ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ

Лекция 8

Transfer Learning и Fine-Tuning CNN

Перенос признаков, заморозка слоёв, ResNet, MobileNet и практическая настройка PyTorch

Преподаватель: Аршын Алтыбай

Кафедра компьютерной инженерии,
Алматинский технологический университет, Алматы, Казахстан

Учебный год: 2026

> **Связь с лекцией 7.** На предыдущем занятии мы построили CNN из свёрток, пулинга и полносвязных слоёв. На практике обучение большой CNN с нуля может потребовать значительного объёма размеченных данных и вычислений. Сегодня мы исследуем, как использовать уже выученные представления и адаптировать их к новой задаче [6, 3, 4].

1. Цели, результаты обучения и план

Цель: изучить постановку переноса обучения, строго различать извлечение признаков и дообучение модели, выбрать стратегию адаптации CNN и построить воспроизводимый эксперимент. По завершении лекции студент должен уметь:

1. формулировать исходный и целевой домены и понимать границы переноса;
2. отличать feature extraction, partial fine-tuning и full fine-tuning;
3. заменять классификационную голову ResNet/MobileNet и подсчитывать её параметры;
4. объяснять градиентный поток, `requires_grad`, `train()` и `eval()`;
5. настраивать preprocessing согласно документации выбранных pretrained weights;
6. проводить корректное разбиение на train/validation/test и выбирать метрики;
7. реализовать двухэтапную настройку PyTorch и сравнить её с baseline;

8. обнаруживать отрицательный перенос и диагностировать ошибки эксперимента.

План: мотивация; теория доменов; стратегии переноса; архитектуры; оптимизация и нормализация; data pipeline; PyTorch; численные примеры; методология сравнения; вопросы.

2. Почему перенос обучения необходим?

Предположим, требуется распознавать четыре класса промышленных деталей, но доступны лишь несколько сотен размеченных изображений. Обучение CNN с миллионами параметров с нуля часто приводит к высокой дисперсии оценок и переобучению. Вместо случайных весов можно начать с параметров модели, обученной на крупном внешнем наборе изображений. Ранние слои CNN часто выявляют границы и текстуры, тогда как поздние слои специализируются на более сложных комбинациях признаков [8, 6].

> **Ключевой принцип** Transfer learning — это *повторное использование знаний* из исходной задачи при решении новой. Оно не гарантирует повышения качества: при существенном несовпадении доменов, задач или меток возможен отрицательный перенос.

2.1 Три практических сценария

1. **Малый датасет, близкие изображения:** чаще начинают с замороженного backbone и новой головы.
2. **Средний датасет с умеренным сдвигом:** обучают голову, затем размораживают последние блоки.
3. **Большой датасет со значимыми отличиями:** возможно полное fine-tuning с пониженной скоростью обучения и тщательной регуляризацией.

Эти стратегии являются эвристиками, а не математическими правилами: окончательное решение принимают по валидационным результатам и стоимости обучения.

3. Формализация исходной и целевой задач

Обозначим исходную задачу через $\mathcal{T}_s$ и целевую через $\mathcal{T}_t$. Исходные пары (x_i^s, y_i^s) и целевые пары (x_j^t, y_j^t) могут иметь различающиеся распределения $P_s(X, Y)$ и $P_t(X, Y)$. Пусть обученная сеть представлена композицией

$$f_{\theta,\phi}(x) = g_\phi(h_\theta(x)), \qquad h_\theta : \mathcal{X} \to \mathbb{R}^d, \quad g_\phi : \mathbb{R}^d \to \mathbb{R}^K. \tag{1}$$

Здесь h_θ — *backbone* (экстрактор признаков), а g_ϕ — голова классификации. Для новой задачи число классов K может измениться.

3.1 Оптимизационная постановка

Для целевой обучающей выборки D_t эмпирический риск равен

$$\widehat{R}_t(\theta, \phi) = \frac{1}{n} \sum_{i=1}^{n} \ell\big(g_\phi(h_\theta(x_i)), y_i\big). \tag{2}$$

При **feature extraction** решается $\min_\phi \widehat{R}_t(\theta_s, \phi)$ с фиксированным θ_s; при **full fine-tuning** минимизируется $\widehat{R}_t(\theta, \phi)$ по обоим наборам параметров. Частичная разморозка означает, что оптимизируется лишь подмножество компонент θ [3, 5].

3.2 Регуляризация относительно исходной модели

Иногда обновление backbone ограничивают штрафом за отклонение от исходных весов:

$$J(\theta, \phi) = \widehat{R}_t(\theta, \phi) + \frac{\lambda}{2} \|\theta - \theta_s\|_2^2. \tag{3}$$

Это один из возможных способов стабилизации переноса; он не эквивалентен обычному weight decay $\|\theta\|_2^2$. В реальной практике чаще используют малый learning rate, early stopping и сохранение лучшей модели.

4. Что именно переносится в CNN?

4.1 Иерархия признаков

После входного изображения x сеть последовательно формирует представления $h_1(x), h_2(x), \ldots, h_L(x)$. Для типичной CNN первые свёртки реагируют на локальные цветовые переходы; промежуточные блоки формируют текстуры и части объектов; последние представления специализируются под исходные классы. Однако степень общности зависит от данных, архитектуры и глубины слоя [8].

4.2 Почему замена последнего слоя полезна?

Пусть pretrained ResNet заканчивается вектором признаков $z \in \mathbb{R}^{512}$ и классификатором на 1000 классов. Для четырёх классов можно заменить последний линейный слой функцией

$$s = Wz + b, \qquad W \in \mathbb{R}^{4\times 512}, \quad b \in \mathbb{R}^4.$$

Число параметров новой головы:

$$N_{\text{head}} = K(d+1) = 4(512+1) = 2052. \tag{4}$$

Сравнение этих 2052 параметров с полным backbone иллюстрирует экономию при обучении только классификатора. Значение d для *конкретных весов и варианта архитектуры* необходимо проверять программно.

> **Не путайте количество обучаемых параметров и стоимость инференса.** При замороженном backbone его веса не обновляются, но изображение всё равно проходит через все свёрточные блоки при каждом предсказании.

5. Три стратегии адаптации: строгое сравнение

Стратегия	Что обучается?	Главный компромисс
Feature extraction	Только новая голова	Меньше обновляемых весов; возможно недостаточное приспособление
Partial fine-tuning	Голова и несколько последних блоков	Баланс адаптации и риска переобучения
Full fine-tuning	Все слои сети	Максимальная гибкость, но выше стоимость и риск забывания

5.1 Feature extraction

Фиксируем $\theta = \theta_s$. Градиент для головы вычисляется по цепному правилу,

$$\nabla_\phi \ell = \frac{\partial \ell}{\partial s}\frac{\partial s}{\partial \phi},$$

а веса backbone остаются неизменными. В PyTorch удобно установить `requires_grad_(False)` у всех параметров backbone, затем заменить классификатор. Оптимизатор должен получать только обучаемые параметры.

5.2 Partial fine-tuning

После обучения головы можно разморозить последний residual stage (например, `layer4` у ResNet18) и обучать его с меньшим learning rate, чем новый классификатор. Обоснование: поздние признаки обычно сильнее специализированы под старую задачу. Обязателен контроль переобучения и правильный режим BatchNorm.

5.3 Full fine-tuning

При существенном различии изображений новая задача может требовать адаптации почти всех представлений. Можно разморозить сеть полностью; при этом важно использовать достаточно малую скорость обучения для pretrained layers и больший rate для новой головы. Универсального соотношения rates нет: значения выбираются на validation set.

6. Градиентный поток и заморозка слоёв

6.1 Цепное правило для двухэтапной сети

Для $z = h_\theta(x)$ и $\widehat{y} = g_\phi(z)$ получаем

$$\nabla_\theta \ell = \left(\frac{\partial h_\theta}{\partial \theta}\right)^\top \left(\frac{\partial g_\phi}{\partial z}\right)^\top \nabla_{\widehat{y}} \ell. \tag{5}$$

Если backbone заморожен, оптимизатор не изменяет θ. При последующей разморозке необходимо *пересоздать или обновить* оптимизатор, чтобы в нём появились размороженные параметры.

Пример 1. Заморозка и последующая разморозка ResNet18

```python
import torch
from torchvision.models import resnet18, ResNet18_Weights
weights = ResNet18_Weights.DEFAULT
model = resnet18(weights=weights)
for p in model.parameters():
    p.requires_grad = False
in_features = model.fc.in_features
model.fc = torch.nn.Linear(in_features, 4)
# Этап 1: только голова
optimizer = torch.optim.AdamW(model.fc.parameters(), lr=1e-3)
# Позже: разморозить блок и ПЕРЕСОЗДАТЬ optimizer
for p in model.layer4.parameters():
    p.requires_grad = True
optimizer = torch.optim.AdamW([
    {"params": model.layer4.parameters(), "lr": 1e-5},
    {"params": model.fc.parameters(), "lr": 1e-4},
], weight_decay=1e-4)
```

6.2 Особый случай Batch Normalization

BatchNorm хранит не только обучаемые γ, β, но и буферы скользящих средних и дисперсий. Следовательно, установка `requires_grad=False` *не гарантирует фиксацию running statistics*. В режиме `train()` буферы могут обновляться. Для небольшого целевого набора данных иногда предпочтительно удерживать замороженные BatchNorm-слои в `eval()` и сравнивать такую стратегию с адаптацией статистик [11].

> **Типичная ошибка** Выражения `model.eval()` и `torch.no_grad()` отвечают на *разные вопросы*: первое меняет поведение Dropout/BatchNorm, второе отключает построение вычислительного графа для операций в соответствующем блоке. Для валидации обычно нужны *оба*.

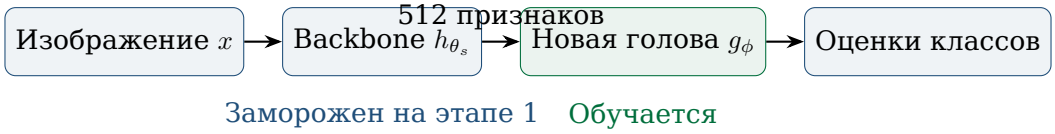
7. ResNet: остаточные блоки и перенос обучения

ResNet использует остаточное отображение

$$y = F(x; W) + S(x), \tag{6}$$

где $S(x) = x$ для совпадающих размерностей либо проекция при их изменении. Это облегчает обучение глубоких CNN [9]. В ResNet18 последняя голова доступна как `model.fc`; выход backbone перед ней имеет размерность 512.

7.1 Архитектурная схема



8. MobileNet: эффективный перенос на ограниченных устройствах

MobileNet использует *depthwise separable convolutions*: пространственная depthwise-свёртка отдельно по каждому каналу, затем pointwise-свёртка 1×1 смешивает каналы [10]. Для входных C_{in} и выходных C_{out} каналов и ядра $k \times k$ числа весов (без bias) равны

$$N_{\text{std}} = k^2 C_{\text{in}} C_{\text{out}}, \tag{7}$$

$$N_{\text{sep}} = k^2 C_{\text{in}} + C_{\text{in}} C_{\text{out}}. \tag{8}$$

При $k = 3$, $C_{\text{in}} = 32$, $C_{\text{out}} = 64$ получаем

$$N_{\text{std}} = 9 \cdot 32 \cdot 64 = 18432, \qquad N_{\text{sep}} = 9 \cdot 32 + 32 \cdot 64 = 2336.$$

Это снижение числа весов примерно в 7.9 раза для *данной пары операций*. На практике учитывают также нормализацию, активации, память и особенности оборудования; уменьшение числа параметров не гарантирует равного ускорения исполнения.

8.1 Как выбирают между ResNet и MobileNet?

ResNet часто удобна как прозрачная baseline-архитектура; MobileNet предпочтительна, когда есть строгие ограничения по размеру модели, задержке или энергопотреблению. Решение принимают по совместному сравнению accuracy, latency, memory footprint и доступности pretrained weights [3, 5].

9. Подготовка данных: критически важные условия

9.1 Корректное разбиение

Данные делят на train/validation/test. Разметка тестовой выборки не используется для выбора архитектуры или learning rate. Если изображения происходят от одного объекта, пациента, серии кадров или партии производства, нужно рассмотреть *group split*, иначе почти идентичные объекты могут попасть в разные части набора и создать иллюзию высокой точности.

9.2 Preprocessing, связанный с весами

Torchvision предоставляет объект весов с рекомендованной трансформацией через `weights.transforms()`. Он может включать resizing, cropping, преобразование в тензор и нормализацию. Несоответствие статистик или цветовых каналов способно ухудшить inference [12]. Воспроизводимая система фиксирует не только weights, но и полный preprocessing pipeline.

Пример 2. Подготовка изображений для предобученной ResNet

```
from torchvision.models import ResNet18_Weights
weights = ResNet18_Weights.DEFAULT
val_transform = weights.transforms()
# Для обучения можно использовать умеренную аугментацию,
# обеспечив сопоставимые выходные размеры и нормализацию.
```

9.3 Data augmentation без разрушения меток

Для природных изображений полезны случайные отражения, небольшие повороты и изменение яркости, если они сохраняют класс. Для медицинских или специализированных изображений любое преобразование должно быть физически и предметно обосновано. *Не следует* аугментировать validation/test случайным образом при стандартном сравнении моделей.

10. Математика классификационной головы

10.1 От признаков к вероятностям

Для K классов при $z = h_\theta(x)$ определим logits $s = Wz + b$ и softmax

$$p_k = \frac{e^{s_k}}{\sum_{j=1}^{K} e^{s_j}}. \tag{9}$$

Для правильного класса y cross-entropy:

$$\ell(s, y) = -\log p_y = -s_y + \log\left(\sum_{j=1}^{K} e^{s_j}\right). \tag{10}$$

Производная по logit s_k равна

$$\frac{\partial \ell}{\partial s_k} = p_k - \mathbb{1}\{k = y\}. \tag{11}$$

Следовательно, для одного объекта $\nabla_W \ell = (p - e_y)z^\top$, а $\nabla_b \ell = p - e_y$. Этой простой производной достаточно для эффективного обучения линейной головы [7, 6].

10.2 Численный пример

Пусть $z = (1, 2)^\top$, три класса, $s = (1, 0, -1)^\top$ и правильный класс $y = 1$ (при нумерации с нуля). Тогда

$$(p_0, p_1, p_2) \approx (0.6652, 0.2447, 0.0900), \qquad \ell = -\log(0.2447) \approx 1.4076.$$

Градиент по смещению $(0.6652, -0.7553, 0.0900)^\top$; каждая строка $\nabla_W \ell$ равна соответствующему компоненту этого вектора, умноженному на $(1, 2)$. При этом `torch.nn.CrossEntropyLoss` принимает *logits*, а не заранее преобразованные softmax-вероятности.

11. Пошаговый учебный проект на PyTorch

Мы решаем задачу классификации пользовательских изображений по четырём классам. Предполагается структура каталога `data/train/class_a`, `data/train/class_b`, ...и аналогичная `data/val`. Эти папки и изображения должны быть созданы студентом заранее. Для тестирования можно выделить отдельный каталог `data/test`; его нельзя использовать при выборе гиперпараметров.

11.1 Шаг 1. Data loaders

Пример 3. Загрузка train/validation наборов

```python
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.models import resnet18, ResNet18_Weights

weights = ResNet18_Weights.DEFAULT
val_tf = weights.transforms()
train_tf = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.75, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.485, 0.456, 0.406),
                         std=(0.229, 0.224, 0.225)),
])
train_ds = datasets.ImageFolder('data/train', transform=train_tf)
val_ds = datasets.ImageFolder('data/val', transform=val_tf)
assert train_ds.class_to_idx == val_ds.class_to_idx
num_classes = len(train_ds.classes)
train_loader = DataLoader(train_ds, batch_size=16, shuffle=True)
val_loader = DataLoader(val_ds, batch_size=32, shuffle=False)
```

Здесь `ImageFolder` создаёт номера классов по именам каталогов. Проверка `class_to_idx` обязательна: иначе индекс класса при оценке может означать другую категорию. При малой выборке лучше отдельно контролировать баланс классов.

11.2 Шаг 2. Предобученная модель и новая голова

Пример 4. Адаптация ResNet18

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = resnet18(weights=weights)
for p in model.parameters():
    p.requires_grad = False
model.fc = nn.Linear(model.fc.in_features, num_classes)
model = model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model.fc.parameters(),
                              lr=1e-3, weight_decay=1e-4)
print('Classes:', train_ds.classes)
print('Trainable:', sum(p.numel() for p in model.parameters()
                        if p.requires_grad))
```

Код не предполагает заранее полученного численного результата: число классов зависит от фактического набора. Весовые коэффициенты модели при первом запуске могут автоматически загружаться из интернета; в офлайн-среде pretrained checkpoint нужно подготовить отдельно.

11.3 Шаг 3. Корректная функция обучения/оценки

Пример 5. Одна эпоха с контролем режима обучения

```
def run_epoch(model, loader, criterion, optimizer=None):
    training = optimizer is not None
    model.train(training)
    # Для замороженного backbone не обновляем BNстатистики-.
    if training:
        for module in model.modules():
            if isinstance(module, nn.modules.batchnorm._BatchNorm):
                module.eval()
    total_loss = 0.0
    correct = 0
    total = 0
    with torch.set_grad_enabled(training):
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            if training:
                optimizer.zero_grad(set_to_none=True)
            logits = model(x)
            loss = criterion(logits, y)
            if training:
                loss.backward()
                optimizer.step()
            total_loss += loss.item() * len(y)
            correct += (logits.argmax(dim=1) == y).sum().item()
            total += len(y)
```

Пример 5 (продолжение)

```
    return total_loss / total, correct / total
```

Функция возвращает *средний loss на объект* и accuracy. При разном размере последнего batch простое среднее по batch losses было бы некорректным. Замороженные BatchNorm-статистики фиксируем явно; если преподаватель намеренно исследует адаптацию BatchNorm, соответствующий блок кода надо изменить.

11.4 Шаг 4. Двухэтапное обучение

Пример 6. Этап 1: обучение головы; этап 2: частичное fine-tuning

```
best_acc = -1.0
best_state = None
for epoch in range(4):
    tr_loss, tr_acc = run_epoch(model, train_loader, criterion,
                                optimizer=optimizer)
    va_loss, va_acc = run_epoch(model, val_loader, criterion)
    print(epoch + 1, round(tr_loss, 4), round(va_acc, 4))
    if va_acc > best_acc:
        best_acc = va_acc
        best_state = {k: v.detach().cpu().clone()
                      for k, v in model.state_dict().items()}

model.load_state_dict(best_state)
for p in model.layer4.parameters():
    p.requires_grad = True
optimizer = torch.optim.AdamW([
    {'params': model.layer4.parameters(), 'lr': 1e-5},
    {'params': model.fc.parameters(), 'lr': 1e-4},
], weight_decay=1e-4)
# Аналогично обучите 2--4 эпохи и вновь сохраните
# лучший checkpoint по validation metric.
```

> Приведённые 4 эпохи и learning rates — учебные отправные значения, *не* универсальные оптимумы. Для настоящего исследования следует фиксировать seed, число прогонов, вычислительную среду, версии библиотек и итоговое время обучения.

11.5 Шаг 5. Воспроизводимое сохранение модели

Пример 7. Сохранение checkpoint и метаданных

```
checkpoint = {
    'model_state': model.state_dict(),
    'classes': train_ds.classes,
    'architecture': 'resnet18',
    'weights_id': str(weights),
    'num_classes': num_classes,
}
torch.save(checkpoint, 'transfer_resnet18.pt')
```

При повторной загрузке создайте ту же архитектуру и голову соответствующего размера, затем загрузите `model_state` и установите `model.eval()`. Сохраняйте также параметры preprocessing. Файлы с сериализованными объектами загружайте только из доверенных источников.

12. Эксперимент: как честно сравнить стратегии

12.1 Baseline и варианты

Планируем три эксперимента при одинаковом датасете и протоколе:

1. **Baseline с нуля:** ResNet18 с случайными весами (при достаточно больших данных и реалистичном бюджете) либо небольшая CNN, обученная с нуля.
2. **Замороженный backbone:** pretrained ResNet18 + новая голова.
3. **Partial fine-tuning:** pretrained ResNet18 + новая голова + `layer4`.

Сравним validation loss, macro-F1, balanced accuracy, training time, inference latency и число обучаемых параметров. Окончательно оцениваем выбранный вариант на независимом test set один раз. В случаях неравных классов одной accuracy недостаточно.

12.2 Шаблон протокола

Вариант	Val F1	Test F1	Время	Параметры
Обучение с нуля	—	—	—	—
Frozen backbone	—	—	—	—
Fine-tuning layer4	—	—	—	—

Важно: прочерки означают отсутствие проведённого эксперимента, а не нулевые результаты. Не следует представлять придуманные метрики как измеренные.

12.3 Проверка дисперсии

Если данных мало, результаты зависят от разделения и случайной инициализации головы. Для надежной оценки можно повторить обучение с несколькими seed и сообщать среднее и стандартное отклонение на validation; окончательный test оставить закрытым до принятия решения.

13. Проблема отрицательного переноса

Отрицательный перенос возникает, если использование pretrained модели ухудшает качество по сравнению с разумной baseline-моделью. Возможные причины: иной тип изображений, недостаточная адаптация, неверная нормализация, изменение цветовой модели, смещение домена и переобучение новой головы.

13.1 Наблюдаемые признаки

- Train accuracy быстро растёт, validation accuracy не растёт: возможны переобучение, малая выборка или сильное различие распределений.
- И train, и validation остаются слабыми: признаки неадекватны или задача требует глубокой адаптации.
- После разморозки качество падает: learning rate велик либо BatchNorm-статистики неустойчивы.
- Уверенные ошибки на новых объектах: возможно смещение домена и недостаточная калибровка.

Для диагностики полезны confusion matrix, per-class recall, ошибочные изображения и анализ уверенности предсказаний.

14. Трудности в реальных проектах

14.1 Domain shift

Промышленные фото могут отличаться от ImageNet по освещению, текстурам и масштабу объектов. Медицинские изображения отличаются ещё сильнее. Поэтому успешный перенос в одной задаче не означает такой же успех в другой.

14.2 Class imbalance

При редком дефектном классе можно использовать class-weighted loss, сбалансированное формирование mini-batches, подходящие метрики и настройку threshold. Все решения принимаются по train/validation, а не по тестовой выборке.

14.3 Аугментация и leakage

Аугментация применяется *после* группового разделения данных. Если сначала создать несколько копий одного изображения, а потом случайно разбить их на train и test, практически одинаковые картинки окажутся по обе стороны разбиения.

14.4 Лицензии и документация

Наличие pretrained weights не означает автоматического права применять модель и данные в любом продукте. Проверяйте лицензии весов, модели и данных, а также документируйте классы, происхождение набора и известные ограничения.

15. Рекомендации по проектированию эксперимента

1. Сначала проверьте датасет и гарантируйте отсутствие повторов между train/val/test.
2. Запустите простой baseline и сохраните его результаты.
3. Подберите архитектуру pretrained модели под ограничения вычислений.
4. Сопоставьте трансформации данных и предобученные веса.
5. Обучите новую голову без изменения backbone.
6. При необходимости разморозьте последние блоки и уменьшите learning rate.
7. Контролируйте BatchNorm/Dropout и сохраняйте лучший validation checkpoint.
8. После выбора конфигурации оцените независимый test, выполните error analysis.
9. Сохраните weights, карту классов, preprocessing, версии библиотек и протокол обучения.

16. Выводы

Перенос обучения превращает уже изученное глубокое представление в полезный начальный ресурс для новой задачи. Feature extraction даёт лёгкую и быструю адаптацию головы; partial fine-tuning повышает гибкость; full fine-tuning позволяет глубоко изменить представления, но требует большей осторожности. Сопоставимый preprocessing, корректная фиксация параметров, контроль BatchNorm, валидационная дисциплина и качественный error analysis не менее важны, чем выбранная архитектура.

17. Контрольные вопросы

1. Что называется transfer learning и чем оно отличается от обучения с нуля?
2. Что такое исходный и целевой домены?
3. Какие слои CNN обычно формируют более общие признаки?
4. Что такое backbone и classification head?
5. Объясните feature extraction.
6. Чем partial fine-tuning отличается от full fine-tuning?
7. Почему pretrained модель может ухудшить качество на новом домене?
8. Запишите формулу эмпирического целевого риска (2).
9. Почему штраф $\|\theta - \theta_s\|^2$ отличается от weight decay?
10. Для чего нужно замораживать веса?
11. Что делает параметр `requires_grad` в PyTorch?
12. Почему после разморозки иногда нужно пересоздавать оптимизатор?
13. Чем `eval()` отличается от `no_grad()`?
14. Как изменяется поведение BatchNorm при переводе в режим обучения?
15. Объясните формулу остаточного блока ResNet.
16. Как заменить классификатор ResNet18 для 5 классов?
17. Подсчитайте количество параметров головы $512 \to 5$.
18. Чем MobileNet отличается от обычной CNN?
19. Выведите формулу числа весов depthwise separable convolution.
20. Почему важно применять `weights.transforms()`?
21. Как правильно разделить изображения одной производственной партии?
22. Что показывает macro-F1 при дисбалансе классов?
23. Почему нельзя выбирать learning rate по тестовому набору?
24. Какие признаки указывают на отрицательный перенос?
25. Какие артефакты необходимо сохранить для воспроизводимого inference?

18. Задания для самостоятельной работы

1. Для $d = 512$, $K \in \{2, 5, 20\}$ вычислите число параметров новой головы.
2. Вычислите числа весов обычной и depthwise separable свёртки при $C_{\text{in}} = 64$, $C_{\text{out}} = 128$, $k = 3$.
3. Докажите формулу (11) для softmax cross-entropy.
4. Реализуйте feature extraction для двух- или трёхклассовой задачи с ResNet18.
5. Сравните frozen backbone и partial fine-tuning при одном и том же разделении данных.
6. Проведите отдельный эксперимент с замороженными и адаптируемыми BatchNorm-статистиками и сформулируйте вывод.
7. Подготовьте confusion matrix и определите классы с наименьшей полнотой.
8. Сравните ResNet18 и MobileNetV3 по размеру checkpoint, latency и качеству для вашей задачи.
9. Продемонстрируйте пример утечки из-за дубликатов и исправьте split.
10. Подготовьте техническую карточку модели с назначением, ограничениями, набором данных и preprocessing.

19. Литература

Основные учебные издания [1, 2, 3, 4, 5, 6, 7] дополнены оригинальными исследованиями о переносе и архитектурах CNN [8, 9, 10] и официальной документацией PyTorch [12].

Список литературы

[1] Trask A. W. *Grokking Deep Learning*. Manning Publications, 2019. Рус. изд.: «Грокаем глубокое обучение». https://www.manning.com/books/grokking-deep-learning.

[2] Rashid T. *Make Your Own Neural Network*. 2016. Рус. изд.: «Глубокое обучение. Лёгкая теория и практика на Python». https://makeyourownneuralnetwork.blogspot.com/.

[3] Géron A. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 3rd ed. O'Reilly, 2022. https://www.oreilly.com/library/view/hands-on-machine-learning/9781098125967/.

[4] Chollet F. *Deep Learning with Python*. 2nd ed. Manning, 2021. https://www.manning.com/books/deep-learning-with-python-second-edition.

[5] Stevens E., Antiga L., Viehmann T. *Deep Learning with PyTorch*. Manning, 2020. https://www.manning.com/books/deep-learning-with-pytorch.

[6] Goodfellow I., Bengio Y., Courville A. *Deep Learning*. MIT Press, 2016. https://www.deeplearningbook.org/.

[7] Nielsen M. A. *Neural Networks and Deep Learning*. 2015. http://neuralnetworksanddeeplearning.com/.

[8] Yosinski J., Clune J., Bengio Y., Lipson H. How Transferable Are Features in Deep Neural Networks? *NeurIPS*, 2014. https://arxiv.org/abs/1411.1792.

[9] He K., Zhang X., Ren S., Sun J. Deep Residual Learning for Image Recognition. *CVPR*, 2016, pp. 770–778. https://doi.org/10.1109/CVPR.2016.90.

[10] Howard A. G. et al. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. 2017. https://arxiv.org/abs/1704.04861.

[11] Ioffe S., Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *ICML*, 2015. https://arxiv.org/abs/1502.03167.

[12] PyTorch/Torchvision. Models and Pre-trained Weights, official documentation. https://docs.pytorch.org/vision/stable/models.html.