DIGITAL TWIN AI and Machine Learning: Deep Learning I: Neural Networks

Prof. Andrew D. Bagdanov
`andrew.bagdanov AT unifi.it`

MICC
Media Integration and Communication Center

Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Firenze

20 November 2020

Outline

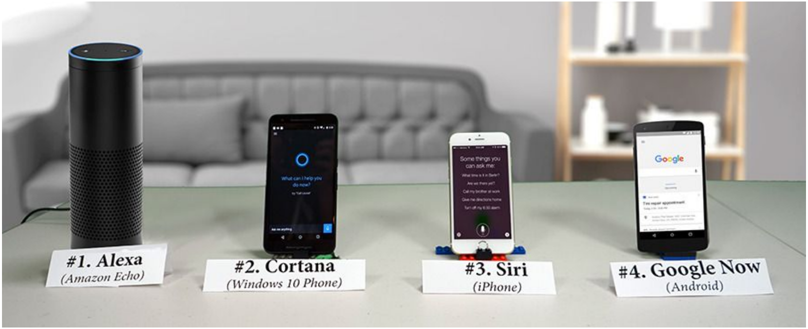
Introduction

Remaining lectures

- This afternoon: Deep Learning and Keras/Tensorflow.
- Next Thursday: Recurrent Neural Networks (RNNs).
- Next Friday: Convolutional Neural Networks (CNNs) + Final Exam

Digital assistants

- Deep learning is profoundly changing our lives.



Natural language processing

- Deep Recurrent Neural Networks (RNNs) are powering the latest generation of natural language translation technologies.

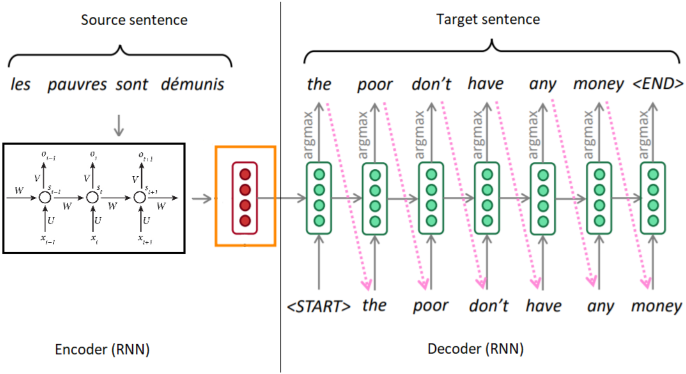


Image captioning

- Convolutional Neural Networks (CNNs) are able to extract high-level semantics from images.

Two pug dogs sitting on a bench at the beach.

A child is sitting on a couch and holding an umbrella.

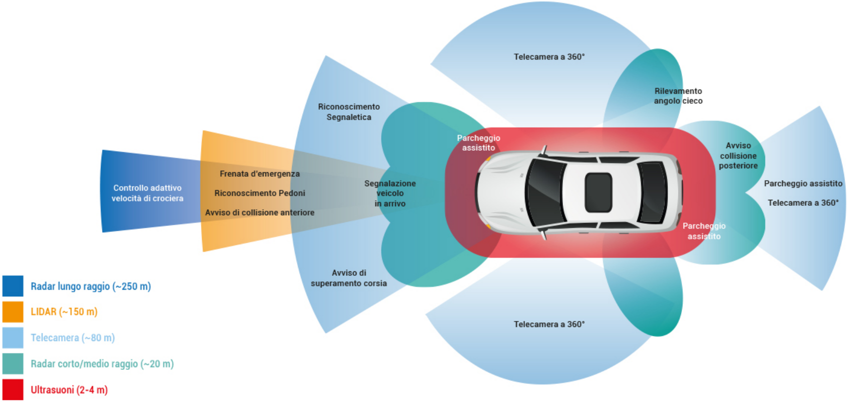
The person in the brown suit is directing a dog.

A person holding a black umbrella and an accordion.

Some dolphins are swimming close to the base of the ocean.

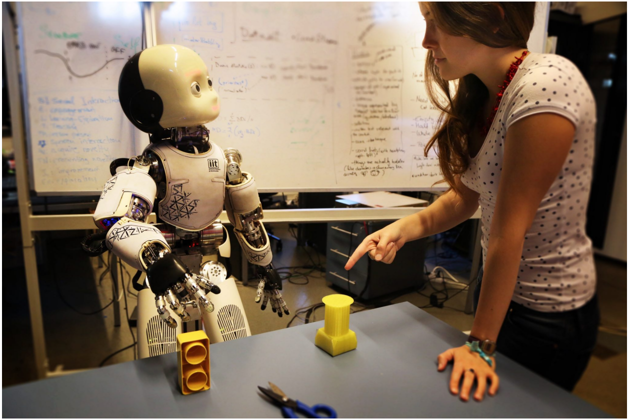
Self-driving cars

- CNNs are able to integrate multi-modal inputs and are driving the latest advances in Automatic Driving Assistance (ADAS) systems.



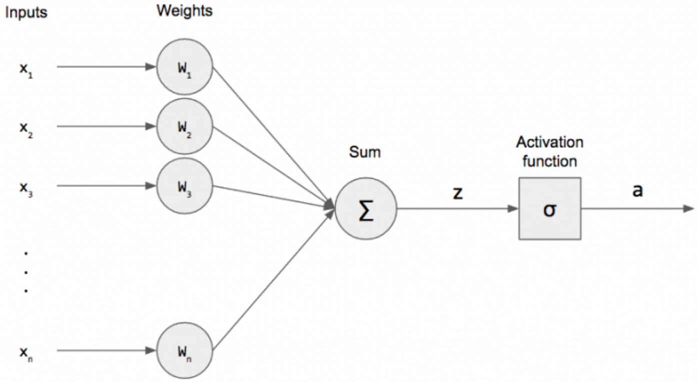
Reinforcement learning

- Deep Reinforcement Learning is being used to train robots who can learn from experience and interactions with their environment.



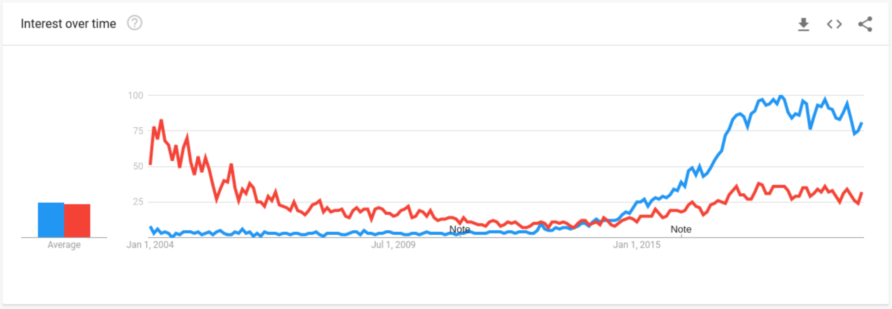
All thanks to. . .

- The humble Neural Network.
- Artificial Neural Networks (ANNs) are extremely simple, yet also extremely powerful models.
- They are, in fact, universal function approximators.



Neural Networks are not new

- As we will see, neural networks have a storied history.
- Deep Learning, however, is their modern incarnation.



Overview

- Today we will see what puts the deep into Deep Learning.
- We will start with an overview of the major historical milestones in the development of artificial neural networks.
- Then we will look at how modern deep neural networks are actually built:
- We will see how the basic Multilayer Perceptron (MLP) model provides a modular architecture for machine learning problems.
- We will see how to fit model parameters in order to minimize a loss function.
- And we will see how modern tools (e.g. Keras/Tensorflow) makes it easy to apply Deep Models to new problems.

Connectionist Models

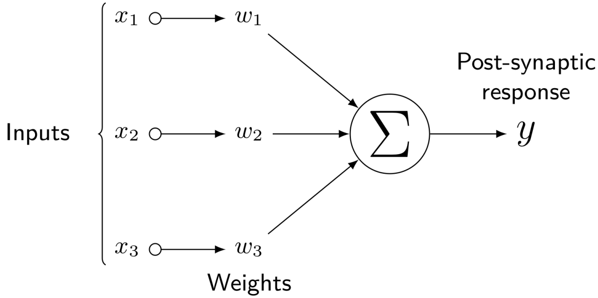
What is a Deep Neural Network?

- Deep Neural Networks are a connectionist model.
- Connectionism has deep roots reaching back to Classical Greece.
- To understand this rich inheritance it is useful to go back in time and trace the roots of modern Deep Models.
- Connectionism arose from the neuroscience and psychological research communities of the 1940s and 1950s.
- These were the nascent beginning of what would become Cognitive Science.
- Though founded on solid experimental practice, what was lacking was any sort of computation basis for learning.

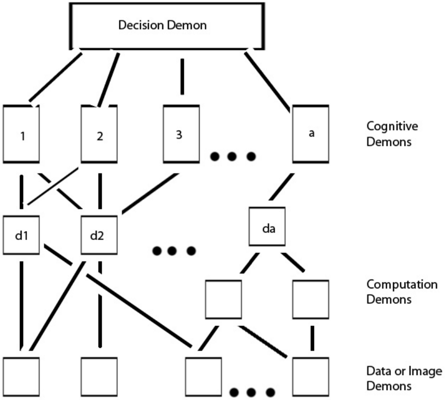
Connectionism: Hebbian Learning

- One of the first concrete learning rules for connectionist models (both artificial and biological).
- Hebb's Rule: if cell A consistently contributes to the activity of cell B, then the synapse from A to B should be strengthened.
- More quaintly: *neurons that fire together, wire together; neurons that fire out of sync, fail to link*.

$$\begin{aligned} w_{ij} &= x_i x_j \\ w_{ij} &= \frac{1}{N} \sum_p x_i^p x_j^p \\ \Delta w_i &= \eta x_i y \\ &= \eta x_i \sum_j w_i x_i \end{aligned}$$



Connectionism: The Pandemonium Model



- In 1958 Selfridge proposed a multi-layer, parallel model of machine learning.
- The model consists of four layers, each inhabited by demons.
- Network architecture fixed *a priori*, connections updated using supervised learning.
- Demons yell upwards, higher-level ones listen and respond.
- High-worth demons can replace low-worth ones via combination.

Connectionism: The Perceptron

- The Perceptron is probably the simplest (and most famous) feedforward neural network.
- The perceptron algorithm was invented by Rosenblatt in 1958.
- It was designed to be a machine, and its original purpose was to perform image recognition.

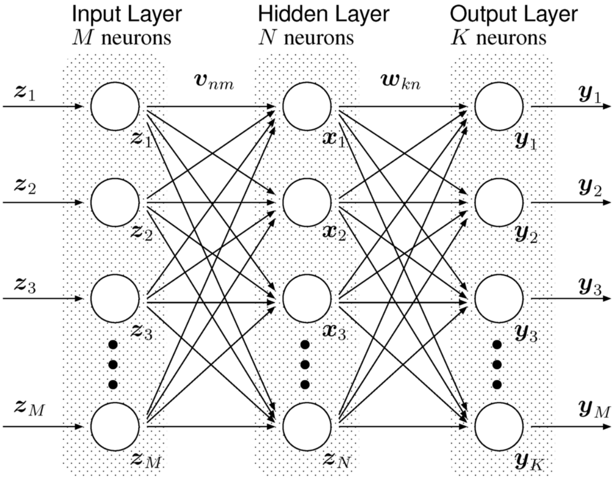
$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

The perceptron algorithm

```
Input: $D = \{ (\mathrm{x}_i, y_i) \}_{i=1}^{N}$ (training data)
Output: learned weights $\mathrm{w}$
$\mathrm{w}_0 \leftarrow$ random initialization
$t \leftarrow 1$
while not converged do
    for $(\mathrm{x}, y) \in D$ do
        $\hat{y} = f(\mathrm{w}^T \mathrm{x})$
        $\mathrm{w}_t \leftarrow w_{t-1} + \eta(y - \hat{y})\mathrm{x}$
        $t \leftarrow t + 1$
```

Connectionism: The Multilayer Perceptron

- Let's look at a simple Neural Network architecture known as the Multilayer Perceptron (MLP):

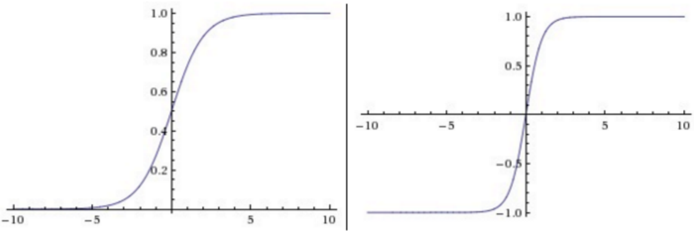


Connectionism: The Multilayer Perceptron

- The MLP equation (one hidden layer):

$$\hat{\mathbf{y}}(\mathbf{x}) = \sigma(\mathbf{w}_2^T \sigma(\mathbf{w}_1^T \mathbf{x} + b_1) + b_2)$$

- Except for the activation function σ, this is a linear system.
- Common activation functions (elementwise):
  - $\sigma(\mathbf{x}) = \tanh(\mathbf{x})$
  - $\sigma(\mathbf{x}) = (1 + e^{-x})^{-1}$
  - $\sigma(\mathbf{x}) = \frac{\exp(x)}{\sum_i e^{x_i}}$ (softmax, used for outputs).



Connectionism: The Multilayer Perceptron

- How do you train a model?
- Decide on a loss function (like the negative log-likelihood):

$$L(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{x})) = -\frac{1}{C}\sum_i y_i \log(\hat{y}_i)$$

- And perform gradient descent w.r.t. all model parameters:

$$\begin{aligned} \boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n - \varepsilon \nabla_{\boldsymbol{\theta}} L(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{x})) \\ \boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n - \varepsilon \sum_{i=1}^{N} \frac{1}{N} \nabla_{\boldsymbol{\theta}} L(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{x}_i)) \end{aligned}$$

- Where ε is the learning rate.
- The standard algorithm for this is known as backpropagation and it is very clever and efficient.

Connectionism: The Multilayer Perceptron

- Problems with this approach:
    - Model size: many, many parameters for even small-sized images. This leads to memory and efficiency problems.
    - Overfitting: many parameters (and limited training data) mean that it is easy to overfit the model to your training set.
    - Undergeneralization: overfitting means that a trained model is unlikely to generalize to new data.
    - Vanishing gradients: a known problem with backpropagation (due to application of the chain rule) leads to very small gradient values near the beginning of the network.
    - Saturating units: traditional activation functions can lead to saturated units (outputs near 1 or 0 (or -1)), which have near-zero derivatives.
- These problems (and others) led the community to largely ignore the potential of these models for decades.

Connectionism: from MLP to CNNs

- However, MLPs have a number extremely attractive features:
  - It is an end-to-end model: we can train everything in the model using a single optimization algorithm.
  - MLPs learn representations of input and classifier.
  - Why can't we just use this model for image recognition problems?
  - An MLP should be able to learn feature representations that are in turn good representations for classification.
- Why is this model problematic? Especially for images?

Deep Neural Networks

Backprop: the basics

- How do you train a model?
- Decide on a loss function (like the mean-squared error):

$$\mathcal{L}(D;\theta) = \frac{1}{N} \sum_{(\boldsymbol{x},y)\in D} (y - f(\mathbf{x};\theta))^2$$

- And perform gradient descent w.r.t. all model parameters:

$$\begin{aligned}\boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n - \varepsilon \nabla_{\boldsymbol{\theta}} \mathcal{L}(D;\theta) \\ \boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n - \varepsilon \frac{1}{N} \sum_{(\boldsymbol{x},y)\in D} \nabla_{\boldsymbol{\theta}} (y - f(\mathbf{x};\theta))^2\end{aligned}$$

- Where ε is the learning rate.
- The key is the gradient, but how can we easily compute this?
- Well, high-school analysis gives us the answer: the chain rule.

Backprop: the basics (continued)

- In this formulation, $f(\mathbf{x}; \theta)$ is our deep neural network parameterized by θ.
- The MLP equation for one hidden layer is:

$$\hat{\mathbf{y}}(\mathbf{x}) = \sigma(\mathbf{W}_2^T \sigma(\mathbf{W}_1^T \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$$

- So, in this case $\theta = (\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2)$
- And σ is some non-linear activation function.
- Question: why is the non-linear activation function important?

Backprop: the basics (continued)

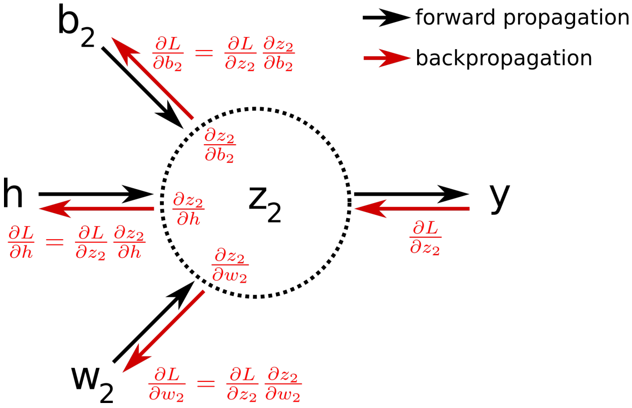
- We need to compute this:

$$
\begin{aligned}
&= \nabla_{\boldsymbol{\theta}}(y - f(\mathbf{x};\theta))^2 \\
&= \nabla_{\boldsymbol{\theta}}(y - \sigma(\mathbf{W}_2^T\sigma(\mathbf{W}_1^T\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2))^2 \\
&= -2(y - \sigma(\mathbf{W}_2^T\sigma(\mathbf{W}_1^T\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2))\nabla_{\boldsymbol{\theta}}\sigma(\mathbf{W}_2^T\sigma(\mathbf{W}_1^T\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)
\end{aligned}
$$

- Now, let's think about the partial derivatives that will make up this gradient computation…

Backprop: in pictures

- Here is a high-level overview of backprop.
- Essence: to compute the gradient wrt a parameter we need the forward activation AND the backpropagated gradient.
- It's really just the Chain Rule:



$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial z_2} \frac{\partial z_2}{\partial b_2}$

$\frac{\partial z_2}{\partial b_2}$

$\frac{\partial z_2}{\partial h}$

$\frac{\partial L}{\partial z_2}$

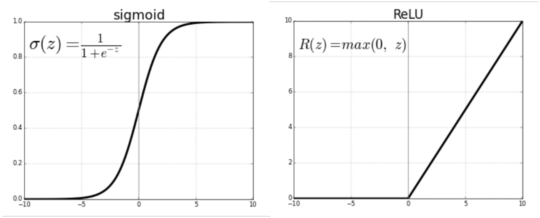
$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial z_2} \frac{\partial z_2}{\partial h}$

$\frac{\partial z_2}{\partial w_2}$

$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z_2} \frac{\partial z_2}{\partial w_2}$

Backprop: problems

- If the backpropagation algorithm is so "simple", why haven't we been using neural networks since the 1970s?
- There are a number of problems:
  - Saturating units: many activation functions are "flat" in their extremal values – this results in near zero gradients
  - Vanishing gradients: backprop creates a long chain of multiplied gradients – all of which are typically very small.
- Partial Solution: use non-saturating activation functions:



Backprop: problems (continued)

- Another problem is overparameterization: the (often very) many parameters in neural networks can lead to easy overfitting.
- Good exercise: count the number of weights in an MLP.
- Partial solution: use regularization to control the magnitude of weights in the network.

Backprop: Stochastic Gradient Descent (SGD)

- Problem: what happens if N (the number of training samples) is very large?
- Well, we end up taking very slow steps – each iteration of gradient descent is an average over the entire dataset.
- Solution: approximate the true gradient with the gradient at a single training example:

Online Stochastic Gradient Descent

- Choose an initial vector of parameters θ and learning rate η.
- Repeat until an approximate minimum is found:
  1. Randomly shuffle training samples in D.
  2. For $(\boldsymbol{x}, y) \in D$:
     - $\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\{\, \boldsymbol{x}, y \,\} ; \boldsymbol{\theta})$

Backprop: Stochastic Gradient Descent (continued)

- Another problem: evaluating the gradient on single examples leads to very noisy steps in parameter space.
- One trick to mitigate this is to use momentum: keep a running average of gradients that is slowly updated.
- Another solution is to use mini-batches: instead of a single sample, average the gradients over a small batch of samples.
- It is common to use a combination of mini-batches and momentum to stabilize training.

Backprop: ADAM

- Even with momentum and mini-batches, SGD can be slow to converge.
- One remaining problem is that the learning rate η is constant for all model parameters.
- Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network.
- That is, it adapts to the scale of each network parameter and to the sensitivity of the loss to each.

Diederik P. Kingma and Jimmy Lei Ba. Adam : A method for stochastic optimization. 2014. arXiv:1412.6980v9

Backprop: Terminology

- Some useful terminology for deep learning optimization:
  - 1 epoch: one complete pass over the data.
  - 1 iteration: a single gradient step.
  - N: number of training samples.
  - B: batch size.

Algorithm	iterations per epoch
Batch gradient descent	1
Stochastic Gradient Descent	N
Mini-batch Gradient Descent	$\frac{N}{B}$

Modular Construction

Tensorflow: graph-based, numerical meta-programming

- Tensorflow is a numerical programming framework originally created by Google.
- It is a comprehensive framework for working with machine learning in general – and Deep Learning in particular.
- It has flexible ecosystem of tools, libraries, and community resources.
- It provides official APIs for Python and C++.
- It also provides transparent access to Graphics Processing Unit (GPU) and Tensor Processing Unit (TPU).
- This means: write once, run pretty much anywhere.

Tensorflow: graph-based, numerical meta-programming

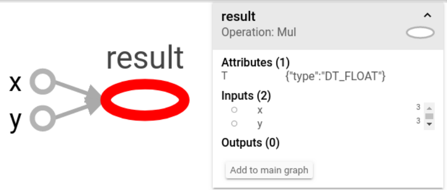
- One of Tensorflow's defining features is that it is a meta-programming environment for numerical programming.
- When you write an expression (e.g. `foo = a * b`) in Tensorflow, it does not execute it.
- Rather, it constructs a computation graph that represents the expressed computation.
- From this representation, we can do things like automatic differentiation.
- Good news: we never have to compute gradients by hand!
- Less good news: graph-based programming can be confusing.
- Let's take a very brief tour.

Tensorflow: `tf.Graph`

- Consider the following:

```
import tensorflow as tf

# Make a new graph.
graph = tf.Graph()
with graph.as_default():
    x = tf.constant([1.0, 2.0, 3.0], name='x')
    y = tf.constant([4.0, 5.0, 6.0], name='y')
    result = tf.multiply(x, y, name='result')
```



result
Operation: Mul

Attributes (1)
T {"type":"DT_FLOAT"}

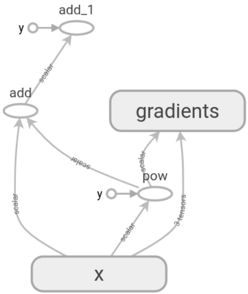
Inputs (2)
○ x 3
○ y 3

Outputs (0)

Tensorflow: `tf.Graph` (continued)

- Let's do something a bit more interesting:

```
graph2 = tf.Graph()
with graph2.as_default():
    x = tf.Variable(1.0, name='x')
    result = x**2 + x + 10
    dr_dx = tf.gradients([result], [x])
```



Tensorflow: `tf.Session`

- Well, that's fine and all... I guess.
- But, how do we actually make it do something?
- In order to execute a computation in Tensorflow, you must do so in a session:

```
graph2 = tf.Graph()
with graph2.as_default():
    x = tf.Variable(1.0, name='x')
    result = x**2 + x + 10
    dr_dx = tf.gradients([result], [x])

with tf.Session(graph=graph2) as sess:
    print(sess.run(result, feed_dict={x: 2.0}))
    print(sess.run(dr_dx, feed_dict={x: 2.0}))
```

```
16.0
[5.0]
```

Tensorflow: generality at a price

- The graph-based nature of Tensorflow is both a curse and a blessing.
- It is extremely powerful – these simple examples don't even scratch the surface.
- All operations, for example, can be defined in terms of arbitrary tensors.

```
import numpy as np
graph2 = tf.Graph()
with graph2.as_default():
    # Define an input variable.
    x = tf.Variable(np.random.rand(13, 1).astype('float32'), name='x')
    # Define our weight matrix and bias.
    W = tf.Variable(np.random.rand(13, 1).astype('float32'), name='W')
    b = tf.Variable(0.0, name='b')
    result = tf.matmul(tf.transpose(W), x) + b
    dr_dTheta = tf.gradients([result], [W, b])
```

Tensorflow: generality at a price

```
with tf.Session(graph=graph2) as sess:
    sess.run(tf.global_variables_initializer())
    sample = np.random.rand(13, 1)
    print(f'Output:\n{sess.run(result, feed_dict={x: sample})}')
    print(f'Gradient:\n{sess.run(dr_dTheta, feed_dict={x: sample})}')
```

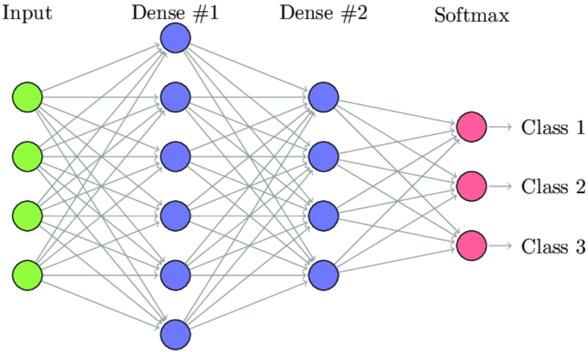
```
Output:
[[3.8426936]]
Gradient:
[array([[0.9956546 ], [0.5471455 ], [0.55688864],
       [0.4037869 ],  [0.5371017 ], [0.51199615],
       [0.22210105],  [0.98923653], [0.8349015 ],
       [0.11137984],  [0.96884817], [0.67522067],
       [0.1807481 ]], dtype=float32), 1.0]
```

Tensorflow: the Good News

- The good news is that we don't have to program at such a low level all the time.
- There are several high-level frameworks built on top of Tensorflow.
- These frameworks hide the graph-based, meta-programming complexity of the underlying library.
- One such framework is Keras, which is specifically designed to support high-level programming for Deep Learning.
- It effectively encapsulates models in a way that makes is easy (well, easier) to define, train, execute, and test.
- We usually write Keras/Tensorflow to indicate that we are using Keras with the Tensorflow backlend.

Keras: layer-wise composition

- This is another view of a Multi-layer Perceptron (MLP) for classification:



- Let's see how to build a model like this in Keras.

Keras: a catalog of layer types

`tf.keras.layers.Dense`

- Just your regular densely-connected NN layer:

  `output = activation(dot(input, kernel) + bias)` where `activation` is the element-wise activation function passed as the `activation` argument, `kernel` is a weights matrix created by the layer, and `bias` is a bias vector created by the layer (only applicable if `use_bias` is `True`).

- We only need to specify the number of outputs and (if it's the first layer) number of inputs:

```
fc1 = tf.keras.Dense(6, input_shape=(4,))
fc2 = tf.keras.Dense(4)
fc3 = tf.keras.Dense(3)
```

Keras: a catalog of layer types (continued)

`keras.layers.Activation`

- Apply a function elementwise to its input:

  Applies an activation function to an output.

  Arguments:
  `activation`: Activation function, such as `tf.nn.relu`, or string name of built-in activation function, such as "relu" or "softmax"

- Let's use it to create our output layer:

```
output = tf.keras.Activation('softmax')
```

Keras: the `tf.keras.Sequential` model type

- But wait… How does `fc2` know what it's input should be? Or even what it's input size should be?
- Well, `fc1` at least knows it's input size (if not its input tensor).
- The answer is doesn't until we compose them together into a model.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Dense, Activation

# Create a Sequential model, add layers in sequence.
model = tf.keras.Sequential()
model.add(Dense(6, input_shape=(4,)))
model.add(Dense(4))
model.add(Dense(3))
model.add(Activation('softmax'))

model.predict(np.array([[1, 2, 3, 4]]))

-> array([[0.00421561, 0.9916352 , 0.00414928]], dtype=float32)
```

Keras: a regression model

- Let's change our model a little first:

```
from tensorflow.keras import models, layers

# Define our first model: a simple Ordinary Linear Regression
model = models.Sequential()
model.add(layers.Dense(1, activation='linear', input_shape=(13,)))
```

- What does this remind you of?

Keras: compiling the model

- We have a randomly initialized Ordinary Linear Regression model.
- Now we have to fit the model; first we compile it, specifying the loss and optimizer.
- In Keras, *compiling* refers to preparing the model for optimization: computing the gradient wrt the loss, and adding any graph nodes needed by the optimizer.

```
# Compile the model, specifying optimizer and loss.
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
```

`model.compile()`

To the documentation!

Keras: fitting the model

- Whew, that's a lot of steps. . .
- Now, given some training data, we can fit the model:

```
# Fit the model parameters.
history = model.fit(X_train, y_train, validation_split=0.2, epochs=100)
```

`model.fit()`

Back to the docs!

Keras: interpreting console spam

- Keras model fitting generates a ton of console spam:

```
Train on 323 samples, validate on 81 samples
Epoch 1/2000
323/323 [=====] - 0s 248us/sample - loss: 33611.6093 - mean_absolute_error: 175.2687 - val_loss: 32327.7965 - val_mean_absolute_error: 174.9598

Epoch 2/2000
323/323 [=====] - 0s 62us/sample - loss: 30264.9680 - mean_absolute_error: 165.8515 - val_loss: 29066.1841 - val_mean_absolute_error: 165.2556
...
```

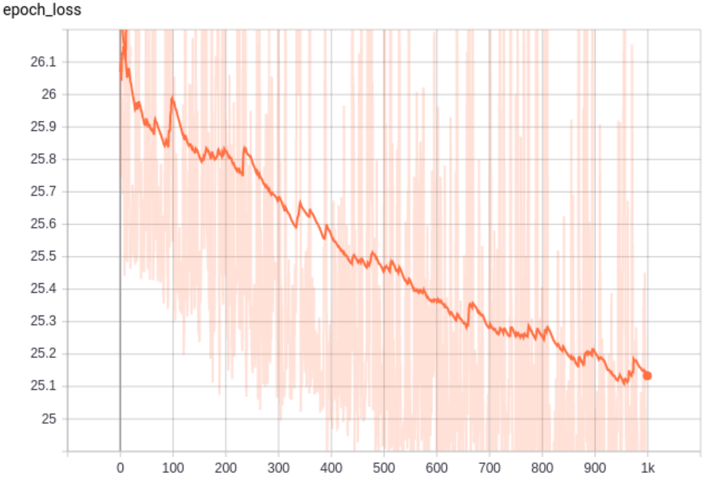
Keras: Tensorboard

- The `history` object returned from `model.fit()` contains a ton of information about the training process.
- However, a much better way to monitor training is to use Tensorboard.
- We setup a log directory, a Tensorboard callback, and tell Keras to call it while fitting:

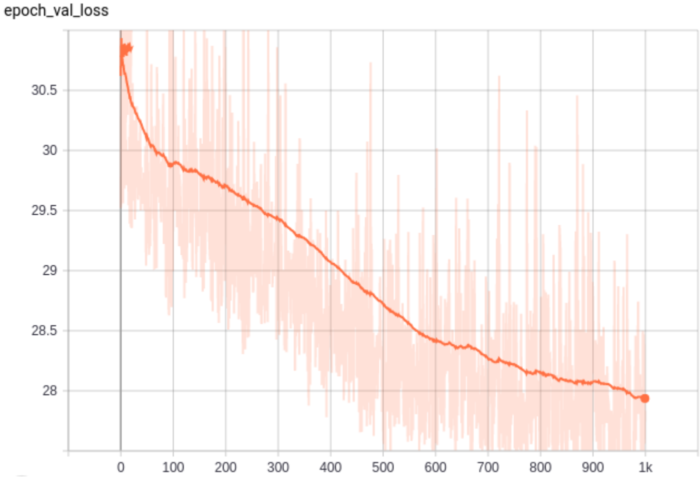
```
logdir = 'logs/'
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)
history = model.fit(X_train, y_train, validation_split=0.2,
                    epochs=1000, callbacks=[tensorboard_callback])

# Some magic to make tensorboard work in Jupyter.
%load_ext tensorboard
%tensorboard -logdir logs
```

Keras: Tensorboard (continued)



Keras: Tensorboard (continued)



Keras: evaluating the final model

- Keras models also have a built-in `evaluate` method.
- With this we can run the trained model on a test set to obtain the loss on the test set as well as any registered metrics.

```python
from tensorflow.keras import models, layers

# Define our first model: a simple Ordinary Linear Regression
model = models.Sequential()
model.add(layers.Dense(1, activation='linear', input_shape=[X_train.shape[

# Compile the model, specifying optimizer and loss.
model.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Fit the model parameters.
history = model.fit(X_train, y_train, validation_split=0.2, epochs=2000)
model.evaluate(X_test, y_test)

-> [31.26155943029067, 4.102251]
```

Reflections

Deep Learning

- Deep models like Multilayer Perceptrons (MLPs) are extremely flexible function approximators.
- They can be trained to approximate optimal functions by minimizing a loss over a set of training samples.
- Their composable nature is what makes them deep – you can keep increasing the power of your approximation by adding layers or by increasing the width of layers.
- Their power is also their weakness: they can be hard to optimize and they can easily overfit even large training sets
- Nonetheless, with a little bit of (good) practice they can also be very effective in the real world.

Keras/Tensorflow

- Numerical frameworks like Keras make life MUCH easier when working with Deep Models.
- Their ability to automatically differentiate frees us from the need to manually computer gradients for optimization.
- They reflect our intuition about models: their APIs are more or less direct mappings from our modular diagrams of deep modules.
- They also facilitate transparent use of GPU/TPU resources, when available.
- The exercises we will see today do not benefit hugely from GPUs, but tomorrow when we look a Convolution Neural Networks, this will all change.

First Steps with Keras Lab

- The laboratory notebook for today:

  `http://bit.ly/DTwin-ML6`