Jak udělat CNN nad 25%

Vysvětlení pojmů

Epoch	Jeden průběh skrze celý trénovací dataset
Batch	Náhodně vybraných N obrazů z datasetu

Výběr hyperparametrů

Learning rate

learning_rate

Dost záleží, jak vám záleží na přesnosti/rychlosti učení. Čím vyšší, tím to bude rychlejší, ale šance na zlepšení bude hlavně v dalších epochách výrazně menší. Ideální hodnoty jsou přibližně $\alpha_t \in$ [**0.001, 0.1**], s tím, že pro menší sítě bude větší hodnota lepší.

Velikost batche

batch_sz

K *Stochastic Gradient Descentu* potřebujeme z datasetu vybrat jen pár hodnot. Doporučuju minimálně pár desítek (**32** je taková pěkná hodnota) pokud nemáte moc výkonné PC a nevejde se to do paměti. Pokud můžete zkuste i víc, ale nešel bych přes pár stovek **(128, 256)**.

Počet epoch

epochs

Ten jejich testovací dataset na BRUTE je pravděpodobně dost rozdílný než standartní FashionMNIST. Z pár testů bych řekl, že to začne dost silně overfittovat po tak **5** epoše (rozdíl mojí validace proti BRUTE validaci $> 20\%$). Používám celý dataset, pokud používáte jen zlomek, víc epoch bude ok.

Velikost použitého datasetu

Záleží, jak dlouho chcete cvičit. Čím víc dat použijeme, tím bude lepší generalizace, ale poběží to déle. Většinou se to splituje **9:1**.

Optimizátor

Nic moc bych tady nevymýšlel, jen přidat momentum.

optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)

Nemusí být 0.9, ale funguje to pěkně.
(Další možnost je použít Adam, ale to už je dost mimo.)

Architektura sítě

Základní struktura konvoluční části je

Konvoluční vrstva ⇒ (Normalizace batche) ⇒ ReLU ⇒ Max pooling.

Dávat pozor, aby počet kanálů vycházející z jedné vrstvy byl stejný jako počet vstupních kanálů další vrstvy.

Konvoluční vrstva

Hlavním úkolem konvoluční vrstvy je najít spojitosti mezi nějakým okolím daného pixelu buď ve vstupu, nebo feature mapě. Co se týče velikosti kernelu, je podle mě lepší víc kanálů a menší kernel, než obří kernel a pár kanálů. Určitě ne víc než **7**, s tím, že **3** je asi nejčastěji používaná velikost.

Padding zamezí snižování velikosti vstupu na krajích, např. pro kernel_size=(3, 3) je pro zachování velikosti padding=1. Není potřeba, ale může pomoci.

Stride bych vůbec nepoužíval. Normálně to downsampluje, ale to udělá pooling taky a je v našem případě mnohem lepší.

Příklad:

nn.Conv2d(in_channels, out_channels, (5, 5), padding=2)

Nelineární vrstva

Použít ReLU. Pokud chcete trochu grilovat s benzínem možná parametrický ReLU, ale na tohle je to zbytečný.

nn.ReLU()

Poolovací vstrva

Máme buď max pooling, nebo average pooling. Max bude pro nás většinou lepší, ale určitě určitě má smysl zkusit avg.
Příklad 3x3 max-pooling vrstvy:

nn.MaxPool2d((3, 3))

Pokud nechceme mít na tvrdo tvar poolu, má PyTorch adaptive pooling, ale to je jen třešnička.

```
class ConvolutionalPart(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ConvolutionalPart, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, (5, 5), padding=2)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d((3, 3))

    def forward(self, x):
        x = self.conv(x)
        x = self.relu(x)
        x = self.pool(x)
        return x
```

Obrázek 1: Příklad konvoluční části sítě

Většinou použijeme několik těchto částí za sebou...

Přechod z konvoluční části ke spojité

Obecně máme po průchodu konvolučními vrstvami nějakou 4D matici o batchSize x channelCount x width x height. Potřebujeme, aby navazující lineární vrstva měla velikost vstupu přesně channelCount x width x height. Například pokud máme na vstupu 5x5 mapu o 64 kanálech, pak musíme vytvořil lineární vrstvu s velikostí 1600. Pozor na velikost lineárních vrstev, žerou hodně paměti. Dále je dobrý nápad nemít ReLU těsně před přechodem do lineárních vrstev (samozřejmě mezi konvolucemi a mezi lineárními vrstvami musí být)

```python
class ExampleNet(nn.Module):
    def __init__(self):
        self.conv_part = ConvolutionalPart(1, 64)
        self.flatten = nn.Flatten()
        self.linear = nn.Linear(64 * 5 * 5, 10)

    def forward(self, x):
        x = self.conv_part(x)
        x = self.flatten(x)
        x = self.linear(x)
        return x
```

Obrázek 2: Přechod mezi konvolucemi a lineární vrstvou

Lineární vrstvy

Dobré pravidlo je, méně je více. Čím víc redukujeme velikost obrázku v konvoluční části, tím v našem případě lépe. Pokud to uděláme správně, stačí jen jediná lineární vrstva z nějakého shapu na těch 10 co potřebujeme. Pokud je tam chcete mít, tak zase velikost výstupu předchozího je roven velikosti vstupu dalšího. ReLU mezi. **NEDÁVAT** ReLU na konec! Znovu, chceme použít minimum lin. vrstev, neříká se tomu **konvoluční** sítě pro nic za nic.

```python
class LinearNet(nn.Module):
    def __init__(self):
        self.linear1 = nn.Linear(1600, 400)
        self.linear2 = nn.Linear(400, 64)
        self.linear3 = nn.Linear(64, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.linear1(x))
        x = self.relu(self.linear2(x))
        x = self.linear3(x)
        return F.log_softmax(x, dim=1)
```

Obrázek 3: Příklad lineárních vrstev

Další doporučení a triky

Nic, co je tady není potřeba k 25, takže to vezmu stroze s odkazy.

CUDA

Pokud máte nVidia GPU, použití CUDA toolkitu zrychlí několikanásobně učení (závisí na rychlosti procesoru a grafické karty). Každopádně to má pár GB, takže asi tak. Na Intel a AMD smůla. `https://developer.nvidia.com/cuda-downloads`

nn.Sequential

Pro líné naprosto ideální. Pokud zabalíte jednotlivé vrstvy do nn.Sequential(), nemusíme, pak ve forward metodě všechno vypisovat.

```
class MyNet(nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 64, (5, 5), padding=1),
            nn.ReLU(),
            nn.MaxPool2d((2, 2)),
            nn.Conv2d(64, 256, (3, 3)),
            nn.ReLU(),
            nn.MaxPool2d((2, 2)),
            nn.Conv2d(256, 512, (3, 3)),
            nn.AdaptiveMaxPool2d((1, 1)),
            nn.Flatten(),
            nn.Linear(512, 10)
            )

    def forward(self, x):
        return F.log_softmax(self.net(x), dim=1)
```

Obrázek 4: Příklad nn.Sequential, není moc dobrá, těžká práce je na vás ;)

Learning Rate Scheduler

Silně doporučuju. Co je v obrázku, mimo loop, na konec loopu pak zavoláme .step() na scheduler. Hodnoty gamma a step_size jsou dosti náhodne, zvolte vlastní.

```
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=2, gamma=0.5)
for epoch in range(1, epochs + 1):
    # training
    t1 = time.time()
    model.train()
    for i_batch, (x, y) in enumerate(trn_loader):
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()
        net_output = model(x)
        loss = F.nll_loss(net_output, y)
        loss.backward()
        optimizer.step()

        if i_batch % 100 == 0:
            t_batch = time.time() - t1
            print('[TRN] Train epoch: {}, batch: {}\tLoss: {:.4f}\tEpoch time: {:.4f}'.format(
                epoch, i_batch, loss.item(), t_batch))

    lr_scheduler.step()
```

Obrázek 5: Jak to implementovat

Data augment

`https://pytorch.org/vision/stable/transforms.html`