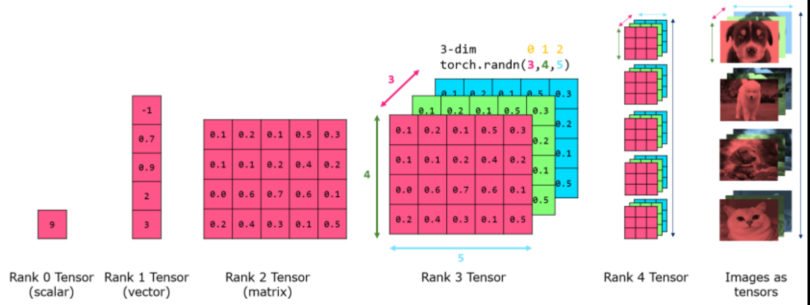
PyTorch CHEAT SHEET

Imports

```
import torch                                        # root package
from torch.utils.data import Dataset, DataLoader    # dataset representation and loading
```




```
# N-dim of tensor = number of square brackets [ on the left side
# [batch_size, hight, width, color_channels] NHWC # images alternative 1
# [batch_size, color_channels, hight, width] NCHW # images alternative 2
```

Tensors Creation

```
x = torch.tensor(data,dtype,device) # tensor with no autograd history
x = torch.arange(start,end,step)    # seqeuence
x = torch.randn(*size)              # tensor of random numbers from normal distribution N(0,1)
x = torch.[ones|zeros](*size)       # tensor with all 1's [or 0's]
x = torch.tensor(L)                 # tensor from [nested] list or ndarray L
x = torch.from_numpy(numpy_arr)     # tensor from a NumPy array
y = x.clone()                       # clone of x
with torch.no_grad():               # code wrap that stops autograd from tracking tensor history
requires_grad=True                  # arg, when set to True, tracks computation
                                    # history for future derivative calculations
```

Dimensionality

```
x.size()                            # return tuple-like object of dimensions
x = torch.cat(tensors,dim=0)        # concatenates tensors along dim WITHOUT changing the dim of
                                    # tensors
y = torch.stack(tensors,dim=0)      # stacks a sequence of tensors along a NEW dimension
y = x.view(a,b,...)                 # reshapes x into size (a,b,...)
y = x.view(-1,a)                    # reshapes x into size (b,a) for some b
y = x.transpose(a,b)                # swaps dimensions a and b
y = x.permute(*dims)                # Returns a view of the original input with its dimensions
                                    # permuted (rearranged) to dims
y = x.unsqueeze(dim)                # tensor with added axis
y = x.unsqueeze(dim=2)              # (a,b,c) tensor -> (a,b,1,c) tensor
y = x.squeeze()                     # removes all dimensions of size 1 (a,1,b,1) -> (a,b)
y = x.squeeze(dim=1)                # removes specified dimension of size 1 (a,1,b,1) -> (a,b,1)
y = x.reshape(shape)                # Reshapes input to shape (if compatible)
```

Math

```
# Algebra
ret = A * B         # element-wise multiplication
ret = A.mm(B)       # matrix multiplication / dot product
ret = A.mv(x)       # matrix-vector multiplication
x = x.t()           # matrix transpose

torch.abs(tensor)
torch.add(tensor, tensor2) # or tensor+scalar
torch.div(tensor, tensor2) # or tensor/scalar
torch.mult(tensor,tensor2) # or tensor*scalar
torch.sub(tensor, tensor2) # or tensor-scalar
torch.ceil(tensor)
torch.floor(tensor)
torch.remainder(tensor, devisor) #or torch.fmod()
torch.sqrt(tensor)
```

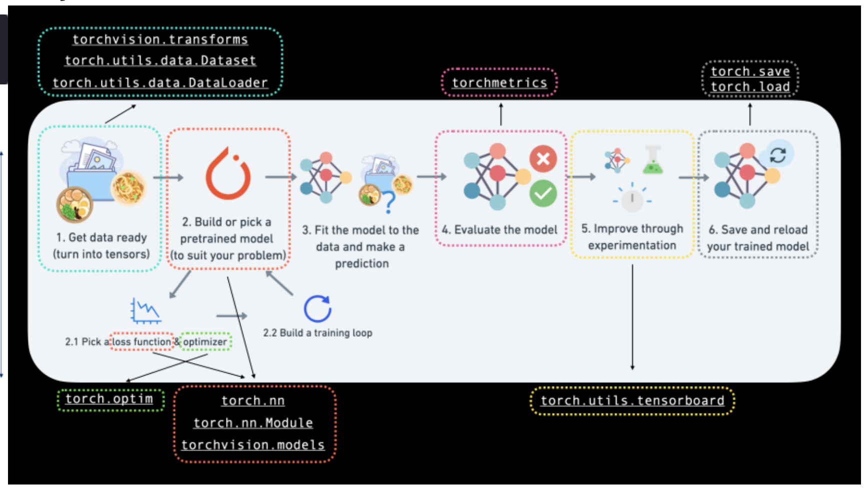
Torchscript and JIT

```
torch.jit.trace()      # takes your module or function and an example
                       # data input, and traces the computational steps
                       # that the data encounters as it progresses through the model

@script                # decorator used to indicate data-dependent
                       # control flow within the code being traced
```

ONNX

```
torch.onnx.export(model, dummy data, xxxx.proto)     # exports an ONNX formatted
                                                     # model using a trained model, dummy
                                                     # data and the desired file name

model = onnx.load("alexnet.proto")                   # load an ONNX model
onnx.checker.check_model(model)                      # check that the model
                                                     # IR is well formed

onnx.helper.printable_graph(model.graph)             # print a human readable
                                                     # representation of the graph
```

PyTorch Workflow



Data Utilities

```
Dataset                                 # abstract class representing dataset
TensorDataset                           # labelled dataset in the form of tensors
Concat Dataset                          # concatenation of Datasets

DataLoader(dataset, batch_size=1, ...)  # loads data batches agnostic
                                        # of structure of individual data points

sampler.Sampler(dataset,...)            # abstract class dealing with
                                        # ways to sample from dataset

sampler.XSampler where ...              # Sequential, Random, SubsetRandom,
                                        # WeightedRandom, Batch, Distributed
```

Vision

```
# Base computer vision library
import torchvision

# Other components of TorchVision (premade datasets, pretrained models and image transforms)
from torchvision import datasets, models, transforms
```

Text

```
# Base text and natural language processing library
import torchtext

# Other components of TorchText (premade datasets, pretrained models and text transforms)
from torchtext import datasets, models, transforms
```

Audio and Speech

```
# Base audio and speech processing library
import torchaudio

# Other components of TorchAudio (premade datasets, pretrained models and text transforms)
from torchaudio import datasets, models, transforms
```

Recommendation systems

```
# Base recommendation system library
import torchrec

# Other components of TorchRec
from torchrec import datasets, models
```

GPU Usage

```
torch.cuda.is_available                      # check for cuda
x = x.cuda()                                 # move x's data from CPU to GPU
                                             # and return new object

x = x.cpu()                                  # move x's data from GPU to CPU
                                             # and return new object

if torch.cuda.is_available():                 # Setup device-agnostic code
    device = "cuda"                           # NVIDIA GPU
elif torch.backends.mps_is_available():
    device = "mps"                            # Apple GPU
else:
    device = "cpu"


net.to(device)                # recursively convert their parameters
                              # and buffers to device specific tensors

x = x.to(device)              # copy your tensors to a device (gpu, cpu)
```

Neural Network API

```
import torch.autograd as autograd       # computation graph
from torch import Tensor                # tensor node in the computation graph
import torch.nn as nn                   # neural networks
import torch.nn.functional as F         # layers, activations and more
import torch.optim as optim             # optimizers e.g. gradient descent, ADAM, etc.
from torch.jit import script, trace     # hybrid frontend decorator and tracing jit
```

Deep Learning

```
nn.Linear(m,n)                                 # fully connected layer from
                                               # m to n units

nn.ConvXd(m,n,s)                               # X dimensional conv layer from
                                               # m to n channels where Xϵ{1,2,3}
                                               # and the kernel size is s

nn.MaxPoolXd(s)                                # X dimension pooling layer
                                               # (notation as above)

nn.BatchNormXd                                 # batch norm layer
nn.RNN/LSTM/GRU                                # recurrent layers
nn.Dropout(p=0.5, inplace=False)               # dropout layer for any dimensional input
nn.Dropout2d(p=0.5, inplace=False)             # 2-dimensional channel-wise dropout
nn.Embedding(num_embeddings, embedding_dim)    # (tensor-wise) mapping from
                                               # indices to embedding vectors
```

Model Building

```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

model = NeuralNet(input_size, hidden_size, num_classes).to(device)
```

Loss Functions

```
nn.X                          # where X is L1Loss, MSELoss, CrossEntropyLoss
                              # CTCLoss, NLLLoss, PoissonNLLLoss,
                              # KLDivLoss, BCELoss, BCEWithLogitsLoss,
                              # MarginRankingLoss, HingeEmbeddingLoss,
                              # MultiLabelMarginLoss, SmoothL1Loss,
                              # SoftMarginLoss, MultiLabelSoftMarginLoss,
                              # CosineEmbeddingLoss, MultiMarginLoss,
                              # or TripletMarginLoss
```

Activation Functions

```
nn.X                          # where X is ReLU, ReLU6, ELU, SELU, PReLU, LeakyReLU,
                              # RReLu, CELU, GELU, Threshold, Hardshrink, HardTanh,
                              # Sigmoid, LogSigmoid, Softplus, SoftShrink,
                              # Softsign, Tanh, TanhShrink, Softmin, Softmax,
                              # Softmax2d, LogSoftmax or AdaptiveSoftmaxWithLoss
```

Optimizers

```
opt = optim.x(model.parameters(), ...)      # create optimizer
opt.step()                                  # update weights
optim.X                                     # where X is SGD, Adadelta, Adagrad, Adam,
                                            # AdamW, SparseAdam, Adamax, ASGD,
                                            # LBFGS, RMSprop or Rprop
```

Learning rate scheduling

```
scheduler = optim.X(optimizer,...)      # create lr scheduler
scheduler.step()                        # update lr after optimizer updates weights
optim.lr_scheduler.X                    # where X is LambdaLR, MultiplicativeLR,
                                        # StepLR, MultiStepLR, ExponentialLR,
                                        # CosineAnnealingLR, ReduceLROnPlateau, CyclicLR,
                                        # OneCycleLR, CosineAnnealingWarmRestarts,
```

Distributed Training

```
import torch.distributed as dist                # distributed communication
from torch.multiprocessing import Process       # memory sharing processes
```


Ahmad Alismail, linkedin.com/in/ahmadalismail1

Last updated: 11/2022

PyTorch Workflow: Daniel Bourke https://github.com/mrdbourke/pytorch-deep-learning

Get data ready

- What shapes are my inputs and what shapes are my outputs? (All samples vs. Single sample)
- PyTorch prefers to work with PyTorch tensors - > Turn data into tensors
- Split our data into training and test sets

Load our data into PyTorch

- Turn int into tensors (e.g., `torchvision.transforms`)
- turn our raw images into `Dataset`'s (features mapped to labels or X's mapped to y's) through `torch.utils.data.Dataset`
- Turn our `Dataset` subsequently into `DataLoader` through `torch.utils.data.DataLoader`

`DataLoader`

- `DataLoader` combines dataset and sampler
- It helps load data into a model. For training and for inference.
- It turns a large **Dataset** into a Python `iterable` of smaller chunks.
- These smaller chunks are called batches or mini-batches and can be set by the `batch_size` parameter.
- Why do this?
- Because it's more computationally efficient.

Going Modular

- One way to go from notebooks to scripts
- Train a model built in a jupyter notebook with one line of code on the command line `!python train.py`
- Create folder to store scrpits using `os.makedirs()`
- Create the script using `%%writefile folder_name/file_name.py`

Build or pick a pretrained Model

- Setting up device agnostic code (so our model can run on CPU or GPU if it's available)
- Costructing a model by subclassing nn.Module
- Create layers capable of handling X and y (considereing input and output shapes)
- Define a forward method (make use of operator fusion to improve performance on GPU: `return self.layer_3(self.relu(self.layer_2((x)))`
- Create an instance of the model and send it to target device
- Create a loss function (a.k.a loss criterion)
- Create an optimizer

PyTorch Training Loop

- Put data on the available device `X_train.to(device)` (without this error will happen!)
- Loop through epochs
  - Loop through training batches, perform **training steps**, caculate loss per batch.
- Put model in training mode
- **Forward pass** - The model goes through all of the training data once, performing its `forward()` function calculations (`model(x_train)`)
- **Calculate the loss** - The model's output/predictions (logits for classification if the loss function has a layer to convert logits to probabilities) are compared to the ground truth (labels) and evaluated to see how wrong they are (`loss = loss_fn(y_pred, y_train)`).
- **Zero gradients** - The optimizers gradients are set to zero (they are accumulated by default) so they can be recalculated for the specific training step (`optimizer.zero_grad()`).
- **Perform backpropagation on the loss** - Computes the gradient of the loss with respect for every model parameter to be updated (each parameter with `requires_grad=True`). This is known as backpropagation, hence "backwards" (`loss.backward()`).
- **Step the optimizer (gradient descent)** - Update the parameters with `requires_grad=True` with respect to the loss gradients in order to improve them (`optimizer.step()`).

PyTorch Testing Loop

- Put the model in evaluation mode
- Turn on `torch.inference_mode()` context manager to disable functionality such as gradient tracking for inference (since gradient trackingis not needed for inference)
- All predictions should be made with objects on the same device (e.g. data and model on **GPU** only or data and model on **CPU** only), i.e., `model.to(device)` and `X_test = X_test.to(device)`
- Pass the test data through the model. The model will generate logits (the raw outputs). Convert **logits** -> prediction **probabilities** (with `sigmoid/softmax`) -> predictions **labels** (with `argmax(dim=1)`)
- NOTE: call `y_pred.cpu()` on your target tensor to return a copy of your target tensor on the CPU because some libraries aren't capable of using data that is stored on GPU

Top PyTorch and DL Errors

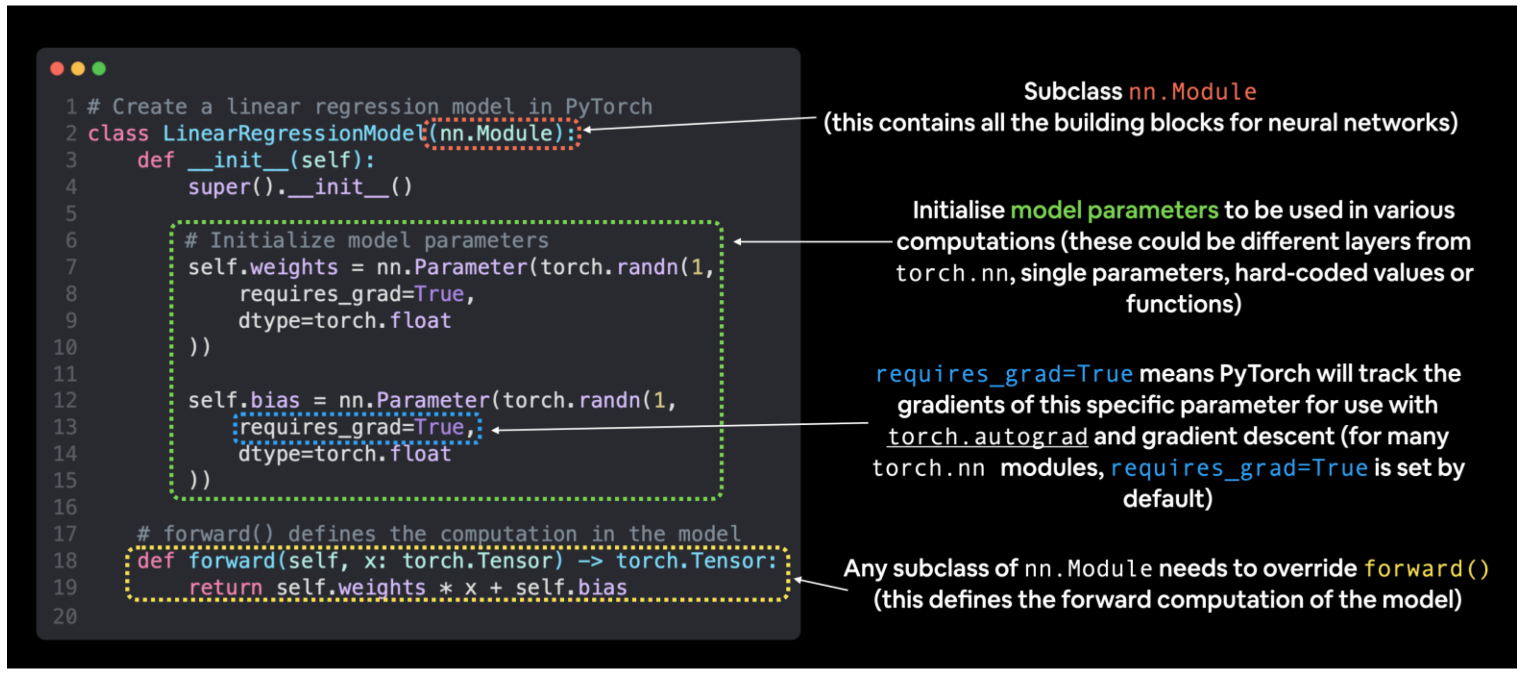
- **Wrong datatypes**: Your Model expected `torch.float32` when your data is `torch.uint8`
- **Wrong data shapes**: Your model expected `[batch_size, color_channels, height, width]` when your data is `[color_channels, height, width]`
- **Wrong devices:** Your model is on the GPU but your data is on the CPU.

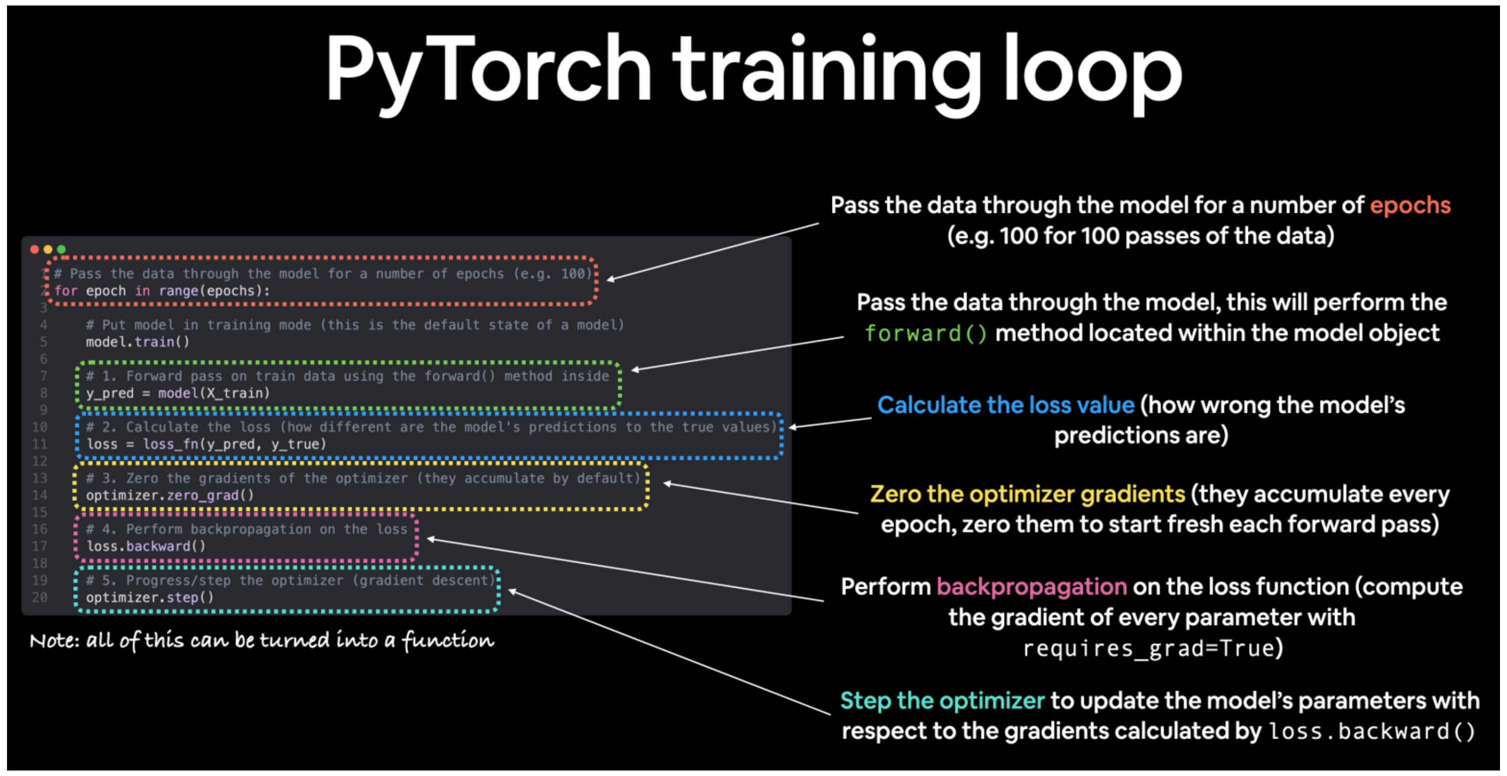
Prevent Overfitting

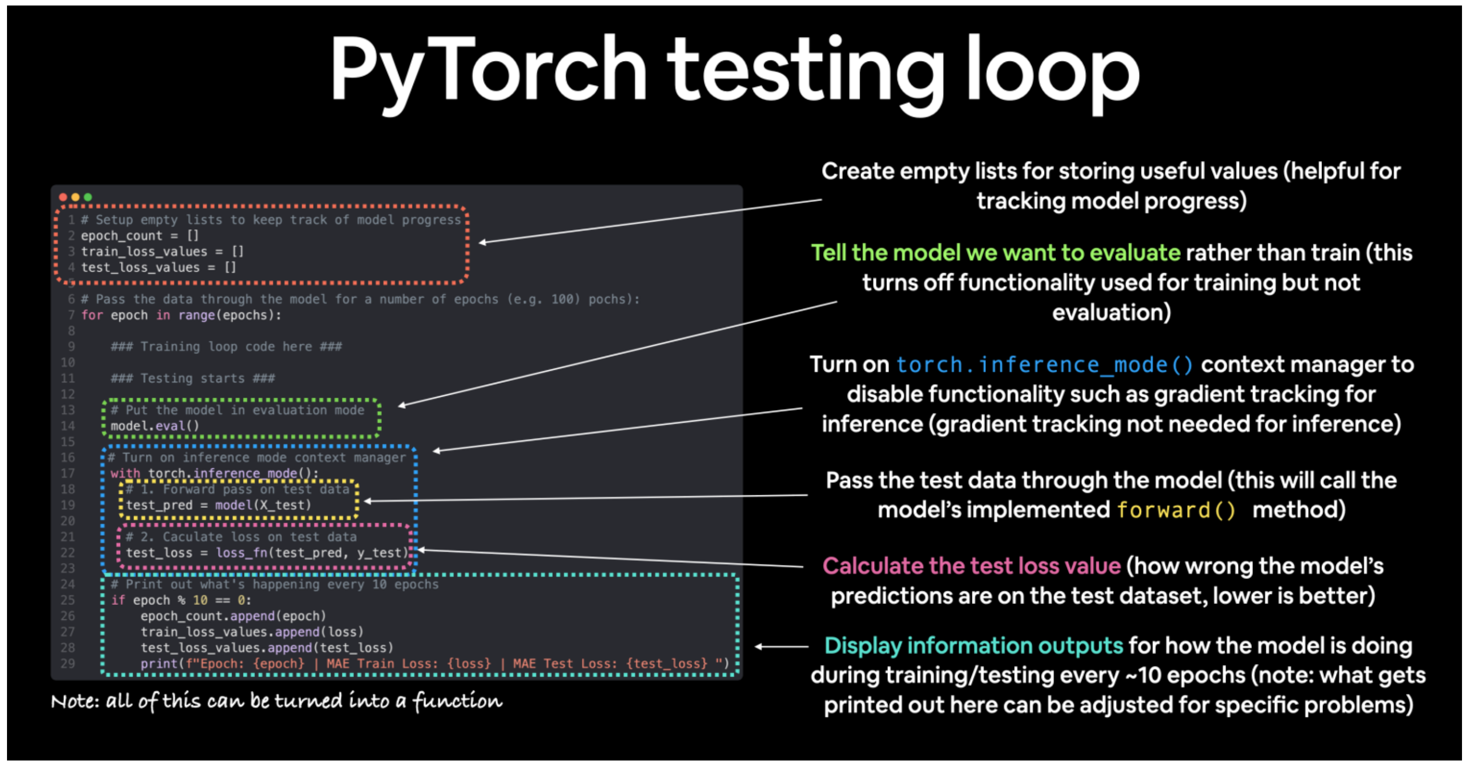
- Interpreting Learning Curve
- Get more data
- Use regularization
- Simplify your model
- Use data augmentation
- Use transfer learning
- Use dropout layers
- Use learning rate decay
- Use early stopping

Prevent Underfitting

- Add more layers/units to your model
- Tweak the learning rate
- Use transfer learning
- Train for longer
- Use less regularization







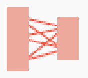
General

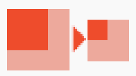
PyTorch is a open source machine learning framework. It uses **torch.Tensor** – multi-dimensional matrices – to process. A core feature of neural networks in PyTorch is the autograd package, which provides automatic derivative calculations for all operations on tensors.

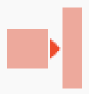
import torch	Root package
import torch.nn as nn	Neural networks
from torchvision import datasets, models, transforms	Popular image datasets, architectures & transforms
import torch.nn.functional as F	Collection of layers, activations & more

torch.randn(*size)	Create random tensor
torch.Tensor(L)	Create tensor from list
tnsr.view(a,b, ...)	Reshape tensor to size (a, b, ...)
requires_grad=True	tracks computation history for derivative calculations

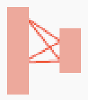
Layers

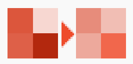

nn.Linear(m, n): Fully Connected layer (or dense layer) from m to n neurons

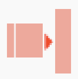

nn.ConvXd(m, n, s): X-dimensional convolutional layer from m to n channels with kernel size s; X ϵ {1, 2, 3}

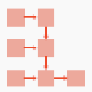

nn.Flatten(): Flattens a contiguous range of dimensions into a tensor


nn.MaxPoolXd(s): X-dimensional pooling layer with kernel size s; X ϵ {1, 2, 3}


nn.Dropout(p=0.5): Randomly sets input elements to zero during training to prevent overfitting


nn.BatchNormXd(n): Normalizes a X-dimensional input batch with n features; X ϵ {1, 2, 3}


nn.Embedding(m, n): Lookup table to map dictionary of size m to embedding vector of size n


nn.RNN/LSTM/GRU: Recurrent networks connect neurons of one layer with neurons of the same or a previous layer

torch.nn offers a bunch of other building blocks.
A list of state-of-the-art architectures can be found at https://paperswithcode.com/sota.

Load data

A dataset is represented by a class that inherits from **Dataset** (resembles a list of tuples of the form (features, label)).

DataLoader allows to load a dataset without caring about its structure.

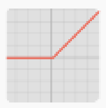
Usually the dataset is split into training (e.g. 80%) and test data (e.g. 20%).

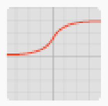
```python
from torch.utils.data
 import Dataset, TensorDataset,
        DataLoader, random_split

train_data, test_data =
 random_split(
    TensorDataset(inps, tgts),
    [train_size,test_size]
 )

train_loader =
      DataLoader(
         dataset=train_data,
         batch_size=16,
         shuffle=True)
```

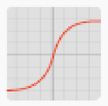
Activation functions

Common activation functions include **ReLU**, **Sigmoid** and **Tanh**, but there are other activation functions as well.

nn.ReLU() *creates a* nn.Module *for example to be used in Sequential models.* F.relu() *ist just a call of the ReLU function e.g. to be used in the* forward *method.*


nn.ReLU() or F.relu()
Output between 0 and ∞, most frequently used activation function


nn.Sigmoid() or F.sigmoid()
Output between 0 and 1, often used for predicting probabilities


nn.Tanh() or F.tanh()
Output between -1 and 1, often used for classification with two classes

Define model

There are several ways to define a neural network in PyTorch, e.g. with **nn.Sequential** (a), as a class (b) or using a combination of both.

```python
model = nn.Sequential(
    nn.Conv2D(■,■,■)
    nn.MaxPool2D(■)
    nn.ReLU()
    nn.Flatten()
    nn.Linear(■,■)
)
```
a

```python
class Net(nn.Module):
  def __init__():
    super(Net, self).__init__()

    self.conv
      = nn.Conv2D(■,■,■)

    self.pool
      = nn.MaxPool2D(■)

    self.fc = nn.Linear(■,■)

  def forward(self, x):
    x = self.pool(
        F.relu(self.conv(x))
    )
    x = x.view(-1,■)
    x = self.fc(x)
    return x
model = Net()
```
b

Save/Load model

model = torch.load('PATH')	Load model
torch.save(model, 'PATH')	Save model

It is common practice to save only the model parameters, not the whole model using model.state_dict()

```python
torch.save(model.state_dict(), 'params.ckpt')
model.load_state_dict(
        torch.load('params.ckpt'))
```

GPU Training

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

If a GPU with CUDA support is available, computations are sent to the GPU with ID 0 using model.to(device) or inputs, labels = data[0].to(device), data[1].to(device).

Train model

LOSS FUNCTIONS

PyTorch already offers a bunch of different loss fuctions, e.g.:

nn.L1Loss	Mean absolute error
nn.MSELoss	Mean squared error (L2Loss)
nn.CrossEntropyLoss	Cross entropy, e.g. for single-label classification or unbalanced training set
nn.BCELoss	Binary cross entropy, e.g. for multi-label classification or autoencoders

OPTIMIZATION (torch.optim)

Optimization algorithms are used to update weights and dynamically adapt the learning rate with gradient descent, e.g.:

optim.SGD	Stochastic gradient descent
optim.Adam	Adaptive moment estimation
optim.Adagrad	Adaptive gradient
optim.RMSProp	Root mean square prop

```python
import torch.optim as optim

# Define loss function
loss_fn = nn.CrossEntropyLoss()

# Choose optimization method
optimizer = optim.SGD(model.parameters(),
                      lr=0.001, momentum=0.9)

# Loop over dataset multiple times (epochs)
for epoch in range(2):
    model.train() # activate training mode
    for i, data in enumerate(train_loader, 0):
        # data is a batch of [inputs, labels]
        inputs, labels = data

        # zero gradients
        optimizer.zero_grad()

        # calculate outputs
        outputs = model(inputs)
        # calculate loss & backpropagate error
        loss = loss_fn(outputs, labels)
        loss.backward()
        # update weights & learning rate
        optimizer.step()
```

```python
correct = 0 # correctly classified
total   = 0 # classified in total

model.eval()
with torch.no_grad():
  for data in test_loader:
    inputs, labels = data
    outputs = model(inputs)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0) # batch size
    correct += (predicted==labels)
                    .sum().item()

print('Accuracy: %s' % (correct/total))
```

Evaluate model

The evaluation examines whether the model provides satisfactory results on previously withheld data. Depending on the objective, different metrics are used, such as acurracy, precision, recall, F1, or BLEU.

model.eval()	Activates evaluation mode, some layers behave differently
torch.no_grad()	Prevents tracking history, reduces memory usage, speeds up calculations