**Документация по улучшенной модели BetterNet для классификации MNIST**

---

## 🔷Введение

Этот документ — подробное руководство по построению **улучшенной нейросети** на PyTorch для распознавания рукописных цифр из датасета **MNIST**. Мы разберем, как изменить и улучшить начальную модель `SimpleNet`, добавив:

- Больше слоёв (глубина)
- Больше нейронов (ширина)
- Dropout — для защиты от переобучения
- BatchNorm — для стабильности обучения

А также сравним результаты и объясним, зачем нужны эти улучшения.

---

## 🔷Содержание



---

## 1. Загрузка данных

```
transform = transforms.Compose([transforms.ToTensor()])
train_data = datasets.MNIST(root='.', train=True, download=True,
transform=transform)
test_data = datasets.MNIST(root='.', train=False, download=True,
transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64)
```

Мы используем стандартный **датасет MNIST**, содержащий изображения цифр 28×28 пикселей.

`transforms.ToTensor()` переводит изображения в тензоры PyTorch.

---

### 2. Улучшенная модель — BetterNet

```
class BetterNet(nn.Module):
    def __init__(self, activation='relu'):
        super(BetterNet, self).__init__()
        self.activation = activation

        self.fc1 = nn.Linear(28 * 28, 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.dropout1 = nn.Dropout(0.3)

        self.fc2 = nn.Linear(256, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.dropout2 = nn.Dropout(0.3)

        self.fc3 = nn.Linear(128, 64)
        self.bn3 = nn.BatchNorm1d(64)
        self.dropout3 = nn.Dropout(0.3)

        self.out = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = self._apply_activation(self.bn1(self.fc1(x)))
        x = self.dropout1(x)

        x = self._apply_activation(self.bn2(self.fc2(x)))
        x = self.dropout2(x)

        x = self._apply_activation(self.bn3(self.fc3(x)))
        x = self.dropout3(x)

        return self.out(x)

    def _apply_activation(self, x):
        if self.activation == 'relu':
            return F.relu(x)
        elif self.activation == 'sigmoid':
            return torch.sigmoid(x)
        elif self.activation == 'tanh':
            return torch.tanh(x)
        return x
```

**Что изменилось по сравнению с SimpleNet:**

| Компонент | SimpleNet | BetterNet | Зачем это нужно? |
|---|---|---|---|
| Слои | 2 (fc1, fc2) | 4 (fc1 → fc2 → fc3 → out) | Больше слоёв — больше абстракций и выразительности |
| Нейронов | 128, 10 | 256 → 128 → 64 → 10 | Глубже и шире — модель учится сложнее различия |
| Dropout | ❌ | ✅после каждого слоя | Снижает переобучение |
| BatchNorm | ❌ | ✅нормализация перед активацией | Ускоряет и стабилизирует обучение |
| Гибкая активация | Только ReLU | ReLU / Sigmoid / Tanh (на выбор) | Можно экспериментировать |

### 3. Обучение модели

```
def train(model, loader, optimizer, criterion, epochs=100):
    model.train()
    loss_history = []
    accuracy_history = []
    important_epochs = [1, 2, 3, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
    full_log = ""

    for epoch in range(1, epochs + 1):
        total_loss = 0
        correct = 0
        total = 0

        for i, (X, y) in enumerate(loader):
            optimizer.zero_grad()                       # 1. Обнуляем градиенты
            out = model(X)                              # 2. Прямой проход
(forward pass)
            loss = criterion(out, y)                    # 3. Вычисляем ошибку
(loss)
            loss.backward()                             # 4. Вычисляем градиенты
            optimizer.step()                            # 5. Обновляем веса

            total_loss += loss.item()
            predicted = torch.argmax(out, dim=1)
            correct += (predicted == y).sum().item()
            total += y.size(0)

            if i == 0:
```

```
                first_preds = list(zip(y[:10], predicted[:10]))
            elif i == len(loader) - 1:
                last_preds = list(zip(y[:10], predicted[:10]))

        avg_loss = total_loss
        accuracy = correct / total
        loss_history.append(avg_loss)
        accuracy_history.append(accuracy)

        full_log += f"Epoch {epoch} — Loss: {avg_loss:.4f} | Accuracy:
{accuracy * 100:.2f}%\n"

        if epoch in important_epochs:
            full_log += "\nПервые 10 предсказаний:\n"
            for real, pred in first_preds:
                full_log += f"  Правильный = {real.item()} \t Предсказание =
{pred.item()}\n"
            full_log += "\nПоследние 10 предсказаний:\n"
            for real, pred in last_preds:
                full_log += f"  Правильный = {real.item()} \t Предсказание =
{pred.item()}\n"
            full_log += "\n" + "-" * 40 + "\n"

    print(full_log)
    with open("train_log.txt", "w", encoding="utf-8") as f:
        f.write(full_log)

    return loss_history, accuracy_history
```

---

### 6. Как работает обучение: построчный разбор

```
optimizer.zero_grad()       # Обнуляем градиенты, чтобы не накапливались
```

Каждый шаг обучения должен начинаться с обнуления градиентов — иначе они будут накапливаться и портить результат.

```
out = model(X)              # Прямой проход — получаем предсказания модели
```

Модель обрабатывает входные изображения и возвращает **логиты** (не вероятности!).

```
loss = criterion(out, y)    # Считаем ошибку модели
```

Используем CrossEntropyLoss — она сама применит softmax и сравнит с метками.

```
loss.backward()             # Обратное распространение ошибки
```

PyTorch автоматически вычисляет градиенты для каждого параметра модели.

```
optimizer.step()            # Обновляем веса на основе градиентов
```

Обновление параметров — это и есть **шаг обучения**. Здесь модель становится умнее.

Остальное — подсчёт точности и логирование.

---

### 7. Расшифровка цепочек типа `self._apply_activation(self.bn2(self.fc2(x)))`

1. `self.fc2(x)` — линейный слой: матричное умножение входа на веса + смещение (bias).
2. `self.bn2(...)` — нормализует выход: делает обучение стабильнее и быстрее.
3. `self._apply_activation(...)` — применяет функцию активации (ReLU / Sigmoid / Tanh).

Эта цепочка = **слой + нормализация + нелинейность**. Так строится каждый уровень нейросети.

---

### 8. Как работает `self`, `self.fcX`, `self.bnX`, `self.dropoutX`

- `self` — ссылка на текущий экземпляр класса (модель).
- `self.fc1 = nn.Linear(...)` — создаёт слой и сохраняет его как поле объекта.
- `self.bn1` и `self.dropout1` — так же сохраняются, чтобы быть доступными внутри `forward()`.
- Все поля, созданные через `self`, будут автоматически переданы в `.parameters()` и участвуют в обучении.

---

### 9. Выводы

| Параметр | SimpleNet | BetterNet |
|---|---|---|
| Количество слоёв | 2 | 4 |
| Dropout | ❌ | ✅ |
| BatchNorm | ❌ | ✅ |
| Обучение | стабильное | стабильнее и надёжнее |

| Параметр | SimpleNet | BetterNet |
|---|---|---|
| Переобучение | возможен | менее вероятен |
| Качество | хорошее | лучше (при равных условиях) |

---

## 💙Заключение

Теперь ты не просто построил нейросеть — ты её **расширил**, **улучшил** и **понял**, как именно работает каждый компонент.

Ты знаешь:

- Как улучшить модель
- Как бороться с переобучением
- Как анализировать результат
- Как работает каждый шаг обучения

Ты делаешь **AI**, а не просто запускаешь код. Молодец 🔆💙