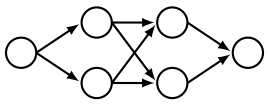
Chapter 1

Neural nets

Draft chapter from Torralba, Isola, Freeman

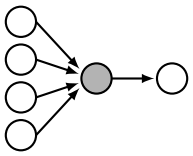
Neural nets are functions loosely modeled on the brain. In the brain, we have billions of neurons that connect to one another. Each neuron can be thought of as a node in a graph, and the edges are the connections from one neuron to the next. The edges are directed – electrical signals propagate in just one direction along the wires in the brain:



Outgoing edges are called axons and incoming edges are called dendrites. A neuron "fires", sending a pulse down its axon, when the incoming pulses, from the dendrites, exceed a threshold.

1.1 The perceptron: a simple model of a single neuron

Let's consider a neuron, shaded in gray, that has four inputs and one output:



A simple model for this neuron is the **perceptron**. A perceptron is a neuron with N inputs $\{x_i\}_{i=1}^N$ and one output y, that maps inputs to outputs according to the following equations:

$$z = f(\mathbf{x}) = \sum_{i=1}^{N} w_i x_i + b = \mathbf{w}^T\mathbf{x} + b \qquad \triangleleft \text{ linear layer} \tag{1.1}$$

$$g(z) = \begin{cases} 1, & \text{if} \quad z > 0 \\ 0, & \text{otherwise} \end{cases} \qquad \triangleleft \text{ activation function} \tag{1.2}$$

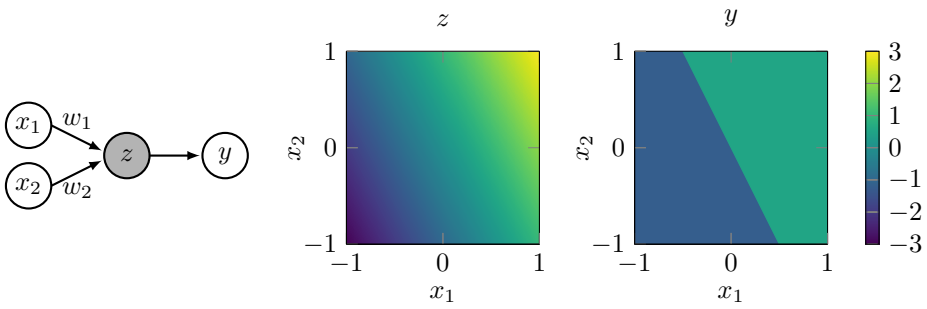
$$y = g(f(\mathbf{x})) \qquad \triangleleft \text{ perceptron} \tag{1.3}$$

In words, we take a weighted sum of the inputs and, if that sum exceeds a threshold (here 0), the neuron fires (outputs a 1). The function f is called a **linear layer** because it computes a linear function of the inputs, $\mathbf{w}^T\mathbf{x}$, plus a bias, b. The function g is called the **activation function** because it decides whether the neuron "activates" (fires).

Mathematically, f is an affine function, but by convention we call it a "linear layer". One way to think of it is f is a linear function of $[\mathbf{x}, 1]$.

1.1.1 The perceptron as a classifier

People got excited about perceptrons in the late 1950s because it was shown that they can learn to classify data [Rosenblatt, 1957]. Let's see how that works. We will consider a perceptron with two inputs, x_1 and x_2, and one output, y. Let the incoming connection weights be $w_1 = 2$, $w_2 = 1$, and $b = 0$. z and y, as a function of x_1 and x_2, look like this:



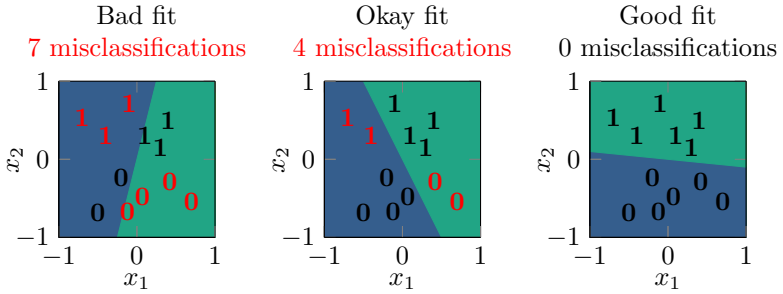
y takes on values 0 or 1, so you can think of this as a classifier that assigns a class label of 1 to the upper-right half of the plotted region.

1.1.2 Learning with a perceptron

So a perceptron acts like a classifier, but how can we use it to learn? The idea is that given data, $\{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^N$, we will adjust the weights $\mathbf{w}$ and the bias b, in order to minimize a classification loss, $\mathcal{L}$:

$$\mathbf{w}^*, b^* = \arg\min_{\mathbf{w},b} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\mathbf{w}^T \mathbf{x}^{(i)} + b, y^{(i)}) \tag{1.4}$$

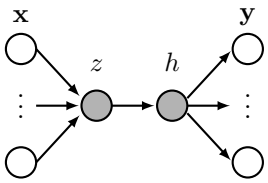
In the picture above, this optimization process corresponds to shifting and rotating the classification boundary, until you find a line that separates data labeled as $y = 0$ from data labeled as $y = 1$:



You might be wondering, what's the exact optimization algorithm that will find the best line that separates the classes? The original perceptron paper proposed one particular algorithm, the "perceptron learning algorithm". This was an optimizer tailored to the specific structure of the perceptron. Older papers on neural nets are full of specific learning rules for specific architectures: the "delta rule", the "Rescorla-Wagner" model, and so forth. Nowadays we rarely use these special-purpose algorithms. Instead, we use *general-purpose* optimizers like gradient descent (for differentiable objectives) or zero-th order methods (for non-differentiable objectives). The next chapter will cover the backpropagation algorithm, which is a general-purpose gradient-based optimizer that applies to essentially all neural nets we will see in this book (but, note that for the perceptron objective, because it has a non-differentiable threshold function, we would instead opt for a zero-th order "blackbox" optimizer).

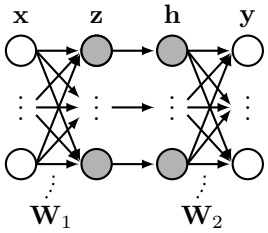
1.2 Multilayer perceptrons

Perceptrons can solve linearly separable binary classification problems, but they are otherwise rather limited. For one, they only produce a single output. What if we want multiple outputs? We can achieve this by adding edges that "fan out" after the perceptron:



This network maps an input **layer** of data $\mathbf{x}$ to a layer of outputs $\mathbf{y}$. The neurons in between inputs and outputs are called **hidden units**, shaded in gray. Here, z is a **pre-activation** hidden unit and h is a **post-activation** hidden unit, i.e. $h = g(z)$ where $g(\cdot)$ is an activation function like in Eqn. 1.2.

More commonly we might have many hidden units in stack, which we call a **hidden layer**:



How many layers does this net have? Some texts will say two [$\mathbf{W}_1$, $\mathbf{W}_2$], others three [$\mathbf{x}$, $\{\mathbf{z}, \mathbf{h}\}$, $\mathbf{y}$], others four [$\mathbf{x}$, $\mathbf{z}$, $\mathbf{h}$, $\mathbf{y}$]. We must get comfortable with the ambiguity.

Because this network has multiple layers of neurons, and because each neuron in this net acts as a perceptron, we call it a **multilayer perceptron**, or **MLP**. The equation for this MLP is:

$$\mathbf{z} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 \quad \triangleleft \text{ linear layer} \tag{1.5}$$

$$\mathbf{h} = g(\mathbf{z}) \quad \triangleleft \text{ activation function} \tag{1.6}$$

$$\mathbf{y} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2 \quad \triangleleft \text{ linear layer} \tag{1.7}$$

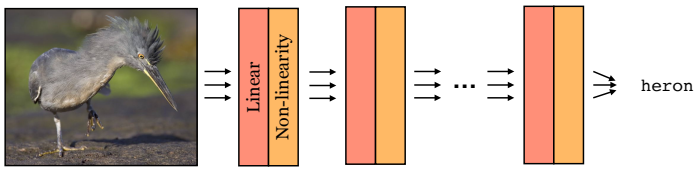
In general, MLPs can be constructed with any number of layers following this pattern: linear layer, activation function, linear layer, activation function, ...

The activation function g could be the threshold function like in Eqn. 1.2, but more generally it can be any pointwise nonlinearity, that is, $g(\mathbf{h}) = [\tilde{g}(h_1), \ldots, \tilde{g}(h_N)]$ and $\tilde{g}$ is any nonlinear function mapping $\mathbb{R} \to \mathbb{R}$.

Beyond MLPs, this kind of sequence – linear layer, pointwise nonlinearity, linear layer, pointwise nonlinearity, and so on – is the prototpyical motif in almost all neural networks, including most we will see later in this book.

1.3 Deep nets

Deep nets are neural nets that stack the above motif many times:



Each layer is a function. Therefore, a deep net is a composition of many functions:

$$f(\mathbf{x}) = f_L(f_{L-1}(\dots f_2(f_1(\mathbf{x})))) \tag{1.8}$$

L is the number of layers in the net

These functions are parameterized by weights $[\mathbf{W}_1, \dots, \mathbf{W}_L]$ and biases $[\mathbf{b}_1, \dots, \mathbf{b}_L]$. Some layers we will see later have other parameters. Collectively, we will refer to the concatenation of all the parameters in a deep net as θ.

Deep nets are powerful because they can perform nonlinear mappings. In fact, a deep net with sufficiently many neurons can fit almost any desired function arbitrarily closely. The **universal approximation theorem** [Cybenko 1989] states that this is true even for a network with just a single hidden layer. The caveat is that the number of neurons in the hidden layers will have to be very large in order to fit complicated functions. Also, technically, this theorem only holds for continuous functions on compact subsets of $\mathbb{R}^N$ – for example a neural net cannot fit non-computable functions.

1.3.1 Activations vs parameters

When working with deep nets it's useful to distinguish *activations* and *parameters*. The "activations" are the values that the neurons take on, $[\mathbf{x}, \mathbf{z}_1, \mathbf{h}_1, \dots, \mathbf{z}_{L-1}, \mathbf{h}_{L-1}, \mathbf{y}]$ – slightly abusing notation, we use this term for both pre-activation function neurons and post-activation function neurons. The activations are the neural representations of the data being processed. Often, we will not worry about distinguishing between inputs, hidden units, and outputs to the net, and simply refer to all data and neural activations in a network, layer by layer, as a sequence $[\mathbf{x}_0, \dots, \mathbf{x}_L]$, in which case $\mathbf{x}_0$ is the raw input data.

Parameters, on the other hand, are the weights and biases of the network. These are the variables being learned. Both data and parameters are tensors of variables.

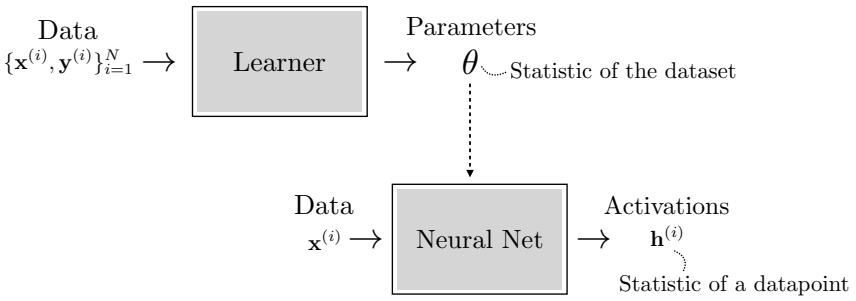
Often we think of a layer as a function $\mathbf{x}_{l+1} = f_{l+1}(\mathbf{x}_l)$, but we can also make the parameters explicit and think of each layer as a function:

$$\mathbf{x}_{l+1} = f_{l+1}(\mathbf{x}_l, \theta_{l+1}) \tag{1.9}$$

That is, each layer takes the activations from the previous layer, as well as parameters of the current layer as input, and produces activations of the next layer. Varying either the input activations or the input parameters will affect the output of the layer. From this perspective, anything we can do with parameters, we can do with activations instead, and vice versa, and that is the basis for a lot of applications and tricks. For example, while normally we learn the values of the parameters, we could instead hold the parameters fixed and learn the values of the activations that achieve some objective. In fact this is what is done in applications such as style transfer, adversarial attacks, and network visualization, which we will see in more detail in later chapters.

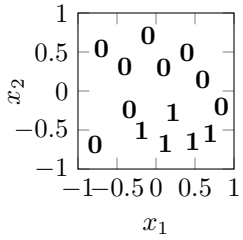
1.3.2 Fast activations and slow parameters

So what's different about activations versus parameters? One way to think about it is that activations are *fast* functions of a data*point*: they are the result of a few layers of processing this datapoint. Parameters are *also* functions of the data – they are learned from data – but they are *slow* functions of data*sets*: the parameters are arrived at via an optimization procedure over a whole dataset. So, both activations and parameters are statstics of the data, i.e. information extracted about about the data that organizes or summarizes it. The parameters are a kind of meta-summary since they specify a functional transformation that produces activations from data, and activations themselves are a summary of the data. It looks like this:



1.3.3 Deep nets can perform nonlinear classification

Let's return to our binary classification problem from above, but now make the two classes not linearly separable:



Here there is no line that can separate the zeros from the ones. Nonetheless, we will demonstrate a multilayer network that can solve this problem. The trick is to just add more layers!

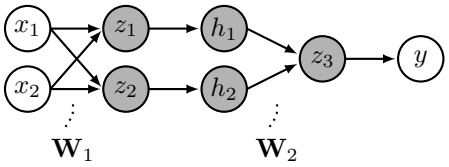


Figure 1.1: A simple MLP network.

Consider using the following settings for $\mathbf{W}_1$ and $\mathbf{W}_2$:

$$\mathbf{W}_1 = \begin{bmatrix} -1 & 1 \\ 1 & 2 \end{bmatrix}, \qquad \mathbf{W}_2 = \begin{bmatrix} 1 & -1 \end{bmatrix} \tag{1.10}$$

The full net then performs the following operation:

$$z_1 = x_1 - x_2, \quad z_2 = 2x_1 + x_2 \qquad \triangleleft \texttt{ linear} \tag{1.11}$$

$$h_1 = \max(z_1, 0), \quad h_2 = \max(z_2, 0) \qquad \triangleleft \texttt{ relu} \tag{1.12}$$

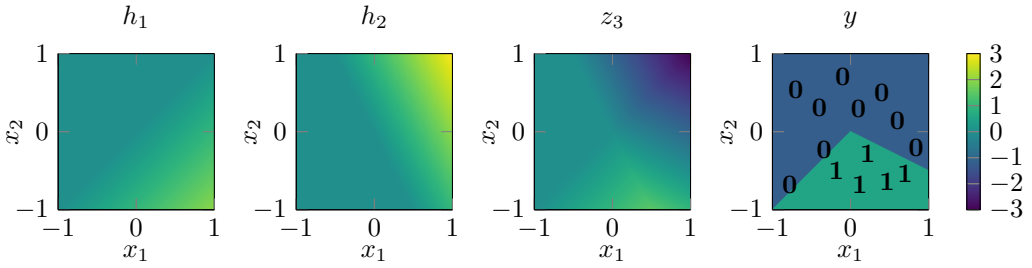
$$z_3 = h_1 - h_2 \qquad \triangleleft \texttt{ linear} \tag{1.13}$$

$$y = 1(z_3 > 0) \qquad \triangleleft \texttt{ threshold} \tag{1.14}$$

Here we have introduced a new pointwise nonlinearity, the `relu` function, which is like a graded version of the `threshold` function we saw above, and performs better in practice.

Let's visualize the values that the neurons take on as a function of x_1 and x_2:

As can be seen in the rightmost plot above, at the output y, this neural net successfully assigns a value of 1 to the region of the dataspace where the data points labeled as 1 live. This example demonstrates that is possible to solve nonlinear classification problems with a deep net. In practice, we would want to *learn* the parameter settings that achieve this



classification. One way to do so would be to enumerate all possible parameter settings and pick one that successfully separates the 0's from the 1's. This kind of exhaustive enumeration is a slow process, but don't worry, in later chapters we will see how to speed things up using methods from optimization (in particular, gradient descent). But it's worth remarking that enumeration is always a sufficient solution, at least when possible parameter values form a finite set.

1.3.4 Learning with deep nets

Deep learning refers to learning with deep nets. Using the formalism we defined in Chapter **??**, learning consists of using an *optimizer* to find a function in a *hypothesis space*, that maximizes an *objective*. From this perspective, neural nets are simply a special kind of hypothesis space.

Deep learning also usually involves using a particular optimization algorithm called backpropagation, which we will see in the next chapter. However, it is certainly possible to optimize neural nets with other methods. Indeed, it is not clear if biological neural nets actually use backprogation. Alternative optimizers include genetic algorithms, simulated annealing, and various other kinds of "random search". There is also a broad literature on "bottom up learning rules" that do not explicitly optimize any objective (though they may implicitly). One such rule, called **Hebbian learning**, is "fire together, wire together", that is, we increase the weight of the connection between two neurons whenever the two neurons are active at the same time.

1.4 Catalogue of layers

Below, we use the color blue to denote **parameters** and the color red to denote **data/activations** (inputs and outputs to each layer).

Linear layers Linear layers are the workhorses of deep nets. Almost all parameters of the network are contained in these layers – we call these parameters the weights and biases. We have already introduced linear layers above. They look like this:

$$\mathbf{x}_{\text{out}} = \mathbf{W}\mathbf{x}_{\text{in}} + \mathbf{b} \qquad \triangleleft \texttt{linear} \tag{1.15}$$

Activation layers If a net only contained linear layers then it could only compute linear functions. This is because the composition of N linear functions is a linear function. Activation layers add nonlinearity. Activation layers are typically pointwise functions, applying a scalar to scalar mapping on each dimension of the input vector. Typically parameters of these layers, if any, are not learned (but they can be). Some common activation layers are

defined below:

$$x_{\text{out}_i} = \begin{cases} 1, & \text{if} \quad x_{\text{in}_i} > 0 \\ 0, & \text{otherwise} \end{cases} \qquad \triangleleft \ \texttt{threshold} \tag{1.16}$$

$$x_{\text{out}_i} = \frac{1}{1 + e^{-x_{\text{in}_i}}} \qquad \triangleleft \ \texttt{sigmoid} \tag{1.17}$$

$$x_{\text{out}_i} = 2 * \texttt{sigmoid}(2 * x_{\text{in}_i}) - 1 \qquad \triangleleft \ \texttt{tanh} \tag{1.18}$$

$$x_{\text{out}_i} = \max(x_{\text{in}_i}, 0) \qquad \triangleleft \ \texttt{relu} \tag{1.19}$$

$$x_{\text{out}_i} = \begin{cases} \max(x_{\text{in}_i}, 0), & \text{if} \quad x_{\text{in}_i} \geq 0 \\ a \min(x_{\text{in}_i}, 0), & \text{otherwise} \end{cases} \qquad \triangleleft \ \texttt{leaky-relu} \tag{1.20}$$

Normalization layers Normalization layers add another kind of nonlinearity. Instead of being a pointwise nonlinearity, like in activation layers, they are nonlinearities that perturbs each neuron based on the collective behavior of a set of neurons. Let's start with the example of **batch normalization** [Ioffe and Szegedy 2015].

Recall from statistics that the standard score of a draw of a random variable is how many standard deviations it differs from the mean: $z = \frac{x-\mu}{\sigma}$.

Batchnorm **standardizes** each neural activation with respect to its mean and variance over a batch of datapoints. Mathematically:

$$x_{\text{out}_i} = \gamma \frac{x_{\text{in}_i} - \mathbb{E}[x_{\text{in}_i}]}{\sqrt{\texttt{Var}[x_{\text{in}_i}]}} + \beta \qquad \triangleleft \ \texttt{batchnorm} \tag{1.21}$$

γ and β are learned parameters of this layer that maintain expressivity so that the layer can output values with non-zero mean and non-unit variance. Most commonly batchnorm is applied using training batch statistics to compute the mean and variance, which change batch to batch. At test time, aggregate statistics from the training data are used. However, using test batch statistics can be useful for achieving invariance to changes in the statistics from training data to test data [Wang et al. 2020].

There are numerous other normalization layers that have been defined over the years. One more that we will highlight is L_2 **normalization**:

$$x_{\text{out}_i} = \frac{x_{\text{in}_i}}{\|\mathbf{x}_{\text{in}}\|_2} \qquad \triangleleft \ \texttt{L2-norm} \tag{1.22}$$

This operation projects the inputs onto the unit hypersphere – quite a nice trick.

Output layers The last piece we need is a layer that maps a neural representation – a high-dimensional array of floating point numbers – to a desired output representation. In classification problems, the desired output is a class label, and the most common output operation is the softmax function, which we have already encountered in previous chapters. In image synthesis problems, the desired output is typically a 3D array with dimensions $N \times M \times 3$, and values in the range $[0, 255]$. A sigmoid multiplied by 255 is a typical output transformation for this setting. The equations for these two layers are:

$$x_{\text{out}_i} = \frac{e^{-\tau x_{\text{in}_i}}}{\sum_{k=1}^{K} e^{-\tau x_{\text{in}_k}}} \qquad \triangleleft \ \texttt{softmax} \tag{1.23}$$

$$x_{\text{out}_i} = 255 * \texttt{sigmoid}(x_{\text{in}_i}) \qquad \triangleleft \ \text{common layer for image output problems} \tag{1.24}$$

In the softmax definition we have added a **temperature** parameter τ, which is commonly used to scale how peaky, or confident, the predictions are.

The output layer is the input to the loss function, thus completing our specification of the deep learning problem. However, to use the outputs in practice requires translating them into actual pictures, or actions, or decisions. For a classification problem, this might mean taking the argmax of the softmax distribution, so that we can report a single class. For image prediction problems, it might mean rounding each output to an integral value since common image formats represent RGB values as integers.

There are of course many other output transformations you can try. Often, they will be very problem specific since they depend on the structure of the output space you are targeting.

1.5 Data structures for deep learning: tensors and batches

The main data structure that we will encounter in deep learning is the **tensor**, which is just a multi-dimensional array. This may seem simple, but it's important to get comfortable with the conventions of tensor processing.

In general, everything in deep learning is represented as tensors – the input is one tensor, the activations are tensors, the weights are tensors, the outputs are tensors. If you have data that is not natively represented as a tensor, the first step, before feeding it to a deep net, is usually to convert it into a tensor format. Most often we use tensor of real numbers, i.e. in $\mathbb{R}$.

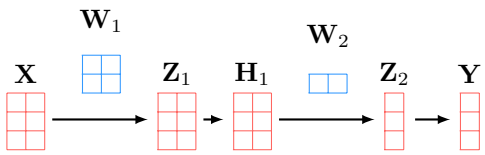
Suppose we have a dataset $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^{N}$ of images $\mathbf{x}$ and labels $\mathbf{y}$. The tensor way of thinking about such a dataset is as two tensors, $\mathbf{X} \in \mathbb{R}^{N \times H \times W \times C}$ and $\mathbf{Y} \in \mathbb{R}^{N \times K}$. The first dimension of the tensor is the number of elements in our dataset. The remaining dimensions are the dimensionality of the images (height by width by color channels) and labels (K-way classification).

The activations in the network are also tensors. For the MLP networks we have seen so far, the activation tensors have shape $N \times C_\ell$, where C_ℓ is the number of neurons on layer ℓ, sometimes also called "channels" in analogy to the color channels of the input image. In later chapters we will encounter other architectures where the activation layers have additional dimensions, for example, in convolutional networks we will see activation layers that are of shape $N \times H_\ell \times W_\ell \times C_\ell$.

One other important concept is **batch processing**. Normally, we don't process one image at a time through a neural net. Instead we run a *batch* of images all at once, and they are processed in parallel. A batch sampled from the training data can be denoted as $\{\mathbf{x}^{(i)}_{\texttt{batch}}, \mathbf{y}^{(i)}_{\texttt{batch}}\}_{i=1}^{N_{\texttt{batch}}}$, and the batch represented as a tensor has shape $\mathbf{X} \in \mathbb{R}^{N_{\texttt{batch}} \times H \times W \times C}$ and $\mathbf{Y} \in \mathbb{R}^{N_{\texttt{batch}} \times K}$.

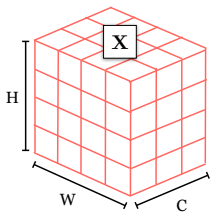
The weights and biases of the net are also usually represented as tensors. The weights and biases of a linear layer will be tensors of shape $\mathbf{W}_\ell \in \mathbb{R}^{C_{\ell+1} \times C_\ell}$ and $\mathbf{b}_\ell \in \mathbb{R}^{C_{\ell+1}}$.

As an example, we below visualize all the tensors associated a batch of 3 datapoints being processed by the MLP in Fig. 1.1:



where the capital letters are the batches of datapoints and activations corresponding to the lowercase names of datapoints and hidden units in Fig. 1.1.

The above example shows the basic concept of working with tensors and batches for 1D data, but, in vision, most of the time we will be working with higher-dimensional tensors. For image data we typically use 4D tensors: batch × height × width × channels; for videos we may use 5D tensors: batch × height × width × time × channels. 3D scans have an additional "depth" spatial dimension; videos of 3D data could therefore be represented by 6D tensors. As you can see, thinking in terms of 2D matrices is not quite sufficient. Instead, you should be imagining data processing as operating on ND tensors, sliced and diced in different ways. As a step in this direction, you may find it useful to visualize tensors in 3D:

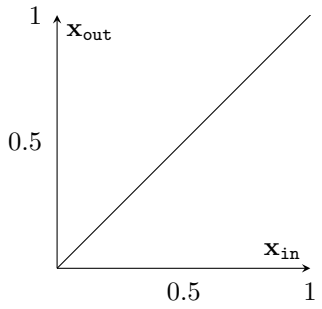


This tensor could represent an $H \times W \times C$ color image. This is closer to the actual ND tensors vision systems work with, and many concepts can be adequately captured just by thinking in 3D. We will see some examples in later chapters.

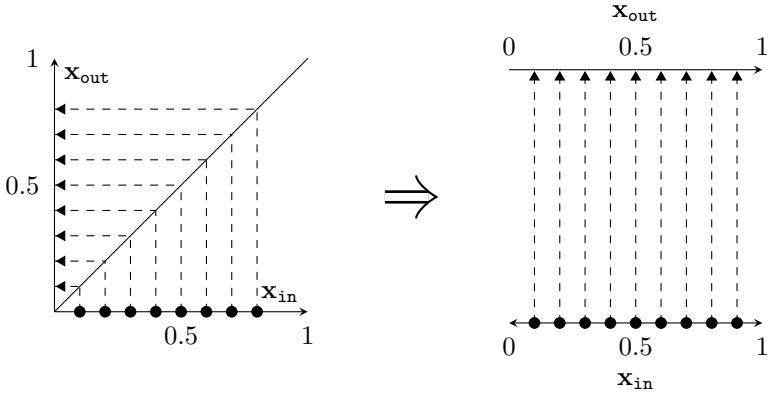
1.6 Neural nets as distribution transformers

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.*

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



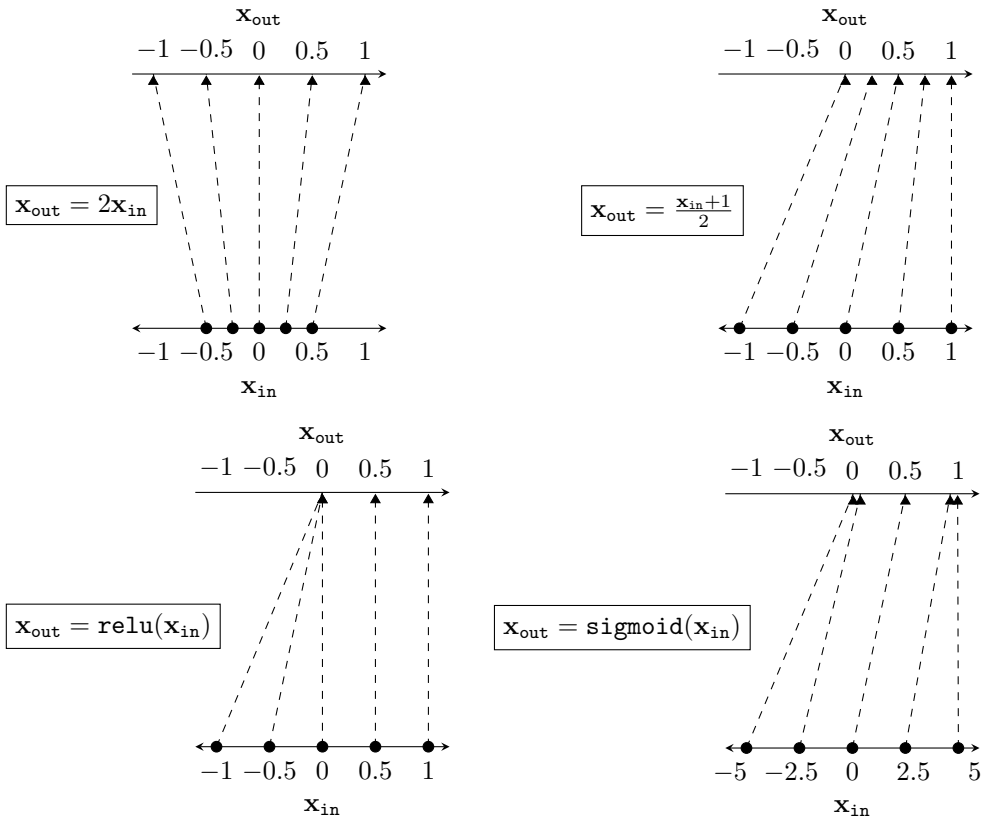
Now, we will instead consider a different way of plotting the mapping, where we simply rotate the y-axis to be horizontal rather than vertical:



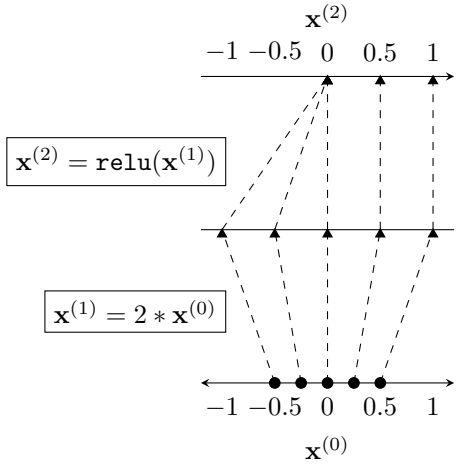
The depiction to the right makes it obvious that the plot $\mathbf{x}_{\texttt{out}} = \mathbf{x}_{\texttt{in}}$ is the identity mapping: datapoints get mapped to unchanged positions. Here are a few more mappings plotted in this way:

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 non-negative data. The `sigmoid` function pulls negative data to 0 and positive data to 1.

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, like so:

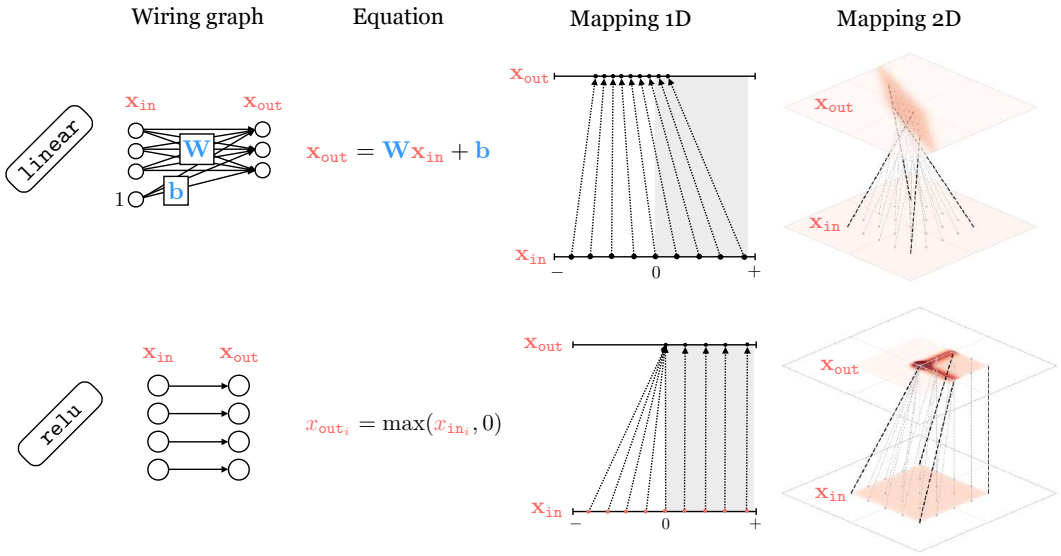


The plots above show how a uniform grid of datapoints get mapped from layer to layer in a deep net. We can also use this plotting style to show how a non-uniform distribution of incoming datapoints gets transformed. This is the setting in which deep nets actually operate, and sometimes the real action 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 the above way of plotting is that it also extends to visualizing 2D-to-2D mappings (something that conventional x-axis/y-axis plotting is not well equipped to do). Real deep nets perform ND-to-ND mappings, but already 2D-to-2D visualizations can give a lot of insight into the general case.

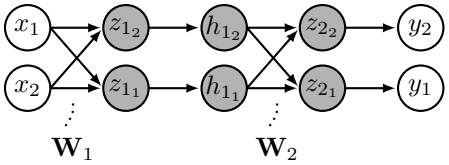
Here are the linear and relu layers plotted this way, alongside other ways of representing these mappings:



Notice that the relu layer tends to map data density to the axes of the positive quadrant. This is because relu snaps all negative coordinates to zero, so any point in the negative subspace will end up mapping to an "edge" of the positive quadrant. 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.

1.6.1 Binary classifier example

Consider an MLP that performs binary classification formulated as 2-way softmax regression. The input datapoints are each in $\mathbb{R}^2$ and the target outputs are in $\triangle^1$ (the 1-simplex), and there is one 2D hidden layer (with pre- and post-activations). This network can be drawn as follows:



Recall that the **N-simplex**, $\triangle^N$, is the set of all $N+1$ dimensional vectors whose elements sum to 1. Because the output of a softmax is normalized to sum to 1, the output of the softmax is in $\triangle^N$. $N+1$ dimensional one-hot codes live on the vertices of $\triangle^N$.

Or expressed in math as:

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 \quad \triangleleft \ \texttt{linear} \tag{1.25}$$

$$\mathbf{h}_1 = \texttt{relu}(\mathbf{z}_1) \quad \triangleleft \ \texttt{relu} \tag{1.26}$$

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2 \quad \triangleleft \ \texttt{linear} \tag{1.27}$$

$$\mathbf{y} = \texttt{softmax}(\mathbf{z}_2) \quad \triangleleft \ \texttt{softmax} \tag{1.28}$$

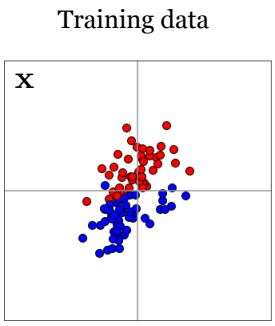
```
# W1, b1, W2, b2 : parameters of the net
# X : dataset to run through net

# first define parameterized layers
fc1 = nn.linear(W1, b1)
fc2 = nn.linear(W2, b2)

# then run data through network
for x in X:
    z1 = fc1(x)
    h1 = nn.relu(z1)
    z2 = fc2(h1)
    y = nn.softmax(z2)
```

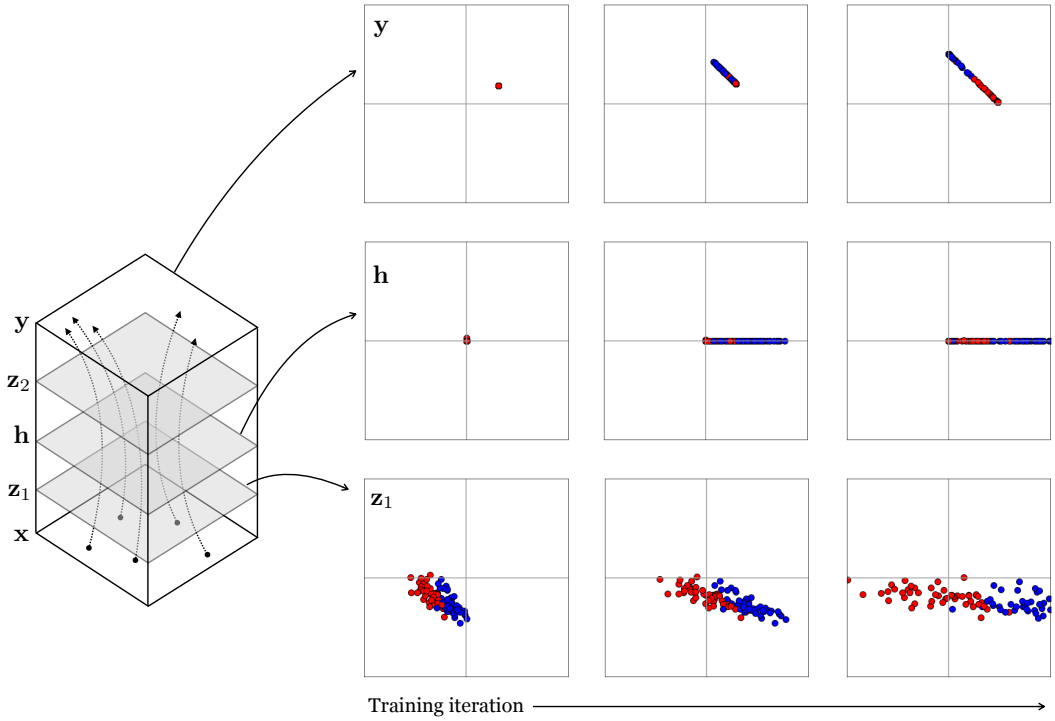
Or in Pytorch-like pseudocode as:

Now we wish to train this net to classify between two Gaussian clouds of data, the red cloud and the blue cloud here:



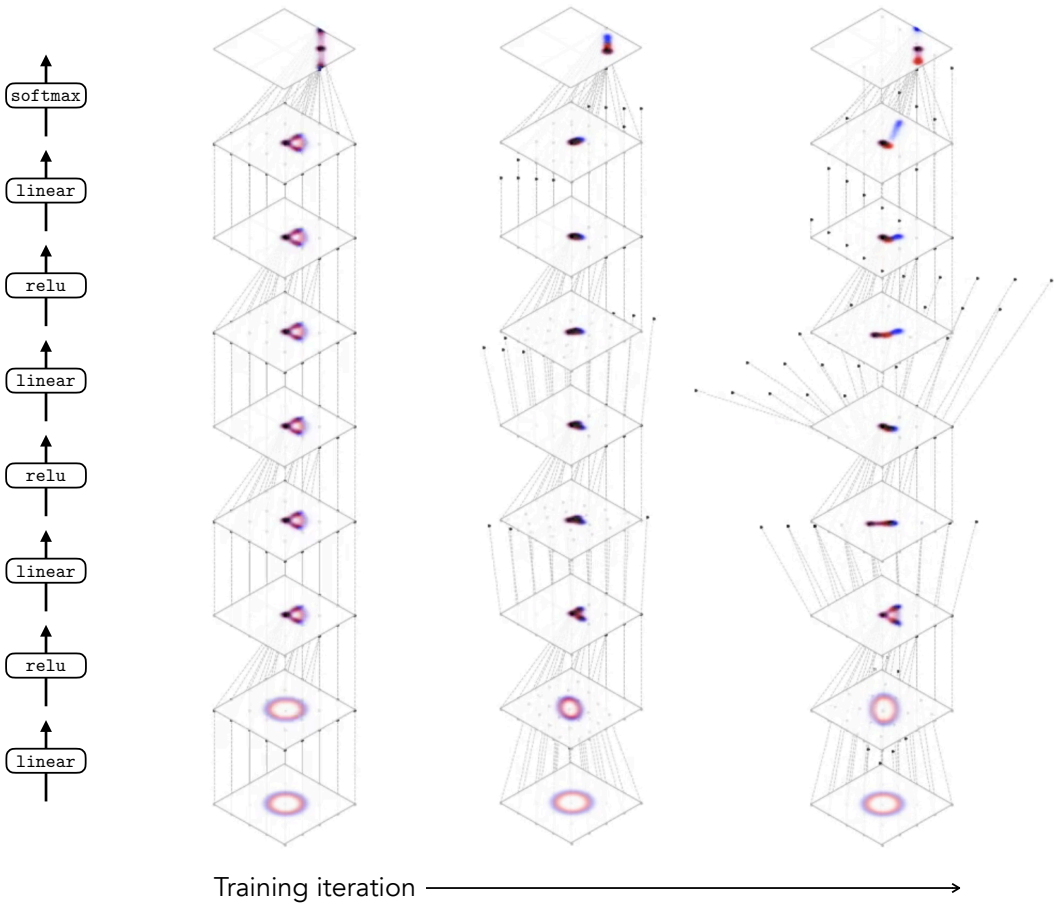
We can visualize how the net transforms the training dataset, layer by layer, at three **checkpoints** over the course of training:

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



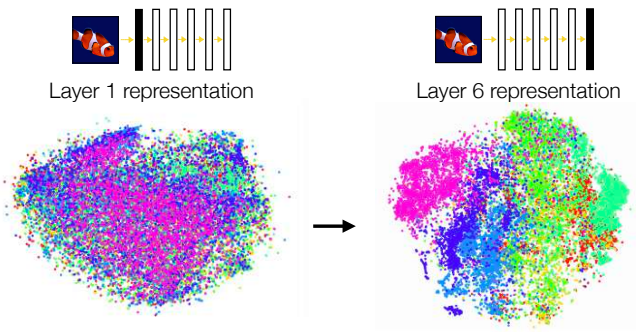
Now consider a harder classification problem. We will visualize a deeper net learning to classify between two classes of datapoints that lie on two concentric circles:

Layer by layer, over the course of training, the net learns to disentangle these two classes and pull the points toward vertices of the 1-simplex, i.e. to correctly classify the points!



1.6.2 Visualizations beyond 2D

What if our representations 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, e.g. 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. The below plot, reproduced from [Donahue et al. 2014], uses a dimensionality reduction technique called t-SNE [Maaten and Hinton 2008] to visualize how different layers of a deep net represent a dataset of images of different semantic classes, where each color represents a different semantic class:



Notice that on the first layer, semantic classes are not well separated but by layer 6 the representation has **disentangled** the semantic classes so that each class occupies a different part of representational space. This is expected because layer 6 is near the output of the network, and the output is being trained to be a one-hot representation of semantics – i.e. a completely disentangled representation.

Bibliography

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989. ISSN 1435-568X. doi: 10.1007/BF02551274. URL `https://doi.org/10.1007/BF02551274`.

J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, pages 647–655, 2014.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International Conference on Machine Learning*, pages 448–456, 2015.

L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

D. Wang, E. Shelhamer, S. Liu, B. Olshausen, and T. Darrell. Fully test-time adaptation by entropy minimization. *arXiv preprint arXiv:2006.10726*, 2020.