Computer Vision (CE316 and CE866): Neural Networks

Adrian F. Clark
⟨*alien@essex.ac.uk*⟩

Neural networks, especially those with many hidden layers, represent the current state of the art in machine learning applied to computer vision. We review two neural network approaches, the well-established *Multi-Layer Perceptron* and the more recent *Convolutional Neural Network*.

Contents

List of Figures

1 Introduction

The brains of animals consist of large numbers of simple neurons which have many connections to other neurons. A neural network is a computer model of this: an individual neuron contains little information but is connected to many other neurons. They were first devised in the 1950s but came to prominence in the 1980s, when they were shown to be able to solve some problems that other techniques could not. Interest waned in the 1990s because techniques such as the support vector machine (SVM) out-performed them, but has been re-invigorated in recent years because networks with many layers (so-called 'deep learning' networks) out-perform the SVM. (This kind of 'arms race' between machine learning techniques will undoubtedly continue.)

We shall start by considering a 'traditional' neural network, the multi-layer perceptron (MLP), because that establishes the fundamental idea of how neurons are modelled and how training proceeds. We then move on to the convolutional neural network (CNN), looking at its basic idea and some common instances of it in recent research.

2 The multi-layer perceptron

An MLP is a network of neurons that maps a set of inputs onto a set of outputs — for example, with the MNIST dataset in mind, the pixels of an image onto the 10 classes of digit. It consists of a number of layers of *neurons* (simple computational elements) as illustrated in Figure 1, where the circles represent neurons and the lines between them *interconncetions*. An MLP is a *fully-connected* network as each node in one layer connects with some weight w to every node in the following layer. The data are presented at the *input layer*, which passes them to a single *hidden layer* in Figure 1, and that in turn passes them to a *output layer*. The layers can of course be organised in two dimensions to accept images as input, and 'deep learning' architectures have more than one hidden layer.

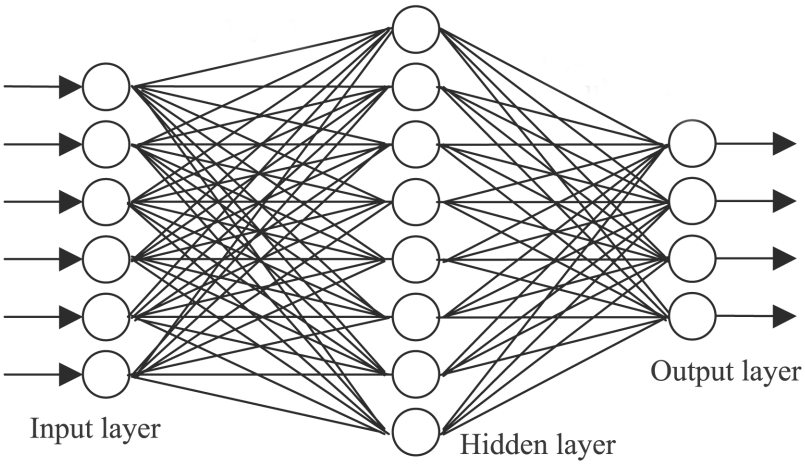

Figure 1: Illustration of a multi-layer perceptron. (Image from http://www.slideshare.net/steveomohundro/tedx-talk-whats-happening-with-artificial-intelli

Neural networks of the 1980s normally had a single hidden layer as training a network took a long time; the longest time the author has al-

lowed a network to train was 88 days! These days however, computational performance has increased to the point that a dozen or more hidden layers are now not uncommon, though training times remain long.

The simplest neuron sums its N weighted inputs and uses the result as the argument of a non-linear function:

$$y = f\left(\sum_{i=1}^{N} x_i w_i\right) \tag{1}$$

where f is the non-linear function, x_i the inputs to a neuron and w_i the corresponding connection strengths. Strictly speaking, to be a *perceptron*, the output of the neuron should be zero or unity; but it has become common to use smoother functions, and popular choices are 'sigmoid' functions such as

$$y = \tanh\left(\sum_{i} x_i w_i\right) \tag{2}$$

and

$$y = \frac{1}{1+\exp(-\sum_i x_i w_i)} \tag{3}$$

as shown in Figure 2. More sophisticated choices include various different *radial basis functions* such as Gaussians. The activation function needs to be differentiable.

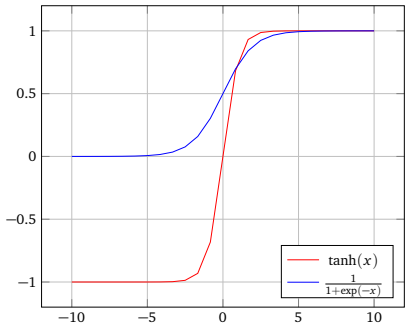

Figure 2: Common choices for the activation function of a neuron

Training, or learning, rules specify an initial set of connection weightings and indicate how these weights should be altered during use to enhance performance. There are two types of training, supervised and unsupervised. In the former, which is much more common, the neural network is supplied with a target output for each input pattern. The quality of the output may be measured by computing the RMS error, and one of several iterative schemes may be employed to modify the weights so that the RMS error is reduced. When the RMS error has been minimised (or, more commonly, a given number of iterations or *epochs* have been done), the network is said to be trained. An acceptable RMS target error is normally allowed so that, if the convergence error is less than or equal to this target error, the neural network is said to have found the 'correct' result. For the unsupervised case, the network adjusts its weights in response to input patterns without any target answers and classifies the input into similarity classes.

The learning rule is at the heart of a neural network as it determines how the connection weights are adjusted as the neural network learns. The error in output node j in the n^{th} training example is given by

$$e_j(n) = d_j(n) - y_j(n) \tag{4}$$

where d_j is the target value and y_j the value produced by the neuron. We then make corrections to the weights of the nodes based on those corrections that minimize the error in the entire output, given by

$$\mathscr{E}(n) = \sum_j e_j^2(n). \tag{5}$$

Using gradient descent, the change in each weight is

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathscr{E}(n)}{\partial v_j(n)} y_i(n) \tag{6}$$

where y_i is the output of the previous neuron and η is the *learning rate*, which has to be selected to ensure that the weights converge to a response fast enough without producing oscillations.

For an output node, this derivative can be simplified to

$$-\frac{\partial \mathscr{E}(n)}{\partial v_j(n)} = e_j(n)\phi'(v_j(n)) \tag{7}$$

where ϕ' is the derivative of the activation function. Calculating the change in weights to a hidden node is more difficult but it can be shown that the relevant derivative is

$$-\frac{\partial \mathscr{E}(n)}{\partial v_j(n)} = \phi'(v_j(n)) \sum_k -\frac{\partial \mathscr{E}(n)}{\partial v_k(n)} w_{kj}(n) \tag{8}$$

which depends on the change in weights of the k^{th} nodes, which represent the output layer. So to change hidden layer weights, one must first change the output layer weights according to the derivative of the activation function, and so this algorithm represents a *back-propagation* of the activation function; hence, this learning algorithm is known as *back-propagation*. (Note that there are quite a few refinements beyond this set of equations that improve the speed of learning.)

The training process requires many iterations and often takes considerable time. A single example of each input pattern may in principle be applied repeatedly but that tends to lead to 'over-training,' where the network recognises only that pattern. More effective training uses a large set of training data; where that is not available, it is common to present noisy, shifted or rotated versions of the images, and to vary the order of presentation. Networks trained in this way tend to be more effective on unseen data because they have learned the types of variations that input of the different classes contain.

The trained network may be 'run' on test data simply and with great speed: the data are applied to the input layer and the outputs of each layer, calculated using the previously-determined weights, are propagated forwards until the final outputs are found. The strength of the outputs indicates the degree of resemblance between the input and the previously-learned patterns.

3 MLP in software

There are quite a few pieces of software freely available that you can download and use to explore different neural networks. The software the author has used most recently are the Python packages `Theano`, `Lasagne` and `nolearn`. The first of these is actually a matrix manipulation package that is able to make use of any GPUs that you may have on your computer. `Lasagne` is a neural network package built on top of `Theano`, providing functionality for both MLP and CNN types of network, and `nolearn` is a layer on top of `Lasagne` that simplifies the definition of the network architecture. If you want to install these on your own machine, do read around on the web before starting because you need to install specific versions of all three packages for them to work together — if you try running an example program and it fails, there is likely to be a version mismatch.

With these installed correctly, an MLP suitable for applying to the MNIST dataset might be defined as follows:

```
net1 = NeuralNet(
    layers = [  # three layers, one of them hidden
        ('input', layers.InputLayer),
        ('hidden', layers.DenseLayer),
        ('output', layers.DenseLayer),
        ],
    # layer parameters:
    input_shape = (None, 784),  # 23 x 23 input pixels per batch
    hidden_num_units = 100,     # number of units in hidden layer
    output_nonlinearity = None, # output layer uses identity function
    output_num_units = 10,      # 10 target values

    # optimization method:
    update = nesterov_momentum,
    update_learning_rate = 0.01,
    update_momentum = 0.9,

    regression = True, # indicate it's a regression problem
    max_epochs = 400,  # train for this many epochs
    verbose = 1,
    )
```

You will see that the input layer is of the same size as the individual MNIST images ($23 \times 23 = 784$), that the hidden layer has 100 neurons, and that there are ten possible outputs. The maximum number of times ("epochs") that the training data are presented is 400.

There is actually a Python MNIST example program distributed as part of `Lasagne`. If you run this with an MLP architecture for 100 generations:

```
./mnist.py mlp 100
```

you should find that the resulting accuracy is about 98.31%. Each training epoch takes about 4.5 seconds on the author's laptop — we shall compare this with the time taken to train a CNN below.

4 Convolutional neural networks

A CNN is not totally different to an MLP, more a kind of refinement of it. Firstly, it is assumed that its inputs are images, and this allows the network to be simplified somewhat. For a conventional MLP, even with the minuscule images in MNIST (only 28×28 pixels), there are 784 weights to be determined. Instead, a CNN consists of layers that can be thought of as having a 3D arrangement of neurons, so they have a height and width that correspond to the dimensions of the image and a 'depth' that corresponds to an *activation volume*. Within each layer, the neurons are connected only to a small region of the layer preceding it. CNN are conventionally constructed from a few different types of layer:

INPUT: this receives the raw pixels of the data; for colour data, the red, green and blue (or HSV) values are normally presented to different input neurons;

CONV: a convolutional or CONV layer computes the output of neurons that are connected to local regions of in the input — in other words, it performs a convolution with coefficients that are learned from the data;

RELU: this applies an element-wise activation function, which may be as simple as `max(0,x)` to threshold at zero — clearly, this leaves the width and height of the network unchanged;

POOL: this down-samples or averages regions of its input, so that the overall width and height of the network is reduced;

FC: the final layer is usually fully-connected, just as in an MLP; it computes the class scores and hence returns the calculated class of the pattern presented at the network's input.

You will see that a CNN transforms its input image, layer by layer, from the original pixel values to the final class scores. CONV and FC layers transform the data as a function of their inputs according to the weights and biases of the neurons, which means they must be trained. Conversely, RELU and POOL layers implement a fixed function and do not require training.

Although this is a generic architecture for CNNs, some specific instances of it have proven successful.

LeNet. The first CNN was *LeNet*, developed by Yann LeCun, the inventor of the CNN [LeCun et al., 1998] (Figure 3). This was used to read handwritten digits, such as the numbers on cheques or the zip-code on envelopes in the USA; indeed, LeCun had a large rôle in popularising MNIST as a standard task for comparing machine learning algorithms. It has the structure

```
INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => RELU => FC
```

which is a pair of `CONV => RELU => POOL` triplets followed by a pair of fully-connected layers with thresholding.

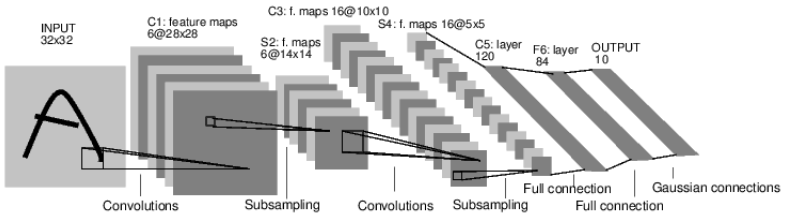

Figure 3: LeNet (image from `http://eblearn.sourceforge.net/beginner_tutorial2_train.html`)

AlexNet. The paper that popularised CNNs in computer vision was AlexNet [Krizhevsky, Sutskever and Hinton, 2012], which presented a solution to the ImageNet Large Scale Visual Recognition Challenge in 2012 and won by a substantial margin. AlexNet is similar to LeNet but features successive CONV layers; previously, it was common to have a single CONV layer

followed by RELU and POOL layers. The architecture is shown in Figure 4, accompanied by the types of feature that the different layers are purported to help identify.

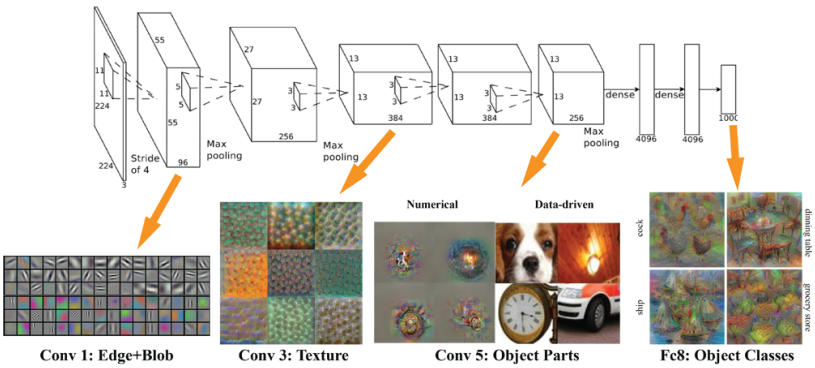

Figure 4: AlexNet (image from http://www.cc.gatech.edu/~hays/compvision/proj6/)

GoogLeNet. The ILSVRC 2014 winner was a CNN from Google [Szegedy et al., 2015]. Its main contribution was an *inception module* that dramatically reduced the number of parameters in the network. Additionally, GoogLeNet uses average pooling instead of FC layers at output stage of the network, eliminating many parameters that seem to have little effect. There are also several followup versions to GoogLeNet, most recently Inception-v4.

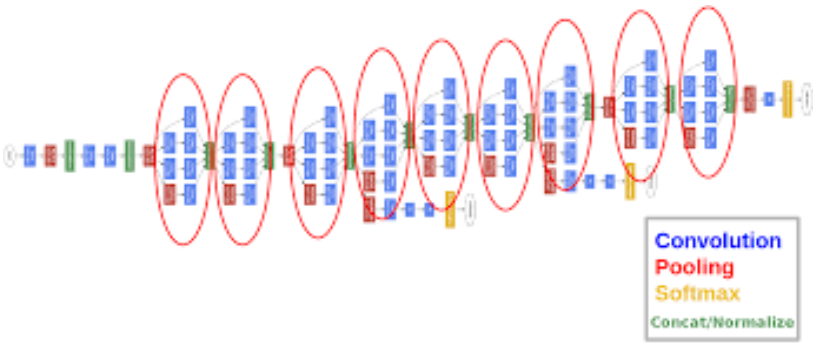

Figure 5: GoogLeNet (image from https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/googlenet.html)

VGGNet. The runner-up in ILSVRC 2014 was the that became known as VGGNet [Simonyan and Zisserman, 2015]. Its main contribution was in showing that the depth of the network is a critical component for good performance. Their final best network contained 16 CONV/FC layers and featured an homogeneous architecture, with only 3×3 convolutions and 2×2 pooling stages — but at the cost of being expensive to evaluate, using more memory and having more parameters.

ResNet. The *Residual Network* of [He et al., 2016] was the winner of ILSVRC 2015. Its designers observed that adding more layers to a neural network tends to increase its training error, and hypothesized that it might be effective to encourage the network to learn the residual error instead

Figure 6: VGGNet (image from http://matthijshollemans.com/2016/08/30/vggnet-convolutional-neural-network-iphone/

of the original mapping. As well as having a ludicrous number of layers, some 200, it features special 'skip' connections and a heavy use of batch normalization in training. The architecture also omits fully connected layers at the end of the network. Nevertheless, due to their superior performance, ResNets are currently regarded as being the state of the art in performance on vision tasks.

My take on this. You will see that all these architectures are variations on a theme. The author's view of this is that the research community is still exploring what these types of architectures do and how they do it. When there is a little more understanding, I fully expect that someone will present a series of stages that are better tailored to performing texture analysis, feature extraction *etc*. and are able to be trained in isolation — as things stand, very few research groups have the resources to train 200-layer CNNs.

There is a trend towards "tapping off" the intermediate results of some CNN stages — for example Figure 4 shows a stage appears to distinguish different textures — and using these as generic feature detectors. The outputs from these stages are then used as input to (say) a SVM, which is trained up on a specific problem. People have reported some successes using this approach. The problem with it, in the author's opinion, is that the stages are not deliberately trained to exhibit these types of behaviour; researchers are avoiding having to find a really good solution to the individual stages by hoping that the neural network will do it for them.

5 CNN in software

The MNIST example program described in Section 3 is also able to train up a CNN:

```
./mnist.py cnn 100
```

It takes about 80 minutes to do this, and the CNN has an accuracy on the MNIST test set of 99.27%, better than the MLP, SVM and the other machine learning (ML) techniques we have reviewed.

Note that there are other CNN implementations in common use, all of which exploit GPU speeds-up where possible. The two that the author hears mentioned most often are Caffe (from Berkeley) and TensorFlow (originally from Google).

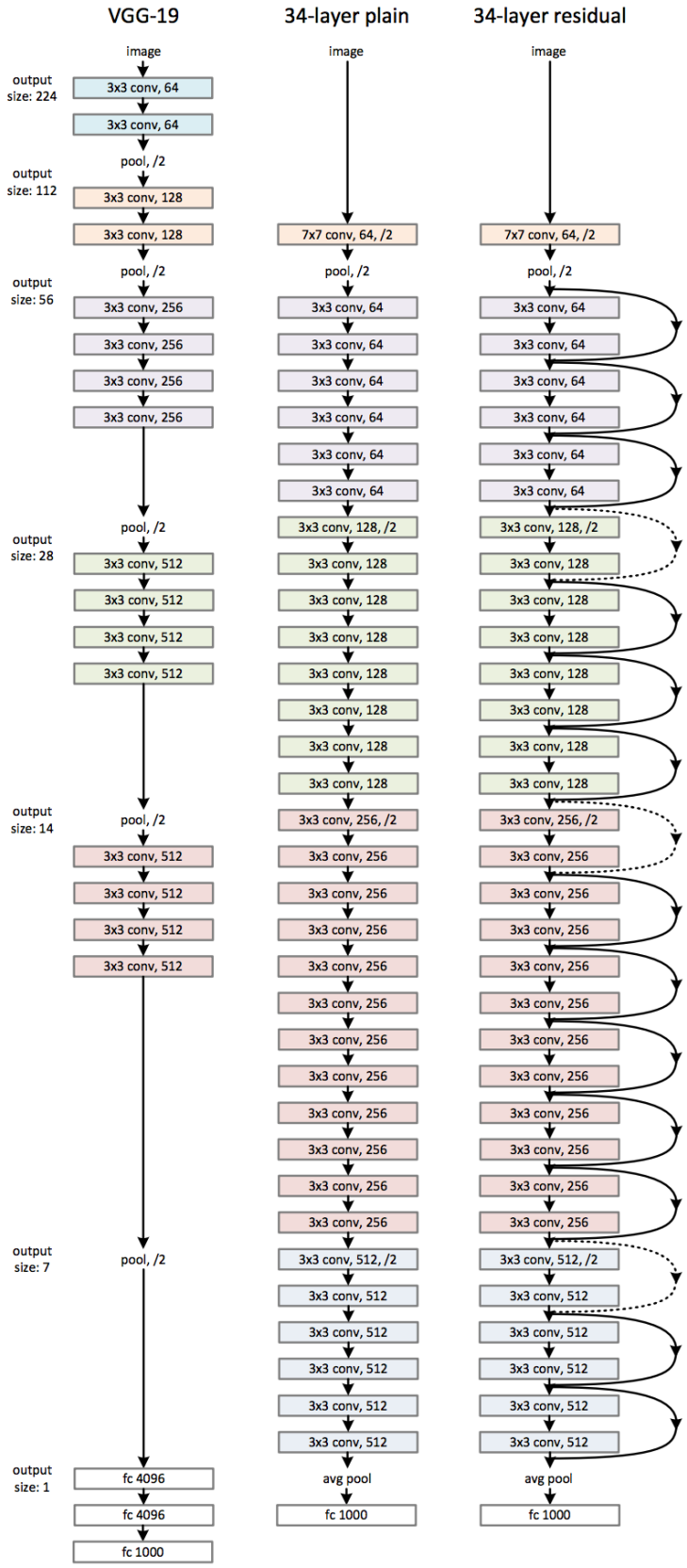

Figure 7: ResNet (image from https://arxiv.org/pdf/1512.03385v1.pdf)

To conclude our examination of neural networks in general and CNN in particular, the following is a complete program (based on code in a Lasagne tutorial) that implements a CNN, trains it on the MNIST dataset (downloading it if necessary), plots the resulting confusion matrix, and displays one of the layers of the network. If you run the program, you will see that the displayed layer of the CNN will be similar to that shown in Figure 8.

```
#!/usr/bin/env python

import matplotlib.pyplot as plt
import matplotlib.cm as cm
from urllib import urlretrieve
import numpy as np
import cPickle as pickle
import os, gzip, theano, lasagne
from lasagne import layers
from lasagne.updates import nesterov_momentum
from nolearn.lasagne import NeuralNet
from nolearn.lasagne import visualize
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

def load_dataset():
    url = 'http://deeplearning.net/data/mnist/mnist.pkl.gz'
    filename = 'mnist.pkl.gz'
    if not os.path.exists(filename):
        print("Downloading MNIST dataset...")
        urlretrieve(url, filename)

    with gzip.open(filename, 'rb') as f:
        data = pickle.load(f)

    X_train, y_train = data[0]
    X_val, y_val = data[1]
    X_test, y_test = data[2]

    X_train = X_train.reshape((-1, 1, 28, 28))
    X_val = X_val.reshape((-1, 1, 28, 28))
    X_test = X_test.reshape((-1, 1, 28, 28))

    y_train = y_train.astype(np.uint8)
    y_val = y_val.astype(np.uint8)
    y_test = y_test.astype(np.uint8)

    return X_train, y_train, X_val, y_val, X_test, y_test


X_train, y_train, X_val, y_val, X_test, y_test = load_dataset()
plt.imshow(X_train[0][0], cmap=cm.binary)

net1 = NeuralNet(
    layers=[('input', layers.InputLayer),
            ('conv2d1', layers.Conv2DLayer),
            ('maxpool1', layers.MaxPool2DLayer),
```

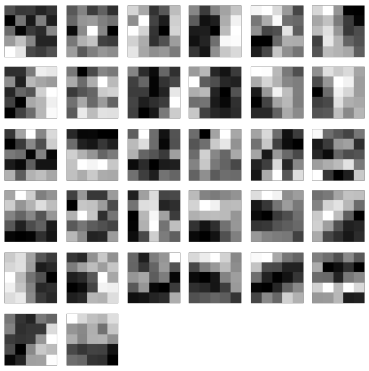

Figure 8: One of the CNN layers. (Image from http://blog.christianperone.com/2015/08/convolutional-neural-networks-and-feature-extract

```
            ('conv2d2', layers.Conv2DLayer),
            ('maxpool2', layers.MaxPool2DLayer),
            ('dropout1', layers.DropoutLayer),
            ('dense', layers.DenseLayer),
            ('dropout2', layers.DropoutLayer),
            ('output', layers.DenseLayer),
            ],
    # Input layer
    input_shape=(None, 1, 28, 28),
    # Layer conv2d1
    conv2d1_num_filters=32,
    conv2d1_filter_size=(5, 5),
    conv2d1_nonlinearity=lasagne.nonlinearities.rectify,
    conv2d1_W=lasagne.init.GlorotUniform(),
    # Layer maxpool1
    maxpool1_pool_size=(2, 2),
    # Layer conv2d2
    conv2d2_num_filters=32,
    conv2d2_filter_size=(5, 5),
    conv2d2_nonlinearity=lasagne.nonlinearities.rectify,
    # Layer maxpool2
    maxpool2_pool_size=(2, 2),
    # Dropout1
    dropout1_p=0.5,
    # Dense
    dense_num_units=256,
    dense_nonlinearity=lasagne.nonlinearities.rectify,
    # Dropout2
    dropout2_p=0.5,
    # Output
    output_nonlinearity=lasagne.nonlinearities.softmax,
    output_num_units=10,
    # Optimization method params
    update=nesterov_momentum,
    update_learning_rate=0.01,
    update_momentum=0.9,
    max_epochs=10,
    verbose=1,
    )

# Train the network.
nn = net1.fit(X_train, y_train)

# Apply the network to the test data and, from its results, determine the
# class confusion matrix.
preds = net1.predict(X_test)
cm = confusion_matrix(y_test, preds)
plt.matshow(cm)
plt.title('Confusion matrix')
plt.colorbar()
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.show()

# Take a look at one of the layers of the network.
visualize.plot_conv_weights(net1.layers_['conv2d1'])
```

References

He, K. et al. (2016). "Deep Residual Learning for Image Recognition". In: *Proceedings of the International Conference on Computer Vision and Pattern Recognition*.

Krizhevsky, Alex, Ilya Sutskever and Geoffrey E Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

LeCun, Y. et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Simonyan, Karen and Andrew Zisserman (2015). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *Proceeedings of the International Conference on Learning Representations*.

Szegedy, Christian et al. (2015). "Going Deeper with Convolutions". In: *Proceedings of the International Conference on Computer Vision and Pattern Recognition*. URL: http://arxiv.org/abs/1409.4842.