Introduction to deep learning

Lecture 4

Sam van Gool
vangool@irif.fr

Master 2 informatique
Université Paris Cité

28 January 2025

Today's lecture

Last week

CM:

- Manual derivation of *gradient* for softmax.
- How to *vectorize*: treating an entire batch in one go.
- Numeric stability in calculating softmax.
- The *ReLU* activation function and its gradient.

TP:

- *Vectorizing* softmax regression in Numpy.
- Faster training.

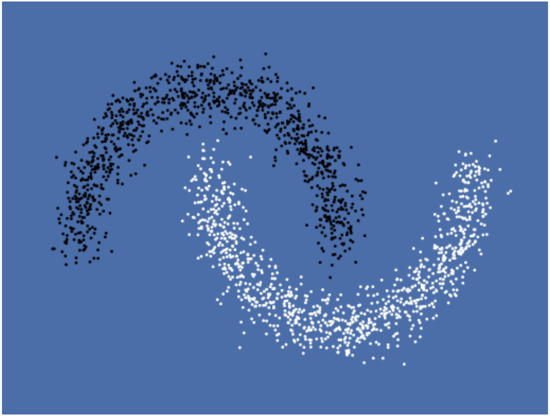
Reminder: Course webpage: `https://samvangool.net/iap/`

TO DO last week

- Finish TP3.
- Read [P, Chapter 3].
- Review the calculation of the gradient of softmax and the justification of the numeric stability trick.

Today's lecture

Motivation: not all data is linear

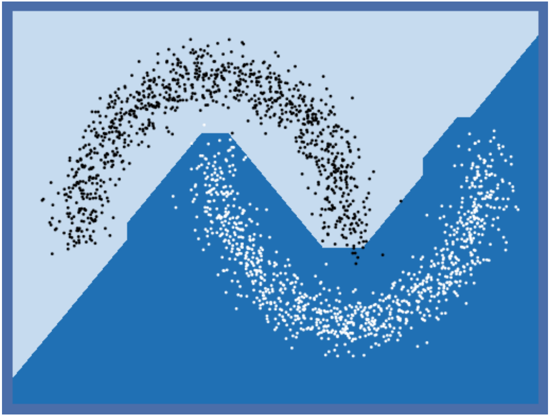


(Source: the function `make_moons` in scikit-learn.)

Reflection

How to use a linear classifier for these data?

Piecewise linear decision boundaries



The best decision boundaries are not straight lines!

Two-layer perceptron

- In week 2, we saw a fully connected linear layer (4 inputs, 3 outputs):

$$o_1 = \phi_{10} + \phi_{11}x_1 + \phi_{12}x_2 + \phi_{13}x_3 + \phi_{14}x_4$$
$$o_2 = \phi_{20} + \phi_{21}x_1 + \phi_{22}x_2 + \phi_{23}x_3 + \phi_{24}x_4$$
$$o_3 = \phi_{30} + \phi_{31}x_1 + \phi_{32}x_2 + \phi_{33}x_3 + \phi_{34}x_4$$

- We saw that it can be written concisely as: $\mathbf{o} = \mathbf{x}^{\mathrm{T}} \cdot \boldsymbol{\Omega} + \boldsymbol{\beta}$.
- We now apply an activation function $a\colon \mathbb{R} \to \mathbb{R}$ after this layer:

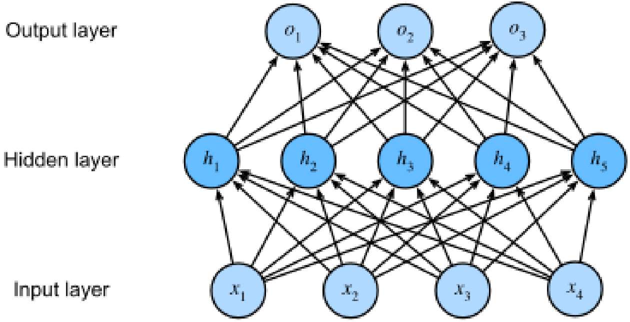
$$\mathbf{h} = a(\mathbf{o}).$$

- And then we apply another linear layer to $\mathbf{h}$:

$$\mathbf{o}' = \mathbf{h}^{\mathrm{T}} \cdot \boldsymbol{\Omega}' + \boldsymbol{\beta}'.$$

- 2-layer perceptron = multilayer perceptron with 1 hidden layer.

Two-layer perceptron, visually



[Z, Figure 5.1.1]

Reflection

In this example, what are the shapes of the matrices Ω and Ω', and of the bias vectors β and β'?

Note: The bias vectors and the activation functions are usually *not visualized* in drawings like the above.

Two-layer perceptron, generally

- Parameters:
  - n the batch size,
  - d the number of input features,
  - h the number of hidden features (this is a *hyperparameter*),
  - q the number of output features ($=$ number of classes).
- the input matrix $\mathbf{X}$ is of size $n \times d$.
- the **first layer's weights** matrix $\boldsymbol{\Omega}$ of size $d \times h$.
- the **first layer's biases**, a vector $\boldsymbol{\beta}$ of length h.
- the **second layer's weights** matrix $\boldsymbol{\Omega}'$ of size $h \times q$.
- the **second layer's biases**, a vector $\boldsymbol{\beta}$ of length q.
- We define the **hidden layer activations**: $\mathbf{H} = a(\mathbf{X} \cdot \boldsymbol{\Omega} + \boldsymbol{\beta})$,
- and the **output logits**: $\mathbf{O} = \mathbf{H} \cdot \boldsymbol{\Omega}' + \boldsymbol{\beta}'$.
- For classification, we get probabilities by applying **softmax** to every row of logits: $\mathbf{P}[i] = \mathrm{softmax}(\mathbf{O}[i])$.

Why do we need activation functions?

- What happens if we omit the activation function a? We could define:
- $\mathbf{P} = \mathbf{X} \cdot \boldsymbol{\Omega} + \boldsymbol{\beta}$ (**pre-activation** of the hidden layer)
- $\mathbf{F} = \mathbf{P} \cdot \boldsymbol{\Omega}' + \boldsymbol{\beta}'$ (**faulty** output logits).
- But then $\mathbf{F} = \mathbf{X} \cdot \mathbf{W} + \mathbf{b}$ for some other weights matrix $\mathbf{W}$ and bias vector $\mathbf{b}$! (See blackboard calculation.)
- So, if we omit a, two linear layers **collapse** to one linear layer.

Two-layer perceptrons describe piecewise linear functions

- To understand better what two-layer perceptrons do, consider a low-dimension example: $n = 1$, $d = 1$, $h = 3$, $q = 1$.
- The equations for the hidden layer activations are:

$$
\begin{aligned}
h_1 &= a(\theta_{10} + \theta_{11} x) \\
h_2 &= a(\theta_{20} + \theta_{21} x) \\
h_3 &= a(\theta_{30} + \theta_{31} x)
\end{aligned}
$$

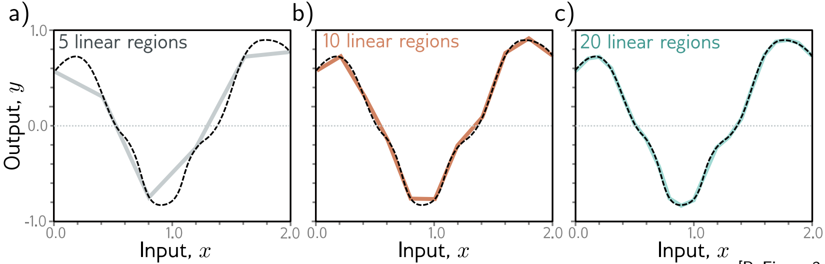
- We use the *ReLU* function for a: $a(y) = \max(y, 0)$.
- The equation for the output is:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3.$$

- (Check your understanding: What is this in matrix notation?)
- We can visualize the result ([P, 3.3a]).

Universal approximation theorem

- *Every* continuous function can be learned by a two-layer perceptron.



[P, Figure 3.5]

- "You might think of your neural network as being a bit like the C programming language. The language, like any other **modern** language, is capable of expressing any computable program. But actually coming up with a program that meets your specification is the hard part."
  [Z, Sec 5.1.1] (emphasis is mine).

Visualization with two input features

- We increase $d = 2$.
- Visualization ([P, 3.8]).
- In general, with h hidden features, we can realize $\approx 2^h$ different linear regions.

TP 4

- The goal of TP 4 is to implement a two-layer perceptron in NumPy and train it on a small dataset.
- You will hand it in via Moodle, it will count for 25% of the final grade.
- Deadline: Monday 3 February, 23h59.
- Please **read** and **follow** the TP instructions.

Today's lecture

Training vs. inference mode

- A neural network model has two *modes*: training and inference.
- In training mode, we try to improve the weights by looking at data.
- In inference mode, we *freeze* weights, and try to predict an output, typically for data that the network *has not seen during training*.
- We **separate** our data into three sets: training, validation, test.

Training overview

- One epoch of training:
  For every minibatch in the training set:
  1. *forward pass* the data through the entire model, compute *loss*,
  2. *backward pass* through the model, compute *gradient* of all parameters,
  3. *update* ("step") each parameter, using the gradients computed in the backward pass, and some *hyperparameters* (for example, learning rate).
- At the end of an epoch, compute the accuracy on the validation data.
- Repeat this for several epochs.
- At the end of training, compute the accuracy on the test data.
- For the accuracy computations, *use inference mode*.

Core training loop

```
for e in range(num_epochs):
    avg_loss = 0
    for (X_batch, y_batch) in batches(X_train, y_train):
        net.forward(X_batch)
        avg_loss += net.loss(y_batch) / num_batches
        net.backward()
        net.step()
    validation_accuracy = net.validate(X_val, y_val)
```

Backpropagation

Reflection

How can we calculate the gradient of the parameters?

- Last week: manual calculation on the blackboard.
- From now on: *backpropagation*.

Object-oriented design

- Each layer of the network is implemented as a “module” *class*.
- The API looks as follows (this is a slight simplification of PyTorch):

```
class MyModule:
    def __init__(self, params):
        """Initialize the layer's attributes,
        for example weight matrix."""
    def forward(self, x):
        """Forward pass through the module,
        using the data in x."""
    def backward(self, out_grad):
        """Backward pass the gradient in out_grad.
        Store parameter gradients and return in_grad."""
    def step(self):
        """Update parameters of the layer."""
```

Today's lecture

1. Recap
2. Multilayer perceptron (MLP)
3. Training and backpropagation
4. **Backpropagation**
5. Dropout

Backpropagation

- The chain rule says: if $Y = f(X)$ and $Z = g(Y)$, then

$$\frac{\partial Z}{\partial X} = \mathrm{prod}\left(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X}\right).$$

- Here, which prod we use depends on the **shapes** of X, Y, Z.
- In practice, we *store* useful information during the forward pass (as a class attribute) and we then *use* this information during the backward pass.

Backpropagation for a two-layer perceptron

- *Forward pass* on data **X** with labels **y**:

$$
\begin{aligned}
\mathbf{f} &= \mathbf{X}\boldsymbol{\Omega}_0 + \boldsymbol{\beta}_0 \\
\mathbf{h} &= a(\mathbf{f}) \\
\mathbf{o} &= \mathbf{h}\boldsymbol{\Omega}_1 + \boldsymbol{\beta}_1 \\
\mathbf{p} &= \mathrm{softmax}(\mathbf{o}) \\
\ell &= \mathrm{loss}(\mathbf{p}, \mathbf{y})
\end{aligned}
$$

Backpropagation for a two-layer perceptron

- *Backward pass*:
  - First compute the *loss gradient* $\dfrac{\partial \ell}{\partial \mathbf{o}}$ (see last week).
  - Pass the result to the second linear layer.
  - The gradient of $\mathbf{\Omega}_1$ is:

$$\frac{\partial \ell}{\partial \mathbf{\Omega}_1} = \left(\frac{\partial \ell}{\partial \mathbf{o}}\right)^{\mathrm{T}} \cdot \mathbf{h}$$

    (The value of $\mathbf{h}$ was *stored* during the forward pass.)
  - Pass the result backward through the ReLU layer (gradient: see last week), call the result `outgrad`.
  - The gradient of $\mathbf{\Omega}_0$ is:

$$\frac{\partial \ell}{\partial \mathbf{\Omega}_0} = (\texttt{outgrad})^{\mathrm{T}} \cdot \mathbf{X}.$$

    (The value of $\mathbf{X}$ was *stored* during the forward pass.)

Today's lecture

1. Recap
2. Multilayer perceptron (MLP)
3. Training and backpropagation
4. Backpropagation
5. Dropout

Dropout

- An efficient way for improving *generalization*.
- Idea: if we perturb the input, the network should still perform well.
- In practice: **turn off** a random set of connections for every minibatch.
- The hyperparameter `dropout` determines the fraction that is turned off.
- The other connections are **rescaled** by a factor $\dfrac{1}{1 - \texttt{dropout}}$.

TODO for next week

- TP4 (hand in via Moodle, 25% of the final grade)
- Read [P, Ch. 7] and [Z, Ch. 5]

References

[P] Prince, S., **Understanding Deep Learning**, https://udlbook.com

[Z] Zhang, A. et al., **Dive into Deep Learning**, https://www.d2l.ai