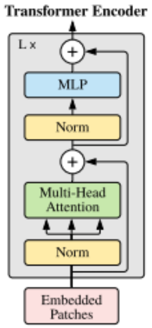
## Vit encoder 구조에 Bert encoder 구조를 참고하여 Layer Normalization 단계 추가

0. 자연어 처리 모델에서 사용하는 Transformer를 이미지 분류에 적용시킨 Vision Transformer에 관한 논문입니다.

1. 기존 Vit Transformer Encoder의 구조는 Layer Normalize 과정이 Multi-Head Attention 이전, MLP 이전에 두 번 진행된다.
   Transformer Encoder의 내부구조는 다음과 같다.

[내부구조]. Transformer Encoder 내부에서는 먼저 Layer Normalization을 거치고, Multi-Head Attention을 지난 결과를 통과하지 않은 패치와 Skip Connection 시켜줍니다. 그리고 다시 Layer Nomalization, MLP를 거쳐 Skip Connection으로 다시 더해주는 것이 한 번 Transformer Encoder를 통과한 것입니다. 이러한 Transformer Encoder를 L번 반복합니다.

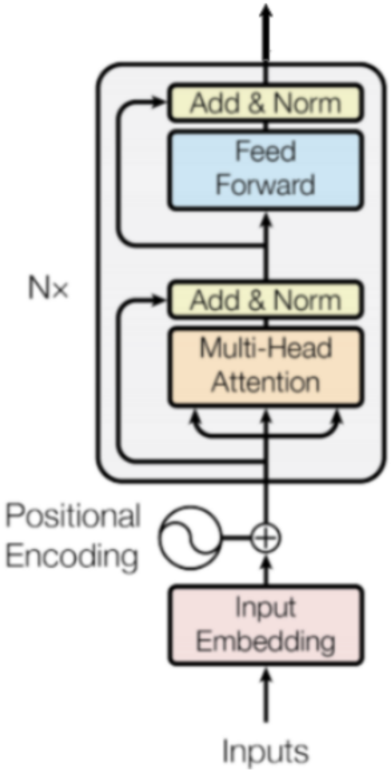



[내부구조 수식]

$$\mathbf{z}'_\ell = \mathrm{MSA}(\mathrm{LN}(\mathbf{z}_{\ell-1})) + \mathbf{z}_{\ell-1},$$
$$\mathbf{z}_\ell = \mathrm{MLP}(\mathrm{LN}(\mathbf{z}'_\ell)) + \mathbf{z}'_\ell,$$

이러한 과정을 L번 반복합니다.

3. Bert의 Transformer Encoder 구조를 차용하여 성능향상을 기대한다.




Bert의 Transformer Encoder는 output을 추가로 Linear Norm 시켜주는데 이를 기존 Transformer 에 적용한다.

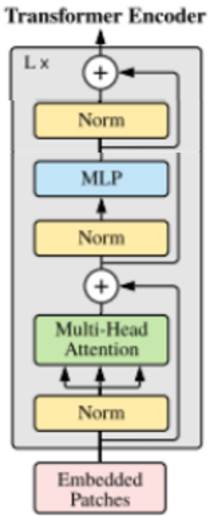
4. 기존의 Transformer Encoder의 구조는 다음과 같다.

```
1) Norm
2) Multi-Head Attention
3) Skip Connection
4) Norm
5) MLP
6) Skip Connection
```

```
x1 = layers.LayerNormalizatio(epsilon=1e-6)(encoded_patches)

attention_output = layers.MultiHeadAttention(
num_heads = num_heads, key_dim = projection_dim, dropout=0.1)(x1, x1)

x2 = layers.Add()([attention_output, encoded_patches])

x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
```

```
x3 = mlp(x3, hidden_units = transformer_units, dropout_rate = 0.1)

encoded_patches = layers.Add()([x3, x2])
```

5. Bert의 구조와 유사하게 MLP 이후에 Linear Normalization을 추가한 Transformer Encoder의 구조는 다음과 같다.




```
 1) Norm
 2) Multi-Head Attention
 3) Skip Connection
 4) Norm
 5) MLP
 6) Norm (Added)
 7) Skip Connection
```

```
x1 = layers.LayerNormalizatio(epsilon=1e-6)(encoded_patches)

attention_output = layers.MultiHeadAttention(
num_heads = num_heads, key_dim = projection_dim, dropout=0.1)(x1, x1)

x2 = layers.Add()([attention_output, encoded_patches])

x3 = layers.LayerNormalization(epsilon=1e-6)(x2)

x3 = mlp(x3, hidden_units = transformer_units, dropout_rate = 0.1)
```

```
x3 = layers.LayerNormalization(epsilon=1e-6)(x3)  (Added)

encoded_patches = layers.Add()([x3, x2])
```

6. 결과분석

기존 Vit 학습 결과 (before)

```
Epoch 1/20
176/176 [==============================] - 60s 216ms/step - loss: 4.3106 - accuracy: 0.0631 - top-5-accuracy: 0.2060
Epoch 2/20
176/176 [==============================] - 33s 188ms/step - loss: 3.7281 - accuracy: 0.1287 - top-5-accuracy: 0.3596
Epoch 3/20
176/176 [==============================] - 33s 185ms/step - loss: 3.4607 - accuracy: 0.1699 - top-5-accuracy: 0.4338
Epoch 4/20
176/176 [==============================] - 32s 182ms/step - loss: 3.3109 - accuracy: 0.1956 - top-5-accuracy: 0.4742
Epoch 5/20
176/176 [==============================] - 33s 189ms/step - loss: 3.1939 - accuracy: 0.2187 - top-5-accuracy: 0.5092
Epoch 6/20
176/176 [==============================] - 32s 184ms/step - loss: 3.1016 - accuracy: 0.2351 - top-5-accuracy: 0.5311
Epoch 7/20
176/176 [==============================] - 32s 184ms/step - loss: 3.0281 - accuracy: 0.2503 - top-5-accuracy: 0.5501
Epoch 8/20
176/176 [==============================] - 32s 183ms/step - loss: 2.9734 - accuracy: 0.2596 - top-5-accuracy: 0.5622
Epoch 9/20
176/176 [==============================] - 32s 185ms/step - loss: 2.9308 - accuracy: 0.2665 - top-5-accuracy: 0.5709
Epoch 10/20
176/176 [==============================] - 32s 183ms/step - loss: 2.8870 - accuracy: 0.2774 - top-5-accuracy: 0.5849

Epoch 11/20
176/176 [==============================] - 32s 183ms/step - loss: 2.8583 - accuracy: 0.2836 - top-5-accuracy: 0.5903
Epoch 12/20
176/176 [==============================] - 33s 185ms/step - loss: 2.8295 - accuracy: 0.2873 - top-5-accuracy: 0.5960
Epoch 13/20
176/176 [==============================] - 32s 183ms/step - loss: 2.8127 - accuracy: 0.2912 - top-5-accuracy: 0.6038
Epoch 14/20
176/176 [==============================] - 32s 183ms/step - loss: 2.7896 - accuracy: 0.2959 - top-5-accuracy: 0.6042
Epoch 15/20
176/176 [==============================] - 32s 184ms/step - loss: 2.7696 - accuracy: 0.2980 - top-5-accuracy: 0.6120
Epoch 16/20
176/176 [==============================] - 32s 183ms/step - loss: 2.7486 - accuracy: 0.3039 - top-5-accuracy: 0.6165
Epoch 17/20
176/176 [==============================] - 32s 183ms/step - loss: 2.7359 - accuracy: 0.3036 - top-5-accuracy: 0.6178
Epoch 18/20
176/176 [==============================] - 32s 183ms/step - loss: 2.7235 - accuracy: 0.3089 - top-5-accuracy: 0.6210
Epoch 19/20
176/176 [==============================] - 32s 183ms/step - loss: 2.7062 - accuracy: 0.3130 - top-5-accuracy: 0.6265
Epoch 20/20
176/176 [==============================] - 32s 182ms/step - loss: 2.6868 - accuracy: 0.3128 - top-5-accuracy: 0.6306
```

20epochs 이후 loss가 2.6868, accuracy가 0.3128 이다.

Bert의 Transformer Encoder 구조를 차용해 Linear Normalization을 추가한 Vit의 학습 결과 (after)

```
Epoch 1/20
176/176 [==============================] - 56s 192ms/step - loss: 4.3285 - accuracy: 0.0577 - top-5-accuracy: 0.1935
Epoch 2/20
176/176 [==============================] - 34s 195ms/step - loss: 3.7324 - accuracy: 0.1262 - top-5-accuracy: 0.3546
Epoch 3/20
176/176 [==============================] - 33s 186ms/step - loss: 3.4427 - accuracy: 0.1716 - top-5-accuracy: 0.4401
Epoch 4/20
176/176 [==============================] - 33s 187ms/step - loss: 3.2517 - accuracy: 0.2058 - top-5-accuracy: 0.4920
Epoch 5/20
176/176 [==============================] - 33s 186ms/step - loss: 3.1227 - accuracy: 0.2318 - top-5-accuracy: 0.5251
Epoch 6/20
176/176 [==============================] - 33s 185ms/step - loss: 3.0134 - accuracy: 0.2499 - top-5-accuracy: 0.5533
Epoch 7/20
176/176 [==============================] - 33s 186ms/step - loss: 2.9072 - accuracy: 0.2734 - top-5-accuracy: 0.5798
Epoch 8/20
176/176 [==============================] - 33s 186ms/step - loss: 2.8150 - accuracy: 0.2905 - top-5-accuracy: 0.6021
Epoch 9/20
176/176 [==============================] - 33s 185ms/step - loss: 2.7570 - accuracy: 0.3015 - top-5-accuracy: 0.6156
Epoch 10/20
176/176 [==============================] - 34s 194ms/step - loss: 2.6972 - accuracy: 0.3149 - top-5-accuracy: 0.6267
Epoch 11/20
176/176 [==============================] - 33s 185ms/step - loss: 2.6449 - accuracy: 0.3213 - top-5-accuracy: 0.6444
Epoch 12/20
176/176 [==============================] - 33s 187ms/step - loss: 2.6097 - accuracy: 0.3298 - top-5-accuracy: 0.6497
Epoch 13/20
176/176 [==============================] - 33s 187ms/step - loss: 2.5737 - accuracy: 0.3390 - top-5-accuracy: 0.6576
Epoch 14/20
176/176 [==============================] - 33s 185ms/step - loss: 2.5307 - accuracy: 0.3453 - top-5-accuracy: 0.6657
Epoch 15/20
176/176 [==============================] - 33s 187ms/step - loss: 2.4990 - accuracy: 0.3521 - top-5-accuracy: 0.6754
Epoch 16/20
176/176 [==============================] - 33s 185ms/step - loss: 2.4740 - accuracy: 0.3573 - top-5-accuracy: 0.6812
Epoch 17/20
176/176 [==============================] - 33s 186ms/step - loss: 2.4490 - accuracy: 0.3629 - top-5-accuracy: 0.6830
Epoch 18/20
176/176 [==============================] - 33s 187ms/step - loss: 2.4269 - accuracy: 0.3670 - top-5-accuracy: 0.6885
Epoch 19/20
176/176 [==============================] - 33s 185ms/step - loss: 2.4191 - accuracy: 0.3696 - top-5-accuracy: 0.6906
Epoch 20/20
176/176 [==============================] - 33s 186ms/step - loss: 2.3949 - accuracy: 0.3728 - top-5-accuracy: 0.6958
```

20epochs 이후 loss가 2.3949, accuracy가 0.3728 이다.

loss가 0.2919 감소, accuracy가 0.06로 소폭 증가했다.

## 구현

구현할 모델에서는 Layer의 개수를 12개, D의 크기를 64, MLP의 크기를 1024, Head 개수를 4로 설정하여 진행한다.

```
# tensorflow_addons를 사용하기 위해 설치해줘야 한다.
%pip install tensorflow_addons
```

```
Collecting tensorflow_addons
  Downloading tensorflow_addons-0.23.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (611 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 611.8/611.8 kB 8.3 MB/s eta 0:00:00
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from tensorflow_addons) (23.2)
Collecting typeguard<3.0.0,>=2.7 (from tensorflow_addons)
  Downloading typeguard-2.13.3-py3-none-any.whl (17 kB)
Installing collected packages: typeguard, tensorflow_addons
Successfully installed tensorflow_addons-0.23.0 typeguard-2.13.3
```

```
import numpy as np
import tensorflow as tf
import keras
from keras import layers          # vision transformer를 구성하는데 사용 (vit함수)
import tensorflow_addons as tfa
import matplotlib.pyplot as plt
```

```
print(tf.__version__)   # tensorflow 2.14.0 버전을 사용한다.
```

```
2.14.0
```

데이터는 cifar100을 사용한다. 32 x 32 크기의 60000개의 이미지로 이루어져 있으며, 100개의 클래스로 분류(dolphin, fish ...) 되며 각각의 클래스는 600개의 이미지로 이루어져 있다. 또, 500개는 학습 데이터, 100개는 데이터 데이터로 이루어져 있어 총 50000개의 학습 데이터, 10000개의 테스트 데이터로 이루어져있다.

```
#  0 사용하므로 class를 100개로 지정
input_shape = (32,32,3)   # input shape는 32x32의 RGB 채널을 가진 이미지이다.

# 데이터 로드 (train과 test를 나눠서 로드한다)
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar100.load_data()

print(f"x_train shape : {x_train.shape}, y_train shape : {y_train.shape}")
print(f"x_test shape : {x_test.shape}, y_test shape : {y_test.shape}")
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz
169001437/169001437 [==============================] - 3s 0us/step
x_train shape : (50000, 32, 32, 3), y_train shape : (50000, 1)
x_test shape : (10000, 32, 32, 3), y_test shape : (10000, 1)
```

```
batch_size = 256

image_size = 224    # 16x16 이미지를 업사이징하여 224x224로 만들것이다.
patch_size = 32     # 패치 사이즈는 32x32,
num_patches = (image_size//patch_size)**2     # 패치의 개수는 이미지 크기를 패치사이즈로 나누고 제곱한다.

# D 차원으로 벡터화
```

```
projection_dim = 64   # D = 64
num_heads = 4

# mlp에서 사용하는 transformer unit (128, 64)
transformer_units = [
    projection_dim*2,
    projection_dim,
]

transformer_layers = 12        # layer의 개수
mlp_head_units = [2048, 1024]


# 이미지 업사이징, 전처리
data_augmentation = keras.Sequential(
    [
            layers.Normalization(),                        # Normalize
            layers.Resizing(image_size, image_size),       # 224 x 224
            layers.RandomFlip('horizontal'),
            layers.RandomRotation(factor=0.02),
            layers.RandomZoom(height_factor=0.2, width_factor=0.2),
    ],
    name = 'data_augmentation',
)

data_augmentation.layers[0].adapt(x_train)


# mlp 함수
def mlp(x, hidden_units, dropout_rate):
  for units in hidden_units:
      x = layers.Dense(units, activation=tf.nn.gelu)(x)   # 활성화함수로 gelu를 사용
      x = layers.Dropout(dropout_rate)(x)                 # dropout을 사용한다.
  return x


# 패치화하는 클래스
class Patches(layers.Layer):
  def __init__(self, patch_size):
    super().__init__()
    self.patch_size = patch_size

  def call(self, images):
    batch_size = tf.shape(images)[0]
    patches = tf.image.extract_patches(
        images = images,
        sizes = [1, self.patch_size, self.patch_size, 1],
        strides = [1, self.patch_size, self.patch_size, 1],
        rates = [1,1,1,1],
        padding = "VALID",    # padding 사용 X
    )
    patch_dims = patches.shape[-1]
    patches = tf.reshape(patches, [batch_size, -1, patch_dims])
    return patches
```

```
# PatchEncoder를 class로 정의
# (Linear Projection -> Position Embedding)
class PatchEncoder(layers.Layer):
  def __init__(self, num_patches, projection_dim):
    super().__init__()
    self.num_patches = num_patches

    self.projection = layers.Dense(units=projection_dim)      # D차원으로 Linear Projection
    self.position_embedding = layers.Embedding(               # position embedding
        input_dim = num_patches, output_dim = projection_dim
    )

  def call(self, patch):
    positions = tf.range(start=0, limit=self.num_patches, delta=1)         # 0부터 patch개수만큼 1씩 증가하는 position
    encoded = self.projection(patch) + self.position_embedding(positions)   # position embedding 과정
    return encoded  # z0


# vision transformer
def vit():

  # 1) Patch화 ->  patch를 Linear Projection -> Position Embedding
  inputs = layers.Input(shape=input_shape)
  augmented = data_augmentation(inputs)     # inputs를 업사이징
  patches = Patches(patch_size)(augmented)  # patch 생성 (patches)
  encoded_patches = PatchEncoder(num_patches, projection_dim)(patches) # patch를 Linear Projection -> Position Embedding


  # 2) Transformer Encoder L번 반복
  for _ in range(transformer_layers):   # L번 반복

    x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)       # 2) Layer Normalize

    attention_output = layers.MultiHeadAttention(                       # 1) Multi-Head Attention
        num_heads = num_heads, key_dim = projection_dim, dropout=0.1
    )(x1, x1)

    x2 = layers.Add()([attention_output, encoded_patches])              # 3) Skip Connection


    x3 = mlp(x2, hidden_units = transformer_units, dropout_rate = 0.1)  # 5) MLP

    x3 = layers.LayerNormalization(epsilon=1e-6)(x3)                    # 4) Layer Normalize (Added)


    encoded_patches = layers.Add()([x3, x2])                            # 6) Skip Connection



  # 3) MLP Head에 들어가기 전 레이어정규화
  representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
  representation = layers.Flatten()(representation)
  representation = layers.Dropout(0.5)(representation)

  # 4) MLP Head
  features = mlp(representation, hidden_units = mlp_head_units, dropout_rate = 0.5)
```

```
    # 5) class화
    logits = layers.Dense(100)(features)

    # 6) 모델 생성
    model = keras.Model(inputs=inputs, outputs = logits)

    return model


model = vit()
model.summary()
```

```
 dropout_26 (Dropout)       (None, 1024)              0         ['dense_26[0][0]']

 dense_27 (Dense)           (None, 100)               102500    ['dropout_26[0][0]']

==================================================================================================
Total params: 9823595 (37.47 MB)
Trainable params: 9823588 (37.47 MB)
Non-trainable params: 7 (32.00 Byte)
```

```
# 학습 / 테스트
num_epochs = 20
weight_decay = 0.001
learning_rate = 0.001

optimizer = tfa.optimizers.AdamW(
    learning_rate = learning_rate, weight_decay = weight_decay
)

model.compile(
    optimizer = optimizer,
    loss = keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[
        keras.metrics.SparseCategoricalAccuracy(name='accuracy'),
        keras.metrics.SparseTopKCategoricalAccuracy(5, name='top-5-accuracy'),
    ],
)

history = model.fit(
    x=x_train,
    y=y_train,
    batch_size = batch_size,
    epochs = num_epochs,
    validation_split = 0.1,
)
```

```
Epoch 1/20
176/176 [==============================] - 61s 197ms/step - loss: 4.3012 - accuracy: 0.0627 - top-5-accuracy: 0.2051 - val_loss: 3.7173 - val_accuracy: 0.1368 - val_top-5-accuracy: 0.3606
Epoch 2/20
176/176 [==============================] - 33s 189ms/step - loss: 3.7138 - accuracy: 0.1282 - top-5-accuracy: 0.3617 - val_loss: 3.3825 - val_accuracy: 0.1816 - val_top-5-accuracy: 0.4548
Epoch 3/20
176/176 [==============================] - 33s 189ms/step - loss: 3.4305 - accuracy: 0.1764 - top-5-accuracy: 0.4435 - val_loss: 3.1477 - val_accuracy: 0.2294 - val_top-5-accuracy: 0.5142
Epoch 4/20
176/176 [==============================] - 33s 186ms/step - loss: 3.2467 - accuracy: 0.2075 - top-5-accuracy: 0.4914 - val_loss: 2.9985 - val_accuracy: 0.2540 - val_top-5-accuracy: 0.5536
Epoch 5/20
176/176 [==============================] - 34s 194ms/step - loss: 3.1073 - accuracy: 0.2352 - top-5-accuracy: 0.5307 - val_loss: 2.9019 - val_accuracy: 0.2746 - val_top-5-accuracy: 0.5716
Epoch 6/20
176/176 [==============================] - 33s 186ms/step - loss: 2.9879 - accuracy: 0.2576 - top-5-accuracy: 0.5603 - val_loss: 2.7953 - val_accuracy: 0.2956 - val_top-5-accuracy: 0.6022
Epoch 7/20
176/176 [==============================] - 33s 186ms/step - loss: 2.8868 - accuracy: 0.2775 - top-5-accuracy: 0.5846 - val_loss: 2.7012 - val_accuracy: 0.3166 - val_top-5-accuracy: 0.6260
Epoch 8/20
176/176 [==============================] - 33s 186ms/step - loss: 2.7996 - accuracy: 0.2925 - top-5-accuracy: 0.6035 - val_loss: 2.6245 - val_accuracy: 0.3240 - val_top-5-accuracy: 0.6448
Epoch 9/20
176/176 [==============================] - 33s 185ms/step - loss: 2.7471 - accuracy: 0.3037 - top-5-accuracy: 0.6160 - val_loss: 2.6328 - val_accuracy: 0.3324 - val_top-5-accuracy: 0.6404
Epoch 10/20
176/176 [==============================] - 33s 188ms/step - loss: 2.6921 - accuracy: 0.3154 - top-5-accuracy: 0.6324 - val_loss: 2.5382 - val_accuracy: 0.3504 - val_top-5-accuracy: 0.6624
Epoch 11/20
176/176 [==============================] - 33s 185ms/step - loss: 2.6496 - accuracy: 0.3237 - top-5-accuracy: 0.6422 - val_loss: 2.6040 - val_accuracy: 0.3354 - val_top-5-accuracy: 0.6450
Epoch 12/20
```

```
176/176 [==============================] - 34s 193ms/step - loss: 2.6040 - accuracy: 0.3319 - top-5-accuracy: 0.6513 - val_loss: 2.4604 - val_accuracy: 0.3724 - val_top-5-accuracy: 0.6832
Epoch 13/20
176/176 [==============================] - 35s 197ms/step - loss: 2.5692 - accuracy: 0.3393 - top-5-accuracy: 0.6589 - val_loss: 2.4418 - val_accuracy: 0.3720 - val_top-5-accuracy: 0.6810
Epoch 14/20
176/176 [==============================] - 33s 187ms/step - loss: 2.5498 - accuracy: 0.3420 - top-5-accuracy: 0.6626 - val_loss: 2.4450 - val_accuracy: 0.3676 - val_top-5-accuracy: 0.6844
Epoch 15/20
176/176 [==============================] - 33s 187ms/step - loss: 2.5114 - accuracy: 0.3505 - top-5-accuracy: 0.6704 - val_loss: 2.4393 - val_accuracy: 0.3760 - val_top-5-accuracy: 0.6876
Epoch 16/20
176/176 [==============================] - 34s 194ms/step - loss: 2.4859 - accuracy: 0.3556 - top-5-accuracy: 0.6784 - val_loss: 2.3791 - val_accuracy: 0.3850 - val_top-5-accuracy: 0.6996
Epoch 17/20
176/176 [==============================] - 33s 186ms/step - loss: 2.4630 - accuracy: 0.3624 - top-5-accuracy: 0.6844 - val_loss: 2.3875 - val_accuracy: 0.3830 - val_top-5-accuracy: 0.6978
Epoch 18/20
176/176 [==============================] - 33s 186ms/step - loss: 2.4413 - accuracy: 0.3666 - top-5-accuracy: 0.6858 - val_loss: 2.3614 - val_accuracy: 0.3846 - val_top-5-accuracy: 0.6978
Epoch 19/20
176/176 [==============================] - 33s 188ms/step - loss: 2.4102 - accuracy: 0.3711 - top-5-accuracy: 0.6929 - val_loss: 2.3207 - val_accuracy: 0.4014 - val_top-5-accuracy: 0.7104
Epoch 20/20
176/176 [==============================] - 33s 185ms/step - loss: 2.4049 - accuracy: 0.3738 - top-5-accuracy: 0.6931 - val_loss: 2.2921 - val_accuracy: 0.3984 - val_top-5-accuracy: 0.7212
```

```
plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
plt.plot(history.history['loss'], 'b--', label='loss')
plt.plot(history.history['val_loss'], 'r-', label='val_loss')
plt.xlabel('Epochs')
plt.grid()
plt.legend()

plt.subplot(1,2,2)
plt.plot(history.history['accuracy'], 'b--', label='accuracy')
plt.plot(history.history['val_accuracy'], 'r-', label='val_accuracy')
plt.xlabel('Epochs')
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7ba9e4ff07c0>
```