Introduction to Deep Neural Networks with Keras/TensorFlow

Greg Teichert

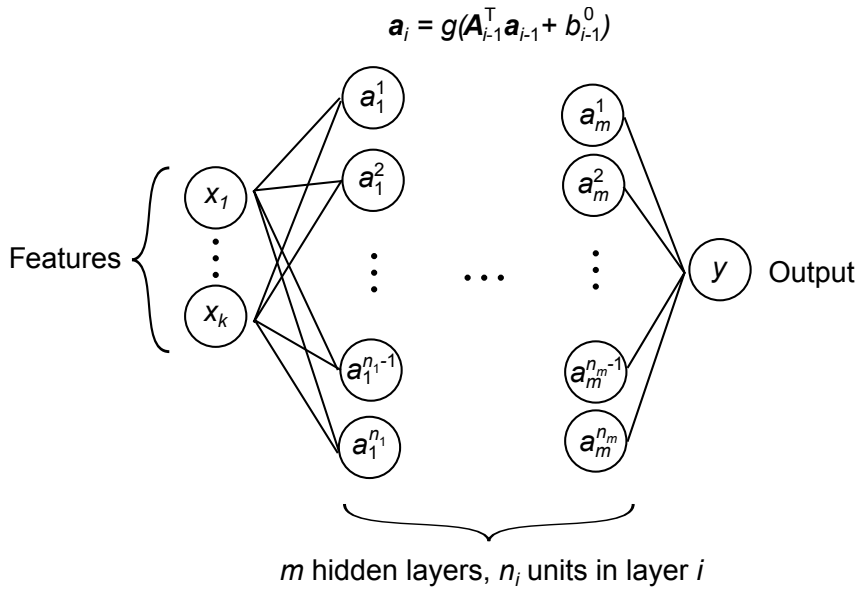
Consulting for Statistics, Computing and Analytics Research

Workshop materials

- Go to: `github.com/greght/Workshop-Keras-DNN`
- Find links to Python code, which is set up in Google Colab
- Slides also available

Deep Neural Networks (DNNs)

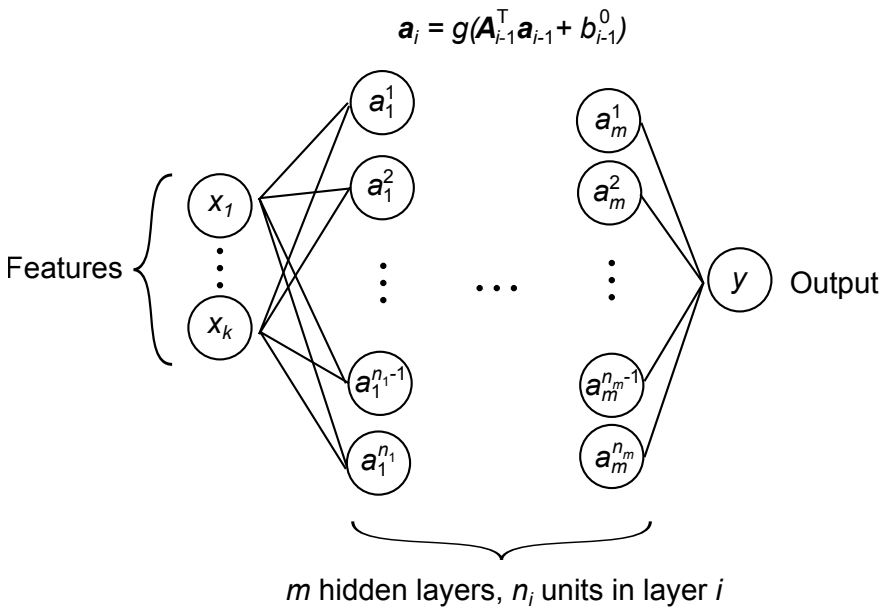
- A DNN is a mathematical function inspired by neural networks in the brain.
- Input layer (features), hidden layers, output layer (targets).
- Your data determines number of features and targets.
- You choose number of hidden layers and "neurons" (activation units) in each hidden layer.

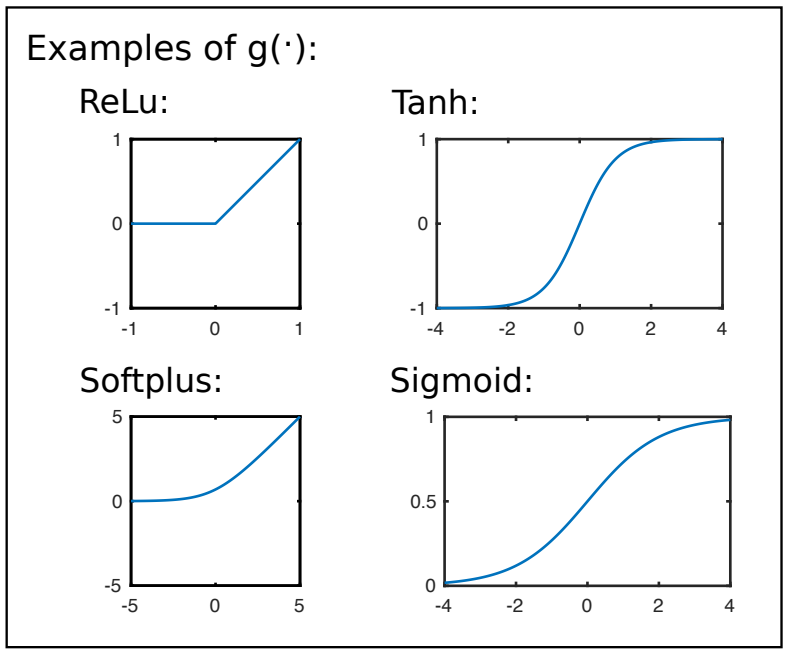


Deep Neural Networks (DNNs), cont'd

- Hidden layers have variables (weights, biases) that are trained.
- Mathematical structure: Composite of nonlinear activation functions acting on matrix/vector operations, e.g.

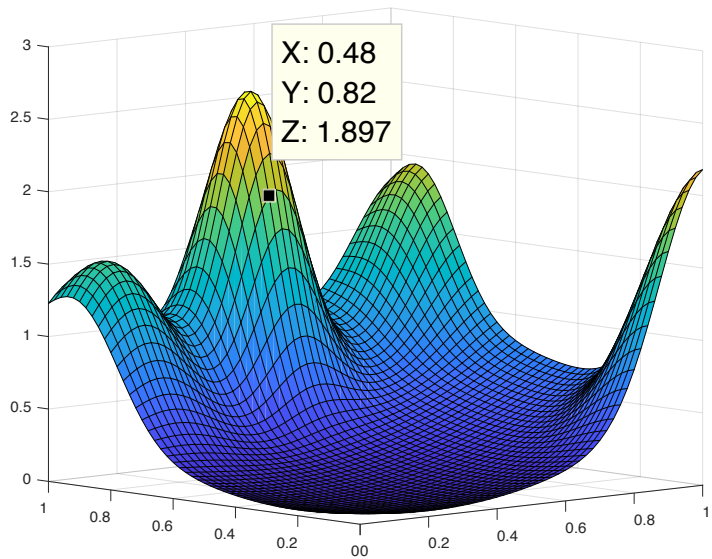
$$f(x) = A_2 g(A_1 g(A_0 x + b_0) + b_1) + b_2$$





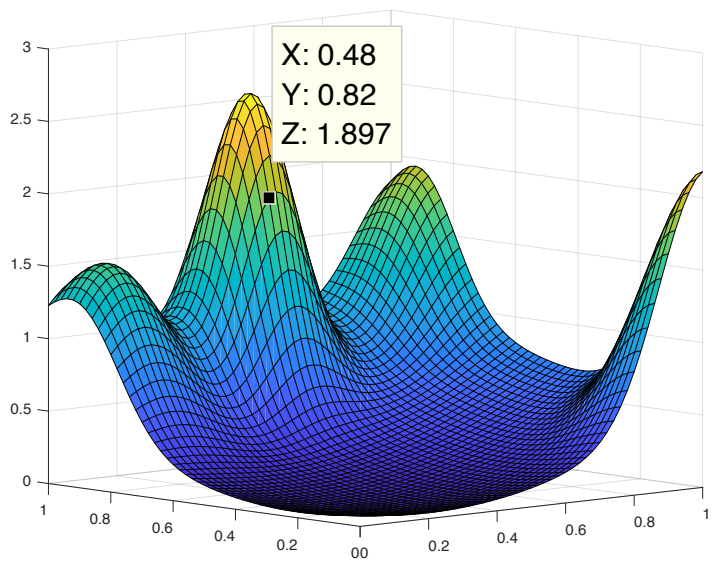
Training DNNs

- Training a DNN means optimizing the weights and biases to "fit" given data
  - i.e. minimize error between DNN prediction and the given data
- Optimization: Think of mountains and valleys. Your location is like the value of the weights/biases. Your elevation is like the value of the error. As you "walk down the mountain", you are changing the values of the weights/biases to decrease the value of the error.

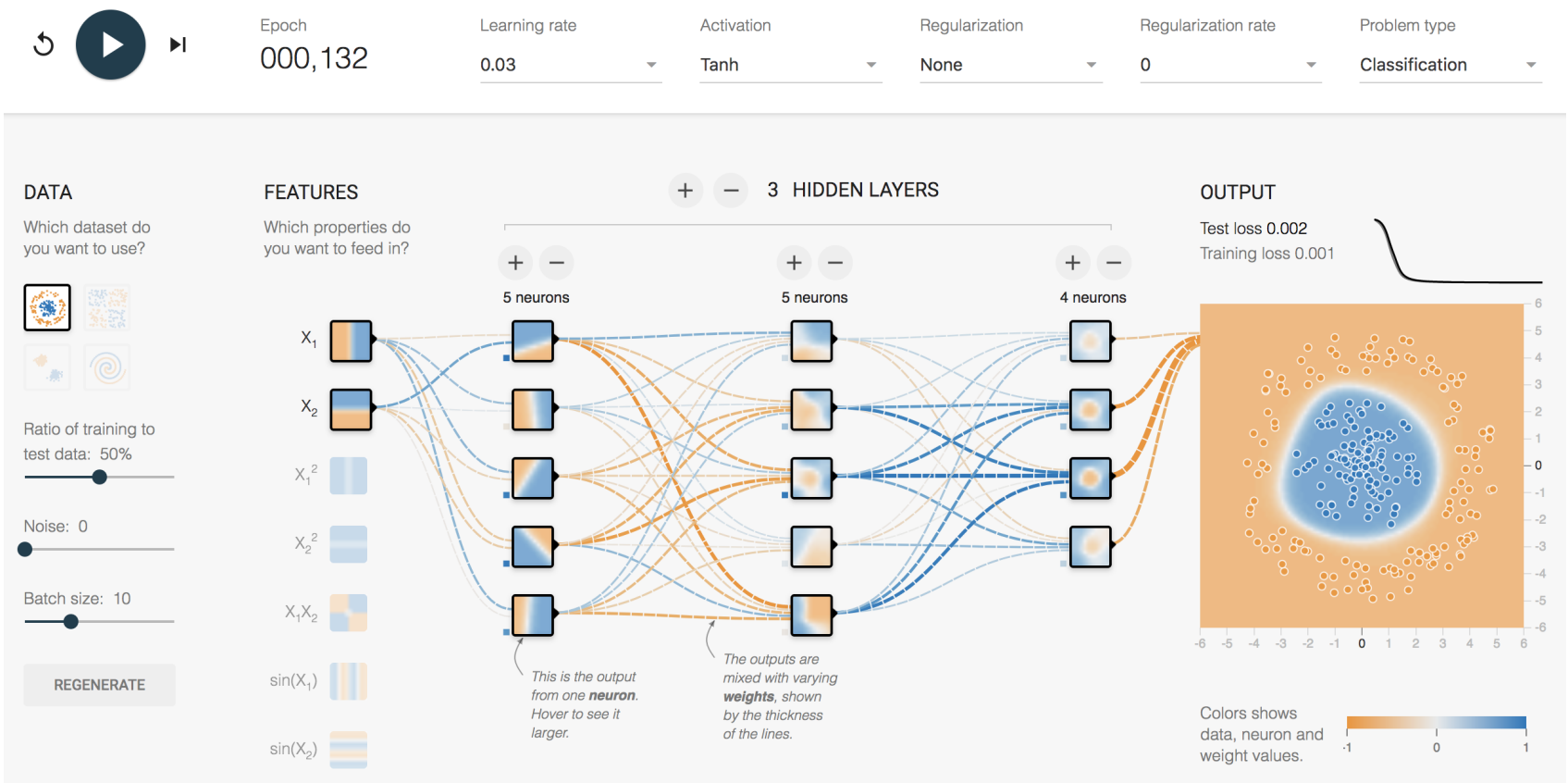


Training DNNs, cont'd

- Usually a variant of **stochastic gradient descent**:
  - **Gradient**: Points toward steepest slope
  - **Gradient descent** method: Take steps down steepest slope to get to minimum
  - **Stochastic gradient descent**: Calculate the error based on a small number of data (a **batch**) instead of the entire data set
- You choose: step size (learning rate), batch size



playground.tensorflow.org



Note: playground.tensorflow.org is an educational tool. It does not actually use the TensorFlow library, nor can you use it to train with your data.

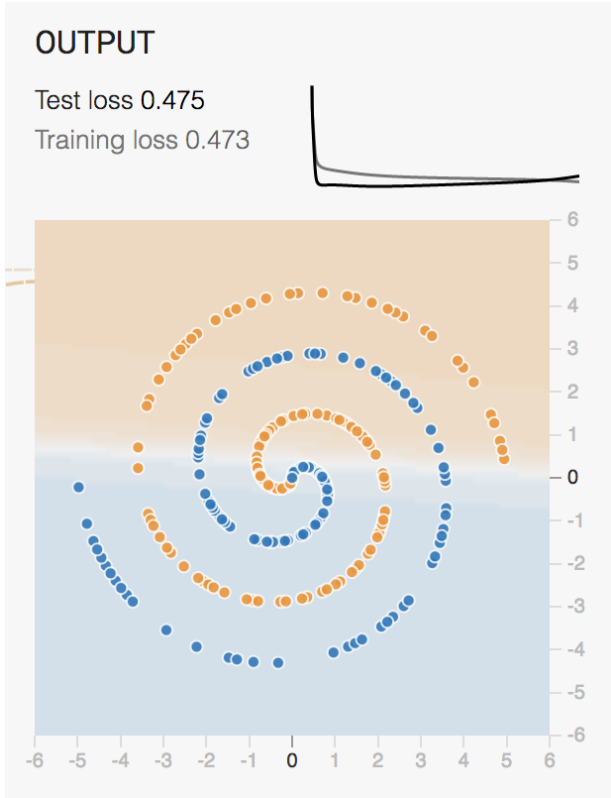
Underfitting (high bias)

Symptoms:

- High training and testing error

Possible treatments:

- Make the model larger (more layers, more neurons)
- Increase the number of features, artificially if necessary (e.g. $x_1 x_2$, $\sin(x)$, etc.)
- More training



see Andrew Ng's "Machine Learning" Coursera class, or his upcoming book "Machine Learning Yearning."

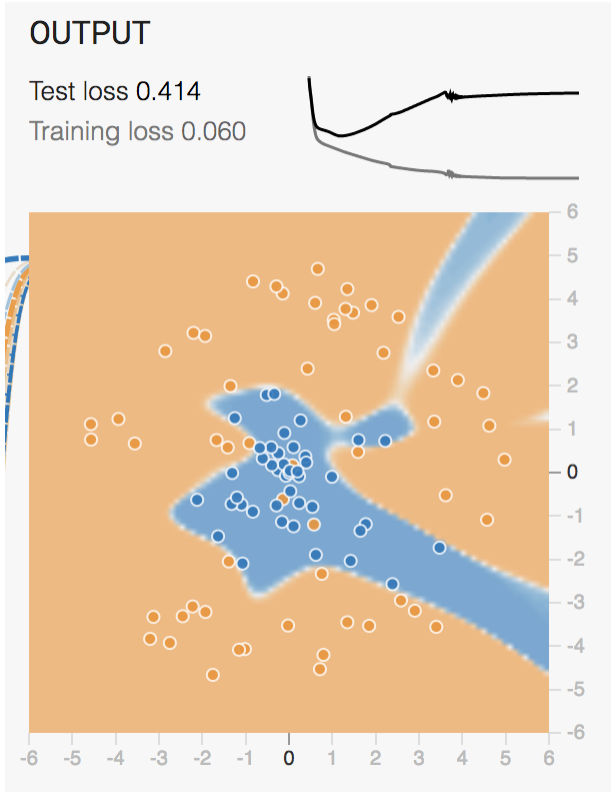
Overfitting (high variance)

Symptoms:

- Low training error, high testing error
- (Made worse by noisy data)

Possible treatments:

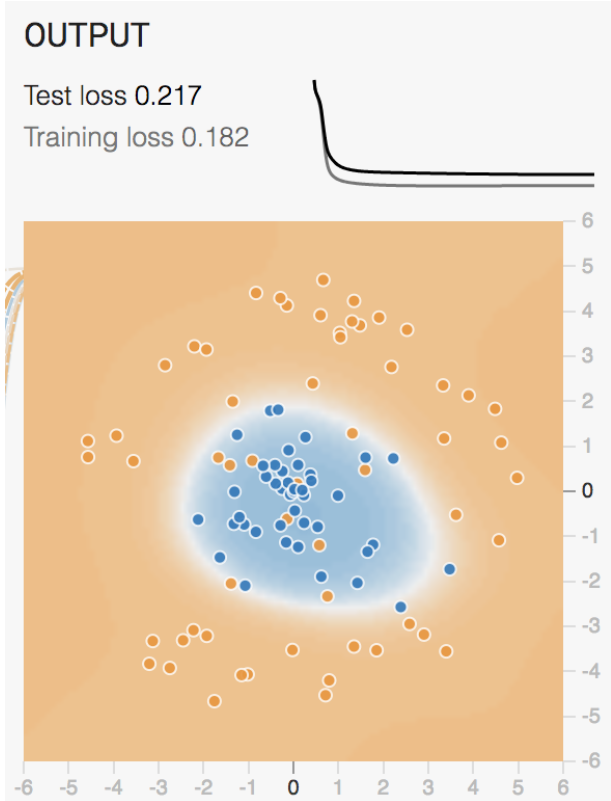
- More data
- Regularization (L1, L2, dropout)
- Less training (early stopping)
- Simplify model (use w/ caution)



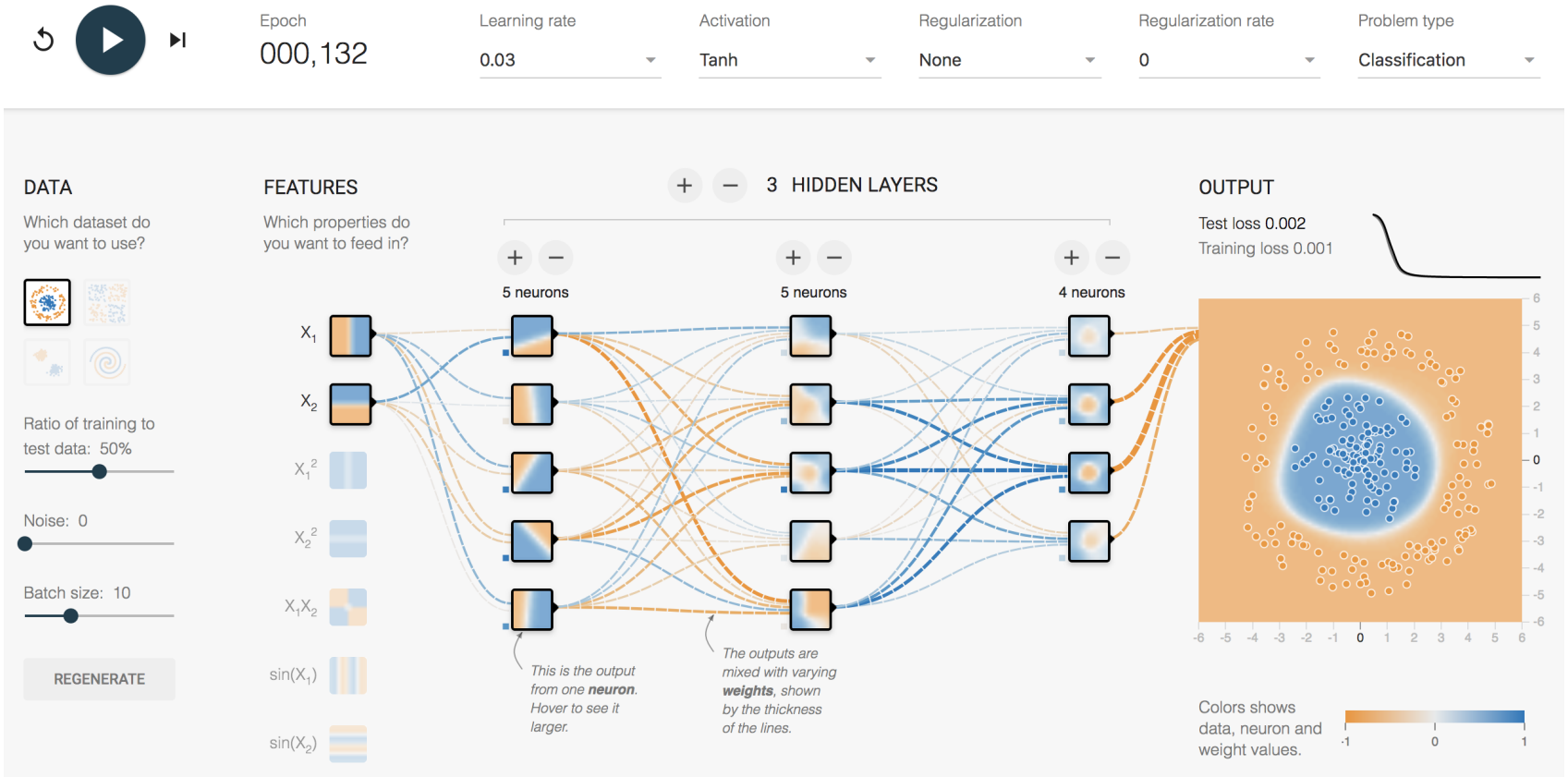
see Andrew Ng's "Machine Learning" Coursera class, or his upcoming book "Machine Learning Yearning."

Regularization

- Regularization smooths the model; reduces complexity in the output (Wikipedia).
- In neural networks, this is done by keeping the weights at a similar, low magnitude.
- L1 regularization adds the L1 norm of the weights to the loss.
- L2 regularization adds the L2 norm of the weights (more sensitive to outliers).
- Dropout randomly and temporarily drops weights to zero during training.



playground.tensorflow.org



Note: playground.tensorflow.org is an educational tool. It does not actually use the TensorFlow library, nor can you use it to train with your data.

TensorFlow

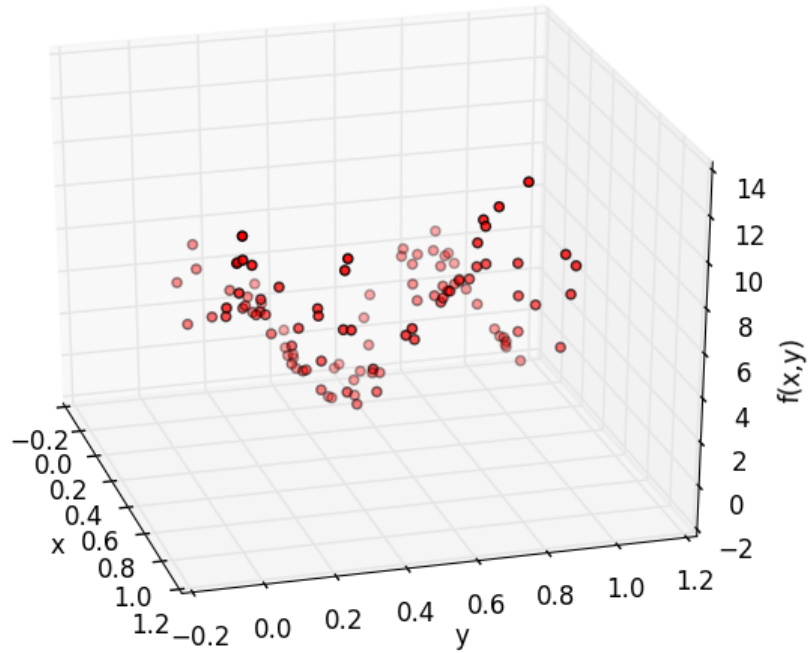
- tensorflow.org
- Open-source software library widely used for deep learning.
- Most commonly used with Python.
- Provides high-level and low-level user interface functions.
- Adopted the Keras libary for high-level interface.

Keras

- keras.io
- Python library that provides an interface with TensorFlow (or other deep learning libraries).
- (From their website:)
  - Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
  - Supports both convolutional networks and recurrent networks, as well as combinations of the two.
  - Runs seamlessly on CPU and GPU.

Nonlinear regression

- Begin with example of nonlinear regression.
- Use a standard DNN to map continuous inputs to continuous outputs.
- Example in `RegressProb.py`
- Data in example has two inputs, one output (slices parallel to x-axis are parabolic, slices parallel to y-axis are sinusoidal).



Import modules

Import modules:

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout
import numpy as np
```

Import modules for plotting results:

```
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

Import data

Here, we read in training and testing data from .csv files using NumPy.

```
# Import data
dataIn = np.genfromtxt('dataRegression_train.csv',delimiter=',')
x_train = dataIn[:,0:2]
y_train = dataIn[:,2]
dataIn = np.genfromtxt('dataRegression_test.csv',delimiter=',')
x_test = dataIn[:,0:2]
y_test = dataIn[:,2]
```

Build the model

Define the structure of the DNN. Here, we define two hidden layers, with 5 neurons in each layer.

We also specify the activation function here. The `relu` function is commonly used, but you can use others (examples: Keras, Wikipedia):

`sigmoid`, `softplus`, `tanh`, etc.

Note that no activation is used on the final layer.

Experiment with the hidden units and activation function.

```
# Create model
model = Sequential()
model.add(Dense(units=5, activation='relu', input_dim=2))
model.add(Dense(units=5, activation='relu'))
model.add(Dense(units=1))
```

L1, L2 regularization

L1 and L2 regularization are added through the loss function. In keras, this is specified within each layer, if desired. (Note that regularization is not always needed and, in the case of underfitting, can be counter productive.)

E.g. for L2 (and similarly for L1):

```
# Create model
model = Sequential()
model.add(Dense(units=5, activation='relu', input_dim=2,
          kernel_regularizer=keras.regularizers.l2(0.01)))
model.add(Dense(units=5, activation='relu',
          kernel_regularizer=keras.regularizers.l2(0.01)))
model.add(Dense(units=1))
```

The number (0.01 in this example) is the coefficient multipled by the respective norm in the loss function. A higher number imposes more regularization.

Dropout

Dropout can be added as a layer within in the model. In keras, it affects only the previous layer, so you can add it after every dense layer, if desired, except the output layer, e.g.

```
# Create model
model = Sequential()
model.add(Dense(units=5, activation='relu', input_dim=2))
model.add(Dropout(0.4))
model.add(Dense(units=5, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(units=1))
```

This line would cause dropout to happen in the previous layers at a rate of 40% (i.e. 40% of weights are temporarily set to zero at each training iteration).

Compile

With the `compile` function, we define the loss function and the optimizer. For regression, we use the mean squared error (mse) loss function.

Keras (and TensorFlow) has several optimizers available, with different variations on gradient descent:

`SGD`, `Adadelta`,
`Adam`, `Adagrad`, etc.

It is possible to define a learning rate (the step size). Experiment with the different optimizer types and learning rates.

```
model.compile(loss='mse',optimizer=keras.optimizers.Adagrad(lr=0.1))
```

Training

Stochastic gradient descent methods use shuffled mini-batches instead of the entire data set for each training iteration. We specify batch size, and how many epochs to train the code.

An epoch is the number of training iterations required to go through the entire training set once. For example, 1,000 datapoints and a batch size of 10, one epoch would take 100 training iteration.

We can also specify validation data to see how the validation loss changes during training.

Experiment with batch size and number of epochs.

```
# Train
model.fit(x_train, y_train,
          validation_data=(x_test,y_test),
          epochs=1000,
          batch_size=10)
```

Early stopping

If we want to use early stopping (i.e. stop training based on if the validation error is still decreasing), we can use the following:

```
# Train
earlyStopping = keras.callbacks.EarlyStopping(patience=10)
model.fit(x_train, y_train,
          validation_data=(x_test,y_test),
          epochs=1000,
          batch_size=10,
          callbacks=[earlyStopping])
```

The "patience=10" means that after 10 epochs with no improvement in the validation loss, the training will stop.

Prediction

We create a set of (x_1, x_2) points to use for prediction. The `model.predict` function returns the predicted value for the inputs.

```
# Create a prediction set
x_pred = np.mgrid[0:1:25j, 0:1:25j].reshape(2,-1).T
y_pred = model.predict(x_pred)
```

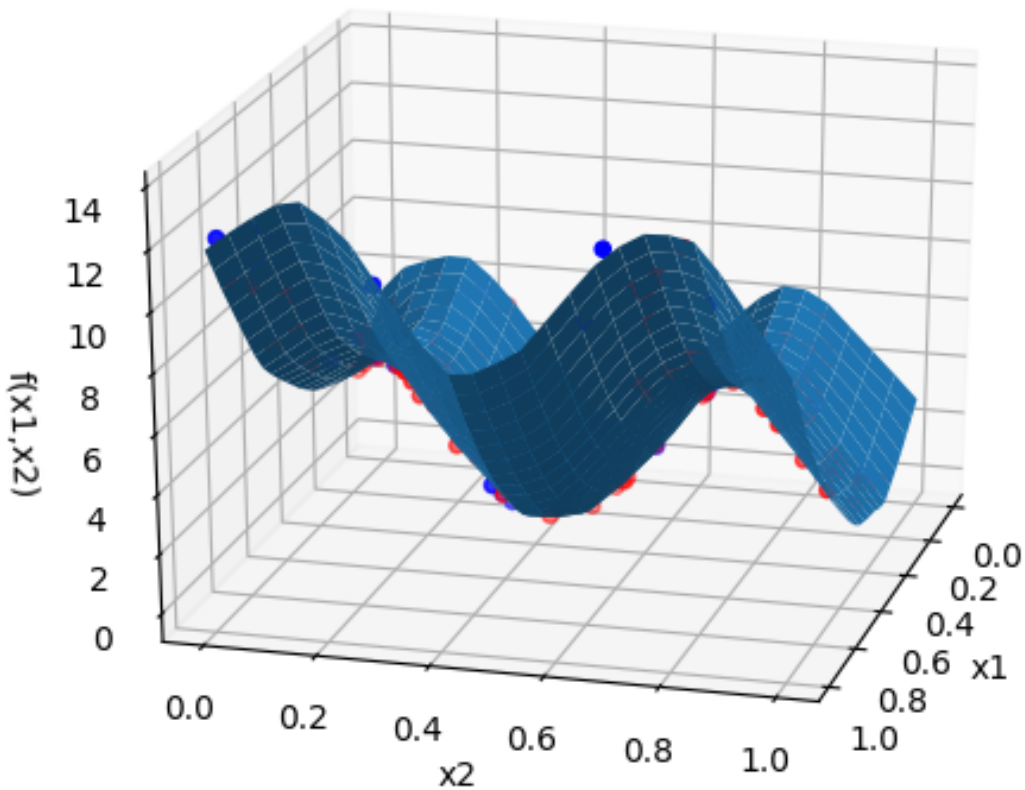
Plotting

The rest of the code plots the data and predicted surface with the Python Matplotlib library (this code is independent of Keras).

```
# Plot the actual and predicted values
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x1 = x_pred[:,0].reshape(25,-1)
x2 = x_pred[:,1].reshape(25,-1)
y = y_pred.reshape(25,-1)

ax.scatter(x_train[:,0], x_train[:,1], y_train, c='r', marker='o')
ax.scatter(x_test[:,0], x_test[:,1], y_test, c='b', marker='o')
ax.plot_surface(x1,x2,y)

ax.update({'xlabel':'x1', 'ylabel':'x2', 'zlabel':'f(x1,x2)'})

plt.show()
```

Results

With good settings in the code (not the current settings), we can get the following fit:



Minimum example for training

```
import keras
from keras.models import Sequential
from keras.layers import Dense
import numpy as np

# Import data
dataIn = np.genfromtxt('dataRegression_train.csv',delimiter=',')
x_train = dataIn[:,0:2]
y_train = dataIn[:,2]

# Create model
model = Sequential()
model.add(Dense(units=5, activation='relu', input_dim=2))
model.add(Dense(units=5, activation='relu'))
model.add(Dense(units=1))

model.compile(loss='mse',optimizer=keras.optimizers.Adagrad(lr=0.1))

# Train
model.fit(x_train, y_train,
          epochs=1000,
          batch_size=10)
```

Exercise 1

- Open the file `ChallengeProblems/RegressProb.py`.
- Run the code using:
  `python RegressProb.py`
- Identify the problem (underfitting or overfitting).
- Try possible solutions to get a better fit.

Classification

- Consider the problem of classification.
- Maps feature values to a category.
- Use the example of irises
  - Four features: sepal length, sepal width, petal length, petal width
  - Three classes: Iris setosa, Iris virginica, Iris versicolor

Figure: Iris versicolor, by Danielle Langlois (CC BY-SA 3.0), commons.wikimedia.org/w/index.php?curid=248095

Import modules

There are a few changes between the regression and this classification example.

Import modules (no plotting in this example):

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout
import numpy as np
```

Import data

We again import training and testing data using NumPy.

```
# Import data
dataIn = np.genfromtxt('iris_training.csv',delimiter=',')
x_train = dataIn[:,0:-1]
y_train = keras.utils.to_categorical(dataIn[:,-1], num_classes=3)

dataIn = np.genfromtxt('iris_test.csv',delimiter=',')
x_test = dataIn[:,0:-1]
y_test = keras.utils.to_categorical(dataIn[:,-1], num_classes=3)
```

Data label format: Usually given as 0, 1, or 2; we need it to be [1,0,0], [0,1,0], or [0,0,1]. This conversion is done with the `keras.utils.to_categorical` function.

Build the model

Define the structure of the DNN. Here, we define three hidden layers, with 1000, 500, and 70 neurons in each respective layer.

```
# Create model
model = Sequential()
model.add(Dense(units=1000, activation='relu', input_dim=4))
model.add(Dense(units=500, activation='relu'))
model.add(Dense(units=70, activation='relu'))
model.add(Dense(units=3, activation='softmax'))
```

Since this is classification, apply the softmax function to the last layer. This transforms the output to be a vector of probabilities that sum to one:

$$p_i = \frac{\exp(f_i)}{\sum_j \exp(f_j)}$$

where p_i is probability of category i being true, f_i is i-th component of the final layer's output.

Compile

With the `compile` function, we again define the loss function and the optimizer. For classification, we use the cross entropy loss function. We are also interested in the accuracy metric (% correctly classified), in addition to the loss.

```
model.compile(loss='categorical_crossentropy',
              optimizer=keras.optimizers.Adagrad(lr=0.01),
              metrics=['accuracy'])
```

$$\text{cross\_entropy} = \frac{1}{n_{\text{samples}}} \sum_{j}^{n_{\text{samples}}} \sum_{i}^{n_{\text{classes}}} \hat{p}_i^j \log(p_i^j)$$

where $\hat{p}_i^j$ is the data and p_i^j is the prediction for class i, sample j.

Training

Training is done as before.

```
# Train
model.fit(x_train, y_train,
          validation_data=(x_test,y_test),
          epochs=2000,
          batch_size=10)
```

Prediction

We create a set of (x_1, x_2) points to use for prediction. The `model.predict` function returns the predicted value for the inputs.

```
# Create a prediction set
x_predict = np.array([[6.4, 3.2, 4.5, 1.5],
                      [5.8, 3.1, 5.0, 1.7]], dtype=np.float32)
y_predict = model.predict(x_predict)
```

The code should predict classes 1 and 2, respectively, for the two new samples.

Exercise 2

- Open the file `ChallengeProblems/ClassifyProb.py`.
- Run the code.
- Identify the problem (underfitting or overfitting).
- Try possible solutions to get a better result.

Convolutional Neural Network (CNN)

- Image recognition is often done with CNNs.
- CNNs perform classification by adding new types of layers, primarily "convolutions" and "pooling".
- The "convolution": scanning a filter across the image.
- The "pooling": take the most significant features from a group of pixels.
- Some nice explanations of CNNs by Adam Geitgey and ujjwalkarn.
- Our example will use the MNIST database of handwritten digits.
- Based on this example.

Import modules, data

There are a few more modules to import for the CNN:

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D, Dropout,
    Flatten, BatchNormalization
```

This dataset is available directly through Keras:

```
# Load data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
y_train = keras.utils.to_categorical(y_train, num_classes=10)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
y_test = keras.utils.to_categorical(y_test, num_classes=10)
```

Initialize model, Normalize input

We initialize the model as before:

```
model = Sequential()
```

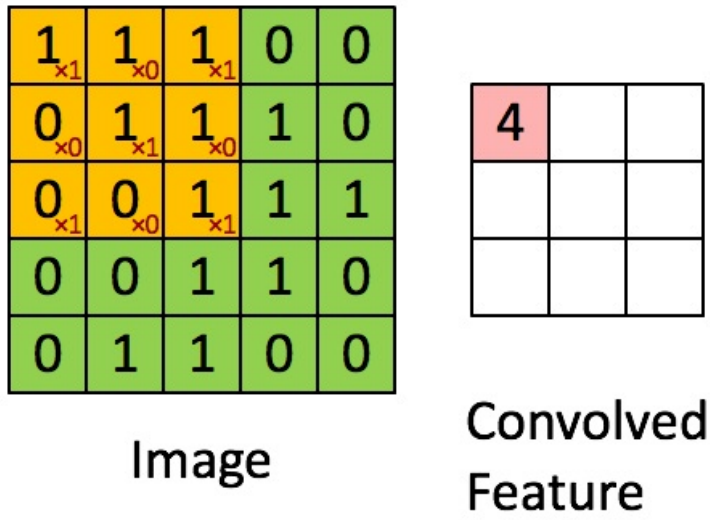
We shift and normalize the inputs for better fitting.

```
model.add(BatchNormalization(input_shape=(28,28,1)))
```

We also define the input shape. The images are 28 by 28 pixels, with a grayscale value. This means each image is defined by a 3D tensor, $28 \times 28 \times 1$ (a color image of the same size would be $28 \times 28 \times 3$).

Convolutional layer

The first convolutional layer is applied. This involves sweeping a filter across the image (gif source: deeplearning.stanford.edu).



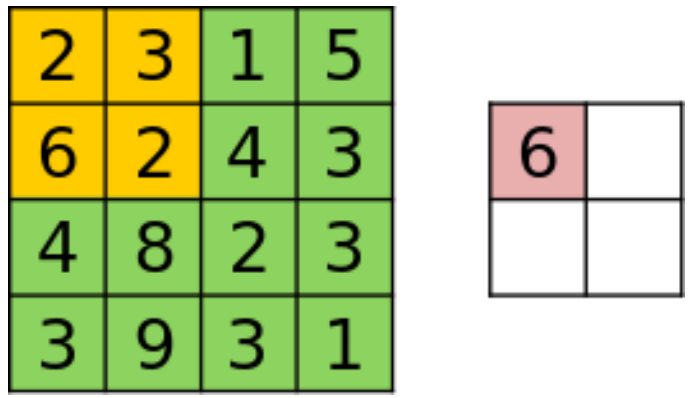
Convolution $\Longrightarrow$ translational invariance (it doesn't matter where the object of interest is located).

We use 4 filters with a size of 5×5 pixels, with ReLU activation.

```
model.add(Conv2D(4,kernel_size=(5,5),activation='relu'))
```

Max pooling

Max pooling involves looking at clusters of the output (in this example, 2×2 clusters), and sets the maximum filter value as the value for the cluster.



I.e. a "match" anywhere in the cluster $\Longrightarrow$ a "match" for the cluster.

Since we are also using stride of 2, the clusters don't overlap.

```
model.add(MaxPooling2D(pool_size=(2,2)))
```

Pooling reduces the size of the neural net, speeding up computations.

2nd convolution and pooling

A second convolutional layer, followed by max pooling, is used.

```
model.add(Conv2D(16,(5,5),activation='relu'))
model.add(MaxPooling2D((2,2)))
```

Fully-connected layer

The 3D tensor is converted back to a 1D tensor to act as input for a dense or fully-connected layer, the same type used with the previous regression and classification examples.

```
model.add(Flatten())
model.add(Dense(100,activation='relu'))
```

Dropout, Softmax

We add a dropout layer here. In this example, dropout happens at a rate of 40% (i.e. 40% of weights are temporarily set to zero at each training iteration).

```
model.add(Dropout(0.4))
```

As in the Iris classification problem, we finish with a dense layer and softmax activation function to return probabilities for each category:

```
model.add(Dense(10,activation='softmax'))
```

Compile, Train

We compile and train as in the previous classification example:

```
model.compile(loss='categorical_crossentropy',
              optimizer=keras.optimizers.Adagrad(lr=0.01),
              metrics=['accuracy'])

# Train
model.fit(x_train, y_train,
          validation_data=(x_test,y_test),
          epochs=5, batch_size=100, shuffle=True)
```

With the setup in this example, you should achieve an accuracy of over 98%:

Exercise 3

- Use the file `ChallengeProblems/CNNProb.py`.
- Modify the CNN and training to see how high of a validation accuracy you can get.