ЛЕКЦІЯ №10
Ресурси Keras.TensorFlow.

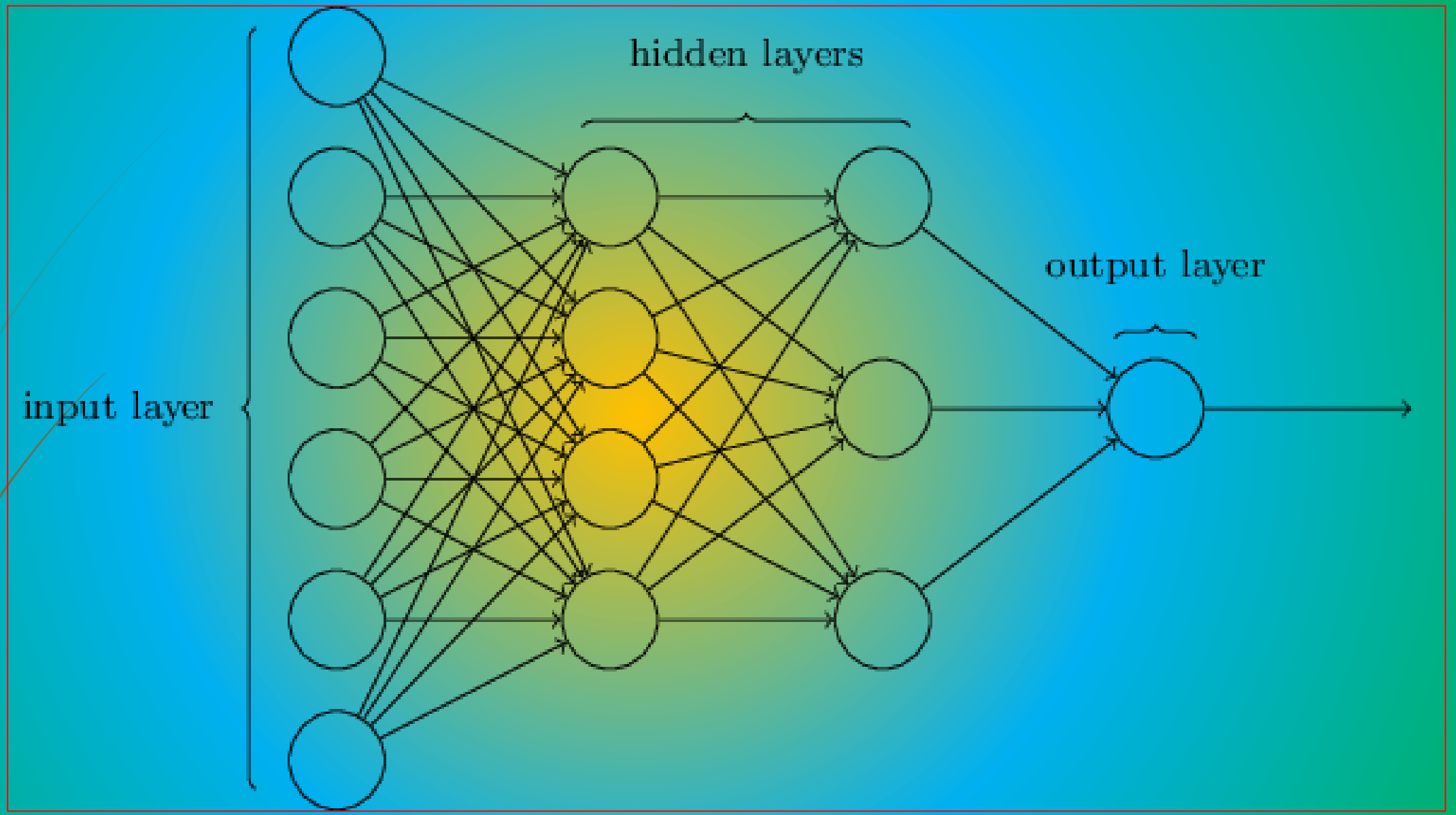
Ресурси Keras.

Keras є високорівневими нейронними мережами API, написаними на Python і можуть працювати поверх TensorFlow, CNTK або Теано. Він був розроблений з упором на можливість швидкого експериментування. Здатність йти від ідеї до результату з найменшою можливою затримкою є ключем до проведення добрих досліджень.

Творці першої нейромережі надихалися структурою кори мозку. Базовий рівень мережі, перцептрон, є насправді математичним аналогом біологічного нейрона. І, як і в головному мозку, у нейромережі можуть з'являтися пересічені один з одним перцептрони.

Перший шар нейромережі називається вхідним. Кожен вузол цього шару отримує на вхід будь-яку інформацію та передає її на наступні вузли в інших шарах. Найчастіше між вузлами одного шару немає зв'язків, а останній вузол ланцюжка виводить результат роботи нейромережі.

Вузли посередині називаються прихованими, оскільки не мають з'єднань із зовнішнім світом, як вузли виведення та введення. Вони викликаються лише у разі активації попередніх шарів.


hidden layers
input layer
output layer

Основні принципи Keras

Встановлення та налаштування. Спочатку Keras виріс як зручна надбудова над Theano. Keras став спочатку підтримувати Tensorflow, а потім взагалі став його частиною.
Keras встановлюється як звичайний пітонівський пакет: pip install keras
Щоб працювати з Keras, має бути встановлений хоча б один із фреймворків — Theano або Tensorflow.
Бекенди - це те, через що Keras став відомим і популярним. Фронтенд (англ. front-end) - клієнтська сторона інтерфейсу користувача до програмно-апаратної частини сервісу. Бекенд (англ. back-end) - програмно-апаратна частина сервісу. Фронт- та бекенд – це варіант архітектури програмного забезпечення. Терміни з'явилися у програмній інженерії внаслідок розвитку принципу поділу відповідальності між зовнішнім уявленням та внутрішньою реалізацією. Back-end створює деякий API, який використовує front-end. Таким чином, front-end розробнику не потрібно знати особливостей реалізації сервера, а back-end розробнику - реалізацію front-end. Keras дозволяє використовувати як бекенд різні інші фреймворки. При цьому написаний код буде виконуватися незалежно від використовуваного бекенда. Починалася розробка, як було зазначено, з Theano, але згодом додався Tensorflow. Зараз Keras за замовчуванням працює саме з ним, але якщо потрібно використовувати Theano, то є два варіанти, як це зробити:

1. Відредагувати файл конфігурації keras.json, який лежить на шляху $HOME/.keras/keras.json(або %USERPROFILE%\.keras\keras.json у разі операційних систем сімейства Windows). Нам потрібне поле backend:

```
{
"image_data_format": "channels_last",
"epsilon": 1e-07,
"floatx": "float32",
"backend": "theano"
}
```

2. Другий шлях – це задати змінну оточення KERAS_BACKEND, наприклад, так:

KERAS_BACKEND=theano python -c "з keras import backend" Using Theano backend.

Моделі Keras.

Основна структура даних Keras – це модель, спосіб організації шарів. У Keras є два основних типи моделей: послідовна модель Sequential і клас Model, що використовується з функціональним API. Найпростішим типом моделі є Sequential модель, яка є лінійною сукупністю шарів. Для складніших архітектур необхідно використовувати функціональний API Keras, який дозволяє створювати довільні графіки шарів.

Ці моделі мають ряд загальних властивостей та загальних методів:

- model.layers - це список шарів, що містяться в моделі.
- model.inputs - це список вхідних тензорів моделі.
- model.outputs – це список вихідних тензорів моделі.
- model.summary() – друкує зведене уявлення про модель.
- model.get_config() – повертає словник, який містить конфігурацію моделі.
- model.get_weights() – повертає список усіх вагових тензорів у моделі.
- model.set_weights(weights) - встановлює значення ваги моделі з масиву. Масиви у списку повинні мати ту саму форму, що й повертаються get_weights().
- model.to_json() - повертає представлення моделі як у вигляді рядка JSON. Це уявлення не включає ваги, лише архітектуру.

Наведемо приклад використання методу model.to_json():

```
from keras.models import model_from_json
json_string = model.to_json()
model = model.from_json(json_string)
```

API класу Model.

Використовуючи функціональний API, можна створити екземпляр класу Model для деякого вхідного тензора і вихідний тензора використовуючи наступний код:

```
from keras.models import Model from keras.layers import Input, Dense
a = Input(shape=(32,)) b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

Ця модель включатиме всі рівні, необхідні для обчислення b на основі a.

У випадку моделей з кількома входами або кількома виходами також можна використовувати списки:

```
model = Model(inputs=[a1, a2], outputs=[b1, b2, b3])
```

Основні методи класу Model

1. Метод налаштування моделі для навчання:

compile(self, optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_mode=None, weighted_metrics=None, target_tensors=None)

2. Навчання моделі для певної кількості епох:

fit(self, x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial 0, steps_per_epoch=None, validation_steps=None)

Основні аргументи цього методу:

x: масив даних навчання (якщо модель має один вхід) або список масивів (якщо модель має кілька входів).

- y: масив цільових даних (якщо модель має один висновок) або список масивів (якщо модель має кілька виходів).
- batch_size: кількість вибірок оновлення градієнта. Якщо не вказано, batch_size буде встановлено за замовчуванням значення 32.
- epochs: кількість епох для навчання моделі.
- validation_split: Float між 0 та 1. Частка даних навчання, які будуть використовуватися як дані валідації. Модель виділятиме цю частину даних навчання, не тренуватиметься на ній і оцінюватиме помилку та будь-які модельні показники за цими даними наприкінці кожної епохи. - initial_epoch: епоха, з якої розпочати навчання (корисно для відновлення попереднього циклу навчання).

3. Метод для оцінки якості навченості моделі. Цей метод повертає значення помилок та показників для моделі у тестовому режимі.

evaluate(self, x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None)

4. Метод створення вихідних прогнозів для вхідних вибірок.predict(self, x, batch_size=None, verbose=0, steps=None)

Основні аргументи:

- x: вхідні дані, як масив (або список масивів Numpy, якщо модель має кілька входів).
- steps: загальна кількість кроків (партій вибірок) до оголошення раунду прогнозування.

5. Метод вилучення шару на основі його імені чи індексу.

Цей метод повертає екземпляр шару.

get_layer(self, name=None, index=None)

Основні аргументи:

- ❑ name: String, ім'я шару.
- ❑ index: Integer, індекс шару.

Шари в Keras.

Усі шари Keras мають низку загальних методів:

- layer.get_weights() – повертає ваги шару у вигляді списку масивів Numpy.
- layer.set_weights(weights)- встановлює ваги шару зі списку масивів (з тими самими формами, як і вихід get_weights).
- layer.get_config() - повертає словник, який містить конфігурацію шару.

Шар може бути відновлений з його конфігурації, використовуючи наступний:

```
layer = Dense(32)
config = layer.get_config()
reconstructed_layer = Dense.from_config(config)
```

Якщо шар має один вузол (тобто якщо він не є загальним шаром), то можна отримати його вхідний тензор, вихідний тензор, розмірність вхідного масиву та розмірність вихідного масиву через властивості:

- layer.input
- layer.output
- layer.input_shape
- layer.output_shape

Щільний шар Dense

Шар Dense реалізує операцію: output = activation(dot(input, kernel) + bias) де activation функція активації, передана як activation аргументу, kernel є матрицею шару ваг, і bias є вектор зміщення, створений шаром.

Щільний шар створюється використанням методу:

keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None)

Розглянемо приклад створення щільного шару.

```
#Спочатку створюємо послідовну модель model = Sequential()
# додаємо перший щільний шар
# модель прийматиме на вході масив (*, 16) і вихідний масив (*, 32)
model.add(Dense(32, input_shape=(16,)))
# при додаванні наступних шарів не потрібно вказувати розміри вхідних масивів model.add(Dense(32))
```

Щоб вказати функцію активації, яка буде застосована до виходу, необхідно використовувати метод:
keras.layers.Activation(activation)
Як аргументи activation необхідно вказати ім'я активації, що використовується.

Перенавчання (overfitting) — одна з проблем глибоких нейронних мереж, яка полягає в тому, що модель добре розпізнає лише приклади з навчальної вибірки, адаптуючись до навчальних прикладів, замість того, щоб вчитися класифікувати приклади, що не брали участь у навчанні (втрачаючи здатність до узагальнення). Найбільш ефективним вирішенням проблеми перенавчання є метод виключення (Dropout).
keras.layers.Dropout(rate, noise_shape=None, seed=None)

Мережі для уникнення перенавчання виходять за допомогою виключення з мережі (dropping out) нейронів з ймовірністю rate, таким чином, ймовірність того, що нейрон залишиться в мережі, становить 1 rate. "Виключення" нейрона означає, що при будь-яких вхідних даних або параметрах він повертає 0.

Для перетворення результату на певну форму необхідно використовувати метод: keras.layers.Reshape(target_shape)
Як аргумент target_shape вказується кортеж цілих чисел.
Розглянемо приклад: model.add(Reshape((3, 4), input_shape=(12,)))
Розмірність масиву вихідного шару: model.output_shape == (None, 3, 4) model.add(Reshape((6, 2)))
Розмірність масиву вихідного шару: model.output_shape == (None, 6, 2)
Для зміни розмірів вхідного масиву можна використати метод: keras.layers.Permute(dims)
Приклад:

```
model = Sequential()
model.add(Permute((2, 1), input_shape=(10, 64)))
```

Ресурс TensorFlow.

Серед бібліотек загального призначення, які здатні будувати граф обчислень та виконувати автоматичне диференціювання, довгий час була Theano, розроблена в університеті Монреаля. Згодом Google випустила бібліотеку TensorFlow.

Програма, яка використовує TensorFlow, задає граф обчислень, а потім запускає процедуру типу session.run, яка виконує обчислення і отримує результати.

TensorFlow при цьому звертається до того чи іншого backend, низькорівневої бібліотеки, яка запускає процес обчислення. TensorFlow як бібліотека для символьного диференціювання і навчання нейронних мереж може працювати як на процесорі так і на відеокарті. Основний обєкт, яким оперує TensorFlow це тензор, або багаторівневий масив чисел (в математичному аналізі тензор має інший зміст).

Змінна в TensorFlow це деякий буфер в пам'яті, який вміщує тензори. Змінні необхідно явним чином ініціювати. Щоб заявити змінну, необхідно задати спосіб її ініціювання. При необхідності, їй назначається ім'я, на яке потім можна посилатися.

Наприклад:

```
w = tf.Variable(tf.random_normal([3, 2], mean=0.0, stddev=0.4), name='weights')
b = tf.Variable(tf.zeros([2]), name='biases')
```

Для змінних, можна явно вказати, де саме їм належить знаходитись в памяті. Якщо необхідно, щоб змінна знаходилась на відеокарті (нульовій відеокарті):

```
with tf.device('/gpu:0'):
    w = tf.Variable(tf.random_normal([3, 2], mean=0.0, stddev=0.4), name='weights')
```



На компьютері з відповідною відеокартою достатньо встановити версію TensorFlow з підтримкою GPU і всі тензори будуть по замовчуванню ініціюватися на відео карті.

Всі змінні необхідно ініціювати. Це бажано зробити перед початком обчислень:

```
init = tf.initialize_global_variables()
```

Але це не працює у тих випадках, коли потрібно ініціалізувати змінні із значень інших змінних; у таких випадках синтаксис буде наступним:

```
w2 = tf.Variable(w.initialized_value(), name='w2')
```

Усі змінні поточної сесії можна в будь-який момент зберегти у файл:

```
saved = saver.save(sess, 'model.ckpt')
```

Ця процедура збереже всі змінні сесії sess у файл model.ckpt, а щоб потім відновити сесію, достатньо запустити:

```
saver.restore('model.ckpt')
```

У завданнях машинного навчання необхідно повторно застосовувати ту саму послідовність операцій до різних наборів даних. Зокрема, навчання за допомогою міні-батчів передбачає періодичне обчислення результату та помилки на нових прикладах. Для зручності передачі нових даних на відеокарту TensorFlow існують спеціальні тензори - так звані заглушки, tf.placeholder, яким потрібно спочатку передати тільки тип даних і розмірності тензора, а потім самі дані будуть підставлені вже в момент обчислень:

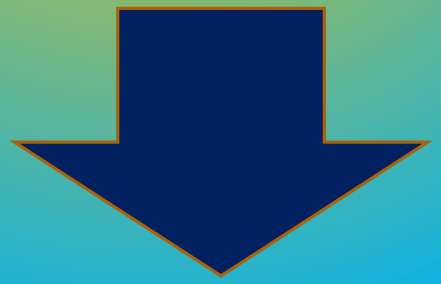
```
x =tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
output = tf.mul(x, y)
with tf.Session() as sess:
result = sess.run(output, feed_dict={x: 2, y: 3})
```

В result після виконання операцій повинно бути 6.



У TensorFlow реалізовано повний набір операцій над тензорами з NumPy за допомогою матричних обчислень над масивами різної форми та конвертування між цими формами (broadcasting). Наприклад, у реальних завданнях часто виникає необхідність до кожного стовпця матриці поелементно додати той самий вектор. У TensorFlow це робиться найпростішим із можливих способів:

```
m = tf.Variable(tf.random_normal([10, 100], mean=0.0, stddev=0.4), name='matrix')
v = tf.Variable(tf.random_normal([100], mean=0.0, stddev=0.4), name='vector')
result = m + v
```



Тут m - це матриця розміру 10×100, а v - вектор довжини 100, і при додаванні v буде додано до кожного стовпця m. Broadcasting застосовується для всіх поелементних операцій над двома тензорами та влаштований наступним чином. Розмірності двох тензорів послідовно порівнюються з кінця; при кожному порівнянні необхідно виконання однієї з двох умов:

- чи розмірності рівні;
- або одна з розмірностей дорівнює 1.

При цьому тензори не повинні мати однакову розмірність: відсутні виміри меншого з тензорів будуть інтерпретовані як одиничні. Розмірність тензора, що отримується на виході, якщо всі умови виконані, обчислюється як максимум з відповідних розмірностей вихідних тензорів. Втім, це звучить досить складно, тому рекомендуємо уважно перевіряти, як саме працюватиме broadcasting у кожному конкретному нетривіальному випадку.

Крім бінарних операцій, TensorFlow реалізує широкий асортимент унарних операцій: зведення в квадрат, взяття експоненти або логарифму, а також широкий спектр редукцій. Наприклад, іноді нам буває необхідно вирахувати середнє значення не по всьому тензору, а, скажімо, по кожному елементу міні-батчу. Якщо першу розмірність тензора ми інтерпретуємо як розмір міні-батчу, то відповідний код може бути таким:

```
tensor = tf.placeholder(tf.float32, [10, 100])
result = tf.reduce_mean(tensor, axis=1)
```

При цьому середнє буде обчислюватися по другій розмірності тензора tensor, тобто десять разів середнім по 100 чисел і отримаємо вектор довжини 10.

У деяких задачах може виникнути необхідність використання тих самих тензорів змінних для декількох різних шляхів обчислень. Припустимо, що ми маємо функцію, що створює тензор лінійного перетворення над вектором:

```
def linear_transform(vec, shape):
    w = tf.Variable(tf.random_normal(shape, mean=0.0,
        stddev=1.0), name='matrix')
    return tf.matmul(vec,w)
```

І необхідно застосувати це перетворення до двох різних векторів:

```
result1 = linear_transform(vec1, shape)
result2 = linear_transform(vec2, shape)
```

Зрозуміло, що в цьому випадку кожна з функцій створить власну матрицю перетворення, що не призведе до бажаного результату. Можна, звичайно, задати тензор w заздалегідь і передати його в функцію `linear_transform` як один з аргументів, проте це порушить принцип інкапсуляції. Для подібних випадків у TensorFlow існують простори змінних (`variable scopes`). Цей механізм складається з двох основних функцій:

- `tf.get_variable(<name>, <shape>, <initializer>)` створює або повертає змінну із заданим ім'ям;
- `tf.variable_scope(<scope_name>)` керує просторами імен, що використовуються у `tf.get_variable()`.

Одним із параметрів функції `tf.get_variable()` є ініціалізатор: замість того, щоб при оголошенні нової змінної в явному вигляді передавати значення, якими вона повинна бути ініціалізована, ми можемо передати функцію ініціалізації, яка, отримавши при необхідності дані про розмірність змінної, ініціалізує її. Давайте трохи змінимо `linear_transform()`:

```
def linear_transform(vec, shape):
    with tf.variable_scope('transform'):
    w = tf.get_variable('matrix', shape,
initializer=tf.random_normal_initializer())
    return tf.matmul(vec, w)
```

Тепер, якщо спробуємо застосувати наше перетворення до двох векторів послідовно, під час другого виклику побачимо помилку:

```
    # Raises ValueError(... transform/matrix already exists
...)
```

Так відбувається тому, що `tf.get_variable()` перевіряє існування змінної в поточному просторі імен, щоб запобігти пов'язанним з іменами помилкам, які зазвичай важко налагодити. Щоб повторно використовувати змінні в просторі імен, потрібно явно повідомити про це TensorFlow:

```
with tf.variable_scope('linear_transformers') as scope:
    result1 = linear_transform(vec1, shape)
     scope.reuse_variables()
    result2 = linear_transform(vec2, shape)
```



TensorFlow – навчання лінійної регресії.

Нагадаємо, що лінійна регресія - це фактично один нейрон, який отримує на вхід вектор значень x, видає число, і на даних $\{(x_1,y_1),(x_2,y_2),\ldots,(x_N,y_N)\}$ завдання полягає у тому, щоб мінімізувати суму квадратів відхилень оцінок нейрона $\hat{y}_i$ від істинних значень y_i:

У коді, наведеному нижче, ми наближатимемо лінійною регресією функцію виду

f = kx + b для k = 2 і b = 1; k та b будуть параметрами, які ми хочемо навчити.

```
import numpy as np,tensorflow as tf
n_samples, batch_size, num_steps = 1000, 100, 20000
X_data = np.random.uniform(1, 10, (n_samples, 1))
y_data = 2 * X_data + 1 + np.random.normal(0, 2, (n_samples, 1))

X = tf.placeholder(tf.float32, shape=(batch_size, 1))
y = tf.placeholder(tf.float32, shape=(batch_size, 1))

with tf.variable_scope('linear-regression'):
    k = tf.Variable(tf.random_normal((1, 1)), name='slope')
    b = tf.Variable(tf.zeros((1,)), name='bias')

y_pred = tf.matmul(X, k) + b
loss = tf.reduce_sum((y - y_pred) ** 2)
optimizer = tf.train.GradientDescentOptimizer().minimize(loss)

display_step = 100
with tf.Session() as sess:
    sess.run(tf.initialize_global_variables())
    for i in range(num_steps):
        indices = np.random.choice(n_samples, batch_size)
        X_batch, y_batch = X_data[indices], y_data[indices]
        _, loss_val, k_val, b_val = sess.run([ optimizer, loss, k, b ],
            feed_dict = { X : X_batch, y : y_batch })
        if (i+1) % display_step == 0:
            print('Эпоха %d: %.8f, k=%.4f, b=%.4f' %
                (i+1, loss_val, k_val, b_val))
```

1) спочатку створюємо випадковим чином вхідні дані X_data та y_data за таким алгоритмом:

▪ створюємо 1000 випадкових точок рівномірно на інтервалі [0; 1];

▪ підраховуємо для кожної точки x відповідну «правильну відповідь» y за формулою y = 2x+1+є, де є – випадково розподілений шум із дисперсією 2,

$є \sim N(є;0,2)$;

2) потім оголошуємо tf.placeholder для змінних X та y; на цьому етапі вже потрібно задати їм розмірність, і це в нашому випадку матриця розмірності (розмір міні-батча × 1) для X і просто вектор довжини розмір міні-батча для y;

3) далі ініціалізуємо змінні k та b; це змінні TensorFlow, які поки що жодних значень не мають, але будуть ініціалізовані стандартним нормальним розподілом для k і нулем для b;

4) потім ми встановлюємо власне суть моделі і при цьому будуємо функцію помилки $\sum(\hat{y}_i - y_i)^2$; зверніть увагу на функцію `reduce_sum`: на виході вона лише підраховує суму матриці по рядках, але користуватися треба саме нею, а не звичайною сумою або відповідними функціями з numpy, тому що так TensorFlow зможе куди більш ефективно оптимізувати процес обчислень;

5) вводимо змінну `optimizer` - оптимізатор, тобто власне алгоритм, який підраховуватиме градієнти та оновлюватиме ваги; ми вибрали стандартний оптимізатор стохастичного градієнтного спуску; Тепер нам важливо лише відзначити, що тепер щоразу, коли ми просимо TensorFlow підрахувати значення змінної optimizer, десь за лаштунками відбуватимуться оновлення змінних, від яких залежить оптимізована змінна `loss`, тобто k і b; по X та y оптимізації не буде, тому що значення `tf.placeholder` повинні бути жорстко задані, це вхідні дані;

6) записуємо великий цикл, що робить ці оновлення (тобто багато разів обчислює змінну `optimizer`); на кожній ітерації циклу ми беремо випадкове підмножина з `batch_size` (тобто 100) індексів даних та підраховуємо значення потрібних змінних; ми подаємо в функцію `sess.run` список змінних, які потрібно підрахувати (головне - "обчислити" змінну `optimizer`, інші потрібні тільки для виводу налагодження), і словник `feed_dict`, в який записуємо значення вхідних змінних, позначених раніше як `tf.placeholder`.

В результаті цей код виписуватиме поетапно зменшувану помилку і поступово уточнюються значення k і b:

```
Епоха 100: 5962.15625000, k=0.8925, b=0.0988
Епоха 200: 5312.11621094, k=0.9862, b=0.1927
Епоха 300: 3904.57006836, k=1.0761, b=0.2825...
Епоха 19900: 429.79974365, k=2.0267, b=0.9006
Епоха 20000: 378.41503906, k=2.0179, b=0.8902
```