# Sieci neuronowe w Keras i Tensorflow

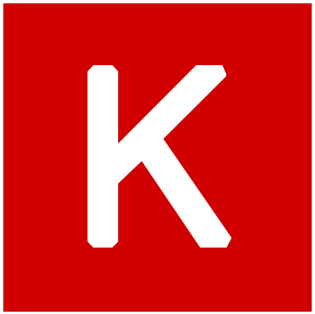
dr inż. Sebastian Ernst

Przedmiot: Uczenie Maszynowe

# Keras + TensorFlow

# Keras

- Interfejs Python do wygodnego modelowania sieci neuronowych
- Licencja Apache 2.0
- Za przetwarzanie odpowiedzialny jest backend:
  - do wersji 2.3 – TensorFlow, MS Cognitive Toolkit, Theano, PlaidML:
    ```
    import tensorflow as tf
    import keras
    ```
  - od wersji 2.4 – tylko TensorFlow:
    ```
    import tensorflow as tf
    from tensorflow import keras
    ```



# TensorFlow

- Biblioteka udostępniająca szereg narzędzi uczenia maszynowego, ale ukierunkowana na budowanie głębokich sieci neuronowych
- Stworzona przez Google
- Wsparcie dla GPU, dostępne wersje zoptymalizowane dla określonych środowisk sprzętowych (np. Apple Metal)

# Keras Sequential API

- Zakłada, że warstwy połączone są w sekwencję (szeregowo)
- Tworzenie modelu:

```
model = keras.models.Sequential()
```

- Warstwy dodajemy jako instancje odpowiednich klas z pakietu `keras.layers`:

```
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

# Keras Sequential API

- Parametry często przekazujemy jako ciągi znaków, jest to zapis uproszczony: zamiast np. `"relu"` możemy przekazać `keras.activations.relu`.
- Normalizację robimy sami (wcześniej), lub używamy warstwy Normalization:

```
normalizer = keras.layers.Normalization(
  input_shape=[1,], axis=None)
normalizer.adapt(y)
# ...
model.add(normalizer)
```

# Analiza modelu

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 300)               235500

 dense_1 (Dense)             (None, 100)               30100

 dense_2 (Dense)             (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

# Kompilacja modelu

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

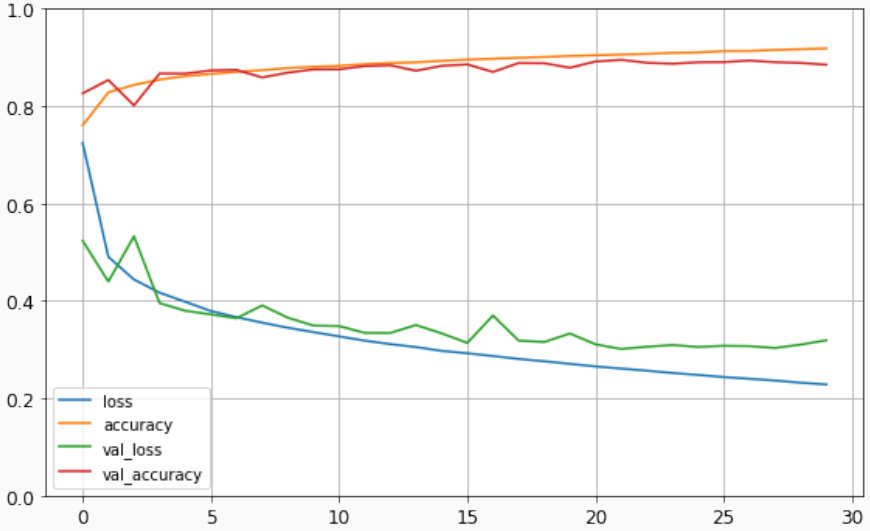
- Tu znów zamiast `"sgd"` moglibyśmy podać `keras.losses.sparse_categorical_crossentropy`.
- W przypadku metryki, Keras zachowa się „inteligentnie” i w oparciu o `"accuracy"` wybierze `keras.metrics.sparse_categorical_accuracy`.

# Uczenie modelu

```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

- Zamiast `validation_data` moglibyśmy podać `validation_split` (np. `0.2`); wtedy zbiór walidacyjny zostałby wydzielony automatycznie ze zbioru uczącego
- Jeżeli klasy nie są reprezentowane równomiernie w zbiorze danych, można zmienić ich wagi przy pomocy argumentu `class_weight`
- Podobnie, możemy „faworyzować" określone próbki przy pomocy `sample_weight`

# Analiza procesu uczenia



# Regresja przy pomocy sieci neuronowych

# Regresja w Sequential API

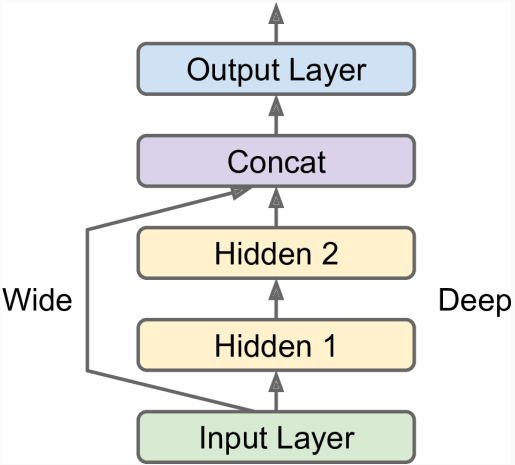
```
model = keras.models.Sequential([
    keras.layers.Dense(30,
                       activation="relu",
                       input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
```

- Nie stosujemy funkcji aktywacji (chcemy, aby neurony przekazywały wartości bezpośrednio)
- Funkcja straty odpowiednia dla regresji, np. MSE

# (Bardziej) zaawansowane funkcje TensorFlow

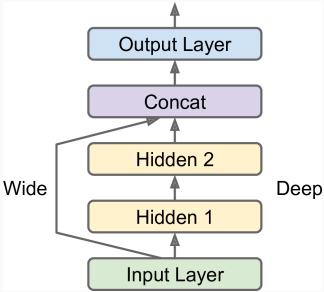
# Bardziej złożone struktury sieci

- Obiekty reprezentujące poszczególne warstwy wiążemy z innymi warstwami „ręcznie”
- Model tworzymy przy pomocy bardziej ogólnej klasy `keras.Model`, podając tylko warstwy wejściowe i wyjściowe



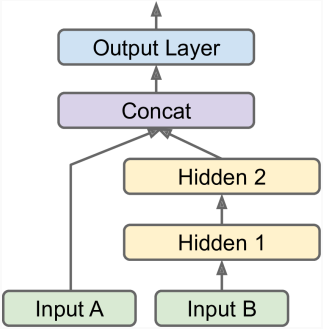
# Przykład: *Wide & Deep*

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input_], outputs=[output])
```



# Przykład: różne ścieżki dla różnych cech

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="main_output")(concat)
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
model = keras.models.Model(inputs=[input_A, input_B],
                           outputs=[output, aux_output])
```



# Keras Subclassing API

```
class WideAndDeepModel(keras.models.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output
```

# Zapisywanie i odtwarzanie modelu

- Zazwyczaj w formacie HDF5
- Zapisywanie:

```
model.compile(...)
model.fit(...)
model.save("my_keras_model.h5")
```

- Odtwarzanie:

```
model = keras.models.load_model("my_keras_model.h5")
model.predict(...)
```

# Korzystanie z callbacków

- Do metody `fit()` możemy przekazać listę callbacków
- Na przykład do zapisywania punktów kontrolnych:

```
checkpoint_cb = keras.callbacks.ModelCheckpoint(
  "my_keras_model.h5", save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])
```

- Bardzo częste zastosowanie – *early stopping*:

```
early_stopping_cb = keras.callbacks.EarlyStopping(
  patience=10,
  restore_best_weights=True)
```

# Tensorboard

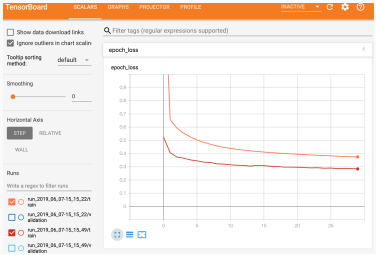
# Tensorboard

- Interaktywna wizualizacja procesu uczenia
- Komunikacja przy pomocy *event files*
- Osobny proces, uruchamia serwer HTTP:

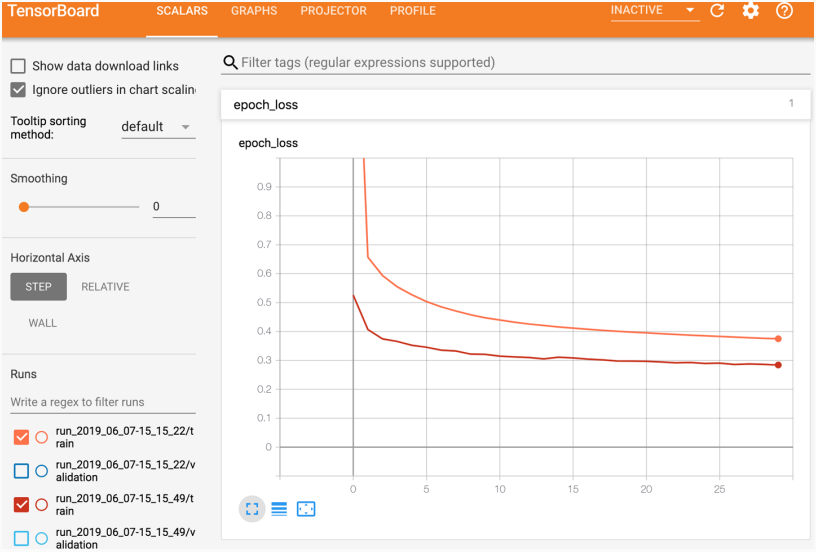
  ```
  $ tensorboard --logdir=./logs --port=6006
  ```

- Można wykorzystać w Jupyterze:

  ```
  %load_ext tensorboard
  %tensorboard --logdir=./logs --port=6006
  ```



# Tensorboard



# Generowanie *event files*

```
root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
    import time
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir()

# ...

tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)

history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, tensorboard_cb])
```

# Strojenie hiperparametrów modelu

# Strojenie hiperparametrów

Nawet przy prostym modelu MLP mamy wiele decyzji do podjęcia:

- liczba warstw ukrytych,
- liczba neuronów w warstwach ukrytych,
- metoda inicjalizacji wag,
- krok algorytmu uczenia.

Jak rozwiązać ten problem? Metodą prób i błędów!

# Systematyczne przeszukiwanie przestrzeni parametrów

Z pomocą przychodzi scikit-learn i jego moduł `model_selection`:

- `GridSearchCV` przeszukuje *n*-wymiarową przestrzeń poprzez nałożenie siatki,
- `RandomizedSearchCV` prowadzi poszukiwania w sposób stochastyczny,
- obie metody posiadają warianty `Halving*`.

Modele Keras obudowujemy przy pomocy obiektów KerasRegressor i KerasClassifier biblioteki scikeras (dawniej wchodziły w skład modułu `tf.keras.wrappers.scikit_learn`).

# Przykład: budowanie modelu

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3,
                input_shape=[8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation="relu"))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(learning_rate=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model
```

# Przykład: użycie scikeras

```
from scikeras.wrappers import KerasRegressor
keras_reg = KerasRegressor(build_model)

keras_reg.fit(X_train, y_train, epochs=100,
              validation_data=(X_valid, y_valid),
              callbacks=[keras.callbacks.EarlyStopping(patience=10)])

mse_test = keras_reg.score(X_test, y_test)
y_pred = keras_reg.predict(X_new)
```

Uwagi:

- dodatkowe argumenty `fit()` zostaną przekazane do modelu
- `score` jest odwrotnością MSE

# Przykład: RandomizedSearchCV

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distribs = {
    "model__n_hidden": [0, 1, 2, 3],
    "model__n_neurons": np.arange(1, 100),
    "model__learning_rate": reciprocal(3e-4, 3e-2)
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distribs,
                                   n_iter=10, cv=3, verbose=2)
rnd_search_cv.fit(X_train, y_train, epochs=3,
                  validation_data=(X_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

# Narzędzia do optymalizacji hiperparametrów

- Hyperopt
- Hyperas
- Keras Tuner
- scikit-optimize
- sklearn-deap

# Omówienie wybranych hiperparametrów

# Liczba warstw ukrytych

- Dla danego problemu, sieci głębokie potrzebują wykładniczo niższej liczby neuronów niż sieci płytkie:
  - warstwy niższe odpowiadają strukturom niskiego poziomu (np. liniom na obrazie).
  - warstwy pośrednie łączą te struktury w bardziej złożone (np. figury geometryczne).
  - najwyższe warstwy łączą je w struktury wysokiego poziomu (np. twarze).
- Typowe podejście: zwiększanie liczby warstw aż do przeuczenia sieci.
- Uczenie transferowe: wykorzystanie wag z części (najczęściej niższych) warstw istniejącego modelu zamiast ich losowej inicjalizacji.

# Liczba neuronów na warstwę

- Klasyczne podejście: „piramida”
- Obecnie porzucone, gdyż sieci „prostokątne” radzą sobie równie dobrze, a mamy tylko jeden hiperparametr (zamiast tylu, ile jest warstw).
- Tu znów zwiększamy liczbę neuronów aż do wystąpienia przeuczenia.
- Ale można też zacząć z wartościami nadmiarowymi i wykorzystanie technik regularyzacji (np. *early stopping*) – tzw. „stretch pants approach”.

# Inne hiperparametry

- Krok uczenia (*learning rate*)
- Algorytm optymalizacji
- Rozmiar wsadu
- Funkcja aktywacji
- Liczba epok