Hands-on Introduction to Deep Learning with PyTorch

More on Training Deep Learning Models

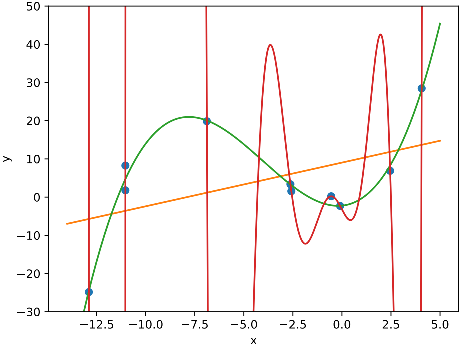
Rocco Meli and Rafael Sarmiento
ETHZürich / CSCS

Lugano, February 28th - March 1st 2024

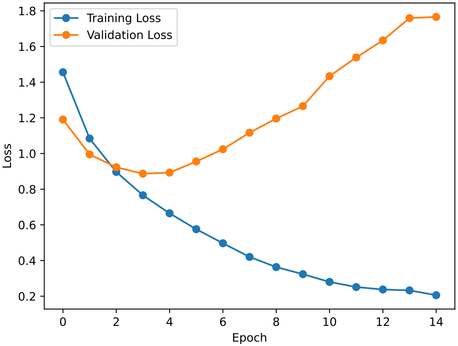
ETH zürich
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Overfitting

Overfitting happens when the model is too complex (compared to the noisiness of the training data)



Overfitting



- Low training loss
- High validation loss
- Large gap: generalization error
- The model performed better early on

Early Stopping

Stop training when the validation error reaches a minimum.

An trivial implementation of early stopping is to save the best model.

PyTorch: State Dictionary

- Learnable parameters (weights, biases) are stored in `torch.nn.Module`'s parameters
  - `module.parameters()`
- A state dictionary maps each layer of the model to the corresponding parameter
  - `model.state_dict()`
- Optimizers (`torch.optim`) also have a state dictionary
  - Useful to resume training

PyTorch: Save and Load Parameters to File

```
model = ...
optimizer = ...

torch.save(
  model.state_dict(), PATH
)
torch.save(
  optimizer.state_dict(), PATH
)
```

```
model = ...
optimizer = ...

model.load_state_dict(
  torch.load(PATH)
)
optimizer.load_state_dict(
  torch.load(PATH)
)
```

Convention: use `.pt` or `.pth` file extensions.

PyTorch: Save and Load Parameters

- `state_dict()` returns a reference to the state dictionary
- Use `deepcopy` to copy the state dictionary

```
model = ...

best_model_state = deepcopy(model.state_dict())
```

Regularisation: Loss Funtion

Add constraints on the model (model weights) via the loss function

PyTorch: L_2 Regularization

$$L_2 = \lambda \sum_i w_i^2$$

L_2 loss is quite common, and easily available in PyTorch via the `weight_decay` parameter in various optimizers (`SGD`, `Adam`, ...)

PyTorch: L_1 Regularization

L_1 regularization often produces a sparse model (many weights close to or equal to 0)

$$L_1 = \lambda \sum_i |w_i|$$

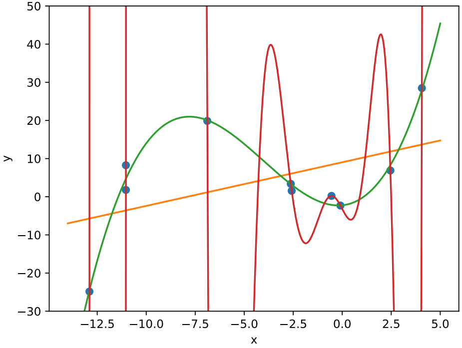
```python
def l1_loss(params, lambda):
    l1 = 0
    for p in params:
      l1 += torch.sum(torch.abs(p))

    return lambda * l1

total_loss = loss + l1_loss(model.parameters(), lambda = 0.01)
total_loss.backward()
```

Regularization: Reduce Model Capacity

Reducing the model capacity (number of parameters) can reduce over-fitting.




```python
# Compute number of parameters

num_elements_list = [
  p.numel()
  for p in model.parameters()
  if p.requires_grad
]

sum(num_elements_list)
```

Regularization: Dropout Layers

- Randomly zero some elements of the input with probability p
  - Each channel zeroed out independently
- Outputs scaled by $(1-p)^{-1}$
- Prevents co-adaptation of neurons

Monte Carlo Dropout for Error Estimation

- Perform multiple predictions with dropouts
  - Only have dropout layers in `train()` mode
- Average predictions
- Standard deviation is also available

```
y_all = np.stack([
  model(x).cpu().numpy()
  for i in range(n)
])

y = np.mean(y_all, axis=0)
```

```
def enable_dropout(model):
  for m in model.modules():
    mname = m.__class__.__name__
    if mname.startswith('Dropout'):
      m.train()
```

Monte Carlo Dropout for Error Estimation

- Perform multiple predictions with dropouts
  - Only have dropout layers in `train()` mode
- Average predictions
- Standard deviation is also available

```
y_all = np.stack([
  model(x).cpu().numpy()
  for i in range(n)
])

y = np.mean(y_all, axis=0)
```

```
def enable_dropout(model):
  for m in model.modules():
    mname = m.__class__.__name__
    if mname.startswith('Dropout'):
      m.train()
```

If you finish the exercises early, try to implement Monte Carlo Dropout

Data Augmentation

Data augmentation artificially inflates the size of the training set by transforming the data (on-the-fly) at each iteration

- Random rotations
- Random translations
- Random re-sizing and clipping
- Random flips
- ...

Data augmentation helps reducing overfitting

PyTorch: Data Augmentation with torchvision's transforms (V1)

```python
from torchvision import transforms

transform = transforms.Compose([
  transforms.ToTensor(), # Deprecated in V2
  transforms.RandomResizedCrop(),
  transforms.RandomHorizontalFlip(),
  # ...
  transforms.Normalize(
    mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
  ),
])
```

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

Adaptive Learning Rate

- A fixed learning rate might not be optimal for the whole training
- The learning rate can be adjusted during training
  - Based on the number of epochs
  - Based on some validation metrics (`ReduceLROnPlateau`)

PyTorch: Learning Rate Scheduler

PyTorch's `torch.optim.lr_scheduler` provides several methods/policies to adjust the learning rate.

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = ExponentialLR(optimizer, gamma=0.9)

for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step()
```

https://pytorch.org/docs/stable/optim.html

Optimizers

SDG takes small, regular steps

$$\theta_t = \theta_{t-1} - \eta \nabla_{\theta_{t-1}} \mathcal{L}(\theta_{t-1})$$

Momentum optimization:

$$\mathbf{m}_t = -\eta \nabla_{\theta_{t-1}} \mathcal{L}(\theta_{t-1}) - \beta \mathbf{m}_{t-1}$$

$$\theta_t = \theta_{t-1} + \mathbf{m}_t$$

Weights Initialisation

- Weights are randomly initialized
  - Uniform
  - Normal
  - ...
- Initialization has an impact on training
- Sensible choices of random distribution by default

https://xkcd.com/1838

PyTorch: Weights Initialisation

```python
def custom_weights_init(module):
    if isinstance(module, nn.Linear):
        module.weight.data.normal_(mean=0.0, std=1.0)
        if module.bias is not None:
          module.bias.data.zero_()

    if isinstance(module, nn.Conv2D):
        # Use torch.nn.init module
        torch.nn.init.xavier_uniform_(module.weight.data)

model.apply(custom_weights_init)
```

Transfer Learning and Fine Tuning: Motivation

- Training a deep learning model can be very expensive
- Many tasks are closely related
- Some data sets are too small to train a performant deep learning model
- Random initialization is far from optimal

Transfer learning and fine tuning aim at re-using pre-trained models and fine tune them for the specific task at hand.

Transfer Learning and Fine Tuning: CNN Example

CNNs for classification have two conceptual building blocks:

- Feature extractor (convolution layers)
- Classifiers (linear layers)

Feature extraction layers can be re-used from successfully trained models, while the classifier can be re-trained for the task at hand.

PyTorch: Load TorchVision Pre-Trained Models

```python
from torchvision.models import resnet50, ResNet50_Weights

resnet50(weights=ResNet50_Weights.IMAGENET1K_V2)

resnet50(weights=ResNet50_Weights.DEFAULT)

resnet50(weights="IMAGENET1K_V2")

# No weights (random initialization)
resnet50(weights=None)
```

PyTorch Hub

PyTorch Hub supports publishing pre-trained models!

```
# hubconf.py (on GitHub)
dependencies = ['torch']

def my_model(
  pretrained=False,
  **kwargs
  ):
  model = MyModel()
  if pretrained:
     ptw = 'https://url.com/w.pth'
     model.load_state_dict(
      torch.hub.load_state_dict_from_url(ptw)
    )
  return model
```

```
repo = ...
model = torch.hub.load(
  repo,
  my_model,
  pretrained=True
)
```

PyTorch: Freeze Model Parameters

```python
# Do not compute gradients in backward pass
for param in model_conv.parameters():
  param.requires_grad = False
```

PyTorch: Overwrite Pre-Trained Layers

```
model = nn.Sequential(
  nn.Linear(784, 256),
  nn.ReLU(),
  nn.Linear(256, 64),
  nn.ReLU(),
  nn.Linear(64, 10)
)

print(model)
# Sequential(
#   (0): Linear(...)
#   (1): ReLU()
#   (2): Linear(...)
#   (3): ReLU()
#   (4): Linear(...)
# )
```

```
for param in model.parameters()[:-1]:
  param.requires_grad = False

n_in_features = model[-1].in_features

# Swap last linear layer
# with another (untrained) one
model[-1] = nn.Linear(n_in_features, 5)
```

Hyperparameter Tuning

Hyper-parameters can have a huge impact on the model, and the hyperparameter space is large.

There are several libraries for hyperparameter tuning:

- Ray Tune (`https://docs.ray.io`)
- Optuna (`https://optuna.org`)
- Hyperopt (`http://hyperopt.github.io/hyperopt/`)
- ...

High-Level Libraries for PyTorch

PyTorch is rather bare-bones. There are many library built on top of it which require to write much less boilerplate code (training loop, builtin metrics, learning rate scheduling, ...):

- PyTorch Ignite
- FastAI
- Keras 3.0 (PyTorch, TensorFlow, JAX)
- ...

PyTorch Ignite Example

```python
from ignite.engine import create_supervised_trainer
...
trainer = create_supervised_trainer(model, optimizer, loss_fn, device)
trainer.run(tran_loader, max_epochs=5)
```

PyTorch Ignite Example

```
from ignite.engine import Engine
...
def train_step(engine, batch):
    model.train()
    optimizer.zero_grad()
    x, y = batch[0].to(device), batch[1].to(device)
    y_pred = model(x)
    loss = criterion(y_pred, y)
    loss.backward()
    optimizer.step()
    return loss.item()

trainer = Engine(train_step)
trainer.run(tran_loader, max_epochs=5)
```

Keras 3.0 Example

```
model = keras.Sequential([
  keras.Input(shape=input_shape),
  layers.Conv2D(32,
    kernel_size=(3, 3),
    activation="relu"
  ),
  layers.MaxPooling2D(
    pool_size=(2, 2)
  ),
  ...
  layers.Flatten(),
  layers.Dense(
    num_classes,
    activation="softmax"
  )])
```

```
model.compile(
  loss="categorical_crossentropy",
  optimizer="adam",
  metrics=["accuracy"]
)

model.fit(
  x_train, y_train,
  batch_size=batch_size,
  epochs=epochs,
  validation_split=0.1
)

score = model.evaluate(
  x_test, y_test, verbose=0
)
```

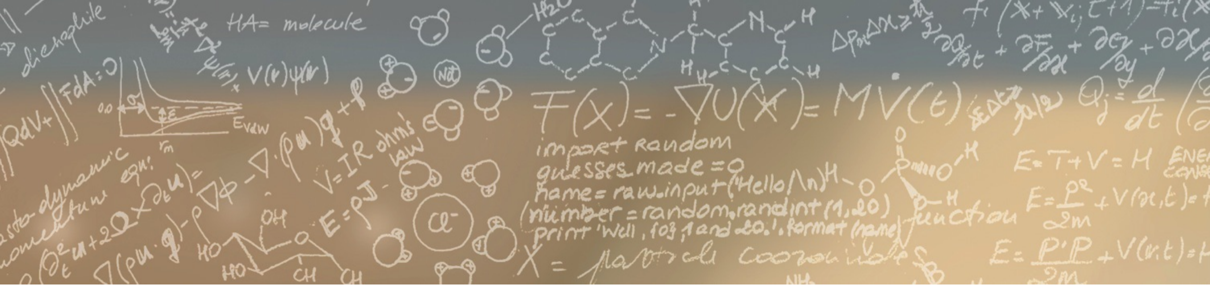
Exercises

Regularization and data augmentation:

- Train a CNN with and without dropout layers
- Train a CNN with and without data augmentation
- Save and load model weights (early stopping)

Transfer learning:

- Modify VGG-19 to work with only 5 classes
- Freeze retained VGG-19 parameters
- Apply transfer learning to the new data



Thank you for your attention!

Why the Validation Loss is Lower than the Traning Loss?

- Regularization is applied during training but not inference
- Training loss is computed during each epoch, validation loss after each epoch
  - Shift loss of $1/2$ epoch
- Validation set might be easier than the training set
- Training set leaked into the validation set