KogSys-ML-B: Einführung in Maschinelles Lernen

Deep Learning 2: Training Procedure

---

2 Deconstructing the Basic Training Loop

At the beggining of training, we initialize the Model to random weights (which is done by just calling the constructor) and moving it to the optimal device (which is set in one of the recap-cells).

```
model = Model()
model = model.to(get_device())
```

We then choose our loss function and optimizer. The loss function needs to be callable (in this case it is a callable object), and the optimizer is a `torch.optim.Optimizer` object, in this case Stochastic Gradient Descent. The optimizer must always be passed the parameters which it should optimize. We get those by calling `model.parameters()`.

```
loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), momentum=0.9)
```

In the training loop, we first iterate over epochs, and secondly over a dataloader – the training loader for training the model, obviously. We unpack the batch provided by the dataloader to labels and images (inputs). `optimizer.zero_grad()` sets the previously calculated gradients to zero. We then send the inputs to the models and calculate the loss function using the labels and model outputs. We then backpropagate the loss by calling `loss.backward()`, and optimize the parameters by calling `optimizer.step()`.

We record the batch `idx` using `enumerate` to be able to calculate average losses for recording training information.

At the end of training, we save the model to a file.

```
print("Starting Training")
model.train()

for epoch in range(10):  # Limit to 10 epochs to keep the runtime short
    sum_loss = 0.0

    for idx, data in enumerate(train_loader, 0):
        inputs, labels = data[0].to(get_device()),
            data[1].to(get_device())

        optimizer.zero_grad()
```

```
12
13          outputs = model(inputs)
14          loss = loss_func(outputs, labels)
15          loss.backward()
16          optimizer.step()
17
18          sum_loss += loss.item()
19          if idx % 1000 == 999:
20              print(f"Epoch {epoch + 1}, batch {idx + 1}:",
21                      f"loss {sum_loss / 1000:.3f}")
22              sum_loss = 0.0
23
24  print("Finished Training")
25
26  model.eval()
27  torch.save(model.state_dict(), "cifar10_cnn.pth")
```

3 Building a Better Training Loop

3.1 Optimizers: Adam **What is Adam?** Adam is an optimization algorithm (Kingma & Ba, 2015). Since then, it has found wide application for optimizing neural network parameters. It extends Stochastic Gradient Descent with both Momentum (regulated by β_1) and Root Mean Square Propagation (RMSprop), which essentially adapts the learning rate for each to-be-optimized parameter individually (regulated by β_2).

```
1  optimizer = torch.optim.Adam(
2      params=model.parameters(),  # parameters to optimize
3      lr=0.001,                   # learning rate
4      betas=(
5          0.9,
6          0.999,
7      ),                          # beta 1 momentum factor, beta 2
8                                  # is RMSprop factor for per-parameter
9                                  # learning rate adjustment
10     eps=1e-8,                   # parameter avoiding RMSprop
11                                 # denominator collapse
12     weight_decay=0,             # Factor to way in the L2 Norm
13                                 # (Euclidean distance) of all weights,
14                                 # i.e. not only minimize loss, but
15                                 # also weight values
16 )
```

3.2 Learning Rate Scheduling Training performance can be increased by adapting the learning rate, with techniques like learning rate warmup (not starting with the full learning rate but increasing it over the first episodes) or decay (reducing the learning rate as training goes on). A very effective technique

demonstrated for example in the Deep Residual Learning for Image Recognition paper (He et al., 2016). See how the classification error decreases visibly after reducing the learning rate:

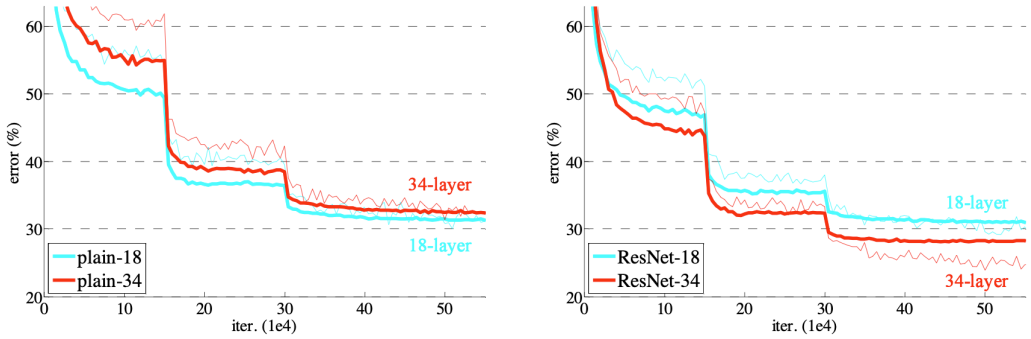

Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

Figure 1: Effect of reducing learning rate on plateauing performance (He et al., 2016)

```python
lr_scheduler = ReduceLROnPlateau(
    optimizer=optimizer,    # the optimizer for which learning rate
                            # should be adapted
    mode="max",             # whether the metric should increase or
                            # decrease. Default is 'min' to be used
                            # with loss, when used with acc should
                            # be 'max'
    patience=3,             # how long a metric must stop improving
                            # by at least best * (1 +/- threshold)
                            # per default, can also be set to
                            # absolute threshold mode
    threshold=1e-4,         # Threshold for patience calculation
    factor=0.1,             # The value to multiply the learning rate
                            # with
)
```

3.3 Validation To track training progress, looking only at the loss is not the best option. While we describe a performance criterion using the loss, it does not tell us as much about model *performance* as say a calculated accuracy on a validation dataset would. Let's do exactly that!

We already have a function for calculating accuracy given a model and a DataLoader. Now we just need to split our dataset, and torch has a function exactly for that: `torch.utils.data.random_split()`. Note that this task (splitting an existing dataset) is different from the one presented in the assignment, where we want to build the split into the Dataset class itself.

Note: Validation Dataset. For datasets that are intended to be used as benchmarks, the test dataset is often either provided without labels or not provided at all. This is done to prevent models from being

trained on the test data to cheat on the leaderboards. The validation dataset is thus all we have to get an estimate of how well our model performs.
In such cases, we should not use the validation dataset for the classic validation tasks (e.g., to calculate metrics for early stopping), but rather treat it as our test dataset. Instead, it is advisable to create our own validation dataset from the training dataset.

```python
train, val = torch.utils.data.random_split(
    dataset=train_data,     # Dataset object to split
    lengths=(0.85, 0.15),   # Fractions of the returned datasets,
                            # must sum to 1
    generator=torch.Generator().manual_seed(
        2025
    ),                      # Ensures reproducibility,
                            # optional parameter
)

train_loader = torch.utils.data.DataLoader(
    train,
    batch_size=8,
    shuffle=True
)
val_loader = torch.utils.data.DataLoader(
    val,
    batch_size=8,
    shuffle=False
)
```

3.4 Checkpointing Using `torch.save()` you can save components of your model and training process by passing a `dictionary`. Such checkpoint files are by convention ending in either `.pth` or `.pt`. A common dictionary to save could be the following:

```python
{
    "epoch": epoch,
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    "scheduler_state_dict": lr_scheduler.state_dict(),
    "loss": loss,
}
```

...which can then be loaded using torch.load, and be accessed like the originally saved dictionary. Models, optimizers and schedulers have `.load_state_dict()` methods to load the stored `state_dict`s.

What Else?

Of course, this isn't everything to learn about PyTorch! Here is an (incomplete) list of resources for you to look at if you want to dive deeper into this framework! Note that some of these are really advanced – so don't worry if you don't understand them, you won't need them for this course.

- Tensorboard visualization of training metrics
- Distributed Data Parallel (DDP), i.e. Multi-GPU Training
- Technically not PyTorch, but an important tool for academic experiments: Running experiments from configs, e.g. via YACS or JSON argparse

Bibliography

Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. In Y. Bengio & Y. LeCun (Hrsg.), 3rd international conference on learning representations, ICLR 2015, san diego, CA, USA, may 7-9, 2015, conference track proceedings. http://arxiv.org/abs/1412.6980

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. 770–778. https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html