3, 4주차 (11/4~11/14)

3L

Target Paper Searching

https://www.notion.so/modulabs/Lists-36411f7219784950ac2d1c46e5a80fc1

Lists

- Domain-adaptive deep network compression
- Compressing Neural Networks: Towards Determining the Optimal Layer-wise Decom...
- DyLoRA: Parameter-Efficient Tuning of Pretrained Models using Dynamic Search-Free...
- Decomposable-Net: Scalable Low-Rank Compression for Neural Networks
- A flexible, extensible software framework for model compression based on the LC alg...
- Low-rank Compression of Neural Nets: Learning the Rank of Each Layer
- THE EXPRESSIVE POWER OF LOW-RANK ADAPTATION
- One-for-All: Generalized LoRA for Parameter-Efficient Fine-tuning

'/'를 입력해 명령어 사용

Target Paper Searching : For Experimental Setup

- Low Rank Approximation 관련 논문들을 Review 했을 때, 대부분 CV Model 을 위한 것이었음.
- 그중 비교적 최근 논문이면서, 실험 셋팅 세부 사항을 확인 가능한 논문의 실험 셋팅을 참조하기로 함.

G] 18 Nov 2021

Compressing Neural Networks: Towards Determining the Optimal Layer-wise Decomposition

Lucas Liebenwein* MIT CSAIL lucas@csail.mit.edu
Alaa Maalouf* University of Haifa alaamalouf12@gmail.com
Oren Gal University of Haifa orengal@alumni.technion.ac.il
Dan Feldman University of Haifa dannyf.post@gmail.com
Daniela Rus MIT CSAIL rus@csail.mit.edu

Abstract

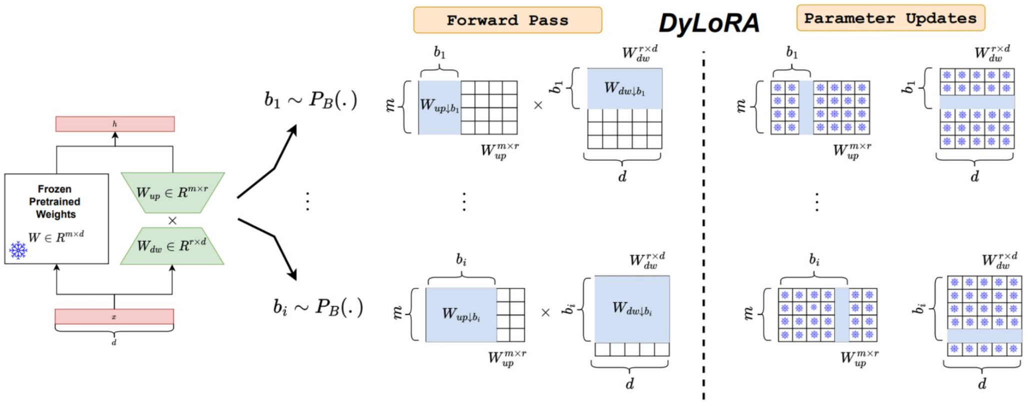
We present a novel global compression framework for deep neural networks that automatically analyzes each layer to identify the optimal per-layer compression ratio, while simultaneously achieving the desired overall compression. Our algo-

Table 3: The experimental hyperparameters for training, compression, and retraining for the tested **CIFAR10** network architectures. "LR" and "LR decay" hereby denote the learning and the (multiplicative) learning rate decay, respectively, that is deployed at the epochs as specified. "$\{x, \ldots\}$" indicates that the learning rate is decayed every x epochs.

	Hyperparameters		VGG16	Resnet20	DenseNet22	WRN-16-8
CIFAR10	(Re-)Training	Test accuracy (%)	92.81	91.4	89.90	95.19
		Loss	cross-entropy	cross-entropy	cross-entropy	cross-entropy
		Optimizer	SGD	SGD	SGD	SGD
		Epochs	300	182	300	200
		Warm-up	10	5	10	5
		Batch size	256	128	64	128
		LR	0.05	0.1	0.1	0.1
		LR decay	0.5@{30, ...}	0.1@{91, 136}	0.1@{150, 225}	0.2@{60, ...}
		Momentum	0.9	0.9	0.9	0.9
		Nesterov	✗	✗	✓	✓
		Weight decay	5.0e-4	1.0e-4	1.0e-4	5.0e-4
	Compression	α	0.80	0.80	0.80	0.80
		n_{seed}	15	15	15	15

Target Paper Searching : For Experimental Setup (미정)

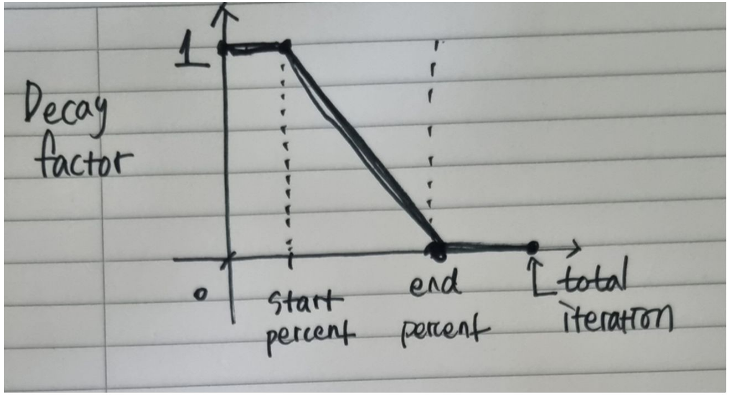
효율적으로 rank searching, selection 가능한 DyLoRA 방식을 사용해 보는것도 좋을 수 있다



진행 사항

- **Weight Decay Scheduler**
- Conv LoRA Layer

Weight Decay scheduler




```python
# Calculate the linear decay factor
if self.current_step < self.start_step:
    self.decay_factor.assign(1.0)  # Decay has not started yet
elif self.current_step > self.end_step:
    self.decay_factor.assign(tf.cast(self.min_decay_factor, dtype=tf.float32))  # Ensure float32 type fo
else:
    # Linear decay between start_step and end_step
    self.decay_factor.assign(1.0 - ((tf.cast(self.current_step, dtype=tf.float32) - self.start_step) /
                                    (self.end_step - self.start_step) *
                                    (1.0 - tf.cast(self.min_decay_factor, dtype=tf.float32))))

# Matrix multiplication for A and B weights with inputs
lora_A_output = tf.matmul(self.A_weight, tf.transpose(inputs))  # Ax
lora_output = tf.transpose(tf.matmul(self.B_weight, lora_A_output) * self._scale)  # BAx Transpose back

if training:
    original_output = self.original_layer(inputs) * self.decay_factor
    # Increment the step counter
    self.current_step.assign_add(1)

    return original_output + lora_output

else:
    # 추론 모드에서는 LoRA 출력만 반환
    # Modify lora_output based on decay_factor
    #if tf.not_equal(self.decay_factor, 1.0):
      #lora_output /= (1 - self.decay_factor)
    return lora_output
```

Weight Decay (in Toy Project)

https://colab.research.google.com/drive/16qBMU346ABOpNxjzCN99TRMTt95q_Pg7?usp=sharing#scrollTo=tps9ex9N4MyY

- 우리의 이론적 예상은, LoRA Output 에 weight transfer 가 서서히 진행된다는 것임.

- 따라서, Weight Decay 가 곱해진, Original Output + LoRa Output 에 의한 acc 는 점점 감소하고, LoRa Output 에 의한 acc 는 점점 증가할 것이라고 예상함.

- 그런데 예상과 다르게, Epoch 24/30 정도 까지, Original Output + LoRa Output 에 의한 acc 는 거의 유지 되었고 LoRa Output 은 0.05 정도의 굉장히 낮은 (단독적으로는 의미가 없어보이는) acc 를 보임.

- 이후로는 Original Output + LoRa Output 의 성능은 10 ~ 20 % 정도 감소하고, LoRa Output 에 대한 acc 는 0.77 정도까지 높게 나옴.

Weight B 초기화 방식 변화

```
self.B_weight = self.add_weight(
    name="lora_B_weight",
    shape=(self.original_layer.units, self.rank),
    initializer="zeros",
    trainable=self.trainable,
)
```

Adapter 단독으로 full training 할 때 학습 안 되는 현상 발견

```
self.B_weight = self.add_weight(
    name="lora_B_weight",
    shape=(self.original_layer.units, self.rank),
    initializer=keras.initializers.VarianceScaling(
        scale=math.sqrt(5), mode="fan_in", distribution="uniform"
    ),
    trainable=self.trainable,
)
```

LoRA call

```python
def call(self, inputs, training=None):
        if training is None:
            training = self.trainable

        # Calculate the linear decay factor
        if self.current_step < self.start_step:
            self.decay_factor.assign(1.0)  # Decay has not started yet
        elif self.current_step > self.end_step:
            self.decay_factor.assign(tf.cast(self.min_decay_factor, dtype=tf.float32))  # Ensure float32 type for consistency
        else:
            # Linear decay between start_step and end_step
            self.decay_factor.assign(1.0 - ((tf.cast(self.current_step, dtype=tf.float32) - self.start_step) /
                                    (self.end_step - self.start_step) *
                                    (1.0 - tf.cast(self.min_decay_factor, dtype=tf.float32))))

        # Matrix multiplication for A and B weights with inputs
        lora_A_output = tf.matmul(self.A_weight, tf.transpose(inputs))  # Ax
        lora_output = tf.transpose(tf.matmul(self.B_weight, lora_A_output) * self._scale)  # BAx Transpose back to [batch_size, original_layer.units]

        if training:
            original_output = self.original_layer(inputs) * self.decay_factor
            # Increment the step counter
            self.current_step.assign_add(1)

            return original_output + lora_output

        else:
            # 추론 모드에서는 LoRA 출력만 반환
            # Modify lora_output based on decay_factor
            #if tf.not_equal(self.decay_factor, 1.0):
              #lora_output /= (1 - self.decay_factor)
            return lora_output
```

original layer decay factor compute

```
Epoch 1/30
1997/2000 [============================>.] - ETA: 0s - loss: 0.3502 - accuracy: 0.8960
End of epoch 1, LoraLayer 0: 2000 Step
End of epoch 1, LoraLayer 0: Decay factor: 1.0
End of epoch 1, LoraLayer 1: 2000 Step
End of epoch 1, LoraLayer 1: Decay factor: 1.0
End of epoch 1, LoraLayer 2: 2000 Step
End of epoch 1, LoraLayer 2: Decay factor: 1.0

Testing loss: 3.0721025466918945, acc: 0.08816666901111603

2000/2000 [==============================] - 15s 7ms/step - loss: 0.3501 - accuracy: 0.8960 - val_loss: 3.0721 - val_accuracy: 0.0882
Epoch 2/30
1998/2000 [============================>.] - ETA: 0s - loss: 0.2394 - accuracy: 0.9125
End of epoch 2, LoraLayer 0: 4000 Step
End of epoch 2, LoraLayer 0: Decay factor: 1.0
End of epoch 2, LoraLayer 1: 4000 Step
End of epoch 2, LoraLayer 1: Decay factor: 1.0
End of epoch 2, LoraLayer 2: 4000 Step
End of epoch 2, LoraLayer 2: Decay factor: 1.0

Testing loss: 3.277391195297241, acc: 0.045249998569488525

2000/2000 [==============================] - 13s 7ms/step - loss: 0.2393 - accuracy: 0.9125 - val_loss: 3.2774 - val_accuracy: 0.0452
Epoch 3/30
2000/2000 [==============================] - ETA: 0s - loss: 0.2263 - accuracy: 0.9160
End of epoch 3, LoraLayer 0: 6000 Step
End of epoch 3, LoraLayer 0: Decay factor: 1.0
End of epoch 3, LoraLayer 1: 6000 Step
End of epoch 3, LoraLayer 1: Decay factor: 1.0
End of epoch 3, LoraLayer 2: 6000 Step
End of epoch 3, LoraLayer 2: Decay factor: 1.0

Testing loss: 2.6456737518310547, acc: 0.08524999767541885

2000/2000 [==============================] - 13s 7ms/step - loss: 0.2263 - accuracy: 0.9160 - val_loss: 2.6457 - val_accuracy: 0.0852
Epoch 4/30
1994/2000 [============================>.] - ETA: 0s - loss: 0.2153 - accuracy: 0.9199
End of epoch 4, LoraLayer 0: 8000 Step
End of epoch 4, LoraLayer 0: Decay factor: 0.9583333134651184
End of epoch 4, LoraLayer 1: 8000 Step
End of epoch 4, LoraLayer 1: Decay factor: 0.9583333134651184
End of epoch 4, LoraLayer 2: 8000 Step
End of epoch 4, LoraLayer 2: Decay factor: 0.9583333134651184

Testing loss: 2.4953575134277344, acc: 0.10341666638851166
```

```
2000/2000 [==============================] - 15s 7ms/step - loss: 0.2008 - accuracy: 0.9234 - val_loss: 2.3798 - val_accuracy: 0.0835
Epoch 9/30
2000/2000 [==============================] - ETA: 0s - loss: 0.1982 - accuracy: 0.9247
End of epoch 9, LoraLayer 0: 18000 Step
End of epoch 9, LoraLayer 0: Decay factor: 0.75
End of epoch 9, LoraLayer 1: 18000 Step
End of epoch 9, LoraLayer 1: Decay factor: 0.75
End of epoch 9, LoraLayer 2: 18000 Step
End of epoch 9, LoraLayer 2: Decay factor: 0.75

Testing loss: 2.3738210201263428, acc: 0.07141666859388351

2000/2000 [==============================] - 14s 7ms/step - loss: 0.1982 - accuracy: 0.9247 - val_loss: 2.3738 - val_accuracy: 0.0714
Epoch 10/30
1999/2000 [============================>.] - ETA: 0s - loss: 0.1988 - accuracy: 0.9238
End of epoch 10, LoraLayer 0: 20000 Step
End of epoch 10, LoraLayer 0: Decay factor: 0.7083333730697632
End of epoch 10, LoraLayer 1: 20000 Step
End of epoch 10, LoraLayer 1: Decay factor: 0.7083333730697632
End of epoch 10, LoraLayer 2: 20000 Step
End of epoch 10, LoraLayer 2: Decay factor: 0.7083333730697632

Testing loss: 2.4048027992248535, acc: 0.06758332997560501

2000/2000 [==============================] - 13s 6ms/step - loss: 0.1988 - accuracy: 0.9238 - val_loss: 2.4048 - val_accuracy: 0.0676
Epoch 11/30
1992/2000 [============================>.] - ETA: 0s - loss: 0.1981 - accuracy: 0.9252
End of epoch 11, LoraLayer 0: 22000 Step
End of epoch 11, LoraLayer 0: Decay factor: 0.6666666269302368
End of epoch 11, LoraLayer 1: 22000 Step
End of epoch 11, LoraLayer 1: Decay factor: 0.6666666269302368
End of epoch 11, LoraLayer 2: 22000 Step
End of epoch 11, LoraLayer 2: Decay factor: 0.6666666269302368

Testing loss: 2.410620927810669, acc: 0.06849999725818634

2000/2000 [==============================] - 12s 6ms/step - loss: 0.1978 - accuracy: 0.9252 - val_loss: 2.4106 - val_accuracy: 0.0685
Epoch 12/30
1991/2000 [============================>.] - ETA: 0s - loss: 0.1959 - accuracy: 0.9277
End of epoch 12, LoraLayer 0: 24000 Step
End of epoch 12, LoraLayer 0: Decay factor: 0.625
End of epoch 12, LoraLayer 1: 24000 Step
End of epoch 12, LoraLayer 1: Decay factor: 0.625
End of epoch 12, LoraLayer 2: 24000 Step
End of epoch 12, LoraLayer 2: Decay factor: 0.625

Testing loss: 2.4252254962921143, acc: 0.12083332985639572
```

```
2000/2000 [==============================] - 13s 7ms/step - loss: 0.2972 - accuracy: 0.8922 - val_loss: 2.4693 - val_accuracy: 0.1739
Epoch 26/30
1992/2000 [============================>.] - ETA: 0s - loss: 0.3515 - accuracy: 0.8742
End of epoch 26, LoraLayer 0: 52000 Step
End of epoch 26, LoraLayer 0: Decay factor: 0.04166668653488159
End of epoch 26, LoraLayer 1: 52000 Step
End of epoch 26, LoraLayer 1: Decay factor: 0.04166668653488159
End of epoch 26, LoraLayer 2: 52000 Step
End of epoch 26, LoraLayer 2: Decay factor: 0.04166668653488159

Testing loss: 1.5427169799804688, acc: 0.35616666078567505

2000/2000 [==============================] - 17s 8ms/step - loss: 0.3518 - accuracy: 0.8741 - val_loss: 1.5427 - val_accuracy: 0.3562
Epoch 27/30
1995/2000 [============================>.] - ETA: 0s - loss: 0.4663 - accuracy: 0.8369
End of epoch 27, LoraLayer 0: 54000 Step
End of epoch 27, LoraLayer 0: Decay factor: 0.0
End of epoch 27, LoraLayer 1: 54000 Step
End of epoch 27, LoraLayer 1: Decay factor: 0.0
End of epoch 27, LoraLayer 2: 54000 Step
End of epoch 27, LoraLayer 2: Decay factor: 0.0

Testing loss: 0.5927927494049072, acc: 0.7962499856948853

2000/2000 [==============================] - 12s 6ms/step - loss: 0.4663 - accuracy: 0.8370 - val_loss: 0.5928 - val_accuracy: 0.7962
Epoch 28/30
1993/2000 [============================>.] - ETA: 0s - loss: 0.5305 - accuracy: 0.8193
End of epoch 28, LoraLayer 0: 56000 Step
End of epoch 28, LoraLayer 0: Decay factor: 0.0
End of epoch 28, LoraLayer 1: 56000 Step
End of epoch 28, LoraLayer 1: Decay factor: 0.0
End of epoch 28, LoraLayer 2: 56000 Step
End of epoch 28, LoraLayer 2: Decay factor: 0.0

Testing loss: 0.5939328670501709, acc: 0.7910000085830688

2000/2000 [==============================] - 14s 7ms/step - loss: 0.5300 - accuracy: 0.8194 - val_loss: 0.5939 - val_accuracy: 0.7910
Epoch 29/30
1994/2000 [============================>.] - ETA: 0s - loss: 0.5003 - accuracy: 0.8283
End of epoch 29, LoraLayer 0: 58000 Step
End of epoch 29, LoraLayer 0: Decay factor: 0.0
End of epoch 29, LoraLayer 1: 58000 Step
End of epoch 29, LoraLayer 1: Decay factor: 0.0
End of epoch 29, LoraLayer 2: 58000 Step
End of epoch 29, LoraLayer 2: Decay factor: 0.0

Testing loss: 0.4545723795890808, acc: 0.840583324432373
```

```
2000/2000 [==============================] - 14s 7ms/step - loss: 0.5000 - accuracy: 0.8284 - val_loss: 0.4546 - val_accuracy: 0.8406
Epoch 30/30
2000/2000 [==============================] - ETA: 0s - loss: 0.4832 - accuracy: 0.8342
End of epoch 30, LoraLayer 0: 60000 Step
End of epoch 30, LoraLayer 0: Decay factor: 0.0
End of epoch 30, LoraLayer 1: 60000 Step
End of epoch 30, LoraLayer 1: Decay factor: 0.0
End of epoch 30, LoraLayer 2: 60000 Step
End of epoch 30, LoraLayer 2: Decay factor: 0.0

Testing loss: 0.4521879553794861, acc: 0.844083309173584

2000/2000 [==============================] - 14s 7ms/step - loss: 0.4832 - accuracy: 0.8342 - val_loss: 0.4522 - val_accuracy: 0.8441
<keras.src.callbacks.History at 0x78f63f56a2c0>
```

```
313/313 - 1s - loss: 0.3787 - accuracy: 0.8779 - 997ms/epoch - 3ms/step

Test accuracy: 0.8779000043869019
```

rank=64, weight B 초기화 조건:
정규분포

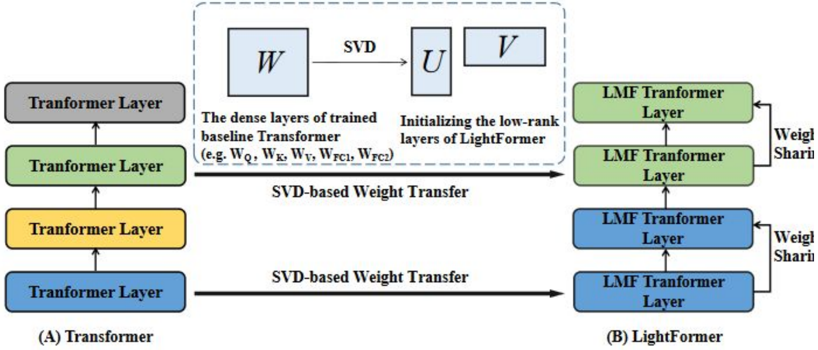
New Experiment

구현은 안됨. 그러나 다음 Experiment 일 예정.

Idea 1. Original Weights 를 특정 rank 로 Truncated SVD 를하고, SVD 로 잃어버린 Information 을 LoRA 에서 복원

Idea 2. LoRA BA 를 Original Weights 의 Truncated SVD 로 구성하고, 학습을 통해 잃어버린 Information 을 LoRA 에서 복원

참조 논문 : https://aclanthology.org/2023.findings-acl.656.pdf



Model	IWSLT'14 De-En				WMT'14 En-De			
	Params.	Ratio	Speed	BLEU	Params.	Ratio	Speed	BLEU
Transformer	36.8M	1.0×	1.0×	34.5	63.2M	1.0×	1.0×	27.3
Lite Transformer	13.9M	2.6×	1.5×	33.6	33.6M	1.9×	1.1×	26.5
HAT Transformer	28.2M	1.3×	1.7×	34.5	46.2M	1.4×	1.7×	26.9
DelighT	19.9M	1.8×	0.8 ×	34.4	23.3M	2.7×	1.2×	26.7
LightFormer	7.7M	4.8×	2.1×	34.6	22.5M	2.8×	1.5×	27.1
w/o SVD WT	7.7M	4.8×	2.1×	34.0	22.5M	2.8×	1.5×	26.5

진행 사항

- Weight Decay
- **Conv LoRA Layer**

Conv LoRA Layer

CNN 태스크에 LoRA layer 을 적용

Pytorch 기반의 ConvLoRA layer 을 Tensorflow 기반으로 conversion

실험 셋팅:

- dataset: CIFAR 10
- model: VGG 16
- task: image classification
- https://github.com/KwanHoo/ImageProcessing/blob/main/14.VGG16/VGG16.ipynb

Conv LoRA Layer

microsoft 의 LoRA layer 코드:

- in_channels = 입력 이미지의 채널 수 (int)
- out_channels = conv에 의해 생성된 채널 수 (int)
- kernel size = Size of the conv kernel (int or tuple)
- lora_A의 형태 =
  (r * kernel_size, in_channels * kernel_size)
- lora_B의 형태 =
  (out_channels//self.conv.groups*kernel_size,
  r*kernel_size)

```
def build(self, input_shape):
    # Ensure the original convolutional layer is built.
    #if not self.original_conv_layer.built:
    #    self.original_conv_layer.build(input_shape)

    # Calculate the shape for LoRA weights A and B.
    #self.kernel = self.original_conv_layer.kernel
    self.in_channels = input_shape[-1]

    in_channels = self.in_channels
    out_channels = self.filters
    kernel_size = self.original_conv_layer.kernel_size[0]

    # LoRA weights A and B.
    self.A_weight = self.add_weight(
        name="lora_A_weight",
        shape=(self.rank*kernel_size, in_channels*kernel_size),
        initializer=initializers.VarianceScaling(scale=1.0, mode='fan_in', distribution='uniform'),
        trainable=self.trainable
    )

    self.B_weight = self.add_weight(
        name="lora_B_weight",
        shape=(out_channels*kernel_size, self.rank*kernel_size),
        initializer=initializers.VarianceScaling(scale=1.0, mode='fan_in', distribution='uniform'),
        trainable=self.trainable
    )

    super().build(input_shape)
```

Conv LoRA Layer

```
def forward(self, x):
    if self.r > 0 and not self.merged:
        return self.conv._conv_forward(
            x,
            self.conv.weight + (self.lora_B @ self.lora_A).view(self.conv.weight.shape) * self.scaling,
            self.conv.bias
        )
    return self.conv(x)
```

↓

```
    # lora_BA의 형태 변환
    # lora_BA가 (out_channels*kernel_size*kernel_size, in_channels*kernel_size*kernel_size) 형태라고 가정
    # 이를 (kernel_size, kernel_size, in_channels, out_channels)로 변환
lora_BA_reshaped = tf.reshape(lora_BA, (out_channels, kernel_size, kernel_size, in_channels))
lora_BA_reshaped = tf.transpose(lora_BA_reshaped, [1, 2, 3, 0])
lora_output = tf.nn.conv2d(inputs, lora_BA_reshaped, strides=[1, 1, 1, 1], padding='SAME')
```

Conv LoRA Layer

tensorflow 형태의 conv lora layer 최종본

```
class ConvLoRALayer(layers.Layer):
    def __init__(
        self,
        original_conv_layer,
        rank=2,
        alpha=32,
        trainable=True,
        **kwargs
    ):
        # Capture the original layer's configuration.
        original_layer_config = original_conv_layer.get_config()
        name = original_layer_config["name"]
        kwargs.pop("name", None)

        super().__init__(name=name, trainable=trainable, **kwargs)

        self.rank = rank
        self.alpha = alpha
        self._scale = alpha / rank

        # The original convolutional layer is set to non-trainable to freeze its weights.
        self.original_conv_layer = original_conv_layer
        self.original_conv_layer.trainable = False

        self.kernel = None
        self.filters = original_conv_layer.filters #
        self.kernel_size = original_conv_layer.kernel_size[0] #
        self.in_channels = None
```

```
    def build(self, input_shape):
        # Ensure the original convolutional layer is built.
        #if not self.original_conv_layer.built:
        #    self.original_conv_layer.build(input_shape)

        # Calculate the shape for LoRA weights A and B.
        #self.kernel = self.original_conv_layer.kernel
        self.in_channels = input_shape[-1]

        in_channels = self.in_channels
        out_channels = self.filters
        kernel_size = self.original_conv_layer.kernel_size[0]

        # LoRA weights A and B.
        self.A_weight = self.add_weight(
            name="lora_A_weight",
            shape=(self.rank*kernel_size, in_channels*kernel_size),
            initializer=initializers.VarianceScaling(scale=1.0, mode='fan_in', distribution='uniform'),
            trainable=self.trainable
        )

        self.B_weight = self.add_weight(
            name="lora_B_weight",
            shape=(out_channels*kernel_size, self.rank*kernel_size),
            initializer=initializers.VarianceScaling(scale=1.0, mode='fan_in', distribution='uniform'),
            trainable=self.trainable
        )

        super().build(input_shape)
```

```
    def call(self, inputs, training=None):
        if training is None:
                training = self.trainable

        original_output = self.original_conv_layer(inputs)


        lora_BA = (self.B_weight@self.A_weight)

        kernel_size = self.original_conv_layer.kernel_size[0]
        in_channels = self.in_channels
        out_channels = self.filters

          # lora_BA의 형태 변환
          # lora_BA가 (out_channels*kernel_size*kernel_size, in_channels*kernel_size*kernel_size) 형태라고 가
          # 이를 (kernel_size, kernel_size, in_channels, out_channels)로 변환
        lora_BA_reshaped = tf.reshape(lora_BA, (out_channels, kernel_size, kernel_size, in_channels))
        lora_BA_reshaped = tf.transpose(lora_BA_reshaped, [1, 2, 3, 0])
        lora_output = tf.nn.conv2d(inputs, lora_BA_reshaped, strides=[1, 1, 1, 1], padding='SAME')
        lora_output /= 10

        if training:
                #return original_output
                return original_output + lora_output * self._scale

        else:
                # 추론 모드에서는 LoRA 출력만 반환
                return lora_output
```

Trouble Shooting

1. pytorch 와 달리 keras 에는 in_channels 와 out_channels 가 없기 때문에 같은 의미를 가진 변수를 찾았어야했음.
   처음에는 kernel_shape 변수를 사용하려 했으나 keras 에 kernel_shape 라는 변수가 없음을 알게 되어 input[-1] 과 filters 로 생성.

2. ConvLoRALayer 를 적용시킨 모델의 total params 가 각 layer 의 params 의 합과 값이 다르고 trainable params 도 값이 지나치게 큼.

   conv layer 의 weight 수 만큼만 증가되어있음을 확인.

   ‘__init__’ 메서드에서 불필요하게 self.kernel 변수를 생성한 것이 원인.

3. 학습시킨 모델의 가중치를 복사한 모델에 convlora 를 적용시키면 val_acc 가 과하게 낮게 나옴.