11 Advanced deep learning algorithms

This chapter covers

- Variational autoencoders for time series anomaly detection
- Mixture density networks using amortized variational inference
- Attention and transformers
- Graph neural networks
- ML research, including deep learning

In the previous chapter, we looked at fundamental deep learning algorithms to help represent numerical, image, and text data. In this chapter, we will continue our discussion with advanced deep learning algorithms. The algorithms have been selected for their state-of-the-art performance architectures and a wide range of applications. We will investigate generative models based on variational autoencoders (VAEs) and implement, from scratch, an anomaly detector for time series data. We'll continue our journey with an intriguing combination of neural networks and classical Gaussian mixture models (GMMs) via amortized variational inference (VI)

and implement mixture density networks (MDNs). We will then focus on the concept of attention and implement a transformer architecture from scratch for a classification task. Finally, we'll examine graph neural networks (GNNs) and use one to perform node classification for a citation graph. We will be using the Keras/TensorFlow deep learning library throughout this chapter.

11.1 Autoencoders

An *autoencoder* is an unsupervised neural network used for dimensionality reduction or feature extraction. They consist of an encoder followed by a hidden bottleneck layer followed by a decoder, where the encoder and decoder are both trainable neural networks, as shown in figure 11.1.

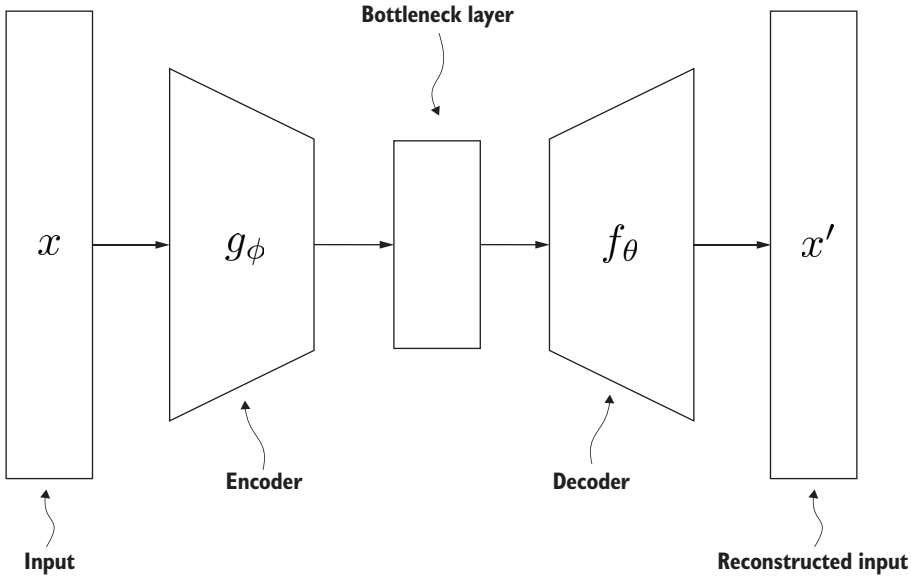


Figure 11.1 Autoencoder architecture, showing an encoder, a bottleneck layer, and a decoder

Autoencoders are trained to reconstruct their input. In other words, an autoencoder output should match the input as closely as possible. To prevent the neural network from learning a trivial identity function, the hidden layer in the middle is constrained to be a narrow bottleneck. Autoencoder training minimizes the reconstruction error by ensuring the hidden units capture the most relevant information in the input data.

In practice, autoencoders serve as feature extractors and do not lead to well-structured latent spaces. This is where VAEs come in (D. P. Kingma's "Variational Inference and Deep Learning: A New Synthesis," University of Amsterdam, 2017). A VAE also consists of an encoder and a decoder; however, instead of compressing its input image into a bottleneck layer, a VAE turns the image into parameters of statistical distribution (e.g., the mean and variance of a Gaussian random variable). The VAE

then draws a sample from this distribution and uses it to decode the sample back to its original input. Figure 11.2 shows the architecture of the VAE.

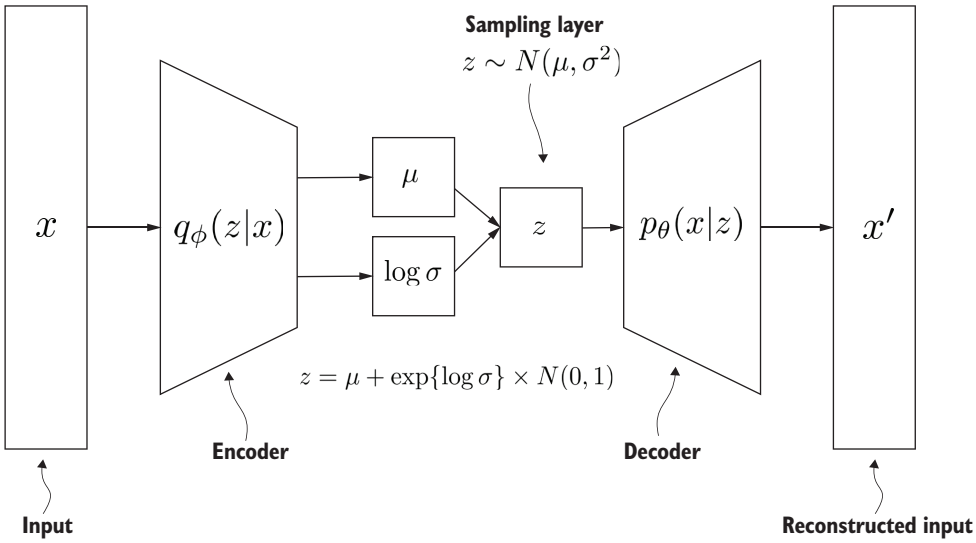


Figure 11.2 Variational autoencoder architecture, showing an encoder, a sampling layer, and a decoder

VAE training imposes a structure on the latent space such that every point can be decoded to a valid output. The optimization objective of the variational autoencoder is the evidence lower bound (ELBO). Let x represent the input space and z the latent space. Let $p(x|z)$ be the decoder distribution parameterized by θ that, conditioned on a sample z from the latent space, reconstructs the original input x. Similarly, let $q(z|x)$ be the encoder distribution parameterized by ϕ that takes the input x and encodes it into latent variable z. Note that both theta and phi parameters are trainable. Finally, let $p(z)$ be the prior distribution over the latent space. Since our goal for the variational posterior is to be as close as possible to the true posterior, the VAE is trained with the following loss function.

$$\min_{\theta,\phi} \text{Loss}(x;\theta,\phi) = E_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}\left(q_\phi(z|x)||p_\theta(z)\right) \tag{11.1}$$

The first term controls how well the VAE reconstructs a data point x from a sample z of the variational posterior, while the second term controls how close the variational posterior $q(z|x)$ is to the prior $p(z)$. If we assume the prior distribution $p(z)$ is Gaussian, we can write the KL divergence term as follows. In the following section, we will examine how we can apply VAE to anomaly detection in time series.

$$D_{KL}(q_\phi(z|x)||p_\theta(z)) = -\frac{1}{2}\sum_{d=1}^{D}\left(1 + \log\sigma_d^2 - \mu_d^2 - \sigma_d^2\right) \tag{11.2}$$

11.1.1 VAE anomaly detection in time series

Let's look at a VAE model that operates on time series data with the goal of anomaly detection. The architecture is shown in figure 11.3.

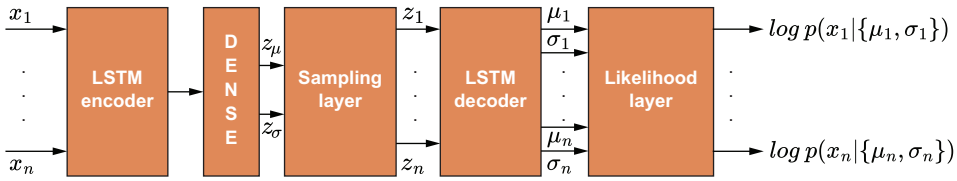


Figure 11.3 LSTM-VAE anomaly detector architecture

The input consists of n signals $x_1,\ldots, x_n$, and the output is the log probability of observing input x_i under normal (nonanomalous) training parameters μ_i, σ_i. This means the model is trained on nonanomalous data in an unsupervised fashion and when an anomaly does occur on a given input x_i, the corresponding log likelihood log $p(x_i|\{\mu_i, \sigma_i\})$ drops and we can threshold the resulting drop to signal an anomaly.

We assume a Gaussian likelihood; thus, every sensor has two degrees of freedom (μ, σ) to represent an anomaly. As a result, for n input sensors, we learn $2n$ output parameters (mean and variance) that can differentiate between anomalous and normal behavior.

While the input signals are independent, they are embedded in a joint latent space by the VAE in the sampling layer. The embedding is structured as a Gaussian that approximates standard normal $N(0,1)$ by minimizing KL divergence. The model is trained in an unsupervised fashion, with an objective function that achieves two goals: (1) maximizing the log likelihood output of the model, averaged over sensors and (2) structuring the embedding space to approximate $N(0, 1)$.

$$\begin{aligned}\min_\theta \text{Loss}(\theta) \quad = \quad & \min_\theta D_{KL}\left(N(z_\mu, z_\sigma)||N(0, 1)\right) \\ & -\frac{1}{n}\sum_{i=1}^{n}\log p(x_i|\mu_i, \sigma_i)\end{aligned} \tag{11.3}$$

We are now ready to implement the LSTM-VAE anomaly detector from scratch, using Keras/TensorFlow. In the following listing, we will load the NAB dataset (which can be found in the data folder of the code repo) and prepare the data for training. The Numenta Anomaly Benchmark (NAB) is a novel benchmark for evaluating algorithms for anomaly detection in streaming, online applications. Next, we define the anomaly

detector architecture along with the custom loss function and train the model. You may want to run the code listing in a Google Colab notebook (accessible at https://colab.research.google.com/) to understand the code step by step and accelerate model training via GPU.

Listing 11.1 LSTM-VAE anomaly detector

```
import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras
import tensorflow_probability as tfp

from keras.layers import Input, Dense, Lambda, Layer
from keras.layers import LSTM, RepeatVector
from keras.models import Model
from keras import backend as K
from keras import metrics
from keras import optimizers

import math
import json
from scipy.stats import norm
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import StandardScaler

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/"

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

nab_path = DATA_PATH + 'NAB/'
nab_data_path = nab_path

labels_filename = '/labels/combined_labels.json'
train_file_name = 'artificialNoAnomaly/art_daily_no_noise.csv'
test_file_name = 'artificialWithAnomaly/art_daily_jumpsup.csv'
```

```
#train_file_name = 'realAWSCloudwatch/rds_cpu_utilization_cc0c53.csv'
#test_file_name = 'realAWSCloudwatch/rds_cpu_utilization_e47b3b.csv'

labels_file = open(nab_path + labels_filename, 'r')
labels = json.loads(labels_file.read())
labels_file.close()

def load_data_frame_with_labels(file_name):
    data_frame = pd.read_csv(nab_data_path + file_name)
    data_frame['anomaly_label'] = data_frame['timestamp'].isin(
        labels[file_name]).astype(int)
    return data_frame

train_data_frame = load_data_frame_with_labels(train_file_name)
test_data_frame = load_data_frame_with_labels(test_file_name)

plt.plot(train_data_frame.loc[0:3000,'value'])
plt.plot(test_data_frame['value'])

train_data_frame_final = train_data_frame.loc[0:3000,:]
test_data_frame_final = test_data_frame

data_scaler = StandardScaler()
data_scaler.fit(train_data_frame_final[['value']].values)
train_data = data_scaler.transform(train_data_frame_final[['value']].values)
test_data = data_scaler.transform(test_data_frame_final[['value']].values)

def create_dataset(dataset, look_back=64):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        dataX.append(dataset[i:(i+look_back),:])
        dataY.append(dataset[i+look_back,:])

    return np.array(dataX), np.array(dataY)

X_data, y_data = create_dataset(train_data, look_back=64) #look_back =
➥ window_size
X_train, X_val, y_train, y_val = train_test_split(X_data, y_data,
➥ test_size=0.1, random_state=42)
X_test, y_test = create_dataset(test_data, look_back=64)  #look_back =
➥ window_size

batch_size = 256
num_epochs = 32
timesteps = X_train.shape[1]
input_dim = X_train.shape[-1]
intermediate_dim = 16
latent_dim = 2
epsilon_std = 1.0

class Sampling(Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
```

Training params

Model params

Sampling layer

```
        dim = tf.shape(z_mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

class Likelihood(Layer):   <-- Likelihood layer
    def call(self, inputs):
        x, x_decoded_mean, x_decoded_scale = inputs
        dist = tfp.distributions.MultivariateNormalDiag(x_decoded_mean,
        ➥ x_decoded_scale)
        likelihood = dist.log_prob(x)
        return likelihood

#VAE architecture

#encoder
x = Input(shape=(timesteps, input_dim,))
h = LSTM(intermediate_dim)(x)

z_mean = Dense(latent_dim)(h)
z_log_sigma = Dense(latent_dim, activation='softplus')(h)

#sampling
z = Sampling()((z_mean, z_log_sigma))

#decoder
decoder_h = LSTM(intermediate_dim, return_sequences=True)
decoder_loc = LSTM(input_dim, return_sequences=True)
decoder_scale = LSTM(input_dim, activation='softplus', return_sequences=True)

h_decoded = RepeatVector(timesteps)(z)
h_decoded = decoder_h(h_decoded)

x_decoded_mean = decoder_loc(h_decoded)
x_decoded_scale = decoder_scale(h_decoded)

#log-likelihood
llh = Likelihood()([x, x_decoded_mean, x_decoded_scale])

vae = Model(inputs=x, outputs=llh)   <-- Defines the VAE model

# Add KL divergence regularization loss and likelihood loss
kl_loss = - 0.5 * K.mean(1 + z_log_sigma - K.square(z_mean) -
➥ K.exp(z_log_sigma))
tot_loss = -K.mean(llh - kl_loss)
vae.add_loss(tot_loss)

# Loss and optimizer.
loss_fn = tf.keras.losses.MeanSquaredError()
optimizer = tf.keras.optimizers.Adam()

@tf.function
def training_step(x):
    with tf.GradientTape() as tape:
        reconstructed = vae(x)  # Compute input reconstruction.
        # Compute loss.
```

```
            loss = 0 #loss_fn(x, reconstructed)
            loss += sum(vae.losses)
        # Update the weights of the VAE.
        grads = tape.gradient(loss, vae.trainable_weights)
        optimizer.apply_gradients(zip(grads, vae.trainable_weights))
        return loss

losses = []  # Keep track of the losses over time.
dataset = tf.data.Dataset.from_tensor_slices(X_train).batch(batch_size)
for epoch in range(num_epochs):
    for step, x in enumerate(dataset):
        loss = training_step(x)
        losses.append(float(loss))
    print("Epoch:", epoch, "Loss:", sum(losses) / len(losses))

plt.figure()
plt.plot(losses, c='b', lw=2.0, label='train')
plt.title('LSTM-VAE model')
plt.xlabel('Epochs')
plt.ylabel('Total Loss')
plt.legend(loc='upper right')
plt.show()

pred_test = vae.predict(X_test)

plt.plot(pred_test[:,0])

is_anomaly = pred_test[:,0] < -1e1
plt.figure()
plt.plot(test_data, color='b')
plt.figure()
plt.plot(is_anomaly, color='r')
```

We can see that for a simple square wave input in figure 11.4, we can detect a drop in log likelihood and threshold it to signal an anomaly. Figure 11.5 shows the decrease in total loss over epochs.

In the following section, we will explore amortized variational inference as applied to mixture density networks.

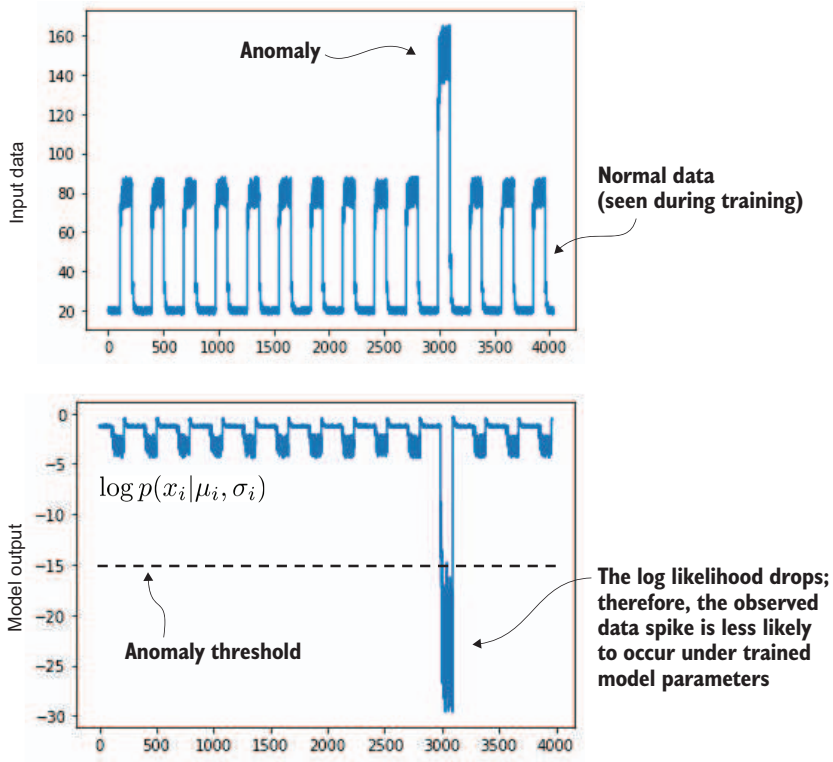


Figure 11.4 LSTM-VAE anomaly detection result

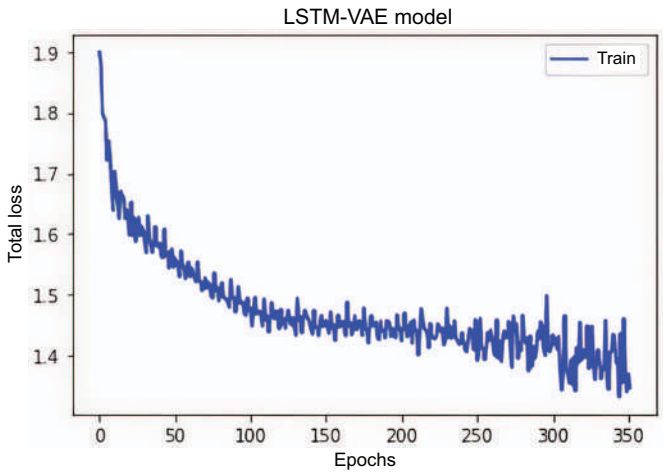
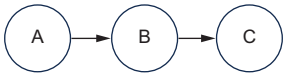


Figure 11.5 LSTM-VAE training loss

11.2 Amortized variational inference

Amortized *variational inference* (VI) is the idea that instead of optimizing a set of free parameters, we can introduce a parameterized function that maps from observation space to the parameters of the approximate posterior distribution. In practice, this could be a neural network that takes observations as input and outputs the mean and variance parameters for the latent variable associated with that observation (as we encountered in the VAE architecture). We can then optimize the parameters of this neural network instead of the individual parameters of each observation.

One advantage of amortized VI is memoized reuse of past inference, in a similar way to dynamic programming, in which we remember solutions to previously computed subproblems. For example, consider the following two queries in figure 11.6 (Samuel J. Gershman and Noah D. Goodman's "Amortized Inference in Probabilistic Reasoning," *Cognitive Science*, 2014).



Query 1: $P(B|C) = P(C|B)P(B)/P(C)$

Query 2: $P(A|C) = \sum_B P(A|B)P(B|C)$

Figure 11.6 A simple Bayesian network. Query 1 is a subquery of query 2.

We can see that query 1 is a subquery of query 2. Thus, the conditional distribution computed for query 1 can be reused to answer query 2. Another advantage of amortized VI is that it omits the requirement to derive the ELBO analytically, since the optimization takes place via stochastic gradient descent (SGD). The limitation of amortized VI is that the generalization gap depends on the capacity of the chosen neural network as the stochastic function.

Let's look at one example of amortized VI—namely, the mixture density network (MDN), in which we'll be using a multilayer perceptron (MLP) neural network to parameterize a Gaussian mixture model (GMM).

11.2.1 Mixture density networks

Mixture density networks (MDNs) are mixture models in which the parameters, such as means, covariances, and mixture proportions, are learned by a neural network. MDNs combine a structured data representation (a density mixture) with unstructured parameter inference (an MLP neural network). MDNs learn the mixture parameters by maximizing the log likelihood or, equivalently, minimizing a negative log likelihood loss.

Assuming a Gaussian mixture model (GMM) with K components, we can write down the probability of a test data point y_i conditioned on training data x as follows.

$$p(y_i|x) = \sum_{k=1}^{K} \pi_k(x) N\left(y_i|\mu_k(x), \Sigma_k(x)\right) \tag{11.4}$$

Here, the parameters μ_k, σ_k, π_k are learned by a neural network (e.g., an MLP) parameterized by θ.

$$\mu_k, \sigma_k, \pi_k = \mathrm{NN}(x; \theta) \tag{11.5}$$

The MDN architecture is captured in figure 11.7.

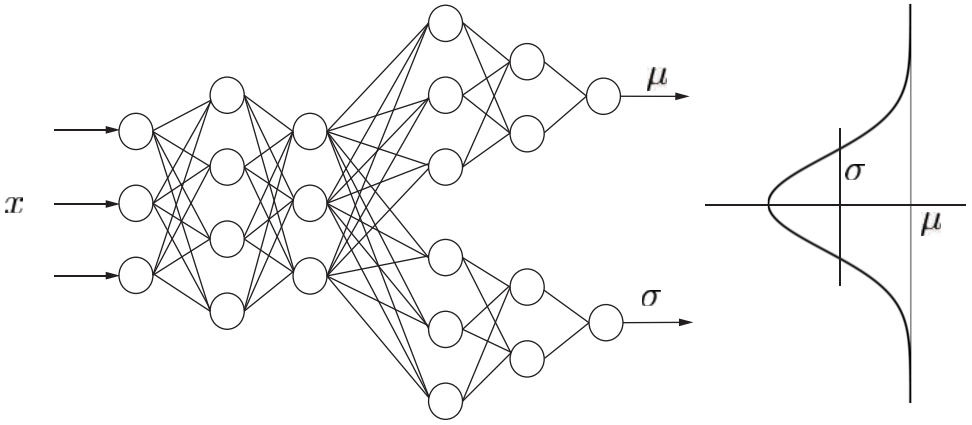


Figure 11.7 A multioutput neural network used to learn Gaussian parameters

As a result, the neural network (NN) is a multi-output model, subject to the following constraints on the output.

$$\begin{aligned} \forall k \ \sigma_k(x) &> 0 \\ \sum_{k=1}^{K} \pi_k(x) &= 1 \end{aligned} \tag{11.6}$$

In other words, we would like to enforce that the variance is strictly positive and the mixture weights add up to 1. The first constraint can be achieved using exponential activations, while the second constraint can be achieved using softmax activations. Finally, by making use of the iid assumption, we can attempt to minimize the following loss function.

$$\begin{aligned} \min_{\theta} L(\theta) &= \mathrm{NLLLoss}(\theta) = -\log \prod_{i=1}^{n} p(y_i|x) = -\sum_{i=1}^{n} \log p(y_i|x) \\ &= -\sum_{i=1}^{n} \log \left[\sum_{k=1}^{K} \pi_k(x_i, \theta) N(y_i | \mu_k(x_i, \theta), \Sigma_k(x_i, \theta)) \right] \end{aligned} \tag{11.7}$$

In our example, we assume an isotropic covariance $\Sigma_k = \sigma_k^2 I$; thus, we can write a d-dimensional Gaussian as a product. Given the multivariate Gaussian, we get the following.

$$N(y_i|\mu_k, \Sigma_k) = \frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma_k|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(y_i - \mu_k)^T \Sigma_k^{-1}(y_i - \mu_k)\right) \tag{11.8}$$

Since the covariance matrix is isotropic, we can rewrite equation 11.8 as follows.

$$\begin{aligned} N(y_i|\mu_k, \Sigma_k) &= \frac{1}{(2\pi\sigma_k^2)^{\frac{d}{2}}} \exp\left[-\frac{1}{2\sigma_k^2}\sum_{d=1}^{D}(y_{i,d} - \mu_{k,d})^2\right] \\ &= \prod_{d=1}^{D}\frac{1}{\sigma_k\sqrt{2\pi}} \exp\left[-\frac{1}{2\sigma_k^2}(y_{i,d} - \mu_{k,d})^2\right] \end{aligned} \tag{11.9}$$

Let's implement a Gaussian MDN using Keras/TensorFlow. We use synthetic data in our example with ground truth mean and covariance. We generate the data by sampling from a multivariate distribution. We then define the MDN multioutput architecture with constraints on mixing proportions and variance. Finally, we compute the negative log likelihood loss, train the model, and display the prediction results on test data. You may want to run the code listing in a Google Colab notebook (accessible at https://colab.research.google.com/) to understand the code step by step and accelerate model training via GPU.

Listing 11.2 Gaussian mixture density network

```
import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras

from keras.models import Model
from keras.layers import concatenate, Input
from keras.layers import Dense, Activation, Dropout, Flatten
from keras.layers import BatchNormalization

from keras import regularizers
from keras import backend as K
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
```

```
from sklearn.datasets import make_blobs
from sklearn.metrics import adjusted_rand_score
from sklearn.metrics import normalized_mutual_info_score
from sklearn.model_selection import train_test_split

import math
import matplotlib.pyplot as plt
import matplotlib.cm as cm

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)

def generate_data(N):
    pi = np.array([0.2, 0.4, 0.3, 0.1])
    mu = [[2,2], [-2,2], [-2,-2], [2,-2]]
    std = [[0.5,0.5], [1.0,1.0], [0.5,0.5], [1.0,1.0]]
    x = np.zeros((N,2), dtype=np.float32)
    y = np.zeros((N,2), dtype=np.float32)
    z = np.zeros((N,1), dtype=np.int32)
    for n in range(N):
        k = np.argmax(np.random.multinomial(1, pi))
        x[n,:] = np.random.multivariate_normal(mu[k], np.diag(std[k]))
        y[n,:] = mu[k]
        z[n,:] = k
    #end for
    z = z.flatten()
    return x, y, z, pi, mu, std

def tf_normal(y, mu, sigma):
    y_tile = K.stack([y]*num_clusters, axis=1) #[batch_size, K, D]
    result = y_tile - mu
    sigma_tile = K.stack([sigma]*data_dim, axis=-1) #[batch_size, K, D]
    result = result * 1.0/(sigma_tile+1e-8)
    result = -K.square(result)/2.0
    oneDivSqrtTwoPI = 1.0/math.sqrt(2*math.pi)
    result = K.exp(result) * (1.0/(sigma_tile + 1e-8))*oneDivSqrtTwoPI
    result = K.prod(result, axis=-1)    #[batch_size, K] iid Gaussians
    return result

def NLLLoss(y_true, y_pred):
    out_mu = y_pred[:,:num_clusters*data_dim]
    out_sigma = y_pred[:,num_clusters*data_dim : num_clusters*(data_dim+1)]
    out_pi = y_pred[:,num_clusters*(data_dim+1):]

    out_mu = K.reshape(out_mu, [-1, num_clusters, data_dim])

    result = tf_normal(y_true, out_mu, out_sigma)
```

Learning rate schedule

Synthetic ground truth data

Isotropic multivariate Gaussian

Negative log likelihood Loss

```
    result = result * out_pi
    result = K.sum(result, axis=1, keepdims=True)
    result = -K.log(result + 1e-8)
    result = K.mean(result)
    return tf.maximum(result, 0)

#generate data
X_data, y_data, z_data, pi_true, mu_true, sigma_true = generate_data(4096)

data_dim = X_data.shape[1]
num_clusters = len(mu_true)

num_train = 3500
X_train, X_test, y_train, y_test = X_data[:num_train,:],
➥ X_data[num_train:,:], y_data[:num_train,:], y_data[num_train:,:]
z_train, z_test = z_data[:num_train], z_data[num_train:]

#visualize data
plt.figure()
plt.scatter(X_train[:,0], X_train[:,1], c=z_train, cmap=cm.bwr)
plt.title('training data')
plt.show()
#plt.savefig(SAVE_PATH + '/mdn_training_data.png')

batch_size = 128
num_epochs = 128

hidden_size = 32
weight_decay = 1e-4

#MDN architecture
input_data = Input(shape=(data_dim,))
x = Dense(32, activation='relu')(input_data)
x = Dropout(0.2)(x)
x = BatchNormalization()(x)
x = Dense(32, activation='relu')(x)
x = Dropout(0.2)(x)
x = BatchNormalization()(x)

mu = Dense(num_clusters * data_dim, activation
➥ ='linear')(x)
sigma = Dense(num_clusters, activation=K.exp)(x)
pi = Dense(num_clusters, activation='softmax')(x)
out = concatenate([mu, sigma, pi], axis=-1)

model = Model(input_data, out)

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

model.summary()
```

Training params

Model params

Cluster means

Diagonal covariance

Mixture proportions

```
#define callbacks
file_name = SAVE_PATH + 'mdn-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_split=0.2,
➥ shuffle=True, verbose=2)          <-- Model training

y_pred = model.predict(X_test)          <-- Model evaluation

mu_pred = y_pred[:,:num_clusters*data_dim]
mu_pred = np.reshape(mu_pred, [-1, num_clusters, data_dim])
sigma_pred = y_pred[:,num_clusters*data_dim : num_clusters*(data_dim+1)]
pi_pred = y_pred[:,num_clusters*(data_dim+1):]
z_pred = np.argmax(pi_pred, axis=-1)

rand_score = adjusted_rand_score(z_test, z_pred)
print("adjusted rand score: ", rand_score)

nmi_score = normalized_mutual_info_score(z_test, z_pred)
print("normalized MI score: ", nmi_score)

mu_pred_list = []
sigma_pred_list = []
for label in np.unique(z_pred):
    z_idx = np.where(z_pred == label)[0]
    mu_pred_lbl = np.mean(mu_pred[z_idx,label,:], axis=0)
    mu_pred_list.append(mu_pred_lbl)

    sigma_pred_lbl = np.mean(sigma_pred[z_idx,label], axis=0)
    sigma_pred_list.append(sigma_pred_lbl)
#end for

print("true means:")
print(np.array(mu_true))

print("predicted means:")
print(np.array(mu_pred_list))

print("true sigmas:")
print(np.array(sigma_true))

print("predicted sigmas:")
print(np.array(sigma_pred_list))

#generate plots
plt.figure()
plt.scatter(X_test[:,0], X_test[:,1], c=z_pred, cmap=cm.bwr)
```

```
plt.scatter(np.array(mu_pred_list)[:,0], np.array(mu_pred_list)[:,1],
➥ s=100, marker='x', lw=4.0, color='k')
plt.title('test data')

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('Mixture Density Network')
plt.xlabel('Epochs')
plt.ylabel('Negative Log Likelihood Loss')
plt.legend(loc='upper left')
```

Figure 11.8 shows the cluster centroids overlayed with test data in addition to training and validation loss.

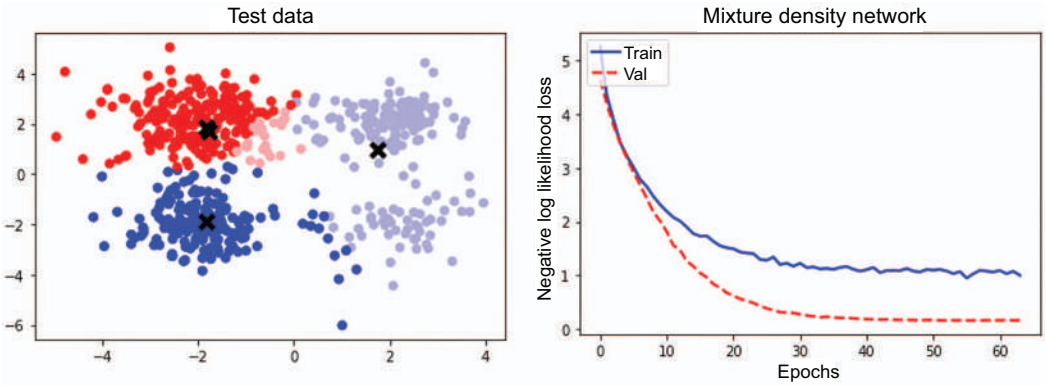


Figure 11.8 Mixture density network: cluster centroids and training and validation loss

We can see that inferred means are close to cluster centers. It's interesting to note that for this initialization, two of the cluster means coincide. Feel free to experiment with different seeds and numbers of training points to understand the behavior of the model. Also, we find that both training and validation loss are decreasing with the number of epochs. In the following section, we will look at a powerful transformer architecture based on self-supervised learning.

11.3 Attention and transformers

Attention allows a model to adaptively pay attention to different parts of the input by adjusting attention weights. Attention mechanisms can be applied to many kinds of neural networks but were first used in the context of recurrent neural networks (RNNs). In Seq2Seq RNN models, such as those used in machine translation, the output context vector that summarizes the input sentence does not have access to individual input words. This results in poor performance, as measured by the BLEU score. We can avoid this bottleneck by letting the output attend to input words directly, in a weighted fashion. In other words, we can compute the context vector as a weighted sum of input word vectors h_s^{enc}.

$$c_t = \sum_s a_{ts} h_s^{enc} \tag{11.10}$$

Here, a_{ts} are learned attention weights given by the following.

$$a_{ts} = \frac{\exp\left\{\text{score}\left(h_{t-1}^{dec}, h_s^{enc}\right)\right\}}{\sum_{s'} \exp\left\{\text{score}\left(h_{t-1}^{dec}, h_{s'}^{enc}\right)\right\}} \tag{11.11}$$

There are several ways to learn the scoring function (e.g., the multiplicative style).

$$\text{score}(a, b) = a^T W b \tag{11.12}$$

Here, W is an intermediate trainable matrix. We can generalize attention as a soft dictionary lookup. A soft dictionary lookup refers to a type of search in which an exact match is not found. This is useful when searching for words that may have been misspelled or are related to the search term. We can think of attention as comparing a set of target vectors or queries q_i with a set of candidate vectors or keys k_j. For each query, we compute how much the query is related to every key and then use these scores to weigh and sum the values v_j associated with every key. Thus, we can define the attention matrix A as follows.

$$A_{ij} = \text{score}(q_i, k_j) \tag{11.13}$$

Given the attention weights in A_{ij}, we compute a weighted combination of values v_j associated with each key. As a result, for the i-th query, we have the following.

$$r_i = \sum_j A_{ij} v_j = \sum_j \text{score}(q_i, k_j) v_j \tag{11.14}$$

Here, we can choose a normalized multiplicative score with W=1.

$$\text{score}(q_i, k_j) = \frac{q_i^T k_j}{\sqrt{D}} \tag{11.15}$$

In matrix notation, where we have N queries (Q of size $N \times D$) and N key–value pairs (K of size $N \times D$), we can write down the result (weighted set of values V, whose keys K most resemble the query Q).

$$\begin{aligned} R &= \text{attention}(Q, K, V) = AV \\ &= \text{score}(Q, K)V = \text{softmax}\left(\frac{QK^T}{\sqrt{D}}\right)V \end{aligned} \tag{11.16}$$

The softmax part of the product ensures the distribution adds up to 1 and can be thought of as where to look in the value matrix *V*.

Figure 11.9 shows the attention matrix heatmap for neural machine translation (NMT). In NMT, the query is the target sequence, while the keys and values are the source sequence. We can see which source English words the model pays attention to when producing a target translated sequence in Spanish.

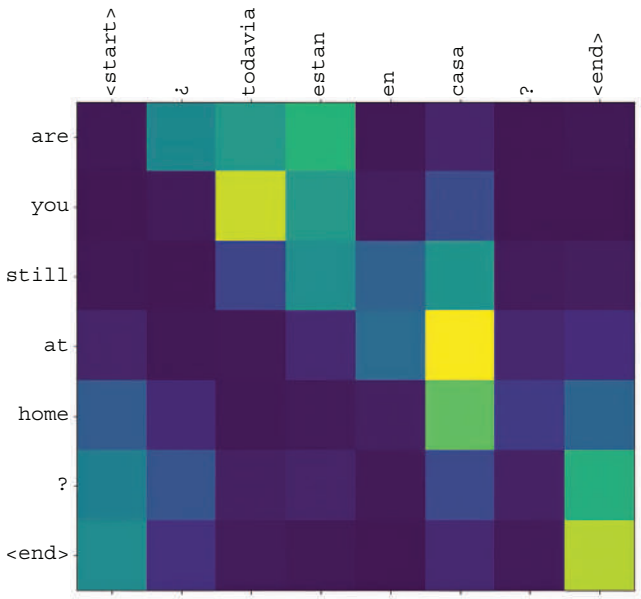


Figure 11.9 Attention matrix heatmap for neural machine translation

Self-attention is the key component of transformer architecture. A *transformer* is a Seq2Seq model that uses attention in the encoder as well as the decoder, thus eliminating the need for RNNs. Figure 11.10 shows the Transformer architecture (Vaswani et al.'s "Attention Is All You Need," NeurIPS, 2017). Let's look more closely at the building blocks of transfomers.

At a high level, we see an encoder–decoder architecture with inputs in the left branch and outputs in the right branch. We can identify multihead attention, positional encodings, dense and normalization layers, and residual connections for ease of gradient backpropagation.

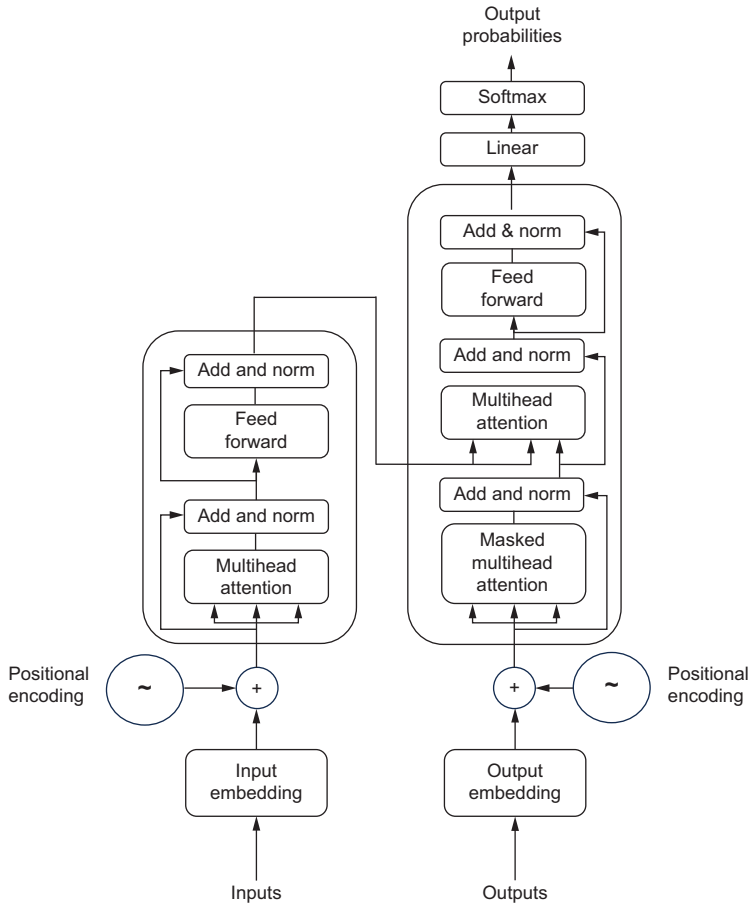


Figure 11.10 Transformer architecture

The idea of self-attention can be expanded to multihead attention. In essence, we run the attention mechanism in parallel and form several output heads, as shown in figure 11.11. The heads are then concatenated and transformed using a dense layer. With multihead attention, the model has multiple independent ways to understand the input.

For the model to make use of the order of the sequence, we need to inject some information about the position of tokens in the sequence. This is precisely the purpose of positional encodings that get added to the input embeddings at the bottom of the encoder and decoder stacks. The positional encodings have the same dimensions as the embeddings so that the two can be summed. Thus,

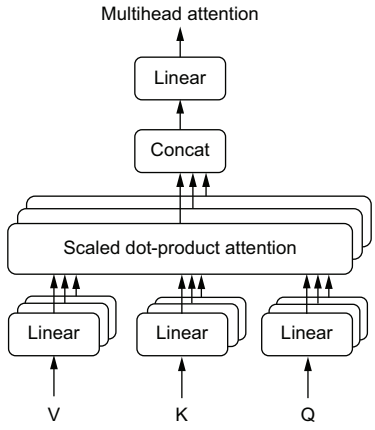


Figure 11.11 Multihead attention

if the same word appears in a different position, the actual word representation will be different, depending on where it appears in the sentence.

Finally, layer normalization is used to normalize the input by computing mean and variance across channels and spatial dimensions and a linear (dense) layer is added to complete the encoder. The linear layer comes after multihead self-attention to project the representation in higher dimensional space and then back to the original space. This helps solve stability issues and counters bad initializations.

If we look closely at the decoder, we'll notice it contains all of the earlier components, in addition to a masked multihead self-attention layer and a new multihead attention layer, known as *encoder-decoder attention*. The final output of the decoder is transformed through a final linear layer, and the output probabilities that predict the next token in the output sequence are calculated with the standard softmax function.

The purpose of masked attention is to respect causality when generating the output sentence. Since the entire sentence is not yet available and is generated one token at a time, we mask the output by introducing a mask matrix `M` that contains only two types of values: zero and negative infinity.

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T + M}{\sqrt{D}}\right)V \tag{11.17}$$

Eventually, the zeros will be transformed into ones via softmax, while negative infinities will become zero, effectively removing the corresponding connection from the output.

The encoder–decoder attention is simply the multihead self-attention we are already familiar with, except the query Q comes from a different source than the keys K and values V. This attention is also known in the literature as *cross-attention*. Remember that in the machine translation example, our target sequence or query Q comes from the decoder, while the encoder acts like a database and provides us with keys K and values V. The intuition behind the encoder–decoder attention layer is to combine the input and output sentences. Thus, the encoder–decoder attention is trained to associate the input sentence with the corresponding output word, determining how related each target word is with respect to the input words.

And that's it! We are now ready to implement a transformer-based classifier from scratch! You may want to run the code listing in a Google Colab notebook (accessible at https://colab.research.google.com/) to understand the code step by step and accelerate model training via GPU.

Listing 11.3 Transformer for text classification

```
import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras
```

```
from keras.models import Model, Sequential
from keras.layers import Layer, Dense, Dropout, Activation
from keras.layers import LayerNormalization, MultiHeadAttention
from keras.layers import Input, Embedding, GlobalAveragePooling1D

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

#load dataset
max_words = 20000
seq_len = 200
(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)

class TransformerBlock(Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super(TransformerBlock, self).__init__()
        self.att = MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
        self.ffn = Sequential(
            [Dense(ff_dim, activation="relu"), Dense(embed_dim)]
        )
        self.layernorm1 = LayerNormalization(epsilon=1e-6)
        self.layernorm2 = LayerNormalization(epsilon=1e-6)
        self.dropout1 = Dropout(rate)
        self.dropout2 = Dropout(rate)

    def call(self, inputs, training):
        attn_output = self.att(inputs, inputs)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)

class TokenAndPositionEmbedding(Layer):
```

Learning rate schedule

Top 20,000 most frequent words

First 200 words of each movie review

```
    def __init__(self, maxlen, vocab_size, embed_dim):
        super(TokenAndPositionEmbedding, self).__init__()
        self.token_emb = Embedding(input_dim=vocab_size,
        ➥ output_dim=embed_dim)
        self.pos_emb = Embedding(input_dim=maxlen, output_dim=embed_dim)

    def call(self, x):
        maxlen = tf.shape(x)[-1]
        positions = tf.range(start=0, limit=maxlen, delta=1)
        positions = self.pos_emb(positions)
        x = self.token_emb(x)
        return x + positions

batch_size = 32
num_epochs = 8          Training params

embed_dim = 32
num_heads = 2           Model parameters
ff_dim = 32

#transformer architecture
inputs = Input(shape=(seq_len,))
embedding_layer = TokenAndPositionEmbedding(seq_len, max_words, embed_dim)
x = embedding_layer(inputs)
transformer_block = TransformerBlock(embed_dim, num_heads, ff_dim)
x = transformer_block(x)
x = GlobalAveragePooling1D()(x)
x = Dropout(0.1)(x)
x = Dense(20, activation="relu")(x)
x = Dropout(0.1)(x)
outputs = Dense(2, activation="softmax")(x)

model = Model(inputs=inputs, outputs=outputs)

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

#define callbacks
file_name = SAVE_PATH + 'transformer-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 = [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 evaluation

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

```
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('Transformer 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('Transformer 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('Transformer model')
plt.xlabel('Epochs')
plt.ylabel('Learning Rate')
plt.legend()
plt.show()
```

From figure 11.12, we can see early signs of overfitting. By preserving the model with lowest validation loss, we avoid the problem of overfitting.

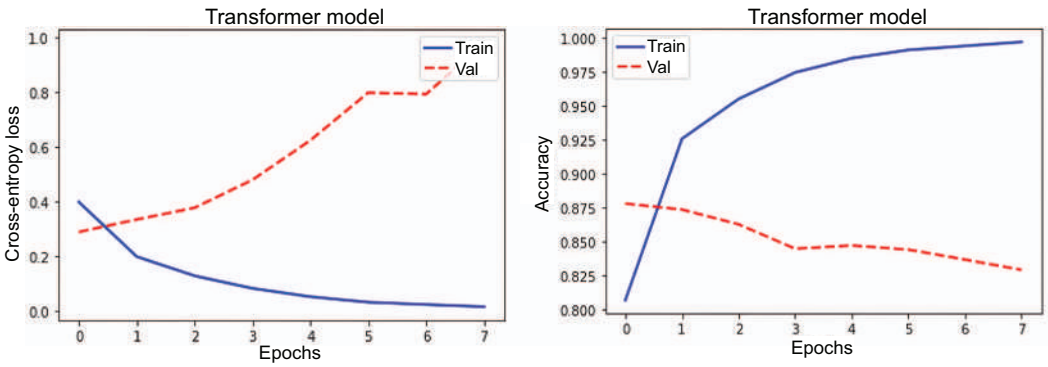


Figure 11.12 Transformer classifier loss (left) and accuracy (right) on training and validation datasets

In this section, we observed how self-attention can be used inside the transformer architecture to learn text sentiment. In the following section, we will study neural networks in application to graph data.

11.4 Graph neural networks

A wide variety of types of information can be represented by graphs. Some examples of graphs include knowledge graphs, social networks, molecular structures, and document citation networks. Graph neural networks (GNNs) operate on graphs and relational data. In this section, we'll study graph convolutional networks (GCNs) for classifying nodes in the CORA citation network dataset. But first, let's look at GNN fundamentals.

Let $G=(V,E)$ be our graph with vertices V and edges E. We can construct an adjacency matrix A to capture the interconnection of edges, such that $A_{ij} = 1$ if an edge exists between nodes i and j and 0 otherwise. For undirected graphs, we have a symmetric adjacency matrix such that $A=A^T$. Another matrix that will prove useful in training GNNs is the node feature matrix `X`. Assuming we have N nodes and F features per node, the size of X is $N\times F$.

As an example, in a CORA dataset, each node is a document and each edge is a citation that forms a directed edge between two nodes. We can capture the citation relationships in the adjacency matrix `A`. Since each document is a collection of words, we can introduce indicator features that tell us whether a particular dictionary word is present or absent from the document. In this case, N will be the number of documents and F will be the dictionary size. Thus, we can represent the text information in each document via a binary $N\times F$ matrix X.

It's important to note that edges can have their own features as well. In this case, if the size of edge features is S and the number of edges is M, we can construct an edge feature matrix E of size $M\times S$. It's also important to make a distinction between classifying the entire graph as a whole (e.g., as in molecule classification) and classifying nodes within a graph (e.g., as in the CORA citation network). The former method is called *batch mode classification*, while the latter is referred to as *single mode*.

It's interesting to draw a parallel between GCNs and CNNs. Images can also be seen as graphs, albeit with regular structure. For example, each node can represent a pixel, the node feature can represent the pixel value, and the edge feature can represent the Euclidean distance between each pixel in a complete graph. In this light, GCNs can be seen as a generalization of CNNs, since they operate on arbitrarily connected graphs.

We can think of information propagation in a spectral GCN as signal propagation along the nodes. Spectral GCN uses Eigen decomposition of a graph Laplacian matrix to propagate the signal with key forward equation as follows.

$$X' = D^{-\frac{1}{2}}(A+I)D^{-\frac{1}{2}}XW + b \tag{11.18}$$

Let's understand this equation step by step. If we take adjacency matrix A and multiply it by the feature matrix X, the product AX represents the sum of neighboring node features. However, this sum is over all the neighboring nodes except the node itself. To fix this, we add a self-loop in the adjacency matrix by summing it with an

identity. This brings us to $(A + I)X$; however, this product is not normalized. To do so, we will divide by the degree of each node. Thus, we form a diagonal degree matrix D, pre and post multiplying our expression by the square root of D. Next, we add the familiar product by the learnable weight `W` and a bias term `b`. We wrap this in a nonlinearity, and voila! We have our forward GCN equation.

We are now ready to implement our graph neural network using the Spektral Keras/Tensorflow library. In listing 11.4, we begin by importing the dataset that can be found in the data folder of the code repository. We prepare the data for processing and define graph neural network architecture. We proceed by training the model and displaying the results. You may want to run the code listing in a Google Colab notebook (accessible at https://colab.research.google.com/) to understand the code step by step and accelerate model training via GPU.

Listing 11.4 Graph convolutional neural network for classifying citation graphs

```
import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras

import networkx as nx
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import LabelEncoder
from sklearn.utils import shuffle
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split

from spektral.layers import GCNConv

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dropout, Dense
from tensorflow.keras import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import TensorBoard, EarlyStopping
from tensorflow.keras.regularizers import l2

import os
from collections import Counter
from sklearn.manifold import TSNE
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/cora/"

column_names = ["paper_id"] + [f"term_{idx}" for idx in range(1433)] +
➥ ["subject"]
node_df = pd.read_csv(DATA_PATH + "cora.content", sep="\t", header=None,
➥ names=column_names)
```

```
print("Node df shape:", node_df.shape)

edge_df = pd.read_csv(DATA_PATH + "cora.cites", sep="\t", header=None,
➥ names=["target", "source"])
print("Edge df shape:", edge_df.shape)

nodes = node_df.iloc[:,0].tolist()
labels = node_df.iloc[:,-1].tolist()
X = node_df.iloc[:,1:-1].values

X = np.array(X,dtype=int)
N = X.shape[0] #the number of nodes
F = X.shape[1] #the size of node features

edge_list = [(x, y) for x, y in zip(edge_df['target'],
➥ edge_df['source'])]

num_classes = len(set(labels))

print('Number of nodes:', N)
print('Number of features of each node:', F)
print('Labels:', set(labels))
print('Number of classes:', num_classes)

def sample_data(labels, limit=20, val_num=500, test_num=1000):
    label_counter = dict((l, 0) for l in labels)
    train_idx = []

    for i in range(len(labels)):
        label = labels[i]
        if label_counter[label]<limit:
            #add the example to the training data
            train_idx.append(i)
            label_counter[label]+=1

        #exit the loop once we found 20 examples for each class
        if all(count == limit for count in label_counter.values()):
            break

    #get the indices that do not go to traning data
    rest_idx = [x for x in range(len(labels)) if x not in train_idx]
    #get the first val_num
    val_idx = rest_idx[:val_num]
    test_idx = rest_idx[val_num:(val_num+test_num)]
    return train_idx, val_idx,test_idx


train_idx,val_idx,test_idx = sample_data(labels)

#set the mask
train_mask = np.zeros((N,),dtype=bool)
train_mask[train_idx] = True

val_mask = np.zeros((N,),dtype=bool)
val_mask[val_idx] = True
```

Parses node data

Parses edge data

```
test_mask = np.zeros((N,),dtype=bool)
test_mask[test_idx] = True

print("Training data distribution:\n{}".format(Counter([labels[i] for i in
➥ train_idx])))
print("Validation data distribution:\n{}".format(Counter([labels[i] for i
➥ in val_idx])))

def encode_label(labels):
    label_encoder = LabelEncoder()
    labels = label_encoder.fit_transform(labels)
    labels = to_categorical(labels)
    return labels, label_encoder.classes_

labels_encoded, classes = encode_label(labels)

G = nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edge_list)

A = nx.adjacency_matrix(G)
print('Graph info: ', nx.info(G))

# Parameters
channels = 16
dropout = 0.5
l2_reg = 5e-4
learning_rate = 1e-2
epochs = 200
es_patience = 10

# Preprocessing operations
A = GCNConv.preprocess(A).astype('f4')

# Model definition
X_in = Input(shape=(F, ))
fltr_in = Input((N, ), sparse=True)

dropout_1 = Dropout(dropout)(X_in)
graph_conv_1 = GCNConv(channels,
                       activation='relu',
                       kernel_regularizer=l2(l2_reg),
                       use_bias=False)([dropout_1, fltr_in])

dropout_2 = Dropout(dropout)(graph_conv_1)
graph_conv_2 = GCNConv(num_classes,
                       activation='softmax',
                       use_bias=False)([dropout_2, fltr_in])

# Build model
model = Model(inputs=[X_in, fltr_in], outputs=graph_conv_2)
model.compile(optimizer=Adam(learning_rate=learning_rate),
              loss='categorical_crossentropy',
              weighted_metrics=['accuracy'])
```

Builds the graph

Obtains the adjacency matrix

Number of channels in the first layer

Dropout rate for the features

Learning rate

L2 regularization rate

Number of training epochs

Patience for early stopping

```
model.summary()

# Train model
validation_data = ([X, A], labels_encoded, val_mask)
hist = model.fit([X, A],
          labels_encoded,
          sample_weight=train_mask,
          epochs=epochs,
          batch_size=N,
          validation_data=validation_data,
          shuffle=False,
          callbacks=[
              EarlyStopping(patience=es_patience,  restore_best_weights=True)
          ])

# Evaluate model
X_test = X
A_test = A
y_test = labels_encoded

y_pred = model.predict([X_test, A_test], batch_size=N)
report = classification_report(np.argmax(y_test,axis=1),
➥ np.argmax(y_pred,axis=1), target_names=classes)
print('GCN Classification Report: \n {}'.format(report))

layer_outputs = [layer.output for layer in model.layers]
activation_model = Model(inputs=model.input, outputs=layer_outputs)
activations = activation_model.predict([X,A],batch_size=N)

#Get t-SNE Representation
#get the hidden layer representation after the first GCN layer
x_tsne = TSNE(n_components=2).fit_transform(activations[3])

def plot_tSNE(labels_encoded,x_tsne):
    color_map = np.argmax(labels_encoded, axis=1)
    plt.figure(figsize=(10,10))
    for cl in range(num_classes):
        indices = np.where(color_map==cl)
        indices = indices[0]
        plt.scatter(x_tsne[indices,0], x_tsne[indices, 1], label=cl)
    plt.legend()
    plt.show()

plot_tSNE(labels_encoded,x_tsne)

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('GNN model')
plt.xlabel('Epochs')
plt.ylabel('Cross-Entropy Loss')
plt.legend(loc='upper right')
plt.show()

plt.figure()
```

Model training

Model evaluation

```
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('GNN model')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')
plt.show()
```

Figure 11.13 shows classification loss and accuracy for training and validation datasets.

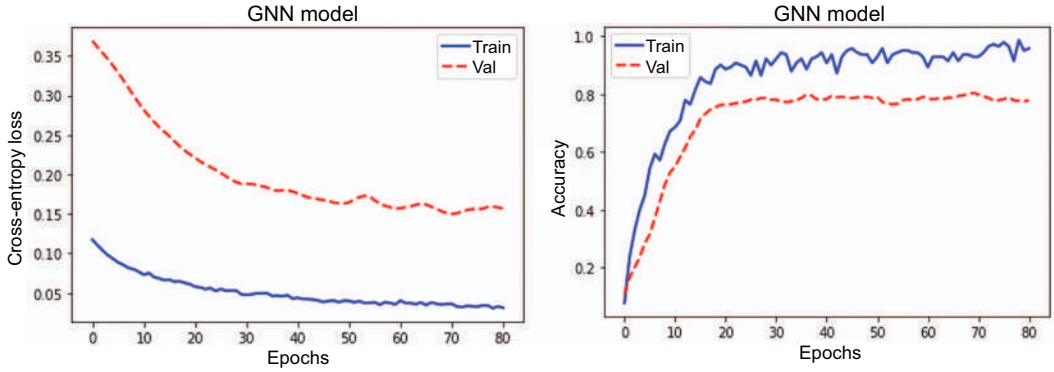


Figure 11.13 GNN model loss and accuracy for training and validation datasets

Figure 11.14 shows the t-SNE hidden layer representation of the CORA dataset embedding after the first GCN layer.

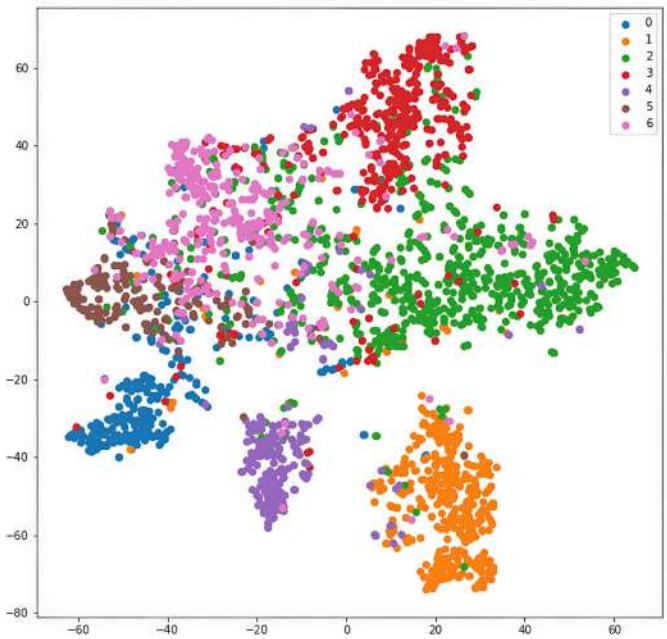


Figure 11.14 t-SNE hidden layer representation of the CORA dataset embedding after the first GCN layer

In the following section, we will review deep learning research in the area of computer vision and natural language processing.

11.5 ML research: Deep learning

In the area of computer vision (CV), the CNN architecture has evolved from convolutional, ReLU, and max pooling operations of LeNet (Yann LeCun et al.'s "Gradient-Based Learning Applied to Document Recognition," In Proceedings of the IEEE, 1998) to inception modules of GoogLeNet (Christian Szegedy et al.'s "Going Deeper with Convolutions," *Computer Vision and Pattern Recognition*, 2015). The inception module creates variable receptive fields by employing different kernel sizes. This, in turn, allows the capturing of sparse correlation patterns in the new feature map stack. GoogLeNet offers high accuracy on the ImageNet dataset with fewer parameters than AlexNet (Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton's "ImageNet Classification with Deep Convolutional Neural Networks," Conference on Neural Information Processing Systems, 2012) or VGG (Karen Simonyan and Andrew Zisserman's "Very Deep Convolutional Networks for Large-Scale Image Recognition," International Conference on Learning Representations, 2015). We saw the introduction of residual connections in the ResNet model (Kaiming He et al.'s "Deep Residual Learning for Image Recognition," Conference on Computer Vision and Pattern Recognition, 2015), which became the standard architectural choice for modern neural networks of arbitrary depth. Fast-forwarding to ision transformer (ViT) model (Alexey Dosovitskiy et al.'s "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," International Conference on Learning Representations, 2021) employs a transformer-like architecture over patches of the image. The image is divided into fixed-size patches, each of which being linearly and positionally embedded and the resulting sequence of vectors fed to a standard transformer encoder followed by a MLP head for classification.

In the area of generative models, generative adversarial networks (GANs), introduced by Ian Goodfellow et al in "Generative Adversarial Networks" (Conference and Workshop on Neural Information Processing Systems, 2014), draw on the ideas from game theory and use two networks trained to approximate the distribution of the data: a generator to generate images and a discriminator to discriminate between real and fake images. On the other hand, likelihood-based generative models model the distribution of the data, using a likelihood function. The most popular subclass of likelihood-based generative models is the variational autoencoder (VAE), introduced by Diederik P. Kingma and Max Welling in "Auto-Encoding Variational Bayes" (International Conference on Learning Representations, 2014). A VAE consists of an encoder that takes in data as input, transforming it into a latent representation, and a decoder that takes a latent representation and returns a reconstruction. A beta-VAE is a modification of VAE with a special emphasis on discovering disentangled latent factors (Irina Higgins et al.'s "beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework," International Conference on Learning Representations, 2017).

Shifting the spotlight to diffusion models, which gained popularity for their ability to generate high resolution images, are parametrized Markov chains trained using variational inference to produce samples matching the input data. Diffusion models consist of a forward pass, in which an image gets progressively noisier via additive Gaussian noise, and a reverse pass, in which the noise is transformed back into a sample from the target distribution (Jonathan Ho, Ajay Jain, and Pieter Abbeel's "Denoising Diffusion Probabilistic Models," Conference and Workshop on Neural Information Processing Systems, 2020).

In the area of natural language processing (NLP), the architecture for machine translation, for example, has evolved from Seq2Seq LSTM models to transformer-based models, such as BERT (Jacob Devlin et al.'s "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," North American Chapter of the Association for Computational Linguistics, 2019). The pretrained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architectural modifications. Zero-shot and few-shot learning gained popularity with large Transformer models, such as GPT-3, introduced by Tom B. Brown et al. in "Language Models are Few-Shot Learners" (Conference and Workshop on Neural Information Processing Systems, 2020). The applications of GPT-3 range from text completion to code completion. For example, OpenAI's codex model, which is a descendant of the GPT-3 series that's trained on natural language as well as billions of lines of code, serves as an AI pair programmer that can turn comments into code and complete the next line or function in context.

A number of learning tasks require processing graph data that contains rich relational information, from modeling social networks to predicting protein structure calls for a model that can work with graph data. Graph neural networks (GNNs) are neural models that capture dependence relationships via message passing between the nodes of the graph (Thomas N. Kipf and Max Welling's "Semi-Supervised Classification with Graph Convolutional Networks," International Conference on Learning Representations, 2017). In recent years, variants of GNNs, such as graph convolutional networks (GCNs), graph attention networks (GATs), and graph recurrent networks (GRNs), have demonstrated groundbreaking performance on many deep learning tasks.

Finally, amortized variational inference (D. P. Kingma's "Variational Inference and Deep Learning: A New Synthesis," University of Amsterdam, 2017) is an interesting research area, as it combines the expressive power and representation learning of deep neural networks with domain knowledge of probabilistic graphical models. We saw one such application of MDNs when we used a neural network to map from the observation space to the parameters of the approximate posterior distribution.

Deep learning research is a very active field. For state-of-the-art developments and applications, the reader is encouraged to review the Conference and Workshop on Neural Information Processing Systems, the International Conference on Learning Representations, and the research conferences mentioned in appendix A.

11.6 Exercises

11.1 What is a receptive field in a CNN?

11.2 Explain the benefit of residual connections by deriving the backward pass.

11.3 Compare the pros and cons of using CNNs, RNNs, and transformer neural networks.

11.4 Give an example of a neural network that uses amortized variational inference.

11.5 Show via an example the intuition behind the GCN forward equation: $D^{-1/2}(A+I)D^{-1/2}XW+b$.

Summary

- Autoencoders are unsupervised neural networks trained to reconstruct the input. The bottleneck layer of the autoencoder can be used for dimensionality reduction or feature extraction.
- A variational autoencoder consists of an encoder that takes in data as input, transforming it into a latent representation, and a decoder that takes a latent representation, returning a reconstruction.
- Amortized variational inference is the idea that instead of optimizing a set of free parameters, we can introduce a parameterized function that maps from observation space to the parameters of the approximate posterior distribution.
- Mixture density networks are mixture models in which the parameters, such as means, covariances, and mixture proportions, are learned by a neural network.
- Attention allows a model to adaptively focus on different parts of the input by adjusting the attention weights.
- A transformer is a Seq2Seq model that uses attention in the encoder as well as the decoder. Self-attention is the key component of transformer architecture.
- Graph neural networks are neural models that capture dependence relationships via passing messages between the nodes of the graph.