UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING

Introduction to ML
Lecture 2: Neural Networks

Hamed Tabkhi
Department of Electrical and Computer Engineering,
University of North Carolina Charlotte (UNCC)
htabkhiv@uncc.edu

UNC CHARLOTTE

Linear Regression Formulation

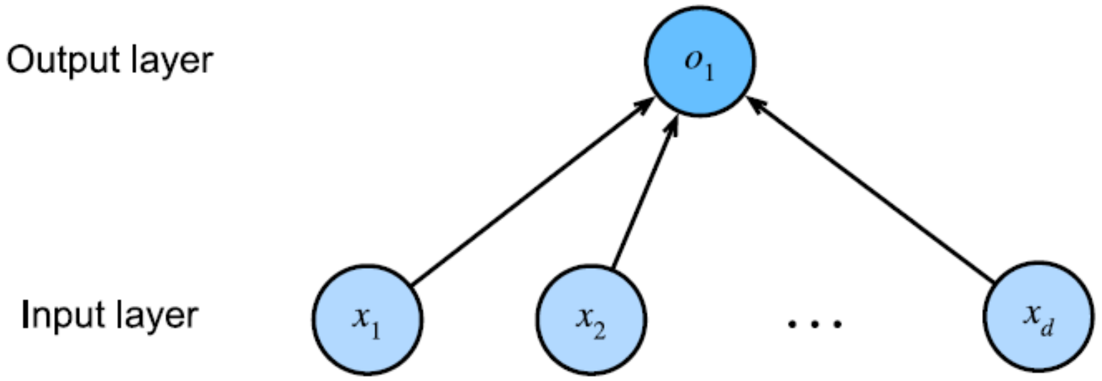
$$\hat{y} = w_1 x_1 + ... + w_d x_d + b.$$

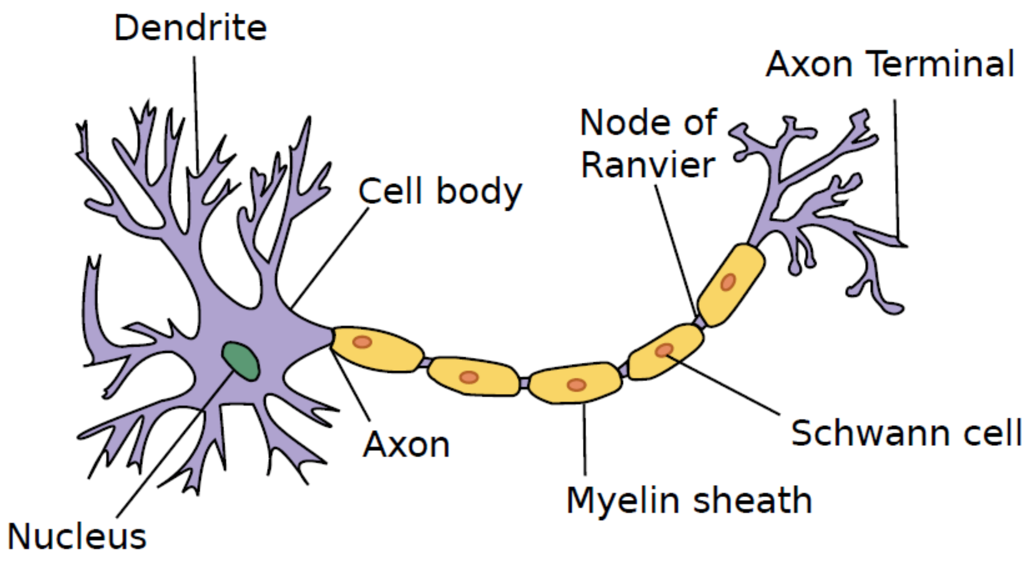
Collecting all features into a vector **x** and all weights into a vector **w**, we can
express our model compactl uct between **w** and **x**:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b.$$

Here, **X** contains one row for every example and one column for every feature.
For a collection of features **X** the predictions ^**y** \can be expressed via the matrix-vector product: $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b,$

An Artificial Neural Neuron





A Neural Linear Regression Implementation from Scratch

- We’re now ready to work through a fully functioning implementation of linear regression.
- We will implement the entire method from scratch, including:
  (i) the model;
  (ii) the loss function
  (iii) a minibatch stochastic gradient descent optimizer
  (iv) The training function that stitches all of these pieces together.

- While modern deep learning frameworks can automate nearly all of this work, implementing things from scratch is the only way to make sure that you really know what you are doing.
- Moreover, when it comes time to customize models, defining our own layers or loss functions, understanding how things work under the hood will prove handy.

Defining the Optimization Algorithm

- Linear regression has a closed-form solution.
- However, our goal here is to illustrate how to train more general neural networks, and that requires using minibatch SGD.

- A working example of SGD:
  1- At each step, using a minibatch randomly drawn from our dataset, we estimate the gradient of the loss with respect to the parameters.
  2- we update the parameters in the direction that may reduce the loss.

- Since our loss is computed as an average over the minibatch, we don't need to adjust the learning rate against the batch size.
- In later chapters we will investigate how learning rates should be adjusted for very large minibatches as they arise in distributed large scale learning.

Training

Now that we have all of the parts in place (parameters, loss function, model, and optimizer), we are ready to implement the main training loop. It is crucial that you understand this code well since you will employ similar training loops for every other deep learning model.
In each *epoch*:

1. we iterate through the entire training dataset, passing once through every example (assuming that the number of examples is divisible by the batch size).
2. In each iteration (minibatch size granularity), we grab a minibatch of training examples, and compute its loss through the model's training_step method.
3. Next, we compute the gradients with respect to each parameter.
4. Finally, we will call the optimization

- Initialize parameters $(\mathbf{w}, b)$
- Repeat until done
  - Compute gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$
  - Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

Concise Implementation of Linear Regression

- Deep learning has witnessed a Cambrian explosion of sorts over the past decade.
- The sheer number of techniques, applications and algorithms by far surpasses the progress of previous decades.
- This is due to a fortuitous combination of multiple factors, one of which is the powerful free tools offered by a number of open-source deep learning frameworks offering automatic differentiation and the convenience of Python.
- These frameworks allow us to automate and modularize the repetitive work of implementing gradient-based learning algorithms.
- In practice, because data iterators, loss functions, optimizers, and neural network layers are so common, modern libraries implement these components for us as well.
- In this section, we will show you how to implement the linear regression model from concisely by using high-level APIs of deep learning frameworks.

Concise Implementation: Loss Function and Optimization

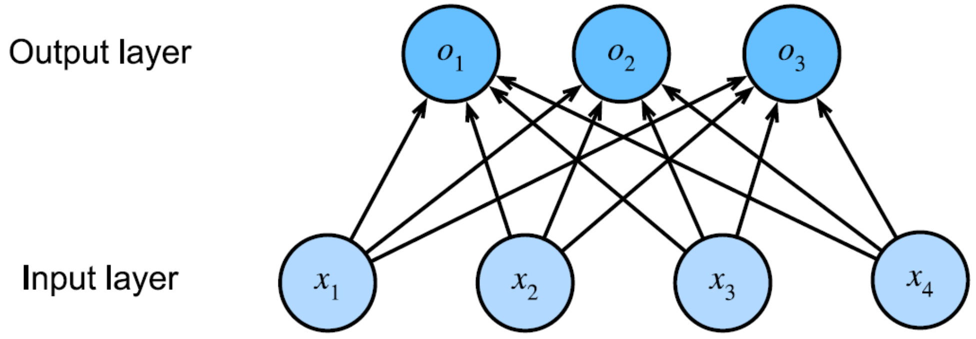
- The MSELoss class computes the mean squared error (without the 1/2 factor).
- By default, MSELoss returns the average loss over examples. It is faster (and easier to use) than implementing our own.

- Minibatch SGD is a standard tool for optimizing neural networks and thus PyTorch supports it in the optim module.

Classification

- To address classification with linear models, we will need as many affine functions as we have outputs.
- Strictly speaking, we only need one fewer, since the last category has to be the difference between 1 and the sum of the other categories but for reasons of symmetry we use a slightly redundant parametrization.
- Each output corresponds to its own affine function. In our case, since we have 4 features and 3 possible output categories, we need 12 scalars to represent the weights (*w* with subscripts), and 3 scalars to represent the biases (*b* with subscripts). This yields:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1,\\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2,\\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned}$$



UNC CHARLOTTE

Softmax

- Assuming a suitable loss function, we could try, directly, to minimize the difference between **o** and the labels **y**.
- While it turns out that treating classification as a vector-valued regression problem works surprisingly well, it is nonetheless lacking in the following ways:
  - There is no guarantee that the outputs o_i sum up to 1 in the way we expect probabilities to behave.
  - There is no guarantee that the outputs o_i are even nonnegative, even if their outputs sum up to 1, or that they do not exceed 1.

We can then transform these values so that they add up to 1 by dividing each by their sum. This process is called *normalization*. Putting these two pieces together gives us the ***softmax* function**:

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}.$$

- Note that the largest coordinate of **o** corresponds to the most likely class according to ^**y**.
- Moreover, because the softmax operation preserves the ordering among its arguments, we do not need to compute the softmax to determine which class has been assigned the highest probability.

UNC CHARLOTTE

Loss Function: Log-Likelihood

We can compare the estimates with reality by checking how probable the actual classes are according to our model, given the features:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}).$$

Since maximizing the product of terms is awkward, we take the negative logarithm to obtain the equivalent problem of minimizing the negative log-likelihood:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}),$$

where for any pair of label **y** and model prediction ^**y** over *q* classes, the loss function
is

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j .$$

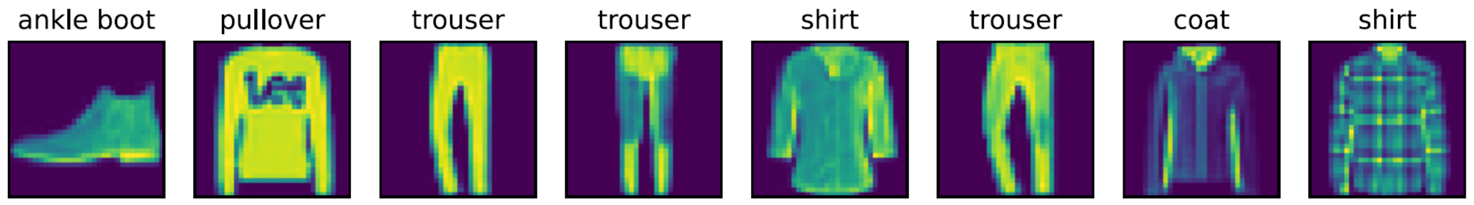
is commonly called the *crossentropy loss*.

The Image Classification Dataset

- Fashion-MNIST dataset (Xiao *et al.*, 2017), which was released in 2017.
- It contains images of 10 categories of clothing at 28 28 pixels resolution

- The categories of Fashion-MNIST have human-understandable names.
- The following convenience function converts between numeric labels and their names.

```
batch = next(iter(data.val_dataloader()))
data.visualize(batch)
```



Additional Notes

- In the validation_step we report both the loss value and the classification accuracy on a validation batch.

- By default we use a stochastic gradient descent optimizer, operating on minibatches, just as we did in the context of linear regression.
- The classification accuracy is the fraction of all predictions that are correct