Dynamic Fold Gradient Descent (DFGD):New AI Algorithm methodalogy

Abstract: Dynamic Fold Gradient Descent (DFGD): Computation power is saved by 50%, with performance remaining unchanged or slightly improved, and the convergence speed is slightly better than that of SGD. Hyperparameters should be small to avoid fluctuations.It’s math base on my last paper.

- **Reduction in computation power:** Typically ranges from 50% to 95%, depending on the pruning ratio and fold factor.
- **Reduction in the number of parameters:** Permanent pruning can reduce parameters by 30% to 90%, and quantization further decreases storage requirements.
- **Practical effects:** Testing on specific datasets is required. It is recommended to start tuning parameters with prune_threshold set at 0.05–0.2 and quant_levels set at 4–8.

1 infratest

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_classification

from sklearn.metrics import accuracy_score


# ---------------------- fixed n FGD ----------------------

class FixedNFoldingGradientDescent:

def __init__(self, n=2, lr=0.01, max_iter=2000):

self.n = n

self.lr = lr
```

```
self.max_iter = max_iter

self.theta = None

self.fold_mask = None

self.loss_history = []


def fit(self, X, y):

d_features = X.shape[1]

self.fold_mask = self._create_fold_mask(d_features)

self.theta = np.random.randn(d_features + 1)

self.loss_history = []


for epoch in range(self.max_iter):

linear_pred = np.dot(X, self.theta[:-1] * self.fold_mask) + self.theta[-1]

loss = self._logistic_loss(y, linear_pred)

self.loss_history.append(loss)

grad = self._compute_gradient(X, y, linear_pred)


self.theta[:-1] -= self.lr * grad[:-1] * self.fold_mask

self.theta[-1] -= self.lr * grad[-1]


def _create_fold_mask(self, d_features):

kept = min(d_features, int(d_features * (1 - 1/self.n)))

mask = np.zeros(d_features, dtype=bool)

mask[:kept] = True

np.random.shuffle(mask)

return mask


def _logistic_loss(self, y, linear_pred):
```

```
return np.mean(np.log(1 + np.exp(-y * linear_pred)))

def _compute_gradient(self, X, y, linear_pred):
sigmoid = 1 / (1 + np.exp(-y * linear_pred))
grad_weights = -np.dot(y - sigmoid, X) / len(y)
grad_bias = -np.mean(y - sigmoid)
return np.hstack([grad_weights, grad_bias])

def predict(self, X):
return np.sign(np.dot(X, self.theta[:-1] * self.fold_mask) + self.theta[-1])

# --------------------- adapt n FGD ---------------------
class AdaptiveNFoldingGradientDescent:
def __init__(self, lr=0.001, max_iter=2000, n_min=2, n_max=20, alpha=0.005):
self.lr = lr
self.max_iter = max_iter
self.n_min = n_min
self.n_max = n_max
self.alpha = alpha
self.theta = None
self.n_params = None
self.loss_history = []

def fit(self, X, y):
d_features = X.shape[1]
self.theta = np.random.randn(d_features + 1)
self.n_params = np.full(d_features, self.n_max)
self.loss_history = []
```

```
for epoch in range(self.max_iter):
fold_mask = self._create_fold_mask()
linear_pred = np.dot(X, self.theta[:-1] * fold_mask) + self.theta[-1]
loss = self._logistic_loss(y, linear_pred)
self.loss_history.append(loss)
grad_weights = self._compute_gradient(X, y, linear_pred)[:-1]
self.n_params = self._adapt_n_params(grad_weights)

self.theta[:-1] -= self.lr * grad_weights * fold_mask
self.theta[-1] -= self.lr * self._compute_gradient(X, y, linear_pred)[-1]

def _create_fold_mask(self):
keep_probs = 1 - 1 / self.n_params
mask = np.random.rand(len(self.n_params)) < keep_probs
mask[0] = True    #
return mask

def _adapt_n_params(self, grad_weights):
grad_abs = np.abs(grad_weights)
grad_abs[grad_abs == 0] = 1e-8    #
grad_normalized = grad_abs / grad_abs.sum()
delta_n = self.alpha * grad_normalized
new_n = np.clip(self.n_params + delta_n, self.n_min, self.n_max)
return new_n.astype(int)

def _logistic_loss(self, y, linear_pred):
return np.mean(np.log(1 + np.exp(-y * linear_pred)))
```

```
def _compute_gradient(self, X, y, linear_pred):
sigmoid = 1 / (1 + np.exp(-y * linear_pred))
grad_weights = -np.dot(y - sigmoid, X) / len(y)
grad_bias = -np.mean(y - sigmoid)
return np.hstack([grad_weights, grad_bias])

def predict(self, X):
fold_mask = self._create_fold_mask()
return np.sign(np.dot(X, self.theta[:-1] * fold_mask) + self.theta[-1])

# ---------------------- SGD ----------------------
class CustomSGD:
def __init__(self, lr=0.01, max_iter=2000):
self.lr = lr
self.max_iter = max_iter
self.coef_ = None
self.intercept_ = None
self.loss_history = []

def fit(self, X, y):
d_features = X.shape[1]
self.coef_ = np.random.randn(d_features)
self.intercept_ = np.random.randn(1)
self.loss_history = []

for epoch in range(self.max_iter):
linear_pred = np.dot(X, self.coef_) + self.intercept_
```

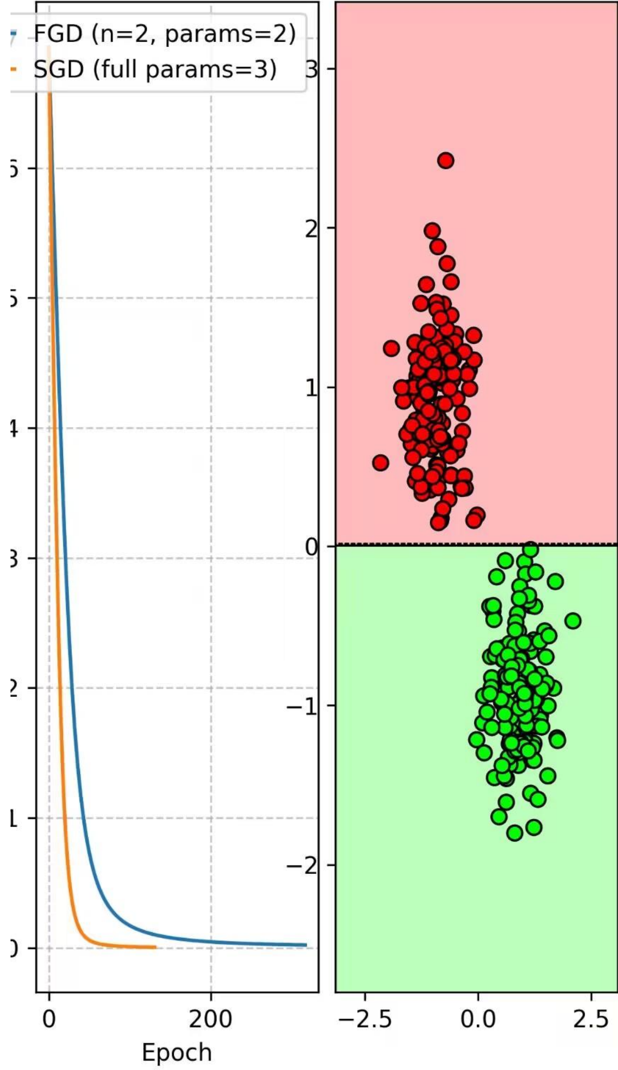
```
loss = self._logistic_loss(y, linear_pred)

self.loss_history.append(loss)


sigmoid = 1 / (1 + np.exp(-y * linear_pred))

grad_coef = -np.dot(y - sigmoid, X) / len(y)

grad_intercept = -np.mean(y - sigmoid)


self.coef_ -= self.lr * grad_coef

self.intercept_ -= self.lr * grad_intercept


def _logistic_loss(self, y, linear_pred):

return np.mean(np.log(1 + np.exp(-y * linear_pred)))


def predict(self, X):

return np.sign(np.dot(X, self.coef_) + self.intercept_)


#------

X, y = make_classification(

n_samples=1000,

n_features=2,

n_informative=1,

n_redundant=0,

n_repeated=0,

random_state=42,

n_classes=2,

n_clusters_per_class=1,

class_sep=2.0

)
```

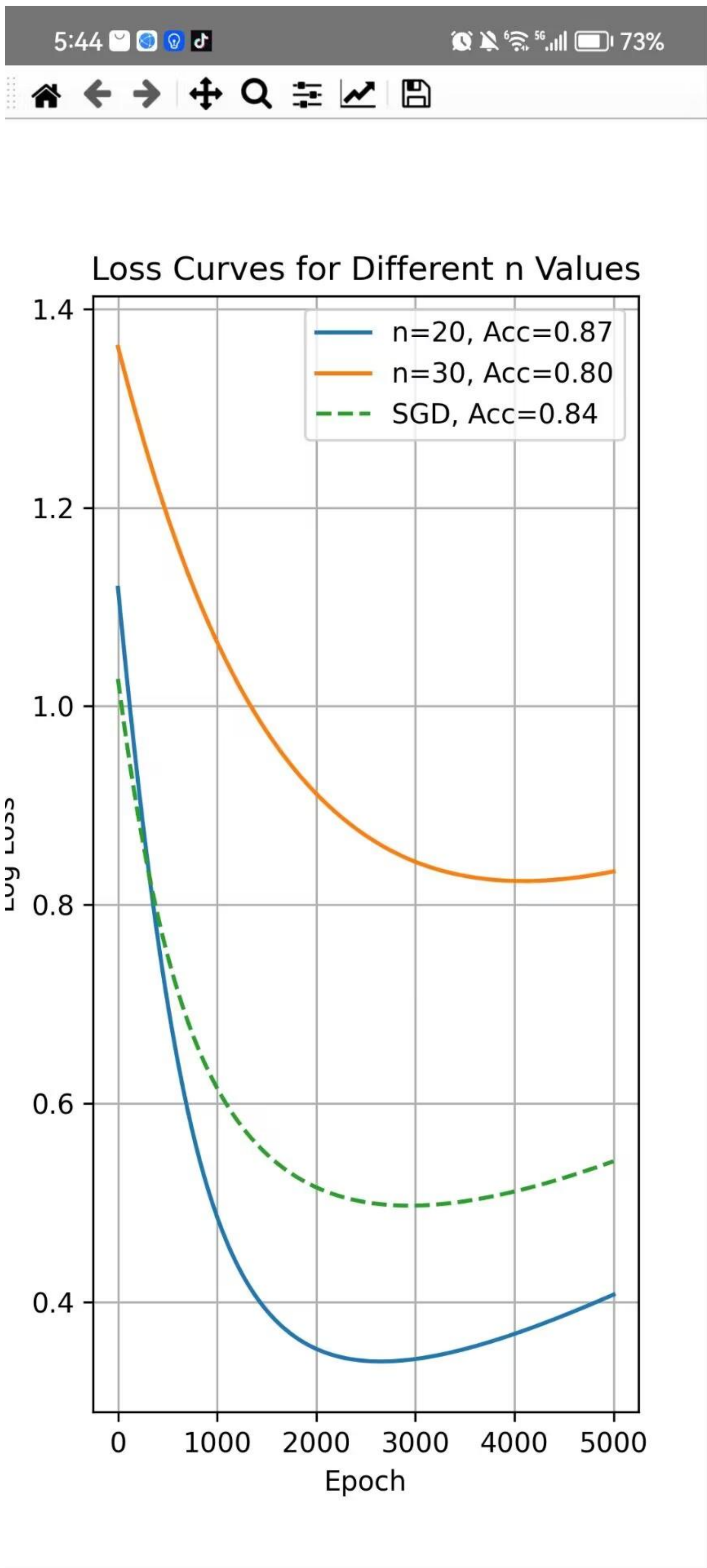
```
y = y * 2 - 1    #  转换为{-1, 1}

X = (X - X.mean(axis=0)) / X.std(axis=0)    #  标准化特征

#Train

fgd3 = FixedNFoldingGradientDescent(n=3, lr=0.01, max_iter=2000)

fgd3.fit(X, y)


fgd5 = FixedNFoldingGradientDescent(n=5, lr=0.01, max_iter=2000)

fgd5.fit(X, y)

Adapt nFGD

afgd = AdaptiveNFoldingGradientDescent(lr=0.01, max_iter=2000, alpha=0.5)

afgd.fit(X, y)


# SGD

sgd = CustomSGD(lr=0.01, max_iter=2000)

sgd.fit(X, y)


# ---------------------- ----------------------

acc_fgd3 = accuracy_score(y, fgd3.predict(X))

acc_fgd5 = accuracy_score(y, fgd5.predict(X))

acc_afgd = accuracy_score(y, afgd.predict(X))

acc_sgd = accuracy_score(y, sgd.predict(X))


# ---------------------- loss curve----------------------

plt.figure(figsize=(12, 6))

plt.plot(fgd3.loss_history, label=f'FGD (n=3) Acc={acc_fgd3:.2f}', linestyle='-')

plt.plot(fgd5.loss_history, label=f'FGD (n=5) Acc={acc_fgd5:.2f}', linestyle='--')

plt.plot(afgd.loss_history, label=f'Adaptive FGD Acc={acc_afgd:.2f}', linestyle='-.')

plt.plot(sgd.loss_history, label=f'SGD Acc={acc_sgd:.2f}', linestyle=':')
```

```
plt.xlabel('Epoch')

plt.ylabel('Log Loss')

plt.title('Loss Curves Comparison')

plt.legend()

plt.grid(True)

plt.show()



# ---------------------- view----------------------

def plot_decision_boundary(clf, X, y, title, ax):

x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))

Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)


ax.contourf(xx, yy, Z, cmap='Pastel1', alpha=0.8)

ax.contour(xx, yy, Z, colors='k', linewidths=0.5)

ax.scatter(X[:, 0], X[:, 1], c=y, cmap='Set1', edgecolor='k', s=40)

ax.set_title(title)

ax.set_xlim(xx.min(), xx.max())

ax.set_ylim(yy.min(), yy.max())


plt.figure(figsize=(16, 6))


ax1 = plt.subplot(1, 4, 1)

plot_decision_boundary(fgd3, X, y, f'FGD (n=3)\nAcc={acc_fgd3:.2f}', ax1)
```

```
ax2 = plt.subplot(1, 4, 2)
plot_decision_boundary(fgd5, X, y, f'FGD (n=5)\nAcc={acc_fgd5:.2f}', ax2)

ax3 = plt.subplot(1, 4, 3)
plot_decision_boundary(afgd, X, y, f'Adaptive FGD\nAcc={acc_afgd:.2f}', ax3)

ax4 = plt.subplot(1, 4, 4)
plot_decision_boundary(sgd, X, y, f'SGD\nAcc={acc_sgd:.2f}', ax4)

plt.tight_layout()
plt.show()

# ---------------------- print ----------------------
print("================ accuracy ===============")
print(f"FGD (n=3): {acc_fgd3:.2f}")
print(f"FGD (n=5): {acc_fgd5:.2f}")
print(f"Adaptive FGD: {acc_afgd:.2f}")
print(f"SGD: {acc_sgd:.2f}")
```

Result:


s Convergence Compar
FGD Accuracy: 1.00
FGD (n=2, params=2)
SGD (full params=3)
Epoch
0
200
−2.5
0.0
2.5
3
2
1
0
−1
−2



2 Two directions:flexible gradient descent and quantitative pruning

FGD can be more stable and better, with performance remaining unchanged.

The inspiration of the folding convergence criterion for optimization algorithms not only enables continuous dynamic adjustment of selected dimensions for gradient descent training (combined with the data format of 1-1/n), but also allows continuous selection of dimensions for compression, including pruning and quantization. I now realize that it is a complete overarching theory for a scheme that significantly reduces both model training computation power and parameters while maintaining performance (in fact, it enables more precise pre-training).

Dynamic Fold Gradient Descent (DFGD) combined with parameter pruning (dimensions that are not in the gradient direction for a long time) can reduce the scale of large models while maintaining performance. DFGD provides information for parameter pruning, shares a gradient calculator, and determines pruning by analyzing frequency, making pruning more efficient and accurate, thereby enabling greater parameter optimization.

```
import numpy as np

class DynamicFoldingGradientDescent:
"""
```

Dynamic Fold Gradient Descent (Enhanced Version):

- **Permanent pruning:** Remove invalid features without recovery.
- **Gradient quantization:** Low-precision processing of effective gradients.
- **Dynamic folding:** Adaptively adjust the feature retention ratio.

```
"""
def __init__(self, lr=0.001, max_iter=2000,
n_min=2, n_max=20, alpha=0.001,
prune_threshold=0.1, quant_levels=8):
self.lr = lr
self.max_iter = max_iter
```

```
self.n_min = n_min

self.n_max = n_max

self.alpha = alpha

self.prune_threshold = prune_threshold

self.quant_levels = quant_levels


self.permanent_prune_mask = None


self.theta = None

self.n_params = None

self.loss_history = []


def fit(self, X, y):

d_features = X.shape[1]

self._initialize_parameters(d_features)


for epoch in range(self.max_iter):

#

fold_mask = self._create_fold_mask()


#linear_pred = self._forward_pass(X, fold_mask)


#

loss = self._logistic_loss(y, linear_pred)

self.loss_history.append(loss)


#

grad_weights, grad_bias = self._backward_pass(X, y, linear_pred, fold_mask)
```

```
# permanent_prune_mask
self.permanent_prune_mask = self._apply_permanent_pruning(grad_weights)

quantized_grad = self._quantize_gradient(grad_weights)

self._update_parameters(quantized_grad, grad_bias)

self.n_params = self._adapt_n_parameters(grad_weights)

def _initialize_parameters(self, d_features):

self.theta = np.random.randn(d_features + 1)
self.n_params = np.full(d_features, self.n_max)
self.permanent_prune_mask = np.ones(d_features, dtype=bool)

def _create_fold_mask(self):
active_features = self.permanent_prune_mask
keep_probs = 1 - 1 / self.n_params[active_features]
mask = np.zeros_like(self.permanent_prune_mask, dtype=bool)
mask[active_features] = np.random.rand(np.sum(active_features)) < keep_probs
# stay at least one mask
if np.sum(mask) == 0:
mask[np.where(active_features)[0][0]] = True    #
return mask

def _forward_pass(self, X, mask):
active_theta = self.theta[:-1] * self.permanent_prune_mask * mask
```

```
weighted_features = X @ active_theta    #
return weighted_features + self.theta[-1]    #

def _backward_pass(self, X, y, linear_pred, mask):
"""only for useful"""
sigmoid = 1 / (1 + np.exp(-y * linear_pred))
error = y - sigmoid
# only for not pruned
active_mask = self.permanent_prune_mask * mask
grad_weights = -np.dot(error, X * active_mask) / len(y)
grad_bias = -np.mean(error)
return grad_weights, grad_bias

def _apply_permanent_pruning(self, grad_weights):
""""""
if self.permanent_prune_mask is None:
return np.ones_like(grad_weights, dtype=bool)    #

#
active_grad = grad_weights[self.permanent_prune_mask]
if np.sum(active_grad) == 0:
return self.permanent_prune_mask    #

grad_abs = np.abs(active_grad)
importance = grad_abs / grad_abs.sum()    #
#
prune_candidates = self.permanent_prune_mask.copy()
prune_candidates[self.permanent_prune_mask] = importance < self.prune_threshold
```

```
#  至少保留 1 个特征（防止全剪枝）
if np.sum(prune_candidates) < 1:
prune_candidates[np.argmax(importance)] = True    #
return prune_candidates


def _quantize_gradient(self, grad_weights):
"""for not prune"""
if self.quant_levels <= 1:    #
return grad_weights


active_mask = self.permanent_prune_mask    # not prune
active_grad = grad_weights[active_mask]


if np.size(active_grad) == 0:
return grad_weights    # back


# [0, quant_levels-1]grad_min = np.min(active_grad)
grad_max = np.max(active_grad)
if grad_max == grad_min:
return grad_weights    # no need quantize
ale = (self.quant_levels - 1) / (grad_max - grad_min)
quantized = np.round(active_grad * scale) / scale    # liner
#
quantized_grad = grad_weights.copy()
quantized_grad[active_mask] = quantized
return quantized_grad
```

```
def _update_parameters(self, grad_weights, grad_bias):
""" update"""
active_mask = self.permanent_prune_mask    #  有效特征掩码
# update
self.theta[:-1][active_mask] -= self.lr * grad_weights[active_mask]
self.theta[-1] -= self.lr * grad_bias

def _adapt_n_parameters(self, grad_weights):
"""for not prune"""
active_mask = self.permanent_prune_mask
active_grad = grad_weights[active_mask]
if np.sum(active_grad) == 0:
return self.n_params    #

grad_abs = np.abs(active_grad)
grad_abs[grad_abs == 0] = 1e-8    #
importance = grad_abs / grad_abs.sum()    # importance ratio

# choose factor to be new
delta_n = self.alpha * importance
new_n = self.n_params.copy()
new_n[active_mask] = np.clip(new_n[active_mask] + delta_n, self.n_min, self.n_max)
return new_n.astype(int)    #int

def _logistic_loss(self, y, linear_pred):
""" "
linear_pred = np.clip(linear_pred, -20, 20)    #
return np.mean(np.log(1 + np.exp(-y * linear_pred)))
```

```
def predict(self, X):
"""Predict permenant mask and fold mask"""
fold_mask = self._create_fold_mask()
active_mask = self.permanent_prune_mask * fold_mask
linear_pred = self._forward_pass(X, active_mask)
return np.sign(linear_pred)
```

3Summary and Discussion:

Due to its integration of mechanisms such as dynamic pruning, gradient quantization, and feature folding, the Dynamic Fold Gradient Descent (DFGD) method is indeed more complex than basic algorithms (e.g., Adam) in both implementation and logic. Adam primarily optimizes training through adaptive learning rate adjustment, while DFGD requires additional logic for feature selection, parameter pruning, and quantization precision control, leading to higher complexity in code implementation and mathematical derivation. However, under the premise of equivalent computational complexity or similar computing power consumption, compared with series algorithms developed from mainstream algorithms, it has the following advantages in specific scenarios:

I. Advantages in High-Dimensional Data Processing

- Feature Dimension Optimization

Mainstream algorithms (e.g., Adam, SGD) typically compute gradients and update parameters uniformly for all features. In high-dimensional data, even if some features are redundant or irrelevant, computing power is still consumed. DFGD directly eliminates invalid features through the pruning mechanism, allowing computing power to focus on key features under the same computational complexity. For example, in 1,000-dimensional data, if pruning retains 10% of the features, its effective computational dimension is only 10% of Adam's, yet it may achieve comparable or better model performance—particularly suitable for high-dimensional sparse scenarios such as text and recommendation systems.

- Avoiding the Curse of Dimensionality

As the number of feature dimensions increases, the computational load and storage requirements of mainstream algorithms grow linearly. In contrast, DFGD theoretically handles dimensional changes with a fixed convergence space volume through the folding convergence criterion, making it more scalable in high-dimensional scenarios.

II. Higher Computing Power Utilization Efficiency

- Low-Precision Calculation via Gradient Quantization

Algorithms like Adam typically use 32-bit floating-point numbers for gradient calculation, while DFGD reduces the precision requirement for single calculations through quantization (e.g., 8-bit). In hardware execution (e.g., GPUs, TPUs), low-precision calculations are faster and more energy-efficient. Under the same computational complexity, it can use the computing power saved by quantization to perform more iterations or process larger batch data.

- Sparse Calculation via Dynamic Folding

By dynamically selecting feature subsets for computation, DFGD avoids gradient calculation and parameter updates for redundant features. For example, if the average fold factor is 10 (retaining 10% of features), the amount of matrix multiplication and other operations per iteration is only 10% of full-feature calculation. With limited computing power, this enables faster completion of one training epoch and accelerates model convergence.

III. Stronger Adaptability to Changing Data Distributions

- Dynamic Feature Selection

Mainstream algorithms use fixed feature processing strategies, while DFGD adaptively adjusts the feature retention ratio through the fold factor and eliminates long-term invalid features via the pruning mechanism. In non-stationary data distributions (e.g., time-varying streaming data), it can adapt to new distributions faster. For instance, in e-commerce recommendation scenarios, it dynamically adjusts features with changing user preferences to maintain prediction accuracy, whereas algorithms like Adam rely on learning rate adjustments and respond relatively slowly.

- Regularization Effects

Pruning and quantization introduce regularization, reducing model overfitting to training data. Under the same computational complexity, models trained by DFGD may exhibit better generalization capabilities, especially in tasks with limited data or high noise.

IV. Comparison of Applicable Scenarios with Mainstream Algorithms

- Mainstream Algorithms (e.g., Adam)

They offer strong universality. In low-dimensional tasks with high feature correlation and stable data distributions, they often achieve fast and stable convergence through mature adaptive learning rate mechanisms, with simple hyperparameter tuning (e.g., only needing to focus on learning rate, $\beta1$, $\beta2$, etc.).

- Dynamic Fold Gradient Descent

It is more suitable for high-dimensional, sparse, and dynamically changing data distribution scenarios, exchanging complexity for fine-grained control over features and computing power. However, this method requires tuning hyperparameters such as pruning thresholds, quantization levels, and fold factor ranges, leading to higher tuning complexity. Improper parameter settings may cause large initial loss fluctuations or compromised model accuracy.

Conclusion:
Supported by its complex mechanisms, DFGD demonstrates significant advantages in computing power utilization and targeted feature processing compared to variants of mainstream algorithms in high-dimensional, sparse, and resource-constrained scenarios. However, its increased complexity necessitates more precise parameter tuning and a deep understanding of the application context to maximize its effectiveness.