3.1. Deep learning

> *In the mid-1980s two powerful new algorithms for fitting data became available: neural nets and decision trees. A new research community using these tools sprang up. Their goal was predictive accuracy. The community consisted of young computer scientists, physicists and engineers plus a few aging statisticians. They began using the new tools in working on complex prediction problems where it was obvious that data models were not applicable: speech recognition, image recognition, nonlinear time series prediction, handwriting recognition, prediction in financial markets.*
>
> Leo Breiman, *The Two Cultures*[20]

> *One of the biggest traps for smart engineers is optimizing a thing that shouldn't exist.*
>
> Elon Musk

Any number of deep learning tutorials and blog posts will tell you all about prediction accuracy as the be-all and end-all of machine learning — but hardly any even mention probabilistic learning of the sort we have been studying. Why?

The 'prediction accuracy' mindset is unhelpful and restrictive. It's unhelpful because it's made up of magic recipes and there's no guidance other than folk wisdom about how to choose them. It's restrictive because it's limited to the supervised learning setup, in which we're given labelled data, and it has nothing to say about training neural networks for unsupervised learning tasks such as clustering or generative modelling.

Instead, neural networks should be thought of as probability models. They should be evaluated by the log likelihood they ascribe to the dataset. Training a neural network for prediction accuracy *is* maximum likelihood estimation.

The probability modelling mindset, and the maximum likelihood interpretation, does demand a mental leap. The mindset of a probabilistic modeller is 'mildly unnatural' (as quoted on page v); and for simple problems in supervised learning it doesn't give us any concrete new techniques, only a new spin on old techniques. Perhaps this is why it's not more widely understood. It's only when we come to unsupervised learning in section 3.4 that the elegant simplicity of the probabilistic interpretation will become clear.

Neither the probabilistic interpretation nor the prediction accuracy mindset have anything to say about what should be *inside* the neural network. The *content* of a model depends entirely on the dataset that we want to fit the model to, and that is the proper place for folk wisdom and magic recipes. The *way* we model—the training mechanism, the evaluation metric, and so on—are what we're concerned with here.

In this section we'll describe the 'prediction accuracy' mindset. The probabilistic interpretation is in the next section.

THE 'PREDICTION ACCURACY' MINDSET

Suppose we have a labelled dataset $(x_1, y_1), \ldots, (x_n, y_n)$, where x_i is the input and y_i is the label for record i. The goal is to invent a function to predict the label, given the input. It doesn't matter at all what the function is, but typically it's a neural network with millions of edge-weight parameters. Write the prediction as $f(x_i\,;\,\theta)$, where θ is the set of parameters. 'Training the network' means choosing the parameters.

We don't want the neural network to cheat by just memorizing the dataset we give it, so we split the dataset into two parts. One part, the training data, we use to choose the θ parameters. The other part, the holdout data, we set aside until after training, and then we evaluate our neural network by measuring how good its predictions are on the holdout data. A sensible way to measure how good a prediction is is by defining a *prediction loss function*,

$$L\big(\text{true label},\ \text{prediction}\big) \qquad \text{where lower is better,}$$

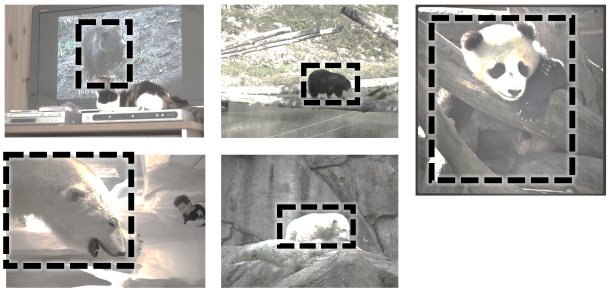
(we're free to invent whatever we think is useful for our task), and then reporting the average prediction loss over the holdout set,

$$\text{holdout loss} = \frac{1}{|\text{holdout}|} \sum_{i \in \text{holdout}} L\big(y_i,\ f(x_i\,;\,\theta)\big).$$

[20] Leo Breiman. "Statistical Modeling: The Two Cultures". In: *Statistical Science* (2001). URL: https://doi.org/10.1214/ss/1009213726.

There's an obvious way to train the neural network: simply pick θ to minimize the average loss on the training set. We can do this easily by computer, using a numerical optimization routine called gradient descent, so long as the loss $L(y,\ f(x;\theta))$ is a differentiable function of θ.

Example 3.1.1 (Regression / object detection).
The COCO dataset[21] consists of colour images of various sizes, each annotated with what it contains. The annotations contain, among other things, labels and bounding boxes. Here for example are some images that have one bear, with the bounding box shown.



One of the competitive tasks associated with this dataset is object detection — given an image, predict the coordinates of the bounding box.

In this example, the labels are values in $\mathbb{R}^4$: the coordinates of the box's center, and its width and height. A reasonable prediction loss function is squared Euclidean distance in $\mathbb{R}^4$,

$$L(\text{label}, \text{pred}) = \|\text{label} - \text{pred}\|^2.$$

In other words, our overall average loss is

$$\text{loss} = \frac{1}{|\text{dataset}|} \sum_i \|y_i - \hat{y}_i\|^2, \qquad \hat{y}_i = f(x_i\ ;\ \theta).$$

This is differentiable with respect to θ, as long as $f(x\ ;\ \theta)$ is made out of the standard neural network building blocks.

Example 3.1.2 (Image classification).
The MNIST dataset[22] consists of hand-written digits stored as 28×28 greyscale images, annotated with the digit they represent. Let $x_i \in \mathbb{R}^{28\times 28}$ be the ith image, and let $y_i \in \{0, 1, \ldots, 9\}$ be its label. The goal is to predict the label, given the image.

When there's a finite set of values that the labels can take, it's common to split the prediction into two steps. First let $f(x_i\ ;\ \theta)$ output a vector of 'confidence values',

$$\vec{f}(x_i\ ;\ \theta) = \big[f_1(x_i;\theta), \ldots, f_K(x_i;\theta)\big]$$

where K is the number of classes (writing $\vec{f}$ to remind ourselves that it returns a vector). Then, we simply take as our prediction the class with the highest confidence. All the cleverness is wrapped up inside the function $\vec{f}(x\ ;\ \theta)$. This can be as complicated a function as we like, and θ could consist of millions of unknown parameters. We'd like to make it expressive enough so that, once the best parameters have been chosen, $\vec{f}(x\ ;\ \theta)$ puts a large positive weight on the best guess for image x and a large negative weight on all the others.

Let the prediction loss function just count whether or not our prediction was correct:

$$L(\text{label}, \text{pred}) = 1_{\text{label}\neq\text{pred}}.$$

[21] Dataset at `https://cocodataset.org/`, described in Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *European Conference on Computer Vision* (2014)

[22] Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. *The MNIST Database of handwritten digits*. 1998. URL: `http://yann.lecun.com/exdb/mnist/`

Our overall loss can then be written as

$$\text{holdout loss} = \frac{1}{|\text{holdout}|} \sum_{i \in \text{holdout}} 1_{y_i \neq \arg\max_k f_k(x_i;\theta)}.$$

This loss isn't differentiable, because arg max isn't differentiable, so we can't use it for training with gradient descent. Instead, we pluck out of thin air a differentiable loss function

$$\text{training loss} = \frac{1}{|\text{training}|} \sum_{i \in \text{training}} \left(-\sum_{k=1}^{K} 1_{y_i=k} \log p_{ik} \right), \qquad p_{ik} = \frac{e^{f_k(x_i;\theta)}}{\sum_{\ell=1}^{K} e^{f_\ell(x_i;\theta)}}$$

and cross our fingers and hope to do well in evaluation. This loss function is called 'softmax cross-entropy loss with one-hot coding'. It's a waste of time trying to justify why we should use this particular loss function and not some other: the only real justification comes from ditching the 'prediction accuracy' mindset altogether and switching to the probabilistic interpretation. Nevertheless, here's some time wasting. This loss function has a contribution $-\log p_{iy_i}$ from record i, so to make the loss small we want p_{iy_i} to be large. To make this large, we want $f_k(x_i\,;\,\theta)$ to be larger for $k = y_i$ than for $k \neq y_i$. This makes $y_i = \arg\max_k f_k(x_i\,;\,\theta)$. In conclusion, making this training loss small should have the effect of making the 'real' loss function small.

Incidentally, the training loss function as we've written it here has a term $\sum_k 1_{y_i=k} \log p_{ik}$. This is just an algebraic trick to say "only keep the p_{ik} term where $k = y_i$", in other words it's just $\log p_{iy_i}$. When we're aiming for fast code, a lookup like p_{iy_i} isn't helpful whereas a matrix multiplication like $\sum_k 1_{y_i=k} \log p_{ij}$ is better.