

Problemy uczenia sieci neuronowych

dr inż. Sebastian Ernst

Przedmiot: Uczenie Maszynowe

Strojenie hiperparametrów modelu

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!

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_distributions = {
    "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_distributions,
                                   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)])
```

- Moduł pozwalający na strojenie parametrów modelu:

```
import keras_tuner as kt
```

- Najprostsza forma użycia polega na zdefiniowaniu funkcji budującej model, która przyjmuje obiekt `HyperParameters` jako argument; składa się ona zazwyczaj z dwóch części:
 - część definiująca hiperparametry, ich dziedziny (*int*, *float*, *choice*, *boolean*, *fixed*) oraz zakresy,
 - część budująca model w oparciu o zdefiniowane hiperparametry.
- Funkcja ta wykorzystywana jest następnie przez wybraną *klasę tunera*: random search, grid search, optymalizacja bayesowska, hyperband oraz wrapper scikit-learn.

Przykład: Keras Tuner – budowanie modelu

```
def build_model(hp):  
    n_hidden = hp.Int("n_hidden", min_value=0, max_value=8, default=2)  
    n_neurons = hp.Int("n_neurons", min_value=16, max_value=256)  
    learning_rate = hp.Float("learning_rate", min_value=1e-4, max_value=1e-2,  
                             sampling="log")  
    optimizer = hp.Choice("optimizer", values=["sgd", "adam"])  
    if optimizer == "sgd":  
        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)  
    else:  
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)  
  
    model = tf.keras.Sequential()  
    model.add(tf.keras.layers.Flatten())  
    for _ in range(n_hidden):  
        model.add(tf.keras.layers.Dense(n_neurons, activation="relu"))  
    model.add(tf.keras.layers.Dense(10, activation="softmax"))  
    model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,  
                  metrics=["accuracy"])  
    return model
```

Przykład: Keras Tuner – *random search*

```
random_search_tuner = kt.RandomSearch(  
    build_model, objective="val_accuracy", max_trials=5, overwrite=True,  
    directory="my_fashion_mnist", project_name="my_rnd_search", seed=42)  
  
random_search_tuner.search(X_train, y_train, epochs=10,  
                           validation_data=(X_valid, y_valid))
```

Keras Tuner – sterowanie procesem uczenia

Aby sterować procesem uczenia, czyli mieć wpływ na funkcję `fit()`, możemy zdefiniować swój własny model oparty o klasę `HyperModel`:

```
class MyClassificationHyperModel(kt.HyperModel):  
    def build(self, hp):  
        return build_model(hp)  
  
    def fit(self, hp, model, X, y, **kwargs):  
        if hp.Boolean("normalize"):  
            norm_layer = tf.keras.layers.Normalization()  
            X = norm_layer(X)  
        return model.fit(X, y, **kwargs)
```

Przykład: Keras Tuner – Hyperband

```
hyperband_tuner = kt.Hyperband(  
    MyClassificationHyperModel(), objective="val_accuracy", seed=42,  
    max_epochs=10, factor=3, hyperband_iterations=2,  
    overwrite=True, directory="my_fashion_mnist", project_name="hyperband".
```

Przykład: Keras Tuner + Tensorboard

```
root_logdir = Path(hyperband_tuner.project_dir) / "tensorboard"
tensorboard_cb = tf.keras.callbacks.TensorBoard(root_logdir)
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=2)
hyperband_tuner.search(X_train, y_train, epochs=10,
                       validation_data=(X_valid, y_valid),
                       callbacks=[early_stopping_cb, tensorboard_cb])
```

Inne narzędzia do optymalizacji hiperparametrów

- Hyperopt
- Hyperas
- scikit-optimize
- sklearn-deap

Omówienie wybranych hiperparametrów

- 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.

- 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”.

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

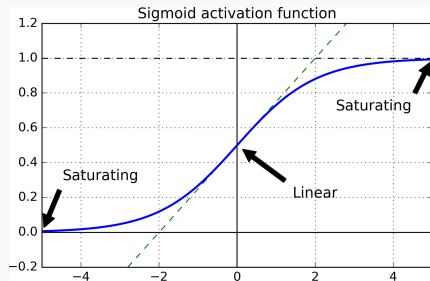
Problemy przy uczeniu sieci

Co może pójść źle?

- Zbyt szybko malejące lub rosnące gradienty podczas „powrotu” przez sieć – dolne warstwy się nie uczą
- Za mało danych etykietowanych, szczególnie przy dużych sieciach
- Bardzo wolne uczenie sieci
- Im więcej parametrów, tym większe ryzyko przeuczenia

Problem znikających/eksplodujących gradientów

- Algorytm propagacji wstecznej przenosi gradienty błędów idąc od warstwy wyjściowej do wejściowej (czyli wstecz)
- Często wartości gradientów spadają przy przechodzeniu do coraz niższych warstw – problem *znikających gradientów*
- Czasami jest wręcz przeciwnie – problem *eksplodujących gradientów*
- W 2010 Glorot i Bengio odkryli związek między niestabilnością gradientów a używaniem funkcji sigmoidalnej oraz popularnego wówczas sposobu inicjalizacji wag (rozkład normalny $\mu = 0$, $\sigma = 1$)



Inicjalizacja Glorota i He

- Teoretycznie: wariancja wejść każdej warstwy musi być równa wariancji jej wyjść
- W praktyce: wagi połączeń warstwy powinny być równe:
 - rozkładowi normalnemu o $\mu = 0$, $\sigma^2 = \frac{1}{fan_{avg}}$ lub
 - rozkładowi jednostajnemu pomiędzy $-r$ a $+r$ przy $r = \sqrt{\frac{3}{fan_{avg}}}$

Metoda	Funkcje aktywacji	σ^2
Glorot	brak, tanh, sigmoid, softmax	$1/fan_{avg}$
He	ReLU & co.	$2/fan_{in}$
LeCun	SELU	$1/fan_{in}$

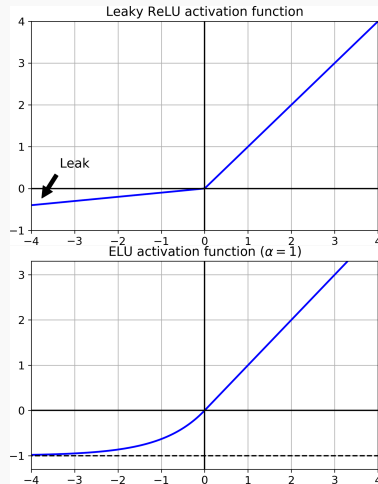
Nienasycające funkcje aktywacji

- Przy ReLU: problem „umierających” neuronów
 - suma ważona wejść zawsze ujemna
- Dwa rozwiązania:
 - leaky ReLU:

$$\text{LeakyReLU}_{\alpha}(z) = \max(\alpha z, z)$$

- Exponential Linear Unit:

$$\text{ELU}_{\alpha}(z) = \begin{cases} \alpha(e^z - 1) & \text{dla } z < 0 \\ z & \text{dla } z \geq 0 \end{cases}$$



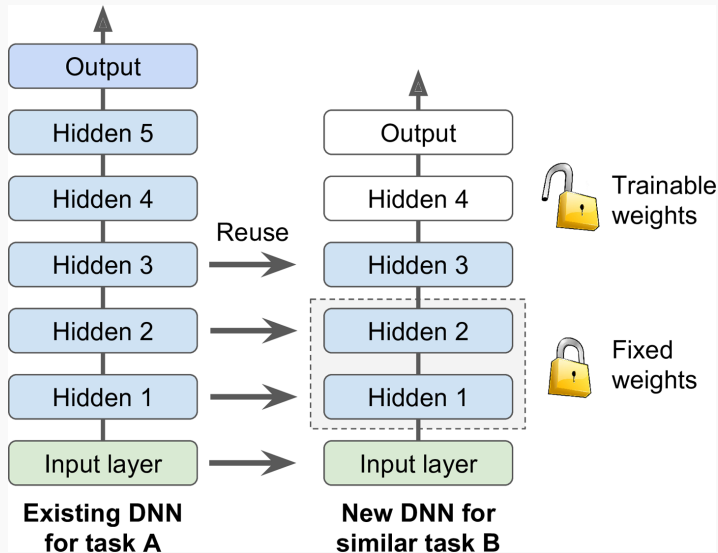
Normalizacja wsadów

- Stosowanie inicjalizacji He i ELU/ReLU & co. zmniejsza ryzyko niestabilności gradientów na początku uczenia, ale problem może pojawić się później.
- Technika *Batch Normalization* polega na dodaniu operacji tuż przed lub po funkcji aktywacji każdej warstwy ukrytej – wycentrowanie i normalizacja wejścia + skalowanie i przesunięcie wyniku.

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


Uczenie transferowe

Wykorzystanie wytrenowanych warstw



```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Uwaga: trening nowego modelu będzie modyfikował wagi warstw również w `model_A` – aby tego uniknąć, klonujemy model:

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
model_B_on_A = keras.models.Sequential(model_A_clone.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Wstępne przyuczenie warstwy wyjściowej

Początkowo, warstwa wyjściowa może zwracać wartości dalekie od ideału, a propagacja zmian może zniszczyć wagi przeniesione z pierwotnego modelu. W tym celu blokujemy możliwość modyfikacji wag przeniesionych warstw.

```
for layer in model_B_on_A.layers[:-1]:  
    layer.trainable = False
```

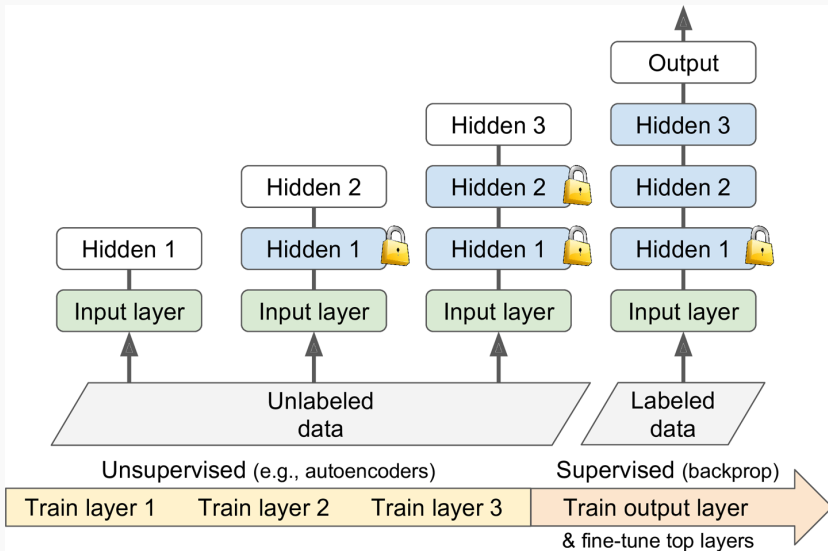
```
model_B_on_A.compile(loss="binary_crossentropy",  
                      optimizer=keras.optimizers.SGD(learning_rate=1e-3),  
                      metrics=["accuracy"])
```

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,  
                           validation_data=(X_valid_B, y_valid_B))
```

Uczenie nowego modelu

```
for layer in model_B_on_A.layers[:-1]:  
    layer.trainable = True  
  
model_B_on_A.compile(loss="binary_crossentropy",  
                      optimizer=keras.optimizers.SGD(learning_rate=1e-3),  
                      metrics=["accuracy"])  
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,  
                           validation_data=(X_valid_B, y_valid_B))
```

Nienadzorowane uczenie wstępne



Inne (szybsze) algorytmy optymalizacji

Przyspieszanie procesu uczenia

Dotychczas znamy **cztery sposoby** na przyspieszenie uczenia:

1. dobra strategia inicjalizacji wag połączeń
2. dobra funkcja aktywacji
3. korzystanie z normalizacji wsadów (BN)
4. wykorzystanie części wstępnie przyuczonej sieci

Piąty sposób: wykorzystanie algorytmu optymalizacji innego niż gradientowy (*gradient descent*).

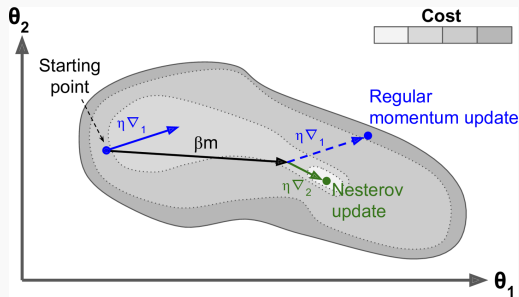
Algorytm gradientowy z pędem

- Klasyczny algorytm gradientowy nie uwzględnia wcześniejszych wartości gradientów – bierze pod uwagę tylko wartość chwilową.
- Algorytm z pędem dodaje *wektor pędu*, a więc wartość gradientu przekłada się na *przyspieszenie* a nie *prędkość*:
 1. $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\Theta} J(\Theta)$
 2. $\Theta \leftarrow \Theta + \mathbf{m}$
- Np. dla $\beta = 0.9$ prędkość może wzrosnąć 10x w stosunku do zwykłego algorytmu gradientowego.
- Przyspieszenie widoczne szczególnie przy różnych skalach wejść (efekt wydłużonej misy).
- W Keras wystarczy ustawić parametr momentum:

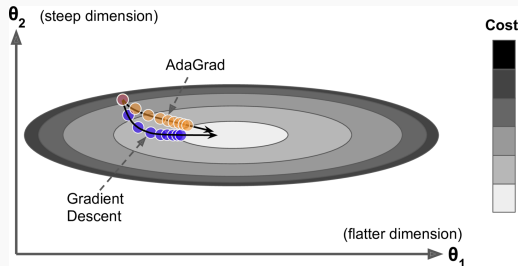
```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

Nesterov Accelerated Gradient (NAG)

- Zaproponowany w 1983 przez Nesterowa, prawie zawsze szybszy niż „zwykły” gradient z pędem.
- Patrzy „z wyprzedzeniem”:
 1. $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\Theta} J(\Theta + \beta \mathbf{m})$
 2. $\Theta \leftarrow \Theta + \mathbf{m}$
- W Keras dodajemy argument `nesterov=True`.



- Koryguje kierunek gradientu w stronę globalnego minimum, skalując wektor gradientów wedle „stromości”:
 1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\Theta} J(\Theta) \otimes \nabla_{\Theta} J(\Theta)$
 2. $\Theta \leftarrow \Theta - \eta \nabla_{\Theta} J(\Theta) \oslash \sqrt{\mathbf{S} + \epsilon}$
- Problem przy sieciach głębokich: zatrzymuje się za wcześnie.



- Rozwinięcie AdaGrad – bierze pod uwagę tylko gradienty z ostatnich iteracji (a nie od początku uczenia)
- Patrzy „z wyprzedzeniem”:

1. $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\Theta} J(\Theta) \otimes \nabla_{\Theta} J(\Theta)$
2. $\Theta \leftarrow \Theta - \eta \nabla_{\Theta} J(\Theta) \oslash \sqrt{\mathbf{s} + \varepsilon}$

- Typowa wartość β : 0.9.
- W Keras:

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Adam

- Adam = *adaptive moment estimation*
- Łączy cechy gradientu z pędem oraz RMSProp:
 - tak jak gradient z pędem, pamięta wykładniczo zanikającą średnią poprzednich gradientów,
 - tak jak RMSProp, pamięta wykładniczo zanikającą średnią kwadratów poprzednich gradientów.

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\Theta} J(\Theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\Theta} J(\Theta) \otimes \nabla_{\Theta} J(\Theta)$
3. $\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4. $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5. $\Theta \leftarrow \Theta + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}$

```
keras.optimizers.Adam(lr=0.001  
                        beta_1=0.9,  
                        beta_2=0.999)
```