Deep learning in Python

Instructor: Jorge Mendez
Logistics

• There is a survey out about time-zones on Piazza. Please fill it out now!
• Homework 6 is out, due this Friday
• Homework 7 will be out this Friday
Deep learning
Why deep learning?

• Naïve features often require highly complex functions
  • E.g., all pixels of a 480 x 320 image as one long vector of features

• SVMs use the kernel trick to get around this
  • Doesn’t work well if there are too many data points (e.g., images)

• Deep learning learns the “features” necessary for the model to make good predictions
Examples of the success of deep learning

• Pix2pix models to generate images

• Machine translation systems

• Super-human game-playing
Perceptron

- Hyper-simplified computational model of neuron

\[ y = f \left( \sum_i x_i w_i \right) = f(x^T w) \]

\[ f(g) = \begin{cases} 
0 & \text{if } g < 0 \\
1 & \text{if } g \geq 0 
\end{cases} \quad \text{(step function)} \]

- Look closely: this is the same as logistic regression
Problem with the perceptron

• It is too simple, and can’t learn complex functions

• The classic example: XOR
  • There is no perceptron (weight assignment) that can solve the XOR

• This shouldn’t be surprising... it’s just a linear classifier!
Artificial neural networks

• If we can make neurons, we can make a whole brain, too
  • Human brain has 100 billion neurons. We can’t make neural nets with 100 billion of these...
  • Plus real neurons and their connections are way more complex
• Each node represents a perceptron or neuron
• Each edge represents a weight
Notes on neural nets

• The function $f()$ can be any non-linear function
• Theorem: infinite-width networks can learn any function
  • In practice: well-designed networks can learn functions if the data is good
• Neural nets are trained via backpropagation
  • Fancy word for chain rule
  • It’s really just gradient descent
  • Then $f()$ needs to be a differentiable non-linear function
Activation functions

- $f()$ is called an activation function
  - It determines when a neuron fires
- Sigmoid is the traditional activation
  - Differentiable approximation of the step function from the first slide
  - It suffers from a variety of problems in practice
- Nowadays, ReLU is the go-to choice for *hidden* layers
Output layer activation

• It is the prediction function computed over “learned features”
• It is chosen based on the problem
  • Binary classification — sigmoid: approximates 0-1
  • Multi-class classification — softmax: a probability over all labels
  • Regression — linear: no activation, so output can be in any range
Loss functions

• Loss function: objective to minimize via gradient descent
  • Tells the network where to move weights based on correctness of output
• Also varies depending on the problem
• Binary classification: \textit{binary cross-entropy} loss (log-likelihood)
• Multi-class classification: \textit{cross-entropy} loss
• Regression: \textit{quadratic} loss
• It is possible to also add regularization terms
  • L2 regularization ($\lambda \| w \|_2^2$) is called weight decay in neural net speak
Overfitting and underfitting

• Underfitting: too simple model, can’t capture real function
• Overfitting: too complex model, learns even the noise
  • Can’t generalize well to unseen data
How to avoid overfitting?

• Use simpler models
  • Deep nets that are less deep and/or less wide

• Use weight decay

• Use dropout
  • Somewhat weird concept
  • Randomly drop neurons during training
  • Randomization makes the network robust
Features vs Deep learning

• Traditional ML models required *feature engineering*
  • Construct features from raw data
  • E.g., image processing to find edges, corners, segments...

• Deep learning can train models directly from raw data
  • Requires less knowledge of the structure of the data
  • Can “learn” better features than can be constructed manually
  • Simply works better for a lot of problems
Example features learned by deep learning
Deep learning in PyTorch
Contrast to scikit-learn

• Deep learning is still in a somewhat early stage
• There is no `model = NeuralNetwork()` or `model.fit()` or `model.predict()`
• Instead, you create your own models and parts of the training process
Tensors

• PyTorch tensors are similar to NumPy arrays
  • In fact, the underlying data *is* a NumPy array
• They have additional information that is used for backpropagation
• `torch.Tensor(iterable)` — like the NumPy constructor, parses the iterable recursively
• `torch.from_numpy(ndarray)` — does not copy the ndarray
• Input type to all Pytorch functions
Layer types

• Layers return a *function*, that can then be applied to any input

• `nn.Linear(in_size, out_size)` — linear layer: this is the one we’ve studied so far
  • Activation function is added after the linear layer

• `nn.ReLU()`, `nn.Tanh()`, `nn.Sigmoid()` — activation functions

• `nn.Dropout(p)` — dropout with probability \( p \)
Layer types

- **nn.Conv2d(in_channels, out_channels, kernel_size)** — convolutional layer: this is typically used for image processing. Incorporates notion of spatial relation between pixels
  - Assumes input is of shape $n \times \text{in\_channels} \times H_{\text{in}} \times W_{\text{in}}$
  - Output is of shape $n \times \text{out\_channels} \times H_{\text{out}} \times W_{\text{out}}$
  - kernel_size selects how many pixels are looked at simultaneously
- **nn.MaxPool2d(kernel_size)** — select the maximum value from each window of size kernel_size
  - Used to reduce the size of the image

- Typically, the pipeline for conv nets is

  
  image(128x128x3) $\rightarrow$ conv(3, 8, 3) $\rightarrow$ maxpool(3) $\rightarrow$ ReLU() $\rightarrow$ conv(8, 16, 3) $\rightarrow$ maxpool(3) $\rightarrow$ ReLU()

  
  or something similar, until the image width is reduced to $1 \times 1 \times \text{out\_channels}$, and then one or more linear layers with ReLU
Live example
Constructing a simple model

• `model = nn.Sequential(*args)`
• Each arg in `args` is a layer
• The model stacks the layers in the order in which they are passed
• `model(X)` gives the result of the output layer after passing X through each layer in the model
Live example
Constructing more complex models

• You can’t typically create conv nets with `nn.Sequential`
  • Or any other model that requires operations other than PyTorch layers
• You must create a class that sub-classes `nn.Module`
• In the `__init__()` method you create the layers
• In the `forward(X)` method you pass the input through each layer sequentially, and apply any other needed functions
• We will see an example
Loss functions

- Loss functions (like layers) return a function that can be applied to any input.
- `nn.MSELoss()` — mean squared error (for regression)
- `nn.BCEWithLogitsLoss()` — computes sigmoid and then the binary cross-entropy loss (for binary classification)
- `nn.CrossEntropyLoss()` — computes softmax and then cross-entropy loss (for multi-class classification)
- Note: both `BCEWithLogitsLoss` and `CrossEntropyLoss` take as input the output of a linear layer *without* activation
Backpropagation

• Crucial: PyTorch automatically computes derivatives for you
• The process is typically:

```python
model = SomeModel()
optimizer = SomeOptimizer()
y_hat = model(X)
loss = LossFn(y_hat, y)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```
Training neural nets

- A single backpropagation step is equivalent to a single gradient descent step
- Iteratively do backpropagation to train the net
- Typically done in mini-batches
  - It is too expensive to evaluate the network on all data
  - Choose a mini-batch of data at each time
Live example
Data handling functions

- `torch.utils.data.DataSet` — abstract class to represent datasets
  - Includes `__getitem__` method for bracket indexing
  - `torchvision` includes implementations for many common computer vision datasets

- `torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False)` — class to sample data from a dataset
  - Useful for mini-batch training!
  - Optionally shuffles the data (recommended)
  - It is an iterable, which means you can do a `for` over it
Torchvision datasets

- `torchvision.datasets.<DatasetName>(root, train=True, transform=None, download=False)`
  - `<DatasetName>` can be any of many datasets (e.g., MNIST, ImageNet CIFAR10)
  - `root`: path to where the data is located
  - `train`: whether to load the training set or the test set
  - `transform`: a transformation function to apply to each data point
  - `download`: whether to download the data set

- `__getitem__` returns a `transform(image), label` tuple
Torchvision transforms

• Classes that output a function to apply to each input
• `torchvision.transforms.Compose(transforms)` — chain transforms one after the other from list of transforms
• `torchvision.transforms.ToTensor()` — transform a PIL image or NumPy array to a tensor
  • Also scales images in \([0, 255]\) to \([0.0, 1.0]\)
• `torchvision.transforms.Normalize(mean, std)` — normalize a tensor image with given mean and std to standard Gaussian
  • mean and std have one scalar for each channel in the image
• `torchvision.transforms.Lambda(lambda_)` — apply user-defined function
Live example
More notes

• Typically, for large models you train on GPU
  • Unfortunately, recent versions of PyTorch don’t work on GPUs for Mac

• Deep learning is a huge topic, with lots of new concepts introduced every year

• It is widely studied in academia, and it is rapidly being adopted in industry as well
Takeaways

• PyTorch is substantially harder to use than scikit-learn
  • Although it is easier than TensorFlow
• Deep models are all the rave, they work great with lots of data
• If you don’t have a lot of data, you are better off using simpler models