

شبکه های عمیق کانولوشنی قادرند کار های بی سابقه ای را در حوزه پردازش تصویر انجام دهند ولی قابلیت آن ها در کار با چرخش های محلی و سراسری تصویر همچنان محدودیت دارد.

در این مقاله فیلتر های چرخان فعال (ARF) معرفی میشوند که به طور فعال در حین عمل کانولوشن میچرخدند و میتوانند ویژگی های چرخشی و مکانی را به طور صریح به دست آورند و در یک نقشه نگاشت کنند.

ARF ها مانند یک بانک فیلتر مجازی عمل میکنند که از فیلتر اصلی و نسخه های مجازی چرخش داده آن تشکیل شده اند. یک ARF فیلتری با ابعاد $N \times N \times W \times W$ است که به طور فعال در حین کانولوشن گیری به تعداد $1 - N$ بار میچرخد تا یک نقشه ویژگی با تعداد N کanal جهت ایجاد کند.

در نتیجه یک ARF را میتوان به یک بانک از N فیلتر با سایز $(W \times W \times N)$ تعبیر کرد که فقط فیلتر کانونی آن واقعی است و آموزش داده میشود، سایر نسخه ها صرفا چرخش یافته فیلتر کانونی هستند.

فیلتر n ام، $[1, N - 1]$ در این بانک فیلتر از چرخش دادن فیلتر کانونی به اندازه $\frac{2\pi n}{N}$ به دست می آید. یک ARF از N کanal جهت تشکیل شده و به صورت نقاط N جهته در یک شبکه $W \times W$ مشاهده میشود.

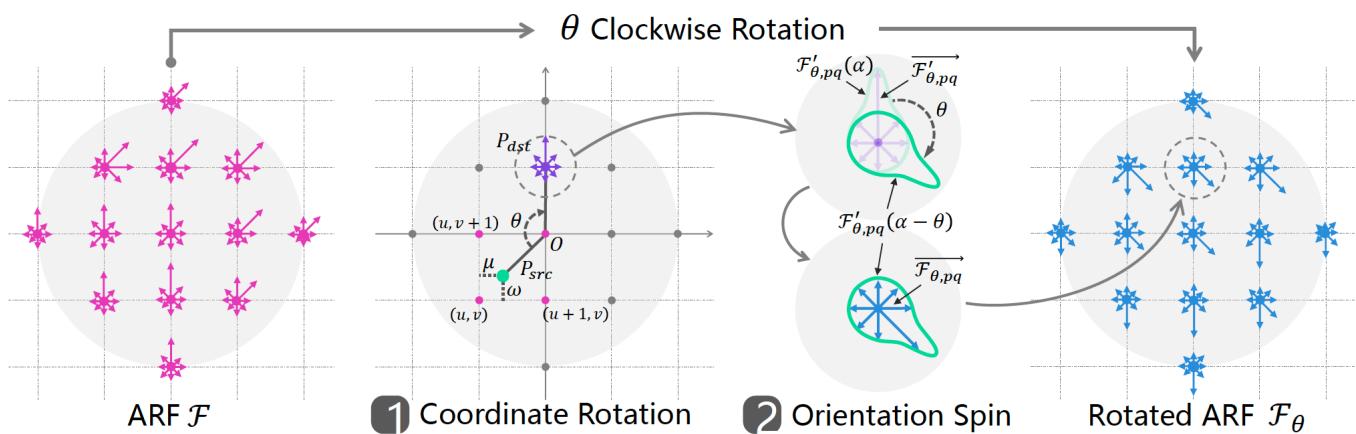
توضیح بیشتر (برداشت شخصی):

هر کدام از لایه های بعد سوم این فیلتر $N \times W \times W$ ، نماینده یک جهت خاص است هر کدام از این لایه ها یک شبکه $W \times W$ است و نقاط آن شبکه نمایانگر مکان های مختلف تصویر هستند.

برای درک بیشتر فرض کنیم لایه θ درجه را انتخاب کنیم و مشاهده کنیم در موقعیت (x, y) مقدار t را داریم، این به این معناست که در موقعیت (x, y) مولفه های θ درجه و با شدت t مشاهده شده اند.

حال برای یادگیری مولفه های این فیلتر سه بعدی، آن را $N-1$ بار چرخش میدهیم و سپس مقادیر پارامتر های فیلتر را به روز رسانی میکنیم.

برای چرخاندن مانند شکل زیر عمل میکنیم که شامل دو مرحله است:



1 coordinate rotation- چرخش مختصات

فیلتر ARF را حول نقطه مبدأ O میچرخانیم و نقطه (p, q) در فیلتر چرخش یافته از رابطه زیر بر حسب چهار همسایه نقطه متناظرش (p', q') در فیلتر اولیه محاسبه میکنیم.

$$\begin{aligned} \overrightarrow{\mathcal{F}'_{\theta,pq}} &= (1-\mu)(1-\omega)\overrightarrow{\mathcal{F}_{uv}} + (1-\mu)\omega\overrightarrow{\mathcal{F}_{u,v+1}} \\ &\quad + \mu(1-\omega)\overrightarrow{\mathcal{F}_{u+1,v}} + \mu\omega\overrightarrow{\mathcal{F}_{u+1,v+1}}, \end{aligned} \quad (1)$$

$$u = \lfloor p' \rfloor, v = \lfloor q' \rfloor, \mu = p' - u, \omega = q' - v$$

چرخش جهت (orientation spin-2)

پس از چرخاندن مختصات یک چرخش ساعتگرد به اندازه θ برای رسیدن به $F_{\theta,pq}$ نیاز است. این مرحله در واقع quantize کردن $(\alpha - \theta) F'_{\theta,pq}$ است که با شیفت چرخشی در فضای فرکانسی انجام می‌گیرد.

$$\begin{aligned} X(k) &\equiv \mathbf{DFT}\{\overrightarrow{\mathcal{F}'_{\theta,pq}}^{(n)}\} \\ &= \sum_{n=0}^{N-1} \overrightarrow{\mathcal{F}'_{\theta,pq}}^{(n)} e^{-jk\frac{2\pi n}{N}}, \quad k=0,1,\dots,N-1, \end{aligned} \quad (2)$$

$$\begin{aligned} \overrightarrow{\mathcal{F}_{\theta,pq}}^{(n)} &\equiv \mathbf{IDFT}\{X(k)e^{-jk\theta}\} \\ &= \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{jk(\frac{2\pi n}{N} - \theta)}, \quad n=0,1,\dots,N-1. \end{aligned} \quad (3)$$

برای مثال یک ARF با ابعاد $8 \times 3 \times 3$ را در نظر بگیرید، نسخه چرخش ساعتگرد آن به اندازه θ از رابطه زیر به دست می‌آید:

$$\begin{aligned} \overrightarrow{\hat{\mathcal{F}}'_{\theta,\langle i \rangle}} &= \overrightarrow{\hat{\mathcal{F}}'_{\langle (i-k) \bmod N \rangle}}, \quad i \in \mathcal{I}, \\ \overrightarrow{\hat{\mathcal{F}}_\theta}^{(n)} &= \overrightarrow{\hat{\mathcal{F}}'_\theta}^{((n-k) \bmod N)}, \quad n=0,1,\dots,N-1, \end{aligned} \quad (4)$$

در این رابطه k عضو اعداد طبیعی است، $\theta = k \frac{2\pi}{N}$ و $N=8$. یک جدول نگاشت است که اندیس المان های پیرامون را نشان میدهد.

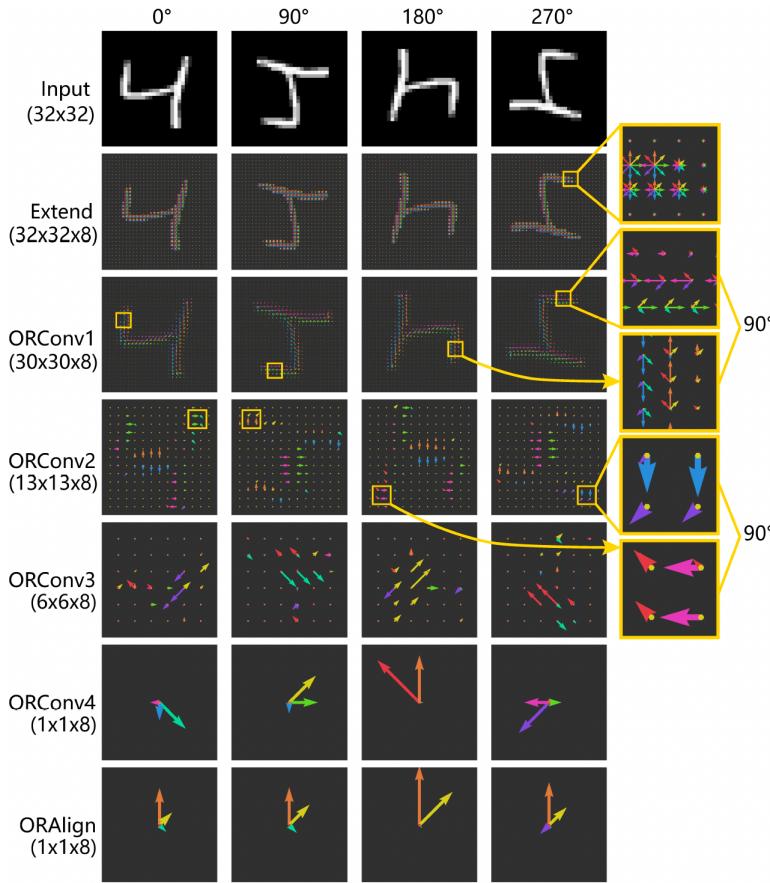
کانولوشن با استفاده از فیلتر F و یک نقشه ویژگی با M به صورت زیر نمایش داده می‌شود و آن را یک Oriented Response Convolution روی F (فیلتر ARF) و M (نقشه ویژگی) مینامیم.

$$\tilde{\mathcal{M}} = ORConv(\mathcal{F}, \mathcal{M})$$

نتیجه این کانولوشن نیز از N کanal تشکیل شده و کanal k ام آن را از رابطه زیر میتوان به دست آورد:

$$\tilde{\mathcal{M}}^{(k)} = \sum_{n=0}^{N-1} \mathcal{F}_{\theta_k}^{(n)} * \mathcal{M}^{(n)}, \quad \theta_k = k \frac{2\pi}{N}, \quad k=0,\dots,N-1, \quad (5)$$

حال عملکرد این کانولوشن را بررسی می‌کنیم:



نمونه نقشه ویژگی تولید شده توسط یک ARF روی هریک از لایه های ORN و آموزش داده شده روی دیتاست MNIST چرخش یافته.

هر ردیف نشان دهنده یک لایه از شبکه و هر ستون نشان دهنده یک داده ورودی است.

راستی ترین ستون ناحیه هایی از نقشه ویژگی را بزرگنمایی کرده است و به طور واضح مشخص است که نقشه ویژگی توانسته مشخصات جهتی و مکانی را یاد بگیرد.

در لایه دوم تصویر به یک نقشه همه جهته تبدیل شده تا بتواند برای لایه های ORConv fit شود.

در لایه های دوم تا آخر ویژگی های عمیق با مقادیر مشابه ولی در جهات مختلف مشاهده میشود که نشان میدهد که ویژگی ها جهت به خوبی توسط ORN ها استخراج شده.

در لایه آخر نیز تراز کردنی شبیه SIFT انجام داده تا تغییرناپذیری چرخشی به دست آید.

هنگام backpropagation مشتق خطای آموزشی را نسبت به تمام نسخه های دوران یافته محاسبه میکنیم و سپس آن ها را جمع میکنیم:

$$\delta^{(k)} = \frac{\partial L}{\partial \mathcal{F}_{\theta_k}}, \theta_k = k \frac{2\pi}{N}, k=0,1,\dots,N-1,$$

$$\mathcal{F} \leftarrow \mathcal{F} - \eta \sum_0^{N-1} \delta_{-\theta_k}^{(k)}, \quad (6)$$

به منظور orientation_invariant کردن نتیجه میتوان خروجی ORN را به یک لایه ORPooling یا ORAlign داد.

در لایه‌ی ORAlign میتوان از alignment SIFT-like استفاده کرد تا یک خروجی orientation_invariant حاصل شود.

روش ORPooling در واقع بیشینه جهت‌های خروجی نهایی انتخاب میکند (چون خروجی ORN دارای رزولوشن $1 \times N$ است).

فرمول محاسبه هیستوگرام قابل آموزش به صورت زیر است.

$$\psi_{k,b}(x_k) = \max \left\{ 0, 1 - \frac{|x_k - \mu_{k,b}|}{w_{k,b}} \right\}$$

برای پیاده سازی آن هر قسمت را به صورت یک لایه اجرا میکنیم

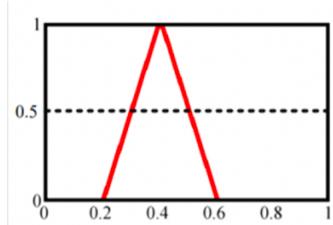
ابتدا با استفاده از یک لایه کانولوشنی با وزن ثابت ۱ و بایاس $\mu_{k,b} - \mu_k$ عبارت $x_k - \mu_{k,b}$ را محاسبه میکنیم. با استفاده از لایه قدر مطلق، از خروجی لایه قبل قدر مطلق میگیریم.

برای محاسبه عبارت $1 - \frac{|x_k - \mu_{k,b}|}{w_{k,b}}$ از یک لایه کانولوشنی با وزن $\frac{1}{w_{k,b}}$ و بایاس ۱ استفاده میکنیم.

در نهایت با استفاده از لایه *Relu*. بین صفر و عبارتی که محاسبه کردیم ماکزیمم میگیریم.

برای محاسبه مقدار هر *bin* از یک لایه *Global Average pooling* تا مقادیر متناظر با هر *bin* را جمع بزنند.

هیستوگرام در حالت عادی به دلیل مشتق پذیر نبودن قابل استفاده در شبکه های عصبی نیست به همین دلیل تعریف جدیدی از هیستوگرام ارائه میدهیم که در آن از یکتابع مثلثی شکل که مشتق پذیر است برای محاسبه مقادیر *bin* استفاده میکنیم.



در لایه کانولوشنی اول وزن ها ثابت هستند و بایاس ها قابل آموزش، ۶ عدد *bin* داریم پس در این لایه ۶ پارامتر قابل آموزش داریم که مربوط به وزن *bin* هاست

در لایه کانولوشنی دوم بایاس ها ثابت هستند و وزن ها قابل آموزش، ۶ عدد *bin* داریم پس در این لایه نیز ۶ پارامتر قابل آموزش داریم که مربوط به بایاس *bin* هاست پس در مجموع ۱۲ پارامتر قابل آموزش داریم.

```

class MyModel(tf.keras.Model):

    def __init__(self):
        super().__init__()
        #define your network layers here
        self.conv2D_1 = layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 1))
        self.conv2D_2 = layers.Conv2D(64, (3, 3), activation='relu')
        self.conv2D_3 = layers.Conv2D(64, (3, 3), activation='relu')
        self.maxPool_1 = layers.MaxPooling2D((2, 2))
        self.maxPool_2 = layers.MaxPooling2D((2, 2))
        self.flatten = layers.Flatten()
        self.dropOut = layers.Dropout(0.5)
        self.dense = layers.Dense(64, activation='relu')
        self.outputs = layers.Dense(10, activation='softmax')

    def call(self, inputs, training=False):
        #pass forward the inputs then return the output
        x = self.conv2D_1(inputs)
        x = self.maxPool_1(x)
        x = self.conv2D_2(x)
        x = self.maxPool_2(x)
        x = self.conv2D_3(x)
        x = self.flatten(x)
        x = self.dense(x)
        x = self.dropOut(x)

        # return output
        return self.outputs(x)

```

در قسمت init لایه های مختلف را تعریف میکنیم.

به سه لایه کانولوشنی نیاز داریم که پارامتر های آن به صورت زیر تعریف میشود:

tf.keras.layers.Conv2D(filters, kernel size, activation function)

دو لایه مکس پول نیاز داریم که پارامتر های آن به صورت زیر تعریف میشود:

tf.keras.layers.MaxPooling2D(kernel size)

لایه flatten که ماتریس دو بعدی را تبدیل به وکتور یک بعدی میکند.

tf.keras.layers.Flatten

لایه dropout برای جلوگیری از overfit شدن شبکه، به طور تصادفی و با احتمال مشخص شده، برخی نورون ها را صفر میکند.

tf.keras.layers.Dropout(rate)

لایه dense نیز با پارامترهای زیر تعریف میشود:

tf.keras.layers.Dense(filters, activation function)

در قسمت call لایه ها را به صورت فانکشنال به ترتیب به هم متصل میکنیم و در واقع توپولوژی شبکه را مشخص میکنیم.

ساختار شبکه به صورت زیر است:

```

model = MyModel()
model.build(input_shape=(None, 32, 32, 1))
model.summary()

```

Model: "my_model_3"

Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	multiple	320
max_pooling2d_6 (MaxPooling 2D)	multiple	0
conv2d_10 (Conv2D)	multiple	18496
max_pooling2d_7 (MaxPooling 2D)	multiple	0
conv2d_11 (Conv2D)	multiple	36928
flatten_3 (Flatten)	multiple	0
dense_6 (Dense)	multiple	65600
dropout_3 (Dropout)	multiple	0
dense_7 (Dense)	multiple	650

Total params:	121,994
Trainable params:	121,994
Non-trainable params:	0

```
model.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
history = model.fit(x=train_images, y=train_labels, validation_split=0.1, batch_size=64, epochs=5, shuffle=True)
```

```

Epoch 1/5
844/844 [=====] - 67s 78ms/step - loss: 0.2242 - accuracy: 0.9319 - val_loss: 0.0386 - val_accuracy: 0.9878
Epoch 2/5
844/844 [=====] - 64s 76ms/step - loss: 0.0564 - accuracy: 0.9838 - val_loss: 0.0274 - val_accuracy: 0.9910
Epoch 3/5
844/844 [=====] - 66s 79ms/step - loss: 0.0376 - accuracy: 0.9891 - val_loss: 0.0246 - val_accuracy: 0.9928
Epoch 4/5
844/844 [=====] - 65s 77ms/step - loss: 0.0312 - accuracy: 0.9907 - val_loss: 0.0281 - val_accuracy: 0.9918
Epoch 5/5
844/844 [=====] - 66s 78ms/step - loss: 0.0226 - accuracy: 0.9931 - val_loss: 0.0163 - val_accuracy: 0.9953

```

۱۰ درصد از داده های آموزشی را به عنوان validation set جدا میکنیم و شبکه را آموزش میدهیم.

```

train_accuracy = history.history['accuracy']
validation_accuracy = history.history['val_accuracy']
train_loss = history.history['loss']
validation_loss = history.history['val_loss']

epochs = range(len(train_accuracy))

plt.plot(epochs, train_accuracy, 'b', label='Train accuracy')
plt.plot(epochs, validation_accuracy, 'r', label='Validation accuracy')
plt.title('Training and Validation accuracy')
plt.legend()

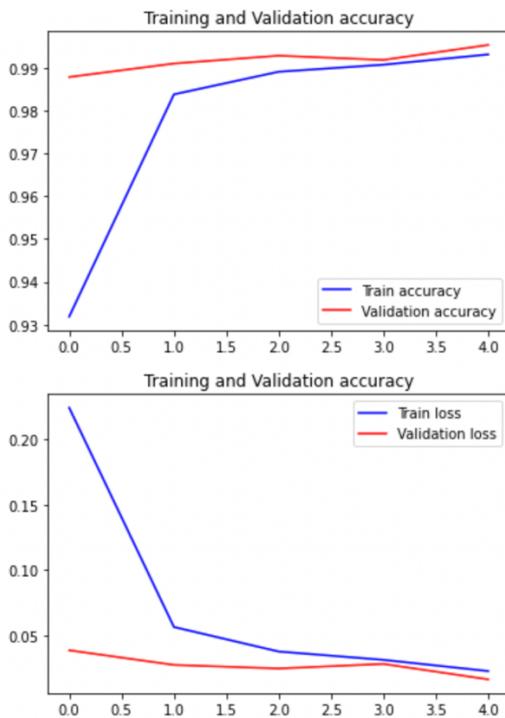
plt.figure()

plt.plot(epochs, train_loss, 'b', label='Train loss')
plt.plot(epochs, validation_loss, 'r', label='Validation loss')
plt.title('Training and Validation accuracy')
plt.legend()

plt.show()

```

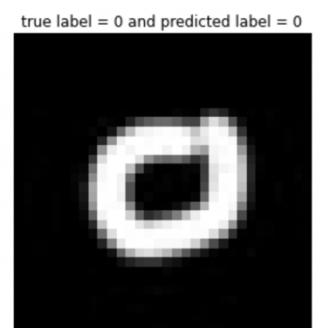
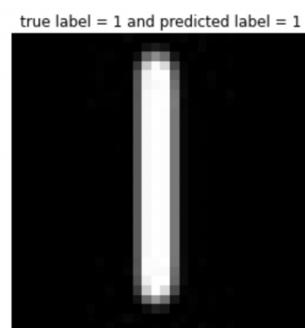
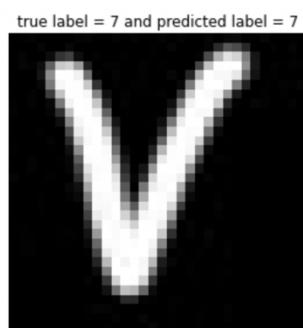
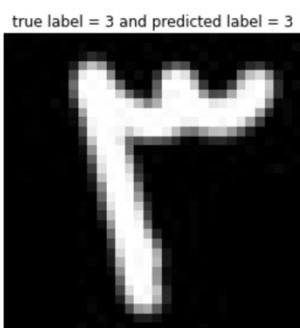
حال نتایج دقیق و خطای شبکه را ترسیم میکنیم:



برای تست شبکه، تعدادی عدد دست نویس با فتوشاپ ایجاد کردیم.
سپس یکی از آن ها را انتخاب کرده، لیبل آن ها را میخوانیم و با استفاده از `model.predict` به شبکه عرضه میکنیم تا شبکه لیبل آن ها را حدث بزند.
نتایج نشان میدهد شبکه به خوبی توانسته لیبل تصاویر تست را تشخیص دهد.

```
from glob import glob
import cv2
test_dir = glob('test/*.jpg')

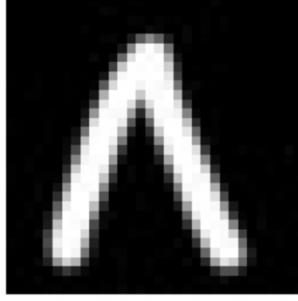
figure = plt.figure(figsize=(20, 20))
for i in range(1,len(test_dir)+1):
    test_img = cv2.imread(test_dir[i-1], cv2.IMREAD_GRAYSCALE).astype('float32')
    test_img.shape = (1,32, 32, 1)
    test_lbl = test_dir[i-1].split('/')[1].split('.')[0].split('_')[0]
    prediction = model.predict(test_img, verbose=0)
    predicted_lbl = np.argmax(prediction)
    figure.add_subplot(3, 4, i)
    plt.title('true label = ' + str(test_lbl)
              + ' and predicted label = ' + str(predicted_lbl))
    plt.axis("off")
    plt.imshow(test_img.reshape(32,32), cmap="gray")
plt.show()
```



true label = 6 and predicted label = 6



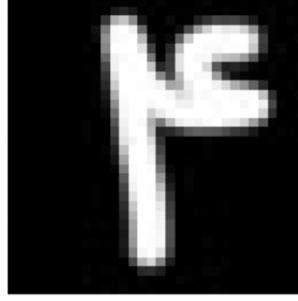
true label = 8 and predicted label = 8



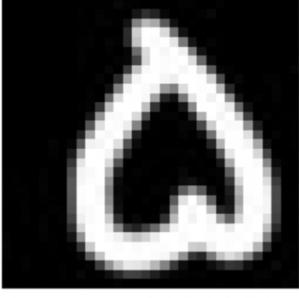
true label = 9 and predicted label = 9



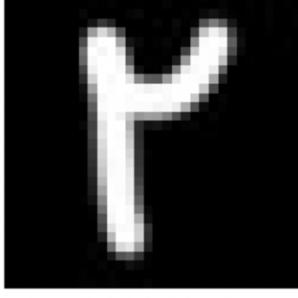
true label = 4 and predicted label = 4



true label = 5 and predicted label = 5



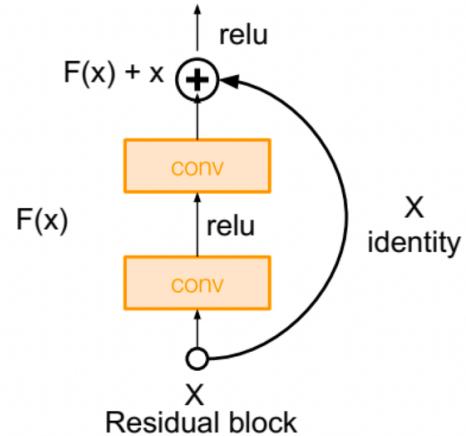
true label = 2 and predicted label = 2



-4

```
class ResBlock(Model):
    def __init__(self, channels, stride=1):
        super(ResBlock, self).__init__(name='ResBlock')
        self.flag = (stride != 1)
        self.conv1 = Conv2D(channels, 3, stride, padding='same')
        self.batchnorm1 = BatchNormalization()
        self.conv2 = Conv2D(channels, 3, padding='same')
        self.batchnorm2 = BatchNormalization()
        self.relu = ReLU()
        if self.flag:
            self.batchnorm3 = BatchNormalization()
            self.conv3 = Conv2D(channels, 1, stride)

    def call(self, x):
        y = self.conv1(x)
        y = self.batchnorm1(y)
        y = self.relu(y)
        y = self.conv2(y)
        y = self.batchnorm2(y)
        if self.flag:
            x = self.conv3(x)
            x = self.batchnorm3(x)
        y = Layers.add([x, y])
        y = self.relu(y)
        return y
```



ابتدا بلاک Residual را تعریف میکنیم.

هر بلاک، دو بار از ورودی کانولوشن میگیرد و بین آن دو مرحله کانولوشن نیز تابع relu اعمال میکند سپس حاصل را با خود ورودی جمع میکند.

در شبکه Resnet، گاهی اوقات ابعاد را نصف میکنیم و در عوض تعداد ویژگی ها را دو برابر میکنیم در این موقع قبل از جمع کردن X ، ابعاد آن را نیز باید تغییر بدھیم به همین دلیل یک flag تعریف میکنیم تا هنگامی که میخواهیم از stride بیشتر از 1 استفاده کنیم و ابعاد داده ها را تغییر دهیم، ابعاد X را نیز متناسب با آن تغییر دهیم.

در قسمت بعد مشخصات بلاک هایی که لازم داریم را تعریف میکنیم.

```

class ResNet34(Model):
    def __init__(self):
        super(ResNet34, self).__init__(name='ResNet34')
        self.conv = Conv2D(64, 7, 2, padding='same')
        self.batchnorm = BatchNormalization()
        self.relu = ReLU()
        self.maxpool = MaxPooling2D(3, 2)

        self.resnet_1_1 = ResBlock(64)
        self.resnet_1_2 = ResBlock(64)
        self.resnet_1_3 = ResBlock(64)

        self.resnet_2_1 = ResBlock(128, 2)
        self.resnet_2_2 = ResBlock(128)
        self.resnet_2_3 = ResBlock(128)
        self.resnet_2_4 = ResBlock(128)

        self.resnet_3_1 = ResBlock(256, 2)
        self.resnet_3_2 = ResBlock(256)
        self.resnet_3_3 = ResBlock(256)
        self.resnet_3_4 = ResBlock(256)
        self.resnet_3_5 = ResBlock(256)
        self.resnet_3_6 = ResBlock(256)

        self.resnet_4_1 = ResBlock(512, 2)
        self.resnet_4_2 = ResBlock(512)
        self.resnet_4_3 = ResBlock(512)

        self.globalpool = GlobalAveragePooling2D()
        self.fc1 = Dense(512, activation='relu')
        self.dp1 = Dropout(0.5)
        self.fc2 = Dense(512, activation='relu')
        self.dp2 = Dropout(0.5)
        self.fcf = Dense(7, activation='softmax')

```

سپس ساختار شبکه را با ارتباط دادن بلاک های تعریف شده تشکیل میدهیم.

```

def call(self, x):
    x = self.conv(x)
    x = self.batchnorm(x)
    x = self.relu(x)
    x = self.maxpool(x)

    x = self.resnet_1_1(x)
    x = self.resnet_1_2(x)
    x = self.resnet_1_3(x)

    x = self.resnet_2_1(x)
    x = self.resnet_2_2(x)
    x = self.resnet_2_3(x)
    x = self.resnet_2_4(x)

    x = self.resnet_3_1(x)
    x = self.resnet_3_2(x)
    x = self.resnet_3_3(x)
    x = self.resnet_3_4(x)
    x = self.resnet_3_5(x)
    x = self.resnet_3_6(x)

    x = self.resnet_4_1(x)
    x = self.resnet_4_2(x)
    x = self.resnet_4_3(x)

    x = self.globalpool(x)
    x = self.fc1(x)
    x = self.dp1(x)
    x = self.fc2(x)
    x = self.dp2(x)
    x = self.fcf(x)
    return x

```

ساختار شبکه به صورت زیر است:

```

model = ResNet34()
model.build(input_shape=(None, 48, 48, 1))
model.summary()

batch_normalization_36 (Batch Normalization) multiple 256
re_lu_17 (ReLU) multiple 0
max_pooling2d_1 (MaxPooling 2D) multiple 0
ResBlock (ResBlock) multiple 74368
ResBlock (ResBlock) multiple 74368
ResBlock (ResBlock) multiple 74368
ResBlock (ResBlock) multiple 231296
ResBlock (ResBlock) multiple 296192
ResBlock (ResBlock) multiple 296192
ResBlock (ResBlock) multiple 296192
ResBlock (ResBlock) multiple 921344
ResBlock (ResBlock) multiple 1182208
ResBlock (ResBlock) multiple 1182208
ResBlock (ResBlock) multiple 1182208
ResBlock (ResBlock) multiple 1182208
ResBlock (ResBlock) multiple 3677696
ResBlock (ResBlock) multiple 4723712
ResBlock (ResBlock) multiple 4723712
global_average_pooling2d_1 (GlobalAveragePooling2D) multiple 0
dense_3 (Dense) multiple 262656
dropout_2 (Dropout) multiple 0
dense_4 (Dense) multiple 262656
dropout_3 (Dropout) multiple 0
dense_5 (Dense) multiple 3591
=====
Total params: 21,832,839
Trainable params: 21,815,815
Non-trainable params: 17,024

```

Now compile the model with a proper optimizer and loss function.

```
[ ] model.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

then fit your model with enough epochs.

```
[ ] train_x = tf.expand_dims(ds.images, axis=-1)
train_x = tf.cast(x=train_x, dtype=np.float32)
train_y = tf.keras.utils.to_categorical(ds.labels)
```

```
[ ] history = model.fit(train_x, train_y, batch_size=32, validation_split=0.15, epochs=50)
```

```
Epoch 1/50
763/763 [=====] - 37s 44ms/step - loss: 1.8168 - accuracy: 0.2633 - val_loss: 1.7464 - val_accuracy: 0.2909
Epoch 2/50
763/763 [=====] - 32s 42ms/step - loss: 1.7220 - accuracy: 0.3038 - val_loss: 1.8431 - val_accuracy: 0.2515
Epoch 3/50
763/763 [=====] - 32s 42ms/step - loss: 1.6232 - accuracy: 0.3602 - val_loss: 1.5619 - val_accuracy: 0.3912
Epoch 4/50
763/763 [=====] - 31s 41ms/step - loss: 1.5598 - accuracy: 0.3910 - val_loss: 2.2857 - val_accuracy: 0.1785
-
```

چند مرحله آخر نیز به صورت زیر است:

```
Epoch 49/50
763/763 [=====] - 31s 41ms/step - loss: 0.8319 - accuracy: 0.6967 - val_loss: 1.3698 - val_accuracy: 0.5254
Epoch 50/50
763/763 [=====] - 31s 41ms/step - loss: 0.7799 - accuracy: 0.7150 - val_loss: 1.6568 - val_accuracy: 0.4467
Epoch 50/50
763/763 [=====] - 31s 41ms/step - loss: 0.7526 - accuracy: 0.7279 - val_loss: 1.5390 - val_accuracy: 0.5277
```

```
test_x = tf.expand_dims(ds_test.images, axis=-1)
test_x = tf.cast(x=test_x,dtype=np.float32)
test_y = tf.keras.utils.to_categorical(ds_test.labels)

y_pred = model.evaluate(test_x, test_y)

113/113 [=====] - 2s 14ms/step - loss: 1.5435 - accuracy: 0.5255
```

حال با استفاده از داده های تست شبکه را تست میکنیم، مشاهده میکنیم که به دقت ۵۲ درصد دست یافته ایم که با توجه به ماهیت شبکه Resnet که به داده های با کیفیت و تعداد epoch بالا نیاز دارد و داده های ما آنچنان کیفیت بالای ندارند و همچنین تعداد epoch را ۵۰ در نظر گرفتیم، قابل قبول است.