

# Шпаргалка по PyTorch и TensorFlow / Torch Cheatsheet (XeLaTeX)

## Краткий справочник по основным операциям

## Содержание

- 1 **Тензор: Основа Всего** 1
- 2 **Автодифференцирование: Как Сеть Учится** 1
- 3 **Модель: Рецепт Преобразования Данных** 2
- 4 **Цикл Обучения: Шаг за Шагом к Результату** 2

### Зачем нужны DL Фреймворки?

PyTorch и TensorFlow — это мощные инструменты, которые сильно упрощают создание и обучение глубоких нейронных сетей (Deep Learning, DL). Они берут на себя сложную математику и оптимизацию, позволяя тебе сосредоточиться на архитектуре модели и данных.

- **Аналогия:** Представь, что строишь сложный механизм. Вместо того чтобы вытачивать каждую шестеренку вручную, ты используешь готовые стандартные блоки (слои, функции активации) и инструменты (оптимизаторы, расчет градиентов), которые предоставляют фреймворки.
- **Цель этой шпаргалки:** Понять самые базовые "строительные блоки" и процесс "сборки" (обучения) в PyTorch и TensorFlow. Не бойся синтаксиса, концепции очень похожи!

## 1 Тензор: Основа Всего

### Что такое Тензор?

**Тензор (Tensor)** — это фундаментальная структура данных в DL фреймворках. По сути, это многомерный массив чисел.

- 0-мерный тензор: скаляр (просто число).
- 1-мерный тензор: вектор (массив).
- 2-мерный тензор: матрица (таблица).
- 3-мерный тензор: куб чисел (например, цветное изображение RGB).
- и так далее...

**Аналогия:** Думай о тензоре как о супер-продвинутом NumPy массиве, который умеет работать на GPU и "помнит" вычисления для градиентов. В тензорах хранятся входные данные, веса модели, выходы слоев и т.д. *Микро-уточнение для TensorFlow:* В TF основной неизменяемый

тип — `tf.Tensor` (результат операций, создается через `tf.constant`, `tf.zeros` и т.д.). Для изменяемых параметров модели, которые нужно отслеживать для градиентов, используется `tf.Variable`.

### Создание Тензоров

```
1 # === PyTorch ===
2 import torch
3
4 # Создание из списка Python
5 t_list_pt = torch.tensor([[1., 2.], [3., 4.]])
6
7 # Создание тензора из нулей
8 t_zeros_pt = torch.zeros((2, 3)) # Форма (2 строки, 3 столбца)
9
10 # Создание тензора из случайных чисел
11 t_rand_pt = torch.rand((2, 2))
12
13 # === TensorFlow ===
14 import tensorflow as tf
15
16 # Создание из списка Python (создает tf.Tensor)
17 t_list_tf = tf.constant([[1., 2.], [3., 4.]])
18
19 # Создание тензора из нулей (создает tf.Tensor)
20 t_zeros_tf = tf.zeros((2, 3)) # Форма (2 строки, 3 столбца)
21
22 # Создание тензора из случайных чисел (создает tf.Tensor)
23 t_rand_tf = tf.random.uniform((2, 2))
24 # tf.Variable обычно создается отдельно для весов модели
```

## 2 Автодифференцирование: Как Сеть Учится

### Зачем это нужно?

Обучение нейросети — это подбор таких весов, чтобы ошибка (loss) была минимальной. Для этого используется **градиентный спуск**. Нам нужно знать, как небольшое изменение каждого веса влияет на итоговую ошибку. Это и есть **градиент**. Вычислять его вручную для сложных сетей нереально. **Автоматическое дифференцирование (Autograd)** — это механизм, который автоматически вычисляет градиенты функции потерь по всем параметрам модели. **Аналогия:** Представь "волшебного бухгалтера", который следит за каждой математической операцией в сети. Когда ты получаешь итоговую ошибку, ты можешь спросить у него: "Эй, как изменение вот этого конкретного веса в самом начале повлияло на эту ошибку?". И он мгновенно даст

тебе ответ (градиент).

### Autograd/GradientTape на практике

```
1 # === PyTorch (Autograd) ===
2 import torch
3
4 # Тензор, для которого нужно считать градиенты
5 x_pt = torch.tensor(2.0, requires_grad=True)
6 w_pt = torch.tensor(3.0, requires_grad=True)
7 b_pt = torch.tensor(1.0, requires_grad=True)
8
9 # Операции
10 y_pt = w_pt * x_pt + b_pt # y = 3 * 2 + 1 = 7
11
12 # Вычисляем градиенты d(y)/d(w), d(y)/d(x), d(y)/d(b)
13 y_pt.backward()
14
15 # Градиенты сохраняются в атрибуте .grad
16 print(f"PyTorch dy/dw: {w_pt.grad}") # Ожидаем x = 2
17 print(f"PyTorch dy/dx: {x_pt.grad}") # Ожидаем w = 3
18 print(f"PyTorch dy/db: {b_pt.grad}") # Ожидаем 1
19
20 # === TensorFlow (GradientTape) ===
21 import tensorflow as tf
22
23 # Используем tf.Variable, чтобы TF отслеживал их
24 x_tf = tf.Variable(2.0)
25 w_tf = tf.Variable(3.0)
26 b_tf = tf.Variable(1.0)
27
28 # Операции должны быть внутри контекста GradientTape
29 with tf.GradientTape() as tape:
30     y_tf = w_tf * x_tf + b_tf # y = 3 * 2 + 1 = 7
31
32 # Вычисляем градиенты
33 # tape.gradient(целевая_переменная,
34 #               список_переменных_по_которым_считаем)
35 gradients = tape.gradient(y_tf, {'w': w_tf, 'x': x_tf, 'b': b_tf})
36
37 print(f"TensorFlow dy/dw: {gradients['w']}") # Ожидаем
38 print(f"TensorFlow dy/dx: {gradients['x']}") # Ожидаем
39 print(f"TensorFlow dy/db: {gradients['b']}") # Ожидаем
40
```

### Важно!

В PyTorch нужно явно указывать `requires_grad=True` для тензоров, по которым будут считаться градиенты (обычно это параметры модели). В TensorFlow для `tf.Variable` отслеживание включено по умолчанию, а операции нужно помещать внутрь `tf.GradientTape()`.

### 3 Модель: Рецепт Преобразования Данных

#### Что такое модель?

**Модель (Model)** — это, по сути, функция (часто очень сложная), которая преобразует входные данные в выходные (предсказание). В DL она обычно состоит из последовательности **слоев (Layers)**. Каждый слой выполняет определенное преобразование над данными, используя свои **параметры (веса)**. **Аналогия:** Модель — это как кулинарный рецепт. Входные данные — ингредиенты. Слои — шаги рецепта (смешать, запечь, нарезать). Веса слоя — это параметры шага (сколько муки, температура духовки). Выход модели — готовое блюдо (предсказание).

#### Пример простой линейной модели

```
1 # === PyTorch (nn.Module) ===
2 import torch
3 import torch.nn as nn
4
5 class SimpleLinearPT(nn.Module):
6     def __init__(self, input_dim, output_dim):
7         super(SimpleLinearPT, self).__init__()
8         # Определяем слои в конструкторе
9         self.linear = nn.Linear(input_dim, output_dim)
10
11     # Метод forward определяет прямой проход
12     def forward(self, x):
13         # Определяем порядок вычислений (прямой проход)
14         out = self.linear(x)
15         return out
16
17 # Создаем модель: входная размерность 5, выходная 1
18 model_pt = SimpleLinearPT(input_dim=5, output_dim=1)
19 print("PyTorch Model:", model_pt)
20
21 # === TensorFlow (tf.keras.Model / tf.keras.Sequential)
22 ===
23 import tensorflow as tf
24 from tensorflow.keras.layers import Dense
25 from tensorflow.keras import Model, Sequential
26
27 # Простой способ через Sequential (для линейной
28 # последовательности слоев)
29 model_tf_seq = Sequential([
30     Dense(units=1, input_shape=(5,)) # units - выходная
31     # размерность
32 ])
33 print("\nTensorFlow Sequential Model:")
34 model_tf_seq.build(input_shape=(None, 5)) # Явно строим
35 # модель (None - размер батча)
36 model_tf_seq.summary()
37
38 # Более гибкий способ через наследование Model
39 class SimpleLinearTF(Model):
40     def __init__(self, output_dim):
41         super(SimpleLinearTF, self).__init__()
42         # Определяем слои
43         self.dense = Dense(units=output_dim)
44
45     # Метод call определяет прямой проход (аналог
46     # forward в PyTorch)
47     def call(self, x):
48         # Определяем прямой проход
49         return self.dense(x)
```

```
46 # Создаем модель: выходная размерность 1
47 model_tf_custom = SimpleLinearTF(output_dim=1)
48 # Чтобы увидеть summary, нужно "вызвать" модель на
49 # данных нужной формы
50 _ = model_tf_custom(tf.zeros((1, 5))) # Прогоняем
51 # пустой тензор для построения
52 print("\nTensorFlow Custom Model:")
53 model_tf_custom.summary()
```

### 4 Цикл Обучения: Шаг за Шагом к Результату

#### Как происходит обучение?

Обучение — это итеративный процесс, где модель настраивает свои веса, чтобы минимизировать ошибку на обучающих данных. Каждый проход по данным называется **эпохой**. Внутри эпохи данные обычно делятся на **батчи (batches)**. **Аналогия:** Студент готовится к экзамену (обучение). У него есть учебник (данные). Он читает главу за главой (эпохи). Каждую главу он прорабатывает по частям (батчи). Для каждой части: 1. Пытается ответить на вопросы (делает предсказание - **forward pass**). 2. Сравнивает свои ответы с правильными (считает ошибку - **loss**). 3. Анализирует, где ошибся и почему (считает градиенты - **backward pass**). 4. Корректирует свое понимание материала (обновляет веса - **optimizer step**). 5. Перед следующей частью "очищает" голову от предыдущих размышлений (обнуляет градиенты - **zero grad**).

## Концептуальный цикл обучения (Псевдокод)

```
1 # --- Общие компоненты ---
2 # model = ... (ваша модель PyTorch или TensorFlow)
3 # criterion = ... (функция потерь, напр., nn.MSELoss()
  или tf.keras.losses.MeanSquaredError())
4 # optimizer = ... (оптимизатор, напр.,
  torch.optim.SGD() или tf.keras.optimizers.SGD())
5 # train_loader = ... (загрузчик данных, который выдает
  батчи X, y)
6 # num_epochs = ... (количество эпох)
7
8 # === PyTorch ===
9 # for epoch in range(num_epochs):
10 #     for i, (inputs, labels) in
  enumerate(train_loader):
11 #         # 1. Forward pass: получить предсказания
12 #         outputs = model(inputs)
13 #
14 #         # 2. Calculate loss: сравнить с реальными
  метками
15 #         loss = criterion(outputs, labels)
16 #
17 #         # 3. Backward pass: вычислить градиенты
18 #         optimizer.zero_grad() # !!! ОБНУЛИТЬ
  градиенты перед backward() !!!
19 #         loss.backward()
20 #
21 #         # 4. Optimizer step: обновить веса модели
22 #         optimizer.step()
23 #
24 #     # (Опционально) Логирование потерь за эпоху...
25
```

```
26 # === TensorFlow (с использованием model.fit - проще
  для старта) ===
27 # model.compile(optimizer=optimizer, loss=criterion,
  metrics=['accuracy']) # Опционально метрики
28 #
29 # history = model.fit(train_loader, # Или просто
  X_train, y_train
30 #                       epochs=num_epochs,
31 #                       #
  validation_data=validation_loader # Опционально
32 #                       )
33 # print("Обучение завершено!")
34
35 # === TensorFlow (ручной цикл - для понимания) ===
36 # for epoch in range(num_epochs):
37 #     for step, (x_batch_train, y_batch_train) in
  enumerate(train_loader):
38 #         with tf.GradientTape() as tape:
39 #             # 1. Forward pass
40 #             logits = model(x_batch_train,
  training=True) # Указать training=True для некоторых
  слоев
41 #             # 2. Calculate loss
42 #             loss_value = criterion(y_batch_train,
  logits)
43 #
44 #             # 3. Backward pass (вычисление градиентов)
45 #             grads = tape.gradient(loss_value,
  model.trainable_weights)
46 #
47 #             # 4. Optimizer step (применение градиентов)
48 #             optimizer.apply_gradients(zip(grads,
  model.trainable_weights))
49 #
50 #     # (Опционально) Логирование потерь за эпоху...
```

## Ключевые моменты цикла:

**\* Forward Pass:** Прогон данных через модель для получения предсказания. **\* Loss Calculation:** Вычисление функции потерь (насколько предсказание отличается от истины). **\* Backward Pass:** Вычисление градиентов функции потерь по параметрам модели (используя autograd / GradientTape). **\* Optimizer Step:** Обновление параметров модели с использованием вычисленных градиентов и выбранного алгоритма оптимизации (SGD, Adam и т.д.). **\* Zero Gradients (PyTorch):** \*\*Критически важно\*\* обнулять градиенты перед каждым backward() вызовом в PyTorch (optimizer.zero\_grad()), иначе градиенты будут накапливаться от предыдущих батчей. В TensorFlow при использовании GradientTape это происходит автоматически для каждого вызова tape.gradient.