BAB 4:

CNN-1: Convolutional Neural Networks (CNN)

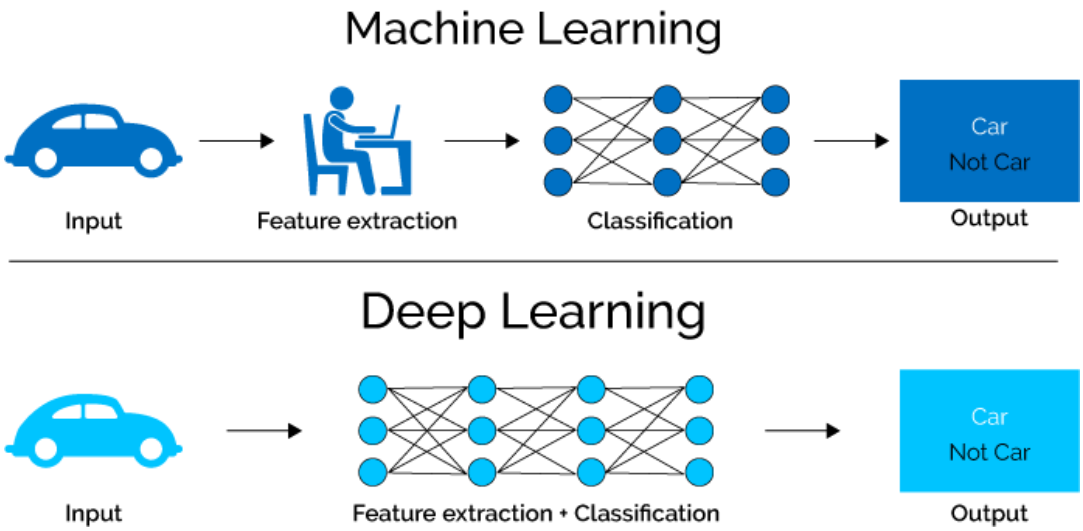
Di bab sebelumnya, kita melihat secara mendalam berbagai aspek `TensorFlow API` , mengenal tensor, memberi nama variabel, dan operator, serta mempelajari cara bekerja dengan cakupan variabel. Dalam bab ini, kita sekarang akan belajar tentang Convolutional Neural Networks (CNNs), dan bagaimana kita dapat mengimplementasikan CNN di `TensorFlow` .

Kita juga akan melakukan perjalanan yang menarik di bab ini saat kita menerapkan jenis arsitektur jaringan syaraf dalam ini ke klasifikasi gambar. Jadi kita akan mulai dengan membahas blok dasar CNN, menggunakan pendekatan dari bawah ke atas. Kemudian kita akan mendalami arsitektur CNN dan cara mengimplementasikan CNN mendalam di `TensorFlow` . Sepanjang jalan kita akan membahas topik-topik berikut:

> 1. Memahami operasi konvolusi dalam satu dan dua dimensi
> 2. Mempelajari tentang blok penyusun arsitektur CNN
> 3. Menerapkan *deep convolutional Neural Networks* di `TensorFlow`

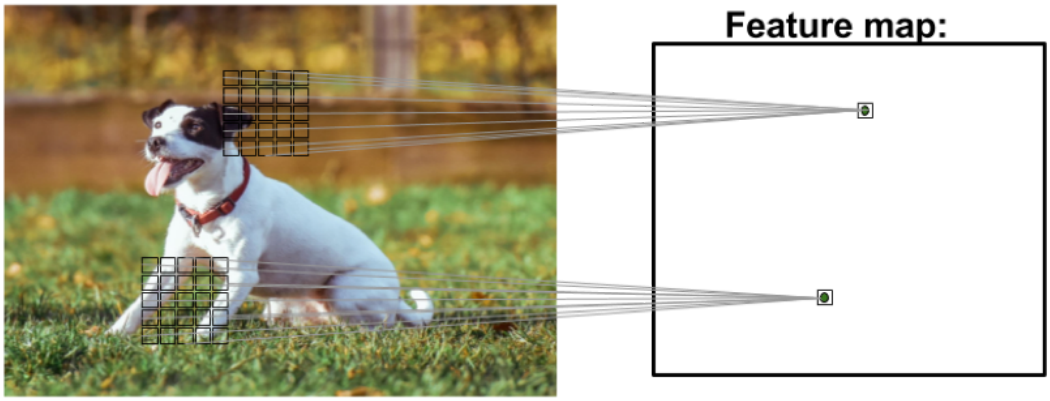
A. Memahami CNN dan mempelajari hierarki fitur

Berhasil mengekstraksi fitur yang menonjol (relevan) adalah kunci untuk kinerja setiap algoritma *machine learning*, tentu saja, dan model *machine learning* tradisional bergantung pada fitur input yang mungkin berasal dari domain expert, atau didasarkan pada teknik komputasi dari ekstraksi fitur. *Neural networks* memiliki kemampuan dapat secara otomatis mempelajari fitur dari raw data hal tersebut dapat menjadi sangat berguna untuk tugas tertentu. Untuk alasan ini, Neural networks dianggap sebagai mesin ekstraksi fitur: lapisan awal (yang tepat setelah lapisan input) mengekstraksi fitur tingkat rendah. Gambar 1. berikut perbandingan Deep Learning dan Machine Learning dimana di Deep Learning punya kemampuan mengekstraksi fitur didalam architecturenya.



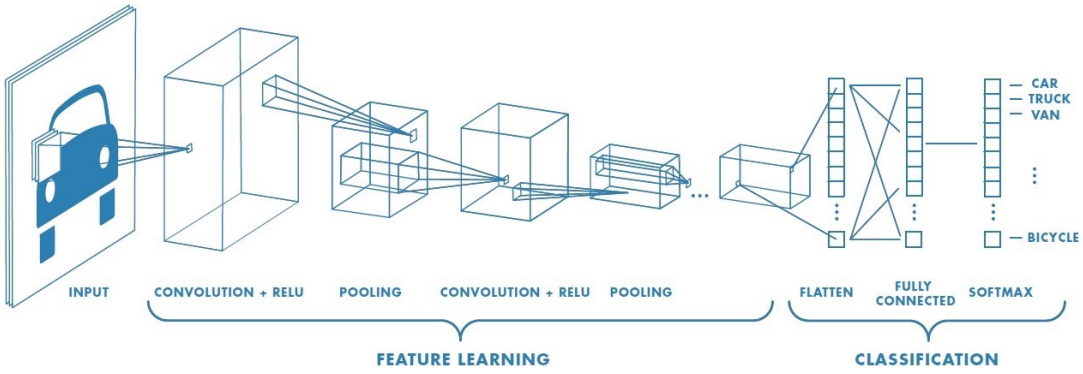
Gambar 1. Deep Learning dan Machine Learning

Multilayer neural network, dan khususnya, *deep convolutional neural networks*, membangun apa yang disebut **hierarki fitur** dengan menggabungkan fitur tingkat rendah dalam mode lapisan untuk membentuk fitur tingkat tinggi. Misalnya, jika kita berurusan dengan gambar, maka fitur tingkat rendah, seperti tepi dan blob, diekstraksi dari lapisan sebelumnya, yang digabungkan bersama untuk membentuk fitur tingkat tinggi – seperti bentuk objek seperti gedung, mobil , atau seekor anjing.



Gambar 2. Image hierarki fitur

Biasanya, CNN terdiri dari beberapa *Convolutional layer* (conv) dan subsampling (juga dikenal sebagai Pooling (P)) yang diikuti oleh satu atau lebih *Fully Connected (FC) layer* di bagian akhir. *Fully Connected (FC) layer* pada dasarnya adalah *multilayer perceptron*, di mana setiap unit input i terhubung ke setiap unit output j dengan $w_{i,j}$. Gambar 3. merupakan gambar arsitektur contoh CNN.



Gambar 3. Contoh arsitektur CNN

Harap dicatat bahwa *subsampling layer*, umumnya dikenal sebagai **Pooling layers**, tidak memiliki parameter yang dapat dipelajari; misalnya, tidak ada unit bobot atau bias dalam *pooling layer*. Namun, baik *convolution layer* maupun *fully connected layers* memiliki bobot dan bias seperti itu.

Pada bagian berikut, kita akan mempelajari *convolutional* dan *pooling layer* secara lebih detail dan melihat cara kerjanya. Untuk memahami bagaimana operasi *convolution* bekerja, mari kita mulai dengan *convolution* dalam satu dimensi sebelum membahas kasus dua dimensi tipikal sebagai aplikasi untuk gambar dua dimensi nanti.

B. Operasi Discreate convolution

Sebuah *discrete convolution* (atau sederhananya **convolution**) adalah operasi mendasar dalam CNN. Oleh karena itu, penting untuk memahami cara kerja operasi ini. *Convolution* didefinisikan sebagai cara untuk mengkombinasikan dua buah deret angka yang menghasilkan deret angka yang ketiga. Pada bagian ini, kita akan mempelajari definisi matematis dan membahas beberapa algoritma dasar untuk menghitung convolution dari dua vektor satu dimensi atau dua matriks dua dimensi.

Harap perhatikan bahwa uraian ini semata-mata untuk memahami cara kerja convolution. Memang, implementasi operasi *convolution* yang jauh lebih efisien sudah ada dalam paket seperti `TensorFlow`, seperti yang akan kita lihat nanti di bab ini.

Mathematical notation

Pada bab ini, kita akan menggunakan subskrip untuk menunjukkan ukuran array multidimensi; Sebagai contoh $w_{n_i \times n_i}$ adalah sebuah array dua dimensi dengan size $n_i \times n_i$. Kita menggunakan kurung seperti ini $[.]$ untuk menotasikan index dari sebua array multidimensi. Sebagai contoh $A[i, j]$ yang artinya adalah element pada index i, j dari matrik A. Selain itu, perhatikan bahwa kami menggunakan simbol $*$ khusus untuk menunjukkan operasi *convolution* antara dua vektor atau matriks, yang tidak sama dengan operator perkalian $*$ di Python.

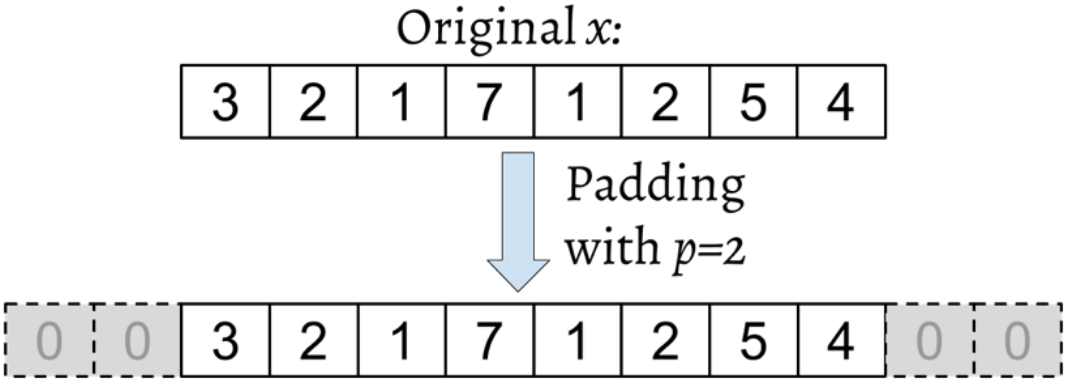
Operasi sebuah *discrete convolution* di array satu dimensi

Mari kita mulai dengan beberapa definisi dasar dan notasi yang akan kita gunakan. *Dicreate convolution* untuk dua vektor satu dimensi x dan w dilambangkan dengan $y = x * w$, di mana vektor x adalah input(kadang-kadang disebut **sinyal**) dan w disebut **filter atau kernel**. Konvolusi diskrit secara matematis didefinisikan sebagai berikut:

$$y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{\infty} x[i-k]w[k]$$

Di sini, tanda kurung [ ] digunakan untuk menunjukkan pengindeksan elemen vektor. Indeks i berjalan melalui setiap elemen dari vektor keluaran y. Ada dua hal aneh dalam rumus sebelumnya yang perlu kita perjelas: $-\infty$ ke ∞ indeks dan pengindeksan negatif untuk x.

Aturan tambahan adalah penambahan padding untuk input x untuk mengisi nilai lain selain data hingga dan stride untuk seberapa jauh operasi pergeseran input untuk operasi *convolution* untuk setia outputnya. Masalah pertama di mana penjumlahan dilakukan melalui indeks dari $-\infty$ ke ∞ tampak aneh terutama karena dalam aplikasi pembelajaran mesin, yang selalu berurusan dengan vektor fitur hingga/ *finite* . Misalnya, jika x memiliki 10 fitur dengan indeks $0, 1, 2, 3, \ldots, 8, 9$, maka indeks dari $-\infty : 1$ ke $10 : +\infty$ di luar batas untuk x. Oleh karena itu, untuk menghitung penjumlahan yang ditunjukkan pada rumus sebelumnya dengan benar, diasumsikan bahwa x dan w diisi dengan nol. Ini akan menghasilkan vektor keluaran y yang juga memiliki ukuran tak terbatas dengan banyak nol juga. Karena ini tidak berguna dalam situasi praktis, x diberikan *padding* hanya dengan jumlah nol yang terbatas. Gambar 3. merupakan gambaran process pemberian padding pada input x dengan nilai padding sama dengan 2.



Gambar 4. Padding input dengan jumlah padding = 2

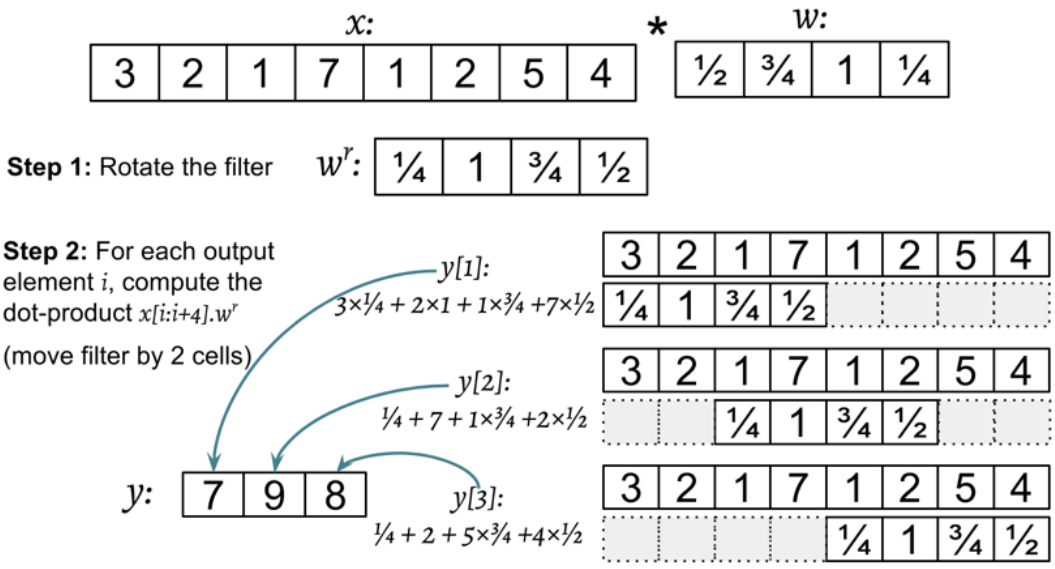
Mari kita asumsikan input x dan filter w masing-masing memiliki elemen n dan m, dimana dimensi filter lebih kecil dari dimensi inpu $m <= n$. Oleh karena itu, vektor yang telah diberi padding dinotasikan x^p memiliki ukuran $n + 2p$. Kemudian, rumus praktis untuk menghitung konvolusi diskrit akan berubah menjadi sebagai berikut:

$$y = x * w \rightarrow y[i] = \sum_{k=-0}^{k=m-1} x^p[i+m-k]w[k]$$

Dengan modifikasi diatas,kita telah memecahkan masalah indeks tak terhingga, masalah kedua adalah mengindeks x dengan $i+m-k$. Poin penting untuk diperhatikan di sini adalah bahwa x dan w diindeks dalam arah yang berbeda dalam penjumlahan ini. Untuk alasan ini, kita dapat membalikkan salah satu vektor tersebut, x atau w, setelah diberikan padding. Kemudian, kita cukup menghitung perkalian titik mereka.

Mari kita asumsikan kita membalik filter w untuk mendapatkan filter w^r yang diputar. Kemudian, dot product dihitung $X[i:i+m].w^r$ untuk mendapatkan satu elemen $y[i]$, dimana $x[i:i+m]$ adalah blok x dengan ukuran m.

Operasi ini diulangi seperti pada pendekatan jendela geser untuk mendapatkan semua elemen output. Gambar berikut memberikan contoh dengan $x = (3,2,1,7,1,2,5,4)$ dan $w = \left(\frac{1}{2}, \frac{3}{4}, 1, \frac{1}{4}\right)$ sehingga tiga elemen output pertama dihitung:



Gambar 5. Perhitungan convolution 1 vektor

Anda dapat melihat pada contoh sebelumnya bahwa ukuran padding adalah nol (p = 0). Perhatikan bahwa filter yang diputar w^r digeser dua sel setiap kali kita menggeser. Pergeseran ini adalah hyperparameter lain dari *convolution*, yang disebut **stride** s. Dalam contoh ini, *stride* adalah dua, s = 2. Perhatikan bahwa langkahnya harus bilangan positif yang lebih kecil dari ukuran input vektor. Kita akan berbicara lebih banyak tentang padding dan langkah di bagian selanjutnya!

Menentukan ukuran dari output convolution

Ukuran output dari sebuah *convolution* ditentukan oleh jumlah total kali process menggeser filter w sepanjang vektor input. Misalkan vektor input berukuran n dan filter berukuran m. Kemudian, ukuran output yang dihasilkan dari penambahan padding p dan stride s ditentukan sebagai berikut:

$$o = \left[\frac{n + 2p - m}{s}\right] + 1$$

Contoh:Hitung ukuran output untuk vektor input ukuran (n)10 dengan kernel *convolution (w)* ukuran (w) 5, padding (p)2, dan stride (s) 1:

$$n = 10, m = 5, p = 2, s = 1 \rightarrow o = \left[\frac{10 + 2 \times 2 - 5}{1}\right] + 1 = 10$$

In [ ]:
```
import numpy as np

def conv1d(x, w, p=0, s=1):
    w_rot = np.array(w[::-1])
    x_padded = np.array(x)
    if p>0:
        zero_pad = np.zeros(shape=p)
        x_padded = np.concatenate([zero_pad,x_padded,zero_pad])
    res = []
    for i in range(0, int(len(x)/s),s):
        res.append(np.sum(x_padded[i:i+w_rot.shape[0]] * w_rot))
    return np.array(res)
```

In [ ]:
```
x = [1, 3, 2, 4, 5, 6, 1, 3]
w = [1, 0, 3, 1, 2]
print('Conv1d Implementation:',conv1d(x, w, p=2, s=1))
```

```
Conv1d Implementation: [ 5. 14. 16. 26. 24. 34. 19. 22.]
```

In [ ]:
```
#numpy function
print('Numpy Results:',np.convolve(x, w, mode='same'))
```

```
Numpy Results: [ 5 14 16 26 24 34 19 22]
```

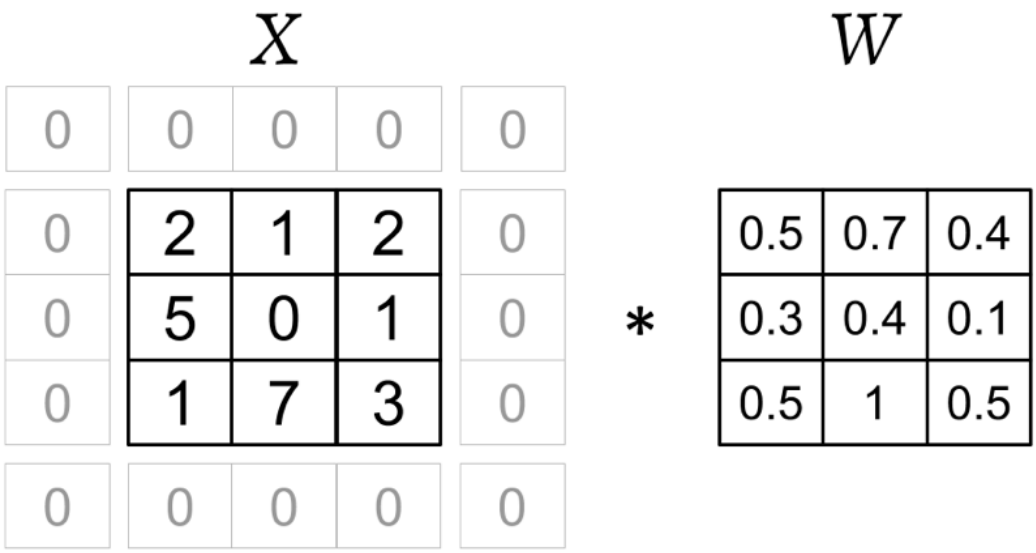
Sejauh ini, di sini kita telah menjelajahi *convolution* dalam 1D. Kami mulai dengan kasus 1D untuk membuat konsep lebih mudah dipahami. Pada bagian selanjutnya, kita akan memperluas ini menjadi dua dimensi.

Operasi sebuah *discrete convolution* di array dua dimensi

Konsep yang dipelajari di bagian sebelumnya diperluas ke dua dimensi. Ketika kita berurusan dengan input dua dimensi, seperti matriks $X_{n_1 \times n_2}$ dan matriks filter $W_{m_1 \times m_2}$, di mana $m_1 \leq n_1$ dan $m_2 \leq n_2$, maka matriks $Y = X * W$ tersebut adalah hasil *convolution* 2D dari X dengan W. Ini secara matematis didefinisikan sebagai berikut:

$$Y = X * W \rightarrow y[i,j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} x[i-k_1, j-k_2] w[k_1, k_2]$$

Perhatikan bahwa jika kita menghilangkan salah satu dimensi, rumus yang tersisa sama persis dengan rumus yang digunakan sebelumnya untuk menghitung *convolution* dalam 1D. Faktanya, semua teknik yang disebutkan sebelumnya, seperti zero-padding, memutar matriks filter, dan penggunaan strides, juga berlaku untuk *convolution* 2D, asalkan diperluas ke kedua dimensi secara terpisah. Contoh berikut mengilustrasikan perhitungan *convolution* 2D antara matriks input $X_{3\times3}$, matriks kernel $W_{3\times3}$, padding $P = (1,1)$, dan stride $S = (2,2)$. Menurut padding yang ditentukan, satu lapisan nol diisi pada setiap sisi matriks input, yang menghasilkan matriks padding $X_{3\times3}^{padded}$, sebagai berikut:

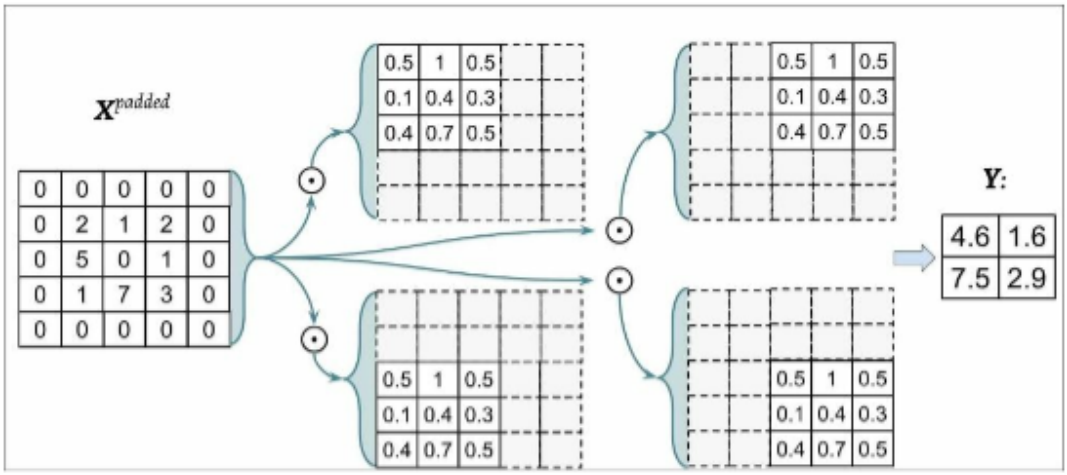


Gambar 5. Input matriks X dan Matriks kernel W

Dengan filter sebelumnya, filter yang diputar adalah:

$$W^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

Perhatikan bahwa rotasi ini tidak sama dengan matriks transpos. Untuk mendapatkan filter yang diputar di NumPy, kita dapat menulis W_rot=W[::-1,::-1]. Selanjutnya, kita dapat menggeser matriks filter yang diputar di sepanjang input matriks padding X^{padded} seperti window bergeser dan menghitung jumlah produk element dari dua matriks, yang dilambangkan dengan operator · pada gambar berikut dengan hasil 2 x 2 matriks Y:



Gambar 6. Hasil Convolution

In [ ]:
```
X = [[1, 3, 2, 4], [5, 6, 1, 3], [1, 2, 0, 2], [3, 4, 3, 2]]
W = [[1, 0, 3], [1, 2, 1], [0, 1, 1]]
```

In [ ]:
```
import scipy.signal
print('SciPy Results:\n', scipy.signal.convolve2d(X, W, mode='same'))
```

```
SciPy Results:
 [[11 25 32 13]
 [19 25 24 13]
 [13 28 25 17]
 [11 17 14  9]]
```

C. Operasi Subsampling (Pooling)

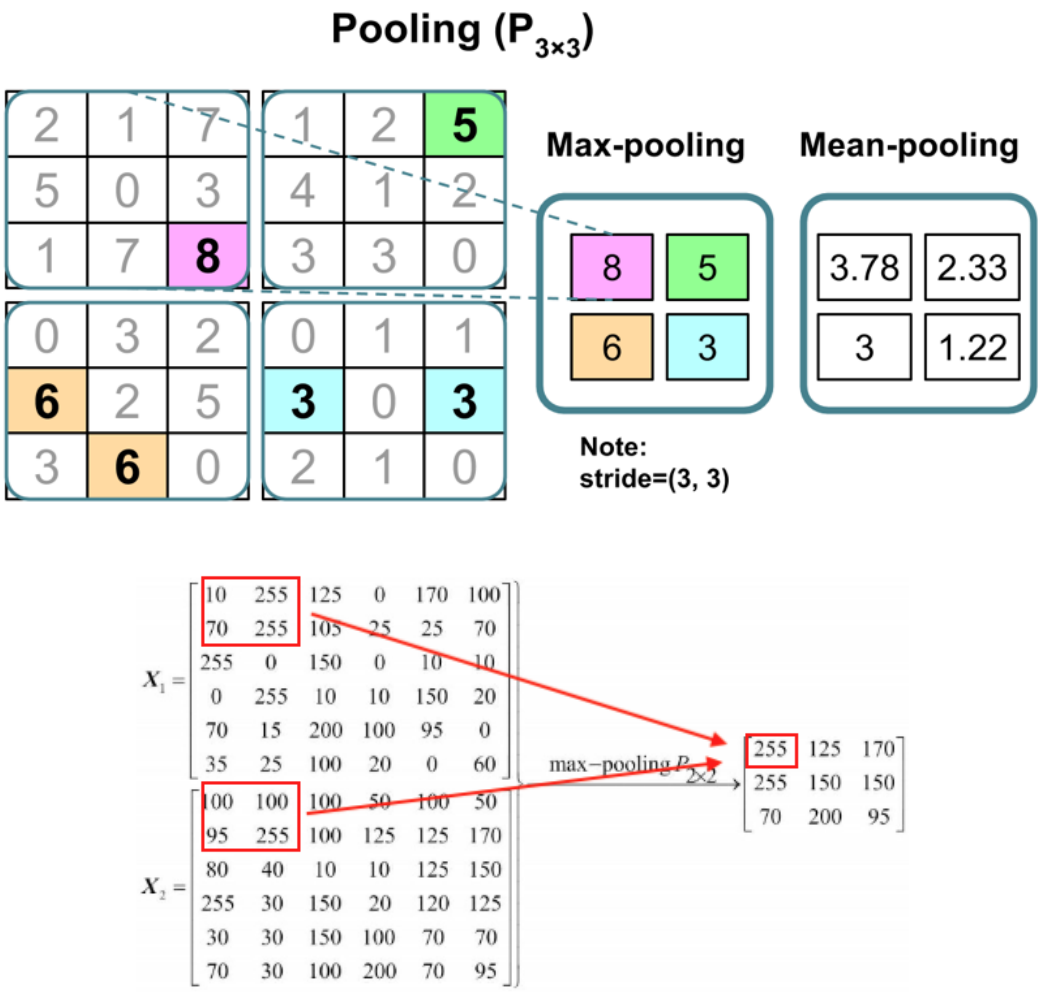
Subsampling biasanya diterapkan dalam dua bentuk operasi pooling di *convolutional neural networks* : **max-pooling** dan **mean-pooling** (juga dikenal sebagai average-pooling).

Pooling layer biasanya dilambangkan dengan $P_{n_1 \times n_2}$. Di sini, subskrip menentukan ukuran neighborhood (jumlah piksel yang berdekatan di setiap dimensi), dimana operasi maks atau rata-rata dilakukan. Kami menyebut neighborhood seperti itu sebagai **ukuran pooling**.

Gambar dibawah ini mendeskripsikan operasi max pooling.Di sini, max-pooling mengambil nilai maksimum dari neighborhood piksel, dan mean-pooling menghitung rata-ratanya.

Keuntungan dari pooling ada dua:

1. Pooling (max-pooling) memperkenalkan semacam invarian lokal. Ini berarti bahwa perubahan kecil di lingkungan lokal tidak mengubah hasil max pooling. Oleh karena itu, ini membantu menghasilkan fitur yang lebih kuat terhadap derau dalam data masukan. Lihat contoh berikut yang menunjukkan penyatuan maksimum dari dua matriks masukan yang berbeda X_1 dan X_2 menghasilkan keluaran yang sama:



Gambar 7. Max-pooling dan mean-pooling 3 x 3

2. Pooling mengurangi ukuran fitur, yang menghasilkan efisiensi komputasi yang lebih tinggi. Selain itu, mengurangi jumlah fitur juga dapat mengurangi tingkat overfitting.

D. Menyatukan semuanya untuk membangun sebuah CNN

Sejauh ini, kita telah belajar tentang blok bangunan dasar *convolutional neural networks*. Konsep yang diilustrasikan dalam bab ini sebenarnya tidak lebih sulit daripada *multilayer neural networks* tradisional. Secara intuitif, kita dapat mengatakan bahwa operasi terpenting dalam neural network tradisional adalah perkalian matriks-vektor.

Misalnya, kita menggunakan perkalian matriks-vektor untuk pra-aktivasi (atau net input) seperti pada $a = Wx + b$. Di sini, x adalah vektor kolom yang mewakili piksel, dan W adalah matriks bobot yang menghubungkan input piksel ke setiap unit hidden layer. Dalam *convolutional neural networks*, operasi ini digantikan oleh operasi convolutional, seperti di mana X adalah matriks yang mewakili piksel dalam susunan tinggi x lebar.Dalam kedua kasus, pra-aktivasi diteruskan ke fungsi aktivasi untuk mendapatkan aktivasi dari sebuah unit hidden leyer $H = \phi(A)$ disetiap convolution layer, dimana ϕ adalah fungsi aktivasi layaknya MLP. Selain itu, ingatlah bahwa

subsampling adalah blok bangunan lain dari *convolutional neural networks*, yang mungkin muncul dalam bentuk *pooling*, seperti yang telah kami jelaskan di bagian sebelumnya.

Bekerja dengan beberapa input atau channel warna

Sampel input ke *Convotional layer* dapat berisi satu atau lebih dari 2D matriks array dengan dimensi $N_1 \times N_2$ (misalnya, tinggi dan lebar gambar dalam piksel). Dimensi $N_1 \times N_2$ kumpulan matrik disebut **channels**. Oleh karena itu, menggunakan *multiple channel* sebagai input ke *convotional leyer* mengharuskan menggunakan tensor rank-3 atau 3D array: $X_{N_1 \times N_2 \times C_{in}}$, C_{in} merupakan jumlah input channel. Sebagai contoh, mari kita lihat gambar sebagai input ke lapisan pertama CNN. Jika gambar diwarnai dan menggunakan mode warna RGB, maka $C_{in} = 3$ (untuk saluran warna merah, hijau, dan biru dalam RGB). Namun, jika gambar dalam skala abu-abu *(grayscale)*, maka kita memiliki $C_{in} = 1$ karena hanya ada satu channel dengan nilai intensitas piksel skala abu-abu.

Selanjutnya, mari kita lihat contoh bagaimana kita bisa membaca gambar ke dalam sesi Python kita menggunakan SciPy. Namun, perlu diketahui bahwa membaca gambar dengan SciPy mengharuskan Anda menginstal paket Python Imaging Library (PIL). Kita dapat menginstal pillow (https://python-pillow.org).

In [ ]:
```
pip install pillow
```

```
Requirement already satisfied: pillow in /Users/istiqomah/miniforge3/envs/
tensorEnv/lib/python3.8/site-packages (9.5.0)
Note: you may need to restart the kernel to use updated packages.
```

Sekali Pillow diinstall, kita dapat menggunakan fungsi imread dari modul scipy.misc untuk membaca gambar RGB (contoh gambar ini terletak di folder bundel kode yang disediakan dengan bab ini di https://github.com/rasbt/python-machine-learning-book-2nd-edition/tree/master/code/ch15):

In [ ]:
```
from PIL import Image
import numpy as np
with Image.open('example-image.png').convert('RGB') as img:
    img = np.array(img)
    print('Image shape:', img.shape)
    print('Number of channels:', img.shape[2])
    print('Image data type:', img.dtype)
    print(img[100:102, 100:102, :])
```

```
Image shape: (252, 221, 3)
Number of channels: 3
Image data type: uint8
[[[179 134 110]
  [182 136 112]]

 [[180 135 111]
  [182 137 113]]]
```

Sekarang setelah kita memahami struktur input data, pertanyaan selanjutnya adalah bagaimana kita bisa menggabungkan beberapa saluran input dalam operasi

convolution yang telah kita bahas di bagian sebelumnya?

Jawabannya sangat sederhana: kita melakukan operasi *convolution* untuk setiap channel secara terpisah dan kemudian menjumlahkan hasilnya menggunakan penjumlahan matriks. *Convolution* yang terkait dengan setiap channel (c) memiliki matriks kernelnya sendiri $W[:, :, c]$. Total hasil pra-aktivasi dihitung dalam rumus berikut:

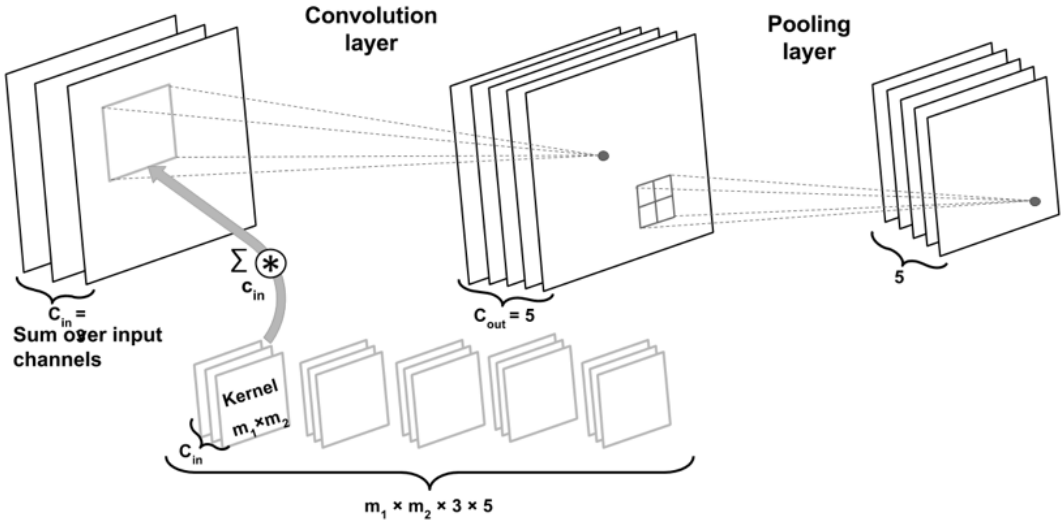
$$\begin{aligned} &\text{Given a sample } \mathbf{X}_{n_1 \times n_2 \times c_{in}}, \\ &\text{a kernel matrix } \mathbf{W}_{m_1 \times m_2 \times c_{in}}, \\ &\text{and bias value } b \end{aligned} \quad \Rightarrow \quad \begin{cases} \boldsymbol{Y}^{Conv} = \sum_{c=1}^{C_{in}} \boldsymbol{W}[:,:,c] * \boldsymbol{X}[:,:,c] \\ \text{pre-activation}: \quad \mathbf{A} = \mathbf{Y}^{\text{Conv}} + \mathrm{b} \\ \text{Feature map}: \quad \boldsymbol{H} = \phi(\boldsymbol{A}) \end{cases}$$

Hasil akhir, H, disebut **feature map**. Biasanya, convolutional layer di CNN memiliki lebih dari satu *feature map* (k). Jika kita menggunakan multiple feature map, tensor kernel menjadi empat dimensi: ($width \times height \times C_{in} \times C_{out}$). Di sini, ($width \times height$) lebar x tinggi adalah ukuran kernel, C_{in} jumlah channel input, dan C_{out} jumlah *feature map* output. Jadi, sekarang mari sertakan jumlah *feature map* keluaran dalam rumus sebelumnya dan perbarui sebagai berikut:

$$\begin{aligned} &\text{Given a sample } \boldsymbol{X}_{n_1 \times n_2 \times C_{in}}, \\ &\text{kernel matrix } \boldsymbol{W}_{m_1 \times m_2 \times C_{in} \times C_{out}}, \\ &\text{and bias vector } \boldsymbol{b}_{C_{out}} \end{aligned} \quad \Rightarrow \quad \begin{cases} \boldsymbol{Y}^{Conv}[:,:,k] = \sum_{c=1}^{C_{in}} \boldsymbol{W}[:,:,c,k] * \boldsymbol{X}[:,:,c] \\ \boldsymbol{A}[:,:,k] = \boldsymbol{Y}^{Conv}[:,:,k] + \boldsymbol{b}[k] \\ \boldsymbol{H}[:,:,k] = \phi(\boldsymbol{A}[:,:,k]) \end{cases}$$

Sebagai penutup pembahasan kita mengenai komputasi *convolution* dalam konteks neural network, mari kita lihat contoh pada gambar berikut yang menunjukkan convolutional layer, diikuti dengan pooling layer.

Dalam contoh ini, ada tiga channel input. Tensor kernel adalah empat dimensi. Setiap matriks kernel dilambangkan sebagai $m_1 \times m_2$, dan ada tiga array 2D di antaranya, satu untuk setiap channel input. Selain itu, ada lima kernel seperti itu, terhitung untuk lima *feature map* keluaran. Terakhir, ada pooling layer untuk membuat subsampling peta fitur, seperti yang ditunjukkan pada gambar berikut:



Gambar 8. Process komputasi Convolution dan pooling layer

E. Implementasi sebuah *deep convolutional neural networks* menggunakan TensorFlow

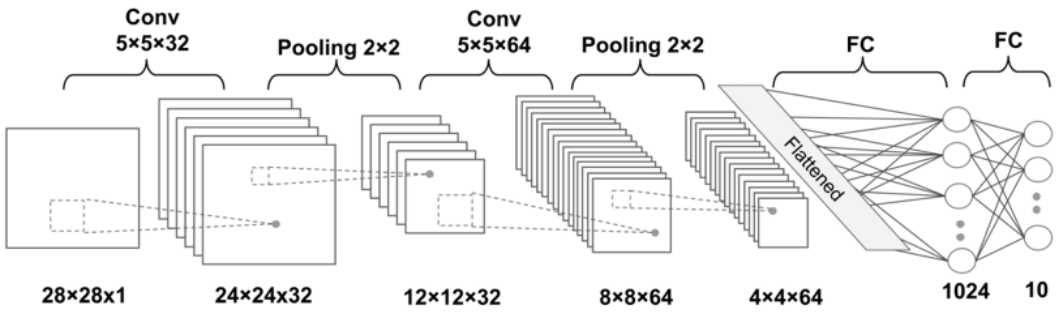
Pada pejelasan di bab sebelumnya telah digunakan diberikan contoh implementasi *Multilayer neural networks*. Pada chapter ini akan diberkan pemaparan pengimplementasian API leber dari TansorFlow untuk CNN. dataset yang digunakan adalah dataset tulisan tangga mnist.

Arsitektur multilayer CNN

Arsitektur network yang akan kita implementasikan ditunjukkan pada gambar berikut. Inputnya adalah grayscale berukuran 28 x 28. Mempertimbangkan jumlah channel (yaitu 1 untuk gambar skala abu-abu) dan bacth gambar input, dimensi tensor masukan akan menjadi batchsize x 28 x 28 x 1.

Data input melewati dua *convolutional layer* yang memiliki ukuran kernel 5 x 5. *Convolution* pertama memiliki 32 *feature map* output, dan yang kedua memiliki 64 *feature map* output. Setiap lapisan konvolusi diikuti oleh lapisan subsampling dalam bentuk operasi max-pooling.

Kemudian sebuah *fully-connected layer* meneruskan output ke *fully-connected layer* kedua, yang bertindak sebagai lapisan keluaran softmax terakhir. Arsitektur network yang akan kita terapkan ditunjukkan pada gambar berikut:



Gambar 9. Implementasi Arsitektur CNN

Dimensi tensor di setiap lapisan adalah sebagai berikut:

> 1. **Input:** $[batchsize \times 28 \times 28 \times 1]$
> 2. **Conv_1:** $[batchsize \times 24 \times 24 \times 32]$
> 3. **Pooling_1:** $[batchsize \times 12 \times 12 \times 32]$
> 4. **Conv_2:** $[batchsize \times 8 \times 8 \times 64]$
> 5. **Pooling_1:** $[batchsize \times 4 \times 4 \times 64]$
> 6. **FC_1:** $[batchsize \times 1024]$
> 7. **FC_2 dan softmax layer:** $[batchsize \times 10]$

Loading dan preprocessing data

Kita akan gunakan fungsi load_minst dibawah ini untuk membaca dataset tulisan tangan.

In [ ]:
```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
```

```
/Users/istiqomah/miniforge3/envs/tensorEnv/lib/python3.8/site-packages/sklearn/datasets/_openml.py:968: FutureWarning: The default value of `parser` will change from `'liac-arff'` to `'auto'` in 1.4. You can set `parser='auto'` to silence this warning. Therefore, an `ImportError` will be raised from 1.4 if the dataset is dense and pandas is not installed. Note that the pandas parser may return different data types. See the Notes Section in fetch_openml's API doc for details.
  warn(
```

Out[ ]:
```
dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_names', 'DESCR', 'details', 'url'])
```

In [ ]:
```
X, y = mnist["data"], mnist["target"]
X.shape
```

Out[ ]:
```
(70000, 784)
```

In [ ]:
```
import numpy as np
from sklearn.model_selection import train_test_split
X= X.to_numpy()
X_data, y_data = X[:60000,:], y[:60000]
X_test, y_test = X[60000:,:], y[60000:]
```

In [ ]:
```
X_train, y_train = X_data[:50000,:], y_data[:50000]
X_valid, y_valid = X_data[50000:,:], y_data[50000:]
```

In [ ]:
```
print('Training:   ', X_train.shape, y_train.shape)
print('Validation: ', X_valid.shape, y_valid.shape)
print('Test Set:   ', X_test.shape, y_test.shape)
```

```
Training:    (50000, 784) (50000,)
Validation:  (10000, 784) (10000,)
Test Set:    (10000, 784) (10000,)
```

Setelah kami memuat data, kami membutuhkan fungsi untuk iterasi melalui mini-batches dari data, sebagai berikut:

In [ ]:
```
def batch_generator(X, y, batch_size=64,
                    shuffle=False, random_seed=None):

    idx = np.arange(y.shape[0])
    if shuffle:
        rng = np.random.RandomState(random_seed)
        rng.shuffle(idx)
        X = X[idx]
        y = y[idx]
    for i in range(0, X.shape[0], batch_size):
        yield (X[i:i+batch_size, :], y[i:i+batch_size])
```

Fungsi ini mengembalikan generator dengan tuple untuk kecocokan sampel, misalnya, data X dan label y. kita kemudian perlu menormalkan data (pemusatan rata-rata data dan deviasinya dengan standar deviasi) untuk kinerja dan konvergensi dari process training yang lebih baik.

Kami menghitung rata-rata setiap fitur menggunakan data training (X_train) dan menghitung standar deviasi di semua fitur. Alasan mengapa kami tidak menghitung deviasi standar untuk setiap fitur satu per satu adalah karena beberapa fitur (posisi piksel) dalam dataset gambar seperti MNIST memiliki nilai konstan 255 di semua gambar yang sesuai dengan piksel putih dalam gambar skala abu-abu.

Nilai konstan di semua sampel menunjukkan tidak ada variasi, dan oleh karena itu, standar deviasi dari fitur tersebut akan menjadi nol, dan hasilnya akan menghasilkan kesalahan pembagian dengan nol, itulah sebabnya kami menghitung standar deviasi dari array X_train menggunakan np .std tanpa menentukan argumen axis:

In [ ]:
```
mean_vals = np.mean(X_train, axis=0)
std_val = np.std(X_train)

X_train_centered = (X_train - mean_vals)/std_val
X_valid_centered = (X_valid - mean_vals)/std_val
X_test_centered = (X_test - mean_vals)/std_val

del X_data, y_data, X_train, X_valid, X_test
```

Implementasi CNN di TensorFlow Layers API

In [ ]:
```
import tensorflow.compat.v1 as tf
import numpy as np
import os


class ConvNN(object):
    def __init__(self, batchsize=64,
                 epochs=20, learning_rate=1e-4,
                 dropout_rate=0.5,
                 shuffle=True, random_seed=None):
        np.random.seed(random_seed)
        self.batchsize = batchsize
```

```
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.dropout_rate = dropout_rate
        self.shuffle = shuffle

        g = tf.Graph()
        with g.as_default():
            ## set random-seed:
            tf.set_random_seed(random_seed)

            ## build the network:
            self.build()

            ## initializer
            self.init_op = \
                tf.global_variables_initializer()

            ## saver
            self.saver = tf.train.Saver()

        ## create a session
        self.sess = tf.Session(graph=g)

    def build(self):

        ## Placeholders for X and y:
        tf_x = tf.placeholder(tf.float32,
                              shape=[None, 784],
                              name='tf_x')
        tf_y = tf.placeholder(tf.int32,
                              shape=[None],
                              name='tf_y')
        is_train = tf.placeholder(tf.bool,
                              shape=(),
                              name='is_train')

        ## reshape x to a 4D tensor:
        ##  [batchsize, width, height, 1]
        tf_x_image = tf.reshape(tf_x, shape=[-1, 28, 28, 1],
                              name='input_x_2dimages')
        ## One-hot encoding:
        tf_y_onehot = tf.one_hot(indices=tf_y, depth=10,
                              dtype=tf.float32,
                              name='input_y_onehot')

        ## 1st layer: Conv_1
        h1 = tf.layers.conv2d(tf_x_image,
                              kernel_size=(5, 5),
                              filters=32,
                              activation=tf.nn.relu)
        ## MaxPooling
        h1_pool = tf.layers.max_pooling2d(h1,
                              pool_size=(2, 2),
                              strides=(2, 2))
        ## 2n layer: Conv_2
        h2 = tf.layers.conv2d(h1_pool, kernel_size=(5,5),
                              filters=64,
                              activation=tf.nn.relu)
        ## MaxPooling
        h2_pool = tf.layers.max_pooling2d(h2,
```

```
                            pool_size=(2, 2),
                            strides=(2, 2))

        ## 3rd layer: Fully Connected
        input_shape = h2_pool.get_shape().as_list()
        n_input_units = np.prod(input_shape[1:])
        h2_pool_flat = tf.reshape(h2_pool,
                            shape=[-1, n_input_units])
        h3 = tf.layers.dense(h2_pool_flat, 1024,
                            activation=tf.nn.relu)

        ## Dropout
        h3_drop = tf.layers.dropout(h3,
                            rate=self.dropout_rate,
                            training=is_train)

        ## 4th layer: Fully Connected (linear activation)
        h4 = tf.layers.dense(h3_drop, 10,
                            activation=None)

        ## Prediction
        predictions = {
            'probabilities': tf.nn.softmax(h4,
                            name='probabilities'),
            'labels': tf.cast(tf.argmax(h4, axis=1),
                            tf.int32, name='labels')}

        ## Loss Function and Optimization
        cross_entropy_loss = tf.reduce_mean(
            tf.nn.softmax_cross_entropy_with_logits(
                logits=h4, labels=tf_y_onehot),
            name='cross_entropy_loss')

        ## Optimizer
        optimizer = tf.train.AdamOptimizer(self.learning_rate)
        optimizer = optimizer.minimize(cross_entropy_loss,
                                       name='train_op')

        ## Finding accuracy
        correct_predictions = tf.equal(
            predictions['labels'],
            tf_y, name='correct_preds')

        accuracy = tf.reduce_mean(
            tf.cast(correct_predictions, tf.float32),
            name='accuracy')

    def save(self, epoch):

        self.saver.save(self.sess, 'model.ckpt',global_step=epoch)

    def load(self, epoch, path):
        print('Loading model from %s' % path)
        self.saver.restore(self.sess,
             os.path.join(path, 'model.ckpt-%d' % epoch))

    def train(self, training_set,
              validation_set=None,
              initialize=True):
        ## initialize variables
```

```
        if initialize:
            self.sess.run(self.init_op)

        self.train_cost_ = []
        X_data = np.array(training_set[0])
        y_data = np.array(training_set[1])

        for epoch in range(1, self.epochs + 1):
            batch_gen = \
                batch_generator(X_data, y_data,
                                shuffle=self.shuffle)
            avg_loss = 0.0
            for i, (batch_x,batch_y) in \
                enumerate(batch_gen):
                feed = {'tf_x:0': batch_x,
                        'tf_y:0': batch_y,
                        'is_train:0': True} ## for dropout
                loss, _ = self.sess.run(
                        ['cross_entropy_loss:0', 'train_op'],
                        feed_dict=feed)
                avg_loss += loss

            print('Epoch %02d: Training Avg. Loss: '
                  '%7.3f' % (epoch, avg_loss), end=' ')
            if validation_set is not None:
                feed = {'tf_x:0': batch_x,
                        'tf_y:0': batch_y,
                        'is_train:0': False} ## for dropout
                valid_acc = self.sess.run('accuracy:0',
                                          feed_dict=feed)
                print('Validation Acc: %7.3f' % valid_acc)
            else:
                print()

    def predict(self, X_test, return_proba = False):
        feed = {'tf_x:0': X_test,
                'is_train:0': False} ## for dropout
        if return_proba:
            return self.sess.run('probabilities:0',
                                 feed_dict=feed)
        else:
            return self.sess.run('labels:0',
                                 feed_dict=feed)
```

In [ ]:
```
cnn = ConvNN(random_seed=123)
```

```
WARNING:tensorflow:From /Users/istiqomah/miniforge3/envs/tensorEnv/lib/pyth
on3.8/site-packages/tensorflow/python/util/dispatch.py:1176: softmax_cros
s_entropy_with_logits (from tensorflow.python.ops.nn_ops) is deprecated an
d will be removed in a future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See `tf.nn.softmax_cross_entropy_with_logits_v2`.
```

```
<ipython-input-15-bc8086ffb266>:59: UserWarning: `tf.layers.conv2d` is deprecated and will be removed in a future version. Please Use `tf.keras.layers.Conv2D` instead.
  h1 = tf.layers.conv2d(tf_x_image,
<ipython-input-15-bc8086ffb266>:64: UserWarning: `tf.layers.max_pooling2d` is deprecated and will be removed in a future version. Please use `tf.keras.layers.MaxPooling2D` instead.
  h1_pool = tf.layers.max_pooling2d(h1,
<ipython-input-15-bc8086ffb266>:68: UserWarning: `tf.layers.conv2d` is deprecated and will be removed in a future version. Please Use `tf.keras.layers.Conv2D` instead.
  h2 = tf.layers.conv2d(h1_pool, kernel_size=(5,5),
<ipython-input-15-bc8086ffb266>:72: UserWarning: `tf.layers.max_pooling2d` is deprecated and will be removed in a future version. Please use `tf.keras.layers.MaxPooling2D` instead.
  h2_pool = tf.layers.max_pooling2d(h2,
<ipython-input-15-bc8086ffb266>:81: UserWarning: `tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.keras.layers.Dense` instead.
  h3 = tf.layers.dense(h2_pool_flat, 1024,
<ipython-input-15-bc8086ffb266>:85: UserWarning: `tf.layers.dropout` is deprecated and will be removed in a future version. Please use `tf.keras.layers.Dropout` instead.
  h3_drop = tf.layers.dropout(h3,
<ipython-input-15-bc8086ffb266>:90: UserWarning: `tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.keras.layers.Dense` instead.
  h4 = tf.layers.dense(h3_drop, 10,
```

In [ ]:
```
cnn.train(training_set=(X_train_centered, y_train),
          validation_set=(X_valid_centered, y_valid),
          initialize=True)
```

```
Epoch 01: Training Avg. Loss: 258.414 Validation Acc:   1.000
Epoch 02: Training Avg. Loss:  70.070 Validation Acc:   1.000
Epoch 03: Training Avg. Loss:  47.011 Validation Acc:   1.000
Epoch 04: Training Avg. Loss:  36.280 Validation Acc:   1.000
Epoch 05: Training Avg. Loss:  29.789 Validation Acc:   1.000
Epoch 06: Training Avg. Loss:  25.995 Validation Acc:   1.000
Epoch 07: Training Avg. Loss:  21.324 Validation Acc:   0.938
Epoch 08: Training Avg. Loss:  19.806 Validation Acc:   1.000
Epoch 09: Training Avg. Loss:  15.669 Validation Acc:   1.000
Epoch 10: Training Avg. Loss:  13.667 Validation Acc:   1.000
Epoch 11: Training Avg. Loss:  11.654 Validation Acc:   1.000
Epoch 12: Training Avg. Loss:  10.735 Validation Acc:   1.000
Epoch 13: Training Avg. Loss:   9.348 Validation Acc:   1.000
Epoch 14: Training Avg. Loss:   8.140 Validation Acc:   1.000
Epoch 15: Training Avg. Loss:   7.006 Validation Acc:   1.000
Epoch 16: Training Avg. Loss:   6.906 Validation Acc:   1.000
Epoch 17: Training Avg. Loss:   5.835 Validation Acc:   1.000
Epoch 18: Training Avg. Loss:   4.879 Validation Acc:   1.000
Epoch 19: Training Avg. Loss:   4.294 Validation Acc:   1.000
Epoch 20: Training Avg. Loss:   4.917 Validation Acc:   1.000
```

In [ ]:
```
print(cnn.predict(X_test_centered[:10,:]))
```

```
[7 2 1 0 4 1 4 9 5 9]
```

In [ ]:
```
preds = cnn.predict(X_test_centered)
y_test = [int(i) for i in y_test]
```

```
print('Test Accuracy: %.2f%%' % (100*
      np.sum(y_test == preds)/len(y_test)))
```

```
Test Accuracy: 99.28%
```