EDAN96
Applied Machine Learning
Lecture 9: Training Techniques, Backward Propagation, and Automatic Differentiation

Pierre Nugues

Pierre.Nugues@cs.lth.se

November 28, 2022

Content

Overview and practice of some neural network architectures:

- Logistic loss
- The `Module` class
- Dense vectors
- A word on data loaders
- Backward propagation
- A word on automatic differentiation

Creating a Network with PyTorch

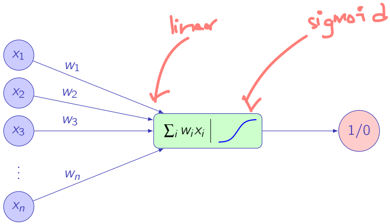
So far, we have used the `Sequential` class to create networks

For more complex architectures, we need to derive a class from `nn.Module`

This class must have the `__init__()` and `forward()` methods:

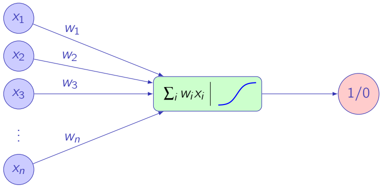
- In the `__init__()` constructor, you declare and create all the trainable parameters
- `forward()` implements the computation of the forward pass

Logistic Regression



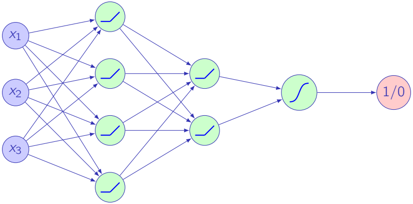

```
model = nn.Sequential(
    torch.nn.Linear(input_dim, 1),
    torch.nn.Sigmoid())
```

Logistic Regression




```
class Model(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 1)

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        return x
```

Neural Networks with Hidden Layers



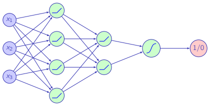

```
model = nn.Sequential(
    nn.Linear(input_dim, 4),
    nn.ReLU(),
    nn.Linear(4, 2),
    nn.ReLU()
    nn.Linear(2, 1),
    torch.nn.Sigmoid()
    )
```

Weights in this layer will be learned during training

Activation function to introduce non-linearity into the model

Squash input values to the range [0 ,1] for binary classification, which can be interpreted as probability.

Neural Networks with Hidden Layers



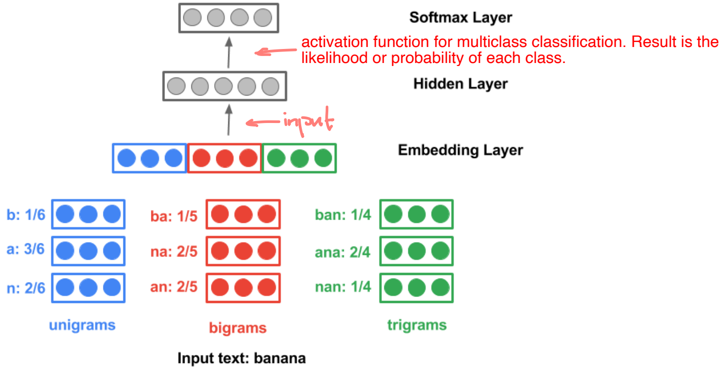

```
class Model2(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 4)
        self.fc2 = nn.Linear(4, 2)
        self.fc3 = nn.Linear(2, 1)
                             input_dim

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.sigmoid(self.fc3(x))
        return x
```

Code Example

Experiment: Deriving a class with a Jupyter Notebook:
https://github.com/pnugues/edan96/blob/main/programs/11-Salammbo_class_torch.ipynb

Sum of Embeddings in CLD3

CLD3 computes the weighted sum of embeddings



Categorical Values: Multi-hot encoding

A collection of two documents D1 and D2:

D1: Chrysler plans new investments in Latin America.

D2: Chrysler plans major investments in Mexico.

Multi-hot encoding (also called a bag-of-words representation):

D.	america	chrysler	in	investments	latin	major	mexico	new	plans
1	1	1	1	1	1	0	0	1	1
2	0	1	1	1	0	1	1	0	1

This technique can create extremely large sparse vectors

In bag-of-words representation, document is represented as an unordered set of words, disregarding grammar, and word order but keeping track of word frequency. The idea is to convert a document to a multiset of words and keep track of its frequency in the document.

Dense Vectors

Binary vectors basically

-

We can replace one-hot vectors by dense ones using embeddings

A dense representation is a trainable vector of 10 to 300 dimensions.

The vector parameters are learned in the fitting procedure.

Dimensionality reduction inside a neural network.

Many techniques, often based on language modeling, here CBOW

Cloze Test

Guess a missing word given its context. Using the example:

Sing, O **goddess**, *the anger of Achilles son of Peleus,*

Cloze test: A reader, given the incomplete phrase:

```
Sing, O ____, the anger of Achilles
```

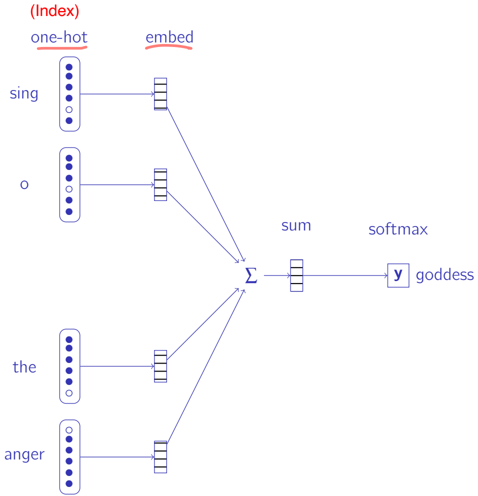
has to fill in the blank with the correct word, here **goddess**.

Easy to create a dataset for a Cloze test

$$\mathbf{X} = \begin{bmatrix} \text{sing} & \text{o} & \text{the} & \text{anger} \\ \text{o} & \text{goddess} & \text{anger} & \text{of} \\ \text{goddess} & \text{the} & \text{of} & \text{achilles} \\ \text{the} & \text{anger} & \text{achilles} & \text{son} \\ \text{anger} & \text{of} & \text{son} & \text{of} \\ \text{of} & \text{achilles} & \text{of} & \text{peleus} \end{bmatrix}; \mathbf{y} = \begin{bmatrix} \text{goddess} \\ \text{the} \\ \text{anger} \\ \text{of} \\ \text{achilles} \\ \text{son} \end{bmatrix}$$

CBOW Architecture

Context words one-hot encoded, in practice just an index, followed by an **embedding layer.**



Embeddings in PyTorch

PyTorch has an `Embedding(num_embeddings, embedding_dim, ...)` class

An embedding object is a matrix from which we can extract embedding vectors using an index

This is just a lookup table

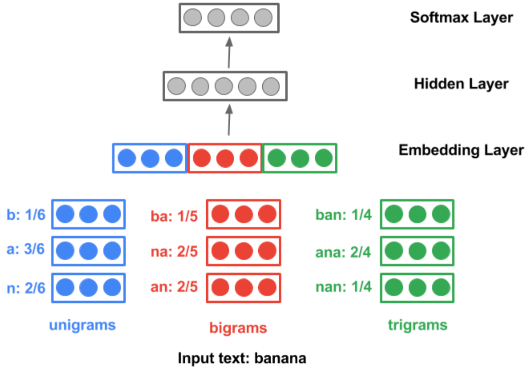
```
# Creates trainable vectors of size 64
embedding_chars = nn.Embedding(MAX_CHARS, 64)

# Extracts embeddings in rows 3 and 2,
# corresponding to two characters
embedding_chars(torch.LongTensor([3, 2]))
```

Code Example

Experiment: Embeddings with a Jupyter Notebook:
`https://github.com/pnugues/edan96/blob/main/programs/8-pytorch_embeddings.ipynb`
To create a batch, we would need to pad the character, bigram, and trigram hash codes.

Sum of Embeddings in CLD3

CLD3 computes the weighted sum of embeddings



Embedding Bags in PyTorch

Calc weighted sum of embeddings

EmbeddingBags class creates embedding objects.

```
embedding_bag = nn.EmbeddingBag(MAX_CHARS, 64, mode='sum')
```

Given a list of embeddings (a list of rows) as input, an embedding bag returns the weighted sum of the embeddings.
We specify the weights with a `per_sample_weights` parameter.
https://pytorch.org/docs/stable/generated/torch.nn.EmbeddingBag.html

Programming Embedding Bags in PyTorch (I)

```
embedding_bag = nn.EmbeddingBag(MAX_CHARS, 64, mode='sum')

# Computes the sum of rows 1 and 2 and rows 3 and 4
# The result is a matrix of two rows
embedding_bag(torch.tensor([[1, 2], [3, 4]]))
```

Result is matrix of 2 rows

```
embedding_bag(torch.tensor([[1, 2], [3, 4]]),
           per_sample_weights=torch.tensor([[0.5, 0.5],
           [0.2, 0.8]]))
```

For first sequence [1,2] the weights are:
The resulting embedding is the weighted sum of the embeddings for indices 1,2

For second sequence [3,4] the weights are:
The resulting embedding is the weighted sum of the embeddings for indices 3,4

Programming Embedding Bags in PyTorch (II)

With bags of unequal sizes, we have to use a list of offsets

Result is Matrix of 1 row

```
embedding_bag(torch.tensor([1, 2, 3, 4]),
      offsets=torch.tensor([0, 2]),
      per_sample_weights=torch.tensor([0.5, 0.5, 0.2, 0.8]))
```

Code Example

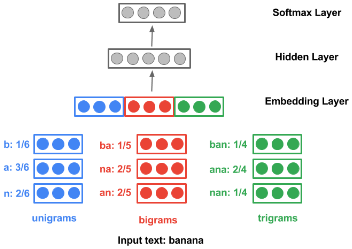
Experiment: Embedding bags with a Jupyter Notebook:
https://github.com/pnugues/edan96/blob/main/programs/8-pytorch_embeddings.ipynb

Adding the embeddings

Describe a language detector: Given a string predict the language:

- *Bonjour* –> French
- Guten Tag –> German

Follow the complete compact language detector (CLD3)
`https://github.com/google/cld3`



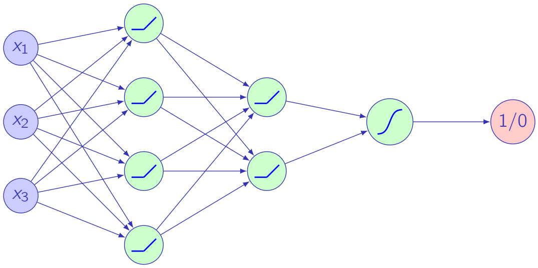
Code Example

Experiment: Classification with embedding bags Jupyter Notebook:
`https://github.com/pnugues/edan96/blob/main/programs/12-classification_embedding.ipynb`

Data Loaders

- Current datasets have now terabytes of data
- Impossible to fit into memory (even Tatoeba)
- For real world applications, you will have to use or write a data loader that can create smaller, processable batches from your storage
- This involves the `Dataset` and `DataLoader` classes
- Beyond the scope of this lecture
- Read on this here: `https://pytorch.org/blog/efficient-pytorch-io-library-for-large-datasets-many-file`

Backpropagation: The Forward Pass



The forward pass:

1. Layer 1 $f^{(1)}(\mathbf{W}^{(1)}\mathbf{x})$, where $f^{(1)}$ is the activation function.
2. For the second layer, $f^{(2)}(\mathbf{W}^{(2)}f^{(1)}(\mathbf{W}^{(1)}\mathbf{x}))$, Takes the previous results and apply new weight
3. Last layer (L) and output the prediction:

$$\hat{y} = f^{(L)}(\mathbf{W}^{(L)} \ldots f^{(2)}(\mathbf{W}^{(2)} f^{(1)}(\mathbf{W}^{(1)}\mathbf{x}))\ldots).$$

4. For the figure $\hat{y} = f^{(3)}(\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x})$, where $f^{(3)}(x)$ is the logistic function.

Naive Gradient Descent

Try to minimize the difference between the predicted and observed annotations: $Loss(y, \hat{y})$.

$$\mathbf{w}_{(k+1)} = \mathbf{w}_{(k)} - \underbrace{\alpha_{(k)} \nabla Loss(\mathbf{w}_{(k)})}_{\text{Differences in weight}}.$$

We compute the gradient:

$$\begin{aligned}
\frac{\partial Loss(\mathbf{w})}{\partial w_{ij}^{(l)}} &= \frac{\partial(-y \ln \hat{y} - (1-y)\ln(1-\hat{y}))}{\partial w_{ij}^{(l)}} \\
&= \frac{\partial(-y \ln f^{(3)}(\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x}) - (1-y)\ln(1-f^{(3)}(\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x})))}{\partial w_{ij}^{(l)}},
\end{aligned}$$

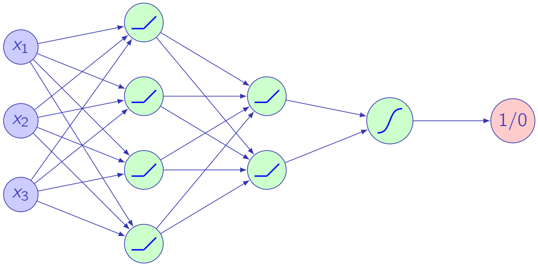
derivision of loss function over weight

for all the weights $w_{ij}^{(l)}$.

Method impractical in real cases (billions of weights)

Breaking Down the Computation

We first compute the gradient with respect to the inputs.



$$
\begin{aligned}
\hat{y} &= \mathbf{a}^{(L)}, \\
&= f^{(L)}(\mathbf{z}^{(L)}), \\
&= f^{(L)}(\mathbf{W}^{(L)}\mathbf{a}^{(L-1)})
\end{aligned}
$$

Recurrence Relation

We can show that this relation applies for any pair of adjacent layers l and $l-1$ in the network:

$$\nabla_{\mathbf{z}^{(l-1)}}\mathbf{z}^{(l)} = f^{(l-1)\prime}(\mathbf{z}^{(l-1)})^{\intercal} \odot \mathbf{W}^{(l)}.$$

For our network:

$$\begin{aligned}
\nabla_{\mathbf{x}} Loss(y, \hat{y}) &= -\frac{\partial(y \ln \hat{y} + (1-y)\ln(1-\hat{y}))}{\partial \mathbf{x}}, \\
&= -\frac{y-\hat{y}}{\hat{y}(1-\hat{y})}\hat{y}(1-\hat{y})\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}, \\
&= (\hat{y}-y)\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}, \\
&= (f^{(3)}(\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x}) - y)\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}.
\end{aligned}$$

Gradient with Respect to the Weights

We now compute the gradient with respect to $\mathbf{W}^{(l)}$, l being the index of any layer. From the chain rule, for the last layer, L, we have:

$$\nabla_{\mathbf{W}^{(L)}} Loss(y, \hat{y}) = \frac{\partial Loss(y, \hat{y})}{\partial \mathbf{z}^{(L)}} \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{W}^{(L)}}$$

and

$$\begin{aligned} \mathbf{z}^{(L)} &= \mathbf{W}^{(L)} f^{(L-1)}(\mathbf{z}^{(L-1)}), \\ &= \mathbf{W}^{(L)} \mathbf{a}^{(L-1)}. \end{aligned}$$

The partial derivatives of $\mathbf{z}^{(L)}$ with respect to $\mathbf{W}^{(L)}$ simply consist of the transpose of $\mathbf{a}^{(L-1)}$. Then, we have:

$$\boxed{\frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{W}^{(L)}} = \mathbf{a}^{(L-1)\intercal}.}$$

We can show:

$$\nabla_{\mathbf{W}^{(l)}} Loss(y, \hat{y}) = \nabla_{\mathbf{z}^{(l)}} Loss(y, \hat{y}) \mathbf{a}^{(l-1)\intercal}.$$

Code Example

Experiment: Checking the gradient with PyTorch Jupyter Notebook:
`https://github.com/pnugues/edan96/blob/main/programs/backprop_mse_test.ipynb`

Backward Differentiation

A generalization of backpropagation

PyTorch records all the operations in the forward pass Saving them in tensors

It then computes the graph of derivatives using the chain rule proceeding backwards By collecting the derivates of error with respect to the parameters of the function (gradients), we can optimise the parameters using gradient decent by subtracting a small quantity proportional to the gradients, the learning rate

1. `https://pytorch.org/blog/overview-of-pytorch-autograd-engine/`
2. `https://github.com/pytorch/pytorch/blob/master/tools/autograd/derivatives.yaml`

Using PyTorch's example:

$$f(x, y) = \log xy$$

We have:

$$g(x, y) = xy$$

$$\begin{aligned} \frac{\partial f}{\partial x} &= \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} = \frac{1}{xy} y = \frac{1}{x} \\ \frac{\partial f}{\partial y} &= \frac{\partial f}{\partial g}\frac{\partial g}{\partial y} = \frac{1}{xy} x = \frac{1}{y} \end{aligned}$$

Computing the Gradient

Modern machine-learning platforms use an automatic differentiation algorithm. Collects gradients

1. For a video overview: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi, especially the two last lectures;
2. PyTorch https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
3. Functions: https://github.com/pytorch/pytorch/blob/master/tools/autograd/derivatives.yaml
4. For a description of it in Tensorflow, see https://www.tensorflow.org/guide/autodiff
5. For a description of the `tf.gradients` class: https://www.tensorflow.org/api_docs/python/tf/gradients
6. For a more elaborate description: http://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec06.pdf