10 Fundamental deep learning algorithms

This chapter covers

- Multilayer perceptron
- Convolutional neural nets: LeNet on MNIST and ResNet image search
- Recurrent neural nets: LSTM sequence classification and multi-input neural net
- Neural network optimizers

In the previous chapter, we discussed selected unsupervised ML algorithms to help discover patterns in our data. In this chapter, we introduce deep learning algorithms. Deep learning algorithms are part of supervised learning, which we encountered in chapters 5, 6, and 7. Deep learning algorithms revolutionized the industry and enabled many research and business applications previously thought to be out of reach by classic ML algorithms. We'll begin this chapter with the fundamentals, such as multilayer perceptron (MLP) and LeNet convolutional model for MNIST digit classification. We will follow these topics with more advanced applications, such as image search based on the ResNet50 convolutional neural network (CNN). We will delve into recurrent neural networks (RNNs) applied to sequence classification using long short-term memory (LSTM) and implement, from scratch,

a multi-input model for sequence similarity. We'll then discuss different optimization algorithms used for training neural networks and conduct a comparative study. We will be using the Keras/TensorFlow deep learning library throughout this chapter.

10.1 Multilayer perceptron

Multilayer perceptron (MLP) is commonly used for classification and regression prediction tasks. An MLP consists of multiple densely connected layers that perform the following linear (affine) operation.

$$f(x;\theta) = Wx + b \tag{10.1}$$

Here, x is the input vector, W is the weight matrix, and b is the bias term. While the linearity of the operation makes it easy to compute, it is limiting when it comes to stacking multiple layers. Introducing nonlinearity between layers via activation functions enables the model to have greater expressive power. One common nonlinearity is ReLU, defined as follows.

$$\text{ReLU}(x) = \max(0, x) \tag{10.2}$$

We can define MLP with L layers as a composition of $f_l(x;\theta) = \text{ReLU}(W_{lx} + b_l)$ functions as follows.

$$\text{MLP}(x;\theta) = f_L(f_{L-1}(\cdots(f_1(x;\theta_1))\cdots)) \tag{10.3}$$

The activation of the last layer function f_L will depend on the task at hand: softmax for classification or identity for regression. The softmax activation computes class probabilities that sum to 1, based on real-valued input. It is defined as follows.

$$\text{softmax}(x) = \frac{e^x}{\sum_{i=1}^{K} e^{x_i}} \tag{10.4}$$

Here, K is the number of classes. Figure 10.1 shows an MLP architecture.

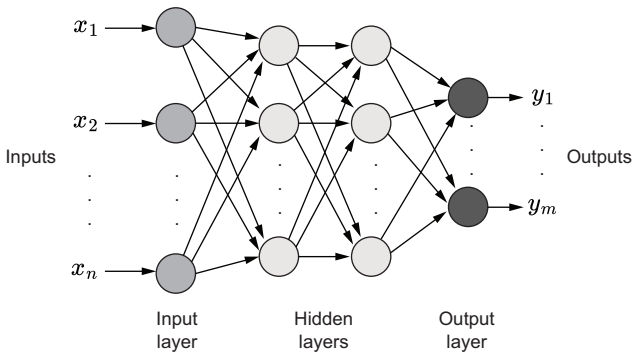

Figure 10.1 MLP architecture

To train the neural network, we need to introduce the loss function. A *loss function* tells us how far away the network output is from expected ground truth labels. The loss depends on the task we are optimizing for, and in the case of our MLP, it can be, for example, cross-entropy $H(p, q)$ loss for classification or mean squared error (MSE) loss for regression.

$$\begin{aligned} L_H(y, \hat{y}) &= H(p, q) = -\sum_{x \in X} p(x) \log q(x) = -\sum_{k=1}^{K} y_k \log \hat{y}_k \\ L_{MSE}(y, \hat{y}) &= \text{MSE}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \end{aligned} \tag{10.5}$$

Here, y is the true label and $p(x)$ is the true distribution; $\hat{y}$ is the predicted label and $q(x)$ is the predicted distribution.

MLPs are known as feed-forward networks because they have a direct computational graph from input to output. A neural network training consists of a forward and backward pass. During inference (forward pass), each layer produces a forward message $z = f(x; \theta)$ as the output (given the current weights of the network) followed by computation of the loss function. During the backward pass, each layer takes a backward message dL/dz of the next layer and produces two backward messages at the output dL/dx gradient wrt to the previous layer x and $dL/d\theta$ gradient wrt to parameters of the current layer.

This backpropagation algorithm is based on the chain rule and can be summarized as follows.

$$\begin{aligned} \frac{\partial L}{\partial x_i} &= \sum_j \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial x_i} \\ \frac{\partial L}{\partial \theta_i} &= \sum_j \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial \theta_i} \end{aligned} \tag{10.6}$$

Once we know the gradient with respect to parameters for each layer, we can update the layer parameters as follows.

$$\theta_i^t = \theta_i^{t-1} - \lambda \frac{\partial L}{\partial \theta_i} \tag{10.7}$$

Here, λ is the learning rate. Note that the gradients are computed automatically by deep learning frameworks, such as TensorFlow/PyTorch.

During training, we may want to adopt a learning rate schedule to avoid local optima and converge on a solution. We typically start with the learning rate of some constant and reduce it according to a schedule over epochs, where one epoch is a single pass through the training dataset. Throughout this chapter, we'll be using an exponential learning rate schedule. Other alternatives include a piecewise linear schedule with successive halving, cosine annealing schedule, and one-cycle schedule.

Before we get to the implementation of MLP, it's advantageous to talk about model capacity, to avoid underfitting, and regularization, to prevent overfitting. Regularization can occur at multiple levels, such as weight decay, early stopping, and dropout. At a high level, we would like to increase model capacity by changing the architecture (e.g., increasing the number of layers and the number of hidden units per layer) if we find that the model is underfitting or not achieving high validation accuracy. On the other hand, to avoid overfitting, we can introduce weight decay or l2 regularization applied to nonbias weights (W but not b) of each layer. Weight decay encourages smaller weights and, therefore, simpler models. Another form of regularization is stopping the training early when we notice validation loss is starting to increase away from the training loss. This early stopping criterion can be used to save time and computational resources, while saving a checkpoint of the model. Finally, dropout is an effective regularization technique, where the dense connections are dropped or zeroed at random according to a fixed rate, as shown in figure 10.2. Dropout enables better generalization (see Nitish Srivastava et al.'s "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," JMLR 2014).

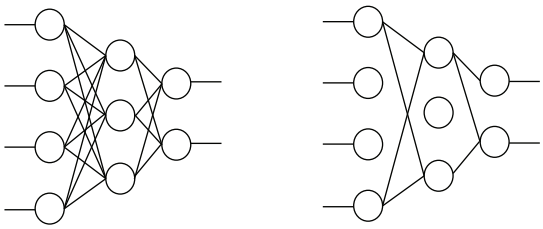

Figure 10.2 Dropout applied to multilayer perceptron at one training epoch.

Let's combine the principles we learned so far in our first implementation of the multilayer perceptron (MLP) in Keras/TensorFlow! For more information regarding the Keras library, readers are encouraged to visit https://keras.io/ or review François Chollet's *Deep Learning with Python*, Manning, 2021.

Listing 10.1 Multilayer perceptron

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
```

```
from keras.models import Sequential
from keras.layers import Dense, Dropout

from keras.callbacks import ModelCheckpoint
from keras.callbacks import TensorBoard          <-- For visualizing metrics
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping

import math
import matplotlib.pyplot as plt

tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"

def scheduler(epoch, lr):        <-- Learning rate schedule
    if epoch < 4:
        return lr
    else:
        return lr * tf.math.exp(-0.1)

if __name__ == "__main__":

    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
    x_train = x_train.reshape(x_train.shape[0], x_train.shape[1] *
    ➥ x_train.shape[2]).astype("float32") / 255
    x_test = x_test.reshape(x_test.shape[0], x_test.shape[1] *
    ➥ x_test.shape[2]).astype("float32") / 255

    y_train_label = keras.utils.to_categorical(y_train)
    y_test_label = keras.utils.to_categorical(y_test)
    num_classes = y_train_label.shape[1]

    batch_size = 64
    num_epochs = 16          <-- Training params

    model = Sequential()
    model.add(Dense(128, input_shape=(784, ), activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(10, activation='softmax'))

    model.compile(
        loss=keras.losses.CategoricalCrossentropy(),
        optimizer=tf.keras.optimizers.RMSprop(),
        metrics=["accuracy"]
    )

    model.summary()

    #define callbacks
    file_name = SAVE_PATH + 'mlp-weights-checkpoint.h5'
    checkpoint = ModelCheckpoint(file_name, monitor='val_loss', verbose=1,
    ➥ save_best_only=True, mode='min')
    reduce_lr = LearningRateScheduler(scheduler, verbose=1)
```

```
early_stopping = EarlyStopping(monitor='val_loss', min_delta=0.01,
➥ patience=16, verbose=1)
#tensor_board = TensorBoard(log_dir='./logs', write_graph=True)
callbacks_list = [checkpoint, reduce_lr, early_stopping]

hist = model.fit(x_train, y_train_label,
➥ batch_size=batch_size, epochs=num_epochs,
➥ callbacks=callbacks_list, validation_split=0.2)   <-- Model training

test_scores = model.evaluate(x_test, y_test_label,
➥ verbose=2)                                        <-- Model evaluation
print("Test loss:", test_scores[0])
print("Test accuracy:", test_scores[1])

plt.figure()
plt.plot(hist.history['loss'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_loss'], '--r', lw=2.0, label='val')
plt.title('MLP model')
plt.xlabel('Epochs')
plt.ylabel('Cross-Entropy Loss')
plt.legend(loc='upper right')
plt.show()

plt.figure()
plt.plot(hist.history['accuracy'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_accuracy'], '--r', lw=2.0, label='val')
plt.title('MLP model')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')
plt.show()

plt.figure()
plt.plot(hist.history['lr'], lw=2.0, label='learning rate')
plt.title('MLP model')
plt.xlabel('Epochs')
plt.ylabel('Learning Rate')
plt.legend()
plt.show()
```

Figure 10.3 shows the cross-entropy loss and accuracy for both training and validation datasets. The MLP model achieves 98% accuracy on the test dataset after only 16 training epochs. Notice how the training loss drops below the validation and, similarly, the accuracy on the training set is higher than the validation set. Since we are interested in how our model generalizes to unseen data, the validation set is the real indicator of performance. In the next section, we will examine a class of neural networks better suited for processing image data and, therefore, widely used in computer vision.

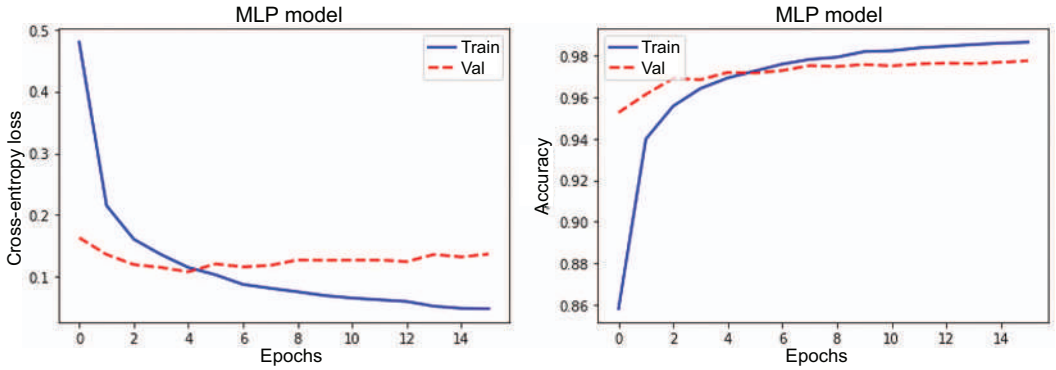

Figure 10.3 MLP cross-entropy loss and accuracy for training and validation datasets

10.2 Convolutional neural nets

Convolutional neural networks (CNNs) use convolution and pooling operations in place of vectorized matrix multiplications. In this section, we'll motivate the use of these operations and describe two architectures: LeNet, for MNIST digit classification, and ResNet50, applied to image search.

CNNs work exceptionally well for image data. Images are high dimensional objects of size $W \times H \times C$, where W is the width, H is the height, and C is the number of channels (e.g., $C=3$ for RGB image and $C=1$ for grayscale). CNNs consist of trainable filters or kernels that get convolved with the input to produce a feature map. This results in a vast reduction in the number of parameters and ensures invariance to translations of the input (since we would like to classify an object at any location of the image). The convolution operation is 2D in the case of images, 1D in the case of time series, and 3D in the case of volumetric data. A 2D convolution is defined as follows.

$$(f * g)(i, j) = \sum_a \sum_b f(a, b) g(i - a, i - b) \tag{10.8}$$

In other words, we slide the kernel across every possible position of the input matrix. At each position, we perform a dot-product operation and compute a scalar value. Finally, we gather these scalar values in a feature map output.

Convolutional layers can be stacked. Since convolutions are linear operators, we include nonlinear activation functions in between, just as we did in fully connected layers.

Many popular CNN architectures include a pooling layer. Pooling layers down-sample the feature maps by aggregating information. For example, max pooling computes a maximum over incoming input values, while average pooling replaces the max operation with the average. Similarly, global average pooling could be used to reduce a $W \times H \times D$ feature map into a $1 \times 1 \times D$ aggregate, which can then be reshaped to a D-dimensional vector for further processing. Let's look at a few applications of CNNs.

10.2.1 LeNet on MNIST

In this section, we will study the classic LeNet architecture (Yann LeCun et al.'s "Gradient-Based Learning Applied to Document Recognition," Proceedings of the IEEE, 1998) developed by Yann LeCun for handwritten digit classification trained on the MNIST dataset. It follows the design pattern of a convolutional layer, followed by ReLU activation, followed by max-pooling operation, as shown in figure 10.4.

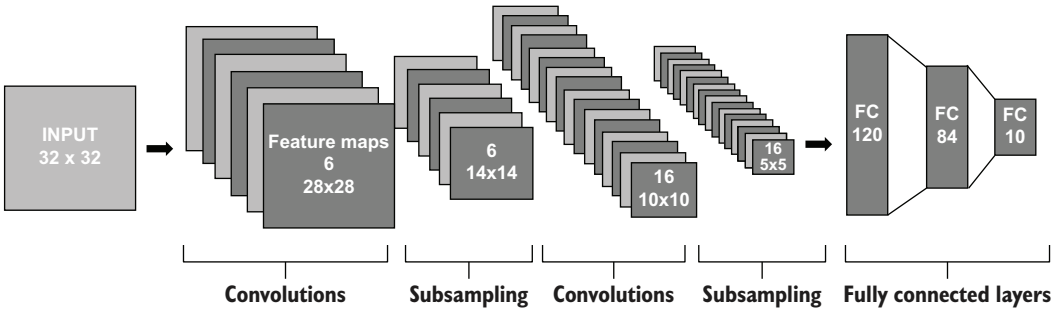

Figure 10.4 LeNet architecture for MNIST digits classification

This sequence is stacked, and the number of filters is increased as we go from the input to the output of the CNN. Notice, that we are both learning more features by increasing the number of filters and reducing the feature map dimensions through the max-pooling operations. In the last layer, we flatten the feature map and add a dense layer followed by a softmax classifier.

In the following code listing, we start off by loading the MNIST dataset and reshaping the images to the correct image size. We define our training parameters and model parameters, followed by the definition of CNN architecture. Notice a sequence of convolutional, ReLU, and max-pooling operations. We compile the model and define a set of callback functions that will be executed during model training. We record the training history and evaluate the model on the test dataset. Finally, we save the prediction results and generate accuracy and loss plots. We are now ready to implement a simple MNIST CNN architecture from scratch using Keras/TensorFlow!

Listing 10.2 Simple CNN for MNIST classification

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras

from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D, Activation

from keras.callbacks import ModelCheckpoint
```

```
from keras.callbacks import TensorBoard
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping

import math
import matplotlib.pyplot as plt

tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"

def scheduler(epoch, lr):   <-- Learning rate schedule
    if epoch < 4:
        return lr
    else:
        return lr * tf.math.exp(-0.1)


if __name__ == "__main__":

    img_rows, img_cols = 28, 28
    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols,
    ➥ 1).astype("float32") / 255
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols,
    ➥ 1).astype("float32") / 255

    y_train_label = keras.utils.to_categorical(y_train)
    y_test_label = keras.utils.to_categorical(y_test)
    num_classes = y_train_label.shape[1]

    batch_size = 128      | Training
    num_epochs = 8        | parameters

    num_filters_l1 = 32   | Model
    num_filters_l2 = 64   | parameters

    #CNN architecture
    cnn = Sequential()
    cnn.add(Conv2D(num_filters_l1, kernel_size = (5, 5),
    ➥ input_shape=(img_rows, img_cols, 1), padding='same'))
    cnn.add(Activation('relu'))
    cnn.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    cnn.add(Conv2D(num_filters_l2, kernel_size = (5, 5), padding='same'))
    cnn.add(Activation('relu'))
    cnn.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    cnn.add(Flatten())
    cnn.add(Dense(128))
    cnn.add(Activation('relu'))

    cnn.add(Dense(num_classes))
    cnn.add(Activation('softmax'))
```

```
cnn.compile(
    loss=keras.losses.CategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=["accuracy"]
)

cnn.summary()

#define callbacks
file_name = SAVE_PATH + 'lenet-weights-checkpoint.h5'
checkpoint = ModelCheckpoint(file_name, monitor='val_loss', verbose=1,
➥ save_best_only=True, mode='min')
reduce_lr = LearningRateScheduler(scheduler, verbose=1)
early_stopping = EarlyStopping(monitor='val_loss', min_delta=0.01,
➥ patience=16, verbose=1)
#tensor_board = TensorBoard(log_dir='./logs', write_graph=True)
callbacks_list = [checkpoint, reduce_lr, early_stopping]

hist = cnn.fit(x_train, y_train_label, batch_size=batch_size,
➥ epochs=num_epochs, callbacks=callbacks_list,
➥ validation_split=0.2)

test_scores = cnn.evaluate(x_test, y_test_label,
➥ verbose=2)

print("Test loss:", test_scores[0])
print("Test accuracy:", test_scores[1])

y_prob = cnn.predict(x_test)
y_pred = y_prob.argmax(axis=-1)

submission = pd.DataFrame(index=pd.RangeIndex(start=1, stop=10001,
➥ step=1), columns=['Label'])
submission['Label'] = y_pred.reshape(-1,1)
submission.index.name = "ImageId"
submission.to_csv(SAVE_PATH + '/lenet_pred.csv', index=True, header=True)

plt.figure()
plt.plot(hist.history['loss'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_loss'], '--r', lw=2.0, label='val')
plt.title('LeNet model')
plt.xlabel('Epochs')
plt.ylabel('Cross-Entropy Loss')
plt.legend(loc='upper right')
plt.show()

plt.figure()
plt.plot(hist.history['accuracy'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_accuracy'], '--r', lw=2.0, label='val')
plt.title('LeNet model')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')
plt.show()
```

Model training

Model evaluation

```
plt.figure()
plt.plot(hist.history['lr'], lw=2.0, label='learning rate')
plt.title('LeNet model')
plt.xlabel('Epochs')
plt.ylabel('Learning Rate')
plt.legend()
plt.show()
```

With CNN architecture, we can achieve an impressive 99% accuracy on the test set! Notice how the training and validation curves differ for both cross-entropy loss and accuracy. Early stopping enables us to stop the training when the validation loss does not improve over several epochs.

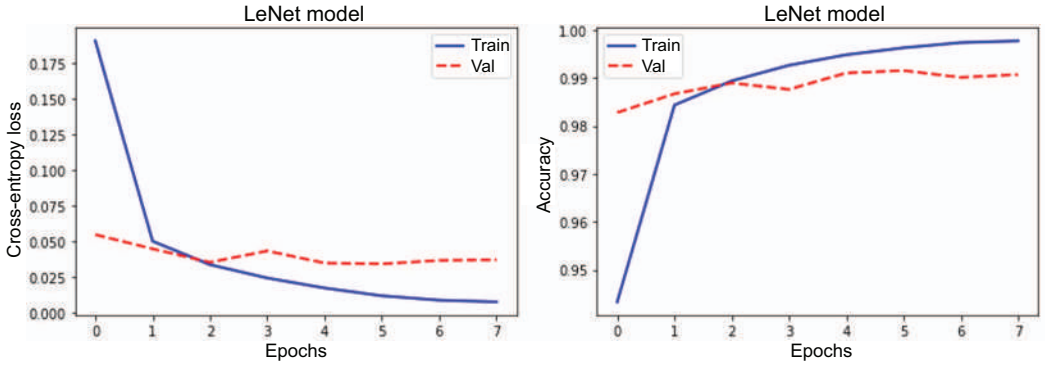

Figure 10.5 LeNet CNN loss and accuracy plots for training and validation datasets

10.2.2 ResNet image search

The goal of image search is to retrieve images from a database similar to the query image. In this section, we'll be using a pretrained ResNet50 CNN to encode a collection of images into dense vectors. This allows us to find similar images by computing distances between vectors.

The ResNet50 architecture (He et al.'s "Deep Residual Learning for Image Recognition," Conference on Computer Vision and Pattern Recognition, 2016) uses skip connections to avoid the vanishing gradient problem. Skip connections skip some layers in the network, as shown in figure 10.6.

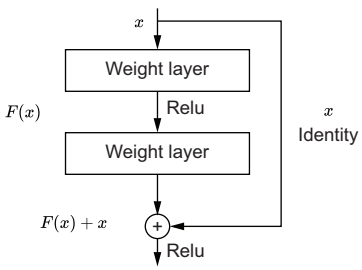

Figure 10.6 Skip connection

The core idea behind ResNet is to backpropagate through the identity function to preserve the gradient. The gradient is then multiplied by one and its value will be maintained in the earlier layers. This enables us to stack many of such layers and create very deep architectures. Let $H = F(x) + x$, and then we get the following equation.

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H}\frac{\partial H}{\partial x} = \frac{\partial L}{\partial H}\left(\frac{\partial F}{\partial x} + 1\right) = \frac{\partial L}{\partial H}\frac{\partial F}{\partial x} + \frac{\partial L}{\partial H} \tag{10.9}$$

In this section, we'll be using the pretrained on the ImageNet convolutional base of ResNet50 CNN to encode every image in the Caltech 101 dataset. We start off by downloading the Caltech 101 dataset from http://www.vision.caltech.edu/datasets/. We select the model pretrained on ImageNet ResNet50 as our base model, and we set the output layer as the average pool layer, which effectively encodes an input image into a 2048-dimensional vector. We compute ResNet50 encoding of the dataset and store it in the activations list. To further save space, we compress the 2048-dimensional ResNet50 encoded vectors using principal component analysis (PCA) down to 300-dimensional vectors. We retrieve the nearest neighbor images by sorting vectors according to cosine similarity. We are now ready to implement our image search architecture from scratch using Keras/TensorFlow.

Listing 10.3 ResNet50 image search

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras

from keras import Model
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input

from keras.callbacks import ModelCheckpoint
from keras.callbacks import TensorBoard
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping

import os
import random
from PIL import Image
from scipy.spatial import distance
from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"
```

```
DATA_PATH = "/content/drive/MyDrive/data/101_ObjectCategories/"

def get_closest_images(acts, query_image_idx, num_results=5):

    num_images, dim = acts.shape
    distances = []
    for image_idx in range(num_images):
        distances.append(distance.euclidean(acts[query_image_idx, :],
        ➥ acts[image_idx, :]))
    #end for
    idx_closest  = sorted(range(len(distances)), key=lambda k:
    ➥ distances[k])[1:num_results+1]

    return idx_closest

def get_concatenated_images(images, indexes, thumb_height):

    thumbs = []
    for idx in indexes:
        img = Image.open(images[idx])
        img = img.resize((int(img.width * thumb_height / img.height),
        ➥ int(thumb_height)), Image.ANTIALIAS)
        if img.mode != "RGB":
            img = img.convert("RGB")
        thumbs.append(img)
    concat_image = np.concatenate([np.asarray(t) for t in thumbs], axis=1)

    return concat_image

if __name__ == "__main__":

    num_images = 5000
    images = [os.path.join(dp,f) for dp, dn, filenames in
    ➥ os.walk(DATA_PATH) for f in filenames \
              if os.path.splitext(f)[1].lower() in ['.jpg','.png','.jpeg']]
    images = [images[i] for i in sorted(random.sample(range(len(images)),
    ➥ num_images))]

    #CNN encodings
    base_model = ResNet50(weights='imagenet')
    model = Model(inputs=base_model.input,
     outputs=base_model.get_layer('avg_pool').output)

    activations = []
    for idx, image_path in enumerate(images):
        if idx % 100 == 0:
            print('getting activations for %d/%d image...'
            ➥ %(idx,len(images)))
        img = image.load_img(image_path, target_size=(224, 224))
        x = image.img_to_array(img)
        x = np.expand_dims(x, axis=0)
        x = preprocess_input(x)
        features = model.predict(x)
        activations.append(features.flatten().reshape(1,-1))
```

Pretrained on the ImageNet ResNet50 model

```
print('computing PCA...')
acts = np.concatenate(activations, axis=0)
pca = PCA(n_components=300)
pca.fit(acts)
acts = pca.transform(acts)

print('image search...')
query_image_idx = int(num_images*random.random())
idx_closest = get_closest_images(acts, query_image_idx)
query_image = get_concatenated_images(images, [query_image_idx], 300)
results_image = get_concatenated_images(images, idx_closest, 300)

plt.figure()
plt.imshow(query_image)
plt.title("query image (%d)" %query_image_idx)
plt.show()

plt.figure()
plt.imshow(results_image)
plt.title("result images")
plt.show()
```

Reduces the activation dimension

Figure 10.7 shows the query image of a chair (left) and a retrieved nearest neighbor image (right). The retrieved images closely resemble the query image. In the next section, we will introduce neural networks for sequential data.

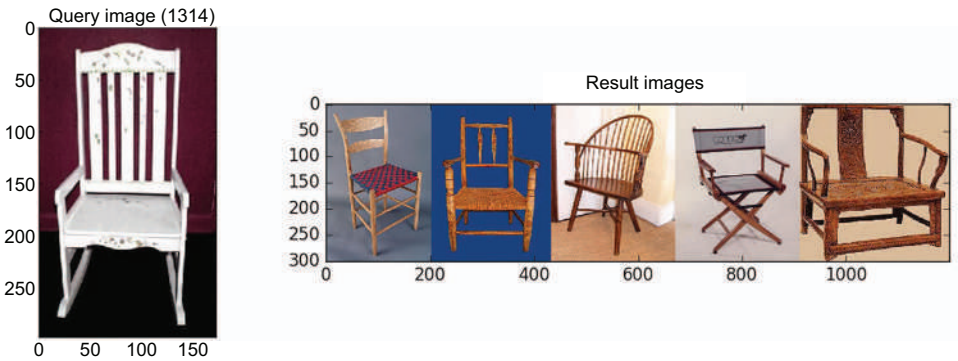

Figure 10.7 ResNet50 image search results

10.3 Recurrent neural nets

Recurrent neural nets (RNNs) are designed to process sequential data. RNNs maintain an internal state of the past and provide a natural way to encode a sequence (Seq) into a vector (Vec), and vice versa. Application of RNNs range from language generation (Vec2Seq) to sequence classification (Seq2Vec) to sequence translation (Seq2Seq). In this section, we'll focus on sequence classification using bidirectional LSTM and sequence similarity using pretrained word embeddings.

RNNs use a hidden layer that incorporates the current input x_t and prior state h_{t-1}.

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \tag{10.10}$$

Here, W_{hh} are the hidden-to-hidden weights, W_{xh} are the input-to-hidden weights, and b_h is the bias term. Optionally, the outputs y_t can be produced at every step.

$$y_t = g(W_{hy}h_t + b_y) \tag{10.11}$$

The overall architecture is captured in figure 10.8. Let's look at a few applications of RNNs in the next subsection.

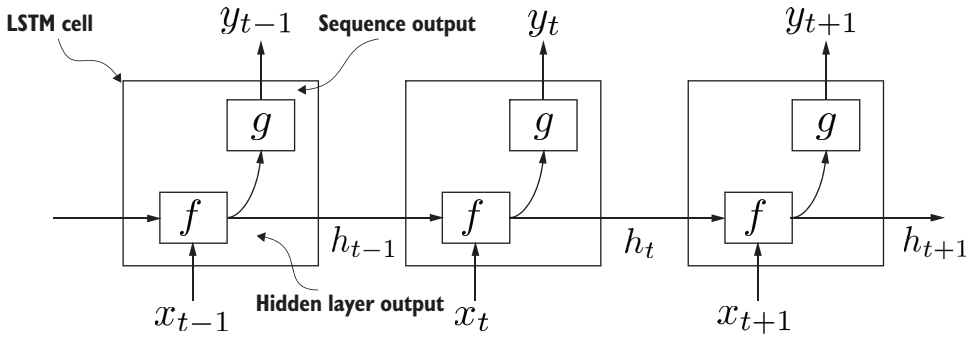

Figure 10.8 LSTM recurrent neural network architecture

10.3.1 LSTM sequence classification

In this section, our goal is to determine the sentiment of IMDb movie reviews. We start by tokenizing each word in the review and converting it into a vector via a trainable embedding layer. The sequence of word vectors forms the input to a forward LSTM model, which encodes the sequence into a vector. In parallel, we input our sequence into a backward LSTM model and concatenate its vector output with the forward model to produce a latent representation of the movie review.

$$\begin{aligned} h_t^{\rightarrow} &= f(W_{hh}^{\rightarrow}h_{t-1}^{\rightarrow} + W_{xh}^{\rightarrow}x_t + b_h^{\rightarrow}) \\ h_t^{\leftarrow} &= f(W_{hh}^{\leftarrow}h_{t-1}^{\leftarrow} + W_{xh}^{\leftarrow}x_t + b_h^{\leftarrow}) \\ h_t &= [h_t^{\rightarrow}, h_t^{\leftarrow}] \end{aligned} \tag{10.12}$$

Here, h_t takes into account information from the past and the future. Combined, the two LSTMs are known as *bidirectional LSTM* and process input data as if time is running forward and backward. Figure 10.9 shows the architecture of bidirectional RNN.

Having encoded our review into a vector, we average several dense layers with l2 regularization and dropout to classify the sentiment as positive or negative via the sigmoid

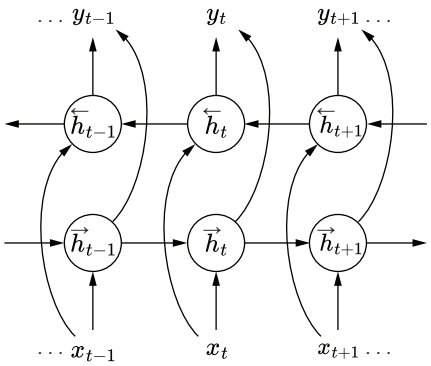

Figure 10.9 Bidirectional RNN architecture

activation function in the output layer. Let's take a look at the code for LSTM sequence classification based on the Keras/TensorFlow library.

Listing 10.4 Bidirectional LSTM for sentiment classification

```
import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras

from keras.models import Sequential
from keras.layers import LSTM, Bidirectional
from keras.layers import Dense, Dropout, Activation, Embedding

from keras import regularizers
from keras.preprocessing import sequence
from keras.utils import np_utils

from keras.callbacks import ModelCheckpoint
from keras.callbacks import TensorBoard
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping

import matplotlib.pyplot as plt

tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"

def scheduler(epoch, lr):    <-- Learning rate schedule
    if epoch < 4:
        return lr
    else:
        return lr * tf.math.exp(-0.1)

if __name__ == "__main__":

    #load dataset
```

```
max_words = 20000                                   <-- Top 20K most frequent words
seq_len = 200                                       <-- First 200 words of each movie review
(x_train, y_train), (x_val, y_val) = keras.datasets.imdb.load_data(
➥ num_words=max_words)

x_train = keras.utils.pad_sequences(x_train, maxlen=seq_len)
x_val = keras.utils.pad_sequences(x_val, maxlen=seq_len)

batch_size = 256                                    <-- Training paramaters
num_epochs = 8

hidden_size = 64                                    <-- Model parameters
embed_dim = 128
lstm_dropout = 0.2
dense_dropout = 0.5
weight_decay = 1e-3

#LSTM architecture
model = Sequential()
model.add(Embedding(max_words, embed_dim, input_length=seq_len))
model.add(Bidirectional(LSTM(hidden_size, dropout=lstm_dropout,
➥ recurrent_dropout=lstm_dropout)))
model.add(Dense(hidden_size,
 kernel_regularizer=regularizers.l2(weight_decay),
 ➥ activation='relu'))
model.add(Dropout(dense_dropout))
model.add(Dense(hidden_size/4,
 kernel_regularizer=regularizers.l2(weight_decay), activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(
    loss=keras.losses.BinaryCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=["accuracy"]
)

model.summary()

#define callbacks
file_name = SAVE_PATH + 'lstm-weights-checkpoint.h5'
checkpoint = ModelCheckpoint(file_name, monitor='val_loss', verbose=1,
➥ save_best_only=True, mode='min')
reduce_lr = LearningRateScheduler(scheduler, verbose=1)
early_stopping = EarlyStopping(monitor='val_loss', min_delta=0.01,
➥ patience=16, verbose=1)
#tensor_board = TensorBoard(log_dir='./logs', write_graph=True)
callbacks_list = [checkpoint, reduce_lr, early_stopping]

hist = model.fit(x_train, y_train, batch_size=
➥ batch_size, epochs=num_epochs, callbacks=
➥ callbacks_list, validation_data=(x_val, y_val))    <-- Model training

test_scores = model.evaluate(x_val, y_val,
➥ verbose=2)                                        <-- Model evaluation
```

```
print("Test loss:", test_scores[0])
print("Test accuracy:", test_scores[1])

plt.figure()
plt.plot(hist.history['loss'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_loss'], '--r', lw=2.0, label='val')
plt.title('LSTM model')
plt.xlabel('Epochs')
plt.ylabel('Cross-Entropy Loss')
plt.legend(loc='upper right')
plt.show()

plt.figure()
plt.plot(hist.history['accuracy'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_accuracy'], '--r', lw=2.0, label='val')
plt.title('LSTM model')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')
plt.show()

plt.figure()
plt.plot(hist.history['lr'], lw=2.0, label='learning rate')
plt.title('LSTM model')
plt.xlabel('Epochs')
plt.ylabel('Learning Rate')
plt.legend()
plt.show()
```

As we can see from figure 10.10, we achieve a test classification accuracy of 85% on the IMDb movie review database. In the next section, we'll explore multiple-input neural network models.

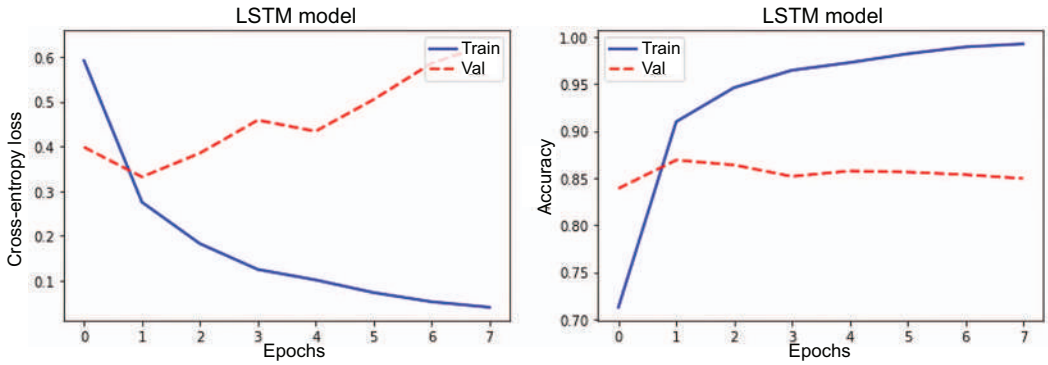

Figure 10.10 Bidirectional LSTM for sentiment classification loss and accuracy on training and validation datasets

10.3.2 Multi-input model

In this section, we will study a multi-input model as applied to sequence similarity, comparing two questions (one for each input branch) and deciding whether they have a similar meaning. This will help us determine whether the questions are redundant or duplicates of each other. This challenge appeared as part of the Quora Questions Pairs Kaggle data science competition, and the model we'll develop could be used as part of an ensemble leading to a high-performing solution.

In this case, we will use pretrained Glove word embeddings (Jeffrey Pennington, Richard Socher, Christopher Manning's "GloVe: Global Vectors for Word Representation", EMNLP, 2014) to encode each word into a 300-dimensional dense vector. We'll use 1D convolutions and max-pooling operations to process the sequence of data for each input branch and then concatenate the results into a single vector. After several dense layers, we will compute the probability of duplicate questions based on the sigmoid activation function.

The advantage of using 1D convolutions is that they are computationally faster and can be done in parallel (since there's no recurrent loop to unroll). Their accuracy will depend on whether the information in the recent past is more important than the distant past (in which case, LSTMs will be more accurate) or whether the information is equally important in different timeframes of the past (in which case, 1D convolution will be more accurate).

We'll be using the Quora question pairs dataset available for download from Kaggle: https://www.kaggle.com/datasets/quora/question-pairs-dataset. Let's examine how we can implement a multi-input sequence similarity model end-to-end in Keras/TensorFlow!

Listing 10.5 Multi-input neural network model for sequence similarity

```
import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras

import os
import re
import csv
import codecs

from keras.models import Model
from keras.layers import Input, Flatten, Concatenate, LSTM, Lambda, Dropout
from keras.layers import Dense, Dropout, Activation, Embedding
from keras.layers import Conv1D, MaxPooling1D
from keras.layers import TimeDistributed, Bidirectional, BatchNormalization

from keras import backend as K
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
```

```
from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer

from keras import regularizers
from keras.preprocessing import sequence
from keras.utils import np_utils

from keras.callbacks import ModelCheckpoint
from keras.callbacks import TensorBoard
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping

import matplotlib.pyplot as plt

tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"
DATA_PATH = "/content/drive/MyDrive/data/"

GLOVE_DIR = DATA_PATH
TRAIN_DATA_FILE = DATA_PATH + 'quora_train.csv'
TEST_DATA_FILE = DATA_PATH + 'quora_test.csv'
MAX_SEQUENCE_LENGTH = 30
MAX_NB_WORDS = 200000
EMBEDDING_DIM = 300
VALIDATION_SPLIT = 0.01

def scheduler(epoch, lr):
    if epoch < 4:
        return lr
    else:
        return lr * tf.math.exp(-0.1)

def text_to_wordlist(row, remove_stopwords=False, stem_words=False):

    text = row['question']
    if type(text) is str:
        text = text.lower().split()
    else:
        return " "


    if remove_stopwords:
        stops = set(stopwords.words("english"))
        text = [w for w in text if not w in stops]

    text = " ".join(text)

    # Clean the text
    text = re.sub(r"[^A-Za-z0-9^,!.\/'+-=]", " ", text)

    if stem_words:
        text = text.split()
        stemmer = SnowballStemmer('english')
```

Converts words to lowercase and splits them

Removes stop words

```
        stemmed_words = [stemmer.stem(word) for word in text]
        text = " ".join(stemmed_words)

    return(text)

if __name__ == "__main__":


    print('Indexing word vectors...')
    embeddings_index = {}
    f = codecs.open(os.path.join(GLOVE_DIR, 'glove.6B.
    ➥ 300d.txt'), encoding='utf-8')
    for line in f:
        values = line.split(' ')
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs
    f.close()
    print('Found %s word vectors.' % len(embeddings_index))

    train_df = pd.read_csv(TRAIN_DATA_FILE)
    test_df  = pd.read_csv(TEST_DATA_FILE)

    q1df = train_df['question1'].reset_index()
    q2df = train_df['question2'].reset_index()
    q1df.columns = ['index', 'question']
    q2df.columns = ['index', 'question']
    texts_1 = q1df.apply(text_to_wordlist, axis=1, raw=False).tolist()
    texts_2 = q2df.apply(text_to_wordlist, axis=1, raw=False).tolist()
    labels = train_df['is_duplicate'].astype(int).tolist()
    print('Found %s texts.' % len(texts_1))
    del q1df
    del q2df

    q1df = test_df['question1'].reset_index()
    q2df = test_df['question2'].reset_index()
    q1df.columns = ['index', 'question']
    q2df.columns = ['index', 'question']
    test_texts_1 = q1df.apply(text_to_wordlist, axis=1, raw=False).tolist()
    test_texts_2 = q2df.apply(text_to_wordlist, axis=1, raw=False).tolist()
    test_labels = np.arange(0, test_df.shape[0])
    print('Found %s texts.' % len(test_texts_1))
    del q1df
    del q2df

    tokenizer = Tokenizer(nb_words=MAX_NB_WORDS)
    tokenizer.fit_on_texts(texts_1 + texts_2 + test_texts_1 + test_texts_2)
    sequences_1 = tokenizer.texts_to_sequences(texts_1)
    sequences_2 = tokenizer.texts_to_sequences(texts_2)
    word_index = tokenizer.word_index
    print('Found %s unique tokens.' % len(word_index))

    test_sequences_1 = tokenizer.texts_to_sequences(test_texts_1)
    test_sequences_2 = tokenizer.texts_to_sequences(test_texts_2)
```

Shortens words to their stems

Loads embeddings

Loads the dataset

```
data_1 = pad_sequences(sequences_1, maxlen=MAX_SEQUENCE_LENGTH)
data_2 = pad_sequences(sequences_2, maxlen=MAX_SEQUENCE_LENGTH)
labels = np.array(labels)
print('Shape of data tensor:', data_1.shape)
print('Shape of label tensor:', labels.shape)

test_data_1 = pad_sequences(test_sequences_1, maxlen=MAX_SEQUENCE_LENGTH)
test_data_2 = pad_sequences(test_sequences_2, maxlen=MAX_SEQUENCE_LENGTH)
test_labels = np.array(test_labels)
del test_sequences_1
del test_sequences_2
del sequences_1
del sequences_2

print('Preparing embedding matrix...')
nb_words = min(MAX_NB_WORDS, len(word_index))

embedding_matrix = np.zeros((nb_words, EMBEDDING_DIM))
for word, i in word_index.items():
    if i >= nb_words:
        continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
print('Null word embeddings: %d' % np.sum(np.sum(embedding_matrix,
➥ axis=1) == 0))

embedding_layer = Embedding(nb_words,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False)

sequence_1_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences_1 = embedding_layer(sequence_1_input)
x1 = Conv1D(128, 3, activation='relu')(embedded_sequences_1)
x1 = MaxPooling1D(10)(x1)
x1 = Flatten()(x1)
x1 = Dense(64, activation='relu')(x1)
x1 = Dropout(0.2)(x1)

sequence_2_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences_2 = embedding_layer(sequence_2_input)
y1 = Conv1D(128, 3, activation='relu')(embedded_sequences_2)
y1 = MaxPooling1D(10)(y1)
y1 = Flatten()(y1)
y1 = Dense(64, activation='relu')(y1)
y1 = Dropout(0.2)(y1)

merged = Concatenate()([x1, y1])
merged = BatchNormalization()(merged)
merged = Dense(64, activation='relu')(merged)
merged = Dropout(0.2)(merged)
merged = BatchNormalization()(merged)
```

```
preds = Dense(1, activation='sigmoid')(merged)

model = Model(inputs=[sequence_1_input,sequence_2_input], outputs=preds)

model.compile(
    loss=keras.losses.BinaryCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=["accuracy"]
)

model.summary()

file_name = SAVE_PATH + 'multi-input-weights-checkpoint.h5'
checkpoint = ModelCheckpoint(file_name, monitor='val_loss', verbose=1,
➥ save_best_only=True, mode='min')
reduce_lr = LearningRateScheduler(scheduler, verbose=1)
early_stopping = EarlyStopping(monitor='val_loss', min_delta=0.01,
➥ patience=16, verbose=1)
#tensor_board = TensorBoard(log_dir='./logs', write_graph=True)
callbacks_list = [checkpoint, reduce_lr, early_stopping]

hist = model.fit([data_1, data_2], labels,
➥ batch_size=1024, epochs=10, callbacks=callbacks_list,
➥ validation_split=VALIDATION_SPLIT)

num_test = 100000
preds = model.predict([test_data_1[:num_test,:],
➥ test_data_2[:num_test,:]])

quora_submission = pd.DataFrame({"test_id":test_labels[:num_test],
➥ "is_duplicate":preds.ravel()})
quora_submission.to_csv(SAVE_PATH + "quora_submission.csv", index=False)

plt.figure()
plt.plot(hist.history['loss'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_loss'], '--r', lw=2.0, label='val')
plt.title('Multi-Input model')
plt.xlabel('Epochs')
plt.ylabel('Cross-Entropy Loss')
plt.legend(loc='upper right')
plt.show()
#plt.savefig('./figures/lstm_loss.png')

plt.figure()
plt.plot(hist.history['accuracy'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_accuracy'], '--r', lw=2.0, label='val')
plt.title('Multi-Input model')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')
plt.show()
#plt.savefig('./figures/lstm_acc.png')

plt.figure()
plt.plot(hist.history['lr'], lw=2.0, label='learning rate')
```

Multi-input architecture

Model training

Model evaluation

```
plt.title('Multi-Input model')
plt.xlabel('Epochs')
plt.ylabel('Learning Rate')
plt.legend()
plt.show()
#plt.savefig('./figures/lstm_learning_rate.png')
```

We can see the accuracy hovering around 69% on the validation dataset in figure 10.11, and we know there's room for improvement. We can potentially increase the capacity of our model or improve data representation.

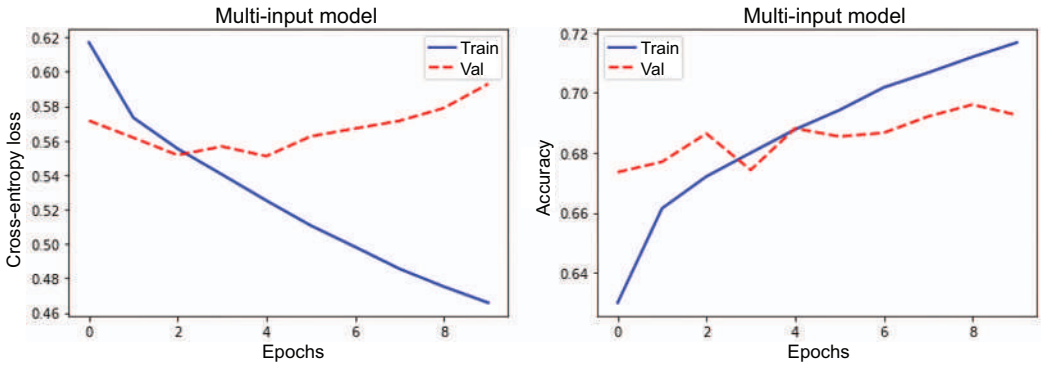

Figure 10.11 Multi-input model loss and accuracy for training and validation datasets

In this section, we focused on understanding an important class of neural nets: recurrent neural networks. We studied how they work and focused on two applications: sequence classification and sequence similarity. In the next section, we will examine different neural network optimizers.

10.4 Neural network optimizers

What are some of the most popular optimization algorithms used for training neural networks? We will attempt to answer this question using a convolutional neural network (CNN) trained on the CIFAR-100 dataset with Keras/TensorFlow.

Stochastic gradient descent (SGD) updates parameters θ in the negative direction of the gradient g by taking a subset or mini-batch of data of size m.

$$\begin{aligned} g &= \frac{1}{m}\nabla_\theta \sum_i L\left(f\left(x^{(i)};\theta\right), y^{(i)}\right) \\ \theta &= \theta - \epsilon_k \times g \end{aligned} \tag{10.13}$$

Here, $f(x_i; \theta)$ is a neural network trained on examples x_i and labels y_i and L is the loss function. The gradient of loss L is computed with respect to model parameters θ. The learning rate ϵ_k determines the size of the step that the algorithm takes along the

gradient (in the negative direction in the case of minimization and in the positive direction in the case of maximization).

The learning rate is a function of iteration k and is the single most important hyperparameter. A learning rate that is too high (e.g., > 0.1) can lead to parameter updates that miss the optimum value; a learning rate that is too low (e.g., $<$ 1e–5) will result in an unnecessarily long training time. A good strategy is to start with a learning rate of 1e–3 and use a learning rate schedule that reduces the learning rate as a function of iterations. In general, we want the learning rate to satisfy the Robbins-Monroe conditions.

$$\begin{aligned} \sum_k \epsilon_k &= \infty \\ \sum_k \epsilon_k^2 &< \infty \end{aligned} \tag{10.14}$$

The first condition ensures the algorithm will be able to find a locally optimal solution, regardless of the starting point, and the second one controls oscillations.

Momentum accumulates the exponentially decaying moving average of past gradients and continues to move in their direction; thus, the step size depends on how large and aligned the sequence of gradients are. Common values of momentum parameter α are 0.5 and 0.9.

$$\begin{aligned} v &= \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_i L\left(f\left(x^{(i)}; \theta \right), y^{(i)} \right) \right) \\ \theta &= \theta + v \end{aligned} \tag{10.15}$$

Nesterov momentum is inspired by Nesterov's accelerated gradient method. The difference between Nesterov and standard momentum is where the gradient is evaluated, with the gradient being evaluated after the current velocity is applied in Nesterov momentum. Thus, Nesterov momentum adds a correction factor to the gradient.

$$\begin{aligned} v &= \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_i L\left(f\left(x^{(i)}; \theta + \alpha \times v \right), y^{(i)} \right) \right) \\ \theta &= \theta + v \end{aligned} \tag{10.16}$$

AdaGrad is an adaptive method for setting the learning rate (John Duchi, Elad Hazan, and Yoram Singer's "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, 2011). Consider the two scenarios in figure 10.12.

In the case of a slowly varying objective (left), the gradient would typically (at most points) have a small magnitude. As a result, we would need a large learning rate to

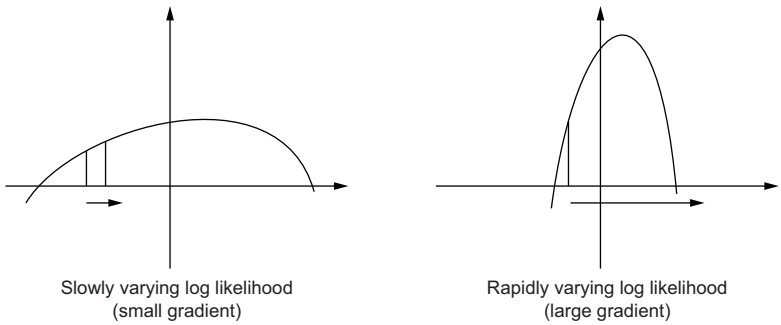

Figure 10.12 Slowly varying (left) and rapidly varying (right) log likelihood

quickly reach the optimum. In the case of a rapidly varying objective (right), the gradient would typically be very large. Using a large learning rate would result in very large, steps, oscillating around but not reaching the optimum. These two situations occur because the learning rate is set independently of the gradient. AdaGrad solves this by accumulating squared norms of gradients seen so far and dividing the learning rate by the square root of the following sum.

$$\begin{aligned} g &= \frac{1}{m}\nabla_\theta \sum_i L\left(f\left(x^{(i)};\theta\right), y^{(i)}\right) \\ s &= s + g^T g \\ \theta &= \theta - \epsilon_k \times \frac{g}{\sqrt{s + eps}} \end{aligned} \tag{10.17}$$

As a result, parameters that receive high gradients have their effective learning rate reduced and parameters that receive small gradients have their effective learning rate increased. The net effect is greater progress in the more gently sloped directions of parameter space and more cautious updates in the presence of large gradients.

RMSProp modifies AdaGrad by changing the gradient accumulation into an exponentially weighted moving average (i.e., it discards history from the distant past).

$$\begin{aligned} g &= \frac{1}{m}\nabla_\theta \sum_i L\left(f\left(x^{(i)};\theta\right), y^{(i)}\right) \\ s &= \text{decay\_rate} \times s + (1 - \text{decay\_rate}) g^T g \\ \theta &= \theta - \epsilon_k \times \frac{g}{\sqrt{s + eps}} \end{aligned} \tag{10.18}$$

Notice that AdaGrad implies a decreasing learning rate, even if the gradients remain constant, due to the accumulation of gradients from the beginning of training. By introducing an exponentially weighted moving average, we are weighing the recent past

more heavily in comparison to the distant past. As a result, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam derives from *adaptive moments*. It can be seen as a variant on the combination of RMSProp and momentum, and the update looks like RMSProp, except a smooth version of the gradient is used instead of the raw stochastic gradient. The full Adam update also includes a bias correction mechanism (Diederik P. Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization," *International Conference on Learning Representations*, 2015).

$$\begin{aligned} g &= \frac{1}{m}\nabla_\theta \sum_i L\left(\left(x^{(i)};\theta\right), y^{(i)}\right) \\ m &= \beta_1 m + (1-\beta_1)g \\ s &= \beta_2 v + (1-\beta_2)g^T g \\ \theta &= \theta - \epsilon_k \times \frac{m}{\sqrt{s+eps}} \end{aligned} \tag{10.19}$$

The recommended values in the paper are ϵ = 1e–8, β_1 = 0.9, and β_2 = 0.999. Now, let's examine the performance of different optimizers using Keras/TensorFlow!

Listing 10.6 Neural network optimizers

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras

from keras import backend as K
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D, Activation

from keras.callbacks import ModelCheckpoint
from keras.callbacks import TensorBoard
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping

import math
import matplotlib.pyplot as plt

tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"

def scheduler(epoch, lr):
    if epoch < 4:
        return lr
    else:
        return lr * tf.math.exp(-0.1)
```

```
if __name__ == "__main__":

    img_rows, img_cols = 32, 32
    (x_train, y_train), (x_test, y_test) =
    ➥ keras.datasets.cifar100.load_data()
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols,
    ➥ 3).astype("float32") / 255
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols,
    ➥ 3).astype("float32") / 255

    y_train_label = keras.utils.to_categorical(y_train)
    y_test_label = keras.utils.to_categorical(y_test)
    num_classes = y_train_label.shape[1]

    batch_size = 256
    num_epochs = 32

    num_filters_l1 = 64
    num_filters_l2 = 128

    #CNN architecture
    cnn = Sequential()
    cnn.add(Conv2D(num_filters_l1, kernel_size = (5, 5),
    ➥ input_shape=(img_rows, img_cols, 3), padding='same'))
    cnn.add(Activation('relu'))
    cnn.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    cnn.add(Conv2D(num_filters_l2, kernel_size = (5, 5), padding='same'))
    cnn.add(Activation('relu'))
    cnn.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    cnn.add(Flatten())
    cnn.add(Dense(128))
    cnn.add(Activation('relu'))

    cnn.add(Dense(num_classes))
    cnn.add(Activation('softmax'))

    opt1 = tf.keras.optimizers.SGD()
    opt2 = tf.keras.optimizers.SGD(momentum=0.9,
    ➥ nesterov=True)
    opt3 = tf.keras.optimizers.RMSprop()
    opt4 = tf.keras.optimizers.Adam()

    optimizer_list = [opt1, opt2, opt3, opt4]

    history_list = []

    for idx in range(len(optimizer_list)):

        K.clear_session()

        opt = optimizer_list[idx]

        cnn.compile(
            loss=keras.losses.CategoricalCrossentropy(),
            optimizer=opt,
            metrics=["accuracy"]
```

cifar100 dataset

Training parameters

Model parameters

Optimizers

```
    )

    #define callbacks
    reduce_lr = LearningRateScheduler(scheduler, verbose=1)
    callbacks_list = [reduce_lr]

    #training loop
    hist = cnn.fit(x_train, y_train_label, batch_size=batch_size,
    ➥ epochs=num_epochs, callbacks=callbacks_list, validation_split=0.2)
    history_list.append(hist)

#end for

plt.figure()
plt.plot(history_list[0].history['loss'], 'b', lw=2.0, label='SGD')
plt.plot(history_list[1].history['loss'], '--r', lw=2.0, label='SGD
➥ Nesterov')
plt.plot(history_list[2].history['loss'], ':g', lw=2.0, label='RMSProp')
plt.plot(history_list[3].history['loss'], '-.k', lw=2.0, label='ADAM')
plt.title('LeNet, CIFAR-100, Optimizers')
plt.xlabel('Epochs')
plt.ylabel('Cross-Entropy Training Loss')
plt.legend(loc='upper right')
plt.show()
#plt.savefig('./figures/lenet_loss.png')

plt.figure()
plt.plot(history_list[0].history['val_accuracy'], 'b', lw=2.0,
➥ label='SGD')
plt.plot(history_list[1].history['val_accuracy'], '--r', lw=2.0,
➥ label='SGD Nesterov')
plt.plot(history_list[2].history['val_accuracy'], ':g', lw=2.0,
➥ label='RMSProp')
plt.plot(history_list[3].history['val_accuracy'], '-.k', lw=2.0,
➥ label='ADAM')
plt.title('LeNet, CIFAR-100, Optimizers')
plt.xlabel('Epochs')
plt.ylabel('Validation Accuracy')
plt.legend(loc='upper right')
plt.show()
#plt.savefig('./figures/lenet_loss.png')

plt.figure()
plt.plot(history_list[0].history['lr'], 'b', lw=2.0, label='SGD')
plt.plot(history_list[1].history['lr'], '--r', lw=2.0, label='SGD
➥ Nesterov')
plt.plot(history_list[2].history['lr'], ':g', lw=2.0, label='RMSProp')
plt.plot(history_list[3].history['lr'], '-.k', lw=2.0, label='ADAM')
plt.title('LeNet, CIFAR-100, Optimizers')
plt.xlabel('Epochs')
plt.ylabel('Learning Rate Schedule')
plt.legend(loc='upper right')
plt.show()
#plt.savefig('./figures/lenet_loss.png')
```

Figure 10.13 shows the loss and validation dataset accuracy for a LeNet neural network trained on the cifar100 dataset using different optimizers.

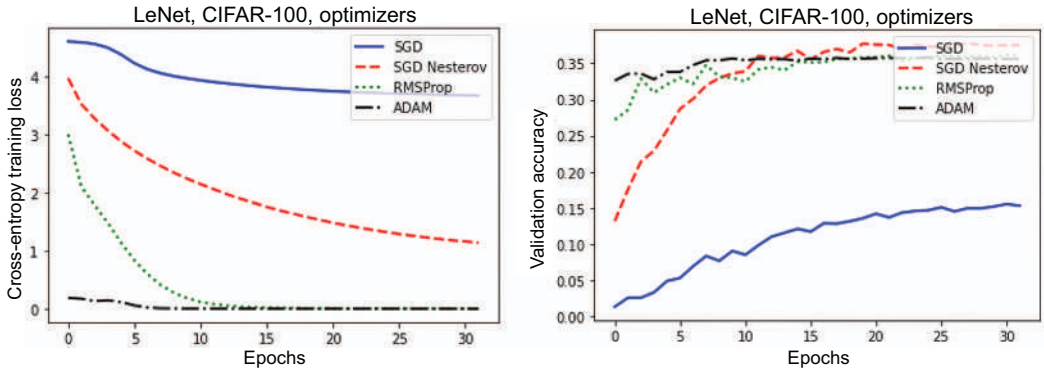

Figure 10.13 Cross-entropy loss and validation accuracy of LeNet trained on cifar100 using different optimizers

We can see that Adam and Nesterov Momentum optimizers experimentally produce the highest validation accuracy. In the following chapter, we will focus on advanced deep learning algorithms. We will learn to detect anomalies in time series data using a variational autoencoder (VAE), determine clusters using a mixture density network (MDN), learn to classify text with transformers, and classify citation graphs using a graph neural network (GNN).

10.5 Exercises

10.1 Explain the purpose of nonlinearities in neural networks.
10.2 Explain the vanishing/exploding gradient problem.
10.3 Describe some of the ways to increase neural model capacity and avoid overfitting.
10.4 Why does the number of filters increase in LeNet architecture as we go from input to output?
10.5 Explain how an Adam optimizer works.

Summary

- Multilayer perceptron consists of multiple densely connected layers followed by a nonlinearity.
- Cross-entropy loss is used in classification tasks, while mean squared error loss is used in regression tasks.
- Neural networks are optimized via a backpropagation algorithm, based on the chain rule.
- Increasing model capacity to avoid underfitting can be achieved by changing the model architecture (e.g., increasing the number of layers and hidden units per layer).

- Regularization to avoid overfitting can occur on multiple levels: weight decay, early stopping, and dropout.
- Convolutional neural nets work exceptionally well for image data and use convolution and pooling layers instead of vectorized matrix multiplications.
- Classic CNN architectures consist of a convolutional layer followed by ReLU nonlinear activation function and a max pooling layer.
- Pretrained CNNs can be used to extract feature vectors for images and used in applications such as image search.
- Recurrent neural nets are designed to process sequential data.
- Application of RNNs include language generation, sequence classification, and sequence translation.
- Multi-input neural nets can be constructed by concatenating the feature vector representations from individual input branches.
- Neural network optimizers include stochastic gradient descent, momentum, Nesterov momentum, AdaGrad, RMSProp, and Adam.