APPENDIX A

Training DNN with Keras

This appendix will discuss using the Keras framework to train deep learning and explore some example applications on image segmentation using a fully convolutional network (FCN) and click-rate prediction with a wide and deep model (inspired by the TensorFlow implementation).

Despite their massive size, successful deep artificial neural networks can exhibit a remarkably small difference between training and test performance; see https://blog.acolyer.org/2017/05/11/understanding-deep-learning-requires-re-thinking-generalization/. In a blog post (https://beamandrew.github.io/deeplearning/2017/06/04/deep_learning_works.html), Andrew Beam explains why it's possible to apply very large neural networks even if you have small data sets without the risk of overfitting.

A.1 The Keras Framework

Keras.io is an excellent framework to start deploying a deep learning model. The author, Francois Chollet, has created a great library, following a minimalist approach and with many hyperparameters and optimizers already preconfigured. You can run complex models in less than ten lines of code using Theano, TensorFlow, and CNTK backends.

A. Vieira and B. Ribeiro, *Introduction to Deep Learning Business Applications for Developers*,
https://doi.org/10.1007/978-1-4842-3453-2

A.1.1 Installing Keras in Linux

Keras is pretty straightforward to install. The first step is to install Theano or TensorFlow. Installing TensorFlow is easy with Pip. Be careful with the version you install, though. If you use a GPU, you have to choose a compatible installation that will run Cuda. There are some obvious dependencies like Numpy or less obvious ones like hdf5 to compress files. See the full instructions for a Linux installation at www.pyimagesearch.com/2016/11/14/installing-keras-with-tensorflow-backend/.

A.1.2 Model

Models in Keras are defined as a sequence of layers. A network is a stack of layers forming a network topology. The input layer needs to have the same dimensions as the input data. This can be specified when creating the first layer with the *input_dim* argument.

Finding the best network architecture (number of layers, size of layers, activation functions) is done mostly by trial and error. Generally, you need a network large enough to accommodate the complexity of the problem but one that is not too complex.

Fully connected layers are defined using the `Dense` class. You can specify the number of neurons in the layer as the first argument.

The network weights should be initialized to a small random number generated from a uniform distribution. The initialization method can be specified as an `int` argument. The activation function is also specified as an argument. If you are unsure about these initializations, simply use the defaults.

A.1.3 The Core Layers

A neural network is composed of a set of (mostly sequential) layers that are connected with each other. These are the most common layers:

- Input
- Dense
- Convolution1D and convolution2D
- Embedding
- LSTM

A neural network works with tensors. Before you perform computation, you need to convert your data (as a Numpy array of a Pandas data frame) into a tensor. The input layer is the entry point of a neural network.

The dense layer is the most basic (and common) type of layer. It has as arguments the number of unities and the activation function. The rectifier linear unit (ReLU) activation function is the most common one. The convolution layers (1D or 2D) are mostly used for text and images and the required parameters are the number of filters and the kernel size. The embedding layer is very useful for text data as they can convert a very high dimensional data into a denser representation - they require two parameters `input_dim` and `output_dim`. The LSTM layer is very useful to learn temporal or sequential data - the only required parameter is the number of units - careful since these networks with these layers are very computational intensive and they overfit easily.

Some other common activation functions are tanh, softmax, and argmax.

The following is a simple example of a Keras model to classify data (the response variable is the last column of the file `xxx.csv`, either 0 or 1). In this example, you will train a classifier, minimize the cross entropy over 150 epochs, and print the predictions. The data is assumed to be normalized. As the activation function in the last layer, you are using `sigmoid`, but

normally `softmax` should be used. It is assumed that input data is contained in the initial `X_dim` columns - parameter that should be provided.

```
from keras.models import Sequential
from keras.layers import Dense
import numpy as np
# load a dataset
dataset = np.loadtxt("xxx.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:X_dim]
Y = dataset[:,X_dim]
# create model
model = Sequential()
model.add(Dense(12, input_dim=X_dim, init='uniform',
activation='relu'))
model.add(Dense(5, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam',
    metrics=['accuracy'])
# Fit the model
model.fit(X, Y, epochs=150, batch_size=10, verbose=2)
# calculate predictions
predictions = model.predict(X)
# round predictions
rounded = [round(x[0]) for x in predictions]
print(rounded)
```

A.1.4 The Loss Function

Keras comes with the most common loss functions, including these basic ones:

- Cross entropy and binary cross entropy for classification problems
- Categorical cross entropy
- Mean Square Error (MSE) for regression problems

Building a personalized loss function is quite straightforward. An example is provided in the code of the FCN later in this chapter to weight the cross entropy to account for imbalanced categorical data, using the `binary_crossentropy_2d_w()` function. Care should be taken because loss functions have to be fully differentiable. For instance, you cannot use `if`, `then`, `else`.

A.1.5 Training and Testing

Normally you specify the metrics of interest by calling the `compile` method. For instance, you can compile this model using the Adam optimizer with a learning rate of 0.001, minimizing the binary cross entropy loss and displaying the accuracy.

```
model.compile(Adam(0.001), loss='binary_crossentropy',
metrics='accuracy')
```

To display all metrics from training a model, just use this:

```
history=model.fit(X_train,Y_train,epochs=50)
print(history.history.keys())
```

A.1.6 Callbacks

Keras can register a set of callbacks when training neural networks.

The default callback tracks the training metrics for each epoch, including the loss and the accuracy for training and validation data.

An object named `history` is returned from a call to the `fit()` function. Metrics are stored in the form of a dictionary in the history member of the object returned.

The following is an example using a checkpoint to save the weights (in the file `weights.hdf5`) of the best model:

```
from keras.callbacks import ModelCheckpoint
checkpointbest = ModelCheckpoint(filepath='weights.hdf5',
verbose=1, save_best_only=True)
model.fit(x_train, y_train, epochs=20, validation_data=
(x_test, y_test), callbacks=[checkpointbest])
```

A.1.7 Compile and Fit

After the model is defined, it can be compiled; only at this point is the computational graph effectively generated. Compiling uses the numerical libraries from the Keras backend such as Theano or TensorFlow. The backend automatically chooses the best way to represent the network for training and makes predictions for running on hardware, such as a CPU or GPU and single or multiple. You can run models on a CPU, but a GPU is advisable if you are dealing with large image data sets because it will speed up the training by an order of magnitude.

Compiling requires additional properties for training the network for finding the best set of weights connecting the neurons. You must specify the loss function to use to evaluate the network, the optimizer used to search through different weights for the network, and any optional metrics you would like to collect and report during training.

For classification, you typically use logarithmic loss, which for a binary classification problem is defined in Keras as `binary_crossentropy`. For optimization, the gradient descent algorithm `adam` is commonly used.

```
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics ['accuracy'])
```

Other common optimizers include Adadelta, SGD, and Adagrad.

To train, or fit, the model on data, you call the `fit()` function on the model. The training process will run for a fixed number of iterations through the data set called *epochs*, which is specified through the `epochs` argument. You can also set the number of instances that are evaluated before a weight update in the network is performed, called the *batch size*, using the `batch_size` argument.

A.2 The Deep and Wide Model

Wide and deep models can be jointly trained using linear models and deep neural networks. The wide component consists of a generalized linear model, and the cross-product interaction is modeled as a neural network with embedding layers (see Figure A-1).

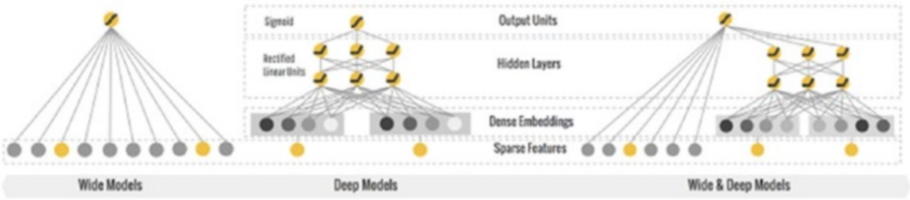

Figure A-1. *Wide and deep neural network model*

The following code, in Python 2.7, is the Keras implementation of the code originally presented in TensorFlow. To run it, you need to download the adult data set from `http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/adult.data`. It was provided by Javier Zaurin (`https://github.com/jrzaurin/Wide-and-Deep-Keras`).

First you will do the imports and define some functions to be used later.

```
# to run : python wide_and_deep.py –method method
# example: python wide_and_deep.py –method deep
import numpy as np
import pandas as pd
import argparse
from sklearn.preprocessing import StandardScaler
from copy import copy

from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from keras.layers import Input, concatenate, Embedding,
Reshape, Merge, Flatten, merge, Lambda
from keras.layers.normalization import BatchNormalization
from keras.models import Model
from keras.regularizers import l2, l1_l2

def cross_columns(x_cols):
    """simple helper to build the crossed columns in a pandas
    dataframe
    """
    crossed_columns = dict()
    colnames = ['_'.join(x_c) for x_c in x_cols]
    for cname,x_c in zip(colnames,x_cols):
        crossed_columns[cname] = x_c
    return crossed_columns

def val2idx(DF_deep,cols):
    """helper to index categorical columns before embeddings.
    """ DF_deep = pd.concat([df_train, df_test])
    val_types = dict()
    for c in cols:
```

```
        val_types[c] = DF_deep[c].unique()

    val_to_idx = dict()
    for k, v in val_types.iteritems():
        val_to_idx[k] = o: i for i, o in enumerate(val_
        types[k])

    for k, v in val_to_idx.iteritems():
        DF_deep[k] = DF_deep[k].apply(lambda x: v[x])

    unique_vals = dict()
    for c in cols:
        unique_vals[c] = DF_deep[c].nunique()

    return DF_deep,unique_vals

def embedding_input(name, n_in, n_out, reg):
    inp = Input(shape=(1,), dtype='int64', name=name)
    return inp, Embedding(n_in, n_out, input_length=1,
        embeddings_regularizer=l2(reg))(inp)

def continous_input(name):
    inp = Input(shape=(1,), dtype='float32', name=name)
    return inp, Reshape((1, 1))(inp)
```

Then you define the wide model.

```
def wide():

    target = 'cr'

    wide_cols = ["gender", "xyz_campaign_id", "fb_campaign_id",
    "age", "interest"]
    x_cols = (['gender', 'age'],['age', 'interest'])

    DF_wide = pd.concat([df_train,df_test])
```

```
# my understanding on how to replicate what layers.crossed_
column does One
# can read here: https://www.tensorflow.org/tutorials/linear.
crossed_columns_d = cross_columns(x_cols)

categorical_columns =
    list(DF_wide.select_dtypes(include=['object']).columns)
wide_columns = wide_cols + crossed_columns_d.keys()

for k, v in crossed_columns_d.iteritems():
    DF_wide[k] = DF_wide[v].apply(lambda x: '-'.join(x),
    axis=1)

DF_wide = DF_wide[wide_columns + [target] + ['IS_TRAIN']]
dummy_cols = [
    c for c in wide_columns if c in categorical_columns +
        crossed_columns_d.keys()]
DF_wide = pd.get_dummies(DF_wide, columns=[x for x in
dummy_cols])

train = DF_wide[DF_wide.IS_TRAIN == 1].drop('IS_TRAIN',
axis=1)
test = DF_wide[DF_wide.IS_TRAIN == 0].drop('IS_TRAIN', axis=1)

# sanity check: make sure all columns are in the same order
cols = ['cr'] + [c for c in train.columns if c != 'cr']
train = train[cols]
test = test[cols]

X_train = train.values[:, 1:]
Y_train = train.values[:, 0]
X_test = test.values[:, 1:]
Y_test = test.values[:, 0]
```

```
    # WIDE MODEL
    wide_inp = Input(shape=(X_train.shape[1],),
    dtype='float32', name='wide_inp')
    w = Dense(1, activation="sigmoid", name = "wide_model")
    (wide_inp)
    wide = Model(wide_inp, w)
    wide.compile(Adam(0.01), loss='mse', metrics=['accuracy'])
    wide.fit(X_train,Y_train,nb_epoch=10,batch_size=64)
    results = wide.evaluate(X_test,Y_test)

    print " Results with wide model:
```

Then you define the wide model.

```
def deep():
    DF_deep = pd.concat([df_train,df_test])
    target = 'cr'
    embedding_cols = ["gender", "xyz_campaign_id",
    "fb_campaign_id", "age", "interest"]
    deep_cols = embedding_cols + ['cpc','cpco','cpcoa']
    DF_deep,unique_vals = val2idx(DF_deep, embedding_cols)
    train = DF_deep[DF_deep.IS_TRAIN == 1].drop('IS_TRAIN',
    axis=1)
    test = DF_deep[DF_deep.IS_TRAIN == 0].drop('IS_TRAIN', axis=1)

    n_factors = 5
    gender, gd = embedding_input('gender_in', unique_vals[
                                 'gender'], n_factors, 1e-3)
    xyz_campaign, xyz = embedding_input('xyz_campaign_id_in',
    unique_vals[
                                        'xyz_campaign_id'], n_
                                        factors, 1e-3)
```

```
fb_campaign_id, fb = embedding_input('fb_campaign_id_in',
unique_vals[
                                    'fb_campaign_id'], n_
                                    factors, 1e-3)
age, ag = embedding_input('age_in', unique_vals[
                                'age'], n_factors, 1e-3)
interest, it = embedding_input('interest_in', unique_vals[
                                    'interest'], n_factors,
                                    1e-3)

# adding numerical columns to the deep model
cpco, cp = continous_input('cpco_in')
cpcoa, cpa = continous_input('cpcoa_in')

X_train = [train[c] for c in deep_cols]
Y_train = train[target]
X_test = [test[c] for c in deep_cols]
Y_test = test[target]

# DEEP MODEL: input same order than in deep_cols:
d = merge([gd, re, xyz, fb, ag, it], mode='concat')
d = Flatten()(d)
# layer to normalise continous columns with the embeddings
d = BatchNormalization()(d)
d = Dense(100, activation='relu',
      kernel_regularizer=l1_l2(l1=0.01, l2=0.01))(d)
d = Dense(50, activation='relu',name='deep_inp')(d)
d = Dense(1, activation="sigmoid")(d)
deep = Model([gender, xyz_campaign, fb_campaign_id, age,
interest,
            cpco, cpcoa], d)
```

```
    deep.compile(Adam(0.001), loss='mse', metrics=['accuracy'])
    deep.fit(X_train,Y_train, batch_size=64, nb_epoch=10)
    results = deep.evaluate(X_test,Y_test)

    print " Results with deep model:
```

Then you compose the wide and deep model using some cross-tabular columns.

```
def wide_deep():

    target = 'cr'
    wide_cols = ["gender", "xyz_campaign_id", "fb_campaign_id",
    "age", "interest"]
    x_cols = (['gender', 'xyz_campaign'],['age', 'interest'])

    DF_wide = pd.concat([df_train,df_test])

    crossed_columns_d = cross_columns(x_cols)

    categorical_columns =
        list(DF_wide.select_dtypes(include=['object']).columns)
    wide_columns = wide_cols + crossed_columns_d.keys()

    for k, v in crossed_columns_d.iteritems(): DF_wide[k] =
        DF_wide[v].apply(lambda x: '-'.join(x), axis=1)

    DF_wide = DF_wide[wide_columns + [target] + ['IS_TRAIN']]
    dummy_cols = [
        c for c in wide_columns if c in categorical_columns +
            crossed_columns_d.keys()]
    DF_wide = pd.get_dummies(DF_wide, columns=[x for x in
    dummy_cols])
```

```
train = DF_wide[DF_wide.IS_TRAIN == 1].drop('IS_TRAIN',
axis=1)
test = DF_wide[DF_wide.IS_TRAIN == 0].drop('IS_TRAIN', axis=1)

# sanity check: make sure all columns are in the same order
cols = ['cr'] + [c for c in train.columns if c != 'cr']
train = train[cols]
test = test[cols]

X_train_wide = train.values[:, 1:]
Y_train_wide = train.values[:, 0]
X_test_wide = test.values[:, 1:]

DF_deep = pd.concat([df_train,df_test])
embedding_cols = ['gender', 'xyz_campaign','fb_campaign_
id', 'age', 'interest']
deep_cols = embedding_cols + ['cpco','cpcoa']
DF_deep,unique_vals = val2idx(DF_deep,embedding_cols)
train = DF_deep[DF_deep.IS_TRAIN == 1].drop('IS_TRAIN',
axis=1)
test = DF_deep[DF_deep.IS_TRAIN == 0].drop('IS_TRAIN', axis=1)

n_factors = 5
gender, gd = embedding_input('gender_in', unique_vals[
                              'gender'], n_factors, 1e-3)
xyz_campaign, xyz = embedding_input('xyz_campaign_id_in',
unique_vals[
                                    'xyz_campaign_id'],
                                    n_factors, 1e-3)
fb_campaign_id, fb = embedding_input('fb_campaign_id_in',
unique_vals[
                                     'fb_campaign_id'], n_
                                     factors, 1e-3)
```

```
age, ag = embedding_input('age_in', unique_vals[
                            'age'], n_factors, 1e-3)
interest, it = embedding_input('interest_in', unique_vals[
                                'interest'], n_factors, 1e-3)

# adding numerical columns to the deep model
cpco, cp = continous_input('cpco_in')
cpcoa, cpa = continous_input('cpcoa_in')

X_train_deep = [train[c] for c in deep_cols]
Y_train_deep = train[target]
X_test_deep = [test[c] for c in deep_cols]
Y_test_deep = test[target]

X_tr_wd = [X_train_wide] + X_train_deep
Y_tr_wd = Y_train_deep # wide or deep is the same here
X_te_wd = [X_test_wide] + X_test_deep
Y_te_wd = Y_test_deep # wide or deep is the same here

#WIDE
wide_inp = Input(shape=(X_train_wide.shape[1],),
dtype='float32',
    name='wide_inp')

#DEEP
deep_inp = merge([ge, xyz, ag, fb, it, cp, cpa],
mode='concat')
deep_inp = Flatten()(deep_inp)
# layer to normalise continous columns with the embeddings
deep_inp = BatchNormalization()(deep_inp)
deep_inp = Dense(100, activation='relu',
         kernel_regularizer=l1_l2(l1=0.01, l2=0.01))
         (deep_inp)
```

```
    deep_inp = Dense(50, activation='relu',name='deep_inp')
    (deep_inp)

    #WIDE + DEEP
    wide_deep_inp = concatenate([wide_inp, deep_inp])
    wide_deep_out = Dense(1, activation='sigmoid',
        name='wide_deep_out')(wide_deep_inp)
    wide_deep = Model(inputs=[wide_inp, gender, age, xyz_
    campaign,
                            fb_campaign_id,cpco, cpcoa],
                                outputs=wide_deep_out)
    wide_deep.compile(optimizer=Adam(lr=0.001),loss='mse',
        metrics=['accuracy'])
    wide_deep.fit(X_tr_wd, Y_tr_wd, nb_epoch=50, batch_size=80)
    # wide_deep.optimizer.lr = 0.001
    # wide_deep.fit(X_tr_wd, Y_tr_wd, nb_epoch=5, batch_
    size=64)
    results = wide_deep.evaluate(X_te_wd, Y_te_wd)

    print " Results with wide and deep model:
```

The main module is finally assembled.

```
if __name__ == '__main__':

    ap = argparse.ArgumentParser()
    ap.add_argument("-method", type=str, default="wide_deep",
        help="fitting method")
    args = vars(ap.parse_args())
    method      = args["method"]

    df_train = pd.read_csv("train.csv")
    df_test = pd.read_csv("test.csv")
    df_train['IS_TRAIN'] = 1
    df_test['IS_TRAIN'] = 0
```

```
    if method == 'wide':
        wide()
    elif method == 'deep':
        deep()
    else:
        wide_deep()
```

A.3 An FCN for Image Segmentation

This section will provide the code for image segmentation using a fully convolutional network.

You will begin by doing some imports and setting some functions, as shown here:

```
import glob
import os
from PIL import Image
import numpy as np

from keras.layers import Input, Convolution2D, MaxPooling2D,
UpSampling2D, Dropout
from keras.models import Model
from keras import backend as K
from keras.callbacks import ModelCheckpoint
smooth = 1.

# define a weighted binary cross entropy function
def binary_crossentropy_2d_w(alpha):
    def loss(y_true, y_pred):
        bce = K.binary_crossentropy(y_pred, y_true)
        bce *= 1 + alpha * y_true
        bce /= alpha
        return K.mean(K.batch_flatten(bce), axis=-1)
    return loss
```

```
# define dice score to assess predictions
def dice_coef(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (2. * intersection + smooth) / (K.sum(y_true_f) +
        K.sum(y_pred_f) + smooth)

def dice_coef_loss(y_true, y_pred):
    return 1 - dice_coef(y_true, y_pred)
```

Then you load the data and the respective masks. The transpose can be skipped if you use TensorFlow as the backend (because it assumes images are specified as width×height×channels). A low-resolution image is 640×480×3.

```
def load_data(dir, boundary=False):
    X = []
    y = []
    # load images
    for f in sorted(glob.glob(dir + '/image??.png')):
        img = np.array(Image.open(f).convert('RGB'))
        X.append(img)
    # load masks
    for i, f in enumerate(sorted(glob.glob(dir + '/image??_
    mask.txt'))):
        if boundary:
            a = get_boundary_mask(f)
            y.append(np.expand_dims(a, axis=0))
        else:
            content = open(f).read().split('')[1:-1]
            a = np.array(content, 'i').reshape(X[i].shape[:2])
            a = np.clip(a, 0, 1).astype('uint8')
            y.append(np.expand_dims(a, axis=0))
```

```
    # stack data
    X = np.array(X) / 255.
    y = np.array(y)
    X = np.transpose(X, (0, 3, 1, 2))
    return X, y
```

Then you define the network used for training. You start with eight filters, and each time you do max pooling, it doubles: 16, 32, and so on.

```
# define the network model
def net_2_outputs(input_shape):
    input_img = Input(input_shape, name='input')

    x = Convolution2D(8, 3, 3, activation='relu',
        border_mode='same')(input_img)
    x = Convolution2D(8, 3, 3, activation='relu', border_
        mode='same')(x)
    x = Convolution2D(8, 3, 3, subsample=(1, 1),
        activation='relu', border_mode='same')(x)
    x = MaxPooling2D((2, 2), border_mode='same')(x)
    x = Convolution2D(16, 3, 3, activation='relu', border_
        mode='same')(x)
    x = Convolution2D(16, 3, 3, activation='relu', border_
        mode='same')(x)
    x = Convolution2D(16, 3, 3, subsample=(1, 1),
        activation='relu',
        border_mode='same')(x)
    x = MaxPooling2D((2, 2), border_mode='same')(x)
    x = Convolution2D(32, 3, 3, activation='relu', border_
        mode='same')(x)
    x = Convolution2D(32, 3, 3, activation='relu', border_
        mode='same')(x)
    x = Convolution2D(32, 3, 3, activation='relu', border_
        mode='same')(x)
```

```
    # up
    x = UpSampling2D((2, 2))(x)
    x = Convolution2D(16, 3, 3, activation='relu', border_
        mode='same')(x)
    x = UpSampling2D((2, 2))(x)
    x = Convolution2D(8, 3, 3, activation='relu', border_
        mode='same')(x)
    output = Convolution2D(1, 3, 3, activation='sigmoid',
        border_mode='same', name='output')(x)

    model = Model(input_img, output=[output])
    model.compile(optimizer='adam', loss='output':
        binary_crossentropy_2d_w(5))
return model
```

Next, you train the model.

```
def train():
    X, y = load_data(DATA_DIR_TRAIN.replace('c_type', c_type),
        boundary=False) # load the data

    print(X.shape, y.shape) # make sure it's the right shape
    h = X.shape[2]
    w = X.shape[3]
    training_data = ShuffleBatchGenerator(input_data='input': X,
        output_data='output': y, 'output_b': y_b) # generate
        batches for
        training and testing
    training_data_aug = DataAugmentation(training_data,
        inplace_transfo=['mirror', 'transpose']) # apply some data
        augmentation
    net = net_2_outputs((X.shape[1], h, w))
    net.summary()
```

```
    model = net
    model.fit(training_data_aug, 300, 1, callbacks=[ProgressBar
    Callback()])
    net.save('model.hdf5' )

    # save predictions to disk
    res = model.predict(training_data, training_data.nb_
    elements)
    if not os.path.isdir('res'):
        os.makedirs('res')
    for i, img in enumerate(res[0]):
        Image.fromarray(np.squeeze(img) *
            255).convert('RGB').save('res/
    for i, img in enumerate(res[1]):
        Image.fromarray(np.squeeze(img) *
            255).convert('RGB').save('res/

if __name__ == '__main__':
        train()
```

A.3.1 Sequence to Sequence

Sequence-to-sequence models (seq2seq) convert a sequence from one domain (e.g., sentences in English) to a sequence in another domain (e.g., the same sentences translated to French) or convert from past observations to a sequence of future observations (prediction).

When both sequences have the same length, a simple Keras LSTM is enough. In the general case of arbitrary lengths where the entire input sequence is required, an RNN layer will act as the encoder. It projects the input sequence into its own internal state (the context), and another RNN layer is trained as the decoder to predict the next elements of the target sequence. The encoder uses as the initial state the vectors from

the encoder. The decoder learns to generate `targets[t+1...]` given `targets[...t]`, conditioned on the input sequence. The following example was created by F. Chollet and is available online at https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html:

```
from keras.models import Model
from keras.layers import Input, LSTM, Dense

encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
# We discard 'encoder_outputs' and only keep the states.
encoder_states = [state_h, state_c]

# Set up the decoder, using 'encoder_states' as initial state.
decoder_inputs = Input(shape=(None, num_decoder_tokens))
# We set up our decoder to return full output sequences,
# and to return internal states as well. We don't use the
# return states in the training model, but we will use them in
inference.
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_
state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                     initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

# Define the model that will turn
# 'encoder_input_data' 'decoder_input_data' into 'decoder_
target_data'
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

# Run training
```

```
model.compile(optimizer='rmsprop', loss='categorical_
crossentropy')
model.fit([encoder_input_data, decoder_input_data], decoder_
target_data,
        batch_size=batch_size,
        epochs=epochs,
        validation_split=0.2)

encoder_model = Model(encoder_inputs, encoder_states)

decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_
input_c]
decoder_outputs, state_h, state_c = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = Model(
    [decoder_inputs] + decoder_states_inputs,
    [decoder_outputs] + decoder_states)

def decode_sequence(input_seq):
    # Encode the input as state vectors.
    states_value = encoder_model.predict(input_seq)

    # Generate empty target sequence of length 1.
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    # Populate the first character of target sequence with the
    start character.
    target_seq[0, 0, target_token_index['']] = 1.

    # Sampling loop for a batch of sequences
    # (to simplify, here we assume a batch of size 1).
```

```
    stop_condition = False
    decoded_sentence = ''
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict(
            [target_seq] + states_value)

        # Sample a token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = reverse_target_char_index[sampled_token_
        index]
        decoded_sentence += sampled_char

        # Exit condition: either hit max length
        # or find stop character.
        if (sampled_char == '' or
            len(decoded_sentence) > max_decoder_seq_length):
            stop_condition = True

        # Update the target sequence (of length 1).
        target_seq = np.zeros((1, 1, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.

        # Update states
        states_value = [h, c]

    return decoded_sentence
```

A.4 The Backpropagation on a Multilayer Perceptron

In this section, we will consider a rather general neural network consisting of L layers (of course not counting the input layer). Let's consider an arbitrary layer, say ℓ, which has N_ℓ neurons, $X_1^{(\ell)}, X_2^{(\ell)}, \ldots, X_{N_\ell}^{(\ell)}$, each with

a transfer function, $f^{(\ell)}$. Notice that the transfer function may be different from layer to layer. As in the extended Delta rule, the transfer function may be given by any differentiable function but does not need to be linear. These neurons receive signals from the neurons in the preceding layer, $\ell-1$. For example, neuron $X_j^{(\ell)}$ receives a signal from $X_i^{(\ell-1)}$ with a weight factor of $w_{ij}^{(\ell)}$. Therefore, you have an $N_{\ell-1}$ by N_ℓ weight matrix, $\mathbf{W}^{(\ell)}$, whose elements are given by $W_{ij}^{(\ell)}$, for $i=1,2,\ldots,N_{\ell-1}$ and $j=1,2,\ldots,N_\ell$. Neuron $X_j^{(\ell)}$ also has a bias given by $b_j^{(\ell)}$, and its activation is $a_j^{(\ell)}$.

To simplify the notation, you will use $n_j^{(\ell)}\left(=y_{in,j}\right)$ to denote the net input into neuron $X_j^{(\ell)}$. It is given as follows:

$$n_j^{(\ell)}=\sum_{i=1}^{N_{\ell-1}}a_i^{(\ell-1)}w_{ij}^{(\ell)}+b_j^{(\ell)},\ j=1,2,\ldots,N_\ell.$$

Thus, the activation of neuron $X_j^{(\ell)}$ is as follows:

$$a_j^{(\ell)}=f^{(\ell)}\left(n_j^{(\ell)}\right)=f^{(\ell)}\left(\sum_{i=1}^{N_{\ell-1}}a_i^{(\ell-1)}w_{ij}^{(\ell)}+b_j^{(\ell)}\right).$$

You can consider the zeroth layer as the input layer. If an input vector $\mathbf{x}$ has N components, then $N_0=N$, and neurons in the input layer have activations $a_i^{(0)}=x_i, i=1,2,\ldots,N_0$.

Layer L of the network is the output layer. Assuming that the output vector $\mathbf{y}$ has M components, you must have $N_L=M$. These components are given by $y_j=a_j^{(L)}, j=1,2,\ldots,M$.

For any given input vector, the previous equations can be used to find the activation for each neuron for any given set of weights and biases. In particular, the network output vector $\mathbf{y}$ can be found. The remaining question is how to train the network to find a set of weights and biases for it to perform a certain task.

You will now consider training a rather general multilayer perceptron for pattern association using the BP algorithm. Training is carried out supervised, so you can assume that a set of pattern pairs (or associations), as in $\mathbf{s}^{(q)}:\mathbf{t}^{(q)}, q=1,2,\ldots,Q$, is given. The training vectors $\mathbf{s}^{(q)}$ have N components, as shown here:

$$\mathbf{s}^{(q)}=\begin{bmatrix} s_1^{(q)} & s_2^{(q)} & \ldots & s_N^{(q)} \end{bmatrix},$$

Their targets, $\mathbf{t}^{(q)}$, have M components, as shown here:

$$\mathbf{t}^{(q)}=\begin{bmatrix} t_1^{(q)} & t_2^{(q)} & \ldots & t_M^{(q)} \end{bmatrix}.$$

Just like in the Delta rule, the training vectors are presented one at a time to the network during training. Suppose in time step t of the training process, a training vector $\mathbf{s}^{(q)}$ for a particular q is presented as input, $\mathbf{x}(t)$, to the network. The input signal can be propagated forward through the network using the equations in the previous section and the current set of weights and biases to obtain the corresponding network output, $\mathbf{y}(t)$. The weights and biases are then adjusted using the steepest descent algorithm to minimize the square of the error for this training vector:

$$E=\left\|\mathbf{y}(t)-\mathbf{t}(t)\right\|^2,$$

Here, $\mathbf{t}(t)=\mathbf{t}^{(q)}$ is the corresponding target vector for the chosen training vector $\mathbf{s}^{(q)}$.

This square error E is a function of all the weights and biases of the entire network since $\mathbf{y}(t)$ depends on them. You need to find the set of updating rules for them based on the steepest descent algorithm.

$$w_{ij}^{(\ell)}(t+1)=w_{ij}^{(\ell)}(t)-\alpha\frac{\partial E}{\partial w_{ij}^{(\ell)}(t)}$$

$$b_j^{(\ell)}(t+1)=b_j^{(\ell)}(t)-\alpha\frac{\partial E}{\partial b_j^{(\ell)}(t)},$$

Here, $\alpha(>0)$ is the learning rate.

To compute these partial derivatives, you need to understand how E depends on the weights and biases. First, E depends explicitly on the network output $\mathbf{y}(t)$ (the activations of the last layer, $\mathbf{a}^{(L)}$), which then depends on the net input into the L-th layer, $\mathbf{n}^{(L)}$. In turn, $\mathbf{n}^{(L)}$ is given by the activations of the preceding layer and the weights and biases of layer L. The explicit relation is as follows (for brevity, the dependence on step t is omitted):

$$E=\left\|\mathbf{y}-\mathbf{t}(t)\right\|^2=\left\|\mathbf{a}^{(L)}-\mathbf{t}(t)\right\|^2=\left\|f^{(L)}\left(\mathbf{n}^{(L)}\right)-\mathbf{t}(t)\right\|^2$$

$$=\left\|f^{(L)}\left(\sum_{i=1}^{N_{L-1}}a_i^{(L-1)}w_{ij}^{(L)}+b_j^{(L)}\right)-\mathbf{t}(t)\right\|^2.$$

It is then easy to compute the partial derivatives of E with respect to the elements of $\mathbf{W}^{(L)}$ and $\mathbf{b}^{(L)}$ using the chain rule for differentiation.

$$\frac{\partial E}{\partial w_{ij}^{(L)}}=\sum_{n=1}^{N_L}\frac{\partial E}{\partial n_n^{(L)}}\frac{\partial n_n^{(L)}}{\partial w_{ij}^{(L)}}.$$

Notice the sum is needed in the previous equation for the correct application of the chain rule. You now define the sensitivity vector for a general layer ℓ to have components.

$$s_n^{(\ell)}=\frac{\partial E}{\partial n_n^{(\ell)}}\,n=1,2,\ldots,N_\ell.$$

This is called the sensitivity of neuron $X_n^{(\ell)}$ because it gives the change in the output error, *E*, per unit change in the net input it receives.

For layer *L*, it is easy to compute the sensitivity vector directly using the chain rule to obtain this.

$$s_n^{(L)} = 2\left(a_n^{(L)} - t_n(t)\right)\dot{f}^{(L)}\left(n_n^{(L)}\right), n = 1,2,\ldots,N_L.$$

Here, $\dot{f}$ denotes the derivative of the transfer function *f*. You also know the following:

$$\frac{\partial n_n^{(L)}}{\partial w_{ij}^{(L)}} = \frac{\partial}{\partial w_{ij}^{(L)}}\left(\sum_{m=1}^{N_{L-1}} a_m^{(L-1)} w_{mn}^{(L)} + b_n^{(L)}\right) = \delta_{nj} a_i^{(L-1)}.$$

Therefore, you have this:

$$\frac{\partial E}{\partial w_{ij}^{(L)}} = a_i^{(L-1)}\, s_j^{(L)}.$$

Similarly, you have this:

$$\frac{\partial E}{\partial b_j^{(L)}} = \sum_{n=1}^{N_L} \frac{\partial E}{\partial n_n^{(L)}} \frac{\partial n_n^{(L)}}{\partial b_j^{(L)}},$$

In addition, since you have this:

$$\frac{\partial n_n^{(L)}}{\partial b_j^{(L)}} = \delta_{nj},$$

then you get the following:

$$\frac{\partial E}{\partial b_j^{(L)}} = s_j^{(L)}.$$

For a general layer, ℓ, you can write this:

$$\frac{\partial E}{\partial w_{ij}^{(\ell)}} = \sum_{n=1}^{N_\ell} \frac{\partial E}{\partial n_n^{(\ell)}} \frac{\partial n_n^{(\ell)}}{\partial w_{ij}^{(\ell)}} = \sum_{n=1}^{N_\ell} s_n^{(\ell)} \frac{\partial n_n^{(\ell)}}{\partial w_{ij}^{(\ell)}}.$$

$$\frac{\partial E}{\partial b_j^{(\ell)}} = \sum_{n=1}^{N_\ell} \frac{\partial E}{\partial n_n^{(\ell)}} \frac{\partial n_n^{(\ell)}}{\partial b_j^{(\ell)}} = \sum_{n=1}^{N_\ell} s_n^{(\ell)} \frac{\partial n_n^{(\ell)}}{\partial b_j^{(\ell)}}.$$

Since you have this:

$$n_n^{(\ell)} = \sum_{m=1}^{N_{\ell-1}} a_m^{(\ell-1)} w_{mn}^{(\ell)} + b_n^{(\ell)},\ j = 1,2,\ldots,N_\ell,$$

the you have the following:

$$\frac{\partial n_n^{(\ell)}}{\partial w_{ij}^{(\ell)}} = \delta_{nj} a_i^{(\ell-1)}$$

$$\frac{\partial n_n^{(\ell)}}{\partial b_j^{(\ell)}} = \delta_{nj},$$

and finally the following:

$$\frac{\partial E}{\partial w_{ij}^{(\ell)}} = a_i^{(\ell-1)}\, s_j^{(\ell)},\ \frac{\partial E}{\partial b_j^{(\ell)}} = s_j^{(\ell)}.$$

Therefore, the updating rules for the weights and biases are as follows (now you put back the dependency on the step index t):

$$w_{ij}^{(\ell)}(t+1) = w_{ij}^{(\ell)}(t) - \alpha\, a_i^{(\ell-1)}(t)\, s_j^{(\ell)}(t)$$

$$b_j^{(\ell)}(t+1) = b_j^{(\ell)}(t) - \alpha\, s_j^{(\ell)}(t),$$

To use these updating rules, you need to be able to compute the sensitivity vectors $\mathbf{s}^{(\ell)}$ for $\ell = 1,2,\ldots,L-1$. From their definition, you have this:

$$s_j^{(\ell)} = \frac{\partial E}{\partial n_j^{(\ell)}} \; j = 1,2,\ldots,N_\ell,$$

You need to know how E depends on $n_j^{(\ell)}$. The key to computing these partial derivatives is to note that $n_j^{(\ell)}$ in turn depends on $n_i^{(\ell-1)}$ for $i = 1,2,\ldots,N_{\ell-1}$, because the net input for layer ℓ depends on the activation of the previous layer, $\ell-1$, which in turn depends on the net input for layer $\ell-1$. Specifically, you have this for $j = 1,2,\ldots,N_\ell$:

$$n_j^{(\ell)} = \sum_{i=1}^{N_{\ell-1}} a_i^{(\ell-1)} w_{ij}^{(\ell)} + b_j^{(\ell)} = \sum_{i=1}^{N_{\ell-1}} f^{(\ell-1)}\left(n_i^{(\ell-1)}\right) w_{ij}^{(\ell)} + b_j^{(\ell)}$$

Therefore, you have the following for the sensitivity of layer $\ell-1$:

$$s_j^{(\ell-1)} = \frac{\partial E}{\partial n_j^{(\ell-1)}} = \sum_{i=1}^{N_\ell} \frac{\partial E}{\partial n_i^{(\ell)}} \frac{\partial n_i^{(\ell)}}{\partial n_j^{(\ell-1)}}$$

$$= \sum_{i=1}^{N_\ell} s_i^{(\ell)} \frac{\partial}{\partial n_j^{(\ell-1)}} \left(\sum_{m=1}^{N_{\ell-1}} f^{(\ell-1)}\left(n_m^{(\ell-1)}\right) w_{mi}^{(\ell)} + b_i^{(\ell)} \right)$$

$$= \sum_{i=1}^{N_\ell} s_i^{(\ell)} \dot{f}^{(\ell-1)}\left(n_j^{(\ell-1)}\right) w_{ji}^{(\ell)} = \dot{f}^{(\ell-1)}\left(n_j^{(\ell-1)}\right) \sum_{i=1}^{N_\ell} w_{ji}^{(\ell)} s_i^{(\ell)}.$$

Thus, the sensitivity of a neuron in layer $\ell-1$ depends on the sensitivities of all the neurons in layer ℓ. This is a recursion relation for the sensitivities of the network since the sensitivities of the last layer L is known. To find the activations or the net inputs for any given layer, you need to feed the input from the left of the network and proceed forward to the layer in question. However, to find the sensitivities for any given layer,

you need to start from the last layer and use the recursion relation going backward to the given layer. This is why the training algorithm is called *backpropagation*.

To compute the updates for the weights and biases, you need to find the activations and sensitivities for all the layers. To obtain the sensitivities, you also need $\dot{f}^{(\ell)}\left(n_j^{(\ell)}\right)$. That means that in general you need to keep track of all the $n_j^{(\ell)}$ as well.

In neural networks trained using the backpropagation algorithm, there are two functions often used as the transfer functions. One is the log-sigmoid function, shown here:

$$f_{logsig}(x)=\frac{1}{1+e^{-x}}$$

This is differentiable, and its value goes smoothly and monotonically between 0 and 1 for x around 0. The other is the hyperbolic tangent sigmoid function, shown here:

$$f_{tansig}(x)=\frac{1-e^{-x}}{1+e^{-x}}=\tanh(x/2)$$

This is also differentiable, but its value goes smoothly between -1 and 1 for x around 0. It is easy to see that the first derivatives of these functions are given in terms of the same functions alone.

$$\dot{f}_{logsig}(x)=f_{logsig}(x)\left[1-f_{logsig}(x)\right]$$

$$\dot{f}_{tansig}(x)=\frac{1}{2}\left[1+f_{tansig}(x)\right]\left[1-f_{tansig}(x)\right]$$

Since $f^{(\ell)}\left(n_j^{(\ell)}\right)=a_j^{(\ell)}$, in implementing the neural network on a computer, there is actually no need to keep track of $n_j^{(\ell)}$ at all (thus saving memory).

References

[AAB+15] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Y. Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. *CoRR*, abs/1512.02595, 2015.

[AAL+15] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh. VQA: Visual Question Answering. In *International Conference on Computer Vision (ICCV)*, 2015.

[AG13] G. Hinton A. Graves, A. Mohamed. Speech recognition with deep recurrent neural networks. *Arxiv*, 2013.

[AHS85] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9(1):147–169, 1985.

A. Vieira and B. Ribeiro, *Introduction to Deep Learning Business Applications for Developers*,
https://doi.org/10.1007/978-1-4842-3453-2

[AIG12] Krizhevsky A., Sutskever I., and Hinton G. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems. Curran Associates*, 25:1106–1114, 2012. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[AM15] E. Asgari, M. R. Mofrad. Continuous Distributed Representation of Biological Sequences for Deep Proteomics and Genomics. *PloS one*, 10(11):e0141287, 2015.

[AOS+16] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *CoRR*, abs/1606.06565, 2016.

[ARDK16] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Learning to compose neural networks for question answering. *CoRR*, abs/1601.01705, 2016.

[AV03] N. P. Barradas A. Vieira. A training algorithm for classification of high-dimensional data. *Neurocomputing*, 50:461–472, 2003.

[AV18] Attul Sehgal Armando Vieira. How banks can better serve their customers through artificial techniques. In *Digital Markets Unleashed*, page 311. Springer-Verlag, 2018.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[BLPL06] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS'06, pages 153–160, Cambridge, MA, USA, 2006. MIT Press.

[BUGD+13] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2787–2795. Curran Associates, Inc., 2013.

[CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15:1–15:58, 2009.

[CCB15] KyungHyun Cho, Aaron C. Courville, and Yoshua Bengio. Describing multimedia content using attention-based encoder-decoder networks. *CoRR*, abs/1507.01053, 2015.

[CHY+14] Charles F. Cadieu, Ha Hong, Daniel L. K. Yamins, Nicolas Pinto, Diego Ardila, Ethan A. Solomon, Najib J. Majaj, and James J. DiCarlo. Deep neural networks rival the representation of primate IT cortex for core visual object recognition. *PLOS Computational Biology*, 10(12):1–18, 12 2014.

[CLN+16] Y. Chen, Y. Li, R. Narayan et al. Gene expression inference with deep learning. *Bioinformatics*, 2016(btw074).

[DCH+16] Yan Duan, Xi Chen, Rein Houthooft, John Schulman and Pieter Abbeel. Benchmarking Deep Reinforcement Learning for Continuous Control. *CoRR*, abs/1604.06778, 2016.

[DDT+16] Nan Du, Hanjun Dai, Rakshit Trivedi, Utkarsh Upadhyay, Manuel Gomez-Rodriguez, and Le Song. Recurrent marked temporal point processes: Embedding event history to vector. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'16, pages 1555–1564, New York, NY, USA, 2016. ACM.

[DHG+14] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. *CoRR*, abs/1411.4389, 2014.

[Doe16] Carl Doersch. Tutorial on variational autoencoders. *CoRR*, 2016.

[E12] Dumbill E. What is big data? an introduction to the big data landscape. *In Strata*, 2012. Making Data Work. O'Reilly, Santa Clara, CA O'Reilly.

[EBC+10] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *J. Mach. Learn. Res.*, 11:625–660, March 2010.

[EHW+16] S. M. Ali Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, Koray Kavukcuoglu, and Geoffrey E. Hinton. Attend, infer, repeat: Fast scene understanding with generative models. *CoRR*, abs/1603.08575, 2016.

[Elm90] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[FF15] Ralph Fehrer and Stefan Feuerriegel. Improving decision analytics with deep learning: The case of financial disclosures. 2015. https://arxiv.org/pdf/1508.01993v1.pdf.

[FG16] Basura Fernando and Stephen Gould. Learning end-to-end video classification with rank-pooling. *ICML*, 2016. http://jmlr.org/proceedings/papers/v48/fernando16.pdf.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. www.deeplearningbook.org.

[GBWB13] Xavier Glorot, Antoine Bordes, Jason Weston, and Yoshua Bengio. A semantic matching energy function for learning with multi-relational data. *CoRR*, abs/1301.3485, 2013.

[GEB15] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015.

[GLO+16] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S. Lew. Deep learning for visual understanding. *Neurocomput.*, 187(C):27–48, April 2016.

[GMZ+15] Haoyuan Gao, Junhua Mao, Jie Zhou, Zhiheng Huang, Lei Wang, and Wei Xu. Are you talking to a machine? dataset and methods for multilingual image question answering. pages 2296–2304, 2015.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[GR06] Hinton GE and Salakhutdinov RR. Reducing the dimensionality of data with neural networks. *Science 313(5786): 504–507*, 2006.

[GVS+16] Shalini Ghosh, Oriol Vinyals, Brian Strope, Scott Roy, Tom Dean, and Larry Heck. Contextual LSTM (CLSTM) models for large scale NLP tasks. *CoRR*, abs/1602.06291, 2016.

[HDFN95] G. E. Hinton, P. Dayan, B. J. Frey, and R. M. Neal. The wake-sleep algorithm for unsupervised neural networks. *Science*, 268:1158–1161, 1995.

[Hin02] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Comput.*, 14(8):1771–1800, August 2002.

[HKG+15] Karl Moritz Hermann, Tomáš Kočiský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, pages 1693–1701. MIT Press, Cambridge, MA, USA, 2015.

[HOT06] G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[HSL+16] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. Deep networks with stochastic depth. *CoRR*, abs/1603.09382, 2016.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[J15] Schmidhuber J. Deep learning in neural networks: An overview. *Neural Networks*, 61, 2015.

[Jor90] Michael I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In Joachim Diederich, editor, *Artificial Neural Networks*, pages 112–127. IEEE Press, Piscataway, NJ, USA, 1990.

[KFF17] A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):664–676, 2017.

[KZS+15] Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Antonio Torralba, Raquel Urtasun, and Sanja Fidler. Skip-thought vectors. *CoRR*, abs/1506.06726, 2015.

[LBD+89] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, December 1989.

[LBP+16] B. Lee, J. Baek, S. Park et al. deepTarget: End-to-end Learning Framework for microRNA Target Prediction using Deep Recurrent Neural Networks. *arXiv preprint arXiv*, 1603.09123, 2016.

[LCWJ15] Mingsheng Long, Yue Cao, Jianmin Wang, and Michael I. Jordan. Learning transferable features with deep adaptation networks. *ICML*, 2015. http://jmlr.org/proceedings/papers/v37/long15.pdf.

[LFDA16] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17:1–40, 2016. https://arxiv.org/abs/1504.00702.

[LM14] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[LST15] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science 350.6266*, pages 1332–1338, 2015.

[M13] Grobelnik M. Big data tutorial. *European Data Forum*, 2013. www.slideshare.net/EUDataForum/edf2013-big-datatutorialmarkogrobelnik.

[Mac03] D.J.C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[MBM+16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. http://arxiv.org/abs/1602.01783.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[MLS13] Tomas Mikolov, Quoc V. Le, and Ilya Sutskever. Exploiting similarities among languages for machine translation. *CoRR*, abs/1309.4168, 2013.

[MVPZ16] Polina Mamoshina, Armando Vieira, Evgeny Putin, and Alex Zhavoronkov. Applications of deep learning in biomedicine. *Molecular Pharmaceutics*, 13(5):1445–1454, 2016.

[MXY+14] Junhua Mao, Wei Xu, Yi Yang, Jiang Wang, and Alan L. Yuille. Explain images with multimodal recurrent neural networks. *CoRR*, abs/1410.1090, 2014.

[MZMG15] Ishan Misra, C. Lawrence Zitnick, Margaret Mitchell, and Ross B. Girshick. Learning visual classifiers using human-centric annotations. *CoRR*, abs/1512.06974, 2015.

[NSH15] Hyeonwoo Noh, Paul Hongsuck Seo, and Bohyung Han. Image question answering using convolutional neural network with dynamic parameter prediction. *CoRR*, abs/1511.05756, 2015.

[O'N03] Cathy O'Neil. *Weapons of Math Destruction*. Penguin, 2003.

[PBvdP16] Xue Bin Peng, Glen Berseth, and Michiel van de Panne. Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Trans. Graph.*, 35(4):81:1–81:12, July 2016.

[RG09] Salakhutdinov R and Hinton GE. Deep Boltzmann Machines. *JMLR*, 2009.

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015. http://arxiv.org/abs/1511.06434.

[SA08] Saratha Sathasivam and Wan Ahmad Tajuddin Wan Abdullah. Logic learning in hopfield networks. *CoRR*, abs/0804.4075, 2008.

[SGS15] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *CoRR*, abs/1505.00387, 2015.

[SHM+16] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman anc Dominik Grewe anc John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[SKM84] R. Snow, P. Kyllonen, and B. Marshalek. The topography of ability and learning correlations. In *Advances in the Psychology of Human Intelligence*, pages 47–103, June 1984.

[SMB10] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *Proceedings of the 20th International Conference on Artificial Neural Networks: Part III*, ICANN'10, pages 92–101, Berlin, Heidelberg, 2010. Springer-Verlag.

[SMGS15] Marijn F. Stollenga, Jonathan Masci, Faustino Gomez, and Juergen Schmidhuber. Deep networks with internal selective attention through feedback connections. *NIPS*, 2015. https://papers.nips.cc/paper/5276-deep-networks-with-internal-selective-attention-through-feedback-connections.pdf.

[Smo86] Paul Smolensky. Chapter 6: Information processing in dynamical systems: Foundations of harmony theory. In David E. Rumelhart and James L. McLelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, volume 1, pages 194–281. MIT Press, 1986.

[SSB+15] Iulian Vlad Serban, Alessandro Sordoni, Yoshua Bengio, Aaron C. Courville, and Joelle Pineau. Hierarchical neural network generative models for movie dialogues. *CoRR*, abs/1507.04808, 2015.

[SSN+15] S. K. Sønderby, C. K. Sønderby, H. Nielsen et al. Convolutional LSTM Networks for Subcellular Localization of Proteins. *arXiv preprint arXiv*, 1503.01919, 2015.

[SSS+15] Basu Saikat, Ganguly Sangram, Mukhopadhyay Supratik, DiBiano Robert, Karki Manohar, and Nemani Ramakrishna. Deepsat: A learning framework for satellite imagery. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '15, pages 37:1–37:10, New York, NY, USA, 2015. ACM.

[SsWF15] Sainbayar Sukhbaatar, arthur szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2440–2448. Curran Associates, Inc., 2015.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[SZ16] Falong Shen and Gang Zeng. Weighted residuals for very deep networks. *CoRR*, abs/1605.08831, 2016.

[ULVL16] Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor S. Lempitsky. Texture networks: Feed-forward synthesis of textures and stylized images. *CoRR*, abs/1603.03417, 2016.

[vdOKV+16] Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelCNN decoders. *CoRR*, abs/1606.05328, 2016.

[VKFU15] Ivan Vendrov, Ryan Kiros, Sanja Fidler, and Raquel Urtasun. Order-embeddings of images and language. *CoRR*, abs/1511.06361, 2015.

[VLBM08] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1096–1103, New York, NY, USA, 2008. ACM.

[VTBE14] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. *CoRR*, abs/1411.4555, 2014.

[VXD+14] Subhashini Venugopalan, Huijuan Xu, Jeff Donahue, Marcus Rohrbach, Raymond J. Mooney, and Kate Saenko. Translating videos to natural language using deep recurrent neural networks. *CoRR*, abs/1412.4729, 2014.

[Wes16] Jason Weston. Dialog-based language learning. *CoRR*, abs/1604.06045, 2016.

[WS98] M. Wiering and J. Schmidhuber. HQ-learning. *Adaptive Behavior*, 6(2):219–246, 1998.

[WWY15] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1235–1244, New York, NY, USA, 2015. ACM.

[WZFC14] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In Carla E. Brodley and Peter Stone, editors, *AAAI*, pages 1112–1119. AAAI Press, 2014.

[YAP13] Bengio Y, Courville A, and Vincent P. Representation learning: A review and new perspectives. pattern analysis and machine intelligence. *IEEE Transactions*, 35(8):1798–1828, 2013.

[YZP15] Kaisheng Yao, Geoffrey Zweig, and Baolin Peng. Attention with intention for a neural network conversation model. *CoRR*, abs/1510.08565, 2015.

[ZCLZ16] Shuangfei Zhai, Yu Cheng, Weining Lu, and Zhongfei Zhang. Deep structured energy based models for anomaly detection. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning – Volume 48*, ICML'16, pages 1100–1109. JMLR.org, 2016.

[ZCSG16] Ke Zhang, Wei-Lun Chao, Fei Sha, and Kristen Grauman. *Video Summarization with Long Short-Term Memory*, pages 766–782. Springer International Publishing, Cham, 2016.

[ZKZ+15] Yukun Zhu, Ryan Kiros, Richard S. Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. *CoRR*, abs/1506.06724, 2015.

[ZT15] J. Zhou, O. G. Troyanskaya. Predicting effects of noncoding variants with deep learning-based sequence model. *Nature methods*, 12(10):931-4, 2015.

Index

A

A. Vieira and B. Ribeiro, *Introduction to Deep Learning Business Applications for Developers*,
https://doi.org/10.1007/978-1-4842-3453-2

B

C

D

E

F

G

H

I, J

K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z