第3章 深度学习框架入门

苏统华
软件学院
哈尔滨工业大学

本章内容

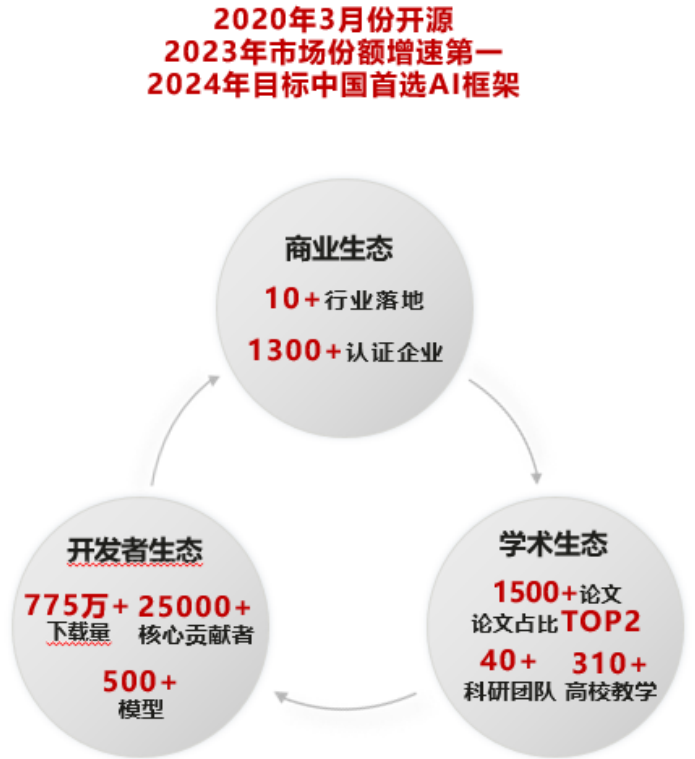
1. 昇思MindSpore框架介绍
2. 基于昇思MindSpore的手写数字识别实践
3. 昇思MindSpore的编程范式
4. 昇思MindSpore的动静统一
5. 昇思MindSpore的高阶操作

3.1昇思MindSpore框架介绍

苏统华
软件学院
哈尔滨工业大学

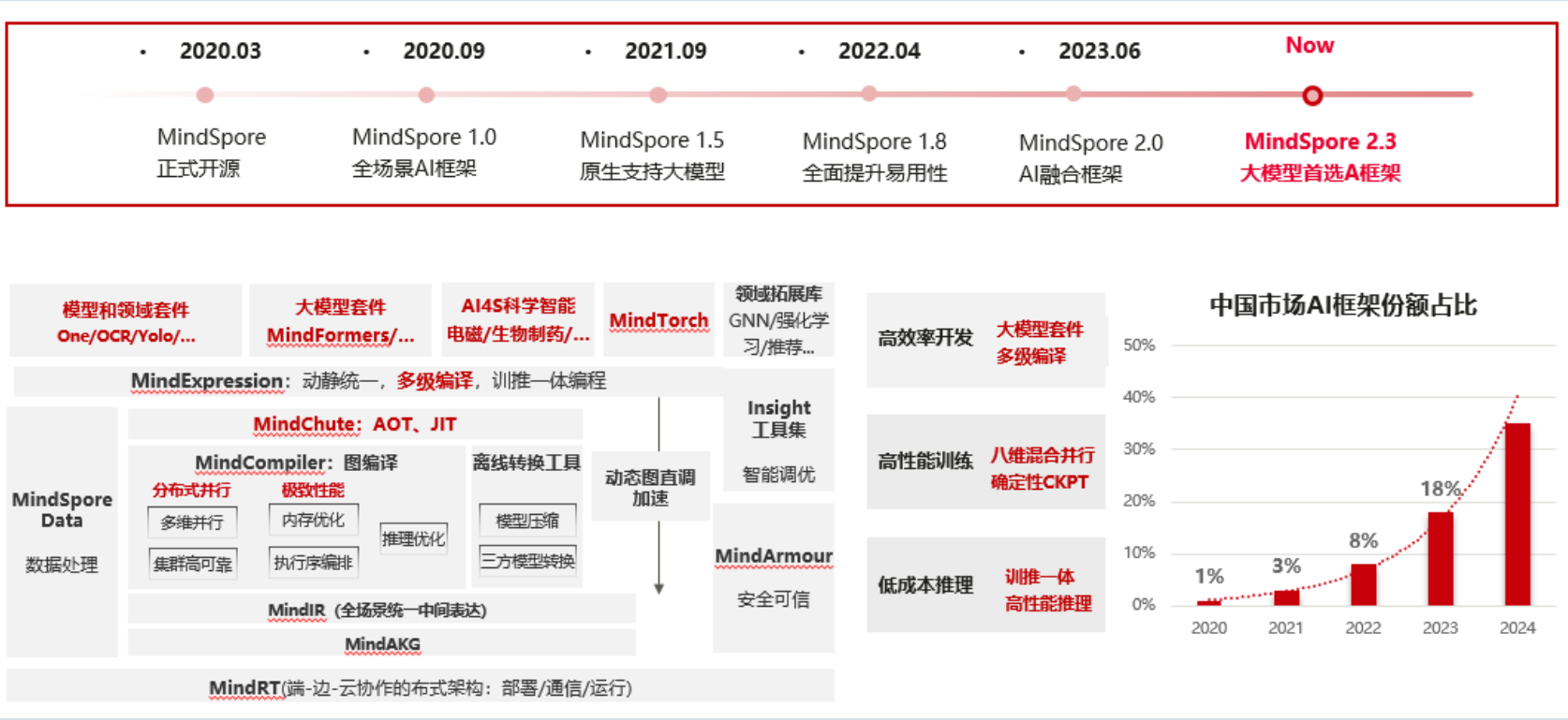
昇思Mindspore整体介绍

- MindSpore开源以来，秉持全场景协同、全流程极简、全架构统一三大价值主张，致力于增强开发易用性、提升原生支持大模型和AI+科学计算的体验。”向上使能AI模型创新，对下兼容多样性算力（NPU、GPU、CPU），如今已演进至2.3版本。
- To B：面向AI计算中心、电信、制造、金融、互联网、交通、政府、能源、高校科研、安平，给国计民生行业提供一个更有的AI选择；
- To C：使能终端1+8+N，智能手机，大屏、音箱、眼镜、手表、车机、耳机、平板、PC等消费级设备；



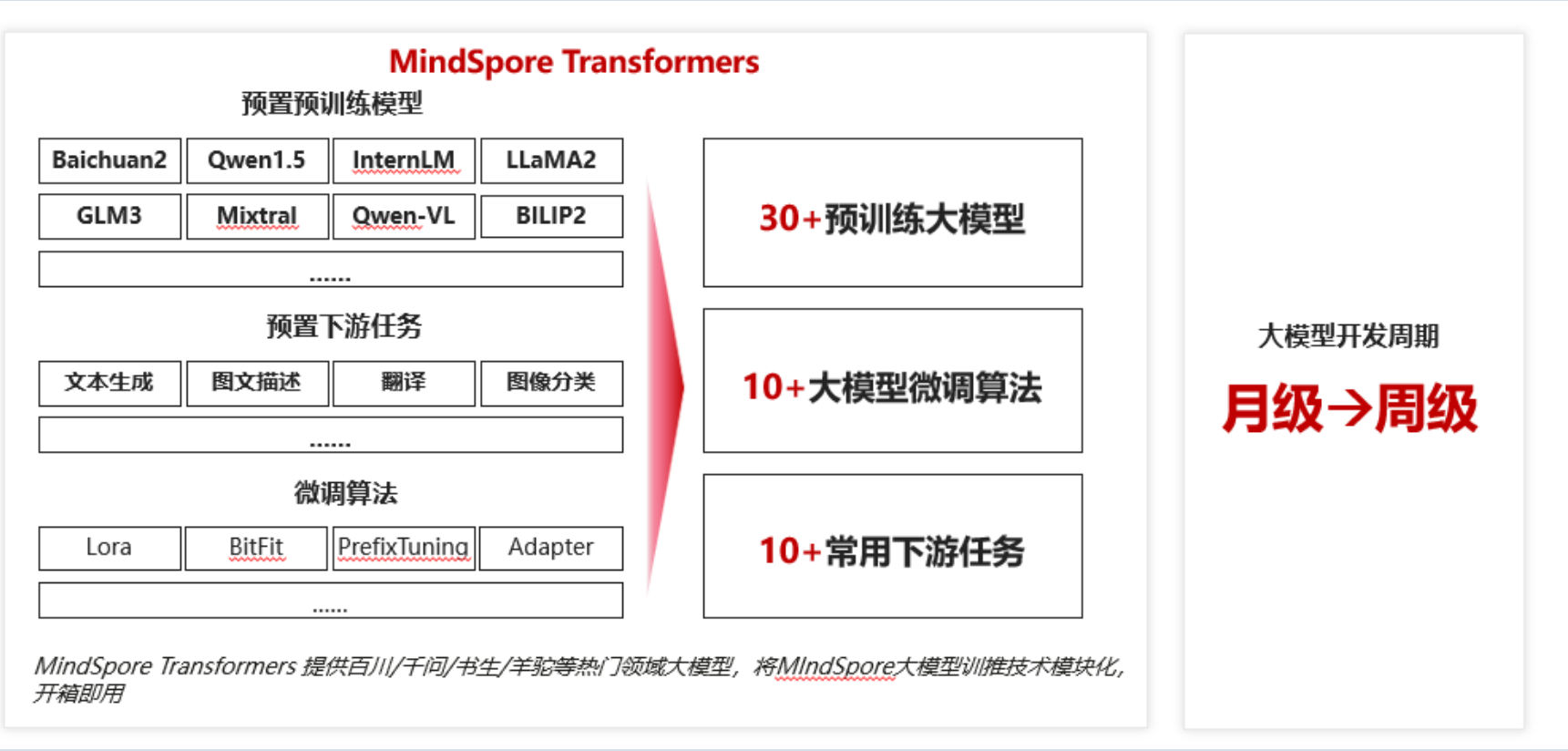
昇思Mindspore整体介绍

- **核心框架层面**，完善动态图、以及静态图语法支持度，保持易用性的同时，全面提升JIT图编译性能
- **大模型使能方面**，基于大模型套件降低开发成本，同时构建大模型分布式训练推理加速能力，提供高性能的分布式训推基础设施，万亿参数模型TTA效率提升。
- **AI4S领域**，构建了电磁仿真、流体力学、生物计算三个套件，同时构建融合框架能力，提供函数式微分，计算图编译加速，支撑AI4S突破前研特性



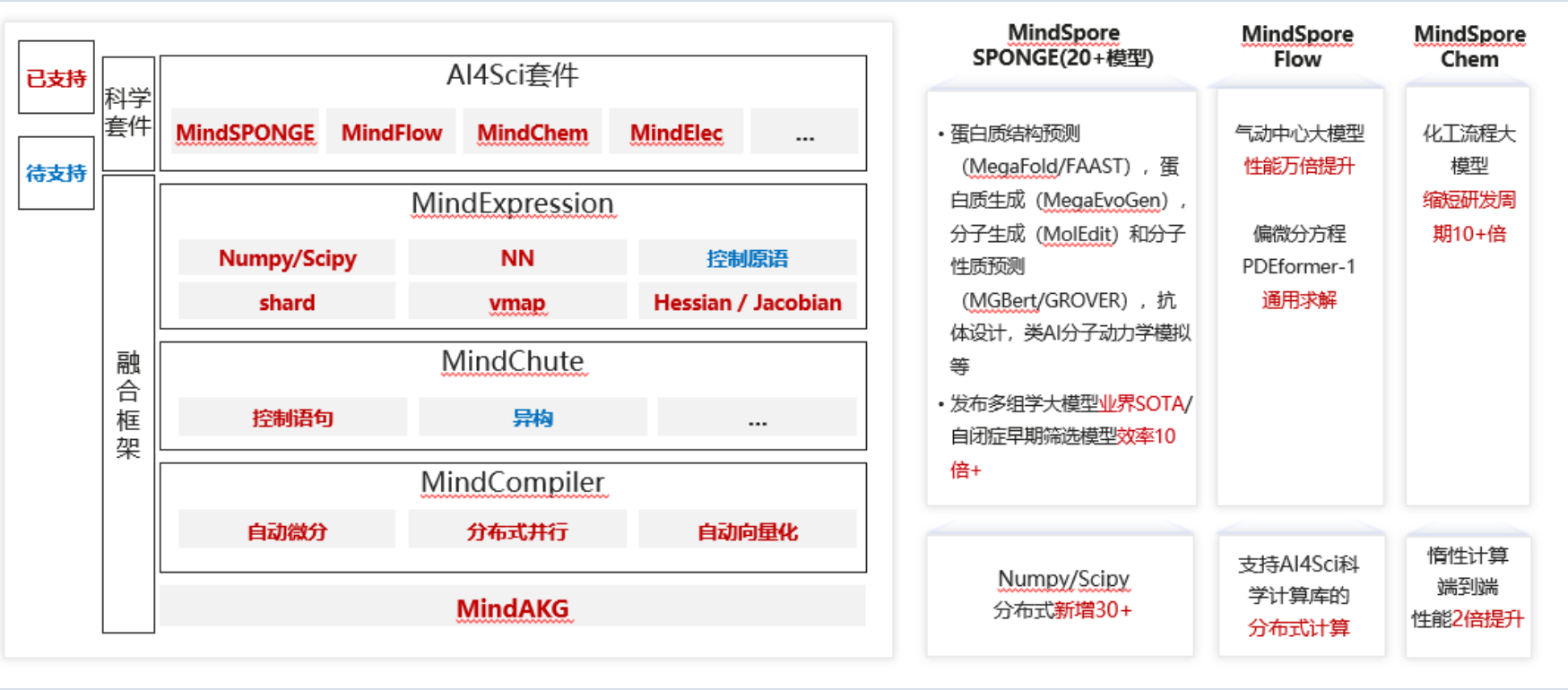
昇思Mindspore套件介绍

- 为了降低开发门槛，昇思MindSpore不断完善MindSpore Transformers大模型套件，新版中新增了预置Qwen、Llama2、Qwen-VL和Mistral等10+主流大模型，目前我们大模型套件已经预制了30+预训练大模型，用户不需要通过传统的逐个算子的构图方式，极大地提升了开发效率；
- 此外，相比于上一个版本增加了多种微调算法和下游任务，目前已经支持10+大模型算法和10+常用下游任务，用户就可以针对特有业务场景快速开发。相比于传统的开发方式，基于大模型套件，一周时间即可完成整个大模型全流程的开发工作。



昇思Mindspore套件介绍

- 在科学计算（AI4S）领域，大模型+AI4S创新了科学计算范式，昇思MindSpore也在同步不断孵化科学领域基础大模型，覆盖AI生物计算领域、流体仿真领域、化学材料领域等等。融合框架以动态图优先，提升用户开发体验，加速应用性能，完善算子覆盖度，提升向量化等基础能力，提供控制算子（scan）提升控制流场景编译性能。



昇思Mindspore套件介绍

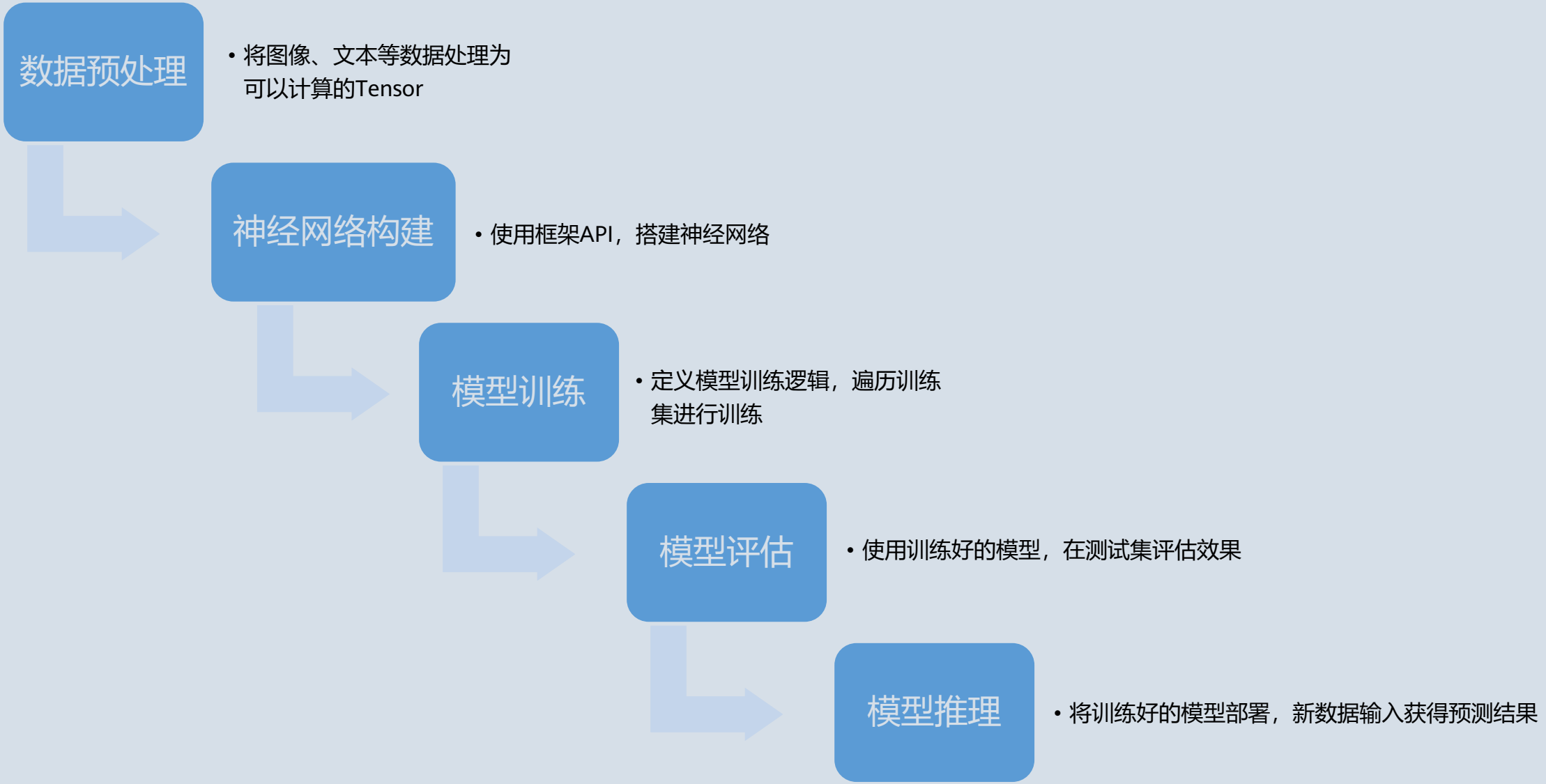
- **MindSpore NLP**：全面对标Huggingface Transformers，实现高效易用，可无缝对接Huggingface社区；
- **MindSpore CV**：面向CV 类基础骨干模型，昇思 MindSpore 使用新的训练策略刷新了模型精度，方便开发者做下游任务；
- **MindSpore OCR**：OCR 套件集成业界 SOTA 模型，并刷新精度，平均提升 1 个点，同时做了推理性能加速。整体端到端提速 20%；
- **MindSpore One**：生成式套件，提供了生成式文生图/视频/音频/3D视觉SOTA模型集合，SD/Video LDM/AudioLM等丰富的全量训练微调组件。

MindSpore NLP	**NLP套件** 全面对标Huggingface Transformers的高效易用NLP套件，支持60+Transformer大模型，可无缝衔接Huggingface社区。 代码地址：https://github.com/mindspore-lab/mindnlp
MindSpore CV	**CV基础套件** 提供简单易用模块接口(数据增强/模型构建等)，模型训练性能提速1.x~2.x倍，精度提升1~2% 代码地址：https://github.com/mindspore-lab/mindcv
MindSpore OCR	**OCR套件** 提供极简易用API、流SOTA模型，模型组件分类覆盖OCR模型开发全流程，典型场景精度领先 代码地址：https://github.com/mindspore-lab/mindocr
MindSpore One	**生成式套件** 提供生成式文生图/视频/音频/3D视觉SOTA模型集合，SD/Video LDM/AudioLM等丰富的全量训练微调组件 代码地址：https://github.com/mindspore-lab/mindone

3.2 基于昇思MindSpore的手写数字识别实践

苏统华
软件学院
哈尔滨工业大学

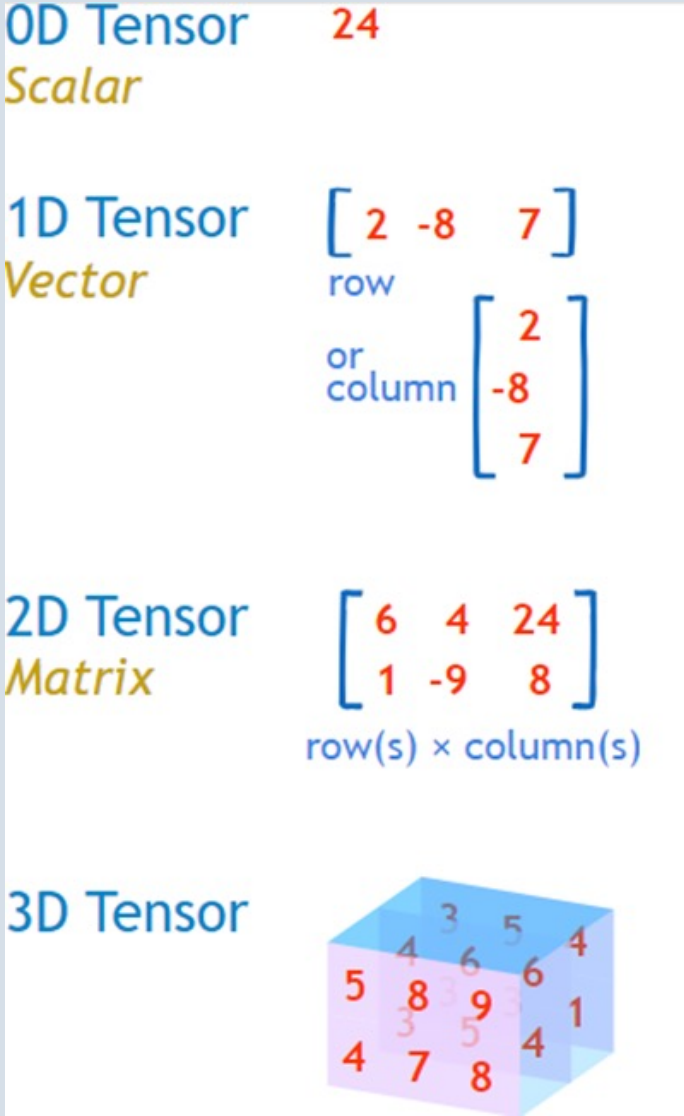
深度学习原理简介——深度学习开发全流程



张量——网络运算中的基本数据结构

数据预处理的目的是将图像、文本、音频等数据转换为模型可计算的张量（Tensor），让我们首先了解下张量的概念：张量（Tensor）在数学中是一个代数对象，描述了与矢量空间相关的代数对象集之间的多重线性映射。在深度学习里，Tensor实际上就是一个多维数组，而Tensor的目的是能够创造更高维度的矩阵、向量。

- **0维张量代表的是标量（数字）**
- **1维张量代表的是向量**
- **2维张量代表的是矩阵**
- **3维张量可表示内容更为广泛，如单张RGB图片、文本数据、时间序列数据等等。随着维度的上升，张量所能表达的信息会愈加丰富。**



张量的构建

在MindSpore中，我们可以通过mindspore.Tensor来构建张量。构造张量时，支持传入Tensor、float、int、bool、tuple、list和numpy.ndarray类型。

此处我们演示最简单直观的一种方式 —— 直接根据数据，依次生成图片中不同维度的Tensor。

```
1. from mindspore import Tensor
2. #0D Tensor
3. tensor_0d = Tensor(24)
4.
5. #1D Tensor
6. tensor_1d = Tensor([2, -8 ,7])
7.
8. #2D Tensor
9. tensor_2d = Tensor([(6, 4 ,24), (1, -9 ,8)])
10.
11. #3D Tensor
12. tensor_3d = Tensor([[(5, 8 ,9), (4, 7 ,8)], [(4, 6 ,6),
(3, 5 ,3)], [(3, 5 ,4), (3, 2 ,1)]])
```

张量的属性

张量的属性包括形状、数据类型、维数等，我们可以通过打印上述属性来检查生成的张量。

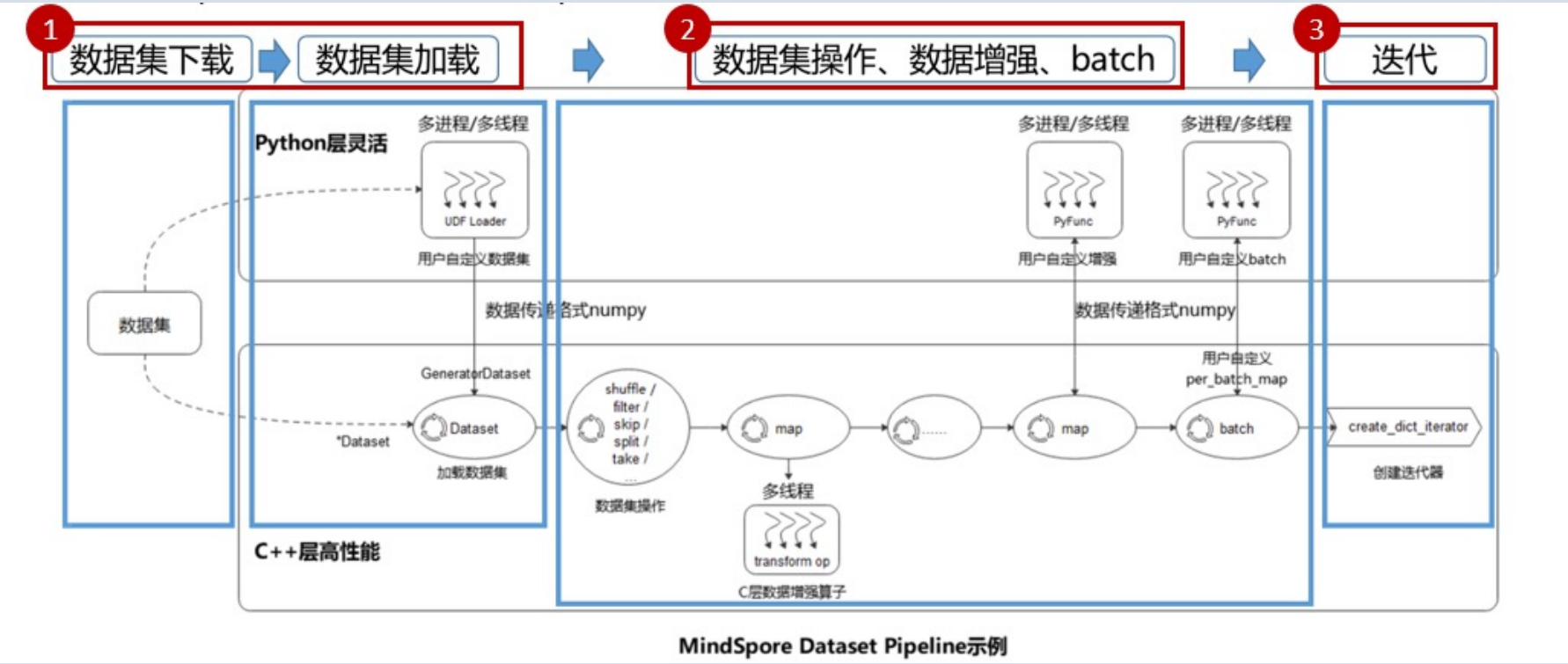
- **形状（shape）：Tensor的shape，是一个tuple。**
- **数据类型（dtype）：张量的数据类型，用于表示张量中每个元素的类型，例如浮点数、整数等。**
- **维数（ndim）： 张量的维度数，即其形状中的轴数，是一个整数值。**

```
1. print (f" 1. 3D tensor: \n{tensor_3d}")
2. print (f" 2. Shape of 3D tensor: {tensor_3d.shape}")
3. print (f" 3. Data type of 3D tensor: {tensor_3d.dtype}")
4. print (f" 4. Dimension of 3D tensor: {tensor_3d. ndim}")
5.
6. outputs:
7. 1. 3D tensor:
8. [[[5 8 9]
9. [4 7 8]]
10.
11. [[4 6 6]
12. [3 5 4]]
13.
14. [[3 5 4]
15. [3 2 1]]]
16.
17. 2. Shape of 3D tensor: (3, 2, 3)
18. 3. Data type of 3D tensor: Int64
19. 4. Dimension of 3D tensor: 3
```

数据预处理——将图像、文本等数据处理为可以计算的Tensor

在数据预处理中，MindSpore提供了基于Pipeline的数据引擎，通过数据集和数据变换实现高效的数据预处理。一般在将数据送入神经网络进行训练前，会遵循如下步骤进行处理。

1. **数据集的下载与加载**
2. **数据集操作、数据增强、batch**
3. **数据集迭代**



MindSpore Dataset Pipeline示例

数据集下载与加载

mindspore.dataset提供的接口仅支持解压后的数据文件，因此我们使用download库下载数据集并解压。

解压后，可通过调用mindspore.dataset对应的数据集API进行加载。 mindspore.dataset提供了很多内置的文本、图像、音频等数据集加载接口，在本案例中，我们使用MnistDataset加载MNIST数据集。

```
from mindspore.dataset import MnistDataset, vision,transforms
1. # Download data from open datasets
2. from download import download
3.
4. url = "https://mindspore-website.obs.cn-north-4.myhuaweicloud.com/" \
5. "notebook/datasets/MNIST_Data.zip"
6. path = download(url, "./", kind="zip", replace=True)
7.
8. train_dataset = MnistDataset('MNIST_Data/train')
9. test_dataset = MnistDataset('MNIST_Data/test')
```

数据集下载与加载

完成数据集加载后，我们可以看到数据集返回的对象为一个28X28X1的Tensor。
其中28X28表示图片的高（Height）和宽（Width），1表示图片的通道数（Channels）。常见的图片类型中，灰度图的通道数为1，RGB的通道数为3

```
1. image, label = next(train_dataset.create_tuple_iterator())
2. print(image.shape, image.dtype)
3.
4. outputs:
5. <class 'mindspore.common.tensor.Tensor'>
6. Shape of image [H W C]: (28, 28 ,1)
```

数据集处理和数据增强

通常情况下，直接加载的原始数据并不能直接送入神经网络进行训练，此时我们需要对其进行数据预处理。MindSpore提供不同种类的数据变换（Transforms），配合数据处理Pipeline来实现数据预处理。所有的Transforms均可通过map方法传入，实现对指定数据列的处理。

我们可以通过get_col_names获取数据列，用于后续制定数据列进行数据预处理

```
1. print(train_dataset.get_col_names())
2.
3. outputs:
4. ['image', 'label']
```

mindspore.dataset.vision模块提供一系列针对图像数据的Transforms。
在Mnist数据处理过程中，使用了Rescale、Normalize，HWC2CHW变换和批处理（batch）。

rescale

rescale变换用于调整图像像素值的大小，包括两个参数：

- rescale：缩放因子。
- shift：平移因子。

输出图像的像素大小为：output = image * rescale + shift

```
1. rescale = vision.Rescale(1.0 / 255.0, 0)
2. rescaled_image = rescale(random_image)
3. print(rescaled_image)
```

随机生成一个像素值在[0, 255]的图像

```
1. outputs:
2. [[170 10 218 ... 81 128 96]
3. [ 2 107 146 ... 239 178 165]
4. [232 137 235 ... 222 109 216]
5. ...
6. [193 140 60 ... 72 133 144]
7. [232 175 58 ... 55 110 94]
8. [152 241 105 ... 187 45 43]]
```

使用Rescale后的每个像素值都缩放到了原本数值的1/255

```
1. outputs:
2. [[0.6666667 0.03921569 0.854902 ... 0.31764707 0.5019608 0.37647063]
3. [0.00784314 0.41960788 0.57254905 ... 0.93725497 0.69803923 0.64705884]
4. [0.909804 0.5372549 0.9215687 ... 0.8705883 0.427451 0.8470589 ]
5. ...
6. [0.7568628 0.54901963 0.23529413 ... 0.28235295 0.52156866 0.5647059 ]
7. [0.909804 0.6862745 0.227451 ... 0.21568629 0.43137258 0.36862746]
8. [0.59607846 0.9450981 0.41176474 ... 0.73333335 0.1764706 0.16862746]]
```

normalize

normalize变换用于对输入图像的归一化，包括三个参数：

- mean：图像每个通道的均值。
- std：图像每个通道的标准差。
- is_hwc：bool值，输入图像的格式。True为(height, width, channel)，False为(channel, height, width)，默认为True。

```
1. normalize = vision.Normalize(mean=(0.1307,), std=(0.3081,))
2. normalized_image = normalize(rescaled_image)
3. print(normalized_image)
```

我们在rescale的基础上进一步将图片进行归一化，经过normalize处理后的图片像素值

```
1. outputs:
2. [[ 1.7395868 -0.29693064 2.3505423 ... 0.60677403 1.2050011
3. 0.7976976 ]
4. [-0.3987565 0.9377082 1.4341093 ... 2.617835 1.8414128
5. 1.6759458 ]
6. [ 2.5287375 1.3195552 2.5669222 ... 2.4014552 0.9631647
7. 2.3250859 ]
8. ...
9. [ 2.0323365 1.3577399 0.33948112 ... 0.49221992 1.2686423
10. 1.4086528 ]
11. [ 2.5287375 1.803228 0.31402466 ... 0.27583995 0.9758929
12. 0.77224106]
13. [ 1.5104787 2.6432917 0.9122518 ... 1.9559668 0.14855757
14. 0.12310111]]
```

HWC2CHW

HWC2CHW变换用于转换图像格式。在不同的硬件设备中可能会对(height，width，channel)或(channel，height，width)两种不同格式有针对性优化。MindSpore设置HWC为默认图像格式，在有CHW格式需求时，可使用该变换进行处理。

这里我们先将前文中normalized_image处理为HWC格式，然后进行转换。可以看到转换前后的shape发生了变化。

```
1. hwc2chw = vision.HWC2CHW()
2. chw_image = hwc2chw(hwc_image)
3. print(hwc_image.shape, chw_image.shape)
4.
5. outputs:
6. (28, 28, 1) (1, 28, 28)
```

batch

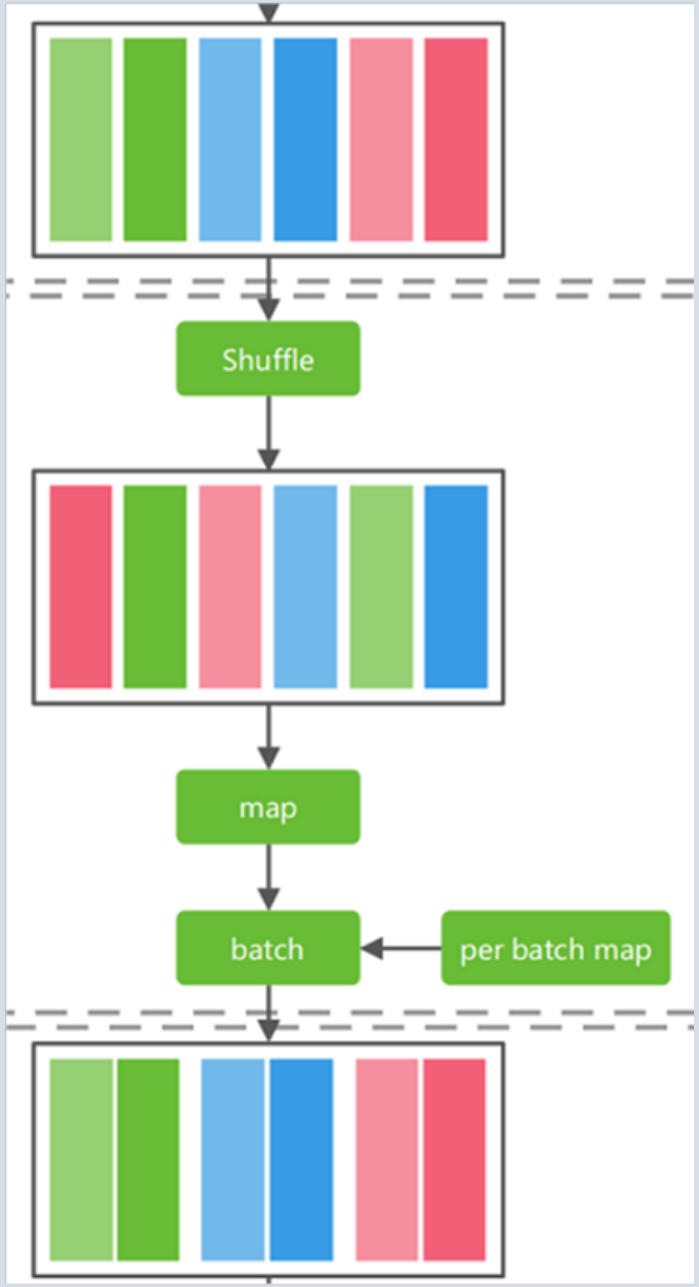
将数据集打包为固定大小的batch是在有限硬件资源下使用梯度下降进行模型优化的折中方法，可以保证梯度下降的随机性和优化计算量。一般我们会设置一个固定的batch size，将连续的数据分为若干批（batch）。

经过预处理后，数据集的图片变成了四维Tensor；打印的这四维分别是batch_size, channels, height, weight

```
1. train_dataset = train_dataset.batch(batch_size=32)
2. for image, label in train_dataset.create_tuple_iterator():
3. print(f"shape of image [N C H W]: {image.shape}")
4. break
5.
6. outputs:
7. shape of image [N C H W]: (32, 28, 28, 1)
```

pipeline

当然，我们也可以把上述独立的操作整合成一个数据预处理的流水线（pipeline）。将数据操作序列写为列表，并通过map指定对应的数据列，过程中将按照列表中的顺序执行数据增强操作，最后通过batch进行批处理。



数据集迭代

完成数据集操作后，我们可以用create_tuple_iterator或create_dict_iterator接口创建数据迭代器，迭代访问数据，然后送入神经网络中进行训练。

- create_dict_iterator返回的是dict形式的数据，字典的key便是之前在get_col_names()时获取到的数据列名称：

```
1. for data in train_dataset.create_dict_iterator():
2. print(data['image'].shape)
3. break
4.
5. outputs:
6. (64, 1, 28, 28)
```

- create_tuple_iterator返回tuple形式的数据：

```
1. image, label =
next(train_dataset.create_tuple_iterator())
2. print(image.shape)
3. print(label.shape)
4.
5. outputs:
6. (64, 1, 28, 28)
7. (64, )
```

神经网络构建---定义模型类

昇思MindSpore通过定义模型类来构建神经网络，在定义模型类时，可以继承nn.Cell类，在__init__中进行实例化神经网络层、设置状态等，在construct方法中书写正向计算逻辑。

```
1. class Network(nn.Cell):
2. def __init__(self):
3. super().__init__()
4. self.flatten = nn.Flatten()
5. self.dense_relu_sequential = nn.SequentialCell(
6. nn.Dense(28*28, 512, weight_init="normal", bias_init="zeros"),
7. nn.ReLU(),
8. nn.Dense(512, 512, weight_init="normal", bias_init="zeros"),
9. nn.ReLU(),
10. nn.Dense(512, 10, weight_init="normal", bias_init="zeros")
11. )
12.
13. def construct(self, x):
14. x = self.flatten(x)
15. logits = self.dense_relu_sequential(x)
16. return logits
17.
18. model = Network()
19. print(model)
```

模型层

在构建网络中使用了以下网络层或框架API，接下来我们将对其进行详解。

1. nn.Flatten
2. nn.Dense
3. nn.ReLU
4. nn.SequentialCell

本节中我们分解上节构造的神经网络模型中的每一层。回顾数据预处理阶段，我们将图片数据处理成了形状为（64， 1， 28， 28）的Tensor，作为模型的输入。

```
1. image, label = next(train_dataset.create_tuple_iterator())
2. print(image.shape)
3.
4. outputs:
5. (64, 1, 28, 28)
```

nn.Flatten

Flatten层的作用是将数据展平，常被放在卷积层和全联接层之间，将卷积层输出的特征图转换为向量序列的形式。

nn.Flatten默认从维度1开始到最后一维进行展平，在本案例中，形状为（64， 1， 28， 28）的图像Tensor要展平的范围是（1， 28， 28），展平后的形状为（64， 1x28x28），也就是（64，784）。

```
1. flatten_layer = nn.Flatten()
2. image_after_flatten = flatten_layer(image)
3. print(image_after_flatten.shape)
4.
5. outputs:
6. (64, 784)
```

nn.Dense

nn.Dense为全连接层，其使用权重和偏差对输入进行线性变换。

在进行实例化时，会要求指定in_channels和out_channels，也就是输入的空间维度和输出的空间维度。在本案例中，展平后的图像Tensor形状为（64， 784），所以in_channels应设置为784。在图中我们可以看到out_channels为512，所以输出的Tensor形状应为（64，512）

```
1. print(f"Shape of image before nn.Dense: {image_after_flatten.shape}")
2. dense_layer = nn.Dense(in_channels=784, out_channels=512)
3. image_after_dense = dense_layer(image_after_flatten)
4. print(f"Shape of image after nn.Dense: {image_after_dense.shape}")
5.
6. outputs:
7. Shape of image before nn.Dense: (64, 784)
8. Shape of image after nn.Dense: (64,512)
```

nn.ReLU

nn.ReLU层给网络中加入非线性的激活函数，帮助神经网络学习各种复杂的特征。

nn.ReLU层将Tensor中大于0的元素保留，将小于0的元素截断为0。

在本案例中，我们可以看到relu之前的图像Tensor中的负值在经过ReLU层后变为了0。

```
print(f"Image before ReLU: \n{image_after_dense.shape[0, :5]}")
relu_layer = nn.ReLU()
image_after_relu = relu_layer(image_after_dense[0, :5])
print(f"Image after ReLU: \n{image_after_relu}")

outputs:
Image before ReLU:
[ 0.11175375 -1.5815861 1.2966743 0.80992675 -0.17070138]
Image after ReLU:
[0.11175375 0. 1.2966743 0.80992675 0. ]
```

nn.SequentialCell

nn.SequentialCell是一个有序的Cell容器。输入Tensor将按照定义的顺序通过所有Cell。我们可以使用nn.SequentialCell将多个网络层快速组合构造一个神经网络模型。

在本案例中，我们将三个Dense和ReLU层交替组合，因手写数字识别任务的分类数为10（数字0-9），故最后一层Dense层的out_channel设为10，输出的形状也为(64, 10)。

```
1. dense_relu_sequential = nn.SequentialCell(
2. nn.Dense(28*28, 512),
3. nn.ReLU(),
4. nn.Dense(512, 512),
5. nn.ReLU(),
6. nn.Dense(512, 10))
7. image_after_sequential = dense_relu_sequential(image_after_flatten)
8. print(f"Shape of image after SequentialCell: {image_after_sequential.shape}")
9.
10. outputs:
11. Shape of image after SequentialCell: (64, 10)
```

nn.Softmax

nn.Softmax将神经网络最后一个全连接层返回的logits的值缩放为[0，1]，表示每个类别的预测概率。axis指定的维度数值和为1。

```
1. softmax = nn.Softmax(axis=-1)
2. pred_probab = softmax(logits)
```

模型参数

模型参数(Parameter)：神经网络中需要训练的参数矩阵或向量，在模型定义时采用随机初始化，通常是nn层内的weight/bias/gamma/beta等属性。

在MindSpore中，我们可以通过get_parameters获取网络的参数。

```
1. model = Network()
2.
3. for param in model.get_parameters():
4. print(param)
5. break
6.
7. outputs:
8. Parameter (name=dense_relu_sequential.0.weight, shape=(512, 784), dtype=Float32, requires_grad=True)
```

name：参数的名称，第一段是参数所在网络层名称，最后一段是参数名称，例子中的是sequentialcell中第一层Dense的权重

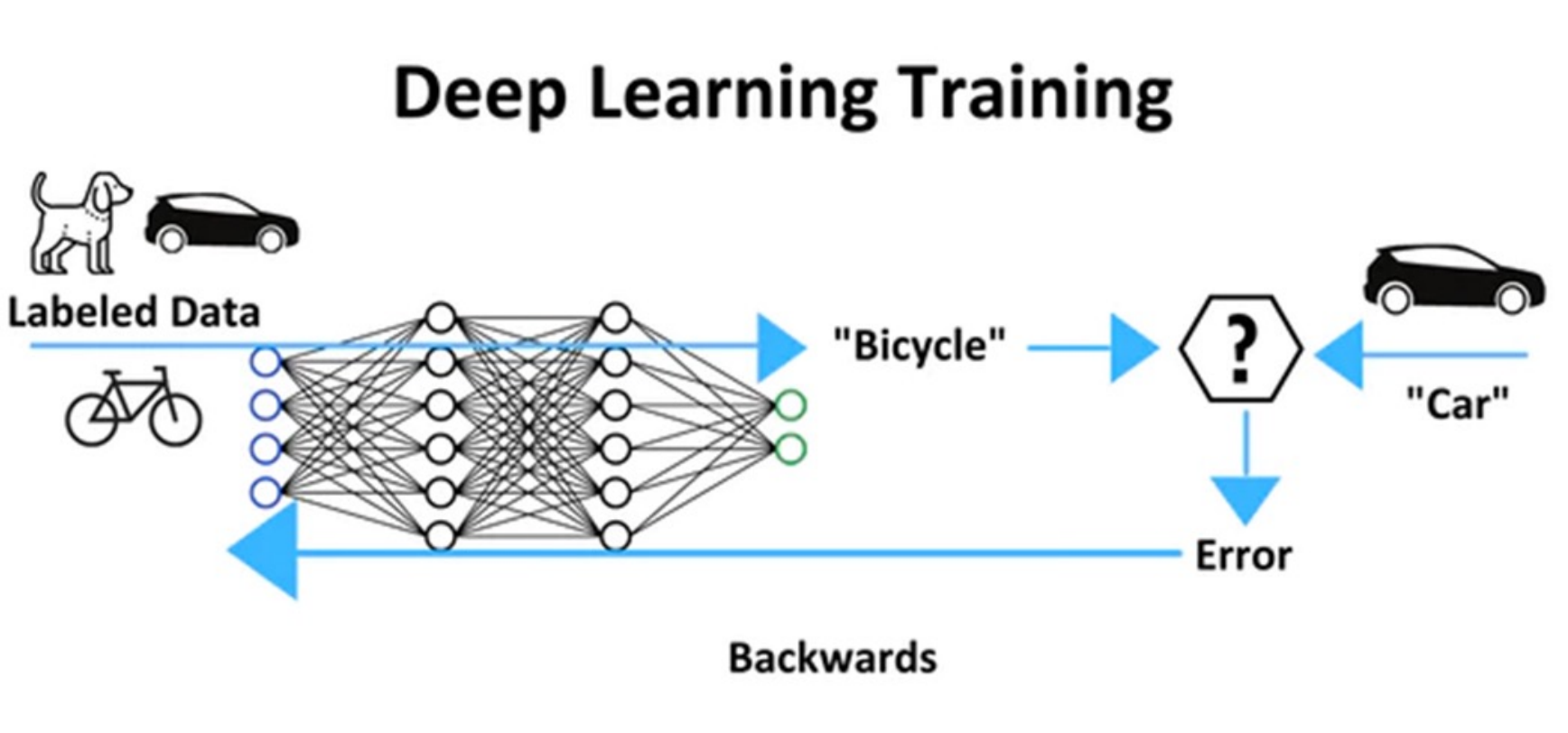
shape：参数的形状

dtype：参数的数据类型

requires_grad：参数是否进行梯度更新，requires_grad=True的参数可以通过model.trainable_params获取

模型训练——定义模型训练逻辑，遍历训练集进行训练

对于框架而言，深度学习整体的训练流程包含以下几个部分，第一个是正向计算得到logits；其次通过损失函数计算正向计算结果logits和正确标签targets之间的误差，也就是loss；然后根据loss进行反向传播，获得整个权重对应的梯度，最后把梯度更新到网络权重上



正向计算

我们直接调用模型，将数据输入模型中，可以获得一个二维的Tensor输出（logits），其包含每个类别的原始预测值。随后通过损失函数（loss_fn）评估模型的预测值与目标值之间的误差。

```
1. # Define forward function
2. def forward_fn(data, label):
3. logits = model(data)
4. loss = loss_fn(logits, label)
5. return loss, logits
```

其中，损失函数（loss function）用于评估模型的预测值（logits）和目标值（targets）之间的误差。训练模型时，随机初始化的神经网络模型开始时会预测出错误的结果。损失函数会评估预测结果与目标值的相异程度，模型训练的目标即为降低损失函数求得的误差。不同的损失函数适用于不同的任务，如手写数字识别任务是多分类问题，适合使用交叉熵损失(CrossEntropyLoss)。mindspore.nn提供了很多开箱即用的损失函数供选择。

```
loss_fn = nn.CrossEntropyLoss()
```

反向传播

为了优化模型参数，需要求参数对loss的导数，此时我们调用mindspore.value_and_grad函数，来获得function的微分函数，并在train_step中调用微分函数，反向传播获得梯度。

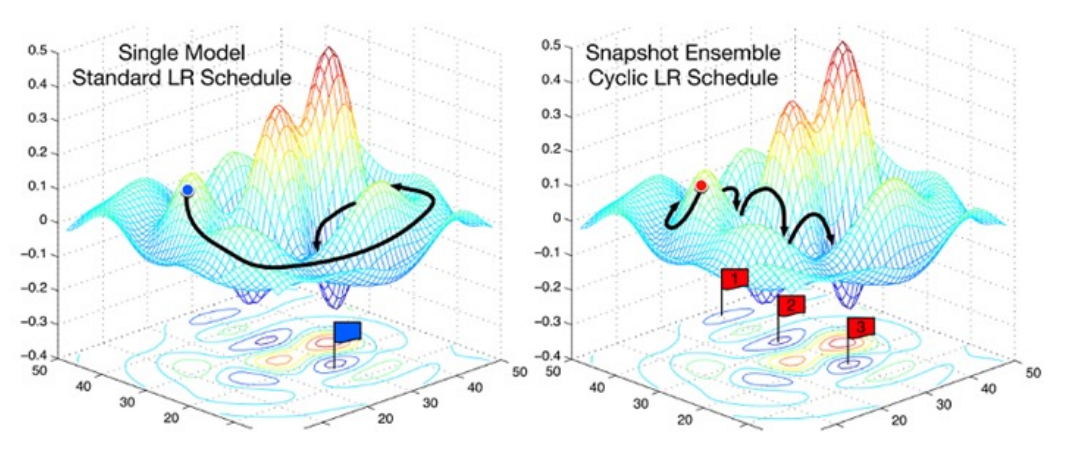
```
# Get gradient function
grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)

# Define function of one-step training
def train_step(data, label):
(loss, _), grads = grad_fn(data, label)
optimizer(grads)
return loss
```

权重更新

在函数返回loss、logits以及grads后，就可以将梯度值grads放入optimizer中进行权重更新，完成一个完整的训练step。权重更新的过程也被称为模型优化（Model Optimization），它是在每个训练步骤中调整模型参数以减少模型误差的过程。

```
1. # Define function of one-step training
2. def train_step(data, label):
3. (loss, _), grads = grad_fn(data, label)
4. optimizer(grads)
5. return loss
```

模型优化是MindSpore提供多种优化算法的实现，称之为优化器（Optimizer）。优化器内部定义了模型的参数优化过程（即梯度如何更新至模型参数），所有优化逻辑都封装在优化器对象中。在这里，我们使用SGD（Stochastic Gradient Descent）优化器。




```
optimizer = nn.SGD(model.trainable_params(),
learning_rate=learning_rate)
```

模型评估——使用训练好的模型，在测试集评估效果

在训练时，一般会进行边训练边评估，每完成一轮训练便基于验证集进行一次评估，目的是在训练中定位到评估结果最好的模型权重。与训练过程类似，评估时也是将验证数据集进行了一次完整的遍历，并将预测结果输入评价指标函数进行计算，如这里是计算了准确率，差别在于在执行评估前，需要将模型的状态设置为非训练(model.set_train(False))。

```
1. def test_loop(model, dataset, loss_fn):
2. num_batches = dataset.get_dataset_size()
3. model.set_train(False)
4. total, test_loss, correct = 0, 0, 0
5. for data, label in dataset.create_tuple_iterator():
6. pred = model(data)
7. total += len(data)
8. test_loss += loss_fn(pred, label).asnumpy()
9. correct += (pred.argmax(1) == label).asnumpy().sum()
10. test_loss /= num_batches
11. correct /= total
12. print(f"Test: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
13.
14. epochs = 3
15. for t in range(epochs):
16. print(f"Epoch {t+1}\n-------------------------------")
17. train(model, train_dataset)
18. test(model, test_dataset, loss_fn)
19. print("Done!")
```

模型保存与加载

在训练网络模型的过程中，实际上我们希望保存中间和最后的结果，用于微调（fine-tune）和后续的模型推理与部署。

MindSpore提供了 save_checkpoint接口，通过传入训练好的model对象进行保存，后续使用模型时使用load_checkpoint接口加载checkpoint文件，并通过load_param_into_net将其加载到构造好的模型中。

模型保存

```
1. model = network()
2. mindspore.save_checkpoint(model, "model.ckpt")
```

```
1. model = network()
2. param_dict = mindspore.load_checkpoint("model.ckpt")
3. param_not_load, _ = mindspore.load_param_into_net(model, param_dict)
4. print(param_not_load)
```

模型加载

模型推理

在完成模型的保存与加载后，下一步便是模型推理，即在加载的已训练模型基础上，利用其学习到的参数和结构，对输入数据进行前向传播计算，从而生成预测结果或推导出目标输出。这一过程是模型应用的关键环节

```
1. model.set_train(False)
2. for data, label in test_dataset:
3. pred = model(data)
4. predicted = pred.argmax(1)
5. print(f'Predicted: "{predicted[:10]}", Actual: "{label[:10]}"')
6. break
7.
8. outputs:
9. Predicted: "[3 9 6 1 6 7 4 5 2 2]",
10. Actual: "[3 9 6 1 6 7 4 5 2 2]"
```

昇思MindSpore的编程范式

在完成模型的保存与加载后，下一步便是模型推理，即在加载的已训练模型基础上，利用其学习到的参数和结构，对输入数据进行前向传播计算，从而生成预测结果或推导出目标输出。这一过程是模型应用的关键环节

```
1. model.set_train(False)
2. for data, label in test_dataset:
3. pred = model(data)
4. predicted = pred.argmax(1)
5. print(f'Predicted: "{predicted[:10]}", Actual: "{label[:10]}"')
6. break
7.
8. outputs:
9. Predicted: "[3 9 6 1 6 7 4 5 2 2]",
10. Actual: "[3 9 6 1 6 7 4 5 2 2]"
```

3.3 昇思MindSpore的编程范式

苏统华
软件学院
哈尔滨工业大学

面向对象(OOP)

面向对象编程（OOP）：现在主流编程语言也都是面向对象这种方式。它的一个主要的使用习惯，同时也是PyTorch构造神经网络的主要使用的习惯，是先构造一个类，然后在使用的时候把类进行实例化，从而获得一个实例化的对象，最后直接去调用这个对象。下图就是一个典型的使用PyTorch书写这样一个神经网络的例子，先在init中实例化nn.Linear全连接层，紧接着实例化整体网络network，把输入送进去，进行对象调用。

```
1. class Network(nn.Module):
2. def __init__(self):
3. super().__init__()
4. self.linear = nn.Linear(10, 20)
5.
6. def forward(self, inputs):
7. return self.linear(inputs)
8.
9. net = Network()
10. outputs = net(torch.randn(2, 10))
```

函数式(FP)

函数式编程（FP）：而另一种是一个叫做函数式编程的写法，函数式主要的思想其实是把我们所谓的数学语义的函数思想迁移到实际编程过程中来，我们会先构造一个函数，将函数进行变换，最后调用变换后的函数。

这里举的是一个基于Jax框架进行网络构建的例子。假设我们去求的也同样是CrossEntropy loss，函数会有两个输入w和b，在反向传播中并不是像PyTorch那样调用loss.backward，而是使用grad接口进行了一次函数变换。函数变换的输入是函数，得到的输出同样也是函数，用于后续调用。并且并不是针对 nn层，或者说面向对象里面的对象的属性去进行求导，而是对它输入的位置进行求导，在该例子中，函数的输入为w和b，函数式编程便针对w和b对应的位置，也就是0和1来进行求导。

```
1. def loss(W, b):
2. preds = predict(W, b, inputs)
3. label_probs = preds * targets + \
4. (1 - preds) * (1 - targets)
5. return -jnp.sum(jnp.log(label_probs))
6.
7. # Initialize random model coefficients
8. key, W_key, b_key = random.split(key, 3)
9. W = random.normal(W_key, (3,))
10. b = random.normal(b_key, ())
11.
12. # Including tuple values
13. W_grad, b_grad = grad(loss, (0, 1))(W, b)
```

MindSpore —— OOP+FP混合编程

MindSpore主打的是一个面向对象与函数式融合编程范式。

1）在构造神经网络时保留PyTorch的使用习惯，也就是进行面向对象编程，通过类来构建神经网络，并且在使用的时候直接实例化神经网络的对象。

2）在训练流程的构造中使用了函数式编程，更加符合数学逻辑。深度学习的正向计算包括将输入送到神经网络中计算得到logits，然后将logits和targets输入损失函数loss function中得到loss值，我们使用函数式编程把以上流程构造为一个正向函数，随后进行函数变换获得一个求梯度的反向函数，执行反向函数就可以得到梯度值并进行如梯度裁剪等相关处理。我们将处理好的梯度送到优化器optimizer中，最后更新到神经网络上面去。那我们也可以直接再把正向函数与反向函数封装成一个表示整体训练过程的函数，在遍历数据集时调用函数进行训练。

OOP

用类构建神经网络，再实例化Network对象

```
1. # Define model
2. class Network(nn.Cell):
3. def __init__(self):
4. super().__init__()
5. self.faltten = nn.Flatten()
6. self.dense_relu_sequential = nn.SequentialCell(
7. nn.Dense(28*28, 512),
8. nn.ReLU(),
9. nn.Dense(512, 512),
10. nn.ReLU(),
11. nn.Dense(512, 10)
12. )
13.
14. def construct(self, x):
15. x = self.flatten(x)
16. logits = self.dense_relu_sequential(x)
17. return logits
18.
19. model = Network()
20. print(model)
```

FP

Network+Loss直接构造正向函数，通过函数变换，获得梯度计算（反向传播）函数，然后构造训练过程函数，并调用函数进行训练

```
# 1. Define forward function
def forward_fn(data, label):
logits = model(data)
loss = loss_fn(logits, label)
return loss, logits

# 2. Get gradient function
grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)

# 3. Define function of one-step training
def train_step(data, label):
(loss, _), grads = grad_fn(data, label)
optimizer(grads)
return loss
```

3.4 昇思MindSpore的动静统一

苏统华
软件学院
哈尔滨工业大学

动态图

动态图：其核心特点是计算图的构建和计算同时发生（Define by run）

- 原理： 类似Python解释器，在计算图中定义一个Tensor时，其值就已经被计算且确定了。
- 优点： Pythonic语法，在调试模型时较为方便，能够实时得到中间结果的值。
- 缺点： 由于所有节点都需要被保存，导致难以对整个计算图进行优化。

现在大家使用的框架可能大都以动态图为主，动态图的一个核心特点就是计算图的构建和计算是同时发生的，叫做define by run。它的写法其实就是现在所有主流框架的统一写法，类似于Python解释器，在计算图中定义一个Tenso的时候，其数据就已经被确定了，它的优势在于Pythonic语法，调试的时候也比较方便，可以打上断点进行单部的调试，能够实时地看到中间的结果；但它缺点是因为每一次都是执行完了之后获得结果，在这种情况下所有节点都需要被保存，很难去进行整个的编译优化，性能会相对差一些。

静态图

静态图：将计算图的构建和实际计算分开（Define and run）

- 原理： 在构建阶段，根据完整的计算流程对原始的计算图进行优化和调整， 编译得到更省内存和计算量更少的计算图。编译之后图的结构不再改变，所以称之为“静态图”。 在计算阶段，根据输入数据执行编译好的计算图得到计算结果。
- 优点： 静态图相比起动态图，对全局的信息掌握更丰富，可做的优化也会更多。
- 缺点： 中间过程对于用户来说是个黑盒，无法像动态图一样实时拿到中间计算结果。

静态图以TensorFlow的思路为首，从编译优化的角度考虑进行设计。静态图的构建和实际计算是分开的，先把神经网络写好了之后，再对神经网络进行编译，编译后图的结构不再更改，如此得到一整个大的计算图，根据输入数据执行编译好的计算图。它的优点是因为可以编译优化，所以静态图的性能相比动态图会更好一些。但缺点是灵活性较差，由于整图编译，没有办法进行单步调试，无法获取中间计算结果。

运行时切换

context模式是一种全局的设置模式，在构建网络之前进行配置。

```
1. import numpy as np
2. import mindspore as ms
3. from mindspore import nn, Tensor
4. ms.set_context(mode=ms.GRAPH_MODE) # 使用set_context进行运行静态图模式的配置
5.
6. class Network(nn.Cell):
7. def __init__(self):
8. super().__init__()
9. self.faltten = nn.Flatten()
10. self.dense_relu_sequential = nn.SequentialCell(
11. nn.Dense(28*28, 512),
12. nn.ReLU(),
13. nn.Dense(512, 512),
14. nn.ReLU(),
15. nn.Dense(512, 10)
16. )
17.
18. def construct(self, x):
19. x = self.flatten(x)
20. logits = self.dense_relu_sequential(x)
21. return logits
22.
23. model = Network()
24. input = Tensor(np.ones([64, 1, 28, 28]).astype(np.float32))
25. output = model(input)
26. print(output)
```

即时编译(Just In Time, JIT)

结合动静态图的优缺点，为了保留动态图的灵活度，同时又可以利用静态图的性能优势，MindSpore采用了即时编译的方式。这里就需要把train_step构造为一个function，如果想使用动态图的话，直接执行function即可；如果想要使用静态图或是进行编译优化，便打上mindspore.jit这样的修饰器，通过一行代码去切换动静态图。

```
1. def train_step(data, label):
2. loss, grads = grad_fn(data, label)
3. optimizer(grads)
4. return loss
5.
6. @ms.jit
7. def train_step(data, label):
8. loss, grads = grad_fn(data, label)
9. optimizer(grads)
10. return loss
```

局部静态加速

Cell.construct添加ms.jit修饰器：当我们需要为神经网络的某部分进行加速时，可以直接在construct方法上使用jit修饰器，在调用实例化对象时，该模块自动被编译为静态图。

```
1. class Network(nn.Cell):
2. def __init__(self):
3. super().__init__()
4. self.faltten = nn.Flatten()
5. self.dense_relu_sequential = nn.SequentialCell(
6. nn.Dense(28*28, 512),
7. nn.ReLU(),
8. nn.Dense(512, 512),
9. nn.ReLU(),
10. nn.Dense(512, 10)
11. )
12.
13. @ms.jit # 使用ms.jit装饰器，使被装饰的函数以静态图模式运行
14. def construct(self, x):
15. x = self.flatten(x)
16. logits = self.dense_relu_sequential(x)
17. return logits
```

3.5 昇思MindSpore的高阶操作

苏统华
软件学院
哈尔滨工业大学

函数式自动微分

为了优化模型参数，需要求参数对loss的导数：∂loss/∂w 和 ∂loss/∂b ，此时我们调用mindspore.value_and_grad函数，来获得function的微分函数。这里使用了grad函数的两个入参，分别为：

- fn：待求导的函数。
- grad_position：指定求导输入位置的索引。

由于使用Cell封装神经网络模型，模型参数为Cell的内部属性，此时我们不需要使用grad_position指定对函数输入求导，因此将其配置为None。对模型参数求导时，我们使用weights参数，使用model.trainable_params()方法从Cell中取出可以求导的参数。

```
1. # Define model
2. class Network(nn.Cell):
3. def __init__(self):
4. super().__init__()
5. self.w = w
6. self.b = b
7.
8. def construct(self, x):
9. z = ops.matmul(x, self.w) + self.b
10. return z
11.
12. # Instantiate model
13. model = Network()
14. # Instantiate loss function
15. loss_fn = nn.BCEWithLogitsLoss()
16.
17. # Define forward function
18. def forward_fn(x, y):
19. z = model(x)
20. loss = loss_fn(z, y)
21. return loss
22.
23. grad_fn = mindspore.value_and_grad(forward_fn, \
24. None, weights=model.trainable_params())
25. loss, grads = grad_fn(x, y)
26. print(grads)
```

高阶梯度

这种函数式的编程方式，对于高阶梯度，尤其是大家如果做GAN这一类涉及gradient penalty的模型，写法体现会比较易懂。大家可以看到下面求高阶微分的时候，数学公式上的表达是可以和实际代码的求导一一对应的。使用一次ops.grad得到求一阶导的对应公式，在此基础上再使用一次ops.grad得到求二阶导的对应公式。

```
1. def f(x):
2. return ops.sin(x) + ops.cos(x)
3.
4.
5. f_grad = ops.grad(f)
6.
7. f_grad_grad = ops.grad(f_grad)
```

梯度裁剪

反向传播得到梯度后，我们也可以针对梯度进行一些额外的操作，如梯度裁剪等。MindSpore对于梯度裁剪同样提供了相应的接口，大家可以直接根据Tensor值的大小进行裁剪（调用clip_by_value接口），或者是做L2 Norm进行裁剪（对应clip_by_global_norm接口）。以上大家常用的两种梯度裁剪的方式，通过这种方法得到裁剪后的梯度后将其送入优化器中。整体写法上看起来相对简单，没有那么复杂。

常见的梯度裁剪有两种：

a) 确定一个范围，如果参数的gradient超过了，直接裁剪；

b) 根据若干个参数的gradient组成的vector的L2 Norm进行裁剪。

```
1. def train_step(data, label):
2. (loss, _), grads = grad_fn(data, label)
3. grads = ops,clip_by_value(grads, 1.0)
4. optimizer(grads)
5. return loss
```