1 Neural Networks as Distribution Transformers

1.1 Introduction

So far we have seen that deep nets are stacks of simple functions, which compose to achieve interesting mappings from inputs to outputs. This section will introduce a slightly different way of thinking about deep nets. The idea is think of each layer as a *geometric transformation of a data distribution*.

1.2 A Different Way of Plotting Functions

Each layer in a deep net is a mapping from one representation of the data to another: $f : \mathbf{x}_{\mathtt{in}} \rightarrow \mathbf{x}_{\mathtt{out}}$. If $\mathbf{x}_{\mathtt{in}}$ and $\mathbf{x}_{\mathtt{out}}$ are both one-dimensional (1D), then we can plot the mapping as a function with $\mathbf{x}_{\mathtt{in}}$ on the x-axis and $\mathbf{x}_{\mathtt{out}}$ on the y-axis (figure 1.1).

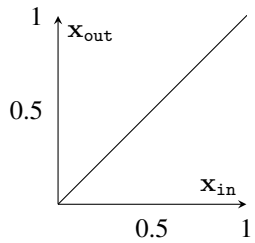

Figure 1.1: The traditional way of plotting the function $\mathbf{x}_{\mathtt{out}} = \mathbf{x}_{\mathtt{in}}$.

Now, we will instead consider a different way of plotting the mapping, where we simply rotate the y-axis to be horizontal rather than vertical (figure **??**).

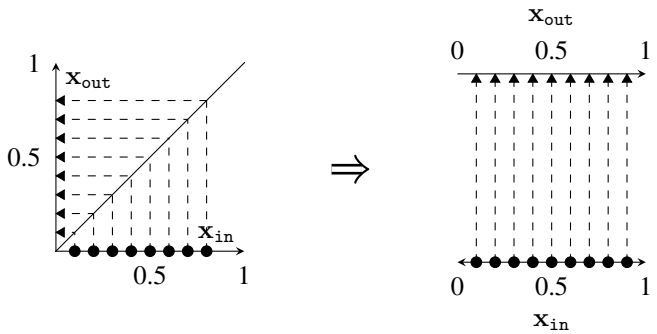

Figure 1.2: An alternative way of plotting a function (right). Functions are mappings that rearrange the input space. The identity function $\mathbf{x}_{\mathtt{out}} = \mathbf{x}_{\mathtt{in}}$, shown here, means "no rearrangement," so the mapping is straight lines.

The depiction to the right makes it obvious that the plot $\mathtt{x}_{\mathtt{out}} = \mathtt{x}_{\mathtt{in}}$ is the identity mapping: datapoints get mapped to unchanged positions. Figure **??** shows a few more mappings plotted in this way.

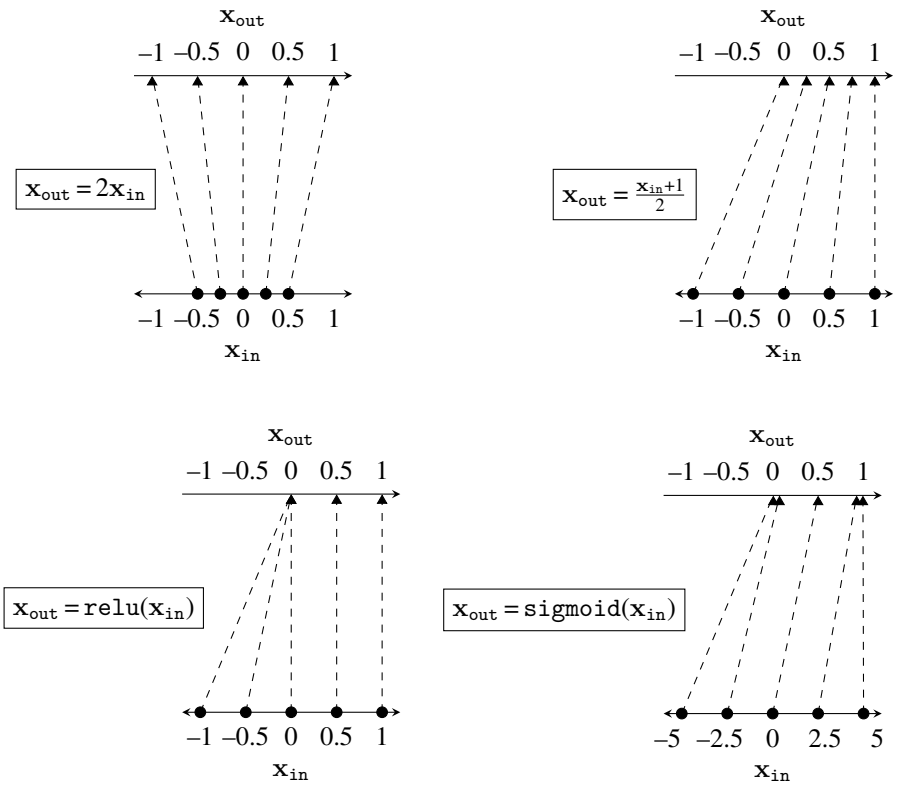

Figure 1.3: Mapping plots for several simple functions that could be neural layers.

Each of the above are layers that could be found in a deep net. Linear layers, like those in the top row above, stretch and squash the data distribution. The `relu` nonlinearity maps all negative data to 0, and applies an identity map to all nonnegative data. The `sigmoid` function pulls negative data to 0 and positive data to 1.

1.3 How Deep Nets Remap a Data Distribution

In this way, an incoming data distribution can be reshaped layer by layer into a desired configuration. The goal of a binary softmax classifier, for example, is to move the datapoints around until all the class 0 points end up moved to [1,0] on the output layer and all the class 1 end up moved to [0,1].

A deep net stacks these operations; an example is given in figure 1.4.

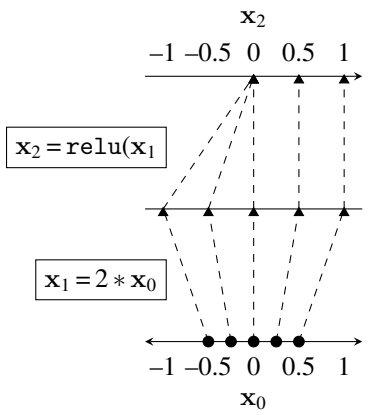

Figure 1.4: Mapping plot for a `linear-relu` stack.

The plots above show how a uniform grid of datapoints get mapped from layer to layer in a deep net. We can also use this plotting style to show how a nonuniform distribution of incoming datapoints gets transformed. This is the setting in which deep nets actually operate, and sometimes the real action of the network looks very different when viewed this way. We can think of a deep net as transforming an input data distribution, $p_{\texttt{data}}$, into an output data distribution, $p_{\texttt{out}}$. Each layer of activations in a network is a different representation or **embedding** of the data, and we can consider the distribution of activations on some layer ℓ to be p_ℓ. Then, layer by layer, a deep net transforms $p_{\texttt{data}}$ into p_1 into p_2, and so on until finally transforming the data to the distribution $p_{\texttt{out}}$. Most loss functions can also be interpreted from this angle: they penalize the divergence, in one form or another, between the output distribution $p_{\texttt{out}}$ and a target distribution $p_{\texttt{target}}$.

A nice property of this way of plotting is that it also extends to visualizing two-dimensional (2D)-to-2D mappings (something that conventional x-axis/y-axis plotting is not well equipped to do). Real deep nets perform N-dimensional (ND)-to-ND mappings, but already 2D-to-2D visualizations can give a lot of insight into the general case. In figure 1.5, we show how three common neural net layers may act to transform a Gaussian blob of data centered at the origin.

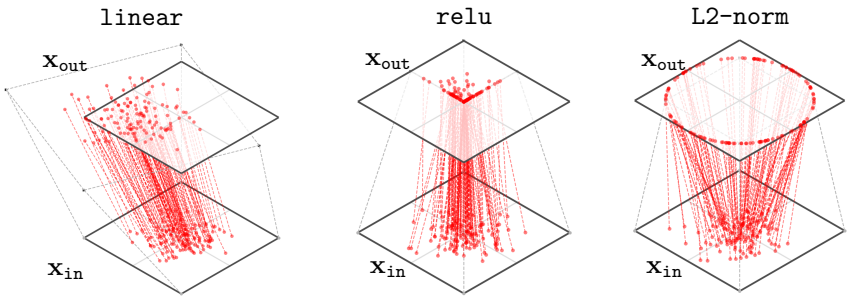

Figure 1.5: 2D mapping diagrams for several neural layers.

One interesting thing to notice here is that the `relu` layer maps many points to the axes of the positive quadrant. In general, with `relu`-nets, a lot of data density will build up along these axes, because any point *not* in the positive quadrant snaps to an axis. This effect becomes exaggerated in real networks with high-dimensional embeddings. In particular, for a width-N layer, the region that is strictly positive occupies only $\frac{1}{2^N}$ proportion of the embedding space, so almost the entire space gets mapped to the axes after a `relu` layer. The geometry of high-dimensional neural representations may become very sparse because of this, where most of the volume of representational space is not occupied by any datapoints.

1.4 Binary Classifier Example

Consider a multilayer perceptron (MLP) that performs binary classification formulated as two-way softmax regression. The input datapoints are each in $\mathbb{R}^2$ and the target outputs are in $\triangle^1$ (the one-simplex), with layer structure `linear-relu-linear-relu-linear`. This network is drawn below (figure 1.6).

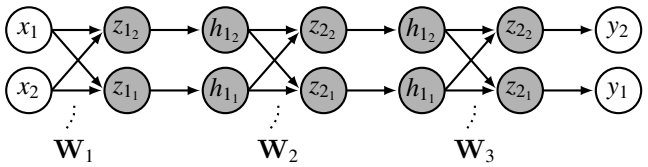

Figure 1.6: An MLP with three linear layers and two outputs, suitable for performing binary softmax regression.

Or expressed in math as follows:

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 \quad \triangleleft \ \texttt{linear} \tag{1.1}$$

$$\mathbf{h}_1 = \texttt{relu}(\mathbf{z}_1) \quad \triangleleft \ \texttt{relu} \tag{1.2}$$

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2 \quad \triangleleft \ \texttt{linear} \tag{1.3}$$

$$\mathbf{h}_2 = \texttt{relu}(\mathbf{z}_2) \quad \triangleleft \ \texttt{relu} \tag{1.4}$$

$$\mathbf{z}_3 = \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3 \quad \triangleleft \ \texttt{linear} \tag{1.5}$$

$$\mathbf{y} = \texttt{softmax}(\mathbf{z}_3) \quad \triangleleft \ \texttt{softmax} \tag{1.6}$$

Now we wish to train this net to classify between two data distributions. The goal is to transform the input distribution into a target distribution that separates the classes, as shown in figure 1.7.

In this example, the target output places all the red dots at $(0, 1)$ and all the blue dots at $(1, 0)$. These are the coordinates of one-hot codes for our two classes. Training the network consists of find the series of geometric transformations that rearrange the input distribution to this target output distribution.

A checkpoint is a record of the parameters at some iteration of training, that is, if iterates of the parameter vector are $\theta^0, \theta^1, \ldots, \theta_T$, then any θ^k can be recorded as a checkpoint.

We will visualize how the net transforms the training dataset, layer by layer, at four **checkpoints** over the course of training. In figure 1.8, we plot this as a 3D visualization of $\mathbb{R}^2 \to \mathbb{R}^2$ mappings. Each dotted line connects the representation of a datapoint at one layer to its representation at the next. The gray lines show, for each layer, how a square region

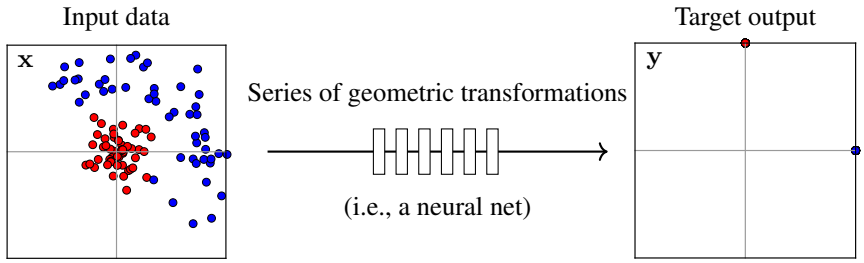

Figure 1.7: The goal of a neural net classifier is to rearrange the input data distribution to match the target label distribution. (left) Input dataset with two classes in red and blue. (right) Target output (one-hot codes).

around the origin gets stretched, rotated, or squished to map to a transformed region on the next layer.

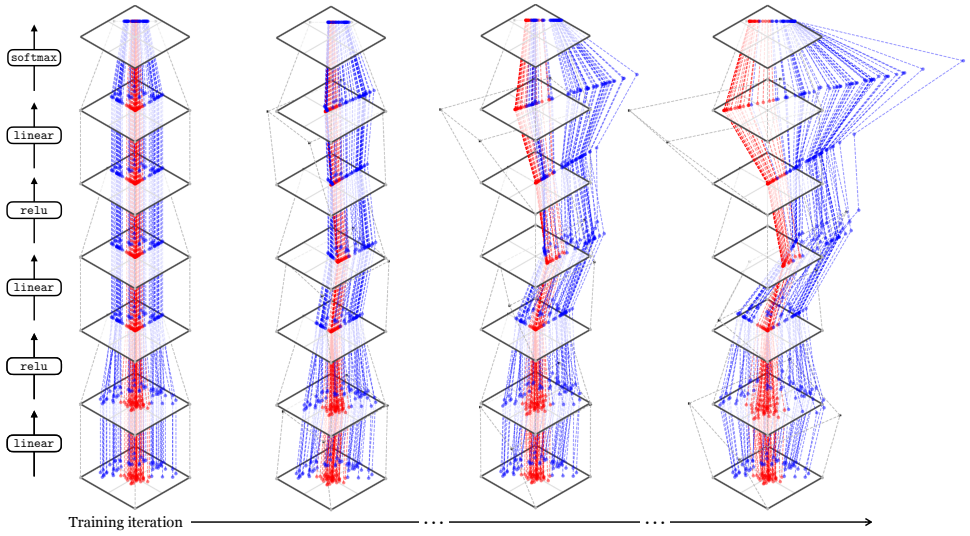

Figure 1.8: How a deep net remaps input data layer by layer. The target output is to move all the red points to $(0, 1)$ and all the blue points to $(1, 0)$ (one-hot codes for the two classes). As training progresses the network gradually achieves this separation.

Of course, "stretched, rotated, or squished" is just an intuitive way to think about it, but we can be more precise. The linear layers perform an affine transformation of the space, while the relus project all negative values to the boundaries of the cone whose dimensions are strictly positive. Layer by layer, over the course of training, the net learns to disentangle these two classes and pull the points toward vertices of the one-simplex, achieving a correct classification of the points!

1.5 How High-Dimensional Datapoints Get Remapped by Deep Net

What if our data and activations are high-dimensional? The plots above only can visualize 1D and 2D data distributions. Deep representations are typically much higher-dimensional than this, and to visualize them, we need to apply tools from **dimensionality reduction**. These tools project the high-dimensional data to a lower dimensionality, for example 2D,

which can be visualized. A common objective is to perform the projection such that the distance between two datapoints in the 2D projection is roughly proportional to their actual distance in the high-dimensional space. In the next plot, figure 1.9, we use a dimensionality reduction technique called **t-Distributed Stochastic Neighbor Embedding t-SNE** [2] to visualize how different layers of a modern deep net represent a dataset of images of different semantic classes, where each color represents a different semantic class. The network we are visualizing is of the transformer architecture, and we will learn about all its details in chapter **??**. For now, the important things to note are that (1) we are only showing a selected few of the layers (this net actually has dozen of layers) and (2) the embeddings are high-dimensional (in particular, they are 38,400-dimensional) but mapped down to 2D by t-SNE. Therefore, this visualization is just a rough view of what is going on in the net, unlike the visualizations in the previous section, which showed the exact embeddings and every single step of the layer-to-layer transformations.

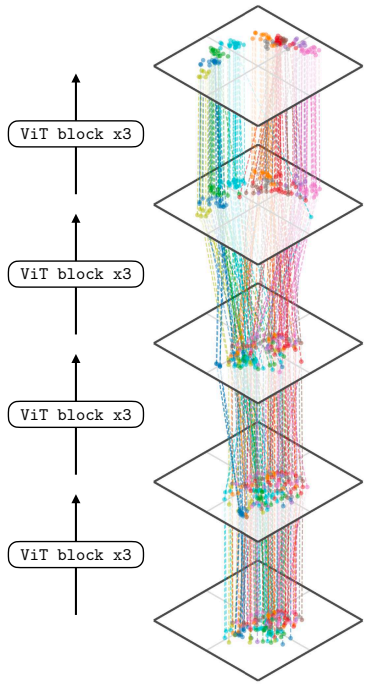

Figure 1.9: How a powerful deep net remaps input images into a disentangled representation where semantic classes (colors) are separated. This deep net is a vision transformer (ViT [1]), which we will learn about in section **??**. It was trained using contrastive language-image pre-training (CLIP [3], see section **??**). Each `ViT block` contains multiple layers of neural processing (see figure **??**; we visualize the embeddings right after the first `token norm` in a block). We apply t-SNE jointly across all shown layers.

Notice that on the first layer, semantic classes are not well separated but by the last layer the representation has **disentangled** the semantic classes so that each class occupies a different part of representational space. This is expected because the final layer is near the output of the network, and this network has been trained to output a direct representation of semantics (in particular, this net was trained with contrastive language-image pre-training [CLIP [3]], which is a method for learning semantic representations that we will learn about in section **??**).

1.6 Concluding Remarks

Layer by layer, deep nets transform data from its raw format to ever more abstracted and useful representations. It can be useful to think about this process as a set of geometric transformations of a data distribution, or a kind of disentangling where initially messy data gets reorganized so that different data classes are cleanly separated.

References

[1] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *ICLR* (2021).

[2] Laurens van der Maaten and Geoffrey Hinton. “Visualizing Data Using t-SNE”. In: *Journal of Machine Learning Research* 9.Nov (2008), pp. 2579–2605.

[3] Alec Radford et al. “Learning Transferable Visual Models from Natural Language Supervision”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 8748–8763.