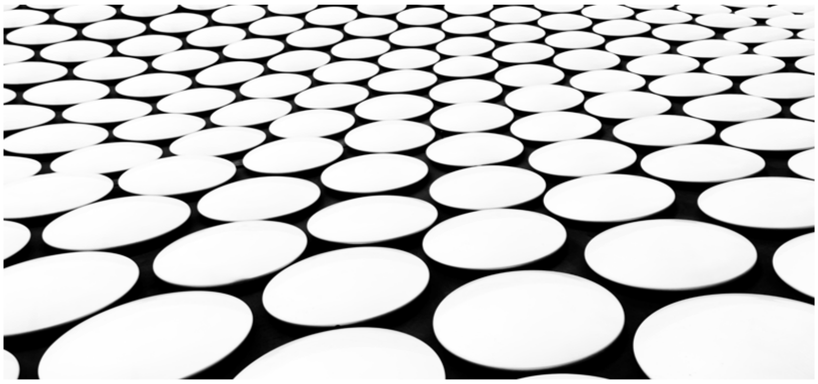


SCHOOL OF AI
ITALIA

MACHINE LEARNING AVANZATO DA ZERO

ANTONIO DI CECCO - SCHOOL OF AI

Reti neurali introduzione

History

- Nearly everything we talk about today existed ~1990
- What changed?

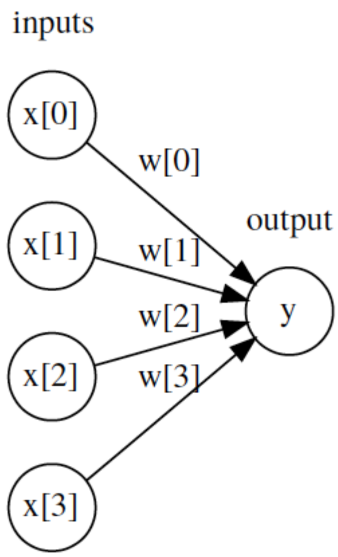
  – More data

  – Faster computers (GPUs TPUs)

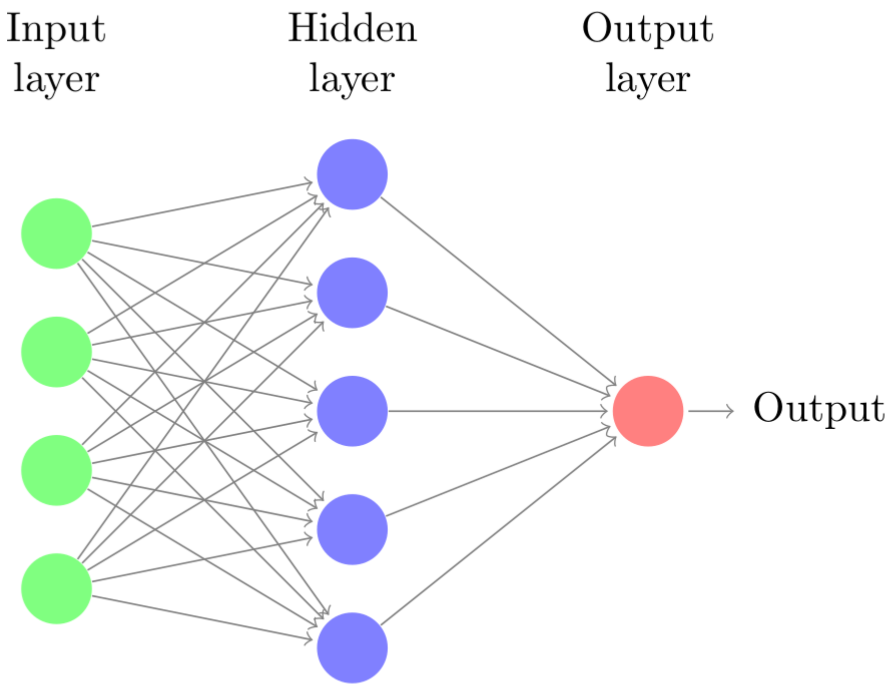
  – Some improvements:

    - relu
    - Drop-out
    - adam
    - batch-normalization
    - residual networks

Logistic regression as neural net



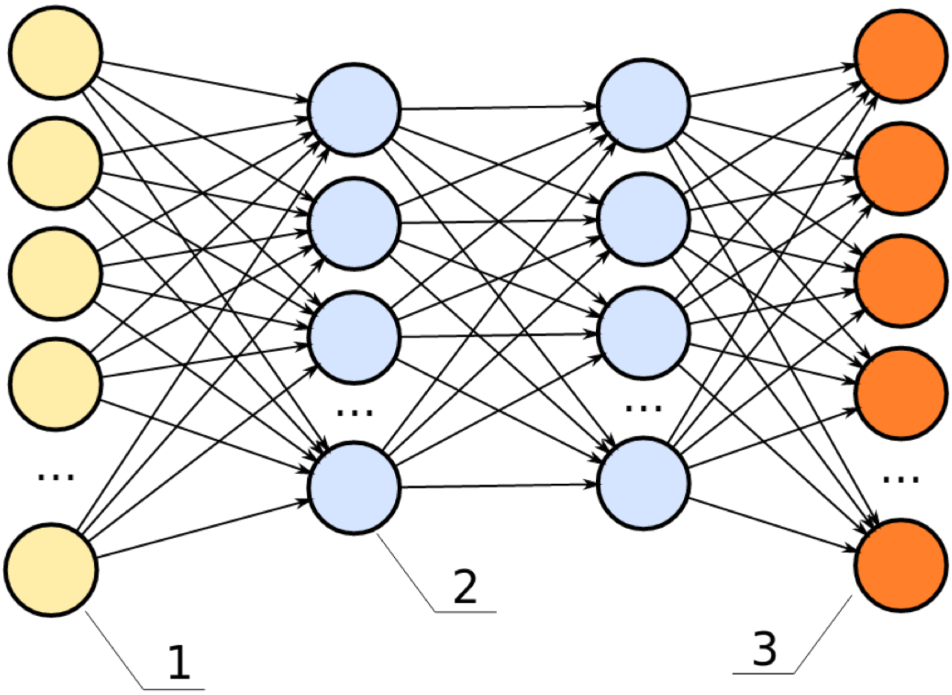
Basic Architecture



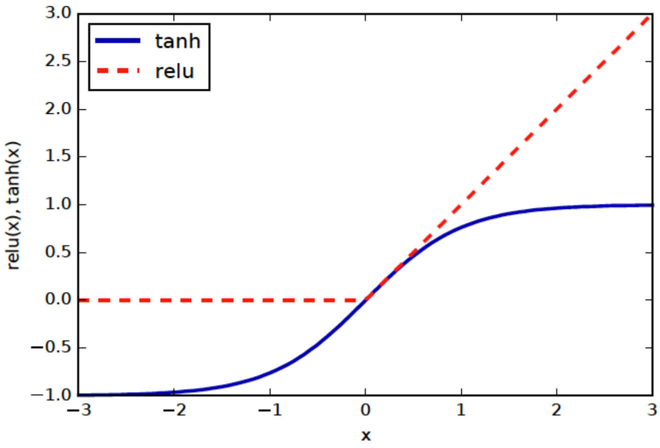

```
h(x)=f(W1 * x+b1)
```

```
o(x)=g(W2 * h(x)+b2)
```

More layers



Nonlinear activation function



Supervised Neural Networks

- Non-linear models for classification and regression
- Work well for very large datasets
- Non-convex optimization
- Notoriously slow to train – need for GPUs
- Use dot products etc require preprocessing, → similar to SVM or linear models, unlike trees
- MANY variants (Convolutional nets, Gated Recurrent neural networks, Long-Short-Term Memory, recursive neural networks, variational autoencoders, generative adverserial networks, deep reinforcement learning, ...)

Training Objective

$$h(x) = f(W_1 x + b_1)$$

$$o(x) = g(W_2 h(x) + b_2) = g(W_2 f(W_1 x + b_1) + b_2)$$

$$\min_{W_1, W_2, b_1, b_2} \sum_{i=1}^{N} l(y_i, o(x_i))$$

$$= \min_{W_1, W_2, b_1, b_2} \sum_{i=1}^{N} l(y_i, g(W_2 f(W_1 x + b_1) + b_2)$$

- Squared loss for regression.
- Cross-entropy loss for classification

Backpropagation

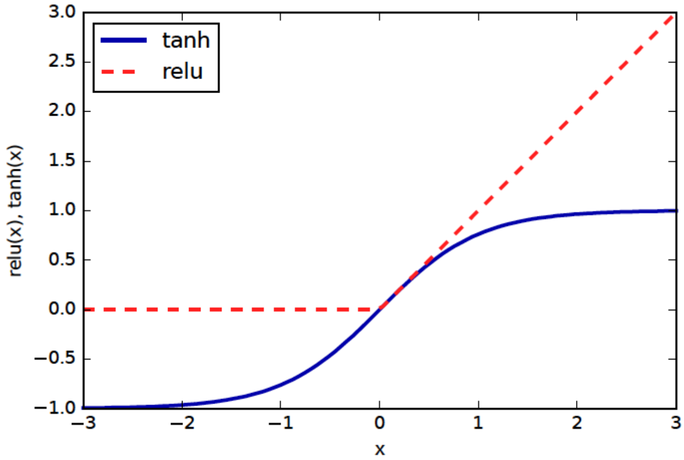
- Need $\frac{\partial l(y,o)}{\partial W_i}$ and $\frac{\partial l(y,o)}{\partial b_i}$

$$\text{net}(x) := W_1 x + b_1$$

$$\frac{\partial o(\mathbf{x})}{\partial W_1} = \underbrace{\frac{\partial o(\mathbf{x})}{\partial h(\mathbf{x})}}_{\text{backpropagation of gradient of layer above.}} \underbrace{\frac{\partial h(\mathbf{x})}{\partial \text{net}(\mathbf{x})}}_{\text{Gradient of Non-linearity f}} \underbrace{\frac{\partial \text{net}(\mathbf{x})}{\partial W_1}}_{\text{Input to 1st layer x}}$$

- la backpropagation non è un algoritmo di ottimizzazione ma un modo di calcolare il gradiente!

MA!



- subgradients
- differenziabilità numerica

Optimizing W, b

Batch

$$W_i \leftarrow W_i - \eta \sum_{j=1}^{N} \frac{\partial l(x_j, y_j)}{\partial W_i}$$

Online/Stochastic

$$W_i \leftarrow W_i - \eta \frac{\partial l(x_j, y_j)}{\partial W_i}$$

Minibatch

$$W_i \leftarrow W_i - \eta \sum_{j=k}^{k+m} \frac{\partial l(x_j, y_j)}{\partial W_i}$$

Learning Heuristics

- Constant `η` not good

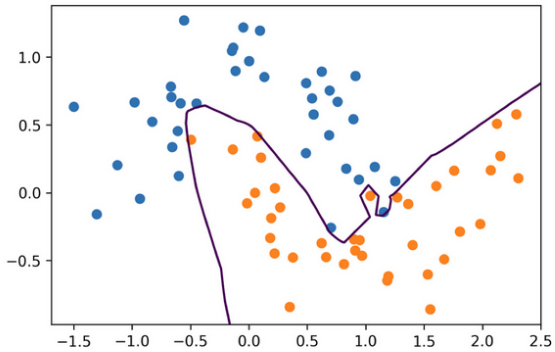
- Can decrease `η`
- Better: adaptive `η` for each entry if W_i
- State-of-the-art: adam (with magic numbers)
- https://arxiv.org/pdf/1412.6980.pdf
- http://sebastianruder.com/optimizing-gradient-descent/

Picking Optimization Algorithms

- Small dataset: off the shelf like l-bfgs
- Big dataset: adam / rmsprop
- Have time & nerve: tune the schedule

Neural Network in practica

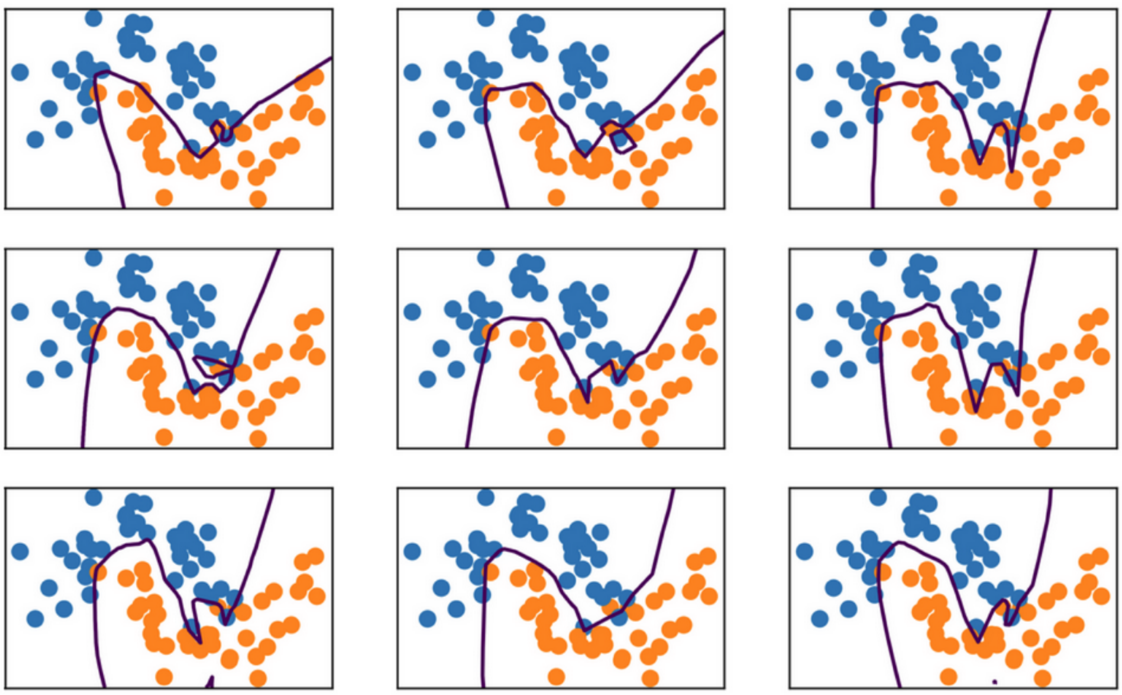
Neural Nets with sklearn




```
mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(X_train, y_train)
print(mlp.score(X_train, y_train))
print(mlp.score(X_test, y_test))
```

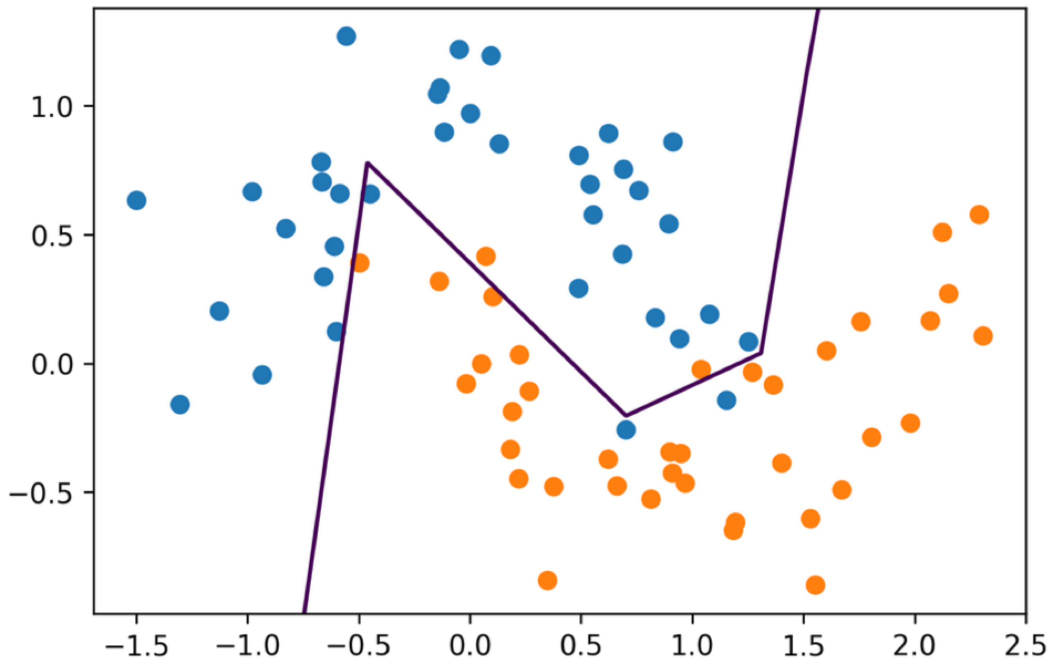
```
1.0
0.88
```

Random State

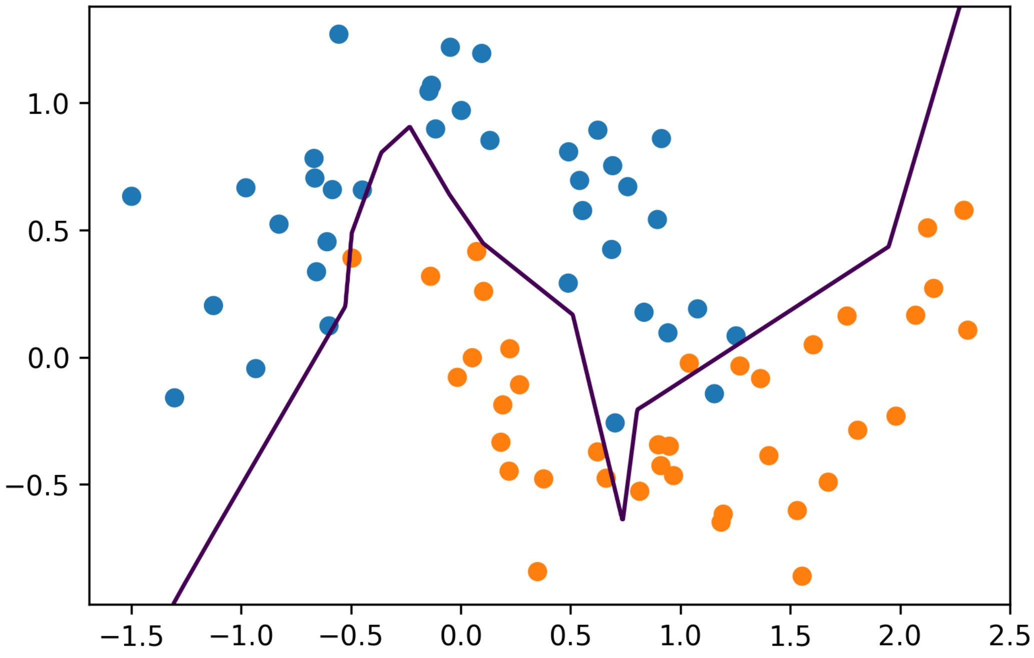


Hidden Layer Size

```
mlp = MLPClassifier(solver='lbfgs', hidden_layer_size=(5,), random_state=10)
mlp.fit(X_train, y_train)
print(mlp.score(X_train, y_train))
print(mlp.score(X_test, y_test))
0.93
0.82
```

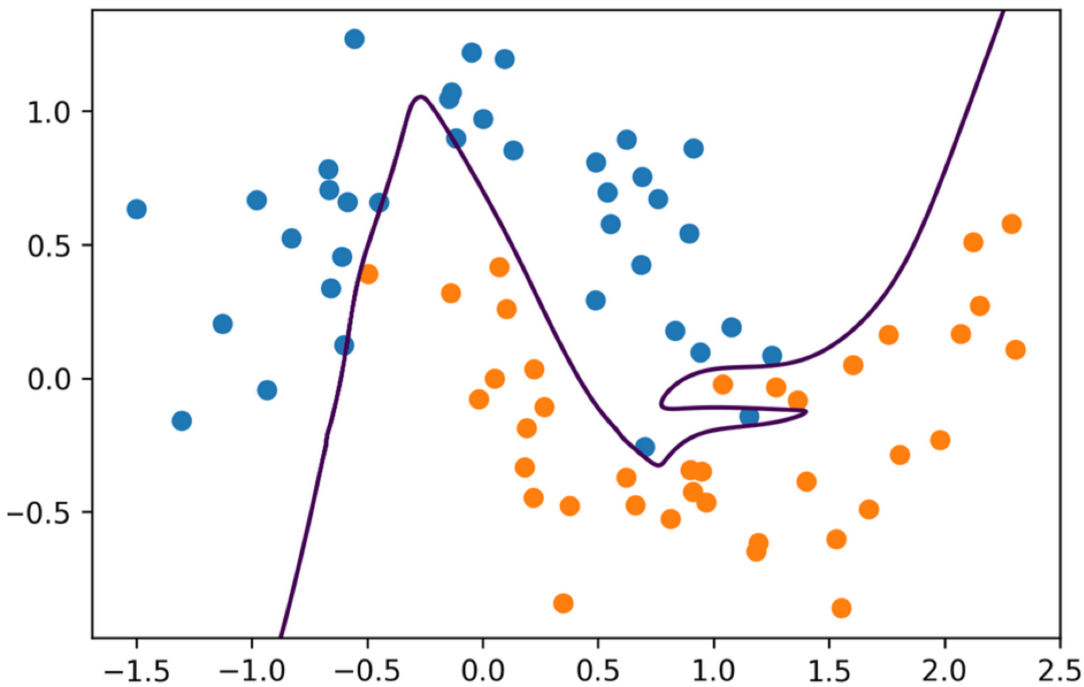



```
mlp = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(10, 10, 10),
                    random_state=0)
mlp.fit(X_train, y_train)
print(mlp.score(X_train, y_train))
print(mlp.score(X_test, y_test))
0.97
0.84
```

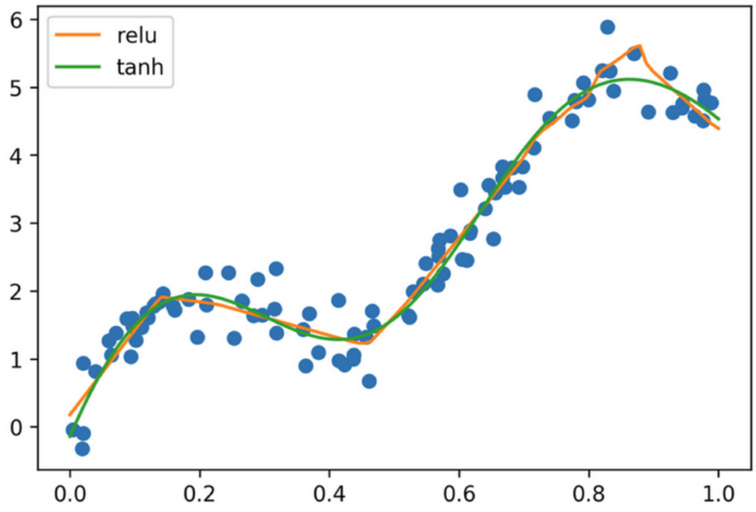



```
mlp = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(10, 10, 10),
                    activation='tanh', random_state=0)
mlp.fit(X_train, y_train)
print(mlp.score(X_train, y_train))
print(mlp.score(X_test, y_test))
```

```
1.0
0.92
```



Regression




```python
from sklearn.neural_network import MLPRegressor
mlp_relu = MLPRegressor(solver="lbfgs").fit(X, y)
mlp_tanh = MLPRegressor(solver="lbfgs", activation='tanh').fit(X, y)
```

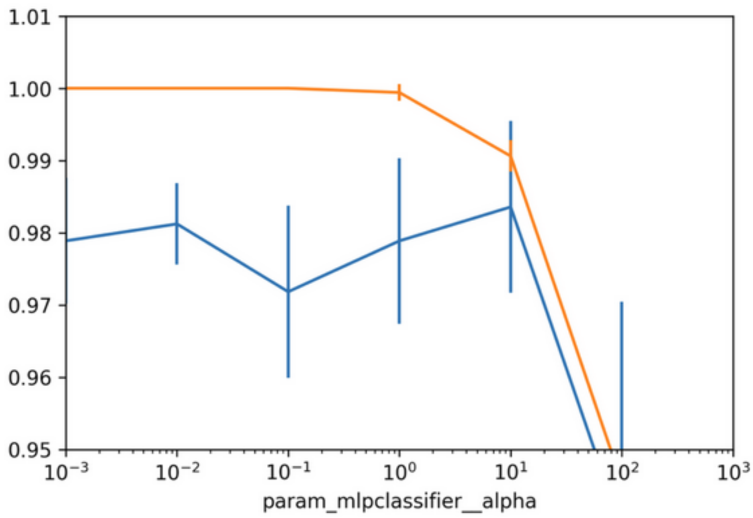
Complexity Control

- Number of parameters
- Regularization
- Early Stopping
- drop-out

Grid-Searching Neural Nets

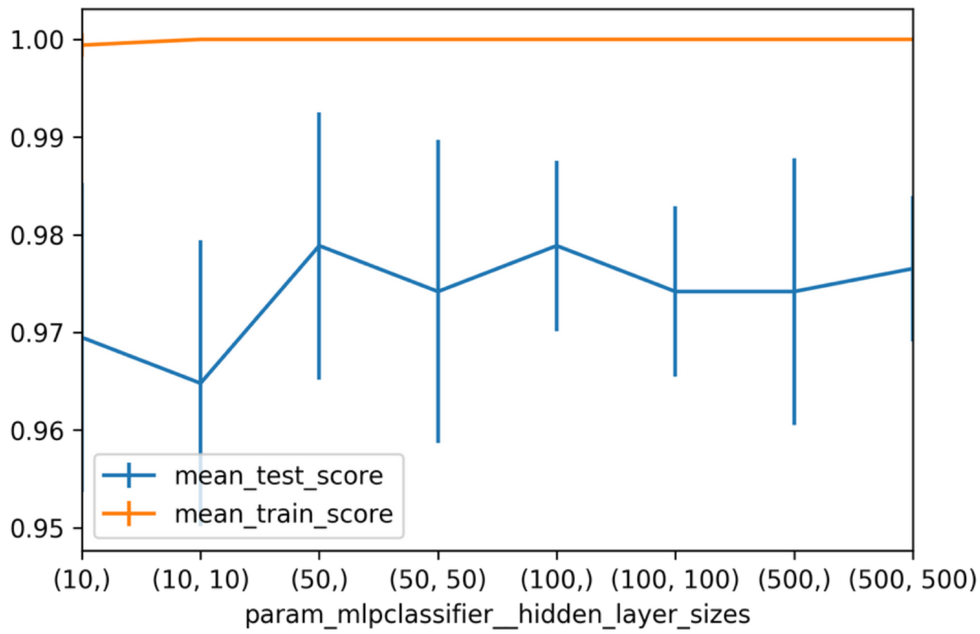
```
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    data.data, data.target, stratify=data.target, random_state=0)

from sklearn.model_selection import GridSearchCV
pipe = make_pipeline(StandardScaler(), MLPClassifier(solver="lbfgs",
                     random_state=1))
param_grid = {'mlpclassifier__alpha': np.logspace(-3, 3, 7)}
grid = GridSearchCV(pipe, param_grid)
results = pd.DataFrame(grid.cv_results_)
res = results.pivot_table(index="param_mlpclassifier__alpha",
                          values=["mean_test_score", "mean_train_score"])
res
```

	mean_test_score	mean_train_score
param_mlpclassifier__alpha		
0.001	0.978873	1.000000
0.010	0.981221	1.000000
0.100	0.971831	1.000000
1.000	0.978873	0.999412
10.000	0.983568	0.990612
100.000	0.938967	0.945427
1000.000	0.626761	0.626761



Searching hidden layer sizes

```
from sklearn.model_selection import GridSearchCV
pipe = make_pipeline(StandardScaler(), MLPClassifier(solver="lbfgs"
                     ,random_state=1))
param_grid = {'mlpclassifier__hidden_layer_sizes':
              [(10,), (50,), (100,), (500,), (10, 10), (50, 50), (100, 100),
(500,
              500)]
             }
grid = GridSearchCV(pipe, param_grid)
grid.fit(X_train, y_train)
```



Getting Flexible and Scaling Up

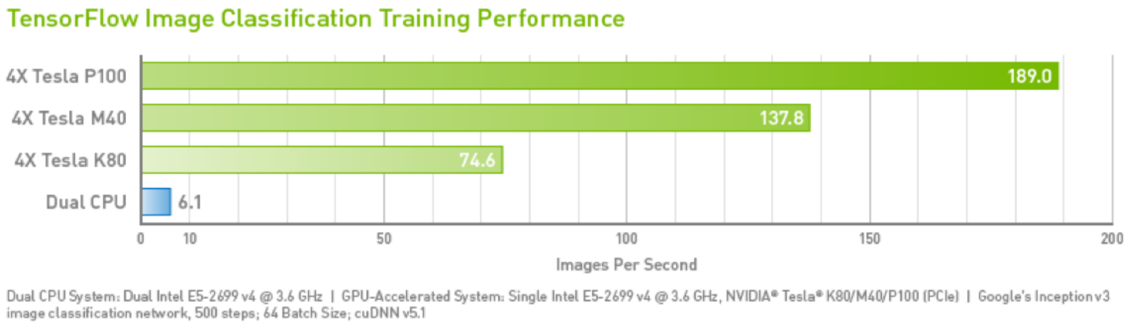
Write your own neural networks

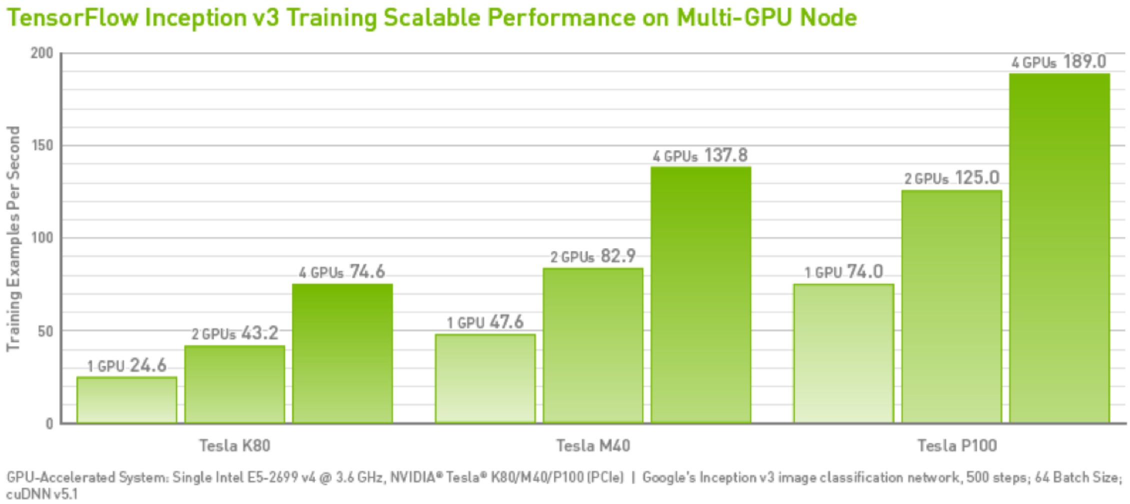
```
class NeuralNetwork(object):
    def __init__(self):
        # initialize coefficients and biases
        pass
    def forward(self, x):
        activation = x
        for coef, bias in zip(self.coef_, self.bias_):
            activation = self.nonlinearity(np.dot(activation, coef) + bias)
        return activation
    def backward(self, x):
        # compute gradient of stuff in forward pass
        pass
```

Autodiff

```
# http://mxnet.io/architecture/program_model.html
class array(object) :
    """Simple Array object that support autodiff."""
    def __init__(self, value, name=None):
        self.value = value
        if name:
            self.grad = lambda g : {name : g}
    def __add__(self, other):
        assert isinstance(other, int)
        ret = array(self.value + other)
        ret.grad = lambda g : self.grad(g)
        return ret
    def __mul__(self, other):
        assert isinstance(other, array)
        ret = array(self.value * other.value)
        def grad(g):
            x = self.grad(g * other.value)
            x.update(other.grad(g * self.value))
```

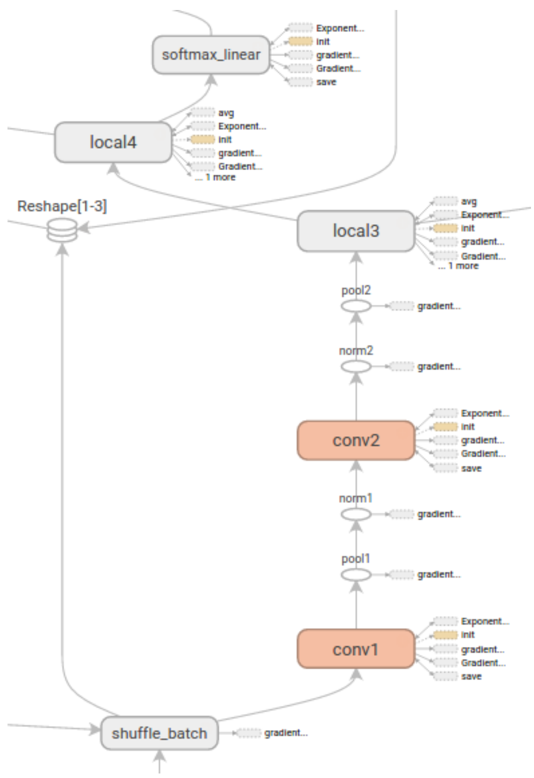
```
a = array(np.array([1, 2]), 'a')
b = array(np.array([3, 4]), 'b')
c = b * a
d = c + 1
print(d.value)
print(d.grad(1))
[4 9]
{'b': array([1, 2]), 'a': array([3, 4])}
```





https://developer.nvidia.com/deep-learning-performance-training-inference

---



All I want from a deep learning framework

- Autodiff
- GPU support
- Optimization and inspection of computation graph
- on-the-fly generation of the graph (?)
- distribution over muliple GPUs and/or cluster (?)
- Choices (right now):
  - Skorch
  - TensorFlow
  - PyTorch / Torch
  - (Theano)

Deep Learning Libraries

- Keras (Tensorflow, CNTK, Theano)
- PyTorch (torch)
- MXNet (MXNet)
- Also see: http://mxnet.io/architecture/program_model.html

Quick look at TensorFlow

programmazione imperativa

- “down to the metal” - don’t use for everyday tasks
- Three steps for learning (originally):
    - Build the computation graph (using array operations and functions etc)
    - Create an Optimizer (gradient descent, adam, ...) attached to the graph.
    - Run the actual computation.
- Eager mode (default in Tensorflow 2.0):
    - Write imperative code directly

```python
import tensorflow as tf
import numpy as np

# Create 100 phony x, y data points in NumPy, y = x * 0.1 + 0.3
x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3

# create graph: model
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# create graph: loss
loss = tf.reduce_mean(tf.square(y - y_data))

# bind optimizer
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

# run graph
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# Fit the line.
for step in range(201):
    sess.run(train)
    if step % 20 == 0:
        print(step, sess.run(W), sess.run(b))
```

No computation

Allocate variables

All the work / computation

https://www.tensorflow.org/versions/r0.10/get_started/

PyTorch example

```
dtype = torch.float
device = torch.device("cpu")
# device = torch.device("cuda:0") # Uncomment this to run on GPU

N = 100

# Create random input and output data
x = torch.randn(N, 1, device=device, dtype=dtype)
y = torch.randn(N, 1, device=device, dtype=dtype)

# Randomly initialize weights
w = torch.randn(D_in, H, device=device, dtype=dtype)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    y_pred = x.mm(w1)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum().item()
    if t % 100 == 99:
        print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    loss.backward()

    # Update weights using gradient descent
    w1 -= learning_rate * w1.grad
    w1.grad.zero_()
```

Don't go down to the metal unless you have to!

Don't write TensorFlow, write Keras!

Don't write PyTorch, write **pytorch.nn** or **FastAI** (or **Skorch** or ignite)