Orthogonal Gradient Descent (OGD) in Continual Learning

Ali Ahmadi Esfidi

July 14, 2025

Abstract

Abstract: Continual learning is a fundamental challenge in the development of artificial intelligence systems capable of sequentially learning new tasks without forgetting previously acquired knowledge. Despite their impressive capabilities in solving complex problems, deep neural networks face the phenomenon of "Catastrophic Forgetting"; meaning that when trained on new tasks, their performance on previous tasks significantly degrades. In this report, we examine the "Orthogonal Gradient Descent" approach, which has been introduced to address this challenge. OGD, by constraining the direction of gradient updates in the parameter space, ensures that the neural network's output on previous tasks does not change, while still providing a useful direction for learning new tasks. This method has demonstrated its effectiveness on benchmark criteria by optimally utilizing the high capacity of neural networks and without requiring the storage of past data (which may raise privacy concerns). Experimental results indicate the effectiveness of OGD in preserving past knowledge and efficiently learning new tasks in continual learning scenarios.

1 Introduction

In recent years, deep neural networks have achieved significant advancements in various fields of artificial intelligence, including computer vision, natural language processing, and robotics. However, one of the main limitations of these models in dynamic learning environments is the problem of Continual Learning. In many real-world applications, intelligent systems must continuously acquire new knowledge without retraining from scratch and without forgetting previously learned information. The phenomenon of "catastrophic forgetting" refers to the fact that training a neural network on a new task can suddenly and severely degrade its performance on previous tasks. This problem arises from the interference between parameter updates for different tasks.

To overcome this challenge, several approaches have been proposed, which can be categorized into three main types: rehearsal-based methods, regularization-based methods, and architecture-based methods. Each of these approaches has its own advantages and disadvantages.

In this report, our focus is on the "Orthogonal Gradient Descent" (OGD) method, which offers a novel approach to combat catastrophic forgetting in continual learning. OGD, by leveraging the concept of orthogonality in the gradient space, aims to update network parameters in a way that minimizes negative impact on the performance of previous tasks, while enabling the network to effectively learn new tasks. This method ensures that parameter changes do not harm prior knowledge by projecting the new task's gradient into a specific subspace. OGD not only helps preserve prior knowledge but also does so without requiring the storage of previous data samples, which is an important advantage from a privacy and memory efficiency perspective.

The purpose of this report is to provide a comprehensive overview of the OGD concept, its theoretical foundations, and an evaluation of its performance compared to other existing methods in the field of continual learning. Furthermore, potential applications and limitations of this method will be discussed.

2 Problem Formulation

Let θ be the parameter vector of a neural network. In continual learning, the network is sequentially trained on a sequence of tasks $T_1, T_2, \ldots, T_k$. When the network is being trained on task T_k, the goal is to improve performance on T_k while preserving performance on previous tasks $T_1, \ldots, T_{k-1}$.

We consider the loss function for the current task T_k as $L_k(\theta)$. The gradient of this loss function with respect to the parameters is $\nabla L_k(\theta)$. Parameter updates in the standard gradient descent method are performed as $\theta \leftarrow \theta - \eta \nabla L_k(\theta)$, where η is the learning rate. This update may cause unwanted changes in the outputs of previous tasks, leading to forgetting.

3 GD Algorithm

Suppose we want to minimize a loss function of the form

$$L(w) = \frac{1}{N} \sum_{i=1}^{N} \ell\big(y^{(i)}, f(x^{(i)}; w)\big)$$

with respect to the parameters $w \in \mathbb{R}^p$. The classical Gradient Descent algorithm is as follows:

The Gradient Descent algorithm operates as follows: Given a loss function $J(w)$, a learning rate $\eta > 0$, and an initial value $w^{(0)}$. In each iteration t, the gradient of the loss function with respect to w is first computed:

$$g^{(t)} = \nabla_w L(w^{(t)})$$

Then the parameters are updated according to the following relation:

$$w^{(t+1)} = w^{(t)} - \eta\, g^{(t)}$$

This process continues until a stopping condition is met. Finally, the value $w^{(T)}$ is considered as the output of the algorithm.

Properties:

- If J is convex and differentiable, and the learning rate η is chosen appropriately, the algorithm converges to the global minimum.
- For large datasets, the Stochastic Gradient Descent (SGD) or Mini-batch GD versions are typically used, which approximate the gradient over a subset of samples instead of the full sum.

4 OGD Algorithm

OGD seeks to find an update $\Delta\theta$ that satisfies two main conditions:

1. $\Delta\theta$ should effectively learn the new task T_k.
2. $\Delta\theta$ should have minimal impact on the outputs of previous tasks $T_1, \ldots, T_{k-1}$.

To achieve the second condition, OGD utilizes the concept of orthogonality. Specifically, OGD assumes that changes in the network's output for previous tasks can be expressed as a linear constraint on $\Delta\theta$.

4.1 Linear Approximation of Network Output

Let $f(x_i, \theta)$ be the network's output for sample x_i with parameters θ. The change in output for a previous task T_j (where $j < k$) resulting from a small change $\Delta\theta$ can be linearly approximated as:

$$\Delta f(x_i, \theta) \approx \nabla_\theta f(x_i, \theta)^T \Delta\theta$$

To prevent forgetting, OGD aims for $\Delta f(x_i, \theta)$ to be close to zero for all samples and previous tasks. This implies that $\Delta\theta$ should be orthogonal to the gradients of the outputs of previous tasks.

4.2 Definition of the Ineffectual Space

For each task T_j and a sample x_i from that task, we can compute the gradient of the network's output with respect to its parameters:

$$g_{j,i} = \nabla_\theta f(x_i, \theta)$$

To preserve performance on previous tasks, OGD aims for $\Delta\theta$ to lie in the subspace orthogonal to the set of gradients $g_{j,i}$ (for all $j < k$ and samples x_i from T_j). This space is called the "ineffectual space" or "orthogonal space."

4.3 Projection of the New Task Gradient

The goal of OGD is to project the new task's gradient $\nabla L_k(\theta)$ into a subspace that is orthogonal to all gradients of previous tasks.

The basis for the orthogonal space to previous task gradients can be obtained using Singular Value Decomposition (SVD) or other methods. In practice, OGD uses the concept of a projection matrix.

If P_V is a projection matrix that projects vectors onto the space V (the space orthogonal to the gradients of previous tasks), then the gradient update is performed as follows:

$$\Delta\theta = -\eta P_V g_k$$

The matrix P_V is constructed such that $P_V G = 0$.

Lemma: Let $\mathbf{g}$ be the gradient of the loss function $L(w)$, and let $\mathcal{S} = \{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ be an orthonormal basis of a parameter subspace. Define:

$$\tilde{\mathbf{g}} = \mathbf{g} - \sum_{i=1}^{k} \text{proj}_{\mathbf{v}_i}(\mathbf{g})$$

Then the vector $-\tilde{\mathbf{g}}$ is a descent direction for the function $L(w)$.

Proof: For a vector $\mathbf{u}$ to be a descent direction for the loss function $L(w)$, we must have:

$$\langle \mathbf{u}, \mathbf{g} \rangle \leq 0$$

Here:

$$\tilde{\mathbf{g}} = \mathbf{g} - \sum_{i=1}^{k} \text{proj}_{\mathbf{v}_i}(\mathbf{g})$$

And since $\tilde{\mathbf{g}}$ is orthogonal to the subspace $\text{span}(\mathcal{S})$ and the sum $\sum_{i=1}^{k} \text{proj}_{\mathbf{v}_i}(\mathbf{g})$ is also in this subspace, therefore:

$$\left\langle \tilde{\mathbf{g}}, \sum_{i=1}^{k} \text{proj}_{\mathbf{v}_i}(\mathbf{g}) \right\rangle = 0$$

Now we have:

$$\begin{aligned}
\langle -\tilde{\mathbf{g}}, \mathbf{g} \rangle &= \left\langle -\tilde{\mathbf{g}}, \tilde{\mathbf{g}} + \sum_{i=1}^{k} \text{proj}_{\mathbf{v}_i}(\mathbf{g}) \right\rangle \\
&= -\langle \tilde{\mathbf{g}}, \tilde{\mathbf{g}} \rangle - \left\langle \tilde{\mathbf{g}}, \sum_{i=1}^{k} \text{proj}_{\mathbf{v}_i}(\mathbf{g}) \right\rangle \\
&= -\|\tilde{\mathbf{g}}\|^2 - 0 = -\|\tilde{\mathbf{g}}\|^2 \leq 0
\end{aligned}$$

Thus, $-\tilde{\mathbf{g}}$ is a valid descent direction for $L(w)$.

4.4 Constructing the Projection Matrix

Let G_{prev} be a matrix whose columns contain the gradients corresponding to previous tasks. These gradients can be the gradients of the network output for each sample or average gradients for each task. In the paper, specifically, the gradients of the loss function for each previous task are used. That is, $g_j = \nabla L_j(\theta)$ for $j < k$.

At each training step for task T_k:

1. The gradient of the current task $g_k = \nabla L_k(\theta)$ is calculated.
2. For each previous task T_j ($j < k$, and usually only for the last task or a few previous tasks), the gradient $g_j = \nabla L_j(\theta)$ is calculated. These gradients must be kept up-to-date or retrieved from a small memory.
3. We consider the set of gradients $G_{prev} = \{g_1, g_2, \ldots, g_{k-1}\}$ (or a subset of them).
4. To project g_k onto the space orthogonal to G_{prev}, we use the projection formula:
$$g_k^{proj} = g_k - \sum_{j \in \text{prev\_tasks}} \frac{g_k^T g_j}{\|g_j\|^2} g_j$$
This is a Gram-Schmidt process, where components of g_k parallel to each of the previous task gradients are subtracted. If the previous gradients are not orthogonal, a more precise projection method is needed. The OGD paper proposes a different approach that uses projected projection matrices.

4.5 Algorithm Steps

The general steps of the OGD algorithm for learning task T_k are as follows:

1. **Initialization:**
   - Initialize the network parameters θ.
   - Initialize a memory set of gradients (e.g., M_G) that stores important gradients of previous tasks. In the simplest case, this memory can be empty, and only the gradient of the previous task (or nearby previous tasks) is calculated.
2. **For each task T_k in the learning sequence:**
   - **Training on T_k:**
     - For each mini-batch of data (X, Y) from task T_k:
       (a) Calculate the loss function gradient for the current task: $g_k = \nabla_\theta L_k(\theta; X, Y)$.
       (b) **Calculate gradients of previous tasks for projection:**

* If $k = 1$ (first task), there are no previous tasks, and the gradient update is performed without projection.
* If $k > 1$, retrieve the gradients related to previous tasks $G_{prev} = \{g_j\}_{j<k}$ (or a subset of them stored in memory M_G). These gradients can be:
  - The final gradients after training task T_j.
  - Gradients related to specific samples from previous tasks.
  - In practice, a small number of samples from previous tasks are usually used to calculate their gradients during the training of the new task.

  The OGD paper specifically uses the idea that the gradients of the previous tasks' loss functions can be calculated (either based on a small set of previous task data stored in a "replay memory") at each step. Assume $g_j = \nabla_\theta L_j(\theta; X_{replay}, Y_{replay})$ for tasks T_j where $j < k$ and X_{replay}, Y_{replay} are samples from previous tasks.

(c) **Gradient Projection:**

* Project the current gradient g_k onto the space orthogonal to the previous gradients G_{prev}. This process is performed as a linear projection operation. In the OGD paper, this projection is proposed as follows (which is a closed-form solution for linear constraints): Let G be a matrix whose columns are the gradients of previous tasks g_j (typically $g_j = \nabla L_j(\theta)$ for $j < k$). The desired update $\Delta\theta$ should minimize $\nabla L_k(\theta)$, with the constraint that $G^T \Delta\theta = 0$. This optimization problem can be solved using Lagrange multipliers, leading to the following solution for the projected gradient $\tilde{g}_k$:

$$\tilde{g}_k = g_k - \sum_{v \in s} proj_v(g)$$

  This formula projects the gradient g_k onto the space orthogonal to the column vectors of G.

(d) **Parameter Update:**

* Update network parameters using the projected gradient:

$$\theta \leftarrow \theta - \eta \tilde{g}_k$$

  where η is the learning rate.

(e) **Gradient Memory Update:**

* After training task T_k is complete, add the gradient corresponding to T_k (e.g., the final gradient $\nabla L_k(\theta)$) to the gradient memory set M_G for future use in projecting subsequent tasks.

5 Practical Notes and Considerations

1. **Storing Gradients:** Storing all gradients of previous tasks can be memory intensive. OGD suggests storing only a small and important subset of gradients or gradients of recent tasks.

2. **Replay Memory Samples:** To calculate the gradients of previous tasks during the training of a new task (on which the network is not explicitly trained), OGD uses a small replay memory to store a few samples from each previous task. These samples are used to calculate the gradients g_j at each step.

3. **Evaluation Metrics:** OGD's performance is evaluated by measuring accuracy on previous tasks after training on new tasks (to assess the extent of forgetting) as well as accuracy on the current task (to assess learning ability).

4. **Comparison with Other Methods:** OGD is compared with methods such as EWC (Elastic Weight Consolidation), LWF (Learning without Forgetting), A-GEM (Averaged Gradient Episodic Memory), and SGD (Stochastic Gradient Descent). The main advantage of OGD is that it directly operates in the gradient space and avoids maintaining more complex information such as Fisher matrices (in EWC) or separate models (in LWF).

6 Numerical Example

6.1 Numerical Example for GD

A simple one-dimensional loss function:

$$L(w) = \frac{1}{2}(w-3)^2.$$

Gradient:

$$\nabla L(w) = w - 3.$$

Starting point $w^{(0)} = 0$ and learning rate $\eta = 0.1$. Initial steps:

$$w^{(1)} = w^{(0)} - 0.1\,(w^{(0)} - 3) = 0 - 0.1\,(0-3) = 0.3,$$
$$w^{(2)} = 0.3 - 0.1\,(0.3-3) = 0.3 - 0.1\,(-2.7) = 0.57,$$
$$w^{(3)} = 0.57 - 0.1\,(0.57-3) = 0.57 - 0.1\,(-2.43) = 0.813.$$

This continues until convergence to an approximate value of $w^* \approx 3$.

6.2 Numerical Example for OGD

In this simple example, assume the model parameter is $w = (w_1, w_2)$ and we have two tasks:

$$L_1(w) = \frac{1}{2}[(w_1 - 1)^2 + (w_2)^2],$$
$$L_2(w) = \frac{1}{2}[(w_1)^2 + (w_2 - 1)^2].$$

Learning rate $\eta = 0.1$ and initial value $w_0 = (0, 0)$.

1. **Task 1**: Gradient $g_1 = \nabla L_1(w_0) = (w_1 - 1, w_2)\big|_{(0,0)} = (-1, 0)$.
2. Since $S = \emptyset$, projection has no effect, so $\widetilde{g}_1 = (-1, 0)$.
3. Update: $w \leftarrow w_0 - 0.1\,(-1, 0) = (0.1, 0)$.
4. Store gradient (for this simple example, the same g_1):
$$u_1 = g_1 = (-1, 0), \quad S = \{(-1, 0)\}.$$
5. **Task 2**: Gradient $g_2 = \nabla L_2(w) = (w_1, w_2 - 1)\big|_{(0.1,0)} = (0.1, -1)$.
6. Project onto vector u_1:
$$\text{proj}_{u_1}(g_2) = \frac{\langle (0.1, -1), (-1, 0) \rangle}{\langle (-1, 0), (-1, 0) \rangle}(-1, 0) = \frac{-0.1}{1}(-1, 0) = (0.1, 0).$$
7. Orthogonal gradient:
$$\widetilde{g}_2 = g_2 - \text{proj}_{u_1}(g_2) = (0.1, -1) - (0.1, 0) = (0, -1).$$
8. Update:
$$w \leftarrow (0.1, 0) - 0.1\,(0, -1) = (0.1, 0.1).$$
9. Store new gradient after projection:
$$u_2 = \widetilde{g}_2 = (0, -1), \quad S = \{(-1, 0), (0, -1)\}.$$

7 OGD Implementation in PyTorch

This section provides the main code snippets and explanations.

7.1 Importing Libraries and Setting Random Seed

```
import torch, torch.nn as nn, torch.optim as optim
import matplotlib.pyplot as plt, numpy as np
from copy import deepcopy

torch.manual_seed(42)
np.random.seed(42)
```

- `torch` and its submodules `nn`, `optim` for model definition and training.
- `matplotlib` for plotting and `numpy` for numerical computations.
- `manual_seed` fixes the random seed for reproducibility.

7.2 Device Selection (CPU or GPU)

```
device = "cuda" if torch.cuda.is_available() else "cpu"
```

The model and data are moved to `device` to utilize the GPU if available.

7.3 Defining Target Functions

```
def f_A(x): return torch.sin(x)
def f_B(x): return torch.sin(x**3)
def f_C(x): return torch.exp(-x**2)
```

Each function has different characteristics to simulate sequential tasks.

7.4 Creating the Dataset

```
def make_dataset(f, n=128, rng=(-5, 5), noise_std=0.1):
    a, b = rng
    x = torch.rand(n, 1, device=device) * (b - a) + a
    x, _ = torch.sort(x, dim=0)
    y = f(x) + noise_std * torch.randn_like(x)
    return x, y
```

Adding Gaussian noise to the output makes the training data more realistic.

7.5 Defining a Simple Model

```
class SimpleNN(nn.Module):
    def __init__(self, hidden=50):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, hidden), nn.ReLU(),
            nn.Linear(hidden, 1)
```

```
        )
    def forward(self, x):
        return self.net(x)
```

A two-layer network with 50 hidden neurons and ReLU activation function.

7.6 Helper Functions for OGD

```
def flat_gradients(model):
    return torch.cat([p.grad.view(-1)
        for p in model.parameters() if p.grad is not None])

def write_flat_to_grads(model, vec):
    i=0
    for p in model.parameters():
        if p.grad is None: continue
        n=p.numel()
        p.grad.copy_(vec[i:i+n].view_as(p))
        i+=n
```

- `flat_gradients`: Extracts all gradients into a flattened vector.
  - `write_flat_to_grads`: Writes a flattened vector back to parameter gradients.

7.7 Training Function with OGD

```
def train(model, x, y, epochs=250, lr=1e-2, memory=None):
    crit=nn.MSELoss()
    opt=optim.SGD(model.parameters(), lr=lr)
    for _ in range(epochs):
        opt.zero_grad()
        loss=crit(model(x), y)
        loss.backward()
        if memory:
            g=flat_gradients(model)
            for v in memory:
                g -= torch.dot(g,v)/(v.norm()**2+1e-8)*v
            write_flat_to_grads(model, g)
        opt.step()
    return loss.item()
```

- If `memory` (list of previous task gradients) is provided, the current gradient is first projected.

7.8 Storing Gradient of Each Task

```
def gradient_of_task(model, x, y):
```

```
    model.zero_grad()
    nn.MSELoss()(model(x), y).backward()
    return flat_gradients(model).detach()
```

To be used in OGD's memory, the task gradient is detached from the computational graph.

7.9 Simulating SGD and OGD Branches

The following code first trains on Task A, then trains on B and C respectively with and without OGD, and reports the MSE on A after each step.

```
def run_branch(name, use_ogd):
    base_model = SimpleNN().to(device)
    train(base_model, x_A, y_A)
    mse_A_start = nn.MSELoss()(base_model(x_A), y_A).item()
    memory = []
    if use_ogd:
        memory.append(gradient_of_task(base_model, x_A, y_A))
    model_B = deepcopy(base_model)
    model_C = deepcopy(base_model)
    train(model_B, x_B, y_B, memory=memory if use_ogd else None)
    mse_A_after_B = nn.MSELoss()(model_B(x_A), y_A).item()
    if use_ogd:
        memory.append(gradient_of_task(model_B, x_B, y_B))
    train(model_C, x_C, y_C, memory=memory if use_ogd else None)
    mse_A_after_C = nn.MSELoss()(model_C(x_A), y_A).item()
    return base_model, model_B, model_C

model_A0_sgd, model_B_sgd, model_C_sgd = run_branch("SGD branch", False)
model_A0_ogd, model_B_ogd, model_C_ogd = run_branch("OGD branch", True)
...
```

7.10 Plotting Results

```
def plot_fit(model, x, y, title):
    model.eval()
    with torch.no_grad(): p = model(x).cpu()
    plt.scatter(x.cpu(), y.cpu(), s=10)
    plt.plot(x.cpu(), p, c="r")
    plt.title(title); plt.grid(); plt.show()

plot_fit(...)
```

The plots show how the OGD branch model better preserves Task A's information.

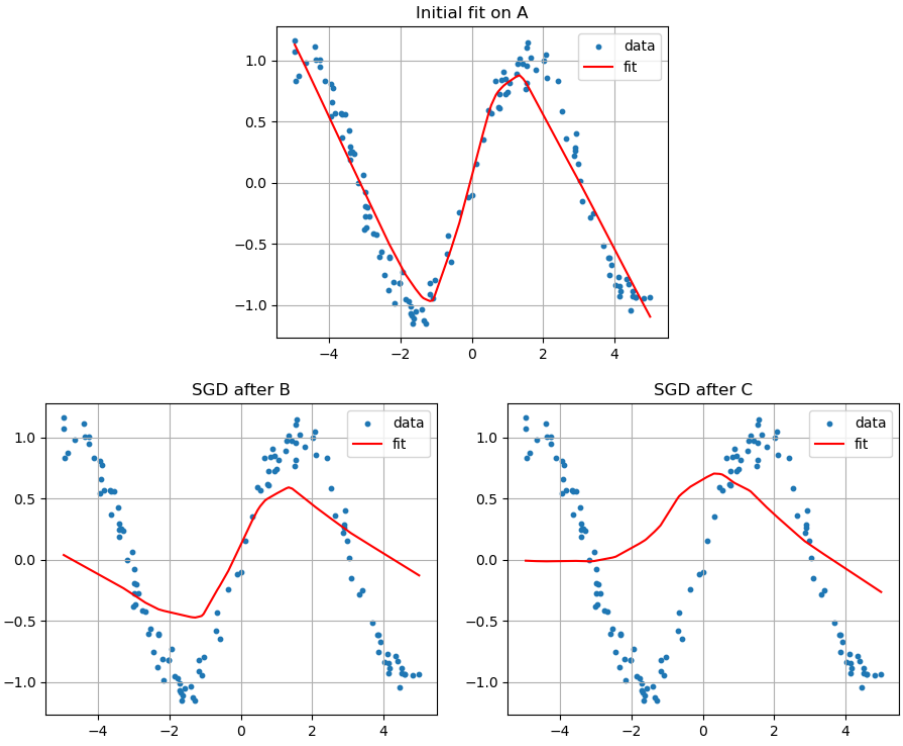
8 Analysis of Execution Results

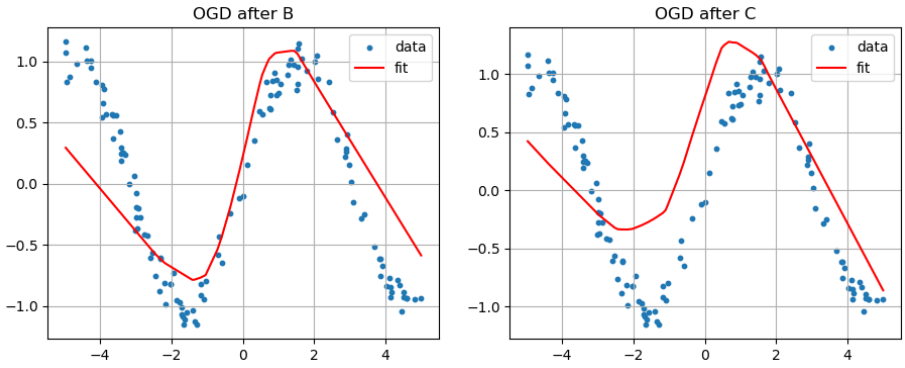
The execution results compare two different branches (SGD and OGD) on Task A (the function $\sin(x)$):

```
=== SGD branch | SGD ===
MSE_A after Task A: 0.0399
Training on Task B …
MSE_A after B: 0.3274
Training on Task C …
MSE_A after C: 0.5061

=== OGD branch | OGD ===
MSE_A after Task A: 0.0563
Training on Task B …
MSE_A after B: 0.1959
Training on Task C …
MSE_A after C: 0.2544
```

Plots:





9 Comparison

- OGD minimizes interference between task learning by orthogonally projecting gradients.
- The final error in OGD is approximately half the equivalent error in simple SGD.
- This method reduces catastrophic forgetting without needing to store previous data.

10 Conclusion

The Orthogonal Gradient Descent (OGD) algorithm provides an effective approach to combat catastrophic forgetting in continual learning. By projecting the new task's gradient into a space orthogonal to the gradients of previous tasks, this algorithm ensures that parameter updates have minimal impact on prior knowledge. OGD offers a solution from the perspective of parameter space that can be beneficial in various practical continual learning applications.

11 References

- *PyTorch: An Imperative Style, High-Performance Deep Learning Library*
- *Orthogonal Gradient Descent for Continual Learning (Farajtabar, et al.)*