# Автокодировщик MNIST

Направление: нейронные сети

Выполнил: Калиниченко Владислав Константинович

Дата отчета: 01.07.2025

# Введение

Автокодирщик (autoencoder) — это архетиктура нейронной сети, которая посредством обучения без учителя способна «сжимать» представление об объекте и «разжимать» в близкое к исходному состоянию.

В процессе сжатия (encode) модель переводить данные об объекте в латентное пространство, где схожие объекты находятся ближе. В таком пространстве объекты хранят меньше признаков, но они сложно интрепретируемы для человека.

В процессе разжимания (decode) модель переводить данные об объекте из латентного пространство в исходное, где признаки обладают тем же смыслом, что и изначально.

Автокодировщик можно использовать как обертку: сжать → передать → разжать. Это полезно для уменьшении передаваемой информации и для безопасности, потому что данные автоматически шифруются.

Автокодировщик можно использовать как востановитель данных. Если данные об исходном объекте повреждены или защумлены, то его можно провести через автокодировщик и он заполнит утраченную информацию.

Как и любая нейронная сеть, автокодировщик требует определения: архитектуры, обучения, тестирования.

В части «архитектура» будет определена архитектура обучаемой модели и объяснены причины построения именно такой архитектуры.

В части «обучение» будут определены инструменты обучения и построен процесс обучения.

В части «тестирование» будет протестирована обученная модель и получены необходимые графики и диаграммы.

# Архитектура

В данной работе автокодировщик будет представлять собой линейную нейронную сеть с несколькими линейными слоями, между которыми будут функции активации. Особенностями данной модели будут:

1) Плавное уменьшение размеров слоев до середины, с последующий увеличением их до конца

2) Симметричность архитектуры относительно центра (размер слоев, функции активации)

Отличаться может лишь начало модели (для подготовки данных) и её конец (для правильного интрепретирования). Также будет произведено условное разделение модели на сжимающую (encode), отвечающую за часть сжатия информации об объекте и переводе его в латентное пространство, и разжимающую (decode), отвечающую за часть разжатия информации об объекте и возвращении его из латентного пространства в исходное.

Так как обучение модели сильно зависит от архитектуры, то имеет смысл построить несколько для дальнейшего тестирования и анализа.

1) При малом количестве слоев модель может не научиться выделять важные признаки.

2) При большом количестве слоев увеличивается шанс переобучении, а так же замедляется процесс обучения

3) При малом размере латентного пространства может не хватить признаков для описания объектов

4) При большой размер латентного пространства может быть избыточным (может привести к переобучению?)

5) Резкое изменение слоев может привести к потере важных признаков при сжатии.

Учитывая эти факты, будет произведено построение трех моделей схожей архитектуры

Linear – линейный слой

ReLU – функция активации ReLU

Tanh – функция активации Tanh

**model_1 – эталонная**

```
(encoder): Sequential(
    (0): Linear(in_features=784, out_features=196, bias=True)
    (1): ReLU()
    (2): Linear(in_features=196, out_features=49, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=49, out_features=196, bias=True)
```

```
    (1): ReLU()
    (2): Linear(in_features=196, out_features=784, bias=True)
    (3): Tanh()
  )
```

**model_2 – с ReLU в конце**

```
(encoder): Sequential(
    (0): Linear(in_features=784, out_features=196, bias=True)
    (1): ReLU()
    (2): Linear(in_features=196, out_features=49, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=49, out_features=196, bias=True)
    (1): ReLU()
    (2): Linear(in_features=196, out_features=784, bias=True)
    (3): ReLU()
  )
```

**model_3 - глубокая**

```
(encoder): Sequential(
    (0): Linear(in_features=784, out_features=392, bias=True)
    (1): ReLU()
    (2): Linear(in_features=392, out_features=196, bias=True)
    (3): ReLU()
    (4): Linear(in_features=196, out_features=98, bias=True)
    (5): ReLU()
    (6): Linear(in_features=98, out_features=49, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=49, out_features=98, bias=True)
    (1): ReLU()
    (2): Linear(in_features=98, out_features=196, bias=True)
    (3): ReLU()
    (4): Linear(in_features=196, out_features=392, bias=True)
    (5): ReLU()
    (6): Linear(in_features=392, out_features=784, bias=True)
    (7): Tanh()
  )
```

**model_4 – резкое уменьшение**

```
(encoder): Sequential(
    (0): Linear(in_features=784, out_features=156, bias=True)
    (1): ReLU()
    (2): Linear(in_features=156, out_features=32, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=32, out_features=156, bias=True)
    (1): ReLU()
    (2): Linear(in_features=156, out_features=784, bias=True)
    (3): Tanh()
  )
```

# Обучение

В качестве функции ошибки был выбран MSE, а метод оптимизации — Adam с lr = 0.001. Количество эпох для каждой модели изначально выбрано 20, так как при тестировании разных архитектур такого количества было достаточно. Размер батча был выбран 64.

Так как все модели являются линейными, то перед использованием изображений чисел в обучении, матрицу (представление числа в виде двумерного массива) следует превратить в вектор (представление числа в виде одномерного массива) размера 28*28=784.

Для последующего анализа результатов, были также сохранены итоговое время обучения каждой модели (20 эпох).

# Тестирование

Все 4 модели были обучены.

Model_1:

- Time: 397.1 сек

- Loss: 0.0163

Model_2:

- Time: 438.1 сек

- Loss: 0.8864

- Особое: значение loss почти не менялось

Model_3:

- Time: 666.5 сек

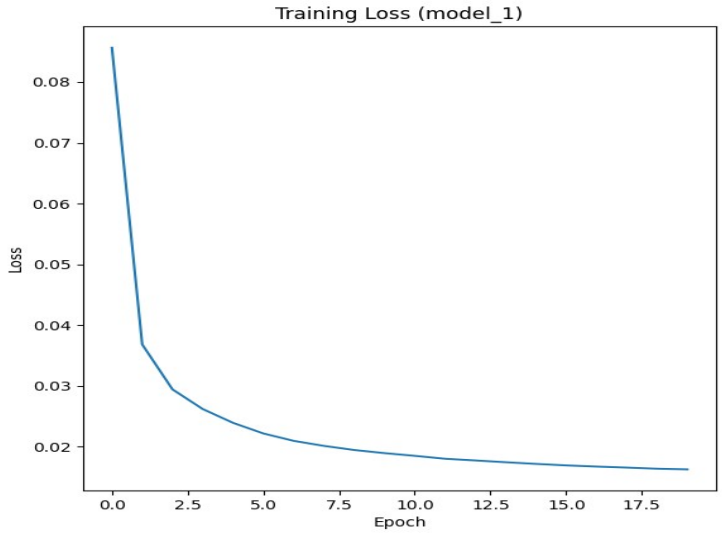
- Loss: 0.0275

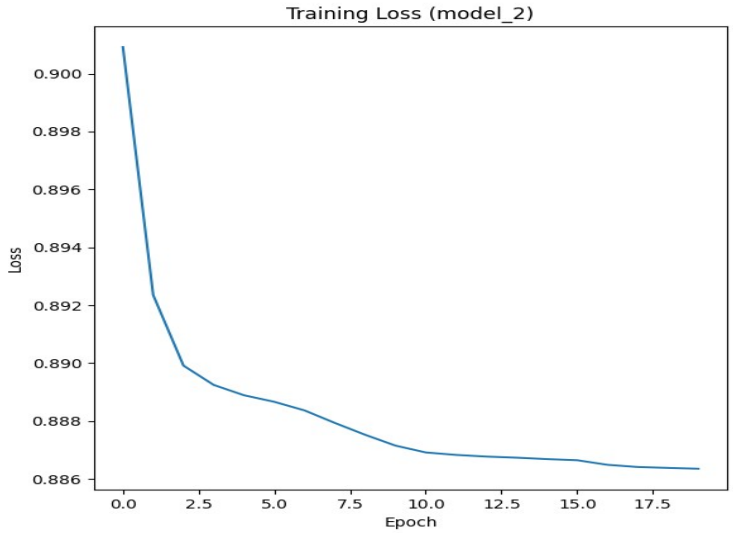
Model_4:

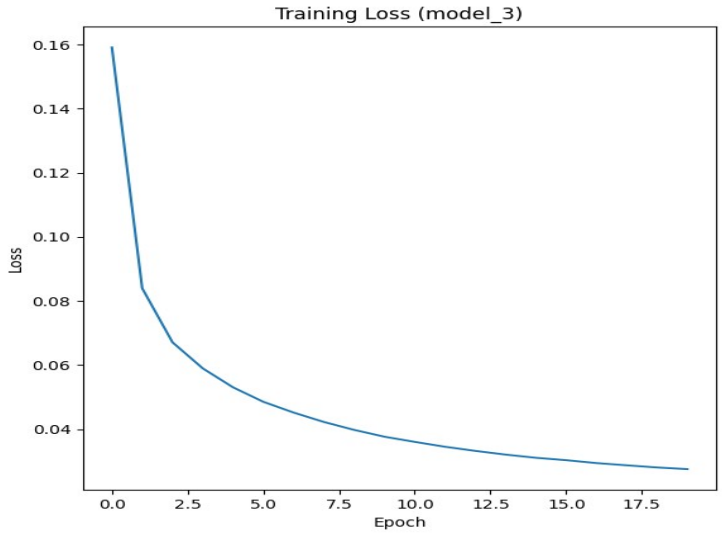
- Time: 375.4 сек

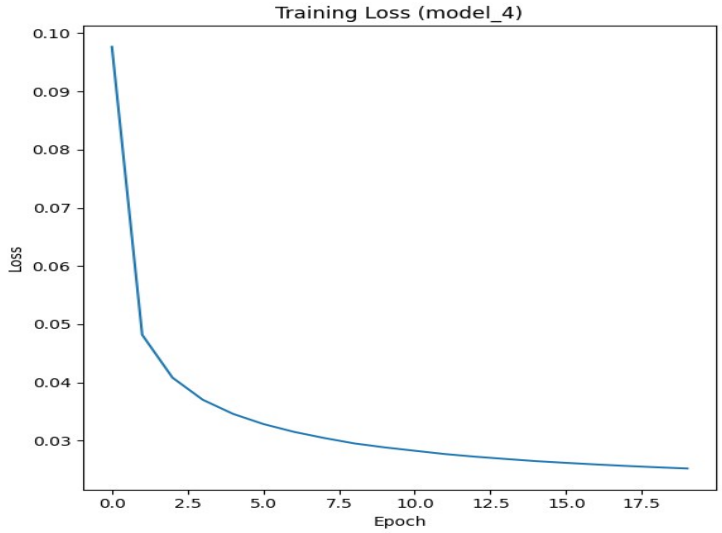
- Loss: 0.0252

Графики функции потерь:

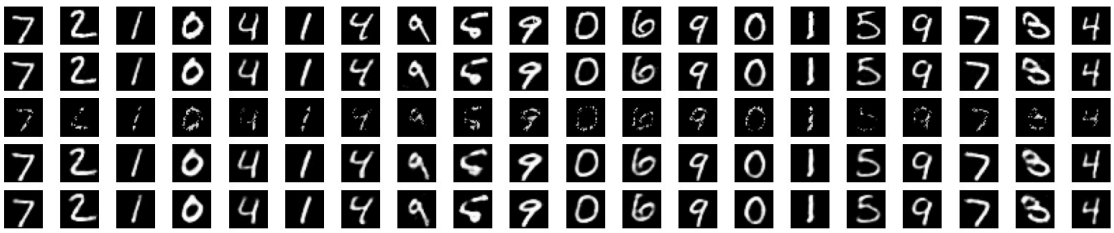

Training Loss (model_1)
Loss
Epoch
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Training Loss (model_2)
Loss
Epoch
0.900
0.898
0.896
0.894
0.892
0.890
0.888
0.886
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Training Loss (model_3)
Loss
Epoch
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Training Loss (model_4)
Loss
Epoch
0.10
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

Изначальные изображения цифр и выданные моделью (сначала изначальные изображения цифры, а дальше по порядку модели с соответствующим номером):



Как видно из изображений вторая модель не справилась с задачей, а остальные выдают необходимый результат. Если сравнивать оставшиеся модели, то разницы невооруженным взглядом не заметить.

Однако из всех моделей с установленными параметрами обучения первая обладает наименьшей loss, меньше даже чем у более глубокой. Однако это не означает, что более глубокая модель всегда обладает хуже результатом на MNIST, но следует сразу учитывать более длительное обучение.

Также следует отметить необходимость подбора оптимального размера слоев. Так четвертая модель, отличавшаяся от первой меньшими размерами слоев, обладает существенно большим значением loss. Хотя иногда даже такой результат может быть приемлемым, а меньший размер модели и более быстрое обучение могут оказаться более значимыми, из-за чего такие архитектуры тоже могут быть использованы.

# Вывод

В данной работе была изучена такая архитектура нейронной сети, как автокодировщик. Модели разной архитектуры были спроектированы, обучены и протестированы, а результаты проанализированы.

Во-первых, следует аккуратно выбирать функции активации, потому что неправильная функция даже в одном месте (в данном случае в конце) может сильно ухудшить результат.

Во-вторых, следует правильно подбирать количество слоев и их размеры: более глубокие модели обучаются дольше, а их результат не всегда лучше менее глубоких; слои меньшего размера могут ухудшить итоговый результат. Однако следует учитывать необходимые условия для модели. Иногда требуется легковесная модель, даже если её результат будет хуже.