D Neural Representations

Neural networks are parametric representations of nonlinear functions.[1] The function represented by a neural network is differentiable, allowing gradient-based optimization algorithms such as stochastic gradient descent to optimize their parameters to better approximate desired input-output relationships.[2] Neural representations can be helpful in a variety of contexts related to decision making, such as representing probabilistic models, utility functions, and decision policies. This appendix outlines several relevant architectures.

[1] The name derives from the loose inspiration of networks of neurons in biological brains. We will not discuss these biological connections, but an overview and historical perspective is provided by B. Müller, J. Reinhardt, and M. T. Strickland, *Neural Networks*. Springer, 1995.

[2] This optimization process when applied to neural networks with many layers, as we will discuss shortly, is often called *deep learning*. Several textbooks are dedicated entirely to these techniques, including I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. The Julia package `Flux.jl` provides efficient implementations of various learning algorithms.

D.1 Neural Networks

A *neural network* is a differentiable function $\mathbf{y} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x})$ that maps inputs $\mathbf{x}$ to produce outputs $\mathbf{y}$ and is parameterized by $\boldsymbol{\theta}$. Modern neural networks may have millions of parameters and can be used to convert inputs in the form of high-dimensional images or video into high-dimensional outputs like multidimensional classifications or speech.

The parameters of the network $\boldsymbol{\theta}$ are generally tuned to minimize a scalar *loss function* $\ell(\mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y})$ that is related to how far the network output is from the desired output. Both the loss function and the neural network are differentiable, allowing us to use the gradient of the loss function with respect to the parameterization $\nabla_{\boldsymbol{\theta}}\ell$ to iteratively improve the parameterization. This process is often referred to as neural network *training* or *parameter tuning*. It is demonstrated in example D.1.

Neural networks are typically trained on a data set of input-output pairs $\mathbf{D}$. In this case, we tune the parameters to minimize the aggregate loss over the data set:

$$\underset{\boldsymbol{\theta}}{\arg\min} \sum_{(\mathbf{x},\mathbf{y})\in\mathbf{D}} \ell(\mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y}) \tag{D.1}$$

Consider a very simple neural network, $f_{\boldsymbol{\theta}}(x) = \theta_1 + \theta_2 x$. We wish our neural network to take the square footage x of a home and predict its price y_{pred}. We want to minimize the square deviation between the predicted housing price and the true housing price by the loss function $\ell(y_{\text{pred}}, y_{\text{true}}) = (y_{\text{pred}} - y_{\text{true}})^2$. Given a training pair, we can compute the gradient:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \ell(f(x), y_{\text{true}}) &= \nabla_{\boldsymbol{\theta}} (\theta_1 + \theta_2 x - y_{\text{true}})^2 \\
&= \begin{bmatrix} 2(\theta_1 + \theta_2 x - y_{\text{true}}) \\ 2(\theta_1 + \theta_2 x - y_{\text{true}})x \end{bmatrix}
\end{aligned}$$

If our initial parameterization were $\boldsymbol{\theta} = [10{,}000, 123]$ and we had the input-output pair $(x = 2{,}500, y_{\text{true}} = 360{,}000)$, then the loss gradient would be $\nabla_{\boldsymbol{\theta}} \ell = [-85{,}000, -2.125 \times 10^8]$. We would take a small step in the opposite direction to improve our function approximation.

Example D.1. The fundamentals of neural networks and parameter tuning.

Data sets for modern problems tend to be very large, making the gradient of equation (D.1) expensive to evaluate. It is common to sample random subsets of the training data in each iteration, using these *batches* to compute the loss gradient. In addition to reducing computation, computing gradients with smaller batch sizes introduces some stochasticity to the gradient, which helps training to avoid getting stuck in local minima.

D.2 Feedforward Networks

Neural networks are typically constructed to pass the input through a series of layers.[3] Networks with many layers are often called *deep*. In *feedforward networks*, each layer applies an affine transform, followed by a nonlinear *activation function* applied elementwise:[4]

$$\mathbf{x}' = \phi.(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{D.2}$$

where matrix $\mathbf{W}$ and vector $\mathbf{b}$ are parameters associated with the layer. A fully connected layer is shown in figure D.1. The dimension of the output layer is different from that of the input layer when $\mathbf{W}$ is nonsquare. Figure D.2 shows a more compact depiction of the same network.

[3] A sufficiently large, single-layer neural network can, in theory, approximate any function. See A. Pinkus, "Approximation Theory of the MLP Model in Neural Networks," *Acta Numerica*, vol. 8, pp. 143–195, 1999.

[4] The nonlinearity introduced by the activation function provides something analogous to the activation behavior of biological neurons, in which input buildup eventually causes a neuron to fire. A. L. Hodgkin and A. F. Huxley, "A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve," *Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952.

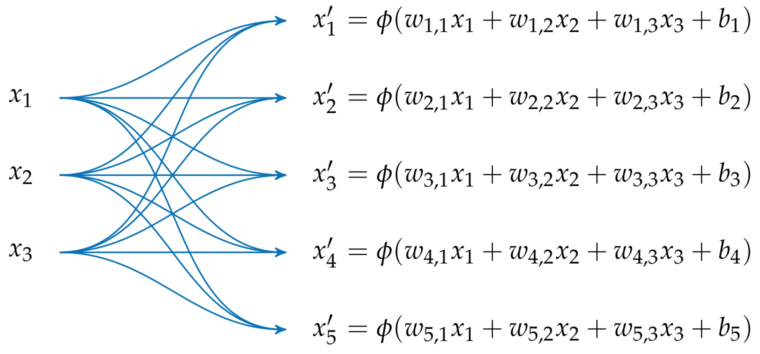

x_1
x_2
x_3
$x_1' = \phi(w_{1,1}x_1 + w_{1,2}x_2 + w_{1,3}x_3 + b_1)$
$x_2' = \phi(w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3}x_3 + b_2)$
$x_3' = \phi(w_{3,1}x_1 + w_{3,2}x_2 + w_{3,3}x_3 + b_3)$
$x_4' = \phi(w_{4,1}x_1 + w_{4,2}x_2 + w_{4,3}x_3 + b_4)$
$x_5' = \phi(w_{5,1}x_1 + w_{5,2}x_2 + w_{5,3}x_3 + b_5)$

Figure D.1. A fully connected layer with a three-component input and a five-component output.

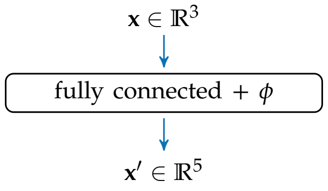


Figure D.2. A more compact depiction of figure D.1. Neural network layers are often represented as blocks or slices for simplicity.

If there are no activation functions between them, multiple successive affine transformations can be collapsed into a single, equivalent affine transform:

$$\mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = \mathbf{W}_2\mathbf{W}_1\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2) \tag{D.3}$$

These nonlinearities are necessary to allow neural networks to adapt to fit arbitrary target functions. To illustrate, figure D.3 shows the output of a neural network trained to approximate a nonlinear function.

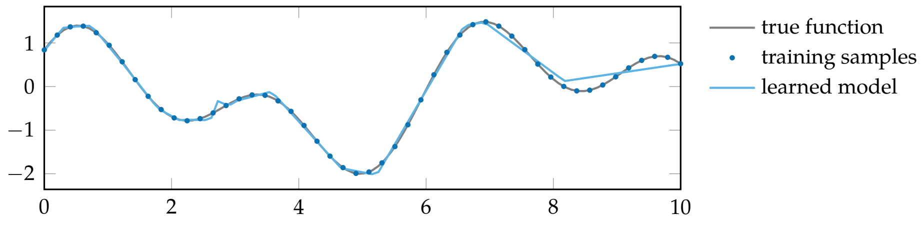
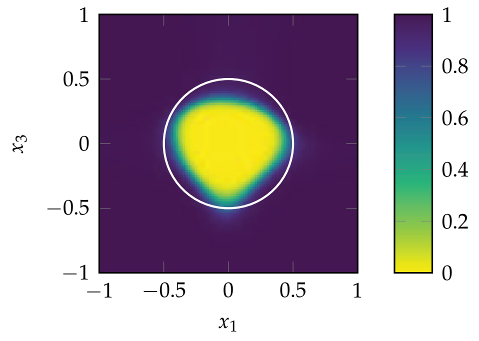
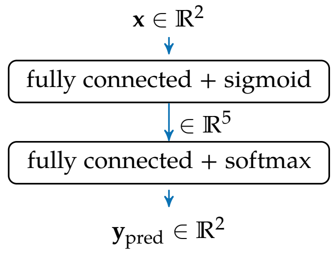


Figure D.3. A deep neural network fit to samples from a nonlinear function so as to minimize the squared error. This neural network has four affine layers, with 10 neurons in each intermediate representation.

There are many types of activation functions that are commonly used. Similar to their biological inspiration, they tend to be close to zero when their input is low and large when their input is high. Some common activation functions are shown in figure D.5.

Sometimes special layers are incorporated to achieve certain effects. For example, in figure D.4, we used a *softmax* layer at the end to force the output to represent a two-element categorical distribution. The softmax function applies





$\mathbf{x} \in \mathbb{R}^2$
fully connected + sigmoid
$\in \mathbb{R}^5$
fully connected + softmax
$\mathbf{y}_{\text{pred}} \in \mathbb{R}^2$

Figure D.4. A simple, two-layer, fully connected network trained to classify whether a given coordinate lies within a circle (shown in white). The nonlinearities allow neural networks to form complicated, nonlinear decision boundaries.

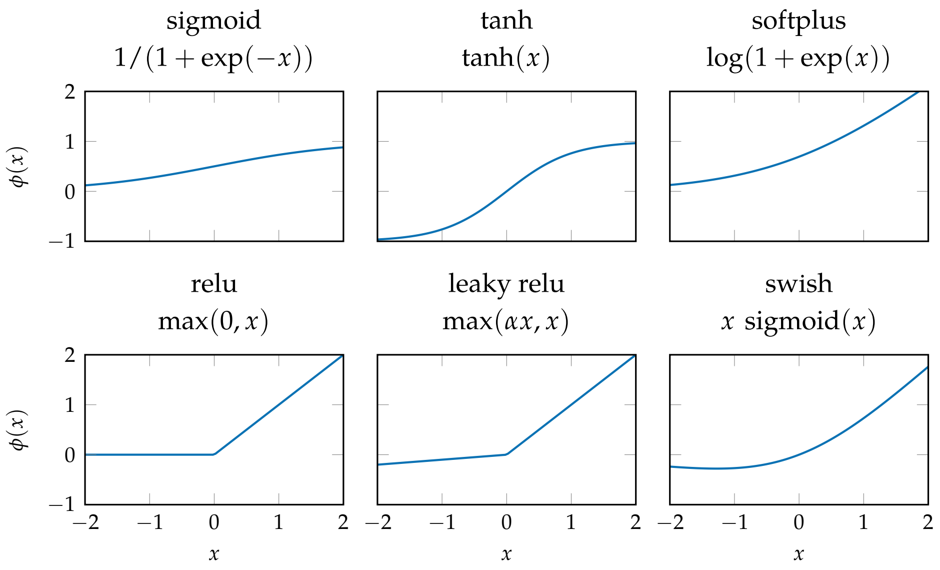


Figure D.5. Several common activation functions.

the exponential function to each element, which ensures that they are positive and then renormalizes the resulting values:

$$\text{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{D.4}$$

Gradients for neural networks are typically computed using *reverse accumulation*.[5] The method begins with a forward step, in which the neural network is evaluated using all input parameters. In the backward step, the gradient of each term of interest is computed working from the output back to the input. Reverse accumulation uses the chain rule for derivatives:

$$\frac{\partial \mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{x})))}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}(\mathbf{g}(\mathbf{h}))}{\partial \mathbf{h}} \frac{\partial \mathbf{h}(\mathbf{x})}{\partial \mathbf{x}} = \left(\frac{\partial \mathbf{f}(\mathbf{g})}{\partial \mathbf{g}} \frac{\partial \mathbf{g}(\mathbf{h})}{\partial \mathbf{h}} \right) \frac{\partial \mathbf{h}(\mathbf{x})}{\partial \mathbf{x}} \tag{D.5}$$

[5] This process is commonly called *backpropagation*, which specifically refers to reverse accumulation applied to a scalar loss function. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning Representations by Back-Propagating Errors,” *Nature*, vol. 323, pp. 533–536, 1986.

Example D.2 demonstrates this process. Many deep learning packages compute gradients using such automatic differentiation techniques.[6] Users rarely have to provide their own gradients.

[6] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. SIAM, 2008.

D.3 Parameter Regularization

Neural networks are typically *underdetermined*, meaning that there are multiple parameter instantiations that can result in the same optimal training loss.[7] It is common to use *parameter regularization*, also called *weight regularization*, to introduce an additional term to the loss function that penalizes large parameter values. Regularization also helps prevent *overfitting*, which occurs when a network over-specializes to the training data but fails to generalize to unseen data.

[7] For example, suppose that we have a neural network with a final softmax layer. The inputs to that layer can be scaled while producing the same output, and therefore the same loss.

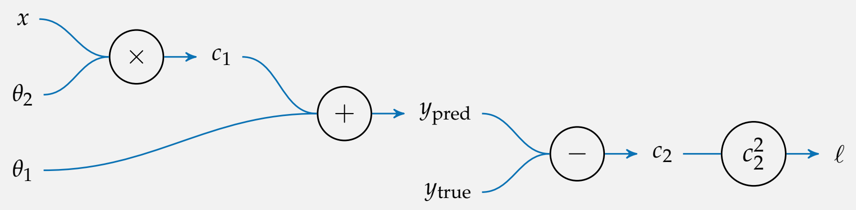
Regularization often takes the form of an L_2-norm of the parameterization vector:

$$\underset{\boldsymbol{\theta}}{\arg\min} \sum_{(x,y)\in\mathbf{D}} \ell(f_{\boldsymbol{\theta}}(x), y) + \beta \|\boldsymbol{\theta}\|^2 \tag{D.6}$$

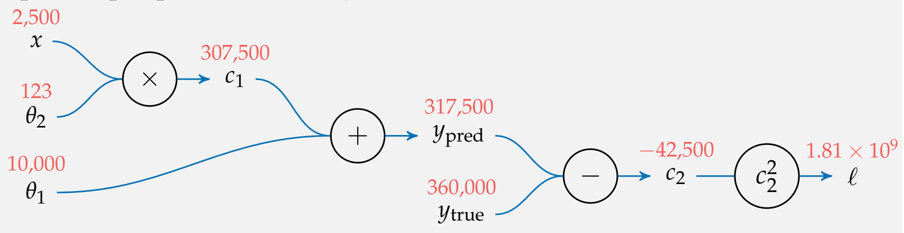
where the positive scalar β controls the strength of the parameter regularization. The scalar is often quite small, with values as low as 10^{-6}, to minimize the degree to which matching the training set is sacrificed by introducing regularization.

Example D.2. How reverse accumulation is used to compute parameter gradients given training data.

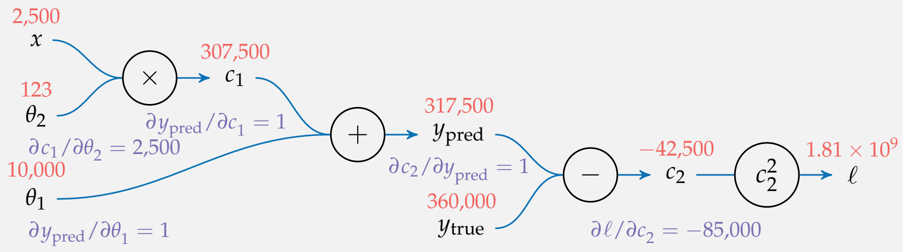
Recall the neural network and loss function from example D.1. Here we have drawn the computational graph for the loss calculation:



Reverse accumulation begins with a forward pass, in which the computational graph is evaluated. We will again use $\boldsymbol{\theta} = [10{,}000, 123]$ and the input-output pair $(x = 2{,}500, y_{\text{true}} = 360{,}000)$ as follows:



The gradient is then computed by working back up the tree:



2,500
x
123
θ_2
$\partial c_1/\partial\theta_2 = 2{,}500$
10,000
θ_1
$\partial y_{\text{pred}}/\partial\theta_1 = 1$
307,500
c_1
$\partial y_{\text{pred}}/\partial c_1 = 1$
317,500
y_{pred}
$\partial c_2/\partial y_{\text{pred}} = 1$
360,000
y_{true}
−42,500
c_2
$\partial \ell/\partial c_2 = -85{,}000$
1.81×10^9
ℓ

Finally, we compute:

$$\frac{\partial \ell}{\partial \theta_1} = \frac{\partial \ell}{\partial c_2}\frac{\partial c_2}{\partial y_{\text{pred}}}\frac{\partial y_{\text{pred}}}{\partial \theta_1} = -85{,}000 \cdot 1 \cdot 1 = -85{,}000$$

$$\frac{\partial \ell}{\partial \theta_2} = \frac{\partial \ell}{\partial c_2}\frac{\partial c_2}{\partial y_{\text{pred}}}\frac{\partial y_{\text{pred}}}{\partial c_1}\frac{\partial c_1}{\partial \theta_2} = -85{,}000 \cdot 1 \cdot 1 \cdot 2500 = -2.125 \times 10^8$$

D.4 Convolutional Neural Networks

Neural networks may have images or other multidimensional structures such as lidar scans as inputs. Even a relatively small 256×256 RGB image (similar to figure D.6) has $256 \times 256 \times 3 = 196{,}608$ entries. Any fully connected layer taking an $m \times m \times 3$ image as input and producing a vector of n outputs would have a weight matrix with $3m^2n$ values. The large number of parameters to learn is not only computationally expensive, it is also wasteful. Information in images is typically translation-invariant; an object in an image that is shifted right by 1 pixel should produce a similar, if not identical, output.

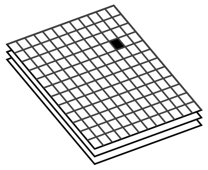


Figure D.6. Multidimensional inputs like images generalize vectors to tensors. Here, we show a three-layer RGB image. Such inputs can have very many entries.

Convolutional layers[8] both significantly reduce the amount of computation and support translation invariance by sliding a smaller, fully connected window to produce their output. Significantly fewer parameters need to be learned. These parameters tend to be receptive to local textures in much the same way that the neurons in the visual cortex respond to stimuli in their receptive fields.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

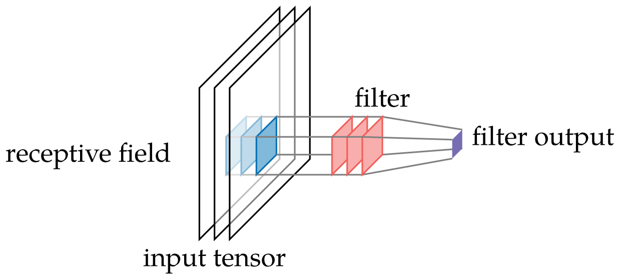


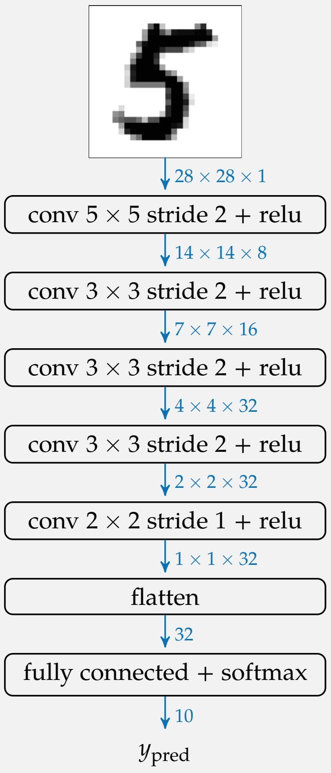
Figure D.7. A convolutional layer repeatedly applies filters across an input tensor, such as an image, to produce an output tensor. This illustration shows how each application of the filter acts like a small, fully connected layer applied to a small receptive field to produce a single entry in the output tensor. Each filter is shifted across the input according to a prescribed stride. The resulting output has as many layers as there are filters.

The convolutional layer consists of a set of *features*, or *kernels*, each of which is equivalent to a fully connected layer into which one can input a smaller region of the input tensor. A single kernel is being applied once in figure D.7. These features have full depth, meaning that if an input tensor is $n \times m \times d$, the features will also have a third dimension of d. The features are applied many times by sliding them over the input in both the first and second dimensions. If the *stride* is 1×1, then all k filters are applied to every possible position and the output dimension will be $n \times m \times k$. If the stride is 2×2, then the filters are shifted by 2 in the first and second dimensions with every application, resulting in an output of size $n/2 \times m/2 \times k$. It is common for convolutional neural networks to increase in the third dimension and reduce in the first two dimensions with each layer.

Convolutional layers are translation-invariant because each filter behaves the same regardless of where in the input is applied. This property is especially useful in spatial processing because shifts in an input image can yield similar outputs, making it easier for neural networks to extract common features. Individual features tend to learn how to recognize local attributes such as colors and textures.

Example D.3. A convolutional neural network for the MNIST data set. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

The MNIST data set contains handwritten digits in the form of 28 × 28 monochromatic images. It is a often used to test image classification networks. To the right, we have a sample convolutional neural network that takes an MNIST image as input and produces a categorical probability distribution over the 10 possible digits. Convolutional layers are used to efficiently extract features. The model shrinks in the first two dimensions and expands in the third dimension (the number of features) as the network depth increases. Eventually reaching a first and second dimension of 1 ensures that information from across the entire image can affect every feature. The flatten operation takes the 1 × 1 × 32 input and flattens it into a 32-component output. Such operations are common when transitioning between convolutional and fully connected layers. This model has 19,722 parameters. The parameters can be tuned to maximize the likelihood of the training data.



28 × 28 × 1
conv 5 × 5 stride 2 + relu
14 × 14 × 8
conv 3 × 3 stride 2 + relu
7 × 7 × 16
conv 3 × 3 stride 2 + relu
4 × 4 × 32
conv 3 × 3 stride 2 + relu
2 × 2 × 32
conv 2 × 2 stride 1 + relu
1 × 1 × 32
flatten
32
fully connected + softmax
10
y_{pred}

D.5 *Recurrent Networks*

The neural network architectures discussed so far are ill suited for temporal or sequential inputs. Operations on sequences occur when processing images from videos, when translating a sequence of words, or when tracking time-series data. In such cases, the outputs depend on more than just the most recent input. In addition, the neural network architectures discussed so far do not naturally produce variable-length outputs. For example, a neural network that writes an essay would be difficult to train using a conventional, fully connected neural network.

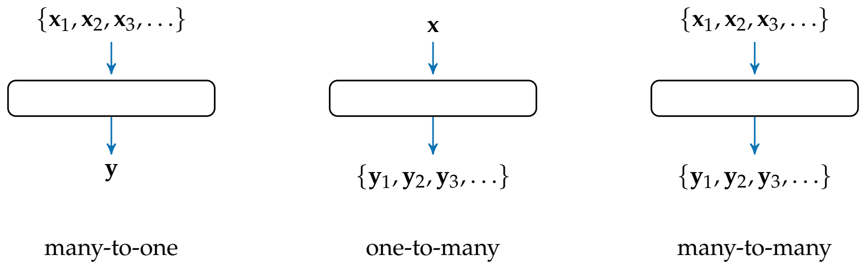

$\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots\}$
$\mathbf{y}$
many-to-one
$\mathbf{x}$
$\{\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \ldots\}$
one-to-many
$\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots\}$
$\{\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \ldots\}$
many-to-many

Figure D.8. Traditional neural networks do not naturally accept variable-length inputs or produce variable-length outputs.

When a neural network has sequential input, sequential output, or both (figure D.8), we can use a *recurrent neural network* to act over multiple iterations. These neural networks maintain a recurrent state $\mathbf{r}$, sometimes called its *memory*, to retain information over time. For example, in translation, a word used early in a sentence may be relevant to the proper translation of words later in the sentence. Figure D.9 shows the structure of a basic recurrent neural network and how the same neural network can be understood to be a larger network unrolled in time.

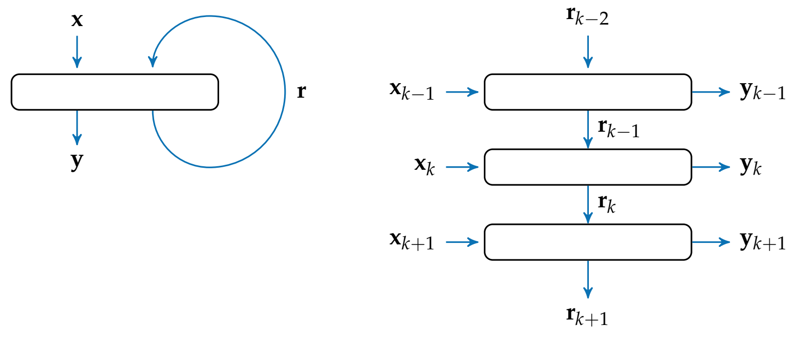


Figure D.9. A recurrent neural network (left) and the same recurrent neural network unrolled in time (right). These networks maintain a recurrent state $\mathbf{r}$ that allows the network to develop a sort of memory, transferring information across iterations.

This unrolled structure can be used to produce a rich diversity of sequential neural networks, as shown in figure D.10. Many-to-many structures come in multiple forms. In one form, the output sequence begins with the input sequence. In another form, the output sequence does not begin with the input sequence. When using variable-length outputs, the neural network output itself often indicates when a sequence begins or ends. The recurrent state is often initialized to zero, as are extra inputs after the input sequence has been passed in, but this need not be the case.

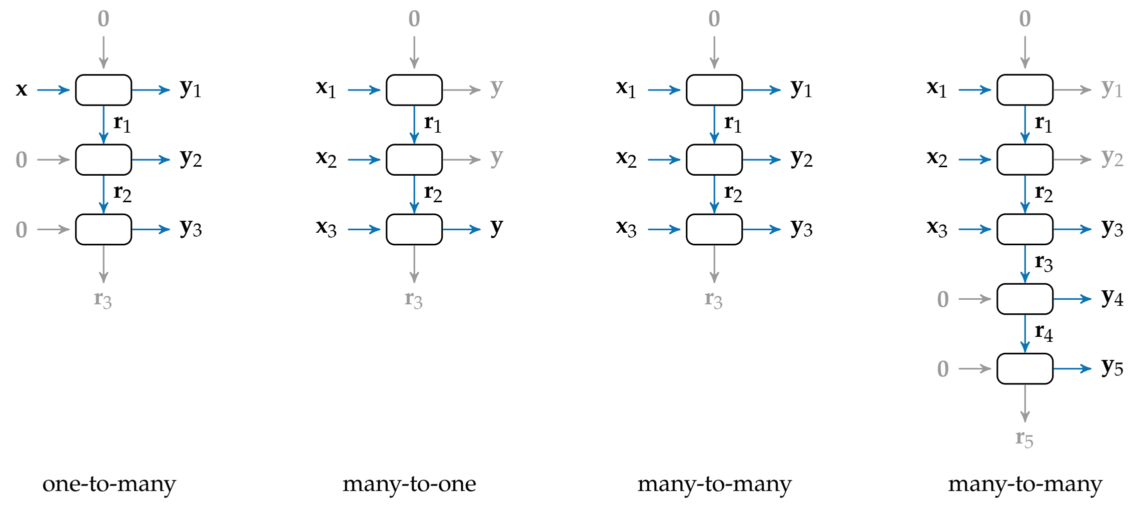


Figure D.10. A recurrent neural network can be unrolled in time to produce different relationships. Unused or default inputs and outputs are grayed out.

Recurrent neural networks with many layers, unrolled over multiple time steps, effectively produce a very deep neural network. During training, gradients are computed with respect to the loss function. The contribution of layers farther from the loss function tends to be smaller than that of layers close to the loss function. This leads to the *vanishing gradient* problem, in which deep neural networks have vanishingly small gradients in their upper layers. These small gradients slow training.

Very deep neural networks can also suffer from *exploding gradients*, in which successive gradient contributions through the layers combine to produce very large values. Such large values make learning unstable. Example D.4 shows both exploding and vanishing gradients.

To illustrate vanishing and exploding gradients, consider a deep neural network made of one-dimensional, fully connected layers with relu activations. For example, if the network has three layers, its output is

$$f_{\boldsymbol{\theta}}(x) = \text{relu}(w_3 \ \text{relu}(w_2 \ \text{relu}(w_1 x_1 + b_1) + b_2) + b_3)$$

The gradient with respect to a loss function depends on the gradient of $f_{\boldsymbol{\theta}}$.

We can get vanishing gradients in the parameters of the first layer, w_1 and b_1, if the gradient contributions in successive layers are less than 1. For example, if any of the layers has a negative input to its relu function, the gradient of its inputs will be zero, so the gradient vanishes entirely. In a less extreme case, suppose that the weights are all $\mathbf{w} = 0.5\,\mathbf{1}$, the offsets are all $\mathbf{b} = \mathbf{0}$, and the input x is positive. In this case, the gradient with respect to w_1 is

$$\frac{\partial f}{\partial w_1} = x_1 \cdot w_2 \cdot w_3 \cdot w_4 \cdot w_5 \ldots$$

The deeper the network, the smaller the gradient will be.

We can get exploding gradients in the parameters of the first layer if the gradient contributions in successive layers are greater than 1. If we merely increase our weights to $\mathbf{w} = 2\,\mathbf{1}$, the very same gradient is suddenly doubling every layer.

Example D.4. A demonstration of how vanishing and exploding gradients arise in deep neural networks. This example uses a very simple neural network. In larger, fully connected layers, the same principles apply.

While exploding gradients can often be handled with gradient clipping, regularization, and initializing parameters to small values, these solutions merely shift the problem toward that of vanishing gradients. Recurrent neural networks often use layers specifically constructed to mitigate the vanishing gradients problem. They function by selectively choosing whether to retain memory, and these gates help regulate the memory and the gradient. Two common recurrent layers are *long short-term memory* (*LSTM*)[9] and *gated recurrent units* (*GRU*).[10]

[9] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[10] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation," in *Conference on Empirical Methods in Natural Language Processing* (*EMNLP*), 2014.

D.6 Autoencoder Networks

Neural networks are often used to process high-dimensional inputs such as images or point clouds. These high-dimensional inputs are often highly structured, with the actual information content being much lower-dimensional than the high-dimensional space in which it is presented. Pixels in images tend to be highly correlated with their neighbors, and point clouds often have many regions of continuity. Sometimes we wish to build an understanding of the information content of our data sets by converting them to a much smaller set of features, or an *embedding*. This compression, or *representation learning*, has many advantages.[11] Lower-dimensional representations can help facilitate the application of traditional machine learning techniques like Bayesian networks to what would have otherwise been intractable. The features can be inspected to develop an understanding of the information content of the data set, and these features can be used as inputs to other models.

[11] Such dimensionality reduction can also be done using traditional machine learning techniques, such as principal component analysis. Neural models allow more flexibility and can handle nonlinear representations.

An *autoencoder* is a neural network trained to discover a low-dimensional feature representation of a higher-level input. An autoencoder network takes in a high-dimensional input $\mathbf{x}$ and produces an output $\mathbf{x}'$ with the same dimensionality. We design the network architecture to pass through a lower-dimensional intermediate representation called a *bottleneck*. The activations $\mathbf{z}$ at this bottleneck are our low-dimensional features, which exist in a *latent space* that is not explicitly observed. Such an architecture is shown in figure D.11.

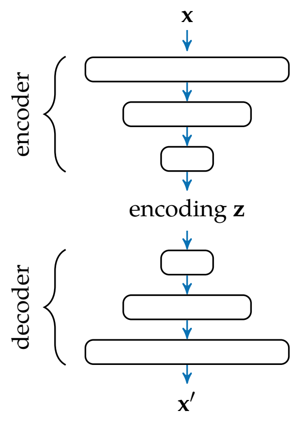


Figure D.11. An autoencoder passes a high-dimensional input through a low-dimensional bottleneck and then reconstructs the original input. Minimizing reconstruction loss can result in an efficient low-dimensional encoding.

We train the autoencoder to reproduce its input. For example, to encourage the output $\mathbf{x}'$ to match $\mathbf{x}$ as closely as possible, we may simply minimize the L_2-norm,

$$\underset{\boldsymbol{\theta}}{\text{minimize}} \quad \mathop{\mathbb{E}}_{\mathbf{x}\in\mathbf{D}}[\|f_{\boldsymbol{\theta}}(\mathbf{x}) - \mathbf{x}\|_2] \tag{D.7}$$

Noise is often added to the input to produce a more robust feature embedding:

$$\underset{\boldsymbol{\theta}}{\text{minimize}} \quad \underset{\mathbf{x}\in\mathbf{D}}{\mathbb{E}}[\|f_{\boldsymbol{\theta}}(\mathbf{x}+\boldsymbol{\epsilon})-\mathbf{x}\|_2] \tag{D.8}$$

Training to minimize the reconstruction loss forces the autoencoder to find the most efficient low-dimensional encoding that is sufficient to accurately reconstruct the original input. Furthermore, training is *unsupervised*, in that we do not need to guide the training to a particular feature set.

After training, the upper portion of the autoencoder above the bottleneck can be used as an *encoder* that transforms an input into the feature representation. The lower portion of the autoencoder can be used as a *decoder* that transforms the feature representation into the input representation. Decoding is useful when training neural networks to generate images or other high-dimensional outputs. Example D.5 shows an embedding learned for handwritten digits.

A *variational autoencoder*, shown in figure D.12, extends the autoencoder framework to learn a probabilistic encoder.[12] Rather than outputting a deterministic sample, the encoder produces a distribution over the encoding, which allows the model to assign confidence to its encoding. Multivariate Gaussian distributions with diagonal covariance matrices are often used for their mathematical convenience. In such a case, the encoder outputs both an encoding mean and diagonal covariance matrix.

Variational autoencoders are trained to both minimize the expected reconstruction loss while keeping the encoding components close to unit Gaussian. The former is achieved by taking a single sample from the encoding distribution with each passthrough, $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^\top \mathbf{I}\boldsymbol{\sigma})$. For backpropagation to work, we typically include random noise $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ as an additional input to the neural network and obtain our sample according to $\mathbf{z} = \boldsymbol{\mu} + \mathbf{w} \odot \boldsymbol{\sigma}$.

The components are kept close to unit Gaussian by also minimizing the KL divergence (appendix A.10).[13] This objective encourages smooth latent space representations. The network is penalized for spreading out the latent representations (large values for $\|\boldsymbol{\mu}\|$) and for focusing each representation into a very small encoding space (small values for $\|\boldsymbol{\sigma}\|$), ensuring better coverage of the latent space. As a result, smooth variations into the decoder can result in smoothly varying outputs. This property allows decoders to be used as *generative models*, where samples from a unit multivariate Gaussian can be input to the decoder to

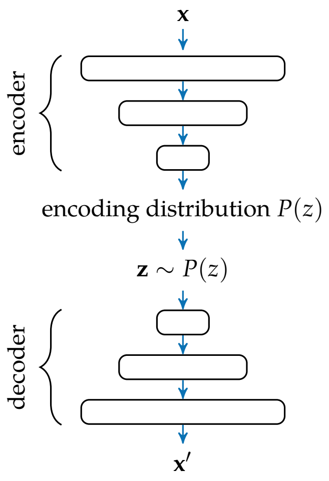


Figure D.12. A variational autoencoder passes a high-dimensional input through a low-dimensional bottleneck that produces a probability distribution over the encoding. The decoder reconstructs samples from this encoding to reconstruct the original input. Variational autoencoders can therefore assign confidence to each encoded feature. The decoder can thereafter be used as a generative model.

[12] D. Kingma and M. Welling, "Auto-Encoding Variational Bayes," in *International Conference on Learning Representations (ICLR)*, 2013.

[13] The KL divergence for two unit Gaussians is

$$\log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}$$

produce realistic samples in the original space. The combined loss function is

$$\begin{aligned} \underset{\boldsymbol{\theta}}{\text{minimize}} \quad & \mathop{\mathbb{E}}_{\mathbf{x} \in \mathbf{D}} \left[\|\mathbf{x}' - \mathbf{x}\|_2 + c \sum_{i=1}^{|\boldsymbol{\mu}|} D_{\mathrm{KL}}\Big(\mathcal{N}\Big(\mu_i, \sigma_i^2, \Big) \Big|\Big| \mathcal{N}(0,1)\Big) \right] \\ \text{subject to} \quad & \boldsymbol{\mu}, \boldsymbol{\sigma} = \text{encoder}(\mathbf{x} + \boldsymbol{\epsilon}) \\ & \mathbf{x}' = \text{decoder}(\boldsymbol{\mu} + \mathbf{w} \odot \boldsymbol{\sigma}) \end{aligned} \tag{D.9}$$

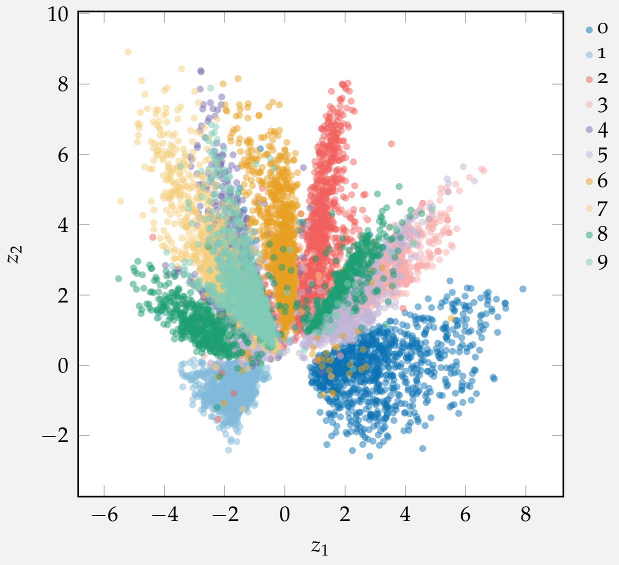
where the trade-off between the two losses is tuned by the scalar $c > 0$. Example D.6 demonstrates this process on a latent space learned from handwritten digits.

Variational autoencoders are derived by representing the encoder as a conditional distribution $q(\mathbf{z} \mid \mathbf{x})$, where $\mathbf{x}$ belongs to the observed input space and $\mathbf{z}$ is in the unobserved embedding space. The decoder performs inference in the other direction, representing $p(\mathbf{x} \mid \mathbf{z})$, in which case it also outputs a probability distribution. We seek to minimize the KL divergence between $q(\mathbf{z} \mid \mathbf{x})$ and $p(\mathbf{z} \mid \mathbf{x})$, which is the same as minimizing $\mathbb{E}[\log p(\mathbf{x} \mid \mathbf{z})] - D_{\mathrm{KL}}(q(\mathbf{z} \mid \mathbf{x}) \mid\mid p(\mathbf{z}))$, where $p(\mathbf{z})$ is our prior, the unit multivariate Gaussian to which we bias our encoding distribution. We thus obtain our reconstruction loss and our KL divergence.

D.7 Adversarial Networks

We often want to train neural networks to produce high-dimensional outputs, such as images or sequences of helicopter control inputs. When the output space is large, the training data may cover only a very small region of the state space. Hence, training purely on the available data can cause unrealistic results or overfitting. We generally want the neural network to produce plausible outputs. For example, when producing images, we want the images to look realistic. When mimicking human driving, such as in imitation learning (chapter 18), we want the vehicle to typically stay in its lane and to react appropriately to other vehicles.

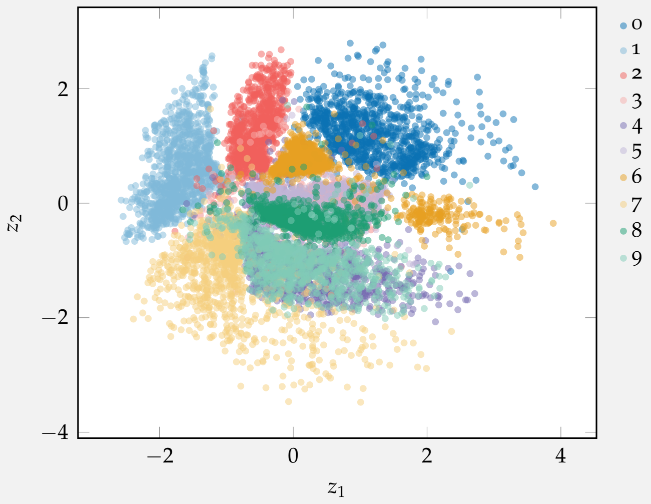
We can use an autoencoder to train an embedding for the MNIST data set. In this example, we use an encoder similar to the convolutional network in example D.3, except with a two-dimensional output and no softmax layer. We construct a decoder that mirrors the encoder and train the full network to minimize the reconstruction loss. Here are the encodings for 10,000 images from the MNIST data set after training. Each encoding is colored according to the corresponding digit:



We find that the digits tend to be clustered into regions that are roughly radially distributed from the origin. Note how the encodings for 1 and 7 are similar, as the two digits look alike. Recall that training is unsupervised, and the network is not given any information about the digit values. Nevertheless, these clusterings are produced.

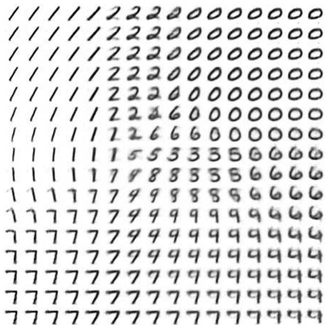
Example D.5. A visualization of a two-dimensional embedding learned for the MNIST digits.

In example D.5, we trained an autoencoder on the MNIST data set. We can adapt the same network to produce two-dimensional mean and variance vectors at the bottleneck instead of a two-dimensional embedding, and then train it to minimize both the reconstruction loss and the KL divergence. Here, we show the mean encodings for the same 10,000 images for the MNIST data set. Each encoding is again colored according to the corresponding digit:



The variational autoencoder also produces clusters in the embedding space for each digit, but this time they are roughly distributed according to a zero-mean, unit variance Gaussian distribution. We again see how some encodings are similar, such as the significant overlap for 4 and 9.

Example D.6. A visualization of a two-dimensional embedding learned using a variational autoencoder for the MNIST digits. Here, we show decoded outputs from inputs panned over the encoding space:



One common approach to penalize off-nominal outputs or behavior is to use *adversarial learning* by including a discriminator, as shown in figure D.13.[14] A *discriminator* is a neural network that acts as a binary classifier that takes in neural network outputs and learns to distinguish between real outputs from the training set and the outputs from the primary neural network. The primary neural network, also called a *generator*, is then trained to deceive the discriminator, thereby naturally producing outputs that are more difficult to distinguish from the data set. The primary advantage of this technique is that we do not need to design special features to identify or quantify how the output fails to match the training data, but we can allow the discriminator to naturally learn such differences.

Learning is adversarial in the sense that we have two neural networks: the primary neural network that we would like to produce realistic outputs and the discriminator network that distinguishes between primary network outputs and real examples. They are each training to outperform the other. Training is an iterative process in which each network is improved in turn. It can sometimes be challenging to balance their relative performance; if one network becomes too good, the other can become stuck.

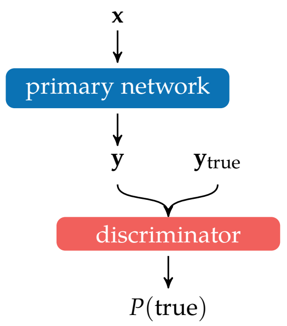


Figure D.13. A generative adversarial network causes a primary network's output to be more realistic by using a discriminator to force the primary network to produce more realistic output.

[14] These techniques were introduced by I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems (NIPS)*, 2014.