CS330 Review Session: MAML

Stanford
ARTIFICIAL
INTELLIGENCE

What We'll Cover Today

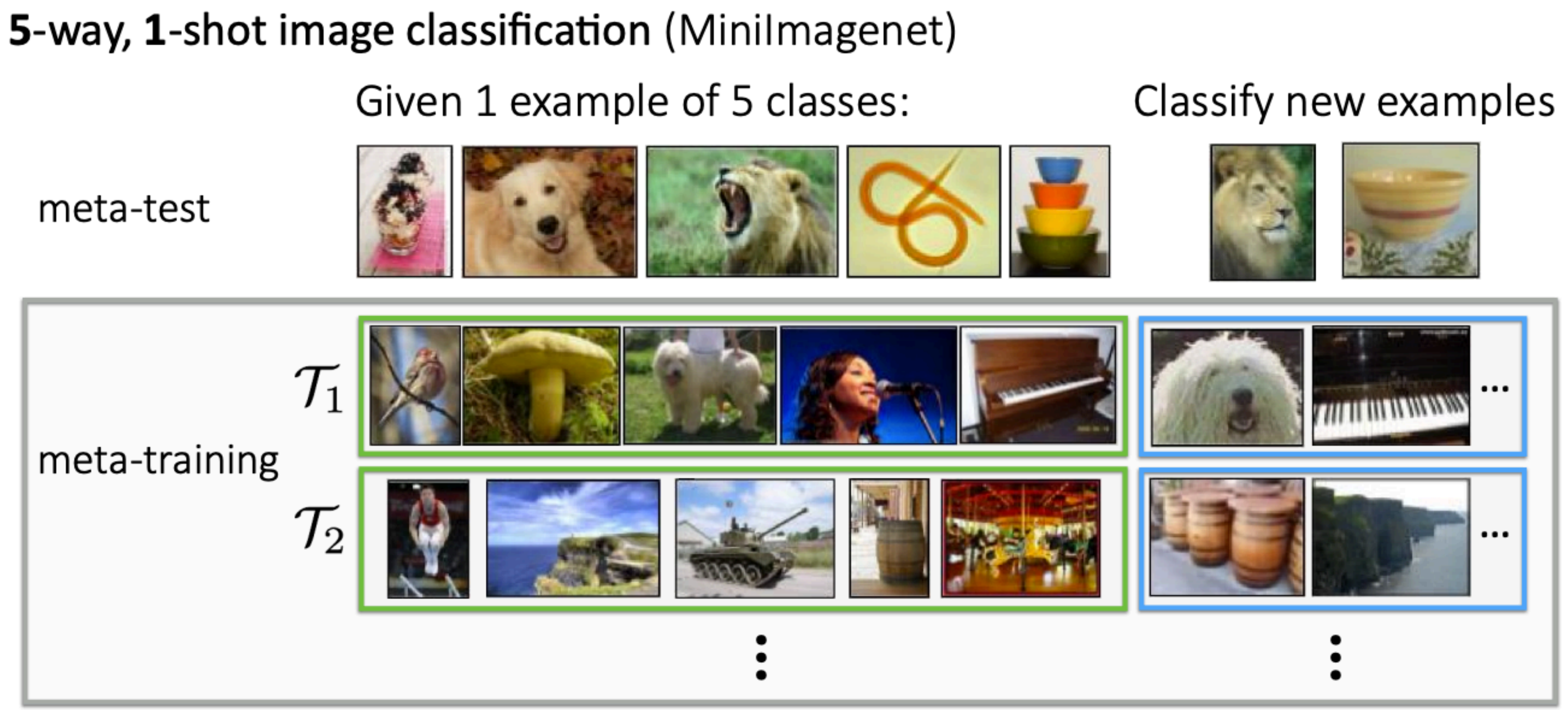
1. Review of the meta-learning problem setup
2. Model-Agnostic Meta-Learning (MAML)
3. Useful PyTorch functions

No pytorch code, but will connect the lecture materials to the details of practical implementation.

What We'll Cover Today

1. **Review of the meta-learning problem setup**
2. Model-Agnostic Meta-Learning (MAML)
3. Useful PyTorch functions

Running Example


5-way, **1**-shot image classification (MiniImagenet)
Given 1 example of 5 classes:
Classify new examples
meta-test
meta-training
$\mathcal{T}_1$
$\mathcal{T}_2$
...

The task can be ***any ML problem***: regression, language generation…

Inner Loop Learning

T_i

i-th task

D_i^{tr}

Training data

D_i^{ts}

Test data

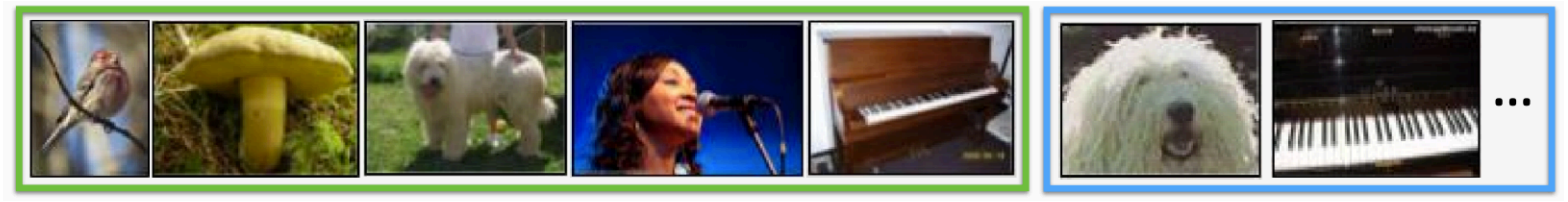
We adapt the model on this data…

and test on this data

We repeat this over ***many*** tasks

Task Sampling (5-way 1-shot classication)

T_i



D_i^{tr} D_i^{ts}

To sample one task:

1. Sample 5 classes
2. Training set: sample 1 image from each class
3. Test set: sample N images from each class

(training and test set must not overlap!)

Meta-Train vs Meta-Test Tasks

We partition classes into:
(train, val, test) classes.
→ sample 5 classes from the appropriate split!

To sample one task:

1. **Sample 5 classes**
2. Training set: sample 1 image from each class
3. Test set: sample N images from each class

(training and test set must not overlap!)

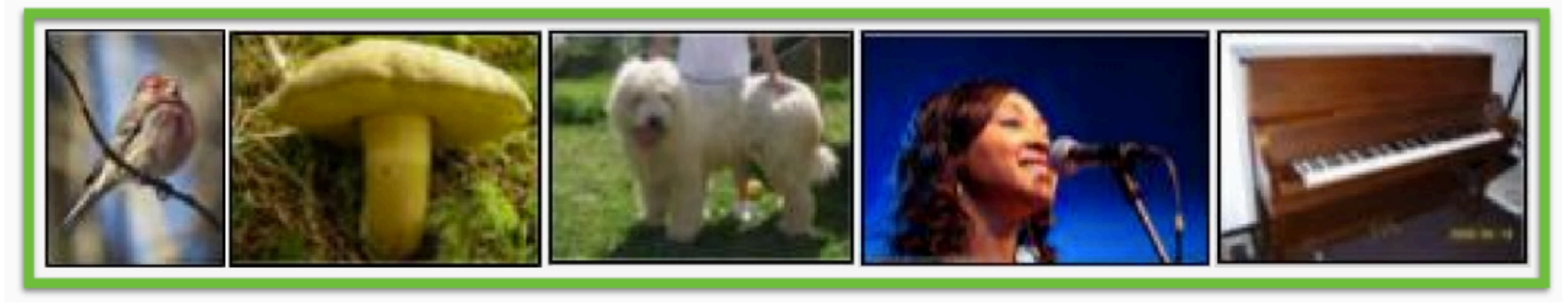
What We'll Cover Today

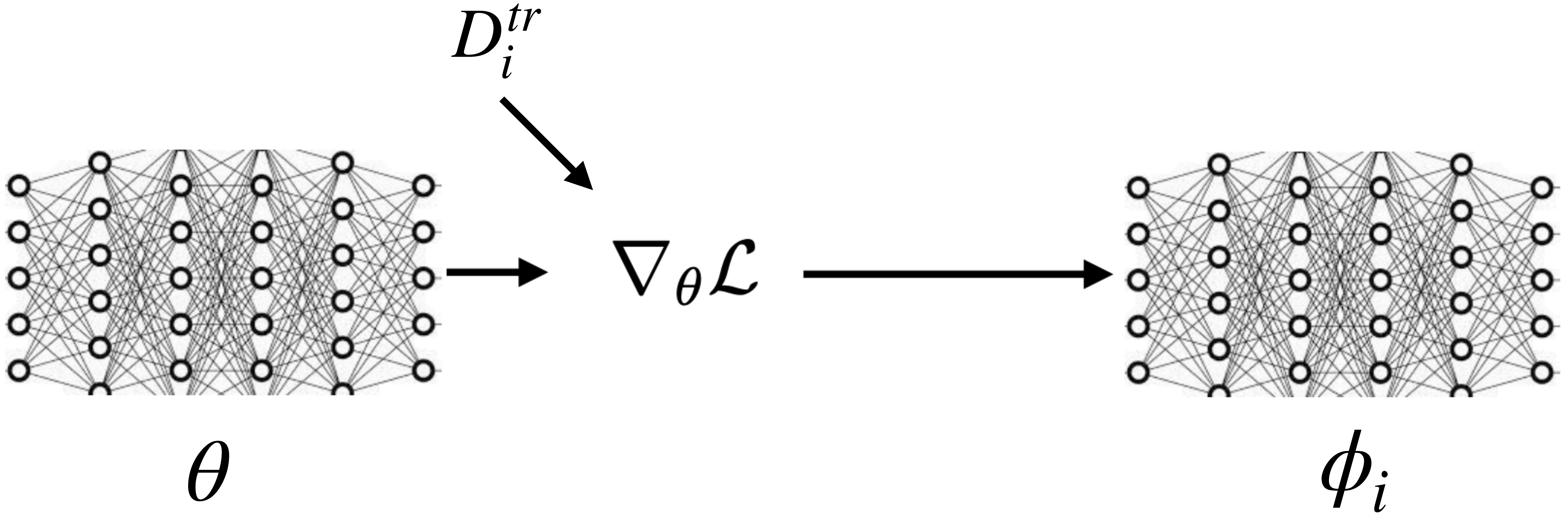
1. Review of the meta-learning problem setup
2. **Model-Agnostic Meta-Learning (MAML)**
3. Useful PyTorch functions

MAML Inner Loop

$$\min_{\theta} \sum_{\text{task } i} \mathcal{L}(\theta - \alpha \nabla_{\theta} \mathcal{L}(\theta, \mathcal{D}_i^{\text{tr}}), \mathcal{D}_i^{\text{ts}})$$

T_i



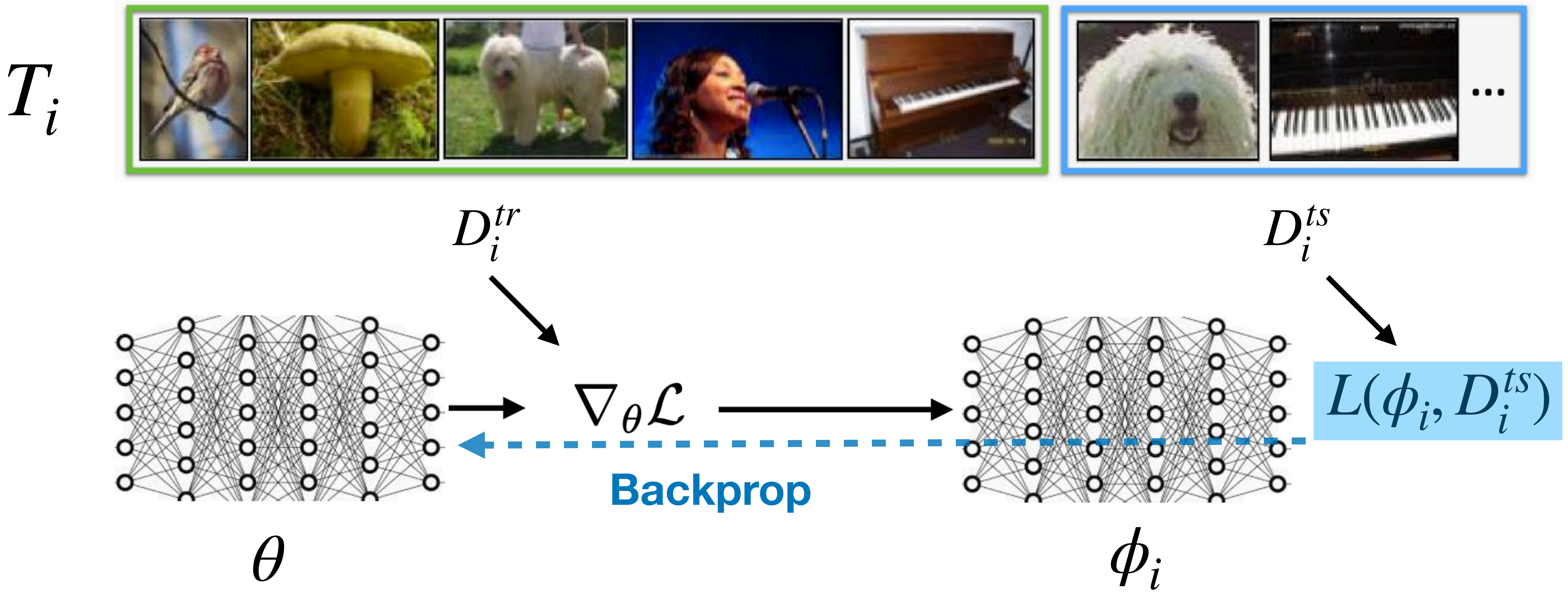


Initial network parameters

Parameters adapted to task i

MAML Outer Loop

$$\min_{\theta} \sum_{\text{task } i} \mathcal{L}(\theta - \alpha \nabla_{\theta} \mathcal{L}(\theta, \mathcal{D}_i^{\text{tr}}), \mathcal{D}_i^{\text{ts}})$$

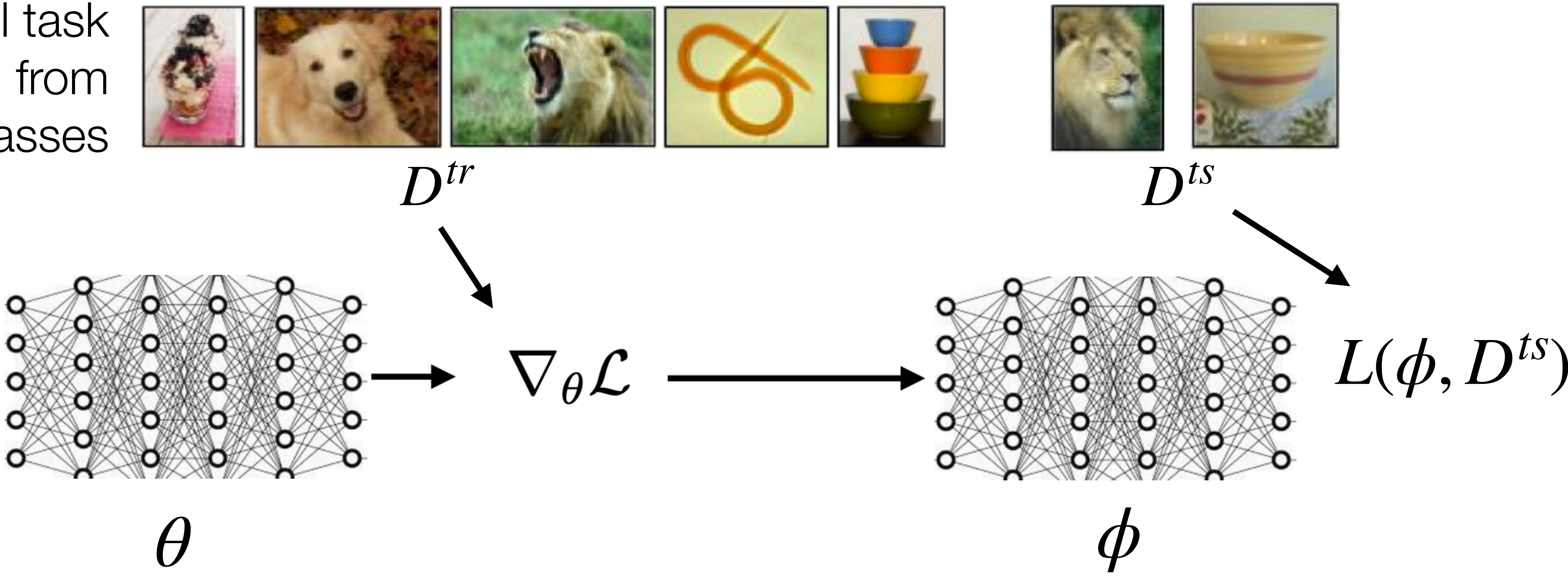


Initial network parameters

Parameters adapted to task i

MAML Meta-Testing

Novel task constructed from unseen classes



Meta-learned network parameters

Parameters adapted to test task

MAML Summary

Meta-Training

Repeat until convergence:

1. Sample task $T_i = (D_i^{tr}, D_i^{ts})$
2. Optimize $\phi_i \leftarrow \theta - \alpha \nabla_\theta L(\theta, D_i^{tr})$
3. Update $\theta \leftarrow \theta - \beta \nabla_\theta L(\phi_i, D_i^{ts})$

Meta-Testing

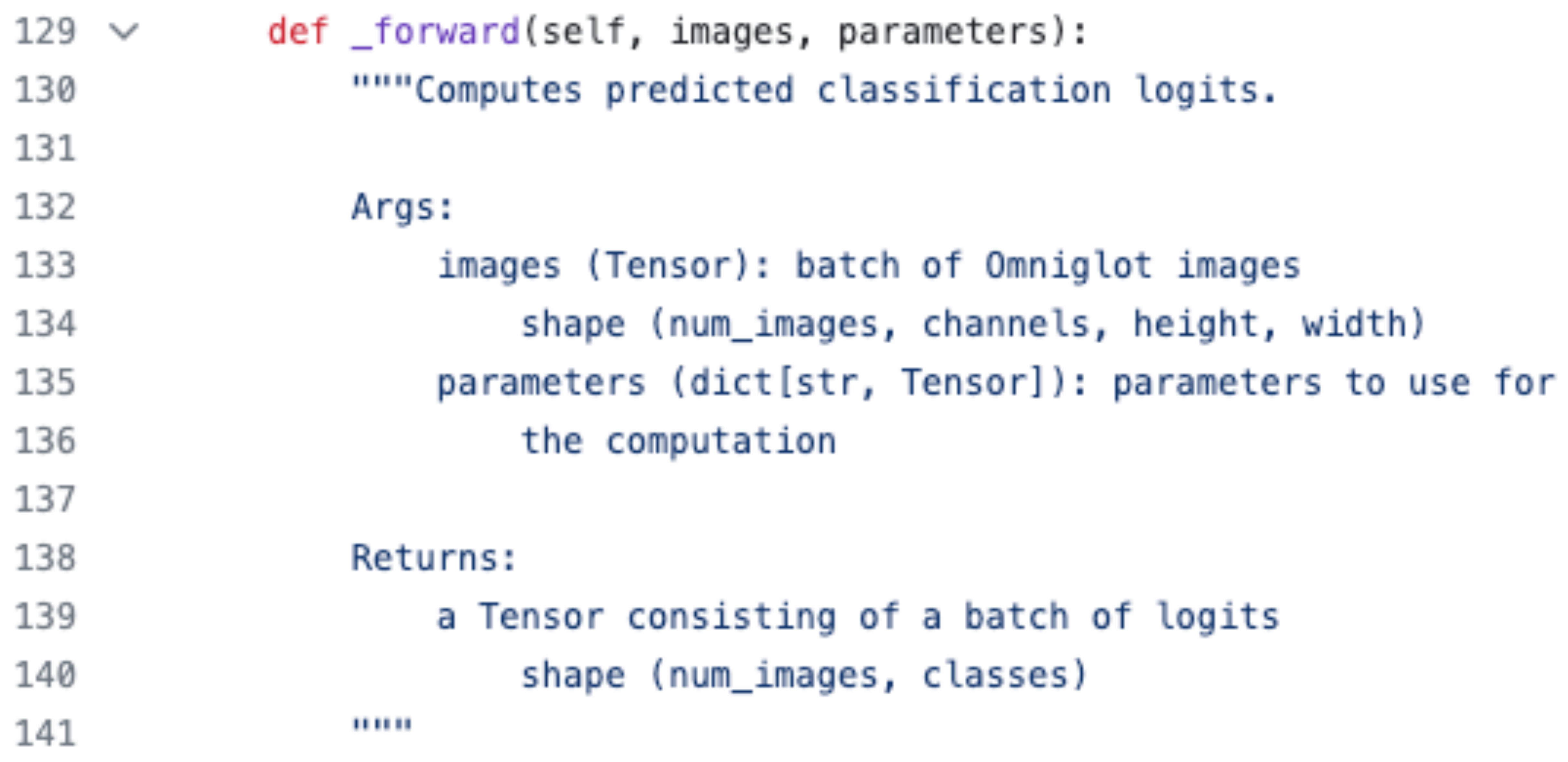
1. Given task $T = (D^{tr}, D^{ts})$
2. Optimize $\phi \leftarrow \theta - \alpha \nabla_\theta L(\theta, D^{tr})$
3. Make predictions on D^{ts} using ϕ

In practice, we parallelize both meta-training and meta-testing with **minibatches of tasks**.

What We'll Cover Today

1. Review of the meta-learning problem setup
2. Model-Agnostic Meta-Learning (MAML)
3. **Useful PyTorch functions**

_forward()

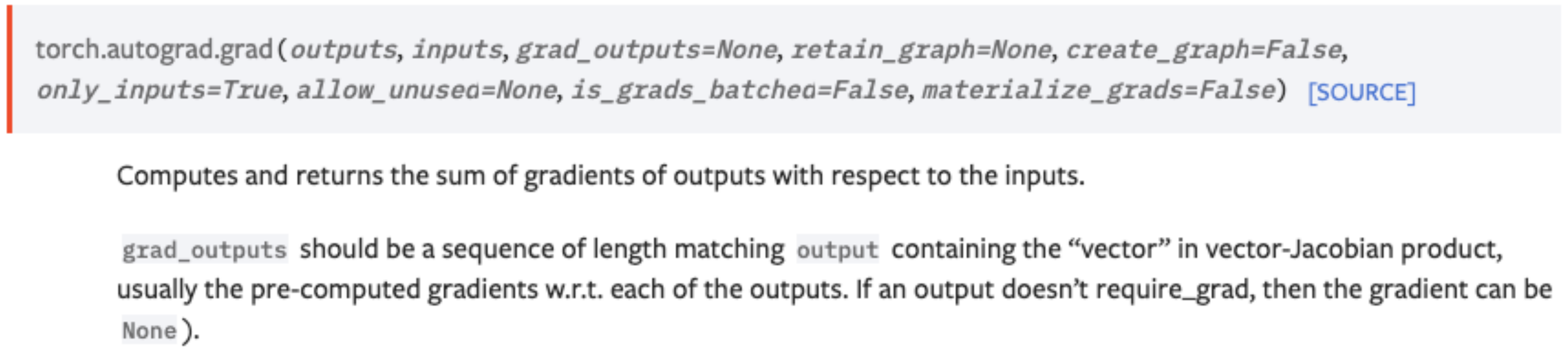


```
    def _forward(self, images, parameters):
        """Computes predicted classification logits.

        Args:
            images (Tensor): batch of Omniglot images
                shape (num_images, channels, height, width)
            parameters (dict[str, Tensor]): parameters to use for
                the computation

        Returns:
            a Tensor consisting of a batch of logits
                shape (num_images, classes)
        """
```

In the provided code, the provided _forward() is stateless: it takes current model parameters as input.

torch.autograd.grad()

$$\min_{\theta} \sum_{\text{task } i} \mathcal{L}(\theta - \alpha \nabla_{\theta} \mathcal{L}(\theta, \mathcal{D}_i^{\text{tr}}), \mathcal{D}_i^{\text{ts}})$$


torch.autograd.grad(*outputs*, *inputs*, *grad_outputs=None*, *retain_graph=None*, *create_graph=False*, *only_inputs=True*, *allow_unused=None*, *is_grads_batched=False*, *materialize_grads=False*) [SOURCE]

Computes and returns the sum of gradients of outputs with respect to the inputs.

`grad_outputs` should be a sequence of length matching `output` containing the "vector" in vector-Jacobian product, usually the pre-computed gradients w.r.t. each of the outputs. If an output doesn't require_grad, then the gradient can be `None`).

If you want to backpropagate through the gradient later:
torch.autograd.grad(outputs, inputs, create_graph=True)

Otherwise:
torch.autograd.grad(outputs, inputs, create_graph=False)

parameters

$$\min_{\theta} \sum_{\text{task } i} \mathcal{L}(\theta - \alpha \nabla_{\theta} \mathcal{L}(\theta, \mathcal{D}_i^{\text{tr}}), \mathcal{D}_i^{\text{ts}})$$

```
parameters = {
    k: torch.clone(v)
    for k, v in self._meta_parameters.items()
}
```

Parameters are a dictionary with (parameter_name, parameter_value) pairs.
You should explicitly compute the updated parameter.

What We Covered Today

1. Review of the meta-learning problem setup
2. Model-Agnostic Meta-Learning (MAML)
3. Useful PyTorch functions