Building our own Neural Network code, introduction to Tensorflow

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and Facility for Rare Ion Beams, Michigan State University, USA

Jan 24, 2023

Plan for week 4

- Building our own Feed-forward Neural Network

Reading suggestions: Aurelien Geron's chapters 10-11. For a more in depth discussion on neural networks we recommend Goodfellow et al chapters 6 and 7. For CNNs, see Goodfellow et al chapter 9. chapter 11 and 12 on practicalities and applications. We strongly recommend reading these chapters from Goodfellow et on Deep Learning, that is chapters 6-12. Bishop's chapter 5 on Neural Networks is an additional good read.

Videos on Neural Networks

- Video on Neural Networks
- Video on the back propagation algorithm

Setting up the Back propagation algorithm

We have the following equations:

First, we set up the input data $\boldsymbol{x}$ and the activations $\boldsymbol{z}_1$ of the input layer and compute the activation function and the pertinent outputs $\boldsymbol{a}^1$.

Secondly, we perform then the feed forward till we reach the output layer and compute all $\boldsymbol{z}_l$ of the input layer and compute the activation function and the pertinent outputs $\boldsymbol{a}^l$ for $l = 2, 3, \ldots, L$.

Thereafter we compute the ouput error $\boldsymbol{\delta}^L$ by computing all

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial(a_j^L)}.$$

Then we compute the back propagate error for each $l = L-1, L-2, \ldots, 2$ as

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l).$$

Finally, we update the weights and the biases using gradient descent for each $l = L-1, L-2, \ldots, 2$ and update the weights and biases according to the rules

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1},$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l,$$

The parameter η is the learning parameter discussed in connection with the gradient descent methods. Here it is convenient to use stochastic gradient descent (see the examples below) with mini-batches with an outer loop that steps through multiple epochs of training.

Setting up a Multi-layer perceptron model for classification

We are now gong to develop an example based on the MNIST data base. This is a classification problem and we need to use our cross-entropy function we discussed in connection with logistic regression. The cross-entropy defines our cost function for the classificaton problems with neural networks.

In binary classification with two classes $(0, 1)$ we define the logistic/sigmoid function as the probability that a particular input is in class 0 or 1. This is possible because the logistic function takes any input from the real numbers and inputs a number between 0 and 1, and can therefore be interpreted as a probability. It also has other nice properties, such as a derivative that is simple to calculate.

For an input $\boldsymbol{a}$ from the hidden layer, the probability that the input $\boldsymbol{x}$ is in class 0 or 1 is just. We let θ represent the unknown weights and biases to be adjusted by our equations). The variable x represents our activation values z. We have

$$P(y = 0 \mid \boldsymbol{x}, \boldsymbol{\theta}) = \frac{1}{1 + \exp(-\boldsymbol{x})},$$

and

$$P(y = 1 \mid \boldsymbol{x}, \boldsymbol{\theta}) = 1 - P(y = 0 \mid \boldsymbol{x}, \boldsymbol{\theta}),$$

where $y \in \{0, 1\}$ and $\boldsymbol{\theta}$ represents the weights and biases of our network.

Defining the cost function

Our cost function is given as (see the Logistic regression lectures)

$$\mathcal{C}(\boldsymbol{\theta}) = -\ln P(\mathcal{D} \mid \boldsymbol{\theta}) = -\sum_{i=1}^{n} y_i \ln[P(y_i = 0)] + (1-y_i)\ln[1-P(y_i=0)] = \sum_{i=1}^{n} \mathcal{L}_i(\boldsymbol{\theta}).$$

This last equality means that we can interpret our *cost* function as a sum over the *loss* function for each point in the dataset $\mathcal{L}_i(\boldsymbol{\theta})$. The negative sign is just so that we can think about our algorithm as minimizing a positive number, rather than maximizing a negative number.

In *multiclass* classification it is common to treat each integer label as a so called *one-hot* vector:

$y = 5 \quad \rightarrow \quad \boldsymbol{y} = (0,0,0,0,0,1,0,0,0,0)$, and

$y = 1 \quad \rightarrow \quad \boldsymbol{y} = (0,1,0,0,0,0,0,0,0,0)$,

i.e. a binary bit string of length C, where $C = 10$ is the number of classes in the MNIST dataset (numbers from 0 to 9)..

If $\boldsymbol{x}_i$ is the i-th input (image), y_{ic} refers to the c-th component of the i-th output vector $\boldsymbol{y}_i$. The probability of $\boldsymbol{x}_i$ being in class c will be given by the softmax function:

$$P(y_{ic} = 1 \mid \boldsymbol{x}_i, \boldsymbol{\theta}) = \frac{\exp\left((\boldsymbol{a}_i^{hidden})^T \boldsymbol{w}_c\right)}{\sum_{c'=0}^{C-1} \exp\left((\boldsymbol{a}_i^{hidden})^T \boldsymbol{w}_{c'}\right)},$$

which reduces to the logistic function in the binary case. The likelihood of this C-class classifier is now given as:

$$P(\mathcal{D} \mid \boldsymbol{\theta}) = \prod_{i=1}^{n} \prod_{c=0}^{C-1} [P(y_{ic} = 1)]^{y_{ic}}.$$

Again we take the negative log-likelihood to define our cost function:

$$\mathcal{C}(\boldsymbol{\theta}) = -\log P(\mathcal{D} \mid \boldsymbol{\theta}).$$

See the logistic regression lectures for a full definition of the cost function.

The back propagation equations need now only a small change, namely the definition of a new cost function. We are thus ready to use the same equations as before!

Example: binary classification problem

As an example of the above, relevant for project 2 as well, let us consider a binary class. As discussed in our logistic regression lectures, we defined a cost function in terms of the parameters β as

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(y_i \log p(y_i|x_i, \boldsymbol{\beta}) + (1-y_i) \log 1 - p(y_i|x_i, \boldsymbol{\beta}) \right),$$

where we had defined the logistic (sigmoid) function

$$p(y_i = 1|x_i, \boldsymbol{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)},$$

and

$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta}).$$

The parameters $\boldsymbol{\beta}$ were defined using a minimization method like gradient descent or Newton-Raphson's method.

Now we replace x_i with the activation z_i^l for a given layer l and the outputs as $y_i = a_i^l = f(z_i^l)$, with z_i^l now being a function of the weights w_{ij}^l and biases b_i^l. We have then

$$a_i^l = y_i = \frac{\exp(z_i^l)}{1 + \exp(z_i^l)},$$

with

$$z_i^l = \sum_j w_{ij}^l a_j^{l-1} + b_i^l,$$

where the superscript $l-1$ indicates that these are the outputs from layer $l-1$. Our cost function at the final layer $l = L$ is now

$$\mathcal{C}(\boldsymbol{W}) = -\sum_{i=1}^{n} \left(t_i \log a_i^L + (1 - t_i) \log (1 - a_i^L) \right),$$

where we have defined the targets t_i. The derivatives of the cost function with respect to the output a_i^L are then easily calculated and we get

$$\frac{\partial \mathcal{C}(\boldsymbol{W})}{\partial a_i^L} = \frac{a_i^L - t_i}{a_i^L(1 - a_i^L)}.$$

In case we use another activation function than the logistic one, we need to evaluate other derivatives.

The Softmax function

In case we employ the more general case given by the Softmax equation, we need to evaluate the derivative of the activation function with respect to the activation z_i^l, that is we need

$$\frac{\partial f(z_i^l)}{\partial w_{jk}^l} = \frac{\partial f(z_i^l)}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial f(z_i^l)}{\partial z_j^l} a_k^{l-1}.$$

For the Softmax function we have

$$f(z_i^l) = \frac{\exp(z_i^l)}{\sum_{m=1}^{K} \exp(z_m^l)}.$$

Its derivative with respect to z_j^l gives

$$\frac{\partial f(z_i^l)}{\partial z_j^l} = f(z_i^l)\left(\delta_{ij} - f(z_j^l)\right),$$

which in case of the simply binary model reduces to having $i = j$.

Developing a code for doing neural networks with back propagation

One can identify a set of key steps when using neural networks to solve supervised learning problems:

1. Collect and pre-process data
2. Define model and architecture
3. Choose cost function and optimizer
4. Train the model
5. Evaluate model performance on test data
6. Adjust hyperparameters (if necessary, network architecture)

Collect and pre-process data

Here we will be using the MNIST dataset, which is readily available through the **scikit-learn** package. You may also find it for example here. The *MNIST* (Modified National Institute of Standards and Technology) database is a large database of handwritten digits that is commonly used for training various image processing systems. The MNIST dataset consists of 70 000 images of size 28×28 pixels, each labeled from 0 to 9. The scikit-learn dataset we will use consists of a selection of 1797 images of size 8×8 collected and processed from this database.

To feed data into a feed-forward neural network we need to represent the inputs as a design/feature matrix $X = (n_{inputs}, n_{features})$. Each row represents an *input*, in this case a handwritten digit, and each column represents a *feature*, in this case a pixel. The correct answers, also known as *labels* or *targets* are represented as a 1D array of integers $Y = (n_{inputs}) = (5, 3, 1, 8, ...)$.

As an example, say we want to build a neural network using supervised learning to predict Body-Mass Index (BMI) from measurements of height (in m) and weight (in kg). If we have measurements of 5 people the design/feature matrix could be for example:

$$X = \begin{bmatrix} 1.85\&81 \\ 1.71\&65 \\ 1.95\&103 \\ 1.55\&42 \\ 1.63\&56 \end{bmatrix},$$

and the targets would be:

$$Y = (23.7, 22.2, 27.1, 17.5, 21.1)$$

Since each input image is a 2D matrix, we need to flatten the image (i.e. "unravel" the 2D matrix into a 1D array) to turn the data into a design/feature matrix. This means we lose all spatial information in the image, such as locality and translational invariance. More complicated architectures such as Convolutional Neural Networks can take advantage of such information, and are most commonly applied when analyzing images.

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets


# ensure the same random numbers appear every time
np.random.seed(0)

# display images in notebook
%matplotlib inline
plt.rcParams['figure.figsize'] = (12,12)


# download MNIST dataset
digits = datasets.load_digits()

# define inputs and labels
inputs = digits.images
labels = digits.target

print("inputs = (n_inputs, pixel_width, pixel_height) = " + str(inputs.shape))
print("labels = (n_inputs) = " + str(labels.shape))


# flatten the image
# the value -1 means dimension is inferred from the remaining dimensions: 8x8 = 64
n_inputs = len(inputs)
inputs = inputs.reshape(n_inputs, -1)
print("X = (n_inputs, n_features) = " + str(inputs.shape))


# choose some random images to display
indices = np.arange(n_inputs)
random_indices = np.random.choice(indices, size=5)

for i, image in enumerate(digits.images[random_indices]):
    plt.subplot(1, 5, i+1)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title("Label: %d" % digits.target[random_indices[i]])
plt.show()
```

Train and test datasets

Performing analysis before partitioning the dataset is a major error, that can lead to incorrect conclusions.

We will reserve 80% of our dataset for training and 20% for testing.

It is important that the train and test datasets are drawn randomly from our dataset, to ensure no bias in the sampling. Say you are taking measurements of weather data to predict the weather in the coming 5 days. You don't want to train your model on measurements taken from the hours 00.00 to 12.00, and then test it on data collected from 12.00 to 24.00.

```
from sklearn.model_selection import train_test_split

# one-liner from scikit-learn library
train_size = 0.8
test_size = 1 - train_size
X_train, X_test, Y_train, Y_test = train_test_split(inputs, labels, train_size=train_size,
                                                    test_size=test_size)

# equivalently in numpy
def train_test_split_numpy(inputs, labels, train_size, test_size):
    n_inputs = len(inputs)
    inputs_shuffled = inputs.copy()
    labels_shuffled = labels.copy()

    np.random.shuffle(inputs_shuffled)
    np.random.shuffle(labels_shuffled)

    train_end = int(n_inputs*train_size)
    X_train, X_test = inputs_shuffled[:train_end], inputs_shuffled[train_end:]
    Y_train, Y_test = labels_shuffled[:train_end], labels_shuffled[train_end:]

    return X_train, X_test, Y_train, Y_test

#X_train, X_test, Y_train, Y_test = train_test_split_numpy(inputs, labels, train_size, test_size)

print("Number of training images: " + str(len(X_train)))
print("Number of test images: " + str(len(X_test)))
```

Define model and architecture

Our simple feed-forward neural network will consist of an *input* layer, a single *hidden* layer and an *output* layer. The activation y of each neuron is a weighted sum of inputs, passed through an activation function. In case of the simple perceptron model we have

$$z = \sum_{i=1}^{n} w_i a_i,$$

$$y = f(z),$$

where f is the activation function, a_i represents input from neuron i in the preceding layer and w_i is the weight to input i. The activation of the neurons in the input layer is just the features (e.g. a pixel value).

The simplest activation function for a neuron is the *Heaviside* function:

$$f(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

A feed-forward neural network with this activation is known as a *perceptron*. For a binary classifier (i.e. two classes, 0 or 1, dog or not-dog) we can also use this in our output layer. This activation can be generalized to k classes (using e.g. the *one-against-all* strategy), and we call these architectures *multiclass perceptrons*.

However, it is now common to use the terms Single Layer Perceptron (SLP) (1 hidden layer) and Multilayer Perceptron (MLP) (2 or more hidden layers) to refer to feed-forward neural networks with any activation function.

Typical choices for activation functions include the sigmoid function, hyperbolic tangent, and Rectified Linear Unit (ReLU). We will be using the sigmoid function $\sigma(x)$:

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}},$$

which is inspired by probability theory (see logistic regression) and was most commonly used until about 2011. See the discussion below concerning other activation functions.

Layers

- Input

Since each input image has 8x8 = 64 pixels or features, we have an input layer of 64 neurons.

- Hidden layer

We will use 50 neurons in the hidden layer receiving input from the neurons in the input layer. Since each neuron in the hidden layer is connected to the 64 inputs we have 64x50 = 3200 weights to the hidden layer.

- Output

If we were building a binary classifier, it would be sufficient with a single neuron in the output layer, which could output 0 or 1 according to the Heaviside function. This would be an example of a *hard* classifier, meaning it outputs the class of the input directly. However, if we are dealing with noisy data it is often beneficial to use a *soft* classifier, which outputs the probability of being in class 0 or 1.

For a soft binary classifier, we could use a single neuron and interpret the output as either being the probability of being in class 0 or the probability of being in class 1. Alternatively we could use 2 neurons, and interpret each neuron as the probability of being in each class.

Since we are doing multiclass classification, with 10 categories, it is natural to use 10 neurons in the output layer. We number the neurons $j = 0, 1, ..., 9$. The activation of each output neuron j will be according to the *softmax* function:

$$P(\text{class } j \mid \text{input } \boldsymbol{a}) = \frac{\exp\left(\boldsymbol{a}^T \boldsymbol{w}_j\right)}{\sum_{c=0}^{9} \exp\left(\boldsymbol{a}^T \boldsymbol{w}_c\right)},$$

i.e. each neuron j outputs the probability of being in class j given an input from the hidden layer $\boldsymbol{a}$, with $\boldsymbol{w}_j$ the weights of neuron j to the inputs. The denominator is a normalization factor to ensure the outputs (probabilities) sum up to 1. The exponent is just the weighted sum of inputs as before:

$$z_j = \sum_{i=1}^{n} w_{ij} a_i + b_j.$$

Since each neuron in the output layer is connected to the 50 inputs from the hidden layer we have 50x10 = 500 weights to the output layer.

Weights and biases

Typically weights are initialized with small values distributed around zero, drawn from a uniform or normal distribution. Setting all weights to zero means all neurons give the same output, making the network useless.

Adding a bias value to the weighted sum of inputs allows the neural network to represent a greater range of values. Without it, any input with the value 0 will be mapped to zero (before being passed through the activation). The bias unit has an output of 1, and a weight to each neuron j, b_j:

$$z_j = \sum_{i=1}^{n} w_{ij} a_i + b_j.$$

The bias weights $\boldsymbol{b}$ are often initialized to zero, but a small value like 0.01 ensures all neurons have some output which can be backpropagated in the first training cycle.

```
# building our neural network

n_inputs, n_features = X_train.shape
n_hidden_neurons = 50
n_categories = 10

# we make the weights normally distributed using numpy.random.randn

# weights and bias in the hidden layer
hidden_weights = np.random.randn(n_features, n_hidden_neurons)
hidden_bias = np.zeros(n_hidden_neurons) + 0.01

# weights and bias in the output layer
output_weights = np.random.randn(n_hidden_neurons, n_categories)
output_bias = np.zeros(n_categories) + 0.01
```

Feed-forward pass

Denote F the number of features, H the number of hidden neurons and C the number of categories. For each input image we calculate a weighted sum of input features (pixel values) to each neuron j in the hidden layer l:

$$z_j^l = \sum_{i=1}^{F} w_{ij}^l x_i + b_j^l,$$

this is then passed through our activation function

$$a_j^l = f(z_j^l).$$

We calculate a weighted sum of inputs (activations in the hidden layer) to each neuron j in the output layer:

$$z_j^L = \sum_{i=1}^{H} w_{ij}^L a_i^l + b_j^L.$$

Finally we calculate the output of neuron j in the output layer using the softmax function:

$$a_j^L = \frac{\exp\left(z_j^L\right)}{\sum_{c=0}^{C-1} \exp\left(z_c^L\right)}.$$

Matrix multiplications

Since our data has the dimensions $X = (n_{inputs}, n_{features})$ and our weights to the hidden layer have the dimensions $W_{hidden} = (n_{features}, n_{hidden})$, we can easily feed the network all our training data in one go by taking the matrix product

$$XW^h = (n_{inputs}, n_{hidden}),$$

and obtain a matrix that holds the weighted sum of inputs to the hidden layer for each input image and each hidden neuron. We also add the bias to obtain a matrix of weighted sums to the hidden layer Z^h:

$$\boldsymbol{z}^l = \boldsymbol{X}\boldsymbol{W}^l + \boldsymbol{b}^l,$$

meaning the same bias (1D array with size equal number of hidden neurons) is added to each input image. This is then passed through the activation:

$$\boldsymbol{a}^l = f(\boldsymbol{z}^l).$$

This is fed to the output layer:

$$\boldsymbol{z}^L = \boldsymbol{a}^L\boldsymbol{W}^L + \boldsymbol{b}^L.$$

Finally we receive our output values for each image and each category by passing it through the softmax function:

$$output = softmax(\boldsymbol{z}^L) = (n_{inputs}, n_{categories}).$$

```
# setup the feed-forward pass, subscript h = hidden layer

def sigmoid(x):
    return 1/(1 + np.exp(-x))

def feed_forward(X):
    # weighted sum of inputs to the hidden layer
    z_h = np.matmul(X, hidden_weights) + hidden_bias
    # activation in the hidden layer
    a_h = sigmoid(z_h)

    # weighted sum of inputs to the output layer
    z_o = np.matmul(a_h, output_weights) + output_bias
    # softmax output
    # axis 0 holds each input and axis 1 the probabilities of each category
    exp_term = np.exp(z_o)
    probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)

    return probabilities

probabilities = feed_forward(X_train)
print("probabilities = (n_inputs, n_categories) = " + str(probabilities.shape))
print("probability that image 0 is in category 0,1,2,...,9 = \n" + str(probabilities[0]))
print("probabilities sum up to: " + str(probabilities[0].sum()))
print()

# we obtain a prediction by taking the class with the highest likelihood
def predict(X):
    probabilities = feed_forward(X)
    return np.argmax(probabilities, axis=1)

predictions = predict(X_train)
print("predictions = (n_inputs) = " + str(predictions.shape))
print("prediction for image 0: " + str(predictions[0]))
print("correct label for image 0: " + str(Y_train[0]))
```

Choose cost function and optimizer

To measure how well our neural network is doing we need to introduce a cost function. We will call the function that gives the error of a single sample output the *loss* function, and the function that gives the total error of our network across all samples the *cost* function. A typical choice for multiclass classification is the *cross-entropy* loss, also known as the negative log likelihood.

In *multiclass* classification it is common to treat each integer label as a so called *one-hot* vector:

$$y = 5 \quad \rightarrow \quad \boldsymbol{y} = (0, 0, 0, 0, 0, 1, 0, 0, 0, 0),$$

$$y = 1 \quad \rightarrow \quad \boldsymbol{y} = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0),$$

i.e. a binary bit string of length C, where $C = 10$ is the number of classes in the MNIST dataset.

Let y_{ic} denote the c-th component of the i-th one-hot vector. We define the cost function $\mathcal{C}$ as a sum over the cross-entropy loss for each point $\boldsymbol{x}_i$ in the dataset.

In the one-hot representation only one of the terms in the loss function is non-zero, namely the probability of the correct category c' (i.e. the category c' such that $y_{ic'} = 1$). This means that the cross entropy loss only punishes you for how wrong you got the correct label. The probability of category c is given by the softmax function. The vector $\boldsymbol{\theta}$ represents the parameters of our network, i.e. all the weights and biases.

Optimizing the cost function

The network is trained by finding the weights and biases that minimize the cost function. One of the most widely used classes of methods is *gradient descent* and its generalizations. The idea behind gradient descent is simply to adjust the weights in the direction where the gradient of the cost function is large and negative. This ensures we flow toward a *local* minimum of the cost function. Each parameter θ is iteratively adjusted according to the rule

$$\theta_{i+1} = \theta_i - \eta \nabla \mathcal{C}(\theta_i),$$

where η is known as the *learning rate*, which controls how big a step we take towards the minimum. This update can be repeated for any number of iterations, or until we are satisfied with the result.

A simple and effective improvement is a variant called *Batch Gradient Descent*. Instead of calculating the gradient on the whole dataset, we calculate an approximation of the gradient on a subset of the data called a *minibatch*. If there are N data points and we have a minibatch size of M, the total number of batches is N/M. We denote each minibatch B_k, with $k = 1, 2, ..., N/M$. The gradient then becomes:

$$\nabla \mathcal{C}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{L}_i(\theta) \quad \rightarrow \quad \frac{1}{M} \sum_{i \in B_k} \nabla \mathcal{L}_i(\theta),$$

i.e. instead of averaging the loss over the entire dataset, we average over a minibatch.

This has two important benefits:

1. Introducing stochasticity decreases the chance that the algorithm becomes stuck in a local minima.

2. It significantly speeds up the calculation, since we do not have to use the entire dataset to calculate the gradient.

The various optmization methods, with codes and algorithms, are discussed in our lectures on Gradient descent approaches.

Regularization

It is common to add an extra term to the cost function, proportional to the size of the weights. This is equivalent to constraining the size of the weights, so that they do not grow out of control. Constraining the size of the weights means that the weights cannot grow arbitrarily large to fit the training data, and in this way reduces *overfitting*.

We will measure the size of the weights using the so called *L2-norm*, meaning our cost function becomes:

$$\mathcal{C}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\theta) \quad \rightarrow \quad \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\theta) + \lambda||\boldsymbol{w}||_2^2 = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(\theta) + \lambda\sum_{ij} w_{ij}^2,$$

i.e. we sum up all the weights squared. The factor λ is known as a regularization parameter.

In order to train the model, we need to calculate the derivative of the cost function with respect to every bias and weight in the network. In total our network has $(64+1)\times 50 = 3250$ weights in the hidden layer and $(50+1)\times 10 = 510$ weights to the output layer ($+1$ for the bias), and the gradient must be calculated for every parameter. We use the *backpropagation* algorithm discussed above. This is a clever use of the chain rule that allows us to calculate the gradient efficently.

Matrix multiplication

To more efficently train our network these equations are implemented using matrix operations. The error in the output layer is calculated simply as, with $\boldsymbol{t}$ being our targets,

$$\delta_L = \boldsymbol{t} - \boldsymbol{y} = (n_{inputs}, n_{categories}).$$

The gradient for the output weights is calculated as

$$\nabla W_L = \boldsymbol{a}^T\delta_L = (n_{hidden}, n_{categories}),$$

where $\boldsymbol{a} = (n_{inputs}, n_{hidden})$. This simply means that we are summing up the gradients for each input. Since we are going backwards we have to transpose the activation matrix.

The gradient with respect to the output bias is then

$$\nabla \boldsymbol{b}_L = \sum_{i=1}^{n_{inputs}} \delta_L = (n_{categories}).$$

The error in the hidden layer is

$$\Delta_h = \delta_L W_L^T \circ f'(z_h) = \delta_L W_L^T \circ a_h \circ (1 - a_h) = (n_{inputs}, n_{hidden}),$$

where $f'(a_h)$ is the derivative of the activation in the hidden layer. The matrix products mean that we are summing up the products for each neuron in the output layer. The symbol $\circ$ denotes the *Hadamard product*, meaning element-wise multiplication.

This again gives us the gradients in the hidden layer:

$$\nabla W_h = X^T \delta_h = (n_{features}, n_{hidden}),$$

$$\nabla b_h = \sum_{i=1}^{n_{inputs}} \delta_h = (n_{hidden}).$$

```
# to categorical turns our integer vector into a onehot representation
from sklearn.metrics import accuracy_score

# one-hot in numpy
def to_categorical_numpy(integer_vector):
    n_inputs = len(integer_vector)
    n_categories = np.max(integer_vector) + 1
    onehot_vector = np.zeros((n_inputs, n_categories))
    onehot_vector[range(n_inputs), integer_vector] = 1

    return onehot_vector

#Y_train_onehot, Y_test_onehot = to_categorical(Y_train), to_categorical(Y_test)
Y_train_onehot, Y_test_onehot = to_categorical_numpy(Y_train), to_categorical_numpy(Y_test)

def feed_forward_train(X):
    # weighted sum of inputs to the hidden layer
    z_h = np.matmul(X, hidden_weights) + hidden_bias
    # activation in the hidden layer
    a_h = sigmoid(z_h)

    # weighted sum of inputs to the output layer
    z_o = np.matmul(a_h, output_weights) + output_bias
    # softmax output
    # axis 0 holds each input and axis 1 the probabilities of each category
    exp_term = np.exp(z_o)
    probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)

    # for backpropagation need activations in hidden and output layers
    return a_h, probabilities

def backpropagation(X, Y):
    a_h, probabilities = feed_forward_train(X)

    # error in the output layer
    error_output = probabilities - Y
    # error in the hidden layer
    error_hidden = np.matmul(error_output, output_weights.T) * a_h * (1 - a_h)

    # gradients for the output layer
    output_weights_gradient = np.matmul(a_h.T, error_output)
    output_bias_gradient = np.sum(error_output, axis=0)

    # gradient for the hidden layer
    hidden_weights_gradient = np.matmul(X.T, error_hidden)
    hidden_bias_gradient = np.sum(error_hidden, axis=0)
```

```
    return output_weights_gradient, output_bias_gradient, hidden_weights_gradient, hidden_bias_gra

print("Old accuracy on training data: " + str(accuracy_score(predict(X_train), Y_train)))

eta = 0.01
lmbd = 0.01
for i in range(1000):
    # calculate gradients
    dWo, dBo, dWh, dBh = backpropagation(X_train, Y_train_onehot)

    # regularization term gradients
    dWo += lmbd * output_weights
    dWh += lmbd * hidden_weights

    # update weights and biases
    output_weights -= eta * dWo
    output_bias -= eta * dBo
    hidden_weights -= eta * dWh
    hidden_bias -= eta * dBh

print("New accuracy on training data: " + str(accuracy_score(predict(X_train), Y_train)))
```

Improving performance

As we can see the network does not seem to be learning at all. It seems to be just guessing the label for each image. In order to obtain a network that does something useful, we will have to do a bit more work.

The choice of *hyperparameters* such as learning rate and regularization parameter is hugely influential for the performance of the network. Typically a *grid-search* is performed, wherein we test different hyperparameters separated by orders of magnitude. For example we could test the learning rates $\eta = 10^{-6}, 10^{-5}, ..., 10^{-1}$ with different regularization parameters $\lambda = 10^{-6}, ..., 10^{-0}$.

Next, we haven't implemented minibatching yet, which introduces stochasticity and is though to act as an important regularizer on the weights. We call a feed-forward + backward pass with a minibatch an *iteration*, and a full training period going through the entire dataset (n/M batches) an *epoch*.

If this does not improve network performance, you may want to consider altering the network architecture, adding more neurons or hidden layers. Andrew Ng goes through some of these considerations in this video. You can find a summary of the video here.

Full object-oriented implementation

It is very natural to think of the network as an object, with specific instances of the network being realizations of this object with different hyperparameters. An implementation using Python classes provides a clean structure and interface, and the full implementation of our neural network is given below.

```
class NeuralNetwork:
    def __init__(
```

```
        self,
        X_data,
        Y_data,
        n_hidden_neurons=50,
        n_categories=10,
        epochs=10,
        batch_size=100,
        eta=0.1,
        lmbd=0.0):

    self.X_data_full = X_data
    self.Y_data_full = Y_data

    self.n_inputs = X_data.shape[0]
    self.n_features = X_data.shape[1]
    self.n_hidden_neurons = n_hidden_neurons
    self.n_categories = n_categories

    self.epochs = epochs
    self.batch_size = batch_size
    self.iterations = self.n_inputs // self.batch_size
    self.eta = eta
    self.lmbd = lmbd

    self.create_biases_and_weights()

def create_biases_and_weights(self):
    self.hidden_weights = np.random.randn(self.n_features, self.n_hidden_neurons)
    self.hidden_bias = np.zeros(self.n_hidden_neurons) + 0.01

    self.output_weights = np.random.randn(self.n_hidden_neurons, self.n_categories)
    self.output_bias = np.zeros(self.n_categories) + 0.01

def feed_forward(self):
    # feed-forward for training
    self.z_h = np.matmul(self.X_data, self.hidden_weights) + self.hidden_bias
    self.a_h = sigmoid(self.z_h)

    self.z_o = np.matmul(self.a_h, self.output_weights) + self.output_bias

    exp_term = np.exp(self.z_o)
    self.probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)

def feed_forward_out(self, X):
    # feed-forward for output
    z_h = np.matmul(X, self.hidden_weights) + self.hidden_bias
    a_h = sigmoid(z_h)

    z_o = np.matmul(a_h, self.output_weights) + self.output_bias

    exp_term = np.exp(z_o)
    probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)
    return probabilities

def backpropagation(self):
    error_output = self.probabilities - self.Y_data
    error_hidden = np.matmul(error_output, self.output_weights.T) * self.a_h * (1 - self.a_h)

    self.output_weights_gradient = np.matmul(self.a_h.T, error_output)
    self.output_bias_gradient = np.sum(error_output, axis=0)
```

```
        self.hidden_weights_gradient = np.matmul(self.X_data.T, error_hidden)
        self.hidden_bias_gradient = np.sum(error_hidden, axis=0)

        if self.lmbd > 0.0:
            self.output_weights_gradient += self.lmbd * self.output_weights
            self.hidden_weights_gradient += self.lmbd * self.hidden_weights

        self.output_weights -= self.eta * self.output_weights_gradient
        self.output_bias -= self.eta * self.output_bias_gradient
        self.hidden_weights -= self.eta * self.hidden_weights_gradient
        self.hidden_bias -= self.eta * self.hidden_bias_gradient

    def predict(self, X):
        probabilities = self.feed_forward_out(X)
        return np.argmax(probabilities, axis=1)

    def predict_probabilities(self, X):
        probabilities = self.feed_forward_out(X)
        return probabilities

    def train(self):
        data_indices = np.arange(self.n_inputs)

        for i in range(self.epochs):
            for j in range(self.iterations):
                # pick datapoints with replacement
                chosen_datapoints = np.random.choice(
                    data_indices, size=self.batch_size, replace=False
                )

                # minibatch training data
                self.X_data = self.X_data_full[chosen_datapoints]
                self.Y_data = self.Y_data_full[chosen_datapoints]

                self.feed_forward()
                self.backpropagation()
```

Evaluate model performance on test data

To measure the performance of our network we evaluate how well it does it data it has never seen before, i.e. the test data. We measure the performance of the network using the *accuracy* score. The accuracy is as you would expect just the number of images correctly labeled divided by the total number of images. A perfect classifier will have an accuracy score of 1.

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(\tilde{y}_i = y_i)}{n},$$

where I is the indicator function, 1 if $\tilde{y}_i = y_i$ and 0 otherwise.

```
epochs = 100
batch_size = 100

dnn = NeuralNetwork(X_train, Y_train_onehot, eta=eta, lmbd=lmbd, epochs=epochs, batch_size=batch_s
                    n_hidden_neurons=n_hidden_neurons, n_categories=n_categories)
dnn.train()
test_predict = dnn.predict(X_test)
```

```
# accuracy score from scikit library
print("Accuracy score on test set: ", accuracy_score(Y_test, test_predict))

# equivalent in numpy
def accuracy_score_numpy(Y_test, Y_pred):
    return np.sum(Y_test == Y_pred) / len(Y_test)

#print("Accuracy score on test set: ", accuracy_score_numpy(Y_test, test_predict))
```

Adjust hyperparameters

We now perform a grid search to find the optimal hyperparameters for the network. Note that we are only using 1 layer with 50 neurons, and human performance is estimated to be around 98% (2% error rate).

```
eta_vals = np.logspace(-5, 1, 7)
lmbd_vals = np.logspace(-5, 1, 7)
# store the models for later use
DNN_numpy = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

# grid search
for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        dnn = NeuralNetwork(X_train, Y_train_onehot, eta=eta, lmbd=lmbd, epochs=epochs, batch_size
                            n_hidden_neurons=n_hidden_neurons, n_categories=n_categories)
        dnn.train()

        DNN_numpy[i][j] = dnn

        test_predict = dnn.predict(X_test)

        print("Learning rate  = ", eta)
        print("Lambda = ", lmbd)
        print("Accuracy score on test set: ", accuracy_score(Y_test, test_predict))
        print()
```

Visualization

```
# visual representation of grid search
# uses seaborn heatmap, you can also do this with matplotlib imshow
import seaborn as sns

sns.set()

train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))

for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        dnn = DNN_numpy[i][j]

        train_pred = dnn.predict(X_train)
        test_pred = dnn.predict(X_test)

        train_accuracy[i][j] = accuracy_score(Y_train, train_pred)
```

```
            test_accuracy[i][j] = accuracy_score(Y_test, test_pred)


fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(train_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Training Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()

fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(test_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Test Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()
```

scikit-learn implementation

scikit-learn focuses more on traditional machine learning methods, such as regression, clustering, decision trees, etc. As such, it has only two types of neural networks: Multi Layer Perceptron outputting continuous values, *MPLRegressor*, and Multi Layer Perceptron outputting labels, *MLPClassifier*. We will see how simple it is to use these classes.

scikit-learn implements a few improvements from our neural network, such as early stopping, a varying learning rate, different optimization methods, etc. We would therefore expect a better performance overall.

```
from sklearn.neural_network import MLPClassifier
# store models for later use
DNN_scikit = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        dnn = MLPClassifier(hidden_layer_sizes=(n_hidden_neurons), activation='logistic',
                            alpha=lmbd, learning_rate_init=eta, max_iter=epochs)
        dnn.fit(X_train, Y_train)

        DNN_scikit[i][j] = dnn

        print("Learning rate  = ", eta)
        print("Lambda = ", lmbd)
        print("Accuracy score on test set: ", dnn.score(X_test, Y_test))
        print()
```

Visualization

```
# optional
# visual representation of grid search
# uses seaborn heatmap, could probably do this in matplotlib
import seaborn as sns

sns.set()
```

```
train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))

for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        dnn = DNN_scikit[i][j]

        train_pred = dnn.predict(X_train)
        test_pred = dnn.predict(X_test)

        train_accuracy[i][j] = accuracy_score(Y_train, train_pred)
        test_accuracy[i][j] = accuracy_score(Y_test, test_pred)


fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(train_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Training Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()

fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(test_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Test Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()
```

Testing our code for the XOR, OR and AND gates

Let us now discuss three different types of gates, the so-called XOR, the OR and the AND gates. Their inputs and outputs can be summarized using the following tables, first for the OR gate with inputs x_1 and x_2 and outputs y:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

The AND and XOR Gates

The AND gate is defined as

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

And finally we have the XOR gate

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Representing the Data Sets

Our design matrix is defined by the input values x_1 and x_2. Since we have four possible outputs, our design matrix reads

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix},$$

while the vector of outputs is $\boldsymbol{y}^T = [0,1,1,0]$ for the XOR gate, $\boldsymbol{y}^T = [0,0,0,1]$ for the AND gate and $\boldsymbol{y}^T = [0,1,1,1]$ for the OR gate.

Setting up the Neural Network

We define first our design matrix and the various output vectors for the different gates.

```
"""
Simple code that tests XOR, OR and AND gates with linear regression
"""

# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

def sigmoid(x):
    return 1/(1 + np.exp(-x))

def feed_forward(X):
    # weighted sum of inputs to the hidden layer
    z_h = np.matmul(X, hidden_weights) + hidden_bias
    # activation in the hidden layer
    a_h = sigmoid(z_h)

    # weighted sum of inputs to the output layer
    z_o = np.matmul(a_h, output_weights) + output_bias
    # softmax output
    # axis 0 holds each input and axis 1 the probabilities of each category
    probabilities = sigmoid(z_o)
    return probabilities

# we obtain a prediction by taking the class with the highest likelihood
def predict(X):
    probabilities = feed_forward(X)
    return np.argmax(probabilities, axis=1)

# ensure the same random numbers appear every time
```

```
np.random.seed(0)

# Design matrix
X = np.array([ [0, 0], [0, 1], [1, 0],[1, 1]],dtype=np.float64)

# The XOR gate
yXOR = np.array( [ 0, 1 ,1, 0])
# The OR gate
yOR = np.array( [ 0, 1 ,1, 1])
# The AND gate
yAND = np.array( [ 0, 0 ,0, 1])

# Defining the neural network
n_inputs, n_features = X.shape
n_hidden_neurons = 2
n_categories = 2
n_features = 2

# we make the weights normally distributed using numpy.random.randn

# weights and bias in the hidden layer
hidden_weights = np.random.randn(n_features, n_hidden_neurons)
hidden_bias = np.zeros(n_hidden_neurons) + 0.01

# weights and bias in the output layer
output_weights = np.random.randn(n_hidden_neurons, n_categories)
output_bias = np.zeros(n_categories) + 0.01

probabilities = feed_forward(X)
print(probabilities)


predictions = predict(X)
print(predictions)
```

Not an impressive result, but this was our first forward pass with randomly assigned weights. Let us now add the full network with the back-propagation algorithm discussed above.

The Code using Scikit-Learn

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
import seaborn as sns

# ensure the same random numbers appear every time
np.random.seed(0)

# Design matrix
X = np.array([ [0, 0], [0, 1], [1, 0],[1, 1]],dtype=np.float64)

# The XOR gate
yXOR = np.array( [ 0, 1 ,1, 0])
# The OR gate
```

```
yOR = np.array( [ 0, 1 ,1, 1])
# The AND gate
yAND = np.array( [ 0, 0 ,0, 1])

# Defining the neural network
n_inputs, n_features = X.shape
n_hidden_neurons = 2
n_categories = 2
n_features = 2

eta_vals = np.logspace(-5, 1, 7)
lmbd_vals = np.logspace(-5, 1, 7)
# store models for later use
DNN_scikit = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)
epochs = 100

for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        dnn = MLPClassifier(hidden_layer_sizes=(n_hidden_neurons), activation='logistic',
                            alpha=lmbd, learning_rate_init=eta, max_iter=epochs)
        dnn.fit(X, yXOR)
        DNN_scikit[i][j] = dnn
        print("Learning rate  = ", eta)
        print("Lambda = ", lmbd)
        print("Accuracy score on data set: ", dnn.score(X, yXOR))
        print()

sns.set()
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        dnn = DNN_scikit[i][j]
        test_pred = dnn.predict(X)
        test_accuracy[i][j] = accuracy_score(yXOR, test_pred)

fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(test_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Test Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()
```

Building neural networks in Tensorflow and Keras

Now we want to build on the experience gained from our neural network implementation in NumPy and scikit-learn and use it to construct a neural network in Tensorflow. Once we have constructed a neural network in NumPy and Tensorflow, building one in Keras is really quite trivial, though the performance may suffer.

In our previous example we used only one hidden layer, and in this we will use two. From this it should be quite clear how to build one using an arbitrary number of hidden layers, using data structures such as Python lists or NumPy arrays.

Tensorflow

Tensorflow is an open source library machine learning library developed by the Google Brain team for internal use. It was released under the Apache 2.0 open source license in November 9, 2015.

Tensorflow is a computational framework that allows you to construct machine learning models at different levels of abstraction, from high-level, object-oriented APIs like Keras, down to the C++ kernels that Tensorflow is built upon. The higher levels of abstraction are simpler to use, but less flexible, and our choice of implementation should reflect the problems we are trying to solve.

Tensorflow uses so-called graphs to represent your computation in terms of the dependencies between individual operations, such that you first build a Tensorflow *graph* to represent your model, and then create a Tensorflow *session* to run the graph.

In this guide we will analyze the same data as we did in our NumPy and scikit-learn tutorial, gathered from the MNIST database of images. We will give an introduction to the lower level Python Application Program Interfaces (APIs), and see how we use them to build our graph. Then we will build (effectively) the same graph in Keras, to see just how simple solving a machine learning problem can be.

To install tensorflow on Unix/Linux systems, use pip as

```
pip3 install tensorflow
```

and/or if you use **anaconda**, just write (or install from the graphical user interface) (current release of CPU-only TensorFlow)

```
conda create -n tf tensorflow
conda activate tf
```

To install the current release of GPU TensorFlow

```
conda create -n tf-gpu tensorflow-gpu
conda activate tf-gpu
```

Using Keras

Keras is a high level neural network that supports Tensorflow, CTNK and Theano as backends. If you have Anaconda installed you may run the following command

```
conda install keras
```

You can look up the instructions here for more information.

We will to a large extent use **keras** in this course.

Collect and pre-process data

Let us look again at the MINST data set.

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn import datasets


# ensure the same random numbers appear every time
np.random.seed(0)

# display images in notebook
%matplotlib inline
plt.rcParams['figure.figsize'] = (12,12)


# download MNIST dataset
digits = datasets.load_digits()

# define inputs and labels
inputs = digits.images
labels = digits.target

print("inputs = (n_inputs, pixel_width, pixel_height) = " + str(inputs.shape))
print("labels = (n_inputs) = " + str(labels.shape))


# flatten the image
# the value -1 means dimension is inferred from the remaining dimensions: 8x8 = 64
n_inputs = len(inputs)
inputs = inputs.reshape(n_inputs, -1)
print("X = (n_inputs, n_features) = " + str(inputs.shape))


# choose some random images to display
indices = np.arange(n_inputs)
random_indices = np.random.choice(indices, size=5)

for i, image in enumerate(digits.images[random_indices]):
    plt.subplot(1, 5, i+1)
    plt.axis('off')
    plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    plt.title("Label: %d" % digits.target[random_indices[i]])
plt.show()


from tensorflow.keras.layers import Input
from tensorflow.keras.models import Sequential      #This allows appending layers to existing mod
from tensorflow.keras.layers import Dense           #This allows defining the characteristics of
from tensorflow.keras import optimizers             #This allows using whichever optimiser we wan
from tensorflow.keras import regularizers           #This allows using whichever regularizer we w
from tensorflow.keras.utils import to_categorical   #This allows using categorical cross entropy

from sklearn.model_selection import train_test_split

# one-hot representation of labels
labels = to_categorical(labels)
```

```
# split into train and test data
train_size = 0.8
test_size = 1 - train_size
X_train, X_test, Y_train, Y_test = train_test_split(inputs, labels, train_size=train_size,
                                                    test_size=test_size)



epochs = 100
batch_size = 100
n_neurons_layer1 = 100
n_neurons_layer2 = 50
n_categories = 10
eta_vals = np.logspace(-5, 1, 7)
lmbd_vals = np.logspace(-5, 1, 7)
def create_neural_network_keras(n_neurons_layer1, n_neurons_layer2, n_categories, eta, lmbd):
    model = Sequential()
    model.add(Dense(n_neurons_layer1, activation='sigmoid', kernel_regularizer=regularizers.l2(lmb
    model.add(Dense(n_neurons_layer2, activation='sigmoid', kernel_regularizer=regularizers.l2(lmb
    model.add(Dense(n_categories, activation='softmax'))

    sgd = optimizers.SGD(lr=eta)
    model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

    return model


DNN_keras = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        DNN = create_neural_network_keras(n_neurons_layer1, n_neurons_layer2, n_categories,
                                          eta=eta, lmbd=lmbd)
        DNN.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size, verbose=0)
        scores = DNN.evaluate(X_test, Y_test)

        DNN_keras[i][j] = DNN

        print("Learning rate = ", eta)
        print("Lambda = ", lmbd)
        print("Test accuracy: %.3f" % scores[1])
        print()


# optional
# visual representation of grid search
# uses seaborn heatmap, could probably do this in matplotlib
import seaborn as sns

sns.set()

train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))

for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        DNN = DNN_keras[i][j]

        train_accuracy[i][j] = DNN.evaluate(X_train, Y_train)[1]
        test_accuracy[i][j] = DNN.evaluate(X_test, Y_test)[1]
```

```
fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(train_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Training Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()

fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(test_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Test Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()
```

The Mathematics of Neural Networks

1. Activation functions and vanishing gradients
2. Brief summary of gradient methods
3. Approximation theorems, in particular the *universal approximation theorem* for neural networks by Cybenko and Hornik

I strongly recommend Michael Nielsen's intuitive approach to the neural networks and the universal approximation theorem, see the slides at http://neuralnetworksanddeeplearning.com/chap4.html.

Fine-tuning neural network hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable network topology (how neurons/nodes are interconnected), but even in a simple FFNN you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weight initialization logic, the stochastic gradient optmized and much more. How do you know what combination of hyperparameters is the best for your task?

- You can use grid search with cross-validation to find the right hyperparameters.

However,since there are many hyperparameters to tune, and since training a neural network on a large dataset takes a lot of time, you will only be able to explore a tiny part of the hyperparameter space.

- You can use randomized search.
- Or use tools like Oscar, which implements more complex algorithms to help you find a good set of hyperparameters quickly.

Hidden layers

For many problems you can start with just one or two hidden layers and it will work just fine. For the MNIST data set you ca easily get a high accuracy using just one hidden layer with a few hundred neurons. You can reach for this data set above 98% accuracy using two hidden layers with the same total amount of neurons, in roughly the same amount of training time.

For more complex problems, you can gradually ramp up the number of hidden layers, until you start overfitting the training set. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers and they need a huge amount of training data. However, you will rarely have to train such networks from scratch: it is much more common to reuse parts of a pretrained state-of-the-art network that performs a similar task.

Which activation function should I use?

The Back propagation algorithm we derived above works by going from the output layer to the input layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of the cost function with regards to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent (GD) step.

Unfortunately for us, the gradients often get smaller and smaller as the algorithm progresses down to the first hidden layers. As a result, the GD update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is known in the literature as **the vanishing gradients problem**.

In other cases, the opposite can happen, namely the the gradients can grow bigger and bigger. The result is that many of the layers get large updates of the weights the algorithm diverges. This is the **exploding gradients problem**, which is mostly encountered in recurrent neural networks. More generally, deep neural networks suffer from unstable gradients, different layers may learn at widely different speeds

Is the Logistic activation function (Sigmoid) our choice?

Although this unfortunate behavior has been empirically observed for quite a while (it was one of the reasons why deep neural networks were mostly abandoned for a long time), it is only around 2010 that significant progress was made in understanding it.

A paper titled Understanding the Difficulty of Training Deep Feedforward Neural Networks by Xavier Glorot and Yoshua Bengio found that the problems with the popular logistic sigmoid activation function and the weight initialization technique that was most popular at the time, namely random initialization using a normal distribution with a mean of 0 and a standard deviation of 1.

They showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its

inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers. This is actually made worse by the fact that the logistic function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the logistic function in deep networks).

The derivative of the Logistic funtion

Looking at the logistic activation function, when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0. Thus when backpropagation kicks in, it has virtually no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

In their paper, Glorot and Bengio propose a way to significantly alleviate this problem. We need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. We don't want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs, and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction.

One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing/exploding gradients problems were in part due to a poor choice of activation function. Until then most people had assumed that if Nature had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks, in particular the ReLU activation function, mostly because it does not saturate for positive values (and also because it is quite fast to compute).

The RELU function family

The ReLU activation function suffers from a problem known as the dying ReLUs: during training, some neurons effectively die, meaning they stop outputting anything other than 0.

In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate. During training, if a neuron's weights get updated such that the weighted sum of the neuron's inputs is negative, it will start outputting 0. When this happen, the neuron is unlikely to come back to life since the gradient of the ReLU function is 0 when its input is negative.

To solve this problem, nowadays practitioners use a variant of the ReLU function, such as the leaky ReLU discussed above or the so-called exponential linear unit (ELU) function

$$ELU(z) = \begin{cases} \alpha\left(\exp\left(z\right) - 1\right) & z < 0, \\ z & z \geq 0. \end{cases}$$

Which activation function should we use?

In general it seems that the ELU activation function is better than the leaky ReLU function (and its variants), which is better than ReLU. ReLU performs better than tanh which in turn performs better than the logistic function.

If runtime performance is an issue, then you may opt for the leaky ReLU function over the ELU function If you don't want to tweak yet another hyperparameter, you may just use the default α of 0.01 for the leaky ReLU, and 1 for ELU. If you have spare time and computing power, you can use cross-validation or bootstrap to evaluate other activation functions.

More on activation functions, output layers

In most cases you can use the ReLU activation function in the hidden layers (or one of its variants).

It is a bit faster to compute than other activation functions, and the gradient descent optimization does in general not get stuck.

For the output layer:

- For classification the softmax activation function is generally a good choice for classification tasks (when the classes are mutually exclusive).
- For regression tasks, you can simply use no activation function at all.

Batch Normalization

Batch Normalization aims to address the vanishing/exploding gradients problems, and more generally the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.

The technique consists of adding an operation in the model just before the activation function of each layer, simply zero-centering and normalizing the inputs, then scaling and shifting the result using two new parameters per layer (one for scaling, the other for shifting). In other words, this operation lets the model learn the optimal scale and mean of the inputs for each layer. In order to zero-center and normalize the inputs, the algorithm needs to estimate the inputs' mean and standard deviation. It does so by evaluating the mean and standard deviation of the inputs over the current mini-batch, from this the name batch normalization.

Dropout

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being temporarily dropped out, meaning it will be entirely ignored during this training step, but it may be active during the next step.

The hyperparameter p is called the dropout rate, and it is typically set to 50%. After training, the neurons are not dropped anymore. It is viewed as one of the most popular regularization techniques.

Gradient Clipping

A popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold (this is mostly useful for recurrent neural networks).

This technique is called Gradient Clipping.

In general however, Batch Normalization is preferred.

A very nice website on Neural Networks

You may find this website very useful. Thx a million to Ghadi for sharing.

A top-down perspective on Neural networks

The first thing we would like to do is divide the data into two or three parts. A training set, a validation or dev (development) set, and a test set. The test set is the data on which we want to make predictions. The dev set is a subset of the training data we use to check how well we are doing out-of-sample, after training the model on the training dataset. We use the validation error as a proxy for the test error in order to make tweaks to our model. It is crucial that we do not use any of the test data to train the algorithm. This is a cardinal sin in ML. Then:

- Estimate optimal error rate
- Minimize underfitting (bias) on training data set.
- Make sure you are not overfitting.

If the validation and test sets are drawn from the same distributions, then a good performance on the validation set should lead to similarly good performance on the test set.

However, sometimes the training data and test data differ in subtle ways because, for example, they are collected using slightly different methods, or because it is cheaper to collect data in one way versus another. In this case, there can be a mismatch between the training and test data. This can lead to the neural network overfitting these small differences between the test and training sets, and a poor performance on the test set despite having a good performance on the validation set. To rectify this, Andrew Ng suggests making two validation

or dev sets, one constructed from the training data and one constructed from the test data. The difference between the performance of the algorithm on these two validation sets quantifies the train-test mismatch. This can serve as another important diagnostic when using DNNs for supervised learning.

Limitations of supervised learning with deep networks

Like all statistical methods, supervised learning using neural networks has important limitations. This is especially important when one seeks to apply these methods, especially to physics problems. Like all tools, DNNs are not a universal solution. Often, the same or better performance on a task can be achieved by using a few hand-engineered features (or even a collection of random features).

Here we list some of the important limitations of supervised neural network based models.

- **Need labeled data**. All supervised learning methods, DNNs for supervised learning require labeled data. Often, labeled data is harder to acquire than unlabeled data (e.g. one must pay for human experts to label images).
- **Supervised neural networks are extremely data intensive.** DNNs are data hungry. They perform best when data is plentiful. This is doubly so for supervised methods where the data must also be labeled. The utility of DNNs is extremely limited if data is hard to acquire or the datasets are small (hundreds to a few thousand samples). In this case, the performance of other methods that utilize hand-engineered features can exceed that of DNNs.
- **Homogeneous data.** Almost all DNNs deal with homogeneous data of one type. It is very hard to design architectures that mix and match data types (i.e. some continuous variables, some discrete variables, some time series). In applications beyond images, video, and language, this is often what is required. In contrast, ensemble models like random forests or gradient-boosted trees have no difficulty handling mixed data types.
- **Many problems are not about prediction.** In natural science we are often interested in learning something about the underlying distribution that generates the data. In this case, it is often difficult to cast these ideas in a supervised learning setting. While the problems are related, it is possible to make good predictions with a *wrong* model. The model might or might not be useful for understanding the underlying science.

Some of these remarks are particular to DNNs, others are shared by all supervised learning methods. This motivates the use of unsupervised methods which in part circumvent these problems.