<Study Group 계획서>

Study Group 계획서

과 목 명	Kaggle Code 탐방 Focusing on Machine & Deep Learning, Natural Language Processing
강의유형	이론 + 실습형 + PPT
개요	Kaggle Code 를 필사하며, 기본적인 역량을 키우고, 그에 맞춰서 어떻게 본인의 Algorithm 의 효율을 높힐 수 있을지 연구하고 의논하는 것이 주된 목적을 갖는 스터디 그룹입니다. Step 1. 정형 데이터 필사 Step 2. 이미지 및 텍스트 접근 프로세스 능률 향상 코드 필사 Step 3. 공모전 순으로 진행 될 예정이며, 각 인원이 각 주마다 커널을 하나씩 공유하며, 의논 하는 것을 주요 방법으로 채택할 예정이며, 논문 또한 필요할 때 마다 공유 및 발표가 이뤄질 예정입니다. 공모전에서는 스스로가 그룹에서 어떻게 녹아들면 좋은지 연구할 수 있는 유익한 시간이 되었으면 좋겠습니다.
스케줄	매 주 토요일 12:00 ~ 14:00 1. 처음 10~20 분 정도는 해결하지 못 한 오류&에러 등을 해결하는 시간 or 근황토크 등 Ice break 시간 2. 필사한 코딩을 공유하면서 본인이 맡은 코딩의 특징이나 장 단점을 공유함 (약 10~20 분 가량) 3. 발표에 대한 질의문답시간 및 발표에 포함되어 있지 않았던 정보나 장단점 파악 및 개선방안 (나머지 모든 시간)
그룹목표	개인적인 역량 상승 및 실무 및 프로젝트를 진행함으로서, 팀워크를 향상시키는 것이 목적을 두고 있습니다.

기대효과 및 의도 도움말	프로젝트를 기반으로한 팀워크가 중심이 되는 필드에서 본인 스스로가 가지는 장점 및 문제점을 파악하여 발전 및 보완을 거칠 수 있는 기회가 될 것으로 사려됩니다.		
진행방법	토론(토의) , 세미나 , 팀별 학습 , 수준별학습 ,학습자 참여학습(발표 등)		
책임자	김지성 010-7523-1404 박인호 010-8234-5848	강의장소	온라인 및 오프라인
주교재	Github ,Kaggle, Dacon 등의 Website		
부교재 및 참고도서	Stack Over Flow		

< 필사 예 제 >

기본적인 틀은 딱히 존재하지 않고, 본인의 스타일대로 작업하는 것을 기본으로 합니다.. 다만, 공유를 목적으로 하므로, 가독성 유지에 노력을 기울여주시면 감사드리겠습니다. *^^*

(다음 페이지부터>>>

Building Dense Layer

Residual Network is the origin of Densely Connected Convolutional Network

➔ Skip-Connection plays a significant role in ResNet

➔ Adding feature before output, but Pre-Activation is strongly recommended in order to obtain Identity Mapping

Pre-Activation

➔ Conv-BN-ReLu → BN-ReLU-Conv(Gradient Highway is added)

DenseNet

➔ Suggesting Dense Block

➔ In Dense Block, it uses BN-ReLU-Conv(Pre-Activation Structure) like ResNet

➔ Skip Connection connected without any rules looks complicated;however, in different point of view, the results are exactly same as piling features up into a map and concatenating.

```
import tensorflow as tf
import numpy as np


EPOCHS = 10


class DenseUnit(tf.keras.Model):
    def __init__(self, filter_out, kernel_size):
        super(DenseUnit, self).__init__()
        self.bn = tf.keras.layers.BatchNormalization()
```

BatchNomalization

➔ Its structure consists of Fully Connected, Batch Norm, and Relu layers

➔ While training process happens, it normalizes batches by utilizing avg and distribution

$$BatchNormalization(x) = \gamma(\frac{X - \mu_{batch}}{\sigma_{batch}}) + \beta$$

X input --------- >> Batch Norm ----------→> BatchNormalization(X)

Calculating by a batch

$$\begin{cases} \mu_{\text{batch}} = \frac{1}{B}\sum_i x_i \\ \sigma^2_{\text{batch}} = \frac{1}{B}\sum_i (x_i - \mu_{\text{batch}})^2 \end{cases}$$

➔ Matching same scale for all features benefits to decide training rate

➔ By training more scale and bias, the distribution from a model becomes more suitable to activation

Inference Phase

- a mathematical formula is pretty similar from Batch.Norm. but calculating avg.distance while processing

$$\begin{cases} \mu_{BN} = \frac{1}{N}\sum_i \mu^i_{\text{batch}} \\ \sigma_{BN} = \frac{1}{N}\sum_i \sigma^i_{\text{batch}} \end{cases}$$

학습과정에서 이동평균을 계산

정규화

$$BN(X) = \gamma\left(\frac{X - \mu_{BN}}{\sigma_{BN}}\right) + \beta$$

역전파로 학습된 값

➔ In interference phase, holding an average and distribution are held by avg.distance or avg.exponential

➔ Also, it is possible to shorten sum and multiply equations by normalizing and adding Scale, Bias up into a model

```
self.conv = tf.keras.layers.Conv2D(filter_out, kernel_size, padding = 'same')
        # free activation Structure
        self.concat = tf.keras.layers.Concatenate()

    def call(self, x, training=False, mask=None): # x : (Batch, H, W, Ch_in)
        h = self.bn(x, training = training)
        h = tf.nn.relu(h)
        h = self.conv(h) # h : (Batch, H, W, filter_output)
        return self.concat([x, h]) # (Batch, H, W, Ch_in + filter_output)

class DenseLayer(tf.keras.Model):
    def __init__(self, num_unit, growth_rate, kernel_size):
        super(DenseLayer, self).__init__()
        self.sequence = list()
        for idx in range(num_unit):
```

#As Dense_Unit is being called, its output increase by filter_output; hence, we name it

```
    def call(self, x, training=False, mask=None):
        for unit in self.sequence:
            x = unit(x, training = training)
        return x


class TransitionLayer(tf.keras.Model):

    def __init__(self, filters, kernel_size):
        super(TransitionLayer, self).__init__()
        self.conv = tf.keras.layers.Conv2D(filters, kernel_size, padding = 'same'
)
        self.pool = tf.keras.layers.MaxPool2D()

    def call(self, x, training=False, mask=None):
        x = self.conv(x)
        return self.pool(x)
```

```
# Dense Layer tends to grow up as num_unit * growrh_rate as each time epoch runs,
  meaning that it has high probability to get larger unexpectedly

# Therefore, we found it necessary to use trainition layer to prevent it.
```

```
class DenseNet(tf.keras.Model):
    def __init__(self):
        super(DenseNet, self).__init__()
        self.conv1 = tf.keras.layers.Conv2D(8, (3, 3), padding = 'same', activati
on = 'relu')
      # 28 * 28 * 8 cuz we use mnist
```

```
        self.dl1 = DenseLayer(2, 4, (3, 3)) # 28 * 28 * 16
        self.tl1 = TransitionLayer(16, (3, 3)) # 14 * 14 * 16

        self.dl2 = DenseLayer(2, 8, (3, 3)) # 14 * 14 * 32
        self.tl2 = TransitionLayer(32, (3, 3)) # 7 * 7 * 32

        self.dl3 = DenseLayer(2, 16, (3, 3)) # 7 * 7 * 64

        self.flatten = tf.keras.layers.Flatten()
```

```
# Implement training loop

@tf.function
def train_step(model, images, labels, loss_object, optimizer, train_loss, train_a
ccuracy):
    with tf.GradientTape() as tape:
        predictions = model(images, training = True)
        loss = loss_object(labels, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)

    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
    train_loss(loss)
    train_accuracy(labels, predictions)

# Implement algorithm test
@tf.function
def test_step(model, images, labels, loss_object, test_loss, test_accuracy):
    predictions = model(images, training = True)

    t_loss = loss_object(labels, predictions)
    test_loss(t_loss)
    test_accuracy(labels, predictions)


mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

x_train = x_train[..., tf.newaxis].astype(np.float32)
x_test = x_test[..., tf.newaxis].astype(np.float32)

train_ds = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(10000).
batch(32)
test_ds = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(32)


# Create model
model = DenseNet()

# Define loss and optimizer
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()

# Define performance metrics
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy'
)


test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')


for epoch in range(EPOCHS):
    for images, labels in train_ds:
        train_step(model, images, labels, loss_object, optimizer, train_loss, tra
in_accuracy)

    for test_images, test_labels in test_ds:
        test_step(model, test_images, test_labels, loss_object, test_loss, test_a
ccuracy)

    template = 'Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, Test Accuracy: {
}'
    print(template.format(epoch + 1,
                          train_loss.result(),
                          train_accuracy.result() * 100,
                          test_loss.result(),
                          test_accuracy.result() * 100))
    train_loss.reset_states()
    train_accuracy.reset_states()
    test_loss.reset_states()
    test_accuracy.reset_states()


Result

Operating up to 10 epochs – Final.Loss – 0.025 – Final.Accuracy – 99.365 – Test.Loss – 98.800
```