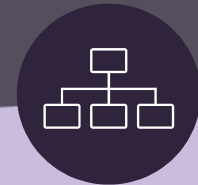
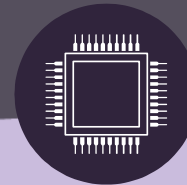


# 딥러닝 V

경남대학교 창의융합대학



담당교수 : 유 현 주

comjoo@uok.ac.kr



# Deep Learning



## Review Practice

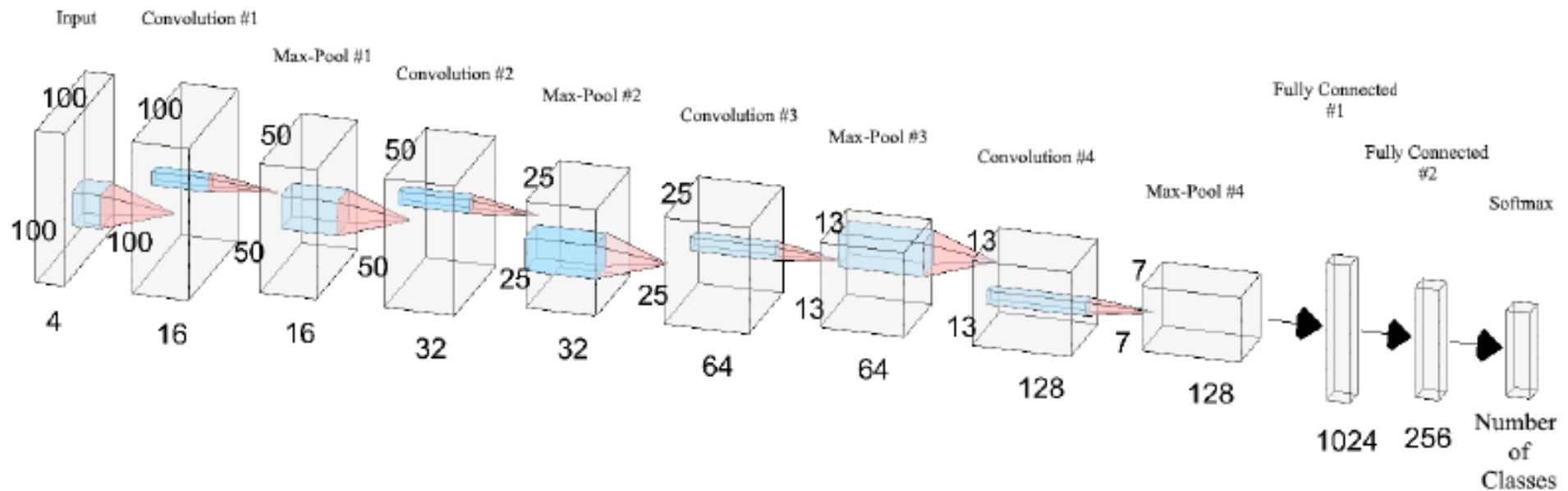
- Image Classification using CNN





# 예제 1: CNN Fruit image Classification

- 캐글 사이트에서 제공되는 데이터 셋으로 예제 살펴보기
- Fruits 360 데이터셋: 과일과 채소가 포함된 이미지 데이터셋
  - 131 종류의 과일과 채소에 대한 90380개의 이미지가 있는 데이터 세트
- CNN 이미지 분류 모델



- 과일 이미지 분류하기

# 예제 1: CNN Fruit image Classification



- 과일 이미지 분류하기: Train data set / Test data set

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import pathlib
from zipfile import ZipFile
import PIL
import tensorflow as tf
from tensorflow import keras
from keras_preprocessing.image import ImageDataGenerator

train_dir = pathlib.Path("../input/fruits/fruits-360_dataset/fruits-360/Training")
test_dir = pathlib.Path("../input/fruits/fruits-360_dataset/fruits-360/Test")

image_count = len(list(train_dir.glob('*/*.jpg')))
image_count
```

# 예제 1: CNN Fruit image Classification



- Visualization

```
fruits = list(train_dir.glob("Apple Pink Lady/*.jpg"))

plt.figure(figsize=(10, 10))

for i in range(9):
    plt.subplot(3, 3, i + 1)
    img = PIL.Image.open(str(fruits[i]))
    plt.imshow(img)
    plt.axis('off')

plt.show()
```



# 예제 1: CNN Fruit image Classification



```
batch_size = 128  
img_height = 100  
img_width = 100
```

```
train_data = tf.keras.preprocessing.image_dataset_from_directory(  
    train_dir,  
    validation_split=0.2,  
    subset='training',  
    seed=42,  
    image_size=(img_height, img_width),  
    batch_size=batch_size  
)
```

# 예제 1: CNN Fruit image Classification



```
val_data = tf.keras.preprocessing.image_dataset_from_directory(  
    train_dir,  
    validation_split=0.2,  
    subset='validation',  
    seed=42,  
    image_size=(img_height, img_width),  
    batch_size=batch_size  
)
```

```
class_names = train_data.class_names  
num_classes = len(class_names)
```

# 예제 1: CNN Fruit image Classification



```
datagen = ImageDataGenerator(rescale = 1./255)
train_generator = datagen.flow_from_directory(
    train_dir,
    target_size=(img_height,img_width),
    batch_size = 128,
    class_mode='categorical'
)

test_generator = datagen.flow_from_directory(
    test_dir,
    target_size=(img_height,img_width),
    batch_size = 128,
    class_mode='categorical'
)
```

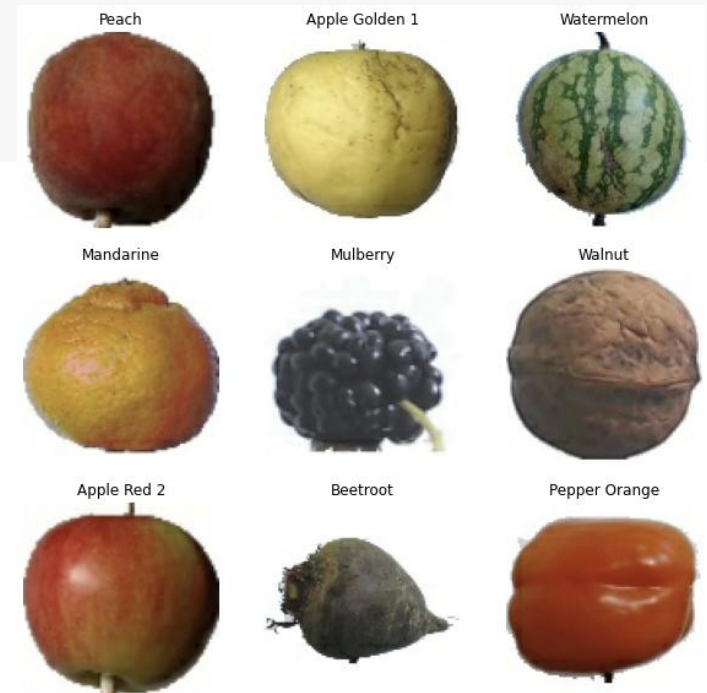


# 예제 1: CNN Fruit image Classification



```
plt.figure(figsize=(10, 10))

#to take 1 batch
for images, labels in train_data.take(1):
    for i in range(9):
        plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype('uint8'))
        plt.title(class_names[labels[i]])
        plt.axis('off')
```



# 예제 1: CNN Fruit image Classification



- Modeling using CNN

```
model = tf.keras.Sequential([
    keras.Input(shape=(img_height, img_width, 3)),

    keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(32, 3, padding='same', activation='relu'),
    keras.layers.MaxPooling2D(),
    keras.layers.Conv2D(64, 3, padding='same', activation='relu'),
    keras.layers.MaxPooling2D(2),
    keras.layers.Dropout(0.5),

    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(num_classes, activation='softmax')
])

model.summary()
```

# 예제 1: CNN Fruit image Classification

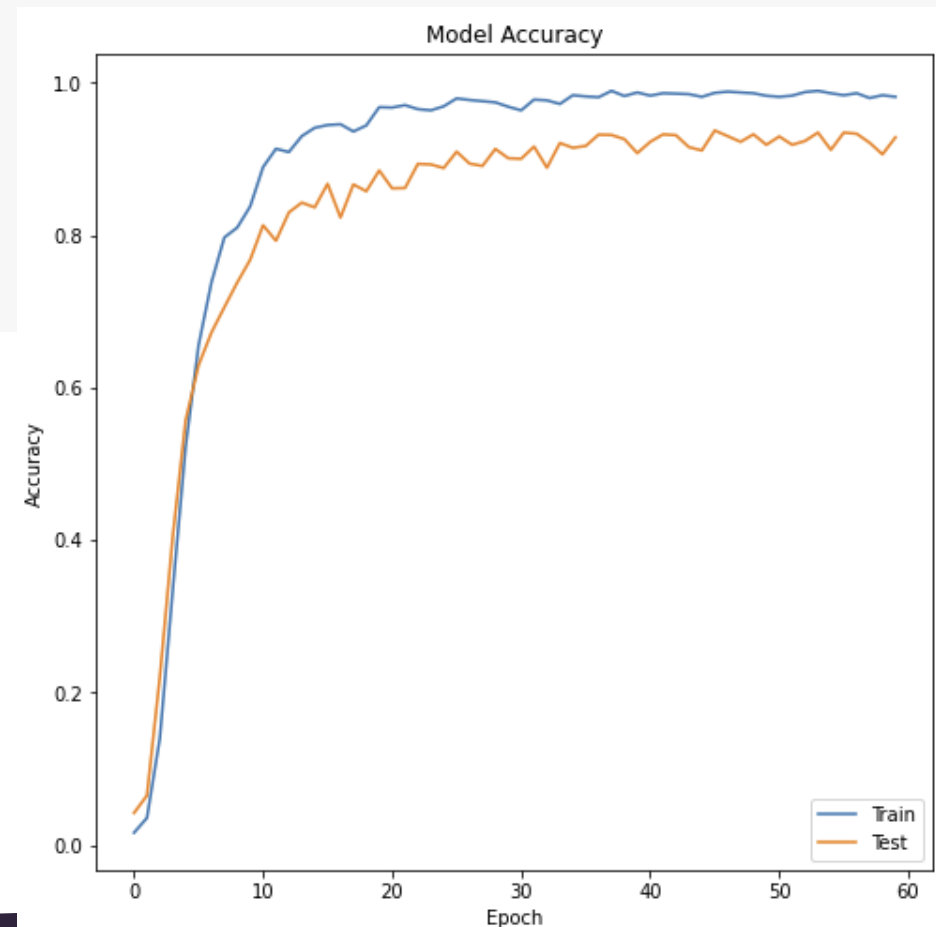


```
model.compile(optimizer='adam',  
              loss='categorical_crossentropy',  
              metrics=['accuracy'])  
  
history = model.fit(  
    train_generator,  
    steps_per_epoch=15,  
    validation_steps=20,  
    validation_data=test_generator,  
    epochs=60,  
    verbose=1  
)
```

# 예제 1: CNN Fruit image Classification



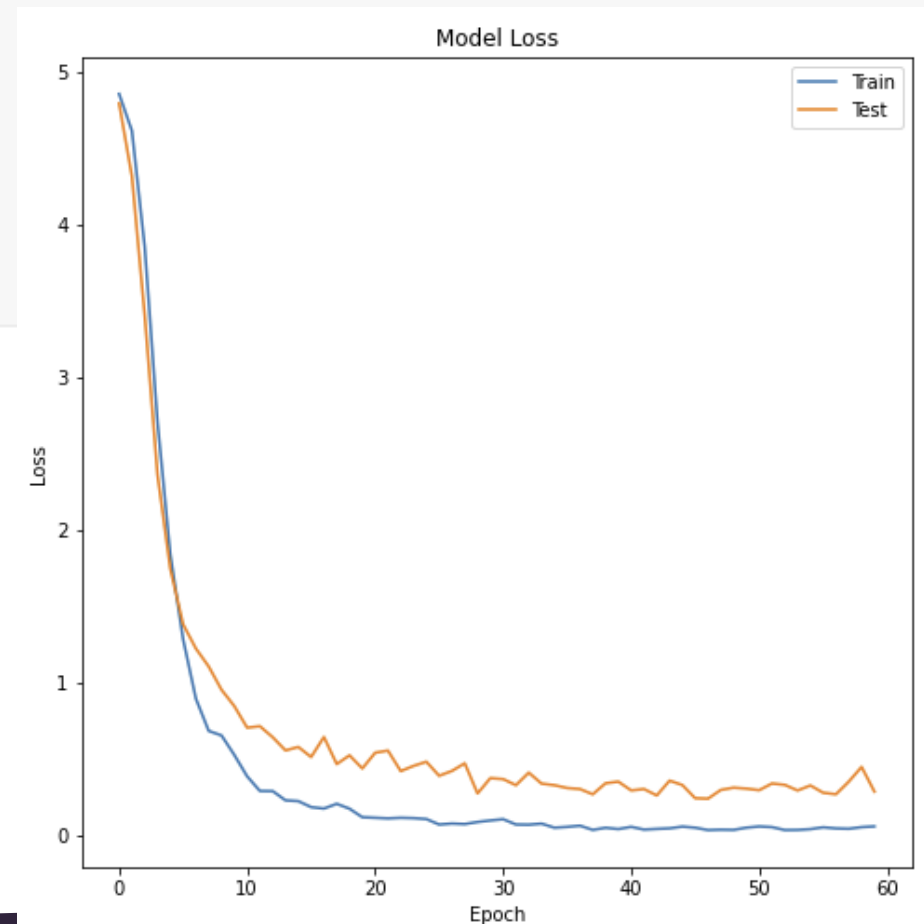
```
# Summarize history for accuracy
plt.figure(figsize=(8,8))
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='lower right')
plt.show()
```



# 예제 1: CNN Fruit image Classification



```
# Summarize history for loss
plt.figure(figsize=(8,8))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper right')
plt.show()
```



# 예제 1: CNN Fruit image Classification



- Test the Model Using the test data

```
loss, acc = model.evaluate(test_generator)
```

```
print('Loss:', loss)
```

```
print('Accuracy:', acc)
```

```
178/178 [=====] - 34s 193ms/step - loss: 0.2842 - accuracy: 0.9290
```

```
Loss: 0.28417617082595825
```

```
Accuracy: 0.928993284702301
```



## 예제2: 카카오 열매 질병 검출

### ● Import Packages

```
import math
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import os
import pandas as pd
import pathlib
import PIL
import PIL.Image
import plotly.graph_objects as go
import seaborn as sns
import shutil
import tensorflow as tf
import tensorflow_hub as hub
import warnings

from distutils.dir_util import copy_tree
from IPython.display import Image
from random import sample
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay

warnings.filterwarnings('ignore')
```



## 예제2: 카카오 열매 질병 검출

- Data Exploration (데이터 탐색)

```
# Place in a variable the dataset's path.
```

```
dataset_path = "../input/cacao-diseases/cacao_diseases/cacao_photos"
```

```
# Display the quantity of each class.
```

```
black_pod_count = os.listdir(dataset_path+"/black_pod_rot")
```

```
print(f"Black Pod Rot : {len(black_pod_count)}")
```

```
healthy_count = os.listdir(dataset_path+"/healthy")
```

```
print(f"Healthy : {len(healthy_count)}")
```

```
pod_borer_count = os.listdir(dataset_path+"/pod_borer")
```

```
print(f"Pod Borer : {len(pod_borer_count)}")
```





## 예제2: 카카오 열매 질병 검출

- Data Exploration (데이터 탐색)

```
# Create various directories for Model Building.
```

```
os.mkdir("train_photos")  
os.mkdir("val_photos")  
os.mkdir("augment_photos")  
os.mkdir("best_models")
```

```
# Copy and place the dataset into the /kaggle/working/ folder.
```

```
copy_tree(dataset_path, "./train_photos")
```

## 예제2: 카카오 열매 질병 검출



```
# Place in a variable the dataset's path in working directory.
```

```
image_path = "./train_photos"
```

```
# Create validation folder for each classes.
```

```
os.mkdir("val_photos/black_pod_rot")
```

```
os.mkdir("val_photos/healthy")
```

```
os.mkdir("val_photos/pod_borer")
```

```
# Create a function that moves a specific number of files from lake_path to processed_path.
```

```
def move_num_of_files(count, lake_path, processed_path, count_end):
```

```
    try:
```

```
        images = os.listdir(lake_path)
```

```
        for file in images:
```

```
            if count < count_end:
```

```
                path = f"{lake_path}/{file}"
```

```
                shutil.move(path, processed_path)
```

```
                count += 1
```

```
    except Exception as e:
```

```
        print(e)
```



## 예제2: 카카오 열매 질병 검출

### ● Creation of Validation Set

```
# Move 20% each class into val_photos directory.

black_pod_20 = math.ceil(len(black_pod_count)*0.20)
healthy_20 = math.ceil(len(healthy_count)*0.20)
pod_borer_20 = math.ceil(len(pod_borer_count)*0.20)

move_num_of_files(0, image_path+"/black_pod_rot", "./val_photos/black_pod_rot",black_pod_20)
move_num_of_files(0, image_path+"/healthy", "./val_photos/healthy",healthy_20)
move_num_of_files(0, image_path+"/pod_borer", "./val_photos/pod_borer",pod_borer_20)
```

### ● Data Cleaning

```
# Reduce healthy class randomly.

black_pod_current_count = os.listdir(image_path+"/black_pod_rot")
healthy_current_count = os.listdir(image_path+"/healthy")
pod_borer_current_count = os.listdir(image_path+"/pod_borer")

healthy_path = image_path + "/healthy"
healthy = os.listdir(healthy_path)
for file in sample(healthy,(len(healthy_current_count)-len(black_pod_current_count))):
    os.remove(healthy_path+"/"+file)
```



## 예제2: 카카오 열매 질병 검출

- Data Augmentation

```
import Augmentor

# Define augmentation pipelines.

pod_borer_augmentation_pipeline = Augmentor.Pipeline(source_directory=image_path+"/pod_borer", output_directory=image_path+"/pod_borer")

# Define different augmentations depending on the pipeline.

pod_borer_augmentation_pipeline.rotate(probability=0.6, max_left_rotation=10, max_right_rotation=10)
pod_borer_augmentation_pipeline.skew_top_bottom(0.3, 0.7)
pod_borer_augmentation_pipeline.skew_left_right(0.3, 0.7)
pod_borer_augmentation_pipeline.flip_random(0.3)

# Augment pod borer class.

pod_borer_augmentation_pipeline.sample(len(black_pod_current_count)-len(pod_borer_current_count))
```



# 예제2: 카카오 열매 질병 검출

## ● Data Preparation

```
# Place hyperparameters in variables for model building.
```

```
batch_size = 1  
img_height = 224  
img_width = 224  
class_mode = "sparse"  
epochs = 30
```

```
# Clears the directory containing the best model.
```

```
shutil.rmtree('./best_models')  
os.mkdir("best_models")
```

```
datagen = tf.keras.preprocessing.image.ImageDataGenerator()  
train_path = "./train_photos"  
val_path = "./val_photos"  
valgen = tf.keras.preprocessing.image.ImageDataGenerator()
```



## 예제2: 카카오 열매 질병 검출

### ● Model Building (모델 설계)

```
# For this dataset we will be imitating google's CNN layer called EfficientNet.

model = tf.keras.applications.EfficientNetB0(include_top=False,
                                             input_shape=train_generator.image_shape)

model.trainable = False
for layer in model.layers:
    if not isinstance(layer, tf.keras.layers.BatchNormalization):
        layer.trainable = True

# Adding 2 fully-connected layers.

x = model.output
x = tf.keras.layers.GlobalAveragePooling2D()(x)
x = tf.keras.layers.Dropout(0.2)(x)
predictions = tf.keras.layers.Dense(3, activation="softmax")(x)
working_model = tf.keras.Model(inputs = model.input, outputs = predictions)
working_model.compile(loss=tf.losses.SparseCategoricalCrossentropy(),
                     optimizer=tf.keras.optimizers.Adam(0.0001),
                     metrics=[tf.metrics.SparseCategoricalAccuracy()])
```



## 예제2: 카카오 열매 질병 검출

- Model Building (모델 설계)

```
# Defining callbacks.

early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                                    mode='min',
                                                    patience=4)

checkpoint_filepath = './best_models/lowest_val_loss.h5'
checkpoint = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_filepath,
                                                  verbose=1,
                                                  monitor='val_loss',
                                                  save_best_only=True,
                                                  mode='min')

# Display Model Configurations.

working_model.summary()
```



## 예제2: 카카오 열매 질병 검출

- Model Training (모델 훈련)

```
model_history = working_model.fit(train_generator,  
                                  epochs=epochs,  
                                  callbacks=[early_stopping,  
                                             checkpoint],  
                                  validation_data=val_generator)
```

- Model Validation and Exploration (모델 검증 및 탐색)

```
# Load the best model produced in model training.  
  
best_model = tf.keras.models.load_model('./best_models/lowest_val_loss.h5')
```



# 예제2: 카카오 열매 질병 검출



## ● Model's Accuracy (모델 정확도)

```
# Define scatter plot visualization on plotly express.
```

```
def display_training_curves(training, validation, yaxis):
```

```
    if yaxis == "loss":
```

```
        ylabel = "Loss"
```

```
        title = "Loss with respect to Epochs"
```

```
    else:
```

```
        ylabel = "Accuracy"
```

```
        title = "Accuracy with respect to Epochs"
```

```
    fig = go.Figure()
```

```
    fig.add_trace(
```

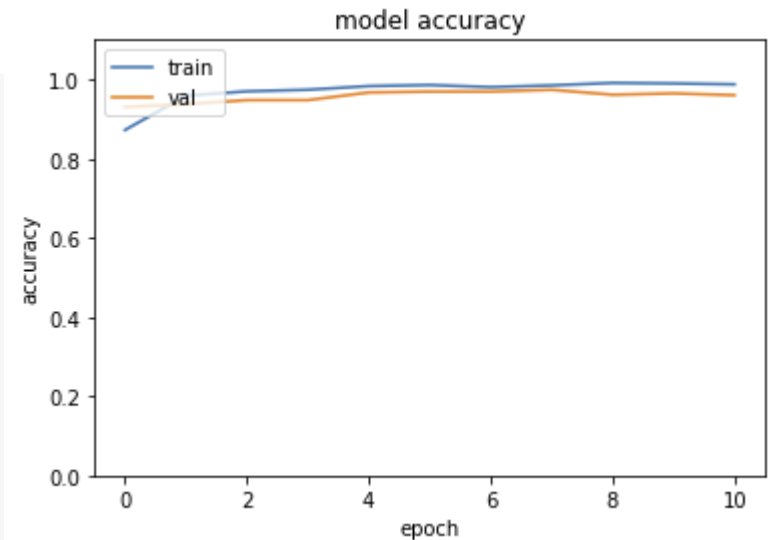
```
        go.Scatter(x=np.arange(1, epochs+1), mode='lines+markers', y=training, marker=dict(color="dodgerblue"),
                    name="Training"))
```

```
    fig.add_trace(
```

```
        go.Scatter(x=np.arange(1, epochs+1), mode='lines+markers', y=validation, marker=dict(color="darkorange"),
                    name="Validation"))
```

```
    fig.update_layout(title_text=title, yaxis_title=ylabel, xaxis_title="Epochs", template="plotly_white")
```

```
    fig.show()
```





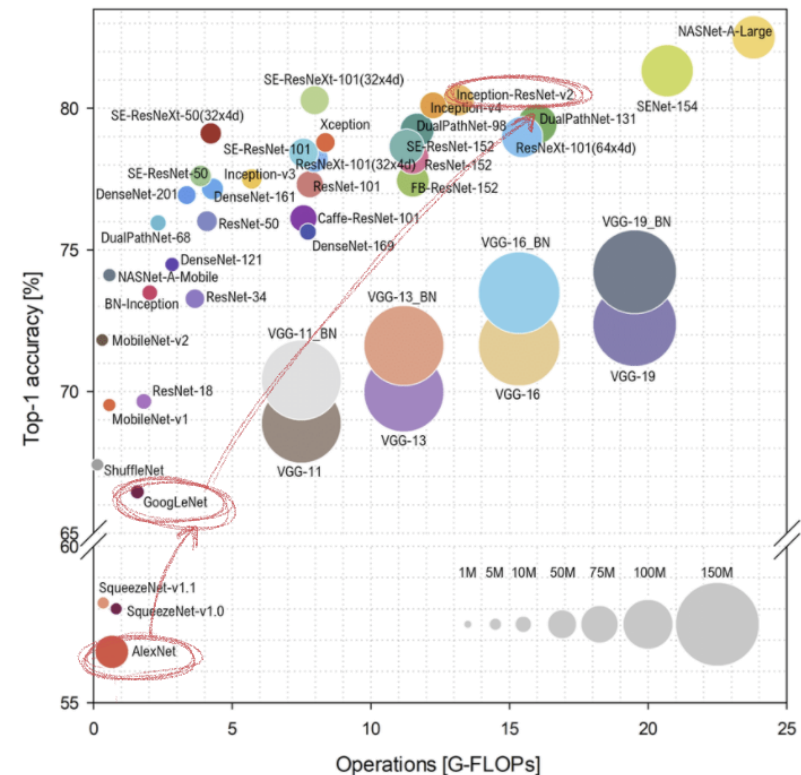
# 대표적인 컨볼루션 신경망(CNN) 모델

르넷-5	얀 르쿤의 1998년 우편물에 필기체로 쓰인 우편번호 인식하는 최초의 CNN 모델
알렉스넷	<p>1,400만개의 이미지로 구성된 방대한 이미지 데이터 베이스로 2만여개의 카테고리를 포함</p> <p>ILSVRC대회 중 2012 대회에서 객체인식오류율을 25.8%에서 16.4%로 낮추어 1위를 차지</p>
제트에프넷	<p>ILSVRC 2013 대회에서 상위 5개 클래스 기준 오류율을 16.4%에서 11.7%로 낮춤</p> <p>알렉스넷의 문제점을 분석하고 개선함</p>
브이지지넷	<p>ILSVRC 2014 대회에서 2위 수상, '작은 필터를 사용하는 대신 깊은 신경망을 만들자'는 아이디어로 설계</p> <p>성능은 상위 5개 클래스 기준으로 오류율 7.3%, 모델 구조가 단순하여 널리 사용됨</p>
구글넷	<p>ILSVRC 2014 대회에서 1위 수상 모델, 성능은 상위 5개 클래스 기준으로 오류율 6.7% 달성</p> <p>전형적인 CNN 구조를 탈피하여 네트워크 속 네트워크 구조로 설계</p>
레즈넷	<p>ILSVRC 2015와 COCO 대회의 분류 및 탐지 모든 부문에서 우승</p> <p>상위 5개 클래스 기준으로 오류율 3.57% 달성 처음으로 인간의 평균 인지 능력인 5.1 오류율을 추월함</p>

# CNN 모델 비교



- 알렉스 넷은 초기 모델로서 모델은 크고 연산량과 정확도는 낮은 편
- VGG Net은 모델이 크고 연산량도 많지만 구조가 단순하다는 장점이 있음
- 구글넷은 모델도 작고 연산량도 적은 효율적인 모델
- 레즈넷은 모델 크기와 연산량은 중간정도이나 정확도가 매우 높음
- 성능 개선 모델이 계속 등장
  - 프랙탈넷
  - 댄스넷
  - 나스넷



Benchmark Analysis of Representative Deep Neural Network Architectures



# 최적의 프레임워크를 선택

- TensorFlow를 선택해야 하는 경우:
  - 빠른 프로토타이핑과 배포가 중요한 프로젝트
  - 모바일/웹/엣지 환경 배포가 필요한 경우
  - 팀의 머신러닝 경험이 제한적인 경우
  - 프로덕션 안정성이 최우선인 엔터프라이즈 환경
  - MLOps 파이프라인 구축이 중요한 프로젝트
- PyTorch를 선택해야 하는 경우:
  - 연구 및 실험이 주목적인 프로젝트
  - 복잡하고 새로운 아키텍처 개발이 필요한 경우
  - 디버깅과 모델 해석이 중요한 작업
  - NLP 연구 및 최신 Transformer 모델 활용
  - 학습 과정의 세밀한 제어가 필요한 경우

# Contents

---



- 1장: 케라스를 사용한 인공 신경망 소개
- 2장: 심층 신경망 훈련
- 3장: 텐서플로를 사용한 사용자 정의 모델과 훈련
- 4장: 텐서플로를 사용한 데이터 적재와 전처리
- 5장: 합성곱 신경망을 사용한 컴퓨터 비전
- 6장: RNN과 CNN을 사용한 시퀀스 처리
- 7장: RNN과 어텐션을 사용한 자연어 처리
- 8장: 오토인코더, GAN 그리고 확산 모델
- 9장: 강화 학습
- 10장: 대규모 텐서플로 모델 훈련과 배포



## 대규모 텐서플로 모델 훈련과 배포

- 텐서플로 모델 서빙
- 모바일 또는 임베디드 디바이스에 모델 배포하기
- 웹 페이지에서 모델 실행하기
- 계산 속도를 높이기 위해 GPU 사용하기
- 다중 장치에서 모델 훈련하기

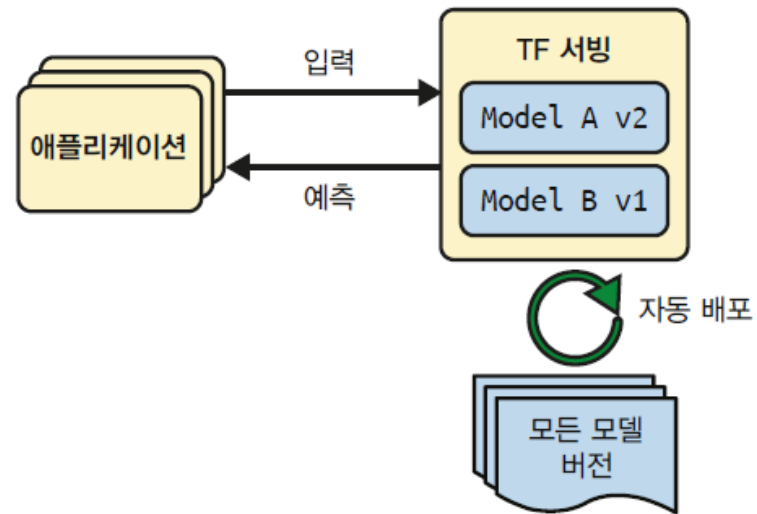




# 텐서플로 모델 서빙(1)

## 텐서플로 서빙 사용하기

- TF 서빙은 C++로 작성되었고 많은 테스트를 거친 매우 효율적인 모델 서버



TF 서빙은 여러 모델을 서비스하고 자동으로 최신 버전을 배포할 수 있음



# 텐서플로 모델 서빙(2)

## SavedModel로 내보내기

- 모델을 저장하는 방법은 `model.save()`를 호출
  - 모델의 버전을 관리하려면 모델 버전마다 서브디렉토리를 만들어 줌

---

```
from pathlib import Path
import tensorflow as tf
X_train, X_valid, X_test = [...] # MNIST 데이터셋을 로드하고 분할합니다.
model = [...] # MNIST 모델을 만들고 훈련합니다(이미지 전처리도 수행합니다).

model_name = "my_mnist_model"
model_version = "0001"
model_path = Path(model_name) / model_version
model.save(model_path, save_format="tf")
```

---

- SavedModel을 검사할 수 있는 `saved_model_cli` 명령줄 인터페이스

---

```
$ saved_model_cli show --dir my_mnist_model/0001
The given SavedModel contains the following tag-sets:
'serve'
```

---



# 텐서플로 모델 서빙(3)



- save() 메서드를 사용하여 케라스 모델을 저장할 때 "serve"로 태그된 단일 메타그래프가 저장

---

```
$ saved_model_cli show --dir my_mnist_model/0001 --tag_set serve
The given SavedModel MetaGraphDef contains SignatureDefs with these keys:
SignatureDef key: "__saved_model_init_op"
SignatureDef key: "serving_default"
```

---

- 초기화 함수 "\_\_saved\_model\_init\_op"와 서빙 함수 "serving\_default"

---

```
$ saved_model_cli show --dir my_mnist_model/0001 --tag_set serve \
    --signature_def serving_default
The given SavedModel SignatureDef contains the following input(s):
  inputs['flatten_input'] tensor_info:
    dtype: DT_UINT8
    shape: (-1, 28, 28)
    name: serving_default_flatten_input:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['dense_1'] tensor_info:
    dtype: DT_FLOAT
    shape: (-1, 10)
    name: StatefulPartitionedCall:0
Method name is: tensorflow/serving/predict
```

---



# 텐서플로 모델 서빙(4)

## 텐서플로 서빙 설치하고 시작하기

- 우분투 apt 패키지 매니저를 사용

---

```
url = "https://storage.googleapis.com/tensorflow-serving-apt"
src = "stable tensorflow-model-server tensorflow-model-server-universal"
!echo 'deb {url} {src}' > /etc/apt/sources.list.d/tensorflow-serving.list
!curl '{url}/tensorflow-serving.release.pub.gpg' | apt-key add -
!apt update -q && apt-get install -y tensorflow-model-server
%pip install -q -U tensorflow-serving-api==2.11.1
```

---

# 텐서플로 모델 서빙(5)



- 서버를 시작

- 이 명령에는 모델의 기본 디렉터리의 절대 경로(0001이 아닌 my\_mnist\_model의 경로)가 필요하므로 MODEL\_DIR 환경 변수에 저장

```
import os

os.environ["MODEL_DIR"] = str(model_path.parent.absolute())
```

- 서버를 실행

```
%%bash --bg
tensorflow_model_server \
  --port=8500 \
  --rest_api_port=8501 \
  --model_name=my_mnist_model \
  --model_base_path="${MODEL_DIR}" >my_server.log 2>&1
```



# 텐서플로 모델 서빙(6)

## REST API로 TF 서빙에 쿼리하기

- 쿼리 만들기
  - 호출할 함수 시그니처의 이름과 입력 데이터가 포함
  - JSON 포맷으로 요청해야 하므로 넘파이 배열인 입력 이미지를 파이썬 리스트로 변경

---

```
import json

X_new = X_test[:3] # 새로운 숫자 이미지 3개를 분류한다고 가정합니다.
request_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

---

- JSON 포맷은 100% 텍스트이므로 request\_json은 다음과 같은 문자열로 출력

---

```
>>> request_json
'{"signature_name": "serving_default", "instances": [[[0, 0, 0, 0, ... ]]]}'
```

---



# 텐서플로 모델 서빙(7)

- 요청 데이터를 HTTP POST 메서드로 TF 서빙에 전송
  - requests 라이브러리를 사용

---

```
import requests

server_url = "http://localhost:8501/v1/models/my_mnist_model:predict"
response = requests.post(server_url, data=request_json)
response.raise_for_status() # 에러가 생길 경우 예외를 발생시킵니다.
response = response.json()
```

---

- 응답은 "predictions" 키 하나를 가진 딕셔너리

---

```
>>> import numpy as np
>>> y_proba = np.array(response["predictions"])
>>> y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.97, 0.01, 0. , 0. , 0. , 0. , 0.01, 0. , 0. ]])
```

---



# 텐서플로 모델 서빙(8)

## gRPC API로 TF 서빙에 쿼리하기

- gRPC API는 직렬화된 PredictRequest 프로토콜 버퍼를 입력으로 기대하고 직렬화된 PredictResponse 프로토콜 버퍼를 출력
  - 요청(request) 만들기

---

```
from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0] # == "flatten_input"
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

---



# 텐서플로 모델 서빙(9)

- 서버로 이 요청을 보내고 응답을 받음
  - grpcio 라이브러리가 필요

---

```
import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)
response = predict_service.Predict(request, timeout=10.0)
```

---

- PredictResponse 프로토콜 버퍼를 텐서로 바꿈

---

```
output_name = model.output_names[0] # == "dense_1"
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

---



# 텐서플로 모델 서빙(10)

## 새로운 버전의 모델 배포하기

- 새로운 버전의 모델을 만들어 SavedModel 포맷으로 내보내기
  - my\_mnist\_model/0002 디렉터리에 저장

---

```
model = [...] # MNIST 모델의 새로운 버전을 만들고 훈련합니다.  
model_version = "0002"  
model_path = Path(model_name) / model_version  
model.save(model_path, save_format="tf")
```

---





# 텐서플로 모델 서빙(11)

- 새로운 요청은 새 버전에서 처리
  - 대기 중인 요청이 모두 응답을 받으면 바로 이전 버전의 모델은 내려감
  - TF 서빙 로그에서 이 작업을 확인

---

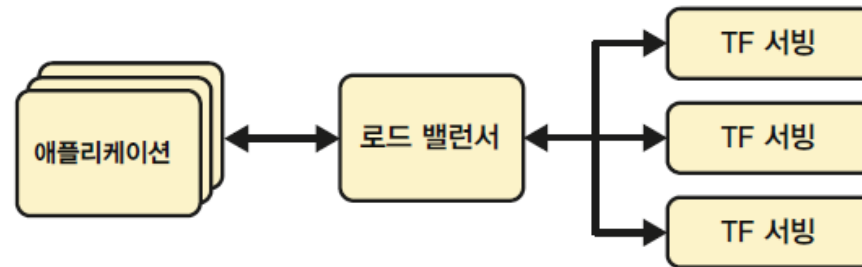
```
[...]
Reading SavedModel from: /models/my_mnist_model/0002
Reading meta graph with tags { serve }
[...]
Successfully loaded servable version {name: my_mnist_model version: 2}
Quiescing servable version {name: my_mnist_model version: 1}
Done quiescing servable version {name: my_mnist_model version: 1}
Unloading servable version {name: my_mnist_model version: 1}
```

---



# 텐서플로 모델 서빙(12)

- 초당 쿼리 요청이 많을 것으로 예상된다면 TF 서빙을 서버 여러 대에 설치하고 쿼리를 로드 밸런싱.
- 이렇게 하려면 많은 TF 서빙 컨테이너를 서버에 배포하고 관리해야 함
- 쿠버네티스(<https://kubernetes.io>) 같은 도구를 사용



로드 밸런싱을 사용한 TF 서빙의 규모 확장

# 텐서플로 모델 서빙(13)



## 버텍스 AI에서 예측 서비스 만들기

- 버텍스 AI는 다양한 AI 관련 도구와 서비스를 제공하는 구글 클라우드 플랫폼(GCP) 내의 한 플랫폼
  - 데이터셋을 업로드하고, 레이블을 부여하고, 자주 사용하는 특성을 피처 스토어(feature store)에 저장하여 학습 또는 제품 환경에서 사용
- 시작하기 전 설정
  1. 구글 계정으로 로그인한 다음 구글 클라우드 플랫폼 콘솔(console)로 이동
  2. GCP를 처음 사용한다면 서비스 약관을 읽고 동의
  3. 전에 GCP를 사용한 적이 있고 무료 체험 기간이 끝났다면 서비스로 인해 약간의 비용이 청구됨
  4. GCP의 모든 자원(가상 서버, 저장 파일, 실행된 훈련)은 하나의 프로젝트에 속함
    - GCP 계정이 생성될 때 자동으로 'My First Project'라는 프로젝트가 만들어짐
  5. GCP 계정과 프로젝트가 있고 결제 정보가 활성화되었으므로 필요한 API를 활성화

# 텐서플로 모델 서빙(14)



📦 무료 평가판을 사용할 수 있습니다. 지금 활성화하여 Google Cloud 제품에 쓸 수 있는 \$300의 크레딧을 받아 보세요. [자세히 알아보기](#)

Google Cloud Platform 프로젝트 선택 🔍




- 홈
- Marketplace
- 결제
- API API 및 서비스 >
- 지원 >
- IAM 및 관리자 >
- 시작하기
- 보안 >
- 컴퓨팅
  - App Engine >
  - Compute Engine >
  - Kubernetes Engine >

## Google Cloud Platform 시작하기

12개월 동안 사용할 수 있는 \$300 상당의 무료 체험판으로 시작해 보세요.  
무료 체험판이 종료되어도 '항상 무료' 제품으로 서비스를 계속 사용할 수 있습니다.

**무료로 사용해 보기**

### 인기 제품

Compute Engine	Cloud Storage	Cloud SQL
 확장 가능한 고성능 가상 머신	 강력하고 간편하며 경제적인 객체 저장소 서비스	 완전 관리형 MySQL 또는 PostgreSQL 데이터베이스 서비스

구글 클라우드 플랫폼 콘솔



# 텐서플로 모델 서빙(15)

- GCP 서비스를 사용하기 전에 인증
  - 코랩을 사용할 때 가장 간단한 방법은 다음 코드를 실행

---

```
from google.colab import auth
```

```
auth.authenticate_user()
```

---



# 텐서플로 모델 서빙(16)

- SavedModel을 저장할 구글 클라우드 스토리지(GCS) 버킷 becket (데이터를 저장하는 컨테이너) 만들기
  - 코랩에 미리 설치되어 있는 google-cloud-storage 라이브러리를 사용
  - 먼저 GCS와의 인터페이스 역할을 할 Client 객체를 생성한 다음 이를 사용하여 버킷을 생성

---

```
from google.cloud import storage
```

```
project_id = "my_project" # 이를 프로젝트 ID로 바꾸세요.  
bucket_name = "my_bucket" # 이를 고유한 버킷 이름으로 바꾸세요.  
location = "us-central1"
```

```
storage_client = storage.Client(project=project_id)  
bucket = storage_client.create_bucket(bucket_name, location=location)
```

---



# 텐서플로 모델 서빙(17)

- my\_mnist\_model 디렉터리를 새 버킷에 업로드

---

```
def upload_directory(bucket, dirpath):  
    dirpath = Path(dirpath)  
    for filepath in dirpath.glob("**/*"):  
        if filepath.is_file():  
            blob = bucket.blob(filepath.relative_to(dirpath.parent).as_posix())  
            blob.upload_from_filename(filepath)  
  
upload_directory(bucket, "my_mnist_model")
```

---

- 멀티스레딩을 통해 속도를 크게 높임
  - 구글 클라우드 CLI를 사용하는 경우에는 다음 명령을 사용

---

```
!gsutil -m cp -r my_mnist_model gs://{bucket_name}/
```

---



# 텐서플로 모델 서빙(18)

- 버텍스 AI와 통신하기 위해 googlecloud-aiplatform 라이브러리를 사용

---

```
from google.cloud import aiplatform

server_image = "gcr.io/cloud-aiplatform/prediction/tf2-gpu.2-8:latest"

aiplatform.init(project=project_id, location=location)
mnist_model = aiplatform.Model.upload(
    display_name="mnist",
    artifact_uri=f"gs://{bucket_name}/my_mnist_model/0001",
    serving_container_image_uri=server_image,
)
```

---





# 텐서플로 모델 서빙(19)

- 모델을 배포하고 gRPC 또는 REST API로 쿼리하여 예측을 수행
  - 엔드포인트는 클라이언트 애플리케이션이 서비스에 액세스하려고 할 때 연결하는 곳
  - 엔드포인트에 모델을 배포

---

```
endpoint = aiplatform.Endpoint.create(display_name="mnist-endpoint")

endpoint.deploy(
    mnist_model,
    min_replica_count=1,
    max_replica_count=5,
    machine_type="n1-standard-4",
    accelerator_type="NVIDIA_TESLA_K80",
    accelerator_count=1
)
```

---

- 예측 서비스를 사용

---

```
response = endpoint.predict(instances=X_new.tolist())
```

---



# 텐서플로 모델 서빙(20)

- 분류하려는 이미지를 먼저 파이썬 리스트로 변환
  - 응답 객체에는 예측값이 포함되어 있으며, 이는 실수로 채워진 파이썬 리스트의 리스트로 표현
  - 소수점 이하 두 자리로 반올림하여 넘파이 배열로 변환

---

```
>>> import numpy as np
>>> np.round(response.predictions, 2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.97, 0.01, 0. , 0. , 0. , 0. , 0.01, 0. , 0. ]])
```

---

- 엔드포인트 사용이 끝나면 비용이 지불되지 않도록 엔드포인트를 삭제

---

```
endpoint.undeploy_all() # 엔드포인트에서 모든 모델을 회수합니다.
endpoint.delete()
```

---



# 텐서플로 모델 서빙(21)

## 버텍스 AI에서 배치 예측 작업 실행하기

- 많은 수의 예측을 수행해야 하는 경우 예측 서비스를 반복적으로 호출하는 대신 버텍스 AI에 예측 작업을 실행하도록 요청
  - 여기에는 엔드포인트가 필요하지 않고 모델만 있으면 됨
  - 한 줄당 하나의 샘플을 JSON 값으로 표현한 파일(이 포맷을 JSON Lines라고 함)을 만든 다음 이 파일을 버텍스 AI에 전달
  - 새 디렉터리에 JSON Lines 파일을 생성하고 이 디렉터리를 GCS에 업로드

---

```
batch_path = Path("my_mnist_batch")
batch_path.mkdir(exist_ok=True)
with open(batch_path / "my_mnist_batch.jsonl", "w") as jsonl_file:
    for image in X_test[:100].tolist():
        jsonl_file.write(json.dumps(image))
        jsonl_file.write("\n")

upload_directory(bucket, batch_path)
```

---



# 텐서플로 모델 서빙(22)

- 작업 이름, 사용할 머신 및 가속기의 유형과 개수, 방금 생성한 JSON Lines 파일의 GCS 경로, 버텍스 AI가 모델의 예측을 저장할 GCS 디렉터리 경로를 지정

```
batch_prediction_job = mnist_model.batch_predict(  
    job_display_name="my_batch_prediction_job",  
    machine_type="n1-standard-4",  
    starting_replica_count=1,  
    max_replica_count=5,  
    accelerator_type="NVIDIA_TESLA_K80",  
    accelerator_count=1,  
    gcs_source=[f"gs://{bucket_name}/{batch_path.name}/my_mnist_batch.jsonl"],  
    gcs_destination_prefix=f"gs://{bucket_name}/my_mnist_predictions/",  
    sync=True # 기다리지 않으려면 False로 지정하세요.  
)
```

# 텐서플로 모델 서빙(23)



- 모든 출력 파일은 `batch_prediction_job.iter_outputs()`을 사용하여 반복할 수 있으므로 모든 예측을 읽어서 `y_probas` 배열에 저장

---

```
y_probas = []
for blob in batch_prediction_job.iter_outputs():
    if "prediction.results" in blob.name:
        for line in blob.download_as_text().splitlines():
            y_proba = json.loads(line)["prediction"]
            y_probas.append(y_proba)
```

---

- 예측 확인

---

```
>>> y_pred = np.argmax(y_probas, axis=1)
>>> accuracy = np.sum(y_pred == y_test[:100]) / 100
0.98
```

---



# 텐서플로 모델 서빙(24)

- 모델 사용을 마쳤으면 `mnist_model.delete()`를 실행하여 원하는 경우 삭제
  - GCS 버킷에서 생성한 디렉터리, (비어 있다면) 선택적으로 버킷 자체, 배치 예측 작업도 삭제 가능

---

```
for prefix in ["my_mnist_model/", "my_mnist_batch/", "my_mnist_predictions/"]:
    blobs = bucket.list_blobs(prefix=prefix)
    for blob in blobs:
        blob.delete()

bucket.delete() # bucket이 비어있다면 삭제
batch_prediction_job.delete()
```

---



# 모바일 또는 임베디드 디바이스에 모델 배포하기(1)

- 엣지 컴퓨팅(edge computing)
  - 머신러닝 모델은 여러 개의 GPU를 갖춘 대규모 중앙 서버에서만 실행되는 것이 아니라 모바일 또는 임베디드 디바이스 같이 데이터 소스에 더 가까운 곳에서도 실행될 수 있음
  - 정확도를 크게 낮추지 않으면서도 효율적이고 경량인 모델이 필요
    - 다운로드 시간과 RAM 사용량을 줄이기 위해 모델 크기를 줄임
    - 응답 속도, 배터리 사용량, 발열을 줄이기 위해 예측에 필요한 계산량을 줄임
    - 모델을 특정 디바이스의 제약 조건에 맞춤



# 모바일 또는 임베디드 디바이스에 모델 배포하기(2)

- 모델 크기를 줄이기 위해 TFLite의 모델 변환기는 SavedModel을 받아 FlatBuffers(<https://google.github.io/flatbuffers>) 기반의 경량 포맷으로 압축
  - 다음 코드는 SavedModel을 FlatBuffers로 변환하여 .tflite 파일로 저장

```
converter = tf.lite.TFLiteConverter.from_saved_model(str(model_path))
tflite_model = converter.convert()
with open("my_converted_savedmodel.tflite", "wb") as f:
    f.write(tflite_model)
```

- 변환기는 크기와 지연 시간을 줄이기 위해 모델 최적화도 수행
  - 예측에 필요하지 않은 모든 연산(예 훈련 연산)을 삭제하고 가능한 연산을 최적화

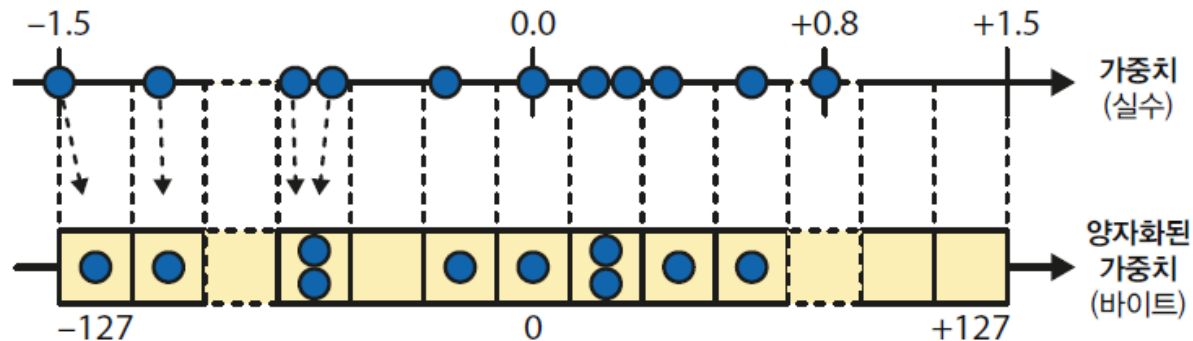




# 모바일 또는 임베디드 디바이스에 모델 배포하기(3)

- TFLite 변환기는 거기에 더해 모델의 가중치를 고정 소수점인 8비트 정수로 압축
  - 32 비트 실수와 비교하면 크기가 4배나 줄어듦
  - 가장 간단한 방법은 훈련 후 양자화(post-training quantization)
    - 훈련 후 양자화를 수행하려면 `convert()` 메서드를 호출하기 전에 `DEFAULT`를 변환기 최적화 리스트에 추가

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```



대칭 양자화를 사용한 32비트 부동소수를 8비트 정수로 변환하기



# 모바일 또는 임베디드 디바이스에 모델 배포하기(4)

- 양자화의 주요 문제는 정확도를 약간 희생하는 것
  - 가중치와 활성화 값에 잡음을 추가하는 것과 같음
  - 정확도 손실이 너무 크면 양자화를 고려한 훈련(quantization-aware training)이 필요



# 웹 페이지에서 모델 실행하기(1)

- 머신러닝 모델을 서버 측이 아닌 클라이언트 측인 사용자 브라우저에서 실행하는 것은 다음과 같은 여러 시나리오에서 유용
  - 사용자의 연결이 간헐적이거나 느린 상황에서 웹 애플리케이션을 자주 사용할 때( 예 등산객을 위한 웹 사이트)는 클라이언트 측에서 직접 모델을 실행하는 것이 웹 사이트를 안정적으로 만드는 유일한 방법
  - 모델의 응답이 가능한 한 빨라야 하는 경우( 예 온라인 게임) 예측을 위해 서버에 쿼리할 필요가 없다면 대기 시간이 확실히 줄어들고 웹 사이트의 응답 속도가 훨씬 빨라짐
  - 웹 서비스에서 일부 비공개 사용자 데이터를 기반으로 예측을 수행할 때 클라이언트 측에서 예측을 수행하여 비공개 데이터가 사용자의 컴퓨터 밖으로 나가지 않게 함으로써 사용자의 개인 정보를 보호할 수 있음
- 이러한 모든 시나리오에 대해 TensorFlow.js (TFJS) 자바스크립트 라이브러리 (<https://tensorflow.org/js>)를 사용할 수 있음



## 웹 페이지에서 모델 실행하기(2)

- TFJS 라이브러리를 임포트하고 사전 훈련된 MobileNet 모델을 다운로드하고 이 모델을 사용하여 이미지를 분류하고 예측을 로그에 기록하는 자바스크립트 모듈

---

```
import "https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest";
import "https://cdn.jsdelivr.net/npm/@tensorflow-models/mobilenet@1.0.0";

const image = document.getElementById("image");

mobilenet.load().then(model => {
  model.classify(image).then(predictions => {
    for (var i = 0; i < predictions.length; i++) {
      let className = predictions[i].className
      let proba = (predictions[i].probability * 100).toFixed(1)
      console.log(className + " : " + proba + "%");
    }
  });
});
```

---



## 웹 페이지에서 모델 실행하기(3)

- TFJS는 웹 브라우저에서 직접 모델을 훈련하는 기능도 지원
  - 실제로 꽤 빠름
  - 컴퓨터에 GPU 카드가 있다면 일반적으로 Nvidia 카드가 아니더라도 TFJS를 사용할 수 있음
  - TFJS는 WebGL을 사용할 수 있는 경우 이를 사용
  - 최신 웹 브라우저는 일반적으로 광범위한 GPU 카드를 지원하므로 실제로 TFJS는 일반 (Nvidia 카드만 지원하는) 텐서플로보다 더 많은 GPU 카드를 지원
  - 사용자의 웹 브라우저에서 모델을 학습시키는 것은 사용자의 데이터를 비공개로 유지하는 데 특히 유용
  - 모델을 중앙에서 학습한 다음 해당 사용자의 데이터를 기반으로 브라우저에서 로컬로 미세 튜닝할 수 있음
  - 연합 학습(federated learning)



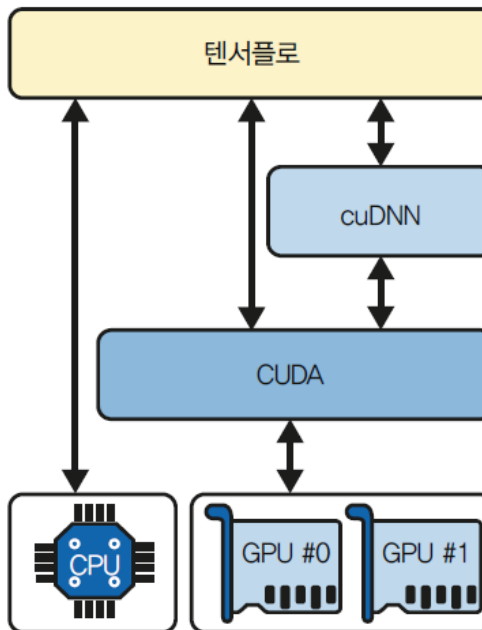


# 계산 속도를 높이기 위해 GPU 사용하기(2)

## GPU 구매하기

- GPU 카드 선택의 고려 사항

- 작업에 필요한 RAM 용량( 예 이미지 처리 또는 NLP의 경우 일반적으로 최소 10GB), 대역폭(GPU에서 데이터를 주고받을 수 있는 속도), 코어 수, 냉각 시스템 등



텐서플로는 CUDA와 cuDNN을 사용해 GPU를 제어하고 DNN 속도를 높임



# 계산 속도를 높이기 위해 GPU 사용하기(3)

- GPU 카드와 필요한 드라이버와 라이브러리를 설치한 후에 nvidia-smi 명령을 사용해 모든 것이 적절히 설치되었는지 확인

```
$ nvidia-smi
Sun Apr 10 04:52:10 2022

+-----+
| NVIDIA-SMI 460.32.03      Driver Version: 460.32.03      CUDA Version: 11.2      |
+-----+-----+
| GPU   Name           Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan   Temp   Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|                                           MIG M.         |
+-----+-----+
|  0   Tesla T4               Off   | 00000000:00:04:0 Off  |            0         |
| N/A    34C    P8          9W / 70W |  3MiB / 15109MiB |      0%      Default  |
|                                           N/A              |
+-----+-----+

+-----+
| Processes:                                |
| GPU   GI    CI          PID Type   Process name                  GPU Memory |
|      ID    ID                                   |             Usage   |
+-----+-----+
|  No running processes found              |
+-----+
```



# 계산 속도를 높이기 위해 **GPU** 사용하기(4)



- 텐서플로가 GPU를 잘 인식하는지 확인하려면 다음 명령을 실행하여 결과값이 비어 있지 않은 지 확인

---

```
>>> physical_gpus = tf.config.list_physical_devices("GPU")
>>> physical_gpus
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

---

# 계산 속도를 높이기 위해 GPU 사용하기(5)



## GPU RAM 관리하기

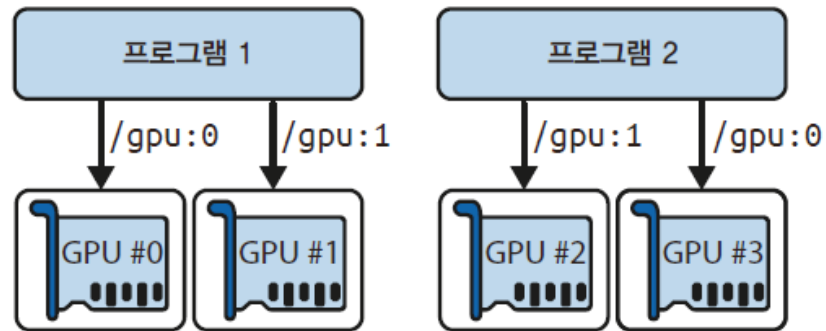
- 기본적으로 텐서플로는 처음 계산을 실행할 때 자동으로 가능한 GPU의 거의 모든 RAM을 확보
  - GPU RAM의 단편화를 막기 위함
  - 두 번째 텐서플로 프로그램(또는 GPU를 사용하는 다른 프로그램)을 시작하면 금방 RAM 부족 현상이 나타남
  - [방법 1] 컴퓨터에 GPU 카드가 여러 개 있다면 각 GPU를 하나의 프로세스에 할당하는 것이 간단한 해결책
    - CUDA\_VISIBLE\_DEVICES 환경 변수를 설정하여 각 프로세스가 해당되는 GPU 카드만 보게 설정
    - CUDA\_DEVICE\_ORDER 환경 변수를 PCI\_BUS\_ID로 설정하여 각 ID가 항상 동일한 GPU 카드를 참조

---

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py  
# 다른 터미널에서  
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

---

# 계산 속도를 높이기 위해 **GPU** 사용하기(6)



각 프로그램은 **GPU**를 두 개 사용



# 계산 속도를 높이기 위해 GPU 사용하기(7)

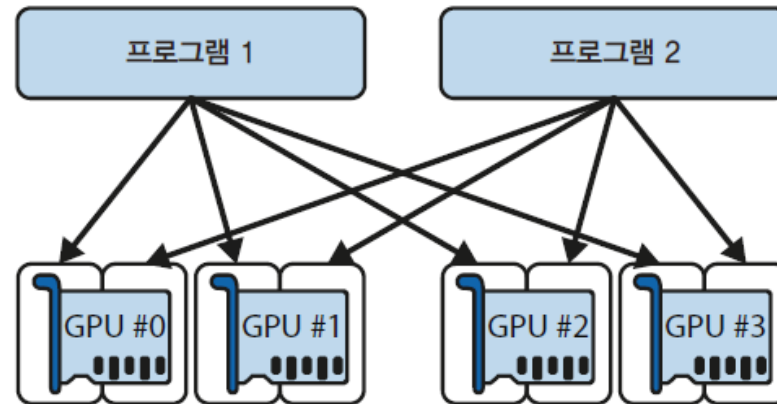
- [방법 2] 텐서플로가 특정한 양의 GPU RAM만 점유하도록 만드는 것
  - 이는 반드시 텐서플로를 임포트한 직후에 수행되어야 함
  - 예를 들어 텐서플로가 GPU마다 2GiB RAM만 점유하게 만들려면 물리적인 GPU 장치에 대한 가상 GPU 장치 (또는 논리적인 GPU 장치)를 만들어야 함  
그다음 가상 GPU 장치의 메모리 한도를 2GiB(즉, 2,048MiB)로 설정

---

```
for gpu in physical_gpus:
    tf.config.set_logical_device_configuration(
        gpu,
        [tf.config.LogicalDeviceConfiguration(memory_limit=2048)]
    )
```

---

# 계산 속도를 높이기 위해 **GPU** 사용하기(8)



각 프로그램이 GPU 네 개를 모두 사용  
하지만 GPU마다 2GiB RAM만 쓸 수 있음



# 계산 속도를 높이기 위해 GPU 사용하기(9)

- [방법 3] 텐서플로가 필요할 때만 메모리를 점유하게 만드는 방법
  - 텐서플로를 임포트한 직후에 설정

---

```
for gpu in physical_gpus:  
    tf.config.experimental.set_memory_growth(gpu, True)
```

---

- [방법 4] TF\_FORCE\_GPU\_ALLOW\_GROWTH 환경 변수를 true로 설정
  - 텐서플로는 프로그램이 종료되기 전까지는 한번 점유한 메모리를 다시 해제하지 않음

# 계산 속도를 높이기 위해 **GPU** 사용하기(10)



- [방법 5] GPU를 두 개 이상의 논리적 장치로 나누기
  - 다음 코드는 GPU #0을 2GiB RAM을 가진 논리적 장치 두 개로 나눔

---

```
tf.config.set_logical_device_configuration(  
    physical_gpus[0],  
    [tf.config.LogicalDeviceConfiguration(memory_limit=2048),  
     tf.config.LogicalDeviceConfiguration(memory_limit=2048)]  
)
```

---

- 논리적 장치를 모두 확인하는 코드

---

```
>>> logical_gpus = tf.config.list_logical_devices("GPU")  
>>> logical_gpus  
[LogicalDevice(name='/device:GPU:0', device_type='GPU'),  
 LogicalDevice(name='/device:GPU:1', device_type='GPU')]
```

---



# 계산 속도를 높이기 위해 GPU 사용하기(11)

## 디바이스에 연산과 변수 할당하기

- 케라스와 tf.data는 일반적으로 연산과 변수를 적절히 잘 배치함
- 하지만 상세하게 제어하고 싶다면 수동으로 각 디바이스에 연산과 변수를 배치할 수도 있음
  - 일반적으로 데이터 전처리를 CPU에 배치하고 신경망 연산은 GPU에 배치
  - GPU는 보통 통신 대역폭에 제약이 많습니다. 따라서 GPU 입출력으로 불필요한 데이터 전송을 피하는 것이 중요
  - CPU RAM을 머신에 추가하는 것은 간단하고 저렴해서 일반적으로 넉넉하게 추가해 사용





# 계산 속도를 높이기 위해 GPU 사용하기(12)

- 기본적으로 GPU 커널이 없는 경우를 제외하고 모든 변수와 연산은 (이름이 "/gpu:0"인) 첫 번째 GPU에 배치됨

---

```
>>> a = tf.Variable([1., 2., 3.]) # float32 변수는 GPU로 이동합니다.
>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b = tf.Variable([1, 2, 3])    # int32 변수는 CPU로 이동합니다.
>>> b.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

---

- 연산을 기본 장치 대신 다른 장치에 배치하고 싶다면 tf.device() 콘텍스트를 사용

---

```
>>> with tf.device("/cpu:0"):
...     c = tf.Variable([1., 2., 3.])
...
>>> c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

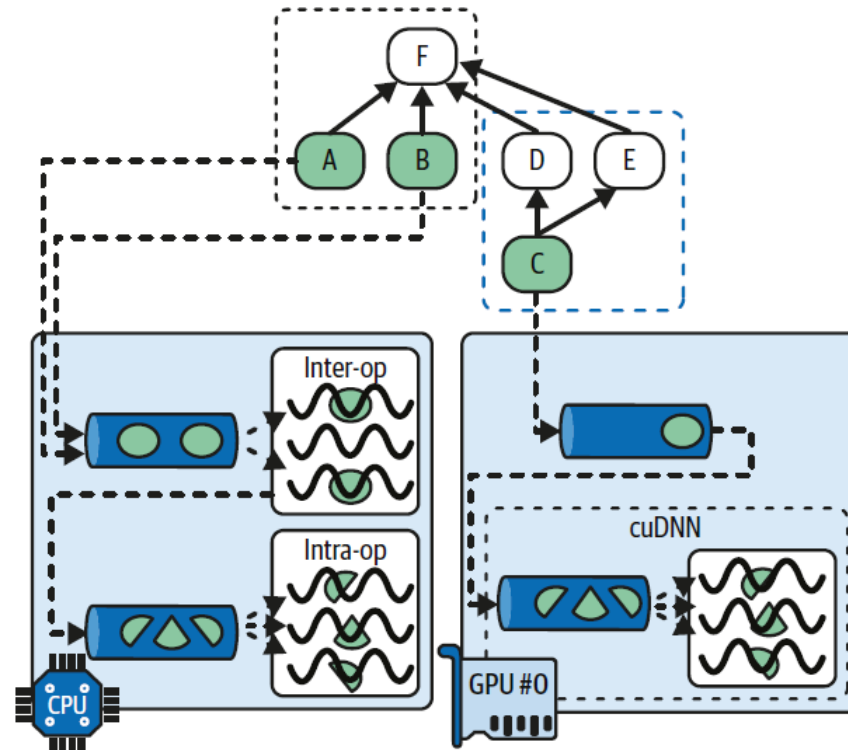
---

- 연산이나 변수를 해당 커널이 없는 장치에 배치하면 텐서플로는 기본 장치로 자동으로 돌아감
  - 이 기능은 GPU 개수가 다른 여러 컴퓨터에서 동일한 코드를 실행할 때 유용



# 계산 속도를 높이기 위해 **GPU** 사용하기(13)

## 다중 장치에서 병렬 실행하기



텐서플로 그래프의 병렬 실행



# 계산 속도를 높이기 위해 GPU 사용하기(14)

- TF 함수가 변수와 같은 상태가 있는 리소스를 수정할 때 텐서플로는 코드 사이에 명시적인 의존성이 없더라도 실행 순서가 코드의 순서와 일치하도록 보장
- GPU의 강력한 성능을 활용한 사례
  - 여러 모델을 각기 다른 GPU에서 병렬로 훈련할 수 있음
  - 하나의 GPU에서 모델을 훈련하면서 CPU에서 병렬로 전처리를 수행할 수 있음
  - 모델이 입력으로 이미지를 두 개 받고 CNN을 두 개 사용해 처리한 다음 그 출력을 합친다면 각 CNN을 다른 GPU에 배치하여 훨씬 빠르게 실행
  - 효율적인 앙상블 모델을 만들 수 있음
    - GPU마다 훈련된 다른 모델을 배치하면 훨씬 빠르게 예측을 모아서 앙상블의 최종 예측을 만들 수 있음



# 다중 장치에서 모델 훈련하기(1)

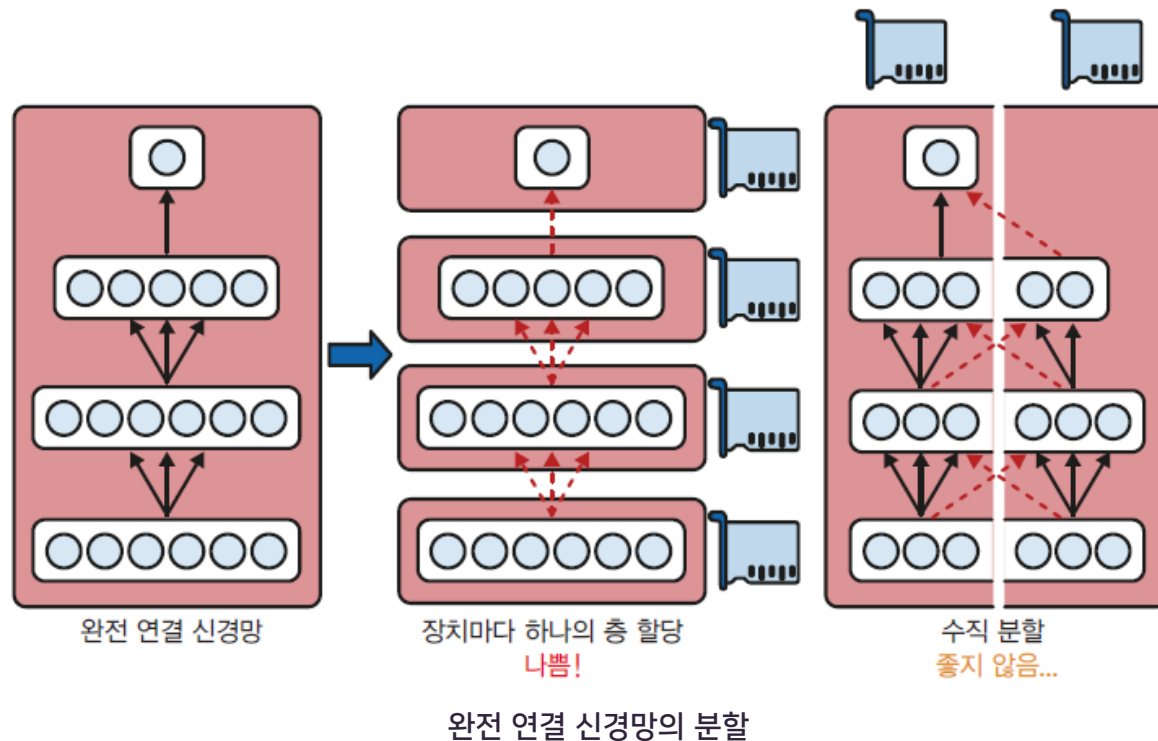
- 여러 장치에서 하나의 모델을 훈련하는 방법
  1. 모델 병렬화(model parallelism) – 모델을 여러 장치에 분할
  2. 데이터 병렬화(data parallelism) - 모델을 각 장치에 복사한 다음 복사본(replica)을 각기 다른 데이터의 일부분에서 훈련



# 다중 장치에서 모델 훈련하기(2)

## 모델 병렬화

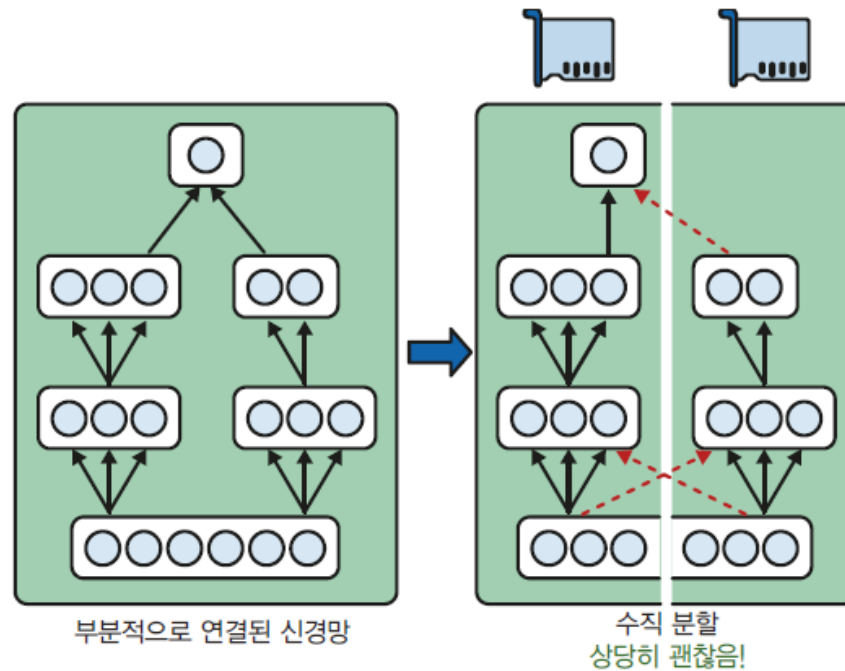
- 완전 연결 신경망의 경우 모델 병렬화는 매우 어렵고 그 효과는 신경망 모델의 구조에 매우 의존적





# 다중 장치에서 모델 훈련하기(3)

- 합성곱 신경망 같은 구조는 아래쪽 층에 부분적으로만 연결된 층을 가집니다. 그래서 여러 장치에 효율적으로 모델을 분산하기 훨씬 쉬움



부분적으로 연결된 신경망

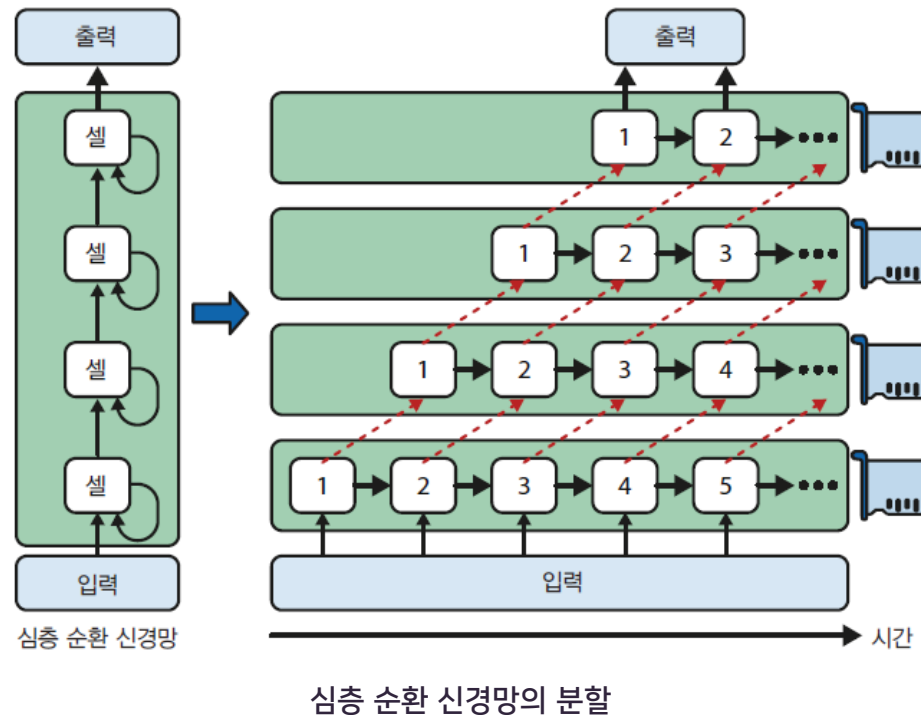
수직 분할  
상당히 괜찮음!

부분적으로 연결된 신경망의 분할



# 다중 장치에서 모델 훈련하기(4)

- 심층 순환 신경망의 경우 여러 GPU에 조금 더 효율적으로 나눌 수 있음
  - 네트워크를 수평으로 분할해서 각 층을 다른 장치에 배치하고 처리할 입력 시퀀스를 네트워크에 주입하면 첫번째 스텝에서는 (시퀀스의 첫 번째 값을 처리하기 위해) 하나의 장치만 사용되고 두 번째 스텝에서는 두 개가 사용





# 다중 장치에서 모델 훈련하기(5)

## 데이터 병렬화

- 데이터 병렬화(data parallelism) 또는 SPMD(single program, multiple data)
  - 각 장치에 모델을 복제해서 각각 다른 미니배치를 사용해 모든 모델이 동시에 훈련 스텝을 실행
  - 복제 모델에서 계산된 그레이디언트를 평균하고 그 결과를 사용해 모델 파라미터를 업데이트

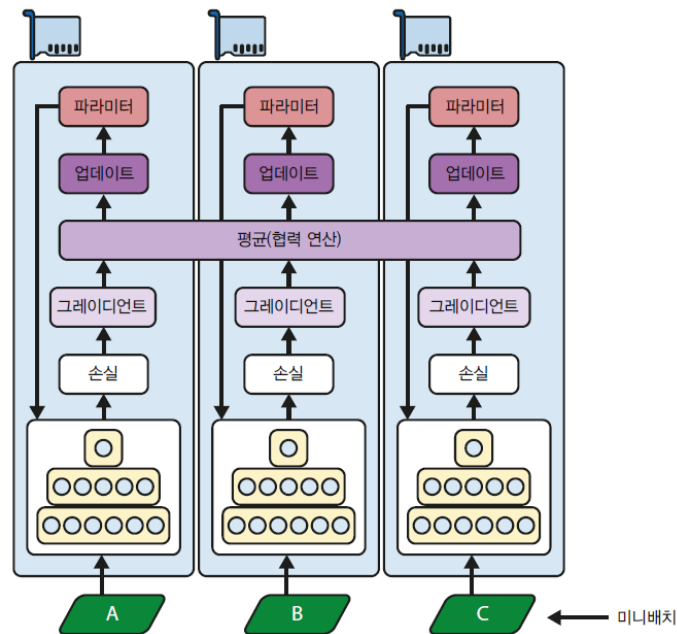




# 다중 장치에서 모델 훈련하기(6)

## 미러드 전략을 사용한 데이터 병렬화

- 모델 파라미터를 모든 GPU에 완전히 똑같이 복사하고 항상 모든 GPU에 동일한 파라미터 업데이트를 적용
- 복제된 모든 모델은 항상 완벽하게 동일한 상태로 유지
- 올리듀스(AllReduce) 알고리즘



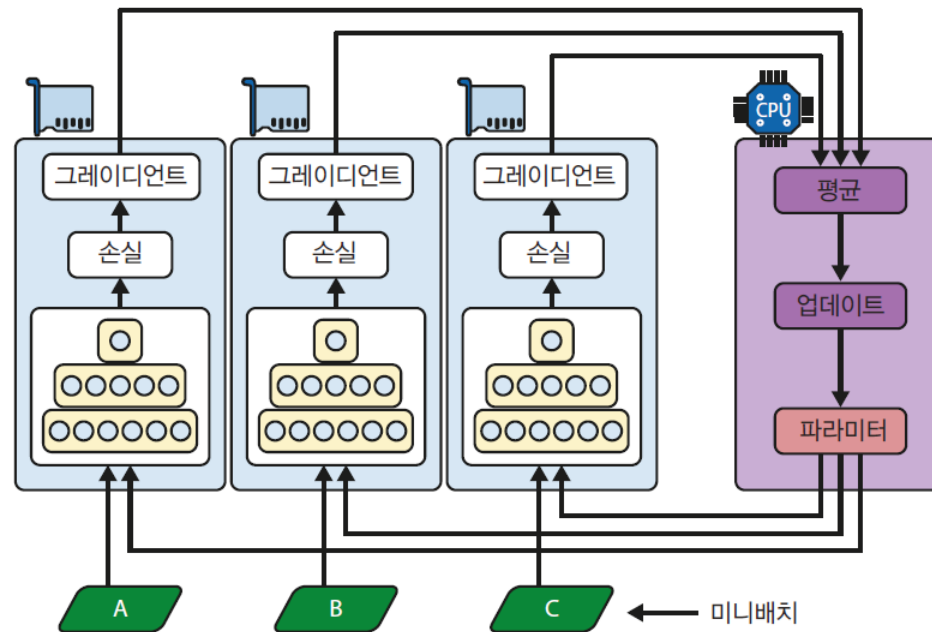
미러드 전략을 사용한 데이터 병렬화



# 다중 장치에서 모델 훈련하기(7)

## 중앙 집중적인 파라미터를 사용한 데이터 병렬화

- 계산을 수행하는 GPU 장치(워커<sup>worker</sup>) 밖에 모델 파라미터를 저장
- 분산 환경에서는 모든 파라미터를 파라미터 서버<sup>parameter server</sup>라 부르는 하나 이상의 CPU만 있는 서버에 저장할 수 있음. 이 서버의 역할은 파라미터를 보관하고 업데이트



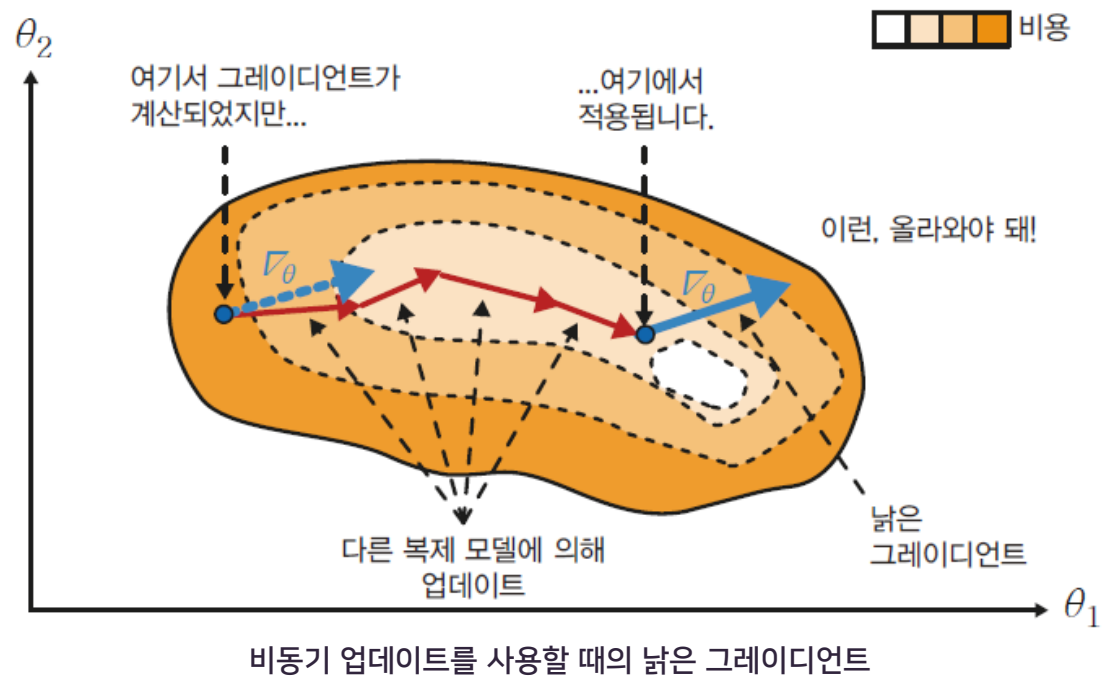
중앙 집중적인 파라미터를 사용한 데이터 병렬화



## 다중 장치에서 모델 훈련하기(8)

- 미러드 전략은 모든 GPU에 동기화된 가중치 업데이트를 사용하지만 중앙 집중적인 방식은 동기 업데이트와 비동기 업데이트를 모두 사용
  - 동기 업데이트(synchronous update)
    - 모든 그레이디언트가 준비될 때까지 그레이디언트 수집기가 기다린 다음 평균 그레이디언트를 계산하여 모델 파라미터를 업데이트할 옵티마이저에게 전달
    - 한 복제 모델이 그레이디언트 계산을 마치더라도 파라미터가 업데이트될 때까지 기다렸다가 다음 미니배치를 처리
    - 단점은 어떤 장치가 다른 장치보다 느리면 빠른 장치가 느린 장치를 매 스텝마다 기다려야 함
  - 비동기 업데이트(asynchronous update)
    - 복제 모델이 그레이디언트 계산을 끝낼 때마다 즉시 모델 파라미터를 업데이트
    - 단순하고, 동기화 지연이 없고, 대역폭을 효율적으로 사용하므로 매력적인 방법
    - 실전에서 이 방식은 잘 작동하지만 놀랍게도 전혀 효과가 없음
      - » 낡은 그레이디언트(stale gradient)

# 다중 장치에서 모델 훈련하기(9)





# 다중 장치에서 모델 훈련하기(10)

- 넓은 그래디언트 현상을 줄일 수 있는 몇 가지 방법
  - 학습률을 감소시킴
  - 넓은 그래디언트를 버리거나 크기를 줄임
  - 미니배치 크기를 조절
  - 하나의 복제 모델만 사용하여 처음 몇 번의 에포크를 시작(준비 단계<sup>warmup phase</sup>)
    - 일반적으로 그래디언트가 크거나 파라미터가 비용 함수의 계곡 부분에 아직 안착하지 못했을 때 다른 복제 모델이 파라미터를 매우 다른 방향으로 이동시킬 수 있으므로 넓은 그래디언트가 훈련 초기에 더 큰 문제를 일으키는 경향이 있음



# 다중 장치에서 모델 훈련하기(11)

## 대역폭 포화

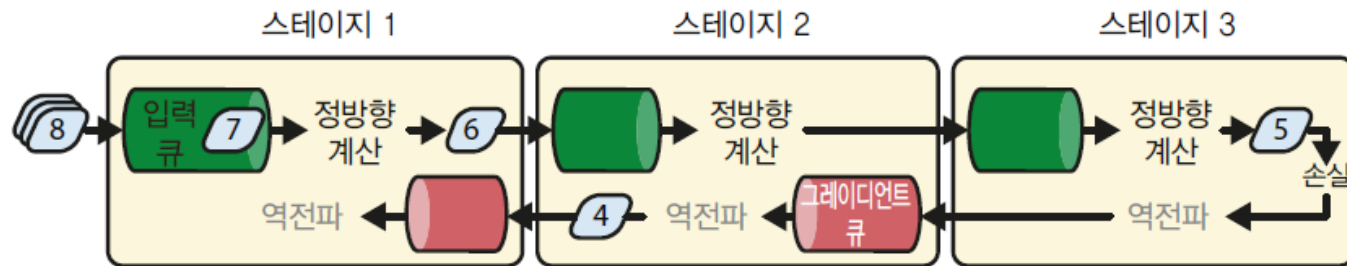
- 대역폭 포화는 전송해야 할 파라미터와 그레이디언트가 많아 대규모 밀집 모델에서 더욱 심각
  - 작은 모델에서는 덜 심하고(하지만 병렬화의 이득이 작음) 크지만 희소한 모델에서는 대부분 그레이디언트가 0이므로 효율적으로 통신할 수 있음
  - 희소 모델의 규모를 늘린 사례
    - 신경망 기계 번역: 8개 GPU에서 6배 속도 증가
    - 인셉션/이미지넷: 50개 GPU에서 32배 속도 증가
    - 랭크브레인(RankBrain): 500개 GPU에서 300배 속도 증가



# 다중 장치에서 모델 훈련하기(12)

- 2018년 마이크로소프트 PipeDream

- 파이프라인 병렬화(pipeline parallelism) – 모델 병렬화와 데이터 병렬화를 결합
- 파이프라인 병렬화는 모델을 스테이지(stage)라고 하는 연속적인 부분으로 나누어 각각 다른 머신에서 훈련
- 그 결과 모든 머신이 유휴 시간이 거의 없이 병렬로 작동하는 비동기식 파이프라인이 생성
- 가중치 스테싱(weight stashing)
  - 각 스테이지가 정방향 계산 중에 가중치를 저장했다가 역전파 중에 복원하여 정방향과 역방향에서 모두 동일한 가중치가 사용되도록 하는 방법



PipeDream의 파이프라인 병렬화

- 2022년 구글 Pathways



# 다중 장치에서 모델 훈련하기(13)

## 분산 전략 API를 사용한 대규모 훈련

- 텐서플로는 여러 장치와 머신에서 모델을 분산하는 복잡성을 모두 대신 처리해주는 매우 간단한 분산 전략 API를 제공
  - 미러드 전략으로 데이터 병렬화를 사용해 모든 GPU에서 케라스 모델을 훈련하려면 MirroredStrategy 객체를 만들고 scope() 메서드를 호출하여 분산 컨텍스트를 얻어야 함
  - 그런 다음 이 컨텍스트로 모델 생성과 컴파일 과정을 감싸고 모델의 fit() 메서드를 호출
    - fit() 메서드는 자동으로 훈련 배치를 모든 복제 모델에 나눔
    - 따라서 배치 크기가 복제 모델의 개수(즉, 가용한 GPU 개수)로 나누어 떨어지는 것이 좋음

---

```
strategy = tf.distribute.MirroredStrategy()

with strategy.scope():
    model = tf.keras.Sequential([...]) # 평소처럼 케라스 모델을 만듭니다.
    model.compile([...])               # 평소처럼 모델을 컴파일합니다.

batch_size = 100                      # 복제 모델 개수로 나누어 떨어져야 합니다.
model.fit(X_train, y_train, epochs=10,
          validation_data=(X_valid, y_valid), batch_size=batch_size)
```

---





# 다중 장치에서 모델 훈련하기(14)

- 모델 가중치를 확인

```
>>> type(model.weights[0])  
tensorflow.python.distribute.values.MirroredVariable
```

- 모델을 로드하여 가능한 모든 장치에서 실행하려면 분산 컨텍스트 안에서 `tf.keras.models.load_model()`을 호출

```
with strategy.scope():  
    model = tf.keras.models.load_model("my_mirrored_model")
```

- 가능한 GPU 장치 중 일부만 사용하고 싶다면 `MirroredStrategy` 생성자에 장치 리스트를 전달

```
strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

- 중앙 집중적인 파라미터로 데이터 병렬화를 사용한다면 `MirroredStrategy`를 `CentralStorageStrategy`로 변경

```
strategy = tf.distribute.experimental.CentralStorageStrategy()
```



# 다중 장치에서 모델 훈련하기(15)

## 텐서플로 클러스터에서 모델 훈련하기

- 텐서플로 클러스터(TensorFlow Cluster)는 보통 다른 머신에서 동시에 실행되는 텐서플로 프로세스 그룹
  - 클러스터는 신경망 모델 훈련이나 실행과 같은 작업을 완료하기 위해 서로 통신
  - 클러스터에 있는 개별 TF 프로세스를 태스크(task) 또는 TF 서버 라고 부름
  - 태스크는 IP 주소, 포트, 타입(역할 role 또는 잡 job)을 가짐
  - 타입은 "worker", "chief", "ps"(파라미터 서버), "evaluator" 중 하나
- 각 워커는 보통 GPU를 한 개 이상 가진 머신에서 계산을 수행
- 치프(chief)도 계산을 수행(하나의 워커)
  - 하지만 텐서보드 로그를 작성하거나 체크포인트를 저장하는 것과 같은 추가적인 일을 처리
  - 클러스터에는 하나의 치프가 있으며, 치프를 명시적으로 지정하지 않으면 첫 번째 워커가 치프가 됨
- 파라미터 서버는 변숫값만 저장하고 일반적으로 CPU만 있는 머신을 사용
  - 이 태스크의 타입은 ParameterServerStrategy만 사용
- 이밸류에이터(evaluator)는 평가를 담당
  - 이 타입은 자주 사용되지 않으며 보통 하나의 이밸류에이터를 사용



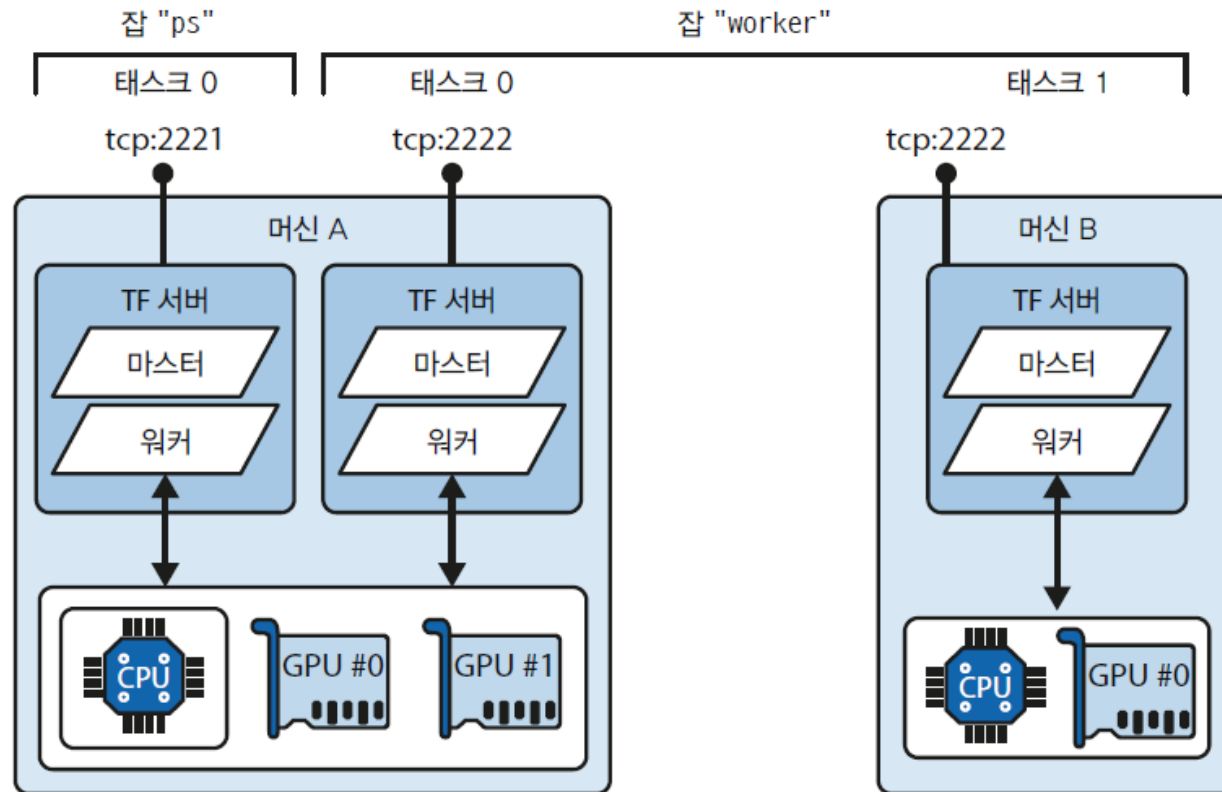
# 다중 장치에서 모델 훈련하기(16)

- 텐서플로 클러스터를 시작하기 위해 사양을 정의
  - 각 태스크의 IP 주소, TCP 포트, 타입을 정의

```
cluster_spec = {  
    "worker": [  
        "machine-a.example.com:2222",    # /job:worker/task:0  
        "machine-b.example.com:2222"    # /job:worker/task:1  
    ],  
    "ps": ["machine-a.example.com:2221"] # /job:ps/task:0  
}
```



# 다중 장치에서 모델 훈련하기(17)



텐서플로 클러스터



# 다중 장치에서 모델 훈련하기(18)

- 텐서플로를 시작하기 전에 TF\_CONFIG 환경 변수를 설정

```
os.environ["TF_CONFIG"] = json.dumps({  
    "cluster": cluster_spec,  
    "task": {"type": "worker", "index": 0}  
})
```



# 다중 장치에서 모델 훈련하기(19)

- 클러스터에서 모델을 훈련 – 미러드 전략을 시작

---

```
import tempfile
import tensorflow as tf

strategy = tf.distribute.MultiWorkerMirroredStrategy() # 시작!
resolver = tf.distribute.cluster_resolver.TFConfigClusterResolver()
print(f"Starting task {resolver.task_type} #{resolver.task_id}")
[...] # MNIST 데이터셋을 로드하고 분할합니다.

with strategy.scope():
    model = tf.keras.Sequential([...]) # 케라스 모델을 만듭니다.
    model.compile([...])               # 모델을 컴파일합니다.

model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10)

if resolver.task_id == 0: # 치프가 모델을 올바른 위치에 저장합니다.
    model.save("my_mnist_multiworker_model", save_format="tf")
else:
    tmpdir = tempfile.mkdtemp() # 다른 워커는 임시 디렉터리에 저장합니다.
    model.save(tmpdir, save_format="tf")
    tf.io.gfile.rmtree(tmpdir) # 마지막에 이 디렉터리를 삭제할 수 있습니다.
```

---



# 다중 장치에서 모델 훈련하기(20)

- 분산 전략을 위해 네트워크 통신에 gRPC를 기반으로 하는 링(ring) 올리듀스 알고리즘과 NCCL 구현 중 하나를 선택할 수 있음
- 기본적으로 텐서플로는 경험적으로 얻은 규칙을 적용하여 적절한 알고리즘을 선택
  - 다음과 같이 NCCL(또는 링)을 강제

---

```
strategy = tf.distribute.MultiWorkerMirroredStrategy(  
    communication_options=tf.distribute.experimental.CommunicationOptions(  
        implementation=tf.distribute.experimental.CollectiveCommunication.NCCL))
```

---

- 구글 클라우드에 있는 TPU를 사용할 수 있다면(예를 들어 코랩에서 하드웨어 가속기를 TPU로 설정하면) 다음과 같이 TPUStrategy를 만들 수 있음

---

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()  
tf.tpu.experimental.initialize_tpu_system(resolver)  
strategy = tf.distribute.experimental.TPUStrategy(resolver)
```

---



# 다중 장치에서 모델 훈련하기(21)

## 버텍스 AI에서 대규모 훈련 작업 실행하기

- 버텍스 AI를 사용하면 자체 훈련 코드로 사용자 정의 훈련 작업을 생성할 수 있음

```
import os
[...] # 임포트, MultiWorkerMirroredStrategy와 resolver 생성

if resolver.task_type == "chief":
    model_dir = os.getenv("AIP_MODEL_DIR") # 버텍스 AI에서 제공한 경로
    tensorboard_log_dir = os.getenv("AIP_TENSORBOARD_LOG_DIR")
    checkpoint_dir = os.getenv("AIP_CHECKPOINT_DIR")
else:
    tmp_dir = Path(tempfile.mkdtemp()) # 다른 워커는 임시 디렉토리를 사용합니다.
    model_dir = tmp_dir / "model"
    tensorboard_log_dir = tmp_dir / "logs"
    checkpoint_dir = tmp_dir / "ckpt"

callbacks = [tf.keras.callbacks.TensorBoard(tensorboard_log_dir),
              tf.keras.callbacks.ModelCheckpoint(checkpoint_dir)]
[...] # 이전처럼 전략 스코프를 사용해 빌드하고 컴파일합니다.
model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10,
          callbacks=callbacks)
model.save(model_dir, save_format="tf")
```





# 다중 장치에서 모델 훈련하기(22)

- 앞 스크립트를 기반으로 버텍스 AI에서 사용자 정의 훈련 작업을 생성

---

```
custom_training_job = aiplatform.CustomTrainingJob(  
    display_name="my_custom_training_job",  
    script_path="my_vertex_ai_training_task.py",  
    container_uri="gcr.io/cloud-aiplatform/training/tf-gpu.2-4:latest",  
    model_serving_container_image_uri=server_image,  
    requirements=["gcsfs==2022.3.0"], # 필수적이지 않으며 하나의 예일 뿐입니다.  
    staging_bucket=f"gs://{bucket_name}/staging"  
)
```

---

- 각각 2개의 GPU를 사용하는 두 대의 워커에서 실행

---

```
mnist_model2 = custom_training_job.run(  
    machine_type="n1-standard-4",  
    replica_count=2,  
    accelerator_type="NVIDIA_TESLA_K80",  
    accelerator_count=2,  
)
```

---



# 다중 장치에서 모델 훈련하기(23)

## 버텍스 AI의 하이퍼파라미터 튜닝

- 하이퍼파라미터 값을 명령줄 인수로 받아들이는 훈련 스크립트
  - 스크립트에서 다음과 같이 argparse 표준 라이브러리를 사용

---

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--n_hidden", type=int, default=2)
parser.add_argument("--n_neurons", type=int, default=256)
parser.add_argument("--learning_rate", type=float, default=1e-2)
parser.add_argument("--optimizer", default="adam")
args = parser.parse_args()
```

---



# 다중 장치에서 모델 훈련하기(24)

- 트라이얼(trial)과 학습(study)
- 훈련 스크립트는 주어진 하이퍼파라미터 값을 사용하여 모델을 빌드하고 컴파일해야 함
  - 각 트라이얼이 멀티 GPU 머신에서 실행되는 경우 미러드 분산 전략을 사용할 수 있음
  - 스크립트가 데이터셋을 로드하고 모델을 학습

---

```
import tensorflow as tf

def build_model(args):
    with tf.distribute.MirroredStrategy().scope():
        model = tf.keras.Sequential()
        model.add(tf.keras.layers.Flatten(input_shape=[28, 28], dtype=tf.uint8))
        for _ in range(args.n_hidden):
            model.add(tf.keras.layers.Dense(args.n_neurons, activation="relu"))
        model.add(tf.keras.layers.Dense(10, activation="softmax"))
        opt = tf.keras.optimizers.get(args.optimizer)
        opt.learning_rate = args.learning_rate
        model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,
                      metrics=["accuracy"])
    return model

[...] # 데이터셋을 로드합니다.
model = build_model(args)
history = model.fit([...])
```

---



## 다중 장치에서 모델 훈련하기(25)

- 스크립트는 모델의 성능을 버텍스 AI의 하이퍼파라미터 튜닝 서비스에 다시 보고하여 다음에 시도할 하이퍼파라미터를 결정할 수 있도록 해야 함
  - 버텍스 AI 훈련 가상 머신에 자동으로 설치되는 hypertune 라이브러리를 사용

---

```
import hypertune

hypertune = hypertune.HyperTune()
hypertune.report_hyperparameter_tuning_metric(
    hyperparameter_metric_tag="accuracy",          # 보고할 측정 지표 이름
    metric_value=max(history.history["val_accuracy"]), # 측정값
    global_step=model.optimizer.iterations.numpy(),
)
```

---



# 다중 장치에서 모델 훈련하기(26)

- 실행할 머신 유형을 정의
  - 사용자 정의 작업을 구성
  - 버텍스 AI가 각 트라이얼의 템플릿으로 사용

---

```
trial_job = aiplatform.CustomJob.from_local_script(  
    display_name="my_search_trial_job",  
    script_path="my_vertex_ai_trial.py", # 훈련 스크립트 경로  
    container_uri="gcr.io/cloud-aiplatform/training/tf-gpu.2-4:latest",  
    staging_bucket=f"gs://{bucket_name}/staging",  
    accelerator_type="NVIDIA_TESLA_K80",  
    accelerator_count=2, # 이 예에서는 각 트라이얼이 2개의 GPU를 가집니다.  
)
```

---



# 다중 장치에서 모델 훈련하기(27)

- 하이퍼파라미터 튜닝 작업을 만들고 실행할 준비 완료

---

```
from google.cloud.aiplatform import hyperparameter_tuning as hpt

hp_job = aiplatform.HyperparameterTuningJob(
    display_name="my_hp_search_job",
    custom_job=trial_job,
    metric_spec={"accuracy": "maximize"},
    parameter_spec={
        "learning_rate": hpt.DoubleParameterSpec(min=1e-3, max=10, scale="log"),
        "n_neurons": hpt.IntegerParameterSpec(min=1, max=300, scale="linear"),
        "n_hidden": hpt.IntegerParameterSpec(min=1, max=10, scale="linear"),
        "optimizer": hpt.CategoricalParameterSpec(["sgd", "adam"]),
    },
    max_trial_count=100,
    parallel_trial_count=20,
)
hp_job.run()
```

---



## 다중 장치에서 모델 훈련하기(28)

- 작업이 완료되면 `hp_job.trials`를 사용하여 트라이얼 결과를 추출
  - 각 트라이얼의 결과는 하이퍼파라미터 값과 결과 측정값을 포함하는 프로토콜 버퍼 객체로 표현

---

```
def get_final_metric(trial, metric_id):
    for metric in trial.final_measurement.metrics:
        if metric.metric_id == metric_id:
            return metric.value

trials = hp_job.trials
trial accuracies = [get_final_metric(trial, "accuracy") for trial in trials]
best_trial = trials[np.argmax(trial accuracies)]
```

---



# 다중 장치에서 모델 훈련하기(29)

- 트라이얼의 정확도와 하이퍼파라미터 값을 확인

---

```
>>> max(trial_accuracies)
0.977400004863739
>>> best_trial.id
'98'
>>> best_trial.parameters
[parameter_id: "learning_rate" value { number_value: 0.001 },
 parameter_id: "n_hidden" value { number_value: 8.0 },
 parameter_id: "n_neurons" value { number_value: 216.0 },
 parameter_id: "optimizer" value { string_value: "adam" }
]
```

---





# 다중 장치에서 모델 훈련하기(30)

버텍스 AI에서 케라스 튜너를 사용한 하이퍼파라미터 튜닝

- 케라스 튜너는 하이퍼파라미터 탐색을 여러 머신에 분산하여 확장하는 간단한 방법을 제공
  - 머신마다 3개의 환경 변수를 설정한 다음 각 머신에서 일반 케라스 튜너 코드를 실행하기만 하면 됨
  - 모든 머신에서 정확히 동일한 스크립트를 사용할 수 있음
  - 각 머신에 설정해야 하는 세 개의 환경 변수
    - `kerastuner_tuner_id`
      - »치프 머신의 경우는 "chief"이거나 워커 머신의 고유 식별자(예: "worker0", "worker1" 등)
    - `kerastuner_oracle_ip`
      - »치프 머신의 IP 주소 또는 호스트 이름
      - »일반적으로 치프는 "0.0.0.0"을 사용하여 머신의 모든 IP 주소에서 대기 listen해야 함
    - `kerastuner_oracle_port`
      - »치프가 수신 대기할 TCP 포트

# 연습문제(1)

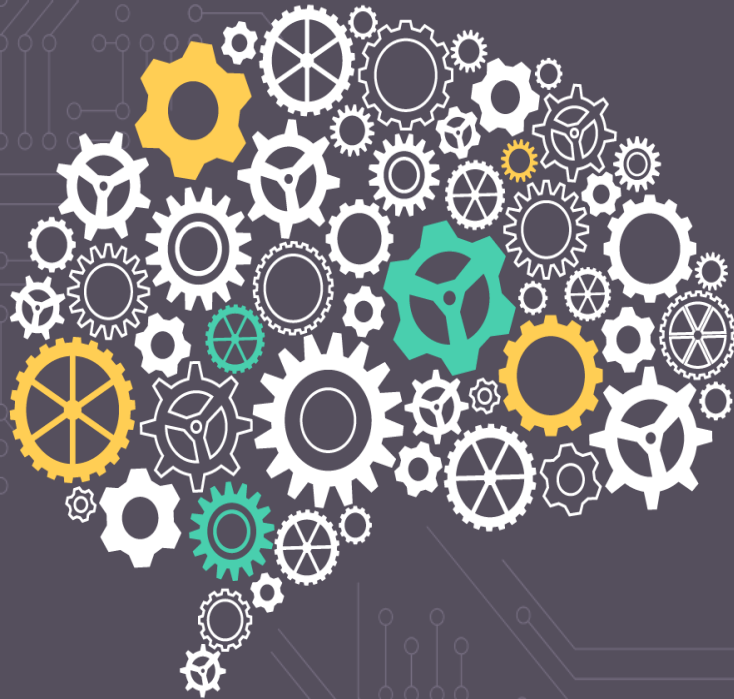


1. SavedModel에 포함된 것은 무엇인가요? 이 내용을 어떻게 조사할 수 있나?
2. TF 서빙을 언제 사용해야 하고 주기능은 무엇인가? 배포를 위해 쓸 수 있는 도구는 무엇인가?
3. 여러 TF 서빙 인스턴스로 모델을 어떻게 배포하나?
4. TF 서빙으로 서비스되는 모델에 쿼리하기 위해 REST API 대신 gRPC API를 사용해야 할 때는 언제인가?
5. TFLite가 모바일이나 임베디드 장치에서 실행되도록 모델 크기를 줄이기 위한 방법은 무엇인가?
6. 양자화를 고려한 훈련(quantization-aware training)은 무엇인가? 왜 필요한가?
7. 모델 병렬화와 데이터 병렬화는 무엇인가? 왜 일반적으로 후자를 권장하나?
8. 서버 여러 대에서 모델을 훈련할 때 어떤 분산 전략을 쓸 수 있나? 선택 기준은 무엇인가?

## 연습문제(2)

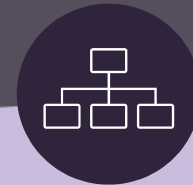
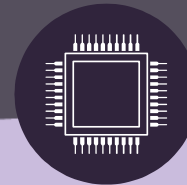


9. (어떤 모델이든) 모델을 훈련하고 TF 서빙이나 구글 클라우드 AI 플랫폼에 배포하기. REST API나 gRPC API를 사용해 쿼리하는 클라이언트 코드를 작성해보기. 모델을 업데이트하고 새로운 버전을 배포해보기. 클라이언트 코드가 새로운 버전으로 쿼리할 것임. 첫 번째 버전으로 롤백하기
10. 하나의 머신에 여러 개의 GPU에서 MirroredStrategy 전략으로 모델을 훈련해보기(코랩 GPU 런타임으로 가상 GPU 두 개를 만들 수도 있음). CentralStorageStrategy 전략으로 모델을 재훈련하고 훈련 시간을 비교해보기
11. 케라스 튜너 또는 버텍스 AI의 하이퍼파라미터 튜닝 서비스를 사용하여 버텍스 AI에서 원하는 모델을 미세 튜닝해보기



# Thank you!

See you next time.



담당교수 : 유 현 주

comjoo@uok.ac.kr

