Bertelsmann Technology Scholarship
AI Track – Course Notes for Lesson 5 by @Karin

Introduction to PyTorch

1. BASICS

PyTorch *neural network framework (Python library)* `import torch`
Developed by Facebook AI team, newer than Keras and Tensorflow & completely open-sourced. Released in Jan 2017. Runs on C++ for the speed but the user interfacing parts are in Python.

Tensors *base data structure in neural network frameworks*
1-d tensor = scalar, 2-d tensor = vector, 3-d tensor = matrix. Tensors can be n-dimensional. Features x, weights w, bias b must have torch tensor format!
Example: A 2d image is stored as a 3d tensor: for every pixel there are three values, one for each color channel (red / green / blue).

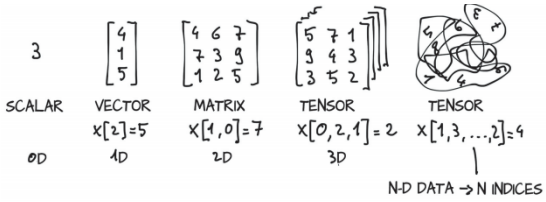


Fig 1: Tensors
Source: Deep Learning with PyTorch. Stevens, Antiga

- Reminder: for **matrix multiplication**, the number of colums of the first matrix A must match the number of rows of the second matrix B. AB ≠ BA. Reshape tensors if necessary!
- Check if the tensors are the correct shape with the `.shape` method during debugging
- Options for **reshaping** tensors with shape specified:
  - `.reshape(a, b, …)` creates new tensor but might return a clone
  - `.resize_(a, b, …)` in-place operation, changes only the tensor shape but might cut off data if new shape has less or more elements
  - `.view(a, b, …)` creates a new tensor with the specified shape but the old data – returns an error if there is a mismatch in number of elements. Most desireable method to safely change the shape of a tensor! Can use -1 for a dimension to be computed automatically.

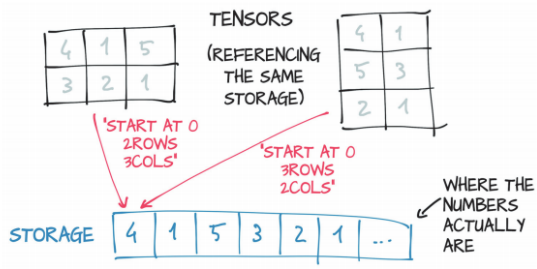


Fig 2: Tensors are views over a Storage Instance
Source: Deep Learning with PyTorch. Stevens, Antiga

Key Formulas in PyTorch Notation

Score ("Logits")	$Wx + b$	`torch.mm(x, w) + b` *most desirable*
		`torch.matmul(x, w) + b` *supports broadcasting → error-prone*
		`torch.sum(x * w) + b`
		`(x * w).sum() + b`

Sigmoid Activation Function	$\frac{1}{1+e^{-z}}$	`1/(1+torch.exp(-z))`
Softmax Activation Function	$\frac{e^{z_i}}{e^{z_1}+\ldots+e^{z_n}}$	`torch.exp(z)/torch.sum(torch.exp(z), dim=1).view(-1, 1)`

Reminder on how Neural Networks work as Universal Function Approximator

During training we pass in many **inputs**, e.g. images, with their actual label (**output**) so that the neural network can build to approximate the function that is converting the images to a probability distribution. How?

1. Initially the weights of the NN are randomly set and after each **forward pass** of data through the network we compare the prediction with the actual label,
2. calculate the prediction error with a **loss function**,
3. change the network weights slightly in the direction that will reduce the error during **backprogation** (by subtracting the gradient multiplied with the learning rate from the weights) and then
4. repeat until the prediction error is small enough.
5. After training, we validate on a separate dataset to see how well the NN generalizes, train/validate again if not happy, and then the NN can be used to make inferences on unseen data.

2. NEURAL NETWORKS IN PYTORCH

Neural Network Definition in PyTorch

We can use torch.nn, torch.nn.functional or torch.nn.sequential.
Example: network with 784 input units, a hidden layer with 128 units and a ReLU activation, a hidden layer with 64 units and a ReLU activation, and an output layer with softmax.

torch.nn.sequential

```
# Hyperparameters for our network
input_size = 784
hidden_sizes = [128, 64]
output_size = 10

# Build a feed-forward network
model = nn.Sequential(nn.Linear(input_size, hidden_sizes[0]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[0], hidden_sizes[1]),
                      nn.ReLU(),
                      nn.Linear(hidden_sizes[1], output_size),
                      nn.Softmax(dim=1))
```

torch.nn.functional

```
class Network(nn.Module):
    def __init__(self):
        super().__init__()
        # Defining the layers, 128, 64, 10 units each
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        # Output layer, 10 units - one for each digit
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        ''' Forward pass through the network'''

        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.softmax(x, dim=1)

        return x

model = Network()
```

Loss Functions in PyTorch

The output of the model has to be adapted depending on which loss function we are using. There are a lot of different loss functions, and in PyTorch there are two possibilities for implementing **negative log-likelihood loss**: CrossEntropyLoss() and NLLLoss(). The latter is preferred.

Table 1: example of model & loss function definition for the MNIST problem (handwritten digits recognition) with 784 input units (28x28 pixels)

CrossEntropyLoss()

*Requires the raw **scores** (aka "logits) of the model as input because PyTorch's CrossEntropyLoss internally applies a softmax function first and then NLLLoss.*

```
model = nn.Sequential(  nn.Linear(784, 128)
                        nn.ReLu()
                        nn.Linear(126, 64)
                        nn.ReLu()
                        nn.Linear(64, 10))

images, labels = next(iter(trainloader))
images = images.view(images.shape[0], -1) # flatten to 1D

criterion = nn.CrossEntropyLoss()
logits = model(images)
loss = criterion(logits, labels)
```

NLLLoss()

*Requires **log-softmax output** of the model as input. So, we explicitly apply a softmax activation function in the model definition. Benefit is that we could convert the log-softmax output to actual probabilities with torch.exp(output).*

```
model = nn.Sequential(  nn.Linear(784, 128)
                        nn.ReLu()
                        nn.Linear(126, 64)
                        nn.ReLu()
                        nn.Linear(64, 10)
                        nn.LogSoftmax(dim=1))
images, labels = next(iter(trainloader))
images = images.view(images.shape[0], -1) # flatten to 1D

criterion = nn.NLLLoss()
logps = model(images)
loss = criterion(logps, labels)
ps = torch.exp(logps)
```

Note about nn.LogSoftmax(dim=1): dim=1 means it calculates softmax across the columns instead of the rows, so across each of our examples (rows are examples) and not across each individual feature (columns are features)
Note about the forward pass: it can be written as model(images) or model.forward(images) – it is equivalent!

Backpropagation in PyTorch

Gradient Calculation PyTorch can keep track of all operations done on tensors with the module **autograd**. If autograd is set on a tensor, it can go backwards through each of the operations done and calculate the gradient with respect to the input parameters.

- **Switching autograd on/off:**
  - Turn on autograd during tensor creation: `torch.zeros(1, requires_grad=True)`
  - Turn on autograd on a tensor x at any time: `x.requires_grad_(True)`
  - Turn off autograd for a block of code: `with torch.no_grad():`
    this is useful to speed up & save memory, e.g. in forward passes to make predictions
  - Turn off autograd entirely: `torch.set_grad_enabled(False)`
  - Check if autograd is enabled for a specific tensor x: `x.requires_grad`

- **Calculating the gradient for a tensor z where autograd was switched on:**
  - Run the backward method on z with respect to x (for example the loss): `z.backward()`
  - Calculate the gradient for x: `x.grad()`

Updating the Weights `from torch import optim`

Once the gradient is calculated, we update the weights & biases of the network with an **optimizer** from PyTorch's optim package, e.g. Stochastic Gradient Descent

- **Define the optimizer** with one of the built in optimizers:
  `optimizer = optim.SGD(model.parameters(), lr = 0.003)` or
  `optimizer = optim.Adam(model.parameters(), lr = 0.003)`, which is like SGD, but it uses momentum to speed up the fitting process. It also adjusts the learning rate for each of the individual parameters of the model
- **Clear the gradients:** PyTorch per default accumulates the gradients, this means they would get summed up in multiple forward/backward passes, so the gradients must be cleared before each training step with `optmizer.zero_grad()`
- **Update the weights** with `optimizer.step()`

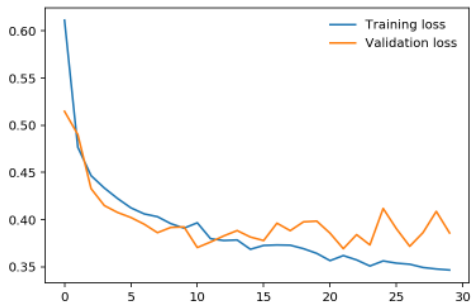
Model Validation *measuring model performance on data that is not part of the training set*

- **Top-5 Error Rate** `.topk()` applied to the probabilities tensor returns a tuple of the *k* highest probabilities and their corresponding class. Usually we are only interested in the highest class:
  top_p, top_class = ps.topk(1, dim=1)
  print(top_class)
- **Accuracy** (% predicted correctly) is checking for how many cases top_class equals the label:
  *equals = top_class == labels.view(*top_class.shape)*
  accuracy = torch.mean(equals.type(torch.FloatTensor))
  print(accuracy)
- **Precision / Recall**

Watchout: If we are using any dropout layers in our model (nn.Dropout(0.2)), we must switch off dropout in the validation pass and for inferences by turning to evaluation mode with `model.eval()`

3. PUTTING IT ALL TOGETHER: TRAINING / VALIDATION LOOP

```
model = Classifier()
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.003)
epochs = 30
steps = 0

train_losses, test_losses = [], []
for e in range(epochs):
    running_loss = 0
    # TRAINING LOOP
    for images, labels in trainloader:
        optimizer.zero_grad()
        log_ps = model(images)
        loss = criterion(log_ps, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    # VALIDATION LOOP
    else:
        test_loss = 0
        accuracy = 0
        # Turn off gradients for validation, saves memory and computations
        with torch.no_grad():
            model.eval() # switch to evaluation mode as we are using drop-out in our model
            for images, labels in testloader:
                log_ps = model(images)
                test_loss += criterion(log_ps, labels)

                ps = torch.exp(log_ps)
                top_p, top_class = ps.topk(1, dim=1)
                equals = top_class == labels.view(*top_class.shape)
                accuracy += torch.mean(equals.type(torch.FloatTensor))

        model.train() # switch back to training mode to include drop-out again

        train_losses.append(running_loss/len(trainloader))
        test_losses.append(test_loss/len(testloader))

        print("Epoch: {}/{}.. ".format(e+1, epochs),
              "Training Loss: {:.3f}.. ".format(train_losses[-1]),
              "Test Loss: {:.3f}.. ".format(test_losses[-1]),
              "Test Accuracy: {:.3f}".format(accuracy/len(testloader)))
```



validation & training loss over epochs

Useful Tip:
Using the GPU will speed up computations massively

- Check for GPU availability:
  `torch.cuda.is_available()`
- In any case, make sure the model and all necessary tensors (e.g. images) are moved to either the GPU or CPU with `.to(device)` where `device` is either `"cuda"` or `"cpu"`.

4. IMAGE CLASSIFICATION WITH TRANSFER LEARNING – STEP BY STEP

Create Folders with Training & Validation Images

Images must be in a **folder structure** with one folder per class (example to the right). The folder name per class becomes the label of each image.

```
root/dog/xxx.png
root/dog/xxy.png
root/dog/xxz.png

root/cat/123.png
root/cat/nsdf3.png
root/cat/asd932_.png
```

Transform Images

1. **Resize** all images to the same size (adjust pixel size / crop the center)
2. **Data Augmentation** as needed (on training images only!) to introduce some randomness
3. Convert images to PyTorch **Tensor**
4. **Normalize** the color channels (subract mean & divide by standard deviation)

All transforms can be combined into a pipeline with `transforms.Compose()`

Load Images

The DataLoader takes a dataset and returns batches of images and the corresponding labels (the label is taken from the folder structure!). It is a **generator**. To get data out of it, you need to loop through it or convert it to an **iterator** and call next().

Load a Pre-Trained Model for Transfer Learning

We **select a pre-trained network** (e.g. VGGNet or DenseNet), **define a classifier, set the criterion and optimizer**. Then we load the model and **freeze the weights** for the whole network except that of the final fully connected layer. This last fully connected layer is replaced with our own classifier with random weights and only this layer is trained in the next step. Note about the classifier: the number of **input features** of the classifier depend on the pretrained model (e.g. 25088 for VGG and 1024 for DenseNet), the number of **output features** depends on the number of classes we're trying to predict.

Note: Such networks were trained on images on ImageNet – a collection of 1 million labelled images from 1,000 categories. Most of the pretrained models require the input to be 224x224 images.

Train the Model (Classifier only)

Training is an iterative process and is done over a series of **Epochs**. Each epoch consists of a training part and evaluation part. We stop the iteration when we're happy with the model's accuracy.

Training Mode: The algorithm computes the output predictions by passing inputs (images) to the model (**forward pass**). Then the batch **loss** is calculated by comparing vs. the actual labels. The gradient of the loss is computed with respect to model parameters via a **backward pass**. Finally, the parameters are updated in an **optimization** step.

Evaluation Mode: The model is asked to make a prediction on validation images with a **forward pass**. Then the batch **loss** is calculated by comparing vs. the actual labels. The optimizer is turned off completely. **Performance measures** are typically *accuracy* (= percentage of classes the network predicted correctly), *precision*, *recall*, *top-5 error rate*.

Could consider to "Retrain" to get an even higher accuracy after training the classifier. We could unfreeze the whole model and train the whole model and classifier again with a small Learning Rate. The very small learning rate of 0.0001 or smaller will tune the model to work better with a specific dataset.

Save Model / Load Model

After having trained a model, we can save it for later fine-tuning or for making predictions. As the tensors for weights and biases are stored in model.state_dict(), we can save this and name the file:

- Saving: `torch.save(model.state_dict(), "checkpoint.pth")`
- Loading: `torch.load("checkpoint.pth")`

The model however must have the same architecture (same number of input/hidden/output layers) as the network that the checkpoint was created with. So, ideally we also save the model architecture in a dictionary with all the necessary information to re-build the model.

Make Predictions

An image is pre-processed so it fullfills the model requirements (correct size & transformed to a tensor) and then passed into the network (forward pass) to predict the most likely class it belongs to.