НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет ПИиКТ

Системы искусственного интеллекта

Лабораторная работа № 5

Выполнил студент

Набокова Алиса Владиславовна

Группа № P3320

Преподаватель: Королёва Юлия Александровна

г. Санкт-Петербург

2025

Задание:

Необходимо реализовать простую нейронную сеть(не обязательно сверточную(!)) и обучить ее на датасете

- Каждый слой прописывается руками, перекладыванием матричек/тензоров из состояния А в состояние Б
- Возможные слои/классы, которые вам понадобятся для успешного закрытия лабы: Linear/Dense, Dropout, Conv2d, MaxPool2D, ReLU, Softmax, Flatten, Adam.
- В классе не должно быть волшебных истинно торчовских методов
- Выводим графики нескольких моделей, с экспериментами (разный порядок слоев, разные входные/выходные параметры), выбираем свою лучшую модельку
- dataset https://www.kaggle.com/datasets/hojjatk/mnist-dataset

На листингах 1-4 представлены реализации основных слоёв нейронной сети: полносвязный слой, функция активации ReLU, слой Softmax и слой преобразования входных данных Flatten. Каждый слой реализует прямое и обратное распространение ошибки

Листинг 1. Реализация полносвязного слоя (Linear)

```
class Linear:
    def __init__(self, in_features, out_features):
        self.W = torch.randn(in_features, out_features) * 0.01
        self.b = torch.zeros(out_features)

    def forward(self, x):
        self.x = x
        return x @ self.W + self.b

    def backward(self, grad):
```

```
        self.dW = self.x.T @ grad
        self.db = grad.sum(dim=0)
        return grad @ self.W.T
```

Листинг 2. Реализация функции активации ReLU

```
class ReLU:
    def forward(self, x):
        self.mask = x > 0
        return x * self.mask

    def backward(self, grad):
        return grad * self.mask
```

Листинг 3. Реализация слоя Softmax

```
class Softmax:
    def forward(self, x):
        exp = torch.exp(x - x.max(dim=1, keepdim=True).values)
        self.out = exp / exp.sum(dim=1, keepdim=True)
        return self.out

    def backward(self, grad):
        return grad
```

Листинг 4. Реализация слоя Flatten

```
class Flatten:
    def forward(self, x):
```

```python
        self.shape = x.shape
        return x.view(x.shape[0], -1)

    def backward(self, grad):
        return grad.view(self.shape)
```

На листингах 5-6 представлены реализация функции потерь кросс-энтропии и вспомогательная функция вычисления точности классификации

Листинг 5. Реализация функции потерь CrossEntropyLoss

```python
class CrossEntropyLoss:
    def forward(self, probs, targets):
        self.probs = probs
        self.targets = targets
        return -torch.log(probs[range(len(targets)), targets]).mean()

    def backward(self):
        grad = self.probs.clone()
        grad[range(len(self.targets)), self.targets] -= 1
        return grad / len(self.targets)
```

Листинг 6. Функция вычисления точности классификации

```python
def accuracy_from_probs(probs, labels):
    preds = torch.argmax(probs, dim=1)
    return (preds == labels).float().mean().item()
```

На листингах 7-8 представлены реализация слоя Dropout для регуляризации модели и алгоритма оптимизации Adam

Листинг 7. Реализация слоя Dropout

```
class Dropout:
   def __init__(self, p=0.5):
       self.p = p

   def forward(self, x, train=True):
       if train:
           self.mask = (torch.rand_like(x) > self.p)
           return x * self.mask
       return x

   def backward(self, grad):
       return grad * self.mask
```

Листинг 8. Реализация алгоритма оптимизации Adam

```
class Adam:
   def __init__(self, params, lr=1e-3, beta1=0.9, beta2=0.999):
       self.params = params
       self.lr = lr
       self.m = [torch.zeros_like(p) for p in params]
```

```
        self.v = [torch.zeros_like(p) for p in params]
        self.t = 0

    def step(self, grads):
        self.t += 1
        for i, (p, g) in enumerate(zip(self.params, grads)):
            self.m[i] = 0.9 * self.m[i] + 0.1 * g
            self.v[i] = 0.999 * self.v[i] + 0.001 * (g ** 2)
            p -= self.lr * self.m[i] / (torch.sqrt(self.v[i]) + 1e-8)
```

На листингах 9-10 представлен процесс инициализации нейронной сети и цикл её обучения с вычислением функции потерь и точности на обучающей и валидационной выборках

Листинг 9. Инициализация архитектуры нейронной сети

```
flatten = Flatten()
fc1 = Linear(784, 256)
relu1 = ReLU()
dropout = Dropout(p=0.2)
fc2 = Linear(256, 128)
relu2 = ReLU()
fc3 = Linear(128, 10)
softmax = Softmax()
loss_fn = CrossEntropyLoss()
params = [fc1.W, fc1.b, fc2.W, fc2.b, fc3.W, fc3.b]
optimizer = Adam(params, lr=1e-3)
```

Листинг 10. Цикл обучения и валидации нейронной сети

```
train_losses = []

val_losses = []

train_accuracies = []

val_accuracies = []


epochs = 15


for epoch in range(epochs):

   #train

   total_train_loss = 0

   total_train_acc = 0

   n_batches = 0

   for images, labels in train_loader:


       x = flatten.forward(images)

       x = fc1.forward(x)

       x = relu1.forward(x)

       x = dropout.forward(x, train=True)

       x = fc2.forward(x)

       x = relu2.forward(x)

       x = fc3.forward(x)

       probs = softmax.forward(x)



       loss = loss_fn.forward(probs, labels)

       total_train_loss += loss.item()
```

```
        total_train_acc += accuracy_from_probs(probs, labels)
        n_batches += 1


        grad = loss_fn.backward()
        grad = softmax.backward(grad)
        grad = fc3.backward(grad)
        grad = relu2.backward(grad)
        grad = fc2.backward(grad)
        grad = dropout.backward(grad)
        grad = relu1.backward(grad)
        grad = fc1.backward(grad)


        grads = [fc1.dW, fc1.db, fc2.dW, fc2.db, fc3.dW, fc3.db]
        optimizer.step(grads)

    train_losses.append(total_train_loss / n_batches)
    train_accuracies.append(total_train_acc / n_batches)

    #validation
    total_val_loss = 0
    total_val_acc = 0
    n_val_batches = 0
    for images, labels in val_loader:
        x = flatten.forward(images)
        x = fc1.forward(x)
```

```
        x = relu1.forward(x)

        x = fc2.forward(x)

        x = relu2.forward(x)

        x = fc3.forward(x)

        probs = softmax.forward(x)


        loss = loss_fn.forward(probs, labels)

        total_val_loss += loss.item()

        total_val_acc += accuracy_from_probs(probs, labels)

        n_val_batches += 1


    val_losses.append(total_val_loss / n_val_batches)

    val_accuracies.append(total_val_acc / n_val_batches)


    print(

        f"Epoch {epoch+1}: "

        f"train_loss={train_losses[-1]:.4f}, "

        f"train_acc={train_accuracies[-1]*100:.2f}%, "

        f"val_loss={val_losses[-1]:.4f}, "

        f"val_acc={val_accuracies[-1]*100:.2f}%"

    )
```

На листингах 11-14 представлены функции оценки качества модели на тестовой выборке, сохранения и загрузки параметров модели, а также пример применения обученной модели для распознавания рукописного изображения

Листинг 11. Оценка точности на тестовой выборке

```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))


axes[0].plot(train_losses, label="train loss")
axes[0].plot(val_losses, label="val loss")
axes[0].set_xlabel("Epoch")
axes[0].set_ylabel("Loss")
axes[0].legend()
axes[0].grid(True)
axes[0].set_title("Loss")


axes[1].plot(train_accuracies, label="train accuracy")
axes[1].plot(val_accuracies, label="val accuracy")
axes[1].set_xlabel("Epoch")
axes[1].set_ylabel("Accuracy")
axes[1].legend()
axes[1].grid(True)
axes[1].set_title("Accuracy")


plt.tight_layout()
plt.show()
```

Листинг 12. Оценка качества модели на тестовой выборке

```python
def accuracy(loader):
    correct = 0
    total = 0
    for images, labels in loader:
        x = flatten.forward(images)
        x = fc1.forward(x)
        x = relu1.forward(x)
        x = fc2.forward(x)
        x = relu2.forward(x)
        x = fc3.forward(x)
        probs = softmax.forward(x)
        preds = torch.argmax(probs, dim=1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)
    return correct / total


print("Test accuracy:", accuracy(test_loader))
```

Листинг 13. Сохранение модели

```python
def save_model(filename="mymodel.pth"):
    state = {
        "fc1.W": fc1.W,
        "fc1.b": fc1.b,
        "fc2.W": fc2.W,
        "fc2.b": fc2.b,
        "fc3.W": fc3.W,
```

```
        "fc3.b": fc3.b,
    }
    torch.save(state, filename)
```

Листинг 14. Загрузка модели

```
def load_model(filename="mymodel.pth"):
    state = torch.load(filename)
    fc1.W = state["fc1.W"]
    fc1.b = state["fc1.b"]
    fc2.W = state["fc2.W"]
    fc2.b = state["fc2.b"]
    fc3.W = state["fc3.W"]
    fc3.b = state["fc3.b"]
```

Листинг 14. Распознавание изображения

```
load_model("/content/mymodel_with_1hidden_15ep_reg.pth")

img = Image.open("/content/0.png").convert("L")
img = img.resize((28, 28))
img = PIL.ImageOps.invert(img)
x = transform(img).unsqueeze(0)
x_flat = flatten.forward(x)
x1 = fc1.forward(x_flat)
x1 = relu1.forward(x1)
x2 = fc2.forward(x1)
x2 = relu2.forward(x2)
x3 = fc3.forward(x2)
```

```
probs = softmax.forward(x3)

pred = torch.argmax(probs, dim=1).item()

print("Predicted digit:", pred)
```

Эксперименты:

Model 1

Linear(784, 128)

relu

Linear(128, 10)

softmax

epochs: 5, lr=0.001

оценка модели:

```
Epoch 1: train_loss=0.3176, train_acc=90.33%, val_loss=0.2385, val_acc=93.23%

Epoch 2: train_loss=0.1711, train_acc=94.92%, val_loss=0.1872, val_acc=94.43%

Epoch 3: train_loss=0.1362, train_acc=95.86%, val_loss=0.1507, val_acc=95.88%

Epoch 4: train_loss=0.1166, train_acc=96.46%, val_loss=0.1555, val_acc=95.83%

Epoch 5: train_loss=0.1026, train_acc=96.93%, val_loss=0.1458, val_acc=95.88%
```

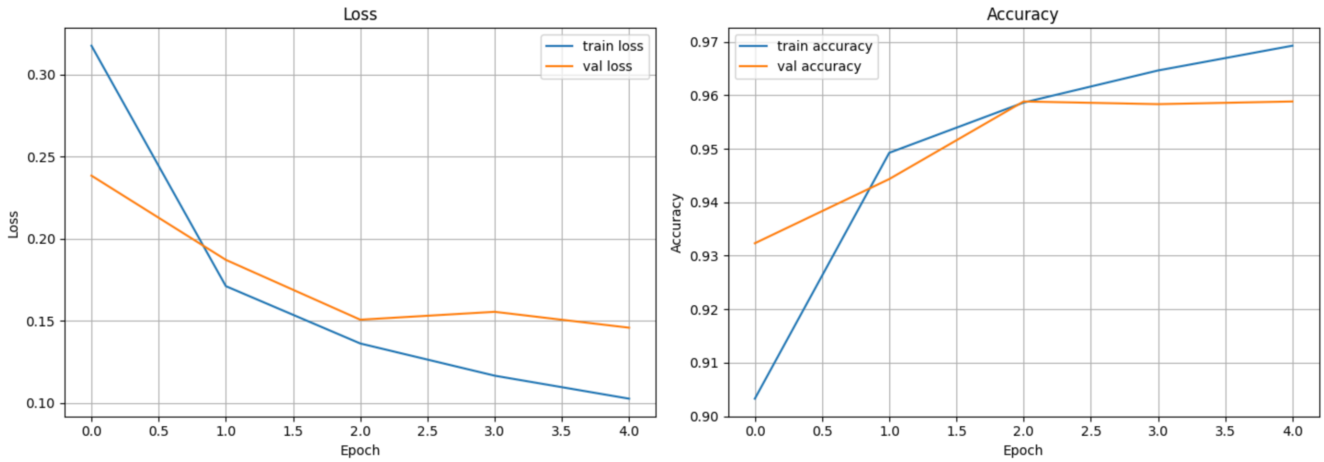

Рисунок 1. Потери и точности для модели 1

```
Test accuracy: 0.9687
```

Model 2

Linear(784, 128)

relu

dropout(p=0.2)

Linear(128, 10)

softmax

epochs: 5, lr=0.001

оценка модели:

```
Epoch 1: train_loss=0.4008, train_acc=87.49%, val_loss=0.2447, val_acc=92.92%

Epoch 2: train_loss=0.2510, train_acc=92.46%, val_loss=0.2220, val_acc=93.82%

Epoch 3: train_loss=0.2144, train_acc=93.56%, val_loss=0.1878, val_acc=94.70%

Epoch 4: train_loss=0.1941, train_acc=94.06%, val_loss=0.1774, val_acc=95.13%
```

```
Epoch 5: train_loss=0.1837, train_acc=94.43%, val_loss=0.1623, val_acc=95.45%
```

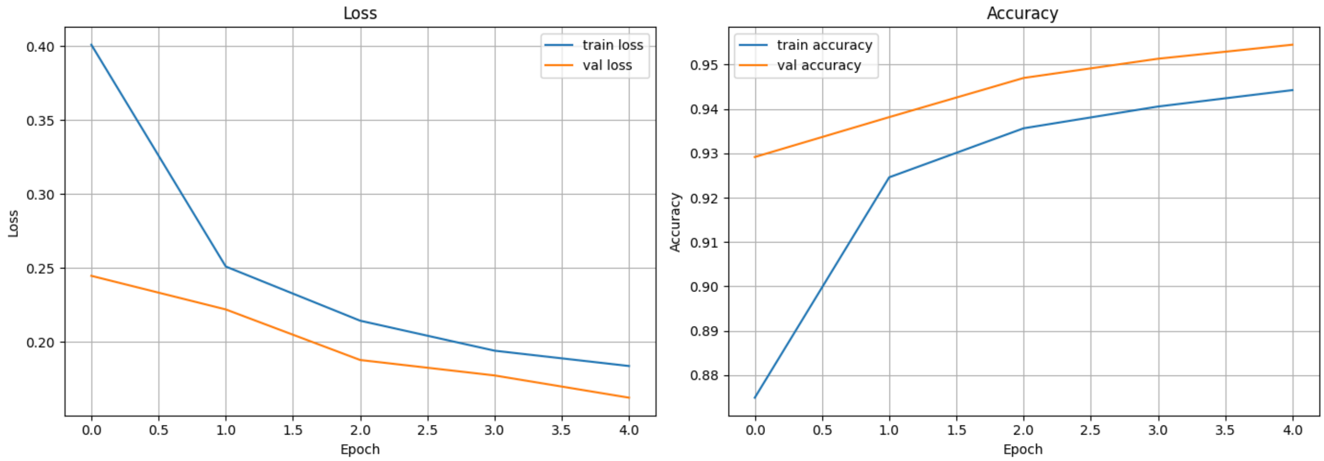

Рисунок 2. Потери и точности для модели 2

```
Test accuracy: 0.9489
```

Model 3

Linear(784, 256)

relu

dropout(p=0.2)

Linear(256, 10)

softmax

epochs: 5, lr=0.001

оценка модели:

```
Epoch 1: train_loss=0.3363, train_acc=89.47%, val_loss=0.1993, val_acc=94.37%

Epoch 2: train_loss=0.1897, train_acc=94.30%, val_loss=0.1610, val_acc=95.50%
```

```
Epoch 3: train_loss=0.1567, train_acc=95.14%, val_loss=0.1435, val_acc=95.88%
Epoch 4: train_loss=0.1379, train_acc=95.77%, val_loss=0.1504, val_acc=95.87%
Epoch 5: train_loss=0.1247, train_acc=96.13%, val_loss=0.1366, val_acc=96.38%
```

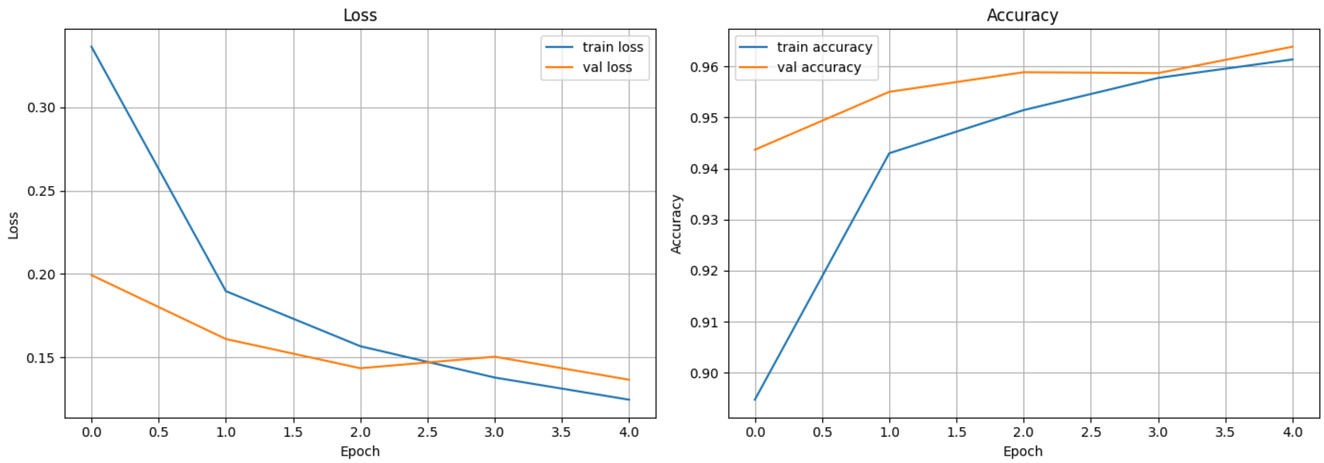

Рисунок 3. Потери и точности для модели 3

```
Test accuracy: 0.9627
```

Model 4

Linear(784, 256)

relu

Linear(256, 128)

relu

Linear(128, 10)

softmax

epochs: 5, lr=0.001

оценка модели:

```
Epoch 1: train_loss=0.3197, train_acc=89.83%, val_loss=0.1885, val_acc=94.60%
Epoch 2: train_loss=0.1465, train_acc=95.43%, val_loss=0.1689, val_acc=94.87%
Epoch 3: train_loss=0.1139, train_acc=96.55%, val_loss=0.1345, val_acc=96.13%
Epoch 4: train_loss=0.0947, train_acc=97.03%, val_loss=0.1315, val_acc=96.35%
Epoch 5: train_loss=0.0804, train_acc=97.41%, val_loss=0.1044, val_acc=96.93%
```

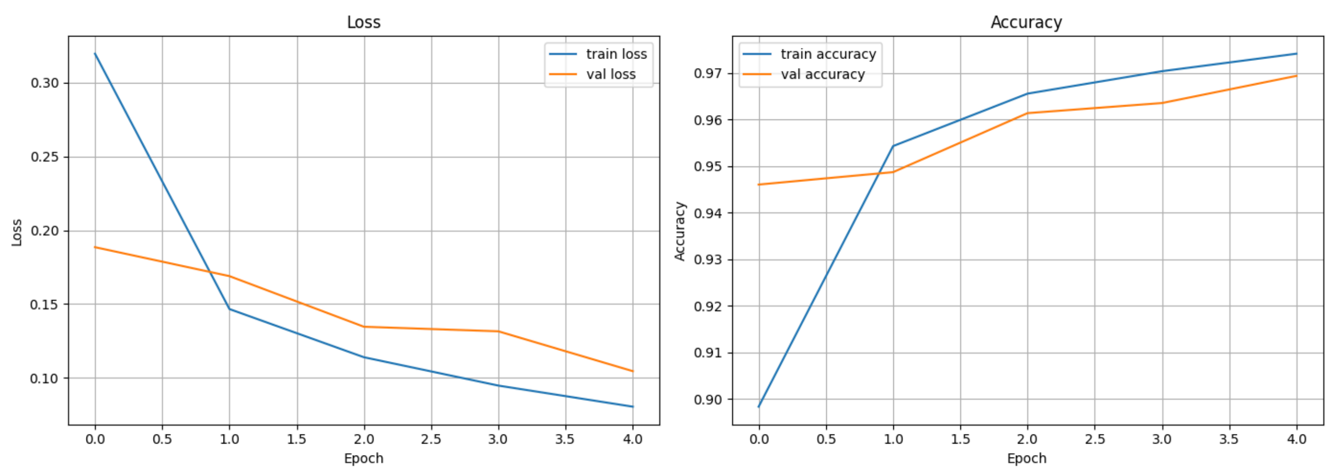

Рисунок 4. Потери и точности для модели 4

```
Test accuracy: 0.9714
```

Model 5

Linear(784, 256)

relu

dropout(p=0.25)

Linear(256, 128)

relu

Linear(128, 10)

softmax

epochs: 10, lr=0.001

```
оценка модели:

Epoch 1: train_loss=0.4001, train_acc=87.20%, val_loss=0.2211, val_acc=93.83%

Epoch 2: train_loss=0.2145, train_acc=93.40%, val_loss=0.1871, val_acc=94.98%

Epoch 3: train_loss=0.1810, train_acc=94.27%, val_loss=0.1795, val_acc=95.27%

Epoch 4: train_loss=0.1631, train_acc=94.91%, val_loss=0.1641, val_acc=95.85%

Epoch 5: train_loss=0.1478, train_acc=95.42%, val_loss=0.1392, val_acc=96.48%

Epoch 6: train_loss=0.1394, train_acc=95.57%, val_loss=0.1398, val_acc=96.60%

Epoch 7: train_loss=0.1305, train_acc=95.90%, val_loss=0.1299, val_acc=96.55%

Epoch 8: train_loss=0.1238, train_acc=96.05%, val_loss=0.1351, val_acc=96.80%

Epoch 9: train_loss=0.1159, train_acc=96.30%, val_loss=0.1404, val_acc=96.75%

Epoch 10: train_loss=0.1163, train_acc=96.28%, val_loss=0.1313, val_acc=96.82%
```

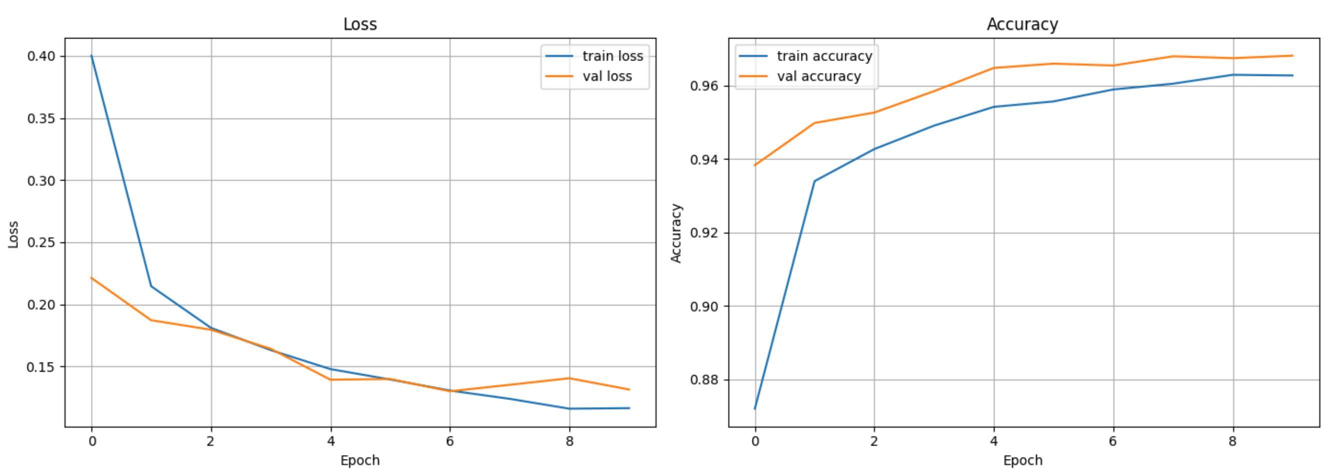

Рисунок 5. Потери и точности для модели 5

```
Test accuracy: 0.9696
```

Model 6

Linear(784, 512)

relu

dropout(p=0.2)

Linear(512, 256)

relu

Linear(256, 10)

softmax

epochs: 15, lr=0.001

```
оценка модели:
Epoch 1: train_loss=0.3692, train_acc=88.34%, val_loss=0.2043, val_acc=94.28%
Epoch 2: train_loss=0.1904, train_acc=94.05%, val_loss=0.1541, val_acc=95.40%
Epoch 3: train_loss=0.1593, train_acc=95.09%, val_loss=0.1559, val_acc=95.82%
Epoch 4: train_loss=0.1405, train_acc=95.56%, val_loss=0.1506, val_acc=95.78%
Epoch 5: train_loss=0.1296, train_acc=96.00%, val_loss=0.1418, val_acc=96.20%
Epoch 6: train_loss=0.1174, train_acc=96.26%, val_loss=0.1290, val_acc=96.53%
Epoch 7: train_loss=0.1102, train_acc=96.52%, val_loss=0.1401, val_acc=96.75%
Epoch 8: train_loss=0.1072, train_acc=96.61%, val_loss=0.1226, val_acc=96.78%
Epoch 9: train_loss=0.0975, train_acc=96.91%, val_loss=0.1168, val_acc=97.07%
```

```
Epoch 10: train_loss=0.0955, train_acc=96.87%, val_loss=0.1076, val_acc=97.40%
Epoch 11: train_loss=0.0911, train_acc=97.05%, val_loss=0.1130, val_acc=97.00%
Epoch 12: train_loss=0.0847, train_acc=97.25%, val_loss=0.1052, val_acc=97.35%
Epoch 13: train_loss=0.0827, train_acc=97.27%, val_loss=0.1201, val_acc=97.17%
Epoch 14: train_loss=0.0846, train_acc=97.23%, val_loss=0.1147, val_acc=97.28%
Epoch 15: train_loss=0.0784, train_acc=97.50%, val_loss=0.1102, val_acc=97.00%
```

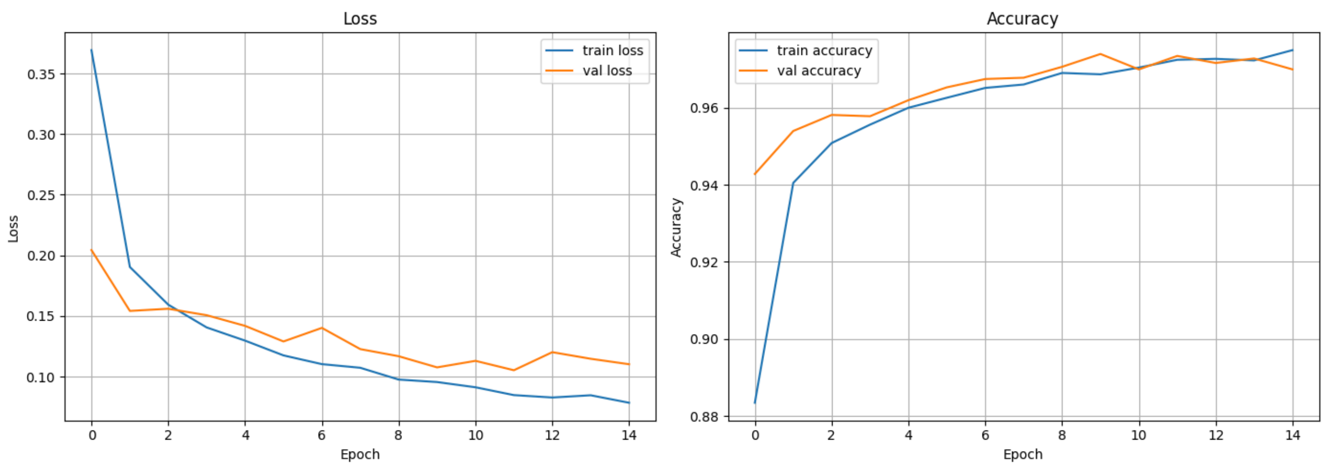

Рисунок 6. Потери и точности для модели 6

```
Test accuracy: 0.9749
```

Model 7

Linear(784, 512)

relu

dropout(p=0.2)

Linear(512, 256)

relu

Linear(256, 10)

softmax

epochs: 30, lr=0.001

оценка модели:

```
Epoch 1: train_loss=0.3599, train_acc=88.46%, val_loss=0.1891, val_acc=94.40%
Epoch 2: train_loss=0.1850, train_acc=94.38%, val_loss=0.1757, val_acc=95.17%
Epoch 3: train_loss=0.1480, train_acc=95.33%, val_loss=0.1434, val_acc=95.95%
Epoch 4: train_loss=0.1319, train_acc=95.87%, val_loss=0.1594, val_acc=95.92%
Epoch 5: train_loss=0.1222, train_acc=96.15%, val_loss=0.1254, val_acc=96.90%
Epoch 6: train_loss=0.1086, train_acc=96.51%, val_loss=0.1319, val_acc=96.57%
Epoch 7: train_loss=0.0994, train_acc=96.80%, val_loss=0.1151, val_acc=97.05%
Epoch 8: train_loss=0.0943, train_acc=96.97%, val_loss=0.1249, val_acc=96.87%
Epoch 9: train_loss=0.0871, train_acc=97.14%, val_loss=0.1168, val_acc=97.20%
Epoch 10: train_loss=0.0839, train_acc=97.30%, val_loss=0.1317, val_acc=96.87%
Epoch 11: train_loss=0.0790, train_acc=97.44%, val_loss=0.1092, val_acc=97.50%
```

Epoch 12: train_loss=0.0767, train_acc=97.43%, val_loss=0.1283, val_acc=97.23%

Epoch 13: train_loss=0.0711, train_acc=97.67%, val_loss=0.1117, val_acc=97.52%

Epoch 14: train_loss=0.0714, train_acc=97.67%, val_loss=0.1288, val_acc=97.10%

Epoch 15: train_loss=0.0698, train_acc=97.69%, val_loss=0.1057, val_acc=97.73%

Epoch 16: train_loss=0.0663, train_acc=97.84%, val_loss=0.1255, val_acc=97.20%

Epoch 17: train_loss=0.0637, train_acc=97.88%, val_loss=0.1130, val_acc=97.62%

Epoch 18: train_loss=0.0609, train_acc=97.96%, val_loss=0.1078, val_acc=97.52%

Epoch 19: train_loss=0.0613, train_acc=97.91%, val_loss=0.1172, val_acc=97.52%

Epoch 20: train_loss=0.0589, train_acc=98.05%, val_loss=0.1155, val_acc=97.47%

Epoch 21: train_loss=0.0612, train_acc=97.92%, val_loss=0.1166, val_acc=97.57%

Epoch 22: train_loss=0.0538, train_acc=98.20%, val_loss=0.1297, val_acc=97.20%

Epoch 23: train_loss=0.0559, train_acc=98.17%, val_loss=0.1111, val_acc=97.55%

Epoch 24: train_loss=0.0552, train_acc=98.16%, val_loss=0.1144, val_acc=97.43%

Epoch 25: train_loss=0.0549, train_acc=98.19%, val_loss=0.1193, val_acc=97.45%

Epoch 26: train_loss=0.0519, train_acc=98.29%, val_loss=0.1139, val_acc=97.47%

Epoch 27: train_loss=0.0509, train_acc=98.31%, val_loss=0.1146, val_acc=97.57%

```
Epoch 28: train_loss=0.0501, train_acc=98.36%, val_loss=0.1217, val_acc=97.50%
Epoch 29: train_loss=0.0515, train_acc=98.28%, val_loss=0.1237, val_acc=97.38%
Epoch 30: train_loss=0.0490, train_acc=98.37%, val_loss=0.1445, val_acc=97.07%
```

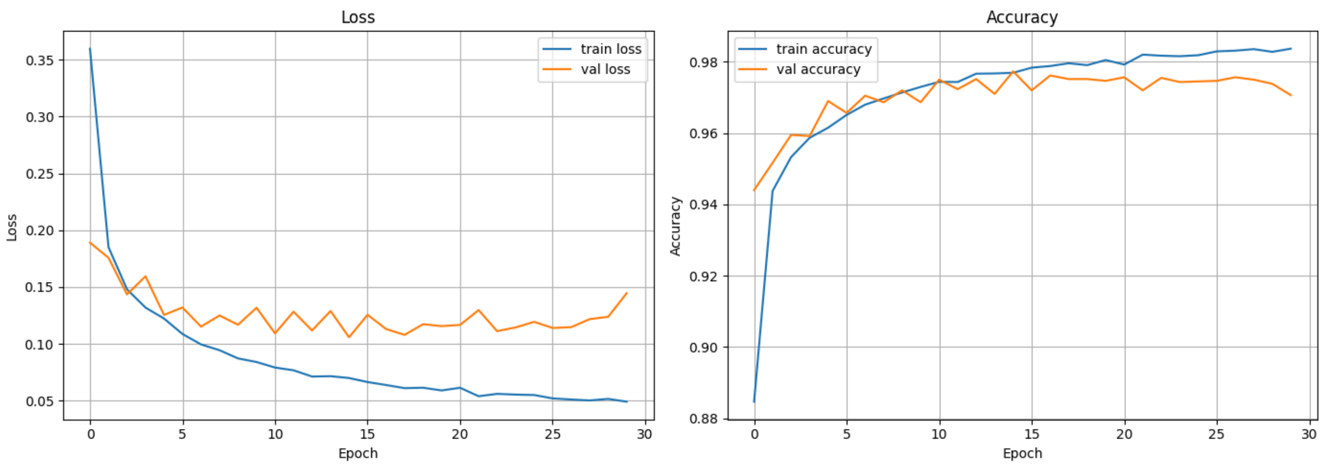

Рисунок 7. Потери и точности для модели 7

```
Test accuracy: 0.9754
```

Model 8

Linear(784, 256)

relu

dropout(p=0.2)

Linear(256, 128)

relu

Linear(128, 10)

softmax

epochs: 15, lr=0.001

оценка модели:

```
Epoch 1: train_loss=0.3823, train_acc=87.75%, val_loss=0.2166, val_acc=93.60%
Epoch 2: train_loss=0.2063, train_acc=93.62%, val_loss=0.1815, val_acc=94.87%
Epoch 3: train_loss=0.1722, train_acc=94.60%, val_loss=0.1521, val_acc=95.75%
Epoch 4: train_loss=0.1521, train_acc=95.21%, val_loss=0.1547, val_acc=95.85%
Epoch 5: train_loss=0.1414, train_acc=95.56%, val_loss=0.1289, val_acc=96.40%
Epoch 6: train_loss=0.1305, train_acc=95.89%, val_loss=0.1444, val_acc=95.72%
Epoch 7: train_loss=0.1209, train_acc=96.13%, val_loss=0.1248, val_acc=96.60%
Epoch 8: train_loss=0.1144, train_acc=96.41%, val_loss=0.1193, val_acc=96.83%
Epoch 9: train_loss=0.1084, train_acc=96.51%, val_loss=0.1279, val_acc=96.93%
Epoch 10: train_loss=0.1029, train_acc=96.70%, val_loss=0.1299, val_acc=96.67%
Epoch 11: train_loss=0.0947, train_acc=96.90%, val_loss=0.1329, val_acc=96.73%
Epoch 12: train_loss=0.0961, train_acc=96.88%, val_loss=0.1152, val_acc=97.20%
```

Epoch 13: train_loss=0.0939, train_acc=96.99%, val_loss=0.1257, val_acc=96.73%

Epoch 14: train_loss=0.0880, train_acc=97.16%, val_loss=0.1196, val_acc=97.22%

Epoch 15: train_loss=0.0853, train_acc=97.20%, val_loss=0.1238, val_acc=97.08%

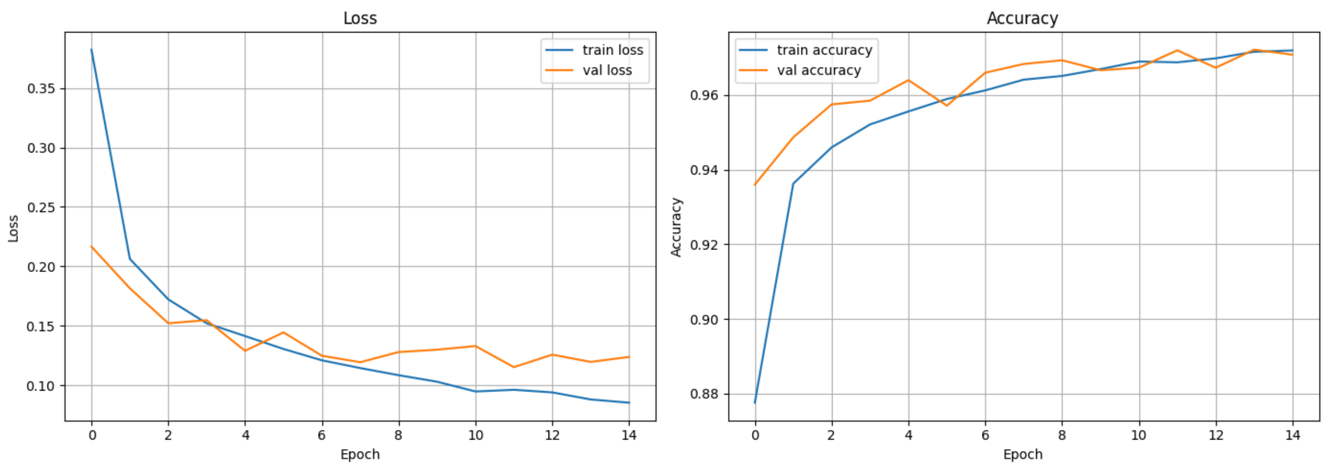

Рисунок 8. Потери и точности для модели 8

Test accuracy: 0.9719

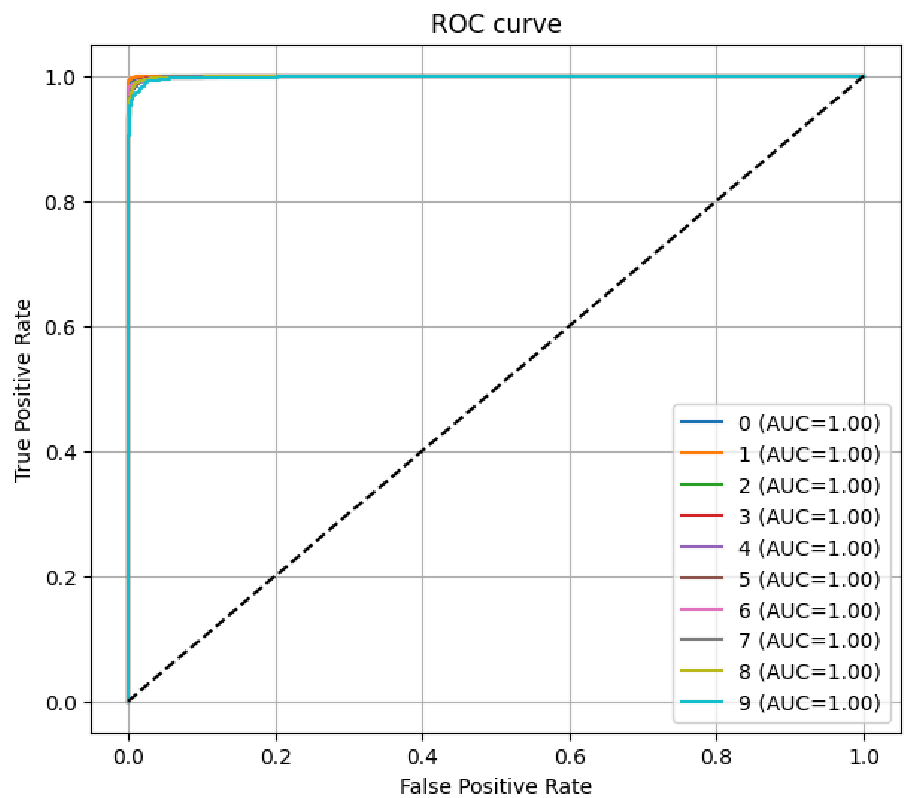

Рисунок 9. ROC-кривая для 10 классов выборки

Вывод

В процессе экспериментов было рассмотрено несколько конфигураций архитектуры и гиперпараметров модели, отличающихся числом скрытых слоёв, размерностью скрытых представлений, использованием регуляризации и количеством эпох обучения. По результатам сравнительного анализа лучшей оказалась модель Model 8 со следующей архитектурой:

- входной слой: Linear(784, 256),
- функция активации ReLU,
- слой регуляризации Dropout с вероятностью отключения нейронов p = 0.2,
- скрытый слой Linear(256, 128) с функцией активации ReLU,
- выходной слой Linear(128, 10) с функцией Softmax.

Обучение модели проводилось в течение 15 эпох с использованием оптимизатора Adam и шагом обучения lr = 0.001

В результате обучения было достигнуто:

- точность на обучающей выборке: 97.20%
- точность на валидационной выборке: 97.22%
- точность классификации на тестовой выборке: 97.19%

Использование слоя Dropout позволило снизить эффект переобучения и обеспечить более стабильное поведение модели на валидационных и тестовых данных по сравнению с моделями без регуляризации. При увеличении количества эпох без Dropout наблюдалось переобучение, выражающееся в росте ошибки на валидационной выборке.

Проведённые эксперименты показали, что увеличение сложности модели сверх двух скрытых слоёв не приводит к существенному росту качества, но повышает риск переобучения. Таким образом, выбранная архитектура является компромиссом между точностью, устойчивостью и вычислительной сложностью.

Разработанная модель успешно решает задачу распознавания рукописных цифр и может применяться для классификации изображений