcing_model import SingleSourcingModel
from idinn.controller import SingleSourcingNeuralController
from idinn.demand import UniformDemand

single_sourcing_model = SingleSourcingModel(
  lead_time=0,
  holding_cost=5,
  shortage_cost=495,
  batch_size=32,
  init_inventory=10,
```

```
    demand_generator=UniformDemand(low=1, high=4),
)
```

The cost at period t, c_t, is therefore

$$c_t = h \max(0, I_t) + b \max(0, -I_t)\,,$$

where I_t is the inventory level at the end of period t. The higher the holding cost, the more costly it is to keep inventory positive and high. The higher the out-of-stock cost, the more costly it is to run out of stock when the inventory level is negative. The joint holding and out-of-stock costs for a period can be calculated using the `get_cost()` method of the sourcing model.

```
single_sourcing_model.get_cost()
```

The expected output is as follows.

```
tensor([[50.],
        ...,
        [50.]], grad_fn=<AddBackward0>)
```

In this example, this function should return 50 for each sample because the initial inventory is 10 and the holding cost is 5. In this case, we have 32 samples because we specified a batch size of 32.

For single-sourcing problems, we initialize the neural network controller using the `SingleSourcingNeuralController` class. For illustration purposes, we use a simple neural network with 1 hidden layer and 2 neurons. The activation function is `torch.nn.CELU(alpha=1)`.

```
single_controller = SingleSourcingNeuralController(
    hidden_layers=[2],
    activation=torch.nn.CELU(alpha=1)
)
```

Training

Although the neural network controller has not yet been trained, we can still compute the total cost associated with its ordering policy. To do this, we integrate it with our previously specified sourcing model and calculate the total cost for 100 periods using `get_total_cost()`.

The `get_total_cost()` function calculates the sum of the costs over a given number of sourcing periods. Within each period, three events occur. First, the current inventory, I_t, and the history of past orders that have not yet arrived, i.e., the vector $(q_{t-1}, q_{t-2}, \dots, q_{t-l})$, are used as inputs for the controller to calculate the order quantity, q_t. Second, the previous order quantity q_{t-l} arrives. Third, the demand for the current period, d_t, is realized, resulting in a new inventory level, $I_t + q_{t-l} - d_t$. Using the updated inventory, the cost for the individual period, c_t, is calculated according to the equation above, and the costs of each period are summed up as the total cost. The interested reader is referred to Böttcher et al. (2023) for further details.

```
single_controller.get_total_cost(
    sourcing_model=single_sourcing_model,
    sourcing_periods=100
)
```

A sample output is as follows.

```
tensor(5775221.5000, grad_fn=<AddBackward0>)
```

Not surprisingly, the very high cost indicates that the model's performance is poor, since we are only using a untrained neural network, where the weights are just (pseudo) random numbers. We can train the neural network controller using the `fit()` method, where the training data is generated from the given sourcing model. To better monitor the training process, we specify the `tensorboard_writer` parameter to log both the training loss and the validation loss. For reproducibility, we also specify the seed of the underlying random number generator using the seed parameter.

```
from torch.utils.tensorboard import SummaryWriter

single_controller.fit(
    sourcing_model=single_sourcing_model,
    sourcing_periods=50,
    validation_sourcing_periods=1000,
    epochs=5000,
    seed=1,
    tensorboard_writer=SummaryWriter(comment="_single_1")
)
```

After training, we can use the trained neural network controller to calculate the total cost for 100 periods using our previously specified sourcing model. The total cost should be significantly lower than the cost associated with the untrained model.

```
single_controller.get_total_cost(
  sourcing_model=single_sourcing_model,
  sourcing_periods=100
)
```

A sample output is shown below.

```
tensor(820., grad_fn=<AddBackward0>)
```

Order calculation

We can then calculate optimal orders using the trained model.

```
# Calculate the optimal order quantity for applications
single_controller.forward(
  current_inventory=10,
  past_orders=[1, 5]
)
```

The expected output is as follows.

```
tensor([[0.]], grad_fn=<SubBackward0>)
```

Dual-sourcing problems

Solving dual-sourcing problems with `idinn` is similar to the workflow for single-sourcing problems described in the previous section. The main difference is that the cost calculation includes the order costs of different suppliers.

Initialization

To solve dual-sourcing problems, we use `DualSourcingModel` and `DualSourcingNeuralController`, which are responsible for setting up the sourcing model and its corresponding controller. In this example, we examine a dual-sourcing model characterized by the following parameters: the regular order lead time is 2; the expedited order lead time is 0; the regular order cost, c_r, is 0; the expedited order cost, c_e, is 20; and the initial inventory is 6. In addition, the holding cost,

h, and the out-of-stock cost, b, are 5 and 495, respectively. The demand is generated from a discrete uniform distribution bounded on $[1, 4]$. In this example, we use a batch size of 256.

```
import torch
from idinn.sourcing_model import DualSourcingModel
from idinn.controller import DualSourcingNeuralController
from idinn.demand import UniformDemand

dual_sourcing_model = DualSourcingModel(
    regular_lead_time=2,
    expedited_lead_time=0,
    regular_order_cost=0,
    expedited_order_cost=20,
    holding_cost=5,
    shortage_cost=495,
    batch_size=256,
    init_inventory=6,
    demand_generator=UniformDemand(low=1, high=4),
)
```

The cost at period t, c_t, is

$$c_t = c_r q_t^r + c_e q_t^e + h \max(0, I_t) + b \max(0, -I_t)\,,$$

where I_t is the inventory level at the end of period t, q_t^r is the regular order placed in period t, and q_t^e is the expedited order placed in period t. The higher the holding cost, the more expensive it is to keep inventory positive and high. The higher the out-of-stock cost, the more expensive it is to run out of stock when inventory is negative. The higher the regular and expedited order costs, the more expensive it is to place those orders. The cost can be calculated using the `get_cost()` method of the sourcing model.

```
dual_sourcing_model.get_cost(regular_q=0, expedited_q=0)
```

The output that is expected is as follows.

```
tensor([[30.],
        ...,
        [30.]], grad_fn=<AddBackward0>)
```

In this example, this function should return 30 for each sample because the initial inventory is 6, the holding cost is 5, and there is neither a regular nor an expedited order. In this case, we have 256 samples because we specified a lot size of 256.

For dual-sourcing problems, we initialize the neural network controller using the `DualSourcingNeuralController` class. We use a simple neural network with 6 hidden layers. The number of neurons in each layer is 128, 64, 32, 16, 8, and 4, respectively. The activation function is `torch.nn.CELU(alpha=1)`.

```
dual_controller = DualSourcingNeuralController(
    hidden_layers=[128, 64, 32, 16, 8, 4],
    activation=torch.nn.CELU(alpha=1)
)
```

Training

Similar to the single-sourcing case, the cost over all periods can be calculated using the controller's `get_total_cost()` method. The inputs to the controller are the inventory level, I_t, and the history of past orders. However, since there are now two suppliers in the system, we need to include the order history of both suppliers. Therefore, the inputs for the past orders

should be written as $(q^r_{t-1}, \dots, q^r_{t-l_r}, q^e_{t-1}, \dots, q^e_{t-l_e})$. The cost for each period is calculated in a similar way as in single-sourcing models: past orders arrive, new orders are placed, and demand is realized. Then the costs for each period are summed to calculate the total cost. The interested reader is referred to Böttcher et al. (2023) for more details.

```
dual_controller.get_total_cost(
    sourcing_model=single_sourcing_model,
    sourcing_periods=100
)
```

A sample output is as follows.

```
tensor(5878623., grad_fn=<AddBackward0>)
```

In the same way as in the previous section, we can train the neural network controller using the `fit()` method.

```
from torch.utils.tensorboard import SummaryWriter

dual_controller.fit(
    sourcing_model=dual_sourcing_model,
    sourcing_periods=50,
    validation_sourcing_periods=1000,
    epochs=1000,
    tensorboard_writer=SummaryWriter(comment="_dual_1234"),
    seed=1234
)
```

After training, we can again use the trained neural network controller to calculate the total cost. The total cost should be significantly lower than the cost associated with the untrained model.

```
dual_controller.get_total_cost(
    sourcing_model=dual_sourcing_model,
    sourcing_periods=100
)
```

The following is a sample output.

```
tensor(1940.0391, grad_fn=<AddBackward0>)
```

Order calculation

Then we can use the trained network to compute near-optimal orders.

```
# Calculate the optimal order quantity for applications
regular_q, expedited_q = dual_controller.forward(
    current_inventory=10,
    past_regular_orders=[1, 5],
    past_expedited_orders=[0, 0],
)
```

Other utility functions

The `idinn` package provides several utility functions for both the `SingleSourcingModel` and `DualSourcingModel` class.

To further examine the controller's performance in the specified sourcing environment, users can plot the inventory and order histories.

```
# Simulate and plot the results
dual_controller.plot(
  sourcing_model=dual_sourcing_model,
  sourcing_periods=100
)
```

In addition to random demands generated by uniform distributions, users can also provide demands in the format of `python` lists, `numpy` arrays and `torch` tensors. For example, the following code generates demands with values 1, 2,..., 10 that repeat every 10 periods.

```
from idinn.demand import CustomDemand

dual_sourcing_model = DualSourcingModel(
    regular_lead_time=2,
    expedited_lead_time=0,
    regular_order_cost=0,
    expedited_order_cost=20,
    holding_cost=5,
    shortage_cost=495,
    batch_size=256,
    init_inventory=6,
    demand_generator=CustomDemand(
      [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    ),
)
```

The `idinn` package also provides functions for saving and loading model checkpoints. To save and load a given model, one can use the `save()` and `load()` methods, respectively.

```
# Save the model
dual_controller.save("optimal_dual_sourcing_controller.pt")
# Load the model
dual_controller_loaded = DualSourcingNeuralController(
    hidden_layers=[128, 64, 32, 16, 8, 4],
    activation=torch.nn.CELU(alpha=1),
)
dual_controller_loaded.init_layers(
    regular_lead_time=2,
    expedited_lead_time=0,
)
dual_controller_loaded.load("optimal_dual_sourcing_controller.pt")
```

Acknowledgements

LB acknowledges financial support from hessian.AI and the Army Research Office (grant W911NF-23-1-0129). TA acknowledges financial support from the Schweizerischer Nationalfonds zur Förderung der Wissenschaftlichen Forschung through NCCR Automation (grant P2EZP2 191888).

References

Asikis, T. (2023). Towards recommendations for value sensitive sustainable consumption. *NeurIPS 2023 Workshop on Tackling Climate Change with Machine Learning: Blending New and Existing Knowledge Systems*. https://nips.cc/virtual/2023/76939

Asikis, T., Böttcher, L., & Antulov-Fantulin, N. (2022). Neural ordinary differential equation control of dynamics on graphs. *Physical Review Research*, *4*(1), 013221.

Barankin, E. (1961). A delivery-lag inventory model with an emergency provision. *Naval Research Logistics Quarterly*, *8*, 285–311.

Böttcher, L. (2023). Gradient-free training of neural ODEs for system identification and control using ensemble Kalman inversion. *ICML Workshop on New Frontiers in Learning, Control, and Dynamical Systems, Honolulu, HI, USA, 2023*.

Böttcher, L., Antulov-Fantulin, N., & Asikis, T. (2022). AI Pontryagin or how artificial neural networks learn to control dynamical systems. *Nature Communications*, *13*(1), 1–9.

Böttcher, L., & Asikis, T. (2022). Near-optimal control of dynamical systems with neural ordinary differential equations. *Machine Learning: Science and Technology*, *3*(4), 045004.

Böttcher, L., Asikis, T., & Fragkos, I. (2023). Control of dual-sourcing inventory systems using recurrent neural networks. *INFORMS Journal on Computing*, *35*(6), 1308–1328.

Böttcher, L., Fonseca, L. L., & Laubenbacher, R. C. (2024). Control of medical digital twins with artificial neural networks. *arXiv Preprint arXiv:2403.13851*.

Chen, T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2018). Neural ordinary differential equations. In S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada* (pp. 6572–6583). https://proceedings.neurips.cc/paper/2018/hash/69386f6bb1dfed68692a24c8686939b9-Abstract.html

Dyer, J., Quera-Bofarull, A., Chopra, A., Farmer, J. D., Calinescu, A., & Wooldridge, M. J. (2023). Gradient-assisted calibration for financial agent-based models. *4th ACM International Conference on AI in Finance, ICAIF 2023, Brooklyn, NY, USA, November 27-29, 2023*, 288–296.

Fronk, C., & Petzold, L. (2023). Interpretable polynomial neural ordinary differential equations. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, *33*(4).

Fukuda, Y. (1964). Optimal policies for the inventory problem with negotiable leadtime. *Management Science*, *10*(4), 690–708.

Mowlavi, S., & Nabi, S. (2023). Optimal control of PDEs using physics-informed neural networks. *Journal of Computational Physics*, *473*, 111731.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., & Lerer, A. (2017). Automatic differentiation in PyTorch. *NIPS 2017 Autodiff Workshop*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada* (pp. 8024–8035). https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

Song, J.-S., Van Houtum, G.-J., & Van Mieghem, J. A. (2020). Capacity and inventory management: Review, trends, and projections. *Manufacturing & Service Operations Management*, *22*(1), 36–46.

Wang, Y.-J., & Lin, C.-T. (1998). Runge–Kutta neural network for identification of dynamical systems in high accuracy. *IEEE Transactions on Neural Networks*, *9*(2), 294–307.

Xin, L., & Van Mieghem, J. A. (2023). Dual-sourcing, dual-mode dynamic stochastic inventory models. In *Research Handbook on Inventory Management* (pp. 165–190). Edward Elgar Publishing.

Yang, Z., Lee, J., & Park, C. (2022). Injecting logical constraints into neural networks via straight-through estimators. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvári, G. Niu, & S. Sabato (Eds.), *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA* (Vol. 162, pp. 25096–25122). PMLR. https://proceedings.mlr.press/v162/yang22h.html