DEEP LEARNING

lecture notes

Samuel S. Watson

Preface

These notes exist as a record of the trajectory of the Spring 2019 DATA 2040 course taught as a part of the Brown University Data Science Initiative Master's Program. In this course we will rely on the following three textbooks:

1. *Neural Networks and Deep Learning: A Textbook*, by Charu C. Aggarwal
2. *Deep Learning with Python*, by Francois Chollet
3. *Hands-on Machine Learning with Scikit-Learn and TensorFlow*, by Aurélien Géron

These lecture notes will reference these books extensively. Note that the first and third of these books are available for free through your library account (go to **`library.brown.edu`** and search for the book title). The second book costs $40 dollars in ebook format or $50 for the paper book and the ebook.

1 Introduction

1.1 Course details

Announcements

1. The course website is `data2040.github.io`.
2. We're using CCV (the university's Center for Computation and Visualization) for compute resources. You should have received an email with credentials from the head TAs yesterday. If you did not, please send an email to `dsi2040headtas@lists.brown.edu`.
3. *Assignment 0* is a scavenger hunt exercise to help you get your computing environment set up. It's due next Tuesday. There is also a collaboration policy form that you have to fill out prior to submitting any assignments.

1.1.1 Course plan

We plan to follow Aggarwal's book closely enough that you can read it extensively without doing a lot of work to map the topics in the book to topics in class. Although this book is fairly mathematically challenging, it is excellent preparation for reading papers that will be advancing the field in the coming years. We will try to strike a balance between elucidating the key ideas as much as possible and equipping you to learn from the literature, because staying abreast of recent developments is essential for keeping your skills current.

1.1.2 Frameworks

Aggarwal's book lacks computational examples, with the intent that instructors may supplement the conceptual and mathematical content in the book with their framework of preference. In this class we will use *TensorFlow*, which is open source and backed by Google. This is the most popular deep learning framework*, and although it has a reputation for being difficult to use, we will be able to work primarily with the convenient *Keras* interface (which is now built into TensorFlow, although it also exists as a standalone package which can work with some other frameworks as well). We will be using TensorFlow 2.0, which is available and soon to be officially released. You can access the documentation at `tensorflow.org` under the `API` pulldown.

* See this article for a popularity comparison of deep learning frameworks

You will be able to use TensorFlow on your computer, using your CCV account, or using *Google Colab*. Colab is a free Google-hosted Jupyter notebook with scientific computing tools built in. Colab is easy to use (and integrates with Google Drive and supports collaboration), but the GPU* time is limited, so official course support will be for CCV.

* *Graphics processing units* are used to accelerate deep learning training, and they are essential for practical deep learning

1.2 The multi-layer perceptron

Reading: *Aggarwal*, Sections 1.1, 1.2

Many machine learning models learn the input-output relationship for a given dataset by introducing a parametric class of functions and identifying the values of the parameters which minimize a specified loss function. For example, logistic regression models the input-output relationship with the function $\mathbf{x} \mapsto \sigma(L(\mathbf{x}))$, where σ is the logistic function and L is a linear* function. Support vector machines use $\mathbf{x} \mapsto \sigma(L(\mathbf{x}))$, where σ is the signum* function.

* more precisely, affine

* which returns the sign of its input

The core idea of deep learning is to enhance these models by *composing* them. Arranging simpler models into layers and training the full composition allows the network to achieve synergy between the layers. Earlier layers are free to learn features which are useful for later layers.

* This is the one we learned in DATA 1010

The prototypical example of a neural network is the *multilayer perceptron**. The idea of the multilayer perceptron is to devise a function which approximately maps each input vector to its corresponding output by composing functions of the form $K. \circ A$ where $K.$ is pointwise application of a specified *activation* function and $A(\mathbf{x}) = W\mathbf{x} + \mathbf{b}$ is an affine map whose entries must be learned. The most popular activation function in modern applications is the ReLU function, which maps x to x if x is positive and to 0 otherwise.

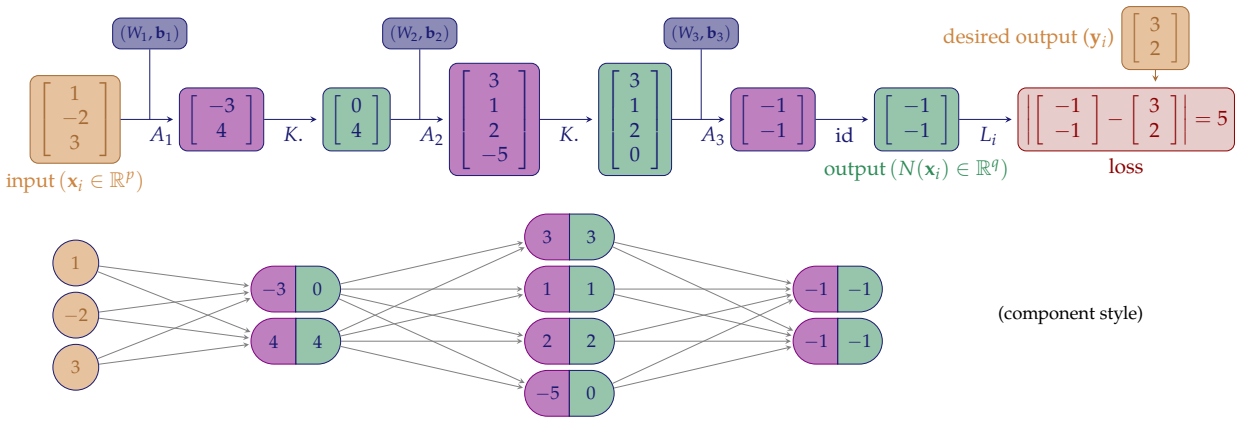


Figure 1.1 Multilayer-perceptron visualizations: vector style and component style. The ith training sample is denoted $(\mathbf{x}_i, \mathbf{y}_i)$.

* like $\mathbb{R}$ or $\mathbb{R}^2$ or $\mathbb{R}^3$, etc.

Note that the input and output values must be vectors in some fixed Euclidean spaces*. In practice we will often work with input values which are images or sentences and output values which are categorical, so some care will be required to encode these data as vectors. The choice of cost function (which we use to quantify how well matched the output and desired output are) is also important, and we will discuss some different possibilities.

We learn the parameters of the affine maps by successively applying the chain rule to find the derivative of the cost function with respect to each matrix W and vector $\mathbf{b}$. This procedure is called *backpropagation*. The idea is to take the derivative of each map in the above diagram and multiply the resulting matrices (the chain rule ensures that multiplying derivatives yields the derivative of the composition). For details, see the backpropagation section in the DATA 1010 lecture notes.

For a classification problem with C classes, we typically set up a network whose last layer has C components. The goal will be the get the neural network to return a vector which is approximately 1 in the position of the correct classification and approximately zero elsewhere. To facilitate this goal, we apply the *softmax*

transformation as the last step in the neural network. This function exponentiates each component of a vector and then normalizes to get a sum-of-components equal to 1. For example, the softmax of $[-1, 2, -3]$ is $\left[\frac{e^{-1}}{e^{-1}+e^{2}+e^{-3}}, \frac{e^{2}}{e^{-1}+e^{2}+e^{-3}}, \frac{e^{-3}}{e^{-1}+e^{2}+e^{-3}}\right]$. We can then interpret the output of the neural network as a *probability** vector which expresses the network's belief about the classification. For example, if the network returns $[0, 0.4, 0.6, 0]$, then it is sure that the input is not class 1 or 4, and it is only somewhat more confident that the correct class is 3 rather than 2.

* The term here means that the components sum to 1, not that they represent the answer to any question about a probability space.

When using softmax for classification, we measure the model accuracy based on how much weight is assigned to the correct classification. For example, we would give a small penalty to the vector $[0.1, 0.9, 0]$ for a class-2 sample and a large penalty to $[0.45, 0.01, 0.54]$ for a class-2 sample. Specifically, we will use the *cross-entropy* penalty, which is defined to be the negation of the log of the value in the position of the correct classification. For example, the penalties for the outputs $[0.1, 0.9, 0]$ and $[0.45, 0.01, 0.54]$ would be $-\log(0.9) = 0.105$ and $-\log(0.01) = 4.6$ (assuming the correct class for the given sample is 2).

1.2.1 Software demos

Visit **`https://playground.tensorflow.org`** for an excellent visualization tool to watch a neural network train in real time.

Note that the pictures shown at each node are *heatmaps* of the function on the (x_1, x_2)-plane which has x_1 and x_2 as the inputs on the first layer and outputs the value at that node.

29 January 2019

Visit **`https://www.tensorflow.org/tutorials/keras/basic_classification`** and work through the notebook in Colab. The basic Keras operations (for the toy example of the architecture in Figure 1.1 for the $\mathbb{R}^2$-valued function which sums the inputs and square-sums the inputs).

`basic-keras-example.ipynb`

1. Store training data as NumPy arrays.

```
import numpy as np
n = pow(10,4)
x_train = np.random.randn(4*n,3)
y_train = np.column_stack((np.sum(x_train,axis=1),
                           np.sum(pow(x_train,2),axis=1)))
x_test = np.random.randn(n,3)
y_test = np.column_stack((np.sum(x_test,axis=1),
                          np.sum(pow(x_test,2),axis=1)))
```

2. **Create** the model architecture.

```
import tensorflow as tf
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(units=2,activation='relu',input_dim=3))
model.add(tf.keras.layers.Dense(units=4,activation='relu'))
model.add(tf.keras.layers.Dense(units=2,activation='linear'))
```

3. **Compile** (specify the loss function, optimizer, and metrics to track during training):

```
model.compile(loss = 'mean_squared_error',
              optimizer = 'sgd', # stochastic gradient descent
              metrics = ['mean_squared_error'])
```

4. **Train** the model.

```
model.fit(x_train, y_train, epochs = 10, batch_size = 100)
```

5. **Evaluate** the model using the test data.

```
model.evaluate(x_test, y_test)
```

6. **Predict** output for a new sample.

```
model.predict(np.expand_dims([1,1,1],0))
```

7. **Save** the model.

```
model.save('my-model.h5')
# to reload:
model = tf.keras.models.load_model('my-model.h5')
```

1.2.2 A history of neural networks

Although neural networks have enjoyed their highest-profile successes in the past 8 years or so, the ideas go back decades. An unofficial timeline:

1943 McCulloch and Pitts proposed the first mathematical model of the neuron.

1950 Minsky and Edmonds build the first neural network machine (SNARC).

1958 Rosenblatt proposed the single-layer perceptron.

1961 Rosenblatt proposed the multi-layer perceptron.

1969 Minsky and Papert's book *Perceptrons* led to the first AI winter.

1970 Seppo Linnainmaa introduced the general backpropagation algorithm.

1974 Paul Werbos applied backpropagation to neural networks.

1986 Backpropagation applied and popularized by Rumelhart, Hinton, and Williams.

1987 The first International Conference on Neural Networks.

1989 Cybenko showed that neural networks can approximate arbitrary continuous functions.

2006 Hinton introduced the term "deep learning".

2010 Fei-Fei Li's team at Stanford built the ImageNet database of millions of labeled real-world images.

2012 Krizhevsky, Sutskever, and Hinton achieved breakthrough improvement on ImageNet.

2014 Facebook's system DeepFace achieves human-level facial recognition.

2016 Google DeepMind's AlphaGo defeated a human world champion Go player.

2018 Google DeepMind's AlphaZero, trained entirely on self-play, defeats best conventional chess engine

1.3 Neural network training

31 January 2019

1.3.1 Activations and losses

Keras includes several activation functions (`https://keras.io/activations`), including each of the following functions*.

...except signum

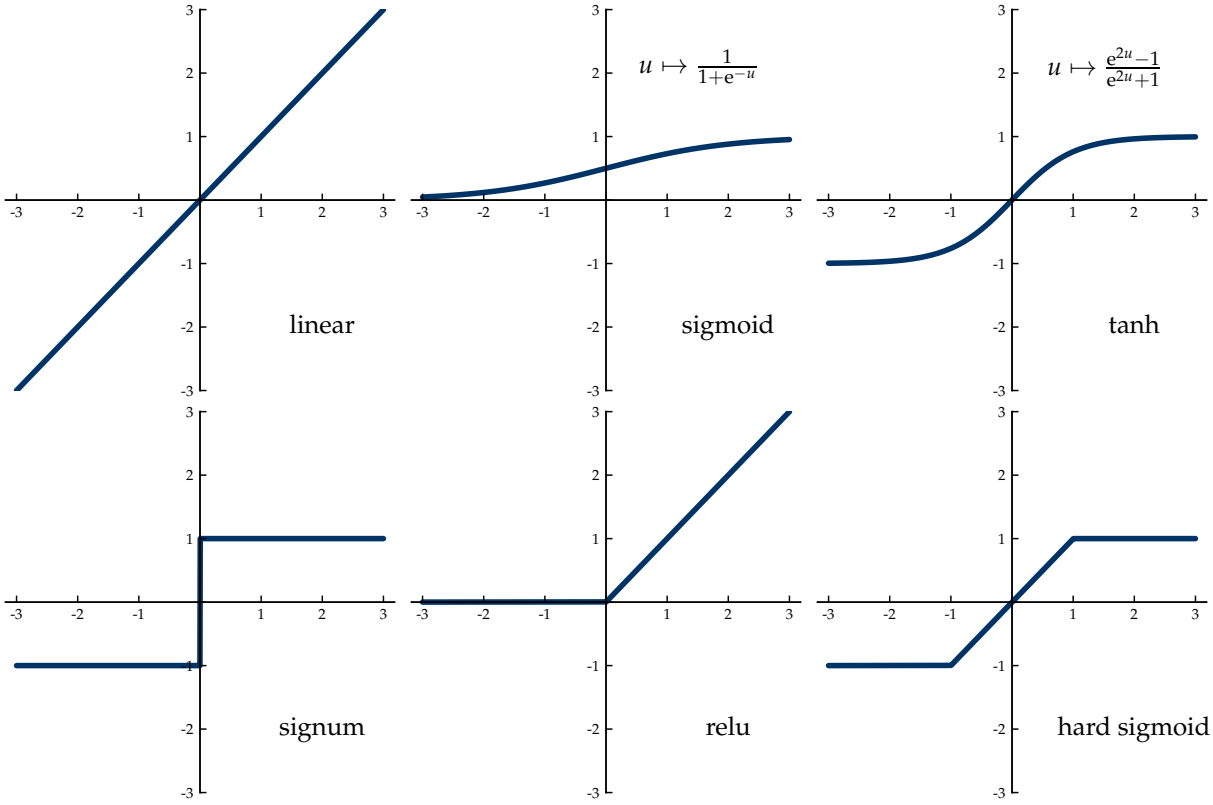


Figure 1.2 Some common activations.

The derivative of the signum function is zero almost everywhere, so it is not amenable to training through gradient-based methods. It's included here because it was important historically—the insight that led to the success of backpropagation was to replace discrete activations with differentiable ones.

Note that some of these functions **saturate**; that is, they map $\mathbb{R}$ to a bounded subset of $\mathbb{R}$.

The loss functions provided by Keras include (`https://keras.io/losses/`):

name	formula	output $\widehat{\mathbf{y}}$	target $\mathbf{y}$
mean squared error	$\lvert\mathbf{y}-\widehat{\mathbf{y}}\rvert^2$	vector or scalar	vector or scalar
logcosh	$\log\cosh(y-\widehat{y})$	scalar	scalar
cosine proximity	$-\frac{\mathbf{y}\cdot\widehat{\mathbf{y}}}{\lvert\mathbf{y}\rvert\lvert\widehat{\mathbf{y}}\rvert}$	vector	vector
cross entropy	$-\log(\mathbf{y}\cdot\widehat{\mathbf{y}})$	prob. vector*	indicator vector*
hinge loss	$\max(0,1-y\widehat{y})$	scalar	binary ($\{-1,+1\}$)

* A **probability vector** has non-negative entries which sum to 1

* An **indicator vector** is a probability vector with one entry equal to 1.

Though the formula for logcosh is given for scalars for simplicity, you can also apply it to vectors (by applying the scalar version componentwise and summing).

Exercise 1.3.1

Suppose that $\mathbf{y} = [1,2,3]$. For what values of $\widehat{\mathbf{y}}$ is the cosine proximity loss as small as possible?

Example 1.3.1

Calculate the logcosh loss and the mean squared loss when the model output value is $\widehat{y} = 3.3$ and the observed output value is $y = 3.2$. Repeat with $(\widehat{y}, y) = (9.1, 3.2)$.

Solution

We can see that the two penalties are comparable when the error is small, but MSE penalizes large errors much more harshly.

	MSE	logcosh
$(\widehat{y}, y) = (3.3, 3.2)$	0.01	0.005
$(\widehat{y}, y) = (9.1, 3.2)$	34.81	5.21

As indicated in the table, these loss functions are suitable for different situations. For example, hinge loss only applies if we get quantitative real values from the neural network and want to use them to solve a binary classification problem (this is how the support vector machine works). Likewise, cross entropy is appropriate if the last-layer activation is softmax. The mean squared error and logcosh losses are very similar for small deviations, but logcosh responds in a less extreme way to large ones (see Figure 1.3).

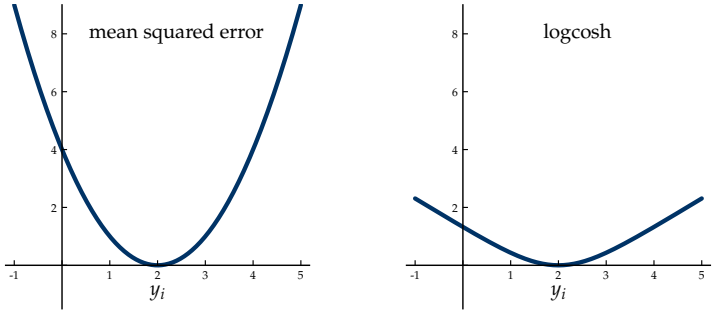


Figure 1.3 Quadratic error and logcosh error.

1.3.2 Overfitting and training issues

Neural networks often have many parameters and may **overfit** the training data. As a rule of thumb, it's a good idea to have at least two to three times as many training samples as parameters. Here are some approaches to dealing with the overfitting problem.

This section is a very high level overview. We are going to cover each item in more detail later in the course.

1. **Regularization**. Penalize large parameters by adding a multiple (λ) of the sum of the squares of the parameters to the objective function being minimized. This has the effect of changing the update rule for each weight from $w \leftarrow w - \alpha \frac{\partial L}{\partial w}$ to $w \leftarrow w(1 - 2\alpha\lambda) + \alpha \frac{\partial L}{\partial w}$, where α is the learning rate and L is the loss.
2. **Architecture**. Build the architecture to reflect an domain-specific knowledge. For example, convolutional neural networks for image data, recurrent neural networks for natural language data. Also, favor deep nets over wide ones, as the layering tends to make better use of parameters.
3. **Early stopping**. Halt the optimization process early, for example when the error on a witheld portion of the training data begins to rise.
4. **Dropout**. Remove each neuron independently with probability p (usually between $\frac{1}{5}$ and $\frac{1}{2}$) and train the reduced network on a mini-batch of samples. Repeat with a freshly sampled subset on the next mini-batch.

05 February 2019

Updates for early layers can be very small or very large for a deep network, because the corresponding gradients are obtained by multiplying many matrices (the **vanishing/exploding gradient problem**).

Exercise 1.3.2

Characterize the square symmetric matrices A with the property that A^n neither converges exponentially fast to 0 nor diverges to exponentially fast to ∞*.

* meaning that $\max_{|\mathbf{x}|=1} |A^n \mathbf{x}|$ converges to ∞ exponentially fast.

Some approaches to resolving the exploding gradient problem:

1. Use ReLU instead of sigmoid activation, since its derivative is 1 on the positive real line instead of being no more than $\frac{1}{4}$ everywhere.
2. Greedily pre-train the network layer-by-layer using unsupervised learning.

Announcements

1. Two mistakes from last time: I forgot the negative sign in the cosine proximity loss function, and the maximum derivative of the tanh function is 1, not $\frac{1}{2}$ (you get a factor of 2 from the vertical scaling, but you get another one from the horizontal scaling).
2. One change in notation from last time: let's use y and $\hat{y}$ as generic notation for the measured value and the model output value (in the table of loss functions).
3. A point I haven't explained clearly enough so far: the output of the softmax is a probability vector in the sense that the components sum to 1 and express a confidence; there is no underlying probability space with events corresponding to those probabilities.
4. Please remember that the TA office hours are being managed through SignMeUp `https://signmeup.cs.brown.edu`. It is required that you use that system rather than walking directly into office hours.

1.4 Convolutional neural networks

* Two kinds of equations: some which assert that specific weights are equal to zero, and some which assert equality between particular weights.

Convolutional neural networks provide one of the most important examples of a domain-specific custom neural network architecture. They can be fit into the multilayer perceptron framework by imposing some equations on the weights*, but it is clearer to think of the intermediate values in the neural network as rank-3 tensors* rather than as vectors.

* Recall that a *tensor* is a multidimensional array of numbers. A rank-0 tensor is a number, a rank-1 tensor is a vector, a rank-2 tensor is a matrix, a rank-3 tensor can be thought of as a rectangular prism full of numbers, and so on.

We use rank-3 tensors (even though images are typically 2D) for two reasons: (i) in the input layer, images can have different *channels* (like red-green-blue for color) that are naturally represented with a shallow third dimension, and (ii) it turns out that for practical purposes we need a stack of "images" in each hidden layer as well.

Convolutional neural networks are built using *convolution* and *subsampling* layers. Let's call a square subgrid of a given image a *stencil**.

Exercise 1.4.1

Show that

1. the set of $s \times s$ stencils in an $n \times n$ image form a $(n-s+1) \times (n-s+1)$ grid, and
2. partitioning an $n \times n$ grid into $s \times s$ stencils yields a $(n/s) \times (n/s)$ grid.

* As far as I can tell, this name is not normally used in this context. I'm borrowing the language from numerical differential equations.

Convolutional layer. We fix a stencil size s and define the second layer to be a tensor whose first two dimensions correspond to the grid of $s \times s$ stencils in the first layer and whose thickness t is arbitrary. Each image in the second layer is associated with an $s \times s \times t$ tensor called a *filter* together with a scalar b called the *bias*, and the pixel values for that image are obtained by dotting the filter with the corresponding stencil in the previous layer and adding the bias to the result. The thickness of the second layer is chosen as a part of the architecture, and the filter entries are parameters to be learned when the neural network is trained.

07 February 2019

Subsampling layer. The transformation from the second layer to the third layer applies a fixed function (commonly `max`) to each stencil in a partition of each image (typically 2×2 stencils are used). Subsampling substantially reduces the size of the images and therefore also the number of parameters to be learned.

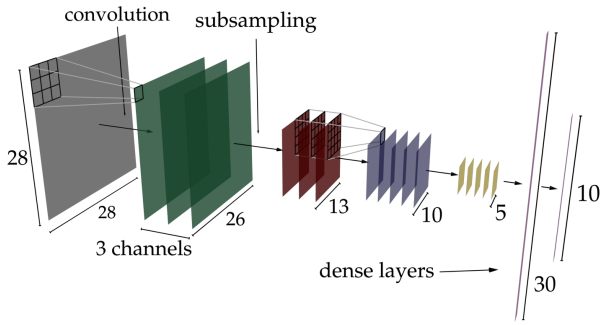


Figure 1.4 A convolutional neural network architecture.

* *may* here means that it is not ruled out by the architecture: we allow for the partial derivative to be zero as long as there is some choice of weights for which the partial derivative is nonzero.

We define the **receptive field** of a neuron in a neural network to be the set of neurons in the previous layers which may* contribute to its value. For example, the receptive field of the green corner pixel highlighted in

Figure 1.4 is the set of 9 pixels in the corresponding stencil in the input layer.

Exercise 1.4.2

Consider the neural net in Figure 1.4. How many pairs of neurons v_1 and v_2 have the property that v_1 is in the input layer, v_2 is in the second layer, and v_1 is in the receptive field of v_2?

Why do you think Keras calls ordinary multilayer perceptron layers `Dense`? (Hint: if the input and first hidden layer were serialized and treated as an ordinary dense layer, how many weights would there be between the two layers?)

Sparse matrices: an aside

Many important matrices in scientific computing applications have mostly zero entries. Such matrices can be stored in a special *sparse matrix* data structure which records only the locations of the nonzero entries. Accordingly, the usual matrix representation (in which all of the entries are stored) is called *dense*.

1.4.1 CNNs in Keras

The following example is based on Notebook 5.1 in Chollet's book. We'll take another look at the Fashion MNIST data set with a convolutional neural network.

We begin with some convolutional layers with a 3×3 stencil interleaved with max pooling layers with a 2×2 stencil.

`cnn-first-example.ipynb`

```
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras import layers
from tensorflow.keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Next, we flatten the tensor to set up a couple of dense layers. The last layer has 10 neurons and uses the softmax activation since the fashion MNIST problem has 10 classes.

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

We pull in the fashion MNIST data set and reshape the arrays to expose the single channel of each grayscale image (in other words, the training data tensor needs to have rank 4: (*samples, height, width, channels*, and if

the channel axis is not present in the data, we need to create it)). Note that you can also create a new axis with `train_images[:,:,:,np.newaxis]`.

```
from tensorflow.keras.datasets import fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
train_images = train_images[:,:,:,np.newaxis]
test_images = test_images[:,:,:,np.newaxis]
```

* we will get into details about how these optimizers work later in the course.

Finally, we compile the model with the `'rmsprop'` optimizer* and train it.

```
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
model.evaluate(test_images, test_labels)
```

We get a test accuracy of about 87.7%.

Since both the training data and the activations carry spatial meaning, convolutional neural nets are particular amenable to visualization.

We can access the activations of a model by indexing its `layers` attribute and making a new model with those layers' outputs as its outputs. For example, we can inspect the first five layers as follows:

```
layer_outputs = [layer.output for layer in model.layers[:5]]
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
activations = activation_model.predict(train_images[:10])
```

12 February 2019

Now `activations` is a list of rank-4 tensors. The ith training sample, j-th channel image in the kth layer is `activations[k][i,:,:,j]`. Let's take a look at these images.

```
def layerimages(image_idx, layer_idx, images_per_row = 8):
    channel_count = activations[layer_idx].shape[3]
    r = np.ceil(channel_count/images_per_row)
    plt.figure(figsize=(12,12*r/images_per_row))
    for channel_idx in range(channel_count):
        plt.subplot(r,8,channel_idx+1)
        plt.grid(False); plt.xticks([]); plt.yticks([])
        plt.imshow(activations[layer_idx][image_idx,:,:,channel_idx])
    plt.show()

def all_layers(image_idx):
    plt.imshow(train_images[image_idx,:,:,0])
    plt.grid(False); plt.xticks([]); plt.yticks([])
    for i in range(5):
        layerimages(image_idx,i)

all_layers(0)
```

1.4.2 Transfer learning

If a convolutional neural network is trained on a large and diverse set of images, then the early layers learn features which are generally useful for recognition tasks. So we can benefit from reusing the initial layers of models which are known to perform well on such data sets. This is an important enough technique that Keras comes with these models built in.

This section is based on Section 5.3 in *Chollet*.

Let's begin by extracting the convolutional layers from the VGG16 model. We set `include_top` to `False`, because we're going to use our own dense layers. We specify the input shape to be `(56,56,3)` because three channels are required for VGG16, and the smallest image accepted is 32×32 (so we're going to upsample our 28×28 images to 56×56 images).

`transfer-learning.ipynb`

```
from tensorflow.keras.applications import VGG16
conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(56,56,3))
```

Now let's add some densely connected layers.

```
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential()
model.add(keras.layers.UpSampling2D((2,2)))
model.add(conv_base)
model.add(keras.layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

The convolutional base is very large and expensive to train, and anyway we only want to train the dense layers we added:

```
conv_base.trainable = False
```

Now let's train the model.

```
model.compile(optimizer='rmsprop',loss='sparse_categorical_crossentropy')
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

Finally, we evaluate it.

```
model.evaluate(test_images, test_labels)
```

1.5 Other deep learning topics

1.5.1 Autoencoders

An autoencoder is a neural network which is trained to approximate the identity function for a given set of samples. If the hidden layers have fewer neurons that the input and output layers, then it is not typically possible to approximate the identity perfectly. As a result, the autoencoder learns hidden layer features from which the original data can best be reconstructed. Those features can then serve as a lower-dimensional representation of the data for other purposes.

autoencoder.ipynb

Here's a really simple autoencoder implementation with one dense 32-neuron layer:

```
import matplotlib.pyplot as plt

from tensorflow.keras import layers
from tensorflow.keras import models

model = models.Sequential()
model.add(layers.Flatten(input_shape=(28,28)))
model.add(layers.Dense(units=784, activation='relu'))
model.add(layers.Dense(units=32, activation='relu')) # encoded
model.add(layers.Dense(units=784, activation='sigmoid'))
model.add(layers.Reshape((28,28))) # decoded
model.compile(optimizer='adadelta', loss='binary_crossentropy')
```

Wwe call the 32-unit layer the *encoded layer* and the output layer the *decoded layer*. We'll train this autoencoder on fashion MNIST.

```
from tensorflow.keras.datasets import fashion_mnist
(x_train, _), (x_test, _) = fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

model.fit(x_train, x_train, epochs=5, batch_size=128)
```

Let's see how similar each output image looks to the corresponding input image.

```
decoded_images = model.predict(x_test)

def before_and_after(i):
    plt.subplot(2,1,1)
    plt.imshow(x_test[i])
    plt.subplot(2,1,2)
    plt.imshow(decoded_images[i])
    plt.show()

for i in range(10):
    before_and_after(i)
```

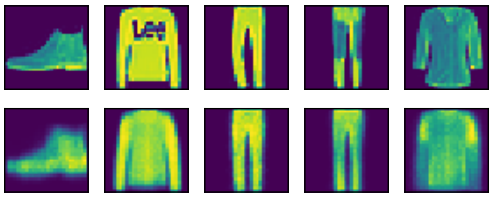


Figure 1.5 Input and output of a dense autoencoder with 32 neurons in the encoded layer.

1.5.1.1 Variational autoencoders

14 February 2019

To build a variational autoencoder, we inject Gaussian noise in the middle of the network and still try reproduce the input training samples in the output layer. The encoding produces *two* vectors, one of which serves as a mean and the other as a diagonal log-covariance matrix for a multivariate Gaussian which is fed into the decoding layers. In addition to trying to match input and output, we will also try to make the mean and log-covariance vectors close to zero. Then we can use a trained variational autoencoder to generate new samples which look like they were drawn from the distribution of the training set (by sampling from the standard Gaussian distribution and mapping the result through the decoding layers).

We will penalize the mean $\boldsymbol{\mu}$ and log-covariance vector Λ according to the *KL divergence* between the standard multivariate Gaussian and the multivariate Gaussian with mean $\boldsymbol{\mu}$ and covariance e^{Λ}.

Theorem 1.5.1: KL divergence

The KL divergence between two continuous probability distributions with densities p and q on $\mathbb{R}^n$ is defined to be

$$\mathrm{KL}(p\|q) = \int_{\mathbb{R}^n} p(x) \log \frac{p(x)}{q(x)} \, \mathrm{d}x.$$

The KL divergence of the normal distribution $\mu \in \mathbb{R}^n$ and covariance Σ with respect to the standard normal dimension is

$$\mathrm{KL}\left(\mathcal{N}(\mu, \Sigma) \| \mathcal{N}(0, I)\right) = -\frac{1}{2}\left(\log\det\Sigma - \mathrm{trace}(\Sigma) - |\mu|^2 + n\right).$$

The architecture of a variational autoencoder is shown in Figure 1.6. The encoding layers of the model output *two* vectors, one of which is interpreted as a mean vector and the other as the log of a diagonal covariance matrix. These data are combined with a standard normal random vector to produce a random vector with the given mean and covariance, and the result is fed into the decoding layers.

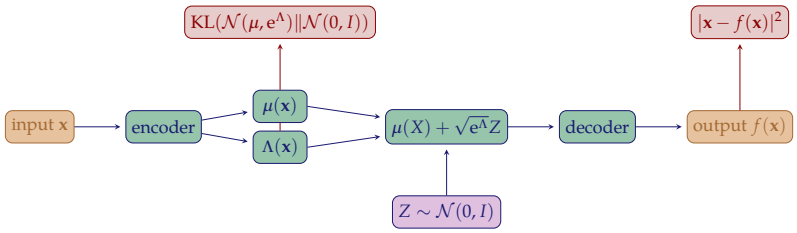


Figure 1.6 A variational autoencoder.

Let's implement a variational autoencoder. The novel architecture will give us an opportunity to explore more Keras features.

This example is based on this Keras blog post

We begin with some imports and standard `fashion_mnist` data loading. Note the use of the `-1` entry in the `reshape` call; this tells NumPy to infer that dimension.

`variational-autoencoder.ipynb`

```
import matplotlib.pyplot as plt
import numpy as np

import tensorflow.keras as keras
import tensorflow.keras.layers as layers
import tensorflow.keras.models as models
import tensorflow.keras.backend as K

from tensorflow.keras.datasets import fashion_mnist
(x_train, _), (x_test, _) = fashion_mnist.load_data()

image_size = x_train.shape[1]
original_dim = image_size * image_size
x_train = x_train.reshape([-1, original_dim]).astype('float32')/255.0
x_test = x_test.reshape([-1, original_dim]).astype('float32')/255.0

batch_size = 100
intermediate_dim = 512
encoded_dim = 20
epochs = 5
```

Next we define a Keras-based normal sampler and the variational autoencoder loss function. Note that we are using the `K`-prefixed versions of `shape`, `exp`, `mean`, and `square` because these functions are defined to handle Keras `Tensor` objects (which act as placeholders for data that will not be supplied until training time). Also, we are calculating the input-output discrepancy loss using cross entropy rather than the mean squared error shown in Figure 1.6.

```
def sample_normal(args):
    "Sample from the normal distribution with given mean and variance"
    mean, logsigma = args
    batch, dim = K.shape(mean)[0], K.int_shape(mean)[1]
    Z = K.random_normal(shape=(batch, dim))
    return mean + K.exp(0.5 * logsigma) * Z

def vae_loss(x, x_decoded):
    "Return cross-entropy loss plus divergence of encoded layer from standard normal"
    crossentropy_loss = keras.losses.binary_crossentropy(x, x_decoded)
    kl_loss = - 0.5 * K.mean(1 + logsigma - K.square(mean) - K.exp(logsigma), axis=-1)
    return crossentropy_loss + kl_loss
```

Now we can build our models. We will want access to several segments of the full model, so we will build it up one step at a time.

```
x = layers.Input(shape=(original_dim,))
h = layers.Dense(intermediate_dim, activation='relu')(x) # hidden encoding layer
mean = layers.Dense(encoded_dim)(h) # μ(x) layer
logsigma = layers.Dense(encoded_dim)(h) # Λ(x) layer
sample = layers.Lambda(sample_normal)([mean, logsigma]) # custom layer

encoder = models.Model(x, [mean, logsigma, sample])

encoded_inputs = layers.Input(shape=(encoded_dim,))
h2 = layers.Dense(intermediate_dim, activation='relu')(encoded_inputs) # hidden decoding layer
```

```
outputs = layers.Dense(original_dim, activation='sigmoid')(h2) # output layer

decoder = models.Model(encoded_inputs, outputs)
vae = models.Model(x, decoder(encoder(x)[2])) # compose to get full variational autoencoder
```

We're ready to train the model:

```
vae.compile(optimizer='adam', loss = vae_loss)
vae.fit(x_train, x_train,
        epochs = epochs,
        batch_size = batch_size)
```

As advertised, we can use the trained model to generate random samples from the training distribution by applying the decoding transformation to a standard normal random variable.

```
def random_image():
    return decoder.predict(
      np.random.randn(encoded_dim)
      [np.newaxis, :]).reshape((28,28))

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.imshow(random_image())
    plt.axis('off')
plt.show()
```

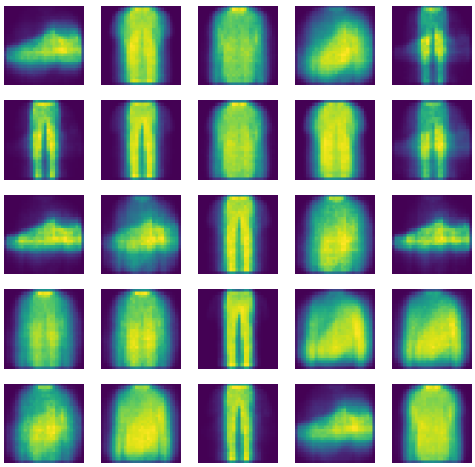


Figure 1.7 Images generated from random noise by the decoding portion of a variational autoencoder.

1.5.2 Recurrent neural networks

All of the neural networks we've studied so far have been *feedforward* neural networks: the connections between nodes do not form cycles. Such networks are not ideal for sequential data (like text or audio) because they don't have a natural mechanism for modeling the sequential structure of the data. *Recurrent* neural networks remedy this problem by introducing loops between nodes.

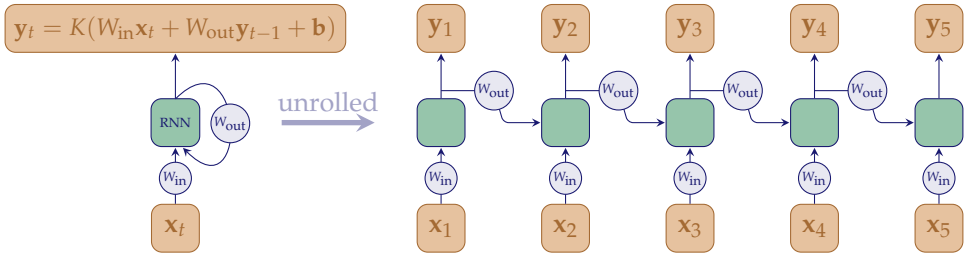


Figure 1.8 A simple recurrent neural network.

* A cell is a neural network layer which preserves state across time steps.

The simplest recurrent neural network (Figure 1.8) consists of a single recurrent *cell**. The input data are fed into the cell sequentially, and the cell also outputs a sequence. The output at each time step depends on the input at that time step and the output at the previous time step. Mathematically, we have

$$\mathbf{y}_t = K(W_{\text{in}}\mathbf{x}_t + W_{\text{out}}\mathbf{y}_{t-1} + \mathbf{b}),$$

21 February 2019

where K is the cell's activation function, W_{in} and W_{out} are matrices and $\mathbf{b}$ is a vector.

Let's apply a recurrent neural network to solve the fashion MNIST classification problem (interpreting the images as sequences of pixels). We'll feed the images in one row of pixels at time, so there will be 28 input vectors, each of which has 28 components. We will only need a single ten-component vector as the output of the model, so we will only be using the final output value (the top right corner of Figure 1.8). Also, we will add a dense layer map the output to $\mathbb{R}^{10}$.

recurrent-net.ipynb

```
import matplotlib.pyplot as plt
import numpy as np

import tensorflow.keras as keras
import tensorflow.keras.layers as layers
import tensorflow.keras.models as models

from tensorflow.keras.datasets import fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

image_size = x_train.shape[1]
x_train = x_train.astype('float32')/255.0
x_test = x_test.astype('float32')/255.0
```

We build a recurrent neural net similarly to a feedforward neural net. Note that the input shape should be supplied in the form (number of time steps, number of features).

```
model = models.Sequential()
model.add(layers.SimpleRNN(50, input_shape = (28,28)))
model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer = 'adam',
              loss = 'sparse_categorical_crossentropy',
              metrics = ['accuracy'])
```

Now let's train the model and evaluate it.

```
model.fit(x_train, y_train, epochs = 5, batch_size = 100)
model.evaluate(x_test, y_test)
```

We get an accuracy of around 82%, which is significantly lower than the accuracy we got with a convolutional neural net. This is not surprising, since convolutional nets directly account for spatial relationships between pixels.

We conclude this section by noting that `SimpleRNN` cells are rarely used in practice, because you can get better results by replacing them with a compound cell (which packages standard configuration of smaller cells inside it). The most popular such cells are called `LSTM` (*long short-term memory*) and `GRU` (*gated recurrent unit*).

1.5.3 Reinforcement learning

Consider the problem of training a computer to play a video game like Mario Brothers. This task does not fit the statistical learning paradigm, because decisions must be made sequentially, and decisions affect the subsequently observed data. Furthermore, the loss function is also incrementally measurable (since the goal of the game is to advance to the right without dying). Therefore, we might train an agent to play the game by having it initially choose moves randomly and update its decision algorithm as it receives continuous feedback on whether it is achieving desirable outcomes. This is called *reinforcement learning*.

1.5.4 Generative adversarial networks

Imagine you're an artist whose goal is to draw photorealistic pictures. You'll start by drawing some pictures, recognizing what doesn't look real about them, and using that information to improve your skill. At some point you'll want to recruit a second person to try distinguishing your pictures from photographs (you have to be skeptical of your own ability to do this, since you drew the pictures). But maybe that person isn't as good as they could be at distinguishing photos from photorealistic drawings, so they should practice and get better at it. The two of you can do this training in tandem as you both improve: your objective is to reduce the critic's success to guessing rates, and their goal is to distinguish perfectly.

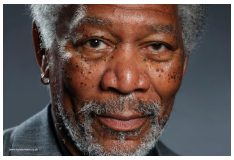


Figure 1.9 A digital painting by Kyle Lambert.

We can use the same idea neural networks. Given a set of (unlabeled) training samples in $\mathbb{R}^n$, we jointly train one neural network to generate samples in $\mathbb{R}^n$ and a second neural network to distinguish the generated samples from the real training samples. This is called a *generative adversarial network*.

2 Neural networks and other ML models

In this chapter, we will develop several connections between neural networks and classical machine learning algorithms.

2.1 Support vector machines

We begin with an overview of the mathematical theory underlying the support vector machine. You can find discussions of this material in *Hands-on ML* and in *Elements of Statistical Learning*, but this presentation is more condensed, with notation entirely vectorized for convenient translation to code.

A **support vector machine** is a binary classifier of the form

$$\mathbf{x} \mapsto \operatorname{sign}(\mathbf{x} \cdot \mathbf{w} + b),$$

where $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$. To train a support vector machine on a set of training data $(X, \mathbf{y})$, with $X \in \mathbb{R}^{n \times d}$ and $\mathbf{y} \in \{-1, 1\}^n$, we choose a value $C > 0$ and solve the optimization problem

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C \operatorname{sum}(\boldsymbol{\zeta}) \\ \text{subject to} \quad & \mathbf{y} \odot (X\mathbf{w} \oplus b) \succcurlyeq 1 \ominus \boldsymbol{\zeta} \text{ and } \boldsymbol{\zeta} \succcurlyeq 0. \end{aligned}$$

where $\odot$ and $\oplus$ indicate elementwise multiplication and addition, respectively (denoted by `.*` and `.+` in Julia and by `*` and `+` in NumPy), and $\succcurlyeq$ indicates that the elementwise comparison holds for every element. The parameter C governs the tradeoff between the margin width term $\frac{1}{2}\|\mathbf{w}\|^2$ and the classification penalty $\operatorname{sum}(\boldsymbol{\zeta})$.

* derivation omitted. See Section 12.2.1 *Computing the Support Vector Classifier* in *Elements of Statistical Learning* for details.

It turns out that this problem can be solved by instead solving the *dual* problem*

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}(\boldsymbol{\alpha} \odot \mathbf{y})' XX' (\boldsymbol{\alpha} \odot \mathbf{y}) - \operatorname{sum}(\boldsymbol{\alpha}) \\ \text{subject to} \quad & 0 \preccurlyeq \boldsymbol{\alpha} \preccurlyeq C \text{ and } \boldsymbol{\alpha} \cdot \mathbf{y} = 0, \end{aligned}$$

where the optimizing value $\widehat{\boldsymbol{\alpha}}$ for the dual problem is related to the optimizing value $\widehat{w}$ for the original problem via

$$\widehat{\mathbf{w}} = X'(\widehat{\boldsymbol{\alpha}} \odot y).$$

The optimizing value of b in the original problem may also be obtained using the solution of the dual problem by looking at any entry of

$$\mathbf{y} - X\widehat{\mathbf{w}}$$

for which the corresponding entry of $\boldsymbol{\alpha}$ is strictly between 0 and C. All such entries can be proved to be equal mathematically, but when working with numerical approximations, (i) a small tolerance should be included when determining which entries of $\boldsymbol{\alpha}$ are between 0 and C, and (ii) the appropriate entries of $\mathbf{y} - X\widehat{\mathbf{w}}$ should be averaged.

The reason for formulating the dual problem is that it permits the application of a useful and extremely common technique called the **kernel trick**. The idea is that if we apply a transformation $\phi : \mathbb{R}^d \to \mathbb{R}^d$ each

row of X and call the resulting matrix $\phi(X)$, then the resulting change to the dual problem is just to replace XX' with $\phi(X)\phi(X)'$. This matrix's entries consist entirely of dot products of rows of $\phi(X)$ with rows of $\phi(X)$, so we can solve the problem as long as we can calculate $\phi(\mathbf{x}) \cdot \phi(\mathbf{y})$ for all $\mathbf{x}$ and $\mathbf{y}$ in $\mathbb{R}^d$. The function $K = (\mathbf{x}, \mathbf{y}) \mapsto \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$ is called the *kernel* associated with the transformation ϕ. Typically we ignore ϕ and use one of the following kernel functions:

$$\begin{aligned} \text{linear} \quad & K(\mathbf{x}, \mathbf{y}) = \mathbf{x}'\mathbf{y} \\ \text{degree-}d\text{ polynomial} \quad & K(\mathbf{x}, \mathbf{y}) = (\gamma \mathbf{x}'\mathbf{y} + r)^d \\ \text{Gaussian radial basis function} \quad & K(\mathbf{x}, \mathbf{y}) = \exp(-\gamma \|\mathbf{x} - \mathbf{y}\|^2) \\ \text{sigmoid} \quad & K(\mathbf{x}, \mathbf{y}) = \tanh(\gamma \mathbf{x}'\mathbf{y} + r), \end{aligned}$$

where γ and r are parameters that may be tuned.

To bring it all together, suppose that K is a kernel function and $\mathcal{K} = \{K(\mathbf{x}_i, \mathbf{x}_j)\}_{1 \le i,j \le n}$ is the resulting matrix of kernel values (where $\mathbf{x}_i$ is the ith row of X). Then the support vector machine with kernel K is obtained by letting $\widehat{\boldsymbol{\alpha}}$ be the optimizing vector $\boldsymbol{\alpha}$ in the optimization problem

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}(\boldsymbol{\alpha} \odot \mathbf{y})'\mathcal{K}(\boldsymbol{\alpha} \odot \mathbf{y}) - \text{sum}(\boldsymbol{\alpha}) \\ \text{subject to} \quad & 0 \preccurlyeq \boldsymbol{\alpha} \preccurlyeq C \text{ and } \boldsymbol{\alpha} \cdot \mathbf{y} = 0. \end{aligned}$$

The prediction vector for an $n_{\text{train}} \times n$ feature matrix X_{test} is

$$\text{sign}(\phi(X)\widehat{\mathbf{w}} \oplus b) = \text{sign}(\phi(X)\phi(X)'(\widehat{\boldsymbol{\alpha}} \odot \mathbf{y}) \oplus \widehat{b}) = \text{sign}(\mathcal{K}_{\text{test}}(\widehat{\boldsymbol{\alpha}} \odot \mathbf{y}) \oplus \widehat{b}),$$

where $\mathcal{K}_{\text{test}}$ is the $n_{\text{test}} \times n$ matrix whose (i,j)th entry is obtained by applying K to the ith row of X_{test} and the jth row of X, and where $\widehat{b}$ is any entry of

$$\mathbf{y} - \mathcal{K}(\widehat{\boldsymbol{\alpha}} \odot \mathbf{y})$$

for which the corresponding entry in $\widehat{\boldsymbol{\alpha}}$ is strictly between 0 and C.

Constrained convex optimization problems like the ones above can be solved in Python using a package called **cvxpy**. For example, to minimize $\mathbf{x}'Q\mathbf{x} + A\mathbf{x}$ subject to $\mathbf{x} \succcurlyeq 0$, where Q and A are represented as $n \times n$ NumPy arrays, we could run:

cvxpy.ipynb

```
import cvxpy as cp
x = cp.Variable(n) # create a symbolic vector of length n
objective = cp.Minimize(cp.quad_form(x, Q) + A*x) # quad_form(x,Q) means x'Qx
constraints = [x >= 0] # a list of symbolic equations/inequalities
problem = cp.Problem(objective, constraints)
problem.solve()
x.value # solving assigns a variable's optimizing value to its value attribute
```

26 February 2019

2.2 A neural architecture for the support vector machine

Note that in the SVM optimization problem

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\,\text{sum}(\boldsymbol{\zeta}) \\ \text{subject to} \quad & \mathbf{y} \odot (X\mathbf{w} \oplus b) \succcurlyeq 1 - \boldsymbol{\zeta} \text{ and } \boldsymbol{\zeta} \succcurlyeq 0, \end{aligned}$$

the minimizing $\boldsymbol{\zeta}$ must have a zero entry in any position where $\mathbf{y} \odot (X\mathbf{w} \oplus b)$ is at least 1, and otherwise it must be only as large as the difference between $\mathbf{y} \odot (X\mathbf{w} \oplus b)$ and 1. In other words, if we define $u_+ = \max(0, u)$ for $u \in \mathbb{R}$, we can rewrite the optimization problem in the form

$$\text{minimize} \quad \frac{1}{2}\|\mathbf{w}\|^2 + C\,\text{sum}([1 \ominus \mathbf{y} \odot (X\mathbf{w} \oplus b)]_+)$$

with no constraints. From this point of view, we can see that the support vector machine is equivalent to a single-layer neural network with d input neurons and one output neuron, with L_2 regularization, linear activation and the **hinge loss** function*

* Here and below, the loss function specifies loss for each sample, with the total loss being the sum of the sample losses.

$$L(y, \widehat{y}) = \max(0, 1 - y\widehat{y}).$$

Exercise 2.2.1

Use the above observation to implement support vector machine with linear kernel in Keras (which supports `loss = 'hinge'`). Compare your results to the output of the Scikit-learn support vector classifier on a simulated dataset.

2.3 Linear regression

Linear regression is equivalent to a single-layer neural network with an input dimension of d, an output dimension of 1, linear activation, and a mean-squared error loss function

$$L(y, \widehat{y}) = (y - \widehat{y})^2.$$

Exercise 2.3.1

Implement a linear regression model in Keras. Compare your results to the output of `sklearn.linear_model.LinearRegression`.

2.4 Logistic regression

Logistic regression is equivalent to a single-layer neural network with an input dimension of d, an output dimension of 1, sigmoid activation, and a loss function

$$L(y, \widehat{y}) = -\log|1 - y - \widehat{y}|,$$

assuming $\{0, 1\}$-encoded classes.

Exercise 2.4.1

Implement a logistic regression model in Keras. Compare your results to the output of `sklearn.linear_model.LogisticRegression`.

2.5 Matrix factorization

Suppose that A is an $n \times n$ matrix, and suppose k is less than n. The rank-k matrix with minimal distance to A (as measured by sum of squared differences between entries, called the *Frobenius distance*) can be computed in terms of the singular value decomposition of A:

Theorem 2.5.1

Suppose that $A = U\Sigma V'$ is the singular value decomposition of a matrix A. For $k \geq 0$, denote by Σ_k the matrix obtained by replacing all but the first k diagonal entries of Σ with zero. Then the rank-k matrix B which minimizes the Frobenius distance $\|A - B\|_F$ is $B = U\Sigma_k V'$.

Proof

Let B be the matrix which minimizes $\|A - B\|_F$. The column space of B is equal to the column space of $A_k = U\Sigma_k V'$, since the column space of A_k is the k-dimensional subspace of $\mathbb{R}^n$ whose sum-of-squared distances to the columns of A is as small as possible (as we learned in DATA 1010).

Given that B and A_k have the same column space, the sum of squared differences between the first column of A and the first column of B is as small as possible when the first column of B is the projection of the first column of A onto the column space of A_k. The first column of A is $U\Sigma V'\mathbf{e}_1$, where $\mathbf{e}_1 = [1, 0, 0, \ldots, 0]$, and the matrix which projects into the column space of A_k is $U_k U_k'$, where U_k is the $n \times k$ submatrix of U obtained by retaining the first k columns. Therefore, the first column of B is $U_k U_k' U\Sigma_k V'\mathbf{e}_1 = U_k(U_k' U)\Sigma_k V'\mathbf{e}_1 = U_k I\Sigma_k V'\mathbf{e}_1 = A_k\mathbf{e_1}$. So the first column of B is equal to the first column of A_k. Similarly, every column of B is equal to the corresponding column of B. Thus $B = A_k$.

On the other hand, finding the rank-k matrix closest to A in the sum-of-squared-differences sense is exactly what a two-layer autoencoder does, assuming that we use linear activation and no bias. Let's run this computation both ways for a 50×50 random matrix and compare the results. We'll begin by computing A_k directly (with $k = 10$).

svd-with-autoencoder.ipynb

```
import tensorflow.keras.layers as layers
import tensorflow.keras.models as models
import matplotlib.pyplot as plt
import numpy as np
import numpy.linalg.norm as norm

np.random.seed(123)
A = np.random.randn(50,50)
U, S, Vt = np.linalg.svd(A);
S[10:] = 0
A_k  = U @ np.diag(S) @ Vt
```

Now let's build and train an autoencoder to perform the same task.

```
svd = models.Sequential()
svd.add(layers.Dense(10, use_bias = False))
svd.add(layers.Dense(50, use_bias = False))
svd.compile(optimizer='sgd', loss = 'mse')
svd.fit(A, A, epochs = 10000)
```

We can use `pcolor` to visualize the two matrices.

```
plt.matshow(A_k)
plt.matshow(svd.predict(A))
```

They appear to be pretty close. We can confirm:

```
norm(A_k - svd.predict(A))
```

returns `0.0107`, so they are reasonably close (but well short of machine precision).

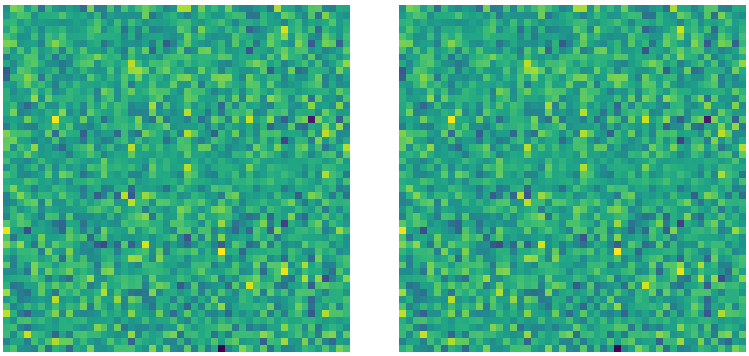


Figure 2.1 The rank-10 matrix closest to a random 50×50 matrix, obtained using the singular value decomposition (left) and using an autoencoder (right).

The neural network solves this problem vastly more slowly than standard SVD algorithms. However, it is also more flexible than the purely linear algebraic approach. For example, we can experiment with nonlinear activations, more layers, regularization, etc., by applying straightforward modifications to the code above.

3 Universal approximation and interpretability

3.1 Universal approximation

28 February 2019

Neural networks can approximate any continuous function from a closed, bounded subset of $\mathbb{R}^n$ to $\mathbb{R}^m$ arbitrarily well. This fact is called the **universality theorem**. An accessible and well-explained exposition of this result, complete with interactive Javascript figures, is available at

`http://neuralnetworksanddeeplearning.com/chap4.html`

The basic idea (for the $\mathbb{R}$ to $\mathbb{R}$ case) is to use two neurons in a single hidden layer to build a function which is approximately the indicator of an arbitrary interval, and then add more pairs of neurons to the hidden layer to control the value of the function's output on each of many small intervals.*

* So actually *shallow* networks are universal approximators.

3.2 Interpretability

Deep learning **interpretability** is human understanding of how a neural net works. Interpretation techniques include (1) **attribution** (ascertaining which components of a given input vector are most important for determining its output) and (2) feature visualization (visualizing the role of a single neuron or set of neurons in a network).

3.2.1 Attribution

Consider an image-classifying neural network. To determine which parts of a given input image are most important for the model's predicted classification of that image, we can ask which pixels maximize the derivative of the correct-class output value with respect to the pixel's value. This derivative is a byproduct of the backpropagation algorithm, so it is straightforward to compute in Keras. Let's walk through an example with the VGG16 model.

`attribution.ipynb`

```
from tensorflow.keras import backend as K
from tensorflow.keras import applications
from tensorflow.keras import activations
from tensorflow.keras import models
from tensorflow.keras import layers

# for manipulating images
from PIL import Image
from PIL import ImageFilter

import numpy as np
```

```
def sigmoid(x, k = 1):
    return 1/(np.exp(-x*k) + 1)

# Load VGG16 model
model = applications.VGG16(include_top=True,
                           weights='imagenet')
```

The softmax activation poses some difficulties with attribution and feature visualization, because it entangles the class activations and allows a particular class neuron to get a larger activation merely by decreasing the pre-softmax values associated with the other classes. Thus we consider the class *logits* (that is, the values obtained just prior to applying the softmax function). To do this in Keras, we have to remove the last layer, change its activation function, and add it back.

```
last_layer = model.layers[-1]
last_layer.activation = activations.linear
model = models.Model(inputs = model.input, outputs = last_layer(model.layers[-2].output))
```

Next we load an image of a dog, resize the image to 224×224 (the *ImageNet* dimensions), determine the index of this image's classification under the model, and define the desired gradient. We wrap this gradient Keras object as a Python function using `keras.backend.function`.

```
img = Image.open("dog.jpg").resize((224,224))
img_array = np.array(img)[np.newaxis,:,:,:]

img_class = np.argmax(model.predict(img_array))

grad = K.gradients(model.layers[-1].output[0,img_class], model.inputs[0])
grad_func = K.function(model.inputs, grad)
```

Finally, we can calculate the array of saliencies.

```
saliency_array = grad_func([img_array])[0][0,...]
```

We can display this image by converting to an `Image` object. We first map the array values to $[0, 1]$.

```
saliency_array -= np.min(saliency_array)
saliency_array /= np.max(saliency_array)
Image.fromarray((255*saliency_array).astype('uint8'), "RGB")
```

Figure 3.1 includes a bit more processing for aesthetic purposes, including the use of an opacity channel, `Image.alpha_composite` to superimpose the images, and a Gaussian blur.

Figure 3.1 The pixels with respect to which the derivative of the "labrador retriever" neuron output is largest.

3.2.2 Feature visualization

In addition to understanding how a network sees a particular input image, we can also try to visualize the components of neural network itself (neurons, layers, or sets of neurons like filters). We'll use the idea of trying to maximize a neuron's (or a set of neurons' average) activation. We can perform this optimization either (i) over a set of testing or training input vectors, or (ii) over the whole input space, using backpropagation and gradient ascent.

The following example is based on Chollet's Keras blog post *How convolutional neural networks see the world*. `feature-viz.ipynb`

```
from tensorflow.keras import applications
import tensorflow.keras.backend as K

model = applications.VGG16(include_top=False,weights='imagenet')
layer_dict = dict([(layer.name, layer) for layer in model.layers])
```

Next we define the function we want to minimize.

```
input_img = model.inputs[0]
layer_output = layer_dict['block3conv3'].output
loss = K.mean(layer_output[:, :, :, 0]) # 0 is the filter index
grads = K.gradients(loss, input_img)[0]
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5) # normalize gradient
iterate = K.function([input_img], [loss, grads]) # create a function
```

Finally, we choose a random starting image with pixel values near the middle of the interval $[0, 255]$ and perform gradient ascent.

```
input_img_data = np.random.random((1, 412, 412, 3)) * 20 + 128.

for i in range(80):
    loss_value, grads_value = iterate([input_img_data])
    input_img_data += grads_value * step
```

For further reading, I recommend *Feature Visualization* and *The Building Blocks of Interpretability* on Distill.

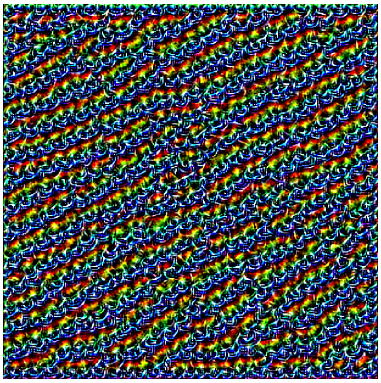


Figure 3.2 An image selected to maximally activate the first filter in the `block3conv3` layer in VGG16.

4 Training

05 March 2019

In this chapter we will take a closer look at some of the practical considerations around training neural networks: choosing hyperparameters, managing the vanishing/exploding gradients problem, various approaches to optimization, and various mechanisms for achieving regularization.

4.1 Vanishing/exploding gradients

This section follows the similarly titled section in Chapter 11 of *Géron*.

4.1.1 Initialization

In this section, we will discuss three strategies for achieving faster training: (1) Xavier-He initialization, (2) a new activation function called ELU, and (3) batch normalization.

One simple way to combat the vanishing gradient problem is to initialize the weights in a given layer by sampling each weight from a mean-zero Gaussian distribution with standard deviation

$$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}},$$

where n_{inputs} and n_{outputs} are the numbers of input and output connections for that layer. The idea is that if there are many neurons in the previous layer or the next layer, each weight should be smaller on average. This particular formula has been found to yield good results. This is called **Xavier-He initialization**.

4.1.2 Activation

Another idea for avoiding vanishing gradients is to use ReLU instead of one of its saturating counterparts, like sigmoid or tanh. However, ReLU contributes to the vanishing gradient problem in its own way, because training information doesn't backpropagate through a neuron with zero activation (in other words, when the value coming into the neuron is negative). Some variants of the ReLU have been introduced mitigate the dying neuron issue, including the **exponential linear unit** (ELU)

$$\text{ELU}_\alpha(u) = \begin{cases} \alpha(e^{-|u|} - 1) & \text{if } u < 0 \\ u & \text{if } u \geq 0 \end{cases}$$

ELU one of the activations available in Keras (`'elu'`).

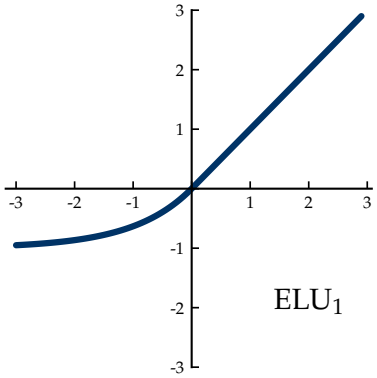


Figure 4.1 The exponential linear unit activation with $\alpha = 1$.

4.1.3 Batch normalization

We customarily standardize* features in our training and testing data so that the model training process is insensitive to arbitrary differences in units.† The idea of **batch normalization** is to grant neural networks the same convenience on a layer-by-layer basis. In other words, for each neuron in a given layer, we can standardize the activations coming out of that neuron and forward the results to the next layer.

Since it would be computationally expensive to standardize over the whole training set, we perform the standardization with reference to each mini-batch being used in the stochastic gradient descent update. Furthermore, rather than standardizing to zero mean and unit variance, we introduce two new trainable parameters per neuron—customarily denoted β and γ—which specify the mean and standard deviation of the activations forwarded to the next layer.

Mathematically, the batch normalization update takes the following form:

Batch normalization

```
μ̂ = mean(x)
σ̂² = std(x)
x_out = β + γ*(x-μ̂)/√(σ̂²+ε)
```

Here `x` is the vector of values coming into the neuron of interest, across a given mini-batch. For example, if we have 6 neurons and a mini-batch size of 100, then we will have 6 such vectors `x`, each of which is in $\mathbb{R}^{100}$. Also, `ε` is a *smoothing term* used to avoid division by zero in the event that `σ̂` is equal to zero. A typical choice is `ε = 1/10^5`.

Example 4.1.1

Show all of the intermediate calculations to perform forward propagation for a dense neural network with `A₁ = [1 2 3; 4 5 6]`, `b₁ = [2,3]`, `A₂ = [7 -2; 4 6]`, `b₂ = [-2,-1]`, with ReLU activation and batch normalization between the two layers. Suppose that the mini-batch input values are $[1,0,1]$, $[-3,0,-1]$, and $[1,2,1]$, and that the β values are -3 and 0 for the two neurons, and the gamma values are 1 and 2 for the two neurons. Try performing the normalization pre-activation and post-activation.

Solution

First some basic setup:*

```
A₁ = [1 2 3; 4 5 6]
b₁ = [2,3]
A₂ = [7 -2; 4 6]
b₂ = [-2,-1]
X = [1.0 -3 1; 0 0 2; 1 -1 1]
relu(x) = max(x,0)
```

Next, we calculate the activations, normalize them row-by-row, and forward the results to the second affine map.

* In other words, z-score: we subtract from each value the mean of that feature value over the available samples, and we divide the result by the standard deviation of the feature over the available samples.

† For example, the weight for a feature measured in centimeters would need to be 100 times larger than the weight for the same feature measured in meters.

* This is in Julia, simply because the syntax is less cluttered. You can think of it as pseudocode if you want,. or you can translate it to Python.

```
activations = relu.(A₁*X .+ b₁)
ε = 1e-5
β = [-3, 0]
γ = [1, 2]
for i=1:2
    activations[i,:] .-= mean(activations[i,:])
    activations[i,:] /= √(var(activations[i,:]) + ε)
    activations[i,:] = γ[i]*activations[i,:] .+ β[i]
end
A₂*activations .+ b₂
```

We get an output of `[-22.4197 -26.2553 -20.3251; -11.4297 -29.7249 2.15454]`.

The pre-activation version is left as an exercise.

`batch-norm-example.ipynb`

Keras has batch normalization available as a layer. Let's add batch normalization to our convnet example. We'll do two post-activation batch normalization layers and one pre-activation one.

```
import tensorflow.keras.activations as activations
import tensorflow.keras.models as models
import tensorflow.keras.layers as layers

import numpy as np

from tensorflow.keras.datasets import fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
train_images = train_images[:,:,:,np.newaxis]
test_images = test_images[:,:,:,np.newaxis]

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, use_bias=False))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.Dense(10, activation='softmax'))

model.compile(optimizer='rmsprop', loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
history = model.fit(train_images, train_labels, epochs = 5)

model.evaluate(test_images, test_labels)
```

The resulting accuracy is noticeably higher than without the batch normalization (91.1% vs. 87.7%). We will see how to combine batch normalization with other methods like early stopping and dropout to get even better results. Also, note that while the batch normalization layers make training slower *per epoch*, they also achieve a given accuracy threshold in fewer epochs.

4.1.3.1 Momentum

07 March 2019

The implementation of `layers.BatchNormalization` in Keras is actually slightly more complicated than the version presented above. If we estimate the mean and standard deviation using only the current batch, those estimates can be quite noisy since the batch size is typically fairly small. We can stabilize these estimates by taking into account the estimates for the previous several batches. There is a tradeoff to be navigated in choosing how many batches to consider, because the network's weight updates mean that the information from earlier batches is stale.*

* In other words, based on conditions which have changed.

One smooth way to handle this tradeoff is to take an **exponential moving average** of the mean and standard deviation estimates from previous batches. We fix a value α (which is 0.99 by default in Keras) called the **momentum**, and we average the mean and standard deviation estimates with weight α^k, where k is the batch index counting backwards. For example, if we estimated the mean activation at a particular neuron to be 0.6 in the first batch, 1.2 in the second batch, and 1.1 in the third batch, then our estimate of the mean for the third-batch normalization step would be

$$\frac{0.6\alpha^2 + 1.2\alpha^1 + 1.1\alpha^0}{\alpha^2 + \alpha^1 + \alpha^0} = 0.96867.$$

This average gives slightly more weight to the recent values than the straight average `mean([0.6, 1.2, 1.1])` $= 0.9\overline{6}$.

One of the reasons for the popularity of the exponential moving average is that it can be performed in a computationally inexpensive way: we do not actually have to record a running history of all of the previous estimates we want to average. We can just keep track of the most recent estimate:

Exercise 4.1.1

Show that the exponential moving average of $[x_1, \ldots, x_n]$ is equal to the $\left(\frac{1-\alpha}{1-\alpha^n}\right) x_n$ plus the product of $\frac{\alpha-\alpha^n}{1-\alpha^n}$ and the exponential moving average of $[x_1, \ldots, x_{n-1}]$.

4.2 Optimization

Let's review the stochastic gradient descent algorithm: we select a random mini-batch of training samples, forward propagate each sample through the network, backpropagate to find the derivative of the cost function with respect to each trainable parameter in the network, and decrement each weight by the learning rate η times the gradient for that weight. We then repeat with the next mini-batch using the updated weights.

This algorithm stands to benefit from the momentum concept we applied to batch normalization: since the mini-batches are small, the estimates we obtain for the weight updates are noisy. We can improve these estimates by taking into account the gradient estimates from previous batches. Since these estimates are based on different weights, we should discount their importance based on how stale they are. As we discussed in Section 4.1.3.1, we can keep track of this computation by tracking the last exponential moving average. We will also apply the simplification of replacing α^n with zero in the formula obtained in Exercise 4.1.1, since $\alpha^n \approx 0$ whenever n is large (as it is except at the beginning of the training process).

This leads to the following **momentum algorithm** for gradient descent. We store the current gradient esti-

* `\nabla` `\scrL` (Julia)

† The $1-\alpha$ factor is sometimes omitted.

mate using the variable `m`, and we let the momentum be `α`. Suppose that the derivative of the loss function with respect to a weight `w` is computed using backpropagation by the function* `∇ℒ`. Then each mini-batch update takes the form†

Gradient descent with momentum

```
m = α*m + (1-α)*∇ℒ(w) # exponential moving average
w -= η*m # update weight
```

There's a second advantage of having an estimate `m` of the gradient prior to estimating the gradient based on the current mini-batch: we can make the gradient computation forward-looking. In other words, we can increment the weights slightly in the direction of `-m` for purposes of estimating the gradient.

Nesterov accelerated gradient

```
m = α*m + (1-α)*∇ℒ(w-α/(1-α)*η*m) # estimate gradient slightly ahead of current location
w -= η*m # update weight
```

Momentum yields substantial speed gains, and Nesterov accelerated gradient is faster still.

The **AdaGrad** algorithm accelerates convergence by addressing the problem of **pathological curvature**. Consider a function of two variables whose graph has a long narrow valley. For example, $f(x,y) = \frac{x^4}{100} + 100y^2$ has a minimum at the origin, and its graph is shaped like a steep-walled valley which runs along the x-axis. Gradient descent is slow for this function unless the starting point happens to be on the y-axis, because the algorithm will proceed quickly to the narrow valley and make its way slowly from there to the origin.

To address this problem, we will scale the steps in each direction based on the accumulated squared of gradients computed so far. Each update takes the following form.

AdaGrad

```
s += ∇ℒ(w).^2
w -= η*∇ℒ(w)./√(s+ε)
```

Once again `ε` is a smoothing term to avoid division by zero.

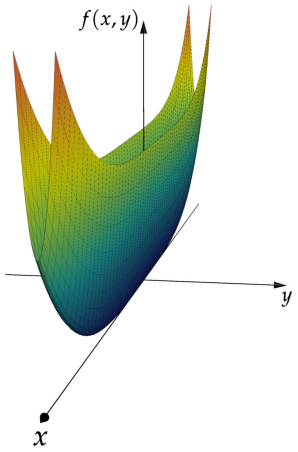


Figure 4.2 The graph of a function exhibiting pathological curvature.

12 March 2019

This algorithm is different from vanilla gradient descent in two ways: (1) the learning rate is different for different components, and (2) the learning rate changes over time. Specifically, the learning rate is smaller for components which have had larger updates, and the learning rate decreases as the number of iterations increases.

AdaGrad tends to decrease the learning rate too quickly, causing the algorithm to stall out. **RMSProp** addresses this problem by replacing `s`'s accumulating behavior with an exponential moving average.

RMSProp

```
s = α*s + (1-α)*∇ℒ(w).^2
w -= η*∇ℒ(w)./√(s+ε)
```

The innovations of momentum gradient descent and RMSProp address different convergence issues, which suggests that they can be combined to get an algorithm with the benefits of both. This is indeed the case. With the values of the variables* $\alpha_1{}^t$ and $\alpha_2{}^t$ initialized to 1, and perform the following updates on each iteration:

* That's three Unicode characters: `\alpha`, `\_1`, and `\^t`

Adam

```
# exponential moving averages for gradient and squared gradient
m = α₁*m + (1-α₁)*∇ℒ(w)
s = α₂*s + (1-α₂)*∇ℒ(w).^2

# give m and s a little boost early in training:
α₁ᵗ *= α₁
α₂ᵗ *= α₂
m̂ = m/(1-α₁ᵗ)
ŝ = s/(1-α₂ᵗ)

# apply AdaGrad scaling to the momentum gradient estimate
w -= η*m̂./√(ŝ+ε)
```

Let's summarize:

Summary of optimization algorithms

Algorithm	*Main idea*
Gradient descent	step in the direction of maximum decrease
Momentum	use exponential moving average of current and previous gradient estimates
Nesterov accelerated gradient	use momentum and look ahead to compute gradients
AdaGrad	adjust learning rate componentwise according to accumulated squared gradient
RMSProp	replace squared gradient *accumulation* with *exponential moving average* in AdaGrad
Adam	combine momentum and RMSProp

Example 4.2.1

Implement each of the above algorithms in NumPy, using the function $f(x, y) = x^4/100 + 100y^2$ and the starting point $(1, 1)$.

optimization-methods.ipynb

Solution

We begin with some setup:

```
import numpy as np
import matplotlib.pyplot as plt

def f(v):
    x,y = v
    return pow(x,4)/100 + 100*pow(y,2)

def gradf(v):
    x,y = v
    return np.array([4*np.pow(x,3)/100, 200*y])
```

We make the optional parameters keyword arguments and keep track of the history of iterates so we can plot it.

```
def GradDescent(gradf, v0, n = 100, η = 0.001):
    v = [np.array(v0)]
    for _ in range(n):
        v.append(v[-1] - η*gradf(v[-1]))
    return v
```

For gradient descent with momentum, we include the `m` vector. Also, we can combine this function with Nesterov Accelerated Gradient.

```
def Momentum(gradf, v0, n = 100, η = 0.001, α = 0.9, nest_acc = False):
    v = [np.array(v0)]
    m = np.array([0,0])
    for _ in range(n):
        m = α*m + (1-α)*gradf(v[-1] - nest_acc*η*α/(1-α)*m)
        v.append(v[-1] - η*m)
    return v
```

For AdaGrad, we keep track of `s` rather than `m`.

```
def AdaGrad(gradf, v0, n = 100, η = 0.001, α = 0.9):
    ε = 1e-5
    v = [np.array(v0)]
    s = 0
    for _ in range(n):
        s += pow(gradf(v[-1]),2)
        v.append(v[-1] - η*gradf(v[-1])/np.sqrt(s+ε))
    return v
```

We can change the line that updates `s` to get RMSProp.

```
def RMSProp(gradf, v0, n = 100, η = 0.001, α = 0.9):
    ε = 1e-5
    v = [np.array(v0)]
    s = 0
    for _ in range(n):
        s = α*s + (1-α)*pow(gradf(v[-1]),2)
        v.append(v[-1] - η*gradf(v[-1])/np.sqrt(s+ε))
    return v
```

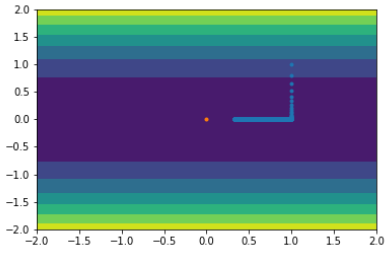
Finally, Adam combines momentum and RMSProp. For convenience, let's implement this algorithm for the special case $\alpha_1 = \alpha_2$.

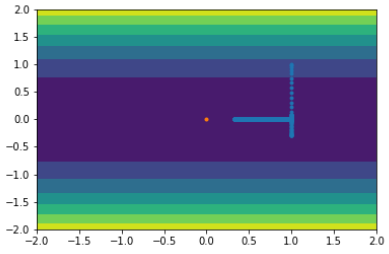
```
def Adam(gradf, v0, n = 100, η = 0.001, α = 0.9):
    ε = 1e-5
    v = [np.array(v0)]
    s = 0
    m = np.array([0,0])
    αt = α
    for _ in range(n):
        m = α*m + (1-α)*gradf(v[-1])
        s = α*s + (1-α)*pow(gradf(v[-1]),2)
        αt *= α
        m̂ = m/(1-αt)
        ŝ = s/(1-αt)
        v.append(v[-1] - η*m̂/np.sqrt(ŝ+ε))
    return v
```

We can visualize how each of these algorithms works by plotting the sequence of iterates.

```
def plot_iterates(v):
    X = Y = np.linspace(-2,2,100)
    Z = np.array([[f([x,y]) for x in X] for y in Y])
    plt.contourf(X,Y,Z)
    plt.plot(np.vstack(v)[:,0],np.vstack(v)[:,1],'.')
    plt.plot([0],[0],'.')
    plt.xlim(-2,2)
    plt.ylim(-2,2)
    plt.show()

plot_iterates(GradDescent(gradf, [1,1], η = 1e-3, n = 10000, α = 0.9))
```





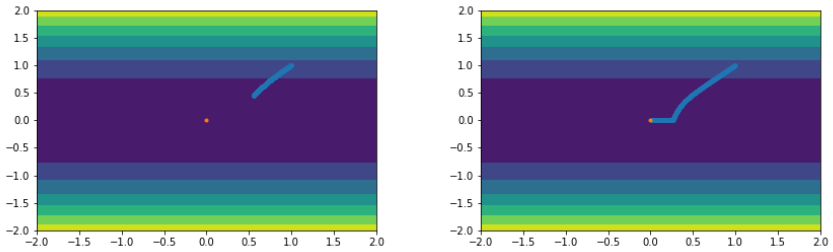


Figure 4.3 Contour lines and gradient descent iterates for the function $f(x,y) = x^4/100 + 100y^2$ starting from $(1,1)$. The figures show AdaGrad, Momentum, RMSProp, and plain gradient descent, in some order. Which is which?

4.3 Regularization

4.3.1 Early stopping

Early stopping is the idea of holding out a portion of the training data—called **validation** data—and halting the training process once the loss or accuracy stops improving. For example, we might stop when we reach a fixed number of steps after the last time a record minimum loss was reached (and then revert to the last record minimum).

* *Callback* is a general programming term which refers to executable code which is passed to another piece of code for execution at a particular time

Early stopping is implemented in Keras as a **callback*** , which is a set of functions to be applied at a specific step in the training process. Callbacks are inserted as a list using the `callbacks` keyword argument of the `fit` method.

```
callbacks = [keras.callbacks.EarlyStopping(monitor = "val_loss",
                                           patience = 10,
                                           mode = 'max')]
...
model.fit(x_train, y_train, callbacks = callbacks)
```

The `monitor` keyword argument is for indicating whether to track loss or accuracy (`val_acc`), `patience` is the number of epochs after the last record optimum when the training should halt, and `mode` indicates whether record minima or record maxima should be tracked.

4.3.2 Weight regularization

We can apply regularization by adding a term to penalize large weights in the objective function. If the penalty is in proportion to the absolute value of each weight, we call this ℓ^1 regularization (or ℓ_1 or L_1 or L^1). If the penalty is in proportion to the square root of the sum of the squares of the weights, we call it ℓ^2 regularization. This terminology is borrowed from math, where the ℓ^p norm of a vector $[x_1, \ldots, x_n]$ is defined to be

$$\|\mathbf{x}\|_{\ell_p} = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{1/p}$$

In Keras, weight regularization is achieved using the `kernel_regularizer` keyword argument to a `Dense` or `Conv2D` layer.

```
layers.Dense(kernel_regularizer=keras.regularizers.l2(0.01))
```

The argument of `l2` is the multiple of the ℓ^2 norm being added to the penalty. Reasonable values range from 0 to 0.1.

4.3.3 Dropout

Applying dropout to a layer makes a random subset of its neurons inactive for a each epoch. Each neuron in a dropout layer is inactivated with a specified probability α, independently of other units and independently of the same neuron's status in other epochs. Dropout reduces overfitting by preventing the neural network from relying too heavily on specific pathways or individual neurons, and it's remarkably effective.

Dropout is implemented in Keras using a layer, which applies Dropout to the *preceding* layer.

```
model.add(keras.layers.Dense(64))
model.add(keras.layers.Dropout(0.2))
```

4.3.4 An example

Let's put several regularization techniques together into a single convnet for Fashion MNIST. The architecture we will use is based on the following Kaggle submission:

https://www.kaggle.com/kentaroyoshioka47/cnn-with-batchnormalization-in-keras-94

fashion-mnist-bn-dropout.ipynb

```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, BatchNormalization, Activation, \
                                    MaxPooling2D, Flatten, Dropout, Dense
from sklearn.model_selection import train_test_split

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
train_images, val_images, train_labels, val_labels = train_test_split(
    train_images, train_labels, test_size = 0.2, random_state = 42)

train_images = train_images[:,:,:,None]/255.0
val_images = val_images[:,:,:,None]/255.0
test_images = test_images[:,:,:,None]/255.0
```

We begin by defining an early stopping callback.

```
callbacks = [EarlyStopping(monitor='val_acc',
                           patience=10,
                           mode='max',
                           verbose=1)]
```

Now we can set up the batch with a batch normalization and dropout after the first convolutional layer.

```
model = Sequential()
model.add(Conv2D(64, kernel_size=(3,3), padding='same',
                 activation='relu',
                 input_shape=(28,28,1)))
model.add(BatchNormalization())
model.add(Dropout(0.25))
```

A couple more convolutional layers:

```
model.add(Conv2D(64, kernel_size=(3,3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D())
model.add(Dropout(0.25))

model.add(Conv2D(64, kernel_size=(3,3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D())
model.add(Dropout(0.25))
```

And finally two fully connected layers:

```
model.add(Flatten())
model.add(Dense(500,use_bias=False))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.25))

model.add(Dense(10))
model.add(Activation('softmax'))
```

* We get 93.5% accuracy, up from the 91.1% accuracy we achieved with just batch normalization.

Finally, we compile our model, train it with the callback, and evaluate it on the test data.*

```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
model.fit(train_images, train_labels,
          batch_size=128,
          epochs=100,
          validation_data=(val_images, val_labels),
          shuffle=True,
          callbacks=callbacks)
model.evaluate(test_images, test_labels)
```

5 Neural networks and computation

19 March 2019

In this chapter we will (1) discuss how to approach neural network modifications that are not directly supported by Keras, and (2) look at some of the hardware advances that have supported the development of deep learning over the last decade or so. We will discuss two types of tools: extension of Keras classes and direct use of low-level TensorFlow functionality.

5.1 Extending Keras classes

5.1.1 Python classes: an example

Let's briefly review how classes work in Python. Suppose we want to be able to work directly with fractions in Python. We can define a new class called `Rational` which stores the data (numerator and denominator) as well as the associated methods (such as addition and multiplication), as follows:

python-classes-rational-numbers.ipynb

```
class Rational(object):
    def __init__(self, num, denom):
        self.num = num
        self.denom = denom

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def add(self, other):
        return Rational(self.num * other.denom +
                        self.denom * other.num,
                        self.denom * other.denom)

    def multiply(self, other):
        return Rational(self.num * other.num,
                        self.denom * other.denom)
```

The special **constructor method** `__init__` runs whenever a new object of this class is **instantiated** with a call like `Rational(3,4)`. We store the given numerator and denominator values as **attributes** of the newly created object. The special* `__str__` method is used to print a `Rational` object.

* Methods surrounded by double underscores are called *dunder* methods.

The `Rational` methods like `add` and `multiply` are called using dot syntax on a `Rational` object. Here's an example:

```
a = Rational(3,4)
b = a.add(Rational(1,2))
b.multiply(a)
```

Whenever a method is called, the object named before the dot is implicitly supplied as the first argument (customarily called `self` in the method definition). For example, in the expression `b.multiply(a)`, the value of `b` is bound to `self` and the value of `a` is bound to `other` in the body of the `multiply` method.

Our Rational class has one drawback: we aren't bothering to reduce the fraction. It would be nice to have `Rational(2,4)` print as `1/2`. We could introduce this functionality by modifying `Rational`, but let's do it by introducing another class so we can use either class depending on whether we want our fractions reduced.

```
import math
class ReducedRational(object):
    def __init__(self, num, denom):
        d = math.gcd(num, denom)
        reduced_num, reduced_denom = num // d, denom // d
        self.num = num
        self.denom = denom

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def add(self, other):
        return ReducedRational(self.num * other.denom +
                               self.denom * other.num,
                               self.denom * other.denom)

    def multiply(self, other):
        return ReducedRational(self.num * other.num,
                               self.denom * other.denom)
```

Clearly there's a lot of redundancy here. The methods work the same way for the two classes; the only real difference is in the constructor method. Finding a good way to eliminate this redundancy is *essential* for building maintainable systems, because making changes in a codebase with substantial copy-pasting is very difficult to do consistently.

Python's way of eliminating redundancy between classes is called **inheritance**. We will make `ReducedRational` *inherit* from `Rational`, which has the effect that methods called on a `ReducedRational` object are looked up in `Rational` if they are not defined in the `ReducedRational` class. That way we can refrain from having to re-define methods that are essentially the same. We will have to solve one small problem, though: the calls to `Rational` in `add` and `multiply` need to be replaced with `ReducedRational`. We can accomplish this by referring to both as `self.__class__` (which will be `Rational` if `self` is a `Rational` and `ReducedRational` if `self` is a `ReducedRational`).

Putting it all together, we arrive at the final version:

```
import math
class Rational(object):
    def __init__(self, num, denom):
        self.num = num
        self.denom = denom

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def add(self, other):
```

```
        return self.__class__(self.num * other.denom +
                              self.denom * other.num,
                              self.denom * other.denom)

    def multiply(self, other):
        return self.__class__(self.num * other.num,
                              self.denom * other.denom)

class ReducedRational(Rational):
    def __init__(self, num, denom):
        d = math.gcd(num, denom)
        reduced_num, reduced_denom = num // d, denom // d
        self.num = num
        self.denom = denom
```

5.1.2 Keras subclassing API

The objects we've been working with in Keras (**Layer**s, **Object**s, etc.) are Python classes, and many of them are subclasses of others. For example, **Dense** and **Flatten** both inherit from **Layer**. You can define your own layers by defining a method which inherits **Layer**. Let's take a look at the example provided in the TensorFlow documentation*. This layer just multiplies by a matrix of weights.

* link here

keras-subclassing.ipynb

```
import tensorflow as tf
import tensorflow.keras.layers as layers

class MyLayer(layers.Layer):
    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(MyLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        shape = tf.TensorShape((input_shape[1], self.output_dim))
        # Create a trainable weight variable for this layer.
        self.kernel = self.add_weight(name='kernel',
                                      shape=shape,
                                      initializer='uniform',
                                      trainable=True)
        # Make sure to call the `build` method at the end
        super(MyLayer, self).build(input_shape)

    def call(self, inputs):
        return inputs @ self.kernel

    def compute_output_shape(self, input_shape):
        shape = tf.TensorShape(input_shape).as_list()
        shape[-1] = self.output_dim
        return tf.TensorShape(shape)

    def get_config(self):
        base_config = super(MyLayer, self).get_config()
        base_config['output_dim'] = self.output_dim
        return base_config
```

Let's work through the details. First, writing `**kwargs` in an argument list for a method definition wraps any keyword arguments supplied when the method is called and stores the result in a dictionary called **kwargs**. Writing `**kwargs` when *calling* a function unpacks the dictionary **kwargs** and supplies the key-value pairs as keyword arguments.

The second unusual thing we see is `super(MyLayer, self)`. This is the Python idiom for calling a method in a superclass which has been overridden in the current class. This is important here because we want the general **Layer** class to be able to do its own initialization steps (including processing any keyword arguments we might supply).

Keras requires that the **build** method be implemented by a class which inherits from **Layer**. Any weights the layer might have should be registered with a call to **add_weight** (which is inherited). Using this method ensures that Keras can properly track the trainable parameters, and it also provides access to convenient Keras tools like the built-in initializers.

The **call** method specifies how the layer maps its input to its output. The final two methods specify the shape of the tensor output by the layer and modify the layer's **config** dictionary, respectively.

At this point, we can incorporate our layer into a **Sequential** model:

```
import numpy as np
model = tf.keras.models.Sequential([MyLayer(10)])
model.predict(np.random.randn(40,40))
```

Likewise, we can define our own models by subclassing **Model**:

```
from tensorflow.keras.models import Model
class MyModel(Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv1 = Conv2D(32, 3, activation='relu')
        self.flatten = Flatten()
        self.d1 = Dense(128, activation='relu')
        self.d2 = Dense(10, activation='softmax')

    def call(self, x):
        x = self.conv1(x)
        x = self.flatten(x)
        x = self.d1(x)
        return self.d2(x)

model = MyModel()
```

5.2 TensorFlow

The core functionality of TensorFlow is to support the definition of a **computational graph**, which defines a calculation to be performed on a set of tensors, and then distribute that computation in an efficient way to the available computing resources.

In TensorFlow 1.0, the process of constructing a computational graph had to be carried out using *placeholders* (to represent the unknown inputs) and TensorFlow-specific versions of the various arithmetic operations involved in the calculation. The actual computation is performed by supplying the input values to the `run` method of a TensorFlow `Session` object. It's recommended to use a Python `with` statement for the session, because this ensures that the session object is appropriately torn down after the computation is performed.*

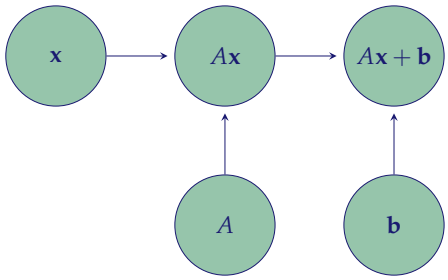


Figure 5.1 A computational graph for evaluating the expression $A\mathbf{x} + \mathbf{b}$. The nodes represent tensors (or *data*), and the arrows indicate how the data flows through the graph. The operations (multiplication and addition, in this example) are also part of the computational graph structure.

* One technicality: the variables have to be initialized within the session.

`tf1.ipynb`

```
import tensorflow as tf
import numpy as np
x = tf.placeholder(tf.float32,shape=(20,1))
A = tf.get_variable(initializer=tf.ones(shape=(20,20)), name='A')
b = tf.get_variable(initializer=tf.ones(shape=(20,1)), name='b')
output = tf.add(tf.matmul(A,x), b)
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    y = sess.run(output, {x: np.ones((20,1))})
```

TensorFlow 2.0* makes this process easier. Since a session is essentially the same idea as a function (with the placeholders playing the role of input values, and the result from `Session.run` the output value), a decorator† `tf.function` was introduced to achieve the same effect using standard Python syntax.

* ...which we have been running and which was recently released in alpha.

`tf2.ipynb`

† A *decorator* in Python is a function which takes a function as input and returns a function as output. It can be used to modify a function with "pie syntax" (e.g., `@tf.function`).

```
!pip install tensorflow==2.0.0-alpha0
import tensorflow as tf
import numpy as np
A = tf.Variable(tf.ones(shape=(20,20)), name='A')
b = tf.Variable(tf.ones(shape=(20,1)), name='b')

@tf.function
def L(x):
    return A @ x + b

y = L(np.ones((20,1),dtype=np.float32))
```

Another major change from TensorFlow 1.x to 2.0 is that Keras is now the recommended interface for building neural networks. Some of the functionality for building neural networks in straight TensorFlow has been deprecated in favor of corresponding versions in Keras.

5.3 Hardware acceleration

This section draws substantially from Section 3.7 in *Aggarwal*.

5.3.1 Processors

The **CPU** (central processing unit) is the primary component in a computer that executes program instructions. In the 1970s, arcade game manufacturers recognized that the general-purpose design of the CPU made it suboptimally suited to the matrix operations used heavily in graphics rendering, so specialized chips called **GPU**s (graphics processing units) were introduced.

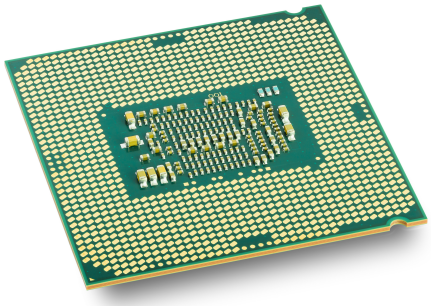


Figure 5.2 A CPU.

GPUs generally have a slower clock speed than CPUs, but they can parallelize computations more efficiently. Matrix operations are ideally suited to parallelization-oriented optimization, since different entries in a matrix sum or product may be computed independently. Since neural network calculations are heavily reliant on matrix operations, using GPUs is a natural way to accelerate them.

* NVIDIA has about 70% and AMD has 30% of the GPU market share, as of late 2018.

In 2007, the GPU manufacturer NVIDIA* introduced an interface called **CUDA** for performing calculations on GPUs without having to heavily re-write code written for a CPU. Today, users take advantage of GPU acceleration without rewriting their code at all, because TensorFlow and other deep learning frameworks use CUDA under the hood to automatically utilize GPUs when they are available.

* in 2016

Unsatisfied with using hardware originally designed for graphics, Google* introduced an even more specialized chip called a **TPU** (tensor processing unit) specifically built for doing deep learning in TensorFlow. One of the main ways in which TPUs are adapted to deep learning is their use of very low-precision floating point (like `Float8`'s!).

5.3.2 Parallelization

The speed-up we get just from using GPUs is limited to the parallelization benefits at the level individual matrix operations. It can also be helpful or necessary to parallelize in other ways to take advantage of multiple CPUs and/or GPUs:

1. **Hyperparameter parallelization**. We can try a variety of choices of model hyperparameters simultaneously. This is an example of an *embarrassingly parallel* problem: the different training runs are not interrelated, so we can just deploy them to different machines and collect the results.
2. **Model parallelization**. We can split each layer into multiple pieces; for example, in a convolutional neural net we could put half the channels in a given layer on one GPU and half on another. This problem is not embarrassingly parallel, since the values from both sets of channels need to be communicated to the next layer. These data transfer costs trade off against the parallelization benefits.
3. **Data parallelization**. We can train our model on separate mini-batches simultaneously. This entails a change to the training algorithm, since typically each mini-batch needs to computed with weights that were updated based on the previous mini-batch. The idea is to store the weights in a centralized server and have each parallel implementation of the model read and write to the server whenever a mini-batch is finished.*

* J. Dean *et al*. Large scale distributed deep networks. NIPS Conference, 2012.

5.3.3 Model compression

In many applications, the resources available at training time vastly exceed the resources available when predictions are needed. Consider, for example, incorporating deep learning into an iOS or Android app. Thus various strategies for **model compression** have been developed. We will discuss just one: **mimic models***. The idea is to train a large model on your training data and then train a much smaller model from scratch on the *softmax outputs* from the larger model. Remarkably, this smaller model typically performs much better than one with the same architecture trained on the original data, and often almost as well as the larger model.

* J. Ba and R. Caruana. Do deep nets really need to be deep? NIPS Conference, pp. 2654–2662, 2014.

6 Modern convolutional architectures

04 April 2019

This material is presented in website format here.

7 Natural language processing

7.1 word2vec

Working with language requires mapping language elements (like words) to vectors. *word2vec* is a simple and remarkably effective technique for doing that in a way that encodes some aspects of language meaning. See here for details.

7.2 LSTMs

Long short term memory cells (LSTMs) are a more advanced and more effective type of recurrent cell than the SimpleRNN cell we learned previously. Please read Chris Olah's blog post for details about how these cells work.

7.3 Predictive text: an example

Check out this blog post for a bare-hands example of how one can use LSTMs to build a recurrent neural network for a familiar task: *predictive text*. The upshot is that even a rather expensively trained (character-by-character) LSTM yields pretty disappointing results.

7.4 Modern NLP techniques

Deep learning NLP has experienced some transformative changes in the last two years, beginning with the paper *Attention is all you need* by Vaswani et al. The ideas introduced in that paper led to the introduction of Google's BERT and OpenAI's GPT-2. BERT provides a transfer learning base that yields state-of-the-art performance on a variety of natural language tasks, and GPT-2 has produced some very impressive results (although only a smaller version of the model is publicly available currently). Notably, these models *do not use convolution or recurrent cells*. Rather, they use a neural network structure called a **transformer**.

The ideas you need to understand transformers are helpfully animated and explained in the following series of blog posts by Jay Allamar:

1. Attention.
2. Transformers
3. BERT

You can train GPT-2 on your own dataset by opening this repo in Colab.

8 Reinforcement learning

See bit.ly/brown-reinforcement-learning for the reinforcement learning chapter.