

Chapter 3:

TensorFlow-2: Model Prediksi Neural Networks

April 2, 2023

Pada bagian sebelumnya, kita telah mempelajari *basic utility* komponen-komponen yang ada di **TensorFlow** untuk memanipulasi tensor-tensor dan mengorganisasikan data ke dalam format-format yang bisa kita iterasi selama training. Di bagian ini, kita akan mengimplementasikan model prediksi pertama pada **TensorFlow**. Karena **TensorFlow** lebih fleksibel dan lebih kompleks dibandingkan dengan library machine learning seperti **scikit-learn**, kita akan mulai dengan masalah sederhana yaitu model regresi linier.

1 TensorFlow Keras API (`tf.keras`)

Keras merupakan *high-level* Neural Networks (NN) API dan asal mulanya dibangun untuk bekerja di atas library lain seperti **TensorFlow** dan **Theano**. **Keras** menyediakan antarmuka pemrograman yang *user-friendly* dan modular sehingga mudah untuk melakukan *prototyping* dan membangun model-model yang kompleks hanya dengan beberapa baris kode. **Keras** terintegrasi ke dalam **TensorFlow** dan modul-modulnya dapat diakses melalui `tf.keras`. Pada **TensorFlow 2.0**, `tf.keras` telah menjadi pendekatan utama dan yang direkomendasikan untuk mengimplementasikan model-model. Selain itu `tf.keras` juga mendukung fungsi-fungsi spesifik **TensorFlow** seperti *pipelines* dataset menggunakan `tf.data` yang telah kita pelajari sebelumnya. Pada modul-modul ini, kita akan menggunakan modul `tf.keras` untuk membangun model-model NN.

API dari **Keras** membuat kita mudah untuk membangun model NN. Pendekatan paling umum untuk membangun model NN pada **TensorFlow** adalah dengan `tf.keras.Sequential()` yang dapat digunakan untuk menumpukan layer-layer (*stacking layers*) membentuk sebuah Network. Sebuah tumpukan dari layer dapat diberikan dalam bentuk list dari **Python** untuk model yang didefinisikan sebagai `tf.keras.Sequential()`. Sebagai alternatif, layer-layer dapat ditambahkan satu demi satu menggunakan metode `.add()`.

Lebih jauh, `tf.keras` dapat digunakan untuk mendefinisikan sebuah model dengan membuat subclass `tf.keras.Model`. Kita akan melihat juga bahwa model-model yang dibangun menggunakan API `tf.keras` dapat dikompilasi dan di-training menggunakan metode `.compile()` dan `.fit()`.

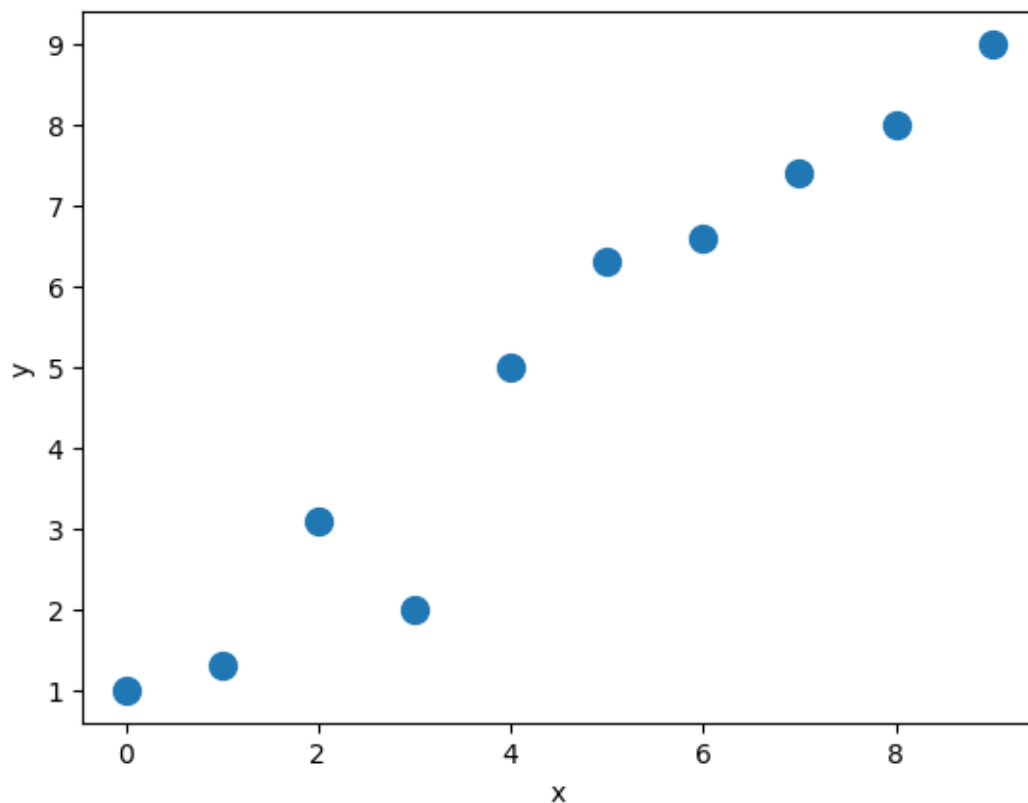
1.1 Membangun Model Regresi Linier

- Kita akan membuat model sederhana untuk memecahkan masalah regresi linier. Pertama kita akan buat sebuah dataset mainan pada **NumPy** dan memberikan visualisasi untuk dataset tersebut.

```
[114]: import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
[115]: X_train = np.arange(10).reshape((10, 1)) # 10 data fitur 1 dimensi
y_train = np.array([1.0, 1.3, 3.1, 2.0, 5.0, 6.3, 6.6, 7.4, 8.0, 9.0]) # 10 data
↳vektor target 1 dimensi

plt.plot(X_train, y_train, 'o', markersize=10)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



- Kita akan standarisasi fitur-fitur (*mean centering* dan membaginya dengan standar deviasi), kemudian membuat dataset TensorFlow.

```
[117]: X_train_norm = (X_train - np.mean(X_train))/np.std(X_train)

ds_train_orig = tf.data.Dataset.from_tensor_slices(
    (tf.cast(X_train_norm, tf.float32),
```

```
tf.cast(y_train, tf.float32)))
```

Kemudian, kita dapat mendefinisikan model untuk regresi linier sebagai $z = wx + b$. Di sini kita akan menggunakan API dari Keras. `tf.keras` menyediakan layer-layer yang telah didefinisikan sebelumnya (*predefined layers*) untuk membangun model NN yang kompleks, tetapi untuk memulai kita akan mempelajari bagaimana mendefinisikan sebuah model dari awal (*from scratch*). Nanti akan kita lihat bagaimana menggunakan *predefined layers* ini.

Untuk masalah regresi ini, kita akan definisikan sebuah class yang berasal dari `tf.keras.Model`. *Subclassing* dengan `tf.keras.Model` membuat kita dapat menggunakan tool-tool Keras untuk mengeksplorasi model, training, dan evaluasi. Pada konstruktor class, kita akan definisikan parameter-parameter dari model, yaitu w adalah parameter bobot dan b adalah parameter bias. Kemudian kita akan memanggil metode `.call()` untuk menentukan bagaimana model ini menggunakan data input dalam *generate* output-outputnya.

```
[118]: class MyModel(tf.keras.Model):
        def __init__(self):
            super(MyModel, self).__init__()
            self.w = tf.Variable(0.0, name='weight')
            self.b = tf.Variable(0.0, name='bias')

        def call(self, x):
            return self.w*x + self.b
```

- Selanjutnya kita akan membuat (*instantiate*) model baru dari class `MyModel()` yang bisa di-training berdasarkan data training. Keras API dari TensorFlow menyediakan sebuah metode `.summary()` untuk model-model yang dibuat dari `tf.keras.Model`, yang dapat digunakan untuk mendapatkan ringkasan (*summary*) dari komponen-komponen model layer demi layer dan jumlah parameter tiap layer. Karena kita telah membuat sub-class model kita dari `tf.keras.Model`, metode `.summary()` tersedia untuk kita gunakan juga. Tetapi, untuk dapat memanggil `model.summary()`, kita perlu mendefinisikan dimensi input (jumlah features) pada model ini dengan memanggil `model.build()` dengan *shape* data input yang diharapkan.

```
[119]: model = MyModel()

model.build(input_shape=(None, 1))
model.summary()
```

```
Model: "my_model_11"
```

```
-----
Layer (type)                 Output Shape          Param #
-----
Total params: 2
Trainable params: 2
Non-trainable params: 0
-----
```

Terlihat bahwa kita menggunakan `None` sebagai sebuah *placeholder* untuk dimensi pertama dari in-

put tensor yang diharapkan melalui `model.build` sehingga kita dapat menggunakan ukuran batch sembarang. Tetapi, jumlah feature adalah tetap (di sini 1) yang sesuai dengan jumlah parameter bobot dari model. Membuat layer-layer dan parameter-parameter model setelah *instantiation*, dengan memanggil metode `.build()` disebut dengan *late variable creation*. Untuk model sederhana ini, kita telah membuat parameter-parameter model di dalam konstruktor, sehingga menyatakan `input_shape` via `.build()` tidak ada efek lebih jauh terhadap parameter-parameter ini. Tetapi tetap diperlukan jika kita ingin memanggil `model.summary`.

Setelah mendefinisikan model, kita dapat mendefinisikan *cost function* yang kita ingin minimalkan untuk mencari pembobot optimal. Di sini, akan kita gunakan *mean squared error* (MSE) sebagai cost function. Lebih jauh, untuk *learn* parameter-parameter model, kita akan gunakan *Stochastic Gradient Descent* (SGD). Pada bagian ini, kita akan mengimplementasikan sendiri training dengan prosedur SGD. Tetapi, pada bagian selanjutnya kita akan menggunakan metode-metode Keras yaitu `compile()` dan `fit()` untuk melakukan hal yang sama.

- Untuk mengimplementasikan algoritma SGD, kita perlu menghitung gradien-gradien. Kita tidak akan menghitung secara manual gradien-gradien tersebut, tetapi akan menggunakan API TensorFlow yaitu `tf.GradientTape` (akan dijelaskan pada bab selanjutnya).

```
[120]: def loss_fn(y_true, y_pred):
        return tf.reduce_mean(tf.square(y_true - y_pred))
        ## testing the function:
        yt = tf.convert_to_tensor([1.0])
        yp = tf.convert_to_tensor([1.5])

        loss_fn(yt, yp)
```

```
[120]: <tf.Tensor: shape=(), dtype=float32, numpy=0.25>
```

```
[121]: def train(model, inputs, outputs, learning_rate):
        with tf.GradientTape() as tape:
            current_loss = loss_fn(model(inputs), outputs)
            dW, db = tape.gradient(current_loss, [model.w, model.b])
            model.w.assign_sub(learning_rate * dW)
            model.b.assign_sub(learning_rate * db)
```

- Selanjutnya, akan kita set hyperparameter dan men-training model untuk 200 epoch. Kita akan membuat versi batch dari dataset dan memanggil metode `.repeat` dataset dengan `count=None`, yang akan menghasilkan pengulangan dataset tanpa batas.

```
[122]: def train(model, inputs, outputs, learning_rate):
        with tf.GradientTape() as tape:
            current_loss = loss_fn(model(inputs), outputs)
            dW, db = tape.gradient(current_loss, [model.w, model.b])
            model.w.assign_sub(learning_rate * dW)
            model.b.assign_sub(learning_rate * db)
```

```

[123]: tf.random.set_seed(1)

num_epochs = 200
log_steps = 100
learning_rate = 0.001
batch_size = 1
steps_per_epoch = int(np.ceil(len(y_train) / batch_size))

ds_train = ds_train_orig.shuffle(buffer_size=len(y_train))
ds_train = ds_train.repeat(count=None)
ds_train = ds_train.batch(1)

Ws, bs = [], []

for i, batch in enumerate(ds_train):
    if i >= steps_per_epoch * num_epochs:
        break
    Ws.append(model.w.numpy())
    bs.append(model.b.numpy())

    bx, by = batch
    loss_val = loss_fn(model(bx), by)

    train(model, bx, by, learning_rate=learning_rate)
    if i%log_steps==0:
        print('Epoch {:4d} Step {:2d} Loss {:.4f}'.format(
            int(i/steps_per_epoch), i, loss_val))

```

```

Epoch    0 Step    0 Loss 43.5600
Epoch   10 Step   100 Loss  0.7530
Epoch   20 Step   200 Loss 20.1759
Epoch   30 Step   300 Loss 23.3976
Epoch   40 Step   400 Loss  6.3481
Epoch   50 Step   500 Loss  4.6356
Epoch   60 Step   600 Loss  0.2411
Epoch   70 Step   700 Loss  0.2036
Epoch   80 Step   800 Loss  3.8177
Epoch   90 Step   900 Loss  0.9416
Epoch  100 Step  1000 Loss  0.7035
Epoch  110 Step  1100 Loss  0.0348
Epoch  120 Step  1200 Loss  0.5404
Epoch  130 Step  1300 Loss  0.1170
Epoch  140 Step  1400 Loss  0.1195
Epoch  150 Step  1500 Loss  0.0944
Epoch  160 Step  1600 Loss  0.4670
Epoch  170 Step  1700 Loss  2.0695

```

Epoch 180 Step 1800 Loss 0.0020

Epoch 190 Step 1900 Loss 0.3612

- Kita akan melihat model yang telah ditraining dan melakukan *plotting*. Untuk data tes, kita akan membuat dengan array NumPy dengan harga-harga dengan jarak sama antara 0 – 9. Karena kita telah men-training model dengan features yang terstandarisasi (*standardized*), kita akan mengimplementasikan standarisasi yang sama pada data test.

```
[124]: print('Final Parameters:', model.w.numpy(), model.b.numpy())

X_test = np.linspace(0, 9, num=100).reshape(-1, 1)
X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)

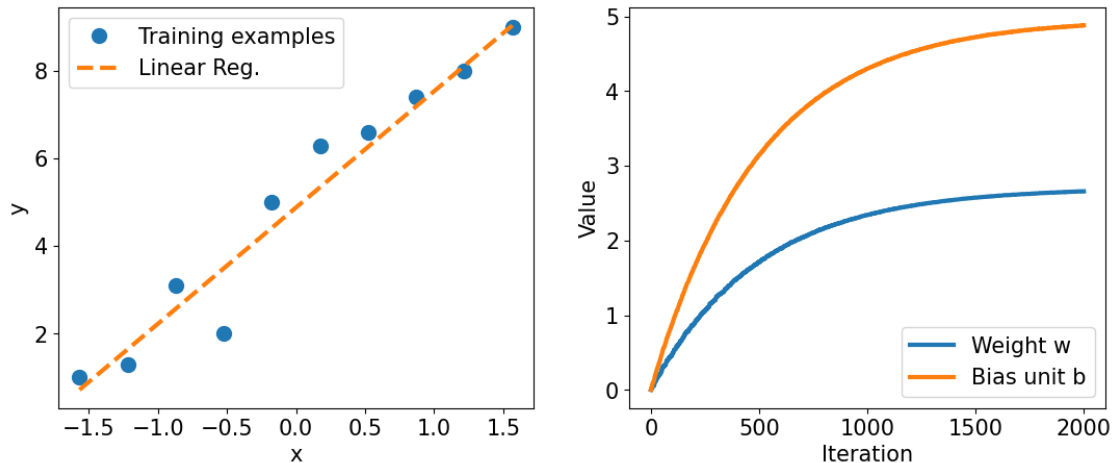
y_pred = model(tf.cast(X_test_norm, dtype=tf.float32))

fig = plt.figure(figsize=(13, 5))
ax = fig.add_subplot(1, 2, 1)
plt.plot(X_train_norm, y_train, 'o', markersize=10)
plt.plot(X_test_norm, y_pred, '--', lw=3)
plt.legend(['Training examples', 'Linear Reg.'], fontsize=15)
ax.set_xlabel('x', size=15)
ax.set_ylabel('y', size=15)
ax.tick_params(axis='both', which='major', labelsize=15)

ax = fig.add_subplot(1, 2, 2)
plt.plot(Ws, lw=3)
plt.plot(bs, lw=3)
plt.legend(['Weight w', 'Bias unit b'], fontsize=15)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Value', size=15)
ax.tick_params(axis='both', which='major', labelsize=15)
#plt.savefig('ch13-linreg-1.pdf')

plt.show()
```

Final Parameters: 2.6576622 4.8798566



Gambar yang dihasilkan setelah mengeksekusi kode-kode terakhir, di sebelah kiri menunjukkan *scatterplot* dari data training dan model regresi linier yang telah dihasilkan. Sedangkan di sebelah kanan menunjukkan proses konvergen dari pembobot w dan unit bias b .

1.2 Training Model via `.compile()` dan `.fit()`

Pada contoh sebelumnya, kita telah mempelajari bagaimana melakukan training sebuah model dengan menuliskan fungsi `train()`, dan mengimplementasikan metode optimasi SGD. Tetapi, menuliskan fungsi `train()` bisa menjadi pekerjaan berulang untuk proyek-proyek berbeda. Keras API pada TensorFlow telah menyediakan metode training yaitu `.fit()` yang dapat dipanggil pada model yang telah dibentuk (*instantiated*). Untuk melihat bagaimana hal ini bekerja, kita akan membuat model dan mengkompilasinya dengan memilih *optimizer*, *loss function* dan metrik evaluasi.

```
[125]: tf.random.set_seed(1)
model = MyModel()
#model.build((None, 1))

model.compile(optimizer='sgd',
              loss=loss_fn,
              metrics=['mae', 'mse'])
```

- Sekarang dapat kita panggil metode `fit()` untuk men-training model. Kita bisa melewati dataset yang telah di-*batch* (seperti `ds_train` yang telah dibuat di contoh sebelumnya). Tetapi kali ini, kita bisa melewati array NumPy untuk x dan y secara langsung tanpa harus membuat sebuah dataset.

```
[ ]: model.fit(X_train_norm, y_train,
              epochs=num_epochs, batch_size=batch_size,
              verbose=1)
```

- Setelah model di-training seperti di atas, kita dapat memvisualisasikan hasilnya dan pastikan

bahwa hasilnya sama dengan metode sebelumnya.

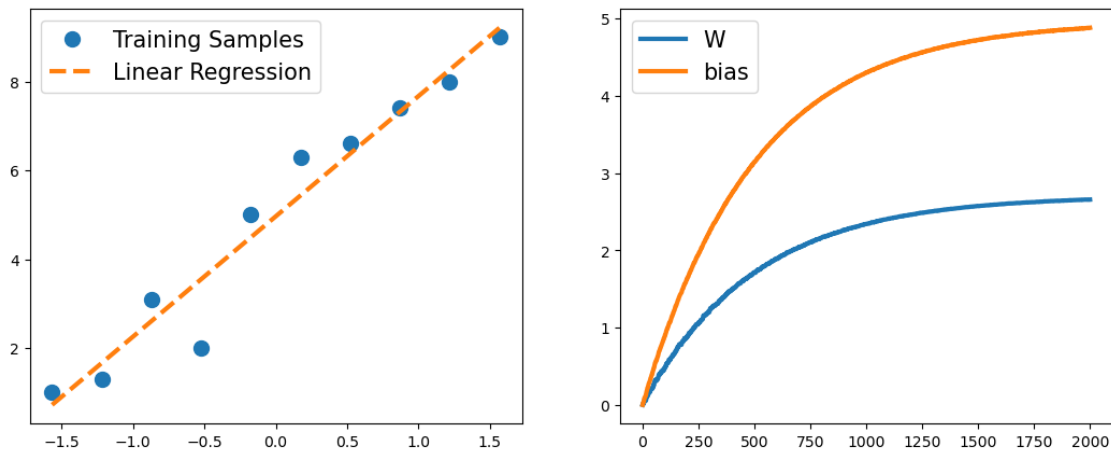
```
[127]: print(model.w.numpy(), model.b.numpy())

X_test = np.linspace(0, 9, num=100).reshape(-1, 1)
X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
y_pred = model(tf.cast(X_test_norm, dtype=tf.float32))

fig = plt.figure(figsize=(13, 5))
ax = fig.add_subplot(1, 2, 1)
plt.plot(X_train_norm, y_train, 'o', markersize=10)
plt.plot(X_test_norm, y_pred, '--', lw=3)
plt.legend(['Training Samples', 'Linear Regression'], fontsize=15)

ax = fig.add_subplot(1, 2, 2)
plt.plot(Ws, lw=3)
plt.plot(bs, lw=3)
plt.legend(['W', 'bias'], fontsize=15)
plt.show()
```

2.7067394 4.9657216



1.3 Membangun Multilayer Perceptron untuk Mengklasifikasikan Bunga pada Dataset Iris

Pada contoh sebelumnya, kita telah melihat bagaimana membuat sebuah model dari awal dan model di-training menggunakan optimasi SGD. Meskipun kita mulai dengan contoh paling sederhana, kita dapat melihat bahwa mendefinisikan model dari awal tidak menarik dan tidak praktis. TensorFlow sebetulnya menyediakan layer-layer yang telah terdefinisi melalui `tf.keras.layers` yang dapat digunakan untuk blok bangunan (*building blocks*) dari model NN. Pada bagian ini, kita akan mempelajari bagaimana menggunakan layer-layer ini dalam memecahkan permasalahan

klasifikasi menggunakan dataset Iris dan membangun perceptron dua layer menggunakan Keras API.

- Kita mulai dengan mendapatkan data dari `tensorflow_datasets`.

```
[ ]: import tensorflow_datasets as tfds
iris, iris_info = tfds.load('iris', with_info=True)
print(iris_info)
```

Informasi hasil eksekusi kode di atas menunjukkan bahwa dataset ini hanya mempunyai satu partisi, sehingga kita harus memisahkan dataset tersebut ke dalam partisi training dan testing. Kita asumsikan bahwa dua pertiga dari dataset akan digunakan sebagai training dan sisanya untuk testing. Library `tensorflow_datasets` menyediakan tool yang baik untuk menentukan *slices* dan *splits* via objek `DatasetBuilder` sebelum *loading* dataset. Lebih jauh menyangkut *splits* dapat ditemukan di link berikut: <https://www.tensorflow.org/datasets/splits>.

Pendekatan lain adalah dengan pertama kali me-*load* keseluruhan dataset, kemudian gunakan `.take()` dan `.skip()` untuk men-*split* dataset ke dalam dua partisi. Jika dataset tersebut tidak di-*shuffle* saat pertama kali, kita juga bisa melakukan *shuffle*. Tetapi, kita harus hati-hati karena bisa jadi akan mencampur data testing dan training yang harus dihindari pada machine learning. Untuk menghindarinya, kita harus melakukan *setting* argumen `reshuffle_each_iteration=False`, pada metode `.shuffle()`.

- Kode untuk *splitting* dataset ke dalam training dan test sebagai berikut.

```
[129]: tf.random.set_seed(1)

ds_orig = iris['train']
ds_orig = ds_orig.shuffle(150, reshuffle_each_iteration=False)

print(next(iter(ds_orig)))

ds_train_orig = ds_orig.take(100)
ds_test = ds_orig.skip(100)
```

```
{'features': <tf.Tensor: shape=(4,), dtype=float32, numpy=array([6.5, 3. , 5.2,
2. ], dtype=float32)>, 'label': <tf.Tensor: shape=(), dtype=int64, numpy=2>}
```

- Melihat jumlah *examples* pada training dan testing.

```
[130]: ## checking the number of examples:

n = 0
for example in ds_train_orig:
    n += 1
print(n)

n = 0
for example in ds_test:
```

```
n += 1
print(n)
```

```
100
50
```

- Selanjutnya, kita telah melihatnya pada bagian sebelumnya, yaitu kita perlu menerapkan transformasi via metode `.map()` untuk merubah dictionary ke sebuah tuple.

```
[131]: ds_train_orig = ds_train_orig.map(
        lambda x: (x['features'], x['label']))

ds_test = ds_test.map(
    lambda x: (x['features'], x['label']))

next(iter(ds_train_orig))
```

```
[131]: (<tf.Tensor: shape=(4,), dtype=float32, numpy=array([6.5, 3. , 5.2, 2. ],
dtype=float32)>,
      <tf.Tensor: shape=(), dtype=int64, numpy=2>)
```

Sekarang kita sudah siap untuk menggunakan API Keras untuk membangun model secara efisien. Khususnya, dengan menggunakan class `tf.keras.Sequential` kita dapat menumpuk beberapa layer-layer Keras yang sudah tersedia di link berikut https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers. Di sini kita akan menggunakan layer `Dense` yaitu `tf.keras.layers.Dense` yang juga dikenal sebagai *fully connected layer* atau *linear layer* dengan representasi terbaik menggunakan $f(w \times x + b)$. Dimana x adalah features input, w dan b adalah matriks pembobot dan vektor bias, dan f menyatakan fungsi aktivasi.

Masing-masing layer pada NN menerima input-input dari layer sebelumnya, oleh sebab itu dimensinya (*rank* dan *shape*) adalah tetap. Biasanya, kita perlu memperhatikan dimensi output hanya ketika kita mendesain arsitektur NN. Pada contoh ini, kita ingin mendefinisikan model dengan dua hidden layer, dimana yang pertama menerima input dari empat features dan memproyeksikannya ke dalam 16 neurons. Hidden layer kedua menerima input dari layer sebelumnya (dengan ukuran 16) dan memproyeksikannya ke dalam tiga neuron output karena kita mempunyai label tiga class.

- Model tersebut bisa kita buat dengan menggunakan class `Sequential` dan layer `Dense` pada 'Keras menggunakan kode berikut.

```
[132]: iris_model = tf.keras.Sequential([
        tf.keras.layers.Dense(16, activation='sigmoid',
                               name='fc1', input_shape=(4,)),
        tf.keras.layers.Dense(3, name='fc2', activation='softmax')])

iris_model.summary()
```

```
Model: "sequential_3"
```

```
-----#
Layer (type)                Output Shape          Param #
```

```

=====
fc1 (Dense)                (None, 16)                80

fc2 (Dense)                (None, 3)                 51

=====
Total params: 131
Trainable params: 131
Non-trainable params: 0
-----

```

Dapat kita perhatikan dari kode di atas, *shape* dari input pada layer pertama ditentukan via `input_shape=(4,)`, oleh sebab itu kita tidak perlu memanggil fungsi `.build()` lagi untuk menggunakan `iris_model_summary()`.

Summary dari model mengindikasikan bahwa layer pertama (`fc1`) mempunyai 80 parameter, dan layer kedua mempunyai 51 parameter. Kita dapat memverifikasi jumlah tersebut dengan $(n_{in} + 1) \times n_{out}$, dimana n_{in} adalah jumlah unit input, dan n_{out} adalah jumlah unit output. Pada *fully (densely) connected layer*, parameter yang harus ditentukan saat training adalah matriks pembobot dengan ukuran $n_{in} \times n_{out}$ dan vektor bias dengan ukuran n_{out} . Lebih jauh, kita gunakan fungsi aktivasi sigmoid untuk layer pertama dan fungsi aktivasi softmax untuk layer output. Fungsi aktivasi softmax pada layer terakhir digunakan untuk mendukung klasifikasi multi-class karena kita mempunyai tiga label class pada luaran. Fungsi-fungsi aktivasi yang lain akan dibahas di bagian selanjutnya pada bab ini.

- Kita akan *compile* model ini dan menspesifikasikan *loss function*, *optimizer* dan juga metrik untuk evaluasi.

```
[133]: iris_model.compile(optimizer='adam',
                        loss='sparse_categorical_crossentropy',
                        metrics=['accuracy'])
```

- Selanjutnya kita akan men-training model dengan jumlah epoch 100 dan ukuran batch 2. Pada kode berikut, kita akan membangun dataset yang berulang yang kemudian akan dilewatkan ke metode `.fit()` untuk mentraining model. Pada kasus ini, supaya metode `.fit()` dapat melacak epoch-epoch maka metode tersebut perlu mengetahui jumlah *steps* untuk tiap epoch. Jika diberikan ukuran data training (100) dan ukuran batch (`batch_size`), kita dapat menentukan jumlah *steps* pada setiap epoch (`step_per_epoch`).

```
[134]: num_epochs = 100
training_size = 100
batch_size = 2
steps_per_epoch = np.ceil(training_size / batch_size)

ds_train = ds_train_orig.shuffle(buffer_size=training_size)
ds_train = ds_train.repeat()
ds_train = ds_train.batch(batch_size=batch_size)
ds_train = ds_train.prefetch(buffer_size=1000)
```

```
history = iris_model.fit(ds_train, epochs=num_epochs,
                        steps_per_epoch=steps_per_epoch,
                        verbose=0)
```

Returned variable `history` menjaga *training loss* dan *training accuracy* (karena dispesifikasikan sebagai metrik pada `iris_model.compile()`) pada setiap epoch.

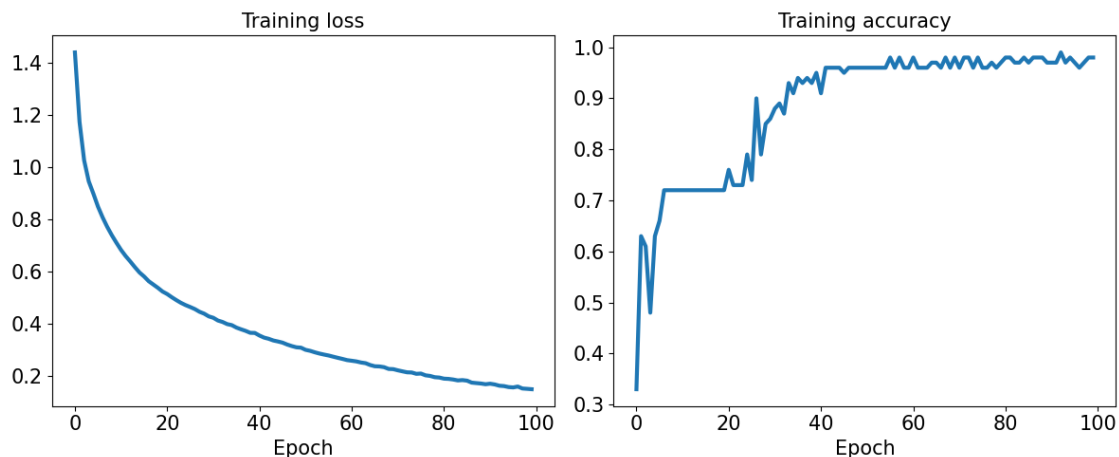
- Kita akan menggunakan kode berikut untuk memvisualisasikan *learning curve*.

```
[135]: hist = history.history

fig = plt.figure(figsize=(12, 5))
ax = fig.add_subplot(1, 2, 1)
ax.plot(hist['loss'], lw=3)
ax.set_title('Training loss', size=15)
ax.set_xlabel('Epoch', size=15)
ax.tick_params(axis='both', which='major', labelsize=15)

ax = fig.add_subplot(1, 2, 2)
ax.plot(hist['accuracy'], lw=3)
ax.set_title('Training accuracy', size=15)
ax.set_xlabel('Epoch', size=15)
ax.tick_params(axis='both', which='major', labelsize=15)
plt.tight_layout()
#plt.savefig('ch13-cls-learning-curve.pdf')

plt.show()
```



1.4 Evaluasi Model Hasil Training dengan Dataset Test

- Karena kita telah menspesifikasikan `accuracy` sebagai metrik evaluasi pada `iris_model.compile()`, sekarang kita dapat langsung mengevaluasi model pada dataset testing.

```
[136]: results = iris_model.evaluate(ds_test.batch(50), verbose=0)
print('Test loss: {:.4f}    Test Acc.: {:.4f}'.format(*results))
```

```
Test loss: 0.1483    Test Acc.: 0.9800
```

Dapat kita perhatikan bahwa kita perlu untuk mem-*batch*-kan juga dataset testing untuk menjamin bahwa input pada model mempunyai dimensi yang tepat (rank). Seperti yang telah kita diskusikan sebelumnya, memanggil metode `.batch()` akan meningkatkan rank dari tensor yang diperoleh (rank bertambah 1). Data input untuk metode `.evaluate()` harus mempunyai satu dimensi yang telah ditentukan untuk batch, meskipun di sini ukuran batch tidak menjadi masalah. Oleh karena itu, jika kita melewatkan `ds_batch.batch(50)` pada metode `.evaluate()`, keseluruhan dataset testing akan diproses dengan batch berukuran 50. Tetapi jika kita melewatkan `ds_batch.batch(1)` maka 50 batch dengan ukuran 1 akan diproses.

1.5 Menyimpan dan Reloading Model yang Telah Dittraining

- Model yang telah ditraining dapat disimpan pada disk untuk digunakan selanjutnya dengan kode berikut.

```
[137]: iris_model.save('iris-classifier.h5',
                    overwrite=True,
                    include_optimizer=True,
                    save_format='h5')
```

Opsi pertama adalah nama file. Memanggil metode `iris_model.save()` akan menyimpan keduanya, baik arsitektur model maupun parameter-parameter yang telah ditentukan dengan training. Tetapi, jika kita ingin menyimpan arsitekturnya saja, maka dapat digunakan metode `iris_model.to_json()` yang akan menyimpan konfigurasi model dengan format JSON. Atau jika hanya ingin menyimpan pembobot-pembobot model, dapat kita gunakan `iris_model.save_weights()`. Sedangkan `save_format` dapat dispesifikasikan menjadi `h5` untuk format HDF5 atau `tf` untuk format TensorFlow.

- Untuk me-reload model yang telah disimpan, dimana kita telah menyimpan arsitektur model dan juga pembobot, dan sekaligus memverifikasinya dengan melihat summary, maka dapat digunakan kode berikut.

```
[138]: iris_model_new = tf.keras.models.load_model('iris-classifier.h5')

iris_model_new.summary()
```

```
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
=====		

fc1 (Dense)	(None, 16)	80
fc2 (Dense)	(None, 3)	51

```
=====
Total params: 131
Trainable params: 131
Non-trainable params: 0
-----
```

- Selanjutnya, kita dapat mengevaluasi model baru yang telah di-*reload* menggunakan dataset testing untuk menverifikasi bahwa hasilnya sama dengan sebelumnya.

```
[ ]: results = iris_model_new.evaluate(ds_test.batch(50), verbose=0)
print('Test loss: {:.4f}    Test Acc.: {:.4f}'.format(*results))
```

```
[140]: labels_train = []
for i,item in enumerate(ds_train_orig):
    labels_train.append(item[1].numpy())

labels_test = []
for i,item in enumerate(ds_test):
    labels_test.append(item[1].numpy())
print('Training Set: ',len(labels_train), 'Test Set: ', len(labels_test))
```

Training Set: 100 Test Set: 50

```
[141]: iris_model_new.to_json()
```

```
[141]: '{"class_name": "Sequential", "config": {"name": "sequential_3", "layers":
[{"class_name": "InputLayer", "config": {"batch_input_shape": [null, 4],
"dtype": "float32", "sparse": false, "ragged": false, "name": "fc1_input"}},
{"class_name": "Dense", "config": {"name": "fc1", "trainable": true,
"batch_input_shape": [null, 4], "dtype": "float32", "units": 16, "activation":
"sigmoid", "use_bias": true, "kernel_initializer": {"class_name":
"GlorotUniform", "config": {"seed": null}}, "bias_initializer": {"class_name":
"Zeros", "config": {}}, "kernel_regularizer": null, "bias_regularizer": null,
"activity_regularizer": null, "kernel_constraint": null, "bias_constraint":
null}}, {"class_name": "Dense", "config": {"name": "fc2", "trainable": true,
"dtype": "float32", "units": 3, "activation": "softmax", "use_bias": true,
"kernel_initializer": {"class_name": "GlorotUniform", "config": {"seed": null}},
"bias_initializer": {"class_name": "Zeros", "config": {}}, "kernel_regularizer":
null, "bias_regularizer": null, "activity_regularizer": null,
"kernel_constraint": null, "bias_constraint": null}}]}, "keras_version":
"2.9.0", "backend": "tensorflow"}'
```

2 Memilih Fungsi Aktivasi untuk Multilayer Neural Networks

Di bagian sebelumnya kita telah mempelajari fungsi aktivasi fungsi sigmoid (logistik) $\sigma(z) = \frac{1}{1 + e^{-z}}$. Pada bagian ini kita akan mempelajari lebih jauh menyangkut fungsi-fungsi non-linier lain yang berguna untuk mengimplementasikan NNs. Secara teknis kita dapat menggunakan fungsi sembarang sebagai fungsi aktivasi pada multilayer NN selama bisa diturunkan (*differentiable*). Bahkan kita bisa menggunakan fungsi linier seperti yang kita pakai di Adaline. Tetapi, penggunaan fungsi linier tidak berguna secara praktis, karena penjumlahan fungsi-fungsi linier akan menghasilkan fungsi linier juga. Non-linieritas diperkenalkan ke dalam ANN supaya bisa menghasilkan solusi untuk masalah-masalah kompleks.

Fungsi aktivasi logistik yang digunakan pada bab sebelumnya mungkin yang paling dekat menyerupai konsep neuron di otak. Tetapi, fungsi aktivasi sigmoid mempunyai permasalahan jika input mempunyai nilai sangat negatif sehingga probabilitas mendekati nol. Jika fungsi sigmoid menghasilkan output yang dekat dengan nol, proses learning pada NN akan berjalan sangat lambat dan lebih mungkin terjebak pada minimum lokal. Dengan alasan inilah fungsi tangen hiperbolik lebih banyak dipilih sebagai fungsi aktivasi pada hidden layer.

Sebelum kita mendiskusikan seperti apa fungsi tangen hiperbolik, rekapitulasi dari fungsi logistik akan dijelaskan kembali secara singkat. Generalisasi dari fungsi tersebut agar lebih berguna untuk masalah-masalah klasifikasi multi-class akan dijelaskan kemudian.

2.1 Fungsi Logistik

Fungsi logistik merupakan kasus spesial dari fungsi sigmoid. Dari bab sebelumnya, fungsi logistik dapat digunakan untuk memodelkan probabilitas bahwa sebuah sampel x milik dari class positif pada kasus *binary classification*.

Net input z merupakan kombinasi linier dari input-input dengan persamaan berikut ini.

$$z = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^m w_ix_i = \mathbf{w}^T \mathbf{x}$$

Fungsi logistik akan menghitung kuantitas berikut:

$$\phi_{logistik}(z) = \frac{1}{1 + e^{-z}}$$

Sebagai catatan w_0 adalah unit bias dengan $x_0 = 1$.

- Untuk melihat hal yang lebih konkrit, kita akan membuat sebuah model untuk data dua dimensi x dan sebuah model untuk vektor pembobot \mathbf{w} .

```
[142]: import numpy as np

X = np.array([1, 1.4, 2.5]) ## first value must be 1
w = np.array([0.4, 0.3, 0.5])

def net_input(X, w):
```

```

    return np.dot(X, w)

def logistic(z):
    return 1.0 / (1.0 + np.exp(-z))

def logistic_activation(X, w):
    z = net_input(X, w)
    return logistic(z)

print('P(y=1|x) = %.3f' % logistic_activation(X, w))

```

$P(y=1|x) = 0.888$

Jika kita hitung net input (z) dan menggunakannya untuk mengaktivasi sebuah neuron logistik dengan harga-harga feature dan koefisien pembobot seperti pada kode di atas, maka kita akan mendapatkan harga 0.88. Dari hasil tersebut dapat diinterpretasikan bahwa probabilitas bahwa x berasal dari positif class adalah 0.88 atau 88.8%.

- Dari bab sebelumnya kita gunakan teknik *one-hot encoding* untuk merepresentasikan label-label *ground truth* dari multi-class dan mendesain layer output yang terdiri dari beberapa unit aktivasi logistik. Tetapi, seperti yang terlihat pada kode berikut, layer output dengan unit aktivasi logistik tidak akan menghasilkan harga-harga probabilitas yang mempunyai arti atau dapat diinterpretasikan.

```

[143]: # W : array with shape = (n_output_units, n_hidden_units+1)
# note that the first column are the bias units

W = np.array([[1.1, 1.2, 0.8, 0.4],
              [0.2, 0.4, 1.0, 0.2],
              [0.6, 1.5, 1.2, 0.7]])

# A : data array with shape = (n_hidden_units + 1, n_samples)
# note that the first column of this array must be 1

A = np.array([[1, 0.1, 0.4, 0.6]])
Z = np.dot(W, A[0])
y_probas = logistic(Z)

print('Net Input: \n', Z)
print('Output Units:\n', y_probas)

```

Net Input:

[1.78 0.76 1.65]

Output Units:

[0.85569687 0.68135373 0.83889105]

Seperti yang kita lihat pada output, harga-harga tersebut tidak dapat diinterpretasikan sebagai harga probabilitas untuk masalah klasifikasi dengan tiga class, karena ketika dijumlahkan tidak akan sama dengan 1. Tetapi, hal ini tidak menjadi masalah, jika kita menggunakan model hanya

untuk memprediksi label class dan bukan probabilitas keanggotaan dari class.

- Cara untuk melakukan prediksi label class dari unit output yang diperoleh sebelumnya dapat menggunakan harga maksimum seperti berikut.

```
[144]: y_class = np.argmax(Z, axis=0)
print('Predicted class label: %d' % y_class)
```

Predicted class label: 0

2.2 Estimasi Probabilitas Class dari Klasifikasi Multi-Class via Fungsi Softmax

Pada bagian sebelumnya kita telah melihat bagaimana kita memperoleh label class dengan menggunakan fungsi `argmax`. Di bab sebelumnya, kita juga menggunakan fungsi aktivasi `activation=softmax` pada layer terakhir model MLP. Fungsi `softmax` merupakan bentuk *soft* dari fungsi `argmax`, dimana fungsi `softmax` memberikan hasil probabilitas tiap class bukan indeks class. Dengan demikian, fungsi tersebut dapat digunakan untuk menghitung probabilitas keanggotaan dari class (jumlah keseluruhan menjadi 1) pada skenario klasifikasi multi-class (*multinomial logistic regression*).

Pada `softmax`, probabilitas untuk sampel tertentu dengan net input z berasal dari class ke- i , dapat dihitung dengan normalisasi pada penyebut (*denominator*) dengan penjumlahan fungsi linier yang diboboti secara eksponensial.

$$p(z) = \phi(z) = \frac{e^{Z_i}}{\sum_{j=i}^M e^{Z_j}}$$

- Berikut kode program `softmax` sebagai contoh.

```
[145]: def softmax(z):
        return np.exp(z) / np.sum(np.exp(z))

y_probas = softmax(Z)
print('Probabilities:\n', y_probas)

np.sum(y_probas)
```

Probabilities:
[0.44668973 0.16107406 0.39223621]

```
[145]: 1.0
```

Dapat dilihat pada hasil kode di atas, probabilitas class-class terprediksi apabila dijumlahkan akan sama dengan 1 seperti yang kita harapkan. Dapat dilihat juga bahwa prediksi label class hasilnya sama dengan hasil dari fungsi `argmax` dari output logistik (probabilitas terbesar adalah class 0).

Dengan demikian, ketika kita akan membangun model klasifikasi multi-class pada `TensorFlow`, maka kita dapat menggunakan fungsi `tf.keras.activations.softmax()` untuk mengestimasi masing-masing probabilitas keanggotaan class. Untuk melihat bagaimana kita menggunakan fungsi aktivasi

softmax pada TensorFlow, kode berikut merubah z menjadi sebuah tensor, dengan tambahan dimensi yang dipesan untuk *batch size*.

```
[146]: import tensorflow as tf
```

```
Z_tensor = tf.expand_dims(Z, axis=0)
tf.keras.activations.softmax(Z_tensor)
```

```
[146]: <tf.Tensor: shape=(1, 3), dtype=float64, numpy=array([[0.44668973, 0.16107406,
0.39223621]])>
```

2.3 Fungsi Tangen Hiperbolik (*Hyperbolic Tangent*)

Fungsi sigmoid lain yang sering digunakan pada hidden layer dari ANNs adalah fungsi hiperbolik, atau biasanya dikenal dengan \tanh , yang bisa diinterpretasikan sebagai versi penskalaan (*rescaled version*) dari fungsi logistik.

$$\phi_{logistik} = \frac{1}{1 + e^{-z}}$$
$$\phi_{tanh} = 2 \times x\phi_{logistik} - 1 = \frac{e^Z - e^{-Z}}{e^Z + e^{-Z}} \quad (1)$$

Keuntungan dari tangen hiperbolik dibandingkan dengan fungsi logistik adalah tangen hiperbolik mempunyai spektrum output yang lebih lebar dengan range antara $(-1, 1)$ yang dapat memperbaiki konvergensi dari algoritma backpropagation. Berbeda dengan fungsi logistik yang memberikan hasil output pada interval $(0, 1)$

- Untuk membandingkan fungsi logistik dan tangen hiperbolik, berikut adalah plotting dari kedua fungsi sigmoid tersebut.

```
[147]: import matplotlib.pyplot as plt
```

```
%matplotlib inline
```

```
def tanh(z):
```

```
    e_p = np.exp(z)
```

```
    e_m = np.exp(-z)
```

```
    return (e_p - e_m) / (e_p + e_m)
```

```
z = np.arange(-5, 5, 0.005)
```

```
log_act = logistic(z)
```

```
tanh_act = tanh(z)
```

```
plt.ylim([-1.5, 1.5])
```

```
plt.xlabel('Net input $z$')
```

```
plt.ylabel('Activation $\phi(z)$')
```

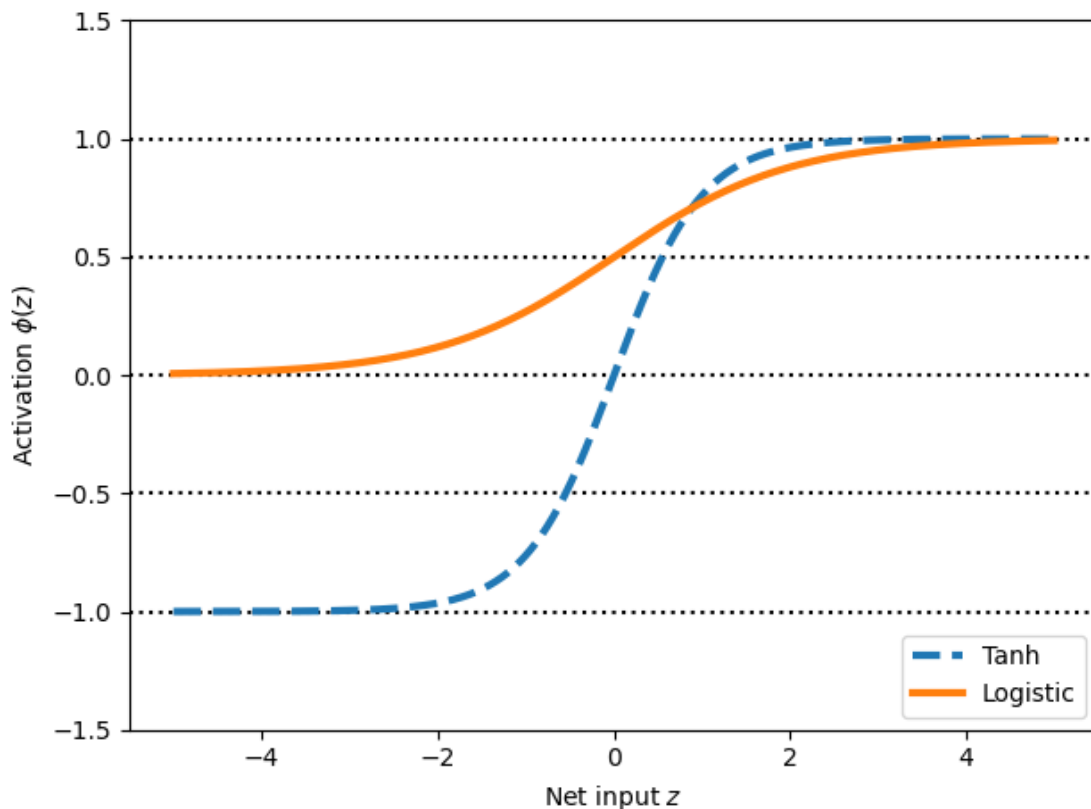
```
plt.axhline(1, color='black', linestyle=':')
```

```
plt.axhline(0.5, color='black', linestyle=':')
```

```
plt.axhline(0, color='black', linestyle=':')
```

```
plt.axhline(-0.5, color='black', linestyle=':')
```

```
plt.axhline(-1, color='black', linestyle=':')
plt.plot(z, tanh_act,
         linewidth=3, linestyle='--',
         label='Tanh')
plt.plot(z, log_act,
         linewidth=3,
         label='Logistic')
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()
```



Dapat kita lihat di hasil gambar bahwa kedua fungsi sigmoid tersebut cukup serupa, tetapi fungsi `tanh` mempunyai ruang dua kali lebih besar dibandingkan dengan fungsi `logistic`. Secara alternatif, kita bisa menggunakan fungsi `tanh` dari NumPy.

```
[148]: np.tanh(z)
```

```
[148]: array([-0.9999092 , -0.99990829, -0.99990737, ...,  0.99990644,
              0.99990737,  0.99990829])
```

- Sebagai opsi lain ketika membangun model NN, kita juga dapat menggunakan fungsi

`tf.keras.activations.tanh()` dari TensorFlow untuk memperoleh hasil yang sama.

```
[149]: import tensorflow as tf
```

```
tf.keras.activations.tanh(z)
```

```
[149]: <tf.Tensor: shape=(2000,), dtype=float64, numpy=
array([-0.9999092 , -0.99990829, -0.99990737, ...,  0.99990644,
        0.99990737,  0.99990829])>
```

- Sebagai tambahan, fungsi logistik juga tersedia pada modul `special` dari `SciPy`.

```
[150]: from scipy.special import expit
```

```
expit(z)
```

```
[150]: array([0.00669285, 0.00672617, 0.00675966, ..., 0.99320669, 0.99324034,
        0.99327383])
```

- Kita juga bisa menggunakan fungsi `tf.keras.activations.sigmoid()` dari TensorFlow untuk menghitung hasil yang sama sebagai berikut.

```
[151]: tf.keras.activations.sigmoid(z)
```

```
[151]: <tf.Tensor: shape=(2000,), dtype=float64, numpy=
array([0.00669285, 0.00672617, 0.00675966, ..., 0.99320669, 0.99324034,
        0.99327383])>
```

2.4 Aktivasi dengan *Rectified Linear Unit* (Re-LU)

Rectified Linear Unit (ReLU) adalah fungsi aktivasi lain yang sering digunakan pada *Deep Neural Networks* (DNN). Sebelum membahas ReLU, kita harus memahami masalah *vanishing gradient problem* dari fungsi aktivasi `tanh` dan logistik.

Kita akan asumsikan bahwa awalnya kita mempunyai net input $z_1 = 20$ yang berubah ke $z_2 = 25$. Dengan menghitung aktivasi `tanh` kita akan peroleh $\phi(z_1) = 1.0$ dan $\phi(z_2) = 1.0$ yang tidak menghasilkan perubahan pada output (akibat perilaku asimptotik dari fungsi `tanh` dan error numerik). Artinya, turunan dari fungsi aktivasi ini terhadap net input z akan semakin mengecil (menghilang) dengan z semakin membesar. Oleh sebab itu, proses *learning* dari pembobot selama fase training akan sangat lambat karena bagian gradien akan semakin mendekati nol. Fungsi aktivasi ReLU memberikan solusi untuk masalah ini. Secara matematis ReLU didefinisikan sebagai berikut.

$$\phi(z) = \max(0, z)$$

ReLU masih bersifat fungsi non-linier yang baik untuk *learning* fungsi-fungsi kompleks dengan NNs. Selain itu, turunan ReLU terhadap input selalu bernilai 1 untuk input bernilai positif. Oleh sebab itu, ReLU dapat memecahkan masalah *vanishing gradient* sehingga cocok digunakan untuk DNN.

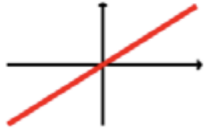





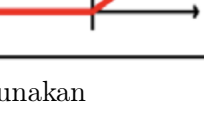
- Pada TensorFlow, kita dapat menggunakan fungsi aktivasi ReLU dengan cara berikut.

```
[152]: import tensorflow as tf
```

```
tf.keras.activations.relu(z)
```

```
[152]: <tf.Tensor: shape=(2000,), dtype=float64, numpy=array([0.    , 0.    , 0.    , ...,
4.985, 4.99 , 4.995])>
```

Sampai sejauh ini kita telah mengetahui fungsi-fungsi aktivasi yang berbeda yang sering digunakan pada ANN. Sebagai tambahan informasi Gambar 3.1 berikut menunjukkan fungsi-fungsi aktivasi yang akan digunakan pada bab-bab selanjutnya.

Activation function	Equation	Example	1D graph
Linear	$\phi(z) = z$	Adaline, linear regression	
Unit step (Heaviside function)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise linear	$\phi(z) = \begin{cases} 0 & z \leq -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \leq z \leq \frac{1}{2} \\ 1 & z \geq \frac{1}{2} \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, multilayer NN	
Hyperbolic tangent (tanh)	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\phi(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Multilayer NN, CNNs	

Gambar 3.2: Fungsi-fungsi aktivasi yang akan digunakan