DFL in Mean-Variance Optimization

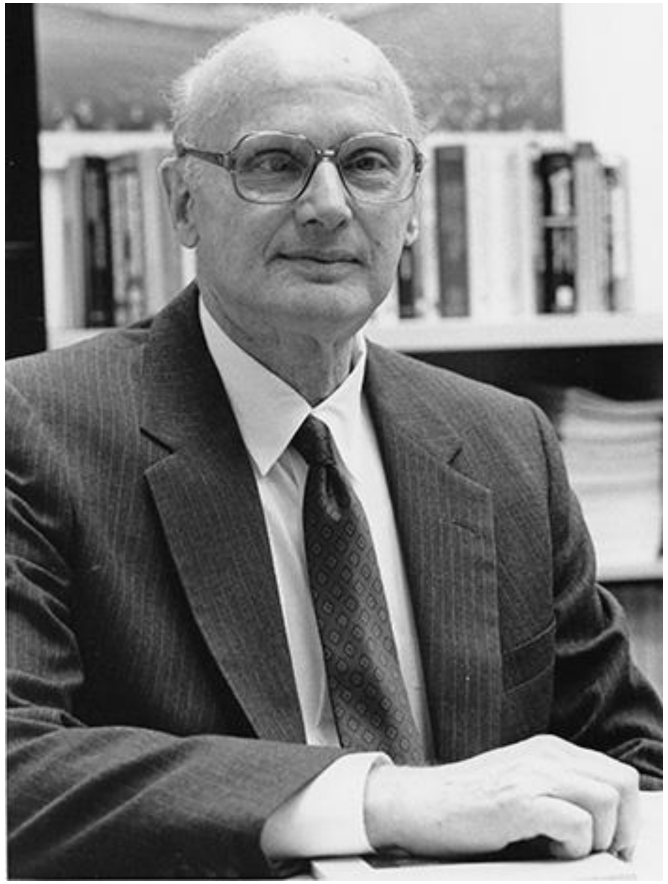
Junhyeong Lee
Financial Engineering Lab
Department of Industrial Engineering

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Korea Advanced Institute of Science and Technology

Mean-Variance Optimization

- Harry Markowitz



Harry Markowitz (1927 – 2023)

- Nobel prize winner
- Introduced the theory of portfolio selection
- Laid the foundation for Modern Portfolio Theory (MPT)

Mean-Variance Optimization

- Mean-Variance Optimization

We want to minimize **RISK**

We want to maximize **RETURN**

$$\max_{w} \quad w^{\mathrm{T}}\mu - \lambda w^{\mathrm{T}}\Sigma w$$
$$\text{subject to} \quad w^{T}\mathbf{1} = 1$$
$$w \geq 0$$

–$w \in \mathbb{R}^n$: portfolio weight of n risky assets
–$\mu \in \mathbb{R}^n$: expected returns of n risky assets
–$\Sigma \in \mathbb{R}^{n\times n}$: return covariance matrix of n risky assets
–$\mathbf{1} \in \mathbb{R}^n$: vector of ones
–$\lambda \in \mathbb{R}_{\geq 0}$: risk-aversion coefficient

DFL for MVO

$$w^*(\mu) = \underset{w}{\arg\max} \; w^{\mathrm{T}}\mu - \lambda w^{\mathrm{T}}\Sigma w$$
$$\text{subject to} \quad w^T\mathbf{1} = 1$$
$$w \geq 0$$

Consider two different μ's

- μ^*: **True** expected returns of n risky assets (unobservable in advance)

- $\hat{\mu}$: **Predicted** expected returns of n risky assets

DFL for MVO

DFL

$$\mathcal{L}_{decision}(\hat{\boldsymbol{c}}, \boldsymbol{c}) := \boldsymbol{Regret}(\boldsymbol{w}^*(\hat{\boldsymbol{c}}), \boldsymbol{c})$$

$$= \underbrace{\boldsymbol{f}(\boldsymbol{w}^*(\hat{\boldsymbol{c}}), \boldsymbol{c})}_{\text{Realized decision quality with estimated } \hat{\boldsymbol{c}}} - \underbrace{\boldsymbol{f}(\boldsymbol{w}^*(\boldsymbol{c}), \boldsymbol{c})}_{\text{Realized decision quality with ground truth } \boldsymbol{c}}$$

DFL for MVO

Define **regret** as the difference between MVO objectives with **predicted** and **true** expected returns

$$\mathcal{L}_{MVO} = Regret(w^*(\hat{\mu}), \mu^*)$$

$$= (w^*(\hat{\mu})^{\mathrm{T}}\mu^* - \lambda w^*(\hat{\mu})^{\mathrm{T}}\Sigma w^*(\hat{\mu}))$$

$$-(w^*(\mu^*)^{\mathrm{T}}\mu^* - \lambda w^*(\mu^*)^{\mathrm{T}}\Sigma w^*(\mu^*))$$

MVO objective
Decision: prediction
Evaluation: true

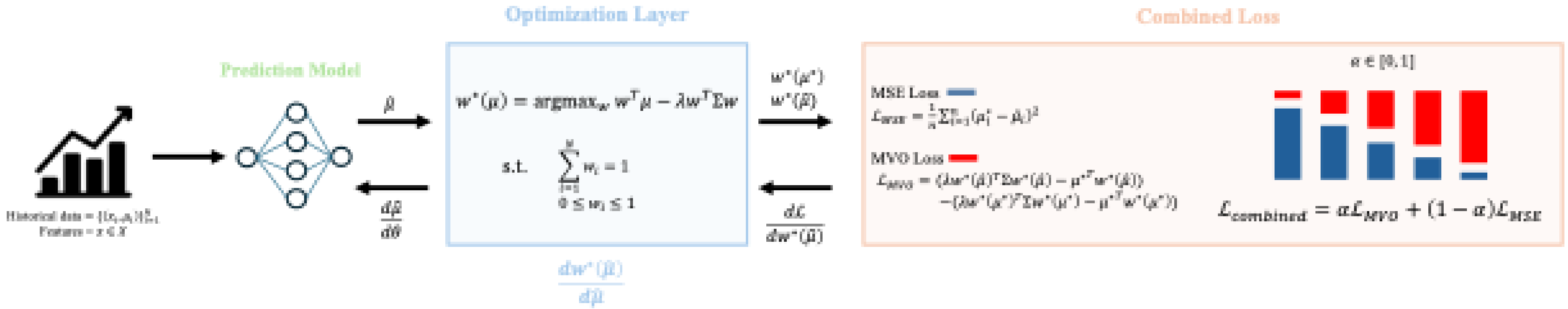
MVO objective
Decision: true
Evaluation: true

Consider two different μ's

- μ^*: **True** expected returns of n risky assets
  (unobservable in advance)

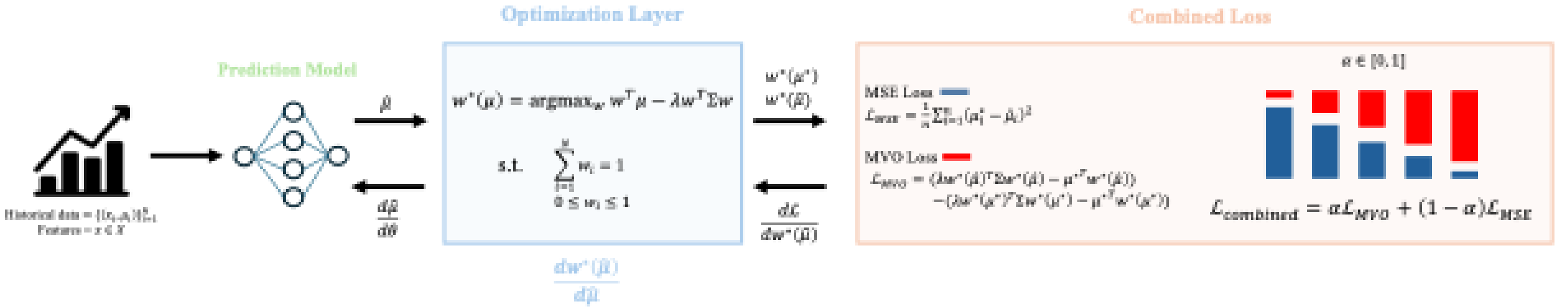
- $\hat{\mu}$: **Predicted** expected returns of n risky assets

DFL for MVO



Optimization Layer
Combined Loss
Prediction Model
$w^*(\mu) = \text{argmax}_w\, w^T\mu - \lambda w^T\Sigma w$
s.t.
MSE Loss
MVO Loss
$\mathcal{L}_{combined} = \alpha\mathcal{L}_{MVO} + (1-\alpha)\mathcal{L}_{MSE}$

DFL for MVO



$$\frac{d\mathcal{L}(w^*(\hat{\mu}),\mu)}{d\theta} = \frac{d\mathcal{L}(w^*(\hat{\mu}),\mu)}{dw^*(\hat{\mu})} \frac{\boldsymbol{dw^*(\hat{\mu})}}{\boldsymbol{d\hat{\mu}}} \frac{d\hat{\mu}}{d\theta}$$

$\frac{\boldsymbol{dw^*(\hat{\mu})}}{\boldsymbol{d\hat{\mu}}}$ is particularly tricky

$w^*(\hat{c})$ may not be uniquely defined or differentiable

$\Rightarrow$ We use ***CvxpyLayers*** to backpropagate gradients through the optimization problem.

(compute $\frac{d\mathcal{L}}{d\theta}$ via implicit differentiation.)

DFL for MVO

- Hands-on session

Model: 3 Layers MLP

```python
# Experiment configuration
config = {
    'tickers': ['AAPL', 'MSFT', 'GOOGL', 'AMZN', 'META',
                'NVDA', 'TSLA', 'JPM', 'V', 'JNJ',
                'WMT', 'PG', 'DIS', 'MA', 'UNH'],  # 15 stocks
    'start_date': '2020-01-01',
    'end_date': '2024-12-31',
    'lookback_window': 60,
    'prediction_horizon': 1,
    'train_split': 0.7,
    'val_split': 0.15,
    'batch_size': 16,
    'epochs': 100,
    'patience': 5,
    'learning_rate': 0.001,
    'lambda_risk': 1.0,
    'alpha_values': [0.0, 0.5],  # Only Two-Stage and Hybrid
    'device': 'cuda' if torch.cuda.is_available() else 'cpu'
}
```

DFL for MVO

- Hands-on session

```python
class StockDataset(Dataset):
    """Dataset class for stock returns with rolling windows"""

    def __init__(self, returns_data: pd.DataFrame, lookback_window: int = 60,
                 prediction_horizon: int = 1):
        self.returns = returns_data.values
        self.dates = returns_data.index
        self.stock_names = returns_data.columns
        self.lookback_window = lookback_window
        self.prediction_horizon = prediction_horizon
        self.n_stocks = len(self.stock_names)
        self.valid_indices = list(range(lookback_window,
                                        len(self.returns) - prediction_horizon))

    def __len__(self):
        return len(self.valid_indices)

    def __getitem__(self, idx):
        actual_idx = self.valid_indices[idx]
        X = self.returns[actual_idx - self.lookback_window:actual_idx]
        y = self.returns[actual_idx:actual_idx + self.prediction_horizon].mean(axis=0)
        cov_matrix = np.cov(X.T)
        return (torch.FloatTensor(X),
                torch.FloatTensor(y),
                torch.FloatTensor(cov_matrix))

print("Dataset class defined")
```

Define Dataset Class

Handling stock return data with rolling windows.

Here, we set
Lookback period = 60
Prediction horizon = 1

DFL for MVO

- Hands-on session

```python
class MVOOptimizationLayer(nn.Module):
    """Differentiable MVO layer using cvxpylayers"""

    def __init__(self, n_stocks: int, lambda_risk: float = 1.0):
        super(MVOOptimizationLayer, self).__init__()
        self.n_stocks = n_stocks
        self.lambda_risk = lambda_risk

        # Define optimization problem
        mu = cp.Parameter(n_stocks)
        L = cp.Parameter((n_stocks, n_stocks))
        w = cp.Variable(n_stocks)

        portfolio_variance = cp.sum_squares(L.T @ w)
        objective = cp.Maximize(mu @ w - self.lambda_risk * portfolio_variance)

        constraints = [cp.sum(w) == 1, w >= 0]

        problem = cp.Problem(objective, constraints)
        self.optimization_layer = CvxpyLayer(problem, parameters=[mu, L], variables=[w])

    def forward(self, predicted_returns, covariance_matrices):
        batch_size = predicted_returns.shape[0]
        weights_list = []

        for i in range(batch_size):
            cov = covariance_matrices[i] + torch.eye(self.n_stocks, device=covariance_matrices.device) * 1e-6
            L = torch.linalg.cholesky(cov)
            # cvxpylayers requires CPU tensors
            weights, = self.optimization_layer(predicted_returns[i].cpu(), L.cpu())
            weights_list.append(weights.to(predicted_returns.device))

        return torch.stack(weights_list)

print("MVOOptimizationLayer defined")
```

Define Optimization Layer

$$
\begin{aligned}
\max_{w} \quad & w^{\mathrm{T}}\mu - \lambda w^{\mathrm{T}}\Sigma w \\
\text{subject to} \quad & w^{T}\mathbf{1} = 1 \\
& w \geq 0
\end{aligned}
$$

DFL for MVO

- Hands-on session

```python
class MVOOptimizationLayer(nn.Module):
    """Differentiable MVO layer using cvxpylayers"""

    def __init__(self, n_stocks: int, lambda_risk: float = 1.0):
        super(MVOOptimizationLayer, self).__init__()
        self.n_stocks = n_stocks
        self.lambda_risk = lambda_risk

        # Define optimization problem
        mu = cp.Parameter(n_stocks)
        L = cp.Parameter((n_stocks, n_stocks))
        w = cp.Variable(n_stocks)

        portfolio_variance = cp.sum_squares(L.T @ w)
        objective = cp.Maximize(mu @ w - self.lambda_risk * portfolio_variance)

        constraints = [cp.sum(w) == 1, w >= 0]

        problem = cp.Problem(objective, constraints)
        self.optimization_layer = CvxpyLayer(problem, parameters=[mu, L], variables=[w])

    def forward(self, predicted_returns, covariance_matrices):
        batch_size = predicted_returns.shape[0]
        weights_list = []

        for i in range(batch_size):
            cov = covariance_matrices[i] + torch.eye(self.n_stocks, device=covariance_matrices.device) * 1e-6
            L = torch.linalg.cholesky(cov)
            # cvxpylayers requires CPU tensors
            weights, = self.optimization_layer(predicted_returns[i].cpu(), L.cpu())
            weights_list.append(weights.to(predicted_returns.device))

        return torch.stack(weights_list)

print("MVOOptimizationLayer defined")
```

Define Optimization Layer

$$\max_{w} \quad w^{\mathrm{T}}\mu - \lambda w^{\mathrm{T}}\Sigma w$$

$$\text{subject to} \quad w^{T}\mathbf{1} = 1$$

$$w \geq 0$$

DFL for MVO

- Hands-on session

```python
class MVOOptimizationLayer(nn.Module):
    """Differentiable MVO layer using cvxpylayers"""

    def __init__(self, n_stocks: int, lambda_risk: float = 1.0):
        super(MVOOptimizationLayer, self).__init__()
        self.n_stocks = n_stocks
        self.lambda_risk = lambda_risk

        # Define optimization problem
        mu = cp.Parameter(n_stocks)
        L = cp.Parameter((n_stocks, n_stocks))
        w = cp.Variable(n_stocks)

        portfolio_variance = cp.sum_squares(L.T @ w)
        objective = cp.Maximize(mu @ w - self.lambda_risk * portfolio_variance)

        constraints = [cp.sum(w) == 1, w >= 0]

        problem = cp.Problem(objective, constraints)
        self.optimization_layer = CvxpyLayer(problem, parameters=[mu, L], variables=[w])
```

Define Optimization Layer

We can use standard cvxpy syntax.
However, CvxpyLayers only supports affine mappings.
So we cannot use ***quad_form*** function directly.

We must reformulate using Cholesky decomposition.
$\Rightarrow$ Express objective via ***sum_squares*** function

1. Decompose: $\Sigma = LL^T$ (Cholesky)
2. Reformulate: $w^T \Sigma w = \|L^T w\|_2^2$
3. Implementation: `cp.sum_squares(L.T @ w)`

DFL for MVO

- Hands-on session

```python
class MVOOptimizationLayer(nn.Module):
    """Differentiable MVO layer using cvxpylayers"""

    def __init__(self, n_stocks: int, lambda_risk: float = 1.0):
        super(MVOOptimizationLayer, self).__init__()
        self.n_stocks = n_stocks
        self.lambda_risk = lambda_risk

        # Define optimization problem
        mu = cp.Parameter(n_stocks)
        L = cp.Parameter((n_stocks, n_stocks))
        w = cp.Variable(n_stocks)

        portfolio_variance = cp.sum_squares(L.T @ w)
        objective = cp.Maximize(mu @ w - self.lambda_risk * portfolio_variance)

        constraints = [cp.sum(w) == 1, w >= 0]

        problem = cp.Problem(objective, constraints)
        self.optimization_layer = CvxpyLayer(problem, parameters=[mu, L], variables=[w])
```

Define Loss functions

We use Combined Loss

: $(1-\alpha)\cdot$ L_pred + $\alpha \cdot$ L_decision

Prediction Loss

: L_pred = MSE(predicted_returns, true_returns)

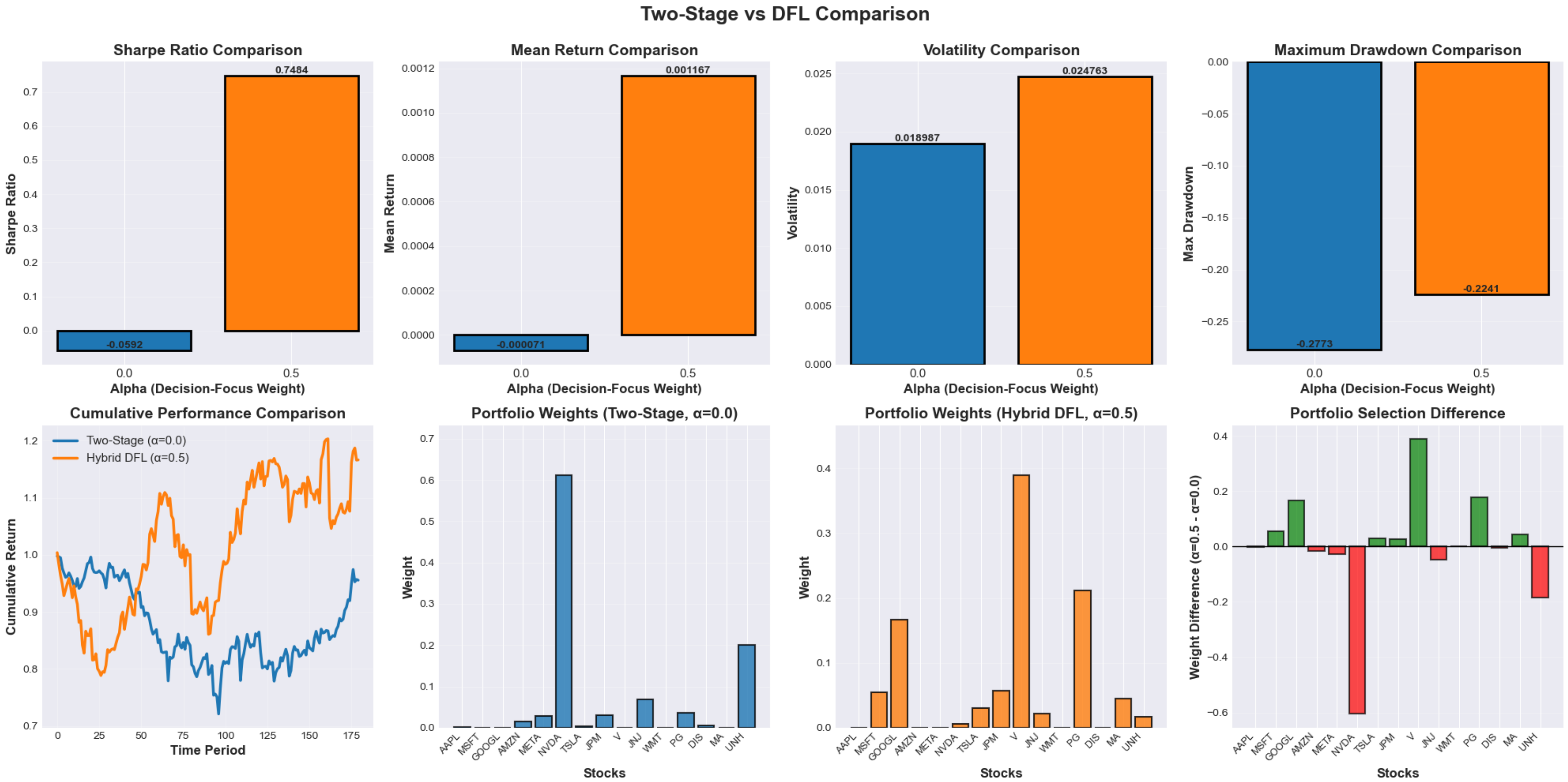
Decision Loss (MVO)

: L_decision = objective(optimal_weights) – objective(predicted_weights)

Since our optimization is a maximization problem

DFL for MVO

- Hands-on session



DFL for MVO

- More details

Return Prediction for Mean-Variance Portfolio Selection : How Decision-Focused Learning Shapes Forecasting Models

Junhyeong Lee[1*], Haeun Jeon[2*], Hyunglip Bae[2†], Yongjae Lee[1†]

[1]UNIST, [2]KAIST
[*]Co-first authors, [†]Corresponding authors

Presenting at ICAIF'25

Junhyeong Lee
Financial Engineering Lab
Department of Industrial Engineering

11/17 Monday
11:30 AM – 12:30 PM
Session: Decision-Aware Portfolio Optimizaiton

Location: Ballroom 3

OUR PAPER

ABOUT ME

DFL for MVO

- Pre-view

Table 2: Portfolio performance metrics for varying λ and α. Models with $\alpha > 0$ consistently outperform pure MSE ($\alpha = 0$), with optimal performance typically at intermediate values ($\alpha \in [0.25, 0.75]$). DFL improves both returns and risk metrics, achieving higher Sharpe ratios and often lower maximum drawdowns. Bold values indicate best performance for each metrics.

Panel A. DOW 30 Dataset						Panel B. S&P 100 Dataset					
λ	α	Return (↑)	Sharpe (↑)	MDD (↓)	Wealth (↑)	λ	α	Return (↑)	Sharpe (↑)	MDD (↓)	Wealth (↑)
	0.00	0.181	0.681	0.253	1.459		0.00	0.209	0.734	0.266	1.184
	0.25	0.323	1.138	0.230	1.669		0.25	0.256	0.744	0.327	1.673
0.1	0.50	**0.398**	**1.228**	0.261	**2.138**	0.1	0.50	0.134	0.513	0.278	1.239
	0.75	0.177	0.699	0.268	1.408		0.75	0.091	0.386	0.319	1.277
	1.00	0.143	0.593	**0.226**	1.275		1.00	**0.292**	**1.179**	**0.249**	**1.701**
	0.00	0.125	0.529	0.251	1.250		0.00	0.153	0.578	0.276	1.126
	0.25	0.192	0.754	0.258	1.459		0.25	0.187	0.769	0.295	1.339
0.5	0.50	**0.347**	**1.302**	**0.211**	**1.813**	0.5	0.50	0.231	0.701	0.295	1.680
	0.75	0.248	1.051	0.230	1.520		0.75	**0.303**	**1.217**	**0.230**	**1.763**
	1.00	0.151	0.641	0.268	1.354		1.00	0.183	0.769	0.267	1.512
	0.00	0.115	0.519	0.242	1.234		0.00	0.129	0.540	0.242	1.090
	0.25	0.165	0.775	0.221	1.383		0.25	0.237	0.746	0.267	1.772
1.0	0.50	**0.327**	1.150	0.218	**1.669**	1.0	0.50	**0.319**	**1.260**	**0.214**	**1.773**
	0.75	0.312	**1.311**	**0.203**	1.651		0.75	0.201	0.801	0.237	1.410
	1.00	0.180	0.763	0.256	1.397		1.00	0.252	0.949	0.287	1.569
	0.00	0.091	0.531	**0.195**	1.163		0.00	0.144	0.838	**0.194**	1.258
	0.25	**0.217**	**0.932**	0.203	**1.566**		0.25	0.203	0.831	0.250	1.523
5.0	0.50	0.187	0.756	0.215	1.473	5.0	0.50	0.136	0.585	0.219	1.347
	0.75	0.197	0.871	0.211	1.379		0.75	0.193	0.855	0.237	1.464
	1.00	0.164	0.738	0.223	1.361		1.00	**0.213**	**0.928**	0.276	**1.611**

DFL for MVO

- Pre-view

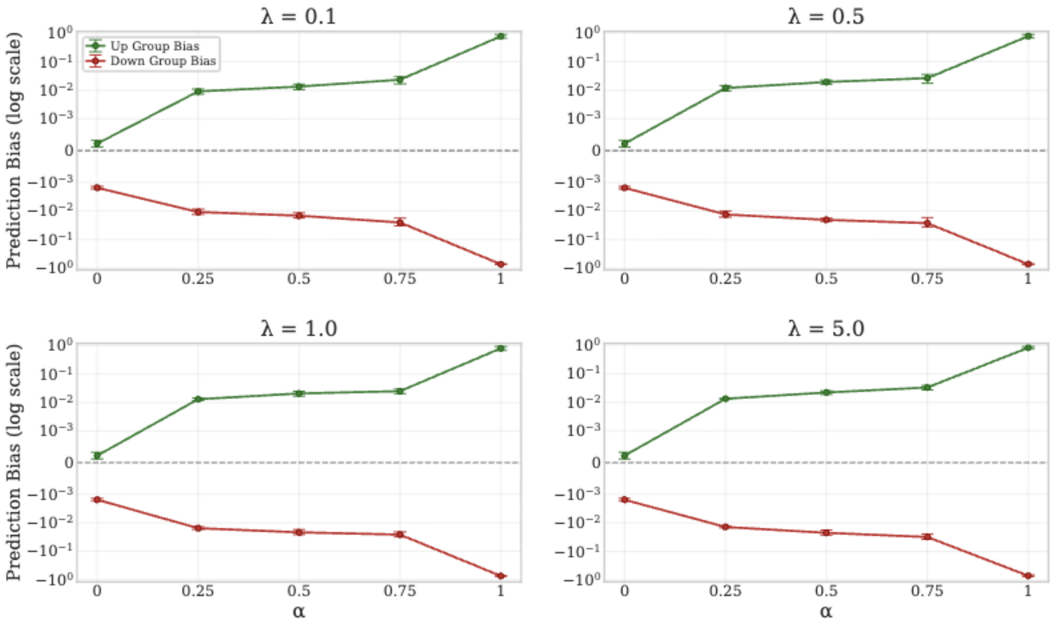
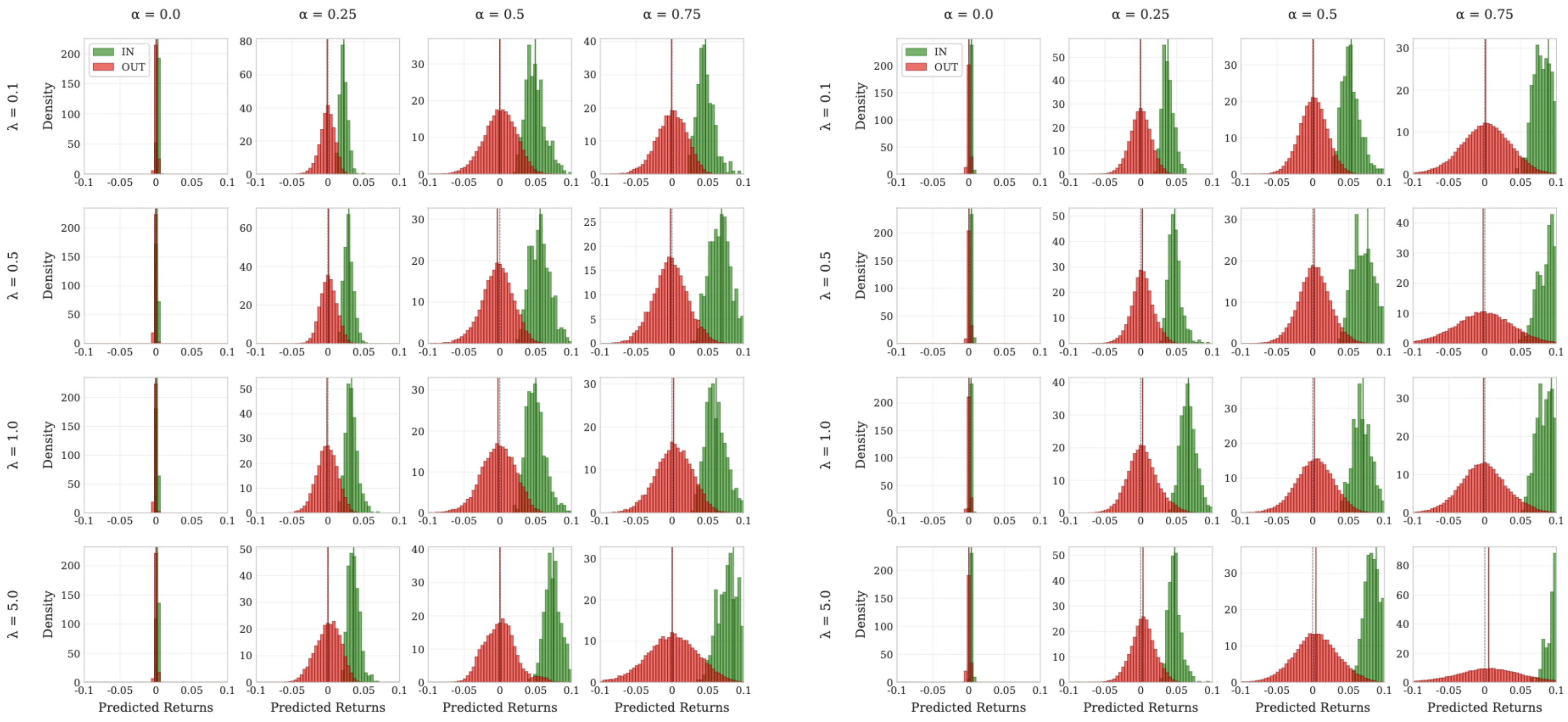


Figure 3: Prediction bias across Up/Down assets in DOW 30. As α increases, the Up group becomes increasingly overestimated while the Down group becomes underestimated, reaching extreme polarization at $\alpha = 1$.

DFL for MVO

- Pre-view



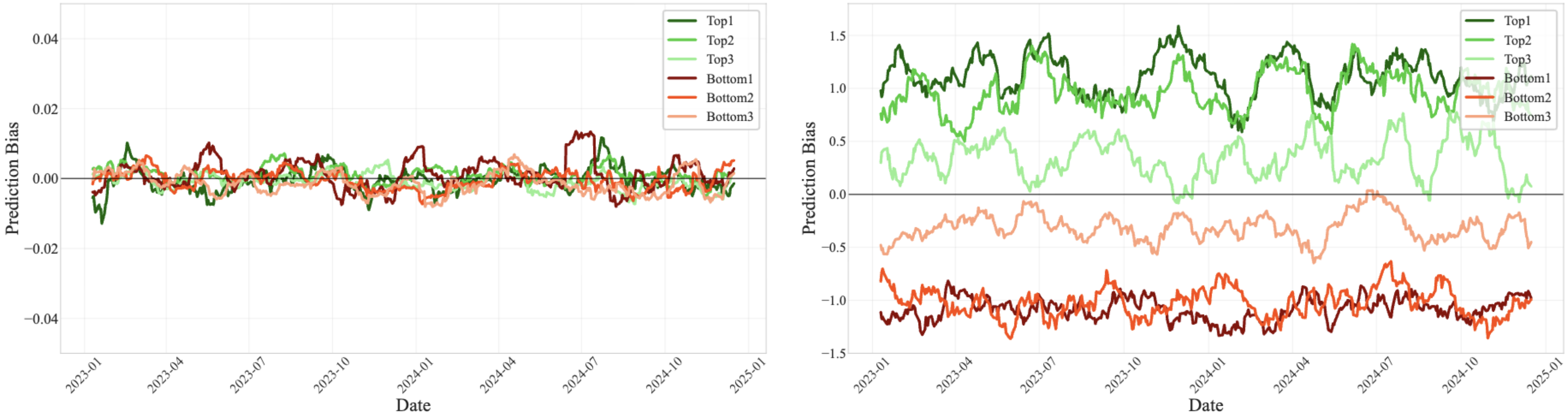
(a) DOW30

(b) S&P100

Figure 4: Predicted return distributions for IN/OUT portfolio groups across different λ and α values. As α increases, the separation between IN and OUT group distributions widens. The case of $\alpha = 1$ is excluded due to extreme distribution separation.

DFL for MVO

- Pre-view



(a) $\alpha = 0$ **(b)** $\alpha = 1$

Figure 5: Prediction bias patterns for portfolio assets under MSE loss ($\alpha = 0$) versus MVO loss ($\alpha = 1$). With MSE loss, prediction biases are randomly distributed across all assets regardless of portfolio inclusion. With MVO loss, prediction biases exhibit clear polarization: assets with high portfolio weights (Top) show positive bias while assets with low weights (Bottom) show negative bias, demonstrating how DFL induces strategic differentiation based on portfolio relevance.

Thank you for listening!