Recap from previous Lab

- We "closed the loop" on our first MultiLayer Perceptron (MLP), exploring how the training routine is implemented in PyTorch (PT):
    - we saw how to use built-in loss functions in PT and we learned how to construct custom losses based upon tensor methods
    - moreover, we also saw how to use vanilla Stochastic Gradient Descent (SGD) in conjunction with backpropagation to enable the parameters updating in our MLP

Agenda for today

- The main topic of our lecture is **regularization**
- First of all, though, we will implement a framework for monitoring the parameters during training (the so called *trajectory*), a simple research exercize
- On to regularization, we will see how to utilize various way to infuse regularization into our MLP training, still with an eye on the trajectory:
    - L2 regularization (aka *weight decay*)
    - dropout
    - normalization layers (not really regularization, but they fit well in this lab)

In [1]:
```
import torch
from scripts.architectures import MLP
from scripts.mnist import get_data
from scripts import train
```

In [2]:
```
minibatch_size_train = 256
minibatch_size_test = 512

trainloader, testloader = get_data(batch_size_train=minibatch_size_test, batch_size_test=minibatch_size_test)
```

In [3]:
```
learn_rate = 0.1
momentum = 0.9
num_epochs = 5

device = "cuda" if torch.cuda.is_available() else "cpu"

model = MLP()
model.to(device)
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learn_rate, momentum=momentum)
```

Train the MLP:

In [4]:
```
train_loss, train_acc = train.train_model(model, trainloader, loss_fn, optimizer, num_epochs, device=device)
```

```
Epoch 1 completed. Loss - total: 44017.42878675461 - average: 0.7336238131125769; Performance: 0.7534333333333333
Epoch 2 completed. Loss - total: 13156.425340652466 - average: 0.2192737556775411; Performance: 0.9339166666666666
Epoch 3 completed. Loss - total: 10717.757884025574 - average: 0.1786292980670929; Performance: 0.946
Epoch 4 completed. Loss - total: 9630.115203380585 - average: 0.1605019200563431; Performance: 0.95085
Epoch 5 completed. Loss - total: 8760.895300149918 - average: 0.1460149216691653; Performance: 0.9561166666666666
```

Regularization

Regularization in DL comes in the form of different tools. We can have:

1. Penalty terms in loss functions (e.g. L1 and L2 norm regularization) which introduce bias in our parameters by actively reducing the magnitude of some weights:
    - L1 norm regularization is also called LASSO regularization
    - L2 norm regularization is also called Ridge regularization or **weight decay**
    - they were originally implemented in linear regression models as a way to infuse *inductive bias* in models originally thought to rely on the complete unbiasedness on training data
2. Normalization layers which normalize the incoming information s.t. their mean is zero and standard deviation one. It comes in different flavors:
    - batch normalization or batchnorm (the most common technique)
    - group normalization or groupnorm
    - there are more possibilities, for additional info on these, please check this lecture by Aaron Defazio, NYU
3. Dropout, a technique patented by Google which consists in randomly *dropping* some neurons from a given layer to prevent overfitting.
4. Pruning (as a side consequence)
5. Early stopping, which we'll see later on during this Lab.

Weight decay or L2 norm/Ridge regularization

Weight Decay (WD) is a simple technique which *appends* a penalty term to the loss function equation. The term is based upon the L2 norm of the weights.

Given our original loss function $\mathcal{L}_0(\hat{y}, y)$ and our parameter vector Θ, our new loss will be:

$\mathcal{L}_0(\hat{y}, y) + \frac{1}{2} \cdot \lambda \cdot ||\Theta||_2^2$

the parameter λ (also called weight decay) controls the strenght of the regularization. λ too high means that the model will not concentrate well enough on the original objective ($\mathcal{L}_0$), hence it will not perform well. Usually, good values form λ fall into the interval $[5 \cdot 10^{-4}, 1 \cdot 10^{-4}]$.

In PT, instead of inserting our penalty term in the loss function, we specify the weight decay parameter in our optimizer. From the lecture, we have learnt that the penalty term added to the loss can equivalently be used as an update term in SGD, which lets us use the regular "unpenalized" cross entropy loss.

In [5]:
```
weight_decay = 5e-4 # lambda
optimizer = torch.optim.SGD(model.parameters(), lr=learn_rate, weight_decay=weight_decay) #called directly inside t
```

L1 norm regularization

L1 norm regularization is analogous to weight decay. The equation is:

$\mathcal{L}_0(\hat{y}, y) + \lambda \cdot ||\Theta||_1$

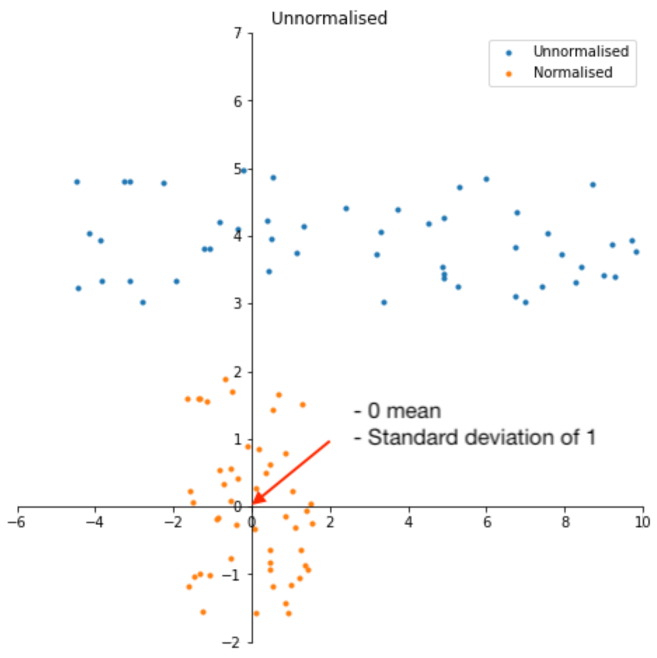
where $||x||_1 = \sum_{j=1}^{d} |x_j|$

unlike weight decay, to my knowledge PT does not provide a built-in for L1 reg. You need to define a custom loss function for this task (**homework**).

batchnorm

Batch Normalization is not really a regularization technique. It operates in such a way that the mean and standard deviation of the incoming batches of data are approximately 0 and 1 respectively.

The ultimate goal of batchnorm is not to normalize each batch, but estimate one vector for mean (a running mean) and one for std (a running std) for the whole dataset and to normalize w.r.t. these. So, these become new parameters of the network. They are not adjusted via backprop but they get adjusted each time the layer *sees* another batch of data.



picture from [towardsdatascience.com]

When the network is evaluated on test data, the running mean and std must not be adjusted, hence PT has implemented a "switch", which we saw during the previous Lab, to tell the network when to adjust and not adjust these two parameters. The switch is triggered via `model.train()` and `model.eval()` (or equivalently `model.train(False)`).

In PT, the batch normalization is found as a regular layer under within the `torch.nn` library

In [6]:
```
class MLP_BN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = torch.nn.Sequential(
            torch.nn.Flatten(),
            torch.nn.Linear(28*28, 16), # 16 activations values to be normalized
            torch.nn.ReLU(),

            torch.nn.BatchNorm1d(num_features=16), # we specify the dimensionality of the incoming data
            # BatchNorm2d we will see in convolutional neural network: when output is more than 1 dim
            # BatchNorm1d: output 1 dim
            torch.nn.Linear(16, 32),
            torch.nn.ReLU(),

            torch.nn.BatchNorm1d(num_features=32),
            torch.nn.Linear(32, 24),
            torch.nn.ReLU(),

            torch.nn.BatchNorm1d(num_features=24),
            torch.nn.Linear(24, 10)
        )


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

Q (for the most skilled students): why didn't we apply batchnorm for the first layer?

By peeking at the PT docs, we can see that actually the batchnorm layers have much more hyperparameters which we can play with if we wanted to:

from PyTorch docs

In addition to what we say till now, there is still some debate in the DL community on whether batchnorm or other normalization techniques help optimization. The claims in the original paper 1 of "reducing internal covariate shift" was confuted in successive works such as 2, which claims that it "makes the optimization landscape significantly smoother". Another things to consider is that, since the data is distributed in a small intervall around 0, there's also a better numerical stability added.

Dropout

Dropout acts by removing (i.e. *zeroing-out*) a random subset of the neurons in a given layer for each forward pass.

It has one hyperparameter (p), which is the fraction of neurons to be dropped out.

During training, each time a layer with backprop produces an output, a fraction p of that output gets discarded. This helps in such a way that co-dependence between neurons gets *forgotten* by the network. To say it in simple terms, it forces each neuron to be independent from the output of other neurons within the same layer.

For the same reason as in batchnorm, since dropout has to apply only during training, we must be careful in activating the switch `model.eval()` when testing our network.

In PT, we find Dropout as a module of `torch.nn`. Instead of placing if *before* the layer (as in batchnorm), we place it *after* the layer (the reason being, the layer produces an output, a portion p of that output gets discarded).

In [7]:
```
class MLP_BN_Drop(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = torch.nn.Sequential(
            torch.nn.Flatten(),
            torch.nn.Linear(28*28, 16),
            torch.nn.ReLU(),

            torch.nn.BatchNorm1d(num_features=16),
            torch.nn.Linear(16, 32),
            torch.nn.ReLU(),
            torch.nn.Dropout(p=.2), # we add a dropout here. it's referred to the previous layer (with 32 neurons)
            # Dropout AFTER the layer!! p is the probability


            torch.nn.BatchNorm1d(num_features=32),
            torch.nn.Linear(32, 24),
            torch.nn.ReLU(),

            torch.nn.BatchNorm1d(num_features=24),
            torch.nn.Linear(24, 10)
        )


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

In [8]:
```
model = MLP_BN_Drop()
device = "cpu"
model.to(device)
weight_decay = 5e-4
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learn_rate, weight_decay=weight_decay, momentum=momentum)
```

In [9]:
```
train_loss, train_acc = train.train_model(model, trainloader, loss_fn, optimizer, num_epochs, device=device)
```

```
Epoch 1 completed. Loss - total: 29362.426805496216 - average: 0.4893737800916036; Performance: 0.8538833333333333
Epoch 2 completed. Loss - total: 14867.405965805054 - average: 0.24779009943008423; Performance: 0.9269666666666667
Epoch 3 completed. Loss - total: 13117.804376125336 - average: 0.21863007293542225; Performance: 0.9353166666666667
Epoch 4 completed. Loss - total: 11872.433501243591 - average: 0.1978738916873932; Performance: 0.94225
Epoch 5 completed. Loss - total: 11469.728811264038 - average: 0.19116214685440064; Performance: 0.9426666666666667
```

Unstructured pruning

Pruning removes parameter w.r.t. a given criterion.

Usually the criterion is the magnitude:

1. Define pruning rate $p \in (0, 1)$
2. Stack the magnitude of the parameters of the MLP in a single vector Θ
3. Order Θ and get its p-th quantile $\tilde{\theta}_p$
4. For each parameter θ_i in the MLP, remove it if $\theta_i < \tilde{\theta}_p$

Since the unstructured removal of parameters is difficult to deal computationally, the process of removing is instead surrogated by the institution of a **pruning mask** M, which can be thought of as a data structure having the same shape as the parameters, but with boolean entries.

The point 4. gets replaced by:

- Create pruning mask M: $m_i = 0$ if $\theta_i < \tilde{\theta}_p;\ 1$ otherwise
- Zero-out the pruned parameters: $\theta_i^{\text{new}} = \theta_i \cdot m_i$
- After backprop, zero-out the gradients of pruned parameters $\nabla \mathcal{L}_{\theta_i}(X)^{\text{new}} = \nabla \mathcal{L}_{\theta_i}(X) \cdot m_i$

In [10]:
```python
x = torch.Tensor([6,5,4,3,2,1])
x.kthvalue(3) # returns values and indices without sorting the data
```

Out[10]:
```
torch.return_types.kthvalue(
values=tensor(3.),
indices=tensor(3))
```

In [ ]:
```python
def prune(model, p):
    params_magnitude = torch.cat([par.abs().flatten() for i, par in model.parameters()]) #tensor
    # it stacks together in a single vector (point 2): list of vectors which I concatenate together
    # in a single vector (.flatten())
    index_of_p = int(len(params_magnitude) * p)
    quantile = params_magnitude.kthvalue(index_of_p).values #scalar
    # mask = (params_magnitude < quantile) # tensor of booleans: compares a (tensor) with a (scalar) and assign it
    # but like this, the mask is flatten but param not, so:
    mask = [par.abs()>quantile for par in model.parameters()] #list comprehension
    for params, m in zip(model.parameters(), mask) # zip: creates tuples of iterables
        params.value *= m
```

a few notes on this implementation:

1. this method prunes everything, even parametric layers which should not be pruned (e.g., batchnorm). Solutions:
   - Keep track of the index of the layers to prune and apply pruning only to them (e.g. params_magnitude = torch.cat([par.abs().flatten() for i, par in enumerate(model.parameters())]) etc..), or
   - Use `model.named_parameters()` instead of `model.parameters()` and apply pruning only to specific parameters names
2. after `backward()`, call `params.grad *= m` as in the last line of `prune` to zero-out the gradient
3. this implementation does not allow for iterative pruning, as the parameters pruned before contribute to the formation of `params_magnitude` and hence to the calculation of the quantile. how can we enhance this implementation to allow for iterative pruning?

In [ ]:
```python
# To iteratevly prune I must update the mask:
params_magnitude = torch.cat([par.abs()[m == 1] for par, m in zip(model.parameters(), mask)]) #mask is a tensor of
# Oss. flattening is done in place!
```

HOMEWORK - Early stopping

Early stopping is yet another example of regularization technique which relies a lot on practical and experimental observations rather than any supporting theory.

It is based upon the concept of **validation**, which is an assessment mode **additional** to *testing*. Actually, what insofar whe have described as testing is technically a validation.

- a validation dataset may be obtained as result of a random splitting of the original training dataset
- a testing dataset should be obtained instead from a model deployed "in the wild" and should consist of data unseen (from both the model and its architect) during the training and designing phase.

In a normal academic setting it's very hard to obtain a proper testing dataset, so usually the meaning of testing and validation get mixed up a little bit.

Anyway, early stopping requires us to assess the model at each epoch to get a proxy for the testing performance(s) (**validation step**). That should gives us an idea of how the model **learns to generalize** (if it ever does...) during training.

The *theoretical trend* ('90 s), which is pretty much absent in modern Deep Learning due to a lot of modern factors, is the following (figures from 4):

Already in that period, different stuff was observed:

In some of my experiments, this happens (blue=training, orange=validation):

(red=training, blue=validation)

As we observe the curves for training and validation performance, we may notice some trends:

- there usually is an intersection between the two curves which marks the moment when the network starts to **overfit** the training data.
    - it might happen that, after that moment, the validation performance stays roughly the same (*white noise*), or that it drops and never recovers
    - it might happen, instead, that the validation performance stays a few points below the training performance but keeps on growing
    - it might happen, eventually, that the validation performance peaks a few epochs after and then it decreases
    - other situations may apply depending on the dataset, the optimizer, the presence of regularization, and a lot of other factors.

A trick which is very often applied is to track the validation performance during training and retain the model with the highest validation performance. **Note**: it may not be the best strategy as the validation dataset may not be representative of the data manifold (!). In the main reference for early stopping (4), it is indicated as E_{opt}.

Homework: implement "early stopping" in the E_{opt} using the test data as validation (since we don't know yet how to create additional `DataLoaders` and operate random splitting). *Suggestion*: try training for more than 5 epochs, maybe 20-30 total would be fine. Use Colab GPUs in case you want to accelerate training.

Homework for the bravest ones: read 4 and try implementing at least one of the techniques there specified (besides E_{opt}, of course).

References

1 Ioffe, S., & Szegedy, C. (2015, June). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International conference on machine learning (pp. 448-456). PMLR.

2 Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization?. arXiv preprint arXiv:1805.11604.

3 Frankle, J., Schwab, D. J., & Morcos, A. S. (2020). The early phase of neural network training. arXiv preprint arXiv:2002.10365.

4 Prechelt, L. (1998). Early stopping-but when?. In Neural Networks: Tricks of the trade (pp. 55-69). Springer, Berlin, Heidelberg.

Homework recap - EASTER HOMEWORK (optional)

1. Implement L1 norm regularization as a custom loss function
2. Implement L1 norm regularization as a custom optimizer
3. Implement early stopping in the E_{opt} specification
4. Implement early stopping in one of the additional specifications as of 4
5. Reconstruct in PyTorch the first experiment in Learning representations by back-propagating errors with learning rule in eq.8 (gradient descent without momentum)
    - Try to be as close as possible to the original protocol, except for what regards the learning rule
    - Read the paper, if you did not do it yet (don't worry if you don't understand the other experiments in detail)
    - Create the data, the model and everything is needed (do not use dataloaders if you don't know how yet how they work)
    - Train the model
    - Inspect the weights you obtained and check if they provide a solution to the problem
    - Compare the solution to the solution reported in the paper
    - Do not get frustrated if the results are not as expected!