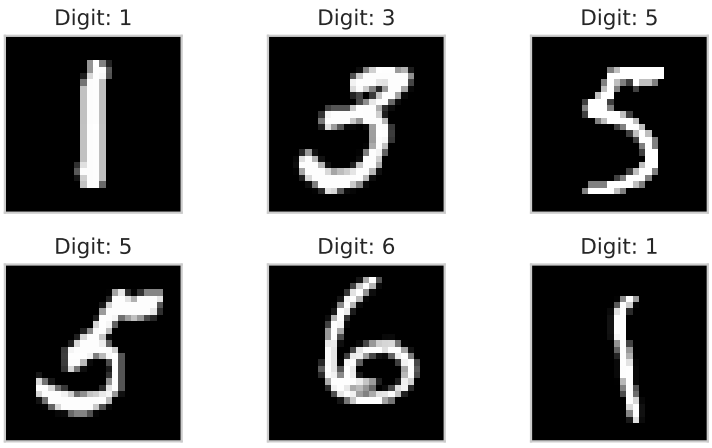
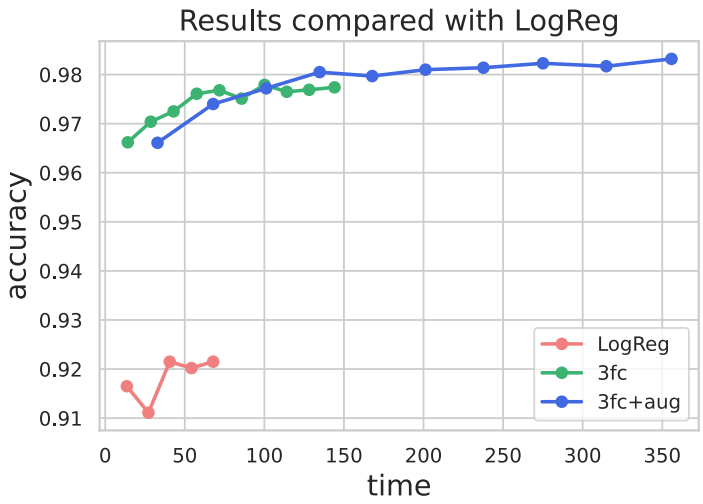
Классификация изображений

MNIST. Multilayer Perceptron



Результаты

Добился accuracy=98.32% на трех полносвязных слоях + аугментация. Сравнение:



Я начал с логистической регрессии и постепенно добавлял разные слои, чтобы увидеть, какой эффект они даёт каждый из них. Ниже хронология этих архитектур.

Все изображения предварительно нормализовал:

```
my_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])
```

Функция потерь во всех экспериментах `torch.nn.functional.cross_entropy`.

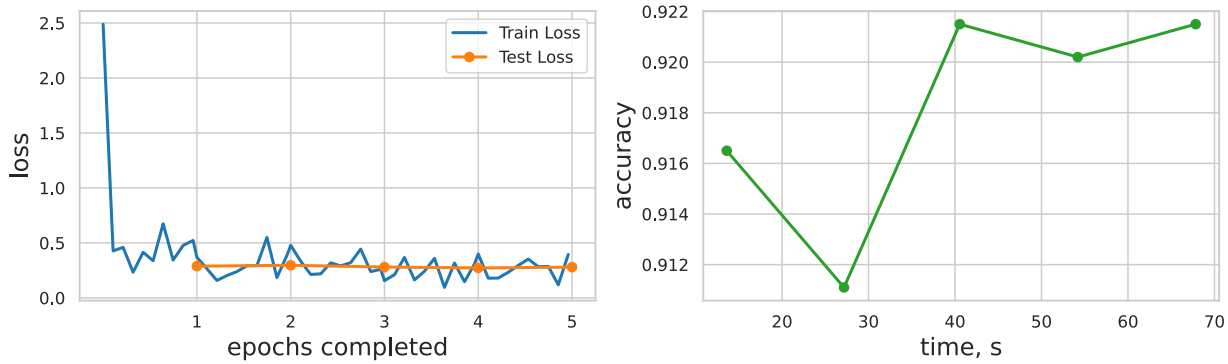
LogReg

Архитектура:

```
# 7840 params
self.layers = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 10)
)
```

Обучение:

Logistic Regression 7K params (max accuracy 92.15%, time 68 s)

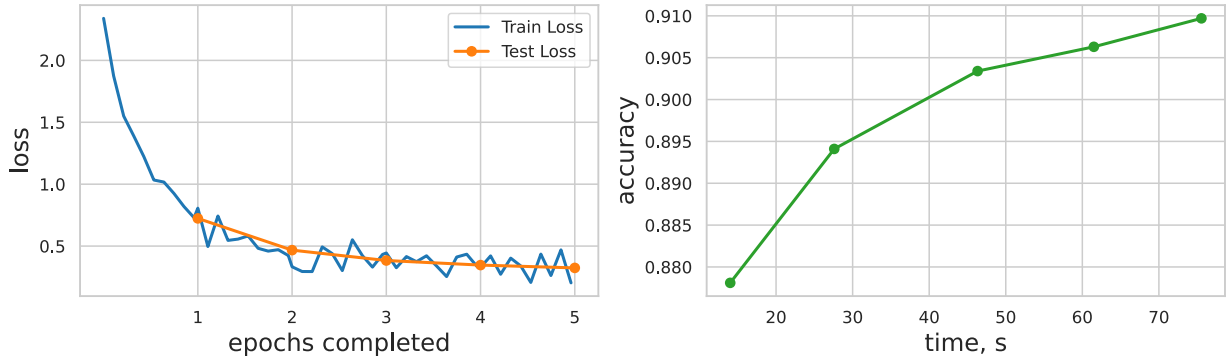


2 fc layers + sigmoid, 8K params

```
# 7940 parameters
self.layers = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 10),
    nn.Sigmoid(),
    nn.Linear(10, 10)
)
```

Обучение:

2 fc layers + sigmoid, 8K params (accuracy 90.97%, time 76 s)

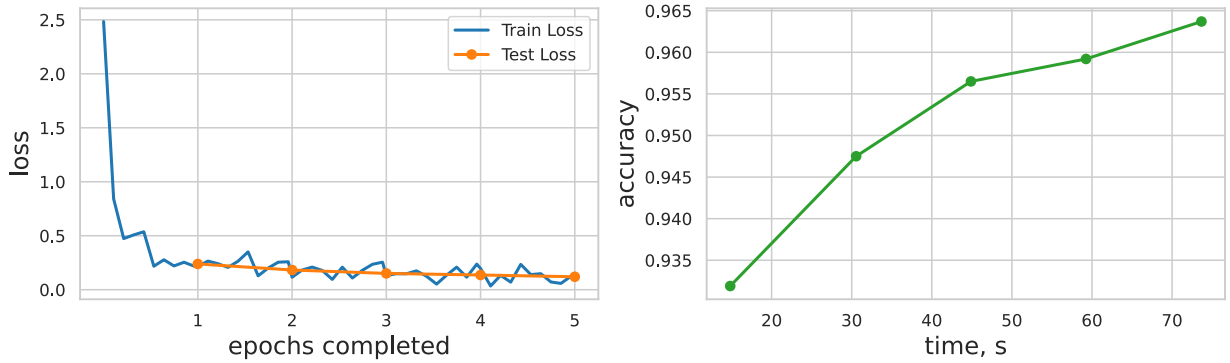


2 fc layers + sigmoid, 55K params

```
# 55580 parameters
self.layers = nn.Sequential(
    nn.Flatten(),
    nn.Linear(787, 70),
    nn.Sigmoid(),
    nn.Linear(70, 10),
)
```

Обучение:

2 fc layers + sigmoid, 55K params (max accuracy 96.37%, time 74 s)

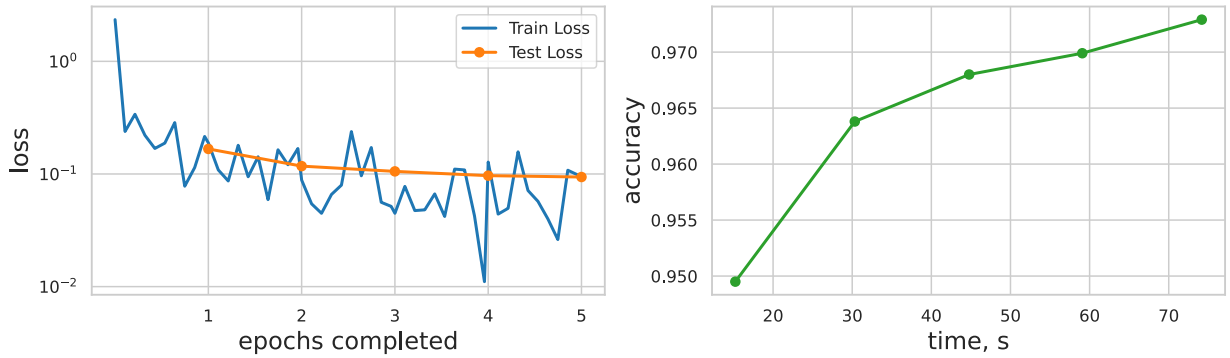


2 fc layers + ReLU, 55K params

```
# 55580 parameters
self.layers = nn.Sequential(
    nn.Flatten(),
    nn.Linear(787, 70),
    nn.ReLU(),
    nn.Linear(70, 10),
)
```

Обучение:

2 fc layers + ReLU, 55K params (max accuracy 97.29%, time 74 s)

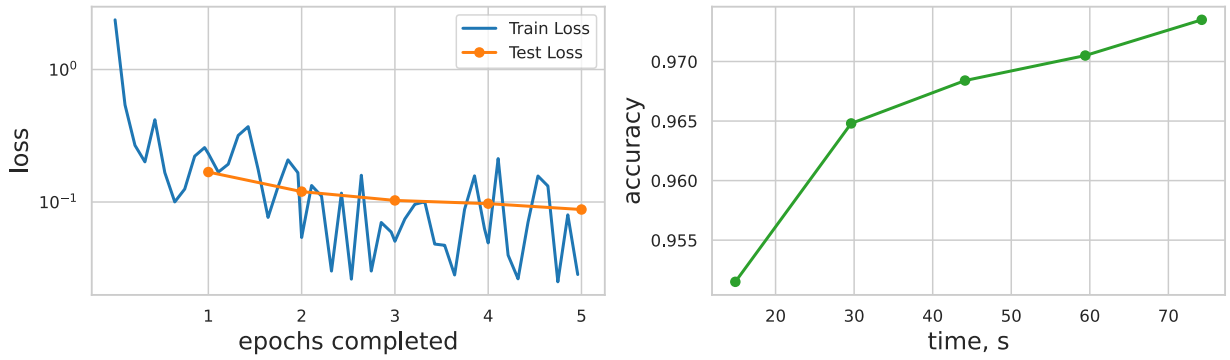


2 fc layers + ReLU + batch-norm, 55K params

```
# 55K parameters
self.layers = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 70),
    nn.BatchNorm1d(70),
    nn.ReLU(),
    nn.Linear(70,10)
)
```

Обучение:

2 fc layers + ReLU + batch-norm, 55K params (max accuracy 97.35%, time 74 s)

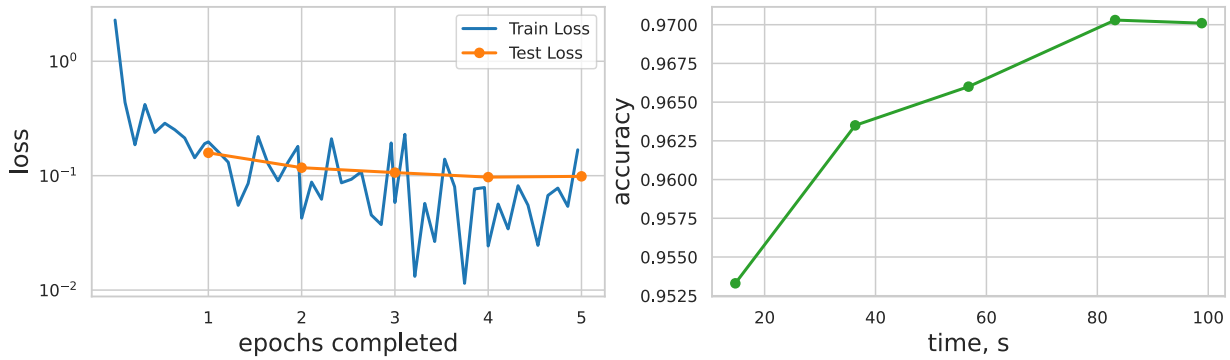


3 fc layers, 51K params

```
# 51K params
self.layers = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 60),
    nn.ReLU(),
    nn.Linear(60, 60),
    nn.ReLU(),
    nn.Linear(60, 10)
)
```

Обучение:

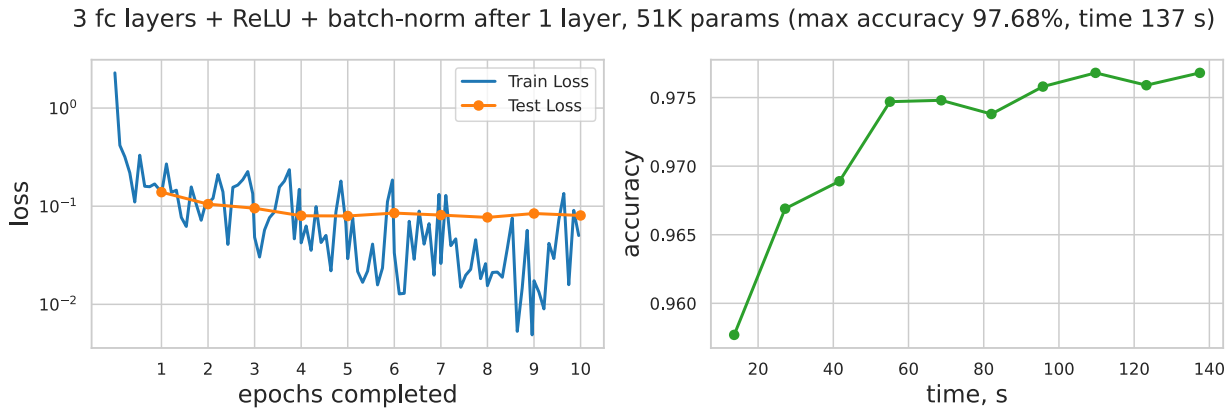
3 fc layers + ReLU, 51K params (max accuracy 97.03%, time 99 s)



3 fc layers + batch-norm after 1 layer, 51K params

```
# 51240 parameters
self.layers = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 60),
    nn.BatchNorm1d(60),
    nn.ReLU(),
    nn.Linear(60, 60),
    nn.ReLU(),
    nn.Linear(60, 10)
)
```

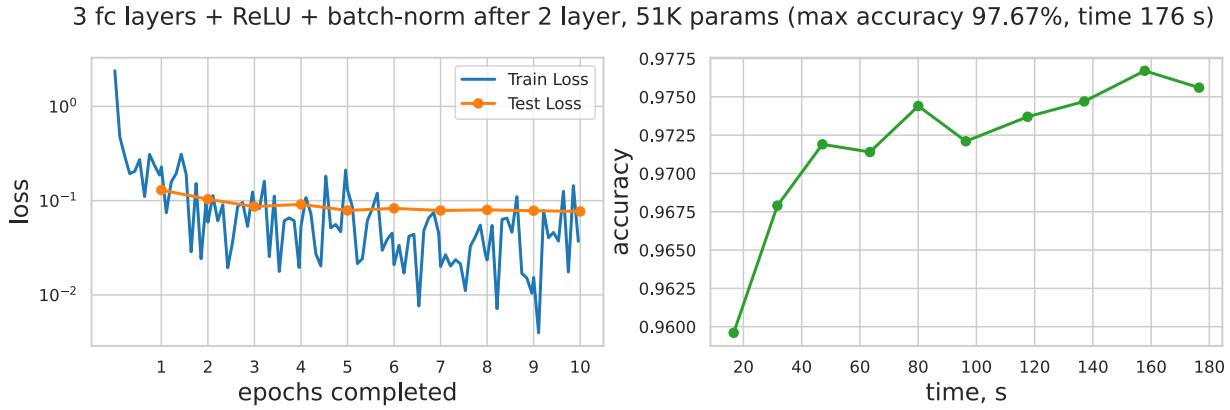
Обучение:



3 fc layers + batch-norm after 2 layer, 51K params

```
# 51240 parameters
self.layers = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 60),
    nn.ReLU(),
    nn.Linear(60, 60),
    nn.BatchNorm1d(60),
    nn.ReLU(),
    nn.Linear(60, 10)
)
```

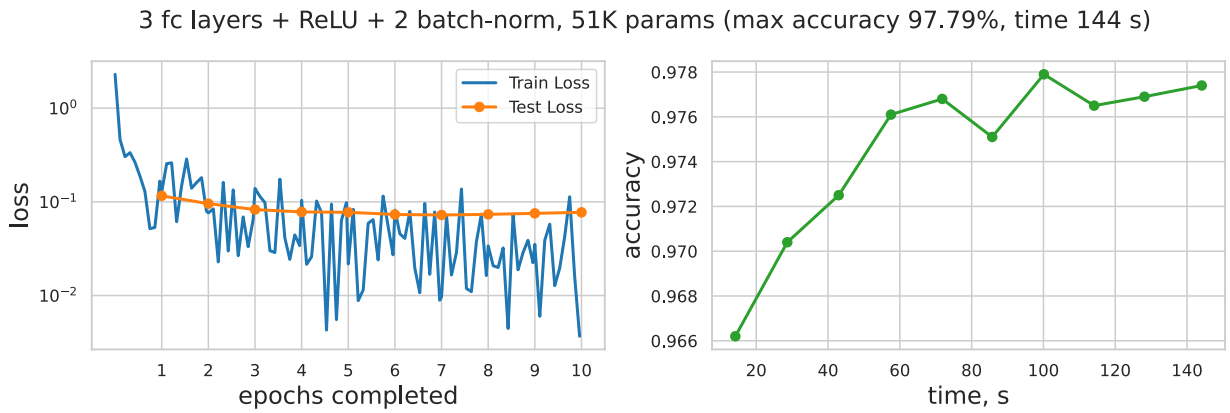
Обучение:



3 fc layers + 2 batch-norm, 51K params

```
# 51K parameters
self.layers = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 60),
    nn.BatchNorm1d(60),
    nn.ReLU(),
    nn.Linear(60, 60),
    nn.BatchNorm1d(60),
    nn.ReLU(),
    nn.Linear(60, 10)
)
```

Обучение:

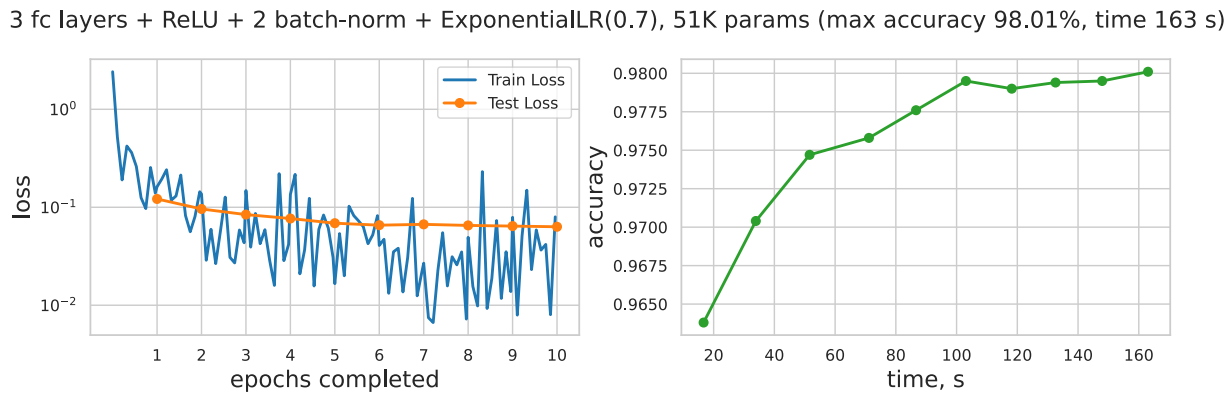


3 fc layers + 2 batch-norm + ExponentialLR(0.7), 51K params

Используется scheduler:

```
sceduler = optim.lr_scheduler.ExponentialLR(wrapped_opt, 0.7, verbose=True)
```

Обучение:

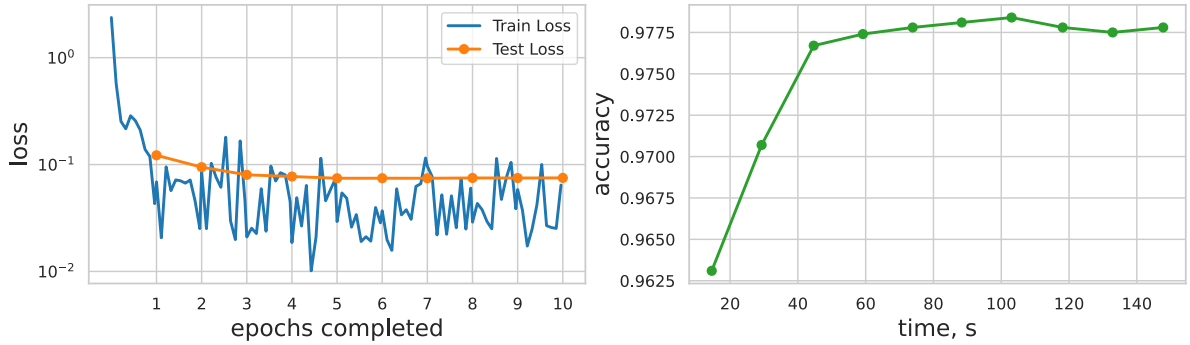


3 fc layers + 2 batch-norm + PolynomialLR(2), 51K params

```
sceduler = optim.lr_scheduler.PolynomialLR(wrapped_opt, power=2, verbose=True)
```

Обучение:

3 fc layers + ReLU + 2 batch-norm + PolynomialLR(2), 51K params (max accuracy 97.84%, time 148 s)



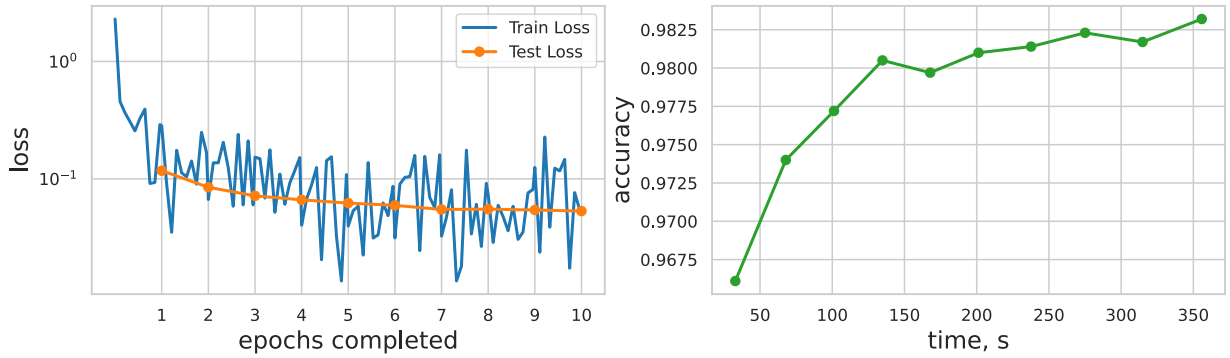
Augmentation

Аугментацию применял только к лучшей архитектуре, потому что она увеличивает время обучения. Архитектура `3 fc layers + 2 batch-norm + ExponentialLR(0.7), 51K params` была обучена с трансформациями обучающей выборки:

```
mnist_aug_transform = transforms.Compose([
    transforms.RandomChoice([
        transforms.GaussianBlur(kernel_size=5,sigma=0.3),
        transforms.RandomRotation(degrees=10),
        transforms.RandomCrop(size=28, padding=1, padding_mode='reflect'),
    ]),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])
```

Обучение:

My best + augmentation (max accuracy 98.32%, time 356 s)



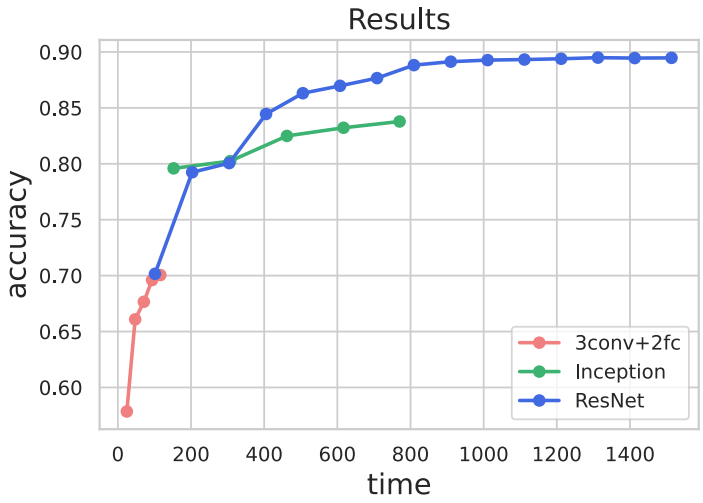
Это лучший полученый результат.

CIFAR-10. Convolutional Neural Networks

Результаты

Начинал с одного свёрточного и одного полносвязного слоев, затем добавлял разные слои, чтобы увидеть какой вклад они вносят. Так достиг accuracy=70.00%.

Затем последовательно прошёлся по архитектурам VGG, Inception, ResNet. Лучшее accuracy=89.87% у ResNet. Сравнение:



Все изображения предварительно нормализовал:

```
my_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])
```

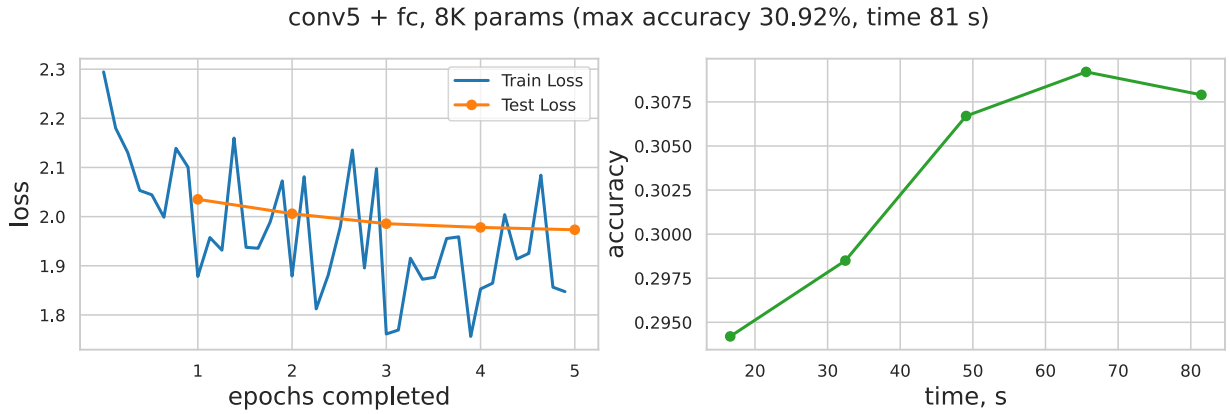
Функция потерь во всех экспериментах `torch.nn.functional.cross_entropy`.

Ниже хронология архитектур.

conv5 + fc, 8K params

```
self.layers = nn.Sequential(
    nn.Conv2d(3, 1, kernel_size=5), # 32x32 -> 28x28
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(784, 10),
)
```

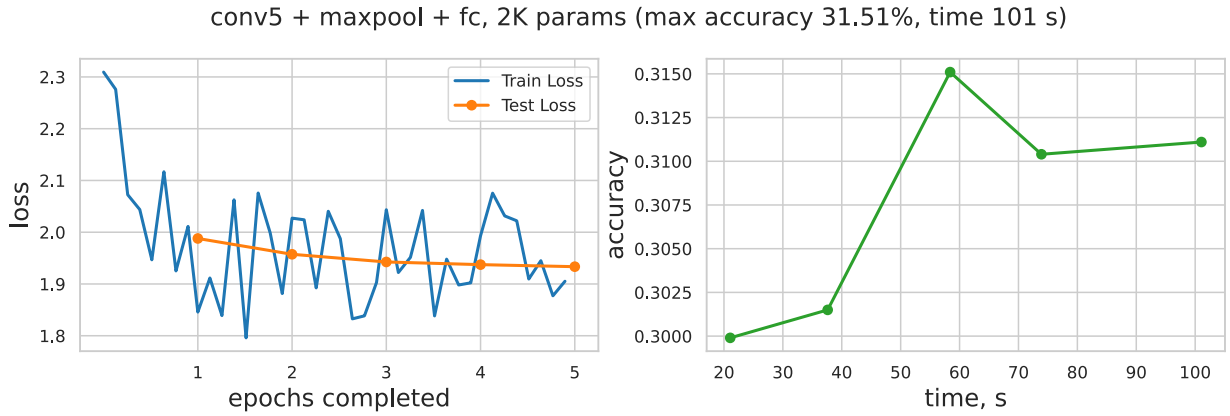
Обучение:



conv5 + maxpool2 + 1 fc, 2K params

```
self.layers = nn.Sequential(
    nn.Conv2d(3, 1, kernel_size=5), # 32x32 -> 28x28
    nn.MaxPool2d(kernel_size=2),    # 28x28 -> 14x14
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(196, 10),
)
```

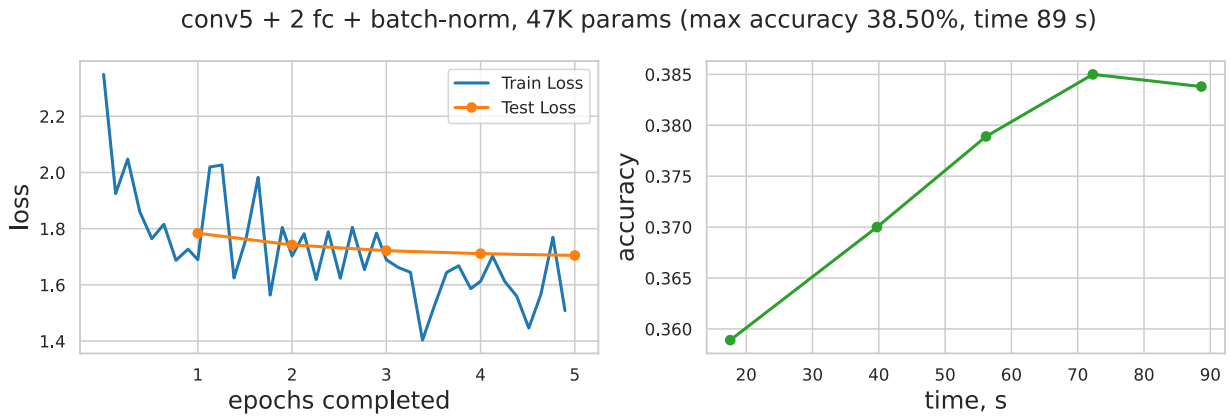
Обучение:



conv5 + 2 fc + batch-norm, 47K params

```
self.layers = nn.Sequential(
    nn.Conv2d(3, 1, kernel_size=5), # 32x32 -> 28x28
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(784, 60),
    nn.BatchNorm1d(60),
    nn.ReLU(),
    nn.Linear(60, 10)
)
```

Обучение:

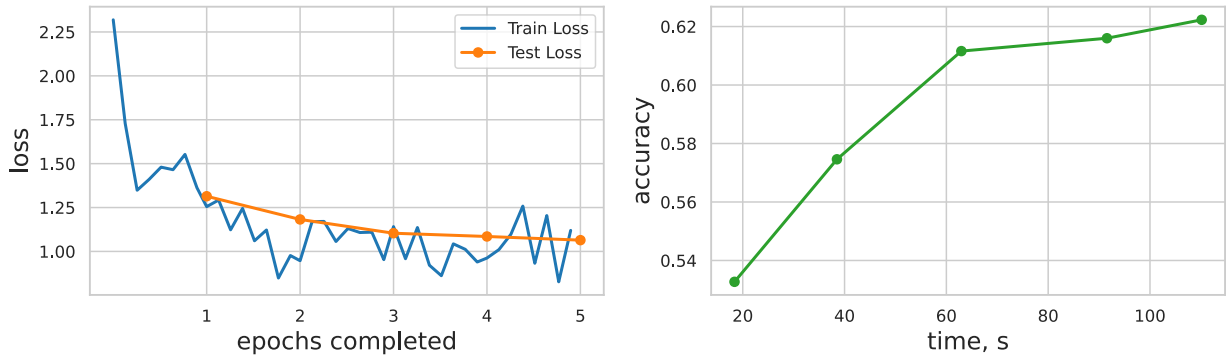


2 conv5 + 2 fc + batch-norm, 21K params

```
self.layers = nn.Sequential(
    nn.Conv2d(3, 6, kernel_size=5),     # 6x28x28
    nn.MaxPool2d(kernel_size=2),        # 6x14x14
    nn.ReLU(),

    nn.Conv2d(6, 12, kernel_size=5),    # 12x10x10
    nn.MaxPool2d(kernel_size=2),        # 12x5x5
    nn.ReLU(),

    nn.Flatten(),                       # 300

    nn.Linear(300, 60),
    nn.BatchNorm1d(60),
    nn.ReLU(),

    nn.Linear(60, 10)
)
```

Обучение:

2 conv5 + 2 fc + batch-norm, 21K params (max accuracy 62.23%, time 110 s)

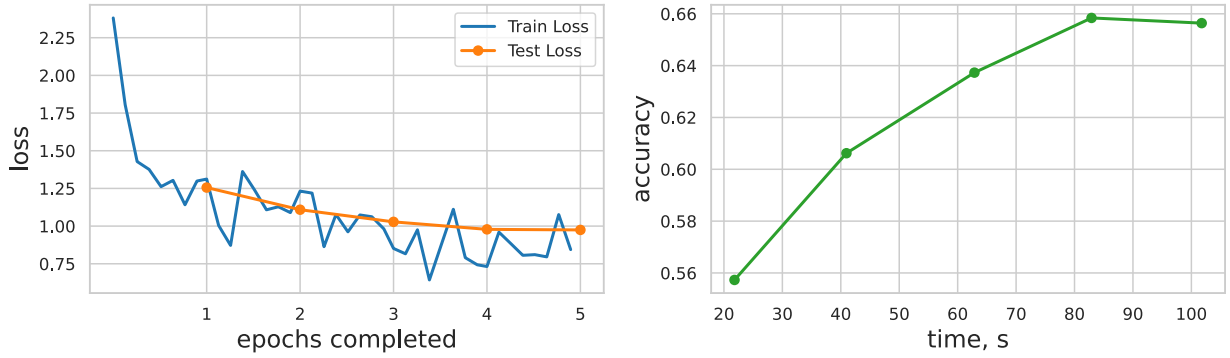


2 conv5 + 2 fc + batch-norm, 36K params

```
self.layers = nn.Sequential(
    nn.Conv2d(3, 10, kernel_size=5),    # 10x28x28
    nn.MaxPool2d(kernel_size=2),        # 10x14x14
    nn.ReLU(),

    nn.Conv2d(10, 20, kernel_size=5),   # 20x10x10
    nn.MaxPool2d(kernel_size=2),        # 20x5x5
    nn.ReLU(),
    nn.Flatten(),                       # 500

    nn.Linear(500, 60),
    nn.BatchNorm1d(60),
    nn.ReLU(),

    nn.Linear(60, 10)
)
```

Обучение:

2 conv5 + 2 fc + batch-norm, 36K params (max accuracy 65.84%, time 102 s)

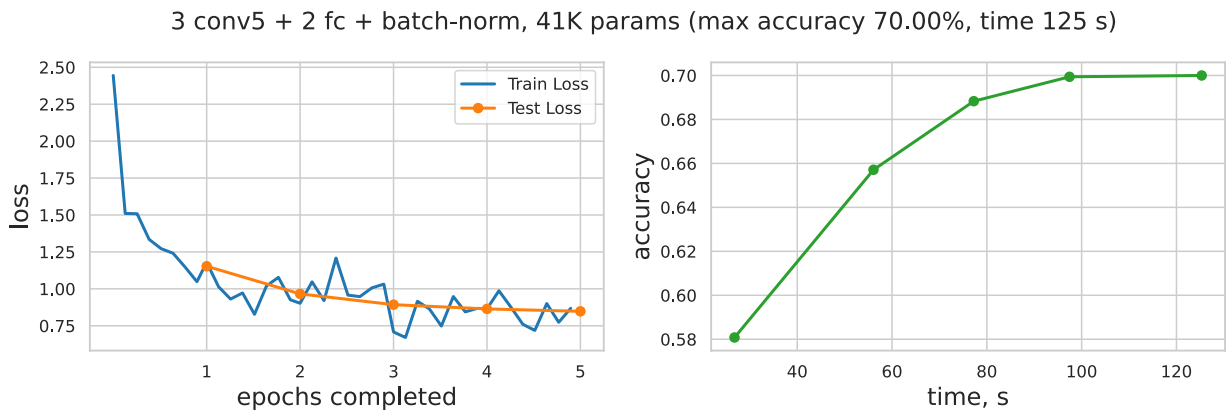


3 conv + 2 fc + batch-norm, 41K params

```
self.layers = nn.Sequential(
    nn.Conv2d(3, 30, kernel_size=5),    # 30x28x28
    nn.MaxPool2d(kernel_size=2),        # 30x14x14
    nn.ReLU(),
```

```
    nn.Conv2d(30, 30, kernel_size=3),    # 30x12x12
    nn.MaxPool2d(kernel_size=2),         # 30x6x6
    nn.ReLU(),

    nn.Conv2d(30, 60, kernel_size=3),    # 60x4x4
    nn.MaxPool2d(kernel_size=2),         # 60x2x2
    nn.ReLU(),

    nn.Flatten(),                        # 240

    nn.Linear(240, 60),
    nn.BatchNorm1d(60),
    nn.ReLU(),

    nn.Linear(60, 10)
)
```

Обучение:



VGG: 6 conv + 2 fc, 179K params

Вспомогательный блок:

```
class ConvBlock(nn.Module):
  def __init__(self, in_channels, out_channels, **kwargs):
    super(ConvBlock, self).__init__()

    self.layers = nn.Sequential(
      nn.Conv2d(in_channels, out_channels, **kwargs),
      nn.BatchNorm2d(out_channels),
      nn.ReLU())

  def forward(self, x):
    return self.layers(x)
```

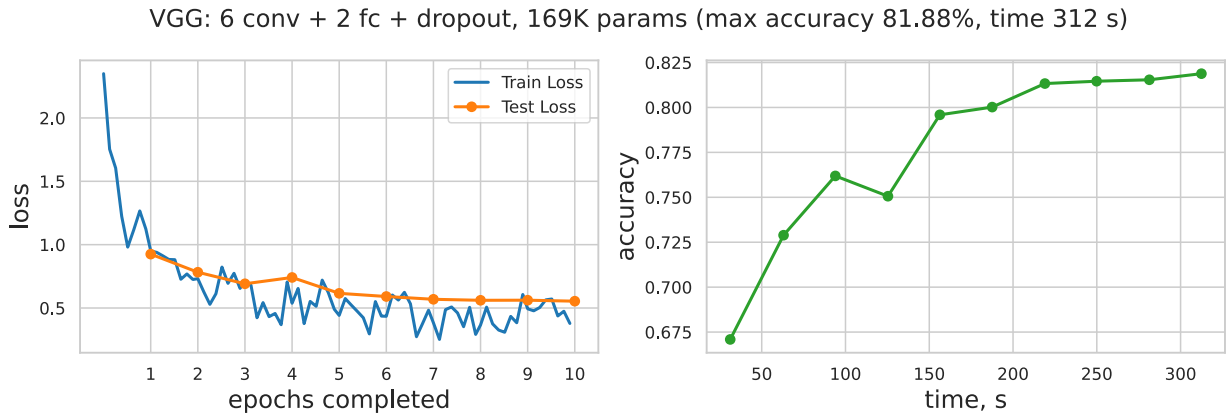
Архитектура:

```
self.layers = nn.Sequential(
```

```
    ConvBlock(3, 15, kernel_size=3, padding=1),     # 15x32x32
    ConvBlock(15, 30, kernel_size=3, padding=1),    # 30x32x32
    nn.MaxPool2d(kernel_size=2),                    # 30x16x16
    ConvBlock(30, 45, kernel_size=3, padding=1),    # 45x16x16
    nn.Dropout(p=0.2),
    ConvBlock(45, 60, kernel_size=3, padding=1),    # 60x16x16
    nn.MaxPool2d(kernel_size=2),                    # 60x8x8
    ConvBlock(60, 75, kernel_size=3),               # 75x6x6
    nn.Dropout(p=0.2),
    ConvBlock(75, 90, kernel_size=3),               # 90x4x4
    nn.MaxPool2d(kernel_size=2),                    # 90x2x2

    nn.Flatten(),                                   # 360
    nn.Linear(360, 70),
    nn.BatchNorm1d(70),
    nn.ReLU(),
    nn.Dropout(p=0.2),
    nn.Linear(70, 10)
)
```

нужно ли ставить дропаут после ReLU? ведь ReLU как раз недифф в нуле

Обучение:



4 inception blocks, 514K params + two-step augmentation

Аугментация:

```
aug_transform = transforms.Compose([
    transforms.AutoAugment(transforms.AutoAugmentPolicy.CIFAR10),
    transforms.ToTensor(),
    transforms.Normalize((0.3945,), (0.2896,))])
```

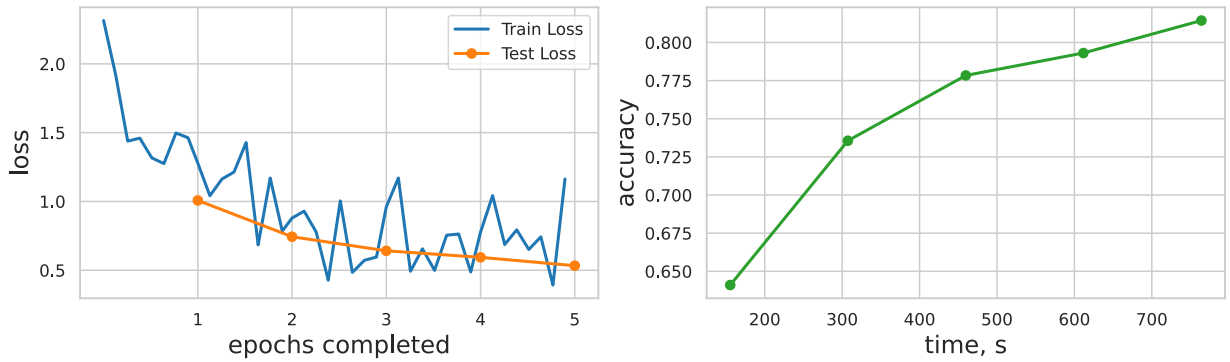
Архитектура:

```
self.layers = nn.Sequential(
    ConvBlock(3, 30, kernel_size=3, padding=1),     # 30x32x32

    ConvBlock(30, 60, kernel_size=3, padding=1),    # 60x32x32
```

```
    nn.MaxPool2d(kernel_size=2),                    # 60x16x16

    nn.Dropout(p=0.2),

    InceptionBlock(in_channels=60, out_1x1=20,
                   red_3x3=30, out_3x3=40,
                   red_5x5=10, out_5x5=20,
                   out_pool=10),                     # 90x16x16

    nn.MaxPool2d(kernel_size=2),                    # 90x8x8

    InceptionBlock(in_channels=90, out_1x1=30,
                   red_3x3=35, out_3x3=60,
                   red_5x5=20, out_5x5=30,
                   out_pool=20),                    # 140x8x8

    nn.Dropout(p=0.2),

    InceptionBlock(in_channels=140, out_1x1=60,
                   red_3x3=50, out_3x3=100,
                   red_5x5=50, out_5x5=50,
                   out_pool=30),                     # 240x8x8

    nn.MaxPool2d(kernel_size=2),                     # 240x4x4

    InceptionBlock(in_channels=240, out_1x1=70,
                   red_3x3=50, out_3x3=140,
                   red_5x5=50, out_5x5=70,
                   out_pool=50),                     # 330x4x4

    nn.MaxPool2d(kernel_size=2),                    # 330x2x2
    nn.Dropout(p=0.2),
    nn.Flatten(),                                   # 1320

    nn.Linear(1320, 70),
    nn.BatchNorm1d(70),
    nn.ReLU(),

    nn.Dropout(p=0.2),

    nn.Linear(70, 10)
)
```

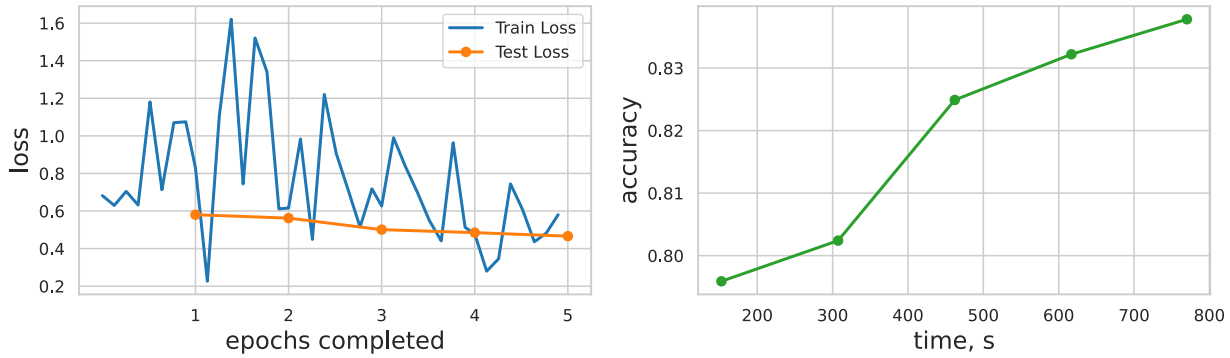
Первые пять эпох без агументации:

4 inception blocks first step, 514K params (max accuracy 81.44%, time 764 s)



Следующие пять эпох на аугментированной выборке:

4 inception blocks second step, 514K params (max accuracy 83.78%, time 770 s)

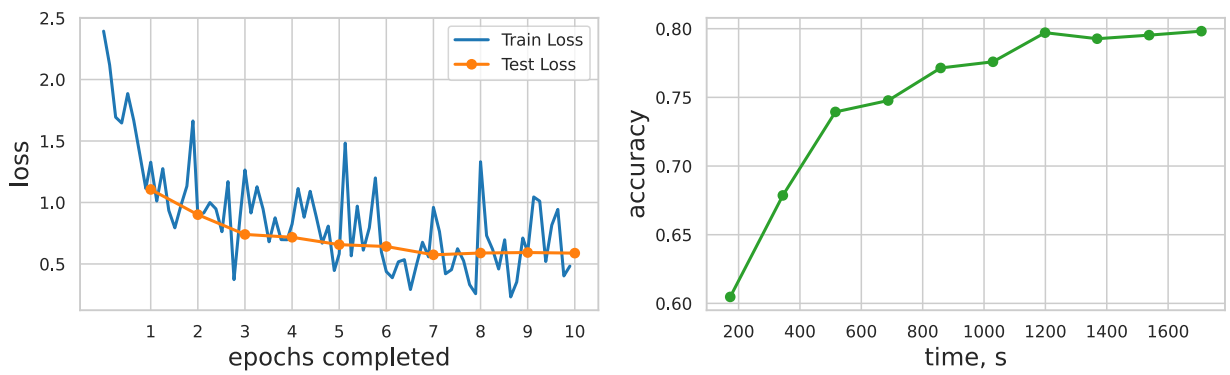


4 inception blocks, 514K params + one-step augmentation

```
# 7940 parameters
self.layers = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 10),
    nn.Sigmoid(),
    nn.Linear(10, 10)
)
```

Обучение:

InceptionV1, 1.2M params (max accuracy 79.82%, time 1709 s)



ResNet, 6.6M params

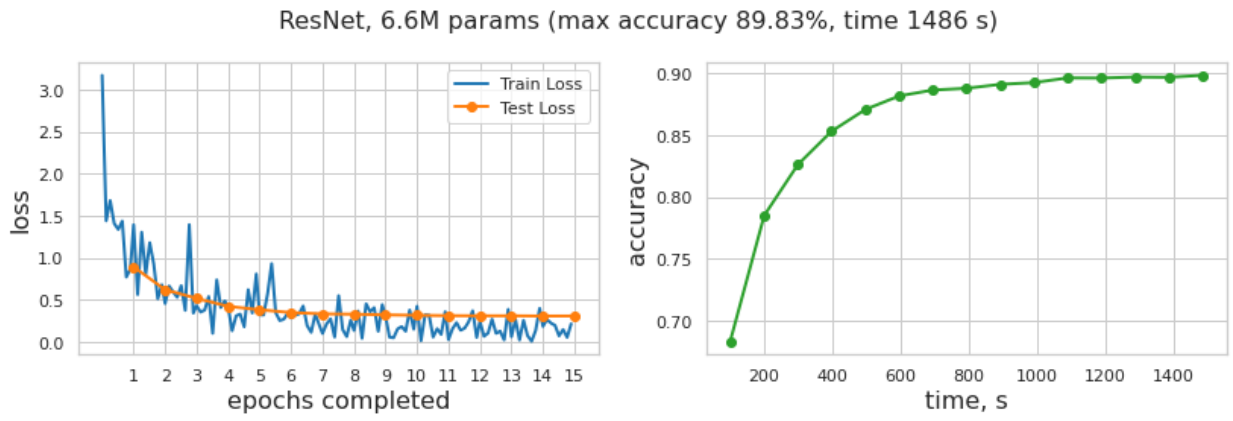
Вспомогательный блок:

```
class ResBlock(nn.Module):
  def __init__(self, in_channels):
    super(ResBlock, self).__init__()

    self.layers = nn.Sequential(
      nn.Conv2d(in_channels, in_channels, kernel_size=3, padding=1),
      nn.BatchNorm2d(in_channels),
      nn.ReLU(),
      nn.Conv2d(in_channels, in_channels, kernel_size=3, padding=1),
      nn.BatchNorm2d(in_channels))

  def forward(self, x):
    return F.relu(self.layers(x) + x)
```

Архитектура:

```
self.layers = nn.Sequential(
    ConvBlock(3, 64, kernel_size=3, padding=1),
    ConvBlock(64, 128, kernel_size=3, padding=1),
    nn.MaxPool2d(kernel_size=2),
    ResBlock(128),
    ConvBlock(128, 256, kernel_size=3, padding=1),
    nn.MaxPool2d(kernel_size=2),
    ConvBlock(256, 512, kernel_size=3, padding=1),
    nn.MaxPool2d(kernel_size=2),
    ResBlock(512),
    nn.MaxPool2d(kernel_size=4),
    nn.Flatten(),
    nn.Linear(512, 10))
```

Обучение:



Наверное, можно и с меньшим числом параметров...