Rekurencyjne sieci neuronowe

dr inż. Sebastian Ernst

Przedmiot: Uczenie Maszynowe

Rekurencyjne sieci neuronowe

Rekurencyjne sieci neuronowe

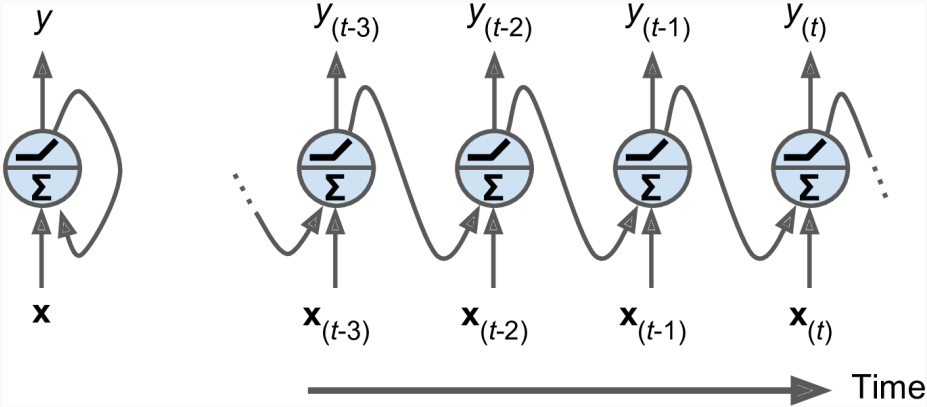
- ang. *recurrent neural networks* (RNN)
- często wykorzystywane do predykcji
- uczą się wzorców w sekwencjach: zdaniach, dokumentach, szeregach czasowych, próbkach audio
- sekwencje mogą być dowolnej długości

Przetwarzanie szeregów czasowych

- najczęściej w celu przewidywania przyszłości
- zastosowania
  - finanse (giełda)
  - pojazdy autonomiczne
  - sterowanie
  - wykrywanie usterek

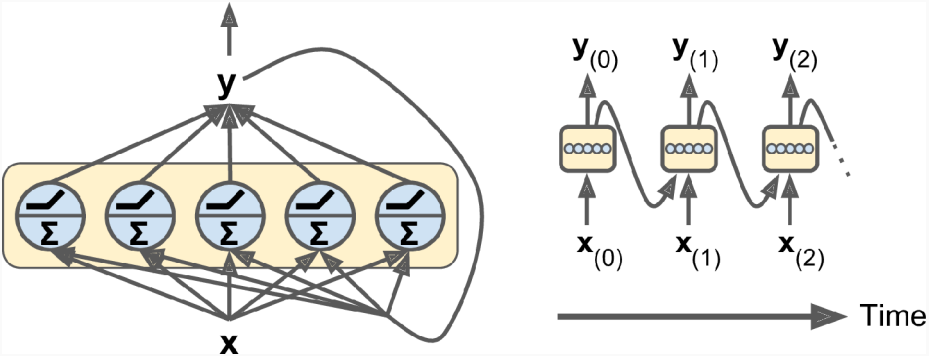
Rekurencyjne neurony

- otrzymują sygnały ze swoich poprzednich wyjść
- krok czasowy t = *ramka*
- rozciągnięcie w czasie – *unrolling*



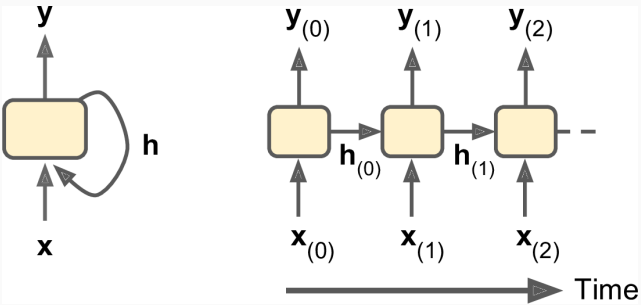
Warstwa rekurencyjnych neuronów

- otrzymuje wektor wejściowy *oraz* wektor wyjściowy z poprzedniego kroku
- każdy neuron ma dwa zestawy wag: $\mathbf{w}_x$ dla wejść $(\mathbf{x}_{(t)})$ oraz $\mathbf{w}_{\hat{y}}$ dla wyjścia w poprzednim kroku $(\hat{\mathbf{y}}_{(t-1)})$

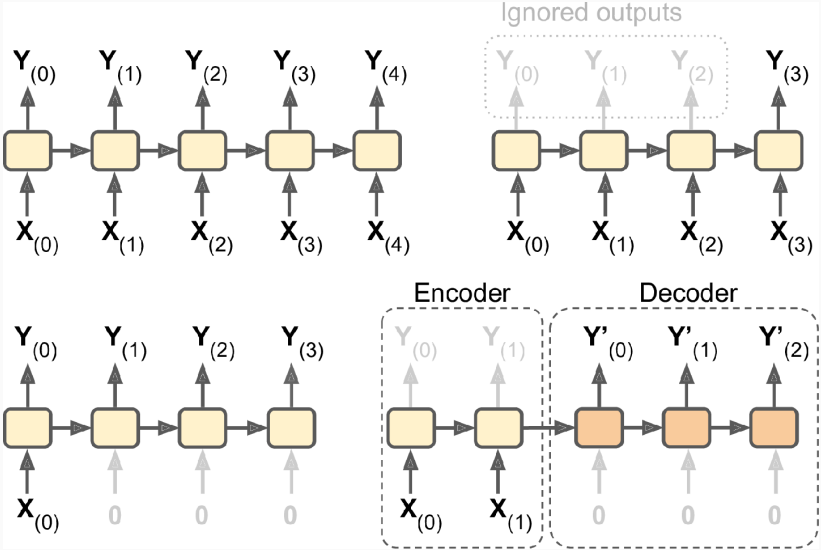


Komórki pamięci

- wyjście w chwili t zależy od wejść w poprzednich chwilach
- neuron ma więc rodzaj *pamięci*
- typowo pamięć jest krótka (ok. 10 kroków)
- wyjście z komórki może nie być tożsame z jej stanem wewnętrznym

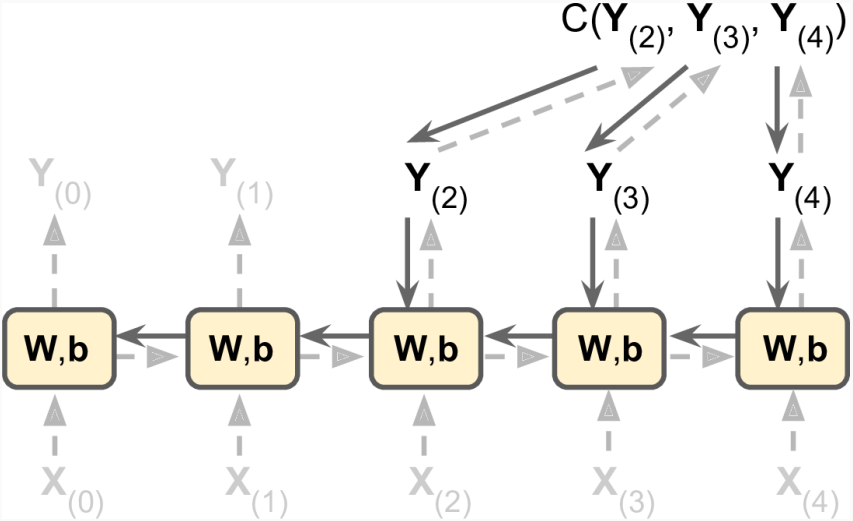


Sekwencje wejściowe i wyjściowe



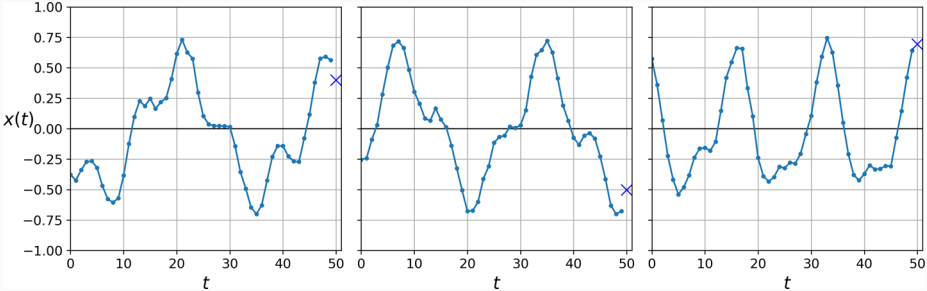
Uczenie RNN

- *unrolling* i zwykła propagacja wsteczna = BPTT (*backpropagation through time*)



Predykcja szeregów czasowych

- przewidywanie – *forecasting*
- wypełnianie luk w przeszłości – *imputation*



Generujemy szeregi czasowe

```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10))  #   wave 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # + wave 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5)   # + noise
    return series[..., np.newaxis].astype(np.float32)
```

Wskaźniki bazowe

Predykcja naiwna:

```
y_pred = X_valid[:, -1]
```

Predykcja siecią gęstą – model liniowy:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])
model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
model.evaluate(X_valid, y_valid)
```

Najprostsza RNN

Jeden neuron, jedna warstwa:

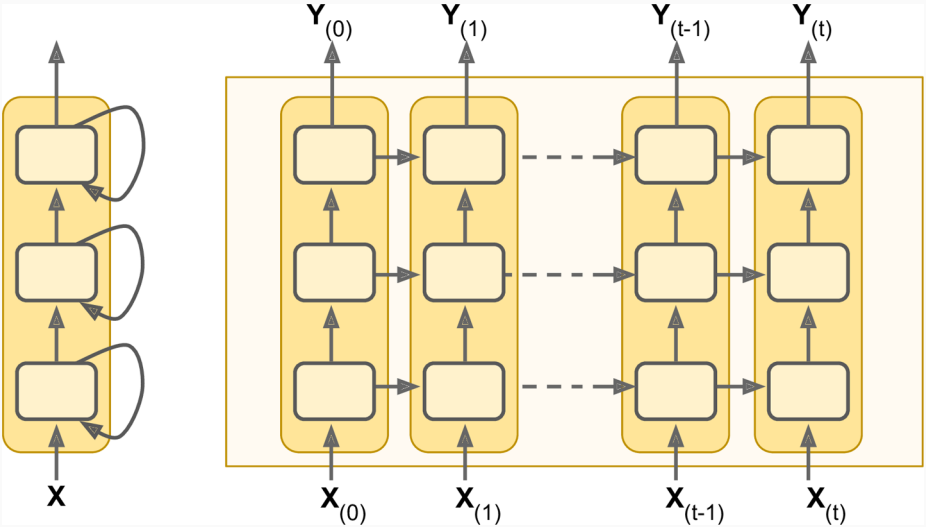
```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])

optimizer = keras.optimizers.Adam(learning_rate=0.005)
model.compile(loss="mse", optimizer=optimizer)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

Ta sieć ma łącznie 3 parametry, a „liniowa” sieć gęsta – 51.

Głębokie RNN

Głęboka RNN



Głęboka RNN, przykład

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

- ukryty stan w warstwie wyjściowej to zaledwie jedna liczba
- warstwy RNN korzystają z *tanh*, a więc wyjście musi być w zakresie $[-1, 1]$.

Głęboka RNN, drugie podejście

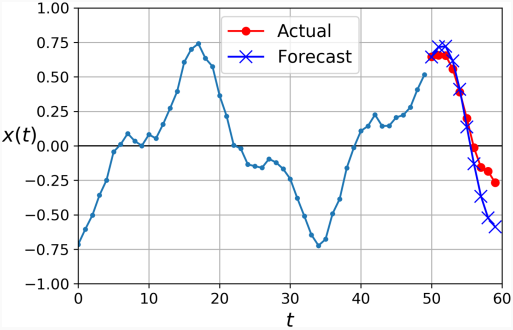
```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

- szybciej zbieżna
- można użyć dowolnej funkcji aktywacji

Prognozowanie kilku kroków naprzód

- pierwsze podejście: „doklejanie” predykowanych wartości na końcu szeregu
- drugie podejście: predykcja n wartości jednocześnie



Prognozowanie wielu kroków, przygotowanie danych

```
n_steps = 50
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10:, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

Prognozowanie wielu kroków, model

Zmieniamy liczbę neuronów w warstwie wyjściowej:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

Model *sequence-to-sequence*

- uczymy model aby zwracał 10 wartości w każdym kroku, nie tylko w ostatnim
- przygotowujemy odpowiednio sekwencje etykiet (*target*)
- warstwy RNN muszą mieć argument `return_sequences=True`
- warstwę wyjściową uruchamiamy dla każdego kroku czasowego przy pomocy warstwy `TimeDistributed`
- do ewaluacji przyda się funkcja licząca MSE w ostatnim kroku

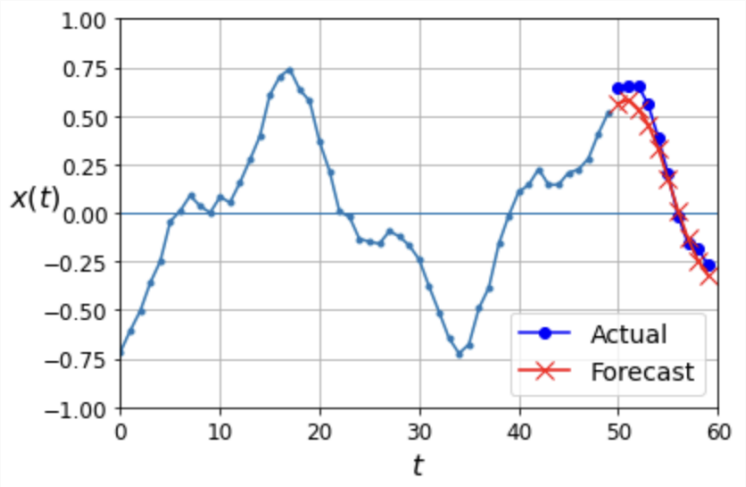
Model *sequence-to-sequence*, implementacja

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

def last_time_step_mse(Y_true, Y_pred):
    return keras.metrics.mean_squared_error(Y_true[:, -1],
                                            Y_pred[:, -1])

model.compile(loss="mse", optimizer=keras.optimizers.Adam(
    learning_rate=0.01), metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

Model *sequence-to-sequence*, wyniki



Przykład praktyczny

Zbiór danych

Zbiór zawiera liczbę pasażerów komunikacji zbiorowej w Chicago z podziałem na autobusy i pociągi.

```
import pandas as pd
from pathlib import Path

path = Path("datasets/ridership/CTA_-_Ridership_-_Daily_Boarding_Totals.csv
df = pd.read_csv(path, parse_dates=["service_date"])
df.columns = ["date", "day_type", "bus", "rail", "total"]  # shorter names
df = df.sort_values("date").set_index("date")
df = df.drop("total", axis=1)  # no need for total, it's just bus + rail
df = df.drop_duplicates()  # remove duplicated months (2011-10 and 2014-07
```

Baseline 1: naiwny – „tydzień wcześniej”

```
>>> diff_7 = df[["bus", "rail"]].diff(7)["2019-03":"2019-05"]

>>> diff_7.abs().mean()
bus     43915.608696
rail    42143.271739
dtype: float64
```

Baseline 2: SARIMA

SARIMA to model autoregresyjny należący do rodziny ARMA.

```
origin, start_date, end_date = "2019-01-01", "2019-03-01", "2019-05-31"
time_period = pd.date_range(start_date, end_date)
rail_series = df.loc[origin:end_date]["rail"].asfreq("D")
y_preds = []
for today in time_period.shift(-1):
    model = ARIMA(rail_series[origin:today],  # train on data up to "today
                  order=(1, 0, 0),
                  seasonal_order=(0, 1, 1, 7))
    model = model.fit()  # note that we retrain the model every day!
    y_pred = model.forecast()[0]
    y_preds.append(y_pred)

y_preds = pd.Series(y_preds, index=time_period)
```

Model liniowy

```
tf.random.set_seed(42)
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=[seq_length])
])
early_stopping_cb = tf.keras.callbacks.EarlyStopping(
    monitor="val_mae", patience=50, restore_best_weights=True)
opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
history = model.fit(train_ds, validation_data=valid_ds, epochs=500,
                    callbacks=[early_stopping_cb])
```

MAE: 37 866

Model liniowy, problemy i poprawka

Słaby wynik związany jest z dwoma problemami wspomnianymi wcześniej:

- ograniczona pamięć – tylko 1 rekurencyjny neuron, a więc patrzy na jedną wartość wstecz; model liniowy patrzył na 56 wartości,
- wartości należą do przedziału $[0, 1.4]$, a funkcja aktywacji *tanh* ogranicza nas do przedziału $[-1; 1]$.

```
univar_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 1]),
    tf.keras.layers.Dense(1)  # no activation function by default
])
```

MAE: 27 703

Model głęboki

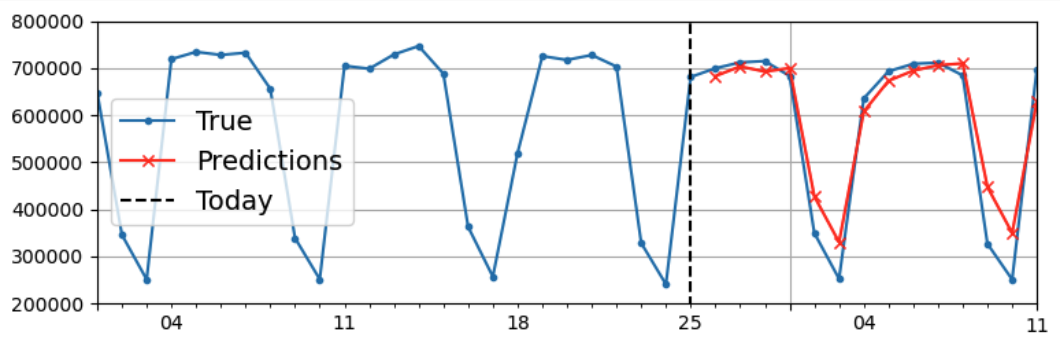
```
deep_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None
    tf.keras.layers.SimpleRNN(32, return_sequences=True),
    tf.keras.layers.SimpleRNN(32),
    tf.keras.layers.Dense(1)
])
```

MAE: 31 211

Kilka kroków naprzód, podejście 1

Doklejamy kolejne predykcje, model uruchomiony 14 razy.

```
X = rail_valid.to_numpy()[np.newaxis, :seq_length, np.newaxis]
for step_ahead in range(14):
    y_pred_one = univar_model.predict(X)
    X = np.concatenate([X, y_pred_one.reshape(1, 1, 1)], axis=1)
```



Kilka kroków naprzód, podejście 2

Przewidujemy 14 wartości jednocześnie.

```
ahead_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

Przy takim podejściu błąd nie narasta tak jak w przypadku poprzedniego.

Model *sequence-to-sequence*

Dodajemy argument `return_sequences=True`, dzięki któremu warstwa zwraca pełne sekwencje.

```
seq2seq_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None
    tf.keras.layers.Dense(14)
])
```

Przetwarzanie długich sekwencji

Wyzwania

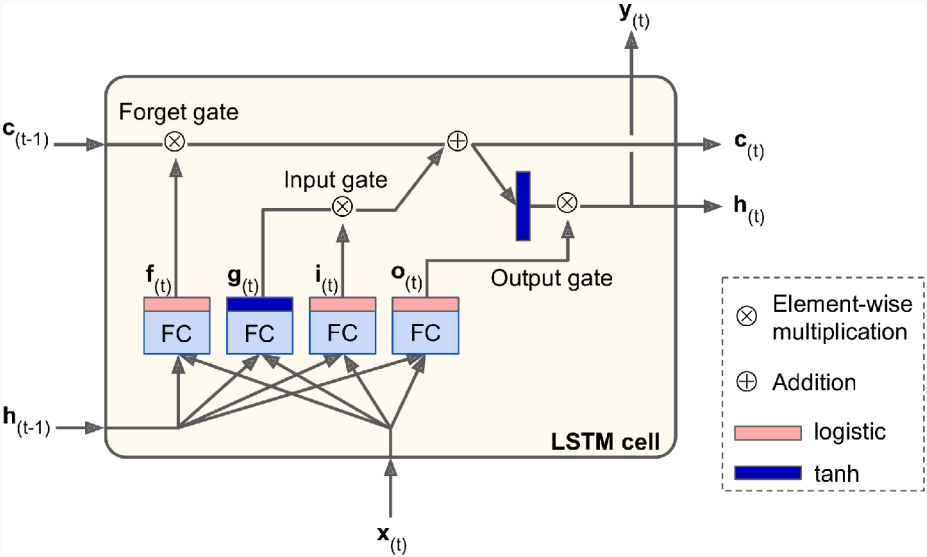
- wiele kroków, a więc sieć *unrolled* robi się bardzo głęboka
- może to powodować problemy niestabilności gradientu
- po pewnym czasie sieć „zapomina” początek sekwencji
- istnieją rodzaje komórek z pamięcią długoterminową

Komórki *Long Short-Term Memory* (LSTM)

- 1997, Sepp Hochreiter, Juergen Schmidhuber
- użycie analogiczne jak przy prostych komórkach rekurencyjnych
- w Keras stosujemy warstwę `LSTM` (lub `RNN` z argumentem `LSTMCell` – ale bez optymalizacji GPU)

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Budowa komórki LSTM



Komórki *Gated Recurrent Unit* (GRU)